Manuel de suivi dynamique Solaris

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Référence : 819–6958–10
Septembre 2008

Copyright 2008 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle de la technologie utilisée par le produit décrit dans le présent document. En particulier, et sans limitation, ces droits de propriété intellectuelle peuvent inclure des brevets américains ou dépôts de brevets en cours d'homologation aux États-Unis et dans d'autres pays.

Droits du gouvernement américain – logiciel commercial. Les utilisateurs gouvernementaux sont soumis au contrat de licence standard de Sun Microsystems, Inc. et aux dispositions du Federal Acquisition Regulation (FAR, règlements des marchés publics fédéraux) et de leurs suppléments.

Cette distribution peut contenir des éléments développés par des tiers.

Des parties du produit peuvent être dérivées de systèmes Berkeley-BSD, sous licence de l'Université de Californie. UNIX est une marque déposée aux États-Unis et dans d'autres pays, sous licence exclusive de X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, le logo Solaris, le logo Java (tasse de café), docs.sun.com, Java, StarOfficeJava et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux États-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux États-Unis et dans d'autres pays. Les produits portant les marques SPARC sont constitués selon une architecture développée par Sun Microsystems, Inc.

L'interface utilisateur graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et détenteurs de licence. Sun reconnaît le travail précurseur de Xerox en matière de recherche et de développement du concept d'interfaces utilisateur visuelles ou graphiques pour le secteur de l'informatique. Sun détient une licence Xerox non exclusive sur l'interface utilisateur graphique Xerox. Cette licence englobe également les détenteurs de licences Sun qui implémentent l'interface utilisateur graphique OPEN LOOK et qui, en outre, se conforment aux accords de licence écrits de Sun.

Les produits cités dans la présente publication et les informations qu'elle contient sont soumis à la législation américaine relative au contrôle sur les exportations et, le cas échéant, aux lois sur les importations ou exportations dans d'autres pays. Il est strictement interdit d'employer ce produit conjointement à des missiles ou armes biologiques, chimiques, nucléaires ou de marine nucléaire, directement ou indirectement. Il est strictement interdit d'effectuer des exportations et réexportations vers des pays soumis à l'embargo américain ou vers des entités identifiées sur les listes noires des exportations américaines, notamment les individus non autorisés et les listes nationales désignées.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, REPRÉSENTATIONS ET GARANTIES EXPRESSES OU TACITES, Y COMPRIS TOUTE GARANTIE IMPLICITE RELATIVE À LA COMMERCIALISATION, L'ADÉQUATION À UN USAGE PARTICULIER OU LA NON-VIOLATION DE DROIT, SONT FORMELLEMENT EXCLUES. CETTE EXCLUSION DE GARANTIE NE S'APPLIQUERAIT PAS DANS LA MESURE OÙ ELLE SERAIT TENUE JURIDIQUEMENT NULLE ET NON AVENUE.

081121@21288

Table des matières

Préface 21

1 **Introduction** 27
Démarrage 27
Fournisseurs et sondes 30
Compilation et instrumentation 32
Variables et expressions arithmétiques 34
Prédicats 37
Format de sortie 41
Tableaux 45
Types et symboles externes 47

2 **Types, opérateurs et expressions** 49
Noms d'identifiant et mots de passe 49
Types et tailles des données 50
Constantes 52
Opérateurs arithmétiques 54
Opérateurs relationnels 55
Opérateurs logiques 56
Opérateurs de bit 56
Opérateurs d'assignation 57
Opérateurs d'incrément et de décrément 58
Expressions conditionnelles 59
Conversions de types 60
Priorité 61

3 Variables 63

Variables scalaires 63

Tableaux associatifs 64

Variables locales de thread 66

Variables locales de clause 69

Variables intégrées 71

Variables externes 74

4 Structure de programme D 77

Clauses et déclarations de sondes 77

Descriptions de sonde 78

Prédicats 80

Actions 80

Utilisation du préprocesseur C 80

5 Pointeurs et ensembles 81

Pointeurs et adresses 81

Sécurité du pointeur 82

Déclarations et stockage d'ensembles 84

Relation entre pointeur et ensemble 85

Arithmétique de pointeur 86

Pointeurs génériques 87

Ensembles multidimensionnels 88

Pointeurs sur des objets DTrace 88

Pointeurs et espaces d'adresse 89

6 Chaînes de caractères 91

Représentation de chaînes 91

Constantes de chaîne 92

Assignation de chaîne 92

Conversion de chaîne 93

Comparaison de chaînes 93

7 Structs et Unions ... 95
Structs ... 95
Pointeurs vers structs ... 98
Unions ... 101
Tailles des membres et décalages ... 105
Champs de bit ... 106

8 Définitions des types et des constantes ... 109
Typedef ... 109
Énumérations ... 110
Inlines ... 111
Espaces de noms de types ... 112

9 Groupements ... 115
Fonctions de groupement ... 115
Groupements ... 117
Impression de groupements ... 124
Normalisation des données ... 125
Effacement de groupements ... 129
Troncature de groupements ... 129
Réduction des abandons ... 131

10 Actions et sous-routines ... 133
Actions ... 133
Action par défaut ... 133
Actions d'enregistrement de données ... 134
`trace()` ... 135
`tracemem()` ... 135
`printf()` ... 135
`printa()` ... 135
`stack()` ... 136
`ustack()` ... 137
`jstack()` ... 142
Actions destructrices ... 142

Actions destructrices de processus 142
Actions destructrices de noyau 145
Actions spéciales 148
Actions spéculatives 148
exit() 148
Sous-routines 149
alloca() 149
basename() 149
bcopy() 149
cleanpath() 150
copyin() 150
copyinstr() 150
copyinto() 151
dirname() 151
msgdsize() 151
msgsize() 151
mutex_owned() 152
mutex_owner() 152
mutex_type_adaptive() 152
progenyof() 152
rand() 152
rw_iswriter() 153
rw_write_held() 153
speculation() 153
strjoin() 153
strlen() 154

11 Tampons et mise en tampon 155
Tampons principaux 155
Stratégies de tampon principal 155
Stratégie switch 156
Stratégie fill 157
Stratégie ring 157
Autres tampons 158
Taille des tampons 159

Stratégie de redimensionnement du tampon 159

12 Format de sortie 161

`printf()` 161

Spécifications de conversion 162

Spécificateurs d'indicateurs 163

Spécificateurs de largeur et de précision 163

Préfixes de taille 164

Formats de conversion 165

`printa()` 167

Format par défaut `trace()` 169

13 Suivi spéculatif 171

Interfaces de spéculation 172

Création d'une spéculation 172

Utilisation d'une spéculation 172

Validation d'une spéculation 173

Annulation d'une spéculation 174

Exemple de spéculation 174

Options de spéculation et réglage 179

14 Utilitaire `dtrace`(1M) 181

Description 181

Options 182

Opérandes 188

État de sortie 188

15 Scripts 189

Fichiers interpréteurs 189

Variables de macro 191

Arguments de macro 192

ID processus cible 194

16 Options et paramètres réglables ... 197
Options client ... 197
Options modificatrices ... 199

17 Fournisseur `dtrace` ... 201
Sonde `BEGIN` ... 201
La sonde `END` ... 202
Sonde `ERROR` ... 203
Stabilité ... 204

18 Fournisseur `lockstat` ... 207
Présentation ... 207
Sondes de verrou adaptatif ... 208
Sondes de verrou de rotation ... 208
Verrous de thread ... 210
Sondes de verrouillage en lecture/écriture ... 210
Stabilité ... 211

19 Fournisseur `profile` ... 213
Sondes `profile-`*n* ... 213
Sondes `tick-`*n* ... 216
Arguments ... 216
Résolution de l'horloge ... 217
Création de sonde ... 218
Stabilité ... 219

20 Fournisseur `fbt` ... 221
Sondes ... 221
Arguments des sondes ... 222
Sondes `entry` ... 222
Sondes `return` ... 222
Exemples ... 222
Optimisation des appels terminaux ... 228
Fonctions d'assemblage ... 230

Limitations du jeu d'instructions ... 230
Limitations x86 ... 230
Limitations SPARC ... 230
Interaction du point d'arrêt ... 231
Chargement des modules ... 231
Stabilité ... 231

21 Fournisseur `syscall` ... 233
Sondes ... 233
Anachronismes des appels système ... 233
Appels système sous-codés ... 234
Appels système de grands fichiers ... 234
Appels système privés ... 235
Arguments ... 235
Stabilité ... 235

22 Fournisseur `sdt` ... 237
Sondes ... 237
Exemples ... 238
Création de sondes SDT ... 242
Déclaration des sondes ... 242
Arguments des sondes ... 243
Stabilité ... 243

23 Fournisseur `sysinfo` ... 245
Sondes ... 245
Arguments ... 248
Exemple ... 250
Stabilité ... 252

24 Fournisseur `vminfo` ... 253
Sondes ... 253
Arguments ... 256
Exemple ... 256

Stabilité 260

25 Fournisseur proc 261
Sondes 261
Arguments 263
lwpsinfo_t 264
psinfo_t 267
Exemples 268
exec 268
start et exit 269
lwp-start et lwp-exit 271
signal-send 273
Stabilité 274

26 Fournisseur sched 275
Sondes 275
Arguments 278
cpuinfo_t 278
Exemples 279
on-cpu et off-cpu 279
enqueue et dequeue 287
sleep et wakeup 293
preempt, remain-cpu 302
change-pri 304
tick 306
Stabilité 308

27 Fournisseur io 309
Sondes 309
Arguments 310
Structure bufinfo_t 311
devinfo_t 312
fileinfo_t 313
Exemples 314

Stabilité 326

28 **Fournisseur `mib`** 329
Sondes 329
Arguments 345
Stabilité 345

29 **Fournisseur `fpuinfo`** 347
Sondes 347
Arguments 349
Stabilité 349

30 **Fournisseur `pid`** 351
Attribution d'un nom à des sondes `pid` 351
Sondes de limite de fonction 353
  Sondes `entry` 353
  Sondes `return` 353
Sondes de décalage de fonction 353
Stabilité 354

31 **Fournisseur `plockstat`** 355
Présentation 355
Sondes Mutex 356
Sondes de verrouillage en lecture/écriture 357
Stabilité 357

32 **Fournisseur `fasttrap`** 359
Sondes 359
Stabilité 359

33 **Suivi des processus utilisateur** 361
Sous-routines `copyin()` et `copyinstr()` 361
  Évitement des erreurs 362

Suppression de l'interférence de dtrace(1M) 363
Fournisseur syscall 363
Action de la fonction ustack() 365
Tableau uregs[] 366
Fournisseur pid 369
Suivi de la limite de fonction utilisateur 369
Suivi des instructions arbitraires 371

34 Suivi défini statiquement pour les applications utilisateur 375
Choix des points de sonde 375
Ajout de sondes à une application 376
Définition des fournisseurs et des sondes 376
Ajout de sondes à un code d'application 377
Création d'applications avec des sondes 378

35 Sécurité 379
Privilèges 379
Utilisation privilégiée de DTrace 380
Privilège dtrace_proc 380
Privilège dtrace_user 381
Privilège dtrace_kernel 382
Privilèges de superutilisateur 382

36 Suivi anonyme 383
Activations anonymes 383
Demande d'état anonyme 384
Exemples de suivi anonyme 384

37 Suivi post-mortem 389
Affichage de clients DTrace 389
Affichage de données de suivi 390

38 Considérations sur les performances 395
Limitation des sondes activées 395

Utilisation de groupements 396
Utilisation de prédicats pouvant être mis en cache 396

39 Stabilité 399
Niveaux de stabilité 399
Classes de dépendance 402
Attributs d'interface 403
Rapports et calculs de stabilité 404
Mise en œuvre de la stabilité 407

40 Translateurs 409
Déclarations du translateur 409
Opérateur de conversion 411
Translateurs du modèle de processus 413
Conversions stables 413

41 Versionnage 415
Versions 415
Options de versionnage 416
Versionnage des fournisseurs 417

Glossaire 419

Index 421

Liste des figures

FIGURE 1–1 Aperçu de l'architecture DTrace et de ses composants 34

FIGURE 5–1 Représentation d'ensembles scalaires 84

FIGURE 5–2 Stockage de pointeur et d'ensemble 86

Liste des tableaux

TABLEAU 2–1 Mots-clés de D ... 49
TABLEAU 2–2 Types de données des nombres entiers en D ... 51
TABLEAU 2–3 Alias de type des nombres entiers en D ... 51
TABLEAU 2–4 Types de données à virgule flottante en langage D ... 52
TABLEAU 2–5 Séquences d'échappement de caractères en langage D ... 53
TABLEAU 2–6 Opérateurs arithmétiques binaires en langage D ... 54
TABLEAU 2–7 Opérateurs relationnels en langage D ... 55
TABLEAU 2–8 Opérateurs logiques de D ... 56
TABLEAU 2–9 Opérateurs de bit en langage D ... 56
TABLEAU 2–10 Opérateurs d'assignation en langage D ... 57
TABLEAU 2–11 Priorité et associativité des opérateurs de D ... 61
TABLEAU 3–1 Variables intégrées de DTrace ... 71
TABLEAU 4–1 Caractères de correspondance de modèle de nom de sonde ... 78
TABLEAU 6–1 Opérateurs relationnels et chaînes en langage D ... 93
TABLEAU 9–1 Fonctions de groupement DTrace ... 117
TABLEAU 13–1 Fonctions de spéculation de DTrace ... 172
TABLEAU 15–1 Variables de macro en D ... 191
TABLEAU 16–1 Options client DTrace ... 197
TABLEAU 18–1 Sondes de verrou adaptatif ... 208
TABLEAU 18–2 Sondes de verrou de rotation ... 209
TABLEAU 18–3 Sonde de verrou de thread ... 210
TABLEAU 18–4 Sondes de verrouillage en lecture/écriture ... 210
TABLEAU 19–1 Suffixes d'heure valides ... 213
TABLEAU 21–1 `syscall` Sonde de grands fichiers ... 234
TABLEAU 22–1 Sondes SDT ... 237
TABLEAU 23–1 Sondes `sysinfo` ... 245
TABLEAU 24–1 Sondes `vminfo` ... 254
TABLEAU 25–1 Sondes `proc` ... 261

TABLEAU 25–2 Arguments de sonde proc 263
TABLEAU 25–3 Valeurs de pr_flag 264
TABLEAU 25–4 Valeurs de pr_stype 265
TABLEAU 25–5 Valeurs de pr_state 266
TABLEAU 26–1 Sondes sched 275
TABLEAU 26–2 Arguments de sonde sched 278
TABLEAU 27–1 Sondes io 309
TABLEAU 27–2 Arguments de sonde io 310
TABLEAU 27–3 Valeurs de b_flags 311
TABLEAU 28–1 Sondes mib 329
TABLEAU 28–2 Sondes mib ICMP 330
TABLEAU 28–3 Sondes mib IP 332
TABLEAU 28–4 Sondes mib IPsec 333
TABLEAU 28–5 Sondes mib IPv6 334
TABLEAU 28–6 Sondes mib IP brutes 339
TABLEAU 28–7 Sondes mib SCTP 339
TABLEAU 28–8 Sondes mib TCP 342
TABLEAU 28–9 Sondes mib UDP 344
TABLEAU 29–1 Sondes fpuinfo 347
TABLEAU 31–1 Sondes Mutex 356
TABLEAU 31–2 Sondes de verrouillage en lecture/écriture 357
TABLEAU 33–1 Constantes uregs[] pour SPARC 366
TABLEAU 33–2 Constantes de uregs[] pour x86 367
TABLEAU 33–3 Constantes uregs[] pour amd64 368
TABLEAU 33–4 Constantes uregs[] communes 369
TABLEAU 40–1 Translateurs procfs.d 413
TABLEAU 41–1 Versions de DTrace 416

Liste des exemples

EXEMPLE 1–1 hello.d : Hello, World en langage de programmation D 29

EXEMPLE 1–2 trussrw.d : suivi des appels système avec le format de sortie truss (1) 42

EXEMPLE 1–3 rwtime.d : heure read(2) et write(2) appels 45

EXEMPLE 3–1 rtime.d : calcul du temps passé dans read(2) 67

EXEMPLE 3–2 clause.d : variables locales de clause 70

EXEMPLE 5–1 badptr.d : démonstration de la gestion d'erreurs DTrace 83

EXEMPLE 7–1 rwinfo.d : collecte des statistiques read(2) et write(2) 96

EXEMPLE 7–2 ksyms.d : suivi de la relation deread(2) et uiomove(9F) 100

EXEMPLE 7–3 kstat.d : appels de suivi vers kstat_data_lookup(3KSTAT) 104

EXEMPLE 9–1 renormalize.d : renormalisation d'un groupement 128

EXEMPLE 13–1 specopen.d : flux de code en cas d'échec open(2) 174

EXEMPLE 17–1 error.d: erreurs d'enregistrement 203

EXEMPLE 33–1 userfunc.d : suivi des entrées et renvois de la fonction utilisateur 369

EXEMPLE 33–2 errorpath.d : suivi du chemin d'accès au code d'une fonction utilisateur 372

EXEMPLE 34–1 myserv.d : sondes d'application définies statiquement 377

Préface

DTrace est un logiciel de suivi dynamique complet dédié au système d'exploitation Solaris™. DTrace offre une puissante infrastructure permettant aux administrateurs, développeurs et techniciens de maintenance de répondre de manière concise aux questions arbitraires relatives au comportement du système d'exploitation et des programmes utilisateur. Le *Guide de suivi dynamique de Solaris* décrit la procédure d'utilisation de DTrace pour observer, déboguer et optimiser le comportement du système. Ce manuel est également un support de référence complet quant aux outils d'observabilité intégrés de Tracer et le langage de programmation D.

Remarque – Cette version de Solaris prend en charge les systèmes utilisant les architectures de processeur SPARC® et x86 : UltraSPARC®, SPARC64, AMD64, Pentium et Xeon EM64T. Les systèmes pris en charge sont répertoriés dans la *liste de compatibilité matérielle de Solaris 10* disponible à l'adresse http://www.sun.com/bigadmin/hcl. Ce document présente les différences d'implémentation en fonction des divers types de plates-formes.

Dans ce document, x86 fait référence aux systèmes 64 bits et 32 bits composés de processeurs compatibles avec les familles de produits AMD64 ou Intel Xeon/Pentium. Pour connaître les systèmes pris en charge, reportez-vous à la *liste de compatibilité matérielle de Solaris 10*.

Utilisateurs de ce manuel

Si vous avez toujours souhaité comprendre le comportement de votre système, DTrace est l'outil qu'il vous faut. DTrace est un outil de suivi dynamique et complet intégré à Solaris. Il vous permet d'étudier le comportement des programmes utilisateur et d'examiner le comportement du système d'exploitation. DTrace peut être utilisé par les administrateurs système et les développeurs d'applications, ainsi que dans les environnements de production directe. DTrace vous permet d'explorer votre système, d'en comprendre le fonctionnement, de suivre les problèmes de performance dans les couches du logiciel ou de localiser la cause d'un comportement anormal. Vous constaterez que DTrace vous permet de créer vos propres programmes personnalisés pour instrumenter le système de manière dynamique et fournir des réponses concises et immédiates aux questions arbitraires que vous pouvez formuler à l'aide du langage de programmation D de DTrace.

DTrace permet à tous les utilisateurs de Solaris d'exécuter les actions suivantes :

- activer et gérer de manière dynamique des milliers de sondes ;

- associer de manière dynamique des prédicats et des actions logiques à des sondes ;
- gérer de manière dynamique des mémoires tampon et des stratégies de suivi ;
- afficher et examiner des données de suivi à partir du système en direct ou un vidage mémoire sur incident.

DTrace permet aux développeurs et aux administrateurs d'exécuter les actions suivantes :

- mettre en œuvre des scripts personnalisés utilisant le logiciel DTrace ;
- mettre en œuvre des outils multicouche utilisant DTrace pour récupérer des données de suivi.

Ce guide vous apprendra tout ce que vous devez savoir sur l'utilisation de DTrace. Une connaissance de base d'un langage de programmation comme C ou d'un langage de script comme `awk(1)` ou `perl(1)` vous permettra d'acquérir plus rapidement la maîtrise de DTrace et du langage de programmation D, mais il n'est pas nécessaire d'être un expert dans ces domaines de compétence. Si vous n'avez jamais rédigé de programme ou de script dans l'un des langages précédemment cités, la section “Informations connexes” à la page 23 vous indique d'autres documents qui pourront vous être utiles.

Organisation de ce document

Le Chapitre1, “Introduction” présente succinctement aux lecteurs le logiciel DTrace et le langage de programmation D. Le Chapitre2, “Types, opérateurs et expressions”, le Chapitre3, “Variables” et le Chapitre4, “Structure de programme D” présentent ensuite de façon plus approfondie les bases du langage D, ainsi que la méthode de conversion des programmes D en instrumentations dynamiques. Tous les lecteurs doivent commencer par lire ce premier groupe de chapitres.

Le Chapitre5, “Pointeurs et ensembles”, le Chapitre6, “Chaînes de caractères”, le Chapitre7, “Structs et Unions” et le Chapitre8, “Définitions des types et des constantes” présentent le reste des fonctions du langage D ; la plupart d'entre elles seront déjà connues des programmeurs en langage C, C++ et Java™. Les lecteurs qui ne connaissent pas ces langages doivent lire ces chapitres alors que les programmeurs plus expérimentés peuvent passer directement aux chapitres ultérieurs.

Le Chapitre9, “Groupements” et le Chapitre10, “Actions et sous-routines” présentent la puissante primitive de DTrace qui permet de *grouper* des données et l'ensemble des actions intégrées pouvant être utilisées pour créer des expériences de suivi. Tous les lecteurs doivent lire attentivement ces chapitres.

Le Chapitre11, “Tampons et mise en tampon” décrit les stratégies de mise en mémoire tampon des données de DTrace et la procédure de configuration correspondante. Les utilisateurs doivent lire ce chapitre une fois qu'ils ont compris la construction et le fonctionnement des programmes en D.

Le Chapitre12, “Format de sortie” décrit les actions de formatage de la sortie en D et la stratégie par défaut de formatage des données de suivi. Les lecteurs qui connaissent la fonction `printf()` du langage C peuvent survoler ce chapitre. Les lecteurs qui n'ont jamais rencontré `printf()` précédemment doivent lire ce chapitre avec beaucoup d'attention.

Le Chapitre13, “Suivi spéculatif” présente l'outil DTrace pour valider de *manière spéculative* les données vers un tampon de suivi. Ce chapitre doit être lu par les utilisateurs qui souhaitent utiliser DTrace dans une situation dans laquelle il est nécessaire d'assurer le suivi des données avant de savoir si elles répondent à la question.

Le Chapitre14, “Utilitaire `dtrace(1M)`” apporte une description compléte de l'utilitaire de ligne de commande `dtrace`, d'une façon similaire à la page du manuel en ligne correspondante. Les lecteurs peuvent souhaiter se reporter à ce chapitre, plusieurs options de ligne de commande étant présentées dans le manuel. Le Chapitre15, “Scripts” explique comment utiliser l'utilitaire `dtrace` pour construire des scripts en D exécutables et traiter les arguments de ligne de commande correspondants. Le Chapitre16, “Options et paramètres réglables” présente les options pouvant être optimisées sur la ligne de commande ou directement dans un programme en D.

Le groupe de chapitres du Chapitre17, “Fournisseur `dtrace`” et se terminant au Chapitre32, “Fournisseur `fasttrap`” présente les divers *fournisseurs* de DTrace pouvant être utilisés pour instrumenter divers aspects du système Solaris. Tous les lecteurs devraient survoler ces chapitres pour se familiariser avec les divers fournisseurs, puis revenir lire plus en détails certains chapitres, au besoin.

Le Chapitre33, “Suivi des processus utilisateur” propose des exemples d'utilisation de DTrace pour instrumenter des processus utilisateur. Le Chapitre34, “Suivi défini statiquement pour les applications utilisateur” explique comment les programmeurs d'applications peuvent ajouter des fournisseurs et des sondes DTrace personnalisés aux applications utilisateur. Les développeurs ou les administrateurs de programmes utilisateur qui souhaitent utiliser DTrace pour étudier le comportement des processus utilisateur doivent lire ces chapitres.

Le Chapitre35, “Sécurité” et les chapitres restants traitent des rubriques avancées sur la sécurité, le versionnage et les attributs de stabilité de DTrace, ainsi que de la manière d'exécuter un suivi pendant l'initialisation et un "post-mortem" avec DTrace. Ces chapitres sont dédiés à des utilisateurs expérimentés de DTrace.

Informations connexes

Les manuels et documents suivants sont liés aux tâches que vous devez exécuter avec DTrace. Nous vous conseillons de vous y référer également :

- Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language*. Prentice Hall, 1988. ISBN 0–13–110370–9
- Vahalia, Uresh. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996. ISBN 0-13-101908-2

- Mauro, Jim et McDougall, Richard. *Solaris Internals: Core Kernel Components*. Sun Microsystems Press, 2001. ISBN 0-13-022496-0

Vous pouvez partager votre expérience et vos scripts DTrace avec la communauté DTrace sur Internet à l'adresse http://www.sun.com/bigadmin/content/dtrace/.

Documentation, support et formation

Le site Web Sun fournit des informations sur les ressources supplémentaires suivantes :

- documentation ; (`http://www.sun.com/documentation/`)
- support ; (`http://www.sun.com/support/`)
- formation. (`http://www.sun.com/training/`)

Conventions typographiques

Le tableau ci-dessous décrit les conventions typographiques utilisées dans ce manuel.

TABLEAU P–1 Conventions typographiques

Type de caractères	Signification	Exemple
`AaBbCc123`	Noms des commandes, fichiers et répertoires, ainsi que messages système.	Modifiez votre fichier `.login`. Utilisez `ls -a` pour afficher la liste de tous les fichiers. `nom_machine% Vous avez reçu du courrier.`
`AaBbCc123`	Ce que vous entrez, par opposition à ce qui s'affiche à l'écran.	`nom_machine%` **`su`** `Mot de passe :`
aabbcc123	Paramètre fictif : à remplacer par un nom ou une valeur réel(le).	La commande permettant de supprimer un fichier est `rm` *nom_fichier*.

TABLEAU P–1 Conventions typographiques *(Suite)*

Type de caractères	Signification	Exemple
AaBbCc123	Titres de manuel, nouveaux termes et termes importants.	Reportez-vous au chapitre 6 du *Guide de l'utilisateur*. Un *cache* est une copie des éléments stockés localement. *N'enregistrez pas* le fichier. **Remarque :** En ligne, certains éléments mis en valeur s'affichent en gras.

Invites de shell dans les exemples de commandes

Le tableau suivant présente les invites système et les invites de superutilisateur UNIX® par défaut des C shell, Bourne shell et Korn shell.

TABLEAU P–2 Invites de shell

Shell	Invite
C shell	`nom_machine%`
C shell pour superutilisateur	`nom_machine#`
Bourne shell et Korn shell	`$`
Bourne shell et Korn shell pour superutilisateur	`#`

◆ ◆ ◆ CHAPITRE 1

Introduction

Bienvenue dans le Guide de suivi dynamique du système d'exploitation Solaris ! Si vous avez toujours souhaité comprendre le comportement de votre système, DTrace est l'outil qu'il vous faut. DTrace est un utilitaire de suivi dynamique complet. Intégré à Solaris, il peut être utilisé par des administrateurs et des développeurs travaillant sur des systèmes de production en direct, à des fins d'analyse du comportement des programmes utilisateur et du système d'exploitation lui-même. DTrace vous permet d'explorer votre système, d'en comprendre le fonctionnement, de suivre les problèmes de performance dans les nombreuses couches du logiciel ou de localiser la cause d'un comportement anormal. Vous constaterez que DTrace vous permet de créer vos propres programmes personnalisés pour instrumenter le système de manière dynamique et fournir des réponses concises et immédiates aux questions arbitraires que vous pouvez formuler à l'aide du langage de programmation D de DTrace. La première section de ce chapitre fournit une brève introduction à DTrace et vous explique comment écrire votre tout premier programme D. Le reste du chapitre présente l'ensemble des règles de programmation en D ainsi que des conseils et techniques pour procéder à une analyse approfondie de votre système. Vous pouvez partager votre expérience et vos scripts DTrace avec la communauté DTrace sur Internet à l'adresse http://www.sun.com/bigadmin/content/dtrace/. Tous les scripts donnés dans ce guide à titre d'exemple figurent dans le répertoire `/usr/demo/dtrace` de votre système Solaris.

Démarrage

DTrace vous aide à comprendre un système logiciel en vous permettant de modifier de manière dynamique le noyau du système d'exploitation et les processus utilisateur afin d'enregistrer les données que vous indiquez à des emplacements choisis, appelés *sondes*. Une sonde est un emplacement ou une activité à laquelle DTrace peut lier une requête pour effectuer une série d'*actions*, comme l'enregistrement d'un suivi de pile, d'un horodatage ou d'un argument à une fonction. Les sondes s'apparentent à des capteurs programmables répartis sur l'ensemble de votre système Solaris, à des emplacements stratégiques. Si vous souhaitez en comprendre le fonctionnement, utilisez DTrace pour programmer les capteurs appropriés afin d'enregistrer les informations que vous estimez utiles. Ensuite, dès qu'une sonde *se déclenche*, DTrace rassemble

les données obtenues et les consigne dans un rapport. Si vous ne spécifiez pas d'action à effectuer pour une sonde, DTrace se contentera de prendre note de chaque déclenchement de sonde.

Dans DTrace, chaque sonde porte deux noms : un ID unique sous forme de nombre entier et une chaîne interprétable par l'utilisateur. Notre apprentissage de DTrace va commencer par la création de requêtes très simples avec la sonde appelée `BEGIN`, qui se déclenche à chaque nouvelle requête de suivi. Vous pouvez utiliser l'option `-n` de l'utilitaire `dtrace(1M)` pour activer une sonde utilisant son nom de chaîne. Tapez la commande suivante :

```
# dtrace -n BEGIN
```

Après un court instant, DTrace vous informe de l'activation de la sonde et une ligne de sortie s'affiche, indiquant que la sonde `BEGIN` s'est déclenchée. Dès l'affichage de cette sortie, `dtrace` reste à l'état de pause dans l'attente du déclenchement d'autres sondes. Puisque vous n'avez pas activé d'autres sondes et que `BEGIN` ne se déclenche qu'une seule fois, appuyez sur Control-C dans votre shell pour quitter `dtrace` et revenir à votre invite de shell :

```
# dtrace -n BEGIN
dtrace: description 'BEGIN' matched 1 probe
CPU     ID              FUNCTION:NAME
  0      1                  :BEGIN
^C
#
```

La sortie indique que la sonde appelée `BEGIN` s'est déclenchée une fois et son nom ainsi que son ID de nombre entier, 1, sont affichés. Notez que, par défaut, le nom sous forme de nombre entier de la CPU sur laquelle la sonde s'est déclenchée s'affiche. Dans cet exemple, la colonne CPU indique que la commande `dtrace` était en cours d'exécution sur la CPU 0 lors du déclenchement de la sonde.

Vous pouvez créer des requêtes DTrace avec n'importe quel nombre de sondes et d'actions. Procédons à la création d'une requête simple avec deux sondes en ajoutant la sonde `END` à la commande de l'exemple précédent. La sonde `END` se déclenche lors de l'achèvement de l'opération de suivi. Saisissez la commande suivante, puis appuyez à nouveau sur Control-C dans votre shell après l'affichage de la ligne de sortie pour la sonde `BEGIN` :

```
# dtrace -n BEGIN -n END
dtrace: description 'BEGIN' matched 1 probe
dtrace: description 'END' matched 1 probe
CPU     ID              FUNCTION:NAME
  0      1                  :BEGIN
^C
  0      2                    :END
#
```

Vous constaterez qu'en appuyant sur Control-C pour quitter `dtrace` la sonde `END` se déclenche. `dtrace` signale le déclenchement de cette sonde avant la fermeture du programme.

Maintenant que vous en savez un peu plus sur le nommage et l'activation de sondes, vous êtes prêt à écrire la version DTrace du programme classique “Hello, World.” Vous pouvez non seulement saisir les tentatives DTrace sur la ligne de commande, mais également les écrire dans des fichiers texte avec le langage de programmation D. Dans un éditeur de texte, créez un nouveau fichier appelé `hello.d` et saisissez votre premier programme D :

EXEMPLE 1–1 `hello.d` : Hello, World en langage de programmation D

```
BEGIN
{
    trace("hello, world");
    exit(0);
}
```

Après avoir enregistré votre programme, vous pouvez l'exécuter avec l'option `-s` de `dtrace.` Tapez la commande suivante :

```
# dtrace -s hello.d
dtrace: script 'hello.d' matched 1 probe
CPU     ID                FUNCTION:NAME
  0      1                      :BEGIN   hello, world
#
```

Comme vous pouvez le constater, `dtrace` affiche la même sortie que précédemment, suivie du texte “`hello, world`”. Contrairement à l'exemple précédent, vous n'avez pas eu besoin d'attendre ou d'appuyer sur Control-C. Ces modifications ont été le résultat des *actions* spécifiées pour la sonde `BEGIN` dans `hello.d`. Procédons à l'exploration de la structure de votre programme D de façon plus détaillée afin de comprendre ce qui s'est produit.

Chaque programme D se compose d'une série de *clauses*, chacune d'entre elles décrivant une ou plusieurs sondes à activer et un ensemble d'actions facultatives à exécuter lors du déclenchement de la sonde. Ces actions sont répertoriées sous la forme d'une série d'instructions entre accolades `{ }` précédées du nom de la sonde. Chaque instruction se termine par un point-virgule (`;`). Votre première instruction utilise la fonction `trace()` pour indiquer que DTrace doit enregistrer l'argument spécifié, la chaîne “`hello, world`”, lors du déclenchement de la sonde `BEGIN` puis afficher le résultat. La seconde instruction utilise la fonction `exit()` pour indiquer que DTrace doit interrompre le suivi et quitter la commande `dtrace`. DTrace fournit un ensemble de fonctions utiles telles que `trace()` et `exit()` pour appeler vos programmes D. Pour appeler une fonction, spécifiez son nom suivi d'une liste d'arguments entre parenthèses. Toutes les fonctions D sont décrites au·Chapitre10, “Actions et sous-routines”.

À présent, si vous connaissez le langage de programmation C, vous avez dû vous rendre compte, à la lumière des exemples et des noms utilisés, que le langage de programmation D présentait de fortes similarités avec C. En effet, D est dérivé d'un grand sous-ensemble de C, combiné à un ensemble spécial de fonctions et de variables permettant de faciliter le suivi. Ces

fonctionnalités vous seront présentées plus en détails dans les chapitres suivants. Si vous avez déjà écrit un programme C auparavant, vous pourrez transférer immédiatement la plupart de vos connaissances pour créer des programmes de suivi en D. Dans le cas contraire, l'apprentissage de D reste très accessible. À la fin de ce chapitre, vous maîtriserez l'ensemble de la syntaxe. Mais pour commencer, oublions un peu les règles du langage pour nous intéresser au fonctionnement de DTrace, avant de revenir à l'apprentissage de la création de programmes D plus intéressants.

Fournisseurs et sondes

Dans les exemples précédents, vous avez appris à utiliser deux sondes simples appelées `BEGIN` et `END`. Mais d'où ces sondes proviennent-elles ? Les sondes DTrace proviennent d'un ensemble de modules de noyau appelés *fournisseurs*, dont chacun exécute une instrumentation particulière pour créer des sondes. Lorsque vous utilisez DTrace, chaque fournisseur a la possibilité de publier les sondes qu'il peut fournir dans la structure DTrace. Vous pouvez ensuite activer vos actions de suivi et les lier à n'importe quelle sonde publiée. Pour répertorier toutes les sondes disponibles sur votre système, tapez la commande suivante :

```
# dtrace -l
  ID   PROVIDER            MODULE          FUNCTION NAME
   1     dtrace                                     BEGIN
   2     dtrace                                     END
   3     dtrace                                     ERROR
   4   lockstat           genunix       mutex_enter adaptive-acquire
   5   lockstat           genunix       mutex_enter adaptive-block
   6   lockstat           genunix       mutex_enter adaptive-spin
   7   lockstat           genunix       mutex_exit  adaptive-release

   ... many lines of output omitted ...

#
```

Vous risquez de devoir patienter quelques instants avant que la sortie ne s'affiche. Pour connaître le nombre total de sondes, vous pouvez taper la commande suivante :

```
# dtrace -l | wc -l
        30122
```

Il est possible que le total affiché par votre machine ne soit pas le même car le nombre de sondes dépend de votre plate-forme d'exploitation ainsi que des logiciels installés. Comme vous pouvez le constater, le nombre de sondes disponibles est très élevé, permettant ainsi d'observer les moindres recoins, autrefois obscurs, de votre système. En fait, même cette sortie n'est pas la liste complète car, comme vous le verrez ultérieurement, certains fournisseurs offrent la possibilité de créer de nouvelles sondes à la volée, en fonction de vos requêtes de suivi, rendant ainsi le nombre réel de sondes DTrace presque illimité.

Revenez maintenant à la sortie de **`dtrace -l`** dans la fenêtre de votre terminal. Vous pouvez remarquer que chaque sonde comporte les deux noms mentionnés précédemment, à savoir un ID sous forme de nombre entier et un nom interprétable par l'utilisateur. Le nom interprétable par l'utilisateur se compose de quatre parties, présentées sous la forme de colonnes séparées dans la sortie `dtrace.` Un nom de sonde se compose des quatre parties suivantes :

Fournisseur	Nom du fournisseur DTrace publiant cette sonde. Il correspond généralement au nom du module de noyau DTrace qui exécute l'instrumentation permettant d'activer la sonde.
Module	Si cette sonde correspond à l'emplacement d'un programme spécifique, nom du module dans lequel se situe la sonde. Il s'agit soit du nom d'un module de noyau, soit du nom d'une bibliothèque utilisateur.
Fonction	Si cette sonde correspond à l'emplacement d'un programme spécifique, nom de la fonction du programme dans laquelle se situe la sonde.
Nom	Le composant final du nom de la sonde est un nom vous donnant une idée approximative de la signification sémantique de la sonde, comme `BEGIN` ou `END`.

Lors de l'écriture du nom complet d'une sonde, interprétable par l'utilisateur, écrivez les quatre parties du nom séparées par deux points, comme suit :

nom:*fonction*:*module*:*fournisseur*

Vous remarquerez que certaines sondes de la liste n'ont ni module ni fonction, comme les sondes `BEGIN` et `END` utilisées précédemment. Dans le cas de certaines sondes, ces champs restent vides car elles ne correspondent ni à une fonction ni à un emplacement de programme instrumenté. À la place, ces sondes font référence à un concept plus abstrait comme l'idée de fin de votre requête de suivi. Les sondes dont le nom est composé d'un module ou d'une fonction sont appelées *sondes ancrées*, tandis que les autres sont qualifiées de *non ancrées*.

Par convention, si vous ne spécifiez pas tous les champs d'un nom de sonde, DTrace effectue la recherche sur *l'ensemble* des sondes dont les parties du nom spécifiées correspondent aux critères. En d'autres termes, lorsque vous avez utilisé précédemment le nom de sonde `BEGIN`, cela revenait en fait à demander à DTrace de rechercher n'importe quelle sonde dont le champ de nom correspondait à `BEGIN`, quelle qu'ait été la valeur des champs fournisseur, module et fonction. Dans le cas présent, une seule sonde correspond à ces critères ; le résultat reste donc identique. Mais vous savez désormais que le vrai nom de la sonde `BEGIN` est `dtrace:::BEGIN`, ce qui indique que cette sonde est fournie par la structure DTrace elle-même et qu'elle n'est ancrée à aucune fonction. Par conséquent, le programme `hello.d` aurait pu être écrit comme suit et donner un résultat identique :

```
dtrace:::BEGIN
{
    trace("hello, world");
```

```
    exit(0);
}
```

Maintenant que vous avez compris d'où viennent les sondes et comment elles sont nommées, nous allons en apprendre davantage sur ce qui se produit lorsque vous activez des sondes et que vous demandez à DTrace d'exécuter des actions. Nous reprendrons ensuite notre présentation de D.

Compilation et instrumentation

Lorsque vous écrivez des programmes classiques dans Solaris, vous utilisez un compilateur pour convertir le code source de votre programme en code objet exécutable. Lorsque vous utilisez la commande `dtrace`, vous demandez au compilateur du langage D utilisé précédemment d'écrire le programme `hello.d.` Une fois compilé, votre programme est envoyé au noyau du système d'exploitation en vue d'être exécuté par DTrace. Les sondes nommées dans votre programme sont activées et le fournisseur correspondant exécute l'instrumentation nécessaire pour les activer.

L'ensemble de l'instrumentation dans DTrace est complètement dynamique : les sondes ne sont activées que lorsque vous les utilisez. Aucun code instrumenté n'est présent pour les sondes inactives ; votre système ne rencontre donc aucune dégradation des performances lorsque vous n'utilisez pas DTrace. Lorsque votre tentative est terminée et que la commande `dtrace` se ferme, toutes les sondes utilisées sont automatiquement désactivées et leur instrumentation supprimée, restaurant votre système à son état exact d'origine. Il n'existe aucune réelle différence entre un système sur lequel DTrace est inactif et un système sur lequel le logiciel DTrace n'est pas installé.

L'instrumentation de chaque sonde est réalisée de façon dynamique sur le système d'exploitation en cours d'exécution en direct ou sur les processus utilisateur sélectionnés. À aucun moment le système ne passe à l'état de veille ou de pause et le code d'instrumentation n'est ajouté que pour les sondes que vous activez. Ainsi, l'impact de l'utilisation de DTrace sur la sonde se limite uniquement aux actions que vous demandez à DTrace d'exécuter : aucune donnée étrangère ne fait l'objet d'un suivi, aucun “commutateur de suivi” d'envergure n'est activé sur le système et l'instrumentation DTrace dans son ensemble est conçue pour être aussi efficace que possible. Ces fonctions vous permettent d'utiliser DTrace en production pour résoudre des problèmes effectifs en temps réel.

La structure DTrace prend également en charge un nombre arbitraire de clients virtuels. Vous pouvez exécutez autant d'expériences et de commandes DTrace que vous le souhaitez, à condition de ne pas dépasser la limite de capacité de mémoire de votre système. Les commandes fonctionnent toutes de manière indépendante en utilisant la même instrumentation sous-jacente. Cette même fonction permet également à un nombre quelconque d'utilisateurs différents présents sur le même système d'utiliser DTrace simultanément : les développeurs, les administrateurs et les techniciens peuvent tous travailler ensemble ou sur des problèmes différents sur le même système en utilisant DTrace, sans qu'il n'y ait d'interférence entre eux.

Contrairement aux programmes écrits en C et C++ et à l'instar des programmes écrits dans le langage de programmation Java™ les programmes D de DTrace sont compilés dans un formulaire intermédiaire sécurisé exécuté lors du déclenchement des sondes. Ce formulaire intermédiaire est validé à des fins de sécurité lorsque votre programme est examiné pour la première fois par le logiciel de noyau DTrace. L'environnement d'exécution DTrace gère également toutes les erreurs d'exécution susceptibles de se produire pendant l'exécution de votre programme D, comme la division par zéro, le déréférencement de mémoire non valide, etc. et les consigne dans un rapport. Il est donc impossible de créer un programme DTrace non sûr, qui risquerait d'endommager par inadvertance le noyau Solaris ou l'un des processus en cours d'exécution sur votre système. Ces fonctions de sécurité vous permettent d'utiliser DTrace dans un environnement de production, sans risque d'arrêt brutal ou d'endommagement de votre système. Si vous faites une erreur de programmation, DTrace la consigne dans un rapport et désactive votre instrumentation. Vous pouvez alors corriger l'erreur et réessayer. Les fonctions DTrace de création de rapports d'erreurs et de débogage sont décrites ultérieurement dans ce manuel.

Le diagramme suivant présente les différents composants de l'architecture DTrace, notamment les fournisseurs, les sondes, le logiciel de noyau DTrace et la commande `dtrace`.

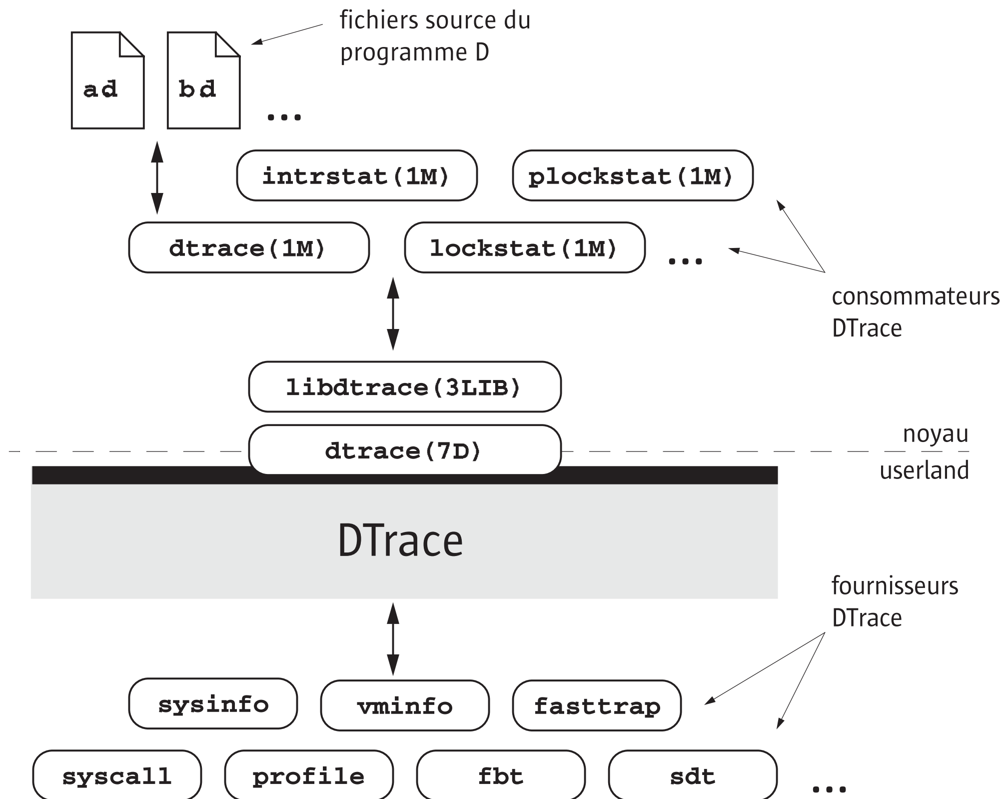

FIGURE 1–1 Aperçu de l'architecture DTrace et de ses composants

Maintenant que vous avez assimilé le fonctionnement de DTrace, revenons à la présentation du langage de programmation D pour commencer à écrire des programmes plus intéressants.

Variables et expressions arithmétiques

Notre prochain programme donné à titre d'exemple utilise le fournisseur `profile` DTrace pour implémenter un compteur temporel simple. Le fournisseur de profil permet de créer de nouvelles sondes en fonction des descriptions trouvées dans votre programme D. Si vous créez une sonde appelée `profile:::tick-`*n*`sec` pour certains nombres entiers *n*, le fournisseur de profil va créer une sonde qui se déclenche toutes les *n* secondes. Entrez le code source suivant et enregistrez-le dans le fichier `counter.d` :

```
/*
 * Count off and report the number of seconds elapsed
 */
dtrace:::BEGIN
```

```
{
    i = 0;
}

profile:::tick-1sec
{
    i = i + 1;
    trace(i);
}

dtrace:::END
{
    trace(i);
}
```

Lorsqu'il est exécuté, le programme compte le nombre de secondes écoulées avant que vous n'appuyiez sur Control-C puis affiche le total à la fin :

```
# dtrace -s counter.d
dtrace: script 'counter.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0  25499                       :tick-1sec         1
  0  25499                       :tick-1sec         2
  0  25499                       :tick-1sec         3
  0  25499                       :tick-1sec         4
  0  25499                       :tick-1sec         5
  0  25499                       :tick-1sec         6
^C
  0      2                             :END         6
#
```

Les trois premières lignes du programme sont des lignes de commentaire expliquant la finalité du programme. À l'instar des langages de programmation C, C++ et Java, le compilateur D ignore tous les caractères compris entre les symboles `/*` et `*/`. Les commentaires peuvent être utilisés n'importe où dans un programme D, à l'intérieur comme à l'extérieur de vos clauses de sonde.

La clause de la sonde `BEGIN` définit une nouvelle variable appelée `i` et lui assigne la valeur de nombre entier égal à zéro avec l'instruction :

```
i = 0;
```

Contrairement aux langages de programmation C, C++ et Java, il suffit d'utiliser les variables D dans une instruction de programme pour les créer ; aucune déclaration de variable explicite n'est requise. Lorsqu'une variable est utilisée pour la première fois dans un programme, son type est défini en fonction du type de sa première assignation. Chaque variable se voit assigner un seul type tout au long de la vie du programme ; les références ultérieures doivent donc correspondre au type de l'assignation d'origine. Dans `counter.d`, la variable `i` se voit d'abord

assigner la valeur constante de nombre entier égal à zéro. Son type est donc défini sur `int`. D fournit les mêmes types de données de nombres entiers de base que C, notamment :

`char`	Caractère ou nombre entier à octet unique
`int`	Nombre entier par défaut
`short`	Nombre entier court
`long`	Nombre entier long
`long long`	Nombre entier long étendu

Les formats de ces types dépendent du modèle de données du noyau du système d'exploitation, décrit dans le Chapitre2, "Types, opérateurs et expressions" D fournit également des noms conviviaux intégrés pour les types de nombres entiers signés et non signés de taille fixe différente, ainsi que des milliers d'autres types définis par le système d'exploitation.

La partie centrale de `counter.d` est la clause de sonde qui incrémente le compteur `i` :

```
profile:::tick-1sec
{
    i = i + 1;
    trace(i);
}
```

Cette clause nomme la sonde `profile:::tick-1sec`, indiquant au fournisseur `profile` qu'il doit créer une nouvelle sonde qui se déclenche une fois par seconde sur un processeur disponible. La clause comporte deux instructions : la première assigne à `i` la valeur précédente plus un, la seconde procède au suivi de la nouvelle valeur de `i`. Tous les opérateurs arithmétiques habituels de C sont disponibles en D ; la liste complète figure dans le Chapitre2, "Types, opérateurs et expressions" Ainsi, comme dans C, l'opérateur ++ peut être utilisé comme abrégé pour incrémenter de un la variable correspondante. La fonction `trace()` prend comme argument n'importe quelle expression D, de façon à écrire `counter.d` de façon plus concise, comme suit :

```
profile:::tick-1sec
{
    trace(++i);
}
```

Si vous souhaitez contrôler de façon explicite le type de variable `i`, vous pouvez mettre entre parenthèses le type souhaité lors de son assignation, afin de *forcer le type* du nombre entier égal à zéro vers un type spécifique. Par exemple, si vous souhaitez déterminer le nombre maximum de `char` en D, vous pouvez modifier la clause `BEGIN` comme suit :

```
dtrace:::BEGIN
{
    i = (char)0;
}
```

Après l'exécution de `counter.d`, vous verrez la valeur faisant l'objet d'un suivi augmenter puis revenir à zéro au bout de quelque temps, une fois la recherche circulaire effectuée. Si l'attente vous semble trop longue, essayez de modifier le nom de la sonde `profile` en `profile:::tick-100msec` pour que l'incrémentation se fasse toutes les 100 millisecondes, soit 10 fois par seconde.

Prédicats

L'une des grandes différences entre D et d'autres langages de programmation comme C, C++ et Java réside dans l'absence de constructions de flux de contrôle comme les instructions Si et les boucles. Les clauses du programme D sont écrites en tant que listes d'instructions simples sur une seule ligne qui procèdent au suivi d'une quantité de données fixe, facultative. D fournit la possibilité de suivre et modifier le flux de contrôle sous certaines conditions à l'aide d'expressions logiques appelées *prédicats* pouvant être utilisées pour placer en préfixe des clauses du programme. Une expression de prédicat est évaluée lors du déclenchement de la sonde avant l'exécution de l'une des instructions associées à la clause correspondante. Si le prédicat renvoie True, représenté par une valeur différente de zéro, la liste d'instructions est exécutée. Si le prédicat renvoie False, représenté par une valeur égale à zéro, aucune des instructions n'est exécutée et le déclenchement de la sonde est ignoré.

Tapez le code source suivant pour le prochain exemple et enregistrez-le dans un fichier appelé `countdown.d` :

```
dtrace:::BEGIN
{
    i = 10;
}

profile:::tick-1sec
/i > 0/
{
    trace(i--);
}

profile:::tick-1sec
/i == 0/
{
    trace("blastoff!");
    exit(0);
}
```

Ce programme D implémente un compte à rebours de 10 secondes avec des prédicats. Lorsqu'il est exécuté, le fichier `countdown.d` effectue un compte à rebours à partir de 10, affiche un message et se ferme :

```
# dtrace -s countdown.d
dtrace: script 'countdown.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
    0  25499                         :tick-1sec        10
    0  25499                         :tick-1sec         9
    0  25499                         :tick-1sec         8
    0  25499                         :tick-1sec         7
    0  25499                         :tick-1sec         6
    0  25499                         :tick-1sec         5
    0  25499                         :tick-1sec         4
    0  25499                         :tick-1sec         3
    0  25499                         :tick-1sec         2
    0  25499                         :tick-1sec         1
    0  25499                         :tick-1sec   blastoff!
#
```

Cet exemple utilise la sonde`BEGIN` pour initialiser un nombre entier `i` sur 10 pour commencer le compte à rebours. Ensuite, comme dans l'exemple précédent, le programme utilise la sonde `tick-1sec` pour implémenter une horloge qui se déclenche une fois par seconde. Vous remarquerez que dans `countdown.d`, la description de la sonde `tick-1sec` est utilisée dans deux clauses différentes, chacune avec un prédicat et une liste d'actions différents. Le prédicat est une expression logique entre barres obliques `/ /` qui suit le nom de la sonde et précède les accolades `{ }` entourant la liste d'instructions.

Le premier prédicat teste si `i` est supérieur à zéro, indiquant que l'horloge est toujours en cours d'exécution :

```
profile:::tick-1sec
/i > 0/
{
    trace(i--);
}
```

L'opérateur relationnel > signifie *supérieur à* et renvoie un nombre entier d'une valeur égale à zéro pour False et égale à un pour True. Tous les opérateurs arithmétiques relationnels de C sont pris en charge en D ; la liste complète figure dans le Chapitre2, “Types, opérateurs et expressions” Si `i` n'est pas encore égal à zéro, le script effectue le suivi de `i`, puis le décrémente en utilisant l'opérateur `--`.

Le second prédicat utilise l'opérateur == pour renvoyer True lorsque `i` est égal à zéro, indiquant que le compte à rebours est terminé :

```
profile:::tick-1sec
/i == 0/
{
    trace("blastoff!");
    exit(0);
}
```

Comme dans le premier exemple, `hello.d` `countdown.d` utilise une séquence de caractères entre guillemets appelée *constante de chaîne*, pour afficher un message final lorsque le compte à rebours est terminé. La fonction `exit()` est utilisée pour quitter `dtrace` et revenir à l'invite de shell.

Si vous regardez la structure de `countdown.d`, vous constaterez qu'en créant deux clauses avec la même description de sonde mais avec différents prédicats et actions, nous avons effectivement créé le flux logique :

```
i = 10
once per second,
  if i is greater than zero
    trace(i--);
  otherwise if i is equal to zero
    trace("blastoff!");
    exit(0);
```

Lorsque vous souhaitez écrire des programmes complexes avec des prédicats, essayez d'abord de visualiser votre algorithme de cette façon, puis transformez chaque chemin de vos constructions conditionnelles en une clause et un prédicat séparés.

Vous pouvez maintenant combiner des prédicats avec un nouveau fournisseur, le fournisseur `syscall` et créer votre premier vrai programme de suivi D. Le fournisseur `syscall` vous permet d'activer des sondes lors de l'entrée ou du renvoi de n'importe quel appel système Solaris. Le prochain exemple utilise DTrace pour observer à chaque fois que votre shell effectue un appel système `read(2)` ou `write(2)`. Ouvrez d'abord deux fenêtres de terminal, l'une pour utiliser DTrace et l'autre contenant le processus de shell que vous allez observer. Dans la seconde fenêtre, tapez la commande suivante pour obtenir l'ID de processus de ce shell :

```
# echo $$
12345
```

Revenez maintenant à votre première fenêtre de terminal, tapez le programme D suivant et enregistrez-le dans un fichier appelé `rw.d`. En tapant le programme, remplacez *12345* par l'ID de processus du shell affiché en réponse à votre commande `echo`.

```
syscall::read:entry,
syscall::write:entry
/pid == 12345/
{
```

```
}
```

Vous remarquerez que la clause de sonde rw.d reste vide car le programme est uniquement destiné à suivre les notifications de déclenchement de sondes et non à suivre des données supplémentaires. Lorsque vous avez terminé la saisie dans rw.d, utilisez dtrace pour commencer votre expérience puis accédez à la seconde fenêtre de shell et tapez quelques commandes, en appuyant sur Entrée après chacune. Lors de la saisie, vous devez voir les déclenchements de sonde dtrace consignés dans un rapport dans votre première fenêtre, comme dans l'exemple qui suit :

```
# dtrace -s rw.d
dtrace: script 'rw.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
    0     34                        write:entry
    0     32                         read:entry
    0     34                        write:entry
    0     32                         read:entry
    0     34                        write:entry
    0     32                         read:entry
    0     34                        write:entry
    0     32                         read:entry
...
```

Vous observez à présent votre shell exécuter des appels système read(2) et write(2) pour lire un caractère à partir de votre fenêtre de terminal et renvoyer le résultat. Cet exemple comprend un grand nombre des concepts décrits jusqu'à présent et quelques nouveaux concepts également. Tout d'abord, pour instrumenter read(2) et write(2) de la même façon, le script utilise une seule clause de sonde avec plusieurs descriptions de sonde en les séparant par des virgules, comme suit :

```
syscall::read:entry,
syscall::write:entry
```

Pour une meilleure lisibilité, chaque description de sonde apparaît sur sa propre ligne. Cette disposition n'est pas strictement requise mais elle contribue à une meilleure lisibilité du script. Le script définit ensuite un prédicat correspondant uniquement aux appels système exécutés par votre processus de shell :

```
/pid == 12345/
```

Le prédicat utilise la variable DTrace prédéfinie pid, qui évalue toujours l'ID de processus associé au thread ayant déclenché la sonde correspondante. DTrace fournit de nombreuses définitions de variables intégrées pour obtenir des informations utiles comme l'ID de processus. Voici une liste de quelques variables DTrace pouvant être utilisées pour écrire vos premiers programmes D :

Nom de variable	Type de données	Signification
errno	int	Valeur errno actuelle pour les appels système
execname	string	Nom du fichier exécutable du processus actuel
pid	pid_t	ID de processus du processus actuel
tid	id_t	ID de thread du thread actuel
probeprov	string	Champ du fournisseur de la description de sonde actuelle
probemod	string	Champ du module de la description de sonde actuelle
probefunc	string	Champ de la fonction de la description de sonde actuelle
probename	string	Champ du nom de la description de sonde actuelle

Maintenant que vous avez écrit un vrai programme d'instrumentation, essayez de l'expérimenter sur différents processus s'exécutant sur votre système en modifiant l'ID de processus et les sondes d'appels système instrumentées. Vous pouvez ensuite effectuer une modification supplémentaire et changer `rw.d` en une version très simple d'un outil de suivi des appels système comme `truss(1)`. Utiliser un champ de description de sonde vide revient à utiliser un caractère générique, correspondant à n'importe quelle sonde. Vous devez donc modifier votre programme selon le nouveau code source suivant pour suivre *n'importe quel* appel système exécuté par votre shell :

```
syscall:::entry
/pid == 12345/
{

}
```

Essayez de taper quelques commandes dans le shell comme `cd`, `ls` et `date` et examinez le rapport créé par votre programme DTrace.

Format de sortie

Le suivi d'appels système est une méthode efficace pour observer le fonctionnement de la plupart des processus utilisateur. Si vous avez utilisé l'utilitaire Solaris `truss(1)` auparavant en tant qu'administrateur ou développeur, vous avez sans doute pu constater qu'il s'agit d'un outil utile qu'il convient de garder à portée de main en cas de problème. Si vous n'avez jamais utilisé `truss` auparavant, essayez-le dès à présent en tapant cette commande dans l'un de vos shells :

```
$ truss date
```

Un suivi formaté de tous les appels système exécutés par `date(1)` s'affiche, suivi de sa propre ligne de sortie à la fin. L'exemple suivant améliore le programme précédent `rw.d` en formatant sa sortie sur `truss(1)` de façon à mieux la comprendre. Tapez le programme suivant et enregistrez-le dans le fichier `trussrw.d` :

EXEMPLE 1–2 `trussrw.d` : suivi des appels système avec le format de sortie `truss` (1)

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
    printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
}

syscall::read:return,
syscall::write:return
/pid == $1/
{
    printf("\t\t = %d\n", arg1);
}
```

Dans cet exemple, la constante `12345` est remplacée par l'étiquette `$1` dans chaque prédicat. Cette étiquette vous permet de spécifier le processus d'intérêt en tant qu'*argument* au script : `$1` est remplacé par la valeur du premier argument lorsque le script est compilé. Pour exécuter `trussrw.d`, utilisez les options `dtrace`, `-q` et `-s`, suivies par l'ID de processus de votre shell en tant qu'argument final. L'option `- q` indique que `dtrace` doit être silencieux et doit supprimer la ligne d'en-tête et les colonnes de CPU et d'ID présentées dans les exemples précédents. Ainsi, seul le résultat des données suivies explicitement s'affichera. Tapez la commande suivante (en remplaçant `12345` par l'ID d'un processus de shell) puis appuyez sur Entrée plusieurs fois dans le shell spécifié :

```
# dtrace -q -s trussrw.d 12345
                     = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)                = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)                = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
```

```
write(2, 0x8089e48,   52)                = 52
read(0, 0x8089878,    1)^C
#
```

Examinons plus en détail votre programme D et sa sortie. D'abord, une clause similaire au programme précédent instrumente chacun des appels du shell sur `read(2)` et `write(2)`. Mais pour cet exemple, une nouvelle fonction, `printf()`, est utilisée pour effectuer le suivi des données et les afficher dans un format spécifique.

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
    printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
}
```

La fonction `printf()` combine la capacité de suivi des données comme dans la fonction `trace()` utilisée précédemment, avec la capacité de sortie des données et d'autres textes dans un format spécifique décrit. La fonction `printf()` indique à DTrace de suivre les données associées à chaque argument faisant suite au premier argument, puis de formater les résultats en utilisant les règles décrites par le premier argument `printf()` connu sous le nom de *chaîne de format*.

La chaîne de format est une chaîne normale contenant n'importe quel nombre de conversions de format, chacune commençant par le caractère `%` qui décrit comment formater l'argument correspondant. La première conversion dans la chaîne de format correspond au second argument `printf()`, la seconde conversion au troisième argument, etc. L'ensemble du texte entre les conversions est affiché textuellement. Le caractère suivant le caractère de conversion `%` décrit le format à utiliser pour l'argument correspondant. Voici la signification des conversions de format utilisées dans `trussrw.d` :

`%d`	Imprime la valeur correspondante en tant que nombre entier décimal
`%s`	Imprime la valeur correspondante en tant que chaîne
`%x`	Imprime la valeur correspondante en tant que nombre entier hexadécimal

La fonction `printf()` de DTrace fonctionne de la même façon que la routine de bibliothèque `printf(3C)` C ou l'utilitaire `printf(1)` du shell. Si vous n'avez jamais rencontré la fonction `printf()` auparavant, les formats et options sont expliqués en détails dans le Chapitre12, “Format de sortie”. Vous devez lire ce chapitre attentivement, même si vous connaissez déjà la fonction `printf()` dans un autre langage. En D, `printf()` est fourni de façon intégrée et certaines nouvelles conversions de format sont disponibles, conçues spécifiquement pour DTrace.

Pour vous aider à écrire des messages corrects, le compilateur D valide chaque chaîne de format `printf()` par rapport à sa liste d'arguments. Essayez de modifier `probefunc` dans la clause

précédente en nombre entier 123. Si vous exécutez le programme modifié, un message d'erreur s'affiche, vous informant que la conversion du format de la chaîne %s ne convient pas pour une utilisation avec un argument de nombre entier.

```
# dtrace -q -s trussrw.d
dtrace: failed to compile script trussrw.d: line 4: printf( )
       argument #2 is incompatible with conversion #1 prototype:
            conversion: %s
             prototype: char [] or string (or use stringof)
              argument: int
#
```

Pour afficher le nom de l'appel système de lecture et d'écriture et ses arguments, utilisez l'instruction printf() :

```
printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
```

pour suivre le nom de la fonction de sonde actuelle et les trois premiers arguments de nombre entier dans l'appel système, disponible dans les variables DTrace arg0, arg1 et arg2. Pour plus d'informations sur les arguments de sonde, reportez-vous au Chapitre3, “Variables”. Le premier argument pour read(2) et write(2) est un descripteur de fichier, imprimé en décimales. Le second argument est une adresse tampon, formatée en tant que valeur hexadécimale. L'argument final est le format tampon, formaté en tant que valeur décimale. Le spécificateur de format %4d est utilisé pour le troisième argument pour indiquer que la valeur doit être imprimée en utilisant la conversion de format %d avec un champ d'une largeur maximale de 4 caractères. Si le nombre entier fait moins de 4 caractères de large, printf() insère des espaces supplémentaires pour aligner le résultat.

Pour afficher le résultat de l'appel système et compléter chaque ligne de sortie, utilisez la clause suivante :

```
syscall::read:return,
syscall::write:return
/pid == $1/
{
    printf("\t\t = %d\n", arg1);
}
```

Vous remarquerez que le fournisseur syscall publie également une sonde appelée return pour chaque appel système, en plus de entry. La variable DTrace arg1 pour les sondes syscall return évalue la valeur de retour de l'appel système. La valeur de retour est formatée en tant que nombre entier décimal. Les séquences de caractères commençant par des barres obliques inversées dans la chaîne de format s'étendent à la tabulation (\t) et à la nouvelle ligne (\n) respectivement. Ces *séquences d'échappement* vous aident à afficher ou à enregistrer des caractères difficiles à saisir. D prend en charge le même ensemble de séquences d'échappement que les langages de programmation C, C++ et Java. Vous trouverez la liste complète des séquences d'échappement au Chapitre2, “Types, opérateurs et expressions”.

Tableaux

D vous permet de définir des variables qui sont des nombres entiers ainsi que d'autres types pour représenter des chaînes et des types composites appelés *structs* et *unions*. Si vous connaissez la programmation en C, vous serez ravi d'apprendre que tout ce que vous pouvez saisir en C peut l'être en D. Si vous n'êtes pas un expert en C, ne vous en faites pas : les différents types de données sont tous décrits dans le Chapitre2, “Types, opérateurs et expressions” D prend également en charge un type de variable appelé *tableau associatif*. Un tableau associatif ressemble à un tableau normal, dans la mesure où il associe un jeu de clés à un jeu de valeurs, mais dans un tableau associatif, les clés ne sont pas limitées aux nombres entiers d'une plage fixe.

Les tableaux associatifs D peuvent être indexés par une liste comportant au moins une valeur, de n'importe quel type. Ensemble, les valeurs des clés individuelles forment un *tuple* utilisé pour l'indexage dans le tableau et l'accès ou la modification de la valeur correspondant à cette clé. Chaque tuple utilisé avec un tableau associatif donné doit correspondre à la signature du même type, c'est-à-dire que toutes les clés de tuple doivent avoir la même longueur et leurs types doivent être identiques et dans le même ordre. La valeur associée à chaque élément d'un tableau associatif donné est également un type fixe unique pour l'ensemble du tableau. Par exemple, l'instruction D suivante définit un nouveau tableau associatif `a`, dont la valeur est de type `int` avec la signature de tuple `[ string, int ]` et stocke la valeur de nombre entier 456 dans le tableau.

```
a["hello", 123] = 456;
```

Dès qu'un tableau est défini, il est possible d'accéder à ces éléments comme n'importe quelle autre variable D. Par exemple, l'instruction D suivante modifie l'élément de tableau précédemment stocké dans `a` en incrémentant la valeur de 456 à 457 :

```
a["hello", 123]++;
```

Les valeurs de tous les éléments de tableau qui n'ont pas encore été attribués sont définies sur zéro. Utilisons maintenant un tableau associatif dans un programme D. Tapez le programme suivant et enregistrez-le dans le fichier `rwtime.d` :

EXEMPLE 1–3 `rwtime.d` : heure `read`(2) et `write`(2) appels

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
    ts[probefunc] = timestamp;
}

syscall::read:return,
syscall::write:return
```

EXEMPLE 1–3 `rwtime.d` : heure `read(2)` et `write(2)` appels *(Suite)*

```
/pid == $1 && ts[probefunc] != 0/
{
    printf("%d nsecs", timestamp - ts[probefunc]);
}
```

Comme dans le cas de `trussrw.d`, spécifiez l'ID de processus shell lorsque vous exécutez `rwtime.d`. Si vous tapez quelques commandes shell, le temps écoulé pendant chaque appel système s'affiche. Tapez la commande suivante puis appuyez sur Entrée plusieurs fois dans votre autre shell :

```
# dtrace -s rwtime.d ‘pgrep -n ksh‘
dtrace: script ’rwtime.d’ matched 4 probes
CPU     ID                      FUNCTION:NAME
  0     33                        read:return 22644 nsecs
  0     33                        read:return 3382 nsecs
  0     35                       write:return 25952 nsecs
  0     33                        read:return 916875239 nsecs
  0     35                       write:return 27320 nsecs
  0     33                        read:return 9022 nsecs
  0     33                        read:return 3776 nsecs
  0     35                       write:return 17164 nsecs
...
^C
#
```

Pour suivre le temps écoulé pour chaque appel système, vous devez instrumenter à la fois l'entrée et le retour de `read(2)` et `write(2)` et tester le temps écoulé à chaque point. Puis, lors du retour d'un appel système donné, vous devez calculer la différence entre notre premier et notre second horodatage. Il est possible d'utiliser des variables séparées pour chaque appel système, mais il peut s'avérer gênant pour le programme de s'étendre vers des appels système supplémentaires. À la place, il est plus facile d'utiliser un tableau associatif indexé par le nom de la fonction de sonde. Voici la première clause de sonde :

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
    ts[probefunc] = timestamp;
}
```

Cette clause définit un tableau appelé `ts` et assigne au membre approprié la valeur de la variable DTrace `timestamp`. Cette variable renvoie la valeur d'un compteur à incrémentation constante en nanosecondes, comme la routine de bibliothèque Solaris `gethrtime(3C)`. Dès que

l'horodatage d'entrée est enregistré, les sondes de retour correspondantes `timestamp` sont à nouveau testées et les différences entre l'heure actuelle et la valeur enregistrée sont consignées dans un rapport.

```
syscall::read:return,
syscall::write:return
/pid == $1 && ts[probefunc] != 0/
{
    printf("%d nsecs", timestamp - ts[probefunc]);
}
```

Le prédicat de la sonde de retour nécessite que DTrace effectue le suivi du processus approprié et que la sonde `entry` correspondante se soit déjà déclenchée et ait déjà assigné une valeur `ts[probefunc]` différente de zéro. Cette opération permet d'éliminer les résultats non valides lors du démarrage de DTrace. Si votre shell est déjà en attente d'un appel système `read(2)` pour la saisie lors de l'exécution de `dtrace`, la sonde `read:return` se déclenchera même si elle n'est pas précédée de `read:entry`, car ce premier `read(2)` et `ts[probefunc]` renverront une valeur égale à zéro car elle n'a pas encore été assignée.

Types et symboles externes

L'instrumentation DTrace s'exécute à l'intérieur du noyau du système d'exploitation Solaris. En plus d'accéder à des variables DTrace spéciales et à des arguments de sonde, vous pouvez donc également accéder à des structures de données du noyau, des symboles et des types. Ces capacités permettent à des utilisateurs, administrateurs, techniciens et développeurs pilotes DTrace d'étudier le fonctionnement à faible niveau du noyau du système d'exploitation et des pilotes de périphériques. La liste de lecture figurant au début de ce manuel comporte des livres pouvant vous aider à en savoir plus sur le fonctionnement interne du système d'exploitation.

D utilise le caractère d'apostrophe inversée (‘) en tant qu'opérateur d'étendue spéciale pour accéder à des symboles définis dans le système d'exploitation et non dans votre programme D. Par exemple, le noyau de Solaris contient une déclaration en C du réglage système `kmem_flags` pour activer les fonctions de débogage du programme d'allocation de mémoire. Reportez-vous au manuel *Solaris Tunable Parameters Reference Manual* pour plus d'informations sur `kmem_flags`. Ce réglage est déclaré en C dans le code source du noyau, comme suit :

```
int kmem_flags;
```

Pour effectuer le suivi de la valeur de cette variable dans un programme D, vous pouvez écrire l'instruction D :

```
trace(‘kmem_flags);
```

DTrace associe chaque symbole de noyau au type utilisé dans le code C du système d'exploitation correspondant, fournissant un accès source facile aux structures de données de

système d'exploitation. Les noms des symboles de noyau sont conservés dans un espace de nom séparé de la variable D et des identificateurs de fonction. Vous n'avez donc jamais besoin de vous soucier des noms en conflit avec vos variables D.

Vous avez maintenant terminé la présentation de DTrace et vous connaissez à présent un grand nombre des blocs de construction DTrace nécessaires à la création de programmes D de plus grande envergure et d'une plus grande complexité. Les chapitres suivants décrivent l'ensemble des règles à connaître en D et montrent comment DTrace peut effectuer des mesures de performance complexes et faciliter l'analyse fonctionnelle du système. Plus tard, vous apprendrez à utiliser DTrace pour établir le lien entre le fonctionnement de l'application utilisateur et celui du système, vous donnant ainsi la possibilité d'analyser votre pile logicielle dans son intégralité.

Ce n'est que le début !

CHAPITRE 2

Types, opérateurs et expressions

D permet d'accéder à de nombreux objets de données et de les manipuler : possibilité de créer et modifier des variables et des structures de données, d'accéder aux objets de données définis dans le noyau du système d'exploitation et aux processus utilisateur et de déclarer les nombres entiers, les virgules flottantes et les constantes de chaîne. D fournit un sur-ensemble des opérateurs ANSI-C utilisés pour manipuler des objets et créer des expressions complexes. Ce chapitre présente l'ensemble détaillé des règles concernant les types, les opérateurs et les expressions.

Noms d'identifiant et mots de passe

Les noms d'identifiant en D comprennent des majuscules et des minuscules, des chiffres et des traits de soulignement, le premier caractère devant être une lettre ou un trait de soulignement. Tous les noms d'identifiant commençant par un trait de soulignement (_) sont réservés aux bibliothèques système du langage D. Vous devez éviter d'utiliser de tels noms dans vos programmes en D. Les programmeurs en langage D attribuent, par convention, des noms combinant majuscules et minuscules aux variables et des noms ne comportant que des majuscules aux constantes.

Les mots-clés du langage D sont des identifiants spéciaux dont l'utilisation est réservée à la syntaxe du langage de programmation elle-même. Ces noms sont toujours spécifiés en minuscules et ne peuvent pas être attribués à des variables du langage D.

TABLEAU 2–1 Mots-clés de D

`auto`*	`goto`*	`sizeof`
`break`*	`if`*	`static`*
`case`*	`import`*+	`string`+

TABLEAU 2–1 Mots-clés de D *(Suite)*

`char`	`inline`	`stringof`+
`const`	`int`	`struct`
`continue`*	`long`	`switch`*
`counter`*+	`offsetof`+	`this`+
`default`*	`probe`*+	`translator`+
`do`*	`provider`*+	`typedef`
`double`	`register`*	`union`
`else`*	`restrict`*	`unsigned`
`enum`	`return`*	`void`
`extern`	`self`+	`volatile`
`float`	`short`	`while`*
`for`*	`signed`	`xlate`+

D réserve un sur-ensemble de mots-clés en ANSI-C à utiliser comme mots-clés. Les mots-clés réservés pour un usage ultérieur par le langage D sont suivis du symbole “*”. Le compilateur D crée une erreur de syntaxe si vous essayez d'utiliser un mot-clé réservé à un usage ultérieur. Les mots-clés définis par D et non par ANSI-C sont suivis du symbole “+”. D fournit tous les types et opérateurs disponibles en ANSI-C. L'absence de structures de flux de commande constitue la principale différence dans la programmation D. Les mots-clés associés au flux de commande en ANSI-C sont réservés à un usage ultérieur dans D.

Types et tailles des données

D fournit des types de données fondamentaux pour les nombres entiers et les constantes à virgule flottante. Dans les programmes en D, l'arithmétique n'est réalisable que sur les nombres entiers. Vous pouvez utiliser les constantes à virgule flottante pour initialiser les structures de données, par contre, l'arithmétique à virgule flottante n'est pas autorisée dans D. D propose un modèle de données 32 et 64 bits pour les programmes d'écriture. Le modèle de données utilisé lors de l'exécution de votre programme correspond au modèle de données natif associé au noyau du système d'exploitation actif. Vous pouvez déterminer le modèle de données natif à l'aide de `isainfo -b`.

Le nom et la taille des types de nombre entier de chacun des deux modèles de données sont indiqués dans le tableau suivant. Les nombres entiers sont toujours représentés sous une forme à deux compléments dans l'ordre de codage natif des octets de votre système.

TABLEAU 2–2 Types de données des nombres entiers en D

Nom du type	En 32–bits	En 64–bits
char	1 octet	1 octet
short	2 octets	2 octets
int	4 octets	4 octets
long	4 octets	8 octets
long long	8 octets	8 octets

Un préfixe ayant un qualificatif `signed` ou `unsigned` peut être ajouté aux types de nombre entier. En l'absence de qualificatif, le type est considéré comme étant signé. Le compilateur D fournit également les alias de type récapitulés dans le tableau suivant :

TABLEAU 2–3 Alias de type des nombres entiers en D

Nom du type	Description
int8_t	Nombre entier signé d'un octet
int16_t	Nombre entier signé de 2 octets
int32_t	Nombre entier signé de 4 octets
int64_t	Nombre entier signé de 8 octets
intptr_t	Nombre entier signé de la taille d'un pointeur
uint8_t	Nombre entier non signé d'un octet
uint16_t	Nombre entier non signé de 2 octets
uint32_t	Nombre entier non signé de 4 octets
uint64_t	Nombre entier non signé de 8 octets
uintptr_t	Nombre entier non signé de la taille d'un pointeur

Ces alias de type reviennent à utiliser le nom du type de base correspondant dans le tableau précédent. Par ailleurs, ils sont correctement définis pour chaque modèle de données. Par exemple, le nom de type `uint8_t` correspond à un alias du type `unsigned char`. Pour plus d'informations sur la procédure de définition de vos propres alias de types à utiliser dans vos programmes en D, reportez-vous au Chapitre8, “Définitions des types et des constantes”.

D propose des types à virgule flottante pour garantir la compatibilité avec les types et les déclarations en ANSI-C. D ne prend pas en charge les opérateurs à virgule flottante mais la fonction `printf()` permet de suivre et de formater les objets de données à virgule flottante. Le tableau suivant présente les types à virgule flottante que vous pouvez utiliser :

TABLEAU 2–4 Types de données à virgule flottante en langage D

Nom du type	En 32–bits	En 64–bits
`float`	4 octets	4 octets
`double`	8 octets	8 octets
`long double`	16 octets	16 octets

D fournit également le type spécial `string` pour représenter les chaînes ASCII. Vous trouverez de plus amples informations sur les chaînes dans le Chapitre6, “Chaînes de caractères”.

Constantes

Vous pouvez écrire des constantes entières au format décimal (`12345`), octal (`012345`) ou hexadécimal (`0x12345`). Les constantes octales (base 8) doivent être précédées d'un zéro comme préfixe. Les constantes hexadécimales (base 16) doivent être précédées du préfixe `0x` ou `0X`. Est attribué aux constantes entières le plus petit type capable de représenter leur valeur entre `int`, `long` et `long long.` Si la valeur est négative, la version signée du type est utilisée. Si la valeur est positive et trop grande pour la représentation de type signé, la représentation de type non signé est utilisée. Vous pouvez appliquer l'un des suffixes suivants à toute constante entière pour spécifier son type D de manière explicite :

`u` ou `U`	Version `unsigned` du type sélectionné par le compilateur
`l` ou `L`	`long`
`ul` ou `UL`	`unsigned long`
`ll` ou `LL`	`long long`
`ull` ou `ULL`	`unsigned long long`

Les constantes à virgule flottante sont toujours rédigées avec une virgule décimale (`12,345`), un exposant (`123e45`) ou les deux (`123,34e-5`). Le type `double` est attribué par défaut aux constantes à virgule flottante. Vous pouvez appliquer l'un des suffixes suivants à toute constante à virgule flottante pour spécifier son type en langage D de manière explicite :

`f` ou `F`	`float`
`l` ou `L`	`long double`

Les constantes caractères sont écrites sous la forme d'un seul caractère ou d'une séquence d'échappement figurant entre guillemets simples (`'a'`). Le type `int` leur est attribué. Elles sont

équivalentes à une constante entière dont la valeur est déterminée par la valeur du caractère dans le jeu de caractères ASCII. Pour obtenir la liste des caractères et de leurs valeurs, reportez-vous à `ascii(5)`. Vous pouvez également utiliser l'une des séquences d'échappement spéciales présentées dans le tableau suivant dans vos constantes caractères. D prend en charge les mêmes séquences d'échappement détectées en ANSI-C.

TABLEAU 2–5 Séquences d'échappement de caractères en langage D

`\a`	alerte	`\\`	barre oblique inversée
`\b`	retour arrière	`\?`	point d'interrogation
`\f`	saut de page	`\'`	guillemet simple
`\n`	retour à la ligne	`\"`	guillemet double
`\r`	retour chariot	`\0`*oo*	valeur octale 0*oo*
`\t`	onglet horizontal	`\x`*hh*	valeur hexadécimale 0x*hh*
`\v`	onglet vertical	`\0`	caractère null

Vous pouvez ajouter plusieurs spécificateurs de caractère entre guillemets simples pour créer des nombres entiers dont chaque octet est initialisé en fonction des spécificateurs correspondants. Les octets sont lus de gauche à droite à partir de la constante caractère et affectés au nombre entier résultant dans l'ordre correspondant à l'endian-ness natif de votre système d'exploitation. Vous pouvez ajouter jusqu'à huit spécificateurs de caractère dans une seule constante caractère.

Vous pouvez créer des constantes chaînes à l'aide de guillemets doubles (`"hello"`). Une constante chaîne ne peut pas contenir de caractère littéral de retour à la ligne. Pour créer des chaînes contenant des retours à la ligne, utilisez la séquence d'échappement `\n` plutôt qu'un retour à la ligne littéral. Par contre, elles peuvent contenir n'importe laquelle des séquences d'échappement des caractères spéciaux récapitulés pour les constantes caractères ci-dessus. De même qu'en ANSI-C, les chaînes sont représentées sous la forme de tableaux de caractères terminés par un caractère null (`\0`) ajouté implicitement à chaque constante chaîne que vous déclarez. Le type D spécial `string` est affecté aux constantes chaînes. Le compilateur D fournit un ensemble de fonctions spéciales de comparaison et de suivi des tableaux de caractères déclarés en tant que chaînes, comme indiqué dans le Chapitre6, “Chaînes de caractères”.

Opérateurs arithmétiques

Vous pouvez utiliser dans vos programmes les opérateurs arithmétiques fournis par D présentés dans le tableau suivant. Ces opérateurs ont la même signification pour les nombres entiers qu'en ANSI-C.

TABLEAU 2–6 Opérateurs arithmétiques binaires en langage D

+	addition de nombres entiers
-	soustraction de nombres entiers
*	multiplication de nombres entiers
/	division de nombres entiers
%	pourcentage de nombres entiers

L'arithmétique en langage D ne peut être exécutée que sur des opérandes de nombre entier ou sur des pointeurs, comme indiqué dans le Chapitre5, “Pointeurs et ensembles”. Les programmes en langage D ne permettent pas d'exécuter des opérations arithmétiques sur des opérandes à virgule flottante. L'environnement d'exécution de DTrace n'exécute aucune action sur le dépassement de capacité supérieur ou inférieur des nombres entiers. Vous devez vérifier vous-même ces conditions dans les situations dans lesquelles un dépassement de capacité inférieur ou supérieur peut se produire.

L'environnement d'exécution de DTrace ne contrôle ni ne signale automatiquement les erreurs de division par zéro induites par une utilisation incorrecte des opérateurs / et %. Si un programme D exécute une opération de division invalide, DTrace désactive automatiquement l'instrumentation affectée et signale l'erreur. Les erreurs détectées par DTrace n'ont aucune incidence sur les autres utilisateurs de DTrace ou sur le noyau du système d'exploitation. Par conséquent, nul besoin de vous inquiéter si votre programme en D contient par inadvertance l'une de ces erreurs.

En plus de ces opérateurs binaires, vous pouvez également utiliser les opérateurs + et - comme opérateurs unaires ; ces opérateurs ont une priorité plus élevée que tout autre opérateur arithmétique binaire. L'ordre de priorité et les propriétés d'associativité pour tous les opérateurs en D sont présentés dans le Tableau 2–11. Vous pouvez contrôler la priorité en regroupant les expressions entre parenthèses ( ).

Opérateurs relationnels

Vous pouvez utiliser dans vos programmes les opérateurs relationnels fournis par D présentés dans le tableau suivant. Ces opérateurs ont la même signification qu'en ANSI-C.

TABLEAU 2–7 Opérateurs relationnels en langage D

`<`	opérande gauche inférieur à l'opérande droit
`<=`	opérande gauche inférieur ou égal à l'opérande droit
`>`	opérande gauche supérieur à l'opérande droit
`>=`	opérande gauche supérieur ou égal à l'opérande droit
`==`	opérande gauche égal à l'opérande droit
`!=`	opérande gauche différent de l'opérande droit

Les opérateurs relationnels sont généralement utilisés pour écrire des prédicats en langage D. Chaque opérateur évalue la valeur du type `int` qui est égale à 1 si la condition est vraie et à 0 si la condition est fausse.

Vous pouvez appliquer les opérateurs relationnels aux paires de nombres entiers, aux pointeurs ou aux chaînes. Si des pointeurs sont comparés, le résultat revient à comparer les nombres entiers de deux pointeurs interprétés comme des nombres entiers non signés. Si des chaînes sont comparées, le résultat est déterminé en l'état en exécutant `strcmp(3C)` sur les deux opérandes. Voici un exemple de comparaisons de chaînes en langage D et des résultats correspondants :

`"coffee" < "espresso"`	... renvoie 1 (vrai)
`"coffee" == "coffee"`	... renvoie 1 (vrai)
`"coffee" >= "mocha"`	... renvoie 0 (faux)

Vous pouvez également utiliser les opérateurs relationnels pour comparer des objets de données associés à un type d'énumération avec l'un des repères d'énumérateur définis par l'énumération. Les énumérations permettent de créer des constantes entières nommées et sont décrites plus en détails dans le Chapitre8, “Définitions des types et des constantes”.

Opérateurs logiques

Vous pouvez utiliser dans vos programmes les opérateurs logiques binaires disponibles en langage D suivants. Les deux premiers opérateurs sont équivalents aux opérateurs ANSI-C correspondants.

TABLEAU 2–8 Opérateurs logiques de D

`&&`	*AND* logique : vrai si les deux opérandes sont vrais
`||`	*OR* logique : vrai si l'un des des deux opérandes est vrai
`^^`	*XOR* logique : vrai si exactement un opérande est vrai

Les opérateurs logiques sont généralement utilisés pour écrire des prédicats en langage D. L'opérateur AND logique exécute une évaluation en court-circuit : Si l'opérande gauche est faux, l'expression droite n'est pas évaluée. L'opérateur OR logique exécute une évaluation en court-circuit : Si l'opérande gauche est vrai, l'expression droite n'est pas évaluée. L'opérateur XOR logique ne fait pas de court-circuit : les deux opérandes d'expression sont toujours évalués.

Outre les opérateurs logiques binaires, vous pouvez également utiliser l'opérateur unaire `!` pour exécuter une négation logique sur un seul opérande : ce dernier convertit un opérande 0 en un opérande 1 et un opérande non-zéro en opérande 0. Par convention, les programmeurs en langage D utilisent `!` avec les nombres entiers destinés à représenter des valeurs booléennes et `== 0` avec des nombres entiers non booléens même si ces deux expressions ont le même sens.

Vous pouvez appliquer les opérateurs logiques aux opérandes des types de nombre entier et de pointeur. Les opérateurs logiques interprètent les opérandes de pointeurs comme des valeurs entières non signées. De même qu'avec tous les opérateurs logiques et relationnels en langage D, les opérandes sont vrais s'ils possèdent une valeur entière non nulle et faux s'ils ont une valeur entière de zéro.

Opérateurs de bit

D propose les opérateurs binaires suivants pour manipuler les bits individuels dans les opérandes entiers. Ces opérateurs ont tous la même signification qu'en ANSI-C.

TABLEAU 2–9 Opérateurs de bit en langage D

`&`	Opérateur de bit AND
`|`	Opérateur de bit OR
`^`	Opérateur de bit XOR

TABLEAU 2–9 Opérateurs de bit en langage D (Suite)

<<	Permet de décaler l'opérande gauche vers la gauche du nombre de bits spécifié par l'opérande droit
>>	Permet de décaler l'opérande gauche vers la droite du nombre de bits spécifié par l'opérande droit

L'opérateur binaire & permet d'effacer les bits à partir d'un opérande entier. L'opérateur binaire | permet de définir des bits dans un opérande entier. L'opérateur binaire ^ renvoie 1 à chaque position de bit à laquelle précisément un des bits d'opérande correspondant est défini.

Les décalages d'opérateurs permettent de déplacer des bits vers la gauche ou la droite dans un opérande entier donné. Le décalage à gauche remplit les emplacements de bit vides sur le côté droit du résultat avec des zéros. Le décalage à droite à l'aide d'un opérande entier non signé remplit les emplacements de bit vides à gauche du résultat avec des zéros. Le décalage à droite à l'aide d'un opérande entier signé remplit les emplacements de bit vides à gauche de la valeur du bit de signe. Cette opération s'appelle également un *décalage arithmétique*.

Le décalage d'une valeur entière par un nombre de bits négatif ou plus grand que le nombre de bits contenu dans l'opérande gauche lui-même crée un résultat non défini. Le compilateur D produit un message d'erreur s'il peut détecter cette condition lors de la compilation de votre programme en D.

Outre les opérateurs logiques binaires, vous pouvez également utiliser l'opérateur unaire ! pour exécuter une négation de bit sur un seul opérande : il convertit chaque bit 0 de l'opérande en bit 1 et chaque bit 1 de l'opérande en bit 0.

Opérateurs d'assignation

D fournit les opérateur d'assignation suivants pour modifier les variables en langage D. Vous ne pouvez modifier que les variables et les tableaux en langage D. Vous ne pouvez pas modifier les constantes et les objets de données du noyau à l'aide des opérateurs d'assignation en langage D. Les opérateurs d'assignation ont la même signification qu'en ANSI-C.

TABLEAU 2–10 Opérateurs d'assignation en langage D

=	permet d'indiquer que l'opérande gauche est égal à la valeur de l'expression droite
+=	Permet d'incrémenter l'opérande gauche de la valeur de l'expression droite
-=	Permet de décrémenter l'opérande gauche de la valeur de l'expression droite
*=	Permet de multiplier l'opérande gauche par la valeur de l'expression droite
/=	Permet de diviser l'opérande gauche par la valeur de l'expression droite

TABLEAU 2–10 Opérateurs d'assignation en langage D *(Suite)*

%=	Permet d'effectuer le modulo de l'opérande gauche par la valeur de l'expression droite
|=	Permet d'appliquer l'opérateur de bit OR à l'opérande de gauche avec la valeur de l'expression droite
&=	Permet d'appliquer l'opérateur de bit AND à l'opérande de gauche avec la valeur de l'expression droite
^=	Permet d'appliquer l'opérateur de bit XOR à l'opérande de gauche avec la valeur de l'expression droite
<<=	Permet de décaler l'opérande gauche vers la gauche du nombre de bits spécifié par la valeur de l'expression droite
>>=	Permet de décaler l'opérande gauche vers la droite du nombre de bits spécifié par la valeur de l'expression droite

Les opérateurs d'assignation, hormis =, sont fournis comme abrégés, afin d'utiliser l'opérateur = avec l'un des autres opérateurs décrits précédemment. Par exemple, l'expression `x = x + 1` équivaut à l'expression `x += 1` si ce n'est que l'expression `x` est déterminée une seule fois. Ces opérateurs d'assignation obéissent aux mêmes règles pour les types d'opérande que les formes binaires décrites précédemment.

