Solaris モジューラデバッガ

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

Part No: 819–0393–11
2007 年 6 月

原典: Solaris Modular Debugger Guide

Part No: 816-5041-11

Revision A

070617@17466

目次

はじめに

モジューラデバッガ (MDB) は、Solaris™ オペレーティングシステム用の拡張性にすぐれた汎用デバッグツールです。『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および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。

注 – このリリースでは、SPARC® および x86 系列のプロセッサアーキテクチャー (UltraSPARC®、SPARC64、AMD64、Pentium、Xeon EM64T) をサポートします。サポートされるシステムについては、Solaris OS Hardware Compatibility Lists (http://www.sun.com/bigadmin/hcl) を参照してください。本書では、プラットフォームにより実装が異なる場合は、それを特記します。

本書では、「x86」という用語は AMD64 あるいは Intel Xeon/Pentium 製品系列と互換性のあるプロセッサを使用して製造された 32 ビットおよび 64 ビットシステムを意味します。サポートされるシステムについては、Solaris OS Hardware Compatibility Lists を参照してください。

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、だれ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います。

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 8 章と第 9 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 のアーキテクチャーを説明し、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、MDB の対話型コマンド行編集機能と出力ページャー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、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 6 章では、実行中のプログラムの実行を制御するための MDB の機能について説明します。この章の対象読者は、アプリケーション開発者とデバイスドライバ開発者です。実行制御機能は、システム管理者にも役立つ場合があります。

第 7 章では、`kmdb` 専用の実行中のオペレーティングシステムカーネルの実行を制御するための MDB 機能について説明します。この章の対象読者は、オペレーティングシステムカーネルの開発者とデバイスドライバ開発者です。

第 8 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 9 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 10 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、デバッガの使用に関連する警告と注意が記載されています。

付録 C には、`adb` 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。`adb` コマンドは、`mdb` によって実装されます。

付録 D には、`crash` 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。`crash` コマンドは Solaris にはもう存在しません。

関連マニュアルと論文

次に、参考となる関連マニュアルと論文を記載します。

- Vahalia、Uresh 著、『UNIX Internals: The New Frontiers』、Prentice Hall 発行、1996 年、ISBN 0-13-101908-2
- Mauro、Jim and McDougall、Richard 著、『Solaris Internals: Core Kernel Components』、Sun Microsystems Press 発行、2001 年、ISBN 0-13-022496-0
- 『The SPARC Architecture Manual, Version 9』、Prentice Hall 発行、1998 年、ISBN 0–13–099227–5
- 『The SPARC Architecture Manual, Version 8』、Prentice Hall 発行、1994 年、ISBN 0-13-825001-4
- 『Pentium Pro Family Developer's Manual, Volumes 1-3』、Intel Corporation 発行、1996 年、ISBN 1-55512-259-0 (第 1 巻)、ISBN 1-55512-260-4 (第 2 巻)、ISBN 1-55512-261-2 (第 3 巻)
- Bonwick、Jeff 著、『The Slab Allocator: An Object-Caching Kernel Memory Allocator』、1994 年夏の Usenix Conference における報告書、1994 年、ISBN 9–99–452010–5
- 『SPARC Assembly Language Reference Manual』、Sun Microsystems 発行、2005 年
- 『x86 Assembly Language Reference Manual』、Sun Microsystems 発行、2005 年
- 『Writing Device Drivers 』、Sun Microsystems 発行、2005 年
- 『STREAMS Programming Guide 』、Sun Microsystems 発行、2005 年
- 『Solaris 64 ビット 開発ガイド』、Sun Microsystems 発行、2005 年

- 『リンカーとライブラリ』、Sun Microsystems 発行、2005 年

Sun のオンラインマニュアル

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、検索を行うこともできます。このサイトの URL は http://docs.sun.com です。

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'**

コード例は次のように表示されます。

- C シェル

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

- C シェルのスーパーユーザー

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

- Bourne シェルおよび Korn シェル

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

- Bourne シェルおよび Korn シェルのスーパーユーザー

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

- mdb

```
> command y|n [filename]
```

- kmdb

```
[cpu] 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。

◆ ◆ ◆ 第 1 章

1 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。

はじめに

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 オペレーティングシステムには、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および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。サードパーティーの開発者にも、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ます。そのため、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(3C)` 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。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、だれ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` や `crash` 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ー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ー
- 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

MDB の使用

MDB は、いくつかの一般的な機能を共有する `mdb` と `kmdb` の 2 つのコマンドとして、Solaris システムで利用できます。`mdb` コマンドを対話形式またはスクリプト内で使用すると、実行中のユーザープロセス、ユーザープロセスのコアファイル、カーネルクラッシュダンプ、実行中のオペレーティングシステム、オブジェクトファイル、およびその他のファイルをデバッグすることができます。`kmdb` コマンドを使用すると、オペレーティングシステムカーネルの実行を制御および停止する必要もあるときに、カーネルやデバイスドライバをデバッグできます。`mdb` を開始するには、`mdb(1)` コマンドを実行します。`kmdb` を開始するには、`kmdb(1)` のマニュアルページの説明に従ってシステムをブートするか、`-K` オプションを指定して `mdb` コマンドを実行します。

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。Solaris オペレーティングシステムの各リリースには、さらに MDB モジュールが追加され、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、一層高性能な機能が提供されていきます。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

2 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ーの利点について説明します。

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
- カーネル実行制御を持たない実行中のオペレーティングシステム (`/dev/kmem` や `/dev/ksyms` を介して)
- カーネル実行制御を備えた実行中のオペレーティングシステム (`kmdb(1)` を介して)
- 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
- raw データファイル

各ターゲットは、プロパティーの標準セットをエクスポートします。プロパティーには、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 アーキテクチャー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ー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 (23 ページの「構文」を参照)。第 5 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、動的に読み込まれるライブラリで、一連の 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 の読み込みや読み込み解除ができます。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」とは、実行するコマンドセットが入っているテキストファイルのことです。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。

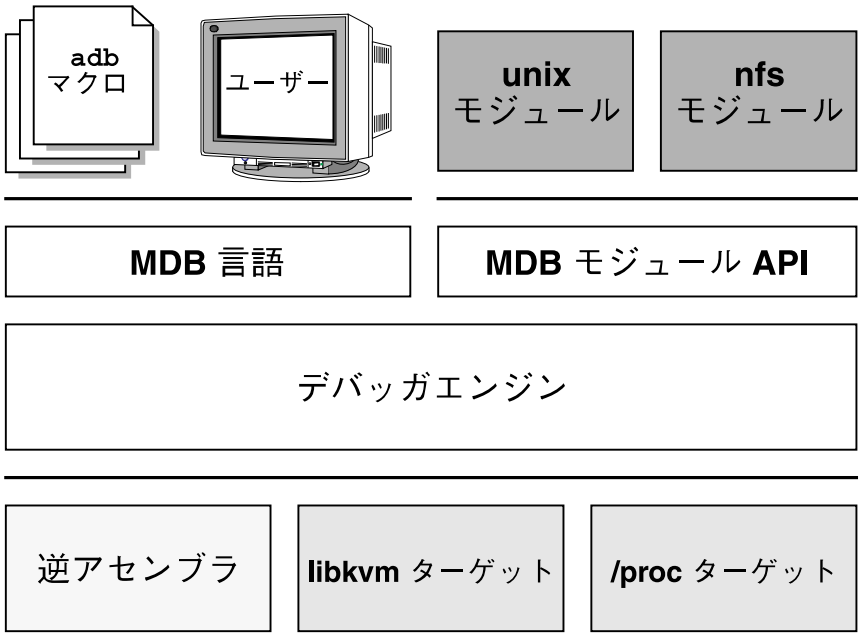

図 2–1　MDB アーキテクチャー

モジュール性

MDB のモジューラアーキテクチャーの利点は、追加デバッガコマンドを含むモジュールを読み込む機能を拡張できることです。MDB アーキテクチャーでは、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` 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ともに、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ー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 は独自のプログラムの稼働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ー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、次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ー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

3 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、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、下線、またはピリオドで始まる、一連の文字列、数字、下線、ピリオド、逆引用符のことです。識別子は、シンボル名、変数、dcmd、 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、`$` か `:` を接頭辞として 1 つ持つ dcmd は、組み込み演算子として提供され、従来の `adb(1)` ユーティリティーのコマンドセットとの

互換性を備えています。dcmd が解析されると、/、\、?、=、>、$、: の各文字は、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、空白文字で区切られた、0 個以上の一連のワードが後ろに続く dcmd のことです。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、25 ページの「演算機能の拡張」と27 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あとに、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、処理されたあとに格納されます (31 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。いずれかの dcmd が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、25 ページの「演算機能の拡張」に示す規則を使用して評価されます。

コマンド

コマンドは、次のうちのどれかです。

pipeline [! word ...] [;]
: 単純コマンドまたはパイプラインは、接尾辞として感嘆符 (!) を付けることもできます。この場合、デバッガは `pipe(2)` を開いたあと、MDB パイプライン内の最後の dcmd の標準出力を、`$SHELL -c` の実行により作成された外部プロセスへ送ります。`-c` オプションのあとには、感嘆符 (!) の後ろのワードを結合して生成される文字列が続きます。詳細については、28 ページの「シェルエスケープ」を参照してください。

expression pipeline [! word ...] [;]
: 単純コマンドまたはパイプラインは、接頭辞として式を付けることもできます。この場合、パイプラインが実行される前に、ドットの値 (「.」で表される変数) が式の値に設定されます。

expression , expression pipeline [! word ...] [;]
: 単純コマンドまたはパイプラインは、接頭辞として式を 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、そのあと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、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、式が評価されたあと、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あと、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、どの dcmd も実行されません。つまり、デバッガは感嘆符 (!) の後ろに指定されたワードを結合して、その文字列を `$SHELL -c` の後ろに指定して実行します。

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、ドル記号の後ろ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。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 (デフォルト) 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 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、32 ページの「フォーマット dcmd」を参照してください。

単項演算子

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。単項演算子を次に示します。

#*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

2 項演算子

2 項演算子は左結合で、単項演算子よりも優先度は低くなります。次に 2 項演算子を、優先度の高いものから順に示します。

`*` 整数の乗算

`%` 整数の除算

`#` 左辺を右辺の最小の倍数に切り上げる

`+` 整数の加算

`-` 整数の減算

`<<` ビット単位で左へシフト

`>>` ビット単位で右へシフト

`==` 等しい

`!=` 異なる

`&` ビット単位の論理積

`^` ビット単位の排他的論理和

`|` ビット単位の論理和

引用

23 ページの「構文」で説明したように、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。シェルエスケープは、mdb の使用時にのみ利用でき、kmdb の使用時には利用できません。$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 オプションのあとには、感嘆符 (!) の後ろのワードを結合して生成される文字列が続きます。

! 文字は、ほか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あと、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

変数は、変数名、対応する整数値、および一連の属性を持ちます。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
cpuid	kmdb が現在実行されている CPU に対応する CPU 識別子
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
hits	一致したソフトウェアイベント指定子が一致した回数。58 ページの「イベントコールバック」を参照してください。
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

thread　現在の代表スレッドのスレッド識別子。この識別子の値は、現在のターゲットが使用しているスレッド化モデルによって変わります。59 ページの「スレッドのサポート」を参照してください。

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ーによって決まります。

シンボルの名前解決

23 ページの「構文」で説明しているように、式コンテキスト内にあるシンボル識別子は当該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 が提供する専用のユーザー定義シンボルテーブルは、ほかのどのターゲットシンボルテーブルよりも先に検索されます。専用シンボルテーブルは、最初は空の状態ですが、`::nmadd` dcmd 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、アドレスをもっと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。

シンボル名の解釈に使用する範囲は次のように指定できます。つまり、*object*‘*name*、*file*‘*name*、または *object*‘*file*‘*name* です。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、次のシンボル名を考えてみましょう。

```
specfs‘_init
```

これは、 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

プロセスターゲットも上記 4 つの形式を受け入れることができますが、この場合、接頭辞としてオプションのリンクマップ ID (lmid) が付きます。lmid 接頭辞は、LM、リンクマップ ID (16 進数)、および逆引用符の順番で構成されます。たとえば、次のシンボル名を考えてみましょう。

```
LM0‘libc.so.1‘_init
```

これは、libc.so.1 ライブラリにある _init シンボルの値として評価して、この値がリンクマップ 0 (LM_ID_BASE) 上に読み込まれます。同じライブラリが複数のリンクマップ上に読み込まれている場合、シンボル名の重複を解決するためにリンクマップ指定子が必要になることもあります。リンクマップの詳細については、『リンカーとライブラリ』と dlopen(3C) のマニュアルページを参照してください。リンクマップ識別子を表示するには、showlmid オプションの設定に従ってシンボルを出力します (151 ページの「コマンド行オプションの概要」を参照)。

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 つの異なる大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。1 つの 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、大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、最初のものが大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、大域定義リスト上の次の dcmd である m2`d が、大域定義として使用されます。dcmd や walker の現在の定義は、次に示すように、::which dcmd を使用して定義できます。大域定義リストは、::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、あるプロパティー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、もっとも左にある 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ーを作成します。

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あと、右側の dcmd が実行されます。パイプラインのもっとも右にある 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。

フォーマット dcmd

/、\、?、= 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、次の表に示すように、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。

フォーマット dcmd 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します。

`\` 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します。

`?` 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します。

= 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します。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す。

また、MDB は、ドットのほかに、インクリメントと呼ばれる大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 が終了したあとには、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。25 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。

次の表に示すように、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)

N 復帰改行

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)

R 2 進数 int (8 バイト)

S　C の文字列表記法を使った文字列 (変数サイズ)
T　水平タブ
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、次の修飾子の 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　各式の値の最下位バイトを、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 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それぞれ検索できます。この場合には、次の修飾子の 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` となるような値 `X` 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

4 対話

この章では、MDB の対話型コマンド行編集機能と履歴機能、出力ページャー、およびデバッガのシグナル処理について説明します。

コマンド行の再入力

端末デバイスから入力した最後の HISTSIZE (デフォルトは 128) 個のコマンドのテキストはメモリーに保存されます。次に説明するインライン編集機能が提供するキーマッピングを使用すると、この履歴リストから以前入力したコマンドを検索および取得できます。

インライン編集機能

標準入力が端末デバイスである場合、コマンド行を編集するために、MDB が提供するいくつかのシンプルな emacs スタイルの機能を使用できます。編集モードの search、previous、および next コマンドを使用すると、履歴リストにアクセスできます。検索するときに一致するのは、パターンではなく、文字列だけです。次に示すリストにおいて、制御記号はキャレット文字 (^) とそれに続く大文字の英字で表記されます。エスケープシーケンスは M- とそれに続く文字で表記されます。たとえば、M-f (「メタエフ」と呼ぶ) を入力するには、まず <ESC> を押して、次に「f」を押すか、あるいは Meta キーをサポートしているキーボード上では、まず Meta キーを押して、次に「f」を押します。コマンド行を発行および実行するには、復帰改行文字 (RETURN または NEWLINE) を使用します。編集コマンドは次のとおりです。

^F　カーソルを 1 文字だけ前方 (右) に移動します。

M-f　カーソルを 1 単語だけ前方に移動します。

^B　カーソルを 1 文字だけ後方 (左) に移動します。

M-b　カーソルを 1 単語だけ後方に移動します。

^A　カーソルを行の先頭に移動します。

^E　カーソルを行の末尾に移動します。

^D　カーソルのある行が空でない場合、カーソル位置の文字を削除します。カーソルのある行が空である場合、^D は EOF を意味し、デバッガは終了します。

M-^H　(メタ - バックスペース) 直前の単語を削除します。

^K　カーソルから行の末尾までを削除します。

^L　カーソルのある行を出力し直します。

^T　現在の文字と次の文字を入れ換えます。

^N　履歴リストから次のコマンドを取得します。^N を入力するたびに、さらに次のコマンドが取得されます。

^P　履歴リストから前のコマンドを取得します。^P を入力するたびに、さらに前のコマンドが取得されます。

^R[string]　履歴リストから文字列を含むコマンドを後方に検索します。文字列は復帰改行文字 (RETURN または NEWLINE) で終了する必要があります。文字列を省略した場合、前回入力した文字列を含むコマンドを検索します。

編集モードはまた、次のようなユーザー定義シーケンスも編集コマンドとして解釈します。ユーザー定義シーケンスを読み取ったり、変更したりするには、`stty(1)` コマンドを使用します。

erase　ユーザー定義の消去文字 (通常は ^H または ^?)。前の 1 文字を削除します。

intr　ユーザー定義の割り込み文字 (通常は ^C)。現在のコマンドを中断して、新しいプロンプトを出力します。

kill　ユーザー定義の強制終了文字 (通常は ^U)。現在のコマンド行全体を強制終了します。

quit　ユーザー定義の終了文字 (通常は ^\)。デバッガを終了します。

suspend　ユーザー定義の中断文字 (通常は ^Z)。デバッガを中断します。

werase　ユーザー定義の単語消去文字 (通常は ^W)。直前の単語を消去します。

矢印キーのある拡張キーパッドをサポートするキーボード上では、`mdb` は次のようなキーストロークも編集コマンドとして解釈します。

up-arrow　履歴リストから前のコマンドを取得します (^P と同じ)。

down-arrow　履歴リストから次のコマンドを取得します (^N と同じ)。

left-arrow　カーソルを 1 文字だけ後方に移動します (^B と同じ)。

right-arrow　カーソルを 1 文字だけ前方に移動します (^F と同じ)。

キーボードショートカット

MDB には、1 組のキーボードショートカットが用意されており、下の表に示すキーストロークが、MDB プロンプトに続く最初の文字として入力されたときに、個々のキーストロークが一般的な MDB コマンドに結合されます。キーボードショートカットは次のとおりです。

[　コマンド `::step over` を実行します

]　コマンド `::step` を実行します

出力ページャー

`mdb` には出力ページャーが組み込まれています。出力ページャーを使用できるのは、デバッガの標準出力が端末デバイスである場合だけです。コマンドを実行するたびに、`mdb` は 1 画面分の出力を生成してから中断し、次のようなページャーのプロンプトを表示します。

