

Tru64 UNIX

プログラミング・ガイド

Part Number: AA-RK3MD-TE

2002 年 11 月

ソフトウェア・バージョン: Tru64 UNIX バージョン 5.1B 以降

本書は、Tru64 UNIX オペレーティング・システムのプログラム開発環境について、C 言語を中心に説明しています。

日本ヒューレット・パッカード株式会社

© 2002 日本ヒューレット・パッカート株式会社

本書の著作権は日本ヒューレット・パッカート株式会社が保有しており、本書中の解説および図、表は日本ヒューレット・パッカートの文書による許可なしに、その全体または一部を、いかなる場合にも再版あるいは複製することを禁じます。

また、本書に記載されている事項は、予告なく変更されることがありますので、あらかじめご承知おきください。万一、本書の記述に誤りがあった場合でも、弊社は一切その責任を負いかねます。

日本ヒューレット・パッカートは、弊社または弊社の指定する会社から納入された機器以外の機器で対象ソフトウェアを使用した場合、その性能あるいは信頼性について一切責任を負いかねます。

本書で解説するソフトウェア(対象ソフトウェア)は、所定のライセンス契約が締結された場合に限り、その使用あるいは複製が許可されます。

Open Software Foundation, OSF, OSF/1, OSF/Motif, および Motif は Open Software Foundation 社の商標です。UNIX は The Open Group の米国ならびに他の国における登録商標です。The Open Group は The Open Group の米国ならびに他の国における商標です。

COMPAQ, Compaq ロゴ, Digital ロゴは U.S. Patent and Trademark Office に登録されています。以下は、Digital Equipment Corporation の商標です: ALL-IN-1, Alpha AXP, AlphaGeneration, AlphaServer, AltaVista, ATMworks, AXP, Bookreader, CDA, DDIS, DEC, DEC Ada, DECevent, DEC Fortran, DEC FUSE, DECnet, DECstation, DECsystem, DECterm, DECUS, DECwindows, DTIF, Massbus, MicroVAX, OpenVMS, POLYCENTER, PrintServer, Q-bus, StorageWorks, Tru64, TruCluster, TURBOchannel, ULTRIX, ULTRIX Mail Connection, ULTRIX Worksystem Software, UNIBUS, VAX, VAXstation, VMS, XUI。このドキュメントに記載されているその他の会社名および製品名は、各社の商標または登録商標です。

原典: Tru64 UNIX Programmer's Guide (AA-RH9VD-TE)
Copyright ©2002 Hewlett-Packard Company

目次

まえがき

1 概要

1.1	アプリケーション開発のフェーズ	1-1
1.2	仕様および設計上の留意事項	1-2
1.2.1	規格	1-2
1.2.2	国際化	1-3
1.2.3	ウィンドウ・アプリケーション	1-4
1.2.4	アプリケーションの保護	1-4
1.3	ソフトウェア開発の主なツール	1-5
1.3.1	Tru64 UNIX 環境でサポートされる言語	1-5
1.3.2	オブジェクト・ファイルのリンク	1-5
1.3.3	デバッグおよびプログラム分析ツール	1-6
1.4	ソース・ファイル制御	1-6
1.5	プログラムのインストール・ツール	1-7
1.6	プロセス間通信機能の概要	1-8

2 コンパイラ・システム

2.1	コンパイラ・システムの構成要素	2-2
2.2	Tru64 UNIX 環境におけるデータ型	2-4
2.2.1	データ型のサイズ	2-5
2.2.2	浮動小数点の範囲と処理	2-5
2.2.3	構造体の位置合わせ	2-6
2.2.4	ビット・フィールドの位置合わせ	2-7
2.2.5	__align 記憶クラス修飾子	2-9

2.3	C プリプロセッサ	2-10
2.3.1	定義済みマクロ	2-10
2.3.2	ヘッダ・ファイル	2-11
2.3.3	各国語対応インクルード・ファイルの設定	2-12
2.3.4	処理系固有のプリプロセッサ指示文 (#pragma)	2-13
2.4	ソース・プログラムのコンパイル	2-14
2.4.1	省略時のコンパイル動作	2-14
2.4.2	多言語プログラムのコンパイル	2-19
2.4.3	配列境界の実行時検査の有効化	2-19
2.5	オブジェクト・ファイルのリンク	2-22
2.5.1	コンパイラ・コマンドによるリンク	2-23
2.5.2	ld コマンドによるリンク	2-24
2.5.3	ライブラリの指定	2-24
2.5.4	リンク出力ファイルでのリンク・エラー問題の回避	2-26
2.6	プログラムの実行	2-26
2.7	オブジェクト・ファイルのツール	2-27
2.7.1	ファイル内の選択した部分のダンプ (odump)	2-28
2.7.2	シンボル・テーブル情報の表示 (nm)	2-29
2.7.3	ファイル・タイプの決定 (file)	2-29
2.7.4	ファイルのセグメント・サイズの決定 (size)	2-30
2.7.5	オブジェクト・ファイルの逆アセンブル (dis)	2-30
2.8	標準 C ライブラリにおける ANSI 名前空間汚染のクリーン アップ	2-31
2.9	インライン・アセンブリ・コード — ASM	2-33

3 プラグマ・プリプロセッサ指示文

3.1	#pragma assert 指示文	3-2
3.1.1	#pragma assert func_attrs	3-2
3.1.2	#pragma assert global_status_variable	3-5
3.1.3	#pragma assert non_zero	3-6

3.2	#pragma environment 指示文	3-7
3.3	#pragma extern_model 指示文	3-9
3.3.1	構文	3-10
3.3.2	#pragma extern_model relaxed_refdef	3-12
3.3.3	#pragma extern_model strict_refdef	3-13
3.3.4	#pragma extern_model save	3-13
3.3.5	#pragma extern_model restore	3-13
3.4	#pragma extern_prefix 指示文	3-14
3.5	#pragma inline 指示文	3-15
3.6	#pragma intrinsic および #pragma function 指示文	3-17
3.7	#pragma linkage 指示文	3-19
3.8	#pragma member_alignment 指示文	3-22
3.9	#pragma message 指示文	3-24
3.9.1	#pragma message option1	3-24
3.9.2	#pragma message option2	3-28
3.9.3	#pragma message (" string ")	3-29
3.10	#pragma optimize 指示文	3-29
3.11	#pragma pack 指示文	3-32
3.12	#pragma pointer_size 指示文	3-34
3.13	#pragma unroll 指示文	3-36
3.14	#pragma use_linkage 指示文	3-36
3.15	#pragma weak 指示文	3-37

4 シェアード・ライブラリ

4.1	シェアード・ライブラリの概要	4-1
4.2	シンボルの解決	4-4
4.2.1	リンクの探索パス	4-4
4.2.2	実行時ローダの探索パス	4-5
4.2.3	名前の解決	4-6
4.2.4	未解決の外部シンボルの処理のオプション	4-7

4.3	シェアード・ライブラリとのリンク	4-8
4.4	シェアード・ライブラリの指定解除	4-8
4.5	シェアード・ライブラリの作成	4-9
4.5.1	オブジェクト・ファイルからのシェアード・ライブラリの作成	4-9
4.5.2	アーカイブ・ライブラリからのシェアード・ライブラリの作成	4-10
4.6	プライベートなシェアード・ライブラリの使用	4-10
4.7	クイックスタートの使用	4-11
4.7.1	オブジェクトのクイックスタートの確認	4-14
4.7.2	手動によるクイックスタート問題の解決	4-14
4.7.3	fixso ユーティリティによるクイックスタート問題の解決	4-17
4.8	シェアード・ライブラリとリンクしているプログラムのデバッグ	4-19
4.9	シェアード・ライブラリの実行時のロード	4-19
4.10	シェアード・ライブラリ・ファイルの保護	4-20
4.11	シェアード・ライブラリのバージョン管理	4-21
4.11.1	バイナリ非互換修正	4-22
4.11.2	シェアード・ライブラリのバージョン管理	4-23
4.11.3	メジャーおよびマイナー・バージョン識別子	4-25
4.11.4	シェアード・ライブラリの完全バージョンと部分バージョン	4-26
4.11.5	シェアード・ライブラリの複数バージョンとのリンク	4-27
4.11.6	ロード時におけるバージョン・チェック	4-29
4.11.7	ロード時における複数バージョンのチェック	4-30
4.12	シンボル割り当て	4-35
4.13	シェアード・ライブラリの制約事項	4-35

5 dbx によるプログラムのデバッグ

5.1	デバッグの一般的な留意事項	5-3
-----	---------------------	-----

5.1.1	ソース・レベルのデバッガを使用する理由	5-3
5.1.2	アクティブ化レベル	5-4
5.1.3	プログラム実行障害箇所の特定	5-5
5.1.4	不正の出力結果の原因分析	5-6
5.1.5	実行プロセスのコア・スナップショットの作成	5-6
5.1.6	障害の回避	5-6
5.2	dbx の実行	5-8
5.2.1	デバッグ用プログラムのコンパイル	5-8
5.2.2	dbx 初期化ファイルの作成	5-9
5.2.3	dbx の起動と終了	5-9
5.3	dbx コマンドの使用方法	5-12
5.3.1	変数名の修飾	5-12
5.3.2	dbx 式と式の優先順位	5-12
5.3.3	dbx のデータ型および定数	5-13
5.4	dbx モニタによる作業	5-14
5.4.1	dbx コマンドの繰り返し	5-15
5.4.2	dbx コマンド行の編集	5-16
5.4.3	複数のコマンドの入力	5-18
5.4.4	シンボル名の補完	5-18
5.5	dbx の制御	5-19
5.5.1	変数の設定および削除	5-19
5.5.2	定義済みの dbx 変数	5-20
5.5.3	別名の定義および削除	5-27
5.5.4	デバッグ・セッション状態の監視	5-29
5.5.5	ブレークポイントの削除あるいは無効化	5-30
5.5.6	ロードされたオブジェクト・ファイル名の表示	5-31
5.5.7	コア・ダンプ用のシェアード・ライブラリの指定	5-31
5.5.8	dbx からのサブシェルの起動	5-32
5.6	ソース・プログラムの検査	5-32

5.6.1	ソース・ファイルのディレクトリ位置の指定	5-32
5.6.2	アクティブ化スタックでの移動	5-33
5.6.2.1	where コマンドおよび tstack コマンド	5-33
5.6.2.2	up コマンド, down コマンド, func コマンド	5-34
5.6.3	現在のソース・ファイルの変更	5-35
5.6.4	ソース・コードのリスト	5-36
5.6.5	ソース・ファイル・テキストの探索	5-37
5.6.6	dbx 内からのソース・ファイルの編集	5-38
5.6.7	同じ名前の変数の識別	5-38
5.6.8	変数およびプロシージャのタイプの確認	5-39
5.7	プログラムの制御	5-39
5.7.1	プログラムの実行および再実行	5-40
5.7.2	step コマンドによるプログラムの実行	5-41
5.7.3	return コマンド	5-43
5.7.4	コード内の特定の場所への移動	5-43
5.7.5	ブレークポイント後のプログラム実行の再開	5-44
5.7.6	プログラム変数値の変更	5-45
5.7.7	実行可能なディスク・ファイルのパッチ	5-45
5.7.8	特定のプロシージャの実行	5-46
5.7.9	環境変数の設定	5-47
5.8	ブレークポイントの設定	5-48
5.8.1	概要	5-48
5.8.2	stop および stopi によるブレークポイントの設定	5-49
5.8.3	実行中の変数のトレース	5-52
5.8.4	dbx での条件コードの記述	5-53
5.8.5	シグナルの受信および無視	5-55
5.9	プログラム状態の検査	5-56
5.9.1	変数および式の値の出力	5-56
5.9.2	dump コマンドによるアクティブ化レベル情報の表示	5-58

5.9.3	メモリの内容の表示	5-59
5.9.4	dbx セッションの入出力の記録および再生	5-61
5.9.4.1	デバッガ入力 of 記録および再実行	5-61
5.9.4.2	デバッガ出力 of 記録および再実行	5-63
5.10	コア・ダンプ・ファイルのネーミング	5-64
5.10.1	システム・レベルでのコア・ファイルのネーミング機能の有効化	5-65
5.10.2	アプリケーション・レベルでのコア・ファイルのネーミング機能の有効化	5-65
5.11	実行中のプロセスのデバッグ	5-66
5.12	マルチスレッド・アプリケーションのデバッグ	5-68
5.13	複数の非同期プロセスのデバッグ	5-71
5.14	サンプル・プログラム	5-72

6 lint による C プログラムの検査

6.1	lint コマンドの構文	6-2
6.2	プログラム・フロー検査	6-4
6.3	データ型検査	6-5
6.3.1	二項演算子および暗黙の代入	6-5
6.3.2	構造体と共用体	6-6
6.3.3	関数定義と使用方法	6-6
6.3.4	列挙値	6-7
6.3.5	型キャスト	6-7
6.4	変数および関数の検査	6-7
6.4.1	矛盾する値を返す関数	6-8
6.4.2	使用されていない関数値	6-9
6.4.3	関数についての検査の禁止	6-9
6.5	初期化前の変数使用のチェック	6-11
6.6	移行検査	6-11

6.7	移植性検査	6-12
6.7.1	文字	6-12
6.7.2	ビット・フィールド	6-13
6.7.3	外部名サイズ	6-13
6.7.4	複雑な式の使用と副作用	6-14
6.8	コーディング・エラーおよびコーディングのスタイルの相違の チェック	6-14
6.8.1	long 型変数の int 型変数への代入	6-14
6.8.2	演算子の優先度	6-15
6.8.3	宣言の矛盾	6-15
6.9	テーブル・サイズの増加	6-15
6.10	lint ライブラリの作成	6-16
6.10.1	入力ファイルの作成	6-16
6.10.2	lint ライブラリ・ファイルの作成	6-18
6.10.3	新しいライブラリによるプログラムの検査	6-18
6.11	lint エラー・メッセージ	6-18
6.12	警告クラス・オプションを使用した lint メッセージの抑制 ...	6-24
6.13	コンパイル時に検出される構文エラーのための関数プロトタイ プの生成	6-30

7 Third Degree によるプログラムのデバッグ

7.1	アプリケーションにおける Third Degree の実行	7-2
7.1.1	シェアード・ライブラリでの Third Degree の使用	7-4
7.2	デバッグ例	7-6
7.2.1	Third Degree のカスタマイズ	7-6
7.2.2	Makefile の変更	7-8
7.2.3	Third Degree のログ・ファイルの検査	7-8
7.2.3.1	実行時メモリ・アクセス・エラーのリスト	7-8
7.2.3.2	メモリ・リーク	7-11

7.2.3.3	ヒープ・ヒストリ	7-12
7.2.3.4	メモリ・レイアウト	7-13
7.3	Third Degree エラー・メッセージの解釈	7-13
7.3.1	エラーの修正とアプリケーションの再試行	7-15
7.3.2	初期化されていない値の検出	7-15
7.3.3	ソース・ファイルの探索	7-16
7.4	アプリケーションによるヒープ使用の検査	7-17
7.4.1	メモリ・リークの検出	7-18
7.4.2	ヒープの読み取りとリーク・レポート	7-19
7.4.3	リークの探索	7-20
7.4.4	ヒープ・ヒストリの解釈	7-21
7.5	シンボル情報が不十分なプログラムにおける Third Degree の 使用	7-24
7.6	Third Degree エラー・レポートの有効性検査	7-25
7.7	検出されないエラー	7-26

8 プログラムのプロファイルによる性能の向上

8.1	プロファイルのサンプル・プログラム	8-2
8.2	プロファイルのコンパイラ・オプション	8-4
8.3	手動による設計とコードの最適化	8-5
8.3.1	手法	8-5
8.3.2	ツールと例	8-5
8.3.2.1	呼び出しグラフを使用した CPU 時間のプロファイル	8-5
8.3.2.1.1	hiprof プロファイラを使用する方法	8-6
8.3.2.1.2	cc コマンドの -pg オプションを使用する方法	8-11
8.3.2.2	ソース行または命令の、CPU 時間またはイベントのプ ロファイル	8-13
8.3.2.2.1	uprofile プロファイラを使用する方法	8-13
8.3.2.2.2	hiprof プロファイラを使用する方法	8-17

8.3.2.2.3	cc コマンドの -p オプションを使用する方法	8-18
8.3.2.2.4	pixie プロファイラを使用する方法	8-21
8.4	システム・リソース使用の最小化	8-23
8.4.1	手法	8-23
8.4.2	ツールと例	8-24
8.4.2.1	システム・モニタ	8-24
8.4.2.2	ヒープ・メモリの解析	8-25
8.5	テスト・ケースの重要性の確認	8-27
8.5.1	手法	8-27
8.5.2	ツールと例題	8-27
8.6	表示するプロファイル情報の選択	8-28
8.6.1	プロファイルの表示を特定プロシージャのみに制限する ..	8-29
8.6.2	ソース行ごとのプロファイル情報の表示	8-29
8.6.3	行ごとのプロファイル表示の制限	8-30
8.6.4	プロファイル情報にシェアード・ライブラリを含める	8-30
8.6.4.1	計測機構付きシェアード・ライブラリの位置の指定 ..	8-31
8.7	プロファイル・データ・ファイルのマージ	8-31
8.7.1	データ・ファイルの命名規則	8-32
8.7.2	データ・ファイルのマージ手法	8-33
8.8	マルチスレッド・アプリケーションのプロファイル	8-34
8.9	monitor ルーチンを使用したプロファイルの制御	8-36

9 Atom ツールの使用および開発

9.1	Atom ツールの実行	9-1
9.1.1	インストール済みの Atom ツールの使用	9-2
9.1.2	開発中のテスト用ツール	9-4
9.1.3	Atom オプション	9-5
9.2	Atom ツールの開発	9-10
9.2.1	Atom によるアプリケーションの表示	9-10

9.2.2	Atom の計測ルーチン	9-11
9.2.3	Atom の計測インタフェース	9-12
9.2.3.1	プログラム内のナビゲーション	9-12
9.2.3.2	オブジェクトの作成	9-14
9.2.3.3	アプリケーションの構成要素に関する情報の取得	9-14
9.2.3.4	名前および呼び出しターゲットの解決	9-18
9.2.3.5	分析ルーチン呼び出しの追加	9-19
9.2.3.6	エントリ・ポイント呼び出しへの介入	9-20
9.2.4	Atom の記述ファイル	9-21
9.2.5	分析プロシージャの作成	9-22
9.2.5.1	入出力	9-23
9.2.5.2	fork および exec システム・コール	9-24
9.2.6	置換された呼び出し側アプリケーションのエントリ・ポ イント	9-24
9.2.7	分析ルーチンからの計測機構付き PC の決定	9-26
9.2.8	サンプル・ツール	9-32
9.2.8.1	プロシージャ・トレース	9-32
9.2.8.2	プロファイル・ツール	9-35
9.2.8.3	データ・キャッシュ・シミュレーション・ツール	9-39

10 プログラムの最適化

10.1	アプリケーション・プログラム作成のガイドライン	10-2
10.1.1	コンパイルに関する考慮事項	10-2
10.1.2	リンクおよびロードに関する考慮事項	10-7
10.1.3	spike およびプロファイル主導の最適化	10-7
10.1.3.1	spike の概要	10-7
10.1.3.2	プロファイル主導の最適化での spike の使用	10-9
10.1.4	前処理と後処理に関する考慮事項	10-13
10.1.5	ライブラリ・ルーチンの選択	10-15

10.2	アプリケーションのコーディング上のガイドライン	10-16
10.2.1	データ型についての考慮事項	10-17
10.2.2	AdvFS ファイルでの直接入出力の使用	10-17
10.2.3	キャッシュ使用とデータの境界合わせに関する考慮事項 .	10-19
10.2.4	一般的なコーディングに関する考慮事項	10-21
11	例外条件の処理	
11.1	例外処理の概要	11-1
11.1.1	C コンパイラ構文	11-2
11.1.2	libexc ライブラリ・ルーチン	11-2
11.1.3	例外処理をサポートするヘッダ・ファイル	11-4
11.2	ユーザ・プログラムで起こす例外	11-4
11.3	構造化例外ハンドラの作成	11-5
11.4	終了ハンドラの作成	11-14
12	スレッド・セーフなライブラリの開発	
12.1	スレッド・サポートの概要	12-1
12.2	POSIX 準拠のための実行時ライブラリの変更	12-3
12.3	スレッド・セーフ・ルーチンおよびリエントラント・ルーチン の特性	12-4
12.3.1	スレッド・セーフでないコーディング例	12-4
12.4	スレッド・セーフ・コードの作成	12-6
12.4.1	スレッド固有データに対する TIS の使用	12-7
12.4.1.1	TIS の概要	12-7
12.4.1.2	スレッド固有データの使用	12-7
12.4.2	TLS (Thread Local Storage) の使用	12-9
12.4.2.1	__thread 属性	12-10
12.4.2.2	ガイドラインと制限	12-10
12.4.3	スレッド間でデータを共用するためのミューテックス・ ロックの使用	12-12

12.5	マルチスレッド・アプリケーションの作成	12-13
12.5.1	マルチスレッド C アプリケーションのコンパイル	12-13
12.5.2	マルチスレッド C アプリケーションのリンク	12-14
12.5.3	その他の言語のマルチスレッド・アプリケーションの作成	12-14
13	OpenMP 並列処理	
13.1	コンパイル・オプション	13-1
13.2	環境変数	13-4
13.3	実行時性能のチューニング	13-4
13.3.1	スケジュール・タイプとチャンクサイズの設定	13-5
13.3.2	その他の制御	13-6
13.4	プログラミング上の一般的な問題	13-7
13.4.1	範囲指定	13-7
13.4.2	デッドロック	13-7
13.4.3	threadprivate ストレージ	13-8
13.4.4	ロックの使用	13-8
13.5	インプリメンテーション固有の動作	13-8
13.6	デバッグ	13-9
13.6.1	デバッグに必要な背景知識	13-9
13.6.2	デバッグおよびアプリケーション分析のツール	13-12
13.6.2.1	Ladefug	13-12
13.6.2.2	Visual Threads	13-14
13.6.2.3	Atom および OpenMP ツール	13-15
13.6.2.4	その他のデバッグ支援機能	13-15
14	EVM イベントの発信と受信	
14.1	イベントとイベント管理	14-2
14.2	EVM イベント処理の概要	14-4
14.3	EVM の起動と停止	14-5

14.4	イベントの発信とアクセスの権限	14-6
14.5	EVM イベントの内容	14-6
14.5.1	標準データ項目	14-7
14.5.1.1	イベント名データ項目	14-8
14.5.1.1.1	予約コンポーネント名	14-11
14.5.1.1.2	イベント名の比較	14-13
14.5.1.2	イベントのフォーマット・データ項目	14-13
14.5.1.3	イベントの優先度データ項目	14-15
14.5.1.4	I18N カタログ名, メッセージ・セット ID, および メッセージ ID データ項目	14-16
14.5.1.5	クラスタ・イベント・データ項目	14-17
14.5.1.6	参照データ項目	14-19
14.5.2	変数データ項目	14-19
14.6	イベント・セットの設計	14-21
14.6.1	イベントに値する状態変更の決定	14-22
14.6.2	イベントの説明テキストの作成	14-23
14.6.3	イベント・テンプレートの設計	14-24
14.6.3.1	イベント・テンプレートに設定する項目の決定	14-25
14.6.3.2	イベント・テンプレート名と発信イベントの名前の照 合	14-26
14.6.3.3	テンプレートと発信イベントのデータ項目のマージ ..	14-27
14.6.3.4	テンプレート・ファイルのインストール — 位置, 命 名, 所有権, および許可の要件	14-28
14.6.3.5	イベント・テンプレートの登録の確認	14-29
14.6.4	イベント・テキストの翻訳の設定 (I18N)	14-29
14.7	EVM プログラミング・インタフェース	14-32
14.7.1	EVM ヘッダ・ファイル	14-32
14.7.2	EVM API ライブラリ	14-32
14.7.3	戻り状態コード	14-33

14.7.4	シグナルの処理	14-33
14.7.5	マルチスレッド・プログラムでの EVM	14-34
14.7.6	EVM イベントの再割り当てと複製	14-34
14.7.7	コールバック関数	14-35
14.7.8	接続ポリシーの選択	14-36
14.7.9	切断の処理	14-37
14.7.10	失われたイベント	14-38
14.7.11	イベント・フィルタの使用	14-40
14.7.12	EVM プログラミング操作の例	14-40
14.7.12.1	簡単なイベント操作の実行	14-41
14.7.12.2	可変長の引数リストの使用	14-43
14.7.12.3	変数の追加と取得	14-44
14.7.12.4	イベントの発信	14-47
14.7.12.5	イベントの読み取りと書き込み	14-49
14.7.12.6	イベント通知の受信	14-51
14.7.12.7	複数の入出力ソースの処理	14-55
14.7.12.8	フィルタ・エバリュエータの使用	14-58
14.7.12.9	イベント名の照合	14-62
14.7.12.10	失われたイベントの処理	14-63
14.8	EVM へのイベント・チャンネルの追加	14-65
14.8.1	取得関数	14-67
14.8.2	詳細関数	14-70
14.8.3	説明関数	14-70
14.8.4	監視関数	14-71
14.8.5	クリーンアップ関数	14-73
14.8.6	チャンネルのセキュリティ	14-74

A Tru64 UNIX システムにおける 32 ビット・ポインタの使用

A.1	コンパイラ・システムと 32 ビット・ポインタの言語サポート	A-2
-----	--------------------------------	-----

A.2	-taso オプションの使用	A-3
A.2.1	-taso オプションの使用と効果	A-5
A.2.2	-taso オプションの効果に対する制限事項	A-8
A.2.3	taso 環境での malloc の動作	A-9
A.3	-xtaso または -xtaso_short オプションの使用	A-10
A.3.1	ポインタ・サイズの変更に関するコーディング上の注意事 項	A-10
A.3.2	32 ビット・ポインタの使用に関する制限事項	A-11
A.3.3	システム・ヘッダ・ファイルに関する問題の回避	A-12
 B System V 実行環境における相違点		
B.1	ソース・コードの互換性	B-1
B.2	システム・コールとライブラリ・ルーチンの要約	B-4
 C 動的に構成可能なカーネル・サブシステムの作成		
C.1	動的に構成可能なサブシステムの概要	C-2
C.2	属性テーブルの概要	C-5
C.2.1	定義属性テーブル	C-6
C.2.2	定義属性テーブルの例	C-8
C.2.3	通信属性テーブル	C-11
C.2.4	通信属性テーブルの例	C-13
C.3	構成ルーチンの作成	C-13
C.3.1	初期構成の実行	C-14
C.3.2	照会要求に対する応答	C-16
C.3.3	再構成要求に対する応答	C-19
C.3.4	サブシステムで定義した操作の実行	C-22
C.3.5	サブシステムの構成除外	C-23
C.3.6	構成ルーチンの終了	C-23
C.4	オペレーティング・システムのリビジョンの確認	C-24
C.5	ロード可能なサブシステムの構築とロード	C-25

C.6	静的な構成可能サブシステムのカーネルへの構築	C-27
C.7	サブシステムのテスト	C-30
D 並列処理 — 従来の方法		
D.1	並列処理プラグマの使用	D-2
D.1.1	一般的なコーディング規則	D-2
D.1.2	一般的な使用法	D-3
D.1.3	並列指示文のネスト	D-5
D.2	並列処理プラグマの構文	D-5
D.2.1	#pragma parallel	D-6
D.2.2	#pragma pfor	D-9
D.2.3	#pragma psection および #pragma section	D-10
D.2.4	#pragma critical	D-10
D.2.5	#pragma one processor	D-11
D.2.6	#pragma synchronize	D-11
D.2.7	#pragma enter gate および #pragma exit gate	D-11
D.3	環境変数	D-12
E デバイス特殊ファイル名の処理		
F -om および cord によるプログラムの最適化		
F.1	-om ポストリンク最適化プログラムの使用	F-1
F.1.1	概要	F-1
F.1.2	-om によるプロファイル主導の最適化	F-2
F.2	-cord によるプロファイル主導の再編成	F-5
索引		
例		
5-1	dbx の例で使用されているサンプル・プログラム	5-73
8-1	プロファイルのサンプル・プログラム	8-2

8-2	gprof を使用した hiprof の省略時のプロファイル例	8-7
8-3	gprof を使用した hiprof の -cycles プロファイルの例	8-10
8-4	gprof を使用した cc -pg プロファイルの例	8-12
8-5	prof を使用した uprofile の CPU 時間プロファイルの例	8-14
8-6	prof を使用した uprofile のデータ・キャッシュ・ミス・プロ ファイルの例	8-15
8-7	hiprof -lines による PC サンプリング・プロファイルの例	8-17
8-8	prof を使用した cc -p プロファイルの例	8-20
8-9	prof を使用した pixie プロファイルの例	8-22
8-10	third ログ・ファイルの例	8-25
8-11	monstartup() と monitor() の使用	8-37
8-12	プログラム内のプロファイル・バッファの割り当て	8-38
8-13	monitor_signal() を使用した、終了しないプログラムのプロ ファイル	8-40
10-1	ポインタと最適化	10-25
11-1	構造化例外としての SIGSEGV シグナルの処理	11-8
11-2	構造化例外としての IEEE 浮動小数点 SIGFPE の処理	11-10
11-3	複数の構造化例外ハンドラ	11-12
11-4	例外による try 本体の異常終了	11-16
12-1	スレッド・プログラム例	12-8
14-1	イベントの説明テキストの例	14-24
14-2	簡単なイベント操作の実行	14-41
14-3	可変長の引数リストの使用	14-43
14-4	変数の追加と取得	14-45
14-5	イベントの発信	14-48
14-6	イベントの読み取りと書き込み	14-50
14-7	イベント通知の受信	14-52
14-8	複数の入出力ソースの処理	14-56
14-9	フィルタ・エバリュエータの使用	14-59

14-10	イベント名の照合	14-62
14-11	失われたイベントの処理	14-63
C-1	定義属性テーブルの例	C-9
C-2	通信属性テーブル	C-11
C-3	初期構成の実行	C-15
C-4	照会要求に対する応答	C-17
C-5	再構成要求に対する応答	C-19

図

2-1	プログラムのコンパイル	2-3
2-2	省略時の構造体の位置合わせ	2-7
2-3	省略時のビット・フィールドの位置合わせ	2-8
2-4	次のバック境界までの埋め込み	2-9
4-1	アーカイブ・ライブラリとシェアード・ライブラリ	4-3
4-2	シェアード・ライブラリの複数バージョンとのリンク	4-28
4-3	共用オブジェクト間の無効な複数バージョンの従属: 例 1	4-31
4-4	共用オブジェクト間の無効な複数バージョンの従属: 例 2	4-32
4-5	共用オブジェクト間の無効な複数バージョンの従属: 例 3	4-33
4-6	シェアード・ライブラリの複数バージョンの有効な使用: 例 1	4-34
4-7	シェアード・ライブラリの複数バージョンの有効な使用: 例 2	4-35
14-1	EVM の概要	14-4
14-2	発信イベントとテンプレートのマージ	14-28
A-1	-tas0 オプションを使用した場合のメモリのレイアウト	A-6
B-1	システム・コールの解決	B-2
C-1	システム属性値の初期化	C-3

表

1-1	プログラミングのフェーズと Tru64 UNIX	1-1
2-1	コンパイラ・システムの機能	2-2

2-2	ファイルの接尾語と対応ファイル	2-4
2-3	cc コマンドのオプション・カテゴリごとの省略時オプション	2-18
3-1	Intrinsics 関数	3-17
4-1	シェアード・ライブラリのバージョンを管理するリンカ・オプション	4-24
5-1	コマンド構文の記述に使用されるキーワード	5-2
5-2	dbx コマンド・オプション	5-10
5-3	dbx の # 式演算子	5-13
5-4	C の式演算子	5-13
5-5	組み込みデータ型	5-13
5-6	入力可能な定数	5-14
5-7	emacs モードの dbx コマンド行編集コマンド	5-17
5-8	定義済みの dbx 変数	5-21
5-9	メモリ・アドレス表示モード	5-60
6-1	lint 警告クラス	6-26
9-1	サンプルのインストール済み Atom ツール	9-2
9-2	Atom のプログラム照会ルーチン	9-14
9-3	Atom のオブジェクト照会ルーチン	9-15
9-4	Atom のプロシージャ照会ルーチン	9-17
9-5	Atom エントリ・ポイント照会ルーチン	9-17
9-6	Atom の基本ブロック照会ルーチン	9-18
9-7	Atom の命令照会ルーチン	9-18
11-1	例外処理をサポートするヘッダ・ファイル	11-4
14-1	標準データ項目	14-7
14-2	イベント・テキストへの変数代入	14-15
14-3	EVM の変数データの型	14-20
14-4	名前照合の例	14-27
14-5	多国語対応のイベントに対するデータ項目値の例	14-30
B-1	システム・コールの要約	B-4

B-2	ライブラリ関数の要約	B-5
C-1	属性データ型	C-7
C-2	属性に対して許可される要求を指定するコード	C-7
C-3	属性状態コード	C-12



まえがき

本書は、C プログラミング言語を中心に、HP Tru64 UNIX オペレーティング・システムのプログラミング環境について説明します。C 以外のプログラミング言語については、システムの設定時あるいは変更時に選択することにより利用することができます。

本書の対象読者

本書は、Tru64 UNIX オペレーティング・システムを使用し、サポートされた言語でプログラムの作成および管理を行う、すべてのプログラマを対象読者にしています。

追加および変更された機能

本書は、前回のリリースの後に次のような追加と変更が行われています。

- 第 4 章 — 4.2.4 項 (未解決の外部シンボルの処理のオプション) に、`-error_unresolved` オプションが有効な場合の `.so` ファイルの処理方法に関する説明を記載しました。
- 第 9 章 — この章は大幅に改訂し、Atom の新しいオプションおよびルーチンの説明を加え、9.2.3.6 項 (エントリ・ポイントへの介入呼び出し) と 9.2.6 項 (置換された呼び出し側アプリケーションのエントリ・ポイント) の 2 つの項を追加しました。
- 第 14 章 — 14.5.1.5 項 (クラスタ・イベント・データ項目) と 14.7.10 項 (失われたイベント) の 2 つの項を追加しました。他にも、章全体を通じて細かい改訂と追加を行っています。

本書の構成

本書の構成は次のとおりです。

- | | |
|-------|--|
| 第 1 章 | プログラム開発のフェーズ、およびそれらのフェーズで使用するプログラミング・ツールについて説明します。 |
|-------|--|

第 2 章	コンパイラ・システムを構成するツールおよびその使用方法について説明します。コンパイラ・コマンド、プリプロセッサ、コンパイラ・オプション、多言語プログラム、およびアーカイバについて説明します。
第 3 章	C コンパイラでサポートされている処理系固有のプログラマについて説明します。
第 4 章	シェアード・ライブラリの用途、作成方法、および保守、ならびにシンボルの解決方法について説明します。
第 5 章	dbx デバッガの使用方法について説明します。また、dbx コマンド、モニタでの動作、ブレークポイントの設定、機械語コードのデバッグについても説明します。
第 6 章	コーディング・エラーのないプログラムを作成するための lint コマンドの使用方法について説明します。
第 7 章	Third Degree ツールを使用して、アプリケーション・プログラムのメモリ・アクセス・チェックおよびリーク検出を行う方法について説明します。
第 8 章	作成したコードのプロファイルを行うためのさまざまなツールや技術の使用方法について説明し、プログラムのどの部分の処理に最も時間を費やしているかを調べることができますようにします。また、プロファイル・データを C コンパイラにフィードバックして、コードの自動最適化を提供する方法についても説明します。
第 9 章	パッケージされている Atom ツールを使用して、プロファイル・データの取得やキャッシュ使用分析の実行など、アプリケーションに対してさまざまな目的で計測を行う方法について説明します。また、この章では、カスタム Atom ツールの設計および作成方法についても説明します。
第 10 章	最適化ツールおよびポストリンク最適化ツール spike によるプログラムの最適化について説明します。
第 11 章	C コンパイラの機能を利用した構造化例外ハンドラおよび終了ハンドラの作成方法について説明します。
第 12 章	マルチスレッド・プログラムの開発について説明します。
第 13 章	OpenMP 並行処理インタフェースを使用する際の考慮事項について説明します。
第 14 章	Event Manager ユーティリティを使用して、イベント情報を通知したり受信したりする方法について説明します。
付録 A	Tru64 UNIX システムにおける 32 ビット・ポインタの使用方法について説明します。
付録 B	System V 実行環境における C 言語プログラムのソース・コードの互換性について説明します。

付録 C	動的に構成可能なカーネル・サブシステムの作成方法について説明します。
付録 D	OpenMP の前に実装されていた古い形式の並行処理プラグマについて説明します。
付録 E	デバイス特殊ファイルの古い形式の名前と新しい形式の名前の変換処理を行うルーチンについて説明します。
付録 F	cc コマンドの <code>-om</code> および <code>-cord</code> オプションを用いてプログラムを最適化する方法について説明します。

関連資料

プログラム開発に関する情報については、本書の他に次のマニュアルを参照してください。

プログラミング (全般):

『*Calling Standard for Alpha Systems*』

『*Assembly Language Programmer's Guide*』

『プログラミング・サポートツール・ガイド』

『ネットワーク・プログラミング・ガイド』

『*Compaq Portable Mathematics Library*』

『国際化ソフトウェア・プログラミング・ガイド』

『*Kernel Debugging*』

『*Ladebug Debugger Manual*』

『*Writing Kernel Modules*』

『*Alpha Architecture Reference Manual*』 第 2 版 (Butterworth-Hinemann Press, ISBN:1-55558-145-5)

プログラミング (リアルタイム):

『*Guide to Realtime Programming*』

プログラミング (STREAMS):

『*Programmer's Guide: STREAMS*』

プログラミング (マルチスレッド・アプリケーション):

『*Guide to the POSIX Threads Library*』

OpenMP C and C++ Application Programming Interface 仕様は、インターネットの <http://www.openmp.org/specs/> にあります。

一般ユーザ情報:

『リリース・ノート』

本書で使用する表記法

本書では、次の表記規約を使用します。

%

\$

パーセント記号は、C シェルのシステム・プロンプトを表します。ドル記号は、Bourne シェル、Korn シェル、および POSIX シェルの場合のシステム・プロンプトを表します。

#

番号記号は root としてログインした場合のシステム・プロンプトを表します。

% **cat**

対話式の例における太字(ボールド体)は、ユーザが入力する文字を示します。

file

イタリック体(斜体)は、変数値、プレースホルダ、および関数の引数名を示します。

[|]

{ | }

構文定義では、大カッコはオプションの項目を示し、中カッコは必須項目を示します。大カッコまたは中カッコの中の項目を縦線で区切っている場合は、そこに併記されている項目の中から1つの項目を選択することを示します。

...

構文定義では、水平の反復記号は、前の項目を1回以上繰り返して使用できることを示します。

cat(1)

リファレンス・ページの参照には、該当するセクション番号をカッコ内に示します。たとえば、cat(1) は、cat コマンドについての情報が、リファレンス・ページのセクション 1 に記載されていることを示します。

Return

四角で囲まれたキー名はユーザがそのキーを押すことを示します。

Ctrl/x

この記号は、スラッシュの前に指定されているキーを押しながら、スラッシュの後のキーまたはマウス・ボタンを押すことを示します。例中では、このようなキーの組み合わせは、四角あるいは大カッコで囲まれて示されます(たとえば、Ctrl/C)。



この章では、アプリケーション開発プロジェクトのさまざまなフェーズについて説明するとともに、これらのフェーズで使用する Tru64 UNIX のツールについて説明します。

この章では、以下に示すトピックについて説明します。

- アプリケーション開発のフェーズ (1.1 節)
- 仕様および設計に関する留意事項 (1.2 節)
- 主なソフトウェア開発ツール (1.3 節)
- ソース・ファイル制御 (1.4 節)
- プログラム・インストール・ツール (1.5 節)
- プロセス間通信 (1.6 節)

1.1 アプリケーション開発のフェーズ

アプリケーション開発には、5 つの主要なフェーズがあります。表 1-1 では、これらのフェーズと、各フェーズで使用可能なツールおよび機能を説明します。

表 1-1: プログラミングのフェーズと Tru64 UNIX

フェーズ	ツール/機能
要件および仕様の決定	規格 国際化 機密保護
設計	ルーチン コーディング上の留意事項 ライブラリ 共通ファイル
開発 (実現)	vi, ex, ed, lint, grep, cxref, sed, time, dbx, third, ld, make, コンパイラ, スレッド

表 1-1: プログラミングのフェーズと Tru64 UNIX (続き)

フェーズ	ツール/機能
テスト	diff, シェル・スクリプト, pixie, prof
保守	setld, tar, sccs, rcs

多くの場合、Tru64 UNIX システムでは 1 つのジョブを実行するために 2 つ以上のツールが提供されています。どのツールおよびプログラミング言語を使用するかはユーザが選択することになります。

1.2 仕様および設計上の留意事項

アプリケーションを設計する場合、アプリケーションの特性に従って設計を行う必要があります。Tru64 UNIX は、アプリケーションを作成するために有用な機能やツールを備えており、移植性、国際化、ウィンドウ対応などの点で優れ、ユーザのニーズを十分に満たすアプリケーションを作成することができます。

設計の際の主な留意事項の 1 つに、UNIX 環境規格および移植性への対応に関する問題があります。Tru64 UNIX システム上だけでなく、他の UNIX のオペレーティング・システムでも実行できるようアプリケーションを作成する場合、X/Open の移植ガイドラインおよび POSIX 規格に準拠するように設計する必要があります。

また、さまざまな国で使用できるようにアプリケーションを設計する場合は、Tru64 UNIX オペレーティング・システムの国際化ツールおよび国際化機能を使用してソフトウェアを作成してください。

アプリケーションが使用される端末環境についても考慮しておく必要があります。エンド・ユーザがワークステーションまたはウィンドウ端末を使用する場合、ウィンドウ・ディスプレイが使用できるようにアプリケーションを設計する必要があります。

1.2.1 規格

プログラミング規格に準拠するようにアプリケーションを作成することにより、ハードウェア間で、あるいはオペレーティング・システム間でのプログラムやアプリケーションの移植性を向上させることができます。移植性規格に準拠してプログラムを記述すると、ユーザはシステム間を容易に移

動できます。規格の中にはプログラムの移植性を構成する一部として、国際化の考え方が含まれている場合もあります。

UNIX プログラミング環境での主な規格は次のとおりです。

- ANSI
- ISO
- POSIX
- X/Open

これらの規格以外に、OSF アプリケーション環境仕様 (AES) では、アプリケーション・レベルのインタフェースが指定され、それによりアプリケーションは、このインタフェースによって指定された移植可能なアプリケーション、意味規則、およびプロトコルをサポートするよう規定されています。

ANSI 規格は、プログラミング言語、ネットワーク、通信プロトコル、文字コード化、およびデータベース・システムなど、特定のプログラミング・ツールに適用されます。個々の ANSI 規格への適合性と拡張機能の情報については、各言語、ネットワーク・システム、またはデータベース・システムのドキュメンテーション・セットを参照してください。ANSI 規格に準拠するように C プログラムをコンパイルする方法については、第 2 章を参照してください。

Tru64 UNIX システムでは、POSIX および X/Open 規格に対応するプログラムを記述することができます。POSIX 規格についての詳細は、IEEE Std. 1003.1c-1994 の『POSIX -- Part 1: System Application Program Interface (API) [C Language]』(ISBN 1-55937-061-0) を参照してください。Tru64 UNIX ヘッダ・ファイルには、POSIX および X/Open に対応する情報があります。

1.2.2 国際化

国際化されたアプリケーションは、実行時インタフェースを提供することにより、ユーザが母国語を使用し、文化的に適切に表現されたデータで作業ができるようにします。Tru64 UNIX オペレーティング・システムでは、X/Open CAE 仕様の Issue 4 に準拠する国際化されたアプリケーション開発のためのインタフェースとユーティリティを提供します。また、X/Open CAE 仕様の Issue 5 の一部である ISO C のマルチバイト・サポート拡張 (MSE) もサポートします。

国際化されたアプリケーションを開発する際には、次の点を考慮します。

- 言語
- 文化的データ
- 文字セット
- 地域化

こうした国際化の要件を満たすには、言語、慣習、またはコード化文字セットのいずれにも依存しないアプリケーションを作成する必要があります。地域文化に特有のデータは、アプリケーション論理から区別して保持します。実行時関数を使用して、アプリケーションに適切な言語メッセージ・テキストを対応させてください。

Tru64 UNIX 上での国際化プログラミングについては、『国際化ソフトウェア・プログラミング・ガイド』を参照してください。

1.2.3 ウィンドウ・アプリケーション

ウィンドウ・アプリケーションの開発については、次のマニュアルを参照してください。

『*OSF/Motif Programmer's Guide*』

『*Common Desktop Environment: プログラマーズ・ガイド*』

『*Common Desktop Environment: プログラマ概要*』

『*Common Desktop Environment: アプリケーション・ビルダ・ユーザーズ・ガイド*』

『*Common Desktop Environment: プログラマーズ・ガイド（国際化対応編）*』

『*Common Desktop Environment: スタイル・ガイド*』

『*Common Desktop Environment: プログラマーズ・ガイド（ヘルプ・システム編）*』

1.2.4 アプリケーションの保護

『セキュリティ・プログラミング・ガイド』には、トラステッド・プログラムを作成するための詳細な説明が、あらゆる側面から記載されています。

Tru64 UNIX では、オペレーティング・システム上にローカルおよび分散セキュリティ認証メカニズムの階層化を可能にする Security Integration Architecture (SIA) を提供しています。SIA 構成フレームワークは、特定のセキュリティ・メカニズムからセキュリティに依存するコマンドを分離します。詳細については、『セキュリティ管理ガイド』のパスワードに関する付録、および sia*(3) リファレンス・ページを参照してください。

1.3 ソフトウェア開発の主なツール

Tru64 UNIX システムは、多数の高水準言語と互換性があり、リンクやプログラムのデバッグを行うツールを備えています。

1.3.1 Tru64 UNIX 環境でサポートされる言語

Tru64 UNIX オペレーティング・システムには、アセンブラ (アセンブリ言語プログラム用) および Java 開発キット (JDK) が含まれています。C, C++, Fortran, Ada, および Pascal などの他の言語のコンパイラは、別途注文してください。

オプションのプロダクトについての詳細は、弊社の各支店および営業所にお問い合わせください。

Java についての詳細は、JDK がインストールされているシステム上の次のディレクトリにある Java のドキュメントを参照してください。

`/usr/share/doclib/java/index.html`

アセンブラについての詳細は、as(1) および『*Assembly Language Programmer's Guide*』を参照してください。

その他の言語のマニュアルは、各言語のコンパイラを注文するときに一緒に注文できます。

1.3.2 オブジェクト・ファイルのリンク

通常、C コンパイラ・ドライバ・コマンド (cc) を使用して、別個のオブジェクト・ファイルを 1 つの実行可能オブジェクト・ファイルにリンクすることができます。

コンパイルのプロセスの一部として、ほとんどのコンパイラ・ドライバは、リンカ (ld) を呼び出して、1 つまたは複数のオブジェクト・ファイルを単一の実行可能オブジェクト・ファイルに結合します。また、リンカは外部参照

の解釈、ライブラリの探索など、実行可能なオブジェクト・ファイルの作成に必要とされるその他の処理をすべて行います。

開発環境では、さまざまな言語で記述されたソース・コードから構成されるアプリケーションを作成することができます。この場合、それぞれのファイルを別々にコンパイルし、次に、別のステップでコンパイルされたオブジェクト・ファイルをリンクすることになります。リンクは、`ld` コマンドを入力することにより、コンパイラとは別に起動します。

シェアード・ライブラリは、コンパイラ・ドライバ・コマンドまたは `ld` コマンドを使用して作成することができます。また、`ar` コマンドを使用してアーカイブ (静的) ライブラリを作成することもできます。ライブラリの作成方法についての詳細は、第 4 章を参照してください。プログラムのコンパイルおよびリンクについての詳細は、第 2 章および第 4 章と、各言語のドキュメンテーション・セットを参照してください。

1.3.3 デバッグおよびプログラム分析ツール

Tru64 UNIX オペレーティング・システムの主なデバッグ・ツールは次のとおりです。

- `dbx` デバッガ (第 5 章または `dbx(1)` を参照)
- プログラム・プロファイル・ツール (第 8 章または `hiprof(1)`, `pixie(1)`, `uprofile(1)`, `gprof(1)` および `prof(1)` を参照)
- Third Degree ツール (第 7 章または `third(1)` を参照)
- `lint` ユーティリティ (第 6 章または `lint(1)` を参照)

`ladebug` デバッガは Tru64 UNIX オペレーティング・システムでもサポートされています。`dbx` デバッガで提供される機能のサポートに加え、マルチスレッド・プログラムのデバッグ機能もサポートしています。C, C++, および Fortran をサポートする `ladebug` デバッガについては、『*Ladebug Debugger Manual*』および `ladebug(1)` を参照してください。

1.4 ソース・ファイル制御

ソフトウェア・アプリケーションの作成に必須の作業として、開発および保守プロセスの管理があります。Tru64 UNIX オペレーティング・システムは、ソース・コード制御システム (SCCS) ユーティリティおよび RCS コード管理システムを備えており、ディレクトリにアプリケーション・モジュール

を保存したり、それらのモジュール・ファイルに行われた変更内容を追跡したり、ユーザによるファイルのアクセスをモニタできるようにします。

Tru64 UNIX オペレーティング・システムの SCCS および RCS も、他の UNIX システムの SCCS および RCS ユーティリティと同様なサポートを行っています。他に、`sccs` プリプロセッサも備えており、これが従来の SCCS コマンドとのインタフェースとなっています。

SCCS および RCS は、このユーティリティを使用して保存されたファイルの変更のレコードを保守します。レコードには、変更が行われた理由、時期、および変更の実施者の情報が含まれています。SCCS あるいは RCS を使用すると、複数のバージョンを同時に保守するだけでなく、前バージョンのファイルを復元することもできます。SCCS はアプリケーション・プロジェクトの管理に適しています。SCCS では、複数の人間が、同じファイルを同時に変更することができないためです。

SCCS と `sccs` コマンドの使用方法の詳細については、`sccs(1)`、`rccs(1)`、および『プログラミング・サポートツール・ガイド』を参照してください。

1.5 プログラムのインストール・ツール

プログラムやアプリケーションを作成した後、他のユーザに簡単に配布できるように、`setld` インストレーション・ユーティリティで処理可能なキットの形態にパッケージ化することができます。Tru64 UNIX オペレーティング・システムの多様なユーティリティで、プログラムやアプリケーションのインストール、削除、結合、検査、構成などを行うことができます。

Tru64 UNIX システムのソフトウェアでは、ファイルやディレクトリが階層状のグループを構成しています。ユーザのアプリケーションやプログラムが 2 つ以上のファイルまたはディレクトリで構成されている場合は、階層内のファイルやディレクトリをどのように分類するか決定する必要があります。個々の製品の階層を開発システムから運用システムに転送する場合、つまり製品をインストールする場合にも、`setld` インストレーション・プロセスでは、各製品の階層を完全に保持します。キットの作成プロセスには、これらの製品の構成ファイルをサブセットに分類する作業も含まれています。それらのサブセットの中から、必要なものをシステム管理者が選択してインストールすることができます。

`setld` ユーティリティおよび関連ツールを使用する利点は次のとおりです。

- インストール・セキュリティ

setld ユーティリティは、各サブセットがあるシステムから別のシステムに転送された直後に、転送が正常に終了したことを確認します。各サブセットは、損傷を受けたり、削除されたりしても再度インストールすることができます。

- 柔軟性

システム管理者は、インストールするオプション・サブセットを選択することができます。また、サブセットを削除したり、必要があれば再びインストールすることもできます。このような機能を利用することにより、アプリケーションで複数の言語をサポートしたり、ユーザによるアプリケーションのオプション機能の選択が可能になります。

- 統一性

setld ユーティリティは、Tru64 UNIX インストレーションを行う上で必要不可欠なものです。

setld を使用することにより、システムへのインストールのための配布メディアとして、次のいずれのメディアにもアプリケーションをロードすることができます。

- CD-ROM
- サポートされているデータ・ディスク上の、任意のマウント可能なファイル・システム。たとえば、他社の SCSI ディスク・カートリッジなど。

setld コマンドの使用方法およびソフトウェア製品キットの管理と作成についての詳細は、『プログラミング・サポートツール・ガイド』を参照してください。

1.6 プロセス間通信機能の概要

プロセス間通信 (IPC) とは 2 つ以上のプロセス間でのデータの交換のことです。単一プロセス・プログラミングでは、単一プロセス内でのモジュールがグローバル変数および関数呼び出しを使用して相互に通信し、関数と呼び出す側の間でデータを受け渡します。別々のアドレス空間にイメージを持ち、別々の処理を使用してプログラムを作成する場合は、別の通信機構を使用することが必要になります。

Tru64 UNIX には、プロセス間通信のための次の機能が提供されています。

- System V IPC

System V IPC には、メッセージ、共用メモリ、およびセマフォのような IPC 機能があります。詳細については、System V のマニュアル、`mq_open(3)`、`shmctl(2)`、`sem_open(3)` を参照してください。

- FIFO 特殊ファイル (名前付きパイプ)

FIFO 特殊ファイルについての詳細は、`mkfifo(3)` および `mknod(8)` を参照してください。

- シグナル

シグナルについての詳細は、『*Guide to Realtime Programming*』を参照してください。

- ソケット

ソケットについての詳細は、『ネットワーク・プログラミング・ガイド』を参照してください。

- STREAMS

STREAMS についての詳細は、『*Programmer's Guide: STREAMS*』を参照してください。

- スレッド

スレッドについての詳細は、『*Guide to the POSIX Threads Library*』および第 12 章を参照してください。

- X/Open トランスポート・インタフェース (XTI)

XTI についての詳細は、『ネットワーク・プログラミング・ガイド』を参照してください。



コンパイラ・システム

この章では、次の項目について説明します。

- コンパイラ・システムの構成要素 (2.1 節)
- Tru64 UNIX 環境におけるデータ型 (2.2 節)
- C 言語プリプロセッサ (2.3 節)
- ソース・プログラムのコンパイル (2.4 節)
- オブジェクト・ファイルのリンク (2.5 節)
- プログラムの実行 (2.6 節)
- オブジェクト・ファイルのツール (2.7 節)
- 標準 C ライブラリにおける ANSI 名前空間汚染のクリーンアップ (2.8 節)
- インライン・アセンブリ・コード ASM (2.9 節)

コンパイラ・システムはソース・コードを実行可能プログラムに変換します。これは、次のステップから構成されます。

- 前処理 — コンパイラ・システムは、マクロ定義の展開やソース・コードへのヘッダ・ファイルの取り込みなどの処理を行います。
- コンパイル — コンパイラ・システムは、ソース・ファイルまたは前処理済みのファイルを、`.o` ファイル接尾語を持つオブジェクト・ファイルに変換します。
- リンク — コンパイラ・システムはバイナリ・イメージを作成します。

これらのステップは、前処理、コンパイル、リンクの別々のコマンドによって実行することも、単一のオペレーションで、コンパイル中にコンパイラ・システムが適切なタイミングで各ツールを呼び出すことによって実行することもできます。

コンパイラ・システムには、この他にも、コンパイルしてリンクされたプログラムのデバッグや、作成されたオブジェクト・ファイルのチェック、

ルーチンのライブラリの作成，プログラムの実行時性能の分析などを行うツールがあります。

表 2-1 に，コンパイラ・システムのツールと，本書および他のマニュアルの参照箇所を示します。

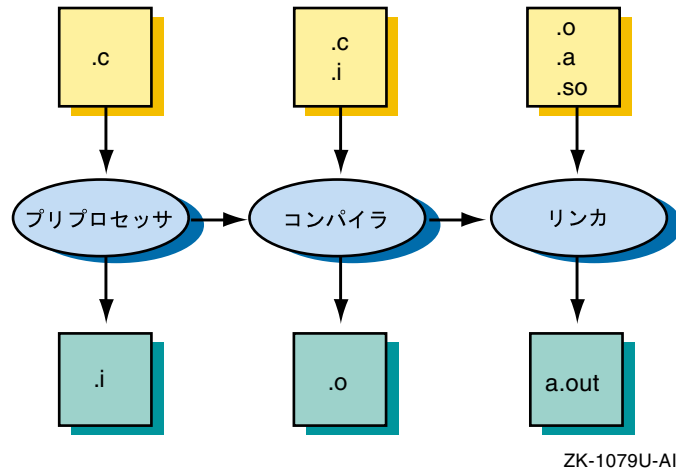
表 2-1: コンパイラ・システムの機能

目的	使用ツール	参照箇所およびマニュアル
プログラムのコンパイル，リンク，ロードおよびシェアード・ライブラリの作成	コンパイラ・ドライバ，リンク・エディタ，動的ローダ	この章，第 4 章，cc(1), c89(1), as(1), ld(1), loader(5), 『 <i>Assembly Language Programmer's Guide</i> 』，『 <i>Compaq C 言語リファレンス・マニュアル</i> 』
プログラムのデバッグ	シンボリック・デバッガ (dbx および ladebug) および Third Degree	第 5 章，第 6 章，第 7 章，dbx(1), third(1), ladebug(1), 『 <i>Ladebug Debugger Manual</i> 』
プログラムのプロファイル	プロファイラ，コール・グラフ・プロファイラ	第 8 章，hiprof(1), pixie(1), gprof(1), prof(1), atom(1)
プログラムの最適化	最適化プログラム，ポストリンク最適化プログラム	この章，第 10 章，cc(1), 第 7 章，third(1)
オブジェクト・ファイルの検査	nm, file, size, dis, odump, stdump のツール	この章，nm(1), file(1), size(1), dis(1), odump(1), stdump(1), 『 <i>プログラミン グ・サポートツール・ガイド</i> 』
必要なライブラリの作成	アーカイバ (ar)，リンカ (ld) のコマンド	この章，第 4 章，ar(1), ld(1)

2.1 コンパイラ・システムの構成要素

図 2-1 に，コンパイラ・システムの主な構成要素とそれらの入力と出力との関係を示します。

図 2-1: プログラムのコンパイル



ZK-1079U-AIJ

コンパイラ・システムのコマンドは、ドライバ・プログラムとも呼ばれ、コンパイラ・システムの構成要素を起動します。各言語にはそれぞれのコンパイラ・コマンドおよびオプションが用意されています。

C コンパイラは `cc` コマンドで起動します。Tru64 UNIX のプログラミング環境では、1 つの `cc` コンパイラ・コマンドで、次のような複数のアクションを実行することができます。

- 各ファイル名の接尾語に基づいて、適切なプリプロセッサ、コンパイラ（またはアセンブラ）、リンカのいずれを呼び出すかどうかを判断する。

コンパイラ・コマンドは、ファイル名に付けられた接尾語によって入力ファイルの内容を識別します。表 2-2 に、サポートされるファイルの接尾語を示します。

- ソース・ファイルをコンパイルおよびリンクして、実行可能プログラムを作成する。

複数のソース・ファイルが指定されている場合には、これらのファイルは、リンクされる前に他のコンパイラに渡されます。

- `as` アセンブラを呼び出すことにより、アセンブラ・コードを含むと考えられる 1 つ以上の `.s` ファイルをアセンブルし、その結果として出力されたオブジェクト・ファイルをリンクする。

`as` コマンドは、アセンブルされたオブジェクト・ファイルを自動的にリンクしないため、アセンブラを直接起動する場合には、別のステップでオブジェクト・ファイルをリンクする必要があります。

- リンクを行わない (すなわち実行可能プログラムを作成しない) よう指定する。そのため、後のリンク・オペレーションのために .o オブジェクト・ファイルを保管する。
- リンク・コマンド (ld) に関連する主要なオプションをリンカに渡す。
たとえば、cc コマンドに -L オプションを指定して実行すると、ライブラリを探索するためのディレクトリ・パスを指定することができます。各言語には、リンク時に別々のライブラリが必要になります。言語のドライバ・プログラムは、対応するライブラリをリンカに渡します。ライブラリとのリンクについての詳細は、第 4 章および 2.5.3 項を参照してください。
- a.out という省略時の名前、またはユーザが指定した名前で実行可能プログラム・ファイルを作成する。

表 2-2: ファイルの接尾語と対応ファイル

接尾語	ファイル
.a	アーカイブ・ライブラリ
.c	C ソース・コード
.i	この接尾語の付くファイルは C プリプロセッサによって処理された、ドライバ用のソース・コードであるとみなされる。たとえば % cc -c source.i のように実行する。ファイル source.i は、C ソース・コードを含んでいるとみなされる。
.o	オブジェクト・ファイル
.s	アセンブリ・ソース・コード
.so	共用オブジェクト (シェアード・ライブラリ)

2.2 Tru64 UNIX 環境におけるデータ型

以降の各項で、Tru64 UNIX システムでデータ項目を表現する方法について説明します。

注意

Tru64 UNIX システムでの省略時のメモリ・アクセス・サイズは 8 バイト (クォドワード) です。したがって、2 つ以上の実行スレッドが同時に隣接するメモリを変更する場合、そのメモリ・アドレ

スをクォードワードに整列させて、個々の変更で誤って上書きされないようにしなければなりません。たとえば、コンポジット・データ構造の同じクォードワードに含まれる別々のデータ項目が同時に変更されるような場合にエラーが生じることがあります。

クォードワード以外での整列や、その他の問題が生じるさまざまな状況については、『*Guide to the POSIX Threads Library*』の「Granularity Considerations」を参照してください。

2.2.1 データ型のサイズ

Tru64 UNIX システムは下位バイト処理型であり、右から左の順にバイトを使用します。C コンパイラでは、下位バイト処理型のバイト順のみをサポートしています。サポートするデータ型のサイズは次のとおりです。

データ型	ビットの大きさ
char	8
short	16
int	32
long	64
long long	64
float	32 (IEEE 単精度)
double	64 (IEEE 倍精度)
pointer	64 ^a
long double	128

^a32 ビット・ポインタは、`-xtaso_short` で利用可能。

2.2.2 浮動小数点の範囲と処理

C コンパイラは、『IEEE Standard for Binary Floating-Point Arithmetic』(ANSI/IEEE Std 754-1985) に定義されているように、IEEE 単精度 (32 ビット float)、倍精度 (64 ビット double)、拡張倍精度 (128 ビット long double) 浮動小数点データをサポートしています。

浮動少数点数は次の範囲になります。

- float: 1.17549435e-38f から 3.40282347e+38f まで

- `double`: 2.2250738585072014e-308 から 1.79769313486231570e+308 まで
- `long double`: 3.3621031431120935062626778173217526026e-4932 から 1.1897314953572317650857593266280070162e+4932

Tru64 UNIX では、規格に定義されている基本的な浮動小数点数フォーマット、演算 (加算, 減算, 乗算, 除算, 平方根, 剰余, 比較), および変換を提供しています。コンパイラ・オプションを指定することにより、完全に IEEE に準拠したトラップ動作 (NaN [番号でない] を含む) を使用することができます。また、IEEE 形式のトラップが必要ない場合には、速いモードを指定することができます。コンパイル時に丸めモードを選択して、IEEE 演算の結果に適用することもできます。IEEE 浮動少数点処理をサポートするオプションについての詳細は、`cc(1)` を参照してください。

ユーザ・プログラムでは、`ieee_set_fp_control()` を呼び出して、浮動小数点トラップのスレッドへの引き渡しを制御したり、`write_rnd()` を呼び出して、IEEE 丸めモードを動的に設定することができます。IEEE 浮動少数点の例外を処理する方法についての詳細は、`ieee(3)` を参照してください。

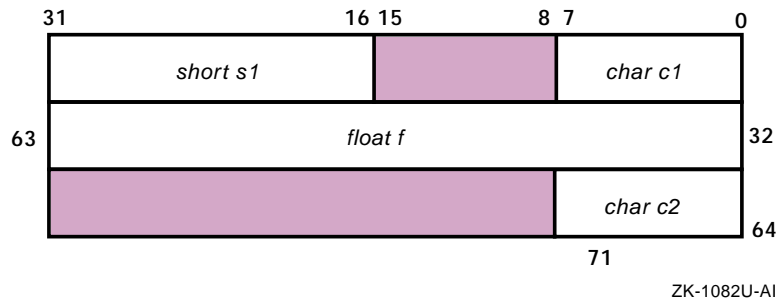
2.2.3 構造体の位置合わせ

C コンパイラは、省略時の設定では、構造体のメンバを自然境界に位置合わせします。つまり、構造体の構成要素は、宣言された順にメモリに配置されます。最初の構成要素は、構造体全体と同じアドレスを持ちます。それに続く各構成要素は、前の構成要素に続いて、その構成要素の型の次の自然境界に合わせられます。