Le résultat d'un opérateur d'assignation correspond à une expression équivalant à la nouvelle valeur de l'expression gauche. Vous pouvez associer les opérateurs d'assignation ou tout autre opérateur décrit dans ce document pour former des expressions d'une complexité arbitraire. Vous pouvez utiliser des parenthèses `( )` pour regrouper des termes dans des expressions complexes.

Opérateurs d'incrément et de décrément

D propose les opérateurs unaires spéciaux ++ et `--` pour incrémenter et décrémenter les pointeurs et les nombres entiers. Ces opérateurs ont la même signification qu'en ANSI-C. Vous ne pouvez les appliquer qu'aux variables, avant ou après le nom de la variable. Si l'opérateur apparaît avant le nom de la variable, la variable est modifiée en premier, puis l'expression qui en résulte équivaut à la nouvelle valeur de la variable. Par exemple, les deux expressions suivantes produisent des résultats identiques :

`x += 1;`	`y = ++x;`
`y = x;`	

Si l'opérateur apparaît après le nom de la variable, cette dernière est modifiée après renvoi de sa valeur actuelle, afin de l'utiliser dans l'expression. Par exemple, les deux expressions suivantes produisent des résultats identiques :

`y = x;`	`y = x--;`
`x -= 1;`	

Vous pouvez utiliser les opérateurs d'incrément et de décrément pour créer de nouvelles variables sans les déclarer. Si la déclaration d'une variable est omise alors que l'opérateur d'incrément ou de décrément est appliqué à une variable, la variable est implicitement déclarée de type `int64_t`.

Vous pouvez appliquer les opérateurs d'incrément et de décrément aux variables de nombres entiers, auquel cas les opérateurs incrémentent ou décrémentent la valeur correspondant de 1, ou aux variables de pointeurs, auquel cas les opérateurs incrémentent ou décrémentent l'adresse du pointeur de la taille du type de données référencé par le pointeur. Les pointeurs et l'arithmétique de pointeur en D sont présentés dans le Chapitre5, “Pointeurs et ensembles”.

Expressions conditionnelles

Bien que le langage D ne prenne pas en charge de constructions if-then-else, il supporte les expressions conditionnelles simples employant les opérateurs `?` et `:`. Ces opérateurs permettent d'associer un triplet d'expressions, la première expression étant utilisée pour déterminer de manière conditionnelle l'une des deux autres. Vous pourriez, par exemple, utiliser l'énoncé en langage D suivant pour définir une variable `x` sur l'une des deux chaînes en fonction de la valeur de `i` :

```
x = i == 0 ? "zero" : "non-zero";
```

Dans cet exemple, l'expression `i == 0` est évaluée en premier pour déterminer si elle est vraie ou fausse. Si la première expression est vraie, la seconde est évaluée et l'expression `?:` renvoie sa valeur. Si la première expression est fausse, la troisième est évaluée et l'expression `?:` renvoie sa valeur.

Comme pour tout opérateur en D, vous pouvez utiliser plusieurs opérateurs `?:` dans une seule expression, de manière à créer plus d'expressions complexes. Par exemple, l'expression suivante devra prendre une variable `char c` contenant l'un des caractères 0-9, a-z ou A-Z et renvoyer la valeur de ce caractère une fois interprété comme un chiffre dans un nombre entier hexadécimal (base 16) :

```
hexval = (c >= ’0’ && c <= ’9’) ? c - ’0’ :
    (c >= ’a’ && c <= ’z’) ? c + 10 - ’a’ : c + 10 - ’A’;
```

La première expression utilisée avec `?:` doit être un pointeur ou un nombre entier pour pouvoir déterminer si sa valeur est vraie. La seconde et la troisième expression peuvent appartenir à n'importe quel type compatible. Vous ne pouvez pas construire d'expression conditionnelle lorsque, par exemple, un chemin d'accès renvoie une chaîne alors qu'un autre chemin d'accès

renvoie un nombre entier. La seconde et la troisième expression ne peuvent pas non plus invoquer de fonction de suivi comme `trace()` ou `printf()`. Si vous souhaitez assurer le suivi conditionnel des données, utilisez plutôt un prédicat, comme indiqué dans le Chapitre1, “Introduction”.

Conversions de types

Lorsque des expressions sont construites à l'aide d'opérandes de types différents mais compatibles, les types sont convertis de manière à déterminer celui de l'expression résultante. Les règles en langage D concernant les conversions de types sont identiques aux règles de conversion arithmétique des nombres entiers en ANSI-C. Ces règles sont parfois appelées *conversions arithmétiques usuelles*.

Il est possible de décrire tout simplement les règles de conversion comme suit : Chaque type d'entier est classé dans l'ordre `char`, `short`, `int`, `long`, `long long`, sachant que les types correspondants non signés apparaissent au-dessus de leur équivalent signé mais en dessous du type d'entier suivant. Lorsque vous construisez une expression à l'aide de deux opérandes entiers comme `x + y` et que ces opérandes appartiennent à un type d'entier différent, le type d'opérande le plus haut classé sert de type de résultat.

Si une conversion s'impose, l'opérande de niveau inférieur est le premier *promu* au type de niveau supérieur. La promotion ne modifie pas vraiment la valeur de l'opérande : elle étend seulement la valeur vers un conteneur plus grand en fonction de son signe. Si un opérande non signé est promu, les bits supérieurs non utilisés du nombre entier résultant sont remplis avec des 0. Si un opérande signé est promu, les bits supérieurs non utilisés sont remplis par l'extension du signe d'exécution. Si un type signé est converti en type non signé, le signe du type signé est d'abord étendu, puis le nouveau type non signé, déterminé par la conversion, lui est affecté.

Les nombres entiers et les autres types peuvent également être *convertis* explicitement d'un type en un autre. En langage D, vous pouvez convertir les pointeurs et les nombres entiers en n'importe quel type de pointeur ou de nombre entier mais pas en un autre type. Les règles de conversion et de promotion des chaînes et des tableaux de caractères sont traitées dans le Chapitre6, “Chaînes de caractères”. La conversion d'un entier ou d'un pointeur est créée à l'aide d'une expression telle que :

```
y = (int)x;
```

le type de destination étant placé entre parenthèses et servant de préfixe à l'expression de la source. Les nombres entiers sont convertis en des types de niveau supérieur par l'intermédiaire d'une promotion. Ils sont convertis en types de niveau inférieur en transformant en zéro les bits supérieurs superflus du nombre entier.

Comme D n'autorise pas l'arithmétique en virgule flottante, aucune conversion d'opérande en virgule flottante n'est autorisée et aucune règle de conversion en virgule flottante implicite n'est définie.

Priorité

Les règles en langage D concernant la priorité et l'associativité des opérateurs sont décrites dans le tableau suivant. Ces règles sont quelque peu complexes mais nécessaires pour assurer une compatibilité précise avec les règles de priorité des opérateurs ANSI-C. Les entrées du tableau sont triées par ordre de priorité, de la plus élevée à la plus faible.

TABLEAU 2–11 Priorité et associativité des opérateurs de D

<table>
<tr><th>Opérateurs</th><th>Associativité</th></tr>
<tr><td>() [] -> .</td><td>gauche à droite</td></tr>
<tr><td>! ~ ++ -- + - * & (type) sizeof stringof offsetof xlate</td><td>droite à gauche</td></tr>
<tr><td>* / %</td><td>gauche à droite</td></tr>
<tr><td>+ -</td><td>gauche à droite</td></tr>
<tr><td><< >></td><td>gauche à droite</td></tr>
<tr><td>< <= > >=</td><td>gauche à droite</td></tr>
<tr><td>== !=</td><td>gauche à droite</td></tr>
<tr><td>&</td><td>gauche à droite</td></tr>
<tr><td>^</td><td>gauche à droite</td></tr>
<tr><td>|</td><td>gauche à droite</td></tr>
<tr><td>&&</td><td>gauche à droite</td></tr>
<tr><td>^^</td><td>gauche à droite</td></tr>
<tr><td>||</td><td>gauche à droite</td></tr>
<tr><td>?:</td><td>droite à gauche</td></tr>
<tr><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>droite à gauche</td></tr>
<tr><td>,</td><td>gauche à droite</td></tr>
</table>

Le tableau comprend plusieurs opérateurs qui n'ont pas encore été présentés ; ils seront traités dans des chapitres ultérieurs :

`sizeof`	Calcule la taille d'un objet (Chapitre7, “Structs et Unions”)
`offsetof`	Calcule le décalage d'un membre type (Chapitre7, “Structs et Unions”)
`stringof`	Convertit l'opérande en chaîne (Chapitre6, “Chaînes de caractères”)
`xlate`	Convertit un type de données (Chapitre40, “Translateurs”)
`&` unaire	Calcule l'adresse d'un objet (Chapitre5, “Pointeurs et ensembles”)
`*` unaire	Déréférence un pointeur sur un objet (Chapitre5, “Pointeurs et ensembles”)
`->` et `.`	Accède à un membre d'une structure ou d'un type d'union (Chapitre7, “Structs et Unions”)

L'opérateur `,` (virgule) présenté dans le tableau est dédié à la compatibilité avec l'opérateur ANSI-C correspondant et permet notamment d'évaluer un ensemble d'expressions de gauche à droite et de retourner la valeur de l'expression de poids faible. Cet opérateur n'est fourni que pour la compatibilité avec le langage C et ne doit généralement pas être utilisé.

L'entrée `()` du tableau sur la priorité des opérateurs représente un appel de fonction. Des exemples d'appel de fonction, comme `printf()` et `trace()`, sont présentés dans le Chapitre1, “Introduction”. La virgule sert également en langage D à énumérer des arguments en fonctions et à constituer des listes de clés de tableaux associatifs. Cette virgule n'a rien à voir avec l'opérateur et *n'*assure aucune évaluation de gauche à droite. Le compilateur D ne garantit en rien l'ordre d'évaluation des arguments en fonction ou des clés en tableau associatif. Vous devez veiller à utiliser des expressions aux effets obliques interagissants, comme la paire d'expressions `i` et `i++`, dans ces contextes.

L'entrée `[]` du tableau de priorité des opérateurs représente un tableau ou une référence de tableau associatif. Des exemples de tableaux associatifs sont présentés dans le Chapitre1, “Introduction”. Un type spécial de tableau associatif, appelé *groupement*, est décrit dans le Chapitre9, “Groupements”. Vous pouvez également utiliser l'opérateur `[]` pour créer des index dans des tableaux C au format fixe, comme indiqué dans le Chapitre5, “Pointeurs et ensembles”.

CHAPITRE 3

Variables

Le langage D fournit deux types de variable à utiliser dans vos programmes de suivi : les variables scalaires et les tableaux associatifs. Les exemples du chapitre 1 illustraient brièvement l'utilisation de ces variables. Ce chapitre présente de façon approfondie les règles relatives aux variables en D et explique comment associer ces variables à diverses étendues. Un type spécial de variable de tableau, appelé *groupement*, est présenté dans le Chapitre9, “Groupements”.

Variables scalaires

Les variables scalaires servent à représenter les objets de données individuels de taille fixe, comme les nombres entiers et les pointeurs. Vous pouvez également utiliser des variables scalaires pour les objets à taille fixe constitués d'un ou de plusieurs types de primitive ou composites. Le langage D permet de créer des tableaux et des objets, ainsi que des structures composites. DTrace représente aussi des chaînes sous forme de données scalaires de taille fixe en leur permettant de grossir jusqu'à une longueur maximale prédéfinie. Le contrôle de la longueur des chaînes dans votre programme en D est abordé plus en détails dans le Chapitre6, “Chaînes de caractères”.

Les variables scalaires sont créées automatiquement la première fois que vous attribuez une valeur à un identificateur précédemment non défini dans votre programme en D. Par exemple, pour créer la variable scalaire `x` de type `int`, vous pouvez simplement affecter une valeur de type `int` dans la clause d'une sonde :

```
BEGIN
{
    x = 123;
}
```

Les variables scalaires créées de cette manière sont des variables *globales* : leur nom et leurs données de stockage sont définis une fois et sont visibles dans chaque clause de votre programme en D. Chaque fois que vous référencez l'identificateur `x`, vous faites référence à un emplacement de stockage unique lié à cette variable.

Contrairement au langage ANSI-C, le langage D ne demande pas de déclarations de variables explicites. Si vous souhaitez déclarer une variable globale pour affecter son nom et son type explicitement avant de l'utiliser, vous pouvez placer une déclaration à l'extérieur des clauses de la sonde dans votre programme, comme illustré dans l'exemple suivant. Les déclarations de variables explicites ne sont pas requises dans la plupart des programmes en D mais peuvent s'avérer utiles lorsque vous souhaitez contrôler soigneusement vos types de variable ou lorsque vous souhaitez commencer votre programme par un ensemble de déclarations et de commentaires documentant les variables de votre programme et leur signification.

```
int x; /* declare an integer x for later use */

BEGIN
{
    x = 123;
    ...
}
```

Contrairement au langage ANSI-C, la déclaration de variables en langage D peut ne pas affecter de valeurs initiales. Vous devez utiliser une clause de sonde `BEGIN` pour affecter n'importe quelle valeur initiale. Le stockage de variable globale est rempli de zéros par DTrace avant que vous ne référenciez la variable.

La définition du langage D n'impose aucune limite de taille ni de nombre de variables en D. Par contre, une limite est définie par l'implémentation de DTrace et par la mémoire disponible sur votre système. Le compilateur D mettra en œuvre toutes les restrictions qu'il est possible d'appliquer au moment de la compilation de votre programme. Vous découvrirez de plus amples informations sur l'ajustement des options en rapport avec les limites du programme dans le Chapitre16, “Options et paramètres réglables”.

Tableaux associatifs

Les tableaux associatifs servent à représenter les ensembles d'éléments de données que vous pouvez récupérer en spécifiant un nom appelé une *clé*. Les clés de tableau associatif en D sont constituées de valeurs d'expression scalaires appelées un *tuple*. Vous pouvez imaginer le tuple de tableau lui-même comme une liste de paramètres imaginaires vers une fonction appelée pour récupérer la valeur de tableau correspondante lors du référencement du tableau. Chaque tableau associatif en D comporte une *signature de clé* fixe consistant en un nombre fixe d'éléments de tuple, chaque élément ayant un type fixe donné. Vous pouvez définir différentes signatures de clé par tableau dans votre programme en D.

Les tableaux associatifs diffèrent des tableaux normaux de taille fixe en ce qu'ils n'ont aucune limite prédéfinie en matière de nombre d'éléments. Les éléments peuvent être indexés par un tuple par opposition à la simple utilisation de nombres entiers comme clés. Par ailleurs, les éléments ne sont pas enregistrés dans des emplacements de stockage consécutifs pré-alloués.

Les tableaux associatifs sont utiles dans les situations dans lesquelles vous devez utiliser une table de hachage ou une autre structure de données de dictionnaire simple dans un programme en langage C, C++ ou Java. ™ Les tableaux associatifs vous permettent de créer un historique dynamique des événements et de l'état capturé dans votre programme en D que vous pouvez utiliser pour créer des flux de contrôle plus complexes.

Pour définir un tableau associatif, vous écrivez une expression d'affectation du formulaire :

name [*key*] = *expression* ;

name correspond à un identificateur D valide et *key* à une liste séparée par des virgules d'une ou plusieurs expressions. Par exemple, l'instruction suivante définit un tableau associatif `a` avec la signature de clé `[ int, string ]` et enregistre la valeur entière 456 à l'emplacement nommé par le tuple `[ 123, "hello" ]`:

```
a[123, "hello"] = 456;
```

Le type de chaque objet contenu dans le tableau est également fixe pour tous les éléments d'un tableau donné. Comme `a` a été affecté en premier à l'aide du nombre entier 456, chaque valeur ultérieure enregistrée dans le tableau aura également comme type `int`. Vous pouvez utiliser les opérateurs d'affectation définis dans le chapitre 2 pour modifier les éléments de tableau associatif en fonction des règles d'opérande définies par l'opérateur. Le compilateur D crée un message d'erreur approprié si vous tentez une affectation incompatible. Vous pouvez utiliser les mêmes types avec une clé ou une valeur de tableau associatif que ceux utilisés avec une variable scalaire. Vous ne pouvez pas imbriquer un tableau associatif dans un autre sous forme de clé ou de valeur.

Vous pouvez référencer un tableau associatif à l'aide d'un tuple compatible avec la signature de la clé du tableau. Les règles de compatibilité du tuple sont similaires à celles des appels de fonction et des affectations de variable : La longueur du tuple doit être la même et chaque type de la liste de paramètres réels doit être compatible avec le type correspondant dans la signature de clé formelle. Par exemple, si un tableau associatif `x` est défini comme suit :

```
x[123ull] = 0;
```

la signature de clé est de type `unsigned long long` et les valeurs sont de type `int`. Vous pouvez également référencer ce tableau à l'aide de l'expression `x['a']`, car le tuple consistant en une constante caractère `'a'` de type `int` et en une longueur de 1 est compatible avec la signature de clé `unsigned long long` conformément aux règles de conversion arithmétique décrites à la section “Conversions de types” à la page 60.

Si vous devez déclarer de manière explicite un tableau associatif en D avant de l'utiliser, vous pouvez créer une déclaration du nom du tableau et de la signature de clé en dehors des clauses de la sonde dans le code source de votre programme :

```
int x[unsigned long long, char];
```

```
BEGIN
{
    x[123ull, 'a'] = 456;
}
```

Une fois un tableau associatif défini, les références à un tuple d'une signature de clé compatible sont autorisées même si le tuple en question n'a pas été affecté précédemment. L'accès à un élément de tableau associatif non affecté est défini pour revenir sur un objet rempli de zéros. Cette définition a notamment pour conséquence que le stockage sous-jacent n'est pas alloué à un élément de tableau associatif tant qu'une valeur différente de zéro est affectée à cet élément. À l'inverse, l'affectation de la valeur zéro à cet élément de tableau associatif pousse DTrace à supprimer l'allocation de stockage sous-jacent. Ce comportement est important car l'espace d'adressage dynamique en dehors duquel les éléments du tableau sont alloués est fini ; si cet espace est épuisé lors d'une tentative d'allocation, cette dernière échoue et un message d'erreur est généré indiquant un abandon de variable dynamique. Affectez toujours des zéros aux éléments de tableaux associatifs qui ne sont plus utilisés. Pour connaître d'autres techniques visant à supprimer les abandons de variables dynamiques, reportez-vous au Chapitre16, “Options et paramètres réglables”.

Variables locales de thread

DTrace permet de déclarer le stockage des variables locales sur chaque thread du système d'exploitation, par opposition aux variables globales comme indiqué précédemment dans ce chapitre. Les variables locales de thread sont utiles lorsque vous souhaitez activer une sonde et marquer chaque thread qui déclenche la sonde avec une balise ou d'autres données. Il est aisé de créer un programme pour résoudre ce problème en langage D car les variables locales de thread partagent un nom commun dans votre code en D mais font référence à un stockage de données distinct associé à chaque thread. Les variables locales de thread sont référencées en appliquant l'opérateur `->` à un identificateur `self` spécial :

```
syscall::read:entry
{
    self->read = 1;
}
```

Cet exemple en langage D active la sonde sur l'appel système `read(2)` et associe la variable locale de thread `read` à chaque thread qui déclenche la sonde. De la même manière que les variables globales, les variables locales de thread sont créées automatiquement sur leur première affectation et adoptent le type utilisé à droite de la première instruction d'affectation (dans cet exemple, `int`).

Chaque fois que la variable `self->read` est référencée dans votre programme en D, l'objet de données référencé correspond à l'objet associé au thread du système d'exploitation en cours d'exécution au déclenchement de la sonde de DTrace correspondante. Vous pouvez imaginer une variable locale de thread comme un tableau associatif implicitement indexé par un tuple qui

décrit l'identité du thread dans le système. L'identité d'un thread est unique tout au long de la durée de vie du système : si le thread est fermé et que la même structure de données du système d'exploitation est utilisée pour créer un nouveau thread, ce dernier *ne* réutilise pas la même identité de stockage local de thread de DTrace.

Une fois que vous avez défini une variable locale de thread, vous pouvez la déréférencer pour n'importe quel thread dans le système même si la variable en question n'a pas été précédemment affectée à ce thread particulier. Si la copie d'un thread d'une variable locale de thread n'a pas encore été affectée, le stockage des données dédié à la copie est défini pour être rempli de zéros. De même qu'avec les éléments de tableau associatif, le stockage sous-jacent n'est pas alloué à une variable locale de thread tant qu'une valeur différente de zéro lui est affectée. De la même manière, l'affectation de zéro à une variable locale de thread entraîne la suppression de l'allocation de stockage sous-jacent au niveau de DTrace. Affectez toujours zéro aux variables locales de thread que vous n'utilisez plus. Pour découvrir les autres techniques de réglage de l'espace des variables dynamiques à partir duquel les variables locales de thread sont allouées, reportez-vous au Chapitre16, “Options et paramètres réglables”.

Vous pouvez définir les variables locales de thread, quel que soit leur type, dans vos programmes en D, y compris les tableaux associatifs. Voici quelques exemples de définitions de variables locales de thread :

```
self->x = 123;              /* integer value */
self->s = "hello";             /* string value */
self->a[123, ’a’] = 456;    /* associative array */
```

De même que les variables en langage D, vous ne pouvez pas déclarer explicitement les variables locales de thread avant de les utiliser. Si vous souhaitez malgré tout créer une déclaration, vous pouvez en placer une en dehors des clauses de votre programme en ajoutant initialement le mot-clé `self` :

```
self int x;    /* declare int x as a thread-local variable */

syscall::read:entry
{
    self->x = 123;
}
```

Les variables locales de thread sont conservées dans un espace de noms distinct des variables globales. Vous pouvez donc réutiliser les noms. N'oubliez pas que `x` et `self->x` sont deux variables différentes si vous chargez des noms dans votre programme ! L'exemple suivant présente la méthode d'utilisation des variables locales de thread. Dans un éditeur de texte, entrez le programme suivant et enregistrez-le dans le fichier `rtime.d` :

EXEMPLE 3–1 `rtime.d` : calcul du temps passé dans `read`(2)

```
syscall::read:entry
{
```

EXEMPLE 3–1 `rtime.d` : calcul du temps passé dans `read`(2) *(Suite)*

```
    self->t = timestamp;
}

syscall::read:return
/self->t != 0/
{
    printf("%d/%d spent %d nsecs in read(2)\n",
        pid, tid, timestamp - self->t);

    /*
     * We're done with this thread-local variable; assign zero to it to
     * allow the DTrace runtime to reclaim the underlying storage.
     */
    self->t = 0;
}
```

Accédez à présent au shell et lancez le programme. Patientez quelques secondes avant de voir apparaître une sortie. En l'absence de sortie, essayez d'exécuter quelques commandes.

```
# dtrace -q -s rtime.d
100480/1 spent 11898 nsecs in read(2)
100441/1 spent 6742 nsecs in read(2)
100480/1 spent 4619 nsecs in read(2)
100452/1 spent 19560 nsecs in read(2)
100452/1 spent 3648 nsecs in read(2)
100441/1 spent 6645 nsecs in read(2)
100452/1 spent 5168 nsecs in read(2)
100452/1 spent 20329 nsecs in read(2)
100452/1 spent 3596 nsecs in read(2)
...
^C
#
```

`rtime.d` utilise la variable locale de thread `t` pour capturer un horodatage à l'entrée de `read(2)` par n'importe quel thread. Dans la clause de retour, le programme imprime ensuite le temps passé dans `read(2)` en retirant `self->t` de l'horodatage actuel. Les variables en D intégrées `pid` et `tid` indiquent l'ID de processus et l'ID de thread exécutant `read(2)`. Comme `self->t` n'est plus requis une fois ces informations reportées, la valeur 0 lui est affectée pour permettre à DTrace de réutiliser le stockage sous-jacent associé à `t` pour le thread actuel.

En règle générale, vous voyez s'afficher de nombreuses lignes de sortie sans rien faire car, en coulisses, les processus serveur et les démons exécutent `read(2)` en permanence et ce, même lorsque vous ne faites rien. Essayez de modifier la seconde clause de `rtime.d` afin d'utiliser la variable `execname` servant à imprimer le nom du processus exécutant `read(2)` dans le but d'en apprendre davantage à son sujet :

```
printf("%s/%d spent %d nsecs in read(2)\n",
    execname, tid, timestamp - self->t);
```

Si vous découvrez un processus particulièrement intéressant, ajoutez un prédicat pour en découvrir davantage sur son comportement read(2) :

```
syscall::read:entry
/execname == "Xsun"/
{
    self->t = timestamp;
}
```

Variables locales de clause

Vous pouvez également définir les variables en D dont le stockage est réutilisé pour chaque clause du programme en D. Les variables locales de clause sont similaires aux variables automatiques des programmes en langage C, C++ ou Java actifs pendant chaque invocation d'une fonction. Comme toutes les variables de programme en D, les variables locales de clause sont créées à leur première affectation. Vous pouvez référencer et affecter ces variables en appliquant l'opérateur -> à l'identificateur spécial this :

```
BEGIN
{
    this->secs = timestamp / 1000000000;
    ...
}
```

Si vous souhaitez déclarer de manière explicite une variable locale de clause avant de l'utiliser, vous pouvez le faire à l'aide du mot-clé this :

```
this int x;   /* an integer clause-local variable */
this char c;  /* a character clause-local variable */

BEGIN
{
    this->x = 123;
    this->c = 'D';
}
```

Les variables locales de clause ne sont actives que pendant la durée de vie d'une clause de sonde donnée. Une fois que DTrace a exécuté les actions liées à ces clauses pour une sonde donnée, le stockage de toutes les variables locales de clause est récupéré et réutilisé pour la clause suivante. C'est pourquoi les variables locales de clause constituent les seules variables en D à ne pas contenir initialement que des zéros. Notez que si votre programme contient plusieurs clauses pour une seule sonde, toutes les variables locales de clause resteront intactes lors de l'exécution des clauses, comme illustré dans l'exemple suivant :

EXEMPLE 3–2 `clause.d` : variables locales de clause

```
int me;            /* an integer global variable */
this int foo;        /* an integer clause-local variable */

tick-1sec
{
    /*
     * Set foo to be 10 if and only if this is the first clause executed.
     */
    this->foo = (me % 3 == 0) ? 10 : this->foo;
    printf("Clause 1 is number %d; foo is %d\n", me++ % 3, this->foo++);
}

tick-1sec
{
    /*
     * Set foo to be 20 if and only if this is the first clause executed.
     */
    this->foo = (me % 3 == 0) ? 20 : this->foo;
    printf("Clause 2 is number %d; foo is %d\n", me++ % 3, this->foo++);
}

tick-1sec
{
    /*
     * Set foo to be 30 if and only if this is the first clause executed.
     */
    this->foo = (me % 3 == 0) ? 30 : this->foo;
    printf("Clause 3 is number %d; foo is %d\n", me++ % 3, this->foo++);
}
```

Comme les clauses sont *toujours* exécutées dans l'ordre du programme et que les variables locales de clause sont persistantes à travers les différentes clauses activant la même sonde, l'exécution du programme ci-dessus entraîne toujours la même sortie :

```
# dtrace -q -s clause.d
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
Clause 1 is number 0; foo is 10
```

```
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
^C
```

Alors que les variables locales de clause sont persistantes à travers les clauses activant la même sonde, leurs valeurs restent indéfinies dans la première clause exécutée pour une sonde donnée. Veillez à affecter chaque variable locale de clause à une valeur appropriée avant de l'utiliser ou votre programme risque de produire un résultat inattendu.

Vous pouvez définir les variables locales de clause à l'aide de n'importe quel type de variable scalaire mais pas les tableaux associatifs au moyen d'une étendue locale de clause. L'étendue des variables locales de clause ne s'applique qu'aux données de variable correspondantes et non à l'identité du nom et du type définie dans la variable. Après avoir défini une variable locale de clause, vous pouvez utiliser le nom et la signature de type correspondants dans n'importe quelle clause de programme en D ultérieure. Vous ne pouvez pas espérer que l'emplacement de stockage soit le même entre les différentes clauses.

Vous pouvez utiliser des variables locales de clause pour accumuler des résultats intermédiaires de calculs ou comme copies temporaires d'autres variables. Il est plus rapide d'accéder à une variable locale de clause qu'à un tableau associatif. Par conséquent, si vous devez référencer à plusieurs reprises une valeur de tableau associatif dans la même clause d'un programme en D, vous gagnerez en efficacité en la copiant dans une variable locale de clause, puis en référençant cette dernière à plusieurs reprises.

Variables intégrées

La table suivante contient la liste complète des variables en D intégrées. Toutes ces variables sont des variables globales scalaires ; les variables locales de thread ou de clause ou les tableaux associatifs ne sont pas définis en langage D pour le moment.

TABLEAU 3–1 Variables intégrées de DTrace

Type et nom	Description
`int64_t arg0, ..., arg9`	Dix premiers arguments d'entrée vers une sonde représentés comme des nombres entiers bruts de 64 bits. Si moins de dix arguments sont transmis à la sonde actuelle, les variables restantes retournent à zéro.

TABLEAU 3–1 Variables intégrées de DTrace (Suite)

Type et nom	Description
`args[]`	Arguments type de la sonde actuelle, le cas échéant. Le tableau `args[]` est accessible au moyen d'un index de nombres entiers mais chaque élément est défini pour représenter le type correspondant à l'argument de sonde donné. Par exemple, si `args[]` est référencé par une sonde d'appel système `read(2)`, `args[0]` appartient au type `int`, `args[1]` au type `void *` et `args[2]` au type `size_t`.
`uintptr_t caller`	Emplacement du compteur du programme du thread actuel juste avant d'entrer dans la sonde actuelle.
`chipid_t chip`	Identificateur de puce de CPU de la puce physique actuelle. Pour plus d'informations, reportez-vous au Chapitre26, “Fournisseur `sched`”.
`processorid_t cpu`	Identificateur de la CPU actuelle. Pour plus d'informations, reportez-vous au Chapitre26, “Fournisseur `sched`”.
`cpuinfo_t *curcpu`	Informations sur la CPU actuelle. Pour plus d'informations, reportez-vous au Chapitre26, “Fournisseur `sched`”.
`lwpsinfo_t *curlwpsinfo`	État du processus léger (LWP) de LWP associé au thread actuel. Cette structure est décrite plus en détails dans la page de manuel `proc(4)`.
`psinfo_t *curpsinfo`	État du processus lié au thread actuel. Cette structure est décrite plus en détails dans la page de manuel `proc(4)`.
`kthread_t *curthread`	Adresse de la structure des données internes du noyau du système d'exploitation du thread actuel, `kthread_t`. `kthread_t` est défini dans `<sys/thread.h>`. Reportez-vous aux *Données internes sur Solaris* pour plus d'informations sur cette variable et les autres structures de données du système d'exploitation.
`string cwd`	Nom du répertoire de travail actuel du processus associé au thread actuel.
`uint_t epid`	ID de sonde activée (EPID) de la sonde actuelle. Ce nombre entier n'identifie qu'une sonde particulière qui est activée par un prédicat et un ensemble d'actions spécifiques.

TABLEAU 3–1 Variables intégrées de DTrace *(Suite)*

Type et nom	Description
`int errno`	Valeur d'erreur retournée par le dernier appel système exécuté par ce thread.
`string execname`	Nom transmis à `exec(2)` pour exécuter le processus actuel.
`gid_t gid`	ID de groupe réel du processus actuel.
`uint_t id`	ID de sonde de la sonde actuelle. Cet ID constitue l'identificateur unique à l'échelle du système de la sonde tel qu'émis par DTrace et répertorié dans la sortie de `dtrace -l`.
`uint_t ipl`	Niveau de priorité d'interruption (IPL) sur la CPU actuelle au moment du déclenchement de la sonde. Reportez-vous aux *Données internes sur Solaris* pour plus d'informations sur les niveaux d'interruption et la gestion des interruptions dans le noyau du système d'exploitation de Solaris.
`lgrp_id_t lgrp`	ID du groupe de latence auquel la CPU actuelle appartient. Pour plus d'informations, reportez-vous au Chapitre26, "Fournisseur `sched`".
`pid_t pid`	ID de processus du processus actuel.
`pid_t ppid`	ID de processus parent du processus actuel.
`string probefunc`	Partie du nom de fonction de la description de la sonde actuelle.
`string probemod`	Partie du nom de module de la description de la sonde actuelle.
`string probename`	Partie du nom de la description de la sonde actuelle.
`string probeprov`	Partie du nom du fournisseur de la description de la sonde actuelle.
`psetid_t pset`	ID de l'ensemble de processeurs contenant la CPU actuelle. Pour plus d'informations, reportez-vous au Chapitre26, "Fournisseur `sched`".
`string root`	Nom du répertoire root du processus associé au thread actuel.
`uint_t stackdepth`	Profondeur du cadre de la pile du thread actuel au moment du déclenchement de la sonde.

TABLEAU 3–1 Variables intégrées de DTrace *(Suite)*

Type et nom	Description
`id_t tid`	ID du thread actuel. Pour les threads liés aux processus utilisateur, cette valeur équivaut au résultat d'un appel vers `pthread_self`(3C).
`uint64_t timestamp`	Valeur actuelle d'un compteur d'horodatage en nanosecondes. Ce compteur procède à l'incrémentation à partir d'un point arbitraire antérieur et ne doit être utilisé que pour la réalisation de calculs relatifs.
`uid_t uid`	ID de l'utilisateur réel du processus actuel.
`uint64_t uregs[]`	Valeurs d'enregistrement en mode utilisateur de sauvegarde du thread actuel au moment du déclenchement de la sonde. L'utilisation du tableau `uregs[]` est présentée dans le Chapitre33, "Suivi des processus utilisateur".
`uint64_t vtimestamp`	Valeur actuelle d'un compteur d'horodatage en nanosecondes, virtualisée par la durée d'exécution du thread actuel sur la CPU moins le temps passé dans les prédicats et actions de DTrace. Ce compteur procède à l'incrémentation à partir d'un point arbitraire antérieur et ne doit être utilisé que pour la réalisation de calculs temporels relatifs.
`uint64_t walltimestamp`	Nombre actuel de nanosecondes depuis 00:00 (heure coordonnée universelle) le 1er janvier 1970.

Les fonctions intégrées au langage D, comme `trace()` sont présentées dans le Chapitre10, "Actions et sous-routines".

Variables externes

Le langage D utilise le caractère ‘ comme opérateur d'étendue spécial pour accéder aux variables définies dans le système d'exploitation et non dans votre programme en D. Par exemple, le noyau de Solaris contient une déclaration en C du réglage système `kmem_flags` pour activer les fonctions de débogage du programme d'allocation de mémoire. Reportez-vous au manuel *Solaris Tunable Parameters Reference Manual* pour plus d'informations sur `kmem_flags`. Ce réglage est déclaré sous forme de variable en C dans la source du noyau comme suit :

```
int kmem_flags;
```

Pour accéder à la valeur de cette variable dans un programme en D, utilisez la notation en D suivante :

```
‘kmem_flags
```

DTrace associe chaque symbole de noyau au type utilisé pour le symbole dans le code en C correspondant du système d'exploitation, simplifiant ainsi l'accès basé sur la source aux structures de données du système d'exploitation natif. Pour utiliser les variables externes du système d'exploitation, vous devez accéder au code source correspondant du système d'exploitation.

Lorsque vous accédez à des variables externes à partir d'un programme en D, vous accédez aux détails d'implémentation interne d'un autre programme comme le noyau du système d'exploitation ou ses pilotes de périphérique. Ces détails d'implémentation ne constituent pas une interface stable sur laquelle vous pouvez compter ! Tout programme en D que vous écrivez et qui repose sur ces détails peut être interrompu lors de la prochaine mise à jour de la partie logicielle correspondante. C'est pourquoi, les variables externes sont généralement utilisées par les développeurs de noyaux et de pilotes de périphérique, ainsi que par les techniciens de maintenance, pour déboguer les problèmes de performances ou de fonctionnalités à l'aide de DTrace. Pour plus d'informations sur la stabilité de vos programmes en D, reportez-vous au Chapitre39, “Stabilité”.

Les noms de symbole de noyau sont conservés dans un espace de noms distinct des identificateurs de fonctions et de variables en D de sorte que vous n'ayiez jamais à vous préoccuper des conflits de noms avec vos variables en D. Lorsque vous ajoutez le préfixe ‘ à une variable, le compilateur D recherche les symboles de noyau connus dans la liste de modules chargés pour trouver une définition de variable correspondante. Comme le noyau de Solaris prend en charge les modules chargés de manière dynamique avec les espaces de noms distincts, le même nom de variable peut être utilisé plusieurs fois dans le noyau actif du système d'exploitation. Vous pouvez résoudre ces conflits de noms en spécifiant le nom du module de noyau dont l'accès à la variable doit être exécuté avant l'accès au symbole ‘ dans le nom du symbole. Par exemple, chaque module de noyau chargeable fournit généralement une fonction _fini(9E). Pour renvoyer à l'adresse de la fonction _fini fournie par le module de noyau foo, vous devez écrire :

```
foo‘_fini
```

Vous pouvez appliquer des opérateurs en D aux variables externes, à l'exception de ceux qui modifient des valeurs, en fonction des règles usuelles des types d'opérande. Lorsque vous lancez DTrace, le compilateur en D charge l'ensemble des noms de variable correspondant aux modules de noyau actifs, par conséquent, les déclarations de ces variables ne sont pas requises. Vous ne pouvez pas appliquer d'opérateur modifiant sa valeur à une variable externe, comme = ou +=. À des fins de sécurité, DTrace vous empêche d'endommager ou de corrompre l'état du logiciel que vous observez.

◆ ◆ ◆ CHAPITRE 4

Structure de programme D

Les programmes D sont constitués d'un ensemble de clauses décrivant des sondes à activer, ainsi que des prédicats et des actions à associer à ces sondes. Les programmes en D peuvent également contenir des déclarations de variables, comme le décrit le Chapitre3, “Variables”, ainsi que des définitions de nouveaux types, comme le décrit le Chapitre8, “Définitions des types et des constantes”. Ce chapitre décrit la structure générale d'un programme D ainsi que les fonctions de développement de descriptions de sondes correspondant à plusieurs sondes. Nous aborderons également l'utilisation du préprocesseur C, `cpp`, avec des programmes D.

Clauses et déclarations de sondes

Tel qu'illustré dans les exemples jusqu'ici, un fichier source de programme D comprend une ou plusieurs clauses de sonde décrivant l'instrumentation à activer par DTrace. Chaque clause de sonde présente la forme générale suivante :

descriptions de sondes
/ *prédicat* / { *instructions d'action*}

Le prédicat et la liste d'instructions d'action peuvent être omis. Toute directive rencontrée en dehors des clauses de sonde sont appelées des *déclarations*. Les déclarations ne peuvent être utilisées qu'en dehors des clauses de sonde. Aucune déclaration placée entre `{ }` n'est autorisée et les déclarations ne peuvent pas s'intercaler entre les éléments de la clause de sonde illustrée ci-dessus. Un espace peut être utilisé pour séparer les éléments d'un programme D et pour indenter des instructions d'action.

Les déclarations peuvent être utilisées pour déclarer des variables D et des symboles C externes, tel que décrit dans le Chapitre3, “Variables”, ou pour définir de nouveaux types d'utilisation dans D, tel que décrit dans le Chapitre8, “Définitions des types et des constantes” Des directives de compilateur C spéciales appelées *instructions pour compilateur* peuvent également se trouver n'importe où dans un programme D, y compris à l'extérieur de clauses de sonde. Les instructions de compilateur D sont spécifiées sur les lignes commençant par `#`. Les instructions

pour compilateur D sont utilisées, par exemple, pour définir des options d'exécution DTrace ; pour plus d'informations, reportez-vous au Chapitre16, “Options et paramètres réglables”.

Descriptions de sonde

Chaque clause de programme D commence par une liste d'une ou plusieurs descriptions de sonde, chacune prenant la forme habituelle suivante :

nom:*fonction*:*module*:*fournisseur*

Si un ou plusieurs champs de la description de sonde sont omis, les champs spécifiés sont interprétés de droite à gauche par le compilateur D. Par exemple, la description de sonde `foo:bar` correspondrait à une sonde dont la fonction est `foo` et le nom `bar` quelle que soit la valeur du fournisseur de la sonde et des champs de module. Une description de sonde est donc plus considérée comme un *modèle* pouvant être utilisé pour correspondre à une ou plusieurs sondes en fonction de leur nom.

Vous devez rédiger vos descriptions de sonde D en spécifiant les quatre délimiteurs de champ afin de pouvoir spécifier le *fournisseur* souhaité à gauche. Si vous ne spécifiez pas de fournisseur, des résultats inattendus sont possibles si plusieurs fournisseurs proposent des sondes du même nom. De même, des versions futures de DTrace peuvent inclure de nouveaux fournisseurs dont les sondes correspondent involontairement à vos descriptions de sonde partiellement définies. Vous pouvez spécifier un fournisseur mais faire correspondre l'une de ses sondes en laissant les champs de module, fonction et nom vides. Par exemple, la description `syscall:::` peut être utilisée pour faire correspondre chaque sonde proposée par le fournisseur `syscall` DTrace.

Les descriptions de sonde prennent également en charge une syntaxe de correspondance de modèle similaire à celle du *globbing* de shell décrite dans `sh(1)`. Avant de faire correspondre une sonde et une description, DTrace analyse chaque champ de description à la recherche des caractères `*`, `?` et `[`. Si l'un de ces caractères est présent dans un champ de description de sonde et qu'il n'est pas précédé de `\`, le champ est considéré comme un modèle. Le modèle de description doit correspondre à l'ensemble du champ correspondant d'une sonde donnée. La description de sonde entière doit correspondre champ par champ pour une correspondance totale et une activation de la sonde. Un champ de description de sonde qui n'est pas un modèle doit correspondre exactement au champ correspondant de la sonde. Un champ de description vide correspond à n'importe quelle sonde.

Les caractères spéciaux du tableau suivant sont reconnus dans des modèles de nom de sonde :

TABLEAU 4–1 Caractères de correspondance de modèle de nom de sonde

Symbole	Description
*	Correspond à toute chaîne, y compris la chaîne nulle.

TABLEAU 4–1 Caractères de correspondance de modèle de nom de sonde *(Suite)*

Symbole	Description
`?`	Correspond à un caractère unique.
`[ ... ]`	Correspond à l'un des caractères entre crochets. Une paire de caractères séparée par `-` correspond à tout caractère situé entre la paire. Si le premier caractère après `[` est `!`, tout caractère non inclus dans l'ensemble est mis en correspondance.
`\`	Interprète le caractère suivant tel quel, sans signification particulière.

Les caractères de correspondance de modèle peuvent être utilisés dans l'un ou les quatre champs de vos descriptions de sonde. Vous pouvez également utiliser des modèles pour répertorier des sondes correspondantes à l'aide des modèles de la ligne de commande `dtrace - l`. Par exemple, la commande `dtrace -l -f kmem_*` répertorie toutes les sondes DTrace dans des fonctions dont le nom commence par le préfixe `kmem_`.

Pour spécifier le même prédicat et les mêmes actions pour plusieurs descriptions ou modèles de sonde, vous pouvez créer une liste de descriptions séparée par une virgule. Par exemple, le programme D suivant permettrait de suivre un horodatage chaque fois que des sondes associées à une entrée d'appels système contenant les mots `lwp` ou `sock` sont déclenchées :

```
syscall::*lwp*:entry, syscall::*sock*:entry
{
    trace(timestamp);
}
```

Une description de sonde peut également spécifier une sonde à l'aide de son ID entier. Par exemple, la clause :

```
12345
{
    trace(timestamp);
}
```

pourrait être utilisée pour activer l'ID de sonde 12345, tel que renvoyé par la commande `dtrace -l -i 12345`. Vous devez toujours rédiger vos programmes D à l'aide de descriptions de sonde compréhensibles. Les ID de sonde entiers ne sont pas nécessairement cohérents car les modules de noyau du fournisseur DTrace sont chargés et déchargés ou suivent une réinitialisation.

Prédicats

Les prédicats sont des expressions comprises entre barres obliques `//` et évaluées au déclenchement d'une sonde afin de déterminer si les actions associées doivent être exécutées. Les prédicats constituent la base conditionnelle principale du développement d'un flux de commande plus complexe d'un programme D. Vous pouvez omettre la section des prédicats de la clause de chaque sonde, auquel cas les actions sont toujours exécutées au déclenchement de la sonde.

Les expressions de prédicat peuvent utiliser l'un des opérateurs D décrits précédemment et peuvent faire référence à des objets de données D comme des variables et des constantes. L'expression de prédicat doit évaluer une valeur de type entier ou pointeur afin de pouvoir être déterminée comme vraie ou fausse. Comme pour toutes les expressions D, une valeur de zéro est interprétée comme fausse et une valeur autre que zéro est interprétée comme vraie.

Actions

Les actions de sonde sont décrites par une liste d'instructions séparée par un point-virgule(`;`) et placée entre `{ }`. Pour ne noter qu'une sonde particulière déclenchée sur une CPU donnée sans suivre de données ou exécuter des actions supplémentaires, vous pouvez spécifier un ensemble d'accolades vides sans instructions.

Utilisation du préprocesseur C

Le langage de programmation C utilisé pour définir des interfaces de système Solaris comprend un *préprocesseur* qui exécute un ensemble d'étapes initiales dans la compilation de programme C. Le préprocesseur C est fréquemment utilisé pour définir des substitutions de macro dans lesquelles un jeton de programme C est remplacé par un autre jeu de jetons prédéfini ou pour inclure des copies de fichiers d'en-tête système. Vous pouvez utiliser le préprocesseur C avec vos programmes D en spécifiant l'option `dtrace -C`. Cette option amène `dtrace` à exécuter tout d'abord le préprocesseur `cpp(1)` sur votre fichier source de programme, puis à transmettre les résultats au compilateur D. Le préprocesseur C est décrit plus en détails dans *The C Programming Language*.

Le compilateur D charge automatiquement l'ensemble de descriptions du type C associé à la mise en œuvre du système d'exploitation, mais vous pouvez utiliser le préprocesseur pour inclure d'autres définitions de types comme ceux utilisés dans vos programmes C. Vous pouvez également utiliser le préprocesseur pour exécuter d'autres tâches comme la création de macros devenant des éléments de code D et d'autres éléments de programme. Si vous utilisez le préprocesseur avec votre programme D, vous ne pouvez inclure que des fichiers contenant des déclarations D valides. Les fichiers d'en-tête C classiques n'incluent que des déclarations externes de types et de symboles, qui seront interprétées correctement par le compilateur D. Le compilateur D ne peut pas analyser les fichiers d'en-tête C comprenant d'autres éléments de programme comme du code source de fonction C et renverra un message d'erreur.

CHAPITRE 5

Pointeurs et ensembles

Pointeurs et adresses mémoire d'objets de données dans le noyau du système d'exploitation ou dans l'espace d'adresse d'un processus utilisateur. D offre la possibilité de créer et de manipuler des pointeurs et de les stocker dans des variables et des ensembles associatifs. Ce chapitre décrit la syntaxe D de pointeurs, d'opérateurs qui peuvent être appliqués pour créer et accéder à des pointeurs, ainsi que la relation entre des pointeurs et des ensembles scalaires de taille fixe. Des problèmes liés à l'utilisation de pointeurs dans différents espaces d'adresse sont également abordés.

Remarque – Si vous êtes un programmeur C ou C++ expérimenté, vous pouvez survoler ce chapitre car la syntaxe de pointeur D est identique à la syntaxe ANSI-C correspondante. Nous vous conseillons de lire les sections "Pointeurs sur des objets DTrace" à la page 88 et "Pointeurs et espaces d'adresse" à la page 89, car elles présentent les fonctions et les problèmes spécifiques à DTrace.

Pointeurs et adresses

Le système d'exploitation Solaris utilise une technique appelée *mémoire virtuelle* pour fournir à chaque processus utilisateur un affichage virtuel propre des ressources mémoire du système. Un affichage virtuel des ressources mémoire est appelé *espace d'adresse*, qui associe une plage de valeurs d'adresses (`[0 ... 0xffffffff]` pour un espace d'adresse 32 bits ou `[0 ... 0xffffffffffffffff]` pour un espace d'adresse 64 bits) avec un ensemble de translations utilisé par le système d'exploitation et le matériel pour convertir chaque adresse virtuelle en un emplacement de mémoire physique correspondant. Les pointeurs dans D sont des objets de données qui stockent une valeur d'adresse virtuelle entière et l'associe à un type D qui décrit le format des données stockées à l'emplacement de mémoire correspondant.

Vous pouvez déclarer une variable D de sorte qu'elle soit un type de pointeur en spécifiant tout d'abord le type des données référencées, puis en ajoutant un astérisque (`*`) au nom du type pour indiquer que vous souhaitez déclarer un pointeur. Par exemple, la déclaration :

```
int *p;
```

déclare une variable globale D nommée p qui est un pointeur sur un entier. Cette déclaration signifie que p est un entier de 32 ou 64 bits dont la valeur est l'adresse d'un autre entier en mémoire. La forme compilée de votre code D étant exécutée au déclenchement de sonde dans le noyau du système d'exploitation, les pointeurs D sont généralement associés à l'espace d'adresse du noyau. Vous pouvez utiliser la commande isainfo(1) -b pour déterminer le nombre de bits utilisés pour les pointeurs par le noyau du système d'exploitation actif.

Pour créer un pointeur sur un objet de données dans le noyau, vous pouvez calculer son adresse à l'aide de l'opérateur &. Par exemple, le code source de noyau du système d'exploitation déclare un paramètre réglable int kmem_flags. Vous pourriez suivre l'adresse de int en suivant le résultat de l'application de l'opérateur & au nom de cet objet dans D :

```
trace(&‘kmem_flags);
```

L'opérateur * peut être utilisé pour faire référence à l'objet pointé et agit de manière inverse à l'opérateur &. Par exemple, les deux fragments de code D ont une signification équivalente :

```
p = &‘kmem_flags;                trace(‘kmem_flags);
trace(*p);
```

Le fragment de gauche crée un pointeur p de variable globale D. L'objet kmem_flags étant du type int, le type du résultat de &‘kmem_flags est int * (à savoir le pointeur sur int). Le fragment de gauche suit la valeur de *p, qui suit le pointeur sur l'objet de données kmem_flags. Ce fragment est donc identique à celui de droite, qui suit simplement la valeur de l'objet de données via son nom.

Sécurité du pointeur

Si vous êtes un programmeur C ou C++, vous pouvez être quelque peu inquiet après avoir lu la section précédente car vous savez qu'une mauvaise utilisation de pointeurs dans vos programmes peut entraîner leur panne. DTrace est un environnement puissant et sécurisé permettant d'exécuter vos programmes D là où des erreurs ne peuvent pas entraîner de panne des programmes. Vous pouvez en effet rédiger un programme D avec bogue, mais les accès d'un pointeur D non valide n'entraîneront pas une défaillance ou une panne de DTrace ou du noyau du système d'exploitation. Le logiciel DTrace détectera plutôt tous les accès de pointeur non valides, désactivera votre instrumentation, et vous signalera le problème pour débogage.

Si vous avez programmé en langage Java, vous savez probablement que ce langage ne prend pas en charge les pointeurs pour ces mêmes raisons de sécurité. Des pointeurs sont nécessaires dans D car ils font intrinsèquement partie de la mise en œuvre du système d'exploitation dans C, mais DTrace met en œuvre le même type de mécanisme de sécurité rencontré dans le langage de programmation Java empêchant une détérioration des programmes avec bogue eux-mêmes ou

les uns par rapport aux autres. Les rapports d'erreurs DTrace sont similaires à l'environnement d'exécution du langage de programmation Java qui détecte une erreur de programmation et qui vous signale une exception.

Pour consulter la gestion et les rapports d'erreurs DTrace, rédigez un programme D incorrect utilisant des pointeurs. Dans un éditeur, entrez le programme D suivant et enregistrez-le sous un fichier nommé `badptr.d` :

EXEMPLE 5–1 `badptr.d` : démonstration de la gestion d'erreurs DTrace

```
BEGIN
{
    x = (int *)NULL;
    y = *x;
    trace(y);
}
```

Le programme `badptr.d` crée un pointeur D nommé `x` qui pointe sur `int`. Le programme assigne à ce pointeur la valeur spéciale de pointeur non valide `NULL`, qui est un alias intégré pour l'adresse 0. Par convention, l'adresse 0 est toujours définie comme non valide, de façon à ce que `NULL` soit utilisé en tant valeur sentinelle dans les programmes en langages C et D. Le programme utilise une expression de forçage de type pour convertir `NULL` en pointeur sur un entier. Le programme déréférence alors le pointeur à l'aide de l'expression `*x` et attribue le résultat à une autre variable `y`, puis tente de suivre `y`. Une fois le programme D exécuté, DTrace détecte un accès de pointeur non valide lorsque l'instruction `y = *x` est exécutée et signale l'erreur suivante :

```
# dtrace -s badptr.d
dtrace: script '/dev/stdin' matched 1 probe
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address
(0x0) in action #2 at DIF offset 4
dtrace: 1 error on CPU 0
^C
#
```

L'autre problème pouvant survenir des programmes utilisant des pointeurs non valides est une *erreur d'alignement*. Par convention architecturale, des objets de données fondamentaux comme des entiers sont alignés dans la mémoire en fonction de leur taille. Par exemple, des entiers 2 octets sont alignés sur des adresses qui sont des multiples d'entiers 2, 4 octets par multiples de 4, etc. Si vous déréférencez un pointeur sur un entier 4 octets et que votre adresse de pointeur est une valeur non valide autre qu'un multiple de 4, votre accès échouera et renverra une erreur d'alignement. Les erreurs d'alignement dans D indiquent quasiment toujours que votre pointeur comporte une valeur non valide ou corrompue en raison d'un bogue dans votre programme D. Vous pouvez créer un exemple d'erreur d'alignement en changeant le code

source de badptr.d pour utiliser l'adresse (int *)2 plutôt que NULL. int étant de 4 octets et 2 n'étant pas un multiple de 4, l'expression *x entraîne une erreur d'alignement DTrace.

Pour plus d'informations sur le mécanisme d'erreur DTrace, reportez-vous à la section “Sonde ERROR” à la page 203.

Déclarations et stockage d'ensembles

Outre les tableaux associatifs dynamiques présentés dans le chapitre 3, D prend également en charge les *tableaux scalaires*. Les tableaux scalaires sont un groupe de longueur fixe contenant des emplacements mémoire consécutifs, chacun d'eux stockant une valeur du même type. Les ensembles scalaires sont accessibles en référençant chaque emplacement avec un entier à partir de zéro. Les ensembles scalaires correspondent au concept et à la syntaxe des ensembles dans C et C++. Les tableaux scalaires ne sont pas utilisés aussi fréquemment dans D que les tableaux associatifs et leurs *groupements* plus avancés, mais ils sont parfois nécessaires pour accéder aux structures de données de tableau du système d'exploitation existantes déclarées dans C. Les groupements sont décrits dans le Chapitre9, “Groupements”.

Un ensemble scalaire D de 5 entiers pourrait être déclaré à l'aide du type int et en suffixant la déclaration par le nombre d'éléments entre crochets comme suit :

```
int a[5];
```

Le diagramme suivant illustre une représentation visuelle du stockage d'ensembles :

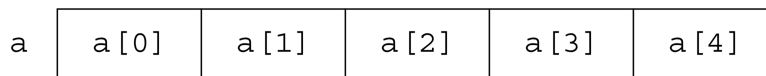

FIGURE 5–1 Représentation d'ensembles scalaires

L'expression D a[0] permet de référencer le premier élément de l'ensemble, a[1] le deuxième, etc. D'un point de vue syntaxique, les ensembles scalaires et associatifs sont très similaires. Vous pouvez déclarer un ensemble associatif de cinq entiers référencé par un entier comme suit :

```
int a[int];
```

et référencer également cet ensemble à l'aide de l'expression a[0]. Cependant, du point de vue stockage et mise en œuvre, les deux ensembles sont très différents. L'ensemble statique a comprend cinq emplacements mémoire consécutifs à partir de zéro et l'index fait référence à un décalage dans le stockage attribué à l'ensemble. D'autre part, un ensemble associatif ne dispose pas de taille prédéfinie et ne stocke pas d'éléments dans des emplacements mémoire consécutifs. De plus, les clés d'ensemble associatif n'ont aucune relation avec l'emplacement de stockage de valeur correspondante. Vous pouvez accéder aux éléments d'ensemble associatif a[0] et a[-5]

et deux mots de stockage seulement seront attribués par DTrace, ceux-ci pouvant être consécutifs ou non. Les clés d'ensemble associatif sont des noms abstraits de la valeur correspondante sans relation avec les emplacements de stockage de valeur.

Si vous créez un ensemble à l'aide d'une affectation initiale et que utilisez une seule expression d'entier comme index d'ensemble (par exemple, `a[0] = 2`), le compilateur D crée toujours un ensemble associatif, même si cette expression `a` peut également être interprétée comme une affectation d'ensemble scalaire. Les ensembles scalaires doivent être prédéclarés dans ce cas pour que le compilateur D puisse afficher la définition de taille de l'ensemble et en déduire que l'ensemble est un ensemble scalaire.

Relation entre pointeur et ensemble

Les pointeurs et ensembles entretiennent une relation spéciale dans D, tout comme dans ANSI-C. Un ensemble est représenté par une variable associée à l'adresse de son premier emplacement de stockage. Un pointeur représente également l'adresse d'un emplacement de stockage d'un type défini. D permet donc d'utiliser la notation d'index d'ensemble [ ] avec des variables de pointeur et des variables d'ensemble. Par exemple, les deux fragments D suivants ont une signification équivalente :

```
p = &a[0];                trace(a[2]);
trace(p[2]);
```

Dans le fragment de gauche, le pointeur `p` est affecté à l'adresse du premier élément de l'ensemble dans `a` en appliquant l'opérateur `&` à l'expression `a[0]`. L'expression `p[2]` suit la valeur du troisième élément (index 2) de l'ensemble. `p` contenant désormais la même adresse associée à `a`, cette expression donne la même valeur que `a[2]`, tel qu'illustré dans le fragment de droite. Une conséquence de cette équivalence est que C et D vous permettent d'accéder à tout index de tout pointeur ou ensemble. La vérification des liaisons d'ensemble n'est pas exécutée par le compilateur ou l'environnement d'exécution DTrace pour vous. Si vous accédez à la mémoire après la fin d'une valeur prédéfinie d'un ensemble, vous obtiendrez un résultat inattendu ou DTrace signalera une erreur d'adresse non valide, tel qu'illustré dans l'exemple précédent. Comme toujours, vous ne pouvez pas détériorer DTrace ou le système d'exploitation, mais vous devrez déboguer le programme D.

La différence entre les pointeurs et les ensembles est telle qu'une variable de pointeur fait référence à un stockage distinct contenant l'adresse d'entier d'un autre stockage. Un variable d'ensemble nomme le stockage de l'ensemble, et non l'emplacement d'un entier contenant à son tour l'emplacement de l'ensemble. Cette différence est illustrée dans le diagramme suivant :

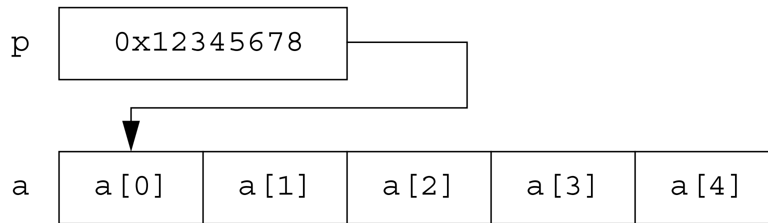

FIGURE 5–2 Stockage de pointeur et d'ensemble

Cette différence est représentée dans la syntaxe D si vous tentez d'affecter des pointeurs et des ensembles scalaires. Si `x` et `y` sont des variables de pointeur, l'expression `x = y` est légale. Elle copie simplement l'adresse de pointeur dans `y` vers l'emplacement de stockage nommé par `x`. Si `x` et `y` sont des variables d'ensemble scalaire, l'expression `x = y` n'est pas légale. Des ensembles peuvent ne pas être intégralement affectés dans D. Cependant, une variable d'ensemble ou un nom de symbole peut être utilisé au cas où un pointeur serait autorisé. Si `p` est un pointeur et que `a` est un ensemble, l'instruction `p = a` est autorisée. Celle-ci équivaut à l'instruction `p = &a[0]`.

Arithmétique de pointeur

Les pointeurs n'étant que des entiers utilisés comme des adresses d'autres objets en mémoire, D propose un ensemble de fonctions permettant d'exécuter une arithmétique sur des pointeurs. L'arithmétique de pointeur n'est cependant pas identique à l'arithmétique d'entier. L'arithmétique de pointeur définit implicitement l'adresse sous-jacente en multipliant ou en divisant les opérandes par la taille du type référencé par le pointeur. Le fragment D suivant illustre cette propriété :

```
int *x;

BEGIN
{
    trace(x);
    trace(x + 1);
    trace(x + 2);
}
```

Ce fragment crée un pointeur entier `x`, puis suit sa valeur, incrémentée de un, et sa valeur incrémentée de deux. Si vous créez et exécutez ce programme, DTrace signale les valeurs d'entier 0, 4 et 8.

`x` étant un pointeur sur un entier (4 octets), l'incrémentation de `x` ajoute 4 à la valeur de pointeur sous-jacente. Cette propriété est utile en cas d'utilisation de pointeur pour faire référence à des emplacements de stockage consécutifs comme des ensembles. Par exemple, si `x` était affecté à l'adresse d'un tableau `a`, similaire à celui illustré dans la Figure 5–2, l'expression `x + 1` serait

équivalente à l'expression &a[1]. De même, l'expression *(x + 1) ferait référence à la valeur a[1]. L'arithmétique de pointeur est mise en œuvre par le compilateur D si une valeur de pointeur est incrémentée à l'aide de l'opérateur +=, + ou ++.

L'arithmétique de pointeur s'applique également lorsqu'un entier est soustrait d'un pointeur à gauche, lorsqu'un pointeur est soustrait d'un autre pointeur ou lorsque l'opérateur -- est appliqué à un pointeur. Par exemple, le programme D suivant suivrait le résultat 2 :

```
int *x, *y;
int a[5];

BEGIN
{
    x = &a[0];
    y = &a[2];
    trace(y - x);
}
```

Pointeurs génériques

Il est parfois utile de représenter ou de manipuler une adresse de pointeur générique dans un programme D sans spécifier le type de données référencées par le pointeur. Des pointeurs génériques peuvent être spécifiés à l'aide du type void *, où le mot-clé void représente l'absence d'informations de type spécifiques, ou à l'aide de l'alias de type intégré uintptr_t qui représente un alias de type de taille entier non signé approprié d'un pointeur dans le modèle de données en cours. Vous ne pouvez pas appliquer l'arithmétique de pointeur à un objet du type void *, et ces pointeurs ne peuvent pas être déréférencés sans qu'ils aient été tout d'abord forcés à un autre type. Vous pouvez forcer le type d'un pointeur sur uintptr_t en cas d'exécution nécessaire d'arithmétique d'entier sur la valeur de pointeur.

Les pointeurs sur void peuvent être utilisés lorsqu'un autre type de données pour un pointeur est nécessaire, comme une expression de tuple d'ensemble associatif ou à droite d'une instruction d'affectation. De même, un pointeur sur un quelconque type de données peut être utilisé lorsqu'un pointeur sur void est nécessaire. Pour utiliser un pointeur de type non-void à la place d'un autre type de pointeur non-void, un forçage de type explicite est nécessaire. Vous devez toujours utiliser des forçages de type explicites pour convertir des pointeurs en types d'entier comme uintptr_t ou pour reconvertir ces entiers dans le type de pointeur approprié.

Ensembles multidimensionnels

Les ensembles scalaires multidimensionnels ne sont pas utilisés fréquemment dans D, mais sont proposés pour une compatibilité avec ANSI-C et pour observer et accéder à des structures de données système créées à l'aide de cette fonctionnalité dans C. Un ensemble multidimensionnel est déclaré comme une série consécutive de tailles d'ensembles scalaires comprise entre crochets [ ] et suivie du type de base. Par exemple, pour déclarer un ensemble rectangulaire à deux dimensions de taille fixe dont les dimensions sont de 12 lignes par 34 colonnes, rédigez la déclaration suivante :

```
int a[12][34];
```

Un ensemble scalaire multidimensionnel est accessible via une notation similaire. Par exemple, pour accéder à la valeur stockée à la ligne 0, colonne 1, rédigez l'expression D suivante :

```
a[0][1]
```

Les emplacements de stockage des valeurs d'ensemble scalaire multidimensionnel sont calculés en multipliant le nombre de lignes par le nombre total de colonnes déclarées, puis en ajoutant le nombre de colonnes.

Veillez à ne pas confondre la syntaxe d'ensemble multidimensionnel avec la syntaxe D d'accès d'ensemble associatif (à savoir que `a[0][1]` est différent de `a[0, 1]`). Si vous utilisez un tuple incompatible avec un ensemble associatif ou si vous tentez un accès d'ensemble associatif d'un ensemble scalaire, le compilateur D renverra un message d'erreur et ne compilera pas votre programme.

Pointeurs sur des objets DTrace

Le compilateur D vous empêche d'utiliser l'opérateur & afin d'obtenir des pointeurs sur des objets DTrace comme des ensembles associatifs, des fonctions intégrées et des variables. Vous ne pouvez pas obtenir l'adresse de ces variables de sorte que l'environnement d'exécution DTrace est libre de les relocaliser si nécessaire entre des déclenchements de sonde afin de gérer plus efficacement la mémoire nécessaire aux programmes. Si vous créez des structures composites, il est possible de développer des expressions permettant de récupérer l'adresse du noyau de votre stockage d'objet DTrace. Vous ne devez pas créer de telles expressions dans vos programmes D. Si vous devez utiliser une telle expression, veillez à ne pas mettre l'adresse en cache entre les déclenchements de sonde.

Dans ANSI-C, les pointeurs peuvent également être utilisés pour exécuter des appels de fonction indirects ou des affectations, comme le placement d'une expression à l'aide de l'opérateur de déréférencement unaire * à gauche d'un opérateur d'affectation. Dans D, ces types d'expression utilisant des pointeurs ne sont pas autorisés. Vous ne pouvez affecter des valeurs directement à des variables D que via leur nom ou en appliquant l'opérateur d'index

d'ensemble [] à un ensemble scalaire ou associatif D. Vous pouvez uniquement appeler des fonctions définies par l'environnement DTrace par leur nom, tel que spécifié dans le Chapitre10, "Actions et sous-routines" Les appels de fonction indirects utilisant des pointeurs ne sont pas autorisés dans D.

Pointeurs et espaces d'adresse

Un pointeur est une adresse proposant une translation dans un *espace d'adresse virtuelle* vers un emplacement de la mémoire physique. DTrace exécute vos programmes D dans l'espace d'adresse virtuelle du noyau du système d'exploitation. Votre système Solaris gère de nombreux espaces d'adresse : un pour le noyau du système d'exploitation et un pour chaque processus utilisateur. Chaque espace d'adresse semblant pouvoir accéder à l'ensemble de la mémoire du système, la même valeur de pointeur d'adresse virtuelle peut être réutilisée pour d'autres espaces d'adresse mais refléter une autre mémoire physique. Par conséquent, lors de la rédaction de programmes D utilisant des pointeurs, vous devez connaître l'espace d'adresse correspondant aux pointeurs que vous souhaitez utiliser.

Par exemple, si vous utilisez le fournisseur `syscall` pour instrumentaliser l'entrée d'un appel système utilisant un pointeur sur un entier ou un ensemble d'entiers comme argument (par exemple, `pipe(2)`), il ne serait pas approprié de déréférencer ce pointeur ou cet ensemble à l'aide de l'opérateur `*` ou `[]`, car l'adresse concernée est une adresse contenue dans l'espace d'adresse du processus utilisateur à l'origine de l'appel système. L'application de l'opérateur `*` ou `[]` à cette adresse dans D entraînerait un accès à l'espace d'adresse du noyau, causant ainsi une erreur d'adresse ou l'obtention de données inattendues dans votre programme D si l'adresse devait correspondre à une adresse de noyau valide.

Pour accéder à la mémoire de processus utilisateur à partir d'une sonde DTrace, vous devez appliquer l'une des fonctions `copyin()`, `copyinstr()` ou `copyinto()`, décrites dans le Chapitre10, "Actions et sous-routines", au pointeur d'espace d'adresse utilisateur. Lors de la rédaction de programmes D, veillez à nommer et à commenter les variables en stockant de manière appropriée les adresses utilisateur pour éviter toute confusion. Vous pouvez également stocker des adresses utilisateur sous `uintptr_t` afin de ne pas compiler par inadvertance le code D les déréférençant. Les techniques d'utilisation de DTrace sur des processus utilisateur sont présentées dans le Chapitre33, "Suivi des processus utilisateur".

CHAPITRE 6

Chaînes de caractères

DTrace assure la prise en charge du suivi et de la manipulation des chaînes. Ce chapitre décrit toutes les fonctionnalités du langage D qui permettent de déclarer et de manipuler des chaînes. Contrairement au langage ANSI-C, les chaînes en langage D possèdent leur propre prise en charge des types et opérateurs. Vous pouvez donc aisément et sans ambiguïté les utiliser dans vos programmes de suivi.

Représentation de chaînes

Les chaînes sont représentées dans DTrace sous la forme d'un tableau de caractères terminé par un octet nul (c'est-à-dire, un octet dont la valeur est égale à zéro, généralement présenté sous la forme `'\0'`). La partie visible de la chaîne correspond à la longueur de la variable, en fonction de l'emplacement de l'octet nul, mais DTrace enregistre chaque chaîne dans un tableau à taille fixe de sorte que chaque sonde suive une quantité conséquente de données. Les chaînes ne peuvent pas excéder la longueur de cette limite prédéfinie. Par contre, il est possible de modifier cette dernière dans votre programme en D ou sur la ligne de commande `dtrace` en réglant l'option `strsize.` Pour plus d'informations sur les options DTrace de réglage, reportez-vous au Chapitre16, “Options et paramètres réglables”. La limite de chaîne par défaut est de 256 octets.

Le langage en D offre un type `string` explicite au lieu d'utiliser le type `char *` pour faire référence aux chaînes. Le type `string` équivaut à `char *` en ce qu'il correspond à l'adresse d'une séquence de caractères, mais les capacités du compilateur D et des fonctions du langage D comme le fournisseur `trace()` sont optimisées pour les expressions de type `string`. Par exemple, le type de chaîne supprime l'ambiguïté du type `char *` lorsque vous devez suivre les octets réels d'une chaîne. Dans un énoncé en D :

```
trace(s);
```

Si `s` est de type `char *`, DTrace suit la valeur du pointeur `s` (il suit une valeur d'adresse entière). Dans un énoncé en D :

```
trace(*s);
```

En définissant l'opérateur `*`, le compilateur D déréférence le pointeur `s` et suit le caractère unique à cet emplacement. Ces comportements sont essentiels pour vous permettre de manipuler les pointeurs de caractère qui font référence, à dessein, à d'autres caractères uniques ou à des tableaux de nombres entiers de la taille d'un octet ne constituant pas des chaînes et ne se terminant pas par un octet nul. Dans un énoncé en D :

```
trace(s);
```

Si `s` est de type `string`, le type `string` indique au compilateur D que vous souhaitez que DTrace suive une chaîne de caractères terminée par un caractère nul dont l'adresse est enregistrée dans la variable `s`. Vous pouvez également procéder à une comparaison lexicale des expressions de type `string`, comme indiqué à la section “Comparaison de chaînes” à la page 93.

Constantes de chaîne

Les constantes de chaîne figurent entre guillemets doubles ("). Le type `string` leur est automatiquement attribué par le compilateur D. Vous pouvez définir des constantes de chaîne de n'importe quelle longueur, la seule limite étant la qualité de mémoire que DTrace peut utiliser sur votre système. L'octet nul de terminaison (`\0`) est automatiquement ajouté par le compilateur D à toutes les constantes de chaîne que vous déclarez. La taille d'un objet de constante de chaîne correspond au nombre d'octets associé à la chaîne auquel vient s'ajouter un octet supplémentaire pour l'octet nul de terminaison.

Une constante chaîne ne peut pas contenir de caractère littéral de retour à la ligne. Pour créer des chaînes contenant des retours à la ligne, utilisez la séquence d'échappement `\n` plutôt qu'un retour à la ligne littéral. Les constantes de chaîne peuvent également contenir l'une des séquences spéciales d'échappement de caractères définies pour les constantes de caractère dans le Tableau 2–5.

Assignation de chaîne

Contrairement à l'assignation des variables `char *`, les chaînes sont copiées par valeur et non par référence. L'assignation de chaîne est réalisée à l'aide de l'opérateur `=` et copie les octets réels de la chaîne de l'opérande source jusqu'à l'octet nul, ce dernier compris, dans la variable située à gauche qui doit être de type `string`. Vous pouvez créer une nouvelle variable de type `string` en lui assignant une expression de type `string`. Par exemple, l'énoncé en D

```
s = "hello";
```

doit créer une nouvelle variable `s` de type `string` et y copier les 6 octets de la chaîne `"hello"` (5 caractères imprimables, plus l'octet nul). L'assignation de chaîne est un processus similaire à la fonction de bibliothèque du langage C `strcpy(3C)`, hormis le fait que si la chaîne source dépasse la limite de stockage de la chaîne de destination, la chaîne qui en résulte est automatiquement tronquée à cette limite.

Vous pouvez également assigner à une variable de chaîne une expression d'un type compatible avec les chaînes. Le cas échéant, le compilateur D promeut automatiquement l'expression source en type de chaîne et exécute une assignation de chaîne. Le compilateur D autorise la promotion de n'importe quelle expression de type `char *` ou `char[`*n*`]` (c'est-à-dire un tableau scalaire de `char` de n'importe quelle taille) en `string`.

Conversion de chaîne

Les expressions d'autres types peuvent être explicitement converties en type `string` à l'aide d'une expression de forçage de type ou en appliquant l'opérateur spécial `stringof`, ces deux solutions ayant une signification identique :

```
s = (string) expression            s = stringof ( expression )
```

L'opérateur `stringof` est très étroitement lié à l'opérande de droite. En règle générale, l'expression est placée entre parenthèses pour plus de clarté, les parenthèses n'étant pas impérativement requises.

Toute expression de type scalaire comme un pointeur, un nombre entier ou une adresse de tableau scalaire est convertible en chaîne. Les expressions d'autres types comme `void` ne sont pas convertibles en `string`. Si vous convertissez par erreur une adresse invalide en chaîne, vous n'endommagerez ni le système ni DTrace grâce aux fonctions de sécurité de DTrace mais vous devrez interrompre le suivi de la séquence de caractères indéchiffrables.

Comparaison de chaînes

D surcharge les opérateurs relationnels binaires et autorise leur utilisation dans les comparaisons de chaînes et de nombres entiers. Les opérateurs relationnels comparent les chaînes chaque fois que les deux opérandes sont de type `string` ou lorsqu'un opérande est de type `string` tandis que l'autre peut être promu en `string`, comme indiqué à la section “Assignation de chaîne” à la page 92. Vous pouvez utiliser tous les opérateurs relationnels pour comparer des chaînes :

TABLEAU 6–1 Opérateurs relationnels et chaînes en langage D

`<`	opérande gauche inférieur à l'opérande droit
`<=`	opérande gauche inférieur ou égal à l'opérande droit
`>`	opérande gauche supérieur à l'opérande droit
`>=`	opérande gauche supérieur ou égal à l'opérande droit

TABLEAU 6–1 Opérateurs relationnels et chaînes en langage D *(Suite)*

`==`	opérande gauche égal à l'opérande droit
`!=`	opérande gauche différent de l'opérande droit

De même qu'avec les nombres entiers, chaque opérateur évalue la valeur du type `int` qui est égale à 1 si la condition est vraie et à 0 si la condition est fausse.

Les opérateurs relationnels comparent les deux chaînes d'entrée octet par octet, de la même manière que la routine de bibliothèque `strcmp`(3C) en langage C. Chaque octet est comparé à l'aide de sa valeur correspondante dans le jeu de caractères ASCII, comme illustré dans `ascii`(5), jusqu'à ce qu'un octet nul soit lu ou que la longueur de chaîne maximale soit atteinte. Exemples de comparaison de chaînes en D et leurs résultats :

`"coffee" < "espresso"`	... renvoie 1 (vrai)
`"coffee" == "coffee"`	... renvoie 1 (vrai)
`"coffee" >= "mocha"`	... renvoie 0 (faux)

CHAPITRE 7

Structs et Unions

Vous pouvez regrouper des ensembles de variables liées au sein d'objets de données composites appelés *structs* et *unions*. Vous pouvez définir ces objets en langage D en créant pour eux de nouvelles définitions de type, puis ajouter les nouveaux types dédiés aux variables en langage D, y compris les valeurs de tableau associatif. Ce chapitre étudie la syntaxe et la sémantique de création et de manipulation de ces types composites, ainsi que les opérateurs en langage D qui interagissent avec eux. La syntaxe des structs et unions est illustrée au moyen de plusieurs exemples de programmes démontrant l'utilisation des fournisseurs `fbt` et `pid` de DTrace.

Structs

Le mot-clé `struct` en langage D (raccourci de *structure*) est utilisé pour introduire un nouveau type composé d'un groupe d'autres types. Vous pouvez utiliser le nouveau type de struct comme type de tableaux et de variables en langage D dans le but de définir des groupes de variables connexes sous un seul nom. Les structs en langage D sont identiques aux constructions correspondantes en C et C++. En cas de programmation en langage java, imaginez une struct en langage D comme une classe qui ne contient que des membres de données et aucune méthode.

Imaginez que vous souhaitez créer, en langage D, un programme plus sophistiqué de suivi des appels système qui enregistre plusieurs données concernant chaque appel système `read(2)` et `write(2)` exécuté par votre shell (temps écoulé, nombre d'appels et plus grand comptage d'octets sous forme d'argument). Vous pouvez écrire une clause en langage D pour enregistrer ces propriétés dans trois tableaux associatifs distincts, comme illustré dans l'exemple suivant :

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{
    ts[probefunc] = timestamp;
    calls[probefunc]++;
    maxbytes[probefunc] = arg2 > maxbytes[probefunc] ?
        arg2 : maxbytes[probefunc];
}
```

Cette clause est toutefois inefficace car DTrace doit créer trois tableaux associatifs distincts et enregistrer les copies distinctes des valeurs de tuple identiques correspondant à `probefunc` pour chacune d'entre elles. Une struct vous permet d'économiser de l'espace et de simplifier la lecture et la préservation de votre programme. Pour ce faire, commencez par déclarer un nouveau type de struct au sommet du fichier source du programme :

```
struct callinfo {
    uint64_t ts;      /* timestamp of last syscall entry */
    uint64_t elapsed; /* total elapsed time in nanoseconds */
    uint64_t calls;   /* number of calls made */
    size_t maxbytes;  /* maximum byte count argument */
};
```

Le mot clé `struct` est suivi d'un identifiant en option qui sert de renvoi à notre nouveau type, appelé `struct callinfo`. Les membres de la struct sont ensuite placés entre crochets `{ }`, puis la déclaration toute entière se termine par un point-virgule (`;`). Chaque membre de la struct est défini à l'aide de la même syntaxe que celle d'une déclaration de variable en langage D, le type du membre étant indiqué en premier, puis suivi d'un identifiant permettant de nommer le membre et d'un autre point-virgule (`;`).

La déclaration de la struct en elle-même se contente de définir le nouveau type. Elle ne crée aucune variable ni n'alloue aucun stockage dans DTrace. La déclaration terminée, vous pouvez utiliser le type `struct callinfo` tout au long du reste de votre programme en D et chaque variable de type `struct callinfo` enregistre une copie des quatre variables décrites par notre modèle de structure. Les membres sont classés en mémoire conformément à l'ordre de la liste des membres, un espace de remplissage étant introduit entre chaque membre afin d'aligner les objets de données.

Vous pouvez utiliser les noms d'identifiant des membres pour accéder à la valeur de chaque membre à l'aide de l'opérateur “`.`” en écrivant une expression se présentant comme suit :

variable-name`.`*member-name*

L'exemple suivant représente un programme amélioré qui utilise le nouveau type de structure. Ouvrez votre éditeur et entrez le programme en langage D suivant, puis enregistrez-le dans le fichier `rwinfo.d`:

EXEMPLE 7–1 `rwinfo.d` : collecte des statistiques `read`(2) et `write`(2)

```
struct callinfo {
    uint64_t ts;      /* timestamp of last syscall entry */
    uint64_t elapsed; /* total elapsed time in nanoseconds */
    uint64_t calls;   /* number of calls made */
    size_t maxbytes;  /* maximum byte count argument */
};

struct callinfo i[string];    /* declare i as an associative array */
```

EXEMPLE 7–1 `rwinfo.d` : collecte des statistiques `read`(2) et `write`(2) *(Suite)*

```
syscall::read:entry, syscall::write:entry
/pid == $1/
{
    i[probefunc].ts = timestamp;
    i[probefunc].calls++;
    i[probefunc].maxbytes = arg2 > i[probefunc].maxbytes ?
        arg2 : i[probefunc].maxbytes;
}

syscall::read:return, syscall::write:return
/i[probefunc].ts != 0 && pid == $1/
{
    i[probefunc].elapsed += timestamp - i[probefunc].ts;
}

END
{
    printf("        calls  max bytes  elapsed nsecs\n");
    printf("------  -----  ---------  -------------\n");
    printf("  read  %5d  %9d  %d\n",
        i["read"].calls, i["read"].maxbytes, i["read"].elapsed);
    printf(" write  %5d  %9d  %d\n",
        i["write"].calls, i["write"].maxbytes, i["write"].elapsed);
}
```

Une fois le programme saisi, exécutez la commande `dtrace -q -s rwinfo.d` en spécifiant l'un de vos processus de shell. Entrez ensuite quelques commandes dans votre shell et, ceci terminé, tapez Control-C dans le terminal `dtrace` pour déclencher la sonde `END` et imprimer les résultats :

```
# dtrace -q -s rwinfo.d `pgrep -n ksh`
^C
        calls  max bytes  elapsed nsecs
------  -----  ---------  -------------
  read     36       1024  3588283144
 write     35         59  14945541
#
```

Pointeurs vers structs

Faire référence à des structs à l'aide de pointeurs est courant en C et en D. Vous pouvez utiliser l'opérateur `->` pour accéder aux membres d'une struct par l'intermédiaire d'un pointeur. Si une `struct s` intègre le membre `m` et qu'un pointeur sur cette struct s'appelle `sp` (`sp` étant une variable du type `struct s *`), vous pouvez utiliser l'opérateur `*` pour déréférencer le pointeur `sp`, afin d'accéder au membre :

```
struct s *sp;

(*sp).m
```

Vous pouvez également utiliser l'opérateur `->` comme raccourci pour cette notation. Les deux exemples en D suivants ont la même signification si `sp` est utilisé comme pointeur vers une struct :

```
(*sp).m                 sp->m
```

DTrace intègre plusieurs variables de type pointeurs vers structs, y compris `curpsinfo` et `curlwpsinfo`. Ces pointeurs renvoient respectivement aux structs `psinfo` et `lwpsinfo` et leur contenu offre un aperçu des informations relatives à l'état du processus courant et du processus léger (LWP) associés au thread ayant déclenché la sonde actuelle. Un processus Solaris léger est la représentation du noyau d'un thread utilisateur sur lequel les interfaces de thread Solaris et POSIX sont créées. Pour plus de facilité, DTrace exporte ces informations sous la même forme que les fichiers du système de fichiers `/proc` `/proc/`*pid*`/psinfo` et `/proc/`*pid*`/lwps/`*lwpid*`/lwpsinfo`. Les structures `/proc` sont utilisées par les outils d'observation et de débogage comme `ps(1)`, `pgrep(1)` et `truss(1)`. Elles sont définies dans le fichier d'en-tête du système `<sys/procfs.h>`, ainsi que sur la page de manuel `proc(4)`. Voici quelques exemples d'expressions utilisant `curpsinfo`, leurs types et leurs significations :

`curpsinfo->pr_pid`	`pid_t`	ID de processus courant
`curpsinfo->pr_fname`	`char []`	nom de fichier exécutable
`curpsinfo->pr_psargs`	`char []`	arguments de ligne de commande initiaux

Vous devez revoir la définition complète de la structure ultérieurement en examinant le fichier d'en-tête `<sys/procfs.h>` et les descriptions correspondantes dans `proc(4)`. L'exemple suivant utilise le membre `pr_psargs` pour identifier le processus qui nous intéresse en adaptant les arguments de la ligne de commande.

Les structs sont fréquemment utilisées pour créer des structures de données complexes dans les programmes en langage C. Par conséquent, l'aptitude à décrire et référencer les structs à partir du langage D offre une puissante capacité d'observation du fonctionnement interne du noyau du système d'exploitation Solaris et de ses interfaces système. En plus de l'utilisation de la struct

curpsinfo mentionnée ci-dessus, l'exemple suivant examine certaines structs du noyau en observant la relation entre le pilote ksyms(7D) et les demandes read(2). Le pilote utilise les deux structs usuelles uio(9S) et iovec(9S) pour répondre aux demandes de lecture à partir du fichier de périphérique caractère /dev/ksyms.

La struct uio, accessible via le nom struct uio ou l'alias de type uio_t, est décrite sur la page de manuel uio(9S) et permet de décrire une demande d'E/S impliquant la copie de données entre le noyau et un processus utilisateur. La struct uio contient en retour un tableau d'une ou plusieurs structures iovec(9S) qui décrivent chacune un élément de l'E/S demandée dans le cas où plusieurs éléments sont demandés à l'aide des appels système readv(2) ou writev(2). L'une des routines DDI (interface de pilote de périphérique) du noyau qui fonctionne sur struct uio est représentée par la fonction uiomove(9F), cette dernière constituant l'un des ensembles de fonctions que les pilotes du noyau utilisent pour répondre aux demandes read(2) du processus utilisateur et recopier les données sur les processus utilisateur.

Le pilote ksyms gère le fichier de périphérique caractère /dev/ksyms, ce dernier semblant être un fichier ELF contenant des informations sur la table de symboles du noyau. En fait, il s'agit d'une illusion créée par le pilote à l'aide de l'ensemble de modules actuellement chargés dans le noyau. Le pilote utilise la routine uiomove(9F) pour répondre aux demandes read(2). L'exemple suivant montre que les arguments et les appels de read(2) à partir de /dev/ksyms correspondent aux appels du pilote vers uiomove(9F) pour recopier les résultats dans l'espace d'adressage utilisateur à l'emplacement spécifié pour read(2).

Il est possible d'utiliser l'utilitaire strings(1) combiné à l'option -a pour forcer un ensemble de lecture à partir de /dev/ksyms. Essayez d'exécuter strings -a /dev/ksyms dans votre shell et voyez la sortie obtenue. Dans un éditeur, tapez la première clause de l'exemple de script et enregistrez-la dans le fichier ksyms.d:

```
syscall::read:entry
/curpsinfo->pr_psargs == "strings -a /dev/ksyms"/
{
    printf("read %u bytes to user address %x\n", arg2, arg1);
}
```

Cette première clause utilise l'expression curpsinfo->pr_psargs pour accéder et s'adapter aux arguments de la ligne de commande de notre commande strings(1) de sorte que le script sélectionne les demandes read(2) appropriées avant de suivre les arguments. Notez qu'en utilisant l'opérateur == avec un argument gauche correspondant à un tableau de char et un argument droit correspondant à une chaîne, le compilateur D déduit que l'argument gauche doit être promu en chaîne et qu'une comparaison de chaîne doit être exécutée. Tapez et exécutez la commande dtrace -q -s ksyms.d dans un shell, puis tapez la commande strings -a /dev/ksyms dans un autre shell. Tandis que strings(1) s'exécute, vous obtenez une sortie à partir de DTrace similaire à l'exemple suivant :

```
# dtrace -q -s ksyms.d
read 8192 bytes to user address 80639fc
read 8192 bytes to user address 80639fc
```

```
read 8192 bytes to user address 80639fc
read 8192 bytes to user address 80639fc
...
^C
#
```

Il est possible d'étendre cet exemple à l'aide d'une technique de programmation en D classique pour suivre un thread plus profondément dans le noyau à partir de cette demande `read(2)` initiale. Une fois entré dans le noyau avec `syscall::read:entry`, le script suivant définit une variable d'indication de thread local signalant le thread qui nous intéresse et efface cet indicateur sur `syscall::read:return`. Une fois l'indicateur défini, vous pouvez l'utiliser comme prédicat sur d'autres sondes pour instrumenter les fonctions du noyau comme `uiomove(9F)`. Le fournisseur de suivi des limites des fonctions de DTrace (`fbt`) édite les sondes d'entrée et revient aux fonctions définies dans le noyau, y compris celles de la DDI. Tapez le code source suivant. Ce dernier utilise le fournisseur `fbt` pour instrumenter `uiomove(9F)`. Enregistrez-le ensuite dans le fichier `ksyms.d`.

EXEMPLE 7–2 `ksyms.d` : suivi de la relation de`read(2)` et `uiomove(9F`)

```
/*
 * When our strings(1) invocation starts a read(2), set a watched flag on
 * the current thread.  When the read(2) finishes, clear the watched flag.
 */
syscall::read:entry
/curpsinfo->pr_psargs == "strings -a /dev/ksyms"/
{
    printf("read %u bytes to user address %x\n", arg2, arg1);
    self->watched = 1;
}

syscall::read:return
/self->watched/
{
    self->watched = 0;
}

/*
 * Instrument uiomove(9F).  The prototype for this function is as follows:
 * int uiomove(caddr_t addr, size_t nbytes, enum uio_rw rwflag, uio_t *uio);
 */
fbt::uiomove:entry
/self->watched/
{
    this->iov = args[3]->uio_iov;

    printf("uiomove %u bytes to %p in pid %d\n",
        this->iov->iov_len, this->iov->iov_base, pid);
}
```

La clause finale de l'exemple utilise la variable de thread local self->watched pour identifier à quel moment le thread de noyau qui nous intéresse lance la routine uiomove(9F) de la DDI. À ce stade, le script utilise le tableau args intégré pour permettre au quatrième argument (args[3]) d'accéder à uiomove(), cette fonction consistant en un pointeur vers la struct uio représentant la demande. Le compilateur D associe automatiquement chaque membre du tableau args au type qui correspond au prototype de fonction en langage C de la routine de noyau instrumentée. Le membre uio_iov contient un pointeur vers la struct iovec pour la demande. Une copie de ce pointeur est enregistrée pour être utilisée dans notre clause dans la variable de clause locale this->iov. Dans l'instruction finale, le script déréférence this->iov pour accéder aux membres de iovec iov_len et iov_base qui représentent respectivement la longueur en octets et l'adresse de base de la destination de uiomove(9F). Ces valeurs doivent correspondre aux paramètres d'entrée de l'appel système read(2) émis sur le pilote. Accédez à votre shell et exécutez dtrace -q -s ksyms.d. Entrez ensuite la commande strings -a /dev/ksyms dans un autre shell. Vous devez obtenir une sortie similaire à l'exemple suivant :

```
# dtrace -q -s ksyms.d
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
...
^C
#
```

Les adresses et les ID de processus sont différents dans votre sortie. Vous devez toutefois observer que les arguments en entrée vers read(2) correspondent aux paramètres transmis à uiomove(9F) par le pilote ksyms.

Unions

Les unions représentent une autre catégorie de type de composite que les langages ANSI-C et D prennent en charge. Elles sont également étroitement liées aux structs. Une union est un type de composite dans lequel un ensemble de membres de différents types sont définis et dans lequel les objets membres occupent tous la même zone de stockage. Une union est, par conséquent, un objet de type variante dans lequel seul un membre est valide à un moment donné en fonction de la manière dont l'union a été affectée. En règle générale, certaines autres variables ou éléments d'état servent à indiquer le membre de l'union actuellement valide. La taille d'une union correspond à la taille du membre le plus grand et l'alignement de la mémoire utilisé pour l'union correspond à l'alignement maximum requis par les membres de l'union.

La structure kstat de Solaris définit une struct contenant une union utilisée dans l'exemple suivant pour illustrer et observer les unions en C et D. La structure kstat sert à exporter un

ensemble de compteurs nommés représentant les statistiques du noyau comme l'utilisation de la mémoire et le débit d'E/S. La structure est utilisée pour implémenter des utilitaires comme mpstat(1M) et iostat(1M). Cette structure utilise struct kstat_named pour représenter un compteur nommé et sa valeur. Elle est définie comme suit :

```
struct kstat_named {
    char name[KSTAT_STRLEN]; /* name of counter */
    uchar_t data_type;    /* data type */
    union {
        char c[16];
        int32_t i32;
        uint32_t ui32;
        long l;
        ulong_t ul;
        ...
    } value;    /* value of counter */
};
```

La déclaration examinée est réduite à titre illustratif. La définition complète de la structure figure dans le fichier d'en-tête <sys/kstat.h> et est décrite dans kstat_named(9S). La déclaration ci-dessus est valide en langage ANSI-C et D et définit une struct dont l'un des membres est une valeur d'union dont les membres appartiennent à différents types en fonction du type de compteur. Étant donné que l'union elle-même est déclarée à l'intérieur d'un autre type, struct kstat_named, notez que le nom formel du type d'union est omis. Ce style de déclaration est connu comme étant une *union anonyme*. Le membre nommé value est un type d'union décrit par la déclaration précédente mais ce type d'union lui-même ne possède pas de nom car il n'a pas vocation à être utilisé autrement. Le membre data_type de la struct est affecté à une valeur qui indique le membre de l'union valide pour chaque objet du type struct kstat_named. Un ensemble de jetons de préprocesseur en C est défini pour les valeurs de data_type. Par exemple, le jeton KSTAT_DATA_CHAR est égal à 0 et indique que le membre value.c se trouve à l'endroit où est actuellement enregistrée la valeur.

L'Exemple 7–3 montre comment accéder à l'union kstat_named.value en assurant le suivi d'un processus utilisateur. Il est possible d'échantillonner les compteurs kstat à partir d'un processus utilisateur à l'aide de la fonction kstat_data_lookup(3KSTAT) qui renvoie un pointeur à une struct kstat_named. L'utilitaire mpstat(1M) appelle cette fonction à plusieurs reprises pendant son exécution afin d'échantillonner les dernières valeurs du compteur. Accédez à votre shell, essayez d'exécuter mpstat 1 et observez la sortie. Appuyez sur Control-C dans votre shell pour abandonner mpstat après quelques secondes. Pour observer l'échantillonnage du compteur, nous aimerions activer une sonde qui se déclenche à chaque fois que la commande mpstat appelle la fonction kstat_data_lookup(3KSTAT) dans libkstat. Pour ce faire, nous allons utiliser un nouveau fournisseur de DTrace : pid. Le fournisseur pid vous permet de créer des sondes de manière dynamique au sein de processus utilisateur au niveau des emplacements de symboles C comme les points d'entrée de fonction. Vous pouvez demander au fournisseur pid de créer une sonde au niveau de l'entrée d'une fonction utilisateur et de retourner les sites en décrivant les sondes comme suit :

pid*ID-processus*:*nom-objet*:*nom-fonction*:entry
pid*ID-processus*:*nom-objet*:*nom-fonction*:return

Par exemple, pour créer une sonde dans l'ID de processus 12345 qui se déclenche lors de l'entrée dans kstat_data_lookup(3KSTAT), vous écririez la description de sonde suivante :

```
pid12345:libkstat:kstat_data_lookup:entry
```

Le fournisseur pid insère l'instrumentation dynamique dans le processus utilisateur spécifié à l'emplacement du programme correspondant à la description de la sonde. L'implémentation de la sonde contraint chaque thread utilisateur atteignant l'emplacement du programme instrumenté à s'interrompre dans le noyau du système d'exploitation et à lancer DTrace, déclenchant ainsi la sonde correspondante. Par conséquent, bien que l'emplacement d'instrumentation soit associé à un processus utilisateur, les prédicats et les actions de DTrace que vous spécifiez continuent de s'exécuter dans le contexte du noyau du système d'exploitation. Le fournisseur pid est décrit plus en détails dans le Chapitre30, “Fournisseur pid”.

Plutôt que de devoir éditer la source de votre programme en D chaque fois que vous souhaitez appliquer votre programme à un processus différent, vous pouvez insérer des identifiants appelés *variables de macro* dans votre programme. Ces identificateurs sont évalués à chaque compilation et remplacement de votre programme par des arguments de ligne de commande dtrace supplémentaires. Les variables de macro sont spécifiées à l'aide du signe dollar $ suivi d'un identifiant ou d'un chiffre. Si vous exécutez la commande dtrace -s *script* foo bar baz, le compilateur D définit automatiquement les variables de macro $1, $2, et $3 sur les jetons foo, bar et baz respectivement. Vous pouvez utiliser des variables de macro dans des expressions de programme en D ou des descriptions de sonde. Par exemple, les descriptions de sonde suivantes sont instrumentées quel que soit l'ID de processus spécifié comme argument supplémentaire vers dtrace:

```
pid$1:libkstat:kstat_data_lookup:entry
{
    self->ksname = arg1;
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 != NULL/
{
    this->ksp = (kstat_named_t *)copyin(arg1, sizeof (kstat_named_t));
    printf("%s has ui64 value %u\n", copyinstr(self->ksname),
        this->ksp->value.ui64);
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 == NULL/
{
```

```
    self->ksname = NULL;
}
```

Les variables de macro et les scripts réutilisables sont décrits plus en détails dans le Chapitre15, “Scripts”. Maintenant que nous savons comment instrumenter des processus utilisateur à l'aide de leur ID de processus, revenons aux unions d'échantillonnage. Ouvrez votre éditeur, tapez le code source de votre exemple complet et enregistrez-le dans un fichier nommé `kstat.d`:

EXEMPLE 7–3 `kstat.d` : appels de suivi vers `kstat_data_lookup`(3KSTAT)

```
pid$1:libkstat:kstat_data_lookup:entry
{
    self->ksname = arg1;
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 != NULL/
{
    this->ksp = (kstat_named_t *) copyin(arg1, sizeof (kstat_named_t));
    printf("%s has ui64 value %u\n",
        copyinstr(self->ksname), this->ksp->value.ui64);
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 == NULL/
{
    self->ksname = NULL;
}
```

Accédez à présent à l'un de vos shells et exécutez la commande `mpstat 1` pour lancer `mpstat`(1M) dans un mode dans lequel elle échantillonne les statistiques et les signale à la cadence d'une par seconde. Une fois `mpstat` en cours d'exécution, exécutez la commande `dtrace -q -s kstat.d ‘pgrep mpstat‘` dans votre autre shell. Vous obtenez une sortie correspondant aux statistiques en cours d'accès. Appuyez sur Control-C pour abandonner `dtrace` et revenir à l'invite du shell.