```
>> More [<space>, <cr>, q, n, c, a] ?
```

出力ページャーは次のようなキーシーケンスを認識します。

空白文字
　次の 1 画面分の出力を表示します。

a、A
　現在のトップレベルのコマンドを中止して、プロンプトに戻ります。

c、C
　現在のトップレベルのコマンドが完了するまで、画面ごとに中断するのではなく、出力を表示し続けます。

n、N、復帰改行文字 (NEWLINE または RETURN)
　次の 1 行分の出力を表示します。

q、Q、^C、^\
　現在の `dcmd` を終了 (中止) します。

シグナル処理

PIPE および QUIT のシグナルの場合、デバッガは無視します。INT シグナルの場合、現在実行中のコマンドが中止されます。ILL、TRAP、EMT、FPE、BUS、および SEGV のシグナルの場合、デバッガは中断され、特別な処理を行います。これらのシグナルが非同期的に生成された場合 (つまり、`kill(2)` を使用して別のプロセスから配信された場合)、mdb はシグナルをそのデフォルトの設定に復元して、コアダンプを生成します。しかし、これらのシグナルがデバッガプロセス自身によって同期的に生成され、外部的に読み込まれた `dmod` から dcmd が現在実行されており、さらに、標準入力が端末である場合、ユーザーは `mdb` が提供するメニューを使用して、強制的にコアダンプを生成するか、コアダンプを生成せずに終了するか、停止してデバッガに接続するか、あるいは、そのまま再開するかを選択できます。再開オプションを選択した場合、すべてのアクティブなコマンドは中止され、障害が発生したときに `dcmd` がアクティブであった `dmod` を読み込み解除します。この後、ユーザーは `dmod` を読み込み直すことができます。`dcmd` にバグがある場合に対して、再開オプションは制限付きの保護機能を提供します。再開オプションの危険性については、159 ページの「警告」の「エラー回復メカニズムの使用」を参照してください。

◆ ◆ ◆ 第 5 章

5 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、`$q` に相当します。`adb(1)` の使用経験を持つプログラマや、簡略符号あるいは難解なコマンドを認識するプログラマは、組み込みコマンドの `$` や `:` 形式の方を好むかもしれません。一方、MDB に慣れていないプログラマは、より詳細で分かりやすい `::` 形式を好むでしょう。次に組み込みコマンドを、アルファベット順に説明します。`$` または `:` の形式に `::identifier` に対応するものがある場合、`::identifier` 形式の下に `$` または `:` の形式を示します。

組み込み dcmd

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

指定され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。`>` の後ろに `//` で囲まれた修飾子がある場合、ドット値は割り当ての一部として変更されます。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ー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ー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*/ 構文を使用する必要があります。

$< *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、指定されたマクロファイルからコマンドを読み取って実行しますが、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、まず、ターゲットのプロセス ID と現在のシグナルを出力して、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指定されている場合には、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。

---

64 ビット **SPARC** のみ – スタックトレースを要求する場合は、バイアスされたフレームポインタ値、つまり、仮想アドレス - (マイナス) `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取得または設定します。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大域的な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。`::set -P` または `-P` コマンド行オプションを使用しても、プロンプトは設定できます。

`$M`
(`kmdb` のみ) `$<` dcmd で使用するために `kmdb` によってキャッシュされるマクロファイルの一覧を示します。

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取得または設定します。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。`::set -s` オプションを使用しても、シンボルマッチングディスタンスは変更できます。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`
指定された変数のうち、ゼロ以外の値を持つ変数のリストを出力します。`::vars` dcmd を使用すると、変数の一覧表示にほか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。通常は、MDB が端末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`
ターゲットを書き込み用にもう一度開きます (`-w` オプションをコマンド行に指定して MDB を実行する場合と同じ)。`::set -w` オプションを使用しても、書き込みモードを有効にできます。

`::array` *type count*
配列の各要素のアドレスを出力します。配列要素のタイプは最初の引数 *type* で指定し、計算される要素の数は 2 番目の引数 *count* で指定します。`::array` の出力を `::print` dcmd にパイプラインで渡すと、配列データ構造体の要素を出力できます。

注 – この dcmd は、`mdb` で使用されるように設計された圧縮シンボルデバッグ情報を含むオブジェクトだけに使用できます。現時点ではこの情報を利用できるのは、特定の Solaris カーネルモジュールだけです。この圧縮シンボルデバッグ情報を処理するには、圧縮解除ソフトウェア SUNWzlib がインストールされている必要があります。

[*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接続して、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ください。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 (PID) 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。

`::branches` [`-v`]
現在の CPU によって取り入れられた最新の分岐を表示します。現時点でこの dcmd を利用できるのは、該当するプロセッサ固有の機能が有効になっている x86 システムで `kmdb` を使用している場合に限られます。表示できる分岐の数と種類は、プロセッサアーキテクチャーによって決まります。`-v` オプションを指定すると、各分岐の前にある命令が表示されます。

`::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、出力ページャーは通りません。この dcmd は、`|` 演算子とともに使用するようになっています。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を処理できます。

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。プロセスのコンテキストを指定するには、カーネルの仮想アドレス空間において、そのプロセスの `proc` 構造体の *address* を使用します。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、指定されたユーザープロセスの物理メモリーページがクラッシュダンプに含まれる場合、クラッシュダンプを検査するときに MDB が実行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、すべてのページまたは現在のユーザープロセスのページをダンプできるようにカーネルクラッシュダンプ機能を構成できます。`::status` dcmd を使用すると、現在のクラッシュダンプの内容を表示できます。

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あと、新しいターゲットは、自身の dcmd を大域レベルに置きます。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` を実行すると、カーネルターゲットを復元できます。

`::cpuregs` [`-c` *cpuid*]
現在の CPU または指定した *cpuid* 用に設定された現在の汎用レジスタセットを表示します。このコマンドは、`kmdb` を使用している場合にのみ利用できます。

`::cpustack` [`-c` *cpuid*]
現在の CPU または指定した *cpuid* で実行されているスレッドの C スタックのバックトレースを表示します。このコマンドは、`kmdb` を使用している場合にのみ利用できます。

`::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` [`-afw`] [`-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の命令を示す「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。デバッガが現在、動作中のプロセス、コアファイル、またはクラッシュダンプに接続されていない場合、`-f` オプションはデフォルトで有効になります。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」を強制的に開くモードに設定できます。デフォルトでは、ウィンドウのサイズは命令 10 個分です。`-n` オプションを使用すれば、命令の数を明示的に指定できます。`-a` オプションを指定すると、アドレスがシンボルではなく数値として出力されます。

`::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。

[*address*] `::dump` [`-eqrstu`] [`-f`|`-p`] [`-g` *bytes*] [`-w` *paragraphs*]
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、ダンプする繰り返し数としてではなく、ダンプするバイト数として解釈されます。また、`::dump` dcmd は、次のオプションも認識します。

`-e` エンディアン性を調整します。`-e` オプションを指定すると、4 バイトワードが使用されます。`-g` オプションを使用すると、デフォルトのワードサイズを変更できます。

`-f` ターゲットの仮想アドレス空間からではなく、指定された仮想アドレスに対応するオブジェクトファイルの位置からデータを読み取ります。デバッガが現在、動作中のプロセス、コアファイル、またはクラッシュダンプに接続されていない場合、`-f` オプションはデフォルトで有効になります。

`-g` *group* バイトをバイトのグループで表示します。デフォルトの *group* サイズは 4 バイトです。*group* サイズは行幅を分割する 2 のべき乗にする必要があります。

`-p` *address* を、仮想アドレスではなく、ターゲットのアドレス空間内の物理的なアドレス位置として解釈します。

`-q` データの ASCII 形式の復号化を出力しません。

`-r` 各行に、絶対的なアドレスではなく、開始アドレスからの相対的な行数を与えます。このオプションを指定すると、`-u` オプションも暗黙的に指定されます。

`-s` 繰り返される行を省略します。

`-t` 行全体を読み取って出力するのではなく、指定されたアドレスの内容だけを読み取って表示します。

`-u` 段落の境界に配列するのではなく、配列せずに出力します。

`-w` *paragraphs* 行ごとに 16 バイトの段落で段落を表示します。デフォルトの段落数は 1 です。`-w` の最大値は 16 です。

`::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。`$[ ]` で囲まれた式は値に評価されて、デフォルトで出力されます。

`::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` [*object*]
`$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。*object* 名を指定すると、出力が、対応するオブジェクトファイル内にあるファイルシンボルに制限されます。

[*address*] `::findsym [-g] [` *address* | *symbol* `...]`
指定されたシンボルまたはアドレスを参照する命令用の命令テキストを検索します。検索リストは、dcmd の前にアドレスとして指定された 1 つまたは複数のアド

レスまたはシンボル名、あるいは、dcmd の後ろに指定された 1 つまたは複数のシンボル名または式から構成される必要があります。-g オプションを指定すると、検索は命令テキスト、つまり、ターゲットのシンボルテーブルにある大域的に参照できる関数の一部に制限されます。

注 – *SPARC* のみ。SPARC 命令セットアーキテクチャーを使用するターゲットをデバッグするときに使用できるのは、::findsym dcmd だけです。

::formats
/、\、?、= などのフォーマット dcmd とともに使用する、利用可能な出力書式制御文字の一覧を表示します。フォーマットとその使用法については、32 ページの「フォーマット dcmd」で説明しています。

[thread] ::fpregs [-dqs]
[thread] $x, $X, $y, $Y
代表スレッドの浮動小数点レジスタセットを出力します。スレッドを指定すると、そのスレッドの浮動小数点レジスタが表示されます。スレッド式は、59 ページの「スレッドのサポート」で説明したスレッド識別子の 1 つである必要があります。

注 – SPARC のみ。-d、-q、および -s オプションを使用すると、倍精度 (-d)、四倍精度 (-q)、あるいは単精度 (-s) の浮動小数点値の集合として、浮動小数点レジスタを表示できます。

::grep *command*
指定されたコマンド文字列を評価したあと、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。

[*address* [, *len*]] ::in [-L *len*]
address で指定した入出力ポートから *len* バイトを読み込み、表示します。-L オプションを指定した場合、その値は左側に指定した繰り返し回数に優先されます。*len* には 1、2、または 4 バイトを指定し、ポートアドレスはその長さに応じて調整する必要があります。このコマンドは、x86 システムで kmdb を使用している場合にのみ利用できます。

[*address*] `::list` *type member* [*variable-name*]
リンクリストデータ構造体の要素を調べて、リスト内の各要素のアドレスを出力します。オプションの *address* を使用すると、リスト内の最初の要素のアドレスを指定できます。その他の場合には、リストは現在のドット値から始まると想定されます。MDB が適切なサイズのオブジェクトから読み取ることができるように、*type* パラメータは C 言語の構造体または共用体を指定する必要があり、リスト内の要素の型を記述するのに使用されます。*member* パラメータは、リスト内の次の要素へのポインタを含む、*type* のメンバーを指定するのに使用されます。`::list` dcmd は、要素を読み取っている間、NULL ポインタを見つけるか、もう一度最初の要素に到達するまで (つまり、循環リスト)、あるいは、エラーが発生するまで、繰り返します。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。

注 – この dcmd は、`mdb` で使用されるように設計された圧縮シンボルデバッグ情報を含むオブジェクトだけに使用できます。現時点ではこの情報を利用できるのは、特定の Solaris カーネルモジュールだけです。この圧縮シンボルデバッグ情報を処理するには、圧縮解除ソフトウェア SUNWzlib がインストールされている必要があります。

`::load` [`-s`] *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読み込み解除する必要があります。`-s` オプションを指定すると、MDB はモジュールを発見または読み込めなくても何も出力せず、エラーメッセージも表示しません。

`::log` [`-d` | [`-e`] *filename*]
`$>` [*filename*]
出力ログを有効にしたり、無効にしたりします。MDB は、相互ログ機能を提供しているので、まだユーザーとの対話処理が行われているときにも、入力コマンドと標準出力の両方が同じファイルに記録できます。`-e` オプションでファイルを指定すると、指定したファイルへのログの書き込みが有効になり、ファイル名を指定しない場合、前回のログファイルへの書き込みが再び有効になります。`-d` オプションは、ログを無効にします。また、`$>` 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てから新しい値を出力します。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* 引数が指定されている場合には、MDB はその記述に一致する割り当てだけを表示します。

[*address*] `::nm` [`-DPdghnopuvx`] [`-t` *types*] [`-f` *format*] [*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。オプションの *address* を dcmd の前に指定した場合、address に対応するシンボル用のシンボルテーブルエントリだけが表示されます。*object* 名を指定すると、この読み込みオブジェクト用のシンボルテーブルだけが表示されます。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) を出力します。

`-P` `.symtab` の代わりに専用シンボルテーブルを出力します。

`-d` 値とサイズフィールドを 10 進数で出力します。

`-f` format [,format...] 指定されたシンボル情報だけを出力します。有効な書式化引数文字列は次のとおりです。

ndx	シンボルテーブルのインデックス
val	シンボルテーブル
size	サイズ (バイト)
type	シンボルの型
bind	結合
oth	その他
shndx	セクションのインデックス
name	シンボル名
ctype	シンボルの C 言語の型 (既知の場合)
obj	シンボルを定義するオブジェクト

`-g` 大域シンボルだけを出力します。

`-h` ヘッダー行を抑制します。

`-n` 名前順にシンボルをソートします。

`-o` 値とサイズフィールドを 8 進数で出力します。

`-p` シンボルを、一連の `::nmadd` コマンドとして出力します。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ます。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みます。

`-t` type [,type...] 指定された型のシンボルだけを出力します。有効な型引数文字列は次のとおりです。

noty	STT_NOTYPE
objt	STT_OBJECT
func	STT_FUNC
sect	STT_SECTION
file	STT_FILE
comm	STT_COMMON
tls	STT_TLS
regi	STT_SPARC_REGISTER

`-u` 未定義のシンボルだけを出力します。

`-v` 値順にシンボルをソートします。

`-x` 値とサイズフィールドを 16 進数で出力します。

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。29 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* - *value* に設定します。

`-f` シンボルのタイプを `STT_FUNC` に設定します。

`-o` シンボルのタイプを `STT_OBJECT` に設定します。

`-s` シンボルのサイズを *size* に設定します。

`::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` [`-v`]
既知の読み込み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。`-v` オプションを指定した場合、バージョン情報がわかっていれ

ば、各オブジェクトのバージョンが表示されます。バージョン情報がわからない場合は、Unknown というバージョンが出力に表示されます。

::offsetof *type member*
指定された型とメンバーのオフセットを出力します。型はC言語の構造体の名前である必要があります。メンバーがビットフィールドでない場合、オフセットはバイトで出力されますが、その場合、オフセットはビットで出力されます。分かりやすくするために、出力の末尾には常に適切な単位が付きます。型名には、29 ページの「シンボルの名前解決」で説明した逆引用符 (‘) 有効範囲規則を使用できます。

注 – この dcmd は、mdb で使用されるように設計された圧縮シンボルデバッグ情報を含むオブジェクトだけに使用できます。現時点ではこの情報を利用できるのは、特定の Solaris カーネルモジュールだけです。この圧縮シンボルデバッグ情報を処理するには、圧縮解除ソフトウェア SUNWzlib がインストールされている必要があります。

[*address* [, *len*]] ::out [-L *len*]
指定された *value* を、*address* によって指定された入出力ポートに書き出します。-L オプションを指定した場合、その値は左側に指定した繰り返し回数に優先されます。*len* には 1、2、または 4 バイトを指定し、ポートアドレスはその長さに応じて調整する必要があります。このコマンドは、x86 システムで kmdb を使用している場合にのみ利用できます。

[*address*] ::print [-aCdiLptx] [-c *lim*] [-l *lim*] [*type* [*member*|*offset* ...]]
指定された仮想 *address* にあるデータ構造体を、指定された *type* 情報を使用して出力します。 *type* パラメータはC言語の構造体、共用体、列挙型、基本的な整数型、あるいは、これらの型へのポインタを指定できます。型名に空白文字が含まれる場合 (たとえば、「struct foo」)、単一引用符または二重引用符で囲む必要があります。型名には、29 ページの「シンボルの名前解決」で説明した逆引用符 (‘) 有効範囲規則を使用できます。 *type* が構造化された型である場合、 ::print dcmd は構造体または共用体の各メンバーを再帰的に出力します。*type* 引数を指定せず、かつ、静的または大域的な STT_OBJECT シンボルがアドレスに一致する場合、::print は自動的に適切な型を推測します。