たとえば、次の構造体は図 2-2 に示すように位置合わせが行なわれます。

```
struct {char c1;
        short s1;
        float f;
        char c2;
}
```

図 2-2: 省略時の構造体の位置合わせ



構造体の最初の構成要素 `c1` は、オフセット 0 から始まって、その最初のバイトを占めています。2 番目の構成要素の `s1` は、`short` であるため、ワード境界から始まる必要があります。したがって、`c1` と `s1` の間に埋め込みを行います。`f` と `c2` を自然境界に合わせるには、埋め込みは必要ありません。ただし、サイズが `f` の位置合わせの倍数に切り上げられるため、`c2` の後ろに 3 バイトの埋め込みが行われます。

構造体メンバの省略時の位置合わせの変更には、次のような方法を使用することができます。

- `#pragma member_alignment` および `#pragma nomember_alignment` 指示文
- `#pragma pack` 指示文
- `-Zpn` オプション

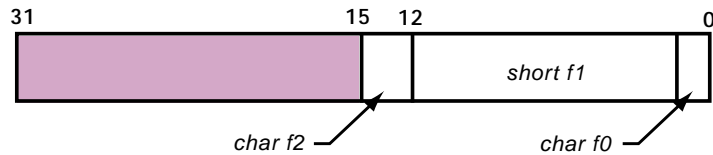
これらの指示文については、3.8 節および 3.11 節を参照してください。

2.2.4 ビット・フィールドの位置合わせ

一般に、ビット・フィールドの位置合わせは、その前のフィールドのビット・サイズとビット・オフセットによって決定されます。たとえば、次の構造体は、図 2-3 に示すような位置合わせが行われます。

```
struct a {
    char f0: 1;
    short f1: 12;
    char f2: 3;
} struct_a;
```

図 2-3: 省略時のビット・フィールドの位置合わせ



ZK-1080U-AI

最初のビット・フィールド `f0` は、ビット・オフセット 0 から始まり 1 ビットを占めています。2 番目の `f1` は、オフセット 1 から始まり 12 ビットを占めています。3 番目の `f2` は、オフセット 13 から始まり 3 ビットを占めています。構造体のサイズは 2 バイトです。

次のような場合には、ビット・フィールドの位置合わせの前に埋め込みが行われます。

- サイズ 0 のビット・フィールドでは、次のパック境界まで埋め込みが行われます。パック境界は、`#pragma pack` 指示文、または `-Zpn` コンパイラ・オプションによって決定されます。サイズが 0 のビット・フィールドの場合、ビット・フィールドの基本タイプは無視されます。たとえば、次の構造体について考えてみてください。

```
struct b {  
    char f0: 1;  
    int   : 0;  
    char f1: 2;  
} struct_b;
```

`-Zp1` オプションを指定してソース・ファイルをコンパイルした場合、またはコンパイル時に `#pragma pack 1` 指示文を検出した場合には、`f0` はオフセット 0 から始まって 1 ビットを占有し、名前のないビット・フィールドはオフセット 8 から始まって 0 ビットを占有し、`f1` はオフセット 8 から始まって 2 ビットを占有します。

同様に、`-Zp2` オプションまたは `#pragma pack 2` 指示文が使用されている場合は、名前のないビット・フィールドはオフセット 16 から始まります。`-Zp4` または `#pragma pack 4` が使用されていれば、オフセット 32 から始まります。

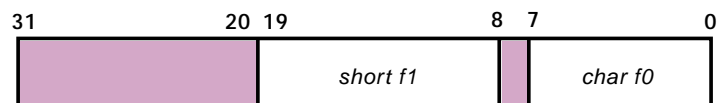
- ビット・フィールドが現在のユニットに入らない場合は、次のパック境界、または次のユニット境界のうち、どちらか近い方まで、埋め込みが行われます。(ユニット境界はビット・フィールドの基本型によって決定されます。たとえば、宣言 `"char foo: 1"` に関連するユニット境界

はバイトです。)現在のユニットは、次の例に示すように、現在のオフセット、ビット・フィールドの基本サイズ、指定されているパックの種類によって決定されます。

```
struct c {  
    char f0: 7;  
    short f1: 11;  
} struct_c;
```

-Zp1 オプションまたは #pragma pack 1 指示文を指定したと仮定すると、この構造体では f0 はビット・オフセット 0 から始まって 7 ビットを占有します。f1 の基本サイズが 8 ビットであり、現在のオフセットが 7 であるため、f1 は現在のユニットには入りません。次のユニット境界または次のパック境界のうち、どちらか近い方まで埋め込みが行われ、この場合はビット 8 になります。この構造体のレイアウトを図 2-4 に示します。

図 2-4: 次のパック境界までの埋め込み



ZK-1081U-AI

2.2.5 __align 記憶クラス修飾子

データ位置合わせはデータ型によって暗黙に定義されます。たとえば、C コンパイラは int (32 ビット) を 4 バイト境界に位置合わせし、long (64 ビット) を 8 バイト境界に位置合わせします。__align 記憶クラス修飾子は、任意の C データ型のオブジェクトを指定された記憶境界に位置合わせします。これは、データ宣言または定義で使用できます。

__align 修飾子は次の形式で使します。

```
__align (keyword)  
__align (n)
```

keyword には定義済み位置合わせ定数を、n には 2 の累乗の整数を指定します。定義済み定数あるいは 2 の累乗は、コンパイラに対してデータの位置合わせのためのバイト数を指定します。

たとえば、整数を次のクォードワード境界に合わせる場合は、次のいずれかの宣言を使用します。

```
int __align( QUADWORD ) data;
int __align( quadword ) data;
int __align( 3 ) data;
```

この例では、int __align (3) は 2x2x2 バイト (8 バイト)、つまりメモリの 1 クォードワードの位置合わせを指定します。

定義済み境界合わせ定数とそれに対応する2 の累乗およびバイト数は次のとおりです。

定数	2 の累乗	バイト数
BYTE あるいは byte	0	1
WORD あるいは word	1	2
LONGWORD あるいは longword	2	4
QUADWORD あるいは quadword	3	8

2.3 C プリプロセッサ

C プリプロセッサは、ソース・ファイルをコンパイルする前に、マクロの展開、ヘッダ・ファイルの取り込み、プリプロセッサ指示文の実行を行います。以降の各項で、C プリプロセッサが実行する Tru64 UNIX に固有の処理について説明します。C プリプロセッサについての詳細は、リファレンス・ページの cc(1) および cpp(1)、および『*Compaq C* 言語リファレンス・マニュアル』を参照してください。

2.3.1 定義済みマクロ

コンパイラが起動されると、入力ファイルの言語とそのコードの実行環境を識別する C プリプロセッサ・マクロが定義されます。プリプロセッサ・マクロの一覧については、cc(1) を参照してください。#ifdef 文でこれらのマクロを参照すると、特定の言語または環境に適用するコードを分離することができます。Tru64 UNIX を一意に識別するには、次の文を使用します。

```
#if defined ( __digital__ ) && defined ( __unix__ )
```

ソース・ファイルのタイプと適用する規格のタイプによって、定義されるマクロが決定されます。C コンパイラはいくつかのレベルの規格をサポートしています。

- `-std` オプションを指定すると、ANSI C 規格を適用しますが、規格で許可されていない共通プログラミング手法のいくつかを許可し、マクロ `__STDC__` を 0 (ゼロ) に設定します。これは省略時の設定です。
- `-std0` オプションを指定すると、Kernighan と Ritchie (K & R) のプログラミング・スタイルを適用しますが、K & R の動作が定義されていないか、または不明瞭であるものについては、特定の ANSI 拡張機能が使用できます。一般に、`-std0` では、ほとんどの ANSI 以前の C プログラムをコンパイルして、期待どおりの結果を得ることができます。この場合、`__STDC__` マクロが定義されません。
- `-std1` オプションを指定すると、ANSI C 規格およびその禁止事項のすべてを厳密に適用します。禁止事項には、`void` の処理に適用されるものや、式における lvalue の定義、整数とポインタの混合、rvalue の変更などがあります。これは `__STDC__` マクロを 1 に定義します。

`cpp` コマンドは、標準の C マクロ、すなわち `__LINE__`、`__FILE__`、`__DATE__`、`__TIME__`、(該当する場合は `__STDC__`) 以外は事前に定義しないので注意してください。

2.3.2 ヘッド・ファイル

ヘッド・ファイルは、通常、次の目的で使用されます。

- システム・ライブラリへのインタフェースを定義する
- 大規模なアプリケーション内で別々にコンパイルされたモジュールに共通な定数、型、関数プロトタイプを定義する

C ヘッド・ファイル (インクルード・ファイルとも呼ばれる) には、`.h` 接尾語が付きます。必要なヘッド・ファイルは、通常、ライブラリ・ルーチンまたはシステム・コールのリファレンス・ページに示されています。ヘッド・ファイルは、さまざまな言語で記述されたプログラムで使用することができます。

注意

`dbx` または `ladebug` を使用してプログラムをデバッグする場合は、ヘッド・ファイルに実行可能コードを入れてはなりません。デバッガがヘッド・ファイルをソース・コードの 1 行として解釈し、デバッグ・セッション時にファイルのソース行が、まったく表示されなくなるためです。dbx デバッガについての詳細

は、第 5 章を参照してください。ladebug についての詳細は、
『Ladebug Debugger Manual』を参照してください。

ヘッダ・ファイルは、次の 2 つの方法のうちのいずれかでプログラム・ソース・ファイルに含めることができます。

```
#include "filename"
```

これは、C マクロ・プリプロセッサが、その指示文を含むファイルを見つけたディレクトリ、`-I` オプションで指定された探索パス、`/usr/include` の順でインクルード・ファイル *filename* を探索することを示します。

```
#include <filename>
```

これは、C マクロ・プリプロセッサが、その指示文を含むファイルを見つけたディレクトリではなく、`-I` オプションで指定された探索パスと `/usr/include` でインクルード・ファイル *filename* の探索を行うことを示します。

また、`-Idir` および `-nocurrent_include` オプションを使用して、C プリプロセッサに `#include` ファイルを探索させる追加のパス名 (ディレクトリ) を指定することもできます。

- `-Idir` については、C プリプロセッサは、最初にその指示文を含むファイルを見つけたディレクトリの中を探索し、続いて、指定されたパス名 (*dir*)、次に省略時のディレクトリ (`/usr/include`) を探索します。*dir* が省略されると、省略時のディレクトリは探索されません。
- 引数のない `-I` については、C プリプロセッサは `/usr/include` を探索しません。
- `-nocurrent_include` については、C プリプロセッサは、`#include` 指示文を含むファイルのあるディレクトリを探索しません。つまり、`#include "filename"` は `#include <filename>` と同様に処理されます。

2.3.3 各国語対応インクルード・ファイルの設定

C, Fortran, およびアセンブリ・コードは、同じインクルード・ファイルに常駐させることができ、必要に応じてプログラムの中に条件付きで挿入することができます。共用可能なインクルード・ファイルを設定するに

は、.h ファイルを作成し、次の例のように、それぞれのコードを入力しなければなりません。

```
#ifdef __LANGUAGE_C__
.
.    (C code)
.
#endif
#ifdef __LANGUAGE_ASSEMBLY__
.
.    (assembly code)
.
#endif
```

コンパイラがこのファイルを C ソース・ファイルに取り込むと、`__LANGUAGE_C__` マクロが定義されて、C コードがコンパイルされます。コンパイラがこのファイルをアセンブリ言語ソース・ファイルに取り込むと、`__LANGUAGE_ASSEMBLY__` マクロが定義されて、アセンブリ言語コードがコンパイルされます。

2.3.4 処理系固有のプリプロセッサ指示文 (#pragma)

#pragma 指示文は、コンパイラごとに異なる機能をインプリメントする標準的な方法です。C コンパイラは、次の処理系固有のプリAGMAをサポートします。

- #pragma assert
- #pragma environment
- #pragma extern_model
- #pragma extern_prefix
- #pragma function
- #pragma inline
- #pragma intrinsic
- #pragma linkage
- #pragma member_alignment
- #pragma message
- #pragma optimize

- `#pragma pack`
- `#pragma pointer_size`
- `#pragma use_linkage`
- `#pragma weak`

これらのプリAGMAについての詳細は、第 3 章で説明します。

2.4 ソース・プログラムのコンパイル

`cc` コマンドで設定されたコンパイル環境は、共通オブジェクト・ファイル・フォーマット (COFF) に準拠しているオブジェクト・ファイルを作成します。

`cc` コマンドでサポートされているオプションは、デバッグ、最適化、プロファイル機能など、さまざまなプログラム開発関数と出力ファイルへ割り当てる名前を選択します。`cc` コマンド行オプションについての詳細は、`cc(1)` を参照してください。

以降の各項で、省略時のコンパイラの動作および各国語対応プログラムのコンパイル方法について説明します。

2.4.1 省略時のコンパイル動作

ほとんどのコンパイラ・オプションには、そのオプションがコマンド行で指定されない場合に使用される省略時の値があります。たとえば、出力ファイルの省略時の名前は、オブジェクト・ファイルに対しては `filename.o` になります。このとき、`filename` は、ソース・ファイルのベース名です。実行可能プログラムのオブジェクトの省略時の名前は、`a.out` になります。次の例では、`prog1.c` および `prog2.c` のソース・ファイルをコンパイルする際に、省略時の設定が使用されます。

```
% cc prog1.c prog2.c
```

このコマンドによって C コンパイラが実行され、`prog1.o` と `prog2.o` というオブジェクト・ファイルと、`a.out` という実行可能プログラムが作成されます。

`cc` コンパイラ・コマンドに他のオプションを指定しないで入力した場合には、次のオプションが有効になります。

`-noansi_alias`

ANSI C 別名化規則をオフにします。これは、最適化プログラムが最適化においてアグレッシブにならないようにします。

`-arch generic`

すべての Alpha プロセッサに適切な命令を生成します。

`-assume aligned_objects`

コンパイラがそのような仮定をして、位置合わせされたポインタ型のポインタ逆参照のためにより効率的なコードを生成できるようにします。

`-assume math_errno`

コンパイラは、`errno` を設定できる算術ライブラリ・ルーチン呼び出した後、プログラムが `errno` を問い合わせる可能性があることを想定できるようにします。

`-call_shared`

実行時に共用可能オブジェクトを使用する動的な実行可能ファイルを作成します。

`-nocheck_bounds`

配列境界の実行時検査を無効にします。

`-cpp`

コンパイル前に C およびアセンブリ・ソース・ファイルで、C マクロ・プリプロセッサが呼び出されるようにします。

`-error_limit 30`

指定のコンパイルに対してコンパイラが出力するエラー・レベルの診断数を 30 までに制限します。

`-float`

型表現を `float` から `double` に昇格する必要がないことをコンパイラに通知します。

`-nofp_reorder`

精度に影響するような浮動小数点演算の順序変更をしないようにコンパイラに指示します。

-fprm n

浮動少数点数の通常の丸め (不偏の丸めから近似値へ) を行います。

-fptm n

浮動小数点アンダフローまたは不正確なトラッピング・モードを生成しない命令を生成します。

-g0

シンボリック・デバッグのためのシンボル情報を生成しません。

-I/usr/include

ディレクトリ /usr/include において、ファイル名がスラッシュ (/) で始まらない #include ファイルが最初に探索されることを指定します。

-inline manual

#pragma inline 指示文によって明示的にインライン化が要求されている関数呼び出しのみをインライン化します。

-intrinsics

特定の関数を intrinsics として認識して、適切な最適化を行うようにコンパイラに指示します。

-member_alignment

コンパイラに指示して、データ構造体のメンバを (ビット・フィールドのメンバを除き) 自然境界に位置合わせするようにします。

-nomisalign

任意に位置合わせされたアドレスの位置合わせフォールトを生成します。

-nestlevel=50

インクルード・ファイルのネスト・レベル制限を 50 に設定します。

-newc

省略時のオプション設定をすべてリストしてコンパイラを起動します。このオプションは、-migrate をオフにするためだけに用意されています。

-O1

グローバルな最適化を可能にします。

-p0

プロファイリングを無効に設定します。

-npg

gprof プロファイリングをオフにします。

-preempt_module

モジュールごとにシンボルの優先使用を可能にします。

-SD/usr/include

パス名が /usr/include で始まるヘッダ・ファイル内の移植性のない構造に対するメッセージを抑制します。

-signed

char 型の表現を signed char 型と同じにします。

-std

ANSI C 規格を拡張しますが、規格によって禁止されている一般的なプログラミング手法を許可します。

-tune generic

Alpha アーキテクチャのすべての処理系に適切な命令のチューニングを選択します。

-writable_strings

文字列リテラルを書き込み可能にします。

表 2-3 に、cc(1) のオプション・カテゴリごとの省略時オプションを示します。

表 2-3: cc コマンドのオプション・カテゴリごとの省略時オプション

オプション・カテゴリ	省略時オプション
コンパイラ選択	-newc
言語モード	-std
コンパイラ全体の動作	-arch generic, -error_limit 30, -nestlevel=50
コンパイラの診断制御	-SD/usr/include
C プリプロセッサ	-cpp, -I/usr/include
リンカまたはローダ	-call_shared
最適化	-noansi_alias, -assume math_er- rno, -float, -nofp_reorder, -inline manual, -intrinsics, -O1, -preempt_symbol, -tune generic
フィードバック主導の最適化	なし
ソース・コードのデバッグ	-g0
プログラムのプロファイリング	-p0, -npg
データの整列	-assume aligned_objects, -member_alignment, -nomisalign
データの揮発性	-weak_volatile
C 言語	-signed, -writable_strings
スタックの処理とポインタの処理	なし
IEEE の浮動小数点サポート	-fprm n, -fptm n
コンパイラの開発	なし

次に示すのは、cc コンパイラのその他の省略時の動作です。

- 省略時の動作では、コンパイル (あるいはアセンブル) が成功した場合はソース・ファイルは自動的にリンクされます。
- -o オプションで出力ファイル名を指定していない場合は、a.out が使用されます。
- 浮動小数点の計算は、IEEE 浮動小数点でなくファスト浮動小数点で行われます。
- ポインタは 64 ビットです。32 ビット・ポインタの使用方法については、付録 A を参照してください。

- 一時ファイルは /tmp ディレクトリか、または \$TMPDIR 環境変数で指定したディレクトリに作成されます。

2.4.2 多言語プログラムのコンパイル

メイン・プログラムのソース言語がサブプログラムのソース言語と異なっている場合は、適切なドライバで各プログラムを個別にコンパイルし、別のステップでリンクしてください。-c オプションを指定すると、リンクに適したオブジェクトを作成することができます。-c オプションは、オブジェクト・ファイルが作成された直後にドライバを停止します。たとえば、次のように入力してください。

```
% cc -c main.c
```

このコマンドは、実行可能ファイル a.out ではなく、オブジェクト・ファイル main.o を作成します。

C 以外の言語で作成したソース・ファイルのオブジェクト・モジュールを作成したのち、cc コマンドを使用して C ソース・ファイルをコンパイルし、すべてのオブジェクト・モジュールをリンクして1つの実行可能ファイルにすることができます。たとえば、次の cc コマンドは c-prog.c をコンパイルし、c-prog.o および nonc-prog.o をリンクして a.out という実行可能ファイルを作成します。

```
% cc nonc-prog.o c-prog.c
```

2.4.3 配列境界の実行時検査の有効化

cc コマンドで -check_bounds オプションを指定すると、実行時コードを生成して、配列境界検査が行われます。-nocheck_bounds オプション (省略時の設定) は、配列境界の実行時検査を無効にします。

コンパイラに対して実行時検査を指示するコード、および指定された検査で使用される正確な境界値は、コンパイラのインプリメンテーション上の特性に影響されるため、ユーザには明白ではありません。これらを左右する厳密な条件は、次のとおりです。以下の説明は、配列を含めて C の言語規則を十分に理解していることを前提としています。

- 検査は、宣言された配列オブジェクト名の使用時に行われます。ポインタ値の使用時には、たとえポインタが添字演算子を使って逆参照されるとしても、検査は行われません。これは、たとえば、1次元の配列とし

て宣言された仮パラメータは、C 言語ではポインタと見なされるため、検査が行われないことを意味します。ただし、仮パラメータが多次元配列の場合、最初の添字はアクセス対象の配列オブジェクトを決定するポインタ操作を表し、そこでは境界は検査されません。2 番目およびそれに続く添字で、境界検査が生成されます。

- 添字演算子 (左または右のオペランド) を使って配列へのアクセスが行われ、かつ添字演算子が演算子のアドレスのオペランドではない場合、インデックスがゼロと配列内の要素数から 1 を引いた数との間の範囲内にあるかどうか検査されます。
- 添字演算子 (左または右のオペランド) を使って配列へのアクセスが行われ、かつ添字演算子が演算子のアドレスのオペランドである場合、インデックスがゼロから配列内の要素数までの範囲内にあるかどうか検査されます。C 言語では、特に、配列の終端を 1 要素超えたアドレスを計算に使用できなければなりません。それは、次のループ終了テストのような一般的なプログラミング手法を可能にするためです。

```
int a[10];
int *b;
for (b = a ; b < &a[10] ; b++) { .... }
```

この場合、`a[10]` が配列の境界外にあっても、`&a[10]` の使用が可能になります。

- 配列へのアクセスがポインタ加算を使って行われる場合、追加された値がゼロから配列内の要素数までの範囲内にあるかどうか検査されます。配列名に整数を追加することには、配列名を最初の要素へのポインタに変換すること、および要素のサイズに応じて決定された整数値を追加することが含まれます。コンパイラ内での境界検査の実装は、配列名からポインタへの変換をトリガとして実行されますが、境界検査の導入時には、生成されるポインタ値が逆参照されるかどうかは不明です。このため、この場合も前述の場合と同様に処理されます。つまり、アドレスの計算のみが検査され、配列の終端を超えた 1 要素のアドレスも計算できます。
- 配列へのアクセスがポインタ減算を使って行われる (つまり、あるポインタから別のポインタを減算することではなく、ポインタから整数値を減算する) 場合、減算される値が配列の要素数を負にした値からゼロまでの範囲内にあるかどうか検査されます。

後者の3つの場合には、オプションのコンパイル時のメッセージ (ident SUBSCRBOUNDS2) から、配列へのアクセスが定数の添字または定数ポインタ算術のいずれかを使って行われたこと、アクセスされた要素が配列の終端を1要素だけ超えた範囲外にあることを正確に検出できます。

- 1つの要素を使って宣言された配列の検査は行われません。ANSI Cでは、structの宣言内に複数の次元のない配列は許可されないため、動的にサイズ変更が行われる配列を、複数メンバに割り当てられた要素数を保持するstructとしてインプリメントし、最後のメンバを1つの要素で宣言された配列とすることが一般的な手法であるためです。最後の配列メンバへのアクセスは、実行時に割り当てられたサイズにより制限されるため、1と宣言された境界に対する検査は役に立ちません。

この場合、オプションのコンパイル時メッセージ (ident SUBSCRBOUNDS1) を有効に設定することにより、単一要素で宣言された配列へのアクセスが定数の添字または定数ポインタ算術のいずれかを使って行われたことを検出できること、およびアクセスされた要素は配列の一部ではないことに注意してください。

- コンパイラは、定数で索引付けされた配列の実行時検査を発行します (コンパイラがコンパイル時にここで論じている事例を検出でき、実際に検出した場合でも)。コンパイラがアクセスが有効であると判断すると、例外として、実行時検査が行われません。
- 多次元配列へのアクセスが行われると、コンパイラは各添字式を検査して、それぞれが対応する境界内にあることを確認します。次のコードの場合、コンパイラは x と y の両方が0から9までの範囲内にあるかどうかを検査します ($10 * x + y$ が0から99までの範囲内にあるかどうかは検査しません)。

```
int a[10][10];
int x,y,z;
x = a[x][y];
```

次の例は、これまで示した規則を例証するものです。

```
int a[10];
int *b;
int c;
int *d;
int one[1];
int vla[c];           // C9X variable-length array

a[c] = 1;              // check c is 0-9, array subscript
c[a] = 1;              // check c is 0-9, array subscript
b[c] = 1;              // no check, b is a pointer
d = a + c;             // check c is 0-10, computing address
```

```

d = b + c;          // no check, b is a pointer
b = &a[c]           // check c is 0-10, computing address
*(a + c) = 1;       // check c is 0-10, computing address
*(a - c) = 1;       // check c is -10 to 0, computing address

a[1] = 1;           // no run-time check - know access is valid
vla[1] = 1;         // run-time check, vla has run-time bounds
a[10] = 1;          // run-time check (and compiler diagnostic)
d = a + 10;         // no run-time check, computing address
                  // SUBSCRBOUNDS2 message can be enabled

c = one[5];         // no run-time check, array of one element
                  // SUBSCRBOUNDS1 message can be enabled

```

境界外のアクセスに遭遇すると、出力は次のようになります。

```
Trace/BPT trap (core dumped)
```

プログラムは、次のコードを使ってこのエラーをトラップできます。

```
signal(SIGTRAP, handler);
```

実行時検査が有効な場合は、ポインタ算術を使って配列への正当なアクセスが行われた場合に、間違った検査が行われることがあります。

コンパイラは、配列名からポインタへの変換により得られたポインタへの最初の算術操作に対する検査コードだけを出力できます。このため、得られたポインタ値に対して再びポインタ算術による操作が行われると、検査が不正確になる可能性があります。 $a = b + c - d$ という式があり、 a がポインタ、 b が配列、 c と d が整数の場合を考えてみます。境界検査が有効な場合、 c が配列の境界内にあるかどうかを検査されます。このとき、 c が配列の境界外にあるが、 $c - d$ は境界内にある場合、間違った実行時トラップとなります。

この場合には、ポインタ式をコーディングし直して、整数部分をカッコ内に含めることができます。これで、式には1つのポインタ算術演算子だけが含まれることになり、正確な検査が行われます。前の例では、式は次のように変更されます。

```
a = b + (c - d);
```

2.5 オブジェクト・ファイルのリンク

cc ドライバ・コマンドは、オブジェクト・ファイルをリンクして、実行プログラムを作成します。場合によっては、ld リンカを直接使用することもあります。アプリケーションの特性に応じて、コンパイルしてから別にリンクを行うか、またはコンパイラ・コマンドを使用して、コンパイルとリン

クを一度に実行するかを決定してください。考慮すべき点は数多くありますが、特に次のことを考慮にいらしてください。

- すべてのソース・ファイルが同一の言語であるかどうか
- ソース形式のファイルがあるかどうか

2.5.1 コンパイラ・コマンドによるリンク

リンカ・コマンドの代わりにコンパイラ・コマンドを使用して、異なるオブジェクトを1つの実行可能プログラムにリンクすることができます。アセンブラを除くコンパイラは、`.o` 接尾語を認識するとすぐにリンクを実行します。この接尾語は、そのファイルにリンクが可能なオブジェクト・コードが含まれていることを示します。

コンパイラのドライバ・プログラムは言語に対応するライブラリを自動的にリンクに渡すため、通常はコンパイラ・コマンドを使用してください。たとえば、`cc` ドライバでは、省略時の設定としてCライブラリ(`libc.so`)を使用します。各コンパイラ・コマンドが使用する省略時のライブラリについての詳細は、`cc(1)` などのリファレンス・ページの該当するコマンドを参照してください。

また、`cc` コマンドの `-l` オプションを使用して追加のライブラリを指定し、未解決の参照を探索することもできます。次の例は、`cc` ドライバを使用して、`-l` オプションで2つのライブラリ名をリンクに渡す方法を示しています。

```
% cc -o all main.o more.o rest.o -lm -lexc
```

`-lm` オプションは算術ライブラリを指定し、`-lexc` オプションは例外ライブラリを指定します。

プログラムを最適化する場合には、1つのコマンドを使用してモジュールのコンパイルとリンクを行ってください。ほとんどのコンパイラには、特定のオプションを使用して、最適化のレベルを上げる機能があります。たとえば、次のようなオプションを使用できます。

- `-O0` オプションは、最適化の要求は行いません。これは通常、デバッグのために使用されます。
- `-O1` オプションは、ローカルなモジュール固有の最適化を要求します。
- モジュール間にまたがる最適化は、`-ifo` オプションを用いて要求しなければなりません。さらに `-O3` オプションを指定すると、モジュール間

にまたがる最適化が改善されます。この場合には、1つの操作で複数のファイルのコンパイルを行うことによって、コンパイラは最大限の最適化を実行することができます。-ifo オプションを指定すると、複数のソース・ファイルに対して1つの .o ファイルが生成されます。

2.5.2 ld コマンドによるリンク

通常、ユーザがリンクを直接実行するのではなく、cc コマンドを使用して間接的に実行します。アセンブラ・オブジェクトから単独で作成する必要のある実行可能プログラムは ld コマンドで作成しなければなりません。

リンカ (ld) は、コマンド行に指定された順番で1つまたは複数のオブジェクト・ファイルを結合して、1つのプログラム・オブジェクト・ファイルを作成します。リンカによって、再配置、外部シンボルの解決、および実行可能ファイルの作成に必要な、他のすべての処理が実行されます。-o オプションで指定しない限り、実行可能プログラム・ファイルは、a.out という名前になります。このプログラム・ファイルを実行することも、別のリンカ・オペレーションへの入力として使用することもできます。

as アセンブラはリンクを自動的に実行しないため、アセンブリ言語で記述されたプログラムをリンクするには、次のいずれかの処理を行います。

- 他のコンパイラ・コマンドのいずれかを使用して、アセンブルおよびリンクを行う。

アセンブリ言語ソース・ファイルの接尾語 .s によって、コンパイラ・コマンドは自動的にアセンブラを起動します。

- as を使用してアセンブルを実行し、次に ld コマンドを使用して、アセンブルが実行されたオブジェクト・ファイルのリンクを行う。

リンクのプロセスに影響を与えるオプションやライブラリについての詳細は、リファレンス・ページの ld(1) を参照してください。

2.5.3 ライブラリの指定

Tru64 UNIX システム上でコンパイルされたプログラムは、自動的に libc.so という C ライブラリとリンクされます。libc.so がないルーチンまたはコンパイラ・コマンドに対応するアーカイブ・ライブラリのうちのいずれかを使用する場合は、プログラムをライブラリと明示的にリンクしなければなりません。ライブラリと明示的にリンクしない場合には、プログラムは正常にリンクされません。

次のような状況では、ライブラリを明示的に指定する必要があります。

- 多言語プログラムのコンパイル

多言語プログラムをコンパイルする場合は、必要な実行時ライブラリがあれば、必ず明示的に要求して未解決の参照を処理してください。
`-lstring` を指定することによって、ライブラリがリンクされます。
`string` はライブラリ名の省略形です。

たとえば、C のメイン・プログラムおよび別の言語のプロシージャを記述する場合は、その言語と算術ライブラリに対して、ライブラリを明示的に指定しなければなりません。これらのオプションを使用すると、リンクは `-l` を `lib` に置き換えて、言語ライブラリおよび算術ライブラリのための指定された文字と、スタティック・ライブラリ (非共有アーカイブ・ライブラリ) かダイナミック・ライブラリ (呼び出し共有オブジェクトまたはシェアード・ライブラリ) により `.a` 接尾語あるいは `.so` 接尾語を追加します。続いて、結果として生成されたライブラリ名を次のディレクトリの中で探索します。

```
/usr/shlib  
/usr/ccs/lib  
/usr/lib/cmplrs/cc  
/usr/lib  
/usr/local/lib  
/var/shlib
```

各言語で使用されるライブラリの一覧については、さまざまな言語のコンパイラ・ドライバのリファレンス・ページを参照してください。

- アーカイブ・ライブラリへのオブジェクト・ファイルの保存

コンパイラまたはリンクのコマンド行にライブラリのパス名を指定しなければなりません。たとえば、次のコマンドでは、`/usr/jones` ディレクトリ内の `libfft.a` というアーカイブ・ライブラリが算術ライブラリとリンクされるように指定されます。

```
% cc main.o more.o rest.o /usr/jones/libfft.a -lm
```

リンクは、指定された順序でライブラリを探索します。したがって、アーカイブ・ライブラリのいずれかのファイルで、算術ライブラリからのデータまたはプロシージャが使用される場合は、算術ライブラリを指定する前に、アーカイブ・ライブラリを指定しなければなりません。

2.5.4 リンカ出力ファイルでのリンク・エラー問題の回避

さまざまな種類のリンク・エラーに対して、リンカはエラーを無視して出力ファイルを作成し、同名の同じ出力ファイルを上書きします。ほとんどのアプリケーションでは、この動作による問題は生じません。問題が生じる場合は、既存の出力ファイルが上書きされないようにすることができます。この場合は、リンカに対して一時ファイルの出力を指定し、次にリンクがエラーなしで完了した時点で一時ファイルの名前を目的の出力ファイルに変更します。call_shared 実行可能ファイルでの実行例を次に示します。

```
cc -o main.exe.X main.o
mv main.exe.X main.exe
```

次の例に示すように、シェアード・ライブラリを構築する際にも同じ手法を使用できます。この場合は、-soname オプションを使用して、ライブラリの内部名を正しく設定する必要があります。

```
ld -shared -soname libfoo.so -o libfoo.so.X foo.o -lc
mv libfoo.so.X libfoo.so
```

2.6 プログラムの実行

現在の作業ディレクトリで実行可能プログラムを実行するには、ほとんどの場合、そのファイル名を入力します。たとえば、現在のディレクトリにあるプログラム a.out を実行するには、次のように入力します。

```
% a.out
```

実行可能プログラムがパスのディレクトリ内に存在しない場合は、次のように、ファイル名の前にディレクトリ・パスを入力します。

```
% ./a.out
```

プログラムが起動されるとき、C プログラムの main 関数が、次のオプションのパラメータのいずれかを指定して定義されている場合には、main 関数はコマンド行から引数を受け取ることができます。

```
int main ( int argc, char *argv[], char *envp[] ) [...]
```

argc パラメータは、プログラムを起動したコマンド行に指定されている引数の数です。argv パラメータは、引数を含む文字列の配列です。envp パラメータは、ユーザ名や制御ターミナルなどのプロセス情報を含む環境配列です。(envp パラメータはコマンド行引数の引き渡しには何の関係もありません。主に、exec および getenv 関数の呼び出し中に使用されます。)

定義したパラメータにのみアクセスすることができます。たとえば、次のプログラムでは、*argc* および *argv* パラメータを定義して、プログラムに渡されたパラメータの値をエコーします。

```
/*
 * Filename: echo-args.c
 * This program echoes command-line arguments.
 */

#include <stdio.h>

int main( int argc, char *argv[] )
{
    int i;

    printf( "program: %s\n", argv[0] ); /* argv[0] is program name */

    for ( i=1; i < argc; i++ )
        printf( "argument %d: %s\n", i, argv[i] );

    return(0);
}
```

このプログラムを次のコマンドでコンパイルすると、*a.out* というプログラム・ファイルが作成されます。

```
$ cc echo-args.c
```

ユーザが *a.out* を起動して、コマンド行引数を引き渡すと、プログラムによりそれらの引数がターミナル上にエコーされます。たとえば、次のように入力します。

```
$ a.out Long Day\'s "Journey into Night"
    program: a.out
    argument 1: Long
    argument 2: Day's
    argument 3: Journey into Night
```

引数を *a.out* に引き渡す前に、シェルによってすべての引数が解析されます。このため、単一引用符の前にはバックスラッシュを指定し、アルファベットの引数は空白またはタブで区切り、引数に空白やタブを含める場合は引用符で囲む必要があります。

2.7 オブジェクト・ファイルのツール

ソース・ファイルをコンパイルした後、次に示すツールを使用すると、オブジェクト・ファイルや実行可能ファイルをチェックすることができます。

- `odump` — シンボル・テーブルとヘッダ情報を含む，オブジェクト・ファイルの内容を表示します。
- `stdump` — オブジェクト・ファイルのシンボル・テーブル情報を表示します。
- `nm` — シンボル・テーブル情報のみを表示します。
- `file` — 使用されているプログラミング言語など，指定されたファイルの一般的なプロパティに関する記述的な情報を表示します。
- `size` — テキスト，データおよび bss セグメントのサイズを表示します。
- `dis` — オブジェクト・ファイルを機械語命令に逆アセンブルを行います。

次の項では，上記のツールについて詳しく説明します。また，オブジェクト・ファイルやバイナリ・ファイル中でプリント可能な文字列を探索する `string` コマンドの使用についての情報は，`strings(1)` を参照してください。

2.7.1 ファイル内の選択した部分のダンプ (`odump`)

`odump` ツールは，ヘッダ・テーブルおよび他のオブジェクト・ファイルやアーカイブ・ファイル内の選択した部分を表示します。たとえば，ファイル `echo-args.o` に対して `odump` を使用すると，次のような情報が表示されます。

```
% odump -at echo-args.o
```

```
***ARCHIVE SYMBOL TABLE***
```

```
***ARCHIVE HEADER***
```

Member Name	Date	Uid	Gid	Mode	Size
-------------	------	-----	-----	------	------

```
***SYMBOL TABLE INFORMATION***
```

```
[Index] Name Value Sclass Symtype Ref
```

```
echo-args.o:
```

[0]	main	0x0000000000000000	0x01	0x06	0xffffffff
[1]	printf	0x0000000000000000	0x06	0x06	0xffffffff
[2]	_fpdata	0x0000000000000000	0x06	0x01	0xffffffff

詳細は、`odump(1)` を参照してください。

2.7.2 シンボル・テーブル情報の表示 (nm)

`nm` ツールは、オブジェクト・ファイルのシンボル・テーブル情報を表示します。たとえば、`nm` では、実行可能ファイル `a.out` を作成するためのオブジェクト・ファイルについて、次のような情報が表示されます。

```
% nm
nm: Warning: - using a.out

Name                               Value             Type      Size
.bss                               | 0000005368709568 | B | 0000000000000000
.data                              | 0000005368709120 | D | 0000000000000000
.lit4                              | 0000005368709296 | G | 0000000000000000
.lit8                              | 0000005368709296 | G | 0000000000000000
.rconst                           | 0000004831842144 | Q | 0000000000000000
.rdata                             | 0000005368709184 | R | 0000000000000000
:
:
```

Name 欄にはシンボルまたは外部名、Value 欄にはシンボルのアドレスまたはデバッグ情報、Type 欄にはシンボルのタイプを示す英文字、Size 欄にはシンボルのサイズ (ソース・ファイルがデバッグ・オプション `-g` などでコンパイルされている場合のみ正確) がそれぞれ表示されています。シンボル・タイプの英文字は、次のことを示しています。

- B — External zeroed data
- D — External initialized data
- G — External small initialized data
- Q — Read-only constants
- R — External read-only data

詳細は、`nm(1)` を参照してください。

2.7.3 ファイル・タイプの決定 (file)

`file` コマンドは、入力ファイルを読み取って各ファイルを検査し、タイプによって分類します。ファイル・タイプは標準出力に書き込まれます。`file` コマンドは、`/etc/magic` ファイルを使用して、マジック・ナンバを含むファイルを識別します。マジック・ナンバは、ファイル・タイプを示す数字または文字列定数です。

次の例は、C ソース・ファイル、オブジェクト・ファイル、および実行可能ファイルを含むディレクトリに対して `file` を使用した場合の出力を示しています。

```
% file *.*
.:          directory
...:       directory
a.out:      COFF format alpha dynamically linked, demand paged executable
or object module not stripped - version 3.11-8
echo-args.c:  c program text
echo-args.o:  COFF format alpha executable or object module not
stripped - version 3.12-6
```

詳細は、`file(1)` を参照してください。

2.7.4 ファイルのセグメント・サイズの設定 (size)

`size` ツールは、指定されたオブジェクト・ファイルまたはアーカイブ・ファイルの `text`、`data`、および `bss` セグメントに関する情報を、8 進数、16 進数、または 10 進数で表示します。たとえば、引数を指定しないで呼び出された場合、`size` コマンドは `a.out` に関する情報を返します。次のように、コマンド行に、オブジェクト・ファイルまたは実行可能ファイルの名前を指定することもできます。

```
% size
text    data    bss    dec    hex
8192    8192    0      16384   4000
% size echo-args.o
text    data    bss    dec    hex
176     96     0      272    110
```

詳細は、`size(1)` を参照してください。

2.7.5 オブジェクト・ファイルの逆アセンブル (dis)

`dis` ツールは、オブジェクト・ファイルのモジュールを機械語に逆アセンブルします。たとえば、`a.out` プログラムの逆アセンブルを行った場合、`dis` コマンドは、次のような出力を作成します。

```
% dis a.out
:
:
      __start:
0x120001080: 23defff0      lda      sp, -16(sp)
0x120001084: b7fe0008      stq      zero, 8(sp)
0x120001088: c0200000      br       t0, 0x12000108c
0x12000108c: a21e0010      ld1      a0, 16(sp)
0x120001090: 223e0018      lda      a1, 24(sp)
```

⋮

詳細については、`dis(1)` を参照してください。

2.8 標準 C ライブラリにおける ANSI 名前空間汚染のクリーンアップ

ANSI C 規格では、`libc` にリンクされるユーザのプログラムは、プログラム内における一定範囲のグローバル識別子を、`libc` 内のグローバル識別子と競合したり、強制排除の問題を起こさないで使用できることが保証されています。

ANSI C 規格はグローバル識別子のある範囲を予約して、`libc` が内部インプリメンテーションで使用できるようにしています。これらは予約識別子と呼ばれ、ANSI の X3.159-1989 に定義されているように、次のものがあります。

- 下線で始まる外部識別子
- 下線で始まり、その後大文字または下線が続く外部識別子

ANSI 準拠のプログラムでは、ANSI ルーチン名と同じか、または前述の予約されている名前空間にあてはまるグローバル識別子は定義できません。その他のグローバル識別子名はすべて、ユーザ・プログラムで使用できます。

歴史的な `libc` のインプリメンテーションには、多数の ANSI でなく、予約されていない識別子が含まれています。これらはどちらも文書化されサポートも行われています。これらのルーチンは、ANSI であっても ANSI でなくても、`libc` 内から他の `libc` ルーチンによって呼び出されることがよくあります。ユーザのプログラムで、このような ANSI でなく、予約されていない項目を独自に定義している場合には、`libc` にある同名のルーチンを強制排除します。このために、ユーザのプログラムが ANSI 準拠であっても、ANSI のものも ANSI でないものも、サポートされている `libc` ルーチンの動作が変わることがあります。このような潜在的な競合は、ANSI 名前空間汚染と呼ばれます。

Tru64 UNIX の `libc` のインプリメンテーションには、文書化されサポートされている ANSI でなく、予約されていない多数のグローバル識別子が含まれています。`libc` 内のこれらグローバル識別子を強制排除から保護し、ユーザの名前空間汚染を避けるため、これらの識別子の大多数の名前を、識別子名の前に 2 つの下線 (`_`) の付く予約されている名前空間に変更していま

す。これらの項目への外部アクセスを維持するために、弱い識別子も、変更前の元の識別子名を使用して追加されています。弱い識別子は、ファイル間のシンボリック・リンクのような働きをします。弱い識別子が参照されると、代わりに強い識別子が使用されます。

`libc` と静的にリンクするユーザ・プログラムは、弱い識別子用の余分のシンボル・テーブル・エントリを持つことがあります。これらの識別子は、それぞれ対応する予約識別子と同じアドレスを持ち、このアドレスもテーブルに含まれています。たとえば、静的にリンクされたプログラムが `libc` から `tzset()` 関数を単に呼び出した場合、シンボル・テーブルにはこの呼び出しに対して、次の 2 つのエントリが含まれます。

```
# stdump -b a.out | grep tzset
18. (file 9) (4831850384) tzset Proc Text symref 23 (weakext)
39. (file 9) (4831850384) __tzset Proc Text symref 23
```

この例では、`tzset` が弱い識別子であり、`__tzset` が強い識別子です。識別子 `__tzset` が、実際に作業を行うルーチンです。

共用としてリンクされたユーザ・プログラムでは、シンボル・テーブルへのこのような追加は見られません。これは、弱い識別子と強い識別子のペアがシェアド・ライブラリに残っているためです。

`libc` から ANSI でなく、予約されていない識別子を参照している既存のユーザ・プログラムは、1 つの例外を除いて、この変更のためにリコンパイルする必要はありません。つまり、`libc` におけるこれらの識別子の強制排除に依存していたプログラムは、予約されていない名前を使用することによってそれらを強制排除することはできなくなります。この種の強制排除は ANSI 準拠ではないため、使用しないでください。ただし、これらの識別子を強制排除する機能は、新しく予約された `__` 名を使用すると、まだ利用できます。

この変更は、動的および静的に `libc` を使用する場合にあてはまります。

- `/usr/shlib/libc.so`
- `/usr/lib/libc.a`

デバッグ・プログラムを `libc` にリンクすると、弱いシンボルへの参照は、次のように強いシンボルへの参照として解決されます。

```
% dbx a.out
dbx version 3.11.4

Type 'help' for help.
```



```
main:    4    tzset

(dbx) stop in tzset

[2] stop in __tzset

(dbx)
```

弱いシンボル `tzset` が参照されている場合、デバッガは代わりに強いシンボル `__tzset` で応答します。これは、強いシンボルが実際に作業を行っているためです。dbx デバッガは、`__tzset` が直接参照された場合と同様に動作します。

2.9 インライン・アセンブリ・コード — ASM

コンパイラは、一般に ASM と呼ばれるインライン・アセンブリ・コードをサポートしています。

組み込み関数と同じように、ASM は関数呼び出しの構文で実装されています。ただし組み込み関数と異なり、ASM を使用するためには、3 種類の ASM のプロトタイプとプリプロセッサ指示文 `#pragma intrinsic` を含むヘッダ・ファイル `<c_asm.h>` をインクルードしなければなりません。

これらの関数のフォーマットは、次のとおりです。

```
__int64 asm(const char *, ...); /* for integer operations, like MULQ */
float fasm(const char *, ...); /* for single precision float instructions, like MULS */
double dasm(const char *, ...); /* for double precision float instructions, like MULT */
#pragma intrinsic (asm, fasm, dasm)
```

`const char*`

`asm`, `fasm`, `dasm` 関数への最初の引数には、インラインで生成される命令と、引数の解釈を記述するメタ言語が含まれています。

...

生成される命令に対するソースおよびデスティネーションの引数 (あれば) と、生成された命令で使われるその他の値。

これらの値は、生成される命令からは、呼び出し基準の引数渡しについての通常の規約に従って利用可能になります (最初の整数引数はレジスタ R16 で使用できます)。

ASM を使用する際は、ヘッダ・ファイル <c_asm.h> 内の #pragma intrinsic 指示文は必須です。これは、次のようなことをコンパイラに知らせます。

- これらの関数はユーザ定義関数ではない。
- コンパイル時に、最初の引数を分析して文字列の内容で指定されたマシンコード命令を生成する、特別な ASM 処理を適用しなければならない。

引数の参照に関するメタ言語の形式は、次のとおりです。

```
<metalanguage_sequence> : <register_alias>
                          | <register_number>
                          | <register_macro>
                          ;

<register_number>       : "$" number
                          ;

<register_macro>       : "%" <macro_sequence>
                          ;

<macro_sequence>      : number
                          | <register_name>
                          | "f" number | "F" number
                          | "r" number | "R" number
                          ;

<register_name> : /* argument registers: R16-R21 */
                 "a0" | "a1" | "a2" | "a3" | "a4" | "a5"

                 /* return value: R0 or F0, depending on type */
                 | "v0"

                 /* scratch registers: R1, R22-R24, R28 */
                 | "t0" | "t1" | "t2" | "t3" | "t4"

                 /* save registers: R2-R15 */
                 | "s0" | "s1" | "s2" | "s3" | "s4" | "s5" | "s6" | "s7"
                 | "s8" | "s7" | "s8" | "s9" | "s10" | "s11" | "s12" | "s13"

                 /* stack pointer: R30 */
                 | "sp" | "SP" | "$sp" | "$SP"

                 | "RA" | "ra"           /* return addr:      R26 */
                 | "PV" | "pv"           /* procedure value:   R27 */
                 | "AI" | "ai"           /* arg info:          R25 */
                 | "FP" | "fp"           /* frame pointer:     R29 */
                 | "RZ" | "rz" | "zero" /* sink/source: R31 == zero */
```

構文上、メタ言語は命令シーケンス内のどの位置にも配置できます。

命令、オペランド、メタ言語を含むリテラル文字列は、次の一般形式に従っていなければなりません。

```
<string_contents>      : <instruction_seq>
                          | <string_contents> ";" <instruction_seq>
                          | error
                          | <string_contents> error
```

```

;
<instruction_seq>      : instruction_operand
                        | directive
;

```

通常，`instruction_operand` は，アセンブリ言語命令と見なされます。これは，コンマで区切られたオペランドのシーケンスとはホワイト・スペースで区切られます。

C 言語では，隣接した文字列リテラルを連結して単一の文字列にするため，連続した命令は，個々の命令がセミコロンで終わっていれば (例に示すように)，1 行に 1 つずつ文字列として記述できます (通常のアセンブリ言語の記述方法と同じ)。

ASM には意味および構文上の規則があります。

- ASM 呼び出しへの最初の引数は，メタ言語の形式で指定し，アセンブルされる命令であると解釈されます。これは，コンパイル時にコンパイラが完全に理解できる内容でなければなりません。したがって，必ずリテラル文字列 (またはリテラル文字列に展開されるマクロ) でなければならず，実行時に値が決まる文字列値を指定することはできません。したがって，間接参照，テーブル参照，構造体の参照などは使用できません。
- その他の引数は，通常の変数引数と同じように引数レジスタにロードされますが，ASM 呼び出しへの 2 番目の引数が，呼び出し標準での最初の引数として処理される点が異なります。

たとえば，次のテストでは 6 個の引数が引数レジスタの `a0` ~ `a5` にロードされ，各部分式の結果はリターン値レジスタの `v0` に格納されます。`v0` は呼び出し標準のリターン値レジスタ (整数関数では `R0`) なので，最後の `MULQ` の結果は「呼び出し」から返される値になります。

```

if (asm("mulq %a0, %a1, %v0;"
        "mulq %a2, %v0, %v0;"
        "mulq %a3, %v0, %v0;"
        "mulq %a4, %v0, %v0;"
        "mulq %a5, %v0, %v0;", 1, 2, 3, 4, 5, 6) != 720){
    error_cnt++;
    printf ("Test failed\n");
}

```

次の例は正しく動作しません。浮動小数点のリターン値レジスタには値がロードされません。また，引数は `r2` ではなく引数レジスタにロードされるため，コンパイル時に，`r2` が設定される前に使用されているという警告が表示されます。

```
z = fasm("mulq %r2, %a1 %r5", x=10, y=5);
```

正しく実行させるには、`r2` の代わりに引数レジスタの番号を指定します。上記の例を正しく記述すると、次のようになります。

```
z = fasm("mulq    %a0, %a1, %a1;"
        "stq      %a1, 0(%a2);"
        "ldt      %f0, 0(%a2);"
        "cvtqf    %f0, %f0;", x=10, y=5, &temp);
```

整数から浮動小数点レジスタへの変換に使用されるメモリ位置が、`asm` のコードで使用できる点に注意してください。これは、この目的のために C コード内で割り当てられた変数のアドレスを引数として渡すことで行います。

- 結果を期待どおりの場所に置くには、リターン値レジスタをメタ言語で指定しなければなりません。
- 引数とリターン値のない命令では、引数を省略します。たとえば次のようになります。

```
asm("MB");
```

プラグマ・プリプロセッサ指示文

#pragma 指示文は、コンパイラごとに異なる機能をインプリメントするための標準的な方法です。この章では、C コンパイラでサポートされている処理系固有のプラグマについて説明します。

- #pragma assert (3.1 節)
- #pragma environment (3.2 節)
- #pragma extern_model (3.3 節)
- #pragma extern_prefix (3.4 節)
- #pragma inline (3.5 節)
- #pragma intrinsic および #pragma function (3.6 節)
- #pragma linkage (3.7 節)
- #pragma member_alignment (3.8 節)
- #pragma message (3.9 節)
- #pragma optimize (3.10 節)
- #pragma pack (3.11 節)
- #pragma pointer_size (3.12 節)
- #pragma unroll (3.13 節)
- #pragma use_linkage (3.14 節)
- #pragma weak (3.15 節)

Compaq C のすべてのインプリメンテーションでサポートされているプラグマについては、『*Compaq C* 言語リファレンス・マニュアル』に記載があります。

プラグマの中には、デフォルトでマクロ展開を実行するものがあります。『*Compaq C* 言語リファレンス・マニュアル』にそのようなプラグマの一覧があります。プラグマ名やキーワードと同じ名前のマクロを定義しているプ

プログラムを移植する際に、マクロ展開を避けるため、接頭語としてプラグマ名に下線を 2 つ付ける方法についても説明しています。

プラグマ名に接尾語の `_m` を付加すると、どのプラグマに対してもマクロ展開を強制することができます。プラグマ名の後に `_m` がある場合、プラグマ名の後のテキストはマクロ置換であると見なされます。3.1.3 項の例を参照してください。

3.1 #pragma assert 指示文

`#pragma assert` 指示文を使用すると、コンパイラがより効率的なコードを生成するために使用できる、プログラムに関するアサーションを指定できます。

`#pragma assert` 指示文は、プログラムを正しく動作させるために必要なものではありません。ただし、`#pragma assert` を指定する場合は、指定するアサーションが正しくないと、プログラムが誤動作するおそれがあります。

`#pragma assert` 指示文の形式は次のとおりです。

```
#pragma assert func_attrs(identifier-list) function-assertions
#pragma assert global_status_variable(variable-list)
#pragma assert non_zero(constant-expression) "string-literal"
```

3.1.1 #pragma assert func_attrs

この形式の `#pragma assert` 指示文を使用すると、関数の属性にアサーションが設定されます。

この形式のプラグマの構文は、次のとおりです。

```
#pragma assert func_attrs(identifier-list) function-assertions
identifier-list
```

関数識別子のリスト。コンパイラは、`function-assertions` に基づいて、これらの関数に仮定条件を設定できます。複数の識別子を指定する場合は、コンマで区切ります。

function-assertions

関数に対してコンパイラが仮定条件を設定するために使用するアサーションのリスト。以下のアサーションの中から 1 つ以上指定し、複数指定する場合はホワイト・スペースで区切ります。

`noreturn`

コンパイラは、ルーチンを呼び出した後、ルーチンから戻らないものと仮定することができます。

`nocalls_back`

コンパイラは、この関数から制御を戻すまでは、そのソース・モジュール内のルーチンが呼び出されないと仮定することができます。

`nostate`

コンパイラは、関数のリターン値が関数の引数のみによって決まり、この関数には副作用がないと仮定することができます。関数に対して `noeffects` と `nostate` の両方がマークされると、コンパイラは、この関数への重複する呼び出しを削除することができます。

`noeffects`

コンパイラは、この関数には、関数のリターン値の設定以外の効果がないと仮定することができます。コンパイラは、関数呼び出しからのリターン値が一度も使用されないと判断した場合、その呼び出しを削除することができます。

`file_scope_vars(option)`

コンパイラは、ファイル範囲で (内部または外部リンケージで) 宣言された変数に関数がアクセスする方法についての仮定条件を設定できます。 *option* は、以下のキーワードのいずれかになります。

`none`

関数は、型が `volatile` でなく、また `#pragma assert global_status_variable` にリストされていないファイル範囲の変数に対して、読み取りも書き込みも行いません。

`noreads`

関数は、型が `volatile` でなく、また `#pragma assert global_status_variable` にリストされていないファイル範囲の変数に対して、読み取りを行いません。

`nowrites`

関数は、型が `volatile` でなく、また `#pragma assert global_status_variable` にリストされていないファイル範囲の変数に対して、書き込みを行いません。

`format (style, format-index, first-to-check-index)`

コンパイラは、フォーマット文字列に対して型検査される `printf` または `scanf` スタイルの引数をとこの関数が取ると仮定することができます。

フォーマット属性により、引数としてフォーマット文字列を取る独自の関数を指定して、コンパイラにこれらの関数への呼び出しのエラーをチェックさせることができます。コンパイラは、ライブラリ関数 `printf`, `fprintf`, `sprintf`, `snprintf`, `scanf`, `fscanf`, `sscanf` に対して、`intrinsic` が有効になっている場合 (省略時) に、フォーマットをチェックします。フォーマット属性を使用すると、これらの関数に対して `intrinsic` が有効でない場合に、フォーマットがチェックされるようにすることができます。

style

フォーマット文字列の解釈方法を指定します。 `printf` と `scanf` のいずれかを指定します。

format-index

どの引数がフォーマット文字列引数かを指定します (1 から始まる)。

first-to-check-index

フォーマット文字列に対してチェックを行う、最初の引数の番号を指定します。

チェックする引数がない場合 (`vprintf` など) は、3 番目のパラメータにゼロを指定します。この場合、コンパイラはフォーマット文字列に矛盾がないことだけをチェックします。たとえば、次のように宣言すると、コンパイラは `your_printf` の呼び出しの引数について、`printf` スタイルのフォーマット文字列引数 `your_format` との整合性をチェックします。


```
extern int
your_printf (void *your_object, const char *your_format, ...);
#pragma assert func_attrs(your_printf) format (printf, 2, 3)
```

フォーマット文字列 (*your_format*) は、関数 *your_printf* の 2 番目の引数であり、チェックする引数は 3 番目の引数から始まるので、フォーマット属性の正しいパラメータは、2 と 3 です。

この形式の `#pragma assert` 指示文は、ファイル範囲に記述しなければなりません。

identifier-list 内の識別子は、`#pragma assert` 指示文の位置から見えるところで宣言されていなければなりません。

各 `#pragma assert func_attrs` 指示文で異なるアサーションを指定する限りは、複数の指示文に同じ関数を記述できます。たとえば、次の記述は有効です。

```
#pragma assert func_attrs(a) noreads_back
#pragma assert func_attrs(a) file_scope_vars(noreads)
```

次の記述は無効です。

```
#pragma assert func_attrs(a) file_scope_vars(noreads)
#pragma assert func_attrs(a) file_scope_vars(nowrites)
```

3.1.2 `#pragma assert global_status_variable`

この形式の `#pragma assert` 指示文を使用すると、グローバル状態変数とみなされる変数を指定できます。グローバル状態変数は、`#pragma assert func_attrs file_scope_vars` 指示文によって関数に指定されるアサーションから除外されます。

この形式の指示文の構文は次のとおりです。

```
#pragma assert global_status_variable(variable-list)
```

variable-list は変数のリストです。

この形式の `#pragma assert` 指示文は、ファイル範囲に記述しなければなりません。

variable-list 内の変数は、`#pragma assert` 指示文の位置から見えるところで宣言されていなければなりません。

3.1.3 #pragma assert non_zero

この形式の #pragma assert 指示文の構文は次のとおりです。

```
#pragma assert non_zero(constant-expression) "string-literal"
```

コンパイラは、この指示文を見つけると *constant-expression* を評価します。式がゼロであれば、コンパイラは、指定した文字列リテラルとコンパイル時定数の両方を含むメッセージを出力します。たとえば、次のように指定したとします。

```
#pragma assert non_zero(sizeof(a) == 12) "a is the wrong size"
```

コンパイラは、`sizeof(a)` が 12 でないと判断した場合に次のようなメッセージを出力します。

```
cc: Warning: a.c, line 4: The assertion "sizeof(a)==12" was
not true, a is the wrong size. (assertfail)
```

`assert` オプションの `func_attrs` および `global_status_variable` の場合と異なり、`#pragma assert non_zero` は関数本体の内部と外部のどちらにも置くことができます。関数本体の内部で使用する場合、`#pragma` は文を記述できる位置に記述できますが、文として扱われることはありません。関数本体の外部で使用する場合、`#pragma` は宣言を記述できる位置に記述できますが、宣言として扱われることはありません。

constant-expression 内の変数は、`#pragma assert` 指示文の位置から見えるところで宣言されていなければなりません。

`#pragma assert` はプリプロセッサに対してマクロ置換を行わないので、`#pragma assert_m` 指示文を使用しなければならない場合があります。次に示すような、`struct` のサイズと、そのいずれかの要素のオフセットの両方を確認するプログラムがあるとします。

```
#include <stddef.h>
typedef struct {
    int a;
    int b;
} s;
#pragma assert non_zero(sizeof(s) == 8) "sizeof assert failed"
#pragma assert_m non_zero(offsetof(s,b) == 4) "offsetof assert failed"
```

`offsetof` はマクロなので、2 番目の指示文を `assert_m` にして、`offsetof` が正しく展開されるようにしなければなりません。

3.2 #pragma environment 指示文

#pragma environment 指示文を使用すると、すべてのコンテキスト・プラグマの状態を設定、保存、リストアすることができます。コンテキスト・プラグマには、次のものがあります。

```
#pragma extern_model
#pragma extern_prefix
#pragma member_alignment
#pragma message
#pragma pack
#pragma pointer_size
```

コンテキスト・プラグマは以前の状態の保存とリストアを行うプラグマであり、通常は、同じタイプのプラグマを使用するヘッダ・ファイルを取り込む前後の状態です。

#pragma environment 指示文は、インクルード・ファイルを周囲のプログラムで設定されるコンパイル・コンテキストから保護するとともに、周囲のプログラムを、それがインクルードするヘッダ・ファイルで設定されるコンテキストから保護します。

このプラグマの構文は次のとおりです。

```
#pragma environment [ command_line | header_defaults | restore | save ]
```

command_line

すべてのコンテキスト・プラグマの状態を、コマンド行に指定されたとおりに設定します。このプラグマを使用すると、ヘッダ・ファイルが取り込まれる前に有効になる環境プラグマからヘッダ・ファイルを保護します。

header_defaults

すべてのコンテキスト・プラグマの状態を省略時の値に設定します。これは、コマンド行オプションもプラグマも指定しないでプログラムをコンパイルした状況と同等ですが、このプラグマはヘッダ・ファイルに適するようにプラグマ・メッセージ状態を #pragma nostandard に設定します。

restore

すべてのコンテキスト・プラグマの現在の状態をリストアします。

save

すべてのコンテキスト・プラグマの現在の状態を保存します。

ソース・コードを変更しなくても、`#pragma environment` を使用することにより、追加のコンパイル・コンテキストをもたらす可能性のある言語の拡張などからヘッダ・ファイルを保護することができます。

ヘッダ・ファイルは取り込んだファイルからプラグマの状態を選択して継承することができ、その後、追加のプラグマを必要に応じて使用することにより、コンパイルを非省略時の状態に設定します。次の例を参照してください。

```
#ifndef __pragma_environment
#pragma __environment save 1
#pragma __environment header_defaults 2
#pragma member_alignment restore 3
#pragma member_alignment save 4
#endif

.
.    /*contents of header file*/
.
#endif __pragma_environment
#pragma __environment restore
#endif
```

この例に関する説明は次のとおりです。

- ❶ すべてのコンテキスト・プラグマの状態を保存します。
- ❷ 省略時のコンパイル環境を設定します。
- ❸ `#pragma __environment save` によってプッシュされた `#pragma member_alignment` スタックから、メンバ位置合わせコンテキストをポップして、メンバ位置合わせコンテキストを以前の状態にリストアします。
- ❹ メンバ位置合わせコンテキストをスタックにプッシュして戻し、`#pragma __environment restore` がエントリをポップできるようにします。

このように、ヘッダ・ファイルが継承するメンバ位置合わせコンテキストを除き、ヘッダ・ファイルはすべてのプラグマから保護されています。

3.3 #pragma extern_model 指示文

#pragma extern_model 指示文は、外部リンケージを持つデータ・オブジェクトをコンパイラがどう解釈するかを制御します。このプリAGMAを使用すると、外部 (extern) オブジェクトに使用するグローバル・シンボル・モデルを次のモデルの中から選択できます。

relaxed_refdef

このモデルでは、宣言には参照と定義があります。同じオブジェクトに対して初期化されていない定義が複数あっても構いません。リンカが解決して1つにまとめます。ただし、参照するためには少なくとも1つの定義がなければなりません。このモデルはほとんどのUNIXシステムのCコンパイラで使用され、Compaq Cの省略時モデルもこのモデルになっています。

strict_refdef

このモデルでは、宣言には参照と定義があります。参照されるどのシンボルに対しても、プログラム内での定義は正確に1つだけである必要があります。このモデルは、ANSI Cへの厳密な準拠が保証される唯一のモデルです。

