Solaris モジューラデバッガ

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650–960–1300

Part Number 806-2728–10
2000 年 3 月

Copyright 2000 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. All rights reserved.

本製品およびそれに関連する文書は著作権法により保護されており、その使用、複製、頒布および逆コンパイルを制限するライセンスのもとにおいて頒布されます。サン・マイクロシステムズ株式会社の書面による事前の許可なく、本製品および関連する文書のいかなる部分も、いかなる方法によっても複製することが禁じられます。

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company, Ltd. が独占的にライセンスしている米国ならびに他の国における登録商標です。フォント技術を含む第三者のソフトウェアは、著作権により保護されており、提供者からライセンスを受けているものです。

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227–14(g)(2)(6/87) and FAR 52.227–19(6/87), or DFAR 252.227–7015(b)(6/95) and DFAR 227.7202–3(a).

本製品に含まれる HG 明朝 L と HG ゴシック B は、株式会社リコーがリョービイマジクス株式会社からライセンス供与されたタイプフェースマスタをもとに作成されたものです。平成明朝体 W3 は、株式会社リコーが財団法人 日本規格協会 文字フォント開発・普及センターからライセンス供与されたタイプフェースマスタをもとに作成されたものです。また、HG 明朝 L と HG ゴシック B の補助漢字部分は、平成明朝体 W3 の補助漢字を使用しています。なお、フォントとして無断複製することは禁止されています。

Sun、Sun Microsystems、docs.sun.com、AnswerBook、AnswerBook2 は、米国およびその他の国における米国 Sun Microsystems, Inc. (以下、米国 Sun Microsystems 社とします) の商標もしくは登録商標です。

サンのロゴマークおよび Solaris は、米国 Sun Microsystems 社の登録商標です。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

OPENLOOK、OpenBoot、JLE は、サン・マイクロシステムズ株式会社の登録商標です。

Wnn は、京都大学、株式会社アステック、オムロン株式会社で共同開発されたソフトウェアです。

Wnn6 は、オムロン株式会社で開発されたソフトウェアです。(Copyright OMRON Co., Ltd. 1999 All Rights Reserved.)

「ATOK」は、株式会社ジャストシステムの登録商標です。

「ATOK8」は株式会社ジャストシステムの著作物であり、「ATOK8」にかかる著作権その他の権利は、すべて株式会社ジャストシステムに帰属します。

「ATOK Server/ATOK12」は、株式会社ジャストシステムの著作物であり、「ATOK Server/ATOK12」にかかる著作権その他の権利は、株式会社ジャストシステムおよび各権利者に帰属します。

本製品に含まれる郵便番号辞書 (7 桁/5 桁) は郵政省が公開したデータを元に制作された物です (一部データの加工を行なっています)。

本製品に含まれるフェイスマーク辞書は、株式会社ビレッジセンターの許諾のもと、同社が発行する『インターネット・パソコン通信フェイスマークガイド '98』に添付のものを使用しています。© 1997 ビレッジセンター

Unicode は、Unicode, Inc. の商標です。

本書で参照されている製品やサービスに関しては、該当する会社または組織に直接お問い合わせください。

OPEN LOOK および Sun Graphical User Interface は、米国 Sun Microsystems 社が自社のユーザおよびライセンス実施権者向けに開発しました。米国 Sun Microsystems 社は、コンピュータ産業用のビジュアルまたはグラフィカル・ユーザインタフェースの概念の研究開発における米国 Xerox 社の先駆者としての成果を認めるものです。米国 Sun Microsystems 社は米国 Xerox 社から Xerox Graphical User Interface の非独占的ライセンスを取得しており、このライセンスは米国 Sun Microsystems 社のライセンス実施権者にも適用されます。

DtComboBox ウィジェットと DtSpinBox ウィジェットのプログラムおよびドキュメントは、Interleaf, Inc. から提供されたものです。(© 1993 Interleaf, Inc.)

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含みそれに限定されない、明示的であるか黙示的であるかを問わない、なんらの保証も行われないものとします。

本製品が、外国為替および外国貿易管理法 (外為法) に定められる戦略物資等 (貨物または役務) に該当する場合、本製品を輸出または日本国外へ持ち出す際には、サン・マイクロシステムズ株式会社の事前の書面による承諾を得ることのほか、外為法および関連法規に基づく輸出手続き、また場合によっては、米国商務省または米国所轄官庁の許可を得ることが必要です。

原典: *Solaris Modular Debugger Guide*

Part No: 806-1583-10

Revision A

Adobe PostScript

目次

はじめに

モジューラデバッガ (MDB) は、Solaris™ オペレーティング環境で使用する新しい汎用デバッグ用ツールです。MDB の主な特長は、その拡張性にあります。『*Solaris* モジューラデバッガ』では、複雑なソフトウェアシステムをデバッグする MDB の使用方法について、特に、Solaris カーネル、関連するデバイスドライバ、モジュールなどをデバッグする場合に使用可能な機能に重点を置いて説明します。さらに、このマニュアルには、MDB 言語構文、デバッガ機能、および MDB モジュールプログラミング API についてのリファレンスと解説も記載されています。

対象読者

もしあなたが刑事で、犯罪の現場を捜査していると仮定した場合は、目撃者に会って、何が起こったか、誰かを見たかと尋ねるでしょう。しかし、目撃者がいない場合や、目撃証言が不十分な場合は、指紋を採取したり、法廷証拠を集めたりしようと考えるでしょう。また、事件の解決を図るためにその証拠を DNA 鑑定することもあるでしょう。ソフトウェアプログラムの障害も、しばしば、これと同様のカテゴリに分類される場合があります。つまり、ソースレベルのデバッグ用ツールで解決できる問題もあれば、誤りを判断して訂正するために、低レベルのデバッギング機能、コアファイルの検査、およびアセンブリ言語の知識が必要な問題もあります。MDB は、このような二次段階の問題分析を支援するよう設計されたデバッグ用ソフトウェアです。

刑事が、顕微鏡や DNA の証拠をすべての事件に対して必要としないのと同様に、MDB が、すべての障害で必要であるとは限りません。しかし、オペレーティングシ

ステムのように、複雑で低レベルなソフトウェアシステムをプログラミングする場合は、MDB が必要になることがしばしばあります。その結果、これらの障害診断を支援するために、MDB は、ユーザーが独自のカスタム診断ツールを構築できるようなデバッギングのフレームワークとして設計されています。また、アセンブリ言語レベルでプログラムの状態を分析できるように、MDB では、強力な組み込みコマンドも用意しています。

アセンブリ言語のプログラミングやデバッギングに慣れていない場合は、11ページの「関連マニュアルと論文」を参照してください。役立つ資料が記載されてます。

また、プログラムをデバッグしている途中で、そのプログラムのソースコードと、それに対応するアセンブリ言語コードとの関係を明らかにするために、プログラム中の対象部分のさまざまな機能を逆アセンブルする必要もあるでしょう。Solaris カーネルソフトウェアをデバッグするために MDB を使用する場合は、第 5 章と第 6 章を熟読してください。これらの章では、Solaris カーネルソフトウェアをデバッグするために必要な MDB コマンドと機能について詳しく説明しています。

内容の紹介

第 1 章では、モジューラデバッガの概要を説明します。この章の対象読者はすべてのユーザーです。

第 2 章では、MDB のアーキテクチャを説明し、このデバッガについて、マニュアル全体で使用されている概念の用語について説明します。この章の対象読者はすべてのユーザーです。

第 3 章では、MDB 言語の構文、演算子、および評価規則について説明します。この章の対象読者はすべてのユーザーです。

第 4 章では、常に使用可能な、組み込みのデバッガコマンドセットについて説明します。この章の対象読者はすべてのユーザーです。

第 5 章では、Solaris カーネルをデバッグする場合に使用する、読み込み可能なデバッガコマンドについて説明します。この章の対象読者は、Solaris カーネルのクラッシュダンプを検査するユーザーや、カーネルソフトウェアの開発者です。

第 6 章では、Solaris カーネルメモリーアロケータのデバッギング機能と、これらの機能を活用するために用意された MDB コマンドについて説明します。この章の対象読者は、上級プログラマとカーネルソフトウェアの開発者です。

第 7 章では、読み込み可能なデバッガモジュールを作成する機能について説明します。この章の対象読者は、上級プログラマと、MDB のカスタムデバッギングを開発するソフトウェア開発者です。

付録 A には、MDB コマンド行のオプションについてのリファレンスが記載されています。

付録 B には、`crash(1M)` コマンドと、それに相当する MDB コマンドのリファレンスが記載されています。

関連マニュアルと論文

以下に、参考となる関連マニュアルと論文を記載します。

- 『*UNIX Internals: The New Frontiers*』、Uresh Vahalia 著、Prentice Hall 発行、1996、ISBN 0-13-101908-2
- 『*The SPARC Architecture Manual, Version 9*』、Prentice Hall 発行、1998、ISBN 0–13–099227–5
- 『*The SPARC Architecture Manual, Version 8*』、Prentice Hall 発行、1994、ISBN 0-13-825001-4
- 『*Pentium Pro Family Developer's Manual, Volumes 1-3*』、Intel Corporation 発行、1996、ISBN 1-55512-259-0 (Volume 1)、ISBN 1-55512-260-4 (Volume 2)、ISBN 1-55512-261-2 (Volume 3)
- 『*The Slab Allocator: An Object-Caching Kernel Memory Allocator*』、Jeff Bonwick 著、Proceedings of the Summer 1994 Usenix Conference 発行、1994、ISBN 9–99–452010–5
- 『*SPARC Assembly Language Reference Manual*』、Sun Microsystems 発行、1998
- 『*x86 Assembly Language Reference Manual*』、Sun Microsystems 発行、1998
- 『*Writing Device Drivers*』、Sun Microsystems 発行、 2000
- 『*STREAMS Programming Guide*』、Sun Microsystems 発行、2000
- 『*Solaris 64 ビット 開発ガイド*』、Sun Microsystems 発行、2000
- 『リンカーとライブラリ』、Sun Microsystems 発行、2000

注 - このマニュアルでは、"IA" という用語は、Intel 32–bit のプロセッサアーキテクチャを意味します。これには、Pentium、Pentium Pro、Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon の各プロセッサ、および AMD と Cyrix が提供する互換マイクロプロセッサチップが含まれます。

注 - Solaris オペレーティング環境は、プラットフォームである—SPARC™ と IA のハードウェア上で動作します。また、64–bit と 32–bit のアドレス空間でも動作します。このマニュアルに記載されている情報は、プラットフォームとアドレス空間のどちらにも適応しています。ただし、特に指定された章、節、項目、注、図表、例、コード例などは除きます。

Sun のマニュアルの注文方法

専門書を扱うインターネットの書店 Fatbrain.com から、米国 Sun Microsystems™, Inc. (以降、Sun™ とします) のマニュアルをご注文いただけます。

マニュアルのリストと注文方法については、`http://www1.fatbrain.com/documentation/sun` の Sun Documentation Center をご覧ください。

Sun のオンラインマニュアル

`http://docs.sun.com` では、Sun が提供しているオンラインマニュアルを参照することができます。マニュアルのタイトルや特定の主題などをキーワードとして、検索をおこなうこともできます。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

ただし AnswerBook2 では、ユーザーが入力する文字と画面上のコンピュータ出力は区別して表示されません。

コード例は次のように表示されます。

- C シェルプロンプト

```
system% command y|n [filename]
```

- Bourne シェルおよび Korn シェルのプロンプト

```
system$ command y|n [filename]
```

- スーパーユーザーのプロンプト

```
system# command y|n [filename]
```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

コマンド例のシェルプロンプト

以下の表に、C シェル、Bourne シェル、および Korn シェル用の、デフォルトのシステムプロンプトとスーパーユーザープロンプトを示します。

表 **P–2** シェルプロンプト

シェル	プロンプト
C シェルプロンプト	machine_name%
C シェルスーパーユーザー プロンプト	machine_name#
Bourne シェルおよび Korn シェルプロンプト	$
Bourne シェルおよび Korn シェルスーパーユーザー プロンプト	#
MDB プロンプト	>

第 1 章

モジューラデバッガの概要

モジューラデバッガ (MDB) は、Solaris で使用する新しい汎用デバッグ用ツールで、主な特長はその拡張性にあります。このマニュアルでは、複雑なソフトウェアシステムをデバッグする MDB の使用方法について、特に、Solaris カーネル、関連するデバイスドライバ、モジュールなどをデバッグする場合に使用可能な機能に重点を置いて説明します。さらに、このマニュアルには、MDB 言語構文、デバッガ機能、および MDB モジュールプログラミング API についてのリファレンスと解説も記載されています。

デバッギング

デバッギングとは、欠陥を取り除くために、ソフトウェアプログラムの実行と状態を分析するプロセスのことです。従来のデバッグ用のツールは、実行制御の機能を備えたもので、それによって、プログラマは制御された環境でプログラムを実行し直したり、プログラムデータの現在の状態を表示したり、プログラム開発に使用するソース言語の表現を評価したりできます。しかし、残念ながら従来の技術では、次のような複雑なソフトウェアシステムをデバッグするには適さない場合がしばしばあります。

- バグが再現されず、プログラムの状態が大規模で分散型になっているオペレーティングシステム
- プログラムが高度に最適化されていたり、デバッグの情報が消去されていたりする
- プログラムそのものが、低レベルのデバッグ用ツールである

- 開発者が顧客からの事後分析情報にしかアクセスできない

MDB は、上記のようなプログラムや状況をデバッグするために、徹底的にカスタマイズできるツールです。MDB に含まれている動的なモジュール機能を使用して、プログラム固有の分析を行う場合に、プログラマ独自のデバッギングコマンドを実行することができます。プログラムの実行時、事後分析時などさまざまな状況で、各 MDB モジュールをプログラム検査に使用することができます。Solaris オペレーティング環境には、MDB モジュールセットが含まれており、プログラマがこれを使用して、Solaris カーネル、関連するデバイスドライバ、カーネルモジュールなどをデバッグできるように設計されています。サードパーティの開発者にも、スーパーバイザーやユーザーソフトウェア用に独自のデバッギングモジュールを開発して配布する場合に、MDB モジュールが役立つと実感していただけるでしょう。

MDB の特長

MDB では、Solaris カーネルやその他のターゲットプログラムを分析する一連の機能を備えているため、次のことが可能になります。

- Solaris カーネルのクラッシュダンプや、ユーザープロセスのコアダンプの事後分析ができます。MDB には、さまざまな機能を備えた一連のデバッガモジュールが含まれています。そのため、標準のデータディスプレイやフォーマット機能に加えて、カーネルやプロセスの状態を詳細に分析することができます。デバッガモジュールを使用して、次のような複雑な照会を行うこともできます。
  - 特定のスレッドによって割り当てられたすべてのメモリーを検出する
  - カーネル STREAM のビジュアル画像を出力する
  - 特定のアドレスが参照している構造タイプを判定する
  - カーネルの中でリークしているメモリーブロックを検出する
  - スタックトレースを検出するためのメモリーを分析する
- デバッガそのものをコンパイルし直したり修正したりすることなく、独自のデバッガコマンドや分析ツールを導入するために有効なプログラミング API が使用できます。MDB では、デバッギング機能が、ロード可能なモジュールセットとして実装されていて、デバッガが `dlopen(3DL)` を実行できる共用ライブラリになっています。各モジュールは、デバッガそのものの機能を拡張するコマンドセットを提供します。同様に、デバッガは、メモリーの読み取りや書き込み、シ

ンボルテーブル情報へのアクセスなど、コアサービスの API を提供します。MDB は、フレームワークを提供しているので、開発者は独自のドライバやモジュール用にデバッギング機能を開発することができます。このため誰もがこれらのモジュールを使用できるようになります。

- `adb(1)` や `crash(1M)` のような旧来のデバッグ用ツールに慣れている場合は、MDB も簡単に使用できます。MDB は、これら既存のデバッギングソリューションに対して下位互換を有しています。MDB 言語そのものは、adb 言語のスーパーセットとして設計されていますが、既存の adb マクロやコマンドは MDB 内でも機能するので、adb 言語を使用している開発者は、MDB 固有のコマンドを知らなくてもすぐに MDB を使用できます。また、MDB では、クラッシュユーティリティで使用される機能よりも強力なコマンドも用意しています。
- 拡張機能を使用できます。MDB は、次のような便利な機能を多数提供しています。
  - コマンド行の編集
  - コマンドの履歴
  - 組み込み型の出力ページャ
  - 構文エラーのチェックと処理
  - オンラインヘルプ
  - 対話型セッションログ

今後の開発

MDB を使用すれば、高度な事後分析ツールの開発を確実に行うことができます。今後は、Solaris オペレーティング環境にさらに MDB モジュールを追加し、カーネルやその他のソフトウェアプログラムのデバッグに、一層高性能な機能を提供していきます。既存のソフトウェアプログラムのデバッグにも、Solaris ドライバとアプリケーションのデバッグ機能を向上させるための独自のモジュール開発にも、MDB を活用してください。

第 2 章

デバッガの概念

この章では、MDB の設計に関する重要な側面とこのアーキテクチャの利点について説明します。

アーキテクチャ

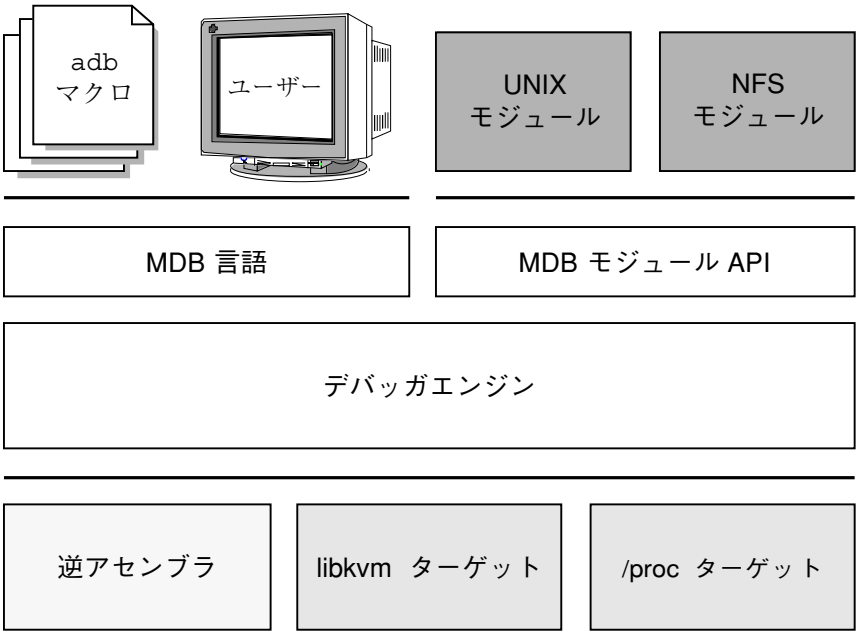

図 2–1　MDB アーキテクチャ

ブロックの構築

ターゲットとは、デバッガによって検査されるプログラムのことです。MDB は、現在次のターゲットをサポートしています。

- ユーザープロセス
- ユーザープロセスのコアファイル
- 実行中のオペレーティングシステム (`/dev/kmem` や `/dev/ksyms` を実行)
- オペレーティングシステムのクラッシュダンプ
- オペレーティングシステムのクラッシュダンプ内に記録されたユーザープロセスイメージ
- ELF オブジェクトファイル

各ターゲットは、プロパティの標準セットをエクスポートします。プロパティには、1 つまたは複数のアドレス空間、1 つまたは複数のシンボルテーブル、ロードオブジェクトセット、およびスレッドセットが含まれます。図 2–1 は、MDB アーキテクチャの概要を示したもので、2 つの組み込みターゲットとサンプルモジュールのペアが入っています。

デバッガコマンド (MDB 用語法では、**dcmd** と表記し、ディーコマンドと読む) は、デバッガルーチンで、現ターゲットのどのプロパティにもアクセスできます。MDB は、標準入力からコマンドを構文解析し、次に対応する dcmd を実行します。各 dcmd は、文字列や数値引数のリストも受け取ることができます (25ページの「構文」を参照)。第 4 章で説明しますが、MDB には、常に使用可能な組み込み dcmd セットが入っています。MDB から提供されるプログラミング API を使用して dcmd を作成することにより、プログラマは MDB そのものの機能を拡張することもできます。

walker は、特定のプログラムデータ構造体の要素を調べたり、繰り返し調べたりする方法を記述するルーチンセットです。walker は、dcmd や MDB そのものからデータ構造体の実装状態をカプセル化します。walker は、対話処理でも使用でき、ほかの dcmd や walker を構築するためのプリミティブとしても使用できます。dcmd の場合と同様に、walker を追加してデバッガモジュールの一部として実装することにより、プログラマは MDB を拡張できます。

デバッガモジュール (**dmod** と表記し、ディーモッドと読む) は、動的に読み込まれたライブラリで、dcmd と walker が含まれています。初期設定の状態では、MDB は、ターゲット内に存在するロードオブジェクトに対応する dmod を読み込もうとします。その後、MDB を実行している間はいつでも、dmod の読み込みや解除ができます。MDB では、Solaris カーネルをデバッグするための標準 dmod セットが提供されています。

マクロファイルは、実行するコマンドセットを含むテキストファイルです。一般的に、マクロファイルは、単純データ構造体の表示プロセスを自動化するときに使用されます。MDB には下位互換性があるので、adb 言語向けに書かれたマクロファイルを実行できます。したがって、Solaris のインストールで提供されるマクロファイルセットは、新旧どちらのツールでも使用可能です。

モジュール性

MDB のモジューラアーキテクチャの利点は、追加デバッガコマンドを含む共用ライブラリを読み込む機能を拡張できることです。MDB アーキテクチャでは、図 2–1 に示すように、各層間でインタフェースの境界を明確に定義しています。マクロファイルは、MDB や adb 言語で書かれたコマンドを実行します。デバッガモジュール内の dcmd や walker は、MDB モジュール API を使って書かれており、アプリケーションバイナリインタフェースの基礎を形成しています。このインタフェースによって、モジュールはデバッガに依存することなく展開できます。

また、walker と dcmd の MDB 名前空間は、デバッギングコード間に二次階層を定義します。このデバッギングコードは、できるかぎりコードを共有し、ターゲットプログラムそのものの展開につれて修正される必要のあるコードの数量を制限します。たとえば、Solaris カーネル内の一次データ構造体の 1 つが、システムのアクティブプロセスを示す `proc_t` 構造体のリストであるとします。この場合、`::ps` dcmd は、その出力を提示するために、このリスト検査を繰り返さなければなりません。しかし、リスト検査を繰り返すコードは `::ps` dcmd 内には存在せず、`genunix` モジュールの `proc` walker 内にカプセル化されています。

MDB では、`::ps` dcmd と `::ptree` dcmd を提供していますが、どちらの dcmd も、`proc_t` 構造体がカーネル内でアクセスされる方法を認識していません。その代わりに、これらの dcmd は、プログラムに従って `proc` walker を呼び出し、次に、戻ってきた構造体を適切にフォーマットします。また、`proc_t` 構造体がすでに変更されている場合には、MDB は新しい `proc` walker を提供するので、従属の dcmd を変更する必要はありません。さらに、`::walk` dcmd を使って、対話処理的に `proc` walker にアクセスすることもできます。そうすれば、デバッギングセッション中に新しいコマンドを作成できます。

階層化とコード共有を行うと共に、MDB モジュール API は、dcmd と walker に、単一で安定したインタフェースを提供します。このインタフェースにより、配下のターゲットのさまざまなプロパティにもアクセスできます。これと同じ API 機能は、ユーザープロセスやカーネルターゲットからの情報にアクセスするときにも使用され、新しいデバッギング機能を開発する作業が簡単になります。

さらに、カスタム MDB モジュールを使用して、さまざまなコンテキストにおいて、デバッギングタスクを実行できます。たとえば、開発中のユーザープログラム用の MDB モジュールを開発したい場合があります。いったんその MDB モジュールを開発すると、そのモジュールを使用して、MDB は独自プログラムの稼動中のプ

ロセスやそのコアダンプ、あるいはプログラムを実行していたシステム上で生じたカーネルクラッシュダンプでさえも検査することができます。

モジュール API には、次のターゲットプロパティにアクセスするための機能が提供されています。

アドレス空間
: モジュール API には、ターゲットの仮想アドレス空間からデータを読み取ったり、書き込んだりする機能が用意されています。また、カーネルデバッギングモジュールには、物理アドレスを使った読み取りや書き込みの機能も提供されます。

シンボルテーブル
: モジュール API には、以下のものの静的シンボルテーブルおよび動的シンボルテーブルへのアクセスが用意されています。ターゲットの一次実行可能ファイル、その実行時リンカー、およびロードオブジェクトセット (ユーザープロセスではライブラリを共有し、Solaris カーネルではロード可能なモジュールとなる)。

外部データ
: モジュール API には、ターゲットに関連した指定外部データバッファの固まりを取り出す機能が用意されています。たとえば、MDB を使用すると、ユーザープロセスやユーザーコアファイルターゲットに関連した `proc(4)` 構造体へのアクセスが、プログラムにより可能になります。

さらに、組み込み MDB の dcmd を使用して、ターゲットメモリーマッピングに関する情報にアクセスしたり、オブジェクトを読み込んだり、値を記録したり、ユーザープロセスターゲットの実行を制御したりできます。

第 3 章

言語構文

この章では、MDB 言語構文、演算子、コマンドの規則、およびシンボルの名前解決について説明します。

構文

デバッガは、標準入力からコマンドを処理します。端末からの標準入力の場合、MDB では端末編集機能が使用できます。また、MDB は、マクロファイルからのコマンドや dcmd パイプラインからのコマンドも処理できます。これについては後述します。言語構文は、ターゲット内のメモリーアドレスに代表されるような、式の値を計算し、dcmd をそのアドレスに適用するという構想に基づいて設計されています。現在のアドレスの位置はドットと呼ばれ、“ . “ は該当する値の参照に使用されます。

メタキャラクタには、次のような文字があります。

`[ ]| ! / \ ? = > $ : ;` 復帰改行文字、空白文字、タブ

空白とは、タブや空白文字のことです。ワード *(word)* とは、1 つまたは複数の引用符なしのメタキャラクタで区切られた文字列のことです。ただし、コンテキストによっては単なる区切り記号として機能するメタキャラクタもあるので、これについては後述します。識別子とは、文字列、数字、下線、ピリオド、または冒頭に文字、下線、ピリオドのどれかを持つ逆引用符のことです。識別子は、シンボル名、変数、dcmd、および walker として使用されます。コマンドは、復帰改行文字やセミコロン (;) で区切ります。

dcmd は、次のようなワードまたはメタキャラクタで表されます。

```
/ \ ? = > $character :character ::identifier
```

メタキャラクタで指定された dcmd や接頭辞 `$` か `:` を 1 つ持つ dcmd は、組み込み演算子として提供されます。また、これらの dcmd は、従来の `adb(1)` ユーティリティのコマンドセットとの互換性を備えています。dcmd が構文解析されると、`/`、`\`、`?`、`=`、`>`、`$`、および `:` は、引数リストが終了するまでメタキャラクタとして認識されなくなります。

単純コマンドとは、後に一連の文字列やワードが続く dcmd のことです。このワードは空白文字で区切られている場合もあります。これらのワードは、呼び出される dcmd に引数として渡されます。ただし、28ページの「演算機能の拡張」と 30ページの「引用」で特に指定されているワードは例外です。各 dcmd は、処理の成功、失敗、または無効な引数を受け取ったことを示す終了ステータスを返します。