type 引数の後ろに *member* 式または *offset* 式のリストをオプションで指定することができます。その場合は、指定された *type* のメンバーまたはサブメンバーだけが表示されます。メンバーは、配列インデックス演算子 ([])、構造体メンバー演算子 (->)、および構造体ポインタ演算子 (.) を含むC構文を使って指定できます。オフセットは、MDB 算術展開の構文 ($[]) を使って指定できます。データ構造体を表示したあと、::print はドットを *type* のサイズ分 (バイト) だけインクリメントします。

注 – ::print dcmd は、MDB で使用されるように設計された圧縮シンボルデバッグ情報を含むオブジェクトだけに使用できます。現時点ではこの情報を利用できるのは、特定の Solaris カーネルモジュールとユーザーライブラリだけです。この圧縮シンボルデバッグ情報を処理するには、圧縮解除ソフトウェア SUNWzlib がインストールされている必要があります。

-a オプションを指定すると、各メンバーのアドレスが表示されます。-i オプションを指定すると、左側の式が即値とみなされ、指定された型を使って表示されます。-p オプションを指定すると、::print はアドレスを、仮想メモリーアドレスではなく、物理メモリーアドレスとして解釈します。-t オプションを指定すると、各メンバーの型が表示されます。-d または -x オプションを指定すると、すべての整数は 10 進数 (-d) または 16 進数 (-x) で表示されます。値を 10 進数または 16 進数のどちらで表示するかを決定するとき、デフォルトではヒューリスティックに行います。文字列として読み取りまたは表示される文字配列内の文字数は -c オプションで制限できます。-C オプションを指定すると、文字数は制限されません。読み取りまたは表示される標準配列内の要素数は -l オプションで制限できます。-L オプションを指定すると、文字数は制限されず、配列内のすべての要素が表示されます。-c と -l オプションのデフォルト値を変更するには、付録 A で説明する ::set または -o コマンド行オプションを使用します。

::quit [- u]
$q [-u]
デバッガを終了します。kmdb のみを使用している場合は、-u オプションによって、デバッガがオペレーティングシステムの実行を再開し、デバッガを読み込み解除します。-u オプションは、kmdb がブート時に読み込まれた場合は使用できません。-u オプションを指定しないと、::quit によって kmdb が終了してファームウェアに戻るか (SPARC システムの場合)、システムがリブートします (x86 システムの場合)。

[*thread*] ::regs
[*thread*] $r
代表スレッドの汎用レジスタセットを出力します。スレッドを指定すると、そのスレッドの汎用レジスタセットが表示されます。スレッド式は、59 ページの「スレッドのサポート」で説明したスレッド識別子の 1 つである必要があります。

::release [-a]
:R [- a]
以前に接続されたプロセスまたはコアファイルを解放します。-a オプションを指定すると、プロセスは解放され、停止および中断されたままになります。このようなプロセスを継続するには prun(1) を使用し、再開するには MDB などのデバッガを適用します。デフォルトでは、解放されたプロセスは、MDB の ::run で作成された場合には強制的に終了され、MDB の -p オプション、::attach または :A dcmd で接続されていた場合には解放および実行状態に設定されます。

`::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ーを取得または設定します。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ー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ぎます (`-F` オプションをコマンド行に指定して MDB を実行する場合と同じ)。

`-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します。パス引数は特殊トークンを使用できます。付録 A の `-I` コマンド行オプションの説明を参照してください。

`-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します。パス引数は特殊トークンを使用できます。付録 A の `-I` コマンド行オプションの説明を参照してください。

`-o` 指定されたデバッガオプションを有効にします。`+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します。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 に記載してあります。

`-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します。

`-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します。詳細は、付録 A の `-s` コマンド行オプションの説明を参照してください。

`-w` ターゲットを書き込み用にもう一度開きます (`-w` オプションをコマンド行に指定して `mdb` を実行する場合と同じ)。

`::showrev` [`-pv`]
現在のターゲットに対応するハードウェアおよびソフトウェアの改訂情報を表示します。オプションを指定しないと、一般的なシステム情報が表示されます。`-p` オプションを指定すると、パッチに含まれる各読み込みオブジェクトの情報が表示されます。`-v` オプションを指定すると、各読み込みオブジェクトの情報が表示されます。`-p` オプションでは、バージョン情報を持たない読み込みオブジェクトは出力から除外されます。バージョン情報を持たない読み込みオブジェクトは、`-v` オプションの出力に `Unknown` として報告されます。

`::sizeof` *type*
指定された型のサイズをバイトで表示します。*type* パラメータは C 言語の構造体、共用体、列挙型、基本的な整数型、あるいは、これらの型へのポインタを指定できます。型名には、29 ページの「シンボルの名前解決」で説明した逆引用符 (‘) 有効範囲規則を使用できます。

注 – この dcmd は、`mdb` で使用されるように設計された圧縮シンボルデバッグ情報を含むオブジェクトだけに使用できます。現時点ではこの情報を利用できるのは、特定の Solaris カーネルモジュールだけです。この圧縮シンボルデバッグ情報を処理するには、圧縮解除ソフトウェア SUNWzlib がインストールされている必要があります。

[*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 の前に明示的な *address* が指定されている場合には、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指定されている場合には、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。

---

64 ビット **SPARC** のみ – スタックトレースを要求する場合は、バイアスされたフレームポインタ値、つまり、仮想アドレス - (マイナス) `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。

cpuid `::switch`
cpuid `:x`
`kmdb` のみを使用している場合は、特定の *cpuid* によって指定された CPU に切り替えられ、この CPU の現在のレジスタ状態がデバッグの代表値として使用されます。

`::term`
MDB がコマンド行編集などの端末に依存したあらゆる入力および出力操作の実行に使用している端末タイプの名前を出力します。

thread `::tls` *symbol*
指定されたスレッドのコンテキストにおいて、指定された TLS (Thread-Local Storage) シンボル用の記憶領域のアドレスを出力します。スレッド式は、59 ページの「スレッドのサポート」で説明したスレッド識別子の 1 つである必要があります。シンボル名には、29 ページの「シンボルの名前解決」で説明している任意の有効範囲規則を使用できます。

`::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、そのタグは削除されます。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 を読み込み解除します。`::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` を指定すると、変数は `$<` dcmd を使用して、デバッガの再処理に適切な形式で出力されます。このオプションを使用すると、変数をマクロファイルに記録しておき、後でこれらの値を復元できます。`-t` を指定すると、タグ付き変数だけが出力されます。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` [`-a` *as*]
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 を利用できるのは、カーネルターゲットを検査しているとき、あるいは、`::context` dcmd を実行したあとで、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ときだけです。

カーネルコンテキストからカーネルターゲットを検査しているとき、`-a` オプションを使用すると、仮想アドレスから物理アドレスへの変換で使用される代替アドレス空間構造体のアドレス (*as*) を指定できます。デフォルトでは、この変換にはカーネルのアドレス空間が使用されます。ダンプに含まれるのがカーネルページだけの場合でも、このオプションはアクティブなアドレス空間に対して利用できます。

[*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、大域的な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形式で出力します。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、31 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。これらの dcmd を使用すると、指定された dcmd または walker の大域定義を現在提供しているのはどの dmod かを判断できます。大域的な名前解決の詳細については、31 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ーを一覧表示します。外部データバッファーは、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。これらのバッファーは、dcmd による使用が可能です。詳細については、148 ページの「`mdb_get_xdata()`」を参照してください。

◆ ◆ ◆ 第 6 章

6 実行制御

MDB は、動作中のプログラムの実行を制御および追跡する機能を提供します。動作中のプログラムには、ユーザーアプリケーションや、稼働中のオペレーティングシステムカーネルおよびデバイスドライバがあります。mdb コマンドを使用すると、すでに実行中のユーザープロセスを制御したり、デバッガの制御のもとで新しいプロセスを作成したりできます。また、kmdb をブートするか読み込んで、オペレーティングシステムカーネルそのものの実行を制御したり、デバイスドライバをデバッグしたりすることも可能です。この章では、ターゲットの実行を制御するために使用できる組み込み dcmd について説明します。これらのコマンドは、説明の中で言及している場合を除き、mdb または kmdb のどちらでも使用できます。kmdb の実行制御のみに関する詳細については、第 7 章を参照してください。

実行制御

MDB はシンプルな実行制御モデルを提供します。ターゲットプロセスを起動するには、::run を使用してデバッガ内から起動するか、:A、::attach、または -p コマンド行オプション (第 5 章を参照) を使用して MDB を既存のプロセスに接続します。別の方法として、kmdb を使用してカーネルをブートしたり、kmdb を後で読み込んだりすることもできます。どちらの場合も、ユーザーは追跡対象「ソフトウェアイベント」のリストを指定できます。追跡対象イベントがターゲットプログラム内で発生するごとに、ターゲット内のすべてのスレッドが停止して、そのイベントをトリガーしたスレッドが代表スレッドとして選択され、デバッガに制御が戻ります。ターゲットプログラムが「running (動作中)」と設定されると、ユーザー定義割り込み文字 (通常は Ctrl + C) を入力して、非同期的にデバッガに制御を戻すことができます。

「ソフトウェアイベント」とは、デバッガが監視しているターゲットプログラムにおける状態遷移のことです。たとえば、デバッガは、プログラムカウンタレジスタが特定の値 (ブレークポイント) に変更されたり、特定のシグナルが送られたりするのを監視できます。

「ソフトウェアイベント指定子」とは、デバッガがどのようなイベントを監視するかをターゲットプログラムに指示するときに使用するソフトウェアイベントのクラスのことです。ソフトウェアイベント指定子のリストを表示するには、`::events` dcmd を使用します。各ソフトウェアイベント指定子には標準プロパティーセットが関連付けられています (59 ページの「組み込み dcmd」の「`::events`」を参照)。

デバッガはさまざまなソフトウェアイベント (たとえば、ブレークポイント、ウォッチポイント、シグナル、マシン障害、およびシステムコールなど) を監視できます。`::bp`、`::fltbp`、`::sigbp`、`::sysbp`、または `::wp` を使用すると、新しいソフトウェアイベント指定子を作成できます。各指定子には、コールバック (コマンド行に入力された場合と同じように動作する MDB コマンド文字列) とプロパティーセットが関連付けられています (59 ページの「組み込み dcmd」の「`::events`」を参照)。異なるコールバックとプロパティーを持つのであれば、任意の数の指定子を同じイベントに作成できます。追跡対象イベントの現在のリストや対応するイベント指定子のプロパティーを表示するには、`::events` dcmd を使用します。イベント指定子のプロパティーの定義については、59 ページの「組み込み dcmd」の「`::events` dcmd」と「`::evset` dcmd」を参照してください。

実行制御組み込み dcmd (59 ページの「組み込み dcmd」を参照) はいつでも利用できますが、実行制御がサポートされないターゲットに適用しようとすると、「このターゲットはサポートされません」というエラーメッセージが表示されます。

イベントコールバック

`::evset` dcmd とイベント追跡用の dcmd を使用すると、(`-c` オプションを使用して)、イベント指定子ごとにイベントコールバックを関連付けることができます。「イベントコールバック」とは、ターゲットにおいて対応するイベントが発生したときに実行される MDB コマンドを表す文字列のことです。このようなコマンドは、コマンドプロンプトに入力された場合と同じように実行されます。各コールバックが実行される前に、*dot* 変数には代表スレッドのプログラムカウンタの値が設定され、*hits* 変数には当該指定子が一致した回数 (現在の一致も含む) が設定されます。

ターゲットを継続するための 1 つまたは複数のコマンド (たとえば、`::cont` や `::step`) が、イベントコールバック自身に含まれる場合、このようなコマンドはターゲットをすぐに継続したり、ターゲットがもう一度停止するまで待機したりしません。その代わりに、このようなターゲットを継続する dcmd は「現在、継続操作が中断されている」ということをイベントコールバック内部に示して、すぐに戻ります。したがって、複数の dcmd がイベントコールバックに含まれている場合、ステップまたは継続用の dcmd は最後に指定する必要があります。「すべて」のイベントコールバックを実行したあと、一致した「すべて」のイベントコールバックが継続を要求している場合、ターゲットはすぐに実行を再開します。要求された継続操作が競合する場合、どの種類の操作を継続するかは、優先順位がもっとも高い継続操作が決定します。優先順位は (高いものから)、「ステップ (step)」、「ステップオーバー (step-over) または次 (next)」、「ステップアウト (step-out)」、そして「継続 (continue)」の順番になります。

スレッドのサポート

MDB は、ターゲットに関連する各スレッドのスタックとレジスタを調査する機能を提供します。持続的な「スレッド」変数には、現在の代表スレッド識別子が入っています。スレッド識別子の書式はターゲットによって異なります。`::regs` dcmd と `::fpregs` dcmd を使用すると、代表スレッドのレジスタセットや、別のスレッドのレジスタセットが現在利用できる場合はそのレジスタセットを調査できます。さらに、代表スレッドのレジスタセットは名前付き変数セットとしてエクスポートされます。1 つまたは複数のレジスタの値を変更するには、対応する名前付き変数に > dcmd を適用します。

MDB カーネルターゲットは、対応する内部スレッド構造体の仮想アドレスを指定されたスレッドの識別子としてエクスポートします。このアドレスは、オペレーティングシステムのソースコード内の `kthread_t` データ構造体に対応しています。`kmdb` を使用しているときは、`kmdb` を実行している CPU の CPU 識別子が `cpuid` 変数に格納されます。

MDB プロセスターゲットは、ネイティブな `lwp_*` インタフェース、`/usr/lib/libthread.so`、または `/usr/lib/libpthread.so` を使用するマルチスレッド化されたユーザープロセスの検査に対する適切なサポートを提供します。動作中のユーザープロセスをデバッグするとき、MDB はシングルスレッド化されたプロセスが `libthread` を `dlopen` または `close` するかどうかを検出して、動作中のユーザープロセスのスレッド化モデルのビューを自動的に調整します。プロセスターゲットスレッド識別子は、アプリケーションが使用するスレッド化モデルに基づいて、代表スレッドの `lwpid_t`、`thread_t`、または `pthread_t` に対応付けられます。

MDB がユーザープロセスターゲットをデバッグしているとき、コンパイラによってサポートされるスレッドに局所的な記憶領域をターゲットが利用している場合、MDB は自動的に、スレッドに局所的な記憶領域を参照するシンボル名を、現在の代表スレッドに対応する記憶領域のアドレスで評価します。`::tls` 組み込み dcmd を使用すると、代表スレッド以外のスレッドのシンボルの値も表示できます。

組み込み **dcmd**

[*addr*] `::bp` [+/-dDestT] [`-c` *cmd*] [`-n` *count*] *sym* ...
addr `:b` [*cmd* ...]

指定された場所にブレークポイントを設定します。`::bp` dcmd は、指定されたアドレスまたはシンボル (dcmd の前にある明示的な式で指定されたオプションのアドレスを含む) ごと、そして、dcmd の後ろにある文字列または即値ごとにブレークポイントを設定します。引数には、指定された特定の仮想アドレスを示すシンボル名または即値を指定できます。シンボル名を指定した場合、ターゲットプロセス内でまだ評価できないシンボルを参照することがあります。つまり、まだ読み込んでいないオブジェクト内のオブジェクト名や関数名を参照することがあります。この場合、ターゲット内のブレークポイントは延期され、指定された名前に

一致するオブジェクトが読み込まれるまでアクティブ (有効) になりません。オブジェクトが読み込まれると、このブレークポイントは自動的に有効になります。共用ライブラリ内に定義されたシンボル上のブレークポイントを設定する場合、アドレスは実際のシンボル定義ではなく、対応する PLT (Procedure Linkage Table) エントリを参照する可能性があるので、必ず、アドレス式ではなく、シンボル名を使用する必要があります。PLT エントリ上で設定されたブレークポイントは、そのPLT エントリが後で実際のシンボル定義に解釈されると、実行時リンクエディタによって上書きされることがあります。`-d`、`-D`、`-e`、`-s`、`-t`、`-T`、`-c`、および `-n` オプションは、`::evset` dcmd で使用するのと同じ意味になります (この節の後半を参照)。dcmd の `:b` 形式を使用した場合、ブレークポイントは dcmd の前にある式で指定した仮想アドレスだけに設定されます。`:b` dcmd の後ろにある引数は連結され、コールバック文字列となります。この文字列にメタキャラクタが含まれる場合、文字列を引用符で囲む必要があります。

function `::call [` *arg* `... ]`
`kmdb` のみを使用している場合、オペレーティングシステムカーネルに定義されている、指定の *function* を呼び出します。*function* 式は、既知のいずれかのカーネルモジュールのシンボルテーブルに定義されている関数のアドレスと一致する必要があります。式の引数を指定すると、それらの引数は値によって渡されます。文字列引数を指定すると、それらの引数は参照によって渡されます。

注 – `::call` コマンドは、十分に注意して使用する必要があり、本稼働システムには絶対に適用してはなりません。オペレーティングシステムカーネルは、指定された関数を実行するために実行を再開しません。したがって、呼び出されている関数は任意のカーネルサービスを決して利用してはならず、いかなる理由でもブロックしてはなりません。このコマンドを使用するときは、呼び出す関数の副作用を十分に認識している必要があります。

`::cont` [*SIG*]
`:c` [*SIG*]
デバッガを中断し、ターゲットプログラムを継続し、指定されたソフトウェアイベントが発生したあとにターゲットプログラムが終了または停止するまで待機します。`-o nostop` オプションを有効にしてデバッガを動作中のプログラムに接続しており、ターゲットがすでに動作している場合、この dcmd は単に、指定されたソフトウェアイベントが発生したあとにターゲットプログラムが終了または停止するまで待機します。オプションのシグナルの名前または番号 (`signal (3HEAD)` のマニュアルページを参照) を引数として指定した場合、このシグナルはターゲットの実行を再開するための一部として、すぐにターゲットに送られます。`SIGINT` シグナルを追跡している場合、ユーザー定義割り込み文字 (通常は ^C) を入力すると、非同期的にデバッガに制御を戻すことができます。この `SIGINT` シグナルは自動的に消去され、次回ターゲットが継続される場合、ターゲットにはこのシグナルは送られません。現在、ターゲットプログラムが動作していない場合、`::cont` は、`::run` を実行した場合と同じように、新しいプログラムの実行を起動します。

addr ::delete [*id* | all]
addr :d [*id* | all]
指定された ID 番号のイベント指定子を削除します。*id* 番号引数はデフォルトで 10 進数として解釈されます。オプションのアドレスを dcmd の前に指定した場合、指定された仮想アドレスに関連するすべてのイベント指定子 (つまり、そのアドレスに影響するすべてのブレークポイントまたはウォッチポイント) が削除されます。特別な引数「all」を指定した場合、すべてのイベントが削除されます。ただしスティッキー (T フラグ) がマークされたイベントは除きます。::events dcmd はイベント指定子の現在のリストを表示します。

::events [-av]
$b [-av]
ソフトウェアイベント指定子のリストを表示します。各イベント指定子には一意の ID 番号が割り当てられ、この ID 番号を使用すると、後でイベント指定子を削除または変更できます。デバッガはまた、独自の内部イベントを追跡用に有効にできます。このようなイベントが表示されるのは、-a オプションを指定した場合だけです。-v オプションを指定した場合、たとえば、指定子がアクティブになっていない理由などの、より詳細な情報が表示されます。次に、::events dcmd の出力例を示します。