注意

HP OpenVMSプラットフォームのCompaq Cでは、これ以外に、common_block および globalvalue という名前の2つのモデルをサポートしていますが、これらはTru64 UNIXではサポートしていません。

extern_model プリAGMAでグローバル・シンボル・モデルを選択すると、別のextern_model プリAGMAを指定するまで、外部記憶クラスを持つオブジェクトの宣言は、すべて指定したモデルに従って処理されます。

たとえば、次のようなプリAGMAがあるとします。

```
#pragma extern_model strict_refdef "progsec3"
```

このプラグマを指定すると、それ以降のファイルレベルの宣言は、`strict_refdef` モデルに従ったグローバル・シンボルの宣言として処理されます。

```
int x = 0;
extern int y;
```

外部モデルが何であっても、コンパイラはANSI Cの規則に従って宣言が定義と参照のどちらかを判断します。外部定義は、記憶クラス・キーワードがないファイルレベルの宣言か、または記憶クラス・キーワード `extern` を含むファイルレベルの宣言で、初期化されているものです。外部参照は、記憶クラス・キーワード `extern` のある宣言で、初期化されていないものです。上記の例では、`x` の宣言はグローバル定義であり、`y` の宣言はグローバル参照です。

ヘッダ・ファイルとプログラム・テキストの小さい範囲で、他の部分への影響なしに `#pragma extern_model` を使用できるように、コンパイラの外部モデル状態のスタックが用意されています。詳細は、3.3.4 項および 3.3.5 項を参照してください。

以下の項では、`#pragma extern_model` 指示文のさまざまな形式について説明します。

3.3.1 構文

`#pragma extern_model` 指示文の構文は次のとおりです。

```
#pragma extern_model model_spec [attr[,attr]...]
model_spec
```

次のいずれかです。

```
relaxed_refdef
strict_refdef "name"
```

"name" は、任意の定義に対するプログラム・セクション (psect) の名前です。

```
[attr[,attr]...]
```

psect 属性の指定 (省略可能)。以下に示す属性指定セットの中から1つだけ選択します。

`shr|noshr`

`psect` は、メモリ内で共有できる (`shr`) 場合と共有できない (`noshr`) 場合があります。省略時は `noshr` です。

`wrt|nowrt`

`psect` に含まれるデータは、変更できる (`wrt`) 場合と変更できない (`nowrt`) 場合があります。省略したときは、`psect` 内の最初の変数によって決まります。変数に `const` 型修飾子 (または `readonly` 識別子) がある場合、`psect` は `nowrt` に設定されます。それ以外の場合は、`wrt` に設定されます。

`ovr|con`

同じ名前の `psect` は、連結されるか (`con`)、または同じメモリ位置にオーバーレイされます (`ovr`)。省略時は、`strict_refdef` では `con`、`relaxed_refdef` では `over` になります。

`4|octa|5|6|7|8|9|10|11|12|13|14|15|16|page`

これは、数値データの整列方法を指定します。省略時の整列は `octa` です。数値を指定すると、`psect` は、2 のその数値乗の位置に整列されます。

`strict_refdef extern_model` では、次のような `psect` 属性を指定することもできます。

`noreorder`

この属性を指定すると、セクション内の変数が、定義した順に割り付けられます。この属性指定は、省略時はオフになっています。

次の例では、初期化された変数が 64 KB (2^{16}) の境界に整列されます。`noreorder` 属性指定は、変数が宣言された順に割り付けられることを意味します。

```
#pragma extern_model save
#pragma extern_model strict_refdef "progsecA" 16,noreorder
int var1 = 5;
int var2 = 6;
#pragma extern_model restore
```

次の例では、(書き込み不可) 変数が、データ・キャッシュ・ラインの境界に整列されます。

```
#pragma extern_model save
#pragma extern_model strict_refdef "progsecB" 3,noreorder,nowrt
const long c_v1 = 1;
const long c_v2 = 2;
const long c_v3 = 2;
const long c_v4 = 2;
#pragma extern_model restore
```

relaxed_refdef モデルでは、psect 属性は、仮定義で宣言した変数に影響を与えません。次のようなコードがあったとします。

```
#pragma extern_model relaxed_refdef 5
int a;
int b=6;
#pragma extern_model strict_refdef 5
int c;
```

変数 a は仮定義なので、省略時の octa ワード (2^{**4} つまり 16 バイト) の境界に整列されます。しかし、b は初期化されているので 32 バイト (2^{**5}) の境界に整列されます。c は仮定義ですが、strict_refdef モデルで定義されているので 32 バイト (2^{**5}) の境界に整列されます。

注意

通常、psect 属性はシステム・プログラマが使用します。システム・プログラマは、通常はマクロ内で行われる宣言をしなければなりません。これらの属性のほとんどは、通常の C プログラムでは不要です。

3.3.2 #pragma extern_model relaxed_refdef

このプリグマは、コンパイラの外部データのモデルを、UNIX システムで使用される relaxed_refdef モデルとします。

#pragma extern_model relaxed_refdef 指示文の構文は、次のとおりです。


```
#pragma extern_model relaxed_refdef [attr[,attr]...]
```

3.3.3 #pragma extern_model strict_refdef

このプラグマは、コンパイラの外部データのモデルを `strict_refdef` モデルとします。このモデルは、プログラムを ANSI C に厳密に準拠させたいときに使用します。

`#pragma extern_model strict_refdef` 指示文の構文は、次のとおりです。

```
#pragma extern_model strict_refdef "name" [attr[,attr]...]
```

引用符に囲まれた `name` を指定すると、定義に対する psect の名前になります。

3.3.4 #pragma extern_model save

このプラグマは、コンパイラの現在の外部モデルをスタックにプッシュします。スタックには、`shr/noshr` 状態や引用符で囲まれた psect 名など、外部モデルに対応する情報がすべて記録されます。

このプラグマの構文は次のとおりです。

```
#pragma extern_model save
```

`#pragma extern_model` スタックに保存できるエントリの数は、コンパイラが使用できるメモリ容量のみによって制限されます。

3.3.5 #pragma extern_model restore

このプラグマは、コンパイラの外部モデルのスタックをポップします。スタックからポップされた内容で外部モデルの状態が設定されます。スタックには、`shr/noshr` 状態や引用符で囲まれた psect 名など、外部モデルに対応する情報がすべて記録されます。

このプラグマの構文は次のとおりです。

```
#pragma extern_model restore
```

空のスタックをポップすると、警告メッセージが出力され、コンパイラの外部モデルは変わりません。

3.4 #pragma extern_prefix 指示文

#pragma extern_prefix 指示文は、コンパイラの外部名合成を制御します。リンカはこの外部名を使って、外部名要求を解決します。

文字列引数を使って #pragma extern_prefix を指定すると、C コンパイラは、プラグマ指定に続く宣言によって生成されたすべての外部名の冒頭にその文字列を付加します。

このプラグマは、ライブラリを作成する上で有用です。このライブラリを利用して、機能コードをライブラリ内の外部名に付加できます。

このプラグマの構文は、次のとおりです。

```
#pragma extern_prefix "string" [(id,...)]
#pragma extern_prefix save
#pragma extern_prefix restore
```

引用符で囲まれた *string* は、プラグマ指定に続く宣言内の外部名に付加されます。

オプション・リスト [(id,...)] を使って、特定の外部識別子に接頭語を指定することもできます。

save および restore キーワードは、それぞれ現在のプラグマ接頭語文字列の保存、および保存済みのプラグマ接頭語文字列のリストアに使用できます。

プラグマによって指定されなかった場合、省略時の外部識別子用の接頭語は空文字列です。

推奨される使用法は、次のとおりです。

```
#pragma extern_prefix save
#pragma extern_prefix " prefix-to-prepend-to-external-names "
...some declarations and definitions ...
#pragma extern_prefix restore
```

extern_prefix が有効であり、#include を使ってヘッダ・ファイルをインクルードしているときに、extern_prefix をヘッダ・ファイルの extern 宣言に適用したくない場合は、次の順序でコードを記述します。

```
#pragma extern_prefix save
#pragma extern_prefix ""
#include ...
#pragma extern_prefix restore
```

上記の順序で指定しない場合、インクルードされたファイル内での定義について、接頭語が外部識別子の冒頭に付加されます。

注意

`#pragma extern_prefix` でオプションの識別子を指定する際に、以下の内容が適用されます。

- 各 *id* について、プラグマの位置にその *id* の可視状態の宣言が存在してはなりません。存在していると、警告が発行されて、その *id* に対する効果はありません。
- 空でない接頭語を持つプラグマによる影響を受けた各 *id* は、同じコンパイル単位内で外部リンケージを使った宣言が行われることが期待されます。コンパイルの終了までにそのような宣言が存在しない場合、コンパイラは省略時の情報を発行します。
- *id* リスト形式のプラグマ、またはリストされた *id* の宣言を、他の形式のプラグマが制御するソース・コード領域内に配置することは許容されています。2つの形式が相互に作用し合うことはありません。*id* リスト形式は、常に他の形式に優先します。
- `save/restore` スタックと *id* リストとが、相互に作用することはありません。
- 複数のプラグマに同一の *id* が使用される場合、2番目のプラグマの接頭語が空 ("") であるか、または前のプラグマの接頭語と一致する場合を除き、省略時のメッセージが発行されます。どんな場合であっても、最後に遭遇した接頭語が他のすべてに優先されます。

3.5 #pragma inline 指示文

関数のインライン化とは、関数呼び出しのインライン展開を意味します。つまり、関数呼び出しを関数コードそのものと置換します。関数のインライン展開は、関数呼び出しのオーバヘッドを無くすとともに、展開したコードにコンパイラの一般的な最適化手法を適用することによって、実行

時間を短縮します。マクロの使用と比較すると関数のインライン化には次のような利点があります。

- 引数の評価は 1 回だけでよい。
- 優先順序の問題を避けるためのカッコの濫用が不要である。
- 実際の展開はコマンド行で制御できる。
- インライン展開をしない場合と意味規則は全く同じである。マクロを使用した場合は、全く同じではない。

次のプリプロセッサ指示文が、関数のインライン化を制御します。

```
#pragma inline (id, ...)
#pragma noinline (id, ...)
```

このとき、*id* は関数 ID です。

- `#pragma inline` 指示文で関数が指定されている場合、その関数の呼び出しは、可能であれば、インライン・コードとして展開されます。
- `#pragma noinline` 指示文で関数が指定されている場合は、その関数はインライン・コードとして展開されません。
- 同じ関数が `#pragma inline` 指示文および `#pragma noinline` 指示文の両方で指定されている場合は、エラー・メッセージが発行されます。

関数をインライン展開する場合は、その関数定義を関数呼び出しと同じモジュールに記述しておく必要があります (`-ifc` オプションを指定してモジュール間でのインライン展開を可能にしている場合を除く)。この定義の位置は、関数呼び出しの前でも後でも構いません。

cc コマンドにオプション `-O3`, `-O4`, `-inline size`, `-inline speed`, あるいは `-inline all` が指定されている場合、コンパイラは、`#pragma inline` あるいは `#pragma noinline` 指示文のどちらにも指定されていない関数の呼び出しのうち適切なものに関して展開します。この場合、展開するかどうかは次の関数特性によって決定されます。

- 大きさ
- その関数の呼び出し回数
- 次の制限を満たしている関数
 - パラメータのアドレスを扱わない。

- struct 引数のフィールド。 struct へのポインタである引数は制限されない。
- 関数の引数にアクセスするために varargs あるいは stdarg パッケージを使用していない。

これらのパッケージは引数が補助メモリ位置にあることを要求しますが、インライン展開はこれを満たしません。

最適化レベル -O2 では、C コンパイラは小さな静的ルーチンだけをインライン化します。

#pragma inline 指示文は、大きさや呼び出し回数にかかわらずインライン展開するようにします。

3.6 #pragma intrinsic および #pragma function 指示文

intrinsic として宣言できる関数もあります。 Intrinsic 関数の中では、ある状況で関数呼び出しを避けるために、C コンパイラが最適化コードを生成します。

表 3-1 に Intrinsic として宣言できる関数を示します。

表 3-1: Intrinsic 関数

abs	fabs	labs
printf	fprintf	sprintf
strcpy	strlen	memcpy
memmove	memset	alloca
bcopy	bzero	

関数が Intrinsic として扱われるかどうかを制御するには、次の指示文のうちのいずれかを使用します。 *func_name_list* は、コンマで区切った複数の関数名をカッコで囲んだリストです。

```
#pragma intrinsic (func_name_list)
#pragma function (func_name_list)
#pragma function ()
```

#pragma intrinsic 指示文は、関数の Intrinsic としての処理を使用可能にします。 #pragma intrinsic 指示文が有効に設定されると、コンパイラは関数がどのように動作するかを認識するため、より効率的なコード

を生成します。関数の宣言は、プリAGMAが処理されるときに有効でなければなりません。

#pragma function 指示文は、関数の Intrinsics としての処理を使用不能にします。空の func_name_list を持つ関数プリAGMAは、すべての関数に対する Intrinsics としての処理を使用不能にします。

コンパイラに対応する組み込み (built-in) を持つ標準ライブラリ関数もあります。組み込みは関数の同義名で、関数を Intrinsics として宣言する処理に似ています。次の表のような組み込みが提供されています。

関数	同義名
abs	__builtin_abs
labs	__builtin_labs
fabs	__builtin_fabs
alloca	__builtin_alloca
strcpy	__builtin_strcpy

Intrinsics や組み込み関数は、いくつかの手法で使うことができます。関数の宣言を持つヘッダ・ファイルは、表 3-1 に示す関数用の #pragma intrinsic 指示文を持っています。このプリAGMAを使用可能にするには、プリプロセッサ・マクロ _INTRINSICS を定義しなければなりません。alloca に対しては、alloca.h をインクルードするだけです。

たとえば、abs の Intrinsics 関数を得るためには、プログラムは stdlib.h をインクルードして、-D_INTRINSICS でコンパイルするか、または stdlib.h をインクルードする前に、_INTRINSICS を #define 指示文で定義する必要があります。

組み込み処理を使用可能にする方法の 1 つに、-D スイッチを使用する方法があります。たとえば、fabs の組み込みを使用可能にするには、次のいずれかの方法でコンパイルします。

```
% cc -Dfabs=__builtin_fabs prog.c
% cc -Dabs=__builtin_abs prog.c
```

ここまでの関数の Intrinsics としての処理は、関数とその使用方法によって異なります。最適化の結果は次のとおりです。

- 次の関数はインライン化される。

abs

```
fabs
labs
alloca
```

関数呼び出しのオーバーヘッドが低減します。

- 場合によっては、`printf` および `fprintf` 関数は変換され、形式文字列と引数の数および種類によって、`puts`、`putc`、`fputs`、`fputc`、あるいはこれらと同等の関数を呼び出す。
- 特定のインスタンスでは、`sprintf` 関数は、`strcpy` の呼び出しにインライン化あるいは変換される。
- `strcpy` 関数は、ソース文字列 (2 番目の引数) が文字列リテラルの場合にはインライン化される。

3.7 #pragma linkage 指示文

`#pragma linkage` 指示文を使用すると、リンケージ・タイプを指定することができます。リンケージ・タイプは、関数によるレジスタの使用方法を指定します。これを使用すると、関数が使用するレジスタを指定することができます。また、関数の特性 (たとえば、パラメータを引き渡したり、値を返すレジスタ) や、関数を変更できるレジスタも指定することができます。 `#pragma use_linkage` 指示文は、以前に定義されたリンケージを関数と対応付けます (3.14 節を参照)。

`#pragma linkage` 指示文は、(関数が C で書かれている場合) 呼び出し側と関数コンパイルの両方に作用します。関数がアセンブラで書かれている場合は、リンケージ・プラグマを使用して、アセンブラがレジスタを使用する方法を記述することができます。

`#pragma linkage` 指示文の構文は、次のとおりです。

```
#pragma linkage linkage-name = (characteristics)
```

linkage-name

定義するリンケージ・タイプを識別します。C 識別子の形式で指定します。リンケージ・タイプは固有の名前空間を持っているため、コンパイル単位内の他の識別子やキーワードと競合することはありません。

characteristics

パラメータが引き渡される場所，関数の結果が返される場所，および関数呼び出しによって変更されるレジスタに関する情報を指定します。

register-list を指定する必要があります。 *register-list* は，*rn* または *fn* のいずれかのレジスタ名をコンマで区切ったリストです。 *register-list* にはカッコで囲んだサブリストを含めることができます。 *register-list* を使用して，構造体の引数や関数の結果型を記述します。 このとき，構造体の各メンバは単一のレジスタで引き渡されます。 たとえば，次のように指定します。

```
parameters(r0, (f0, f1))
```

これは 2 つのパラメータを持つ関数の例です。 最初のパラメータはレジスタ *r0* に引き渡されます。 2 番目のパラメータは 2 つの浮動小数点メンバを持つ構造体型であり，レジスタ *f0* と *f1* に渡されます。

次の *characteristics* のリストは，項目をコンマで区切ってカッコで囲んで指定することができます。 これらのキーワードは任意の順に指定できます。

- `parameters (register-list)`

`parameters` 特性は，引数を特定のレジスタでルーチンへ引き渡します。

register-list の各項目には，ルーチンに引き渡す 1 つのパラメータを記述します。

構造体の引数は値によって引き渡すことができますが，構造体の各メンバは別々のパラメータ位置に引き渡されるという制約があります。 ただし，このようにすると，多数のレジスタが使用されるため，作成されるコードの処理速度が遅くなります。 コンパイラは，この状態を診断しません。

`parameters` オプションで有効なレジスタは，*r0* から *r25* までの整数レジスタと，*f0* から *f30* までの浮動小数点レジスタです。

構造体型は，各フィールドに対し最低 1 つのレジスタを必要とします。 構造体型に必要なレジスタ数が，プラグマで提供される数と同じであることがコンパイラによって確認されます。

- `result (register-list)`

コンパイラは、関数が値を返すためにどのレジスタを使用するかを認識しておく必要があります。この情報は `result` 特性を使用して引き渡します。

関数が値を返さない(つまり、関数のリターン型が `void`) 場合は、リンケージの一部として `result` を指定しないでください。

`register` オプションで有効なレジスタは、`r0` から `r25` までの汎用レジスタと、`f0` から `f30` までの浮動小数点レジスタです。

•

```
preserved (register-list)
nopreserve (register-list)
notused (register-list)
notneeded ((lp))
```

コンパイラは、関数によって使用されるレジスタと使用されないレジスタを区別するとともに、使用されるレジスタは関数呼び出しの間に保存されているかどうかを認識しておく必要があります。この情報を指定するには、`preserved`、`nopreserve`、`notused`、および `notneeded` のオプションを使用します。

- `preserved` レジスタは、関数呼び出しの前後で同じ値を保持します。
- `nopreserve` レジスタは、関数呼び出しの前後で同じ値を保持しているとはかぎりません。
- `notused` レジスタは、呼び出された関数で使用されることはありません。
- `notneeded` 特性は、特定の項目がこのリンケージを使用するルーチンで必要ないことを示します。`lp` キーワードは、指定された関数を呼び出すときに、リンケージ・ポインタ・レジスタ (`r27`) を設定する必要がないことを指定します。リンケージ・ポインタが必要になるのは、呼び出された関数がグローバルまたは `static` データにアクセスする場合です。レジスタが必要ないことを指定するのが有効かどうかを判断する必要があります。

`preserved`、`nopreserve`、`notused` のオプションで有効なレジスタは、`r0` から `r30` までの汎用レジスタと、`f0` から `f30` までの浮動小数点レジスタです。

`#pragma linkage` 指示文では、ネストした副構造体を含む構造体は、パラメータまたは特殊リンケージを持つ関数のリターン型としてはサポートされません。関連する特殊リンケージを持つ関数は、共用型を持つパラメータまたはリターン型はサポートしません。

次の特性は、レジスタ `f3` と `f4` の2つの要素を含む *simple-register-list* と、レジスタ `r0` とサブリスト (レジスタ `f0` と `f1` を含む) の2つの要素を含む *register-list* を指定します。

```
nopreserve(f3,f4)
parameters(r0,(f0,f1))
```

次の例は、そのような特性を使用するリンケージを示しています。

```
#pragma linkage my_link=(nopreserve(f3,f4),
                        parameters(r0,(f0,f1)),
                        notneeded (lp))
```

register-list のカッコで囲んだ表記法は、引数と `struct` 型の関数リターン値を記述します。この `struct` の各メンバは、単一のレジスタに引き渡されます。次の例では、`sample_linkage` は2つのパラメータを指定します。最初のパラメータはレジスタ `r0`, `r1`, および `r2` に引き渡され、2番目のパラメータは `f1` に引き渡されます。

```
struct sample_struct_t {
    int A, B;
    short C;
} sample_struct;

#pragma linkage sample_linkage = (parameters ((r0, r1, r2), f1))
void sub (struct sample_struct_t p1, double p2) { }

main()
{
    double d;

    sub (sample_struct, d);
}
```

3.8 #pragma member_alignment 指示文

省略時の設定では、コンパイラは構造体のメンバを自然境界に合わせます。構造体メンバのバイト合わせを指定する場合は、`#pragma [no]member_alignment` プリプロセッサ指示文を使用してください。

このプラグマの構文は次のとおりです。

```
#pragma member_alignment [save | restore]
```

```
#pragma nomember_alignment [base_alignment]
```

```
save | restore
```

パック境界合わせを含め、メンバの境界合わせの現在の状態を保管するために、あるいは前の状態をリストアするためにそれぞれ使用できます。状態の制御は、`member_alignment` あるいは `nomember_alignment` を必要とするヘッダ・ファイル、もしくは、すでに設定されている `member_alignment` を含む必要のあるヘッダ・ファイルの作成のために必要になります。

base_alignment

base_alignment パラメータを使用すると、構造体のベース境界合わせを指定できます。*base_alignment* には、次のキーワードのいずれかを使用します。

- `byte` (1 byte)
- `word` (2 bytes)
- `longword` (4 bytes)
- `quadword` (8 bytes)
- `octaword` (16 bytes)

構造体メンバの自然境界合わせをリストアするには、`#pragma member_alignment` を使用します。

`#pragma member_alignment` を使用すると、コンパイラは、構造体メンバを次のバイトではなくそのメンバのタイプに合った次の境界に合わせます。たとえば、`int` 変数は次のロングワード境界に合わせられ、`short` 変数は次のワード境界に合わせられます。

`#pragma nomember_alignment` を使用すると、構造体メンバをバイト境界合わせに指定します。

`pragma pack` 指示文では、構造体メンバの境界合わせをバイト、ワード、ロングワード、あるいはクォードワードに指定することができます。`#pragma pack` についての詳細は、3.11 節を参照してください。

#pragma member_alignment, #pragma nomember_alignment および #pragma pack の各プラグマの設定は、次のプラグマを処理するまで有効です。

3.9 #pragma message 指示文

#pragma message 指示文は、個々の診断メッセージまたは診断メッセージ・グループの発行を制御します。このプラグマは、メッセージの発行に関する他のどのコマンド行オプションよりも優先します。

#pragma message 指示文の構文は、次のとおりです。

```
#pragma message option1 (message-list)
#pragma message option2
#pragma message ("string")
```

3.9.1 #pragma message option1

この形式の #pragma message 指示文の構文は、次のとおりです。

```
#pragma message option1 (message-list)
```

option1 パラメータは、以下のキーワードのいずれかでなければなりません。

enable

メッセージ・リストで指定したメッセージの発行を有効にします。

disable

メッセージ・リストで指定したメッセージの発行を無効にします。無効にできるメッセージは、重大度が Warning または Information の場合に限られます。メッセージの重大度が Error または Fatal の場合、そのメッセージは無効にしようとしても関係なく発行されます。

emit_once

指定したメッセージは、コンパイル時に 1 度だけ出力されます。メッセージには、コンパイラがその原因となる条件を初めて検出した場合にのみ出力されるものがあります。コンパイラがその後、プログラムの中で同じ条件を検出しても、メッセージは出力されません。このようなメッセージには、たとえば言語拡張の使用に関するメッセージがあります。原因となる条件を検出するたびに毎回このようなメッセージを出力するには、*emit_always* オプションを使用します。

Errors と FataIs のメッセージは必ず出力されます。このようなメッセージに `emit_once` を指定することはできません。

`emit_always`

条件を検出するたび、毎回メッセージを出力します。

`error`

指定したメッセージの重大度を Error に変更します。Error および Fatal メッセージは、重大度をそれより低くすることはできません。(例外として、メッセージを Error から Fatal に上げ、次に Error に下げることができますが、Error から下げることはできません。Warning と Informational については、重大度を自由に変更できます。)

`fatal`

指定したメッセージの重大度を Fatal に変更します。

`informational`

指定したメッセージの重大度を Informational に変更します。Fatal および Error メッセージは重大度を低くすることができないことに注意してください。

`warning`

`message-list` 内の各メッセージの重大度を Warning に変更します。Fatal および Error メッセージは重大度を低くすることができないことに注意してください。

`message-list` パラメータは、以下のいずれかになります。

注意

省略時は、選択したコンパイラ・モードに対する診断メッセージがすべて発行されます。ただし、`check` グループは例外であり、メッセージの表示を明示的に有効にしなければなりません。

- 単一のメッセージ ID (カッコ付きまたはカッコなし)。メッセージ ID を取得するには、`cc` コマンドで `-verbose` オプションを使用します。

- 単一のメッセージ・グループ名 (カッコ付きまたはカッコなし)。メッセージ・グループ名は、次のとおりです。

`all`

コンパイラのメッセージすべて。

`alignment`

通常と異なる、あるいは効率の悪い整列になっているデータに関するメッセージ。

`c_to_cxx`

C++ コンパイラでコンパイルすると無効になるか別の意味になる C 機能の使用を通知するメッセージ。

`check`

正確で移植性があっても、まぎらわしいか保守性がないために、不適切と見なされたコードや操作を通知するメッセージ。たとえば、`if` 文でのテスト式としての代入などです。 `check` グループは `level5` メッセージを有効にすると定義されます。

`nonansi`

ANSI 規格外の機能の使用を通知するメッセージ。

`defunct`

廃止された機能の使用を通知するメッセージ。これらの機能は、初期の C コンパイラでは受け付けられていましたが、その後言語から削除されたものです。

`obsolescent`

ANSI C 規格では有効だが、廃止予定で将来の標準バージョンでは言語から削除されると思われる機能の使用を通知するメッセージ。

`overflow`

オーバーフローが発生したり、データの有効性が失われるおそれがある代入やキャストを通知するメッセージ。

`performance`

実行時の性能が低下するおそれがあるコードを通知するメッセージ。

portable

他のコンパイラやプラットフォームに移植できないおそれがある言語拡張やその他の構造が使われていることを通知するメッセージ。

preprocessor

疑問がある、あるいは移植性がない前処理構造の使用を通知するメッセージ。

questcode

疑問があるコーディングを通知するメッセージ。check グループと類似していますが、このグループのメッセージは、単に脆弱なスタイルを示すだけでなくプログラミングのエラーを示す場合が多くなっています。

returnchecks

関数のリターン値に関連するメッセージ。

uninit

初期化されていない変数の使用に関するメッセージ。

unused

使用されていない式、宣言、ヘッダ・ファイル、静的関数、コード・パスに関するメッセージ。

- 単一のメッセージ・レベル名 (カッコ付きまたはカッコなし)。メッセージ・レベル名は次のとおりです。

level1

重要なメッセージ。このグループのメッセージは、#pragma nostandard がアクティブの場合は表示されないため、レベル 0 のコア・メッセージほど重要ではありません。

level2

中程度に重要なメッセージ。Compaq の C では、省略時にレベル 2 になります。

level3

あまり重要でないメッセージ。

level4

役に立つ check/portable メッセージ。

level5

それほど役に立たない check/portable メッセージ。

level6

その他の冗長なメッセージ。

`#pragma message` の指定にかかわらず、特に指定しなくても有効になっている非常に重要なコンパイラ・メッセージのコアがあることに注意してください。これは level0 のメッセージと呼ばれ、ヘッダ・ファイルで発生するメッセージをすべて含み、`nostandard` と呼ばれるグループで構成されます。その他のメッセージ・レベルでは、この有効にされたメッセージのコアにメッセージが追加されます。

level0 は変更できません (無効と有効を切り替えたり、重大度を変更したり、`emit_once` 特性を変更することはできません)。ただし、アクションによって変更が許可されていれば、level0 のメッセージを個別に変更することはできます。たとえば、level0 の Warning または Informational を無効にしたり、level0 の Error を Fatal に変更したりすることなどはできます (個別のメッセージに対する変更の制限を参照してください)。また、あるレベルを有効にすると、それより値の小さいレベルのメッセージもすべて有効になります。したがって、level3 メッセージを有効にすると、level2 および level1 のメッセージも有効になります。また、あるレベルを無効にすると、それより値の大きいレベルのメッセージもすべて無効になります。したがって、level4 メッセージを無効にすると、level5 および level6 のメッセージも無効になります。

- メッセージ ID、グループ名、メッセージ・レベルをコンマで区切って、カッコ内に自由に並べたりリスト。

3.9.2 #pragma message option2

この形式の `#pragma message` 指示文の構文は、次のとおりです。

```
#pragma message option2
```

`option2` パラメータは、次のキーワードのいずれかでなければなりません。

save

どのメッセージが有効 (無効) になっているかという現在の状態を保存します。

restore

どのメッセージが有効 (無効) になっているかという、直前の状態をリストアします。

save および restore オプションは、主にヘッダ・ファイル内で役に立ちます。

3.9.3 #pragma message (" string ")

この形式の #pragma message 指示文は、Microsoft の #pragma message 指示文との互換性のためのもので、構文は次のとおりです。

```
#pragma message ("string")
```

この指示文は、指定された *string* をコンパイラ・メッセージとして出力します。たとえば、コンパイラがソース・ファイル内で次のような行に遭遇したとします。

```
#pragma message ("hello")
```

すると、コンパイラは、次のよう出力します。

```
cc: Info: a.c, line 10: hello (simplemessage)
#pragma message ("hello")
-----^
```

この形式のプラグマでは、マクロの置き換えが実行されます。たとえば、次のような使い方ができます。

```
#pragma message ("Compiling file " __FILE__)
```

3.10 #pragma optimize 指示文

#pragma optimize 指示文は、その指示文に続く関数定義の最適化の特性を設定します。このプラグマを使用すると、通常はコマンド行でコンパイル全体に対して設定する最適化制御オプションを、ソース・ファイル内で個別の関数に対して指定できます。このプラグマの形式は次のとおりです。

```
#pragma optimize settings
#pragma optimize save
#pragma optimize restore
#pragma optimize command_line
```

save および restore オプションは、現在の最適化の状態 (level, unroll count, ansi-alias setting, intrinsic setting) の保存と復元を行います。

command_line オプションを使用すると、最適化設定は、コンパイル時にコマンド行オプション cc で指定した時点で有効な設定に戻ります。

settings は、以下の値を組み合わせで指定します。

level

level は、最適化レベルを設定します。レベルは次のように指定します。

level=n

ここで、re n は 0 ~ 5 の整数です。

- | | |
|---|--|
| 0 | すべての最適化を無効にします。代入されていない変数のチェックを行いません。 |
| 1 | ローカルな最適化と、一部の共通部分式の認識を有効にします。コール・グラフによって、プロシージャをコンパイルする順序が決まります。 |
| 2 | level 1 の最適化を含みます。グローバルな最適化を有効にします。この最適化には、データフローの分析、コードの移動、強度の軽減、テストの置換、存在期間分割の分析、コードのスケジューリングが含まれます。 |
| 3 | level 2 の最適化を含みます。さらにグローバルな最適化を有効にして実行速度を改善します(コード・サイズは増える)。たとえば、整数の乗算および除算の展開(シフトを使用)、ループの展開、コードの繰り返しによる分岐の削除を行います。 |
| 4 | level 3 の最適化を含みます。プロシージャの相互作用の分析と小プロシージャの自動インライン化(追加 |

コード量は発見的に制限される)を有効にします。これが省略時のレベルです。

- 5 level 4 の最適化を含みます。ソフトウェアのパイプライン化をアクティブにします。これはループ展開の特別な形式であり、実行時の性能が改善できる場合があります。ソフトウェアのパイプライン化では、命令スケジューリングを用いてループ内での命令の停止をなくし、展開されていない複数のループに命令を再配置して性能を改善します。
- ソフトウェアのパイプライン化の候補になるループは常に、最も内側にあって分岐やプロシージャの呼び出しを含まないループです。ユーザのプログラムで level 5 を使用して効果があるかどうかを判断するには、同じプログラムを level 4 と level 5 でコンパイルして、プログラムの実行にかかる時間を測定する必要があります。使用可能なレジスタを使い尽くすようなループのあるプログラムでは、level 5 の方が実行時間が長くなります。

`unroll`

`unroll` 設定はループの展開を制御します。次のように指定します。

```
unroll=n
```

ここで、 n は負でない整数です。 `unroll= n` は、ループ本体を n 回展開することを意味します。ここで、 n は 0 ~ 16 の数値です。 `unroll=0` は、最適化プログラムの省略時の展開回数を使用することを意味します。 `unroll` は、レベル 3 以上の最適化で使います。

`ansi-alias`

`ansi-alias` は、`ansi-alias` の仮定条件を制御します。次のいずれかを指定します。

```
ansi_alias=on  
ansi_alias=off
```

`intrinsic`

`intrinsic` は、`intrinsic` の認識を制御します。次のいずれかを指定します。

```
intrinsicson  
intrinsicsoff
```

setting 句の間と、各句の “=” の前後のホワイト・スペースはオプションです。 `pragma` ではマクロ置換は行われません。

例:

```
#pragma optimize level=5 unroll=6
```

使用上の注意

- `level=0` 句がある場合、他の句は指定できません。
 - `#pragma optimize` 指示文はファイル範囲で、関数本体の外に記述しなければなりません。
 - `#pragma environment command_line` 指示文は、最適化の状態を、コマンド行で指定した状態にリセットします。
 - `#pragma optimize` で最適化状態のいずれかの設定を指定していない場合、その状態は変更されません。
 - 関数定義が検出されると、ソース内のその位置で現在設定されている最適化設定を用いてコンパイルが行われます。
 - 関数を `level=0` でコンパイルすると、コンパイラはその関数をインライン化しません。一般に、関数がインライン化されると、インライン化されたコードは、インライン化される関数に指定された最適化制御ではなく、呼び出し側で有効な最適化制御を用いて最適化されます。
-

3.11 #pragma pack 指示文

`#pragma pack` 指示文は、構造体の全メンバに関する境界合わせの制約を変更します。指示文は、構造体全体に対して作用するため、構造体の定義全体の前に指定しなければなりません。このプラグマの構文は、次のとおりです。

```
#pragma pack {n | (n) | ()}
```

n は、その後に続く構造体メンバが *n* バイト境界に位置合わせされることを指定する数字 (1, 2, 4 など) です。*n* に 0 (ゼロ) を指定した場合には、境界合わせは省略時の設定に戻ります。これは、cc コマンドの `-Zpn` オプションで設定されている場合があります。

pragma pack 指示文は、Microsoft Visual C++ でサポートしている push および pop 引数もサポートしています。

```
#pragma pack ( {push | pop} [, identifier] [, n])
```

push および pop 引数を使用すると、プログラムのコンポーネントまたがって使用する境界合わせの値を保存したり復元したりすることができます。これにより、境界合わせの指定が異なる場合でも、コンポーネントを単一の変換ユニットに連結できます。

- pragma pack に push 引数があると、そのたびに現在のパッキング境界合わせの値は内部コンパイラ・スタックに格納されます。*n* の値を指定すると、その値が新しい境界合わせの値になります。*identifier* で任意の名前を指定すると、それが新しい値に関連付けられます。
- pragma pack に pop 引数があると、そのたびにスタックの先頭の値が取得されて新しい境界合わせの値になります。空のスタックがポップされた場合、境界合わせの値は省略時の値に戻り、警告が発行されます。*n* の値を指定すると、その値が新しい境界合わせの値になります。

identifier を指定すると、一致する *identifier* の値が見つかるまで、スタックに格納された境界合わせの値はすべてスタックから削除されます。*identifier* に対応する値もスタックから削除され、*identifier* がプッシュされる直前に有効だった値が新しい境界合わせの値になります。*identifier* に一致する値が見つからなければ、境界合わせの値は省略時の値に戻り、警告が発行されます。

pragma pack の push および pop 引数を使用すると、ヘッダ・ファイルを検出した前と後で、境界合わせの値が確実に同じになるように、ヘッダ・ファイルを記述することができるようになります。以下に例を示します。

```
/* File name: myinclude.h */
#pragma pack (push,enter_myinclude)
.
. (your include file code)
.
#pragma pack (pop,enter_myinclude)
```

```
/* End of myinclude.h */
```

この例では、現在の境界合わせの値が識別子の `enter_myinclude` に関連付けられ、ヘッダ・ファイルの処理の前にプッシュされます。次に、インクルード・ファイルのコードが処理されます。インクルード・ファイルの処理が完了すると、ヘッダ・ファイルの末尾にある `#pragma pack` が、ヘッダ・ファイル内で発生する可能性のある中間の境界合わせ値と、`enter_myinclude` に対応する境界合わせの値を削除して、ヘッダ・ファイルの前と後で境界合わせの値が同じになるようにします。

`push` および `pop` 引数を使用して、現プログラムコードの設定と異なる境界合わせの値を設定するようなヘッダ・ファイルをインクルードすることもできます。以下に例を示します。

```
#pragma pack (push,before_myinclude)
#include <myinclude.h>
#pragma pack (pop,before_myinclude)
```

この例では、現在の境界合わせ値の保存、インクルード・ファイルの処理 (境界合わせ値がどうなるかは不明)、オリジナルの境界合わせ値の復元と処理することにより、`myinclude.h` で発生する可能性のある境界合わせ値の変更から、コードを保護しています。

3.12 #pragma pointer_size 指示文

`#pragma pointer_size` 指示文は、次の項目に関するポインタ・サイズの割り当てを制御します。

- 参照
- ポインタ宣言
- 関数宣言
- 配列宣言

このプラグマの構文は次のとおりです。

```
#pragma pointer_size { long | short | 64 | 32 } | { save | restore }
```

```
long | 64
```

コンパイラが別の `#pragma pointer_size` 指示文を処理するまでの間、この指示文に続いて宣言されているすべてのポインタ・サイズを 64 ビットに設定します。

short | 32

コンパイラが別の `#pragma pointer_size` 指示文を処理するまでの間、この指示文に続いて宣言されているすべてのポインタ・サイズを 32 ビットに設定します。

save | restore

それぞれ、現在のポインタ・サイズの保管、および保管されているポインタ・サイズのリストアを行います。save および restore オプションは、混合ポインタのサポートおよび旧オブジェクトへのインタフェースとなるヘッダ・ファイルの保護のため、特に有用です。複数のポインタ・サイズ・プラグマでコンパイルされたオブジェクトは、古いオブジェクトと互換性がなく、コンパイラは互換性のないオブジェクトが混在していることを認識できません。

たとえば、次のとおりです。

```
#pragma pointer_size long
/* pointer sizes in here are 64-bits */
#pragma pointer_size save
#pragma pointer_size short
/* pointer sizes in here are 32-bits */
#pragma pointer_size restore
/* pointer sizes in here are again 64-bits */
```

short ポインタの使用は、Tru64 UNIX システム上の Compaq C++ および Compaq C コンパイラに制限されています。ルーチンが C プログラミング言語以外の言語で書かれている場合は、プログラムで C++ ルーチンから別のルーチンへ short ポインタを引き渡してはなりません。また、Compaq C++ では、short ポインタを使用するアプリケーションで、short ポインタから long ポインタへ明示的に変換する必要があります。まず、short ポインタの使用を検討しているアプリケーションを移植した後、それを分析して、short ポインタを使用することが有益であるかどうかを判断します。関数定義におけるポインタ・サイズの相違は、関数にとってはそれほどのオーバーロードではありません。

C コンパイラは、次の状態のいずれかを検出すると、エラー・レベルの診断メッセージを表示します。

- 定義されている 2 つの関数の、ポインタ・サイズだけが異なっている。

- 2つの関数のリターン型だけが異なっている。

3.13 #pragma unroll 指示文

#pragma unroll 指示文は、その後の for ループで実行されるループ展開の量を制御します。この指示文のフォーマットは次のとおりです。

```
#pragma unroll (unroll-factor)
```

この指示文は、コンパイラに対して、*unroll-factor* 引数で指定した回数だけ for ループを展開するように指示します。

unroll-factor

0 から 255 までの整数の定数。コンパイラは、指示文に続く for ループを、この回数だけ展開します。値が 0 の場合、指示文は無視され、コンパイラは通常の方法でループの展開回数を判断します。値が 1 の場合、ループは展開されません。

制御を行うには、指示文を for 文の直前に置く必要があります。それ以外の位置に置くと、警告が発行され pragma は無視されます。

例

```
#pragma unroll (1)
for (i=0; i<1000; i++) {foo(i);}
```

3.14 #pragma use_linkage 指示文

#pragma linkage 指示文で特殊リンケージを定義した後は (3.7 節で説明)、#pragma use_linkage 指示文を使用して、リンケージを関数に関連付けます

このプラグマの構文は次のとおりです。

```
#pragma use_linkage linkage-name (id1, id2, ...)
```

linkage-name

以前に #pragma linkage 指示文で定義したリンケージの名前を指定します。

id1, id2, ...

指定したリンケージに関連付ける関数の名前または関数の型の typedef 名を指定します。

関数の型の typedef 名を指定すると、その型を用いて宣言した関数または関数へのポインタは、指定されたリンケージを持ちます。

#pragma use_linkage 指示文は、ソース・ファイルで、指定したルーチンを使用したり定義する前に記述する必要があります。このようにしない場合、予測できない結果が生じます。

次の例は、特殊リンケージを定義して、それをルーチンに対応付けます。このルーチンは、3つの整数パラメータをとり、最初のパラメータが引き渡された位置に1つの整数値を返します。

```
#pragma linkage example_linkage (parameters(r16, r17, r19), result(r16))
#pragma use_linkage example_linkage (sub)
int sub (int p1, int p2, short p3);

main()
{
    int result;

    result = sub (1, 2, 3);
}
```

この例では、result(r16) オプションは、関数の結果は通常の位置(r0)ではなく、r16に返されることを示しています。parameters オプションは、subに引き渡される3つのパラメータがr16、r17、およびr19に渡されることを示しています。

次の例では、関数 f1 と関数の型 t はどちらもリンケージ foo を持ちます。関数ポインタ f2 を用いて呼び出すと、特殊リンケージを用いて正しく関数 f1 が呼び出されます。

```
#pragma linkage foo = (parameters(r1), result(r4))
#pragma use_linkage foo(f1,t)

int f1(int a);
typedef int t(int a);

t *f2;

#include <stdio.h>

main() {
    f2 = f1;
    b = (*f2)(1);
}
```

3.15 #pragma weak 指示文

#pragma weak 指示文は、新たに弱外部シンボルを定義し、新しいシンボルと外部シンボルとを関連づけます。このプラグマの構文は、次のとおりです。

```
#pragma weak (secondary-name, primary-name)
```

強シンボルと弱シンボルについては， 2.8 節を参照してください。

シェアード・ライブラリ

シェアード・ライブラリは、省略時のシステム・ライブラリです。C コンパイラがコンパイルとリンクを行う際、省略時の設定ではシェアード・ライブラリを使用します。

この章では、次の内容について説明します。

- シェアード・ライブラリの概要 (4.1 節)
- シンボルの解決 (4.2 節)
- シェアード・ライブラリとのリンク (4.3 節)
- シェアード・ライブラリの指定解除 (4.4 節)
- シェアード・ライブラリの作成 (4.5 節)
- プライベートなシェアード・ライブラリの使用 (4.6 節)
- クイックスタートの使用 (4.7 節)
- シェアード・ライブラリとリンクしたプログラムのデバッグ (4.8 節)
- シェアード・ライブラリの実行時のロード (4.9 節)
- シェアード・ライブラリ・ファイルの保護 (4.10 節)
- シェアード・ライブラリのバージョン管理 (4.11 節)
- シンボル割り当て (4.12 節)
- シェアード・ライブラリの制限事項 (4.13 節)

4.1 シェアード・ライブラリの概要

シェアード・ライブラリは、メモリ内のどのアドレスへも配置できる実行可能ファイル・コードから構成されています。シェアード・ライブラリの命令のコピーは 1 つだけロードされます。アーカイブ (静的) ライブラリのようにライブラリを使用する各プログラムがライブラリのコピーをロードするのではなく、複数のプログラム間で 1 つのコピーを共有します。

シェアード・ライブラリを使用するプログラムは、次の点でアーカイブ・ライブラリを使用するプログラムよりもはるかに有利です。

- シェアード・ライブラリとリンクしたプログラムは、そのシェアード・ライブラリに変更が行われても、再コンパイルおよび再リンクする必要はありません。
- アーカイブ・ライブラリにリンクしたプログラムとは異なり、シェアード・ライブラリにリンクしたプログラムの場合、その実行可能プログラム・ファイルにライブラリ・ルーチンは含まれません。シェアード・ライブラリにリンクしたプログラムには、シェアード・ライブラリをロードするための情報、ロード時にそのルーチンやデータにアクセスするための情報が含まれています。

つまり、シェアード・ライブラリを使用すると、メモリやディスクの専有領域が少なくなります。複数のプログラムが1つのシェアード・ライブラリにリンクされる場合には、それぞれのプロセスで使用される物理メモリの量は、大幅に減少します。

シェアード・ライブラリの使用はユーザからは透過です。さらに、プログラムをアーカイブ・ライブラリとリンクさせる方法を採用することもできます。また、ユーザ独自のシェアード・ライブラリを作成し、他のユーザに使用させることもできます。大部分のオブジェクト・ファイルおよびアーカイブ・ライブラリは、シェアード・ライブラリにすることができます。シェアード・ライブラリにできるファイルについては、4.5 節を参照してください。

シェアード・ライブラリは、次の点でアーカイブ・ライブラリと異なります。

- シェアード・ライブラリは `ld` コマンドに適切なオプションを付けて作成するが、アーカイブ・ライブラリは、`ar` コマンドで作成する。

`ld` コマンドの詳細については、`ld(1)` を参照してください。

- シェアード・ライブラリは、リンクされると使用可能な任意のアドレスに置かれる。

実行時に、ローダ (`/sbin/loader`) がメモリ内のプライベートな仮想アドレス空間をシェアード・ライブラリに割り当てます。アーカイブ・ライブラリは、実行可能ファイルの一部としてリンクされるとメモリ内の定位置に置かれます。

- シェアード・ライブラリは `/usr/shlib` ディレクトリに常駐し、アーカイブ・ライブラリは `/usr/lib` ディレクトリに常駐する。

4-2 シェアード・ライブラリ

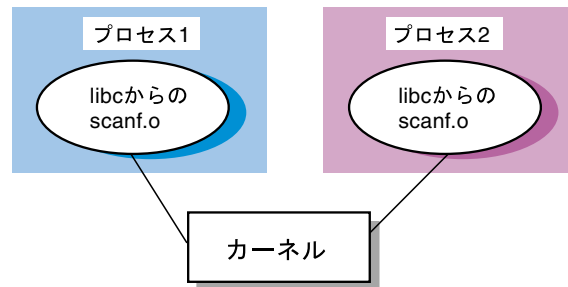
- シェアード・ライブラリの命名規則により、シェアード・ライブラリ名は接頭語 `lib` で始まり、接尾語 `.so` で終わる。

たとえば、共通の C 言語機能を含むライブラリ名は、`libc.so` となります。アーカイブ・ライブラリ名も接頭語 `lib` で始まりますが、接尾語 `.a` で終わります。

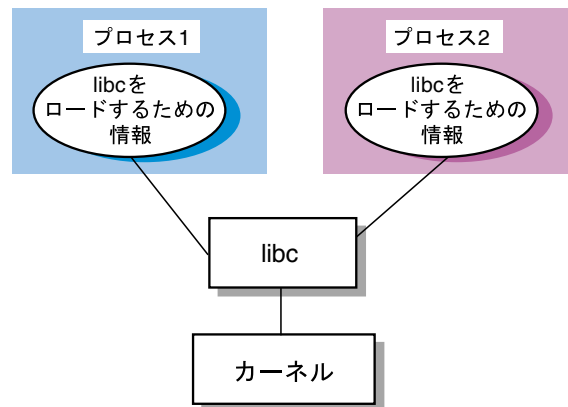
図 4-1 に、アーカイブ・ライブラリとシェアード・ライブラリの違いを示します。

図 4-1: アーカイブ・ライブラリとシェアード・ライブラリ

アーカイブ・ライブラリを使用しているアプリケーション:



シェアード・ライブラリを使用しているアプリケーション:



ZK-0474U-AIJ

4.2 シンボルの解決

シンボルの解決とは、プログラムまたはシェアード・ライブラリによりインポートされた未解決のシンボルを、そのシンボルをエクスポートするシェアード・ライブラリのパス名にマップする処理です。アーカイブ・ライブラリおよびシェアード・ライブラリのシンボル解決は、シェアード・オブジェクトのシンボルの最終的な解決がプログラム起動時まで行われないという点を除いて、ほとんど同じ方法で行なわれます。

次の各項では、以下の項目について説明します。

- リンカの探索パス (ld) (4.2.1 項)
- 実行時ローダの探索パス (/sbin/loader) (4.2.2 項)
- 名前の解決 (4.2.3 項)
- 未解決の外部シンボルの動作を解決するため、ld コマンドに付けるオプション (4.2.4 項)

4.2.1 リンカの探索パス

リンカ (ld) は -l オプションによって指定されたファイルを探索する場合、次に示した順序で各ディレクトリで探索します。リンカは、最初にシェアード・ライブラリ (.so) を探索します。

1. /usr/shlib
2. /usr/ccs/lib
3. /usr/lib/cmplrs/cc
4. /usr/lib
5. /usr/local/lib
6. /var/shlib

シェアード・ライブラリがない場合は、リンカはすべてのディレクトリでアーカイブ (.a) ライブラリを探索します。-no_archive オプションを使用すると、リンカによるアーカイブ・ライブラリの探索を禁止することができます。

4.2.2 実行時ローダの探索パス

特に指示がない限り、実行時ローダ (/sbin/loader) は、リンカと同じ探索パスを経由します。実行時ローダに次のように指示して、省略時の探索パスで指定されているディレクトリ以外のディレクトリを探索することができます。

- ld コマンドを `-rpath string` オプションとともに使用し、*string* を探索するディレクトリのリストに設定して、ディレクトリ・パスを指定する。
- プログラムを実行する前に、環境変数 `LD_LIBRARY_PATH` がプライベートなシェアド・ライブラリを格納しているディレクトリを指すように設定する。

この環境変数は、プログラムの実行時に実行時ローダによって検証されます。このとおり設定されると、ローダは 4.2.1 項で説明されている一連のディレクトリを探索する前に、`LD_LIBRARY_PATH` によって定義されたパスを探索します。

次のいずれかの方法で、`LD_LIBRARY_PATH` 変数を設定することができます。

- シェル・プロンプトが表示されているときに、環境変数として設定する。

C シェルの場合は、次に示すように、コロンで区切ったパスを指定して `setenv` コマンドを実行します。

```
% setenv LD_LIBRARY_PATH .:$HOME/testdir
```

Korn シェルおよび Bourne シェルの場合は、次に示すように、変数を設定した後エクスポートする必要があります。

```
$ LD_LIBRARY_PATH=.:$HOME/testdir
$ export LD_LIBRARY_PATH
```

これらの例では、ローダが最初に現在のディレクトリを探索し、続いて `$HOME/testdir` ディレクトリを探索するようにパスを設定しています。

- ログイン・ファイルまたはシェル・スタートアップ・ファイルに、変数の定義を追加する。

たとえば、C シェルを使用している場合、次の行を `.login` ファイルまたは `.cshrc` ファイルに追加することができます。

```
setenv LD_LIBRARY_PATH .:$HOME/testdir:/usr/shlib
```

これらの手順で定義されたパスの中で必要とするライブラリをローダが探索できない場合、ローダは 4.2.1 項で説明されている省略時のパスの中のディレクトリを探索します。さらに、`_RLD_ROOT` 環境変数を使用して、実行時ローダの探索パスを変更することもできます。詳細については、`loader(5)` を参照してください。

4.2.3 名前の解決

シンボル名の解決の意味規則は、シンボルが入っているオブジェクト・ファイルまたは共用オブジェクトがリンク・コマンド行に現れる順序に基づいています。通常、リンクは一番左側の定義を解決しなければならないシンボルと見なします。

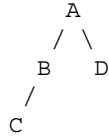
リンク・コマンド行が実行可能ファイルに保存されているかのように、名前が順番に解決されます。プログラムを実行する場合には、実行時にアクセスされるすべてのシンボルが解決されなければなりません。シンボルが解決されないと、ローダはプログラムの実行を打ち切ります。

未解決のシンボルに関してシステムの動作を判断する方法の詳細については、4.2.4 項を参照してください。

次の順序でメイン・プログラムまたはライブラリへのシンボルの参照が解決されます。

1. メイン実行可能ファイルを生成する元になったオブジェクト・ライブラリまたはアーカイブ・ライブラリでシンボルが定義されている場合、そのシンボルはメイン・プログラム・ファイルおよびそれが使用するすべてのシェアード・ライブラリで使用される。
2. シンボルが手順 1 で定義されないで、リンクされている 1 つまたは複数の共用オブジェクトで定義される場合には、定義を指定するリンク・コマンド行の一番左端のライブラリが使用される。
3. リンク・コマンド行に指定したライブラリが他のライブラリに依存するようにリンクされた場合には、ライブラリの依存性は、深さ優先の方法で探索されるのではなく、広さ優先の方法で探索される。

たとえば、次の図のように、実行可能プログラム A がシェアード・ライブラリ B および D にリンクされ、シェアード・ライブラリ B がライブラリ C にリンクされているとします。



この場合、探索順序は A B D C です。幅優先探索では、孫にあたるノードはすべての子ノードが探索された後に探索されます。

4. これまでの手順でシンボルが解決されない場合には、シンボルが解決されないまま残る。

シンボル解決では必ずメイン・オブジェクトのほうが優先されるため、メイン・オブジェクト内で定義されているシンボルにコールバックされるようにシェアード・ライブラリを設定できることに注意してください。さらに、メイン・オブジェクトは、シェアード・ライブラリの定義を指定変更 (強制排除またはフック) するシンボルを定義することができます。

4.2.4 未解決の外部シンボルの処理のオプション

実行可能プログラムを作成する場合とシェアード・ライブラリを作成する場合では、リンカの省略時の動作は次のように異なります。

- 実行可能プログラム — 実行可能プログラムを作成する場合、省略時の設定では未解決のシンボルによりエラーが生成されます (`-error_unresolved` オプション)。
- シェアード・ライブラリ — シェアード・ライブラリを作成する場合、省略時の設定では未解決のシンボルは警告メッセージのみを作成します (`-warning_unresolved` オプション)。

`-error_unresolved` を指定してシェアード・ライブラリを構築すると、未解決のシンボルがある場合でも出力ファイル `.so` ファイル) が生成され、同名の `.so` があればそのファイルは上書きされます。リンカ出力ファイルの上書きについての詳細は『プログラミング・ガイド』を参照してください。

リンカが未解決のシンボルをどのように処理するかは、`ld` コマンドで次のオプションを使用して制御することができます。

- `-expect_unresolved pattern`
- `-warning_unresolved`
- `-error_unresolved`

これらのオプションについての詳細は、`ld(1)` を参照してください。

4.3 シェアード・ライブラリとのリンク

プログラムのコンパイルおよびリンクは、シェアード・ライブラリを使用しても、スタティック・ライブラリを使用しても同じ結果になります。たとえば、次のコマンドは、プログラム `hello.c` をコンパイルして、省略時のシステムの共用 C ライブラリ `libc.so` にリンクします。

```
% cc -o hello hello.c
```

特定の `ld` コマンドのオプションを `cc` コマンドに渡すことによって、シェアード・ライブラリ用の探索パスを自由に指定することができます。たとえば、次に示すように、`cc` コマンドに `-Ldir` オプションを使用して、省略時のディレクトリより先に `dir` を見るように探索パスを変更することができます。

```
% cc -o hello hello.c -L/usr/person -lmylib
```

省略時のディレクトリを探索から除外し、特定のディレクトリおよび特定のライブラリに探索を限定する場合には、次のように指定します。まず、引数を付けずに `-L` オプションを指定し、次に、探索するディレクトリを付けて `-L` オプションを指定し、その後に探索するライブラリ名を付けて `-l` オプションを指定します。たとえば、探索パスを `/usr/person` とプライベートなライブラリ (`libmylib.so`) に限定するには、次のコマンドを入力してください。

```
% cc -o hello hello.c -L -L/usr/person -lmylib
```

`cc` コマンドは常に暗黙に C ライブラリにリンクしているため、前述の例では、`/usr/person` ディレクトリ内に `libc.so` または `libc.a` のコピーが必要であることに注意してください。

4.4 シェアード・ライブラリの指定解除

アプリケーションをリンクする場合は、省略時の設定としてシェアード・ライブラリが使用されます。シェアード・ライブラリを使用しないアプリケーションをリンクする場合には、そのアプリケーションをリンクするときに、`cc` コマンドまたは `ld` コマンドに `-non_shared` オプションを使用しなければなりません。たとえば、次のように入力します。

```
% cc -non_shared -o hello hello.c
```

大部分のアプリケーションではシェアド・ライブラリは省略時の設定ですが、アプリケーションの中にはシェアド・ライブラリを使用できないものがあります。

- シングルユーザ・モードで実行する必要があるアプリケーションは、シェアド・ライブラリとリンクすることができない。

これは、シェアド・ライブラリにアクセスできるようにするためには、`/usr/shlib` ディレクトリをマウントしなければならないためです。

- シングルユーザ・ベンチマークが唯一の用途であるアプリケーションを、シェアド・ライブラリとリンクしてはならない。

4.5 シェアド・ライブラリの作成

シェアド・ライブラリは、`-shared` オプション付きで `ld` コマンドを使用して作成します。オブジェクト・ファイルまたは既存のアーカイブ・ライブラリからシェアド・ライブラリを作成することもできます。

4.5.1 オブジェクト・ファイルからのシェアド・ライブラリの作成

オブジェクト・ファイル `bigmod1.o` および `bigmod2.o` から、シェアド・ライブラリ `libbig.so` を作成するには、次のコマンドを入力します。

```
% ld -shared -no_archive -o libbig.so bigmod1.o bigmod2.o -lc
```

`-no_archive` オプションは、シェアド・ライブラリだけを使用して、すべてのシンボルを解決するようにリンカに指示します。`-lc` オプションは未解決のシンボルがないかどうか、システムの C ライブラリを探索するようにリンカに指示します。

シェアド・ライブラリを `/usr/shlib` ディレクトリにコピーして、システム・レベルでの利用を可能にするには、ルート・ディレクトリに関する特権が必要です。システムで共用するライブラリは、`/usr/shlib` ディレクトリまたは複数ある省略時のディレクトリのうちいずれかに格納し、各ユーザが `LD_LIBRARY_PATH` 変数を省略時のパス以外にあるディレクトリに設定しなくても、実行時ローダ (`/sbin/loader`) がシェアド・ライブラリの場所を探索できるようにしなければなりません。

4.5.2 アーカイブ・ライブラリからのシェアード・ライブラリの作成

ld コマンドを使用して、既存のアーカイブ・ライブラリからシェアード・ライブラリを作成することができます。次に、スタティック・ライブラリ old.a をシェアード・ライブラリ libold.so に変換する例を示します。

```
% ld -shared -no_archive -o libold.so -all old.a -none -lc
```

この例では、-all オプションは、アーカイブ・ライブラリ old.a にあるすべてのオブジェクトをリンクするように、リンクに指示しています。-none オプションは、-all オプションを取り消すようにリンクに指示しています。-no_archive オプションは、-lc オプションの解決には適用されますが、old.a の解決には適用されない点に注意してください (old.a オプションが明示的に記述されているため)。

4.6 プライベートなシェアード・ライブラリの使用

システムのシェアード・ライブラリだけでなく、どのユーザもプライベートなシェアード・ライブラリを作成および使用することができます。たとえば、ある共通のコードを共用する 3 つのアプリケーションがあるとします。これらのアプリケーションは、user、db、および admin という名称であるとします。共用ファイル io_util.c、defines.c、および network.c に定義されているすべてのシンボルが含まれる共通のシェアード・ライブラリ libcommon.so を作成するには、次の手順に従ってください。

1. ライブラリの一部となる各 C ファイルをコンパイルする。

```
% cc -c io_util.c
% cc -c defines.c
% cc -c network.c
```

2. ld コマンドを使用してシェアード・ライブラリ libcommon.so を作成する。

```
% ld -shared -no_archive \
-o libcommon.so io_util.o defines.o network.o -lc
```

3. アプリケーションの一部となる C ファイルをコンパイルする。

```
% cc -c user.c
% cc -o user user.o -L. -lcommon
```

この手順の 2 番目のコマンドは、現在のディレクトリを見てからライブラリ libcommon.so を使用するように、リンクに指示していることに注意してください。

同じ方法で、db.c と admin.c をコンパイルします。

```
% cc -c db.c
% cc -o db db.o -L. -lcommon

% cc -c admin.c
% cc -o admin admin.o -L. -lcommon
```

4. libcommon.so が LD_LIBRARY_PATH で示されているディレクトリ内
にない場合には、libcommon.so をそのディレクトリにコピーする。
5. コンパイル済みのプログラム (user, db, および admin) をそれぞれ
実行する。

4.7 クイックスタートの使用

シェアード・ライブラリの 1 つの利点は、すべての実行イメージをリンクした後にライブラリを変更できるため、ライブラリのバグを修正できることです。この機能は、製品の開発期間中に非常に役立ちます。

しかし、製品出荷サイクルでは、通常、開発および修正したシェアード・ライブラリとアプリケーションは、次のリリースまで変更されません。その場合には、クイックスタートを活用することができます。これは、ユーザのプログラムとライブラリにあるすべてのシンボルに、あらかじめアドレスを設定する方法です。

アプリケーションをクイックスタートさせるための特別なリンク・オプションは必要ありませんが、満たさなければならない一連の条件があります。クイックスタートできない場合でもオブジェクトを実行することはできますが、スタートアップ時間は長くなります。

リンカが共用オブジェクト、すなわちシェアード・ライブラリまたはシェアード・ライブラリを使用するメイン実行可能ファイルを作成する場合には、オブジェクトのテキストおよびデータ部分にアドレスを割り当てます。このアドレスが、いわゆるクイックスタート・アドレスと呼ばれているものです。オブジェクトがクイックスタート・アドレスでロードされるかのように、リンカは事前にすべての動的再配置を実行します。

依存するすべてのオブジェクトは、クイックスタート用アドレスにあるとみなされます。元のオブジェクトから依存先のオブジェクトを参照すると、それに応じて依存先のオブジェクトのアドレスが設定されます。

クイックスタートを使用するためには、オブジェクトは次の条件を満たしていなければなりません。

- オブジェクトの実行時の実際のメモリ位置は、クイックスタート・アドレスと一致しなければならない。

実行時ローダがクイックスタート・アドレスを使用しようとしても、すでに別のライブラリがそのアドレスを占有している場合には使用できません。

- 依存するすべてのオブジェクトは、クイックスタートしなければならない。
- 依存するすべてのオブジェクトは、リンク後に変更があってはならない。
変更があった場合には、ライブラリ内の関数のアドレスが移動し、新しいシンボルの追加によってリンクに影響する可能性があります。
リンク後に変更されたオブジェクトに対して `fixso` ユーティリティを実行することによって、そのオブジェクトをクイックスタートできるようになります。詳細については、`fixso(1)` を参照してください。

オペレーティング・システムは、これらの条件を、チェックサムとタイムスタンプによって確認します。

ライブラリを作成する場合、それらにはクイックスタート・アドレスが与えられます。クイックスタートの制約条件を満たすためには、アプリケーションが使用するすべてのライブラリのクイックスタート・アドレスをそれぞれ一意な値に設定する必要があります。アプリケーション側でアドレスの心配をするよりも、作成した各シェアード・ライブラリに一意なクイックスタート・アドレスを割り当て、すべてのオブジェクトをクイックスタート・アドレスでロードできるようにしてください。

リンクは、ライブラリ作成時にクイックスタート・アドレスが登録される `so_locations` データベースを保守します。新しいライブラリに対してクイックスタート・アドレスを選択する際、リンクは、すでに `so_locations` ファイルに存在するアドレスはクイックスタート・アドレスとして使用しません。