パイプライン *(pipeline)* とは、`|` で区切られた 1 つまたは複数の単純コマンドのことです。シェルの場合とは異なり、MDB パイプライン内の dcmd は分割プロセスとしては実行されません。MDB では、パイプラインが構文解析された後に、それぞれの dcmd が左から右へと順に呼び出されます。各 dcmd の出力は、処理された後に格納されます (34ページの「dcmd パイプライン」を参照)。左側の dcmd 処理が終了すると、その出力はパイプライン内の次の dcmd への入力として使用されます。どの dcmd も終了ステータスとして正常終了を返さない場合、そのパイプラインは強制終了します。

式 *(expression)* は一連のワードで表され、64 ビットの符号なし整数を計算するために評価されます。ワードは、28ページの「演算機能の拡張」に示す規則を用いて評価されます。

コマンド

コマンドは、次のうちのどれかです。

pipeline [! word ...] [;]

パイプライン (pipeline) は単純コマンドですが、接尾辞として感嘆符 (`!`) 文字を付けることもできます。この場合、デバッガは、`pipe(2)` を開いた後、MDB パイプライン内の最後の dcmd の標準出力を、`$SHELL -c` の実行により形成された外部プロセスへ送ります。c オプションの後には、感嘆符 (`!`) で始まる文字列が続きます。詳細については、30ページの「シェルエスケープ」を参照してください。

expression pipeline [! word ...] [;]

パイプラインは単純コマンドですが、先頭に式 (expression) を付けることもできます。この場合、パイプラインの実行前に、ドット値 (“ . “ で表される変数) が式の値に設定されます。

expression , expression pipeline [! word ...] [;]

パイプラインは単純コマンドですが、先頭に式を 2 つ付けることもできます。最初の式は新しいドット値を判定するために評価され、2 番目の式はパイプライン内の最初の dcmd の繰り返し回数を判定するために評価されます。この場合、dcmd は、判定された回数繰り返し実行し、その後にパイプライン内の次の dcmd を実行します。繰り返し回数は、パイプライン内の最初の dcmd にだけ適用されます。

, expression pipeline [! word ...] [;]

式の値に応じて、パイプライン内の最初の dcmd が繰り返されます。ただし、先頭の式が省略されている場合は、ドットは変更されません。

expression [! word ...] [;]

コマンドは、算術式だけで構成される場合があります。この場合、式が評価された後、その値にドット変数が設定されます。次に、直前の dcmd と引数が新しいドット値を使用して実行されます。

expression , expression [! word ...] [;]

コマンドは、ドット式と繰り返し回数式だけで構成される場合があります。この場合、最初の式の値がドットに設定された後、2 番目の式で指定された回数、直前の dcmd と引数が繰り返し実行されます。

, expression [! word ...] [;]

直前の dcmd と引数が、繰り返し回数式の値で指定された回数、繰り返し実行されます。ただし、先頭の式が省略されている場合は、ドットは変更されません。

! word ... [;]

コマンドが感嘆符 (!) で始まっている場合は、どの dcmd も実行されず、`$SHELL –c` がデバッガによって実行されます。c オプションの後には、感嘆符 (!) で始まる文字列が続きます。

コメント

// を付けると、その後の復帰改行文字まで、ワードや文字列がすべて無視されます。

演算機能の拡張

MDB コマンドの前に、開始アドレスを表すオプション式や、開始アドレスと繰り返し回数を表すオプション式がある場合は、演算機能が拡張されます。 また、dcmd に使用する数値引数を計算する場合にも、演算機能が拡張されます。演算式は、ドル記号の後に角括弧で囲んだ引数リスト ($[expression])で表され、その式の値に置き換えられます。

式には、次の特殊ワードのどれかを使用できます。

integer	特定の整数値。デフォルトでは、整数値は、接頭辞として 0i または 0I を付けると 2 進数値、0o または 0O を付けると 8 進数値、0t または 0T を付けると 10 進数値、0x または 0X を付けると 16 進数値を表します。
0[tT][0-9]+.[0-9]+	指定された 10 進数の浮動小数点値。IEEE 倍精度浮動小数点表示に変換されます。
'cccccccc'	各文字を ASCII 値に等しいバイトに変換することによって計算された整数値。最大 8 文字までを文字定数に指定できます。文字列は、最下位のバイトから始まって、右から左へと逆方向に整数の中へ格納されます。
<identifier	識別子 (*identifier*) によって指定される変数値
identifier	識別子 (*identifier*) によって指定されるシンボル値
(*expression*)	式 (*expression*) の値
.	ドット値
&	dcmd を実行するために使用される最新のドット値

+	現在のインクリメントによって増分されるドット値
^	現在のインクリメントによって減分されるドット値

インクリメントとは、最後にフォーマットされた dcmd によって読み込まれる合計バイトを格納する大域変数のことです。インクリメントの詳細については、35ページの「dcmd のフォーマット」を参照してください。

単項演算子は右結合で、2 項演算子よりも高い優先度を持っています。以下で、単項演算子について説明します。

#expression	論理否定
~expression	ビット単位の補数
-expression	整数否定
%expression	ターゲットの仮想アドレス空間内の仮想アドレス式に対応するオブジェクトファイル位置でのポインタサイズの数量値
%/[csil]/expression	ターゲットの仮想アドレス空間内の仮想アドレス式に対応するオブジェクトファイル位置での char サイズ、short サイズ、int サイズ、または long サイズの数量値
%/[1248]/expression	ターゲットの仮想アドレス空間内の仮想アドレス式に対応するオブジェクトファイル位置での 1 バイト、2 バイト、4 バイト、または 8 バイトの数量値
****expression***	ターゲットの仮想アドレス空間内の仮想アドレス式でのポインタサイズの数量値
****/[csil]/expression***	ターゲットの仮想アドレス空間内の仮想アドレス式での char サイズ、short サイズ、int サイズ、または long サイズの数量値
****/[1248]/expression***	ターゲットの仮想アドレス空間内の仮想アドレス式での 1 バイト、2 バイト、4 バイト、または 8 バイトの数量値

引用

上述のように (第 3 章を参照)、各メタキャラクタは、引用符で囲まないとワードを終了します。MDB を使用して各文字を特別な意味のない文字そのものとして解釈させるには、それらを単一引用符 (') または二重引用符 (") で囲めば、文字列として引用できます。単一引用符を、単一引用符で囲んで表示させることはできません。二重引用符内では、MDB は C プログラミング言語の文字エスケープシーケンスを認識します。

シェルエスケープ

! 文字を使用して、MDB コマンドとユーザーのシェル間のパイプラインを作成できます。$SHELL 環境変数が設定されている場合、MDB は、そのプログラムをシェルエスケープのためにフォーク (fork) したり、実行 (exec) したりします。変数が設定されていない場合は、/bin/sh コマンドが使用されます。シェルは、–c オプション付きで呼び出されます。c オプションの後には、感嘆符 (!) で始まる文字列が続きます。

! 文字は、他のどのメタキャラクタよりも高い優先度を持っています。ただし、セミコロン (;) と復帰改行文字は例外です。シェルエスケープが検出された後、次のセミコロンまたは復帰改行文字までの残りの文字列は、そのままシェルへ渡されます。シェルの出力コマンドを MDB dcmd へパイプすることはできません。シェルエスケープによって実行されたコマンドは、その出力を MDB へは送らずに、直接端末へ送ります。

変数

変数とは、対応する整数値と一連の属性を持つ変数名のことです。変数名は、一連の文字列、数字、下線、ピリオドなどで表されます。変数には、> dcmd や ::typeset dcmd を使用して値を割り当てることができます。また、その属性は、::typeset dcmd を使用して変更できます。各変数の値は、64 ビットの符号なし整数として表されます。変数は、1 つまたは複数の属性を持つことができます。

たとえば、読み取り専用 (ユーザーによって変更されない)、固定表示 (ユーザーによって設定解除されない)、タグ (ユーザー定義のインジケータ) などです。

次の変数の属性は固定表示として定義されています。

0	/、\、?、 = の dcmd を使用して出力された最新の値
9	$< dcmd とともに使用された最新のカウント
b	データセクションの基底仮想アドレス
d	データセクションのバイトサイズ
e	エントリポイントの仮想アドレス
m	ターゲットの一次オブジェクトファイルの初期バイト (マジックナンバー)。オブジェクトファイルがまだ読み出されていない場合はゼロ
t	テキストセクションのバイトサイズ

さらに、MDB カーネルとプロセスターゲットは、代表スレッドのレジスタセットの現在値を指定変数としてエクスポートします。これらの変数の名前は、ターゲットのプラットフォームや命令セットのアーキテクチャによって決まります。

シンボルの名前解決

25ページの「構文」で前述したように、式のコンテキスト内のシンボル識別子は、このシンボルの値を求めるために評価します。一般的に、この値は、ターゲットの仮想アドレス空間内のシンボルと関連付けられる、記憶領域の仮想アドレスを表します。ターゲットは複数のシンボルテーブルをサポートできます。そのうちのいくつかを次に示します。

- 一次実行可能シンボルテーブル
- 一次動的シンボルテーブル
- 実行時リンカーシンボルテーブル
- 多数のロードオブジェクトそれぞれのための、標準的で動的なシンボルテーブル (ユーザープロセスでは共用ライブラリ、Solaris カーネルではカーネルモジュール)

一般的に、ターゲットは、最初に一次実行可能シンボルテーブルを検索し、次にほかの 1 つまたは複数のシンボルテーブルを検索します。ELF シンボルテーブルには、外部シンボル、大域シンボル、静的シンボルなどへのエントリだけが含まれます。自動シンボルは、mdb によって処理されるシンボルテーブルにはありません。

さらに、mdb は、専用のユーザー定義シンボルテーブルを提供し、このシンボルテーブルを他のどのターゲットシンボルテーブルよりも先に検索します。専用シンボルテーブルは、最初は空の状態ですが、::nmadd や ::nmdel dcmd を使用して操作できます。

::nm –P オプションは、専用のシンボルテーブルの内容を表示するために使用されます。専用のシンボルテーブルによって、元のプログラムでは抜け落ちていたプログラム機能やデータのシンボル定義を作成できます。次からは、MDB がシンボル名をアドレスに変換したり、アドレスを最も近くのシンボルへ変換したりするときにはいつでも、これらの定義が使用可能となります。

ターゲットには複数のシンボルテーブルが含まれていて、各シンボルテーブルには複数のオブジェクトファイルからシンボルを入れることができるので、同じ名前で異なるシンボルが存在することもあります。このような場合に、プログラマが希望するシンボル値を得られるように、MDB はシンボル名適用範囲演算子として、逆引用符 “ ` “ を使用します。

ユーザーは、*object*`*name*、 *file*`*name*、*object*`*file*`*name* などのように、シンボル名の解釈に使用する範囲を指定できます。オブジェクトの識別子は、ロードオブジェクトの名前を参照します。ファイル識別子は、ソースファイルのベース名を参照します。ソースファイルは、指定されたオブジェクトのシンボルテーブル内に STT_FILE 型のシンボルを持っています。オブジェクト識別子の解釈は、ターゲットタイプによって決まります。

MDB カーネルターゲットでは、オブジェクトが、読み込まれたカーネルモジュールのベース名を指定すると考えられます。たとえば、シンボル名 specfs`_init は、specfs カーネルモジュール内の _init シンボルの値を求めるために評価します。

mdb プロセスターゲットでは、オブジェクトが、実行可能な名前、または読み込まれた共用ライブラリの名前を指定すると考えられます。この場合、次の形式のどれかが使用されます。

- 完全な一致 (つまり、完全なパス名) : /usr/lib/libc.so.1
- ベース名に完全に一致 : libc.so.1

- ベース名の冒頭から接尾辞の "." まで一致 : `libc.so` または `libc`
- 実行可能な名前の別名として受け入れられるリテラル文字列 `a.out`

シンボルと 16 進整数値で名前が重複した場合、MDB は、最初にあいまいなトークンをシンボルとして評価し、次に整数値として評価しようとします。たとえば、`f` というトークンが、デフォルトの 16 進数では 10 進整数の `15` を表し、同時にターゲットのシンボルテーブル内の `f` という名前の大域変数を表す場合もあります。あいまいな名前を持つシンボルが存在するときには、明示的な `0x` または `0X` の接頭辞を用いることによって、整数値を明確に指定できます。

dcmd と walker の名前解決

前述のように、MDB の各 dmod は、一連の dcmd と walker を提供します。dcmd と walker は、2 つの異なる広域名前空間でトラックされます。また、MDB も、各 dmod に関連付けられた dcmd と walker の名前空間をトラックし続けます。 与えられた dmod 内で、dcmd や walker に同じ名前を付けることはできません。このような名前の重複がある dmod は、読み込みに失敗します。

異なる dmod から提供された dcmd や walker 間での名前の重複は、広域名前空間では許されます。名前の重複がある場合、その特定の読み込まれる名前を持つ dcmd または walker のうち、最初のものが広域名前空間で優先権を与えられます。ほかの定義は、読み取り順にリストに保存されます。

逆引用符 “ ` “ は、ほかの定義を選択するための参照範囲演算子として、dcmd や walker の名前に使用されます。たとえば、dmod `m1` と `m2` が、それぞれ dcmd `d` を提供する場合に、`m1` の方が `m2` よりも先に読み込まれたときには、次のようになります。

`::d`	`m1` の `d` 定義を実行する
`::m1`d`	`m1` の `d` 定義を実行する
`::m2`d`	`m2` の `d` 定義を実行する

現時点で `m1` モジュールが読み込まれていない場合は、広域定義リスト上の次の dcmd である `m2`d` が、広域定義として使用されます。dcmd や walker の現在の定

義は、以下に示すように、::which dcmd を使用して定義できます。広域定義リストは、::which –v を使用して表示できます。

dcmd パイプライン

dcmd は、縦棒演算子 (|) を使ってパイプラインの中へ入れることができます。パイプラインの目的は、一般的に仮想アドレスのような値のリストを、1 つの dcmd や walker から次の dcmd や walker へと渡していくことです。パイプラインステージは、あるデータ構造体タイプのポインタを、それに対応するデータ構造体のポインタへと対応付けるために使用します。その目的は、アドレスリストをソートしたり、あるプロパティを持つ構造体のアドレスを選択したりすることです。

MDB は、パイプライン内の各 dcmd を左から右へと順番に実行します。現在のドット値、またはコマンドの開始時に明示的な式によって指定された値を使って、最も左にある dcmd が実行されます。縦棒演算子 (|) を見つけると、MDB は、その左側までの dcmd 出力、MDB 構文解析部、および空の値リストとの間に、パイプすなわち共用バッファを作成します。

dcmd を実行するにしたがって、その標準出力はパイプの中に配置され、次に構文解析部によって使用され、評価されます。それは、あたかも MDB が標準出力からデータを読み込んでいるように見えます。各行には、終端に復帰改行文字またはセミコロン (;) を持つ算術式が含まれます。その算術式の値は、パイプに関連付けられた値のリストに追加されます。構文エラーが発見されると、そのパイプラインは異常終了します。

縦棒演算子 (|) の左側までの dcmd が完了すると、そのパイプに関連付けられた値のリストは、縦棒演算子 (|) の右側の dcmd を呼び出すために使用されます。リストの各値については、ドットにその値が設定された後、右側の dcmd が実行されます。パイプラインの最も右にある dcmd だけが、その出力を標準出力に表示します。パイプライン内のいずれかの dcmd が標準エラー出力を生じた場合は、それらのメッセージを直接標準エラーに出力するので、パイプラインの一部としては処理されません。

dcmd のフォーマット

/、\、?、= などのメタキャラクタを使用して、特別な出力書式の dcmd を表します。各 dcmd では、1 つまたは複数の書式制御文字を含む引数リスト、繰り返し回数、または引用文字列を使用できます。書式制御文字は、以下の表に示すように、ASCII 文字の一種です。

書式制御文字を使用して、ターゲットからデータを読み取り、フォーマットします。繰り返し回数は、書式制御文字の前に位置する正の整数で、基数は、常に 10 進数として解釈されます。また、繰り返し回数は、先頭にドル記号を付けた角括弧で囲まれた式 (`$[ ]`)として指定される場合もあります。文字列の引数は、二重引用符 (`" "`) で囲みます。フォーマット引数の間には、空白は不要です。

dcmd のフォーマットは、次のとおりです。

/	ドットで指定される仮想アドレスで始まるターゲットの仮想アドレス空間からデータを表示する
\	ドットで指定される物理アドレスで始まるターゲットの物理アドレス空間からデータを表示する
?	ドットで指定される仮想アドレスに対応するオブジェクトファイル位置で始まるターゲットの一次オブジェクトファイルからデータを表示する
=	指定されたデータ書式のそれぞれにおいて、ドット値そのものを表示する。したがって、= dcmd は、基底間の変換と計算を行うときに便利である

また、MDB は、ドットのほかに、インクリメントと呼ばれる広域値も絶えずトラックしています。インクリメントは、ドットと、最後のフォーマット dcmd によって読み込まれるすべてのデータが後に続くアドレスとの距離を表します。

たとえば、フォーマット dcmd を、A というアドレスに等しいドットで実行した結果、4 バイトの整数が出力された場合、この dcmd が終了した後には、ドットはまだ A ですが、インクリメントは `4` に設定されています。28ページの「演算機能の拡張」で説明したように、ここでは、正符号 (+) は、`A + 4` の値を出すための評価をします。その後、正符号は、次に続く dcmd 用のデータオブジェクトのアドレスにドットを設定し直します。

以下の表に示すように、ほとんどの書式制御文字は、データ書式のサイズに対応するバイトの数だけ、インクリメントの値を増分します。書式制御文字表は、`::formats` dcmd を使用して、MDB の内部から表示できます。書式制御文字は、次のとおりです。

+	カウントの数だけドットを増分する (変数サイズ)
-	カウントの数だけドットを減分する (変数サイズ)
B	l6 進数 int (1 バイト)
C	C の文字表記法を使う文字 (1 バイト)
D	10 進数の符号付き int (4 バイト)
E	10 進数の符号なし long long (8 バイト)
F	double (8 バイト)
G	8 進数の符号なし long long (8 バイト))
H	スワップバイトと short (4 バイト)
I	アドレスと分解命令 (変数サイズ)
J	16 進数 long long (8 バイト)
K	16 進数 uintptr_t (4 または 8 バイト)
O	8 進数の符号なし int (4 バイト)
P	シンボル (4 または 8 バイト)
Q	8 進数の符号付き int (4 バイト)
S	C の文字列表記法を使った文字列 (変数サイズ)
U	10 進数の符号なし int (4 バイト)
V	10 進数の符号なし int (1 バイト)
W	デフォルト基数の符号なし int (4 バイト)
X	16 進数 int (4 バイト)
Y	復号化される time32_t (4 バイト)
Z	16 進数 long long (8 バイト)
^	インクリメント * カウントの数だけドットを減分する (変数サイズ)
a	symbol+offset としてのドット
b	8 進数の符号なし int (1 バイト)

c	文字 (1 バイト)
d	10 進数の符号付き short (2 バイト)
e	10 進数の符号付き long long (8 バイト)
f	float (4 バイト)
g	8 進数の符号付き long long (8 バイト)
h	スワップバイト (2 バイト)
i	命令の分解 (変数サイズ)
n	復帰改行
o	8 進数の符号なし short (2 バイト)
p	シンボル (4 または 8 バイト)
q	8 進数の符号付き short (2バイト)
r	余白
s	raw 文字列 (変数サイズ)
t	水平タブ
u	10 進数の符号なし short (2 バイト)
v	10 進数の符号付き int (1 バイト)
w	符号なしのデフォルト基数 short (2 バイト)
x	16 進数 short (2 バイト)
y	復号化される time64_t (8 バイト)

/、\、および ? のフォーマット dcmd を使用して、ターゲットの仮想アドレス空間、物理アドレス空間、またはオブジェクトファイルに書き込みを行うことができます。この場合には、以下の修飾子の 1 つを最初の書式制御文字として指定し、次に、即値またはドル記号の後の角括弧に囲まれた式 (`$[ ]`)で表されるワードのリストを指定します。

書き込み修飾子は、次のとおりです。

v, w	各式の値の最下位 2 バイトを、ドットで指定された位置から始まるターゲットに書き込む
W	各式の値の最下位 4 バイトを、ドットで指定された位置から始まるターゲットに書き込む

Z　　各式の値の 8 バイトすべてを、ドットで指定された位置から始まるターゲットに書き込む

/、\、および ? のフォーマット dcmd を使用して、ターゲットの仮想アドレス空間、物理アドレス空間、およびオブジェクトファイル内の特定の整数値を検索できます。この場合には、以下の修飾子の 1 つを最初の書式制御文字として指定し、次に、値とオプションマスクを指定します。各値とマスクは、即値またはドル記号の後の角括弧に囲まれた式として指定されます。

値だけが指定されている場合、MDB は、適当なサイズの整数値を読み取り、一致する値が含まれるアドレスのところで終了します。また、`V` という値と、`M` というマスクが指定されている場合、MDB は、適当なサイズの整数値を読み取り、`(X & M) == V` が存在する `X` という値が含まれるアドレスのところで終了します。dcmd が終了すると、ドットは、一致した値が含まれるアドレスに更新されます。一致する値が見つからなかった場合、ドットは、最後に読み込まれたアドレスに残されます。

検索修飾子は、次のとおりです。

l　　指定された 2 バイトの値を検索する

L　　指定された 4 バイトの値を検索する

M　　指定された 8 バイトの値を検索する

ユーザーターゲットでも、カーネルターゲットでも、アドレス空間は、一般的に不連続セグメントセットで構成されています。対応するセグメントを持たないアドレスから読み込むことはできません。セグメント内で一致するものが検索されない場合には、検索は強制的に終了します。

第 4 章

組み込みコマンド

MDB は、常に定義されている組み込み dcmd セットを備えています。これらの dcmd のなかには、特定のターゲットだけに適用されるものもあります。dcmd が現在のターゲットに適用できない場合、その dcmd は停止し、「コマンドが現在のターゲットに適用されません」という内容のメッセージが表示されます。

MDB は、多くの場合、従来の `adb(1)` dcmd の名前に対応するニーモニック (`::identifier`) を提供します。たとえば、 `::quit` は、`$q` に相当します。`adb(1)` の使用経験を持つプログラマや、簡略符号あるいは難解なコマンドを認識するプログラマは、`$` や `:` 形式の方を好まれるかもしれません。一方、`mdb` に慣れていないプログラマは、詳細記述 `::` 形式の方を好まれるでしょう。以下に組み込みコマンドを、アルファベット順に説明します。ただし、`$` や `:` 形式に相当する `::identifier` がある場合には、`::identifier` 形式のアルファベット順に示します。

組み込み dcmd

> *variable-name*
>*/modifier/ variable-name*

指定した名前の変数にドット値を割り当てます。変数が読み取り専用の場合には、変更できません。`>` の後に `//` で囲まれた修飾子がある場合、修飾子を割り当ての一部として、ドット値は変更されます。修飾子は、次のとおりです。

`c`　　符号なし char の量 (1 バイト)

`s`　　符号なし short の量 (2 バイト)

`i`　　符号なし int の量 (4 バイト)

`l`　　符号なし long の量 (32 ビットでは 4 バイト、64 ビットでは 8 バイト)

ただし、これらの演算子は、キャストを実行しません。したがって、リトルエンディアンのアーキテクチャでは指定数値の下位バイトから先に読み込まれ、ビッグエンディアンのアーキテクチャでは、上位バイトから先に読み込まれます。これらの修飾子には下位互換性があります。ただし、MDB の `*`/*modifier*/ および `%`/*modifier*/ 構文を使用します。

`$<` ***macro-name***

指定したマクロファイルからコマンドを読み取り、実行します。ファイル名は、絶対パスまたは相対パスとして与えられます。ファイル名に「/」が含まれない場合は単純名です。単純名の場合、MDB は、マクロファイル組み込みパス内でそのファイル名を検索します。現時点で別のマクロファイルが処理されている場合、そのファイルは閉じられ、代わりに新しいファイルが処理されます。

`$<<` ***macro-name***

`$<` と同様に、指定したマクロファイルからコマンドを読み取り、実行します。ただし、現在開いているマクロファイルは閉じません。

`$?`

ターゲットがユーザープロセスまたはコアファイルの場合に、そのプロセス ID と現在のシグナルを出力します。次に、代表スレッドの汎用レジスタセットを出力します。

[***address*** **]** `$C` **[** ***count*** **]**

C スタックのバックトレースを、スタックフレームポインタの情報も含めて出力します。この dcmd の前に明示的な *address* がある場合には、その仮想記憶アドレスから始まるバックトレースを表示します。その他の場合には、代表スレッドのスタックを表示します。オプションのカウント値が引数として与えられている場合には、出力の各スタックフレームに対して、*count* で指定された数の引数だけが表示されます。

注 [64 ビットシステム (SPARC 版)] - スタックトレースを要求する場合には、バイアス式フレームポインタ値、つまり、仮想アドレス + `0x7ff` をアドレスとして使用してください。

[***base*** **]** `$d`

デフォルトの出力基数を受け取るか、設定します。この dcmd の前に明示的な式がある場合には、デフォルトの出力基数は、指定された *base* に設定されます。その他の場合には、現在の基数が 10 進数で出力されます。デフォルトの基数は 16 (16 進数) です。

`$e`

既知の外部すなわちグローバルのオブジェクト型シンボルや関数シンボルのリスト、そのシンボルの値、およびターゲットの仮想アドレス空間内の対応位置に格納される最初の 4 バイト (32 ビット `mdb`) または 8 バイト (64 ビット `mdb`) のリストを出力します。`::nm` dcmd には、シンボルテーブルの表示用にさらに柔軟なオプションが用意されています。

`$P` ***prompt-string***

指定された *prompt-string* にプロンプトを設定します。デフォルトのプロンプトは、「`>`」です。プロンプトのセットには、`::set -P` または `-P` コマンド行オプションも使用できます。

distance `$s`

アドレスからシンボル名へ変換するための、シンボルマッチングディスタンスを受け取るか、設定します。シンボルマッチングディスタンスのモードについては、付録 A の `-s` コマンド行オプションで説明します。シンボルマッチングディスタンスは、`::set -s` オプションを使用して変更することもできます。距離が指定されない場合には、現在の設定が表示されます。

`$v`

指定された変数のリストのうち、ゼロ以外の値を持つ変数リストを出力します。`::vars` dcmd を使用すると、変数の一覧表示に他のオプションを付けることができます。

width `$w`

出力のページ幅を指定された値に設定します。一般的には、mdb がターミナルに幅の照会をして、サイズを変更するので、このコマンドは必要ありません。

`$W`

書き込みのために、ターゲットをもう一度開きます。ちょうど、コマンド行の –w オプションで、mdb が実行される場合と同じです。`::set –w` オプションを用いても、書き込みモードにすることができます。

[***pid*** **]** `::attach` **[** ***core*** **|** ***pid*** **]**
[***pid*** **]** `:A` **[** ***core*** **|** ***pid*** **]**

ユーザープロセスターゲットが動作中の場合には、指定されたプロセス ID またはコアファイルに付いて、デバッグします。コアファイルのパス名は、文字列引数として指定されます。プロセス ID は、この dcmd の前で、文字列引数として、または式の値として指定されます。デフォルトは 16 進数であることを忘れないで下さい。したがって、`pgrep(1)` や `ps(1)` を使用して得た 10 進数のプロセス ID を式として指定する場合には、その先頭に「`0t`」を付けてください。