```
> :

:

events
   ID S TA HT LM Description                              Action
----- - -- -- -- ---------------------------------------- -------------
[ 1 ] - T   1  0 stop on SIGINT                           -
[ 2 ] - T   0  0 stop on SIGQUIT                          -
[ 3 ] - T   0  0 stop on SIGILL                           -
 ...

[ 11] - T   0  0 stop on SIGXCPU                          -
[ 12] - T   0  0 stop on SIGXFSZ                          -
[ 13] -     2  0 stop at libc‘printf                      :

:

echo printf
>
```

次に、各カラムの意味を説明します。::help events を使用しても、この情報の要約を表示できます。

ID　　イベント指定子の識別子。この識別子は、指定子が有効である場合は角括弧「[]」で囲まれ、指定子が無効である場合は丸括弧「()」

で囲まれ、指定された指定子に一致するイベントが発生したあとにターゲットプログラムが現在停止している場合は山括弧「<>」で囲まれます。

S　イベント指定子の状態。この状態は次のシンボルのうちの 1 つです。

-	イベント指定子はアイドル状態です。どのターゲットプログラムも動作していないとき、すべての指定子はアイドル状態になります。ターゲットプログラムが動作しているとき、評価できない指定子はアイドル状態になることがあります。(たとえば、共用オブジェクトがまだ読み込まれておらず、共用オブジェクト内のブレークポイントが延期されている場合)。
+	イベント指定子はアクティブです。ターゲットが継続されると、この種類のイベントがデバッガによって検出されます。
*	イベント指定子は作動準備されてます。この状態は、ターゲットが現在動作しており、この種類のイベントが監視されていることを意味します。この状態を表示できるのは、`-o nostop` オプションを使用してデバッガを動作中のプログラムに接続している場合だけです。
!	オペレーティングシステムのエラーのために、イベント指定子は作動準備されていません。`::events -v` オプションを使用すると、監視が失敗した理由についてより詳細な情報が表示されます。

TA　Temporary、Sticky、および Automatic というイベント指定子のプロパティー。次のシンボルのうちの 1 つまたは複数が表示されます。

t	このイベント指定子は一時的であり、次回ターゲットが停止すると、一致するかどうかに関わらず削除されます。
T	このイベント指定子はスティッキーであり、`::delete all` または `:z` では削除されません。このイベント指定子を削除するには、その ID を明示的に `::delete` に指定します。
d	このイベント識別子は、ヒット回数がヒット制限に到達すると自動的に無効になります。
D	このイベント識別子は、ヒット回数がヒット制限に到達すると自動的に削除されます。
s	このターゲットは、ヒット回数がヒット制限に到達すると自動的に停止します。

HT　現在のヒット回数。このカラムには、当該イベント指定子が作成されたあとに、対応するソフトウェアイベントがターゲット内で発生した回数が表示されます。

LM　現在のヒット制限。このカラムには、自動無効、自動削除または自動停止が発生するときのヒット回数の制限が表示されます。このような動作を設定するには、::evset dcmd を使用します。

Description　指定された指定子に一致するソフトウェアイベントの種類の説明。

Action　対応するソフトウェアイベントが発生したときに実行されるコールバック文字列。このコールバックは、コマンドプロンプトに入力された場合と同じように実行されます。

id ::evset [+/-dDestT] [-c *cmd*] [-n *count*] *id* ...

1 つまたは複数のソフトウェアイベント指定子のプロパティーを変更します。プロパティーは、dcmd の前にあるオプションの式と dcmd の後ろにあるオプションの引数リストによって識別される指定子ごとに設定されます。基数が明示的に指定されない限り、引数リストは 10 進数の整数のリストとして解釈されます。::evset dcmd は次のオプションを認識します。

-d　ヒット回数がヒット制限に到達すると、イベント識別子を無効にします。+d 形式のオプションを指定した場合、この動作は無効になります。イベント指定子が無効になると、もう一度有効になるまで、デバッガは対応する監視をすべて削除し、対応するソフトウェアイベントを無視します。-n オプションを指定しない場合、指定子はすぐに無効になります。

-D　ヒット回数がヒット制限に到達すると、イベント識別子を削除します。+D 形式のオプションを指定した場合、この動作は無効になります。-D オプションは -d オプションよりも優先されます。ヒット制限を設定するには、-n オプションを使用します。

-e　イベント指定子を有効にします。+e 形式のオプションを指定した場合、この動作は無効になります。

-s　ヒット回数がヒット制限に到達すると、ターゲットプログラムを停止します。+s 形式のオプションを指定した場合、この動作は無効になります。-s オプションを指定すると、デバッガは指定子のコールバックが実行されるたびに (ただし、N 回目の実行を除く (N は指定子のヒット制限の現在の値))、::cont が発行されたかのように動作します。-s オプションは -D オプションと -d オプションよりも優先されます。

-t　イベント指定子を「一時的」であるとマークします。一時的な指定子は、次回ターゲットが停止すると、指定した指定子に対応するソフトウェアイベントの結果によって停止したかどうかに関わらず、自動的に削除されます。+t 形式のオプションを指定した場合、「一時的」マーカーは削除されます。-t オプションは -T オプションよりも優先されます。

`-T` イベント指定子を「スティッキー」であるとマークします。スティッキーな指定子は、`::delete all` または `:z` では削除されません。スティッキーな指定子を削除するには、対応する ID を明示的な引数として `::delete` に指定します。`+T` 形式のオプションを指定した場合、「スティッキー」プロパティーは削除されます。イベント指定子のデフォルトセットには初めから「スティッキー」のマークが付いています。

`-c` 対応するソフトウェアイベントがターゲットプログラム内で発生するたびに、指定された *cmd* 文字列を実行します。現在のコールバック文字列を表示するには、`::events` を使用します。

`-n` ヒット制限の現在の値を *count* に設定します。ヒット制限を現在設定せずに、`-n` オプションと一緒に `-s` または `-D` オプションを指定していない場合、ヒット制限は 1 に設定されます。

`::help evset` を使用しても、この情報の要約を表示できます。

flt `::fltbp` [+/-dDestT] [`-c` *cmd*] [`-n` *count*] *flt* ...
指定されたマシン障害を追跡します。障害を特定するには、dcmd の前にオプションの障害番号を指定するか、dcmd の後ろに障害の名前または番号のリストを指定します (`<sys/fault.h>` を参照)。`-d`、`-D`、`-e`、`-s`、`-t`、`-T`、`-c`、および `-n` オプションは、`::evset` dcmd で使用するのと同じです。`::fltbp` コマンドは、ユーザープロセスのデバッグにのみ適用されます。

signal `:i`
ターゲットが動作中のユーザープロセスである場合、指定されたシグナルを無視して、透過的にターゲットにそのシグナルを送ります。指定されたシグナルの送り先を追跡しているイベント指定子はすべて、追跡対象イベントのリストから削除されます。デフォルトでは、無視されるシグナルは、デフォルトでプロセスがコアをダンプする原因となるシグナルセット (`signal(3HEAD)` を参照) の補集合に初期化されます (ただし、デフォルトで追跡される `SIGINT` を除く)。`:i` コマンドは、ユーザープロセスのデバッグにのみ適用されます。

`$i`
デバッガが無視して、ターゲットが直接処理するシグナルのリストを表示します。追跡対象シグナルについてのより詳細な情報を取得するには、`::events` dcmd を使用します。`$i` コマンドは、ユーザープロセスのデバッグにのみ適用されます。

`::kill`
`:k`
ターゲットが動作中のユーザープロセスである場合、強制的にターゲットを終了します。また、ターゲットが本来デバッガで `::run` を使用して作成されていた場合、デバッガを終了すると、ターゲットは強制的に終了されます。`::kill` コマンドは、ユーザープロセスのデバッグにのみ適用されます。

`$l`
ターゲットがユーザープロセスである場合、代表スレッドの `LWPID` を出力します。

`$L`
ターゲットがユーザープロセスである場合、ターゲット内にある各 LWP の `LWPID` を出力します。

`::next` [*SIG*]
`:e` [*SIG*]
ターゲットプログラムを 1 命令だけ実行しますが、サブルーチンの呼び出しまでは進めます。オプションのシグナルの名前または番号 (`signal (3HEAD)` のマニュアルページを参照) が引数として指定された場合、このシグナルはターゲットの実行を再開するための一部として、すぐにターゲットに送られます。どのターゲットプログラムも動作していない場合、`::next` は、`::run` を実行したかのように新しいプログラムを起動し、最初の命令で停止します。

`::run` [*args ...*]
`:r` [*args ...*]
指定された引数を使用して、新しいターゲットプログラムの実行を起動し、接続します。引数はシェルによって解釈されません。デバッガがすでに動作中のプログラムを検査している場合、`::release` が実行された場合と同じように、最初にこのプログラムから切断します。

[*signal*] `::sigbp` [+/-dDestT] [`-c` *cmd*] [`-n` *count*] *SIG ...*
[*signal*] `:t` [+/`-dDestT`] [`-c` *cmd*] [`-n` *count*] *SIG ...*
指定されたシグナルの送り先を追跡します。シグナルを特定するには、dcmd の前にオプションのシグナル番号を指定するか、dcmd の後ろにシグナルの名前または番号のリストを指定します (`signal(3HEAD)` を参照)。`-d`、`-D`、`-e`、`-s`、`-t`、`-T`、`-c`、および `-n` オプションは、`::evset` dcmd で使用するのと同じです。最初に、デフォルトでプロセスがコアをダンプする原因となるシグナルセット (`signal(3HEAD)` を参照) と `SIGINT` が追跡されます。`::sigbp` コマンドは、ユーザープロセスのデバッグにのみ適用されます。

`::step` [`branch` | `over` | `out`] [*SIG*]
`:s` *SIG*
`:u` *SIG*
ターゲットプログラムを 1 命令だけ実行します。オプションのシグナルの名前または番号 (`signal (3HEAD)` のマニュアルページを参照) を引数として指定し、ターゲットがユーザープロセスである場合、このシグナルはターゲットの実行を再開するための一部として、すぐにターゲットに送られます。オプションの `branch` 引数を指定した場合、プロセッサの制御フローを分岐する次の命令が出されるまで、ターゲットプログラムは実行を継続します。`::step branch` 機能は、該当するプロセッサ固有の機能が有効になっている x86 システム上で `kmdb` を使用している場合にのみ利用できます。`over` 引数を指定した場合、`::step` はサブルーチンの呼び出しまで進めます。`::step over` 引数は `::next` dcmd と同じです。オプションの `out` 引数を指定した場合、代表スレッドが現在の関数から戻ってくるまで、ター

ゲットプログラムは実行を継続します。どのターゲットプログラムも動作していない場合、`::step over` は、`::run` を実行したかのように新しいプログラムを起動し、最初の命令で停止します。`:s` dcmd は `::step` と同じです。`:u` dcmd は `::step out` と同じです。

[*syscall*] `::sysbp` [+/`-dDestT`] [`-io`] [`-c` *cmd*] [`-n` *count*] *syscall ...*
エントリを追跡するか、指定されたシステムコールから終了します。システムコールを特定するには、dcmd の前にオプションのシステムコール番号を指定するか、dcmd の後ろにシステムコールの名前または番号のリストを指定します (`<sys/syscall.h>` を参照)。`-i` オプションを指定した場合 (デフォルト)、イベント指定子はシステムコールごとにカーネルに入るトリガーとなります。`-o` オプションを指定した場合、イベント指定子はカーネルから終了するトリガーとなります。`-d`、`-D`、`-e`、`-s`、`-t`、`-T`、`-c`、および `-n` オプションは、`::evset` dcmd で使用するのと同じです。`::sysbp` コマンドは、ユーザープロセスのデバッグにのみ適用されます。

addr [,*len*] `::wp` [+/`-dDestT`] [`-rwx`] [`-ip`] [`-c` *cmd*] [`-n` *count*]
addr [,*len*] `:a` [*cmd...*]
addr [,*len*] `:p` [*cmd...*]
addr [,*len*] `:w` [*cmd...*]
指定された場所にウォッチポイントを設定します。監視される領域の長さをバイト数で設定するには、 dcmd の前に繰り返し回数を指定します。長さを明示的に設定しない場合、デフォルトは 1 バイトです。`::wp` dcmd を使用すると、読み取り (`-r` オプション)、書き込み (`-w` オプション)、または実行 (`-x` オプション) のアクセス権の任意の組み合わせでのトリガーが可能となるようにウォッチポイントを構成できます。`-d`、`-D`、`-e`、`-s`、`-t`、`-T`、`-c`、および `-n` オプションは、`::evset` dcmd で使用するのと同じです。x86 システムで `kmdb` を使用している場合のみ、`-i` オプションを使って、ウォッチポイントが入出力ポートのアドレス上に設定されなくてはならないことを指定できます。`kmdb` のみを使用している場合は、`-p` オプションを使って、指定されたアドレスが物理アドレスとみなされるべきであることを指定できます。`:a` dcmd は、指定されたアドレスに読み取り権のウォッチポイントを設定します。`:p` dcmd は、指定されたアドレスに実行権のウォッチポイントを設定します。`:w` dcmd は、指定されたアドレスに書き込み権のウォッチポイントを設定します。`:a`、`:p`、および `:w` dcmd の後ろにある引数は連結され、コールバック文字列となります。この文字列にメタキャラクタが含まれる場合、文字列を引用符で囲む必要があります。

`:z`
すべてのイベント指定子を追跡対象ソフトウェアイベントのリストから削除します。`::delete` を使用しても、イベント指定子は削除できます。

exec との対話

制御されているユーザープロセスが exec(2) を正常に実行するとき、デバッガの動作は ::set -o follow_exec_mode オプションで制御できます (151 ページの「コマンド行オプションの概要」を参照)。デバッガと対象プロセスのデータモデルが同じ場合、exec の後で、MDB が自動的にターゲットを継続するか、デバッガのプロンプトに戻るかは、stop モードか follow モードかによって決定されます。デバッガと対象プロセスのデータモデルが異なる場合、follow モードでは、MDB は自動的に MDB バイナリを適切なデータモデルで再度 exec して、プロセスに接続し直すので、exec から戻っても、プロセスは停止したままになります。このような再実行後、すべてのデバッガの状態が保存されるわけではありません。

32 ビットの対象プロセスが 64 ビットのプログラムを実行した場合、stop モードでは、コマンドプロンプトに戻りますが、この時点で 64 ビットデータモデルを使用しているので、対象プロセスを検査できません。デバッグを再開するには、::release -a dcmd を実行して、MDB を終了し、そして、mdb -p *pid* を実行して、64 ビットデバッガを対象プロセスに接続し直します。

64 ビットの対象プロセスが 32 ビットのプログラムを実行した場合、stop モードでは、コマンドプロンプトに戻りますが、新しいプロセスを検査する機能はかなり制限されます。組み込み dcmd はすべてマニュアルどおりに機能しますが、読み込み可能な dcmd は構造体のデータモデル変換を実行しないので、マニュアルどおりには機能しません。デバッグ機能を完全に復元するには、上記のように、デバッガを解放し、プロセスに接続し直す必要があります。

ジョブ制御との対話

ジョブ制御によって停止されている (つまり、SIGTSTP、SIGTTIN、または SIGTTOU で停止されている) ユーザープロセスにデバッガが接続されている場合、継続用の dcmd で継続しようとしても、そのプロセスは再び「動作中 (running)」に設定できません。対象プロセスが同じセッションのメンバーである場合 (つまり、MDB と同じ制御端末を共用している場合)、MDB は関連するプロセスグループをフォアグラウンドに持ってきて、SIGCONT でプロセスを継続し、そのプロセスをジョブ制御の停止から再開しようとします。このようなプロセスから MDB を切断すると、MDB はそのプロセスグループをバックグラウンドに戻してから終了します。対象プロセスが同じセッションのメンバーでない場合、MDB はそのプロセスを安全にフォアグラウンドに持ってくることができないので、MDB はデバッガに関してはそのプロセスを継続しますが、そのプロセスはジョブ制御によって停止されているままになります。この場合、MDB は警告を出力するので、ユーザーは適切なシェルから fg コマンドを実行して、プロセスを再開する必要があります。

プロセスの接続と解放

MDB が動作中のユーザープロセスに接続するとき、プロセスは停止し、継続用の dcmd の 1 つが適用されるまで、あるいは、デバッガが終了するまで停止したままになります。`-p` オプションでデバッガをプロセスに接続する前、あるいは、`::attach` または `:A` コマンドを発行する前に、`-o nostop` オプションを有効にしている場合、MDB はプロセスに接続しますが、プロセスを停止しません。プロセスが動作している間、(結果の整合性は失われるのにも関わらず) プロセスは通常どおりに監視でき、ブレークポイントや追跡用のフラグも有効にできます。プロセスが動作している間に `:c` または `::cont` dcmd を実行した場合、デバッガはプロセスが停止するまで待機します。どの追跡対象ソフトウェアイベントも発生しない場合、`:c` または `::cont` の後、割り込み文字 (通常は ^C) を入力すると、プロセスを強制的に停止し、デバッガに制御を戻すことができます。

`:R`、`::release`、`:r`、`::run`、`$q`、または `::quit` dcmd を実行した場合、あるいは、EOF またはシグナルなどの結果としてデバッガが終了した場合、MDB は現在動作中のプロセスを (もしあれば) 解放します。プロセスが本来デバッガで `:r` または `::run` を使用して作成されていた場合、そのプロセスは解放されると、`SIGKILL` が送信された場合と同じように強制的に終了されます。プロセスが MDB に接続される前にすでに動作していた場合、そのプロセスは解放されると、もう一度「動作中 (running)」に設定されます。プロセスを解放して停止および放棄したままにするには、`::release -a` オプションを使用します。

◆ ◆ ◆ 第 7 章

7 カーネル実行制御

この章では、kmdb の実行時に利用できる動作中のオペレーティングシステムカーネルの実行制御用の MDB 機能について説明します。kmdb は、特にカーネルの実行制御と動作中のカーネルのデバッグ用に作られた MDB バージョンです。kmdb を使用すると、mdb を使ってユーザープロセスを制御および監視するのと同じように、カーネルを制御および監視できます。カーネル実行制御機能には、各 CPU で実行されているカーネルスレッドを命令レベルで制御する機能が含まれているため、開発者はカーネルをステップ実行して、リアルタイムでデータ構造体を調べることができます。

mdb と kmdb は同じユーザーインタフェースを共有しています。第 6 章で説明している実行制御機能はすべて kmdb でも利用でき、ユーザープロセスを制御するためのコマンドセットと同じものです。カーネルの状態を調べるためのコマンド (第 3 章および第 5 章を参照) も、kmdb の使用時に利用できます。最後に、特に指定がない限り、Solaris カーネルの実装に固有のコマンド (第 8 章を参照) も利用できます。この章では、kmdb に固有の残りの機能について説明します。

ブート、読み込み、読み込み解除

カーネルの起動を簡単にデバッグできるように、ブートプロセスの初期段階で、制御がカーネル実行時リンカー (krtld) からカーネルに渡る前に kmdb を読み込むことができます。 kmdb は、-k ブートフラグ、kmdb ブートファイル、または kadb ブートファイル (互換性のため) を使ってブート時に読み込むことができます。kmdb がブート時に読み込まれると、システムがそのあとリブートされるまで、デバッガを読み込み解除することはできません。ブートの初期段階では、すぐに利用できない機能がいくつかあります。特に、デバッグ用のモジュールは、カーネルモジュールサブシステムが初期化されるまで読み込まれません。プロセッサに固有の機能は、カーネルがプロセッサ識別処理を完了するまで有効になりません。

-k オプションを使ってシステムをブートすると、ブートプロセス時に kmdb が自動的に読み込まれます。-d ブートオプションを使用すると、カーネルを開始する前にデバッガのブレークポイントを要求できます。この機能は、ほかのカーネルと同様に

デフォルトのカーネルでも有効です。たとえば、kmdb を使って SPARC システムをブートし、デバッガに即時エントリを要求するには、次のいずれかのコマンドを入力します。