```
# dtrace -q -s kstat.d ‘pgrep mpstat‘
cpu_ticks_idle has ui64 value 41154176
cpu_ticks_user has ui64 value 1137
cpu_ticks_kernel has ui64 value 12310
cpu_ticks_wait has ui64 value 903
hat_fault has ui64 value 0
as_fault has ui64 value 48053
maj_fault has ui64 value 1144
xcalls has ui64 value 123832170
intr has ui64 value 165264090
intrthread has ui64 value 124094974
```

```
pswitch has ui64 value 840625
inv_swtch has ui64 value 1484
cpumigrate has ui64 value 36284
mutex_adenters has ui64 value 35574
rw_rdfails has ui64 value 2
rw_wrfails has ui64 value 2
...
^C
#
```

Si vous capturez la sortie dans chaque fenêtre de terminal et que vous retirez chaque valeur de la valeur signalée par l'itération précédente par l'intermédiaire des statistiques, vous devriez pouvoir faire correspondre la sortie `dtrace` avec la sortie de `mpstat.` Le programme donné en exemple enregistre le pointeur du nom de compteur en entrée dans la fonction de recherche, puis exécute la majeure partie du travail de suivi en retour à partir de `kstat_data_lookup`(3KSTAT) Les fonctions en D intégrées `copyinstr()` et `copyin()` recopient les résultats du processus utilisateur dans DTrace lorsque `arg1` (valeur retournée) n'est pas `NULL`. Une fois que les données de `kstat` ont été copiées, l'exemple signale la valeur de compteur de `ui64` à partir de l'union. Cet exemple simplifié suppose que `mpstat` échantillonne les compteurs utilisant le membre `value.ui64.` Essayez, sous forme d'exercice, de recoder `kstat.d` pour utiliser des prédicats et d'imprimer le membre de l'union correspondant au membre `data_type.` Vous pouvez également essayer de créer une version de `kstat.d` qui calcule la différence entre des valeurs de données successives et offre généralement une sortie similaire à `mpstat`.

Tailles des membres et décalages

Vous pouvez déterminer la taille en octets de tout type ou expression en D, y compris une struct ou une union, à l'aide de l'opérateur `sizeof.` Vous pouvez appliquer l'opérateur `sizeof` à une expression ou à un nom d'un type placé entre parenthèses, comme illustré dans les deux exemples suivants :

`sizeof` *expression*   `sizeof (`*type-name*`)`

Par exemple, l'expression `sizeof (uint64_t)` doit retourner la valeur 8 tandis que l'expression `sizeof (callinfo.ts)` doit retourner la valeur 8 si elle est insérée dans le code source de notre exemple de programme ci-dessus. Le type de retour formel de l'opérateur `sizeof` est l'alias de type `size_t`. Ce dernier est défini comme un nombre entier non signé de la même taille qu'un pointeur dans le modèle de données courant et sert à représenter des comptes d'octets. Lorsque l'opérateur `sizeof` est appliqué à une expression, cette dernière est validée par le compilateur D mais la taille de l'objet qui en résulte est calculée au moment de la compilation et aucun code n'est généré pour l'expression. Vous pouvez utiliser `sizeof` chaque fois qu'une constante entière est requise.

Vous pouvez utiliser l'opérateur d'accompagnement `offsetof` pour déterminer le décalage en octets d'un membre de la struct ou de l'union depuis le début du stockage par rapport à n'importe quel objet du type de struct ou d'union. L'opérateur `offsetof` est utilisé dans une expression semblable à celle-ci :

```
offsetof (type-name, member-name)
```

Dans cet exemple, *type-name* correspond au nom d'un type de struct ou d'union ou d'un alias de type et *member-name* correspond au nom d'identification d'un membre de cette struct ou union. De même que `sizeof`, la fonction `offsetof` renvoie une valeur `size_t`. Vous pouvez l'utiliser dans un programme en D dès l'instant qu'une constante entière peut être utilisée.

Champs de bit

Le langage D permet également de définir les membres entiers d'une struct ou d'une union constitués d'un nombre arbitraire de bits, les *champs de bit*. Pour déclarer un champ de bit, vous devez spécifier un type de base de nombre entier signé ou non signé, un nom de membre et un suffixe indiquant le nombre de bits à affecter au champ, comme illustré dans l'exemple suivant :

```
struct s {
    int a : 1;
    int b : 3;
    int c : 12;
};
```

La largeur du champ de bit est une constante entière séparée du nom du membre par le symbole : à droite. La largeur du champ de bit doit être positive et doit correspondre à un nombre de bits qui n'excède pas la largeur du type de base de nombre entier correspondant. Il n'est pas possible de déclarer en D les champs de bit dont la largeur est supérieure à 64 bits. Les champs de bit en D assurent la compatibilité et l'accès aux fonctionnalités ANSI-C correspondantes. Les champs de bits sont généralement utilisés dans des situations dans lesquelles le stockage en mémoire est crucial ou lorsque la disposition d'une struct doit correspondre à la disposition d'un registre matériel.

Un champ de bit est un constructeur de compilateur qui automatise la disposition d'un nombre entier et un ensemble de masques pour extraire la valeur des membres. Vous pouvez obtenir le même résultat en définissant simplement les masques vous-même et en utilisant l'opérateur `&`. Les compilateurs C et D essayent d'empiler les bits aussi efficacement que possible, mais ils sont libres d'exécuter cette action comme ils le souhaitent. Par conséquent, les champs de bit ne garantissent pas l'obtention de dispositions de bit identiques d'un compilateur ou d'une architecture à l'autre. Si une disposition de bit stable est requise, vous devez construire les masques de bit vous-même et extraire les valeurs à l'aide de l'opérateur `&`.

Vous pouvez accéder à un membre de champ de bit en spécifiant simplement son nom avec les opérateurs “`.`” ou `->` comme tout autre membre d'une struct ou d'une union. Le champ de bit est automatiquement promu vers le type de nombre entier le plus grand à utiliser dans les

expressions. Le stockage des champs de bit ne s'alignant pas sur une limite d'octets ou leur taille ne correspondant pas à un nombre entier rond, vous ne pouvez pas appliquer les opérateurs `sizeof` ou `offsetof` à un membre de champ de bit. Le compilateur D ne vous permet pas non plus de récupérer l'adresse d'un membre de champ de bit à l'aide de l'opérateur &.

CHAPITRE 8

Définitions des types et des constantes

Ce chapitre décrit la méthode de déclaration des alias de type et des constantes nommées en langage D. Il présente également la gestion de l'espace de noms et des types en langage D pour les identificateurs et les types de système d'exploitation et de programme.

Typedef

Le mot-clé `typedef` permet de déclarer un identificateur en tant qu'alias d'un type existant. Comme toutes les déclarations de type en langage D, le mot-clé `typedef` est utilisé en dehors des clauses de sonde dans une déclaration se présentant sous la forme suivante :

```
typedef existing-type new-type ;
```

existing-type correspondant à une déclaration de type et *new-type* à un identificateur à utiliser comme alias de ce type. Par exemple, la déclaration :

```
typedef unsigned char uint8_t;
```

est utilisée en interne par le compilateur D pour créer l'alias de type `uint8_t`. Chaque fois que vous pouvez utiliser un type normal (comme le type d'une variable, d'une valeur de tableau associatif ou d'un membre du tuple), vous pouvez utiliser les alias de type. Vous pouvez également combiner `typedef` avec des déclarations plus élaborées comme la définition d'une nouvelle `struct` :

```
typedef struct foo {
    int x;
    int y;
} foo_t;
```

Dans cet exemple, `struct foo` est défini comme le même type que son alias, `foo_t`. Les en-têtes du système en langage C de Solaris utilisent fréquemment le suffixe `_t` pour indiquer un alias `typedef`.

Énumérations

La définition de noms symboliques de constantes dans un programme simplifie le processus de lisibilité et de mise à jour ultérieure du programme. Une méthode consiste à définir une *énumération*, qui associe un ensemble de nombres entiers à un ensemble d'identificateurs, appelés énumérateurs, que le compilateur reconnaît et remplace par la valeur entière correspondante. Une énumération est définie à l'aide d'une déclaration comme suit :

```
enum colors {
    RED,
    GREEN,
    BLUE
};
```

Le premier énumérateur de l'énumération, `RED`, reçoit la valeur zéro et chaque identificateur ultérieur reçoit la valeur entière suivante. Vous pouvez également attribuer une valeur entière spécifique à un énumérateur en lui donnant un suffixe comprenant un signe égal (=) et une constante entière, comme dans l'exemple suivant :

```
enum colors {
    RED = 7,
    GREEN = 9,
    BLUE
};
```

Le compilateur assigne à l'énumérateur `BLUE` la valeur 10, car aucune n'est spécifiée et que l'énumérateur précédent est défini sur 9. Une fois l'énumération définie, vous pouvez utiliser les énumérateurs dans tout programme en D où une constante entière peut être utilisée. Par ailleurs, l'énumération `enum colors` est également définie comme un type équivalant à `int`. Le compilateur D permettra également d'utiliser une variable de type `enum` chaque fois que vous pouvez utiliser `int`, ainsi que d'affecter une valeur entière à une variable de type `enum`. Vous pouvez également omettre le nom `enum` dans la déclaration si le nom du type n'est pas requis.

Les énumérateurs sont visibles dans toutes les clauses et déclarations ultérieures de votre programme, de sorte que vous pouvez définir le même identificateur d'énumérateur dans plusieurs énumérations. Vous pouvez, cependant, définir plusieurs énumérateurs possédant la même valeur dans la même ou dans différentes énumérations. Vous pouvez également affecter à une variable de type énumération des nombres entiers ne possédant aucun énumérateur correspondant.

La syntaxe d'énumération en D est identique à la syntaxe correspondante en ANSI-C. Le langage D permet également d'accéder aux énumérations définies dans le noyau du système et ses modules chargeables. Ces énumérateurs ne sont toutefois pas visibles globalement dans votre programme en D. Les énumérateurs du noyau ne sont visibles que lorsqu'ils sont utilisés comme argument vers les opérateurs de comparaison binaire dans le cadre d'une comparaison avec un objet du type d'énumération correspondant. Par exemple, la fonction `uiomove(9F)` possède un paramètre de type `enum uio_rw` défini comme suit :

```
enum uio_rw { UIO_READ, UIO_WRITE };
```

Les énumérateurs `UIO_READ` et `UIO_WRITE` ne sont logiquement pas visibles dans votre programme en D. Vous pouvez toutefois leur conférer une visibilité globale en en comparant un avec une valeur de type `enum uio_rw`, comme illustré dans l'exemple de clause suivant :

```
fbt::uiomove:entry
/args[2] == UIO_WRITE/
{
    ...
}
```

Cet exemple suit les appels à la fonction `uiomove(9F)` pour les demandes d'écriture en comparant `args[2]`, une variable de type `enum uio_rw`, à l'énumérateur `UIO_WRITE`. Comme l'argument de gauche est un type d'énumération, le compilateur D recherche l'énumération lorsqu'il essaye de résoudre l'identificateur de droite. Cette fonction protège vos programmes en D des conflits de noms d'identificateurs avec les nombreuses énumérations définies dans le noyau du système d'exploitation.

Inlines

Vous pouvez également définir les constantes nommées en D à l'aide de directives `inline`. Ces dernières fournissent une méthode plus générale de création d'identificateurs qui sont remplacés par des valeurs ou des expressions prédéfinies pendant la compilation. Les directives intégrées constituent une forme plus puissante de remplacement vertical que la directive `#define` fournie par le préprocesseur C, le type de remplacement étant réel et le remplacement étant exécuté au moyen de l'arborescence de syntaxe compilée au lieu de l'être tout simplement au moyen d'un ensemble de jetons lexicaux. Vous spécifiez une directive intégrée à l'aide d'une déclaration comme suit :

```
inline type name = expression ;
```

type correspondant à la déclaration de type d'un type existant, *name* à un identificateur en D valide non précédemment défini en tant que variable intégrée ou globale et *expression* à une expression en D valide. Une fois la directive intégrée traitée, le compilateur D substitue la forme compilée de *expression* de chaque instance ultérieure de *name* dans la source du programme. Par exemple, le programme en D suivant doit suivre la chaîne "`hello`" et la valeur entière `123` :

```
inline string hello = "hello";
inline int number = 100 + 23;

BEGIN
{
    trace(hello);
    trace(number);
}
```

Vous pouvez utiliser un nom intégré chaque fois que vous pouvez utiliser une variable globale du type correspondant. Si vous pouvez évaluer l'expression intégrée en une constante de chaîne ou entière au moment de la compilation, vous pouvez également utiliser le nom en ligne dans des contextes nécessitant des expressions constantes, comme les dimensions de tableau scalaire.

L'expression intégrée est validée pour les erreurs de syntaxe dans le cadre de l'évaluation de la directive. Le type qui résulte de l'expression doit être compatible avec le type défini dans la variable intégrée, conformément aux règles appliquées à l'opérateur d'affectation en D (=). Une expression intégrée peut ne pas référencer l'identificateur intégré lui-même : les définitions récursives ne sont pas autorisées.

Les packages logiciels de DTrace installent plusieurs fichiers sources en D dans le répertoire système `/usr/lib/dtrace` qui contient les directives intégrées que vous pouvez utiliser dans vos programmes. Par exemple, la bibliothèque `signal.d` comprend des directives qui se présentent comme suit :

```
inline int SIGHUP = 1;
inline int SIGINT = 2;
inline int SIGQUIT = 3;
...
```

Ces définitions intégrées permettent d'accéder à l'ensemble actuel des noms de signaux Solaris décrits dans `signal`(3HEAD). La bibliothèque `errno.d` contient, de façon similaire, des directives intégrées pour les constantes `errno` en C décrites dans `Intro(2)`.

Par défaut, le compilateur D intègre automatiquement tous les fichiers de bibliothèque en D fournis de sorte que vous pouvez utiliser ces définitions dans n'importe quel programme en D.

Espaces de noms de types

Cette section présente les espaces de noms en D et les problèmes d'espace de noms liés aux types. Dans les langages classiques comme l'ANSI-C, la visibilité du type est déterminée par le type d'imbrication (à l'intérieur d'une fonction ou d'une autre déclaration). Les types déclarés dans l'étendue extérieure d'un programme en C sont associés à un espace de noms global unique et sont visibles à travers tout le système. Les types définis dans les fichiers d'en-tête en C figurent généralement dans cette étendue extérieure. Contrairement à ces langages, le langage D permet d'accéder aux types à partir de plusieurs étendues extérieures.

Le langage D facilite l'observation dynamique à travers plusieurs couches d'une pile logicielle, y compris le noyau du système d'exploitation, un ensemble associé de modules de noyau chargeables et les processus utilisateur en cours d'exécution sur le système. Un seul programme en D peut instancier des sondes pour rassembler des données issues de plusieurs modules de noyau ou d'autres entités logicielles qui sont compilées dans des objets binaires indépendants. Par conséquent, plusieurs types de données possédant le même nom mais pas nécessairement la même définition, peuvent figurer parmi les types disponibles pour DTrace et le compilateur D.

Pour gérer cette situation, le compilateur D associe chaque type à un espace de noms identifié par l'objet de programme qu'il contient. Les types d'un objet de programme particulier sont accessibles en spécifiant le nom de l'objet et en utilisant l'opérateur d'étendue ‘ dans n'importe quel nom de type.

Par exemple, si un module de noyau nommé `foo` contient la déclaration de type en langage C suivante :

```
typedef struct bar {
    int x;
} bar_t;
```

les types `struct bar` et `bar_t` sont accessibles à partir du langage D à l'aide des noms de type :

```
struct foo‘bar                foo‘bar_t
```

Vous pouvez utiliser l'opérateur ‘ dans n'importe quel contexte dès lors qu'un nom de type est approprié, y compris lors de la spécification du type pour les déclarations de variable en D ou les expressions de diffusion dans des clauses de sonde en D.

Le compilateur D intègre également deux espaces de noms de types spéciaux qui portent respectivement les noms `C` et `D.` L'espace de noms de types C comprend à l'origine les types ANSI-C standard intrinsèques comme `int`. Par ailleurs, les définitions de type acquises à l'aide du préprocesseur C cpp(1) au moyen de l'option `dtrace -C` sont traitées par et ajoutées à l'étendue de C. Vous pouvez ainsi inclure des fichiers d'en-tête en C contenant des déclarations de types déjà visibles dans un autre espace de noms de types sans engendrer d'erreur de compilation.

L'espace de noms de types D comprend à l'origine les types D intrinsèques comme `int` et `string`, ainsi que des alias de types D intégrés comme `uint32_t`. Toute nouvelle déclaration de types qui apparaît dans la source du programme en D est automatiquement ajoutée à l'espace de noms de types D. Si vous créez dans votre programme en D un type complexe, comme une `struct`, constitué de types de membre provenant d'autres espaces de noms, les types de membre sont copiés dans l'espace de noms D par la déclaration.

Lorsque le compilateur D rencontre une déclaration de type ne spécifiant aucun espace de noms de manière explicite à l'aide de l'opérateur ‘, le compilateur recherche l'ensemble des espaces de noms de types actifs pour trouver une correspondance à l'aide du nom de type spécifié. L'espace de noms C est toujours recherché en premier, suivi de l'espace de noms D. Si le nom du type n'est trouvé ni dans l'espace de noms C ni dans l'espace de noms D, la recherche des espaces de noms de types des modules de noyau actifs est réalisée dans l'ordre ascendant par ID de module de noyau. Cet ordre garantit que la recherche des objets binaires qui constitue le noyau de base est exécutée avant la recherche des modules de noyau chargeables. Par contre, il ne garantit pas l'ordre des propriétés au niveau des modules chargeables. Vous devez utiliser l'opérateur d'étendue lorsque vous accédez aux types définis dans les modules de noyau chargeables pour éviter les conflits de nom de type avec les autres modules de noyau.

Le compilateur D utilise les informations de débogage en ANSI-C compressées fournies avec les modules de noyau de base de Solaris afin d'accéder automatiquement au code source du système d'exploitation sans devoir accéder aux fichiers en C inclus correspondants. Ces informations de débogage symboliques peuvent ne pas être disponibles pour tous les modules de noyau sur votre système. Le compilateur D signale une erreur si vous tentez d'accéder à un type dans un espace de noms d'un module qui empile les informations de débogage en C compressées dont l'utilisation est prévue avec DTrace.

CHAPITRE 9

Groupements

Lors de l'instrumentation du système en vue de répondre aux questions portant sur les problèmes de performance, il est utile d'envisager comment le groupement de données permet de répondre à une question spécifique, plutôt que de raisonner en termes de rassemblement de données par des sondes individuelles. Par exemple, si vous souhaitez connaître le nombre d'appels système par ID utilisateur, vous n'avez pas nécessairement besoin de connaître les données recueillies au niveau de *chaque* appel système. Seule la consultation d'un tableau des ID utilisateur et des appels système présente pour vous un intérêt. Auparavant, pour répondre à cette question, vous deviez rassembler les données obtenues au niveau de chaque appel système pour procéder ensuite au post-traitement des données à l'aide d'un outil comme `awk(1)` ou `perl(1)`. Cependant, dans DTrace, le groupement de données constitue une opération de première classe. Ce chapitre décrit les fonctions de DTrace permettant de manipuler les *groupements*.

Fonctions de groupement

Une *fonction de groupement* se caractérise par les propriétés suivantes :

$$f(f(x_0) \cup f(x_1) \cup \ldots \cup f(x_n)) = f(x_0 \cup x_1 \cup \ldots \cup x_n)$$

où x_n est un jeu de données arbitraires. En bref, appliquer une fonction de groupement à des sous-ensembles puis la réappliquer aux résultats obtenus revient à l'appliquer à l'intégralité des données. Par exemple, prenons une fonction `SUM` permettant d'additionner un jeu de données. Si les données brutes se composent des éléments {2, 1, 2, 5, 4, 3, 6, 4, 2}, on obtient {29} en appliquant la fonction `SUM` à l'intégralité du jeu. De la même façon, l'application de `SUM` au sous-ensemble comprenant les trois premiers éléments donne comme résultat {5}, l'application de `SUM` à l'ensemble comprenant les trois éléments suivants donne comme résultat {12} et l'application de `SUM` aux trois éléments restants donne également comme résultat {12}. `SUM` est une fonction de groupement car son application à l'ensemble formé par les résultats {5, 12, 12} donne le même résultat, {29}, que l'application de `SUM` aux données d'origine.

Toutes les fonctions ne sont pas des fonctions de groupement. C'est notamment le cas de la fonction `MEDIAN` : elle détermine la valeur médiane d'un jeu de données. (La médiane se caractérise comme l'élément d'un ensemble ayant une valeur telle qu'il existe autant d'éléments dont la valeur est supérieure à la sienne que d'éléments dont la valeur est inférieure à la sienne.) La fonction `MEDIAN` est calculée en triant l'ensemble et en sélectionnant l'élément central. Si l'on revient aux données brutes d'origine, l'application de la fonction `MEDIAN` à l'ensemble comprenant les trois premiers éléments donne comme résultat {2}. (L'ensemble trié donne {1, 2, 2} ; {2} constitue l'élément central de l'ensemble.) De même, l'application de la fonction `MEDIAN` aux trois éléments suivants donne comme résultat {4} et son application aux trois derniers éléments donne comme résultat {4}. L'application de `MEDIAN` à chacun des sous-ensembles donne donc {2, 4, 4}. Le résultat donné par l'application de `MEDIAN` à cet ensemble est {4}. Toutefois, le tri de l'ensemble d'origine donne comme résultat {1, 2, 2, 2, 3, 4, 4, 5, 6}. L'application de `MEDIAN` à cet ensemble donne donc comme résultat {3}. Étant donné que les résultats ne correspondent pas, `MEDIAN` n'est pas une fonction de groupement.

De nombreuses fonctions courantes permettant d'interpréter un jeu de données sont des fonctions de groupement. Ces fonctions permettent entre autres de compter le nombre d'éléments dans un ensemble, de calculer les valeurs minimale et maximale d'un ensemble et d'additionner l'ensemble des éléments au sein de cet ensemble. Il est possible de déterminer la signification arithmétique à partir de la fonction permettant de compter le nombre d'éléments dans l'ensemble et celle permettant d'additionner le nombre d'éléments dans l'ensemble.

Toutefois, plusieurs fonctions utiles ne sont pas des fonctions de groupement, comme par exemple les fonctions permettant de calculer le mode (l'élément le plus courant) d'un ensemble, sa valeur médiane ou son écart type.

L'application de fonctions de groupement à des données alors qu'elles font l'objet d'un suivi présente de nombreux avantages :

- Il n'est pas nécessaire de stocker l'intégralité des données. Lorsqu'un nouvel élément doit être ajouté à l'ensemble, la fonction de groupement est calculée en fonction de l'ensemble comprenant le résultat intermédiaire actuel et le nouvel élément. Après le calcul du nouveau résultat, le nouvel élément peut être ignoré. Ce processus réduit la quantité de stockage requise par un facteur du nombre de points de données, souvent très élevé.
- La collection de données ne provoque pas de problèmes d'évolutivité pathologiques. Les fonctions de groupement permettent de conserver les résultats intermédiaires *par CPU* plutôt que dans une structure de données partagées. DTrace applique ensuite la fonction de groupement à l'ensemble comprenant les résultats intermédiaires par CPU pour produire le résultat final à l'échelle du système.

Groupements

DTrace stocke les résultats des fonctions de groupement dans des objets appelés *groupements*. Les résultats des groupements sont indexés avec un tuple d'expressions similaires à celles utilisées pour les tableaux associatifs. En D, la syntaxe d'un groupement est la suivante :

```
@name[ keys ] = aggfunc ( args );
```

où *name* est le nom du groupement, *keys* est une liste d'expressions D, séparées par des virgules, *aggfunc* est l'une des fonctions de groupement DTrace et *args* est une liste d'arguments séparés par des virgules, correspondant à la fonction de groupement. Le groupement *name*est un identificateur D ayant comme préfixe le caractère spécial `@`. Tous les groupements de vos programmes D sont des variables globales ; il n'existe pas de groupement local de clause ou de thread. Les noms de groupement sont conservés dans un espace de noms d'identificateurs séparé des autres variables globales D. N'oubliez pas que `a` et `@a` ne sont pas la même variable si vous réutilisez les noms. Le nom de groupement spécial `@` peut être utilisé pour nommer un groupement anonyme avec des programmes D simples. Le compilateur D traite ce nom en tant qu'alias du nom de groupement `@_`.

Les fonctions de groupement DTrace sont affichées dans le tableau suivant. La plupart des fonctions de groupement ne prennent qu'un seul argument représentant la nouvelle donnée.

TABLEAU 9–1 Fonctions de groupement DTrace

Nom de la fonction	Arguments	Résultat
`count`	aucun	Nombre d'appels.
`sum`	expression scalaire	Valeur totale des expressions spécifiées.
`avg`	expression scalaire	Moyenne arithmétique des expressions spécifiées.
`min`	expression scalaire	Valeur la plus faible parmi les expressions spécifiées.
`max`	expression scalaire	Valeur la plus élevée parmi les expressions spécifiées.
`lquantize`	expression scalaire, limite inférieure, limite supérieure, valeur d'étape	Répartition linéaire des fréquences, comprises dans la plage spécifiée, des valeurs des expressions spécifiées. Incrémente la valeur dans le compartiment *inférieur le plus proche* de l'expression spécifiée.
`quantize`	expression scalaire	Répartition des fréquences multiples de deux des valeurs des expressions spécifiées. Incrémente la valeur dans le compartiment multiple de deux *inférieur le plus proche* de l'expression spécifiée.

Par exemple, pour compter le nombre d'appels système `write(2)` dans le système, vous pouvez utiliser une chaîne informative en tant que clé ainsi que la fonction de groupement `count()` :

```
syscall::write:entry
{
    @counts["write system calls"] = count();
}
```

La commande dtrace affiche les résultats du groupement par défaut lorsque le processus se termine, soit en tant que résultat d'une action explicite END, soit lorsque l'utilisateur appuie sur Control-C. L'exemple de sortie suivant montre le résultat de l'exécution de cette commande, après une attente de quelques secondes et après avoir appuyé sur Control-C :

```
# dtrace -s writes.d
dtrace: script './writes.d' matched 1 probe
^C

  write system calls                                              179
#
```

Vous pouvez compter les appels système par nom de processus en utilisant la variable execname en tant que clé à un groupement :

```
syscall::write:entry
{
    @counts[execname] = count();
}
```

L'exemple de sortie suivant montre le résultat de l'exécution de cette commande, après une attente de quelques secondes et après avoir appuyé sur Control-C :

```
# dtrace -s writesbycmd.d
dtrace: script './writesbycmd.d' matched 1 probe
^C

  dtrace                                                            1
  cat                                                               4
  sed                                                               9
  head                                                              9
  grep                                                             14
  find                                                             15
  tail                                                             25
  mountd                                                           28
  expr                                                             72
  sh                                                              291
  tee                                                             814
  def.dir.flp                                                    1996
  make.bin                                                       2010
#
```

Vous pouvez également vouloir examiner de manière plus approfondie les saisies, organisées par nom exécutable et descripteur de fichier. Le descripteur de fichier est le premier argument de `write(2)` ; l'exemple suivant utilise donc une clé comprenant à la fois `execname` et `arg0` :

```
syscall::write:entry
{
    @counts[execname, arg0] = count();
}
```

L'exécution de cette commande génère un tableau comprenant le nom exécutable et le descripteur de fichier, comme illustré dans l'exemple suivant :

```
# dtrace -s writesbycmdfd.d
dtrace: script './writesbycmdfd.d' matched 1 probe
^C

  cat                                                     1      58
  sed                                                     1      60
  grep                                                    1      89
  tee                                                     1     156
  tee                                                     3     156
  make.bin                                                5     164
  acomp                                                   1     263
  macrogen                                                4     286
  cg                                                      1     397
  acomp                                                   3     736
  make.bin                                                1     880
  iropt                                                   4    1731
#
```

L'exemple suivant affiche le temps moyen écoulé lors de l'appel système en écriture, par nom de processus. Cet exemple utilise la fonction de groupement `avg()`, en spécifiant l'expression sur laquelle effectuer le calcul de la moyenne en tant qu'argument. L'exemple fait la moyenne du temps écoulé dans l'appel système :

```
syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
/self->ts/
{
    @time[execname] = avg(timestamp - self->ts);
    self->ts = 0;
}
```

L'exemple de sortie suivant montre le résultat de l'exécution de cette commande, après une attente de quelques secondes et après avoir appuyé sur Control-C :

```
# dtrace -s writetime.d
dtrace: script './writetime.d' matched 2 probes
^C

  iropt                                                          31315
  acomp                                                          37037
  make.bin                                                       63736
  tee                                                            68702
  date                                                           84020
  sh                                                             91632
  dtrace                                                        159200
  ctfmerge                                                      321560
  install                                                       343300
  mcs                                                           394400
  get                                                           413695
  ctfconvert                                                    594400
  bringover                                                    1332465
  tail                                                         1335260
#
```

La moyenne peut s'avérer utile, mais c'est un élément qui ne fournit pas suffisamment de détails pour comprendre la répartition des points de données. Pour mieux comprendre la répartition, utilisez la fonction de groupement `quantize()`, comme illustré dans l'exemple suivant :

```
syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
/self->ts/
{
    @time[execname] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

Étant donné que chaque ligne de sortie devient un diagramme de répartition de fréquences, ce script affiche une sortie nettement plus longue que les précédents : L'exemple suivant présente une sélection de la sortie de test :

```
  lint
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |                                         2
           32768 |                                         0
```

```
          65536 |@@@@@@@@@@@@@@@@@@@                      74
         131072 |@@@@@@@@@@@@@@@                          59
         262144 |@@@                                      14
         524288 |                                         0

  acomp
          value  ------------- Distribution ------------- count
           4096 |                                         0
           8192 |@@@@@@@@@@@@                             840
          16384 |@@@@@@@@@@@                              750
          32768 |@@                                       165
          65536 |@@@@@@                                   460
         131072 |@@@@@@                                   446
         262144 |                                         16
         524288 |                                         0
        1048576 |                                         1
        2097152 |                                         0

  iropt
          value  ------------- Distribution ------------- count
           4096 |                                         0
           8192 |@@@@@@@@@@@@@@@@@@@@@@@                  4149
          16384 |@@@@@@@@@@                               1798
          32768 |@                                        332
          65536 |@                                        325
         131072 |@@                                       431
         262144 |                                         3
         524288 |                                         2
        1048576 |                                         1
        2097152 |                                         0
```

Vous pouvez remarquer que les lignes de la répartition de fréquences sont *toujours* des multiples de deux. Chaque ligne indique le compte du nombre d'éléments *supérieurs ou égaux* à la valeur correspondante mais *inférieurs* à la valeur supérieure la plus proche. Par exemple, le résultat ci-dessus montre que `iropt` a 4,149 écritures prenant entre 8,192 nanosecondes et 16,383 nanosecondes incluses.

La fonction `quantize()` permet d'obtenir un aperçu rapide des données, mais vous souhaitez peut-être observer à la place une répartition linéaire des valeurs. Pour afficher une répartition linéaire des valeurs, utilisez la fonction de groupement `lquantize()`. La fonction `lquantize()` prend trois arguments, en plus d'une expression D : une limite inférieure, une limite supérieure et une étape. Par exemple, si vous souhaitez observer la répartition des écritures par descripteur de fichier, une quantification de multiples de deux n'est pas efficace. À la place, utilisez une quantification linéaire avec une plage restreinte, comme illustré dans l'exemple suivant :

```
syscall::write:entry
{
```

```
    @fds[execname] = lquantize(arg0, 0, 100, 1);
}
```

L'exécution de ce script pendant plusieurs secondes permet d'obtenir une grande quantité d'informations. L'exemple suivant montre une sélection de sortie classique :

```
  mountd
           value  ------------- Distribution ------------- count
              11 |                                         0
              12 |@                                        4
              13 |                                         0
              14 |@@@@@@@@@@@@@@@@@@@@@@@@@                70
              15 |                                         0
              16 |@@@@@@@@@@@@                             34
              17 |                                         0

  xemacs-20.4
           value  ------------- Distribution ------------- count
               6 |                                         0
               7 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  521
               8 |                                         0
               9 |                                         1
              10 |                                         0

  make.bin
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  3596
               2 |                                         0
               3 |                                         0
               4 |                                         42
               5 |                                         50
               6 |                                         0

  acomp
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@                                    1156
               2 |                                         0
               3 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         6635
               4 |@                                        297
               5 |                                         0

  iropt
           value  ------------- Distribution ------------- count
               2 |                                         0
               3 |                                         299
               4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  20144
               5 |                                         0
```

Vous pouvez également utiliser la fonction de groupement `lquantize()` pour regrouper le temps écoulé depuis un certain point du passé. Cette technique permet d'observer une modification de comportement au fil du temps. L'exemple suivant affiche la modification dans le comportement de l'appel système sur une durée de vie d'un processus exécutant la commande `date(1)` :

```
syscall::exec:return,
syscall::exece:return
/execname == "date"/
{
    self->start = vtimestamp;
}

syscall:::entry
/self->start/
{
    /*
     * We linearly quantize on the current virtual time minus our
     * process's start time.  We divide by 1000 to yield microseconds
     * rather than nanoseconds.  The range runs from 0 to 10 milliseconds
     * in steps of 100 microseconds; we expect that no date(1) process
     * will take longer than 10 milliseconds to complete.
     */
    @a["system calls over time"] =
        lquantize((vtimestamp - self->start) / 1000, 0, 10000, 100);
}

syscall::rexit:entry
/self->start/
{
    self->start = 0;
}
```

Le script précédent fournit plus de détails sur le comportement de l'appel système lorsque de nombreux processus `date(1)` sont exécutés. Pour voir ce résultat, exécutez `sh -c 'while true; do date >/dev/null; done'` dans une fenêtre, tandis que le script D s'exécute dans une autre. Le script produit un profil du comportement de l'appel système de la commande `date(1)` :

```
# dtrace -s dateprof.d
dtrace: script './dateprof.d' matched 218 probes
^C

  system calls over time
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@                                       20530
             100 |@@@@@@                                   48814
             200 |@@@                                      28119
```

```
     300 |@                                        14646
     400 |@@@@@                                    41237
     500 |                                         1259
     600 |                                         218
     700 |                                         116
     800 |@                                        12783
     900 |@@@                                      28133
    1000 |                                         7897
    1100 |@                                        14065
    1200 |@@@                                      27549
    1300 |@@@                                      25715
    1400 |@@@@                                     35011
    1500 |@@                                       16734
    1600 |                                         498
    1700 |                                         256
    1800 |                                         369
    1900 |                                         404
    2000 |                                         320
    2100 |                                         555
    2200 |                                         54
    2300 |                                         17
    2400 |                                         5
    2500 |                                         1
    2600 |                                         7
    2700 |                                         0
```

Ce résultat donne une idée approximative des différentes phases de la commande date(1) par rapport aux services requis du noyau. Pour mieux comprendre ces phases, vous souhaitez peut-être comprendre quels appels système font l'objet d'un appel et à quel moment. Si c'est le cas, vous pouvez modifier le script D pour regrouper la variable `probefunc` à la place d'une chaîne constante.

Impression de groupements

Par défaut, plusieurs groupements s'affichent selon leur ordre d'introduction dans le programme D. Vous pouvez ignorer ce comportement en utilisant la fonction `printa()` pour afficher les groupements. La fonction `printa()` permet également de formater de façon précise les données de groupement en utilisant une chaîne de format, tel que décrit dans Chapitre12, “Format de sortie”.

Si un groupement n'est pas formaté avec une instruction `printa()` dans votre programme D, la commande `dtrace` crée un instantané des données de groupement et affiche les résultats une fois, après l'achèvement du suivi, en utilisant le format de groupement par défaut. Si un groupement donné est formaté avec une instruction `printa()`, le comportement par défaut est désactivé. Vous pouvez obtenir des résultats équivalents en ajoutant l'instruction `printa(@`*nom-groupement*) à une clause de sonde `dtrace:::END` dans votre programme. Le format de

sortie par défaut des fonctions de groupement `avg()`, `count()`, `min()`, `max()` et `sum()` affiche une valeur décimale d'entier correspondant à la valeur groupée de chaque tuple. Le format de sortie par défaut des fonctions de groupement `lquantize()` et `quantize()` affiche un tableau ASCII des résultats. Les tuples de groupement sont affichés comme si la fonction `trace()` était appliquée à chaque élément de tuple.

Normalisation des données

Lors du groupement de données sur une période donnée, vous souhaitez certainement *normaliser* les données en fonction d'un facteur constant. Cette technique permet de comparer des données disjointes plus facilement. Par exemple, lors du groupement d'appels système, vous souhaitez certainement afficher comme sortie les appels système en tant que vitesse par seconde au lieu d'une valeur absolue au cours de l'exécution. L'action `normalize()` de DTrace vous permet ainsi de normaliser des données. Les paramètres sur lesquels doit s'appliquer la fonction `normalize()` sont un facteur de groupement et de normalisation. La sortie du groupement présente chaque valeur divisée par le facteur de normalisation.

L'exemple suivant présente la méthode de groupement des données par appel système :

```
#pragma D option quiet

BEGIN
{
    /*
     * Get the start time, in nanoseconds.
     */
    start = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

END
{
    /*
     * Normalize the aggregation based on the number of seconds we have
     * been running.  (There are 1,000,000,000 nanoseconds in one second.)
     */
    normalize(@func, (timestamp - start) / 1000000000);
}
```

L'exécution du script ci-dessus pendant une brève période permet d'obtenir la sortie suivante sur un ordinateur de bureau :

```
# dtrace -s ./normalize.d
 ^C
  syslogd                                                          0
  rpc.rusersd                                                      0
  utmpd                                                            0
  xbiff                                                            0
  in.routed                                                        1
  sendmail                                                         2
  echo                                                             2
  FvwmAuto                                                         2
  stty                                                             2
  cut                                                              2
  init                                                             2
  pt_chmod                                                         3
  picld                                                            3
  utmp_update                                                      3
  httpd                                                            4
  xclock                                                           5
  basename                                                         6
  tput                                                             6
  sh                                                               7
  tr                                                               7
  arch                                                             9
  expr                                                            10
  uname                                                           11
  mibiisa                                                         15
  dirname                                                         18
  dtrace                                                          40
  ksh                                                             48
  java                                                            58
  xterm                                                          100
  nscd                                                           120
  fvwm2                                                          154
  prstat                                                         180
  perfbar                                                        188
  Xsun                                                          1309
  .netscape.bin                                                 3005
```

La fonction `normalize()` définit le facteur de normalisation pour le groupement spécifié, mais cette action ne modifie pas les données sous-jacentes. La fonction `denormalize()` prend uniquement un groupement. L'ajout de l'action d'annulation de la normalisation à l'exemple précédent renvoie le nombre d'appels système brut ainsi que la vitesse par seconde :

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
```

```
}

syscall:::entry
{
    @func[execname] = count();
}

END
{
    this->seconds = (timestamp - start) / 1000000000;
    printf("Ran for %d seconds.\n", this->seconds);

    printf("Per-second rate:\n");
    normalize(@func, this->seconds);
    printa(@func);

    printf("\nRaw counts:\n");
    denormalize(@func);
    printa(@func);
}
```

L'exécution du script ci-dessus pendant une brève période permet d'obtenir une sortie similaire à l'exemple suivant :

```
# dtrace -s ./denorm.d
^C
Ran for 14 seconds.
Per-second rate:

  syslogd                                                           0
  in.routed                                                         0
  xbiff                                                             1
  sendmail                                                          2
  elm                                                               2
  picld                                                             3
  httpd                                                             4
  xclock                                                            6
  FvwmAuto                                                          7
  mibiisa                                                          22
  dtrace                                                           42
  java                                                             55
  xterm                                                            75
  adeptedit                                                       118
  nscd                                                            127
  prstat                                                          179
  perfbar                                                         184
  fvwm2                                                           296
  Xsun                                                            829
```

```
Raw counts:

  syslogd                                                          1
  in.routed                                                        4
  xbiff                                                           21
  sendmail                                                        30
  elm                                                             36
  picld                                                           43
  httpd                                                           56
  xclock                                                          91
  FvwmAuto                                                       104
  mibiisa                                                        314
  dtrace                                                         592
  java                                                           774
  xterm                                                         1062
  adeptedit                                                     1665
  nscd                                                          1781
  prstat                                                        2506
  perfbar                                                       2581
  fvwm2                                                         4156
  Xsun                                                         11616
```

Les groupements peuvent également être renormalisés. Si la fonction `normalize()` est appelée plusieurs fois pour le même groupement, le facteur de normalisation sera le facteur spécifié dans l'appel le plus récent. L'exemple suivant imprime les vitesses par seconde au fil du temps :

EXEMPLE 9–1 `renormalize.d` : renormalisation d'un groupement

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

tick-10sec
{
    normalize(@func, (timestamp - start) / 1000000000);
    printa(@func);
}
```

Effacement de groupements

Lors de l'utilisation de DTrace pour élaborer des scripts de surveillance simples, vous pouvez, de temps en temps, effacer les valeurs d'un groupement en utilisant la fonction `clear()`. Cette fonction prend comme unique paramètre un groupement. La fonction `clear()` efface uniquement les *valeurs* du groupement ; les clés du groupement sont conservées. Par conséquent, la présence d'une clé dans un groupement ayant une valeur associée de zéro indique que la clé *avait* une valeur différente de zéro qui a été ultérieurement définie sur zéro dans le cadre de l'exécution d'une fonction `clear()`. Pour rejeter à la fois les valeurs d'un groupement et ses clés, utilisez la fonction `trunc()`. Pour plus d'informations, reportez-vous à la section "Troncature de groupements" à la page 129.

L'exemple suivant ajoute la fonction `clear()` à l'Exemple 9–1 :

```
#pragma D option quiet

BEGIN
{
    last = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

tick-10sec
{
    normalize(@func, (timestamp - last) / 1000000000);
    printa(@func);
    clear(@func);
    last = timestamp;
}
```

Tandis que l'Exemple 9–1 affiche le taux d'appels système sur toute la durée de vie de l'appel `dtrace`, l'exemple précédent affiche le taux d'appels système uniquement sur les dix dernières secondes.

Troncature de groupements

Lorsque l'on observe des résultats de groupement, ce sont souvent les dix premiers résultats qui présentent un intérêt. Les clés et valeurs associées aux valeurs autres que les valeurs les plus élevées ne présentent pas d'intérêt. Vous souhaitez peut-être également rejeter un résultat de groupement dans son intégralité, en supprimant à la fois les clés *et* les valeurs. La fonction `trunc()` de DTrace est utilisée dans ces deux cas de figure.

Les paramètres sur lesquels appliquer la fonction `trunc()` sont un groupement et une valeur de troncature facultative. Sans la valeur de troncature, `trunc()` rejette *à la fois* les valeurs *et* les clés de groupement du groupement complet. Lorsqu'une valeur de troncature *n* est présente, `trunc()` rejette les valeurs et les clés de groupement *à l'exception* de celles associées aux valeurs *n* les plus élevées. Cela signifie que `trunc(@foo, 10)` tronque le groupement appelé `foo` après les dix premières valeurs, là où `trunc(@foo)` rejette le groupement dans son intégralité. Le groupement complet est également rejeté si `0` est spécifié comme valeur de troncature.

Pour afficher les valeurs *n* du bas au lieu de celles du haut, spécifiez une valeur de troncature négative dans `trunc()`. Par exemple, `trunc(@foo, -10)` tronque la valeur nommée `foo` après les dix dernières valeurs.

L'exemple suivant augmente l'exemple d'appel système pour n'afficher que le nombre d'appels système par seconde des dix premières applications d'appels système sur dix secondes :

```
#pragma D option quiet

BEGIN
{
    last = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

tick-10sec
{
    trunc(@func, 10);
    normalize(@func, (timestamp - last) / 1000000000);
    printa(@func);
    clear(@func);
    last = timestamp;
}
```

L'exemple suivant affiche la sortie de l'exécution du script ci-dessus sur un ordinateur portable légèrement chargé :

```
  FvwmAuto                                                          7
  telnet                                                           13
  ping                                                             14
  dtrace                                                           27
  xclock                                                           34
  MozillaFirebird-                                                 63
  xterm                                                           133
  fvwm2                                                           146
```

```
acroread                                                          168
Xsun                                                              616

telnet                                                              4
FvwmAuto                                                            5
ping                                                               14
dtrace                                                             27
xclock                                                             35
fvwm2                                                              69
xterm                                                              70
acroread                                                          164
MozillaFirebird-                                                  491
Xsun                                                             1287
```

Réduction des abandons

Étant donné que DTrace met en tampon certaines données de groupement dans le noyau, il risque de manquer d'espace disponible lors de l'ajout d'une nouvelle clé au groupement. Dans ce cas, les données sont abandonnées, le compteur incrémenté et `dtrace` génère un message indiquant l'abandon d'un groupement. Ce cas de figure ne se produit que rarement car DTrace conserve un état d'exécution longue (comprenant la clé du groupement et le résultat intermédiaire) au niveau utilisateur, où l'espace peut être augmenté de façon dynamique. Dans l'éventualité peu probable d'un abandon, vous pouvez augmenter la taille du tampon de groupement avec l'option `aggsize` pour réduire le risque d'abandons. Vous pouvez également utiliser cette option pour réduire l'empreinte de mémoire de DTrace. Comme c'est la cas avec toutes les options de taille, il est possible de spécifier `aggsize` avec n'importe quel suffixe de taille. La stratégie de redimensionnement de ce tampon est dictée par l'option `bufresize.` Pour de plus amples informations sur la mise en tampon, reportez-vous au Chapitre11, “Tampons et mise en tampon”. Pour de plus amples informations sur les détails, reportez-vous au Chapitre16, “Options et paramètres réglables”.

Une autre méthode permettant d'éliminer les abandons de groupement consiste à augmenter la vitesse de consommation des données du groupement au niveau utilisateur. Par défaut, cette vitesse est définie sur une fois par seconde. Elle peut être réglée de façon explicite avec l'option `aggrate.` Comme avec n'importe quelle option de vitesse, `aggrate` peut être spécifié avec n'importe quel suffixe temporel mais le suffixe par défaut est le nombre par seconde. Pour de plus amples informations sur l'option `aggsize`, reportez-vous au Chapitre16, “Options et paramètres réglables”.

CHAPITRE 10

Actions et sous-routines

Vous pouvez utiliser des appels de fonction D tels que `trace()` et `printf()`pour appeler deux types de service différents fournis par DTrace : les*actions* d'une part, qui suivent les données ou modifient des états externes à DTrace et les *sous-routines* d'autre part, qui affectent uniquement des états internes à DTrace. Ce chapitre définit les actions et sous-routines et décrit leur syntaxe et sémantique.

Actions

Les actions permettent à vos programmes DTrace d'interagir avec le système à l'extérieur de DTrace. Les actions les plus courantes enregistrent des données vers un tampon de DTrace. D'autres actions sont disponibles, telles que l'arrêt du processus en cours, l'augmentation d'un signal spécifique sur le processus en cours ou encore l'interruption totale du suivi. Certaines de ces actions sont dites *destructrices* dans la mesure où elles modifient radicalement le système. Il n'est possible de les utiliser que si elles ont été activées de façon explicite. Par défaut, les actions d'enregistrement de données enregistrent des données dans le *tampon principal*. Pour de plus amples informations sur le tampon principal et les stratégies de tampon, reportez-vous au Chapitre11, “Tampons et mise en tampon”.

Action par défaut

Une clause peut contenir un nombre quelconque d'actions et de manipulations de variables. Si une clause reste vide, l'*action par défaut* s'applique. L'action par défaut consiste à suivre l'identificateur de la sonde activée (EPID) sur le tampon principal. L'EPID identifie l'activation d'une sonde donnée avec un prédicat et des actions spécifiques. À partir de l'EPID, les consommateurs DTrace peuvent identifier la sonde à l'origine de l'action. En effet, lorsque des données font l'objet d'un suivi, elles doivent être accompagnées de l'EPID pour que le consommateur puisse les interpréter. L'action par défaut consiste donc à procéder exclusivement au suivi de l'EPID.

L'utilisation de l'action par défaut permet une utilisation simple de dtrace(1M). Par exemple, la commande suivante, donnée à titre d'exemple, permet d'activer toutes les sondes du module de planification de temps partagé TS avec l'action par défaut :

```
# dtrace -m TS
```

La commande précédente donne une sortie similaire à ce qui suit :

```
# dtrace -m TS
dtrace: description 'TS' matched 80 probes
CPU     ID                    FUNCTION:NAME
  0  12077                 ts_trapret:entry
  0  12078                ts_trapret:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12081                  ts_wakeup:entry
  0  12082                 ts_wakeup:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12023                  ts_update:entry
  0  12079             ts_update_list:entry
  0  12080            ts_update_list:return
  0  12079             ts_update_list:entry
...
```

Actions d'enregistrement de données

Les actions d'enregistrement de données constituent les actions principales de DTrace. Chacune de ces actions enregistre des données dans le tampon principal par défaut mais elles peuvent également en enregistrer dans des tampons spéculatifs. Pour de plus amples informations sur le tampon principal, reportez-vous au Chapitre11, “Tampons et mise en tampon”. Pour de plus amples informations sur les tampons spéculatifs, reportez-vous au Chapitre13, “Suivi spéculatif”. Les descriptions données dans cette section se rapportent uniquement au *tampon spécifié*. Elles indiquent si les données sont enregistrées dans le tampon principal ou dans un tampon spéculatif, si l'action fait suite à une action speculate().

trace()

`void trace(`*expression*`)`

L'action la plus élémentaire est l'action `trace()`, qui prend comme argument une expression D et procède au suivi du résultat dans le tampon spécifié. Les instructions suivantes sont des exemples d'actions `trace()` :

```
trace(execname);
trace(curlwpsinfo->pr_pri);
trace(timestamp / 1000);
trace(‘lbolt);
trace("somehow managed to get here");
```

tracemem()

`void tracemem(`*address*`, size_t `*nbytes*`)`

L'action `tracemem()` prend comme premier argument une expression D, *address*, et comme second argument une constante, *nbytes*. `tracemem()` copie la mémoire à partir de l'adresse *addr* dans le tampon approprié en respectant la longueur *nbytes*.

printf()

`void printf(string `*format*`, ...)`

De la même manière que l'action `trace()`, l'action `printf()` procède au suivi d'expressions D. Toutefois, `printf()` autorise un formatage de style `printf(3C)`. Comme `printf(3C)`, les paramètres se composent d'une chaîne *format* suivie d'un nombre variable d'arguments. Par défaut, les arguments font l'objet d'un suivi dans le tampon spécifié. Les arguments sont ensuite formatés pour la sortie dans `dtrace(1M)` en fonction de la chaîne de format spécifiée. Par exemple, les deux premiers exemples de `trace()` de la section “`trace()`” à la page 135 peuvent être combinés dans une seule action `printf()` :

```
printf("execname is %s; priority is %d", execname, curlwpsinfo->pr_pri);
```

Pour plus d'informations sur `printf()`, reportez-vous au Chapitre12, “Format de sortie”.

printa()

`void printa(`*aggregation*`)`
`void printa(string `*format*`, `*aggregation*`)`

L'action `printa()` permet d'afficher des groupements et de les formater. Pour de plus amples informations sur les groupements, reportez-vous au Chapitre9, “Groupements”. Si aucun *format* n'est fourni, `printa()` procède uniquement au suivi, dans le consommateur DTrace, d'une directive en vertu de laquelle le groupement spécifié doit être traité et affiché avec le format par défaut. Si aucun *format* n'est fourni, le groupement sera formaté tel que spécifié. Pour plus d'informations sur la chaîne de format Chapitre12, “Format de sortie”, reportez-vous au `Chapter 12, Output Formatting()`.

`printa()` effectue uniquement le suivi d'une *directive* en vertu de laquelle le groupement doit être traité par le consommateur DTrace. Cette action ne traite pas le groupement dans le noyau. Par conséquent, le temps écoulé entre le suivi de la directive `printa()` et le traitement effectif de la directive dépend des facteurs affectant le traitement du tampon. Parmi ces facteurs figurent le taux de groupement, la stratégie de mise en tampon et, en cas de sélection de la stratégie `switching`, la vitesse de commutation des tampons. Pour plus d'informations sur ces facteurs, reportez-vous au Chapitre9, “Groupements” et au Chapitre11, “Tampons et mise en tampon”.

stack()

```
void stack(int nframes)
void stack(void)
```

L'action `stack()` enregistre un suivi de pile de noyau dans le tampon spécifié. La pile de noyau aura une profondeur de *nframes.* Si *nframes* n'est pas fourni, le nombre de cadres de pile enregistrés correspond au nombre spécifié par l'option `stackframes.` Exemple :

```
# dtrace -n uiomove:entry'{stack()}'
  CPU     ID                    FUNCTION:NAME
    0   9153                    uiomove:entry
                genunix`fop_write+0x1b
                namefs`nm_write+0x1d
                genunix`fop_write+0x1b
                genunix`write+0x1f7

    0   9153                    uiomove:entry
                genunix`fop_read+0x1b
                genunix`read+0x1d4

    0   9153                    uiomove:entry
                genunix`strread+0x394
                specfs`spec_read+0x65
                genunix`fop_read+0x1b
                genunix`read+0x1d4
   ...
```

L'action `stack()` diffère légèrement des autres actions dans la mesure où elle peut également être utilisée en tant que clé de groupement.

```
# dtrace -n kmem_alloc:entry'{@[stack()] = count()}'
dtrace: description 'kmem_alloc:entry' matched 1 probe
^C

                rpcmod`endpnt_get+0x47c
                rpcmod`clnt_clts_kcallit_addr+0x26f
                rpcmod`clnt_clts_kcallit+0x22
                nfs`rfscall+0x350
                nfs`rfs2call+0x60
                nfs`nfs_getattr_otw+0x9e
                nfs`nfsgetattr+0x26
                nfs`nfs_getattr+0xb8
                genunix`fop_getattr+0x18
                genunix`cstat64+0x30
                genunix`cstatat64+0x4a
                genunix`lstat64+0x1c
                  1

                genunix`vfs_rlock_wait+0xc
                genunix`lookuppnvp+0x19d
                genunix`lookuppnat+0xe7
                genunix`lookupnameat+0x87
                genunix`lookupname+0x19
                genunix`chdir+0x18
                  1

                rpcmod`endpnt_get+0x6b1
                rpcmod`clnt_clts_kcallit_addr+0x26f
                rpcmod`clnt_clts_kcallit+0x22
                nfs`rfscall+0x350
                nfs`rfs2call+0x60
                nfs`nfs_getattr_otw+0x9e
                nfs`nfsgetattr+0x26
                nfs`nfs_getattr+0xb8
                genunix`fop_getattr+0x18
                genunix`cstat64+0x30
                genunix`cstatat64+0x4a
                genunix`lstat64+0x1c
                  1
    ...
```

ustack()

```
void ustack(int nframes, int strsize)
void ustack(int nframes)
void ustack(void)
```

L'action `ustack()` enregistre une trace de pile *utilisateur* dans le tampon spécifié. La pile utilisateur aura une profondeur de *nframes*. Si *nframes* n'est pas fourni, le nombre de cadres de pile enregistrés correspond au nombre spécifié par l'option `ustackframes`. Tant que `ustack()` est capable de déterminer l'adresse de cadres d'appel lorsque la sonde se déclenche, les cadres de pile ne seront convertis en symbole qu'au moment où le consommateur DTrace traite l'action `ustack()` au niveau utilisateur. Si *strsize* est spécifié et que sa valeur est différente de zéro, `ustack()` alloue l'espace de chaîne spécifié pour effectuer une conversion d'adresse en symbole directement depuis le noyau. Cette conversion directe en symbole utilisateur est actuellement disponible uniquement sur les machines virtuelles Java 1.5 ou une version supérieure. La conversion Java d'adresse en symbole annote les piles utilisateur contenant des cadres Java avec un nom de méthode et une classe Java. Si ces cadres ne peuvent pas être convertis, ils apparaîtront uniquement en tant qu'adresses hexadécimales.

L'exemple suivant effectue le suivi d'une pile sans espace de chaîne, et donc sans conversion Java d'adresse en symbole.

```
# dtrace -n syscall::write:entry'/pid == $target/{ustack(50, 0);
    exit(0)}' -c "java -version"
dtrace: description 'syscall::write:entry' matched 1 probe
java version "1.5.0-beta3"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta3-b58)
Java HotSpot(TM) Client VM (build 1.5.0-beta3-b58, mixed mode)
dtrace: pid 5312 has exited
CPU     ID                    FUNCTION:NAME
  0     35                      write:entry
              libc.so.1`_write+0x15
              libjvm.so`__1cDhpiFwrite6FipkvI_I_+0xa8
              libjvm.so`JVM_Write+0x2f
              d0c5c946
              libjava.so`Java_java_io_FileOutputStream_writeBytes+0x2c
              cb007fcd
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb000152
              libjvm.so`__1cJJavaCallsLcall_helper6FpnJJavaValue_
                          pnMmethodHandle_pnRJavaCallArguments_
```

```
                pnGThread__v_+0x187
        libjvm.so‘__1cCosUos_exception_wrapper6FpFpnJJavaValue_
                pnMmethodHandle_pnRJavaCallArguments_
                pnGThread__v2468_v_+0x14
        libjvm.so‘__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle_
                pnRJavaCallArguments_pnGThread __v_+0x28
        libjvm.so‘__1cRjni_invoke_static6FpnHJNIEnv__pnJJavaValue_
                pnI_jobject_nLJNICallType_pnK_jmethodID_pnSJNI_
                ArgumentPusher_pnGThread__v_+0x180
        libjvm.so‘jni_CallStaticVoidMethod+0x10f
        java‘main+0x53d
```

Vous remarquerez que les cadres de pile C et C++ de la machine virtuelle Java sont présentés symboliquement, en utilisant des noms symboliques “mutilés” C++ tandis que les cadres de pile Java sont présentés uniquement sous forme d'adresses hexadécimales. L'exemple suivant illustre un appel à `ustack()` avec un espace de chaîne dont la valeur est différente de zéro :

```
# dtrace -n syscall::write:entry’/pid == $target/{ustack(50, 500); exit(0)}’
      -c "java -version"
dtrace: description ’syscall::write:entry’ matched 1 probe
java version "1.5.0-beta3"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta3-b58)
Java HotSpot(TM) Client VM (build 1.5.0-beta3-b58, mixed mode)
dtrace: pid 5308 has exited
CPU     ID                    FUNCTION:NAME
  0     35                     write:entry
              libc.so.1‘_write+0x15
              libjvm.so‘__1cDhpiFwrite6FipkvI_I_+0xa8
              libjvm.so‘JVM_Write+0x2f
              d0c5c946
              libjava.so‘Java_java_io_FileOutputStream_writeBytes+0x2c
              java/io/FileOutputStream.writeBytes
              java/io/FileOutputStream.write
              java/io/BufferedOutputStream.flushBuffer
              java/io/BufferedOutputStream.flush
              java/io/PrintStream.write
              sun/nio/cs/StreamEncoder$CharsetSE.writeBytes
              sun/nio/cs/StreamEncoder$CharsetSE.implFlushBuffer
              sun/nio/cs/StreamEncoder.flushBuffer
              java/io/OutputStreamWriter.flushBuffer
              java/io/PrintStream.write
              java/io/PrintStream.print
              java/io/PrintStream.println
              sun/misc/Version.print
              sun/misc/Version.print
              StubRoutines (1)
              libjvm.so‘__1cJJavaCallsLcall_helper6FpnJJavaValue_
                        pnMmethodHandle_pnRJavaCallArguments_pnGThread
```

```
            __v_+0x187
libjvm.so‘__1cCosUos_exception_wrapper6FpFpnJJavaValue_
            pnMmethodHandle_pnRJavaCallArguments_pnGThread
            __v2468_v_+0x14
libjvm.so‘__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle
            _pnRJavaCallArguments_pnGThread__v_+0x28
libjvm.so‘__1cRjni_invoke_static6FpnHJNIEnv__pnJJavaValue_pnI
            _jobject_nLJNICallType_pnK_jmethodID_pnSJNI
            _ArgumentPusher_pnGThread__v_+0x180
libjvm.so‘jni_CallStaticVoidMethod+0x10f
java‘main+0x53d
8051b9a
```

Le résultat de l'exemple ci-dessus affiche des informations symboliques sur les cadres de pile Java. Des cadres hexadécimaux restent affichés dans cette sortie car certaines fonctions sont statiques et n'ont pas d'entrée dans le tableau des symboles d'application. La conversion est impossible pour ces cadres.

La conversion du symbole `ustack()` pour les cadres autres que Java se produit *après* l'enregistrement des données de pile. Par conséquent, le processus utilisateur correspondant risque de se terminer avant la réalisation de la conversion du symbole, rendant la conversion des cadres de pile impossible. Si le processus utilisateur se termine avant la réalisation de la conversion du symbole, `dtrace` émet un message d'avertissement, suivi des cadres de pile hexadécimaux, comme illustré dans l'exemple suivant :

```
dtrace: failed to grab process 100941: no such process
              c7b834d4
              c7bca85d
              c7bca1a4
              c7bd4374
              c7bc2628
              8047efc
```

Vous trouverez davantage d'informations sur les techniques pour restreindre ce problème au Chapitre33, “Suivi des processus utilisateur”.

Enfin, étant donné que les commandes du débogueur DTrace post-mortem ne peuvent pas exécuter la conversion des cadres, l'utilisation de `ustack()` avec une stratégie de tampon `ring` donnera des données `ustack()` brutes.

Le programme D suivant montre un exemple de `ustack()` laissant *strsize* non spécifié :

```
syscall::brk:entry
/execname == $$1/
{
    @[ustack(40)] = count();
}
```

Pour exécuter cet exemple destiné au navigateur Web Netscape `.netscape.bin` dans les installations Solaris par défaut, utilisez la commande suivante :

```
# dtrace -s brk.d .netscape.bin
dtrace: description 'syscall::brk:entry' matched 1 probe
^C
                libc.so.1`_brk_unlocked+0xc
                88143f6
                88146cd
                .netscape.bin`unlocked_malloc+0x3e
                .netscape.bin`unlocked_calloc+0x22
                .netscape.bin`calloc+0x26
                .netscape.bin`_IMGCB_NewPixmap+0x149
                .netscape.bin`il_size+0x2f7
                .netscape.bin`il_jpeg_write+0xde
                8440c19
                .netscape.bin`il_first_write+0x16b
                8394670
                83928e5
                .netscape.bin`NET_ProcessHTTP+0xa6
                .netscape.bin`NET_ProcessNet+0x49a
                827b323
                libXt.so.4`XtAppProcessEvent+0x38f
                .netscape.bin`fe_EventLoop+0x190
                .netscape.bin`main+0x1875
                   1

                libc.so.1`_brk_unlocked+0xc
                libc.so.1`sbrk+0x29
                88143df
                88146cd
                .netscape.bin`unlocked_malloc+0x3e
                .netscape.bin`unlocked_calloc+0x22
                .netscape.bin`calloc+0x26
                .netscape.bin`_IMGCB_NewPixmap+0x149
                .netscape.bin`il_size+0x2f7
                .netscape.bin`il_jpeg_write+0xde
                8440c19
                .netscape.bin`il_first_write+0x16b
                8394670
                83928e5
                .netscape.bin`NET_ProcessHTTP+0xa6
                .netscape.bin`NET_ProcessNet+0x49a
                827b323
                libXt.so.4`XtAppProcessEvent+0x38f
                .netscape.bin`fe_EventLoop+0x190
                .netscape.bin`main+0x1875
                  1
    ...
```

jstack()

```
void jstack(int nframes, int strsize)
void jstack(int nframes)
void jstack(void)
```

`jstack()` est un alias de `ustack()` qui utilise l'option `jstackframes` en fonction du nombre de cadres de pile et de la taille de l'espace de chaîne, `jstackstrsize.` Par défaut, `jstacksize` prend une valeur différente de zéro. Cela signifie que l'utilisation de `jstack()` produit une pile avec une conversion des cadres Java.

Actions destructrices

Certaines actions DTrace sont destructrices dans la mesure où elles modifient radicalement l'état du système. Il n'est possible de les utiliser que si elles ont été activées de manière explicite. Lors de l'utilisation de `dtrace(1M)`, vous pouvez activer des actions destructrices en utilisant l'option `-w`. En cas de tentative d'activation non explicite d'actions destructrices dans `dtrace(1M)`, `dtrace` échouera, affichant un message similaire à ce qui suit :

```
dtrace: failed to enable 'syscall': destructive actions not allowed
```

Actions destructrices de processus

Certaines actions destructrices ne le sont que dans un processus particulier. Ces actions sont disponibles pour les utilisateurs disposant de privilèges `dtrace_proc` ou `dtrace_user.` Pour plus d'informations sur les privilèges de sécurité DTrace, reportez-vous au Chapitre35, "Sécurité".

stop()

```
void stop(void)
```

L'action `stop()` force le processus qui déclenche l'arrêt de la sonde activée lorsque celle-ci quitte le noyau, comme si elle était arrêtée par une action `proc(4)`. L'utilitaire `prun(1)` peut être utilisé pour la reprise d'un processus ayant été arrêté par l'action `stop()`. L'action `stop()` peut être utilisée pour arrêter un processus à un point de sonde DTrace quelconque. Cette action peut être utilisée pour capturer un programme dans un état particulier, qu'il serait difficile d'obtenir avec un point d'interruption simple puis pour joindre au processus un débogueur classique tel que `mdb(1)`. Vous pouvez également utiliser l'utilitaire `gcore(1)` pour enregistrer l'état d'un processus arrêté dans un fichier Core à des fins d'analyses ultérieures.

raise()

```
void raise(int signal)
```

L'action `raise()` envoie le signal spécifié au processus en cours d'exécution. Recourir à cette action revient à utiliser la commande `kill(1)` permettant d'envoyer un signal à un processus. Il est possible d'utiliser l'action `raise()` pour envoyer un signal à un point précis de l'exécution d'un processus.

copyout()

`void copyout(void *`*buf*`, uintptr_t` *addr*`, size_t` *nbytes*`)`

L'action `copyout()` copie *nbytes* à partir du tampon *buf* à l'adresse *addr* dans l'espace d'adressage du processus associé au thread actuel. Si l'adresse de l'espace utilisateur ne correspond pas à une page valide, par défaut, dans l'espace d'adressage, une erreur est générée.

copyoutstr()

`void copyoutstr(string` *str*`, uintptr_t` *addr*`, size_t` *maxlen*`)`

L'action `copyoutstr()` copie la chaîne *str* vers l'adresse *addr* dans l'espace d'adressage du processus associé au thread actuel. Si l'adresse de l'espace utilisateur ne correspond pas à une page valide, par défaut, dans l'espace d'adressage, une erreur est générée. La longueur de chaîne est limitée à la valeur définie par l'option `strsize.` Pour de plus amples informations, reportez-vous au Chapitre16, "Options et paramètres réglables".

system()

`void system(string` *program*`, ...)`

L'action `system()` provoque l'exécution du programme *program* comme s'il était saisi dans le shell. La chaîne *program* peut contenir n'importe quelle conversion de format `printf()/printa. ()` Des arguments correspondant aux conversions de format doivent être spécifiés. Pour plus d'informations sur les conversions dans des formats valides, reportez-vous au Chapitre12, "Format de sortie".

L'exemple suivant exécute la commande `date(1)` une fois par seconde :

```
# dtrace -wqn tick-1sec'{system("date")}'
 Tue Jul 20 11:56:26 CDT 2004
 Tue Jul 20 11:56:27 CDT 2004
 Tue Jul 20 11:56:28 CDT 2004
 Tue Jul 20 11:56:29 CDT 2004
 Tue Jul 20 11:56:30 CDT 2004
```

L'exemple suivant illustre une utilisation élaborée de l'action, avec des conversions `printf()` dans la chaîne *program* et des outils de filtre traditionnels comme des tubes :

```
#pragma D option destructive
#pragma D option quiet
```

```
proc:::signal-send
/args[2] == SIGINT/
{
    printf("SIGINT sent to %s by ", args[1]->pr_fname);
    system("getent passwd %d | cut -d: -f5", uid);
}
```

Exécuter le script ci-dessus engendre une sortie similaire à l'exemple suivant :

```
# ./whosend.d
SIGINT sent to MozillaFirebird- by Bryan Cantrill
SIGINT sent to run-mozilla.sh by Bryan Cantrill
^C
SIGINT sent to dtrace by Bryan Cantrill
```

L'exécution de la commande spécifiée *n'a pas lieu* dans le cadre du déclenchement d'une sonde, mais lorsque le tampon contenant les détails sur l'action `system()` est traité au niveau utilisateur. La manière dont ce traitement a lieu et le moment auquel il a lieu dépendent de la stratégie de mise en tampon, décrite dans le Chapitre11, “Tampons et mise en tampon” Avec la stratégie de mise en tampon par défaut, la vitesse de traitement du tampon est spécifiée par l'option `switchrate.` Vous pouvez voir le délai inhérent au `system()` si vous réglez de façon explicite `switchrate` sur une vitesse supérieure à la valeur par défaut de une seconde, comme illustré dans l'exemple suivant :

```
#pragma D option quiet
#pragma D option destructive
#pragma D option switchrate=5sec

tick-1sec
/n++ < 5/
{
    printf("walltime  : %Y\n", walltimestamp);
    printf("date      : ");
    system("date");
    printf("\n");
}

tick-1sec
/n == 5/
{
    exit(0);
}
```

Exécuter le script ci-dessus engendre une sortie similaire à l'exemple suivant :

```
# dtrace -s ./time.d
 walltime  : 2004 Jul 20 13:26:30
date      : Tue Jul 20 13:26:35 CDT 2004
```

```
walltime  : 2004 Jul 20 13:26:31
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:32
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:33
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:34
date      : Tue Jul 20 13:26:35 CDT 2004
```

Vous remarquerez que les valeurs `walltime` diffèrent mais que les valeurs `date` sont identiques. Ce résultat reflète le fait que l'exécution de la commande `date(1)` a eu lieu uniquement lors du traitement du tampon et non lors de l'enregistrement de l'action `system()`.

Actions destructrices de noyau

Certaines actions destructrices le sont dans l'intégralité du système. Ces actions doivent évidemment être utilisées avec un très grand soin car elles affectent tous les processus du système ainsi que ceux de tous les autres systèmes dépendant de manière implicite ou explicite des services réseau du système affecté.

breakpoint()

```
void breakpoint(void)
```

L'action `breakpoint()` provoque un point d'interruption dans le noyau, engendrant l'arrêt et le transfert de la commande vers le débogueur du noyau. Le débogueur du noyau émet une chaîne dénotant la sonde DTrace ayant engendré l'action. Par exemple, si l'on exécute le programme suivant :

```
# dtrace -w -n clock:entry'{breakpoint()}'
dtrace: allowing destructive actions
dtrace: description 'clock:entry' matched 1 probe
```

Avec Solaris fonctionnant sous SPARC, le message suivant risque de s'afficher sur la console :

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb 30002765700)
Type  'go' to resume
ok
```

Avec Solaris fonctionnant sous x86, le message suivant risque de s'afficher sur la console :

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb d2b97060)
stopped at      int20+0xb:      ret
kmdb[0]:
```

L'adresse suivant la description de sonde est l'adresse du bloc de contrôle d'activation (ECB) au sein de DTrace. Vous pouvez utiliser cette adresse pour obtenir plus d'informations sur l'activation de sonde ayant provoqué l'action de point d'interruption.

En cas d'erreur relative à l'action `breakpoint()`, le nombre d'appels de cette action risque d'être largement supérieur au nombre prévu. Ce comportement peut par ailleurs vous empêcher de mettre fin au consommateur DTrace engendrant les actions de point d'interruption. Dans ce cas, définissez la variable de nombre entier du noyau, `dtrace_destructive_disallow` sur 1. Ce paramétrage interdira *toutes* les actions destructives sur la machine. Appliquez ce réglage *uniquement* dans ce cas de figure précis.

La méthode exacte permettant de régler `dtrace_destructive_disallow` dépend du débogueur du noyau utilisé. Si vous utilisez le programme OpenBoot PROM sur un système SPARC, utilisez `w!` :

```
ok 1 dtrace_destructive_disallow w!
ok
```

Confirmez que la variable a été définie en utilisant `w?` :

```
ok dtrace_destructive_disallow w?
1
ok
```

Continuez en tapant `go` :

```
ok go
```

En cas d'utilisation de `kmdb(1)` sur des systèmes x86 ou SPARC, utilisez le modificateur d'écriture à 4 octets (`W`) avec le formatage dcmd `/` :

```
kmdb[0]: dtrace_destructive_disallow/W 1
dtrace_destructive_disallow:    0x0             =       0x1
kmdb[0]:
```

Continuez en utilisant `:c` :

```
kadb[0]: :c
```

Pour réactiver des actions destructrices après avoir continué, vous devez réinitialiser de façon explicite `dtrace_destructive_disallow` en lui réattribuant la valeur 0 avec `mdb(1)` :

```
# echo "dtrace_destructive_disallow/W 0" | mdb -kw
dtrace_destructive_disallow:    0x1             =       0x0
#
```

panic()

```
void panic(void)
```

Lorsqu'elle est déclenchée, l'action `panic()` provoque une erreur grave de noyau. Cette action doit être utilisée pour forcer un vidage mémoire sur incident sur un système à un moment stratégique. Vous pouvez utiliser conjointement à cette action des analyses postmortem et de mise en tampon circulaire pour mieux comprendre le problème. Pour plus d'informations, reportez-vous au Chapitre11, “Tampons et mise en tampon” et au Chapitre37, “Suivi post-mortem”. Lorsque l'action d'erreur grave est utilisée, un message d'erreur grave s'affiche, dénotant la sonde qui en est à l'origine. Exemple :

```
panic[cpu0]/thread=30001830b80: dtrace: panic action at probe
syscall::mmap:entry (ecb 300000acfc8)

000002a10050b840 dtrace:dtrace_probe+518 (fffe, 0, 1830f88, 1830f88,
  30002fb8040, 300000acfc8)
  %l0-3: 0000000000000000 00000300030e4d80 0000030003418000 00000300018c0800
  %l4-7: 000002a10050b980 0000000000000500 0000000000000000 0000000000000502
000002a10050ba30 genunix:dtrace_systrace_syscall32+44 (0, 2000, 5,
  80000002, 3, 1898400)
  %l0-3: 00000300030de730 0000000002200008 00000000000000e0 000000000184d928
  %l4-7: 00000300030de000 0000000000000730 0000000000000073 0000000000000010

syncing file systems... 2 done
dumping to /dev/dsk/c0t0d0s1, offset 214827008, content: kernel
100% done: 11837 pages dumped, compression ratio 4.66, dump
succeeded
rebooting...
```

`syslogd`(1M) émet également un message au moment de la réinitialisation :

```
Jun 10 16:56:31 machine1 savecore: [ID 570001 auth.error] reboot after panic:
dtrace: panic action at probe syscall::mmap:entry (ecb 300000acfc8)
```

Le tampon du message de vidage mémoire sur incident contient également la sonde et l'ECB responsable de l'action `panic()`.

chill()

`void chill(int` *nanoseconds*`)`

L'action `chill()` provoque la rotation de DTrace pendant un nombre de nanosecondes spécifié. `chill()` est essentiellement utilisé pour l'exploration de problèmes de synchronisation. Par exemple, vous pouvez utiliser cette action pour ouvrir des fenêtres de condition de compétitivité ou pour mettre des événements périodiques en phase ou hors phase les uns avec les autres. Étant donné que les interruptions sont désactivées lorsqu'elles sont dans un contexte de sonde DTrace, toute utilisation de `chill()` provoquera une latence

d'interruption, de planification et de répartition. Par conséquent, `chill()` peut provoquer des effets inattendus sur le système. Il convient donc de l'utiliser à bon escient. Étant donné que l'activité du système repose sur la gestion des interruptions périodiques, DTrace refusera d'exécuter l'action `chill()` pendant plus de 500 millisecondes par intervalle d'une seconde sur une CPU donnée. En cas de dépassement de l'intervalle `chill()`, l'erreur d'opération illégale fera l'objet d'un rapport dans DTrace, comme illustré dans l'exemple suivant :

```
# dtrace -w -n syscall::open:entry'{chill(500000001)}'
dtrace: allowing destructive actions
dtrace: description 'syscall::open:entry' matched 1 probe
dtrace: 57 errors
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 1 (ID 14: syscall::open:entry): \
  illegal operation in action #1
```

Cette limite s'applique même si le temps est réparti sur plusieurs appels à `chill()` ou sur plusieurs consommateurs DTrace d'une sonde unique. Par exemple, la même erreur peut être générée par la commande suivante :

```
# dtrace -w -n syscall::open:entry'{chill(250000000); chill(250000001);}'
```

Actions spéciales

Cette section décrit les actions qui ne sont ni des actions d'enregistrement de données ni des actions destructrices.

Actions spéculatives

Les actions associées à un suivi spéculatif sont les actions `speculate()`, `commit()`et `discard()`. Vous trouverez plus d'informations sur ces actions au Chapitre13, “Suivi spéculatif”.

exit()

`void exit(int` *status*`)`

L'action `exit()` est utilisée pour interrompre immédiatement le suivi et pour informer le consommateur DTrace qu'il doit interrompre le suivi, effectuer tout traitement final et appeler `exit(3C)` avec l'état spécifié. Étant donné que `exit()` renvoie un état au niveau utilisateur, il s'agit d'une action d'enregistrement de données. Toutefois, contrairement aux autres actions de stockage de données, `exit()` ne peut pas faire l'objet d'un suivi spéculatif. `exit()` provoquera la fermeture du consommateur DTrace, quelle que soit la stratégie de tampon. Étant donné que `exit()` est une action d'enregistrement de données, il est *possible* de l'abandonner.

Lorsque `exit()` est appelé, seules les actions DTrace déjà en cours d'exécution sur d'autres CPU seront terminées. Aucune nouvelle action ne se produira sur aucune CPU. La seule exception à cette règle est le traitement de la sonde `END` qui est appelée après que le consommateur DTrace a traité l'action `exit()` et indiqué que le suivi devait être arrêté.

Sous-routines

Les sous-routines diffèrent des actions car généralement, elles n'affectent que l'état interne de DTrace. Par conséquent, il n'existe pas de sous-routines destructrices et les sous-routines ne procèdent jamais au suivi de données dans des tampons. De nombreuses sous-routines ont des analogues dans les sections 9F ou 3C. Pour plus d'informations sur les sous-routines correspondantes, reportez-vous aux sections `Intro(9F)` et `Intro(3)`.

alloca()

`void *alloca(size_t` *size*`)`

`alloca()` alloue *size* octets depuis un espace de travail, et renvoie un pointeur vers la mémoire allouée. Le pointeur renvoyé présente systématiquement un alignement de 8 octets. L'espace de travail n'est valide que pendant la durée d'une clause. La mémoire allouée avec `alloca()`est libérée lors de l'achèvement de la clause. Si l'espace de travail disponible n'est pas suffisant, aucune mémoire n'est allouée et une erreur est générée.

basename()

`string basename(char *`*str*`)`

`basename()` est un analogue D de `basename(1)`. Cette sous-routine crée une chaîne qui comprend une copie de la chaîne spécifiée, mais sans le préfixe se terminant par `/`. La chaîne renvoyée est allouée à l'extérieur de la mémoire de travail. Sa durée de validité correspond donc à la durée de la clause. Si l'espace de travail disponible est insuffisant, `basename` ne s'exécute pas et une erreur est générée.

bcopy()

`void bcopy(void *`*src*`, void *`*dest*`, size_t` *size*`)`

`bcopy()` copie *size* octets de la mémoire *src* dans la mémoire *dest*. L'ensemble de la mémoire source doit résider à l'extérieur de la mémoire de travail tandis que l'ensemble de la mémoire de destination doit résider à l'intérieur. Si ces conditions ne sont pas remplies, aucune copie n'est effectuée et une erreur est générée.

cleanpath()

`string cleanpath(char *str)`

`cleanpath()` crée une chaîne qui consiste en une copie du chemin *str*, mais dont certains éléments redondants sont éliminés. En particulier, les éléments “`/./`” du chemin sont supprimés et les éléments “`/../`” réduits. La réduction des éléments `/../` est effectuée indépendamment des liens symboliques. Il est donc possible que `cleanpath()` copie un chemin valide et renvoie un chemin plus court, non valide.

Par exemple, si *str* correspond à “ `/foo/../bar`” et que `/foo` est un lien symbolique vers `/net/foo/export` `cleanpath()` renvoie la chaîne “`/bar`” même si `bar` risque de ne figurer que dans `/net/foo` et non dans `/`. Cette limitation est due au fait que `cleanpath()` est appelé dans le contexte d'un déclenchement de sonde, où la résolution de liens symboliques ou de noms arbitraires est impossible. La chaîne renvoyée est allouée à l'extérieur de la mémoire de travail. Sa durée de validité correspond donc à la durée de la clause. Si l'espace de travail disponible est insuffisant, `cleanpath` ne s'exécute pas et une erreur est générée.

copyin()

`void *copyin(uintptr_t addr, size_t size)`

`copyin()` copie le nombre d'octets spécifié depuis l'adresse de l'utilisateur spécifiée dans un tampon de travail DTrace et renvoie l'adresse de ce tampon. L'adresse de l'utilisateur est interprétée en tant qu'adresse de l'espace du processus associé au thread actuel. Le pointeur de tampon qui en résulte comporte obligatoirement un alignement à 8 octets. L'adresse en question *doit* correspondre à une page par défaut dans le processus en cours. Si l'adresse ne correspond pas à une page par défaut, ou si l'espace de travail disponible est insuffisant, la valeur `NULL` est renvoyée et une erreur est générée. Pour connaître les techniques visant à réduire le risque d'apparition d'erreurs `copyin`, reportez-vous au Chapitre33, “Suivi des processus utilisateur”.

copyinstr()

`string copyinstr(uintptr_t addr)`

`copyinstr()` copie une chaîne C terminée par un octet nul à partir de l'adresse de l'utilisateur spécifiée dans un tampon de travail DTrace et renvoie l'adresse de ce tampon. L'adresse de l'utilisateur est interprétée en tant qu'adresse de l'espace du processus associé au thread actuel. La longueur de la chaîne est limitée à la valeur définie par l'option `strsize` ; pour plus d'informations, consultez le Chapitre16, “Options et paramètres réglables”. Comme pour `copyin`, l'adresse spécifiée *doit* correspondre à une page par défaut dans le processus en cours. Si l'adresse ne correspond pas à une page par défaut, ou si l'espace de travail disponible est

insuffisant, la valeur NULL est renvoyée et une erreur est générée. Pour connaître les techniques visant à réduire le risque d'apparition d'erreurs Chapitre33, “Suivi des processus utilisateur”, reportez-vous au Chapter 33, User Process Tracing.

copyinto()

void copyinto(uintptr_t *addr*, size_t *size*, void **dest*)

copyin() copie le nombre d'octets spécifié depuis l'adresse de l'utilisateur spécifiée dans un tampon de travail DTrace spécifié par *dest*. L'adresse de l'utilisateur est interprétée en tant qu'adresse de l'espace du processus associé au thread actuel. L'adresse en question *doit* correspondre à une page par défaut dans le processus en cours. Si l'adresse ne correspond pas à une page par défaut ou qu'aucune mémoire de destination ne réside à l'extérieur de l'espace de travail, aucune copie n'est effectuée et une erreur est générée. Pour connaître les techniques visant à réduire le risque d'apparition d'erreurs Chapitre33, “Suivi des processus utilisateur”, reportez-vous au Chapter 33, User Process Tracing.

dirname()

string dirname(char **str*)

dirname() est un analogue D de dirname(1). Cette sous-routine crée une chaîne comprenant le nom du chemin complet à l'exception du dernier niveau, *str* . La chaîne renvoyée est allouée à l'extérieur de la mémoire de travail. Sa durée de validité correspond donc à la durée de la clause. Si l'espace de travail disponible est insuffisant, dirname ne s'exécute pas et une erreur est générée.

msgdsize()

size_t msgdsize(mblk_t **mp*)

msgdsize() renvoie le nombre d'octets dans le message de données vers lequel pointe *mp*. Pour plus d'informations, consultez msgdsize(9F). msgdsize() ne prend en compte dans le total que les blocs de données de type M_DATA.

msgsize()

size_t msgsize(mblk_t **mp*)

msgsize() renvoie le nombre d'octets dans le message vers lequel pointe *mp*. Contrairement à msgdsize(), qui renvoie uniquement le nombre d'octets des *données*, msgsize() renvoie le nombre *total* d'octets du message.

mutex_owned()

```
int mutex_owned(kmutex_t *mutex)
```

mutex_owned() est une implémentation de mutex_owned(9F). mutex_owned() renvoie une valeur différente de zéro si le thread d'appel est le propriétaire du mutex de noyau, ou une valeur égale à zéro si le mutex adaptatif n'a pas de propriétaire.

mutex_owner()

```
kthread_t *mutex_owner(kmutex_t *mutex)
```

mutex_owner() renvoie le pointeur de thread du propriétaire actuel du mutex de noyau adaptatif. mutex_owner() renvoie NULL si le mutex adaptatif spécifié n'a pas de propriétaire ou s'il s'agit d'un spin mutex. Consultez mutex_owned(9F).

mutex_type_adaptive()

```
int mutex_type_adaptive(kmutex_t *mutex)
```

mutex_type_adaptive() renvoie une valeur différente de zéro si le mutex de noyau spécifié est de type MUTEX_ADAPTIVE, ou une valeur égale à zéro dans le cas contraire. Les mutex sont adaptatifs s'ils remplissent au moins une des conditions suivantes :

- Le mutex est déclaré de façon statique.
- Le mutex est créé avec un cookie de bloc d'interruption NULL.
- Le mutex est créé avec un cookie de bloc d'interruption qui ne correspond pas à une interruption de haut niveau.