`::cat` ***filename ...***

ファイルを連結して、表示します。各ファイル名は、相対パスまたは絶対パス名で指定します。ファイルの内容は標準出力に出力されますが、出力ページャは通りません。この dcmd は、| 演算子とともに使用するようなっています。したがって、プログラマは、外部ファイルに格納されたアドレスリストを使用してパイプライン処理を行えます。

address `::context`
address `$p`

指定されたプロセスへのコンテキストスイッチ。コンテキストスイッチの操作は、カーネルターゲットを使用している場合にだけ有効です。プロセスコンテキストは、カーネルの仮想アドレス空間で、その proc 構造体のアドレスを使用して指定されます。特別なコンテキストアドレス「`0`」は、カーネルそのもののコンテキストを表すときに使用されます。カーネルページだけの場合とは対照的に、クラッシュダンプがすべての物理メモリーページを含んでいる場合には、そのクラッシュダンプを検査するときに、MDB が操作できるのはコンテキストスイッチだけです。`dumpadm(1M)` を使用すると、すべてのページをダンプするために、カーネルクラッシュダンプ機能が構成されます。

ユーザーがカーネルターゲットからコンテキストスイッチを要求した場合には、MDB は指定されたユーザープロセスに相当する新しいターゲットを作成します。スイッチが発生した後、新しいターゲットは、自身の dcmd をグローバルレベルに置きます。したがって、このとき、/ dcmd が、ユーザープロセスの仮想アドレス空間からデータをフォーマットして表示したり、`::mappings` dcmd が、ユーザープロセスのアドレス空間でマッピングを表示したりできます。`0::context` を実行すれば、カーネルターゲットは復元されます。

`::dcmds`

使用可能な dcmd を一覧表示し、各 dcmd の簡単な説明を出力します。

[***address*** **]** `::dis` **[** `–fw` **] [** `–n` ***count*** **] [** ***address*** **]**

最後の引数または現在のドット値によって指定されたアドレス、またはそのアドレス周辺から、逆アセンブルします。そのアドレスが、既知の関数の最初の部分に一致した場合には、その関数全体を逆アセンブルします。その他の場合には、指定されたアドレスの前後に命令を示す「ウィンドウ」が表示され、コンテキストが提供されます。デフォルトでは、命令はターゲットの仮想アドレス空間から読み取られます。ただし、`–f` オプションを指定すると、命令はターゲットのオブジェクトファイルから読み取られます。また、アドレスが既知の関数の最初の部分に一致した場合でも、`–w` オプションを指定すると、「ウィンドウ」が強制的に開かれるモードに設定できます。ウィンドウのサイズは、デフォルトでは 10 個の命令が表示されるようになっています。`–n` オプションを使用すれば、命令の数を明確に指定できます。

`::disasms`

使用可能な逆アセンブラのモードを一覧表示します。ターゲットが初期化されている場合には、MDB は適切な逆アセンブラモードを選択しようとします。また、`::dismode` dcmd を使用して、ユーザーは、初期モードを一覧表のどれかに変更できます。

`::dismode` **[** ***mode*** **]**
`$V` **[** ***mode*** **]**

逆アセンブラモードを受け取るか、設定します。引数が指定されていないと、現在の逆アセンブラモードを出力します。*mode* 引数が指定されている場合には、逆アセンブラを指定されたモードに切り替えます。また、`::disasms` dcmd を使用して、逆アセンブラのリストを表示できます。

::dmods [-l] [*module-name*]

読み込まれたデバッガモジュールを一覧表示します。-l オプションが指定されていると、各 dmod に関連付けられた dcmd や walker の一覧がその dmod 名の下に出力されます。特定の dmod の名前を追加の引数として指定すれば、出力はその dmod に限定されます。

::dump

ドットによって指定されたアドレスを含む、16 バイトで割り当てられた仮想記憶領域のメモリーダンプを 16 進数の ASCII 形式で出力します。::dump に繰り返し回数を指定すると、その数値は繰り返しの数としてではなく、バイト数として解釈されます。

::echo [*string* | *value ...*]

空白文字で区切られ、復帰改行文字で終わる引数を標準出力に出力します。$[ ]で囲まれた式は、評価されて、デフォルト値で出力されます。

::eval *command*

指定された文字列をコマンドとして評価し、実行します。コマンドがメタキャラクタや空白を含む場合は、引用符や二重引用符で囲みます。

```
::files
$f
```

既知のソースファイルの一覧、すなわち、種々のターゲットシンボルテーブルの中にある STT_FILE 型のシンボルを出力します。

```
::fpregs
$x, $X, $y, $Y
```

代表的スレッドの浮動小数点レジスタセットを出力します。

::formats

利用可能な出力書式制御文字の一覧を、/、\、?、= などのフォーマット dcmd とともに使用して一覧表示します。フォーマットとその使用法については、35ページの「dcmd のフォーマット」で説明しています。

`::grep` ***command***

指定されたコマンド文字列を評価した後、新しいドット値がゼロ以外の場合には、古いドット値を出力します。*command* に空白やメタキャラクタが含まれる場合は、必ず引用符で囲んでください。パイプライン内で `::grep` dcmd を使用すると、アドレスリストをフィルタリングできます。

`::help` **[** ***dcmd-name*** **]**

引数がない場合には、`::help` dcmd は、`mdb` で使用可能なヘルプ機能の概要を簡潔に出力します。*dcmd-name* が指定されている場合には、MDB は、その dcmd の使用法の概略を出力します。

`::load` ***module-name***

指定された dmod を読み込みます。モジュール名は、絶対パスまたは相対パスとして指定します。*module-name* が単純名、つまり「/」を含んでいない場合には、MDB はモジュールライブラリパス内で検索します。モジュールの名前に重複があった場合には、そのモジュールは読み込まれません。その場合は、まず既存のモジュール名を解除してください。

`::log` **[** `–d` **|** **[** `–e` **]** ***filename*** **]**
`$>` **[** ***filename*** **]**

出力ログを有効にしたり、無効にしたりします。MDB は、相互ログ機能を提供しているので、まだユーザーとの対話処理が行われているときにも、入力コマンドと標準出力の両方が同じファイルに記録できます。`–e` オプションは、指定されたファイルへのログの書き込みを可能にします。ただし、ファイル名が指定されていない場合には、以前のログファイルに再び記録できます。`–d` オプションは、ログを無効にします。また、`$>` dcmd を使用する場合、ファイル名引数が指定されているときには、ログが有効になります。その他の場合、ログは無効になります。指定されたログファイルがすでに存在する場合、MDB は何らかの新しいログ出力をそのファイルに追加します。

`::map` ***command***

文字列引数として指定される *command* を使用して、ドット値を対応する値へ割り当てます。コマンドに空白やメタキャラクタが含まれる場合には、必ず引用符で囲みます。 `::map` dcmd をパイプライン内で使用すると、アドレスのリストを新しいアドレスリストに変換できます。

[*address*] ::mappings [*name*]
[*address*] $m [*name*]

ターゲットの仮想アドレス空間内の各割り当てを、アドレス、サイズ、それぞれの割り当て記述などを含めて一覧表示します。*address* が dcmd の前にある場合、MDB は指定されたアドレスを含む割り当てだけを表示します。また、文字列で *name* 引数が与えられている場合には、MDB はその記述に一致する割り当てだけを表示します。

::nm [−DPdghnopuvx] [*object*]

現在のターゲットに関連付けられたシンボルテーブルを出力します。*object* 名引数が指定されている場合には、そのロードオブジェクト用のシンボルテーブルだけを表示します。また、::nm dcmd は、次のオプションも認識します。

−D　.symtab の代わりに .dynsym (動的シンボルテーブル) を出力する

−P　.symtab の代わりに専用シンボルテーブルを出力する

−d　値とサイズフィールドを 10 進数で出力する

−g　グローバルシンボルだけを出力する

−h　ヘッダー行を抑制する

−n　名前順にシンボルをソートする

−o　値とサイズフィールドを 8 進数で出力する

−p　シンボルを、一連の ::nmadd コマンドとして出力する。このオプションは −P とともに使用して、マクロファイルを作成できる。その後、$< コマンドを用いて、このマクロファイルをデバッガに読み込む

−u　未定義のシンボルだけを出力する

−v　値順にシンボルをソートする

−x　値とサイズフィールドを 16 進数で出力する

value* ::nmadd [−fo] [−e *end*] [−s *size*] *name

指定されたシンボルの名前を、専用シンボルテーブルへ追加します。MDB は、構成可能な専用シンボルテーブルを用意しています。31ページの「シンボルの名前解決」で説明したように、この専用テーブルは、ターゲットのシンボルテーブル内に置くことができます。また、::nmadd dcmd は、次のオプションも認識します。

-e　　シンボルのサイズを *end* の値に設定する

-f　　シンボルのタイプを STT_FUNC に設定する

-o　　シンボルのタイプを STT_OBJECT に設定する

-s　　シンボルのサイズを *size* に設定する

::nmdel ***name***

指定されたシンボルの名前を専用シンボルテーブルから削除します。

::objects

既知のロードオブジェクトの一次割り当て (通常はテキストセクション) に対応するマッピングだけを表示して、そのターゲットの仮想アドレス空間の割り当てを出力します。

::quit
$q

デバッガを終了します。

::regs
$r

代表スレッドの汎用レジスタセットを出力します。

::release
:R

以前に追加されたプロセスまたはコアファイルを解放します。

::set **[** -wF **] [+/-**o ***option*** **] [** - s ***distance*** **] [** -I ***path*** **] [** -L ***path*** **] [** -P ***prompt*** **]**

デバッガの種々のプロパティを取り込むか、設定します。いずれのオプションも指定されていない場合には、デバッガのプロパティの現在の設定が表示されます。::set dcmd は、次のオプションを認識します。

-F　　その次のユーザープロセスで、::attach が適用されているプロセスを強制的に引き継ぐ。ちょうど、mdb が、コマンド行の -F オプションで実行された場合と同じである

–I　マクロファイルを検出するためのデフォルトパスを設定する。パス引数は特殊トークンを使用できる。付録 A の –I コマンド行オプションの説明を参照

–L　デバッガモジュールを検出するためのデフォルトパスを設定する。パス引数は特殊トークンを使用できる。付録 A の –I コマンド行オプションの説明を参照

–o　指定されたデバッガオプションを有効にする。+o 書式が使用されている場合には、そのデバッガオプションを無効にする。オプションの書式文字列については、–o コマンド行オプションとともに、付録 A に記載してある

–P　コマンドプロンプトを、指定されたプロンプト文字列に設定する

–s　シンボルマッチングディスタンスを指定された距離に設定する。詳細は、付録 A の –s コマンド行オプションの説明を参照

–w　書き込みのためにターゲットを再び開く。ちょうど、mdb が、コマンド行の –w オプションで実行された場合と同じである

[***address*** **]** `::stack` **[** ***count*** **]**
[***address*** **]** `$c` **[** ***count*** **]**

C スタックのバックトレースを出力します。この dcmd の前に明示的に *address* がある場合、その仮想記憶アドレスから始まるバックトレースを表示します。その他の場合には、代表スレッドのスタックを表示します。オプションのカウント値が引数として与えられている場合には、出力の各スタックフレームに対して、*count* 引数で指定された数の引数だけが表示されます。

注 [64 ビットシステム (SPARC 版)] - スタックトレースを要求する場合は、バイアス式フレームポインタ値、つまり、仮想アドレス + `0x7ff` をアドレスとして使用してください。

`::status`

現在のターゲットに関連した情報の概要を出力します。

`::typeset` **[+/-**`t`**]** ***variable-name ...***

指定された変数に属性を設定します。1 つまたは複数の名前が指定されている場合は、それらを定義して、ドット値に設定します。–t オプションを指定すると、各変数に関連付けられたユーザー定義のタグが設定されます。 +t オプションを指定す

ると、そのタグを消去します。変数名が何も指定されていない場合には、変数のリストとその値を出力します。

`::unload` ***module-name***

指定された dmod を解除します。`::dmods` dcmd を使用すると、動作中の dmod のリストを出力できます。組み込みモジュールは解除できません。使用中のモジュール、すなわち現在実行中の dcmd を提供しているモジュールは、解除できません。

`::unset` ***variable-name ...***

定義された変数リストから、指定された変数の設定を解除、すなわち削除します。MDB によってエクスポートされている変数の中には、固定表示と指定されていて、ユーザーが削除できないものがあります。

`::vars` [`–npt`]

指定された変数の一覧を表示します。`–n` オプションを指定すると、その出力は、ゼロ以外の変数に限定されます。`–p` オプションを指定すると、デバッガが `$<` dcmd を用いて再処理するために適した形式で、変数が出力されます。このオプションは、変数をマクロファイルに記録するために使用されますが、終了後は、その変数を元に戻します。`–t` オプションを指定すると、タグのついた変数だけを出力します。また、変数にタグを付けるには、`::typeset` dcmd の `–t` オプションを使用します。

`::version`

デバッガのバージョン番号を出力します。

address `::vtop`

可能な場合、指定された仮想アドレスに対する物理アドレスのマッピングを出力します。`::vtop` dcmd の使用は以下の場合に限られます。カーネルターゲットを検査している場合、または`::context` dcmd の実行後に、カーネルクラッシュダンプ内のユーザープロセスを検査している場合。

[***address***] `::walk` ***walker-name*** [***variable-name***]

指定された walker を使用して、データ構造体の要素を調べます。`::walkers` dcmd を使用すると、使用可能な walker を一覧表示できます。walker は、広域データ構造体について動作する場合もあり、開始アドレスを必要としないものがあります。たとえば、カーネル内の proc 構造体のリストを調べる場合などです。

その他の walker は、アドレスが明示的に指定されている固有のデータ構造体上で動作します。たとえば、アドレス空間でポインタを指定して、セグメントのリストを調べる場合です。

対話処理で使用される場合、 `::walk` dcmd は、データ構造体内の各要素のアドレスをデフォルトで出力します。また、この dcmd は、パイプラインにアドレスリストを提供するときにも使用できます。walker 名には、33ページの「dcmd と walker の名前解決」で説明した逆引用符 “ ` “ 有効範囲規則を使用できます。オプションの *variable-name* が指定されている場合には、MDB がパイプラインの次のステージを呼び出すときに walk の各ステップが返す値に、指定変数が割り当てられます。

`::walkers`

使用可能な walker の一覧と、各 walker の簡潔な説明を出力します。

`::whence` **[**–v**]** ***name*** **...**
`::which` **[**–v**]** ***name*** **...**

指定された dcmd と walker をエクスポートする dmod を出力します。指定された dcmd または walker の広域定義を現在提供しているのはどの dmod かを判定するときに、これらの dcmd は使用されます。広域の名前解決の詳細については、33ページの「dcmd と walker の名前解決」を参照してください。–v オプションを指定すると、各 dcmd や walker の代替定義を優先順に出力します。

`::xdata`

現在のターゲットによってエクスポートされた外部データバッファを一覧表示します。外部データバッファは、現在のターゲットに関連付けられた情報を示します。この情報は、標準ターゲット機能ではアクセスできないもので、アドレス空間、シンボルテーブル、レジスタセットなどが含まれています。これらのバッファは、dcmd による使用が可能です。詳細については、125ページの「`mdb_get_xdata()`」を参照してください。

第 5 章

カーネルデバッギングモジュール

この章では、Solaris カーネルをデバッグするために提供されているデバッガモジュール、dcmd、および walker について説明します。各カーネルデバッガモジュールは、対応する Solaris カーネルモジュールの後に指定されます。したがって、MDB によって自動的に読み込まれます。ここで説明する機能は、Solaris カーネルの現在の実装を反映したものであり、将来変更される場合があります。したがって、これらのコマンドの出力に依存するシェルスクリプトを作成することはお勧めできません。一般的に、この章で説明するカーネルデバッギング機能は、対応するカーネルサブシステム実装のコンテキストだけで意味を持ちます。Solaris カーネルの実装の詳細を説明したリファレンスのリストについては、11ページの「関連マニュアルと論文」を参照してください。

注 - このマニュアルは、Solaris 8 オペレーティング環境での実装を説明しています。つまり、これらのモジュール、dcmd、および walker は、現在のカーネルの実装を反映しているので、過去または将来のリリースに対しては関連せず、適切でない、あるいは適用できない場合があります。これらはどんな種類の恒久的な公開インタフェースを定義するものでもありません。モジュール、dcmd、walker、およびそれらの出力形式および引数に関して提供されている情報は、Solaris オペレーティング環境の将来のリリースでは変更される場合があります。

一般的なカーネルデバッギングサポート (genunix)

カーネルメモリーアロケータ

この節では、Solaris カーネルメモリーアロケータによって識別される問題のデバッグ、およびメモリーとメモリー使用率の検査に使用される dcmd と walker について説明します。ここで説明する dcmd と walker については、第 6 章でさらに詳細に説明します。

dcmd

thread `::allocdby`

カーネルスレッドのアドレスを指定して、そのスレッドが割り当てたメモリーを新しく割り当てた順番に一覧表示します。

bufctl `::bufctl` **[**`–a` ***address***] [`–c` ***caller***] [`–e` ***earliest***] [`–l` ***latest***] [`–t` ***thread***]

指定された bufctl *address* に関する *bufctl* 情報の要約を出力します。1 つまたは複数のオプションが指定されている場合は、オプション引数によって定義される条件に一致する bufctl 情報だけが出力されます。このようにして、dcmd を、パイプラインからの入力のフィルタとして使用することができます。`–a` オプションは、buctl の対応するバッファアドレスが指定されたアドレスと等しくなるように指定します。`–c` オプションは、指定された呼び出し元のプログラムカウント値が bufctl の保存されているスタックトレースの中に存在するように指定します。`–e` オプションは、bufctl の時刻表示が、指定された最も早い時刻表示よりも遅い時刻になるように指定します。`–l` オプションは、bufctl の時刻表示が、指定された最も遅い時刻表示よりも早い時刻になるように指定します。`–t` オプションは、bufctl のスレッドポインタが、指定されたスレッドアドレスと等しくなければならないことを指示します。

[***address*** **]** `::findleaks` **[**`–v`**]**

`::findleaks` dcmd は、フルセットの kmem デバッギング機能が有効になっている場合に、カーネルクラッシュダンプ時に効率的にメモリーリークを検出します。`::findleaks` の最初の実行では、ダンプを処理してメモリーリークを探します。この処理には数分かかる場合があります。次に、割り当てスタックトレース別

にリークがまとめられます。findleaks レポートには、識別されたメモリーリークごとに bufctl アドレスと先頭のスタックフレームが表示されます。

–v オプションが指定されている場合には、この dcmd は実行の際により詳細なメッセージを出力します。dcmd の前に明示的にアドレスが指定されている場合には、レポートがフィルタリングされ、割り当てスタックトレースに指定された関数アドレスが含まれているリークだけが表示されます。

thread `::freedby`

カーネルスレッドのアドレスを指定して、そのスレッドが解放したメモリーを新しく解放した順番に一覧表示します。

value `::kgrep`

カーネルアドレス空間の中で、指定されたポインタサイズ値を含んでいるポインタ整列アドレスを検索します。次に、一致する値を含んでいるアドレスのリストを出力します。MDB の組み込み検索演算子とは異なり、`::kgrep` はカーネルアドレス空間のすべてのセグメントを検索し、不連続セグメント境界にまたがって検索します。大きなカーネルでは、`::kgrep` は実行にかなりの時間がかかる場合があります。

`::kmalog [ slab | fail ]`

カーネルメモリーアロケータトランザクションログの中のイベントを表示します。イベントは、新しく発生した時間の順番に、最新のイベントから先に表示されます。`::kmalog` は、イベントごとに、「T-」表示による最新のイベントを基準にした相対時間 (たとえば、T-0.000151879)、bufctl、バッファアドレス、kmem キャッシュ名、およびイベント発生時刻におけるスタックトレースを表示します。引数を指定しないと、`::kmalog` は kmem トランザクションログを表示しますが、このログは `kmem_flags` で `KMF_AUDIT` が設定されている場合にだけ存在します。`::kmalog fail` は、割り当て障害ログを表示します。このログは必ず存在します。これは、割り当て障害に正しく対処できないドライバのデバッグを行う場合に役立ちます。`::kmalog slab` は、スラブ作成ログを表示します。このログは必ず存在します。`::kmalog slab` は、メモリーリークの検索を行う場合に役立ちます。

`::kmastat`

カーネルメモリーアロケータキャッシュおよび仮想記憶領域のリストと該当する統計を表示します。

::kmausers **[**–ef**] [***cache* **...]**

カーネルメモリーアロケータの現在のメモリーの割り当てが中程度あるいは多いユーザーに関する情報を出力します。この出力は、一意的なスタックトレースごとに 1 つのエントリで構成され、そのスタックトレースを使用して作成された合計メモリー量と割り当ての数が示されます。この dcmd を使用するには、kmem_flags で KMF_AUDIT フラグが設定されている必要があります。

1 つまたは複数のキャッシュ名 (たとえば、kmem_alloc_256) が指定されている場合、メモリー使用率の走査はそれらのキャッシュでだけ行われます。デフォルトでは、すべてのキャッシュが含まれます。–e オプションを指定すると、割り当て量の少ないユーザーが含まれます。割り当ての少ないユーザーとは、同じスタックトレースの合計メモリーが 1024 バイト未満または割り当て数 10 未満であるような割り当てのことです。–f オプションを指定すると、個々の割り当てのスタックトレースが出力されます。

[***address*** **]** ::kmem_cache

指定されたアドレスに格納されている kmem_cache 構造体、またはアクティブ kmem_cache 構造体の完全なセットをフォーマットし、表示します。

::kmem_log

kmem トランザクションログの完全なセットを、発生時間の新しい順にソートして表示します。この dcmd は、::kmalog より簡単な表形式で出力します。

[***address*** **]** ::kmem_verify

指定されたアドレスに格納されている kmem_cache 構造体、またはアクティブ kmem_cache 構造体の完全なセットの完全性を検証します。明示的にキャッシュアドレスが指定されている場合、この dcmd はエラーに関するより冗長な情報を表示します。明示的に指定されていない場合は、要約レポートを表示します。::kmem_verify dcmd については、79ページの「カーネルメモリーキャッシュ」で詳しく説明します。

[***address*** **]** ::vmem

指定されたアドレスに格納されている vmem 構造体、またはアクティブ vmem 構造体の完全なセットをフォーマットし、表示します。この構造体は、<sys/vmem_impl.h> で定義されます。

address ::vmem_seg

指定されたアドレスに格納されている vmem_seg 構造体をフォーマットし、表示します。この構造体は、<sys/vmem_impl.h> で定義されます。

address ::whatis [–abv]

指定されたアドレスに関する情報をレポートします。とくに、::whatis は、そのアドレスが kmem によって管理されているバッファへのポインタまたはスレッドスタックのような別のタイプの特殊メモリー領域へのポインタかどうかを判断し、検出結果をレポートします。–a オプションを指定すると、dcmd は照会に最初に一致するものだけでなく、すべての一致をレポートします。–b オプションを指定すると、dcmd はそのアドレスが既知の kmem bufctl によって参照されているかどうかも判断します。–v オプションを指定すると、この dcmd は、種々のカーネルデータ構造体を検索する際に進行状況をレポートします。

walker

allocdby — 開始点として kthread_t 構造体のアドレスを指定して、このカーネルスレッドによって行われたメモリー割り当てに対応する bufctl 構造体のセットに対して繰り返します。

bufctl — 開始点として kmem_cache_t 構造体のアドレスを指定して、このキャッシュに関連して割り当てられた bufctl のセットに対して繰り返します。

freectl — 開始点として kmem_cache_t 構造体のアドレスを指定して、このキャッシュに関連する未使用 bufctl のセットに対して繰り返します。

freedby — 開始点として kthread_t 構造体のアドレスを指定して、このカーネルスレッドによって行われたメモリー割り当て解除に対応する bufctl 構造体のセットに対して繰り返します。

freemem — 開始点として kmem_cache_t 構造体のアドレスを指定して、このキャッシュに関連する未使用バッファのセットに対して繰り返します。

kmem	開始点として kmem_cache_t 構造体のアドレスを指定して、このキャッシュに関連して割り当てられたバッファのセットに対して繰り返します。
kmem_cache	kmem_cache_t 構造体のアクティブセットに対して繰り返します。この構造体は、<sys/kmem_impl.h> で定義されます。
kmem_cpu_cache	開始点として kmem_cache_t 構造体のアドレスを指定して、このキャッシュに関連する CPU ごとの kmem_cpu_cache_t 構造体に対して繰り返します。この構造体は、<sys/kmem_impl.h> で定義されます。
kmem_slab	開始点として kmem_cache_t 構造体のアドレスを指定して、関連する kmem_slab_t 構造体のセットに対して繰り返します。この構造体は、<sys/kmem_impl.h> で定義されます。
kmem_log	kmem アロケータトランザクションログに格納されている bufctl のセットに対して繰り返します。

ファイルシステム

MDBファイルシステムのデバッギングサポートには、vnode ポインタを対応するファイルシステムのパス名に変換する組み込み機能が含まれています。この変換は、Directory Name Lookup Cache (DNLC) を使用して行われます。その理由は、キャッシュはすべてのアクティブ vnode を保持しているわけではないので、一部の vnode はパス名に変換することができず、名前の代わりに「??」が表示されるからです。

dcmd

```
::fsinfo
```

マウントされているファイルシステムのテーブルを表示します。これには、vfs_t アドレス、ops ベクトル、および各ファイルシステムのマウントポイントが含まれます。

`::lminfo`

ロックマネージャによって登録されたアクティブネットワークロックを持つ vnodes のテーブルを表示します。各 vnode に対応するパス名が示されます。

address `::vnode2path` **[**–v**]**

指定された vnode アドレスに対応するパス名を表示します。–v オプションを指定すると、この dcmd はより詳細な表示を出力します。これには、各中間パスコンポーネントの vnode ポインタが含まれます。

walker

buf — アクティブブロック I/O 転送構造体 (`buf_t` 構造体) のセットに対して繰り返します。`buf` 構造体は、`<sys/buf.h>` で定義されます。詳細については、`buf(9S)` のマニュアルページを参照してください。

仮想記憶

この節では、カーネル仮想記憶サブシステムのデバッギングサポートについて説明します。

dcmd

address `::addr2smap` **[***offset***]**

カーネルの segmap アドレス空間セグメント内の指定されたアドレスに対応する `smap` 構造体アドレスを出力します。

as `::as2proc`

`as_t` アドレス *as* に対応するプロセスの `proc_t` アドレスを表示します。

seg `::seg`

指定されたアドレス空間セグメント (seg_t アドレス) をフォーマットし、表示します。

vnode `::vnode2smap` **[*offset*]**

指定された vnode_t アドレスおよびオフセットに対応する smap 構造体アドレスを出力します。

walker

anon

開始点として anon_map 構造体のアドレスを指定して、関連する anon 構造体のセットに対して繰り返します。anon マップ実装は、<vm/anon.h> で定義されます。

seg

開始点として as_t 構造体のアドレスを指定して、指定されたアドレス空間に関連するアドレス空間セグメント (seg 構造体) のセットに対して繰り返します。seg 構造体は、<vm/seg.h> で定義されます。