```
ok boot -kd
ok boot kmdb -d
ok boot kadb -d
```

同じ方法で x86 システムをブートするには、次のいずれかのコマンドを入力します。

```
Select (b)oot or (i)nterpreter:

b -kd
Select (b)oot or (i)nterpreter:

b kmdb -d
Select (b)oot or (i)nterpreter:

b kadb -d
```

kmdb を使って SPARC システムをブートし、64 ビットの代替カーネルを読み込むには、次のコマンドを入力します。

```
ok boot kernel.

test/sparcv9/unix -k
```

kmdb を使って x86 システムをブートし、64 ビットの代替カーネルを読み込むには、次のコマンドを入力します。

```
Select (b)oot or (i)nterpreter:

b kernel.

test/amd64/unix -k
```

ブートファイルが文字列 kmdb または kadb に設定され、代替カーネルをブートする場合は、-D オプションを使用してブートするカーネルの名前を指定します。この方法で SPARC システムをブートするには、次のコマンドを入力します。

```
ok boot kmdb -D kernel.

test/sparcv9/unix
```

この方法で 32 ビットの x86 システムをブートするには、次のコマンドを入力します。

```
Select (b) or (i)nterpreter:

b kmdb -D kernel.

test/unix
```

この方法で 64 ビットの x86 システムをブートするには、次のコマンドを入力します。

```
Select (b) or (i)nterpreter:

b kmdb -D kernel.

test/amd64/unix
```

すでにブートされているシステムをデバッグするには、mdb -K オプションを使用して kmdb を読み込み、カーネルの実行を停止します。この方法でデバッガが読み込まれると、あとで読み込み解除できます。デバッグが完了して kmdb を読み込み解除するには、::quit dcmd に -u オプションを指定します。あるいは、コマンド mdb -U を使用してオペレーティングシステムの実行を再開します。

端末処理

kmdbでは、対話処理に必ずシステムコンソールが使用されます。kmdb は、次の規則に従って適切な端末を決定します。

- デバッグされるシステムが付属のキーボードとモニターをコンソールとして使用し、デバッガがブート時に読み込まれる場合は、プラットフォームアーキテクチャーとコンソール端末の設定に基づいて端末タイプが自動的に決定されます。
- デバッグされるシステムがシリアルコンソールを使用し、デバッガがブート時に読み込まれる場合は、デフォルトの端末タイプ vt100 が使用されます。
- コンソール上で mdb -K を実行してデバッガを読み込む場合は、$TERM 環境変数の値が端末タイプとして使用されます。
- コンソール以外の端末で mdb -K を実行してデバッガを読み込む場合は、システムコンソールのログインプロンプトで使用するように構成された端末タイプが使用されます。

端末タイプを表示するには、kmdb 内部から ::term dcmd を実行します。

デバッガのエントリ

オペレーティングシステムカーネルは、ブレークポイントに達したとき、またはほかの実行制御設定 (第 6 章を参照) に従って、暗黙的に実行を停止し、kmdb に入ります。kmdb に入ることを明示的に要求するには、mdb -K オプションまたは適切なキーボードのブレークシーケンスを使用します。SPARC システムのコンソールでは、STOP + A キーシーケンスを使ってブレークを送信し、kmdb に入ります。x86 システムのコンソールでは、F1 + A キーシーケンスを使ってブレークを送信し、kmdb に入ります。Solaris システムでエスケープシーケンスをカスタマイズするには、kbd コマンドを使用します。シリアルコンソールを持つシステムで kmdb に入るには、該当するシリアルコンソールコマンドを使用してブレークシーケンスを送信します。

プロセッサ固有の機能

一部の kmdb 機能は個々のプロセッサアーキテクチャーに固有のものです。たとえば、各種の x86 プロセッサでは、ほかのプロセッサアーキテクチャーでは見られないハードウェア分岐トレース機能をサポートしています。プロセッサ固有の機能にアクセスするには、それをサポートしているシステムにのみ存在するプロセッサ固有の dcmd を使用します。プロセッサ固有のサポートを利用できるかどうかは、::status dcmd の出力に示されます。デバッガは、カーネルに基づいてプロセッサタイプを判断します。このため、デバッガが特定のプロセッサアーキテクチャーの機能を提供できる場合でも、カーネルによってプロセッサの識別が完了するまで、このサポートは有効になりません。

◆ ◆ ◆ 第 8 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あとに指定されます。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 – MDB によってカーネルの実装の詳細が公開されますが、これらは随時変更されることがあります。このマニュアルで説明する Solaris カーネルの実装は、このマニュアルの発行時点のものです。このマニュアルでモジュール、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、過去または将来の Solaris リリースに対しては正しくない場合や適用できない場合があります。

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 9 章でさらに詳細に説明します。

dcmds

thread `::allocdby`
指定されたカーネルスレッドのアドレスを使用して、そのスレッドが割り当てたメモリーのリストを新しい順に出力します。

bufctl `::bufctl` [`-a` *address*] [`-c` *caller*] [`-e` *earliest*] [`-l` *latest*] [`-t` *thread*]
指定された bufctl *address* についての *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。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。`-a` オプションは、buctl の対応するバッファー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、指定されたもっとも早い時刻表示と同じかそれ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、指定されたもっとも遅い時刻表示と同じかそれ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るように指定します。

[*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`
指定されたカーネルスレッドのアドレスを使用して、そのスレッドが解放したメモリーのリストを新しい順に出力します。

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内のイベントを表示します。イベントは新しい順に (つまり、最新のイベントから) 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、バッファーアドレス、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メモリー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するこの出力は、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、103 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。特に、::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ー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	指定された kthread_t 構造体のアドレスを開始点として使用して、当該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反復適用します。
bufctl	指定された kmem_cache_t 構造体のアドレスを開始点として使用して、このキャッシュに関連する割り当てられた bufctl のセットに対して反復適用します。
freectl	指定された kmem_cache_t 構造体のアドレスを開始点として使用して、このキャッシュに関連する空き bufctl のセットに対して反復適用します。
freedby	指定された kthread_t 構造体のアドレスを開始点として使用して、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反復適用します。
freemem	指定された kmem_cache_t 構造体のアドレスを開始点として使用して、このキャッシュに関連する空きバッファーのセットに対して反復適用します。
kmem	指定された kmem_cache_t 構造体のアドレスを開始点として使用して、このキャッシュに関連する割り当てられたバッファーのセットに対して反復適用します。
kmem_cache	アクティブな kmem_cache_t 構造体のセットに対して反復適用します。この構造体は、<sys/kmem_impl.h> で定義されます。

kmem_cpu_cache　指定された kmem_cache_t 構造体のアドレスを開始点として使用して、このキャッシュに関連する CPU ごとの kmem_cpu_cache_t 構造体に対して反復適用します。この構造体は、<sys/kmem_impl.h> で定義されます。

kmem_slab　指定された kmem_cache_t 構造体のアドレスを開始点として使用して、関連する kmem_slab_t 構造体のセットに対して反復適用します。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 のセットに対して反復適用します。

leak　指定された bufctl 構造体のアドレスを使用して、同様な割り当てスタックトレースを持つ、リークが発生したメモリーバッファーに対応する bufctl 構造体のセットに対して反復適用します。leak walker を使用する前には、::findleaks dcmd を適用してメモリーリークを発見しておく必要があります。

leakbuf　指定された bufctl 構造体のアドレスを使用して、同様な割り当てスタックトレースを持つ、リークが発生したメモリーバッファーに対応するバッファーアドレスのセットに対して反復適用します。leakbuf walker を使用する前には、::findleaks dcmd を適用してメモリーリークを発見しておく必要があります。

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。

dcmds

::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　ロックマネージャー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 vnode 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　アクティブなブロック I/O 転送構造体 (buf_t 構造体) のセットに対して反復適用します。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)。

仮想メモリー

この節では、カーネル仮想メモリー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。

dcmds

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。

[*address*] ::memlist [-aiv]　指定された memlist 構造体または既知の memlist 構造体の 1 つを表示します。memlist アドレスとオプションを指定しないか、あるいは、-i オプションを指定すると、物理的にインストールされているメモリーを表す memlist が表示されます。-a オプションを指定すると、利用できる物理メモリーを表す memlist が表示されます。-v オプションを指定すると、利用できる仮想メモリーを表す memlist が表示されます。

::memstat　システム全体のメモリー使用状況の要約を表示します。システムメモリーの合計とともに、さまざまなクラスのページ (カーネル、匿名メモリー、実行可能ファイルとライブラリ、ページキャッシュ、および空きリスト) が消費しているシステムメモリーの量とパーセンテージが表示されます。

[*address*] ::page　指定された page_t のプロパティーを表示します。page_t アドレスを指定しないと、dcmd はすべてのシステムページのプロパティーを表示します。

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。
[*address*] ::swapinfo	アクティブな swapinfo 構造体すべて、あるいは指定された swapinfo 構造体についての情報を表示します。各構造体の v ノード、ファイル名、および統計が表示されます。
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	指定された anon_map 構造体のアドレスを開始点として使用して、関連する anon 構造体のセットに対して反復適用します。anon マップ実装は、<vm/anon.h> で定義されます。
memlist	指定された memlist 構造体のスパンに対して反復適用します。この walker を ::memlist dcmd と組み合わせて使用すると、各スパンを表示できます。
page	すべてのシステムの page 構造体に対して反復適用します。明示的なアドレスを walk に指定すると、v ノードのアドレスとして解釈され、walker は v ノードに関連するページだけに対して反復適用します。
seg	指定された as_t 構造体のアドレスを開始点として使用して、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) のセットに対して反復適用します。seg 構造体は、<vm/seg.h> で定義されます。
swapinfo	アクティブな swapinfo 構造体のリストに対して反復適用します。この walker は ::swapinfo dcmd と組み合わせて使用できます。

CPU とディスパッチャー

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

dcmds

::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。オプションの CPU 番号または CPU 構造体アドレスを dcmd 名の前に指定すると、指定した CPU の情報だけ

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反復適用します。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ー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。

dcmds

address ::binding_hash_entry
　指定されたカーネル名とメジャー番号のバインディングハッシュテーブルエントリ (構造体 bind) のアドレスを使用して、ノードバインディング名、メジャー番号、および次の要素へのポインタを表示します。

::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。インスタンスごとに 1 つのエントリの出力、つまり、先頭から、構造体 dev_info へのポインタ ($<devinfo または ::devinfo で表示可能)、ドライバ名、インスタンス番号、そのインスタンスに関連するドライバとシステムのプロパティーが表示されます。

address ::devinfo [-q]
　devinfo ノードに関連するシステムプロパティーおよびドライバプロパティーを出力します。-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が表示されます。

address ::devinfo2driver
　devinfo ノードに関連するドライバ (もしあれば) の名前を出力します。

[*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、各ノードに関連するプロパティーが表示されます。

[*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*
指定されたデバイスドライバ名を使用して、そのメジャー番号を表示します。

[*address*] ::softstate [*instance-number*]
指定された 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使用して、そのインスタンスのソフトの状態を表示します。

walker

binding_hash
指定されたカーネルバインディングハッシュテーブルエントリの配列 (構造体 bind **) のアドレスを使用して、ハッシュテーブル内のすべてのエントリを調べて、各構造体 bind のアドレスを戻します。

devinfo
最初に、指定された devinfo の親に対して反復適用し、それらを最下位から世代順に戻します。次に、指定された devinfo 自身を戻します。その次に、指定された devinfo の子に対して、世代順に最上位から最下位まで反復適用します。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 を戻し、次に、指定された devinfo の子に対して、世代順に最上位から最下位まで反復適用します。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反復適用し、次に、指定された devinfo を戻します。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 の兄弟に対して反復適用します。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反復適用します。この構造体は、<sys/autoconf.h> で定義されます。

softstate
指定された softstate ポインタ (ddi_soft_state_init(9F) を参照) を使用して、ドライバ状態構造体への NULL 以外のポインタをすべて表示します。

softstate_all　　指定された softstate ポインタ (ddi_soft_state_init(9F) を参照) を使用して、ドライバ状態構造体へのポインタをすべて表示します。未使用のインスタンスのポインタが NULL になることに注意してください。

STREAMS

この節では、カーネル開発者やサードパーティーの 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。

dcmds

address ::mblk2dblk
　指定された mblk_t のアドレスを使用して、対応する dblk_t のアドレスを出力します。

[*address*] ::mblk_verify
　1 つまたは複数のメッセージブロックの整合性を確認します。アドレスを明示的に指定すると、そのメッセージブロックの整合性がチェックされます。アドレスを明示的に指定しないと、アクティブなメッセージブロックすべての整合性がチェックされます。この dcmd は、検出したが無効であるメッセージブロックについても出力します。

address ::queue [-v] [-f *flag*] [-F *flag*] [-s *syncq*]
　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ーが表示されます。-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定します。-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定します。-m オプションは、待ち行列に関連する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。-s オプションは、待ち行列に関連する syncq_t が指定された syncq_t アドレスに一致する必要があることを指示します。

address ::q2syncq
　指定された queue_t のアドレスを使用して、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
　指定された queue_t のアドレスを使用して、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
指定された queue_t のアドレスを使用して、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
指定された queue_t のアドレスを使用して、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
指定された STREAM ヘッドを表す stdata_t 構造体のアドレスを使用して、カーネル STREAM データ構造体のイメージ図を表示します。読み取りと書き込みの待ち行列へのポインタ、バイト数、各モジュールのフラグが表示され、さらに、指定された待ち行列に関する追加情報が余白に表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、queue_t の種々のプロパティーが表示されます。-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定します。-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定します。-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定します。-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定します。

address ::syncq2q
指定された syncq_t のアドレスを使用して、対応する queue_t データ構造体のアドレスを出力します。

walker

b_cont　指定された mblk_t のアドレスを使用して、b_cont ポインタに従いながら、関連するメッセージ構造体のセットに対して反復適用します。b_cont ポインタは、指定されたメッセージブロックを、同じメッセージ上にある次の関連するメッセージブロックにリンクするために使用されます。メッセージブロックの詳細については、msgb(9S) のマニュアルページを参照してください。

b_next　指定された mblk_t のアドレスを使用して、b_next ポインタに従いながら、関連するメッセージ構造体のセットに対して反復適用します。b_next ポインタは、指定されたメッセージブロックを、指定された待ち行列上にある次の関連するメッセージブロックにリンクするために使用されます。メッセージブロックの詳細については、msgb(9S) のマニュアルページを参照してください。

qlink　指定された queue_t 構造体のアドレスを使用して、q_link ポインタを使用しながら、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　指定された queue_t 構造体のアドレスを使用して、q_next ポインタを使用しながら、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　指定された stdata_t 構造体のアドレスを使用して、読み取り側待ち行列構造体のリストを調べます。

writeq　指定された stdata_t 構造体のアドレスを使用して、書き込み側待ち行列構造体のリストを調べます。

ネットワーク関連機能

この節では、中心となるカーネルネットワーキングスタックプロトコルをデバッグするときに役に立つ dcmd と walker について説明します。

dcmds

address ::mi [- p] [-d | -m]
　指定されたカーネル MI_O を使用して、MI_O またはそのペイロードをフィルタリングおよび表示します。-p オプションを指定すると、MI_O に対応するペイロードのアドレスが表示され、指定しないと、MI_O 自身が表示されます。フィルタ -d または -m を指定すると、dcmd はそれぞれ、デバイスまたはモジュール MI_O オブジェクトをフィルタリングできます。

::netstat [-av] [-f inet | inet6 | unix] [-P tcp | udp]
　ネットワークの統計とアクティブな接続を表示します。-a オプションを指定すると、すべてのソケットの状態が表示されます。-v オプションを指定すると、より詳細な出力が表示されます。-f オプションを指定すると、指定したアドレスファミリに関連する接続だけが表示されます。-P オプションを指定すると、指定したプロトコルに関連する接続だけが表示されます。

[*address*] ::sonode [- f inet | inet6 | unix | *id*] [-t stream | dgram | raw | *id*] [-p *id*]
　sonode オブジェクトをフィルタリングおよび表示します。アドレスを指定しないと、AF_UNIX ソケットのリストが表示され、指定すると、指定した sonode だけが表示されます。-f オプションを指定すると、指定したファミリのソケットだけが表示されます。-t オプションを指定すると、指定した種類の sonode だけが表示されます。-p オプションを指定すると、指定したプロトコルのソケットだけが表示されます。

[*address*] ::tcpb [- av] [-P v4 | v6]
　tcpb オブジェクトをフィルタリングおよび表示します。アドレスを指定しないと、すべての接続が検査され、指定すると、指定した tcpb だけがフィルタリング

および表示されます。-a オプションを指定すると、アクティブな接続だけがフィルタリングされ、-P オプションを指定すると、TCP の IPv4 または IPv6 接続をフィルタリングできます。tcpb dcmd は TCP 接続のフィルタリング機能を持つため、IPv4 の接続を促進する状態の中で IPv6 の TCP も接続する場合、-P フィルタを指定すると、::netstat とほぼ同じように、接続は IPv4 と IPv6 の両方として認識されます。dcmd をフィルタとして使用せずに -v オプションを指定すると、dcmd からの出力は詳細になります。

walker

ar　指定された ar のアドレスを使用して、指定された ar から最後の ar までのすべての ar オブジェクトを調べます。アドレスを指定しないと、すべての ar オブジェクトが調べられます。

icmp　指定された icmp のアドレスを使用して、指定された icmp から最後の icmp までのすべての icmp オブジェクトを調べます。アドレスを指定しないと、すべての icmp オブジェクトが調べられます。

ill　指定されたインタフェースリンク層構造体 (ill) のアドレスを使用して、指定された ill から最後の ill までのすべての ill オブジェクトを調べます。アドレスを指定しないと、すべての ill オブジェクトが調べられます。

ipc　指定された ipc のアドレスを使用して、指定された ipc から最後の ipc までのすべての ipc オブジェクトを調べます。アドレスを指定しないと、すべての ipc オブジェクトが調べられます。

mi　指定された MI_O のアドレスを使用して、この MI 内にあるすべての MI_O を調べます。

sonode　指定された AF_UNIX sonode を使用して、指定された sonode から始まる関連する AF_UNIX sonode のリストを調べます。アドレスを指定しないと、すべての AF_UNIX ソケットのリストが調べられます。

tcpb　指定された tcpb を使用して、指定された tcpb から最後の TCP 接続までのすべての TCP 接続を調べます。アドレスを指定しないと、すべての tcpb オブジェクトが調べられます。

udp　指定された udp のアドレスを使用して、指定された udp から最後の udp までのすべての udp オブジェクトを調べます。アドレスを指定しないと、すべての udp オブジェクトが調べられます。

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。

dcmds

process `::fd` *fd-num*
指定されたプロセスに関連するファイル記述子 *fd-num* に対応する `file_t` アドレスを出力します。このプロセスを指定するには、その `proc_t` 構造体の仮想アドレスを使用します。

thread `::findstack` [*command*]
指定されたカーネルスレッド (`kthread_t` 構造体の仮想アドレスによって識別される) に関連するスタックトレースを出力します。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。

`::pgrep` [`-x`] [`-n`|`-o`] *regexp*
名前が *regexp* 正規表現パターンに適合するプロセスのプロセス情報を表示します。`::pgrep` dcmd は `pgrep(1)` コマンドに似ています。`::pgrep` dcmd は、すべてのプロセスを対象にしたパターンマッチングに使用されます。`-n` オプションを指定すると、パターンに適合する最新のプロセスだけが表示されます。`-o` オプションを指定すると、パターンに適合するもっとも古いプロセスだけが表示されます。`-x` オプションを指定すると、検索パターンと正確に一致する名前を持つプロセスだけが表示されます。

`kmdb(1)` では、`::pgrep` に使用する *regexp* は、英数字のプレーンテキスト文字列でなければなりません。

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` [`-q`]
指定されたプロセスアドレスに該当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。`-q` オプションを指定すると、出力が省略形式で表示されるので、処理時間が短くなります。

[*address*] ::ps [- fltTP]
指定されたプロセスまたはすべてのアクティブなシステムプロセスに関連する情報の要約を ps(1) に似た形式で出力します。-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、各プロセスに関連する LWP が出力されます。-t オプションを指定すると、各プロセスの LWP に関連するカーネルスレッドが出力されます。-T オプションを指定すると、各プロセスに関連するタスク ID が出力されます。- P オプションを指定すると、各プロセスに関連するプロジェクト ID 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。

address ::task
アクティブなカーネルタスク構造体とそれに関連する ID 番号および属性のリストを出力します。プロセスタスク ID の詳細については、settaskid(2) を参照してください。

[*address*] ::thread [-bdfimps]
指定された kthread_t カーネル構造体のプロパティーを表示します。kthread_t のアドレスを指定しないと、すべてのカーネルスレッドのプロパティーが表示されます。この dcmd オプションを使用すると、どの出力カラムを表示するかを制御できます。オプションを指定しないと、-i オプションがデフォルトで有効になります。-b オプションを指定すると、スレッドのターンスタイルとブロッキング同期オブジェクトに関連する情報が表示されます。-d オプションを指定すると、スレッドのディスパッチャーの優先順位、バインディング、および最終ディスパッチ時刻が表示されます。-f オプションを指定すると、状態が TS_FREE であるスレッドが出力から削除されます。-i オプションを指定すると (デフォルト)、スレッドの状態、フラグ、優先順位、および割り込み情報が表示されます。-m オプションを指定すると、ほかのすべての出力オプションがマージされて、単一の出力行に表示されます。-p オプションを指定すると、スレッドのプロセス、LWP、および資格ポインタが表示されます。-s オプションを指定すると、スレッドのシグナル待ち行列とシグナルの保留と保持を示すマスクが表示されます。

vnode ::whereopen
指定された vnode_t アドレスを使用して、現在この vnode をファイルテーブルに開いているプロセスの proc_t アドレスを出力します。

walker

file　指定された proc_t 構造体のアドレスを開始点として使用して、指定されたプロセスに関連する開いているファイル (file_t 構造体) のセットに対して反復適用します。file_t 構造体は、<sys/file.h> で定義されます。

proc　アクティブなプロセス (proc_t) 構造体に対して反復適用します。この構造体は、<sys/proc.h> で定義されます。

task　指定されたタスクポインタを使用して、指定されたタスクのメンバーであるプロセスの proc_t 構造体のリストに反復適用します。

thread　カーネルスレッド (kthread_t) 構造体のセットに対して反復適用します。大域 walk が起動されると、すべてのカーネルスレッドがこの walker によって戻されます。proc_t アドレスを開始点として使用することによって局所 walk が起動されると、指定したプロセスに関連するスレッドのセットが戻されます。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論については、マニュアルページの対応するセクション (9f) を参照してください。

dcmds

rwlock ::rwlock　指定された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) のアドレスを使用して、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 ::sobj2ts　同期オブジェクトのアドレスを対応するターンスタイルのアドレスに変換して、ターンスタイルのアドレスを出力します。

[*address*] ::turnstile　指定された turnstile_t のプロパティーを表示します。turnstile_t のアドレスを指定しないと、すべてのターンスタイルのプロパティーが表示されます。

[*address*] ::wchaninfo [-v]　指定された条件変数 (condvar(9F) のマニュアルページを参照) またはセマフォー (semaphore(9F) のマニュアルページを参照) のアドレスを使用して、このオブジェクト上で現在待機しているスレッドを表示します。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　指定された同期オブジェクト (たとえば、mutex(9F) または rwlock(9F)) のアドレスを使用して、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反復適用します。

wchan　指定された条件変数 (condvar(9F) のマニュアルページを参照) またはセマフォー (semaphore(9F) のマニュアルページを参照) のアドレスを使用して、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反復適用します。

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、ほか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。

dcmds

`::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` 構造体に対して反復適用します。この構造体は、`<sys/cyclic_impl.h>` で定義されます。

cyctrace　cyclic トレースバッファー構造体に対して反復適用します。この情報は、DEBUG カーネルだけで使用可能です。

タスク待ち行列

タスク待ち行列サブシステムは、カーネル内のさまざまなクライアントに汎用の非同期タスクスケジューリング機能を提供します。

dcmds

address `::taskq_entry`　指定された `taskq_entry` 構造体の内容を出力します。

walker

taskq_entry　指定された taskq 構造体のアドレスを使用して、taskq_entry 構造体のリストに対して反復適用します。

エラー待ち行列

エラー待ち行列サブシステムは、プラットフォームに固有なエラー処理コードに汎用の非同期エラーイベント処理機能を提供します。

dcmds

[*address*] ::errorq　指定されたエラー待ち行列に関連する情報の要約を表示します。アドレスを指定しないと、すべてのシステムエラー待ち行列に関連する情報が表示されます。各待ち行列のアドレス、名前、待ち行列の長さ、データ要素サイズ、および、さまざまな待ち行列統計が表示されます。

walker

errorq　システムエラー待ち行列のリストを調べて、各エラー待ち行列のアドレスを返します。

errorq_data　指定されたエラー待ち行列のアドレスを使用して、保留中の各エラーイベントデータバッファーのアドレスを戻します。

構成

この節では、システム構成データを確認するときに使用できる dcmd について説明します。

dcmds

::system　システム初期化中にカーネルが解析したときの system(4) 構成ファイルの内容を表示します。

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、セマフォー、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。

dcmds

`::ipcs [-l]`	既知のメッセージ待ち行列、セマフォー、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 `::msg [- l] [-t` *type*`]`	指定されたメッセージ待ち行列要素 (構造体 `msg`) のプロパティーを表示します。`-l` オプションを指定すると、メッセージの raw コンテンツが 16 進数と ASCII 形式で表示されます。`-t` オプションを指定すると、出力をフィルタリングして、指定された種類のメッセージだけを表示します。このオプションは、`msgqueue` walker の出力を `::msg` にパイプで渡すときに便利です。
id `::msqid [- k]`	指定されたメッセージ待ち行列 IPC 識別子を対応するカーネル実装構造体へのポインタに変換して、このカーネル構造体のアドレスを出力します。`-k` オプションを指定すると、一致させるメッセージ待ち行列キーとして *id* が代わりに解釈されます (`msgget(2)` を参照)。
[*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、より詳細な情報のリストが表示されます。
id `::semid [- k]`	指定されたセマフォー IPC 識別子を対応するカーネル実装構造体へのポインタに変換して、このカーネル構造体のアドレスを出力します。`-k` オプションを指定すると、一致させるセマフォーキーとして *id* が代わりに解釈されます (`semget(2)` を参照)。
[*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 (セマフォー識別子) 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。

id ::shmid [- k]	指定された共用メモリー IPC 識別子を対応するカーネル実装構造体へのポインタに変換して、このカーネル構造体のアドレスを出力します。-k オプションを指定すると、一致させる共用メモリーキーとして *id* が代わりに解釈されます (shmget(2) を参照)。
[*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。
msgqueue	現在指定されたメッセージ待ち行列に入っている message 構造体に対して反復適用します。
sem	セマフォー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。

dcmds

[*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。

dcmds

[*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行うカーネル実行時リンカーのデバッギングサポートについて説明します。

dcmds

[*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　指定された modctl 構造体のアドレスを使用して、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。

USB フレームワークのデバッギングサポート (uhci)

uchiモジュールは、USB (Universal Serial Bus) フレームワークのホストコントローラインタフェース部分にデバッギングサポートを提供します。

dcmds

address ::uhci_qh [- bd]　指定された USB UHCI コントローラの QH (Queue Head) 構造体のアドレスを使用して、この構造体の内容を出力します。-b オプションを指定すると、link_ptr チェインに対して反復適用され、見つかったすべての QH が出力されます。-d オプションを指定すると、element_ptr チェインに対して反復適用され、見つかったすべての TD が出力されます。

address ::uhci_td [- d]　指定された USB UHCI コントローラの TD (Transaction Descriptor) 構造体のアドレスを使用して、この構造体の内容を出力します。これは、Control TD と Interrupt TD だけに機能することに注意してください。-d オプションを指定すると、element_ptr チェインに対して反復適用され、見つかったすべての TD が出力されます。

walker

uhci_qh　指定された USB UHCI コントローラの QH (Queue Head) 構造体のアドレスを使用して、このような構造体のリストに対して反復適用します。

uhci_td　指定された USB UHCI コントローラの TD (Transaction Descriptor) 構造体のアドレスを使用して、このような構造体のリストに対して反復適用します。

USB フレームワークのデバッギングサポート (usba)

usba モジュールは、プラットフォームに依存しない USB (Universal Serial Bus) にデバッギングサポートを提供します。

dcmds

::usba_debug_buf	USB デバッギング情報バッファーを出力します。
::usba_clear_debug_buf	USB デバッギング情報バッファーを空にします。
[*address*] ::usba_device [- pv]	指定された usba_device 構造体のアドレスを使用して、要約情報を出力します。アドレスを指定しないと、usba_device 構造体の大域リストが調べられます。-p オプションを指定すると、当該デバイス上で開いているすべてのパイプについての情報が表示されます。-v オプションを指定すると、各デバイスのより詳細な情報が表示されます。
address ::usb_pipe_handle	指定された USB パイプハンドル構造体 (usba_ph_impl) のアドレスを使用して、このハンドルの要約情報を出力します。

walker

usba_list_entry	指定された usba_list_entry 構造体のアドレスを使用して、このような構造体のチェインに対して反復適用します。
usba_device	usba_device_t 構造体の大域リストを調べます。
usb_pipe_handle	指定された usba_device_t のアドレスを使用して、USB パイプハンドルを調べます。

x86: x86 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、x86 プラットフォーム用です。

dcmds

[*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 のバッファーに制限されます。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。

dcmds

[*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 のバッファーに制限されます。`-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ー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。クロスコール機能は、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反復適用します。
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反復適用します。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反復適用します。

◆ ◆ ◆ 第 9 章

9 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 – MDB によってカーネルの実装の詳細が公開されますが、これらは随時変更されることがあります。このマニュアルで説明する Solaris カーネルの実装は、このマニュアルの発行時点のものです。このマニュアルで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、過去または将来の Solaris リリースに対しては正しくない場合や適用できない場合があります。

はじめに - 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_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、次の行を /etc/system に追加します。

