Guia de rastreamento dinâmico Solaris

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

Número da peça: 820–0446–10
Setembro de 2008

081121@21288

Conteúdo

Figuras

Tabelas

Exemplos

Prefácio

O DTrace é uma estrutura de rastreamento dinâmico abrangente para o sistema operacional Solaris™. O DTrace fornece uma potente infra-estrutura que permite que administradores, desenvolvedores e pessoal de serviço respondam de forma concisa a perguntas arbitrárias sobre o comportamento do sistema operacional e programas do usuário. O *Guia de rastreamento dinâmico Solaris* descreve como usar o DTrace para observar, depurar a ajustar o comportamento do sistema. Este manual inclui também uma referência completa para ferramentas de observabilidade incluídas no DTrace e a linguagem de programação D.

Observação – Esta versão do Solaris suporta os sistemas que usam as famílias de arquitetura de processadores SPARC® e x86: UltraSPARC®, SPARC64, AMD64, Pentium e Xeon EM64T. Os sistemas aceitos aparecem na *Solaris 10 Hardware Compatibility List* em http://www.sun.com/bigadmin/hcl/. Este documento cita quaisquer diferenças de implementação entre os tipos de plataformas.

Neste documento, o termo "x86" refere-se aos sistemas de 64–bits e 32–bits fabricados por meio de processadores compatíveis com as famílias de produtos AMD64 ou Intel Xeon/Pentium. Para obter os sistemas suportados, consulte a *Lista de Compatibilidade de Hardware Solaris*.

Quem deve usar este livro

Se você quer conhecer o comportamento do seu sistema, o DTrace é a ferramenta ideal. O DTrace é um abrangente recurso de rastreamento dinâmico incluído no Solaris. O recurso DTrace pode ser usado para examinar o comportamento dos programas do usuário. O recurso DTrace também pode ser usado para examinar o comportamento do sistema operacional. O DTrace pode ser usado por administradores do sistema ou desenvolvedores de aplicativo e é adequado para o uso com sistemas de produção dinâmicos. O DTrace permitirá que você explore o sistema para compreender como ele funciona, rastreie problemas de desempenho em muitas camadas do software ou localize a causa de um comportamento anormal. Como você irá perceber, o DTrace permite que você crie seus próprios programas personalizados para instrumentar dinamicamente o sistema e fornecer respostas imediatas e concisas para questões arbitrárias que você pode formular usando a linguagem de programação D do DTrace.

O DTrace permite que todos os usuários do Solaris:

- Ativem e gerenciem dinamicamente milhares de testes

- Associem dinamicamente predicados lógicos e ações com testes
- Gerenciem dinamicamente buffers de rastreamento e diretivas de buffer
- Exibam e examinem os dados do rastreamento do sistema dinâmico ou de um crash dump

O DTrace permite que os desenvolvedores e administradores do Solaris:

- Implementem scripts personalizados que usem o recurso DTrace
- Implementem ferramentas em camadas que usem o DTrace para recuperar dados de rastreamento

Este guia irá ensinar tudo que você precisa saber sobre o DTrace. Se você estiver familiarizado com uma linguagem de programação como a C ou uma linguagem de script como `awk(1)` ou `perl(1)`, aprenderá o DTrace e a linguagem de programação D mais rapidamente, mas não é necessário ser especialista nessas áreas. Se você nunca tiver escrito um programa ou script em qualquer linguagem, o "Informações relacionadas" na página 23 fornece referências a outros documentos que você possa considerar úteis.

Como este livro é organizado

O Capítulo 1, "Introdução" fornece um rápido tour de todo o recurso DTrace e apresenta aos leitores a linguagem de programação D. O Capítulo 2, "Tipos, operadores e expressões", o Capítulo 3, "Variáveis" e o Capítulo 4, "Estrutura de programa em D" discutem os fundamentos da linguagem D com mais detalhes e explicam como os programas em D são convertidos em instrumentação dinâmica. Esse grupo inicial de capítulos deve ser lido primeiro por todos os leitores.

O Capítulo 5, "Ponteiros e matrizes", o Capítulo 6, "Seqüências", Capítulo 7, "Structs e uniões" e o Capítulo 8, "Definições de tipo e de constante" discutem os recursos restantes da linguagem D, sendo que a maioria deles já serão familiares aos programadores de C, C++ e Java™. Os leitores que não estejam familiarizados com alguma dessas linguagens devem ler esses capítulos. Os programadores com mais experiência podem prosseguir diretamente para os capítulos posteriores.

O Capítulo 9, "Agregações" e o Capítulo 10, "Ações e sub-rotinas" discutem as poderosas ferramentas para a *agregação* de dados e o conjunto de ações incorporadas que podem ser usadas para criar experimentos de rastreamentos. Todos os leitores devem ler atentamente esses capítulos.

O Capítulo 11, "Buffers e armazenamento em buffer" descreve as diretivas do DTrace para armazenamento de dados em buffer e como elas podem ser configuradas. Esse capítulo deve ser lido por usuários quando estiverem familiarizados com a construção e execução de programas em D.

O Capítulo 12, "Formatação de saída" descreve as ações de formatação de saída de D assim como a diretiva padrão para formatar os dados do rastreamento. Os leitores que estejam

familiarizados com a função `printf()` de C podem ler rapidamente esse capítulo. Os leitores que nunca tenham visto a função `printf()` antes devem ler esse capítulo com cuidado.

O Capítulo 13, "Rastreio especulativo" discute o recurso DTrace para *especulativamente* confirmar dados em um buffer de rastreamento. Esse capítulo deve ser lido por usuários que precisem usar o DTrace em uma situação em que os dados devam ser rastreados antes de entender se isso é relevante para a questão atual.

O Capítulo 14, "Utilitário `dtrace`(1M)" fornece uma referência completa para o utilitário de linha de comando `dtrace`, similar à página correspondente do manual online. Os leitores podem querer consultar esse capítulo quando várias opções de linha de comando forem apresentadas em outra parte do livro. O Capítulo 15, "Script" depois discute como o utilitário `dtrace` pode ser usado para construir scripts de D executáveis e processar seus argumentos de linha de comando e o Capítulo 16, "Opções e ajustáveis" descreve as opções que podem ser ajustadas na linha de comando ou de um programa em D.

O grupo de capítulos que vai do Capítulo 17, "Provedor do `dtrace`" ao Capítulo 32, "Provedor `fasttrap`" discute os vários *provedores* de DTrace que podem ser usados para instrumentar vários aspectos do sistema Solaris. Todos os leitores devem ler brevemente esses capítulos para se familiarizar com os vários provedores e depois retornar para ler determinados capítulos em detalhes, se necessário.

O Capítulo 33, "Rastreio de processo do usuário" discute exemplos do uso de DTrace para instrumentar processos do usuário. O Capítulo 34, "Rastreio definido estaticamente em aplicativos do usuário" descreve como programadores de aplicativos podem adicionar testes e provedores de DTrace personalizados em aplicativos do usuário. Os leitores que forem administradores ou desenvolvedores de programa do usuário e quiserem usar o DTrace para investigar o comportamento do processo do usuário devem ler esses capítulos.

O Capítulo 35, "Segurança" e os capítulos restantes discutem tópicos avançados como segurança, controle de versão e atributos de estabilidade do DTrace e como realizar rastreamento de inicialização e post-mortem com o DTrace. Esses capítulos são destinados a usuários avançados do DTrace.

Informações relacionadas

Estes livros e documentos são recomendados e relacionados às tarefas que você precisa realizar com o DTrace:

- Kernighan, Brian W. e 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 e McDougall, Richard. *Solaris Internals: Core Kernel Components*. Sun Microsystems Press, 2001. ISBN 0-13-022496-0

Você pode compartilhar suas experiências e scripts do DTrace com o restante da comunidade DTrace na Web em http://www.sun.com/bigadmin/content/dtrace/ .

Documentação, suporte e treinamento

O site da Sun na Web fornece informações sobre os seguintes recursos adicionais:

- Documentação (`http://www.sun.com/documentation/`)
- Suporte (`http://www.sun.com/support/`)
- Treinamento (`http://www.sun.com/training/`)

Convenções tipográficas

A tabela a seguir descreve as convenções tipográficas usadas neste livro.

TABELA P–1 Convenções tipográficas

Fonte	Significado	Exemplo
`AaBbCc123`	Os nomes de comandos, arquivos e diretórios e saída para computador na tela	Edite o arquivo `.login`. Use `ls -a` para listar todos os arquivos. `nome_da_máquina% you have mail.`
`AaBbCc123`	O que você digitou, contrastado com a saída para computador na tela	`nome_da_máquina%` **`su`** `Password:`
aabbcc123	Espaço reservado: substituir por um nome ou valor real	O comando para remover um arquivo é `rm` *nome do arquivo*.
AaBbCc123	Títulos de livro, novos termos e termos a serem enfatizados	Leia o Capítulo 6 do *Guia do Usuário*. Um *cache* é uma cópia que é armazenada localmente. *Não* salve o arquivo. **Nota:** Alguns itens enfatizados aparecem on-line em negrito.

Prompts do shell em exemplos de comando

A tabela a seguir mostra prompts padrão do sistema UNIX® e o prompt de superusuário para o shell C, shell Bourne e shell Korn.

TABELA P–2 Prompts do shell

Shell	Prompt
Shell C	`nome_da_máquina%`
Shell C para superusuário	`nome_da_máquina#`
Shell Bourne e shell Korn	`$`
Shell Bourne e shell Korn para superusuário	`#`

◆ ◆ ◆ CAPÍTULO 1

1 Introdução

Bem-vindo ao rastreio dinâmico do Sistema operacional Solaris! Se você quer conhecer o comportamento do seu sistema, o DTrace é a ferramenta ideal. O DTrace é um recurso de rastreio dinâmico abrangente, incluído no Solaris, que pode ser usado por administradores e desenvolvedores em sistemas de produção dinâmicos para examinar o comportamento dos programas do usuário e do próprio sistema operacional. O DTrace permite que você explore o sistema para compreender como ele funciona, rastreie problemas de desempenho em muitas camadas do software ou localize a causa de um comportamento anormal. Como você irá perceber, o DTrace permite que você crie seus próprios programas personalizados para instrumentar dinamicamente o sistema e fornecer respostas imediatas e concisas para questões arbitrárias que você pode formular usando a linguagem de programação D do DTrace. A primeira seção deste capítulo fornece uma rápida introdução ao DTrace e mostra como você pode escrever seu primeiro programa em linguagem D. O resto do capítulo introduz o conjunto completo de regras para programação em D assim como as dicas e as técnicas para realizar uma análise detalhada do sistema. Você pode compartilhar suas experiências e scripts do DTrace com o restante da comunidade DTrace na Web em http://www.sun.com/bigadmin/content/dtrace/ . Todos os scripts de exemplo apresentados neste guia podem ser encontrados no sistema Solaris no diretório `/usr/demo/dtrace`.

Guia de introdução

O DTrace ajuda-o a compreender um sistema de software permitindo que modifique dinamicamente o kernel do sistema operacional e processos do usuário para registrar dados adicionais que você especificar em locais de interesse, chamados de *testes*. Um teste é um local ou atividade ao qual o DTrace pode vincular uma solicitação para realizar um conjunto de *ações*, como gravar um rastreamento de pilha, um carimbo de data/hora ou o argumento de uma função. Os testes são como sensores programáveis distribuídos em locais interessantes do sistema Solaris. Se você quiser saber o que está acontecendo, use o DTrace para programar os sensores apropriados para registrar as informações do seu interesse. Depois, quando cada teste for *disparado*, o DTrace obterá os dados resultantes e os informará para você. Se você não especificar as ações de um teste, o DTrace apenas anotará quando o teste for disparado.

Cada teste no DTrace possui dois nomes: um ID de inteiro exclusivo e um nome legível. Você irá começar a aprender o DTrace criando algumas solicitações muito simples usando o teste chamado `BEGIN`, que é acionado sempre que você inicia uma nova solicitação de rastreio. Você pode usar a opção `-n` do utilitário `dtrace(1M)` para habilitar um teste usando seu nome de seqüência de texto. Digite o seguinte comando:

```
# dtrace -n BEGIN
```

Depois de uma breve pausa, o DTrace irá informar que um teste foi ativado e você verá uma linha de resultado indicando que o teste `BEGIN` foi acionado. Quando esse resultado é exibido, o `dtrace` permanece em pausa aguardando que os outros testes sejam acionados. Como você não ativou outros testes e `BEGIN` só é acionado uma vez, pressione Control-C no shell para sair de `dtrace` e retornar ao prompt do shell:

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

O resultado informa que o teste chamado `BEGIN` foi disparado uma vez e seu nome e ID de inteiro, 1, são impressos. Observe que, por padrão, o nome de inteiro da CPU na qual esse teste foi disparado é exibido. Neste exemplo, a coluna CPU indica que o comando `dtrace` estava sendo executado na CPU 0 quando o teste foi acionado.

Você pode construir solicitações DTrace usando números arbitrários de testes e ações. Vamos criar uma solicitação simples usando dois testes adicionando o teste `END` ao comando do exemplo anterior. O teste `END` é acionado uma vez quando o rastreio é concluído. Digite o seguinte comando e depois pressione novamente Control-C no shell depois que a linha do resultado do teste `BEGIN` for exibida:

```
# 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
#
```

Como você pode ver, ao pressionar Control-C para sair de `dtrace`, você aciona o teste `END`. `dtrace` reporta o acionamento desse teste antes de sair.

Agora que você já sabe um pouco sobre nomeação e ativação de testes, está pronto para escrever a versão do DTrace do primeiro programa de todas as pessoas, "Olá, mundo". Além de construir experimentos do DTrace na linha de comando, você também pode escrevê-los em

arquivos de texto usando a linguagem de programação D. Em um editor de texto, crie um novo arquivo chamado `hello.d` e digite seu primeiro programa em D:

EXEMPLO 1–1 `hello.d`: Ola, mundo da linguagem de programação D

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

Depois de salvar o programa, você poderá executá-lo usando a opção `-s` do `dtrace`. Digite o seguinte comando:

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

Como você pode ver, o `dtrace` imprimiu o mesmo resultado de antes seguido pelo texto "`ola, mundo`". Ao contrário do exemplo anterior, você não precisou esperar e pressionar Control-C. Essas alterações foram o resultado das *ações* especificadas para o teste `BEGIN` em `hello.d`. Vamos explorar a estrutura do programa em D em mais detalhes para entender o que aconteceu.

Cada programa em D consiste em uma série de *cláusulas*, cada uma descrevendo um ou mais testes a serem ativados e um conjunto opcional de ações a serem realizadas quando o teste for acionado. As ações são listadas como uma série de instruções entre chaves `{ }` após o nome do teste. Cada instrução é encerrada com ponto-e-vírgula (`;`). A primeira instrução usa a função `trace()` para indicar que o DTrace deve registrar o argumento especificado, a seqüência "`ola, mundo `", quando o teste `BEGIN` for acionado e depois imprimi-lo. A segunda instrução usa a função `exit()` para indicar que o DTrace deve encerrar o rastreio e sair do comando `dtrace`. O DTrace fornece um conjunto de funções úteis como `trace()` e `exit()` para que você chame seus programas em D. Para chamar uma função, especifique seu nome seguido por uma lista de argumentos entre parênteses. O conjunto completo das funções D é descrito no Capítulo 10, "Ações e sub-rotinas".

Se você já estiver familiarizado com a linguagem de programação C, provavelmente já percebeu pelo nome e pelos exemplos que a linguagem de programação D do DTrace é muito similar à C. Na verdade, a D é derivada de um grande subconjunto de C combinado com um conjunto especial de funções e variáveis para ajudar a facilitar o rastreamento. Você aprenderá mais sobre esses recursos nos próximos capítulos. Se você já tiver escrito um programa em C antes, poderá transferir imediatamente grande parte do seu conhecimento para construir programas de rastreio em D. Se você nunca tiver escrito um programa em C, aprender a linguagem D, mesmo assim, é muito fácil. Você aprenderá toda a sintaxe até o final do capítulo. Mas primeiro, vamos

deixar as regras de linguagem um pouco de lado e descobrir mais sobre como o DTrace funciona e depois vamos voltar para aprender como construir programas em D mais interessantes.

Provedores e testes

Nos exemplos anteriores, você aprendeu a usar dois testes simples chamados `BEGIN` e `END`. Mas de onde vêm esses testes? Os testes do DTrace vêm de um conjunto de módulos do kernel chamados *provedores*. Cada um deles realiza um tipo particular de instrumentação para criar testes. Quando você usa o DTrace, cada provedor recebe uma oportunidade de publicar os testes que ele pode fornecer na estrutura do DTrace. Você depois pode ativar e vincular suas ações de rastreio a um dos testes que foram publicados. Para listar todos os testes disponíveis no seu sistema, digite o comando:

```
# 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 ...

#
```

Pode demorar algum tempo para que todo o resultado seja exibido. Para contar todos os testes, você pode digitar o comando:

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

Você pode observar um total diferente na sua máquina, pois o número de testes varia dependendo da sua plataforma operacional e do software que você instalou. Como você pode ver, há muitos testes disponíveis , assim, você pode vasculhar todos os cantos escuros do sistema. Na verdade, até mesmo esse resultado não é a lista completa porque, como você verá posteriormente, alguns provedores oferecem a capacidade de criar novos testes dinâmicos com base nas suas solicitações de rastreio, tornado o número real de testes do DTrace virtualmente ilimitados.

Agora, observe o resultado de **`dtrace -l`** na janela do seu terminal. Observe que cada teste possui os dois nomes mencionados anteriormente, um ID de inteiro e um nome legível. O nome legível é composto de quatro partes, mostradas como colunas separadas no resultado de `dtrace` . As quatro partes do nome de um teste são:

Provedor	O nome do provedor do DTrace que está publicando este teste. O nome do provedor geralmente corresponde ao nome do módulo do kernel do DTrace que realiza a instrumentação para ativar o teste.
Módulo	Se este teste corresponder a um local de programa específico, o nome do módulo no qual o teste está localizado. Este nome é o nome de um módulo do kernel ou o nome de uma biblioteca do usuário.
Função	Se este teste corresponder a um local de programa específico, o nome da função do programa na qual o teste está localizado.
Nome	O componente final do nome do teste é um nome que dá alguma idéia do significado semântico do teste, como `BEGIN` ou `END`.

Ao escrever o nome legível completo de um teste, escreva as quatro partes do nome separadas por dois-pontos, como por exemplo:

provedor:*módulo*:*função*:*nome*

Observe que alguns dos testes na lista não possuem um módulo e uma função, como os testes `BEGIN` e `END` usados anteriormente. Esses dois campos permanecem em branco em alguns testes porque esses testes não correspondem a um local ou a uma função de programa instrumentada específica. Em vez disso, esses testes fazem referência a um conceito mais abstrato como a idéia do final da solicitação de rastreio. Um teste que possui um módulo e uma função como parte de seu nome é conhecido como *teste ancorado* e um que não possui é conhecido como *não ancorado*.

Por convenção, se você não especificar todos os campos do nome de um teste, o DTrace corresponderá sua solicitação com *todos* os testes que possuam valores correspondentes nas partes do nome que você especificar. Em outras palavras, quando você usou o nome do teste `BEGIN` anteriormente, na verdade estava dizendo ao DTrace para corresponder qualquer teste cujo campo nome fosse `BEGIN`, independentemente do valor dos campos provedor, módulo e função. Quando isso acontece, há apenas um teste que corresponde a essa descrição, sendo assim, o resultado é o mesmo. Mas agora você sabe que o nome verdadeiro do teste `BEGIN` é `dtrace:::BEGIN`, o que indica que esse teste é fornecido pela própria estrutura do DTrace e não está ancorado a qualquer função. Assim, o programa `>hello.d` poderia ter sido escrito da maneira a seguir e produziria o mesmo resultado:

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

Agora que você entende a origem dos testes e como eles são nomeados, irá aprender um pouco mais sobre o que acontece quando você ativa testes e solicita que o DTrace faça alguma coisa, e depois retornaremos ao rápido tour da linguagem D.

Compilação e instrumentação

Quando você escreve programas tradicionais no Solaris, usa um compilador para converter o seu programa do código-fonte em código de objeto que possa ser executado. Quando você usa o comando `dtrace`, está chamando o compilador da linguagem D usada anteriormente para escrever o programa `hello.d`. Depois que o programa é compilado, ele é enviado para o kernel do sistema operacional para execução pelo DTrace. Lá, os testes que forem nomeados no seu programa são ativados e o provedor correspondente realiza toda a instrumentação necessária para ativá-los.

Toda a instrumentação no DTrace é completamente dinâmica: os testes são ativados separadamente somente quando estiverem sendo utilizados. Testes inativos não possuem código instrumentado, sendo assim, o seu sistema não sofre qualquer tipo de redução de desempenho quando você não está usando o DTrace. Quando o seu experimento estiver completo e o comando `dtrace` for encerrado, todos os testes que você usou serão automaticamente desativados e a instrumentação será removida, retornando o sistema ao seu estado original exato. Não há uma diferença efetiva entre um sistema no qual o DTrace não está ativo e um no qual o software DTrace não está instalado.

A instrumentação de cada teste é realizada dinamicamente no sistema operacional em execução dinâmico ou nos processos do usuário selecionados. O sistema não é desativado ou pausado, e o código de instrumentação é adicionado somente para os testes ativados. Como resultado, o efeito do teste do uso de DTrace é limitado a exatamente o que você pede que o DTrace faça: nenhum dado desnecessário é rastreado, nenhuma grande "opção de rastreio" é ativada no sistema e toda a instrumentação de DTrace é projetada para ser o mais eficiente possível. Esses recursos permitem que você use o DTrace em produção para solucionar problemas reais em tempo real.

A estrutura do DTrace também fornece suporte a um número arbitrário de clientes virtuais. Você pode executar simultaneamente quantos experimentos e comandos do DTrace desejar, limitado somente pela capacidade de memória do seu sistema. E todos os comandos operam independentemente usando a mesma instrumentação base. Essa mesma capacidade também permite que qualquer número de usuários distintos no sistema aproveitem o DTrace simultaneamente: desenvolvedores, administradores e pessoal de serviço podem trabalhar todos juntos ou em problemas distintos no mesmo sistema usando o DTrace sem que um interfira no trabalho do outro.

Ao contrário de programas escritos em C e C++ e similar aos programas escritos na linguagem de programação Java™, os programas em D do DTrace são compilados em uma forma intermediária segura que é usada para execução quando os testes são acionados. Essa forma intermediária é validada por segurança quando o seu programa é examinado pela primeira vez pelo software do kernel do DTrace. O ambiente de execução do DTrace também lida com os erros em tempo de execução que podem ocorrer durante a execução do programa em D, incluindo dividir por zero, apontar para memória inválida e assim por diante, e os informa para você. Como resultado, você não consegue construir um programa inseguro que possa fazer com

que o DTrace danifique inadvertidamente o kernel do Solaris ou um dos processos em execução no sistema. Esses recursos de segurança permitem que você use o DTrace em um ambiente de produção sem se preocupar com a queda do sistema ou que ele seja corrompido. Se você cometer um erro de programação, o DTrace informará o seu erro, desabilitará a instrumentação e você poderá corrigir o erro e tentar novamente. Os recursos de relatório e depuração de erros do DTrace são descritos posteriormente neste livro.

O diagrama a seguir mostra os diferentes componentes da arquitetura do DTrace, incluindo provedores, testes, o software do kernel do DTrace e o comando `dtrace`.

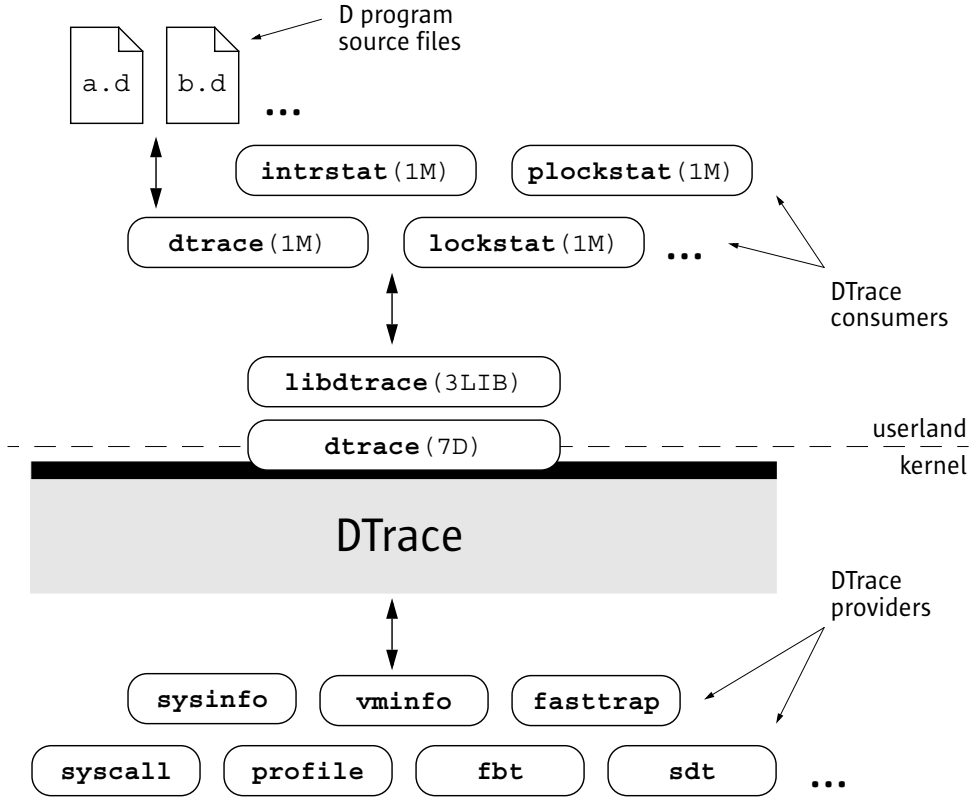

FIGURA 1–1 Visão geral da arquitetura e dos componentes do DTrace

Agora que você já sabe como o DTrace funciona, vamos retornar ao tour da linguagem de programação D e começar a escrever alguns programas mais interessantes.

Variáveis e expressões aritméticas

Nosso próximo programa de exemplo utiliza o provedor `profile` do DTrace para implementar um contador simples baseado em hora. O provedor de perfil pode criar novos testes com base nas descrições encontradas no programa em D. Se você criar um teste chamado `profile:::tick-`*n*`sec` para algum inteiro *n*, o provedor de perfil criará um teste que dispare a cada *n* segundos. Digite o seguinte código-fonte e salve-o em um arquivo chamado `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);
}
```

Quando executado, o programa conta o número de segundos decorridos até que você pressione Control-C e depois imprime o total no final:

```
# 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
#
```

As primeiras três linhas do programa são um comentário para explicar o que o programa faz. Similar às linguagens de programação C, C++ e Java, o compilador de D ignora os caracteres entre os símbolos `/*` e `*/`. Os comentários podem ser usados em qualquer lugar de um programa em D, incluindo dentro e fora das cláusulas do teste.

A cláusula do teste `BEGIN` define uma nova variável denominada `i` e atribui a ela o valor inteiro zero usando a instrução:

```
i = 0;
```

Ao contrário das linguagens de programação C, C++ e Java, as variáveis de D podem ser criadas simplesmente usando-as em uma instrução do programa; declarações explícitas de variável não são necessárias. Quando uma variável é usada pela primeira vez em um programa, o tipo da variável é definido com base no tipo da sua primeira atribuição. Cada variável possui somente um tipo durante a duração do programa, sendo assim, as referências subseqüentes devem estar de acordo com o mesmo tipo da atribuição inicial. Em `counter.d`, a primeira atribuição da variável `i` é uma constante de inteiro zero, sendo assim, seu tipo é definido como `int`. D fornece os mesmos tipos básicos de dados de inteiro que C, incluindo:

`char`	Caractere ou inteiro de byte único
`int`	Inteiro padrão
`short`	Inteiro curto
`long`	Inteiro longo
`long long`	Inteiro longo estendido

Os tamanhos destes tipos dependem do modelo de dados do kernel do sistema operacional, descrito no Capítulo 2, “Tipos, operadores e expressões”. D também fornece nomes amigáveis internos para tipos de inteiros atribuídos e não atribuídos de vários tamanhos fixos, assim como milhares de outros tipos que são definidos pelo sistema operacional.

A parte central de `counter.d` é a cláusula do teste que incrementa o contador `i`:

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

Esta cláusula nomeia o teste `profile:::tick-1sec`, que informa ao provedor `profile` para criar um novo teste que é disparado uma vez por segundo em um processador disponível. A cláusula contém duas instruções, a primeira atribuindo `i` como o valor anterior mais um e a segunda rastreando o novo valor de `i`. Todos os operadores aritméticos usuais de C estão disponíveis em D. A lista completa pode ser encontrada no Capítulo 2, “Tipos, operadores e expressões”. Além disso, como em C, o operador `++` pode ser usado como abreviação para incrementar a variável correspondente em um. A função `trace()` toma qualquer expressão de D como argumento, assim, você poderia escrever `counter.d` de forma mais concisa, como a seguir:

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

Se você desejar controlar explicitamente o tipo da variável `i`, poderá colocar o tipo desejado entre parênteses quando atribuí-lo para *converter* o inteiro zero em um tipo específico. Por exemplo, caso desejasse determinar o tamanho máximo de um `char` em D, você poderia alterar a cláusula `BEGIN` da seguinte maneira:

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

Após executar `counter.d` por um tempo, você verá o valor tracejado aumentar e depois voltar para zero. Se você não tiver paciência para esperar o valor voltar para zero, tente alterar o nome do teste `profile` para `profile:::tick-100msec` para fazer um contador que aumente a cada 100 milissegundos ou 10 vezes por segundo.

Predicados

Uma diferença importante entre D e as outras linguagens de programação como C, C++ e Java é a ausência de construções de fluxo de controle, como instruções if e loops. As cláusulas do programa em D são escritas como listas únicas de instrução em linha reta que rastreiam uma quantidade opcional, fixa de dados. D fornece a capacidade de rastrear dados condicionalmente e modificar o fluxo de controle usando expressões lógicas chamadas *predicados* que podem ser usadas para prefixar cláusulas do programa. Uma expressão de predicado é avaliada quando o teste é disparado, antes da execução de quaisquer instruções associadas à cláusula correspondente. Se o predicado for avaliado como verdadeiro, representado por um valor diferente de zero, a lista de instruções será executada. Se o predicado for falso, representado por um valor zero, nenhuma das instruções será executada e o teste não será acionado.

Digite o seguinte código-fonte para o próximo exemplo e salve-o em um arquivo chamado `countdown.d`:

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

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

```
}

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

Este programa em D implementa um temporizador de contagem regressiva de 10 segundos usando predicados. Quando executado, `countdown.d` começa uma contagem regressiva a partir de 10 e depois imprime uma mensagem e encerra:

```
# 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!
#
```

Este exemplo usa o teste `BEGIN` para inicializar um inteiro `i` como 10 para iniciar a contagem regressiva. Depois, como no exemplo anterior, o programa usa o teste `tick-1sec` para implementar um temporizador que é acionado uma vez por segundo. Observe que em `countdown.d`, a descrição do teste `tick-1sec` é usada em duas cláusulas diferentes, cada uma com um predicado e uma lista de ações diferentes. O predicado é uma expressão lógica entre barras `/ /` que aparece após o nome do teste e antes das chaves `{ }` que delimitam a lista de instruções da cláusula.

O primeiro predicado testa se `i` é maior que zero, indicando que o temporizador ainda está sendo executado:

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

O operador relacional > significa *maior que* e retorna o valor de inteiro zero para falso e um para verdadeiro. Todos os operadores relacionais de C são suportados em D. A lista completa pode

ser encontrada no Capítulo 2, “Tipos, operadores e expressões”. Se `i` ainda não for zero, o script rastreia `i` e depois o diminui em um usando o operador `--`.

O segundo predicado usa o operador == para retornar verdadeiro quando `i` for exatamente igual a zero, indicando que a contagem regressiva está concluída:

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

Similar ao primeiro exemplo, `hello.d`, `countdown.d` usa uma seqüência de caracteres entre aspas duplas, chamada de *constante de seqüências*, para imprimir uma mensagem final quando a contagem regressiva estiver concluída. A função `exit()` é então usada para encerrar `dtrace` e retornar ao prompt do shell.

Se você analisar a estrutura de `countdown.d`, verá que ao criar duas cláusulas com a mesma descrição de teste mas predicados e ações diferentes, você criou o fluxo lógico eficientemente:

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

Quando você desejar escrever programas complexos usando predicados, tente primeiro visualizar seu algoritmo desta maneira, e depois transforme cada caminho de suas construções condicionais em uma cláusula e um predicado separados.

Agora, vamos combinar predicados com um novo provedor, o `syscall` , e criar nosso primeiro programa real de rastreio em D. O provedor `syscall` permite que você ative testes na entrada ou retorno de qualquer chamada do sistema Solaris. O próximo exemplo usa o DTrace para observar cada vez que o shell realiza uma chamada do sistema de `read(2)` ou `write(2)`. Primeiro, abra duas janelas no terminal, uma para o DTrace e a outra contendo o processo do shell que você vai observar. Na segunda janela, digite o seguinte comando para obter o ID do processo deste shell:

```
# echo $$
12345
```

Agora, volte para a primeira janela do terminal e digite o seguinte programa em D e salve-o em um arquivo chamado `rw.d`. Quando você digitar o programa, substitua *12345* pelo ID do processo do shell impresso em resposta ao seu comando `echo`.

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

}
```

Observe que o corpo da cláusula do teste de `rw.d` é deixado em branco porque o programa destina-se somente a rastrear a notificação de disparos de teste e não a rastrear dados adicionais. Quando terminar de digitar no `rw.d`, use o `dtrace` para iniciar o seu experimento e depois vá para a segunda janela do shell e digite alguns comandos, pressionando a tecla de retorno após cada comando. Enquanto você digita, verá `dtrace` reportar testes acionados na primeira janela, similar ao seguinte exemplo:

```
# 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
...
```

Você agora está observando o shell realizar chamadas do sistema de `read(2)` e `write(2)` para ler um caractere da janela do terminal e retornar o resultado! Este exemplo inclui muitos dos conceitos descritos até agora e também alguns novos. Primeiro, para instrumentar `read(2)` e `write(2)` da mesma maneira, o script usa uma única cláusula de teste com várias descrições de teste separando as descrições com vírgulas, da seguinte maneira:

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

Por questões de legibilidade, a descrição de cada teste aparece em sua própria linha. Esta organização não é obrigatória, mas facilita a leitura do script. Em seguida, o script define um predicado que corresponde somente às chamadas do sistema que são executadas pelo processo do shell:

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

O predicado usa a variável predefinida do DTrace `pid`, que sempre tem o valor do ID do processo associado ao segmento que acionou o teste correspondente. O DTrace oferece muitas definições de variáveis internas para coisas úteis como o ID do processo. Veja a seguir uma lista de algumas variáveis do DTrace que você pode usar para escrever seus primeiros programas em D:

Nome da variável	Tipo de dados	Significado
`errno`	`int`	Valor do `errno` atual para chamadas do sistema
`execname`	`string`	Nome do arquivo executável do processo atual
`pid`	`pid_t`	ID do processo atual
`tid`	`id_t`	ID do segmento atual
`probeprov`	`string`	Campo do provedor da descrição do teste atual
`probemod`	`string`	Campo do módulo da descrição do teste atual
`probefunc`	`string`	Campo da função da descrição do teste atual
`probename`	`string`	Campo do nome da descrição do teste atual

Agora que você escreveu um programa de instrumentação real, tente experimentá-lo nos diferentes processos em execução no seu sistema, alterando o ID do processo e os testes de chamada do sistema que são instrumentados. Depois, você pode fazer mais uma simples alteração e transformar o `rw.d` em uma versão muito simples de uma ferramenta de rastreio de chamada do sistema como `truss(1)`. Um campo de descrição de teste vazio atua como um curinga, correspondendo a qualquer teste, sendo assim, altere o programa para o novo código-fonte a seguir para rastrear *qualquer* chamada do sistema executada pelo shell:

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

}
```

Tente digitar alguns comandos no shell como `cd`, `ls` e `date` e veja o que o programa DTrace reporta.

Formatação de saída

O rastreio de chamada do sistema é uma maneira eficiente de observar o comportamento da maioria dos processos do usuário. Se você já tiver usado o recurso `truss(1)` do Solaris antes como administrador ou desenvolvedor, provavelmente já aprendeu que ele é uma ferramenta útil de se ter por perto sempre que houver um problema. Se você nunca tiver usado `truss` antes, faça uma experiência agora digitando este comando em um dos shells:

```
$ truss date
```

Você verá um rastreio formatado de todas as chamadas do sistema executadas por `date(1)` seguido pelo resultado de uma linha no final. O exemplo a seguir apresenta uma melhoria em relação ao programa `rw.d` anterior formatando seu resultado para ficar mais parecido com

truss(1) para que você possa compreender o resultado com mais facilidade. Digite o seguinte programa e salve-o em um arquivo chamado `trussrw.d` :

EXEMPLO 1–2 `trussrw.d`: rastrear chamadas do sistema com formato de saída `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);
}
```

Neste exemplo, a constante `12345` é substituída pelo rótulo `$1` em cada predicado. Esse rótulo permite que você especifique o processo de seu interesse como um *argumento* para o script: `$1` é substituído pelo valor do primeiro argumento quando o script é compilado. Para executar `trussrw.d`, use as opções `-q` e `-s` do `dtrace`, seguidas pelo ID do processo do shell como o argumento final. A opção `-q` indica que `dtrace` deve ser silencioso e suprimir a linha do cabeçalho e as colunas CPU e ID mostradas nos exemplos anteriores. Como resultado, você verá somente a saída dos dados que rastreou explicitamente. Digite o seguinte comando (substituindo `12345` pelo ID de um processo do shell) e depois pressione a tecla de retorno algumas vezes no shell especificado:

```
# 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
#
```

Agora, vamos examinar o programa em D e a sua saída mais detalhadamente. Primeiro, uma cláusula similar ao programa anterior instrumenta cada uma das chamadas do shell para `read(2)` e `write(2)`. Mas, para este exemplo, uma nova função, `printf()`, é usada para rastrear dados e imprimi-los em um formato específico:

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

A função `printf()` combina a capacidade de rastrear dados, como se fosse pela função `trace()` usada anteriormente, com a capacidade de retornar os dados e outro texto em um formato específico descrito por você. A função `printf()` informa ao DTrace para rastrear os dados associados a cada argumento após o primeiro argumento e depois para formatar os resultados usando as regras descritas pelo primeiro argumento de `printf ()`, conhecido como *seqüência de formato*.

A seqüência de formato é uma seqüência regular que contém inúmeras conversões de formato, cada uma começando com o caractere `%`, que descreve como formatar o argumento correspondente. A primeira conversão na seqüência de formato corresponde ao segundo argumento `printf()`, a segunda conversão ao terceiro argumento, e assim por diante. Todo o texto entre conversões é impresso textualmente. O caractere que segue o caractere de conversão `%` descreve o formato a ser usado para o argumento correspondente. Veja a seguir os significados das três conversões de formato usadas em `trussrw.d`:

`%d`	Imprimir o valor correspondente como inteiro decimal
`%s`	Imprimir o valor correspondente como uma seqüência
`%x`	Imprimir o valor correspondente como um inteiro hexadecimal

A função `printf()` do DTrace funciona como a rotina de biblioteca de C `printf(3C)` ou o recurso `printf(1)` do shell. Se você nunca tiver visto a função `printf()` antes, os formatos e as opções são explicados em detalhes no Capítulo 12, “Formatação de saída”. Você deve ler esse capítulo atentamente mesmo que já esteja familiarizado com a `printf()` de outra linguagem. Em D, `printf()` é interna e algumas conversões de formato novas específicas para o DTrace estão disponíveis.

Para ajudá-lo a escrever programas corretos, o compilador de D valida cada seqüência de formato de `printf()` em comparação com a lista de argumentos. Tente alterar `probefunc` na cláusula acima para o inteiro `123`. Se você executar o programa modificado, verá uma mensagem de erro informando que a conversão de formato de seqüência `%s` não é apropriada para uso com um argumento de inteiro:

```
# 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
#
```

Para imprimir o nome da chamada do sistema read ou write e seus argumentos, use a instrução `printf()`:

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

para rastrear o nome da função atual do teste e os primeiros três argumentos de inteiro para a chamada do sistema, disponíveis nas variáveis `arg0`, `arg1` e `arg2` do DTrace. Para obter mais informações sobre os argumentos do teste, consulte o Capítulo 3, "Variáveis". O primeiro argumento para `read(2)` e `write(2)` é um descritor de arquivo, impresso em decimal. O segundo argumento é um endereço de buffer, formatado como um valor hexadecimal. O argumento final é o tamanho do buffer, formatado como um valor decimal. O especificador de formato `%4d` é usado para o terceiro argumento a fim de indicar que o valor deve ser impresso usando-se a conversão de formato `%d` com uma largura de campo mínima de 4 caracteres. Se o inteiro tiver menos que 4 caracteres, `printf()` irá inserir espaços em branco extras para alinhar o resultado.

Para imprimir o resultado da chamada do sistema e completar cada linha do resultado, use a seguinte cláusula:

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

Observe que o provedor `syscall` também publica um teste chamado `return` para cada chamada do sistema além de `entry`. A variável `arg1` do DTrace para os testes `return` de syscall tem o valor de return da chamada do sistema. O valor de return é formatado como um inteiro decimal. As seqüências de caractere que começam com barras invertidas na seqüência de formato se expandem para tabulação (`\t`) e nova linha (`\n`) respectivamente. Essas *seqüências de escape* facilitam a impressão ou o registro de caracteres difíceis de digitar. D oferece suporte ao mesmo conjunto de seqüências de escape que as linguagens de programação C, C++ e Java. A lista completa de seqüências de escape pode ser encontrada no Capítulo 2, "Tipos, operadores e expressões".

Matrizes

D permite que você defina variáveis que sejam inteiros, assim como outros tipos para representar seqüências e tipos compostos chamados *structs* e *uniões*. Se você estiver familiarizado com a programação C, ficará feliz em saber que pode usar em D todos os tipos que pode em C. Se você não for especialista em C, não se preocupe: os tipos diferentes de dados são todos descritos no Capítulo 2, "Tipos, operadores e expressões". D também oferece suporte a um tipo especial de variável chamada de *matriz de associação*. Uma matriz de associação é similar a uma matriz normal, pois ela associa um conjunto de chaves a um conjunto de valores, mas em uma matriz de associação, as chaves não são limitadas a inteiros de um intervalo fixo.

As matrizes de associação em D podem ser indexadas por uma lista de um ou mais valores de qualquer tipo. Juntos, os valores das chaves individuais formam uma *tupla* que é usada para indexar na matriz e acessar ou modificar o valor correspondente a essa chave. Cada tupla usada com uma determinada matriz de associação deve estar de acordo com a assinatura de mesmo tipo; ou seja, cada chave da tupla deve ter o mesmo tamanho e ter os mesmos tipos de chave na mesma ordem. O valor associado a cada elemento de uma determinada matriz de associação é também de um tipo fixo único para toda a matriz. Por exemplo, a seguinte instrução de D define uma nova matriz de associação `a` do tipo de valor `int` com a assinatura de tupla `[ string, int ]` e armazena o valor de inteiro 456 na matriz:

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

Quando uma matriz é definida, seus elementos podem ser acessados como qualquer outra variável de D. Por exemplo, a seguinte instrução de D modifica o elemento da matriz previamente armazenado em `a` aumentando o valor de 456 para 457:

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

Os valores dos elementos de uma matriz que você ainda não tenha atribuído são definidos como zero. Vamos usar uma matriz de associação em um programa em D. Digite o seguinte programa e salve-o em um arquivo chamado `rwtime.d`:

EXEMPLO 1–3 `rwtime.d`: chamadas `read`(2) e `write`(2) de tempo

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

syscall::read:return,
syscall::write:return
/pid == $1 && ts[probefunc] != 0/
{
```

EXEMPLO 1–3 `rwtime.d`: chamadas `read`(2) e `write`(2) de tempo *(Continuação)*

```
    printf("%d nsecs", timestamp - ts[probefunc]);
}
```

Assim como em `trussrw.d`, especifique o ID do processo do shell quando executar `rwtime.d`. Se você digitar alguns comandos do shell, verá o tempo decorrido durante cada chamada do sistema. Digite o seguinte comando e pressione a tecla de retorno algumas vezes no seu outro 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
#
```

Para rastrear o tempo decorrido para cada chamada do sistema, você deve instrumentar a entrada e o retorno de `read`(2) e `write`(2) e fazer amostragem do tempo em cada ponto. Em seguida, no retorno de uma determinada chamada do sistema, você deve calcular a diferença entre o primeiro e o segundo carimbo de data/hora. Você poderia usar variáveis separadas para cada chamada do sistema, mas assim seria complicado estender o programa a chamadas adicionais do sistema. Em vez disso, é mais fácil usar uma matriz de associação indexada pelo nome da função do teste. Esta é a primeira cláusula do teste:

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

Esta cláusula define uma matriz denominada `ts` e atribui ao membro apropriado o valor da variável `timestamp` do DTrace. Essa variável retorna o valor de um contador de nanossegundos progressivo, similar à rotina de biblioteca do Solaris `gethrtime`(3C). Quando o carimbo de data/hora da entrada é salvo, o teste de retorno correspondente realiza a amostragem de `timestamp` novamente e reporta a diferença entre a hora atual e o valor salvo:

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

O predicado no teste de retorno requer que o DTrace esteja rastreando o processo apropriado e que o teste `entry` correspondente já tenha sido acionado e atribuído um valor diferente de zero a `ts[probefunc]`. Esse truque elimina resultados inválidos quando o DTrace é iniciado pela primeira vez. Se o shell já estiver aguardando em uma chamada do sistema `read(2)` por entrada quando você executar `dtrace`, o teste `read:return` será acionado sem um `read:entry` precedente para essa primeira `read(2)` e `ts[probefunc]` terá valor zero porque ainda não foi atribuído.

Tipos e símbolos externos

A instrumentação do DTrace é executada dentro do kernel do sistema operacional Solaris, assim, além de acessar argumentos de teste e variáveis especiais do DTrace, você também pode acessar estruturas de dados do kernel, símbolos e tipos. Esses recursos permitem que usuários avançados do DTrace, administradores, pessoal de serviço e desenvolvedores de drivers examinem o comportamento de baixo nível do kernel do sistema operacional e drivers de dispositivo. A lista de leitura no início deste livro inclui livros que podem ajudá-lo a aprender mais sobre as partes internas do sistema operacional Solaris.

D usa aspa invertida (‘) como um operador de escopo especial para acessar símbolos definidos no sistema operacional e não no programa em D. Por exemplo, o kernel do Solaris contém uma declaração C de um sistema ajustável chamado `kmem_flags` para ativar os recursos de depuração do alocador de memória. Consulte o *Solaris Tunable Parameters Reference Manual* para obter informações sobre `kmem_flags`. Esse ajuste é declarado em C no código-fonte do kernel da seguinte maneira:

```
int kmem_flags;
```

Para rastrear o valor desta variável em um programa em D, você pode escrever a instrução de D:

```
trace(‘kmem_flags);
```

DTrace associa cada símbolo do kernel ao tipo usado para ele no código correspondente de C no sistema operacional, fornecendo fácil acesso baseado em fonte às estruturas de dados do sistema operacional nativo. Os nomes de símbolos do kernel são mantidos em um espaço de nome separado de identificadores de função e variáveis de D, assim, você não precisa se preocupar com o conflito entre esses nomes e o das variáveis de D.

Você agora completou um rápido tour do DTrace e aprendeu muitos dos blocos de construção básicos do DTrace necessários para a construção de programas em D maiores e mais

complexos. Os capítulos a seguir descrevem o conjunto completo de regras de D e demonstram como o DTrace pode estabelecer medidas complexas de desempenho e análise funcional do sistema facilmente. Posteriormente, você verá como usar o DTrace para conectar o comportamento do aplicativo do usuário ao comportamento do sistema, dando a você a capacidade de analisar toda a pilha do software.

Você está apenas começando!

◆ ◆ ◆ CAPÍTULO 2

Tipos, operadores e expressões

D fornece o recurso de acessar e manipular uma variedade de objetos de dados: variáveis e estruturas de dados podem ser criadas e modificadas, objetos de dados podem ser definidos no kernel do sistema operacional, processos do usuário podem ser acessados e constantes de inteiro, de ponto flutuante e de seqüência podem ser declaradas. D fornece um superconjunto dos operadores ANSI-C usados para manipular objetos e criar expressões. Este capítulo descreve o conjunto detalhado de regras de tipos, operadores e expressões.

Nomes e palavras-chave do identificador

Os nomes de identificador de D são compostos de letras maiúsculas e minúsculas, dígitos e sublinhados, onde o primeiro caractere deve ser uma letra ou sublinhado. Todos os nomes de identificador que começam com um sublinhado (_) são reservados para serem usadas pelas bibliotecas do sistema D. Você deve evitar o uso de tais nomes em seus programas em D. Por convenção, os programadores de D geralmente usam nomes com letras maiúsculas e minúsculas para variáveis e todas as letras maiúsculas para constantes.

As palavras-chave da linguagem D são identificadores especiais reservados para serem usados na própria sintaxe da linguagem de programação. Esses nomes são sempre especificados em letra minúscula e talvez não sejam usados nos nomes de variáveis de D.

TABELA 2–1 Palavras-chave de D

`auto`*	`goto`*	`sizeof`
`break`*	`if`*	`static`*
`case`*	`import`*+	`string`+
`char`	`inline`	`stringof`+
`const`	`int`	`struct`

TABELA 2–1 Palavras-chave de D *(Continuação)*

`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 reserva para uso como palavras-chave um superconjunto de palavras-chave ANSI-C. As palavras-chave reservadas para uso futuro pela linguagem D são marcadas com "*". O compilador de D produzirá um erro de sintaxe, se você tentar usar uma palavra-chave que esteja reservada para uso futuro. As palavras-chave definidas por D mas não definidas por ANSI-C são marcadas com "+". D fornece o conjunto completo de tipos e operadores encontrado em ANSI-C. A diferença principal na programação em D é a ausência de construções de fluxo de controle. Palavras-chave associadas ao fluxo de controle em ANSI-C são reservadas para uso futuro em D.

Tipos e tamanhos de dados

D fornece tipos de dados fundamentais para constantes de inteiro e de ponto flutuante. A aritmética só pode ser realizada em inteiros em programas em D. As constantes de ponto flutuante podem ser usadas para inicializar estruturas de dados, mas a aritmética de ponto flutuante não é permitida em D. D fornece modelos de dados de 32 e 64 bits para serem usados na criação de programas. O modelo de dados usado quando você executa o seu programa é o modelo de dados nativo associado ao kernel do sistema operacional ativo. Você pode determinar o modelo de dados nativo do sistema usando `isainfo -b`.

Os nomes dos tipos inteiros e seus tamanhos em cada um dos dois modelos de dados são mostrados na tabela seguinte. Os inteiros são sempre representados no formato de dois complementos na ordem de codificação de byte nativa do sistema.

TABELA 2–2 Tipos de dados inteiros de D

Nome de tipo	Tamanho 32–bit	Tamanho 64–bit
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`long`	4 bytes	8 bytes
`long long`	8 bytes	8 bytes

Tipos inteiros podem ser prefixados com o qualificador `signed` ou `unsigned` . Se não houver qualificador de sinal, o tipo é presumidamente assinado. O compilador de D também fornece os alias de tipo listados na tabela seguinte:

TABELA 2–3 Alias de tipo inteiro de D

Nome de tipo	Descrição
`int8_t`	Inteiro assinado de 1 byte
`int16_t`	Inteiro assinado de 2 bytes
`int32_t`	Inteiro assinado de 4 bytes
`int64_t`	Inteiro assinado de 8 bytes
`intptr_t`	Inteiro assinado do tamanho de um ponteiro
`uint8_t`	Inteiro não assinado de 1 byte
`uint16_t`	Inteiro não assinado de 2 bytes
`uint32_t`	Inteiro não assinado de 4 bytes
`uint64_t`	Inteiro não assinado de 8 bytes
`uintptr_t`	Inteiro não assinado do tamanho de um ponteiro

Esses alias de tipo são equivalentes a usar o nome do tipo base correspondente na tabela anterior e são definidos apropriadamente para cada modelo de dados. Por exemplo, o nome de tipo `uint8_t` é um alias do tipo `unsigned char`. Consulte o Capítulo 8, “Definições de tipo e de constante” para obter informações sobre como definir seus próprios alias de tipo a serem usados em seus programas em D.

D fornece tipos de ponto flutuante para compatibilidade com declarações e tipos ANSI-C. Operadores de ponto flutuante não são aceitos em D, mas objetos de dados de ponto flutuante podem ser rastreados e formatados por meio da função `printf()` . Os tipos de ponto flutuante listados na tabela seguinte podem ser usados:

TABELA 2–4 Tipos de dados de ponto flutuante de D

Nome de tipo	Tamanho 32–bit	Tamanho 64–bit
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`long double`	16 bytes	16 bytes

D também fornece o tipo especial `string` para representar as seqüências ASCII. As seqüências são discutidas em mais detalhes no Capítulo 6, “Seqüências”.

Constantes

Constantes de inteiro podem ser escritas em decimal (`12345`), octal (`012345`) ou hexadecimal (`0x12345`). Constantes octais (base 8) devem ser prefixadas com um zero à esquerda. Constantes hexadecimais (base 16) devem ser prefixadas com `0x` ou `0X`. Os menores tamanhos são atribuídos a constantes de inteiro entre `int`, `long` e `long long` que podem representar seu valor. Se o valor for negativo, a versão assinada do tipo será usada. Se o valor for positivo e muito grande para caber na representação de tipo assinado, a representação de tipo não assinado será usada. Você pode aplicar um dos seguintes sufixos a uma constante inteira para especificar explicitamente seu tipo D:

`u` ou `U`	Versão `unsigned` do tipo selecionado pelo compilador
`l` ou `L`	`long`
`ul` ou `UL`	`unsigned long`
`ll` ou `LL`	`long long`
`ull` ou `ULL`	`unsigned long long`

Constantes de ponto flutuante são sempre escritas em decimal e devem conter um ponto decimal (`12.345`) ou um expoente (`123e45`) ou ambos (`123.34e-5`). O tipo `double` é atribuído por padrão às constantes de ponto flutuante. Você pode aplicar um dos seguintes sufixos a uma constante inteira para especificar explicitamente seu tipo D:

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

Constantes de caractere são escritas como um único caractere ou seqüência de escape entre um par de aspas simples (`'a'`). O tipo `int` é atribuído às constantes de caractere, que são

equivalentes a uma constante inteira cujo valor é determinado pelo valor desse caractere no conjunto de caracteres ASCII. Você pode consultar `ascii(5)` para obter uma lista de caracteres e seus valores. Você também pode usar qualquer uma das seqüências de escape especiais mostradas na tabela seguinte em suas constantes de caractere. D aceita as mesmas seqüências de escape encontradas em ANSI-C.

TABELA 2–5 Seqüências de escape de caractere de D

`\a`	alerta	`\\`	retroreferência
`\b`	backspace	`\?`	ponto de interrogação
`\f`	alimentação de formulário	`\'`	aspas simples
`\n`	nova linha	`\"`	aspas duplas
`\r`	retorno de carro	`\0`*oo*	valor octal 0*oo*
`\t`	guia horizontal	`\x`*hh*	valor hexadecimal 0x*hh*
`\v`	guia vertical	`\0`	caractere nulo

Você pode incluir mais de um especificador de caractere entre aspas simples para criar inteiros cujos bytes individuais são inicializados de acordo com os especificadores de caractere correspondentes. Os bytes são lidos da esquerda para a direita da constante de caractere e atribuídos ao inteiro resultante na ordem correspondente ao endian-ness nativo do seu sistema operacional. Até oito especificadores de caractere podem ser incluídos em uma única constante de caractere.

As constantes de seqüências de qualquer tamanho podem ser compostas quando colocadas entre um par de aspas duplas ("`olá`"). Uma constante de seqüência talvez não contenha um caractere de nova linha literal. Para criar seqüências que contêm novas linhas, use a seqüência de escape `\n` em vez de uma nova linha literal. As constantes de seqüência podem conter qualquer de uma das seqüências de escape de caractere especial mostradas acima para constantes de caractere. Semelhante ao ANSI-C, as seqüências são representadas como matrizes de caracteres terminadas por um caractere nulo (`\0`), que é implicitamente adicionado a cada constante de seqüência que você declarar. O tipo D especial `string` é atribuído às constantes de seqüência. O compilador de D fornece um conjunto de recursos especiais para comparar e rastrear as matrizes de caracteres que são declaradas como seqüências, conforme descrito no Capítulo 6, “Seqüências”.

Operadores aritméticos

D fornece os operadores aritméticos binários mostrados na tabela seguinte a serem usados em seus programas. Todos esses operadores possuem o mesmo significado para inteiros, como acontece em ANSI-C.

TABELA 2–6 Operadores aritméticos de D

`+`	adição de inteiro
`-`	subtração de inteiro
`*`	multiplicação de inteiro
`/`	divisão de inteiro
`%`	módulo de inteiro

A aritmética em D só pode ser realizada em operandos de inteiro, ou em ponteiros, como discutido no Capítulo 5, "Ponteiros e matrizes". A aritmética pode ser realizada em operandos de ponto flutuante em programas em D. O ambiente de execução do DTrace não precisa realizar nenhuma ação de estouro positivo ou de estouro negativo de inteiro. Você mesmo deve verificar essas condições em situações em que o estouro positivo e negativo podem ocorrer.

O ambiente de execução do DTrace não verifica e reporta automaticamente os erros de divisão por zero resultantes do uso incorreto dos operadores `/` e `%`. Se um programa em D executar uma operação de divisão inválida, o DTrace desativará automaticamente a instrumentação afetada e reportará o erro. Os erros detectados por DTrace não têm efeito em outros usuários de DTrace ou no kernel do sistema operacional, sendo assim, você não precisa se preocupar em causar nenhum dano se o seu programa inadvertidamente contiver um desses erros.

Além desses operadores binários, os operadores `+` e `-` também podem ser usados como operadores unários; esses operadores têm maior precedência do que quaisquer operadores aritméticos binários. A ordem de precedência e as propriedades de associação de todos os operadores de D estão disponíveis no Tabela 2–11. Você pode controlar a precedência agrupando expressões entre parênteses `( )`.

Operadores relacionais

D fornece os operadores relacionais binários, mostrados na tabela seguinte, a serem usados em seus programas. Todos esses operadores possuem o mesmo significado que em ANSI-C.

TABELA 2–7 Operadores relacionais de D

`<`	o operando esquerdo é menor que o operando direito
`<=`	o operando esquerdo é menor ou igual ao operando direito
`>`	o operando esquerdo é maior que o operador direito
`>=`	o operando esquerdo é maior ou igual ao operando direito
`==`	o operando esquerdo é igual ao operando direito
`!=`	o operando esquerdo não é igual ao operando direito

Os operadores relacionais são os mais usados para escrever predicados de D. Cada operador é avaliado com um valor de tipo `int` que é igual a um, se a condição for verdadeira, ou zero se for falsa.

Operadores relacionais podem ser aplicados a pares de inteiros, ponteiros ou seqüências. Se os ponteiros forem comparados, o resultado será equivalente a uma comparação de inteiros dos dois ponteiros interpretados como inteiros não assinados. Se as seqüências forem comparadas, o resultado será determinado como se estivesse sendo realizada uma `strcmp(3C)` nos dois operandos. Eis alguns exemplos de comparações de seqüências de D e seus resultados:

`"café" < "expresso"`	... retorna 1 (verdadeiro)
`"café" == "café"`	... retorna 1 (verdadeiro)
`"café" >= "mocha"`	... retorna 0 (falso)

Operadores relacionais também podem ser usados para comparar um objeto de dados associado a um tipo de enumeração com qualquer uma das marcas de enumerador definidas pela enumeração. Enumerações são um recurso para criar constantes de inteiro nomeadas e são descritas em mais detalhes no Capítulo 8, “Definições de tipo e de constante”.

Operadores lógicos

D fornece os seguintes operadores lógicos binários a serem usados em seus programas. Os dois primeiros operadores são equivalentes aos operadores ANSI-C correspondentes.

TABELA 2–8 Operadores lógicos de D

`&&`	*AND* lógico: verdadeiro se ambos os operandos forem verdadeiros
`||`	*OR* lógico: verdadeiro se um ou ambos os operandos forem verdadeiros

TABELA 2–8 Operadores lógicos de D *(Continuação)*

^^	*XOR* lógico: verdadeiro se exatamente um operando for verdadeiro

Os operadores lógicos são os mais usados para escrever predicados de D. O operador lógico AND realiza avaliação de circuito curto: Se o operando esquerdo for falso, a expressão direita não será avaliada. O operador lógico OR realiza avaliação de circuito curto: Se o operando esquerdo for verdadeiro, a expressão direita não será avaliada. O operador XOR lógico não realiza circuito curto: ambos os operandos da expressão são sempre avaliados.

Além dos operadores lógicos binários, o operador unário `!` pode ser usado para realizar uma negação lógica de um único operando: converte um operando zero em um, e um operando diferente de zero em zero. Por convenção, os programadores de D usam `!` quando trabalham com inteiros que representam valores booleanos, e `== 0` quando trabalham com inteiros não booleanos, embora ambas as expressões tenham significado equivalente.

Os operadores lógicos podem ser aplicados a operandos de tipos de inteiro ou de ponteiro. Os operadores lógicos interpretam operandos de ponteiro como valores inteiros não assinados. Como em todos os operadores lógicos e relacionais em D, os operandos são verdadeiros caso possuam um valor inteiro diferente de zero e falsos caso possuam um valor inteiro igual a zero .

Operadores bit a bit

D fornece os seguintes operadores binários para manipular bits individuais dentro de operandos inteiros. Todos esses operadores possuem o mesmo significado em ANSI-C.

TABELA 2–9 Operadores bit a bit de D

&	AND bit a bit
|	OR bit a bit
^	XOR bit a bit
<<	move o operando esquerdo para a esquerda pelo número de bits especificado pelo operando direito
>>	move o operando esquerdo para a direita pelo número de bits especificado pelo operando direito

O operador binário `&` é usado para limpar os bits de um operando inteiro. O operador binário `|` é usado para definir os bits em um operando inteiro. O operador binário `^` retorna um em cada posição de bit onde exatamente um dos bits de operando correspondente está definido.

Os operadores de deslocamento são usados para mover os bits para a esquerda ou para a direita em um determinado operando inteiro. O deslocamento para a esquerda preenche com zeros as posições de bit vazias no lado direito do resultado. O deslocamento para a direita usando um

operando inteiro não assinado preenche com zeros as posições de bit vazias no lado esquerdo do resultado. O deslocamento para a direita usando um operando inteiro assinado preenche o lado esquerdo com o valor do bit de sinal, também conhecido como uma operação de *deslocamento aritmético*.

Mover um valor inteiro por um número negativo de bits ou por um número de bits maior que o número de bits no próprio operando esquerdo produz um resultado indefinido. O compilador de D produzirá uma mensagem de erro, caso detecte essa condição quando você compilar seu programa em D.

Além dos operadores lógicos binários, o operador unário ~ pode ser usado para realizar uma negação bit a bit de um único operando: ele converte cada bit zero no operando em um bit um, e cada bit um no operando em um bit zero.

Operadores de atribuição

D fornece os seguintes operadores de atribuição binários para modificar variáveis de D. Você só pode modificar variáveis e matrizes de D. Objetos de dados e constantes do kernel não podem ser modificados por meio de operadores de atribuição de D. O operador de atribuição possui o mesmo significado que em ANSI-C.

TABELA 2–10 Operadores de atribuição de D

`=`	define o operando esquerdo igual ao valor da expressão direita
`+=`	incrementa o operando esquerdo pelo valor da expressão direita
`-=`	decrementa o operando esquerdo pelo valor da expressão direita
`*=`	multiplica o operando esquerdo pelo valor da expressão direita
`/=`	divide o operando esquerdo pelo valor da expressão direita
`%=`	modula o operando esquerdo pelo valor da expressão direita
`|=`	OR bit a bit do operando esquerdo com o valor da expressão direita
`&=`	AND bit a bit do operando esquerdo com o valor da expressão direita
`^=`	XOR bit a bit do operando esquerdo com o valor da expressão direita
`<<=`	move o operando esquerdo para a esquerda pelo número de bits especificado pelo valor da expressão direita
`>>=`	move o operando esquerdo para a direita pelo número de bits especificado pelo valor da expressão direita

Além do operador de distribuição =, os outros operadores de atribuição são fornecidos como abreviação para o uso do operador = com um dos outros operadores descritos anteriormente.

Por exemplo, a expressão `x = x + 1` é equivalente à expressão `x += 1`, exceto que a expressão `x` é avaliada uma vez. Estes operadores de atribuição obedecem às mesmas regras para tipos de operando que os formatos binários descritos anteriormente.

O resultado de qualquer operador de atribuição é uma expressão igual ao novo valor da expressão esquerda. Você pode usar os operadores de atribuição ou qualquer um dos operadores descritos até agora para formar expressões de complexidade arbitrária. Você pode usar parênteses `( )` para agrupar termos em expressões complexas.

Operadores de incremento e de decremento

D fornece os operadores unários especiais `++` e `--` para incrementar e decrementar ponteiros e inteiros. Esses operadores possuem o mesmo significado que em ANSI-C. Esses operadores só podem ser aplicados a variáveis, e podem ser aplicados antes ou depois do nome da variável. Se o operador aparecer antes do nome da variável, a variável será modificada primeiro e a expressão resultante será igual ao novo valor da variável. Por exemplo, as duas expressões seguintes produzem resultados idênticos:

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

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

Se o operador aparecer depois do nome da variável, a variável será modificada depois que seu valor atual for retornado para ser usado na expressão. Por exemplo, as duas expressões seguintes produzem resultados idênticos:

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

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

Você pode usar os operadores de incremento e de decremento para criar novas variáveis sem declará-las. Se uma declaração de variável for omitida e o operador de incremento ou de decremento for aplicado a uma variável, a variável será declarada implicitamente como tipo `int64_t`.

Os operadores de incremento e de decremento podem ser aplicados às variáveis de inteiro ou ponteiro. Quando aplicados a variáveis de inteiro, os operadores incrementam ou decrementam o valor correspondente por um. Quando aplicados às variáveis de ponteiro, os operadores incrementam ou decrementam o endereço de ponteiro pelo tamanho do tipo de dados referenciado pelo ponteiro. Os ponteiros e a aritmética de ponteiro em D são discutidos no Capítulo 5, “Ponteiros e matrizes”.

Expressões condicionais

Embora D não dê suporte a construções if-then-else, ela não oferece suporte a expressões condicionais simples que usam os operadores `?` e `:` . Esses operadores permitem que três expressões sejam associadas, onde a primeira expressão é usada para avaliar condicionalmente uma das outras duas. Por exemplo, a declaração de D seguinte poderia ser usada para definir uma variável `x` como uma de duas seqüências, dependendo do valor de `i`:

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

Neste exemplo, a expressão `i == 0` é avaliada primeiro para determinar se é verdadeira ou falsa. Se a primeira expressão for verdadeira, a segunda expressão será avaliada e a expressão `?:` retornará seu valor. Se a primeira expressão for falsa, a terceira expressão será avaliada e a expressão `?:` retornará seu valor.

Como em qualquer operador de D, você pode usar vários operadores `?:` em uma única expressão para criar mais expressões complexas. Por exemplo, a expressão seguinte tomaria uma variável `char c` contendo um dos caracteres 0-9, a-z ou A-Z, e retornaria o valor desse caractere quando interpretado como um dígito em um inteiro (base 16) hexadecimal:

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

A primeira expressão usada com `?:` deve ser um ponteiro ou um inteiro para que seja avaliada em seu valor verdadeiro. A segunda e a terceira expressões podem ser de quaisquer tipos compatíveis. Você não pode construir uma expressão condicional onde, por exemplo, um caminho retorne uma seqüência e outro caminho retorne um inteiro. A segunda e a terceira expressões também não podem chamar uma função de rastreamento como `trace()` ou `printf()`. Se você quiser rastrear dados condicionalmente, use um predicado, como discutido no Capítulo 1, “Introdução”.

Tipos de conversões

Quando as expressões são construídas por meio de operandos de tipos diferentes mas compatíveis, as conversões de tipo são realizadas para determinar o tipo da expressão resultante. As regras de D para conversões de tipo são as mesmas regras de conversão aritmética em ANSI-C. Essas regras às vezes são chamadas de *conversões aritméticas usuais*.

Uma forma simples de descrever as regras de conversão é a seguinte: cada tipo de inteiro é classificado na ordem `char, short, int, long, long long`, sendo que os tipos não assinados correspondentes recebem uma classificação acima de seu equivalente assinado, mas abaixo do próximo tipo inteiro. Quando você constrói uma expressão usando dois operandos inteiros, tais como `x + y` e os operandos são tipos de inteiros diferentes, o tipo de operando com a classificação mais alta é usado como o tipo de resultado.

Se for necessária uma conversão, o operando de classificação mais baixa é primeiro *promovido* para o tipo de classificação mais alto. A promoção não altera realmente o valor do operando: ela simplesmente estende o valor para um recipiente maior, de acordo com o seu sinal. Se um operando não assinado for promovido, os bits de ordem alta não utilizados do inteiro resultante serão preenchidos com zeros. Se um operando assinado for promovido, os bits de ordem alta não utilizados serão preenchidos por meio de extensão de sinal. Se um tipo assinado for convertido em um tipo não assinado, o tipo assinado primeiro tem o sinal estendido e, em seguida, é atribuído o novo tipo não assinado determinado pela conversão.

Inteiros e outros tipos também podem ser explicitamente *convertidos* de um tipo para outro. Em D, os ponteiros e os inteiros podem ser convertidos em quaisquer tipos de inteiro ou de ponteiro, mas não para outros tipos. As regras para conversão e promoção de seqüências e matrizes de caracteres são discutidas no Capítulo 6, “Seqüências”. Uma conversão de inteiro ou ponteiro é formado por meio de uma expressão tal como:

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

onde o tipo de destino é colocado entre parênteses e usado para prefixar a expressão de origem. Inteiros são convertidos em tipos de classificação superior por meio de promoção. Inteiros são convertidos em tipos de classificação quando se zera o excesso de bits de ordem alta do inteiro.

Como D não permite a aritmética de ponto flutuante, nenhuma conversão de operando de ponto flutuante é permitida e nenhuma regra para conversão de ponto flutuante implícita é definida.

Precedência

As regras de D para precedência de operador e associação são descritas na tabela seguinte. Essas regras são algo complexas, mas são necessárias para fornecer compatibilidade precisa com as regras de precedência de operador ANSI-C. As entradas da tabela estão na ordem da precedência mais alta para a precedência mais baixa.

TABELA 2–11 Precedência e associação de operador de D

Operadores	Associação
`() [] -> .`	esquerda para direita
`! ~ ++ -- + - * & (`*tipo*`) sizeof stringof offsetof xlate`	direita para esquerda
`* / %`	esquerda para direita
`+ -`	esquerda para direita
`<< >>`	esquerda para direita

TABELA 2–11 Precedência e associação de operador de D *(Continuação)*

Operadores	Associação
`< <= > >=`	esquerda para direita
`== !=`	esquerda para direita
`&`	esquerda para direita
`^`	esquerda para direita
`|`	esquerda para direita
`&&`	esquerda para direita
`^^`	esquerda para direita
`||`	esquerda para direita
`?:`	direita para esquerda
`= += -= *= /= %= &= ^= |= <<= >>=`	direita para esquerda
`,`	esquerda para direita

Existem vários operadores na tabela sobre os quais ainda não discutimos; eles serão abordados nos próximos capítulos:

`sizeof`	Calcula o tamanho de um objeto (Capítulo 7, "Structs e uniões")
`offsetof`	Calcula o deslocamento de um membro de tipo (Capítulo 7, "Structs e uniões")
`stringof`	Converte o operando em uma seqüência (Capítulo 6, "Seqüências")
`xlate`	Traduz um tipo de dados (Capítulo 40, "Tradutores")
unary `&`	Calcula o endereço de um objeto (Capítulo 5, "Ponteiros e matrizes")
* unário	Cancela a referência de um ponteiro a um objeto (Capítulo 5, "Ponteiros e matrizes")
`->` e `.`	Acessa um membro de uma estrutura ou tipo de união (Capítulo 7, "Structs e uniões")

O operador vírgula (`,`) listado na tabela é para compatibilidade com o operador de vírgula ANSI-C, que pode ser usado para avaliar um conjunto de expressões na ordem da esquerda para direita e retornar o valor da expressão da extrema direita. Este operador é fornecido estritamente para compatibilidade com C e geralmente não deve ser usado.

A entrada `()`, na tabela de precedência de operador, representa uma chamada a uma função. Exemplos de chamadas a funções como `printf()` e `trace()` são apresentados no Capítulo 1, "Introdução". Uma vírgula também é usada em D para listar argumentos de funções e para formar listas de chaves de matriz associativa. Essa vírgula não é a mesma que o operador vírgula

e *não* garante a avaliação da esquerda para a direita. O compilador de D não garante a ordem de avaliação de argumentos de uma função ou das chaves de uma matriz associativa. Você deve ter cuidado ao usar expressões com efeitos colaterais interativos, tal como o par de expressões `i` e `i++`, nestes contextos.

A entrada `[]` na tabela de precedência de operador representa uma matriz ou referência de matriz associativa. Exemplos de matriz associativa são apresentados no Capítulo 1, "Introdução". Um tipo especial de matriz associativa chamada *agregação* é descrito no Capítulo 9, "Agregações". O operador `[]` também pode ser usado para indexar matrizes C de tamanho fixo, como descrito no Capítulo 5, "Ponteiros e matrizes".

◆ ◆ ◆ CAPÍTULO 3

Variáveis

D fornece dois tipos básicos de variáveis a serem usadas em seus programas de rastreamento: variáveis escalares e matrizes de associação. Demonstramos brevemente o uso dessas variáveis nos exemplos do Capítulo 1. Este capítulo explora mais detalhadamente as regras de variáveis de D e como as variáveis podem ser associadas a escopos diferentes. Um tipo especial de variável de matriz, chamada *agregação*, é discutido no Capítulo 9, "Agregações".

Variáveis escalares

As variáveis escalares são usadas para representar objetos de dados de tamanho fixo individuais, tais como inteiros e ponteiros. As variáveis escalares também podem ser usadas para objetos de tamanho fixo que são compostos de um ou mais tipos primitivos ou compostos. D fornece o recurso para criar matrizes de objeto, assim como estruturas compostas. DTrace também representa seqüências como escalares de tamanho fixo, permitindo que elas cresçam para um tamanho máximo predefinido. O controle sobre o tamanho da seqüência em seu programa em D é discutido com mais detalhes no Capítulo 6, "Seqüências".

As variáveis escalares são criadas automaticamente na primeira vez em que você atribui um valor a um identificador não definido anteriormente em seu programa em D. Por exemplo, para criar uma variável escalar chamada `x` de tipo `int` , você pode simplesmente atribuir a ela um valor do tipo `int` em qualquer cláusula de teste:

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

Variáveis escalares criadas dessa forma são variáveis *globais* : seu nome e local de armazenamento de dados são definidos uma vez e ficam visíveis em cada cláusula de seu programa em D. Toda vez que você referenciar o identificador `x`, estará fazendo referência a um único local de armazenamento associado a essa variável.

Ao contrário de ANSI-C, D não requerer declarações explícitas de variáveis. Se você quiser declarar uma variável global para atribuir seu nome e tipo explicitamente antes de usá-la, coloque uma declaração fora das cláusulas de teste em seu programa, como mostrado no exemplo seguinte. Declarações de variáveis explícitas não são necessárias na maioria dos programas em D, mas às vezes são úteis quando você quiser controlar cuidadosamente os tipos de variáveis ou quando quiser começar o seu programa com um conjunto de declarações e comentários documentando as variáveis do seu programa e seus respectivos significados.

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

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

Ao contrário das declarações ANSI-C, as declarações de variáveis D podem não atribuir valores iniciais. Você deve usar uma cláusula de teste `BEGIN` para atribuir quaisquer valores iniciais. O armazenamento de todas as variáveis globais é preenchido com zeros pelo DTrace antes de você referenciar a variável.

A definição de linguagem D não limita o tamanho e o número de variáveis D, mas os limites são definidos pela implementação de DTrace e pela memória disponível no sistema. O compilador D reforçará quaisquer limitações que possam ser aplicadas no momento em que você compila o seu programa. Você pode aprender mais sobre como ajustar as opções relacionadas aos limites do programa no Capítulo 16, “Opções e ajustáveis”.

Matrizes de associação

As matrizes de associação são usadas para representar as coleções de elementos de dados que podem ser recuperados pela especificação de um nome chamado *chave*. As chaves de matriz de associação de D são formadas por uma lista de valores de expressão escalar chamado *tupla*. Você pode pensar na tupla de matriz como uma lista de parâmetros imaginários de uma função que é chamada para recuperar o valor da matriz correspondente quando você faz referência à matriz. Cada matriz de associação de D possui uma *assinatura de chave* fixa que consiste em um número fixo de elementos de tupla, onde cada elemento possui um determinado tipo fixo. Você pode definir assinaturas de chave diferentes para cada matriz em seu programa em D.

As matrizes de associação são diferentes das matrizes de tamanho fixo normais, pois não possuem limite predefinido no número de elementos. Os elementos podem ser indexados por qualquer tupla, ao contrário de usar apenas inteiros como chaves, além disso, os elementos não são armazenados em locais de armazenamento consecutivos pré-alocados. As matrizes de associação são úteis em situações em que você usaria uma tabela hash ou outra estrutura de dados de dicionário simples em um programa de linguagem C, C++ ou Java. ™ As matrizes de

associação o ajudam a criar um histórico dinâmico de eventos e estado, capturado em seu programa em D, que você pode usar para criar fluxos de controle mais complexos.

Para definir uma matriz de associação, escreva uma expressão de atribuição no formato:

nome [*chave*] = *expressão* ;

onde *nome* é qualquer identificador D válido e *chave* é uma lista separada por vírgula de uma ou mais expressões. Por exemplo, a declaração seguinte define uma matriz de associação `a` com assinatura de chave `[ int, string ]` e armazena o valor inteiro 456 em um local nomeado pela tupla `[ 123, "olá" ]`:

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

O tipo de cada objeto contido na matriz de todos os elementos de uma determinada matriz. Como `a` foi atribuída primeiro por meio do inteiro 456, cada valor subseqüente armazenado na matriz também será do tipo `int`. Você pode usar qualquer um dos operadores de atribuição definidos no Capítulo 2 para modificar elementos de matriz de associação, sujeitos às regras de operando definidas para cada operador. O compilador D produzirá uma mensagem de erro apropriada, se você tentar uma atribuição incompatível. Você pode usar qualquer tipo com uma chave de matriz de associação ou valor que você pode usar com uma variável escalar. Você não pode aninhar uma matriz de associação em outra matriz de associação como uma chave ou valor.

Você pode fazer referência a uma matriz de associação por meio de qualquer tupla que seja compatível com a assinatura de chave de matriz. As regras de compatibilidade de tupla são semelhantes às de chamadas de função e atribuições de variável: a tupla deve ser do mesmo tamanho e cada tipo da lista de parâmetros atuais deve ser compatível com o tipo correspondente na assinatura de chave formal. Por exemplo, se uma matriz de associação `x` for definida da seguinte forma:

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

então, a assinatura de chave é do tipo `unsigned long long` e os valores são do tipo `int`. Essa matriz também pode ser referenciada por meio da expressão `x['a']` porque a tupla consiste na constante de caractere `'a'` do tipo `int` e o tamanho um é compatível com a assinatura de chave `unsigned long long`, de acordo com as regras de conversão aritmética descritas em "Tipos de conversões" na página 59.

Se você precisar declarar explicitamente uma matriz de associação de D antes de usá-la, crie uma declaração do nome da matriz e da assinatura de chave fora das cláusulas de teste no código-fonte do seu programa:

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

BEGIN
{
```