省略時の設定では、`-update_registry ./so_locations` オプションが選択されたものとして、`ld` が実行されます。この結果、作成したライブラリのディレクトリ内の `so_locations` ファイルが、必要に応じて更新されるかまたは作成されます。

ライブラリがユーザのシステムのシェアード・ライブラリと競合しないよう、次のコマンドを入力してください。

```
% cd <directory_of_build>
% cp /usr/shlib/so_locations .
% chmod +w so_locations
```

ここまでの手順により、ライブラリを作成することができます。複数のディレクトリにライブラリを作成した場合には、ld コマンドを `-update_registry` オプションとともに使用して、共通の `so_locations` ファイルの位置を明示的に指定します。たとえば、次のように入力します。

```
% ld -shared -update_registry /common/directory/so_locations ...
```

システムのすべてのユーザのために、グローバルにシェアード・ライブラリをインストールする場合には、システム全体で使用する `so_locations` ファイルを更新します。 `shared_library.so` に実際のシェアード・ライブラリを指定して、ルート・アカウントで次のコマンドを入力します。

```
# cp shared_library.so /usr/shlib
# mv /usr/shlib/so_locations /usr/shlib/so_locations.old
# cp so_locations /usr/shlib
```

複数の人がシェアード・ライブラリを作成している場合には、他の共用データベースと同様に、共通の `so_locations` ファイルを管理する必要があります。任意のプロセスで使用される各シェアード・ライブラリには、ファイルに一意のクイックスタート・アドレスを指定しなければなりません。リンカがメイン実行可能ファイルに割り当てる省略時の開始アドレスの範囲は、共用オブジェクトに対して作成するクイックスタート・アドレスとは競合しません。1つのメイン実行可能ファイルだけがプロセスにロードされるので、メインの実行可能ファイルとその共用オブジェクトの間でアドレスの競合は起こりません。

既存のシェアード・ライブラリを使用しているだけで、ユーザ独自のライブラリを使用しない場合には、特に何もする必要はありません。ライブラリが前に説明した条件に合っている限り、ライブラリ自身をクイックスタートしない場合を除いて、ユーザのプログラムはクイックスタートされます。オペレーティング・システムに付属するほとんどのライブラリは、クイックスタートするようになっています。

新規にシェアード・ライブラリを作成している場合には、すでに説明しているように、最初に `so_locations` ファイルをコピーしなければなりません。次に、`so_locations` ファイルを使用して、すべてのシェアード・ライブラ

りをボトムアップ式の順番で作成しなければなりません。依存するすべてのライブラリをリンク行に指定してください。すべてのライブラリを作成した後に、ユーザのアプリケーションを作成することができます。

4.7.1 オブジェクトのクイックスタートの確認

アプリケーションの実行可能プログラムがクイックスタートであるかどうかを調べるには、`_RLD_ARGS` 環境変数に `-quickstart_only` を設定して、プログラムを実行します。たとえば、次のように設定します。

```
% setenv _RLD_ARGS -quickstart_only
% foo
```

クイックスタートでない場合には、次のような出力になります。

```
21887:foo: /sbin/loader: Fatal Error: NON-QUICKSTART detected \
-- QUICKSTART must be enforced
```

プログラムが正常に終了すればクイックスタートです。ロード・エラー・メッセージが表示された場合には、そのプログラムはクイックスタートではありません。

4.7.2 手動によるクイックスタート問題の解決

実行可能プログラムがクイックスタートでない原因を判断するには、4.7.3 項で説明する `fixso` ユーティリティを使用するか、あるいは次の条件を調べてみてください。 `fixso` を使用すると簡単ですが、ここで説明する手順を理解しておくことも役に立ちます。

1. 実行可能プログラムはクイックスタート可能でなければならない。

動的ヘッダのクイックスタート・オプションを調べてください。クイックスタート・オプションの値は `0x00000001` です。

```
% odump -D foo | grep FLAGS
```

クイックスタートでない場合の出力は、次のようになります。

```
FLAGS: 0x00000000
```

クイックスタートの場合の出力は、次のようになります。

```
FLAGS: 0x00000001
```

クイックスタート・オプションが設定されていない場合には、次のことが考えられます。

- 実行可能プログラムが未解決のシンボルにリンクされている。

実行可能プログラムをリンクするときに、ld オプションの `-warning_unresolved` と `-expect_unresolved` が使用されていないことを確認してください。実行可能プログラムのリンク時に生じるすべての未解決シンボル・エラーを修正してください。

- 実行可能プログラムが、実行時に使用するすべてのライブラリに直接リンクされていないかった。

実行可能プログラムを作成するときに、使用している ld オプションに `-transitive_link` オプションを追加してください。

2. 実行可能プログラムの依存するライブラリはクイックスタートでなければならない。

実行可能プログラムの依存するライブラリのリストを表示するには、次のコマンドを実行します。

```
%odump -Dl foo
```

クイックスタートの場合の出力は、次のようになります。

```
***LIBRARY LIST SECTION***
Name                Time-Stamp          CheckSum    Flags Version
foo:
  libX11.so          Sep 17 00:51:19 1993 0x78c81c78  NONE
  libc.so            Sep 16 22:29:50 1993 0xba22309c  NONE osf.1
  libdnet_stub.so    Sep 16 22:56:51 1993 0x1d568a0c  NONE osf.1
```

各依存ライブラリの動的ヘッダのクイックスタート・フラグを調べます。

```
% cd /usr/shlib
```

```
% odump -D libX11.so libc.so libdnet_stub.so | grep FLAGS
```

クイックスタートの場合には、次のような出力になります。

```
FLAGS: 0x00000001
FLAGS: 0x00000001
FLAGS: 0x00000001
```

依存ライブラリのいずれかがクイックスタートできないとき、シェアード・ライブラリをユーザが作成し直せる場合には、手順 1 で説明した方法で解決してください。

3. すべての依存ライブラリのタイムスタンプとチェックサム情報が一致していないとなければならない。

手順 2 の依存ライブラリ・リストには、foo の各依存ライブラリのタイムスタンプとチェックサムのフィールドに期待値が表示されています。これらの値と、各ライブラリの現在の値を照合するには、次のようにします。

```
% cd /usr/shlib
% odump -D libX11.so libc.so libdnet_stub.so | \
grep TIME_STAMP
```

クイックスタートの場合は、次のような出力になります。

```
TIME_STAMP: (0x2c994247) Fri Sep 17 00:51:19 1993
TIME_STAMP: (0x2c99211e) Thu Sep 16 22:29:50 1993
TIME_STAMP: (0x2c992773) Thu Sep 16 22:56:51 1993
```

```
% odump -D libX11.so libc.so libdnet_stub.so | grep CHECKSUM
```

クイックスタートの場合には、次のような出力になります。

```
ICHECKSUM: 0x78c81c78
ICHECKSUM: 0xba22309c
ICHECKSUM: 0x1d568a0c
```

どちらかのテストで、タイムスタンプまたはチェックサムの不一致があった場合には、プログラムを再リンクすることにより問題は解決されます。

バージョン・フィールドを使用すると、実行時に正しいライブラリがロードされたことを確認することができます。依存ライブラリのバージョンを調べるには、次の例に示すように `odump` コマンドを使用します。

```
% odump -D libX11.so | grep IVERSION
% odump -D libc.so | grep IVERSION
IVERSION: osf.1
% odump -D libdnet_stub.so | grep IVERSION
IVERSION: osf.1
```

依存情報のエントリが空白の場合には、一致する `IVERSION` エントリはありません。また、特殊なバージョン `_null` に一致するエントリもありません。

バージョンの不一致が見つかった場合には、通常は、依存リストのバージョン識別子を追加するか、または `_null` をパス `/usr/shlib` に追加することによって、シェアード・ライブラリの正しいバージョンを見つけることができます。

4. 実行可能プログラムの各依存ライブラリも、一致するタイムスタンプとチェックサム情報を示す依存リストを含んでいなければならない。

実行可能プログラムの依存リストに表示されている各シェアード・ライブラリに対して、手順 3 を繰り返して実行します。

```
% odump -D1 libX11.so
```

クイックスタートの場合には、次のような出力になります。

```

***LIBRARY LIST SECTION***
Name                Time-Stamp                CheckSum    Flags Version
libX11.so:
  libdnet_stub.so Sep 16 22:56:51 1993 0x1d568a0c  NONE osf.1
  libc.so          Sep 16 22:29:50 1993 0xba22309c  NONE osf.1
% odump -D libdnet_stub.so libc.so | grep TIME_STAMP
TIME_STAMP: (0x2c992773) Thu Sep 16 22:56:51 1993
TIME_STAMP: (0x2c99211e) Thu Sep 16 22:29:50 1993
% odump -D libdnet_stub.so libc.so | grep CHECKSUM
ICHECKSUM: 0x1d568a0c
ICHECKSUM: 0xba22309c

```

タイムスタンプまたはチェックサム情報が一致していない場合には、シェアード・ライブラリを再作成して問題を解決しなければなりません。シェアード・ライブラリを再作成すると、タイムスタンプが変更され、場合によってはチェックサムも変更されます。依存ライブラリをボトムアップ式に再作成してから、実行可能プログラムやシェアード・ライブラリを再作成します。

4.7.3 fixso ユーティリティによるクイックスタート問題の解決

fixso ユーティリティは、タイムスタンプおよびチェックサムの不一致が原因で発生するクイックスタートの問題を識別し修復します。このユーティリティは、プログラムが依存するシェアード・ライブラリと同様に、プログラム自体も修復できます。ただし、シンボルの変更が必要となるようなプログラムの修復はできません。

fixso ユーティリティは、次に示す制限に該当するプログラムあるいはシェアード・ライブラリは修復できません。

- プログラムあるいはシェアード・ライブラリがクイックスタートできない他のシェアード・ライブラリに依存している場合。

この制限は、シェアード・ライブラリに対して逆の順序で fixso を使用することによって回避することができます。

- プログラムあるいはシェアード・ライブラリを作成した後に新しい名前の矛盾が発生した場合。

名前の矛盾は、2 つ以上の依存するシェアード・ライブラリ、あるいはプログラムおよびそれが依存するシェアード・ライブラリによって同じグローバル・シンボル名がエクスポートされている場合に発生します。

- プログラムが依存するシェアード・ライブラリのいずれかが、それらのクイックスタート位置にロードされない場合。

同じクイックスタート位置に他のシェアード・ライブラリがロードされ、既に使用されている場合は、シェアード・ライブラリをクイックスタート位置にロードできません。この規則は個々のプロセスに対してだけでなく、システム全体に適用されます。この制限を回避するには、シェアード・ライブラリに対してユニークなアドレスを登録するための共通 `so_locations` ファイルを使用します。

- プログラムあるいはシェアード・ライブラリが、互換性のないバージョンのシェアード・ライブラリに依存している場合。

この制限は、互換性のあるバージョンのシェアード・ライブラリを探すよう `fixso` に指示することによって回避できます。

`fixso` ユーティリティを使用すると、次の例に示すような形でクイックスタート問題を識別することができます。

```
% fixso -n hello.so
fixso: Warning: found '/usr/shlib/libc.so' (0x2d93b353) which does
      not match timestamp 0x2d6ae076 in liblist of hello.so, will fix
fixso: Warning: found '/usr/shlib/libc.so' (0xc777ff16) which does
      not match checksum 0x70e62eeb in liblist of hello.so, will fix
```

`-n` オプションを指定すると出力ファイルの生成を行いません。この例では、不一致が報告されていますが問題の修復は行われていません。次の例では、`fixso` を使ってクイックスタート問題を修復する方法を説明します。

```
% fixso -o ./fixed/main main
fixso: Warning: found '/usr/shlib/libc.so' (0x2d93b353) which does
      not match timestamp 0x2d7149c9 in liblist of main, will fix
% chmod +x fixed/main
```

`-o` オプションは出力ファイルを指定します。出力ファイル名が指定されない場合は `a.out` が使用されます。`fixso` が作成する出力ファイルは、ファイル保護コードが実行可能ではないことに注意してください。`chmod` コマンドで出力ファイルの保護コードを変更しています。この変更処理は、実行可能プログラムに対してのみ必要になります。`fixso` を使用してシェアード・ライブラリを修復する場合は省略できます。

プログラムあるいはシェアード・ライブラリの修復が不要な場合、`fixso` は、次の例に示すように指摘します。

```
% fixso -n /bin/ls
no fixup needed for /bin/ls
```

4.8 シェアード・ライブラリとリンクしているプログラムのデバッグ

シェアード・ライブラリを使用しているプログラムをデバッグする方法は、アーカイブ・ライブラリを使用しているプログラムをデバッグする方法と基本的には同じです。

dbx デバッガの `listobj` コマンドは、実行プログラムの名前とデバッガが認識できるすべてのシェアード・ライブラリを表示します。dbx についての詳細は、第 5 章を参照してください。

4.9 シェアード・ライブラリの実行時のロード

プログラムからシェアード・ライブラリをロードする場合があります。この節では、2 つの短い C プログラムと `makefile` を例に、実行時にシェアード・ライブラリをロードする方法を示します。

簡単なメッセージを表示する C プログラムのソース・ファイルの例 (`pr.c`) を、次に示します。

```
printmsg()
{
    printf("Hello world from printmsg!\n");
}
```

シンボルを定義し、`dlopen` 関数を使用する方法を示す C プログラムのソース・ファイルの例 (`used1.c`) を、次に示します。

```
#include <stdio.h>
#include <dlfcn.h>

/* dl* ルーチンからのエラーはすべて NULL で戻される。*/
#define BAD(x) ((x) == NULL)

main(int argc, char*argv[])
{
    void *handle;
    void (*fp)();

    /* ./ 接頭語を使用すると、dlopen は現在の
     * ディレクトリを探索する。それ以外の場合には、
     * LD_LIBRARY_PATH などを使用する。
     */
    handle = dlopen("./pr.so", RTLD_LAZY);
    if (!BAD(handle)) {
        fp = dlsym(handle, "printmsg");
        if (!BAD(fp)) {
```

```

        /*
        *   ここで、探索していた関数が呼び出される。
        */
        (*fp)();
    }
    else {
        perror("dlsym");
        fprintf(stderr, "%s\n", dlerror());
    }
}
else {
    perror("dlopen");
    fprintf(stderr, "%s\n", dlerror());
}
}
dlclose(handle);
}

```

次に、pr.o、pr.so、so_locations、usedl.o を生成する makefile を示します。

これは例を検証する makefile である。

```

all: runit

runit: usedl pr.so
      ./usedl

usedl: usedl.c
      $(CC) -o usedl usedl.c

pr.so: pr.o
      $(LD) -o pr.so -shared pr.o -lc

```

4.10 シェアード・ライブラリ・ファイルの保護

シェアード・ライブラリに共用のためのメカニズムが使用されているため、通常のファイル・システムのための保護機能では、不正な読み取りを防ぐことができません。たとえば、シェアード・ライブラリがプログラムの中で使用されている場合、そのライブラリのテキスト部分は、次に示すような状況においても他のプロセスから読み取ることができます。

- ライブラリの保護コードが 600 に指定されている
- ライブラリの所有者ではない、あるいはライブラリ所有者に設定されている UID を使用していない

この場合、テキスト部分だけが共用され、データ・セグメントは共用されません。

必要以外の共用を行わないために、保護する必要のあるシェアード・ライブラリは `-T` オプションや `-D` オプションを使用してリンクし、データ・セクションを次のセクションと同じ 8 MB のセグメントに格納してください。たとえば、次の例のようなコマンドを入力してください。

```
% ld -shared -o libfoo.so -T 30000000000 \
-D 30000400000 object_files
```

さらに、マップされたアドレスが `mmap` を使用している他のファイルと同じメモリ・セグメントを参照している限り、`PROT_WRITE` オプションを使用せずに `mmap` システム・コールを使用しているどのようなファイルにもセグメントの共用が発生する可能性があります。

`mmap` を使用して厳重に保護されている可能性のあるファイルを検査するプログラムを使用すると、`mmap` の前またはその途中セグメントに書き込み可能なページを挿入して、セグメントの共用を防ぐことができます。シェアード・ライブラリを最も簡単に保護するには、保護機能の実行の際に使用可能になる `PROT_WRITE` が使用されているファイルで `mmap` システム・コールを使用し、`mprotect` システム・コールを使用してマップされたメモリを読み取りのみ可能にしてください。別の方法として、すべてのセグメント化を使用不能にし、権限のない共用を防ぐには、構成ファイルに次の行を入力します。

```
segmentation 0
```

4.11 シェアード・ライブラリのバージョン管理

シェアード・ライブラリを使用することの利点の 1 つは、シェアード・ライブラリが変更されても、そのライブラリとリンクしているプログラムを再作成する必要がないということです。変更されたライブラリがインストールされると、アプリケーションは古いライブラリを使用していたときと同様に新しいライブラリを使用して動作します。

注意

新しいバージョンのシェアード・ライブラリがインストールされると、古いバージョンのシェアード・ライブラリを使用する既存のアプリケーションは、追加アドレスの決定が必要になるため、ロード時間が長くなる場合があります。アプリケーションを新しいライブラリにリンクし直すと、このようなアドレスの決定を避けて、ロード時間を短縮することができます。

4.11.1 バイナリ非互換修正

まれに、シェアード・ライブラリを変更すると、そのライブラリと変更前にリンクしていたアプリケーションとの互換性がなくなることがあります。この種の変更をバイナリ非互換といいます。シェアード・ライブラリの新しいバージョンでバイナリ非互換が起こっても、古いバージョンのライブラリに依存するアプリケーションは、必ずしもエラーになる（つまり、ライブラリの下位互換性に違反する）わけではありません。システムではシェアード・ライブラリのバージョン管理を行うことによって、ライブラリでバイナリ非互換が起こった場合に、シェアード・ライブラリの下位互換性を維持する処置がとれるようにしています。

シェアード・ライブラリで非互換を起こす修正には、次のような種類があります。

- 文書化されているインタフェースの削除

たとえば、`libc.so` の `malloc()` 関数を (`_malloc`) という関数と置き換えると、古い関数に依存するプログラムは、シンボル `malloc` がないために異常終了します。

- 文書化されているインタフェースの修正

たとえば、`libc.so` の `malloc()` 関数に 2 番目の引数を追加すると、古い関数に依存するプログラムが、2 番目の引数を未定義の値のままにして、1 つの引数だけに値を引き渡したときに、新しい `malloc()` は失敗する可能性が高くなります。

- グローバル・データ定義の修正

たとえば、`libc.so` のシンボル `errno` の型を `int` から `long` に変更すると、古いライブラリとリンクされていたプログラムは、新しく拡張された 64 ビットのデータ項目との間で 32 ビット値の読み取りや書き込みを行います。この場合には、無効なエラー・コードと不定なプログラム動作が生じます。

もちろん、これだけがバイナリ非互換を引き起こす変更のすべてではありません。シェアード・ライブラリの開発者は常識を働かせて、どのような変更を行うと、変更前のライブラリとリンクしていたアプリケーションに障害が生じるかを判断してください。

4.11.2 シェアード・ライブラリのバージョン管理

非互換変更の影響を受けたシェアード・ライブラリの下位互換性は、ライブラリの複数バージョンを使用することにより維持できます。各シェアード・ライブラリはバージョン識別子によってマークされます。ライブラリの新しいバージョンを省略時のディレクトリにインストールし、そのライブラリのバージョン識別子と一致する名前を持つ、古いバイナリ互換バージョンをサブディレクトリにインストールします。

たとえば、`libc.so` に非互換の変更が行われた場合、新しいライブラリ (`/usr/shlib/libc.so`) は、変更前のライブラリのインスタンス (`/usr/shlib/osf.1/libc.so`) を伴っていなければなりません。

この例では、`libc.so` の古いバイナリ互換バージョンは `osf.1` バージョンです。変更が適用されると、新しい `libc.so` が新しいバージョン識別子を使用して作成されます。シェアード・ライブラリのバージョン識別子は、そのライブラリを使用するプログラムのシェアード・ライブラリ従属レコードにリストされているため、ローダはアプリケーションで必要とするシェアード・ライブラリのバージョンを識別できます (4.11.6 項を参照)。

前述の例では、バイナリ非互換変更の前に `libc.so` を使用して作成したプログラムは、ライブラリの `osf.1` バージョンを必要とします。`/usr/shlib/libc.so` のバージョンはプログラムのシェアード・ライブラリ従属レコードにリストされているバージョン識別子と一致しないため、ローダは `/usr/shlib/osf.1` で一致するバージョンを探します。

非互換変更後に作成されるアプリケーションは、`/usr/shlib/libc.so` を使用して作成され、ライブラリの新しいバージョンに依存します。ローダは、別のバイナリ非互換変更が起こるまで、`/usr/shlib/libc.so` を使用してこれらのアプリケーションをロードします。

表 4-1 は、シェアード・ライブラリのバージョン管理を有効にするためのリンカ・オプションの説明です。

表 4-1: シェアード・ライブラリのバージョンを管理するリンカ・オプション

オプション	説明
<code>-set_version version-string</code>	<p>シェアード・ライブラリに関連するバージョン識別子を設定する。文字列 <i>version-string</i> は、単一のバージョン識別子またはコロンで区切ったバージョン識別子のリストである。バージョン識別子の名前に関する制約はないが、UNIX のディレクトリ命名規則に従うのがよい。</p> <p>シェアード・ライブラリがこのオプションを使用して作成されると、そのライブラリに対して作成されたプログラムは、指定されたバージョン、またはバージョン識別子のリストが指定されている場合には、そのリストに指定されている一番右端のバージョンの従属が記録される。シェアード・ライブラリがバージョン識別子のリストを使用して作成されると、実行時ローダは、リストされている任意のバージョンに関するシェアード・ライブラリの従属を持つプログラムはどれでも実行できるようにする。</p> <p>このオプションはシェアード・ライブラリの作成時のみに (<code>-shared</code> を指定して) 使用できる。</p>
<code>-exact_version</code>	<p><code>ld</code> コマンドで作成された動的オブジェクトにオプションを設定する。このコマンドを使用すると、実行時ローダは確実に、オブジェクトが実行時に使用するシェアード・ライブラリが、リンク時に使用されたシェアード・ライブラリと一致するようにさせる。</p> <p>このオプションは、動的実行可能ファイル (<code>-call_shared</code> を指定) またはシェアード・ライブラリ (<code>-shared</code> を指定) の作成時に使用される。このオプションを使用した場合には、シェアード・ライブラリの従属のより厳密なテストを行う必要がある。シェアード・ライブラリのバージョンが一致することをテストするだけでなく、タイムスタンプとチェックサムも、リンク時におけるシェアード・ライブラリ従属レコードに記録されているタイムスタンプおよびチェックサムと一致することをテストしなければならない。</p>

odump コマンドを使用すると、シェアード・ライブラリのバージョン文字列を確認することができます。この文字列は、そのライブラリを作成した ld コマンドの `-set_version version-string` オプションで設定されるものです。次のように入力します。

```
% odump -D library-name
```

IVERSION フィールドに表示される値が、ライブラリの作成時に指定されたバージョン文字列です。 `-set_version` オプションを指定しないでライブラリを作成した場合には、IVERSION フィールドは表示されません。このようなシェアード・ライブラリは、バージョン識別子として `_null` が指定された場合と同様に扱われます。

ld が共用オブジェクトをリンクするときには、各シェアード・ライブラリの従属のバージョンを記録します。コロンで区切ったリストの一番右端にあるバージョン識別子だけが記録されます。任意の共用されている実行可能ファイルまたはライブラリのこれらの依存関係を確認するには、次のコマンドを使用します。

```
% odump -Dl shared-object-name
```

4.11.3 メジャーおよびマイナー・バージョン識別子

Tru64 UNIX では、シェアード・ライブラリのメジャー・バージョンとマイナー・バージョンを区別しません。

- メジャー・バージョンはシェアード・ライブラリの非互換バージョンを区別するために使用されます。
- マイナー・バージョンは通常、異なっているけれど互換性のあるライブラリ・バージョンを区別します。マイナー・バージョンは、修正版用の識別子として使用されたり、下位互換のシェアード・ライブラリの使用を制限するために使用されます。

Tru64 UNIX のシェアード・ライブラリは、コロンで区切ったバージョン識別子のリストを使用することによって、通常はマイナー・バージョンを使用して行うバージョン管理機能を提供しています。

次に示すライブラリ・バージョンのシーケンスは、シェアード・ライブラリの互換性に影響を与えることなく、修正版用の識別子をシェアード・ライブラリのバージョン・リストに追加する方法を示しています。

シェアード・ライブラリ	バージョン
libminor.so	3.0
libminor.so	3.1:3.0
libminor.so	3.2:3.1:3.0

libminor.so の新しいリリースはそれぞれ、バージョン・リストの先頭に新しい識別子を追加します。この新しい識別子により、前のバージョンと最新バージョンを区別します。libminor.so の任意のバージョンとリンクする実行可能ファイルは、必須バージョンとして 3.0 を記録するため、互換ライブラリと区別が付きません。追加のバージョン識別子だけが情報を提供します。

次のライブラリ・バージョンのシーケンスは、下位互換のシェアード・ライブラリの使用を制限する方法を示しています。

シェアード・ライブラリ	バージョン
libminor2.so	3.0
libminor2.so	3.0:3.1
libminor2.so	3.0:3.1:3.2

この例では、libminor2.so の古いバージョンとリンクされるプログラムは、ライブラリの新しいバージョンで実行できますが、libminor2.so の新しいバージョンとリンクされるプログラムは、古いバージョンでは実行できません。

4.11.4 シェアード・ライブラリの完全バージョンと部分バージョン

シェアード・ライブラリのバイナリ互換バージョンは、2 つの方法でインプリメントできます。つまり、完全に独立したオブジェクトと、直接または間接的に完全に独立したオブジェクトに依存する部分オブジェクトとしてインプリメントできます。完全に重複したシェアード・ライブラリは部分バージョンより多くのディスク・スペースを必要としますが、従属処理が単純で、使用するスワップ領域が少なくてすみます。使用するディスク・スペースを削減できることが、シェアード・ライブラリの部分バージョンの唯一の利点です。

部分シェアード・ライブラリは、ライブラリの新しいバージョンでバイナリ非互換変更が行われる前にリンクされたアプリケーションに対して、下位互

換を維持するために最低限必要なモジュールのサブセットを含みます。これは、ライブラリ・モジュールの完全なセットを持つ同じライブラリの1つまたは複数の以前のバージョンとリンクしています。このように、シェアード・ライブラリの複数のバージョンを連結すると、シェアード・ライブラリの任意のインスタンスは、通常はライブラリでエクスポートされるすべてのシンボルを、間接的に提供できるようになります。

たとえば、`libxyz.so` のバージョン `osf.1` には、モジュール `x.o`、`y.o`、`z.o` が含まれています。これは、次のコマンドを使用して作成され、インストールされました。

```
% ld -shared -o libxyz.so -set_version osf.1 \
    x.o y.o z.o -lc
% mv libxyz.so /usr/shlib/libxyz.so
```

将来、`libxyz.so` において、モジュール `z.o` だけに影響を及ぼす非互換変更が必要になった場合には、`osf.2` と呼ばれる新しいバージョンと、`osf.1` と呼ばれる部分バージョンは、次のように作成することができます。

```
% ld -shared -o libxyz.so -set_version osf.2 x.o \
    y.o new_z.o -lc
% mv libxyz.so /usr/shlib/libxyz.so
% ld -shared -o libxyz.so -set_version osf.1 z.o \
    -lxyz -lc
% mv libxyz.so /usr/shlib/osf.1/libxyz.so
```

4.11.5 シェアード・ライブラリの複数バージョンとのリンク

一般に、アプリケーションはシェアード・ライブラリの最新バージョンとリンクされます。しかし、アプリケーションやシェアード・ライブラリを、シェアード・ライブラリの古い、バイナリ互換バージョンとリンクしたいことがあります。そのような場合には、`ld` コマンドの `-L` オプションを使用して、アプリケーションが使用するシェアード・ライブラリの古いバージョンを識別します。

アプリケーションを同じシェアード・ライブラリの複数のバージョンとリンクすると、リンクは警告メッセージを出します。アプリケーションまたはシェアード・ライブラリの複数バージョンの従属は、実行のためにロードされるまで通知されない場合があります。

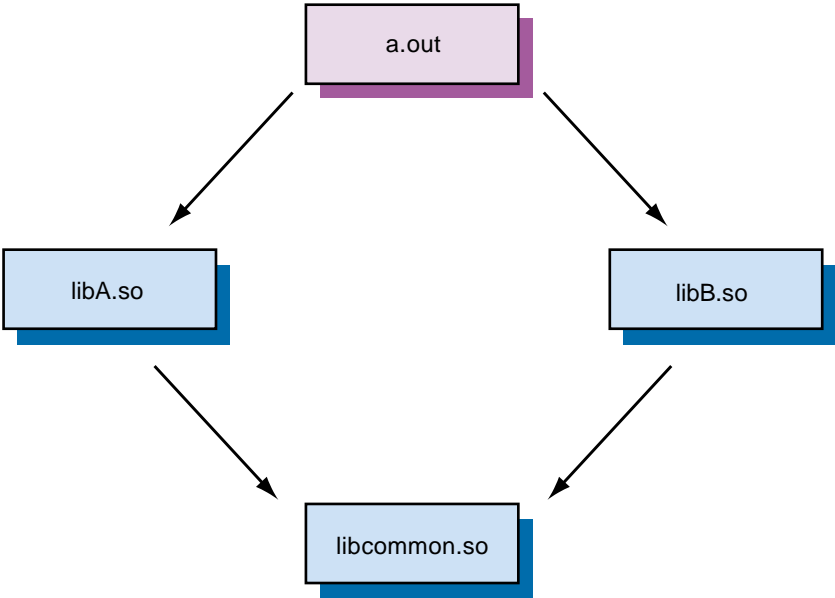
`ld` コマンドは、省略時には、リンクするように指示されているライブラリのみ複数のバージョンの従属を調べます。考えられるすべての複数バージョン

の従属を識別するには、ld コマンドの `-transitive_link` オプションを使用して、リンクのステップに間接シェアード・ライブラリの従属を含めます。

アプリケーションを部分シェアード・ライブラリとリンクする場合には、部分シェアード・ライブラリのインプリメンテーションで生じた複数バージョンの従属を注意深く区別しなければなりません。リンクは、受け入れ可能な複数バージョンの従属と受け入れ不可能なものの区別ができない場合、複数バージョンの警告メッセージを表示します。

場合によっては、実行時にシェアード・ライブラリの複数バージョンを使用しないアプリケーションに対し、リンク時に複数バージョンの従属が通知されます。図 4-2 および次の表に示すライブラリと従属について考えてみてください。

図 4-2: シェアード・ライブラリの複数バージョンとのリンク



ZK-0882U-AI

ライブラリ	バージョン	従属	依存するバージョン
libA.so	v1	libcommon.so	v1
libB.so	v2	libcommon.so	v2
libcommon.so	v1:v2	—	—

この表で、libA.so は、右端のバージョン識別子に v1 を含む libcommon.so にリンクされています。libB.so は、右端のバージョン識別子に v2 を含む libcommon.so にリンクされています。この表に示す libcommon.so は、バージョン文字列に v1 と v2 の両方を含むため、libA.so および libB.so の従属はともに libcommon.so という 1 つのインスタンスで満足できます。

a.out がリンクされる時、libA.so および libB.so だけがリンクのコマンド行に指定されます。ただし、ld は libA.so および libB.so の従属を調べ、libcommon.so の考えられる複数バージョンの従属を承認して警告メッセージを出します。a.out を libcommon.so にもリンクすると、この間違っただ警告を回避することができます。

4.11.6 ロード時におけるバージョン・チェック

ローダは、シェアード・ライブラリがサポートするバージョン・リストと、シェアード・ライブラリの従属レコードに記録されているバージョンの照合を実行します。共用オブジェクトがリンクのコマンド行で `-exact_match` オプションを使用してリンクされている場合には、ローダはシェアード・ライブラリのタイムスタンプとチェックサムも従属レコードに記録されている値と比較します。

ローダはバージョン照合検査が失敗したシェアード・ライブラリでマップした後、`RPATH`、`LD_LIBRARY_PATH`、または省略時の探索パスに指定されている他のディレクトリの探索を続行して、シェアード・ライブラリの正しいバージョンを見つけようとします。

これらのディレクトリをすべて探索しても見つからない場合には、ローダは、従属に記録されているバージョン文字列を、最初に一致しないライブラリのバージョンが見つかったディレクトリ・パスに付加して、一致するバージョンを探索しようとします。

たとえば、シェアード・ライブラリ `libfoo.so` はバージョン `osf.2` を使用してディレクトリ `/usr/local/lib` にロードされますが、このライブラリの従属はバージョン `osf.1` を必要とします。ローダは組み合わせた次のパスを使用して、ライブラリの正しいバージョンの探索を試みます。

```
/usr/local/lib/osf.1/libfoo.so
```

この組み合わせたパスでも正しいライブラリの探索に失敗した場合、あるいは、省略時の探索ディレクトリまたはユーザ指定の探索ディレクトリでもライブラリのバージョンが探索できなかった場合、ローダは、必須のバー

ジョン文字列を標準システムのシェアード・ライブラリのディレクトリ (/usr/shlib) に付加して、最後の探索を試みます。

前述の例を使用すると、libfoo.so を探索するローダの最後の試みでは、次のような組み合わせたパスを使用します。

```
/usr/shlib/osf.1/libfoo.so
```

ローダがシェアード・ライブラリの一致するバージョンを見つけられない場合、ロードは打ち切られて、探索できなかった従属とシェアード・ライブラリのバージョンを示す詳細なエラー・メッセージが報告されます。

setuid 関数を使用してインストールされなかったプログラムのバージョン・チェックは、次の C シェルの例に示すように、ローダの環境変数を設定することにより、禁止することができます。

```
% setenv _RLD_ARGS -ignore_all_versions
```

次の例に示すように、特定のシェアード・ライブラリに対するバージョン・チェックを禁止することも可能です。

```
% setenv _RLD_ARGS -ignore_version libDXm.so
```

4.11.7 ロード時における複数バージョンのチェック

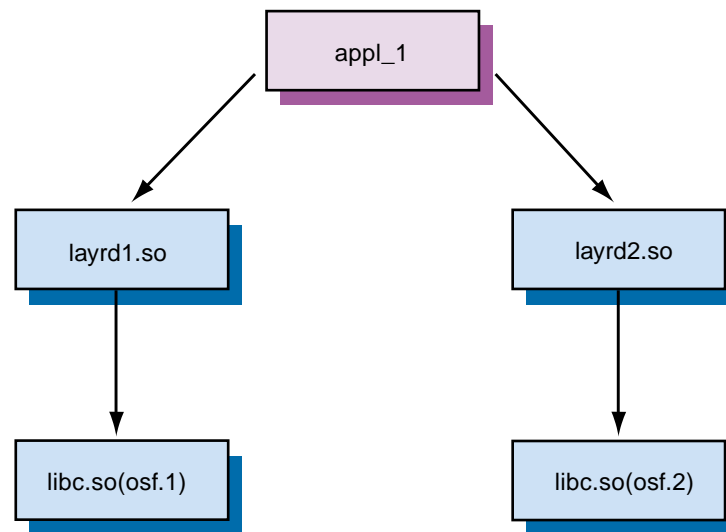
リンカと同様、ローダもシェアード・ライブラリの複数バージョンの有効な使用と無効な使用を区別しなければなりません。

- 複数バージョンの有効な使用は、同じライブラリの他のバージョンに依存する部分シェアード・ライブラリがロードされるときに起こります。場合によっては、これらの部分シェアード・ライブラリは異なる部分シェアード・ライブラリに依存しているため、ローダは誤ったエラーを報告しないように注意深く解釈しなければならないような、複雑な従属図式になることがあります。
- 複数バージョンの無効な使用は、2 つの異なる共用オブジェクトが別の共用オブジェクトの異なるバージョンに依存する場合に起こります。部分シェアード・ライブラリの連鎖はこのルールの例外です。バージョン・チェックのため、連結の最初の部分シェアード・ライブラリは、その連結内の他のメンバの同じ従属を上書きする 1 組の従属を定義します。

次の図は、複数従属エラーになる共用オブジェクトの従属図式を示しています。バージョン識別子はカッコ内に示しています。

図 4-3 では、アプリケーションは、基本システムの非互換バージョンで作成された 2 つのレイヤード・プロダクトを使用しています。

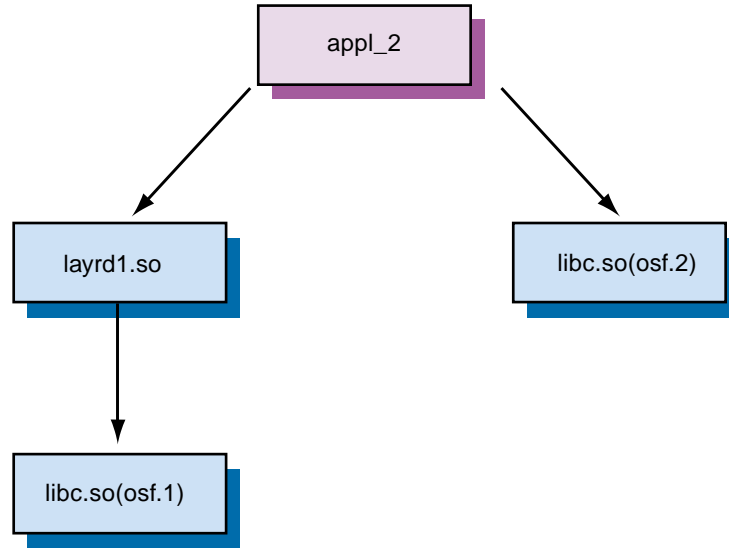
図 4-3: 共用オブジェクト間の無効な複数バージョンの従属: 例 1



ZK-0884U-AI

図 4-4 では、アプリケーションは、基本システムの非互換バージョンを使用して作成されたレイヤード・プロダクトとリンクしています。

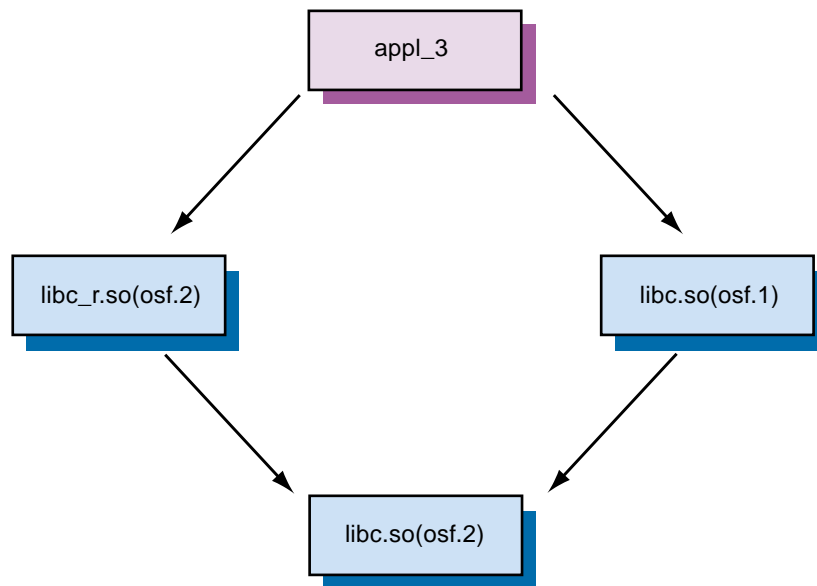
図 4-4: 共用オブジェクト間の無効な複数バージョンの従属: 例 2



ZK-0885U-AI

図 4-5 では、アプリケーションは、部分シェアード・ライブラリとしてインプリメントされた下位互換の不完全なライブラリとリンクしています。

図 4-5: 共用オブジェクト間の無効な複数バージョンの従属: 例 3

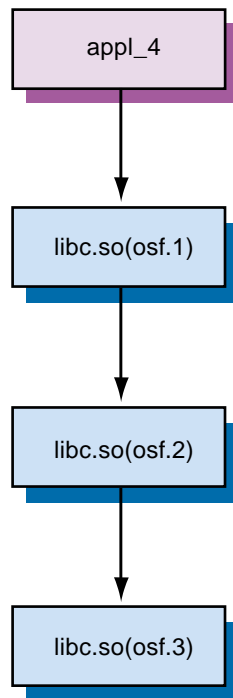


ZK-0886U-AI

次の例は、シェアード・ライブラリの複数バージョンの有効な使用を示しています。

図 4-6 では、アプリケーションは、部分シェアード・ライブラリとしてインプリメントされた下位互換のライブラリを使用しています。

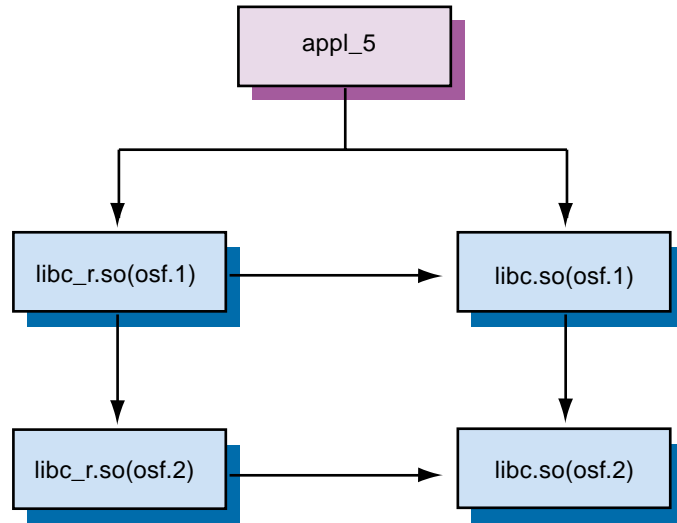
図 4-6: シェアード・ライブラリの複数バージョンの有効な使用: 例 1



ZK-0887U-AI

図 4-7 では、アプリケーションは、2 つの下位互換のライブラリを使用しています。そのうちの 1 つは、もう一方に依存しています。

図 4-7: シェアード・ライブラリの複数バージョンの有効な使用: 例 2



ZK-0888U-AI

4.12 シンボル割り当て

ローダによるシンボル解決の方法には、即時割り当ておよび遅延割り当ての2つの方法があります。即時割り当ての場合は、実行可能プログラムあるいはシェアード・ライブラリがロードされるときにシンボルが解決されます。遅延割り当ての場合、テキスト・シンボルは実行時に解決されます。遅延テキスト・シンボルは、プログラムで最初に参照される際に解決されます。

省略時の設定では、プログラムは遅延割り当てでロードされます。

`LD_BIND_NOW` 環境変数にヌル以外の値に設定すると、その後のプログラムの実行は即時割り当てによって行われます。

即時割り当ては、解決できないシンボルを識別するのに便利です。遅延割り当ての場合は、その部分のコードが実行されるまで、解決できないシンボルを検出できません。

また、即時割り当てを使用するとシンボル解決のオーバーヘッドを軽減できます。

4.13 シェアード・ライブラリの制約事項

シェアード・ライブラリを使用するには、次の制約事項を満たす必要があります。

- シェアード・ライブラリには未定義シンボルがあってはならない。
シェアード・ライブラリが参照するシンボルを定義している他のシェアード・ライブラリと、そのシェアード・ライブラリを明示的にリンクしなければなりません。
実行可能プログラム内のシンボルを参照するシェアード・ライブラリのような場合には、未定義のシンボル参照を回避することは困難です。シェアード・ライブラリにおける未解決の外部シンボルの処理方法については、4.2.4 項を参照してください。
- アセンブラ・ファイルや、レベル 03 で最適化されている古いオブジェクト・ファイルおよび C ファイルは、シェアード・ライブラリでは動作しない場合がある。
シェアード・ライブラリでは、Tru64 UNIX C コンパイラを使用し、最適化レベル 02 以下でコンパイルされた C モジュールが使用可能です。シェアード・ライブラリにリンクされている実行プログラムは、最適化レベル 03 以下でコンパイルすることができます。
- `setuid` または `setgid` サブルーチンを使用してインストールしているプログラムでは、ライブラリ探索を制御するさまざまな環境変数 (`LD_LIBRARY_PATH`, `_RLD_ARGS`, `_RLD_LIST`, `_RLD_ROOT` など) を使用せず、システム・インストールによるライブラリ (`/usr/shlib` にあるライブラリ) だけを使用する。
これにより、プログラムのセキュリティに対する潜在的な危険が取り除かれます。これは、実行時ローダ (`/sbin/loader`) により行われます。

dbx によるプログラムのデバッグ

dbx デバッガは、コマンド行プログラムです。これは、ソース・コード・レベルおよび機械語コード・レベルのプログラム・デバッグ用のツールであり、C 言語、Fortran、Pascal、アセンブリ言語で使用することができます。dbx を起動した後に、dbx コマンドを入力して実行の制御およびトレース、変数および式の値の表示、ソース・ファイルの表示および編集を行うことができます。

代替デバッガの ladebug デバッガは、コマンド行およびグラフィカル・ユーザ・インタフェース (GUI) の両方を提供しています。dbx でサポートされていない言語をいくつかサポートするとともに、ladebug デバッガではマルチスレッド・プログラムのデバッグ機能をサポートしています。ladebug についての詳細は、『*Ladebug Debugger Manual*』または ladebug(1) を参照してください。

この章では、次の項目について説明を行います。

- デバッグの一般的な留意事項 (5.1 節)
- dbx デバッガの実行 (5.2 節)
- dbx コマンドの使用方法 (5.3 節)
- dbx モニタが提供するオプションによる dbx コマンドの入力 (5.4 節)
- dbx の制御 (5.5 節)
- ソース・コードと機械語コードの確認 (5.6 節)
- デバッグ中のプログラムの実行の制御 (5.7 節)
- ブレークポイントの設定 (5.8 節)
- プログラムの状態の確認 (5.9 節)
- 複数のコア・ファイルの保存 (5.10 節)
- 実行中のプロセスのデバッグ (5.11 節)
- マルチスレッド・プロセスのデバッグ (5.12 節)

- 複数の非同期プロセスのデバッグ (5.13 節)
- この章全体の例で参照されている C のサンプル・プログラム `sam.c` (5.14 節)

`dbx` コマンド行オプション, `dbx` コマンド, 変数などについての詳しい説明は, `dbx(1)` を参照してください。

Visual Threads (「Associated Products Volume 1」CD-ROM で提供) を使用しても, 論理および性能上の問題がないかどうかについて, マルチスレッド・アプリケーションを分析することができます。Visual Threads は, POSIX Threads Library アプリケーションおよび Java アプリケーションで 사용할 수 있습니다。

この章の例ではサンプル・プログラム `sam` を引用します。C 言語のソース・プログラム `sam.c` は, 例 5-1 に示してあります。

この章のコマンド定義では, 本書のまえがきで説明した表記法のほかに, 表 5-1 で説明するキーワードを使用します。大文字の特定語は, 特定の規則が適用される変数を示します。

表 5-1: コマンド構文の記述に使用されるキーワード

キーワード	値
ADDRESS	マシン・アドレスを指定する任意の式。
COMMAND_LIST	セミコロンで区切られた 1 つまたは複数のコマンド。
DIR	ディレクトリ名。
EXP	<code>dbx</code> コマンドのプログラム変数名を含む任意の式。式には <code>(<code>\$listwindow</code> + 2)</code> のように <code>dbx</code> 変数を指定することができる。式で変数名 <code>in</code> , <code>to</code> , または <code>at</code> を使用する場合は, カッコで囲む必要がある。カッコで囲まなければ, これらの語はデバッガ・キーワードとみなされる。
FILE	ファイル名。
INT	整数値。
LINE	ソース・コードの行番号。
NAME	<code>dbx</code> コマンドの名前。
PROCEDURE	スタック上にあるプロシージャ名またはアクティブ化レベル。
REGEXP	正規表現の文字列。 <code>ed(1)</code> を参照。
SIGNAL	システム・シグナル。 <code>signal(2)</code> を参照。

5-2 `dbx` によるプログラムのデバッグ

表 5-1: コマンド構文の記述に使用されるキーワード (続き)

キーワード	値
STRING	任意の ASCII 文字列。
VAR	有効なプログラム変数または定義済みの dbx 変数。表 5-8 を参照。機械語レベルのデバッグでは、VAR にはアドレスも指定可。5.3.2 項で説明するように、二重の名前を使用してプログラム変数を指定しなければならない。

コマンドに大文字を使用する例を次に示します。

(dbx) stop VAR in PROCEDURE if EXP

例のように、stop、in、および if を入力します。表 5-1 の定義に従って、VAR、PROCEDURE、および EXP に値を入力してください。

注意

特定の dbx コマンドを拡張して非同期セッションの制御を行うことを含め、複数の非同期プロセスのデバッグについては、5.13 節で説明しています。

5.1 デバッグの一般的な留意事項

この節では、dbx デバッガおよびデバッグの概念のいくつかを説明します。また、デバッグ・セッションを実行する方法については、セッションを開始できる状況、エラーを取り除く方法、および起こしやすいエラーを回避する方法を含めて説明します。プログラマの経験が十分ある場合は、この節を無視しても構いません。

5.1.1 ソース・レベルのデバッガを使用する理由

dbx デバッガを使用すると、ソース・コード・レベルまたは機械語コード・レベルで、ターゲット・プログラムの問題をトレースすることができます。dbx を使用して、プログラムの制御フロー、変数、およびメモリ位置を監視しながらプログラムの実行を制御します。また、他の人が書いたプログラムを理解するために、dbx を使用して、そのプログラムのロジックおよび制御フローをトレースすることができます。

5.1.2 アクティブ化レベル

アクティブ化レベルは、スタックで現在アクティブな範囲で、通常はアクティブなプロシージャです。アクティブ化スタックとは、通常 `main()` という最初のプログラムから始まる呼び出しのリストです。一番最後に呼び出されたプロシージャまたはブロックには 0 の番号が付けられます。次に呼び出されたプロシージャには 1 の番号が付けられます。最後のアクティブ化レベルは、常にメイン・プロシージャ、つまりプログラム全体を制御するプロシージャになります。アクティブ化レベルはまた、プロシージャ内のローカル変数を定義するブロックで構成することもできます。スタック・トレースを行う際 (`where` および `tstack` デバッガ・コマンドを参照) およびアクティブ化スタック内を移動する際 (`up`, `down`, および `func` デバッガ・コマンドを参照) にアクティブ化レベルが表示されます。以下に、`where` コマンドでスタックをトレースした例を示します。

[illegible]

- 1 最後に呼び出されたプロシージャは `print` です。 `print` のアクティブ化レベルは 0 なので、この関数はスタックの最上位にあります。
- 2 メイン・プログラムは `main` です。
- 3 アクティブ化レベル番号
➤ は、現在調査中のアクティブ化レベルを示します。
- 4 プロシージャ名
- 5 プロシージャの引数
- 6 ソース・ファイル名
- 7 現在の行番号
- 8 現在のプログラム・カウンタ

5.1.3 プログラム実行障害箇所の特定

dbx デバッガは、実行時エラーだけを検出するため、デバッグ・セッションを開始する前に、コンパイラ・エラーを訂正する必要があります。実行時エラーとは、プログラムが、コア・ファイルを作成してしまうような実行中の障害、または不正な実行結果の作成につながるものです。実行時に異常終了するプログラムをデバッグする方法と、終了まで実行はするが、不正な実行結果を出力するプログラムをデバッグする方法は異なります。不正な実行結果を作成するプログラムをデバッグする方法については、5.1.4 項を参照してください。

実行時にプログラムが異常終了する場合には、1 行ずつデバッグするのではなく、次の方法を使用してデバッグ・セッションを開始すると、通常は時間を節約することができます。

1. dbx からプログラムを呼び出す。
dbx コマンド行で、適切なオプション、実行可能ファイル、およびコア・ダンプ・ファイルを指定します。
2. where コマンドを使用してスタック・トレースを行い、障害の位置を特定する。

注意

オブジェクト・ファイルからシンボル・テーブルの情報を削除していない場合は、当該プログラムが -g デバッグ・オプションでコンパイルされていなくても、スタック・トレースを行うことができます。

3. stop あるいは stopi コマンドを使用してブレークポイントを設定し、エラーを分離する。
4. print コマンドを使用して変数の値を表示し、変数に不正な値が割り当てられた箇所を調べる。

この時点でまだエラーを発見できない場合には、この章で説明している他の dbx コマンドを使用してください。

5.1.4 不正の出力結果の原因分析

プログラムが終了まで実行しても、不正な値または出力が生成される場合には、次の手順に従ってください。

1. 問題が発生していると考えられる位置、たとえば不正な値または出力のデータを生成する箇所のコードにブレークポイントを設定する。
2. プログラムを実行する。
3. `where` コマンドを使用して、スタック・トレースを行う。
4. `print` コマンドを使用して、問題があると考えられる変数の値を表示する。
5. 問題が発見されるまで、手順 1 以降を繰り返す。

5.1.5 実行プロセスのコア・スナップショットの作成

異常終了や不正な結果を出力しないプログラムでも、正常に動作していない場合があります。たとえば次のような場合です。

- プログラムの実行速度がひどく遅い。
- サーバの応答が停止した。

また、プログラムが無限ループに入ることもあります。

このような場合、プログラムの実行を停止せずに状態を調べることができます。`coredump` コマンドを使用すると、実行中のプロセスのコア・スナップショットを作成できます。省略時、コア・スナップショット・ファイルは現行ディレクトリに作成され、名前は `corefile` になります。`dbx` を使用すると、5.2.3 項での説明どおりにコア・スナップショットを調べることができます。詳細は `coredump(1)` を参照してください。

5.1.6 障害の回避

デバッガですべての問題が解決されるとは限りません。たとえば、プログラムに論理エラーがある場合には、デバッガは問題の発見には利用できませんが、解決はできません。デバッガにより表示される情報が混乱していたり間違っているような場合には、次のような処理でその状況を訂正することができます。

- ソース・コード行を分離可能な箇所，たとえば，`if` 条件の後などで論理単位に分離する。

デバッガは，同じ行に複数のソース文が記述されていると認識することができません。

- 実行可能ファイル・コードが欠落しているように思われても，インクルード・ファイルに含まれている可能性がある。

デバッガは，インクルード・ファイルをファイルでなく単一の行であると思なします。このコードをデバッグする場合には，インクルード・ファイルから削除してプログラムの一部としてコンパイルします。

- ソース・コードを変更した後は，必ず再コンパイルする。

再コンパイルしていなければ，デバッガに表示したソース・コードが実行可能ファイル・コードと一致しません。ソース・ファイルが実行可能ファイルより新しい場合，デバッガは警告メッセージを表示します。

- `Ctrl/Z` を押してデバッガを終了した後に，同じデバッグ・セッションを再開する場合には，デバッガはセッションの開始時に指定したのと同じオブジェクト・モジュールに対する処理を続行する。

これは，デバッガを停止してコード内の問題を修正し，再コンパイルした後，デバッグ・セッションを再開した場合に，デバッガが読み込むオブジェクト・モジュールには，変更が反映されていないことを意味します。新しいセッションを開始する必要があります。

同様に，あるウィンドウでプログラムのデバッグを実行している場合に，他のウィンドウでそのプログラムを編集して再コンパイルしても，`dbx` はこの変更を認識しません。プログラムの変更を `dbx` に認識させるには，変更のたびに `dbx` を終了してから再起動してください。

- `dbx` キーワードと同じ名前が含まれる式を表示するためにコマンドを入力する場合には，その式をカッコで囲まなければならない。

たとえば，`playback` および `record` コマンドのキーワードである `output` の値を表示するには，次のように指定しなければなりません。詳細については 5.9.4 項を参照してください。

```
(dbx) print (output)
```

- デバッガが変数または実行可能ファイル・コードのいずれも表示しない場合には，プログラムをコンパイルしたときに `-g` オプションを使用したかどうかを確認する。

5.2 dbx の実行

dbx を呼び出す前に、デバッグ用にプログラムをコンパイルする必要があります。デバッガ起動時に各 dbx コマンドを実行する dbx 初期化ファイルを作成することもできます。

5.2.1 デバッグ用プログラムのコンパイル

デバッガ用にプログラムの準備をするには、コンパイル時に `-g` オプションを指定します。このオプションにより、シンボル・テーブル情報が dbx プログラムに読み込まれ、最適化レベルを `-O0` に設定します。dbx デバッガは、この情報を使用してソース行をリストします。別の最適化レベル (`-O2` など) を使用すると、最適化プログラムは、プログラム内の制御フローを変更はしませんが、演算命令を移動し、オブジェクト・コードとソース・コードが一致なくなることがあります。これらの変更されたコード・シーケンスは、デバッガを使用する場合に混乱を招く可能性があります。

`-g` オプションを付けずにコンパイルしたコードで、限定されたデバッグを実行することができます。たとえば、次のコマンドは、デバッグ用に再コンパイルしなくても、正常に動作します。

- `stop in PROCEDURE`
- `stepi`
- `cont`
- `conti`
- `(ADDRESS) /<COUNT> <MODE>`
- `tracei`

このコードで限定されたデバッグを実行することはできますが、`-g` を使用してプログラムを再コンパイルする方が有効です。`-g` オプションを付けないでオブジェクト・ファイルをコンパイルしても、デバッガは警告を出さないことに注意してください。

完全なシンボル・テーブル情報が利用できるのは、すべてのモジュールが `-g` オプションを付けてコンパイルされているプログラムだけです。そうでないプログラムの場合は、`-g` オプションを付けてコンパイルされたモジュールで参照されているか、または定義されているシンボルに関するシンボル・テーブル情報しか利用できません。

注意

ブレークポイントを設定するシェード・ライブラリ・アプリケーションのルーチンはすべて、`-g` オプションを使用してコンパイルする必要があります。`-g` オプションを指定しない場合には、`dbx` にブレークポイントを設定するのに必要なシンボル・テーブルの情報が生成されないため、`dbx` は、アプリケーションを停止することができません。

5.2.2 dbx 初期化ファイルの作成

各 `dbx` セッションの初めに、通常入力するコマンドを含む `dbx` 初期化ファイルを作成することができます。たとえば、そのファイルには、次のようにコマンドを指定することができます。

```
set $page = 5
set $lines = 20
set $prompt = "DBX> "
alias du dump
```

初期化ファイルには、`.dbxinit` という名前を付けなければなりません。デバッガを実行するたびに、`dbx` は、`.dbxinit` にあるコマンド群を実行します。`dbx` デバッガは、初めに現在のディレクトリで、次に `$HOME` 環境変数に割り当てられたディレクトリであるホーム・ディレクトリで、`.dbxinit` を探します。

5.2.3 dbx の起動と終了

`dbx` コマンドおよび必要なパラメータを入力して、シェル・コマンド行から `dbx` を実行してください。実行後に、`dbx` は、現在の関数をプログラムの最初のプロシージャに設定します。

`dbx` コマンドの構文は、次のとおりです。

dbx [*options*] [*objectfile*] [*corefile*]

options

`dbx` コマンド行がサポートするオプションについては、表 5-2 を参照してください。

<i>objectfile</i>	デバッグ対象のプログラムの実行可能ファイル名を指定します。 <i>objectfile</i> が指定されない場合， <i>dbx</i> は，省略時の設定 <i>a.out</i> を使用します。
<i>corefile</i>	プログラムの実行が異常終了した場合や， <i>coredump(1)</i> コマンドを実行して実行中プロセスのコア・スナップショットを記録した場合に作成される，コア・ダンプ・ファイルの名前を指定します。ダンプ・ファイルには，プログラムの異常終了時，またはスナップショットが記録された時点でのメモリのイメージが格納されています。コマンド行でコア・ファイルを指定した場合， <i>dbx</i> はプログラムの異常終了またはスナップショットの記録された位置を特定します。 <i>dbx</i> コマンドを使用すると，その時点でのプログラムの状態が判断できます。コア・ダンプ・ファイルはすべて，省略時には <i>core</i> という名前です。システムまたはアプリケーションのレベルでコア・ファイルの命名を有効にする方法については， 5.10 節 を参照してください。

dbx で指定可能な引数の最大数は 1000 です。ただし，使用しているマシンのシステム制限により，この数は削減されていることがあります。

表 5-2: *dbx* コマンド・オプション

オプション	機能
<i>-cfilename</i>	<i>.dbxinit</i> ファイル以外の初期化コマンド・ファイルを選択する。
<i>-Idirname</i>	指定したディレクトリでソース・ファイルを探すように <i>dbx</i> に指示する。複数のディレクトリを指定するには，それぞれのディレクトリに <i>-I</i> を使用しなければならない。 <i>dbx</i> の実行時に <i>-I</i> を指定しないと，デバッガは，現在のディレクトリおよびオブジェクト・ファイルのディレクトリでソース・ファイルを探す。 <i>use</i> コマンドを使用して，ディレクトリを変更することができる。 5.6.1 項を参照。
<i>-i</i>	対話型モードで <i>dbx</i> を実行する。このオプションを付けると，デバッガが番号記号 (#) で始まるソース行をコメントとして扱わないようになる。

表 5-2: dbx コマンド・オプション (続き)

オプション	機能
-k	メモリ・アドレスをマップする。このオプションは、カーネルのデバッグに有効である。カーネルのデバッグについての詳細は、『Kernel Debugging』を参照。
-module_path	dbx がシェアード・ライブラリ (またはロード可能カーネル・モジュール) を検索するディレクトリ・パスを指定する。このオプションは、たとえば、コア・ダンプ (またはカーネル・クラッシュ・ダンプ) をデバッグしていて、そのダンプが発生したときに実行していたシェアード・ライブラリ (またはモジュール) のバージョンが別の場所に移動している場合などに有用である。カーネルのデバッグについての詳細は、『Kernel Debugging』を参照。
-module_verbose	シェアード・ライブラリのロード時に dbx がそのパス (または、カーネルをデバッグしている場合は、ロード可能モジュール) をプリントするようにする。省略時の設定では、dbx はパスをプリントしない。カーネルのデバッグについての詳細は、『Kernel Debugging』を参照。
-pid process-id	dbx を現在実行中のプロセスにアタッチする。
-r	コマンド行で指定するオブジェクト・ファイルをただちに実行する。プログラムの実行がエラーで終了した場合には、dbx は、そのエラーを説明するメッセージを表示する。次に、デバッグを実行するか、またはそのプログラムに関する処理を終了させることができる。-r オプションを指定し、標準入力端末でない場合、dbx デバッグは、/dev/tty から読み取りを行う。プログラムが正常終了した場合には、dbx は入力を求める。

次の例では、オプションを付けずに dbx を実行します。オブジェクト・ファイル名が指定されていないため、dbx は、ファイル名の入力を求めます。この場合は、sam を使用して応答します。省略時のデバッグのプロンプトは、(dbx) です。

```
% dbx
enter object file name (default is 'a.out'): sam
dbx version 3.12
Type 'help' for help.

main: 23  if (argc < 2) {
(dbx)
```

デバッグセッションを終了する場合は、quit コマンドを使用します (簡略形の q コマンドも使用できます)。quit コマンドには引数は不要です。

5.3 dbx コマンドの使用法

入力行には、10,240 までの文字を入力することができます。行が長い場合は、バックスラッシュ (\) を入力してからその後に続けます。行が長すぎると、dbx は、エラー・メッセージを表示します。最大の文字列の長さも、10,240 です。

次の各項では、変数名の修飾、dbx の式と優先順位、および dbx のデータ型と定数について説明します。

5.3.1 変数名の修飾

dbx の変数は、ファイル、プロシージャ、ブロック、または構造体で修飾します。print などのコマンドを使用して変数の値を表示する場合、dbx は、たとえば、2 つ以上のプロシージャに同じ名前の変数がある場合など、変数の有効範囲が曖昧である可能性があるとして、その有効範囲を表示します。その有効範囲が間違っている場合は、次のようにピリオドで区切って、変数の有効範囲のすべてを指定することができます。

sam.main.i

| | |
| | |
1 **2** 3

- 1 現在のファイル
- 2 プロシージャ名
- 3 変数名

5.3.2 dbx 式と式の優先順位

dbx デバッガは、C 言語の式演算子を認識します。また、これらの演算子は、他のサポート言語のデバッグにも使用されます。dbx では、Fortran の場合も配列の添字に [] を使用します。標準の C 演算子に加えて、表 5-3 に示すように、dbx は # 記号も演算子として使用します。

表 5-3: dbx の # 式演算子

演算子	構文の説明
("FILE" #EXP)	FILE に指定されたファイル内の #EXP で指定される行番号を使用する。
(PROCEDURE #EXP)	PROCEDURE に指定されたプロシージャ内の #EXP で指定される相対行番号を使用する。
(#EXP)	(#EXP) で指定される行のアドレスを返す。