```
set kmem_flags=0xf
```

その後、システムを再起動する必要があります。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。システムを稼働状態に戻す前に、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ー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すると、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 savecore(1M) を手動で実行すると、あとで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: 5.10 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、UltraSPARC 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ー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、仮想記憶領域をほか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。

バッファーの状態

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ーの集まりです。これらのバッファーは、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、

バッファーのセットが含まれています。これらのバッファー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。残りのバッファーは割り当て済みです。つまり、そのバッファーへのポインタが、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ーへのポインタを保持していない場合には、そのバッファー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ーは、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、バッファー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、バッファー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。これを制御するには、`kmem_alloc(9F)` の「flag」パラメータを使用します。`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ー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、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) および 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、そのサイズが、そのキャッシュのバッファーサイズと次に小さいキャッシュのバッファーサイズの間にあるという要求を満たします。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ーのサイズは 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、バッファーの残りの 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、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。重要な点は、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です。これらは、キャッシュの作成時に、大域変数 kmem_flags 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、そのあと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ーのサイズです。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ー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ー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。次にその例を示します。

```
> ::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 データ構造体のもっと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。

- バッファー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
- 未使用バッファーの継続使用
- 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。

未使用バッファーの検査 (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ー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次に示します。

```
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        139d
0x70a9ae20:     70ae3200        d1befaed
0x70a9ae28:     deadbeef        deadbeef
0x70a9ae30:     deadbeef        deadbeef
0x70a9ae38:     deadbeef        deadbeef
0x70a9ae40:     feedface        feedface
0x70a9ae48:     70ae31a0        8440c54e
```

0x70a9add8 で始まるバッファーは、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、そのバッファー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ーが始まっています。それらの間の 0x70a9ae00 で始まる領域に割り当てバッファーがあります。

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、108 ページの「レッドゾーン (0xfeedface)」で説明します。

レッドゾーン (0xfeedface)

上記のバッファーには、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」コードがバッファー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。レッドゾーンのあとに追加の情報があります。このデータの内容はほかの要因によって異なります (112 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 9–1 に、この情報の要約を示します。

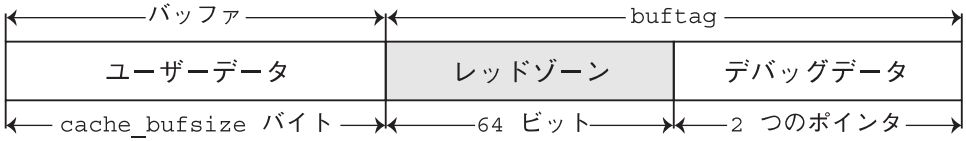

図 9–1 レッドゾーン

バッファーのキャッシュに KMF_AUDIT、KMF_DEADBEEF、または KMF_REDZONE 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ーに 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ーに分解すると、次のように簡単になります。

```
0x70a9add8:     deadbeef        deadbeef  \
0x70a9ade0:     deadbeef        deadbeef   +- User Data (free)
0x70a9ade8:     deadbeef        deadbeef  /
0x70a9adf0:     feedface        feedface  -- REDZONE
0x70a9adf8:     70ae3260        8440c68e  -- Debugging Data

0x70a9ae00:     5               4ef83     \
0x70a9ae08:     0               0          +- User Data (allocated)
0x70a9ae10:     1               bbddcafe  /
0x70a9ae18:     feedface        139d    -- REDZONE
0x70a9ae20:     70ae3200        d1befaed  -- Debugging Data

0x70a9ae28:     deadbeef        deadbeef  \
0x70a9ae30:     deadbeef        deadbeef   +- User Data (free)
0x70a9ae38:     deadbeef        deadbeef  /
0x70a9ae40:     feedface        feedface  -- REDZONE
0x70a9ae48:     70ae31a0        8440c54e  -- Debugging Data
```

0x70a9add8 と 0x70a9ae28 の未使用バッファーでは、レッドゾーンには 0xfeedfacefeedface が使用されています。これは、バッファー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ーでは、状況は異なります。101 ページの「アロケータの基礎」で説明したとおり、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ーのサイズは、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ーのサイズは、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マーカーとしてレッドゾーンバイトを調整して強制的にバッファー境界を合わせます。

```
0x70a9ae00:     5               4ef83     \
0x70a9ae08:     0               0          +- User Data (allocated)
0x70a9ae10:     1               bbddcafe  /
0x70a9ae18:     feedface        139d    -- REDZONE
0x70a9ae20:     70ae3200        d1befaed  -- Debugging Data
```

0x70a9ae18 にある 0xfeedface の後ろには、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、実際にはバッファー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ーのサイズを知るには、次の公式を使用します。

```
size = redzone_value / 251
```

たとえば、この例では次のようになります。

```
size = 0x139d / 251 = 20 bytes.
```

これは、要求されたバッファーのサイズが 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 進数パターン 0xbb です。これは予想通り、0x729084e4 (0x729084d0 + 0t20) に存在しています。