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

Quando uma matriz de associação é definida, as referências a qualquer tupla ou assinatura de chave compatível são permitidas, mesmo que a tupla em questão não tenha sido atribuída anteriormente. O acesso ao elemento de matriz de associação não atribuída é definido para retornar um objeto preenchido com zeros. Uma conseqüência dessa definição é que o armazenamento subjacente não é alocado para um elemento de matriz de associação até que um valor diferente de zero seja atribuído a esse elemento. Contrariamente, atribuir um elemento de matriz de associação a zero faz com que o DTrace desaloque o armazenamento subjacente. Esse comportamento é importante porque o espaço de variável dinâmica, do qual os elementos de matriz de associação são alocados, é finito; caso o espaço termine durante uma tentativa de alocação, a alocação falhará e será gerada uma mensagem de erro indicando uma queda de variável dinâmica. Sempre atribua zero a elementos de matriz de associação que não estão mais sendo usados. Consulte o Capítulo 16, "Opções e ajustáveis" para obter outras técnicas de eliminar interrupções de variável dinâmica.

Variáveis de segmento locais

O DTrace o ajuda a declarar o armazenamento de variável local para cada segmento do segmento do sistema operacional, ao contrário das variáveis globais demonstradas anteriormente neste capítulo. As variáveis de segmento locais são úteis em situações em que você deseja ativar um este e marcar cada segmento que aciona o teste com alguma marca ou outros dados. Criar um programa para resolver esse problema é fácil em D porque as variáveis de segmento locais compartilham um nome comum em seu código D, mas se referem a um armazenamento de dados separado associado a cada segmento. As variáveis de segmento locais são referenciadas por meio da aplicação do operador `->` ao identificador especial `self`:

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

Este exemplo de fragmento de D ativa o teste na chamada do sistema `read(2)` e associa uma variável de segmento local denominada `read` a cada segmento que aciona o teste. Semelhantemente às variáveis globais, as variáveis de segmento globais são criadas automaticamente na primeira atribuição e deduzem o tipo usado no lado direito da primeira declaração de atribuição (neste exemplo, `int`).

Cada vez que a variável `self->read` é referenciada em seu programa em D, o objeto de dados referenciado é aquele associado ao que o segmento do sistema operacional estava executando quando o teste do DTrace correspondente foi acionado. Pense em uma variável de segmento local como uma matriz de associação que é indexada explicitamente por uma tupla que descreve a identidade do segmento no sistema. Uma identidade de segmento é exclusiva durante o ciclo

de vida do sistema: se o segmento sair e a mesma estrutura de dados do sistema operacional for usada para criar um novo segmento, esse segmento *não* reutilizará a mesma identidade de armazenamento de segmentos local do DTrace.

Quando você tiver definido uma variável de segmento local, é possível referenciá-la de qualquer segmento do sistema, mesmo que a variável em questão não tenha sido atribuída anteriormente para esse segmento específico. Se uma cópia do segmento da variável de segmento local ainda não tiver sido atribuída, o armazenamento de dados da cópia será definido para que seja preenchido com zeros. Como nos elementos de matriz de associação, o armazenamento subjacente não é alocado para uma variável de segmento local até que um valor diferente de zero se atribuído a ele. Como também acontece nos elementos de matriz de associação, atribuir zero a uma variável de segmento local faz com que o DTrace desaloque o armazenamento subjacente. Sempre atribua zero a variáveis de segmento locais que não estão mais sendo usada. Consulte o Capítulo 16, “Opções e ajustáveis” para obter outras técnicas de ajustar o espaço de variável dinâmica do qual as variáveis de segmento locais são alocadas.

As variáveis de segmento locais de qualquer tipo podem ser definidas em seu programa em D, incluindo matrizes de associação. Alguns exemplos de definições de variável de segmento local são:

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

Como com qualquer variável de D, você não precisa declarar variáveis de segmento locais explicitamente antes de usá-las. Se você quiser criar uma declaração mesmo assim, coloque uma fora das cláusulas do seu programa, antecedendo-a da palavra-chave `self`:

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

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

As variáveis de segmento globais são mantidas em um espaço de nome separado das variáveis globais, para que você possa reutilizar os nomes. Lembre-se de que `x` e `self->x` não serão a mesma variável, se você sobrecarregar os nomes em seu programa! O exemplo seguinte mostra como usar as variáveis de segmento locais. Em um editor de texto, digite o seguinte programa e salve-o em um arquivo chamado `rtime.d`:

EXEMPLO 3–1 `rtime.d`: tempo de cálculo gasto em `read`(2)

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

EXEMPLO 3–1 `rtime.d`: tempo de cálculo gasto em `read(2)` *(Continuação)*

```
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;
}
```

Agora vá para o shell e comece a executar o programa. Espere alguns segundos e você deverá ver alguma saída. Se nenhuma saída for exibida, tente executar alguns comandos.

```
# 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` usa uma variável de segmento local denominada `t` para capturar um carimbo de data e hora na entrada a `read(2)` por qualquer segmento. Em seguida, na cláusula de retorno, o programa imprime o tempo total gasto em `read(2)` subtraindo `self->t` do carimbo de data e hora atual. As variáveis internas de D `pid` e `tid` informam o ID do processo e o ID do segmento que está realizando o `read(2)`. Como `self->t` não é mais necessária já que essa informação está relatada, 0 é atribuído a ela para permitir que o DTrace reutilize o armazenamento subjacente associado a `t` no segmento atual.

Geralmente, você verá muitas linhas de saída mesmo que não realize nenhuma porque, em segundo plano, os processos e daemons do servidor estão executando o `read(2)` todo o tempo, mesmo que você não esteja fazendo nada. Tente alterar a segunda cláusula de `rtime.d` para usar a variável `execname` para imprimir o nome do processo que está realizando um `read(2)` para aprender mais:

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

Se você encontrar um processo que seja de particular interesse, adicione um predicado para aprender mais sobre o se comportamento de `read(2)`:

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

Variáveis de cláusula locais

Você também pode definir as variáveis de D cujo armazenamento é reutilizado em cada cláusula do programa em D. As variáveis de cláusula locais são semelhantes às variáveis automáticas em um programa da linguagem C, C++ ou Java que estão ativas durante cada chamada de uma função. Como todas as variáveis de programa em D, as variáveis de cláusula locais são criadas na primeira atribuição. Essas variáveis podem ser referenciadas e atribuídas por meio da aplicação do operador `->` ao identificador especial `this`:

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

Se você quiser declarar explicitamente uma variável de cláusula local antes de usá-la, use a palavra-chave `this`:

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

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

As variáveis de cláusula locais só ficam ativas durante um ciclo de vida de uma determinada cláusula de teste. Depois que o DTrace realiza as ações associadas às suas cláusulas em um determinado teste, o armazenamento de todas as variáveis de cláusula locais é recuperado e reutilizado pela próxima cláusula. Por isso, as variáveis de cláusula locais são as únicas variáveis D que não são inicialmente preenchidas com zeros. Observe que se o seu programa contiver várias cláusulas de um único teste, quaisquer variáveis de cláusula locais permanecerão intactas quando as cláusulas forem executadas, como mostrado no exemplo seguinte:

EXEMPLO 3–2 `clause.d`: variáveis de cláusula locais

```
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++);
}
```

Como as cláusulas são *sempre* executadas na ordem do programa, e já que as variáveis de cláusula locais persistem entre cláusulas diferentes, ativando o mesmo teste, executar o programa acima sempre produzirá a mesma saída:

```
# 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
```

Enquanto as variáveis de cláusula locais são persistentes entre as cláusulas, ativando o mesmo teste, seus valores são indefinidos na primeira cláusula executada em um determinado teste. Esteja certo de atribuir um valor apropriado a cada variável de cláusula local antes de usá-la, ou seu programa poderá ter resultados inesperados.

As variáveis de cláusula locais podem ser definidas por meio de qualquer tipo de variável escalar, mas as matrizes de associação não podem ser definidas através de escopo de cláusula local. O escopo de variáveis de cláusula locais só se aplica aos dados da variável correspondente, não ao nome e à identidade de tipo definidos para a variável. Quando uma variável de cláusula local é definida, este nome e assinatura de tipo podem ser usados em qualquer cláusula do programa em D subseqüente. Você não pode contar que o local de armazenamento seja o mesmo entre cláusulas diferentes.

Você pode usar variáveis de cláusula globais para acumular resultados intermediários de cálculos ou como cópia de outras variáveis. O acesso a uma variável de cláusula global é muito mais rápido do que o acesso a uma matriz de associação. Portanto, se você precisar fazer referência a um valor de matriz de associação várias vezes na mesma cláusula do programa em D, é mais eficaz copiá-la primeiro para uma variável de cláusula local e, em seguida, referenciar a variável locai repetidamente.

Variáveis internas

A tabela seguinte fornece uma lista de variáveis internas de D. Todas essas variáveis são variáveis globais escalares; nenhuma variável de segmento local, ou variável de cláusula local ou matrizes de associação internas, estão definidas no momento por D.

TABELA 3–1 Variáveis internas do DTrace

Tipo e nome	Descrição
`int64_t arg0, ..., arg9`	Os dez primeiros argumentos de entrada para um teste representado como inteiros brutos de 64 bits. Se menos de dez argumentos forem passados para o teste atual, as variáveis restantes retornarão zero.

TABELA 3–1 Variáveis internas do DTrace *(Continuação)*

Tipo e nome	Descrição
`args[]`	Os argumentos digitados para o teste atual, se houver. A matriz `args[]` é acessada por meio de um índice de inteiro, mas cada elemento é definido como o tipo correspondente ao argumento do teste determinado. Por exemplo, se `args[]` for referenciado por um teste de chamada do sistema `read(2)`, `args[0]` é do tipo `int`, `args[1]` é do tipo `void *` e `args[2]` é do tipo `size_t`.
`uintptr_t caller`	O local do contador do programa do segmento atual um pouco antes de entrar no teste atual.
`chipid_t chip`	O identificador de chipe da CPU do chipe físico atual. Para obter mais informações, consulte o Capítulo 26, "Provedor `sched`".
`processorid_t cpu`	O identificador de CPU da CPU atual. Para obter mais informações, consulte o Capítulo 26, "Provedor `sched`".
`cpuinfo_t *curcpu`	As informações de CPU da CPU atual. Para obter mais informações, consulte o Capítulo 26, "Provedor `sched`".
`lwpsinfo_t *curlwpsinfo`	O estado de processo leve (LWP) do LWP associado ao segmento atual. Esta estrutura é descrita mais detalhadamente na página do manual `proc(4)`.
`psinfo_t *curpsinfo`	O estado de processo do processo associado ao segmento atual. Esta estrutura é descrita mais detalhadamente na página do manual `proc(4)`.
`kthread_t *curthread`	O endereço da estrutura de dados interna do kernel do sistema operacional do segmento atual, `kthread_t`. `kthread_t` é definido em `<sys/thread.h>`. Consulte *Solaris Internals* para obter mais informações sobre esta variável e outras estruturas de dados do sistema operacional.
`string cwd`	O nome do diretório de trabalho atual do processo associado ao segmento atual.
`uint_t epid`	O ID de teste ativado (EPID) do teste atual. Este inteiro identifica exclusivamente um teste em particular com um predicado específico e um conjunto de ações.
`int errno`	O valor de erro retornado pela última chamada do sistema executada por este segmento.

TABELA 3–1 Variáveis internas do DTrace *(Continuação)*

Tipo e nome	Descrição
`string execname`	O nome que foi passado a `exec(2)` para que execute o processo atual.
`gid_t gid`	O ID de grupo real do processo atual.
`uint_t id`	O ID de teste do teste atual. Este ID é o identificador de teste exclusivo do sistema conforme publicado por DTrace e listado na saída de `dtrace -l`.
`uint_t ipl`	O nível de prioridade de saída (IPL) da CPU atual na hora do acionamento do teste. Consulte *Solaris Internals* para obter mais informações sobre os níveis de interrupção e manipulação de interrupção no kernel do sistema operacional.
`lgrp_id_t lgrp`	O ID de grupo de latência do grupo de latência do qual a CPU é um membro. Para obter mais informações, consulte o Capítulo 26, "Provedor `sched`".
`pid_t pid`	O ID de processo do processo atual.
`pid_t ppid`	O ID de processo pai do processo atual.
`string probefunc`	A parte do nome de função da descrição do teste atual.
`string probemod`	A parte do nome de módulo da descrição do teste atual.
`string probename`	A parte do nome da descrição do teste atual.
`string probeprov`	A parte do nome do provedor da descrição do teste atual.
`psetid_t pset`	O ID de conjunto de processadores do conjunto de processadores que contém a CPU atual. Para obter mais informações, consulte o Capítulo 26, "Provedor `sched`".
`string root`	O nome do diretório raiz do processo associado ao segmento atual.
`uint_t stackdepth`	A profundidade da moldura da porta na hora de acionamento do teste.
`id_t tid`	O ID de segmento do segmento atual. Para segmentos associados aos processos do usuário, este valor é igual ao resultado de uma chamada ao `pthread_self(3C)`.

TABELA 3–1 Variáveis internas do DTrace *(Continuação)*

Tipo e nome	Descrição
`uint64_t timestamp`	O valor atual de um contador de carimbo de data e hora em nanossegundos. Este contador é incrementado a partir de um ponto arbitrário no passado e deve ser usado somente para cálculos relativos.
`uid_t uid`	O ID de usuário real do processo atual.
`uint64_t uregs[]`	O segmento atual salvou os valores de registro de modo do usuário na hora de acionamento do teste. O uso da matriz `uregs[]` é discutido no Capítulo 33, "Rastreio de processo do usuário".
`uint64_t vtimestamp`	O valor atual de um contador de carimbo de data e hora em nanossegundos que é virtualizado no total de tempo que o segmento atual está sendo executado em uma CPU, menos o tempo gasto nos predicados e ações do DTrace. Este contador é incrementado a partir de um ponto arbitrário no passado e deve ser usado somente para cálculos relativos de tempo.
`uint64_t walltimestamp`	O número atual de nanossegundos desde 00:00 (Horário Coordenado Universal), 1 de janeiro de 1970.

As funções internas da linguagem D, tal como `trace()`, são discutidas no Capítulo 10, "Ações e sub-rotinas".

Variáveis externas

D usa o caractere de aspa invertida (‘) como um operador de escopo especial para acessar variáveis que são definidas no sistema operacional e não em seu programa em D. Por exemplo, o kernel do Solaris contém uma declaração C de um sistema ajustável chamado `kmem_flags` para ativar os recursos de depuração do alocador de memória. Consulte o *Solaris Tunable Parameters Reference Manual* para obter informações sobre `kmem_flags`. Este ajuste é declarado como uma variável C no código-fonte do kernel da seguinte forma:

```
int kmem_flags;
```

Para acessar o valor dessa variável em um programa em D, use a notação D:

```
‘kmem_flags
```

O DTrace associa cada símbolo de kernel ao tipo usado para o símbolo no código C do sistema operacional correspondente, fornecendo acesso fácil baseado na origem às estruturas de dados

do sistema operacional nativo. Para usar as variáveis externas do sistema operacional, você precisará acessar o código-fonte do sistema operacional correspondente.

Quando acessa as variáveis externas a partir de um programa em D, você está acessando os detalhes de implementação interna de outro programa, tal como o kernel do sistema operacional ou seus drivers de dispositivo. Esses detalhes de implementação não formam uma interface estável com a qual você possa contar! Quaisquer programas em D que você escreva que dependam desses detalhes podem parar de funcionar quando você fizer a próxima atualização do software correspondente. Por esse motivo, as variáveis externas são geralmente usadas pelo kernel, desenvolvedores de driver de dispositivo e pessoal de serviço, a fim de depurar o desempenho ou problemas de funcionalidade, usando o DTrace. Para aprender mais sobre a estabilidade dos seus programas em D, consulte o Capítulo 39, “Estabilidade”.

Os nomes de símbolo de kernel são mantidos em um espaço de nome separado da variável D e dos identificadores de função, para que você nunca tenha que se preocupar que esses nomes entrem em conflito com as suas variáveis D. Quando você coloca um prefixo em uma variável com uma aspa invertida, o compilador D procura os símbolos de kernel conhecidos para usar a lista de módulos carregados a fim de encontrar uma definição de variável correspondente. Como o kernel do Solaris aceita módulos carregados dinamicamente com espaços de nome de símbolo separados, o mesmo nome de variável deve ser usado mais de uma vez no kernel do sistema operacional ativo. Você pode resolver esses conflitos de nome especificando o nome do módulo de kernel cuja variável deve ser acessada antes da aspa invertida no nome do símbolo. Por exemplo, cada módulo de kernel carregável geralmente fornece uma função `_fini`(9E), portanto, para fazer referência ao endereço da função `_fini` fornecida por um módulo de kernel chamado `foo`, você deve escrever:

```
foo`_fini
```

Você pode aplicar qualquer um dos operadores D a variáveis externas, exceto aqueles que modificam valores, sujeitos às regras usuais de tipos de operando. Quando você inicia o DTrace, o compilador D carrega o conjunto de nomes de variáveis correspondentes aos módulos do kernel ativo, portanto, as declarações dessas variáveis não são necessárias. Você não pode aplicar qualquer operador a uma variável externa que modifique seus valores, tais como = ou +=. Por motivos de segurança, o DTrace evita que você danifique ou corrompa o estado do software que está observando.

◆ ◆ ◆ CAPÍTULO 4

Estrutura de programa em D

Os programas em D consistem em um conjunto de cláusulas que descrevem testes a serem ativados, além de predicados e ações para vincular a esses testes. Os programas em D também podem conter declarações de variáveis, conforme descrito no Capítulo 3, "Variáveis", e definições de novos tipos, descritos no Capítulo 8, "Definições de tipo e de constante". Este capítulo descreve formalmente a estrutura geral de um programa em D e dos recursos para construir descrições de teste que correspondem a mais de um teste. Também iremos discutir o uso do pré-processador C, `cpp`, com programas em D.

Cláusulas e declarações de teste

Como mostrado em nossos exemplos até aqui, um arquivo-fonte de programa em D consiste em uma ou mais cláusulas de teste que descrevem a instrumentação a ser ativada por DTrace. Cada cláusula de teste possui o formato geral:

```
descrições de teste
/ predicado /
{
   declarações de ação
}
```

O predicado e a lista de declarações de ações podem ser omitidos. Quaisquer diretivas encontradas fora das cláusulas de teste são chamadas de *declarações*. As declarações só podem ser usadas fora das cláusulas de teste. Não são permitidas declarações entre `{ }` e as declarações não podem ser colocadas entre os elementos da cláusula de teste mostrada acima. Pode ser usado espaço em branco para separar quaisquer elementos de programa em D e para recuar as declarações de ação.

As declarações podem ser usadas para declarar variáveis de D e símbolos de C externos, conforme discutido no Capítulo 3, "Variáveis", ou para definir novos tipos para uso em D, conforme descrito no Capítulo 8, "Definições de tipo e de constante". As diretivas de

compilador de D especiais chamadas *pragmas* também podem aparecer em qualquer lugar em um programa em D, inclusive fora das cláusulas de teste. Os pragmas de D são especificados em linhas que começam com um caractere #. Os pragmas de D são usados, por exemplo, para definir as opções de tempo de execução do DTrace. Consulte o Capítulo 16, “Opções e ajustáveis” para obter detalhes.

Descrições de teste

Cada cláusula do programa em D começa com uma lista de uma ou mais descrições de teste, cada uma tem o formato usual:

provedor:*módulo*:*função*:*nome*

Se um ou mais campos da descrição de teste forem omitidos, os campos especificados serão interpretados da direita para a esquerda pelo compilador de D. Por exemplo, a descrição de teste `foo:bar` corresponderia a um teste com função `foo` e nome `bar`, independentemente do valor do provedor de teste e dos campos de módulo. Portanto, uma descrição de teste é realmente exibida de forma mais correta como um *padrão* que pode ser usado para corresponder a um ou mais testes com base em seus nomes.

Você deve escrever suas descrições de teste de D especificando todos os quatro delimitadores de campo que podem ser especificados, para que possa especificar o *provedor* desejado no lado esquerdo. Se você não especificar o provedor, talvez obtenha resultados inesperados, se vários provedores publicarem testes com o mesmo nome. Semelhantemente, as versões futuras do DTrace talvez incluam provedores cujos testes correspondam não intencionalmente às suas descrições de teste parcialmente especificadas. Você pode especificar um provedor mas coincidir qualquer um dos seus respectivos testes, deixando em branco qualquer um dos campos de módulo, função e nome. Por exemplo, a descrição `syscall:::` pode ser usada para coincidir cada teste publicado pelo provedor `syscall` do DTrace.

As descrições de teste também oferecem suporte a uma sintaxe de correspondência padrão semelhante à sintaxe de correspondência padrão *globbing* do shell descrita em `sh(1)`. Antes de coincidir um teste com uma descrição, o DTrace examina cada campo de descrição dos caracteres `*`, `?` e `[`. Se um desses caracteres aparecer em um campo de descrição de teste e não for precedido por um `\`, o campo será considerado como um padrão. O padrão de descrição deve corresponder a todo o campo correspondente de um determinado teste. A descrição de teste completa deve coincidir em cada campo para que corresponda e ative um teste com êxito. Um campo de descrição de teste que não seja um padrão deve coincidir com o campo de teste correspondente. Um campo de descrição vazio corresponde a qualquer teste.

Os caracteres especiais na tabela seguinte são reconhecidos em padrões de nome de teste:

TABELA 4–1 Caracteres correspondentes do padrão de nome de teste

Símbolo	Descrição
*	Corresponde a qualquer seqüência, incluindo a seqüência nula.
?	Corresponde a qualquer caractere único.
[...]	Corresponde a qualquer um dos caracteres incluídos. Um par de caracteres separados por - corresponde a qualquer caractere dentro do par, inclusive. Se o primeiro caractere após [for !, qualquer caractere não incluído no conjunto será correspondido.
\	Interpreta o próximo caractere como ele mesmo, sem qualquer significado especial.

Caracteres de correspondência de padrão podem ser usados em qualquer um dos quatro campos das suas descrições de teste. Você também pode usar padrões para listar testes correspondentes utilizando padrões na linha de comando com `dtrace -l`. Por exemplo, o comando `dtrace -l -f kmem_*` lista todos os testes do DTrace em funções cujos nomes começam com o prefixo `kmem_`.

Se você quiser especificar os mesmos predicados e ações para mais de uma descrição de teste ou padrão de descrição, coloque as descrições em uma lista separada por vírgulas. Por exemplo, o programa em D rastrearia um carimbo de data e hora toda vez que os testes associados a chamadas de entrada no sistema contendo as palavras "`lwp`" ou "`sock`" fossem acionadas:

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

Uma descrição de teste também pode especificar um teste usando seu ID de teste de inteiro. Por exemplo, a cláusula:

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

poderia ser usada para ativar o ID de teste 12345, conforme relatado por `dtrace -l -i 12345`. Você deve sempre escrever seus programas em D usando descrições de teste legíveis. Não há garantias de que o ID de teste de inteiro permaneça consistente, conforme os módulos de kernel do provedor do DTrace são carregados e descarregados, ou após uma reinicialização.

Predicados

Os predicados são expressões entre barras `/ /` que são avaliadas na hora em que o teste é acionado para determinar se as ações associadas devem ser executadas. Os predicados são a construção condicional principal usada para construir um fluxo de controle mais complexo em um programa em D. Você pode omitir inteiramente a seção de predicado da cláusula de teste de qualquer teste, neste caso, as ações são sempre executadas quando o teste é acionado.

As expressões de predicado podem usar qualquer um dos operadores de D descritos anteriormente e podem fazer referência a quaisquer objetos de dados de D, tais como variáveis e constantes. A expressão de predicado deve ser avaliada como um valor inteiro ou tipo de ponteiro, para que possa ser considerada verdadeira ou falsa. Como em todas as expressões de D, um valor zero é interpretado como falso e um valor diferente de zero é interpretado como verdadeiro.

Ações

As ações de teste são descritas por uma lista de declarações separadas por ponto-e-vírgula (`;`) e colocadas entre chaves `{ }`. Se você quiser apenas anotar que um teste foi acionado em uma CPU específica sem rastrear quaisquer dados ou realizar quaisquer ações adicionais, especifique um conjunto de chaves vazio sem declarações dentro.

Uso do pré-processador C

A linguagem de programação em C usada para definir as interfaces do sistema Solaris inclui um *pré-processador* que realiza um conjunto de etapas iniciais na compilação do programa C. Os pré-processadores C são comumente usados para definir substituições de macro, onde um símbolo em um programa C é substituído por outro conjunto de símbolos predefinidos, ou para incluir cópias de arquivos de cabeçalho do sistema. Você pode usar o pré-processador de C junto com seus programas em D, especificando a opção `-C` do `dtrace`. Esta opção faz com que o `dtrace` execute primeiro o pré-processador `cpp(1)` no arquivo-fonte do seu programa e, em seguida, passe os resultados para o compilador de D. O pré-processador C é descrito mais detalhadamente em *The C Programming Language*.

O compilador de D carrega automaticamente o conjunto de descrições de tipos associadas à implementação do sistema operacional, mas você pode usar o pré-processador para incluir outras definições de tipo, tais como tipos usados em seus próprios programas em C. Você também pode usar o pré-processador para realizar outras tarefas, tais como criar macros que se expandam em blocos de código de D e outros elementos do programa. Se você usar o pré-processador com o seu programa em D, talvez só possa incluir arquivos que contêm declarações de D válidas. Os arquivos de cabeçalho C típicos incluem apenas declarações externas de tipos e símbolos, que serão interpretadas corretamente pelo compilador de D. O

compilador de D não pode analisar os arquivos de cabeçalho de C que incluem elementos de programa adicionais, tais como código-fonte da função de C, e produzirá uma mensagem de erro apropriada.

◆ ◆ ◆ CAPÍTULO 5

Ponteiros e matrizes

Ponteiros são endereços de memória de objetos de dados no kernel do sistema operacional ou no espaço de endereço de um processo do usuário. D ajuda você a criar e manipular ponteiros, além de armazená-los em variáveis e matrizes de associação. Este capítulo descreve a sintaxe D dos ponteiros, os operadores que podem ser aplicados para criar ou acessar ponteiros e a relação entre ponteiros e matrizes escalares de tamanho fixo. Também são discutidos problemas relacionados ao uso dos ponteiros em espaços de endereço diferentes.

Observação – Se você for um programador experiente em C ou C++, poderá ler rapidamente este capítulo, pois a sintaxe de ponteiro de D é a mesma sintaxe ANSI-C correspondente. Você deve ler o "Ponteiros para objetos do DTrace" na página 90 e o "Ponteiros e espaços de endereço" na página 91, pois eles descrevem os recursos e os problemas específicos do DTrace.

Ponteiros e endereços

O sistema operacional Solaris usa uma técnica chamada *memória virtual* para fornecer a cada usuário os processos com sua própria visão virtual dos recursos de memória do sistema. Uma visão virtual dos recursos de memória é chamada de *espaço de endereço*, que associa um intervalo de valores de endereço (`[0 ... 0xffffffff]` para um espaço de endereço de 32 bits ou `[0 ... 0xffffffffffffffff]` para um espaço de endereço de 64 bits) a um conjunto de translações que o sistema operacional e o hardware usam para converter cada endereço virtual em um local de memória física correspondente. Os ponteiros em D são objetos de dados que armazenam um valor de endereço virtual de inteiro e associa-o a um tipo de D que descreve o formato dos dados armazenados no local correspondente da memória.

Você pode declarar uma variável de D como o tipo de ponteiro, especificando primeiro o tipo dos dados referenciados e, em seguida, acrescentando um asterisco (`*`) ao nome do tipo para indicar que deseja declarar um tipo de ponteiro. Por exemplo, a declaração:

```
int *p;
```

declara uma variável de D global chamada p que é um ponteiro para um inteiro. Esta declaração significa que a própria p é um inteiro de 32 ou 64 bits, cujo valor é o endereço de outro inteiro localizado em algum lugar da memória. Como o formato compilado do seu código de D é executado na hora em que o teste é acionado dentro do próprio kernel do sistema operacional, os ponteiros de D são geralmente associados ao espaço de endereço do kernel. Você pode usar o comando `isainfo(1)` `-b` para determinar o número de bits usado para ponteiros pelo kernel ativo do sistema operacional.

Se você quiser criar um ponteiro para um objeto de dados dentro do kernel, calcule seu endereço usando o operador &. Por exemplo, o código-fonte do kernel do sistema operacional declara um `int` `kmem_flags` ajustável. Você poderia rastrear o endereço deste `int` rastreando o resultado da aplicação do operador & ao nome desse objeto em D:

```
trace(&‘kmem_flags);
```

O operador * pode ser usado para fazer referência ao objeto endereçado pelo ponteiro, e age inversamente ao operador &. Por exemplo, os dois fragmentos de código de D seguintes têm significados equivalentes:

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

O fragmento esquerdo cria um ponteiro de variável de D global `p`. Como o objeto `kmem_flags` é do tipo `int`, o tipo do resultado de &‘`kmem_flags` é `int *` (ou seja, o ponteiro para `int`). O fragmento esquerdo rastreia o valor de `*p`, que segue o ponteiro de volta para o objeto de dados `kmem_flags`. Portanto, esse fragmento é o mesmo que o fragmento direito, que simplesmente rastreia o valor do objeto de dados diretamente através do seu nome.

Segurança de ponteiro

Se você for um programador em C ou C++, talvez esteja um pouco assustado depois de ler a seção anterior porque sabe que o uso incorreto de ponteiros ponde fazer com que os seus programas travem. O DTrace é um ambiente seguro e robusto para executar seus programas em D, onde esses erros não podem fazer com que o seu programa trave. Você pode até escrever um programa em D com erros, mas acessos inválidos ao ponteiro de D não farão com que o DTrace ou o kernel do sistema operacional falhe ou trave. Em vez disso, o software do DTrace detectará quaisquer acessos inválidos ao ponteiro, desativará sua instrumentação e informará o problema para que você faça a depuração.

Se você programou na linguagem de programação Java, provavelmente sabe que a linguagem Java não aceita ponteiros exatamente pelos mesmos motivos de segurança. Os ponteiros são necessários em D porque eles são uma parte intrínseca da implementação do sistema operacional em C, mas o DTrace implementa os mesmos tipos de mecanismo de segurança encontrados na linguagem de programação Java que evitam que programas com erro causem

danos neles mesmos ou em outros programas. O relatório de erros do DTrace é semelhante ao ambiente de tempo de execução da linguagem de programação Java, que detecta um erro de programação e informa uma exceção para você.

Para ver o relatório e a manipulação de erros do DTrace, escreva deliberadamente um programa D ruim usando ponteiros. Em um editor, digite o seguinte programa em D e salve-o em um arquivo chamado `badptr.d`:

EXEMPLO 5–1 `badptr.d`: demonstração de manipulação de erros do DTrace

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

O programa `badptr.d` cria um ponteiro de D chamado `x` que é um ponteiro para `int`. O programa atribui a esse ponteiro o valor de ponteiro inválido especial `NULL`, que é um alias interno para o endereço 0. Por convenção, o endereço 0 é sempre definido como inválido, para que `NULL` possa ser usado como um valor de sentinela em programas em C e em D. O programa usa uma expressão de conversão para converter `NULL` a fim de que seja usado como um ponteiro para um inteiro. O programa cancela a referência ao ponteiro usando a expressão `*x`, atribui o resultado a outra variável `y` e tenta rastrear `y`. Quando o programa em D é executado, o DTrace detecta um acesso inválido ao ponteiro quando a declaração `y = *x` é executada e informa o erro:

```
# 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
#
```

O outro problema que pode surgir dos programas que usam ponteiros inválidos é um *erro de alinhamento*. Por convenção arquitetural, os objetos de dados fundamentais, tais como inteiros, são alinhados na memória de acordo com o su tamanho. Por exemplo, inteiros de 2 bytes são alinhados em endereços que são múltiplos de 2, inteiros de 4 bytes em múltiplos de 4, e assim por diante. Se você cancelar a referência de um ponteiro a um inteiro de 4 bytes e o endereço do ponteiro for um valor inválido que não seja múltiplo de 4, o acesso falhará devido a um erro de alinhamento. Os erros de alinhamento em D quase sempre indicam que o ponteiro possui um valor inválido ou corrompido devido a um erro em seu programa em D. Você pode criar um

erro de alinhamento de exemplo, alterando o código-fonte de `badptr.d` para usar o endereço `(int *)2` em vez de `NULL`. Como `int` é de 4 bytes e 2 não é um múltiplo de 4, a expressão `*x` resulta em um erro de alinhamento do DTrace.

Para obter detalhes sobre o mecanismo de erro do DTrace, consulte "Teste ERROR" na página 203.

Declarações e armazenamento de matriz

D oferece suporte a *matrizes escalares* além das matrizes de associação dinâmicas descritas no Capítulo 3. As matrizes escalares são um grupo de tamanho fixo de locais de memória consecutivos que armazenam um valor do mesmo tipo. As matrizes escalares são acessadas quando se faz referência a cada local que tenha um inteiro começando a partir de zero. As matrizes escalares correspondem diretamente em conceito e sintaxe às matrizes em C e C++. As matrizes escalares não são usadas tão freqüentemente em D como as matrizes de associação e suas contrapartes mais avançadas, as *agregações*, mas elas às vezes são necessárias durante o acesso a estruturas de dados de matriz existentes do sistema operacional declaradas em C. As agregações são descritas no Capítulo 9, "Agregações".

Uma matriz escalar de D de 5 inteiros será declarada através do tipo `int` e da colocação de um sufixo na declaração com o número de elementos entre colchetes, da seguinte forma:

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

O diagrama seguinte mostra uma representação visual do armazenamento da matriz:

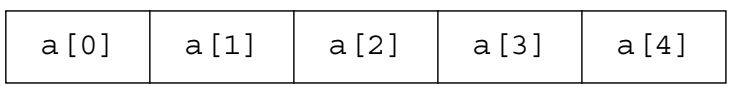

FIGURA 5–1 Representação de matriz escalar

A expressão de D `a[0]` é usada para fazer referência ao primeiro elemento da matriz, `a[1]` se refere ao segundo, e assim por diante. De uma perspectiva sintática, as matrizes escalares e as matrizes de associação são muito semelhantes. Você pode declarar uma matriz associativa de cinco inteiros referenciada por uma chave de inteiro, da seguinte forma:

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

e também pode referenciar essa matriz usando a expressão `a[0]`. Mas de uma perspectiva de armazenamento e de implementação, as duas matrizes são muito diferentes. A matriz estática `a` consiste em cinco locais de memória consecutivos numerados a partir de zero e o índice se refere a um desvio no armazenamento alocado da matriz. Por outro lado, uma matriz associativa não tem tamanho predefinido e não armazena elementos em locais de memória

consecutivos. Além disso, as chaves de matriz de associação não têm relação com o local de armazenamento do valor correspondente. Você pode acessar os elementos de matriz associativa `a[0]` e `a[-5]`, e apenas duas palavras de armazenamento, que podem ou não ser consecutivas, serão alocadas pelo DTrace. As chaves de matriz associativa são nomes abstratos do valor correspondente que não tem relação com os locais de armazenamento de valor.

Se você criar uma matriz usando uma atribuição inicial e utilizar uma única expressão de inteiro como o índice de matriz (por exemplo, `a[0] = 2`), o compilador de D sempre criará uma nova matriz associativa, embora nessa expressão `a` também pudesse ser interpretado como uma atribuição a uma matriz escalar. As matrizes escalares devem ser declaradas previamente nesta situação, para que o compilador de D possa ver a definição do tamanho da matriz e deduzir que a matriz é uma matriz escalar.

Relação entre ponteiro e matriz

Os ponteiros e as matrizes possuem uma relação especial em D, assim como em ANSI-C. Uma matriz é representada por uma variável que é associada ao endereço do seu primeiro local de armazenamento. Um ponteiro também é o endereço de um local de armazenamento com um tipo definido, sendo assim, D permite o uso da notação de índice [ ] de matriz com variáveis de ponteiro e variáveis de matriz. Por exemplo, os dois fragmentos de D seguintes têm significados semelhantes:

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

No fragmento esquerdo, o ponteiro `p` é atribuído ao endereço do primeiro elemento de matriz em `a` por meio da aplicação do operador `&` à expressão `a[0]`. A expressão `p[2]` rastreia o valor do terceiro elemento da matriz (índice 2). Como `p` agora contém o mesmo endereço associado a `a`, esta expressão possui o mesmo valor que `a[2]`, mostrado no fragmento direito. Uma conseqüência dessa equivalência é que C e D permitem que você acesse qualquer índice de qualquer ponteiro ou matriz. O compilador ou o ambiente de tempo de execução do DTrace não realizam a verificação dos limites da matriz para você. Se você acessar a memória após um valor predefinido de uma matriz, obterá um resultado inesperado ou o DTrace reportará um erro de endereço inválido, como mostrado no exemplo anterior. Como sempre, você não pode danificar o próprio DTrace ou o sistema operacional, mas precisará depurar o seu programa em D.

A diferença entre ponteiros e matrizes é que uma variável de ponteiro se refere a um pedaço separado do armazenamento que contém o endereço do inteiro de algum outro armazenamento. Uma variável nomeia o próprio armazenamento da matriz, não o local de um inteiro que em compensação contém o local da matriz. Esta diferença é ilustrada no diagrama seguinte:

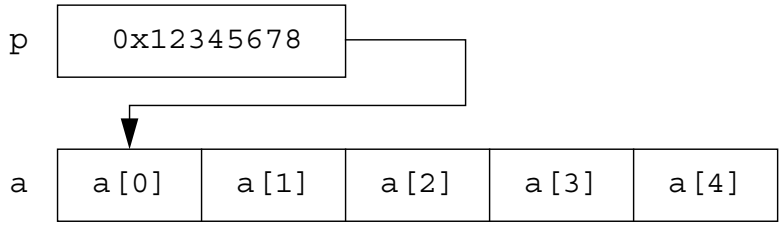

FIGURA 5–2 Armazenamento de ponteiro e de matriz

Esta diferença será manifestada na sintaxe de D, se você tentar atribuir ponteiros e matrizes escalares. Se `x` e `y` forem variáveis de ponteiro, a expressão `x = y` será legal. Ela simplesmente copia o endereço do ponteiro em `y` para o local de armazenamento nomeado por `x`. Se `x` e `y` forem variáveis de matriz escalar, a expressão `x = y` não será legal. As matrizes não podem ser atribuídas como um todo em D. Entretanto, uma variável de matriz ou nome de símbolo pode ser usado em qualquer contexto em que um ponteiro seja permitido. Se `p` for um ponteiro e `a` for uma matriz, a declaração `p = a` será permitida; essa declaração equivale a `p = &a[0]`.

Aritmética de ponteiro

Já que os ponteiros são apenas inteiros usados como endereços de outros objetos na memória, D fornece um conjunto de recursos para realizar aritmética em ponteiros. Entretanto, a aritmética de ponteiro não é idêntica à aritmética de inteiro. A aritmética de ponteiro ajusta implicitamente o endereço subjacente, multiplicando ou dividindo os operandos pelo tamanho do tipo referenciado pelo ponteiro. O fragmento de D seguinte ilustra esta propriedade:

```
int *x;

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

Este fragmento cria um ponteiro de inteiro `x` e depois rastreia o seu valor, seu valor incrementado por um e seu valor incrementado por dois. Se você criar e executar este programa, o DTrace informará os valores inteiros 0, 4 e 8.

Como `x` é um ponteiro para um int (de 4 bytes), incrementar `x` adiciona 4 ao valor de ponteiro subjacente. Esta propriedade é útil quando os ponteiros são usados para fazer referência a locais de armazenamento consecutivos, tais como matrizes. Por exemplo, se `x` fosse atribuído ao endereço de uma matriz `a` conforme a mostrada na Figura 5–2, a expressão `x + 1` seria equivalente à expressão `&a[1]`. Similarmente, a expressão `*(x + 1)` se referiria ao valor `a[1]`. A

aritmética de ponteiro é implementada pelo compilador de D sempre que um valor de ponteiro é incrementado por meio dos operadores +=, + ou ++.

A aritmética de ponteiro também é aplicada quando um inteiro é subtraído de um ponteiro no lado esquerdo, quando um ponteiro é subtraído de outro ponteiro, ou quando o operador `--` é aplicado a um ponteiro. Por exemplo, o programa em D seguinte rastrearia o resultado 2:

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

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

Ponteiros genéricos

Às vezes, é útil representar ou manipular um endereço de ponteiro genérico em um programa em D sem especificar o tipo de dados referenciado pelo ponteiro. Os ponteiros genéricos podem ser especificados por meio do tipo `void *`, onde a palavra-chave `void` representa a ausência de informações de tipo específico, ou por meio do alias de tipo interno `uintptr_t` , que é o alias de um tipo de inteiro não assinado do tamanho apropriado para um ponteiro no modelo de dados atual. Você não pode aplicar a aritmética de ponteiro a um objeto do tipo `void *`, e esses ponteiros não podem ter a referência cancelada sem serem convertidos primeiro em outro tipo. Você pode converter um ponteiro para o tipo `uintptr_t` quando precisar realizar uma aritmética de inteiro no valor do ponteiro.

Os ponteiros para `void` podem ser usados em qualquer contexto em que é necessário um ponteiro para outro tipo de dados, tal como uma expressão de tupla de matriz associativa ou o lado direito de uma declaração de atribuição. Similarmente, um ponteiro para qualquer tipo de dados pode ser usado em um contexto em que é necessário um ponteiro para `void` . Para usar um ponteiro para um tipo não-`void` em vez de outro tipo de ponteiro não-`void`, é necessária uma conversão explícita. Você deve sempre usar conversões explícitas para converter ponteiros em tipos de inteiros, tal como `uintptr_t`, ou para reconverter esses inteiros para o tipo de ponteiro apropriado.

Matrizes multidimensionais

As matrizes escalares multidimensionais são usadas freqüentemente em D, mas são fornecidas para compatibilidade com ANSI-C e para observar e acessar estruturas de dados do sistema operacional, criadas por meio desse recurso em C. Uma matriz multidimensional é declarada como uma série consecutiva de tamanhos de matriz escalar entre colchetes `[ ]` seguindo o tipo base. Por exemplo, para declarar uma matriz retangular bidimensional de tamanho fixo de inteiros com dimensões de 12 linhas por 34 colunas, você escreveria a declaração:

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

Uma matriz escalar multidimensional é acessada por meio de uma notação semelhante. Por exemplo, para acessar o valor armazenado na linha 0 coluna 1, você escreveria a expressão de D:

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

Os locais de armazenamento de valores de matriz escalar multidimensional são calculados pela multiplicação do número de linhas pelo número total de colunas declaradas e, em seguida, pela adição do número de colunas.

Você deve ter cuidado para não confundir a sintaxe de matriz multidimensional com a sintaxe de D de acessos de matriz associativa (ou seja, `a[0][1]` não é o mesmo que `a[0, 1]`). Se você usar uma tupla incompatível com uma matriz associativa ou tentar acessar uma matriz associativa de uma matriz escalar, o compilador de D informará uma mensagem de erro apropriada e recusará a compilação do seu programa.

Ponteiros para objetos do DTrace

O compilador de D proíbe o uso do operador & para obter ponteiros para objetos do DTrace, tais como matrizes de associação, funções internas e variáveis. Você não pode obter o endereço dessas variáveis, para que o ambiente de tempo de execução do DTrace esteja livre para realocá-las, conforme necessário, entre os acionamentos de teste, de forma a gerenciar de maneira mais eficaz a memória necessária para os seus programas. Se você cria estruturas compostas, é possível construir expressões que recuperem o endereço do kernel do armazenamento do objeto do DTrace. Você deve evitar a criação de tais expressões em seus programas em D. Se você precisar usar uma expressão como essa, certifique-se de não armazenar em cache o endereço entre os acionamentos de teste.

Em ANSI-C, os ponteiros também podem ser usados para realizar chamadas de função indireta ou atribuições, tal como a criação de uma expressão usando o operador unário de cancelamento de referência * no lado esquerdo de um operador de atribuição. Em D, esses tipos de expressões usando ponteiros não são permitidos. Você só pode atribuir valores diretamente para variáveis de D usando o nome delas ou aplicando o operador de índice de matriz `[]` a uma matriz escalar ou de associação de D. Você só pode chamar funções definidas pelo ambiente do DTrace por

nome, conforme especificado no Capítulo 10, “Ações e sub-rotinas”. As chamadas de função indiretas usando ponteiros não são permitidas em D.

Ponteiros e espaços de endereço

Um ponteiro é um endereço que fornece uma translação em algum *espaço de endereço virtual* para um pedaço de memória física. O DTrace executa os seus programas em D no espaço de endereço do próprio kernel do sistema operacional. O sistema Solaris inteiro gerencia muitos espaços de endereço: um para o kernel do sistema operacional e outro para cada processo do usuário. Já que cada espaço de endereço dá a ilusão de que pode acessar toda a memória do sistema, o mesmo valor do ponteiro do endereço virtual pode ser reutilizado entre espaços de endereço, mas é transladado para uma memória física diferente. Portanto, ao escrever programas em D que usam ponteiros, você deve estar ciente do espaço de endereço correspondente para os ponteiros que deseja usar.

Por exemplo, se você usar o provedor `syscall` para instrumentar a entrada para uma chamada ao sistema que usa um ponteiro para um inteiro ou uma matriz de inteiros como um argumento (por exemplo, `pipe(2)`), não seria válido cancelar a referência desse ponteiro ou matriz usando os operadores * ou `[]` porque o endereço em questão é de um espaço de endereço do processo do usuário que realizou a chamada ao sistema. Aplicar os operadores * ou `[]` a esse endereço em D resultaria no acesso ao espaço de endereço do kernel, que resultaria em um erro de endereço inválido ou no retorno de dados inesperados para o seu programa em D, dependendo se o endereço coincidiu com um endereço de kernel válido.

Para acessar a memória do processo do usuário de um teste do DTrace, aplique uma das funções `copyin()`, `copyinstr()` ou `copyinto()` descrita no Capítulo 10, “Ações e sub-rotinas” para o ponteiro do espaço de endereço do usuário. Tome cuidado ao escrever seus programas em D para nomear e comentar as variáveis que armazenam endereços do usuário de modo a evitar confusão. Você também pode armazenar endereços do usuário como `uintptr_t`, para que não compile acidentalmente o código de D que cancela a referência deles. As técnicas para usar o DTrace em processos do usuário são descritas no Capítulo 33, “Rastreio de processo do usuário”.

◆ ◆ ◆ CAPÍTULO 6

Seqüências

O DTrace oferece suporte para rastrear e manipular seqüências. Este capítulo descreve o conjunto completo dos recursos da linguagem D para declarar e manipular seqüências. Ao contrário de ANSI-C, as seqüências em D possuem o próprio tipo interno e suporte a operador, para que você possa usá-las facilmente, sem ambigüidade, nos seus programas de rastreio.

Representação de seqüência

As seqüências são representadas no DTrace como uma matriz de caracteres terminados por um byte nulo (ou seja, um byte cujo valor é zero, geralmente escrito como `'\0'`). A parte visível da seqüência é do tamanho da variável, dependendo do local do byte nulo, mas o DTrace armazena cada seqüência em uma matriz de tamanho fixo, para que cada teste rastreie uma quantidade consistente de dados. As seqüências não podem ultrapassar o tamanho desse limite de seqüência predefinido, mas o limite pode ser modificado em seu programa em D ou na linha de comando do `dtrace`, por meio do ajuste da opção `strsize` . Consulte o Capítulo 16, "Opções e ajustáveis" para obter mais informações sobre opções ajustáveis do DTrace. O limite de seqüência padrão é 256 bytes.

A linguagem D oferece um tipo `string` específico em vez de usar o tipo `char *` para fazer referência a seqüências. O tipo `string` é equivalente a `char *` no sentido de que é o endereço de uma seqüência de caracteres, porém o compilador e as funções de D, como `trace()`, oferecem recursos avançados quando aplicados a expressões do tipo `string`. Por exemplo, o tipo string remove a ambigüidade do tipo `char *` quando você precisa rastrear os bytes reais de uma seqüência. Na declaração de D:

```
trace(s);
```

se `s` for do tipo `char *`, o DTrace rastreará o valor do ponteiro `s` (ou seja, ele rastreará um valor de endereço de inteiro). Na declaração de D:

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

por definição do operador *, o compilador D cancelará a referência do ponteiro s e rastreará o único caractere nesse local. Esses comportamentos são essenciais para permitir que você manipule os ponteiros de caractere que por definição se referem a caracteres únicos, ou a matrizes de inteiros de tamanho de byte que não são seqüências e não terminam com um byte nulo. Na declaração de D:

```
trace(s);
```

se s for do tipo `string`, o tipo `string` indicará para o compilador de D que você deseja que o DTrace rastreie uma seqüência de caracteres terminada em nulo, cujo endereço é armazenado na variável s. Você também pode realizar uma comparação lexical de expressões do tipo `string`, conforme descrito em "Comparação de seqüências" na página 95.

Constante de seqüências

As constantes de seqüências são colocadas entre aspas duplas (") e são atribuídas automaticamente ao tipo `string` pelo compilador de D. Você também pode definir constantes de seqüências de qualquer tamanho, limitadas apenas pela quantidade de memória que o DTrace pode consumir em seu sistema. O byte nulo de terminação (`\0`) é adicionado automaticamente pelo compilador de D a quaisquer constantes de seqüências declaradas por você. O tamanho de um objeto de constante de seqüências é o número de bytes associado à seqüência mais um byte adicional para o byte nulo de terminação.

Uma constante de seqüência talvez não contenha um caractere de nova linha literal. Para criar seqüências que contêm novas linhas, use a seqüência de escape `\n` em vez de uma nova linha literal. As constantes de seqüências também contêm quaisquer uma das seqüências de escape de caractere especial definidas para constantes de caracteres na Tabela 2–5.

Atribuição de seqüências

Ao contrário da atribuição de variáveis `char *`, as seqüências são copiadas por valor, não por referência. A atribuição de seqüências é realizada por meio do operador = e copia os bytes reais da seqüência do operando de origem, incluindo o byte nulo, para a variável do lado esquerdo, que deve ser do tipo `string`. Você pode criar uma nova variável do tipo `string` atribuindo-a a uma expressão do tipo `string`. Por exemplo, a declaração de D:

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

criaria uma nova variável s do tipo `string` e copiaria os 4 bytes da seqüência "`olá`" para ela (3 caracteres imprimíveis mais o byte nulo). A atribuição de seqüências é análoga à função de biblioteca de C `strcpy(3C)`, exceto que se a seqüência de origem exceder o limite do armazenamento da seqüência de destino, a seqüência resultante será truncada automaticamente nesse limite.

Você também pode atribuir a uma variável de seqüência uma expressão de um tipo que seja compatível com seqüências. Neste caso, o compilador de D promove automaticamente a expressão de origem para o tipo string e realiza uma atribuição de seqüências. O compilador de D permite que qualquer expressão do tipo `char *` ou do tipo `char[`*n*`]` (ou seja, uma matriz escalar de `char` de qualquer tamanho), seja promovida para `string`.

Conversão de seqüências

Expressões de outros tipo podem ser convertidas explicitamente para o tipo `string` por meio de uma expressão de conversão ou através da aplicação do operador especial `stringof`, que têm significados equivalentes:

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

O operador `stringof` se vincula muito fortemente ao operando do seu lado direito. Geralmente, os parênteses são usados para cercar a expressão por motivo de clareza, embora eles não sejam estritamente necessários.

Qualquer expressão que seja de um tipo escalar, tal como um ponteiro, um inteiro, ou um endereço de matriz escalar, pode ser convertida para string. As expressões de outros tipos, tal como `void`, não podem ser convertidas para `string`. Se você converter erradamente um endereço inválido em uma seqüência, os recursos de segurança do DTrace evitarão que você danifique o sistema ou o DTrace, mas talvez você termine rastreando uma seqüência de caracteres indecifráveis.

Comparação de seqüências

D sobrecarrega os operadores relacionais binários e permite que eles sejam usados para comparações de seqüências, assim como para comparações de inteiros. Os operadores relacionais fazem a comparação de seqüências sempre que ambos os operandos forem do tipo `string`, ou quando um operando for do tipo `string` e o outro operando puder ser promovido para o tipo `string`, conforme descrito em "Atribuição de seqüências" na página 94. Todos os operadores relacionais podem ser usados para comparar seqüências:

TABELA 6–1 Operadores relacionais de D para seqüências

<	o operando esquerdo é menor que o operando direito
<=	o operando esquerdo é menor ou igual ao operando direito
>	o operando esquerdo é maior que o operador direito
>=	o operando esquerdo é maior ou igual ao operando direito

TABELA 6–1 Operadores relacionais de D para seqüências *(Continuação)*

`==`	o operando esquerdo é igual ao operando direito
`!=`	o operando esquerdo não é igual ao operando direito

Como nos inteiros, cada operador é avaliado com um valor de tipo `int` que é igual a um, se a condição for verdadeira, ou zero se for falsa.

Os operadores relacionais comparam as duas seqüências de entrada byte por byte, semelhante à rotina de biblioteca de C `strcmp(3C)`. Cada byte é comparado através do seu valor de inteiro correspondente no conjunto de caracteres ASCII, conforme mostrado em `ascii(5)`, até que um byte nulo seja lido ou o tamanho máximo de seqüência seja alcançado. Alguns exemplos de comparações de seqüências de D e seus resultados:

`"café" < "expresso"`	... retorna 1 (verdadeiro)
`"café" == "café"`	... retorna 1 (verdadeiro)
`"café" >= "mocha"`	... retorna 0 (falso)

◆ ◆ ◆ CAPÍTULO 7

Structs e uniões

As coleções de variáveis relacionadas podem ser agrupadas em objetos de dados compostos chamados *structs* e *uniões*. Você pode definir esses objetos em D criando novas definições de tipo para eles. Você pode usar seus novos tipos para quaisquer variáveis de D, incluindo valores de matriz de associação. Este capítulo explora a sintaxe e a semântica para a criação e manipulação desses tipos compostos e os operadores de D que interagem com eles. A sintaxe das structs e das uniões é ilustrada através de vários programas de exemplo que demonstram o uso dos provedores `fbt` e `pid` do DTrace.

Structs

A palavra-chave de D `struct`, abreviação de *structure* (estrutura) é usada para introduzir um novo tipo composto de um grupo de outros tipos. O novo tipo struct pode ser usado como o tipo de variáveis e matrizes de D, permitindo que você defina grupos de variáveis relacionadas sob um único nome. As structs de D são o mesmo que a construção correspondente em C e C++. Caso você tenha programado na linguagem de programação Java, pense em uma struct de D como uma classe, mas uma classe que tenha apenas membros de dados, não métodos.

Vamos supor que você queira criar um programa de rastreio de chamada do sistema mais sofisticado em D, que registre inúmeras informações sobre cada chamada do sistema a `read(2)` e `write(2)` executada pelo shell, tal como o tempo decorrido, o número de chamadas e a maior contagem de bytes passada como um argumento. Você poderia escrever uma cláusula de D para registrar essas propriedades em três matrizes de associação separadas, como mostrado no exemplo seguinte:

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

```
        arg2 : maxbytes[probefunc];
}
```

Entretanto, esta cláusula é ineficiente porque o DTrace deve criar três matrizes de associação separadas e armazenar cópias separadas dos valores de tupla idênticos correspondentes a `probefunc` para cada uma. Em vez disso, você pode conservar espaço e facilitar a leitura e a manutenção do seu programa usando uma struct. Primeiro, declare um novo tipo de struct no início do arquivo-fonte do programa:

```
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 */
};
```

A palavra-chave `struct` é seguida por um identificador opcional usado para fazer referência ao nosso novo tipo, que agora é conhecido como `struct callinfo`. Os membros de struct são colocados entre um conjunto de chaves `{ }` e a declaração inteira é terminada por um ponto-e-vírgula (`;`). Cada membro de struct é definido através da mesma sintaxe que uma declaração de variável de D, com o tipo do membro listado primeiro, seguido por um identificador que nomeia o membro e outro ponto-e-vírgula (`;`).

A própria declaração struct simplesmente define o novo tipo; ela não cria quaisquer variáveis ou aloca qualquer armazenamento no DTrace. Quando declarada, você pode usar `struct callinfo` como um tipo no restante do seu programa em D, e cada variável de tipo `struct callinfo` armazenará uma cópia das quatro variáveis descritas por nosso modelo de estrutura. Os membros serão organizados na memória na ordem da lista de membros, com espaço de preenchimento introduzido entre membros, conforme necessário, para fins de alinhamento de objeto de dados.