Pour plus d'informations sur les mutex, consultez mutex_init(9F). La majorité des mutex du noyau Solaris sont adaptatifs.

progenyof()

```
int progenyof(pid_t pid)
```

progenyof() renvoie une valeur différente de zéro si le processus d'appel (le processus associé au thread qui déclenche actuellement la sonde correspondant aux critères) fait partie du progeny de l'ID de processus spécifié.

rand()

```
int rand(void)
```

`rand()` renvoie un nombre entier pseudo-aléatoire. Le nombre renvoyé est un nombre pseudo-aléatoire faible. Il ne doit pas être utilisé dans le cadre d'une application cryptographique.

rw_iswriter()

`int rw_iswriter(krwlock_t *rwlock)`

`rw_iswriter()` renvoie une valeur différente de zéro si le verrou de lecture-écriture est possédé ou requis par un programme d'écriture. Si le verrou est possédé par des programmes de lecture et qu'aucun programme d'écriture n'est bloqué, ou que le verrou n'est pas possédé, `rw_iswriter()` renvoie une valeur égale à zéro. Consultez `rw_init`(9F).

rw_write_held()

`int rw_write_held(krwlock_t *rwlock)`

`rw_write_held()` renvoie une valeur différente de zéro si le verrouillage en lecture-écriture spécifié est actuellement possédé par un programme d'écriture. Si le verrou est possédé uniquement par des programmes de lecture ou qu'il n'est pas possédé, `rw_write_held()` renvoie une valeur égale à zéro. Consultez `rw_init`(9F).

speculation()

`int speculation(void)`

`speculation()` réserve un tampon de suivi spéculatif à utiliser avec `speculate()` et renvoie un identificateur correspondant à ce tampon. Pour plus d'informations, reportez-vous au Chapitre13, “Suivi spéculatif”.

strjoin()

`string strjoin(char *str1, char *str2)`

`strjoin()` crée une chaîne composée de l'élément *str1* concaténé avec l'élément *str2*. La chaîne renvoyée est allouée à l'extérieur de la mémoire de travail. Sa durée de validité correspond donc à la durée de la clause. Si l'espace de travail disponible est insuffisant, `strjoin` ne s'exécute pas et une erreur est générée.

strlen()

```
size_t strlen(string str)
```

`strlen()` renvoie la longueur de la chaîne spécifiée en octets, à l'exception de l'octet nul de fin.

CHAPITRE 11

Tampons et mise en tampon

La mise en tampon et la gestion des données est un service essentiel que fournit la structure DTrace à ses clients, comme par exemple dtrace(1M). Ce chapitre explore en détails la mise en tampon de données et décrit les options que vous pouvez utiliser pour modifier les stratégies de gestion de tampon de DTrace.

Tampons principaux

Le *tampon principal* est présent dans chaque appel DTrace. Par défaut, c'est vers ce tampon que les actions de suivi enregistrent leurs données. Parmi ces actions figurent :

`exit()`	`printf()`	`trace()`	`ustack()`
`printa()`	`stack()`	`tracemem()`	

Les tampons principaux sont *toujours* alloués par CPU. Cette stratégie n'est pas réglable mais le suivi et l'allocation de tampon peuvent se limiter à une seule CPU en utilisant l'option `cpu`.

Stratégies de tampon principal

DTrace autorise le suivi dans des contextes de contrainte élevée du noyau. En particulier, DTrace autorise le suivi dans des contextes dans lesquels le logiciel de noyau risque de ne pas allouer de mémoire de façon fiable. La conséquence de cette flexibilité de contexte est la suivante : il existe *toujours* une possibilité pour que DTrace tente de suivre des données lorsque l'espace disponible est insuffisant. DTrace doit disposer d'une stratégie pour gérer de tels cas de figure, le cas échéant, mais il est possible que vous souhaitiez régler la stratégie en fonction des besoins d'une expérimentation donnée. Parfois, la stratégie appropriée risque de rejeter les nouvelles données. Dans d'autres cas, il peut être souhaitable de réutiliser l'espace contenant les données enregistrées les plus anciennes pour suivre les nouvelles données. Le plus souvent, la

stratégie souhaitée consiste à réduire le risque de manquer d'espace disponible, en tout premier lieu. Pour répondre à ces différentes demandes, DTrace prend en charge plusieurs stratégies de tampon. Cette prise en charge est implémentée avec l'option `bufpolicy` et peut être définie par consommateur. Pour de plus amples informations sur les options de paramétrage, reportez-vous au Chapitre16, “Options et paramètres réglables”.

Stratégie `switch`

Par défaut, le tampon principal dispose d'une stratégie de tampon `switch.` Dans le cadre de cette stratégie, les tampons par CPU sont alloués par paires : un tampon est actif, l'autre non. Lorsqu'un consommateur tente de lire un tampon, le noyau commence par *commuter* les tampons actif et inactif. La commutation des tampons s'effectue de telle sorte qu'aucune donnée de suivi ne risque d'être perdue dans une fenêtre. Une fois que les tampons sont commutés, le tampon devenu inactif est copié dans le consommateur DTrace. Grâce à cette stratégie, le consommateur affiche toujours un tampon autocohérent : un tampon ne fait jamais l'objet d'un suivi et d'une copie simultanément. Cette technique permet également d'éviter l'introduction d'une fenêtre dans laquelle le suivi est mis à l'état de pause ou empêché d'une autre manière. La vitesse de commutation et de lecture du tampon est contrôlée par le consommateur avec l'option `switchrate.` Comme avec n'importe quelle option de vitesse, `switchrate` peut être spécifié avec n'importe quel suffixe temporel mais le suffixe par défaut est le nombre par seconde. Pour de plus amples informations sur les options, telles que `switchrate`, reportez-vous au Chapitre16, “Options et paramètres réglables”.

Remarque – Pour procéder au traitement du tampon principal au niveau utilisateur à un taux supérieur que le taux par défaut d'une fois par seconde, réglez la valeur de `switchrate`. Le système traite les actions qui entraînent l'activité au niveau utilisateur (telles que `printa()` et `system()`) lorsque l'enregistrement correspondant du tampon principal est traité. La valeur de `switchrate` détermine le taux auquel le système traite ces actions.

Dans le cadre de la stratégie `switch`, si une sonde activée donnée procède au suivi de davantage de données qu'il n'y a d'espace disponible dans le tampon principal actif, les données sont *abandonnées* et le nombre d'abandons par CPU est incrémenté. Dans l'éventualité d'un ou de plusieurs abandons, `dtrace(1M)` affiche un message similaire à l'exemple suivant :

```
dtrace: 11 drops on CPU 0
```

Si un enregistrement donné est supérieur à la taille totale du tampon, l'enregistrement est supprimé quelle que soit la stratégie de tampon. Vous pouvez réduire ou éliminer les suppressions soit en augmentant la taille du tampon principal avec l'option `bufsize`, soit en augmentant la vitesse de commutation avec l'option `switchrate`.

Dans le cadre de la stratégie `switch`, l'espace de travail pour `copyin()`, `copyinstr()` et `alloca()` est alloué à l'extérieur du tampon actif.

Stratégie fill

Dans le cas de certains problèmes, vous pouvez utiliser un tampon unique interne au noyau. Tant que cette approche peut être implémentée avec la stratégie switch et les constructions D appropriées en incrémentant une variable en D et en réalisant un prédicat d'action exit() correctement, cette implémentation n'élimine pas le risque d'abandons. Pour demander un tampon unique de grande capacité interne au noyau et poursuivre le suivi jusqu'à ce qu'au moins un des tampons par CPU soit rempli, utilisez la stratégie de tampon fill. Dans le cadre de cette stratégie, le suivi se poursuit jusqu'à ce qu'une sonde activée tente de suivre plus de données que ne peut en accueillir l'espace restant sur le tampon principal. Lorsque l'espace restant est insuffisant, le tampon est marqué comme rempli et le consommateur est notifié qu'au moins un de ses tampons par CPU est rempli. Une fois que dtrace(1M) détecte un tampon rempli, le suivi est arrêté, tous les tampons sont traités, puis dtrace se ferme. Aucune autre donnée n'est suivie dans un tampon rempli même s'il reste suffisamment de place pour l'accueillir.

Pour utiliser la stratégie fill, définissez l'option bufpolicy sur fill. Par exemple, la commande suivante procède au suivi de chaque entrée d'appel système dans un tampon par CPU de 2K avec la stratégie de tampon définie sur fill :

```
# dtrace -n syscall:::entry -b 2k -x bufpolicy=fill
```

Stratégie fill et sondes END

En règle générale, les sondes END ne se déclenchent pas avant que le suivi n'ait été arrêté de manière explicite par le consommateur DTrace. Les sondes END se déclenchent toujours sur une CPU, mais celle-ci n'est pas définie. Avec les tampons fill, le suivi est arrêté de manière explicite lorsqu'au moins l'un des tampons principaux par CPU est marqué comme rempli. Si la stratégie fill est sélectionnée, la sonde END est susceptible de se déclencher sur une CPU ayant un tampon rempli. Pour accueillir le suivi END dans des tampons fill, DTrace calcule la quantité d'espace potentiellement consommée par les sondes END et *soustrait* cet espace de la taille du tampon principal. Si la taille nette est négative, DTrace refuse de démarrer et dtrace(1M) affiche un message d'erreur correspondant.

```
dtrace: END enablings exceed size of principal buffer
```

Le mécanisme de réservation garantit qu'un tampon plein dispose toujours d'un espace suffisant pour n'importe quelle sonde END.

Stratégie ring

La stratégie de tampon ring de DTrace vous aide à procéder au suivi d'événéments à l'origine de pannes. Si la reproduction de la panne prend plusieurs heures ou plusieurs jours, vous ne souhaiterez certainement conserver que les données les plus récentes. Dès qu'un tampon

principal est rempli, le suivi effectue une recherche circulaire sur la première entrée, écrasant ainsi des données de suivi antérieures. Vous établissez le tampon circulaire en définissant l'option `bufpolicy` sur la chaîne `ring` :

```
# dtrace -s foo.d -x bufpolicy=ring
```

Lorsqu'il est utilisé pour créer un tampon circulaire, `dtrace(1M)` n'affiche aucune sortie jusqu'à la fin du processus. Lors de l'achèvement du processus, le tampon circulaire est consommé et traité. `dtrace` traite chaque tampon circulaire dans l'ordre des CPU. Dans le tampon d'une CPU, les enregistrements de suivi sont affichés chronologiquement, du plus ancien au plus récent. Tout comme dans le cas d'une stratégie de tampon `switch` aucun tri n'existe entre les enregistrements effectués depuis différentes CPU. Si un tri de ce type est requis, vous devez procéder au suivi de la variable `timestamp` dans le cadre de votre requête de suivi.

L'exemple suivant montre l'utilisation d'une directive `#pragma option` pour activer la mise en tampon circulaire :

```
#pragma D option bufpolicy=ring
#pragma D option bufsize=16k

syscall:::entry
/execname == $1/
{
    trace(timestamp);
}

syscall::rexit:entry
{
    exit(0);
}
```

Autres tampons

Les tampons principaux sont présents dans chaque activation DTrace. Outre les tampons principaux, certains consommateurs DTrace peuvent disposer d'autres tampons de données internes au noyau : un *tampon de groupement*, présenté dans le Chapitre9, “Groupements”, et un ou plusieurs *tampons spéculatifs*, présentés dans le Chapitre13, “Suivi spéculatif”.

Taille des tampons

La taille de chaque tampon peut être réglée par consommateur. Des options séparées sont fournies pour régler chaque taille de tampon, comme indiqué dans le tableau suivant :

Tampon	Option de taille
Principal	bufsize
Spéculatif	specsize
Groupement	aggsize

Chacune de ces options est définie avec une valeur dénotant la taille. Comme avec n'importe quelle option de taille, la valeur peut avoir un suffixe de taille facultatif. Pour de plus amples informations, reportez-vous au Chapitre16, “Options et paramètres réglables”. Par exemple, pour définir la taille de tampon à un méga-octet sur la ligne de commande dans dtrace, vous pouvez utiliser -x pour définir l'option :

```
# dtrace -P syscall -x bufsize=1m
```

Vous pouvez également utiliser l'option -b dans dtrace :

```
# dtrace -P syscall -b 1m
```

Enfin, vous pouvez définir bufsize avec l'option #pragma D option :

```
#pragma D option bufsize=1m
```

La taille de tampon sélectionnée dénote la taille de tampon sur *chaque* CPU. En outre, lorsque la stratégie de tampon switch est sélectionnée, bufsize dénote la taille de *chaque* tampon sur chaque CPU. La taille de tampon par défaut est de quatre méga-octets.

Stratégie de redimensionnement du tampon

Parfois, il arrive que le système ne dispose pas de mémoire disponible dans le noyau pour allouer un tampon de la taille souhaitée car la mémoire disponible est insuffisante ou car le consommateur DTrace a dépassé l'une des limites réglables décrites dans le Chapitre16, “Options et paramètres réglables” Vous pouvez configurer la stratégie en cas de panne de l'allocation de tampon avec l'option bufresize, qui, par défaut, est définie sur auto. Dans le cadre de la stratégie de redimensionnement du tampon auto, la taille du tampon est divisée en deux jusqu'à ce qu'une allocation correcte soit effectuée. dtrace(1M) génère un message si la taille du tampon alloué est inférieure à celle requise :

```
# dtrace -P syscall -b 4g
dtrace: description 'syscall' matched 430 probes
dtrace: buffer size lowered to 128m
...
```

ou

```
# dtrace -P syscall'{@a[probefunc] = count()}' -x aggsize=1g
dtrace: description 'syscall' matched 430 probes
dtrace: aggregation size lowered to 128m
...
```

Vous pouvez aussi nécessiter une intervention manuelle après la panne d'allocation de tampon en définissant `bufresize` sur `manual`. Dans le cadre de cette stratégie, une panne d'allocation provoque un échec de démarrage :

```
# dtrace -P syscall -x bufsize=1g -x bufresize=manual
dtrace: description 'syscall' matched 430 probes
dtrace: could not enable tracing: Not enough space
#
```

La stratégie de redimensionnement de *tous* les tampons (principaux, spéculatifs et de groupement) est dictée par l'option `bufresize`.

CHAPITRE 12

Format de sortie

DTrace fournit des fonctions de formatage intégrées `printf()` et `printa()` que vous pouvez utiliser à partir de programmes D pour formater la sortie. Le compilateur D fournit des fonctions non trouvées dans la routine de bibliothèque `printf`(3C). Vous devez donc lire ce chapitre même si vous connaissez déjà `printf()`. Ce chapitre traite également du comportement de formatage de la fonction `trace()` et du format de sortie par défaut utilisé par `dtrace`(1M) pour afficher les groupements.

printf()

La fonction `printf()` combine la capacité de suivi de données, comme la fonction `trace()`avec la capacité de sortir les données et d'autres textes dans le format spécifique décrit. La fonction `printf()` indique à DTrace de suivre les données associées à chaque argument faisant suite au premier argument, puis de formater les résultats en utilisant les règles décrites par le premier argument `printf()` connu sous le nom de *chaîne de format*.

La chaîne de format est une chaîne normale contenant n'importe quel nombre de conversions de format, chacune commençant par le caractère `%` qui décrit comment formater l'argument correspondant. La première conversion dans la chaîne de format correspond au second argument `printf()`, la seconde conversion au troisième argument, etc. L'ensemble du texte entre les conversions est affiché textuellement. Le caractère suivant le caractère de conversion `%` décrit le format à utiliser pour l'argument correspondant.

Contrairement à `printf`(3C), la fonction `printf()` de DTrace est une fonction intégrée reconnue par le compilateur D. Le compilateur D fournit plusieurs services utiles pour la fonction `printf()` de DTrace, qui sont introuvables dans la bibliothèque C `printf()` :

- Le compilateur D compare les arguments aux conversions en chaîne de format. Si un type d'argument est incompatible avec la conversion de format, le compilateur D fournit un message d'erreur expliquant le problème.

- Le compilateur D ne nécessite pas l'utilisation de préfixes de taille avec les conversions de format `printf. ()` La routine `printf()` C requiert que vous indiquiez la taille des arguments par l'ajout de préfixes tels que `%ld` pour `long` ou `%lld` pour `long long`. Le compilateur D connaît la taille et le type de vos arguments, donc ces préfixes ne sont pas requis dans vos instructions `printf()` D.
- DTrace fournit des caractères de format supplémentaires, utiles pour le débogage et la capacité d'observation. Par exemple, la conversion de format `%a` peut être utilisée pour afficher un pointeur en tant que décalage et nom de symbole.

Afin d'implémenter ces fonctions, la chaîne de format dans la fonction `printf()` de DTrace doit être spécifiée en tant que constante de chaîne dans votre programme D. Les chaînes de format peuvent ne pas être des variables dynamiques de type `string`.

Spécifications de conversion

Chaque spécification de conversion dans la chaîne de format est introduite par le caractère `%` qui précède les informations suivantes dans la séquence :

- Aucun ou plusieurs *indicateurs* (dans n'importe quel ordre), qui modifient la signification de la spécification de conversion comme décrit dans la section suivante.
- Une *largeur de champ* minimale facultative. Si la valeur convertie dispose d'un nombre d'octets inférieur à la largeur de champ, elle sera complétée par des espaces sur la gauche par défaut, ou sur la droite si l'indicateur d'alignement à gauche (`-`) est spécifié. La largeur de champ peut également être spécifiée par un astérisque (`*`), auquel cas la largeur de champ est définie dynamiquement en fonction de la valeur d'un argument supplémentaire de type `int`.
- Une *précision* facultative, indiquant le nombre minimum de chiffres devant apparaître pour les conversions `d`, `i`, `o`, `u`, `x` et `X` (le champ est complété avec des zéros non significatifs) ; le nombre de chiffres devant apparaître après le caractère de base des conversions `e`, `E` et `f`, le nombre maximum de chiffres significatifs pour les conversions `g` et `G` ou le nombre maximum d'octets à afficher à partir d'une chaîne par la conversion `s`. La précision prend la forme d'un point (`.`) suivi soit d'un astérisque (`*`), décrit ci-dessous, soit d'une chaîne numérique décimale.
- Une séquence facultative de *préfixes de taille* indiquant la taille de l'argument correspondant, décrite à la section "Préfixes de taille" à la page 164 . Les préfixes de taille ne sont pas nécessairement en D et sont fournis à des fins de compatibilité avec la fonction `printf()` C.
- Un *spécificateur de conversion* indiquant le type de conversion à appliquer à l'argument.

La fonction `printf(3C)` prend également en charge des spécifications de conversion sous le format `%` *n* `$` où *n* est en nombre entier décimal ; la fonction `printf()` de DTrace ne prend pas en charge ce type de spécification de conversion.

Spécificateurs d'indicateurs

Les indicateurs de conversion `printf()` sont activés en spécifiant au moins l'un des caractères suivants, susceptibles d'apparaître dans n'importe quel ordre :

`'` — La partie de nombre entier du résultat d'une conversion décimale (`%i`, `%d`, `%u`, `%f`, `%g` ou `%G`) est formatée avec des milliers de caractères de groupement utilisant le caractère de groupement non monétaire. Certains environnements linguistiques, notamment l'environnement linguistique POSIX C ne fournissent pas de caractères de groupement non monétaires utilisés avec cet indicateur.

`-` — Le résultat de la conversion est justifié à gauche dans le champ. La conversion est justifiée à droite si cet indicateur n'est pas spécifié.

`+` — Le résultat d'une conversion signée commence toujours par un signe (+ ou `-`). Si cet indicateur n'est pas spécifié, la conversion commence par un signe uniquement lorsqu'une valeur négative est convertie.

`space` — Si le premier caractère d'une conversion signée n'est pas un signe ou si une conversion signée ne renvoie aucun caractère, un espace est placé avant le résultat. Si les indicateurs `space` et + sont tous les deux affichés, l'indicateur d'espace est ignoré.

`#` — La valeur est convertie en un format secondaire s'il y en a un de défini pour la conversion sélectionnée. Les formats secondaires pour des conversions sont décrits avec la conversion correspondante.

`0` — Pour les conversions `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g` et `G`, des zéros non significatifs (suivis par n'importe quelle indication de signe ou de base) sont utilisés pour remplir la largeur de champ. Aucun remplissage avec des espaces n'est réalisé. Si les indicateurs `0` et `-` sont tous les deux affichés, l'indicateur `0` est ignoré. Pour les conversions `d`, `i`, `o`, `u`, `x` et `X`, si une précision est spécifiée, l'indicateur `0` est ignoré. Si les indicateurs `0` et `'` sont tous les deux affichés, les caractères de groupement sont insérés avant le remplissage de zéros.

Spécificateurs de largeur et de précision

La largeur de champ minimum peut être spécifiée en tant que chaîne numérique décimale précédée de n'importe quel spécificateur d'indicateur, auquel cas la largeur de champ est définie sur le nombre spécifié de colonnes. La largeur de champ peut également être spécifiée en tant qu'astérisque (`*`) auquel cas on accède à un argument supplémentaire de type `int` pour déterminer la largeur de champ. Par exemple, pour afficher un nombre entier `x` dans une largeur de champ déterminée par la valeur de la variable `int w`, vous devez écrire l'instruction D suivante :

```
printf("%*d", w, x);
```

La largeur de champ peut également être spécifiée en utilisant un caractère ? pour indiquer que la largeur de champ doit être définie en fonction du nombre de caractères requis pour formater une adresse en hexadécimal dans le modèle de données du noyau du système d'exploitation. La largeur est définie sur 8 si le noyau utilise un modèle de données à 32 bits ou sur 16 s'il en utilise un à 64 bits.

La précision pour la conversion peut être spécifiée en tant que chaîne numérique décimale suivie d'un point (.) ou d'un astérisque précédé d'un (*) point. Si un astérisque est utilisé pour spécifier la précision, on accède à un argument supplémentaire de type `int` avant la conversion pour déterminer la précision. Si la largeur et la précision sont précisées en tant qu'astérisques, l'ordre des arguments de `printf()` pour la conversion doit apparaître dans l'ordre suivant. largeur, précision, valeur.

Préfixes de taille

Les préfixes de taille sont requis dans les programmes ANSI-C utilisant `printf(3C)` afin d'indiquer la taille et le type de l'argument de conversion. Le compilateur D exécute ce traitement pour vos appels `printf()`, donc les préfixes de taille ne sont pas requis. Bien que les préfixes de taille soient fournis à des fins de compatibilité avec C, leur utilisation est fortement déconseillée dans les programmes D car ils établissent une liaison entre votre code et un modèle de données particulier lors de l'utilisation de types dérivés. Par exemple, si un `typedef` est redéfini sur des types de base de nombres entiers différents sur le modèle de données, il n'est pas possible d'utiliser une conversion C unique fonctionnant sur les deux modèles de données sans connaître de manière explicite les deux types sous-jacents et inclure une expression de forçage de type, ou définir plusieurs chaînes de format. Le compilateur D résout ce problème automatiquement en vous autorisant à omettre les préfixes de taille et en déterminant automatiquement la taille de l'argument.

Les préfixes de taille peuvent précéder le nom de la conversion de format et être placés après tout spécificateur d'indicateur, de largeur et de précision. Les préfixes de taille se présentent comme suit :

- Un `h` facultatif spécifie qu'une conversion suivante `d`, `i`, `o`, `u`, `x` ou `X` s'applique à un `short` ou à un `unsigned short`.
- Un `l` facultatif spécifie qu'une conversion suivante `d`, `i`, `o`, `u`, `x` ou `X` s'applique à un `long` ou à un `unsigned long`.
- Un `ll` facultatif spécifie qu'une conversion suivante `d`, `i`, `o`, `u`, `x` ou `X` s'applique à un `long long` ou à un `unsigned long long`.
- Un `L` facultatif spécifie qu'une conversion suivante `e`, `E`, `f`, `g` ou `G` s'applique à un `long double`.
- Un `l` facultatif spécifie qu'une conversion suivante `c` s'applique à un argument `wint_t` et qu'une conversion suivante `s` s'applique à un pointeur vers un argument `wchar_t`.

Formats de conversion

Chaque séquence de caractères de conversion provoque l'extraction d'aucun ou de plusieurs arguments. Si un nombre insuffisant d'arguments est fourni pour la chaîne de format, ou si la chaîne de format est épuisée et que des arguments restent disponibles, le compilateur D génère un message d'erreur approprié. Si un format de conversion est spécifié, le compilateur D génère un message d'erreur approprié. Les séquences de caractères de conversion sont les suivantes :

`a` Le pointeur ou l'argument `uintptr_t` est affiché en tant que nom du symbole de noyau sous le format *module* ‘*nom-symbole* plus un décalage d'octets hexadécimaux facultatif. Si la valeur ne se situe pas dans la plage définie par un symbole de noyau connu, elle est affichée en tant que nombre entier hexadécimal.

`c` L'argument `char`, `short` ou `int` est affiché en tant que caractère ASCII.

`C` L'argument `char`, `short` ou `int` est affiché en tant que caractère ASCII si le caractère est un caractère ASCII imprimable. Dans le cas contraire, il est affiché avec la séquence d'échappement correspondante, comme illustré dans le Tableau 2–5.

`d` L'argument `char`, `short`, `int`, `long` ou `long long` est affiché en tant que nombre entier décimal (base 10). Si l'argument est `signed`, il sera affiché en tant que valeur signée. Si l'argument est `unsigned`, il sera affiché en tant que valeur non signée. Cette conversion a la même signification que `i`.

`e, E` L'argument `float`, `double` ou `long double` est converti sur le style `[-]` *d*`.`*ddd*`e`± *dd*, le caractère de base étant précédé d'un chiffre et suivi d'un nombre de chiffres égal à la précision. Le caractère de base n'est pas égal à zéro si l'argument n'est pas égal à zéro. Si aucune précision n'est spécifiée, sa valeur par défaut est 6. Si la précision est égale à 0 et l'indicateur `#` n'est pas spécifié, aucun caractère de base n'apparaît. Le format de conversion `E` produit un nombre avec `E` au lieu de `e` introduisant l'exposant. L'exposant contient toujours au moins deux chiffres. La valeur est arrondie au nombre approprié de chiffres.

`f` L'argument `float`, `double` ou `long double` est converti sur le style `[-]` *ddd*`.` *ddd*, le nombre de chiffres précédant le caractère de base étant égal à la spécification de précision. Si aucune précision n'est spécifiée, sa valeur par défaut est 6. Si la précision est égale à 0 et l'indicateur `#` n'est pas spécifié, aucun caractère de base n'apparaît. Si un caractère de base s'affiche, il est précédé d'au moins un chiffre. La valeur est arrondie au nombre approprié de chiffres.

`g, G` L'argument `float`, `double` ou `long double` est affiché dans le style `f` ou `e` (ou dans le style `E` dans le cas d'un caractère de conversion `G`), avec la précision spécifiant le nombre de chiffres significatifs. Si une précision explicite est égale à 0, elle prend 1 comme valeur. Le style utilisé dépend de la valeur convertie : le style `e` (ou `E`) n'est utilisé que si l'exposant résultant de la conversion est inférieur à -4 ou est supérieur ou égal à la précision. Les zéros finaux sont supprimés de la partie fractionnelle du

résultat. Un caractère de base ne s'affiche que s'il précède un chiffre. Si l'indicateur # est spécifié, les zéros finaux ne sont pas supprimés du résultat.

`i` L'argument `char`, `short`, `int`, `long` ou `long long` est affiché en tant que nombre entier décimal (base 10). Si l'argument est `signed`, il sera affiché en tant que valeur signée. Si l'argument est `unsigned`, il sera affiché en tant que valeur non signée. Cette conversion a la même signification que `d`.

`o` L'argument `char`, `short`, `int`, `long` ou `long long` est affiché en tant que nombre entier octal non signé (base 8). Les arguments `signed` ou `unsigned` peuvent être utilisés avec cette conversion. Si l'indicateur # est spécifié, la précision du résultat sera augmentée si nécessaire pour forcer le premier chiffre du résultat à prendre la valeur zéro.

`p` Le pointeur ou l'argument `uintptr_t` est affiché en tant que nombre entier hexadécimal (base 16). D accepte des arguments de pointeur de n'importe quel type. Si l'indicateur # est spécifié, `0x` sera pré-ajouté aux résultats différents de zéro.

`s` L'argument doit être un tableau de `char` ou une `string`. Les octets de ce tableau ou de la `string` sont lus jusqu'à un caractère nul de fin ou jusqu'à la fin des données, puis interprétés et affichés en tant que caractères ASCII. Si la précision n'est pas spécifiée, elle prend une valeur infinie, donc tous les caractères jusqu'au premier caractère nul sont affichés. Dans le cas contraire, seule la partie du tableau de caractères affichée dans le nombre correspondant de colonnes à l'écran s'affiche. Si un argument de type `char *` doit être formaté, son type doit être forcé sur `string` ou l'opérateur D `stringof` doit être placé en préfixe pour indiquer que DTrace doit procéder au suivi des octets de la chaîne et les formater.

`S` L'argument doit être un tableau de `char` ou une `string`. L'argument est traité comme avec la conversion `%s` mais tous les caractères ASCII non imprimables sont remplacés par la séquence d'échappement correspondante décrite dans le Tableau 2–5.

`u` L'argument `char`, `short`, `int`, `long` ou `long long` est affiché en tant que nombre entier décimal non signé (base 10). Les arguments `signed` ou `unsigned` peuvent être utilisés avec cette conversion et le résultat est toujours formaté en tant que `unsigned`.

`wc` L'argument `int` est converti en caractère large (`wchar_t`) et celui qui en résulte s'affiche.

`ws` L'argument doit être un tableau de `wchar_t`. Les octets d'un tableau sont lus jusqu'à un caractère nul de fin ou la fin des données, puis sont interprétés et affichés en tant que caractères larges. Si la précision n'est pas spécifiée, elle prend une valeur infinie, donc tous les caractères larges jusqu'au premier caractère nul sont affichés. Dans le cas contraire, seule la partie du tableau de caractères larges affichée dans le nombre correspondant de colonnes à l'écran s'affiche.

`x, X` L'argument `char`, `short`, `int`, `long` ou `long long` est affiché en tant que nombre entier décimal non signé (base 16). Les arguments `signed` ou `unsigned` peuvent être utilisés avec cette conversion. Si le format de conversion `x` est utilisé, les chiffres littéraux

abcdef sont utilisés. Si le format de conversion X est utilisé, les chiffres littéraux abcdef sont utilisés. Si l'indicateur # est spécifié, 0x (pour %x) ou 0X (pour %X) y sera ajouté.

Y — L'argument uint64_t est interprété comme le nombre de nanosecondes écoulées depuis 00:00 Temps universel coordonné, 1er janvier 1970 et s'affiche sous le format cftime(3C) suivant : “%Y %a %b %e %T %Z”. Le nombre actuel de nanosecondes écoulées depuis le 1er janvier 1970, 00:00 UTC, est disponible dans la variable walltimestamp.

% — Permet d'afficher un caractère % littéral. Aucun argument n'est converti. La spécification de conversion complète doit être %%.

printa()

La fonction printa() est utilisée pour formater les résultats de groupements dans un programme D. La fonction est appelée sous l'un des deux formats suivants :

```
printa(@aggregation-name);
printa(format-string, @aggregation-name);
```

Si le premier format de la fonction est utilisé, la commande dtrace(1M) prend un instantané cohérent des données de groupement et produit une sortie équivalente au format de sortie par défaut utilisé pour les groupements, décrit dans le Chapitre9, “Groupements”.

Si le second format de la fonction est utilisé, la commande dtrace(1M) prend un instantané cohérent des données de groupement et produit une sortie selon les conversions spécifiées dans la *chaîne de format*, en fonction des règles suivantes :

- Les conversions de format doivent correspondre à la signature de tuple utilisée pour créer le groupement. Chaque élément de tuple ne peut s'afficher qu'une seule fois. Par exemple, si vous regroupez un compte avec les instructions D suivantes :

  ```
  @a["hello", 123] = count();
  @a["goodbye", 456] = count();
  ```

  puis ajoutez l'instruction D printa(*chaîne-format*, @a) à une clause de sonde, dtrace prendra un instantané des données de groupement et produira une sortie comme si vous aviez saisi les instructions :

  ```
  printf(format-string, "hello", 123);
  printf(format-string, "goodbye", 456);
  ```

  et ainsi de suite pour chaque tuple défini dans le groupement.

- Contrairement à `printf()`, la chaîne de format utilisée pour `printa()` ne doit pas nécessairement inclure tous les éléments du tuple. Cela signifie que vous pouvez disposer d'un tuple ayant comme longueur 3 avec une seule conversion de format. Par conséquent, vous pouvez omettre n'importe quelle clé de tuple de votre sortie `printa()` en modifiant votre déclaration de groupement pour déplacer les clés que vous souhaitez omettre à la fin du tuple, puis omettre les spécificateurs de conversion leur correspondant dans la chaîne de format `printa()`.
- Le résultat du groupement peut être inclus dans la sortie en utilisant le caractère supplémentaire d'indicateur de format `@`, valide uniquement s'il est utilisé avec `printa()`. L'indicateur `@` peut être combiné avec n'importe quel spécificateur de conversion de format et peut s'afficher plusieurs fois dans une chaîne de format de façon à ce que votre résultat de tuple puisse s'afficher n'importe où dans la sortie et à plusieurs reprises. L'ensemble des spécificateurs de conversion pouvant être utilisés avec chaque fonction de groupement dépend du type de résultat de la fonction de groupement. Les types de résultat de groupement sont les suivants :

`avg()`	`uint64_t`
`count()`	`uint64_t`
`lquantize()`	`int64_t`
`max()`	`uint64_t`
`min()`	`uint64_t`
`quantize()`	`int64_t`
`sum()`	`uint64_t`

  Par exemple, pour formater les résultats de `avg()`, vous pouvez appliquer les conversions de format `%d`, `%i`, `%o`, `%u`, ou `%x`. Les fonctions `quantize()` et `lquantize()` formatent leur résultat sous la forme d'un tableau ASCII plutôt que sous la forme d'une valeur unique.

Le programme D suivant montre un exemple complet de `printa()`, avec le fournisseur `profile` pour tester la valeur de `caller` puis en formatant les résultats sous la forme d'un tableau simple :

```
profile:::profile-997
{
    @a[caller] = count();
}

END
{
    printa("%@8u %a\n", @a);
}
```

Si vous utilisez dtrace pour exécuter ce programme, patientez quelques secondes puis appuyez sur Control-C ; une sortie similaire à l'exemple suivant s'affiche :

```
# dtrace -s printa.d
^C
CPU     ID                    FUNCTION:NAME
  1      2                             :END        1 0x1
       1 ohci‘ohci_handle_root_hub_status_change+0x148
       1 specfs‘spec_write+0xe0
       1 0xff14f950
       1 genunix‘cyclic_softint+0x588
       1 0xfef2280c
       1 genunix‘getf+0xdc
       1 ufs‘ufs_icheck+0x50
       1 genunix‘infpollinfo+0x80
       1 genunix‘kmem_log_enter+0x1e8
       ...
```

Format par défaut trace()

Si la fonction trace() est utilisée pour capturer des données plutôt que printf(), la commande dtrace formate les résultats avec un format de sortie par défaut. Si la taille des données est de 1, 2, 4 ou 8 octets, le résultat est formaté en tant que valeur de nombre entier décimal. Si les données ont une taille différente de celles indiquées ci-dessus, et qu'il s'agit d'une séquence de caractères imprimables s'ils sont interprétés en tant que séquence d'octets, ces données seront affichées en tant que chaîne ASCII. Si les données ont une taille différente de celles indiquées ci-dessus, et qu'il ne s'agit pas d'une séquence de caractères imprimables, elles seront affichées sous la forme d'une série de valeurs d'octets formatés en tant que nombres entiers hexadécimaux.

CHAPITRE 13

Suivi spéculatif

Ce chapitre traite de l'utilitaire DTrace pour le *suivi spéculatif* et de la possibilité de réaliser provisoirement le suivi de données, puis de décider de *valider* ou *supprimer* les données à un tampon de suivi. Dans DTrace, le principal mécanisme de filtrage des événements inintéressants, *prédicat*, est abordé dans le Chapitre4, “Structure de programme D”. Les prédicats sont pratiques lorsque vous savez à quel moment une sonde se déclenche, que l'événement qui lui est associé vous intéresse ou non. Par exemple, si vous n'êtes intéressé que par l'activité en rapport avec un certain processus ou un certain descripteur de fichier, vous savez quand la sonde se déclenche si elle est liée au processus ou au descripteur de fichier qui vous intéresse. Dans d'autres situations, cependant, vous pouvez ne pas être mesure de déterminer si un événement de sonde vous intéresse *avant* que la sonde ne se soit déclenchée.

Par exemple, si un appel système échoue exceptionnellement avec un code d'erreur standard, (`EIO` ou `EINVAL`, notamment), vous souhaiterez peut-être examiner le chemin d'accès au code conduisant à l'erreur. Pour capturer le chemin d'accès au code, vous devez activer toutes les sondes — mais seulement si vous êtes en mesure d'isoler le code défaillant de manière à pouvoir créer un prédicat clair. Si les défaillances sont sporadiques et non déterminantes, vous n'aurez peut-être pas d'autre choix que de suivre tous les événements *susceptibles* de vous intéresser, puis de traiter ultérieurement les données pour filtrer celles qui n'ont pas été associées au chemin d'accès au code défaillant. Le cas échéant, même si le nombre d'événements intéressants peut s'avérer raisonnablement faible, le nombre d'événements suivis doit être très élevé, ce qui complique le traitement ultérieur.

Vous pouvez donc recourir à la fonction de suivi spéculatif pour suivre provisoirement les données vers un ou plusieurs emplacements de sonde, puis choisir de valider les données vers le tampon principal au niveau d'un autre emplacement de sonde. En conclusion, les données suivies ne contiennent que les résultats qui vous intéressent, aucun traitement ultérieur n'est requis et le temps système de DTrace est réduit au minimum.

Interfaces de spéculation

Le tableau suivant décrit les fonctions de spéculation de DTrace :

TABLEAU 13–1 Fonctions de spéculation de DTrace

Nom de la fonction	Args	Description
`speculation`	Aucune	Retourne un identificateur pour un nouveau tampon spéculatif
`speculate`	ID	Indique que le reste de la clause doit être suivi vers le tampon spéculatif spécifié par l'ID
`commit`	ID	Valide le tampon spéculatif lié à l'ID
`discard`	ID	Supprime le tampon spéculatif lié à l'ID

Création d'une spéculation

La fonction `speculation()` alloue un tampon spéculatif et renvoie un identificateur de spéculation. L'identificateur de spéculation doit être utilisé dans les appels ultérieurs vers la fonction `speculate. ()` Les tampons spéculatifs constituent une ressource infinie : Si aucun tampon spéculatif n'est disponible lorsque la fonction `speculation()` est appelée, un ID de zéro est retourné et un compteur d'erreur correspondant de DTrace est incrémenté. Un ID de zéro est toujours invalide mais il peut être transmis à une fonction `speculate()`, `commit()` ou `discard()`. Si une fonction `speculation()` échoue, un message `dtrace` similaire à l'exemple suivant est généré :

```
dtrace: 2 failed speculations (no speculative buffer space available)
```

Le nombre de tampons spéculatifs par défaut est de 1 mais ce chiffre peut éventuellement être plus élevé. Pour plus d'informations, reportez-vous à la section “Options de spéculation et réglage” à la page 179.

Utilisation d'une spéculation

Pour utiliser une spéculation, un identificateur retourné par une fonction `speculation()` doit être transféré à une fonction `speculate()` dans une clause *avant* toute action d'enregistrement des données. Toutes les actions d'enregistrement de données ultérieures dans une clause contenant une fonction `speculate()` feront l'objet d'un suivi spéculatif. Le compilateur D générera une erreur de durée de compilation si un appel de la fonction `speculate()` suit les actions d'enregistrement des données dans une clause de la sonde D. Par conséquent, les clauses peuvent contenir des requêtes de suivi spéculatif ou non-spéculatif, mais pas les deux.

Les actions d'agrégation, de destruction et `exit` peuvent ne jamais être spéculatives. Essayer d'entreprendre l'une de ces actions dans une clause contenant une fonction `speculate()`

provoque une erreur de durée de compilation. Une fonction `speculate()` ne suit pas forcément une fonction `speculate()` : une clause ne peut contenir qu'une spéculation. Une clause contenant une *seule* fonction `speculate()` suit l'action par défaut de manière spéculative, cette action étant configurée pour ne suivre que l'ID de la sonde activée. Pour plus d'informations sur l'action par défaut, reportez-vous au Chapitre10, “Actions et sous-routines”.

En règle générale, vous affectez le résultat d'une fonction `speculation()` à une variable de thread locale, puis vous utilisez cette variable comme prédicat ultérieur à d'autres sondes, ainsi que comme argument à la fonction `speculate()`. Exemple :

```
syscall::open:entry
{
    self->spec = speculation();
}

syscall:::
/self->spec/
{
    speculate(self->spec);
    printf("this is speculative");
}
```

Validation d'une spéculation

Vous validez des spéculations à l'aide de la fonction `commit. ()` Lorsqu'un tampon spéculatif est validé, les données qu'il contient sont copiées dans le tampon principal. Si le tampon spéculatif spécifié comporte plus de données que l'espace disponible dans le tampon principal ne peut en recevoir, aucune donnée n'est copiée et le compteur de pose du tampon est incrémenté. Si le tampon a fait l'objet d'un suivi spéculatif sur plus d'une CPU, les données spéculatives sur la CPU de validation sont copiées immédiatement tandis que les données spéculatives sur les autres CPU sont parfois copiées après la fonction `commit()`. Par conséquent, un certain laps de temps peut s'écouler entre le moment où une fonction `commit()` est exécutée sur une CPU et celui où les données sont copiées des tampons spéculatifs sur les tampons principaux sur toutes les CPU. Ce laps de temps ne peut pas excéder la durée dictée par le taux de nettoyage. Pour plus d'informations, reportez-vous à la section “Options de spéculation et réglage” à la page 179.

Aucun tampon spéculatif de validation n'est disponible pour les appels `speculation()` ultérieurs tant que chaque tampon spéculatif par CPU n'a pas été complètement copié dans le tampon principal par CPU correspondant. De même, les appels ultérieurs de la fonction `speculate()` sur le tampon de validation sont silencieusement supprimés et les appels ultérieurs de la fonction `commit()` ou `discard()` échouent silencieusement. Enfin, une clause contenant une fonction `commit()` ne peut pas contenir d'action d'enregistrement de données. Par contre, une clause peut contenir plusieurs appels `commit()` pour valider des tampons disjoints.

Annulation d'une spéculation

Vous annulez les spéculations à l'aide de la fonction `discard. ()` Lorsqu'un tampon spéculatif est annulé, son contenu est perdu. Si la spéculation n'a été activée que sur la CPU qui appelle la fonction `discard()`, le tampon est immédiatement disponible pour les appels ultérieurs de la fonction `speculation ()`. En cas d'activation de la spéculation sur plusieurs CPU, le tampon supprimé est parfois disponible pour la fonction `speculation()` ultérieure après appel de la fonction `discard()`. Le délai entre l'exécution d'une fonction `discard()` sur une CPU et le moment où le tampon est disponible pour les spéculations ultérieures ne peut pas excéder la durée dictée par le taux de nettoyage. Si au moment de l'appel de la fonction`speculation()`, aucun tampon n'est disponible car *tous* les tampons spéculatifs sont actuellement supprimés ou validés, un message `dtrace` similaire à l'exemple suivant est généré :

```
dtrace: 905 failed speculations (available buffer(s) still busy)
```

Vous pouvez réduire la probabilité d'indisponibilité de tous les tampons en ajustant le nombre de tampons de spéculation ou le taux de nettoyage. Pour plus d'informations, reportez-vous à la section "Options de spéculation et réglage" à la page 179.

Exemple de spéculation

Les spéculations sont notamment utilisées pour mettre en valeur un chemin d'accès au code particulier. L'exemple suivant présente l'intégralité du chemin d'accès au code dans un appel système open(2) lorsque la fonction `open()` échoue :

EXEMPLE 13–1 `specopen.d` : flux de code en cas d'échec open(2)

```
#!/usr/sbin/dtrace -Fs

syscall::open:entry,
syscall::open64:entry
{
    /*
     * The call to speculation() creates a new speculation.  If this fails,
     * dtrace(1M) will generate an error message indicating the reason for
     * the failed speculation(), but subsequent speculative tracing will be
     * silently discarded.
     */
    self->spec = speculation();
    speculate(self->spec);

    /*
     * Because this printf() follows the speculate(), it is being
     * speculatively traced; it will only appear in the data buffer if the
     * speculation is subsequently commited.
```

EXEMPLE 13–1 `specopen.d` : flux de code en cas d'échec open(2) *(Suite)*

```
     */
    printf("%s", stringof(copyinstr(arg0)));
}

fbt:::
/self->spec/
{
    /*
     * A speculate() with no other actions speculates the default action:
     * tracing the EPID.
     */
    speculate(self->spec);
}

syscall::open:return,
syscall::open64:return
/self->spec/
{
    /*
     * To balance the output with the -F option, we want to be sure that
     * every entry has a matching return.  Because we speculated the
     * open entry above, we want to also speculate the open return.
     * This is also a convenient time to trace the errno value.
     */
    speculate(self->spec);
    trace(errno);
}

syscall::open:return,
syscall::open64:return
/self->spec && errno != 0/
{
    /*
     * If errno is non-zero, we want to commit the speculation.
     */
    commit(self->spec);
    self->spec = 0;
}

syscall::open:return,
syscall::open64:return
/self->spec && errno == 0/
{
    /*
     * If errno is not set, we discard the speculation.
     */
```

EXEMPLE 13–1 `specopen.d` : flux de code en cas d'échec open(2) *(Suite)*

```
    discard(self->spec);
    self->spec = 0;
}
```

Exécuter le script ci-dessus engendre une sortie similaire à l'exemple suivant :

```
# ./specopen.d
dtrace: script './specopen.d' matched 24282 probes
CPU FUNCTION
  1  => open                                  /var/ld/ld.config
  1    -> open
  1      -> copen
  1        -> falloc
  1          -> ufalloc
  1            -> fd_find
  1              -> mutex_owned
  1              <- mutex_owned
  1            <- fd_find
  1            -> fd_reserve
  1              -> mutex_owned
  1              <- mutex_owned
  1              -> mutex_owned
  1              <- mutex_owned
  1            <- fd_reserve
  1          <- ufalloc
  1          -> kmem_cache_alloc
  1            -> kmem_cache_alloc_debug
  1              -> verify_and_copy_pattern
  1              <- verify_and_copy_pattern
  1              -> file_cache_constructor
  1                -> mutex_init
  1                <- mutex_init
  1              <- file_cache_constructor
  1              -> tsc_gethrtime
  1              <- tsc_gethrtime
  1              -> getpcstack
  1              <- getpcstack
  1              -> kmem_log_enter
  1              <- kmem_log_enter
  1            <- kmem_cache_alloc_debug
  1          <- kmem_cache_alloc
  1          -> crhold
  1          <- crhold
  1        <- falloc
  1        -> vn_openat
```

```
1          -> lookupnameat
1            -> copyinstr
1            <- copyinstr
1            -> lookuppnat
1              -> lookuppnvp
1                -> pn_fixslash
1                <- pn_fixslash
1                -> pn_getcomponent
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                    -> groupmember
1                      -> supgroupmember
1                      <- supgroupmember
1                    <- groupmember
1                  <- ufs_iaccess
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1                -> pn_getcomponent
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                  <- ufs_iaccess
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1                -> pn_getcomponent
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
```

```
  1                    <- crgetuid
  1                  <- ufs_iaccess
  1                  -> vn_rele
  1                  <- vn_rele
  1                <- ufs_lookup
  1                -> vn_rele
  1                <- vn_rele
  1              <- lookuppnvp
  1            <- lookuppnat
  1          <- lookupnameat
  1        <- vn_openat
  1        -> setf
  1          -> fd_reserve
  1            -> mutex_owned
  1            <- mutex_owned
  1            -> mutex_owned
  1            <- mutex_owned
  1          <- fd_reserve
  1          -> cv_broadcast
  1          <- cv_broadcast
  1        <- setf
  1        -> unfalloc
  1          -> mutex_owned
  1          <- mutex_owned
  1          -> crfree
  1          <- crfree
  1          -> kmem_cache_free
  1            -> kmem_cache_free_debug
  1              -> kmem_log_enter
  1              <- kmem_log_enter
  1              -> tsc_gethrtime
  1              <- tsc_gethrtime
  1              -> getpcstack
  1              <- getpcstack
  1              -> kmem_log_enter
  1              <- kmem_log_enter
  1              -> file_cache_destructor
  1                -> mutex_destroy
  1                <- mutex_destroy
  1              <- file_cache_destructor
  1              -> copy_pattern
  1              <- copy_pattern
  1            <- kmem_cache_free_debug
  1          <- kmem_cache_free
  1        <- unfalloc
  1        -> set_errno
  1        <- set_errno
  1      <- copen
```

```
  1    <- open
  1  <= open                                          2
```

Options de spéculation et réglage

Si un tampon spéculatif est plein lorsqu'une action de suivi spéculatif est tentée, aucune donnée n'est enregistrée dans le tampon et un compteur de pose est incrémenté. Le cas échéant, un message `dtrace` similaire à l'exemple suivant est généré :

```
dtrace: 38 speculative drops
```

Les poses spéculatives *n'*empêchent pas la copie du tampon spéculatif plein dans le tampon principal lorsque le tampon est validé. De même, elles peuvent se produire même si les poses ont eut lieu sur un tampon spéculatif qui a finalement été supprimé. Il est possible de réduire les poses spéculatives en augmentant la taille du tampon spéculatif que l'option `specsize` permet de régler. Vous pouvez spécifier l'option `specsize` avec n'importe quel suffixe de taille. La stratégie de redimensionnement de ce tampon est dictée par l'option `bufresize`.

Les tampons spéculatifs peuvent ne pas être disponibles lors de l'appel de la fonction `speculation. ()` S'il reste encore des tampons à n'avoir été ni validés ni supprimés, un message `dtrace` similaire à l'exemple suivant est généré :

```
dtrace: 1 failed speculation (no speculative buffer available)
```

Vous pouvez réduire la probabilité que des spéculations de cette nature échouent en augmentant le nombre de tampons spéculatifs avec l'option `nspec.` La valeur par défaut de `nspec` est de 1.

Par ailleurs, la fonction `speculation()` peut échouer car tous les tampons spéculatifs sont occupés. Le cas échéant, un message `dtrace` similaire à l'exemple suivant est généré :

```
dtrace: 1 failed speculation (available buffer(s) still busy)
```

Ce message indique que la fonction `speculation()` a été appelée après l'appel de la fonction `commit()` pour un tampon spéculatif mais avant que ce tampon n'ait été réellement validé sur toutes les CPU. Vous pouvez réduire la probabilité que des spéculations de cette nature échouent en augmentant le taux de nettoyage des CPU à l'aide de l'option `cleanrate.` La valeur par défaut de `cleanrate` est `101`.

Remarque – Les valeurs spécifiées pour l'option `cleanrate` doivent être en nombre par seconde. Utilisez le suffixe `hz`.

CHAPITRE 14

Utilitaire dtrace(1M)

La commande dtrace(1M) est l'interface frontale générique de l'utilitaire DTrace. La commande implémente une interface simple pour appeler le compilateur de langage D, la capacité de récupérer des données de suivi mises en tampon à partir de l'utilitaire de noyau DTrace ainsi qu'un ensemble de routines de base pour formater et afficher les données faisant l'objet d'un suivi. Ce chapitre fournit une référence complète à la commande dtrace.

Description

La commande dtrace fournit une interface générique à l'ensemble des services essentiels fournis par l'utilitaire DTrace, notamment :

- les options permettant de répertorier l'ensemble des sondes et fournisseurs actuellement publiés par DTrace ;
- les options permettant d'activer des sondes directement en utilisant un spécificateur de description de sonde quelconque (fournisseur, module, fonction, nom) ;
- les options permettant d'exécuter le compilateur D et de compiler un ou plusieurs fichiers de programme D ou des programmes écrits directement sur la ligne de commande. ;
- les options permettant de générer des programmes de suivi anonyme (consultez le Chapitre36, “Suivi anonyme”) ;
- les options permettant de générer les rapports de stabilité du programme (consultez le Chapitre39, “Stabilité”) ;
- les options permettant de modifier le comportement de suivi et de mise en tampon de DTrace et d'activer les fonctions supplémentaires du compilateur D (consultez le Chapitre16, “Options et paramètres réglables”).

dtrace peut également être utilisé pour créer des scripts D en l'utilisant dans une déclaration #! afin de créer un fichier d'interpréteur (consultez le Chapitre15, “Scripts”). Enfin, vous pouvez utiliser dtrace pour tenter de compiler des programmes D et de déterminer leurs propriétés sans réellement activer le moindre suivi avec l'option -e, décrite ci-après.

Options

La commande dtrace accepte les options suivantes :

dtrace [-32 | -64] [-aACeFGHlqSvVwZ] [-b *bufsz*] [-c *cmd*] [-D *name* [*=def*]] [-I *path*] [-L *path*] [-o *output*] [-p *pid*] [-s *script*] [-U *name*] [-x *arg* [*=val*]] [-Xa | c | s | t] [-P *provider* [[*prédicat*]*action*]] [-m [[*provider:*]*module* [[*prédicat*]*action*]]] [-f [[*provider:*]*module:*]*func* [[*prédicat*]*action*]] [-n [[[*provider:*]*module:*]*func:*]*nom* [[*prédicat*]*action*]] [-i *probe-id* [[*prédicat*]*action*]]

où *predicate* est un prédicat quelconque entre barres obliques / / et *action* est une liste quelconque d'instructions D entre accolades { } selon la syntaxe du langage D précédemment décrite. Si le code du programme en D est fourni en tant qu'argument aux options -P, -m, -f, - n ou -i, ce texte doit être cité de manière appropriée pour éviter une interprétation par le shell. Les options sont les suivantes :

-32, -64
: Le compilateur D produit des programmes utilisant le modèle de données natif du noyau du système d'exploitation. Vous pouvez utiliser la commande isainfo(1) -b pour déterminer le modèle de données du système d'exploitation actuel. Si l'option - 32 est spécifiée, dtrace forcera le compilateur D à compiler un programme D utilisant le modèle de données 32 bits. Si l'option - 64 est spécifiée, dtrace forcera le compilateur D à compiler un programme D utilisant le modèle de données 64 bits. Ces options ne sont généralement pas requises tant que dtrace sélectionne le modèle de données natif comme modèle par défaut. Le modèle de données affecte la taille des types de nombre entier et d'autres propriétés du langage. Les programmes D compilés pour l'un ou l'autre modèle de données peuvent être exécutés sur des noyaux 32 bits comme sur des noyaux 64 bits. Les options -32 et -64 déterminent également le format de fichier ELF (ELF32 ou ELF64) produit par l'option -G.

-a
: Permet de réclamer un état de suivi anonyme et d'afficher les données faisant l'objet d'un suivi. Vous pouvez combiner l'option -a avec l'option -e pour forcer la fermeture de DTrace dtrace immédiatement après la consommation de l'état de suivi anonyme plutôt que de poursuivre l'attente de nouvelles données. Pour plus d'informations sur le suivi anonyme, reportez-vous au Chapitre36, “Suivi anonyme”.

-A
: Permet de générer des directives driver.conf(4) de suivi anonyme. Si l'option -A est spécifiée, dtrace compile n'importe quel programme en D spécifié utilisant l'option -s ou présent sur la ligne de commande et construit un ensemble de directives de fichiers de configuration dtrace(7D) pour activer les sondes spécifiées en vue d'un suivi anonyme (consultez le Chapitre36, “Suivi anonyme”) avant de se fermer. Par défaut, dtrace tente de stocker les directives dans le fichier /kernel/drv/dtrace.conf. Ce comportement peut être modifié avec l'option -o pour spécifier un fichier de sortie secondaire.

`-b` Permet de définir la taille d'un tampon de suivi. La taille d'un tampon de suivi peut comprendre tout suffixe de taille `k`, `m`, `g` ou `t` tels que décrits dans le Chapitre36, “Suivi anonyme”. S'il est impossible d'allouer l'espace de tampon, `dtrace` tente de réduire la taille du tampon ou de se fermer, selon la définition de la propriété `bufresize`.

`-c` Permet d'exécuter la commande spécifiée *cmd* puis de quitter le programme une fois celui-ci achevé. Si plusieurs options `-c` sont présentes sur la ligne de commande, `dtrace` se ferme lorsque toutes les commandes sont fermées, en consignant dans un rapport l'état de sortie de chaque processus enfant lorsqu'il se termine. L'ID de processus de la première commande est disponible pour n'importe quel programme D spécifié sur la ligne de commande ou utilisant l'option `-s` par le biais de la variable de macro `$target.` Pour plus d'informations sur les variables de macro, consultez le Chapitre15, “Scripts”.

`-C` Permet d'exécuter le préprocesseur C `cpp(1)` sur les programmes en D avant de les compiler. Des options peuvent être transmises au préprocesseur C en utilisant les options `-D`, `- U`, `-I` et `-H.` Le degré de conformité aux normes C peut être sélectionné en utilisant l'option `-X.` Reportez-vous à l'option `-X` pour obtenir une description du jeu de jetons défini par le compilateur D lors de l'appel du préprocesseur C.

`-D` Permet de définir le *nom* spécifié lors de l'appel `cpp(1)` (activé avec l'option `-C`). Si un signe égal (`=`) et une *value* supplémentaire sont spécifiés, la valeur correspondante est assignée au nom. Cette option transmet l'option `- D` à chaque appel `cpp`.

`-e` Permet de fermer le programme après la compilation des requêtes et la consommation de l'état de suivi anonyme (option `-a`) mais avant l'activation d'une sonde, quelle qu'elle soit. Il est possible de combiner cette option avec l'option `-a` pour afficher les données de suivi anonyme et fermer le programme, ou bien avec les options du compilateur D pour vérifier que la compilation des programmes se fait correctement, sans nécessairement les exécuter ni activer l'instrumentation correspondante.

`-f` Permet de spécifier le nom de la fonction à suivre ou à répertorier (option `-l).` L'argument correspondant peut prendre l'aspect de n'importe quelle description de sonde : *provider:module:function*, *module:function*, ou *function* . Les champs de description de sonde non spécifiés restent vides et renvoient toutes les sondes, quelle que soit la valeur de ces champs. Si aucun qualificateur autre que *fonction* n'est spécifié dans la description, toutes les sondes avec la *fonction* correspondante sont associées. L'argument `-f` peut être placé en suffixe avec une clause de sonde D facultative. Plusieurs options `-f` à la fois peuvent être spécifiées sur la ligne de commande.

`-F` Permet de fondre la sortie de suivi en identifiant l'entrée et le retour de la fonction. Les rapports de sonde de l'entrée de la fonction sont indentés et `->` est placé avant leur sortie. L'indentation des rapports de sonde de retour de la fonction est annulée et `<-` précède leur sortie.

`-G` Permet de générer un fichier ELF contenant un programme DTrace imbriqué. Les sondes DTrace spécifiées dans le programme sont enregistrées à l'intérieur d'un objet ELF réadressable pouvant être lié à un autre programme. Si l'option `-o` est présente, le fichier ELF est enregistré avec le nom de chemin spécifié comme argument pour cette opérande. Si l'option `-o` n'est pas présente et que le programme DTrace est contenu dans un fichier dont le nom est *nom_fichier* `.s`, le fichier ELF est enregistré sous le nom *fichier* `.o` ; sinon, il est enregistré sous le nom `d.out`.

`-H` Permet d'imprimer les noms de chemin des fichiers inclus lors de l'appel `cpp(1)` (activé avec l'option `-C`). Cette option transmet l'option `-H` à chaque appel `cpp`, provoquant l'affichage de la liste des noms de chemin, à raison d'un par ligne, dans `stderr`.

`-i` Permet de spécifier l'identificateur de sonde à suivre ou à répertorier (option `-l).` Les ID de sonde sont spécifiés avec des nombres entier décimaux comme indiqué par `dtrace -l`. L'argument `-i` peut être placé en suffixe avec la clause de sonde D facultative. Plusieurs options `-i` à la fois peuvent être spécifiées sur la ligne de commande.

`-I` Permet d'ajouter le *chemin_daccès* au répertoire spécifié au chemin de recherche pour les fichiers `#include` lors de l'appel de `cpp(1)` (activé avec l'option `-C`). Cette option transmet l'option `- I` à chaque appel `cpp.` Le répertoire spécifié est inséré dans le chemin de recherche avant la liste des répertoires par défaut.

`-l` Permet de répertorier les sondes au lieu de les activer. Si l'option `-l` est spécifiée, `dtrace` produit un rapport des sondes correspondant aux descriptions données, avec les options `-P`, `-m`, `- f`, `-n`, `-i` et `-s.` Si aucune de ces options n'est spécifiée, toutes les sondes sont répertoriées.

`-L` Permet d'ajouter le *path* du répertoire spécifié au chemin de recherche pour les bibliothèques DTrace. Les bibliothèques DTrace sont utilisées pour contenir des définitions courantes susceptibles d'être utilisées lors de l'écriture de programmes D. Le *path* spécifié est ajouté après le chemin de recherche de la bibliothèque par défaut.

`-m` Permet de spécifier le nom du module à suivre ou à répertorier (option `-l).` L'argument correspondant peut prendre l'aspect de n'importe quelle description de sonde : *provider:module* ou *module*. Les champs de description de sonde non spécifiés restent vides et renvoient toutes les sondes, quelle que soit la valeur de ces champs. Si aucun qualificateur autre que *module* n'est spécifié dans la description, toutes les sondes avec le *module* correspondant sont associées.

L'argument `-m` peut être placé en suffixe avec une clause de sonde D facultative. Plusieurs options `-m` à la fois peuvent être spécifiées sur la ligne de commande.

`-n` Permet de spécifier le nom de sonde à suivre ou à répertorier (option `-l`). L'argument correspondant peut prendre l'aspect de n'importe quelle description de sonde : *nom:fonction:module:fournisseur*, *nom:fonction:module*, *nom:fonction* ou *nom*. Les champs de description de sonde non spécifiés restent vides et renvoient toutes les sondes, quelle que soit la valeur de ces champs. Si aucun qualificateur autre que *nom* n'est spécifié dans la description, toutes les sondes avec le *nom* correspondant sont associées. L'argument `-n` peut être placé en suffixe avec une clause de sonde D facultative. Plusieurs options `-n` à la fois peuvent être spécifiées sur la ligne de commande.

`-o` Permet de spécifier le fichier de *output* pour les options `- A` , `-G` et `-l` ou pour les données faisant l'objet d'un suivi. Si l'option `-A` est présente et que `-o` ne l'est pas, le fichier de sortie par défaut est `/kernel/drv/dtrace.conf`. Si l'option `-G` est présente et que l'argument de l'option `-s` est de forme *nom_fichier*`.d` en l'absence de l'option `-o`, le fichier de sortie par défaut est *nom_fichier*`.o` ; sinon, le fichier de sortie par défaut est `d.out`.

`-p` Permet de saisir l'ID de processus spécifié *pid*, mettre en cache ses tableaux de symbole et fermer le programme à son achèvement. Si plusieurs options `-p` sont présentes sur la ligne de commande, `dtrace` se ferme lorsque toutes les commandes sont fermées, en consignant dans un rapport l'état de sortie de chaque processus lorsqu'il se termine. Le premier ID de processus est disponible pour n'importe quel programme D spécifié sur la ligne de commande ou utilisant l'option `-s` par le biais de la variable de macro `$target.` Pour plus d'informations sur les variables de macro, consultez le Chapitre15, “Scripts”.

`-P` Permet de spécifier le nom du fournisseur à suivre ou à répertorier (option `-l`). Le module, la fonction et le nom des descriptions de sondes restantes restent vides et renvoient toutes les sondes, quelles que soient les valeurs de ces champs. L'argument `-P` peut être placé en suffixe avec une clause de sonde D facultative. Plusieurs options `-P` à la fois peuvent être spécifiées sur la ligne de commande.

`-q` Permet de définir le mode silencieux. `dtrace` supprime des messages tels que le nombre de sondes correspondant aux options spécifiées et les programmes D et n'affiche pas les en-têtes de colonne, l'ID de la CPU, l'ID de la sonde ni n'insère de nouvelles lignes dans la sortie. Seules les données suivies et formatées par des instructions de programme D telles que `trace()` et `printf()` sont affichées dans `stdout`.

`-s` Permet de compiler le fichier source du programme D spécifié. Si l'option `-e` est présente, le programme est compilé mais aucune instrumentation n'est activée. Si l'option `-l` est présente, le programme est compilé et l'ensemble des sondes qui lui

correspondent sont répertoriées, mais aucune instrumentation n'est activée. Si aucune option -e ou -l n'est présente, l'instrumentation spécifiée par le programme D est activée et le suivi commence.

-S
: Permet d'afficher le code intermédiaire du compilateur D. Le compilateur D produit un rapport du code intermédiaire généré pour chaque programme D dans stderr.

-U
: Permet d'annuler la définition du *nom* spécifié lors de l'appel cpp(1) (activé avec l'option -C). Cette option transmet l'option - U à chaque appel cpp.

-v
: Permet de définir le mode détaillé. Si l'option -v est spécifiée, dtrace produit un rapport de stabilité du programme montrant la stabilité minimale de l'interface ainsi que le niveau de dépendance des programmes D spécifiés. Les niveaux de stabilité de DTrace sont expliqués plus en détail dans le Chapitre39, “Stabilité”.

-V
: Permet de consigner dans un rapport la version la plus récente de l'interface de programmation D prise en charge par dtrace. Les informations sur la version sont affichées dans stdout et la commande dtrace se ferme. Pour plus d'informations sur les fonctions de versionnage de DTrace, consultez le Chapitre41, “Versionnage”.

-w
: Autorise les actions destructrices dans des programmes D spécifiés avec l'option - s, -P, -m, -f, - n, ou -i. Si l'option -w n'est pas spécifiée, dtrace n'autorise ni la compilation ni l'activation d'un programme D contenant des actions destructrices. Les actions destructrices sont expliquées plus en détail dans le Chapitre10, “Actions et sous-routines”.

-x
: Permet d'activer ou de modifier une option d'exécution DTrace ou une option du compilateur D. Les options sont répertoriées dans le Chapitre16, “Options et paramètres réglables”. Les options booléennes sont activées en spécifiant leur nom. Les options avec des valeurs sont définies en séparant le nom de l'option de sa valeur par un signe égal (=).

-X
: Permet de spécifier le degré de conformité à la norme ISO C devant être sélectionnée lors de l'appel cpp(1) (activé avec l'option -C). L'argument de l'option -X affecte la valeur et la présence de la macro __STDC__ en fonction de la valeur de la lettre de l'argument :

  a (par défaut)
  : ISO C plus extensions de compatibilité K&R, avec les modifications sémantiques requises par ISO C. Ce mode est le mode par défaut si -X n'est pas spécifié. La macro __STDC__ prédéfinie a une valeur égale à 0 lorsque cpp est appelé avec l'option -Xa.

  c (conformité)
  : En conformité avec ISO C, sans les extensions de compatibilité K&R C. La macro prédéfinie __STDC__ a une valeur égale à 1 lorsque cpp est appelé avec l'option -Xc.

s (K&R C) — K&R C uniquement. La macro `__STDC__` n'est pas définie lorsque `cpp` est appelé avec l'option `-Xs`.

t (transition) — ISO C plus les extensions de compatibilité K&R C, sans modification sémantique requise par ISO C. La macro prédéfinie `__STDC__` a une valeur égale à 0 lorsque `cpp` est appelé avec l'option `-Xt`.

Puisque l'option `-X` affecte uniquement la manière dont le compilateur D appelle le préprocesseur C, les options `-Xa` et `- Xt` sont équivalentes du point de vue de D. Les deux options sont fournies afin de réutiliser facilement les paramètres d'un environnement C.

Quel que soit le mode `-X`, les définitions de préprocesseur C supplémentaires suivantes sont toujours spécifiées et valides dans tous les modes :

- `__sun`
- `__unix`
- `__SVR4`
- `__sparc` (sur les systèmes SPARC® uniquement)
- `__sparcv9` (sur des systèmes SPARC® uniquement lorsque des programmes 64 bits sont compilés)
- `__i386` (sur des systèmes x86 uniquement lorsque des programmes 32 bits sont compilés)
- `__amd64` (sur les systèmes x86 uniquement lorsque les programmes 64 bits sont compilés)
- `__`*'uname -s'* _ *'uname -r'*, qui remplace le séparateur décimal dans la sortie de `uname` par un trait de soulignement (_), comme dans `__SunOS_5_10`
- `__SUNW_D=1`
- `__SUNW_D_VERSION=0x`*MMmmmuuu* (où *MM* est la valeur de version majeure hexadécimale, *mmm* est la valeur de version mineure hexadécimale et *uuu* est la valeur de version micro hexadécimale ; pour plus d'informations sur le versionnage DTrace, consultez le Chapitre41, “Versionnage”.

`-Z` — Autorise des descriptions de sonde ne correspondant à aucune sonde. Si l'option `- Z` n'est pas spécifiée `dtrace`consigne l'erreur dans un rapport et se ferme si aucune des descriptions de sonde spécifiées dans les fichiers de programme D (option `- s`) ou sur la ligne de commande (options `-P`, `-m`, `- f`, `-n`, ou `-i`) ne contient de descriptions ne correspondant à aucune sonde connue.

Opérandes

Aucun ou plusieurs arguments supplémentaires peuvent être spécifiés sur la ligne de commande `dtrace` pour définir un ensemble de variables de macro (`$1`, `$2`, et ainsi de suite) à utiliser dans n'importe quel programme D spécifié avec l'option `-s` ou présent sur la ligne de commande. L'utilisation de variables de macro est décrite plus en détail dans le Chapitre15, “Scripts”.

État de sortie

Les valeurs de sortie suivantes sont renvoyées par l'utilitaire `dtrace` :

0 Les requêtes spécifiées se sont achevées sans erreur. Pour les requêtes de programme D, l'état de sortie 0 indique que les programmes ont été compilés correctement, que les sondes ont été activées correctement ou que l'état anonyme a été récupéré sans erreur. `dtrace` renvoie 0 même si les demandes de suivi spécifiées ont rencontré des erreurs ou des abandons.

1 Une erreur fatale s'est produite. Pour les requêtes de programme D, l'état de sortie 1 indique l'échec de la compilation du programme ou l'impossibilité de répondre à la requête spécifiée.

2 Des options de ligne de commande ou des arguments non valides ont été spécifiés.

CHAPITRE 15

Scripts

Vous pouvez utiliser l'utilitaire dtrace(1M) pour créer des fichiers interpréteurs tirés de programmes en D similaires aux scripts de shell que vous pouvez installer en tant qu'outils DTrace interactifs réutilisables. Le compilateur D et la commande `dtrace` fournissent un ensemble de *variables de macro* étendues par le compilateur en D qui facilite la création de scripts à l'aide de DTrace. Ce chapitre fournit des informations de référence sur la fonction de variables de macro et des conseils sur la création de scripts persistants.

Fichiers interpréteurs

De même que votre shell et des utilitaires comme awk(1) et perl(1), vous pouvez utiliser dtrace(1M) pour créer des fichiers interpréteurs exécutables. Un fichier interpréteur commence par une ligne se présentant comme suit :

`#!` *nom_chemin arg*

pathname correspondant au chemin de l'interpréteur et *arg* à un argument en option simple. Lorsqu'un fichier interpréteur est exécuté, le système invoque l'interpréteur spécifié. Si *arg* a été spécifié dans le fichier interpréteur, il est transmis sous forme d'argument à l'interpréteur. Le chemin d'accès au fichier interpréteur lui-même, ainsi que tous les arguments supplémentaires spécifiés lors de son exécution sont ajoutés à la liste des arguments de l'interpréteur. Par conséquent, vous devrez toujours créer des fichiers interpréteurs DTrace à l'aide d'au moins les arguments suivants :

```
#!/usr/sbin/dtrace -s
```

Si votre fichier interpréteur est exécuté, l'argument à l'option `-s` sera donc le nom du chemin d'accès au fichier interpréteur lui-même. `dtrace` va ensuite lire, compiler et exécuter ce fichier comme si vous aviez tapé la commande suivante dans votre shell :

`# dtrace -s` *interpreter-file*

L'exemple suivant montre comment créer et exécuter un fichier interpréteur `dtrace`. Entrez le code source en langage D suivant et enregistrez-le dans le fichier `interp.d` :

```
#!/usr/sbin/dtrace -s
BEGIN
{
    trace("hello");
    exit(0);
}
```

Marquez le fichier `interp.d` comme exécutable et exécutez-le comme suit :

```
# chmod a+rx interp.d
# ./interp.d
dtrace: script './interp.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1      1                           :BEGIN   hello
#
```

N'oubliez pas que la directive `#!` doit comprendre les deux premiers caractères de votre fichier sans espace intermédiaire ou précédent. Le compilateur D ignore automatiquement cette ligne lorsqu'il traite le fichier interpréteur.

`dtrace` utilise `getopt`(3C) pour traiter les options de ligne de commande de manière à vous permettre de combiner plusieurs options dans votre argument interpréteur unique. Par exemple, pour ajouter l'option `-q` à l'exemple précédent, vous pouvez modifier la directive de l'interpréteur comme suit :

```
#!/usr/sbin/dtrace -qs
```

Si vous spécifiez plusieurs lettres d'option, l'option `-s` doit toujours terminer la liste d'options booléennes de sorte que l'argument suivant (le nom du fichier interpréteur) soit traité comme un argument correspondant à l'option `-s`.

Si vous devez spécifier plus d'une option nécessitant un argument dans votre fichier interpréteur, vous ne pourrez pas intégrer toutes vos options et tous vos arguments dans l'argument interpréteur unique. Utilisez plutôt la syntaxe de directive `#pragma D option` pour définir vos options. Toutes les options de ligne de commande de `dtrace` possèdent un équivalent `#pragma` que vous pouvez utiliser, comme illustré dans le Chapitre16, “Options et paramètres réglables”.

Variables de macro

Le compilateur D définit un ensemble de variables de macro que vous pouvez utiliser lors de l'écriture de programmes ou de fichiers interpréteurs en D. Les variables de macro sont des identificateurs précédés du préfixe `$` et sont étendues une seule fois par le compilateur D lors du traitement de votre fichier d'entrée. Le compilateur D fournit les variables de macro suivantes :

TABLEAU 15–1 Variables de macro en D

Nom	Description	Texte de référence
`$[0-9]+`	arguments de la macro	Reportez-vous à la section "Arguments de macro" à la page 192
`$egid`	ID groupe effectif	`getegid(2)`
`$euid`	ID utilisateur effectif	`geteuid(2)`
`$gid`	ID groupe réel	`getgid(2)`
`$pid`	ID de processus	`getpid(2)`
`$pgid`	ID groupe de processus	`getpgid(2)`
`$ppid`	ID processus parent	`getppid(2)`
`$projid`	ID projet	`getprojid(2)`
`$sid`	ID session	`getsid(2)`
`$target`	ID processus cible	Reportez-vous à la section "ID processus cible" à la page 194
`$taskid`	ID tâche	`gettaskid(2)`
`$uid`	ID utilisateur réel	`getuid(2)`

À l'exception de l'argument de macro `$[0-9]+` et de la variable de macro `$target`, les variables de macro sont toutes étendues au nombre entier correspondant aux attributs du système comme l'ID de processus ou l'ID utilisateur. Les variables développent la valeur d'attribut associée au processus `dtrace` courant lui-même ou à n'importe quel processus en cours d'exécution par le compilateur D.

L'utilisation de variables de macro dans les fichiers interpréteurs vous permet de créer des programmes en D persistants que vous n'êtes pas contraint d'éditer à chaque fois que vous souhaitez les utiliser. Par exemple, pour compter tous les appels système à l'exception de ceux exécutés par la commande `dtrace`, vous pouvez utiliser la clause de programme en D suivante dans laquelle figure `$pid` :

```
syscall:::entry
/pid != $pid/
{
    @calls = count();
}
```

Cette clause engendre toujours le résultat souhaité, même si chaque évocation de la commande dtrace a un ID de processus différent.

Vous pouvez utiliser les variables de macro partout où vous pouvez utiliser un nombre entier, un identificateur ou une chaîne dans un programme en D. Les variables de macro ne sont étendues qu'une seule fois (et non de manière récursive) lorsque le fichier d'entrée est analysé. Chaque variable de macro est étendue pour former un jeton d'entrée et ne peut être concaténée avec un autre texte pour produire un seul jeton. Par exemple, si $pid est étendu à la valeur 456, le code en D :

```
123$pid
```

sera étendu aux deux jetons adjacents 123 et 456, engendrant ainsi une syntaxe d'erreur, plutôt qu'au jeton entier unique 123456.

Les variables de macro sont étendues et concaténées au texte adjacent à l'intérieur des descriptions de sondes en D au début des clauses de votre programme. Par exemple, la clause suivante utilise le fournisseur pid de DTrace pour instrumenter la commande dtrace :

```
pid$pid:libc.so:printf:entry
{
    ...
}
```

Les variables de macro ne sont développées qu'une seule fois à l'intérieur de chaque champ de description de sonde. Elles peuvent ne pas contenir de délimiteurs de description de sonde (:).

Arguments de macro

Le compilateur D fournit également un ensemble de variables de macro correspondant à tous les opérandes d'argument supplémentaires spécifiés dans le cadre de l'invocation de la commande dtrace. Il est possible d'accéder à ces *arguments de macro* en utilisant les noms $0 intégrés comme nom de fichier du programme en D ou de la commande dtrace, $1 pour le premier opérande supplémentaire, $2 pour le second opérande, etc. Si vous utilisez l'option dtrace -s, $0 est étendu à la valeur du nom du fichier d'entrée utilisé avec cette option. Pour les programmes en D spécifiés sur la ligne de commande, $0 est étendu à la valeur de argv[0] utilisée pour exécuter dtrace lui-même.

Les arguments de macro peuvent être étendus aux nombres entiers, aux identificateurs ou aux chaînes en fonction de la forme du texte correspondant. De même que toutes les variables de

macro, vous pouvez utiliser les arguments de macro partout où vous pouvez utiliser des jetons de nombre entier, d'identificateur ou de chaîne dans un programme en D. Tous les exemples suivants peuvent constituer des expressions en D valides en adoptant des valeurs d'argument de macro appropriées :

```
execname == $1    /* with a string macro argument */
x += $1           /* with an integer macro argument */
trace(x->$1)      /* with an identifier macro argument */
```

Vous pouvez utiliser les arguments de macro pour créer des fichiers interpréteurs `dtrace` qui agissent comme de vraies commandes Solaris et modifient leur comportement à l'aide des informations spécifiées par un utilisateur ou un autre outil. Par exemple, le fichier interpréteur en D suit les appels système `write(2)` exécutés par un ID de processus particulier :

```
#!/usr/sbin/dtrace -s

syscall::write:entry
/pid == $1/
{
}
```

Si vous rendez ce fichier interpréteur exécutable, vous pouvez spécifier la valeur de `$1` à l'aide d'un argument de ligne de commande supplémentaire dans votre fichier interpréteur :

```
# chmod a+rx ./tracewrite
# ./tracewrite 12345
```

L'invocation de commande qui en résulte compte chaque appel système `write(2)` exécuté par l'ID de processus 12345.

Si votre programme en D référence un argument de macro ne figurant pas sur la ligne de commande, un message d'erreur approprié est imprimé et la compilation de votre programme échoue :

```
# ./tracewrite
dtrace: failed to compile script ./tracewrite: line 4:
  macro argument $1 is not defined
```

Les programmes en D peuvent référencer des arguments de macro non spécifiés si l'option `defaultargs` est définie. Si l'option `defaultargs` est définie, les arguments non spécifiés ont la valeur `0`. Pour plus d'informations sur les options du compilateur D, reportez-vous au Chapitre16, “Options et paramètres réglables”. Le compilateur D engendre également un message d'erreur si d'autres arguments spécifiés sur la ligne de commande ne sont pas référencés par votre programme en D.

Les valeurs des arguments de commande doivent correspondre à la forme d'un nombre entier, d'un identificateur ou d'une chaîne. Si l'argument ne correspond pas à l'un de ces formats, le

compilateur D envoie un message d'erreur approprié. Lorsque vous spécifiez des arguments de macro au format chaîne dans un fichier interpréteur DTrace, ajoutez une paire de guillemets simples supplémentaires pour éviter que les guillemets doubles et le contenu des chaînes ne soient interprétés par votre shell :

```
# ./foo '"a string argument"'
```

Si vous souhaitez que vos arguments de macro en D soient interprétés comme des jetons de chaîne, y compris avec un format de nombre entier ou d'identificateur, ajoutez deux symboles dollar en préfixe au nom de l'argument ou de la variable de macro (par exemple, `$$1`) pour contraindre le compilateur D à interpréter la valeur de l'argument comme une chaîne entre guillemets doubles. Toutes les séquences d'échappement de chaîne en D usuelles (voir le Tableau 2–5) sont étendues à l'intérieur de n'importe quel argument de macro au format chaîne, qu'elles soient référencées sous la forme `$`*arg* ou `$$`*arg* de la macro. Si l'option `defaultargs` est définie, les arguments non spécifiés au format `$$`*arg* possèdent la valeur d'une chaîne vide ("").

ID processus cible

Utilisez la variable de macro `$target` pour créer des scripts applicables au processus utilisateur qui nous intéresse, ce dernier étant sélectionné sur la ligne de commande `dtrace` à l'aide de l'option `-p` ou créé à l'aide de l'option `-c.` Les programmes en D spécifiés sur la ligne de commande ou à l'aide de l'option `-s` sont compilés *après* la création ou l'extraction des processus et l'extension de la variable `$target` à l'ID de traitement des nombres entiers d'un tel processus initial. Par exemple, le script en D suivant peut permettre de déterminer la répartition des appels système exécutés par un processus objet particulier :

```
syscall:::entry
/pid == $target/
{
    @[probefunc] = count();
}
```

Pour déterminer le nombre d'appels système exécutés par la commande `date(1)`, enregistrez le script dans le fichier `syscall.d` et exécutez la commande suivante :

```
# dtrace -s syscall.d -c date
dtrace: script 'syscall.d' matched 227 probes
Fri Jul 30 13:46:06 PDT 2004
dtrace: pid 109058 has exited

  gtime                                                             1
  getpid                                                            1
  getrlimit                                                         1
  rexit                                                             1
```

```
ioctl                                                          1
resolvepath                                                    1
read                                                           1
stat                                                           1
write                                                          1
munmap                                                         1
close                                                          2
fstat64                                                        2
setcontext                                                     2
mmap                                                           2
open                                                           2
brk                                                            4
```

CHAPITRE 16

Options et paramètres réglables

Pour permettre une personnalisation, DTrace propose à ses clients plusieurs degrés de liberté importants. Pour réduire au maximum la nécessité d'un réglage spécifique, DTrace est mis en œuvre à l'aide de valeurs par défaut standard et de stratégies flexibles par défaut. Cependant, certaines situations peuvent nécessiter le réglage du comportement de DTrace au cas par cas. Ce chapitre décrit les options et paramètres réglables DTrace ainsi que les interfaces disponibles pour les modifier.

Options client

DTrace est réglé en définissant ou en activant des options. Les options disponibles sont décrites dans le tableau ci-dessous. Pour certaines options, `dtrace(1M)` propose une option de ligne de commande correspondante.

TABLEAU 16–1 Options client DTrace

Nom de l'option	Valeur	Alias `dtrace`(1M)	Description	Voir le chapitre
`aggrate`	*time*		Vitesse de lecture de groupement	Chapitre9, "Groupements"
`aggsize`	*size*		Taille du tampon de groupement	Chapitre9, "Groupements"
`bufresize`	`auto` ou `manual`		Stratégie de redimensionnement du tampon	Chapitre11, "Tampons et mise en tampon"
`bufsize`	*size*	`-b`	Taille du tampon principal	Chapitre11, "Tampons et mise en tampon"

TABLEAU 16–1 Options client DTrace *(Suite)*

Nom de l'option	Valeur	Alias dtrace(1M)	Description	Voir le chapitre
cleanrate	*time*		Vitesse de nettoyage. À spécifier en nombre par seconde avec le suffixe hz.	Chapitre13, “Suivi spéculatif”
cpu	*scalar*	-c	CPU sur laquelle activer le suivi	Chapitre11, “Tampons et mise en tampon”
defaultargs	—		Autorise des références à des arguments de macro non spécifiés	Chapitre15, “Scripts”
destructive	—	-w	Autorise des actions destructrices	Chapitre10, “Actions et sous-routines”
dynvarsize	*size*		Taille d'espace de variable dynamique	Chapitre3, “Variables”
flowindent	—	-F	Indente une entrée de fonction et la préfixe avec -> ; annule l'indentation du retour de fonction et la préfixe avec <-	Chapitre14, “Utilitaire dtrace(1M)”
grabanon	—	-a	Revendique un état anonyme	Chapitre36, “Suivi anonyme”
jstackframes	*scalar*		Nombre de cadres de la pile jstack() par défaut	Chapitre10, “Actions et sous-routines”
jstackstrsize	*scalar*		Taille d'espace de chaîne par défaut pour jstack()	Chapitre10, “Actions et sous-routines”
nspec	*scalar*		Nombre de spéculations	Chapitre13, “Suivi spéculatif”
quiet	—	-q	Ne fournit que des données explicitement suivies	Chapitre14, “Utilitaire dtrace(1M)”

TABLEAU 16–1 Options client DTrace *(Suite)*

Nom de l'option	Valeur	Alias dtrace(1M)	Description	Voir le chapitre
specsize	*size*		Taille du tampon de spéculation	Chapitre13, “Suivi spéculatif”
strsize	*size*		Taille de chaîne	Chapitre6, “Chaînes de caractères”
stackframes	*scalar*		Nombre de cadres de la pile	Chapitre10, “Actions et sous-routines”
stackindent	*scalar*		Nombre d'espaces à utiliser pour l'indentation de la sortie de stack() et de ustack()	Chapitre10, “Actions et sous-routines”
statusrate	*time*		Vitesse de vérification du statut	
switchrate	*time*		Vitesse de commutation du tampon	Chapitre11, “Tampons et mise en tampon”
ustackframes	*scalar*		Nombre de cadres de la pile utilisateur	Chapitre10, “Actions et sous-routines”

Les valeurs exprimant des tailles peuvent être accompagnées d'un suffixe (facultatif) k, m, g ou t pour indiquer des kiloctets, mégaoctets, gigaoctets et téraoctets, respectivement. Les valeurs exprimant des heures peuvent être accompagnées d'un suffxe (facultatif) ns, us, ms, s ou hz pour indiquer des nanosecondes, microsecondes, millisecondes, secondes et nombre par seconde, respectivement.

Options modificatrices

Des options peuvent être définies dans un script D à l'aide de #pragma D suivi de la chaîne option et du nom de l'option. Si l'option prend une valeur, son nom doit être suivi d'un signe égal (=) et de la valeur. Les exemples suivants illustrent tous des paramètres d'option valides :

```
#pragma D option nspec=4
#pragma D option grabanon
#pragma D option bufsize=2g
#pragma D option switchrate=10hz
#pragma D option aggrate=100us
#pragma D option bufresize=manual
```

La commande dtrace(1M) accepte également la définition d'option via la ligne de commande en tant qu'argument pour l'option -x. Exemple :

```
# dtrace -x nspec=4 -x grabanon -x bufsize=2g \
    -x switchrate=10hz -x aggrate=100us -x bufresize=manual
```

Si une option non valide est spécifiée, dtrace indique que le nom de l'option n'est pas valide et se ferme :

```
# dtrace -x wombats=25
dtrace: failed to set option -x wombats: Invalid option name
#
```

De même, si une valeur d'option n'est pas valide pour l'option concernée, dtrace indique que la valeur n'est pas valide :

```
# dtrace -x bufsize=100wombats
dtrace: failed to set option -x bufsize: Invalid value for specified option
#
```

Si une option est définie plusieurs fois, les paramètres suivants remplacent les paramètres précédents. Certaines options, comme grabanon, peuvent *uniquement* être définies. La présence d'une telle option la définit et vous ne pouvez pas l'annuler ultérieurement.

Les options définies pour une activation anonyme seront honorées par le client DTrace revendiquant l'état anonyme. Pour plus d'informations sur l'activation du suivi anonyme, reportez-vous au Chapitre36, “Suivi anonyme”.

CHAPITRE 17

Fournisseur dtrace

Le fournisseur dtrace offre plusieurs sondes liées à DTrace lui-même. Vous pouvez utiliser ces sondes pour initialiser l'état avant le début du suivi, procéder à son traitement, une fois le suivi effectué et gérer des erreurs d'exécution inattendues dans d'autres sondes.

Sonde BEGIN

La sonde BEGIN se déclenche avant toutes les autres. Aucune autre sonde ne se déclenche avant l'achèvement de toutes les clauses BEGIN. Cette sonde peut être utilisée pour initialiser n'importe quel état nécessaire dans d'autres sondes. L'exemple suivant illustre comment utiliser la sonde BEGIN pour initialiser un tableau associatif pour mapper entre des bits de protection mmap(2) et une représentation textuelle :

```
BEGIN
{
    prot[0] = "---";
    prot[1] = "r--";
    prot[2] = "-w-";
    prot[3] = "rw-";
    prot[4] = "--x";
    prot[5] = "r-x";
    prot[6] = "-wx";
    prot[7] = "rwx";
}

syscall::mmap:entry
{
    printf("mmap with prot = %s", prot[arg2 & 0x7]);
}
```

La sonde BEGIN se déclenche dans un contexte non spécifié. Cela signifie que la sortie de stack() ou ustack(), et la valeur de variables spécifiques au contexte (par exemple, execname),

sont toutes arbitraires. Il convient de ne pas se fier à ces valeurs et de ne pas les interpréter en vue d'en tirer des informations significatives. Aucun argument n'est défini dans la sonde `BEGIN`.

La sonde END

La sonde `END`se déclenche après toutes les autres. Cette sonde ne se déclenche qu'à l'issue de l'achèvement de toutes les autres. Elle peut être utilisée pour procéder au traitement de l'état rassemblé ou pour formater la sortie. L'action `printa()` est par conséquent souvent utilisée dans la sonde `END.` Les sondes `BEGIN` et `END` peuvent être utilisées ensemble pour mesurer le temps total écoulé au cours du suivi :

```
BEGIN
{
    start = timestamp;
}

/*
 * ... other tracing actions...
 */

END
{
    printf("total time: %d secs", (timestamp - start) / 1000000000);
}
```

Pour connaître les autres utilisations courantes de la sonde `END`, reportez-vous aux sections “Normalisation des données” à la page 125 et “`printa()`” à la page 167.

De même qu'avec la sonde `BEGIN`, aucun argument n'est défini pour la sonde `END.` Le contexte dans lequel la sonde `END` se déclenche est arbitraire et il ne faut pas s'y fier.

Lors du suivi avec l'option `bufpolicy` définie sur `fill`, un espace approprié est réservé pour héberger les enregistrements suivis dans la sonde `END.` Pour de plus amples informations, reportez-vous à la section “Stratégie `fill` et sondes `END`” à la page 157.

Remarque – L'action `exit()` provoque l'arrêt du suivi et le déclenchement de la sonde`END.` Toutefois, un certain délai est constaté entre l'appel de l'action `exit()` et le déclenchement de la sonde `END.` Pendant ce délai, aucune sonde ne se déclenche. Après qu'une sonde appelle l'action `exit()`, la sonde `END` ne se déclenche que lorsque le consommateur DTrace détermine que `exit()` a été appelée et qu'il procède à l'arrêt du suivi. La vitesse de vérification du statut de sortie peut être définie avec l'option `statusrate.` Pour plus d'informations, reportez-vous au Chapitre16, “Options et paramètres réglables”.

Sonde ERROR

La sonde ERROR se déclenche lorsqu'une erreur d'exécution se produit lors du déclenchement d'une clause pour une sonde DTrace. Par exemple, si une clause tente de déréférencer un pointeur NULL, la sonde ERROR se déclenche, comme illustré dans l'exemple suivant.

EXEMPLE 17–1 error.d: erreurs d'enregistrement

```
BEGIN
{
    *(char *)NULL;
}

ERROR
{
    printf("Hit an error!");
}
```

Lorsque vous exécutez ce programme, la sortie qui s'affiche est similaire à l'exemple suivant :

```
# dtrace -s ./error.d
dtrace: script './error.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  2      3                           :ERROR Hit an error!
dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address
(0x0) in action #1 at DIF offset 12
dtrace: 1 error on CPU 2
```

Le résultat montre que la sonde ERROR s'est déclenchée et illustre également dtrace(1M) consignant l'erreur dans un rapport. dtrace dispose de sa propre activation de la sonde ERROR pour lui permettre de consigner les erreurs dans un rapport. En utilisant la sonde ERROR, vous pouvez créer votre propre gestion d'erreurs personnalisée.

Les arguments vers la sondeERROR se présentent comme suit :

arg1	L'identificateur de sonde activée (EPID) de la sonde à l'origine de l'erreur.
arg2	L'index de l'action ayant provoqué l'erreur.
arg3	Le décalage DIF de cette action ou -1 s'il ne s'applique pas.
arg4	Le type d'erreur.
arg5	La valeur particulière de ce type d'erreur.