図 9–2 kmem_alloc(9F) バッファーの例

図 9–3 に、メモリー配置の一般的形式を示します。

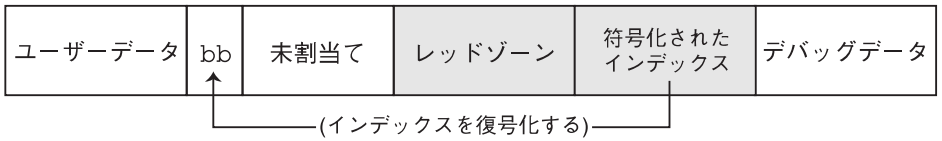

図 9–3 レッドゾーンバイト

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 9–4 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。

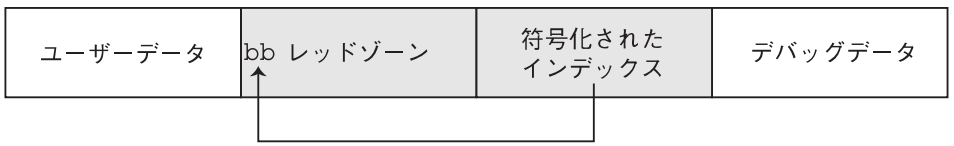

図 9–4 レッドゾーンの先頭にあるレッドゾーンバイト

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とき、アロケータは公式 (251 * 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、アロケータは (size % 251 == 1) になるかをテストすることによって、サイズが有効かどうかをチェックできるので、この 1 の加算 (「+1」) は貴重な役割を果たします。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、 0xbaddcafe と書いてありました。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、バッファー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、問題のバッファーに 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、対応するバッファー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「バッファー制御」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ーの 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 9–5 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。

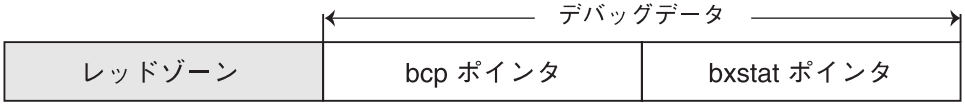

図 9–5 buftag の追加のデバッギングデータ

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。バッファーが割り当て済みの場合には、`bcp XOR bxstat = 0xa110c8ed` (「allocated」) になります。バッファーが未使用の場合には、`bcp XOR bxstat = 0xf4eef4ee` (「freefree」) になります。

注 – 107 ページの「未使用バッファーの検査 (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、そのバッファー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ー制御 (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、各バッファーの kmem_bufctl_t 構造体を割り当てます。この構造体には、各バッファー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、このバッファー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ーは、106 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ーです。

```
> 0x70a9ae00,5/KKn
0x70a9ae00:     5               4ef83
                0               0
                1               bbddcafe
                feedface        139d
                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ーのアドレスです。これはオリジナルアドレス `0x70a9ae00` です。`0x70a9ae00`.「cache」フィールドは、このバッファー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、このバッファー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、117 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` が提供するもっとも有用な情報は、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。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、そのバッファー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ときは、ほかのどのカーネルエンティティー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。データ構造体が解放されたあと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ようにするために、これは重要なことです。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ー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 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。割り当てツリー全体を見るには、`::whatis -a` を代わりに使用します。

```
> 0x705d8640::whatis -a
705d8640 is 705d8640+0, allocated from streams_mblk
705d8640 is 705d8000+640, allocated from kmem_va_8192
705d8640 is 705d8000+640 from kmem_default vmem arena
705d8640 is 705d2000+2640 from kmem_va vmem arena
705d8640 is 705d2000+2640 from heap vmem arena
```

この割り当ては、kmem_va vmem 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。また、フルスタックの vmem 割り当ても表示できます。

kmem キャッシュと vmem 領域の完全なリストを表示するには、::kmastat dcmd を使用します。::kgrep を使用すると、この mblk へのポインタを含むほか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 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 ポインタです。

::kmem_verify を使用したバッファーの障害の発見

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 キャッシュを走査する、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、次に示します。

```
> ::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ーを調べます。

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

::kmem_verify がこのバッファーにフラグを立てた理由が明らかになります。バッファーの最初のワード (0x702babc0 で始まる) には、0xdeadbeef のパターンが使用されるはずでしたが、0 が使用されています。この時点で、このバッファーの bufctl_audit を調べることによって、このバッファー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、アクティビティー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ーと未使用バッファー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。もっとも簡単な方法は、::walk kmem_log で、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
```

::kmem_log コマンドを使用すると、トランザクションログ全体をもっと簡潔に表示できます。

```
> ::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、実際のバッファーを指しています。

これらの数字は、バッファー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。特定のバッファーが壊れた場合、トランザクションログの中でそのバッファーを突き止め、そのトランザクションを行なったスレッドがほかのどのトランザクションにかかわっていたかを判断します。このことは、バッファーの割り当て (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ーをバッファーアドレスによってフィルタリングします。次の例は、バッファー 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、特定のバッファー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 dcmd と ::freedby dcmd 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を示しています。

```
> 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ーを知ることができます。

◆ ◆ ◆ 第 10 章

10 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、SUNWmdbdm 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 API バージョン番号 */
        const mdb_dcmd_t *mi_dcmds;     /* NULL で終了する dcmd リスト */
        const mdb_walker_t *mi_walkers; /* NULL で終了する walk 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に使用され、常に MDB_API_VERSION に設定されます。したがっ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および登録されます。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、あるいはユーザーが `::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がパイプラインに出力するよう設定されて呼び出された

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、次の整数値のいずれかを返します。

DCMD_OK	dcmdは正常に完了しました。
DCMD_ERR	何らかの理由によりdcmdは失敗しました。
DCMD_USAGE	無効な引数が指定されたため、dcmdは失敗しました。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ます。
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ます。
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ます。この戻り値はDCMD_ERRに似ていますが、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しています。

各dcmdは<sys/mdb_modapi.h>に定義されているように、サンプルのdcmd()プロトタイプに従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「::」が付きません。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ません。
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ます。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]」と設定されます。dcmdが引数を受け

	付けない場合、dc_usage は NULL に設定されます。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、つまり flags パラメータに DCMD_ADDRSPEC を設定する必要があることを示します。用法文字列が「?」で始まっている場合、dcmd がオプションアドレスを受け付けることを示します。これらのヒントに従って、用法メッセージも変更されます。
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ます。
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ます。この関数では mdb_printf() を使用して詳細情報や例を表示できます。

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、あるいはユーザーが ::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/mdb_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ー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 のことを大域 walk といいます。NULL 以外の明示的な開始アドレスを持つ walk のことを局所 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です。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* が配下のオブジェクトの walker の局所コピーをポイントするように設定します。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` 次のステップへ進みます。walk init 関数が `WALK_NEXT` を返すと、MDB は walk step 関数を呼び出します。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します。

`WALK_DONE` walk が正常に完了しました。`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ります。

`WALK_ERR` walk がエラーのため終了しました。`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します。`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ます。

*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局所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 反復子 */
        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。図 10–1に、典型的なwalkerのアルゴリズムのフローチャートを示します。

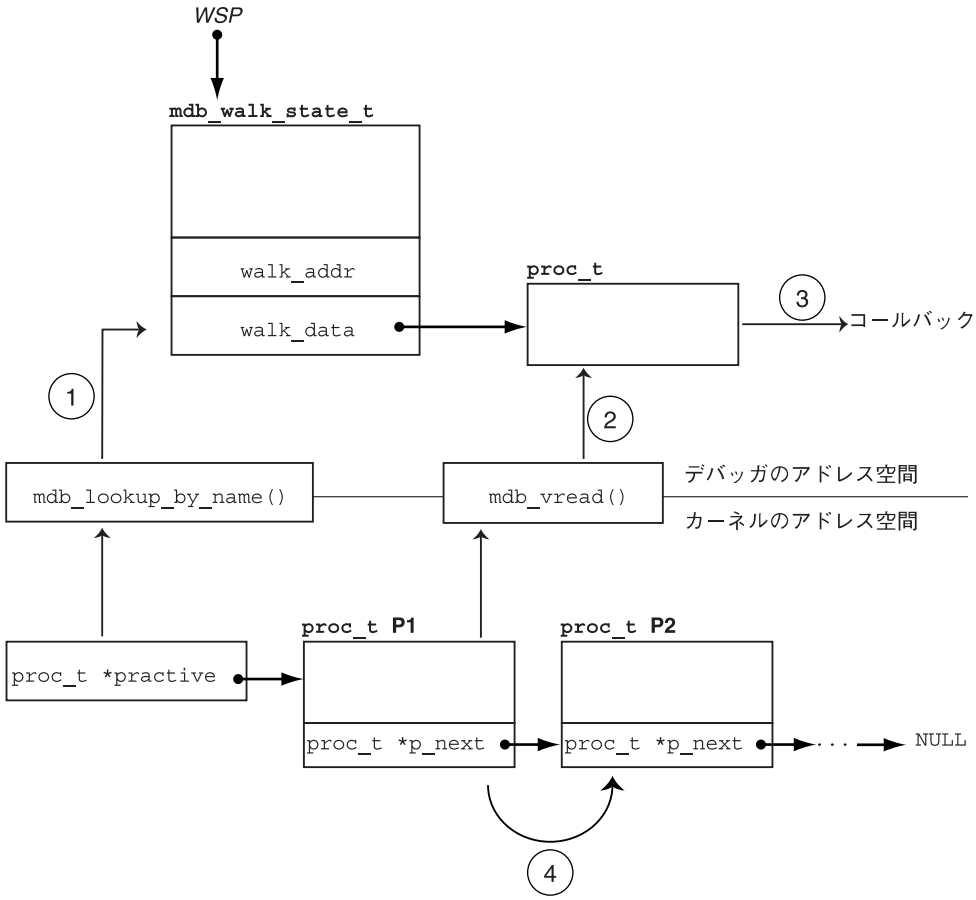

図 10–1　walker の例

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。walker の init ルーチンで、ステップ (1) として practive のシンボルが 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) でこの局所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大域的な情報を検出し、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* から始まる局所 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、大域 walk が実行されます。`mdb_pwalk()` を呼び出すことは *addr* パラメータを追跡せずに `mdb_walk()` を呼び出すことと同じです。この関数は成功すると 0 を返し、エラーが発生すると -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* パラメータは、呼び出し元にだけ意味を持つ隠された引数です。このパラメータは 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* から始まる大域 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。この関数は成功すると 0 を返し、エラーが発生すると -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* パラメータは、呼び出し元にだけ意味を持つ隠された引数です。このパラメータは 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* から始まる局所 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。この関数は成功すると 0 を返し、エラーが発生すると -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 の flags パラメータには `DCMD_LOOP` および `DCMD_ADDRSPEC` ビットが設定され、最初の呼び出しでは `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 を使用して大域 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。この関数は成功すると 0 を返し、エラーが発生すると -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 の flags パラメータには `DCMD_LOOP` および `DCMD_ADDRSPEC` ビットが設定され、最初の呼び出しでは `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);
```

与えられたパラメータで指定された 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。この関数は成功すると 0 を返し、エラーが発生すると -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、ほか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、あるいは組み込み構造体の walker を既存の CPU walker の上に重ねることができます。

`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あと、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あと、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、31 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、およびデバッガの大域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 名がすで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()` を呼び出したあと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。この関数は成功すると 0 を返し、エラーが発生すると -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* パラメータで指定された、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、成功した場合は実際に書き込まれたバイト数を返します。エラーが発生した場合は -1 を返します。

`mdb_fread()` および `mdb_fwrite()`

```
ssize_t mdb_fread(void *buf, size_t nbytes, uintptr_t addr);
ssize_t mdb_fwrite(const void *buf, size_t nbytes, uintptr_t addr);
```

これらの関数は、`addr` パラメータで指定されたターゲット仮想アドレスに対応するオブジェクトファイルの位置からデータを読み書きする機能を提供します。`mdb_fread()` 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_fwrite()` 関数は、成功した場合は実際に書き込まれたバイト数を返します。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* パラメータで指定された、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、成功した場合は実際に書き込まれたバイト数を返します。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ーに読み込みます。バッファーのサイズは *nbytes* で指定されます。この文字列が長すぎてバッファーに収まらない場合、文字列はバッファー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。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。次にその例を示します。

```
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、局所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ーに戻すため、int に格納される分のあとに残る 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。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。

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* パラメータはシンボル名を指定します。*object* パラメータは、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、29 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 パラメー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* パラメータはこのバッファー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ーにコピーし、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ーの長さを超えると、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* パラメー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* パラメータの後ろに、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ータで記述されます。 `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ータで記述されます。 `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ータで記述されます。 `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* の後ろ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ータで記述されます。 `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* の後ろに続く整数引数を *p* の参照している uintptr_t に格納します。
MDB_OPT_UINTPTR_SET	uintptr_t 引数をとります。このオプションは次のパラメータで記述されます。 `char c, uint_t type, boolean_t *flag, uintptr_t *p` タイプが MDB_OPT_UINTPTR_SET 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 *flag* の参照している boolean_t に値 1 (TRUE) を格納し、*c* の後ろに続く整数引数を *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ータで記述されます。

```
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* の後ろに続く整数引数を *p* の参照している uint64_t に格納します。

たとえば、次のソースコードの場合を考えます。

```
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);

        /* ... */
}
```

このコードは、mdb_getopts() を dcmd で使用して、ブール型のオプション「-v」(*opt_v* 変数を TRUE に設定する) と「-s」(*opt_s* 変数に格納された文字列引数を受け入れる) を受け入れる方法を示しています。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、先頭に有効な基底指示子 (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の基底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を指す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* パラメータは、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。バッファー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ーをトリガーします。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、という点です。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。`%la` を指定する必要はありません。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 */
        u_longlong_t bm_mask;       /* ビット用マスク */
        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>
```

%#b を指定した場合、復号化された名前の後ろに、ビットマスク配列内の要素と一致しなかったすべてのビットが 16 進数値として出力されます。

%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。%#d を指定すると、代替書式によって値の先頭に「0t」が付けられます。

`%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` を指定すると、代替書式によって値の先頭に「`0t`」が付けられます。

`%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`」です。

`%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まで進みます。幅を指定しないと、次のタブストップまで出力します。幅を指定すると、フィールド幅によって進む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。`%#u` を指定すると、代替書式によって値の先頭に「`0t`」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。

`%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* パラメータは *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 の 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* パラメータが 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ー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、バッファーには格納されず、出力ページャー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の先頭には自動的に、「`mdb:`」という接頭辞が付けられます。

さらに、*format* パラメータに復帰改行 (\n) 文字が含まれていない場合は、書式文字列の末尾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、次のような出力が得られます。

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

mdb_flush()

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

現在バッファー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ーに格納されます。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、端末またはほか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、復帰改行文字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_nhconvert()

```
void mdb_nhconvert(void *dst, const void *src, size_t nbytes);
```

src で指定されたアドレスに格納されている nbytes バイトのシーケンスをネットワークバイト順からホストバイト順に変換して、その結果を dst で指定されたアドレスに格納します。src パラメータと dst パラメータが同じ場合、オブジェクトはそのアドレスのままで変換されます。変換は同じように行われるので、この関数はホスト順からネットワーク順に変換する場合にも、ネットワーク順からホスト順に変換する場合にも使用できます。

mdb_dumpptr() および mdb_dump64()

```
int mdb_dumpptr(uintptr_t addr, size_t nbytes, uint_t flags,
                    mdb_dumpptr_cb_t func, void *data);
int mdb_dump64(uint64_t addr, uint64_t nbytes, uint_t flags,
                    mdb_dump64_cb_t func, void *data);
```

これらの関数を使用すると、書式化された 16 進数および ASCII のデータダンプを生成して標準出力に出力できます。どちらの関数も、開始場所を指定する *addr* パラメータ、表示するバイト数を指定する *nbytes* パラメータ、フラグセット (次を参照)、表示するデータを読み取るために使用する `func` コールバック関数、そして、`func` コールバック関数の呼び出しごとに最後の引数として渡されるデータパラメータを受け入れます。これらの関数はほとんど同じですが、`mdb_dumpptr` は `uintptr_t` をアドレスパラメータとして使用し、`mdb_dump64` は `uint64_t` を使用します。この区別は、たとえば、`mdb_dump64` を `mdb_pread` に結びつけるときに便利です。組み込み `::dump` dcmd はこれらの関数を使用してデータを表示します。

flags パラメータは、次の 1 つ以上の値のビット単位の論理和となります。

`MDB_DUMP_RELATIVE`	各行に、絶対的なアドレスではなく、開始アドレスからの相対的な行数を与えます。
`MDB_DUMP_ALIGN`	出力を段落境界に揃えます。
`MDB_DUMP_PEDANT`	アドレスを 80 カラムに揃えて切り捨てるのではなく、完全な幅で表示します。
`MDB_DUMP_ASCII`	16 進数データの次に ASCII 値を表示します。
`MDB_DUMP_HEADER`	データについてのヘッダー行を表示します。
`MDB_DUMP_TRIM`	行全体を読み取って出力するのではなく、指定されたアドレスの内容だけを読み取って表示します。
`MDB_DUMP_SQUISH`	前の行を繰り返している行に「*」を表示することによって、繰り返し行を省略します。
`MDB_DUMP_NEWDOT`	ドット値を、関数が読み取った最後のアドレスを超えるアドレスに更新します。
`MDB_DUMP_ENDIAN`	エンディアン性を調整します。このオプションは、ワードサイズが (`MDB_DUMP_GROUP()` で指定される) 現在のグループサイズと同じであることを想定します。このオプションは常に、整列、ヘッダー、および ASCII 表示をオフにして、出力が乱れることを防ぎます。`MDB_DUMP_TRIM` と `MDB_DUMP_ENDIAN` が一緒に設定されている場合、ダンプされるバイト数はもっとも近いワードサイズのバイトに切り捨てられます。
`MDB_DUMP_WIDTH(`*width* `)`	表示される行ごとに 16 バイト段落の数をインクリメントします。*width* のデフォルト値は 1 で、最大値は 16 です。
`MDB_DUMP_GROUP(`*group* `)`	バイトグループのサイズを *group* に設定します。デフォルトの *group* サイズは 4 バイトです。*group* サイズは行幅を分割する 2 のべき乗にする必要があります。

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));
```