各演算子の優先順位は、C 言語における優先順位に従います。表 5-4 に、dbx が認識する言語演算子を優先順位の高い順に示します。

表 5-4: C の式演算子

単項	&, +, -, * (ポインタ), #, sizeof() ^a , ~, /, (type), (type *)
2 項	<<, >>, ", !, ==, !=, <=, >=, <, >, &, &&, , , +, -, *, / ^b , %, [], ->

^asizeof 演算子は、(number-of-bits + 7) / 8 ではなく、ある要素を取得するために読み出したバイト数を指定します。
^bdbx では、除算演算子として // も使用できます。

5.3.3 dbx のデータ型および定数

表 5-5 に、dbx コマンドが使用できる組み込みデータ型をリストします。

表 5-5: 組み込みデータ型

データ型	説明	データ型	説明
\$address	ポインタ	\$real	倍精度実数
\$boolean	ブール	\$short	16 ビット整数
\$char	文字	\$signed	符号付き整数
\$double	倍精度実数	\$uchar	符号なし文字
\$float	単精度実数	\$unsigned	符号なし整数
\$integer	符号付き整数	\$void	空

型強制のための組み込みデータ型を使用して、たとえば、変数の宣言で指定した型以外の型で変数の値を表示することができます。dbx デバッガは C 言語のデータ型を理解しているため、ドル記号 (\$) を付けなくてもデータ型を参照できます。dbx への入力として受け入れ可能な定数の型を表 5-6

に示します。定数は、省略時の設定では、10 進数の値として dbx の出力に表示されます。

表 5-6: 入力可能な定数

定数	説明
false	0
true	非ゼロ
nil	0
0xnumber	16 進数
0tnumber	10 進数
0number	8 進数
number	10 進数
number.[number] [e E] [+ -] EXP	浮動小数点

注意

浮動小数点以外のオーバーフローでは、右端の数字を使用します。
浮動小数点のオーバーフローでは、仮数の左端、また、指数は、最上位または最下位のいずれか可能な方が使用されます。

\$octin 変数は、省略時の入力を 8 進数に変更します。\$hexin 変数は、省略時の入力を 16 進数に変更します。 5.5.2 項を参照してください。

\$octints 変数は、省略時の出力を 8 進数に変更します。
\$hexints 変数は、省略時の出力を 16 進数に変更します。
5.5.2 項を参照してください。

5.4 dbx モニタによる作業

dbx デバッガでは、コマンド・ヒストリ、コマンド行編集、およびシンボル名の補完が可能です。また、dbx デバッガでは、入力行に複数のコマンドを使用することができます。これらの機能を使用すると、必要な入力数を減らしたり、前に実行したコマンドを繰り返すことができます。

5.4.1 dbx コマンドの繰り返し

dbx デバッガは、コマンドの履歴を記憶して、再入力しなくてもデバッガ・コマンドを繰り返し入力できるようにします。これらのコマンドは、`history` コマンドを使用して表示することができます。 `$lines` 変数は、記憶する履歴行の数を制御します。省略時の設定は、20 コマンドです。 `set` コマンドを使用して、`$lines` 変数を変更することができます。 5.5.1 項を参照してください。

コマンドを繰り返すには、Return キー、または感嘆符 (!) コマンドの 1 つを使用します。

次の表に、`history` コマンドの形式を示します。

<code>history</code>	履歴・リストのコマンド群を表示する。
Return キー	最後に入力したコマンドを繰り返す。この機能は、 <code>\$repeatmode</code> 変数を 0 に設定するとオフになる。 5.5.1 項を参照。
<code>!string</code>	指定した文字列で始まるコマンドで最新のものを繰り返す。
<code>!integer</code>	指定した整数に対応するコマンドを繰り返す。
<code>!-integer</code>	最後に入力したコマンドから数えて、指定した数 (<code>integer</code>) だけ前に実行したコマンドを繰り返す。

次の例では、履歴・リストを表示した後、リストの 12 番目のコマンドを再実行しています。

```
(dbx) history
10 print x
11 print y
12 print z
(dbx) !12
(!12 = print z)
123
(dbx)
```

5.4.2 dbx コマンド行の編集

dbx デバッガには、コマンド行編集のためのコマンドが備わっています。これらのコマンドを使用すると、コマンド全部を再入力せずにタイプ・ミスを訂正することができます。コマンド行の編集を使用可能にするには、dbx を呼び出す前に、EDITOR、EDITMODE、または LINEEDIT 環境変数を設定します。C シェルの場合に LINEEDIT を設定するには、次のコマンドを入力します。

```
% setenv LINEEDIT
```

Bourn シェルあるいは Korn シェルの場合は、次のコマンドを入力します。

```
% export LINEEDIT
```

このデバッガでは、コマンド行編集で次のモードが使用できます。

- 環境変数 LINEEDIT が設定されていないときに、環境変数 EDITMODE または EDITOR のいずれかに最後が vi のパスが含まれている場合、デバッガは、Korn シェルの vi モードに似たコマンド行編集モードを使用します。このモードでは、次の編集キーが認識されます。

```
$ + - 0 A B C D E F I R S W X ^  
a b c d e f h i j k l r s w x ~  
Ctrl/D  
Ctrl/H  
Ctrl/J  
Ctrl/L  
Ctrl/M  
Ctrl/V
```

詳細については、ksh(1) を参照してください。

- 環境変数 LINEEDIT が、空文字列を含め、任意の値に設定されているか、または LINEEDIT が設定されていないとき、環境変数 EDITMODE または EDITOR に emacs で終わるパスが含まれている場合には、デバッガは Korn シェルの emacs モードに似たコマンド行編集モードを使用します。このモードは、LINEEDIT、EDITOR、または EDITMODE のいずれにより使用が可能になっているかによって、動作が若干異なります。

表 5-7 に、emacs モードのコマンド行編集コマンドを示します。

表 5-7: emacs モードの dbx コマンド行編集コマンド

コマンド	機能
Ctrl/A	コマンド行の行頭にカーソルを移動する。
Ctrl/B	カーソルを 1 文字前に移動する。
Ctrl/C	行を消去する。
Ctrl/D	カーソルの位置にある文字を削除する。
Ctrl/E	行末へカーソルを移動する。
Ctrl/F	カーソルを 1 文字後に移動する。
Ctrl/H	カーソルの直前の文字を削除する。
Ctrl/J	その行を実行する。
Ctrl/K	(EDITOR または EDITMODE によって使用が可能になっている場合) カーソル位置から行末までを削除する。現在のカーソル位置より小さい値の数値パラメータがその前に指定されている場合は、当該位置からカーソルまでを削除する。現在のカーソル位置より大きな値の数値パラメータがその前に指定されている場合は、カーソル位置から当該位置までを削除する。
Ctrl/K <i>char</i>	(LINEEDIT によって使用が可能になっている場合) <i>char</i> で指定した文字上にあるカーソルまでの文字を削除する。
Ctrl/L	現在の行を再表示する。
Ctrl/M	その行を実行する。
Ctrl/N	ヒストリ・リストの中の 1 行先に移動する。
Ctrl/P	ヒストリ・リストの中の 1 行後に移動する。
Ctrl/R <i>char</i>	指定した文字を現在の行の中で探索する。
Ctrl/T	カーソルの直前の 2 文字を入れ替える。
Ctrl/U	次の文字を 4 回繰り返す。
Ctrl/W	現在の行全体を削除する。
Ctrl/Y	Ctrl/K で削除したテキストをカーソルの直前に挿入する。
Ctrl/Z	ファイル名またはシンボル名を補完する。
Escape	ファイル名またはシンボル名を補完する。
下向き矢印	ヒストリ・リストの中の 1 行先に移動する。
上向き矢印	ヒストリ・リストの中の 1 行後に移動する。

表 5-7: emacs モードの dbx コマンド行編集コマンド (続き)

コマンド	機能
左向き矢印	カーソルを 1 文字前に移動する。
右向き矢印	カーソルを 1 文字後に移動する。

5.4.3 複数のコマンドの入力

セミコロン (;) を区切り記号として使用して、コマンド行に複数のコマンドを入力することができます。この機能は、when コマンドを使用している場合に有効です。 5.8.4 項を参照してください。

1 つのコマンド行に 2 つのコマンドがある例を次に示します。最初のコマンドでプログラムを停止させ、2 つ目のコマンドでそれを再実行しています。

```
(dbx) stop at 40; rerun
[2] stop at "sam.c":40
[2] stopped at [main:40 ,0x120000b40] i=strlen(line1.string);
(dbx)
```

5.4.4 シンボル名の補完

dbx デバッガを使用して、シンボル名を補完することができます。名前の一部を指定して Ctrl/Z を押すと、dbx は、固有の接頭語で始まる名前を補完します。見つかった名前が 1 つの場合、dbx はその名前を補完して再表示します。2 つ以上の名前が見つかった場合には、一致するすべてのシンボル名が表示され、その 1 つを選択することができます。

シンボル名の補完を使用可能にするには、5.4.2 項で説明する方法でコマンド行編集を使用可能に設定する必要があります。

次の例では、英字 i で始まるすべての名前を表示しています。

```
(dbx) i Ctrl/Z
ioctl.ioctl .ioctl isatty.isatty .isatty i int 1
(dbx) i 2
```

- 1 データ型およびライブラリのシンボルが含まれている可能性があります。
- 2 指定の文字で始まるすべての名前をリストした後、別の文字の指定および再探索ができるように、dbx はプロンプトとともに事前に指定された文字列を再表示します。

次に、シンボル名の単純な補完の例を示します。

```
(dbx) print file Ctrl/Z
(dbx) print file_header_ptr
0x124ac
(dbx)
```

5.5 dbx の制御

dbx デバッガには、dbx 変数の設定および削除、別名の作成および削除、サブシェルの起動、状態リスト項目の検査および削除、アプリケーションに関連するオブジェクト・ファイル・リストの表示、入力の記録および再生のためのコマンドが用意されています。

5.5.1 変数の設定および削除

set コマンドは、dbx 変数の定義、既存の dbx 変数値の変更、dbx 定義済み変数のリストの表示などを行います。unset コマンドは、dbx 変数を削除します。プログラム変数およびデバッガ変数の値を表示するには、print コマンドを表示します。dbx の定義済み変数については、表 5-8 を参照してください。プログラム変数と同じ名前でデバッガ変数を定義することはできません。

set および unset コマンドの形式は次のとおりです。

```
set                                     dbx の定義済み変数のリストを表示する。
```

```
set VAR = EXP                         変数に新しい値を割り当てるか、または新しい変数を定義する。
```

```
unset VAR                             dbx 変数の値を設定解除する。
```

次の例は、set および unset コマンドの使用例を示しています。

```
(dbx) set 1
$listwindow      10
$datacache       1
$main            "main"
$pagewindow      22
test             5
$page            1
$maxstrlen       128
$cursrcline      24
more (n if no)? n
```

```

(dbx) set test = 12 2
(dbx) set
$listwindow      10
$datacache       1
$main            "main"
$pagewindow      22
test             12
$page            1
$maxstrlen       128
$cursorline      24
more (n if no)? n
(dbx) unset test 3
(dbx) set
$listwindow      10
$datacache       1
$main            "main"
$pagewindow      22
$page            1
$maxstrlen       128
$cursorline      24
more (n if no)? n
(dbx)

```

- ① dbx 定義済み変数のリストを表示します。
- ② 変数に新しい値を割り当てます。
- ③ 変数を削除します。

5.5.2 定義済みの dbx 変数

表 5-8 に定義済みの dbx の変数を示します。「タイプ」欄の I は整変数, B はブール変数, S は文字変数を示します。テストは可能であるが変更できない変数は R で示します。

表 5-8: 定義済みの dbx 変数

タイプ	変数名	省略時の値	説明
S	\$addrfmt	"0x%lx"	アドレスのフォーマットを指定する。この変数は、C 言語の printf 文でフォーマットできるものであればいずれにも設定することができる。
B	\$assignverify	1	変数に値を割り当てる場合に新しい値を表示するかどうかを指定する。
B	\$asynch_interface	0	複数の非同期プロセスを制御するように dbx を構成する (または、できる) かどうかを制御する。プロセスがアタッチされると 1 だけ増加され、プロセスが終了するかデタッチされると 1 だけ減少される。ユーザによる設定も可能。0 または負の値が設定された場合、非同期的デバッグは使用不能になる。
B	\$break_during_step	0	step/stepi, next/nexti, call, return などの処理中にブレークポイントをチェックするかどうかを制御する。
B	\$casesense	0	ソース探索および変数で大文字と小文字を区別するかどうかを指定する。0 以外の値は区別することを意味し、0 は区別しないことを表す。
I R	\$curevent	0	status コマンドによって最後に報告されたイベント番号を表示する。
I R	\$curline	0	ソース・コードの現在の行を表示する。
I R	\$curpc	-	現在のアドレスを表示する。wi および li の別名で使用する。

表 5-8: 定義済みの dbx 変数 (続き)

タイプ	変数名	省略時の値	説明
I R	\$cursrcline	1	表示された最後の行の行番号に 1 を加えたものを表示する。
B	\$datacache	1	データ領域から情報をキャッシュして、dbx が 1 度だけデータ領域を確認すればすむようにする。オペレーティング・システムをデバッグする場合には、この変数を 0 に設定する。他のシステムをデバッグする場合は、非ゼロ値に設定する。
S R	\$defaultin	ヌル文字列	record input コマンドを使用する場合に、dbx が情報を格納するために使用するファイル名を表示する。
S R	\$defaultout	ヌル文字列	record output コマンドを使用する場合に、dbx が情報を格納するために使用するファイル名を表示する。
B	\$dispix	0	1 の場合、pixie モードでデバッグする際に実命令のみを表示する。
B	\$hexchars	未定義	非ゼロ値は、文字が 16 進数で表示されることを示す。
B	\$hexin	未定義	非ゼロ値は、入力された定数が 16 進数であることを示す。
B	\$hexints	未定義	非ゼロ値は、出力される定数が 16 進数で表示されることを示す。16 進数の設定は 8 進数に優先する。
B	\$hexstrings	未定義	非ゼロ値に設定すると、文字列が 16 進数で表示される。ゼロの場合は、文字列は 16 進数ではなく文字で表示される。

表 5-8: 定義済みの dbx 変数 (続き)

タイプ	変数名	省略時の値	説明
I R	\$historyevent	なし	現在のヒストリ番号を表示する。
I	\$lines	20	dbx ヒストリ・リストのサイズを指定する。
I	\$listwindow	\$pagewindow/2	list コマンドが表示する行の数を指定する。
S	\$main	"main"	実行が開始されるプロシージャの名前を指定する。dbx は、特に指定しない限り、main() で開始される。
I	\$maxstrlen	128	dbx が文字列のポインタに対してプリントする、文字列の最大文字数を指定する。
S	\$module_path	ヌル文字列	dbx がシェアード・ライブラリ (またはロード可能カーネル・モジュール) を検索するディレクトリ・パスを指定する。この変数は、たとえば、コア・ダンプ (またはカーネル・クラッシュ・ダンプ) をデバッグしていて、そのダンプが発生したときに実行されていたシェアード・ライブラリ (またはモジュール) のバージョンが別の場所に移動されている場合に有用。カーネルのデバッグについての詳細は、『 <i>Kernel Debugging</i> 』を参照。

表 5-8: 定義済みの dbx 変数 (続き)

タイプ	変数名	省略時の値	説明
I	\$module_verbose	0	ゼロ以外の値に設定されていると、シェアード・ライブラリのロード時に、dbx はシェアード・ライブラリ (または、カーネルのデバッグ時にはロード可能モジュール) の位置をプリントする。省略時の設定、あるいは、この値が 0 に設定されている場合、dbx は位置をプリントしない。カーネル・デバッグについての詳細は、『 <i>Kernel Debugging</i> 』を参照。
B	\$octin	未定義	非ゼロ値に設定すると、省略時の入力定数を 8 進数に変更する。\$hexint は、この設定に優先する。
B	\$octints	未定義	非ゼロ値に設定すると、省略時の出力定数を 8 進数に変更する。\$hexints はこの設定に優先する。
B	\$page	1	長い情報をページングするかどうか指定する。非ゼロ値でページングを実行し、0 は実行しない。
I	\$pagewindow	種々	1 画面を越える長い情報を表示する場合に、表示する行数を指定する。この変数には、端末の表示行数を設定する。0 を指定すると、最小値の 1 行の設定になる。省略字の値は端末のタイプに依存する。標準ビデオ端末の場合は 24。
B	\$pimode	0	playback input コマンドを使用する場合の入力を表示する。

表 5-8: 定義済みの dbx 変数 (続き)

タイプ	変数名	省略時の値	説明
I	\$printdata	0	非ゼロ値に設定すると、命令を逆アセンブルする場合にレジスタの値が表示される。ゼロの場合は、レジスタの値は表示されない。
B	\$printtargets	1	1 に設定した場合、表示される逆アセンブルのリストに、ジャンプ命令のターゲットのラベルを含める。0 に設定した場合は、このラベルの表示を行わない。
B	\$printwhilestep	0	step [n] および stepi [n] の命令で使用する。ゼロ以外の値を指定すると、n 個のすべての行または命令が表示される。0 を指定した場合は、最後の行または命令だけが表示される。
B	\$printwide	0	変数を表示するフォーマットとして、水平または垂直のフォーマットを指定する。水平フォーマットは、構造体または配列の表示に有効。非ゼロ値は、水平フォーマットを示し、0 は垂直のフォーマットを示す。
S	\$prompt	" (dbx) "	dbx のプロンプトを設定する。
B	\$readtextfile	1	1 に設定した場合、dbx は、プロセスからではなくオブジェクト・ファイルから命令を読み取りとする。この変数は、デバッグ対象のプロセスがデバッグ処理の間に、コードとしてコピーされている場合には、常に 0 に設定する必要がある。ただし、性能は、\$readtextfile を 1 に設定した場合の方が優れている。

表 5-8: 定義済みの dbx 変数 (続き)

タイプ	変数名	省略時の値	説明
B	\$regstyle	1	使用するレジスタ名の型を指定する。1 に設定した場合はハードウェア名を指定する。0 に設定した場合は、ファイル <code>regdefs.h</code> で定義された、ソフトウェア名を指定する。
B	\$repeatmode	1	Return キーを押したときに最後のコマンドを再実行できるようにするかどうかを指定する。非ゼロ値の場合は最後のコマンドの再実行が可能であり、ゼロの場合は繰り返さない。
B	\$rimode	0	<code>record output</code> コマンドを使用する場合に、入力を記録する。
S	\$sigvec	"sigaction"	シグナル・ハンドラを設定するためにシステムが呼び出すコード名を dbx に通知する。
S	\$sigtramp	"_sigtramp"	ユーザ・シグナル・ハンドラを実行するためにシステムが呼び出すコード名を dbx に通知する。
B	\$stopall_on_step	1	1 の場合、dbx はフォークされたすべての子プロセスを停止する。0 の場合、さまざまなシステム・コールおよびライブラリ・コールで生成されたフォークの多くを無視する。\$stop_all_forks が設定されていないければ、\$stop_on_fork の値がフォークによる dbx の動作を決定する。\$stop_all_forks は、通常 \$stop_on_fork によって無視されるライブラリ・コールおよびシステム・コール内でのフォークをトラップする。

表 5-8: 定義済みの dbx 変数 (続き)

タイプ	変数名	省略時の値	説明
B	\$stop_in_main	N/A	使用されない。この変数は set コマンドによって表示されるが、現在では dbx 処理に対して効果はない。
B	\$stop_on_exec	1	dbx が execl() および execv() への呼び出しを検出して、新しく起動されたイメージを実行可能コードの最初の行で停止するかどうかを指定する。
B	\$stop_on_fork	1	1 の場合、dbx は、fork() または vfork() 呼び出しによって起動された新しいイメージを、そのメインの起動ポイントまで進めた後、停止させる。0 の場合、ブレークポイントまたはイベントにより停止するまで、処理を続行する。 \$stop_all_forks が設定されていなければ、dbx プログラムは、システム・コールまたはライブラリ・コールからのフォークでの停止を回避しようとする。
S	\$tagfile	"tags"	tag コマンドおよび tagvalue マクロがタグを探索するファイルであることを示すファイル名が入っている。
I	\$straploops	3	プログラムがトラップ処理ループに入ると dbx が見なす前に、SIGTRAP ハンドラへの連続する呼び出しの回数を指定する。

5.5.3 別名の定義および削除

alias コマンドは、新しい別名を定義したり、すべての現在の別名を表示したりします。

`alias` コマンドを使用すると、どのデバッガ・コマンドにも新しく名前を付けることができます。スペースの含まれるコマンドは、一重または二重の引用符で囲んでください。別名の一部としてマクロを定義することもできます。

`dbx` デバッガには、あらかじめ定義されている一群の別名があります。ユーザは、これらの別名を変更したり、または新しい別名を追加したりすることができます。別名を `.dbxinit` ファイルに入れておくことによって、後でデバッグ・セッションで使用することもできます。コマンドの別名を削除するには、`unalias` コマンドを使用します。`unalias` コマンドには削除する別名を指定します。別名の削除は、現在のデバッグ・セッションでのみ有効です。

`alias` および `unalias` コマンドの形式は次のとおりです。

`alias`

すべての別名のリストを表示する。

```
alias NAME1 [(ARG1,...,ARGN)] "NAME2"
```

新しい別名を定義する。NAME1 には新しい別名を、NAME2 にはコマンド文字列を、ARG1,...,ARGN にはコマンドの引数を指定する。

```
unalias NAME
```

コマンドの別名を削除する。NAME には別名を指定する。

`alias` および `unalias` コマンドの使用例を次に示します。

```
(dbx) alias [1]
h      history
si     stepi
Si     nexti
:

g      goto
s      step
More (n if no) ?n
(dbx) alias ok(x) "stop at x" [2]
(dbx) ok(52) [3]
[2] Stop at "sam.c":52 [4]
(dbx)
(dbx) unalias h [5]
(dbx) alias
si     stepi
Si     nexti
:
```

```
g      goto
s      step
More (n if no)? n
(dbx)
```

- ❶ 別名を表示します。
- ❷ ブレークポイントを設定するために別名を定義します。
- ❸ 52 行目にブレークポイントを設定します。
- ❹ 52 行目にブレークポイントを設定したことをデバッガが認識しています。
- ❺ 別名 `h` を削除します。別名のリストに表示されないことに注意してください。

5.5.4 デバッグ・セッション状態の監視

`status` コマンドは、次のコマンドのうちで現在設定されているものがあるかどうかを確認します。

- ブレークポイントに対する `stop` コマンドまたは `stopi` コマンド
- 行単位の変数トレース用の `trace` コマンドまたは `tracei` コマンド
- `when` コマンド
- ファイルに情報を保管する `record input` コマンドおよび `record output` コマンド

`status` コマンドには引数は不要です。

例

```
(dbx) status
[2] trace i in main
[3] stop in prnt
[4] record output /tmp/dbxt0018898 (0 lines)
(dbx)
```

大カッコ内の番号 (たとえば [2]) は、状態項目番号を示しています。

5.5.5 ブレークポイントの削除あるいは無効化

`delete` コマンドは、ブレークポイントを削除し、入出力の記録をやめます。ブレークポイントの削除および入出力記録の中止は、`status` コマンドが生成する状態リストから該当する項目を削除することによって行われます。

`disable` コマンドは、項目を削除しないでブレークポイントを無効にします。`enable` コマンドを使用すると、無効にしたイベントを再度有効にすることができます。

`delete` コマンドの形式は次のとおりです。

```
delete EXP1[, ..., EXPN]
```

指定した状態項目を削除する。

```
delete all  
delete *
```

すべての状態項目を削除する。

`delete` コマンドの使用例を次に示します。

```
(dbx) status  
[2] record output /tmp/dbxt0018898 (0 lines)  
[3] trace i in main  
[4] print pline at "sam.c":  
[5] stop in prnt  
(dbx) delete 4  
(dbx) status  
[2] record output /tmp/dbxt0018898 (0 lines)  
[3] trace i in main  
[5] stop in prnt  
(dbx)
```

`disable` および `enable` コマンドの形式は次のとおりです。

```
disable EVENT1[, EVENT2, ...]  
enable EVENT1[, EVENT2, ...]
```

指定したイベントを有効/無効にする。

```
disable all  
enable all
```

すべてのイベントを有効/無効にする。

5.5.6 ロードされたオブジェクト・ファイル名の表示

`listobj` コマンドは、`dbx` によってロードされたオブジェクト・ファイルの名前をアドレスおよびサイズとともに表示します。これらのオブジェクトには、メイン・プログラム、およびアプリケーションで使用されるすべてのシェアード・ライブラリが含まれます。`listobj` コマンドには引数は不要です。

例

```
(dbx) listobj
sam                                addr: 0x120000000    size: 0x2000
/usr/shlib/libc.so                addr: 0x3ff80080000  size: 0xbc000
(dbx)
```

5.5.7 コア・ダンプ用のシェアード・ライブラリの指定

コア・ダンプが発生すると、プログラムの使用するすべてのシェアード・ライブラリの位置がコア・ファイルに記録されて、`dbx` からライブラリが検索できるようになります。ダンプの発生時に実行中だったバージョンのシェアード・ライブラリが別の場所に移動されると、`dbx` はそれを検索できなくなります。次のいずれかの方法を使用して、`dbx` がシェアード・ライブラリを検索するディレクトリ・パスを指定できます (詳細については、`dbx(1)` を参照してください)。

- `dbx` コマンド行で、`-module_path` オプションを使ってディレクトリ・パスを指定します。次に例を示します。

```
% dbx a.out core -module_path /usr/project4/lib_dir
```

- `dbx` を呼び出す前に、環境変数 `DBX_MODULE_PATH` を設定します。次に例を示します。

```
% setenv DBX_MODULE_PATH /usr/project4/lib_dir
```

- `dbx` セッション中に、シェアード・ライブラリを動的にロードする場合は、まず `$module_path dbx` 変数を設定してから、`addobj` コマンドを使用してライブラリをロードします。次に例を参照してください。

```
(dbx) set $module_path /usr/project4/lib_dir
(dbx) addobj libdef.so
```

モジュールが適切な場所からロードされたことを確認するには、次のいずれかの方法で、`verbose` モジュール・ロード機能を設定します。

- `-module_verbose dbx` コマンド・オプションを指定する。
- 環境変数 `DBX_MODULE_VERBOSE` を任意の整数値に設定する。

- `$module_verbose dbx` 変数をゼロ以外の値に設定する。

5.5.8 dbx からのサブシェルの起動

dbx プロンプトでサブシェルを起動するには、`sh` コマンドを入力します。サブシェルから dbx に戻るには、`exit` を入力するか、または `Ctrl/D` を押します。単一のコマンドをサブシェルで実行してすぐに dbx に戻るには、`sh` のコマンドの後にシェル・コマンドを続けて入力してください。

例

```
(dbx) sh
% date
Tue Aug 9 17:25:15 EDT 1998
% exit:
(dbx) sh date
Tue Aug 9 17:29:34 EDT 1998
(dbx)
```

5.6 ソース・プログラムの検査

この節では、ソース・コードのリストおよび編集、ディレクトリの変更、ソース・ファイルの変更、ソース・コード内の文字列の探索、修飾されたシンボル名の表示、型宣言の表示などを行う方法について説明します。

5.6.1 ソース・ファイルのディレクトリ位置の指定

デバッガの実行 (5.2.3 項を参照) 時に `-I` オプションが指定されていない場合、dbx は、現在のディレクトリまたはオブジェクト・ファイルのディレクトリでソース・ファイルを探索します。use コマンドには次の機能があります。

- デバッガが探索するディレクトリの変更
- 現在使用中のディレクトリのリスト

use コマンドは、絶対パス名および `./` などの相対パス名を認識します。ただし、C シェルのチルド (`~`) は認識しません。

use コマンドの形式は次のとおりです。
use

現在のディレクトリをリストする。

```
use DIR1 ... DIRN
```

現在のディレクトリ・リストを新しいディレクトリと置換する。

例

```
(dbx) use  
.  
(dbx) use /usr/local/lib  
(dbx) use  
/usr/local/lib  
(dbx)
```

① 現在のディレクトリ

② 新しいディレクトリ

5.6.2 アクティブ化スタックでの移動

5.1.2 項で説明するように、デバッガはアクティブ化スタックのレベルを保守します。特定のプロシージャの名前またはアクティブ化番号を探索するには、`where` あるいは `tstack` コマンドを使用してスタック・トレースを行ってください。アクティブ化スタック内の移動は、`up`、`down` および `func` コマンドで行うことができます。

5.6.2.1 `where` コマンドおよび `tstack` コマンド

`where` コマンドは、スタック・トレースを表示します。スタック・トレースによってデバッグしているプログラムの現在のアクティブ化レベル (実行中のプロシージャ) を知ることができます。`tstack` コマンドは、すべてのスレッドのスタック・トレースを表示します。スレッドのデバッグについては 5.12 節を参照してください。

`where` コマンドおよび `tstack` コマンドの形式は、次のとおりです。

```
where [EXP]
```

```
tstack [EXP]                   スタック・トレースを表示する。
```

`EXP` が指定されている場合、`dbx` はスタックの上位 `EXP` レベルのみを表示します。`EXP` が指定されていない場合は、スタック全体が表示されます。

サンプル・プログラム `sam.c` でブレークポイントが `prnt` に設定されると、プログラム `sam.c` は、プロシージャ `prnt()` で実行/停止します。 `where` を入力すると、スタック・トレースは次のような情報を表示します。

- ① アクティブ化レベル番号
- ② プロシージャ名
- ③ 引数 `pline` の現在の値
- ④ ソース・ファイル名
- ⑤ 行番号
- ⑥ プログラム・カウンタ

up および down コマンドを使用すると、スタック内のアクティブ化レベルを上下に移動できます。これらのコマンドは、あるレベルから別のレベルへの呼び出しを追跡する場合に便利です。

up , down および func コマンドの形式は次のとおりです。

<code>down [EXP]</code>	スタック内のアクティブ化レベルを、指定された数だけ下げる。省略時の値は 1 レベル。
<code>func</code>	現在のアクティブ化レベルを表示する。
<code>func PROCEDURE</code>	PROCEDURE に指定されたアクティブ化レベルに移動する。
<code>func EXP</code>	式によって指定されたアクティブ化レベルに移動する。

これらのコマンドの例を次に示します。

```
(dbx) where
> 0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
   1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
(dbx) up
main: 45 prnt(&line1);
(dbx) where
   0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
> 1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
(dbx) down
prnt: 52 fprintf(stdout,"%3d. (%3d) %s",
(dbx) where
> 0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
   1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
(dbx) func 1
main 47 prnt(&line1)
(dbx)
```

① 1 レベル上げる。

② 1 レベル下げる。

③ main に移動する。

5.6.3 現在のソース・ファイルの変更

`file` コマンドは、現在のソース・ファイルの表示、あるいは現在のソース・ファイルの変更を行います。

注意

[illegible]

file FILE	現在のファイルを指定されたファイルに変更する。
-----------	-------------------------

```
(dbx) file
sam.c 1
(dbx) file data.c
(dbx) file
data.c 2
(dbx)
```

- 1 現在のファイル
- 2 新しいファイル

次に `list` コマンドの形式を示します。

`list EXP1,EXP2` `EXP1` から `EXP2` までの行をリストする。

<code>list EXP:INT</code>	指定された行 (EXP) から指定された行 (INT) だけをリストする。(<code>\$listwindow</code> を無効にする。)
<code>list PROCEDURE</code>	<code>\$listwindow</code> の行数分だけ、指定されたプロシージャをリストする。

次の例では、49 行目から 2 行リストしています。

例

```
(dbx) list 49:2
      49 void prnt(pline)
      50 LINETYPE *pline;
```

`list` コマンドの定義済み別名 `w` を使用した場合は、次のように出力されます。

```
(dbx) w
      45 prnt(&line1);
      46 }
      47 }
      48
      49 void prnt(pline)
>     50 LINETYPE *pline;
      51 {
*     52 fprintf(stdout,"%3d. (%3d) %s",pline->linenumber,
      53 pline->length, pline->string);
      54 fflush(stdout);
```

右向きの山カッコ (>) は現在の行を、アスタリスク (*) はこのアクティブ化レベルのプログラム・カウンタ (pc) の位置を示します。

5.6.5 ソース・ファイル・テキストの探索

/ コマンドおよび ? コマンドは、ソース・コード内で正規表現を探索します。スラッシュ (/) は順方向に探索し、疑問符 (?) は現在の行から逆方向に探索します。いずれのコマンドも、必要に応じてファイルの終わりの部分でラップ・アラウンドを行います。ラップ・アラウンドでは、開始ポイントから再び開始ポイントに戻るまで、ファイル全体の探索を行います。`dbx` 変数 `$casesense` を非ゼロ値に設定した場合は、`dbx` は大文字と小文字を区別します。

次の表に / および ? コマンドの形式を示します。

/ [REGEXP]

指定された正規表現があるかどうか，コード内を順方向に探索する。式が指定されなかった場合には，現在の 1 つ前の探索コマンドで指定された正規表現を探索する。

? [REGEXP]

指定された正規表現があるかどうか，コード内を逆方向に探索する。

例

```
(dbx) /lines
no match
(dbx) /line1
16  LINETYPE line1;
(dbx) /
39  while(fgets(line1.string, sizeof(line1.string), fd) != NULL){
(dbx)
```

5.6.6 dbx 内からのソース・ファイルの編集

edit コマンドを使用すると，dbx 内からソース・コードを変更することができます。変更内容を有効にするには，dbx を終了してプログラムを再コンパイルし，dbx を再起動しなければなりません。

次に edit コマンドの形式を示します。

edit 現在のファイルでエディタを呼び出す。

edit FILE 指定されたファイルでエディタを呼び出す。

edit コマンドは，環境変数 EDITOR によって指定されるエディタをロードします。EDITOR が設定されていない場合は，vi エディタが使用されます。dbx に戻るには，エディタを終了してください。

5.6.7 同じ名前の変数の識別

which コマンドおよび whereis コマンドは，プログラム変数を表示します。これらのコマンドは，異なる有効範囲で複数の同じ名前の変数を含むプログラムのデバッグに有効です。この 2 つのコマンドは，5.3.1 項で説明した規則に従います。

次に which コマンドおよび whereis コマンドの形式を示します。

`which VAR` 変数の省略時のバージョンを表示する。

`whereis VAR` 指定された変数のすべてのバージョンを表示する。

次の例では、省略時の `i` 変数の設定を調べ、次に `i` 変数がプログラム内で 1 つしか定義されていないことを確認しています。

```
(dbx) which i
sam.main.i
(dbx) whereis i
sam.main.i
```

5.6.8 変数およびプロシージャのタイプの確認

`whatis` コマンドは、プログラムの変数およびプロシージャの型宣言をリストします。

次に `whatis` コマンドの形式を示します。

`whatis VAR` 指定された変数またはプロシージャの型宣言を表示する。

例

```
(dbx) whatis main
int main(argc,argv)
int argc;
unsigned char **argv;
(dbx) whatis i
int i;
(dbx)
```

5.7 プログラムの制御

この節では、プログラムを実行するために使用する `dbx` コマンドについて説明します。これらのコマンドでは、プログラムの実行、ソース・コードの行単位の実行、プロシージャ呼び出しからの戻り、指定行からの開始、ブレークポイントで停止した後の再開、プログラム変数への値の割り当て、実行可能なディスク・ファイルのパッチ、特定ルーチンの実行、環境変数の設定、およびシェアード・ライブラリのロードなどを実行することができます。

5.7.1 プログラムの実行および再実行

`run` および `rerun` コマンドは、プログラムの実行を開始します。いずれのコマンドにも引数を指定し、それをプログラムに渡すことができます。`run` コマンドに引数が指定されていない場合は、`dbx` は引数なしでプログラムを実行します。`rerun` コマンドに引数が指定されていない場合は、前回の `run` あるいは `rerun` コマンド実行時の引数が使用されます。`rerun` コマンドを実行する前に、`args` コマンドを使用して前もって引数を指定しておくこともできます。`args` コマンドで指定された引数は、次の `run` コマンドでは無視されます。

これらのコマンドは、C シェルのリダイレクションと同様の方法で、プログラム入出力をリダイレクトすることができます。

- オプションのパラメータ `<FILE1` は、入力を指定されたファイルから実行プログラムにリダイレクトする。
- オプションのパラメータ `>FILE2` は、出力を実行プログラムから指定されたファイルにリダイレクトする。
- オプションのパラメータ `>&FILE2` は、`stderr` および `stdout` を指定されたファイルにリダイレクトする。

注意

この出力内容は、`record output` コマンドで保存した出力内容とは異なります。`record output` コマンドは、プログラム出力ではなくデバッグ出力をファイルに保存します。`record output` コマンドの詳細については、5.9.4.2 項を参照してください。

次に `run` コマンド、`rerun` コマンドおよび `args` コマンドの形式を示します。

```
run [ARG1 ... ARGN] [<FILE1] [>FILE2]
run [ARG1 ... ARGN] [<FILE1] [>&FILE2]
```

指定された引数およびリダイレクションでプログラムを実行する。

```
args [ARG1 ... ARGN] [<FILE1] [>FILE2]
args [ARG1 ... ARGN] [<FILE1] [>&FILE2]
```

以降のコマンドで使用するために、指定した引数およびリダイレクションを設定する。設定した値は、run および rerun コマンドで新たな値が指定されるまで有効。

```
rerun [ARG1 ... ARGN] [<FILE1] [>FILE2]
rerun [ARG1 ... ARGN] [<FILE1] [>&FILE2]
```

前回に指定した引数またはリダイレクションでプログラムを再実行する。

例

```
(dbx) run sam.c 1
0. (19)#include <stdio.h>
1. (14) struct line {
2. (19) char string[256];
:
:

Program terminated normally
(dbx) rerun 2
0. (19)#include <stdio.h>
1. (14) struct line {
2. (19) char string[256];
:
:

Program terminated normally
(dbx)
```

1 引数は sam.c です。

2 前回指定した引数でプログラムを再実行します。

5.7.2 step コマンドによるプログラムの実行

高級言語で作成されているプログラムに対しては、step コマンドおよび next コマンドを使用することによって、指定した行数のソース・コードを実行することができます。アセンブリ言語で作成されているプログラムに対しては、stepi コマンドおよび nexti コマンドを使用します。ただしこの場合、EXP に指定する数はプログラムの行数ではなく機械語命令の数に

なります。EXP を指定しない場合は、dbx はソース・コードを 1 行あるいは 1 機械語命令だけ実行します。

EXP に値を指定する際の規則は次のとおりです。

- dbx デバッガは、コメント行については EXP を解釈する際に無視します。コメント行は EXP で指定した数に含まれません。
- step および stepi に対して指定した EXP は、現在のプロシージャおよび呼び出されるプロシージャに適用されます。EXP で指定しただけのソース・コードを現在のプロシージャおよび呼び出されたプロシージャで実行した後、プログラムは停止します。
- next および nexti に対して指定した EXP は、現在のプロシージャのみに適用されます。EXP で指定しただけのソース行を現在のプロシージャで実行した後、プログラムは停止します。呼び出されたプロシージャで実行するソース・コードは EXP で指定した行数には含まれません。

次に step/stepi および next/nexti コマンドの形式を示します。

step [EXP]

stepi [EXP]

指定した数のソース・コード行あるいは機械語命令を実行する。EXP の値は、現在のプロシージャおよび呼び出されるプロシージャの両方に適用される。省略時の値は 1。

next [EXP]

nexti [EXP]

指定した数のソース・コード行あるいは機械語命令を実行する。EXP の値は、現在のプロシージャのみに適用される。省略時の値は 1。

例

```
(dbx) rerun
[7] stopped at [prnt:52,0x120000c04] fprintf(stdout,"%3d.(%3d) %s",
(dbx) step 2
0. ( 19) #include <stdio.h>
    [prnt:55 ,0x120000c48]  }
(dbx) step
    [main:40 ,0x120000b40]      i=strlen(line1.string);
(dbx)
```


`$break_during_step` および `$printwhilestep` 変数の値が、
`step/stepi` および `next/nexti` コマンドの動作に影響を与えます。詳細
については表 5-8 を参照してください。

5.7.3 return コマンド

`return` コマンドは、呼び出されたプロシージャ内で使用されます。`return`
コマンドは、プロシージャ内の残りの命令を実行し、呼び出しプロシ
ージャへ戻ります。

次に `return` コマンドの形式を示します。

<code>return</code>	現在のプロシージャの残りの命令を実行し、呼び出 しプロシージャの次の行に戻って停止する。
---------------------	---

<code>return PROCEDURE</code>	現在のプロシージャ、および現在のプロシージャ と <code>PROCEDURE</code> で指定されたプロシージャの間に介 在する呼び出しプロシージャの残りの命令を実行 して、指定されたプロシージャ内の呼び出しポイ ントで停止する。
-------------------------------	---

例

```
(dbx) rerun
[7] stopped at [prnt:52,0x120000c04] fprintf(stdout,"%3d.(%3d) %s",
(dbx) return
0. (19) #include <stdio.h>
stopped at [main:45 +0xc,0x120000bb0] prnt(&line1);
(dbx)
```

5.7.4 コード内の特定の場所への移動

`goto` コマンドは、指定された行に移動して実行を続けます。このコマ
ンドは、`when` 文で、問題のあることが分かっている行を飛び越す場合な
どに便利です。

次に `goto` コマンドの形式を示します。

<code>goto LINE</code>	実行を続行するとき、指定されたソース行に移動 する。
------------------------	-------------------------------

例

```
(dbx) when at 40 {goto 43}
[8] start sam.c:43 at "sam.c":40
(dbx)
```

5.7.5 ブレークポイント後のプログラム実行の再開

高級言語で作成されているプログラムをデバッグしている場合、ブレークポイントの後にプログラムの実行を再開するには `cont` コマンドを使用します。アセンブリ言語で作成されているプログラムをデバッグしている場合は、`conti` コマンドを使用します。

次に `cont` および `conti` コマンドの形式を示します。

```
cont
conti
```

現在のソース・コード行あるいは機械語コード・アドレスから続行する。

```
cont to LINE
conti to ADDRESS
```

指定されたソース行あるいは機械語コード・アドレスまで続行する。

```
cont in PROCEDURE
conti in PROCEDURE
```

指定されたプロシージャまで続行する。

```
cont SIGNAL
conti SIGNAL
```

指定されたシグナルを受信した後、現在の行あるいは機械語命令から続行する。

```
cont SIGNAL to LINE
conti SIGNAL to ADDRESS
```

指定されたシグナルを受信した後、指定された行あるいはアドレスに到達するまで続行する。

```
cont SIGNAL in PROCEDURE
conti SIGNAL in PROCEDURE
```

指定されたプロシージャに到達するまで続行し、シグナルを送信する。

C プログラムでの `cont` コマンドの使用例を次に示します。

```
(dbx) stop in prnt
[9] stop in prnt
(dbx) rerun
[9] stopped at [prnt:52,0x120000c04] fprintf(stdout,"%3d.(%3d) %s",
(dbx) cont
0. ( 19) #include <stdio.h>
[9] stopped at [prnt:52,0x120000c04] fprintf(stdout,"%3d.(%3d) %s",
(dbx)
```

アセンブリ言語プログラムでの `conti` コマンドの使用例を次に示します。

```
(dbx) conti
0. ( 19) #include <stdio.h>
[4] stopped at >*[prnt:52 ,0x120000c04]          ldq    r16,-32640(gp)
(dbx)
```

5.7.6 プログラム変数値の変更

`assign` コマンドは、プログラム変数の値を変更します。

次に `assign` コマンドの形式を示します。

```
assign VAR = EXP
assign EXP1 = EXP2
```

`VAR` で指定したプログラム変数に新しい値を割り当てる。または、`EXP1` に指定したアドレスに新しい値を割り当てる。

例

```
(dbx) print i
19
(dbx) assign i = 10
10
(dbx) assign *(int *)0x444 = 1
1
(dbx)
```

- ① `i` の値。
- ② `i` の新しい値。
- ③ アドレスが整数となるように設定し、値 1 を割り当てる。

5.7.7 実行可能なディスク・ファイルのパッチ

`patch` コマンドは、実行可能なディスク・ファイルにパッチを当てて、誤ったデータまたは命令を訂正します。テキスト、初期化されたデータ、

または読み取り専用データ領域のみにパッチを当てることができます。
bss セグメントは、ディスク・ファイルに存在しないためパッチを当てることはできません。

patch コマンドは、実行中のプログラムに対して実行されると、異常終了します。

次に patch コマンドの形式を示します。

```
patch VAR = EXP  
patch EXP1 = EXP2
```

VAR に指定したプログラム変数あるいは EXP1 に指定したアドレスに新しい値を割り当てる。

パッチは省略時のディスク・ファイルに適用されます。省略時のディスク・ファイル以外のファイルにパッチを指定するために、修飾変数名を使用することもできます。このようにパッチを当てることによって、パッチを当てるファイルのメモリ内イメージにもパッチを当てることができます。

例

```
(dbx) patch &main = 0  
(dbx) patch var = 20  
(dbx) patch &var = 20  
(dbx) patch 0xnnnnnn = 0xnnnnnn
```

5.7.8 特定のプロシージャの実行

現在行のポインタをプロシージャの先頭に設定し、ブレークポイントをプロシージャの最後に置いて、プロシージャを実行することができます。ただし、通常は、call あるいは print コマンドを使用してプログラム内でプロシージャを実行するほうが簡単です。call あるいは print コマンドは、コマンド行で指定したプロシージャを実行します。call あるいは print コマンドの引数としてパラメータを指定すると、パラメータをルーチンに渡すことができます。

call あるいは print コマンドは、プログラムの流れには影響を与えません。プロシージャに制御が戻ると、プログラムは call あるいは print コマンドを入力した位置で停止しています。print コマンドは呼び出したプロシージャから返される値を表示しますが、call コマンドの場合は表示しません。

次に call コマンドおよび print コマンドの形式を示します。

```
call PROCEDURE([parameters])
print PROCEDURE([parameters])
```

指定したプロシージャまたは関数に対応するオブジェクト・コードを実行する。指定したパラメータは、指定プロシージャまたは関数に渡される。

例

```
(dbx) stop in prnt 1
[11] stop in prnt
(dbx) call prnt(&line1) 2
[11] stopped at [prnt:52,0x120000c] fprintf(stdout,"%3d.(%3d) %s",
(dbx) status 3
[11] stop in prnt
[12] stop at "sam.c":40
[2] record output example2 (126 lines)
(dbx) delete 11,12 4
(dbx)
```

- 1 stop コマンドで `prnt()` 関数にブレークポイントを設定します。
- 2 call コマンドで `prnt()` に対応するオブジェクト・コードを実行します。参照により、`line1` 引数は文字列を `prnt` に渡します。
- 3 status コマンドで現在有効なブレークポイントを表示します。
- 4 delete コマンドで 52 行と 40 行のブレークポイントを削除します。

`print` コマンドでは、出力する式にプロシージャを使用することができます。次に例を示します。

```
(dbx) print sqrt(2.)+sqrt(3.)
```

5.7.9 環境変数の設定

環境変数を設定するには、`setenv` コマンドを使用します。このコマンドを使用して既存の環境変数の値を設定したり、新しい環境変数を作成したりすることができます。環境変数は、`dbx` および `dbx` の制御のもとに実行しているプログラムで表示可能ですが、`dbx` 環境を終了した後は表示できません。ただし、`dbx` 内で `sh` コマンドを使用してシェルを起動している場合には、そのシェルは `dbx` 環境変数を参照できます。プロセスの環境変数を変更するには、`dbx` 内で `run` コマンドを使用してプロセスを開始する前に、`setenv` コマンドを入力する必要があります。

次に `setenv` コマンドの形式を示します。

```
setenv VAR "STRING"
```

既存の環境変数の値を変更するかまたは新たに値を設定する。環境変数をリセットするには、空文字列を指定する。

例

```
(dbx) setenv TEXT "sam.c"
(dbx) run
[4] stopped at [prnt:52,0x120000e34] fprintf(stdout,"%3d.(%3d) %s",
(dbx) setenv TEXT ""
(dbx) run
Usage: sam filename

Program exited with code 1
```

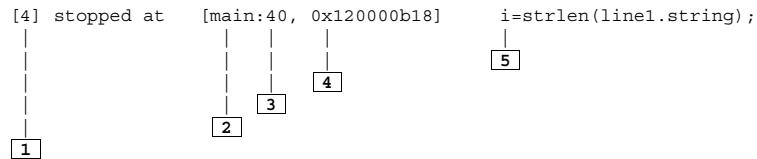
- ❶ `setenv` コマンドで環境変数 `TEXT` を `sam.c` に設定します。
- ❷ `run` コマンドでプログラムを初めから実行します。プログラムは、環境変数 `TEXT` に指定されたファイルから入力を読み取ります。プログラムは 52 行目のブレークポイントで停止します。
- ❸ `setenv` コマンドで環境変数 `TEXT` をヌルに設定します。
- ❹ `run` コマンドでプログラムを実行します。 `TEXT` 環境変数にヌル値が指定されているため、入力ファイルを指定する必要があります。

5.8 ブレークポイントの設定

ブレークポイントは、プログラムの実行を一時的に停止し、その時点でプログラムの状態を検査できるようにします。この節では、`dbx` コマンドを使用して、特定行またはプロシージャ内にブレークポイントを設定したり、各シグナルによってプログラムの実行を停止する方法について説明します。

5.8.1 概要

プログラムがブレークポイントで停止すると、デバッガが情報メッセージを表示します。たとえば、`main()` プロシージャの 23 行目にあるサンプル・プログラム `sam.c` にブレークポイントが設定されると、次のメッセージが表示されます。



① ブレークポイント状態番号

② プロシージャ名

③ 行番号

④ 現在のプログラム・カウンタ

この番号によって示されるポイントからアセンブリ言語命令を表示します。 5.7.5 項を参照。

⑤ ソース行

複数のソース・ファイルがあるプログラムにブレークポイントを設定する前には、正しいファイルにブレークポイントを設定していることを確認します。正しいプロシージャを選択するには、次の手順を実行してください。

1. `file` コマンドを使用してソース・ファイルを選択する。
2. `func` コマンドを使用してプロシージャ名を指定する。
3. `list` コマンドでファイルまたはプロシージャの行をリストする。
5.6.4 項を参照。
4. `stop at` コマンドを使用して必要な行にブレークポイントを設定する。

5.8.2 stop および stopi によるブレークポイントの設定

高級言語で作成されているプログラムのデバッグの場合、`stop` コマンドは、次のように実行を停止するためのブレークポイントを設定します。つまり、変数が変更されるか、あるいは指定された条件が真のときに、特定のプロシージャの特定の行で、プログラムの実行を停止させることができます。アセンブリ言語で作成されているプログラムのデバッグの場合は、`stopi` コマンドを使用します。ただし、`stopi` コマンドは、プログラム行の代わりに機械語命令をトレースします。`$stop_on_exec` 定義済み変数を設定することにより、`dbx` に指示して、`exec()` によって起動された新しいイメージに入ると、停止するように設定することもできます (表 5-8 を参照)。

- `stop at` コマンドおよび `stopi at` コマンド

特定のソース・コード行あるいは機械語コード・アドレスにブレークポイントを設定します。dbx デバッガは、実行可能なコードのある行あるいはアドレスでのみ停止します。実行不可能な行あるいはアドレスを指定した場合、dbx は次の実行可能ポイントにブレークポイントを設定します。VAR パラメータを指定すると、デバッガは VAR に変更があるときのみ、変数を表示してプログラムの実行を停止します。if EXP を指定すると、EXP が真のときのみ dbx が停止します。

- `stop in` コマンドおよび `stopi in` コマンド

プロシージャの先頭にブレークポイントを設定します。または条件付きでプロシージャ中にブレークポイントを設定します。

- `stop if` コマンドまたは `stopi if` コマンド

指定した条件が真のときに、プログラムの実行を停止します。dbx が各行の実行後にそのつど条件の検査を行うため、プログラムの実行速度は、大幅に遅くなります。できる限り、stop/stopi if コマンドではなく stop/stopi at または stop/stopi in を使用してください。

- `$stop_on_exec` 定義済み変数が 1 に設定されている場合

`exec()` 呼び出しにより、dbx が停止して新しいイメージのシンボル・テーブルで読み取りを行った後、イメージのメイン起動ポイントまで進み、ユーザ入力のために停止するようにします。

`delete` コマンドは、`stop` あるいは `stopi` コマンドで設定したブレークポイントを削除します。

次に `stop` コマンド および `stopi` コマンドの形式を示します。

```
stop VAR
stopi VAR
```

VAR の値に変更があると停止する。

```
stop VAR at LINE
stopi VAR at ADDRESS
```

指定したソース・コード行あるいは機械語コード・アドレスで VAR の値に変更があると停止する。


```
stop VAR at LINE if EXP
stopi VAR at ADDRESS if EXP
```

式が真の場合に限り，指定した行あるいはアドレスで VAR の値に変更があると停止する。

```
stop if EXP
stopi if EXP
```

EXP が真のとき停止する。

```
stop VAR if EXP
stopi VAR if EXP
```

VAR の値に変更があり，EXP が真のとき停止する。

```
stop in PROCEDURE
stopi in PROCEDURE
```

プロシージャの先頭で停止する。

```
stop VAR in PROCEDURE
```

VAR の値に変更があると，指定されたプロシージャの中で停止する。

```
stop VAR in PROCEDURE if EXP
stopi VAR in PROCEDURE if EXP
```

EXP が真の場合，VAR の値に変更があると，指定されたプロシージャの中で停止する。

注意

VAR および EXP をともに指定すると，プロシージャの最初だけでなく他の箇所でも，停止する可能性があります。この機能を使用すると，デバッガが各ソース行の実行前と後に条件の検査を行わなければならないため，実行速度が遅くなります。なお，両方の引数が指定されている場合は，EXP は常に VAR の前に検査されます。

C プログラムでの stop コマンドの使用例を次に示します。

```
(dbx) stop at 52
[3] stop at "sam.c":52
(dbx) rerun
[3] stopped at [prnt:52,0x120000fb0] fprintf(stdout,"%3d.(%3d) %s",
(dbx) stop in prnt
[15] stop in prnt
(dbx)
```

stopi コマンドの使用例を次に示します。

```
(dbx) stopi at 0x120000c04
[4] stop at 0x120000c04
(dbx) rerun
[7] stopped at >*[prnt:52 ,0x120000c04] ldq r16, -32640(gp)
```

5.8.3 実行中の変数のトレース

高級言語で作成されているプログラムのデバッグの場合は、trace コマンドを使用して、プログラム実行中に変数の値をリストし、トレースしている変数の有効範囲を決定することができます。アセンブリ言語で作成されているプログラムのデバッグの場合は、tracei コマンドを使用します。ただし、tracei コマンドは、プログラム行の代わりに機械語命令でトレースします。

次に trace および tracei コマンドの形式を示します。

```
trace LINE
```

指定のソース行を実行するたびにリストする。

```
trace VAR
tracei VAR
```

各ソース行あるいは機械語命令の実行後に、指定の変数をリストする。

```
trace [VAR] at LINE
tracei [VAR] at ADDRESS
```

指定した行あるいは命令の指定した変数をリストする。

```
trace [VAR] in PROCEDURE
tracei [VAR] in PROCEDURE
```

指定したプロシージャの指定した変数をリストする。

```
trace [VAR] at LINE if EXP
tracei [VAR] at ADDRESS if EXP
```

式が真である場合に、指定したソース・コード行あるいは機械語コード・アドレスの変数の値に変更があるとき、その変数をリストする。
EXP は VAR より先に検査される。

```
trace [VAR] in PROCEDURE if EXP
tracei [VAR] in PROCEDURE if EXP
```

式が真である場合に、指定されたプロシージャの変数の値に変更があるとき、その変数をリストする。EXP は VAR より先に検査される。

例

```
(dbx) trace i
[5] trace i in main
(dbx) rerun sam.c
[4] [main:25 ,0x400a50]
(dbx) c
[5] i changed before [main: line 41]:
    new value = 19;
[5] i changed before [main: line 41]:
    old value = 19;
    new value = 14;
[5] i changed before [main: line 41]:
    old value = 14;
    new value = 19;
[5] i changed before [main: line 41]:
    old value = 19;
    new value = 13;
[5] i changed before [main: line 41]:
    old value = 13;
    new value = 17;
[5] i changed before [main: line 41]:
    old value = 17;
    new value = 3;
[5] i changed before [main: line 41]:
    old value = 3;
    new value = 1;
[5] i changed before [main: line 41]:
    old value = 1;
    new value = 30;
```

5.8.4 dbx での条件コードの記述

when コマンドは、指定されたある dbx コマンドを実行する条件を制御します。

次に when コマンドの形式を示します。

```
when VAR [if EXP] {COMMAND_LIST}
```

EXP が真で、VAR の値に変更があると、指定したコマンド・リストを実行する。

```
when [VAR] at LINE [if EXP] {COMMAND_LIST}
```

EXP が真で、VAR の値に変更があり、かつデバッガが LINE の行に到達すると、コマンド・リストを実行する。

```
when in PROCEDURE {COMMAND_LIST}
```

PROCEDURE に入ると、コマンド・リストを実行する。

```
when [VAR] in PROCEDURE [if EXP] {COMMAND_LIST}
```

EXP が真で、VAR の値に変更があると、PROCEDURE の各行に対して、指定されたコマンドを実行する。EXP は VAR の前に検査される。

例

```
(dbx) when in prnt {print line1.length}
[6] print line1.length in prnt
(dbx) rerun
19
14
19
:
:
17
59
45
12
More (n if no)?
(dbx) delete 6
(dbx) when in prnt {stop}
[7] stop in prnt
(dbx) rerun
[7] stopped at [prnt:52,0x12000fb0] fprintf(stdout,"%3d.(%3d) %s") 2
```

1 line1.length の値

2 プロシージャ prnt 内で停止します。

5.8.5 シグナルの受信および無視

`catch` コマンドは、`dbx` が受信するシグナルをリストしたり、受信するシグナルを指定したりする場合に使用します。プロセスが指定のシグナルを検出すると、`dbx` はプロセスを停止します。

`ignore` コマンドは、`dbx` が受信しないシグナルをリストしたり、無視リストに追加するシグナルを指定するために使用します。

次に `catch` および `ignore` コマンドの形式を示します。

<code>catch</code>	<code>dbx</code> が受信するすべてのシグナルのリストを表示する。
<code>catch SIGNAL</code>	シグナルを受信リストに追加する。
<code>ignore</code>	<code>dbx</code> が受信しないすべてのシグナルのリストを表示する。
<code>ignore SIGNAL</code>	指定したシグナルを受信リストから削除し、無視リストに追加する。

例

```
(dbx) catch 1
INT QUIT ILL TRAP ABRT EMT FPE BUS SEGV SYS PIPE TERM URG \
STOP TTIN TTOU IO XCPU XFSZ VTALRM PROF WINCH INFO USR1 USR2
(dbx) ignore 2
HUP KILL ALRM TSTP CONT CHLD
(dbx) catch kill 3
(dbx) catch
INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE TERM URG \
STOP TTIN TTOU IO XCPU XFSZ VTALRM PROF WINCH INFO USR1 USR2
(dbx) ignore
HUP ALRM TSTP CONT CHLD
(dbx)
```

1 受信リストを表示します。

2 無視リストを表示します。

3 `KILL` を受信リストに追加し、無視リストから削除します。

前述の例中のバックスラッシュは、行の継続を表しています。catch および ignore の実際の出力は、1 行です。

5.9 プログラム状態の検査

dbx がブレークポイントで停止すると、プログラム状態を検査して問題のありそうな箇所を検査することができます。dbx デバッガには、スタック・トレース、変数値、およびレジスタ値を表示する dbx コマンドがあります。スタック・トレースに記録されているアクティブ化レベルについての情報を表示したり、アクティブ化レベルを上下に変更するコマンドもあります (5.6.2 項を参照)。

5.9.1 変数および式の値の出力

print コマンドは、1 つまたは複数の式の値を表示します。

printf コマンドは、指定されたフォーマットで情報をリストします。printf コマンドは、%s を除いて、printf() 関数でサポートするすべてのフォーマットをサポートします。フォーマットの一覧については、printf(3) を参照してください。printf コマンドは、変数の値を別の基数で参照することができます。

コマンド別名リスト (5.5.3 項参照) には、8 進数 (po)、10 進数 (pd)、および 16 進数 (px) の各基数で変数の値を表示できる有効な別名が用意されています。省略時の基数は 10 進数です。

インクルード・ファイル regdefs.h にある、実際のマシン・レジスタ名あるいはソフトウェア名のどちらを指定することもできます。レジスタ番号の前に指定される接頭語は、レジスタのタイプを示します。次の表に示すように、接頭語は \$f または \$r のいずれかです。

レジスタ名	レジスタ・タイプ
\$f00-\$f31	浮動小数点レジスタ (32 の 1)
\$r00-\$r31	マシン・レジスタ (32 の 1)
\$fpcr	浮動小数点制御レジスタ
\$pc	プログラム・カウンタ値
\$ps	プログラム状態レジスタ ^a

^aプログラム状態レジスタはカーネルのデバッグに便利です。ユーザ・レベル・プログラムの場合は、値は常に 8 です。

接頭語の付いたレジスタを `print` コマンドで指定して、レジスタ値やプログラム・カウンタを表示することができます。次のコマンドは、マシン・レジスタ 3 およびプログラム・カウンタの値を表示します。

```
(dbx) print $r3
(dbx) print $pc
```

次に `print` コマンドの形式を示します。

```
print EXP1,...,EXPN
```

指定された式の値を表示する。

```
printf "STRING", EXP1,...,EXPN
```

`STRING` に指定されたフォーマットで指定された式の値を表示する。

注意

`dbx` キーワードと同一の名前が式にある場合は、その名前をカッコで囲む必要があります。たとえば、`playback` コマンドと `record` コマンドのキーワードである `output` をプリントするときは、次のように指定します。

```
(dbx) print (output)
```

例

```
(dbx) print i
14 [1]
(dbx) po i
016 [2]
(dbx) px i
0xe [3]
(dbx) pd i
14 [4]
(dbx)
```

[1] 10 進数

[2] 8 進数

[3] 16 進数

4 10 進数

`printregs` コマンドはすべてのレジスタ値の一覧を表示します。このコマンドには引数は不要です。 `print` コマンドと同じように、省略時の設定では、`printregs` が表示する値は 10 進数です。16 進で値を表示する場合は、`dbx` 変数 `$hexints` を設定します。

例

```
(dbx) printregs
$vpf= 4831837712          $r0_v0=0
$r1_t0=0                  $r2_t1=0
$r3_t2=18446744069416926720 $r4_t3=18446744071613142936
$r5_t4=1                  $r6_t5=0
:
$fp25= 0.0                $fp26= 0.0
$fp27= 2.3873098155006918e-314 $fp28= 2.6525639909000367e-314
$fp29= 9.8813129168249309e-324 $fp30= 2.3872988413145664e-314
$fp31= 0.0                $pc= 4831840840
```

5.9.2 dump コマンドによるアクティブ化レベル情報の表示

`dump` コマンドは、アクティブ化レベルについての情報を表示します。指定されたアクティブ化レベルにローカルな変数の値もすべて表示されます。プログラムで現在実行中のアクティブ化レベルを調べるには、`where` コマンドでスタック・トレースを行います。

次に `dump` コマンドの形式を示します。

<code>dump</code>	現在のアクティブ化レベルの情報を表示する。
<code>dump .</code>	すべてのアクティブ化レベルの情報を表示する。
<code>dump PROCEDURE</code>	指定されたプロシージャ (アクティブ化レベル) についての情報を表示する。

例

```
(dbx) where
> 0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
(dbx) dump
prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]
(dbx) dump .
```



```

> 0 prnt(pline = 0x11ffffcb8) ["sam.c":52, 0x120000c04]

1 main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
line1 = struct {
    string = "#include <stdio.h>"
    length = 19
    linenumber = 0
}
fd = 0x140000158
fname = 0x11ffffe9c = "sam.c"
i = 19
curlinenumber = 1

(dbx) dump main
main(argc = 2, argv = 0x11ffffe08) ["sam.c":45, 0x120000bac]
line1 = struct {
    string = "#include <stdio.h>"
    length = 19
    linenumber = 0
}
fd = 0x140000158
fname = 0x11ffffe9c = "sam.c"
i = 19
curlinenumber = 1
(dbx)