Le tableau ci-dessous décrit les différents types d'erreur et la valeur que prend `arg5` pour chacun :

Valeur `arg4`	Description	Signification `arg5`
`DTRACEFLT_UNKNOWN`	Type d'erreur inconnu	Aucune
`DTRACEFLT_BADADDR`	Accès à une adresse non valide ou non mappée	Adresse accédée
`DTRACEFLT_BADALIGN`	Accès à une mémoire non alignée	Adresse accédée
`DTRACEFLT_ILLOP`	Opération non valide ou illégale	Aucune
`DTRACEFLT_DIVZERO`	Nombre entier divisé par zéro	Aucune
`DTRACEFLT_NOSCRATCH`	Espace de travail insuffisant pour répondre à l'allocation de travail	Aucune
`DTRACEFLT_KPRIV`	Tentative d'accès à une propriété ou une adresse de noyau avec des privilèges insuffisants.	Adresse accédée ou `0` si cela ne s'applique pas.
`DTRACEFLT_UPRIV`	Tentative d'accès à une propriété ou une adresse utilisateur avec des privilèges insuffisants.	Adresse accédée ou `0` si cela ne s'applique pas.
`DTRACEFLT_TUPOFLOW`	Dépassement de capacité de la pile du paramètre interne DTrace	Aucune

Si les actions effectuées dans la sonde `ERROR` elle-même sont à l'origine d'une erreur, celle-ci est abandonnée de manière silencieuse — la sonde `ERROR` n'est pas appelée récursivement.

Stabilité

Le fournisseur `dtrace` utilise le mécanisme de stabilité DTrace pour décrire ses stabilités, comme illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	Stable	Stable	Commune
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Nom	Stable	Stable	Commune
Arguments	Stable	Stable	Commune

CHAPITRE 18

Fournisseur `lockstat`

Le fournisseur `lockstat` propose des sondes pouvant être utilisées pour dériver des statistiques de contention de verrou pour comprendre quasiment tous les aspects du comportement de verrouillage. La commande `lockstat(1M)` est véritablement un consommateur DTrace utilisant le fournisseur `lockstat` pour collecter ses données brutes.

Présentation

Le fournisseur `lockstat` propose les deux types de sonde suivants : sondes d'événement de contention et sondes d'événement de maintien.

Les sondes *Événement de contention* correspondent à la contention sur une primitive de synchronisation, et se déclenchent lorsqu'un thread est contraint d'attendre qu'une ressource soit disponible. Solaris est généralement optimisé pour une non-contention. Une contention prolongée n'est donc pas prévue. Ces sondes doivent être utilisées pour comprendre les cas où une contention se produit. La contention étant relativement rare, l'activation de sondes d'événement de contention n'affecte généralement pas la performance de manière significative.

Les sondes *Événement de maintien* correspondent à l'acquisition, la libération ou toute autre manipulation d'une primitive de synchronisation. Ces sondes peuvent être utilisées pour répondre à des questions arbitraires sur la manipulation des primitives de synchronisation. Solaris acquérant et libérant très souvent des primitives de synchronisation (de l'ordre de millions de fois par seconde et par CPU sur un système occupé), l'activation de sondes d'événement de maintien entraîne un effet de sonde supérieur à l'activation de sondes d'événement de contention. Alors que l'effet de sonde entraîné par leur activation peut être important, il n'est néanmoins pas pathologique. Elles peuvent toujours être activées en toute confiance sur des systèmes de production.

Le fournisseur `lockstat` propose des sondes correspondant aux différentes primitives de synchronisation de Solaris. Ces primitives et les sondes correspondantes sont expliquées dans ce chapitre.

Sondes de verrou adaptatif

Les *verrous adaptatifs* entraînent une exclusion mutuelle dans une section critique, et peuvent être acquis dans la plupart des contextes du noyau. Les verrous adaptatifs ne comprenant que très peu de restrictions liées au contexte, ils incluent une grande majorité de primitives de synchronisation du noyau Solaris. Ces verrous sont adaptatifs de par leur comportement vis-à-vis de la contention : lorsqu'un thread tente d'acquérir un verrou adaptatif maintenu, il détermine si le thread propriétaire est en cours d'exécution sur une CPU. Si le propriétaire est exécuté sur une autre CPU, le thread demandeur effectuera une *rotation*. Si le propriétaire n'est pas exécuté, le thread demandeur se *bloquera*.

Les quatre sondes lockstat appartenant à des verrous adaptatifs sont répertoriées dans le Tableau 18–1. Pour chaque sonde, `arg0` contient un pointeur vers la structure `kmutex_t` qui représente le verrou adaptatif.

TABLEAU 18–1 Sondes de verrou adaptatif

`adaptive-acquire`	Sonde d'événement de maintien se déclenchant immédiatement après l'acquisition d'un verrou adaptatif.
`adaptive-block`	Sonde d'événement de contention se déclenchant après qu'un thread bloqué sur une mutex adaptative maintenue s'est réveillé et après l'acquisition de la mutex. Si les deux sondes sont activées, `adaptive-block` se déclenche *avant* `adaptive-acquire`. Une seule acquisition de verrouillage peut déclencher les sondes `adaptative-block` et `adaptative-spin`. `arg1` pour `adaptive-block` contient la durée de sommeil en nanosecondes.
`adaptive-spin`	Sonde d'événement de contention se déclenchant après qu'un thread qui a tourné sur une mutex adaptative maintenue a acquis la mutex. Si les deux sondes sont activées, `adaptive-spin` se déclenche *avant* `adaptive-acquire`. Une seule acquisition de verrouillage peut déclencher les sondes `adaptative-block` et `adaptative-spin`. `arg1` pour `adaptive-spin` contient la *durée de rotation* : le nombre de nanosecondes effectuées en boucle de rotation avant l'acquisition du verrouillage.
`adaptive-release`	Sonde d'événement de maintien se déclenchant immédiatement après la libération d'un verrou adaptatif.

Sondes de verrou de rotation

Les threads ne peuvent pas bloquer dans certaines situations dans le noyau, comme en cas d'interruption à niveau élevé et tout autre cas de manipulation d'état de dispatcheur. Dans ces cas, cette restriction empêche l'utilisation de verrous adaptatifs. Les *verrous de rotation* sont plutôt utilisés, dans ces situations, pour effectuer une exclusion mutuelle dans des sections critiques. Comme leur nom l'indique, le comportement de ces verrous en cas de contention

consiste à effectuer une rotation jusqu'à ce que le verrou soit libéré par le thread propriétaire. Les trois sondes appartenant aux verrouillages de rotation sont répertoriées dans le Tableau 18–2.

TABLEAU 18–2 Sondes de verrou de rotation

`spin-acquire`	Sonde d'événement de maintien se déclenchant immédiatement après l'acquisition d'un verrou de rotation.
`spin-spin`	Sonde d'événement de contention se déclenchant après qu'un thread qui a effectué une rotation sur un verrou de rotation maintenu a acquis le verrou de rotation. Si les deux sondes sont activées, `spin-spin` se déclenche *avant* `spin-acquire`. `arg1` pour `spin-spin` contient la *durée de rotation* : le nombre de nanosecondes effectuées en état de rotation avant l'acquisition du verrouillage. Le nombre de rotations n'est pas significatif en lui-même, mais il permet de comparer les durées de rotation.
`spin-release`	Sonde d'événement de maintien se déclenchant immédiatement après la libération d'un verrou de rotation.

Les verrous adaptatifs sont bien plus fréquents que les verrous de rotation. Le script suivant affiche les totaux des deux types de verrou, proposant ainsi des données d'observation.

```
lockstat:::adaptive-acquire
/execname == "date"/
{
    @locks["adaptive"] = count();
}

lockstat:::spin-acquire
/execname == "date"/
{
    @locks["spin"] = count();
}
```

Exécutez ce script dans une fenêtre et une commande `date(1)` dans une autre. Une fois le script DTrace terminé, une sortie similaire à l'exemple suivant s'affiche :

```
# dtrace -s ./whatlock.d
dtrace: script './whatlock.d' matched 5 probes
^C
spin                                                            26
adaptive                                                      2981
```

Comme illustré ici, plus de 99 % des verrous acquis lors de l'exécution de la commande `date` sont des verrous adaptatifs. Il peut être surprenant qu'un *si* grand nombre de verrous soit acquis avec une commande aussi simple que `date`. Le grand nombre de verrous est un artefact classique du verrouillage de précision nécessaire sur un système hautement évolutif comme le noyau Solaris.

Verrous de thread

Les *verrous de thread* sont un type spécifique de verrous de rotation utilisés pour verrouiller un thread en vue de changer son état. Les événements de maintien de verrou de thread sont disponibles sous forme de sondes d'événement de maintien de verrou de rotation (c'est-à-dire `spin-acquire` et `spin-release`), mais les événements de contention disposent de sondes propres spécifiques aux verrous de thread. La sonde d'événement de maintien de verrouillage de thread se trouve dans le Tableau 18–3.

TABLEAU 18–3 Sonde de verrou de thread

`thread-spin`	Sonde d'événement de contention se déclenchant après qu'un thread a effectué une rotation sur un verrou de thread. Comme d'autres sondes d'événement de contention, si la sonde d'événement de contention et la sonde d'événement de maintien sont activées, `thread-spin` se déclenche avant `spin-acquire`. Contrairement à d'autres sondes d'événement de contention cependant, thread-spin se déclenche *avant* la réelle acquisition du verrou. Par conséquent, plusieurs déclenchements de la sonde `thread-spin` peuvent correspondre à un seul déclenchement de la sonde `spin-acquire`.

Sondes de verrouillage en lecture/écriture

Les *verrouillages en lecture/écriture* appliquent une stratégie d'autorisation de plusieurs lecteurs *ou* d'un seul rédacteur — mais pas les deux — dans une section critique. Ces verrous sont généralement utilisés pour les structures plus fréquemment recherchées que modifiées et pour lesquelles la durée est importante dans la section critique. Si les durées de section critique sont courtes, les verrouillages en lecture/écriture sont implicitement placés en série sur la mémoire partagée utilisée pour mettre en œuvre le verrou, aucun avantage sur les verrous adaptatifs ne leur étant accordé. Pour plus d'informations sur les verrouillages en lecture/écriture, reportez-vous à `rwlock(9F)`.

Les sondes appartenant à des verrouillages en lecture/écriture sont répertoriées dans le Tableau 18–4 Pour chaque sonde, `arg0` contient un pointeur vers la structure `krwlock_t` qui représente le verrou adaptatif.

TABLEAU 18–4 Sondes de verrouillage en lecture/écriture

`rw-acquire`	Sonde d'événement de maintien se déclenchant immédiatement après l'acquisition d'un verrou en lecture/écriture. `arg1` contient la constante `RW_READER` si le verrou a été acquis en tant que lecteur, et `RW_WRITER` s'il a été acquis en tant que rédacteur.

TABLEAU 18–4 Sondes de verrouillage en lecture/écriture *(Suite)*

`rw-block`	Sonde d'événement de contention se déclenchant après qu'un thread bloqué sur un verrou en lecture/écriture maintenu s'est réveillé et après l'acquisition du verrou. `arg1` contient la durée (en nanosecondes) pendant laquelle le thread actuel doit sommeiller pour acquérir le verrou. `arg2` contient la constante `RW_READER` si le verrou a été acquis en tant que lecteur, et `RW_WRITER` s'il a été acquis en tant que rédacteur. `arg3` et `arg4` contiennent plus d'informations sur le motif du blocage. `arg3` n'est pas de zéro si et seulement si le verrou était maintenu en tant que lecteur au moment du blocage du thread actuel. `arg4` contient le nombre de lecteurs au moment du blocage du thread actuel. Si les deux sondes `rw-block` et `rw-acquire` sont activées, `rw-block` se déclenche *avant* `rw-acquire`.
`rw-upgrade`	Sonde d'événement de maintien se déclenchant après qu'un thread a mis à niveau un verrou en lecture/écriture de lecture à écriture. Les mises à niveau ne comprennent pas d'événement de contention associé, car cela n'est possible que via une interface non bloquante, `rw_tryupgrade`(9F).
`rw-downgrade`	Sonde d'événement de maintien se déclenchant après qu'un thread a rétrogradé sa propriété de verrou en lecture/écriture de écriture à lecture. Les mises à niveau inférieures ne comprennent pas d'événement de contention associé car elles réussissent toujours sans contention.
`rw-release`	Sonde d'événement de maintien se déclenchant immédiatement après la libération d'un verrou en lecture/écriture. `arg1` contient la constante `RW_READER` si le verrou libéré a été maintenu en tant que lecteur, et `RW_WRITER` s'il a été maintenu en tant que rédacteur. En raison des mises à niveau et rétrogradations, le verrou peut *ne pas* être libéré comme il a été acquis.

Stabilité

Le fournisseur `lockstat` utilise un mécanisme de stabilité DTrace pour décrire ses stabilités, comme illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	Commune
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	Commune
Arguments	En cours d'évolution	En cours d'évolution	Commune

CHAPITRE 19

Fournisseur `profile`

Le fournisseur `profile` propose des sondes associées à un déclenchement d'interruption basé sur le temps à chaque intervalle de temps défini. Ces sondes *non ancrées* ne sont pas associées à un point d'exécution particulier, mais plutôt à l'événement d'interruption asynchrone. Ces sondes peuvent être utilisées pour échantillonner un aspect donné de l'état système à chaque unité de temps et les échantillons peuvent alors être utilisés pour inférer le comportement du système. Si la vitesse d'échantillonnage est élevée ou si la durée d'échantillonnage est longue, une inférence précise est possible. À l'aide d'actions DTrace, le fournisseur `profile` peut être utilisé pour échantillonner n'importe quel élément du système de manière pratique. Par exemple, vous pourriez échantillonner l'état du thread en cours, l'état de la CPU ou l'instruction de la machine en cours.

Remarque – Les sondes n'ont pas accès aux variables locales de thread à partir du fournisseur `profile`. L'utilisation de l'identifiant spécial `self` avec ce type de sonde pour référencer une variable locale de thread ne renverra aucune sortie.

Sondes `profile-`*n*

Une sonde `profile-`*n* se déclenche à chaque intervalle défini sur chaque CPU à un niveau élevé d'interruption. L'intervalle de déclenchement de la sonde est indiqué par la valeur de *n* : la source d'interruption se déclenche *n* fois par seconde. *n* peut également porter un suffixe de temps facultatif, auquel cas *n* est dans les unités exprimées par le suffixe. Les suffixes valides et les unités qu'ils expriment sont répertoriés dans le Tableau 19–1.

TABLEAU 19–1 Suffixes d'heure valides

Suffixe	Unités de temps
`nsec` ou `ns`	nanosecondes

TABLEAU 19–1 Suffixes d'heure valides *(Suite)*

Suffixe	Unités de temps
`usec` ou `us`	microsecondes
`msec` ou `ms`	millièmes de seconde
`sec` ou `s`	secondes
`min` ou `m`	minutes
`hour` ou `h`	heures
`day` ou `d`	jours
`hz`	hertz (fréquence par seconde)

L'exemple suivant crée une sonde se déclenchant à 97 hertz pour échantillonner le processus en cours d'exécution :

```
#pragma D option quiet

profile-97
/pid != 0/
{
    @proc[pid, execname] = count();
}

END
{
    printf("%-8s %-40s %s\n", "PID", "CMD", "COUNT");
    printa("%-8d %-40s %@d\n", @proc);
}
```

L'exécution de l'exemple ci-dessus sur une courte période donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./prof.d
^C
PID      CMD                                      COUNT
223887   sh                                       1
100360   httpd                                    1
100409   mibiisa                                  1
223887   uname                                    1
218848   sh                                       2
218984   adeptedit                                2
100224   nscd                                     3
3        fsflush                                  4
2        pageout                                  6
100372   java                                     7
```

```
115279   xterm                                  7
100460   Xsun                                   7
100475   perfbar                                9
223888   prstat                                 15
```

Vous pouvez également utiliser le fournisseur `profile-`*n* pour échantillonner des informations sur le processus en cours d'exécution. L'exemple suivant de script D utilise une sonde profile de 1 001 hertz pour échantillonner la priorité actuelle d'un processus spécifié :

```
profile-1001
/pid == $1/
{
    @proc[execname] = lquantize(curlwpsinfo->pr_pri, 0, 100, 10);
}
```

Pour voir cet exemple de script en action, entrez les commandes suivantes dans une fenêtre :

```
$ echo $$
12345
$ while true ; do let i=0 ; done
```

Dans une autre fenêtre, exécutez le script D sur une courte durée, en remplaçant *12345* par l'ID de processus renvoyé par la commande `echo` :

```
# dtrace -s ./profpri.d 12345
 dtrace: script './profpri.d' matched 1 probe
^C
ksh
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@                   7443
              10 |@@@@@@                                   2235
              20 |@@@@                                     1679
              30 |@@@                                      1119
              40 |@                                        560
              50 |@                                        554
              60 |                                         0
```

Cette sortie illustre la polarisation de la classe de planification du partage de temps. Le processus de shell étant en rotation sur la CPU, sa priorité est sans cesse diminuée par le système. Si le processus de shell était exécuté moins fréquemment, sa priorité serait supérieure. Pour voir ce résultat, appuyez sur Ctrl+C dans le shell en rotation, puis réexécutez le script :

```
# dtrace -s ./profpri.d 494621
 dtrace: script './profpri.d' matched 1 probe
```

Dans le shell, entrez maintenant quelques caractères. Une fois le script DTrace terminé, une sortie similaire à l'exemple suivant s'affiche :

```
ksh
           value  ------------ Distribution ------------ count
              40 |                                         0
              50 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 14
              60 |                                         0
```

Le processus de shell étant auparavant au repos dans l'attente d'une saisie utilisateur plutôt qu'en rotation sur la CPU, son *exécution* s'est faite à une priorité bien plus élevée.

Sondes `tick-`*n*

Comme les sondes `profile-` *n*, les sondes `tick-` *n* se déclenchent à chaque intervalle défini à un niveau d'interruption élevé. Cependant, contrairement aux sondes `profile-` *n*, qui se déclenchent sur *chaque* CPU, les sondes `tick-` *n* ne se déclenchent que sur *une* CPU par intervalle. La CPU concernée peut changer dans le temps. Comme pour les sondes `profile-` *n*, *n* est défini par défaut sur vitesse par seconde mais peut également être accompagné d'un suffixe de temps facultatif. Les sondes `tick-` *n* ont plusieurs utilisations, comme la fourniture d'une sortie périodique ou l'exécution d'une action périodique.

Arguments

Les arguments de sondes `profile` sont les suivants :

`arg0`	Compteur de programme dans le noyau au moment du déclenchement de la sonde, ou 0 si le processus en cours n'était pas exécuté dans le noyau au moment du déclenchement de la sonde
`arg1`	Compteur de programme dans le processus au niveau utilisateur au moment du déclenchement de la sonde, ou 0 si le processus en cours était exécuté au niveau du noyau au moment du déclenchement de la sonde

Comme indiqué dans leurs descriptions, si `arg0` n'est pas de zéro, `arg1` l'est ; si `arg0` est de zéro, `arg1` ne l'est pas. Ainsi, vous pouvez utiliser `arg0` et `arg1` pour différencier le niveau utilisateur du niveau noyau, comme dans l'exemple simple suivant :

```
profile-1ms
{
    @ticks[arg0 ? "kernel" : "user"] = count();
}
```

Résolution de l'horloge

Le fournisseur `profile` utilise des horloges d'intervalle de résolution arbitraire du système d'exploitation. Dans des architectures ne prenant pas en charge des interruptions basées sur le temps à résolution arbitraire, la fréquence est limitée par la fréquence d'horloge système qui est spécifiée par la variable de noyau `hz`. Les sondes de fréquence supérieure à `hz` dans de telles architectures se déclenchent un certain nombre de fois toutes les 1/ `hz` secondes. Par exemple, une sonde `profile` 1 000 hertz dans une telle architecture avec `hz` définie sur 100 se déclenchera dix fois de manière rapprochée toutes les dix millisecondes. Sur des plates-formes prenant en charge une résolution arbitraire, une sonde `profile` 1 000 hertz se déclencherait chaque milliseconde.

L'exemple suivant teste la résolution d'une architecture :

```
profile-5000
{
    /*
     * We divide by 1,000,000 to convert nanoseconds to milliseconds, and
     * then we take the value mod 10 to get the current millisecond within
     * a 10 millisecond window.  On platforms that do not support truly
     * arbitrary resolution profile probes, all of the profile-5000 probes
     * will fire on roughly the same millisecond.  On platforms that
     * support a truly arbitrary resolution, the probe firings will be
     * evenly distributed across the milliseconds.
     */
    @ms = lquantize((timestamp / 1000000) % 10, 0, 10, 1);
}

tick-1sec
/i++ >= 10/
{
    exit(0);
}
```

Dans une architecture prenant en charge des sondes `profile` à résolution arbitraire, l'exécution de l'exemple de script entraîne une distribution régulière :

```
# dtrace -s ./restest.d
 dtrace: script './restest.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  33631                       :tick-1sec


           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@                                      10760
               1 |@@@@                                     10842
```

```
               2 |@@@@                                     10861
               3 |@@@                                      10820
               4 |@@@                                      10819
               5 |@@@                                      10817
               6 |@@@@                                     10826
               7 |@@@@                                     10847
               8 |@@@@                                     10830
               9 |@@@@                                     10830
```

Dans une architecture ne prenant pas en charge des sondes `profile` à résolution arbitraire, l'exécution de l'exemple de script entraîne une distribution irrégulière :

```
# dtrace -s ./restest.d
 dtrace: script './restest.d' matched 2 probes
 CPU     ID                    FUNCTION:NAME
  0  28321                       :tick-1sec


           value  ------------- Distribution ------------- count
               4 |                                         0
               5 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  107864
               6 |                                         424
               7 |                                         255
               8 |                                         496
               9 |                                         0
```

Dans ces architectures, `hz` peut être réglée manuellement dans `/etc/system` afin d'améliorer la résolution effective.

Toutes les variantes de UltraSPARC (`sun4u`) prennent actuellement en charge les sondes `profile` à résolution arbitraire. De nombreuses variantes de l'architecture x86 (`i86pc`) prennent également en charge les sondes `profile` à résolution arbitraire, alors que ce n'est pas le cas de certaines variantes antérieures.

Création de sonde

Contrairement à d'autres fournisseurs, `profile` crée des sondes de manière dynamique à la demande. Ainsi, la sonde profile souhaitée peut ne pas apparaître dans une liste de toutes les sondes (via `dtrace -l -P profile` par exemple), mais la sonde est créée lorsqu'elle est activée.

Dans des architectures prenant en charge les sondes `profile` à résolution arbitraire, un intervalle de temps trop court entraînerait sans cesse des interruptions basées sur le temps, refusant ainsi le service sur cette machine. Pour éviter cela, le fournisseur `profile` refuse silencieusement de créer toute sonde qui pourrait entraîner un intervalle de moins de deux cents microsecondes.

Stabilité

Le fournisseur profile utilise un mécanisme de stabilité DTrace pour décrire ses stabilités, comme illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	Commune
Module	Instable	Instable	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	Commune
Arguments	En cours d'évolution	En cours d'évolution	Commune

◆ ◆ ◆ CHAPITRE 20

Fournisseur fbt

Ce chapitre décrit le fournisseur Function Boundary Tracing (FBT), qui fournit des sondes associées à l'entrée et au retour de la plupart des fonctions du noyau de Solaris. Cette fonction constitue l'unité fondamentale du texte du programme. Dans un système bien conçu, chaque fonction effectue une opération discrète précise sur un ou plusieurs objets similaires spécifiés. Par conséquent, même sur les plus petits systèmes Solaris, FBT fournit approximativement 20 000 sondes.

À l'instar d'autres fournisseurs DTrace, FBT n'a d'effets sur les sondes que s'il est activé de manière explicite. Lorsqu'il est activé FBT provoque uniquement des effets sur les fonctions sondées. Puisque l'implémentation de FBT est spécifique à l'architecture du jeu d'instructions, FBT a été implémenté sur une plate-forme SPARC et sur une plate-forme x86. Pour chaque jeu d'instructions, il existe un petit nombre de fonctions qui n'en appellent aucune autre, optimisées par le compilateur et que *FBT* ne peut pas instrumenter. On les appelle les fonctions terminales. Les sondes correspondant à ces fonctions ne sont pas présentes dans DTrace.

L'utilisation efficace des sondes FBT requiert une bonne connaissance de l'implémentation du système d'exploitation. Par conséquent, il est recommandé d'utiliser FBT uniquement lors du développement du logiciel de noyau ou lorsque d'autres fournisseurs ne suffisent pas. D'autres fournisseurs DTrace, dont `syscall`, `sched`, `proc` et `io`, peuvent être utilisés pour répondre à la plupart des questions portant sur l'analyse du système, sans qu'aucune connaissance en matière d'implémentation du système d'exploitation ne soit nécessaire.

Sondes

FBT fournit une sonde à la *limite* de la plupart des fonctions du noyau. La limite d'une fonction est franchie lors de l'entrée dans la fonction et lors du retour de celle-ci. FBT fournit ainsi deux rôles à chaque fonction du noyau : l'un à l'entrée dans la fonction, l'autre au retour de la fonction. Ces sondes sont appelées `entry` et `return`, respectivement. Le nom de la fonction et le nom du module sont spécifiés comme faisant partie de la sonde. Toutes les sondes FBT précisent un nom de fonction et un nom de module.

Arguments des sondes

Sondes `entry`

Les arguments des sondes `entry` sont identiques à ceux de la fonction du noyau du système d'exploitation correspondant. Il est possible d'accéder à ces arguments dans un mode de saisie en utilisant le tableau `args[]`. Il est possible d'accéder à ces arguments en tant que `int64_t` en utilisant `arg0` .. Variables `arg`*n*.

Sondes `return`

Une fonction donnée ne dispose que d'un seul point d'entrée mais peut présenter de nombreux points différents lors du retour vers le programme appelant. Généralement, vous vous intéressez à la valeur renvoyée par une fonction ou au fait que la fonction est renvoyée vers tous les chemins et pas uniquement le chemin de retour spécifique. FBT collecte donc plusieurs sites de retour d'une fonction dans une sonde unique `return.` Si le chemin de retour précis présente un intérêt, étudiez la valeur `args[0]` de la sonde `return` qui indique le *décalage* (en octets) de l'instruction de retour dans le texte de la fonction.

Si la fonction a une valeur de retour, celle-ci est stockée dans `args[1]`. Dans le cas contraire, `args[1]` n'est pas défini.

Exemples

Vous pouvez utiliserFBT pour explorer facilement l'implémentation du noyau. Le script suivant, donné à titre d'exemple, enregistre le premier `ioctl(2)` d'un processus `xclock` quelconque puis vient après le chemin de code suivant dans le noyau :

```
/*
 * To make the output more readable, we want to indent every function entry
 * (and unindent every function return).  This is done by setting the
 * "flowindent" option.
 */
#pragma D option flowindent

syscall::ioctl:entry
/execname == "xclock" && guard++ == 0/
{
    self->traceme = 1;
    printf("fd: %d", arg0);
}
```

```
fbt:::
/self->traceme/
{}

syscall::ioctl:return
/self->traceme/
{
    self->traceme = 0;
    exit(0);
}
```

Exécuter ce script engendre une sortie identique à l'exemple suivant :

```
# dtrace -s ./xioctl.d
dtrace: script './xioctl.d' matched 26254 probes
CPU FUNCTION
  0  => ioctl                                 fd: 3
  0    -> ioctl
  0      -> getf
  0        -> set_active_fd
  0        <- set_active_fd
  0      <- getf
  0      -> fop_ioctl
  0        -> sock_ioctl
  0          -> strioctl
  0            -> job_control_type
  0            <- job_control_type
  0            -> strcopyout
  0              -> copyout
  0              <- copyout
  0            <- strcopyout
  0          <- strioctl
  0        <- sock_ioctl
  0      <- fop_ioctl
  0      -> releasef
  0        -> clear_active_fd
  0        <- clear_active_fd
  0        -> cv_broadcast
  0        <- cv_broadcast
  0      <- releasef
  0    <- ioctl
  0  <= ioctl
```

La sortie montre qu'un processus `xclock` a appelé `ioctl()` sur un descripteur de fichier qui semble être associé à un socket.

Vous pouvez également utiliser FBT pour comprendre les pilotes de noyau. Par exemple, le pilote `ssd`(7D) dispose de nombreux chemins de code par l'intermédiaire desquels `EIO` est

susceptible d'être renvoyé. FBT peut facilement être utilisé pour déterminer le chemin de code précis ayant engendré une condition d'erreur, comme illustré dans l'exemple suivant :

```
fbt:ssd::return
/arg1 == EIO/
{
    printf("%s+%x returned EIO.", probefunc, arg0);
}
```

Pour plus d'informations sur les retours de `EIO`, il est possible de procéder au suivi spéculatif des sondes`fbt` puis d'exécuter `commit()`(ou `discard()`) en fonction de la valeur de retour d'une fonction spécifique. Pour de plus amples informations sur le suivi spéculatif, reportez-vous au Chapitre13, “Suivi spéculatif”.

Sinon, vous pouvez utiliser FBT pour comprendre les fonctions appelées au sein d'un module spécifié. L'exemple suivant répertorie toutes les fonctions appelées dans UFS :

```
# dtrace -n fbt:ufs::entry'{@a[probefunc] = count()}'
dtrace: description 'fbt:ufs::entry' matched 353 probes
^C
  ufs_ioctl                                                         1
  ufs_statvfs                                                       1
  ufs_readlink                                                      1
  ufs_trans_touch                                                   1
  wrip                                                              1
  ufs_dirlook                                                       1
  bmap_write                                                        1
  ufs_fsync                                                         1
  ufs_iget                                                          1
  ufs_trans_push_inode                                              1
  ufs_putpages                                                      1
  ufs_putpage                                                       1
  ufs_syncip                                                        1
  ufs_write                                                         1
  ufs_trans_write_resv                                              1
  ufs_log_amt                                                       1
  ufs_getpage_miss                                                  1
  ufs_trans_syncip                                                  1
  getinoquota                                                       1
  ufs_inode_cache_constructor                                       1
  ufs_alloc_inode                                                   1
  ufs_iget_alloced                                                  1
  ufs_iget_internal                                                 2
  ufs_reset_vnode                                                   2
  ufs_notclean                                                      2
  ufs_iupdat                                                        2
  blkatoff                                                          3
  ufs_close                                                         5
```

```
  ufs_open                                                          5
  ufs_access                                                        6
  ufs_map                                                           8
  ufs_seek                                                         11
  ufs_addmap                                                       15
  rdip                                                             15
  ufs_read                                                         15
  ufs_rwunlock                                                     16
  ufs_rwlock                                                       16
  ufs_delmap                                                       18
  ufs_getattr                                                      19
  ufs_getpage_ra                                                   24
  bmap_read                                                        25
  findextent                                                       25
  ufs_lockfs_begin                                                 27
  ufs_lookup                                                       46
  ufs_iaccess                                                      51
  ufs_imark                                                        92
  ufs_lockfs_begin_getpage                                        102
  bmap_has_holes                                                  102
  ufs_getpage                                                     102
  ufs_itimes_nolock                                               107
  ufs_lockfs_end                                                  125
  dirmangled                                                      498
  dirbadname                                                      498
```

Si vous connaissez l'objectif ou les arguments d'une fonction de noyau, vous pouvez utiliser FBT pour comprendre de quelle manière et pour quelle raison la fonction est appelée. Par exemple, `putnext`(9F) prend un pointeur dans une structure `queue`(9S) comme premier membre. Le membre `q_qinfo` de la structure queue est un pointeur sur une structure `qinit`(9S). Le membre `qi_minfo` de la structure `qinit` dispose d'un pointeur sur une structure `module_info`(9S), qui contient le nom du module dans son membre `mi_idname`. L'exemple suivant rassemble ces informations en utilisant la sonde FBT dans `putnext` pour procéder au suivi des appels `putnext`(9F) par nom de module :

```
fbt::putnext:entry
{
    @calls[stringof(args[0]->q_qinfo->qi_minfo->mi_idname)] = count();
}
```

Exécuter le script ci-dessus engendre une sortie similaire à l'exemple suivant :

```
# dtrace -s ./putnext.d
^C

  iprb                                                              1
  rpcmod                                                            1
  pfmod                                                             1
```

```
timod                                                               2
vpnmod                                                              2
pts                                                                40
conskbd                                                            42
kb8042                                                             42
tl                                                                 58
arp                                                               108
tcp                                                               126
ptm                                                               249
ip                                                                313
ptem                                                              340
vuid2ps2                                                          361
ttcompat                                                          412
ldterm                                                            413
udp                                                               569
strwhead                                                          624
mouse8042                                                         726
```

Vous pouvez également utiliser FBT pour déterminer le temps écoulé dans une fonction donnée. L'exemple suivant montre comment déterminer les programmes appelants des routines de délai DDI `drv_usecwait(9F)` et `delay(9F)`.

```
fbt::delay:entry,
fbt::drv_usecwait:entry
{
    self->in = timestamp
}

fbt::delay:return,
fbt::drv_usecwait:return
/self->in/
{
    @snoozers[stack()] = quantize(timestamp - self->in);
    self->in = 0;
}
```

Il est particulièrement intéressant d'exécuter ce script, donné à titre d'exemple, pendant l'initialisation. Le Chapitre36, “Suivi anonyme” décrit la procédure permettant de réaliser un suivi anonyme pendant l'initialisation du système. Lors de la réinitialisation, une sortie similaire à l'exemple suivant s'affichera vraisemblablement :

```
# dtrace -ae

              ata`ata_wait+0x34
              ata`ata_id_common+0xf5
              ata`ata_disk_id+0x20
              ata`ata_drive_type+0x9a
              ata`ata_init_drive+0xa2
```

```
    ata‘ata_attach+0x50
    genunix‘devi_attach+0x75
    genunix‘attach_node+0xb2
    genunix‘i_ndi_config_node+0x97
    genunix‘i_ddi_attachchild+0x4b
    genunix‘devi_attach_node+0x3d
    genunix‘devi_config_one+0x1d0
    genunix‘ndi_devi_config_one+0xb0
    devfs‘dv_find+0x125
    devfs‘devfs_lookup+0x40
    genunix‘fop_lookup+0x21
    genunix‘lookuppnvp+0x236
    genunix‘lookuppnat+0xe7
    genunix‘lookupnameat+0x87
    genunix‘cstatat_getvp+0x134

 value  ------------- Distribution ------------- count
  2048 |                                         0
  4096 |@@@@@@@@@@@@@@@@@@@@@                    4105
  8192 |@@@@                                     783
 16384 |@@@@@@@@@@@@@@                           2793
 32768 |                                         16
 65536 |                                         0


    kb8042‘kb8042_wait_poweron+0x29
    kb8042‘kb8042_init+0x22
    kb8042‘kb8042_attach+0xd6
    genunix‘devi_attach+0x75
    genunix‘attach_node+0xb2
    genunix‘i_ndi_config_node+0x97
    genunix‘i_ddi_attachchild+0x4b
    genunix‘devi_attach_node+0x3d
    genunix‘devi_config_one+0x1d0
    genunix‘ndi_devi_config_one+0xb0
    genunix‘resolve_pathname+0xa5
    genunix‘ddi_pathname_to_dev_t+0x16
    consconfig_dacf‘consconfig_load_drivers+0x14
    consconfig_dacf‘dynamic_console_config+0x6c
    consconfig‘consconfig+0x8
    unix‘stubs_common_code+0x3b

 value  ------------- Distribution ------------- count
  262144 |                                         0
  524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      221
 1048576 |@@@@                                     29
 2097152 |                                         0
```

```
              usba‘hubd_enable_all_port_power+0xed
              usba‘hubd_check_ports+0x8e
              usba‘usba_hubdi_attach+0x275
              usba‘usba_hubdi_bind_root_hub+0x168
              uhci‘uhci_attach+0x191
              genunix‘devi_attach+0x75
              genunix‘attach_node+0xb2
              genunix‘i_ndi_config_node+0x97
              genunix‘i_ddi_attachchild+0x4b
              genunix‘i_ddi_attach_node_hierarchy+0x49
              genunix‘attach_driver_nodes+0x49
              genunix‘ddi_hold_installed_driver+0xe3
              genunix‘attach_drivers+0x28

           value  ------------- Distribution ------------- count
        33554432 |                                         0
        67108864 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
       134217728 |                                         0
```

Optimisation des appels terminaux

Lorsqu'une fonction finit par en appeler une autre, le compilateur peut s'engager dans une *optimisation des appels terminaux*, au cours de laquelle la fonction appelée réutilise le cadre de pile du programme appelant. Cette procédure est le plus souvent utilisée dans l'architecture SPARC, où le compilateur réutilise la fenêtre d'enregistrement du programme appelant dans la fonction appelée, afin de réduire la pression dans la fenêtre d'enregistrement.

La présence de cette optimisation provoque le déclenchement de la fonction d'appel de la sonde `return` *avant* la sonde `entry` de la fonction appelée. Ce tri peut s'avérer confus. Par exemple, si vous souhaitez enregistrer toutes les fonctions appelées à partir d'une fonction donnée ainsi que toutes les fonctions appelées par cette même fonction, vous pouvez utiliser le script suivant :

```
fbt::foo:entry
{
    self->traceme = 1;
}

fbt:::entry
/self->traceme/
{
    printf("called %s", probefunc);
}

fbt::foo:return
/self->traceme/
```

```
{
    self->traceme = 0;
}
```

Toutefois, si `foo()` se termine par un appel terminal optimisé, la fonction d'appel terminal et par conséquent toutes les fonctions appelées par cette dernière ne seront pas capturées. Il est impossible d'annuler l'optimisation du noyau de manière dynamique à la volée et DTrace ne donne aucune information erronée à propos de la structure du code. Par conséquent, vous devez savoir à quel moment l'optimisation des appels terminaux peut être utilisée.

L'optimisation des appels terminaux est susceptible d'être utilisée dans un code source similaire à l'exemple suivant :

```
    return (bar());
```

ou dans un code source similaire à ce qui suit :

```
    (void) bar();
    return;
```

À l'inverse, un code source de fonction se terminant comme dans l'exemple suivant *ne peut pas* avoir son appel à `bar()` optimisé, car l'appel à `bar()` n'est pas un appel terminal :

```
    bar();
    return (rval);
```

Vous pouvez déterminer si un appel a fait l'objet d'une optimisation d'appels terminaux en ayant recours à la technique suivante :

- Lors de l'exécution de DTrace, procédez au suivi de `arg0` de la sonde `return` en question. `arg0` contient le décalage de l'instruction de retour de la fonction.
- Après l'arrêt de DTrace, utilisez `mdb(1)` pour observer la fonction. Si le décalage ayant fait l'objet d'un suivi contient un appel à une autre fonction au lieu d'une instruction à renvoyer de la fonction, cela signifie que l'appel a fait l'objet d'une optimisation d'appels terminaux.

En raison de l'architecture du jeu d'instructions, l'optimisation d'appels terminaux est beaucoup plus courante sur les systèmes SPARC que sur les systèmes x86. L'exemple suivant utilise `mdb` pour découvrir l'optimisation d'appels terminaux dans la fonction `dup()` du noyau :

```
# dtrace -q -n fbt::dup:return'{printf("%s+0x%x", probefunc, arg0);}'
```

Lorsque cette commande est en cours d'exécution, exécutez un programme qui effectue un `dup(2)`, tel qu'un processus `bash`. La commande ci-dessus doit fournir une sortie similaire à l'exemple suivant :

```
dup+0x10
^C
```

À présent, étudiez la fonction avec `mdb` :

```
# echo "dup::dis" | mdb -k
dup:                            sra      %o0, 0, %o0
dup+4:                          mov      %o7, %g1
dup+8:                          clr      %o2
dup+0xc:                        clr      %o1
dup+0x10:                       call     -0x1278       <fcntl>
dup+0x14:                       mov      %g1, %o7
```

La sortie montre que `dup+0x10` est un appel à la fonction `fcntl()` et non une instruction `ret.` Par conséquent, l'appel à `fcntl()` constitue un exemple d'optimisation d'appels terminaux.

Fonctions d'assemblage

Vous pouvez rencontrer des fonctions qui semblent entrer mais n'être jamais renvoyées ou inversement. Ces fonctions rares sont généralement des routines d'assemblage codées manuellement qui se connectent à la partie centrale d'autres fonctions d'assemblage codées manuellement. Ces fonctions ne doivent pas empêcher l'analyse : La fonction connectée à l'entrée doit toujours renvoyer au programme appelant de la fonction connectée au retour. Cela signifie que si vous activez toutes les sondes FBT vous devez voir l'entrée dans une fonction et le retour d'une autre fonction à la même profondeur de pile.

Limitations du jeu d'instructions

Certaines fonctions ne peuvent pas être instrumentées par FBT. La nature exacte des fonctions instrumentables sont spécifiques à l'architecture du jeu d'instructions.

Limitations x86

Les fonctions ne créant pas de cadre de pile sur les systèmes x86 ne peuvent pas être instrumentées par FBT. Étant donné que le jeu d'enregistrement pour les systèmes x86 est extraordinairement petit, la plupart des fonctions doivent placer les données sur la pile et par conséquent, créer un cadre de pile. Toutefois, certaines fonctions x86 ne créent pas de cadre de pile et ne peuvent donc pas être instrumentées. Les chiffres réels varient, mais en règle générale, moins de cinq pour cent des fonctions ne peuvent pas être instrumentées sur la plate-forme x86.

Limitations SPARC

Les routines terminales codées manuellement dans un langage d'assemblage sur les systèmes SPARC ne peuvent pas être instrumentées par FBT. La majorité du noyau est écrit en C et toutes les fonctions écrites en C peuvent être instrumentées par FBT.

Interaction du point d'arrêt

FBT fonctionne en modifiant de manière dynamique le texte de noyau. Étant donné que les points d'arrêt fonctionnent également en modifiant le texte de noyau, si un point d'arrêt du noyau est placé à un site d'entrée ou de retour *avant* le chargement de DTrace, FBT refuse de fournir une sonde pour la fonction, même si le point d'arrêt du noyau est supprimé ultérieurement. Si le point d'arrêt du noyau est placé *après* le chargement de DTrace, le point d'arrêt du noyau et la sonde DTrace correspondront au même point du texte. Dans cette situation, le point d'arrêt se déclenche en premier, puis la sonde se déclenche lorsque le programme de débogage effectue une reprise sur le noyau. Il est recommandé de ne pas utiliser les points d'arrêt simultanément avec DTrace. Si des points d'arrêt sont requis, utilisez l'action `breakpoint()` de DTrace à la place.

Chargement des modules

Le noyau Solaris peut charger et décharger dynamiquement des modules de noyau. Lorsque FBT est chargé et qu'un module est chargé de manière dynamique, FBT fournit automatiquement de nouvelles sondes associées au nouveau module. Si un module chargé dispose de sondes FBT dont *l'activation a été annulée*, le module peut être déchargé ; les sondes correspondantes seront détruites lors du déchargement du module. Si un module chargé dispose de sondes FBT *activées*, le module est considéré occupé et ne peut pas être déchargé.

Stabilité

Le fournisseur FBT utilise le mécanisme de stabilité pour décrire ses stabilités, comme illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	Privé	Privé	ISA

Étant donné que FBT expose l'implémentation du noyau, aucun élément le concernant n'est stable et le nom du module et de la fonction ainsi que le stabilité des données sont explicitement privées. La stabilité des données pour le fournisseur et le nom sont en cours d'évolution mais les

autres stabilités de données sont privées : il s'agit d'artefacts de l'implémentation actuelle. La classe de dépendance de FBT est ISA : lorsque FBT est disponible sur toutes les architectures de jeux d'instructions, il n'y a aucune garantie que FBT sera disponible sur de futures architectures de jeux d'instructions arbitraires.

CHAPITRE 21

Fournisseur syscall

Grâce au fournisseur syscall une sonde est disponible à l'activation et au retour de chaque appel système au sein du système. Comme les appels système sont l'interface principale entre les applications de niveau utilisateur et le noyau du système d'exploitation, le fournisseur syscall peut conférer une connaissance intuitive considérable sur le comportement de l'application relatif au système.

Sondes

syscall fournit une paire de sondes pour chaque appel système : une sonde entry qui se déclenche avant l'activation de l'appel système et une sonde return qui se déclenche après la désactivation de l'appel système mais avant le transfert du contrôle au niveau utilisateur. Quelle que soit la sonde syscall, le nom de la fonction est configuré de manière à correspondre au nom de l'appel système instrumenté et le nom du module est indéfini.

Le nom des appels système tel qu'indiqué par le fournisseur syscall figure normalement dans le fichier /etc/name_to_sysnum. En règle générale, les noms d'appel système fournis par syscall correspondent aux noms figurant dans la Section 2 des pages de manuel. Cependant, certaines sondes fournies par syscall ne correspondent pas directement à un appel système documenté. Cette section en présente les raisons classiques.

Anachronismes des appels système

Dans certains cas, le nom de l'appel système fourni par syscall reflète en fait un précédent détail d'implémentation. Par exemple, pour des motifs remontant aux prémices d'UNIX™, le nom de exit(2) dans /etc/name_to_sysnum est rexit. De même, le nom de time(2) est gtime et le nom de execle(2) et execve(2) est exece.

Appels système sous-codés

Certains appels système (voir Section 2 pour des exemples) sont implémentés sous la forme de sous-opérations d'un appel système non documenté. Par exemple, les appels système se rapportant aux sémaphores du System V (semctl(2), semget(2), semids(2), semop(2) et semtimedop(2)) sont implémentés en tant que sous-opérations d'un seul appel système, semsys. L'appel système semsys prend comme premier argument un *sous-code* spécifique à l'implémentation indiquant l'appel système spécifique requis : SEMCTL, SEMGET, SEMIDS, SEMOP ou SEMTIMEDOP, respectivement. En cas de surcharge d'un seul appel système pour implémenter plusieurs appels système, il n'y a qu'une seule paire de sondes syscall pour les sémaphores du System V : syscall::semsys:entry et syscall::semsys:return .

Appels système de grands fichiers

Un programme 32 bits qui prend en charge de *grands fichiers* dont la taille excède quatre giga-octets doit pouvoir traiter des décalages de fichiers 64–bits. Comme les grands fichiers impliquent de grands décalages, ils sont manipulés par le biais d'un ensemble parallèle d'interfaces système, comme indiqué dans lf64(5). Ces interfaces sont documentées dans lf64 mais aucune page de manuel individuelle n'est spécifiquement dédiée à leur présentation. Chacune de ces interfaces d'appel système de grands fichiers possède sa propre sonde syscall comme illustré dans le Tableau 21–1.

TABLEAU 21–1 syscall Sonde de grands fichiers

Sonde syscall de grands fichiers	Appel système
creat64	creat(2)
fstat64	fstat(2)
fstatvfs64	fstatvfs(2)
getdents64	getdents(2)
getrlimit64	getrlimit(2)
lstat64	lstat(2)
mmap64	mmap(2)
open64	open(2)
pread64	pread(2)
pwrite64	pwrite(2)
setrlimit64	setrlimit(2)

TABLEAU 21–1 `syscall` Sonde de grands fichiers (Suite)

Sonde `syscall` de grands fichiers	Appel système
`stat64`	`stat(2)`
`statvfs64`	`statvfs(2)`

Appels système privés

Certains appels système constituent des détails d'implémentation privée de sous-systèmes Solaris qui forment la limite du noyau utilisateur. C'est pourquoi, ces appels système ne font l'objet d'aucune page de manuel dans la Section 2. Cette catégorie comprend notamment l'appel système `signotify` qui entre dans le cadre de l'implémentation des files d'attente de messages de POSIX.4 et l'appel système `utssys` qui permet d'implémenter `fuser(1M)`.

Arguments

Pour les sondes `entry` , les arguments (`arg0` .. `arg`*n*) correspondent aux arguments de l'appel système. Pour les sondes `return`, `arg0` et `arg1` contiennent la valeur de retour. Une valeur non-zéro dans une variable en D `errno` indique un échec de l'appel système.

Stabilité

Le fournisseur `syscall` utilise le mécanisme de stabilité de DTrace pour décrire sa stabilité, comme illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	Commune
Module	Privé	Privé	Inconnu
Fonction	Instable	Instable	ISA
Nom	En cours d'évolution	En cours d'évolution	Commune
Arguments	Instable	Instable	ISA

CHAPITRE 22

Fournisseur sdt

Le fournisseur SDT (Statically Defined Tracing) crée des sondes sur les sites qu'un programmeur de logiciels a formellement désignés. Le mécanisme du fournisseur SDT permet aux programmeurs de choisir consciemment les emplacements convenant aux utilisateurs de DTrace et de transmettre certaines connaissances sémantiques sur chaque emplacement par le biais du nom de la sonde. Le noyau de Solaris a défini une poignée de sondes SDT et en ajoutera probablement d'autres à l'avenir. DTrace fournit également un mécanisme pour les développeurs d'applications utilisateur de manière à définir les sondes statiques, tel que spécifié dans le Chapitre34, “Suivi défini statiquement pour les applications utilisateur”.

Sondes

Les sondes SDT définies par le noyau de Solaris sont répertoriées dans le Tableau 22–1. La stabilité du nom et des données de ces sondes est Privée car sa description reflète ici l'implémentation du noyau et ne doit pas être influencée par une exécution de l'interface. Pour plus d'informations sur le mécanisme de stabilité de DTrace, reportez-vous à la section “Stabilité” à la page 243.

TABLEAU 22–1 Sondes SDT

Nom de la sonde	Description	arg0
callout-start	Sonde qui se déclenche immédiatement avant l'exécution d'une légende (voir <sys/callo.h>). Les légendes sont exécutées au moyen d'une horloge système périodique et représentent l'implémentation de timeout(9F).	Pointez la sonde callout_t (voir <sys/callo.h>) correspondant à la légende à exécuter.

TABLEAU 22–1 Sondes SDT *(Suite)*

Nom de la sonde	Description	arg0
callout-end	Sonde qui se déclenche immédiatement avant l'exécution d'une légende (voir <sys/callo.h>).	Pointez la sonde callout_t (voir <sys/callo.h>) correspondant à la légende qui vient juste d'être exécutée.
interrupt-start	Sonde qui se déclenche immédiatement avant d'appeler une routine d'interruption du périphérique.	Pointez la structure dev_info (voir <sys/ddi_impldefs.h>) correspondant au périphérique d'interruption.
interrupt-complete	Sonde qui se déclenche immédiatement après l'arrêt d'une routine d'interruption du périphérique.	Pointez la structure dev_info (voir <sys/ddi_impldefs.h>) correspondant au périphérique d'interruption.

Exemples

L'exemple suivant est un script d'observation du comportement des légendes seconde par seconde :

```
#pragma D option quiet

sdt:::callout-start
{
    @callouts[((callout_t *)arg0)->c_func] = count();
}

tick-1sec
{
    printa("%40a %10@d\n", @callouts);
    clear(@callouts);
}
```

Cet exemple permet de savoir qui utilise fréquemment timeout(9F) dans le système, comme illustré dans la sortie suivante :

```
# dtrace -s ./callout.d
                                     FUNC       COUNT
                             TS‘ts_update           1
                  uhci‘uhci_cmd_timeout_hdlr        3
                          genunix‘setrun            5
                      genunix‘schedpaging           5
                         ata‘ghd_timeout           10
  uhci‘uhci_handle_root_hub_status_change         309

                                     FUNC       COUNT
```

```
           ip‘tcp_time_wait_collector          1
                        TS‘ts_update          1
              uhci‘uhci_cmd_timeout_hdlr      3
                   genunix‘schedpaging        4
                       genunix‘setrun         8
                      ata‘ghd_timeout        10
 uhci‘uhci_handle_root_hub_status_change    300

                                   FUNC    COUNT
           ip‘tcp_time_wait_collector          0
                     iprb‘mii_portmon          1
                        TS‘ts_update           1
              uhci‘uhci_cmd_timeout_hdlr       3
                   genunix‘schedpaging         4
                       genunix‘setrun          7
                      ata‘ghd_timeout         10
 uhci‘uhci_handle_root_hub_status_change     300
```

L'interface `timeout(9F)` ne produit qu'une seule expiration d'horloge. Les utilisateurs de `timeout()` qui ont besoin d'une fonction d'horloge d'intervalle, appliquent généralement leur délai d'attente à partir de leur routine `timeout()`. L'exemple suivant présente ce comportement :

```
#pragma D option quiet

sdt:::callout-start
{
    self->callout = ((callout_t *)arg0)->c_func;
}

fbt::timeout:entry
/self->callout && arg2 <= 100/
{
    /*
     * In this case, we are most interested in interval timeout(9F)s that
     * are short.  We therefore do a linear quantization from 0 ticks to
     * 100 ticks.  The system clock’s frequency — set by the variable
     * "hz" — defaults to 100, so 100 system clock ticks is one second.
     */
    @callout[self->callout] = lquantize(arg2, 0, 100);
}

sdt:::callout-end
{
    self->callout = NULL;
}

END
```

```
{
    printa("%a\n%@d\n\n", @callout);
}
```

Exécuter ce script et patienter quelques secondes avant d'appuyer sur Control-C entraîne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./interval.d
^C
genunix`schedpaging

           value  ------------- Distribution ------------- count
              24 |                                         0
              25 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 20
              26 |                                         0


ata`ghd_timeout

           value  ------------- Distribution ------------- count
               9 |                                         0
              10 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 51
              11 |                                         0


uhci`uhci_handle_root_hub_status_change

           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1515
               2 |                                         0
```

La sortie montre que `uhci_handle_root_hub_status_change()` dans le pilote `uhci`(7D) représente l'horloge d'intervalle la plus courte du système : elle est appelée à chaque top d'horloge du système.

Vous pouvez utiliser la sonde `interrupt-start` pour comprendre l'activité d'interruption. L'exemple suivant illustre comment quantifier le temps accordé à l'exécution d'une routine d'interruption par le nom du pilote :

```
interrupt-start
{
    self->ts = vtimestamp;
}

interrupt-complete
/self->ts/
{
```

```
    this->devi = (struct dev_info *)arg0;
    @[stringof(‘devnamesp[this->devi->devi_major].dn_name),
        this->devi->devi_instance] = quantize(vtimestamp - self->ts);
}
```

Exécuter ce script engendre une sortie identique à l'exemple suivant :

```
# dtrace -s ./intr.d
dtrace: script './intr.d' matched 2 probes
^C
 isp                                                      0
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
           32768 |                                         0

  pcf8584                                                  0
           value  ------------- Distribution ------------- count
              64 |                                         0
             128 |                                         2
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         157
             512 |@@@@@@                                   31
            1024 |                                         3
            2048 |                                         0

  pcf8584                                                  1
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@          154
            8192 |@@@@@@@                                  37
           16384 |                                         2
           32768 |                                         0

  qlc                                                      0
           value  ------------- Distribution ------------- count
           16384 |                                         0
           32768 |@@                                       9
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      126
          131072 |@                                        5
          262144 |                                         2
          524288 |                                         0

  hme                                                      0
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |                                         6
            4096 |                                         2
            8192 |@@@@                                     89
```

```
           16384 |@@@@@@@@@@@@@                           262
           32768 |@                                       37
           65536 |@@@@@@@                                 139
          131072 |@@@@@@@@                                161
          262144 |@@@                                     73
          524288 |                                        4
         1048576 |                                        0
         2097152 |                                        1
         4194304 |                                        0

  ohci                                                     0
           value  ------------- Distribution ------------- count
            8192 |                                        0
           16384 |                                        3
           32768 |                                        1
           65536 |@@@                                     143
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    1368
          262144 |                                        0
```

Création de sondes SDT

En tant que développeur de pilotes de périphériques, vous souhaiterez peut-être créer vos propres sondes SDT dans votre pilote Solaris. Les sondes désactivées du fournisseur SDT se traduisent essentiellement par le coût de plusieurs instructions machine non opérationnelles. Vous êtes, par conséquent, encouragé à ajouter des sondes SDT à vos pilotes de périphérique, au besoin. À moins que ces sondes n'aient une incidence négative sur les performances, vous pouvez les conserver dans votre code de livraison.

Déclaration des sondes

Les sondes SDT sont déclarées à l'aide des macros `DTRACE_PROBE`, `DTRACE_PROBE1`, `DTRACE_PROBE2`, `DTRACE_PROBE3` et `DTRACE_PROBE4` à partir de `<sys/sdt.h>`. Le nom du module et le nom des fonctions d'une sonde basée sur le fournisseur SDT correspondent au module du noyau et aux fonctions de la sonde. Le nom de la sonde dépend du nom donné dans la macro `DTRACE_PROBE`*n*. Si le nom ne contient pas deux traits de soulignement consécutifs (__), le nom de la sonde est écrit dans la macro. Si le nom contient deux traits de soulignement consécutifs, ils sont convertis en un tiret simple (-) dans le nom de la sonde. Par exemple, si une macro `DTRACE_PROBE` spécifie `transaction__start`, la sonde SDT s'appellera `transaction-start`. Cette substitution permet au code C de fournir des noms de macro ne constituant pas des identificateurs en C valides sans spécifier de chaîne.

Comme DTrace intègre au tuple identifiant une sonde le nom du module du noyau et des fonctions, vous n'avez pas à ajouter ces informations dans le nom de la sonde pour éviter les collisions d'espace de noms. Vous pouvez utiliser la commande `dtrace -l -P sdt -m` *module*

sur votre pilote *module* pour répertorier les sondes que vous avez installées, ainsi que leurs noms complets, tels que les utilisateurs de DTrace pourront les afficher.

Arguments des sondes

Les arguments de chaque sonde SDT sont les arguments spécifiés dans la référence de macro `DTRACE_PROBE`*n* correspondante. Le nombre d'arguments dépend de la macro utilisée pour créer la sonde : `DTRACE_PROBE1` spécifie un argument, `DTRACE_PROBE2` spécifie deux arguments, etc. Lors de la déclaration de vos sondes SDT, vous pouvez réduire l'incidence des sondes désactivées en ne déréférençant pas les pointeurs et en ne chargeant pas les variables globales dans les arguments des sondes. Vous devez effectuer avec précaution le déréférencement des pointeurs et le chargement des variables dans les actions en D qui activent les sondes de sorte que les utilisateurs de DTrace puissent ne demander ces actions que lorsqu'ils en ont besoin.

Stabilité

Le fournisseur SDT utilise le mécanisme de stabilité de DTrace pour présenter sa stabilité, comme illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	Privé	Privé	ISA
Arguments	Privé	Privé	ISA

CHAPITRE 23

Fournisseur sysinfo

Le fournisseur sysinfo permet d'accéder aux sondes correspondant aux statistiques du noyau classées en fonction du nom sys. Grâce à ces statistiques fournies aux utilitaires de surveillance du système comme mpstat(1M), le fournisseur sysinfo permet d'étudier rapidement le comportement anormal observé.

Sondes

Le fournisseur sysinfo permet d'accéder aux sondes correspondant aux champs des statistiques de noyau nommées sys : une sonde fournie par sysinfo se déclenche juste avant que la valeur sys correspondante ne soit incrémentée. L'exemple suivant indique comment afficher les noms et les valeurs courantes des statistiques du noyau nommées sys à l'aide de la commande kstat(1M).

```
$ kstat -n sys
module: cpu                             instance: 0
name:   sys                             class:    misc
    bawrite                         123
    bread                           2899
    bwrite                          17995
...
```

Les sondes sysinfo sont décrites dans le Tableau 23–1.

TABLEAU 23–1 Sondes sysinfo

bawrite	Sonde qui se déclenche à chaque écriture asynchrone d'un tampon sur un périphérique.

TABLEAU 23–1 Sondes sysinfo *(Suite)*

bread	Sonde qui se déclenche à chaque lecture physique d'un tampon à partir d'un périphérique. bread se déclenche *après* la demande du tampon à partir du périphérique mais *avant* le blocage de son exécution.
bwrite	Sonde qui se déclenche à chaque écriture d'un tampon sur un périphérique, que ce soit de manière synchrone *ou* asynchrone.
idlethread	Sonde qui se déclenche chaque fois qu'une CPU entre dans la boucle inactive.
intrblk	Sonde qui se déclenche chaque fois qu'un thread d'interruption bloque.
inv_swtch	Sonde qui se déclenche chaque fois qu'un thread en cours d'exécution est contraint d'abandonner involontairement la CPU.
lread	Sonde qui se déclenche à chaque lecture logique d'un tampon à partir d'un périphérique.
lwrite	Sonde qui se déclenche à chaque écriture logique d'un tampon sur un périphérique.
modload	Sonde qui se déclenche à chaque chargement d'un module du noyau.
modunload	Sonde qui se déclenche à chaque déchargement d'un module du noyau.
msg	Sonde qui se déclenche chaque fois qu'un appel système msgsnd(2) ou msgrcv(2) est passé mais avant l'exécution des opérations de file d'attente du message.
mutex_adenters	Sonde qui se déclenche à chaque tentative d'acquisition d'un verrou adaptatif propriétaire. Si cette sonde se déclenche, l'une des sondes adaptive-block ou adaptive-spin du fournisseur lockstat se déclenche également. Pour plus d'informations, reportez-vous au Chapitre18, “Fournisseur lockstat”.
namei	Sonde qui se déclenche à chaque tentative de recherche de nom dans le système de fichiers.
nthreads	Sonde qui se déclenche à chaque création d'un thread.
phread	Sonde qui se déclenche chaque fois qu'une lecture d'E/S brute va être exécutée.
phwrite	Sonde qui se déclenche chaque fois qu'une écriture d'E/S brute va être exécutée.
procovf	Sonde qui se déclenche chaque fois qu'il est impossible de créer un nouveau processus car le système manque d'entrées de tableau de processus.
pswitch	Sonde qui se déclenche chaque fois qu'une CPU passe de l'exécution d'un thread à l'exécution d'un autre thread.
readch	Sonde qui se déclenche après chaque lecture réussie mais avant que le contrôle soit retourné au thread exécutant la lecture. Une lecture est possible par l'intermédiaire des appels système read(2), readv(2) ou pread(2). arg0 contient le nombre d'octets qui ont été correctement lus.

TABLEAU 23–1 Sondes sysinfo *(Suite)*

rw_rdfails	Sonde qui se déclenche à chaque exécution d'une tentative de verrouillage de la lecture sur des lecteurs/un graveur lorsque le verrou est détenu ou demandé par un graveur. Si cette sonde se déclenche, la sonde rw-block du fournisseur lockstat se déclenche également. Pour plus d'informations, reportez-vous au Chapitre18, "Fournisseur lockstat".
rw_wrfails	Sonde qui se déclenche à chaque tentative de verrouillage de l'écriture d'un verrou de lecteurs/graveur lorsque le verrou est détenu par plusieurs lecteurs ou par un autre graveur. Si cette sonde se déclenche, la sonde rw-block du fournisseur lockstat se déclenche également. Pour plus d'informations, reportez-vous au Chapitre18, "Fournisseur lockstat".
sema	Sonde qui se déclenche à chaque appel système semop(2) mais avant l'exécution d'opérations de sémaphore.
sysexec	Sonde qui se déclenche à chaque appel système exec(2).
sysfork	Sonde qui se déclenche à chaque appel système fork(2).
sysread	Sonde qui se déclenche à chaque appel système read(2), readv(2) ou pread(2).
sysvfork	Sonde qui se déclenche à chaque appel système vfork(2).
syswrite	Sonde qui se déclenche à chaque appel système write(2), writev(2) ou pwrite(2).
trap	Sonde qui se déclenche à chaque déroutement du processeur. Notez que certains processeurs, notamment de la gamme UltraSPARC, gèrent certains déroutements légers par l'intermédiaire d'un mécanisme qui n'entraîne pas le déclenchement de cette sonde.
ufsdirblk	Sonde qui se déclenche à chaque lecture d'un bloc de répertoires depuis le système de fichiers UFS. Pour plus de détails sur l'UFS, reportez-vous à ufs(7FS).
ufsiget	Sonde qui se déclenche à chaque récupération d'un inode. Pour plus de détails sur l'UFS, reportez-vous à ufs(7FS).
ufsinopage	Sonde qui se déclenche après la mise à disposition pour réutilisation d'un inode interne *sans* page de données liée. Pour plus de détails sur l'UFS, reportez-vous à ufs(7FS).
ufsipage	Sonde qui se déclenche après la mise à disposition pour réutilisation d'un inode interne *avec* pages de données liées. Cette sonde se déclenche après vidage des pages de données liées sur le disque. Pour plus de détails sur l'UFS, reportez-vous à ufs(7FS).
writech	Sonde qui se déclenche après chaque écriture réussie mais avant que le contrôle soit retourné au thread exécutant l'écriture. Une écriture est possible par l'intermédiaire des appels système write(2), writev(2) ou pwrite(2). arg0 contient le nombre d'octets qui ont été correctement écrits.

TABLEAU 23–1 Sondes sysinfo (Suite)

xcalls	Sonde qui se déclenche à chaque appel croisé sur le point d'être passé. Un appel croisé est un mécanisme du système d'exploitation permettant à une CPU de demander un travail immédiat à une autre CPU.

Arguments

Les arguments vers les sondes sysinfo se présentent comme suit :

arg0	Valeur de laquelle les statistiques sont incrémentées. Cet argument est toujours égal à 1 pour la plupart des sondes. Il peut, toutefois, prendre une autre valeur pour certaines sondes.
arg1	Pointeur vers la valeur courante des statistiques à incrémenter. Cette valeur, d'une quantité de 64–bits, sera incrémentée de la valeur de arg0. Déréférencer ce pointeur permet aux utilisateurs de déterminer la valeur courante des statistiques correspondant à la sonde.
arg2	Pointeur vers la structure cpu_tqui correspond à la CPU sur laquelle les statistiques doivent être incrémentées. Cette structure est définie dans <sys/cpuvar.h>, mais elle fait partie intégrante de l'implémentation du noyau et doit être considérée comme Privée.

La valeur de arg0 est de 1 pour la plupart des sondes sysinfo. Cependant, les sondes readch et writech définissent arg0 sur le nombre d'octets lus ou écrits, respectivement. Cette fonctionnalité vous permet de déterminer la taille des lectures par nom exécutable, comme illustré dans l'exemple ci-dessous :

```
# dtrace -n readch'{@[execname] = quantize(arg0)}'
dtrace: description 'readch' matched 4 probes
^C
  xclock
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
              64 |                                         0

  acroread
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
              64 |                                         0

  FvwmAuto
           value  ------------- Distribution ------------- count
```

```
               2 |                                         0
               4 |@@@@@@@@@@@@@                            13
               8 |@@@@@@@@@@@@@@@@@@@@@                    21
              16 |@@@@@                                    5
              32 |                                         0

      xterm
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@                 19
              64 |@@@@@@@@@                                7
             128 |@@@@@@                                   5
             256 |                                         0

      fvwm2
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@                                186
               1 |                                         0
               2 |                                         0
               4 |@@                                       51
               8 |                                         17
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@               503
              64 |                                         9
             128 |                                         0

      Xsun
           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@                              269
               1 |                                         0
               2 |                                         0
               4 |                                         2
               8 |@                                        31
              16 |@@@@@                                    128
              32 |@@@@@@@                                  171
              64 |@                                        33
             128 |@@@                                      85
             256 |@                                        24
             512 |                                         8
            1024 |                                         21
            2048 |@                                        26
            4096 |                                         21
            8192 |@@@@                                     94
           16384 |                                         0
```

Le fournisseur `sysinfo` définit `arg2` en tant que pointeur vers une structure `cpu_t` (une structure interne vers l'implémentation du noyau). Les sondes `sysinfo` se déclenchent sur la CPU sur laquelle la statistique est incrémentée. Utilisez le membre `cpu_id` de la structure `cpu_t` pour déterminer la CPU qui vous intéresse.

Exemple

Examinez la sortie de `mpstat(1M)` suivante :

```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 12   90  22 5760   422  299  435   26   71  116   11  1372    5  19  17  60
 13   46  18 4585   193  162  431   25   69  117   12  1039    3  17  14  66
 14   33  13 3186   405  381  397   21   58  105   10   770    2  17  11  70
 15   34  19 4769   109   78  417   23   57  115   13   962    3  14  14  69
 16   74  16 4421   437  406  448   29   77  111    8  1020    4  23  14  59
 17   51  15 4493   139  110  378   23   62  109    9   928    4  18  14  65
 18   41  14 4204   494  468  360   23   56  102    9   849    4  17  12  68
 19   37  14 4229   115   87  363   22   50  106   10   845    3  15  14  67
 20   78  17 5170   200  169  456   26   69  108    9  1119    5  21  25  49
 21   53  16 4817    78   51  394   22   56  106    9   978    4  17  22  57
 22   32  13 3474   486  463  347   22   48  106    9   769    3  17  17  63
 23   43  15 4572    59   34  361   21   46  102   10   947    4  15  22  59
```

À partir de la sortie ci-dessus, vous pouvez conclure que le champ `xcal` semble trop grand, notamment du fait de la relative inactivité du système. `mpstat` détermine la valeur du champ `xcal` en examinant le champ `xcalls` des statistiques du noyau `sys`. Il est, par conséquent, facile d'étudier cette aberration en activant la sonde `xcalls` `sysinfo`, comme illustré dans l'exemple suivant :

```
# dtrace -n xcalls'{@[execname] = count()}'
dtrace: description 'xcalls' matched 4 probes
^C
  dtterm                                                            1
  nsrd                                                              1
  in.mpathd                                                         2
  top                                                               3
  lockd                                                             4
  java_vm                                                          10
  ksh                                                              19
  iCald.pl6+RPATH                                                  28
  nwadmin                                                          30
  fsflush                                                          34
  nsrindexd                                                        45
  in.rlogind                                                       56
  in.routed                                                       100
  dtrace                                                          153
```

```
  rpc.rstatd                                                     246
  imapd                                                          377
  sched                                                          431
  nfsd                                                          1227
  find                                                          3767
```

La sortie montre où chercher la source des appels croisés. Un certain nombre de processus `find(1)` provoquent la majorité des appels croisés. Le script en D suivant peut être utilisé pour comprendre le problème plus en détails :

```
syscall:::entry
/execname == "find"/
{
    self->syscall = probefunc;
    self->insys = 1;
}

sysinfo:::xcalls
/execname == "find"/
{
    @[self->insys ? self->syscall : "<none>"] = count();
}

syscall:::return
/self->insys/
{
    self->insys = 0;
    self->syscall = NULL;
}
```

Ce script utilise le fournisseur `syscall` pour attribuer des appels croisés à un appel système particulier à partir de `find.` Certains appels croisés, dont ceux induits par les défauts de page, peuvent ne pas émaner des appels système. Le cas échéant, le script imprime “<none>”. Exécuter le script engendre une sortie similaire à l'exemple suivant :

```
# dtrace -s ./find.d
 dtrace: script './find.d' matched 444 probes
^C
  <none>                                                           2
  lstat64                                                       2433
  getdents64                                                   14873
```

Cette sortie indique que la majorité des appels croisés induits par `find` sont en retour induits par les appels système `getdents(2)`. Une étude plus approfondie dépend de ce que vous souhaitez étudier. Si vous souhaitez comprendre pourquoi les processus `find` appellent `getdents`, vous pouvez écrire en langage D un script d'agrégation sur `ustack()` lorsque `find` induit un appel croisé. Si vous souhaitez comprendre pourquoi les appels vers `getdents` induisent des appels croisés, vous pouvez écrire en langage D un script d'agrégation sur

`stack()` lorsque `find` induit un appel croisé. Quelle que soit l'étape suivante, la présence de la sonde `xcalls` vous a permis de découvrir rapidement la cause première d'une sortie de contrôle inhabituelle.

Stabilité

Le fournisseur `sysinfo` utilise un mécanisme de stabilité pour présenter sa stabilité, comme illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	Privé	Privé	ISA

◆ ◆ ◆ CHAPITRE 24

Fournisseur vminfo

Le fournisseur vminfo permet d'accéder aux sondes qui correspondent aux statistiques de noyau vm. Ces statistiques permettant d'accéder aux utilitaires de surveillance du système comme vmstat(1M), le fournisseur vminfo permet d'étudier rapidement le comportement anormal observé.

Sondes

Le fournisseur vminfo permet d'accéder aux sondes disponibles qui correspondent aux champs de statistiques de noyau nommées vm : une sonde fournie par vminfo se déclenche juste avant que la valeur vm correspondante ne soit incrémentée. Pour afficher les noms et les valeurs actuelles des statistiques de noyau nommées vm, utilisez la commande kstat(1M), comme illustré dans l'exemple suivant :

```
$ kstat -n vm
module: cpu                             instance: 0
name:   vm                              class:    misc
        anonfree                        13
        anonpgin                        2620
        anonpgout                       13
        as_fault                        12528831
        cow_fault                       2278711
        crtime                          202.10625712
        dfree                           1328740
        execfree                        0
        execpgin                        5541
        ...
```

Les sondes vminfo sont décrites dans le Tableau 24–1.

TABLEAU 24–1 Sondes vminfo

anonfree	Sonde qui se déclenche chaque fois qu'une page anonyme non modifiée est libérée dans le cadre de l'activité de pagination. Les pages anonymes correspondent aux pages qui ne sont pas associées à un fichier. Ces pages sont contenues dans la mémoire du tas, la mémoire de la pile ou la mémoire obtenue en mappant explicitement zero(7D).
anonpgin	Sonde qui se déclenche chaque fois qu'une page anonyme est chargée à partir d'un périphérique de swap.
anonpgout	Sonde qui se déclenche chaque fois qu'une page anonyme modifiée est renvoyée vers un périphérique de swap.
as_fault	Sonde qui se déclenche chaque fois qu'une erreur se produit sur une page, dès l'instant qu'il ne s'agit pas d'une erreur de protection ni d'une défaillance de copie-écriture.
cow_fault	Sonde qui se déclenche chaque fois qu'une défaillance de copie-écriture se produit sur une page. arg0 contient le nombre de pages créées suite à la copie-écriture.
dfree	Sonde qui se déclenche chaque fois qu'une page est libérée suite à l'activité de pagination. Chaque fois que dfree se déclenche, une seule sonde anonfree, execfree ou fsfree se déclenche ultérieurement.
execfree	Sonde qui se déclenche chaque fois qu'une page exécutable non modifiée est libérée dans le cadre de l'activité de pagination.
execpgin	Sonde qui se déclenche chaque fois qu'une page exécutable est chargée à partir de la sauvegarde de secours.
execpgout	Sonde qui se déclenche chaque fois qu'une page exécutable modifiée est renvoyée vers la sauvegarde de secours. La pagination de la plupart des pages exécutables se produit en fonction de la sonde execfree. La sonde execpgout ne peut se déclencher que si une page exécutable est modifiée en mémoire, ce qui est rare dans la plupart des systèmes.
fsfree	Sonde qui se déclenche chaque fois qu'une page non modifiée de données système de fichiers est libérée dans le cadre de l'activité de pagination.
fspgin	Sonde qui se déclenche chaque fois qu'une page de système de fichiers est chargée à partir de la sauvegarde de secours.
fspgout	Sonde qui se déclenche chaque fois qu'une page de système de fichiers modifiée est renvoyée vers la sauvegarde de secours.
kernel_asflt	Sonde qui se déclenche chaque fois qu'une erreur de page est détectée par le noyau sur une page de son propre espace d'adressage. Chaque fois que la sonde kernel_asflt se déclenche, elle est immédiatement précédée par le déclenchement de la sonde as_fault.
maj_fault	Sonde qui se déclenche chaque fois qu'une erreur de page se produit suite à une E/S à partir d'une sauvegarde de secours ou d'un périphérique de swap. Chaque fois que la sonde maj_fault se déclenche, elle est immédiatement précédée par le déclenchement de la sonde pgin.

TABLEAU 24–1 Sondes `vminfo` *(Suite)*

`pgfrec`	Sonde qui se déclenche chaque fois qu'une page est retirée d'une liste de pages disponibles.
`pgin`	Sonde qui se déclenche chaque fois qu'une page est chargée à partir d'une sauvegarde de secours ou d'un périphérique de swap. Cette sonde diffère de la sonde `maj_fault` en ce que `maj_fault` ne se déclenche que lorsqu'une page est chargée suite à une erreur de page. La sonde `pgin` se déclenche chaque fois qu'une page est chargée, quelle qu'en soit la raison.
`pgout`	Sonde qui se déclenche chaque fois qu'une page est renvoyée vers une sauvegarde de secours ou un périphérique de swap.
`pgpgin`	Sonde qui se déclenche chaque fois qu'une page est chargée à partir d'une sauvegarde de secours ou d'un périphérique de swap. Les sondes `pgpgin` et `pgin` se différencient uniquement en ce que `pgpgin` contient le nombre de pages chargées en tant que `arg0`. `arg0` est toujours égal à 1 avec la sonde `pgin`.
`pgpgout`	Sonde qui se déclenche chaque fois qu'une page est renvoyée vers une sauvegarde de secours ou un périphérique de swap. Les sondes `pgpgout` et `pgout` se différencient uniquement en ce que `pgpgout` contient le nombre de pages chargées en tant que `arg0`. `arg0` est toujours égal à 1 avec la sonde `pgout`.
`pgrec`	Sonde qui se déclenche à chaque récupération d'une page.
`pgrrun`	Sonde qui se déclenche chaque fois que le pager est programmé.
`pgswapin`	Sonde qui se déclenche chaque fois que des pages d'un processus déchargé sont chargées en mémoire. Le nombre de pages chargées en mémoire est indiqué dans `arg0`.
`pgswapout`	Sonde qui se déclenche chaque fois que des pages sont déchargées dans le cadre du déchargement d'un processus. Le nombre de pages déchargées de la mémoire est indiqué dans `arg0`.
`prot_fault`	Sonde qui se déclenche chaque fois qu'une erreur de page se produit à cause d'une violation de la protection.
`rev`	Sonde qui se déclenche chaque fois que la page Daemon entame une nouvelle révolution de toutes les pages.
`scan`	Sonde qui se déclenche chaque fois que la page Daemon examine une page.
`softlock`	Sonde qui se déclenche chaque fois qu'une page est défaillante dans le cadre du placement d'un verrou logiciel sur la page.
`swapin`	Sonde qui se déclenche chaque fois qu'un processus déchargé est rechargé.
`swapout`	Sonde qui se déclenche chaque fois qu'un processus est déchargé.
`zfod`	Sonde qui se déclenche chaque fois qu'une page remplie de zéros est créée à la demande.

Arguments

`arg0`	Valeur de laquelle les statistiques sont incrémentées. Cet argument est toujours de 1 pour la plupart des sondes, mais il peut également prendre une autre valeur avec les sondes présentées dans le Tableau 24–1.
`arg1`	Pointeur vers la valeur courante des statistiques à incrémenter. Cette valeur, d'une quantité de 64–bits, sera incrémentée de la valeur de `arg0`. Déréférencer ce pointeur permet aux utilisateurs de déterminer la valeur courante des statistiques correspondant à la sonde.

Exemple

Examinez la sortie suivante de `vmstat(1M)` :

```
kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd s0 — —   in   sy   cs us sy id
 0 1 0 1341844 836720 26 311 1644 0 0 0  0 216 0  0  0  797  817  697  9 10 81
 0 1 0 1341344 835300 238 934 1576 0 0 0 0 194 0  0  0  750 2795  791  7 14 79
 0 1 0 1340764 833668 24 165 1149 0 0 0  0 133 0  0  0  637  813  547  5  4 91
 0 1 0 1340420 833024 24 394 1002 0 0 0  0 130 0  0  0  621 2284  653 14  7 79
 0 1 0 1340068 831520 14 202 380 0 0  0  0 59  0  0  0  482 5688 1434 25  7 68
```

La colonne `pi` de la sortie ci-dessus indique le nombre de pages chargées. Le fournisseur `vminfo` vous permet d'obtenir davantage d'informations sur la source de ces chargements de page, comme illustré dans l'exemple suivant :

```
dtrace -n pgin'{@[execname] = count()}'
dtrace: description 'pgin' matched 1 probe
^C
  xterm                                                             1
  ksh                                                               1
  ls                                                                2
  lpstat                                                            7
  sh                                                               17
  soffice                                                          39
  javaldx                                                         103
  soffice.bin                                                    3065
```

La sortie indique qu'un processus associé au logiciel StarOffice™, `soffice.bin`, est responsable de la plupart des chargements de page. Pour vous faire une meilleure idée du comportement de `soffice.bin` dans la mémoire virtuelle, vous pouvez activer toutes les sondes `vminfo`. L'exemple suivant exécute `dtrace(1M)` tout en lançant le logiciel StarOffice :

```
dtrace -P vminfo'/execname == "soffice.bin"/{@[probename] = count()}'
dtrace: description 'vminfo' matched 42 probes
```

```
^C

  kernel_asflt                                                      1
  fspgin                                                           10
  pgout                                                            16
  execfree                                                         16
  execpgout                                                        16
  fsfree                                                           16
  fspgout                                                          16
  anonfree                                                         16
  anonpgout                                                        16
  pgpgout                                                          16
  dfree                                                            16
  execpgin                                                         80
  prot_fault                                                       85
  maj_fault                                                        88
  pgin                                                             90
  pgpgin                                                           90
  cow_fault                                                       859
  zfod                                                           1619
  pgfrec                                                         8811
  pgrec                                                          8827
  as_fault                                                       9495
```

L'exemple de script suivant fournit davantage d'informations sur le comportement en mémoire virtuelle du logiciel StarOffice pendant le démarrage :

```
vminfo:::maj_fault,
vminfo:::zfod,
vminfo:::as_fault
/execname == "soffice.bin" && start == 0/
{
    /*
     * This is the first time that a vminfo probe has been hit; record
     * our initial timestamp.
     */
    start = timestamp;
}

vminfo:::maj_fault,
vminfo:::zfod,
vminfo:::as_fault
/execname == "soffice.bin"/
{
    /*
     * Aggregate on the probename, and lquantize() the number of seconds
     * since our initial timestamp.  (There are 1,000,000,000 nanoseconds
     * in a second.)  We assume that the script will be terminated before
```

```
     * 60 seconds elapses.
     */
    @[probename] =
        lquantize((timestamp - start) / 1000000000, 0, 60);
}
```

Exécutez le script tout en lançant à nouveau le logiciel StarOffice. Créez ensuite un nouveau dessin, puis une nouvelle présentation, fermez tous les fichiers et quittez l'application. Appuyez sur Control-C dans le shell dans lequel le script D est exécuté. Les résultats fournissent un aperçu du comportement en mémoire virtuelle dans le temps :

```
# dtrace -s ./soffice.d
dtrace: script './soffice.d' matched 10 probes
^C

 maj_fault
          value  ------------- Distribution ------------- count
              7 |                                         0
              8 |@@@@@@@@@                                88
              9 |@@@@@@@@@@@@@@@@@@@@                     194
             10 |@                                        18
             11 |                                         0
             12 |                                         0
             13 |                                         2
             14 |                                         0
             15 |                                         1
             16 |@@@@@@@@                                 82
             17 |                                         0
             18 |                                         0
             19 |                                         2
             20 |                                         0

  zfod
          value  ------------- Distribution ------------- count
            < 0 |                                         0
              0 |@@@@@@@                                  525
              1 |@@@@@@@@                                 605
              2 |@@                                       208
              3 |@@@                                      280
              4 |                                         4
              5 |                                         0
              6 |                                         0
              7 |                                         0
              8 |                                         44
              9 |@@                                       161
             10 |                                         2
             11 |                                         0
             12 |                                         0
```

```
                 13 |                                         4
                 14 |                                         0
                 15 |                                         29
                 16 |@@@@@@@@@@@@@@                           1048
                 17 |                                         24
                 18 |                                         0
                 19 |                                         0
                 20 |                                         1
                 21 |                                         0
                 22 |                                         3
                 23 |                                         0

  as_fault
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@                            4139
               1 |@@@@@@@                                  2249
               2 |@@@@@@@                                  2402
               3 |@                                        594
               4 |                                         56
               5 |                                         0
               6 |                                         0
               7 |                                         0
               8 |                                         189
               9 |@@                                       929
              10 |                                         39
              11 |                                         0
              12 |                                         0
              13 |                                         6
              14 |                                         0
              15 |                                         297
              16 |@@@@                                     1349
              17 |                                         24
              18 |                                         0
              19 |                                         21
              20 |                                         1
              21 |                                         0
              22 |                                         92
              23 |                                         0
```

La sortie montre un comportement de StarOffice en fonction du système de mémoire virtuelle. Par exemple, la sonde `maj_fault` ne s'est pas déclenchée avant le démarrage d'une nouvelle instance de l'application. Comme vous l'espériez, un “démarrage à chaud” de StarOffice n'a pas engendré de nouvelles défaillances majeures. La sortie de `as_fault` indique une hausse d'activité initiale, une latence pendant la localisation du menu par l'utilisateur pour créer un nouveau dessin, une autre période d'inactivité et un dernier pic d'activité lorsque l'utilisateur a

cliqué sur une nouvelle présentation. La sortie de `zfod` indique que la création de la nouvelle présentation s'est traduite, pendant une courte période, par une importante pression pour les pages remplies de zéros.

La nouvelle itération de la recherche de DTrace dans cet exemple va dépendre du sens que vous souhaitez donner à votre exploration. Si vous souhaitez comprendre la source de la demande de pages remplies de zéros, vous pouvez regrouper `ustack()` avec l'activation de `zfod.` Vous souhaiterez peut-être établir un seuil pour les pages remplies de zéros et utiliser une action `stop()` destructrice pour interrompre le processus fautif lorsque le seuil est franchi. Cette approche doit vous permettre d'utiliser des outils de débogage plus conventionnels comme `truss(1)` ou `mdb(1)`. Le fournisseur `vminfo` vous permet d'associer les statistiques résultant des outils conventionnels comme `vmstat(1M)` aux applications provoquant le comportement du système.

Stabilité

Le fournisseur `vminfo` utilise le mécanisme de stabilité de DTrace pour décrire sa stabilité, comme illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	Privé	Privé	ISA

CHAPITRE 25

Fournisseur proc

Le fournisseur proc propose des sondes relevant des activités suivantes : création et fin de processus, création et fin de LWP, exécution d'images de nouveau programme et envoi et gestion de signaux.

Sondes

Les sondes proc sont décrites dans le Tableau 25–1.

TABLEAU 25–1 Sondes proc

Sonder	Description
create	Sonde se déclenchant lors de la création d'un processus à l'aide de fork(2), forkall(2), fork1(2) ou vfork(2). Le psinfo_t correspondant au nouveau processus enfant est indiqué par args[0]. Vous pouvez différencier vfork des autres variantes fork en vérifiant PR_VFORKP dans l'élément pr_flag du thread de fork lwpsinfo_t. Vous pouvez différencier fork1 de forkall en étudiant les membres pr_nlwp de psinfo_t du processus parent (curpsinfo) et psinfo_t du processus enfant (args[0]). La sonde create ne se déclenchant qu'une fois le processus créé et étant donné que la création LWP fait partie de la création d'un processus, lwp-create ne se déclenchera que pour un ou des LWP créés au moment de la création d'un processus *avant* que la sonde create ne se déclenche pour le nouveau processus.
exec	Sonde se déclenchant si un processus charge une image de nouveau processus avec une variante de l'appel système exec(2) : exec(2), execle(2), execlp(2), execv(2), execve(2), execvp(2). La sonde exec se déclenche *avant* le chargement de l'image de processus. Des variables de processus comme execname et curpsinfo contiennent donc l'état du processus avant le chargement de l'image. Suite au déclenchement de la sonde exec, la sonde exec-failure ou exec-success se déclenche dans le même thread. Le chemin d'accès à l'image du nouveau processus est indiqué par args[0].

TABLEAU 25–1 Sondes proc *(Suite)*

Sonder	Description
exec-failure	Sonde se déclenchant lors de l'échec d'une variante exec(2). La sonde exec-failure ne se déclenche qu'après le déclenchement de la sonde exec dans le même thread. La valeur errno(3C) est indiquée dans args[0].
exec-success	Sonde se déclenchant lors de la réussite d'une variante exec(2). Comme la sonde exec-failure, exec-success ne se déclenche qu'après le déclenchement de la sonde exec dans le même thread. Au moment du déclenchement de la sonde exec-success, des variables de processus comme execname et curpsinfo contiennent l'état du processus après le chargement de l'image du nouveau processus.
exit	Sonde se déclenchant lors de la sortie du processus en cours. La raison, qui est exprimée par l'un des codes SIGCHLD siginfo.h(3HEAD), est contenue dans args[0].
fault	Sonde se déclenchant lorsqu'un thread rencontre une panne machine. Le code de la panne (tel que défini dans proc(4)) se trouve dans args[0]. La structure siginfo correspondant à la panne est indiquée par args[1]. Seules les pannes accompagnées d'un signal peuvent déclencher la sonde fault.
lwp-create	Sonde se déclenchant lors de la création d'un LWP, résultant généralement de thr_create(3C). Le lwpsinfo_t correspondant au nouveau thread est indiqué par args[0]. psinfo_t du processus contenant le thread est indiqué par args[1].
lwp-start	Sonde se déclenchant dans le cadre d'un nouveau LWP. La sonde lwp-start se déclenche avant l'exécution d'une quelconque instruction au niveau utilisateur. Si le LWP est le premier du processus, la sonde start se déclenche, suivie de lwp-start.
lwp-exit	Sonde se déclenchant lors de la sortie d'un LWP, en raison d'un signal ou d'un appel explicite de thr_exit(3C).
signal-discard	Sonde se déclenchant lorsqu'un signal est envoyé à un processus à un seul thread. Le signal n'est pas bloqué et est ignoré par le processus. Dans ces conditions, le signal est ignoré à la génération. Les lwpsinfo_t et psinfo_t du processus cible et le thread se trouvent dans args[0] et args[1], respectivement. Le numéro de signal est indiqué dans args[2].
signal-send	Sonde se déclenchant lorsqu'un signal est envoyé à un thread ou un processus. La sonde signal-send se déclenche en cas d'envoi du processus ou du thread. Les lwpsinfo_t et psinfo_t du processus et du thread de réception se trouvent dans args[0] et args[1], respectivement. Le numéro de signal est indiqué dans args[2]. signal-send est toujours suivi de signal-handle ou de signal-clear dans le processus et le thread de réception.

TABLEAU 25–1 Sondes proc (Suite)

Sonder	Description
signal-handle	Sonde se déclenchant immédiatement après la gestion d'un signal par un thread. La sonde signal-handle se déclenche en cas de gestion du signal par le thread. Le numéro de signal est indiqué dans args[0]. Un pointeur vers la structure siginfo_t correspondant au signal est indiqué dans args[1]. La valeur de args[1] si aucune structure siginfo_t n'est présente ou si l'indicateur SA_SIGINFO du gestionnaire de signaux n'est pas défini. L'adresse du gestionnaire de signaux du processus est indiquée dans args[2].
signal-clear	Sonde se déclenchant lorsqu'un signal en attente est effacé car le thread cible attendait le signal dans sigwait(2), sigwaitinfo(3RT) ou sigtimedwait(3RT). Dans ces conditions, le signal en attente est effacé et le numéro de signal est renvoyé à l'appelant. Le numéro de signal est indiqué dans args[0]. signal-clear se déclenche dans le cadre du thread initial en attente.
start	Sonde se déclenchant dans le cadre d'un nouveau processus. La sonde start se déclenche avant l'exécution d'une quelconque instruction au niveau utilisateur dans le processus.

Arguments

Les types d'argument des sondes proc sont répertoriés dans le Tableau 25–2. Les arguments sont décrits dans le Tableau 25–1.

TABLEAU 25–2 Arguments de sonde proc

Sonder	args[0]	args[1]	args[2]
create	psinfo_t *	—	—
exec	char *	—	—
exec-failure	int	—	—
exit	int	—	—
fault	int	siginfo_t *	—
lwp-create	lwpsinfo_t *	psinfo_t *	—
lwp-start	—	—	—
lwp-exit	—	—	—
signal-discard	lwpsinfo_t *	psinfo_t *	int
signal-discard	lwpsinfo_t *	psinfo_t *	int

TABLEAU 25–2 Arguments de sonde `proc` *(Suite)*

Sonder	args[0]	args[1]	args[2]
signal-send	lwpsinfo_t *	psinfo_t *	int
signal-handle	int	siginfo_t *	void (*)(void)
signal-clear	int	—	—
start	—	—	—

lwpsinfo_t

Plusieurs sondes `proc` comprennent des arguments du type `lwpsinfo_t`, structure abordée dans `proc`(4). La définition de la structure `lwpsinfo_t`, telle qu'accessible aux clients DTrace, est la suivante :

```
typedef struct lwpsinfo {
    int pr_flag;              /* flags; see below */
    id_t pr_lwpid;            /* LWP id */
    uintptr_t pr_addr;        /* internal address of thread */
    uintptr_t pr_wchan;       /* wait addr for sleeping thread */
    char pr_stype;            /* synchronization event type */
    char pr_state;            /* numeric thread state */
    char pr_sname;            /* printable character for pr_state */
    char pr_nice;             /* nice for cpu usage */
    short pr_syscall;         /* system call number (if in syscall) */
    int pr_pri;               /* priority, high value = high priority */
    char pr_clname[PRCLSZ];   /* scheduling class name */
    processorid_t pr_onpro;   /* processor which last ran this thread */
    processorid_t pr_bindpro; /* processor to which thread is bound */
    psetid_t pr_bindpset;     /* processor set to which thread is bound */
} lwpsinfo_t;
```

Le champ `pr_flag` est un masque contenant des indicateurs décrivant le processus. Ces indicateurs et leur signification sont décrits dans le Tableau 25–3.