Você pode usar os nomes de identificador de membro para acessar valores de membro individual usando o operador "`.`" escrevendo uma expressão do formato:

nome da variável`.` *nome do membro*

O exemplo seguinte é um programa aprimorado que usa o novo tipo de estrutura. Em um editor, digite o seguinte programa em D e salve-o em um arquivo chamado `rwinfo.d`:

EXEMPLO 7–1 `rwinfo.d`: coletar estatísticas de `read`(2) e `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 */
};
```

EXEMPLO 7–1 `rwinfo.d`: coletar estatísticas de `read`(2) e `write`(2) *(Continuação)*

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

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);
}
```

Depois que você digitar o programa, execute o `dtrace -q -s rwinfo.d`, especificando um dos processos do shell. Em seguida, digite alguns comandos no shell e, quando terminar de inserir os comandos do shell, digite Control-C no terminal do `dtrace` para acionar o teste `END` e imprimir os resultados:

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

Ponteiros para structs

Fazer referência a structs usando ponteiros é muito comum em C e em D. Você pode usar o operador `->` para acessar membros de struct através de um ponteiro. Se uma `struct s` tiver um membro `m` e você possuir um ponteiro para esse struct chamado `sp` (ou seja, `sp` é uma variável do tipo `struct s *`), use o operador * para primeiro cancelar a referência do ponteiro `sp` a fim de acessar o membro:

```
struct s *sp;

(*sp).m
```

ou use o operador `->` como um atalho para essa notação. Os dois fragmentos seguintes de D terão significados equivalentes, se `sp` for um ponteiro para uma struct:

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

O DTrace oferece algumas variáveis incorporadas que são ponteiros para structs, incluindo `curpsinfo` e `curlwpsinfo`. Esses ponteiros se referem respectivamente às structs `psinfo` e `lwpsinfo`, e seu conteúdo fornece um instantâneo de informações sobre o estado do processo atual e o processo leve (LWP) associado ao segmento que acionou o teste atual. Um Solaris LWP é a representação do kernel de um segmento do usuário, sobre o qual os segmentos do Solaris e as interfaces dos segmentos do POSIX são construídos. Por questões de conveniência, o DTrace exporta as informações no mesmo formato que os arquivos do sistema de arquivos `/proc` `/proc/` *pid*`/psinfo` e `/proc/`*pid*`/lwps/` *lwpid*`/lwpsinfo`. As estruturas `/proc` são usadas por ferramentas de observação e de depuração, tais como `ps(1)`, `pgrep(1)` e `truss(1)` e são definidas no arquivo de cabeçalho do sistema de arquivos `<sys/procfs.h>` e são descritas na página do manual `proc(4)`. Aqui estão algumas expressões de exemplo que usam `curpsinfo`, seus tipos e significados:

`curpsinfo->pr_pid`	`pid_t`	ID do processo atual
`curpsinfo->pr_fname`	`char []`	nome do arquivo executável
`curpsinfo->pr_psargs`	`char []`	argumentos iniciais da linha de comando

Você deve revisar a definição de estrutura completa mais tarde, examinando o arquivo de cabeçalho `<sys/procfs.h>` e as descrições correspondentes em `proc(4)`. O próximo exemplo usa o membro `pr_psargs` para identificar um processo do seu interesse, coincidindo os argumentos da linha de comando.

As structs são usadas freqüentemente para criar estruturas de dados complexas de programas em C,sendo assim, a capacidade de descrever e referenciar structs a partir de D também oferece um recurso poderoso de observação do funcionamento interno do kernel do sistema operacional Solaris e suas interfaces de sistema. Além disso, para usar a struct `curpsinfo`,

mencionada anteriormente, o próximo exemplo também examina algumas structs do kernel, observando a relação entre o driver de ksyms(7D) e as solicitações de read(2) O driver usa duas structs comuns, conhecidas como uio(9S) and iovec(9S), para responder a solicitações de leitura a partir do arquivo de dispositivo de caractere /dev/ksyms.

A struct uio, acessada através do nome struct uio ou do alias de tipo uio_t, é descrita na página do manual uio(9S) e é usada para descrever uma solicitação de E/S que envolve a cópia de dados entre o kernel e um processo do usuário. uio, por sua vez, contém uma matriz de uma ou mais estruturas iovec(9S), cada uma descrevendo uma parte da E/S solicitada, caso vários blocos sejam solicitados por meio das chamadas do sistemareadv(2) ou writev(2) Uma das rotinas de interface de driver de dispositivo (DDI) do kernel que opera na struct uio é a função uiomove(9F), que é uma das famílias que os drivers do kernel de funções usam para responder às solicitações read(2) do processo do usuário e copiar os dados de volta para os processos do usuário.

O driver ksyms gerencia um arquivo de dispositivo de caractere chamado /dev/ksyms, que parece ser um arquivo ELF que contém informações sobre a tabela de símbolo do kernel, mas que é, de fato, uma ilusão criada pelo driver usando o conjunto de módulos que estão carregados no momento no kernel. O driver usa a rotina uiomove(9F) para responder às solicitações de read(2). O próximo exemplo ilustra que os argumentos e as chamadas para read(2) de /dev/ksyms correspondem às chamadas do driver para uiomove(9F) a fim de copiar os resultados de volta para o espaço de endereço do usuário no local especificado para read(2).

Podemos usar o utilitário strings(1) com a opção -a para forçar várias leituras de /dev/ksyms . Tente executar strings -a /dev/ksyms em seu shell e veja qual saída é produzida. Em um editor, digite a primeira cláusula do script de exemplo e salve-a em um arquivo chamado ksyms.d:

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

Esta primeira cláusula usa a expressão curpsinfo->pr_psargs para acessar e coincidir os argumentos da linha de comando do nosso comando strings(1), para que o script selecione as solicitações corretas de read(2) antes de rastrear os argumentos. Observe que usando o operador == com um argumento esquerdo que seja uma matriz de char e um argumento direito que seja uma seqüência, o compilador de D deduz que o argumento esquerdo deve ser promovido para uma seqüência e uma comparação de seqüência deve ser realizada. Digite e execute o comando dtrace -q -s ksyms.d em um shell e, em seguida, digite o comando strings -a /dev/ksyms em outro shell. Enquanto strings(1) é executado, você verá a saída do DTrace semelhante ao exemplo seguinte:

```
# 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
#
```

Este exemplo pode ser estendido com uma técnica de programação comum em D para seguir um segmento a partir desta solicitação de read(2) inicial até um local mais profundo no kernel. Na entrada do kernel em `syscall::read:entry`, o próximo script define uma variável de sinalizador local de segmento, indicando que esse segmento é do seu interesse, e limpa esse sinalizador em `syscall::read:return`. Quando é definido, o sinalizador pode ser usado como um predicado em outros testes para instrumentar as funções do kernel, tal como uiomove(9F). O provedor de rastreio de limite de função do DTrace (`fbt`) publica os testes de entrada e retorna para as funções definidas no kernel, incluindo as da DDI. Digite o seguinte código-fonte, que usa o provedor `fbt` para instrumentaruiomove(9F) e salve-o novamente no arquivo `ksyms.d`:

EXEMPLO 7–2 `ksyms.d`: rastreio de relação de `read`(2) e `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);
}
```

A cláusula final do exemplo usa a variável local de segmento `self->watched` para identificar quando um segmento do kernel de seu interesse entra na rotina DDI `uiomove`(9F). Uma vez lá, o script usa a matriz incorporada `args` para acessar o quarto argumento (`args[3]`) para `uiomove()`, que é um ponteiro para a `struct uio` que representa a solicitação. O compilador de D associa automaticamente cada membro da matriz `args` com o tipo correspondente ao protótipo da função de C da rotina do kernel instrumentado. O membro `uio_iov` contém um ponteiro para `struct iovec` da solicitação. Uma cópia deste ponteiro é salva para que seja usada na nossa cláusula na variável local de cláusula `this->iov`. Na declaração final, o script cancela a referência de `this->iov` para acessar os membros `iovec`, `iov_len` e `iov_base`, que representam o tamanho em bytes e o endereço base de destino de `uiomove`(9F), respectivamente. Esses valores devem corresponder aos parâmetros de entrada para a chamada do sistema de `read`(2) emitida no driver. Vá para o shell, execute `dtrace -q -s ksyms.d` e insira novamente o comando `strings -a /dev/ksyms` em outro shell. Você deve ver uma saída semelhante ao exemplo seguinte:

```
# 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
#
```

Os endereços e os IDs de processo serão diferentes na sua saída, mas você deve observar que os argumentos de entrada para `read`(2) correspondem aos parâmetros passados para `uiomove`(9F) pelo driver `ksyms`.

Uniões

As uniões são outra espécie de tipo composto aceito por ANSI-C e D, e estão fortemente relacionadas às structs. Uma união é um tipo composto onde um conjunto de membros de tipos diferentes são definidos e todos os objetos de membro ocupam a mesma região de armazenamento. Uma união é, portanto, um objeto de tipo variado, onde somente um membro é válido a qualquer momento determinado, dependendo de como a união foi atribuída. Geralmente, alguma outra variável ou parte de estado é usada para indicar que o membro da união é válido no momento. O tamanho de uma união é o tamanho do maior membro, e o alinhamento de memória usado para a união é o alinhamento máximo necessário para os membros da união.

A estrutura `kstat` do Solaris define uma struct contendo uma união que é usada no exemplo seguinte para ilustrar e observar as uniões de C e de D. A estrutura `kstat` é usada para exportar

um conjunto de contadores nomeados representando as estatísticas do kernel, tal como o uso da memória e o throughput de E/S. A estrutura é usada para implementar utilitários como mpstat(1M) e iostat(1M) Esta estrutura usa struct kstat_named para representar um contador nomeado e seu valor, e é definido da seguinte forma:

```
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 */
};
```

A declaração de inspeção é abreviada para fins ilustrativos. A definição de estrutura completa pode ser encontrada no arquivo de cabeçalho <sys/kstat.h> e é descrita em kstat_named(9S). A declaração acima é válida em ANSI-C e em D, e define uma struct contendo, como um de seus membros, um valor de união com membros de vários tipos, dependendo do tipo do contador. Observe que, já que a própria união é declarada dentro de outro tipo, struct kstat_named, um nome formal do tipo de união é omitido. Este estilo de declaração é conhecido como *união anônima*. O membro nomeado valor é de um tipo de união descrito pela declaração precedente, mas esse tipo de união não tem nome porque ele não precisa ser usado em nenhum outro local. É atribuído ao membro struct data_type um valor que indica qual membro da união é válido para cada objeto do tipo struct kstat_named. Um conjunto de símbolos de pré-processadores de C são definidos para os valores de data_type. Por exemplo, o símbolo KSTAT_DATA_CHAR é igual a zero e indica que o membro value.c se encontra onde o valor está armazenado no momento.

O Exemplo 7–3 demonstra o acesso à união kstat_named.value rastreando um processo do usuário. Os contadores kstat podem ser exemplificados a partir de um processo do usuário por meio da função kstat_data_lookup(3KSTAT), que retorna um ponteiro para struct kstat_named. O utilitário mpstat(1M) chama esta função repetidamente enquanto é executado, a fim de exemplificar os valores de contador mais recentes. Vá para o seu shell e tente executar mpstat 1 e observe a saída. Pressione Control-C em seu shell para anular mpstat após alguns segundos. Para observar a amostra do contador, gostaríamos de ativar um teste que seja acionado cada vez que o comando mpstat chamar a função kstat_data_lookup(3KSTAT) em libkstat. Para fazê-lo, usaremos um novo provedor do DTrace: pid. O provedor pid permite que você crie os testes dinamicamente nos processos do usuário em locais de símbolo de C, tais como pontos de entrada de função. Você pode pedir ao provedor pid para criar um teste em uma entrada de função do usuário e retornar os sites, escrevendo as descrições do teste no formato:

pid*ID do processo*:*nome do objeto*:*nome da função*:entry pid*ID do processo*:*nome do objeto*:*nome da funç

Por exemplo, se você quisesse criar um teste no ID do processo 3 que é acionado na entrada para `kstat_data_lookup(3KSTAT)`, você escreveria a seguinte descrição de teste:

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

O provedor `pid` insere a instrumentação dinâmica no processo do usuário especificado no local do programa correspondente à descrição do teste. A implementação do teste força cada segmento do usuário que alcança o local do programa instrumentado a entrar no kernel do sistema operacional e no DTrace, acionando o teste correspondente. Sendo assim, embora o local da instrumentação esteja associado a um processo do usuário, os predicados e as ações do DTrace que você especificar ainda serão executados no contexto do kernel do sistema operacional. O provedor `pid` é descrito em maiores detalhes no Capítulo 30, “Provedor `pid`”.

Em vez de ter que editar o fonte do programa em D toda vez que desejar aplicar seu programa a um processo diferente, você pode inserir identificadores chamados *variáveis de macro* em seu programa que são avaliados no momento em que o programa é compilado e substituído pelos argumentos da linha de comando adicionais do `dtrace`. As variáveis de macro são especificadas por meio do sinal de dólar `$` seguido por um identificador ou dígito. Se você executar o comando `dtrace -s` *script* `foo bar baz`, o compilador de D definirá automaticamente as variáveis de macro `$1`, `$2` e `$3` aos símbolos `foo`, `bar` e `baz`, respectivamente. Você pode usar as variáveis de macro nas expressões de programa em D ou em descrições de teste. Por exemplo, as descrições de teste seguintes instrumentam qualquer ID de processo que seja especificado como um argumento adicional para `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;
}
```

As variáveis de macro e os scripts reutilizáveis são descritos em maiores detalhes no Capítulo 15, “Script”. Agora que sabemos como instrumentar os processos do usuário usando o ID de processo deles, vamos retornar às uniões de amostra. Vá para o editor e digite o código-fonte do nosso exemplo completo e salve-o em um arquivo chamado `kstat.d`:

EXEMPLO 7–3 kstat.d: rastrear chamadas para 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;
}
```

Agora vá para um dos seus shells e execute o comando `mpstat 1` para iniciar o mpstat(1M) sendo executado em um modo em que ele faz amostra de estatísticas e informa-as uma por segundo. Quando o `mpstat` estiver em execução, execute o comando `dtrace -q -s kstat.d ‘pgrep mpstat‘` em seu outro shell. Você verá a saída correspondente às estatísticas que estão sendo acessadas. Pressione Control-C para anular `dtrace` e retornar para o prompt do 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
#
```

Se capturar a saída em cada janela do terminal e subtrair cada valor do valor informado pela iteração anterior através das estatísticas, você deve ser capaz de correlacionar a saída do `dtrace` com a saída do `mpstat`. O programa de exemplo registra o ponteiro do nome do contador na entrada para a função de consulta e, em seguida, realiza a maior parte do trabalho de rastreio no retorno de `kstat_data_lookup`(3KSTAT). As funções incorporadas de D `copyinstr()` e `copyin()` copiam os resultados da função do processo do usuário para o DTrace quando `arg1` (o valor de retorno) não é `NULL`. Quando os dados de `kstat` tiverem sido copiados, o exemplo informará o valor do contador `ui64` a partir da união. Este exemplo simplificado assume que `mpstat` faz amostras de contadores que usam o membro `value.ui64`. Como um exercício, tente registrar `kstat.d` para usar vários predicados e imprimir a união correspondente ao membro `data_type`. Você também pode tentar criar uma versão de `kstat.d` que calcule a diferença entre valores de dados sucessivos e produza realmente uma saída semelhante a `mpstat`.

Deslocamentos e tamanhos de membro

Você pode determinar o tamanho em bytes de qualquer tipo ou expressão de D, incluindo uma struct ou união, usando o operador `sizeof`. O operador `sizeof` pode ser aplicado a uma expressão ou ao nome de um tipo entre parênteses, como ilustrado pelos dois exemplos seguintes:

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

Por exemplo, a expressão `sizeof (uint64_t)` retornaria o valor 8, e a expressão `sizeof (callinfo.ts)` retornaria 8, se inserida no código-fonte do nosso programa de exemplo acima. O tipo de retorno formal do operador `sizeof` é o alias de tipo `size_t`, que é definido como um inteiro não-assinado do mesmo tamanho que um ponteiro no modelo de dados atual, e é usado para representar as contagens de byte. Quando o operador `sizeof` é aplicado a uma expressão, a expressão é validada pelo compilador de D, mas o tamanho de objeto resultante é calculado no momento da compilação e nenhum código da expressão é gerado. Você pode usar `sizeof` em qualquer lugar que uma constante de inteiro é necessária.

Você pode usar o operador acompanhante `offsetof` para determinar o desvio em bytes de um membro de struct ou de união a partir do início do armazenamento associado a qualquer objeto do tipo de união ou de struct. O operador `offsetof` é usado em uma expressão do seguinte formato:

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

Aqui *nome do tipo* é o nome de qualquer tipo de struct ou de união ou alias de tipo, e *nome do membro* é o identificador que nomeia um membro dessa struct ou união. Semelhante a `sizeof`, `offsetof` retorna `size_t` e pode ser usado em qualquer lugar em um programa em D onde uma constante de inteiro possa ser usada.

Campos de bit

D também permite a definição de membros de struct de inteiro e de união com números de bits arbitrários, conhecidos como *campos de bit*. Um campo de bit é declarado pela especificação de um tipo base de inteiro assinado ou não assinado, um nome de membro e um sufixo indicando o número de bits a ser atribuído ao campo, como mostrado no exemplo seguinte:

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

A largura do campo de bit é uma constante de inteiro separada do nome do membro por uma vírgula à direita. A largura do campo de bit deve ser positiva e deve ter um número de bits que não seja maior que a largura do tipo base de inteiro correspondente. Os campos de bit maiores que 64 bits não podem ser declarados em D. Os campos de bits de D fornecem compatibilidade e acesso ao recurso ANSI-C correspondente. Os campos de bit são geralmente usados em situações em que o armazenamento de memória seja pelo menos premium ou quando um layout de struct deve corresponder ao layout de registro de hardware.

Um campo de bits é uma construção de compilador que automatiza o layout de um inteiro e um conjunto de máscaras para extrair os valores de membro. O mesmo resultado pode ser atingido pela simples definição de máscaras e pelo uso do operador &. Os compiladores de C e de D tentam empacotar bits, da forma mais eficiente possível, mas eles estão livres para fazê-lo em qualquer ordem ou de qualquer maneira, sendo assim, não há garantias de que os campos de bit produzam layouts de bit idênticos entre compiladores ou arquiteturas diferentes. Se você precisa de layout de bit estável, construa as máscaras de bits e extraia os valores usando o operador &.

Um membro de campo de bit é acessado pela simples especificação de seu nome em combinação com os operadores " . " ou ->, como qualquer outra struct ou membro de união. O campo de bit é automaticamente promovido para o próximo tipo de inteiro maior a fim de ser usado em quaisquer expressões. Como o armazenamento de campo de bit pode não estar alinhado em um limite de bytes ou ser um número de bytes arredondado em tamanho, você não pode aplicar os operadores `sizeof` ou `offsetof` a um membro de campo de bits. O compilador de D também proíbe você de pegar o endereço de um membro de campo de bit usando o operador &.

◆ ◆ ◆ CAPÍTULO 8

Definições de tipo e de constante

Este capítulo descreve como declarar alias de tipo e constantes nomeadas em D. Este capítulo também discute o gerenciamento do espaço de nome e do tipo de D para identificadores e tipos de sistemas operacionais e programas.

Typedef

A palavra-chave `typedef` é usada para declarar um identificador como um alias de um tipo existente. Como todas as declarações de tipo de D, a palavra-chave `typedef` é usada fora das cláusulas do teste em uma declaração no formato:

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

onde *tipo existente* é uma declaração de qualquer tipo e *tipo novo* é um identificador a ser usado como alias desse tipo. Por exemplo, a declaração:

```
typedef unsigned char uint8_t;
```

é usada internamente pelo compilador de D para criar o alias do tipo `uint8_t` . Os alias de tipo podem ser usados em qualquer lugar que um tipo normal possa ser usado, como o tipo de uma variável ou valor de matriz de associação ou membro de tupla. Você também pode combinar `typedef` com declarações mais elaboradas como a definição de uma nova `struct`:

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

Neste exemplo, `struct foo` é definida como o mesmo tipo que o seu alias, `foo_t`. Os cabeçalhos do sistema C do Solaris geralmente usam o sufixo `_t` para indicar um alias de `typedef`.

Enumerações

Definir nomes simbólicos para constantes em um programa facilita a legibilidade e simplifica o processo de manutenção do programa no futuro. Um método é definir uma *enumeração* que associa um conjunto de inteiros a um conjunto de identificadores chamados enumeradores, que o compilador reconhece e substitui pelo valor de inteiro correspondente. Uma enumeração é definida usando-se uma declaração como:

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

O primeiro enumerador na enumeração, `RED`, recebe o valor zero e cada identificador subseqüente recebe o próximo valor de inteiro. Você também pode especificar um valor de inteiro explícito para qualquer enumerador usando como sufixo um sinal de igual e uma constante de inteiro, como no exemplo a seguir:

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

O compilador atribui o valor 10 ao enumerador `BLUE` porque ele não tem valor especificado e o enumerador anterior está definido como 9. Quando a enumeração é definida, os enumeradores podem ser usados em qualquer lugar de um programa em D em que uma constante de inteiro possa ser usada. Além disso, a enumeração `enum colors` também é definida como um tipo que seja equivalente a um `int`. O compilador de D permitirá que uma variável do tipo `enum` seja usada em qualquer lugar em que um `int` possa ser usado, e permitirá que qualquer valor de inteiro seja atribuído a uma variável do tipo `enum`. Você também pode omitir o nome da `enum` na declaração se o nome do tipo não for necessário.

Os enumeradores são visíveis em todas as cláusulas e declarações subseqüentes no seu programa, sendo assim, você não pode definir o mesmo identificador de enumerador em mais de uma enumeração. Entretanto, você pode definir mais de um enumerador que tenha o mesmo valor na mesma enumeração ou em enumerações diferentes. Você também pode atribuir inteiros que não possuam um enumerador correspondente a uma variável do tipo da enumeração.

A sintaxe da enumeração de D é a mesma que a sintaxe correspondente em ANSI-C. D também fornece acesso a enumerações definidas no kernel do sistema operacional e seus módulos carregáveis, mas esses enumeradores não são globalmente visíveis no programa em D. Os enumeradores do kernel só ficam visíveis quando usados como argumento para um dos

operadores de comparação binários, quando comparados a um objeto do tipo de enumeração correspondente. Por exemplo, a função uiomove(9F) possui um parâmetro do tipo enum uio_rw definido da seguinte maneira:

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

Os enumeradores UIO_READ e UIO_WRITE normalmente não são visíveis no seu programa em D, mas você pode promover sua visibilidade global comparando um deles a um valor do tipo enum uio_rw, conforme mostrado na cláusula do seguinte exemplo:

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

Este exemplo rastreia chamadas à função uiomove(9F) para solicitações de gravação, comparando args[2], uma variável do tipo enum uio_rw, ao enumerador UIO_WRITE. Como o argumento do lado esquerdo é um tipo de enumeração, o compilador de D pesquisa a enumeração ao tentar resolver o identificador do lado direito. Esse recurso protege seus programas em D contra conflitos de nome de identificador acidentais com a grande coleção de enumerações definidas no kernel do sistema operacional.

In-lines

As constantes nomeadas de D também podem ser definidas usando-se diretivas in-line, que fornecem um meio mais geral de criar identificadores que são substituídos por expressões ou valores predefinidos durante a compilação. As diretivas embutidas são uma forma de substituição lexical mais eficiente que a diretiva #define fornecida pelo pré-processador de C porque um tipo real é atribuído à substituição e ela é realizada usando-se a árvore de sintaxe compilada e não simplesmente um conjunto de símbolos lexicais. Uma diretiva in-line é especificada usando-se uma declaração da seguinte forma:

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

onde *tipo* é uma declaração de tipo de um tipo existente, *nome* é qualquer identificador válido de D que não tenha sido definido anteriormente como uma variável in-line ou global e *expressão* é qualquer expressão válida de D. Quando a diretiva in-line é processada, o compilador de D substitui a forma compilada da *expressão* de cada instância subseqüente de *nome* na origem do programa. Por exemplo, o programa em D a seguir faria o rastreio da seqüência "olá" e do valor de inteiro 123:

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

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

Um nome in-line pode ser usado em qualquer lugar que uma variável global do tipo correspondente pode ser usada. Se a expressão in-line puder ter como valor uma constante de seqüências ou de inteiro na compilação, então o nome in-line também poderá ser usado em contextos que requeiram expressões de constante, como dimensões de matriz escalar.

A expressão in-line é validada quanto a erros de sintaxe como parte da avaliação da diretiva. O tipo de resultado da expressão deve ser compatível com o tipo definido pela diretiva in-line, de acordo com as mesmas regras usadas para o operador de atribuição de D (=). Uma expressão in-line não pode fazer referência ao próprio identificador in-line: definições recursivas não são permitidas.

Os pacotes de software do DTrace instalam inúmeros arquivos de origem de D no diretório do sistema `/usr/lib/dtrace` que contém diretivas in-line que você pode usar em seus programas em D. Por exemplo, a biblioteca `signal.d` inclui diretivas no formato:

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

Estas definições in-line fornecem acesso ao conjunto atual de nomes de sinal do Solaris descritos em `signal(3HEAD)`. Semelhantemente, a biblioteca `errno.d` contém diretivas in-line das constantes `errno` de C descritas em `Intro(2)`.

Por padrão, o compilador de D inclui automaticamente todos os arquivos de biblioteca de D fornecidos para que você possa usar estas definições em qualquer programa em D.

Espaços de nome de tipo

Esta seção discute os espaços de nome de D e as questões de espaço de nome relacionadas a tipos. Em linguagens tradicionais como ANSI-C, a visibilidade do tipo é determinada pelo fato de um tipo estar ou não aninhado em uma função ou outra declaração. Os tipos declarados no escopo externo de um programa em C são associados a um espaço de nome global único e são visíveis em todo o programa. Os tipos definidos nos arquivos de cabeçalho de C são geralmente incluídos nesse escopo externo. Ao contrário dessas linguagens, D fornece acesso a tipos de vários escopos externos.

A linguagem D facilita a observação dinâmica em várias camadas de uma pilha de software, incluindo o kernel do sistema operacional, um conjunto associado de módulos carregáveis do kernel e processos do usuário em execução no sistema. Um único programa em D pode

instanciar testes para coletar dados de vários módulos do kernel ou outras entidades de software compiladas em objetos binários independentes. Portanto, pode haver mais de um tipo de dados do mesmo nome, talvez com definições diferentes, no universo de tipos disponíveis para o DTrace e o compilador de D. Para gerenciar essa situação, o compilador de D associa cada tipo a um espaço de nome identificado pelo objeto do programa que o contém. Tipos de um objeto de programa em particular podem ser acessados especificando-se o nome do objeto e o operador de escopo de aspa invertida (‘) em qualquer nome de tipo.

Por exemplo, se um módulo do kernel denominado `foo` contiver a seguinte declaração do tipo C:

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

então os tipos `struct bar` e `bar_t` poderiam ser acessados de D usando-se os nomes de tipo:

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

O operador de aspa invertida pode ser usado em qualquer contexto em que um nome de tipo seja apropriado, incluindo a especificação do tipo de declarações de variável de D ou expressões de difusão em cláusulas de teste de D.

O compilador de D também fornece dois espaços de nome de tipo incorporados especiais que usam os nomes `C` e `D` respectivamente. O espaço de nome de tipo de C é inicialmente preenchido com os tipos intrínsecos padrão de ANSI-C como `int`. Além disso, as definições de tipo obtidas usando-se o pré-processador de C cpp(1) e a opção `dtrace -C` serão processadas pelo escopo de C e adicionadas nele. Como resultado, você pode incluir arquivos de cabeçalho de C, contendo declarações de tipo que já estão visíveis, em outro espaço de nome de tipo sem causar um erro de compilação.

O espaço de nome do tipo de D é preenchido inicialmente com elementos intrínsecos do tipo de D como `int` e `string`, assim como os alias de tipo de D incorporados como `uint32_t`. Todas as novas declarações de tipo que aparecerem na origem do programa em D serão automaticamente adicionadas ao espaço de nome de tipo D. Se você criar um tipo complexo como uma `struct` no seu programa em D, consistindo em tipos de membro de outros espaços de nome, os tipos de membro serão copiados para o espaço de nome de D pela declaração.

Quando o compilador de D encontra uma declaração de tipo que não especifica um espaço de nome explícito usando o operador de aspa invertida, o compilador pesquisa o conjunto de espaços de nome de tipo ativo para localizar uma correspondência usando o nome de tipo especificado. O espaço de nome de C sempre é pesquisado primeiro, seguido pelo espaço de nome de D. Se o nome do tipo não for encontrado no espaço de nome de C ou D, os espaços de nome de tipo dos módulos ativos do kernel serão pesquisados em ordem crescente por ID do módulo do kernel. Essa ordenação garante que os objetos binários que formam o kernel do núcleo sejam pesquisados antes de quaisquer módulos carregáveis do kernel, mas não garante

quaisquer propriedades de ordenação entre os módulos carregáveis. Você deve usar o operador de escopo ao acessar tipos definidos em módulos carregáveis do kernel para evitar conflitos de nome de tipo com outros módulos do kernel.

O compilador de D usa informações de depuração compactadas de ANSI-C fornecidas com os módulos do kernel do núcleo do Solaris para acessar automaticamente os tipos associados ao código-fonte do sistema operacional sem a necessidade de acessar os arquivos de inclusão correspondentes de C. Essas informações de depuração simbólicas podem não estar disponíveis para todos os módulos do kernel no seu sistema. O compilador de D irá reportar um erro se você tentar acessar um tipo dentro do espaço de nome de um módulo que não possua as informações de depuração compactadas de C destinadas para uso com o DTrace.

◆ ◆ ◆ CAPÍTULO 9

9 Agregações

Ao instrumentar o sistema para responder a questões relacionadas ao desempenho, é útil considerar como os dados podem ser agregados para responder a uma questão específica em vez de pensar em termos de dados coletados por testes individuais. Por exemplo, se quisesse saber o número de chamadas do sistema por ID de usuário, você não se preocuparia necessariamente com os dados coletados em *cada* chamada do sistema. Você simplesmente deseja ver uma tabela de IDs de usuário e chamadas do sistema. Historicamente, você responderia a essa questão coletando dados em cada chamada do sistema e pós-processando os dados usando uma ferramenta como a `awk(1)` ou `perl(1)`. Entretanto, no DTrace a agregação de dados é uma operação de primeira classe. Este capítulo descreve os recursos do DTrace para manipular *agregações*.

Funções de agregação

Uma *função de agregação* tem a seguinte propriedade:

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

onde x_n é um conjunto de dados arbitrários. Ou seja, ao aplicar uma função de agregação a subconjuntos do todo e depois aplicá-la de novo aos resultados, o resultado é o mesmo que aplicá-la ao próprio todo. Por exemplo, considere uma função `SUM` que retorna a soma de um determinado conjunto de dados. Se os dados não processados consistirem em {2, 1, 2, 5, 4, 3, 6, 4, 2}, o resultado da aplicação de `SUM` a todo o conjunto será {29}. De forma similar, o resultado da aplicação de `SUM` ao subconjunto consistindo nos três primeiros elementos é {5}, o resultado da aplicação de `SUM` ao conjunto consistindo nos três elementos subseqüentes é {12}, e o resultado da aplicação de `SUM` aos três elementos restantes também é {12}. `SUM` é uma função de agregação porque quando aplicada ao conjunto desses resultados, {5, 12, 12}, é produzido o mesmo resultado, {29}, que ao aplicar `SUM` aos dados originais.

Nem todas as funções são de agregação. Um exemplo de uma função que não é de agregação é a `MEDIAN` que determina o elemento mediano do conjunto. (O elemento mediano de um conjunto é aquele para o qual existem tantos elementos maiores que ele quanto menores). `MEDIAN` é

derivada classificando-se o conjunto e selecionando-se o elemento do meio. Retornando aos dados não processados originais, se `MEDIAN` for aplicada ao conjunto consistindo nos três primeiros elementos, o resultado será {2}. (O conjunto classificado é {1, 2, 2}; {2} é o conjunto que consiste no elemento do meio). Da mesma forma, quando `MEDIAN` é aplicada aos próximos três elementos, o resultado é {4} e quando `MEDIAN` é aplicada aos três elementos finais, o resultado é {4}. Quando `MEDIAN` é aplicada a cada um dos subconjuntos, o resultado é o conjunto {2, 4, 4}. Se `MEDIAN` for aplicada a esse conjunto, o resultado será {4}. Entretanto, a classificação do conjunto original resulta em {1, 2, 2, 2, 3, 4, 4, 5, 6}. Se `MEDIAN` for aplicada a esse conjunto, o resultado será {3}. Como esses resultados não coincidem, `MEDIAN` não é uma função de agregação.

Muitas funções comuns para compreensão de um conjunto de dados são funções de agregação. Essas funções incluem a contagem do número de elementos do conjunto, o cálculo do valor mínimo do conjunto, o cálculo do valor máximo do conjunto e a soma de todos os elementos no conjunto. A determinação da média aritmética do conjunto pode ser construída a partir da função para contar o número de elementos do conjunto e a função para somar o número de elementos do conjunto.

Entretanto, muitas funções úteis não são funções de agregação. Essas funções incluem o cálculo do modo (o elemento mais comum) de um conjunto, o valor mediano do conjunto ou o desvio padrão do conjunto.

A aplicação de funções de agregação aos dados quando são rastreados possui algumas vantagens:

- O conjunto inteiro de dados não precisa ser armazenado. Sempre que um novo elemento for adicionado ao conjunto, a função de agregação será calculada de acordo com o conjunto que consiste no resultado intermediário atual e no novo elemento. Depois que o novo resultado for calculado, o novo elemento poderá ser descartado. Esse processo reduz a quantidade de armazenamento exigido por um fator do número de pontos de dados, que geralmente é muito grande.
- A coleção de dados não induz a problemas de escalabilidade patológicos. As funções de agregação permitem que os resultados intermediários sejam mantidos *por CPU* em vez de em uma estrutura de dados compartilhada. O DTrace depois aplica a função de agregação ao conjunto que consiste nos resultados intermediários por CPU para produzir o resultado final de todo o sistema.

Agregações

O DTrace armazena os resultados das funções de agregação em objetos chamados *agregações*. Os resultados da agregação são indexados usando-se uma tupla de expressões similar às usadas nas matrizes de associação. Em D, a sintaxe de uma agregação é:

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

onde *nome* é o nome da agregação, *chaves* é uma lista separada por vírgulas de expressões de D, *funçagr* é uma das funções de agregação do DTrace e *args* é uma lista de argumentos separados por vírgulas apropriada para a função de agregação. O *nome* da agregação é um identificador de D que possui como prefixo o caractere especial `@`. Todas as agregações nomeadas nos programas em D são variáveis globais; não há agregações locais de cláusula ou segmento. Os nomes de agregação são mantidos em um espaço de nome de identificador separado de outras variáveis globais de D. Lembre-se que `a` e `@a` não são a mesma variável se você reutilizar nomes. O nome de agregação especial `@` pode ser usado para nomear uma agregação anônima em programas em D simples. O compilador de D trata esse nome como um alias do nome de agregação `@_`.

As funções de agregação de DTrace são mostradas na tabela a seguir. A maioria das funções de agregação utilizam apenas um único argumento que representa os novos dados.

TABELA 9–1 Funções de agregação do DTrace

Nome da função	Argumentos	Resultado
`count`	nenhum	O número de vezes chamada.
`sum`	expressão escalar	O valor total das expressões especificadas.
`avg`	expressão escalar	A média aritmética das expressões especificadas.
`min`	expressão escalar	O menor valor entre as expressões especificadas.
`max`	expressão escalar	O maior valor entre as expressões especificadas.
`lquantize`	expressão escalar, limite inferior, limite superior, valor da etapa	Uma distribuição de freqüência linear, dimensionada pelo intervalo especificado, dos valores das expressões especificadas. Incrementa o valor no depósito *mais alto* que é *menor* que a expressão especificada.
`quantize`	expressão escalar	Uma distribuição de freqüência em potência de dois dos valores das expressões especificadas. Incrementa o valor no depósito *mais alto* em potência de dois que é *menor* que a expressão especificada.

Por exemplo, para contar o número de chamadas do sistema `write(2)` no sistema, você poderia usar uma seqüência informativa como uma chave e a função de agregação `count()`:

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

O comando `dtrace` imprime resultados de agregação por padrão quando o processo é encerrado, seja como o resultado de uma ação `END` explícita ou quando o usuário pressiona Control-C. O seguinte exemplo mostra o resultado da execução desse comando, aguardando-se alguns segundos e pressionando-se Control-C:

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

  write system calls                                              179
#
```

Você pode contar chamadas do sistema por nome de processo usando a variável `execname` como chave de uma agregação:

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

O seguinte exemplo mostra o resultado da execução deste comando, aguardando-se alguns segundos e pressionando-se 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
#
```

Como alternativa, você pode querer examinar em detalhes as gravações organizadas pelo nome do executável e o descritor de arquivos. O descritor de arquivos é o primeiro argumento de `write(2)`, sendo assim, o seguinte exemplo usa uma chave que consiste em `execname` e `arg0`:

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

A execução deste comando resulta em uma tabela com o nome do executável e o descritor de arquivos, conforme mostrado no exemplo a seguir:

```
# 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
#
```

O exemplo a seguir exibe o tempo médio gasto na chamada do sistema write, organizado por nome de processo. Este exemplo usa a função de agregação `avg()`, especificando a expressão para realizar média como o argumento. O exemplo faz a média do tempo gasto na chamada do sistema:

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

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

O seguinte exemplo mostra o resultado da execução deste comando, aguardando-se alguns segundos e pressionando-se 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
#
```

A média pode ser útil, mas geralmente não fornece detalhes suficientes para compreender a distribuição dos pontos de dados. Para compreender a distribuição com mais detalhes, use a função de agregação `quantize()` conforme mostrado no exemplo a seguir:

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

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

Como cada linha de resultado torna-se um diagrama de distribuição de freqüência, o resultado deste script é substancialmente maior que os anteriores. O exemplo a seguir mostra uma seleção de resultado de amostra:

```
  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
```

Observe que as linhas da distribuição de freqüência são *sempre* valores de potência de dois. Cada linha indica a contagem do número de elementos *maiores que ou iguais* ao valor correspondente, mas *menores que* o próximo valor maior de linha. Por exemplo, o resultado acima mostra que `iropt` tinha 4.149 gravações ocorrendo entre 8.192 nanossegundos e 16.383 nanossegundos, inclusive.

Embora `quantize()` seja útil para obter uma rápida visualização dos dados, talvez você queira examinar uma distribuição em valores lineares. Para exibir uma distribuição de valor linear, use a função de agregação `lquantize ()`. A função `lquantize()` possui três argumentos além de uma expressão de D. um limite inferior, um limite superior e uma etapa. Por exemplo, se você quisesse visualizar a distribuição de gravações por descritor de arquivo, uma quantização em potência de dois não seria eficiente. Em vez disso, use uma quantização linear com um pequeno intervalo, conforme mostrado no exemplo a seguir:

```
syscall::write:entry
{
    @fds[execname] = lquantize(arg0, 0, 100, 1);
}
```

A execução deste script por alguns segundos resulta em uma grande quantidade de informação. O exemplo a seguir mostra uma seleção de resultado típico:

```
  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
```

Você também pode usar a função de agregação `lquantize()` para agregar no momento a partir de um ponto no passado. Essa técnica permite que você observe uma alteração no comportamento ao longo do tempo. O exemplo a seguir exibe a alteração no comportamento da chamada do sistema durante um processo executando o comando `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;
}
```

O script anterior fornece uma melhor avaliação do comportamento da chamada do sistema quando muitos processos `date(1)` são executados. Para ver esse resultado, execute `sh -c 'while true; do date >/dev/null; done'` em uma janela, enquanto executa o script de D em outra. O script produz um perfil do comportamento da chamada do sistema do comando `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
```

Este resultado fornece uma breve idéia das diferentes fases do comando `date(1)`com relação aos serviços solicitados do kernel. Para entender melhor essas fases, talvez você queira compreender quando quais chamadas do sistema estão sendo chamadas. Se esse for o caso, você poderá alterar o script de D para agregar na variável `probefunc` em vez de uma seqüência de constante.

Imprimindo agregações

Por padrão, várias agregações são exibidas na ordem em que são introduzidas no programa em D. Você pode ignorar esse comportamento usando a função `printa()` para imprimir as agregações. A função `printa ()` também permite que você formate precisamente os dados da agregação usando uma seqüência de formato, conforme descrito no Capítulo 12, “Formatação de saída”.

Se uma agregação não for formatada com uma instrução `printa()` no seu programa em D, o comando `dtrace` obterá um instantâneo dos dados da agregação e imprimirá os resultados uma vez, depois que o rastreio tiver sido concluído com o formato de agregação padrão. Se uma determinada agregação for formatada com uma instrução `printa()`, o comportamento padrão será desativado. Você pode alcançar resultados equivalentes adicionando a instrução `printa(@`*nome da agregação*) a uma cláusula do teste `dtrace:::END` no programa. O formato de resultado padrão das funções de agregação `avg()`, `count()`, `min()`, `max()` e `sum()` exibe um valor decimal inteiro correspondente ao valor agregado de cada tupla. O formato de resultado padrão das funções de agregação `lquantize()` e `quantize()` exibe uma tabela ASCII dos resultados. As tuplas de agregação são impressas como se `trace()` tivesse sido aplicada a cada elemento da tupla.

Normalização de dados

Ao agregar dados durante algum período de tempo, talvez você queira *normalizar* os dados em relação a algum fator de constante. Essa técnica permite que você compare dados separados com mais facilidade. Por exemplo, ao agregar chamadas do sistema, talvez você queira resultados de chamadas do sistema como uma taxa por segundo em vez de um valor absoluto durante o curso da execução. A ação `normalize()` do DTrace permite que você normalize os dados dessa forma. Os parâmetros de `normalize()` são uma agregação e um fator de normalização. O resultado da agregação mostra cada valor dividido pelo fator de normalização.

O exemplo a seguir mostra como agregar dados por chamada do sistema:

```
#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);
}
```

A execução do script acima por um breve período de tempo gera o seguinte resultado em um computador desktop:

```
# 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
```

`normalize()` define o fator de normalização da agregação especificada, mas essa ação não modifica os dados subjacentes. A função `denormalize()` possui somente uma agregação. Ao adicionar a ação denormalize ao exemplo anterior, são retornadas contagens de chamada do sistema sem processamento e taxas por segundo:

```
#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);
}
```

A execução do script acima por um breve período de tempo produz um resultado similar ao seguinte exemplo:

```
# 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
```

As agregações também podem ser renormalizadas. Se `normalize()` for chamada mais de uma vez para a mesma agregação, o fator de normalização será o fator especificado na chamada mais recente. O exemplo a seguir imprime taxas por segundo ao longo do tempo:

EXEMPLO 9–1 `renormalize.d`: Renormalizando uma agregação

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
}

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

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

Cancelando agregações

Ao usar o DTrace para criar scripts de monitoração simples, você pode cancelar periodicamente os valores em uma agregação usando a função `clear()`. Essa função possui uma agregação como seu único parâmetro. A função `clear()` cancela somente os *valores* da agregação; as chaves da agregação são mantidas. Assim, a presença de uma chave em uma agregação que possui um valor associado de zero indica que a chave *tinha* um valor diferente de zero que foi

subseqüentemente definido como zero como parte de `clear()`. Para descartar os valores de uma agregação e suas chaves, use `trunc()`. Consulte "Truncando agregações" na página 129 para obter detalhes.

O exemplo a seguir adiciona `clear()` ao Exemplo 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;
}
```

Enquanto o Exemplo 9–1 mostra a taxa de chamada do sistema durante a invocação de `dtrace`, o exemplo anterior mostra a taxa da chamada do sistema somente para o período de dez segundos mais recente.

Truncando agregações

Ao olhar os resultados da agregação, você geralmente só se preocupa com os vários resultados superiores. As chaves e os valores associados a qualquer coisa além dos valores mais altos não são interessantes. Talvez você também queira descartar todo o resultado de uma agregação, removendo as chaves *e* os valores. A função `trunc()` do DTrace é usada para ambas as situações.

Os parâmetros de `trunc()` são uma agregação e um valor opcional de truncamento. Sem o valor de truncamento, `trunc ()` descarta *ambos* os valores de agregação *e* as chaves de agregação de toda a agregação. Quando um valor de truncamento *n* está presente, `trunc()` descarta valores de agregação e chaves *exceto* para os valores e chaves associados aos valores de *n* mais altos. Ou seja, `trunc(@foo, 10)` trunca a agregação chamada `foo` após os dez valores principais, enquanto `trunc(@foo)` descarta toda a agregação. Toda a agregação também será descartada se `0` for especificado como o valor de truncamento.

Para ver os valores inferiores de *n* em vez dos superiores de *n*, especifique um valor de truncamento negativo para `trunc()`. Por exemplo, `trunc(@foo, -10)` trunca a agregação denominada `foo` depois dos dez valores inferiores.

O exemplo a seguir aumenta o exemplo da chamada do sistema para exibir somente as taxas de chamada do sistema por segundo dos dez principais aplicativos de chamada do sistema em um período de dez segundos:

```
#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;
}
```

O exemplo a seguir mostra o resultado da execução do script acima em um laptop com pouca carga:

```
  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
```

Minimizando cancelamentos

Como o DTrace armazena em buffer alguns dados de agregação no kernel, pode não haver espaço disponível quando uma nova chave for adicionada a uma agregação. Nesse caso, os dados serão cancelados, um contador será incrementado e o `dtrace` irá gerar uma mensagem indicando um cancelamento de agregação. Essa situação raramente acontece porque o DTrace mantém o estado de longa duração (que consiste na chave e no resultado intermediário da agregação) no nível do usuário onde o espaço pode crescer dinamicamente. Caso ocorram cancelamentos de agregação, você poderá aumentar o tamanho do buffer da agregação com a opção `aggsize` para reduzir a probabilidade de cancelamentos. Você também pode usar essa opção para minimizar as marcas de memória do DTrace. Assim como com qualquer opção de tamanho, `aggsize` pode ser especificada com qualquer sufixo de tamanho. A política de redimensionamento deste buffer é definida pela opção `bufresize` . Para obter mais detalhes sobre o armazenamento em buffer, consulte o Capítulo 11, “Buffers e armazenamento em buffer”. Para obter mais detalhes sobre as opções, consulte o Capítulo 16, “Opções e ajustáveis”.

Um método alternativo de eliminar cancelamentos de agregação é aumentar a taxa na qual os dados de agregação são consumidos no nível do usuário. O padrão dessa taxa é de uma vez por segundo e pode ser explicitamente ajustado com a opção `aggrate` . Assim como com qualquer opção de taxa, `aggrate` pode ser especificada com qualquer sufixo de tempo, mas o padrão é taxa por segundo. Para obter mais detalhes sobre a opção `aggsize`, consulte o Capítulo 16, “Opções e ajustáveis”.

◆ ◆ ◆ CAPÍTULO 10

10 Ações e sub-rotinas

Você pode usar chamadas de função de D como `trace()` e `printf()` para chamar dois tipos diferentes de serviços fornecidos pelo DTrace: *ações* que rastreiam dados ou modificam o estado externo do DTrace e *sub-rotinas* que afetam somente o estado interno do DTrace. Este capítulo define as ações e as sub-rotinas e descreve sua sintaxe e semântica.

Ações

As ações permitem que os programas do DTrace interajam com o sistema fora do DTrace. As ações mais comuns registram dados em um buffer do DTrace. Outras ações estão disponíveis, como a interrupção do processo atual, a elevação de um sinal específico no processo atual ou a interrupção do rastreio totalmente. Algumas dessas ações são *destrutivas*, pois alteram o sistema, ainda que numa maneira bem definida. Essas ações só podem ser usadas se as ações destrutivas tiverem sido explicitamente ativadas. Por padrão, as ações de registro de dados registram os dados no *buffer principal*. Para obter mais detalhes sobre o buffer principal e as diretivas de buffer, consulte o Capítulo 11, “Buffers e armazenamento em buffer”.

Ação padrão

Uma cláusula pode conter qualquer número de ações e manipulações de variáveis. Se uma cláusula for deixada em branco, a *ação padrão* será realizada. A ação padrão é rastrear o identificador do teste ativado (EPID) para o buffer principal. O EPID identifica uma ativação particular de um teste específico com ações e predicados particulares. Do EPID, os consumidores do DTrace podem determinar o teste que induziu a ação. Na verdade, sempre que quaisquer dados forem rastreados, eles devem ser acompanhados pelo EPID para que façam sentido para o consumidor. Por isso, a ação padrão é rastrear o EPID e nada mais.

O uso da ação padrão permite o uso simples de `dtrace(1M)`. Por exemplo, o seguinte comando ativa todos os testes no módulo de agendamento de uso compartilhado `TS` com a ação padrão:

```
# dtrace -m TS
```

O comando anterior pode produzir um resultado similar ao do seguinte exemplo:

```
# 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
...
```

Ações de registro de dados

As ações de registro de dados abrangem as principais ações do DTrace. Cada uma dessas ações registra dados no buffer principal por padrão, mas cada ação também pode ser usada para registrar dados em buffers especulativos. Consulte o Capítulo 11, "Buffers e armazenamento em buffer" para obter mais detalhes sobre o buffer principal. Consulte o Capítulo 13, "Rastreio especulativo" para obter mais detalhes sobre buffers especulativos. As descrições desta seção referem-se somente ao *buffer direcionado*, indicando que os dados são registrados no buffer principal ou em um buffer especulativo se a ação for após uma `speculate()`.

`trace()`

`void trace(`*expression*`)`

A ação mais básica é a `trace()`, que possui uma expressão de D como seu argumento e rastreia o resultado para o buffer direcionado. As seguintes instruções são exemplos de ações `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*`)`

A ação `tracemem()` possui uma expressão de D como seu primeiro argumento, *endereço* e uma constante como seu segundo argumento, *nbytes*. `tracemem ()` copia a memória do endereço especificado por *addr* para o buffer direcionado, de acordo com o tamanho especificado por *nbytes*.

printf()

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

Como `trace()`, a ação `printf()` rastreia expressões de D. Entretanto, `printf()` permite uma formatação de estilo `printf(3C)` elaborada. Como `printf(3C)`, os parâmetros consistem em uma seqüência de *formato* seguida por um número variável de argumentos. Por padrão, os argumentos são rastreados para o buffer direcionado. Os argumentos depois são formatados como resultado por `dtrace(1M)`, de acordo com a seqüência de formato especificada. Por exemplo, os dois primeiros exemplos de `trace()` de “`trace()`” na página 134 poderiam ser combinados em uma única `printf()`:

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

Para obter mais informações sobre `printf()`, consulte o Capítulo 12, “Formatação de saída”.

printa()

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

A ação `printa()` permite que você exiba e formate agregações. Consulte o Capítulo 9, “Agregações” para obter mais detalhes sobre agregações. Se um *formato* não for fornecido, `printa()` rastreará somente uma diretiva para o consumidor do DTrace de que a agregação especificada deve ser processada e exibida usando-se o formato padrão. Se um *formato* for fornecido, a agregação será formatada como especificado. Consulte o Capítulo 12, “Formatação de saída” para obter uma descrição mais detalhada da seqüência de formato `printa()`.

`printa()` rastreia somente uma *diretiva* de que a agregação deve ser processada pelo consumidor do DTrace. Ela não processa a agregação no kernel. Portanto, o tempo entre o rastreio da diretiva de `printa()` e o processamento real da diretiva depende dos fatores que afetam o processamento do buffer. Esses fatores incluem a taxa de agregação, a política de armazenamento em buffer e, se essa política for de `alternância`, a taxa na qual os buffers são alternados. Consulte o Capítulo 9, "Agregações" e o Capítulo 11, "Buffers e armazenamento em buffer" para obter descrições detalhadas sobre esses fatores.

stack()

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

A ação `stack()` registra um rastreio de pilha do kernel para o buffer direcionado. A pilha do kernel terá *nframes* de profundidade. Se *nframes* não for fornecida, o número de quadros de pilha registrado será o número especificado pela opção `stackframes` . Por exemplo:

```
# 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
   ...
```

A ação `stack()` é um pouco diferente de outras ações pois ela também pode ser usada como a chave para uma agregação:

```
# 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)
```

A ação `ustack()` registra um rastreio de pilha do *usuário* para o buffer direcionado. A pilha do usuário terá *nframes* de profundidade. Se *nframes* não for fornecida, o número de quadros de pilha registrado será o número especificado pela opção `ustackframes`. Embora `ustack()` possa determinar o endereço dos quadros de chamada quando o teste for acionado, os quadros de pilha não serão traduzidos em símbolos até que a ação `ustack()` seja processada no nível do usuário pelo consumidor do DTrace. Se *strsize* for especificado e diferente de zero, `ustack()` irá alocar a quantidade especificada de espaço de seqüência e usá-la para realizar a tradução de

endereço para símbolo diretamente do kernel. Essa tradução de símbolo direta do usuário atualmente está disponível apenas para máquinas virtuais com Java, versão 1.5 e posterior. A tradução de endereço em símbolo de Java anota as pilhas de usuário que contêm quadros de Java com o nome de método e classe de Java. Se tais quadros não puderem ser traduzidos, os quadros aparecerão somente como endereços hexadecimais.

O exemplo a seguir rastreia uma pilha sem espaço de seqüência e, portanto, nenhuma tradução de endereço em símbolo de Java:

```
# 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
              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
```

Observe que os quadros de pilhas de C e C++ da máquina virtual com Java são apresentados simbolicamente usando-se nomes de símbolos “corrompidos” de C++ e os quadros de pilhas de Java são apresentados somente como endereços hexadecimais. O exemplo a seguir mostra uma chamada para `ustack()` com um espaço de seqüência diferente de zero:

```
# 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
```

O exemplo de resultado acima demonstra informações simbólicas de quadro de pilhas de Java. Ainda existem alguns quadros hexadecimais nesse resultado porque algumas funções são estáticas e não possuem entradas na tabela de símbolos do aplicativo. A translação não é possível nesses quadros.

A translação de símbolo de `ustack()` para quadros que não sejam de Java ocorre *depois* que os dados da pilha são registrados. Por isso, o processo do usuário correspondente deve ser encerrado antes que a tradução de símbolo possa ser realizada, tornando a tradução do quadro de pilhas impossível. Se o processo do usuário for encerrado antes que a translação de símbolo seja realizada, o `dtrace` emitirá uma mensagem de aviso, seguida pelos quadros de pilhas hexadecimais, conforme mostrado no exemplo a seguir:

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

As técnicas para atenuar esse problema são descritas no Capítulo 33, “Rastreio de processo do usuário”.

Finalmente, como os comandos post mortem do depurador do DTrace não podem realizar a tradução do quadro, o uso de `ustack()` com uma política de buffer `ring` sempre resulta em dados `ustack()` não processados.

O programa em D a seguir mostra um exemplo de `ustack()` que deixa *strsize* sem especificação:

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

Para executar este exemplo para o navegador da Web Netscape, `.netscape.bin` em instalações padrão do Solaris, use o seguinte comando:

```
# 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()` é um alias para `ustack()` que usa a opção `jstackframes` para o número de quadros de pilha e para o tamanho de espaço de seqüência cujo valor é especificado pela opção `jstackstrsize`. O padrão de `jstacksize` é um valor diferente de zero. Isso significa que o uso de `jstack()` resultará em um pilha com tradução de quadro de Java no local.

Ações destrutivas

Algumas ações do DTrace são destrutivas pois alteram o estado do sistema de uma forma bem definida. As ações destrutivas não podem ser usadas a menos que tenham sido explicitamente ativadas. Ao usar o `dtrace(1M)`, você pode ativar ações destrutivas usando a opção `-w`. Se for feita uma tentativa de ativar ações destrutivas no `dtrace(1M)` sem ativá-las explicitamente, haverá uma falha no `dtrace` com uma mensagem similar ao seguinte exemplo:

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

Ações destrutivas de processo

Algumas ações são destrutivas somente para um processo em particular. Essas ações estão disponíveis para usuários com os privilégios `dtrace_proc` ou `dtrace_user`. Consulte o Capítulo 35, "Segurança" para obter detalhes sobre privilégios de segurança do DTrace.

`stop()`

```
void stop(void)
```

A ação `stop()` força o processo que aciona o teste ativado para parar quando ele deixar o kernel da próxima vez, como se fosse interrompido por uma ação `proc(4)`. O utilitário `prun(1)` pode ser usado para resumir um processo que foi interrompido pela ação `stop()`. A ação `stop()` pode ser usada para interromper um processo em qualquer ponto do teste de DTrace. Essa ação pode ser usada para capturar um programa em um estado particular que seria difícil de alcançar com um simples ponto de interrupção e depois anexar um depurador tradicional como o `mdb(1)` ao processo. Você também pode usar o utilitário `gcore(1)` para salvar o estado de um processo interrompido em um arquivo de núcleo para análise posterior.

`raise()`

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

A ação `raise()` envia o sinal especificado para o processo em execução no momento. Essa ação é similar ao uso do comando `kill(1)` para enviar um sinal a um processo. A ação `raise()` pode ser usada para enviar um sinal em um ponto preciso na execução de um processo.

`copyout()`

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

A ação `copyout()` copia *nbytes* do buffer especificado por *buf* para o endereço especificado por *addr* no espaço de endereço do processo associado ao segmento atual. Se o endereço do espaço do usuário não corresponder a uma página válida, com falhas, no espaço de endereço atual, um erro será gerado.

`copyoutstr()`

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

A ação `copyoutstr()` copia a seqüência especificada por *str* para o endereço especificado por *addr* no espaço de endereço do processo associado ao segmento atual. Se o endereço do espaço do usuário não corresponder a uma página válida, com falhas, no espaço de endereço atual, um erro será gerado. O tamanho da seqüência é limitado ao valor definido pela opção `strsize`. Consulte o Capítulo 16, “Opções e ajustáveis” para obter detalhes.

`system()`

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

A ação `system ()` faz com que o programa especificado por *programa* seja executado como se fosse fornecido ao shell como entrada. A seqüência de *programa* pode conter qualquer uma das conversões de formato de `printf()`/ `printa. ()` Os argumentos que correspondam às conversões de formato devem ser especificados. Consulte o Capítulo 12, “Formatação de saída” para obter detalhes sobre as conversões de formato válidas.

O exemplo a seguir executa o comando `date(1)` uma vez por segundo:

```
# 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
```

O exemplo a seguir mostra um uso mais elaborado da ação, usando conversões de `printf()` na seqüência de *programa* junto com ferramentas de filtragem tradicionais como pipes:

```
#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);
}
```

Executar o script acima resultará numa saída semelhante ao exemplo seguinte:

```
# ./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
```

A execução do comando especificado *não* ocorre no contexto do teste acionado – ela ocorre quando o buffer que contém os detalhes da ação `system()` é processado no nível do usuário. Como e quando esse processamento ocorre depende da política de buffer, descrita no Capítulo 11, “Buffers e armazenamento em buffer”. Com a política de buffer padrão, a taxa de processamento do buffer é especificada pela opção `switchrate` . Você poderá ver o atraso inerente em `system()` se ajustar explicitamente `switchrate` com um valor mais alto que o padrão de um segundo, conforme mostrado no exemplo a seguir:

```
#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);
}
```

Executar o script acima resultará numa saída semelhante ao exemplo seguinte:

```
# 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
```

Observe que os valores de `walltime` diferem, mas os valores de `date` são idênticos. Esse resultado reflete o fato de que a execução do comando `date(1)` ocorreu somente quando o buffer foi processado, não quando a ação `system()` foi registrada.

Ações destrutivas do kernel

Algumas ações são destrutivas para todo o sistema. Essas ações obviamente devem ser usadas com muito cuidado, pois elas irão afetar todos os processos no sistema e qualquer outro sistema que dependa implicita ou explicitamente dos serviços de rede do sistema afetado.

`breakpoint()`

```
void breakpoint(void)
```

A ação `breakpoint()` induz a um ponto de interrupção do kernel, fazendo com que o sistema pare e transfira o controle para o depurador do kernel. O depurador do kernel emitirá uma seqüência indicando o teste do DTrace que acionou a ação. Por exemplo, se a seguinte ação fosse realizada:

```
# dtrace -w -n clock:entry'{breakpoint()}'
dtrace: allowing destructive actions
dtrace: description 'clock:entry' matched 1 probe
```

No Solaris em execução no SPARC, a seguinte mensagem poderia aparecer no console:

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb 30002765700)
Type  'go' to resume
ok
```

No Solaris em execução no x86, a seguinte mensagem poderia aparecer no console:

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb d2b97060)
stopped at      int20+0xb:      ret
kmdb[0]:
```

O endereço após a descrição do teste é o endereço do bloco de controle de ativação (ECB) no DTrace. Você pode usar esse endereço para determinar mais detalhes sobre a ativação do teste que induziu a ação de ponto de interrupção.

Um erro na ação `breakpoint()` pode fazer com que ela seja chamada mais vezes do que o pretendido. Esse comportamento pode, por sua vez, impedir até mesmo que você encerre o consumidor do DTrace que está acionando as ações de ponto de interrupção. Nessa situação,

defina a variável de inteiro do kernel `dtrace_destructive_disallow` como 1. Essa configuração irá impedir *todas* as ações destrutivas na máquina. Aplique essa configuração *somente* nessa situação em particular.

O método exato para a definição de `dtrace_destructive_disallow` dependerá do depurador do kernel que você está usando. Se estiver usando o PROM de OpenBoot em um sistema SPARC, use `w!`:

```
ok 1 dtrace_destructive_disallow w!
ok
```

Confirme que a variável foi definida usando-se `w?`:

```
ok dtrace_destructive_disallow w?
1
ok
```

Continue digitando `go`:

```
ok go
```

Se estiver usando `kmdb(1)` em sistemas x86 ou SPARC, use o modificador de gravação de 4 bytes (`W`) com o dcmd de formatação `/`:

```
kmdb[0]: dtrace_destructive_disallow/W 1
dtrace_destructive_disallow:    0x0             =       0x1
kmdb[0]:
```

Continue usando `:c`:

```
kadb[0]: :c
```

Para reativar ações destrutivas após prosseguir, você precisará redefinir explicitamente `dtrace_destructive_disallow` de volta para 0 usando `mdb(1)`:

```
# echo "dtrace_destructive_disallow/W 0" | mdb -kw
dtrace_destructive_disallow:    0x1             =       0x0
#
```

panic()

```
void panic(void)
```

A ação `panic ()` causa um aviso grave no kernel quando acionada. Essa ação deve ser usada para forçar um despejo de memória do sistema quando for necessário. Você pode usar essa ação junto com o buffer de anel e a análise post mortem para compreender um problema. Para obter mais informações, consulte o Capítulo 11, “Buffers e armazenamento em buffer” e o Capítulo 37, “Rastreio post-mortem” respectivamente. Quando a ação panic é usada, uma mensagem é exibida indicando o teste que causou o aviso grave. Por exemplo:

```
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) também emitirá uma mensagem na reinicialização:

```
Jun 10 16:56:31 machine1 savecore: [ID 570001 auth.error] reboot after panic:
dtrace: panic action at probe syscall::mmap:entry (ecb 300000acfc8)
```

O buffer de mensagem do despejo de memória também contém o teste e o ECB responsável pela ação `panic()`.

`chill()`

`void chill(int` *nanoseconds*`)`

A ação `chill ()` faz com que o DTrace gire para o número especificado de nanossegundos. `chill()` é principalmente útil para explorar problemas que possam estar relacionados ao tempo. Por exemplo, você pode usar essa ação para abrir janelas de condição de corrida ou para colocar ou tirar eventos periódicos de fase entre si. Como as interrupções são desativadas durante o contexto de teste do DTrace, qualquer uso de `chill()` irá induzir à latência de interrupção, latência de agendamento e latência de distribuição. Portanto, `chill()` pode causar efeitos sistêmicos não esperados e não deve ser usada indiscriminadamente. Como a atividade do sistema depende de uma manipulação periódica de interrupção, o DTrace irá se recusar a executar a ação `chill()` por mais de 500 milissegundos de cada intervalo de um segundo em uma determinada CPU. Se o intervalo máximo de `chill()` for excedido, o DTrace irá relatar um erro de operação ilegal, conforme mostrado no seguinte exemplo:

```
# 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
```

Esse limite é aplicado mesmo que o tempo seja distribuído entre várias chamadas para `chill()` ou vários consumidores de DTrace de um único teste. Por exemplo, o mesmo erro seria gerado pelo seguinte comando:

```
# dtrace -w -n syscall::open:entry'{chill(250000000); chill(250000001);}'
```

Ações especiais

Esta seção descreve ações que não são de registro de dados ou destrutivas.

Ações especulativas

As ações associadas ao rastreio especulativo são `speculate ()`, `commit()` e `discard()`. Essas ações são tratadas no Capítulo 13, “Rastreio especulativo”.

`exit()`

`void exit(int` *`status`*`)`

A ação `exit()` é usada para interromper o rastreio imediatamente e para informar ao consumidor do DTrace que ele deve interromper o rastreio, realizar o processamento final e chamar `exit(3C)` com o status especificado. Como `exit()` retorna um status para o nível do usuário, ela é uma ação de registro de dados. Entretanto, ao contrário de outras ações de armazenamento de dados, `exit()` não pode ser rastreada especulativamente. `exit()` fará com que o consumidor do DTrace seja encerrado independentemente da política do buffer. Como `exit()` é uma ação de registro de dados, ela *pode* ser cancelada.

Quando `exit()` é chamada, somente as ações do DTrace já em andamento em outras CPUs serão concluídas. Nenhuma ação nova ocorrerá em qualquer CPU. A única exceção a essa regra é o processamento do teste `END`, que será chamado depois que o consumidor do DTrace tiver processado a ação `exit()` e indicado que o rastreio deve ser encerrado.

Sub-rotinas

As sub-rotinas são diferentes das ações porque elas geralmente afetam apenas o estado interno do DTrace. Portanto, não há sub-rotinas destrutivas e as sub-rotinas nunca rastreiam dados para buffers. Muitas sub-rotinas possuem análogos nas interfaces da Seção 9F ou da Seção 3C. Consulte `Intro(9F)` e `Intro(3)` para obter mais informações sobre as sub-rotinas correspondentes.

alloca()

```
void *alloca(size_t size)
```

`alloca()` aloca *tamanho* bytes fora do espaço temporário e retorna um ponteiro para a memória alocada. É garantido que o ponteiro retornado tenha um alinhamento de 8 bytes. O espaço temporário só é válido enquanto durar uma cláusula. A memória alocada com `alloca()` será desalocada quando a cláusula for concluída. Se não houver espaço temporário suficiente disponível, não será alocada a memória e um erro será gerado.

basename()

```
string basename(char *str)
```

`basename()` é um análogo de D para `basename(1)`. Esta sub-rotina cria uma seqüência que consiste em uma cópia da seqüência especificada, mas sem um prefixo que termine em `/`. A seqüência retornada é alocada fora da memória temporária e, portanto, é válida somente enquanto durar a cláusula. Se não houver espaço temporário suficiente disponível, `basename` não será executada e um erro será gerado.

bcopy()

```
void bcopy(void *src, void *dest, size_t size)
```

`bcopy()` copia *tamanho* bytes da memória apontada por *src* para a memória apontada por *dest*. Toda a memória de origem deve estar fora da memória temporária e toda a memória de destino deve estar dentro dela. Se essas condições não forem atendidas, a cópia não será realizada e um erro é gerado.

cleanpath()

```
string cleanpath(char *str)
```

`cleanpath()` cria uma seqüência que consiste em uma cópia do caminho indicado por *str*, mas com alguns elementos redundantes eliminados. Em particular, os elementos "`/./`" no caminho são removidos e os elementos "`/../`" são recolhidos. Os elementos `/../` são recolhidos no caminho sem levar em consideração os links simbólicos. Portanto, é possível que `cleanpath()` obtenha um caminho válido e retorne um mais curto, inválido.