```

5.9.3 メモリの内容の表示

メモリの内容は、アドレスおよび表示フォーマットを指定することによって表示することができます。

コマンドの形式は次のとおりで、3つの部分の間にはスペースは不要です。

address/count mode

address には表示する最初の項目のアドレス、*count* には表示する項目の数、*mode* には項目の表示フォーマットを指定します。次に例を示します。

`prnt/20i`

この例では、`prnt` 関数のアドレスで始まる 20 の機械語命令の内容を表示します。

mode に指定できる値を表 5-9 に示します。

表 5-9: メモリ・アドレス表示モード

モード	表示フォーマット
b	8 進数でバイトの内容を表示する。
c	1 バイト文字として表示する。
D	10 進数でロングワード (64 ビット) を表示する。
d	10 進数でショートワード (16 ビット) を表示する。
dd	10 進数でワード (32 ビット) を表示する。
f	単精度実数を表示する。
g	倍精度実数を表示する。
i	機械語命令を表示する。
O	8 進数でロングワードを表示する。
o	8 進数でショートワードを表示する。
oo	8 進数でワード (32 ビット) を表示する。
s	ヌル・バイトで終わる文字列を表示する。
X	16 進数でロングワードを表示する。
x	16 進数でショートワードを表示する。
xx	16 進数でワード (32 ビット) を表示する。

メモリのアドレスを命令として表示したときの出力は次の例のようになります。

```
(dbx) &prnt/20i
[prnt:51, 0x120000bf0] ldah gp, 8193(r27)
[prnt:51, 0x120000bf4] lda gp, -25616(gp)
[prnt:51, 0x120000bf8] lda sp, -64(sp)
[prnt:51, 0x120000bfc] stq r26, 8(sp)
[prnt:51, 0x120000c00] stq r16, 16(sp)
[prnt:52, 0x120000c04] ldq r16, -32640(gp)
>* [prnt:52, 0x120000c08] addq r16, 0x38, r16
[prnt:52, 0x120000c0c] ldq r17, -32552(gp)
[prnt:52, 0x120000c10] ldq r1, 16(sp)
[prnt:52, 0x120000c14] ldl r18, 260(r1)
[prnt:52, 0x120000c18] ldl r19, 256(r1)
[prnt:52, 0x120000c1c] bis r1, r1, r20
[prnt:52, 0x120000c20] ldq r27, -32624(gp)
[prnt:52, 0x120000c24] jsr r26, (r27), 0x4800030a0
[prnt:52, 0x120000c28] ldah gp, 8193(r26)
[prnt:52, 0x120000c2c] lda gp, -25672(gp)
```

```
[prnt:54, 0x120000c30] ldq r16, -32640(gp)
[prnt:54, 0x120000c34] addq r16, 0x38, r16
[prnt:54, 0x120000c38] ldq r27, -32544(gp)
[prnt:54, 0x120000c3c] jsr r26, (r27), 0x480003100
```

5.9.4 dbx セッションの入出力の記録および再生

dbx デバッガは、プログラムへの入力および出力を記録したり、記録した入力コマンドを再実行することができます。記録情報は、後で利用できるようにファイルに書き込まれます。

入力の記録は、コマンド・シーケンスを繰り返して実行するために利用するコマンド・ファイルの作成に有効です。また、出力の記録は、画面に表示するには多すぎる情報を保管して後で分析する場合などに便利です。出力ファイルを後で参照する場合には、記録ファイルを直接読むことも、dbx で再現することもできます。

5.9.4.1 デバッガ入力の記録および再実行

入力したデバッガ・コマンドを記録するには、`record input` コマンドを使用します。`record input` コマンドによって記録されたコマンドを繰り返すには、`playback input` コマンドを使用します。

`record input` コマンドおよび `playback input` コマンドの形式は次のとおりです。

```
record input [FILE]
```

すべての dbx コマンドをファイルに記録する。記録ファイルを指定しない場合は `/tmp` ディレクトリに一時ファイルが作成される。

```
playback input [FILE]
source [FILE]
```

指定されたファイル、またはファイル名が指定されていない場合は一時ファイルからコマンドを実行する。

一時ファイルの名前はデバッグ変数 `$defaultin` にあります。一時ファイル名を表示するには、次のように `print` コマンドを実行してください。

```
(dbx) print $defaultin
```

記録した出力を現在のデバッグ・セッションでのみ参照する必要がある場合は、一時ファイルを使用します。現在のデバッグ・セッション終了後に記録

情報を利用する場合は、ファイル名を指定して記録しておきます。記録が行われているかどうかを調べる場合は `status` コマンドを実行します。記録を中止する場合は `delete` コマンドを使用します。これらのファイルも記録されることに注意してください。後で利用するためにファイルを作成している場合は、ファイルを編集してこの種のコマンドの記録を削除しておく必要があるかもしれません。

`record input` コマンドで記録したコマンドを再実行する場合は、`playback input` コマンドを使用します。省略時の設定では、`playback input` によるコマンドの再実行はサイレント・モードで行われるため、再実行の様子は表示されません。dbx 変数 `$pimode` が 1 に設定されている場合は、コマンドの再実行の様子が表示されます。

次の例では、入力されたコマンドを記録した後、作成したファイルを表示しています。

```
(dbx) record input 1
[2] record input /tmp/dbxtX026963 (0 lines)
(dbx) status
[2] record input /tmp/dbxtX026963 (1 lines)
(dbx) stop in prnt
[3] stop in prnt
(dbx) when i = 19 {stop}
[4] stop ifchanged i = 19
(dbx) delete 2 2
(dbx) playback input 3
[3] stop in prnt
[4] stop ifchanged i = 19
[5] stop in prnt
[6] stop ifchanged i = 19
/tmp/dbxtX026963: 4: unknown event 2 4
(dbx)
```

- 1 記録を開始します。
- 2 記録を中止します。
- 3 入力コマンドを再現します。
- 4 記録を中止したときにイベント 2 (記録の開始) を削除しているため、エラー・メッセージが表示されます。

この例では、dbx コマンドで作成した一時ファイルには、次の内容が記録されています。

```
status
stop in prnt
when i = 19 {stop}
delete 2
```

5.9.4.2 デバッガ出力の記録および再実行

デバッグ・セッション中の dbx 出力を記録するには、record output コマンドを使用します。出力とともに入力も記録する場合は、dbx 変数 \$rimode を設定します。記録された情報を参照するには、playback output コマンドまたは適当なテキスト・エディタを使用します。

次に record output コマンドおよび playback output コマンドの形式を示します。

```
record output [FILE]
```

指定したファイルにすべての dbx コマンドを記録する。ファイルを省略した場合は /tmp ディレクトリに一時ファイルが作成される。

```
playback output [FILE]
```

指定されたファイル、またはファイル名が指定されない場合は一時ファイルからコマンドを表示する。

一時ファイルの名前はデバッガ変数 \$defaultout にあります。一時ファイル名を表示するには、次のように print コマンドを実行してください。

```
(dbx) print $defaultout
```

playback output コマンドは cat コマンドと同じように機能します。record output コマンドによる表示は、記録ファイルの内容と同一のものです。

記録した出力を現在のデバッグ・セッションでのみ参照する必要がある場合は、一時ファイルを使用します。現在のデバッグ・セッション終了後に記録情報を利用する場合は、ファイル名を指定して記録しておきます。記録が行われているかどうかを調べる場合は status コマンドを実行します。記録を中止する場合は delete コマンドを使用します。

次の例では、dbx コマンドの実行結果、および code というファイルに記録されたこのコマンド実行の出力を示しています。

```
(dbx) record output code
[3] record output code (0 lines)
```

```
(dbx) stop at 25
[4] stop at "sam.c":25
(dbx) run sam.c
[4] stopped at [main:25 ,0x120000a48] if (argc < 2) {
(dbx) delete 3
(dbx) playback output code
[3] record output code (0 lines)
(dbx) [4] stop at "sam.c":25
(dbx) [4] stopped at [main:25 ,0x120000a48] if (argc < 2) {
(dbx)
```

5.10 コア・ダンプ・ファイルのネーミング

この節では、オペレーティング・システムのコア・ダンプ・ファイルのネーミング機能を有効にして、複数のコア・ファイルを保存できるようにする方法について説明します。

コア・ファイルのネーミング機能を有効にすると、システムは、次の形式の名前でコア・ファイルを生成します。

core.prog-name.host-name.tag

コア・ファイル名は、ピリオドで区切った次の 4 つの部分からなります。

- 短い文字列の *core*。
- *ps* コマンドで表示される 16 文字までのプログラム名。
- 16 文字までのシステムのネットワーク・ホスト名 (ホスト名フォーマットの最初のドットの前にあるホスト名の部分から)。
- ホスト上のプログラムで生成されたすべてのコア・ファイルを一意にするためにコア・ファイルに割り当てられた数字のタグ。

このタグの最大値、つまり、このプログラムおよびホストで生成されるコア・ファイルの最大数は、システム構成パラメータで設定されます (5.10.1 項を参照)。

タグは、文字どおりのバージョン番号ではありません。システムは、コア・ファイルに対して最初に使用可能な一意のタグを選択します。たとえば、プログラムのコア・ファイルがタグの 0, 1, 3, を使用している場合、システムはそのプログラムについて作成する次のコア・ファイルには、タグ 2 を使用します。コア・ファイル・インスタンスのシステム構成上の上限に達した場合には、そのプログラムとホストの組み合わせ

に対しては、それ以上コア・ファイルは作成されません。省略時の設定では、16 バージョンまでのコア・ファイルが作成されます。

ディスク・スペースを節約するため、コア・ファイルは、チェックした後に必ず削除してください。名前を付けたコア・ファイルは上書きされないので、この作業が必要になります。

コア・ファイルのネーミング機能は、システム・レベル (5.10.1 項) でも、個々のアプリケーション・レベル (5.10.2 項) でも有効にすることができます。

5.10.1 システム・レベルでのコア・ファイルのネーミング機能の有効化

システム・レベルでのコア・ファイルのネーミング機能は、`dxkerneltuner(8X)` (グラフィカル・インタフェース) または `sysconfig(8)` ユーティリティを使用して、有効にすることができます。コア・ファイルのネーミングを有効にするには、`enhanced-core-name` プロセス・サブシステム属性を 1 に設定します。プログラムが特定のホスト・システム上に作成できる一意のコア・ファイルのバージョン数を制限するには、プロセス・サブシステム属性の `enhanced-core-max-versions` を必要な値に設定します。次の例を参照してください。

```
proc:
```

```
enhanced-core-name = 1
enhanced-core-max-versions = 8
```

バージョンの最小値，最大値，省略値はそれぞれ，1，99999，および 16 です。

5.10.2 アプリケーション・レベルでのコア・ファイルのネーミング機能の有効化

アプリケーション・レベルでコア・ファイルのネーミングを有効にするには、次の例に示すように、プログラムで `UWS_CORE` フラグを設定して `uswitch` システム・コールを使用します。

```
#include <signal.h>
#include <xsys/uswitch.h>
/*
 * Request enhanced core-file naming for
 * this process, then create a core file.
 */
main()
{
```

```

        long uval = uswitch(USC_GET, 0);
        uval = uswitch(USC_SET, uval | USW_CORE);
        if (uval < 0) {
            perror("uswitch");
            exit(1);
        }
        raise(SIGQUIT);
    }
}

```

5.11 実行中のプロセスのデバッグ

dbx デバッガを使用して、dbx 環境の外で開始された実行中のプロセスをデバッグすることができます。dbx デバッガは、`/proc` ファイル・システムによってこのようなプロセスのデバッグをサポートしており、親プロセスも子プロセスもサポートされます。実行中のプロセスのデバッグは、`/proc` ファイル・システムがマウントされている場合にのみ可能です。`/proc` ファイル・システムがマウントされていない場合は、スーパーユーザは次のコマンドを使用してマウントすることができます。

```
# mount -t procfs /proc /proc
```

`/etc/fstab` ファイルに次のエントリを追加すると、ブート時に `/proc` をマウントできます。

```
/proc          /proc    procfs rw 0 0
```

dbx デバッガは、まず `/proc` がマウントされているかどうかをチェックしますが、マウントされていない場合でも機能します。

実行中のプロセスにアタッチするためには、dbx コマンドの `attach` を使用します。`attach` コマンドの構文は次のとおりです。

```
attach process-id
```

process-id には、アタッチするプロセスのプロセス ID を指定します。

デバッグのためにプロセスにアタッチするには、コマンド行オプションの `-pid process id` を使用することもできます。

実行中のプロセスからデタッチするには、dbx コマンドの `detach` を使用します。このコマンドの構文は次のとおりです。

`detach [process-id]`

`process-id` には、デタッチするプロセスのプロセス ID を指定します。引数を省略した場合は、`dbx` は現在のプロセスをデタッチします。

あるプロセスから別のプロセスに変更する場合は、`switch` コマンドを使用します。このコマンドの構文は次のとおりです。

`switch process-id`

`process-id` には、変更先プロセスのプロセス ID を指定します。変更先プロセスは、`switch` コマンドを使用する前にアタッチしておく必要があります。`switch` コマンドに対して別名 `sw` を使用することができます。

`attach` コマンドは、まず `/proc` がマウントされているかどうかをチェックします。マウントされていない場合には、`dbx` は実行すべき動作を通知する警告メッセージを発行します。マウントされている場合には、`dbx` は `/proc` でプロセス ID を探します。`/proc` でプロセス ID が見つかったら、`dbx` はプロセスをオープンして、`stop` コマンドを発行します。プロセスがそこになければ、またはそのプロセスへのアタッチが許可されていない場合には、`dbx` はアタッチが失敗したことを通知します。

`stop` コマンドが実行されると、`dbx` は現在の位置を通知して、`dbx` プロンプトを表示し、ユーザがコマンドを入力するのを待ちます。プログラムはおそらくユーザ・コード内では直ちに停止されませんが、ユーザ・コードから呼び出されたライブラリ・コールまたはシステム・コール内で停止されます。

`detach` コマンドは現在のすべてのブレークポイントを削除して、`'run on last close'` フラグを設定し、プロセスをクローズ (リリース) します。`dbx` 内で明示的に終了されていない場合、その後プログラムは実行を継続します。

`dbx` の制御下にあるすべてのアクティブなプロセスを参照するには、`plist` コマンドを使用します。このコマンドの構文は次のとおりです。

`plist`

アクティブなプロセスとその状態を表示します。現在のプロセスに対しては、`-->` のマーカが表示されます。

5.12 マルチスレッド・アプリケーションのデバッグ

dbx デバッガは、スレッドを使用するアプリケーションのデバッグを支援する次の 3 つの基本コマンドをサポートします。

- `tlist` コマンドは、すべてのスレッドの一覧を表示し、それらが現在プログラムのどの位置にあるかを表示します。このコマンドには引数は不要です。

`tlist` コマンドを使用すると、現在プログラム内にあるすべてのスレッドを ID とともに表示することができます。

- `tset` コマンドは現在のスレッドを設定します。デバッガは、1 つのスレッドを現在のスレッドとして保持します。このスレッドが、ブレークポイントにヒットしたり、停止信号を受信して、制御を dbx に引き渡すスレッドです。

`tset` を使用すると、現在のスレッドとして異なるスレッドを選択し、通常の dbx コマンドで状態を検査できるようになります。選択したスレッドは、次に `tset` コマンドを入力するまで、現在のスレッドであることに注意してください。また、ブロックされていたり、別のスレッドに加わるために待機している場合、`continue`、`step`、または `next` コマンドはそのスレッドについて適切ではありません。

- `tstack` コマンドはアプリケーション内のすべてのスレッドのスタックをリストします。このコマンドは `where` コマンドに似ていて、引数として数値を指定することにより、表示するスタック・レベル数を制限することができます。

`tset` および `tstack` コマンドの形式は次のとおりです。

`tset` [EXP] 現在のスレッドを選択する。EXP 引数には ID を 16 進数で指定する。

`tstack` [EXP] すべてのスレッドのスタック・トレースを表示する。EXP を指定すると、dbx は上位 EXP レベルのスタックのみを表示する。この引数を省略すると、すべてのスタックを表示する。

ご使用のシステムに POSIX Threads Library がインストールされている場合は、dbx セッションで `call cma_debug()` コマンドを実行することによって、POSIX Threads Library pthread デバッガにアクセスすることができます。

まず、pthread デバッガは、プログラムのスレッドに関する多くの有用な情報を提供します。pthread デバッガの使用方法については、debug> プロンプトで help コマンドを入力してください。

サンプル・スレッド・プログラム twait.c を例 12-1 に示してあります。次の例では、dbx セッションで twait.c を使用しています。

```
% dbx twait
dbx version 3.11.6
Type 'help' for help.

main: 50 pthread_t      me = pthread_self(), timer_thread;
(dbx) stop in do_tick
[2] stop in do_tick
(dbx) stop at 85
[3] stop at "twait.c":85
(dbx) stop at 35
[4] stop at "twait.c":35
(dbx) run
1: main thread starting up
1: exit lock initialized
1: exit lock obtained
1: exit cv initialized
1: timer_thread 2 created
1: exit lock released
[2] thread 0x81c62e80 stopped at [do_tick:21 ,0x12000730c] pthread_
t      me = pthread_self();
(dbx) tlist
thread 0x81c623a0 stopped at [msg_receive_trap:74 +0x8,0x3ff808edf04]
Source not available
thread 0x81c62e80 stopped at [do_tick:21 ,0x12000730c] pthread_
t      me = pthread_self();
(dbx) where
> 0 do_tick(argP = (nil)) ["twait.c":21, 0x12000730c]
1 cma_thread_base(0x0, 0x0, 0x0, 0x0, 0x0) [".././.././../src/usr/
ccs/lib/DECThreads/COMMON/cma_thread.c":1441, 0x3ff80931410]
(dbx) tset 0x81c623a0
thread 0x81c623a0 stopped at [msg_receive_trap:74 +0x8,0x3ff808edf04]
Source not available
(dbx) where
> 0 msg_receive_trap(0x3ff8087b8dc, 0x3ffc00a2480, 0x3ff8087b928, 0x181
57f0d0d, 0x3ff8087b68c) ["/usr/build/osf1/goldos.bld/export/alpha/usr/in
clude/mach/syscall_sw.h":74, 0x3ff808edf00]
1 msg_receive(0x61746164782e, 0x3ffc009a420, 0x3ffc009a420, 0x3c20, 0
xe0420) [".././.././../src/usr/ccs/lib/libmach/msg.c":95, 0x3ff808e474
4]
2 cma_vp_sleep(0x280187f578, 0x3990, 0x7, 0x3ffc1032848, 0x0) ["../.
.././../src/usr/ccs/lib/DECThreads/COMMON/cma_vp.c":1471, 0x3ff809375
cc]
3 cma_dispatch(0x7, 0x3ffc1032848, 0x0, 0x3ffc100ee08, 0x3ff80917e3c
) [".././.././../src/usr/ccs/lib/DECThreads/COMMON/cma_dispatch.c":967
, 0x3ff80920e48]
4 cma_int_wait(0x11ffff228, 0x140009850, 0x3ffc040cdb0, 0x5, 0x3ffc0
014c00) [".././.././../src/usr/ccs/lib/DECThreads/COMMON/cma_condition
.c":2202, 0x3ff80917e38]
5 cma_thread_join(0x11ffff648, 0x11ffff9f0, 0x11ffff9e8, 0x60aaec4, 0
x3ff8000cf38) [".././.././../src/usr/ccs/lib/DECThreads/COMMON/cma_thr
ead.c":825, 0x3ff80930a58]
6 pthread_join(0x140003110, 0x40002, 0x11ffffa68, 0x3ffc040cdb0, 0x0)
[".././.././../src/usr/ccs/lib/DECThreads/COMMON/cma_pthread.c":2193,
0x3ff809286c8]
7 main() ["twait.c":81, 0x12000788c]
```

```

(dbx) tlist
thread 0x81c623a0 stopped at [msg_receive_trap:74 +0x8,0x3ff808edf04]
Source not available
thread 0x81c62e80 stopped at [do_tick:21 ,0x12000730c] pthread_
t me = pthread_self();
(dbx) tset 0x81c62e80
thread 0x81c62e80 stopped at [do_tick:21 ,0x12000730c] pthread_
t me = pthread_self();
(dbx) cont
2: timer thread starting up, argP=0x0
[4] thread 0x81c62e80 stopped at [do_tick:35 ,0x120007430] printf("
%d: wait for next tick\n", THRID(&me));
(dbx) cont
2: wait for next tick
2: TICK #1
[4] thread 0x81c62e80 stopped at [do_tick:35 ,0x120007430] printf("
%d: wait for next tick\n", THRID(&me));
(dbx) tstack
Thread 0x81c623a0:
> 0 msg_receive_trap(0x3ff8087b8dc, 0x3ffc00a2480, 0x3ff8087b928, 0x181
57f0d0d, 0x3ff8087b68c) ["/usr/build/osfl/goldos.bld/export/alpha/usr/in
clude/mach/syscall_sw.h":74, 0x3ff808edf00]
1 msg_receive(0x61746164782e, 0x3ffc009a420, 0x3ffc009a420, 0x3c20, 0
xe0420) ["/usr/build/osfl/goldos.bld/export/alpha/usr/include/mach/msg.c":95, 0x3ff808e474
4]
2 cma_vp_sleep(0x280187f578, 0x3990, 0x7, 0x3ffc1032848, 0x0) ["/usr/b
uild/osfl/goldos.bld/export/alpha/usr/include/mach/cma_vp.c":1471, 0x3ff809375
cc]
3 cma_dispatch(0x7, 0x3ffc1032848, 0x0, 0x3ffc100ee08, 0x3ff80917e3c
) ["/usr/build/osfl/goldos.bld/export/alpha/usr/include/mach/cma_dispatch.c":967
, 0x3ff80920e48]
4 cma_int_wait(0x11ffff228, 0x140009850, 0x3ffc040cdb0, 0x5, 0x3ffc0
014c00) ["/usr/build/osfl/goldos.bld/export/alpha/usr/include/mach/cma_condition
.c":2202, 0x3ff80917e38]
5 cma_thread_join(0x11ffff648, 0x11ffff9f0, 0x11ffff9e8, 0x60aaec4, 0
x3ff8000cf38) ["/usr/build/osfl/goldos.bld/export/alpha/usr/include/mach/cma_thr
ead.c":825, 0x3ff80930a58]
6 pthread_join(0x140003110, 0x40002, 0x11ffffa68, 0x3ffc040cdb0, 0x0)
["/usr/build/osfl/goldos.bld/export/alpha/usr/include/mach/pthread.c":2193,
0x3ff809286c8]
7 main() ["twait.c":81, 0x12000788c]
Thread 0x81c62e80:
> 0 do_tick(argP = (nil)) ["twait.c":35, 0x120007430]
1 cma_thread_base(0x0, 0x0, 0x0, 0x0, 0x0) ["/usr/build/osfl/goldos.bld/export/alpha/usr/in
clude/mach/cma_thread.c":1441, 0x3ff80931410]
More (n if no)?
(dbx) tstack 3
Thread 0x81c623a0:
> 0 msg_receive_trap(0x3ff8087b8dc, 0x3ffc00a2480, 0x3ff8087b928, 0x181
57f0d0d, 0x3ff8087b68c) ["/usr/build/osfl/goldos.bld/export/alpha/usr/in
clude/mach/syscall_sw.h":74, 0x3ff808edf00]
1 msg_receive(0x61746164782e, 0x3ffc009a420, 0x3ffc009a420, 0x3c20, 0
xe0420) ["/usr/build/osfl/goldos.bld/export/alpha/usr/include/mach/msg.c":95, 0x3ff808e474
4]
2 cma_vp_sleep(0x280187f578, 0x3990, 0x7, 0x3ffc1032848, 0x0) ["/usr/b
uild/osfl/goldos.bld/export/alpha/usr/include/mach/cma_vp.c":1471, 0x3ff809375
cc]
Thread 0x81c62e80:
> 0 do_tick(argP = (nil)) ["twait.c":35, 0x120007430]
1 cma_thread_base(0x0, 0x0, 0x0, 0x0, 0x0) ["/usr/build/osfl/goldos.bld/export/alpha/usr/in
clude/mach/cma_thread.c":1441, 0x3ff80931410]
(dbx) cont
2: wait for next tick
2: TICK #2
[4] thread 0x81c62e80 stopped at [do_tick:35 ,0x120007430] printf("

```

```

%d: wait for next tick\n", THRID(&me));
(dbx) assign ticks = 29
29
(dbx) cont
2: wait for next tick
2: TICK #29
[4] thread 0x81c62e80 stopped at [do_tick:35 ,0x120007430]    printf("
%d: wait for next tick\n", THRID(&me));
(dbx) cont
2: wait for next tick
2: TICK #30
2: exiting after #31 ticks
1: joined with timer_thread 2
[3] thread 0x81c623a0 stopped at [main:85 ,0x1200078ec]    if (errno
o != 0) printf("errno 7 = %d\n",errno);
(dbx) tlist
thread 0x81c623a0 stopped at [main:85 ,0x1200078ec]    if (errno != 0)
printf("errno 7 = %d\n",errno);
thread 0x81c62e80 stopped at [msg_rpc_trap:75 +0x8,0x3ff808edf10]
Source not available
(dbx) cont

Program terminated normally

(dbx) tlist
(dbx) quit

```

5.13 複数の非同期プロセスのデバッグ

dbx デバッガで複数の非同期プロセスをデバッグすることができます。非同期プロセスをデバッグしている間、dbx は状態を表示して、非同期にコマンドを受け入れることができます。非同期に実行していると、デバッガの動作が混乱しているように見えることがあります。これは、停止している別のプロセスを調べるコマンドを入力しているときに、実行プロセスが出力を画面に表示することがあるためです。

次のいずれかの状況の場合、デバッガは自動的に非同期モードになります。

- 以前のプロセスがまだアタッチされているときに、新しいプロセスにアタッチするように命じた場合
- dbx がアタッチしているプロセスが子プロセスにフォークして、デバッガが親プロセスからデタッチせずにその子プロセスに自動的にアタッチした場合

デバッガは、多数の定義済み変数を使用して、非同期デバッグの動作を定義します (表 5-8 を参照)。変数 \$asynch_interface は、カウンタとしても見ることができ、新しいプロセスがアタッチされると 1 だけ増加し、プロセスが終了するかデタッチされると 1 だけ減少します。省略時の設定は 0 です。

`$asynch_interface` の値が正で非ゼロの場合、非同期デバグは使用可能であり、変数の値が 0 (ゼロ) または負の場合、非同期デバグは使用不能です。dbx が非同期モードになるのを防ぐには、`$asynch_interface` 変数に負の値を設定します。ただし、停止されている子を親が待っている場合には、非同期モードを使用不能にすると、デバグがより困難になります。

プロセスが `fork()` または `vfork()` 呼び出しを実行すると、dbx は子プロセスにアタッチして、自動的に非同期モードになります (`$asynch_interface` で許可されている場合)。省略時の動作では、フォークした後、直ちに子プロセスを停止します。変数 `$stop_on_fork` を 0 に設定すると、この省略時の設定を変更することができます。この場合、dbx は子プロセスにアタッチしますが、それを停止されることはありません。

dbx デバグは、さまざまなシステム・コールおよびライブラリ・コールによって行われたフォーク呼び出しの多くにフィルタをかけることにより、フォークの処理にインテリジェンス度を適用しようとします。これらのフォークでプロセスを停止させる場合にも、定義済み変数 `$stop_all_forks` を 1 に設定します。この変数の省略時の値は 0 です。ライブラリ・ルーチンをデバグしている際には、すべてのフォークで停止させることは特に有効です。

デバグの `plist` および `switch` コマンドを使用すると、プロセス間のモニタおよび切り替えを行うことができます。

5.14 サンプル・プログラム

例 5-1 のサンプル C プログラム (`sam.c`) は、この章のいくつかのコマンド使用例で引用しているプログラムです。

例 5-1: dbx の例で使われているサンプル・プログラム

```
#include <stdio.h>
struct line {
    char string[256];
    int length;
    int linenumber;
};

typedef struct line LINETYPE;

void prnt();

main(argc,argv)
    int argc;
    char **argv;
{
    LINETYPE line1;
    FILE *fd;
    extern FILE *fopen();
    extern char *fgets();
    extern char *getenv();
    char *fname;
    int i;
    static curlinenumber=0;

    if (argc < 2) {
        if((fname = getenv("TEXT")) == NULL || *fname == ' ') {
            fprintf(stderr, "Usage: sam filename\n");
            exit(1);
        }
    } else
        fname = argv[1];

    fd = fopen(fname,"r");
    if (fd == NULL) {
        fprintf(stderr, "cannot open %s\n",fname);
        exit(1);
    }

    while(fgets(line1.string, sizeof(line1.string), fd) != NULL){
        i=strlen(line1.string);
        if (i==1 && line1.string[0] == '\n')
            continue;
        line1.length = i;
        line1.linenumber = curlinenumber++;
        prnt(&line1);
    }
}
```

例 5-1: dbx の例で使用されているサンプル・プログラム (続き)

```
void prnt(pline)
LINETYPE *pline;
{
    fprintf(stdout, "%3d. (%3d) %s",
            pline->linenumber, pline->length, pline->string);
    fflush(stdout);
}
```

6

lint による C プログラムの検査

lint プログラムを使用して、C プログラムにコーディング上の問題が含まれているかどうかを確認することができます。lint プログラムは C コンパイラよりも厳密にプログラムをチェックして、問題となりそうな点を指摘するメッセージを表示します。メッセージのいくつかはソース・コードの修正が必要ですが、情報メッセージで修正を必要としないものもあります。

この章では、次の項目について説明します。

- lint コマンドの構文 (6.1 節)
- プログラム・フロー・チェック (6.2 節)
- データ型チェック (6.3 節)
- 変数と関数のチェック (6.4 節)
- 初期化前の変数使用のチェック (6.5 節)
- 移行チェック (6.6 節)
- 移植性チェック (6.7 節)
- コーディング・エラーとコーディング・スタイルの相違のチェック (6.8 節)
- 大規模なプログラムのためのテーブル・サイズの増加 (6.9 節)
- lint ライブラリの作成 (6.10 節)
- lint エラー・メッセージ (6.11 節)
- lint メッセージ抑制のための警告クラス・オプションの使用 (6.12 節)
- 構文エラーのコンパイル時検出のための関数プロトタイプの生成 (6.13 節)

lint のオプションの詳細については lint(1) を参照してください。

6.1 lint コマンドの構文

lint コマンドの構文は次のとおりです。

lint [*options*] [*file* ...]

options

lint の検査を制御するオプション。

lint のオプションとして、cc ドライバ・オプション `-std`、`-std0`、`-std1` を使用することができます。これらのオプションは、使用する lint ライブラリの選択に影響を与えると同時に、ソースの解析に影響を与えます。`-std` あるいは `-std1` オプションを指定すると、lint で ANSI 解析規則が有効になります。

`-MA lint` オプションを使用すると、C プリプロセッシングで `-std1` が使用され、`-D` プリプロセッサ・オプションで `_ANSI_C_SOURCES` が定義されます。次に示すのは、各オプションに対する lint の動作です。

lint オプション	プリプロセッサ・スイッチ	lint 解析	lint ライブラリ
<code>-MA</code>	<code>-std1</code> および <code>-D_ANSI_C_SOURCE</code>	ANSI	<code>llib-lansi.ln</code>
<code>-std</code>	<code>-std</code>	ANSI	<code>llib-lcstd.ln</code>
<code>-std1</code>	<code>-std1</code>	ANSI	<code>llib-lcstd.ln</code>
<code>-std0</code>	<code>-std0</code>	EXTD ^a	<code>llib-lc.ln</code>

^aEXTD は、K&R C として知られている拡張 C 言語です。

file

lint が検査する C 言語のソース・ファイルの名前。ファイル名には、次のいずれかの接尾語が必要です。

接尾語	説明
<code>.c</code>	C ソース・ファイル
<code>.i</code>	C プリプロセッサ (cpp) によって作成されるファイル
<code>.ln</code>	lint ライブラリ・ファイル

lint ライブラリ・ファイルは、以前に lint プログラムを `-c` または `-o` オプションを指定して呼び出した結果です。これは、cc コマンドに

入力として .c ファイルを指定した場合に , .o ファイルが作成されるのと似ています。lint プログラムに入力として lint ライブラリが指定できる機能により , 大規模なアプリケーションにおけるモジュール間のインタフェース・チェックが容易になります。lint ライブラリの構造を makefile に指定する規則を追加すると , そのようなアプリケーションをもっと効率よく作成できます。lint ライブラリの作成方法については , 6.10 節を参照してください。

-lx オプションを使用して , システムの省略時のライブラリ探索ディレクトリのいずれかにある lint ライブラリを入力として指定することもできます。ライブラリ名は次の形式でなければなりません。

```
llib-llibname.ln
```

省略時の設定により , lint プログラムは , 拡張 C (K&R C) lint ライブラリ (llib-lc.ln) を , コマンド行に指定されたファイルのリストに追加します。-std または -std1 オプションが使用された場合は , 標準 C lint ライブラリ (llib-lcstd.ln) が代わりに追加されます。

システムには次の追加ライブラリが含まれています。

ライブラリ	説明	指定方法
crses	curses ライブラリ呼び出し構文を検査する。	-lcrses
m	算術ライブラリ呼び出し構文を検査する。	-lm
port	他のシステムとの移植性を検査する。	-p (-lport ではない)
ansi	ANSI C 規格の規則に準拠させる。	-MA (-lansi ではない)

コマンド行にオプションを指定しなければ , lint プログラムは , 指定の C ソース・ファイルを検査して , 次のいずれかのコーディング上の問題が見つかった場合にはメッセージを書き込みます。

- 正常に開始および終了しないループがある。
- 正しく使用されていないデータ型がある。
- 正しく使用されていない関数がある。
- 正しく使用されていない変数がある。

- プログラムを別のシステムに移植する際に、問題が発生する可能性があるコーディング技法が使用されている。
- 問題が発生する可能性がある、非標準的なコーディング技法やスタイルの違いがある。

また、`lint` プログラムは、ソース・プログラムの文中に構文エラーがないかどうかを検査します。構文検査は、`lint` コマンドに指定したオプションに関係なく常に行われます。

`lint` がエラーを報告しなければ、プログラムの構文に誤りがなく、正常にコンパイルされます。ただし、検査をパスしても、プログラムが正確に動作する、またはプログラムの論理設計が正確であるということにはなりません。

ユーザ固有の `lint` ライブラリの作成方法については、6.10 節を参照してください。

6.2 プログラム・フロー検査

`lint` プログラムは、デッド・コード、すなわち到達不可能なために実行されないプログラム部分がないかどうかを検査します。プログラムの流れを変更する、`goto`、`break`、`continue`、および `return` などの文の直後にあって、ラベルが付いていない文についてのメッセージを出力します。

`lint` プログラムは、先頭から開始できないループについても検出して、メッセージを書き込みます。このような型のループを含んだプログラムの中には、正しい結果を出すものもありますが、問題を引き起こす可能性があります。

`lint` プログラムは、呼び出されても呼び出し元のプログラムに戻らない関数を認識しません。たとえば、`exit` を呼び出すと到達不可能なコードを生成しますが、`lint` ではそのコードを検出できません。

`yacc` と `lex` によって生成されたプログラムには、到達不可能な `break` 文があります。`lint` プログラムは通常、これらの各 `break` 文に対するエラー・メッセージを出力します。これらの `break` 文に関連する余計なコードを取り除くには、プログラムをコンパイルするときに、`cc` コマンドで `-O` オプションを使用します。`yacc` と `lex` の出力コードを検査するときには、`lint` プログラムに `-b` オプションを使用すると、これらのメッセージは出力されません。`yacc` および `lex` についての情報は、『プログラミング・サポートツール・ガイド』を参照してください。

6.3 データ型検査

lint プログラムは、C 言語のデータ型の検査規則をコンパイラが行うよりも厳密に適用します。コンパイラが行う検査のほかに、lint は、次のような点について、データ型エラーの可能性がないかどうかを検査します。

- 二項演算子と暗黙の代入
- 構造体と共用体
- 関数の定義および使用方法
- 列挙値
- 型検査制御
- 型キャスト

これらの問題の可能性についての詳細は、以降の各項で説明します。

6.3.1 二項演算子および暗黙の代入

C 言語では、文中に次のデータ型を混在させることができ、コンパイラは、その混在についてのエラーを示しません。

```
char
short
int
long
unsigned
float
double
```

C 言語では、上記のグループ内のデータ型を互いに自動的に変換するため、より柔軟にプログラミングできます。ただし、この柔軟性は、C 言語自体によって保証されるものではないため、プログラマ自身が、データ型を混在させても確実に所定の結果が得られるようにする必要があります。

次の方法で使用する場合に、これらのデータ型を混在させることができます。例では、alpha が char 型で、num が int 型です。

- 代入演算子の両辺のオペランドとして次のように使用する。

```
alpha = num; /* alpha を int に変換する。*/
```

- 条件式のエンドとして次のように使用する。

```
value=(alpha < num) ? alpha : num; /* alpha を int に変換する。*/
```

- 関係演算子の両辺のオペランドとして次のように使用する。

```
if( alpha != num ) /* alpha を int に変換する。*/
```

- return 文の引数の型は、次の例のように関数が返す値の型に変換される。

```
funct(x) /* integer を返す。*/
{
    return( alpha );
}
```

任意のデータ型の配列とその同じ型を指すポインタが混在する場合以外は、ポインタのデータ型は正確に一致しなければなりません。

6.3.2 構造体と共用体

lint プログラムは、次のような条件の場合の構造体演算を検査します。

- 構造体ポインタ演算子 (->) の左辺のオペランドは、構造体へのポインタでなければならない。
- 構造体メンバ演算子 (.) の左辺のオペランドは、構造体でなければならない。
- これらの演算子の右辺のオペランドは、同じ構造体のメンバでなければならない。

lint プログラムは、共用体の参照についても同様の検査を行います。

6.3.3 関数定義と使用方法

lint プログラムは、関数の引数とリターン値の照合に厳密な規則を適用します。引数とリターン値は、型が一致している必要があります。ただし、次のような例外があります。

- float 型の引数と double 型の引数を照合できる。
- 次の型の引数を互いに照合できる。

```
char
short
int
unsigned
```

- ポインタを、それに対応する配列と照合できる。

6.3.4 列挙値

lint プログラムは、列挙データ型の変数が次の条件を満たしているかどうかを検査します。

- 列挙値変数または列挙型のメンバが、他の列挙値変数または他の型と混在していない。
- 列挙データ型の変数は、次の項目でのみ使用される。

代入 (=)
初期化
等価 (==)
不等価 (!=)
関数引数
リターン値

6.3.5 型キャスト

C 言語のプログラムでは、型キャストによって、ある型のデータを別の型のデータのように扱うことができます。lint プログラムは、型キャストがないかどうかを検査し、もしあればメッセージを出力します。

lint コマンド行に `-wp` および `-h` オプションを指定すると、キャストに関する警告メッセージの出力を制御します。どちらのオプションも使用しなければ、lint は移植性について問題が生じる可能性があるキャストに関して、警告メッセージを生成します。

移行検査モードでは、`-Qc` でキャストの警告メッセージの出力が抑制されます(6.6 節を参照)。

6.4 変数および関数の検査

lint プログラムは、プログラム内で宣言されていても使用されていない変数がないかどうかを検査します。また、変数および関数の使用方法について次のエラーがないかどうかを検査します。

- 矛盾する値を返す関数
- 定義されているが使用されていない関数
- 関数呼び出しの引数で、使用されていない引数
- 値を返す関数または値を返さない関数

- 使用されない値を返す関数
- 関数が値を返さないのに、その関数の値を使用するプログラム

これらの問題の可能性について、以降の各項で説明します。

6.4.1 矛盾する値を返す関数

関数が、ある条件がそろっているときにしか値を返さない場合、プログラムの結果は予測することができません。lint プログラムは、関数がこのような動作をしないかどうかを検査します。たとえば、次の2つの文が1つの関数定義にある場合には、その関数を呼び出すプログラムは、リターン値を受け取ったり、受け取らなかったりします。

```
return(expr);
:

```

```
return;
```

これらの文によって、lint プログラムは、次のようなメッセージを出力し、問題の可能性を指摘します。

```
function name has return(e); and return
```

lint プログラムはまた、関数コードの終端に到達したために発生するリターン (暗黙のリターン) がないかどうかについて、関数を調べます。たとえば、次は関数の一部ですが、a が偽と判断された場合は、checkout は fix_it を呼び出し、リターン値なしで戻ります。

```
checkout (a)
{
    if (a) return (3);
    fix_it ();
}
```

これらの文に対して、lint プログラムは、次のメッセージを出力します。

```
function checkout has return(e); and return
```

fix_it が、exit と同様に戻らない場合、lint は、エラーがなくてもメッセージを出力します。

6.4.2 使用されていない関数値

lint プログラムは、関数が値を返しても、呼び出しているプログラムがその値を使用していない場合がないかどうかを検査します。値が使用されていない場合には、関数定義が無効である可能性があるため、調べて、変更するかまたは削除するかを決定します。値が使用されることがある場合は、関数は、呼び出し側プログラムが検査をしていないという内容のエラー・コードを戻します。

6.4.3 関数についての検査の禁止

lint が、関数についての問題を検査しないようにするには、lint コマンドに次の表にあるオプションを 1 つまたは複数指定します。

- | | |
|----|---|
| -x | extern 文で宣言されているが、使用されていない変数を検査しない。 |
| -v | レジスタ引数としても宣言されているものを除き、使用されていない関数の引数を検査しない。 |
| -u | 使用されているのに定義されていない、または定義されているのに使用されていない関数および外部変数を検査しない。大きなプログラムのファイルのサブセットで lint を実行する場合にこのオプションを使用すると、無駄なメッセージが削除される。一緒に作動する一部の (全部ではない) ファイルで lint を使用すると、それらのファイルで定義されている関数および変数は、ほとんど使用されない。また、その他のファイルで定義されている多くの関数および変数が使用される。 |

プログラムに指示文を記述しても、チェックを制御することができます。

- 使用されていない関数の引数について lint が警告メッセージを出力しないようにするには、プログラム中の当該関数定義の前に次の指示文を追加します。

```
/*ARGSUSED*/
```

- 関数の呼び出し時に、複数の引数について lint がメッセージを出力しないようにするには、関数定義の前に次の指示文を追加します。

```
/*VARARGS n*/
```

初めのいくつかの引数のみを検査し、その後の引数を検査しないようにするには、次のように VARARGS 指示文の後に、検査すべき引数の数を 1 桁の数字 (n) で付加します。

```
/*VARARGS2*/
```

lint は、この指示文を読み取った場合、先頭の引数を 2 つだけ検査します。

- ファイル全体で使用されていない関数や関数の引数についてメッセージを抑制するには、次の指示文をファイルの先頭に記述します。

```
/*LINTLIBRARY*/
```

これは、`-v` および `-x` オプションを使用する場合と同等です。

- 関数プロトタイプ宣言を関数定義のように記述することにより、標準プロトタイプ・チェック・ライブラリをヘッダ・ファイルから作成できるようにするには、次の指示文を使用します。

```
/*LINTSTDLIB[ _filename ]
```

`/*LINTSTDLIB*/` 指示文は、`/*NOTUSED*/` および `/*LINTLIBRARY*/` 指示文の関数を暗黙に起動して、警告レベルを引き下げます。ファイルが参照されると (*filename*)、そのファイル内のプロトタイプのみが展開されます。`/*LINTSTDLIB_filename*/` 文は複数記述することができます。`/*LINTSTDLIB*/` 指示文の使用方法については、6.10.1 項を参照してください。

- ファイル内で検出され、使用されているが定義されていない外部シンボルおよび関数についての警告を抑制するには、次の指示文を使用します。

```
/*NOTDEFINED*/
```

- 到達不能なコードについてのメッセージを抑制するには、次の指示文を使用します。

```
/*NOTREACHED*/
```

プログラム内の適切な位置 (通常は `return`、`break`、または `continue` 文の直後) に記述すると、`/*NOTREACHED*/` 指示文は、到達不能なコードについてのメッセージ出力を停止します。lint は、`exit` 関数およびリターンしないかも知れない他の関数を認識しません。

- ファイル内で検出されるが、使用されていない外部シンボル、関数、および関数パラメータについての警告メッセージを抑制するには、次の指示文を使用します。

```
/*NOTUSED*/
```

`/*NOTUSED*/` 指示文は `/*LINTLIBRARY*/` 指示文と同等ですが、`/*NOTUSED*/` は外部シンボルに対しても適用されます。

6.5 初期化前の変数使用のチェック

lint プログラムは、ローカル変数 (auto と register の記憶クラス) が、値を割り当てられていないのに使用されていないかどうかを検査します。auto (自動) 記憶クラスまたは register 記憶クラスで変数を使用するには、変数のアドレスも取得することが必要です。これは、プログラムが変数のアドレスを認識すれば、いつでもそのアドレスによって変数を使用できるためです。したがって、プログラムが、変数に値を割り当てていないにもかかわらず変数のアドレスを見つけようとした場合、lint はエラーを報告します。lint はファイル内での変数の物理的な順番と使用されているかどうかを検査するだけであるため、適切に初期化された変数についてのメッセージを実行順に出力することがあります。

lint プログラムは、次の項目を認識してメッセージを出力します。

- 初期化された自動変数
- 最初に変数を設定する式に使用される変数
- 設定されていても使用されていない変数

注意

Tru64 UNIX オペレーティング・システムは、static 変数と extern 変数をゼロに初期化します。したがって、lint はプログラムの開始時にこれらの変数が 0 に設定されていると見なし、その変数にすでに値が割り当てられているかどうかについては、変数の使用時には確認しません。このような初期化を行っていないシステムのプログラムを開発する場合は、プログラムが static 変数と extern 変数を初期値に設定していることを確認してください。

6.6 移行検査

lint を使用して、32 ビット・オペレーティング・システムから Tru64 UNIX オペレーティング・システムに移行する場合に、問題が発生する可能性のあるすべての一般的なプログラミング技法を検査します。-Q オプション

は、64 ビット・システムに移行する ULTRIX および DEC OSF/1 バージョン 1.0 プログラムのチェックをサポートします。

-Q オプションを指定すると、その他のプログラム上の問題のチェックができなくなります。このため、移行検査の目的だけで使用してください。サブオプションは、特定のカテゴリの検査を抑止するために使用できます。たとえば、-Qa と入力すると、ポインタの位置合わせに関する問題を検査しません。-Q オプションには複数のサブオプションを指定することができます。たとえば -QacP は、ポインタ位置合わせの問題、問題のある型キャスト、および関数プロトタイプの検査をそれぞれ抑止します。移行検査についての詳細は、lint(1) を参照してください。

6.7 移植性検査

lint を使用すると、異なる C 言語コンパイラや他のシステムを使用している C プログラムのコンパイルおよび実行が確実にできるようになります。

以降の各項で、他のシステム上でプログラムをコンパイルする前に確認すべき点を示します。ただし、これらの点だけを確認すれば、どのシステムでもプログラムが実行可能であるというわけではありません。

注意

l1ib-port.ln ライブラリは、-lport オプションではなく、-p オプションを使用すると、取り込むことができます。

6.7.1 文字

システムの中には、C 言語プログラムで使用する文字を、-128 以上 127 以下の符号付きの数として定義しているものもあります。その他のシステムでは文字を正の値で定義しています。lint プログラムは、他のシステムに移植できない可能性のある文字の比較または文字の代入がないかどうかを検査します。たとえば、次のコードの一部は、あるシステムでは動作しますが、文字が常に正の値を取るシステムでは動作しません。

```
char c;

:

if( ( c = getchar() ) < 0 )...
```

この文により、lint プログラムは、次のようなメッセージを出力します。

```
nonportable character comparison
```

文字を正の値で定義しているシステムでプログラムを動作させるには、`getchar` が整数値を返すため、`c` を整数として宣言します。

6.7.2 ビット・フィールド

プログラムを別のシステムに移植する際、ビット・フィールドに問題がある可能性があります。新しいシステムではビット・フィールドも符号付き数である可能性もあります。したがって、ビット・フィールドに定数値を代入する場合は、フィールドが小さすぎて値を保持できない可能性があります。すべてのシステムで定数値の代入ができるようにするには、定数値を設定する前に、ビット・フィールドを `unsigned` 型として宣言します。

6.7.3 外部名サイズ

あるタイプのシステムから別のタイプのシステムに変更する場合は、プロセスのロード時に得られる外部名についての情報に、次のような相違があることに注意してください。

- 外部名に許される文字数が異なる。
- コンパイラ・コマンドが呼び出すプログラム、およびプログラムが呼び出す関数が、識別子の有効文字数をさらに制限する場合がある。
また、コンパイラはすべての名前の前に下線を追加して、大文字と小文字を区別します。
- 大文字と小文字を区別しないシステムと、区別され混在が許されないシステムがある。

あるシステムから別のシステムに移植する場合は、プログラムのロード時の問題を避けるため、常に次の手順を行ってください。

1. 各システムの条件を調べる。
2. `-p` オプションを付けて `lint` を実行する。

`-p` オプションを付けると、`lint` は、すべての外部シンボルを小文字に変更し、入力ファイルの検査時には 6 文字以内に制限します。出力されたメッセージには、変更が必要な用語が示されます。

6.7.4 複雑な式の使用と副作用

複雑な式を使用する場合には、次の点に注意してください。

- 複雑な式が評価される順序は、各 C コンパイラによって異なる。
- 他の関数の引数となっている関数の呼び出しは、通常の引数としては扱われない。
- 代入、インクリメント、およびデクリメントなどの演算子を異なるシステムで使用すると、問題が起こる可能性がある。

次に、これらの相違が原因で起こる 3 種類の問題の例を示します。

- 前述のいずれかの演算子の副作用によって任意の変数を変更され、その変数が同じ式の別の場所で使用されている場合、結果は未定義になる。
- 次の `printf` 文では、変数 `years` の評価は混乱する。

これは、`years` を関数呼び出しの前に増分するマシンもあれば、関数呼び出しの後で増分するマシンもあるためです。

```
printf( "%d %d\n", ++years, amort( interest, years ) );
```

- `lint` プログラムは、次の文に示されるような、評価の順序に影響されるおそれのある単純なスカラー変数を検査する。

```
a[i]=b[i++];
```

この文により、`lint` プログラムは次のメッセージを出力します。

```
warning: i evaluation order undefined
```

6.8 コーディング・エラーおよびコーディングのスタイルの相違のチェック

`lint` は、発生し得るコーディング・エラーを検出したり、`lint` が予期しているコーディング・スタイルとの相違を検出したりします。コーディング・スタイルは主として個人の趣味の問題ですが、相違する点は、正しくかつ必要なものであることを確認してください。以降の各項で、`lint` が検出するコーディング・エラーおよびコーディング・スタイルについての問題を示します。

6.8.1 `long` 型変数の `int` 型変数への代入

`long` 型の変数を `int` 型の変数に代入すると、プログラムは正しく動作しません。`long` 型変数は、`int` 型変数のスペースに適合するように切り捨てられるため、データが失われる可能性があります。

typedef を使用するプログラムを変換して異なるシステム上で実行すると、この種のエラーが頻繁に発生します。

long 変数の int 変数への代入を検出した場合、lint がメッセージを書き込まないようにするには、-a オプションを使用します。

6.8.2 演算子の優先度

lint プログラムは、演算子の優先度に関するエラーを検出します。複雑なシーケンス内では順番を表すカッコがなければ、このエラーの検出は困難です。たとえば、次の文では優先度がはっきりしていません。

```
if(x&077==0) . . . /* if(x & (077 == 0)) と評価される。*/
/* 本来はif((x & 077) == 0)である。*/
x<<2+40             /* x <<(2+40) と評価される。*/
/* 本来は (x<<2) + 40 である。*/
/* x を左へ 42 文字だけシフトする。*/
```

カッコを使用して演算がより明解になるようにしてください。カッコを使用しない場合には、lint はメッセージを出力します。

6.8.3 宣言の矛盾

lint プログラムは、内部ブロックで宣言される変数が、外部ブロックでの宣言と矛盾するような場合に、メッセージを出力します。この変数を実行することはできませんが、プログラムに問題が起こる可能性があります。

lint プログラムに -h オプションを付けると、lint は、宣言の矛盾がないかどうかを検査しません。

6.9 テーブル・サイズの増加

lint コマンドの -N オプションおよび関連のサブオプションは、さまざまな内部テーブルのサイズが、省略時の値ではプログラムを動作させるのに不十分な場合、実行時に増加させます。内部テーブルには次のようなものがあります。

- シンボル・テーブル
- ディメンション・テーブル
- ローカルの型テーブル
- 解析ツリー

これらのテーブルは、lint プログラムによって動的に割り当てられます。大きなソース・ファイルについて -N オプションを使用すると、性能を向上させることができます。

6.10 lint ライブラリの作成

システム・ライブラリ・ルーチン以外のライブラリ・ルーチンを作成するプログラム開発プロジェクトでは、さらに lint ライブラリを作成して、プログラムの構文を検査することができます。lint プログラムは、lint ライブラリを使用して、C 言語の標準関数だけでなく、新しい関数を検査することができます。新しい lint ライブラリを作成するには、次の手順を実行してください。

1. 新しい関数群を定義する 1 つの入力ファイルを作成する。
2. 1. の入力ファイルを処理して lint ライブラリ・ファイルを作成する。
3. 2. で作成した lint ライブラリを使用して、lint を実行する。

以降の各項で、これらの手順についての詳細を説明します。

6.10.1 入力ファイルの作成

次の例は、lint に検査させる関数を 3 つ定義した入力ファイルです。

```
/*LINTLIBRARY*/

#include <dms.h>

int  dmsadd( rmsdes, recbuf, reclen )
        int rmsdes;
        char *recbuf;
        unsigned reclen;
        { return 0; }
int  dmsclos( rmsdes)
        int rmsdes;
        { return 0; }
int  dmscrea( path, mode, recfm, reclen )
        char *path;
        int mode;
        int recfm;
        unsigned reclen;
        { return 0; }
```

入力ファイルとは、エディタで作成するテキスト・ファイルのことで、次のものから構成されます。

- /*LINTLIBRARY*/ の指示文

その後続く情報を lint 定義のライブラリとして作成するように cpp プログラムに指示します。

- 次の項目を定義する一連の関数定義
 - 関数の型 (上記の例では int)
 - 関数の名前
 - 関数が予期するパラメータ
 - パラメータの型
 - 関数が返す値

もう 1 つの方法として、関数プロトタイプから lint ライブラリ・ファイルを作成することができます。たとえば、dms.h ファイルに次のプロトタイプが含まれているとします。

```
int dmsadd(int,
           char*,
           unsigned);
int dmsclose(int);
int dmscrea(char*,
            int,
            int,
            unsigned);
```

この場合、入力ファイルには次の記述が含まれます。

```
/*LINTSTDLIB*/
#include <dms.h>
```

ヘッダ・ファイルに別のヘッダ・ファイルが含まれている場合は、LINTSTDLIB コマンドは特定のファイルに制限されます。

```
/*LINTSTDLIB_dms.h*/
```

この場合は、dms.h で宣言されたプロトタイプのみが展開されます。LINTSTDLIB コマンドは複数記述することができます。

いずれの場合にも、入力ファイルの名前には、接頭語 llib-1 がなければなりません。たとえば、この項で作成されるサンプルの入力ファイルの名前は、llib-ldms となります。ファイルに名前を付ける場合は、/usr/ccs/lib ディレクトリにある既存のファイルと同じ名前でないことを確認してください。

6.10.2 lint ライブラリ・ファイルの作成

次のコマンドは、前項で説明した入力ファイルから lint ライブラリ・ファイルを作成します。

```
% lint [options] -c llib_ldms.c
```

このコマンドは、入力ファイルとして llib-ldms.c を使用して、lint ライブラリ・ファイル llib-ldms.ln を作成するように lint に指示しています。その後 llib-ldms.ln をシステム lint ライブラリ (つまり、lint コマンドの -lx オプションで指定するライブラリ) として使用するには、それを /usr/ccs/lib に移動します。ANSI 前処理規則を使用してライブラリを構築するには、-std または -std1 オプションを使用します。

6.10.3 新しいライブラリによるプログラムの検査

新しいライブラリを使用してプログラムを検査するには、次の形式で lint コマンドを使用します。

```
lint -lpgm filename.c
```

pgm 変数にはライブラリの識別子を、filename.c 変数には、検査する C 言語のソース・コードの入っているファイル名を指定します。他のオプションが指定されていない場合、lint プログラムは、指定された特定の lint ライブラリだけでなく、標準の lint ライブラリとも照合して、C 言語ソース・コードを検査します。

6.11 lint エラー・メッセージ

lint のエラー・メッセージのほとんどはわかりやすいものですが、説明を加えなければ誤解が生じやすいメッセージもあります。通常、メッセージの意味を理解すると、エラーの修正は簡単です。次に、意味のあいまいな lint メッセージを示します。

```
constant argument to NOT
```

定数が NOT 演算子 (!) と一緒に使用されています。

これは一般的に行われるコーディングであり、必ずしも問題を示しているわけではありません。次のようなコーディングをすると、このメッセージが出力されることがあります。

```
% cat x.c
#include <stdio.h>
#define SUCCESS    0
```

```

main()
{
    int value = !SUCCESS;

    printf("value = %d\n", value);
    return 0;
}
% lint -u x.c
"x.c", line 7: warning: constant argument to NOT
% ./x
value = 1
%

```

lint はメッセージを出力しますが、プログラムは期待どおりに実行されます。

対応策: -wC オプションを使用して、lint の警告メッセージを出力しないようにします。

constant in conditional context

条件を指定すべきところに定数が使用されています。

この問題は、マクロの使用方法によって、ソース・コードでよく起こります。たとえば、次のような場合です。

```

typedef struct _dummy_q {
    int lock;
    struct _dummy_q *head, *tail;
} DUMMY_Q;

#define QWAIT 1
#define QNOWAIT 0
#define DEQUEUE(q, elt, wait) 1 \
    for (;;) {
        simple_lock(&(q)->lock);
        if (queue_empty(&(q)->head))
            if (wait) { 1 \
                assert(q);
                simple_unlock(&(q)->lock);
                continue;
            } else
                *(elt) = 0;
        else
            dequeue_head(&(q)->head);
            simple_unlock(&(q)->lock);
        break;
    }

```

```
int doit(DUMMY_Q *q, int *elt)
{
    DEQUEUE(q, elt, QNOWAIT);
}
```

- ❶ QWAIT または QNOWAIT オプションは 3 番目の引数 (wait) として渡され、その後 if 文で使用されます。コードは正しいのですが、lint は警告メッセージを出力します。これは、通常このように使用される定数は不必要であり、無駄な命令を生成するためです。

対応策: -wC オプションを使用して、lint の警告メッセージを出力しないようにします。

conversion from long may lose accuracy

符号付き long が小さい要素 (int など) にコピーされています。このメッセージは必ずしも誤解を招くものではありませんが、よく起こり、次の例に示すように、コーディングに問題がある場合もあれば、問題がない場合もあります。

```
long BuffLim = 512;           ❶

void foo (buffer, size)
char *buffer;
int size;
{
    register int count;
    register int limit = size < (int)BufLimit ? size : (int)BufLim; ❶
```

- ❶ 適切な (int) キャストが使用されていますが、lint プログラムは変換エラーを報告します。

対応策: lint がメッセージを報告するコード・セクションを調べるか、または -wC オプションを使用して、lint の警告メッセージを出力しないようにします。

declaration is missing declarator

プログラムの宣言節にセミコロン (;) だけの行があります。

このようなコードを書かなくても、マクロの後ろにセミコロンを記述した場合などに、このようなコードが生成されることがあります。条件化により、マクロが空として定義された場合、このメッセージが出力されることがあります。

対応策: 後ろのセミコロンを削除してください。

degenerate unsigned comparison

結果がゼロより小さくなると思われる場合に、符号なし比較が符号付きの値に対して実行されました。

次のようなプログラムでこのメッセージが出力されます。

```
% cat x.c
#include <stdio.h>
unsigned long offset = -1;

main()
{
    if (offset < 0) {
        puts ("code is Ok...");
        return 0;
    } else {
        puts ("unsigned comparison failed...");
        return 1;
    }
}
% cc -g -o x x.c
% lint x.c
"x.c" line 7: warning: degenerate unsigned comparison
% ./x
unsigned comparison failed...
%
```

- ① このような符号なし比較は、符号なし変数に負の値が入っている場合には失敗します。符号付き比較を意図していたかどうかによって、結果のコードは正しいこともあります。

対応策: この例は、次の 2 つの方法で修正することができます。

- if 比較の offset の前に (long) キャストを追加する。
- offset の宣言を unsigned long から long に変更する。

場合によっては、符号付きの値を符号なしにキャストする必要があります。

function prototype not in scope

このエラーは厳密には関数プロトタイプに関連していません。実際、このエラーは前もって宣言や定義を行っていない関数を呼び出した場合に起こります。

対応策: 関数プロトタイプ宣言を追加してください。

null effect

lint プログラムが何も行わないキャストまたは文を検出しました。

次のコードは、lint がこのメッセージを出力するさまざまなコーディング例を示しています。

```
scsi_slot = device->ctrl_hd->slot,unit_str; [1]

#define MCLUNREF(p) \
    (MCLMAPPED(p) && --mclrefcnt[mtocl(p)] == 0)

(void) MCLUNREF(m); [2]
```

[1] 理由: unit_str は何も行いません。

[2] 理由: MCLUNREF はマクロであるため (void) は不要です。

対応策: 不必要なキャストや文を削除したり、マクロを修正してください。

possible pointer alignment problem

位置合わせの問題が生じるような方法でポインタが使用されています。

次のコードは、lint によりこのメッセージが生成されるコード例を示しています。

```
read(p, args, retval)
    struct proc *p;
    void *args;
    long *retval;
{
    register struct args {
        long    fdes;
        char    *cbuf;
        unsigned long    count;
    } *uap = (struct args *) args; [1]
    struct uio auio;
    struct iovec aiov;
```

- ① *uap = (struct args *) args という文が、このエラーを引き起こします。この構文は有効であり、カーネル・ソース内で使用されるため、このメッセージはフィルタされます。

precision lost in field assignment

ビット・フィールドがある値を保持するには小さすぎる場合に、そのフィールドにその定数を代入しようとした。

次のコードはこの問題を示しています。

```
% cat x.c
struct bitfield {
    unsigned int block_len : 4;
} bt;

void
test()
{
    bt.block_len = 0xff;
}

% lint -u x.c
"x.c", line 8: warning: precision lost in field assignment
% cc -c -o x x.c
%
```

このコードはエラーを生じることなくコンパイルされます。ただし、ビット・フィールドがその定数を保持するには小さすぎるため、意図した結果にならず、実行時エラーが生じる場合もあります。

対応策: ビット・フィールドのサイズを変更するか、または異なる定数値を代入してください。

unsigned comparison with 0

結果がゼロに等しいか大きいと予想される場合に、ゼロに対して符号なしの比較が実行されました。

次のプログラムは、このような状況を示しています。

```
% cat z.c
#include <stdio.h>
unsigned offset = -1;

main()
{
    if (offset > 0) {
```

①

```

        puts("unsigned comparison with 0 Failed");
        return 1;
    } else {
        puts("unsigned comparison with 0 is Ok");
        return 0;
    }
}
% cc -o z z.c
% lint z.c
"z.c", line 7: warning: unsigned comparison with 0?
% ./z
unsigned comparison with 0 Failed
%

```

- ❶ このような符号なし比較は、符号なし変数に負の値が入っている場合に失敗します。符号付き比較を意図していたかどうかによって、結果のコードは正しくない場合があります。

対応策: この例は、次の 2 つの方法で修正できます。

- `if` 比較の `offset` の前に、`(int)` キャストを追加する。
- `offset` の宣言を `unsigned` から `int` へ変更する。

6.12 警告クラス・オプションを使用した lint メッセージの抑制

`lint` プログラムにいくつかの `lint` 警告クラスを追加することによって、条件処理で使用される定数、移植性、およびプロトタイプの検査に関連したメッセージの出力を抑制できるようになりました。`lint` コマンドで警告クラス・オプションを使用すると、どの警告クラスのメッセージでも抑制することができます。

警告クラス・オプションのフォーマットは次のとおりです。

-wclass [*class...*]

省略時の設定ではすべての警告クラスがアクティブですが、*class* 引数として適切なオプションを指定すると、個々のクラスを非アクティブにすることができます。表 6-1 で個々のオプションについて説明します。

注意

lint メッセージには、複数の警告クラスに依存するものがいくつかあります。したがって、これらのメッセージを抑制するには、複数の警告クラスを指定する必要があります。表 6-1 の脚注は、複数の警告クラスを指定して、どのメッセージが抑制できるかだけを示しています。