CPU とディスパッチャ

この節では、CPU 構造体とカーネルディスパッチャの状態を調べる機能について説明します。

dcmd

`::callout`

コールアウトテーブルを表示します。各コールアウトの関数、引数、有効期限が表示されます。

`::class`

スケジューリングクラステーブルを表示します。

[***cpuid*** **]** `::cpuinfo` **[**–v**]**

各 CPU 上で現在実行されているスレッドのテーブルを表示します。dcmd 名の前にオプションの CPU ID 番号を指定すると、指定された CPU に関する情報だけが表示されます。–v オプションを指定すると、`::cpuinfo` は、各 CPU 上で実行されるまで待機している実行可能スレッドとアクティブ割り込みスレッドも表示します。

walker

cpu

カーネル CPU 構造体のセットに対して繰り返します。`cpu_t` 構造体は、`<sys/cpuvar.h>` で定義されます。

デバイスドライバと DDI フレームワーク

この節では、カーネル開発者やサードパーティのデバイスドライバ開発者にとって有用な dcmd と walker について説明します。

dcmd

`::devbindings` ***device-name***

名前を指定されたドライバについてすべてのインスタンスのリストを表示します。出力は、インスタンスごとに 1 つのエントリからなり、先頭が `dev_info` 構造へのポインタ (`$<devinfo` または `::devinfo` で表示可能)、続いてドライバ名、インスタンス番号、そのインスタンスに関連付けられたドライバプロパティおよびシステムプロパティが表示されます。

address `::devinfo` **[** `–q` **]**

devinfo ノードに関連付けられたシステムプロパティおよびドライバプロパティを出力します。–q オプションを指定すると、そのデバイスノードのクイックサマリだけを表示します。

[*address*] ::devnames [–v]

カーネルの devnames テーブルと dn_head ポインタを表示します。このポインタは、ドライバインスタンスリストを指しています。–v フラグを指定すると、devnames テーブルの各エントリに格納されている追加情報が表示されます。

[*devinfo*] ::prtconf [–cpv]

devinfo で指定されたデバイスノードからカーネルデバイスツリーを表示します。*devinfo* を指定しないと、デフォルトでルートからデバイスツリーが表示されます。–c オプションを指定すると、指定されたデバイスノードの子だけが表示されます。–p オプションを指定すると、指定されたデバイスノードの祖先だけが表示されます。–v オプションを指定すると、各ノードに関連付けられたプロパティが表示されます。

[*major-num*] ::major2name [*major-num*]

指定されたメジャー番号に該当するドライバ名を表示します。メジャー番号は、dcmd の前に式の形で、またはコマンド行引数として指定できます。

[*address*] ::modctl2devinfo

指定された modctl アドレスに対応するすべてのデバイスノードを出力します。

::name2major *driver-name*

デバイスドライバ名を指定すると、そのメジャー番号を表示します。

[*address*] ::softstate [*instance-number*]

softstate 状態ポインタ (ddi_soft_state_init(9F) のマニュアルページを参照) とデバイスインスタンス番号を指定すると、そのインスタンスのソフトの状態を表示します。

walker

devinfo

最初に、指定された devinfo の親に対して繰り返し、それらを最下位から世代順に返します。次に、指定された devinfo 自体を返します。その次に、指定された devinfo の子どもに対し

て、世代順に最上位から最下位まで繰り返します。dev_info 構造体は、<sys/ddi_impldefs.h> で定義されます。

devinfo_children 最初に、指定された devinfo を返し、次に、指定された devinfo の子どもに対して、世代順に最上位から最下位まで繰り返します。dev_info 構造体は、<sys/ddi_impldefs.h> で定義されます。

devinfo_parents 指定された devinfo の親に対して、世代順に最上位から最下位まで繰り返し、次に、指定された devinfo を返します。dev_info 構造体は、<sys/ddi_impldefs.h> で定義されます。

devi_next 指定された devinfo の兄弟ウィジェットに対して繰り返します。dev_info 構造体は、<sys/ddi_impldefs.h> で定義されます。

devnames devnames 配列のエントリに対して繰り返します。この構造体は、<sys/autoconf.h> で定義されます。

STREAMS

この節では、カーネル開発者やサードパーティの STREAMS モジュールやドライバの開発者にとって有用な dcmd と walker について説明します。

dcmd

address `::queue` **[**`–v`**] [**`–f` ***flag***] [`–F` ***flag***] [`–m` ***modname***]

指定された queue_t データ構造体をフィルタリングし表示します。オプションを指定しないと、queue_t の種々のプロパティが表示されます。–v オプションを指定すると、待ち行列フラグが詳細に復号化されます。–f、–F、または –m オプションを指定すると、これらのオプションの引数によって定義される条件に一致した場合にだけ、待ち行列が表示されます。したがって、この dcmd をパイプラインからの入力のフィルタとして使用できます。–f オプションは、指定したフラグ (<sys/stream.h> の Q フラグ名の 1 つ) が待ち行列フラグの中に存在しなければ

ならないことを指示します。–F オプションは、指定したフラグが待ち行列フラグの中に存在してはならないことを指示します。–m オプションは、待ち行列に関連付けられたモジュール名が指定された modname に一致しなければならないことを指示します。

address ::q2syncq

queue_t のアドレスを指定して、対応する syncq_t データ構造体のアドレスを出力します。

address ::q2otherq

queue_t のアドレスを指定して、ピアな読み取りまたは書き込み待ち行列構造体のアドレスを出力します。

address ::q2rdq

queue_t のアドレスを指定して、対応する読み取り待ち行列のアドレスを出力します。

address ::q2wrq

queue_t のアドレスを指定して、対応する書き込み待ち行列のアドレスを出力します。

[***address*** **]** ::stream

STREAM ヘッドを表す stdata_t 構造体のアドレスを指定して、カーネル STREAM データ構造体の画像を表示します。読み取りおよび書き込み待ち行列ポインタ、バイト数、各モジュールのフラグが示され、さらに、余白に指定された待ち行列に関する追加情報が示される場合もあります。

address ::syncq **[**–v**]** **[**–f ***flag***] **[**–F ***flag***] **[**–t ***type***] **[**–T ***type***]

指定された syncq_t データ構造体をフィルタリングし表示します。オプションを指定しないと、syncq_t の種々のプロパティが表示されます。–v オプションを指定すると、syncq フラグが詳細に復号化されます。–f、–F、–t、または –T オプションを指定すると、これらのオプションの引数によって定義される条件に一致する場合にだけ、syncq が表示されます。したがって、この dcmd をパイプラインの入力のフィルタとして使用できます。–f オプションは、指定したフラグ (<sys/strsubr.h> の SQ_ フラグ名の 1 つ) が syncq フラグの中に存在しなけれ

ばならないことを指示します。–F オプションは、指定したフラグが syncq フラグの中に存在してはならないことを指示します。–t オプションは、指定したタイプ (<sys/strsubr.h> の SQ_CI または SQ_CO タイプ名の 1 つ) が syncq タイプビットの中に存在しなければならないことを指示します。–T オプションは、指定したタイプが syncq タイプビットの中に存在してはならないことを指示します。

address ::syncq2q

syncq_t のアドレスを指定して、対応する queue_t データ構造体のアドレスを出力します。

walker

qlink queue_t 構造体のアドレスを指定して、q_link ポインタを使用して関連する待ち行列のリストを調べます。この構造体は、<sys/stream.h> で定義されます。

qnext queue_t 構造体のアドレスを指定して、q_next ポインタを使用して関連する待ち行列のリストを調べます。この構造体は、<sys/stream.h> で定義されます。

readq stdata_t 構造体のアドレスを指定して、読み取り側待ち行列構造体のリストを調べます。

writeq stdata_t 構造体のアドレスを指定して、書き込み側待ち行列構造体のリストを調べます。

ファイル、プロセス、およびスレッド

この節では、Solaris カーネルの種々の基本的ファイル、プロセス、およびスレッド構造体をフォーマットし調べるために使用される dcmd と walker について説明します。

dcmd

process `::fd` ***fd-num***

指定されたプロセスに関連付けられたファイル記述子 *fd-num* に対応する `file_t` アドレスを出力します。このプロセスは、`proc_t` 構造体の仮想アドレスによって指定されます。

thread `::findstack` **[** ***command*** **]**

指定されたカーネルスレッドに関連付けられたスタックトレースを出力します。このスレッドは、`kthread_t` 構造体の仮想アドレスによって識別されます。この dcmd は、複数の異なるアルゴリズムを使用して該当するスタックバックトレースを見つけます。オプションのコマンド文字列を指定すると、ドット変数はスタックフレームの最先頭のフレームポインタアドレスにリセットされ、指定されたコマンドはコマンド行に入力された場合と同じように評価されます。デフォルトのコマンド文字列は、`<.$C0` です。すなわち、フレームポインタを含め、引数を付けずにスタックトレースを出力します。

pid `::pid2proc`

指定されたプロセス ID に対応する `proc_t` アドレスを出力します。MDB のデフォルトは 16 進数であることを思い出してください。したがって、`pgrep(1)` または `ps(1)` を使用して取得した 10 進数のプロセス ID には接頭辞 `0t` を付ける必要があります。

process `::pmap`

指定されたプロセスアドレスが指示するプロセスのメモリーマップを出力します。この dcmd は、`pmap(1)` に似た書式を使用して出力を表示します。

[***process*** **]** `::ps` **[**`–flt`**]**

指定されたプロセスまたはすべてのアクティブシステムプロセスに関連する情報の要約を出力します。`ps(1)` に似ています。`–f` オプションを指定すると、完全なコマンド名と初期引数が出力されます。`–l` オプションを指定すると、各プロセスに関連付けられた LWP が出力されます。`–t` オプションを指定すると、各プロセス LWP に関連付けられたカーネルスレッドが出力されます。

`::ptree`

それぞれの親プロセスから派生した子プロセスを含むプロセスツリーを出力します。この dcmd は、`ptree(1)` に似た書式を使用して出力を表示します。

vnode `::whereopen`

`vnode_t` アドレスを指定して、現在ファイルテーブルの中で開かれている、この vnode を持つすべてのプロセスの `proc_t` アドレスを出力します。

walker

file — 開始点として`proc_t` 構造体のアドレスを指定して、指定されたプロセスに関連付けられた開かれているファイル (`file_t` 構造体) のセットに対して繰り返します。`file_t` 構造体は、`<sys/file.h>` で定義されます。

proc — アクティブプロセス (`proc_t`) 構造体に対して繰り返します。この構造体は、`<sys/proc.h>` で定義されます。

thread — カーネルスレッド (`kthread_t`) 構造体のセットに対して繰り返します。グローバル walk が起動されると、すべてのカーネルスレッドがこの walker によって返されます。開始点として `proc_t` アドレスを使用してローカル walk が起動されると、指定されたプロセスに関連付けられたスレッドのセットが返されます。`kthread_t` 構造体は、`<sys/thread.h>` で定義されます。

同期プリミティブ

この節では、特定のカーネル同期プリミティブを調べるために使用される dcmd と walker について説明します。各プリミティブの意味については、『*man pages section 9F: DDI and DKI Kernel Functions*』を参照してください。

dcmd

rwlock `::rwlock`

読み取り・書き込みロック (`rwlock(9F)` のマニュアルページを参照) のアドレスを指定して、現在のロックの状態と待機しているスレッドのリストを表示します。

[***address*** **]** `::wchaninfo` **[–v]**

条件変数 (`condvar(9F)` のマニュアルページを参照) またはセマフォ (`semaphore(9F)` のマニュアルページを参照) のアドレスを指定して、このオブジェクトでの現在の待機数を表示します。明示的にアドレスを指定しないと、待機しているスレッドがあるすべてのオブジェクトが表示されます。–v オプションを指定すると、オブジェクトで現在ブロックされているスレッドのリストが表示されます。

walker

blocked

同期オブジェクト (たとえば、`mutex(9F)` や `rwlock(9F)`) のアドレスを指定して、ブロックされているカーネルスレッドのリストに対して繰り返します。

wchan

条件変数 (`condvar(9F)` のマニュアルページを参照) またはセマフォ (`semaphore(9F)` のマニュアルページを参照) のアドレスを指定して、ブロックされているカーネルスレッドのリストに対して繰り返します。

cyclic

cyclic サブシステムは下位レベルのカーネルサブシステムで、高解像度、他のカーネルサービスに対する各 CPU インターバルタイマー機能、およびプログラミングインタフェースを提供します。

dcmd

`::cycinfo` [–vV]

CPU ごとに cyclic サブシステムの各 CPU の状態を表示します。–v オプションを指定すると、より詳細な表示が示されます。–V オプションを指定すると、–v よりもさらに詳細な表示が示されます。

address `::cyclic`

指定されたアドレスの `cyclic_t` をフォーマットし、表示します。

`::cyccover`

cyclic サブシステムのコードカバレージ情報を表示します。この情報は、DEBUG カーネルだけで使用可能です。

`::cyctrace`

cyclic サブシステムのトレース情報を表示します。この情報は、DEBUG カーネルだけで使用可能です。

walker

cyccpu — 各 CPU の `cyc_cpu_t` 構造体に対して繰り返します。この構造体は、`<sys/cyclic_impl.h>` で定義されます。

cyctrace — cyclic トレースバッファ構造体に対して繰り返します。この情報は、DEBUG カーネルだけで使用可能です。

プロセス間通信のデバッギングサポート (ipc)

`ipc` モジュールは、メッセージ待ち行列、セマフォ、および共用メモリープロセス間通信プリミティブの実装に対してデバッギングサポートを提供します。

dcmd

`::ipcs` [`-l`]

既知のメッセージ待ち行列、セマフォ、および共用メモリーセグメントに対応するシステム全体の IPC 識別子のリストを表示します。`-l` オプションを指定すると、より詳細な情報のリストが示されます。

[***address***] `::msqid_ds` [`-l`]

指定された `msqid_ds` 構造体またはアクティブな `msqid_ds` 構造体 (メッセージ待ち行列識別子) のテーブルを出力します。`-l` オプションを指定すると、情報のより長いリストが表示されます。

[***address***] `::semid_ds` [`-l`]

指定された `semid_ds` 構造体またはアクティブな `semid_ds` 構造体 (セマフォ識別子) のテーブルを出力します。`-l` オプションを指定すると、より詳細な情報のリストが表示されます。

[***address***] `::shmid_ds` [`-l`]

指定された `shmid_ds` 構造体またはアクティブな `shmid_ds` 構造体 (共用メモリーセグメント識別子) のテーブルを出力します。`-l` オプションを指定すると、より詳細な情報のリストが表示されます。

walker

msg — メッセージ待ち行列識別子に対応するアクティブな `msqid_ds` 構造体を調べます。この構造体は、`<sys/msg.h>` で定義されます。

sem — セマフォ識別子に対応するアクティブな `semid_ds` 構造体を調べます。この構造体は、`<sys/sem.h>` で定義されます。

shm — 共用メモリーセグメント識別子に対応するアクティブな `shmid_ds` 構造体を調べます。この構造体は、`<sys/shm.h>` で定義されます。

ループバックファイルシステムのデバッギングサポート (lofs)

lofs モジュールは、lofs(7FS) ファイルシステムのデバッギングサポートを提供します。

dcmd

[***address*** **]** ::lnode

指定された lnode_t、またはカーネルのアクティブな lnode_t 構造体のテーブルを出力します。

address ::lnode2dev

指定された lnode_t アドレスに対応する配下のループバックマウントファイルシステムの dev_t (vfs_dev) を出力します。

address ::lnode2rdev

指定された lnode_t アドレスに対応する配下のループバックマウントファイルシステムの dev_t (li_rdev) を出力します。

walker

lnode カーネルのアクティブな lnode_t 構造体を調べます。この構造体は、<sys/fs/lofs_node.h> で定義されます。

インターネットプロトコルモジュールのデバッギングサポート (ip)

ip モジュールは、ip(7P) ドライバのデバッギングサポートを提供します。

dcmd

[***address*** **]** `::ire` **[**`-q`**]**

指定された `ire_t`、またはカーネルのアクティブな `ire_t` 構造体のテーブルを出力します。`-q` フラグを指定すると、発信元および宛先アドレスではなく、送信および受信待ち行列ポインタが出力されます。

walker

ire カーネルのアクティブな `ire` (Internet Route Entry) 構造体を調べます。この構造体は、`<inet/ip.h>` で定義されます。

カーネル実行時リンカーエディタのデバッギングサポート (krtld)

この節では、カーネルモジュールとドライバの読み込みを担当するカーネル実行時リンカーエディタのデバッギングサポートについて説明します。

dcmd

[***address*** **]** `::modctl`

指定された `modctl`、またはカーネルのアクティブな `modctl` 構造体のテーブルを出力します。

address `::modhdrs`

`modctl` 構造体のアドレスを指定して、モジュールの ELF 実行可能ヘッダーとセクションヘッダーを出力します。

`::modinfo`

アクティブカーネルモジュールに関する情報を出力します。`/usr/sbin/modinfo` コマンドの出力に似ています。

walker

modctl

カーネルのアクティブな `modctl` 構造体のリストを調べます。この構造体は、`<sys/modctl.h>` で定義されます。

IA: プラットフォームのデバッギングサポート (unix)

これらの dcmd と walker は、IA 用です。

dcmd

[***cpuid*** **|** ***address*** **]** `::ttrace` **[–x]**

トラップトレースレコードを新しい順に表示します。トラップトレース機能は、DEBUG カーネルだけで使用可能です。明示的にドット値を指定すると、その正確な値に応じてこれが CPU ID 番号またはトラップトレースレコードアドレスとして解釈されます。CPU ID 番号を指定すると、出力はその CPU のバッファに制限されます。レコードアドレスを指定すると、そのレコードだけがフォーマットされます。–x オプションを指定すると、完全な raw レコードが表示されます。

walker

ttrace

トラップトレースレコードアドレスのリストを新しい順に調べます。トラップトレース機能は、DEBUG カーネルだけで使用可能です。

SPARC: sun4d プラットフォームのデバッギングサポート (unix)

これらの dcmd と walker は、SPARC sun4d プラットフォーム用です。

dcmd

[***cpuid*** **|** ***address*** **]** `::ttrace` **[**–x**]**

トラップトレースレコードを新しい順に表示します。トラップトレース機能は、DEBUG カーネルだけで使用可能です。明示的にドット値を指定すると、その正確な値に応じてこれが CPU ID 番号またはトラップトレースレコードアドレスとして解釈されます。CPU ID 番号を指定すると、出力はその CPU のバッファに制限されます。レコードアドレスを指定すると、そのレコードだけがフォーマットされます。–x オプションを指定すると、完全な raw レコードが表示されます。

walker

ttrace	トラップトレースレコードアドレスのリストを新しい順に調べます。トラップトレース機能は、DEBUG カーネルだけで使用可能です。

SPARC: sun4m プラットフォームのデバッギングサポート (unix)

これらの dcmd と walker は、SPARC sun4m プラットフォーム用です。

dcmd

[***cpuid*** **|** ***address*** **]** `::ttrace` **[**–x**]**

トラップトレースレコードを新しい順に表示します。トラップトレース機能は、DEBUG カーネルだけで使用可能です。明示的にドット値を指定すると、その正確な値に応じてこれが CPU ID 番号またはトラップトレースレコードアドレスとして解釈されます。CPU ID 番号を指定すると、出力はその CPU のバッファに制限されます。レコードアドレスを指定すると、そのレコードだけがフォーマットされます。–x オプションを指定すると、完全な raw レコードが表示されます。

walker

ttrace	トラップトレースレコードアドレスのリストを新しい順に調べます。トラップトレース機能は、DEBUG カーネルだけで使用可能です。

SPARC: sun4u プラットフォームのデバッギングサポート (unix)

これらの dcmd と walker は、SPARC sun4u プラットフォーム用です。

dcmd

[***address*** **]** `::softint`

指定されたアドレスのソフト割り込みベクトル構造体を表示するか、またはすべてのアクティブなソフト割り込みベクトルを表示します。各構造体の保留数、PIL、引数、およびハンドラ関数が表示されます。

`::ttctl`

トラップトレース制御レコードを表示します。トラップトレース機能は、DEBUG カーネルだけで使用可能です。

[***cpuid*** **]** `::ttrace` **[–x]**

トラップトレースレコードを新しい順に表示します。トラップトレース機能は、DEBUG カーネルだけで使用可能です。明示的にドット値を指定すると、CPU ID として解釈され、出力はその CPU のバッファに制限されます。**–x** オプションを指定すると、完全な raw レコードが表示されます。

[***address*** **]** `::xc_mbox`

指定されたアドレスのクロスコールメールボックスを表示するか、または保留中の要求を持っているすべてのクロスコールメールボックスをフォーマットします。

```
::xctrace
```

CPU クロスコールアクティビティに関連するトラップトレースレコードを新しい順にフォーマットし、表示します。トラップトレース機能は、DEBUG カーネルだけで使用可能です。

walker

softint	ソフト割り込みベクトルテーブルエントリに対して繰り返します。
ttrace	トラップトレースレコードアドレスに対して新しい順に繰り返します。トラップトレース機能は、DEBUG カーネルだけで使用可能です。
xc_mbox	CPU ハンドシェイクとクロスコール (x-call) 要求に使用されるメールボックスに対して繰り返します。

第 6 章

カーネルメモリーアロケータを使用するデバッギング

Solaris カーネルメモリー (kmem) アロケータでは、カーネルクラッシュダンプの分析を容易にする強力なデバッギング機能のセットを用意しています。この章では、これらのデバッギング機能について説明し、またこのアロケータ用に特に設計された MDB dcmd と walker について説明します。Bonwick (11ページの「関連マニュアルと論文」を参照) に、このアロケータ自体の原理の概要が説明されています。アロケータデータ構造体の定義については、ヘッダーファイル `<sys/kmem_impl.h>` を参照してください。システム上で kmem デバッギング機能を有効にして問題の分析能力を向上させたり、あるいは開発システム上で kmem デバッギング機能を有効にしてカーネルソフトウェアやデバイスドライバのデバッギングを支援したりすることができます。

注 - このマニュアルは、Solaris 8 での実装を反映しています。つまり、現在のカーネルの実装を反映しているので、過去または将来のリリースに対しては関連せず、適切でない、あるいは適用できない場合があります。これはどんな種類の公開インタフェースを定義するものでもありません。このカーネルメモリーアロケータに関して提供されている情報は、将来の Solaris リリースでは変更される場合があります。

入門ーサンプルクラッシュダンプの作成

この節では、サンプルクラッシュダンプの作成方法およびそれを調べるために MDB を起動する方法について説明します。

kmem_flags の設定

カーネルメモリーアロケータには多くの高度なデバッギング機能が含まれていますが、それらは性能の低下をもたらす可能性があるので、デフォルトでは有効になっていません。このマニュアルの例を実行するには、これらの機能を有効にする必要があります。性能の低下やほかの問題を引き起こす可能性があるため、これらの機能を有効にするのは、テストシステムに対してだけにしておくべきです。

アロケータのデバッギング機能は、調整可能な kmem_flags によって制御されます。この機能を使用する前に、kmem_flagskmem_flags が次のように正しく設定されていることを確認します。

```
# mdb -k
> kmem_flags/X
kmem_flags:
kmem_flags:     f
```

kmem_flags が「f」に設定されていない場合には、次の行を /etc/system に追加して、システムを再起動する必要があります。

```
set kmem_flags=0xf
```

システムを再起動し、kmem_flags が「f」に設定されていることを確認します。システムを稼動状態に戻す前に、この /etc/system の変更を元に戻すことを忘れないようにしてください。

クラッシュダンプの保存

次に、クラッシュダンプが正しく設定されていることを確認します。最初に、dumpadm が、カーネルクラッシュダンプを保存し、savecore が有効であるように設定されていることを確認します。クラッシュダンプパラメタの詳細については、dumpadm(1M) のマニュアルページを参照してください。

```
# dumpadm
        Dump content: kernel pages
         Dump device: /dev/dsk/c0t0d0s1 (swap)
  Savecore directory: /var/crash/testsystem
    Savecore enabled: yes
```

次に、reboot(1M) に「-d」フラグを設定してシステムを再起動します。これによってカーネルが強制的にパニック状態になり、クラッシュダンプが保存されます。

```
# reboot -d
Sep 28 17:51:18 testsystem reboot: rebooted by root

panic[cpu0]/thread=70aacde0: forced crash dump initiated at user request
```

```
401fbb10 genunix:uadmin+55c (1, 1, 0, 6d700000, 5, 0)
  %l0-7: 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  00000000
...
```

システムが再起動されたら、クラッシュダンプが成功したことを確認します。

```
$ cd /var/crash/testsystem
$ ls
bounds    unix.0    unix.1    vmcore.0  vmcore.1
```

ダンプディレクトリにダンプが見当たらない場合には、このパーティションが容量不足である可能性があります。スペースを解放し、root として手動で savecore(1M) を実行し、続いてダンプを保存することができます。ダンプディレクトリに複数のクラッシュダンプが含まれている場合には、今作成したクラッシュダンプは、変更時刻が最新である unix.[n] と vmcore.[n] のペアになります。

MDB の起動

次に、作成したクラッシュダンプに対して mdb を実行し、その状態をチェックします。

```
$ mdb unix.1 vmcore.1
Loading modules: [ unix krtld genunix ip nfs ipc ]
> ::status
debugging crash dump vmcore.1 (32-bit) from testsystem
operating system: 5.8 Generic (sun4u)
panic message: forced crash dump initiated at user request
```

このマニュアルに示す例では、32 ビットカーネルからのクラッシュダンプを使用します。ここに示す手法はすべて 64 ビットカーネルにも適用可能であり、ポインタ (32 ビットシステムと 64 ビットシステムではサイズが異なる) を固定サイズ量 (カーネルデータモデルに関して不変) と区別するよう注意が払われています。

ここに示す例の生成には、Sun Ultra-1 ワークステーションを使用しました。結果は、使用するアーキテクチャとシステムのモデルによって異なる場合があります。

アロケータの基礎

カーネルメモリーアロケータの仕事は、仮想記憶領域を他のカーネルサブシステムに区分けすることです。これらのカーネルサブシステムは、通常、クライアントと

呼ばれます。この節では、アロケータの操作の基礎を説明し、また、このマニュアルで後で使用するいくつかの用語を説明します。

バッファの状態

カーネルメモリーアロケータが作用する領域は、カーネルヒープを構成する仮想記憶のバッファの集まりです。これらのバッファは、キャッシュと呼ばれる一様なサイズと目的を持ったセットにグループ化されます。各キャッシュには、バッファのセットが含まれています。これらのバッファの一部は現在未使用です。つまり、これらはまだアロケータのクライアントに割り当てられていません。残りのバッファは割り当て済みです。つまり、そのバッファへのポインタが、アロケータのクライアントに提供されています。アロケータのクライアントが、割り当てられているバッファへのポインタを保持していない場合には、そのバッファは解放することができないので、リークしていると言われます。リークしているバッファは、カーネル資源を無駄使いしている正しくないコードを示しています。

トランザクション

kmem トランザクションとは、バッファの割り当て済み状態と未使用状態の間の移行のことです。アロケータは、各トランザクションの一部として、バッファの状態が有効であることを確認することができます。さらに、アロケータには、事後分析のためにトランザクションを記録しておく機能があります。

スリーピング割り当てと非スリーピング割り当て

標準 C ライブラリの `malloc(3C)` 関数とは異なり、カーネルメモリーアロケータは、ブロックする (またはスリープする) ことができ、クライアントの要求を満たすのに十分な仮想記憶が使用可能になるまで待機することができます。これは、`kmem_alloc(9F)` の「flag」パラメタによって制御 されます。`KM_SLEEP` フラグが設定されている `kmem_alloc(9F)` への呼び出しは、決して失敗することはありません。この呼び出しは、資源が使用可能になるまでいつまでも待機します。