TABLEAU 25–3 Valeurs de `pr_flag`

PR_ISSYS	Le processus est un processus système.
PR_VFORKP	Le processus est le parent d'un enfant `vfork`(2).
PR_FORK	Le processus est défini en mode d'héritage fork.
PR_RLC	Le processus est défini en mode d'exécution à la dernière fermeture.
PR_KLC	Le processus est défini en mode de destruction à la dernière fermeture.

TABLEAU 25–3 Valeurs de pr_flag (Suite)

PR_ASYNC	Le processus est défini en mode d'arrêt asynchrone.
PR_MSACCT	Le processus est activé en gestion microscopique.
PR_MSFORK	La gestion microscopique du processus est héritée de fork.
PR_BPTADJ	Le processus est défini en mode d'ajustement du point d'arrêt.
PR_PTRACE	Le processus est défini en mode de compatibilité ptrace(3C).
PR_STOPPED	Le thread est un LWP arrêté.
PR_ISTOP	Le thread est un LWP arrêté en cas d'événement intéressant.
PR_DSTOP	Le thread est un LWP disposant d'une directive d'arrêt en vigueur.
PR_STEP	Le thread est un LWP disposant d'une directive d'étape unique en vigueur.
PR_ASLEEP	Le thread est un LWP en sommeil interruptible dans un appel système.
PR_DETACH	Le thread est un LWP détaché. Reportez-vous aux pages Web de pthread_create(3C) et pthread_join(3C).
PR_DAEMON	Le thread est un LWP démon. Reportez-vous à la page Web de pthread_create(3C).
PR_AGENT	Le thread est le LWP agent du processus.
PR_IDLE	Le thread est un thread inactif pour une CPU. Les threads inactifs ne sont exécutés sur une CPU que lorsque les files d'attente d'exécution de la CPU sont vides.

Le champ pr_addr indique l'adresse d'une structure de données privée et du noyau représentant le thread. Alors que la structure de données est privée, le champ pr_addr peut être utilisé en tant que jeton unique pour déterminer la durée de vie d'un thread.

Le champ pr_wchan est défini lorsque le thread est en sommeil sur un objet de synchronisation. La signification du champ pr_wchan est spécifique à la mise en œuvre du noyau, mais le champ peut être utilisé en tant que jeton unique pour l'objet de synchronisation.

Le champ pr_stype est défini lorsque le thread est en sommeil sur un objet de synchronisation. Les valeurs possibles du champ pr_stype sont indiquées dans le Tableau 25–4.

TABLEAU 25–4 Valeurs de pr_stype

SOBJ_MUTEX	Objet de synchronisation mutex du noyau. Utilisé pour sérialiser l'accès aux zones de données partagées du noyau. Pour plus d'informations sur les objets de synchronisation mutex du noyau, reportez-vous au Chapitre18, “Fournisseur lockstat” et à la page Web mutex_init(9F).

TABLEAU 25–4 Valeurs de `pr_stype` *(Suite)*

`SOBJ_RWLOCK`	Objet de synchronisation de lecture/écriture du noyau. Utilisé pour synchroniser l'accès aux objets partagés du noyau pouvant autoriser plusieurs lecteurs simultanés ou un seul rédacteur. Pour plus d'informations sur les objets de synchronisation de lecture/écriture du noyau, reportez-vous au Chapitre18, “Fournisseur `lockstat`” et à la page Web `rwlock`(9F).
`SOBJ_CV`	Objet de synchronisation de variable de condition. Une variable de condition est conçue pour attendre indéfiniment jusqu'à ce qu'une condition soit vraie. Les variables de condition sont généralement utilisées pour synchroniser pour des raisons autres que l'accès à une zone de données partagées, et constituent le mécanisme généralement utilisé lorsqu'un processus attend indéfiniment un programme. Par exemple, un blocage dans `poll`(2), `pause`(2), `wait`(3C) et similaire.
`SOBJ_SEMA`	Objet de synchronisation de sémaphore. Objet de synchronisation universel qui, comme des objets de variable de condition, ne suit pas la notion de propriété. La propriété étant nécessaire pour mettre en œuvre l'héritage de la priorité dans le noyau Solaris, l'absence de propriété inhérente aux objets de sémaphore empêche leur totale utilisation. Pour plus d'informations, reportez-vous à la page Web `semaphore`(9F).
`SOBJ_USER`	Objet de synchronisation au niveau utilisateur. Tout blocage d'objets de synchronisation au niveau utilisateur est géré avec des objets de synchronisation `SOBJ_USER`. Les objets de synchronisation au niveau utilisateur incluent ceux créés avec `mutex_init`(3C), `sema_init`(3C), `rwlock_init`(3C), `cond_init`(3C) et leurs équivalents POSIX.
`SOBJ_USER_PI`	Objet de synchronisation au niveau utilisateur mettant en œuvre l'héritage de la priorité. Certains objets de synchronisation au niveau utilisateur suivant la propriété autorisent en outre l'héritage de la priorité. Par exemple, des objets mutex créés avec `pthread_mutex_init`(3C) peuvent servir à hériter la priorité via `pthread_mutexattr_setprotocol`(3C).
`SOBJ_SHUTTLE`	Objet de synchronisation d'accélération. Les objets d'accélération sont utilisés pour mettre en œuvre des portes. Pour plus d'informations, voir door_create(3DOOR).

Le champ `pr_state` est défini sur l'une des valeurs indiquées dans le Tableau 25–5. Le champ `pr_sname` est défini sur un caractère correspondant indiqué entre parenthèses dans le même tableau.

TABLEAU 25–5 Valeurs de `pr_state`

`SSLEEP` (`S`)	Le thread est au repos. La sonde `sched:::sleep` se déclenche immédiatement avant la transition de l'état d'un thread en `SSLEEP`.

TABLEAU 25–5 Valeurs de pr_state *(Suite)*

SRUN (R)	Le thread peut être exécuté, mais ne l'est pas actuellement. La sonde sched:::enqueue se déclenche immédiatement avant la transition de l'état d'un thread en SRUN.
SZOMB (Z)	Le thread est un LWP zombie.
SSTOP (T)	Le thread est arrêté, en raison d'une directive proc(4) explicite ou d'un autre mécanisme d'arrêt.
SIDL (I)	Le thread se trouve dans un état intermédiaire lors de la création du processus.
SONPROC (O)	Le thread est exécuté sur une CPU. La sonde sched:::on-cpu se déclenche dans le cadre du thread SONPROC sur une courte durée après la transition de l'état du thread en SONPROC.

psinfo_t

Plusieurs sondes proc comprennent un argument du type psinfo_t, structure abordée dans proc(4). La définition de la structure psinfo_t, telle qu'accessible aux clients DTrace, est la suivante :

```
typedef struct psinfo {
    int      pr_nlwp;            /* number of active lwps in the process */
    pid_t    pr_pid;             /* unique process id */
    pid_t    pr_ppid;            /* process id of parent */
    pid_t    pr_pgid;            /* pid of process group leader */
    pid_t    pr_sid;             /* session id */
    uid_t    pr_uid;             /* real user id */
    uid_t    pr_euid;            /* effective user id */
    gid_t    pr_gid;             /* real group id */
    gid_t    pr_egid;            /* effective group id */
    uintptr_t pr_addr;           /* address of process */
    dev_t    pr_ttydev;          /* controlling tty device (or PRNODEV) */
    timestruc_t pr_start;        /* process start time, from the epoch */
    char     pr_fname[PRFNSZ];   /* name of execed file */
    char     pr_psargs[PRARGSZ]; /* initial characters of arg list */
    int      pr_argc;            /* initial argument count */
    uintptr_t pr_argv;           /* address of initial argument vector */
    uintptr_t pr_envp;           /* address of initial environment vector */
    char     pr_dmodel;          /* data model of the process */
    taskid_t pr_taskid;          /* task id */
    projid_t pr_projid;          /* project id */
    poolid_t pr_poolid;          /* pool id */
    zoneid_t pr_zoneid;          /* zone id */
} psinfo_t;
```

Le champ pr_dmodel est défini sur PR_MODEL_ILP32, représentant un processus 32 bits ou sur PR_MODEL_LP64, représentant un processus 64 bits.

Exemples

exec

Vous pouvez utiliser la sonde exec pour déterminer facilement les programmes en cours d'exécution et leurs auteurs, tel qu'illustré dans l'exemple suivant :

```
#pragma D option quiet

proc:::exec
{
        self->parent = execname;
}

proc:::exec-success
/self->parent != NULL/
{
    @[self->parent, execname] = count();
    self->parent = NULL;
}

proc:::exec-failure
/self->parent != NULL/
{
    self->parent = NULL;
}

END
{
    printf("%-20s %-20s %s\n", "WHO", "WHAT", "COUNT");
    printa("%-20s %-20s %@d\n", @);
}
```

L'exécution de l'exemple de script sur une courte période sur une machine intégrée entraîne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./whoexec.d
^C
WHO                  WHAT                 COUNT
make.bin             yacc                 1
tcsh                 make                 1
make.bin             spec2map             1
sh                   grep                 1
lint                 lint2                1
sh                   lint                 1
sh                   ln                   1
```

```
cc                    ld                    1
make.bin              cc                    1
lint                  lint1                 1
sh                    lex                   1
make.bin              mv                    2
sh                    sh                    3
sh                    make                  3
sh                    sed                   4
sh                    tr                    4
make                  make.bin              4
sh                    install.bin           5
sh                    rm                    6
cc                    ir2hf                 33
cc                    ube                   33
sh                    date                  34
sh                    mcs                   34
cc                    acomp                 34
sh                    cc                    34
sh                    basename              34
basename              expr                  34
make.bin              sh                    87
```

start **et** exit

Pour connaître la durée d'exécution de programmes de la création à la fin, vous pouvez activer les sondes `start` et `exit`, tel qu'illustré dans l'exemple suivant :

```
proc:::start
{
    self->start = timestamp;
}

proc:::exit
/self->start/
{
    @[execname] = quantize(timestamp - self->start);
    self->start = 0;
}
```

L'exécution de l'exemple de script sur le serveur intégré pendant plusieurs secondes entraîne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./progtime.d
dtrace: script './progtime.d' matched 2 probes
^C

  ir2hf
```

```
           value  ------------- Distribution ------------- count
         4194304 |                                         0
         8388608 |@                                        1
        16777216 |@@@@@@@@@@@@@@@@                         14
        33554432 |@@@@@@@@@@                               9
        67108864 |@@@                                      3
       134217728 |@                                        1
       268435456 |@@@@                                     4
       536870912 |@                                        1
      1073741824 |                                         0

  ube
           value  ------------- Distribution ------------- count
        16777216 |                                         0
        33554432 |@@@@@@@                                  6
        67108864 |@@@                                      3
       134217728 |@@                                       2
       268435456 |@@@@                                     4
       536870912 |@@@@@@@@@@@@                             10
      1073741824 |@@@@@@@                                  6
      2147483648 |@@                                       2
      4294967296 |                                         0

  acomp
           value  ------------- Distribution ------------- count
         8388608 |                                         0
        16777216 |@@                                       2
        33554432 |                                         0
        67108864 |@                                        1
       134217728 |@@@                                      3
       268435456 |                                         0
       536870912 |@@@@@                                    5
      1073741824 |@@@@@@@@@@@@@@@@@@@@@@@@@                22
      2147483648 |@                                        1
      4294967296 |                                         0

  cc
           value  ------------- Distribution ------------- count
        33554432 |                                         0
        67108864 |@@@                                      3
       134217728 |@                                        1
       268435456 |                                         0
       536870912 |@@@@                                     4
      1073741824 |@@@@@@@@@@@@@@                           13
      2147483648 |@@@@@@@@@@@@                             11
      4294967296 |@@@                                      3
      8589934592 |                                         0
```

```
  sh
           value  ------------- Distribution ------------- count
          262144 |                                         0
          524288 |@                                        5
         1048576 |@@@@@@@                                  29
         2097152 |                                         0
         4194304 |                                         0
         8388608 |@@@                                      12
        16777216 |@@                                       9
        33554432 |@@                                       9
        67108864 |@@                                       8
       134217728 |@                                        7
       268435456 |@@@@@                                    20
       536870912 |@@@@@@                                   26
      1073741824 |@@@                                      14
      2147483648 |@@                                       11
      4294967296 |                                         3
      8589934592 |                                         1
     17179869184 |                                         0

  make.bin
           value  ------------- Distribution ------------- count
        16777216 |                                         0
        33554432 |@                                        1
        67108864 |@                                        1
       134217728 |@@                                       2
       268435456 |                                         0
       536870912 |@@                                       2
      1073741824 |@@@@@@@@@                                9
      2147483648 |@@@@@@@@@@@@@@@                          14
      4294967296 |@@@@@@                                   6
      8589934592 |@@                                       2
     17179869184 |                                         0
```

lwp-start et lwp-exit

Plutôt que de connaître la durée d'exécution d'un processus particulier, vous pouvez souhaiter connaître la durée d'exécution de chaque thread. L'exemple suivant illustre l'utilisation des sondes `lwp-start` et `lwp-exit` dans ce but :

```
proc:::lwp-start
/tid != 1/
{
    self->start = timestamp;
}

proc:::lwp-exit
```

```
/self->start/
{
    @[execname] = quantize(timestamp - self->start);
    self->start = 0;
}
```

L'exécution de l'exemple de script sur un serveur NFS ou de calendriers entraîne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./lwptime.d
dtrace: script './lwptime.d' matched 3 probes
^C

  nscd
           value  ------------- Distribution ------------- count
          131072 |                                         0
          262144 |@                                        18
          524288 |@@                                       24
         1048576 |@@@@@@@                                  75
         2097152 |@@@@@@@@@@@@@@@@@@@@@@@                  245
         4194304 |@@                                       22
         8388608 |@@                                       24
        16777216 |                                         6
        33554432 |                                         3
        67108864 |                                         1
       134217728 |                                         1
       268435456 |                                         0

  mountd
           value  ------------- Distribution ------------- count
          524288 |                                         0
         1048576 |@                                        15
         2097152 |@                                        24
         4194304 |@@@                                      51
         8388608 |@                                        17
        16777216 |@                                        24
        33554432 |@                                        15
        67108864 |@@@@                                     57
       134217728 |@                                        28
       268435456 |@                                        26
       536870912 |@@                                       39
      1073741824 |@@@                                      45
      2147483648 |@@@@@                                    72
      4294967296 |@@@@@                                    77
      8589934592 |@@@                                      55
     17179869184 |                                         14
     34359738368 |                                         2
     68719476736 |                                         0
```

```
automountd
         value  ------------- Distribution ------------- count
       1048576 |                                         0
       2097152 |                                         3
       4194304 |@@@@                                     146
       8388608 |                                         6
      16777216 |                                         6
      33554432 |                                         9
      67108864 |@@@@@                                    203
     134217728 |@@                                       87
     268435456 |@@@@@@@@@@@@@@@                          534
     536870912 |@@@@@@                                   223
    1073741824 |@                                        45
    2147483648 |                                         20
    4294967296 |                                         26
    8589934592 |                                         20
   17179869184 |                                         19
   34359738368 |                                         7
   68719476736 |                                         2
  137438953472 |                                         0

iCald
         value  ------------- Distribution ------------- count
       8388608 |                                         0
      16777216 |@@@@@@@                                  20
      33554432 |@@@                                      9
      67108864 |@@                                       8
     134217728 |@@@@@                                    16
     268435456 |@@@@                                     11
     536870912 |@@@@                                     11
    1073741824 |@                                        4
    2147483648 |                                         2
    4294967296 |                                         0
    8589934592 |@@                                       8
   17179869184 |@                                        5
   34359738368 |@                                        4
   68719476736 |@@                                       6
  137438953472 |@                                        4
  274877906944 |                                         2
  549755813888 |                                         0
```

signal-send

Vous pouvez utiliser la sonde signal-send pour déterminer le processus d'envoi et de réception associé à chaque signal, tel qu'illustré dans l'exemple suivant :

```
#pragma D option quiet

proc:::signal-send
{
    @[execname, stringof(args[1]->pr_fname), args[2]] = count();
}

END
{
    printf("%20s %20s %12s %s\n",
        "SENDER", "RECIPIENT", "SIG", "COUNT");
    printa("%20s %20s %12d %@d\n", @);
}
```

Exécuter ce script engendre une sortie identique à l'exemple suivant :

```
# dtrace -s ./sig.d
^C
              SENDER            RECIPIENT          SIG COUNT
               xterm               dtrace            2 1
               xterm          soffice.bin            2 1
                  tr                 init           18 1
               sched                 test           18 1
               sched                fvwm2           18 1
                bash                 bash           20 1
                 sed                 init           18 2
               sched                  ksh           18 15
               sched                 Xsun           22 471
```

Stabilité

Le fournisseur proc utilise un mécanisme de stabilité DTrace pour décrire ses stabilités, comme illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	En cours d'évolution	En cours d'évolution	ISA

CHAPITRE 26

Fournisseur sched

Le fournisseur sched propose des sondes liées à la planification de CPU. Les CPU étant la seule ressource que tous les threads doivent consommer, le fournisseur sched est très utile pour comprendre le comportement systémique. Par exemple, grâce au fournisseur sched, vous pouvez savoir quand et pourquoi les threads sont en sommeil, exécutés, changent de priorité ou réveillent d'autres threads.

Sondes

Les sondes sched sont décrites dans le Tableau 26–1.

TABLEAU 26–1 Sondes sched

Sonder	Description
change-pri	Sonde se déclenchant si la priorité d'un thread va être changée. lwpsinfo_t du thread est indiqué par args[0]. La priorité actuelle du thread est indiquée dans le champ pr_pri de cette structure. psinfo_t du processus contenant le thread est indiqué par args[1]. La nouvelle priorité du thread est indiquée dans args[2].
dequeue	Sonde se déclenchant immédiatement avant qu'un thread exécutable ne soit retiré d'une file d'attente d'exécution. lwpsinfo_t du thread retiré de la file d'attente est indiqué par args[0]. psinfo_t du processus contenant le thread est indiqué par args[1]. cpuinfo_t de la CPU à partir de laquelle le thread est retiré de la file d'attente est indiqué par args[2]. Si le thread est retiré d'une file d'attente d'exécution non associée à une CPU particulière, l'élément cpu_id de cette structure sera -1.

TABLEAU 26–1 Sondes sched *(Suite)*

Sonder	Description
enqueue	Sonde se déclenchant immédiatement avant qu'un thread exécutable ne soit placé dans une file d'attente d'exécution. lwpsinfo_t du thread placé dans une file d'attente est indiqué par args[0]. psinfo_t du processus contenant le thread est indiqué par args[1]. cpuinfo_t de la CPU pour laquelle le thread est placé dans une file d'attente est indiqué par args[2]. Si le thread est placé dans une file d'attente d'exécution non associée à une CPU particulière, l'élément cpu_id de cette structure sera -1. La valeur args[3] est un booléen indiquant si le thread sera placé en tête de la file d'attente. La valeur n'est pas de zéro si le thread doit être placé en tête de la file d'attente, et de zéro s'il doit l'être à la fin.
off-cpu	Sonde se déclenchant lorsque la CPU actuelle va terminer l'exécution d'un thread. La variable curcpu indique la CPU en cours. La variable curlwpsinfo indique le thread dont l'exécution se termine. La variable curpsinfo indique le processus contenant le thread actuel. La structure lwpsinfo_t du thread que la CPU actuelle n'exécutera pas est indiquée par args[0] . psinfo_t du processus contenant le thread suivant est indiqué par args[1].
on-cpu	Sonde se déclenchant lorsqu'une CPU vient de commencer l'exécution d'un thread. La variable curcpu indique la CPU en cours. La variable curlwpsinfo indique le thread dont l'exécution commence. La variable curpsinfo indique le processus contenant le thread actuel.
preempt	Sonde se déclenchant immédiatement après la préemption du thread actuel. Après le déclenchement de cette sonde, le thread actuel sélectionnera un thread à exécuter et la sonde off-cpu se déclenchera pour le thread actuel. Dans certains cas, un thread sur une CPU sera déplacé, mais le thread concerné sera exécuté simultanément sur une autre CPU. Dans cette situation, la sonde preempt se déclenche, mais le dispatcheur n'est pas en mesure de trouver un thread de priorité supérieure à exécuter et la sonde remain-cpu se déclenche à la place de la sonde off-cpu.
remain-cpu	Sonde se déclenchant en cas de décision de planification, alors que le dispatcheur a choisi de poursuivre l'exécution du thread actuel. La variable curcpu indique la CPU en cours. La variable curlwpsinfo indique le thread dont l'exécution commence. La variable curpsinfo indique le processus contenant le thread actuel.
schedctl-nopreempt	Sonde se déclenchant lorsqu'un thread est déplacé puis replacé en *tête* de la file d'attente en raison d'une demande de contrôle de préemption. Pour plus d'informations sur le contrôle de préemption, reportez-vous à schedctl_init(3C) Comme pour preempt, off-cpu ou remain-cpu se déclenche après schedctl-nopreempt. schedctl-nopreempt représentant un replacement du thread actuel en tête de la file d'attente, remain-cpu est plus susceptible de se déclencher après schedctl-nopreempt que off-cpu. lwpsinfo_t du thread déplacé est indiqué par args[0]. psinfo_t du processus contenant le thread est indiqué par args[1].

TABLEAU 26–1 Sondes sched (Suite)

Sonder	Description
schedctl-preempt	Sonde se déclenchant lorsqu'un thread utilisant le contrôle de préemption n'est pas déplacé et replacé à la *fin* de la file d'attente d'exécution. Pour plus d'informations sur le contrôle de préemption, reportez-vous à schedctl_init(3C) Comme pour preempt, off-cpu ou remain-cpu se déclenche après schedctl-preempt. Comme preempt (et contrairement à schedctl-nopreempt), schedctl-preempt représente un replacement du thread actuel à la fin de la file d'attente d'exécution. Par conséquent, off-cpu est plus susceptible de se déclencher après schedctl-preempt que remain-cpu. lwpsinfo_t du thread déplacé est indiqué par args[0]. psinfo_t du processus contenant le thread est indiqué par args[1].
schedctl-yield	Sonde se déclenchant lorsqu'un thread dont le contrôle de préemption est activé et le code exécuté a prolongé artificiellement la tranche de temps pour amener la CPU vers d'autres threads.
sleep	Sonde se déclenchant immédiatement avant que le thread actuel ne sommeille sur un objet de synchronisation. Le type de l'objet de synchronisation est contenu dans l'élément pr_stype de lwpsinfo_t indiqué par curlwpsinfo . L'adresse de l'objet de synchronisation est contenue dans l'élément pr_wchan de lwpsinfo_t indiqué par curlwpsinfo. La signification de cette adresse est un détail privé de mise en œuvre, mais la valeur d'adresse peut être considérée comme un jeton unique de l'objet de synchronisation.
surrender	Sonde se déclenchant lorsqu'une CPU a été amenée, par une autre CPU, à prendre une décision de planification – souvent car un thread de priorité supérieure est devenu exécutable.
tick	Sonde se déclenchant dans le cadre d'une comptabilisation des tops d'horloge. Dans la comptabilisation des tops d'horloge, celle de la CPU est exécutée en déterminant les threads et processus en cours d'exécution en cas d'interruption à intervalle fixe. lwpsinfo_t correspondant au thread, auquel une heure CPU est affectée, est indiqué par args[0]. psinfo_t correspondant au processus contenant le thread est indiqué par args[1] .
wakeup	Sonde se déclenchant immédiatement avant que le thread actuel ne réveille un thread en sommeil sur un objet de synchronisation. lwpsinfo_t du thread en sommeil est indiqué par args[0]. psinfo_t du processus contenant le thread en sommeil est indiqué par args[1]. Le type de l'objet de synchronisation est contenu dans l'élément pr_stype de lwpsinfo_t du thread en sommeil. L'adresse de l'objet de synchronisation est contenue dans l'élément pr_wchan de lwpsinfo_t du thread en sommeil. La signification de cette adresse est un détail privé de mise en œuvre, mais la valeur d'adresse peut être considérée comme un jeton unique de l'objet de synchronisation.

Arguments

Les types d'argument des sondes sched sont répertoriés dans le Tableau 26–2. Les arguments sont décrits dans le Tableau 26–1.

TABLEAU 26–2 Arguments de sonde sched

Sonder	args[0]	args[1]	args[2]	args[3]
change-pri	lwpsinfo_t *	psinfo_t *	pri_t	—
dequeue	lwpsinfo_t *	psinfo_t *	cpuinfo_t *	—
enqueue	lwpsinfo_t *	psinfo_t *	cpuinfo_t *	int
off-cpu	lwpsinfo_t *	psinfo_t *	—	—
on-cpu	—	—	—	—
preempt	—	—	—	—
remain-cpu	—	—	—	—
schedctl-nopreempt	lwpsinfo_t *	psinfo_t *	—	—
schedctl-preempt	lwpsinfo_t *	psinfo_t *	—	—
schedctl-yield	lwpsinfo_t *	psinfo_t *	—	—
sleep	—	—	—	—
surrender	lwpsinfo_t *	psinfo_t *	—	—
tick	lwpsinfo_t *	psinfo_t *	—	—
wakeup	lwpsinfo_t *	psinfo_t *	—	—

Comme indiqué dans le Tableau 26–2, de nombreuses sondes sched disposent d'arguments constitués d'un pointeur sur lwpsinfo_t et d'un pointeur sur psinfo_t, indiquant ainsi un thread et le processus contenant le thread, respectivement. Ces structures sont décrites en détail aux sections “lwpsinfo_t” à la page 264 et “psinfo_t” à la page 267.

cpuinfo_t

La structure cpuinfo_t définit une CPU. Comme indiqué dans le Tableau 26–2, les arguments des sondes enqueue et dequeue incluent un pointeur sur cpuinfo_t. De plus, la structure cpuinfo_t correspondant à la CPU actuelle est indiquée par la variable curcpu. La définition de la structure cpuinfo_t est la suivante :

```
typedef struct cpuinfo {
    processorid_t cpu_id;           /* CPU identifier */
    psetid_t cpu_pset;              /* processor set identifier */
    chipid_t cpu_chip;              /* chip identifier */
    lgrp_id_t cpu_lgrp;             /* locality group identifer */
    processor_info_t cpu_info;      /* CPU information */
} cpuinfo_t;
```

Le membre cpu_id est l'identificateur de processeur, tel que renvoyé par psrinfo(1M) et p_online(2).

L'élément cpu_pset est le processeur défini contenant la CPU, le cas échéant. Pour plus d'informations sur les définitions du processeur, reportez-vous à psrset(1M).

L'élément cpu_chip est l'identificateur de la puce physique. Les puces physiques peuvent contenir plusieurs CPU. Pour plus d'informations, reportez-vous à psrinfo(1M).

L'élément cpu_lgrp est l'identificateur du groupe de latence associé à la CPU. Pour plus d'informations sur les groupes de latence, reportez-vous à liblgrp(3LIB).

Le membre cpu_info est la structure processor_info_t associée à la CPU, tel que renvoyé par processor_info(2).

Exemples

on-cpu et off-cpu

Une question commune que vous pourriez vous poser est de connaître les CPU exécutant des threads ainsi que la durée du processus. Vous pouvez utiliser les sondes on-cpu et off-cpu pour répondre facilement à cette question sur l'ensemble d'un système, tel qu'illustré dans l'exemple suivant :

```
sched:::on-cpu
{
    self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
    @[cpu] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

Exécuter le script ci-dessus engendre une sortie similaire à l'exemple suivant :

```
# dtrace -s ./where.d
dtrace: script './where.d' matched 5 probes
^C

        0
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@                                       37
            8192 |@@@@@@@@@@@@@                            212
           16384 |@                                        30
           32768 |                                         10
           65536 |@                                        17
          131072 |                                         12
          262144 |                                         9
          524288 |                                         6
         1048576 |                                         5
         2097152 |                                         1
         4194304 |                                         3
         8388608 |@@@@                                     75
        16777216 |@@@@@@@@@@@@                             201
        33554432 |                                         6
        67108864 |                                         0

        1
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@                                        6
            8192 |@@@@                                     23
           16384 |@@@                                      18
           32768 |@@@@                                     22
           65536 |@@@@                                     22
          131072 |@                                        7
          262144 |                                         5
          524288 |                                         2
         1048576 |                                         3
         2097152 |@                                        9
         4194304 |                                         4
         8388608 |@@@                                      18
        16777216 |@@@                                      19
        33554432 |@@@                                      16
        67108864 |@@@@                                     21
       134217728 |@@                                       14
       268435456 |                                         0
```

La sortie ci-dessus indique que les threads de la CPU 1 tendent à être exécutés moins de 100 microsecondes d'affilée ou pendant 10 millisecondes environ. Un écart considérable entre deux clusters de données est illustré dans l'histogramme. Vous pouvez également être intéressé de connaître les CPU exécutant un processus particulier. Vous pouvez également utiliser les

sondes `on-cpu` et `off-cpu` pour répondre à cette question. Le script suivant affiche les CPU exécutant une application spécifiée pendant dix secondes :

```
#pragma D option quiet

dtrace:::BEGIN
{
    start = timestamp;
}

sched:::on-cpu
/execname == $$1/
{
    self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
    @[cpu] = sum(timestamp - self->ts);
    self->ts = 0;
}

profile:::tick-1sec
/++x == 10/
{
    exit(0);
}

dtrace:::END
{
    printf("CPU distribution of imapd over %d seconds:\n\n",
        (timestamp - start) / 1000000000);
    printf("CPU microseconds\n--- ------------\n");
    normalize(@, 1000);
    printa("%3d %@d\n", @);
}
```

L'exécution du script ci-dessus sur un serveur de messagerie d'envergure et la spécification du démon IMAP donnent une sortie similaire à l'exemple suivant :

```
# dtrace -s ./whererun.d imapd
CPU distribution of imapd over 10 seconds:

CPU microseconds
--- ------------
 15 10102
 12 16377
```

```
 21 25317
 19 25504
 17 35653
 13 41539
 14 46669
 20 57753
 22 70088
 16 115860
 23 127775
 18 160517
```

Solaris prend en compte la durée de sommeil d'un thread lors de la sélection d'une CPU sur laquelle exécuter le thread : un thread en sommeil moins longtemps n'est généralement pas migré. Vous pouvez utiliser les sondes `off-cpu` et `on-cpu` pour observer ce comportement :

```
sched:::off-cpu
/curlwpsinfo->pr_state == SSLEEP/
{
    self->cpu = cpu;
    self->ts = timestamp;
}

sched:::on-cpu
/self->ts/
{
    @[self->cpu == cpu ?
        "sleep time, no CPU migration" : "sleep time, CPU migration"] =
        lquantize((timestamp - self->ts) / 1000000, 0, 500, 25);
    self->ts = 0;
    self->cpu = 0;
}
```

L'exécution du script ci-dessus pendant 30 secondes environ donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./howlong.d
dtrace: script './howlong.d' matched 5 probes
^C
 sleep time, CPU migration
           value  ------------- Distribution ------------ count
             < 0 |                                         0
               0 |@@@@@@@                                  6838
              25 |@@@@@                                    4714
              50 |@@@                                      3108
              75 |@                                        1304
             100 |@                                        1557
             125 |@                                        1425
             150 |                                         894
```

```
           175 |@                                        1526
           200 |@@                                       2010
           225 |@@                                       1933
           250 |@@                                       1982
           275 |@@                                       2051
           300 |@@                                       2021
           325 |@                                        1708
           350 |@                                        1113
           375 |                                         502
           400 |                                         220
           425 |                                         106
           450 |                                         54
           475 |                                         40
        >= 500 |@                                        1716

sleep time, no CPU migration
         value  -------------- Distribution ------------ count
           < 0 |                                         0
             0 |@@@@@@@@@@@@                             58413
            25 |@@@                                      14793
            50 |@@                                       10050
            75 |                                         3858
           100 |@                                        6242
           125 |@                                        6555
           150 |                                         3980
           175 |@                                        5987
           200 |@                                        9024
           225 |@                                        9070
           250 |@@                                       10745
           275 |@@                                       11898
           300 |@@                                       11704
           325 |@@                                       10846
           350 |@                                        6962
           375 |                                         3292
           400 |                                         1713
           425 |                                         585
           450 |                                         201
           475 |                                         96
        >= 500 |                                         3946
```

L'exemple de sortie indique qu'il existe bien plus de cas de non migrations que de migrations. De plus, plus la durée de sommeil est longue, plus la migration est probable. Les répartitions sont considérablement différentes dans la plage inférieure aux 100 millisecondes, mais semblent très similaires à mesure que les durées de sommeil augmentent. Ce résultat semblerait indiquer que la durée de sommeil n'est pas un facteur dans la prise de décision de planification lorsqu'un seuil donné est dépassé.

Le dernier exemple utilisant `off-cpu` et `on-cpu` illustre l'utilisation de ces sondes avec le champ `pr_stype` pour déterminer pourquoi des threads sommeillent et pendant combien de temps :

```
sched:::off-cpu
/curlwpsinfo->pr_state == SSLEEP/
{
    /*
     * We're sleeping.  Track our sobj type.
     */
    self->sobj = curlwpsinfo->pr_stype;
    self->bedtime = timestamp;
}

sched:::off-cpu
/curlwpsinfo->pr_state == SRUN/
{
    self->bedtime = timestamp;
}

sched:::on-cpu
/self->bedtime && !self->sobj/
{
    @["preempted"] = quantize(timestamp - self->bedtime);
    self->bedtime = 0;
}

sched:::on-cpu
/self->sobj/
{
    @[self->sobj == SOBJ_MUTEX ? "kernel-level lock" :
        self->sobj == SOBJ_RWLOCK ? "rwlock" :
        self->sobj == SOBJ_CV ? "condition variable" :
        self->sobj == SOBJ_SEMA ? "semaphore" :
        self->sobj == SOBJ_USER ? "user-level lock" :
        self->sobj == SOBJ_USER_PI ? "user-level prio-inheriting lock" :
        self->sobj == SOBJ_SHUTTLE ? "shuttle" : "unknown"] =
        quantize(timestamp - self->bedtime);

    self->sobj = 0;
    self->bedtime = 0;
}
```

L'exécution du script ci-dessus pendant plusieurs secondes donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./whatfor.d
dtrace: script './whatfor.d' matched 12 probes
^C
 kernel-level lock
           value  ------------- Distribution ------------ count
           16384 |                                         0
```

```
         32768 |@@@@@@@@                                 3
         65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            11
        131072 |@@                                       1
        262144 |                                         0

preempted
         value  -------------- Distribution ------------ count
         16384 |                                         0
         32768 |                                         4
         65536 |@@@@@@@@                                 408
        131072 |@@@@@@@@@@@@@@@@@@@@@@                   1031
        262144 |@@@                                      156
        524288 |@@                                       116
       1048576 |@                                        51
       2097152 |                                         42
       4194304 |                                         16
       8388608 |                                         15
      16777216 |                                         4
      33554432 |                                         8
      67108864 |                                         0

semaphore
         value  -------------- Distribution ------------ count
         32768 |                                         0
         65536 |@@                                       61
        131072 |@@@@@@@@@@@@@@@@@@@@@@@@                 553
        262144 |@@                                       63
        524288 |@                                        36
       1048576 |                                         7
       2097152 |                                         22
       4194304 |@                                        44
       8388608 |@@@                                      84
      16777216 |@                                        36
      33554432 |                                         3
      67108864 |                                         6
     134217728 |                                         0
     268435456 |                                         0
     536870912 |                                         0
    1073741824 |                                         0
    2147483648 |                                         0
    4294967296 |                                         0
    8589934592 |                                         0
   17179869184 |                                         1
   34359738368 |                                         0

shuttle
         value  -------------- Distribution ------------ count
         32768 |                                         0
```

```
           65536 |@@@@@                                    2
          131072 |@@@@@@@@@@@@@@@@                         6
          262144 |@@@@@                                    2
          524288 |                                         0
         1048576 |                                         0
         2097152 |                                         0
         4194304 |@@@@@                                    2
         8388608 |                                         0
        16777216 |                                         0
        33554432 |                                         0
        67108864 |                                         0
       134217728 |                                         0
       268435456 |                                         0
       536870912 |                                         0
      1073741824 |                                         0
      2147483648 |                                         0
      4294967296 |@@@@@                                    2
      8589934592 |                                         0
     17179869184 |@@                                       1
     34359738368 |                                         0

  condition variable
           value  ------------- Distribution ------------- count
           32768 |                                         0
           65536 |                                         122
          131072 |@@@@@                                    1579
          262144 |@                                        340
          524288 |                                         268
         1048576 |@@@                                      1028
         2097152 |@@@                                      1007
         4194304 |@@@                                      1176
         8388608 |@@@@                                     1257
        16777216 |@@@@@@@@@@@@@@                           4385
        33554432 |                                         295
        67108864 |                                         157
       134217728 |                                         96
       268435456 |                                         48
       536870912 |                                         144
      1073741824 |                                         10
      2147483648 |                                         22
      4294967296 |                                         18
      8589934592 |                                         5
     17179869184 |                                         6
     34359738368 |                                         4
     68719476736 |                                         0
```

enqueue et dequeue

Lorsqu'une CPU passe au repos, le dispatcheur recherche des tâches mises en file d'attente sur d'autres CPU (pas au repos). L'exemple suivant utilise la sonde dequeue pour connaître la fréquence de transfert des applications et par quelle CPU :

```
#pragma D option quiet

sched:::dequeue
/args[2]->cpu_id != --1 && cpu != args[2]->cpu_id &&
    (curlwpsinfo->pr_flag & PR_IDLE)/
{
    @[stringof(args[1]->pr_fname), args[2]->cpu_id] =
        lquantize(cpu, 0, 100);
}

END
{
    printa("%s stolen from CPU %d by:\n%@d\n", @);
}
```

La fin de la sortie de l'exécution du script ci-dessus sur un système à 4 CPU donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./whosteal.d
^C
...
 nscd stolen from CPU 1 by:

           value  -------------- Distribution ------------ count
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 28
               3 |                                         0

snmpd stolen from CPU 1 by:

           value  -------------- Distribution ------------ count
             < 0 |                                         0
               0 |@                                        1
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     31
               3 |@@                                       2
               4 |                                         0

sched stolen from CPU 1 by:

           value  -------------- Distribution ------------ count
             < 0 |                                         0
```

```
               0 |@@                                      3
               1 |                                        0
               2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@        36
               3 |@@@@                                    5
               4 |                                        0
```

Plutôt que de connaître les CPU actives, vous pouvez souhaiter connaître les CPU sur lesquelles des processus et des threads sont en attente d'exécution. Vous pouvez utiliser les sondes enqueue et dequeue ensemble pour répondre à cette question :

```
sched:::enqueue
{
    self->ts = timestamp;
}

sched:::dequeue
/self->ts/
{
    @[args[2]->cpu_id] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

L'exécution du script ci-dessus pendant plusieurs secondes donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./qtime.d
dtrace: script './qtime.d' matched 5 probes
^C
       -1
           value  -------------- Distribution ------------ count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
           16384 |                                         0

        0
           value  -------------- Distribution ------------ count
            1024 |                                         0
            2048 |@@@@@@@@@@@@@@@                          262
            4096 |@@@@@@@@@@@@@                            227
            8192 |@@@@@                                    87
           16384 |@@@                                      54
           32768 |                                         7
           65536 |                                         9
          131072 |                                         1
          262144 |                                         5
          524288 |                                         4
         1048576 |                                         2
         2097152 |                                         0
```

```
     4194304 |                                         0
     8388608 |                                         0
    16777216 |                                         1
    33554432 |                                         2
    67108864 |                                         2
   134217728 |                                         0
   268435456 |                                         0
   536870912 |                                         0
  1073741824 |                                         1
  2147483648 |                                         1
  4294967296 |                                         0

   1
        value  ------------- Distribution ------------ count
         1024 |                                         0
         2048 |@@@@                                     49
         4096 |@@@@@@@@@@@@@@@@@@@@                     241
         8192 |@@@@@@@                                  91
        16384 |@@@@                                     55
        32768 |                                         7
        65536 |                                         3
       131072 |                                         2
       262144 |                                         1
       524288 |                                         0
      1048576 |                                         0
      2097152 |                                         0
      4194304 |                                         0
      8388608 |                                         0
     16777216 |                                         0
     33554432 |                                         3
     67108864 |                                         1
    134217728 |                                         4
    268435456 |                                         2
    536870912 |                                         0
   1073741824 |                                         3
   2147483648 |                                         2
   4294967296 |                                         0
```

Notez les valeurs différentes de zéro au bas de la sortie. Ces points de données représentent plusieurs instances sur les CPU où un thread a été placé en file d'attente d'exécution pendant plusieurs *secondes*.

Plutôt que de rechercher les délais d'attente, vous pouvez souhaiter observer la longueur de la file d'attente dans le temps. Grâce aux sondes enqueue et dequeue, vous pouvez définir un ensemble associatif pour suivre la longueur de la file d'attente :

```
sched:::enqueue
{
    this->len = qlen[args[2]->cpu_id]++;
```

```
    @[args[2]->cpu_id] = lquantize(this->len, 0, 100);
}

sched:::dequeue
/qlen[args[2]->cpu_id]/
{
    qlen[args[2]->cpu_id]—;
}
```

L'exécution du script ci-dessus pendant 30 secondes environ sur un système portable à un processeur au repos donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./qlen.d
dtrace: script './qlen.d' matched 5 probes
^C
        0
           value  ------------- Distribution ------------ count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@                110626
               1 |@@@@@@@@@@                                41142
               2 |@@                                        12655
               3 |@                                         5074
               4 |                                          1722
               5 |                                          701
               6 |                                          302
               7 |                                          63
               8 |                                          23
               9 |                                          12
              10 |                                          24
              11 |                                          58
              12 |                                          14
              13 |                                          3
              14 |                                          0
```

La sortie est grossièrement ce que vous auriez pu attendre d'un système au repos : le thread exécutable est placé en file d'attente quasiment tout le temps et la file d'attente d'exécution est très courte (trois threads maximum). Cependant, étant donné que le système était au repos, les points de données exceptionnels au bas du tableau peuvent être inattendus. Par exemple, pourquoi la file d'attente comportait-elle 13 threads exécutables ? Pour répondre à cette question, vous pourriez rédiger un script D affichant le contenu de la file d'attente lorsque celle-ci est longue. Ce problème est complexe car les activations D ne peuvent pas être itérées sur des structures de données, et donc pas sur l'ensemble de la file d'attente. Même si des activations D le pouvaient, vous devez éviter toute dépendance aux structures de données internes du noyau.

Pour ce type de script, vous devriez activer les sondes enqueue et dequeue et utiliser des spéculations et des ensembles associatifs. Lorsqu'un thread est placé en file d'attente, le script incrémente la longueur de la file d'attente et enregistre l'horodatage dans un ensemble associatif

affecté au thread. Vous ne pouvez pas utiliser une variable locale de thread dans ce cas car un thread peut être placé en file d'attente par un autre thread. Le script vérifie alors si la longueur de la file d'attente est supérieure à la valeur maximale. Si c'est le cas, le script entame une nouvelle spéculation, et enregistre l'horodatage et la nouvelle valeur maximale. Ensuite, lorsqu'un thread est retiré de la file d'attente, le script compare l'horotadage de mise en file d'attente avec celui de la longueur la plus importante : si le thread a été placé en file d'attente *avant* l'horodatage de la longueur la plus importante, le thread se trouvait en file d'attente au moment de l'enregistrement de la longueur la plus importante. Dans ce cas, le script suit de manière spéculative les informations du thread. Lorsque le noyau retire de la file d'attente le dernier thread placé au niveau de l'horodatage de la longueur la plus importante, le script fournit les données de spéculation. Ce script est illustré ci-dessous :

```
#pragma D option quiet
#pragma D option nspec=4
#pragma D option specsize=100k

int maxlen;
int spec[int];

sched:::enqueue
{
    this->len = ++qlen[this->cpu = args[2]->cpu_id];
    in[args[0]->pr_addr] = timestamp;
}

sched:::enqueue
/this->len > maxlen && spec[this->cpu]/
{
    /*
     * There is already a speculation for this CPU.  We just set a new
     * record, so we’ll discard the old one.
     */
    discard(spec[this->cpu]);
}

sched:::enqueue
/this->len > maxlen/
{
    /*
     * We have a winner.  Set the new maximum length and set the timestamp
     * of the longest length.
     */
    maxlen = this->len;
    longtime[this->cpu] = timestamp;

    /*
     * Now start a new speculation, and speculatively trace the length.
```

```
     */
    this->spec = spec[this->cpu] = speculation();
    speculate(this->spec);
    printf("Run queue of length %d:\n", this->len);
}

sched:::dequeue
/(this->in = in[args[0]->pr_addr]) &&
    this->in <= longtime[this->cpu = args[2]->cpu_id]/
{
    speculate(spec[this->cpu]);
    printf("  %d/%d (%s)\n",
        args[1]->pr_pid, args[0]->pr_lwpid,
        stringof(args[1]->pr_fname));
}

sched:::dequeue
/qlen[args[2]->cpu_id]/
{
    in[args[0]->pr_addr] = 0;
    this->len = --qlen[args[2]->cpu_id];
}

sched:::dequeue
/this->len == 0 && spec[this->cpu]/
{
    /*
     * We just processed the last thread that was enqueued at the time
     * of longest length; commit the speculation, which by now contains
     * each thread that was enqueued when the queue was longest.
     */
    commit(spec[this->cpu]);
    spec[this->cpu] = 0;
}
```

L'exécution du script ci-dessus sur le même système portable à un processeur donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./whoqueue.d
Run queue of length 3:
 0/0 (sched)
  0/0 (sched)
  101170/1 (dtrace)
Run queue of length 4:
  0/0 (sched)
  100356/1 (Xsun)
  100420/1 (xterm)
  101170/1 (dtrace)
```

```
Run queue of length 5:
  0/0 (sched)
  0/0 (sched)
  100356/1 (Xsun)
  100420/1 (xterm)
  101170/1 (dtrace)
Run queue of length 7:
  0/0 (sched)
  100221/18 (nscd)
  100221/17 (nscd)
  100221/16 (nscd)
  100221/13 (nscd)
  100221/14 (nscd)
  100221/15 (nscd)
Run queue of length 16:
  100821/1 (xterm)
  100768/1 (xterm)
  100365/1 (fvwm2)
  101118/1 (xterm)
  100577/1 (xterm)
  101170/1 (dtrace)
  101020/1 (xterm)
  101089/1 (xterm)
  100795/1 (xterm)
  100741/1 (xterm)
  100710/1 (xterm)
  101048/1 (xterm)
  100697/1 (MozillaFirebird-)
  100420/1 (xterm)
  100394/1 (xterm)
  100368/1 (xterm)
^C
```

La sortie indique que les longues files d'attente d'exécution sont dues à de nombreux processus `xterm` exécutables. Cette expérimentation coïncide avec un changement de bureau virtuel et les résultats sont donc probablement dus à une sorte de traitement d'événement X.

`sleep` et `wakeup`

Dans la section “enqueue et dequeue” à la page 287, le dernier exemple démontrait qu'une augmentation de la longueur de la file d'attente était due aux processus `xterm` exécutables. Une hypothèse est que les observations résultaient d'un changement de bureau virtuel. Vous pouvez utiliser la sonde `wakeup` pour étudier cette hypothèse en déterminant l'origine et le moment du réveil des processus `xterm`, tel qu'illustré dans l'exemple suivant :

```
#pragma D option quiet
```

```
dtrace:::BEGIN
{
    start = timestamp;
}

sched:::wakeup
/stringof(args[1]->pr_fname) == "xterm"/
{
    @[execname] = lquantize((timestamp - start) / 1000000000, 0, 10);
}

profile:::tick-1sec
/++x == 10/
{
    exit(0);
}
```

Pour étudier cette hypothèse, exécutez le script ci-dessus, patientez cinq secondes environ, puis changez une fois votre bureau virtuel. Si l'augmentation de processus `xterm` exécutables est due au changement de bureau virtuel, la sortie doit indiquer une activité de réveil à la cinquième seconde.

```
# dtrace -s ./xterm.d

  Xsun

           value  ------------- Distribution ------------- count
               4 |                                         0
               5 |@                                        1
               6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   32
               7 |                                         0
```

La sortie n'indique pas que le serveur X réveille des processus xterm, mis en cluster au moment du changement de bureau virtuel. Pour comprendre l'interaction entre le serveur X et les processus `xterm`, vous pourriez effectuer un groupement dans une pile utilisateur au moment où le serveur X déclenche la sonde `wakeup`.

La compréhension de la performance des systèmes client/serveur comme le système de fenêtrage X nécessite de connaître les clients pour lesquels le serveur travaille. La réponse à ce type de question n'est pas simple avec les outils d'analyse de performance conventionnels. Cependant, si vous disposez d'un modèle dans lequel un client envoie un message au serveur puis sommeille dans l'attente du traitement du serveur, vous pouvez utiliser la sonde `wakeup` pour déterminer le client à l'origine de la demande, tel qu'illustré dans l'exemple suivant :

```
self int last;

sched:::wakeup
```

```
/self->last && args[0]->pr_stype == SOBJ_CV/
{
    @[stringof(args[1]->pr_fname)] = sum(vtimestamp - self->last);
    self->last = 0;
}

sched:::wakeup
/execname == "Xsun" && self->last == 0/
{
    self->last = vtimestamp;
}
```

Exécuter le script ci-dessus engendre une sortie similaire à l'exemple suivant :

```
dtrace -s ./xwork.d
dtrace: script './xwork.d' matched 14 probes
^C
  xterm                                                         9522510
  soffice.bin                                                   9912594
  fvwm2                                                       100423123
  MozillaFirebird                                             312227077
  acroread                                                    345901577
```

Cette sortie révèle que la plupart des tâches `Xsun` sont exécutées pour les processus `acroread`, `MozillaFirebird` et, dans une moindre mesure, `fvwm2`. Notez que le script n'a abordé que les réveils à partir d'objets de synchronisation de variable de condition (`SOBJ_CV`). Tel que décrit dans le Tableau 25–4, les variables de condition représentent le type d'objet de synchronisation généralement utilisé pour une synchronisation dans un but autre que l'accès à une zone de données partagées. Dans le cas du serveur X, un client attend les données d'une file en sommeillant sur une variable de condition.

Vous pouvez également utiliser la sonde `sleep` avec la sonde `wakeup` pour connaître les applications bloquantes sur d'autres applications, ainsi que la durée, tel qu'illustré dans l'exemple suivant :

```
#pragma D option quiet

sched:::sleep
/!(curlwpsinfo->pr_flag & PR_ISSYS) && curlwpsinfo->pr_stype == SOBJ_CV/
{
    bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
    @[stringof(args[1]->pr_fname), execname] =
        quantize(timestamp - bedtime[args[0]->pr_addr]);
```

```
    bedtime[args[0]->pr_addr] = 0;
}

END
{
    printa("%s sleeping on %s:\n%@d\n", @);
}
```

La fin de la sortie de l'exécution de l'exemple de script pendant plusieurs secondes sur un système de bureau est similaire à l'exemple suivant :

```
# dtrace -s ./whofor.d
^C
...
 xterm sleeping on Xsun:

           value  ------------- Distribution ------------- count
          131072 |                                         0
          262144 |                                         12
          524288 |                                         2
         1048576 |                                         0
         2097152 |                                         5
         4194304 |@@@                                      45
         8388608 |                                         1
        16777216 |                                         9
        33554432 |@@@@@                                    83
        67108864 |@@@@@@@@@@@                              164
       134217728 |@@@@@@@@@@                               147
       268435456 |@@@@                                     56
       536870912 |@                                        17
      1073741824 |                                         9
      2147483648 |                                         1
      4294967296 |                                         3
      8589934592 |                                         1
     17179869184 |                                         0

fvwm2 sleeping on Xsun:

           value  ------------- Distribution ------------- count
           32768 |                                         0
           65536 |@@@@@@@@@@@@@@@@@@@@@@                   67
          131072 |@@@@@                                    16
          262144 |@@                                       6
          524288 |@                                        3
         1048576 |@@@@@                                    15
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         1
```

```
        16777216 |                                         0
        33554432 |                                         0
        67108864 |                                         1
       134217728 |                                         0
       268435456 |                                         0
       536870912 |                                         1
      1073741824 |                                         1
      2147483648 |                                         2
      4294967296 |                                         2
      8589934592 |                                         2
     17179869184 |                                         0
     34359738368 |                                         2
     68719476736 |                                         0

syslogd sleeping on syslogd:

           value  ------------- Distribution ------------- count
     17179869184 |                                         0
     34359738368 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
     68719476736 |                                         0

MozillaFirebird sleeping on MozillaFirebird:

           value  ------------- Distribution ------------- count
           65536 |                                         0
          131072 |                                         3
          262144 |@@                                       14
          524288 |                                         0
         1048576 |@@@                                      18
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         1
        16777216 |                                         0
        33554432 |                                         1
        67108864 |                                         3
       134217728 |@                                        7
       268435456 |@@@@@@@@@@                               53
       536870912 |@@@@@@@@@@@@@@                           78
      1073741824 |@@@@                                     25
      2147483648 |                                         0
      4294967296 |                                         0
      8589934592 |@                                        7
     17179869184 |                                         0
```

Vous pouvez souhaiter comprendre comment et pourquoi `MozillaFirebird` se bloque de lui-même. Vous pourriez modifier le script ci-dessus tel qu'illustré dans l'exemple suivant pour répondre à cette question :

```
#pragma D option quiet

sched:::sleep
/execname == "MozillaFirebird" && curlwpsinfo->pr_stype == SOBJ_CV/
{
    bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/execname == "MozillaFirebird" && bedtime[args[0]->pr_addr]/
{
    @[args[1]->pr_pid, args[0]->pr_lwpid, pid, curlwpsinfo->pr_lwpid] =
        quantize(timestamp - bedtime[args[0]->pr_addr]);
    bedtime[args[0]->pr_addr] = 0;
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
    bedtime[args[0]->pr_addr] = 0;
}

END
{
    printa("%d/%d sleeping on %d/%d:\n%@d\n", @);
}
```

L'exécution du script modifié pendant plusieurs secondes donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./firebird.d
^C

 100459/1 sleeping on 100459/13:

           value  ------------- Distribution ------------ count
          262144 |                                         0
          524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
         1048576 |                                         0

100459/13 sleeping on 100459/1:

           value  ------------- Distribution ------------ count
        16777216 |                                         0
        33554432 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
        67108864 |                                         0

100459/1 sleeping on 100459/2:
```

```
           value  ------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@                                     5
           65536 |@                                        2
          131072 |@@@@@                                    6
          262144 |                                         1
          524288 |@                                        2
         1048576 |                                         0
         2097152 |@@                                       3
         4194304 |@@@@                                     5
         8388608 |@@@@@@@@                                 9
        16777216 |@@@@@                                    6
        33554432 |@@                                       3
        67108864 |                                         0

100459/1 sleeping on 100459/5:

           value  ------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@@                                    12
           65536 |@@                                       5
          131072 |@@@@@@                                   15
          262144 |                                         1
          524288 |                                         1
         1048576 |                                         2
         2097152 |@                                        4
         4194304 |@@@@@                                    13
         8388608 |@@@                                      8
        16777216 |@@@@@                                    13
        33554432 |@@                                       6
        67108864 |@@                                       5
       134217728 |@                                        4
       268435456 |                                         0
       536870912 |                                         1
      1073741824 |                                         0

100459/2 sleeping on 100459/1:

           value  ------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@@@@@@@@@@@                           11
           65536 |                                         0
          131072 |@@                                       2
          262144 |                                         0
          524288 |                                         0
         1048576 |@@@@                                     3
         2097152 |@                                        1
```

```
         4194304 |@@                                       2
         8388608 |@@                                       2
        16777216 |@                                        1
        33554432 |@@@@@@                                   5
        67108864 |                                         0
       134217728 |                                        0
       268435456 |                                        0
       536870912 |@                                        1
      1073741824 |@                                        1
      2147483648 |@                                        1
      4294967296 |                                         0

100459/5 sleeping on 100459/1:

           value  ------------- Distribution ----------- count
           16384 |                                         0
           32768 |                                         1
           65536 |                                         2
          131072 |                                         4
          262144 |                                         7
          524288 |                                         1
         1048576 |                                         5
         2097152 |                                         10
         4194304 |@@@@@@                                   77
         8388608 |@@@@@@@@@@@@@@@@@@@@@@@                  270
        16777216 |@@@                                      43
        33554432 |@                                        20
        67108864 |@                                        14
       134217728 |                                         5
       268435456 |                                         2
       536870912 |                                         1
      1073741824 |                                         0
```

Vous pouvez également utiliser les sondes `sleep` et `wakeup` pour connaître la performance de serveurs à portes comme le démon de cache de service de nom, tel qu'illustré dans l'exemple suivant :

```
sched:::sleep
/curlwpsinfo->pr_stype == SOBJ_SHUTTLE/
{
    bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/execname == "nscd" && bedtime[args[0]->pr_addr]/
{
    @[stringof(curpsinfo->pr_fname), stringof(args[1]->pr_fname)] =
        quantize(timestamp - bedtime[args[0]->pr_addr]);
```

```
    bedtime[args[0]->pr_addr] = 0;
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
    bedtime[args[0]->pr_addr] = 0;
}
```

La fin de la sortie de l'exécution du script ci-dessus sur un serveur de messagerie d'envergure est similaire à l'exemple suivant :

```
imapd
           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |                                         2
           65536 |@@@@@@@@@@@@@@@@@                        57
          131072 |@@@@@@@@@@@                              37
          262144 |                                         3
          524288 |@@@                                      11
         1048576 |@@@                                      10
         2097152 |@@                                       9
         4194304 |                                         1
         8388608 |                                         0

  mountd
           value  -------------- Distribution ------------ count
           65536 |                                         0
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            49
          262144 |@@@                                      6
          524288 |                                         1
         1048576 |                                         0
         2097152 |                                         0
         4194304 |@@@@                                     7
         8388608 |@                                        3
        16777216 |                                         0

  sendmail
           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@                                        18
           65536 |@@@@@@@@@@@@@@@@@                        205
          131072 |@@@@@@@@@@@@@                            154
          262144 |@                                        23
          524288 |                                         5
         1048576 |@@@@                                     50
         2097152 |                                         7
         4194304 |                                         5
```

```
         8388608 |                                         2
        16777216 |                                         0

  automountd
           value  ------------- Distribution ------------ count
           32768 |                                         0
           65536 |@@@@@@@@@@                               22
          131072 |@@@@@@@@@@@@@@@@@@@@@@@                  51
          262144 |@@                                       6
          524288 |                                         1
         1048576 |                                         0
         2097152 |                                         2
         4194304 |                                         2
         8388608 |                                         1
        16777216 |                                         1
        33554432 |                                         1
        67108864 |                                         0
       134217728 |                                         0
       268435456 |                                         1
       536870912 |                                         0
```

Vous pouvez être intéressé par les points de données inhabituels pour `automountd` ou le point de données persistent de plus d'une milliseconde pour `sendmail`. Vous pouvez ajouter d'autres prédicats au script ci-dessus pour comprendre les causes de résultats exceptionnels ou anormaux.

preempt, remain-cpu

Solaris étant un système préemptif, les threads à priorité supérieure déplacent ceux à priorité inférieure. La préemption peut entraîner une bulle de latence importante dans un thread à priorité inférieure. Vous pouvez ainsi souhaiter connaître les threads déplacés, et par quels threads. L'exemple suivant illustre comment utiliser les sondes `preempt` et `remain-cpu` pour afficher ces informations :

```
#pragma D option quiet

sched:::preempt
{
    self->preempt = 1;
}

sched:::remain-cpu
/self->preempt/
{
    self->preempt = 0;
}
```

```
sched:::off-cpu
/self->preempt/
{
    /*
     * If we were told to preempt ourselves, see who we ended up giving
     * the CPU to.
     */
    @[stringof(args[1]->pr_fname), args[0]->pr_pri, execname,
        curlwpsinfo->pr_pri] = count();
    self->preempt = 0;
}

END
{
    printf("%30s %3s %30s %3s %5s\n", "PREEMPTOR", "PRI",
        "PREEMPTED", "PRI", "#");
    printa("%30s %3d %30s %3d %5@d\n", @);
}
```

L'exécution du script ci-dessus pendant plusieurs secondes sur un système de bureau donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./whopreempt.d
^C
                     PREEMPTOR PRI                      PREEMPTED PRI     #
                         sched  60                           Xsun  53     1
                         xterm  59                           Xsun  53     1
               MozillaFirebird  57                           Xsun  53     1
                        mpstat 100                          fvwm2  59     1
                         sched  99                MozillaFirebird  57     1
                         sched  60                         dtrace  30     1
                        mpstat 100                           Xsun  59     2
                         sched  60                           Xsun  54     2
                         sched  99                          sched  60     2
                         fvwm2  59                           Xsun  44     2
                         sched  99                           Xsun  44     2
                         sched  60                          xterm  59     2
                         sched  99                           Xsun  53     2
                         sched  99                           Xsun  54     3
                         sched  60                          fvwm2  59     3
                         sched  60                           Xsun  59     3
                         sched  99                           Xsun  59     4
                         fvwm2  59                           Xsun  54     8
                         fvwm2  59                           Xsun  53     9
                          Xsun  59                MozillaFirebird  57    10
                         sched  60                MozillaFirebird  57    14
               MozillaFirebird  57                           Xsun  44    16
               MozillaFirebird  57                           Xsun  54    18
```

change-pri

La préemption est basée sur des priorités. Vous pouvez ainsi souhaiter observer les changements de priorité dans le temps. L'exemple suivant utilise la sonde `change-pri` pour afficher ces informations :

```
sched:::change-pri
{
    @[stringof(args[0]->pr_clname)] =
        lquantize(args[2] - args[0]->pr_pri, -50, 50, 5);
}
```

L'exemple de script capture le degré auquel la priorité est augmentée ou réduite, et regroupe par classe de planification. Exécuter le script ci-dessus engendre une sortie similaire à l'exemple suivant :

```
# dtrace -s ./pri.d
dtrace: script './pri.d' matched 10 probes
^C
 IA
           value  ------------- Distribution ------------ count
           < -50 |                                        20
             -50 |@                                       38
             -45 |                                        4
             -40 |                                        13
             -35 |                                        12
             -30 |                                        18
             -25 |                                        18
             -20 |                                        23
             -15 |                                        6
             -10 |@@@@@@@@                                201
              -5 |@@@@@@                                  160
               0 |@@@@@                                   138
               5 |@                                       47
              10 |@@                                      66
              15 |@                                       36
              20 |@                                       26
              25 |@                                       28
              30 |                                        18
              35 |                                        22
              40 |                                        8
              45 |                                        11
           >= 50 |@                                       34

  TS
           value  ------------- Distribution ------------ count
             -15 |                                        0
             -10 |@                                       1
```

```
           -5 |@@@@@@@@@@@@                              7
            0 |@@@@@@@@@@@@@@@@@@@@@                     12
            5 |                                          0
           10 |@@@@@                                     3
           15 |                                          0
```

La sortie illustre la manipulation de priorité de la classe de planification Interactive (IA). Plutôt que d'observer la *manipulation* de priorité, vous pouvez souhaiter observer les *valeurs* de priorité d'un processus et d'un thread particuliers dans le temps. Le script suivant utilise la sonde `change-pri` pour afficher ces informations :

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
}

sched:::change-pri
/args[1]->pr_pid == $1 && args[0]->pr_lwpid == $2/
{
    printf("%d %d\n", timestamp - start, args[2]);
}

tick-1sec
/++n == 5/
{
    exit(0);
}
```

Pour observer le changement de priorité dans le temps, entrez la commande suivante dans une fenêtre :

```
$ echo $$
139208
$ while true ; do let i=0 ; done
```

Dans une autre fenêtre, exécutez le script et redirigez la sortie vers un fichier :

```
# dtrace -s ./pritime.d 139208 1 > /tmp/pritime.out
#
```

Vous pouvez utiliser le fichier `/tmp/pritime.out` généré ci-dessus en entrée pour tracer le logiciel afin d'afficher sous forme graphique la priorité dans le temps. `gnuplot` est un package de traçage gratuit inclus sur le CD d'accompagnement du freeware Solaris. Par défaut, `gnuplot` est installé dans `/opt/sfw/bin`.

tick

Solaris utilise la *comptabilisation de CPU sur top*, dans laquelle une interruption d'horloge système se déclenche à intervalle fixe et attribue une utilisation de la CPU aux threads et aux processus exécutés au moment du top. L'exemple suivant illustre comment utiliser la sonde `tick` pour observer cette attribution :

```
# dtrace -n sched:::tick'{@[stringof(args[1]->pr_fname)] = count()}'
^C
  arch                                                              1
  sh                                                                1
  sed                                                               1
  echo                                                              1
  ls                                                                1
  FvwmAuto                                                          1
  pwd                                                               1
  awk                                                               2
  basename                                                          2
  expr                                                              2
  resize                                                            2
  tput                                                              2
  uname                                                             2
  fsflush                                                           2
  dirname                                                           4
  vim                                                               9
  fvwm2                                                            10
  ksh                                                              19
  xterm                                                            21
  Xsun                                                             93
  MozillaFirebird                                                 260
```

La fréquence de l'horloge système varie d'un système d'exploitation à un autre, mais est généralement comprise entre 25 et 1 024 hertz. La fréquence de l'horloge système Solaris peut être réglée et est par défaut de 100 hertz.

La sonde `tick` ne se déclenche que si l'horloge système détecte un thread exécutable. Pour utiliser la sonde `tick` afin d'observer la fréquence de l'horloge système, vous devez disposer d'un thread exécutable en permanence. Dans une fenêtre, créez un shell de bouclage tel qu'illustré dans l'exemple suivant :

```
$ while true ; do let i=0 ; done
```

Dans une autre fenêtre, exécutez le script suivant :

```
uint64_t last[int];

sched:::tick
```

```
/last[cpu]/
{
    @[cpu] = min(timestamp - last[cpu]);
}

sched:::tick
{
    last[cpu] = timestamp;
}

# dtrace -s ./ticktime.d
dtrace: script './ticktime.d' matched 2 probes
^C

  0          9883789
```

L'intervalle minimum est de 9,8 millisecondes, qui indique que la fréquence de top d'horloge est par défaut de 10 millisecondes (100 hertz). Le minimum observé est quelque peu inférieur à 10 millisecondes en raison d'une gigue.

Une déficience de la comptabilisation sur top est telle que l'horloge système exécutant la comptabilisation est également souvent chargée de répartir les activités de planification temporelle. Par conséquent, si un thread travaille à chaque top (c'est-à-dire toutes les 10 millisecondes), le système sur- ou sous-comptabilisera le thread, si la comptabilisation est effectuée avant ou après une activité de planification de répartition temporelle. Dans Solaris, la comptabilisation est effectuée avant la répartition temporelle. Par conséquent, le système sous-comptabilisera les threads exécutés à intervalle régulier. Si de tels threads sont exécutés pendant moins longtemps que l'intervalle de top d'horloge, ils peuvent en réalité “se cacher” derrière le top. L'exemple suivant illustre le degré auquel le système comporte de tels threads :

```
sched:::tick,
sched:::enqueue
{
    @[probename] = lquantize((timestamp / 1000000) % 10, 0, 10);
}
```

La sortie de l'exemple de script est deux distributions du décalage de milliseconde dans un intervalle de dix millisecondes, une pour la sonde `tick` et l'autre pour `enqueue` :

```
# dtrace -s ./tick.d
dtrace: script './tick.d' matched 4 probes
^C
  tick
           value  -------------- Distribution ------------ count
               6 |                                         0
               7 |@                                        3
               8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   79
```

```
              9 |                                        0

  enqueue
           value  ------------- Distribution ------------ count
             < 0 |                                        0
               0 |@@                                      267
               1 |@@                                      300
               2 |@@                                      259
               3 |@@                                      291
               4 |@@@                                     360
               5 |@@                                      305
               6 |@@                                      295
               7 |@@@@                                    522
               8 |@@@@@@@@@@@@                            1315
               9 |@@@                                     337
```

L'histogramme obtenu nommé `tick` indique que le top d'horloge se déclenche selon un décalage de 8 millisecondes. Si la planification n'était pas associée au top d'horloge, la sortie de `enqueue` serait répartie de manière égale pendant l'intervalle de dix millisecondes. La sortie illustre cependant un pic au même décalage de 8 millisecondes, indiquant qu'au moins certains threads du système *sont* planifiés sur une base temporelle.

Stabilité

Le fournisseur `sched` utilise un mécanisme de stabilité DTrace pour décrire ses stabilités, tel qu'illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	En cours d'évolution	En cours d'évolution	ISA

CHAPITRE 27

Fournisseur io

Le fournisseur io propose des sondes liées à l'entrée et la sortie de disque. Le fournisseur io permet d'étudier rapidement le comportement observé via des outils de contrôle d'E/S comme iostat(1M). Par exemple, grâce au fournisseur io, vous pouvez connaître les E/S par périphérique, type d'E/S, taille d'E/S, processus, nom d'application, nom de fichier ou par décalage de fichier.

Sondes

Les sondes io sont décrites dans Tableau 27–1.

TABLEAU 27–1 Sondes io

Sonder	Description
start	Sonde se déclenchant lorsqu'une demande d'E/S va être formulée sur un périphérique ou un serveur NFS. bufinfo_t correspondant à la demande d'E/S est indiqué par args[0]. devinfo_t du périphérique d'où les E/S proviennent est indiqué par args[1]. fileinfo_t du fichier correspondant à la demande d'E/S est indiqué par args[2]. Notez que la disponibilité des informations de fichier dépend du système de fichiers formulant la demande d'E/S. Pour plus d'informations, reportez-vous à la section “fileinfo_t” à la page 313.
done	Sonde se déclenchant après la formulation d'une demande d'E/S. bufinfo_t correspondant à la demande d'E/S est indiqué par args[0]. La sonde done se déclenche une fois les E/S terminées, mais avant la fin du traitement sur le tampon. Par conséquent, B_DONE n'est *pas* défini dans b_flags au moment du déclenchement de la sonde done. devinfo_t du périphérique d'où les E/S proviennent est indiqué par args[1]. fileinfo_t du fichier correspondant à la demande d'E/S est indiqué par args[2].

TABLEAU 27–1 Sondes io *(Suite)*

Sonder	Description
wait-start	Sonde se déclenchant immédiatement avant qu'un thread commence à attendre la fin en attente d'une demande d'E/S particulière. La structure buf(9S) correspondant à la demande d'E/S pour laquelle le thread attend est indiquée par args[0]. devinfo_t du périphérique d'où les E/S proviennent est indiqué par args[1] . fileinfo_t du fichier correspondant à la demande d'E/S est indiqué par args[2]. Parfois, après le déclenchement de la sonde wait-start, la sonde wait-done est déclenchée dans le même thread.
wait-done	Sonde se déclenchant lorsqu'un thread doit attendre la fin d'une demande d'E/S donnée. bufinfo_t correspondant à la demande d'E/S pour laquelle le thread attend est indiqué par args[0]. devinfo_t du périphérique d'où les E/S proviennent est indiqué par args[1]. fileinfo_t du fichier correspondant à la demande d'E/S est indiqué par args[2]. La sonde wait-done ne se déclenche qu'après le déclenchement de la sonde wait-start dans le même thread.

Notez que les sondes io se déclenchent pour toutes les demandes d'E/S vers des périphériques, et pour toutes les demandes de lecture et d'écriture de fichier vers un serveur NFS. Des demandes de métadonnées à partir d'un serveur NFS, par exemple, ne déclenchent *pas* de sondes io en raison d'une demande readdir(3C).

Arguments

Les types d'argument des sondes io sont répertoriés dans Tableau 27–2. Les arguments sont décrits dans Tableau 27–1.

TABLEAU 27–2 Arguments de sonde io

Sonder	args[0]	args[1]	args[2]
start	struct buf *	devinfo_t *	fileinfo_t *
done	struct buf *	devinfo_t *	fileinfo_t *
wait-start	struct buf *	devinfo_t *	fileinfo_t *
wait-done	struct buf *	devinfo_t *	fileinfo_t *

Chaque sonde io comprend des arguments constitués d'un pointeur sur une structure buf(9S), d'un pointeur sur devinfo_t et d'un pointeur sur fileinfo_t. Ces structures sont décrites plus en détails dans cette section.

Structure bufinfo_t

La structure bufinfo_t représente l'abstraction décrivant une demande d'E/S. Le tampon correspondant à une demande d'E/S est indiqué par args[0] des sondes start, done, wait-start et wait-done. La définition de la structure bufinfo_t est la suivante :

```
typedef struct bufinfo {
    int b_flags;                    /* flags */
    size_t b_bcount;                /* number of bytes */
    caddr_t b_addr;                 /* buffer address */
    uint64_t b_blkno;               /* expanded block # on device */
    uint64_t b_lblkno;              /* block # on device */
    size_t b_resid;                 /* # of bytes not transferred */
    size_t b_bufsize;               /* size of allocated buffer */
    caddr_t b_iodone;               /* I/O completion routine */
    dev_t b_edev;                   /* extended device */
 } bufinfo_t;
```

L'élément b_flags indique l'état du tampon d'E/S et comprend des valeurs d'état bitwise ou autres. Les valeurs d'état valides sont indiquées dans Tableau 27–3.

TABLEAU 27–3 Valeurs de b_flags

B_DONE	Indique que le transfert de données est terminé.
B_ERROR	Indique une erreur de transfert d'E/S. Défini avec le champ b_error.
B_PAGEIO	Indique que le tampon est en cours d'utilisation dans une demande d'E/S chargée. Pour plus d'informations, reportez-vous à la description du champ b_addr.
B_PHYS	Indique que le tampon est en cours d'utilisation pour des E/S physiques (directes) d'une zone de données utilisateur.
B_READ	Indique que les données doivent être lues du périphérique à la mémoire principale.
B_WRITE	Indique que les données doivent être transférées de la mémoire principale au périphérique.
B_ASYNC	La demande d'E/S est asynchrone et sera mise en attente. Les sondes wait-start et wait-done ne se déclenchent pas pour des demandes d'E/S asynchrones. Notez que B_ASYNC peut ne pas être défini pour certaines E/S conçues pour être asynchrones : le sous-système d'E/S asynchrone peut mettre en œuvre la demande asynchrone en disposant d'un thread distinct pour exécuter une opération d'E/S synchrone.

Le champ b_bcount représente le nombre d'octets à transférer dans la demande d'E/S.

Le champ b_addr représente l'adresse virtuelle de la demande d'E/S, à moins que B_PAGEIO soit défini. L'adresse est une adresse virtuelle de noyau à moins que B_PHYS soit défini, auquel cas il s'agit d'une adresse virtuelle d'utilisateur. Si B_PAGEIO est défini, le champ b_addr contient des données privées de noyau. Un de B_PHYS et de B_PAGEIO peut être défini, sinon aucun indicateur n'est défini.

Le champ b_lblkno identifie le bloc logique du périphérique auquel accéder. Le mappage d'un bloc logique avec un bloc physique (comme le cylindre, la piste, etc.) est défini par le périphérique.

Le champ b_resid est défini sur le nombre d'octets non transférés en raison d'une erreur.

Le champ b_bufsize contient la taille du tampon affecté.

Le champ b_iodone identifie une routine spécifique dans le noyau appelée à la fin de l'E/S.

Le champ b_error peut contenir un code d'erreur renvoyé par le périphérique en cas d'erreur d'E/S. b_error est défini avec le bit B_ERROR défini dans le membre b_flags.

Le champ b_edev contient les numéros supérieur et inférieur de périphériques du périphérique accessible. Les clients peuvent utiliser les sous-routines D getmajor() et getminor() pour extraire les numéros supérieur et inférieur de périphériques du champ b_edev.

devinfo_t

La structure devinfo_t fournit des informations sur un périphérique. La structure devinfo_t correspondant au périphérique de destination d'E/S est indiquée par args[1] des sondes start, done, wait-start et wait-done. Les éléments de devinfo_t sont les suivants :

```
typedef struct devinfo {
    int dev_major;                  /* major number */
    int dev_minor;                  /* minor number */
    int dev_instance;               /* instance number */
    string dev_name;                /* name of device */
    string dev_statname;            /* name of device + instance/minor */
    string dev_pathname;            /* pathname of device */
} devinfo_t;
```

Le champ dev_major représente le numéro supérieur du périphérique. Pour plus d'informations, reportez-vous à getmajor(9F).

Le champ dev_minor représente le numéro inférieur du périphérique. Pour plus d'informations, reportez-vous à getminor(9F).

Le champ dev_instance représente le numéro d'instance du périphérique. L'instance d'un périphérique est différente du numéro inférieur. Le numéro inférieur est une abstraction gérée par le pilote du périphérique. Le numéro d'instance est une propriété du nœud de périphérique. Vous pouvez afficher des numéros d'instance de nœud de périphérique via prtconf(1M).

Le champ dev_name représente le nom du pilote de périphérique qui gère le périphérique. Vous pouvez afficher les noms des pilotes de périphérique via l'option -D pour prtconf(1M).

Le champ dev_statname représente le nom du périphérique tel qu'indiqué par iostat(1M). Ce nom correspond également au nom d'une statistique de noyau tel qu'indiqué par kstat(1M). Ce champ est conçu pour qu'une sortie iostat ou kstat anormale puisse être rapidement mise en corrélation avec l'activité d'E/S réelle.

Le champ dev_pathname représente le chemin d'accès complet au périphérique. Ce chemin peut être spécifié en tant qu'argument pour prtconf(1M) afin d'obtenir des informations détaillées sur le périphérique. Le chemin spécifié par dev_pathname inclut des composants indiquant le nœud du périphérique, le numéro d'instance et le nœud inférieur. Ces trois éléments ne sont cependant pas nécessairement indiqués dans le nom de statistique. Pour certains périphériques, le nom de statistique est composé du nom du périphérique et du numéro d'instance. Pour d'autres, il est composé du nom du périphérique et du numéro de nœud inférieur. Par conséquent, deux périphériques portant le même dev_statname peuvent avoir un dev_pathname différent .

fileinfo_t

La structure fileinfo_t fournit des informations sur un fichier. Le fichier auquel correspond des E/S est indiqué par args[0] des sondes start, done, wait-start et wait-done. La présence d'informations sur un fichier est liée au système de fichiers fournissant ces informations lors de la répartition de demandes d'E/S. Certains systèmes de fichiers, notamment de tiers, peuvent ne pas fournir ces informations. De même, des demandes d'E/S peuvent provenir d'un système de fichiers pour lequel il n'existe aucune information de fichier. Par exemple, les E/S pour des métadonnées de système de fichiers ne seront associées à aucun fichier. Enfin, certains systèmes de fichiers hautement optimisés peuvent regrouper des E/S de fichiers distincts dans une seule demande d'E/S. Dans ce cas, le système de fichiers peut fournir les informations relatives au fichier représentant la plupart des E/S ou au fichier représentant *certaines* E/S. Le système de fichiers peut également ne fournir aucune information de fichier.

La définition de la structure fileinfo_t est la suivante :

```
typedef struct fileinfo {
    string fi_name;                 /* name (basename of fi_pathname) */
    string fi_dirname;              /* directory (dirname of fi_pathname) */
    string fi_pathname;             /* full pathname */
    offset_t fi_offset;             /* offset within file */
    string fi_fs;                   /* filesystem */
    string fi_mount;                /* mount point of file system */
} fileinfo_t;
```

Le champ fi_name contient le nom du fichier mais aucun composant de répertoire. Si aucune information de fichier n'est associée à une E/S, le champ fi_name sera défini sur la chaîne

<none>. Dans quelques rares cas, le nom de chemin associé à un fichier peut être inconnu. Dans ce cas, le champ `fi_name` sera défini sur la chaîne <unknown>.

Le champ `fi_dirname` ne contient *que* le composant de répertoire du nom de fichier. Comme `fi_name`, cette chaîne peut être définie sur <none> s'il n'existe aucune information de fichier ou sur <unknown> si le nom du chemin associé au fichier est inconnu.

Le champ `fi_pathname` contient le nom du chemin complet au fichier. Comme `fi_name`, cette chaîne peut être définie sur <none> s'il n'existe aucune information de fichier ou sur <unknown> si le nom du chemin associé au fichier est inconnu.

Le champ `fi_offset` contient le décalage dans le fichier ou -1 s'il n'existe aucune information de fichier ou si le décalage n'est pas spécifié par le système de fichiers.

Exemples

L'exemple de script suivant indique des informations pertinentes sur chaque E/S telle que renvoyée :

```
#pragma D option quiet

BEGIN
{
    printf("%10s %58s %2s\n", "DEVICE", "FILE", "RW");
}

io:::start
{
    printf("%10s %58s %2s\n", args[1]->dev_statname,
        args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W");
}
```

La sortie de l'exemple lors d'un démarrage à froid d'Acrobat Reader sur un système portable x86 est similaire à l'exemple suivant :

```
# dtrace -s ./iosnoop.d
    DEVICE                                                       FILE RW
     cmdk0                                 /opt/Acrobat4/bin/acroread  R
     cmdk0                                 /opt/Acrobat4/bin/acroread  R
     cmdk0                                                  <unknown>  R
     cmdk0                           /opt/Acrobat4/Reader/AcroVersion  R
     cmdk0                                                  <unknown>  R
     cmdk0                                                  <unknown>  R
     cmdk0                                                     <none>  R
     cmdk0                                                  <unknown>  R
     cmdk0                                                     <none>  R
```

```
cmdk0                  /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                  /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                  /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                                                      <none>  R
cmdk0                                                   <unknown>  R
cmdk0                                                   <unknown>  R
cmdk0                                                   <unknown>  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                                      <none>  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                                   <unknown>  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0                                                      <none>  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0              /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                                   <unknown>  R
cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R
cmdk0                                                      <none>  R
cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R
cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R
  ...
```

Les entrées <none> de la sortie indiquent que l'E/S ne correspond pas aux données d'un fichier particulier : ces E/S sont dues à des métadonnées d'une quelconque forme. Les entrées <unknown> de la sortie indiquent que le nom du chemin d'accès au fichier est inconnu. Cette situation est relativement rare.

Vous pourriez appliquer un exemple de script un peu plus complexe en utilisant un ensemble associatif pour suivre la durée de chaque E/S, tel qu'illustré dans l'exemple suivant :

```
#pragma D option quiet

BEGIN
```

```
{
    printf("%10s %58s %2s %7s\n", "DEVICE", "FILE", "RW", "MS");
}

io:::start
{
    start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
}

io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
    this->elapsed = timestamp - start[args[0]->b_edev, args[0]->b_blkno];
    printf("%10s %58s %2s %3d.%03d\n", args[1]->dev_statname,
        args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W",
        this->elapsed / 10000000, (this->elapsed / 1000) % 1000);
    start[args[0]->b_edev, args[0]->b_blkno] = 0;
}
```

La sortie de l'exemple ci-dessus pendant la connexion à chaud d'un périphérique de stockage USB dans un système portable x86 au repos est illustré dans l'exemple suivant :

```
# dtrace -s ./iotime.d
    DEVICE                                                 FILE RW      MS
     cmdk0                                 /kernel/drv/scsa2usb  R  24.781
     cmdk0                                 /kernel/drv/scsa2usb  R  25.208
     cmdk0                                    /var/adm/messages  W  25.981
     cmdk0                                 /kernel/drv/scsa2usb  R   5.448
     cmdk0                                               <none>  W   4.172
     cmdk0                                 /kernel/drv/scsa2usb  R   2.620
     cmdk0                                    /var/adm/messages  W   0.252
     cmdk0                                            <unknown>  R   3.213
     cmdk0                                               <none>  W   3.011
     cmdk0                                            <unknown>  R   2.197
     cmdk0                                    /var/adm/messages  W   2.680
     cmdk0                                               <none>  W   0.436
     cmdk0                                    /var/adm/messages  W   0.542
     cmdk0                                               <none>  W   0.339
     cmdk0                                    /var/adm/messages  W   0.414
     cmdk0                                               <none>  W   0.344
     cmdk0                                    /var/adm/messages  W   0.361
     cmdk0                                               <none>  W   0.315
     cmdk0                                    /var/adm/messages  W   0.421
     cmdk0                                               <none>  W   0.349
     cmdk0                                               <none>  R   1.524
     cmdk0                                            <unknown>  R   3.648
     cmdk0                                 /usr/lib/librcm.so.1  R   2.553
     cmdk0                                 /usr/lib/librcm.so.1  R   1.332
```

```
cmdk0                            /usr/lib/librcm.so.1  R   0.222
cmdk0                            /usr/lib/librcm.so.1  R   0.228
cmdk0                            /usr/lib/librcm.so.1  R   0.927
cmdk0                                          <none>  R   1.189
  ...
cmdk0                       /usr/lib/devfsadm/linkmod  R   1.110
cmdk0       /usr/lib/devfsadm/linkmod/SUNW_audio_link.so  R   1.763
cmdk0       /usr/lib/devfsadm/linkmod/SUNW_audio_link.so  R   0.161
cmdk0         /usr/lib/devfsadm/linkmod/SUNW_cfg_link.so  R   0.819
cmdk0         /usr/lib/devfsadm/linkmod/SUNW_cfg_link.so  R   0.168
cmdk0        /usr/lib/devfsadm/linkmod/SUNW_disk_link.so  R   0.886
cmdk0        /usr/lib/devfsadm/linkmod/SUNW_disk_link.so  R   0.185
cmdk0      /usr/lib/devfsadm/linkmod/SUNW_fssnap_link.so  R   0.778
cmdk0      /usr/lib/devfsadm/linkmod/SUNW_fssnap_link.so  R   0.166
cmdk0        /usr/lib/devfsadm/linkmod/SUNW_lofi_link.so  R   1.634
cmdk0        /usr/lib/devfsadm/linkmod/SUNW_lofi_link.so  R   0.163
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_md_link.so  R   0.477
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_md_link.so  R   0.161
cmdk0        /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.198
cmdk0        /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.168
cmdk0        /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.247
cmdk0   /usr/lib/devfsadm/linkmod/SUNW_misc_link_i386.so  R   1.735
  ...
```

Plusieurs observations sur le mécanisme du système en fonction de cette sortie sont possibles. Premièrement, notez la durée prolongée pour exécuter les premières E/S, qui est d'environ 25 millisecondes pour chacune. Cette durée peut être due au périphérique `cmdk0` dont l'alimentation est gérée sur le portable. Deuxièmement, observez l'E/S due au chargement du pilote `scsa2usb`(7D) pour traiter le périphérique de stockage de masse USB. Troisièmement, notez les messages dans `/var/adm/messages` renvoyés pour le périphérique. Enfin, observez la lecture des générateurs de liens de périphérique (fichiers se terminant par `link.so`) , qui traitent probablement le nouveau périphérique.