たとえば、条件式の定数に関連する lint メッセージは、(6.11 節で説明しているように) 必ずしもコーディング上の問題を示していないので、`-wC` オプションを使用してメッセージを抑制するとします。`-wC` オプションは、次のメッセージの出力を抑制します。

- `constant argument to NOT`
- `constant in conditional context`

移植性の検査に関するメッセージの多くは、非 ANSI コンパイラおよび Tru64 UNIX 用の C コンパイラに存在しない制限に関連するため、`-wp` オプションを使用して、メッセージを抑制することができます。`-wp` オプションは、次のメッセージの出力を抑制します。

- `ambiguous assignment for non-ansi compilers`
- `illegal cast in a constant expression`
- `long in case or switch statement may be truncated in non-ansi compilers`
- `nonportable character comparison`
- `possible pointer alignment problem, op %s`
- `precision lost in assignment to (sign-extended?) field`
- `precision lost in field assignment`
- `too many characters in character constant`

関数プロトタイプの使用は (6.13 節で説明しているように) 好ましいコーディング方法ですが、多くのプログラムでは使用されていません。`-wP` オプションを使用すると、プロトタイプの検査を禁止することができます。`-wP` オプションは、次のメッセージの出力を抑制します。

- `function prototype not in scope`

- mismatched type in function argument
- mix of old and new style function declaration
- old style argument declaration
- use of old-style function definition in presence of prototype

表 6-1: lint 警告クラス

警告クラス	クラスの説明
a	<p>非 ANSI 機能。次のメッセージの出力を抑制する。</p> <ul style="list-style-type: none"> • Partially elided initialization^a • Static function %s not defined or used^a
c	<p>符号なし値の比較。次のメッセージの出力を抑制する。</p> <ul style="list-style-type: none"> • Comparison of unsigned with negative constant • Degenerate unsigned comparison • Possible unsigned comparison with 0
d	<p>宣言の一貫性。次のメッセージの出力を抑制する。</p> <ul style="list-style-type: none"> • External symbol type clash for %s • Illegal member use: perhaps %s.%s^b • Incomplete type for %s has already been completed • Redclaration of %s • Struct/union %s never defined^b • %s redefinition hides earlier one^{a b}

表 6-1: lint 警告クラス (続き)

警告クラス	クラスの説明
h	<p>ヒューリスティックな障害。次のメッセージの出力を抑制する。</p> <ul style="list-style-type: none">• Constant argument to NOT^c• Constant in conditional context^c• Enumeration type clash, op %s• Illegal member use: perhaps %s.%s^d• Null effect ^e• Possible pointer alignment problem, op %s^f• Precedence confusion possible: parenthesize!^g• Struct/union %s never defined^d• %s redefinition hides earlier one^d
k	<p>K & R 型のコードを期待する。次のメッセージの出力を抑制する。</p> <ul style="list-style-type: none">• Argument %s is unused in function %s^h• Function prototype not in scope^h• Partially elided initialization^h• Static function %s is not defined or used^h• %s may be used before set^{b d}• %s redefinition hides earlier one^{b d}• %s set but not used in function %s ^h
l	<p>非 long 型変数に long 型の値を代入した。次のメッセージの出力を抑制する。</p> <ul style="list-style-type: none">• Conversion from long may lose accuracy• Conversion to long may sign-extend incorrectly
n	<p>空作用コード。次のメッセージの出力を抑制する。</p> <ul style="list-style-type: none">• Null effect ^b

表 6-1: lint 警告クラス (続き)

警告クラス	クラスの説明
o	未知の評価順序。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">• Precedence confusion possible: parenthesize! ^b• %s evaluation order undefined
p	移植性に関連するさまざまな事柄。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">• Ambiguous assignment for non-ANSI compilers• Illegal cast in a constant expression• Long in case or switch statement may be truncated in non-ANSI compilers• Nonportable character comparison• Possible pointer alignment problem, op %s ^b• Precision lost in assignment to (possibly) sign-extended field• Precision lost in field assignment• Too many characters in character constant
r	Return 文の一貫性。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">• Function %s has return(e); and return;• Function %s must return a value• main() returns random value to invocation environment
s	記憶容量の検査。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">• Array not large enough to store terminating null• Constant value (0x%x) exceeds (0x%x)
u	変数および関数の適切な使用。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">• Argument %s unused in function %s^a• Static function %s not defined or used^a• %s set but not used in function %s^a• %s unused in function %s^h

表 6-1: lint 警告クラス (続き)

警告クラス	クラスの説明
A	すべての警告クラスをアクティブにする。lint スクリプトにおける省略時のオプション。別の A クラスを指定すると、全クラスの設定が切り替わる。
C	条件処理で使用されている定数。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">Constant argument to NOT^bConstant in conditional context^b
D	外部宣言が使用されていない。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">Static %s %s unused
O	使用されていない機能。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">Storage class not the first type specifier
P	プロトタイプの検査。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">Function prototype not in scope^aMismatched type in function argumentMix of old- and new-style function declarationOld-style argument declaration^aUse of old-style function definition in presence of prototype
R	実行されないコードの検出。次のメッセージの出力を抑制する。 <ul style="list-style-type: none">Statement not reached

^ak 警告クラスを非アクティブにすることによってもこのメッセージを抑制できる。
^bこのメッセージを抑制するには、h 警告クラスも非アクティブにしなければならない。
^cこのメッセージを抑制するには、c 警告クラスも非アクティブにしなければならない。
^dこのメッセージを抑制するには、d 警告クラスも非アクティブにしなければならない。
^eこのメッセージを抑制するには、n 警告クラスも非アクティブにしなければならない。
^fこのメッセージを抑制するには、p 警告クラスも非アクティブにしなければならない。
^gこのメッセージを抑制するには、o 警告クラスも非アクティブにしなければならない。
^h他のオプションもこれらのメッセージを抑制する。

6.13 コンパイル時に検出される構文エラーのための関数プロトタイプの生成

lint プログラムが報告するさまざまなエラーの修正に加えて、外部関数および静的関数の両方に対して関数プロトタイプを追加することをお勧めします。この宣言は、コンパイラが検査する必要がある引数およびリターン値の情報を提供します。

cc コンパイラには、自動的にプロトタイプ宣言を生成するオプションがあります。コンパイル時に `-proto[is]` オプションを指定すると、関数プロトタイプを含む出力ファイル(入力ファイルと同じファイル名でファイル・タイプが `.H`)を作成することができます。`-i` オプションを指定するとプロトタイプに識別子を含み、`-s` オプションを指定すると静的関数に対してもプロトタイプを生成します。

`.H` ファイルから関数プロトタイプをコピーしてソース内の適切な場所に挿入し、ファイルを取り込みます。

Third Degree によるプログラムのデバッグ

Third Degree ツールは、C および C++ プログラムにおけるヒープ・メモリのリーク、無効アドレスの参照、および初期化されていないメモリの読み込みを検査します。始めに、`-g` または `-gn` オプションを使用してプログラムをコンパイルしておく必要があります (n は 0 より大きな値)。また、Third Degree は、ヒープ・オブジェクトのリストを作成し、浪費されているメモリを検出するので、プログラムの割り当てパターンの決定にも役立ちます。これは、メモリ管理サービスを自動的に監視し、実行時に命令をロード/格納するための追加のコードで、実行可能オブジェクトを計測することによって行われます。要求されたレポートは、オプションで表示できる 1 つまたは複数のログ・ファイルに書き込まれるか、`xemacs(1)` エディタを使用してソース・コードに関連付けられます。

Third Degree は、省略時の設定ではメモリ・リークのみを検査するため、計測および実行時分析を短時間で行うことができます。費用がかかり、処理のじゃまになるその他の検査は、コマンド行のオプションで選択します。詳細については、`third(1)` を参照してください。

Third Degree は、次のタイプのアプリケーションに使用できます。

- `malloc`, `calloc`, `realloc`, `valloc`, `alloca`, `sbrk` の各関数、および C++ の `new` 関数を使用してメモリを割り当てるアプリケーション。

Third Degree を使用して、`mmap` 関数など、他のメモリ割り当て機能を使用するプログラムも計測できますが、この方法で取得したメモリへのアクセスは検査しません。アプリケーションが `mmap` を使用する場合は、`third(1)` の `-mapbase` オプションについての説明を参照してください。

Third Degree は `brk` 関数への呼び出しを検出して、禁止します。さらに、プログラムが、`sbrk` 関数を使用して取得した大きなブロックを分割することによってメモリを割り当てる場合、Third Degree は、エラーが発生したメモリ・ブロックを厳密に特定できないことがあります。

- `fork(2)` を呼び出すアプリケーション。

`third(1)` コマンドで `-fork` オプションを指定する必要があります。

- POSIX スレッド (pthread(3)) の Tru64 UNIX による実装を使用するアプリケーション。

third(1) コマンドで `-pthread` オプションを指定する必要があります。pthread プログラムでは、Third Degree は、無効なアドレスまたは初期化されていない変数にアクセスする際に、システム・ライブラリ・ルーチン (libc, libpthread など) を検査しません。したがって、strcpy およびその他のルーチンも検査されません。

- 31 ビットのヒープ・アドレスを使用するアプリケーション。

7.1 アプリケーションにおける Third Degree の実行

Third Degree を起動するには、次のように third(1) コマンドを使用します。

```
third [ option... ] app [ argument...]
```

このコマンド形式では、*option* には、省略時のスレッドを使用しないリーク検査以外に 1 つまたは複数のオプションを選択します。app は、アプリケーションの名前です。argument は、計測機構付きプログラムをすぐに実行したい場合にアプリケーションに渡される 1 つまたは複数のオプションの引数を表します (app が引数を必要としない場合は、`-run` オプションを使用します)。

app.third (third(1) を参照) と名付けられた計測機構付きプログラムは、次の点で元のアプリケーションと異なる動作をします。

- 計測コードが追加で挿入されたため、コードが大きくなり、実行速度は遅くなります。オーバーヘッドの大きさは指定されたオプションの数と性質によって異なります。
- 初期化されていないデータの誤使用を検出するために、Third Degree は、アプリケーションで初期化されないデータをすべて特殊なパターン (0xff8a5a5, または `-uninit` オプションで指定されたパターン) で初期化します。これにより、計測機構付きプログラムの動作が変わったり、誤動作したり、またはクラッシュしたりすることがあります (特に、この特別なパターンがポインタとして使用される場合)。これらの動作はすべて、プログラムにバグがあることを示しています。これは、計測機構付きプログラムで回帰テストを行う際に活用できます。また、third(1) コマンドの `-g` オプションを使用してデバグを実行すると、問題を調査することができます。Third Degree は、`-uninit` オプションが指定

された場合 (-uninit heap+stack など) にのみ、このようにメモリに特殊なパターンで埋め込みを行います。それ以外の場合、ほとんどの計測機構付きプログラムは、元のプログラムと同様に動作します。

- Third Degree は、境界チェックができるようにするために、割り当てられた各ヒープ・メモリ・オブジェクトに埋め込みを行っているので、それらのオブジェクトが大きくなっています。埋め込みの量は、-pad オプションを指定することによって調整できます。
- free または delete を使用してメモリを解放する場合、不正なアクセスを検出するため、空きプールによって阻止されます。これは、です。-free オプションを使用して、待機時間を調節します。

Third Degree のエラー・メッセージの記述形式は、C コンパイラが使用する記述形式と類似しています。省略時の設定では、Third Degree は、app.3log という名前のログ・ファイルにエラー・メッセージを書き込みます。emacs を使用すると、各エラーを自動的に順番に指定することができます。emacs では、Esc/X compile を使用して、省略時の make コマンドを cat app.3log などのコマンドに置き換え、Ctrl/X、を使用して、コンパイル・エラーの場合のように、順番にエラーを表示していきます。

ログ・ファイルに使用される名前は、次のオプションのうちいずれかを指定することによって変更できます。

-pids

プロセス識別番号 (PID) をログ・ファイル名に含めます。

-dirname *directory-name*

Third Degree がそのログ・ファイルを作成するディレクトリ・パスを指定します。

-fork

フォークした各プロセスのログ・ファイル名に PID を含めます。

使用するオプションによって、ログ・ファイル名は次のようになります。

オプション	ファイル名	使用
なし、または -fork <i>parent</i>	app.3log	省略時の設定

オプション	ファイル名	使用
<code>-pids</code> または <code>-fork child</code>	<code>app.12345.3log</code>	PID を取り込む
<code>-dirname /tmp</code>	<code>/tmp/app.3log</code>	ディレクトリを設定
<code>-dirname /tmp -pids</code>	<code>/tmp/app.12345.3log</code>	ディレクトリと PID を設定

シグナル・ハンドラでのエラーは、追加の `.sig.3log` オプションにより通知されます。

7.1.1 シェアード・ライブラリでの Third Degree の使用

`strcpy` 関数に渡すバッファが小さすぎるなど、アプリケーション内のエラーは、ライブラリ・ルーチンで見つかることがよくあります。Third Degree はシェアード・ライブラリの計測をサポートしており、`-non_shared` または `-call_shared` オプションを指定してリンクされたプログラムを計測します。

次のオプションを指定すると、Third Degree が計測するシェアード・ライブラリを決定できます。

`-all`

呼び出し共用実行可能プログラムにリンクされたすべてのシェアード・ライブラリを計測します。

`-excobj objname`

指定したシェアード・ライブラリを計測から除外します。`-excobj` オプションを複数回使用すれば、複数のシェアード・ライブラリを指定できます。

`-incobj objname`

指定したシェアード・ライブラリを計測します。`-incobj` オプションを複数回使用すれば、`dlopen()` を使用してロードしたシェアード・ライブラリを含め、複数のシェアード・ライブラリを指定できます。

`-Ldirectory`

リンカやローダが認識する通常的位置にプログラムのシェアード・ライブラリがない場合、Third Degree にその検索位置を通知します。

Third Degree がアプリケーションの計測を終了すると、現在のディレクトリには、指定した各シェアド・ライブラリの計測機構付きバージョンが含まれています。また、必要に応じて `libc.so`、`libcxx.so`、および `libpthread.so` の最低限の計測機構付きバージョンが作成されます。計測機構付きアプリケーションは、ライブラリのこれらのバージョンを使用する必要があります。LD_LIBRARY_PATH 環境変数を定義して、計測機構付きシェアド・ライブラリがどこに常駐するかを、計測機構付きアプリケーションに通知してください。third(1) コマンドは、`-run` オプションを指定する場合、またはアプリケーションに引数を指定している場合にこれを自動的に実行します。また、計測機構付きプログラムを自動的に起動します。

省略時の設定では、Third Degree は、アプリケーションが使用するシェアド・ライブラリを完全には計測しません。ただし、使用するときには、最低限 `libc.so`、`libcxx.so`、および `libpthread.so` を計測する必要があります。これにより、計測の動作がはるかに高速になるとともに、計測機構付きアプリケーションも実行速度がより高速になります。Third Degree は、通常、計測機構付き部分のエラーを検出して報告しますが、計測機構のないライブラリのエラーは検出しません。部分的に計測したアプリケーションがクラッシュまたは誤動作し、Third Degree によって報告されたエラーをすべて修正した場合は、そのすべてのシェアド・ライブラリとともにアプリケーションを再計測し、新しい計測機構付きバージョンを実行するか、または Third Degree の `-g` オプションを使用してデバッガで問題を調査してください。

Third Degree は、プロシージャのスタック・トレースを含むエラー・レポートを生成するため、シェアド・ライブラリを計測する必要があります(ただし、省略時の設定では最小限のみ)。また、デバッグ可能なプロシージャ(たとえば、`-g` オプションを使用してコンパイルしたもの)は、エラーに最も近い最初のいくつかのスタック・フレーム内に表示される必要があります。これにより、高度に最適化され、システム・ライブラリのアセンブリ・コードによって生成される可能性がある見せかけのエラーのプリントが回避されます。この機能を無効にするには、`-hide` オプションを使用してください。

pthread プログラムでは、Third Degree は、ある種のシステム・シェアド・ライブラリ(`libc` を含む)のエラーを検査しません。これは、そうした場合にスレッド・セーフでなくなるためです。

7.2 デバッグ例

次のソース・コード `ex.c` で表される小さなアプリケーションをデバッグしなければならないとします。

```
1  #include <assert.h>
2
3  int GetValue() {
4      int q;
5      int *r=&q;
6      return q;          /* q is uninitialized */
7  }
8
9  long* GetArray(int n) {
10     long* t = (long*) malloc(n * sizeof(long));
11     t[0] = GetValue();
12     t[0] = t[1]+1;      /* t[1] is uninitialized */
13     t[1] = -1;
14     t[n] = n;           /* array bounds error*/
15     if (n<10) free(t); /* may be a leak */
16     return t;
17 }
18
19 main() {
20     long* t = GetArray(20);
21     t = GetArray(4);
22     free(t);           /* already freed */
23     exit(0);
24 }
```

以降の各項で、Third Degree を使用して、このサンプル・アプリケーションをデバッグする方法について説明します。

7.2.1 Third Degree のカスタマイズ

コマンド行オプションを使って、Third Degree のさまざまな機能のオン/オフを切り替えることができます。

オプションを何も指定しない場合、Third Degree はプログラムを次のように計測しますが、計測機構付きプログラムの実行や、生成される `.3log` ファイルの表示は行いません。

- プログラム終了時にリーク検出を行う。
- メモリ・エラー (無効なアドレスまたは初期化されていない値) を検査しない。
- ヒープの使用履歴を分析しない。

計測機構付きアプリケーションは、LD_LIBRARY_PATH 環境変数の設定後に `./app.third arg1 arg2` などのコマンドを使って実行できます。また、アプリケーションの引数を `third(1)` コマンド行に追加したり、`-run` または `-display` オプションを指定したりすることもできます。生成された `.3log` ファイルは、手動または `-display` オプションを指定することにより表示できます。

メモリ・エラー検査を追加するには、`-invalid` オプションと `-uninit` オプションのいずれかまたは両方を指定します。

`-invalid` オプションは、すべての `third(1)` オプションと同様に 3 文字 (`-inv`) に短縮できます。このオプションは、Third Degree に対し、重要なロードおよび格納命令のすべてが、アプリケーション・コードが使用するべき有効なメモリ・アドレスにアクセスしているかどうかを検査します。このオプションを使用すると、目立った性能のオーバーヘッドが生じますが、実行時環境にはほとんど影響を与えません。

`-uninit` オプションには、キーワード引数を “+” で区切ってリストします。これは、通常 `heap+stack` (または `h+s`) となり、すべての重要なロード命令で、ヒープ・メモリとスタック・メモリの両方を検査するよう指示します。検査には、`malloc` など (`calloc` ではなく) を使って割り当てられたすべてのスタック・フレームおよびヒープ・オブジェクトに、通常ではあり得ないパターン `0xffff8a5a5` を充填し、この値をメモリから読み取るすべてのロード命令を報告することが含まれます。この場合、選択されたメモリは、初期化されていないデータ領域を読み取るコードを強調表示する点で、`cc -trapuv` と同様に重大な問題があります。問題を引き起こすコードが計測として選択されている場合、Third Degree はそれぞれの場合を `.3log` ファイル内で一度だけ報告します。ただし、コードが計測されたかどうかに関係なく、コードは、元のプログラムがロードするはずの値の代わりに問題のあるパターンをロードおよび処理します。この場合、パターンが有効なポインタ、文字、浮動小数点数ではなく、負の整数であるために、プログラムの誤動作またはクラッシュが引き起こされる場合があります。このような動作は、プログラムにバグが含まれている印となります。

計測機構付きプログラム上で回帰テストを実行することにより、誤動作を識別できます。`third(1)` 内で実行する場合には、`-quiet` を指定して、`-display` を省略します。`.3log` ファイル内のエラー・メッセージを参照し、計測機構付きプログラムを、`dbx(1)` や、C++ 用の `ldebug(1)`、および `pthread` アプリケーションなどのデバッガ内で実行することにより、誤動作

またはクラッシュのデバッグを行うことができます。デバッガを使用するには、`-g` オプションを付けてコンパイルし、`third(1)` のコマンド行でも同様に `-g` を指定します。

`-uninit` オプションを指定すると、偽のエラー、特に変数、配列要素、32 ビット未満の構造体のメンバ (`short`、`char` ビットフィールドなど) がレポートされます。詳細は、7.6 節を参照してください。ただし、`-uninit heap+stack` オプションを使用すると、リーク・レポートの正確性が向上します。

ヒープの使用分析を追加する場合は、`-history` オプションを指定します。これにより、`-uninit heap` オプションが有効になります。

7.2.2 Makefile の変更

アプリケーションの `makefile` に、次のエントリを追加します。

```
ex.third: ex
        third ex
```

次のように `ex.third` を作成します。

```
> make ex.third
third ex
```

ここで、計測機構付きアプリケーション `ex.third` を実行して、ログ `ex.3log` をチェックします。あるいは、プログラム名の前に `-display` を追加すると、そのアプリケーションを実行して、`.3log` ファイルが直ちに表示されます。

7.2.3 Third Degree のログ・ファイルの検査

`ex.3log` ファイルには、以降の項で説明する部分がいくつか含まれています。このコマンド行は、次のとおりであると想定します。

```
> third -invalid -uninit h+s -history -display ex
```

7.2.3.1 実行時メモリ・アクセス・エラーのリスト

実行時に Third Degree が検出できるエラーのタイプには、初期化されていないメモリの読み取り、割り当てられていないメモリの読み取りまたは書き込み、誤ったメモリの解放、および例外を起こす可能性の高い一定の重大エラーなどの状態が含まれます。各エラーに対して、次の項目を持つエラー・エントリが生成されます。

- エラーのタイプと番号を記した見出し行

エラー見出し行には、各エラーの英字 3 文字の短縮形が表示されます (短縮形のリストについては、7.3 節を参照)。エラーを引き起こしたプロセスがルート・プロセスではない場合 (たとえば、アプリケーションが 1 つまたは複数の子プロセスをフォークするため) は、エラーを引き起こしたプロセスの PID も見出し行に表示されます。

- コンパイラのエラー・メッセージ行と似た書式のエラー・メッセージ

Third Degree は、ファイル名およびエラーが発生した位置に最も近い行番号をリストします。通常、これは、エラーが発生した正確な位置ですが、エラーがライブラリ・ルーチン内で発生した場合は、ライブラリ・コールが行われた場所を示すこともあります。

- 1 つ以上のスタック・トレース

エラー・エントリの最後の部分はスタック・トレースです。スタック・トレースにリストされる最初のプロシージャは、エラーが発生したプロシージャです。

次の例に、ログ・ファイルからのエントリを示します。

- 次のログ・エントリは、プロシージャ GetValue のローカル変数が、初期化される前に読み取られたことを示しています。行番号が表示されるのは、q に値が与えられなかったことを確認するためです。

```
----- rus -- 0 --
ex.c: 6: reading uninitialized local variable q of GetValue
  GetValue          ex, ex.c, line 6
  GetArray          ex, ex.c, line 11
  main              ex, ex.c, line 20
  __start           ex
```

- 次のログ・エントリは、12 行目でエラーが発生したことを示しています。

```
t[0] = t[1]+1
```

配列が初期化されなかったため、プログラムは、加算で t[1] という初期化されていない値を使用しています。配列 t を含むメモリ・ブロックは、それを割り当てた呼び出しスタックによって識別されます。-g オプションを指定してコードがコンパイルされている場合、スタック変数は名前で識別されます。

```
----- ruh -- 1 --
ex.c: 12: reading uninitialized heap at byte 8 of 160-byte block
  GetArray          ex, ex.c, line 12
  main              ex, ex.c, line 20
```

```

__start                                ex

This block at address 0x14000ca20 was allocated at:
malloc                                ex
GetArray                             ex, ex.c, line 10
main                                 ex, ex.c, line 20
__start                                ex

```

- 次のログ・エントリは、プログラムが、配列の終わりの位置を1つだけ過ぎたメモリ位置に書き込んで、重要なデータ、または Third Degree 内部のデータ構造体を重ね書きした可能性があることを示しています。後で報告されるエラーの中には、このエラーの結果である可能性があります。

```

----- wih -- 2 --
ex.c: 14: writing invalid heap 1 byte beyond 160-byte block
GetArray                             ex, ex.c, line 14
main                                 ex, ex.c, line 20
__start                                ex

This block at address 0x14000ca20 was allocated at:
malloc                                ex
GetArray                             ex, ex.c, line 10
main                                 ex, ex.c, line 20
__start                                ex

```

- 次のログ・エントリは、以前に解放されたメモリを解放しているときにエラーが発生したことを示しています。free 関数の呼び出しに伴うエラーでは、Third Degree は、通常、3つの呼び出しスタックを与えます。

- エラーが発生した呼び出しスタック
- オブジェクトが割り当てられた呼び出しスタック
- オブジェクトが解放された呼び出しスタック

プログラムをチェックしてみると、GetArray への2回目の呼び出し (20行目) によってオブジェクトが解放されたこと (14行目) と、同じオブジェクトをもう一度解放しようとしていること (21行目) がわかります。

```

----- fof -- 3 --
ex.c: 22: freeing already freed heap at byte 0 of 32-byte block
free                                  ex
main                                 ex, ex.c, line 22
__start                                ex

This block at address 0x14000d1a0 was allocated at:
malloc                                ex
GetArray                             ex, ex.c, line 10
main                                 ex, ex.c, line 21
__start                                ex

This block was freed at:
free                                  ex
GetArray                             ex, ex.c, line 15
main                                 ex, ex.c, line 21
__start                                ex

```

7-10 Third Degree によるプログラムのデバッグ

詳細については、7.3 節を参照してください。

7.2.3.2 メモリ・リーク

次のプログラムの抜粋は、プログラム終了時のリーク検出を選択した場合 (省略時の設定) に生成されたレポートを示しています。このレポートは、重要度と呼び出しスタックでソートされたメモリ・リークのリストを示しています。

```
-----
New blocks in heap after program exit

Leaks - blocks not yet deallocated but apparently not in use:
* A leak is not referenced by static memory, active stack frames,
  or leaked blocks, though it may be referenced by other leaks.
* A leak "not referenced by other leaks" may be the root of a leaked tree.
* A block referenced only by registers, unseen thread stacks, mapped memory,
  or uninstrumented library data is falsely reported as a leak. Instrumenting
  shared libraries, if any, may reduce the number of such cases.
* Any new leak lost its last reference since the previous heap report, if any.

A total of 160 bytes in 1 leak were found:

160 bytes in 1 leak (including 1 not referenced by other leaks) created at:
  malloc                ex
  GetArray              ex, ex.c, line 10
  main                  ex, ex.c, line 20
  __start                ex

Objects - blocks not yet deallocated and apparently still in use:
* An object is referenced by static memory, active stack, or other objects.
* A leaked block may be falsely reported as an object if a pointer to it
  remains when a new stack frame or heap block reuses the pointer's memory.
  Using the option to report uninitialized stack and heap may avoid such cases.
* Any new object was allocated since the previous heap report, if any.

A total of 0 bytes in 0 objects were found:
```

ソースをチェックすると、GetArray の最初の呼び出しによってメモリ・オブジェクトが解放されなかったことと、それがプログラムの他の箇所でも解放されていないことがわかります。さらに、このオブジェクトへのポインタがプログラムのどこにもないため、“not referenced by other leaks” と認定されます。この区別は、大きなメモリ・リークの本当の原因を発見するのに役立つことがよくあります。

たとえば、大きなツリー構造で、ルートへのポインタが消去されたと仮定します。構造内のすべてのブロックがリークしていますが、ルートを指すポインタが失われていることがリークの本当の原因です。ルート以外のすべてのブロックは、その構造へのポインタを持っているため、他のリークからだけですが、ルートだけが特別に識別されます。したがって、これがメモリ損失の原因である可能性が高いと考えられます。

詳細については，7.4 節を参照してください。

7.2.3.3 ヒープ・ヒストリ

ヒープ・ヒストリが使用可能な場合，Third Degree は動的に割り当てられたメモリに関する情報を収集します。この情報は，アプリケーションが解放するすべてのブロック，およびプログラムの実行が終了した時点でまだ存在しているすべてのブロック（メモリ・リークを含む）について収集されます。次のプログラムの抜粋は，ヒープ割り当てヒストリ・レポートを示しています。

```
-----
Heap Allocation History for parent process

Legend for object contents:
  There is one character for each 32-bit word of contents.
  There are 64 characters, representing 256 bytes of memory per line.
  '.' : word never written in any object.
  'z' : zero in every object.
  'i' : a non-zero non-pointer value in at least one object.
  'pp': a valid pointer or zero in every object.
  'ss': a valid pointer or zero in some but not all objects.

192 bytes in 2 objects were allocated during program execution:

-----
160 bytes allocated (8% written) in 1 objects created at:
  malloc          ex
  GetArray        ex, ex.c, line 10
  main            ex, ex.c, line 20
  __start         ex

Contents:
  0: i.ii.....

-----
32 bytes allocated (38% written) in 1 objects created at:
  malloc          ex
  GetArray        ex, ex.c, line 10
  main            ex, ex.c, line 21
  __start         ex

Contents:
  0: i.ii....
```

このサンプル・プログラムでは，合計 192 バイト ($8 \times (20+4)$) の 2 個のオブジェクトが割り当てられています。各オブジェクトは，異なる呼び出しスタックから割り当てられているため，ヒストリには 2 つのエントリがあります。各配列の最初の数バイトだけが有効な値に設定され，その結果，ここに示されているような書き込み率になります。

このサンプル・プログラムが実際のアプリケーションだとすると、初期化されている動的メモリが極端に少ないことから、アプリケーションによるメモリの使用が非効率的であることがわかります。

詳細については、7.4.4 項を参照してください。

7.2.3.4 メモリ・レイアウト

レポートのメモリ・レイアウト・セクションでは、プログラムが使用するメモリの概要を、サイズおよびアドレス範囲により示します。次のプログラムの抜粋は、メモリ・レイアウト・セクションを示しています。

```
-----
memory layout at program exit
  heap      40960 bytes [0x14000c000-0x140016000]
  stack     2720 bytes [0x11ffff560-0x120000000]
ex data     48528 bytes [0x140000000-0x1400bd90]
ex text     1179648 bytes [0x120000000-0x120110000]
-----
```

示されたヒープ・サイズおよびアドレス範囲は、プログラムの終了時に `sbrk(0)` により返された値（ヒープ・ブレイク）を反映します。このため、サイズは、プロセスに割り当てられたヒープ・スペース全体を表します。Third Degree は、この `sbrk(0)` の解釈を変更するような `malloc` 変数の使用をサポートしません。

スタック・サイズとアドレス範囲は、プログラムの実行中にメイン・スレッドのスタック・ポインタが達する最下位アドレスを反映します。つまり、Third Degree は、計測機構付きプロシージャが呼び出されるたびに、最下位アドレスを追跡します。この値がスタック・サイズの最大値を反映するには、すべてのシェアード・ライブラリが計測機構付きである必要があります（たとえば、スレッド化されていないプログラムでは、`third(1)` コマンドの `-all` オプションを使用し、`dlopen(3)` を使ってロードしたライブラリでは `-incobj` オプションを使用します）。スレッドのスタック（`pthread_create` を使用して作成された）は含まれません。

データやテキストのサイズおよびアドレス範囲は、実行可能プログラムの静的な部分や各シェアード・ライブラリがロードされた場所を示します。

7.3 Third Degree エラー・メッセージの解釈

Third Degree は、回復不可能なエラーとメモリ・アクセス・エラーの両方を報告します。回復不可能なエラーには、次のものが含まれます。

- 不正パラメータ
たとえば, `malloc(-10)` など。
- 失敗した割り当て機能
たとえば, `malloc` が 0 を返して, 利用可能なメモリがないことを示した場合など。
- 非ゼロの引数を指定した `brk` 関数への呼び出し
Third Degree では, 非ゼロの引数を指定して `brk` を呼び出すことはできません。
- シグナル・ハンドラにおけるメモリ割り当ての不許可

回復不可能なエラーが起こると, 計測機構付きアプリケーションが, ログ・ファイルをフラッシュした後にクラッシュします。アプリケーションがクラッシュした場合には, まずログ・ファイルをチェックしたのち, `third(1)` コマンド行に `-g` を指定して, デバッガのもとでそれを再実行します。

メモリ・エラーには, 次のものが含まれます (英字 3 文字の短縮形で表します)。

名前	エラー
<code>ror</code>	範囲外の読み込み。ヒープ, スタック, 静的領域のいずれでもない (たとえば <code>NULL</code>)。
<code>ris</code>	スタック内の無効なデータの読み取り。多くの場合, 配列境界エラー。
<code>rus</code>	スタック内の, 有効だが初期化されていない記憶位置の読み取り。
<code>rih</code>	ヒープ内の無効なデータの読み取り。多くの場合, 配列境界エラー。
<code>ruh</code>	ヒープ内の, 有効だが初期化されていない記憶位置の読み取り。
<code>wor</code>	範囲外の書き込み。ヒープ, スタック, 静的領域のいずれでもない。
<code>wis</code>	スタックへの無効なデータの書き込み。多くの場合, 配列境界エラー。
<code>wih</code>	ヒープへの無効なデータの書き込み。多くの場合, 配列境界エラー。
<code>for</code>	範囲外の解放。ヒープ, スタックのいずれでもない。
<code>fis</code>	スタック内のアドレスの解放。

名前	エラー
fih	ヒープ内の無効なアドレスの解放。有効なオブジェクトがない。
fof	すでに解放されたオブジェクトの解放。
fon	null ポインタの解放 (単なる警告)。
mrn	malloc による null の戻し。

特定のメモリ・エラーの報告は、`-ignore` オプションを 1 つまたは複数指定することによって抑制できます。これは、ソースのないライブラリ関数内でエラーが発生した場合に役立つことがあります。Third Degree を使用すると、個々のプロシージャおよびファイル内の特定のメモリ・エラーを特定の行番号で抑制できます。詳細は、`third(1)` を参照してください。

代わりに、`-excobj` を指定するか、または `-all` や `-incobj` オプションを省略することによって、チェックのためにライブラリを選択しないという方法もあります。

7.3.1 エラーの修正とアプリケーションの再試行

Third Degree が、計測機構付きプログラムから多数の書き込みエラーを報告する場合は、最初のいくつかのエラーを修正してから、プログラムを再計測します。書き込みエラーは悪化する可能性があるだけでなく、Third Degree 内部のデータ構造体を破壊することもあります。

7.3.2 初期化されていない値の検出

初期化されていない値の使用を検出するという Third Degree の手法により、動作していたプログラムが、計測時に異常終了することがあります。たとえば、プログラムが、`malloc` 関数への最初の呼び出しでゼロに初期化されたブロックが返されるということを前提にしている場合は、Third Degree がすべてのブロックを非ゼロ値 (省略時の設定では `0xff8a5a5`) にしてしまうため、そのプログラムの計測機構付きバージョンは異常終了します。

逆参照またはその他の方法でこの初期化されていない値を使用したために発生したシグナルを検出した場合には、Third Degree は次の形式のメッセージを表示します。

```
*** Fatal signal SIGSEGV detected.
*** This can be caused by the use of uninitialized data.
*** Please check all errors reported in app.3log.
```

初期化されていないデータの使用は、計測機構付きプログラムがクラッシュする最大の原因です。問題の原因を判別するには、まず、初期化されていないスタックの読み取りと初期化されていないヒープの読み取りによるエラーがないかどうかを、ログ・ファイルでチェックします。ほとんどの場合、ログ・ファイルの最後のエラーのうちの 1 つが問題の原因です。

エラーの発生源を示している問題がある場合は、それが本当に初期化されていないデータの読み取りに起因しているかどうかを、`-uninit` オプション (またはオプション全体) から `heap` および `stack` オプションを削除することで確認できます。`stack` を削除すると、通常、Third Degree が各プロセスのエントリで実行する、新規に割り当てられたスタック・メモリの非ゼロ値による埋め込みが不能に設定されます。同様に、`heap` の削除は、各動的メモリ割り当てで実行される、ヒープ・メモリの初期化を不能に設定します。これらのオプションの一方または両方を使用することにより、計測機構付きプログラムの動作を変更して、そのプログラムを正常に完了させることができる場合があります。これは、計測機構付きプログラムがどのタイプのエラーでクラッシュしたかを判別するのに役立ちます。その結果、ログ・ファイル内の特定のメッセージに集中できるようになります。

代わりに、計測機構付きプログラムをデバッガで実行して (`third(1)` コマンドの `-g` オプションを使用)、失敗の原因を削除する方法もあります。メモリ・リークをチェックしたいだけの場合は、`-uninit` オプションを使用する必要はありませんが、`-uninit` オプションを使用すると、より正確なリークのレポートを得ることができます。

プログラムによってシグナル・ハンドラが設定される場合は、Third Degree が省略時のシグナル・ハンドラを変更しても、それを妨げる可能性はほとんどありません。Third Degree は、通常プログラムのクラッシュを起こすシグナル (`SIGILL`, `SIGTRAP`, `SIGABRT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, `SIGXCPU`, および `SIGXFSZ` を含む) に対してだけ、シグナル・ハンドラを定義します。Third Degree によるシグナルの処理は、`-signals` オプションを指定することにより不能にすることができます。

7.3.3 ソース・ファイルの探索

Third Degree は、各エラー・メッセージの前に、コンパイラが使用するスタイルでファイルおよび行番号を付加します。たとえば、次のようになります。

```
----- fof -- 3 --
ex.c: 21: freeing already freed heap at byte 0 of 32-byte block
    free                                malloc.c
    main                                ex.c, line 21
    __start                             crt0.s
```

Third Degree は、エラーの発生源にできる限り近づいて指摘しようとしません。通常は、この例のように、エラー発生時の呼び出しスタックの最上位に近いプロシージャのファイルと行番号を表示します。ただし、それがライブラリの中にあったり、現在のディレクトリになかったりすると、Third Degree はこのソース・ファイルを見つけることができません。この場合、Third Degree は、指摘可能なソース・ファイルを見つけるまで、呼び出しスタックを下へ移動します。通常は、これはライブラリ・ルーチン呼び出しのポイントです。

これらのエラー・メッセージにタグをつけるには、Third Degree がプログラムのソース・ファイルの位置を判断しなければなりません。ソース・ファイルを含むディレクトリで Third Degree を実行している場合、Third Degree はそのディレクトリにあるソース・ファイルを探索します。そうでない場合、Third Degree の探索パスにディレクトリを追加するには、`-use` オプションを 1 つまたは複数指定します。これにより、Third Degree は、他のディレクトリに含まれるソース・ファイルを見つけることができますようになります。各ソース・ファイルの位置は、そのファイルが探索された探索パスの最初のディレクトリです。

7.4 アプリケーションによるヒープ使用の検査

適切に割り当てられたメモリだけがアクセスおよび解放されていることを確認する実行時の検査に加え、Third Degree は、アプリケーションによるヒープの使用法を理解するために、次の 2 つの方法を提供しています。

- メモリ・リークを見つけて、報告できる。
- ヒープの内容をリストできる。

省略時の設定では、Third Degree はプログラムの終了時にリークがないかどうかを検査します。

この節では、Third Degree により供給された情報を使用して、アプリケーションによるヒープの使用法を分析する方法を説明します。

7.4.1 メモリ・リークの検出

メモリ・リークは、使用中のポインタが存在しないヒープ内のオブジェクトです。このオブジェクトは、アクセスすることができず、使用も解放もできません。これは役に立たず、消えることもなく、メモリの無駄使いです。

Third Degree は、単純なトレース・アンド・スイープ (trace-and-sweep) のアルゴリズムを使用して、メモリ・リークを見つけます。Third Degree はルートのセット (現在アクティブなスタックおよび静的領域) から開始して、ヒープ内のオブジェクトへのポインタを見つけ、これらのオブジェクトに訪問済みのマークをつけます。次に、これらのオブジェクトの中にある潜在的なすべてのポインタを再帰的に見つけ、最後にヒープをスイープして、マークが付いていないオブジェクトをすべて報告します。マークの付いていないこれらのオブジェクトがリークです。

トレース・アンド・スイープのアルゴリズムによって、循環構造も含め、すべてのリークが見つかります。このアルゴリズムは保守的であり、型情報がない場合、適切に境界合わせされ、ヒープ内の有効なオブジェクトの内部を指し示す 64 ビット・パターンは、すべてポインタとみなされます。このように想定することにより、次のような問題が滅多に起こらないようにします。

- Third Degree は、オブジェクトの先頭または内部のいずれかへのポインタを実ポインタとみなします。これらが含むアドレスへのポインタをもたないオブジェクトだけがリークとみなされます。
- 他のプロセスのアドレス空間に格納したり、またはコード化することにより、計測機構付きアプリケーションが本当のポインタを隠している場合は、Third Degree は見かけ上のリークを報告します。Third Degree でそのようなアプリケーションを計測するときは、`-mask` オプションを指定します。このオプションを使用すると、すべての潜在的なポインタに対して AND 演算子として適用されるマスクが指定できます。たとえば、ポインタの先頭の 3 ビットをフラグとして使用する場合は、`0x1fffffffffffff` のマスクを指定します。詳細については、`third(1)` を参照してください。
- Third Degree は、ヒープ・ポインタと似ているビット・パターン (文字列、整数、浮動小数点数、およびパック構造体など) があると、それを本当のポインタと混同し、そのために本当のリークを見落とす可能性があります。

- Third Degree は、最適化されたコードがメモリではなくレジスタにだけ格納したポインタは認識しません。その結果、誤ったリーク・レポートが作成されることがあります。

リーク・レポートの精度を最大限にするには、`-uninit h+s` および `-all` オプションを使用します。ただし、`-uninit` オプションを指定すると、プログラムが異常終了する恐れがあり、`-all` オプションでは、計測と実行時間が増加します。このため、Leaks と Objects のリストだけをチェックして、プログラム・エラーの可能性を調べてください。

7.4.2 ヒープの読み取りとリーク・レポート

コマンド・オプションを指定すると、すべてのヒープ・オブジェクトまたはリークについての前回のレポートまたはリスト以降の、新しいヒープ・オブジェクトまたはリークだけをリストした、ヒープおよびリークの増分レポートを Third Degree に生成させることができます。これらのレポートは、プログラムの終了時、あるいはユーザ指定の関数への n 回目の呼び出しの前または後に要求できます。`-blocks`、`-every`、`-before`、および `-after` オプションの詳細については、`third(1)` を参照してください。`-blocks` オプション (省略時の設定) では、ヒープ内のリークとオブジェクトの両方がレポートされるため、間違ったタイプとして分類されたイベント内のリークやオブジェクトを見落とすことがありません。`.3log` ファイルには、間違った分類が生じる可能性のある状況が、レポートの正確さを向上させる方法と一緒に記述されています。

Third Degree は、レポートされたブロックが本当にリークしていることを示す証拠を見つけているが、一方、その証拠は、オブジェクトとしてレポートされたブロックがそうでないことを示しているので、リークのレポートは注意深く見る必要があります。ただし、プログラムのデバッグやチェックにより、そうでないことがわかった場合は、ツールが証拠を誤って解釈したと推断することができます。

Third Degree は、関係するバイト数に基づいて重要度を減少させることにより、メモリ・オブジェクトおよびリークをレポートにリストします。そこでは、同じ呼び出しスタックで割り当てられたオブジェクトは、まとめてグループ化されます。たとえば、同一の呼び出しシーケンスによって、1 バイトのオブジェクトが 100 万割り当てられた場合には、Third Degree はそれらを 100 万の割り当てを含む 1 MB のグループとして報告します。

オブジェクトまたはリークが同じであり、レポートの中でグループ化する必要がある場合 (またはオブジェクトやリークが異なり、そのためにグループ化するべきでない場合) に Third Degree に指示するには、`-depth` オプションを指定します。このオプションは、Third Degree がリークまたはオブジェクトを区別するために使用する呼び出しスタックの深さを設定します。たとえば、オブジェクトに対して 1 という深さを指定した場合、Third Degree は、呼び出し元の関数に関係なく、ヒープ内の有効なオブジェクトを割り当てた関数と行番号により、それらをグループ化します。逆に、リークに対して非常に大きな深さを指定した場合、Third Degree は、`main` から上方の、同一の呼び出しスタックのあるポイントに割り当てられたリークだけをグループ化します。

ほとんどのヒープのレポートでは、最初の 2, 3 のエントリが記憶域のほとんどの原因となっていますが、小さなエントリの非常に長いリストがあることもあります。レポートの長さを制限するには、`-min` オプションが使用できます。このオプションは、リークしたメモリ総量、またはオブジェクトが使用中のメモリ総量のパーセンテージをしきい値として定義します。残りの小さいリークまたはオブジェクトの合計が、このしきい値よりも小さい場合には、Third Degree は、これらを最後の 1 つのエントリにまとめてグループ化します。

注意

`realloc` 関数は (`malloc`, `copy`, および `free` への呼び出しを含めることにより)、常に新規のオブジェクトを割り当てるため、これを使用すると、Third Degree レポートの解釈が直観に反したものになります。オブジェクトは、異なる識別で 2 回リストされることがあります。

リークとオブジェクトは相互に排他的です。オブジェクトは、ルートから到達可能でなければなりません。

7.4.3 リークの探索

メモリ・リークを探索する時点は、常にはっきりしているわけではありません。省略時の設定では、Third Degree はプログラムの終了後にリークがないかどうかをチェックしますが、これが常にユーザの要求するものとは限りません。

リークの検出は、使用されている構造体がすべて有効範囲内にあるうちに、プログラムの終了にできるだけ近いところで行うのが最もよい方法です。ただし、リーク検出用のルートは、スタックおよび静的領域の内容であることを憶えておいてください。プログラムが `main` から戻って終了し、その構造体の 1 つへの唯一のポインタがスタックに保持されている場合には、リーク探索の間、このポインタはルートとして参照されず、リークしたメモリとして誤って報告されることになります。たとえば、次のような場合です。

```
1 main (int argc, char* argv[]) {
2     char* bytes = (char*) malloc(100);
3     exit(0);
4 }
```

このプログラムを計測する場合に、`-blocks all -before exit` を指定すると、Third Degree はリークを検出しません。プログラムが `exit` 関数を呼び出すとき、`main` の変数はすべてまだ有効範囲内にあります。

ここで、次の例を検討してください。

```
1 main (int argc, char* argv[]) {
2     char* bytes = (char*) malloc(100);
3 }
```

同じオプションを指定して(または指定しないで)このプログラムを計測すると、チェックが行われる前に `main` が戻ったために、Third Degree のリーク検査によって記憶域リークが報告されることがあります。これら 2 つの動作内容は、`bytes` が本当のリークか、単に `main` が戻った時点でまだ使用中のデータ構造体であるかにより、いずれも正しいものです。

リーク検出を実行する時点を理解するために、プログラムを慎重に読むのではなく、指定したプロシージャへの指定した回数の呼び出しの後に、新しいリークがないかを検査できます。次のオプションを指定して、省略時のリーク・チェックを無効にし、プロシージャ `proc_name` への各 10,000 回目の呼び出しの前にリークを要求します。

```
-blocks cancel
-blocks new -every 10000 -before proc_name
```

7.4.4 ヒープ・ヒストリの解釈

`-history` オプションを使用して、このプログラムを計測すると、Third Degree がプログラムのヒープ・ヒストリを生成できます。ヒープ・ヒストリを使用することにより、プログラムが実行中に動的メモリをどのように使用したかがわかります。たとえば、この機能を使用すると、データ構造体の未使

用のフィールドを削除したり，または処理中のフィールドをパックしてメモリを効率的に使用することができます。ヒープ・ヒストリは，割り当てられたけれど，アプリケーションが使用しなかったメモリ・ブロックも示します。

ヒープ・ヒストリが使用可能に設定されている場合，Third Degree は，動的に割り当てられた各オブジェクトについての情報を，それがアプリケーションによって解放される時点で収集します。プログラムの実行が完了すると，Third Degree は，まだ有効なすべてのオブジェクトについてこの情報（メモリ・リークも含む）をアセンブルします。各オブジェクトごとに，Third Degree はオブジェクトの内容を見て，各ワードを，アプリケーションが書き込まなかったもの，0，有効なポインタ，またはその他の値に分類します。

Third Degree は次に，各オブジェクトに関する情報を，プログラム内の同じ呼び出しスタックで割り当てられたその他のオブジェクトすべてについて収集した情報とマージします。その結果，任意の型のすべてのオブジェクトの使用に関する累積的な様子が提供されます。

Third Degree では，プログラムの存在期間中に割り当てられたすべてのオブジェクトと，それらの内容が使用された目的の要約が提供されます。このレポートは，割り当てポイント（たとえば `malloc` または `new` などの割り当て機能を持つ関数が呼び出された箇所の呼び出しスタック）ごとに 1 つのエントリを示します。エントリは，割り当て量で降順にソートされます。

各エントリは，次の内容を提供します。

- 割り当てられているすべてのオブジェクトに関する情報
- 割り当てられている総バイト数
- 割り当てられている総オブジェクト数
- 割り当てられたオブジェクトのうち，書き込みが行われたバイトのパーセンテージ
- 呼び出しスタック，およびその呼び出しスタックが割り当てたすべてのオブジェクトの内容の累積マップ

各エントリの内容部分は，オブジェクトがどのように使用されたかを記述します。割り当てられたすべてのオブジェクトが同じサイズではない場合，Third Degree は，すべてのオブジェクトに共通な最小のサイズだけを考慮します。非常に大きな割り当てに関しては，オブジェクトの最初の部分の内容

だけを要約します。省略時の値では、これは最初の 1 KB です。-size オプションを指定することによって、最大サイズの値が調整できます。

エントリの内容部分では、Third Degree は次の文字の 1 つを使用して、チェックする 32 ビットの各ロングワードを表します。

文字	説明
ドット(.)	オブジェクトに書き込まれなかったロングワードを示し、メモリが浪費されている明らかなしるしです。通常、より詳しい分析を行って、これが単にテストが不十分でこのフィールドを使用しなかっただけなのか、フィールドのスワップまたはよりよい型を選択すれば解決する埋め込みの問題であるのか、あるいはこのフィールドがもはや使われていないものなのかを確認する必要があります。
z	すべてのオブジェクトで、値が常に 0 (ゼロ) であるフィールドを示します。
pp	ポインタ、つまりスタック、静的データ領域、ヒープへの有効なポインタだった (または 0 だった) 64 ビットの量を示します。
ss	サムタイム・ポインタを示します。このロングワードは、最低 1 つのオブジェクトではポインタと似ていましたが、すべてのオブジェクトでそうであったわけではありません。これは、一部のインスタンスの初期化されていないポインタまたは共用体である可能性があります。ただし、これは重大なプログラミング・エラーのしるしかもしれません。
i	最低 1 つのオブジェクトでなんらかの非ゼロ値で書き込まれたが、どのオブジェクトでもポインタ値を含んでいなかったロングワードを示します。

エントリが 100 MB を割り当てているとリストされていても、割り当てられたオブジェクトが任意の時点で 100 MB のヒープ・ストレージを使用したことを意味するわけではありません。これは累積の数字であり、この時点までのプログラムの存在期間全体で 100 MB が割り当てられていることを示しています。この 100 MB は、解放されたかもしれないし、リークしたか、あるいはまだヒープ内にあるかもしれません。この数字は、単にこの割り当て関数が非常に多くの処理をしていることを示しています。

理想的には、実際に書き込まれたバイトの小数部は、常に 100 パーセントに近くならなければなりません。ずっと低い場合には、割り当てられたバイトの一部が使用されていません。パーセンテージが低い一般的な理由には、次のものがあります。

- 大きなバッファが割り当てられたが、これまで小さな部分しか使用されていません。

- 任意の型のすべてのオブジェクトの各部分が使用されていません。これらは、忘れられたフィールドか、または C 構造体の境界合わせ規則に起因する実在のフィールド間の埋め込みの場合があります。
- あるオブジェクトが割り当てられたが、使用されていません。オブジェクトのポインタが破棄された場合、リーク検出によりこれらのオブジェクトが発見されることがあります。しかし、それらが空きリストに保持されている場合は、ヒープ・ヒストリでしか見つけれられません。

7.5 シンボル情報が不十分なプログラムにおける Third Degree の使用

計測した実行可能プログラムにシンボル情報がほとんどなく、Third Degree がいくつかのプログラム記憶位置を正確に指摘することが困難な場合、Third Degree は、プロシージャ名、ファイル名、または行番号が認識できないというメッセージを出力します。たとえば、次のようになります。

```
----- rus -- 0 --
reading uninitialized stack at byte 40 of 176 in frame of main
  proc_at_0x1200286f0      libc.so
  pc = 0x12004a268        libc.so
  main                   app, app.c, line 16
  __start                 app
```

Third Degree はスタック・トレースのプロシージャ名をプリントしようとしていますが、(これが静的なプロシージャであるために) プロシージャ名が失われている場合は、Third Degree は計測機構付きプログラムのプログラム・カウンタをプリントします。この情報により、デバッガを使って記憶位置を見つけることができます。プログラム・カウンタが使用できない場合には、Third Degree は名前のないプロシージャのアドレスをプリントします。

もっと頻繁に起こるのは、ファイルが省略時の `-g0` オプションでコンパイルされたためにファイル名または行番号が使用できない場合です。この場合、Third Degree は、プロシージャが見つかったオブジェクトの名前をプリントします。このオブジェクトは、メイン・アプリケーションまたはシェアド・ライブラリのいずれかです。

省略時の設定では、エラー・レポートがプリントされるのは、ソース・ファイル名と行番号のあるスタック・フレームが、スタックの上の 2 つのフレーム内に表示される場合に限ります。これにより、システム・ライブラリ内の高度に最適化されたアセンブラ言語コードによって生成される可能性のある偽のレポートが隠されます。デバッグ不能なプロシージャに関連するレポートが隠されるのを少なく(多く)するには、`-hide` オプションを使用します。

シンボル情報がないためにデバッグが困難な場合は、シンボル情報を多くしてプログラムを再コンパイルすることを検討します。 `-g` または `-g1` オプションをつけて再コンパイルし、`-x` オプションなしでリンクします。 `-g` オプションを使用すると、レポートには、前述の例にあるようなバイト・オフセットではなく、変数名が表示されます。

7.6 Third Degree エラー・レポートの有効性検査

まれに、次のような見かけ上のエラーが発生することがあります。

- 次に示す例のように、変数、配列の要素、または 32 ビットより小さい構造体のメンバ (たとえば `short`, `char`, ビット・フィールドなど) に対して行われた変更。

```
void Packed() {  
    char c[4];  
    struct { int a:6; int b:9; int c:4 } x;  
    c[0] = c[1] = 1;          /* rus errors here ... */  
    x.a = x.c = x.e = 3;      /* ... maybe here */  
}
```

`strcpy`, `memcpy`, `printf` などについてレポートされたありそうもないエラー・メッセージは無視してください。

- Third Degree は、新規に割り当てられたメモリに特別な値を埋め込んで、初期化されていない変数への参照を検出します (7.3.2 項を参照)。この特別な値をメモリに明示的に格納し、それを後で読み取るプログラムは、見かけ上の「初期化されていないメモリの読み取り」エラーを起こすことがあります。
- 可変サイズのスタック・フレームはサポートされていません。“invalid stack” に関するメッセージはすべて無視してください。

誤った正の数を見つけたと思う場合は、エラーが報告されたプロシージャにデバッガを使用することにより確認できます。Third Degree によって報告されるすべてのエラーは、アプリケーションのロードおよび格納時に検出され、エラー・レポートに示される行番号は、逆アセンブルによる出力に示されるものと一致します。 `-g` オプションを指定してプログラムのコンパイルおよび計測を行ってから、デバッグをします。

7.7 検出されないエラー

Third Degree は、次のような、実在のエラーの検出に失敗することがあります。

- 32 ビットよりも小さい量の演算のエラー (たとえば, `char`, `short`, ビット・フィールドなど) は、検出されない可能性があります。
`-uninit repeat` オプションは、より多くのロード/ストア操作をチェックすることにより、そのようなエラーを見つけることができます。これらは、Third Degree では、通常、リスクが非常に小さいためチェックの必要がないとみなされているものです。
- Third Degree は、ヒープにおける誤ったオブジェクトの偶発的なアクセスは検出できません。検出できるのは、オブジェクトからのメモリ・アクセスだけです。たとえば、Third Degree は、`a[last+100]` が `b[0]` と同じアドレスであることを判断することはできません。オブジェクトに付加される埋め込みの量を変更することにより、これが発生する可能性を減少させることができます。これを行うには、`-pad` オプションを指定します。
- Third Degree は、アプリケーションが配列のスタック・フレームの終わりまたはそのヒープ・オブジェクトを超えた場合に、これを検出できないことがあります。Third Degree は、ヒープ内のオブジェクトを「ガード・ワード」で囲むため、小さな配列境界エラーを見落とすことがあります (ガードはオーバーシュートを検出します)。スタックでは、隣接メモリにローカル変数が含まれている可能性が高く、Third Degree は、より大きな境界エラーの検出に失敗することがあります。たとえば、`sprintf` 演算をはるかに小さなローカル・バッファに対して実行することは検出されますが、配列の境界を 2, 3 のワード分超えただけで、十分なローカル変数が配列を取り囲んでいる場合には、エラーは検出されない可能性があります。配列の境界違反をもっと厳密に検出するには、`cc` コマンドの `-check_bounds` オプションを使用します。
- ポインタをコード化したり、あるいはそれらをヒープ・オブジェクトの内部だけにとどめておくことによってポインタを隠すと、Third Degree のリーク検出の有効性が低下します。
- コンパイラの最適化が無効に設定されている (つまり、`-O0` および `-inline none` オプションが指定されている) 場合、Third Degree はより多くの初期化されていない変数を検出することがあります。

- 時々、古いポインタがメモリ内で見つかったために、リークが報告されないことがあります。初期化されていないヒープ・メモリのチェックを選択すると (`-uninit heap`)、この問題を減らすことができます。
- コンパイラが本質的でないと見なす命令は最適化されることがあるため、どの程度の最適化であってもリーク報告の結果がゆがめられます。



プログラムのプロファイルによる性能の向上

プロファイルとは、全実行時間のうちの大きな割合をどのコード・セクションが消費しているかを識別するためのメソッドです。通常のプログラムでは、実行時間のほとんどはコード内の比較的少数のセクションで消費されます。プログラムの性能を向上させるためには、実行時間を集中的に消費するセクションのコーディング効率を向上させることが最も有効です。

Tru64 UNIX では、性能の向上のために次の 4 つの方法をサポートしています。

- 自動最適化とプロファイル主導の最適化 (10.1 節 でこのような最適化技術の全体を説明)
- 手動による設計とコードの最適化 (8.3 節)
- システム・リソースの使用の最小化 (8.4 節)
- テスト・ケースの重要性の確認 (8.5 節)

これらの方法のいずれか 1 つだけで十分な場合もありますが、1 つの方法でプログラムの性能のすべての面に対処できない場合は、複数の方法を組み合わせる方がよいかもしれません。自動最適化とプロファイル主導の最適化は、性能を改善するための最も簡単な方法です。この章では、最後の 3 つ (手動) の性能改善方法について説明するとともに、それをサポートするために Tru64 UNIX で提供するツールについて説明します。また、次の項目についても説明します。

- この章の例で使用されているサンプル・プログラムのソース・コード sample (8.1 節)
- プロファイルのコンパイル・オプション (8.2 節)
- 表示するプロファイル情報の選択 (8.6 節)
- プロファイル・データ・ファイルのマージ (8.7 節)
- マルチスレッド・アプリケーションのプロファイル (8.8 節)

- monitor ルーチンを使用したプロファイルの制御 (8.9 節)

詳細については、次のリファレンス・ページおよびドキュメントを参照してください。

- プロファイル: cc(1), hiprof(1), pixie(1), third(1), uprofile(1), prof(1), gprof(1)
- システムのモニタリング: ps(1), swapon(8), vmstat(1)
- Performance Manager (「Tru64 UNIX Associated Products Volume 1」CD-ROM で提供): pmgr(8X)
- グラフィカル・ツール (「Tru64 UNIX Associated Products Volume 1」CD-ROM の Graphical Program Analysis サブセット, または Compaq Enterprise Toolkit for Windows/NT desktops with the Microsoft VisualStudio97 の一部として提供): dxheap(1), dxprof(1), mview(1), pview(1)
- Visual Threads (「Tru64 UNIX Associated Products Volume 1」CD-ROM で提供): dxthreads(1)。Visual Thread を使用すると、マルチスレッド・アプリケーションの潜在的な論理および性能上の問題を解析できます。
- 『システムの構成とチューニング』

8.1 プロファイルのサンプル・プログラム

この章で取り上げる例では、sample プログラムを参照しています。sample プログラムのソース・コードのファイルは、profsample.c (main プログラム), add_vector.c, mul_by_scalar.c, print_it.c, profsample.h です。これらのファイルを例 8-1 に示します。

例 8-1: プロファイルのサンプル・プログラム

```
***** profsample.c *****
#include <math.h>
#include <stdio.h>
#include "profsample.h"

static void mul_by_pi(double ary[])
{
    mul_by_scalar(ary, LEN/2, 3.14159);
}
```

8-2 プログラムのプロファイルによる性能の向上

例 8-1: プロファイルのサンプル・プログラム (続き)

```
void main()
{
    double ary1[LEN];
    double *ary2;
    double sum = 0.0;
    int i;

    ary2 = malloc(LEN * sizeof(double));
    for (i=0; i<LEN; i++) {
        ary1[i] = 0.0;
        ary2[i] = sqrt((double)i);
    }
    mul_by_pi(ary1);
    mul_by_scalar(ary2, LEN, 2.71828);
    for (i=0; i<100; i++)
        add_vector(ary1, ary2, LEN);
    for (i=0; i<100; i++)
        sum += ary1[i];
    if (sum < 0.0)
        print_it(0.0);
    else
        print_it(sum);
}
***** profsample.h: *****

void mul_by_scalar(double ary[], int len, double num);
void add_vector(double arya[], double aryb[], int len);
void print_it(double value);
#define LEN 100000

***** add_vector.c: *****

#include "profsample.h"

void add_vector(double arya[], double aryb[], int len)
{
    int i;
    for (i=0; i<LEN; i++) {
        arya[i] += aryb[i];
    }
}
***** mul_by_scalar.c: *****

#include "profsample.h"

void mul_by_scalar(double ary[], int len, double num)
{

```

例 8-1: プロファイルのサンプル・プログラム (続き)

```
int i;
for (i=0; i<LEN; i++) {
    ary[i] *= num;
}
}
***** print_it.c: *****

#include <stdio.h>
#include "profsample.h"

void print_it(double value)
{
    printf("Value = %f\n", value);
}
```

8.2 プロファイルのコンパイラ・オプション

cc コマンドのデバッグ・オプションと最適化オプションを使用する場合には、次の点に注意してください。これらの注意事項は、特に断らない限り、この章で説明するすべてのプロファイル・ツールに適用されます。

- -g1 オプションを使用すると、必要最低限のデバッグ情報 (行番号とプロシージャ名) が得られ、これはすべてのプロファイルに対して十分なものです。cc コマンドの省略時の設定の -g0 でも構いませんが、ローカルなプロシージャ (静的なプロシージャなど) の名前は得られません。-g2 以上のオプションでは、最適とはいえないコードが、不必要な情報とともに提供されます。
- 手動の最適化を行う場合は、どのプロファイル・ツールでもインライン・プロシージャをそれぞれの名前で別々に表示しないことに注意してください。インライン・プロシージャのプロファイル統計情報は、呼び出し元のプロシージャの統計情報に含まれます。たとえば、proc1 が proc2 (インライン化されている) を呼び出す場合は、proc2 で消費される時間が proc1 の統計情報に含まれます。したがって、プロファイル時に最小限のインライン化を行って、ある程度の最適化効果を得るためには、-O2 (または -O) オプションを使用します。場合によっては、-inline none オプションを指定して、すべてのインライン化を排除する必要があります。この制限は、10.1 節で述べるように自動最適化には適用されません。

8.3 手動による設計とコードの最適化

この節では、手動による設計とコードの最適化に使用する手法とツールについて説明します。

8.3.1 手法

10.1 節で説明した自動最適化の効果は、プログラムが使用するアルゴリズムの効率によって制限されます。アルゴリズムとデータ構造を手動で最適化することにより、プログラムの性能をさらに向上させることができます。このような最適化には、 N 乗から対数 N へ複雑さを低減したり、データのコピーを避けたり、使用するデータの量を減らしたりすることなどがあります。また、プログラムを実行する特定のマシンのアーキテクチャに合わせてアルゴリズムを調整する場合があります。たとえば、処理のフェーズごとに配列全体をデータ・キャッシュに読み込むかわりに、大きな配列を小さなブロックに分けて処理し、処理全体に対して各々のブロックがデータ・キャッシュに残るようにします。