Por exemplo, se *str* fosse "`/foo/../bar`" e `/foo` fosse um link simbólico para `/net/foo/export`, `cleanpath()` retornaria a seqüência "`/bar`" embora `bar` só possa estar em `/net/foo` e não em `/`. Essa limitação deve-se ao fato de que `cleanpath()` é chamada no contexto de um teste acionado, onde a resolução completa de link simbólico ou nomes arbitrários não são possíveis. A seqüência retornada é alocada fora da memória temporária e, portanto, é válida somente enquanto durar a cláusula. Se não houver espaço temporário suficiente disponível, `cleanpath` não será executada e um erro será gerado.

copyin()

```
void *copyin(uintptr_t addr, size_t size)
```

`copyin()` copia o tamanho especificado em bytes do endereço de usuário especificado em um buffer temporário do DTrace e retorna o endereço desse buffer. O endereço do usuário é interpretado como um endereço no espaço do processo associado ao segmento atual. É garantido que o ponteiro de buffer resultante tenha um alinhamento de 8 bytes. O endereço em questão *deve* corresponder a uma página com falhas no processo atual. Se o endereço não corresponder a uma página com falhas, ou não houver espaço temporário suficiente disponível, `NULL` será retornado e um erro será gerado. Consulte o Capítulo 33, "Rastreio de processo do usuário" para saber as técnicas para reduzir a probabilidade de erros de `copyin`.

copyinstr()

```
string copyinstr(uintptr_t addr)
```

`copyinstr()` copia uma seqüência de C terminada com caractere nulo do endereço de usuário especificado para um buffer temporário do DTrace e retorna o endereço desse buffer. O endereço do usuário é interpretado como um endereço no espaço do processo associado ao segmento atual. O tamanho da seqüência é limitado pelo valor definido pela opção `strsize`. Consulte o Capítulo 16, "Opções e ajustáveis" para obter detalhes. Assim como com `copyin`, o endereço especificado *deve* corresponder a uma página com falhas no processo atual. Se o endereço não corresponder a uma página com falhas, ou não houver espaço temporário suficiente disponível, `NULL` será retornado e um erro será gerado. Consulte o Capítulo 33, "Rastreio de processo do usuário" para saber as técnicas para reduzir a probabilidade de erros de `copyinstr`.

copyinto()

`void copyinto(uintptr_t` *addr*`, size_t` *size*`, void *`*dest*`)`

`copyinto()` copia o tamanho especificado em bytes do endereço de usuário especificado para o buffer temporário do DTrace especificado por *dest*. O endereço do usuário é interpretado como um endereço no espaço do processo associado ao segmento atual. O endereço em questão *deve* corresponder a uma página com falhas no processo atual. Se o endereço não corresponder a uma página com falhas, ou se uma das memórias de destino estiver fora do espaço temporário, não ocorrerá uma cópia e um erro será gerado. Consulte o Capítulo 33, "Rastreio de processo do usuário" para saber as técnicas para reduzir a probabilidade de erros de `copyinto`.

dirname()

`string dirname(char *`*str*`)`

`dirname()` é um análogo de D para `dirname(1)`. Essa sub-rotina cria uma seqüência que consiste em todos os níveis, exceto o último, do nome de caminho especificado por *str*. A seqüência retornada é alocada fora da memória temporária e, portanto, é válida somente enquanto durar a cláusula. Se não houver espaço temporário suficiente disponível, `dirname` não será executada e um erro é gerado.

msgdsize()

`size_t msgdsize(mblk_t *`*mp*`)`

`msgdsize()` retorna o número de bytes na mensagem de dados apontada por *mp*. Consulte `msgdsize(9F)` para obter detalhes. `msgdsize()` inclui somente blocos de dados do tipo `M_DATA` na contagem.

msgsize()

`size_t msgsize(mblk_t *`*mp*`)`

`msgsize()` retorna o número de bytes na mensagem apontada por *mp*. Ao contrário de `msgdsize ()`, que retorna somente o número de bytes dos *dados*, `msgsize()` retorna o número *total* de bytes na mensagem.

mutex_owned()

`int mutex_owned(kmutex_t *`*mutex*`)`

`mutex_owned()` é uma implementação de `mutex_owned(9F)`. `mutex_owned()` retorna um valor diferente de zero se o segmento de chamada possuir atualmente o mutex do kernel especificado ou zero se o mutex adaptável especificado não possuir proprietário no momento.

mutex_owner()

`kthread_t *mutex_owner(kmutex_t *`*mutex*`)`

`mutex_owner()` retorna o ponteiro do segmento do proprietário atual do mutex do kernel adaptável especificado. `mutex_owner()` retorna `NULL` se o mutex adaptável especificado não possuir proprietário no momento ou se o mutex especificado for um mutex de rotação. Consulte `mutex_owned(9F)`.

mutex_type_adaptive()

`int mutex_type_adaptive(kmutex_t *`*mutex*`)`

`mutex_type_adaptive()` retorna um valor diferente de zero se o mutex do kernel especificado for do tipo `MUTEX_ADAPTIVE` , ou zero se não for. Os mutexes são adaptáveis se atenderem a uma ou mais das seguintes condições:

- O mutex é declarado estaticamente
- O mutex é criado com um cookie de bloqueio de interrupção de NULL
- O mutex é criado com um cookie de bloqueio de interrupção que não corresponde a uma interrupção de alto nível

Consulte `mutex_init(9F)` para obter mais detalhes sobre mutexes. A maioria dos mutexes no kernel do Solaris são adaptáveis.

progenyof()

`int progenyof(pid_t` *pid*`)`

`progenyof()` retorna um valor diferente de zero se o processo de chamada (o processo associado ao segmento que está acionando atualmente o teste correspondido) estiver entre a origem do ID especificado do processo.

rand()

`int rand(void)`

`rand()` retorna um inteiro pseudo-aleatório. O número retornado é um número pseudo-aleatório fraco e não deve ser usado para qualquer aplicação criptográfica.

rw_iswriter()

```
int rw_iswriter(krwlock_t *rwlock)
```

`rw_iswriter()` retorna um valor diferente de zero se o bloqueio de leitor-autor especificado for mantido ou desejado por um autor. Se o bloqueio for mantido somente por leitores e nenhum autor estiver bloqueado ou se o bloqueio não for mantido, `rw_iswriter()` retornará zero. Consulte `rw_init`(9F).

rw_write_held()

```
int rw_write_held(krwlock_t *rwlock)
```

`rw_write_held()` retorna um valor diferente de zero se o bloqueio de leitor-autor especificado estiver sendo mantido atualmente por um autor. Se o bloqueio for mantido somente por leitores ou não for mantido, `rw_write_held ()` retornará zero. Consulte `rw_init`(9F).

speculation()

```
int speculation(void)
```

`speculation()` reserva um buffer de rastreio especulativo para uso com `speculate()` e retorna um identificador para esse buffer. Consulte o Capítulo 13, “Rastreio especulativo” para obter detalhes.

strjoin()

```
string strjoin(char *str1, char *str2)
```

`strjoin()` cria uma seqüência que consiste em *str1* concatenado com *str2*. A seqüência retornada é alocada fora da memória temporária e, portanto, é válida somente enquanto durar a cláusula. Se não houver espaço temporário suficiente disponível, `strjoin` não será executada e um erro será gerado.

strlen()

```
size_t strlen(string str)
```

`strlen()` retorna o tamanho da seqüência especificada em bytes, excluindo o byte nulo de terminação.

◆ ◆ ◆ CAPÍTULO 11

Buffers e armazenamento em buffer

O armazenamento de dados em buffer e o seu gerenciamento são serviços essenciais fornecidos pela estrutura do DTrace a seus clientes, como o `dtrace(1M)`. Este capítulo explora o armazenamento de dados em buffer detalhadamente e descreve opções que podem ser usadas para alterar as políticas de gerenciamento do buffer do DTrace.

Buffers principais

O *buffer principal* está presente em cada chamada do DTrace e é o buffer no qual as ações de rastreio registram seus dados por padrão. Estas ações incluem:

`exit()`	`printf()`	`trace()`	`ustack()`
`printa()`	`stack()`	`tracemem()`	

Os buffers principais *sempre* são alocados em uma base por CPU. Essa política não é ajustável, mas o rastreio e a alocação do buffer podem ser restritos a uma única CPU usando-se a opção `cpu`.

Políticas do buffer principal

O DTrace permite o rastreio em contextos altamente restritos no kernel. Em particular, o DTrace permite o rastreio em contextos nos quais a alocação de memória pelo software do kernel possa não ser confiável. A conseqüência dessa flexibilidade de contexto é que *sempre* existe a possibilidade de que o DTrace irá tentar rastrear os dados quando não houver espaço disponível. O DTrace deve ter uma política para lidar com tais situações quando elas ocorrerem, mas você pode querer ajustar a política com base nas necessidades de um determinado experimento. Algumas vezes a política apropriada pode ser de descartar os novos dados. Outras vezes, talvez seja desejável reutilizar o espaço que contém os dados registrados mais antigos

para rastrear novos dados. Mais freqüentemente, a política desejada é de minimizar a probabilidade de ficar sem espaço disponível em primeiro lugar. Para acomodar todas essas demandas, o DTrace oferece suporte a várias políticas de buffer diferentes. Esse suporte é implementado com a opção `bufpolicy`, e pode ser definido em uma base por consumidor. Consulte o Capítulo 16, “Opções e ajustáveis” para obter detalhes sobre as opções de configuração.

Política `switch`

Por padrão, o buffer principal possui uma política `switch`. Nessa política, os buffers por CPU são alocados em pares: um buffer é ativo e ou outro, inativo. Quando um consumidor do DTrace tenta ler um buffer, o kernel primeiro *alterna* os buffers inativos e ativos. A alternância de buffer é feita de tal maneira que não haja uma janela na qual os dados do rastreio possam se perder. Depois que os buffers forem alternados, o buffer recém-inativo é copiado para o consumidor do DTrace. Essa política garante que o consumidor sempre veja um buffer auto-consistente: um buffer nunca é simultaneamente rastreado e copiado. Essa técnica também impede que seja introduzida uma janela na qual o rastreio seja pausado ou, de alguma forma, impedido. A taxa na qual o buffer é alternado e lido é controlada pelo consumidor com a opção `switchrate`. Assim como com qualquer opção de taxa, `switchrate` pode ser especificada com qualquer sufixo de tempo, mas o padrão é taxa por segundo. Para obter mais detalhes sobre `switchrate` e outras opções, consulte o Capítulo 16, “Opções e ajustáveis”.

Observação – Para processar o buffer principal no nível do usuário mais rápido que o padrão que é de uma vez por segundo, ajuste o valor de `switchrate`. O sistema processa as ações que produzem atividade no nível do usuário (como `printa()` e `system()`) quando o registro correspondente no buffer principal é processado. O valor de `switchrate` determina a velocidade na qual o sistema processa tais ações.

Na política `switch`, se um determinado teste ativado rastrear mais dados do que o espaço disponível no buffer principal ativo, os dados serão *cancelados* e uma contagem de cancelamento por CPU é incrementada. No caso de um ou mais cancelamentos, o `dtrace(1M)` exibe uma mensagem similar ao seguinte exemplo:

```
dtrace: 11 drops on CPU 0
```

Se um determinado registro for maior que o tamanho total do buffer, o registro será cancelado independentemente da política do buffer. Você pode reduzir ou eliminar cancelamentos, seja aumentando o tamanho do buffer principal com a opção `bufsize` ou aumentando a taxa de alternância com a opção `switchrate`.

Na política `switch`, o espaço temporário para `copyin()`, `copyinstr()` e `alloca()` é alocado fora do buffer ativo.

Política `fill`

Para alguns problemas, você pode querer usar um único buffer em kernel. Embora essa abordagem possa ser implementada com a política `switch` e construções apropriadas de D incrementando-se uma variável em D e criando-se um predicado para uma ação `exit()` apropriadamente, tal implementação não elimina a possibilidade de cancelamentos. Para solicitar um único buffer grande em kernel e prosseguir com o rastreio até que um ou mais dos buffers por CPU tenha sido preenchido, use a política de buffer `fill`. Nessa política, o rastreio continua até que um teste ativado tente rastrear mais dados que possam caber no espaço restante do buffer principal. Quando não houver espaço suficiente restante, o buffer é marcado como preenchido e o consumidor é notificado que pelo menos um dos buffers por CPU foi preenchido. Quando o `dtrace`(1M) detecta um único buffer preenchido, o rastreio é interrompido, todos os buffers são processados e `dtrace` é encerrado. Dados adicionais não serão rastreados para um buffer preenchido mesmo que esses dados caibam nele.

Para usar a política `fill`, defina a opção `bufpolicy` como `fill`. Por exemplo, o seguinte comando rastreia cada entrada de chamada do sistema em um buffer por CPU de 2K com a política de buffer definida como `fill`:

```
# dtrace -n syscall:::entry -b 2k -x bufpolicy=fill
```

Política `fill` e testes `END`

Os testes `END` normalmente não são acionados até que o rastreio tenha sido explicitamente interrompido pelo consumidor do DTrace. É garantido que os testes `END` sejam acionados apenas em uma CPU, mas a CPU na qual o teste é acionado não é definida. Com buffers `fill`, o rastreio é explicitamente interrompido quando pelo menos um dos buffers principais por CPU tiver sido marcado como preenchido. Se a política `fill` estiver selecionada, o teste `END` pode ser acionado em uma CPU que possua um buffer preenchido. Para acomodar o rastreio `END` em buffers `fill`, o DTrace calcula a quantidade de espaço potencialmente consumido pelos testes `END` e *subtrai* esse espaço do tamanho do buffer principal. Se o tamanho líquido for negativo, o DTrace se recusará a iniciar e o `dtrace`(1M) retornará uma mensagem de erro correspondente:

```
dtrace: END enablings exceed size of principal buffer
```

O mecanismo de reserva garante que um buffer completo sempre possui espaço suficiente para quaisquer testes `END`.

Política `ring`

A política de buffer `ring` do DTrace auxilia no rastreio de eventos que levam a falha. Se a reprodução da falha demorar horas ou dias, talvez você queira manter somente os dados mais recentes. Depois que um buffer principal for preenchido, o rastreio retorna para a primeira entrada, sobrescrevendo dados de rastreio antigos. Estabeleça o buffer de anel, configurando a opção `bufpolicy` para a seqüência `ring`:

```
# dtrace -s foo.d -x bufpolicy=ring
```

Quando usado para criar um buffer de anel, o dtrace(1M) não exibirá qualquer resultado até que o processo esteja terminado. Nesse momento, o buffer de anel estará consumido e processado. O `dtrace` processa cada buffer de anel em ordem de CPU. Em um buffer de CPU, os registros do rastreio serão exibidos na ordem do mais antigo para o mais novo. Assim como na política de buffer `switch`, não existe ordem entre os registros de CPUs diferentes. Se for necessária uma ordem, você deve rastrear a variável `timestamp` como parte da solicitação de rastreio.

O exemplo a seguir demonstra o uso de uma diretiva `#pragma option` para ativar o buffer em anel:

```
#pragma D option bufpolicy=ring
#pragma D option bufsize=16k

syscall::::entry
/execname == $1/
{
    trace(timestamp);
}

syscall::rexit:entry
{
    exit(0);
}
```

Outros buffers

Os buffers principais existem em todas as ativações do DTrace. Além dos buffers principais, alguns consumidores do DTrace podem ter buffers de dados adicionais em kernel: um *buffer de agregação*, discutido no Capítulo 9, “Agregações” e um ou mais *buffers especulativos*, discutidos no Capítulo 13, “Rastreio especulativo”.

Tamanhos de buffer

O tamanho de cada buffer pode ser ajustado em uma base por consumidor. Opções separadas são fornecidas para ajustar o tamanho de cada buffer, conforme mostrado na tabela a seguir:

Buffer	Opção de tamanho
Principal	`bufsize`

Buffer	Opção de tamanho
Especulativo	`specsize`
Agregação	`aggsize`

Cada uma destas opções é definida com um valor que denota o tamanho. Assim como com qualquer opção de tamanho, o valor pode ter um sufixo de tamanho opcional. Consulte o Capítulo 16, "Opções e ajustáveis" para obter mais detalhes. Por exemplo, para definir o tamanho do buffer como um megabyte na linha de comando de `dtrace`, você pode usar `-x` para definir a opção:

```
# dtrace -P syscall -x bufsize=1m
```

Como alternativa, você pode usar a opção `-b` para `dtrace` :

```
# dtrace -P syscall -b 1m
```

Finalmente, você pode definir `bufsize` usando `#pragma D option`:

```
#pragma D option bufsize=1m
```

O tamanho de buffer selecionado denota o tamanho do buffer em *cada* CPU. Além disso, para a política de buffer `switch`, `bufsize` denota o tamanho de *cada* buffer em cada CPU. O padrão do tamanho de buffer é quatro megabytes.

Política de redimensionamento de buffer

Ocasionalmente, o sistema pode não ter memória de kernel livre adequada para alocar um buffer de tamanho desejado seja porque não há memória suficiente disponível ou porque o consumidor do DTrace excedeu um dos limites ajustáveis descritos no Capítulo 16, "Opções e ajustáveis". Você pode configurar a política para falha de alocação de buffer usando a opção `bufresize` , que tem como padrão `auto`. Na política de redimensionamento de buffer `auto`, o tamanho de um buffer é dividido até que uma alocação bem-sucedida ocorra. `dtrace`(1M) gera uma mensagem se um buffer alocado for menor que o tamanho solicitado:

```
# 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
...
```

Como alternativa, você pode requerer intervenção manual após falha na alocação do buffer definindo `bufresize` como `manual`. Nessa política, uma falha de alocação fará com que o DTrace deixe de ser iniciado:

```
# dtrace -P syscall -x bufsize=1g -x bufresize=manual
dtrace: description 'syscall' matched 430 probes
dtrace: could not enable tracing: Not enough space
#
```

A política de redimensionamento de buffer de *todos* os buffers, principal, especulativo e agregação, é ditada pela opção `bufresize`.

◆ ◆ ◆ CAPÍTULO 12

Formatação de saída

O DTrace fornece funções de formatação incorporadas `printf()` e `printa()` que você pode usar em seus programas em D para formatar a saída. O compilador de D fornece recursos não encontrados na rotina de biblioteca `printf(3C)`, sendo assim, você deve ler este capítulo mesmo que já esteja familiarizado com `printf()`. Este capítulo também discute o comportamento da formatação da função `trace()` e o formato de saída padrão usado por `dtrace(1M)` para exibir agregações.

printf()

A função `printf()` combina a capacidade de rastrear dados, como se fosse a função `trace()`, com a capacidade de produzir a saída de dados e de outro texto em um formato específico descrito por você. A função `printf()` informa ao DTrace para rastrear os dados associados a cada argumento após o primeiro argumento e depois para formatar os resultados usando as regras descritas pelo primeiro argumento de `printf ()`, conhecido como *seqüência de formato*.

A seqüência de formato é uma seqüência regular que contém inúmeras conversões de formato, cada uma começando com o caractere `%`, que descreve como formatar o argumento correspondente. A primeira conversão na seqüência de formato corresponde ao segundo argumento `printf()`, a segunda conversão ao terceiro argumento, e assim por diante. Todo o texto entre conversões é impresso textualmente. O caractere que segue o caractere de conversão `%` descreve o formato a ser usado para o argumento correspondente.

Ao contrário de `printf(3C)`, DTrace `printf()` é uma função incorporada que é reconhecida pelo compilador de D. O compilador de D fornece vários serviços úteis do DTrace `printf()` que não são encontrados na biblioteca de C `printf()`:

- O compilador de D compara os argumentos com as conversões na seqüência de formato. Se um tipo de argumento for incompatível com a conversão de formato, o compilador de D exibirá uma mensagem de erro explicando o problema.

- O compilador de D não requer o uso de prefixos de tamanho com conversões de formato `printf. ()` A rotina `printf()` de C requer que você indique o tamanho dos argumentos, adicionando prefixos, tais como `%ld` for `long` ou `%lld` for `long long`. O compilador de D conhece o tamanho e o tipo dos seus argumentos, sendo assim, esses prefixos não são necessários em suas declarações `printf()` de D.
- O DTrace fornece caracteres de formato adicionais que são úteis para depuração e observação. Por exemplo, a conversão de formato `%a` pode ser usada para imprimir um ponteiro como um nome de símbolo e um deslocamento.

Para implementar esses recursos, a seqüência de formato na função `printf()` do DTrace deve ser especificada como uma constante de seqüência em seu programa em D. As seqüências de formato podem ser variáveis dinâmicas do tipo `string`.

Especificações de conversão

Cada especificação de conversão na seqüência de formato é introduzida pelo caractere `%`, após o qual as informações seguintes aparecem em seqüência:

- Zero ou mais *sinalizadores* (em qualquer ordem), que modificam o significado das especificações de conversão, conforme descrito na próxima seção.
- Uma *largura de campo* mínima ótima. Se o valor convertido tiver menos bytes que a largura do campo, o valor será preenchido com espaços à esquerda, por padrão, ou à direita, se o sinalizador de ajuste esquerdo (`-`) for especificado. A largura do campo também pode ser especificada como um asterisco (`*`), nesse caso, a largura do campo é definida dinamicamente com base no valor de um argumento adicional do tipo `int`.
- Uma *precisão* opcional que indica o número mínimo de dígitos que aparecem nas conversões `d`, `i`, `o`, `u`, `x` e `X` (o campo é preenchido com zeros à esquerda); o número de dígitos que aparecem após o caractere fracionário das conversões `e`, `E` e `f`, o número máximo de dígitos significativos das conversões `g` e `G`; ou o número máximo de bytes a serem impressos de uma seqüência pela conversão `s`. A precisão toma o formato de um ponto (`.`) seguido por um asterisco (`*`), descrito abaixo, ou uma seqüência de dígito decimal.
- Uma seqüência opcional de *prefixos de tamanho* que indica o tamanho do argumento correspondente, descrito em "Prefixos de tamanho" na página 164. Os prefixos de tamanho não são necessários em D e são fornecidos para compatibilidade com a função `printf()` de C.
- Um *especificador de conversão* que indica o tipo de conversão a ser aplicado ao argumento.

A função `printf(3C)` também aceita as especificações de conversão do formato `%` *n*`$` onde *n* é um inteiro decimal. A `printf()` do DTrace não aceita este tipo de especificação de conversão.

Especificadores de sinalizador

Os sinalizadores de conversão `printf()` são ativados pela especificação de um ou mais dos seguintes caracteres, que podem aparecer em qualquer ordem:

`'` A porção inteira do resultado de uma conversão decimal (`%i`, `%d`, `%u`, `%f`, `%g` ou `%G`) é formatada com milhares de caracteres de agrupamento por meio do caractere de agrupamento não-monetário. Algumas localidades, incluindo a localidade POSIX C, não fornece caracteres de agrupamento não-monetários a serem usados com este sinalizador.

`-` O resultado da conversão é justificado à esquerda no campo. A conversão será justificada à direita, se este sinalizador não for especificado.

`+` O resultado da conversão assinada sempre começa com um sinal (`+` ou `-`). Se este sinalizador não for especificado, a conversão começa com um sinal somente quando um valor negativo for convertido.

`espaço` Se o primeiro caractere de uma conversão assinada não for um sinal ou se uma conversão assinada resultar em nenhum caractere, um espaço será colocado antes do resultado. Se os sinalizadores `espaço` e `+` aparecerem, o sinalizador de espaço será ignorado.

`#` O valor será convertido em um formato alternativo, se for definido um formato alternativo para a conversão selecionada. Os formatos alternativos para conversões são descritos junto com a conversão correspondente.

`0` Para as conversões `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, e `G`, zeros à esquerda (seguindo qualquer indicação de sinal ou base) são usados para preencher a largura do campo. Nenhum preenchimento de espaço é realizado. Se os sinalizadores `0` e `-` aparecerem, o sinalizador `0` será ignorado. Para as conversões `d`, `i`, `o`, `u`, `x` e `X`, se uma precisão for especificada, o sinalizador `0` será ignorado. Se os sinalizadores `0` e `'` aparecerem, os caracteres de agrupamento serão inseridos antes do preenchimento de zero.

Especificadores de largura e de precisão

A largura de campo mínima pode ser especificada como uma seqüência de dígitos decimais seguindo qualquer especificador de sinalizador, nesse caso, a largura do campo é definida como o número de colunas especificadas. A largura do campo também pode ser especificada como um asterisco (`*`), nesse caso, um argumento adicional do tipo `int` é acessado para determinar a largura do campo. Por exemplo, para imprimir um inteiro `x` em uma largura de campo determinada pelo valor da variável `int w`, você escreveria a declaração de D:

```
printf("%*d", w, x);
```

A largura do campo também pode ser especificada com um caractere ? para indicar que a largura do campo deve ser definida com base no número de caracteres necessários para formatar um endereço em hexadecimal no modelo de dados do kernel do sistema operacional. A largura será definida como 8, se o kernel estiver usando o modelo de dados de 32 bits, ou como 16 se o kernel estiver usando o modelo de dados de 64 bits.

A precisão da conversão pode ser especificada como uma seqüência de dígitos decimais seguindo um ponto (`.`) ou por um asterisco (`*`) seguindo um ponto. Se um asterisco for usado para especificar a precisão, um argumento adicional do tipo `int` anterior ao argumento de conversão será acessado para determinar a precisão. Se a largura e a precisão forem especificadas como asteriscos, a ordem de argumentos para `printf()` da conversão deve aparecer na seguinte ordem: largura, precisão, valor.

Prefixos de tamanho

Os prefixos de tamanho são necessários nos programas em ANSI-C que usam `printf(3C)` a fim de indicar o tamanho e o tipo do argumento de conversão. O compilador de D realiza esse processamento para as suas chamadas `printf ()` automaticamente, sendo assim, os prefixos de tamanho não são necessários. Embora os prefixos de tamanho sejam fornecidos para compatibilidade com C, seu uso é explicitamente desencorajado nos programas em D porque eles vinculam o código a um modelo de dados específico quando usam tipos derivados. Por exemplo, se um `typedef` for redefinido como tipos base de inteiro diferentes, dependendo do modelo de dados, não será possível usar uma única conversão de C que funcione em ambos os modelos de dados sem conhecer explicitamente os dois tipos subjacentes, incluindo uma expressão de conversão ou definindo várias seqüências de formato. O compilador de D resolve esse problema automaticamente, permitindo que você omita os prefixos de tamanho e determine automaticamente o tamanho do argumento.

Os prefixos de tamanho podem ser colocados logo antes do nome de conversão e após quaisquer sinalizadores, larguras e especificadores de precisão. Os prefixos de tamanho são da seguinte forma:

- Um `h` opcional especifica que a seguinte conversão `d`, `i`, `o`, `u`, `x` ou `X` se aplica a um `short` ou `unsigned short`.
- Um `l` opcional especifica que a seguinte conversão `d`, `i`, `o`, `u`, `x` ou `X` se aplica a um `long` ou `unsigned long`.
- Um `ll` opcional especifica que a seguinte conversão `d`, `i`, `o`, `u`, `x` ou `X` se aplica a um `long long` ou `unsigned long long`.
- Um `L` opcional especifica que a seguinte conversão `e`, `E`, `f`, `g` ou `G` se aplica a um `long double`.
- Um `l` opcional especifica que a seguinte conversão `c` se aplica a um argumento `wint_t`, e que o seguinte caractere de conversão `s` se aplica a um ponteiro para um argumento `wchar_t`.

Formatos de conversão

Cada seqüência de caractere de conversão resulta na busca de zero ou mais argumentos. Se forem fornecidos argumentos insuficientes para a seqüência de formato, ou se a seqüência de formato estiver exaurida e os argumentos permanecerem, o compilador de D emitirá uma mensagem de erro apropriada. Se um formato de conversão indefinido for especificado, o compilador de D emite uma mensagem de erro apropriada. As seqüências de caracteres de conversão são:

`a` O ponteiro ou argumento `uintptr_t` é impresso como um nome de símbolo de kernel no formato *módulo*`‘`*nome do símbolo* mais um deslocamento de byte hexadecimal opcional. Se o valor não estiver no intervalo definido por um símbolo de kernel definido, o valor será impresso como um inteiro hexadecimal.

`c` O argumento `char`, `short` ou `int` é impresso como um caractere ASCII.

`C` O argumento `char`, `short` ou `int` será impresso como um caractere ASCII, se o caractere for um caractere ASCII imprimível. Se o caractere não for um caractere imprimível, ele será impresso com a seqüência de escape correspondente, conforme mostrado na Tabela 2–5.

`d` O argumento `char`, `short`, `int`, `long` ou `long long` é impresso como um inteiro decimal (base 10). Se o argumento for `signed`, ele será impresso como um valor assinado. Se o argumento for `unsigned`, ele será impresso como um valor não assinado. Esta conversão possui o mesmo significado que `i`.

`e, E` O argumento `float`, `double` ou `long double` é convertido para o estilo `[-]` *d*`.`*ddd*`e±`*dd*, onde existe um dígito antes do caractere fracionário e o número de dígitos depois dele é igual à precisão. O caractere fracionário será diferente de zero, se o argumento for diferente de zero. Se a precisão não for especificada, o valor de precisão padrão será 6. Se a precisão for 0 e o sinalizador # não estiver especificado, nenhum caractere fracionário aparecerá. O formato de conversão `E` produz um número com `E` em vez de `e` introduzindo o expoente. O expoente sempre contém pelo menos dois dígitos. O valor é arredondado pra o número apropriado de dígitos.

`f` O argumento `float`, `double` ou `long double` é convertido para o estilo `[-]` *ddd*`.`*ddd*, onde número de dígitos após o caractere de precisão é igual à especificação de precisão. Se a precisão não for especificada, o valor de precisão padrão será 6. Se a precisão for 0 e o sinalizador # não estiver especificado, nenhum caractere fracionário aparecerá. Se um caractere fracionário aparecer, pelo menos um dígito aparecerá antes dele. O valor é arredondado pra o número apropriado de dígitos.

`g, G` O argumento `float`, `double` ou `long double` é impresso no estilo `f` ou `e` (ou no estilo `E` no caso de um caractere de conversão `G`), com a precisão especificando o número de dígitos significativos. Se uma precisão explícita for 0, ela é considerada como 1. O estilo usado depende do valor convertido: o estilo `e` (ou `E`) será usado somente se o expoente resultante da conversão for menor que -4 ou maior ou igual à precisão. Zeros

à direita são removidos da parte fracionária do resultado. Um caractere fracionário aparecerá somente se for seguido por um dígito. Se o sinalizador # for especificado, os zeros à direita não serão removidos do resultado.

`i` O argumento `char`, `short`, `int`, `long` ou `long long` é impresso como um inteiro decimal (base 10). Se o argumento for `signed`, ele será impresso como um valor assinado. Se o argumento for `unsigned`, ele será impresso como um valor não assinado. Esta conversão tem o mesmo sentido de `d`.

`o` O argumento `char`, `short`, `int`, `long` ou `long long` é impresso como um inteiro octal não assinado (base 8). Os argumentos que são `signed` ou `unsigned` podem ser usados com esta conversão. Se o sinalizador # for especificado, a precisão do resultado será aumentada, se necessário, para forçar o primeiro dígito do resultado a ser zero.

`p` O ponteiro ou argumento `uintptr_t` é impresso como um inteiro hexadecimal (base 16). D aceita argumentos de ponteiro de qualquer tipo. Se o sinalizador # for especificado, um resultado diferente de zero terá `0x` anteposto a ele.

`s` O argumento deve ser uma matriz de `char` ou `string`. Os bytes da matriz ou `string` são lidos até um caractere nulo de terminação ou o fim dos dados, e interpretados e impressos como caracteres ASCII. Se a precisão não for especificada, ela é considerada como infinita, sendo assim, todos os caracteres até o primeiro caractere nulo serão impressos. Se a precisão for especificada, somente essa parte da matriz de caracteres, que será exibida no número correspondente de colunas da tela, será impressa. Se um argumento do tipo `char *` for formatado, ele deverá ser convertido para `string` ou prefixado com o operador `stringof` de D para indicar que o DTrace deve rastrear os bytes da seqüência e formatá-los.

`S` O argumento deve ser uma matriz de `char` ou `string`. O argumento é processado como se fosse pela conversão `%s`, mas quaisquer caracteres ASCII que não sejam imprimíveis serão substituídos pela seqüência de escape correspondente descrita na Tabela 2–5.

`u` O argumento `char`, `short`, `int`, `long` ou `long long` é impresso como um inteiro decimal não assinado (base 10). Os argumentos que são `signed` ou `unsigned` podem ser usados com esta conversão, e o resultado é sempre formatado como `unsigned`.

`wc` O argumento `int` é convertido em um caractere extenso (`wchar_t`) e o caractere extenso resultante é impresso.

`ws` O argumento deve ser uma matriz de `wchar_t`. Os bytes da matriz ou são lidos até um caractere nulo de terminação ou o fim dos dados, e interpretados e impressos como caracteres extensos. Se a precisão não for especificada, ela é considerada como infinita, sendo assim, todos os caracteres extensos até o primeiro caractere nulo serão impressos. Se a precisão for especificada, somente essa parte da matriz de caracteres extensos, que será exibida no número correspondente de colunas da tela, será impressa.

`x, X` O argumento `char`, `short`, `int`, `long` ou `long long` é impresso como um inteiro hexadecimal não assinado (base 16). Os argumentos que são `signed` ou `unsigned` podem ser usados com esta conversão. Se o formato `x` da conversão for usado, os dígitos de letra `abcdef` serão usados. Se o formato `X` da conversão for usado, os dígitos de letra `ABCDEF` serão usados. Se o sinalizador `#` for especificado, um resultado diferente de zero terá `0x` (para `%x`) ou `0X` (para `%X`) anteposto a ele.

`Y` O argumento `uint64_t` será interpretado como o número de nanossegundos desde 00:00 Horário Coordenado Universal, 1 de janeiro de 1970, e será impresso no seguinte formato `cftime(3C)` form: "`%Y %a %b %e %T %Z`". O número de nanossegundos desde 00:00 UTC, 1 de janeiro, 1970, está disponível na variável `walltimestamp`.

`%` Imprime um caractere literal `%`. Nenhum argumento é convertido. A especificação de conversão inteira deve ser `%%`.

printa()

A função `printa()` é usada para formatar os resultados de agregações em um programa em D. A função é chamada de uma de duas formas:

```
printa(@aggregation-name);
printa(format-string, @aggregation-name);
```

Se o primeiro formato da função for usado, o comando `dtrace(1M)` tirará um instantâneo consistente dos dados da agregação e produzirá uma saída equivalente ao formato de saída padrão usado para agregações, descrito no Capítulo 9, "Agregações".

Se o segundo formato da função for usado, o comando `dtrace(1M)` tirará um instantâneo consistente dos dados da agregação e produzirá uma saída de acordo com as conversões especificadas na *seqüência de formato*, de acordo com as seguintes regras.

- As conversões de formato devem corresponder à assinatura de tupla usada para criar a agregação. Cada elemento de tupla só pode aparecer uma vez. Por exemplo, se você agregar uma contagem usando as declarações de D seguintes:

  ```
  @a["hello", 123] = count();
  @a["goodbye", 456] = count();
  ```

  e, em seguida, adicionar a declaração de D `printa(`*seqüência de formato*`, @a)` a uma cláusula de teste, `dtrace` tirará um instantâneo dos dados da agregação e produzirá uma saída, se você tiver inserido as declarações:

  ```
  printf(format-string, "hello", 123);
  printf(format-string, "goodbye", 456);
  ```

e assim por diante para cada tupla definida na agregação.

- Ao contrário de `printf()`, a seqüência de formato usada para `printa()` não precisa incluir todos os elementos da tupla. Ou seja, você pode ter uma tupla de tamanho 3 e apenas uma conversão de formato. Portanto, você pode omitir quaisquer chaves de tupla da saída de `printa()`, alterando a declaração de agregação para mover as chaves que você deseja omitir para o fim da tupla e, em seguida, omitir os especificadores de conversão correspondentes delas na seqüência de formato `printa()`.
- O resultado da agregação pode ser incluído na saída, usando o caractere de sinalizador de formato adicional `@`, que só é válido quando usado com `printa()`. O sinalizador `@` pode ser combinado com qualquer especificador de conversão de formato apropriado, e pode aparecer mais de uma vez em uma seqüência de formato, para que o resultado da tupla possa aparecer em qualquer local na saída e possa aparecer mais de uma vez. O conjunto de especificadores de conversão que podem ser usados com cada função de agregação são deduzidos pelo tipo de resultado da função de agregação. Os tipos de resultado de agregação são:

`avg()`	`uint64_t`
`count()`	`uint64_t`
`lquantize()`	`int64_t`
`max()`	`uint64_t`
`min()`	`uint64_t`
`quantize()`	`int64_t`
`sum()`	`uint64_t`

  Por exemplo, para formatar os resultados de `avg()`, você pode aplicar as conversões de formato `%d`, `%i`, `%o`, `%u` ou `%x`. As funções `quantize()` e `lquantize()` formatam seus resultados como uma tabela ASCII em vez de um único valor.

O programa em D seguinte mostra um exemplo completo de `printa()`, usando o provedor de perfil `profile` para fazer uma amostra do valor de `caller` e, em seguida, formatando os resultados como uma tabela simples:

```
profile:::profile-997
{
    @a[caller] = count();
}

END
{
```

```
    printa("%@8u %a\n", @a);
}
```

Se você usar o dtrace para executar este programa, espere alguns segundos, e pressione Control-C, para ver uma saída semelhante ao exemplo seguinte:

```
# 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
       ...
```

Formato padrão de trace()

Se a função trace() for usada para capturar os dados em vez de printf(), o comando dtrace formatará os resultados usando um formato de saída padrão. Se os dados tiverem 1, 2, 4, ou 8 bytes de tamanho, o resultado será formatado como um valor inteiro decimal. Se os dados forem de qualquer outro tamanho e forem uma seqüência de caracteres imprimíveis, se interpretados como uma seqüência de bytes, eles serão impressos como uma seqüência ASCII. Se os dados forem de qualquer outro tamanho e não uma seqüência de caracteres imprimíveis, eles serão impressos como uma série de valores de byte formatados como inteiros hexadecimais.

◆ ◆ ◆ CAPÍTULO 13

13 Rastreio especulativo

Este capítulo discute o recurso do DTrace de *rastreio especulativo*, a capacidade de rastrear tentativamente os dados e, em seguida, decidir entre *comprometer* os dados em um buffer de rastreio ou *eliminá-los*. No DTrace, o mecanismo principal para filtrar eventos desinteressantes é o mecanismo de *predicado*, discutido no Capítulo 4, "Estrutura de programa em D". Os predicados são úteis quando você sabe no momento que um teste é acionado, se o evento do teste é ou não interessante. Por exemplo, se estiver interessado somente na atividade associada a um determinado processo ou a determinado descritor de arquivo, você sabe quando o teste é acionado, caso esteja associado ao processo ou descritor de arquivo de interesse. Entretanto, em outras situações, talvez você não saiba se um determinado evento de teste é de seu interesse até algum tempo *depois* que o teste é acionado.

Por exemplo, se uma chamada do sistema estiver falhando ocasionalmente com um código de erro comum (por exemplo, `EIO` ou `EINVAL`), talvez você queira examinar o caminho do código que está causando a condição de erro. Para capturar o caminho do código, você poderia ativar cada teste — mas apenas se a chamada que está falhando puder ser isolada de forma que um predicado consistente possa ser construído. Se as falhas forem esporádicas ou não determinísticas, você seria forçado a rastrear todos os eventos que *devem* ser interessantes, e mais tarde pós-processar os dados para filtrar aqueles que não estavam associados ao caminho de código que estava falhando. Neste caso, embora o número de eventos interessantes possa ser razoavelmente pequeno, o número de eventos que devem ser rastreados é muito grande, dificultando o pós-processamento.

Você pode usar o recurso de rastreio especulativo nessas situações para tentativamente rastrear os dados em um ou mais locais de teste e, em seguida, decidir entre comprometer os dados no buffer principal em outro local de teste. Como resultado, seus dados de rastreio contêm somente a saída interessante, nenhum pós-processamento é necessário, e a sobrecarga do DTrace é minimizada.

Interfaces de especulação

A tabela seguinte descreve as funções de especulação do DTrace:

TABELA 13–1 Funções de especulação do DTrace

Nome da função	Args	Descrição
`speculation`	Nenhum	Retorna um identificador de um novo buffer especulativo
`speculate`	ID	Significa que o restante da cláusula deve ser rastreado para o buffer especulativo especificado pelo ID
`commit`	ID	Compromete o buffer especulativo associado ao ID
`discard`	ID	Descarta o buffer especulativo associado ao ID

Criando uma especulação

A função `speculation()` aloca um buffer especulativo e retorna um identificador de especulação. O identificador de especulação deve ser usado em chamadas subseqüentes à função `speculate. ()` Os buffers especulativos são um recurso finito: Se nenhum buffer especulativo estiver disponível quando `speculation()` for chamada, um ID zero será retornado e um contador de erro do DTrace correspondente será incrementado. Um ID zero é sempre inválido, mas pode ser passado para `speculate()`, `commit()` ou `discard()`. Se uma chamada a `speculation()` falhar, uma mensagem do `dtrace` semelhante ao exemplo seguinte será gerada:

```
dtrace: 2 failed speculations (no speculative buffer space available)
```

O número de buffers especulativos é um como padrão, mas pode ser opcionalmente ajustado para mais. Consulte "Opções e ajuste de especulação" na página 179 para obter mais informações.

Usando uma especulação

Para usar uma especulação, um identificador retornado de `speculation ()` deve ser passado para a função `speculate()` em uma cláusula *antes* de quaisquer ações de gravação de dados. Todas as ações de gravação de dados subseqüentes em uma cláusula contendo `speculate()` serão rastreadas especulativamente. O compilador de D gerará um erro de tempo de compilação, se uma chamada a `speculate()` seguir as ações de gravação de dados em uma cláusula de teste de D. Portanto, as cláusulas podem conter rastreio especulativo ou solicitações de rastreio não especulativo, mas não ambos.

As ações de agregação, as ações destrutivas e a ação `exit` nunca podem ser especulativas. Qualquer tentativa de usar uma dessas ações em uma cláusula contendo os resultados de

`speculate()` causará um erro de tempo de compilação. Uma `speculate()` não pode seguir uma `speculate()`: somente uma especulação é permitida por cláusula. Uma cláusula que contém *somente* uma `speculate()` rastreará especulativamente a ação padrão, que é definida para rastrear somente o ID de teste ativado. Consulte o Capítulo 10, "Ações e sub-rotinas" para obter uma descrição da ação padrão.

Geralmente, você atribui o resultado de `speculation()` a uma variável de segmento local e, em seguida, usa essa variável como um predicado subseqüente para outros testes, assim como um argumento `speculate()`. Por exemplo:

```
syscall::open:entry
{
    self->spec = speculation();
}

syscall:::
/self->spec/
{
    speculate(self->spec);
    printf("this is speculative");
}
```

Comprometendo uma especulação

Você compromete as especulações usando a função `commit. ()` Quando um buffer especulativo é comprometido, seus dados são copiados para o buffer principal. Se houver mais dados no buffer especulativo especificado do que o espaço disponível no buffer principal, nenhum dado será copiado e a contagem de soltar do buffer será incrementada. Se o buffer tiver sido rastreado especulativamente para mais de uma CPU, os dados especulativos da CPU comprometida serão copiados imediatamente, enquanto os dados especulativos em outras CPUs serão copiados algum tempo depois de `commit()`. Portanto, algum tempo pode passar entre uma `commit()` iniciando em uma CPU e os dados sendo copiados dos buffers especulativos para os buffers principais em todas as CPUs. Esse tempo não pode ser mais longo do que o tempo determinado pela taxa de limpeza. Consulte "Opções e ajuste de especulação" na página 179 para obter mais detalhes.

Um buffer especulativo de comprometimento pode não estar disponível para chamadas de `speculation()` subseqüentes, até que cada buffer especulativo por CPU tenha sido completamente copiado para seu buffer principal por CPU correspondente. Semelhantemente, as chamadas subseqüentes a `speculate()` para o buffer de comprometimento serão silenciosamente descartadas, e as chamadas subseqüentes para `commit()` ou `discard()` falharão silenciosamente. Finalmente, uma cláusula contendo uma `commit()` não pode conter uma ação de gravação de dados, mas uma cláusula pode conter várias chamadas `commit()` para comprometer os buffers de deslocamento.

Descartando uma especulação

Você descarta as especulações usando a função `discard()` . Quando um buffer especulativo é descartado, seu conteúdo é jogado fora. Se a especulação estiver ativa apenas na CPU que chama `discard()`, o buffer será imediatamente disponibilizado para chamadas subseqüentes a `speculation ()`. Se a especulação estiver ativa em mais de uma CPU, o buffer descartado será disponibilizado para a `speculation()` subseqüente algum tempo depois da chamada a `discard()`. O tempo entre uma `discard()` em uma CPU e o buffer que está sendo disponibilizado para especulações subseqüentes é garantidamente mais curto que o tempo definido pela taxa de limpeza. Se, na hora em que `speculation ()` for chamada, não houver nenhum buffer disponível porque *todos* os buffers especulativos estão sendo descartados no momento ou comprometidos, será gerada uma mensagem do `dtrace` semelhante ao exemplo seguinte:

```
dtrace: 905 failed speculations (available buffer(s) still busy)
```

A probabilidade de todos os buffers estarem indisponíveis pode ser reduzida, através do ajuste do número de buffers de especulação ou a taxa de limpeza. Consulte "Opções e ajuste de especulação" na página 179 para obter detalhes.

Exemplo de especulação

Um uso potencial de especulações é realçar um caminho de código particular. O exemplo seguinte mostra o caminho de código inteiro sob a chamada do sistema a open(2) somente quando `open()` falha:

EXEMPLO 13–1 `specopen.d`: fluxo de código para open(2) falha

```
#!/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
```

EXEMPLO 13–1 specopen.d: fluxo de código para open(2) falha *(Continuação)*

```
     * speculation is subsequently commited.
     */
    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.
```

EXEMPLO 13–1 `specopen.d`: fluxo de código para `open(2)` falha *(Continuação)*

```
     */
    discard(self->spec);
    self->spec = 0;
}
```

Executar o script acima produzirá uma saída semelhante ao exemplo seguinte:

```
# ./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
```

Opções e ajuste de especulação

Se um buffer especulativo estiver cheio quando uma ação de rastreio especulativo for tentada, nenhum dado será armazenado no buffer e uma contagem de soltar será incrementada. Caso isso ocorra, uma mensagem do `dtrace` semelhante ao exemplo seguinte será gerada:

```
dtrace: 38 speculative drops
```

As solturas especulativas *não* evitarão que o buffer especulativo cheio seja copiado para o buffer principal quando o buffer for comprometido. Semelhantemente, as solturas especulativas poderão ocorrer, mesmo se houver tido solturas em um buffer especulativo que foi finalmente descartado. As solturas especulativas podem ser reduzidas, através do aumento do tamanho do buffer especulativo, que é ajustado através da opção `specsize` . A opção `specsize` pode ser especificada com qualquer sufixo de tamanho. A política de redimensionamento deste buffer é definida pela opção `bufresize` .

Os buffers especulativos talvez estejam indisponíveis quando `speculation ()` for chamada. Se houver buffers que ainda não foram comprometidos ou descartados, uma mensagem do `dtrace` semelhante ao exemplo seguinte será gerada:

```
dtrace: 1 failed speculation (no speculative buffer available)
```

Você pode reduzir a probabilidade de especulações falhas desta natureza, aumentando o número de buffers especulativo com a opção `nspec` . O valor padrão de `nspec` é um.

Como alternativa, `speculation()` pode falhar porque todos os buffers especulativos estão ocupados. Neste caso, uma mensagem do `dtrace` semelhante ao exemplo seguinte será gerada:

```
dtrace: 1 failed speculation (available buffer(s) still busy)
```

Esta mensagem indica que `speculation()` foi chamada depois que `commit()` foi chamada por um buffer especulativo, mas antes que o buffer fosse realmente comprometido em todas as CPUs. Você pode reduzir a probabilidade de especulações falhas desta natureza, aumentando a taxa na qual as CPUs são limpas com a opção `cleanrate.` O valor padrão de `cleanrate` é`101hz`.

Observação – Os valores da opção `cleanrate` devem ser especificados em número por segundo. Use o sufixo `hz`.

◆ ◆ ◆ CAPÍTULO 14

Utilitário dtrace(1M)

O comando dtrace(1M) é um front-end genérico para o recurso DTrace. O comando implementa uma única interface simples para chamar o compilador da linguagem D, a capacidade de recuperar dados de rastreio armazenados em buffer a partir do recurso de kernel do DTrace, e um conjunto de rotinas básicas para formatar e imprimir dados rastreados. Este capítulo fornece uma referência completa ao comando dtrace.

Descrição

O comando dtrace oferece uma interface genérica para todos os serviços essenciais fornecidos pelo recurso DTrace, incluindo:

- Opções para listar o conjunto de testes e provedores publicados atualmente pelo DTrace
- Opções para ativar testes diretamente usando qualquer um dos especificadores de descrição de teste (provedor, módulo, função, nome)
- Opções para executar o compilador de D e compilar um ou mais arquivos de programa de D escritos diretamente na linha de comando
- Opções para gerar programas de rastreio anônimos (consulte o Capítulo 36, "Rastreio anônimo")
- Opções para gerar relatórios de estabilidade do programa (consulte o Capítulo 39, "Estabilidade")
- Opções para modificar o comportamento de rastreio e armazenamento em buffer do DTrace, e ativar arquivos de compilador de D adicionais (consulte o Capítulo 16, "Opções e ajustáveis")

O dtrace também pode ser usado para criar scripts de D ao ser usado em uma declaração #! para criar um arquivo de intérprete (consulte o Capítulo 15, "Script"). Finalmente, você pode usar o dtrace para tentar compilar programas em D e determinar suas propriedades sem realmente ativar qualquer rastreamento com a opção -e, descrita abaixo.

Opções

O comando `dtrace` aceita as seguintes opções:

`dtrace` `[-32 | -64] [-aACeFGHlqSvVwZ] [-b` *bufsz*`] [-c` *cmd*`] [-D` *nome* `[=`*def*`]] [-I` *caminho*`] [-L` *caminho*`] [-o` *saída*`] [-p` *pid*`] [-s` *script*`] [-U` *nome*`] [-x` *arg* `[=`*val*`]] [-Xa | f | s | t] [-P` *provedor* `[ [`*predicado*`]`*ação*`]] [-m [ [`*provedor:*`]`*módulo* `[ [`*predicado*`]`*ação*`]]] [-f [ [`*provedor:*`]`*módulo:*`]`*func* `[ [`*predicado*`]`*ação*`]] [-n [ [ [`*provedor:*`]`*módulo:*`]`*func:*`]`*nome* `[ [`*predicado*`]`*ação*`]] [-i` *id de teste* `[ [`*predicado*`]`*ação*`]]`

onde *predicado* é qualquer predicado de D entre barras `/ /` e *ação* é qualquer lista de declarações de D entre chaves `{ }`, de acordo com a sintaxe da linguagem D descrita anteriormente. Se o código do programa em D for fornecido como um argumento para as opções `-P`, `-m`, `-f`, `- n` ou `-i`, este texto deve ser colocado entre aspas apropriadamente para evitar interpretação pelo shell. As opções são as seguintes:

`-32`, `-64` O compilador de D produz programas usando o modelo de dados nativo do kernel do sistema operacional. Você pode usar o comando `isainfo(1)` `-b` para determinar o modelo de dados atual do sistema operacional. Se a opção `- 32` for especificada, o `dtrace` forçará o compilador de D a compilar um programa em D usando o modelo de dados de 32 bits. Se a opção `- 64` for especificada, o `dtrace` forçará o compilador de D a compilar um programa em D usando o modelo de dados de 64 bits. Estas opções geralmente não são necessárias, quando o `dtrace` seleciona o modelo de dados nativo como o padrão. O modelo de dados afeta o tamanho dos tipos de inteiro e de outras propriedades de outras linguagens. Os programas em D compilados para qualquer modelo de dados pode ser executado em kernels de 32 bits e de 64 bits. As opções `-32` e `-64` também determinam o formato de arquivo ELF (ELF32 ou ELF64) produzido pela opção `-G`.

`-a` Declara o estado de rastreio anônimo e exibe os dados rastreados. Você pode combinar a opção `-a` com a opção `-e` para forçar o `dtrace` a sair imediatamente após consumir o estado de rastreio anônimo em vez de continuar a esperar por novos dados. Consulte o Capítulo 36, “Rastreio anônimo” para obter mais informações sobre rastreio anônimo.

`-A` Gera diretivas de `driver.conf(4)` para rastreio anônimo. Se a opção `-A` for especificada, o `dtrace` compila quaisquer programas em D especificados usando a opção `-s` ou na linha de comando, e constrói um conjunto de diretivas de arquivo de configuração de `dtrace(7D)` para ativar os testes especificados para rastreio anônimo (consulte o Capítulo 36, “Rastreio anônimo”) e, a seguir, sai. Por padrão, o `dtrace` tenta armazenar as diretivas no arquivo `/kernel/drv/dtrace.conf`. Este comportamento pode ser modificado com a opção `-o` para especificar um arquivo de saída alternativo.

`-b` Define o tamanho de buffer de rastreio principal. O tamanho de buffer de rastreio pode incluir quaisquer sufixos de tamanho `k`, `m`, `g` ou `t`, conforme descrito no Capítulo 36, "Rastreio anônimo". Se o espaço do buffer não puder ser alocado, o `dtrace` tentará reduzir o tamanho do buffer ou sairá, dependendo da configuração da propriedade `bufresize`.

`-c` Executa o comando *cmd* especificado e sai após a sua conclusão. Se mais de uma opção `-c` estiver presente na linha compilada, o `dtrace` sairá quando todos os comandos tiverem saído, informando o status de saída de cada processo filho, quando ele termina. O ID do processo do primeiro comando é disponibilizado para quaisquer programas em D especificados na linha de comando ou com a opção `-s` através da variável de macro `$target`. Consulte o Capítulo 15, "Script" para obter mais informações sobre variáveis de macro.

`-C` Executa o pré-processador de C `cpp(1)` sobre programas em D antes de compilá-los. As opções podem ser passadas para o pré-processador de C usando as opções `-D`, `-U`, `-I` e `-H`. O grau de conformidade padrão de C pode ser selecionado com a opção `-X`. Consulte a descrição da opção `-X` para obter informações sobre o conjunto de símbolos definidos pelo compilador de D ao chamar o pré-processador de C.

`-D` Define o *nome* especificado ao chamar `cpp(1)` (ativado com a opção `-C`). Se um sinal de igual (`=`) e um *valor* adicional forem especificados, o valor correspondente será atribuído ao nome. Esta opção passa a opção `-D` para cada chamada de `cpp`.

`-e` Sai depois de compilar quaisquer solicitações e consumir o estado de rastreio anônimo (opção `-a`) mas antes de ativar quaisquer testes. Esta opção pode ser combinada com a opção `-a` para imprimir dados de rastreio anônimo e sair, ou ela pode ser combinada com as opções do compilador de D para verificar se os programas compilam sem realmente serem executados e ativar a instrumentação correspondente.

`-f` Especifica o nome da função para rastrear ou listar (opção `-l`). O argumento correspondente pode incluir quaisquer formatos de descrição de teste *provedor:módulo:função*, *módulo:função* ou *função*. Os campos de descrição de teste não especificados são deixados em branco e correspondem a quaisquer testes, independentemente dos valores desses campos. Se nenhum qualificador diferente da *função* for especificado na descrição, todos os testes com a *função* correspondente serão correspondidos. O argumento `-f` pode ser sufixado com uma cláusula de teste de D opcional. Mais de uma opção `-f` pode ser especificada na linha de comando de cada vez.

`-F` Adere a saída do rastreio identificando a entrada e o retorno da função. Os relatórios do teste de entrada da função são recuados e sua saída é prefixada com `->`. Os relatórios de teste de retorno da função têm o recuo cancelado e sua saída é prefixada com `<-`.

`-G` Gera um arquivo ELF contendo um programa do DTrace incorporado. Os testes do DTrace especificados no programa são salvos dentro de um objeto ELF realocável que pode ser vinculado em outro programa. Se a opção `-o` estiver presente, o arquivo ELF será salvo com o nome de caminho especificado como o argumento deste operando. Se a opção `-o` não estiver presente e o programa do DTrace estiver em um arquivo cujo nome é *nome do arquivo*`.s`, então o arquivo ELF será salvo com o nome *arquivo*`.o`. Caso contrário, o arquivo ELF é salvo com o nome `d.out`.

`-H` Imprime os nomes de caminho de arquivos incluídos ao chamar `cpp(1)` (ativado com a opção `-C`). Esta opção passa a opção `-H` para cada chamada de `cpp`, fazendo com que seja exibida a lista de nomes de caminho, um por linha, para `stderr`.

`-i` Especifica o identificador de teste para rastrear ou listar (opção `-l`). Os IDs de teste são especificados com inteiros decimais, conforme mostrado por `dtrace -l`. O argumento `-i` pode ser sufixado com uma cláusula de teste de D opcional. Mais de uma opção `-i` pode ser especificada na linha de comando de cada vez.

`-I` Adiciona o *caminho* do diretório especificado ao caminho de pesquisa dos arquivos `#include` ao chamar `cpp(1)` (ativado com a opção `-C`). Esta opção passa a opção `-I` para cada chamada de `cpp`. O diretório especificado é inserido no caminho de pesquisa antes da lista de diretórios padrão.

`-l` Lista os testes em vez de ativá-los. Se a opção `-l` for especificada, o `dtrace` produzirá um relatório dos testes que correspondem às descrições fornecidas com as opções `-P`, `-m`, `-f`, `-n`, `-i` e `-s`. Se nenhuma destas opções forem especificadas, todos os testes serão listados.

`-L` Adiciona o *caminho* do diretório especificado ao caminho de pesquisa das bibliotecas do DTrace. As bibliotecas do DTrace são usadas para armazenar definições comuns que podem ser usadas quando são escritos os programas em D. O *caminho* especificado é adicionado depois do caminho de pesquisa de biblioteca padrão.

`-m` Especifica o nome do módulo a ser rastreado ou listado (opção `-l`). O argumento correspondente pode incluir quaisquer formatos de descrição de teste *provedor:módulo* ou *módulo*. Os campos de descrição de teste não especificados são deixados em branco e correspondem a quaisquer testes, independentemente dos valores desses campos. Se nenhum qualificador diferente do *módulo* for especificado na descrição, todos os testes com o *módulo* correspondente serão correspondidos. O argumento `-m` pode ser sufixado com uma cláusula de teste de D opcional. Mais de uma opção `-m` pode ser especificada na linha de comando de cada vez.

`-n` Especifica o nome do teste para rastrear ou listar (opção `-l`). O argumento correspondente pode incluir qualquer um dos formatos de descrição de teste *provedor:módulo:função:nome*, *módulo:função:nome* , *função:nome* ou *nome*. Os campos de descrição de teste não especificados são deixados em branco e

correspondem a quaisquer testes, independentemente dos valores desses campos. Se nenhum qualificador diferente do *nome* for especificado na descrição, todos os testes com um *nome* correspondente serão correspondidos. O argumento `-n` pode ser sufixado com uma cláusula de teste de D opcional. Mais de uma opção `-n` pode ser especificada na linha de comando de cada vez.

`-o` Especifica o arquivo de *saída* das opções `-A`, `-G` e `-l`, ou dos dados rastreados. Se a opção `-A` estiver presente e `-o` não estiver presente, o arquivo de saída padrão será `/kernel/drv/dtrace.conf`. Se a opção `-G` estiver presente e o argumento da opção `-s` for do formato *nome de arquivo* `.d` e `-o` não estiver presente, o arquivo de saída padrão será *nome de arquivo* `.o`. Caso contrário, o arquivo de saída padrão será `d.out`.

`-p` Obtém o ID de processo especificado *pid*, armazena em cache suas tabelas de símbolos e sai após a sua conclusão. Se mais de uma opção `-p` estiver presente na linha de comando, o `dtrace` sairá quando todos os comandos tiverem saído, informando o status de saída de cada processo quando ele termina. O primeiro ID do processo é disponibilizado para quaisquer programas em D especificados na linha de comando ou com a opção `-s` através da variável de macro `$target`. Consulte o Capítulo 15, "Script" para obter mais informações sobre variáveis de macro.

`-P` Especifica o nome do provedor para rastrear ou listar (opção `-l`). O nome, a função e o módulo dos campos de descrição de teste restantes são deixados em branco e correspondem a quaisquer testes, independentemente dos valores desses campos. O argumento `-P` pode ser sufixado com uma cláusula de teste de D opcional. Mais de uma opção `-P` pode ser especificada na linha de comando de cada vez.

`-q` Define o modo silencioso. O `dtrace` eliminará mensagens tais como o número de testes correspondidos pelas opções especificadas e os programas em D não imprimirão os cabeçalhos de coluna, o ID da CPU, o ID do teste, ou inserirão linhas novas na saída. Somente dados rastreados e formatados por declarações de programas em D, tais como `trace()` e `printf()`, serão exibidos para `stdout`.

`-s` Compila o arquivo-fonte do programa em D especificado. Se a opção `-e` não estiver presente, o programa será compilado, mas nenhuma instrumentação será ativada. Se a opção `-l` estiver presente, o programa será compilado e o conjunto de testes correspondidos por ele será lista, mas nenhuma instrumentação será ativada. Se nem `-e` ou `-l` estiver presente, a instrumentação especificada pelo programa em D será ativada e o rastreio começa.

`-S` Mostra o código intermediário do compilador de D. O compilador de D produzirá um relatório do código intermediário gerado para cada programa em D para `stderr`.

`-U` Indefine o *nome* especificado ao chamar `cpp(1)` (ativado com a opção `-C`). Esta opção passa a opção `-U` para cada chamada de `cpp`.

`-v` Define o modo verboso. Se a opção `-v` for especificada, o `dtrace` produzirá um relatório de estabilidade do programa mostrando a estabilidade mínima da interface e o nível de dependência dos programas em D especificados. Os níveis de estabilidade do DTrace são explicados em maiores detalhes no Capítulo 39, "Estabilidade".

`-V` Informa a versão da interface de programação em D mais alta aceita pelo `dtrace`. As informações de versão são impressas no `stdout` e o comando `dtrace` sai. Consulte o Capítulo 41, "Versionamento" para obter mais informações sobre os recursos de versão do DTrace.

`-w` Permite ações destrutivas nos programas em D especificadas com as opções `-s`, `-P`, `-m`, `-f`, `-n` ou `-i`. Se a opção `-w` não for especificada, o `dtrace` não permitirá a compilação ou ativação de um programa em D que contenha ações destrutivas. As ações destrutivas são descritas em maiores detalhes no Capítulo 10, "Ações e sub-rotinas".

`-x` Ativa ou modifica uma opção de tempo de execução do DTrace ou a opção do compilador de D. As opções estão listadas no Capítulo 16, "Opções e ajustáveis". As opções booleanas são ativadas através da especificação de seus nomes. As opções com valores são definidas, separando o nome e o valor da opção com um sinal de igual (`=`).

`-X` Especifica o grau de conformidade com o padrão ISO C que deve ser selecionado ao chamar `cpp(1)` (ativado com a opção `-C`). O argumento da opção `-X` afeta o valor e a presença da macro `__STDC__`, dependendo do valor da letra do argumento:

`a` (padrão)	As extensões de compatibilidade ISO C mais K&R, com alterações semânticas requeridas pela ISO C. Este modo será o padrão, se `-X` não for especificada. A macro predefinida `__STDC__` possui um valor 0 quando `cpp` é chamado junto com a opção `-Xa`.
`c` (conformidade)	Conforme estritamente com ISO C, sem extensões de compatibilidade K&R C. A macro predefinida `__STDC__` possui um valor 1 quando `cpp` é chamado junto com a opção `-Xa`.
`s` (K&R C)	K&R C somente. A macro `__STDC__` não é definida quando `cpp` é chamado junto com a opção `-Xs`.
`t` (transição)	ISO C mais extensões de compatibilidade K&R C, sem alterações semânticas requeridas pela ISO C. A macro predefinida `__STDC__` possui um valor 0 quando `cpp` é chamada em conjunto com a opção `- Xt`.

Como a opção `-X` afeta somente como o compilador de D chama o pré-processador de C, as opções `-Xa` e `-Xt` são equivalentes da perspectiva de D. Ambas as opções são fornecidas para fácil reutilização das configurações de um ambiente de construção de C.

Independentemente do modo `-X`, as definições do pré-processador de C adicionais são sempre especificadas e válidas em todos os modos:

- `__sun`
- `__unix`
- `__SVR4`
- `__sparc` (somente em sistemas SPARC®)
- `__sparcv9` (somente em sistemas SPARC® quando programas de 64–bits são compilados)
- `__i386` (somente em sistemas x86 quando programas de 32 bits são compilados)
- `__amd64` (somente em sistemas x86 quando programas de 64–bits são compilados)
- *__'uname -s'_'uname -r'*, substituindo o ponto decimal na saída de `uname` por um sublinhado (_), como em `__SunOS_5_10`
- `__SUNW_D=1`
- `__SUNW_D_VERSION=0x`*MMmmmuuu* (onde *MM* é o valor de versão Principal em hexadecimal, *mmm* é o valor de versão Secundário em hexadecimal, e *uuu* é o valor de versão Micro em hexadecimal. Consulte o Capítulo 41, "Versionamento" para obter mais informações sobre a versão do DTrace)

`-Z` Permite as descrições de teste que correspondem a zero teste. Se a opção `-Z` não for especificada, o `dtrace` informará um erro e sairá, se nenhuma descrição de teste especificada em arquivos de programa em D (opção `-s`) ou na linha de comando (opções `-P`, `-m`, `-f`, `-n` ou `-i`) contiver descrições que não correspondam a quaisquer testes conhecidos.

Operandos

Zero ou mais argumentos adicionais podem ser especificados na linha de comando do `dtrace` para definir um conjunto de variáveis de macro (`$1`, `$2`, e assim por diante) a serem usadas em quaisquer programas em D especificados com a opção `-s` ou na linha de comando. O uso de variáveis de macro é descrito em mais detalhes no Capítulo 15, "Script".

Status de saída

Os valores de saída seguintes são retornados pelo utilitário `dtrace`:

0 As solicitações especificadas foram concluídas com êxito. Para solicitações de programas em D, o status de saída 0 indica que os programas foram compilados com êxito, os testes foram ativados com êxito ou o estado anônimo foi recuperado com êxito. `dtrace` retorna 0 mesmo se as solicitações de rastreio especificadas encontraram erros ou solturas.

1 Ocorreu um erro fatal. Para solicitações de programa em D, o status de saída 1 indica que a compilação do programa falhou ou que a solicitação especificada não foi satisfeita.

2 Opções de linha de comando inválidas ou argumentos não especificados.

◆ ◆ ◆ CAPÍTULO 15

15 Script

Você pode usar o utilitário dtrace(1M) para criar arquivos do intérprete a partir de programas em D semelhantes para criar shell de scripts que você pode instalar como ferramentas interativas reutilizáveis do DTrace. O compilador de D e o comando `dtrace` oferecem um conjunto de *variáveis de macro* expandidas pelo compilador de D que facilitam a criação de scripts. Este capítulo fornece uma referência do recurso de variável de macro e dicas para a criação de scripts persistentes.

Arquivos de intérprete

Semelhante ao shell e aos utilitários, tais como awk(1) e perl(1), o dtrace(1M) pode ser usado para criar arquivos de intérprete executáveis. Um arquivo de intérprete começa com uma linha do formato:

`#!` *nome do caminho arg*

onde *nome do caminho* é o caminho do intérprete e *arg* é um único argumento opcional. Quando um arquivo de intérprete é executado, o sistema chama o intérprete especificado. Se *arg* foi especificado no arquivo do intérprete, ele é passado como um argumento para o intérprete. O caminho para o próprio arquivo do intérprete e quaisquer argumentos adicionais especificados quando ele foi executado são, em seguida, adicionados à lista de argumentos do intérprete. Portanto, você sempre precisará criar arquivos de intérprete do DTrace com pelo menos estes argumentos:

```
#!/usr/sbin/dtrace -s
```

Quando o seu arquivo de intérprete é executado, o argumento para a opção `-s` será, portanto, o nome do caminho do próprio arquivo do intérprete. `dtrace` irá, em seguida, ler, compilar e executar este arquivo, como se você tivesse digitado o seguinte comando no shell:

```
# dtrace -s interpreter-file
```

O exemplo seguinte mostra como criar e executar um arquivo de intérprete do `dtrace` . Digite o seguinte código-fonte de D e salve-o em um arquivo chamado `interp.d`:

```
#!/usr/sbin/dtrace -s
BEGIN
{
    trace("hello");
    exit(0);
}
```

Marque o arquivo `interp.d` como executável e execute-o da seguinte forma:

```
# chmod a+rx interp.d
# ./interp.d
dtrace: script ’./interp.d’ matched 1 probe
CPU     ID                    FUNCTION:NAME
  1      1                           :BEGIN   hello
#
```

Lembre-se que a diretiva `#!` deve compreender os dois primeiros caracteres do seu arquivo sem espaço em branco precedente ou intermediário. O compilador de D sabe como ignorar automaticamente esta linha quando processa o arquivo de intérprete.

O `dtrace` usa `getopt(3C)` para processar opções da linha de comando, para que você possa combinar várias opções em seu único argumento de intérprete. Por exemplo, para adicionar a opção `-q` ao exemplo anterior, você poderia alterar a diretiva do intérprete para:

```
#!/usr/sbin/dtrace -qs
```

Se você especificar letras de várias opções, a opção `-s` deve sempre terminar a lista de opções booleanas, para que o próximo argumento (o nome do arquivo do intérprete) seja processado como o argumento correspondente à opção `-s`.