Le fournisseur `io` permet de comprendre de manière approfondie la sortie de `iostat`(1M). Supposons l'observation d'une sortie `iostat` similaire à l'exemple suivant :

```
extended device statistics
device       r/s    w/s   kr/s   kw/s wait actv  svc_t  %w  %b
cmdk0        8.0    0.0  399.8    0.0  0.0  0.0    0.8   0   1
sd0          0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd2          0.0  109.0    0.0  435.9  0.0  1.0    8.9   0  97
nfs1         0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
nfs2         0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
```

Vous pouvez utiliser le script `iotime.d` pour afficher ces E/S lorsqu'elles se produisent, tel qu'illustré dans l'exemple suivant :

```
      DEVICE                                            FILE RW      MS
         sd2                                /mnt/archives.tar  W   0.856
         sd2                                /mnt/archives.tar  W   0.729
         sd2                                /mnt/archives.tar  W   0.890
         sd2                                /mnt/archives.tar  W   0.759
         sd2                                /mnt/archives.tar  W   0.884
         sd2                                /mnt/archives.tar  W   0.746
         sd2                                /mnt/archives.tar  W   0.891
         sd2                                /mnt/archives.tar  W   0.760
         sd2                                /mnt/archives.tar  W   0.889
       cmdk0                        /export/archives/archives.tar  R   0.827
         sd2                                /mnt/archives.tar  W   0.537
         sd2                                /mnt/archives.tar  W   0.887
         sd2                                /mnt/archives.tar  W   0.763
         sd2                                /mnt/archives.tar  W   0.878
         sd2                                /mnt/archives.tar  W   0.751
         sd2                                /mnt/archives.tar  W   0.884
         sd2                                /mnt/archives.tar  W   0.760
         sd2                                /mnt/archives.tar  W   3.994
         sd2                                /mnt/archives.tar  W   0.653
         sd2                                /mnt/archives.tar  W   0.896
         sd2                                /mnt/archives.tar  W   0.975
         sd2                                /mnt/archives.tar  W   1.405
         sd2                                /mnt/archives.tar  W   0.724
         sd2                                /mnt/archives.tar  W   1.841
       cmdk0                        /export/archives/archives.tar  R   0.549
         sd2                                /mnt/archives.tar  W   0.543
         sd2                                /mnt/archives.tar  W   0.863
         sd2                                /mnt/archives.tar  W   0.734
         sd2                                /mnt/archives.tar  W   0.859
         sd2                                /mnt/archives.tar  W   0.754
         sd2                                /mnt/archives.tar  W   0.914
         sd2                                /mnt/archives.tar  W   0.751
         sd2                                /mnt/archives.tar  W   0.902
         sd2                                /mnt/archives.tar  W   0.735
         sd2                                /mnt/archives.tar  W   0.908
         sd2                                /mnt/archives.tar  W   0.753
```

Cette sortie semble indiquer que le fichier `archives.tar` est lu depuis `cmdk0` (dans `/export/archives`), et écrit sur le périphérique `sd2` (dans `/mnt`). La présence de deux fichiers nommés `archives.tar` en cours d'utilisation parallèle et distincte semble improbable. Pour en savoir plus, vous pouvez regrouper l'ID de périphérique, d'application, de processus et les octets transférés, tel qu'illustré dans l'exemple suivant :

```
#pragma D option quiet

io:::start
{
```

```
    @[args[1]->dev_statname, execname, pid] = sum(args[0]->b_bcount);
}

END
{
    printf("%10s %20s %10s %15s\n", "DEVICE", "APP", "PID", "BYTES");
    printa("%10s %20s %10d %15@d\n", @);
}
```

L'exécution de ce script pendant quelques secondes entraîne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./whoio.d
^C
    DEVICE                  APP        PID           BYTES
     cmdk0                   cp        790         1515520
       sd2                   cp        790         1527808
```

Cette sortie indique que cette activité *est* une copie du fichier `archives.tar` d'un périphérique à un autre. Cette conclusion pose une autre question : l'un de ces périphériques est-il plus rapide que l'autre ? Quel périphérique limite la copie ? Pour répondre à ces questions, vous devez connaître le débit effectif de chaque périphérique plutôt que le nombre d'octets par seconde transférés par chaque périphérique. Vous pouvez déterminer le débit à l'aide de l'exemple de script suivant :

```
#pragma D option quiet

io:::start
{
    start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
}

io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
    /*
     * We want to get an idea of our throughput to this device in KB/sec.
     * What we have, however, is nanoseconds and bytes.  That is we want
     * to calculate:
     *
     *                        bytes / 1024
     *                  ------------------------
     *                  nanoseconds / 1000000000
     *
     * But we can’t calculate this using integer arithmetic without losing
     * precision (the denomenator, for one, is between 0 and 1 for nearly
     * all I/Os).  So we restate the fraction, and cancel:
     *
```

```
     *     bytes      1000000000         bytes        976562
     *   --------- * -------------  =  --------- * -------------
     *      1024      nanoseconds          1        nanoseconds
     *
     * This is easy to calculate using integer arithmetic; this is what
     * we do below.
     */
    this->elapsed = timestamp - start[args[0]->b_edev, args[0]->b_blkno];
    @[args[1]->dev_statname, args[1]->dev_pathname] =
        quantize((args[0]->b_bcount * 976562) / this->elapsed);
    start[args[0]->b_edev, args[0]->b_blkno] = 0;
}

END
{
    printa("  %s (%s)\n%@d\n", @);
}
```

L'exécution de l'exemple de script pendant plusieurs secondes produit la sortie suivante :

```
  sd2 (/devices/pci@0,0/pci1179,1@1d/storage@2/disk@0,0:r)

           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |                                         3
             128 |                                         1
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  2257
             512 |                                         1
            1024 |                                         0

  cmdk0 (/devices/pci@0,0/pci-ide@1f,1/ide@0/cmdk@0,0:a)

           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |                                         1
             512 |                                         0
            1024 |                                         2
            2048 |                                         0
            4096 |                                         2
            8192 |@@@@@@@@@@@@@@@@@@                       172
           16384 |@@@@@                                    52
           32768 |@@@@@@@@@@@                              108
           65536 |@@@                                      34
          131072 |                                         0
```

La sortie indique clairement que `sd2` est le périphérique limiteur. Le débit `sd2` est compris entre 256 Ko/s et 512 Ko/s, alors que `cmdk0` fournit des E/S à un débit compris entre 8 Mo/s et plus de 64 Mo/s. Le script renvoie le nom tel qu'indiqué dans `iostat` et le chemin complet du

périphérique. Pour en savoir plus sur le périphérique, vous pourriez spécifier le chemin du périphérique dans `prtconf`, tel qu'illustré dans l'exemple suivant :

```
# prtconf -v /devices/pci@0,0/pci1179,1@1d/storage@2/disk@0,0
disk, instance #2 (driver name: sd)
    Driver properties:
        name='lba-access-ok' type=boolean dev=(29,128)
        name='removable-media' type=boolean dev=none
        name='pm-components' type=string items=3 dev=none
            value='NAME=spindle-motor' + '0=off' + '1=on'
        name='pm-hardware-state' type=string items=1 dev=none
            value='needs-suspend-resume'
        name='ddi-failfast-supported' type=boolean dev=none
        name='ddi-kernel-ioctl' type=boolean dev=none
    Hardware properties:
        name='inquiry-revision-id' type=string items=1
            value='1.04'
        name='inquiry-product-id' type=string items=1
            value='STORAGE DEVICE'
        name='inquiry-vendor-id' type=string items=1
            value='Generic'
        name='inquiry-device-type' type=int items=1
            value=00000000
        name='usb' type=boolean
        name='compatible' type=string items=1
            value='sd'
        name='lun' type=int items=1
            value=00000000
        name='target' type=int items=1
            value=00000000
```

Comme souligné, ce périphérique est un périphérique de stockage USB amovible.

Les exemples de cette section ont abordé toutes les demandes d'E/S. Cependant, un seul type de demande peut vous intéresser. L'exemple suivant permet de suivre les répertoires dans lesquels des écritures se produisent, ainsi que les applications à leur origine :

```
#pragma D option quiet

io:::start
/args[0]->b_flags & B_WRITE/
{
    @[execname, args[2]->fi_dirname] = count();
}

END
{
    printf("%20s %51s %5s\n", "WHO", "WHERE", "COUNT");
```

```
    printa("%20s %51s %5@d\n", @);
}
```

L'exécution de cet exemple de script sur une charge de travail de bureau sur une période donne des résultats intéressants, tel qu'illustré dans l'exemple suivant:

```
# dtrace -s ./whowrite.d
^C
                 WHO                                               WHERE COUNT
                  su                                            /var/adm     1
             fsflush                                                /etc     1
             fsflush                                                   /     1
             fsflush                                            /var/log     1
             fsflush                                    /export/bmc/lisa     1
                 esd   /export/bmc/.phoenix/default/78cxczuy.slt/Cache     1
             fsflush                                /export/bmc/.phoenix     1
                 esd         /export/bmc/.phoenix/default/78cxczuy.slt     1
                  vi                                            /var/tmp     2
                  vi                                                /etc     2
                 cat                                              <none>     2
                bash                                                   /     2
                  vi                                              <none>     3
               xterm                                            /var/adm     3
             fsflush                                         /export/bmc     7
     MozillaFirebird                                              <none>     8
                 vim                                         /export/bmc     9
     MozillaFirebird                                         /export/bmc    10
             fsflush                                            /var/adm    11
            devfsadm                                                /dev    14
                 ksh                                              <none>    71
                 ksh                                         /export/bmc    71
             fsflush         /export/bmc/.phoenix/default/78cxczuy.slt   119
     MozillaFirebird         /export/bmc/.phoenix/default/78cxczuy.slt   119
             fsflush                                              <none>   211
     MozillaFirebird   /export/bmc/.phoenix/default/78cxczuy.slt/Cache   591
             fsflush   /export/bmc/.phoenix/default/78cxczuy.slt/Cache   666
               sched                                              <none>  2385
```

Comme l'indique la sortie, quasiment toutes les écritures sont associées au cache Mozilla Firebird. Les écritures intitulées <none> sont probablement dues à des écritures associées au journal UFS, écritures dues elles-mêmes à d'autres écritures du système de fichiers. Pour plus d'informations sur la journalisation, reportez-vous à `ufs`(7FS). Cet exemple illustre comment utiliser le fournisseur `io` pour relever un problème à un niveau logiciel supérieur. Dans ce cas, le script révèle un problème de configuration : le navigateur Web serait à l'origine de moins d'E/S (voire probablement aucune) si son cache était situé dans un répertoire d'un système de fichiers `tmpfs`(7FS).

Les exemples précédents n'ont utilisé que les sondes `start` et `done.` Vous pouvez utiliser les sondes `wait-start` et `wait-done` pour comprendre pourquoi les applications bloquent des E/S, et pendant combien de temps. L'exemple de script suivant utilise des sondes `io` et des sondes `sched` (voir le Chapitre26, “Fournisseur `sched`”) pour dériver la durée CPU par rapport au délai d'attente d'E/S pour le logiciel StarOffice :

```
#pragma D option quiet

sched:::on-cpu
/execname == "soffice.bin"/
{
    self->on = vtimestamp;
}

sched:::off-cpu
/self->on/
{
    @time["<on cpu>"] = sum(vtimestamp - self->on);
    self->on = 0;
}

io:::wait-start
/execname == "soffice.bin"/
{
    self->wait = timestamp;
}

io:::wait-done
/self->wait/
{
    @io[args[2]->fi_name] = sum(timestamp - self->wait);
    @time["<I/O wait>"] = sum(timestamp - self->wait);
    self->wait = 0;
}

END
{
    printf("Time breakdown (milliseconds):\n");
    normalize(@time, 1000000);
    printa("  %-50s %15@d\n", @time);

    printf("\nI/O wait breakdown (milliseconds):\n");
    normalize(@io, 1000000);
    printa("  %-50s %15@d\n", @io);
}
```

L'exécution de l'exemple de script pendant un démarrage à froid du logiciel StarOffice donne la sortie suivante :

```
Time breakdown (milliseconds):
  <on cpu>                                                 3634
  <I/O wait>                                              13114

I/O wait breakdown (milliseconds):
  soffice.tmp                                                 0
  Office                                                      0
  unorc                                                       0
  sbasic.cfg                                                  0
  en                                                          0
  smath.cfg                                                   0
  toolboxlayout.xml                                           0
  sdraw.cfg                                                   0
  swriter.cfg                                                 0
  Linguistic.dat                                              0
  scalc.cfg                                                   0
  Views.dat                                                   0
  Store.dat                                                   0
  META-INF                                                    0
  Common.xml.tmp                                              0
  afm                                                         0
  libsimreg.so                                                1
  xiiimp.so.2                                                 3
  outline                                                     4
  Inet.dat                                                    6
  fontmetric                                                  6
  ...
  libucb1.so                                                 44
  libj641si_g.so                                             46
  libX11.so.4                                                46
  liblng641si.so                                             48
  swriter.db                                                 53
  libwrp641si.so                                             53
  liblocaledata_ascii.so                                     56
  libi18npool641si.so                                        65
  libdbtools2.so                                             69
  ofa64101.res                                               74
  libxcr641si.so                                             82
  libucpchelp1.so                                            83
  libsot641si.so                                             86
  libcppuhelper3C52.so                                       98
  libfwl641si.so                                            100
  libsb641si.so                                             104
  libcomphelp2.so                                           105
  libxo641si.so                                             106
  libucpfile1.so                                            110
  libcppu.so.3                                              111
  sw64101.res                                               114
```

```
  libdb-3.2.so                                                  119
  libtk641si.so                                                 126
  libdtransX11641si.so                                          127
  libgo641si.so                                                 132
  libfwe641si.so                                                150
  libi18n641si.so                                               152
  libfwi641si.so                                                154
  libso641si.so                                                 173
  libpsp641si.so                                                186
  libtl641si.so                                                 189
  <unknown>                                                     189
  libucbhelper1C52.so                                           195
  libutl641si.so                                                213
  libofa641si.so                                                216
  libfwk641si.so                                                229
  libsvl641si.so                                                261
  libcfgmgr2.so                                                 368
  libsvt641si.so                                                373
  libvcl641si.so                                                741
  libsvx641si.so                                                885
  libsfx641si.so                                                993
  <none>                                                       1096
  libsw641si.so                                                1365
  applicat.rdb                                                 1580
```

Comme le montre cette sortie, la lenteur du démarrage à froid de StarOffice est souvent due au délai d'attente d'E/S (13,1 secondes, par rapport à un délai de 3,6 secondes sur CPU). L'exécution du script sur un démarrage à chaud du logiciel StarOffice révèle que la mise en cache de page a supprimé la durée d'E/S, tel qu'illustré dans l'exemple suivant :

```
Time breakdown (milliseconds):
  <I/O wait>                                                      0
  <on cpu>                                                     2860

I/O wait breakdown (milliseconds):
  temp                                                            0
  soffice.tmp                                                     0
  <unknown>                                                       0
  Office                                                          0
```

La sortie du démarrage à froid illustre que le fichier `applicat.rdb` compte un délai d'attente d'E/S supérieur aux autres fichiers. Ce résultat est probablement dû au nombre important d'E/S pour ce fichier. Pour étudier les E/S pour ce fichier, vous pouvez utiliser le script D suivant :

```
io:::start
/execname == "soffice.bin" && args[2]->fi_name == "applicat.rdb"/
{
    @ = lquantize(args[2]->fi_offset != -1 ?
```

```
        args[2]->fi_offset / (1000 * 1024) : -1, 0, 1000);
}
```

Ce script utilise le champ `fi_offset` de la structure `fileinfo_t` pour connaître les parties accessibles du fichier, de l'ordre du mégaoctet. L'exécution de ce script pendant un démarrage à froid du logiciel StarOffice donne une sortie similaire à l'exemple suivant :

```
# dtrace -s ./applicat.d
dtrace: script './applicat.d' matched 4 probes
^C


           value  ------------ Distribution -----------  count
             < 0 |                                       0
               0 |@@@                                    28
               1 |@@                                     17
               2 |@@@@                                   35
               3 |@@@@@@@@@                              72
               4 |@@@@@@@@@@                             78
               5 |@@@@@@@@                               65
               6 |                                       0
```

Cette sortie indique que seuls les six premiers mégaoctets du fichier sont accessibles, probablement car la taille du fichier est de six mégaoctets. La sortie indique également que le fichier n'est pas accessible dans son intégralité. Si vous souhaitez améliorer la durée de démarrage à froid de StarOffice, vous pouvez souhaiter comprendre le modèle d'accès du fichier. Si les sections nécessaires du fichier pouvaient être contiguës, une méthode d'amélioration du démarrage à froid de StarOffice pourrait être d'exécuter un thread scout en amont de l'application, entraînant l'E/S du fichier plus tôt que requis (cette approche est tout particulièrement simple si le fichier est accessible via `mmap(2)`). Cependant, 1,6 seconde environ de démarrage à froid qui pourrait être économisée grâce à cette approche ne justifie pas la complexité et la charge de maintenance supplémentaires de l'application. D'autre part, les données collectées avec le fournisseur `io` permettent de comprendre précisément l'avantage qu'une telle tâche pourrait apporter.

Stabilité

Le fournisseur `io` utilise un mécanisme de stabilité DTrace pour décrire ses stabilités, tel qu'illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	En cours d'évolution	En cours d'évolution	ISA

CHAPITRE 28

Fournisseur mib

Le fournisseur mib propose des sondes correspondant à des compteurs dans les bases d'informations de gestion (MIB) Solaris. Les compteurs MIB sont utilisés par le protocole SNMP (simple network management protocol) permettant un contrôle à distance d'entités réseau hétérogènes. Vous pouvez également afficher les compteurs à l'aide des commandes kstat(1M) et netstat(1M). Le fournisseur mib permet une exploration rapide d'un comportement réseau anormal observé via des moniteurs réseau à distance ou locaux.

Sondes

Le fournisseur mib propose des sondes de compteurs de plusieurs MIB. Les protocoles permettant d'exporter des MIB instrumentalisées par le fournisseur mib sont répertoriés dans le Tableau 28–1. Le tableau inclut une référence à la documentation spécifiant tout ou partie de la MIB, le nom de la statistique du noyau pouvant être utilisée pour accéder aux comptes en cours (via l'option kstat(1M) *statistique* -n), ainsi qu'une référence au tableau indiquant une définition complète des sondes. Tous les compteurs MIB sont également disponibles via l'option -s pour netstat(1M).

TABLEAU 28–1 Sondes mib

Protocole	Description de MIB	Statistique du noyau	Table de sondes mib
ICMP	RFC 1213	icmp	Tableau 28–2
IP	RFC 1213	ip	Tableau 28–3
IPsec	—	ip	Tableau 28–4
IPv6	RFC 2465	—	Tableau 28–5
SCTP	MIB SCTP (brouillon Internet)	sctp	Tableau 28–7

TABLEAU 28–1 Sondes `mib` *(Suite)*

Protocole	Description de MIB	Statistique du noyau	Table de sondes `mib`
TCP	RFC 1213	`tcp`	Tableau 28–8
UDP	RFC 1213	`udp`	Tableau 28–9

TABLEAU 28–2 Sondes `mib` ICMP

`icmpInAddrMaskReps`	Sonde se déclenchant en cas de réception d'un message de réponse de masque d'adresse ICMP.
`icmpInAddrMasks`	Sonde se déclenchant en cas de réception d'un message de demande de masque d'adresse ICMP.
`icmpInBadRedirects`	Sonde se déclenchant en cas de réception d'un message de redirection ICMP considéré comme étant malformé (code ICMP inconnu, expéditeur ou destinataire hors ligne, etc.).
`icmpInCksumErrs`	Sonde se déclenchant en cas de réception d'un message ICMP dont la somme de contrôle est incorrecte.
`icmpInDestUnreachs`	Sonde se déclenchant en cas de réception d'un message de destination ICMP impossible.
`icmpInEchoReps`	Sonde se déclenchant en cas de réception d'un message de réponse d'écho ICMP.
`icmpInEchos`	Sonde se déclenchant en cas de réception d'un message de demande d'écho ICMP.
`icmpInErrors`	Sonde se déclenchant en cas de réception d'un message ICMP considéré comme comprenant une erreur spécifique à ICMP (somme de contrôle ICMP incorrecte, longueur incorrecte, etc.).
`icmpInFragNeeded`	Sonde se déclenchant en cas de réception d'un message de destination ICMP impossible (fragmentation nécessaire), indiquant qu'un paquet envoyé a été perdu car il était supérieur à une MTU et que l'indicateur de non-fragmentation était défini.
`icmpInMsgs`	Sonde se déclenchant en cas de réception d'un message ICMP. Lorsque cette sonde se déclenche, la sonde `icmpInErrors` peut également se déclencher si le message comprend une erreur spécifique à ICMP.
`icmpInOverflows`	Sonde se déclenchant en cas de réception d'un message ICMP, mais le message est ensuite abandonné en raison d'espace de tampon insuffisant.
`icmpInParmProbs`	Sonde se déclenchant en cas de réception d'un message de problème de paramètre ICMP.
`icmpInRedirects`	Sonde se déclenchant en cas de réception d'un message de redirection ICMP.

TABLEAU 28–2 Sondes `mib` ICMP *(Suite)*

`icmpInSrcQuenchs`	Sonde se déclenchant en cas de réception d'un message de source quench ICMP.
`icmpInTimeExcds`	Sonde se déclenchant en cas de réception d'un message de durée expirée ICMP.
`icmpInTimestampReps`	Sonde se déclenchant en cas de réception d'un message de réponse d'horodatage ICMP.
`icmpInTimestamps`	Sonde se déclenchant en cas de réception d'un message de demande d'horodatage ICMP.
`icmpInUnknowns`	Sonde se déclenchant en cas de réception d'un message ICMP de type inconnu.
`icmpOutAddrMaskReps`	Sonde se déclenchant en cas d'envoi d'un message de réponse de masque d'adresse ICMP.
`icmpOutDestUnreachs`	Sonde se déclenchant en cas d'envoi d'un message de destination ICMP impossible.
`icmpOutDrops`	Sonde se déclenchant si un message ICMP sortant est abandonné (échec d'allocation de mémoire, source ou destination de diffusion/multidiffusion, etc.).
`icmpOutEchoReps`	Sonde se déclenchant en cas d'envoi d'un message de réponse d'écho ICMP.
`icmpOutErrors`	Sonde se déclenchant si un message ICMP n'est pas envoyé en raison de problèmes rencontrés dans ICMP, tampons insuffisants par exemple. Cette sonde ne se déclenche pas si des erreurs sont rencontrées en dehors de la couche ICMP, comme l'incapacité d'IP à router le datagramme obtenu.
`icmpOutFragNeeded`	Sonde se déclenchant en cas d'envoi d'un message de destination ICMP impossible (fragmentation nécessaire).
`icmpOutMsgs`	Sonde se déclenchant en cas d'envoi d'un message ICMP. Lorsque cette sonde se déclenche, la sonde `icmpInErrors` peut également se déclencher si le message comprend des erreurs spécifiques à ICMP.
`icmpOutParmProbs`	Sonde se déclenchant en cas d'envoi d'un message de problème de paramètre ICMP.
`icmpOutRedirects`	Sonde se déclenchant en cas d'envoi d'un message de redirection ICMP. Cette sonde ne se déclenche jamais pour un hôte car les hôtes n'envoient pas de redirections.
`icmpOutTimeExcds`	Sonde se déclenchant en cas d'envoi d'un message de durée expirée ICMP.
`icmpOutTimestampReps`	Sonde se déclenchant en cas d'envoi d'un message de réponse d'horodatage ICMP.

TABLEAU 28–3 Sondes `mib` IP

`ipForwDatagrams`	Sonde se déclenchant en cas de réception d'un datagramme n'incluant pas cette machine comme destination IP finale, et en cas de tentative de recherche de route pour transférer le datagramme vers la destination finale. Sur des machines n'agissant pas en tant que passerelles IP, cette sonde ne se déclenche que pour les paquets routés à la source via cette machine et pour lesquels le traitement de l'option de route source a réussi.
`ipForwProhibits`	Sonde se déclenchant en cas de réception d'un datagramme n'incluant pas cette machine comme destination IP finale mais, étant donné que la machine n'est pas autorisée à agir en tant que routeur, aucune tentative de recherche de route pour transférer le datagramme vers cette destination finale n'est réalisée.
`ipFragCreates`	Sonde se déclenchant en cas de génération d'un fragment de datagramme IP suite à une fragmentation.
`ipFragFails`	Sonde se déclenclant si un datagramme IP est ignoré car il n'a pas pu être fragmenté, par exemple car une fragmentation était nécessaire et que l'indicateur de non-fragmentation était défini.
`ipFragOKs`	Sonde se déclenchant si un datagramme IP a été fragmenté avec succès.
`ipInCksumErrs`	Sonde se déclenchant si un datagramme d'entrée est ignoré en raison d'une somme de contrôle d'en-tête IP incorrecte.
`ipInDelivers`	Sonde se déclenchant si un datagramme d'entrée est fourni avec succès aux protocoles utilisateur IP, ICMP inclus.
`ipInDiscards`	Sonde se déclenchant si un datagramme IP d'entrée est ignoré pour des motifs autres que le paquet (espace de tampon insuffisant par exemple). Cette sonde ne se déclenche pas pour un datagramme ignoré dans l'attente d'un réassemblage.
`ipInHdrErrors`	Sonde se déclenchant si un datagramme d'entrée est ignoré en raison d'une erreur dans son en-tête IP, notamment une discordance de numéro de version, une erreur de format, une durée de vie expirée, une erreur rencontrée dans le traitement des options IP, etc.
`ipInIPv6`	Sonde se déclenchant si un paquet IPv6 arrive par erreur dans une file d'attente IPv4.
`ipInReceives`	Sonde se déclenchant en cas de réception d'un datagramme à partir d'une interface, même si ce datagramme est reçu par erreur.
`ipInUnknownProtos`	Sonde se déclenchant en cas de réception d'un datagramme adressé localement mais ignoré ensuite en raison d'un protocole inconnu ou non pris en charge.
`ipOutDiscards`	Sonde se déclenchant si un datagramme IP de sortie est ignoré pour des motifs autres que le paquet (espace de tampon insuffisant par exemple). Cette sonde se déclenche pour un paquet répertorié dans le compteur MIB `ipForwDatagrams` si le paquet satisfait un critère d'omission (discrétionnaire).

TABLEAU 28–3 Sondes `mib` IP *(Suite)*

`ipOutIPv6`	Sonde se déclenchant si un paquet IPv6 est envoyé via une connexion IPv4.
`ipOutNoRoutes`	Sonde se déclenchant si un datagramme IP est ignoré car aucune route n'a pu être trouvée pour le transmettre à sa destination. Cette sonde se déclenche pour un paquet répertorié dans le compteur MIB `ipForwDatagrams` si le paquet satisfait ce critère d'absence de route. Cette sonde se déclenche également pour tous les datagrammes ne pouvant pas être routés car toutes les passerelles par défaut sont hors service.
`ipOutRequests`	Sonde se déclenchant si un datagramme IP est fourni à IP pour être transmis à partir de protocoles utilisateur IP locaux (ICMP inclus). Notez que cette sonde ne se déclenche pas pour tout paquet répertorié dans le compteur MIB `ipForwDatagrams`.
`ipOutSwitchIPv6`	Sonde se déclenchant si une connexion passe de l'utilisation de IPv4 à IPv6 comme protocole IP.
`ipReasmDuplicates`	Sonde se déclenchant si l'algorithme de réassemblage IP détermine qu'un fragment IP ne contient *que* les données préalablement reçues.
`ipReasmFails`	Sonde se déclenchant si une panne est détectée par l'algorithme de réassemblage IP. Cette sonde ne se déclenche pas nécessairement pour tout fragment IP ignoré car certains algorithmes, notamment dans RFC 815, peuvent perdre la trace de fragments en les combinant à leur réception.
`ipReasmOKs`	Sonde se déclenchant si un datagramme IP est réassemblé avec succès.
`ipReasmPartDups`	Sonde se déclenchant si l'algorithme de réassemblage IP détermine qu'un fragment IP contient des données préalablement reçues et de nouvelles données.
`ipReasmReqds`	Sonde se déclenchant en cas de réception d'un fragment IP devant être réassemblé.

TABLEAU 28–4 Sondes `mib` IPsec

`ipsecInFailed`	Sonde se déclenchant si un paquet reçu est abandonné car il ne répond pas à la stratégie IPsec spécifiée.
`ipsecInSucceeded`	Sonde se déclenchant si un paquet reçu correspond à la stratégie IPsec spécifiée et si la poursuite du traitement est autorisée.

TABLEAU 28–5 Sondes mib IPv6

ipv6ForwProhibits	Sonde se déclenchant en cas de réception d'un datagramme IPv6 n'incluant pas cette machine comme destination IPv6 finale mais, étant donné que la machine n'est pas autorisée à agir en tant que routeur, aucune tentative de recherche de route pour transférer le datagramme vers cette destination finale n'est réalisée.
ipv6IfIcmpBadHoplimit	Sonde se déclenchant en cas de réception d'un message de protocole de découverte de voisin ICMPv6 dont la limite de saut est constatée comme étant inférieure au maximum défini. De tels messages peuvent ne pas provenir d'un voisin et être alors ignorés.
ipv6IfIcmpInAdminProhibs	Sonde se déclenchant en cas de réception d'un message de destination ICMPv6 impossible (communication administrativement interdite).
ipv6IfIcmpInBadNeighborAdvertisements	Sonde se déclenchant en cas de réception d'un message d'annonce de voisin ICMPv6 malformé.
ipv6IfIcmpInBadNeighborSolicitations	Sonde se déclenchant en cas de réception d'un message de sollicitation de voisin ICMPv6 malformé.
ipv6IfIcmpInBadRedirects	Sonde se déclenchant en cas de réception d'un message de redirection ICMPv6 malformé.
ipv6IfIcmpInDestUnreachs	Sonde se déclenchant en cas de réception d'un message de destination ICMPv6 impossible.
ipv6IfIcmpInEchoReplies	Sonde se déclenchant en cas de réception d'un message de réponse d'écho ICMPv6.
ipv6IfIcmpInEchos	Sonde se déclenchant en cas de réception d'un message de demande d'écho ICMPv6.
ipv6IfIcmpInErrors	Sonde se déclenchant en cas de réception d'un message ICMPv6 considéré comme comprenant une erreur spécifique à ICMPv6 (somme de contrôle ICMPv6 incorrecte, longueur incorrecte, etc.).
ipv6IfIcmpInGroupMembBadQueries	Sonde se déclenchant en cas de réception d'un message de requête d'appartenance à un groupe ICMPv6 malformé.
ipv6IfIcmpInGroupMembBadReports	Sonde se déclenchant en cas de réception d'un message de rapport d'appartenance à un groupe ICMPv6 malformé.
ipv6IfIcmpInGroupMembOurReports	Sonde se déclenchant en cas de réception d'un message de rapport d'appartenance à un groupe ICMPv6.

TABLEAU 28–5 Sondes `mib` IPv6 *(Suite)*

`ipv6IfIcmpInGroupMembQueries`	Sonde se déclenchant en cas de réception d'un message de requête d'appartenance à un groupe ICMPv6.
`ipv6IfIcmpInGroupMembReductions`	Sonde se déclenchant en cas de réception d'un message de réduction d'appartenance à un groupe ICMPv6.
`ipv6IfIcmpInGroupMembResponses`	Sonde se déclenchant en cas de réception d'un message de réponse d'appartenance à un groupe ICMPv6.
`ipv6IfIcmpInGroupMembTotal`	Sonde se déclenchant en cas de réception d'un message de découverte de listener de multidiffusion ICMPv6.
`ipv6IfIcmpInMsgs`	Sonde se déclenchant en cas de réception d'un message ICMPv6. Lorsque cette sonde se déclenche, la sonde `ipv6IfIcmpInErrors` peut également se déclencher si le message comprend une erreur spécifique à ICMPv6.
`ipv6IfIcmpInNeighborAdvertisements`	Sonde se déclenchant en cas de réception d'un message d'annonce de voisin ICMPv6.
`ipv6IfIcmpInNeighborSolicits`	Sonde se déclenchant en cas de réception d'un message de sollicitation de voisin ICMPv6.
`ipv6IfIcmpInOverflows`	Sonde se déclenchant en cas de réception d'un message ICMPv6, mais le message est ensuite abandonné en raison d'espace de tampon insuffisant.
`ipv6IfIcmpInParmProblems`	Sonde se déclenchant en cas de réception d'un message de problème de paramètre ICMPv6.
`ipv6IfIcmpInRedirects`	Sonde se déclenchant en cas de réception d'un message de redirection ICMPv6.
`ipv6IfIcmpInRouterAdvertisements`	Sonde se déclenchant en cas de réception d'un message d'annonce de routeur ICMPv6.
`ipv6IfIcmpInRouterSolicits`	Sonde se déclenchant en cas de réception d'un message de sollicitation de routeur ICMPv6.
`ipv6IfIcmpInTimeExcds`	Sonde se déclenchant en cas de réception d'un message de durée expirée ICMPv6.
`ipv6IfIcmpOutAdminProhibs`	Sonde se déclenchant en cas d'envoi d'un message de destination ICMPv6 impossible (communication administrativement interdite).
`ipv6IfIcmpOutDestUnreachs`	Sonde se déclenchant en cas d'envoi d'un message de destination ICMPv6 impossible.
`ipv6IfIcmpOutEchoReplies`	Sonde se déclenchant en cas d'envoi d'un message de réponse d'écho ICMPv6.

TABLEAU 28–5 Sondes mib IPv6 *(Suite)*

`ipv6IfIcmpOutEchos`	Sonde se déclenchant en cas d'envoi d'un message d'écho ICMPv6.
`ipv6IfIcmpOutErrors`	Sonde se déclenchant si un message ICMPv6 n'est pas envoyé en raison de problèmes rencontrés dans ICMPv6, tampons insuffisants par exemple. Cette sonde ne se déclenche pas si des erreurs sont rencontrées en dehors de la couche ICMPv6, comme l'incapacité d'IPv6 à router le datagramme obtenu.
`ipv6IfIcmpOutGroupMembQueries`	Sonde se déclenchant en cas d'envoi d'un message de requête d'appartenance à un groupe ICMPv6.
`ipv6IfIcmpOutGroupMembReductions`	Sonde se déclenchant en cas d'envoi d'un message de réduction d'appartenance à un groupe ICMPv6.
`ipv6IfIcmpOutGroupMembResponses`	Sonde se déclenchant en cas d'envoi d'un message de réponse d'appartenance à un groupe ICMPv6.
`ipv6IfIcmpOutMsgs`	Sonde se déclenchant en cas d'envoi d'un message ICMPv6. Lorsque cette sonde se déclenche, la sonde `ipv6IfIcmpOutErrors` peut également se déclencher si le message comprend des erreurs spécifiques à ICMPv6.
`ipv6IfIcmpOutNeighborAdvertisements`	Sonde se déclenchant en cas d'envoi d'un message d'annonce de voisin ICMPv6.
`ipv6IfIcmpOutNeighborSolicits`	Sonde se déclenchant en cas d'envoi d'un message de sollicitation de voisin ICMPv6.
`ipv6IfIcmpOutParmProblems`	Sonde se déclenchant en cas d'envoi d'un message de problème de paramètre ICMPv6.
`ipv6IfIcmpOutPktTooBigs`	Sonde se déclenchant en cas d'envoi d'un message de paquet ICMPv6 trop volumineux.
`ipv6IfIcmpOutRedirects`	Sonde se déclenchant en cas d'envoi d'un message de redirection ICMPv6. Cette sonde ne se déclenche jamais pour un hôte car les hôtes n'envoient pas de redirections.
`ipv6IfIcmpOutRouterAdvertisements`	Sonde se déclenchant en cas d'envoi d'un message d'annonce de routeur ICMPv6.
`ipv6IfIcmpOutRouterSolicits`	Sonde se déclenchant en cas d'envoi d'un message de sollicitation de routeur ICMPv6.
`ipv6IfIcmpOutTimeExcds`	Sonde se déclenchant en cas d'envoi d'un message de durée expirée ICMPv6.

TABLEAU 28–5 Sondes `mib` IPv6 *(Suite)*

`ipv6InAddrErrors`	Sonde se déclenchant si un datagramme d'entrée est ignoré car l'adresse IPv6 de son champ de destination d'en-tête IPv6 n'est pas une adresse valide de réception par cette entité. Cette sonde se déchenche pour des adresses non valides (par exemple, : : 0) et pour des adresses non prises en charge (par exemple, des adresses sans préfixes alloués). Pour les machines non configurées pour agir en tant que routeurs IPv6 et ne transférant donc pas de datagrammes, cette sonde se déclenche pour des datagrammes ignorés car l'adresse de destination n'était pas une adresse locale.
`ipv6InDelivers`	Sonde se déclenchant si un datagramme d'entrée est fourni avec succès aux protocoles utilisateur IPv6 (ICMPv6 inclus).
`ipv6InDiscards`	Sonde se déclenchant si un datagramme IPv6 d'entrée est ignoré pour des motifs autres que le paquet (espace de tampon insuffisant par exemple). Cette sonde ne se déclenche pas pour un datagramme ignoré dans l'attente d'un réassemblage.
`ipv6InHdrErrors`	Sonde se déclenchant si un datagramme d'entrée est ignoré en raison d'une erreur dans son en-tête IPv6, notamment une discordance de numéro de version, une erreur de format, un nombre de sauts supérieur, une erreur rencontrée dans le traitement des options IPv6, etc.
`ipv6InIPv4`	Sonde se déclenchant si un paquet IPv4 arrive par erreur dans une file d'attente IPv6.
`ipv6InMcastPkts`	Sonde se déclenchant en cas de réception d'un paquet de multidiffusion IPv6.
`ipv6InNoRoutes`	Sonde se déclenchant si un datagramme IPv6 routé est ignoré car aucune route n'a pu être trouvée pour le transmettre à sa destination. Cette sonde ne se déclenche *que* pour des paquets provenant de l'extérieur.
`ipv6InReceives`	Sonde se déclenchant en cas de réception d'un datagramme IPv6 à partir d'une interface, même si ce datagramme est reçu par erreur.
`ipv6InTooBigErrors`	Sonde se déclenchant en cas de réception d'un fragment supérieur à la taille maximale de fragment.
`ipv6InTruncatedPkts`	Sonde se déclenchant si un datagramme d'entrée est ignoré car sa structure ne contenait pas suffisamment de données.

TABLEAU 28–5 Sondes `mib` IPv6 *(Suite)*

`ipv6InUnknownProtos`	Sonde se déclenchant en cas de réception d'un datagramme IPv6 adressé localement mais ignoré ensuite en raison d'un protocole inconnu ou non pris en charge.
`ipv6OutDiscards`	Sonde se déclenchant si un datagramme IPv6 de sortie est ignoré pour des motifs autres que le paquet (espace de tampon insuffisant par exemple). Cette sonde se déclenche pour un paquet répertorié dans le compteur MIB `ipv6OutForwDatagrams` si le paquet satisfait un tel critère d'omission (discrétionnaire).
`ipv6OutForwDatagrams`	Sonde se déclenchant en cas de réception d'un datagramme n'incluant pas cette machine comme destination IPv6 finale, et en cas de tentative de recherche de route pour transférer le datagramme vers la destination finale. Sur une machine n'agissant pas en tant que routeur IPv6, cette sonde ne se déclenche que pour les paquets routés à la source via cette machine et pour lesquels le traitement de l'option de route source a réussi.
`ipv6OutFragCreates`	Sonde se déclenchant en cas de génération d'un fragment de datagramme IPv6 suite à une fragmentation.
`ipv6OutFragFails`	Sonde se déclenclant si un datagramme IPv6 est ignoré car il n'a pas pu être fragmenté, par exemple car son indicateur de non-fragmentation était défini.
`ipv6OutFragOKs`	Sonde se déclenchant si des datagrammes IPv6 ont été fragmentés avec succès.
`ipv6OutIPv4`	Sonde se déclenchant si un paquet IPv6 est envoyé via une connexion IPv4.
`ipv6OutMcastPkts`	Sonde se déclenchant en cas d'envoi d'un paquet de multidiffusion.
`ipv6OutNoRoutes`	Sonde se déclenchant si un datagramme IPv6 est ignoré car aucune route n'a pu être trouvée pour le transmettre à sa destination. Cette sonde ne se déclenche *pas* pour des paquets provenant de l'extérieur.
`ipv6OutRequests`	Sonde se déclenchant si un datagramme IPv6 est fourni à IPv6 pour être transmis à partir de protocoles utilisateur IPv6 locaux (ICMPv6 inclus). Cette sonde ne se déclenche pas pour tout paquet répertorié dans le compteur MIB `ipv6ForwDatagrams`.

TABLEAU 28–5 Sondes `mib` IPv6 *(Suite)*

`ipv6OutSwitchIPv4`	Sonde se déclenchant si une connexion passe de l'utilisation de IPv6 à IPv4 comme protocole IP.
`ipv6ReasmDuplicates`	Sonde se déclenchant si l'algorithme de réassemblage IPv6 détermine qu'un fragment IPv6 ne contient *que* les données préalablement reçues.
`ipv6ReasmFails`	Sonde se déclenchant si une panne est détectée par l'algorithme de réassemblage IPv6. Cette sonde ne se déclenche pas nécessairement pour chaque fragment IPv6 ignoré puisque certains algorithmes peuvent perdre la trace de fragments en les combinant à leur réception.
`ipv6ReasmOKs`	Sonde se déclenchant si un datagramme IPv6 est réassemblé avec succès.
`ipv6ReasmPartDups`	Sonde se déclenchant si l'algorithme de réassemblage IPv6 détermine qu'un fragment IPv6 contient des données préalablement reçues et de nouvelles données.
`ipv6ReasmReqds`	Sonde se déclenchant en cas de réception d'un fragment IPv6 devant être réassemblé.

TABLEAU 28–6 Sondes `mib` IP brutes

`rawipInCksumErrs`	Sonde se déclenchant en cas de réception d'un paquet IP brut dont la somme de contrôle IP est incorrecte.
`rawipInDatagrams`	Sonde se déclenchant en cas de réception d'un paquet IP brut.
`rawipInErrors`	Sonde se déclenchant en cas de réception d'un paquet IP brut malformé.
`rawipInOverflows`	Sonde se déclenchant en cas de réception d'un paquet IP brut, mais ce paquet est ensuite abandonné en raison d'espace de tampon insuffisant.
`rawipOutDatagrams`	Sonde se déclenchant en cas d'envoi d'un paquet IP brut.
`rawipOutErrors`	Sonde se déclenchant si un paquet IP brut n'est pas envoyé en raison d'une condition d'erreur, généralement car le paquet IP brut est malformé.

TABLEAU 28–7 Sondes `mib` SCTP

`sctpAborted`	Sonde se déclenchant si une association SCTP a créé une transition directe d'un quelconque état à l'état CLOSED à l'aide de la primitive ABORT, indiquant ainsi une fin indésirable de l'association.
`sctpActiveEstab`	Sonde se déclenchant si une association SCTP a créé une transition directe de l'état COOKIE-ECHOED à l'état ESTABLISHED, indiquant ainsi que la couche supérieure a initié la tentative d'association.

TABLEAU 28–7 Sondes `mib` SCTP *(Suite)*

`sctpChecksumError`	Sonde se déclenchant en cas de réception d'un paquet SCTP de pairs dont la somme de contrôle n'est pas valide.
`sctpCurrEstab`	Sonde se déclenchant si une association SCTP est pointée dans le cadre de la lecture du compteur MIB `sctpCurrEstab`. Une association SCTP est pointée si l'état actuel est ESTABLISHED, SHUTDOWN-RECEIVED ou SHUTDOWN-PENDING.
`sctpFragUsrMsgs`	Sonde se déclenchant si un message utilisateur doit être fragmenté en raison de la MTU.
`sctpInClosed`	Sonde se déclenchant en cas de réception de données sur une association SCTP fermée.
`sctpInCtrlChunks`	Sonde se déclenchant si le compteur MIB `sctpInCtrlChunks` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.
`sctpInDupAck`	Sonde se déclenchant en cas de réception d'un doublon d'ACK.
`sctpInInvalidCookie`	Sonde se déclenchant en cas de réception d'un cookie non valide.
`sctpInOrderChunks`	Sonde se déclenchant si le compteur MIB `sctpInOrderChunks` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.
`sctpInSCTPPkts`	Sonde se déclenchant si le compteur MIB `sctpInSCTPPkts` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.
`sctpInUnorderChunks`	Sonde se déclenchant si le compteur MIB `sctpInUnorderChunks` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.
`sctpListenDrop`	Sonde se déclenchant si une connexion entrante est arrêtée, quel que soit le motif.
`sctpOutAck`	Sonde se déclenchant en cas d'envoi d'un accusé réception sélectif.
`sctpOutAckDelayed`	Sonde se déclenchant si un traitement d'accusé réception différé est effectué pour une association SCTP. Tout accusé réception envoyé dans le cadre du traitement d'accusé réception différé entraînera le déclenchement de la sonde `sctpOutAck`.
`sctpOutCtrlChunks`	Sonde se déclenchant si le compteur MIB `sctpOutCtrlChunks` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.

TABLEAU 28–7 Sondes mib SCTP *(Suite)*

sctpOutOfBlue	Sonde se déclenchant en cas de réception d'un paquet SCTP correct pour lequel le destinataire n'est pas en mesure d'identifier l'association à laquelle le paquet appartient.
sctpOutOrderChunks	Sonde se déclenchant si le compteur MIB `sctpOutOrderChunks` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.
sctpOutSCTPPkts	Sonde se déclenchant si le compteur MIB `sctpOutSCTPPkts` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.
sctpOutUnorderChunks	Sonde se déclenchant si le compteur MIB `sctpOutUnorderChunks` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.
sctpOutWinProbe	Sonde se déclenchant en cas d'envoi d'une sonde de fenêtre.
sctpOutWinUpdate	Sonde se déclenchant en cas d'envoi d'une mise à jour de fenêtre.
sctpPassiveEstab	Sonde se déclenchant si des associations SCTP ont créé une transition directe de l'état CLOSED à l'état ESTABLISHED. Le point d'extrémité distant a initié la tentative d'association.
sctpReasmUsrMsgs	Sonde se déclenchant si le compteur MIB `sctpReasmUsrMsgs` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.
sctpRetransChunks	Sonde se déclenchant si le compteur MIB `sctpRetransChunks` est mis à jour, car il est interrogé explicitement ou car une connexion SCTP est fermée. La valeur selon laquelle le compteur MIB doit être augmenté est indiquée dans `args[0]`.
sctpShutdowns	Sonde se déclenchant si une association SCTP crée la transition directe de l'état SHUTDOWN-SENT ou SHUTDOWN-ACK-SENT à l'état CLOSED, indiquant ainsi une fin souhaitable de l'association.
sctpTimHeartBeatDrop	Sonde se déclenchant en cas d'abandon d'une association SCTP en raison de l'échec de réception d'un accusé réception de pulsation.
sctpTimHeartBeatProbe	Sonde se déclenchant en cas d'envoi d'une pulsation SCTP.
sctpTimRetrans	Sonde se déclenchant si un traitement de retransmission basé sur le temps est effectué sur une association.
sctpTimRetransDrop	Sonde se déclenchant si un échec prolongé de la retransmission basée sur le temps entraîne l'abandon de l'association.

TABLEAU 28–8 Sondes `mib` TCP

`tcpActiveOpens`	Sonde se déclenchant si une connexion TCP crée une transition directe de l'état CLOSED à l'état SYN_SENT.
`tcpAttemptFails`	Sonde se déclenchant si une connexion TCP crée une transition directe de l'état SYN_SENT ou SYN_RCVD à l'état CLOSED et si une connexion TCP crée une transition directe de l'état SYN_RCVD à l'état LISTEN.
`tcpCurrEstab`	Sonde se déclenchant si une connexion SCTP est pointée dans le cadre de la lecture du compteur MIB `tcpCurrEstab.` Une connexion TCP est pointée si l'état actuel est ESTABLISHED ou CLOSE_WAIT.
`tcpEstabResets`	Sonde se déclenchant si une connexion TCP crée la transition directe de l'état ESTABLISHED ou CLOSE_WAIT à l'état CLOSED.
`tcpHalfOpenDrop`	Sonde se déclenchant si une connexion est arrêtée en raison d'une file d'attente saturée de connexions à l'état SYN_RCVD.
`tcpInAckBytes`	Sonde se déclenchant en cas de réception d'un ACK pour des données préalablement envoyées. Le nombre d'octets accusés réception est indiqué dans `args[0]`.
`tcpInAckSegs`	Sonde se déclenchant en cas de réception d'un ACK pour un segment préalablement envoyé.
`tcpInAckUnsent`	Sonde se déclenchant en cas de réception d'un ACK pour un segment non envoyé.
`tcpInClosed`	Sonde se déclenchant si des données ont été envoyées pour une connexion à l'état de fermeture.
`tcpInDataDupBytes`	Sonde se déclenchant en cas de réception d'un segment de sorte que toutes les données du segment ont été préalablement reçues. Le nombre d'octets du segment dupliqué est indiqué dans `args[0]`.
`tcpInDataDupSegs`	Sonde se déclenchant en cas de réception d'un segment de sorte que toutes les données du segment ont été préalablement reçues. Le nombre d'octets du segment dupliqué est indiqué dans `args[0]`.
`tcpInDataInorderBytes`	Sonde se déclenchant en cas de réception de données de sorte que *toutes* les données antérieures au numéro de séquence des nouvelles données ont été préalablement reçues. Le nombre d'octets reçus dans l'ordre est indiqué dans `args[0]`.
`tcpInDataInorderSegs`	Sonde se déclenchant en cas de réception d'un segment de sorte que *toutes* les données antérieures au numéro de séquence du nouveau segment ont été préalablement reçues.
`tcpInDataPartDupBytes`	Sonde se déclenchant en cas de réception d'un segment de sorte que certaines données du segment ont été préalablement reçues, mais que certaines données du segment sont nouvelles. Le nombre d'octets dupliqués est indiqué dans `args[0]`.

TABLEAU 28–8 Sondes mib TCP (Suite)

tcpInDataPartDupSegs	Sonde se déclenchant en cas de réception d'un segment de sorte que certaines données du segment ont été préalablement reçues, mais que certaines données du segment sont nouvelles. Le nombre d'octets dupliqués est indiqué dans args[0].
tcpInDataPastWinBytes	Sonde se déclenchant en cas de réception de données basées sur la fenêtre de réception actuelle. Le nombre d'octets est indiqué dans args[0].
tcpInDataPastWinSegs	Sonde se déclenchant en cas de réception d'un segment basé sur la fenêtre de réception actuelle.
tcpInDataUnorderBytes	Sonde se déclenchant en cas de réception de données de sorte que certaines données antérieures au numéro de séquence des nouvelles données sont manquantes. Le nombre d'octets reçus non classés est indiqué dans args[0].
tcpInDataUnorderSegs	Sonde se déclenchant en cas de réception d'un segment de sorte que certaines données antérieures au numéro de séquence des nouvelles données sont manquantes.
tcpInDupAck	Sonde se déclenchant en cas de réception d'un doublon d'ACK.
tcpInErrs	Sonde se déclenchant si une erreur TCP (par exemple, une somme de contrôle TCP incorrecte) est rencontrée dans un segment reçu.
tcpInSegs	Sonde se déclenchant en cas de réception d'un segment, même s'il est constaté ultérieurement que celui-ci comprend une erreur empêchant la poursuite du traitement.
tcpInWinProbe	Sonde se déclenchant en cas de réception d'une sonde de fenêtre.
tcpInWinUpdate	Sonde se déclenchant en cas de réception d'une mise à jour de fenêtre.
tcpListenDrop	Sonde se déclenchant si une connexion entrante est arrêtée en raison d'une file d'attente d'écoute saturée.
tcpListenDropQ0	Sonde se déclenchant si une connexion est arrêtée en raison d'une file d'attente saturée de connexions à l'état SYN_RCVD.
tcpOutAck	Sonde se déclenchant en cas d'envoi d'un ACK.
tcpOutAckDelayed	Sonde se déclenchant en cas d'envoi d'un ACK après qu'il a été initialement différé.
tcpOutControl	Sonde se déclenchant en cas d'envoi de SYN, FIN ou RST.
tcpOutDataBytes	Sonde se déclenchant en cas d'envoi de données. Le nombre d'octets envoyés est indiqué dans args[0].
tcpOutDataSegs	Sonde se déclenchant en cas d'envoi d'un segment.
tcpOutFastRetrans	Sonde se déclenchant en cas de transmission d'un segment dans le cadre de l'algorithme de retransmission rapide.

TABLEAU 28–8 Sondes `mib` TCP *(Suite)*

`tcpOutRsts`	Sonde se déclenchant en cas d'envoi d'un segment avec l'indicateur RST défini.
`tcpOutSackRetransSegs`	Sonde se déclenchant en cas de retransmission d'un segment sur une connexion sur laquelle l'accusé réception sélectif est activé.
`tcpOutSegs`	Sonde se déclenchant en cas d'envoi d'un segment contenant au moins un octet non retransmis.
`tcpOutUrg`	Sonde se déclenchant en cas d'envoi d'un segment avec l'indicateur URG défini et un pointeur d'urgence valide.
`tcpOutWinProbe`	Sonde se déclenchant en cas d'envoi d'une sonde de fenêtre.
`tcpOutWinUpdate`	Sonde se déclenchant en cas d'envoi d'une mise à jour de fenêtre.
`tcpPassiveOpens`	Sonde se déclenchant si une connexion TCP a créé une transition directe de l'état LISTEN à l'état SYN_RCVD.
`tcpRetransBytes`	Sonde se déclenchant en cas de retransmission de données. Le nombre d'octets retransmis est indiqué dans `args[0]`.
`tcpRetransSegs`	Sonde se déclenchant en cas d'envoi d'un segment contenant un ou plusieurs octets retransmis.
`tcpRttNoUpdate`	Sonde se déclenchant en cas de réception de données mais sans informations d'horodatage disponibles grâce auxquelles il est possible de mettre à jour RTT.
`tcpRttUpdate`	Sonde se déclenchant en cas de réception de données contenant les informations d'horodatage nécessaires à la mise à jour de RTT.
`tcpTimKeepalive`	Sonde se déclenchant si un traitement de connexion persistante basé sur le temps est effectué sur une connexion.
`tcpTimKeepaliveDrop`	Sonde se déclenchant si le traitement de connexion persistante entraîne l'arrêt d'une connexion.
`tcpTimKeepaliveProbe`	Sonde se déclenchant si une sonde de connexion persistante est envoyée dans le cadre du traitement de connexion persistante.
`tcpTimRetrans`	Sonde se déclenchant si un traitement de retransmission basé sur le temps est effectué sur une connexion.
`tcpTimRetransDrop`	Sonde se déclenchant si un échec prolongé de la retransmission basée sur le temps entraîne l'arrêt de la connexion.

TABLEAU 28–9 Sondes `mib` UDP

`udpInCksumErrs`	Sonde se déclenchant si un datagramme est ignoré en raison d'une somme de contrôle UDP incorrecte.
`udpInDatagrams`	Sonde se déclenchant en cas de réception d'un datagramme UDP.

TABLEAU 28–9 Sondes mib UDP (Suite)

udpInErrors	Sonde se déclenchant en cas de réception d'un datagramme UDP, ignoré en raison d'un en-tête de paquet malformé ou d'un échec d'allocation dans le tampon interne.
udpInOverflows	Sonde se déclenchant en cas de réception d'un datagramme UDP, abandonné ensuite en raison d'espace de tampon insuffisant.
udpNoPorts	Sonde se déclenchant en cas de réception d'un datagramme UDP sur un port auquel aucun socket n'est lié.
udpOutDatagrams	Sonde se déclenchant en cas d'envoi d'un datagramme UDP.
udpOutErrors	Sonde se déclenchant si un datagramme UDP n'est pas envoyé en raison d'une condition d'erreur, généralement car le datagramme est malformé.

Arguments

L'unique argument de chaque sonde mib présente la même sémantique : args[0] contient la valeur selon laquelle le compteur doit être incrémenté. Pour la plupart des sondes mib, args[0] contient toujours la valeur 1, mais pour certaines sondes, args[0] peut prendre des valeurs positives arbitraires. Pour ces sondes, la signification de args[0] est indiquée dans la description de la sonde.

Stabilité

Le fournisseur mib utilise un mécanisme de stabilité DTrace pour décrire ses stabilités, tel qu'illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	En cours d'évolution	En cours d'évolution	ISA

CHAPITRE 29

Fournisseur fpuinfo

Le fournisseur fpuinfo propose des sondes correspondant à la simulation d'instructions à virgule flottante sur des microprocesseurs SPARC. Alors que la plupart des instructions à virgule flottante sont exécutées sur du matériel, certaines opérations correspondantes sont déroutées dans le système d'exploitation pour simulation. Les conditions dans lesquelles des opérations à virgule flottante requièrent une simulation de système d'exploitation sont spécifiques à la mise en œuvre du microprocesseur. De telles opérations nécessitant une simulation sont rares. Cependant, si une application utilise fréquemment l'une de ces opérations, l'effet sur la performance peut être critique. Le fournisseur fpuinfo offre une recherche rapide de simulation en virgule flottante dont le résultat est consultable via kstat(1M) et la statistique de noyau fpu_info ou via trapstat(1M) et le déroutement fp-xcp-other.

Sondes

Le fournisseur fpuinfo propose une sonde de chaque type d'instruction à virgule flottante pouvant être simulée. Le fournisseur fpuinfo comprend un Nom de stabilité de CPU ; les noms des sondes sont spécifiques à la mise en œuvre du microprocesseur, et peuvent ne pas être disponibles sur des microprocesseurs différents d'une même famille. Par exemple, certaines sondes répertoriées ne peuvent être disponibles que sur UltraSPARC-III et pas sur UltraSPARC-III+ et vice-versa.

Les sondes fpuinfo sont décrites dans le Tableau 29–1.

TABLEAU 29–1 Sondes fpuinfo

fpu_sim_fitoq	Sonde se déclenchant si une instruction fitoq est simulée par le noyau.
fpu_sim_fitod	Sonde se déclenchant si une instruction fitod est simulée par le noyau.
fpu_sim_fitos	Sonde se déclenchant si une instruction fitos est simulée par le noyau.

TABLEAU 29–1 Sondes fpuinfo *(Suite)*

fpu_sim_fxtoq	Sonde se déclenchant si une instruction fxtoq est simulée par le noyau.
fpu_sim_fxtod	Sonde se déclenchant si une instruction fxtod est simulée par le noyau.
fpu_sim_fxtos	Sonde se déclenchant si une instruction fxtos est simulée par le noyau.
fpu_sim_fqtox	Sonde se déclenchant si une instruction fqtox est simulée par le noyau.
fpu_sim_fdtox	Sonde se déclenchant si une instruction fdtox est simulée par le noyau.
fpu_sim_fstox	Sonde se déclenchant si une instruction fstox est simulée par le noyau.
fpu_sim_fqtoi	Sonde se déclenchant si une instruction fqtoi est simulée par le noyau.
fpu_sim_fdtoi	Sonde se déclenchant si une instruction fdtoi est simulée par le noyau.
fpu_sim_fstoi	Sonde se déclenchant si une instruction fstoi est simulée par le noyau.
fpu_sim_fsqrtq	Sonde se déclenchant si une instruction fsqrtq est simulée par le noyau.
fpu_sim_fsqrtd	Sonde se déclenchant si une instruction fsqrtd est simulée par le noyau.
fpu_sim_fsqrts	Sonde se déclenchant si une instruction fsqrts est simulée par le noyau.
fpu_sim_fcmpeq	Sonde se déclenchant si une instruction fcmpeq est simulée par le noyau.
fpu_sim_fcmped	Sonde se déclenchant si une instruction fcmped est simulée par le noyau.
fpu_sim_fcmpes	Sonde se déclenchant si une instruction fcmpes est simulée par le noyau.
fpu_sim_fcmpq	Sonde se déclenchant si une instruction fcmpq est simulée par le noyau.
fpu_sim_fcmpd	Sonde se déclenchant si une instruction fcmpd est simulée par le noyau.
fpu_sim_fcmps	Sonde se déclenchant si une instruction fcmps est simulée par le noyau.
fpu_sim_fdivq	Sonde se déclenchant si une instruction fdivq est simulée par le noyau.
fpu_sim_fdivd	Sonde se déclenchant si une instruction fdivd est simulée par le noyau.
fpu_sim_fdivs	Sonde se déclenchant si une instruction fdivs est simulée par le noyau.
fpu_sim_fdmulx	Sonde se déclenchant si une instruction fdmulx est simulée par le noyau.
fpu_sim_fsmuld	Sonde se déclenchant si une instruction fsmuld est simulée par le noyau.
fpu_sim_fmulq	Sonde se déclenchant si une instruction fmulq est simulée par le noyau.
fpu_sim_fmuld	Sonde se déclenchant si une instruction fmuld est simulée par le noyau.
fpu_sim_fmuls	Sonde se déclenchant si une instruction fmuls est simulée par le noyau.
fpu_sim_fsubq	Sonde se déclenchant si une instruction fsubq est simulée par le noyau.
fpu_sim_fsubd	Sonde se déclenchant si une instruction fsubd est simulée par le noyau.
fpu_sim_fsubs	Sonde se déclenchant si une instruction fsubs est simulée par le noyau.

TABLEAU 29–1 Sondes fpuinfo *(Suite)*

fpu_sim_faddq	Sonde se déclenchant si une instruction faddq est simulée par le noyau.
fpu_sim_faddd	Sonde se déclenchant si une instruction faddd est simulée par le noyau.
fpu_sim_fadds	Sonde se déclenchant si une instruction fadds est simulée par le noyau.
fpu_sim_fnegd	Sonde se déclenchant si une instruction fnegd est simulée par le noyau.
fpu_sim_fnegq	Sonde se déclenchant si une instruction fneqq est simulée par le noyau.
fpu_sim_fnegs	Sonde se déclenchant si une instruction fnegs est simulée par le noyau.
fpu_sim_fabsd	Sonde se déclenchant si une instruction fabsd est simulée par le noyau.
fpu_sim_fabsq	Sonde se déclenchant si une instruction fabsq est simulée par le noyau.
fpu_sim_fabss	Sonde se déclenchant si une instruction fabss est simulée par le noyau.
fpu_sim_fmovd	Sonde se déclenchant si une instruction fmovd est simulée par le noyau.
fpu_sim_fmovq	Sonde se déclenchant si une instruction fmovq est simulée par le noyau.
fpu_sim_fmovs	Sonde se déclenchant si une instruction fmovs est simulée par le noyau.
fpu_sim_fmovr	Sonde se déclenchant si une instruction fmovr est simulée par le noyau.
fpu_sim_fmovcc	Sonde se déclenchant si une instruction fmovcc est simulée par le noyau.

Arguments

Il n'existe aucun argument pour les sondes fpuinfo.

Stabilité

Le fournisseur fpuinfo utilise un mécanisme de stabilité DTrace pour décrire ses stabilités, tel qu'illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	CPU
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	CPU

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Arguments	En cours d'évolution	En cours d'évolution	CPU

CHAPITRE 30

Fournisseur pid

Le fournisseur pid permet de suivre l'entrée et le retour d'une fonction d'un processus utilisateur, ainsi que de toute instruction telle que spécifiée par une adresse absolue ou un décalage de fonction. Le fournisseur pid n'entraîne aucun effet de sonde lorsqu'aucune sonde n'est activée. Lorsque des sondes sont activées, elles n'entraînent d'effet que sur les processus suivis.

Remarque – Lorsque le compilateur intègre une fonction, la sonde pid du fournisseur ne se déclenche pas. Pour éviter l'intégration d'une fonction lors de la compilation, consultez la documentation de votre compilateur.

Remarque – Le fournisseur pid agit de façon imprévisible lorsqu'il sonde une fonction utilisant des pointeurs de fonction pour appeler une sous-fonction. Pour analyser cette fonction, vous pouvez placer des sondes de façon explicite à l'adresse de l'entrée et du retour de la fonction.

Attribution d'un nom à des sondes pid

Le fournisseur pid définit en réalité une *classe* de fournisseurs. Chaque processus peut éventuellement disposer d'un fournisseur pid associé propre. Un processus avec l'ID 123, par exemple, pourrait être suivi à l'aide du fournisseur pid123. Pour les sondes de l'un de ces fournisseurs, la partie du module de la description de la sonde fait référence à un objet chargé dans l'espace d'adresse du processus correspondant. L'exemple suivant utilise mdb(1) pour afficher une liste d'objets :

```
$ mdb -p 1234
Loading modules: [ ld.so.1 libc.so.1 ]
> ::objects
    BASE    LIMIT     SIZE NAME
```

```
   10000    34000    24000 /usr/bin/csh
ff3c0000 ff3e8000    28000 /lib/ld.so.1
ff350000 ff37a000    2a000 /lib/libcurses.so.1
ff200000 ff2be000    be000 /lib/libc.so.1
ff3a0000 ff3a2000     2000 /lib/libdl.so.1
ff320000 ff324000     4000 /platform/sun4u/lib/libc_psr.so.1
```

Dans la description de la sonde, attribuez à l'objet le nom du fichier, et non son chemin d'accès complet. Vous pouvez également omettre le suffixe `.1` ou `so.1`. Tous les exemples suivants nomment la même sonde :

```
pid123:libc.so.1:strcpy:entry
pid123:libc.so:strcpy:entry
pid123:libc:strcpy:entry
```

Le premier exemple illustre le nom réel de la sonde. Les autres exemples sont des alias pratiques remplacés par le nom complet de l'objet de charge en interne.

Pour l'objet de charge de l'exécutable, vous pouvez utiliser l'alias `a.out`. Les deux descriptions de sonde suivantes nomment la même sonde :

```
pid123:csh:main:return
pid123:a.out:main:return
```

Comme pour toutes les sondes DTrace ancrées, le champ de fonction de la description de la sonde nomme une fonction du champ de module. Un binaire d'application utilisateur peut comporter plusieurs noms pour la même fonction. Par exemple, `mutex_lock` peut être un autre nom de la fonction `pthread_mutex_lock` dans `libc.so.1`. DTrace choisit un nom canonique pour de telles fonctions et utilise ce nom en interne. L'exemple suivant illustre comment DTrace remappe en interne les noms de module et de fonction sous forme canonique :

```
# dtrace -q -n pid101267:libc:mutex_lock:entry'{ \
    printf("%s:%s:%s:%s\n", probeprov, probemod, probefunc, probename); }'
pid101267:libc.so.1:pthread_mutex_lock:entry
^C
```

Ce renommage automatique signifie que les noms des sondes que vous activez peuvent être légèrement différents de celles réellement activées. Le nom canonique est toujours cohérent entre des exécutions de DTrace sur des systèmes utilisant la même version de Solaris.

Pour obtenir des exemples d'utilisation effective du fournisseur `pid`, reportez-vous au Chapitre33, “Suivi des processus utilisateur”.

Sondes de limite de fonction

Le fournisseur `pid` vous permet de suivre une entrée et un retour de fonction dans des programmes utilisateur tout comme le fournisseur FBT qui propose cette fonctionnalité pour le noyau. La plupart des exemples dans ce manuel qui utilisent le fournisseur FBT pour suivre des appels de fonction du noyau peuvent être légèrement modifiés pour être appliqués à des processus utilisateur.

Sondes `entry`

Une sonde `entry` se déclenche lorsque la fonction suivie est invoquée. Les arguments des sondes d'entrée sont les valeurs des arguments de la fonction suivie.

Sondes `return`

Une sonde `return` se déclenche lorsque la fonction suivie est renvoyée ou appelle une autre fonction. La valeur de `arg0` est le décalage dans la fonction de l'instruction de retour ; `arg1` contient la valeur de retour.

Remarque – L'utilisation de `arg`*N* renvoie les valeurs brutes non filtrées, comme `int64_t`. Le fournisseur `pid` ne prend pas en charge le format `args[`*N*`]`.

Sondes de décalage de fonction

Le fournisseur `pid` vous permet de suivre toute instruction d'une fonction. Par exemple, pour suivre l'instruction 4 octets dans une fonction `main()`, vous pourriez utiliser une commande similaire à l'exemple suivant :

```
pid123:a.out:main:4
```

Cette sonde est activée chaque fois que le programme exécute l'instruction à l'adresse `main+4`. Les arguments de sondes de décalage ne sont pas définis. L'ensemble `uregs[]` vous permet de connaître l'état du processus au niveau de ces sites de sonde. Pour plus d'informations, reportez-vous à la section “Tableau `uregs[]` à la page 366”.

Stabilité

Le fournisseur `pid` utilise un mécanisme de stabilité DTrace pour décrire ses stabilités, tel qu'illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	Privé	Privé	Inconnue

CHAPITRE 31

Fournisseur `plockstat`

Le fournisseur `plockstat` propose des sondes pouvant être utilisées pour observer le comportement de primitives de synchronisation au niveau utilisateur incluant des heures de contention et de maintien du verrou. La commande `plockstat`(1M) est un client DTrace utilisant le fournisseur `plockstat` pour collecter des données sur des événements de verrouillage au niveau utilisateur.

Présentation

Le fournisseur `plockstat` propose des sondes pour les types d'événement suivants :

Événements de contention — Ces sondes correspondent à la contention sur une primitive de synchronisation au niveau utilisateur, et se déclenchent lorsqu'un thread est forcé pour attendre qu'une ressource soit disponible. Solaris est généralement optimisé pour un cas de non-contention, une contention prolongée n'est donc pas prévue. Ces sondes doivent être utilisées pour comprendre les cas dans lesquels une contention se produit. La contention étant conçue pour être (relativement) rare, l'activation de sondes d'événement de contention n'entraîne généralement pas d'effet de sonde critique. Elles peuvent être activées sans se soucier de l'impact sur la performance.

Événements de maintien — Ces sondes correspondent à l'acquisition, la libération ou une autre manipulation d'une primitive de synchronisation au niveau utilisateur. En tant que telles, ces sondes peuvent être utilisées pour répondre à des questions arbitraires sur la manipulation des primitives de synchronisation au niveau utilisateur. Les applications acquérant et libérant généralement très souvent des primitives de synchronisation, l'activation de sondes d'événement de maintien peut entraîner un effet de sonde plus important que l'activation de sondes d'événement de contention.

	Alors que l'effet de sonde entraîné par leur activation peut être important, il n'est néanmoins pas pathologique. Elles peuvent toujours être activées en toute confiance sur des applications de production.
Événements d'erreur	Ces sondes correspondent à n'importe quel type de comportement anormal rencontré lors de l'acquisition ou de la libération d'une primitive de synchronisation au niveau utilisateur. Ces événements peuvent être utilisés pour détecter des erreurs rencontrées lorsqu'un thread bloque sur une primitive de synchronisation au niveau utilisateur. Les événements d'erreur doivent être extrêmement rares, leur activation ne doit ainsi pas entraîner d'effet de sonde critique.

Sondes Mutex

Les sondes *Mutex* impliquent une exclusion mutuelle de sections critiques. Lorsqu'un thread tente d'acquérir une sonde mutex détenue par un autre thread à l'aide de `mutex_lock(3C)` ou de `pthread_mutex_lock(3C)`, il détermine si le thread maître est exécuté sur une autre CPU. Si c'est le cas, le thread demandeur effectuera une brève *rotation* en attendant que la sonde mutex soit disponible. Si le propriétaire n'est pas exécuté sur une autre CPU, le thread demandeur se *bloquera*.

Les quatre sondes `plockstat` appartenant aux mutex sont répertoriées dans le Tableau 31–1. Pour chaque sonde, `arg0` contient un pointeur vers la structure `mutex_t` ou `pthread_mutex_t` (il s'agit de types identiques) représentant la mutex.

TABLEAU 31–1 Sondes Mutex

`mutex-acquire`	Sonde d'événement de maintien se déclenchant immédiatement après l'acquisition d'une mutex. `arg1` contient une valeur booléenne indiquant si l'acquisition a été récursive sur une mutex récursive. `arg2` indique le nombre d'itérations engagées par le thread demandeur pour la rotation sur cette mutex. `arg2` ne sera pas de zéro uniquement si la sonde `mutex-spin` s'est déclenchée à l'acquisition de cette mutex.
`mutex-block`	Sonde d'événement de contention se déclenchant avant qu'un thread ne bloque sur une mutex maintenue. `mutex-block` et `mutex-spin` peuvent se déclencher pour une seule acquisition de verrouillage.
`mutex-spin`	Sonde d'événement de contention se déclenchant avant qu'un thread ne commence sa rotation sur une mutex maintenue. `mutex-block` et `mutex-spin` peuvent se déclencher pour une seule acquisition de verrouillage.

TABLEAU 31–1 Sondes Mutex *(Suite)*

`mutex-release`	Sonde d'événement de maintien se déclenchant immédiatement après la libération d'une mutex. `arg1` contient une valeur booléenne indiquant si l'événement correspond à une libération récursive sur une mutex récursive.
`mutex-error`	Sonde d'événement d'erreur se déclenchant lorsqu'une erreur est rencontrée sur une opération de mutex. `arg1` est la valeur `errno` de l'erreur rencontrée.

Sondes de verrouillage en lecture/écriture

Les *verrouillages en lecture/écriture* autorisent plusieurs lecteurs *ou* un seul rédacteur, mais pas les deux, dans une section critique en même temps. Ces verrous sont généralement utilisés pour les structures plus fréquemment recherchées que modifiées, ou lorsque des threads prennent du temps dans une section critique. Les utilisateurs interagissent avec des verrous en lecture/écriture à l'aide des interfaces Solaris `rwlock(3C)` ou POSIX `pthread_rwlock_init(3C)`.

Les sondes appartenant à des verrouillages en lecture/écriture sont répertoriées dans le Tableau 31–2 Pour chaque sonde, `arg0` contient un pointeur vers la structure `rwlock_t` ou `pthread_rwlock_t` (il s'agit de types identiques) représentant le verrou adaptatif. `arg1` contient une valeur booléenne indiquant si l'opération concerne un rédacteur.

TABLEAU 31–2 Sondes de verrouillage en lecture/écriture

`rw-acquire`	Sonde d'événement de maintien se déclenchant immédiatement après l'acquisition d'un verrou en lecture/écriture.
`rw-block`	Sonde d'événement de contention se déclenchant avant qu'un thread ne bloque pendant une tentative d'acquisition de verrou. Si activée, la sonde `rw-acquire` ou la sonde `rw-error` est déclenchée après `rw-block`.
`rw-release`	Sonde d'événement de maintien se déclenchant immédiatement après la libération d'un verrou en lecture/écriture.
`rw-error`	Sonde d'événement d'erreur se déclenchant lorsqu'une erreur est rencontrée lors d'une opération de verrouillage en lecture/écriture. `arg1` est la valeur `errno` de l'erreur rencontrée.

Stabilité

Le fournisseur `plockstat` utilise un mécanisme de stabilité DTrace pour décrire ses stabilités, tel qu'illustré dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	En cours d'évolution	En cours d'évolution	ISA

CHAPITRE 32

Fournisseur fasttrap

Le fournisseur fasttrap permet le suivi d'emplacements de processus utilisateur spécifiques, préprogrammés. Contrairement à d'autres fournisseurs DTrace, fasttrap n'est pas conçu pour effectuer le suivi de l'activité du système. Ce fournisseur vise plutôt à permettre aux consommateurs DTrace d'introduire des informations dans la structure DTrace en activant la sonde fasttrap.

Sondes

Le fournisseur fasttrap rend disponible une seule sonde fasttrap:::fasttrap, qui se déclenche à chaque fois qu'un processus de niveau utilisateur lance un appel DTrace dans le noyau. L'appel DTrace permettant d'activer la sonde n'est pas disponible à l'heure actuelle.

Stabilité

Le fournisseur fasttrap utilise le mécanisme de stabilité de DTrace pour décrire ses stabilités, comme décrit dans le tableau suivant. Pour plus d'informations sur le mécanisme de stabilité, reportez-vous au Chapitre39, “Stabilité”.

Élément	Stabilité des noms	Stabilité des données	Classe de dépendance
Fournisseur	En cours d'évolution	En cours d'évolution	ISA
Module	Privé	Privé	Inconnu
Fonction	Privé	Privé	Inconnu
Nom	En cours d'évolution	En cours d'évolution	ISA
Arguments	En cours d'évolution	En cours d'évolution	ISA

CHAPITRE 33

Suivi des processus utilisateur

DTrace est un outil extrêmement puissant de compréhension du comportement des processus utilisateur. DTrace peut s'avérer précieux lors du débogage, de l'analyse des problèmes de performance ou de la simple compréhension du comportement d'une application complexe. Ce chapitre présente les fonctions de DTrace dédiées au suivi de l'activité des processus utilisateur, ainsi que des exemples illustrant leur utilisation.

Sous-routines copyin() et copyinstr()

L'interaction de DTrace avec les processus diffère légèrement de la plupart des débogueurs ou des outils d'observation classiques. La plupart de ces outils s'exécutent dans l'étendue du processus, ce qui permet aux utilisateurs de déréférencer les pointeurs pour programmer directement les variables. Plutôt que de s'exécuter dans le cadre ou au sein du processus lui-même, les sondes de DTrace s'exécutent dans le noyau de Solaris. Pour accéder aux données du processus, une sonde doit utiliser les sous-routines copyin() ou copyinstr() pour copier les données de processus utilisateur dans l'espace d'adressage du noyau.

Étudiez, par exemple, l'appel système write(2) suivant :

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

Le programme en D suivant illustre une tentative erronée d'impression du contenu d'une chaîne transmise à l'appel système write(2).

```
syscall::write:entry
{
    printf("%s", stringof(arg1)); /* incorrect use of arg1 */
}
```

Si vous essayez d'exécuter ce script, DTrace crée des messages d'erreur similaires à l'exemple suivant :

```
dtrace: error on enabled probe ID 1 (ID 37: syscall::write:entry): \
    invalid address (0x10038a000) in action #1
```

La variable arg1, contenant la valeur du paramètre *buf*, correspond à une adresse renvoyant à la mémoire dans le processus exécutant le système d'appel. Pour lire la chaîne à cette adresse, utilisez la sous-routine copyinstr() et enregistrez son résultat avec l'action printf() :

```
syscall::write:entry
{
    printf("%s", copyinstr(arg1)); /* correct use of arg1 */
```

La sortie de ce script présente toutes les chaînes dont la transmission à l'appel système write(2) est en cours. Il se peut, cependant, que vous constatiez occasionnellement des sorties irrégulières similaires à l'exemple suivant :

```
  0     37                      write:entry madaï¿½ï¿½ï¿½
```

La sous-routine copyinstr() prend en charge un argument d'entrée correspondant à l'adresse utilisateur d'une chaîne ASCII nulle terminée. Cependant, les tampons transmis à l'appel système write(2) peuvent renvoyer à des données binaires plutôt qu'à des chaînes ASCII. Pour n'imprimer que le nombre de chaînes prévu par le programme appelant, utilisez la sous-routine copyin() dont le second argument correspond à une taille :

```
syscall::write:entry
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

Notez l'utilisation obligatoire de l'opérateur stringof de sorte que DTrace convertisse correctement les données utilisateur récupérées à l'aide de copyin() vers une chaîne. L'utilisation de stringof ne s'impose pas si vous utilisez copyinstr() car cette fonction retourne toujours le type string.

Évitement des erreurs

Les sous-routines copyin() et copyinstr() ne peuvent pas lire les adresses utilisateur qui n'ont pas encore été touchées. Par conséquent, même une adresse valide peut provoquer une erreur si la page contenant cette adresse n'a pas déjà subi de défaillance lors de l'accès. Examinez l'exemple suivant :

```
# dtrace -n syscall::open:entry'{ trace(copyinstr(arg0)); }'
dtrace: description 'syscall::open:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 2 (ID 50: syscall::open:entry): invalid address
(0x9af1b) in action #1 at DIF offset 52
```

Dans la sortie de l'exemple ci-dessus, l'application fonctionnait correctement et l'adresse dans arg0 était valide mais renvoyait à une page à laquelle le processus correspondant n'avait pas encore accédé. Pour résoudre ce problème, attendez que le noyau ou l'application utilisent les données avant de procéder au suivi. Attendez, par exemple, le retour de l'appel système pour appliquer copyinstr(), comme illustré dans l'exemple suivant :

```
# dtrace -n syscall::open:entry'{ self->file = arg0; }' \
-n syscall::open:return'{ trace(copyinstr(self->file)); self->file = 0; }'
dtrace: description 'syscall::open:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  2     51                      open:return   /dev/null
```

Suppression de l'interférence de dtrace(1M)

Si vous suivez chaque appel vers l'appel système write(2), vous allez créer une cascade de sortie. Suite à chaque appel de la fonction write(), la commande dtrace(1M) appelle la fonction write() lorsqu'elle affiche la sortie. Cette boucle d'évaluation est un bon exemple de la manière dont la commande dtrace peut interférer avec les données souhaitées. Vous pouvez utiliser un simple prédicat pour empêcher le suivi des données non désirées :

```
syscall::write:entry
/pid != $pid/
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

La variable de macro $pid développe l'identifiant de processus du processus qui a activé les sondes. La variable pid contient l'identifiant de processus du processus dont le thread est exécuté sur la CPU sur laquelle la sonde a été déclenchée. Par conséquent, le prédicat /pid != $pid/ garantit que le script n'assure le suivi d'aucun événement lié à l'exécution de ce script.

Fournisseur syscall

Le fournisseur syscall vous permet de suivre chaque entrée et retour d'appel système. Les appels système peuvent constituer un excellent point de départ à la compréhension du comportement d'un processus, notamment si le temps d'exécution ou de blocage de ce dernier dans le noyau est considérable. Vous pouvez utiliser la commande prstat(1M) pour déterminer à quelles activités ces processus consacrent du temps :

```
$ prstat -m -p 31337
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/NLWP
 13499 user1     53  44 0.0 0.0 0.0 0.0 2.5 0.0  4K  24  9K   0 mystery/6
```

Cet exemple montre que le processus utilise un temps système considérable. Ce comportement peut notamment s'expliquer par le fait que ce processus exécute un grand nombre d'appels

système. Vous pouvez utiliser un programme en D simple spécifié sur la ligne de commande pour déterminer les appels système les plus fréquents :

```
# dtrace -n syscall:::entry'/pid == 31337/{ @syscalls[probefunc] = count(); }'
dtrace: description 'syscall:::entry' matched 215 probes
^C

  open                                                              1
  lwp_park                                                          2
  times                                                             4
  fcntl                                                             5
  close                                                             6
  sigaction                                                         6
  read                                                             10
  ioctl                                                            14
  sigprocmask                                                     106
  write                                                          1092
```

Ce rapport indique les appels systèmes les plus fréquents. Dans cet exemple, il s'agit de l'appel système write(2). Vous pouvez utiliser le fournisseur syscall pour examiner plus attentivement la source de tous les appels système write() :

```
# dtrace -n syscall::write:entry'/pid == 31337/{ @writes[arg2] = quantize(arg2); }'
dtrace: description 'syscall::write:entry' matched 1 probe
^C

           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  1037
               2 |@                                        3
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |@                                        3
              64 |                                         0
             128 |                                         0
             256 |                                         0
             512 |                                         0
            1024 |@                                        5
            2048 |                                         0
```

La sortie indique que le processus exécute de nombreux appels système write() avec une quantité de données relativement faible. Ce ratio pourrait bien être à l'origine du problème de performances de ce processus particulier. Cet exemple illustre une méthodologie générale d'étude du comportement d'un appel système.

Action de la fonction ustack()

Il est souvent utile de suivre une pile de thread de processus au moment précis de l'activation d'une sonde pour examiner un problème plus en détails. L'action de la fonction `ustack()` suit la pile du thread utilisateur. Si, par exemple, un processus ouvrant plusieurs fichiers rencontre occasionnellement une défaillance dans l'appel système open(2), vous pouvez utiliser l'action de la fonction `ustack()` pour rechercher le chemin d'accès au code exécutant la fonction `open()` en échec :

```
syscall::open:entry
/pid == $1/
{
    self->path = copyinstr(arg0);
}

syscall::open:return
/self->path != NULL && arg1 == -1/
{
    printf("open for '%s' failed", self->path);
    ustack();
}
```

Ce script illustre également l'utilisation de la variable de macro `$1` qui récupère la valeur du premier opérande spécifié sur la ligne de commande de `dtrace`(1M).

```
# dtrace -s ./badopen.d 31337
dtrace: script './badopen.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0     40                      open:return open for '/usr/lib/foo' failed
              libc.so.1`__open+0x4
              libc.so.1`open+0x6c
              420b0
              tcsh`dosource+0xe0
              tcsh`execute+0x978
              tcsh`execute+0xba0
              tcsh`process+0x50c
              tcsh`main+0x1d54
              tcsh`_start+0xdc
```

L'action de la fonction `ustack()` enregistre les valeurs (PC) du compteur du programme pour la pile et `dtrace`(1M) résout ces valeurs PC pour symboliser les noms en effectuant une recherche dans les tables des symboles du processus. Si `dtrace` ne parvient pas à retrouver le symbole d'une valeur PC, cette dernière est imprimée sous la forme d'un entier hexadécimal.

Si un processus se termine ou est avorté avant le formatage final des données de la fonction `ustack()`, `dtrace` risque de ne pas pouvoir convertir les valeurs PC du suivi de pile en noms de symbole et devra les afficher sous forme de nombres hexadécimaux. Pour contourner cette

restriction, spécifiez un processus qui vous intéresse avec l'option -dtrace -c ou p. Pour plus d'informations sur ces options (et bien d'autres), reportez-vous au Chapitre14, “Utilitaire dtrace(1M)”. En cas d'ignorance préalable de la commande ou de l'ID de processus, vous pouvez utiliser l'exemple suivant de programme en D pour contourner cette restriction :

```
/*
 * This example uses the open(2) system call probe, but this technique
 * is applicable to any script using the ustack() action where the stack
 * being traced is in a process that may exit soon.
 */
 syscall::open:entry
{
    ustack();
    stop_pids[pid] = 1;
}

syscall::rexit:entry
/stop_pids[pid] != 0/
{
    printf("stopping pid %d", pid);
    stop();
    stop_pids[pid] = 0;
}
```

Le script ci-dessus interrompt un processus juste avant qu'il ne se termine si l'action de la fonction ustack() a été appliquée à un thread dans ce processus. Grâce à cette technique, la commande dtrace peut résoudre les valeurs PC en noms symboliques. Notez que la valeur de stop_pids[pid] est réglée sur 0 après avoir servi à supprimer la variable dynamique. N'oubliez pas de configurer la réexécution des processus interrompus à l'aide de la commande prun(1). Dans le cas contraire, votre système accumulera de nombreux processus interrompus.

Tableau uregs[]

Le tableau uregs[] vous permet d'accéder aux enregistrements utilisateur individuels. Les tables suivantes répertorient les index dans le tableau uregs[] correspondant à chaque architecture système Solaris prise en charge.

TABLEAU 33–1 Constantes uregs[] pour SPARC

Constante	Enregistrer
R_G0..R_G7	Enregistrements globaux %g0..%g7
R_O0..R_O7	Enregistrements en sortie %o0..%o7

TABLEAU 33–1 Constantes uregs[] pour SPARC *(Suite)*

Constante	Enregistrer
R_L0..R_L7	Enregistrements locaux %l0..%l7
R_I0..R_I7	Enregistrements en entrée %i0..%i7
R_CCR	Enregistrement du code de condition %ccr
R_PC	Compteur de programme %pc
R_NPC	Compteur de programme suivant %npc
R_Y	Multiplier/diviser l'enregistrement %y
R_ASI	Enregistrement de l'identificateur d'espace d'adressage %asi
R_FPRS	État des enregistrements en virgule flottante %fprs

TABLEAU 33–2 Constantes de uregs[] pour x86

Constante	Enregistrer
R_CS	%cs
R_GS	%gs
R_ES	%es
R_DS	%ds
R_EDI	%edi
R_ESI	%esi
R_EBP	%ebp
R_EAX	%eax
R_ESP	%esp
R_EAX	%eax
R_EBX	%ebx
R_ECX	%ecx
R_EDX	%edx
R_TRAPNO	%trapno
R_ERR	%err
R_EIP	%eip

TABLEAU 33–2 Constantes de uregs[] pour x86 *(Suite)*

Constante	Enregistrer
R_CS	%cs
R_ERR	%err
R_EFL	%efl
R_UESP	%uesp
R_SS	%ss

Sur les plates-formes AMD64, le tableau uregs possède le même contenu que sur les plates-formes x86, plus les éléments supplémentaires répertoriés dans la table suivante :

TABLEAU 33–3 Constantes uregs[] pour amd64

Constante	Enregistrer
R_RSP	%rsp
R_RFL	%rfl
R_RIP	%rip
R_RAX	%rax
R_RCX	%rcx
R_RDX	%rdx
R_RBX	%rbx
R_RBP	%rbp
R_RSI	%rsi
R_RDI	%rdi
R_R8	%r8
R_R9	%r9
R_R10	%r10
R_R11	%r11
R_R12	%r12
R_R13	%r13
R_R14	%r14
R_R15	%r15

Les alias répertoriés dans la table suivante sont utilisables sur toutes les plates-formes :

TABLEAU 33–4 Constantes `uregs[]` communes

Constante	Enregistrer
R_PC	Enregistrement du compteur de programme
R_SP	Enregistrement du pointeur de pile
R_R0	Premier code de retour
R_R1	Second code de retour

Fournisseur pid

Le fournisseur `pid` vous permet de suivre toutes les instructions d'un processus. Contrairement à la plupart des autres fournisseurs, les sondes `pid` sont créées à la demande en fonction des descriptions de sonde figurant dans vos programmes en D. En conséquence, aucune sonde `pid` n'est répertoriée dans la sortie de `dtrace -l` tant que vous ne les activez pas vous-même.

Suivi de la limite de fonction utilisateur

Le mode de fonctionnement le plus simple du fournisseur `pid` est, tout comme l'espace utilisateur, analogue au fournisseur `fbt.` L'exemple de programme suivant suit toutes les entrées et tous les renvois de fonction effectués à partir d'une fonction simple. La variable de macro `$1` (le premier opérande sur la ligne de commande) correspond à l'ID de processus du processus à suivre. La variable de macro `$2` (le second opérande sur la ligne de commande) correspond au nom de la fonction à partir de laquelle les appels de fonction doivent être suivis.

EXEMPLE 33–1 `userfunc.d` : suivi des entrées et renvois de la fonction utilisateur

```
pid$1::$2:entry
{
    self->trace = 1;
}

pid$1::$2:return
/self->trace/
{
    self->trace = 0;
}

pid$1:::entry,
pid$1:::return
/self->trace/
```

EXEMPLE 33–1 `userfunc.d` : suivi des entrées et renvois de la fonction utilisateur *(Suite)*

```
{
}
```

Saisissez l'exemple de script ci-dessus et enregistrez-le dans le fichier `userfunc.d`, puis appliquez-lui la commande `chmod` pour le rendre exécutable. Ce script engendre une sortie similaire à l'exemple suivant :

```
# ./userfunc.d 15032 execute
dtrace: script './userfunc.d' matched 11594 probes
  0  -> execute
  0    -> execute
  0      -> Dfix
  0      <- Dfix
  0      -> s_strsave
  0        -> malloc
  0        <- malloc
  0      <- s_strsave
  0      -> set
  0        -> malloc
  0        <- malloc
  0      <- set
  0      -> set1
  0        -> tglob
  0        <- tglob
  0      <- set1
  0      -> setq
  0        -> s_strcmp
  0        <- s_strcmp
...
```

Vous ne pouvez utiliser le fournisseur `pid` que sur les processus dont l'exécution est déjà en cours. Vous pouvez utiliser la variable de macro `$target` (reportez-vous au Chapitre15, “Scripts”) et les options `dtrace -c` et `-p` pour créer et extraire les processus qui vous intéressent, puis les instrumenter à l'aide de DTrace. Par exemple, vous pouvez utiliser l'exemple de script en D suivant pour déterminer la répartition des appels de fonction exécutés vers `libc` par un processus sujet particulier :

```
pid$target:libc.so::entry
{
    @[probefunc] = count();
}
```

Pour déterminer la répartition des appels exécutés par la commande `date(1)`, enregistrez le script dans le fichier `libc.d` et exécutez la commande suivante :

```
# dtrace -s libc.d -c date
dtrace: script 'libc.d' matched 2476 probes
Fri Jul 30 14:08:54 PDT 2004
dtrace: pid 109196 has exited

  pthread_rwlock_unlock                                             1
  _fflush_u                                                         1
  rwlock_lock                                                       1
  rw_write_held                                                     1
  strftime                                                          1
  _close                                                            1
  _read                                                             1
  __open                                                            1
  _open                                                             1
  strstr                                                            1
  load_zoneinfo                                                     1

...
  _ti_bind_guard                                                   47
  _ti_bind_clear                                                   94
```

Suivi des instructions arbitraires

Vous pouvez utiliser le fournisseur pid pour suivre une instruction dans une fonction utilisateur. Le fournisseur pid crée une sonde à la demande pour chaque instruction d'une fonction. Le nom de chaque sonde correspond au décalage de l'instruction correspondante dans la fonction exprimée sous la forme d'un entier hexadécimal. Par exemple, pour activer une sonde associée à l'instruction au niveau du décalage 0x1c dans la fonction foo du module bar.so dans le processus avec PID 123, vous pouvez utiliser la commande suivante :

```
# dtrace -n pid123:bar.so:foo:1c
```

Pour activer toutes les sondes de la fonction foo, y compris la sonde de chaque instruction, vous pouvez utiliser la commande :

```
# dtrace -n pid123:bar.so:foo:
```

Cette commande fait preuve d'une technique extrêmement puissante de débogage et d'analyse des applications utilisateur. Il peut s'avérer difficile de déboguer les erreurs peu fréquentes car elles peuvent être difficiles à reproduire. En règle générale, vous pouvez identifier un problème après l'apparition de la défaillance, soit trop tard pour reconstruire le chemin d'accès au code. L'exemple suivant montre la méthode de combinaison du fournisseur pid avec le suivi spéculatif (reportez-vous au Chapitre13, “Suivi spéculatif”) pour résoudre ce problème en suivant chaque instruction d'une fonction.

EXEMPLE 33–2 `errorpath.d` : suivi du chemin d'accès au code d'une fonction utilisateur

```
pid$1::$2:entry
{
    self->spec = speculation();
    speculate(self->spec);
    printf("%x %x %x %x %x", arg0, arg1, arg2, arg3, arg4);
}

pid$1::$2:
/self->spec/
{
    speculate(self->spec);
}

pid$1::$2:return
/self->spec && arg1 == 0/
{
    discard(self->spec);
    self->spec = 0;
}

pid$1::$2:return
/self->spec && arg1 != 0/
{
    commit(self->spec);
    self->spec = 0;
}
```

L'exécution de `errorpath.d` produit une sortie similaire à l'exemple suivant :

```
# ./errorpath.d 100461 _chdir
dtrace: script './errorpath.d' matched 19 probes
CPU     ID                    FUNCTION:NAME
  0  25253                     _chdir:entry 81e08 6d140 ffbfcb20 656c73 0
  0  25253                     _chdir:entry
  0  25269                         _chdir:0
  0  25270                         _chdir:4
  0  25271                         _chdir:8
  0  25272                         _chdir:c
  0  25273                        _chdir:10
  0  25274                        _chdir:14
  0  25275                        _chdir:18
  0  25276                        _chdir:1c
  0  25277                        _chdir:20
  0  25278                        _chdir:24
  0  25279                        _chdir:28
```

```
0  25280                      _chdir:2c
0  25268                    _chdir:return
```

◆ ◆ ◆ CHAPITRE 34

Suivi défini statiquement pour les applications utilisateur

DTrace offre aux développeurs d'applications utilisateur une fonction permettant de définir des sondes personnalisées en code d'application pour augmenter les capacités du fournisseur `pid`. Ces sondes statistiques imposent peu, voire aucun temps système lorsqu'elles sont désactivées. Par ailleurs, elles sont activées dynamiquement comme les autres sondes DTrace. Vous pouvez utiliser des sondes statiques pour décrire la sémantique des applications aux utilisateurs de DTrace sans faire preuve ni nécessiter de connaissance en matière d'implémentation de vos applications. Ce chapitre décrit la procédure de définition de sondes statiques dans les applications utilisateur et d'utilisation de DTrace pour activer ces sondes dans des processus utilisateur.

Choix des points de sonde

DTrace permet aux développeurs d'intégrer des points de sonde statiques dans du code d'application, y compris des applications complètes et des bibliothèques partagées. Vous pouvez activer ces sondes peu importe l'endroit où l'application ou la bibliothèque est exécutée, que ce soit en développement ou en production. Vous devez définir des sondes dont la signification sémantique est facilement compréhensible par les utilisateurs de DTrace de votre entreprise. Par exemple, vous pouvez définir les sondes `query-receive` et `query-respond` pour un serveur Web, ces sondes correspondant à un client soumettant une demande et à la réponse du serveur Web. Ces exemples de sondes sont facilement compréhensibles par la plupart des utilisateurs de DTrace et correspondent au niveau d'abstraction le plus élevé de l'application et non aux détails d'implémentation de niveau inférieur. Les utilisateurs de DTrace peuvent utiliser ces sondes pour comprendre la répartition des demandes dans le temps. Si la sonde `query-receive` présente les chaînes de demande URL comme un argument, un utilisateur de DTrace peut déterminer les demandes qui génèrent le plus d'E/S disque en combinant cette sonde au fournisseur `io`.

Vous devez également prendre en considération la stabilité des abstractions que vous décrivez en choisissant les noms et les emplacements des sondes. Cette sonde sera-t-elle encore présente dans les versions à venir de l'application même en cas de changements d'implémentation ? Cette

sonde a-t-elle une utilité sur toutes les architectures système ou est-elle spécifique à un jeu d'instructions particulier ? Ce chapitre décrit en détails la manière dont ces décisions déterminent vos définitions de suivi des statistiques.

Ajout de sondes à une application

Les sondes DTrace dédiées aux bibliothèques et aux exécutables sont définies dans une section ELF du code binaire applicatif correspondant. Cette section décrit comment définir vos sondes, les ajouter au code source de votre application et augmenter le processus de construction de votre application pour intégrer les définitions des sondes DTrace.

Définition des fournisseurs et des sondes

Vous définissez les sondes DTrace dans un fichier source `.d` utilisé par la suite pour compiler et lier votre application. Commencez par attribuer un nom approprié au fournisseur de l'application utilisateur. Le nom de fournisseur que vous choisissez sera ajouté à un identificateur de processus pour chaque processus exécutant votre code d'application. Par exemple, si vous avez donné à un serveur Web exécuté sous l'ID de processus 1203 le nom de fournisseur `myserv`, le nom du fournisseur DTrace correspondant à ce processus sera `myserv1203`. Dans le fichier source `.d`, définissez le fournisseur comme dans l'exemple suivant :

```
provider myserv {
    ...
};
```

Ajoutez ensuite une définition pour chaque sonde et les arguments correspondants. L'exemple suivant définit les deux sondes présentées dans “Choix des points de sonde” à la page 375. La première sonde possède deux arguments de type `string` tandis que la seconde n'en possède aucun. Le compilateur D convertit deux traits de soulignement consécutifs (`__`) en un tiret (`-`) dans le nom d'une sonde.