カーネルメモリーキャッシュ

カーネルメモリーアロケータは、管理しているメモリーをキャッシュのセットに分割します。すべての割り当てはこれらのキャッシュから供給され、これらのキャッ

シュは kmem_cache_t データ構造体によって表されます。各キャッシュは固定のバッファサイズを持っており、これはそのキャッシュが満たす最大割り当てサイズを表します。各キャッシュは、管理するデータのタイプを示す文字列による名前を持っています。

一部のカーネルメモリーキャッシュは特殊な目的用で、特定の種類のデータ構造体だけを割り当てるために初期化されます。この一例を挙げると、「thread_cache」があります。これは、kthread_t タイプの構造体だけを割り当てます。このキャッシュのメモリーは、kmem_cache_alloc() 関数によってクライアントに割り当てられ、kmem_cache_free() 関数によって解放されます。

注 - kmem_cache_alloc() と kmem_cache_free() は公開 DDI インタフェースではありません。これらは Solaris の将来のリリースでは変更または削除される場合があるので、これらに依存したコードを作成しないでください。

---

kmem_alloc_で始まる名前を持つキャッシュは、カーネルの汎用メモリー割り当てスキーマを実装します。これらのキャッシュは、kmem_alloc(9F) と kmem_zalloc(9F) のクライアントにメモリーを提供します。これらの各キャッシュは、そのキャッシュとそれに次ぐ長さのキャッシュ間のバッファサイズの要求を満たします。たとえば、このカーネルは kmem_alloc_8 と kmem_alloc_16 キャッシュを持っています。この場合、kmem_alloc_16 キャッシュは、9 ～ 16 バイトのメモリーを要求するすべてのクライアント要求を処理します。クライアント要求のサイズに関係なく、kmem_alloc_16 キャッシュの各バッファのサイズは 16 バイトです。14 バイト要求の場合、要求は kmem_alloc_16 キャッシュによって満たされるので、バッファの残りの 2 バイトは使用されません。

キャッシュの最後のセットは、カーネルメモリーアロケータ自体が記述するために内部的に使用するキャッシュです。これには、名前が kmem_magazine_ または kmem_va_ で始まるキャッシュ、kmem_slab_cache、kmem_bufctl_cache などがあります。

カーネルメモリーキャッシュ

この節では、カーネルメモリーキャッシュを検索し調べる方法について説明します。::kmastat コマンドを実行することによって、システムの種々の kmem キャッシュについて調べることができます。

```
> ::kmastat
cache                        buf    buf    buf    memory     alloc alloc
```

```
name                        size in use  total    in use   succeed  fail
------------------------- ------ ------ ------ --------- --------- -----
kmem_magazine_1                8     24   1020      8192        24     0
kmem_magazine_3               16    141    510      8192       141     0
kmem_magazine_7               32     96    255      8192        96     0
...
kmem_alloc_8                   8   3614   3751     90112   9834113     0
kmem_alloc_16                 16   2781   3072     98304   8278603     0
kmem_alloc_24                 24    517    612     24576    680537     0
kmem_alloc_32                 32    398    510     24576    903214     0
kmem_alloc_40                 40    482    584     32768    672089     0
...
thread_cache                 368    107    126     49152    669881     0
lwp_cache                    576    107    117     73728       182     0
turnstile_cache               36    149    292     16384    670506     0
cred_cache                    96      6     73      8192   2677787     0
...
```

::kmastat を実行すれば、「正常な」システムの感じをつかむことができます。これは、システムのメモリーをリークしている過度に大きなキャッシュを見つけるのに役立ちます。::kmastat を実行した結果は、それを実行しているシステム、実行しているプロセスの数などによって異なります。

種々の kmem キャッシュのリストを表示するもう 1 つの方法は、::kmem_cache コマンドを使用することです。

```
> ::kmem_cache
ADDR     NAME                       FLAG  CFLAG  BUFSIZE  BUFTOTL
70036028 kmem_magazine_1            0020 0e0000        8     1020
700362a8 kmem_magazine_3            0020 0e0000       16      510
70036528 kmem_magazine_7            0020 0e0000       32      255
...
70039428 kmem_alloc_8               020f 000000        8     3751
700396a8 kmem_alloc_16              020f 000000       16     3072
70039928 kmem_alloc_24              020f 000000       24      612
70039ba8 kmem_alloc_32              020f 000000       32      510
7003a028 kmem_alloc_40              020f 000000       40      584
...
```

このコマンドは、キャッシュ名をアドレスに対応づける点で有用です。FLAG 欄には各キャッシュのデバッギングフラグが示されます。重要なことは、アロケータのデバッギング機能の選択がキャッシュごとにこのフラグのセットに基づいて行われていることを理解することです。これらは、キャッシュの作成時に、大域 kmem_flags 変数とともに設定されます。システムの実行中に kmem_flags を設定しても、その後に作成されたキャッシュを除いて (起動後にキャッシュが作成されることはまれです)、デバッギング動作には影響を与えません。

次に、MDB の kmem_cache walker を使用して、直接 kmem キャッシュのリストを調べます。

```
> ::walk kmem_cache
70036028
```

```
700362a8
70036528
700367a8
...
```

これによって、カーネルの各 kmem キャッシュに対応するポインタのリストが作成されます。特定のキャッシュを見つけるには、kmem_cache マクロを適用します。

```
> 0x70039928$<kmem_cache
0x70039928:     lock
0x70039928:     owner/waiters
                0
0x70039930:     flags           freelist        offset
                20f             707c86a0        24
0x7003993c:     global_alloc    global_free     alloc_fail
                523             0               0
0x70039948:     hash_shift      hash_mask       hash_table
                5               1ff             70444858
0x70039954:     nullslab
0x70039954:     cache           base            next
                70039928        0               702d5de0
0x70039960:     prev            head            tail
                707c86a0        0               0
0x7003996c:     refcnt          chunks
                -1              0
0x70039974:     constructor     destructor      reclaim
                0               0               0
0x70039980:     private         arena           cflags
                0               104444f8        0
0x70039994:     bufsize         align           chunksize
                24              8               40
0x700399a0:     slabsize        color           maxcolor
                8192            24              32
0x700399ac:     slab_create     slab_destroy    buftotal
                3               0               612
0x700399b8:     bufmax          rescale         lookup_depth
                612             1               0
0x700399c4:     kstat           next            prev
                702c8608        70039ba8        700396a8
0x700399d0:     name    kmem_alloc_24
0x700399f0:     bufctl_cache    magazine_cache  magazine_size
                70037ba8        700367a8        15
...
```

デバッギングにとって重要なフィールドは、「bufsize」、「flags」、および「name」です。kmem_cache の名前 (この場合は「kmem_alloc_24」) は、このシステムでの目的を示しています。bufsize は、このキャッシュの各バッファのサイズです。この場合、このキャッシュは、サイズ 24 以下の割り当てに使用されます。flags は、このキャッシュに対してどのデバッギング機能が有効になっているかを示しています。これは <sys/kmem_impl.h> の中に定義されているデバッギングフラグです。この場合には「flags」は 0x20f ですが、これは KMF_AUDIT | KMF_DEADBEEF | KMF_REDZONE | KMF_CONTENTS | KMF_HASH です。各デバッギング機能については、この後の各節で説明します。

特定のキャッシュのバッファを調べたい場合には、そのキャッシュの割り当て済みおよび未使用バッファを直接調べることができます。

```
> 0x70039928::walk kmem
704ba010
702ba008
704ba038
702ba030
...

> 0x70039928::walk freemem
70a9ae50
70a9ae28
704bb730
704bb2f8
...
```

MDB は、kmem walker にキャッシュアドレスを供給するショートカットを用意しています。kmem キャッシュごとに特定の walker が提供され、その walker の名前はキャッシュの名前と同じです。たとえば、次のようになります。

```
> ::walk kmem_alloc_24
704ba010
702ba008
704ba038
702ba030
...

> ::walk thread_cache
70b38080
70aac060
705c4020
70aac1e0
...
```

これで、カーネルメモリーアロケータの内部データ構造体に対して繰り返す方法や、kmem_cache データ構造体の最も重要なメンバーを調べる方法がわかります。

メモリー破壊の検出

アロケータの主要デバッギング機能の 1 つは、データの損傷をすばやく認識するアルゴリズムです。破壊が検出されると、アロケータによりただちにシステムでパニックが発生します。

この節では、アロケータがどのようにしてデータの損傷を認識するかを説明します。これらの問題をデバッグするには、この点を理解しておく必要があります。メモリーの誤用は、一般的に次のいずれかの原因によるものです。

- バッファの限度を超える書き込み
- 初期化されていないデータへのアクセス
- 解放されたバッファの継続使用
- カーネルメモリーの破壊

この後の 3 つの節を読む際には、これらの問題を覚えておいてください。アロケータの設計を理解する上で役立ち、問題を効率的に診断できます。

未使用バッファの検査 (0xdeadbeef)

kmem_cache の flags フィールドの KMF_DEADBEEF (0x2) ビットが設定されている場合、アロケータは、すべての未使用バッファに特殊なパターンを書き込むためメモリー破壊を簡単に検出できます。このパターンは 0xdeadbeef です。一般的なメモリーの領域は、割り当て済みメモリーと未使用メモリーの両方を含んでいるので、各種のブロックのセクションが混在します。kmem_alloc_24 キャッシュの一例を以下に示します。

```
0x70a9add8:     deadbeef        deadbeef
0x70a9ade0:     deadbeef        deadbeef
0x70a9ade8:     deadbeef        deadbeef
0x70a9adf0:     feedface        feedface
0x70a9adf8:     70ae3260        8440c68e
0x70a9ae00:     5               4ef83
0x70a9ae08:     0               0
0x70a9ae10:     1               bbddcafe
0x70a9ae18:     feedface        4fffed
0x70a9ae20:     70ae3200        d1befaed
0x70a9ae28:     deadbeef        deadbeef
0x70a9ae30:     deadbeef        deadbeef
0x70a9ae38:     deadbeef        deadbeef
0x70a9ae40:     feedface        feedface
0x70a9ae48:     70ae31a0        8440c54e
```

0x70a9add8 で始まるバッファは、0xdeadbeef のパターンが使用されています。このパターンによって、そのバッファが現在未使用であることがただちにわかります。0x70a9ae28 から次の未使用バッファが始まっています。それらの間の 0x70a9ae00 で始まる領域に割り当て済みバッファがあります。

注 - この図にはいくつかの穴があいていて、ここに示された 120 バイトのうち、3 つの 24 バイト領域によって 72 バイトのメモリーしか占有されていません。この不一致については、84ページの「レッドゾーン (0xfeedface)」で説明します。

レッドゾーン (0xfeedface)

上記のバッファには、0xfeedface のパターンが頻繁に現れています。このパターンは、レッドゾーンインジケータと呼ばれるものです。これによって、アロケータ (および問題のデバッギングを行なっているプログラマ) は、「バグのある」コードがバッファの境界を超えているかどうかを判断することができます。レッドゾーンの後に追加の情報があります。このデータの内容は他の要因によって異なります (88 ページの「メモリー割り当てログ」を参照)。レッドゾーンとそのあとのデータ領域は、まとめて *buftag* 領域と呼ばれます。 図 6–1 に、この情報の要約を示します。

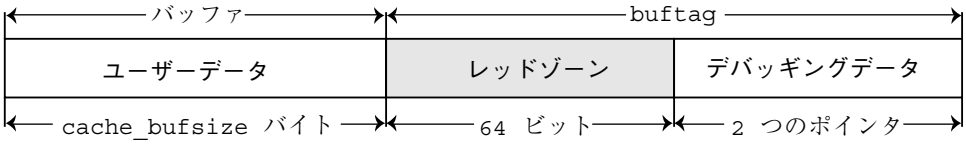

図 6–1　レッドゾーン

バッファのキャッシュに KMF_AUDIT、KMF_DEADBEEF、KMF_REDZONE、または KMF_CONTENTS フラグが設定されている場合には、そのキャッシュの各バッファに buftag が付加されます。buftag の内容は、KMF_AUDIT が設定されているかどうかにより異なります。

前述のメモリー領域を個別のバッファに分解すると、次のように簡単になります。

```
0x70a9add8:     deadbeef        deadbeef  \
0x70a9ade0:     deadbeef        deadbeef   +- ユーザーデータ (未使用)
0x70a9ade8:     deadbeef        deadbeef  /
0x70a9adf0:     feedface        feedface  -- レッドゾーン
0x70a9adf8:     70ae3260        8440c68e  -- デバッギングデータ

0x70a9ae00:     5               4ef83     \
0x70a9ae08:     0               0          +- ユーザーデータ (割り当て済み)
0x70a9ae10:     1               bbddcafe  /
0x70a9ae18:     feedface        4fffed    -- レッドゾーン
0x70a9ae20:     70ae3200        d1befaed  -- デバッギングデータ

0x70a9ae28:     deadbeef        deadbeef  \
0x70a9ae30:     deadbeef        deadbeef   +- ユーザーデータ (未使用)
0x70a9ae38:     deadbeef        deadbeef  /
0x70a9ae40:     feedface        feedface  -- レッドゾーン
0x70a9ae48:     70ae31a0        8440c54e  -- デバッギングデータ
```

0x70a9add8 と 0x70a9ae28 の未使用バッファでは、レッドゾーンには 0xfeedfacefeedface が使用されています。これは、バッファが未使用であることを判断する便利な方法です。

0x70a9ae00 で始まる割り当て済みバッファでは、状況は異なります。レッドゾーンの前半は (0x70a9ae18 で) バッファが終わることを示すために使用されてお

り、後半にはレッドゾーンバイトが書き込まれています。77ページの「アロケータの基礎」で説明したことを思い出してください。割り当てには、次の 2 つのタイプがあります。

1) クライアントが、kmem_cache_alloc() を使用してメモリーを要求した場合。この場合には、要求されたバッファのサイズは、キャッシュの bufsize と等しくなります。

2) クライアントが、kmem_alloc(9F) を使用してメモリーを要求した場合。この場合には、要求されたバッファのサイズは、キャッシュの bufsize 以下になります。たとえば、20 バイトの要求は、kmem_alloc_24 キャッシュによって満たされます。アロケータは、クライアントデータのすぐ後にレッドゾーンバイトを調整して強制的にバッファ境界を合わせます。

```
0x70a9ae00:     5               4ef83     \
0x70a9ae08:     0               0          +- ユーザーデータ (割り当て済み)
0x70a9ae10:     1               bbddcafe  /
0x70a9ae18:     feedface        4fffed    -- レッドゾーン
0x70a9ae20:     70ae3200        d1befaed  -- デバッギングデータ
```

0x70a9ae18 にある 0xfeedface の後には、ランダムな値のように見える 32 ビットのワードがあります。この数字は、実際にはバッファサイズの符号化された表現です。この数字を復号化して割り当て済みバッファのサイズを知るには、次の公式を使用します。

```
size = redzone_value / (UINT_MAX / KMEM_MAXBUF)
```

KMEM_MAXBUF の値は 16384 であり、UINT_MAX の値は 4294967295 です。したがってこの例では、次のようになります。

```
size = 0x4fffed / (4294967295 / 16384) = 20 bytes.
```

これは、要求されたバッファのサイズが 20 バイトであることを示しています。アロケータはこの復号化操作を行なって、レッドゾーンバイトがオフセット 20 であることを知ります。レッドゾーンバイトは 16 進パターン 0xbb です。これは予想通り、0x729084e4 (0x729084d0 + 0t20) に存在しています。

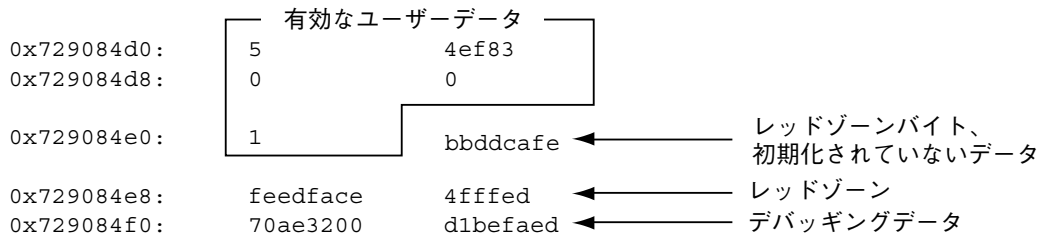

図 6–2　kmem_alloc(9F) バッファの例

図 6–3 に、メモリー配置の一般的形式を示します。

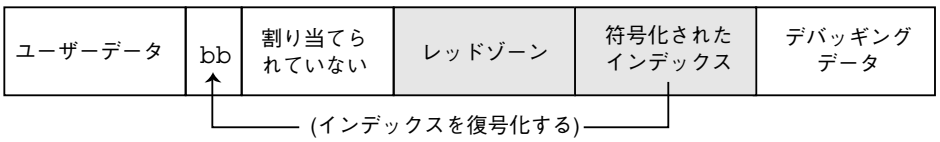

図 6–3　レッドゾーンバイト

割り当てサイズがキャッシュの bufsize と同じである場合には、図 6–4 に示すように、レッドゾーン自体の最初のバイトにレッドゾーンバイトが上書きされます。

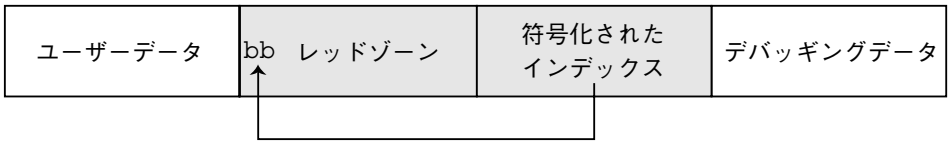

図 6–4　レッドゾーンの先頭にあるレッドゾーンバイト

この上書きの結果、レッドゾーンの最初の 32 ビットワードは 0xbbedface または 0xfeedfabb になります。このどちらになるかは、システムを実行しているハードウェアのエンディアンによります。

注 - 割り当てサイズがこのような方法で符号化されるのはなぜでしょうか。サイズを符号化するために、アロケータは公式 ((UINT_MAX / KMEM_MAXBUF) * size + 1) を使用します。サイズを復号化する際には、整数の割り算を行い、余りの「+1」は捨てられます。しかし、この追加された 1 は貴重な役割を果たします。なぜなら、アロケータは (size % (UINT_MAX / KMEM_MAXBUF) == 1) になるかどうかをテストすることにより、サイズが有効かどうかをチェックできるからです。このようにして、アロケータはレッドゾーンバイトインデックスの破壊に対処します。

初期化されていないデータ (0xbaddcafe)

アドレス 0x729084d4 の 0xbbddcafe は、ワードの最初のバイトにレッドゾーンバイトが上書きされる前には何と書いてあったのでしょうか。0xbaddcafe だったのです。キャッシュに KMF_DEADBEEF フラグが設定されると、割り当てられたけれども初期化されていないメモリーには、パターン 0xbaddcafe が使用されます。アロケータが割り当てを行う際には、バッファの各ワードをループし、各ワードに 0xdeadbeef が含まれていることを検証し、次にそのワードに 0xbaddcafe を使用します。

システムが次のようなメッセージを出してパニックを引き起こす場合があります。

```
panic[cpu1]/thread=e1979420: BAD TRAP: type=e (Page Fault)
rp=ef641e88 addr=baddcafe occurred in module "unix" due to an
illegal access to a user address
```

この場合、障害の原因になったアドレスは 0xbaddcafe です。スレッドがパニックを起こしたのは、初期化されていないデータにアクセスしたためです。

パニックメッセージと障害の関係

カーネルメモリーアロケータは、前述した障害モードに対応してパニックメッセージを出します。たとえば、システムが次のようなメッセージを出してパニックを引き起こす場合があります。

```
kernel memory allocator: buffer modified after being freed
modification occurred at offset 0x30
```

アロケータは、問題のバッファに 0xdeadbeef が使用されていることを確認するので、この場合を検出することができます。オフセット 0x30 ではこの条件が満たさ

れていませんでした。この状態はメモリー破壊を示しているので、アロケータによりシステムにパニックが発生しました。

障害メッセージのもう 1 つの例を次に示します。

```
kernel memory allocator: redzone violation: write past end of buffer
```

アロケータは、レッドゾーンサイズの符号化から判定した場所にレッドゾーンバイト (0xbb) が存在することを確認するので、この問題を検出することができます。しかし、アロケータは正しい場所にこのシグニチャーバイトを見つけることができませんでした。これはメモリー破壊を示しているので、アロケータによりシステムにパニックが発生しました。その他のアロケータパニックメッセージについては、後で説明します。

メモリー割り当てログ

この節では、カーネルメモリーアロケータのログ機能と、この機能を使用してシステムクラッシュのデバッギングを行う方法について説明します。

buftag データの完全性

前述のように、各 buftag の後半には、対応するバッファに関する追加情報が含まれています。この情報の一部はデバッギング情報であり、また、アロケータの内部データも含まれています。この補助的データは種々の形式をとりますが、まとめて「バッファ制御」データあるいは *bufctl* データと呼ばれます。

しかし、誤ったコードによってこの bufctl ポインタも破壊される場合があるので、アロケータはバッファの bufctl ポインタが有効であるかどうかを知る必要があります。アロケータは、このポインタとその符号化されたバージョンを格納し、2 つのバージョンのクロスチェックを行うことにより、この補助ポインタの完全性を確認します。

図 6–5 に示すように、ポインタの 2 つのバージョンは、*bcp* (buffer control pointer) と *bxstat* (buffer control XOR status) です。アロケータは、式 `bcp XOR bxstat` がわかりやすい既知の値に等しくなるように bcp と bxstat を調整します。

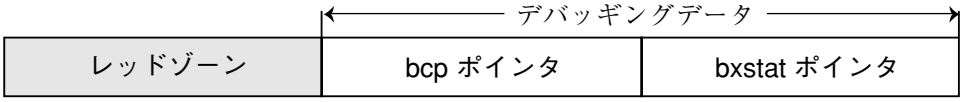

図 6–5　buftag の追加のデバッギングデータ

これらのポインタの一方または両方が壊れている場合には、アロケータは容易に破壊を検出し、システムにパニックを発生させます。バッファが割り当て済みの場合には、`bcp XOR bxstat = 0xa110c8ed` (「allocated」) になります。バッファが未使用の場合には、`bcp XOR bxstat = 0xf4eef4ee` (「freefree」) になります。

注 - 83ページの「未使用バッファの検査 (0xdeadbeef)」に示されている例をもう一度調べて、例に示されている buftag ポインタがこの説明どおりであることを確認してください。

アロケータは、buftag が壊れていることを発見した場合には、システムにパニックを発生させ、次のようなメッセージを出します。

```
kernel memory allocator: boundary tag corrupted
    bcp ^ bxstat = 0xffeef4ee, should be f4eef4ee
```

bcp が壊れていても、そのバッファが割り当て済みか未使用かによって、それぞれ `bxstat XOR 0xf4eef4ee` または `bxstat XOR 0xa110c8ed` の値からその値を取り出すことが可能です。

bufctl ポインタ

buftag 領域に含まれているバッファ制御 (bufctl) ポインタは、そのキャッシュの `kmem_flags` に応じて種々の意味を持ちます。`KMF_AUDIT` フラグによって切り替えられる動作は、特に興味深いものです。KMF_AUDIT フラグが設定されていない場合には、カーネルメモリーアロケータは、各バッファの `kmem_bufctl_t` 構造体を割り当てます。この構造体には、各バッファに関する最小限のアカウンティング情報が含まれています。`KMF_AUDIT` フラグが設定されている場合には、アロケータはこの代わりに、`kmem_bufctl_t` の拡張バージョンである `kmem_bufctl_audit_t` を割り当てます。

この節では、`KMF_AUDIT` フラグが設定されていることを前提とします。このビットが設定されていないキャッシュは、使用可能なデバッギング情報の量が少なくなります。

kmem_bufctl_audit_t (略称は bufctl_audit) には、このバッファに対して発生した最後のトランザクションに関する追加情報が含まれています。次の例で、bufctl_audit マクロを適用して監査レコードを調べる方法を示します。ここに示したバッファは、82ページの「メモリー破壊の検出」で使用したサンプルバッファです。

```
> 0x70a9ae00,5/KKn
0x70a9ae00:     5               4ef83
                0               0
                1               bbddcafe
                feedface        4fffed
                70ae3200        d1befaed
```

上記の手法を使用すると、0x70ae3200 が bufctl_audit レコードを指していることが容易にわかります。これはレッドゾーンの後の最初のポインタです。bufctl_audit レコードを調べるには、bufctl_audit マクロを適用します。

```
> 0x70ae3200$<bufctl_audit
0x70ae3200:     next            addr            slab
                70378000        70a9ae00        707c86a0
0x70ae320c:     cache           timestamp       thread
                70039928        e1bd0e26afe     70aac4e0
0x70ae321c:     lastlog         contents        stackdepth
                7011c7c0        7018a0b0        4
0x70ae3228:
                kmem_zalloc+0x30
                pid_assign+8
                getproc+0x68
                cfork+0x60
```

「addr」フィールドは、この bufctl_audit レコードに対応するバッファのアドレスです。これはオリジナルアドレス 0x70a9ae00 です。「cache」フィールドは、このバッファが割り当てられている kmem_cache を指します。::kmem_cache dcmd を使用して、次のようにしてこのキャッシュを調べることができます。

```
> 0x70039928::kmem_cache
ADDR     NAME                    FLAG  CFLAG  BUFSIZE  BUFTOTL
70039928 kmem_alloc_24           020f 000000       24      612
```

「timestamp」フィールドは、このトランザクションが発生した時刻を表します。この時刻は gethrtime(3C) と同じ形式で表されます。

「thread」は、このバッファに対して最後のトランザクションを行なったスレッドへのポインタです。「lastlog」および「contents」ポインタは、アロケータのトランザクションログの中の位置を指します。これらのログについては、94ページの「アロケータのログ機能」で詳しく説明します。

一般的に、bufctl_audit が提供する最も有用な情報は、トランザクションが発生した時点で記録されるスタックトレースです。この場合、このトランザクションは fork(2) の実行の一部として呼び出された割り当てです。

拡張メモリー解析

この節では、メモリーリークとデータ破壊の原因などの拡張メモリーの解析について説明します。

メモリーリークの発見

::findleaks dcmd を使用して、フルセットの kmem デバッギング機能が有効になっている場合に、カーネルクラッシュダンプの際に効率的にメモリーリークの検出を行うことができます。::findleaks の最初の実行では、ダンプを処理してメモリーリークを探します。この処理には数分かかる場合があります。次に、割り当てスタックトレース別にリークがまとめられます。findleaks レポートには、識別されたメモリーリークごとに bufctl アドレスと先頭のスタックフレームが示されます。

```
> ::findleaks
CACHE     LEAKED   BUFCTL CALLER
70039ba8       1 703746c0 pm_autoconfig+0x708
70039ba8       1 703748a0 pm_autoconfig+0x708
7003a028       1 70d3b1a0 sigaddq+0x108
7003c7a8       1 70515200 pm_ioctl+0x187c
------------------------------------------------------------------------
   Total       4 buffers, 376 bytes
```

bufctl ポインタを使用し、bufctl_audit マクロを適用して、その割り当ての完全なスタックバックトレースを得ることができます。

```
> 70d3b1a0$<bufctl_audit
0x70d3b1a0:     next            addr            slab
                70a049c0        70d03b28        70bb7480
0x70d3b1ac:     cache           timestamp       thread
                7003a028        13f7cf63b3      70b38380
0x70d3b1bc:     lastlog         contents        stackdepth
                700d6e60        0               5
0x70d3b1c8:
                kmem_alloc+0x30
                sigaddq+0x108
                sigsendproc+0x210
                sigqkill+0x90
                kill+0x28
```