Se você precisar especificar mais de uma opção que requeira um argumento em seu arquivo de intérprete, suas opções e argumentos não caberão em um único argumento de intérprete. Em vez disso, use a sintaxe de diretiva da `opção #pragma de D` para definir suas opções. Todas as opções da linha de comando do `dtrace` possuem `#pragma` equivalentes que você pode usar, conforme mostrado no Capítulo 16, “Opções e ajustáveis”.

Variáveis de macro

O compilador de D define um conjunto de variáveis de macro incorporadas que você pode usar ao escrever programas em D ou arquivos de intérprete. As variáveis de macro são identificadores prefixados com um sinal de dólar (`$`) e são expandidos apenas pelo compilador de D ao processar o seu arquivo de entrada. O compilador de D fornece as seguintes variáveis de macro:

TABELA 15–1 Variáveis de macro de D

Nome	Descrição	Referência
`$[0-9]+`	argumentos de macro	Consulte "Argumentos de macro" na página 192
`$egid`	ID de grupo efetivo	`getegid(2)`
`$euid`	ID de usuário efetivo	`geteuid(2)`
`$gid`	ID de grupo real	`getgid(2)`
`$pid`	ID do processo	`getpid(2)`
`$pgid`	ID de grupo de processo	`getpgid(2)`
`$ppid`	ID de processo-pai	`getppid(2)`
`$projid`	ID do projeto	`getprojid(2)`
`$sid`	ID de sessão	`getsid(2)`
`$target`	ID do processo de destino	Consulte "ID do processo de destino" na página 194
`$taskid`	ID de tarefa	`gettaskid(2)`
`$uid`	ID de usuário real	`getuid(2)`

Exceto para os argumentos de macro `$[0-9]+` e a variável de macro `$target`, todas as variáveis de macro se expandem para inteiros correspondentes aos atributos do sistema, tais como ID do processo e ID do usuário. As variáveis se expandem para o valor de atributo associado ao próprio processo do `dtrace` atual, ou a qualquer processo que esteja sendo executado no compilador de D.

Usar variáveis de macro em arquivos de intérprete permite que você crie programas em D duradouros que não precisam ser editados toda vez que deseja usá-los. Por exemplo, para contar todas as chamadas do sistema exceto aquelas executadas pelo comando `dtrace`, use a seguinte cláusula do programa em D contendo `$pid`:

```
syscall:::entry
/pid != $pid/
{
    @calls = count();
}
```

Esta cláusula sempre produz o resultado desejado, embora cada chamada do comando `dtrace` tenha um ID de processo diferente.

As variáveis de macro podem ser usadas em qualquer lugar que um inteiro, identificador ou seqüência possam ser usados em um programa em D. As variáveis de macro são expandidas apenas uma vez (ou seja, não recursivamente) quando a saída é analisada. Cada variável de macro é expandida para formar um símbolo de entrada separado, e não pode ser concatenada com outro texto para produzir um símbolo único. Por exemplo, se `$pid` se expandisse para o valor 456, o código de D:

```
123$pid
```

se expandiria para os dois símbolos adjacentes 123 e 456, resultando em um erro de sintaxe, em vez de um único símbolo de inteiro 123456.

As variáveis de macro são expandidas e concatenadas com texto adjacente dentro de descrições de teste de D no início das suas cláusulas de programa. Por exemplo, a cláusula seguinte usa o provedor `pid` do DTrace para instrumentar o comando `dtrace`:

```
pid$pid:libc.so:printf:entry
{
    ...
}
```

As variáveis de macro são expandidas apenas uma vez em cada campo de descrição de teste; elas talvez não contenham delimitadores de descrição de teste (`:`).

Argumentos de macro

O compilador de D também fornece um conjunto de variáveis de macro correspondentes a quaisquer operandos de argumento adicionais especificados como parte da chamada do comando `dtrace`. Estes *argumentos de macro* são acessados através de nomes incorporados `$0` do nome do arquivo do programa em D ou do comando `dtrace` , `$1` do primeiro operando adicional, `$2` do segundo operando, e assim por diante. Se você usar a opção `-s` do `dtrace` , `$0` se expandirá para o valor do nome do arquivo de entrada usado com esta opção. Para programas em D especificados na linha de comando, `$0` se expande para o valor de `argv[0]` usado para executar o próprio `dtrace`.

Os argumentos de macro podem se expandir para inteiros, identificadores ou seqüências, dependendo do formato do texto correspondente. Como acontece com todas as variáveis de

macro, os argumentos de macro podem ser usados em qualquer lugar em que um inteiro, um identificador e os símbolos de seqüência possam ser usados em um programa em D. Todos os exemplos seguintes poderiam formar expressões de D válidas, presumindo-se que os valores de argumento de macro estejam apropriados:

```
execname == $1    /* with a string macro argument */
x += $1           /* with an integer macro argument */
trace(x->$1)      /* with an identifier macro argument */
```

Os argumentos de macro podem ser usados para criar arquivos de intérprete do `dtrace` que agem como comandos reais do Solaris e usam informações especificadas por um usuário ou por outra ferramenta para modificar seu comportamento. Por exemplo, o arquivo de intérprete de D rasteia as chamadas do sistema `write(2)` executadas por um ID de processo específico:

```
#!/usr/sbin/dtrace -s

syscall::write:entry
/pid == $1/
{
}
```

Se você tornar este arquivo de intérprete executável, poderá especificar o valor de `$1` usando argumentos de linha de comando adicionais em seu arquivo de intérprete:

```
# chmod a+rx ./tracewrite
# ./tracewrite 12345
```

A chamada de comando resultante conta cada chamada do sistema `write(2)` executada pelo ID de processo 12345.

Se o seu programa em D fizer referência a um argumento de macro que não seja fornecido na linha de comando, será impressa uma mensagem de erro apropriada e o seu programa não será compilado:

```
# ./tracewrite
dtrace: failed to compile script ./tracewrite: line 4:
  macro argument $1 is not defined
```

Os programas em D poderão fazer referência a argumentos de macro não especificados, se a opção `defaultargs` estiver definida. Se `defaultargs` estiver definida, os argumentos não especificados terão o valor `0`. Consulte o Capítulo 16, “Opções e ajustáveis” para obter mais informações sobre as opções do compilador de D. O compilador de D também produzirá uma mensagem de erro, se forem especificados argumentos adicionais na linha de comando que não tenham sido referenciados por seu programa em D.

Os valores de argumento de macro devem correspondem ao formato de um inteiro, identificador ou seqüência. Se o argumento não corresponder a nenhum desses formatos, o

compilador de D informará uma mensagem de erro apropriada. Ao especificar argumentos de macro de seqüência para um arquivo de intérprete do DTrace, coloque o argumento entre um par de aspas simples extra a fim de evitar a interpretação das aspas duplas e do conteúdo da seqüência pelo seu shell:

```
# ./foo '"a string argument"'
```

Se você quiser que os seus argumentos de macro de D sejam interpretados como símbolos de seqüência mesmo que eles correspondam a um inteiro ou identificador, coloque um prefixo na variável de macro ou nome do argumento com dois sinais de dólar à esquerda (por exemplo, `$$1`) para forçar o compilador de D a interpretar o valor do argumento como se ele fosse uma seqüência entre aspas duplas. Todas as seqüências de escape de seqüência de D (consulte a Tabela 2–5) são expandidas dentro de quaisquer argumentos de macro de seqüência, independentemente de serem referenciadas através do formato $*arg* ou $$*arg* da macro. Se a opção `defaultargs` estiver definida, os argumentos não especificados que são referenciados com o formato $$*arg* possuem o valor da seqüência vazia ("").

ID do processo de destino

Use a variável de macro `$target` para criar scripts que possam ser aplicados a um processo de usuário específico de seu interesse que seja selecionado na linha de comando do `dtrace` com a opção `-p` ou criado com a opção `-c`. Os programas em D especificados na linha de comando ou com a opção `-s` são compilados *depois* que os processos são criados ou capturados, e a variável `$target` se expande para o ID de processo de inteiro do primeiro processo desse tipo. Por exemplo, o script de D poderia ser usado para determinar a distribuição de chamadas do sistema executadas por processo de assunto específico:

```
syscall::entry
/pid == $target/
{
    @[probefunc] = count();
}
```

Para determinar o número de chamadas do sistema executadas pelo comando `date(1)` salve o script no arquivo `syscall.d` e execute o seguinte comando:

```
# 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
```

◆ ◆ ◆ CAPÍTULO 16

16 Opções e ajustáveis

Para permitir a personalização, o DTrace dá aos seus consumidores vários graus de liberdade importantes. Para minimizar a probabilidade de precisar de ajuste específico, o DTrace é implementado com valores padrão razoáveis e políticas padrão flexíveis. Entretanto, pode haver situações em que seja necessário ajustar o comportamento do DTrace numa base consumidor a consumidor. Este capítulo descreve as opções e os ajustáveis do DTrace, e as interfaces que você pode usar para modificá-los.

Opções do consumidor

O DTrace é ajustado para definir e ativar as opções. As opções disponíveis são descritas na tabela abaixo. Para obter algumas opções, `dtrace`(1M) fornece uma opção de linha de comando correspondente.

TABELA 16–1 Opções do consumidor do DTrace

Nome da opção	Valor	Alias do `dtrace`(1M)	Descrição	Consulte o capítulo
`aggrate`	*time*		Taxa de leitura de agregação	Capítulo 9, “Agregações”
`aggsize`	*size*		Tamanho do buffer de agregação	Capítulo 9, “Agregações”
`bufresize`	`auto` ou `manual`		Política de redimensionamento do buffer	Capítulo 11, “Buffers e armazenamento em buffer”
`bufsize`	*size*	`-b`	Tamanho do buffer principal	Capítulo 11, “Buffers e armazenamento em buffer”

TABELA 16–1 Opções do consumidor do DTrace *(Continuação)*

Nome da opção	Valor	Alias do `dtrace`(1M)	Descrição	Consulte o capítulo
`cleanrate`	*time*		Taxa de limpeza. Deve ser especificado em número por segundo com o sufixo `hz`.	Capítulo 13, “Rastreio especulativo”
`cpu`	*escalar*	`-c`	CPU na qual ativar o rastreio	Capítulo 11, “Buffers e armazenamento em buffer”
`defaultargs`	—		Permite referências a argumentos de macro não especificados	Capítulo 15, “Script”
`destructive`	—	`-w`	Permite ações destrutivas	Capítulo 10, “Ações e sub-rotinas”
`dynvarsize`	*size*		Tamanho de espaço de variável dinâmica	Capítulo 3, “Variáveis”
`flowindent`	—	`-F`	Recua a entrada da função e prefixa com `->`; cancela o recuo do retorno da função e prefixa com `<-`	Capítulo 14, “Utilitário `dtrace`(1M)”
`grabanon`	—	`-a`	Declara o estado anônimo	Capítulo 36, “Rastreio anônimo”
`jstackframes`	*escalar*		Número de quadros de pilha padrão `jstack()`	Capítulo 10, “Ações e sub-rotinas”
`jstackstrsize`	*escalar*		Tamanho de espaço de seqüência padrão de `jstack()`	Capítulo 10, “Ações e sub-rotinas”
`nspec`	*escalar*		Número de especulações	Capítulo 13, “Rastreio especulativo”
`quiet`	—	`-q`	Saída apenas de dados rastreados explicitamente	Capítulo 14, “Utilitário `dtrace`(1M)”

TABELA 16–1 Opções do consumidor do DTrace *(Continuação)*

Nome da opção	Valor	Alias do `dtrace`(1M)	Descrição	Consulte o capítulo
`specsize`	*size*		Tamanho do buffer de especulação	Capítulo 13, "Rastreio especulativo"
`strsize`	*size*		Tamanho da seqüência	Capítulo 6, "Seqüências"
`stackframes`	*escalar*		Número de quadros de pilha	Capítulo 10, "Ações e sub-rotinas"
`stackindent`	*escalar*		Número de caracteres de espaço em branco a serem usados ao recuar a saída de `stack()` e `ustack()`	Capítulo 10, "Ações e sub-rotinas"
`statusrate`	*time*		Taxa de verificação de status	
`switchrate`	*time*		Taxa de alternância de buffer	Capítulo 11, "Buffers e armazenamento em buffer"
`ustackframes`	*escalar*		Número de quadros de pilha do usuário	Capítulo 10, "Ações e sub-rotinas"

Valores que indicam tamanhos podem receber um sufixo opcional `k`, `m`, `g` ou `t` para indicar kilobytes, megabytes, gigabytes e terabytes, respectivamente. Valores que indicam horas podem receber um sufixo adicional `ns`, `us`, `ms`, `s` ou `hz` para indicar nanossegundos, microssegundos, milissegundos, segundos e número por segundo, respectivamente.

Modificando opções

As opções podem ser definidas em um script de D com `#pragma D` seguido pela `opção` de seqüência e o nome da opção. Se a opção tiver um valor, o nome dela deve ser seguido por um sinal de igual (=) e o valor da opção. Os exemplos seguintes são configurações de opção válidas:

```
#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
```

O comando dtrace(1M) também aceita configurações de opção na linha de comando, como um argumento para a opção -x. Por exemplo:

```
# dtrace -x nspec=4 -x grabanon -x bufsize=2g \
    -x switchrate=10hz -x aggrate=100us -x bufresize=manual
```

Se uma opção inválida for especificada, o dtrace indica que o nome da opção é inválido e existe:

```
# dtrace -x wombats=25
dtrace: failed to set option -x wombats: Invalid option name
#
```

Semelhantemente, se um valor de opção não for válido para a opção fornecida, o dtrace indicará que o valor é inválido:

```
# dtrace -x bufsize=100wombats
dtrace: failed to set option -x bufsize: Invalid value for specified option
#
```

Se uma opção for definida mais de uma vez, as configurações subseqüentes sobrescrevem as configurações anteriores. Algumas opções, tais como grabanon, podem ser *somente* definidas. A presença de tal opção a define, e você não pode cancelar a sua definição mais tarde.

As opções que são definidas para uma ativação anônima serão honradas pelo consumidor do DTrace que declara o estado anônimo. Consulte o Capítulo 36, “Rastreio anônimo” para obter informações sobre a ativação do rastreio anônimo.

◆ ◆ ◆ CAPÍTULO 17

Provedor do `dtrace`

O provedor do `dtrace` oferece vários testes relacionados ao próprio DTrace. Você pode usar esses testes para inicializar o estado antes que o rastreio comece, processar o estado depois que o rastreio foi concluído, e manipular os erros de execução inesperados em outros testes.

Teste `BEGIN`

O teste `BEGIN` é acionado antes de qualquer outro teste. Nenhum outro teste será acionado até que todas as cláusulas de `BEGIN` tiverem sido concluídas. Este teste pode ser usado para inicializar qualquer estado que seja necessário em outros testes. O exemplo seguinte mostra como usar o teste `BEGIN` para inicializar uma matriz de associação para mapear entre os bits de proteção de `mmap(2)` e uma representação textual:

```
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]);
}
```

O teste `BEGIN` é acionado em um contexto inesperado. Isso significa que a saída de `stack()` ou `ustack ()`, e o valor de variáveis específicas de contexto (por exemplo, `execname`), são todos

arbitrários. Não se deve confiar nem interpretar esses valores para definir quaisquer informações significativas. Nenhum argumento é definido para o teste `BEGIN`.

Teste END

O teste `END` é acionado depois de todos os outros testes. Este teste não será acionado até que todas as cláusulas de teste tenham sido concluídas. Este teste pode ser usado para processar o estado que foi coletado ou para formatar a saída. A ação `printa()` é, portanto, freqüentemente usada no teste `END`. Os testes `BEGIN` e `END` podem ser usados juntos para medir o tempo total gasto no rastreio:

```
BEGIN
{
    start = timestamp;
}

/*
 * ... other tracing actions...
 */

END
{
    printf("total time: %d secs", (timestamp - start) / 1000000000);
}
```

Consulte "Normalização de dados" na página 125 e "`printa()` na página 167 para obter outros usos do teste `END`.

Como acontece com o teste `BEGIN`, nenhum argumento é definido para o teste `END`. O contexto no qual o teste `END` é acionado é arbitrário e não deve depender dele.

Durante o rastreio com a opção `bufpolicy` definida como `fill`, o espaço adequado é reservado para acomodar quaisquer registros rastreados no teste `END`. Consulte "Política `fill` e testes `END`" na página 157 para obter detalhes.

Observação – A ação `exit()` faz com que o rastreio pare e o teste `END` seja acionado. Entretanto, existe algum atraso entre a chamada da ação `exit()` e o acionamento do teste `END`. Durante esse atraso, nenhum teste será acionado. Depois que um teste chama a ação `exit()`, o teste `END` não é acionado até que o consumidor do DTrace determine que `exit()` tenha sido chamada e o rastreio pare. A taxa na qual o status de saída é verificado pode ser definida através da opção `statusrate`. Para obter mais informações, consulte o Capítulo 16, "Opções e ajustáveis".

Teste ERROR

O teste ERROR é acionado quando um erro de tempo de execução ocorre na execução de uma cláusula de um teste do DTrace. Por exemplo, se uma cláusula tentar cancelar a referência de um ponteiro NULL, o teste ERROR será acionado, como mostrado no exemplo seguinte.

EXEMPLO 17–1 error.d: erros de registro

```
BEGIN
{
    *(char *)NULL;
}

ERROR
{
    printf("Hit an error!");
}
```

Ao executar este programa, você verá uma saída parecida com o exemplo seguinte:

```
# 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
```

A saída mostra que o teste ERROR foi acionado, e também ilustra o dtrace(1M) informando o erro. dtrace possui sua própria ativação do teste ERROR para permitir que ele informe os erros. Usando o teste ERROR, você pode criar sua própria manipulação de erro personalizada.

Os argumentos para o teste ERROR são da seguinte forma:

arg1	O identificador de teste ativado (EPID) do teste que causou o erro
arg2	O índice da ação que causou a falha
arg3	O deslocamento DIF dessa ação ou -1, caso não seja aplicável
arg4	O tipo de falha
arg5	Valor particular para o tipo de falha

A tabela abaixo descreve os vários tipos de falha e o valor que arg5 terá para cada um:

Valor de `arg4`	Descrição	Significado de `arg5`
`DTRACEFLT_UNKNOWN`	Tipo de falha desconhecido	Nenhum
`DTRACEFLT_BADADDR`	Acesso a endereço não mapeado ou inválido	Endereço acessado
`DTRACEFLT_BADALIGN`	Acesso de memória não alinhada	Endereço acessado
`DTRACEFLT_ILLOP`	Operação ilegal ou inválida	Nenhum
`DTRACEFLT_DIVZERO`	Divisão de inteiro por zero	Nenhum
`DTRACEFLT_NOSCRATCH`	Espaço temporário insuficiente para satisfazer a alocação temporária	Nenhum
`DTRACEFLT_KPRIV`	Tentativa de acessar um endereço ou propriedade de kernel sem privilégios suficientes	Endereço acessado ou `0`, caso não seja aplicável
`DTRACEFLT_UPRIV`	Tentativa de acessar um endereço de usuário ou propriedade sem privilégios suficientes	Endereço acessado ou `0`, caso não seja aplicável
`DTRACEFLT_TUPOFLOW`	Estouro de fluxo de pilha de parâmetro interno do DTrace	Nenhum

Se as ações tomadas no próprio teste `ERROR` causarem um erro, esse erro será silenciosamente eliminado — o teste `ERROR` não será chamado recursivamente.

Estabilidade

O provedor do `dtrace` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Estável	Estável	Comum
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Estável	Estável	Comum
Argumentos	Estável	Estável	Comum

◆ ◆ ◆ CAPÍTULO 18

Provedor `lockstat`

O provedor `lockstat` disponibiliza os testes que podem ser usados para discernir estatísticas de contenção de bloqueio ou para entender praticamente qualquer aspecto do comportamento de bloqueio. O comando `lockstat(1M)` é realmente um consumidor do DTrace que usa o provedor `lockstat` para recolher seus dados básicos.

Visão geral

O provedor `lockstat` disponibiliza dois tipos de testes: testes de evento de conteúdo e testes de evento de manutenção.

Evento de contenção os testes correspondem à contenção em uma sincronização primitiva, e são acionados quando um segmento é forçado a aguardar que um recurso se torne disponível. O Solaris é geralmente otimizado para o caso de não-contenção, sendo assim, a contenção prolongada não é esperada. Estes testes devem ser usados para compreender esses casos onde a contenção acontece. Como a contenção é relativamente rara, ativar os testes de evento de contenção geralmente não afeta substancialmente o desempenho.

Testes de *evento de manutenção* que correspondem a adquirir, liberar ou manipular uma sincronização primitiva. Estes testes podem ser usados para responder a questões arbitrárias sobre a forma em que as sincronizações primitivas são manipuladas. Como o Solaris adquire e libera as sincronizações primitivas muito freqüentemente (na ordem de milhões de vezes por segundo por CPU em um sistema ocupado), ativar testes de evento de manutenção tem um efeito de teste muito mais alto que ativar os testes de evento de contenção. Enquanto o efeito de teste induzido pela ativação deles pode ser substancial, ele não é patológico; eles ainda podem ser ativados com confiança em sistemas de produção.

O provedor `lockstat` disponibiliza os testes que correspondem às sincronizações primitivas diferentes no Solaris; estas primitivas e os testes que correspondem a elas são discutidas no restante deste capítulo.

Testes de bloqueio adaptativo

Os bloqueios adaptativos reforçam a exclusão mútua de uma seção crítica, e podem ser adquiridos na maioria dos contextos do kernel. Como os bloqueios adaptativos possuem algumas restrições de contexto, eles possuem a vasta maioria de sincronizações primitivas no kernel do Solaris. Estes bloqueios são adaptativos em seu comportamento com relação à contenção: quando um segmento tenta adquirir um bloqueio adaptativo mantido, ele determinará se o segmento proprietário está sendo executado atualmente em uma CPU. Se o proprietário estiver sendo executado em outra CPU, o segmento de aquisição irá *girar*. Se o proprietário não estiver sendo executado, o segmento de aquisição será *bloqueado*.

Os quatro testes lockstat que pertencem aos bloqueios adaptativos estão na Tabela 18–1. Para cada teste, `arg0` contém um ponteiro para a estrutura `kmutex_t` que representa o bloqueio adaptativo.

TABELA 18–1 Testes de bloqueio adaptativo

`adaptive-acquire`	Teste de evento de manutenção que é acionado imediatamente depois que um bloqueio adaptativo é adquirido.
`adaptive-block`	Teste de evento de contenção que é acionado depois que um segmento bloqueado em um mutex adaptativo mantido foi acionado novamente e adquiriu o mutex. Se ambos os testes forem ativados, `adaptive-block` é acionado *antes* de `adaptive-acquire`. Uma aquisição de bloqueio simples aciona os testes `adaptive-block` e `adaptive-spin`. `arg1` de `adaptive-block` contém o tempo de espera em nanossegundos.
`adaptive-spin`	Teste de evento de contenção que é acionado depois que um segmento girado em um mutex adaptativo mantido adquiriu o mutex com êxito. Se ambos os testes forem ativados, `adaptive-spin` é acionado *antes* de `adaptive-acquire`. Uma aquisição de bloqueio simples aciona os testes `adaptive-block` e `adaptive-spin`. `arg1` de `adaptive-spin` contém o *tempo dos giros*: a quantidade de nanossegundos que o ciclo de giros gastou antes que o bloqueio fosse adquirido.
`adaptive-release`	Teste de evento de manutenção que é acionado imediatamente depois que um bloqueio adaptativo é liberado.

Testes de bloqueio de giro

Os segmentos não podem ser bloqueados em alguns contextos no kernel, tais como contexto de interrupção de alto nível e qualquer estado do distribuidor de manipulação de contexto. Nestes contextos, esta restrição impede o uso de bloqueios adaptativos. *Bloqueios de giro* são usados para realizar exclusão mútua de seções críticas nestes contextos. Como o nome sugere, o comportamento destes bloqueios na presença de contenção é gerar até que o bloqueio seja liberado por seu segmento proprietário. Os três testes que pertencem aos bloqueios de giro estão na Tabela 18–2.

TABELA 18–2 Testes de bloqueio de giro

`spin-acquire`	Teste de evento de manutenção que é acionado imediatamente depois que um bloqueio de giro é adquirido.
`spin-spin`	Teste de evento de contenção que é acionado depois que um segmento girado em um bloqueio de giro mantido adquiriu o bloqueio de giro com êxito. Se ambos os testes forem ativados, `spin-spin` é acionado *antes* de `spin-acquire`. `arg1` de `spin-spin` contém o *tempo dos giros*: a quantidade de nanossegundos gasta no estado do giro antes que o bloqueio fosse adquirido. A contagem de giros tem pouca importância, mas pode ser usada para comparar tempos de giro.
`spin-release`	Teste de evento de manutenção que é acionado imediatamente depois que um bloqueio de giro é liberado.

Os bloqueios adaptativos são muito mais comuns que os bloqueios de giro. O script seguinte exibe os totais de ambos os tipos de bloqueio para fornecer dados que dêem suporte a esta observação.

```
lockstat:::adaptive-acquire
/execname == "date"/
{
    @locks["adaptive"] = count();
}

lockstat:::spin-acquire
/execname == "date"/
{
    @locks["spin"] = count();
}
```

Execute este script em uma janela e o comando `date(1)` em outra. Quando terminar o script do DTrace, você verá uma saída semelhante ao exemplo seguinte:

```
# dtrace -s ./whatlock.d
dtrace: script ’./whatlock.d’ matched 5 probes
^C
spin                                                             26
adaptive                                                       2981
```

Como esta saída indica, mais de 99% dos bloqueios adquiridos na execução do comando `date` são bloqueios adaptativos. Talvez seja surpreendente que *tantos* bloqueios sejam adquiridos quando se faz algo tão simples como executar o comando `date`. O grande número de bloqueios é um artefato natural do bloqueio refinado requerido de um sistema extremamente escalável como o kernel do Solaris.

Bloqueios de segmento

Os bloqueios de segmento são um tipo especial de bloqueio de giro usados para bloquear um segmento com a finalidade de alterar o estado do segmento. Os eventos de manutenção de bloqueio de segmento estão disponíveis como testes de evento de manutenção de bloqueio de giro (ou seja, `spin-acquire` e `spin-release`), mas os eventos de contenção possuem seu próprio teste específico para bloqueios de segmento. O teste do evento de manutenção de bloqueio de segmento está na Tabela 18–3.

TABELA 18–3 Testes de bloqueio de segmento

`thread-spin`	Teste de evento de contenção que é acionado depois que um segmento tiver girado em um bloqueio de segmento. Como em outros testes de evento de contenção, se o teste de evento de contenção e o teste de evento de manutenção forem acionados, `thread-spin` será acionado antes de `spin-acquire`. Ao contrário dos testes de evento de contenção, entretanto, thread-spin é acionado *antes* que o bloqueio seja realmente adquirido. Conseqüentemente, vários acionamentos do teste `thread-spin` podem corresponder a um único acionamento do teste `spin-acquire`.

Testes de bloqueio de leitores/gravador

Os bloqueios de leitor/gravador forçam uma política de permissão de vários leitores *ou* um único gravador, mas não ambos, em uma seção crítica. Estes bloqueios são geralmente usados para estruturas que são pesquisadas mais freqüentemente do que modificadas, e para as quais existe tempo substancial na seção crítica. Se os tempos de seção crítica forem curtos, os bloqueios de leitor/gravador serão implicitamente serializados na memória compartilhada usada para implementar o bloqueio, o que não dá a eles nenhuma vantagem sobre os bloqueios adaptativos. Consulte `rwlock(9F)` para obter mais detalhes sobre bloqueios de leitores/gravador.

Os testes que pertencem aos bloqueios de leitores/gravador estão na Tabela 18–4. Para cada teste, `arg0` contém um ponteiro para a estrutura `krwlock_t` que representa o bloqueio adaptativo.

TABELA 18–4 Testes de bloqueio de leitores/gravador

`rw-acquire`	Teste de evento de manutenção que é acionado imediatamente depois que um bloqueio de leitores/gravador é adquirido. `arg1` conterá a constante `RW_READER`, se o bloqueio tiver sido adquirido como um leitor, e `RW_WRITER` se o bloqueio tiver sido adquirido como um escritor.

TABELA 18–4 Testes de bloqueio de leitores/gravador *(Continuação)*

`rw-block`	Teste de evento de contenção que é acionado depois que um segmento bloqueado em um bloqueio de leitores/gravador mantido foi acionado novamente e adquiriu o bloqueio. `arg1` contém a duração do tempo (em nanossegundos) que o segmento atual teve que ficar inativo para adquirir o bloqueio. `arg2` conterá a constante `RW_READER`, se o bloqueio tiver sido adquirido como um leitor, e `RW_WRITER` se o bloqueio tiver sido adquirido como um escritor. `arg3` e `arg4` contêm mais informações sobre o motivo do bloqueio. `arg3` será diferente de zero, se e somente se o bloqueio foi mantido como um escritor quando o segmento atual foi bloqueado. `arg4` contém a contagem de leitores quando o segmento atual foi bloqueado. Se os testes `rw-block` e `rw-acquire` estiverem ativados, `rw-block` é acionado *antes* de `rw-acquire`.
`rw-upgrade`	Teste de evento de manutenção que é acionado depois que um segmento tiver atualizado com êxito um bloqueio de leitores/gravador a partir de um leitor para um gravador. As atualizações não possuem um evento de contenção associado porque elas só são possíveis através de uma interface de não-bloqueio, `rw_tryupgrade(9F)`.
`rw-downgrade`	Teste de evento de manutenção que é acionado depois que um segmento tiver feito downgrade de sua propriedade de um bloqueio de leitores/gravador de gravador para leitor. Os downgrades não possuem um evento de contenção associados porque eles sempre acontecem sem contenção.
`rw-release`	Teste de evento de manutenção que é acionado imediatamente depois que um bloqueio de leitores/gravador é liberado. `arg1` conterá a constante `RW_READER`, se o bloqueio liberado tiver sido mantido como um leitor, e `RW_WRITER` se o bloqueio liberado tiver sido mantido como um gravador. Devido aos upgrades e downgrades, o bloqueio pode *não* ter sido liberado quando foi adquirido.

Estabilidade

O provedor `lockstat` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	Comum
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	Comum
Argumentos	Desenvolvendo	Desenvolvendo	Comum

◆ ◆ ◆ CAPÍTULO 19

Provedor `profile`

O provedor `profile` fornece testes associados a um acionamento de interrupção baseado no tempo sempre fixo, intervalo de tempo especificado. Estes testes *não-ancorados* que não estão associados a qualquer ponto de execução específico, mas ao evento de interrupção assíncrono. Estes testes podem ser usados para fazer amostragem de alguns aspectos do estado do sistema periodicamente e as amostras podem ser usadas para deduzir o comportamento do sistema. Se a taxa de amostragem for alta, ou o tempo de amostragem for longo, uma dedução correta será possível. Através de ações do DTrace, o provedor `profile` pode ser usado para fazer amostragem de praticamente tudo no sistema. Por exemplo, você poderia fazer amostragem do estado do segmento atual, o estado da CPU, ou a instrução da máquina atual.

Observação – As variáveis de segmento locais não estão acessíveis para testes a partir do provedor `profile` . O uso do identificador especial `self` para fazer referência a uma variável de segmento local com tal teste não produzirá nenhuma saída.

Testes `profile-`*n*

Um teste `profile-` *n* é acionado a cada intervalo fixo em cada CPU no nível de interrupção alto. O intervalo de acionamento do teste é indicado pelo valor *n*: a origem da interrupção será acionada *n* vezes por segundo. *n* também pode ter um sufixo de tempo opcional, em cujo caso *n* é interpretado em unidades indicadas pelo sufixo. Os sufixos válidos e as unidades que eles indicam estão listados na Tabela 19–1.

TABELA 19–1 Sufixos de tempo válidos

Sufixo	Unidades de tempo
`nsec` ou `ns`	nanossegundos
`usec` ou `us`	microssegundos

TABELA 19–1 Sufixos de tempo válidos *(Continuação)*

Sufixo	Unidades de tempo
`msec` ou `ms`	milissegundos
`sec` ou `s`	segundos
`min` ou `m`	minutos
`hour` ou `h`	horas
`day` ou `d`	dias
`hz`	hertz (freqüência por segundo)

O exemplo seguinte cria um teste a ser acionado a 97 hertz para fazer amostragem do processo que está em execução no momento:

```
#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);
}
```

Executar o exemplo acima por um breve período de tempo resulta em uma saída semelhante ao exemplo seguinte:

```
# 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
```

Você também usa o provedor `profile-`*n* para fazer amostragem de informações sobre o processo em execução. O script de D de exemplo seguinte usa um teste profile de 1.001 hertz para fazer amostragem da prioridade atual de um processo especificado:

```
profile-1001
/pid == $1/
{
    @proc[execname] = lquantize(curlwpsinfo->pr_pri, 0, 100, 10);
}
```

Para ver este script de exemplo em ação, digite os seguintes comandos em uma janela:

```
$ echo $$
12345
$ while true ; do let i=0 ; done
```

Em outra janela, execute o script de D por um breve período de tempo, substituindo *12345* pelo PID que o comando `echo` retornou:

```
# 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
```

Esta saída mostra a diferença da classe de programação de compartilhamento de tempo. Como o processo do shell está girando na CPU, sua prioridade está constantemente sendo baixada pelo sistema. Se o processo do shell estivesse sendo executado menos freqüentemente, sua prioridade seria mais alta. Para ver este resultado, digite Control-C no shell que está girando e execute o script novamente:

```
# dtrace -s ./profpri.d 494621
 dtrace: script './profpri.d' matched 1 probe
```

Agora no shell, digite alguns caracteres. Quando você terminar o script do DTrace, uma saída parecida com o exemplo seguinte aparecerá:

```
ksh
           value  ------------- Distribution ------------- count
              40 |                                         0
              50 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 14
              60 |                                         0
```

Como o processo do shell estava inativo, esperando a entrada do usuário em vez de girando na CPU, quando ele *foi* executado, sua prioridade era muito mais alta.

Testes tick-*n*

Assim como os testes profile- *n*, os testes tick- *n* são acionados a cada intervalo fixo em um nível de interrupção alto. Entretanto, ao contrário dos testes profile-*n*, que são acionados em *cada* CPU, os testes tick-*n* são acionados somente em *uma* CPU por intervalo. A CPU real pode mudar com o tempo. Como acontece com os testes profile- *n*, *n* é padronizado como taxa por segundo, mas também pode ter um sufixo de tempo opcional. Os testes tick- *n* têm vários usos, tais como fornecer alguma saída periódica ou tomar uma ação periódica.

Argumentos

Os argumentos dos testes profile são os seguintes:

arg0	O contador do programa (PC) no kernel na hora em que o teste foi acionado, ou 0 se o processo atual não estava sendo executado no kernel na hora em que o teste foi acionado
arg1	O PC no processo no nível do usuário na hora em que o teste foi acionado, ou 0 se o processo atual estava sendo executado no kernel na hora em que o teste foi acionado

Conforme as descrições sugerem, se arg0 for diferente de zero, então arg1 é zero; se arg0 for zero, então arg1 é diferente de zero. Portanto, você pode usar arg0 e arg1 para diferenciar o nível do usuário do nível do kernel, como neste exemplo simples:

```
profile-1ms
{
    @ticks[arg0 ? "kernel" : "user"] = count();
}
```

Resolução do temporizador

O provedor `profile` usa temporizadores de intervalo de resolução arbitrário no sistema operacional. Em arquiteturas que não aceitam interrupções baseadas em tempo de resolução realmente arbitrário, a freqüência é limitada pela freqüência do relógio do sistema, que é especificada pela variável do kernel `hz`. Os testes de freqüência mais alta que `hz` em tais arquiteturas serão acionados algumas vezes a cada 1/ `hz` segundos. Por exemplo, um teste `profile` de 1000 hertz em uma arquitetura como tal com `hz` definido como 100 será acionado dez vezes em sucessão rápida a cada dez milessegundos. Em plataformas que aceitam resolução arbitrária, um teste `profile` de 1000 hertz seria acionado exatamente a cada um milissegundo.

O exemplo seguinte testa uma determinada resolução de arquitetura:

```
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);
}
```

Em uma arquitetura que aceita testes `profile` de resolução arbitrária, executar o script de exemplo produzirá uma distribuição uniforme:

```
# 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
```

Em uma arquitetura que não aceita testes `profile` de resolução arbitrária, executar o script de exemplo produzirá uma distribuição não-uniforme:

```
# 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
```

Nestas arquiteturas, `hz` pode ser ajustado manualmente em `/etc/system` para aprimorar a resolução efetiva de profile.

Atualmente, todas as variantes de UltraSPARC (`sun4u`) aceitam testes `profile` de resolução arbitrária. Muitas variantes da arquitetura x86 (`i86pc`) também aceitam testes `profile` de resolução arbitrária, embora algumas variantes mais antigas não aceitem.

Criação de teste

Ao contrário de outros provedores, o provedor `profile` cria testes dinamicamente, conforme a necessidade. Sendo assim, o teste profile desejado pode não aparecer em uma listagem de todos os testes (por exemplo, usando `dtrace -l -P profile`), mas o teste será criado quando for ativado explicitamente.

Em arquiteturas que aceitam testes `profile` de resolução arbitrária, um intervalo de tempo que seja muito curto faria com que a máquina continuamente fizesse interrupções baseadas no tempo, portanto, negando serviço na máquina. Para evitar esta situação, o provedor `profile` se recusará silenciosamente a criar qualquer teste que resultaria em um intervalo de menos de duzentos microssegundos.

Estabilidade

O provedor `profile` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	Comum
Módulo	Instável	Instável	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	Comum
Argumentos	Desenvolvendo	Desenvolvendo	Comum

◆ ◆ ◆ CAPÍTULO 20

20 Provedor `fbt`

Este capítulo descreve o provedor Function Boundary Tracing (FBT), que fornece testes associados à entrada e ao retorno da maioria das funções no kernel do Solaris. A função é a unidade fundamental de texto do programa. Em um sistema bem projetado, cada função realiza uma operação discreta e bem definida em um objeto especificado ou uma série de objetos semelhantes. Portanto, inclusive nos menores sistemas Solaris, o FBT fornecerá na ordem de 20.000 testes.

Semelhante a outros provedores do DTrace, o FBT não tem efeito de teste quando não é ativado explicitamente. Quando ativado, o FBT apenas induz um efeito de teste em funções testadas. Embora a implementação do FBT seja altamente específica da arquitetura do conjunto de instruções, o FBT foi implementado em plataformas SPARC e x86. Para cada conjunto de instruções, existe um pequeno número de funções que não chamam outras funções, e são altamente otimizadas pelo compilador (chamadas de *funções em folha*) que não podem ser instrumentadas pelo FBT. Testes destas funções não estão presentes no DTrace.

O uso efetivo de testes do FBT requer o conhecimento da implementação do sistema operacional. Portanto, recomenda-se o uso do FBT somente quando você estiver desenvolvendo o software do kernel ou quanto outros provedores não são suficientes. Outros provedores do DTrace, incluindo `syscall`, `sched`, `proc` e `io`, podem ser usados para responder à maioria das questões de análise do sistema sem requerer o conhecimento da implementação do sistema operacional.

Testes

FBT fornece um teste no *limite* da maioria das funções do kernel. O limite de uma função é ultrapassado ao entrar na função e retornar da função. O FBT, portanto, fornece duas funções para cada função no kernel: uma na entrada para a função e uma no retorno da função. Estes testes são chamados de `entry` e `return`, respectivamente. O nome da função e o nome do módulo são especificados como parte do teste. Todos os testes do FBT especificam um nome de função e um nome de módulo.

Argumentos de teste

Testes `entry`

Os argumentos de testes `entry` são os mesmos argumentos da função do kernel do sistema operacional correspondente. Estes argumentos podem ser acessados em forma de digitação, usando a matriz `args[]`. Estes argumentos podem ser acessados como `int64_t` usando o `arg0` .. variáveis `arg`*n*.

Testes `return`

Enquanto uma determinada função possui somente um ponto de entrada, ela pode ter muitos pontos diferentes onde ela retorna para o seu chamador. Você está geralmente interessado no valor que uma função retornou ou no fato de a função ter retornado em vez do caminho de retorno específico que foi tomado. O FBT , portanto, coleta os vários locais de retorno de uma função em um único teste `return`. Se o caminho de retorno exato for do seu interesse, examine o valor `args[0]` do teste `return`, que indica o *deslocamento* (em bytes) da instrução de retorno no texto da função.

Se a função tiver um valor de retorno, este valor será armazenado em `args[1]`. Se uma função não tiver um valor de retorno, `args[1]` não é definida.

Exemplos

Você pode usar o FBT para explorar facilmente a implementação do kernel. O script de exemplo seguinte registra o primeiro `ioctl(2)` de qualquer processo `xclock` e, em seguida, segue o caminho de código subseqüente através do kernel:

```
/*
 * 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);
}
```

Executar este script acima resultará numa saída semelhante ao exemplo seguinte:

```
# 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
```

A saída mostra que um processo `xclock` chamou `ioctl()` em um descritor de arquivo que parece estar associado a um soquete.

Você também pode usar o FBT ao tentar entender os drivers do kernel. Por exemplo, o driver `ssd`(7D) possui muitos caminhos de código, pelos quais `EIO` pode ser retornado. O FBT pode

facilmente ser usado para determinar o caminho de código exato que resultou em uma condição de erro, como mostrado no exemplo seguinte:

```
fbt:ssd::return
/arg1 == EIO/
{
    printf("%s+%x returned EIO.", probefunc, arg0);
}
```

Para obter mais informações sobre qualquer retorno de `EIO`, talvez você queira rastrear especulativamente todos os testes de `fbt` e usar `commit()`(ou `discard()`) com base no valor de retorno de uma função específica. Consulte o Capítulo 13, “Rastreio especulativo” para obter mais detalhes sobre rastreios especulativos.

Alternativamente, você pode usar o FBT para entender as funções chamadas em um módulo especificado. O exemplo seguinte lista todas as funções chamadas em 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
```

Se você souber a finalidade ou os argumentos de uma função do kernel, use o FBT para entender como ou por que a função está sendo chamada. Por exemplo, `putnext`(9F) considera um ponteiro para uma estrutura `queue`(9S) como seu primeiro membro. O membro `q_qinfo` da estrutura `queue` é um ponteiro para uma estrutura `qinit`(9S). O membro `qi_minfo` da estrutura `qinit` tem um ponteiro para uma estrutura `module_info`(9S), que contém o nome do módulo em seu membro `mi_idname`. O exemplo seguinte reúne suas informações usando o teste do FBT em `putnext` para controlar chamadas `putnext`(9F) por nome de módulo:

```
fbt::putnext:entry
{
    @calls[stringof(args[0]->q_qinfo->qi_minfo->mi_idname)] = count();
}
```

Executar o script acima resultará numa saída semelhante ao exemplo seguinte:

```
# 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
```

Você também pode usar o FBT para determinar o tempo gasto em uma função específica. O exemplo seguinte mostra como determinar os chamadores das rotinas de atraso DDI `drv_usecwait(9F)` e `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;
}
```

Este script de exemplo é particularmente interessante de ser executado durante a reinicialização. O Capítulo 36, "Rastreio anônimo" descreve o procedimento para realizar o rastreio anônimo durante a inicialização do sistema. Durante a reinicialização, você talvez veja uma saída semelhante ao exemplo seguinte:

```
# dtrace -ae

              ataʻata_wait+0x34
              ataʻata_id_common+0xf5
              ataʻata_disk_id+0x20
              ataʻata_drive_type+0x9a
              ataʻata_init_drive+0xa2
              ataʻata_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
```

Otimização de chamada de laço

Quando uma função termina ao chamar outra função, o compilador pode se envolver na *otimização de chamada de laço*, na qual a função que está sendo chamada reutiliza o quadro de pilha do chamador. Este procedimento é mais usado na arquitetura SPARC, onde o compilador reutiliza a janela do registro do chamador na função que está sendo chamada para minimizar a pressão da janela do registro.

A presença desta otimização faz com que o teste `return` da função que está chamando seja acionado *antes* do teste `entry` da função chamada. Esta ordem pode causar um pouco de confusão. Por exemplo, se você quisesse registrar todas as funções de uma função específica e quaisquer funções que esta função chama, teria que usar o script seguinte:

```
fbt::foo:entry
{
    self->traceme = 1;
}

fbt:::entry
/self->traceme/
{
    printf("called %s", probefunc);
}

fbt::foo:return
/self->traceme/
{
```

```
    self->traceme = 0;
}
```

Entretanto, se `foo()` terminar em uma chamada de laço otimizada, a função de chamada de laço, e portanto quaisquer funções que ela chama, não seriam capturadas. O kernel não pode ser dinamicamente desotimizado durante o uso, e o DTrace não quer entrar em uma situação falsa sobre como o código é reestruturado. Portanto, você deve estar ciente de quando a otimização de chamada de laço deve ser usada.

A otimização de chamada de laço deve ser usada em um código-fonte semelhante ao exemplo seguinte:

```
    return (bar());
```

Ou em um código-fonte semelhante ao exemplo seguinte:

```
    (void) bar();
    return;
```

Contrariamente, o código-fonte da função que termina como o exemplo seguinte *não pode* ter a sua chamada à `bar()` otimizada, pois a chamada à `bar()` não é uma chamada de laço:

```
    bar();
    return (rval);
```

Você pode determinar se uma chamada sofreu otimização de chamada de laço usando a técnica seguinte:

- Ao executar o DTrace, rastreie `arg0` do teste `return` em questão. `arg0` contém o deslocamento da instrução de retorno na função.
- Depois que o DTrace tiver parado, use `mdb(1)` para examinar a função. Se o deslocamento rastreado contiver uma chamava para outra função em vez de uma instrução para retornar da função, a chamada sofreu otimização de chamada de laço.

Devido à arquitetura do conjunto de instruções, a otimização de chamada de laço é muito mais comum em sistemas SPARC do que em sistemas x86. O exemplo seguinte usa `mdb` para descobrir a otimização de chamada de laço na função `dup()` do kernel:

```
# dtrace -q -n fbt::dup:return'{printf("%s+0x%x", probefunc, arg0);}'
```

Enquanto este comando está sendo executado, execute um programa que realize um `dup(2)`, tal como um processo `bash`. O comando acima deve fornecer uma saída semelhante ao exemplo seguinte:

```
dup+0x10
^C
```

Agora examine a função com `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
```

A saída mostra que `dup+0x10` é uma chamada para a função `fcntl()` e não uma instrução `ret`. Portanto, a chamada à `fcntl()` é um exemplo de otimização de chamada de laço.

Funções de composição

Você talvez observe funções que parecem entrar mas nunca retornar ou vice-versa. Funções tão raras são geralmente rotinas de composição codificadas à mão que se ramificam para o meio das funções de composição codificadas à mão. Estas funções não devem impedir a análise: a função ramificada-para ainda devem retornar para o chamador da função ramificada-de. Ou seja, se você ativar todos os testes de FBT, verá a entrada para uma função e o retorno de outra função na mesma profundidade da pilha.

Limitações do conjunto de instruções

Algumas funções não podem ser instrumentadas pelo FBT. A natureza exata das funções não-instrumentáveis é específica da arquitetura do conjunto de instruções.

Limitações de x86

As funções que não criam um quadro de pilha em sistemas x86 não podem ser instrumentadas por FBT. Como o conjunto de registros de x86 é extraordinariamente pequeno, muitas funções devem colocar dados na pilha e, portanto, criar um quadro de pilha. Entretanto, algumas funções de x86 não criam um quadro de pilha e, portanto, não podem ser instrumentadas. Os números reais variam, mas geralmente menos de 5% das funções não podem ser instrumentadas na plataforma x86.

Limitações de SPARC

As rotinas em folha codificadas à mão em linguagem de composição em sistemas SPARC não podem ser instrumentadas por FBT. A maior parte do kernel é escrita em C, e todas as funções escritas em C podem ser instrumentadas por FBT.

Interação de ponto de interrupção

FBT funciona modificando dinamicamente o texto do kernel. Como os pontos de interrupção do kernel também funcionam modificando o texto do kernel, se um ponto de interrupção do kernel for colocado em um local de entrada ou de retorno *antes* de carregar o DTrace, o FBT se recusará a fornecer um teste da função, mesmo se o ponto de interrupção do kernel for subseqüentemente removido. Se o ponto de interrupção do kernel for colocado *depois* de carregar o DTrace, o ponto de interrupção do kernel e o teste do DTrace corresponderão ao mesmo ponto no texto. Nesta situação, o ponto de interrupção disparará primeiro e, em seguida, o teste será acionado quando o depurador continuar o kernel. É recomendado que os pontos de interrupção do kernel não sejam usados ao mesmo tempo que o DTrace. Se os pontos de interrupção forem necessários, use a ação `breakpoint()` do DTrace.

Carregamento de módulo

O kernel do Solaris pode carregar e descarregar dinamicamente os módulos do kernel. Quando o FBT é carregado e um módulo é dinamicamente carregado, o FBT fornece automaticamente novos testes associados ao novo módulo. Se um módulo carregado possuir testes FBT *não-ativados*, o módulo pode ser descarregado; os testes correspondentes serão destruídos quando o módulo for descarregado. Se um módulo carregado tiver testes FBT *ativados*, o módulo será considerado ocupado, e não poderão ser descarregados.

Estabilidade

O provedor FBT usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA
Argumentos	Privada	Privada	ISA

Como o FBT expõe a implementação do kernel, nada sobre seu nome Estável — e do Módulo e da Função e a estabilidade dos dados é explicitamente Privado. A estabilidade dos dados de Provedor e Nome são Desenvolvendo, mas todas as outras estabilidades de dados são Privadas:

elas são artefatos da implementação atual. A classe de dependência de FBT é ISA: embora o FBT esteja disponível em todas as arquiteturas do conjunto de instruções atual, não há garantia de que o FBT esteja disponível em futuras arquiteturas de conjunto de instruções arbitrárias.

◆ ◆ ◆ CAPÍTULO 21

Provedor `syscall`

O provedor `syscall` disponibiliza um teste na entrada e retorno de cada chamada do sistema no sistema. Como as chamadas do sistema são a principal interface entre os aplicativos do nível do usuário e o kernel do sistema operacional, o provedor `syscall` pode oferecer muitas idéias sobre o comportamento do aplicativo com relação ao sistema.

Testes

`syscall` fornece um par de testes para cada chamada do sistema: um teste `entry` que é acionado antes de a chamada do sistema ser inserida, e um teste `return` que é acionado depois que a chamada do sistema tenha sido concluída, mas antes de o controle ter sido transferido de volta para o nível do usuário. Para todos os testes `syscall`, o nome da função é definido como o nome da chamada do sistema instrumentado e o nome do módulo é indefinido.

Os nomes das chamadas do sistema, conforme fornecidos pelo provedor `syscall`, podem ser encontrados no arquivo `/etc/name_to_sysnum`. Com freqüência, os nomes da chamada do sistema fornecidos por `syscall` correspondem aos nomes na Seção 2 das páginas do manual. Entretanto, alguns testes fornecidos pelo provedor `syscall` não correspondem diretamente a qualquer chamada do sistema documentada. As razões comuns para esta discrepância são descritas nesta seção.

Anacronismos da chamada do sistema

Em alguns casos, o nome da chamada do sistema, conforme fornecido pelo provedor `syscall`, é realmente uma reflexão de um detalhe de implementação antigo. Por exemplo, por motivos remontam à época do UNIX™, o nome de `exit(2)` em `/etc/name_to_sysnum` é `rexit`. Da mesma forma, o nome de `time(2)` é `gtime`, e o nome de `execle(2)` e de `execve(2)` é `exece`.

Chamadas do sistema subcodificadas

Algumas chamadas do sistema apresentadas na Seção 2 são implementadas como suboperações de uma chamada do sistema não documentada. Por exemplo, as chamadas do sistema relacionadas aos semáforos System V (`semctl(2)`, `semget(2)`, `semids(2)`, `semop(2)` e `semtimedop(2)`) são implementados como superoperações de uma única chamada do sistema, `semsys`. A chamada do sistema `semsys` considera como seu primeiro argumento um *subcódigo* implementação específica, indicando a chamada do sistema específica necessária: `SEMCTL`, `SEMGET` , `SEMIDS`, `SEMOP` ou `SEMTIMEDOP`, respectivamente. Como um resultado da sobrecarga de uma chamada do sistema para implementar várias chamadas do sistema, existe apenas um único par de testes syscall de semáforos System V: `syscall::semsys:entry` e `syscall::semsys:return` .

Chamadas do sistema de arquivo grande

Um programa de 32 bits que aceite *arquivos grandes* que excedam quatro gigabytes de tamanho devem ser capazes de processar deslocamentos de arquivo de 64 bits. Como os arquivos grandes requerem o uso de deslocamentos grandes, tais arquivos são manipulados através de um conjunto paralelo de interfaces do sistema, conforme descrito em `lf64(5)`. Estas interfaces são documentadas em `lf64`, mas elas não possuem páginas do manual individuais. Cada uma dessas interfaces de chamada do sistema de arquivo grande aparecem como seu próprio teste `syscall` , conforme mostrado na Tabela 21–1.

TABELA 21–1 Testes de arquivo grande `syscall`

Teste `syscall` de arquivo grande	**Chamada do sistema**
`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)`

TABELA 21–1 Testes de arquivo grande `syscall` *(Continuação)*

Teste `syscall` de arquivo grande	Chamada do sistema
`stat64`	`stat(2)`
`statvfs64`	`statvfs(2)`

Chamadas do sistema privadas

Algumas chamadas do sistema são detalhes de implementação privada de subsistemas do Solaris que expandem o limite do kernel do usuário. Em si, estas chamadas do sistema não possuem páginas do manual na Seção 2. Exemplos de chamadas do sistema nesta categoria incluem a chamada do sistema `signotify`, que é usada como parte da implementação das filas de mensagens POSIX.4, e a chamada do sistema `utssys`, que é usada para implementar `fuser(1M)`.

Argumentos

Para testes `entry`, os argumentos (`arg0` .. `arg`*n*) são os argumentos da chamada do sistema. Para testes `return`, `arg0` e `arg1` contêm o valor de retorno. Um valor diferente de zero na variável de D `errno` indica a falha da chamada do sistema.

Estabilidade

O provedor `syscall` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	Comum
Módulo	Privada	Privada	Desconhecida
Função	Instável	Instável	ISA
Nome	Desenvolvendo	Desenvolvendo	Comum
Argumentos	Instável	Instável	ISA

◆ ◆ ◆ CAPÍTULO 22

Provedor sdt

O provedor Statically Defined Tracing (SDT) cria testes nos locais que um programador de software designou formalmente. O mecanismo do SDT permite aos programadores escolher conscientemente os locais de interesse para usuários do DTrace e transmitir algum conhecimento semântico sobre cada local através do nome do teste. O kernel do Solaris definiu uma série de testes SDT, e provavelmente adicionará outros mais ao longo do tempo. O DTrace também fornece um mecanismo para desenvolvedores de aplicativos do usuário definirem testes estáticos, descrito no Capítulo 34, "Rastreio definido estaticamente em aplicativos do usuário".

Testes

Os testes do SDT definidos pelo kernel do Solaris estão listados na Tabela 22–1. A estabilidade do nome e a estabilidade dos dados destes testes são ambas Privadas, pois sua descrição aqui reflete a implementação do kernel e não deve ser considerada como um comprometimento de interface. Para obter mais informações sobre o mecanismo de estabilidade do DTrace, consulte "Estabilidade" na página 241.

TABELA 22–1 Testes SDT

Nome do teste	Descrição	arg0
callout-start	Teste que é acionado imediatamente antes da execução de um callout (consulte <sys/callo.h>). Os callouts são executados pelo relógio periódico do sistema e representam a implementação de timeout(9F).	O ponteiro para callout_t (consulte <sys/callo.h>) correspondente ao callout a ser executado.

TABELA 22–1 Testes SDT *(Continuação)*

Nome do teste	Descrição	arg0
`callout-end`	Teste que é acionado imediatamente depois da execução de um callout (consulte `<sys/callo.h>`).	O ponteiro para `callout_t` (consulte `<sys/callo.h>`) correspondente ao callout que acabou de ser executado.
`interrupt-start`	Teste que é acionado imediatamente antes da chamada do manipulador de interrupção do dispositivo.	O ponteiro para a estrutura `dev_info` (consulte `<sys/ddi_impldefs.h>`) correspondente ao dispositivo de interrupção.
`interrupt-complete`	Teste que é acionado imediatamente depois do retorno do manipulador de interrupção de um dispositivo.	O ponteiro para a estrutura `dev_info` (consulte `<sys/ddi_impldefs.h>`) correspondente ao dispositivo de interrupção.

Exemplos

O exemplo seguinte é um script para observar o comportamento do callout por segundo:

```
#pragma D option quiet

sdt:::callout-start
{
    @callouts[((callout_t *)arg0)->c_func] = count();
}

tick-1sec
{
    printa("%40a %10@d\n", @callouts);
    clear(@callouts);
}
```

Executar este exemplo revela os usuários freqüentes do `timeout(9F)` no sistema, conforme mostrado na saída seguinte:

```
# 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
```

A interface de timeout(9F) somente produz uma única expiração de temporizador. Os consumidores de `timeout()` que requerem a funcionalidade de temporizador de intervalo geralmente reinstalam seu timeout a partir do manipulador `timeout()`. O exemplo seguinte mostra este comportamento:

```
#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);
}
```

Executar este script e esperar alguns segundos antes de digitar Control-C resulta em uma saída semelhante ao exemplo seguinte:

```
# 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
```

A saída mostra que a `uhci_handle_root_hub_status_change()` no driver `uhci`(7D) representa o temporizador de intervalo mais curto no sistema: ele é chamado a cada tique-taque do relógio do sistema.

O teste `interrupt-start` pode ser usado para entender a atividade de interrupção. O exemplo seguinte mostra como quantizar o tempo gasto ao executar um manipulador de interrupção pelo nome do driver:

```
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);
}
```

Executar este script acima resultará numa saída semelhante ao exemplo seguinte:

```
# 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
```

Criando testes do SDT

Se você for um desenvolvedor de driver de dispositivo, talvez esteja interessado em criar seus próprios testes do SDT em seu driver do Solaris. O efeito de teste desativado do SDT é essencialmente o custo de várias instruções de máquina não operacionais. Portanto, você é encorajado a adicionar testes do SDT aos seus drivers de dispositivo, se necessário. A menos que estes testes afetem negativamente o desempenho, você pode deixá-los em seu código de envio.

Declarando testes

Os testes do SDT são declarados através das macros `DTRACE_PROBE` , `DTRACE_PROBE1`, `DTRACE_PROBE2`, `DTRACE_PROBE3` e `DTRACE_PROBE4` do `<sys/sdt.h>`. O nome do módulo e o nome da função de um teste baseado em um SDT correspondem ao módulo do kernel e à função do teste. O nome do teste depende do nome fornecido na macro `DTRACE_PROBE`*n*. Se o nome não contiver duas barras inferiores consecutivas (`__`), o nome do teste será conforme escrito na macro. Se o nome contiver quaisquer duas barras inferiores consecutivas, o nome do teste converterá as barras inferiores consecutivas em um único traço (`-`). Por exemplo, se uma macro `DTRACE_PROBE` especificar `transaction__start`, o teste do SDT será chamado de `transaction-start`. Esta substituição permite que o código de C forneça nomes de macro que não são identificadores válidos de C sem especificar uma seqüência.

O DTrace inclui o nome do módulo do kernel e o nome da função como parte da tupla que identifica o teste, sendo assim, você não precisa incluir esta informação no nome do teste para evitar colisões de espaço de nome. Você pode usar o comando `dtrace -l -P sdt -m` *módulo* em seu *módulo* do driver para listar os testes que instalou e os nomes completos que serão vistos pelos usuários do DTrace.

Argumentos de teste

Os argumentos de cada teste do SDT são os argumentos especificados na referência de macro `DTRACE_PROBE`*n* correspondente. O número de argumentos depende de qual macro foi usada para criar o teste: `DTRACE_PROBE1` especifica um argumento, `DTRACE_PROBE2` especifica dois argumentos, e assim por diante. Ao declarar seus testes do SDT, você pode minimizar seu efeito de teste desativado, não referenciando os ponteiros e não carregando a partir de variáveis globais nos argumentos do teste. O cancelamento de referência de ponteiro e o carregamento de variável global podem ser feitos com segurança em ações de D que ativam testes, sendo assim, os usuários do DTrace podem solicitar estas ações somente quando necessárias.

Estabilidade

O provedor SDT usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Privada	Privada	ISA
Argumentos	Privada	Privada	ISA

◆ ◆ ◆ CAPÍTULO 23

Provedor sysinfo

O provedor sysinfo disponibiliza testes que correspondem às estatísticas do kernel classificadas pelo nome sys. Como estas estatísticas fornecem a entrada dos utilitários de monitoramento do sistema como o mpstat(1M), o provedor sysinfo ativa a rápida exploração do comportamento anormal observado.

Testes

O provedor sysinfo disponibiliza testes que correspondem aos campos na estatística do kernel chamada sys: um teste fornecido por sysinfo é acionado imediatamente antes do valor sys correspondente ser incrementado. O exemplo seguinte mostra como exibir ambos os nomes e os valores atuais da estatística do sistema chamada sys usando o comando kstat(1M).