この場合、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* パラメータの設定に従って、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 パラメー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* パラメータで指定します。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() とともに使用して、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 (.) で表現されますが、start および stop パラメー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あとに、`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ー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する場合などです。

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ー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、`flags` パラメータに `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ーをパイプ構造体 *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ー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ータで `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ーの内容を、*buf* で指定されたバッファーの中に読み取ります。*buf* のサイズは *nbytes* パラメータで指定します。*nbytes* を超えるバイト数は呼び出し元のバッファー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。呼び出し元は、指定した特定のバッファーのサイズを調べる場合には、*buf* を NULL に、*nbytes* をゼロに指定します。この場合、`mdb_get_xdata()` はバッファー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ーを使用することで、モジュールの作成者は、モジュール API を通してほか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ーのセットは、`::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

```
mdb [ -fkmuwyAFMS ]  [ +o option ]  [ -p pid ]  [ -s distance]
     [ -I path ]  [ -L path ]  [ -P prompt ]  [ -R root ]
     [ -V dis-version ]  [ object  [ core ]  | core  | suffix ]
```

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、または稼働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。これによって、mdb とプロセスを制御しようとしているほかのツールとの間で予期しない相互作用が発生する可能性があります。

`-f` 強制的に raw ファイルデバッギングモードに入ります。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。ファイルのタイプを推測できない場合、デバッガはデフォルトでそのファイ

ルをプレーンなバイナリデータとして調査します。-f オプションを使用すると、mdb は引数を調査すべき raw ファイルセットとして解釈します。

-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ー (ISA) の名前に展開されます。sparc、sparcv9、i386、または amd64 です。
- %o　変更対象のパスの古い値まで展開されます。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) に展開されます。
- %r　ルートディレクトリのパス名まで展開されます。-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 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展開されます。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働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。

-K
kmdb を読み込み、実行中のオペレーティングシステムカーネルを停止して、kmdb デバッガプロンプトを表示します。このオプションは、システムコンソールでしか使用してはなりません。それに続く kmdb プロンプトがシステムコンソールに表示されます。

-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。次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
より厳密な adb(1) との互換性を有効にする。プロンプトは空の文字列に設定され、出力ページャーなどの多数の mdb 機能が無効になる

array_mem_limit=*limit*
::print が表示する配列のメンバー数に対してデフォルトの制限を設定する。*limit* が特別なトークン none である場合、デフォルトで配列のすべてのメンバーが表示される

array_str_limit=*limit*
::print が char 型の配列を出力するときに ASCII 文字列として表示する文字数に対してデフォルトの制限を設定する。*limit* が特別なトークン none である場合、デフォルトで char 配列全体が string として表示される

follow_exec_mode=mode
exec(2) システムコールに従って動作するようにデバッガを設定する。mode は次の名前付き定数の 1 つである必要がある

ask	stdout が端末デバイスである場合、デバッガは exec(2) システムコールが返ったあとに停止して、exec または stop のどちらに従うかをユーザーに尋ねる。stdout が端末デバイスではない場合、ask モードはデフォルトで stop になる
follow	デバッガは自動的にターゲットプロセスを継続し、新しい実行可能ファイルに基づいて、ターゲットプロセスのマッピングとシンボルテーブルをすべてリセットし、exec に従うfollow の動作の詳細については、67 ページの「exec との対話」を参照
stop	exec システムコールから戻ったあと、デバッガは exec システムコールに従うことを停止する。stop の動作の詳細については、67 ページの「exec との対話」を参照

follow_fork_mode=mode
fork(2)、fork1(2)、または vfork(2) システムコールの動作に従って動作するようにデバッガを設定する。mode は次の名前付き定数の 1 つである必要がある

ask	stdout が端末デバイスである場合、デバッガは fork システムコールが返ったあとに停止して、親プロセスまたは子プロセスのどちらに従うかをユーザーに尋ねる。stdout が端末デバイスではない場合、ask モードはデフォルトで親プロセスになる
parent	デバッガは親プロセスに従って、子プロセスを切り離し、「動作中 (running)」に設定する
child	デバッガは子プロセスに従って、親プロセスを切り離し、「動作中 (running)」に設定する

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

nostop
-p オプションを指定したか、::attach または :A dcmd を適用した場合、ユーザープロセスと接続しているときには、そのユーザープロセスを停止しない。nostop の動作の詳細については、68 ページの「プロセスの接続と解放」を参照

pager
出力ページャーが有効になる (デフォルト設定)

repeatlast
復帰改行文字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る

showlmid
MDB は LM_ID_BASE と LM_ID_LDSO 以外のリンクマップを使用するユーザーアプリケーションでシンボルの命名と識別をサポートする (29 ページの「シンボルの名前解決」を参照)。LM_ID_BASE または LM_ID_LDSO 以外のリンクマップ上のシンボルは LMlmid‘library‘symbol のように表示される (このとき、lmid はデフォルトの出力基数 (16 進数) のリンクマップ ID)。showlmid オプションを有効にすると、すべてのシンボルとオブジェクト (LM_ID_BASE と LM_ID_LDSO に関連するものも含む) のリンクマップ ID 有効範囲を表示するように MDB を構成できる。オブジェクトファイル名を扱う組み込み dcmd (::nm、::mappings、$m、::objects など) は、上記 showlmid の値に従ってリンクマップ ID を表示する

-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、パス名の展開の際に %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。

`-U` kmdb が読み込まれた場合は、それを読み込み解除します。kmdb が使用中でない場合は、それを読み込み解除し、カーネルデバッガで使用されたメモリーを解放してオペレーティングシステムが利用できるようにします。

`-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。端末には、tty モードに切り換えるために明示的な初期化シーケンスを必要とするものがあります。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。

オペランド

次のオペランドがサポートされています。

object　調査する ELF 書式のオブジェクトファイルを指定します。mdb が調査および編集できるのは、ELF 書式の実行可能ファイル (ET_EXEC)、ELF 動的ライブラリファイル (ET_DYN)、ELF 再配置可能オブジェクトファイル (ET_REL)、およびオペレーティングシステムのシンボルテーブル (unix.X ファイル) です。

core　ELF プロセスコアファイル (ET_CORE) またはオペレーティングシステムのクラッシュダンプ (vmcore.X ファイル) を指定します。ELF コアファイルオペランド (core) を指定する場合、対応するオブジェクトファイルを指定しないと、mdb はいくつかの異なるアルゴリズムを使用して、コアファイルを生成した実行可能ファイルの名前を推測しようとします。このような実行可能ファイルが見つからなかった場合、mdb は動作し続けますが、いくつかのシンボル情報が利用できなくなっている可能性があります。

suffix　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のペアを表す数値の接尾辞を指定します。たとえば、接尾辞が「3」である場合、mdb は「unix.3」や「vmcore.3」などの名前のファイルを調査すると推測します。同じ名前の実際のファイルが現在のディレクトリに存在する場合、この数字文字列は接尾辞としては解釈されません。

終了ステータス

次の終了値が返されます。

0　デバッガは正常に実行を終了しました。

1　致命的なエラーが発生しました。

2　無効なコマンド行オプションが指定されました。

環境変数

次の環境変数がサポートされます。

HISTSIZE　この変数は、コマンド履歴リストの最大の長さを決定するために使用されます。この変数が存在しない場合、デフォルトの長さは 128 です。

HOME　この変数は、ユーザーのホームディレクトリ (.mdbrc ファイルが存在する場所) のパス名を決定するために使用されます。この変数が存在しない場合、.mdbrc は処理されません。

SHELL　この変数は、! メタキャラクタを使用して要求されたシェルエスケープを処理するシェルのパス名を決定するために使用されます。この変数が存在しない場合、/bin/sh が使用されます。

◆ ◆ ◆ 付 録 B

注意

警告

次の警告情報は MDB の使用に適用されます。

エラー回復メカニズムの使用

デバッガとその dmod は同じアドレス空間内で動作するので、dmod にバグがあると、MDB がコアをダンプしたり、誤動作したりする可能性があります。MDB の `resume` 機能 (40 ページの「シグナル処理」を参照) はこのような状況に対して、制限付きで回復メカニズムを提供します。しかし、dmod 自身の状態やデバッガの大域的な状態だけでは、MDB は 当該 dmod が壊れているかどうかを最終的には判断できません。したがって、`resume` 操作は安全であるとは保証されず、また、その後のデバッガのクラッシュを防ぐこともできません。`resume` によるもっとも安全な対処方法は、重要なデバッグ情報を保存し、デバッガを停止し再起動します。

動作中のオペレーティングシステムのデバッガによる修正

動作中のオペレーティングシステムのアドレス空間をデバッガを使用して修正する (書き込む) ことは非常に危険であり、ユーザーがカーネルデータの構造を破損するとシステムパニックが発生する可能性があります。

動作中のオペレーティングシステムの kmdb による停止

kmdb を使って動作中のオペレーティングシステムを停止するには、mdb -K を使用するか、動作中のオペレーティングシステムにブレークポイントを設定しますが、この操作は、開発者が本稼働システム以外のシステムで行うためのものです。kmdb によってオペレーティングシステムカーネルが停止すると、オペレーティングシステムサービスとネットワーク関連機能が実行されないため、ターゲットシステムに依存するネットワーク上のほかのシステムはターゲットシステムに接続できなくなります。

注意

プロセスコアファイルの調査に関する制限

MDB は、Solaris 2.6 より前の Solaris オペレーティングシステムのリリースで生成されたプロセスコアファイルの調査をサポートしません。あるオペレーティングシステムのリリースで生成されたコアファイルを別のオペレーティングシステムのリリースで調査する場合、実行時リンカーのデバッギングインタフェース (librtld_db) は初期化できない可能性があります。この場合、共用ライブラリのシンボル情報は利用できません。さらに、共用マッピングはユーザーのコアファイル内には存在しないので、共用ライブラリのテキストセクションと読み取り専用データは、コアがダンプされた時点にプロセス内に存在していたデータと一致しない可能性があります。x86 版の Solaris システムから生成されたコアファイルは SPARC 版の Solaris システムでは調査できず、その逆もできません。

クラッシュダンプファイルの調査に関する制限

Solaris 7 以前のリリースで生成されたクラッシュダンプを調査するには、対応するオペレーティングシステムのリリース用の libkvm が必要です。あるオペレーティングシステムのリリースで生成されたクラッシュダンプを別のオペレーティングシステムのリリースで dmod を使用して調査する場合、カーネルの実装によっては、いくつかの dcmd や walker が適切に動作しない可能性があります。この状況を検出すると、MDB は警告メッセージを発行します。x86 版の Solaris システムから生成されたクラッシュダンプは SPARC 版の Solaris システムでは調査できず、その逆もできません。

32 ビットと 64 ビットのデバッガ間の関係

MDB は 32 ビットと 64 ビットの両方のプログラムのデバッグをサポートします。ターゲットのプログラムを調査して、そのデータモデルを決定したあと、MDB は必要に応じて自動的に、ターゲットと同じデータモデルを持つ `mdb` バイナリを実行し直します。このアプローチによって、読み込まれたモジュールがプライマリターゲットと同じデータモデルを使用するので、デバッガモジュールを作成する作業が簡単になります。64 ビットのターゲットプログラムをデバッグできるのは 64 ビットのデバッガだけです。64 ビットのデバッガを使用できるのは 64 ビットのオペレーティング環境が動作しているシステム上だけです。

kmdb に使用できるメモリーに関する制限

`kmdb` に使用できるメモリーは、デバッガが読み込まれるときに割り当てられますが、それ以降は拡張することができません。デバッガのコマンドで、使用可能な容量よりも多くのメモリーを割り当てようとすると、コマンドが実行できなくなります。デバッガは、メモリーが不足している状況からの正常な回復を試みますが、極度にメモリーが不足した環境のもとではシステムを強制終了せざるを得ない場合もあります。システムメモリーの制約は、特に 32 ビットのオペレーティングシステムカーネルを使用する x86 プラットフォームで厳しくなっています。

開発者向けの情報

`mdb`(1) のマニュアルページには、組み込み `mdb` 機能についての開発者向けの詳細な情報が記載されています。ヘッダーファイル `<sys/mdb_modapi.h>` には MDB モジュール API にある関数用のプロトタイプが入っており、`SUNWmdbdm` 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用のソースコードが入っています。

◆ ◆ ◆ 付録 C

adb および kadb からの移行

従来の adb(1) ユーティリティーから mdb(1) への移行は比較的簡単です。MDB は adb の構文、組み込みコマンド、およびコマンド行オプションに進化的な互換性を提供します。MDB は adb(1) の既存の機能すべてに互換性を提供しようとしますが、バグレベルで adb(1) との互換性はありません。この付録では、mdb(1) で正確にはエミュレートされない adb(1) の機能について簡単に説明し、ユーザーに新しい機能を紹介します。

コマンド行オプション

MDB は adb(1) が認識するコマンド行オプションのスーパーセットを提供します。adb(1) のオプションはすべてサポートされ、以前と同じ意味を持ちます。/usr/bin/adb パス名は mdb(1) を呼び出すリンクとして提供されるので、自動的に拡張 adb(1) 互換モードが有効になります。/usr/bin/adb リンクを実行することは、-o adb オプション付きで mdb を実行すること、あるいは、デバッガを起動したあとに ::set -o adb を実行することと同じです。

構文

MDB 言語は adb(1) 言語と同じ構文を厳守することによって、従来のマクロやスクリプトファイルにも互換性を提供します。新しい MDB の dcmd は、拡張形式の ::name を使用することによって、: または $ の接頭辞付きの従来のコマンドと区別されます。dcmd 名の右側にある式も、ドル記号の接頭辞付きの各括弧 ($[ ]) で式を囲めば評価できます。adb(1) と同様に、感嘆符 (!) で始まる入力行は、コマンド行がユーザーのシェルで実行されることを示します。MDB では、デバッガコマンドの接尾辞として感嘆符を付けると、その出力が感嘆符の後ろにあるシェルコマンドにパイプで渡されることを示します。

adbでは、2 項演算子は左結合で、単項演算子よりも優先順位は低くなります。入力行の 2 項演算子は厳密な左から右の順番で評価されます。MDB では、2 項演算子は左結合で、単項演算子よりも優先順位は低くなりますが、2 項演算子は 27 ページ

の「2 項演算子」の表の優先順位に従って動作します。演算子は ANSI C の優先順位に準拠します。あいまいな式を明示的に括弧で囲んでいない従来の adb(1) マクロファイルを MDB で使用するには、そのファイルを更新する必要がある可能性があります。たとえば、adb では、次のコマンドは整数値 9 に評価されます。

```
$ echo "4-1*3=X" | adb
        9
```

MDB では、ANSI C と同様に、演算子「*」は「-」よりも優先順位が高いので、結果は整数値 1 になります。

```
$ echo "4-1*3=X" | mdb
        1
```

ウォッチポイント長さ指示子

MDB が認識するウォッチポイント長さ指示子の構文は、adb(1) で記述されている構文とは異なります。特に、adb のウォッチポイント用コマンド :w、:a、および :p では、整数の長さ (バイト数) をコロンとコマンド文字の間に挿入していました。MDB では、このカウントは初期アドレスの後ろに繰り返しカウントとして指定する必要があります。簡単に言うと、adb(1) コマンドでは次のようになります。

```
123:456w
123:456a
123:456p
```

MDB では次のようになります。

```
123,456:w
123,456:a
123,456:p
```

ユーザープロセスのウォッチポイントを作成する場合、MDB の ::wp dcmd の方がより完全な機能を提供します。同様に、従来の kadb 長さ修飾子コマンド $l はサポートされません。このため、kmdb で使用される各 ::wp コマンドにはウォッチポイントサイズを指定する必要があります。

アドレスマップ修飾子

仮想アドレスマップとオブジェクトファイルマップのセグメントを変更するための adb(1) コマンド (に相当するコマンド) は MDB にはありません。特に、/m、/*m、?m、および ?*m 書式指定子は、MDB では認識またはサポートされません。このような指定子は、現在のオブジェクトやコアファイルの有効なアドレス可能な範囲を手動で変更するときに使用されていました。MDB はこのようなファイルのアドレス可能な範囲を適切かつ自動的に認識し、その範囲をライブプロセスをデバッグしているときに更新するので、上記コマンドは必要ありません。

出力

MDB では、いくつかのコマンドからのテキスト出力形式が厳密には異なります。マクロファイルは同じ基本規則を使用して書式化されますが、あるコマンドの文字単位の出力に厳密に依存するシェルスクリプトは変更する必要がある可能性があります。adb コマンドの出力を構文解析するシェルスクリプトは、その妥当性を検査し直して、MDB の移行の一部として更新する必要があります。

遅延ブレークポイント

従来の kadb ユーティリティーでは、既存の adb 構文と互換性のない遅延ブレークポイントの構文がサポートされていました。kadb では、これらの遅延ブレークポイントは構文 *module#symbol* :b を使って指定しました。kmdb で遅延ブレークポイントを設定するには、第 6 章で説明しているように、MDB の ::bp dcmd を使用します。

x86: 入出力ポートアクセス

従来の kadb ユーティリティーでは、:i および :o コマンドを使って x86 システム上の入出力ポートにアクセスできました。mdb または kmdb では、これらのコマンドはサポートされていません。x86 システム上の入出力ポートにアクセスするには、::in および ::out コマンドを使用します。

◆ ◆ ◆ 付録 D

crash からの移行

従来の crash ユーティリティーから 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 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。稼働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 を起動したあと、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、関数名 (MDB では dcmd 名) の前に「::」を付けること以外、MDB における入力は 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 名の後ろに続けて文字列オプションを指定できます。関数呼び出しのあとに ! 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 D–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 D–1 基数指示子

crash	mdb	基数
0x	0x	16 進数 (ベース 16)
0d	0t	10 進数 (ベース 10)
0b	0i	2 進数 (ベース 2)

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 オペレーティングシステム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
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

crash 関数	mdb dcmd	コメント
kmausers	::kmausers	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	デバッガを終了します。
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、::log を使用して、大域的にコマンドの入出力内容の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	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
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 構造体のリストが必要な場合、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

crash 関数	mdb dcmd	コメント
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	大域的な var 構造体の調整可能なシステムパラメータを出力する
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

K

M

R

S

U

W