プログラマは、通常、bufctl_audit 情報と割り当てスタックトレースの割り当てを使用して、そのバッファのリークの原因となったコードパスをすばやく突き止めることができます。

データへの参照の発見

メモリー破壊の診断を行う際は、他のどのカーネルエンティティが特定のポインタのコピーを保持しているかを知る必要があります。これは重要なことです。なぜならデータ構造体が解放された後どのスレッドがこれにアクセスしたかを明らかにできるからです。また、特定の (有効な) データ項目の知識をどのカーネルエンティティが共有しているかを知ることが容易になります。このためには ::whatis dcmd と ::kgrep dcmd を使用します。次のようにして、問題の値に対して ::whatis を適用します。

```
> 0x705d8640::whatis
705d8640 is 705d8000+640, allocated from kmem_va_8192
705d8640 is 705d8640+0, allocated from streams_mblk
```

この場合は、0x705d8640 が STREAMS mblk 構造体へのポインタであることが明らかになりました。この割り当ては、kmem_va 仮想記憶領域の前の段階の kmem キャッシュである kmem_va_8192 キャッシュにも見られます。::kmastat dcmd を使用すれば、kmem キャッシュと vmem 領域のリストが表示されます。::kgrep を使用して、この mblk へのポインタを含む他のカーネルアドレスを突き止めることができます。これによって、システムのメモリー割り当ての階層的特徴が明らかになります。一般的に、特殊な kmem キャッシュの名前から、そのアドレスによって参照されるオブジェクトのタイプを判断することができます。

```
> 0x705d8640::kgrep
400a3720
70580d24
7069d7f0
706a37ec
706add34
```

再び ::whatis を適用します。

```
> 400a3720::whatis
400a3720 is in thread 7095b240's stack

> 706add34::whatis
706add34 is 706ac000+1d34, allocated from kmem_va_8192
706add34 is 706add20+14, allocated from streams_dblk_120
```

1 つのポインタは既知のカーネルスレッドのスタック上にあり、もう 1 つのポインタは対応する STREAMS dblk 構造体の内部の mblk ポインタであることがわかりました。

::kmem_verify を使用したバッファの障害の発見

MDB の ::kmem_verify dcmd を使用すると、kmem アロケータが実行時に行う検査とほぼ同じ検査を行います。::kmem_verify を起動して、該当する kmem_flags が設定されている場合にすべての kmem キャッシュを走査し、あるいは特定のキャッシュを調べることができます。

::kmem_verify を使用して問題を突き止める例を、以下に示します。

```
> ::kmem_verify
Cache Name                      Addr     Cache Integrity
kmem_alloc_8                    70039428 clean
kmem_alloc_16                   700396a8 clean
kmem_alloc_24                   70039928 1 corrupt buffer
kmem_alloc_32                   70039ba8 clean
kmem_alloc_40                   7003a028 clean
kmem_alloc_48                   7003a2a8 clean
...
```

::kmem_verify によれば、明らかに kmem_alloc_24 キャッシュには問題が存在します。明示的なキャッシュ引数を指定すると、::kmem_verify dcmd はこの問題に関するより詳細な情報を提供します。

```
> 70039928::kmem_verify
Summary for cache 'kmem_alloc_24'
  buffer 702babc0 (free) seems corrupted, at 702babc0
```

次に、::kmem_verify によって障害があると認識されたバッファを調べます。

```
> 0x702babc0,5/KKn
0x702babc0:     0               deadbeef
                deadbeef        deadbeef
                deadbeef        deadbeef
                feedface        feedface
                703785a0        84d9714e
```

::kmem_verify がこのバッファにフラグを立てた理由が明らかになりました。バッファの最初のワード (0x702babc0 で始まる) には、0xdeadbeef のパターンが使用されるはずであったのに、0 が使用されています。この時点で、このバッファの bufctl_audit を調べることによって、このバッファにどのコードが最近書き込みを行なったか、どこでいつ解放されたかについての手がかりが得られます。

この状況でのもう 1 つの有用な手法は、::kgrep を使用してアドレス空間を調べてアドレス 0x702babc0 への参照を検索し、この解放されたデータへの参照を依然として保持しているスレッドまたはデータ構造体を発見することです。

アロケータのログ機能

キャッシュの KMF_AUDIT が設定されている場合、カーネルメモリーのアロケータは、アクティビティの最近の履歴を記録するログを維持します。このトランザクションログには、bufctl_audit レコードが記録されます。KMF_AUDIT と KMF_CONTENTS の両方のフラグが設定されている場合には、アロケータは、割り当て済みバッファと解放されたバッファの実際の内容の一部を記録したログを生成します。このログの構造と使用法については、このマニュアルでは記載していません。この節では、トランザクションログについて説明します。

MDB は、トランザクションログを表示するための複数の機能を備えています。最も簡単な方法は、::walk kmem_log です。これは、このログに記録されているトランザクションを一連の bufctl_audit_t ポインタの形で出力します。

```
> ::walk kmem_log
70128340
701282e0
70128280
70128220
701281c0
...
> 70128340$<bufctl_audit
0x70128340:     next            addr            slab
                70ac1d40        70bc4ea8        70bb7c00
0x7012834c:     cache           timestamp       thread
                70039428        e1bd7abe721     70aacde0
0x7012835c:     lastlog         contents        stackdepth
                701282e0        7018f340        4
0x70128368:
                kmem_cache_free+0x24
                nfs3_sync+0x3c
                vfs_sync+0x84
                syssync+4
```

トランザクションログ全体を表示するもっと簡潔な方法は、::kmem_log コマンドを使用することです。

```
> ::kmem_log
CPU ADDR     BUFADDR         TIMESTAMP THREAD
  0 70128340 70bc4ea8      e1bd7abe721 70aacde0
  0 701282e0 70bc4ea8      e1bd7aa86fa 70aacde0
  0 70128280 70bc4ea8      e1bd7aa27dd 70aacde0
  0 70128220 70bc4ea8      e1bd7a98a6e 70aacde0
  0 701281c0 70d03738      e1bd7a8e3e0 70aacde0
  ...
  0 70127140 70cf78a0      e1bd78035ad 70aacde0
```

```
  0 701270e0 709cf6c0      e1bd6d2573a 40033e60
  0 70127080 70cedf20      e1bd6d1e984 40033e60
  0 70127020 70b09578      e1bd5fc1791 40033e60
  0 70126fc0 70cf78a0      e1bd5fb6b5a 40033e60
  0 70126f60 705ed388      e1bd5fb080d 40033e60
  0 70126f00 705ed388      e1bd551ff73 70aacde0
  ...
```

::kmem_log の出力は、時刻表示の降順にソートされます。ADDR 欄は、このトランザクションに対応する bufctl_audit 構造体です。BUFADDR は、実際のバッファを指しています。

これらの数字は、バッファに対するトランザクション (割り当てと解放) を表しています。特定のバッファが壊れた場合、トランザクションログの中でそのバッファを突き止め、そのトランザクションを行なったスレッドが他のどのトランザクションにかかわっていたかを判断することは有用です。このことは、バッファの割り当て (または解放) の前後に発生したイベントのシーケンスの全体像を理解するのに役立ちます。

::bufctl コマンドを使用して、トランザクションログの調査の出力をフィルタリングすることができます。::bufctl -a コマンドは、トランザクションログの中のバッファをバッファアドレスによってフィルタリングします。次の例は、バッファ 0x70b09578 のフィルタリングの結果です。

```
> ::walk kmem_log | ::bufctl -a 0x70b09578
ADDR     BUFADDR   TIMESTAMP   THREAD   CALLER
70127020 70b09578  e1bd5fc1791 40033e60 biodone+0x108
70126e40 70b09578  e1bd55062da 70aacde0 pageio_setup+0x268
70126de0 70b09578  e1bd52b2317 40033e60 biodone+0x108
70126c00 70b09578  e1bd497ee8e 70aacde0 pageio_setup+0x268
70120480 70b09578  e1bd21c5e2a 70aacde0 elfexec+0x9f0
70120060 70b09578  e1bd20f5ab5 70aacde0 getelfhead+0x100
7011ef20 70b09578  e1bd1e9a1dd 70aacde0 ufs_getpage_miss+0x354
7011d720 70b09578  e1bd1170dc4 70aacde0 pageio_setup+0x268
70117d80 70b09578  e1bcff6ff27 70bc2480 elfexec+0x9f0
70117960 70b09578  e1bcfea4a9f 70bc2480 getelfhead+0x100
...
```

この例は、特定のバッファが多くのトランザクションに使用される場合があることを示しています。

注 - kmem トランザクションログは、カーネルメモリーアロケータが行なったトランザクションのすべての記録ではないことを忘れないでください。ログのサイズを一定に保つために、ログの中の古い記録は消去されます。

::allocdby dcmds と ::freedby dcmds を使用して、特定のスレッドに関連するトランザクションの要約を示すことができます。次の例では、スレッド 0x70aacde0 によって行われた最近の割り当てのリストが示されています。

```
> 0x70aacde0::allocdby
BUFCTL      TIMESTAMP CALLER
70d4d8c0  e1edb14511a allocb+0x88
70d4e8a0  e1edb142472 dblk_constructor+0xc
70d4a240  e1edb13dd4f allocb+0x88
70d4e840  e1edb13aeec dblk_constructor+0xc
70d4d860  e1ed8344071 allocb+0x88
70d4e7e0  e1ed8342536 dblk_constructor+0xc
70d4a1e0  e1ed82b3a3c allocb+0x88
70a53f80  e1ed82b0b91 dblk_constructor+0xc
70d4d800  e1e9b663b92 allocb+0x88
```

bufctl_audit レコードを調べることにより、特定のスレッドの最近のアクティビティを理解することができます。

第 7 章

モジュールプログラミング API

この章では、MDB デバッガモジュール API に含まれている構造体と関数について説明します。ヘッダーファイル <sys/mdb_modapi.h> にこれらの関数のプロトタイプが含まれているほか、SUNWmdbdem パッケージには、ディレクトリ /usr/demo/mdb にあるサンプルモジュールのソースコードが入っています。

デバッガモジュールのリンケージ

_mdb_init()

```
const mdb_modinfo_t *_mdb_init(void);
```

リンケージと識別を可能にするために、各デバッガモジュールには _mdb_init() という関数を提供する必要があります。この関数は、自動変数として宣言されない固定の mdb_modinfo_t 構造体を指すポインタを返します。これについては、</sys/mdb_modapi.h> に次のように定義されています。

```
typedef struct mdb_modinfo {
   ushort_t mi_dvers;              /* デバッガの API のバージョン番号 */
   const mdb_dcmd_t *mi_dcmds;     /* NULL で終了する dcmd のリスト */
   const mdb_walker_t *mi_walkers; /* NULL で終了する walk のリスト */
} mdb_modinfo_t;
```

mi_dvers メンバーは API のバージョン番号を識別するためのもので、常に MDB_API_VERSION に設定されます。このようにして現在のバージョン番号が各デバッガモジュールの中にコンパイルされているので、デバッガは、モジュールが使用するアプリケーションのバイナリインタフェースを識別し、検証できます。デ

バッガは、自らのバージョンより新しいバージョンの API に対してコンパイルされているモジュールは読み込みません。

mi_dcmds と *mi_walkers* というメンバーは、NULL でない場合はそれぞれ dcmd と walker の定義構造体の配列を指しています。どちらの配列も NULL 要素で終了していなければなりません。これらの dcmd と walker は、モジュールを読み込むプロセスの一部としてデバッガによってインストールされ、登録されます。dcmd または walker が正しく定義されていなかったり、名前が重複していたり無効であったりした場合、デバッガはそのモジュールの読み込みを拒否します。dcmd と walker の名前には、引用符や括弧など、デバッガにとって特別な意味を持つ文字を入れることはできません。

モジュールでは、モジュール API を使用して `_mdb_init()` のコードを実行し、読み込むべきかどうかを判定することもできます。たとえば、特定のシンボルが存在する場合だけ、特定のターゲットに対して該当するモジュールは存在します。これらのシンボルが見つからない場合、このモジュールは `_mdb_init()` 関数から NULL を返します。この場合、デバッガによってこのモジュールの読み込みは拒否され、該当するエラーメッセージが出力されます。

`_mdb_fini()`

```
void _mdb_fini(void);
```

`mdb_alloc()` によって以前に割り当てられた固定メモリーの解放など、読み込み解除に先立って一定のタスクを実行するモジュールの場合は、`_mdb_fini()` という関数を宣言してこれを行うことができます。この関数はデバッガでは必要とされません。この関数を宣言すると、モジュールの解除の前に一度呼び出されます。ユーザーがデバッガの終了を要求したとき、またはユーザーが `::unload` 組み込み dcmd を使用して明示的にモジュールを解除したときに、モジュールは解除されます。

dcmd の定義

```
int dcmd(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv);
```

dcmd は `dcmd()` の宣言に似た関数によって実装されます。この関数は次の 4 つの引数を受け取り、整数のステータスを返します。

addr 現在のアドレス。ドットともいう。dcmd の開始時点では、このアドレスはデバッガのドット“ . “ 変数の値に対応しています。

flags 次のフラグの 1 つ以上の論理和を含む整数

`DCMD_ADDRSPEC`	明示的なアドレスが `::dcmd` の左に指定された
`DCMD_LOOP`	dcmd が `,count` 構文を使ってループの中で呼び出されたか、またはパイプラインによってループの中で呼び出された
`DCMD_LOOPFIRST`	この dcmd 関数の呼び出しは、最初のループまたはパイプラインの呼び出しに対応している
`DCMD_PIPE`	dcmd がパイプラインからの入力に伴って呼び出された
`DCMD_PIPE_OUT`	dcmd がパイプラインに対して設定された出力に伴って呼び出された

便利な `DCMD_HDRSPEC()` マクロが用意されており、dcmd はフラグをテストしてヘッダーラインを出力するかどうかを決定できます。ヘッダーラインを出力するのは、ループの一部として呼び出されていない、あるいはループまたはパイプラインの繰り返しの最初の場合です。

argc *argv* 配列内の引数の数

argv コマンド行の `::dcmd` の右側に指定された引数の配列。この引数は文字列の場合と整数値の場合があります。

dcmd 関数は、`<sys/mdb_modapi.h>`に定義されている、次の整数値のどれかを返します。

`DCMD_OK`	dcmd は正常に完了した
`DCMD_ERR`	何らかの理由により dcmd は失敗した

DCMD_USAGE	無効な引数が指定されたため、dcmd は失敗した。この値が返される場合は、以降に述べるように dcmd の使用に関するメッセージが自動的に出力される
DCMD_NEXT	次の dcmd 定義がある場合は、同じ引数で自動的に呼び出される
DCMD_ABORT	dcmd が失敗したため、現在のループまたはパイプラインは強制終了される。この戻り値は DCMD_ERR に似ているが、現在のループまたはパイプラインを続行できないことを示す

各 dcmd は <sys/mdb_modapi.h> に定義されているように、サンプルの dcmd() プロトタイプにしたがって定義された関数と、それに対応する mdb_dcmd_t 構造体から構成されています。この構造体は、次のフィールドから構成されています。

const char *dc_name	dcmd の文字列名。先頭に "::" が付かない。この名前には、$ または ` などの MDB メタキャラクタを含めることはできない
const char *dc_usage	dcmd に対するオプションの用法文字列。dcmd が DCMD_USAGE を返すとこの文字列が出力される。たとえば、dcmd がオプション –a と –b を受け付ける場合、dc_usage は "[–ab]" と設定される。dcmd が引数を受け付けない場合、dc_usage は NULL に設定される。用法文字列が ":" で始まっている場合、dcmd でアドレスを明示的に指定する必要があり、フラグパラメタに DCMD_ADDRSPEC を設定する必要があることを示す。用法文字列が "?" で始まっている場合、dcmd はオプションでアドレスを受け付けることを示す。 これらのヒントに従って、用法メッセージも変更される
const char *dc_descr	dcmd の目的を簡単に説明する、必須の記述文字列。この文字列は単一行のテキストで構成される
mdb_dcmd_f *dc_funcp	dcmd を実行するために呼び出される関数を指すポインタ

`void (*dc_help)(void)`	dcmd のヘルプ関数を指すオプションの関数ポインタ。このポインタが NULL 以外の値の場合、ユーザーが `::help` *dcmd* を実行すると、この関数が呼び出される。この関数では `mdb_printf()` を使用して詳細情報や例を表示できる

walker の定義

```
int walk_init(mdb_walk_state_t *wsp);
int walk_step(mdb_walk_state_t *wsp);
void walk_fini(mdb_walk_state_t *wsp);
```

walker は `init`、`step`、および `fini` の 3 つの関数で構成されており、これらの関数は上記のプロトタイプの例に従って定義されています。walker は、`mdb_walk()` などのいずれかの walk 関数が呼び出されたとき、またはユーザーが `::walk` 組み込み dcmd を実行したときに、デバッガによって起動されます。`<sys/modapi.h>` に定義されているように、walk が開始されると、MDB は walker の init 関数を呼び出し、新規 `mdb_walk_state_t` 構造体のアドレスをこの関数に渡します。

```
typedef struct mdb_walk_state {
   mdb_walk_cb_t walk_callback;     /* 実行のためのコールバック */
   void *walk_cbdata;               /* 専用データのコールバック */
   uintptr_t walk_addr;             /* 現在のアドレス */
   void *walk_data;                 /* walk 専用データ */
   void *walk_arg;                  /* walk 専用引数 */
   void *walk_layer;                /* 配下の層からのデータ */
} mdb_walk_state_t;
```

walk ごとに個別に `mdb_walk_state_t` が作成されるため、同じ walker の複数のインスタンスを同時にアクティブにすることができます。たとえば `mdb_walk()` に指定されているように、state 構造体には、各ステップにおいて walker が呼び出すコールバック (*walk_callback*)、およびそのコールバックに対する専用データ (*walk_cbdata*) が含まれています。*walk_cbdata* ポインタは walker からは隠されているため、この値を変更したり、参照を解除したりすることはもちろん、有効なメモリーを指すポインタとみなすこともできません。

walk の開始アドレスは *walk_addr* に格納されています。このアドレスは `mdb_walk()` が呼び出された場合の NULL か、または `mdb_pwalk()` に指定されているアドレスパラメタのどちらかの値となります。`::walk` 組み込みコマンドが使用された場合、明示的なアドレスが `::walk` の左側に指定されているときは、*walk_addr* は NULL 以外の値となります。開始アドレスが NULL の walk のこ

とをグローバル walk といいます。NULL 以外の明示的な開始アドレスを持つ walk のことをローカル walk といいます。

walker 専用の記憶領域として *walk_data* および *walk_arg* フィールドが用意されています。複雑な walker の場合、補助的な state 構造体を割り当てて、この構造体を指すように *walk_data* を設定する必要があります。walk が開始されるたびに、*walk_arg* は、対応する walker の `mdb_walker_t` 構造体の `walk_init_arg` メンバーが持つ値に初期設定されます。

場合によっては、複数の walker に同じ init、step、および fini ルーチンを共有させると便利な場合があります。たとえば、MDB `genunix` モジュールは、各カーネルのメモリーキャッシュに対する walker を提供しています。これらの walker は同じ init、step、および fini 関数を共有しているため、`mdb_walker_t` の `walk_init_arg` メンバーを使用して、適切なキャッシュのアドレスを *walk_arg* として指定できます。

walker が `mdb_layered_walk()` を呼び出して配下の層をインスタンス化した場合、配下の層は walker の step 関数を呼び出す前に *walk_addr* と *walk_layer* をリセットします。配下の層は *walk_addr* を配下のオブジェクトのターゲットの仮想アドレスに設定し、*walk_layer* を配下のオブジェクトの walker のローカルコピーに設定します。階層化された walk については、以降の `mdb_layered_walk()` の説明を参照してください。

walker の init および step 関数は、次の状態値のどれかを返します。

`WALK_NEXT`	次のステップへ進む。walk init 関数が `WALK_NEXT` を返すと、MDB は walk step 関数を呼び出す。walk step 関数から `WALK_NEXT` が返されたときは、MDB がもう一度 step 関数を呼び出す必要があることを示す
`WALK_DONE`	walk が正常に完了した。`WALK_DONE` は、walk が完了したことを示すために step 関数から返される場合と、与えられたデータ構造体が空である場合などに step が不要であることを示すために init 関数から返される場合がある
`WALK_ERR`	walk がエラーのため終了した。`WALK_ERR` が init 関数から返された場合、`mdb_walk()` (またはそれに相当するもの) は –1 を返して、walker が初期化に失敗したことを示す。`WALK_ERR` が step 関数から返された場合、walk は終了するが、`mdb_walk()` 関数からは成功が返される

walk_callback からも、上記のどれかの値が返されます。したがって walk step 関数は次のオブジェクトのアドレスを決定し、このオブジェクトのローカルコピーを読み取って、*walk_callback* 関数を呼び出し、その状態を返します。walk が完了したか、またはエラーが発生した場合、step 関数もコールバックを呼び出さずに WALK_DONE または WALK_ERR を返すことがあります。

次に示すように、walker 自体は mdb_walker_t 構造体を使用して定義されます。

```
typedef struct mdb_walker {
   const char *walk_name;                 /* walk のタイプ名 */
   const char *walk_descr;                /* walk の記述 */
   int (*walk_init)(mdb_walk_state_t *);  /* walk コンストラクタ */
   int (*walk_step)(mdb_walk_state_t *);  /* walk イタレータ */
   void (*walk_fini)(mdb_walk_state_t *); /* walk デストラクタ */
   void *walk_init_arg;                   /* コンストラクタの引数 */
} mdb_walker_t;
```

walk_name および walk_descr フィールドは、それぞれ walker の名前と短い説明を含む文字列を指すように初期化されます。walker は NULL 以外の名前と説明を持つ必要があり、名前には MDB メタキャラクタを入れることはできません。説明の文字列は ::walkers および ::dmods 組み込み dcmd によって出力されます。

walk_init、walk_step、および walk_fini メンバーは、前述のように walk 関数自体を指しています。特別な初期化またはクリーンアップ措置が必要でないことを示すには、walk_init および walk_fini メンバーを NULL に設定します。

walk_step メンバーは NULL には設定できません。前述のように、walk_init_arg メンバーは、指定された walker に対して新規に作成された mdb_walk_state_t ごとに walk_arg メンバーを初期化するのに使用されます。一般的な walker の各ステップについては、図 7–1 を参照してください。

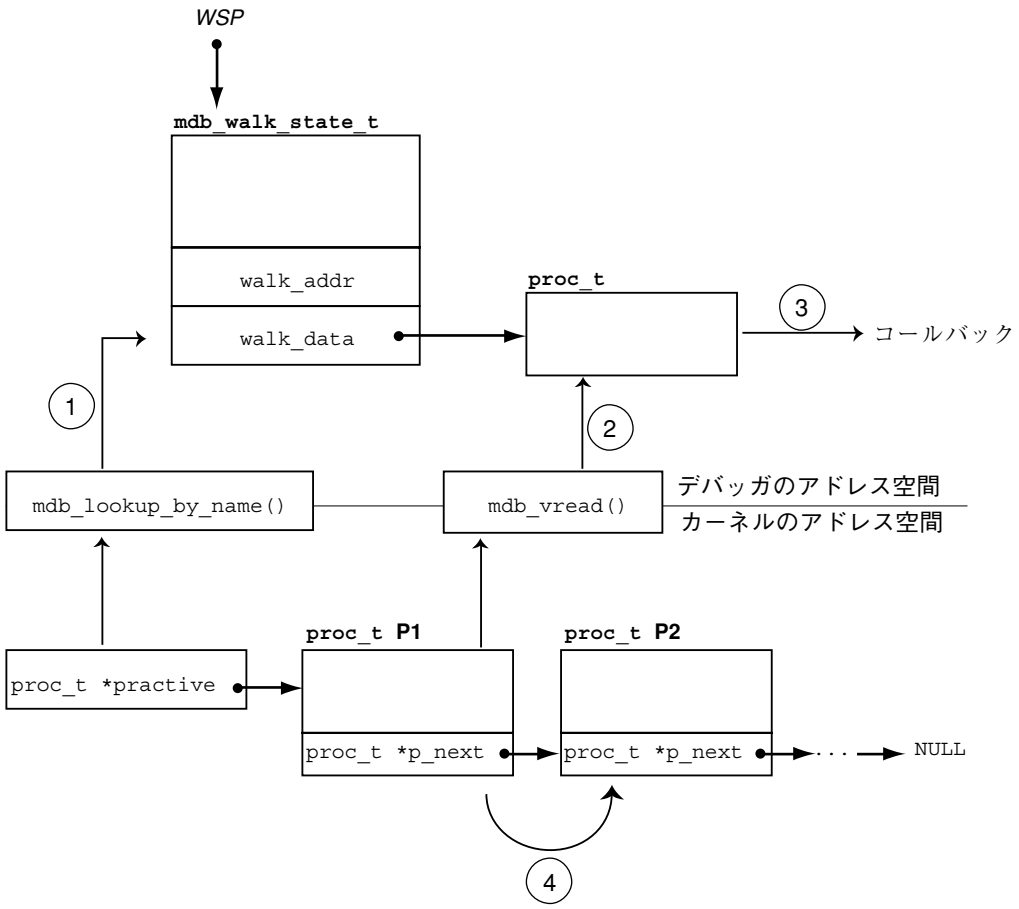

図 7–1　walker の例

walker はカーネル内の proc_t 構造体のリストに対して繰り返すように設計されています。リストの先頭は practive 大域変数に格納されており、各要素の p_next ポインタはリスト内の次の proc_t を指しています。リストの末尾は NULL ポインタになっています。ステップ (1) で walker の init ルーチン、practive のシンボルは mdb_lookup_by_name() を使用して検出され、この値が *wsp* の指す mdb_walk_state_t にコピーされます。

walker の step 関数では、ステップ (2) としてリスト内の次の proc_t 構造体が mdb_vread() を使用してデバッガのアドレス空間にコピーされます。ステップ (3) でこのローカルコピーを指すポインタによってコールバック関数が起動され、mdb_walk_state_t が次の繰り返しに対する proc_t 構造体のアドレスで更新されます。ステップ (4) では、この更新は、リスト内の次の要素を指す次のポインタに対応しています。

これらのステップは一般的な walker の構造を示しています。init ルーチンが特定のデータ構造体に関するグローバルな情報を検出し、step 関数が次のデータ項目のコピーを読み取ってコールバック関数に渡し、次の要素のアドレスが読み取られます。最終的に walk が終了すると、fini 関数によってすべての専用記憶領域が解放されます。