```
$ kstat -n sys
module: cpu                             instance: 0
name:   sys                             class:    misc
    bawrite                         123
    bread                           2899
    bwrite                          17995
...
```

Os testes do sysinfo estão descritos na Tabela 23–1.

TABELA 23–1 Testes do sysinfo

bawrite	Teste que é acionado sempre que um buffer está prestes a ser gravado assincronamente em um dispositivo.
bread	Teste que é acionado sempre que um buffer é fisicamente lido a partir de um dispositivo. bread é acionado *depois* que o buffer tiver sido solicitado do dispositivo, mas *antes* que o bloqueio impeça a sua conclusão.

TABELA 23–1 Testes do `sysinfo` *(Continuação)*

`bwrite`	Teste que é acionado sempre que um buffer está prestes a ser gravado em um dispositivo, seja síncrona *ou* assincronamente.
`idlethread`	Teste que é acionado sempre que uma CPU entra no loop inativo.
`intrblk`	Teste que é acionado sempre que um segmento de interrupção é bloqueado.
`inv_swtch`	Teste que é acionado sempre que um segmento em execução é forçado a desistir involuntariamente da CPU.
`lread`	Teste que é acionado sempre que um buffer é logicamente lido a partir de um dispositivo.
`lwrite`	Teste que é acionado sempre que um buffer é logicamente gravado a partir de um dispositivo.
`modload`	Teste que é acionado sempre que um módulo do kernel é carregado.
`modunload`	Teste que é acionado sempre que um módulo do kernel é descarregado.
`msg`	Teste que é acionado sempre que uma chamada do sistema `msgsnd(2)` ou `msgrcv(2)` é feita, mas antes que as operações da fila de mensagens tenha sido realizada.
`mutex_adenters`	Teste que é acionado sempre que é feita uma tentativa de adquirir o bloqueio adaptativo com proprietário. Se este teste for acionado, um dos `adaptive-block` do provedor `lockstat` ou testes do `adaptive-spin` também serão acionados. Consulte o Capítulo 18, "Provedor `lockstat`" para obter detalhes.
`namei`	Teste que é acionado sempre que uma pesquisa de nome é tentada no sistema de arquivos.
`nthreads`	Teste que é acionado sempre que um segmento é criado.
`phread`	Teste que é acionado sempre que uma leitura de E/S básica está prestes a ser realizada.
`phwrite`	Teste que é acionado sempre que uma gravação de E/S básica está prestes a ser realizada.
`procovf`	Teste que é acionado sempre que um novo processo não pode ser criado porque o sistema está fora das entradas de tabela do processo.
`pswitch`	Teste que é acionado sempre que uma CPU alterna entre executar um segmento para executar outro.
`readch`	Teste que é acionado depois de cada leitura bem sucedida, mas antes que o controle seja retornado para o segmento que está realizando a leitura. Uma leitura pode ocorrer através das chamadas do sistema `read(2)`, `readv(2)` ou `pread(2)` `arg0` contém o número de bytes que foram lidos com êxito.

TABELA 23–1 Testes do `sysinfo` *(Continuação)*

`rw_rdfails`	Teste que é acionado sempre que há uma tentativa de ler-bloquear um bloqueio de leitores/gravador quando o bloqueio já está sendo mantido por um gravador, ou desejado por um gravador. Se este teste for acionado, o teste `rw-block` do provedor `lockstat` também será acionado. Consulte o Capítulo 18, "Provedor `lockstat`" para obter detalhes.
`rw_wrfails`	Teste que é acionado sempre que há uma tentativa de gravar-bloquear um bloqueio de leitores/gravador quando o bloqueio é mantido por alguns leitores ou por outro gravador. Se este teste for acionado, o teste `rw-block` do provedor `lockstat` também será acionado. Consulte o Capítulo 18, "Provedor `lockstat`" para obter detalhes.
`sema`	Teste que é acionado sempre que uma chamada do sistema `semop(2)` é feita, mas antes que quaisquer operações de semáforo tenham sido realizadas.
`sysexec`	Teste que é acionado sempre que uma chamada do sistema `exec(2)` é feita.
`sysfork`	Teste que é acionado sempre que uma chamada do sistema `fork(2)` é feita.
`sysread`	Teste que é acionado sempre que uma chamada do sistema `read(2)`, `readv(2)` ou `pread(2)` é feita.
`sysvfork`	Teste que é acionado sempre que uma chamada do sistema `vfork(2)` é feita.
`syswrite`	Teste que é acionado sempre que uma chamada do sistema `write(2)`, `writev(2)` ou `pwrite(2)` é feita.
`trap`	Teste que é acionado sempre que uma interceptação do processador ocorre. Observe que alguns processadores, em particular variantes do UltraSPARC, manipulam algumas interceptações leves através de um mecanismo que não faz com que este teste seja acionado.
`ufsdirblk`	Teste que é acionado sempre que um bloqueio de diretório é lido a partir de um sistema de arquivos UFS. Consulte `ufs(7FS)` para obter detalhes sobre UFS.
`ufsiget`	Teste que é acionado sempre que um inode é recuperado. Consulte `ufs(7FS)` para obter detalhes sobre UFS.
`ufsinopage`	Teste que é acionado depois que um inode no núcleo *sem* quaisquer páginas de dados associadas tiver sido disponibilizado para reutilização. Consulte `ufs(7FS)` para obter detalhes sobre UFS.
`ufsipage`	Teste que é acionado depois que um inode no núcleo *com* quaisquer páginas de dados associadas tiver sido disponibilizado para reutilização. Este teste é acionado depois que as páginas de dados associadas forem movidas para o disco. Consulte `ufs(7FS)` para obter detalhes sobre UFS.
`writech`	Teste que é acionado depois de cada gravação bem sucedida, mas antes que o controle seja retornado para o segmento que está realizando a gravação. Uma leitura pode ocorrer através de chamadas do sistema `write(2)`, `writev(2)` ou `pwrite(2)`. `arg0` contém o número de bytes que foram gravados com êxito.

TABELA 23–1 Testes do `sysinfo` *(Continuação)*

`xcalls`	Teste que é acionado sempre que uma chamada cruzada está prestes a ser feita. Uma chamada cruzada é o mecanismo do sistema operacional de uma CPU para solicitar o funcionamento imediato de outra CPU.

Argumentos

Os argumentos para testes `sysinfo` são os seguintes:

`arg0`	O valor pelo qual as estatísticas devem ser incrementadas. Para a maioria dos testes, este argumento é sempre 1, mas para alguns testes este argumento pode ter outros valores.
`arg1`	Um ponteiro para o valor atual da estatística a ser incrementada. Este valor é uma quantidade de 64 bits que será incrementada pelo valor de `arg0`. Cancelar a referência deste ponteiro permitirá que os consumidores determinem a contagem atual das estatísticas correspondentes ao teste.
`arg2`	Um ponteiro para a estrutura `cpu_t` que corresponde à CPU na qual as estatísticas devem ser incrementadas. Esta estrutura é definida em `<sys/cpuvar.h>`, mas é parte da implementação do kernel e deve ser considerada Privada.

O valor de `arg0` é 1 para a maioria dos testes `sysinfo` . Entretanto, os testes `readch` e `writech` definem `arg0` como o número de bytes lidos ou gravados, respectivamente. Este recurso permite que você determine o tamanho das leituras por nome de executável, conforme mostrado no exemplo seguinte:

```
# 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
```

O provedor `sysinfo` define `arg2` como um ponteiro para `cpu_t`, uma estrutura interna para a implementação do kernel. Os testes `sysinfo` são acionados na CPU na qual as estatísticas estão sendo incrementadas. Use o membro `cpu_id` da estrutura `cpu_t` para determinar a CPU de seu interesse.

Exemplo

Examine a seguinte saída de `mpstat(1M)`:

```
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
```

A partir da saída acima, você pode concluir que o campo `xcal` parece estar muito alto, especialmente dada a inatividade relativa do sistema. `mpstat` determina o valor no campo `xcal`, examinando o campo `xcalls` da estatística do kernel `sys`. Portanto, esta aberração pode ser explorada facilmente, ativando o teste `sysinfo` de `xcalls`, conforme mostrado no exemplo seguinte:

```
# 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
```

A saída mostra onde procurar a origem das chamadas cruzadas. Alguns processos do `find(1)` estão causando a maioria das chamadas cruzadas. O script de D seguinte pode ser usado para entender o problema em maiores detalhes:

```
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;
}
```

Este script usa o provedor `syscall` para atribuir chamadas cruzadas de `find` para uma chamada do sistema específica. Algumas chamadas cruzadas, tais como aquelas resultantes de falhas de página, podem não emanar de chamadas do sistema. O script imprime "<none>" nestes casos. Executar o script resultará numa saída semelhante ao exemplo seguinte:

```
# dtrace -s ./find.d
 dtrace: script './find.d' matched 444 probes
^C
  <none>                                                            2
  lstat64                                                        2433
  getdents64                                                    14873
```

Esta saída indica que a maioria das chamadas cruzadas induzidas por `find` são, por sua vez, induzidas pelas chamadas do sistema `getdents(2)`. Uma exploração mais profunda dependerá da direção em que você deseja explorar. Se quiser entender por que os processos `find` estão fazendo chamadas a `getdents` , você poderia escrever um script de D para agregar em `ustack()` quando `find` induzir uma chamada cruzada. Se quiser entender por que as chamadas para `getdents` estão induzindo chamadas cruzadas, você poderia escrever um script de D para

agregar em `stack()` quando `find` induzir uma chamada cruzada. Qualquer que seja o seu próximo passo, a presença do teste `xcalls` permitiu que você descobrisse rapidamente a causa principal da saída de monitoramento incomum.

Estabilidade

O provedor `sysinfo` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA
Argumentos	Privada	Privada	ISA

◆ ◆ ◆ CAPÍTULO 24

Provedor vminfo

O provedor vminfo disponibiliza os teste que correspondem às estatísticas do kernel vm. Como estas estatísticas fornecem a entrada dos utilitários de monitoramento do sistema como o vmstat(1M), o provedor vminfo ativa a rápida exploração do comportamento anormal observado.

Testes

O provedor vminfo disponibiliza os testes que correspondem aos campos na estatística do kernel chamada vm: um teste fornecido por vmsinfo é acionado imediatamente antes que o valor vm correspondente seja incrementado. Para exibir ambos os nomes e os valores atuais da estatística do kernel chamada vm, use o comando kstat(1M), conforme mostrado no exemplo seguinte:

```
$ 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
        ...
```

Os testes do sysinfo estão descritos na Tabela 24–1.

TABELA 24–1 Testes do `vminfo`

`anonfree`	Teste que é acionado sempre que uma página anônima não modificada é liberada como parte da atividade de paginação. As páginas anônimas são aquelas que não estão associadas a um arquivo. A memória que contém tais páginas inclui a memória de pilha, a memória de empilhamento ou a memória obtida pelo mapeamento explícito de `zero`(7D).
`anonpgin`	Teste que é acionado sempre que uma página anônima é paginada a partir de um dispositivo de troca.
`anonpgout`	Teste que é acionado sempre que uma página anônima modificada é paginada para um dispositivo de troca.
`as_fault`	Teste que é acionado sempre que uma falha é detectada em uma página, sendo que a falha pode ser de proteção ou de copiar ao gravar.
`cow_fault`	Teste que é acionado sempre que uma falha de copiar ao gravar é detectada em uma página. `arg0` contém o número de páginas que são criadas como um resultado de copiar ao gravar.
`dfree`	Teste que é acionado sempre que uma página é liberada como um resultado da atividade de paginação. Sempre que `dfree` é acionado, exatamente um entre `anonfree`, `execfree` ou `fsfree` também será acionado subseqüentemente.
`execfree`	Teste que é acionado sempre que uma página executável não modificada é liberada como resultado de uma atividade de paginação.
`execpgin`	Teste que é acionado sempre que uma página executável é paginada a partir do armazenamento de apoio.
`execpgout`	Teste que é acionado sempre que uma página executável modificada é paginada para o armazenamento de apoio. A maior parte da paginação de páginas executáveis ocorre em termos de `execfree`. `execpgout` só poderá ser acionada, se uma página executável for modificada na memória, uma ocorrência incomum na maioria dos sistemas.
`fsfree`	Teste que é acionado sempre que uma página de dados do sistema de arquivos não modificada é liberada como parte da atividade de paginação.
`fspgin`	Teste que é acionado sempre que uma página do sistema de arquivos é paginada a partir do armazenamento de apoio.
`fspgout`	Teste que é acionado sempre que uma página do sistema de arquivos modificada é paginada para o armazenamento de apoio.
`kernel_asflt`	Teste que é acionado sempre que uma falha de página é detectada pelo kernel em uma página em seu próprio espaço de endereço. Sempre que `kernel_asflt` for acionado, ele será imediatamente precedido por um acionamento do teste `as_fault`.
`maj_fault`	Teste que é acionado sempre que é detectada uma falha de página que resulta na E/S de um armazenamento de apoio ou dispositivo de troca. Sempre que `maj_fault` for acionado, ele será imediatamente precedido de um acionamento do teste `pgin`.

TABELA 24–1 Testes do `vminfo` *(Continuação)*

`pgfrec`	Teste que é acionado sempre que uma página é reclamada da lista de páginas livres.
`pgin`	Teste que é acionado sempre que uma página é paginada a partir do armazenamento de apoio ou de um dispositivo de troca. Este teste é diferente de `maj_fault` no sentido de que `maj_fault` só é acionado quando uma página é paginada como resultado de uma falha de página. `pgin` é acionado toda vez que uma página é paginada, independentemente do motivo.
`pgout`	Teste que é acionado sempre que uma página é paginada para o armazenamento de apoio ou para um dispositivo de troca.
`pgpgin`	Teste que é acionado sempre que uma página é paginada a partir do armazenamento de apoio ou de um dispositivo de troca. A única diferença entre `pgpgin` e `pgin` é que `pgpgin` contém um número de páginas paginadas como `arg0`. `pgin` sempre contém 1 em `arg0`.
`pgpgout`	Teste que é acionado sempre que uma página é paginada para o armazenamento de apoio ou para um dispositivo de troca. A única diferença entre `pgpgout` e `pgout` é que `pgpgout` contém o número de páginas paginadas como `arg0`. (`pgout` sempre contém 1 em `arg0`.)
`pgrec`	Teste que é acionado sempre que uma página é reclamada.
`pgrrun`	Teste que é acionado sempre que o paginador é agendado.
`pgswapin`	Teste que é acionado sempre que as páginas de um processo trocado externamente são trocadas internamente. O número de páginas trocadas internamente está em `arg0`.
`pgswapout`	Teste que é acionado sempre que as páginas são trocadas externamente como parte de um processo de troca externa. O número de páginas trocadas externamente está em `arg0`.
`prot_fault`	Teste que é acionado sempre que uma falha de página é detectada devido a uma violação de proteção.
`rev`	Teste que é acionado sempre que o deamon da página começa uma nova revolução através de todas as páginas.
`scan`	Teste que é acionado sempre que o deamon da página examina uma página.
`softlock`	Teste que é acionado sempre que ocorre uma falha da página como parte da colocação de um bloqueio de software na página.
`swapin`	Teste que é acionado sempre que um processo trocado externamente é trocado internamente.
`swapout`	Teste que é acionado sempre que um processo é trocado externamente.
`zfod`	Teste que é acionado sempre que uma página preenchida com zeros é criada por demanda.

Argumentos

`arg0`	O valor pelo qual as estatísticas devem ser incrementadas. Para a maioria dos testes, este argumento é sempre 1, mas para alguns ele pode ter outros valores; estes testes são comentados na Tabela 24–1.
`arg1`	Um ponteiro para o valor atual da estatística a ser incrementada. Este valor é uma quantidade de 64 bits que será incrementada pelo valor de `arg0`. Cancelar a referência deste ponteiro permitirá que os consumidores determinem a contagem atual das estatísticas correspondentes ao teste.

Exemplo

Examine a seguinte saída 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
```

A coluna `pi` na saída acima indica o número de páginas paginadas. O provedor `vminfo` permite que você aprenda mais sobre a origem destas paginações, conforme mostrado no exemplo seguinte:

```
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
```

A saída mostra que um processo associado ao software StarOffice ™, `soffice.bin`, é responsável pela maioria das paginações. Para ter uma melhor idéia do `soffice.bin` em termos do comportamento da memória virtual, você poderia ativar todos os testes `vminfo` . O exemplo seguinte executa o `dtrace`(1M) enquanto inicia o software 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
```

O script de exemplo seguinte fornece mais informações sobre o comportamento da memória virtual do software StarOffice durante a sua inicialização:

```
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);
}
```

Execute o script ao iniciar novamente o software StarOffice. Em seguida, crie um novo desenho, crie uma nova apresentação e feche todos os arquivos e saia do aplicativo. Pressione Control-C no shell que está executando o script de D. Os resultados fornecem uma visão do comportamento da memória virtual ao longo do tempo:

```
# 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
```

A saída mostra o comportamento do StarOffice com relação ao sistema de memória virtual. Por exemplo, o teste `maj_fault` não foi acionado até que uma nova instância do aplicativo fosse iniciada. Como você poderia esperar, um "início quente" do StarOffice não resultou em novas falhas graves. A saída `as_fault` mostra uma seqüência ininterrupta inicial de atividades, latência quando o usuário localizou o menu para criar um novo desenho, outro período de inatividade, e uma seqüência ininterrupta de atividades finais quando o usuário clicou em uma

nova apresentação. A saída `zfod` mostra que a criação de uma nova apresentação induziu uma pressão significativa por páginas preenchidas com zero, mas somente por um curto período de tempo.

A próxima iteração da investigação do DTrace neste exemplo dependeria da direção em que você deseja explorar. Se quiser entender a origem das páginas preenchidas com zero, você poderia agregar `ustack()` em uma ativação de `zfod`. Talvez você queira estabelecer um limite para páginas preenchidas com zero e usar a ação destrutiva `stop()` para parar o processo incorreto quando o limite for excedido. Esta abordagem permitiria que você usasse mais ferramentas de depuração como `truss(1)` ou `mdb(1)`. O provedor `vminfo` permite que você associe as estatísticas vistas na saída das ferramentas convencionais como `vmstat(1M)` aos aplicativos que estão induzindo o comportamento do sistema.

Estabilidade

O provedor `vminfo` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA
Argumentos	Privada	Privada	ISA

◆ ◆ ◆ CAPÍTULO 25

Provedor proc

O provedor proc disponibiliza os testes que pertencem às seguintes atividades: criação e finalização de processo, criação e finalização de LWP, execução de novas imagens de programa e envio e tratamento de sinais.

Testes

Os testes proc são descritos na Tabela 25–1.

TABELA 25–1 Testes proc

Teste	Descrição
create	Teste que é acionado quando um processo é criado usando fork(2), forkall(2), fork1(2) ou vfork(2). A psinfo_t correspondente ao novo processo filho é apontada por args[0]. Você pode distinguir vfork das outras variantes de fork procurando por PR_VFORKPno membro pr_flag da lwpsinfo_t do segmento de fork. Você pode distinguir fork1 de forkall examinando os membros pr_nlwp de psinfo_t (curpsinfo) do processo pai e psinfo_t (args[0]) do processo filho. Como o teste create só é acionado depois que o processo é criado com êxito, e como a criação de LWP faz parte da criação de um processo, lwp-create será acionado para quaisquer LWPs criados durante a criação do processo *antes* que o teste create seja acionado no novo processo.

TABELA 25–1 Testes proc *(Continuação)*

Teste	Descrição
exec	Teste que é acionado sempre que um processo carregar uma nova imagem do processo com uma variante da chamada do sistema exec(2): exec(2), execle(2), execlp(2), execv(2), execve(2), execvp(2). O teste exec é acionado *antes* que a imagem do processo seja carregada. As variáveis de processo como execname e curpsinfo, portanto, contêm o estado do processo antes da imagem ser carregada. Algum tempo depois que o teste exec é acionado, o teste exec-failure ou o teste exec-success será, subseqüentemente, acionado no mesmo segmento. O caminho da nova imagem do processo é apontado por args[0].
exec-failure	Teste que é acionado quando uma variante exec(2) tiver falhado. O teste exec-failure é acionado somente depois que o teste exec tiver sido acionado no mesmo segmento. O valor de errno(3C) é fornecido em args[0] .
exec-success	Teste que é acionado quando uma variante exec(2) tiver sido bem-sucedida. Como o teste exec-failure, o teste exec-success é acionado somente depois que o teste exec é acionado no mesmo segmento. No momento em que o teste exec-success é acionado, variáveis de processo como execname e curpsinfo contêm o estado do processo depois que a nova imagem do processo tiver sido carregada.
exit	Teste que é acionado quando o processo atual está sendo encerrado. A razão do encerramento, que é expressada como um dos códigos SIGCHLD siginfo.h(3HEAD), está contida em args[0].
fault	Teste que é acionado quando uma falha de máquina ocorre em um segmento. O código da falha (conforme definido em proc(4)) está em args[0]. A estrutura siginfo correspondente à falha é apontada por args[1]. Somente as falhas que induzem a um sinal podem acionar o teste fault.
lwp-create	Teste que é acionado quando um LWP é criado, geralmente como resultado de thr_create(3C). A lwpsinfo_t correspondente ao novo segmento é apontada por args[0]. A psinfo_t do processo que contém o segmento é apontada por args[1].
lwp-start	Teste que é acionado no contexto de um LWP recém-criado. O teste lwp-start será acionado antes que quaisquer instruções no nível do usuário sejam executadas. Se o LWP for o primeiro LWP no processo, o teste start será acionado, seguido por lwp-start.
lwp-exit	Teste que é acionado quando um LWP está sendo encerrado, seja devido a um sinal ou a uma chamada explícita para thr_exit(3C).
signal-discard	Teste que é acionado quando um sinal é enviado para um processo de segmento único, e o sinal é desbloqueado e ignorado pelo processo. Sob essas condições, o sinal é descartado ao ser gerado. A lwpsinfo_t e a psinfo_t do processo e do segmento de destino estão, respectivamente, em args[0] e args[1]. O número do sinal está em args[2] .

TABELA 25–1 Testes proc *(Continuação)*

Teste	Descrição
signal-send	Teste que é acionado quando um sinal é enviado para um segmento ou processo. O teste signal-send é acionado no contexto do processo e do segmento de envio. A lwpsinfo_t e a psinfo_t do processo e do segmento de recebimento estão, respectivamente, em args[0] e args[1]. O número do sinal está em args[2] . signal-send é sempre seguida por signal-handle ou signal-clear no processo e no segmento de recebimento.
signal-handle	Teste que é acionado imediatamente antes de um teste manipular um sinal. O teste signal-handle é acionado no contexto do segmento que irá manipular o sinal. O número do sinal está em args[0] . Um ponteiro para a estrutura siginfo_t que corresponde ao sinal está em args[1]. O valor de args[1] é NULL se não houver nenhuma estrutura siginfo_t ou se o manipulador de sinal não tiver o sinalizador SA_SIGINFO definido. O endereço do manipulador de sinal no processo está em args[2].
signal-clear	Teste que é acionado quando um sinal pendente é cancelado porque o segmento de destino estava aguardando pelo sinal em sigwait(2), sigwaitinfo(3RT) ou sigtimedwait(3RT). Sob essas condições, o sinal pendente é cancelado e o número do sinal é retornado ao chamador. O número do sinal está em args[0] . signal-clear é acionado no contexto do segmento que estava aguardando anteriormente.
start	Teste que é acionado no contexto de um processo recém-criado. O teste start será acionado antes que quaisquer instruções no nível do usuário sejam executadas no processo.

Argumentos

Os tipos de argumentos dos testes proc são listados na Tabela 25–2. Os argumentos são descritos na Tabela 25–1.

TABELA 25–2 Argumentos do teste proc

Teste	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 *	—

TABELA 25–2 Argumentos do teste `proc` *(Continuação)*

Teste	`args[0]`	`args[1]`	`args[2]`
`lwp-start`	—	—	—
`lwp-exit`	—	—	—
`signal-discard`	`lwpsinfo_t *`	`psinfo_t *`	`int`
`signal-discard`	`lwpsinfo_t *`	`psinfo_t *`	`int`
`signal-send`	`lwpsinfo_t *`	`psinfo_t *`	`int`
`signal-handle`	`int`	`siginfo_t *`	`void (*)(void)`
`signal-clear`	`int`	—	—
`start`	—	—	—

`lwpsinfo_t`

Vários testes `proc` possuem argumentos do tipo `lwpsinfo_t`, uma estrutura documentada em `proc`(4). A definição da estrutura `lwpsinfo_t` conforme disponível para os consumidores do DTrace é a seguinte:

```
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;
```

O campo `pr_flag` é uma máscara de bits com sinalizadores que descrevem o processo. Esses sinalizadores e seus significados são descritos na Tabela 25–3.

TABELA 25–3 Valores de `pr_flag`

`PR_ISSYS`	O processo é um processo do sistema.

TABELA 25–3 Valores de pr_flag *(Continuação)*

PR_VFORKP	O processo é pai de um filho de vfork(2).
PR_FORK	O processo possui o modo herança-em-bifurcação definido.
PR_RLC	O processo possui o modo execução-em-último-encerramento definido.
PR_KLC	O processo possui o modo eliminação-em-último-encerramento definido.
PR_ASYNC	O processo possui o modo interrupção-assíncrona definido.
PR_MSACCT	O processo possui a contabilidade de micro-estado ativada.
PR_MSFORK	A contabilidade do micro-estado do processo é herdada na bifurcação.
PR_BPTADJ	O processo possui o modo de ajuste de ponto de interrupção definido.
PR_PTRACE	O processo possui o modo de compatibilidade com ptrace(3C) definido.
PR_STOPPED	O segmento é um LWP que está interrompido.
PR_ISTOP	O segmento é um LWP interrompido em um evento de interesse.
PR_DSTOP	O segmento é um LWP que possui uma diretiva de interrupção em efeito.
PR_STEP	O segmento é um LWP que possui uma diretiva de etapa única em efeito.
PR_ASLEEP	O segmento é um LWP em uma espera que pode ser interrompida em uma chamada do sistema.
PR_DETACH	O segmento é um LWP desanexado. Consulte pthread_create(3C) e pthread_join(3C).
PR_DAEMON	O segmento é um LWP daemon. Consulte pthread_create(3C).
PR_AGENT	O segmento é o LWP do agente do processo.
PR_IDLE	O segmento é o segmento inativo de uma CPU. Segmentos inativos somente são executados em uma CPU quando as filas de execução da CPU estão vazias.

O campo pr_addr é o endereço de uma estrutura de dados no kernel, privada, que representa o segmento. Embora a estrutura de dados seja privada, o campo pr_addr pode ser usado como um símbolo exclusivo de um segmento enquanto o segmento existir.

O campo pr_wchan é definido quando o segmento está em modo de espera em um objeto de sincronização. O significado do campo pr_wchan é privado para a implementação do kernel, mas o campo pode ser usado como um símbolo exclusivo para o objeto de sincronização.

O campo pr_stype é definido quando o segmento está em modo de espera em um objeto de sincronização. Os possíveis valores do campo pr_stype estão na Tabela 25–4.

TABELA 25–4 Valores de `pr_stype`

`SOBJ_MUTEX`	Objeto de sincronização do mutex do kernel. Usado para serializar o acesso a regiões de dados compartilhados no kernel. Consulte o Capítulo 18, "Provedor `lockstat`" e `mutex_init`(9F) para obter detalhes sobre objetos de sincronização do mutex no kernel.
`SOBJ_RWLOCK`	Objeto de sincronização de leitores/gravador do kernel. Usado para sincronizar o acesso aos objetos compartilhados no kernel que pode permitir vários leitores simultâneos ou um único gravador. Consulte o Capítulo 18, "Provedor `lockstat`" e `rwlock`(9F) para obter detalhes sobre objetos de sincronização de leitores/gravador no kernel.
`SOBJ_CV`	Objeto de sincronização de variável de condição. Uma variável de condição aguarda indefinidamente até que alguma condição torne-se verdadeira. Variáveis de condição geralmente são usadas para sincronizar por razões diferentes do acesso a uma região de dados compartilhados, e são o mecanismo geralmente usado quando um processo realiza uma espera indefinida direcionada pelo programa. Por exemplo, o bloqueio em `poll`(2), `pause`(2), `wait`(3C) e semelhantes.
`SOBJ_SEMA`	Objeto de sincronização de semáforo. Um objeto de sincronização de vários objetivos que, assim como os objetos de variável de condição, não controla uma noção de propriedade. Como a propriedade é necessária para implementar a herança de prioridade no kernel do Solaris, a falta de propriedade inerente em objetos de semáforo impede seu uso amplo. Consulte `semaphore`(9F) para obter detalhes.
`SOBJ_USER`	Objeto de sincronização no nível do usuário. Todos os bloqueios em objetos de sincronização no nível do usuário são manipulados com objetos de sincronização `SOBJ_USER` . Os objetos de sincronização no nível do usuário incluem aqueles criados com `mutex_init`(3C), `sema_init`(3C), `rwlock_init`(3C), `cond_init`(3C) e seus equivalentes POSIX.
`SOBJ_USER_PI`	Objeto de sincronização no nível do usuário que implementa a herança de prioridade. Alguns objetos de sincronização no nível do usuário que controlam a propriedade adicionalmente permitem a herança de prioridade. Por exemplo, objetos do mutex criados com `pthread_mutex_init`(3C) podem ser feitos para herdar prioridade usando-se `pthread_mutexattr_setprotocol`(3C).
`SOBJ_SHUTTLE`	Objeto de sincronização de deslocamento. Objetos de deslocamento são usados para implementar portas. Consulte door_create(3DOOR) para obter mais informações.

O campo `pr_state` é definido como um dos valores na Tabela 25–5. O campo `pr_sname` é definido como um caractere correspondente mostrado entre parênteses na mesma tabela.

TABELA 25–5 Valores de `pr_state`

`SSLEEP` (`S`)	O segmento está em espera. O teste `sched:::sleep` será acionado imediatamente antes que o estado de um segmento passe para `SSLEEP`.
`SRUN` (`R`)	O segmento é executável, mas não está sendo executado no momento. O teste `sched:::enqueue` será acionado imediatamente antes que o estado de um segmento passe para `SRUN`.
`SZOMB` (`Z`)	O segmento é um LWP zumbi.
`SSTOP` (`T`)	O segmento é interrompido, seja devido a uma diretiva `proc(4)` explícita ou a algum outro mecanismo de interrupção.
`SIDL` (`I`)	O segmento é um estado intermediário durante a criação do processo.
`SONPROC` (`O`)	O segmento está sendo executado em uma CPU. O teste `sched:::on-cpu` será acionado no contexto do segmento `SONPROC` logo depois que o estado do segmento passar para `SONPROC`.

psinfo_t

Vários testes `proc` possuem um argumento do tipo `psinfo_t`, uma estrutura documentada em `proc(4)`. A definição da estrutura de `psinfo_t` conforme disponível aos consumidores do DTrace é a seguinte:

```
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;
```

O campo `pr_dmodel` é definido como `PR_MODEL_ILP32` , indicando um processo de 32 bits, ou `PR_MODEL_LP64`, indicando um processo de 64 bits.

Exemplos

exec

Você pode usar o teste `exec` para determinar facilmente quais programas estão sendo executados, e por quem, conforme mostrado no seguinte exemplo:

```
#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", @);
}
```

A execução do script de exemplo por um curto período de tempo em uma máquina de compilação produz um resultado similar ao seguinte exemplo:

```
# 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` e `exit`

Se você desejar saber o tempo de execução dos programas da criação ao encerramento, poderá ativar os testes `start` e `exit`, conforme mostrado no exemplo a seguir:

```
proc:::start
{
    self->start = timestamp;
}

proc:::exit
/self->start/
{
    @[execname] = quantize(timestamp - self->start);
    self->start = 0;
}
```

A execução do script de exemplo no servidor de compilação por vários segundos produz um resultado similar ao seguinte exemplo:

```
# 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 e lwp-exit

Em vez de saber quanto tempo um processo em particular demora para ser executado, talvez você queira saber quanto tempo segmentos individuais demoram para ser executados. O exemplo a seguir mostra como usar os testes `lwp-start` e `lwp-exit` com esta finalidade:

```
proc:::lwp-start
/tid != 1/
{
    self->start = timestamp;
}

proc:::lwp-exit
/self->start/
{
    @[execname] = quantize(timestamp - self->start);
    self->start = 0;
}
```

A execução do script de exemplo em um servidor de calendário e NFS produz um resultado similar ao seguinte exemplo:

```
# 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

Você pode usar o teste `signal-send` para determinar o processo de envio e recebimento associado a qualquer sinal, conforme mostrado no exemplo a seguir:

```
#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", @);
}
```

Executar este script acima resultará numa saída semelhante ao exemplo seguinte:

```
# 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
```

Estabilidade

O provedor `proc` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela a seguir. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA
Argumentos	Desenvolvendo	Desenvolvendo	ISA

◆ ◆ ◆ CAPÍTULO 26

Provedor sched

O provedor sched disponibiliza os testes relacionados ao agendamento de CPU. Como as CPUs são o único recurso que todos os segmentos devem consumir, o provedor sched é muito útil para a compreensão do comportamento sistêmico. Por exemplo, ao usar o provedor sched, você pode compreender quando e por que os segmentos entram em espera, são executados, mudam de prioridade ou ativam outros segmentos.

Testes

Os testes sched são descritos na Tabela 26–1.

TABELA 26–1 Testes sched

Teste	Descrição
change-pri	Teste que é acionado sempre que a prioridade de um segmento está para ser alterada. A lwpsinfo_t do segmento é apontada por args[0] . A prioridade atual do segmento está no campo pr_pri desta estrutura. A psinfo_t do processo que contém o segmento é apontada por args[1]. A nova prioridade do segmento está contida em args[2].
dequeue	Teste que é acionado imediatamente antes de um segmento executável ser desenfileirado de uma fila de execução. A lwpsinfo_t do segmento que está sendo desenfileirado é apontada por args[0]. A psinfo_t do processo que contém o segmento é apontada por args[1]. A cpuinfo_t da CPU da qual o segmento está sendo desenfileirado é apontada por args[2]. Se o segmento estiver sendo desenfileirado de uma fila de execução que não esteja associada a uma CPU em particular, o membro cpu_id dessa estrutura será -1.

TABELA 26–1 Testes sched *(Continuação)*

Teste	Descrição
enqueue	Teste que é acionado imediatamente antes de um segmento executável ser enfileirado em uma fila de execução. A `lwpsinfo_t` do segmento que está sendo enfileirado é apontada por `args[0]`. A `psinfo_t` do processo que contém o segmento é apontada por `args[1]`. A `cpuinfo_t` da CPU na qual o segmento está sendo enfileirado é apontada por `args[2]`. Se o segmento estiver sendo enfileirado em uma fila de execução que não esteja associada a uma CPU em particular, o membro `cpu_id` dessa estrutura será `-1`. O valor em `args[3]` é um booleano que indica se o segmento será enfileirado na frente da fila de execução. O valor será diferente de zero se o segmento for enfileirado na frente da fila de execução, e zero se o segmento for enfileirado na parte de trás da fila de execução.
off-cpu	Teste que é acionado quando a CPU atual estiver prestes a encerrar a execução de um segmento. A variável `curcpu` indica a CPU atual. A variável `curlwpsinfo` indica o segmento que está encerrando a execução. A variável `curpsinfo` descreve o processo que contém o segmento atual. A estrutura `lwpsinfo_t` do segmento que a CPU atual executará em seguida é apontada por `args[0]` . A `psinfo_t` do processo que contém o próximo segmento é apontada por `args[1]`.
on-cpu	Teste que é acionado quando uma CPU inicia a execução de um segmento. A variável `curcpu` indica a CPU atual. A variável `curlwpsinfo` indica o segmento que está iniciando a execução. A variável `curpsinfo` descreve o processo que contém o segmento atual.
preempt	Teste que é acionado imediatamente antes que o segmento atual seja superado. Depois que esse teste for acionado, o segmento atual irá selecionar um segmento a ser executado e o teste `off-cpu` será acionado no segmento atual. Em alguns casos, um segmento em uma CPU será superado, mas o segmento de maior prioridade será executado em outra CPU enquanto isso. Nessa situação, o teste `preempt` será acionado, mas o distribuidor não conseguirá localizar um segmento de maior prioridade para ser executado e o teste `remain-cpu` será acionado em vez do teste `off-cpu`.
remain-cpu	Teste que é acionado quando uma decisão de agendamento tiver sido tomada, mas o distribuidor tiver optado por continuar executando o segmento atual. A variável `curcpu` indica a CPU atual. A variável `curlwpsinfo` indica o segmento que está iniciando a execução. A variável `curpsinfo` descreve o processo que contém o segmento atual.

TABELA 26–1 Testes sched *(Continuação)*

Teste	Descrição
schedctl-nopreempt	Teste que é acionado quando um segmento é superado e depois enfileirado novamente na *frente* da fila de execução devido a uma solicitação de controle de preempção. Consulte schedctl_init(3C) para obter detalhes sobre controle de preempção. Assim como com preempt, off-cpu ou remain-cpu será acionado após schedctl-nopreempt. Como schedctl-nopreempt indica um re-enfileiramento do segmento atual na frente da fila de execução, é mais provável que remain-cpu seja acionado após schedctl-nopreempt do que off-cpu. A lwpsinfo_t do segmento que está sendo superado é apontada por args[0]. A psinfo_t do processo que contém o segmento é apontada por args[1].
schedctl-preempt	Teste que é acionado quando um segmento que está usando o controle de preempção é, ainda sim, superado e enfileirado novamente na *parte de trás* da fila de execução. Consulte schedctl_init(3C) para obter detalhes sobre controle de preempção. Assim como com preempt, off-cpu ou remain-cpu será acionado após schedctl-preempt. Assim como preempt (e diferente de schedctl-nopreempt), schedctl-preempt indica o re-enfileiramento do segmento atual na parte de trás da fila de execução. Como resultado, é mais provável que off-cpu seja acionado após schedctl-preempt do que remain-cpu. A lwpsinfo_t do segmento que está sendo superado é apontada por args[0]. A psinfo_t do processo que contém o segmento é apontada por args[1].
schedctl-yield	Teste que é acionado quando um segmento que tinha o controle de preempção ativado e seu fracionamento de tempo estendido artificialmente executou o código para ceder a CPU para outros segmentos.
sleep	Teste que é acionado imediatamente antes de o segmento atual entrar em espera em um objeto de sincronização. O tipo do objeto de sincronização está contido no membro pr_stype da lwpsinfo_t apontada por curlwpsinfo . O endereço do objeto de sincronização está contido no membro pr_wchan da lwpsinfo_t apontada por curlwpsinfo. O significado desse endereço é um detalhe de implementação privado, mas o valor de endereço deve ser tratado como um símbolo exclusivo do objeto de sincronização.
surrender	Teste que é acionado quando uma CPU tiver sido instruída por outra CPU para tomar uma decisão de agendamento; geralmente porque um segmento de prioridade maior se tornou executável.
tick	Teste que é acionado como parte de uma contagem baseada no tique-taque do relógio. Na contagem baseada no tique-taque do relógio, a contagem de CPU é realizada examinando-se quais segmentos e processos estão sendo executados quando uma interrupção de intervalo fixo é acionada. A lwpsinfo_t que corresponde ao segmento ao qual está sendo atribuído o tempo de CPU é apontada por args[0]. A psinfo_t que corresponde ao processo que contém o segmento é apontada por args[1] .

TABELA 26–1 Testes sched *(Continuação)*

Teste	Descrição
wakeup	Teste que é acionado imediatamente antes do segmento atual ativar um segmento em espera em um objeto de sincronização. A lwpsinfo_t do segmento em espera é apontada por args[0]. A psinfo_t do processo que contém o segmento em espera é apontada por args[1]. O tipo do objeto de sincronização está contido no membro pr_stype da lwpsinfo_t do segmento em espera. O endereço do objeto de sincronização está contido no membro >pr_wchan da lwpsinfo_t do segmento em espera. O significado desse endereço é um detalhe de implementação privado, mas o valor de endereço deve ser tratado como um símbolo exclusivo do objeto de sincronização.

Argumentos

Os tipos de argumento dos testes sched estão listados na Tabela 26–2; os argumentos são descritos na Tabela 26–1.

TABELA 26–2 Argumentos do teste sched

Teste	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 *	—	—

Conforme indica a Tabela 26–2, muitos testes sched possuem argumentos que consistem em um ponteiro para uma lwpsinfo_t e um ponteiro para uma psinfo_t, indicando um segmento

e o processo que contém o segmento, respectivamente. Essas estruturas são descritas detalhadamente em "lwpsinfo_t" na página 262 e "psinfo_t" na página 265, respectivamente.

cpuinfo_t

A estrutura cpuinfo_t define uma CPU. Conforme indica a Tabela 26–2, os argumentos para ambos os testes enqueue e dequeue incluem um ponteiro para uma cpuinfo_t. Além disso, a cpuinfo_t correspondente à CPU atual é apontada pela variável curcpu. A definição da estrutura de cpuinfo_t é a seguinte:

```
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;
```

O membro cpu_id é o identificador de processador, conforme retornado por psrinfo(1M) e p_online(2).

O membro cpu_pset é o conjunto de processadores que contém a CPU, se houver alguma. Consulte psrset(1M) para obter mais detalhes sobre conjuntos de processadores.

O membro cpu_chip é o identificador do chipe físico. Chipes físicos podem conter várias CPUs. Consulte psrinfo(1M) para obter mais informações.

O membro cpu_lgrp é o identificador do grupo de latência associado à CPU. Consulte liblgrp(3LIB) para obter detalhes sobre grupos de latência.

O membro cpu_info é a estrutura de processor_info_t associada à CPU, conforme retornado por processor_info(2).

Exemplos

on-cpu e off-cpu

Uma pergunta comum que você pode fazer é quais CPUs estão executando segmentos e por quanto tempo. Você pode usar os testes on-cpu e off-cpu para responder facilmente a essa questão considerando todo o sistema, conforme mostrado no exemplo a seguir:

```
sched:::on-cpu
{
    self->ts = timestamp;
```

```
}

sched:::off-cpu
/self->ts/
{
    @[cpu] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

Executar o script acima resultará numa saída semelhante ao exemplo seguinte:

```
# 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
```

O resultado acima mostra que na CPU 1 os segmentos tendem a ser executados por menos de 100 microssegundos em um período, ou por aproximadamente 10 milissegundos. Um intervalo evidente entre os dois clusters de dados mostrados no histograma. Talvez você também esteja interessado em saber quais CPUs estão executando um processo em particular. Você também pode usar os testes `on-cpu` e `off-cpu` para responder a essa pergunta. O script a seguir exibe quais CPUs executam um aplicativo específico durante um período de dez segundos:

```
#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", @);
}
```

A execução do script acima em um grande servidor de correio e a especificação do daemon IMAP produz um resultado similar ao exemplo a seguir:

```
# 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
```

O Solaris leva em consideração quanto tempo um segmento está em espera ao selecionar uma CPU na qual executar o segmento: um segmento que esteja em espera por menos tempo tende a não migrar. Você pode usar os testes `off-cpu` e `on-cpu` para observar esse comportamento:

```
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;
}
```

A execução do script acima por aproximadamente 30 segundos produz um resultado similar ao seguinte exemplo:

```
# 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
```

O resultado de exemplo mostra que há muito mais ocorrências de não-migração do que de migração. Além disso, quando os períodos de espera são maiores, as migrações são mais prováveis. As distribuições são evidentemente diferentes no sub-intervalo de 100 milissegundos, mas tornam-se muito similares à medida que os tempos de espera aumentam. Esse resultado poderia indicar que o tempo de espera não é calculado na decisão de agendamento quando um determinado limite é excedido.

O exemplo final usando `off-cpu` e `on-cpu` mostra como usar estes testes com o campo `pr_stype` para determinar por que os segmentos estão em espera e por quanto tempo:

```
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;
}
```

A execução do script acima por vários segundos produz um resultado similar ao seguinte exemplo:

```
# 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 e dequeue

Quando uma CPU fica inativa, o distribuidor procura trabalhos enfileirados em outras CPUs (ativas). O exemplo a seguir usa o teste dequeue para entender com que freqüência os aplicativos são transferidos e por qual 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", @);
}
```

A parte final do resultado da execução do script acima em um sistema de 4 CPUs produz um resultado similar ao seguinte exemplo:

```
# 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
```

Em vez de saber quais CPUs pegaram qual tarefa, talvez você queira saber as CPUs nas quais os processos e segmentos estão aguardando para ser executados. Você pode usar os testes enqueue e dequeue juntos para responder a essa questão:

```
sched:::enqueue
{
    self->ts = timestamp;
}

sched:::dequeue
/self->ts/
{
    @[args[2]->cpu_id] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

A execução do script acima por vários segundos produz um resultado similar ao seguinte exemplo:

```
# 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
```

Observe os valores diferentes de zero na parte inferior do resultado de exemplo. Esses pontos de dados revelam várias instâncias em ambas as CPUs nas quais um segmento foi enfileirado para ser executado por vários *segundos*.

Em vez de procurar por tempos de espera, talvez você queira examinar o tamanho da fila de execução ao longo do tempo. Ao usar os testes `enqueue` e `dequeue`, você pode definir uma matriz de associação para controlar o tamanho da fila:

```
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]—;
}
```

A execução do script acima por aproximadamente 30 segundos em um sistema de laptop com um processador totalmente inativo produz um resultado similar ao seguinte exemplo:

```
# 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
```

O resultado é aproximadamente o que você esperaria de um sistema inativo: na maior parte do tempo que um segmento executável estava enfileirado, a fila de execução estava muito curta (com três ou menos segmentos). Entretanto, como o sistema estava totalmente inativo, os

pontos de dados excepcionais na parte inferior da tabela poderiam ser inesperados. Por exemplo, por que a fila de execução tinha 13 segmentos executáveis? Para explorar essa questão, você pode escrever um script em D que exiba o conteúdo da fila de execução quando a fila de execução for longa. Esse problema é complicado porque as habilitações em D não podem ser repetidas em estruturas de dados, e, portanto, não podem simplesmente ser repetidas em toda a fila de execução. Mesmo que as habilitações em D pudessem fazer isso, você deve evitar dependências nas estruturas internas de dados do kernel.

Para esse tipo de script, você ativaria os testes `enqueue` e `dequeue` e usaria matrizes de associação e de especulação. Sempre que um segmento é enfileirado, o script incrementa o tamanho da fila e registra o carimbo de data/hora em uma matriz de associação inserida pelo segmento. Você não pode usar uma variável de segmento local neste caso porque um segmento pode ser enfileirado por outro segmento. O script depois verifica se o tamanho da fila excede o máximo permitido. Caso exceda, o script iniciará uma nova especulação, e registrará o carimbo de data/hora e o novo máximo. Em seguida, quando um segmento é desenfileirado, o script compara o carimbo de data/hora do enfileiramento com o carimbo de data/hora do maior tamanho: se o segmento tiver sido enfileirado *antes* do carimbo de data/hora do maior tamanho, o segmento estava na fila quando o maior tamanho foi registrado. Nesse caso, o script especulativamente rastreia as informações do segmento. Depois que o kernel desenfileira o último segmento que estava enfileirado no carimbo de data/hora do maior tamanho, o script compromete os dados de especulação. Esse script é mostrado abaixo:

```
#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;
}
```

A execução do script acima no mesmo laptop com um processador produz um resultado similar ao seguinte exemplo:

```
# 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
```

O resultado revela que as filas de execução longas são devido a muitos processos `xterm` executáveis. Esse experimento coincidiu com uma alteração na área de trabalho virtual, e, portanto, os resultados provavelmente são devido a algum tipo de processamento de evento X.

sleep e wakeup

Em "enqueue e dequeue" na página 287, o exemplo final demonstrou que um crescimento súbito da fila de execução foi devido a processos `xterm` executáveis. Uma hipótese é que as observações resultaram de uma alteração na área de trabalho virtual. Você pode usar o teste `wakeup` para explorar essa hipótese determinando quem está ativando os processos `xterm`, e quando, conforme mostrado no exemplo a seguir:

```
#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);
}
```

Para investigar a hipótese, execute o script acima, aguardando aproximadamente cinco segundos, e alterne a área de trabalho virtual exatamente uma vez. Se o crescimento súbito de processos `xterm` executáveis for devido à alternância da área de trabalho virtual, o resultado deveria mostrar um crescimento súbito de atividade de ativação na marca de cinco segundos.

```
# dtrace -s ./xterm.d

  Xsun

           value  ------------- Distribution ------------ count
               4 |                                       0
               5 |@                                      1
               6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 32
               7 |                                       0
```

O resultado mostra que o servidor X está ativando processos xterm, agrupados por volta da hora que você alternou as áreas de trabalho virtuais. Se você desejasse compreender a interação entre o servidor X e os processos `xterm`, poderia agregar rastreios de pilha do usuário quando o servidor X acionasse o teste `wakeup`.

Para se compreender o desempenho dos sistemas cliente/servidor como o sistema de janelas X, deve-se compreender os clientes em nome dos quais o servidor está realizando trabalho. Esse tipo de pergunta é difícil de ser respondida com ferramentas de análise de desempenho convencionais. Entretanto, se você tiver um modelo no qual um cliente envie uma mensagem para o servidor e entre em espera com o processamento do servidor pendente, você poderá usar o teste `wakeup` para determinar o cliente para o qual a solicitação está sendo realizada, conforme mostrado no exemplo a seguir:

```
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;
}
```

Executar o script acima resultará numa saída semelhante ao exemplo seguinte:

```
dtrace -s ./xwork.d
dtrace: script './xwork.d' matched 14 probes
^C
  xterm                                                    9522510
  soffice.bin                                              9912594
  fvwm2                                                  100423123
  MozillaFirebird                                        312227077
  acroread                                               345901577
```

Este resultado revela que a maior parte do trabalho de `Xsun` está sendo feita em nome dos processos `acroread`, `MozillaFirebird` e, em menor grau, `fvwm2`. Observe que o script só examinou ativações de objetos de sincronização de variável de condição (`SOBJ_CV`). Conforme descrito na Tabela 25–4, as variáveis de condição são o tipo de objeto de sincronização geralmente usado para sincronizar por razões diferentes do acesso a uma região de dados compartilhados. No caso do servidor X, um cliente irá aguardar por dados em um pipe entrando em espera em uma variável de condição.

Você pode adicionalmente usar o teste `sleep` junto com o teste `wakeup` para compreender quais aplicativos estão bloqueando quais aplicativos, e por quanto tempo, conforme mostrado no seguinte exemplo:

```
#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", @);
}
```

A parte final do resultado da execução do script de exemplo por vários segundos em um sistema de área de trabalho parece com o seguinte exemplo:

```
# 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
```

Talvez você queira entender como e por que o `MozillaFirebird` está se bloqueando. Você pode modificar o script acima como mostrado no exemplo a seguir para responder a essa pergunta:

```
#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", @);
}
```

A execução do script modificado por vários segundos produz um resultado similar ao seguinte exemplo:

```
# 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
```

Você também pode usar os testes `sleep` e `wakeup` para compreender o desempenho dos servidores de porta como o daemon do cache de serviço de nome, conforme mostrado no exemplo a seguir:

```
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;
}
```

A parte final do resultado da execução do script acima em um servidor de correio grande parece com o seguinte exemplo:

```
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
```

Você pode estar interessado nos pontos de dados incomuns de `automountd` ou os pontos de dados persistentes de mais de um milissegundo de `sendmail`. Você pode incluir predicados adicionais no script acima para focalizar nas causas de quaisquer resultados excepcionais ou anormais.

`preempt,remain-cpu`

Como o Solaris é um sistema preemptivo, os segmentos de prioridade mais alta têm precedência sobre os de prioridade mais baixa. A preempção pode induzir a uma bolha de latência significativa no segmento de prioridade mais baixa. Por isso, talvez você queira saber quais segmentos estão sendo superados por quais segmentos. O exemplo a seguir mostra como usar os testes `preempt` e `remain-cpu` para exibir estas informações:

```
#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", @);
}
```

A execução do script acima por vários segundos em um sistema de área de trabalho produz um resultado similar ao seguinte exemplo:

```
# 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

A preempção é baseada em prioridades, por isso, talvez você queira observar alterações na prioridade ao longo do tempo. O exemplo a seguir usa o teste `change-pri` para exibir estas informações:

```
sched:::change-pri
{
    @[stringof(args[0]->pr_clname)] =
        lquantize(args[2] - args[0]->pr_pri, -50, 50, 5);
}
```

O script de exemplo captura o grau para o qual a prioridade é elevada ou diminuída e agrega por classe de agendamento. Executar o script acima resultará numa saída semelhante ao exemplo seguinte:

```
# 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
```

O resultado mostra a manipulação de prioridade da classe de agendamento Interactive (IA). Em vez de ver a *manipulação* de prioridade, talvez você queira ver os *valores* de prioridade de um determinado processo e segmento ao longo do tempo. O script a seguir usa o teste `change-pri` para exibir estas informações:

```
#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);
}
```

Para ver as alterações em prioridades ao longo do tempo, digite o seguinte comando em uma janela:

```
$ echo $$
139208
$ while true ; do let i=0 ; done
```

Em outra janela, execute o script e redirecione o resultado para um arquivo:

```
# dtrace -s ./pritime.d 139208 1 > /tmp/pritime.out
#
```

Você pode usar o arquivo `/tmp/pritime.out` que é gerado acima como entrada para um software de plotagem para exibir a prioridade graficamente ao longo do tempo. O `gnuplot` é um pacote de plotagem disponível livremente que está incluído no CD complementar do Solaris Freeware. Por padrão, o `gnuplot` é instalado em `/opt/sfw/bin`.

tick

O Solaris usa uma *contagem de CPU baseada em tique-taque*, na qual uma interrupção do relógio do sistema é acionada em um intervalo fixo e atribui a utilização da CPU aos segmentos e processos em execução no momento do tique-taque. O exemplo a seguir mostra como usar o teste `tick` para observar essa atribuição:

```
# 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
```

A freqüência do relógio do sistema varia de acordo com o sistema operacional, mas geralmente vai de 25 hertz a 1024 hertz. A freqüência do relógio do sistema é ajustável, mas o padrão é 100 hertz.

O teste `tick` só é acionado se o relógio do sistema detecta um segmento executável. Para usar o teste `tick` para observar a freqüência do relógio do sistema, você deve ter um segmento que seja sempre executável. Em uma janela, crie um shell de looping conforme mostrado no exemplo a seguir:

```
$ while true ; do let i=0 ; done
```

Em outra janela, execute o seguinte script:

```
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
```

O intervalo mínimo é de 9,8 milissegundos, o que indica que a freqüência de tique-taque padrão do relógio é de 10 milissegundos (100 hertz). O mínimo observado é algo menos que 10 milissegundos devido à variação.

Uma deficiência da contagem baseada em tique-taque é que o relógio do sistema que realiza a contagem é, freqüentemente, também responsável por distribuir atividades de agendamento relacionadas ao tempo. Como resultado, se um segmento tiver que realizar alguma quantidade de trabalho a cada tique-taque do relógio (ou seja, a cada 10 milissegundos), o sistema irá contar a mais ou a menos no segmento, dependendo se a contagem for feita antes ou depois de uma atividade de agendamento de distribuição relacionada ao tempo. No Solaris, a contagem é realizada antes da distribuição relacionada ao tempo. Como resultado, o sistema irá contar a menos nos segmentos executados em intervalo regular. Se tais segmentos forem executados por menos tempo que o intervalo do tique-taque do relógio, eles podem se "esconder" eficientemente por trás do tique-taque. O exemplo a seguir mostra o grau para o qual o sistema possui tais segmentos:

```
sched:::tick,
sched:::enqueue
{
    @[probename] = lquantize((timestamp / 1000000) % 10, 0, 10);
}
```

O script de exemplo resulta em duas distribuições do deslocamento de milissegundo em um intervalo de dez milissegundos: uma para o teste `tick` e outra para `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
```

O histograma de resultado chamado `tick` mostra que o tique-taque do relógio está sendo acionado em um deslocamento de 8 milissegundos. Se o agendamento não fosse associado ao tique-taque do relógio, o resultado de `enqueue` seria distribuído uniformemente no intervalo de dez milissegundos. Entretanto, o resultado mostra um pico no mesmo deslocamento de 8 milissegundos, indicando que pelo menos alguns segmentos no sistema *estão* sendo agendados de acordo com o tempo.

Estabilidade

O provedor sched usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela a seguir. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA
Argumentos	Desenvolvendo	Desenvolvendo	ISA

◆ ◆ ◆ CAPÍTULO 27

Provedor io

O provedor io disponibiliza os testes relacionados à entrada e saída em disco. O provedor io possibilita a rápida exploração do comportamento observado através de ferramentas de monitoração de E/S como iostat(1M). Por exemplo, ao usar o provedor io, você pode compreender a E/S por dispositivo, por tipo de E/S, por tamanho de E/S, por processo, por nome de aplicativo, por nome de arquivo ou por deslocamento de arquivo.

Testes

Os testes io estão descritos na Tabela 27–1.

TABELA 27–1 Testes io

Teste	Descrição
start	Teste que é acionado quando uma solicitação de E/S está para ser feita seja em um dispositivo periférico ou em um servidor NFS. A bufinfo_t correspondente à solicitação de E/S é apontada por args[0]. A devinfo_t do dispositivo para o qual a E/S está sendo emitida é apontada por args[1]. A fileinfo_t do arquivo que corresponde à solicitação de E/S é apontada por args[2]. Observe que a disponibilidade das informações do arquivo depende do sistema de arquivos fazer a solicitação de E/S. Consulte "fileinfo_t" na página 315 para obter mais informações.
done	Teste que é acionado depois que uma solicitação de E/S tiver sido atendida. A bufinfo_t correspondente à solicitação de E/S é apontada por args[0]. O teste done é acionado depois da conclusão da E/S, mas antes do processamento da conclusão ser realizado no buffer. Como resultado, B_DONE *não* é definido em b_flags quando o teste done é acionado. A devinfo_t do dispositivo para o qual a E/S foi emitida é apontada por args[1]. A fileinfo_t do arquivo que corresponde à solicitação de E/S é apontada por args[2].

TABELA 27–1 Testes `io` *(Continuação)*

Teste	Descrição
`wait-start`	Teste que é acionado imediatamente antes de um segmento começar a aguardar a conclusão pendente de uma determinada solicitação de E/S. A estrutura `buf(9S)` correspondente à solicitação de E/S pela qual o segmento irá aguardar é apontada por `args[0]`. A `devinfo_t` do dispositivo para o qual a E/S foi emitida é apontada por `args[1]` . A `fileinfo_t` do arquivo que corresponde à solicitação de E/S é apontada por `args[2]`. Algum tempo depois do teste `wait-start` ser acionado, o teste `wait-done` será acionado no mesmo segmento.
`wait-done`	Teste que é acionado quando um segmento for concluído aguardando a conclusão de uma determinada solicitação de E/S. A `bufinfo_t` correspondente à solicitação de E/S pela qual o segmento irá aguardar é apontada por `args[0]`. A `devinfo_t` do dispositivo para o qual a E/S foi emitida é apontada por `args[1]`. A `fileinfo_t` do arquivo que corresponde à solicitação de E/S é apontada por `args[2]`. O teste `wait-done` é acionado somente depois que o teste `wait-start` tiver sido acionado no mesmo segmento.

Observe que os testes `io` são acionados para todas as solicitações de E/S em dispositivos periféricos, e para todas as solicitações de leitura e gravação de arquivo em um servidor NFS. As solicitações por metadados de um servidor NFS, por exemplo, *não* acionam testes `io` devido a uma solicitação `readdir(3C)`.

Argumentos

Os tipos de argumento dos testes `io` são listados na Tabela 27–2. Os argumentos são descritos na Tabela 27–1.

TABELA 27–2 Argumentos do teste `io`.

Teste	`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 *`

Cada teste `io` possui argumentos que consistem em um ponteiro para uma estrutura `buf(9S)`, um ponteiro para `devinfo_t` e um ponteiro para `fileinfo_t` . Essas estruturas são descritas com mais detalhes nesta seção.

Estrutura `bufinfo_t`

A estrutura `bufinfo_t` é a abstração que descreve uma solicitação de E/S. O buffer que corresponde a uma solicitação de E/S é apontado por `args[0]` nos testes `start`, `done`, `wait-start` e `wait-done`. A definição da estrutura `bufinfo_t` é a seguinte:

```
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;
```

O membro `b_flags` indica o estado do buffer de E/S e consiste em um operador bit a bit de diferentes valores de estado. Os valores de estado válidos estão na Tabela 27–3.

TABELA 27–3 Valores de `b_flags`

`B_DONE`	Indica que a transferência de dados foi concluída.
`B_ERROR`	Indica um erro de transferência de E/S. É definido junto com o campo `b_error`.
`B_PAGEIO`	Indica que o buffer está sendo usado em uma solicitação de E/S paginada. Veja a descrição do campo `b_addr` para obter mais informações.
`B_PHYS`	Indica que o buffer está sendo usado para E/S física (direta) para uma área de dados do usuário.
`B_READ`	Indica que os dados devem ser lidos a partir do dispositivo periférico para a memória principal.
`B_WRITE`	Indica que os dados devem ser transferidos a partir da memória principal para o dispositivo periférico.
`B_ASYNC`	A solicitação de E/S é assíncrona, e não será aguardada. Os testes `wait-start` e `wait-done` não são acionados em solicitações de E/S assíncronas. Observe que algumas E/Ss direcionadas para ser assíncronas podem não ter `B_ASYNC` definida: o subsistema de E/S assíncrona pode implementar a solicitação assíncrona fazendo com que um segmento de trabalho separado realize uma operação de E/S síncrona.

O campo `b_bcount` é o número de bytes a ser transferido como parte da solicitação de E/S.

O campo b_addr é o endereço virtual da solicitação de E/S, a menos que B_PAGEIO esteja definida. O endereço é um endereço virtual do kernel a menos que B_PHYS esteja definida. Nesse caso, é um endereço virtual do usuário. Se B_PAGEIO estiver definida, o campo b_addr contém dados privados do kernel. Exatamente um de B_PHYS e B_PAGEIO pode ser definido, ou nenhum dos sinalizadores será definido.

O campo b_lblkno identifica qual bloco lógico no dispositivo será acessado. O mapeamento de um bloco lógico para um bloco físico (como o cilindro, a trilha e assim por diante) é definido pelo dispositivo.

O campo b_resid é definido como o número de bytes não transferidos devido a um erro.

O campo b_bufsize contém o tamanho do buffer alocado.

O campo b_iodone identifica uma rotina específica no kernel que é chamada quando a E/S está concluída.

O campo b_error pode conter um código de erro retornado do driver no caso de um erro de E/S. b_error é definido com o conjunto de bits B_ERROR no membro b_flags.

O campo b_edev contém os números principal e secundário do dispositivo acessado. Os consumidores podem usar as sub-rotinas de D getmajor() e getminor() para extrair os números principal e secundário do dispositivo do campo b_edev.

devinfo_t

A estrutura devinfo_t fornece informações sobre um dispositivo. A estrutura devinfo_t correspondente ao dispositivo de destino de uma E/S é apontada por args[1] nos testes start, done, wait-start e wait-done. Os membros de devinfo_t são os seguintes:

```
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;
```

O campo dev_major é o número principal do dispositivo. Consulte getmajor(9F) para obter mais informações.

O campo dev_minor é o número secundário do dispositivo. Consulte getminor(9F) para obter mais informações.

O campo `dev_instance` é o número da instância do dispositivo. A instância de um dispositivo é diferente do número secundário. O número secundário é uma abstração gerenciada pelo driver do dispositivo. O número da instância é uma propriedade do nó do dispositivo. Você pode exibir números de instância de um nó de dispositivo com `prtconf`(1M).

O campo `dev_name` é o nome do driver de dispositivo que gerencia o dispositivo. Você pode exibir nomes de driver de dispositivo com a opção `-D` para `prtconf`(1M).

O campo `dev_statname` é o nome do dispositivo conforme informado por `iostat`(1M). Esse nome também corresponde ao nome de uma estatística do kernel conforme informado por `kstat`(1M). Esse campo é fornecido para que um resultado `iostat` ou `kstat` anormal possa rapidamente ser correlacionado a uma atividade de E/S verdadeira.

O campo `dev_pathname` é o caminho total do dispositivo. Esse caminho pode ser especificado como um argumento para `prtconf`(1M) a fim de obter informações detalhadas sobre o dispositivo. O caminho especificado por `dev_pathname` inclui componentes que expressam o nó do dispositivo, o número de instância e o nó menor. Entretanto, todos esses três elementos não são necessariamente expressados no nome da estatística. Para alguns dispositivos, o nome da estatística consiste no nome do dispositivo e no número da instância. Para outros dispositivos, o nome consiste no nome do dispositivo e no número do nó secundário. Como resultado, dois dispositivos que possuam o mesmo `dev_statname` podem diferir no `dev_pathname` .

`fileinfo_t`

A estrutura `fileinfo_t` fornece informações sobre um arquivo. O arquivo ao qual uma E/S corresponde é apontado por `args[2]` nos testes `start`, `done`, `wait-start` e `wait-done`. A presença de informações do arquivo depende do sistema de arquivos fornecer essas informações ao distribuir solicitações de E/S. Alguns sistemas de arquivos, especialmente de terceiros, podem não fornecer essas informações. Além disso, solicitações de E/S podem surgir de um sistema de arquivos para o qual não existam informações do arquivo. Por exemplo, uma E/S para os metadados do sistema de arquivos não será associada a um arquivo. Finalmente, alguns sistemas de arquivos altamente otimizados podem agregar E/S de arquivos separados em uma única solicitação de E/S. Nesse caso, o sistema de arquivos pode fornecer as informações de arquivo seja para o arquivo que representa a maioria da E/S ou para o arquivo que representa *uma parte* da E/S. De forma alternativa, o sistema de arquivos pode não fornecer as informações de arquivo nesse caso.

A definição da estrutura `fileinfo_t` é a seguinte:

```
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;
```

O campo `fi_name` contém o nome do arquivo mas não inclui componentes do diretório. Se nenhuma informação do arquivo for associada a uma E/S, o campo `fi_name` será definido como a seqüência `<none>`. Em alguns casos raros, o nome de caminho associado a um arquivo pode ser desconhecido. Nesse caso, o campo `fi_name` será definido como a seqüência `<unknown>`.

O campo `fi_dirname` contém *somente* o componente de diretório do nome do arquivo. Assim como com `fi_name`, essa seqüência pode ser definida como `<none>` se nenhuma informação de arquivo estiver presente, ou `<unknown>` se o nome de caminho associado ao arquivo não for conhecido.

O campo `fi_pathname` contém o nome de caminho completo para o arquivo. Assim como com `fi_name`, essa seqüência pode ser definida como `<none>` se nenhuma informação de arquivo estiver presente, ou `<unknown>` se o nome de caminho associado ao arquivo não for conhecido.

O campo `fi_offset` contém o deslocamento no arquivo, ou -1 se a informação do arquivo não estiver presente ou se o deslocamento não estiver especificado pelo sistema de arquivos.

Exemplos

O script de exemplo a seguir exibe informações pertinentes a cada E/S conforme é emitida:

```
#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");
}
```

O resultado da partida a frio do Acrobat Reader em um sistema de laptop x86 lembra o seguinte exemplo:

```
# 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
  ...
```

As entradas `<none>` no resultado indicam que a E/S não corresponde aos dados em nenhum arquivo em particular: essas E/Ss são devido a metadados de um formato ou outro. As entradas `<unknown>` no resultado indicam que o nome de caminho do arquivo não é conhecido. Essa situação é relativamente rara.

Você poderia tornar o script de exemplo um pouco mais sofisticado usando uma matriz de associação para controlar o tempo gasto em cada E/S, conforme mostrado no exemplo a seguir:

```
#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;
}
```

O resultado do exemplo acima durante a conexão automática de um dispositivo de armazenamento USB em um sistema de laptop x86 inativo é mostrado no exemplo a seguir:

```
# 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
  ...
```

Você pode fazer várias observações sobre o mecanismo do sistema com base neste resultado. Primeiro, observe que o tempo de realização das primeiras E/Ss foi longo: aproximadamente 25 milissegundos cada. A causa dessa demora pode ter sido que o dispositivo `cmdk0` teve a energia gerenciada no laptop. Segundo, observe a E/S devido ao driver `scsa2usb`(7D) que está sendo carregado para lidar com o dispositivo de armazenamento em massa USB. Terceiro, observe as gravações em `/var/adm/messages` à medida que o dispositivo é informado. Finalmente, observe a leitura dos geradores de link do dispositivo (os arquivos que terminam em `link.so`) , que supostamente lidam com o novo dispositivo.

O provedor `io` possibilita uma compreensão abrangente do resultado de `iostat`(1M). Suponha que você observe o resultado de `iostat` de forma similar ao exemplo a seguir:

```
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
```

Você pode usar o script `iotime.d` para ver estas E/Ss à medida que acontecem, conforme mostrado no exemplo a seguir:

```
    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
```

Este resultado parece mostrar que o arquivo `archives.tar` está sendo lido a partir de `cmdk0` (em `/export/archives`) e sendo gravado no dispositivo `sd2` (em `/mnt`). Essa existência de dois arquivos chamados `archives.tar` operados separadamente em paralelo parece pouco provável. Para investigar melhor, você pode agregar dispositivo, aplicativo, ID de processo e bytes transferidos, conforme mostrado no exemplo a seguir:

```
#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", @);
}
```

A execução deste script por alguns segundos produz um resultado similar ao exemplo a seguir:

```
# dtrace -s ./whoio.d
^C
    DEVICE                  APP        PID           BYTES
     cmdk0                   cp        790         1515520
       sd2                   cp        790         1527808
```

Este resultado mostra que esta atividade *é* uma cópia do arquivo `archives.tar` de um dispositivo para outro. Essa conclusão leva a uma outra pergunta natural: um desses dispositivos é mais rápido que o outro? Qual dispositivo atua como limitador na cópia? Para responder a estas perguntas, você precisa conhecer o throughput efetivo de cada dispositivo em vez do número de bytes por segundo que cada dispositivo está transferindo. Você pode determinar o throughput com o seguinte script de exemplo:

```
#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", @);
}
```

A execução do script de exemplo por vários segundos produz o seguinte resultado:

```
  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
```

O resultado mostra que `sd2` é claramente o dispositivo limitador. O throughput `sd2` está entre 256K/seg. e 512K/seg., enquanto `cmdk0` está entregando E/S em qualquer ponto de 8 MB/segundo a mais de 64 MB/segundo. O script imprime o nome conforme visto em `iostat` e o caminho completo do dispositivo. Para descobrir mais sobre o dispositivo, você pode especificar o caminho do dispositivo para `prtconf`, conforme mostrado no exemplo a seguir:

```
# 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
```

Conforme os termos enfatizados indicam, este dispositivo é um dispositivo de armazenamento USB removível.

Os exemplos nesta seção exploraram todas as solicitações de E/S. Entretanto, você pode estar interessado em somente um tipo de solicitação. O exemplo a seguir controla os diretórios nos quais as gravações estão ocorrendo, junto com os aplicativos que estão realizando as gravações:

```
#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", @);
}
```

A execução deste script de exemplo em uma carga de área de trabalho por um período de tempo produz alguns resultados interessantes, conforme mostrado no seguinte resultado de exemplo:

```
# 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
```

Como o resultado indica, virtualmente todas as gravações são associadas ao cache do Mozilla Firebird. As gravações rotuladas <none> são provavelmente devido a gravações associadas ao log UFS, gravações que são, elas mesmas, induzidas por outras gravações no sistema de arquivos. Consulte `ufs`(7FS) para obter detalhes sobre log. Este exemplo mostra como usar o provedor `io` para descobrir um problema em uma camada mais elevada de software. Neste caso,

o script revelou um problema de configuração: o navegador da Web induziria muito menos E/S (e provavelmente nenhuma) se o seu cache estivesse em um diretório em um sistema de arquivos tmpfs(7FS).