```
provider myserv {
    probe query__receive(string, string);
    probe query__respond();
};
```

Vous devez ajouter des attributs de stabilité à la définition de votre fournisseur pour que les utilisateurs de vos sondes sachent qu'il est probable que des changements interviennent dans les versions ultérieures de votre application. Pour plus d'informations sur les attributs de stabilité DTrace, reportez-vous au Chapitre39, “Stabilité”. Les attributs de stabilité sont définis comme illustré dans l'exemple suivant :

EXEMPLE 34–1 `myserv.d` : sondes d'application définies statiquement

```
#pragma D attributes Evolving/Evolving/Common provider myserv provider
#pragma D attributes Private/Private/Unknown provider myserv module
#pragma D attributes Private/Private/Unknown provider myserv function
#pragma D attributes Evolving/Evolving/Common provider myserv name
#pragma D attributes Evolving/Evolving/Common provider myserv args

provider myserv {
    probe query__receive(string, string);
    probe query__respond();
};
```

Remarque – Les scripts en D utilisant des arguments non entiers à partir de sondes ajoutées par l'utilisateur doivent avoir recours aux fonctions `copyin()` et `copyinstr()` pour récupérer ces arguments. Pour plus d'informations, reportez-vous au Chapitre33, “Suivi des processus utilisateur”.

Ajout de sondes à un code d'application

Une fois les sondes définies dans un fichier `.d`, vous devez augmenter votre code source pour indiquer leur position de déclenchement. Examinez l'exemple suivant de code source d'application C :

```
void
main_look(void)
{
    ...
    query = wait_for_new_query();
    process_query(query)
    ...
}
```

Pour ajouter un site de sondes, ajoutez une référence à la macro `DTRACE_PROBE()` définie dans `<sys/sdt.h>`, comme illustré dans l'exemple suivant :

```
#include <sys/sdt.h>
...

void
main_look(void)
{
    ...
    query = wait_for_new_query();
```

```
    DTRACE_PROBE2(myserv, query__receive, query->clientname, query->msg);
    process_query(query)
    ...
}
```

Le suffixe 2 du nom de la macro DTRACE_PROBE2 indique le nombre d'arguments transmis à la sonde. Les deux premiers arguments de la macro de la sonde correspondent au nom du fournisseur et au nom de la sonde. De ce fait, ils doivent être identiques à la définition de votre sonde et de votre fournisseur de langage D. Les arguments de macro restants sont ceux assignés aux variables arg0..9 de DTrace lors du déclenchement de la sonde. Le code source de votre application peut contenir plusieurs références au même nom de fournisseur et de sonde. Si plusieurs références à la même sonde figurent dans votre code source, elles sont toutes susceptibles de déclencher la sonde.

Création d'applications avec des sondes

Vous devez augmenter le processus de construction de votre application pour intégrer les définitions du fournisseur DTrace et des sondes. Un processus de construction type compile chaque fichier source de manière à créer un fichier objet correspondant. Les fichiers objets compilés sont ensuite liés entre eux pour créer le code d'application binaire terminé, comme illustré dans l'exemple suivant :

```
cc -c src1.c
cc -c src2.c
...
cc -o myserv src1.o src2.o ...
```

Pour intégrer les définitions de sonde DTrace à votre application, ajoutez les règles Makefile appropriées à votre processus de construction pour exécuter la commande dtrace, comme illustré dans l'exemple suivant :

```
cc -c src1.c
cc -c src2.c
...
dtrace -G -32 -s myserv.d src1.o src2.o ...
cc -o myserv myserv.o src1.o src2.o ...
```

La commande dtrace présentée ci-dessus assure le posttraitement des fichiers objets générés par les commandes précédentes du compilateur et génère le fichier objet myserv.o à partir de myserv.d et des autres fichiers objets. L'option dtrace -G permet de lier à une application utilisateur les définitions du fournisseur et des sondes. L'option -32 permet de créer des codes binaires d'application 32 bits. L'option -64 permet de créer des codes binaires d'application 64 bits.

CHAPITRE 35

Sécurité

Ce chapitre décrit les privilèges dont peuvent user les administrateurs système pour permettre à des utilisateurs ou des processus particuliers d'accéder à DTrace. DTrace assure la visibilité de tous les aspects du système, y compris les fonctions de niveau utilisateur, les appels système, les fonctions du noyau, etc. et repose sur de puissantes actions dont certaines peuvent modifier l'état d'un programme. Comme il ne serait pas approprié d'accorder un accès utilisateur aux fichiers privés d'un autre utilisateur, un administrateur système ne doit pas autoriser chaque utilisateur à accéder à toutes les fonctionnalités de DTrace. Par défaut, seul un superutilisateur peut utiliser DTrace. Vous pouvez utiliser la fonction Privilège minimum pour permettre à d'autres utilisateurs d'utiliser DTrace de manière contrôlée.

Privilèges

La fonction Privilège minimum de Solaris permet aux administrateurs d'accorder des privilèges spécifiques à des utilisateurs particuliers de Solaris. Pour accorder à un utilisateur un privilège à la connexion, insérez dans le fichier `/etc/user_attr` une ligne similaire à l'exemple suivant :

user-name`::::defaultpriv=basic,`*privilege*

Pour accorder à un processus en cours d'exécution un privilège supplémentaire, utilisez la commande `ppriv(1)` :

`# ppriv -s A+`*privilege process-ID*

Les trois privilèges qui contrôlent un accès utilisateur aux fonctionnalités de DTrace sont `dtrace_proc`, `dtrace_user` et `dtrace_kernel`. Chaque privilège permet d'utiliser un certain ensemble de fournisseurs, actions et variables de DTrace qui correspondent tous à un type particulier d'utilisation de DTrace. Les modes de privilèges sont décrits en détails dans les sections suivantes. Les administrateurs système doivent évaluer soigneusement les besoins de chaque utilisateur par rapport à la visibilité et à l'incidence sur les performances des différents modes de privilèges. Les utilisateurs doivent disposer d'au moins l'un des trois privilèges de DTrace pour pouvoir en utiliser les fonctionnalités.

Utilisation privilégiée de DTrace

Les utilisateurs dotés de l'un des trois privilèges de DTrace peuvent activer les sondes fournies par le fournisseur `dtrace` (voir le Chapitre17, “Fournisseur `dtrace`”) et peuvent utiliser les actions et les variables suivantes :

Fournisseurs	dtrace		
Actions	exit	printf	tracemem
	discard	speculate	
	printa	trace	
Variables	args	probemod	this
	epid	probename	timestamp
	id	probeprov	vtimestamp
	probefunc	self	
Espaces d'adressage	Aucune		

Privilège `dtrace_proc`

Le privilège `dtrace_proc` permet d'utiliser le fournisseur `fasttrap` pour assurer le suivi au niveau des processus. Il permet également d'utiliser les actions et variables suivantes :

Actions	copyin	copyout	stop
	copyinstr	raise	ustack
Variables	execname	pid	uregs
Espaces d'adressage	Utilisateur		

Ce privilège n'accorde aucune visibilité aux structures de données du noyau de Solaris ou aux processus pour lesquels l'utilisateur ne possède aucun droit.

Les utilisateurs possédant ce privilège peuvent créer et activer des sondes dans les processus qu'ils détiennent. Si l'utilisateur dispose également du privilège `proc_owner`, il est possible de créer et d'activer des sondes dans n'importe quel processus. Le privilège `dtrace_proc` est dédié aux utilisateurs intéressés par le débogage ou l'analyse des performances des processus utilisateur. Ce privilège convient parfaitement à un développeur qui travaille sur une nouvelle application ou à un ingénieur qui essaie d'améliorer les performances d'une application dans un environnement de production.

Remarque – Les utilisateurs possédant les privilèges dtrace_proc et proc_owner peuvent *activer* n'importe quelle sonde pid depuis n'importe quel processus. Par contre, ils ne peuvent créer de sondes que dans les processus dont l'ensemble de privilèges est un sous-ensemble de leur propre ensemble de privilèges. Reportez-vous à la documentation sur le Privilège minimum pour plus d'informations.

Le privilège dtrace_proc permet un accès à DTrace tout en imposant une pénalité de performances sur les processus pour lesquels l'utilisateur possède des droits uniquement. Les processus instrumentés imposent une charge supplémentaire sur les ressources système et, de ce fait, il peut en résulter une légère incidence sur les performances globales du système. Hormis cette augmentation de la charge globale, ce privilège n'offre aucune instrumentation se répercutant sur les performances des processus autres que les processus suivis. Comme ce privilège n'accorde aux utilisateurs aucune visibilité supplémentaire sur les autres processus ou le noyau lui-même, il est recommandé d'octroyer ce privilège à tous les utilisateurs qui pourraient nécessiter une meilleure compréhension des tâches internes de leurs propres processus.

Privilège dtrace_user

Le privilège dtrace_user permet l'utilisation des fournisseurs profile et syscall avec certaines remarques, ainsi que des actions et variables suivantes :

Fournisseurs	profil	syscall	fasttrap
Actions	copyin	copyout	stop
	copyinstr	raise	ustack
Variables	execname	pid	uregs
Espaces d'adressage	Utilisateur		

Le privilège dtrace_user n'offre une visibilité que sur les processus pour lesquels l'utilisateur possède déjà des droits et n'offre aucune visibilité sur l'activité ou l'état du noyau. Avec ce privilège, les utilisateurs peuvent activer le fournisseur syscall mais les sondes activées ne s'activeront que dans les processus pour lesquels l'utilisateur possède des droits. Il est possible, de la même manière, d'activer le fournisseur profile mais les sondes activées ne s'activeront que dans les processus pour lesquels l'utilisateur possède des droits et jamais dans le noyau de Solaris.

Ce privilège permet l'utilisation d'une instrumentation qui, tout en offrant uniquement une visibilité dans des processus particuliers, peut affecter les performances globales du système. Le fournisseur syscall a une faible incidence sur les performances de chaque appel système de

chaque processus. Le fournisseur `profile` affecte les performances globales du système en s'exécutant à intervalle régulier comme une horloge en temps réel. Aucune de ces baisses de performances n'est suffisamment importante pour limiter sérieusement la progression du système mais les administrateurs système doivent prendre en compte les implications de l'octroi de ce privilège à un utilisateur. Pour plus d'informations relatives à l'incidence sur les performances des fournisseurs `syscall` et `profile`, reportez-vous au Chapitre21, “Fournisseur `syscall`” et au Chapitre19, “Fournisseur `profile`”.

Privilège dtrace_kernel

Le privilège `dtrace_kernel` permet l'utilisation de tous les fournisseurs à l'exception de `pid` et `fasttrap` sur les processus que l'utilisateur ne détient pas. Ce privilège permet également d'utiliser toutes les actions et variables à l'exception des actions détruisant le noyau (`breakpoint()`, `panic()`, `chill ()`). Ce privilège offre une visibilité complète du noyau et de l'état des utilisateurs. Les fonctions qu'active le privilège `dtrace_user` constituent un sous-ensemble strict des fonctions activées par `dtrace_kernel`.

Fournisseurs	Tous avec les restrictions ci-dessus	
Actions	Toutes sauf les actions destructrices	
Variables	Tous	
Espaces d'adressage	Utilisateur	Noyau

Privilèges de superutilisateur

Un utilisateur possédant tous les privilèges peut utiliser tous les fournisseurs et toutes les actions, y compris celles qui détruisent le noyau et sont indisponibles à toutes les autres classes d'utilisateur.

Fournisseurs	Tous	
Actions	Toutes y compris les actions destructrices	
Variables	Tous	
Espaces d'adressage	Utilisateur	Noyau

CHAPITRE 36

Suivi anonyme

Ce chapitre décrit le suivi *anonyme*, c'est-à-dire le suivi qui n'est associé à aucun consommateur DTrace. Le suivi anonyme est utilisé dans les situations dans lesquelles aucun processus de consommateur DTrace ne peut s'exécuter. Le suivi anonyme est le plus souvent utilisé pour permettre aux développeurs de pilotes de périphériques de déboguer et de suivre les activités qui surviennent pendant l'initialisation du système. Tous les suivis pouvant être effectués de manière interactive peuvent l'être de manière anonyme. Toutefois, seul le superutilisateur peut créer une activation anonyme et il ne peut en exister qu'une à la fois.

Activations anonymes

Pour créer une activation anonyme, utilisez l'option `-A` avec un appel `dtrace(1M)` spécifiant les sondes, prédicats, actions et options requis. `dtrace` ajoute une série de propriétés de pilote correspondant à votre requête à la configuration du pilote `dtrace(7D)`, généralement `/kernel/drv/dtrace.conf`. Ces propriétés sont lues par le pilote `dtrace(7D)` lors de son chargement. Le pilote active les sondes spécifiées et crée un *état anonyme* à associer avec la nouvelle activation. Habituellement, le pilote `dtrace(7D)` est chargé à la demande, à l'instar de tous les pilotes agissant en qualité de fournisseurs DTrace. Pour autoriser le suivi pendant l'initialisation, le pilote `dtrace(7D)` doit être chargé le plus tôt possible. `dtrace` ajoute les instructions `forceload` nécessaires à `/etc/system` (consultez `system(4)`) pour chaque fournisseur DTrace et pour `dtrace(7D)` lui-même.

Ensuite, lors de l'initialisation du système, un message est émis par `dtrace(7D)` pour indiquer que le fichier de configuration a été traité correctement.

Toutes les options peuvent être définies par une activation anonyme, notamment la taille du tampon, la taille de variable dynamique, la taille de spéculation, le nombre de spéculations, etc.

Pour supprimer une activation anonyme, spécifiez `-A` dans `dtrace` sans description de sonde.

Demande d'état anonyme

Dès que la machine est complètement initialisée, n'importe quel état anonyme peut être demandé en spécifiant l'option `-a` avec `dtrace`. Par défaut, `-a` demande l'état anonyme, traite les données existantes et poursuit l'exécution. Pour consommer l'état anonyme puis quitter le programme, ajoutez l'option `-e`.

Dès que l'état anonyme est consommé depuis le noyau, il ne peut pas être remplacé : les tampons internes au noyau qui le contiennent sont réutilisés. En cas de tentative d'une demande d'état de suivi anonyme lorsqu'il n'en existe aucun,`dtrace` génère un message similaire à l'exemple suivant :

```
dtrace: could not enable tracing: No anonymous tracing state
```

Si des abandons ou des erreurs se sont produits, `dtrace` génère les messages appropriés lors de la demande d'état anonyme. Les messages d'abandons et d'erreurs sont les mêmes qu'il s'agisse d'un état anonyme ou non.

Exemples de suivi anonyme

L'exemple suivant illustre une activation anonyme DTrace pour chaque sonde du module `iprb(7D)` :

```
# dtrace -A -m iprb
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
# reboot
```

Au terme de la réinitialisation, `dtrace(7D)` affiche un message sur la console permettant d'activer les sondes spécifiées.

```
  ...
  Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.
  Use is subject to license terms.
  NOTICE: enabling probe 0 (:iprb::)
  NOTICE: enabling probe 1 (dtrace:::ERROR)
  configuring IPv4 interfaces: iprb0.
  ...
```

Lors de la réinitialisation de la machine, l'état anonyme peut être consommé en spécifiant l'option `-a` avec `dtrace` :

```
# dtrace -a
  CPU     ID                    FUNCTION:NAME
    0  22954                      _init:entry
```

```
    0  22955                     _init:return
    0  22800                   iprbprobe:entry
    0  22934           iprb_get_dev_type:entry
    0  22935          iprb_get_dev_type:return
    0  22801                  iprbprobe:return
    0  22802                  iprbattach:entry
    0  22874               iprb_getprop:entry
    0  22875              iprb_getprop:return
    0  22934           iprb_get_dev_type:entry
    0  22935          iprb_get_dev_type:return
    0  22870             iprb_self_test:entry
    0  22871            iprb_self_test:return
    0  22958            iprb_hard_reset:entry
    0  22959           iprb_hard_reset:return
    0  22862        iprb_get_eeprom_size:entry
    0  22826              iprb_shiftout:entry
    0  22828            iprb_raiseclock:entry
    0  22829           iprb_raiseclock:return
  ...
```

L'exemple suivant se concentre uniquement sur les fonctions appelées depuis `iprbattach()`. Dans un éditeur, tapez le script suivant et enregistrez-le dans un fichier appelé `iprb.d`.

```
fbt::iprbattach:entry
{
    self->trace = 1;
}

fbt:::
/self->trace/
{}

fbt::iprbattach:return
{
    self->trace = 0;
}
```

Exécutez les commandes suivantes pour effacer les paramètres précédents du fichier de configuration du pilote, installez la requête de suivi anonyme et réinitialisez.

```
# dtrace -AFs iprb.d
dtrace: cleaned up old anonymous enabling in /kernel/drv/dtrace.conf
dtrace: cleaned up forceload directives in /etc/system
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
# reboot
```

À l'issue de la réinitialisation, dtrace(7D) affiche un message différent sur la console pour indiquer l'activation légèrement différente :

```
...
Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.
Use is subject to license terms.
NOTICE: enabling probe 0 (fbt::iprbattach:entry)
NOTICE: enabling probe 1 (fbt:::)
NOTICE: enabling probe 2 (fbt::iprbattach:return)
NOTICE: enabling probe 3 (dtrace:::ERROR)
configuring IPv4 interfaces: iprb0.
...
```

Une fois que la machine est complètement réinitialisée, exécutez dtrace avec l'option -a et l'option -e pour consommer les données anonymes puis quitter le programme.

```
# dtrace -ae
  CPU FUNCTION
    0  -> iprbattach
    0    -> gld_mac_alloc
    0      -> kmem_zalloc
    0        -> kmem_cache_alloc
    0          -> kmem_cache_alloc_debug
    0            -> verify_and_copy_pattern
    0            <- verify_and_copy_pattern
    0            -> tsc_gethrtime
    0            <- tsc_gethrtime
    0            -> getpcstack
    0            <- getpcstack
    0            -> kmem_log_enter
    0            <- kmem_log_enter
    0          <- kmem_cache_alloc_debug
    0        <- kmem_cache_alloc
    0      <- kmem_zalloc
    0    <- gld_mac_alloc
    0    -> kmem_zalloc
    0      -> kmem_alloc
    0        -> vmem_alloc
    0          -> highbit
    0          <- highbit
    0          -> lowbit
    0          <- lowbit
    0          -> vmem_xalloc
    0            -> highbit
    0            <- highbit
    0            -> lowbit
    0            <- lowbit
    0            -> segkmem_alloc
```

```
   0                 -> segkmem_xalloc
   0                   -> vmem_alloc
   0                     -> highbit
   0                     <- highbit
   0                     -> lowbit
   0                     <- lowbit
   0                     -> vmem_seg_alloc
   0                       -> highbit
   0                       <- highbit
   0                       -> highbit
   0                       <- highbit
   0                       -> vmem_seg_create
...
```

CHAPITRE 37

Suivi post-mortem

Ce chapitre décrit les fonctionnalités DTrace pour l'extraction et le traitement *post-mortem* des données du noyau de clients DTrace. En cas de panne système, les informations enregistrées avec DTrace peuvent fournir des indices essentiels sur le motif principal de la panne. Des données DTrace peuvent être extraites et traitées du vidage de panne système pour vous aider à comprendre les pannes système fatales. En associant ces fonctionnalités post-mortem de DTrace avec sa stratégie de mise en tampon circulaire (voir Chapitre11, "Tampons et mise en tampon"), DTrace peut servir de système d'exploitation analogue à l'enregistreur de données de vol de la *boîte noire* présent dans les avions commerciaux.

Pour extraire des données DTrace d'un vidage de panne spécifique, vous devez tout d'abord exécuter Solaris Modular Debugger, `mdb(1)`, sur le vidage de panne concerné. Le module MDB contenant les fonctions est chargé automatiquement. Pour en savoir plus sur MDB, reportez-vous au document *Solaris Modular Debugger Guide* .

Affichage de clients DTrace

Pour extraire des données DTrace d'un client DTrace, vous devez tout d'abord déterminer le client DTrace concerné en exécutant la commande MDB `::dtrace_state` :

```
> ::dtrace_state
    ADDR MINOR     PROC NAME                 FILE
ccaba400     2        - <anonymous>             -
ccab9d80     3 d1d6d7e0 intrstat         cda37078
cbfb56c0     4 d71377f0 dtrace           ceb51bd0
ccabb100     5 d713b0c0 lockstat         ceb51b60
d7ac97c0     6 d713b7e8 dtrace           ceb51ab8
```

Cette commande affiche un tableau de structures d'état DTrace. Chaque ligne du tableau contient les informations suivantes :

- l'adresse de la structure d'état ;

- le nombre inférieur associé au périphérique dtrace(7D) ;
- l'adresse de la structure de processus correspondant au client DTrace ;
- le nom du client DTrace (ou <anonymous> pour les clients anonymes) ;
- le nom de la structure de fichiers correspondant au périphérique dtrace(7D) ouvert.

Pour plus d'informations sur un client DTrace spécifique, indiquez l'adresse de sa structure de processus dans la commande ::ps :

```
> d71377f0::ps
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R 100647 100642 100647 100638      0 0x00004008 d71377f0 dtrace
```

Affichage de données de suivi

Une fois le client déterminé, vous pouvez récupérer les données correspondant à tous les tampons non consommés en indiquant l'adresse de la structure d'état dans la commande ::dtrace. L'exemple suivant illustre la sortie de la commande ::dtrace sur une activation anonyme de syscall:::entry avec l'action trace(execname) :

```
> ::dtrace_state
    ADDR MINOR     PROC NAME                 FILE
cbfb7a40     2        - <anonymous>             -

> cbfb7a40::dtrace
CPU     ID                    FUNCTION:NAME
  0    344                resolvepath:entry   init
  0     16                      close:entry   init
  0    202                      xstat:entry   init
  0    202                      xstat:entry   init
  0     14                       open:entry   init
  0    206                     fxstat:entry   init
  0    186                       mmap:entry   init
  0    186                       mmap:entry   init
  0    186                       mmap:entry   init
  0    190                     munmap:entry   init
  0    344                resolvepath:entry   init
  0    216                    memcntl:entry   init
  0     16                      close:entry   init
  0    202                      xstat:entry   init
  0     14                       open:entry   init
  0    206                     fxstat:entry   init
  0    186                       mmap:entry   init
  0    186                       mmap:entry   init
  0    186                       mmap:entry   init
  0    190                     munmap:entry   init
...
```

La commande `::dtrace` gère les erreurs comme `dtrace`(1M) : si des dépôts, des erreurs, des dépôts de spéculation ou similaires sont rencontrés pendant l'exécution du consommateur, `::dtrace` renvoie un message correspondant au message `dtrace`(1M).

L'ordre des événements affichés par `::dtrace` est toujours du plus ancien au plus récent pour une CPU donnée. Les tampons de la CPU s'affichent par ordre numérique. Si des événements de différentes CPU doivent être classés, suivez la variable `timestamp`.

Vous ne pouvez afficher que les données d'une CPU spécifique en indiquant l'option `-c` pour `::dtrace` :

```
> cbfb7a40::dtrace -c 1
CPU     ID                    FUNCTION:NAME
  1     14                       open:entry   init
  1    206                     fxstat:entry   init
  1    186                       mmap:entry   init
  1    344                resolvepath:entry   init
  1     16                      close:entry   init
  1    202                      xstat:entry   init
  1    202                      xstat:entry   init
  1     14                       open:entry   init
  1    206                     fxstat:entry   init
  1    186                       mmap:entry   init
...
```

Notez que `::dtrace` ne traite que les données DTrace *du noyau.* Les données consommées depuis le noyau et traitées (via `dtrace`(1M) ou autres) ne peuvent pas être traitées avec `::dtrace`. Pour s'assurer qu'une majorité de données soit disponible au moment d'une panne, utilisez une stratégie de mise en tampon. Pour plus d'informations sur les stratégies de tampon, reportez-vous au Chapitre11, “Tampons et mise en tampon”.

L'exemple suivant crée un tampon de très petite taille (16 K) et enregistre tous les appels système et le processus de leur création :

```
# dtrace -P syscall'{trace(curpsinfo->pr_psargs)}' -b 16k -x bufpolicy=ring
dtrace: description 'syscall:::entry' matched 214 probes
```

Supposons un vidage de panne lors de l'exécution de la commande ci-dessus ; les résultats sont similaires à l'exemple suivant :

```
> ::dtrace_state
    ADDR MINOR     PROC NAME                 FILE
cdccd400     3 d15e80a0 dtrace           ced065f0

> cdccd400::dtrace
CPU     ID                    FUNCTION:NAME
  0    139                    getmsg:return   mibiisa -r -p 25216
```

```
  0    138                  getmsg:entry   mibiisa -r -p 25216
  0    139                 getmsg:return   mibiisa -r -p 25216
  0    138                  getmsg:entry   mibiisa -r -p 25216
  0    139                 getmsg:return   mibiisa -r -p 25216
  0    138                  getmsg:entry   mibiisa -r -p 25216
  0    139                 getmsg:return   mibiisa -r -p 25216
  0    138                  getmsg:entry   mibiisa -r -p 25216
  0    139                 getmsg:return   mibiisa -r -p 25216
  0    138                  getmsg:entry   mibiisa -r -p 25216
  0     17                  close:return   mibiisa -r -p 25216
...
  0     96                   ioctl:entry   mibiisa -r -p 25216
  0     97                  ioctl:return   mibiisa -r -p 25216
  0     96                   ioctl:entry   mibiisa -r -p 25216
  0     97                  ioctl:return   mibiisa -r -p 25216
  0     96                   ioctl:entry   mibiisa -r -p 25216
  0     97                  ioctl:return   mibiisa -r -p 25216
  0     96                   ioctl:entry   mibiisa -r -p 25216
  0     97                  ioctl:return   mibiisa -r -p 25216
  0     16                   close:entry   mibiisa -r -p 25216
  0     17                  close:return   mibiisa -r -p 25216
  0    124                lwp_park:entry   mibiisa -r -p 25216
  1     68                  access:entry   mdb -kw
  1     69                 access:return   mdb -kw
  1    202                   xstat:entry   mdb -kw
  1    203                  xstat:return   mdb -kw
  1     14                    open:entry   mdb -kw
  1     15                   open:return   mdb -kw
  1    206                  fxstat:entry   mdb -kw
  1    207                 fxstat:return   mdb -kw
  1    186                    mmap:entry   mdb -kw
...
  1     13                  write:return   mdb -kw
  1     10                    read:entry   mdb -kw
  1     11                   read:return   mdb -kw
  1     12                   write:entry   mdb -kw
  1     13                  write:return   mdb -kw
  1     96                   ioctl:entry   mdb -kw
  1     97                  ioctl:return   mdb -kw
  1    364                 pread64:entry   mdb -kw
  1    365                pread64:return   mdb -kw
  1    366                pwrite64:entry   mdb -kw
  1    367               pwrite64:return   mdb -kw
  1    364                 pread64:entry   mdb -kw
  1    365                pread64:return   mdb -kw
  1     38                     brk:entry   mdb -kw
  1     39                    brk:return   mdb -kw
>
```

Notez que les enregistrements les plus récents de la CPU 1 comprennent une série d'appels système write(2) selon un processus mdb -kw. Ce résultat est probablement lié au motif de la panne système car un utilisateur peut modifier des données ou du texte du noyau en cours d'exécution avec mdb(1) lors d'une exécution avec les options -k et -w. Dans ce cas, les données DTrace fournissent au moins un élément de recherche intéressant, sinon le motif principal de la panne.

◆ ◆ ◆ CHAPITRE 38

Considérations sur les performances

DTrace entraînant une charge supplémentaire sur le système, son activation affecte toujours la performance du système. Cet effet est souvent minime, mais il peut devenir important si plusieurs sondes utilisant beaucoup de ressources sont activées. Ce chapitre décrit des techniques de réduction de l'effet de DTrace sur la performance.

Limitation des sondes activées

Des techniques d'instrumentation dynamiques permettent à DTrace de proposer une couverture de suivi inégalée du noyau et des processus utilisateur arbitraires. Cette couverture offre non seulement une nouvelle approche révolutionnaire du comportement du système, mais peut également entraîner un effet de sonde considérable. Si des centaines ou des milliers de sondes sont activées, l'effet sur le système peut devenir facilement substantiel. Vous ne devez donc activer que les sondes nécessaires à la résolution d'un problème. Vous ne devez pas, par exemple, activer toutes les sondes FBT si une activation plus précise peut répondre à votre question. Par exemple, votre question peut vous permettre de vous concentrer sur un module ou une fonction spécifique.

Soyez tout particulièrement vigilant en cas d'utilisation du fournisseur `pid.` Le fournisseur `pid` pouvant instrumentaliser chaque *instruction*, vous pourriez activer des millions de sondes dans une application, et ralentir ainsi le processus cible.

DTrace peut également être utilisé dans des situations où un grand nombre de sondes *doit* être activé pour répondre à une question. L'activation d'un grand nombre de sondes peut ralentir légèrement le système, mais n'entraînera jamais d'erreur fatale sur la machine. N'hésitez donc pas à activer de nombreuses sondes si nécessaire.

Utilisation de groupements

Tel que décrit dans le Chapitre9, "Groupements", les groupements DTrace offrent une méthode évolutive de regroupement de données. Des ensembles associatifs peuvent sembler offrir des fonctions similaires aux groupements. Cependant, en raison de leur nature globale propre aux variables universelles, ils n'offrent pas l'évolutivité linéaire des groupements. Dans la mesure du possible, privilégiez donc l'utilisation de groupements sur les ensembles associatifs. L'exemple suivant n'est pas recommandé :

```
syscall:::entry
{
    totals[execname]++;
}

syscall::rexit:entry
{
    printf("%40s %d\n", execname, totals[execname]);
    totals[execname] = 0;
}
```

L'exemple suivant est préférable :

```
syscall:::entry
{
    @totals[execname] = count();
}

END
{
    printa("%40s %@d\n", @totals);
}
```

Utilisation de prédicats pouvant être mis en cache

Les prédicats DTrace permettent de filtrer des données inutiles de l'expérimentation en suivant des données suivies uniquement si une condition spécifiée s'avère ne pas être vraie. Lors de l'activation de nombreuses sondes, vous utilisez généralement des prédicats sous une forme identifiant un ou plusieurs threads spécifiques, comme `/self->traceme/` ou `/pid == 12345/`. Bien qu'un grand nombre de ces prédicats évaluent une valeur fausse pour la plupart des threads d'une majorité de sondes, l'évaluation elle-même peut consommer beaucoup de ressources si elle est effectuée par plusieurs centaines de sondes. Pour réduire cette consommation, DTrace met en cache l'évaluation d'un prédicat s'il ne contient que des variables locales de thread (par exemple, `/self->traceme/`) ou des variables non mutables (par exemple, `/pid == 12345/`). La consommation de l'évaluation d'un prédicat mis en cache est bien inférieure à celle de l'évaluation d'un prédicat non mis en cache, tout particulièrement si le

prédicat implique des variables locales de thread, des comparaisons de chaîne ou d'autres opérations qui consomment relativement beaucoup de ressources. Alors que la mise en cache d'un prédicat est transparente pour l'utilisateur, elle implique certaines autres directives de construction de prédicats optimum, tel qu'illustré dans le tableau suivant :

Peut être mis en cache	Ne peut pas être mis en cache
`self->mumble`	`mumble[curthread]`, `mumble[pid, tid]`
`execname`	`curpsinfo->pr_fname`, `curthread->t_procp->p_user.u_comm`
`pid`	`curpsinfo->pr_pid`, `curthread->t_procp->p_pipd->pid_id`
`tid`	`curlwpsinfo->pr_lwpid`, `curthread->t_tid`
`curthread`	`curthread->`*tout membre*, `curlwpsinfo->`*tout membre*, `curpsinfo->`*tout membre*

L'exemple suivant n'est pas recommandé :

```
syscall::read:entry
{
    follow[pid, tid] = 1;
}

fbt:::
/follow[pid, tid]/
{}

syscall::read:return
/follow[pid, tid]/
{
    follow[pid, tid] = 0;
}
```

L'exemple suivant utilisant des variables locales de thread est préférable :

```
syscall::read:entry
{
    self->follow = 1;
}

fbt:::
/self->follow/
{}

syscall::read:return
/self->follow/
```

```
{
    self->follow = 0;
}
```

Pour pouvoir être mis en cache, un prédicat doit être constitué *exclusivement* d'expressions pouvant être mises en cache. Les prédicats suivants peuvent tous être mis en cache :

```
/execname == "myprogram"/
/execname == $$1/
/pid == 12345/
/pid == $1/
/self->traceme == 1/
```

Les exemples suivants, qui utilisent des variables globales, ne peuvent pas être mis en cache :

```
/execname == one_to_watch/
/traceme[execname]/
/pid == pid_i_care_about/
/self->traceme == my_global/
```

CHAPITRE 39

Stabilité

Sun permet fréquemment aux développeurs d'accéder en avance aux nouvelles technologies, ainsi qu'aux outils d'observation grâce auxquels les utilisateurs peuvent accepter bilatéralement les détails d'implémentation interne des logiciels utilisateur et de noyau. Malheureusement, les nouvelles technologies et les détails d'implémentation interne sont destinés à évoluer au fil de l'évolution et de la stabilisation des interfaces et des implémentations lorsque les logiciels sont mis à jour ou corrigés à l'aide d'un correctif. Sun documente les niveaux de stabilité des applications et des interfaces à l'aide d'un ensemble d'étiquettes décrites dans la page de manuel `attributes(5)` pour permettre de définir plus facilement les attentes des utilisateurs quant aux types de changement qui pourraient être mis en œuvre dans les diverses sortes de versions ultérieures.

Aucun attribut de sécurité ne décrit de manière appropriée l'ensemble arbitraire d'entités et de propriétés accessibles depuis un programme en D. Par conséquent, DTrace et le compilateur D intègrent des fonctions de calcul et de description dynamique des niveaux de stabilité des programmes en D que vous créez. Ce chapitre présente les fonctions de DTrace dédiées à la détermination de la stabilité du programme pour faciliter votre conception de programmes en D stables. Vous pouvez utiliser les fonctions de stabilité de DTrace pour rester informé des attributs de stabilité de vos programmes en D ou pour créer des erreurs de durée de compilation lorsque votre programme possède des dépendances d'interface indésirables.

Niveaux de stabilité

DTrace fournit deux types d'attribut de stabilité aux entités comme les variables intégrées, les fonctions et les sondes : un *niveau de stabilité* et une *classe de dépendance* architecturale. Le niveau de stabilité de DTrace vous aide à évaluer les risques lors du développement de scripts et d'outils basés sur DTrace en indiquant la probabilité selon laquelle une interface ou une entité de DTrace sera modifiée dans une version ou un correctif à venir. La classe de dépendance de DTrace vous indique si une interface est commune à toutes les plates-formes et à tous les

processeurs Solaris ou si elle est associée à une architecture particulière comme les processeurs SPARC uniquement. Les deux types d'attribut utilisés pour décrire les interfaces peuvent varier indépendamment l'un de l'autre.

Les valeurs de stabilité utilisées par DTrace apparaissent dans la liste suivante du niveau de stabilité le plus faible au plus élevé. Les interfaces les plus stables peuvent être utilisées par tous les programmes en D et les applications en couches car Sun s'efforce de garantir leur compatibilité avec les versions mineures ultérieures. Les applications qui ne dépendent que des interfaces stables continueront de fonctionner de manière fiable sur les versions mineures ultérieures et ne seront pas interrompues par des correctifs intermédiaires. Les interfaces les moins stables permettent l'expérimentation, le prototypage, le réglage et le débogage de votre système actuel. Il est nécessaire, lors de leur utilisation, de savoir qu'elles peuvent s'avérer incompatibles avec les versions mineures ultérieures, de même qu'elles peuvent être supprimées ou remplacées.

Les valeurs de stabilité de DTrace permettent de comprendre plus facilement la stabilité des entités logicielles que vous observez, en plus de la stabilité des interfaces de DTrace elles-mêmes. Par conséquent, les valeurs de stabilité de DTrace vous indiquent également la probabilité suivant laquelle les changements correspondants sont requis au niveau de vos programmes en D et des outils en couches lors de la mise à jour ou de la modification de la pile logicielle observée.

Interne
: L'interface est privée à DTrace et représente un détail d'implémentation de DTrace. Les interfaces internes peuvent changer dans les micro-versions et les versions mineures.

Privée
: L'interface est privée à Sun. Elle constitue une interface dédiée aux autres produits Sun et n'est pas publiquement documentée pour pouvoir être utilisée par les clients et les ISV. Les interfaces privées peuvent subir des modifications dans les micro-versions et les versions mineures.

Obsolète
: L'interface est prise en charge dans la version actuelle mais sa suppression, dans une prochaine version mineure, est programmée. Lorsque la prise en charge d'une interface est sur le point d'être interrompue, Sun essaye de le notifier à l'avance. Le compilateur D peut émettre des messages d'avertissement si vous tentez d'utiliser une interface obsolète.

Externe
: L'interface est contrôlée par une entité autre que Sun. Sun, selon son appréciation exclusive, peut fournir des versions de mises à jour de ces interfaces, suceptibles d'être incompatibles, en fonction de leur disponibilité au niveau de l'entité de contrôle. Sun ne fait aucune déclaration sur la compatibilité source ou binaire des interfaces externes entre deux versions différentes. Les applications basées sur ces interfaces peuvent ne pas fonctionner dans les prochaines versions, y compris les correctifs qui contiennent des interfaces externes.

Instable

L'interface est fournie pour permettre aux développeurs d'accéder en avance aux nouvelles technologies et aux technologies en pleine évolution ou à un artefact d'implémentation essentiel à l'observation ou au débogage du comportement du système pour lequel une solution plus stable sera anticipée prochainement. Sun ne fait aucune déclaration sur la compatibilité source ou binaire des interfaces instables d'une version mineure à une autre.

Evolutive

L'interface peut éventuellement devenir standard ou stable mais reste en phase de transition. Sun mettra en œuvre les efforts raisonnables pour assurer la compatibilité avec les versions précédentes au fur et à mesure de l'évolution. Lorsque des modifications à compatibilité non ascendante s'imposent, elles sont implémentées dans le cadre d'une version mineure ou majeure. Les micro-versions ne contiendront pas, dans la mesure du possible, de telles modifications. Si de telles modifications s'imposent, elles seront documentées dans les notes de version de la version concernée, et dans la mesure du possible, Sun proposera une aide à la migration pour assurer la compatibilité binaire et le développement continu des programmes en D.

Stable

L'interface est une interface stabilisée sous le contrôle de Sun. Sun essayera de ne pas intégrer de modifications à compatibilité non ascendante à ces interfaces, notamment dans les versions mineures ou les micro-versions. En cas d'interruption de la prise en charge d'une interface stable, Sun essayera de le notifier. En outre, Ie niveau de stabilité bascule sur Obsolète.

Standard

L'interface est conforme au standard de l'industrie. La documentation correspondante de l'interface décrit le standard auquel l'interface se conforme. Les standards sont généralement contrôlés par un organisme de développement de standards. Par ailleurs, l'interface peut être modifiée en fonction des changements approuvés au niveau du standard observé. Ce niveau de stabilité peut également s'appliquer aux interfaces adoptées sans qu'une convention industrielle n'impose de standard formel. La prise en charge n'est fournie que pour les versions spécifiées d'un standard ; de ce fait, la prise en charge des versions ultérieures n'est pas garantie. Si l'organisme de développement de standards approuve une modification à compatibilité non ascendante d'une interface standard que Sun choisit de prendre en charge, Sun annoncera une stratégie de compatibilité et de migration.

Classes de dépendance

Depuis que Solaris et DTrace prennent en charge un grand nombre de processeurs et de plates-formes d'exploitation, DTrace étiquette également les interfaces avec une *classe de dépendance* vous indiquant si une interface est commune à toutes les plates-formes et à tous les processeurs Solaris ou si elle est associée à une architecture système particulière. La classe de dépendance est orthogonale aux niveaux de stabilité décrits précédemment. Par exemple, une interface DTrace peut être stable tout en n'étant prise en charge que sur les microprocesseurs SPARC ou instable mais commune à tous les systèmes Solaris. Les classes de dépendance de DTrace sont décrites dans la liste suivante, de la moins commune (c'est-à-dire la plus propre à une architecture particulière) à la plus commune (c'est-à-dire, commune à toutes les architectures).

Inconnue — L'interface intègre un ensemble inconnu de dépendances architecturales. DTrace ne connaît pas nécessairement les dépendances architecturales de toutes les entités, comme les types de données définis dans l'implémentation du système d'exploitation. L'étiquette Inconnue est généralement appliquée aux interfaces dont la stabilité est très faible et pour lesquelles il est impossible de calculer des dépendances. L'interface peut ne pas être disponible lorsque vous utilisez DTrace sur *une* architecture différente de celle que vous utilisez habituellement.

CPU — L'interface est spécifique au modèle de la CPU du système actuel. Vous pouvez utiliser l'option `-v` de l'utilitaire `psrinfo(1M)` pour afficher le modèle actuel de la CPU et les noms d'implémentation. Les interfaces possédant des dépendances de modèle de CPU peuvent ne pas être disponibles sur les autres implémentations des CPU, même si ces CPU exportent la même architecture ISA. Par exemple, une interface qui dépend de la CPU sur un microprocesseur UltraSPARC-III+ peut ne pas être disponible sur un microprocesseur UltraSPARC-II, même si les deux processeurs prennent en charge le jeu d'instructions SPARC.

Plate-forme — L'interface est spécifique à la plate-forme matérielle du système actuel. Une plate-forme associe généralement un ensemble de composants système et des caractéristiques architecturales comme un ensemble de modèles de CPU pris en charge à un nom système comme `SUNW,Ultra-Enterprise-10000`. Vous pouvez afficher le nom de la plate-forme actuelle à l'aide de l'option `uname(1)` `-i`. L'interface peut ne pas être disponible sur les autres plates-formes matérielles.

Groupe — L'interface est spécifique au groupe de plates-formes matérielles du système actuel. Un groupe de plates-formes regroupe généralement un ensemble de plates-formes et les caractéristiques connexes sous un nom unique, comme `sun4u`. Vous pouvez afficher le nom du groupe de plates-formes actuel à l'aide de l'option `uname(1)` `-m`. L'interface est

	disponible sur les autres plates-formes du groupe de plates-formes mais peut ne pas l'être sur les plates-formes matérielles qui n'appartiennent pas à ce groupe.
ISA	L'interface est spécifique à l'architecture ISA prise en charge par les microprocesseurs de ce système. L'architecture ISA décrit les spécifications d'un logiciel dont l'exécution est possible sur le microprocesseur sans oublier les détails comme les instructions du langage d'assemblage et les enregistrements. Vous pouvez afficher les jeux d'instructions natives que le système prend en charge à l'aide de l'utilitaire `isainfo(1)`. Il se peut que l'interface ne soit pas prise en charge sur les systèmes qui n'exportent aucun jeu d'instructions identique. Il est possible, par exemple, qu'une interface dépendante de l'architecture ISA sur un système Solaris SPARC ne soit pas prise en charge sur un système Solaris x86.
Commune	L'interface est commune à tous les systèmes Solaris indépendamment du matériel sous-jacent. Les programmes DTrace et les applications en couches qui ne dépendent que des interfaces communes sont exécutables et déployables sur d'autres systèmes Solaris intégrant les mêmes révisions Solaris et DTrace. La plupart des interfaces DTrace étant communes, vous pouvez les utiliser où que vous utilisiez Solaris.

Attributs d'interface

DTrace décrit les interfaces au moyen d'un triplet d'attributs consistant en deux niveaux de stabilité et en une classe de dépendance. Par convention, les attributs d'interface sont écrits dans l'ordre suivant et séparés par des barres obliques :

stabilité-noms / *stabilité-noms* / *classe-dépendance*

La *stabilité des noms* d'une interface décrit le niveau de stabilité associé au nom tel qu'il apparaît dans votre programme en D ou sur la ligne de commande `dtrace(1M)`. Par exemple, la variable `execname` en langage D est un nom stable : Sun garantit que cet identificateur continuera d'être pris en charge dans vos programmes en D conformément aux règles décrites pour les interfaces stables ci-dessus.

La *stabilité des données* d'une interface se distingue de la stabilité associée au nom de l'interface. Ce niveau de stabilité décrit l'engagement de Sun à conserver les formats de données utilisés par l'interface, ainsi que la sémantique des données connexes. Par exemple, la variable `pid` en langage D est une interface stable : les ID de processus sont un concept stable dans Solaris et Sun garantit que la variable `pid` appartiendra au type `pid_t`, la sémantique étant définie sur l'ID de processus correspondant au thread ayant déclenché une sonde donnée conformément aux règles des interfaces stables.

La *classe de dépendance* d'une interface est distincte de la stabilité de son nom et de ses données et indique si l'interface est propre à la plate-forme d'exploitation ou au microprocesseur actuels.

DTrace et le compilateur D suivent les attributs de stabilité de toutes les entités d'interface de DTrace, y compris les fournisseurs, les descriptions de sonde, les variables en D, les fonctions en D, les types et les instructions de programme, comme nous allons brièvement l'étudier. Notez que les trois valeurs peuvent varier indépendamment les unes des autres. Par exemple, la variable `curthread` en langage D possède les attributs Stable/Privé/Commun : le nom de la variable est stable et il est commun à toutes les plates-formes d'exploitation Solaris mais cette variable permet d'accéder à un format de données privé qui s'avère être un artefact de l'implémentation du noyau Solaris. La plupart des variables en D possèdent les attributs Stable/Stable/Commun, comme les variables que vous définissez.

Rapports et calculs de stabilité

Le compilateur D exécute les calculs de stabilité de chaque description de sonde et instruction d'action dans vos programmes en D. Vous pouvez utiliser l'option `dtrace -v` pour afficher un rapport sur la stabilité de votre programme. L'exemple suivant utilise un programme écrit sur la ligne de commande :

```
# dtrace -v -n dtrace:::BEGIN'{exit(0);}'
dtrace: description 'dtrace:::BEGIN' matched 1 probe
Stability data for description dtrace:::BEGIN:
        Minimum probe description attributes
                Identifier Names: Evolving
                Data Semantics:   Evolving
                Dependency Class: Common
        Minimum probe statement attributes
                Identifier Names: Stable
                Data Semantics:   Stable
                Dependency Class: Common
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN
```

Vous pouvez également souhaiter combiner l'option `dtrace - v` à l'option `-e`, afin d'indiquer à dtrace de compiler votre programme en D sans l'exécuter de manière à pouvoir déterminer la stabilité du programme sans devoir activer les sondes et exécuter votre programme. Voici un autre exemple de rapport de stabilité :

```
# dtrace -ev -n dtrace:::BEGIN'{trace(curthread->t_procp);}'
Stability data for description dtrace:::BEGIN:
        Minimum probe description attributes
                Identifier Names: Evolving
                Data Semantics:   Evolving
```

```
                Dependency Class: Common
        Minimum probe statement attributes
                Identifier Names: Stable
                Data Semantics:   Private
                Dependency Class: Common
#
```

Notez que dans notre nouveau programme, nous avons référencé la variable `curthread` en langage D, dont le nom est stable mais la sémantique des données privée (ce qui signifie qu'en les consultant, vous accédez aux détails d'implémentation privés du noyau). Ce statut se reflète désormais dans le rapport de stabilité du programme. Les attributs de stabilité dans le rapport du programme sont calculés en sélectionnant la classe et le niveau de stabilité minimale en dehors de la plage de valeurs correspondantes de chaque triplet d'attributs d'interface.

Les attributs de stabilité d'une description de sonde sont calculés à l'aide des attributs de stabilité minimale de tous les champs de description de sonde *spécifiés* conformément aux attributs publiés par le fournisseur. Les attributs des fournisseurs DTrace disponibles sont présentés dans le chapitre traitant de chaque fournisseur. Les fournisseurs de DTrace exportent un triplet d'attributs de stabilité pour chacun des quatre champs de description de toutes les sondes publiées par ce fournisseur. Par conséquent, la stabilité d'un nom de fournisseur peut être plus élevée que celle des sondes individuelles exportées. Par exemple, la description de la sonde :

```
fbt:::
```

indiquant que DTrace doit suivre les E/S de toutes les fonctions du noyau, possède une stabilité plus grande que la description de la sonde :

```
fbt:foo:bar:entry
```

qui nomme une fonction `bar()` interne spécifique dans le module du noyau `foo`. Par souci de simplicité, la plupart des fournisseurs utilisent un seul ensemble d'attributs pour toutes les valeurs individuelles *nom*:*fonction*:*module* émises. Les fournisseurs spécifient également des attributs pour le tableau `args[]`, la stabilité des arguments de sonde variant d'un fournisseur à l'autre.

Si aucun champ de fournisseur n'est spécifié dans une description de sonde, cette description reçoit les attributs de stabilité Instable/Instable/Commun car elle peut aboutir à des sondes de fournisseurs qui n'existent pas encore lorsqu'utilisée sur une prochaine version de Solaris. Le cas échéant, Sun n'est pas en mesure de garantir la stabilité et le comportement à venir de son programme. Vous devez toujours spécifier explicitement le fournisseur lors de l'écriture des clauses de votre programme en D. En outre, chaque champ de description de sonde qui contient des caractères correspondant au modèle (reportez-vous au Chapitre4, “Structure de programme D”) ou des variables de macro comme `$1` (reportez-vous au Chapitre15, “Scripts”) est traité comme s'il n'était pas spécifié car il est possible de développer ces modèles de

description pour les adapter aux fournisseurs et aux sondes revus par Sun dans les versions ultérieures de DTrace et du système d'exploitation Solaris.

Les attributs de stabilité sont calculés pour la plupart des instructions en langage D en s'appuyant sur la classe et la stabilité minimale des entités figurant dans l'instruction. Par exemple, les entités en langage D ci-après possèdent les attributs suivants :

Entité	Attributs
Variable `curthread` en langage D intégrée	Stable/Privé/Commun
Variable `x` en langage D définie par l'utilisateur	Stable/Stable/Commun

Si vous écrivez l'instruction de programme en D suivante :

```
x += curthread->t_pri;
```

les attributs résultant de l'instruction sont Stable/Privé/Commun, les attributs minimaux associés aux opérandes `curthread` et `x`. La stabilité d'une expression est calculée à l'aide des attributs de stabilité minimale de chaque opérande.

Toutes les variables en D que vous définissez dans votre programme reçoivent automatiquement les attributs Stable/Stable/Commun. Par ailleurs, les attributs Stable/Stable/Commun sont affectés de manière implicite aux opérateurs en D et à la grammaire du langage D. Les attributs Privé/Privé/Inconnu sont toujours attribués aux références aux symboles du noyau utilisant l'opérateur (‘) car ils reflètent les artefacts d'implémentation. Les types que vous définissez dans le code source de vos programmes en D, et plus particulièrement celui qui est associé à l'espace de noms de types C et D, reçoivent les attributs Stable/Stable/Commun. Les types définis dans l'implémentation du système d'exploitation et fournis par d'autres espaces de noms reçoivent les attributs Privé/Privé/Inconnu. L'opérateur de stabilisation de type en langage D intègre une expression dont les attributs de stabilité correspondent aux valeurs minimales des attributs de l'expression d'entrée et des attributs du type de sortie stabilisé.

Si vous utilisez le préprocesseur C pour intégrer des fichiers d'en-tête système en C, ces types sont associés à l'espace de noms de type C et reçoivent les attributs Stable/Stable/Commun, le compilateur D n'ayant pas d'autre choix que de supposer que vous assumez la responsabilité de ces déclarations. Vous pouvez toutefois vous tromper sur la stabilité de votre programme si vous utilisez le préprocesseur C pour intégrer un fichier d'en-tête contenant des artefacts d'implémentation. Vous devez toujours consulter la documentation correspondant aux fichiers d'en-tête que vous intégrez pour déterminer les niveaux de stabilité appropriés.

Mise en œuvre de la stabilité

Lors du développement d'un script ou d'un outil en couches DTrace, vous souhaiterez peut-être identifier la source propre aux problèmes de stabilité ou vérifier que votre programme dispose de l'ensemble des attributs de stabilité souhaités. Vous pouvez utiliser l'option `dtrace -x amin=`*attributes* pour contraindre le compilateur en D à créer une erreur lorsque des calculs d'attributs engendrent un triplet d'attributs inférieur aux valeurs minimales que vous spécifiez sur la ligne de commande. L'exemple suivant présente l'utilisation de `-x amin` à l'aide d'un snippet de la source du programme en D. Notez que les attributs sont spécifiés à l'aide de trois étiquettes délimitées par `/` dans l'ordre usuel.

```
# dtrace -x amin=Evolving/Evolving/Common \
    -ev -n dtrace:::BEGIN'{trace(curthread->t_procp);}'
dtrace: invalid probe specifier dtrace:::BEGIN{trace(curthread->t_procp);}: \
    in action list: attributes for scalar curthread (Stable/Private/Common) \
    are less than predefined minimum
#
```

CHAPITRE 40

Translateurs

Vous avez appris dans le Chapitre39, “Stabilité” comment DTrace calcule et signale les attributs de stabilité du programme. Nous aimerions, dans l'idéal, créer nos programmes DTrace en n'utilisant que des interfaces stables ou évolutives. Malheureusement, lors du débogage d'un problème de bas niveau ou de l'évaluation des performances du système, il est possible que vous souhaitiez activer des sondes associées à des routines du système d'exploitation interne comme les fonctions présentes dans le noyau, plutôt que des sondes associées à des interfaces plus stables comme les appels système. Les données disponibles à l'emplacement des sondes et figurant tout au bas de la pile logicielle constituent souvent un ensemble d'artefacts d'implémentation plutôt que des structures de données plus fiables comme celles associées aux interfaces d'appel système de Solaris. Afin de vous assister dans l'écriture de programmes en D stables, DTrace propose une fonction de conversion des artefacts d'implémentation en structures de données stables accessibles à partir des instructions des programmes en D.

Déclarations du translateur

Un *translateur* est un ensemble d'instructions d'affectation en D proposées par le fournisseur d'une interface permettant de convertir une expression d'entrée en un objet de type struct. Afin de comprendre la nécessité des translateurs et de leur utilisation, examinons les routines de bibliothèque standard ANSI-C définies dans `stdio.h`. Ces routines fonctionnent sur une structure de données nommée `FILE` dont les artefacts d'implémentation sont retirés par les programmeurs en langage C. Une technique standard de création d'une abstraction de structure de données ne consiste qu'à faire suivre une déclaration de structure de données dans des fichiers d'en-tête publics tout en conservant la définition struct correspondante dans un fichier d'en-tête privé.

Si vous écrivez un programme en C et souhaitez connaître le descripteur de fichier correspondant à une struct `FILE`, vous pouvez utiliser la fonction `fileno(3C)` pour obtenir le descripteur au lieu de déréférencer directement un membre de la struct FILE. Les fichiers d'en-tête Solaris appliquent cette règle en définissant FILE comme une balise de suivi de déclaration opaque de manière à empêcher son déréférencement direct par les programmes en

C qui intègrent <stdio.h>. Dans la bibliothèque libc.so.1, vous pouvez imaginer que la fonction fileno() est implémentée dans C de la manière suivante :

```
int
fileno(FILE *fp)
{
    struct file_impl *ip = (struct file_impl *)fp;

    return (ip->fd);
}
```

Notre exemple de fonction fileno() utilise en argument un pointeur FILE et le transmet à un pointeur sur une structure libc interne correspondante, struct file_impl, puis retourne la valeur du membre fd de la structure d'implémentation. Pourquoi Solaris implémente-t-il des interfaces de cette manière ? La suppression des détails de l'implémentation actuelle de libc des programmes client permet à Sun de préserver une compatibilité binaire forte tout en continuant à évoluer et à modifier les détails d'implémentation interne de libc. Dans notre exemple, le membre fd peut changer de taille ou de position dans struct file_impl, y compris dans un correctif, alors que les codes binaires existants appelant fileno(3C) ne sont pas affectés par cette modification, car ils ne dépendent pas de ces artefacts.

Malheureusement, les logiciels d'observation comme DTrace doivent inspecter l'implémentation de l'intérieur pour fournir des résultats utiles sans se permettre le luxe d'appeler des fonctions arbitraires en C définies dans les bibliothèques de Solaris ou dans le noyau. Vous pouvez déclarer une copie de struct file_impl dans vos programmes en D pour instrumenter les routines déclarées dans stdio.h. Cependant, votre programme en D risque de reposer sur des artefacts d'implémentation privés de la bibliothèque qui ne seront peut-être plus valables dans une future micro-version ou version mineure, voire même dans un prochain correctif. Nous souhaitons, dans l'idéal, fournir une construction à utiliser dans les programmes en D, construction qui est liée à l'implémentation de la bibliothèque et mise à jour en conséquence tout en continuant de fournir une couche supplémentaire d'abstraction associée à une meilleure stabilité.

Un nouveau translateur est créé à l'aide d'une déclaration se présentant comme suit :

```
translator output-type < input-type input-identifier > {
    member-name = expression ;
    member-name = expression ;
    ...
};
```

output-type nomme une struct qui sera le type de résultat de la conversion. *input-type* spécifie le type de l'expression d'entrée et figure entre crochets angulaires < >. Il est suivi de *input-identifier* que vous pouvez utiliser en tant qu'alias pour l'expression d'entrée dans les expressions du translateur. Le corps du translateur figure entre accolades { } et se termine par un point-virgule (;). Il consiste en une liste de *member-name* et d'identificateurs correspondant

aux expressions de conversion. Chaque déclaration de membre doit nommer un membre unique de *output-type* et doit se voir affecter une expression dont le type est compatible avec celui du membre, conformément aux règles de l'opérateur d'affectation en D (=).

Par exemple, nous pouvons définir une struct d'informations stables sur les fichiers `stdio` en fonction de quelques-unes des interfaces `libc`disponibles :

```
struct file_info {
    int file_fd;   /* file descriptor from fileno(3C) */
    int file_eof;  /* eof flag from feof(3C) */
};
```

Il est ensuite possible de déclarer en langage D un translateur en D hypothétique de `FILE` vers `file_info` comme suit :

```
translator struct file_info < FILE *F > {
    file_fd = ((struct file_impl *)F)->fd;
    file_eof = ((struct file_impl *)F)->eof;
};
```

Dans notre translateur hypothétique, l'expression d'entrée est de type `FILE *` et *input-identifier* `F` lui est affecté. Il est ensuite possible d'utiliser l'identificateur `F` dans les expressions du membre du translateur comme une variable de type `FILE *` qui n'est visible que dans le corps de la déclaration du translateur. Pour déterminer la valeur du membre de sortie `file_fd`, le translateur exécute un forçage de type et un déréférencement identiques à l'implémentation hypothétique de `fileno`(3C), illustrée ci-dessus. Une conversion similaire est réalisée pour obtenir la valeur de l'indicateur EOF.

Sun fournit un ensemble de translateurs à utiliser avec les interfaces Solaris que vous pouvez invoquer à partir de vos programmes en D et promet de conserver ces translateurs conformément aux règles sur la stabilité des interfaces définies précédemment, au moment des changements dans l'implémentation de l'interface correspondante. Nous étudierons ces translateurs un peu plus loin dans ce chapitre, une fois que vous saurez invoquer des translateurs à partir du langage D. La fonction de conversion elle-même est également fournie par l'application et les développeurs de bibliothèque qui souhaitent proposer leurs propres translateurs que les programmeurs en C peuvent utiliser pour observer l'état de leurs packages logiciels.

Opérateur de conversion

L'opérateur D `xlate` permet de convertir une expression d'entrée en une expression des structures de sortie de conversion définies. L'opérateur `xlate` est utilisé dans une expression identique à l'exemple suivant :

```
xlate < output-type > ( input-expression )
```

Par exemple, pour invoquer un translateur hypothétique pour les structs FILE définies ci-dessus et accéder au membre `file_fd`, vous devriez utiliser l'expression suivante :

```
xlate <struct file_info *>(f)->file_fd;
```

`f` correspondant à une variable en D du type `FILE *`. L'expression `xlate` elle-même se voit attribuer le type défini par *output-type*. Un translateur, après avoir été défini, permet de convertir des expressions d'entrée en type de struct de sortie du translateur ou en pointeur vers cette struct.

Si vous convertissez une expression d'entrée en struct, vous pouvez soit déréférencer un membre particulier de la sortie immédiatement à l'aide de l'opérateur “ `.` ” soit affecter l'intégralité de la struct convertie à une autre variable D pour réaliser une copie des valeurs de tous les membres. S vous déréférencez un seul membre, le compilateur D ne générera que le code correspondant à l'expression de ce membre. Vous ne pouvez pas appliquer l'opérateur `&` à une struct convertie pour obtenir son adresse étant donné que l'objet de données lui-même n'existe pas tant qu'il n'est pas copié ou qu'un de ses membres n'est pas référencé.

Si vous convertissez une expression d'entrée vers un pointeur en struct, vous pouvez soit déréférencer un membre particulier de la sortie immédiatement à l'aide de l'opérateur `->`, soit déréférencer le pointeur à l'aide de l'opérateur unaire `*`, auquel cas le résultat se comporte comme si vous traduisiez l'expression en une struct. S vous déréférencez un seul membre, le compilateur D ne générera que le code correspondant à l'expression de ce membre. Vous ne pouvez pas affecter de pointeur converti en une autre variable en D étant donné que l'objet de donnée en lui-même n'existe pas tant qu'il n'a pas été copié ou que l'un de ses membres n'a pas été référencé car, de fait, son adressage est impossible.

Une déclaration du translateur peut omettre les expressions d'un ou de plusieurs membres du type de sortie. Si une expression `xlate` est utilisée pour accéder à un membre pour lequel aucune expression de conversion n'est définie, le compilateur D engendre un message d'erreur approprié et interrompt la compilation du programme. Si l'intégralité du type de sortie est copié au moyen d'une affectation de structure, tous les membres pour lesquels aucune expression de conversion n'a été définie sont remplis de zéros.

Afin de trouver un translateur correspondant à une opération `xlate`, le compilateur D examine l'ensemble des translateurs disponibles dans l'ordre suivant :

- Le compilateur commence par rechercher une conversion du type d'expression d'entrée exact vers le type de sortie exact.
- Il *résout* ensuite les types d'entrée et de sortie en suivant les alias typedef sur les noms de type sous-jacents, puis recherche une conversion du type d'entrée résolu vers le type de sortie résolu.

- Enfin, le compilateur recherche une conversion d'un type d'entrée compatible vers un type de sortie résolu. Le compilateur utilise les mêmes règles que celles qui lui permettent de déterminer la compatibilité des arguments d'appel de fonction avec les prototypes de fonction dans le but de déterminer si un type d'expression d'entrée est compatible avec un type d'entrée du translateur.

Si le compilateur D ne trouve aucun translateur conforme à ces règles, il produit un message d'erreur approprié et la compilation du programme échoue.

Translateurs du modèle de processus

Le fichier de bibliothèque de DTrace `/usr/lib/dtrace/procfs.d` fournit un ensemble de translateurs à utiliser avec vos programmes en D pour convertir à partir du noyau du système d'exploitation des structures d'implémentation des processus et des threads en structures `proc(4)` `psinfo` et `lwpsinfo` stables. Ces structures sont également utilisées dans les fichiers du système de fichiers `/proc` de Solaris, `/proc/`*pid*`/psinfo` et `/proc/`*pid*`/lwps/`*lwpid*`/lwpsinfo`, et sont définies dans le fichier d'en-tête du système `/usr/include/sys/procfs.h`. Ces structures définissent les informations stables utiles sur les processus et les threads comme l'ID de processus, l'ID LWP, les arguments initiaux et les autres données affichées par la commande `ps(1)`. Reportez-vous à `proc(4)` pour obtenir une description complète des membres et de la sémantique des structs.

TABLEAU 40–1 Translateurs `procfs.d`

Type d'entrée	Attributs du type d'entrée	Type de sortie	Attributs du type de sortie
`proc_t *`	Privé/Privé/Commun	`psinfo_t *`	Stable/Stable/Commun
`kthread_t *`	Privé/Privé/Commun	`lwpsinfo_t *`	Stable/Stable/Commun

Conversions stables

Alors qu'un translateur permet de convertir des informations en structure de données stables, il ne résout pas nécessairement tous les problèmes de stabilité que les données en cours de conversion peuvent rencontrer. Par exemple, si l'expression d'entrée d'une opération xlate référence elle-même des données instables, le programme en D qui en résulte est également instable car la stabilité du programme est toujours calculée sur la base de la stabilité minimale des instructions et des expressions accumulées du programme en D. Il est, par conséquent, parfois nécessaire de définir pour un translateur une expression d'entrée stable spécifique afin de permettre la création de programmes stables. Il est possible d'utiliser le mécanisme intégré en langage D pour faciliter ces *conversions stables*.

La bibliothèque de DTrace `procfs.d` fournit les variables `curlwpsinfo` et `curpsinfo` décrites précédemment comme des conversions stables. Par exemple, la variable `curlwpsinfo` est en fait une déclaration `inline` se présentant comme suit :

```
inline lwpsinfo_t *curlwpsinfo = xlate <lwpsinfo_t *> (curthread);
#pragma D attributes Stable/Stable/Common curlwpsinfo
```

La variable `curlwpsinfo` est définie comme une conversion intégrée de la variable `curthread` (un pointeur vers la structure de données privée du noyau) vers le type `lwpsinfo_t` stable. Le compilateur D traite ce fichier de bibliothèque et met en cache la déclaration `inline`, faisant ainsi apparaître `curlwpsinfo` comme une autre variable en D. L'instruction `#pragma` qui suit la déclaration est utilisée pour rétablir explicitement les attributs de l'identificateur `curlwpsinfo` sur Stable/Stable/Commun, masquant la référence à `curthread` dans l'expression intégrée. Grâce à cette combinaison de fonctionnalités en langage D, les programmeurs en D peuvent utiliser curthread comme source de conversion dans un mode sécurisé que Sun peut mettre à jour en fonction des changements correspondants dans l'implémentation de Solaris.

CHAPITRE 41

Versionnage

Dans le Chapitre39, “Stabilité”, vous avez découvert les fonctionnalités de DTrace permettant de déterminer les attributs de stabilité des programmes en D que vous créez. Une fois que vous avez créé un programme en D avec les attributs de stabilité appropriés, il se peut que vous souhaitiez lier ce programme à une *version* particulière de l'interface de programmation en D. La version de l'interface en langage D est une étiquette appliquée à un ensemble particulier de types, variables, fonctions, constantes et translateurs que le compilateur D vous permet d'utiliser. Si vous spécifiez une liaison à une version spécifique de l'interface de programmation en D, vérifiez que vous pourrez recompiler votre programme sur les versions ultérieures de DTrace sans rencontrer de conflits entre les identificateurs des programmes que vous définissez et les identificateurs définis dans les futures versions de l'interface de programmation en D. Vous devez lier les versions de tous les programmes en D que vous souhaitez installer en tant que scripts permanents (reportez-vous au Chapitre15, “Scripts”) ou utiliser dans les outils en couches.

Versions

Le compilateur D étiquette les ensembles de types, variables, fonctions, constantes et translateurs correspondant à une version logicielle particulière à l'aide d'une *chaîne de version*. Une chaîne de version consiste en une séquence, délimitée par un point, de décimaux entiers présentés sous la forme “*x*” (version majeure), “*x.y*” (version mineure) ou “*x.y.z*” (micro-version). Les versions sont comparées, nombre entier par nombre entier, de la gauche vers la droite. Si les nombres entiers les plus à gauche ne sont pas équivalents, la chaîne contenant le nombre entier le plus grand correspond à la version la plus récente. Si les nombres entiers les plus à gauche sont égaux, les nombres entiers suivants sont comparés, toujours de gauche à droite, pour déterminer le résultat. Lors de la comparaison des versions, tous les nombres entiers spécifiés dans une chaîne de version sont interprétés comme ayant pour valeur zéro.

Les chaînes de version de DTrace sont conformes à la nomenclature standard de Sun en matière de version d'interface, comme indiqué à la page Web `attributes(5)`. Tout changement dans

l'interface de programmation en D se traduit par une nouvelle chaîne de version. Le tableau suivant récapitule les chaînes de version utilisées par DTrace et la probable signification de la version logicielle de DTrace correspondante.

TABLEAU 41–1 Versions de DTrace

Version	Version	Signification
Majeure	*x*.0	Une version majeure se caractérise comme suit : elle comprend généralement d'importants ajouts de fonctionnalités, elle se conforme à des révisions standard différentes et parfois incompatibles et, bien que cela soit peu probable, elle peut modifier, déplacer ou remplacer des interfaces standard ou stables (reportez-vous au Chapitre39, “Stabilité”). La version initiale de l'interface de programmation en D s'appelle la version 1.0.
Mineure	*x.y*	Comparée à une version *x*.0 ou antérieure (y n'étant pas égal à zéro), une nouvelle version mineure se caractérise comme suit : ajouts probables de fonctionnalités mineurs, interfaces standard et stables compatibles, possibilité d'incompatibilité avec les interfaces évolutives et incompatibilité vraisemblable avec les interfaces instables. Ces changements peuvent intégrer de nouveaux types, variables, fonctions, constantes et translateurs en D. Par ailleurs, les interfaces précédemment dites obsolètes peuvent ne plus être prises en charge avec une version mineure (reportez-vous au Chapitre39, “Stabilité”).
Micro	*x.y.z*	Les micro-versions sont logiquement compatibles avec la version précédente (dans laquelle z est différent de zéro). Il est probable qu'elles contiennent des résolutions de bogues, une optimisation des performances et la prise en charge de matériel supplémentaire.

En règle générale, chaque nouvelle version d'une interface de programmation en D fournit un sur-ensemble des fonctionnalités proposées par la version précédente, à l'exception des interfaces obsolètes qui sont supprimées.

Options de versionnage

Tous les programmes en D que vous compilez à l'aide de `dtrace -s` ou spécifiez à l'aide des options de ligne de commande `dtrace -P`, `-m`, `-f`, `-n` ou `-i` sont liés, par défaut, à l'interface de programmation en D la plus récente que le compilateur D propose. Vous pouvez déterminer l'interface de programmation en D actuelle à l'aide de l'option `dtrace -V` :

```
$ dtrace -V
dtrace: Sun D 1.0
$
```

Si vous souhaitez établir une liaison vers une version spécifique de l'interface de programmation en D, vous pouvez configurer l'option `version` sur une chaîne de version appropriée. De même

que pour les autres options DTrace (reportez-vous au Chapitre16, “Options et paramètres réglables”), vous pouvez configurer l'option de version sur la ligne de commande à l'aide de `dtrace -x` :

```
# dtrace -x version=1.0 -n 'BEGIN{trace("hello");}'
```

ou bien utiliser la syntaxe `#pragma D option` pour configurer l'option dans votre fichier source de programme en D :

```
#pragma D option version=1.0

BEGIN
{
    trace("hello");
}
```

Si vous utilisez la syntaxe `#pragma D option` pour demander une liaison de version, vous devez l'insérer en haut du fichier de votre programme en D avant les autres déclarations et les clauses des sondes. Si l'argument de liaison de version ne constitue pas une chaîne de version ou renvoie à une version que le compilateur D ne propose pas, un message d'erreur approprié est émis et la compilation échoue. Par conséquent, vous pouvez également utiliser la fonction de liaison de version pour faire échouer un script en D sur une version *antérieure* de DTrace avec un message d'erreur évident.

Avant de compiler les déclarations et les clauses de votre programme, le compilateur D charge la configuration des types, fonctions, constantes et translateurs de la version appropriée de l'interface dans les espaces de noms du compilateur. Par conséquent, les options de liaison de version que vous spécifiez se contentent de contrôler l'ensemble des identificateurs, types et translateurs que votre programme voit, en plus des variables, types et translateurs que votre programme définit. La liaison de version empêche le compilateur D de charger des interfaces plus récentes dont la définition d'identificateurs ou de translateurs pourrait être en conflit avec les déclarations contenues dans le code source de votre programme, ce qui, par conséquent, pourrait entraîner une erreur de compilation. Reportez-vous à la section “Noms d'identifiant et mots de passe” à la page 49 pour y découvrir des conseils sur la manière de récupérer des noms d'identificateur qui ne risquent pas de créer des conflits avec les interfaces proposées par les prochaines versions de DTrace.

Versionnage des fournisseurs

Contrairement aux interfaces proposées par le compilateur D, les interfaces proposées par les fournisseurs de DTrace (c'est-à-dire les sondes et les arguments de sondes) ne sont pas affectées par ni associées à l'interface de programmation en D ou aux options de liaison de version précédemment décrites. Les interfaces de fournisseur disponibles sont établies dans le cadre du chargement de votre instrumentation compilée dans le logiciel DTrace figurant dans le noyau

de votre système d'exploitation. Elles varient, en outre, en fonction de l'architecture de votre ensemble d'instructions, de votre plate-forme d'exploitation, de votre processeur, du logiciel installé sur votre système Solaris et des privilèges de sécurité actuels. L'exécution du compilateur D et de DTrace examine les sondes décrites dans les clauses de votre programme en D et signale les messages d'erreur appropriés lorsque les sondes demandées par votre programme en D ne sont pas disponibles. Ces fonctionnalités sont orthogonales à la version de l'interface de programmation en D car les fournisseurs de DTrace n'exportent pas les interfaces pouvant créer des conflits avec les définitions contenues dans vos programmes en D. En conséquence, vous ne pouvez qu'activer les sondes en langage D, vous ne pouvez pas les définir. Enfin, les noms de sonde sont conservés dans un espace de noms distinct de celui des identificateurs de programmes en D.

Les fournisseurs de DTrace accompagnent une version particulière de Solaris et sont décrits dans la version correspondante du guide de suivi dynamique de Solaris. Le chapitre de ce guide qui présente chaque fournisseur comprend également les modifications appropriées touchant ce fournisseur, de même que les nouvelles fonctionnalités qu'il propose. Vous pouvez utiliser l'option `dtrace -l` pour parcourir l'ensemble des fournisseurs et des sondes disponibles sur votre système Solaris. Les fournisseurs étiquettent leurs interfaces à l'aide des attributs de stabilité de DTrace. Vous pouvez utiliser les fonctions de génération de rapports sur la stabilité de DTrace (reportez-vous au Chapitre39, “Stabilité”, Stability) pour déterminer si les interfaces de fournisseur utilisées par votre programme en D risquent de subir des modifications ou d'être proposées dans les prochaines versions de Solaris.

Glossaire

action Comportement mis en œuvre par le logiciel DTrace que vous pouvez exécuter au déclenchement de la sonde pour suivre les données ou modifier l'état du système externe à DTrace. Entre autres actions, vous pouvez suivre des données, arrêter des processus et capturer des suivis de pile.

activation Ensemble de sondes activées et leurs prédicats et actions connexes.

clause Déclaration d'un programme en D consistant en une liste de spécificateurs de sonde, un prédicat optionnel et une liste optionnelle d'énoncés d'action entre crochets `{ }`.

consommateur Programme utilisant DTrace pour activer l'instrumentation et lisant le flux résultant de données de suivi. La commande `dtrace` est un consommateur canonique de DTrace ; l'utilitaire `lockstat(1M)` est un autre consommateur spécialisé de DTrace.

DTrace Outil de suivi dynamique offrant des réponses concises à des questions arbitraires.

fournisseur Module du noyau implémentant un type particulier d'instrumentation au nom du logiciel DTrace. Le fournisseur exporte un espace de noms de sondes et une matrice de stabilité pour son nom et sa sémantique de données, comme illustré dans les chapitres de ce manuel.

groupement Objet qui enregistre le résultat d'une *fonction de groupement* tel que formellement défini dans le Chapitre9, “Groupements”, indexé par un tuple d'expressions que vous pouvez utiliser pour organiser les résultats.

prédicat Expression logique déterminant si un ensemble d'actions de suivi doit être exécuté au déclenchement de sondes. Chaque clause d'un programme en D peut être associée à un prédicat entre barres obliques `/ /`.

sonde Emplacement ou activité au sein du système auquel DTrace peut lier dynamiquement son instrumentation, y compris un prédicat et des actions. Chaque sonde est nommée par un tuple indiquant le fournisseur, le module, la fonction et le nom sémantique. Une sonde peut être *ancrée* sur un module ou une fonction spécifique ou *non ancrée* si elle n'est pas associée à un emplacement de programme particulier (par exemple, une horloge `profile`).

sous-routine Comportement mis en œuvre par le logiciel DTrace qui peut être exécuté au déclenchement de la sonde et qui modifie l'état interne de DTrace sans assurer le suivi des données. Similaires aux actions, les sous-routines sont demandées à l'aide de la syntaxe d'appel de fonction en D.

translateur Ensemble d'énoncés d'assignation en D qui convertissent les détails d'implémentation d'un sous-système instrumenté particulier au sein d'un objet de type `struct` formant une interface d'une stabilité supérieure à l'expression de sortie.

Index

Nombres et symboles

$ (signe dollar), 103
*curlwpsinfo, 71
*curpsinfo, 71
*curthread, 71

A

Actions
  alloca, 149
  basename, 149
  bcopy, 149
  cleanpath, 150
  copyin, 150
  copyinstr, 150
  copyinto, 151
  Destructrices, 142
    breakpoint, 145
    chill, 147
    copyout, 143
    copyoutstr, 143
    panic, 147
    raise, 143
    stop, 142
    system, 143
  dirname, 151
  Enregistrement de données, 134
  exit, 148
  jstack, 142
  msgsize, 151
  mutex_owned, 152
Actions *(Suite)*
  mutex_owner, 152
  mutex_type_adaptive, 152
  Par défaut, 133
  printa, 136
  printf, 135
  progenyof, 152
  rand, 153
  rw_iswriter, 153
  rw_write_held, 153
  Spéciales, 148
  speculation, 153
  stack, 136
    Et regroupeurs, 136
  strjoin, 153
  strlen, 154
  trace, 135
  tracemem, 135
  ustack, 138
Actions d'enregistrement de données, 134
Actions de sonde, 80
Actions destructrices, 142
  Noyau, 145
  Processus, 142
Activation anonyme, 383
Adresses mémoire, 81
Affichage de clients, 389
Affichage de données de suivi, 390
Appels système, Grands fichiers, 234
Appels système de grands fichiers, 234
arg0, 71
arg1, 71

arg2, 71
arg3, 71
arg4, 71
arg5, 71
arg6, 71
arg7, 71
arg8, 71
arg9, 71
args[], 71
Arguments de macro, 192
Attributs d'interface, 403
avg, 117

C

caller, 71
Caractère ‘, 74
Chaîne de version, 415
Chaînes, 91
  Assignation, 92
  Comparaison, 93
  Conversion, 93
  Opérateurs relationnels, 93
  Surcharge d'opérateur, 93
  Type, 91
Champs de bit, 106
Chargement de modules, 231
Ciblage d'un ID de processus, 194
Classes de dépendance, 402
Classes de dépendance d'interface, 402
  Commune, 403
  CPU, 402
  Groupe, 402
  Inconnue, 402
  ISA, 403
  Plate-forme, 402
Clause de sonde, Durée de vie et variables locales de clause, 69
Clauses de sonde, 77
Constantes de chaîne, 92
Construction binaire avec des sondes, 376
copyin(), 361
copyinstr(), 361
count, 117
cwd, 71

D

Décalages, 105
Déclaration de variable explicite
  Tableaux associatifs, 65
  Variables locales de clause, 69
  Variables scalaires, 64
Déclarations, 77
Déclarations de variables explicites, Variables locales de thread, 67
Définitions des constantes, 109
Définitions des types, 109
Descriptions de sonde, 78
  Caractères spéciaux dans, 78
  Syntaxe recommandée, 78
Directives intégrées, 111
Données de suivi
  Affichage, 390
  Extraction, 389
dtrace, 118
  Opérandes, 188
  Options, 182
DTrace
  Options, 197
dtrace
  Options
    32, 182
    64, 182
    A, 182
    a, 182
    b, 183
    C, 183
    c, 183
    D, 183
    e, 183
    F, 184
    f, 183
    G, 184
    H, 184
    I, 184
    i, 184
    L, 184

dtrace, Options *(Suite)*
l, 184
m, 184, 185
DTrace
Options
Modificatrices, 199, 355
dtrace
Options
o, 185
P, 185
p, 185
q, 185
S, 186
s, 185
U, 186
V, 186
v, 186
w, 186
X, 186
x, 186
Z, 187
Valeurs de sortie, 188

E
Élaboration d'un code binaire, 376
Ensembles
Et pointeurs, 85
Scalaires multidimensionnels, 88
Ensembles scalaires multidimensionnels, 88
Énumération, 110
Syntaxe, 110
Visibilité de UIO_READ, 111
Visibilité de UIO_WRITE, 111
Énumération de noms symboliques, 110
epid, 71
errno, 71
Espaces de noms de types, 112
Intégration, 113
execname, 71, 118
Exemples
D'utilisation de sonde pid, 352
Énumération, 111
FBT, 222

Exemples *(Suite)*
Rapports de stabilité, 404
Sonde exec, 268
Sonde sdt, 238
Spéculation, 174
Suivi anonyme, 384
Utilisation de la sonde io, 314
Utilisation des unions, 102
Variables locales de clause, 69
Variables locales de thread, 67
Extraction de données DTrace, 389

F
fasttrap Sonde, 359
Fichiers interpréteurs, 189
Fonction speculation(), 172
Fonctions non actives, 230
Fonctions non instrumentables, 230
Fournisseur lockstat, 207
Sondes, 207
Sondes d'événement de contention, 207
Sondes d'événement de maintien, 207
Fournisseur pid, 369, 371
fpuinfo, 347
Stabilité, 349

G
Groupements, 396

I
id, 71
Instructions pour compilateur, 77
Intégration de points de sonde, 375
Interférence de dtrace, 363
ipl, 71

L

Langage de programmation D
- Différences avec le langage ANSI-C, 64
- Différences par rapport à ANSI-C, 88
- Et le préprocesseur C, 80

Langage de programme en D, Déclarations de variables, 64
`lockstat`, Stabilité de, 211
`lquantize`, 117
`lwpsinfo_t`, 264

M

`max`, 117
Mémoire de processus utilisateur, 89
Mémoire virtuelle, 81
`min`, 117
Module de noyau, spécification, 75

O

`offsetof`, 105
Options, 197
- Modificatrices, 199, 355

Options modificatrices, 199

P

Paramètres réglables, 197
Performance, 395
- Prédicats pouvant être mis en cache, 396

`pid`, 71
`plockstat`, 355
Pointeurs, 81
- Déclaration, 81
- Et conversion de type, 87
- Et ensembles, 85
- Et forçages de type explicites, 87
- et `struct`, 98
- Opérations arithmétiques sur, 86
- Sur des objets DTrace, 88
- Utilisation sécurisée de, 82

Points d'arrêt, 231
Points de sonde, 375
Poses spéculatives, 179
Prédicats, 80
Prédicats pouvant être mis en cache, 396
Préprocesseur C, Et le langage de programmation D, 80
`printa`, 167
`printf`, 161
- Formats de conversion, 165
- Indicateurs de conversion, 163
- Préfixes de taille, 164
- Spécificateurs de largeur et de précision, 163
- Spécifications de conversion, 162

Privilège `dtrace_kernel`, 382
Privilège `dtrace_proc`, 380
Privilège `dtrace_user`, 381
Privilèges, 379
- DTrace, 380
- `dtrace_kernel`, 382
- `dtrace_proc`, 380
- `dtrace_user`, 381
- Superutilisateur, 382

Privilèges de superutilisateur, 382
`probefunc`, 71
`probemod`, 71
`probename`, 71
`probeprov`, 71
`psinfo_t`, 267

Q

`quantize`, 117

R

Regroupeur
- Abandons, 131
- Effacement, 129
- Normalisation, 125
- Sortie, 124
- Tronquage, 129

Regroupeurs, 117

root, 71

S

Scripts, 189
Sécurité, 379
Signe dollar ($), 103
sizeof, 105
Sonde exit, 269
Sonde fasttrap, Stabilité, 359
Sonde FBT, 221
Sonde io, 309
Sonde lwp-exit, 271
Sonde lwp-start, 271
Sonde mib, 329
  Arguments, 345
  Stabilité, 345
Sonde proc, 261
  Arguments, 263
  Stabilité, 274
Sonde sched, 275
  Stabilité, 308
Sonde sdt, 237
  Arguments, 243
  Création, 242
Sonde signal-send, 273
Sonde start, 269
Sonde syscall, 233
Sonde vminfo, 253
  Arguments, 256
  Exemple, 256
  Stabilité, 260
SondeBEGIN, 201
SondeEND, 202
SondeERROR, 203
Sondes
  BEGIN, 201
  Décalage de fonction, 353
  done, 309
  entry, 221, 353
  ERROR, 203
  Événement d'erreur, 356
  Événement de contention, 207, 355
  Événement de maintien, 207, 355

Sondes *(Suite)*
  exec, 268
  exit, 269
  fasttrap, 359
  FBT, 221
    Chargement de modules, 231
    Et optimisation des appels terminaux, 228
    Exemple d'utilisation, 222
    Fonctions non actives, 230
    Fonctions non instrumentables, 230
    Points d'arrêt, 231
    Stabilité, 231
  fpuinfo, 347
  io, 309
    Arguments, 310
    Exemple d'utilisation, 314
    Stabilité, 326
    Structure bufinfo_t, 311
    Structure devinfo_t, 312
    Structure fileinfo_t, 313
  Lecture/écriture, 210
  Limitation, 395
  Limite de fonction, 353
  lwp-exit, 271
  lwp-start, 271
  mib, 329
  mutex, 356
  pid, 351, 354
  plockstat
    Stabilité, 357
  Pour lockstat, 207
  proc, 261
  profile, 213
  return, 221, 353
  sched, 275
  sdt, 237
    Arguments, 243
    Création, 242
    Exemple d'utilisation, 238
    Stabilité, 243
  signal-send, 273
  SondeEND, 202
  start, 269, 309
  syscall(), 363

Sondes *(Suite)*
- syscall, 233
- tick, 216
- Verrou adaptatif, 208
- Verrou de rotation, 208
- Verrou de thread, 210
- Verrouillages en lecture/écriture, 357
- vminfo, 253
  - Arguments, 256
  - Exemple d'utilisation, 256
- wait-done, 309
- wait-start, 309

Sondes d'événement d'erreur, 356
Sondes d'événement de contention, 207, 355
Sondes d'événement de maintien, 207, 355
Sondes de décalage de fonction, 353
Sondes de limite du noyau, 221
Sondes de verrou adaptatif, 208
Sondes de verrou de rotation, 208
Sondes de verrou de thread, 210
Sondes de verrouillage en lecture/écriture, 210, 357
Sondes entry, 353, 354
Sondes exec, 268
Sondes FBT
- Et chargement de modules, 231
- Et points d'arrêt, 231
- Fonctions non actives, 230
- Fonctions non instrumentables, 230
- Optimisation des appels terminaux, 228

Sondes FBT Sondes, Stabilité, 231
Sondes mutex, 356
Sondes pid, 351-352
- Et limites de fonction, 353
- Exemple d'utilisation, 352

Sondes profile, 213
- Arguments, 216
- Création, 218
- Résolution de l'horloge, 217
- Stabilité, 219

Sondes return, 353
Sondes syscall
- Arguments, 235
- Interface système de grands fichiers, 234
- Stabilité, 235

Sondes tick, 216
Sous-routines, 149
- copyin(), 361
- copyinstr(), 361

Spéculation, 172
- Annulation, 174
- Création, 172
- Exemple d'utilisation, 174
- Options, 179
- Réglage, 179
- Utilisation, 172
- Validation, 173

Stabilité, 399
- Calculs, 404
- De lockstat, 211
- De sondesdtrace, 204
- fasttrap, 359
- FBT Sondes, 231
- io, 326
- mib, 345
- Mise en œuvre, 407
- Niveaux, 399
- plockstat, 357
- proc, 274
- Rapports, 404
  - Exemple d'utilisation, 404
- sched, 308
- Sonde sdt, 243
- Sondes syscall, 235
- Valeurs, 400
  - Évolutive, 401
  - Externe, 400
  - Instable, 401
  - Interne, 400
  - Obsolète, 400
  - Privée, 400
  - Stable, 401
  - Standard, 401
- vminfo, 260

Stabilité de lockstat, 211
Stabilité de sondedtrace, 204
stackdepth, 71
Stratégie de tampon, Redimensionnement, 159
Stratégie de tampon fill, 157

Stratégie de tamponfill, Et sondes END, 157
Stratégie de tamponring, 157
Struct, 95
  et pointeurs, 98
  Exemple d'utilisation, 98
Structure bufinfo_t, 311
Structure devinfo_t, 312
Structure fileinfo_t, 313
Structure kstat, Structs, 101
Suivi anonyme, 383
  Demande d'état anonyme, 384
  Exemple d'utilisation, 384
Suivi défini statiquement, *Voir* SDT
Suivi des instructions, 371
Suivi des processus utilisateur, 361
sum, 117
Surcharge d'opérateur, 93
switchStratégie de tampon, 156
Symbole de noyau
  Espace de noms, 75
  Résolution des conflits de noms, 75
Symbole du noyau, Associations de type, 75

T

Tableau uregs[], 366
Tableaux associatifs, 64
  Abandons de variables dynamiques, 66
  Affectés à zéro, 66
  Déclarations de variables explicites, 65
  Définition, 65
  Différences par rapport aux tableaux normaux, 64
  Non affectés, 66
  Touches, 64
  Tuples, 64, 65
  Types d'objet, 65
  Utilisation, 64
Tableaux scalaires, 84
Tailles des membres, 105
Tampon
  Stratégie de redimensionnement, 159
  Tailles, 159
Tampon principal
  Stratégies, 155
Tampon principal, Stratégies *(Suite)*
    fill, 157
    ring, 157
    switch, 156
Test de la limite de fonction (FBT), 369
tid, 71
timestamp, 71
trace, 169
typedef, 109

U

Unions, 101
  Exemple d'utilisation, 102
  Structure kstat, 101
uregs[], 71
ustack(), 365
Utilitaire dtrace, 181

V

Valeur de stabilité évolutive, 401
Valeur de stabilité externe, 400
Valeur de stabilité instable, 401
Valeur de stabilité interne, 400
Valeur de stabilité obsolète, 400
Valeur de stabilité privée, 400
Valeur de stabilité stable, 401
Valeur de stabilité standard, 401
Valeurs b_flags, 311
Variable de macro $target, 194
Variables de macro, 103, 191
Variables externes, 74
  Opérateurs en D, 75
  Stabilité de l'interface, 75
Variables intégrées, 71, 98
Variables locales de clause, 69
  Déclaration de variable explicite, 69
  Définition, 71
  Durée de vie d'une clause de sonde, 69
  Exemple d'utilisation, 69
  Persistance des valeurs, 71
  Utilisation, 71

Variables locales de thread, 66
Abandons de variables dynamiques, 67
Affectées à zéro, 67
Déclarations de variables explicites, 67
Exemple d'utilisation, 67
Identité de thread, 66
Non affectées, 67
Référencement, 66, 67
Types, 66
Variables scalaire, Création, 63
Variables scalaires, 63
Déclaration de variable explicite, 64
Versionnage, 415
Fournisseurs, 417
Liaison de version, 417
Options, 416
Versionnage des fournisseurs, 417
vtimestamp, 71

W

walltimestamp, 71