API 関数

mdb_pwalk()

```
int mdb_pwalk(const char *name, mdb_walk_cb_t func, void *data, uintptr_t addr);
```

name で指定された walker を使用して *addr* から始まるローカル walk を開始し、各ステップでコールバック関数 *func* を呼び出します。*addr* が NULL の場合、グローバル walk が実行されます。`mdb_pwalk()` を呼び出すことは *addr* パラメタを追跡せずに `mdb_walk()` を呼び出すことと同じです。この関数は成功した場合 0 を、エラーの場合 -1 を返します。walker 自体が致命的なエラーを返した場合、または指定された walker 名がデバッガに認識されない場合、`mdb_pwalk()` 関数は失敗します。walker 名に重複があった場合、逆引用符 (`) 演算子を使用して名前の有効範囲を指定できます。*data* パラメタは、呼び出し元にだけ意味を持つ隠された引数です。このパラメタは walk の各ステップで *func* に戻されます。

mdb_walk()

```
int mdb_walk(const char *name, mdb_walk_cb_t func, void *data);
```

name で指定された walker を使用して *addr* から始まるグローバル walk を開始し、各ステップでコールバック関数 *func* を起動します。この関数は成功した場合 0 を、エラーの場合 -1 を返します。walker 自体が致命的なエラーを返した場合、または指定された walker 名がデバッガに認識されない場合、`mdb_walk()` 関数は失敗します。walker 名に重複があった場合、逆引用符 (`) 演算子を使用して名前の有効範囲を指定できます。data パラメタは、呼び出し元にだけ意味を持つ隠された引数です。このパラメタは walk の各ステップで *func* に戻されます。

mdb_pwalk_dcmd()

```
int mdb_pwalk_dcmd(const char *wname, const char *dcname, int argc,
   const mdb_arg_t *argv, uintptr_t addr);
```

wname で指定された walker を使用して *addr* から始まるローカル walk を開始し、各ステップで *argc* および *argv* を指定して、*dcname* で指定された dcmd を起動します。この関数は成功した場合 0 を、エラーの場合 -1 を返します。walker 自体が致命的なエラーを返した場合、指定された walker 名または dcmd 名がデバッガに認識されない場合、あるいは dcmd 自体が walker に DCMD_ABORT または DCMD_USAGE を返した場合、この関数は失敗します。名前の重複があった場合、walker 名と dcmd 名は逆引用符 (`) 演算子を使用して名前の有効範囲を指定できます。mdb_pwalk_dcmd() から起動された場合、dcmd はフラグパラメタに DCMD_LOOP および DCMD_ADDRSPEC ビットを設定し、最初の呼び出しで DCMD_LOOPFIRST が設定されます。

mdb_walk_dcmd()

```
int mdb_walk_dcmd(const char *wname, const char *dcname, int argc,
   const mdb_arg_t *argv);
```

wname で指定された walker を使用してグローバル walk を開始し、各ステップで *argc* および *argv* を指定して、*dcname* で指定された dcmd を起動します。この関数は成功した場合 0 を、エラーの場合 -1 を返します。walker 自体が致命的なエラーを返した場合、指定された walker 名または dcmd 名がデバッガに認識されない場合、あるいは dcmd 自体が walker に DCMD_ABORT または DCMD_USAGE を返した場合、この関数は失敗します。名前の重複があった場合、walker 名と dcmd 名は逆引用符 (`) 演算子を使用して名前の有効範囲を指定できます。mdb_walk_dcmd() から起動された場合、dcmd はフラグパラメタに DCMD_LOOP および DCMD_ADDRSPEC ビットを設定し、最初の呼び出しで DCMD_LOOPFIRST が設定されます。

mdb_call_dcmd()

```
int mdb_call_dcmd(const char *name, uintptr_t addr, uint_t flags,
   int argc, const mdb_arg_t *argv);
```

与えられたパラメタで指定された dcmd 名を起動します。ドット変数が *addr* にリセットされ、*addr*、*flags*、*argc*、および *argv* が dcmd に渡されます。この関数は成功した場合 0 を、エラーの場合 -1 を返します。dcmd が DCMD_ERR、DCMD_ABORT、または DCMD_USAGE を返した場合、あるいは指定された dcmd 名がデバッガに認識

されない場合、この関数は失敗します。名前の重複があった場合、dcmd 名は逆引用符 (`) 演算子を使用して名前の有効範囲を指定できます。

mdb_layered_walk()

```
int mdb_layered_walk(const char *name, mdb_walk_state_t *wsp);
```

wsp で指定された walk を、指定された walker 名を使用して開始された walk の上の層に置きます。名前の重複があった場合、dcmd 名は逆引用符 (`) 演算子を使用して名前の有効範囲を指定できます。階層化された walk を使用すると、他のデータ構造体に組み込まれたデータ構造体に対する walker を簡単に作成することができます。

たとえば、カーネルの各 CPU 構造体に組み込み構造体を指すポインタが含まれているとします。組み込み構造体タイプに対する walker を作成するときに、CPU 構造体を繰り返すコードを複製して各 CPU 構造体の該当するメンバーの参照を解除することもできますが、組み込み構造体の walker を既存の CPU walker の上に重ねることもできます。

`mdb_layered_walk()` 関数は、現在の walk に新規の層を追加するために walker の init ルーチンの中から使用されます。配下の層は `mdb_layered_walk()` の呼び出しの一部として初期化されます。呼び出し元の walk ルーチンは、現在の walk の状態を指すポインタを渡します。この状態を使用して階層化された walk が構築されます。階層化された各 walk は、呼び出し元の walk fini 関数が呼び出された後、クリーンアップされます。複数の層が walk に追加されている場合、呼び出し元の walk step 関数は最初の層から返された各要素を処理した後、次に 2 番目の層へ進み、以降も同様に処理します。

`mdb_layered_walk()` 関数は成功した場合 0 を、エラーの場合 -1 を返します。指定された walker 名がデバッガに認識されない場合、*wsp* ポインタが有効かつアクティブな walk 状態ポインタでない場合、階層化された walker 自体が初期化に失敗した場合、または呼び出し元が自分自身の上に walker を重ねようとした場合、この関数は失敗します。

mdb_add_walker()

```
int mdb_add_walker(const mdb_walker_t *w);
```

新規の walker をデバッガに登録します。walker は、33ページの「dcmd と walker の名前解決」に説明されている名前解決規則に従って、モジュールの名前空間、およびデバッガのグローバルな名前空間に追加されます。この関数は成功した

場合 0 を返しますが、指定された walker 名が既にこのモジュールによって登録済みであったり、walker の構造体 *w* が正しく構築されていなかったりした場合、エラーとして -1 を返します。mdb_walker_t *w* の情報が内部のデバッガ構造体にコピーされるため、呼び出し元では mdb_add_walker() を呼び出した後にこの構造体を再使用または解放できます。

mdb_remove_walker()

```
int mdb_remove_walker(const char *name);
```

指定された *name* の walker を削除します。この関数は成功した場合 0 を、エラーの場合 -1 を返します。walker は現在のモジュールの名前空間から削除されます。walker 名が認識されない場合や、別のモジュールの名前空間だけに登録されている場合、この関数は失敗します。mdb_remove_walker() 関数を使用すると、mdb_add_walker() を使用して動的に追加された walker、またはモジュールのリンク構造の一部として静的に追加された walker を削除することができます。walker 名の有効範囲を指定する演算子は、ここでは使用できません。mdb_remove_walker() の呼び出し元が、別のモジュールからエクスポートされた walker を削除しようとしても無効です。

mdb_vread() および mdb_vwrite()

```
ssize_t mdb_vread(void *buf, size_t nbytes, uintptr_t addr);
ssize_t mdb_vwrite(const void *buf, size_t nbytes, uintptr_t addr);
```

これらの関数は、*addr* パラメタで指定された、所定のターゲットの仮想アドレスからデータを読み取ったり、そのアドレスにデータを書き込んだりするのに使用します。mdb_vread() 関数は成功した場合 *nbytes* を、エラーの場合 -1 を返します。指定されたアドレスからデータの一部しか読み取れなかったためにデータが切り捨てられた場合、-1 が返されます。mdb_vwrite() 関数は成功した場合、実際に書き込まれたバイト数を返し、エラーが発生した場合は -1 を返します。

mdb_pread() および mdb_pwrite()

```
ssize_t mdb_pread(void *buf, size_t nbytes, uint64_t addr);
ssize_t mdb_pwrite(const void *buf, size_t nbytes, uint64_t addr);
```

これらの関数は、addr パラメタで指定された、所定のターゲット物理アドレスからデータを読み取ったり、そのアドレスにデータを書き込んだりするのに使用しま

す。mdb_pread() 関数は成功した場合 *nbytes* を、エラーの場合 -1 を返します。指定されたアドレスからデータの一部しか読み取れなかったためにデータが切り捨てられた場合、-1 が返されます。mdb_pwrite() 関数は成功した場合、実際に書き込まれたバイト数を返し、エラーが発生した場合は -1 を返します。

mdb_readstr()

```
ssize_t mdb_readstr(char *s, size_t nbytes, uintptr_t addr);
```

mdb_readstr() 関数は、ターゲットの仮想アドレス *addr* から始まる NULL で終了する C 文字列を、*s* で指定されたバッファに読み込みます。バッファのサイズは *nbytes* で指定されます。この文字列が長すぎてバッファに収まらない場合、文字列はバッファサイズで切り捨てられ、s[nbytes - 1] に NULL バイトが格納されます。成功した場合、末尾の NULL バイトを含めずに *s* に格納された文字列の長さが返され、失敗した場合はエラーを示す -1 が返されます。

mdb_writestr()

```
ssize_t mdb_writestr(const char *s, uintptr_t addr);
```

mdb_writestr() 関数は、NULL で終了する C 文字列を末尾の NULL バイトも含めて *s* から、ターゲットの仮想アドレス空間の *addr* で指定されたアドレスに書き込みます。成功した場合、末尾の NULL バイトを含めずに実際に書き込まれたバイト数が返され、失敗した場合はエラーを示す -1 が返されます。

mdb_readsym()

```
ssize_t mdb_readsym(void *buf, size_t nbytes, const char *name);
```

読み取りが開始される仮想アドレスが *name* で指定されたシンボルの値から取得される点以外は、mdb_readsym() は mdb_vread() に似ています。その名前でシンボルが見つからなかった場合、または読み取りエラーが発生した場合は -1 が返されます。成功した場合は *nbytes* が返されます。

シンボルの検索の失敗と読み取りの失敗を区別する必要がある場合、呼び出し元ではまずシンボルを別に調べます。一次実行可能ファイルのシンボルテーブルを使用してシンボルが検索されます。シンボルが別のシンボルテーブルに存在する場合、最初に mdb_lookup_by_obj()、次に mdb_vread() の順で適用する必要があります。

mdb_writesym()

```
ssize_t mdb_writesym(const void *buf, size_t nbytes, const char *name);
```

mdb_writesym() は、書き込みが開始される仮想アドレスが name で指定されたシンボルの値から取得される点以外は、mdb_vwrite() と同じです。その名前でシンボルが見つからなかった場合は -1 が返されます。それ以外の場合、成功すると正常に書き込まれたバイト数が返され、エラーが発生すると -1 が返されます。一次実行可能ファイルのシンボルテーブルを使用してシンボルが検索されます。シンボルが別のシンボルテーブルに存在する場合、最初に mdb_lookup_by_obj()、次に mdb_vwrite() の順で適用する必要があります。

mdb_readvar() および mdb_writevar()

```
ssize_t mdb_readvar(void *buf, const char *name);
ssize_t mdb_writevar(const void *buf, const char *name);
```

読み取りが開始される仮想アドレスと読み取るバイト数が *name* で指定されたシンボルの値とサイズから取得される点以外は、mdb_readvar() は mdb_vread() に似ています。その名前でシンボルが見つからなかった場合は -1 が返されます。成功するとシンボルのサイズ、すなわち正常に読み取られたバイト数が返され、エラーが発生すると -1 が返されます。たとえば次のように、この関数はサイズの固定している既知の変数を読み取る場合に有用です。

```
int hz;  /* システムクロックレート */
mdb_readvar(&hz, "hz");
```

シンボルの検索の失敗と読み取りの失敗を区別する必要がある場合、呼び出し元ではまずシンボルを別に調べます。また、ローカルの宣言がターゲットの定義とまったく同じであることを確認するために、呼び出し元では当該のシンボルの定義を注意して調べる必要があります。たとえば、呼び出し元が int を宣言しているのに当該のシンボルが実際には long であったため、デバッガが 64 ビットのカーネルターゲットを調べている場合、mdb_readvar() は 8 バイトを呼び出し元のバッファに戻すため、int に格納される分の後に残る 4 バイトが破壊されてしまいます。

書き込みが開始される仮想アドレスと書き込むバイト数がname で指定されたシンボルの値とサイズから取得される点以外は、mdb_writevar() は mdb_vwrite()と同じです。その名前でシンボルが見つからなかった場合は -1 が返されます。成功すると正常に書き込まれたバイト数が返され、エラーが発生すると -1 が返されます。

どちらの関数も、シンボルの検索では一次実行可能ファイルのシンボルテーブルが使用されます。シンボルが別のシンボルテーブルに存在する場合、最初に

mdb_lookup_by_obj()、次に mdb_vread() または mdb_vwrite() の順で適用する必要があります。

mdb_lookup_by_name() および mdb_lookup_by_obj()

```
int mdb_lookup_by_name(const char *name, GElf_Sym *sym);
int mdb_lookup_by_obj(const char *object, const char *name, GElf_Sym *sym);
```

指定されたシンボル名を検索し、ELF シンボル情報を *sym* の指す GElf_Sym にコピーします。シンボルが見つかった場合、この関数は 0 を返します。それ以外の場合は -1 を返します。*name* パラメタはシンボル名を指定します。*object* パラメタは、デバッガにシンボルを検索する場所を指示します。mdb_lookup_by_name() 関数では、オブジェクトファイルは MDB_OBJ_EXEC にデフォルト設定されます。mdb_lookup_by_obj() では、オブジェクト名は次のどれかになります。

MDB_OBJ_EXEC　実行可能ファイルのシンボルテーブル (.symtab セクション) を検索します。カーネルクラッシュダンプの場合、このテーブルは unix.X ファイルまたは /dev/ksyms のシンボルテーブルに相当します。

MDB_OBJ_RTLD　実行時リンカーのシンボルテーブルを検索します。カーネルクラッシュダンプの場合、このテーブルは krtld モジュールのシンボルテーブルに相当します。

MDB_OBJ_EVERY　すべての既知のシンボルテーブルを検索します。カーネルクラッシュダンプの場合、この中には unix.X ファイルまたは /dev/ksyms の .symtab および .dynsym セクションのほか、モジュール単位のシンボルテーブルが処理されていればそれらのテーブルも含まれます。

object　特定のロードオブジェクト名が明示的に指定されている場合、検索はこのオブジェクトのシンボルテーブルだけに限定されます。オブジェクトは、31ページの「シンボルの名前解決」に説明されているロードオブジェクトのための命名規則に従って命名されます。

mdb_lookup_by_addr()

```
int mdb_lookup_by_addr(uintptr_t addr, uint_t flag, char *buf,
    size_t len, GElf_Sym *sym);
```

指定されたアドレスに対応するシンボルを検索し、ELF シンボル情報を *sym* の指す GElf_Sym に、シンボル名を *buf* で指定された文字配列にコピーします。対応するシンボルが見つかった場合、この関数は 0 を返します。見つからない場合は -1 を返します。

flag パラメタは検索モードを指定するもので、次のどれかになります。

MDB_SYM_FUZZY	現在のシンボルディスタンスの設定に基づいて、あいまい一致検索を実行できます。シンボルディスタンスは、::set –s 組み込みコマンドを使用して制御することができます。シンボルディスタンスが明示的に設定されている場合、すなわち絶対モードの場合、シンボルの値からアドレスまでの距離が絶対シンボルディスタンスを超えなければ、アドレスはシンボルと一致します。スマートモードが有効な場合、すなわちシンボルディスタンス = 0 の場合、アドレスが有効範囲内、すなわちシンボルの値からシンボルの値 + シンボルのサイズまでの範囲であればシンボルと一致します。
MDB_SYM_EXACT	あいまい一致検索を許可しません。シンボル値が指定されたアドレスと厳密に等しい場合だけ、シンボルはアドレスと一致します。

シンボルが一致すると、シンボル名が呼び出し元の提供した *buf* にコピーされます。*len* パラメタはこのバッファの長さをバイト単位で指定します。呼び出し元の *buf* は、少なくとも MDB_SYM_NAMLEN バイト必要です。デバッガはシンボル名をこのバッファにコピーし、後ろに NULL の 1 バイトを追加します。名前の長さがバッファの長さを超えると、シンボル名は切り捨てられますが、末尾には常に NULL の 1 バイトが存在します。

mdb_getopts()

```
int mdb_getopts(int argc, const mdb_arg_t *argv, ...);
```

指定された引数の配列 (*argv*) からオプションとオプションの引数を構文解析し、処理します。*argc* パラメタは引数配列の長さを示します。この関数は各引数を順に処理し、処理できない引数があると停止して、その配列の索引を返します。すべての引数が正常に処理できた場合、*argc* を返します。

argc および *argv* パラメタの後に、mdb_getopts() 関数では、*argv* 配列に入る予定のオプションを記述した可変の引数リストを指定できます。各オプションはオプション文字 (char 引数)、オプションタイプ (uint_t 引数)、および次の表に示すような 1 つまたは 2 つのその他の引数で記述されます。オプション引数のリストの末尾は NULL 引数となっています。タイプは次のどれかです。

MDB_OPT_SETBITS　指定されたビットとフラグワードとの論理和をとります。このオプションは次のパラメタで記述されます。

```
char c, uint_t type, uint_t bits, uint_t *p
```

タイプが MDB_OPT_SETBITS であり、*argv* リストでオプション *c* が検出された場合、デバッガはポインタ *p* の参照する整数と指定ビットの論理和をとります。

MDB_OPT_CLRBITS　指定されたビットをフラグワードから消去します。このオプションは次のパラメタで記述されます。

```
char c, uint_t type, uint_t bits, uint_t *p
```

タイプが MDB_OPT_CLRBITS であり、*argv* リストでオプション *c* が検出された場合、デバッガはポインタ *p* の参照する整数から指定ビットを消去します。

MDB_OPT_STR　文字列引数をとります。このオプションは次のパラメタで記述されます。

```
char c, uint_t type, const char **p
```

タイプが MDB_OPT_STR であり、*argv* リストでオプション *c* が検出された場合、デバッガは *c* の後に続く文字列引数を指すポインタを *p* の参照しているポインタに格納します。

MDB_OPT_UINTPTR　uintptr_t 引数をとります。このオプションは次のパラメタで記述されます。

```
char c, uint_t type, uintptr_t *p
```

タイプが MDB_OPT_UINTPTR であり、*argv* リストでオプション c が検出された場合、デバッガは *c* の後に続く整数引数を *p* の参照している uintptr_t に格納します。

MDB_OPT_UINT64　uint64_t 引数をとります。このオプションは次のパラメタで記述されます。

char c, uint_t type, uint64_t *p

タイプが MDB_OPT_UINT64 であり、*argv* リストでオプション *c* が検出された場合、デバッガは *c* の後に続く整数引数を *p* の参照している uint64_t に格納します。

たとえば、次のソースコードは、dcmd で mdb_getopts() を使用して、*opt_v* 変数を TRUE に設定するブール型オプション "–v" 、および *opt_s* 変数に格納されている文字列引数をとるオプション "–s" をとる方法を示しています。

```
int
dcmd(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
{
        uint_t opt_v = FALSE;
        const char *opt_s = NULL;

        if (mdb_getopts(argc, argv,
            'v', MDB_OPT_SETBITS, TRUE, &opt_v,
            's', MDB_OPT_STR, &opt_s, NULL) != argc)
                return (DCMD_USAGE);

        /* ... */
}
```

また、mdb_getopts() 関数は、呼び出し元に戻る前に無効なオプション文字やオプション引数の欠落を検出すると、自動的に警告メッセージを表示します。引数文字列および *argv* 配列のための記憶領域は、dcmd が完了するとデバッガにより自動的にガベージコレクションに集められます。

mdb_strtoull()

```
u_longlong_t mdb_strtoull(const char *s);
```

指定された文字列 *s* を符号なし long long 表現に変換します。この関数は、mdb_getopts() が適当でない状況において文字列引数を処理し変換します。文字列引数が有効な整数表現に変換できない場合、関数は失敗し、該当するエラー

メッセージが出力され、dcmd は異常終了します。したがって、エラーチェックコードは不要です。文字列には、先頭に有効な指示子 (0i、0I、0o、0O、0t、0T、0x、または 0X) を付けることができますが、付けない場合はデフォルトを使用するものと解釈されます。*s* の中に基底文字として適切でない文字があったり、整数のオーバーフローが発生したりすると、この関数は失敗し、dcmd は異常終了します。

mdb_alloc()、mdb_zalloc() および mdb_free()

```
void *mdb_alloc(size_t size, uint_t flags);
void *mdb_zalloc(size_t size, uint_t flags);
void mdb_free(void *buf, size_t size);
```

mdb_alloc() は *size* バイトのデバッガメモリーを割り当て、割り当てたメモリーにポインタを返します。割り当て済みメモリーは、どのような C 構造体でも保持できるように、少なくともダブルワードが割り当てられます。それ以上の割り当てはできません。*flags* パラメタは、次の 1 つ以上の値のビット単位の論理和となります。

UM_NOSLEEP
: 要求を満たすだけの十分なメモリーがすぐに使用可能でない場合、失敗を示す NULL が返されます。呼び出し元は NULL が返されたかどうかをチェックして、NULL の場合には適切に対処する必要があります。

UM_SLEEP
: 要求を満たすだけの十分なメモリーがすぐに使用可能でない場合、要求を満たすことができるまでの間、スリープ (休眠) します。したがって、UM_SLEEP 割り当ての場合、成功することが保証されています。呼び出し元で NULL 戻り値をチェックする必要はありません。

UM_GC
: このデバッガコマンドの終わりに自動的に割り当てのガベージコレクションを行います。割り当ての解除はデバッガによって自動的に行われるので、呼び出し元はこのブロックにおいてそれ以降 mdb_free() を呼び出すことはできません。dcmd がユーザーによって中断された場合、デバッガが不要メモリーのガベージコレクションを実行できるように、dcmd の中からメモリーの割り当てを行うときは、必ず UM_GC を使用する必要があります。

mdb_zalloc() は mdb_alloc() と似ていますが、呼び出し元に戻る前に割り当てたメモリーにはゼロが入ります。mdb_alloc() から戻されるメモリーの初期内容

は、保証されません。mdb_free() は、UM_GC で割り当てられたメモリー以外の、以前に割り当て済みのメモリーを解放するのに使用します。バッファアドレスとサイズは元の割り当てと正確に一致している必要があります。mdb_free() を使用して割り当ての一部だけを解放することはできません。また、二度以上割り当てを解放することもできません。ゼロバイトの割り当てでは、常に NULL が返されます。サイズがゼロの NULL ポインタの解放は、常に成功します。

mdb_printf()

```
void mdb_printf(const char *format, ...);
```

指定された書式文字列と引数を使用して、書式付き出力を書き出します。警告とエラーメッセージを除いて、モジュール作成者はあらゆる出力に対して mdb_printf() を使用する必要があります。この関数は必要に応じて自動的に組み込み出力ページャをトリガーします。mdb_printf() 関数は printf(3C) に似ていますが、次のような例外があります。ワイド文字列に対して %C、%S、および %ws 指示子はサポートされていない、%f 浮動小数点形式がサポートされていない、代替ダブルフォーマットに対する %e、%E、%g、および%G 指示子では、単一形式の出力だけが生成される、書式 %.n の精度の指定はサポートされていない。サポートされている指示子のリストを次に示します。

フラグ指示子

%# 書式文字列の中に # 記号があった場合、与えられたフォーマットの代替書式を選択します。すべてのフォーマットに代替書式があるとは限りません。代替書式はフォーマットによって異なります。代替書式の詳細については、以降のフォーマットの説明を参照してください。

%+ 符号付きの値を出力する場合、常に符号として '+' または '-' の接頭辞を表示します。%+ を指定しない場合、正の値には符号の接頭辞が付かず、負の値には先頭に '-' の接頭辞が付けられます。

%- 指定されたフィールド幅の中で出力を左詰めにします。出力の幅が指定されたフィールド幅より小さい場合、右側には空白文字が入ります。%- を指定しない場合、デフォルトの設定では値は右詰めになります。

`%0` 出力が右詰めで出力幅が指定されたフィールドの幅より小さい場合、出力フィールドがゼロで埋められます。`%0` を指定しない場合、右詰めにした値の前の残りのフィールドには空白文字が入ります。

フィールド幅の指示子

`%n` フィールド幅は指定された 10 進数値に設定されます。

`%?` フィールド幅は 16 進数のポインタ値の最大幅に設定されます。この値は ILP32 環境では 8、LP64 環境では 16 です。

`%*` フィールド幅は引数リストの現在の位置で指定された値に設定されます。この値は `int` であるとみなされます。64 ビットのコンパイル環境では、`long` 値を `int` にキャストしなければならない場合があります。

整数指示子

`%h` `short` 型の整数値が出力されます。

`%l` `long` 型の整数値が出力されます。

`%ll` `long long` 型の整数値が出力されます。

端末属性指示子

デバッガの標準出力が端末であり、terminfo データベースから端末属性を変更できる場合、次の端末エスケープコンストラクトが使用できます。

`%<n>` *n* に対応する端末属性を有効にします。`%<>` の各インスタンスごとに、1 つの属性だけを有効にできます。

`%</n>` *n* に対応する端末属性を無効にします。反転表示、選択不可テキスト、およびボールドテキストの場合、これらの属性を無効にする端末コードは同じである可能性があります。したがって、これらの属性を互いに独立して無効にはできない場合があります。

端末情報が使用できない場合、各端末属性コンストラクトは mdb_printf() で無視されます。端末属性については、terminfo(4) のマニュアルページを参照してください。使用可能な terminfo 属性は次のとおりです。

a	代替文字セット
b	ボールドテキスト
d	選択不可テキスト
r	反転表示
s	強調表示機能
u	下線

書式指示子

%%	'%' 記号が出力されます。
%a	アドレスが記号形式で出力されます。%a に関連付けられている値の最小サイズは uintptr_t ですが、%la の指定は必須ではありません。アドレスからシンボルへの変換が有効な場合、デバッガはアドレスを現在の出力の基数でのシンボル名とそれに続くオフセットに変換して、この文字列を出力しようとします。変換が有効でない場合、アドレス値はデフォルトの出力の基数で出力されます。%#a を使用した場合、代替書式によって出力に ':' 接尾辞が付加されます。
%A	この書式は %a と同じですが、アドレスがシンボル名とオフセットに変換できない場合は何も出力されない点が異なっています。%#A を使用した場合、アドレス変換が失敗したとき、代替書式によって '?' が出力されます。
%b	ビットフィールドを記号書式で復号化し、出力します。この指示子は 2 つの連続する引数をとります。この 2 つの引数はビットフィールド値 (%b に対する int、%lb に対する long など) および mdb_bitmask_t 構造体の配列を指すポインタです。

```
typedef struct mdb_bitmask {
  const char *bm_name;       /* 出力する文字列名 */
  u_longlong_t bm_mask;      /* ビットのマスク */
  u_longlong_t bm_bits;      /* 値とマスクの結果 */
```

```
} mdb_bitmask_t;
```

配列の末尾は bm_name フィールドが NULL に設定されている構造体でなければなりません。%b を使用した場合、デバッガは値の引数を読み取り、各 mdb_bitmask 構造体を繰り返して、次の条件をチェックします。

```
(value & bitmask->bm_mask) == bitmask->bm_bits
```