Os exemplos anteriores usaram somente os testes `start` e `done`. Você pode usar os testes `wait-start` e `wait-done` para compreender por que os aplicativos são bloqueados para E/S e por quanto tempo. O seguinte script de exemplo usa os testes de `io` e de `sched` (consulte o Capítulo 26, "Provedor `sched`") para derivar tempo de CPU em comparação ao tempo de espera de E/S do software 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);
}
```

A execução do script de exemplo durante uma partida a frio do software StarOffice produz o seguinte resultado:

```
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
```

Como este resultado mostra, grande parte do tempo de partida a frio do StarOffice é devido à espera por E/S. (13,1 segundos de espera por E/S em comparação a 3,6 segundos na CPU.) A execução do script em uma partida a quente do software StarOffice revela que o armazenamento em cache da página eliminou o tempo de E/S, conforme mostrado no seguinte resultado de exemplo:

```
Time breakdown (milliseconds):
  <I/O wait>                                                  0
  <on cpu>                                                 2860

I/O wait breakdown (milliseconds):
  temp                                                        0
  soffice.tmp                                                 0
  <unknown>                                                   0
  Office                                                      0
```

O resultado da partida a frio mostra que o arquivo `applicat.rdb` contabiliza mais tempo de espera por E/S que qualquer outro arquivo. Esse resultado provavelmente é devido à grande quantidade de E/Ss no arquivo. Para explorar as E/Ss realizadas nesse arquivo, você pode usar o seguinte script de D:

```
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);
}
```

Este script usa o campo `fi_offset` da estrutura `fileinfo_t` para compreender quais partes do arquivo estão sendo acessadas, na granularidade de um megabyte. A execução desse script durante uma partida a frio do software StarOffice produz um resultado similar ao seguinte exemplo:

```
# 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
```

Este resultado indica que somente os primeiros seis megabytes do arquivo são acessados, talvez porque o tamanho do arquivo seja de seis megabytes. O resultado também indica que o arquivo completo não é acessado. Se você desejasse aumentar o tempo da partida a frio do StarOffice, talvez desejasse compreender o padrão de acesso do arquivo. Se as seções necessárias do arquivo pudessem ser totalmente contíguas, uma forma de aumentar o tempo de partida a frio do StarOffice seria executar um segmento verificador antes do aplicativo, induzindo a E/S no arquivo antes do necessário. (Essa abordagem é particularmente fácil se o arquivo for acessado usando-se `mmap(2)`.) Entretanto, o tempo de aproximadamente 1,6 segundos que essa estratégia ganharia na partida a frio não compensa a complexidade adicional e o aumento de manutenção no aplicativo. De qualquer forma, os dados coletados com o provedor `io` permitem uma compreensão precisa do benefício que tal trabalho poderia oferecer.

Estabilidade

O provedor `io` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela a seguir. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA
Argumentos	Desenvolvendo	Desenvolvendo	ISA

◆ ◆ ◆ CAPÍTULO 28

Provedor mib

O provedor mib disponibiliza testes que correspondem a contadores nas bases de informações de gerenciamento do Solaris (MIBs). Os contadores MIB são usados pelo protocolo simples de gerenciamento de rede (SNMP) que permite o monitoramento remoto de entidades de rede heterogêneas. Você também pode exibir os contadores com os comandos kstat(1M) e netstat(1M). O provedor mib facilita a rápida exploração de um comportamento de rede aberrante que é observado durante os monitoramentos de rede remota ou local.

Testes

O provedor mib disponibiliza testes de contadores de vários MIBs. Os protocolos que exportam MIBs instrumentados pelo provedor mib estão listados na Tabela 28–1. A tabela inclui uma referência à documentação que especifica parte ou todo o MIB, o nome da estatística do kernel que pode ser usado para acessar as contagens que estão em execução (usando a opção kstat(1M) -n *estática*), e uma referência à tabela que possui uma definição completa dos testes. Todos os contadores MIB também estão disponíveis através da opção -s para netstat(1M).

TABELA 28–1 Testes mib

Protocolo	Descrição de MIB	Estatística do Kernel	Tabela de testes de mib
ICMP	RFC 1213	icmp	Tabela 28–2
IP	RFC 1213	ip	Tabela 28–3
IPsec	—	ip	Tabela 28–4
IPv6	RFC 2465	—	Tabela 28–5
SCTP	"SCTP MIB" (rascunho da Internet)	sctp	Tabela 28–7
TCP	RFC 1213	tcp	Tabela 28–8

TABELA 28–1 Testes `mib` *(Continuação)*

Protocolo	Descrição de MIB	Estatística do Kernel	Tabela de testes de `mib`
UDP	RFC 1213	`udp`	Tabela 28–9

TABELA 28–2 Testes `mib` ICMP

`icmpInAddrMaskReps`	Teste que é acionado sempre que uma mensagem de resposta de máscara de endereço ICMP é recebida.
`icmpInAddrMasks`	Teste que é acionado sempre que uma mensagem de solicitação de máscara de endereço ICMP é recebida.
`icmpInBadRedirects`	Teste que é acionado sempre que uma mensagem de redirecionamento ICMP é recebida e determinada como mal formada de alguma maneira (código ICMP desconhecido, remetente ou destino off-link, etc.).
`icmpInCksumErrs`	Teste que é acionado sempre que uma mensagem ICMP com uma soma de verificação incorreta é recebida.
`icmpInDestUnreachs`	Teste que é acionado sempre que uma mensagem ICMP de destino não-alcançável é recebida.
`icmpInEchoReps`	Teste que é acionado sempre que uma mensagem de resposta de eco ICMP é recebida.
`icmpInEchos`	Teste que é acionado sempre que uma mensagem de solicitação de eco ICMP é recebida.
`icmpInErrors`	Teste que é acionado sempre que uma mensagem ICMP é recebida e é determinado que ela possui um erro específico de ICMP (soma de verificação ICMP incorreta, tamanho incorreto, etc.).
`icmpInFragNeeded`	Teste que é acionado sempre que uma mensagem ICMP de destino inalcançável (fragmentação necessária) é recebida, indicando que um pacote enviado foi perdido porque era maior do que alguma MTU e o sinalizador Não fragmentar foi enviado.
`icmpInMsgs`	Teste que é acionado sempre que uma mensagem ICMP é recebida. Sempre que esse teste é acionado, o teste `icmpInErrors` também poderá ser acionado, se for determinado que a mensagem possui um erro específico de ICMP.
`icmpInOverflows`	Teste que é acionado sempre que uma mensagem ICMP é recebida, mas é subseqüentemente descartada por falta de espaço no buffer.
`icmpInParmProbs`	Teste que é acionado sempre que uma mensagem de problema de parâmetro ICMP é recebida.
`icmpInRedirects`	Teste que é acionado sempre que uma mensagem de redirecionamento ICMP é recebida.
`icmpInSrcQuenchs`	Teste que é acionado sempre que uma mensagem de exclusão de origem ICMP é recebida.

TABELA 28–2 Testes `mib` ICMP *(Continuação)*

`icmpInTimeExcds`	Teste que é acionado sempre que uma mensagem de tempo excedido ICMP é recebida.
`icmpInTimestampReps`	Teste que é acionado sempre que uma mensagem de resposta de carimbo de data/hora ICMP é recebida.
`icmpInTimestamps`	Teste que é acionado sempre que uma mensagem de solicitação de carimbo de data/hora é recebida.
`icmpInUnknowns`	Teste que é acionado sempre que uma mensagem ICMP de tipo desconhecido é recebida.
`icmpOutAddrMaskReps`	Teste que é acionado sempre que uma mensagem de resposta de máscara de endereço ICMP é enviada.
`icmpOutDestUnreachs`	Teste que é acionado sempre que uma mensagem ICMP de destino não-alcançável é enviada.
`icmpOutDrops`	Teste que é acionado sempre que uma mensagem ICMP externa é descartada por algum motivo (tal como falha de alocação de memória, origem ou destino de transmissão/multicast, etc.).
`icmpOutEchoReps`	Teste que é acionado sempre que uma mensagem de resposta de eco ICMP é enviada.
`icmpOutErrors`	Teste que é acionado sempre que uma mensagem ICMP não é enviada devido a problemas descobertos no ICMP, tal como falta de buffers. Esse teste não será acionado, se forem descobertos erros fora da camada do ICMP, tal como a inabilidade do IP de rotear o datagrama resultante.
`icmpOutFragNeeded`	Teste que é acionado sempre que uma mensagem ICMP de destino inalcançável (fragmentação necessária) é enviada.
`icmpOutMsgs`	Teste que é acionado sempre que uma mensagem ICMP é enviada. Sempre que esse teste é acionado, o teste `icmpOutErrors` também poderá ser acionado, se for determinado que a mensagem possui um erro específico de ICMP.
`icmpOutParmProbs`	Teste que é acionado sempre que uma mensagem de problema de parâmetro ICMP é enviada.
`icmpOutRedirects`	Teste que é acionado sempre que uma mensagem de redirecionamento ICMP é enviada. Para um host, esse teste nunca será acionado, pois os hosts não enviam redirecionamentos.
`icmpOutTimeExcds`	Teste que é acionado sempre que uma mensagem de tempo excedido ICMP é enviada.
`icmpOutTimestampReps`	Teste que é acionado sempre que uma mensagem de resposta de carimbo de data/hora ICMP é enviada.

TABELA 28–3 Testes `mib` IP

`ipForwDatagrams`	Teste que é acionado sempre que é recebido um datagrama que não possui esta máquina como seu destino IP final, e é feita uma tentativa de encontrar uma rota para encaminhar o datagrama para esse destino final. Em máquinas que não agem como gateways IP, esse teste só será acionado para os pacotes que são roteados na origem através desta máquina, e para os quais o processamento da opção de origem de rota foi bem sucedido.
`ipForwProhibits`	Teste que é acionado sempre que é recebido um datagrama que não possui esta máquina como seu destino IP final, mas como a máquina não tem permissão para agir como um roteador, não é feita uma tentativa de encontrar uma rota para encaminhar o datagrama para esse destino final.
`ipFragCreates`	Teste que é acionado sempre que um fragmento de datagrama IP é gerado como um resultado de fragmentação.
`ipFragFails`	Teste que é acionado sempre que um datagrama IP é descartado porque não pôde ser fragmentado, por exemplo, porque a fragmentação era necessária e o sinalizador Não fragmentar foi enviado.
`ipFragOKs`	Teste que é acionado sempre que um datagrama IP foi fragmentado com êxito.
`ipInCksumErrs`	Teste que é acionado sempre que um datagrama de entrada é descartado devido a uma soma de verificação de cabeçalho IP incorreta.
`ipInDelivers`	Teste que é acionado sempre que um datagrama de entrada é entregue com êxito para protocolos IP do usuário, incluindo ICMP.
`ipInDiscards`	Teste que é acionado sempre que um datagrama IP de entrada é descartado por motivos não relacionados ao pacote (por exemplo, por falta de espaço no buffer). Esse teste não é acionado para qualquer datagrama descartado durante a espera da recomposição.
`ipInHdrErrors`	Teste que é acionado sempre que um datagrama de entrada é descartado devido a um erro em seu cabeçalho IP, incluindo uma incompatibilidade de número de versão, um erro de formato, um tempo de conexão excedido, uma descoberta de erro nas opções de processamento de IP, etc.
`ipInIPv6`	Teste que é acionado sempre que um pacote IPv6 chega erradamente em uma fila IPv4.
`ipInReceives`	Teste que é acionado sempre que um datagrama é recebido de uma interface, mesmo se esse datagrama for recebido erradamente.
`ipInUnknownProtos`	Teste que é acionado sempre que um datagrama endereçado localmente é recebido com êxito, mas descartado subseqüentemente devido a um protocolo desconhecido ou sem suporte.

TABELA 28–3 Testes `mib` IP *(Continuação)*

`ipOutDiscards`	Teste que é acionado sempre que um datagrama IP de saída é descartado por motivos não relacionados ao pacote (por exemplo, por falta de espaço no buffer). Esse teste será acionado para um pacote contado no contador MIB `ipForwDatagrams`, se o pacote atender a um critério de descarte (sem restrições).
`ipOutIPv6`	Teste que é acionado sempre que um pacote IPv6 é enviado através de uma conexão IPv4.
`ipOutNoRoutes`	Teste que é acionado sempre que um datagrama IP é descartado porque nenhuma rota foi encontrada para transmiti-lo para o seu destino. Esse teste será acionado para um pacote contado no contador MIB `ipForwDatagrams`, se o pacote atender ao critério "sem rota". Esse teste também será acionado para quaisquer datagramas que não podem ser roteados porque todos os gateways estão inativos.
`ipOutRequests`	Teste que é acionado sempre que um datagrama IP é fornecido para o IP para transmissão a partir de protocolos IP locais do usuário (incluindo ICMP). Observe que esse teste não será acionado para qualquer pacote contado no contador MIB `ipForwDatagrams`.
`ipOutSwitchIPv6`	Teste que é acionado sempre que uma conexão é alterada de IPv4 para IPv6, como seu protocolo IP.
`ipReasmDuplicates`	Teste que é acionado sempre que o algoritmo de recomposição IP determina que um fragmento IP contém *apenas* dados recebidos anteriormente.
`ipReasmFails`	Teste que é acionado sempre que qualquer falha é detectada pelo algoritmo de recomposição IP. Esse teste não necessariamente é acionado para cada fragmento IP descartado porque alguns algoritmos, notadamente o algoritmo em RFC 815, podem perder o controle de fragmentos, combinando-os quando são recebidos.
`ipReasmOKs`	Teste que é acionado sempre que um datagrama IP é recomposto com êxito.
`ipReasmPartDups`	Teste que é acionado sempre que o algoritmo de recomposição IP determina que um fragmento IP contém dados recebidos anteriormente e alguns novos dados.
`ipReasmReqds`	Teste que é acionado sempre que um fragmento IP recebido precisa ser recomposto.

TABELA 28–4 Testes `mib` IPsec

`ipsecInFailed`	Teste que é acionado sempre que um pacote recebido é descartado porque não corresponde à política IPsec especificada.
`ipsecInSucceeded`	Teste que é acionado sempre que um pacote recebido corresponde à política IPsec especificada e o processamento tem permissão para continuar.

TABELA 28–5 Testes `mib` IPv6

`ipv6ForwProhibits`	Teste que é acionado sempre que é recebido um datagrama IPv6 que não possui esta máquina como seu destino IPv6 final, mas porque a máquina não tem permissão para agir como um roteador, não é feita uma tentativa de encontrar uma rota para encaminhar o datagrama para esse destino final.
`ipv6IfIcmpBadHoplimit`	Teste que é acionado sempre que uma mensagem de protocolo de descoberta de vizinho ICMPv6 recebida é considerada como tendo um Limite de salto menor que o máximo definido. Tais mensagens podem não ter sido originadas de um vizinho e, portanto, são descartadas.
`ipv6IfIcmpInAdminProhibs`	Teste que é acionado sempre que uma mensagem de destino ICMPv6 inalcançável (comunicação proibida administrativamente) é recebida.
`ipv6IfIcmpInBadNeighborAdvertisements`	Teste que é acionado sempre que uma mensagem de anúncio de vizinho ICMPv6 recebida é malformada de alguma maneira.
`ipv6IfIcmpInBadNeighborSolicitations`	Teste que é acionado sempre que uma mensagem de requisição de vizinho ICMPv6 recebida é malformada de alguma maneira.
`ipv6IfIcmpInBadRedirects`	Teste que é acionado sempre que uma mensagem de redirecionamento ICMPv6 recebida é malformada de alguma maneira.
`ipv6IfIcmpInDestUnreachs`	Teste que é acionado sempre que uma mensagem ICMPv6 de destino não-alcançável é recebida.
`ipv6IfIcmpInEchoReplies`	Teste que é acionado sempre que uma mensagem de resposta de eco ICMPv6 é recebida.
`ipv6IfIcmpInEchos`	Teste que é acionado sempre que uma mensagem de solicitação de eco ICMPv6 é recebida.
`ipv6IfIcmpInErrors`	Teste que é acionado sempre que é recebida uma mensagem ICMPv6 e é determinado que ela possui erro específico de ICMPv6 (tal como uma soma de verificação ICMPv6 incorreta, tamanho incorreto, etc).
`ipv6IfIcmpInGroupMembBadQueries`	Teste que é acionado sempre que uma mensagem de consulta de associação de grupo ICMPv6 recebida é malformada de alguma maneira.

TABELA 28–5 Testes `mib` IPv6 *(Continuação)*

`ipv6IfIcmpInGroupMembBadReports`	Teste que é acionado sempre que uma mensagem de relatório de associação de grupo ICMPv6 recebida é malformada de alguma maneira.
`ipv6IfIcmpInGroupMembOurReports`	Teste que é acionado sempre que uma mensagem de relatório de associação de grupo ICMPv6 é recebida.
`ipv6IfIcmpInGroupMembQueries`	Teste que é acionado sempre que uma mensagem de consulta de associação de grupo ICMPv6 é recebida.
`ipv6IfIcmpInGroupMembReductions`	Teste que é acionado sempre que uma mensagem de redução de associação de grupo ICMPv6 é recebida.
`ipv6IfIcmpInGroupMembResponses`	Teste que é acionado sempre que uma mensagem de resposta de associação de grupo ICMPv6 é recebida.
`ipv6IfIcmpInGroupMembTotal`	Teste que é acionado sempre que uma mensagem de descoberta de ouvinte multicast ICMPv6 é recebida.
`ipv6IfIcmpInMsgs`	Teste que é acionado sempre que uma mensagem ICMPv6 é recebida. Quando esse teste é acionado, o teste `ipv6IfIcmpInErrors` também poderá ser acionado, se a mensagem tiver um erro ICMPv6 específico.
`ipv6IfIcmpInNeighborAdvertisements`	Teste que é acionado sempre que uma mensagem de anúncio de vizinho ICMPv6 é recebida.
`ipv6IfIcmpInNeighborSolicits`	Teste que é acionado sempre que uma mensagem de requisição de vizinho ICMPv6 é recebida.
`ipv6IfIcmpInOverflows`	Teste que é acionado sempre que uma mensagem ICMPv6 é recebida, mas é subseqüentemente descartada por falta de espaço no buffer.
`ipv6IfIcmpInParmProblems`	Teste que é acionado sempre que uma mensagem de problema de parâmetro ICMPv6 é recebida.
`ipv6IfIcmpInRedirects`	Teste que é acionado sempre que uma mensagem de redirecionamento ICMPv6 é recebida.
`ipv6IfIcmpInRouterAdvertisements`	Teste que é acionado sempre que uma mensagem de anúncio de roteador ICMPv6 é recebida.
`ipv6IfIcmpInRouterSolicits`	Teste que é acionado sempre que uma mensagem de requisição de roteador ICMPv6 é recebida.
`ipv6IfIcmpInTimeExcds`	Teste que é acionado sempre que uma mensagem de tempo excedido ICMPv6 é recebida.
`ipv6IfIcmpOutAdminProhibs`	Teste que é acionado sempre que uma mensagem de destino ICMPv6 inalcançável (comunicação proibida administrativamente) é enviada.

TABELA 28–5 Testes `mib` IPv6 *(Continuação)*

`ipv6IfIcmpOutDestUnreachs`	Teste que é acionado sempre que uma mensagem ICMPv6 de destino não-alcançável é enviada.
`ipv6IfIcmpOutEchoReplies`	Teste que é acionado sempre que uma mensagem de resposta de eco ICMPv6 é enviada.
`ipv6IfIcmpOutEchos`	Teste que é acionado sempre que uma mensagem de eco ICMPv6 é enviada.
`ipv6IfIcmpOutErrors`	Teste que é acionado sempre que uma mensagem ICMPv6 não é enviada devido a problemas descobertos no ICMPv6, tal como falta de buffers. Esse teste não será acionado, se forem descobertos erros fora da camada do ICMPv6, tal como a inabilidade do IPv6 de rotear o datagrama resultante.
`ipv6IfIcmpOutGroupMembQueries`	Teste que é acionado sempre que uma mensagem de consulta de associação de grupo ICMPv6 é enviada.
`ipv6IfIcmpOutGroupMembReductions`	Teste que é acionado sempre que uma mensagem de redução de associação de grupo ICMPv6 é enviada.
`ipv6IfIcmpOutGroupMembResponses`	Teste que é acionado sempre que uma mensagem de resposta de associação de grupo ICMPv6 é enviada.
`ipv6IfIcmpOutMsgs`	Teste que é acionado sempre que uma mensagem ICMPv6 é enviada. Quando esse teste é acionado, o teste `ipv6IfIcmpOutErrors` também poderá ser acionado, se a mensagem tiver erros ICMPv6 específicos.
`ipv6IfIcmpOutNeighborAdvertisements`	Teste que é acionado sempre que uma mensagem de anúncio de vizinho ICMPv6 é enviada.
`ipv6IfIcmpOutNeighborSolicits`	Teste que é acionado sempre que uma mensagem de solicitação de vizinho ICMPv6 é enviada.
`ipv6IfIcmpOutParmProblems`	Teste que é acionado sempre que uma mensagem de problema de parâmetro ICMPv6 é enviada.
`ipv6IfIcmpOutPktTooBigs`	Teste que é acionado sempre que uma mensagem de pacote ICMPv6 muito grande é enviada.
`ipv6IfIcmpOutRedirects`	Teste que é acionado sempre que uma mensagem de redirecionamento ICMPv6 é enviada. Para um host, esse teste nunca será acionado, pois os hosts não enviam redirecionamentos.
`ipv6IfIcmpOutRouterAdvertisements`	Teste que é acionado sempre que uma mensagem de anúncio de roteador ICMPv6 é enviada.

TABELA 28–5 Testes `mib` IPv6 *(Continuação)*

`ipv6IfIcmpOutRouterSolicits`	Teste que é acionado sempre que uma mensagem de requisição de roteador ICMPv6 é enviada.
`ipv6IfIcmpOutTimeExcds`	Teste que é acionado sempre que uma mensagem de tempo excedido ICMPv6 é enviada.
`ipv6InAddrErrors`	Teste que é acionado sempre que um datagrama de entrada é descartado porque o endereço IPv6 em seu campo de destino de cabeçalho IPv6 não é um endereço válido para ser recebido por esta entidade. Esse teste será acionado para endereços inválidos (por exemplo, : : 0) e para endereços não suportados (por exemplo, endereços com prefixos não alocados). Para máquinas que não são configuradas para agir como roteadores IPv6 e, portanto, não encaminham datagramas, esse teste será acionado para datagramas descartados porque o endereço de destino não era um endereço local.
`ipv6InDelivers`	Teste que é acionado sempre que um datagrama de entrada é entregue para protocolos IPv6 do usuário (incluindo ICMPv6).
`ipv6InDiscards`	Teste que é acionado sempre que um datagrama IPv6 de entrada é descartado por motivos não relacionados ao pacote (por exemplo, por falta de espaço no buffer). Esse teste não é acionado para qualquer datagrama descartado durante a espera da recomposição.
`ipv6InHdrErrors`	Teste que é acionado sempre que um datagrama de entrada é descartado devido a um erro em seu cabeçalho IPv6, incluindo uma incompatibilidade de número de versão, um erro de formato, uma contagem de salto excedida, uma descoberta de erro nas opções de processamento de IPv6, etc.
`ipv6InIPv4`	Teste que é acionado sempre que um pacote IPv4 chega erradamente em uma fila IPv6.
`ipv6InMcastPkts`	Teste que é acionado sempre que um pacote IPv6 de multicast é recebido.
`ipv6InNoRoutes`	Teste que é acionado sempre que um datagrama IPv6 roteado é descartado porque nenhuma rota foi encontrada para transmiti-lo para o seu destino. Esse teste *só* será acionado para pacotes que foram originados externamente.
`ipv6InReceives`	Teste que é acionado sempre que um datagrama IPv6 é recebido de uma interface, mesmo se esse datagrama for recebido erradamente.

TABELA 28–5 Testes `mib` IPv6 *(Continuação)*

`ipv6InTooBigErrors`	Teste que é acionado sempre que um fragmento recebido é maior que o tamanho máximo de fragmento.
`ipv6InTruncatedPkts`	Teste que é acionado sempre que um diagrama de entrada é descartado porque o quadro do datagrama não carregava dados suficientes.
`ipv6InUnknownProtos`	Teste que é acionado sempre que um datagrama IPv5 endereçado localmente é recebido com êxito, mas descartado subseqüentemente devido a um protocolo desconhecido ou sem suporte.
`ipv6OutDiscards`	Teste que é acionado sempre que um datagrama IPv6 de saída é descartado por motivos não relacionados ao pacote (por exemplo, por falta de espaço no buffer). Esse teste será acionado para um pacote contado no contador MIB `ipv6OutForwDatagrams`, se o pacote atender a um critério de descarte (sem restrições).
`ipv6OutForwDatagrams`	Teste que é acionado sempre que é recebido um datagrama que não possui esta máquina como seu destino IPv6 final, e é feita uma tentativa de encontrar uma rota para encaminhar o datagrama para esse destino final. Em uma máquina que não age como um roteador IPv6, esse teste só será acionado para os pacotes que são roteados na origem através da máquina, e para os quais o processamento da opção de rota de origem foi bem sucedido.
`ipv6OutFragCreates`	Teste que é acionado sempre que um fragmento de datagrama IPv6 é gerado como resultado da fragmentação.
`ipv6OutFragFails`	Teste que é acionado sempre que um datagrama IPv6 é descartado porque não pôde ser fragmentado, por exemplo, porque o sinalizador Não fragmentar foi enviado.
`ipv6OutFragOKs`	Teste que é acionado sempre que datagramas IPv6 foram fragmentados com êxito.
`ipv6OutIPv4`	Teste que é acionado sempre que um pacote IPv6 é enviado através de uma conexão IPv4.
`ipv6OutMcastPkts`	Teste que é acionado sempre que um pacote de multicast é enviado.

TABELA 28–5 Testes `mib` IPv6 *(Continuação)*

`ipv6OutNoRoutes`	Teste que é acionado sempre que um datagrama IPv6 é descartado porque nenhuma rota foi encontrada para transmiti-lo para o seu destino. Esse teste *não* será acionado para pacotes que foram originados externamente.
`ipv6OutRequests`	Teste que é acionado sempre que um datagrama IPv6 é fornecido para o IPv6 para transmissão a partir de protocolos IPv6 locais do usuário (incluindo ICMPv6). Esse teste não será acionado para qualquer pacote contado no contador MIB `ipv6ForwDatagrams`.
`ipv6OutSwitchIPv4`	Teste que é acionado sempre que uma conexão é alterada de IPv6 para IPv4, como seu protocolo IP.
`ipv6ReasmDuplicates`	Teste que é acionado sempre que o algoritmo de recomposição IPv6 determina que um fragmento IPv6 contém *apenas* dados recebidos anteriormente.
`ipv6ReasmFails`	Teste que é acionado sempre que uma falha é detectada pelo algoritmo de recomposição IPv6. Esse teste não necessariamente é acionado para cada fragmento IPv6 descartado, já que alguns algoritmos podem perder o controle de fragmentos, combinando-os quando são recebidos.
`ipv6ReasmOKs`	Teste que é acionado sempre que um datagrama IPv6 é recomposto com êxito.
`ipv6ReasmPartDups`	Teste que é acionado sempre que o algoritmo de recomposição IPv6 determina que um fragmento IPv6 contém dados recebidos anteriormente e alguns novos dados.
`ipv6ReasmReqds`	Teste que é acionado sempre que um fragmento IPv6 recebido precisa ser recomposto.

TABELA 28–6 Testes `mib` IP básicos

`rawipInCksumErrs`	Teste que é acionado sempre que um pacote IP básico recebido possui uma soma de verificação IP incorreta.
`rawipInDatagrams`	Teste que é acionado sempre que um pacote IP básico é recebido.
`rawipInErrors`	Teste que é acionado sempre que um pacote IP básico recebido é malformado de alguma maneira.
`rawipInOverflows`	Teste que é acionado sempre que um pacote IP básico é recebido, mas esse pacote é subseqüentemente descartado por falta de espaço no buffer.
`rawipOutDatagrams`	Teste que é acionado sempre que um pacote IP básico é enviado.

TABELA 28–6 Testes `mib` IP básicos *(Continuação)*

`rawipOutErrors`	Teste que é enviado sempre que um pacote IP básico não é enviado devido a alguma condição de erro, geralmente porque o pacote IP básico era malformado de alguma maneira.

TABELA 28–7 Testes `mib` SCTP

`sctpAborted`	Teste que é acionado sempre que uma associação SCTP tenha feito uma transição direta para o estado CLOSED a partir de qualquer estado, usando o primitivo ABORT, indicando a terminação incorreta da associação.
`sctpActiveEstab`	Teste que é acionado sempre que uma associação SCTP tenha feito uma transição direta para o estado ESTABLISHED a partir do estado COOKIE-ECHOED, indicando que a camada superior iniciou uma tentativa de associação.
`sctpChecksumError`	Teste que é acionado sempre que um pacote SCTP é recebido dos iguais com uma soma de verificação válida.
`sctpCurrEstab`	Teste que é acionado sempre que uma associação SCTP é contada como uma parte da leitura do contador MIB `sctpCurrEstab`. Uma associação SCTP será contada, se seu estado atual for ESTABLISHED, SHUTDOWN-RECEIVED ou SHUTDOWN-PENDING.
`sctpFragUsrMsgs`	Teste que é acionado sempre que uma mensagem do usuário tem que ser fragmentada por causa da MTU.
`sctpInClosed`	Teste que é acionado sempre que os dados são recebidos em uma associação SCTP fechada.
`sctpInCtrlChunks`	Teste que é acionado sempre que o contador MIB `sctpInCtrlChunks` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.
`sctpInDupAck`	Teste que é acionado sempre que um ACK duplicado é recebido.
`sctpInInvalidCookie`	Teste que é acionado sempre que um cookie inválido é recebido.
`sctpInOrderChunks`	Teste que é acionado sempre que o contador MIB `sctpInOrderChunks` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.
`sctpInSCTPPkts`	Teste que é acionado sempre que o contador MIB `sctpInSCTPPkts` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.

TABELA 28–7 Testes `mib` SCTP *(Continuação)*

`sctpInUnorderChunks`	Teste que é acionado sempre que o contador MIB `sctpInUnorderChunks` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.
`sctpListenDrop`	Teste que é acionado sempre que uma conexão de entrada é descartada por qualquer motivo.
`sctpOutAck`	Teste que é acionado sempre que um aviso de recebimento seletivo é enviado.
`sctpOutAckDelayed`	Teste que é acionado sempre que um processamento de aviso de recebimento atrasado é realizado para uma associação SCTP. Quaisquer avisos de recebimento enviados como parte de um processamento de aviso de recebimento atrasado fará com que o teste `sctpOutAck` seja acionado.
`sctpOutCtrlChunks`	Teste que é acionado sempre que o contador MIB `sctpOutCtrlChunks` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.
`sctpOutOfBlue`	Teste que é acionado sempre é enviado que um pacote SCTP correto para o qual o receptor não consegue identificar a associação a qual o pacote pertence.
`sctpOutOrderChunks`	Teste que é acionado sempre que o contador MIB `sctpOutOrderChunks` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.
`sctpOutSCTPPkts`	Teste que é acionado sempre que o contador MIB `sctpOutSCTPPkts` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.
`sctpOutUnorderChunks`	Teste que é acionado sempre que o contador MIB `sctpOutUnorderChunks` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.
`sctpOutWinProbe`	Teste que é acionado sempre que uma janela de teste é enviada.
`sctpOutWinUpdate`	Teste que é acionado sempre que uma atualização de janela é enviada.
`sctpPassiveEstab`	Teste que é acionado sempre que associações SCTP tenham feito uma transição direta para o estado ESTABLISHED a partir do estado CLOSED. O ponto final remoto iniciou a tentativa de associação.
`sctpReasmUsrMsgs`	Teste que é acionado sempre que o contador MIB `sctpReasmUsrMsgs` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.

TABELA 28–7 Testes `mib` SCTP *(Continuação)*

`sctpRetransChunks`	Teste que é acionado sempre que o contador MIB `sctpRetransChunks` é atualizado, seja porque o contador MIB é explicitamente consultado ou porque a conexão SCTP é fechada. O valor pelo qual o contador MIB deve ser aumentado está em `args[0]`.
`sctpShutdowns`	Teste que é acionado sempre que uma associação SCTP faz a transição direta do estado CLOSED a partir do estado SHUTDOWN-SENT ou do estado SHUTDOWN-ACK-SENT, indicando a terminação bem sucedida da associação.
`sctpTimHeartBeatDrop`	Teste que é acionado sempre que uma associação SCTP é anulada devido a uma falha ao receber um aviso de recebimento de pulsação.
`sctpTimHeartBeatProbe`	Teste que é acionado sempre que uma pulsação SCTP é enviada.
`sctpTimRetrans`	Teste que é acionado sempre que um processamento de retransmissão baseado no temporizador é realizado em uma associação.
`sctpTimRetransDrop`	Teste que é acionado sempre que uma falha prolongada para realizar a retransmissão baseada no temporizador resulta na anulação da associação.

TABELA 28–8 Testes `mib` TCP

`tcpActiveOpens`	Teste que é acionado sempre que uma conexão TCP faz uma transição direta do estado CLOSED para o estado SYN_SENT.
`tcpAttemptFails`	Teste que é acionado sempre que uma conexão TCP faz uma transição direta para o estado CLOSED a partir do estado SYN_SENT ou do estado SYN_RCVD, e sempre que uma conexão TCP faz uma transição direta para o estado LISTEN a partir do estado SYN_RCVD.
`tcpCurrEstab`	Teste que é acionado sempre que uma conexão TCP é contada como uma parte da leitura do contador MIB `tcpCurrEstab`. Uma conexão TCP será contada, se seu estado atual for ESTABLISHED ou CLOSE_WAIT.
`tcpEstabResets`	Teste que é acionado sempre que uma conexão TCP faz a transição direta do estado CLOSED a partir do estado ESTABLISHED ou do estado CLOSE_WAIT.
`tcpHalfOpenDrop`	Teste que é acionado sempre que uma conexão é descartada devido a uma fila cheia de conexões no estado SYN_RCVD.
`tcpInAckBytes`	Teste que é acionado sempre que um ACK é recebido para dados enviados anteriormente. O número de bytes recebidos é passado em `args[0]`.
`tcpInAckSegs`	Teste que é acionado sempre que um ACK é recebido para um segmento enviado anteriormente.
`tcpInAckUnsent`	Teste que é acionado sempre que um ACK é recebido para um segmento não-enviado.

TABELA 28–8 Testes `mib` TCP *(Continuação)*

`tcpInClosed`	Teste que é acionado sempre que dados foram recebidos de uma conexão em um estado de fechamento.
`tcpInDataDupBytes`	Teste que é acionado sempre que um segmento é recebido, sendo que todos os dados do segmento tenham sido recebidos anteriormente. O número de bytes no segmento duplicado é passado em `args[0]`.
`tcpInDataDupSegs`	Teste que é acionado sempre que um segmento é recebido, sendo que todos os dados do segmento tenham sido recebidos anteriormente. O número de bytes no segmento duplicado é passado em `args[0]`.
`tcpInDataInorderBytes`	Teste que é acionado sempre que os dados são recebidos, sendo que *todos* os dados anteriores ao número de seqüência dos dados novos tenham sido recebidos anteriormente. O número de bytes recebidos em ordem é passado em `args[0]`.
`tcpInDataInorderSegs`	Teste que é acionado sempre que um segmento é recebido, sendo que *todos* os dados anteriores ao número de seqüência do novo segmento tenham sido recebidos anteriormente.
`tcpInDataPartDupBytes`	Teste que é acionado sempre que um segmento é recebido, sendo que alguns dados do segmento tenham sido recebidos anteriormente, mas alguns dos dados do segmento sejam novos. O número de bytes duplicados é passado em `args[0]`.
`tcpInDataPartDupSegs`	Teste que é acionado sempre que um segmento é recebido, sendo que alguns dados do segmento tenham sido recebidos anteriormente, mas alguns dos dados do segmento sejam novos. O número de bytes duplicados é passado em `args[0]`.
`tcpInDataPastWinBytes`	Teste que é acionado sempre que são recebidos dados que ficam atrás da janela de recebimento atual. O número de bytes está em `args[0]`.
`tcpInDataPastWinSegs`	Teste que é acionado sempre que um segmento recebido fica atrás da janela de recebimento atual.
`tcpInDataUnorderBytes`	Teste que é acionado sempre que os dados são recebidos, sendo que alguns dados anteriores ao número de seqüência dos novos dados estão ausentes. O número de bytes recebidos fora de ordem é passado em `args[0]`.
`tcpInDataUnorderSegs`	Teste que é acionado sempre que um segmento é recebido, sendo que alguns dados anteriores ao número de seqüência dos novos dados estão ausentes.
`tcpInDupAck`	Teste que é acionado sempre que um ACK duplicado é recebido.
`tcpInErrs`	Teste que é acionado sempre que um erro TCP (por exemplo, uma soma de verificação TCP incorreta) é encontrado no segmento recebido.
`tcpInSegs`	Teste que é acionado sempre que um segmento é recebido, mesmo que mais tarde seja considerado que esse segmento possui um erro que impede a continuação do processamento.

TABELA 28–8 Testes `mib` TCP *(Continuação)*

`tcpInWinProbe`	Teste que é acionado sempre que uma janela de teste é recebida.
`tcpInWinUpdate`	Teste que é acionado sempre que uma atualização de janela é recebida.
`tcpListenDrop`	Teste que é acionado sempre que uma conexão de entrada é descartada devido a uma fila de escuta cheia.
`tcpListenDropQ0`	Teste que é acionado sempre que uma conexão é descartada devido a uma fila cheia de conexões no estado SYN_RCVD.
`tcpOutAck`	Teste que é acionado sempre que um ACK é enviado.
`tcpOutAckDelayed`	Teste que é acionado sempre que um ACK é enviado após ter sido inicialmente atrasado.
`tcpOutControl`	Teste que é acionado sempre que um SYN, FIN ou RST é enviado.
`tcpOutDataBytes`	Teste que é acionado sempre que os dados são enviados. O número de bytes enviados está em `args[0]`.
`tcpOutDataSegs`	Teste que é acionado sempre que um segmento é enviado.
`tcpOutFastRetrans`	Teste que é enviado sempre que um segmento é retransmitido como parte do algoritmo de retransmissão rápido.
`tcpOutRsts`	Teste que é acionado sempre que um segmento é enviado com o sinalizador RST definido.
`tcpOutSackRetransSegs`	Teste que é acionado sempre que um segmento é retransmitido em uma conexão que possui o aviso de recebimento seletivo ativado.
`tcpOutSegs`	Teste que é acionado sempre que é enviado um segmento que contém pelo menos um byte não-retransmitido.
`tcpOutUrg`	Teste que é acionado sempre que é enviado um segmento com o sinalizador URG definido, e com um ponteiro urgente válido.
`tcpOutWinProbe`	Teste que é acionado sempre que uma janela de teste é enviada.
`tcpOutWinUpdate`	Teste que é acionado sempre que uma atualização de janela é enviada.
`tcpPassiveOpens`	Teste que é acionado sempre que conexões TCP tenham feito uma transição direta para o estado SYN_RCVD a partir do estado LISTEN.
`tcpRetransBytes`	Teste que é acionado sempre que os dados são retransmitidos. O número de bytes retransmitidos está em `args[0]`.
`tcpRetransSegs`	Teste que é acionado sempre que é enviado um segmento que contém um ou mais bytes retransmitidos.
`tcpRttNoUpdate`	Teste que é acionado quando os dados foram recebidos, mas não havia informações de carimbo de data/hora disponíveis com as quais atualizar o RTT.

TABELA 28–8 Testes `mib` TCP *(Continuação)*

`tcpRttUpdate`	Teste que é acionado quando foram recebidos dados contendo informações de carimbo de data/hora necessárias para atualizar o RTT.
`tcpTimKeepalive`	Teste que é acionado sempre que um processamento keep-alive é realizado em uma conexão.
`tcpTimKeepaliveDrop`	Teste que é acionado sempre que um processamento keep-alive resulta na terminação de uma conexão.
`tcpTimKeepaliveProbe`	Teste que é acionado sempre que um teste keep-alive é enviado como parte do processamento keep-alive.
`tcpTimRetrans`	Teste que é acionado sempre que um processamento de retransmissão baseado no temporizador é realizado em uma conexão.
`tcpTimRetransDrop`	Teste que é acionado sempre que uma falha prolongada para realizar a retransmissão baseada no temporizador resulta na terminação da conexão.

TABELA 28–9 Testes `mib` UDP

`udpInCksumErrs`	Teste que é acionado sempre que um datagrama é descartado devido a uma soma de verificação UDP incorreta.
`udpInDatagrams`	Teste que é acionado sempre que datagrama UDP é recebido.
`udpInErrors`	Teste que é acionado sempre que um datagrama UDP é recebido, mas é descartado devido a um cabeçalho de pacote malformado ou uma falha na alocação de um buffer interno.
`udpInOverflows`	Teste que é acionado sempre que um datagrama UDP é recebido, mas é subseqüentemente descartado devido à falta de espaço no buffer.
`udpNoPorts`	Teste que é acionado sempre que um datagrama UDP é recebido em uma porta a qual nenhum soquete está vinculado.
`udpOutDatagrams`	Teste que é acionado sempre que datagrama UDP é enviado.
`udpOutErrors`	Teste que é acionado sempre que um datagrama UDP não é enviado devido a uma condição de erro, geralmente porque o datagrama era malformado de alguma maneira.

Argumentos

O único argumento de cada teste `mib` possui a mesma semântica: `args[0]` contém o valor com o qual o contador deve ser incrementado. Para alguns testes `mib`, `args[0]` sempre contém o valor `1`, mas para alguns testes `args[0]` pode tomar valores positivos arbitrários. Para esses testes, o significado de `args[0]` é indicado na descrição do teste.

Estabilidade

O provedor `mib` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA
Argumentos	Desenvolvendo	Desenvolvendo	ISA

◆ ◆ ◆ CAPÍTULO 29

Provedor `fpuinfo`

O provedor `fpuinfo` disponibiliza os testes que correspondem à simulação de instruções de ponto flutuante nos microprocessadores SPARC. Enquanto a maioria das instruções de ponto flutuante são executadas em hardware, algumas operações de ponto flutuante são feitas no sistema operacional para simulação. As condições sob as quais as operações de ponto flutuante requerem simulação do sistema operacional são específicas de uma implementação de microprocessador. As operações que requerem simulação são raras. Entretanto, se um aplicativo usasse uma dessas operações freqüentemente, o efeito no desempenho poderia ser grave. O provedor `fpuinfo` permite a rápida investigação da simulação de ponto flutuante vista através de `kstat(1M)` e da estatística do kernel `fpu_info` ou `trapstat(1M)` e da interceptação `fp-xcp-other`.

Testes

O provedor `fpuinfo` disponibiliza um teste para cada tipo de instrução de ponto flutuante que pode ser simulada. O provedor `fpuinfo` possui uma Estabilidade de Nome de `CPU`; os nomes dos testes são específicos de uma implementação de microprocessador, e podem não estar disponíveis em microprocessadores diferentes na mesma família. Por exemplo, alguns dos testes listados podem estar disponíveis somente no UltraSPARC-III e não no UltraSPARC-III+ ou vice-versa.

Os testes do `fpuinfo` estão descritos na Tabela 29–1.

TABELA 29–1 Testes `fpuinfo`

`fpu_sim_fitoq`	Teste que é acionado sempre que uma instrução `fitoq` é simulada pelo kernel.
`fpu_sim_fitod`	Teste que é acionado sempre que uma instrução `fitod` é simulada pelo kernel.
`fpu_sim_fitos`	Teste que é acionado sempre que uma instrução `fitos` é simulada pelo kernel.

TABELA 29–1 Testes `fpuinfo` *(Continuação)*

`fpu_sim_fxtoq`	Teste que é acionado sempre que uma instrução `fxtoq` é simulada pelo kernel.
`fpu_sim_fxtod`	Teste que é acionado sempre que uma instrução `fxtod` é simulada pelo kernel.
`fpu_sim_fxtos`	Teste que é acionado sempre que uma instrução `fxtos` é simulada pelo kernel.
`fpu_sim_fqtox`	Teste que é acionado sempre que uma instrução `fqtox` é simulada pelo kernel.
`fpu_sim_fdtox`	Teste que é acionado sempre que uma instrução `fdtox` é simulada pelo kernel.
`fpu_sim_fstox`	Teste que é acionado sempre que uma instrução `fstox` é simulada pelo kernel.
`fpu_sim_fqtoi`	Teste que é acionado sempre que uma instrução `fqtoi` é simulada pelo kernel.
`fpu_sim_fdtoi`	Teste que é acionado sempre que uma instrução `fdtoi` é simulada pelo kernel.
`fpu_sim_fstoi`	Teste que é acionado sempre que uma instrução `fstoi` é simulada pelo kernel.
`fpu_sim_fsqrtq`	Teste que é acionado sempre que uma instrução `fsqrtq` é simulada pelo kernel.
`fpu_sim_fsqrtd`	Teste que é acionado sempre que uma instrução `fsqrtd` é simulada pelo kernel.
`fpu_sim_fsqrts`	Teste que é acionado sempre que uma instrução `fsqrts` é simulada pelo kernel.
`fpu_sim_fcmpeq`	Teste que é acionado sempre que uma instrução `fcmpeq` é simulada pelo kernel.
`fpu_sim_fcmped`	Teste que é acionado sempre que uma instrução `fcmped` é simulada pelo kernel.
`fpu_sim_fcmpes`	Teste que é acionado sempre que uma instrução `fcmpes` é simulada pelo kernel.
`fpu_sim_fcmpq`	Teste que é acionado sempre que uma instrução `fcmpq` é simulada pelo kernel.
`fpu_sim_fcmpd`	Teste que é acionado sempre que uma instrução `fcmpd` é simulada pelo kernel.
`fpu_sim_fcmps`	Teste que é acionado sempre que uma instrução `fcmps` é simulada pelo kernel.
`fpu_sim_fdivq`	Teste que é acionado sempre que uma instrução `fdivq` é simulada pelo kernel.
`fpu_sim_fdivd`	Teste que é acionado sempre que uma instrução `fdivd` é simulada pelo kernel.
`fpu_sim_fdivs`	Teste que é acionado sempre que uma instrução `fdivs` é simulada pelo kernel.
`fpu_sim_fdmulx`	Teste que é acionado sempre que uma instrução `fdmulx` é simulada pelo kernel.
`fpu_sim_fsmuld`	Teste que é acionado sempre que uma instrução `fsmuld` é simulada pelo kernel.
`fpu_sim_fmulq`	Teste que é acionado sempre que uma instrução `fmulq` é simulada pelo kernel.
`fpu_sim_fmuld`	Teste que é acionado sempre que uma instrução `fmuld` é simulada pelo kernel.
`fpu_sim_fmuls`	Teste que é acionado sempre que uma instrução `fmuls` é simulada pelo kernel.
`fpu_sim_fsubq`	Teste que é acionado sempre que uma instrução `fsubq` é simulada pelo kernel.
`fpu_sim_fsubd`	Teste que é acionado sempre que uma instrução `fsubd` é simulada pelo kernel.
`fpu_sim_fsubs`	Teste que é acionado sempre que uma instrução `fsubs` é simulada pelo kernel.

TABELA 29–1 Testes `fpuinfo` *(Continuação)*

`fpu_sim_faddq`	Teste que é acionado sempre que uma instrução `faddq` é simulada pelo kernel.
`fpu_sim_faddd`	Teste que é acionado sempre que uma instrução `faddd` é simulada pelo kernel.
`fpu_sim_fadds`	Teste que é acionado sempre que uma instrução `fadds` é simulada pelo kernel.
`fpu_sim_fnegd`	Teste que é acionado sempre que uma instrução `fnegd` é simulada pelo kernel.
`fpu_sim_fnegq`	Teste que é acionado sempre que uma instrução `fneqq` é simulada pelo kernel.
`fpu_sim_fnegs`	Teste que é acionado sempre que uma instrução `fnegs` é simulada pelo kernel.
`fpu_sim_fabsd`	Teste que é acionado sempre que uma instrução `fabsd` é simulada pelo kernel.
`fpu_sim_fabsq`	Teste que é acionado sempre que uma instrução `fabsq` é simulada pelo kernel.
`fpu_sim_fabss`	Teste que é acionado sempre que uma instrução `fabss` é simulada pelo kernel.
`fpu_sim_fmovd`	Teste que é acionado sempre que uma instrução `fmovd` é simulada pelo kernel.
`fpu_sim_fmovq`	Teste que é acionado sempre que uma instrução `fmovq` é simulada pelo kernel.
`fpu_sim_fmovs`	Teste que é acionado sempre que uma instrução `fmovs` é simulada pelo kernel.
`fpu_sim_fmovr`	Teste que é acionado sempre que uma instrução `fmovr` é simulada pelo kernel.
`fpu_sim_fmovcc`	Teste que é acionado sempre que uma instrução `fmovcc` é simulada pelo kernel.

Argumentos

Não há argumentos para testes do `fpuinfo`.

Estabilidade

O provedor `fpuinfo` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	CPU
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	CPU

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Argumentos	Desenvolvendo	Desenvolvendo	CPU

◆ ◆ ◆ CAPÍTULO 30

Provedor `pid`

O provedor `pid` permite rastrear a entrada e o retorno de uma função em um processo do usuário, assim como qualquer instrução conforme especificada por um endereço absoluto ou deslocamento de função. O provedor `pid` não tem efeito quando os testes não estão ativados. Quando os testes estão ativados, eles só induzem efeito de teste nos processos que são rastreados.

Observação – Quando o compilador coloca uma função inline, o teste do provedor `pid` não é acionado. Para evitar que uma função seja colocada inline no momento da compilação, consulte a documentação do seu compilador.

Observação – O provedor `pid` se comporta de forma imprevisível quando testa uma função que usa ponteiros de função para chamar uma subfunção. Você pode colocar testes explicitamente nos endereços da entrada e retorno da função para analisar tais funções.

Nomeando testes `pid`

O provedor `pid` realmente define uma *classe* de provedores. Cada processo pode ter potencialmente seu próprio provedor `pid` associado. Um processo com ID 123, por exemplo, seria rastreado através do provedor `pid123`. Para testes de um desses provedores, a porção do módulo da descrição do teste se refere a um objeto carregado no espaço de endereço do processo correspondente. O exemplo seguinte usa `mdb(1)` para exibir uma lista de objetos:

```
$ 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
```

Na descrição do teste, você nomeia o objeto pelo nome do arquivo, não por seu nome de caminho completo. Você também pode omitir o sufixo `.1` ou `so.1`. Todos os exemplos seguintes nomeiam o mesmo teste:

```
pid123:libc.so.1:strcpy:entry
pid123:libc.so:strcpy:entry
pid123:libc:strcpy:entry
```

O primeiro exemplo é o nome real do teste. Os outros exemplos são alias convenientes que são substituídos internamente pelo nome de objeto de carga completo.

Para o objeto de carga do executável, use o alias `a.out` . As duas descrições de teste seguintes nomeiam o mesmo teste:

```
pid123:csh:main:return
pid123:a.out:main:return
```

Como acontece com todos os testes ancorados do DTrace, o campo de função da descrição do teste nomeia uma função no campo do módulo. Um binário de aplicativo do usuário possui vários nomes para a mesma função. Por exemplo, `mutex_lock` pode ser um nome alternativo da função `pthread_mutex_lock` em `libc.so.1` . O DTrace escolhe um nome canônico para tais funções e usa esse nome internamente. O exemplo seguinte mostra como o DTrace remapeia internamente os nomes de módulo e de função para um formato canônico:

```
# 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
```

Esta renomeação automática significa que os nomes dos testes ativados por você podem ser ligeiramente diferentes daqueles realmente ativados. O nome canônico sempre será consistente entre as execuções do DTrace nos sistemas que executam a mesma versão do Solaris.

Consulte o Capítulo 33, “Rastreio de processo do usuário” para obter exemplos de como usar o provedor `pid` eficientemente.

Testes de limite de função

O provedor `pid` permite que você rastreie a entrada e o retorno de função nos programas do usuário, assim como o provedor FBT fornece esse recurso para o kernel. A maioria dos exemplos deste manual que usam o provedor FBT para rastrear as chamadas da função do kernel podem ser ligeiramente modificados para que se apliquem aos processos do usuário.

Testes `entry`

Um teste `entry` é acionado quando a função rastreada é chamada. Os argumentos dos testes "entry" são valores dos argumentos para a função rastreada.

Testes `return`

Um teste `return` é acionado quando a função rastreada retorna ou faz uma chamada de cauda para outra função. O valor de `arg0` é o deslocamento na função da instrução de return; `arg1` mantém o valor de return.

Observação – O uso de `arg`*N* retorna valores não filtrados e não processados como o tipo `int64_t`. O provedor `pid` não oferece suporte ao formato `args[`*N*`]`.

Testes de deslocamento de função

O provedor `pid` permite que você rastreie qualquer instrução em uma função. Por exemplo, para rastrear os 4 bytes da instrução em uma função `main()`, você poderia usar um comando semelhante ao exemplo seguinte:

```
pid123:a.out:main:4
```

Toda vez que o programa executa a instrução no endereço `main+4`, este teste será ativado. Os argumentos dos testes de deslocamento são indefinidos. A matriz `uregs[]` irá ajudá-lo a examinar o estado do processo nestes locais de teste. Consulte "Matriz `uregs[]`" na página 368 para obter mais informações.

Estabilidade

O provedor `pid` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA
Argumentos	Privada	Privada	Desconhecida

◆ ◆ ◆ CAPÍTULO 31

Provedor `plockstat`

O provedor `plockstat` disponibiliza os testes que podem ser usados para observar o comportamento dos primitivos de sincronização no nível do usuário e manter as horas. O comando `plockstat(1M)` é um consumidor do DTrace que usa o provedor `plockstat` para colher dados nos eventos de bloqueio no nível do usuário.

Visão geral

O provedor `plockstat` disponibiliza testes para os seguintes tipos de evento:

Eventos de contenção Esses testes correspondem à contenção em um primitivo de sincronização no nível do usuário, e são acionados quando um segmento é forçado a aguardar que um recurso se torne disponível. O Solaris é geralmente otimizado para o caso de não-contenção, sendo assim, a contenção prolongada não é esperada; esses testes devem ser usados para o entendimento desses casos onde acontece a contenção. Como a contenção é criada para ser (relativamente) rara, a ativação dos testes de evento de contenção geralmente não possui um efeito de teste sério; os testes podem ser ativados sem preocupação de afetar substancialmente o desempenho.

Eventos de manutenção Esses testes correspondem a adquirir, liberar ou manipular um primitivo de sincronização no nível do usuário. Como tal, esses testes podem ser usados para responder a questões arbitrárias sobre a forma em que os primitivos de sincronização no nível do usuário são manipulados. Como os aplicativos geralmente adquirem e liberam os primitivos de sincronização muito freqüentemente, a ativação dos testes de evento de manutenção pode ter um maior efeito de teste do que os testes de evento de contenção. Enquanto o efeito de teste induzido pela ativação deles pode ser substancial, ele não é patológico; eles ainda podem ser ativados com confiança em aplicativos de produção.

Eventos de erro Esses testes correspondem a qualquer tipo de comportamento anômalo encontrado ao adquirir ou liberar um primitivo de sincronização no nível do usuário. Estes eventos podem ser usados para detectar erros encontrados enquanto um segmento está bloqueando um primitivo de sincronização no nível do usuário. Os eventos de erro devem ser extremamente incomuns, sendo assim, ativá-los não deve induzir um efeito de teste sério.

Testes de mutex

Mutexes reforçam a exclusão mútua em seções críticas. Quando um segmento tentar adquirir um mutex mantido por outro segmento usando o `mutex_lock(3C)` ou `pthread_mutex_lock(3C)`, ele determinará se o segmento proprietário está sendo executado em uma CPU diferente. Se for o caso, o segmento de aquisição irá *girar* por um curto período esperando que o mutex se torne disponível. Se o proprietário não estiver sendo executado em outra CPU, o segmento de aquisição será *bloqueado*.

Os quatro testes `plockstat` pertencentes aos mutexes estão listados na Tabela 31–1. Para cada teste, `arg0` contém um ponteiro para a estrutura `mutex_t` ou `pthread_mutex_t` (estes são tipos idênticos) que representa o mutex.

TABELA 31–1 Testes de mutex

`mutex-acquire`	Teste de evento de manutenção que é acionado imediatamente depois que um mutex é adquirido. `arg1` contém um valor booleano que indica se a aquisição foi recursiva em um mutex recursivo. `arg2` indica o número de iterações que o segmento de aquisição gastou girando neste mutex. `arg2` será diferente de zero somente se o teste `mutex-spin` foi acionado nesta aquisição do mutex.
`mutex-block`	Teste de evento de contenção que é acionado antes de um segmento ser bloqueado em um mutex mantido. `mutex-block` e `mutex-spin` devem ser acionados por uma aquisição de bloqueio simples.
`mutex-spin`	Teste de evento de contenção que é acionado antes de um segmento começar a girar em um mutex mantido. `mutex-block` e `mutex-spin` devem ser acionados por uma aquisição de bloqueio simples.
`mutex-release`	Teste de evento de manutenção que é acionado imediatamente depois que um mutex é liberado. `arg1` contém um valor booleano que indica se o evento corresponde a uma liberação recursiva em um mutex recursivo.
`mutex-error`	Teste de evento de erro que é acionado quando um erro é encontrado em uma operação de mutex. `arg1` é o valor `errno` do erro encontrado.

Testes de bloqueio de leitor/gravador

Os bloqueios de leitor/gravador permitem que vários leitores *ou* um único gravador, mas não ambos, existam em uma seção crítica de cada vez. Esses bloqueios são geralmente usados para estruturas que são pesquisadas mais freqüentemente do que modificadas, ou quando os segmentos gastam tempo substancial em uma seção crítica. Os usuários interagem com bloqueios de leitor/gravador usando as interfaces `rwlock(3C)` ou POSIX `pthread_rwlock_init(3C)` do Solaris .

Os testes que pertencem aos bloqueios de leitores/gravador estão na Tabela 31–2. Para cada teste, `arg0` contém um ponteiro para a estrutura `rwlock_t` ou `pthread_rwlock_t`(estes são tipos idênticos) que representa o bloqueio adaptativo. `arg1` contém um valor booleano que indica se a operação foi um gravador.

TABELA 31–2 Testes de bloqueio de leitores/gravador

`rw-acquire`	Teste de evento de manutenção que é acionado imediatamente depois que um bloqueio de leitores/gravador é adquirido.
`rw-block`	Teste de evento de contenção que é acionado antes que um segmento seja bloqueado ao tentar adquirir um bloqueio. Se ativado, o teste `rw-acquire` ou o teste `rw-error` será acionado depois de `rw-block`.
`rw-release`	Teste de evento de manutenção que é acionado imediatamente depois que um bloqueio de leitor/gravador é liberado
`rw-error`	Teste de evento de erro que é acionado quando um erro é encontrado durante uma operação de bloqueio de leitor/gravador. `arg1` é o valor `errno` do erro encontrado.

Estabilidade

O provedor `plockstat` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA
Argumentos	Desenvolvendo	Desenvolvendo	ISA

◆ ◆ ◆ CAPÍTULO 32

Provedor `fasttrap`

O provedor `fasttrap` permite o rastreio em locais do processo do usuário específicos e pré-programados. Ao contrário da maioria dos outros provedores do DTrace, o provedor `fasttrap` não é projetado para rastrear a atividade do sistema. Em vez disso, esse provedor é uma forma que os consumidores do DTrace têm de inserir informações no framework do DTrace ativando o teste `fasttrap`.

Testes

O provedor `fasttrap` disponibiliza um único teste, `fasttrap:::fasttrap`, que é acionado sempre que um processo no nível do usuário faz uma determinada chamada do DTrace no kernel. A chamada do DTrace para ativar o teste não está publicamente disponível no momento.

Estabilidade

O provedor `fasttrap` usa o mecanismo de estabilidade do DTrace para descrever suas estabilidades, conforme mostrado na tabela seguinte. Para obter mais informações sobre o mecanismo de estabilidade, consulte o Capítulo 39, “Estabilidade”.

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Provedor	Desenvolvendo	Desenvolvendo	ISA
Módulo	Privada	Privada	Desconhecida
Função	Privada	Privada	Desconhecida
Nome	Desenvolvendo	Desenvolvendo	ISA

Elemento	Estabilidade de nome	Estabilidade de dados	Classe de dependência
Argumentos	Desenvolvendo	Desenvolvendo	ISA

◆ ◆ ◆ CAPÍTULO 33

33 Rastreio de processo do usuário

O DTrace é uma ferramenta poderosa para entender o comportamento dos processos do usuário. O DTrace pode ser valiosíssimo durante a depuração, analisando os problemas de desempenho, ou simplesmente entendendo o comportamento de um aplicativo complexo. Este capítulo focaliza os recursos do DTrace relevantes para o rastreio da atividade do processo do usuário e fornece exemplos para o uso deles.

Sub-rotinas `copyin()` e `copyinstr()`

A interação do DTrace com os processos é um pouco diferente da maioria dos depuradores tradicionais ou das ferramentas de observação. Muitas dessas ferramentas parecer ser executadas no escopo do processo, permitindo que os usuários cancelem diretamente a referência aos ponteiros para variáveis de programa. Em vez de serem executados em um processo ou em parte dele, os testes do DTrace são executados no kernel do Solaris. Para acessar os dados do processo, um teste precisa usar as sub-rotinas `copyin()` ou `copyinstr()` para copiar dados do processo do usuário para o espaço de endereço do kernel.

Por exemplo, considere a seguinte chamada do sistema `write(2)`:

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

O programa em D seguinte ilustra uma tentativa incorreta de imprimir o conteúdo de uma seqüência passada para a chamada do sistema `write(2)`:

```
syscall::write:entry
{
    printf("%s", stringof(arg1)); /* incorrect use of arg1 */
}
```

Se você tentar executar este script, o DTrace produzirá mensagens de erro semelhantes ao exemplo seguinte:

```
dtrace: error on enabled probe ID 1 (ID 37: syscall::write:entry): \
    invalid address (0x10038a000) in action #1
```

A variável `arg1`, que contém o valor do parâmetro *buf*, é um endereço que se refere à memória no processo que está executando a chamada do sistema. Para ler a seqüência nesse endereço, use a sub-rotina `copyinstr()` e registre seu resultado com a ação `printf()` :

```
syscall::write:entry
{
    printf("%s", copyinstr(arg1)); /* correct use of arg1 */
```

A saída deste script mostra todas as seqüências que estão sendo passadas para a chamada do sistema `write(2)`. Entretanto, ocasionalmente, você talvez veja uma saída irregular semelhante ao exemplo seguinte:

```
  0     37                      write:entry madaï¿½ï¿½ï¿½
```

A sub-rotina `copyinstr()` age como um argumento de entrada que é o endereço do usuário de uma seqüência ASCII terminada com nulo. Entretanto, os buffers passados para a chamada do sistema `write(2)` podem se referir a dados binários em vez de seqüências ASCII. Para imprimir somente a quantidade da seqüência que o chamador pretende, use a sub-rotina `copyin()`, que usa um tamanho como seu segundo argumento:

```
syscall::write:entry
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

Observe que o operador `stringof` é necessário para que o DTrace converta corretamente os dados do usuário recuperados, usando `copyin ()`, em uma seqüência. O uso de `stringof` não é necessário quando se usa `copyinstr()` porque esta função sempre retorna o tipo `string`.

Evitando erros

As sub-rotinas `copyin()` e `copyinstr()` não podem ler a partir de endereços do usuário que ainda não foram tocados, sendo assim, mesmo um endereço válido pode causar um erro, se a página que contém esse endereço ainda não tiver tido uma falha ao ser acessada. Considere o seguinte exemplo:

```
# 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
```

Na saída do exemplo acima, o aplicativo estava funcionando corretamente, e o endereço em `arg0` era válido, mas ele se referia a uma página que ainda não tinha sido acessada pelo processo correspondente. Para resolver este problema, aguarde que o kernel ou o aplicativo use os dados antes de rastreá-lo. Por exemplo, você pode esperar até que a chamada retorne para aplicar `copyinstr()`, como mostrado no exemplo seguinte:

```
# 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
```

Eliminando a interferência de `dtrace`(1M)

Se rastrear cada chamada para a chamada do sistema `write(2)` , você provocará uma saída em cascada. Cada chamada para `write()` faz o comando `dtrace(1M)` chamar `write()`, já que ela exibe a saída, e assim por diante. Este loop de comentários é um bom exemplo de como o comando `dtrace` pode interferir nos dados desejados. Você pode usar um predicado simples para evitar que esses dados indesejados sejam rastreados:

```
syscall::write:entry
/pid != $pid/
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

A variável de macro `$pid` se expande para o identificador do processo que ativou os testes. A variável `pid` contém o identificador do processo cujo segmento estava sendo executado na CPU onde o teste foi acionado. Portanto, o predicado `/pid != $pid/` garante que o script não rastreie quaisquer eventos relacionados à execução deste próprio script.

Provedor `syscall`

O provedor `syscall` permite que você rastreie cada entrada e retorno da chamada do sistema. As chamadas do sistema podem ser um bom começo para entender o comportamento do processo, especialmente se o processo estiver gastando uma grande quantidade de tempo sendo executado ou bloqueado no kernel. Você pode usar o comando `prstat(1M)` para ver onde os processos estão gastando tempo:

```
$ 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
```

Este exemplo mostra que o processo está consumindo uma grande quantidade de tempo do sistema. Uma possível explicação para este comportamento é que o processo está executando

um grande número de chamadas do sistema. Você pode usar um simples programa em D especificado na linha de comando para ver quais chamadas do sistema estão acontecendo com mais freqüência:

```
# 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
```

Este relatório mostra quais chamadas do sistema estão sendo chamadas com mais freqüência, neste caso, a chamada do sistema write(2). Você pode usar o provedor `syscall` para examinar mais detalhadamente a origem das chamadas do sistema `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
```

A saída mostra que o processo está executando muitas chamadas do sistema `write ()` com uma quantidade relativamente pequena de dados. Esta taxa poderia ser a origem do problema de desempenho deste processo específico. Este exemplo ilustra uma metodologia geral para investigar o comportamento da chamada do sistema.

Ação ustack()

Rastrear uma pilha do segmento do processo na hora em que um teste específico é ativado é freqüentemente útil para examinar um problema em maiores detalhes. A ação `ustack()` rastreia a pilha do segmento do usuário. Se, por exemplo, um processo que abre muitos arquivos falhar ocasionalmente na chamada do sistema open(2), use a ação `ustack()` para descobrir o caminho do código que executa a função `open()` falha:

```
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();
}
```

Este script também ilustra o uso da variável de macro `$1` que usa o valor do primeiro operando especificado na linha de comando do 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
```

A ação `ustack()` registra os valores do contador do programa (PC) da pilha e o dtrace(1M) resolve esses valores de PC para nomes de símbolo, procurando nas tabelas de símbolos do processo. Se o `dtrace` não puder resolver o valor de PC para um símbolo, ele imprimirá o valor como um inteiro hexadecimal.

Se um processo sair ou for interrompido antes de os dados de `ustack()` serem formatados para saída, o `dtrace` talvez não consiga converter os valores de PC no rastreamento da pilha para nomes de símbolo, e será forçado a exibi-los como inteiros hexadecimais. Para resolver esta limitação, especifique um processo de interesse com a opção `-c` ou `-p` para `dtrace`. Consulte o

Capítulo 14, "Utilitário dtrace(1M)" para obter detalhes sobre estas e outras opções. Se o ID do processo ou o comando não for conhecido com antecedência, o exemplo seguinte de programa em D poderá ser usado para resolver a limitação:

```
/*
 * 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;
}
```

O script acima irá parar um processo antes que ele saia, se a ação `ustack()` tiver sido aplicada a um segmento nesse processo. Esta técnica garante que o comando `dtrace` será capaz de resolver os valores de PC para nomes simbólicos. Observe que o valor de `stop_pids[pid]` é definido como 0 depois que ele tiver sido usado para limpar a variável dinâmica. Lembre-se de definir os processos interrompidos que estão sendo executados novamente usando o comando `prun(1)` ou o seu sistema acumulará muitos processos interrompidos.

Matriz uregs[]

A matriz `uregs[]` permite que você acesse registros individuais do usuário. As tabelas seguintes listam índices para a matriz `uregs[]` correspondente a cada arquitetura suportada do sistema Solaris.

TABELA 33–1 Constantes `uregs[]` SPARC

Constante	Registro
`0..R_G..R_G7`	`%g0....%g7` - registros globais
`0..R_O..R_O7`	`%o0....%o7` - registros externos
`0..R_L..R_L7`	`%l0....%l7` - registros locais

TABELA 33–1 Constantes uregs[] SPARC *(Continuação)*

Constante	Registro
0..R_I..R_I7	%i0....%i7 - registros internos
R_CCR	%ccr - registro de código de condição
R_PC	%pc - contador de programa
R_NPC	%npc - próximo contador de programa
R_Y	%y - registro de multiplicação/divisão
R_ASI	%asi - registro do identificador de espaço de endereço
R_FPRS	%fprs - data dos registros de ponto flutuante

TABELA 33–2 Constantes uregs[] x86

Constante	Registro
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
R_CS	%cs
R_ERR	%err

TABELA 33–2 Constantes uregs[] x86 *(Continuação)*

Constante	Registro
R_EFL	%efl
R_UESP	%uesp
R_SS	%ss

Em plataformas AMD64, a matriz uregs possui o mesmo conteúdo que em plataformas x86, além dos elementos adicionais listados na tabela seguinte:

TABELA 33–3 Constantes uregs[] amd64

Constante	Registro
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

Os alias listados na tabela seguinte podem ser usados em todas as plataformas:

TABELA 33–4 Constantes uregs[] comuns

Constante	Registro
R_PC	registro do contador de programa
R_SP	registro do ponteiro de pilha
R_R0	primeiro código de retorno
R_R1	segundo código de retorno

Provedor pid

O provedor pid permite que você rastreie qualquer instrução em um processo. Ao contrário da maioria dos outros provedores, os testes pid são criados por demanda com base nas descrições de teste encontradas em seus programas em D. Como resultado, nenhum teste pid estará listado na saída do dtrace -l até que você os tenha ativado.

Rastreio de limite de função do usuário

O modo mais simples de operação do provedor pid é como um espaço do usuário análogo ao provedor fbt. O programa de exemplo seguinte rastreia todas as entradas e os retornos da função feitos a partir de uma única função. A variável de macro $1 (o primeiro operando na linha de comando) é o ID do processo a ser rastreado. A variável de macro $2 (o segundo operando na linha de comando) é o nome da função a partir da qual serão rastreadas todas as chamadas de função.

EXEMPLO 33–1 userfunc.d: rastrear retorno e entrada de função do usuário

```
pid$1::$2:entry
{
    self->trace = 1;
}

pid$1::$2:return
/self->trace/
{
    self->trace = 0;
}

pid$1:::entry,
pid$1:::return
/self->trace/
{
}
```

Digite o script de exemplo acima e salve-o em um arquivo chamado `userfunc.d` e, em seguida, execute o comando `chmod` para torná-lo executável. Este script produz uma saída semelhante ao exemplo seguinte:

```
# ./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
...
```

O provedor `pid` só pode ser usado em processos que já estão sendo executados. Você pode usar a variável de macro `$target` (consulte o Capítulo 15, "Script") e as opções `-c` and `-p` do `dtrace` para criar e usar processos de interesse e instrumentá-los usando o DTrace. Por exemplo, o script de D seguinte pode ser usado para determinar a distribuição de chamadas de função feitas para `libc` por um processo de assunto específico:

```
pid$target:libc.so::entry
{
    @[probefunc] = count();
}
```

Para determinar a distribuição de tais chamadas feitas pelo comando `date(1)`, salve o script em um arquivo chamado `libc.d` e execute o comando seguinte:

```
# 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
```

Rastreando instruções arbitrárias

Você pode usar o provedor `pid` para rastrear qualquer instrução em qualquer função do usuário. Por demanda, o provedor `pid` criará um teste para cada instrução em uma função. O nome de cada teste é o deslocamento de sua instrução correspondente na função expressa como um inteiro hexadecimal. Por exemplo, para ativar um teste associado à instrução no deslocamento 0x1c na função `foo` do módulo `bar.so` no processo com PID 123, use o seguinte comando:

```
# dtrace -n pid123:bar.so:foo:1c
```

Para ativar todos os testes na função `foo`, incluindo o teste de cada instrução, use o comando:

```
# dtrace -n pid123:bar.so:foo:
```

Esse comando demonstra uma técnica extremamente poderosa para depurar e analisar os aplicativos do usuário. Erros infreqüentes podem ser difíceis de depurar por serem difíceis de reproduzir. Freqüentemente, você pode identificar um problema após a ocorrência de uma falha muito tardiamente para reconstruir o caminho do código. O exemplo seguinte demonstra como combinar o provedor `pid` com rastreio especulativo (consulte o Capítulo 13, “Rastreio especulativo”) para resolver este problema rastreando cada instrução em uma função.

EXEMPLO 33–2 `errorpath.d`: rastrear o caminho de erro da chamada da função do usuário

```
pid$1::$2:entry
{
    self->spec = speculation();
    speculate(self->spec);
    printf("%x %x %x %x %x", arg0, arg1, arg2, arg3, arg4);
}

pid$1::$2:
```

EXEMPLO 33–2 errorpath.d: rastrear o caminho de erro da chamada da função do usuário *(Continuação)*

```
/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;
}
```

Executar errorpath.d resulta em uma saída semelhante ao exemplo seguinte:

```
# ./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
```

◆ ◆ ◆ CAPÍTULO 34

Rastreio definido estaticamente em aplicativos do usuário

O DTrace fornece um recurso para que os desenvolvedores de aplicativo do usuário definam testes personalizados no código do aplicativo para argumentar as capacidades do provedor `pid`. Esses testes estáticos impõem pouca ou nenhuma sobrecarga quando desativados e são ativados dinamicamente como todos os testes do DTrace. Você pode usar os testes estáticos para descrever a semântica do aplicativo para usuários do DTrace sem expor ou requerer conhecimento de implementação dos seus aplicativos. Este capítulo descreve como definir testes estáticos nos aplicativos do usuário e como usar o DTrace para ativar esses testes nos processos do usuário.

Escolhendo os pontos de teste

O DTrace permite que os desenvolvedores incorporem pontos de teste estáticos no código do aplicativo, incluindo aplicativos completos e bibliotecas compartilhadas. Esses testes podem ser ativados sempre que o aplicativo ou a biblioteca está sendo executado, seja no desenvolvimento ou na produção. Você deve definir testes que possuem um significado semântico que já seja conhecido pela comunidade de usuários do DTrace. Por exemplo, você poderia definir os testes `query-receive` e `query-respond` em um servidor da Web que corresponda a um cliente que envia uma solicitação e o servidor da Web que responde a essa solicitação. Esses testes de exemplo são facilmente entendidos pela maioria dos usuários do DTrace e correspondem às abstrações de mais alto nível do aplicativo, em vez dos detalhes de implementação de mais baixo nível. Os usuários do DTrace usam esses testes para entender a distribuição de tempo das solicitações. Se o seu teste `query-receive` apresentou as seqüências de solicitação de URL como um argumento, um usuário do DTrace poderia determinar quais solicitações estavam gerando a maior parte da E/S de disco, combinando esse teste com o provedor `io` .

Você também deve considerar a estabilidade das abstrações que descreve ao escolher os nomes e os locais de teste. Esse teste persistirá nas versões futuras do aplicativo, mesmo se a implementação for alterada? O teste faz sentido em todas as arquiteturas de sistema ou é específico de um conjunto de instruções em particular? Este capítulo discutirá os detalhes de como estas decisões orientam as definições de rastreio estático.

Adicionando testes a um aplicativo

Os testes do DTrace em bibliotecas e executáveis são definidos em uma seção ELF no binário do aplicativo correspondente. Esta seção descreve como definir seus testes, adicioná-los ao código-fonte do seu aplicativo, e argumenta o processo de construção do seu aplicativo para incluir as definições de teste do DTrace.

Definindo provedores e testes

Você define os testes do DTrace em um arquivo-fonte `.d` que é, em seguida, usado durante a compilação e vinculação do seu aplicativo. Primeiro, selecione um nome apropriado no seu provedor de aplicativo do usuário. O nome do provedor que você escolher será anexado ao identificador de cada processo que esteja executando o código do seu aplicativo. Por exemplo, se você escolhesse o nome de provedor `myserv` para um servidor da Web que estava sendo executado como o ID de processo 1203, o nome do provedor do DTrace correspondente a este processo seria `myserv1203`. Em seu arquivo-fonte `.d`, adicione uma definição de provedor semelhante ao exemplo seguinte:

```
provider myserv {
    ...
};
```

Em seguida, adicione uma definição para cada teste e os argumentos correspondentes. O exemplo seguinte define os dois testes discutidos em "Escolhendo os pontos de teste" na página 375. O primeiro teste possui dois argumentos, ambos do tipo `string`, e o segundo teste não possui argumentos. O compilador de D converte dois sublinhados consecutivos (`__`) em qualquer nome de teste em um hífen (`-`).

```
provider myserv {
    probe query__receive(string, string);
    probe query__respond();
};
```

Você deve adicionar os atributos de estabilidade à definição do seu provedor para que os consumidores de testes entendam a probabilidade de alteração nas futuras versões do seu aplicativo. Consulte o Capítulo 39, "Estabilidade" para obter mais informações sobre os atributos de estabilidade do DTrace. Os atributos de estabilidade são definidos como mostrado no exemplo seguinte:

EXEMPLO 34–1 `myserv.d`: testes de aplicativo definidos estaticamente

```
#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
```

EXEMPLO 34–1 `myserv.d`: testes de aplicativo definidos estaticamente *(Continuação)*

```
#pragma D attributes Evolving/Evolving/Common provider myserv args

provider myserv {
    probe query__receive(string, string);
    probe query__respond();
};
```

Observação – Os scripts de D que usam argumentos não inteiros dos testes adicionados pelo usuário devem usar as funções `copyin()` e `copyinstr()` para recuperar tais argumentos. Consulte o Capítulo 33, "Rastreio de processo do usuário" para obter mais informações.

Adicionando testes ao código do aplicativo

Agora que você definiu seus testes em um arquivo `.d`, é necessário argumentar seu código-fonte para indicar os locais que devem disparar seus testes. Considere o exemplo seguinte de código-fonte de aplicativo em C:

```
void
main_look(void)
{
    ...
    query = wait_for_new_query();
    process_query(query)
    ...
}
```

Para adicionar um local de teste, acrescente uma referência à macro `DTRACE_PROBE()` definida em `<sys/sdt.h>` como mostrado no exemplo seguinte:

```
#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)
    ...
}
```

O sufixo 2 no nome da macro DTRACE_PROBE2 se refere ao número de argumentos que são passados para o teste. Os dois primeiros argumentos para a macro do teste são o nome do provedor e o nome do teste e devem corresponder ao seu provedor e definições de teste de D. Os argumentos da macro restantes são aqueles atribuídos às variáveis arg0..9 do DTrace quando o teste é acionado. O código-fonte do seu aplicativo pode conter várias referências ao mesmo provedor e nome de teste. Se houver várias referências ao mesmo teste em seu código-fonte, qualquer uma dessas referências de macro farão com que o teste seja acionado.

Construindo aplicativos com testes

Você deve aumentar o processo de construção em seu aplicativo para incluir o provedor e as definições de teste do DTrace. Um processo de construção típico pega cada arquivo-fonte e compila-o para criar o arquivo de objeto correspondente. Os arquivos de objeto compilados são, então, vinculados para criar o binário do aplicativo finalizado, conforme mostrado no exemplo seguinte:

```
cc -c src1.c
cc -c src2.c
...
cc -o myserv src1.o src2.o ...
```

Para incluir as definições de teste do DTrace em seu aplicativo, adicione as regras Makefile apropriadas ao seu processo de construção para executar o comando dtrace, conforme mostrado no exemplo seguinte:

```
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 ...
```

O comando dtrace mostrado acima pós-processa os arquivos de objeto gerados pelos comandos do compilador precedente e gera o arquivo de objeto myserv.o a partir de myserv.d e os outros arquivos de objeto. O opção -G do dtrace é usada para vincular o provedor e as definições de teste ao aplicativo do usuário. A opção - 32 é usada para construir binários de 32 bits. A opção - 64 é usada para construir binários de 64 bits.

◆ ◆ ◆ CAPÍTULO 35

35 Segurança

Este capítulo descreve os privilégios que os administradores de sistema podem usar para conceder acesso ao DTrace a usuários ou processos específicos. O DTrace permite a visibilidade em todos os aspectos do sistema incluindo funções no nível do usuário, chamadas do sistema, funções do kernel, etc. Ele permite ações poderosas, algumas das quais podem modificar o estado do programa. Assim como seria inapropriado permitir que um usuário tivesse acesso aos arquivos privados de outro usuário, um administrador de sistema não deve conceder acesso completo a cada usuário para todos os recursos que o DTrace oferece. Por padrão, somente o super-usuário pode usar o DTrace. O recurso Least Privilege pode ser usado para permitir a outros usuários o uso controlado do DTrace.

Privilégios

O recurso Least Privilege do Solaris permite aos administradores concederem privilégios específicos a usuários específicos do Solaris. Para dar a um usuário um privilégio durante o logon, insira uma linha no arquivo `/etc/user_attr` no formato:

user-name`::::defaultpriv=basic,`*privilege*

Para dar a um processo em execução um privilégio adicional, use o comando `ppriv(1)`.

`# ppriv -s A+`*privilege process-ID*

Os três privilégios que controlam o acesso de um usuário aos recursos do DTrace são `dtrace_proc`, `dtrace_user` e `dtrace_kernel`. Cada privilégio permite o uso de um certo conjunto de provedores, ações e variáveis do DTrace, e cada um corresponde a um tipo específico de uso do DTrace. Os modos de privilégio são descritos em detalhes nas seções seguintes. Os administradores de sistema devem pesar cuidadosamente as necessidades de cada usuário em relação à visibilidade e o impacto no desempenho dos diferentes modos de privilégio. Os usuários precisam de pelo menos um dos três privilégios do DTrace para usar qualquer funcionalidade do DTrace.

Uso privilegiado do DTrace

Os usuários com qualquer um dos três privilégios do DTrace podem ativar os testes fornecidos pelo provedor `dtrace` (consulte o Capítulo 17, “Provedor do `dtrace`”) e podem usar as seguintes ações e variáveis:

Provedores	`dtrace`		
Ações	`exit`	`printf`	`tracemem`
	`discard`	`speculate`	
	`printa`	`trace`	
Variáveis	`args`	`probemod`	`this`
	`epid`	`probename`	`timestamp`
	`id`	`probeprov`	`vtimestamp`
	`probefunc`	`self`	
Espaços de endereço	Nenhum		

Privilégio `dtrace_proc`

O privilégio `dtrace_proc` permite usar o provedor `fasttrap` em rastreios no nível do processo. Ele também permite o uso das seguintes ações e variáveis:

Ações	`copyin`	`copyout`	`stop`
	`copyinstr`	`raise`	`ustack`
Variáveis	`execname`	`pid`	`uregs`
Espaços de endereço	Usuário		

Este privilégio não concede qualquer visibilidade às estruturadas de dados do kernel do Solaris ou aos processos para os quais o usuário não tem permissão.

Os usuários com esse privilégio podem criar e ativar testes nos processos que possuem. Se o usuário também possuir o privilégio `proc_owner`, os testes poderão ser criados e ativados em qualquer processo. O privilégio `dtrace_proc` destina-se a usuários interessados nas análises de depuração ou de desempenho dos processos do usuário. Esse privilégio é ideal para um desenvolvedor que está trabalhando em um novo aplicativo ou um engenheiro que está tentando aprimorar o desempenho de um aplicativo em um ambiente de produção.

Observação – Os usuários com os privilégios `dtrace_proc` e `proc_owner` podem *ativar* qualquer teste `pid` a partir de quaisquer processos, mas podem somente criar testes em processos cujo privilégio definido é um subconjunto do seu próprio conjunto de privilégios. Consulte a documentação do Least Privilege para obter detalhes completos.

O privilégio `dtrace_proc` permite o acesso ao DTrace que pode impor uma penalidade de desempenho somente naqueles processos para os quais o usuário tem permissão. Os processos instrumentados irão impor mais carga nos recursos do sistema, sendo assim, pode haver um pequeno impacto no desempenho geral do sistema. Além desse aumento na sobrecarga geral, esse privilégio não permite qualquer instrumentação que impacte no desempenho de quaisquer processos que não estejam sendo rastreados. Como esse privilégio não concede aos usuários qualquer visibilidade adicional de outros processos ou do próprio kernel, recomenda-se que esse privilégio seja concedido a todos os usuários que possam precisar entender melhor o funcionamento interno dos seus próprios processos.

Privilégio `dtrace_user`

O privilégio `dtrace_user` permite o uso dos provedores `profile` e `syscall` com algumas precauções, e o uso das seguintes ações e variáveis:

Provedores	`profile`	`syscall`	`fasttrap`
Ações	`copyin`	`copyout`	`stop`
	`copyinstr`	`raise`	`ustack`
Variáveis	`execname`	`pid`	`uregs`
Espaços de endereço	Usuário		

O privilégio `dtrace_user` fornece visibilidade apenas dos processos para os quais o usuário já tem permissão; ele não permite qualquer visibilidade do estado ou da atividade do kernel. Com esse privilégio, os usuários podem ativar o provedor `syscall`, mas os testes somente serão ativados nos processos para os quais o usuário tem permissão. Similarmente, o provedor `profile` pode ser ativado, mas os testes só serão ativados nos processos para os quais o usuário tem permissão, nunca no kernel do Solaris.

Esse privilégio permite o uso da instrumentação que, embora permita a visibilidade somente de processos específicos, pode afetar o desempenho geral do sistema. O provedor `syscall` possui um pequeno impacto no desempenho em cada chamada do sistema em cada processo. O provedor `profile` afeta o desempenho geral do sistema, executando cada intervalo de tempo, semelhante a um temporizador em tempo real. Nenhuma dessas degradações de desempenho é

tão grande que limite gravemente o progresso do sistema, mas os administradores do sistema devem considerar as implicações de conceder esse privilégio a um usuário. Consulte o Capítulo 21, "Provedor syscall" e o Capítulo 19, "Provedor profile" para ver uma discussão do impacto no desempenho dos provedores syscall e profile.

Privilégio dtrace_kernel

O privilégio dtrace_kernel permite o uso de todos os provedores, exceto os provedores pid e fasttrap, em processos que não pertencem ao usuário. Esse privilégio também permite o uso de todas as ações e variáveis, exceto de ações destrutivas do kernel (breakpoint(), panic (), chill ()). Esse privilégio permite a visibilidade completa do kernel e do estado do usuário. Os recursos ativados pelo privilégio dtrace_user são um subconjunto restrito daqueles ativados por dtrace_kernel.

Provedores	Todos com as restrições acima	
Ações	Todas menos as ações destrutivas	
Variáveis	Todas	
Espaços de endereço	Usuário	Kernel

Privilégios de superusuário

Um usuário com todos os privilégios pode usar cada provedor e cada ação, incluindo as ações destrutivas do kernel indisponíveis para todas as outras classes de usuário.

Provedores	Todas	
Ações	Todas, incluindo as ações destrutivas	
Variáveis	Todas	
Espaços de endereço	Usuário	Kernel

◆ ◆ ◆ CAPÍTULO 36

36 Rastreio anônimo

Este capítulo descreve o rastreio *anônimo*: aquele que não é associado a nenhum consumidor do DTrace. O rastreio anônimo é usado quando não é possível executar processos de consumidor do DTrace. O uso mais comum do rastreio anônimo é permitir que desenvolvedores de driver de dispositivo depurem e rastreiem uma atividade que ocorra durante a inicialização do sistema. Qualquer rastreio que possa ser feito interativamente pode ser feito anonimamente. Entretanto, somente o superusuário pode criar uma ativação anônima, e somente uma ativação anônima pode existir a qualquer momento.

Ativações anônimas

Para criar uma ativação anônima, use a opção `-A` com uma chamada `dtrace`(1M) que especifica os testes, predicados, ações e opções desejados. O `dtrace` adicionará várias propriedades de driver representando sua solicitação ao arquivo de configuração do driver `dtrace`(7D), geralmente `/kernel/drv/dtrace.conf`. Essas propriedades serão lidas pelo driver `dtrace`(7D) quando ele for carregado. O driver ativará as ações especificadas nos testes especificados e criará um *estado anônimo* para associar à nova ativação. Normalmente, o driver `dtrace`(7D) é carregado por demanda, assim como os drivers que atuam como provedores do DTrace. Para permitir o rastreio durante a inicialização, o driver `dtrace`(7D) deve ser carregado o mais breve possível. O `dtrace` adiciona as instruções `forceload` necessárias a `/etc/system` (consulte `system`(4)) de cada provedor solicitado do DTrace e do próprio `dtrace`(7D).

Em seguida, quando o sistema é inicializado, uma mensagem é emitida pelo `dtrace`(7D) para indicar que o arquivo de configuração foi processado com êxito.

Todas as opções devem ser definidas com uma ativação anônima, incluindo o tamanho do buffer, o tamanho de variável dinâmica, o tamanho de especulação, o número de especulações, etc.

Para remover uma ativação anônima, especifique `-A` para `dtrace` sem descrições de teste.

Declarando estado anônimo

Quando o computador tiver sido totalmente inicializado, o estado anônimo deve ser declarado especificando-se a opção `-a` com `dtrace`. Por padrão, `-a` declara o estado anônimo, processa os dados existentes e continua sendo executada. Para consumir o estado anônimo e encerrar, adicione a opção `-e`.

Depois que o estado anônimo tiver sido consumido no kernel, ele não poderá ser substituído: os buffers no kernel que o continham são reutilizados. Se você tentar declarar um estado de rastreio anônimo onde não houver um, o `dtrace` irá gerar uma mensagem similar ao seguinte exemplo:

```
dtrace: could not enable tracing: No anonymous tracing state
```

Se eliminações ou erros tiverem ocorrido, o `dtrace` irá gerar as mensagens apropriadas quando o estado anônimo for declarado. As mensagens de eliminações e erros são as mesmas do estado anônimo e do não-anônimo.

Exemplos de rastreio anônimo

O exemplo a seguir mostra uma ativação anônima do DTrace em cada teste no módulo `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
```

Após a reinicialização, o `dtrace`(7D) imprime uma mensagem no console para indicar que ele está ativando os testes especificados:

```
  ...
  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.
  ...
```

Quando o computador for reinicializado, o estado anônimo deve ser consumido especificando-se a opção `-a` com `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
  ...
```

O exemplo a seguir focaliza somente nas funções chamadas de `iprbattach()`. Em um editor, digite o seguinte script e salve-o em um arquivo chamado `iprb.d`.

```
fbt::iprbattach:entry
{
    self->trace = 1;
}

fbt:::
/self->trace/
{}

fbt::iprbattach:return
{
    self->trace = 0;
}
```

Execute os seguintes comandos para cancelar as configurações anteriores do arquivo de configuração do driver, instale a nova solicitação de rastreio anônimo e reinicialize:

```
# 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
```

Após a reinicialização, o dtrace(7D) imprime uma mensagem diferente no console para indicar a ativação ligeiramente diferente:

```
...
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.
...
```

Após a completa reinicialização do computador, execute o `dtrace` com as opções `-a` e `-e` para consumir os dados anônimos e depois encerrar.

```
# 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
...
```

◆ ◆ ◆ CAPÍTULO 37

Rastreio post-mortem

Este capítulo descreve os recursos do DTrace para extração e processamento *post-mortem* dos dados no kernel de consumidores do DTrace. No caso de um travamento do sistema, as informações que foram registradas com o DTrace podem fornecer dicas fundamentais para descobrir a causa da falha do sistema. Os dados do DTrace podem ser extraídos e processados do despejo de memória do sistema para ajudá-lo na compreensão de falhas graves do sistema. Ao unir esses recursos post-mortem do DTrace com sua política de armazenamento em buffer em anel (consulte o Capítulo 11, "Buffers e armazenamento em buffer"), o DTrace pode ser usado como um sistema operacional semelhante à *caixa preta*, o gravador de dados de vôo existente na aviação comercial.

Para extrair dados do DTrace de um despejo de memória específico, você deve começar executando o Depurador modular Solaris, `mdb(1)`, no despejo de memória de seu interesse. O módulo MDB que contém o recurso do DTrace será carregado automaticamente. Para aprender mais sobre o MDB, consulte o *Solaris Modular Debugger Guide*.

Exibindo consumidores do DTrace

Para extrair dados do DTrace de um consumidor do DTrace, você deve primeiro determinar o consumidor do DTrace desejado, executando o dcmd do 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
```

Este comando exibe uma tabela das estruturas de estado do DTrace. Cada linha da tabela consiste nas seguintes informações:

- O endereço da estrutura de estado.

- O número secundário associado ao dispositivo dtrace(7D)
- O endereço da estrutura do processo que corresponde ao consumidor do DTrace.
- O nome do consumidor do DTrace (ou <anonymous> dos consumidores anônimos).
- O nome da estrutura do arquivo que corresponde ao dispositivo dtrace(7D) aberto.

Para obter mais informações sobre um consumidor específico do DTrace, especifique o endereço de sua estrutura de processo como o dcmd ::ps:

```
> d71377f0::ps
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R 100647 100642 100647 100638      0 0x00004008 d71377f0 dtrace
```

Exibindo dados de rastreio

Depois que você determinar o consumidor de seu interesse, poderá recuperar os dados correspondentes aos buffers não consumidos, especificando o endereço da estrutura do estado como o dcmd ::dtrace. O exemplo a seguir mostra o resultado do dcmd ::dtrace em uma habilitação anônima de syscall:::entry com a ação 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
...
```

O dcmd `::dtrace` trata os erros da mesma maneira que o `dtrace(1M)`: se eliminações, erros, eliminações especulativas ou similares forem encontrados durante a execução do consumidor, o `::dtrace` emitirá uma mensagem correspondente à mensagem do `dtrace(1M)`.

A ordem dos eventos exibidos por `::dtrace` é sempre do mais antigo para o mais recente em uma determinada CPU. Os buffers da CPU são exibidos em ordem numérica. Se uma ordem for necessária para eventos em CPUs diferentes, rastreie a variável `timestamp`.

Você pode exibir somente os dados de uma determinada CPU, especificando a opção `-c` como `::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
...
```

Observe que `::dtrace` processa somente dados do DTrace *no kernel* . Os dados que tenham sido consumidos no kernel e processados (através do `dtrace(1M)` ou outros meios) não estarão disponíveis para ser processados com o `::dtrace`. Para garantir que a maior quantidade possível de dados esteja disponível no momento da falha, use uma política de armazenamento em buffer em anel. Consulte o Capítulo 11, “Buffers e armazenamento em buffer” para obter mais informações sobre as políticas de buffer.

O exemplo a seguir cria um buffer em anel muito pequeno (16K) e registra todas as chamadas do sistema e o processo que as faz:

```
# dtrace -P syscall'{trace(curpsinfo->pr_psargs)}' -b 16k -x bufpolicy=ring
dtrace: description 'syscall:::entry' matched 214 probes
```

A observação de um despejo de memória obtido quando o comando acima estava sendo executado produz um resultado similar ao seguinte exemplo:

```
> ::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
>
```

Observe que os registros mais recentes da CPU 1 incluem uma série de chamadas do sistema de `write(2)` por um processo `mdb -kw`. Esse resultado provavelmente está relacionado ao motivo da falha do sistema porque um usuário pode modificar texto ou dados em execução no kernel com `mdb(1)` quando executados com as opções `-k e -w`. Nesse caso, os dados do DTrace fornecem pelo menos um meio interessante de investigação, se não a causa principal da falha.

◆ ◆ ◆ CAPÍTULO 38

Considerações sobre o desempenho

Como o DTrace causa trabalho adicional no sistema, sua ativação sempre afeta o desempenho do sistema de alguma maneira. Geralmente, esse efeito não é significativo, mas pode se tornar substancial se houver muitas ativações de testes de custo elevado. Este capítulo descreve técnicas para minimizar o efeito no desempenho do DTrace.

Limitar testes ativados

Técnicas de instrumentação dinâmicas permitem que o DTrace forneça uma cobertura de rastreio sem igual do kernel e dos processos arbitrários do usuário. Embora essa cobertura permita uma nova e revolucionária abordagem do comportamento do sistema, ela também pode causar um efeito enorme nos testes. Se milhares de testes forem ativados, o efeito no sistema pode, facilmente, ser substancial. Portanto, você deve ativar somente o número de testes necessários para solucionar um problema. Você não deve, por exemplo, ativar todos os testes FBT se uma ativação mais concisa for capaz de responder à sua pergunta. Por exemplo, sua pergunta pode permitir que você se concentre em um módulo específico de interesse ou uma função específica.

Ao usar o provedor `pid`, você deve ter um cuidado especial. Como o provedor `pid` pode instrumentar todas as *instruções*, você pode ativar milhões de testes em um aplicativo, tornando o processo de destino extremamente lento.

O DTrace também pode ser usado em situações nas quais grandes quantidades de testes *precisem* ser ativadas para que uma pergunta seja respondida. A ativação de muitos testes pode tornar um sistema um pouco lento, mas nunca induzirá a uma falha grave no computador. Assim, você não deve hesitar se for necessário ativar muitos testes.

Usar agregações

Conforme discutido no Capítulo 9, "Agregações", as agregações do DTrace permitem uma maneira escalável de agregar dados. As matrizes de associação podem parecer possuir uma funcionalidade similar às agregações. Entretanto, por serem variáveis de natureza global, com várias finalidades, elas não podem oferecer a escalabilidade linear das agregações. Você deve, portanto, preferir usar agregações em matrizes de associação quando possível. O exemplo a seguir não é recomendável:

```
syscall::entry
{
    totals[execname]++;
}

syscall::rexit:entry
{
    printf("%40s %d\n", execname, totals[execname]);
    totals[execname] = 0;
}
```

O exemplo a seguir é preferível:

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

END
{
    printa("%40s %@d\n", @totals);
}
```

Usar predicados em cache

Os predicados do DTrace são usados para filtrar dados indesejados do experimento, rastreando dados que só são rastreados se uma condição especificada for verdadeira. Ao ativar muitos testes, você geralmente usa predicados de uma forma que identifique um segmento específico ou segmentos do seu interesse, como o `/self->traceme/` ou o `/pid == 12345/`. Embora muitos desses predicados tenham o valor falso na maioria dos segmentos de grande parte dos testes, a própria avaliação pode ter um alto custo quando feita para milhares de testes. Para reduzir esse custo, o DTrace armazena em cache o valor do predicado se ele incluir somente variáveis de segmentos locais (por exemplo, `/self->traceme/`) ou variáveis imutáveis (por exemplo, `/pid == 12345/`). O custo da avaliação de um predicado em cache é muito menor que o custo da avaliação de um predicado que não esteja em cache, especialmente se o predicado envolver variáveis de segmentos locais, comparações de seqüências, ou outras operações de

custo relativamente elevado. Embora o cache de predicados seja transparente para o usuário, é necessário que sejam seguidas algumas instruções para a construção de predicados perfeitos, conforme mostrado na tabela a seguir:

Pode ser armazenado em cache	Não pode ser armazenado em 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->`*qualquer membro* , `curlwpsinfo->`*qualquer membro*, `curpsinfo->`*qualquer membro*

O exemplo a seguir não é recomendável:

```
syscall::read:entry
{
    follow[pid, tid] = 1;
}

fbt:::
/follow[pid, tid]/
{}

syscall::read:return
/follow[pid, tid]/
{
    follow[pid, tid] = 0;
}
```

O exemplo a seguir usando variáveis de segmentos locais é preferível:

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

fbt:::
/self->follow/
{}

syscall::read:return
/self->follow/
{
```

```
    self->follow = 0;
}
```

Um predicado deve consistir *exclusivamente* em expressões armazenáveis em cache para poder ser armazenado em cache. Todos os predicados a seguir podem ser armazenados em cache:

```
/execname == "myprogram"/
/execname == $$1/
/pid == 12345/
/pid == $1/
/self->traceme == 1/
```

Os exemplos a seguir, que usam variáveis globais, não podem ser armazenados em cache:

```
/execname == one_to_watch/
/traceme[execname]/
/pid == pid_i_care_about/
/self->traceme == my_global/
```

◆ ◆ ◆ CAPÍTULO 39

39 Estabilidade

A Sun freqüentemente fornece aos desenvolvedores o acesso imediato às novas tecnologias, assim como ferramentas de observação que permitem que os usuários analisem profundamente os detalhes da implementação interna do software do kernel e do usuário. Infelizmente, as novas tecnologias e os detalhes da implementação interna estão propensos a alterações à medida que as interfaces e as implementações evoluem e se desenvolvem quando o software é atualizado ou corrigido. A Sun documenta os níveis de estabilidade de aplicativo e da interface usando um conjunto de rótulos descritos na página `attributes(5)` do manual para ajudar a definir as expectativas do usuário em relação a quais tipos de alterações devem ocorrer em tipos diferentes de versões futuras.

Nenhum atributo de estabilidade descreve apropriadamente o conjunto arbitrário de entidades e serviços que podem ser acessados de um programa em D. O DTrace e o compilador de D, portanto, incluem recursos para calcular e descrever dinamicamente os níveis de estabilidade dos programas em D que forem criados. Este capítulo aborda os recursos do DTrace para determinar a estabilidade do programa para ajudá-lo a criar programas em D estáveis. Você pode usar os recursos de estabilidade do DTrace para informá-lo dos atributos de estabilidade dos programas em D, ou para produzir erros em tempo de compilação quando o programa tiver dependências de interface indesejáveis.

Níveis de estabilidade

O DTrace fornece dois tipos de atributos de estabilidade para entidades como variáveis incorporadas, funções e testes: um *nível de estabilidade* e uma *classe de dependência* de arquitetura. O nível de estabilidade do DTrace auxilia-o na avaliação de riscos ao desenvolver scripts e ferramentas com base no DTrace, indicando a probabilidade de uma interface ou entidade do DTrace ser alterada em uma futura versão ou patch. A classe de dependência do DTrace informa se uma interface é comum a todas as plataformas e processadores do Solaris ou se a interface é associada a uma arquitetura em particular como somente os processadores SPARC. Os dois tipos de atributos usados para descrever interfaces podem variar independentemente.

Os valores de estabilidade usados pelo DTrace aparecem na seguinte lista da menor para a maior estabilidade. As interfaces mais estáveis podem ser usadas por todos os programas em D e aplicativos em camadas porque a Sun fará o possível para garantir que eles continuem a funcionar em versões secundárias futuras. Os aplicativos que dependem somente de interfaces estáveis devem continuar funcionando corretamente em versões secundárias futuras e não serão interrompidos por patches interinos. As interfaces menos estáveis permitem a experimentação, criação de protótipos, ajuste e depuração no sistema atual, mas devem ser usadas sabendo-se que elas podem se tornar incompatíveis ou até mesmo ser eliminadas ou substituídas por alternativas em versões secundárias futuras.

Os valores de estabilidade do DTrace também auxiliam na compreensão da estabilidade das entidades de software que estão sendo observadas, além da estabilidade das próprias interfaces do DTrace. Portanto, os valores de estabilidade do DTrace também informam a probabilidade dos programas em D e ferramentas em camadas precisarem de alterações correspondentes quando você atualizar ou alterar a pilha de software que está sendo observada.

Interna	A interface é privada do DTrace e representa um detalhe de implementação do DTrace. As interfaces internas podem ser alteradas em micro versões ou versões secundárias.
Privada	A interface é privada da Sun e representa uma interface desenvolvida para uso por outros produtos da Sun que ainda não foram publicamente documentados para uso por clientes e fornecedores independentes de software. As interfaces privadas podem ser alteradas em micro versões ou versões secundárias.
Obsoleta	Há suporte para a interface na versão atual, mas ela está agendada para ser removida, mais provavelmente em uma versão secundária futura. Quando o suporte a uma interface estiver para ser interrompido, a Sun tentará fornecer uma notificação antes de interromper o suporte à interface. O compilador de D produzirá mensagens de aviso se você tentar usar uma interface obsoleta.
Externa	A Sun pode fornecer, por decisão própria, versões atualizadas e possivelmente incompatíveis de tais interfaces como parte de qualquer versão, de acordo com a disponibilidade da entidade controladora. A Sun não garante compatibilidade binária ou de fonte para interfaces internas entre duas versões. Os aplicativos baseados nessas interfaces podem não funcionar em versões futuras, incluindo patches que contenham interfaces externas.
Instável	A interface é fornecida para possibilitar aos desenvolvedores um acesso imediato a tecnologias novas ou em constante alteração ou a um artefato de implementação que seja essencial para a observação ou depuração do comportamento do sistema para o qual uma solução mais estável seja prevista no futuro. A Sun não garante compatibilidade binária ou de fonte para interfaces instáveis de uma versão secundária para outra.

Desenvolvendo	A interface pode eventualmente se tornar padrão ou estável, mas ainda está em transição. A Sun se esforçará para garantir a compatibilidade com versões anteriores à medida que ela se desenvolver. Quando alterações de compatibilidade não-ascendente forem necessárias, elas ocorrerão em versões secundárias e principais. Essas alterações serão evitadas nas micro versões sempre que possível. Se tal alteração for necessária, ela será documentada nas notas de versão da versão afetada e, quando viável, a Sun fornecerá auxílio de migração para compatibilidade binária e contínuo desenvolvimento do programa em D.
Estável	A interface é uma interface desenvolvida sob controle da Sun. A Sun tentará evitar alterações de compatibilidade não-ascendente nessas interfaces, especialmente em micro versões ou versões secundárias. Se o suporte à interface estável tiver que ser interrompido, a Sun tentará fornecer notificação e o nível de estabilidade será alterado para obsoleta.
Padrão	A interface está em conformidade com um padrão de mercado. A documentação correspondente da interface irá descrever o padrão com o qual a interface está em conformidade. Os padrões geralmente são controlados por uma organização de desenvolvimento de padrões e alterações podem ser feitas na interface de acordo com as alterações aprovadas no padrão. O nível de estabilidade também pode ser aplicado em interfaces que tenham sido adotadas sem um padrão formal por uma convenção de mercado. O suporte é fornecido somente para as versões especificadas de um padrão; o suporte para versões posteriores não é garantido. Se a organização de desenvolvimento de padrões aprovar uma alteração de compatibilidade não-ascendente em uma interface padrão para a qual a Sun decida fornecer suporte, a Sun anunciará a estratégia de compatibilidade e migração.

Classes de dependência

Como o Solaris e o DTrace oferecem suporte a várias plataformas operacionais e processadores, o DTrace também rotula interfaces com uma *classe de dependência* que informa se uma interface é comum a todas as plataformas e aos processadores Solaris ou se a interface é associada a uma arquitetura de sistema em particular. A classe de dependência é ortogonal para os níveis de estabilidade descritos anteriormente. Por exemplo, uma interface do DTrace pode ser estável mas com suporte somente em microprocessadores SPARC ou pode ser instável mas comum a todos os sistemas Solaris. As classes de dependência do DTrace são descritas na lista a seguir ordenadas da menos comum (ou seja, mais específica a uma arquitetura em particular) para a mais comum (ou seja, comum a todas as arquiteturas).

Desconhecida	A interface possui um conjunto desconhecido de dependências de arquitetura. O DTrace não sabe, necessariamente, as dependências de

	arquitetura de todas as entidades, como os tipos de dados definidos na implementação do sistema operacional. O rótulo Desconhecida geralmente é aplicado a interfaces de estabilidade muito baixa para as quais as dependências não podem ser calculadas. A interface pode não estar disponível durante o uso do DTrace em *qualquer* arquitetura diferente da que você está usando atualmente.
CPU	A interface é específica do modelo de CPU do sistema atual. Você pode usar a opção `-v` do utilitário `psrinfo(1M)` para exibir o modelo de CPU atual e os nomes de implementação. As interfaces com dependências em modelo de CPU podem não estar disponíveis em outras implementações de CPU, mesmo que essas CPUs exportem a mesma ISA (Arquitetura de conjunto de instruções). Por exemplo, uma interface dependente de CPU em um microprocessador UltraSPARC-III+ pode não estar disponível em um microprocessador UltraSPARC-II, mesmo que ambos os processadores ofereçam suporte ao conjunto de instruções SPARC.
Plataforma	A interface é específica da plataforma de hardware do sistema atual. Uma plataforma geralmente associa um conjunto de componentes do sistema e características de arquitetura como um conjunto de modelos de CPU com suporte a um nome de sistema como `SUNW,Ultra-Enterprise-10000`. Você pode exibir o nome da plataforma atual usando a opção `uname(1) -i`. A interface pode não estar disponível em outras plataformas de hardware.
Grupo	A interface é específica do grupo de plataformas de hardware do sistema atual. Um grupo de plataformas geralmente associa um conjunto de plataformas com características relacionadas sob um único nome, como `sun4u`. Você pode exibir o nome do grupo de plataformas atual usando a opção `uname(1) -m`. A interface está disponível em outras plataformas no grupo de plataformas, mas pode não estar disponível em plataformas de hardware que não sejam membros do grupo.
ISA	A interface é específica da arquitetura de conjunto de instruções (ISA) com suporte nos microprocessadores deste sistema. A ISA descreve uma especificação por software que pode ser executado no microprocessador, incluindo detalhes como instruções de linguagem de composição e registros. Você pode exibir os conjuntos de instrução nativos com suporte no sistema usando o utilitário `isainfo(1)`. Pode não haver suporte para a interface em sistemas que não exportem nenhum dos mesmos conjuntos de instrução. Por exemplo, uma interface dependente de ISA em um sistema Solaris SPARC pode não ter suporte em um sistema Solaris x86.
Comum	A interface é comum a todos os sistemas Solaris independentemente do hardware subjacente. Os aplicativos em camadas e os programas do DTrace que dependem somente das interfaces comuns podem ser executados e implantados em outros sistemas Solaris com as mesmas

revisões do Solaris e do DTrace. A maioria das interfaces do DTrace é comum, assim, você pode usá-las onde quer que use o Solaris.

Atributos de interface

O DTrace descreve as interfaces usando um trio de atributos que consiste em dois níveis de estabilidade e uma classe de dependência. Por convenção, os atributos de interface são escritos na seguinte ordem, separados por barras:

estabilidade de nome / *estabilidade de dados* / *classe de dependência*

A *estabilidade de nome* de uma interface descreve o nível de estabilidade associado ao seu nome conforme aparece no programa em D ou na linha de comando do `dtrace(1M)`. Por exemplo, a variável de D `execname` é um nome estável: A Sun garante que esse identificador continuará com suporte nos programas em D de acordo com as regras descritas para as interfaces estáveis acima.

A *estabilidade de dados* de uma interface é diferente da estabilidade associada ao nome da interface. Esse nível de estabilidade descreve o comprometimento da Sun com a manutenção dos formatos de dados usados pela interface e quaisquer semânticas de dados associadas. Por exemplo, a variável de D `pid` é uma interface estável: as IDs de processo são um conceito estável no Solaris, e a Sun garante que a variável `pid` será do tipo `pid_t` com a semântica que está definida como o ID de processo correspondente ao segmento que acionou um determinado teste de acordo com as regras para interfaces estáveis.

A *classe de dependência* de uma interface é diferente da estabilidade de nome e dados e descreve se a interface é especifica da plataforma operacional atual ou do microprocessador.

O DTrace e o compilador de D controlam os atributos de estabilidade de todas as entidades de interface do DTrace, incluindo provedores, descrições de teste, variáveis de D, funções de D, tipos e as próprias instruções de programa, como veremos em breve. Observe que todos os três valores podem variar independentemente. Por exemplo, a variável de D `curthread` possui os atributos Estável/Privada/Comum: o nome da variável é estável e comum a todas as plataformas operacionais Solaris, mas essa variável fornece acesso a um formato de dados privado que é um artefato da implementação no kernel do Solaris. A maioria das variáveis de D são fornecidas com os atributos Estável/Estável/Comum, assim como as variáveis que você definir.

Cálculos e relatórios de estabilidade

O compilador de D realiza cálculos de estabilidade para cada uma das descrições de teste e instruções de ação nos programas em D. Você pode usar a opção `-v` do `dtrace` para exibir um relatório da estabilidade do programa. O exemplo a seguir usa um programa escrito na linha de comando:

```
# 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
```

Você também pode desejar combinar a opção `-v` do `dtrace` com a opção `-e`, que informa ao dtrace para compilar mas não executar o programa em D, para que você possa determinar a estabilidade do programa sem ter que ativar testes e executar o programa. Veja a seguir outro exemplo de relatório de estabilidade:

```
# 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
#
```

Observe que no novo programa, foi feita referência à variável de D `curthread`, que possui um nome estável, mas uma semântica de dados privada (ou seja, se olhar para ela, estará acessando detalhes de implementação privada do kernel), e seu status agora é refletido no relatório de estabilidade do programa. Os atributos de estabilidade no relatório do programa são calculados selecionando-se o nível de estabilidade mínimo e a classe dos valores correspondentes para cada trio de atributos de interface.

Os atributos de estabilidade são calculados para uma descrição de teste pegando-se os atributos de estabilidade mínimos de todos os campos de descrição de teste *especificados* de acordo com os atributos publicados pelo provedor. Os atributos dos provedores disponíveis do DTrace são mostrados no capítulo correspondente a cada provedor. Os provedores do DTrace exportam um trio de atributos de estabilidade para cada um dos quatro campos de descrição de todos os testes publicados para esse provedor. Portanto, o nome de um provedor deve ter uma maior estabilidade que os testes individuais que ele exporta. Por exemplo, a descrição do teste:

```
fbt:::
```

indicando que o DTrace deve rastrear a entrada e o retorno de todas as funções do kernel, possui uma estabilidade maior que a descrição do teste:

```
fbt:foo:bar:entry
```

que nomeia uma função interna específica `bar()` no módulo do kernel `foo`. Para simplificar, a maioria dos provedores usa um único conjunto de atributos para todos os valores individuais de *módulo*:*função* :*nome* que eles publicam. Os provedores também especificam atributos para a matriz `args[]`, pois a estabilidade de quaisquer argumentos de teste varia de acordo com o provedor.

Se o campo do provedor não for especificado em uma descrição de teste, então os atributos de estabilidade Instável/Instável/ Comum são atribuídos à descrição porque ela pode terminar correspondendo a testes de provedores que ainda não existem quando usada em uma versão futura do Solaris. A própria Sun não é capaz de fornecer garantias sobre a estabilidade futura e o comportamento deste programa. Você deve sempre especificar explicitamente o provedor ao escrever suas cláusulas do programa em D. Além disso, quaisquer campos de descrição de teste que contenham caracteres de correspondência de padrão (consulte o Capítulo 4, “Estrutura de programa em D”) ou variáveis de macro como `$1` (consulte o Capítulo 15, “Script”) são tratados como se não fossem especificados porque esses padrões de descrição podem se expandir para corresponder a provedores ou testes lançados pela Sun em versões futuras do DTrace e do SO Solaris.

Os atributos de estabilidade são calculados para a maioria das instruções da linguagem D utilizando a estabilidade mínima e a classe das entidades na instrução. Por exemplo, as seguintes entidades da linguagem D possuem os seguintes atributos:

Entidade	Atributos
Variável incorporada de D `curthread`	Estável/Privada/Comum
Variável definida pelo usuário de D `x`	Estável/Estável/Comum

Se você escrever a seguinte instrução do programa em D:

```
x += curthread->t_pri;
```

os atributos resultantes da instrução serão Estável/Privada/Comum, os atributos mínimos associados aos operandos `curthread` e `x`. A estabilidade de uma expressão é calculada pegando-se os atributos de estabilidade mínima de cada um dos operandos.

Quaisquer variáveis de D que você defina no programa recebem automaticamente os atributos Estável/Estável/Comum. Além disso, a gramática da linguagem D e os operadores de D recebem, implicitamente, os atributos Estável/Estável/Comum. As referências aos símbolos do kernel usando o operador aspa invertida (‘) recebem sempre os atributos Privada/Privada/Desconhecida porque refletem artefatos de implementação. Os tipos definidos no código-fonte do programa em D, especificamente os que são associados ao espaço de nome do tipo C e D, recebem os atributos Estável/Estável/Comum. Os tipos que são definidos na implementação do sistema operacional e fornecidos por outros tipos de espaço de nome recebem os atributos Privada/Privada/Desconhecida. O operador de difusão do tipo D produz uma expressão cujos atributos de estabilidade são o mínimo dos atributos da expressão de entrada e dos atributos do tipo de resultado de difusão.

Se você usar o pré-processador de C para incluir arquivos de cabeçalho do sistema C, esses tipos serão associados ao espaço de nome do tipo C e receberão os atributos Estável/Estável/Comum, pois o compilador de D não tem outra opção senão supor que você está assumindo a responsabilidade por essas declarações. Portanto, é possível que você se engane sobre a estabilidade do programa caso use o pré-processador de C para incluir um arquivo de cabeçalho contendo artefatos de implementação. Você deve sempre consultar a documentação correspondente aos arquivos de cabeçalho que está incluindo para determinar os níveis corretos de estabilidade.

Aplicação de estabilidade

Ao desenvolver uma ferramenta em camada ou script do DTrace, talvez você queira identificar a fonte específica de questões de estabilidade ou garantir que o seu programa possui um conjunto desejado de atributos de estabilidade. Você pode usar a opção `dtrace -x amin=`*atributos* para forçar o compilador de D a produzir um erro quando qualquer cálculo de atributo resultar em um trio de atributos menor que os valores mínimos especificados na linha de comando. O exemplo a seguir demonstra o uso de `-x amin` através de um trecho da fonte do programa em D. Observe que os atributos são especificados usando-se três rótulos delimitados por / na ordem usual.

```
# 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
#
```

◆ ◆ ◆ CAPÍTULO 40

Tradutores

No Capítulo 39, "Estabilidade", aprendemos sobre como o DTrace computa e relata os atributos de estabilidade do programa. Idealmente, gostaríamos de construir nossos programas do DTrace consumindo apenas as interfaces Estável ou Desenvolvendo. Infelizmente, ao depurar um problema de baixo nível ou medir o desempenho do sistema, talvez seja necessário ativar testes que estejam associados a rotinas internas do sistema operacional, tais como as funções do kernel, em vez de testes associados a interfaces mais estáveis, tais como as chamadas do sistema. Os dados disponíveis nos locais de teste profundos na pilha do software geralmente são uma coleção de artefatos de implementação em vez de estruturas de dados mais estáveis, tais como aquelas associadas às interfaces de chamada do sistema do Solaris. Para ajudá-lo a escrever programas em D estáveis, o DTrace fornece um recurso para traduzir os artefatos de implementação em estruturas de dados estáveis acessíveis a partir das declarações do seu programa em D.

Declarações do tradutor

Um *tradutor* é uma coleção de declarações de atribuições em D oferecidas pelo fornecedor de uma interface que pode ser usada para traduzir uma expressão de entrada em um objeto do tipo struct. Para entender a necessidade e o uso de tradutores, consideraremos como um exemplo as rotinas da biblioteca padrão ANSI-C definidas em `stdio.h`. Essas rotinas operam em estruturas de dados chamadas `FILE`, cujos artefatos de implementação são abstraídos dos programadores de C. Uma técnica padrão para criar uma abstração de estrutura de dados é fornecer somente uma declaração de reenvio de uma estrutura de dados em arquivos de cabeçalho públicos, enquanto mantém a definição de struct correspondente em um arquivo de cabeçalho privado separado.

Se você estiver escrevendo um programa em C e deseja saber o descritor do arquivo correspondente a uma struct `FILE`, use a função `fileno(3C)` para obter o descritor em vez de cancelar diretamente a referência a um membro da struct FILE . Os arquivo de cabeçalho do Solaris reforçam essa regra, definindo a struct FILE como uma marca de declaração de reenvio

opaca para que ela não tenha a sua referência cancelada diretamente pelos programas em C que incluem <stdio.h>. Dentro da biblioteca libc.so.1, imagine que fileno() é implementado em C, da seguinte forma:

```
int
fileno(FILE *fp)
{
    struct file_impl *ip = (struct file_impl *)fp;

    return (ip->fd);
}
```

Nossa fileno() hipotética usa um ponteiro FILE como um argumento e o intercala com um ponteiro para uma estrutura libc interna correspondente, struct file_impl, em seguida, retorna o valor do membro fd da estrutura da implementação. Por que o Solaris implementa interfaces dessa forma? Abstraindo os detalhes da implementação libc atual dos programas clientes, a Sun é capaz de manter um compromisso com uma compatibilidade binária forte enquanto continua a desenvolver e alterar os detalhes da implementação interna de libc. Em nosso exemplo, o membro fd poderia mudar de tamanho ou de posição na struct file_impl, mesmo em um patch, e os binários existentes que chamam fileno(3C) não seriam afetados por essa mudança porque eles não dependem desses artefatos.

Infelizmente, um software de observação como o DTrace tem a necessidade de analisar a implementação para fornecer resultados úteis, e não tem o luxo de chamar funções arbitrárias de C definidas nas bibliotecas ou no kernel do Solaris. Você poderia declarar uma cópia de struct file_impl em seu programa em D para instrumentar as rotinas declaradas em stdio.h, mas, então, o seu programa em D confiaria em artefatos de implementação Privada da biblioteca que talvez quebrem em uma versão futura micro ou secundária, ou mesmo em um patch. Idealmente, desejamos fornecer uma construção a ser usada em programas em D que seja vinculada à implementação da biblioteca e seja atualizada de acordo, mas que ainda forneça uma camada adicional de abstração associada a uma estabilidade maior.

Um novo tradutor é criado através da uma declaração no formato:

```
translator output-type < input-type input-identifier > {
    member-name = expression ;
    member-name = expression ;
    ...
};
```

O *tipo de saída* nomeia uma struct que será o tipo de resultado da tradução. O *tipo de entrada* especifica o tipo da expressão de entrada, e é colocado entre colchetes angulares < > e seguido por um *identificador de entrada* que pode ser usado nas expressões do tradutor como um alias da expressão de entrada. O corpo do tradutor é colocado entre chaves { } e terminado com um ponto-e-vírgula (;), e consiste em uma lista de *nomes de membro* e identificadores correspondentes às expressões de tradução. Cada declaração de membro deve nomear um

membro exclusivo do *tipo de saída* e deve ser ter atribuída uma expressão de um tipo compatível ao tipo do membro, de acordo com as regras do operador (=) da atribuição de D.

Por exemplo, poderíamos definir uma struct de informações estáveis sobre os arquivos `stdio` com base em algumas interfaces `libc` disponíveis:

```
struct file_info {
    int file_fd;   /* file descriptor from fileno(3C) */
    int file_eof;  /* eof flag from feof(3C) */
};
```

Um tradutor de D hipotético de `FILE` para `file_info` poderia ser declarado em D da seguinte forma:

```
translator struct file_info < FILE *F > {
    file_fd = ((struct file_impl *)F)->fd;
    file_eof = ((struct file_impl *)F)->eof;
};
```

Em nosso tradutor hipotético, a expressão de entrada é do tipo `FILE *` e tem atribuído o *identificador de entrada* `F`. O identificador `F` pode ser usado nas expressões do membro do tradutor como uma variável do tipo `FILE *` que só esteja visível no corpo da declaração do tradutor. Para determinar o valor do membro `file_fd` da saída, o tradutor realiza uma transmissão e cancela a referência semelhante à implementação hipotética de `fileno(3C)`, mostrada acima. Uma tradução semelhante é realizada para obter o valor do indicador EOF.

A Sun fornece um conjunto de tradutores a serem usados com as interfaces do Solaris que você chama a partir dos seus programas em D, e promete manter esses tradutores de acordo com as regras de estabilidade da interface definidas anteriormente quando a implementação da interface correspondente for alterada. Aprenderemos sobre esses tradutores mais tarde neste capítulo, depois que aprendermos como chamar os tradutores a partir de D. O recurso do tradutor em si também é fornecido para ser usado pelos desenvolvedores do aplicativo e da biblioteca que desejam oferecer seus próprios tradutores para que os programadores de D possam usar a fim de observar o estado de seus pacotes de software.

Operador Translate

O operador `xlate` de D é usado para realizar uma tradução de uma expressão de entrada para uma das estruturas de saída de tradução definidas. O operador `xlate` é usado em uma expressão no formato:

`xlate <` *output-type* `> (` *input-expression* `)`

Por exemplo, para chamar o tradutor hipotético das structs FILE definidas acima e acessar o membro `file_fd`, você escreveria a expressão:

```
xlate <struct file_info *>(f)->file_fd;
```

onde `f` é uma variável de D do tipo `FILE *`. À própria expressão `xlate` é atribuído o tipo definido pelo *tipo de saída*. Ao ser definido, um tradutor pode ser usado para traduzir expressões de entrada para o tipo de struct da saída do tradutor, ou para um ponteiro para essa struct.

Se você traduzir uma expressão de entrada para uma struct, poderá cancelar a referência de um membro específico da saída imediatamente usando operador " . ", ou pode atribuir a struct traduzida inteira para outra variável de D a fim de fazer uma cópia dos valores de todos os membros. Se você cancelar a referência de um único membro, o compilador de D irá gerar somente o código correspondente à expressão desse membro. Você não pode aplicar o operador `&` a uma struct traduzida para obter seu endereço, já que o próprio objeto de dados não existe até que seja copiado ou que um dos seus membros seja referenciado.

Se você traduzir uma expressão de entrada em um ponteiro para uma struct, poderá cancelar a referência de um membro específico da saída imediatamente usando operador `->`, ou poderá cancelar a referência do ponteiro usando o operador unário `*`, nesse caso, o resultado se comportará como se você traduzisse a expressão em uma struct. Se você cancelar a referência de um único membro, o compilador de D irá gerar somente o código correspondente à expressão desse membro. Você não pode atribuir um ponteiro traduzido para outra variável de D, já que o próprio objeto de dados não existe até que seja copiado ou um dos seus membros seja referenciado e, portanto, não pode ser endereçado.

Uma declaração do tradutor não pode omitir expressões de um ou mais membros do tipo de saída. Se uma expressão `xlate` for usada para acessar um membro para o qual não haja expressão de tradução definida, o compilador de D produzirá uma mensagem de erro apropriada e anulará a compilação do programa. Se o tipo de saída inteiro for copiado através de uma atribuição de estrutura, quaisquer membros para os quais não há expressões de tradução definidas serão preenchidos com zeros.

Para localizar um tradutor correspondente em uma operação `xlate`, o compilador de D examina o conjunto de tradutores disponíveis na seguinte ordem:

- Primeiro, o compilador procura uma tradução do tipo de expressão de entrada exato para o tipo de saída exato.
- Segundo, o compilador *resolve* os tipos de entrada e de saída, seguindo os alias typedef para os nomes de tipo subjacentes e, em seguida, procura uma tradução do tipo de entrada resolvido para o tipo de saída resolvido.
- Terceiro, o compilador procura uma tradução de um tipo de entrada compatível para o tipo de saída resolvido. O compilador usa as mesmas regras utilizadas para determinar a compatibilidade dos argumentos de chamada de função com os protótipos de função a fim de determinar se um tipo de expressão de entrada é compatível com um tipo de entrada do tradutor.

Se nenhum tradutor correspondente for encontrado de acordo com essas regras, o compilador de D produzirá uma mensagem de erro apropriada e a compilação do programa falhará.

Tradutores de modelo do processo

O arquivo de bibliotecas do DTrace `/usr/lib/dtrace/procfs.d`fornece um conjunto de tradutores a serem usados em seus programas em D para traduzir as estruturas de implementação do kernel do sistema operacional em processos e segmentos para as estruturas estáveis `psinfo` e `lwpsinfo` de `proc(4)`. Essas estruturas também são usadas nos arquivos do sistema de arquivos do Solaris `/proc` `/proc/`*pid*`/psinfo` e `/proc/`*pid*`/lwps/`*lwpid*`/lwpsinfo` e são definidas no arquivo de cabeçalho `/usr/include/sys/procfs.h`. Essas estruturas definem informações sobre Estável úteis sobre processos e segmentos, tais como o ID, o LWP ID do processo, os argumentos iniciais e outros dados exibidos pelo comando `ps(1)`. Consulte `proc(4)` para obter uma descrição completa dos membros e semânticas de struct.

TABELA 40–1 Tradutores `procfs.d`

Tipo de entrada	Atributos de tipo de entrada	Tipo de saída	Atributos de tipo de saída
`proc_t *`	Privado/Privado/Comum	`psinfo_t *`	Estável/Estável/Comum
`kthread_t *`	Privado/Privado/Comum	`lwpsinfo_t *`	Estável/Estável/Comum

Traduções de Estável

Embora um tradutor forneça a habilidade de converter informações em uma estrutura de dados estável, ele não necessariamente resolve todos os problemas de estabilidade que podem surgir na tradução de dados. Por exemplo, se a expressão de entrada em uma operação xlate fizer referência a dados de Instável, o programa em D resultante também será Instável porque a estabilidade do programa sempre é calculada como a estabilidade mínima das declarações e expressões acumuladas dos programas em D. Portanto, às vezes é necessário definir uma expressão de entrada estável específica em um tradutor a fim de permitir que sejam construídos programas estáveis. O mecanismo in-line de D pode ser usado para facilitar essas *traduções estáveis*.

A biblioteca `procfs.d` do DTrace fornece as variáveis `curlwpsinfo` e `curpsinfo` descritas anteriormente como traduções estáveis. Por exemplo, a variável `curlwpsinfo` é realmente uma `inline` declarada da seguinte forma:

```
inline lwpsinfo_t *curlwpsinfo = xlate <lwpsinfo_t *> (curthread);
#pragma D attributes Stable/Stable/Common curlwpsinfo
```

A variável `curlwpsinfo` é definida como uma tradução in-line da variável `curthread`, um ponteiro para a estrutura de dados Privada do kernel que representa um segmento, para o tipo Estável `lwpsinfo_t`. O compilador de D processa este arquivo de biblioteca e armazena em cache a declaração `inline`, fazendo com que `curlwpsinfo` apareça como qualquer outra variável de D. A instrução `#pragma` que segue a declaração é usada para redefinir explicitamente os atributos do identificador `curlwpsinfo` como Estável/Estável/Comum, mascarando a

referência a `curthread` na expressão in-line. Esta combinação de recursos de D permite que os programadores de D usem curthread como a origem de uma tradução de uma forma segura, que pode ser atualizada pela Sun, coincidente com as alterações correspondentes na implementação do Solaris.

◆ ◆ ◆ CAPÍTULO 41

Versionamento

No Capítulo 39, "Estabilidade", aprendemos sobre os recursos do DTrace para determinar os atributos de estabilidade dos programas em D criados por você. Ao criar um programa em D com os atributos de estabilidade apropriados, você talvez queira também vincular esse programa a uma *versão* específica da interface de programação de D. A versão da interface de D é um rótulo aplicado a um conjunto específico de tipos, variáveis, funções, constantes e tradutores disponibilizados para você pelo compilador de D. Se você especificar uma vinculação a uma versão específica da interface de programação de D, garanta que pode recompilar o seu programa em versões futuras do DTrace sem encontrar conflitos entre os identificadores do programa que você define e os identificadores definidos nas versões futuras da interface de programação de D. Você deve estabelecer as vinculações de versão de quaisquer programas em D que deseja instalar como scripts persistentes (consulte o Capítulo 15, "Script") ou usar em ferramentas em camadas.

Versões e releases

O compilador de D rotula conjuntos de tipos, variáveis, funções, constantes e tradutores, correspondentes a um release de software específico, usando uma *seqüência de versão*. Uma seqüência de versão é uma seqüência delimitada por ponto de inteiros decimais no formato "*x*" (um release Principal), "*x.y*" (um release Secundário) ou "*x.y.z*" (um release Micro). As versões são comparadas quando se comparam os inteiros da esquerda para a direita. Se os inteiros mais à esquerda não forem iguais, a seqüência com o maior inteiro será a maior versão (e, portanto, a mais recente). Se os inteiros mais à esquerda forem iguais, a comparação prosseguirá para o próximo inteiro da esquerda para a direita, a fim de determinar o resultado. Todos os inteiros não especificados em uma seqüência de versão são interpretados como se tivessem o valor zero durante uma comparação de versão.

As seqüências de versão do DTrace correspondem à nomenclatura padrão da Sun para versões de interface, conforme descrito em `attributes(5)`. Uma alteração na interface de programação

de D é acompanhada por uma nova seqüência de versão. A tabela seguinte resume as seqüências de versão usadas pelo DTrace e o provável significado do release de software do DTrace correspondente.

TABELA 41–1 Versões de release do DTrace

Release	Versão	Significado
Principal	*x*.0	Um release Principal provavelmente contém adições de recursos principais; aderem a revisões diferentes, possivelmente incompatíveis com o Padrão; e embora seja improvável, poderiam alterar, descartar ou substituir interfaces Padrão ou Estável (consulte o Capítulo 39, "Estabilidade"). A versão inicial da interface de programação de D é rotulada como versão 1.0.
Secundários	*x.y*	Comparada a *x*.0 ou versão anterior (onde y não é igual a zero), um novo release Secundário provavelmente contém adições de recursos secundários, compatíveis com as interfaces Padrão e Estável, possivelmente incompatíveis com interfaces Desenvolvendo, ou provavelmente incompatíveis com interfaces Instável. Essas alterações podem incluir novos tipos, variáveis, funções, constantes e tradutores de D incorporados. Além disso, um release Secundário pode remover o suporte a interfaces anteriormente rotuladas como Obsoletas (consulte o Capítulo 39, "Estabilidade").
Micro	*x.y.z*	Os releases micros possuem interface compatível com o release anterior (onde z não é igual a zero), mas provavelmente incluem consertos de erros, melhorias de desempenho e suporte a hardware adicional.

Em geral, cada nova versão da interface de programação de D fornecerá um superconjunto dos recursos oferecidos pela versão anterior, com a exceção de quaisquer interfaces Obsoletas que tenham sido removidas.

Opções de versionamento

Por padrão, quaisquer programas em D que você compila usando o `dtrace -s` ou especifica usando as opções de linha de comando `dtrace -P`, `-m`, `-f`, `-n` ou `-i` são vinculados à versão da interface de programação de D mais recente oferecida pelo compilador de D. Você pode determinar a versão da interface de programação de D atual usando a opção `-V` do `dtrace`:

```
$ dtrace -V
dtrace: Sun D 1.0
$
```

Se você quiser estabelecer uma vinculação a uma versão específica da interface de programação de D, defina a opção de `versão` como uma seqüência de versão apropriada. Semelhante a outras opções do DTrace (consulte o Capítulo 16, "Opções e ajustáveis"), você pode definir a opção de versão na linha de comando usando `dtrace -x`:

```
# dtrace -x version=1.0 -n 'BEGIN{trace("hello");}'
```

ou pode usar a sintaxe `#pragma D option` para definir a opção em seu arquivo-fonte do programa em D:

```
#pragma D option version=1.0

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

Se você usar a sintaxe `#pragma D option` para solicitar uma vinculação de versão, deverá colocar esta diretiva no início do arquivo do programa em D antes de quaisquer outras declarações e cláusulas de teste. Se a versão do argumento de vinculação não for válida ou se referir a uma versão não oferecida pelo compilador de D, uma mensagem de erro apropriada será produzida e a compilação falhará. Portanto, você também pode usar o recurso de vinculação de versão para fazer com que a execução de um script de D em uma versão *mais antiga* do DTrace falhe com uma mensagem de erro óbvia.

Antes de compilar as declarações e as cláusulas do seu programa, o compilador de D carrega o conjunto de tipos, funções, constantes e tradutores de D da versão de interface apropriada nos espaços de nome do compilador. Portanto, quaisquer opções de vinculação de versão que você especifica simplesmente controla o conjunto de identificadores, tipos e tradutores que estão visíveis para o seu programa, além das variáveis, tipos e tradutores que o seu programa define. A vinculação de versão impede que o compilador de D carregue interfaces mais recentes que possam definir identificadores ou tradutores que entrem em conflito com as declarações existentes no código-fonte do seu programa e que, portanto, causariam um erro de compilação. Consulte "Nomes e palavras-chave do identificador" na página 49 para obter dicas sobre como selecionar nomes de identificador que provavelmente não entrarão em conflito com as interfaces oferecida por versões futuras do DTrace.

Versionamento do provedor

Ao contrário das interfaces oferecidas pelo compilador de D, as interfaces oferecidas pelos provedores do DTrace (ou seja, testes e argumentos de teste) não são afetadas ou associadas à interface de programação de D ou às opções de vinculação de versão descritas anteriormente. As interfaces de provedor disponíveis são estabelecidas como parte do carregamento da sua instrumentação compilada no software do DTrace no kernel do sistema operacional e variam dependendo da arquitetura definida em sua instrução, da plataforma operacional, do processador, do software instalado em seu sistema Solaris e dos seus privilégios de segurança atuais. O compilador de D e o tempo de execução do DTrace examinam os testes descritos nas cláusulas do seu programa em D e relatam as mensagens de erro apropriadas quando os testes solicitados por seu programa em D não estão disponíveis. Esses recursos são ortogonais para a versão da interface de programação em D porque os provedores do DTrace não exportam as interfaces que possam entrar em conflito com as definições de seus programas em D; ou seja,

você só pode ativar os testes em D, não pode defini-los, e os nomes de teste são mantidos em um espaço de nome separado dos outros identificadores do programa em D.

Os provedores do DTrace são entregues com uma release específico do Solaris e são descritos na versão correspondente do Guia de rastreio dinâmico do Solaris. O capítulo deste guia correspondente a cada provedor também descreverá quaisquer alterações relevantes ou novos recursos oferecidos por um determinado provedor. Você pode usar a opção `-l` do `dtrace` para explorar o conjunto de provedores e testes disponíveis em seu sistema Solaris. Os provedores rotulam suas interfaces usando os atributos de estabilidade do DTrace, e você pode usar os recursos de relatório de estabilidade do DTrace (consulte o Capítulo 39, “Estabilidade”) para determinar se as interfaces do provedor usadas pelo seu programa em D são passíveis de ser alteradas ou oferecidas em releases futuros do Solaris.

Glossário

ação Um comportamento implementado pelo framework do DTrace que pode ser realizado na hora de acionamento do teste que rastreia dados ou modifica o estado do sistema externo ao DTrace. As ações incluem o rastreio de dados, a interrupção de processos e a captura de rastreamentos de pilha, entre outras.

agregação Um objeto que armazena o resultado de uma *função de agregação* conforme definido formalmente no Capítulo 9, "Agregações", indexado por uma tupla de expressões que podem ser usadas para organizar os resultados

ativação Um grupo de testes ativados e seus predicados e ações associados.

cláusula Uma declaração de um programa em D que consiste em uma lista de especificadores de teste, um predicado opcional e uma lista opcional de declarações de ação colocadas entre chaves `{ }`.

consumidor Um programa que usa o DTrace para ativar a instrumentação e ler o fluxo resultante de dados de rastreamento. O comando `dtrace` é o consumidor canônico do DTrace; o utilitário `lockstat(1M)` é outro consumidor especializado do DTrace.

DTrace Um recurso de rastreio dinâmico que oferece respostas concisas para questões arbitrárias.

predicado A expressão lógica que determina se um conjunto de ações de rastreio devem ou não ser executadas quando um teste é acionado. Cada cláusula do programa em D pode ter um predicado associado a ela, colocado entre barras `/ /`.

provedor Um módulo do kernel implementa um tipo específico de instrumentação em nome do framework do DTrace. O provedor exporta um espaço de nome de testes e uma matriz de estabilidade de seu nome e semânticas de dados, conforme mostrado nos capítulos deste manual.

sub-rotina Um comportamento implementado pelo framework do DTrace que pode ser realizado na hora de acionamento do teste que modifica o estado interno do DTrace, mas não rastreia quaisquer dados. Semelhante às ações, as sub-rotinas são solicitadas através do uso da sintaxe de chamada de função de D.

teste Um local ou atividade no sistema ao qual o DTrace pode vincular dinamicamente a instrumentação, incluindo um predicado e ações. Cada teste é nomeado por uma tupla, indicando seu provedor, módulo, função e nome semântico. Um teste pode ser *ancorado* a um módulo e a uma função em particular, ou poderá ser *desancorado*, se não estiver associado a um local de programa específico (por exemplo, um temporizador `profile`).

tradutor Uma coleção de declarações de atribuição de D que convertem os detalhes da implementação de um subsistema instrumentado específico em um objeto do tipo `struct` que forma uma interface de estabilidade maior do que a expressão de entrada.

Índice

B

C

D

E

F

I

L

M

O

P

Q

R

S

T

U

V

W