この式が真の場合、bm_name 文字列が出力されます。各文字列はコンマで区切って出力されます。次の例は、%b を使用して kthread_t の t_flag フィールドを復号化する方法を示しています。

```
const mdb_bitmask_t t_flag_bits[] = {
  { "T_INTR_THREAD", T_INTR_THREAD, T_INTR_THREAD },
  { "T_WAKEABLE", T_WAKEABLE, T_WAKEABLE },
  { "T_TOMASK", T_TOMASK, T_TOMASK },
  { "T_TALLOCSTK", T_TALLOCSTK, T_TALLOCSTK },
   ...
  { NULL, 0, 0 }
};

void
thr_dump(kthread_t *t)
{
  mdb_printf("t_flag = <%hb>\n", t->t_flag, t_flag_bits);

  ...
}
```

t_flag が 0x000a に設定されている場合、この関数によって次のように出力されます。

```
t_flag = <T_WAKEABLE,T_TALLOCSTK>
```

%c　指定された整数を ASCII 文字として出力します。

%d　指定された整数を符号付き 10 進数値として出力します。%i と同じです。

%e　指定された倍精度数を浮動小数点形式 [+/-]d.ddddddde[+/-]dd で出力します。小数点の前が 1 桁、小数点以下が 7 桁で、指数の後は少なくとも 2 桁です。

%E　指定された倍精度数を %e と同じ規則を使用して出力しますが、指数文字として 'e' ではなく 'E' を使用する点が異なっています。

`%g` 指定された倍精度数を `%e` と同じ浮動小数点形式で出力しますが、16 桁を使用します。`%llg` を指定した場合、引数の型は 4 倍精度浮動小数点数の `long double` となります。

`%G` 指定された倍精度数を `%g` と同じ規則を使用して出力しますが、指数文字として `'e'` ではなく `'E'` を使用する点が異なっています。

`%i` 指定された整数を符号付き 10 進数値として出力します。`%d` と同じです。

`%I` 指定された 32 ビット符号なし整数をドット付き 10 進形式のインターネット IPv4 アドレスとして出力します。たとえば、16 進数値の `0xffffffff` は `255.255.255.255` として出力されます。

`%m` 空白のマージンを印刷します。フィールドを指定しないと、デフォルトの出力マージン幅が使用されます。フィールド幅を指定すると、フィールド幅によって出力される空白の文字数が決定されます。

`%o` 指定された整数を符号なし 8 進数値として出力します。`%#o` を使用した場合、代替書式によって出力の先頭に `'0'` が付けられます。

`%p` 指定されたポインタ (`void *`) を 16 進数値として出力します。

`%q` 指定された整数を符号付き 8 進数値として出力します。`%#o` を使用した場合、代替書式によって出力の先頭に `'0'` が付けられます。

`%r` 指定された整数を現在の出力の基数での符号なし値として出力します。ユーザーは `$d` dcmd を使用して出力の基数を変更することができます。`%#r` を指定すると、代替書式によって値の先頭に該当する基底接頭辞が付けられます。2 進数の場合 `'0i'`、8 進数の場合`'0o'`、10 進数の場合 `'0t'`、16 進数の場合 `'0x'`。

`%R` 指定された整数を現在の出力の基数で符号付きの値として出力します。`%#R` を指定すると、代替書式によって値の先頭に該当する基底接頭辞が付けられます。

`%s` 指定された文字列 (`char *`) を出力します。文字列のポインタが NULL の場合、文字列 `'<NULL>'` が出力されます。

`%t` 1 つまたは複数のタブストップ分出力され、幅を指定しないと次のタブストップまで出力され、幅を指定するとフィールド幅によって出力されるタブストップの数が決定されます。

`%T` カラムをフィールド幅の倍数分出力します。フィールド幅を指定しないと、何の処理も実行されません。現在出力されているカラムがフィールド幅の倍数でない場合、空白が付加されてカラムが出力されます。

`%u` 指定された整数を符号なし 10 進数値として出力します。

`%x` 指定された整数を 16 進数値として出力します。10 から 15 までの値を表す数字として、a から f までの文字を使用します。`%#X` を指定すると、代替書式によって値の先頭に '0x' が付けられます。

`%X` 指定された整数を 16 進数値として出力します。10 から 15 までの値を表す数字として、A から F までの文字を使用します。`%#x` を指定すると、代替書式によって値の先頭に '0X' が付けられます。

`%Y` 指定された `time_t` を文字列 'year month day HH:MM:SS' として出力します。

mdb_snprintf()

```
size_t mdb_snprintf(char *buf, size_t len, const char *format, ...);
```

指定された書式文字列と引数に基づいて書式付き文字列を作成し、作成した文字列を指定された *buf* に格納します。`mdb_snprintf()` 関数は `mdb_printf()` 関数と同じ書式指示子と引数をとります。*len* パラメタは *buf* のサイズをバイト単位で指定します。フォーマットされた *len* - 1 バイト以下のバイトが *buf* に格納されます。`mdb_snprintf()` では、常に *buf* の末尾は NULL 1 バイトで終了します。この関数は、末尾の NULL のバイトを除外した、完全な書式付き文字列に必要なバイト数を返します。*buf* パラメタが NULL で *len* がゼロに設定されている場合、*buf* には何も格納されず、完全な書式付き文字列に必要なバイト数が返されます。この方法を使用して、動的メモリー割り当て用のバッファの適切なサイズが決定されます。

mdb_warn()

```
void mdb_warn(const char *format, ...);
```

エラーまたは警告メッセージを標準エラーに出力します。mdb_warn() 関数では、書式文字列と mdb_printf() で掲げられているすべての指示子を含む可変の引数リストを指定することができます。ただし、mdb_warn() の出力が標準エラーに送られる場合は、バッファには格納されず、出力ページャを通して送信されたり、dcmd パイプラインの一部として処理されることはありません。すべてのエラーメッセージには、自動的に先頭に "mdb: " という接頭辞が付けられます。

さらに、*format* パラメタには復帰改行 (\n) 文字は含まれず、書式文字列の先頭には暗黙的に文字列 ": %s\n" が付けられます。 ここで、%s は、モジュール API 関数が最後に記録したエラーに対応するエラーメッセージ文字列に置換されます。たとえば、次のソースコードの場合を考えます。

```
if (mdb_lookup_by_name("no_such_symbol", &sym) == -1)
    mdb_warn("lookup_by_name failed");
```

この場合、次のような出力が生成されます。

```
mdb: lookup_by_name failed: unknown symbol name
```

mdb_flush()

```
void mdb_flush(void);
```

現在バッファ化されているすべての出力をフラッシュします。通常、mdb の標準出力はラインバッファに格納されます。mdb_printf() で生成された出力は、復帰改行文字を見つけるまで、あるいは現在の dcmd の終わりまで、端末または他の標準出力の出力先にフラッシュされません。しかし、状況によっては、復帰改行を出力する前に標準出力を明示的にフラッシュする必要がある場合もあります。そのような場合にこの mdb_flush() 関数が使用できます。

mdb_one_bit()

```
const char *mdb_one_bit(int width, int bit, int on);
```

mdb_one_bit() 関数を使用して、関連のある 1 つのビットをオンまたはオフにして、ビットフィールドを図式化して出力することができます。この関数は snoop(1M) -v を使用して出力する場合と同様に、ビットフィールドを詳細に表示するのに有用です。たとえば、次のソースコードの場合を考えます。

```
#define FLAG_BUSY       0x1

uint_t flags;

/* ... */
```

```
mdb_printf("%s = BUSY\n", mdb_one_bit(8, 0, flags & FLAG_BUSY));
```

次のような出力が得られます。

```
.... ...1 = BUSY
```

4 ビットごとに空白で区切られ、ビットフィールドの各ビットがピリオド (.) として出力されます。*on* パラメタの設定に従って、関連のあるビットは 1 または 0 で出力されます。ビットフィールドの合計の幅は width パラメタによりビット単位で指定し、関連のあるビットの位置は *bit* パラメタで指定します。ビットにはゼロから番号が付けられます。この関数はフォーマットされたビット表現を含む、適切なサイズの、NULL で終了する文字列を返します。現在の dcmd が完了すると、不要な文字列は自動的に回収されます。

mdb_inval_bits()

```
const char *mdb_inval_bits(int width, int start, int stop);
```

mdb_inval_bits() 関数は mdb_one_bit() と共に使用して、ビットフィールドを図式化して出力します。この関数は 、該当するビット位置に 'x' を表示することによって、そのビットを無効または予約済みとしてマーク付けします。ビットフィールドの各ビットはピリオド (.) として表現されますが、start および stop パラメタで指定された範囲のビット位置にあるビットは対象外です。ビットにはゼロから番号が付けられます。たとえば、次のソースコードの場合を考えます。

```
mdb_printf("%s = reserved\n", mdb_inval_bits(8, 7, 7));
```

この場合、次のような出力が得られます。

```
x... .... = reserved
```

この関数はフォーマットされたビット表現を含む、適切なサイズの、NULL で終了する文字列を返します。現在の dcmd が完了すると、不要な文字列は自動的に回収されます。

mdb_inc_indent() および mdb_dec_indent()

```
ulong_t mdb_inc_indent(ulong_t n);
ulong_t mdb_dec_indent(ulong_t n);
```

これらの関数は、1 行出力する前に MDB によって空白で自動インデントされるカラム数を増減させます。デルタのサイズはカラム数、*n* で指定します。どちらの関数も

前の絶対インデント値を返します。ゼロより小さい値にインデントを設定しようとしても無効です。どちらかの関数を呼び出した後に、mdb_printf() を呼び出すと、適切にインデントされています。dcmd が完了するか、ユーザーによって強制終了された場合、デバッガによってインデントは自動的にデフォルトの設定に戻ります。

mdb_eval()

```
int mdb_eval(const char *s);
```

指定されたコマンド文字列 *s* を、デバッガによって標準入力から読み取られたものとして、評価し実行します。この関数は成功すると 0 を返し、エラーが発生すると -1 を返します。コマンド文字列に構文エラーがあったり、mdb_eval() によって実行されたコマンド文字列がユーザーからのページャまたは割り込みの発生によって強制終了されたりすると、この関数は失敗します。

mdb_set_dot() および mdb_get_dot()

```
void mdb_set_dot(uintmax_t dot);
uintmax_t mdb_get_dot(void);
```

ドット (“.” 変数) の現在の値を設定するか、または取得します。モジュールを開発する上でドットの位置を再設定する必要があるのは、たとえば dcmd が前回読み取ったアドレスに続くアドレスを参照したままにしておく場合などです。

mdb_get_pipe()

```
void mdb_get_pipe(mdb_pipe_t *p);
```

現在の dcmd に対するパイプライン入力バッファの内容を取り出します。mdb_get_pipe() 関数は、dcmd で実行されますがパイプ入力要素ごとにデバッガから繰り返し呼び出されるのではなく、パイプ入力された要素全体を一度に呼び出して、一度だけ実行されます。mdb_get_pipe() がいったん呼び出されると、その dcmd は現在のコマンドの一部として再び起動されることはありません。これは、たとえば入力値のセットをソートする dcmd を作成するときに使用できます。

不要になったパイプの内容は、dcmd が完了すると配列の中に回収されます。この配列のポインタは p->pipe_data に格納されます。配列の長さは p->pipe_len に格納されます。dcmd がパイプラインの右側で実行されなかった場合、すなわち

フラグパラメタで DCMD_PIPE フラグが設定されなかった場合、p->pipe_data は NULL に設定され、p->pipe_len はゼロに設定されます。

mdb_set_pipe()

```
void mdb_set_pipe(const mdb_pipe_t *p);
```

パイプラインの出力バッファをパイプ構造体 *p* によって記述された内容に設定します。パイプの値は配列 p->pipe_data に置かれ、配列の長さは p->pipe_len に格納されます。デバッガはこの情報の独自のコピーを作成するため、呼び出し元で必要に応じて p->pipe_data を解放する必要があります。パイプライン出力バッファが以前に空でなかった場合、その内容は新しい配列に格納されます。dcmd がパイプラインの左側で実行されなかった場合、すなわちフラグパラメタで DCMD_PIPE_OUT フラグが設定されなかった場合、この関数は無効です。

mdb_get_xdata()

```
ssize_t mdb_get_xdata(const char *name, void *buf, size_t nbytes);
```

name で指定されたターゲットの外部データバッファの内容を、*buf* で指定されたバッファの中に読み取ります。*buf* のサイズは *nbytes* パラメタで指定します。*nbytes* を超えるバイト数は呼び出し元のバッファにコピーされません。成功すると読み取った合計バイト数が返され、エラーが発生すると -1 が返されます。呼び出し元が、指定した特定のバッファのサイズを決定する必要がある場合は、*buf* を NULL に、*nbytes* をゼロに指定します。この場合、mdb_get_xdata() はバッファの合計サイズをバイト単位で返しますが、データは読み取りません。外部データバッファを使用することで、モジュールの作成者は、モジュール API を通して他の方法ではアクセスできないターゲットデータにアクセスすることができます。現在のターゲットによってエクスポートされた指定されたバッファのセットは、::xdata 組み込み dcmd を使用して参照できます。

その他の関数

さらに、モジュールの作成者は、次の string(3C) および bstring(3C) 関数も使用できます。これらの関数は、Solaris のマニュアルページに掲載されている関数と同じ意味を持っています。

strcat()　　strcpy()　　strncpy()

```
strchr()        strrchr()           strcmp()
strncmp()       strcasecmp()        strncasecmp()
strlen()        bcmp()              bcopy()
bzero()         bsearch()           qsort()
```

付録 **A**

オプション

この付録では、MDB コマンド行オプションについて説明します。

コマンド行オプションの概要

次のオプションがサポートされています。

`–A`
: mdb モジュールの自動読み込みを無効にします。デフォルトでは、mdb は、ユーザープロセスまたはコアファイルのアクティブな共用ライブラリに対応しているデバッガモジュール、または稼動中のオペレーティングシステムかオペレーティングシステムのクラッシュダンプにある読み込み済みのカーネルモジュールに対応しているデバッガモジュールを読み込もうとします。

`–F`
: 必要に応じて、指定されたユーザープロセスに強制的に接続します。デフォルトでは、mdb は、すでに `truss(1)` など別のデバッグ用ツールの制御下にあるユーザープロセスへの接続を拒否します。`–F` オプションを指定すると、mdb はこれらのプロセスに接続します。こうすることで、mdb とプロセスを制御しようとしている他のツールとの間で本来は行われない対話処理が行われます。

`-I`

マクロファイルを検出するためのデフォルトのパスを設定します。マクロファイルは、`$<` または `$<<` dcmd を使用して読み取ります。このときのパスは、一連のディレクトリ名をコロン (`:`) 文字で区切ったものです。`-I` include パスと `-L` library パス (以降を参照) には、次のトークンを含めることができます。

%i 現在の命令セットアーキテクチャ (ISA) の名前 (sparc、sparcv9、または i386) まで拡大します。

%o 変更対象のパスの古い値まで拡大します。これは、既存のパスの前または後ろにディレクトリを追加するときに有用です。

%p 現在のプラットフォーム文字列 (`uname -i` またはプロセスのコアファイルあるいはクラッシュダンプに格納されているプラットフォーム文字列) まで拡大します。

%r ルートディレクトリのパス名まで拡大します。`-R` オプションを使用すると、代替ルートディレクトリを指定できます。`-R` オプションを指定しないと、ルートディレクトリは mdb 実行可能ファイル自体へのパスから動的に決定されます。たとえば、`/bin/mdb` を実行した場合、ルートディレクトリは `/` です。`/net/hostname/bin/mdb` を実行した場合、ルートディレクトリは `/net/hostname` となります。

%t 現在のターゲット名まで拡大します。これはリテラル文字列 'proc' (ユーザープロセスまたはユーザープロセスのコアファイル)、あるいは 'kvm' (カーネルクラッシュダンプまたは稼動中のオペレーティングシステム) のどちらかです。

32 ビットの mdb に対するデフォルトのインクルードパスは、次のとおりです。

```
%r/usr/platform/%p/lib/adb:%r/usr/lib/adb
```

64 ビットの mdb に対するデフォルトのインクルードパスは、次のとおりです。

```
%r/usr/platform/%p/lib/adb/%i:%r/usr/lib/adb/%i
```

–k 強制的にカーネルデバッギングモードにします。デフォルトでは、mdb は、オブジェクトとコアファイルのオペランドがユーザーの実行可能ファイルとコアダンプを参照しているのか、または 1 組のオペレーティングシステムのクラッシュダンプファイルを参照しているのかを判断しようとします。–k オプションを指定すると、mdb は、これらのファイルがオペレーティングシステムのクラッシュダンプファイルであるとみなします。オブジェクトまたはコアオペランドを指定せずに –k オプションを指定すると、mdb は、オブジェクトファイルを /dev/ksyms に、コアファイルを /dev/kmem にデフォルト設定します。/dev/kmem にアクセスできるのはグループ sys だけです。

–L デバッガモジュールを検索するためのデフォルトのパスを設定します。モジュールは起動時に自動的に読み込まれるか、または ::load dcmd を使用して読み込まれます。このときのパスは一連のディレクトリ名をコロン (:) 文字で区切ったものです。–L ライブラリパスには、上記の –I オプションで示したトークンも含めることができます。

–m カーネルモジュールシンボルのデマンドローディングを無効にします。デフォルトでは、mdb は読み込まれたカーネルモジュールのリストを処理し、モジュールごとにシンボルテーブルのデマンドローディングを実行します。–m オプションを指定すると、mdb はカーネルモジュールのリストを処理したり、モジュールごとにシンボルテーブルを提供したりしなくなります。したがって、アクティブなカーネルモジュールに対応する mdb モジュールは起動時に読み込まれません。

–M

すべてのカーネルモジュールシンボルを事前に読み込みます。 デフォルトでは、mdb はカーネルモジュールシンボルのデマンドローディングを実行します。アドレスがそのモジュールのテキストであるとき、またはデータセクションが参照されているとき、モジュールのシンボルテーブルが完全に読み取られます。–M オプションを指定すると、mdb は起動時にすべてのカーネルモジュールのシンボルテーブルを完全に読み込みます。

–o *option*

指定したデバッガオプションを有効にします。 +o 形式のオプションを使用した場合は、指定したオプションが無効になります。以下に掲載しているものを除いて、各オプションともデフォルトでは無効になっています。mdb は次のオプション引数を認識します。

adb

ストリクタ adb(1) の互換性を有効にします。プロンプトは空の文字列に設定され、出力ページャなどの多数の mdb 機能が無効になります。

follow_child

fork(2) システムコールが発生すると、デバッガは子プロセスを追跡します。デフォルトでは、デバッガはオリジナルのターゲットプロセス (親プロセス) に接続されています。

ignoreeof

端末に EOF シーケンス (^D) が入力されても、デバッガは終了しません。終了するには ::quit dcmd を使用する必要があります。

pager

出力ページャが有効になります (デフォルト設定)。

repeatlast

NEWLINE がコマンドとして端末に入力された場合、mdb は前のコマンドを現在のドットの値で繰り返します。–o adb を指定した場合、このオプションも自動的に指定されています。

–p *pid*

指定されたプロセス ID に接続し、そのプロセスを停止します。mdb は、/proc/*pid*/object/a.out ファイルを実行可能ファイルのパス名として使用します。

–P

コマンドプロンプトを設定します。デフォルトのプロンプトは '> 'です。

–R

パス名を拡張するためのルートディレクトリを設定します。デフォルトでは、ルートディレクトリは mdb 実行可能ファイル自体のパス名から導かれます。ルートディレクトリは、パス名の拡大の際に %r トークンと置き換えられます。

–s *distance*

アドレスからシンボル名への変換用のシンボルマッチングディスタンスを、指定した *distance* に設定します。デフォルトでは、mdb はこの距離をゼロに設定し、スマートマッチングモードを有効にします。ELF シンボルテーブルのエントリには値 V とサイズ S が含まれ、関数またはデータオブジェクトのサイズがバイト単位で示されます。スマートモードでは、mdb は、A が [V, V + S)の範囲にある場合、アドレス A と与えられたシンボルとを一致させます。ゼロ以外の距離を指定した場合も同じアルゴリズムが使用されますが、式に S を指定した場合、常に絶対距離が指定され、シンボルのサイズは無視されます。

–S

ユーザーの ~/.mdbrc ファイルの処理を抑制します。デフォルトでは、mdb は、$HOME で定義されているユーザーのホームディレクトリにマクロファイル .mdbrc があれば、それを読み取って処理します。–S オプションを指定すると、このファイルは読み取られません。

–u

強制的にユーザーデバッギングモードにします。デフォルトでは、mdb は、オブジェクトとコアファイルのオペランドがユーザーの実行可能ファイルとコアダンプを参照しているのか、または 1 組のオペレーティングシステムのクラッシュダンプファイルを参照しているのかを判断しようとします。–u オプションを使用すると、mdb は、これらのファイルがオペレーティングシステムのクラッシュダンプファイルではないとみなします。

`-V`
: 逆アセンブラのバージョンを設定します。デフォルトでは、mdb は、デバッグターゲットに対する適切な逆アセンブラのバージョンを判断しようとします。-V オプションを使用すると、逆アセンブラを明示的に設定できます。::disasms dcmd によって、使用可能な逆アセンブラのバージョンが一覧表示されます。

`-w`
: 指定したオブジェクトとコアファイルを書き込み用に開きます。

`-y`
: tty モードに対する明示的な端末初期化シーケンスを送信します。cmdtool(1) など、端末によっては、tty モードに切り替えるのに明示的な初期化シーケンスが必要です。この初期化シーケンスがないと、mdb からスタンドアウトモードなどの端末機能を使用できない場合があります。

付録 **B**

crash からの移行

従来の crash(1M) ユーティリティから mdb(1) への移行は比較的簡単です。MDB では、crash コマンドの多くを提供しています。MDB での拡張機能および対話機能が追加されたことによって、プログラマは現在のコマンドセットでは調べることのできないシステムの側面を調べることができるようになりました。

この付録では、crash(1M) のいくつかの機能について簡単に説明し、それに相当する MDB の機能を紹介します。

コマンド行オプション

crash –d、–n、および –w コマンド行オプションは、mdb ではサポートされていません。crash ダンプファイルとネームリスト (シンボルテーブルファイル) は mdb への引数として、ネームリスト、クラッシュダンプファイルの順に指定します。稼動しているカーネルを調べるには、追加の引数を付けずに mdb –k オプションを指定します。ユーザーが mdb の出力先をファイルまたは別の出力先に変更するには、コマンド行で mdb を起動した後、適切なシェルリダイレクション演算子を使用するか、::log 組み込み dcmd を使用する必要があります。

MDB での入力

一般的に、MDB における入力は関数名 (MDB では dcmd 名) の前に "::" を付けること以外は crash と似ています。一部の MDB dcmd では、dcmd 名の前に式の引数を指定できます。crash と同様、dcmd 名の後に続けて文字列オプションを指定できます。関数呼び出しの後に！文字を指定すると、MDB は指定されたシェルパイプラインへのパイプラインも作成します。MDB で指定されたすべての即値は、デフォルトでは 16 進数で解釈されます。表 B–1 に示すように、即値に対する基数の指示子は crash と MDB とでは異なっています。

表 B–1 基数指示子

crash	**mdb**	基数
0x	0x	16 進数
0d	0t	10 進数
0b	0i	2 進数

多くの crash コマンドでは、スロット番号またはスロット範囲を入力引数としてとることができました。Solaris オペレーティング環境はスロットという点では構成されなくなったので、MDB dcmd はスロット番号の処理をサポートしていません。

関数

crash 関数	**mdb dcmd**	コメント
?	::dcmds	使用可能な関数を一覧表示する
!command	!command	シェルへのエスケープおよびコマンドを実行する

crash 関数	**mdb dcmd**	コメント
base	=	mdb では、= 書式文字を使用して、左側の式の値を既知の書式に変換できる。8 進数、10 進数、および 16 進数のための書式が用意されている
callout	::callout	コールアウトテーブルを出力する
class	::class	スケジューリングクラスを出力する
cpu	::cpuinfo	システム CPU に振り分けられたスレッドに関する情報を出力する。特定の CPU 構造体の内容が必要な場合、ユーザーは `$<cpu` マクロを mdb の CPU アドレスに適用する必要がある
help	::help	名前を指定した dcmd、または一般的なヘルプ情報を出力する
kfp	::regs	mdb ::regs dcmd は、現在のスタックフレームポインタを含む、完全なカーネルレジスタセットを表示する。$C dcmd を使用すると、フレームポインタを含めてスタックのバックトレースを表示できる
kmalog	::kmalog	カーネルメモリーアロケータのトランザクションログのイベントを表示する
kmastat	::kmastat	カーネルメモリーアロケータのトランザクションログを出力する
kmausers	::kmausers	カーネルメモリーアロケータで現在のメモリー割り当て量が中程度あるいは大きいユーザーに関する情報を出力する
mount	::fsinfo	マウントされているファイルシステムに関する情報を出力する
nm	::nm	シンボルのタイプと値に関する情報を出力する
od	::dump	指定された領域の書式付きメモリーダンプを出力する。mdb では、::dump により領域が ASCII と 16 進数の混合形式で表示される
proc	::ps	アクティブなプロセスの表を出力する
quit	::quit	デバッガを終了する

crash 関数	**mdb dcmd**	コメント
rd	::dump	指定された領域の書式付きメモリーダンプを出力する。mdb では、::dump により領域が ASCII と 16 進数の混合形式で表示される
redirect	::log	mdb では、入力に対する出力と出力は、::log を使用してグローバルに出力先をログファイルに変更できる
search	::kgrep	mdb では、::kgrep dcmd を使用してカーネルのアドレス空間で特定の値を検索できる。パターンマッチング組み込み dcmd を使用すると、物理、仮想、またはオブジェクトファイルのアドレス空間でパターンを検索することもできる
stack	::stack	::stack を使用すると、現在のスタックトレースを取得できる。特定のカーネルスレッドのスタックトレースは、::findstack dcmd を使用して判定できる。現在のスタックのメモリーダンプは、/ または ::dump dcmd と現在のスタックポインタを使用して取得できる。`$<stackregs` マクロをスタックポインタに適用すると、フレームごとに保存されたレジスタ値を取得できる
status	::status	デバッガが調査しているシステムまたはダンプに関する状態情報を表示する
stream	::stream	mdb ::stream dcmd を使用すると、特定のカーネル STREAM の構造体をフォーマットし表示できる。アクティブな STREAM 構造体が必要な場合、ユーザーは mdb で ::walk stream_head_cache を実行し、その結果得られたアドレスを適切なフォーマット dcmd またはマクロにパイプする必要がある
strstat	::kmastat	::kmastat dcmd は、`strstat()` 関数によってレポートされた情報のスーパーセットを表示する
trace	::stack	現在のスタックトレースは、::stack を使用して取得できる。特定のカーネルスレッドのスタックトレースは、::findstack dcmd を使用して決定できる。現在のスタックのメモリーダンプは、/ または ::dump dcmd と現在のスタックポインタを使用して取得できる。`$<stackregs` マクロをスタックポインタに適用すると、フレームごとに保存されたレジスタ値を取得できる
var	$<v	グローバルな `var` 構造体の調整可能なシステムパラメタを出力する

crash 関数	**mdb dcmd**	コメント
vfs	::fsinfo	マウントされているファイルシステムに関する情報を出力する
vtop	::vtop	指定された仮想アドレスの物理的なアドレス変換を出力する

索引

F

K

M

Q

R

S

U

W

い

え

か

け