Tru64 UNIX では、手動による最適化にプロファイル・ツールを使用できます。このツールでは、アプリケーションの中で最大の CPU 負荷を課す部分 (CPU サイクル、キャッシュ・ミスなど) を特定します。プログラムのさまざまなプロファイルを評価することにより、プログラムのどの部分が最も多くの CPU リソースを使用しているかを特定することができ、それらの部分のアルゴリズムの設計やコーディングを見直して、リソースの使用量を減らすことができます。また、プロファイルを使用すると、リソース使用の少ないコードではなく、最もリソース使用の激しいコードに作業を集中することができるため、作業の費用対効果が大きくなります。

8.3.2 ツールと例

以降の各項では、呼び出しグラフを使用して CPU 時間のプロファイルを行うためのツールと、ソース行と命令を使用して CPU 時間またはイベントのプロファイルを行うためのツールについて説明します。

8.3.2.1 呼び出しグラフを使用した CPU 時間のプロファイル

呼び出しグラフのプロファイルは、各プロシージャで使用された CPU 時間と、そのプロシージャが呼び出した他のすべてのプロシージャで使用された CPU 時間を示します。このプロファイルでは、プログラムの中のどのフェー

ズまたはサブシステムが全 CPU 時間のうち最も多くの時間を費やしたのかを示すことができるため、プログラムの性能を全般的に理解することができます。この項では、この情報を得るための 2 つのツールについて説明します。

- hiprof プロファイラ (8.3.2.1.1 項)
- cc コマンドの -pg オプション (8.3.2.1.2 項)

どちらのツールも gprof ツールとともに明示的または暗示的に使用して、プロファイルのフォーマットと表示を行います。

オプションとして dxprof コマンドを使用すると、CPU 時間の呼び出しグラフのプロファイルをグラフィカル表示できます。

8.3.2.1.1 hiprof プロファイラを使用する方法

hiprof プロファイラ (hiprof(1) を参照) は、プログラムを計測し、計測機構付きプログラムの実行中に呼び出しグラフを生成します。このプロファイラでは、プログラムは 8.2 節に示した以外の特定の方法でコンパイルされている必要はありません。hiprof コマンドでは、シェアード・ライブラリとプログラム・コードの呼び出しグラフを作成できます。その際には、適度な最適化を行い、最小限のデバッグ情報を付けることができます。たとえば、次のようにコマンドを実行します。

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c[1]
% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm[2]
% hiprof -numbers -L. -inc libsample.so sample[3]
```

- ^[1] 8.2 節で説明したように、3 つのソース・モジュールからシェアード・ライブラリ libsample.so を、最適化およびデバッグ情報付きで作成します。
- ^[2] ソース・モジュール profsample をコンパイルし、シェアード・ライブラリ libsample.so (現在のディレクトリ上) とリンクして、実行可能ファイル sample を作成します。
- ^[3] -inc[obj] オプションにより、実行可能ファイル (sample) に加えて libsample.so のプロファイルを行うよう hiprof に指示します。hiprof コマンドはプログラムの計測機構付きバージョン (sample.hiprof) を作成します。gprof オプション (-numbers) が少なくとも 1 つ指定されているため、hiprof は計測機構付きプログラムを自動的に実行してプロファイル・データ・ファイル (sample.hiout) を作成し、gprof を実行してプロファイルを表示します。-numbers オプションにより、各プロシージャの開始行番号、ソース・ファイル名、

ライブラリ名が表示されます。これにより、同じ名前の静的プロシージャが複数ある場合にも識別が可能になります。

結果のサンプル・プロファイルを例 8-2 に示します。呼び出しグラフのプロファイルは、そのプロシージャが呼び出す他のプロシージャを含め、あるプロシージャの呼び出しにかかる総コストを概算します。この概算では、特定のプロシージャの呼び出しには毎回同じ時間がかかると仮定しています。これは、多くの場合に事実とは異なりますが、呼び出し元が単一の場合には正確な概算になります。

省略時の設定では、hiprof は低頻度のサンプリング手法を使用します。この手法では、選択されているすべてのライブラリを含め、プログラムが実行するすべてのコードのプロファイルを行うことができます。また、選択されているすべてのプロシージャ (スレッド関係のシステム・ライブラリを除く) の呼び出しグラフのプロファイルと、ソース行または機械語命令のレベルでの詳細なプロファイル (選択されている場合) も作成します。

例 8-2: gprof を使用した hiprof の省略時のプロファイル例

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
add_vector.c:
mul_by_scalar.c:
print_it.c:

% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm

% hiprof -numbers -L. -inc libsample.so sample
hiprof: info: instrumenting sample ...

hiprof: info: the LD_LIBRARY_PATH environment variable is not defined
hiprof: info: setting LD_LIBRARY_PATH=..[1]
hiprof: info: running instrumented program sample.hiprof ...

Value = 179804.149985

hiprof: info: instrumented program exited with status 0
hiprof: info: displaying profile for sample ...

gprof -b -scaled -all -numbers -L. sample.hiprof ./sample.hiout[2]

***** call-graph profile *****[3]

granularity: samples per 4 bytes; units: seconds*1e-3; total: 323e-3 seconds

index  %total   self  descendents      called /      total   parents
      called +      self   name      index
      called /      total   children

[1]    100.0         2      321         1 /         1   __start [2]
              2      321         1      main [1][4]
              318       0      100 /        100   add_vector [3][5]
              3       0         2 /         2   mul_by_scalar [4]
              0       0         1 /         1   print_it [5]
```

例 8-2: gprof を使用した hiprof の省略時のプロファイル例 (続き)

```
-----
[3]      98.5      318      0      100 /      100      main [1] [6]
      add_vector [3]
-----

[4]      0.9        3      0      2 /      2      main [1]
      mul_by_scalar [4]
-----

[5]      0.0        0      0      1 /      1      main [1]
      print_it [5]
-----

***** timing profile section *****
granularity: samples per 4 bytes; units: seconds*1e-3; total: 323e-3 seconds

%   cumulative   self      self/call total/call
total  units     units     calls  seconds  seconds  name
98.5   318        318        100    3184e-6   3184e-6   add_vector [3]
0.9    321        3         2     1465e-6   1465e-6   mul_by_scalar [4]
0.6    323        2         1     1953e-6   323242e-6 main [1]
0.0    323        0         1         0         0     print_it [5]

***** index section *****
Index by function name - 4 entries

[3] add_vector      : "add_vector.c":1
[1] main           : "profsample.c":12
[4] mul_by_scalar  : "mul_by_scalar.c":1
[5] print_it       : "print_it.c":4
```

- ❶ LD_LIBRARY_PATH 環境変数は、計測機構付きシェアード・ライブラリ libsample がある作業ディレクトリを指定するよう自動的に設定されます (8.6.4.1 項を参照)。
- ❷ 自動的に生成された gprof コマンド行は、省略時の設定により -scaled オプションを使用します。このオプションでは、表示対象として選択されたプロシージャの実行時間が短い場合に、プロファイルを CPU サイクルの精度とミリ秒単位で表示できます。
- ❸ gprof の出力は、呼び出しグラフのプロファイル、タイミングのプロファイル、およびインデックス (各プロシージャを識別する簡潔な手段) の 3 つのセクションからなります。この例では、3 つのセクションはアスタリスク (とセクション名) の行で区切られていますが、この行は

gprof が作成する出力にはありません。呼び出しグラフのプロファイルのセクションでは、プログラムの各ルーチンのサブセクションが破線で区切って示され、最初の欄のインデックス番号によって識別されます。

- ④ この行は main ルーチンを示します。このルーチンは、このセクションの右端の欄で最も左に表示されているので、このルーチンが、呼び出しグラフのこの部分の主体であることを示しています。左端の欄のインデックス番号 [1] は、出力の最後のインデックス・セクションのインデックス番号 [1] に対応します。2 番目の欄のパーセンテージは、main とその子孫 (この場合は add_vector, mul_by_scalar, および print_it) によって占められている、サブグラフ内の時間の合計を示します。called 欄の 1 は、main ルーチンが呼び出された回数の合計を示します。
- ⑤ この行は、add_vector と main の関係を示します。add_vector はこのセクションで main の下にあるので、main の子であることがわかります。分数 100/100 は、add_vector への合計 100 回の呼び出し (分母) のうち、100 回 (分子) が main からのものであることを示します。
- ⑥ この行は、main と add_vector の関係を示します。main は最後の欄で add_vector の上にあるので、add_vector の親であることがわかります。

スレッドのないプログラムでは、hiprof は、そのプログラムが使用したマシン・サイクルまたは発生したページ・フォールトの数をカウントすることもできます。それぞれの親プロシージャの呼び出しコストが正確に計算されるので、親のコストしか概算できない省略時のモードよりも、呼び出しグラフのプロファイルが格段に有用になります。また、計測機構付きルーチンの CPU 時間 (短いテストのためにはナノ秒単位) またはページ・フォールトのカウントの報告には、そのルーチンが呼び出す計測機構のないルーチンのカウントの報告が含まれます。こうしてコストの概算を行って、実行時のオーバーヘッドを減らすことができますが、計測機構付きプロシージャが少なくとも数秒ごとに呼び出されない場合には、マシン・サイクルのカウントがラップするため、注意が必要です。

次の例では、hiprof コマンドの -cycles オプションを使用して、sample プログラムが使用するマシン・サイクルのプロファイルを表示します。

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm
% hiprof -cycles -numbers -L. -inc libsample.so sample
```

結果のサンプル・プロファイルを例 8-3 に示します。

例 8-3: gprof を使用した hiprof の -cycles プロファイルの例

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
add_vector.c:
mul_by_scalar.c:
print_it.c:

% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm

% hiprof -cycles -numbers -L. -inc libsample.so sample
hiprof: info: instrumenting sample ...

hiprof: info: the LD_LIBRARY_PATH environment variable is not defined
hiprof: info: setting LD_LIBRARY_PATH=...
hiprof: info: running instrumented program sample.hiprof ...

Value = 179804.149985

hiprof: info: instrumented program exited with status 0
hiprof: info: displaying profile for sample ...

gprof -b -scaled -all -numbers -L. sample ./sample.hiout

granularity: cycles; units: seconds*1e-9; total: 362320292e-9 seconds

index  %total  self  descendents  called /  total  parents
              called +  self  name      index
              called /  total  children

[1]    100.0   893310 361426982      1      1  <spontaneous>
              361371860    1 /      1  __start [1]
                                   main [2]

-----

[2]      99.7  36316218 325055642      1      1  __start [1]
              321107275    100 /    100  main [2]
              3519530      2 /      2  add_vector [3]
              428838      1 /      1  mul_by_scalar [4]
                                   print_it [5]

-----

[3]      88.6  321107275      0    100 /    100  main [2]
                                   add_vector [3]

-----

[4]       1.0   3519530      0      2      2  main [2]
                                   mul_by_scalar [4]

-----

[5]      0.1   428838      0      1      1  main [2]
                                   print_it [5]

-----

granularity: cycles; units: seconds*1e-9; total: 362320292e-9 seconds

% cumulative      self      self/call total/call
```

8-10 プログラムのプロファイルによる性能の向上

例 8-3: gprof を使用した hiprof の -cycles プロファイルの例 (続き)

```
total      units      units      calls      seconds      seconds      name
88.6 321107275 321107275      100      3211e-6      3211e-6 add_vector [3]
10.0 357423492 36316218      1      36316e-6 361372e-6 main [2]
1.0 360943022 3519530      2      1760e-6      1760e-6 mul_by_scalar [4]
0.2 361836332 893310      1 893310e-9 362320e-6 __start [1]
0.1 362265170 428838      1 428838e-9 428838e-9 print_it [5]
```

Index by function name - 11 entries

```
[1] __start      <sample>
[3] add_vector   <libsampl.so>:"add_vector.c":1
[2] main         <sample>:"profsampl.c":12
[4] mul_by_scalar <libsampl.so>:"mul_by_scalar.c":1
[5] print_it     <libsampl.so>:"print_it.c":4
```

8.3.2.1.2 cc コマンドの -pg オプションを使用する方法

cc コマンドの -pg オプションは hiprof と同じサンプリング手法を使用しますが、プログラムは -pg オプションでコンパイルすることにより、計測機構が付加されている必要があります (プログラムのコンパイルには、8.2 節で説明したデバッグおよび最適化のオプションも必要です)。実行可能ファイルのみのプロファイルが行われ (シェアード・ライブラリは除外される)、呼び出しグラフのプロファイルの生成のためにコンパイルされるシステム・ライブラリはほとんどありません。したがって、hiprof を使用する方がよいかもしれません。ただし、cc コマンドの -pg オプションと gprof は他のベンダの UNIX システムでも良く似た方法でサポートされているので、これが利点となる場合もあります。たとえば、次のようにコマンドを実行します。

```
% cc -pg -o sample -g1 -O2 *.c -lm1
% ./sample2
% gprof -scaled -b -numbers sample3
```

- ¹ cc コマンドの呼び出しグラフのプロファイル・オプション -pg は、プログラムの計測機構付きバージョン sample を作成します (コンパイルとリンクの両方のフェーズで -pg オプションを指定する必要があります)。
- ² 計測機構付きバージョンのプログラムを実行すると、gprof ツールが使用するプロファイル・データ・ファイル (省略時の名前は gmon.out) が作成されます。複数のデータ・ファイルで作業する場合についての情報は、8.7 節を参照してください。
- ³ gprof コマンドは、プロファイル・データ・ファイルから情報を取り出して表示します。-scaled オプションは、レポートの欄幅を超え

ることなく，最高の精度が得られる単位で CPU 時間を表示します。
-b オプションは，プロファイル内の各フォールドの説明が表示されないようにします。

結果のサンプル・プロファイルを例 8-4 に示します。gprof ツールは，呼び出しグラフのプロファイルでプロシージャ (そのプロシージャが呼び出す他のプロシージャも含む) を呼び出す際にかかる総コストを概算します。

例 8-4: gprof を使用した cc -pg プロファイルの例

```
% cc -pg -o sample -g1 -O2 add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:

% ./sample
Value = 179804.149985

% gprof -scaled -b -numbers sample

granularity: samples per 8 bytes; units: seconds*1e-3; total: 326e-3 seconds

index  %total    self  descendents      called /
              self  called +
              total  called /
              total  parents
              name  index
              children

[1]    100.0          5      321          1 /
              5      321          1      1  __start [2]
              317          0      100 /      main [1]
              4          0          2 /      add_vector [3]
              0          0          1 /          2  mul_by_scalar [4]
              1          1          1  print_it [5]

-----

[3]    97.3          317          0      100 /      100  main [1]
              317          0      100      add_vector [3]

-----

[4]    1.2          4          0          2 /          2  main [1]
              4          0          2      mul_by_scalar [4]

-----

[5]    0.0          0          0          1 /          1  main [1]
              0          0          1      print_it [5]

-----

granularity: samples per 8 bytes; units: seconds*1e-3; total: 326e-3 seconds

% cumulative    self      self/call total/call
total  units  units  calls  seconds  seconds  name
97.3    317    317    100    3174e-6    3174e-6  add_vector [3]
1.5     322      5      1     4883e-6    326172e-6  main [1]
1.2     326      4      2    1953e-6    1953e-6  mul_by_scalar [4]
0.0     326      0      1          0          0  print_it [5]
```

8-12 プログラムのプロファイルによる性能の向上

例 8-4: gprof を使用した cc -pg プロファイルの例 (続き)

Index by function name - 4 entries

[3] add_vector	<sample>:"add_vector.c":1
[1] main	<sample>:"profsample.c":12
[4] mul_by_scalar	<sample>:"mul_by_scalar.c":1
[5] print_it	<sample>:"print_it.c":4

8.3.2.2 ソース行または命令の、CPU 時間またはイベントのプロファイル

性能の向上を図るための優れた方法は、プロシージャ・レベルのプロファイルをプログラム全体で行うことから始まります (全体像を得るために呼び出しグラフを付けることもあります) が、個別のソース行および命令の詳細なプロファイルに発展することも多くあります。この情報を得るには、次のツールを使用します。

- uprofile プロファイラ (8.3.2.2.1 項)
- hiprof プロファイラ (8.3.2.2.2 項)
- cc コマンドの -p オプション (8.3.2.2.3 項)
- pixie プロファイラ (8.3.2.2.4 項)

8.3.2.2.1 uprofile プロファイラを使用する方法

uprofile プロファイラ (uprofile(1) を参照) は、サンプリング手法を使用して、各プロシージャ、ソース行、または命令に関連する CPU 時間またはイベント (キャッシュ・ミスなど) のプロファイルを生成します。サンプリングの頻度は、プロセッサのタイプとサンプリングする統計情報によって異なりますが、CPU 時間でミリ秒のオーダーです。このプロファイラは、アプリケーション・プログラムの修正を全く行うことなく、Alpha CPU に組み込まれているハードウェア・カウンタを使用してこの作業を行います。uprofile コマンドを引数なしで実行すると、マシンのアーキテクチャ特性に基づいて、特定のマシンでプロファイルを行うことができるすべての種類のイベントのリストが作成されます。省略時の設定では、マシン・サイクルをプロファイルし、その結果、CPU 時間のプロファイルが作成されます。次の例では、CPU 時間の上位 90% を使用する命令のプロファイルを表示する方法を示します。

```
% cc -o sample -g1 -O2 *.c -lm 1
% uprofile -asm -quit 90cum% sample 2
```

結果のサンプル・プロファイルと説明文を例 8-5 に示します。

- ❶ -g1 オプションと -O2 オプションの詳細については、8.2 節を参照してください。
- ❷ uprofile コマンドは sample プログラムを実行し、性能カウンタのデータをプロファイル・データ・ファイル (省略時の名前は umon.out) に収集します。prof のオプション (-asm と -quit) が指定されているので、uprofile は次に自動的に prof ツールを実行して、プロファイルを表示します。

-asm オプションは、命令ごとにサイクル (および、指定されている場合は、データ・キャッシュ・ミスなど、その他の CPU 統計情報) のプロファイルを行います。ここではカウンタ統計情報は指定されていないので、uprofile は各命令の CPU 時間プロファイルを表示します。
-quit 90cum% オプションは、プロファイルの全体の 90% が表示されると、その後の表示を切り捨てます (8.6.3 項を参照)。-heavy オプションも使用できます。このオプションは、最も多くの命令を実行した行を報告します。また、各プロシージャ内のソース行ごとにプロファイルを報告する -lines オプションも使用できます (8.6.2 項を参照)。

例 8-5: prof を使用した uprofile の CPU 時間プロファイルの例

```
% cc -o sample -g1 -O2 add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:

% uprofile -asm -quit 90cum% sample
Value = 179804.149985
Writing umon.out

Displaying profile for sample:

Profile listing generated Thu Dec  3 10:29:25 1998 with:
  prof -asm -quit 90cum% sample umon.out

-----
* -a[sm] using performance counters:                                     *
*   cycles0: 1 sample every 65536 Cycles (0.000164 seconds)             *
* sorted in descending order by total time spent in each procedure;      *
* unexecuted procedures excluded                                         *
-----

Each sample covers 4.00 byte(s) for 0.052% of 0.3123 seconds

      millisec      % cum %      address:line  instruction
add_vector (add_vector.c)
      0.0      0.00      0.00  0x120001260:2      addl      zero, a2, a2
```

8-14 プログラムのプロファイルによる性能の向上

例 8-5: prof を使用した uprofile の CPU 時間プロファイルの例 (続き)

```

0.0 0.00 0.00 0x120001264:5 bis zero, zero, t0
0.0 0.00 0.00 0x120001268:5 ble a2, 0x12000131c
0.0 0.00 0.00 0x12000126c:5 subl a2, 0x3, t1
0.0 0.00 0.00 0x120001270:5 cmple t1, a2, t2
0.0 0.00 0.00 0x120001274:5 beq t2, 0x1200012f4
0.0 0.00 0.00 0x120001278:5 ble t1, 0x1200012f4
0.0 0.00 0.00 0x12000127c:5 subq a0, a1, t3
0.0 0.00 0.00 0x120001280:5 lda t3, 31(t3)
0.0 0.00 0.00 0x120001284:5 cmpule t3, 0x3e, t3
0.0 0.00 0.00 0x120001288:5 bne t3, 0x1200012f4
0.0 0.00 0.00 0x12000128c:5 ldq_u zero, 0(sp)
20.2 6.45 6.45 0x120001290:6 ldl zero, 128(a1)
1.3 0.42 6.87 0x120001294:6 lds $f31, 128(a0)
35.6 11.39 18.26 0x120001298:6 ldt $f0, 0(a1)
20.0 6.40 24.66 0x12000129c:6 ldt $f1, 0(a0)
9.7 3.10 27.75 0x1200012a0:6 ldt $f10, 8(a1)
14.9 4.77 32.53 0x1200012a4:6 ldt $f11, 8(a0)
17.4 5.56 38.09 0x1200012a8:6 ldt $f12, 16(a1)
7.0 2.26 40.35 0x1200012ac:6 ldt $f13, 16(a0)
8.2 2.62 42.97 0x1200012b0:6 ldt $f14, 24(a1)
12.9 4.14 47.11 0x1200012b4:6 ldt $f15, 24(a0)
24.9 7.97 55.09 0x1200012b8:6 addt $f1,$f0,$f0
24.7 7.92 63.01 0x1200012bc:6 addt $f11,$f10,$f10
37.8 12.12 75.13 0x1200012c0:6 addt $f13,$f12,$f12
39.2 12.54 87.67 0x1200012c4:6 addt $f15,$f14,$f14
0.8 0.26 87.93 0x1200012c8:5 addl t0, 0x4, t0
0.0 0.00 87.93 0x1200012cc:5 lda a1, 32(a1)
8.4 2.68 90.61 0x1200012d0:6 stt $f0, 0(a0)

```

比較として、次の例に、EV56 Alpha システムでのデータ・キャッシュ・ミスの上位 90% が発生した命令のプロファイルを表示する方法を示します。

```
% cc -o sample -g1 -O2 *.c -lm
% uprofile -asm -quit 90cum% dcacheldmisses sample
```

結果のサンプル・プロファイルを例 8-6 に示します。

例 8-6: prof を使用した uprofile のデータ・キャッシュ・ミス・プロファイルの例

```
% uprofile -asm -quit 90cum% dcacheldmisses sample
Value = 179804.149985
Writing umon.out
```

Displaying profile for sample:

```
Profile listing generated Thu Dec 3 10:34:25 1998 with:
prof -asm -quit 90cum% sample umon.out
```

```

-----
* -a[sm] using performance counters: *
* dcacheldmiss: 1 sample every 16384 DCache LD Misses[1] *
* sorted in descending order by samples recorded for each procedure; *
* unexecuted procedures excluded *

```

例 8-6: prof を使用した uprofile のデータ・キャッシュ・ミス・プロファイルの例 (続き)

Each sample covers 4.00 byte(s) for 0.18% of 550 samples^[2]

samples	%	cum %	address:line	instruction
add_vector (add_vector.c)				
0.0	0.00	0.00	0x120001260:2	addl zero, a2, a2
0.0	0.00	0.00	0x120001264:5	bis zero, zero, t0
0.0	0.00	0.00	0x120001268:5	ble a2, 0x12000131c
0.0	0.00	0.00	0x12000126c:5	subl a2, 0x3, t1
0.0	0.00	0.00	0x120001270:5	cmple t1, a2, t2
0.0	0.00	0.00	0x120001274:5	beq t2, 0x1200012f4
0.0	0.00	0.00	0x120001278:5	ble t1, 0x1200012f4
0.0	0.00	0.00	0x12000127c:5	subq a0, a1, t3
0.0	0.00	0.00	0x120001280:5	lda t3, 31(t3)
0.0	0.00	0.00	0x120001284:5	cmpule t3, 0x3e, t3
0.0	0.00	0.00	0x120001288:5	bne t3, 0x1200012f4
0.0	0.00	0.00	0x12000128c:5	ldq_u zero, 0(sp)
1.0	0.18	0.18	0x120001290:6	ldl zero, 128(a1)
3.0	0.55	0.73	0x120001294:6	lds \$f31, 128(a0)
62.0	11.27	12.00	0x120001298:6	ldt \$f0, 0(a1) ^[3]
64.0	11.64	23.64	0x12000129c:6	ldt \$f1, 0(a0)
8.0	1.45	25.09	0x1200012a0:6	ldt \$f10, 8(a1)
47.0	8.55	33.64	0x1200012a4:6	ldt \$f11, 8(a0)
13.0	2.36	36.00	0x1200012a8:6	ldt \$f12, 16(a1)
38.0	6.91	42.91	0x1200012ac:6	ldt \$f13, 16(a0)
15.0	2.73	45.64	0x1200012b0:6	ldt \$f14, 24(a1)
51.0	9.27	54.91	0x1200012b4:6	ldt \$f15, 24(a0)
49.0	8.91	63.82	0x1200012b8:6	addt \$f1, \$f0, \$f0
142.0	25.82	89.64	0x1200012bc:6	addt \$f11, \$f10, \$f10
13.0	2.36	92.00	0x1200012c0:6	addt \$f13, \$f12, \$f12

- ^[1] "1 sample every 16384" というサンプリング・レートは、プロファイルに示されるサンプルがそれぞれ 16384 回のデータ・キャッシュ・ミスの後に発生したことを意味しますが、その命令で発生したすべてのサンプルが示されるわけではありません。
- ^[2] データ・キャッシュ・ミスの数ではなく、サンプルの総数が示されます。
- ^[3] 1 つの命令に記録されたサンプルの数を示します。これは、統計的にデータ・キャッシュ・ミスの比率を暗示します。

uprofile プロファイル手法には、実行時のオーバーヘッドが非常に小さいという長所があります。また、個々の命令またはソース行を実行する手間がかかりますが、これにより得られる詳細情報は、プロシージャ内のどのオペレーションがプログラムの実行速度を遅くしているのかを正確に識別するために不可欠です。

uprofile には、次の短所があります。

- 実行可能ファイル以外はプロファイルできない。
ライブラリ内のコードのプロファイルを行うには、まず、`-non_shared` オプションを使用してプログラムをリンクする必要があります。
- ハードウェア・カウンタでは同時に複数のプログラムのプロファイルを行うことができない。
- スレッドのプロファイルを個別に行うことができない。
- Alpha EV6 アーキテクチャでは、命令がシーケンスどおりの順序で実行されないため、詳細なプロファイルの精度が大幅に下がる。

8.3.2.2.2 hiprof プロファイラを使用する方法

すでに説明したとおり、hiprof コマンドの省略時の PC サンプリング手法でも、1 ミリ秒弱のサンプリング頻度で、uprofile と同様に CPU 時間プロファイルを生成できます。呼び出しのカウンタが性能に影響するため、このプロファイルはあまり正確にはなりません、次のような長所があります。

- シェアード・ライブラリのプロファイルを行うことができる。
- スレッドのプロファイルを個別に行うことができる (大量のメモリと大きなファイル・サイズを使用)。
- ハードウェアのリソースとアーキテクチャに依存しない。

次の例では、hiprof コマンドの `-lines` オプションを使用して、各ソース行が使用した CPU 時間のプロファイルをプロシージャごとに分けて表示します。

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm
% hiprof -lines -numbers -L. -inc libsample.so sample
```

結果のサンプル・プロファイルを例 8-7 に示します。

例 8-7: hiprof -lines による PC サンプリング・プロファイルの例

```
% cc -o libsample.so -shared -g1 -O2 add_vector.c mul_by_scalar.c print_it.c
add_vector.c:
mul_by_scalar.c:
print_it.c:

% cc -o sample -g1 -O2 profsample.c -L. -lsample -lm

% hiprof -lines -numbers -L. -inc libsample.so sample
hiprof: info: instrumenting sample ...
```

例 8-7: hiprof -lines による PC サンプリング・プロファイルの例 (続き)

```
hiprof: info: the LD_LIBRARY_PATH environment variable is not defined
hiprof: info: setting LD_LIBRARY_PATH=...
hiprof: info: running instrumented program sample.hiprof ...

Value = 179804.149985

hiprof: info: instrumented program exited with status 0
hiprof: info: displaying profile for sample ...

gprof -b -scaled -all -lines -numbers -L. sample.hiprof ./sample.hiout

Milliseconds per source line, in source order within functions

procedure (file)                line bytes   millisec      %  cum %
add_vector (add_vector.c)        2    52        0.0    0.00   0.00
                                5    92        1.0    0.30   0.30
                                6    92       318.4  98.19  98.49
                                8     4         0.0    0.00  98.49
mul_by_scalar (mul_by_scalar.c)  2    52         0.0    0.00  98.49
                                5    72         0.0    0.00  98.49
                                6    64         3.9    1.20  99.70
                                8     4         0.0    0.00  99.70
main (profsample.c)              9    32         0.0    0.00  99.70
                                12   128         0.0    0.00  99.70
                                16     8         0.0    0.00  99.70
                                19    32         0.0    0.00  99.70
                                20    36         0.0    0.00  99.70
                                21    16         0.0    0.00  99.70
                                22    24         0.0    0.00  99.70
                                24     4         0.0    0.00  99.70
                                25    36         0.0    0.00  99.70
                                26    16         0.0    0.00  99.70
                                27    28         1.0    0.30 100.00
                                28    20         0.0    0.00 100.00
                                29    40         0.0    0.00 100.00
                                30     4         0.0    0.00 100.00
                                31    20         0.0    0.00 100.00
                                32     4         0.0    0.00 100.00
                                33    20         0.0    0.00 100.00
                                34    56         0.0    0.00 100.00
```

8.3.2.2.3 cc コマンドの -p オプションを使用する方法

cc コマンドの -p オプションは、低頻度のサンプリング手法を使用して CPU 時間のプロファイルを生成します。このプロファイルは uprofile の場合と似ていますが、統計的に精度が低くなります。しかし、-p オプションには次の長所があります。

- シェアード・ライブラリのプロファイルを行うことができる。
- スレッドのプロファイルを個別に行うことができる。

8-18 プログラムのプロファイルによる性能の向上

- ハードウェアのリソースとアーキテクチャに依存しない。
- 多くの UNIX オペレーティング・システムに共通である。
- Tru64 UNIX では、1 つのプログラムが使用するすべてのシェアード・ライブラリのプロファイルを行うことができる。

`-p` オプションを使用してプログラムを再リンクする必要がありますが、`cc` コマンドの `-g1` オプションを使用した場合など、元のコンパイルがある程度のデバッグ・レベルを使用して行われている場合は、ソースからコンパイルし直す必要はありません (8.2 節を参照)。たとえば、プログラムの個々のソース行とプロシージャのプロファイルを行うには、次のようにコマンドを実行します (密度の高いサンプル配列を作成するのに十分な時間プログラムが実行される場合)。

```
% cc -p -o sample -g1 -O2 *.c -lm 1
% setenv PROFFLAGS '-all -stride 1' 2
% ./sample 3
% prof -all -proc -heavy -numbers sample 4
```

- 1 `cc` コマンドの PC サンプル・プロファイル・オプション `-p` は、計測機構付きバージョンのプログラム `sample` を作成します。
 - 2 `PROFFLAGS` 環境変数で指定された `-all` オプションにより、すべてのシェアード・ライブラリのプロファイルが要求されます (8.6.4 項を参照)。その結果、CPU 時間の第 2 位の使用者として `sqrt (libm.so から)` がプロファイル内に表示されます。変数は、計測機構付きバージョンのプログラムを実行する前にセットする必要があります。
- `prof` の `-heavy` オプションを使用したソース行ごとのプロファイルを正確にするため、`PROFFLAGS` の `-stride 1` オプションにより、各命令に別々のカウンタを使用するよう要求します。
- 3 計測機構付きバージョンのプログラムを実行すると、`prof` ツールが使用する PC サンプリング・データ・ファイル (省略時の名前は `mon.out`) が作成されます。複数のデータ・ファイルを使用した作業については、8.7 節を参照してください。
 - 4 `prof` ツール (`prof(1)`を参照) は PC サンプリング・データ・ファイルを使用してプロファイルを作成します。この手法はプログラム・カウンタの周期的なサンプリングによって機能するため、同じプログラムを複数回プロファイルすると、出力が異なる場合があります。

この例のように `prof` を手動で実行する場合は、プロファイルの表示に含めるシェアード・ライブラリを指定することができます。 `-all` オプションを指定すると、`prof` はすべてのライブラリを表示します (8.6.4 項を参照)。 `-proc[edures]` オプションを指定すると、各プロセスで実行された命令とプロセスへの呼び出しがプロファイルされます。 `-heavy` オプションを指定すると、最も多くの命令を実行した行が報告されます。 また、 `-lines` を指定すると行ごとのプロファイルが、 `-asm` を指定すると命令ごとのプロファイルが、どちらもプロセスごとに分けて表示されます。

結果のサンプル・プロファイルを例 8-8 に示します。

例 8-8: `prof` を使用した `cc -p` プロファイルの例

```
% cc -p -o sample -g1 -O2 add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:

% setenv PROFLFLAGS "-all -stride 1"

% ./sample
Value = 179804.149985

% prof -all -proc -heavy -numbers sample
Profile listing generated Mon Feb 23 15:33:07 1998 with:
  prof -all -proc -heavy -numbers sample

-----
* -p[rocedures] using pc-sampling;                                     *
* sorted in descending order by total time spent in each procedure;    *
* unexecuted procedures excluded                                         *
-----

Each sample covers 4.00 byte(s) for 0.25% of 0.3955 seconds

%time      seconds  cum %   cum sec  procedure (file)
-----
93.1       0.3682   93.1    0.37    add_vector (<sample>:"add_vector.c":1)
5.4        0.0215   98.5    0.39    sqrt (<libm.so>)
1.0        0.0039   99.5    0.39    mul_by_scalar (<sample>:"mul_by_scalar.c":1)
0.5        0.0020  100.0    0.40    main (<sample>:"profsample.c":12)

-----
* -h[eavy] using pc-sampling;                                           *
* sorted in descending order by time spent in each source line;        *
* unexecuted lines excluded                                             *
-----

Each sample covers 4.00 byte(s) for 0.25% of 0.3955 seconds

procedure (file)                                line bytes  millisec    % cum %
-----
add_vector (add_vector.c)                        6    80      363.3  91.85  91.85
```

8-20 プログラムのプロファイルによる性能の向上

例 8-8: prof を使用した cc -p プロファイルの例 (続き)

add_vector (add_vector.c)	5	96	4.9	1.23	93.09
mul_by_scalar (mul_by_scalar.c)	6	60	3.9	0.99	94.07
main (profsample.c)	20	36	2.0	0.49	94.57

8.3.2.2.4 pixie プロファイラを使用する方法

pixie ツール (pixie(1) を参照) でも、ソース行と命令 (シェアード・ライブラリを含む) のプロファイルを行うことができますが、cycles のカウントの表示はマシン・サイクルではなく、実行された命令のカウントの報告であるため、注意が必要です。-truecycles 2 オプションにより、キャッシュによるメモリ・アクセスがすべて満足された場合に使用されるサイクルの数を概算することができます。ただし、この概算を正確なものにするのに十分なだけプログラムがデータをキャッシュできる場合はほとんどなく、また、この方法で完全にシミュレートできるのは Alpha EV4 および EV5 ファミリだけです。たとえば、次のようにコマンドを実行します。

```
% cc -o sample -g1 -O2 *.c -lm[1]
% pixie -all -proc -heavy -quit 5 sample[2]
```

^[1] -g1 オプションと -O2 オプションについては、8.2 節を参照してください。

^[2] pixie コマンドは、計測機構付きバージョンのプログラム (sample.pixie) と命令アドレス・ファイル (sample.Addrs) を作成します。prof のオプション (-proc, -heavy, -quit) が指定されているので、pixie は計測機構付きプログラムを自動的に実行して命令カウント・ファイル (sample.Counts) を作成し、次に .Addrs ファイルと .Counts ファイルを入力として使用して prof を実行し、プロファイルを表示します。

-all オプションにより、シェアード・ライブラリのプロファイルが要求されます。pixie プロファイルにはシェアード・ライブラリを含めることができますが、システム・ライブラリ (sqrt 関数が含まれる libm.so など) にはソース行番号がないので、-heavy オプションの行ごとのプロファイルには含まれません。また、静的プロシージャには proc_at... という名前が作成されて与えられます。

-heavy オプションにより、最も多くの命令を実行した行が報告されます。-proc[edures] オプションにより、各プロシージャで実行された命令とプロシージャへの呼び出しのプロファイルが行われます。-quit 5 オプションにより、各モード (-heavy, -proc[edures]) で5行以降の報告が切り捨てられます。

結果のサンプル・プロファイルを例 8-9 に示します。

例 8-9: prof を使用した pixie プロファイルの例

```
% cc -o sample -g1 -O2 add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:

% pixie -all -proc -heavy -quit 5 sample
pixie: info: instrumenting sample ...

pixie: info: the LD_LIBRARY_PATH environment variable is not defined
pixie: info: setting LD_LIBRARY_PATH=.
pixie: info: running instrumented program sample.pixie ...

Value = 179804.149985

pixie: info: instrumented program exited with status 0
pixie: info: displaying profile for sample ...

Profile listing generated Mon Feb 23 15:33:55 1998 with:
  prof -pixie -all -procedures -heavy -quit 5 sample ./sample.Counts

-----
* -p[rocedures] using basic-block counts;                                     *
* sorted in descending order by the number of cycles executed in each       *
* procedure; unexecuted procedures are excluded                             *
-----

69089823 cycles (0.1727 seconds at 400.00 megahertz)

      cycles %cycles  cum %   seconds      cycles  bytes procedure (file)
              /call   /line
60001400    86.85   86.85   0.1500    600014    48 add_vector (add_vector.c)
7100008     10.28   97.12   0.0178        72      ? sqrt (<libm.so>)
1301816      1.88   99.01   0.0033   1301816    27 main (profsample.c)
675020       0.98   99.98   0.0017   337510    36 mul_by_scalar (mul_by_scalar.c)
      854       0.00   99.98   0.0000      854      ? __cvtas_t_to_a (<libc.so>)

-----
* -p[rocedures] using invocation counts;                                     *
* sorted in descending order by number of calls per procedure;               *
* unexecuted procedures are excluded                                         *
-----

100504 invocations total

      calls  %calls   cum%   bytes procedure (file)
100000     99.50   99.50    820 sqrt (<libm.so>)
```

8-22 プログラムのプロファイルによる性能の向上

例 8-9: prof を使用した pixie プロファイルの例 (続き)

```
100 0.10 99.60 192 add_vector (add_vector.c)
39 0.04 99.64 164 proc_at_0x3ff815bc1e0 (<libc.so>)
38 0.04 99.67 144 proc_at_0x3ff815bc150 (<libc.so>)
38 0.04 99.71 16 proc_at_0x3ff815bc140 (<libc.so>)

-----
* -h[heavy] using basic-block counts; *
* sorted in descending order by the number of cycles executed in each *
* line; unexecuted lines are excluded *
-----

procedure (file)                line bytes    cycles      % cum %
add_vector (add_vector.c)        6    92    45000000    65.13  65.13
add_vector (add_vector.c)        5    92    15001200    21.71  86.85
main (profsample.c)             20    36     600003    0.87  87.71
main (profsample.c)             22    24     600000    0.87  88.58
mul_by_scalar (mul_by_scalar.c)  6    64     487500    0.71  89.29
```

-procedures プロファイルには、呼び出しカウントのプロファイルが含まれます。

オプションとして dxprof コマンドを使用すると、pixie または cc コマンドの -p オプションにより収集されたプロファイルがグラフィカル表示されます。

実行可能ファイル sample について prof -pixstats コマンドを実行して、演算コードの頻度、インターロック、ミニプロファイルなどの詳細な報告を生成することもできます。詳細については、prof(1) を参照してください。

8.4 システム・リソース使用の最小化

この節では、アプリケーションによるシステム・リソースの使用を最小限に抑えるための手法とツールについて説明します。

8.4.1 手法

ここまでに説明した手法では、アプリケーションによる CPU の使用状況だけを改善できます。コンピュータ・システムのその他のコンポーネント (ヒープ・メモリ、ディスク・ファイル、ネットワーク接続など) のアプリケーションによる使用状況を効率化することにより、さらに性能を向上できます。

リソース使用状況を改善するための処理の最初のフェーズは、CPU のプロファイルと同様に、メモリの使用量、データ I/O、ディスク・スペース、経

過時間などを監視することです。次に、コンピュータのスループットを増加または調整して、コンピュータ・リソースの使用状況を改善できるようプログラムあるいはプログラムの設計を調整します。たとえば、次のような調整を行います。

- プログラムが読み取りまたは書き込みを行うデータ・ファイルのサイズを小さくする。
- 通常の I/O ではなくメモリ・マップ・ファイルを使用する。
- 要求される可能性のある最大量のメモリを起動時に割り当てるのではなく、要求された時点で増分的にメモリを割り当てる。
- ヒープ・リークを修正し、割り当てられたメモリが未使用のままにならないようにする。

Tru64 UNIX システムの解析とチューニングについての広範な説明は、『システムの構成とチューニング』を参照してください。

8.4.2 ツールと例

ここでは、システム・モニタと Third Degree ツールを使用して、システム・リソースの使用を最小限に抑える方法を説明します。

8.4.2.1 システム・モニタ

Tru64 UNIX ベースのシステム・コマンド `ps u`, `swapon -s`, および `vmstat 3` を使用すると、現在のアクティブ・プロセスのシステム・リソース (CPU 時間、物理メモリと仮想メモリ、スワップ領域、ページ・フォールトなど) の使用状況を表示できます。詳細については、`ps(1)`, `swapon(8)`, および `vmstat(3)` を参照してください。

オプションとして `pview` コマンドを使用すると、アプリケーションを構成するプロセスについての同様の情報がグラフィカル表示されます。 `pview(1)` を参照してください。

Tru64 UNIX システムの `time` コマンドとコマンド・シェルを使用すると、プログラムとその子孫の合計の経過時間と CPU 時間を簡単に測定できます。 `time(1)` を参照してください。

Performance Manager は、システム性能の監視と管理を行うためのオプションのツールで、グラフィカル・インタフェースを備えています。 `pmgr(8X)` を参照してください。

関連ツールの詳細については、『システムの構成とチューニング』を参照してください。

8.4.2.2 ヒープ・メモリの解析

Third Degree ツール (third(1) を参照) は、Third Degree のメモリ使用チェッカでプログラムを計測し、それを実行して、検出したリークのログをプログラムの終了時に表示することにより、プログラムにおけるヒープ・メモリのリークを報告します。たとえば、次のようにコマンドを実行します。

```
% cc -o sample -g -non_shared *.c -lm[1]
% third -display sample[2]
```

- ^[1] Thrid Degree を使用する場合には、通常は完全なデバッグ情報 (-g オプションでコンパイル) を利用できることが望ましいのですが、完全な情報が利用できない場合、レポートはソース・レベルではなく機械語レベルになります。メモリ・リークをチェックするだけの場合は、-g1 オプションが適しています。
- ^[2] third コマンドは、プログラムの計測機構付きバージョン (sample.third) を作成します。-display オプションが指定されているため、third は作成した計測機構付きプログラムを自動的に実行してログ・ファイル (sample.3log) を作成し、次に more を実行してログ・ファイルを表示します。

結果のサンプル・ログ・ファイルを例 8-10 に示します。

例 8-10: third ログ・ファイルの例

```
cc -o sample -g -non_shared add_vector.c mul_by_scalar.c print_it.c profsample.c -lm
add_vector.c:
mul_by_scalar.c:
print_it.c:
profsample.c:
third -display sample
third: info: instrumenting sample ...

third: info: running instrumented program sample.third ...

Value = 179804.149985

third: info: instrumented program exited with status 0
third: info: error log for sample ...

more ./sample.3log
Third Degree version 5.0
sample.third run by jpx on frumpy.abc.dec.com at Mon Jun 21 16:59:50 1999

//////////////////////////////// Options //////////////////////////////////
-----
```

例 8-10: third ログ・ファイルの例 (続き)

```
-----
-----
-----
New blocks in heap after program exit

Leaks - blocks not yet deallocated but apparently not in use:
* A leak is not referenced by static memory, active stack frames,
  or leaked blocks, though it may be referenced by other leaks.
* A leak "not referenced by other leaks" may be the root of a leaked tree.
* A block referenced only by registers, unseen thread stacks, mapped memory,
  or uninstrumented library data is falsely reported as a leak. Instrumenting
  shared libraries, if any, may reduce the number of such cases.
* Any new leak lost its last reference since the previous heap report, if any.

A total of 800000 bytes in 1 leak were found:

800000 bytes in 1 leak (including 1 not referenced by other leaks) created at:
    malloc                sample
    main                  sample, profsample.c, line 19
    __start               sample

Objects - blocks not yet deallocated and apparently still in use:
* An object is referenced by static memory, active stack, or other objects.
* A leaked block may be falsely reported as an object if a pointer to it
  remains when a new stack frame or heap block reuses the pointer's memory.
  Using the option to report uninitialized stack and heap may avoid such cases.
* Any new object was allocated since the previous heap report, if any.

A total of 0 bytes in 0 objects were found:

-----
-----
memory layout at program exit
    heap      933888 bytes [0x140012000-0x1400f6000]
    stack     819504 bytes [0x11ff37ed0-0x120000000]
    sample data 66464 bytes [0x140000000-0x1400103a0]
    sample text 802816 bytes [0x120000000-0x1200c4000]

=====
```

省略時の設定では、Third Degree はメモリ・リークのプロファイルを行い、オーバーヘッドはほとんどありません。

プログラムの起動時やシャットダウン時ではなく、通常の動作中に発生したリークのみを調べたい場合は、以前に報告されなかったリークを調べるための場所を追加指定できます。たとえば、次のようにしてシャットダウン前のリーク報告にこの情報を含めます。

```
% third -display -after startup -before shutdown sample
```

Thrid Degree ではまた、プログラムの正確さや性能に影響を及ぼすさまざまな種類のバグを検出できます。デバッグとリーク検出の詳細については、第 7 章を参照してください。

オプションとして `dxheap` コマンドを使用すると、Third Degree のヒープとバグのレポートがグラフィカル表示されます。`dxheap(1)` を参照してください。

オプションとして `mview` コマンドを使用すると、時間の経過に伴うヒープの使用状況の解析がグラフィカル表示されます。こうしてプログラムのヒープを表示すると、大量のリークやメモリの浪費などの望ましくない傾向の存在を (原因ではないとしても) はっきりと示すことができます。`mview(1)` を参照してください。

8.5 テスト・ケースの重要性の確認

この節では、テスト・ケースの重要性を確認するための手法とツールについて説明します。

8.5.1 手法

これまでに説明したプロファイル手法のほとんどは、性能が重視される場面で実行されるプログラムの一部について、プロファイルと最適化または調整を部分的に行う場合にのみ効果があります。テスト実行のプロファイルでは、ほとんどの場合、使用するデータを慎重に選択すれば十分ですが、1 組のテストで実行されたコードと実行されなかったコードを定量的に解析したい場合もあります。

8.5.2 ツールと例題

`pixie` コマンドの `-t[estcoverage]` オプションは、テストで実行されなかったコード行を報告します。たとえば、次のようにコマンドを実行します。

```
% cc -o sample -g1 -O2 *.c -lm
% pixie -t sample
```

同様に、`-zero` オプションは、一度も実行されなかったプロシージャの名前を報告します。逆に、`pixie` の `-p[rocedure]`、`-h[eavy]`、および `-a[sm]` オプションは、実行されたプロシージャ、ソース行、および命令を報告します。

典型的なシナリオを作成するためにテストを複数回実行する必要がある場合は、1 組のプロファイル・データ・ファイルに `prof` コマンドを単独で実行します。たとえば、次のようにします。

```
% cc -o sample -g1 -O2 *.c -lm 1
% pixie -pids sample 2
% ./sample.pixie 3
% ./sample.pixie
% prof -pixie -t sample sample.Counts.* 4
```

- 1 `-g1` オプションと `-O2` オプションの詳細については、8.2 節を参照してください。
- 2 `pixie` コマンドはプログラムの計測機構付きバージョン (`sample.pixie`) と命令アドレス・ファイル (`sample.Addrs`) を作成します。 `-pids` オプションは、作成したプロファイル・データ・ファイルの名前に、計測機構付きバージョンのプログラムのテスト実行のプロセス ID (item 3) を追加して、一回の実行ごとに別々のファイルが残るようにします。複数のデータ・ファイルを使用した作業の詳細については、8.7 節を参照してください。
- 3 計測機構付きプログラムを 2 回実行 (通常は異なる入力データを使用) して、`sample.Counts.pid` という名前の 2 つのプロファイル・データ・ファイルを作成します。
- 4 `-pixie` オプションは、省略時の PC サンプリング・モードではなく `pixie` モードを使用するよう `prof` に指示します。 `prof` ツールは、`sample.Addrs` ファイルと 2 つの `sample.Counts.pid` ファイルを使用して、2 回の実行結果からプロファイルを作成します。

8.6 表示するプロファイル情報の選択

アプリケーションのサイズおよび指定したプロファイルのタイプによっては、プロファイラが生成する出力の量が非常に大きくなる場合があります。しかし、アプリケーションの特定の一部分に関するデータのみをプロファイルしたい場合も多くあります。ここでは、適切な情報を選択するための方法について説明します。次のオプションを使用します。

- `prof` オプション (`pixie`, `uprofile`, `prof` コマンドで使用)
- `gprof` オプション (`hiprof` または `gprof` コマンドで使用)

`prof` オプションと `gprof` オプションの多くは類似した機能を実行し、類似した名前と構文を持っています。ここでは `prof` オプションの例を示し

ます。詳細については、`hiprof(1)`、`pixie(1)`、`uprofile(1)`、`prof(1)`、`gprof(1)` を参照してください。

`monitor` ルーチンを使用したプロファイルの制御については、8.9 節を参照してください。

8.6.1 プロファイルの表示を特定プロシージャのみに制限する

`prof` コマンドの `-only` オプションを使用すると、指定したプロシージャだけのプロファイル情報が表示されます。このオプションは、コマンド行に複数回使用できます。たとえば、次のようにコマンドを実行します。

```
% pixie -only mul_by_scalar -only add_vector sample
```

`-exclude` オプションを使用すると、指定したプロシージャ以外のすべてのプロシージャのプロファイル情報が表示されます。このオプションは、コマンド行に複数回使用できます。同じコマンド行に `-only` オプションと `-exclude` オプションを一緒に使用することはできません。

`prof` プロファイル・オプションの多くは、出力をパーセンテージで表示します。たとえば、合計の実行時間に対する特定のプロシージャの実行時間をパーセンテージで示す場合などがあります。

省略時の設定では、`-only` オプションおよび `-exclude` オプションを使用すると、プロシージャがリストから省かれている場合でも、`prof` はアプリケーションのすべてのプロシージャに基づいてパーセンテージを計算します。`-Only` および `-Exclude` オプションを使用すると、この動作を変更することができます。`-Only` および `-Exclude` オプションは、`-only` および `-exclude` と同じ働きをしますが、リストに含まれるプロシージャだけに基いて `prof` が計算を行うようにします。

`-totals` オプションを `-procedures` と `-invocations` のリストとともに使用すると、オブジェクトの個々のプロシージャではなく、オブジェクト・ファイル全体の累計の統計情報が表示されます。

8.6.2 ソース行ごとのプロファイル情報の表示

`prof` と `gprof` の `-heavy` および `-lines` オプションを使用すると、マシン・サイクル、命令、ページ・フォールト、キャッシュ・ミスなどの数がアプリケーションのソース行ごとに表示されます。`-asm` オプションでは、これらの情報が命令ごとに表示されます。

`-heavy` オプションでは、ソース行ごとにエントリが表示されます。エントリは、その行について最も使用量の多いものから順に並べられます。`-heavy` オプションを使用すると多数のエントリが表示されることが多いので、`-quit`、`-only`、`-exclude` オプションのいずれかを使用して、扱いやすい量に出力を減らすこともできます (8.6.3 項を参照)。

`-lines` オプションは `-heavy` オプションに似ていますが、出力のソート順序が異なります。このオプションでは、個々のプロシージャの行を、ソース・ファイル内での出現順に表示します。一度も実行されなかった行も表示されます。プロシージャそのものは、合計使用量の多いものから順に並べられます。

8.6.3 行ごとのプロファイル表示の制限

`-quit` オプションを使用すると、表示されるプロファイル出力の量が制限されます。このオプションは、`-procedures`、`-heavy`、および `-lines` プロファイル・モードからの出力に影響を及ぼします。

`-quit` オプションには、次の 3 つの形式があります。

<code>-quit N</code>	N 行より後のリストを切り捨てる。
<code>-quit N%</code>	表示が合計の $N\%$ より少ない最初の行より後のリストを切り捨てる。
<code>-quit Ncum%</code>	累計が全体の $N\%$ に達した行より後のリストを切り捨てる。

同一のコマンド行に複数のモードを指定した場合、`-quit` オプションは各モードの出力に影響を及ぼします。次のコマンドは、消費時間が上位 20 個のプロシージャと消費時間が上位 20 個のソース行のみを表示します。

```
% pixie -procedures -heavy -quit 20 sample
```

合計は、プロファイル・モードによって、時間の合計、実行された命令の合計数、呼び出しカウントの合計のいずれかに相当します。

8.6.4 プロファイル情報にシェアード・ライブラリを含める

`hiprof`、`pixie`、`third`、`prof`、`gprof` のいずれかのコマンドを使用する場合は、次のオプションにより、プログラムが使用するシェアード・ライブラリのプロファイル情報を選択して表示することができます。

- `-all` オプションは、実行可能ファイルに加え、データ・ファイル (1 つまたは複数) に記述されているすべてのシェアード・ライブラリのプロファイルを表示します (ある場合)。
- `-incobj lib` オプションは、実行可能ファイルに加え、指定されたシェアード・ライブラリのプロファイルを表示します。
- `-excobj lib` オプションは、`-all` が指定されている場合、指定されたシェアード・ライブラリのプロファイルを表示しません。

たとえば、次のコマンドは `user1.so` 以外のすべてのシェアード・ライブラリのプロファイル情報を表示します。

```
% prof -all -excobj user1.so sample
```

`cc` コマンドの `-p` オプションを使用して PC サンプリング・プロファイルを取得する場合は、`PROFFLAGS` 環境変数を使用して、計測機構付きバージョンのプログラムの実行時にシェアード・ライブラリのプロファイル情報を含めるか、または除外するかを指定できます (例 8-8 を参照)。`PROFFLAGS` 変数には `-all`、`-incobj`、および `excobj` オプションを指定できます。

シェアード・ライブラリに固有の情報については、8.6.4.1 項を参照してください。

8.6.4.1 計測機構付きシェアード・ライブラリの位置の指定

`LD_LIBRARY_PATH` 環境変数を使用すると、ローダに対して、計測機構付きシェアード・ライブラリを探す位置を指定することができます。

省略時の設定では、`hiprof`、`pixie`、または `third` を実行すると、`LD_LIBRARY_PATH` 変数は作業ディレクトリを指すよう自動的に設定されます。任意のディレクトリを指定するように、この変数を設定することができます。次に C シェルの例を示します。

```
% setenv LD_LIBRARY_PATH "my_inst_libraries"
```

8.7 プロファイル・データ・ファイルのマージ

プロファイルするプログラムがかなり複雑な場合は、プロファイルの正確な像を得るため、異なる入力でプログラムを複数回実行することがあります。どのプロファイル・ツールでも、プログラムの複数回の実行から得られたプロファイル・データをマージすることができます。ただし、計測機構付きプログラムが省略時の動作で既存のデータ・ファイルを上書きする場合は、ま

ず，その動作を無効にする必要があります。 8.7.1 項ではその方法について説明し， 8.7.2 項ではデータ・ファイルをマージする方法について説明します。

8.7.1 データ・ファイルの命名規則

プロファイル・データ・ファイルの省略時の名前は，それを作成するのに使用したツールによって異なります。

ツール	プロファイル・データ・ファイルの省略時の名前
hiprof	<i>program.hiout</i>
pixie	<i>program.Counts</i> (<i>program.Addr</i> s ファイルとともに使用)
uprofile	<i>umon.out</i>
cc -p	<i>mon.out</i>
cc -pg	<i>gmon.out</i>

省略時の設定では，複数のプロファイルを実行すると，作業ディレクトリ内の既存のデータ・ファイルは実行のたびに上書きされます。 どのツールにもデータ・ファイルの名称変更オプションがあるので，各実行時にファイルを保存することができます。 *hiprof*，*pixie*，および *uprofile* コマンドには，次のオプションがあります。

<code>-dirname path</code>	データ・ファイルを作成するディレクトリを指定する。
<code>-pids</code>	計測機構付きプログラムの実行のプロセス ID をデータ・ファイル名に追加する。

たとえば，次のコマンド・シーケンスでは，*sample.pid.hiout* 形式の 2 つのデータ・ファイルがサブディレクトリ *profdata* に作成されます。

```
% hiprof -dir profdata -pids sample
% ./sample
% ./sample
```

次に，*gprof* コマンドを使用するときにワイルドカードを指定して，ディレクトリ内のプロファイル・データ・ファイルをすべて含めることができます。

```
% gprof -b sample profdata/*
```

プロファイル・オプション *cc -p* または *cc -pg* を使用する場合は，*PROF_FLAGS* 環境変数を (計測機構付きプログラムを実行する前に) 設定する必要があります。 たとえば，次のコマンド・シーケンスでは，*pid.sample*

形式の 2 つのデータ・ファイルがサブディレクトリ `profdata` に作成されます (C シェルの例)。

```
% cc -pg -o sample -g1 -O2 *.c -lm
% setenv PROFFLAGS "-dir profdata -pids"
% ./sample
% ./sample
```

`PROFFLAGS` 環境変数は、プログラムの実行中に、プログラムのプロファイルの動作に影響を及ぼしますが、ポストプロセッサ `prof` と `gprof` には影響を与えません。 `PROFFLAGS` にヌル文字列を設定すると、プロファイルは行われません。

ファイルの命名規則の詳細については、ツールのリファレンス・ページを参照してください。 マルチスレッド・プログラムのファイル命名規則については、8.8 節を参照してください。

8.7.2 データ・ファイルのマージ手法

プログラムを複数回実行し、複数のプロファイル・データ・ファイルを作成した後、これらのファイルの平均に基づいてプロファイル情報を表示することができます。

ポストプロセッサ `prof` または `gprof` を、使用するプロファイル手法に応じて使い分けます。

プロファイル・ツール	ポストプロセッサ
<code>cc -p</code> , <code>uprofile</code> , または <code>pixie</code>	<code>prof</code>
<code>cc -pg</code> , または <code>hiprof</code>	<code>gprof</code>

マージ手法の 1 つは、各データ・ファイルの名前をコマンド行に明示的に指定することです。たとえば、次のコマンドでは、`hiprof` を使用して生成した 2 つのプロファイル・データ・ファイルのプロファイル情報を表示します。

```
% gprof sample sample.1510.hiout sample.1522.hiout
```

しかし、多くの異なるプロファイル・データ・ファイルを追跡することが困難な場合があります。そのため、`prof` と `gprof` では、複数のデータ・ファイルをまとめて 1 つのファイルにマージする `-merge` オプションを提供しています。 `prof` が `pixie` モードで動作する場合 (`prof -pixie`)、`-merge` オプションは `.Counts` ファイルを結合します。 `prof` が PC サンプリング・

モードで動作する場合は、`-merge` オプションは `mon.out` や他のプロファイル・データ・ファイルを結合します。

次の例では、2 つのプロファイル・データ・ファイルを `total.out` という名前の 1 つのデータ・ファイルにマージします。

```
% prof -merge total.out sample 1773.sample 1777.sample
```

この後、通常の `mon.out` ファイルを使用する場合と同様に、結合されたファイルを使用してプロファイル・データを表示できます。

```
% prof -procedures sample total.out
```

`-pixie` モードの場合も、マージ処理は類似しています。実行可能ファイル名、`.Addr`s ファイル、およびそれぞれの `.Count`s ファイルを指定します。たとえば、次のようにコマンドを実行します。

```
% prof -pixie -merge total.Counts a.out a.out.Addr \
a.out.Counts.1866 a.out.Counts.1868
```

8.8 マルチスレッド・アプリケーションのプロファイル

注意

予想される論理および性能上の問題についてマルチスレッドのアプリケーションを解析するには、Visual Threads を使用できます。Visual Threads は、「Tru64 UNIX Associated Products Volume 1」CD-ROM に含まれています。Visual Threads は、POSIX Threads Library アプリケーションと Java アプリケーションに使用できます。dxthreads(1) を参照してください。

マルチスレッド・アプリケーションのプロファイルは、スレッドなしのアプリケーションのプロファイルと本質的に同じです。ただし、マルチスレッド・アプリケーションをプロファイルするには、`cc` コマンドの定義に従い、`-pthread` または `-threads` オプションを使用して、プログラムをコンパイルする必要があります。これ以外のスレッド・パッケージはサポートされていません。

`hiprof(1)`、`pixie(1)`、または `third(1)` を使用する場合は、`-pthread` オプションを指定して、そのツールでスレッド・セーフなプロファイルがサポートされるようにします。uprofile(1) コマンドと `cc` コマンドの `-p` およ

び `-pg` オプションには、スレッド・セーフにするための追加のオプションは必要ありません。

マルチスレッド・アプリケーションのプロファイルを行う場合、省略時の設定では 1 つのプロファイルですべてのスレッドをカバーします。この場合は、プロセス全体にわたるプロファイルが行われ、すべてのスレッドからのプロファイル・データを含む 1 つの出力ファイルが作成されます。

`hiprof(1)` または `pixie(1)` を使用する場合は、`-threads` オプションを使用するとスレッドごとのデータが得られます。

`cc` コマンドの `-p` または `-pg` オプションを使用する場合は、次の C シェルの例に示すように、`PROFFLAGS` 環境変数に `-threads` を設定してスレッドごとのプロファイルを指定します。

```
setenv PROFFLAGS "-threads"
```

`uprofile(1)` および `third(1)` ツールでは、スレッドごとのデータは得られません。

スレッドごとのプロファイル・データ・ファイルには、一意のスレッド番号を含む名前が付けられます。この番号は、スレッドが作成されたシーケンスまたはプロファイルが開始されたシーケンスを反映します。

`monitor()` または `monstartup()` 呼び出し (8.9 節を参照) をスレッド付きプログラムで使用する場合は、アプリケーションのプロファイルを完全に制御できるように、最初に `PROFFLAGS` に `"-disable_default -threads"` を設定しておく必要があります。

アプリケーションが `monitor()` を使用して、プロファイルされる各スレッドに別個のバッファを割り当てる場合には、最初に `PROFFLAGS` に `"-disable_default -threads"` を設定する必要があります。これは、使用されるファイル命名規則はこの設定の影響を受けるからです。`-threads` オプションがない場合は、`monitor` または `monstartup` の最初の呼び出しの結果として使用されるバッファとアドレス範囲が、その後にプロファイルを要求するすべてのスレッドに適用されます。この場合、プロファイルされるすべてのスレッドをカバーする 1 つのデータ・ファイルが作成されます。

プロセス内の各スレッドは、`monitor()` または `monstartup()` ルーチンを呼び出して、自らのプロファイルを開始する必要があります。

8.9 monitor ルーチンを使用したプロファイルの制御

cc コマンドの `-p` および `-pg` モードの Tru64 UNIX システムでの省略時の動作は、プログラムのテキスト・セグメント全体のプロファイルを行い、プロファイル・データを `mon.out` (`prof` プロファイルの場合) または `gmon.out` (`gprof` プロファイルの場合) に出力することです。大規模なプログラムでは、テキスト・セグメント全体のプロファイルを行う必要がない場合があります。monitor ルーチンには、関数アドレス範囲の下位アドレスと上位アドレスの境界によって指定されるプログラムの一部をプロファイルする機能があります。

monitor ルーチンには、次のものがあります。

monitor	このルーチンを使用すると、特定のテキスト範囲のプロファイルをオン/オフすることにより、プロファイルを明示的に制御できます。このルーチンは、 <code>gprof</code> プロファイルでは使用できません。
monstartup	monitor と似ていますが、このルーチンではアドレス範囲の指定のみが可能で、 <code>gprof</code> プロファイルで使用できます。
moncontrol	このルーチンを monitor および monstartup と併用すると、プログラムの実行中に特定のプロセスまたはスレッドに対して、PC サンプリングのオン/オフを切り替えることができます。
monitor_signal	このルーチンを使用すると、デーモンなど終了しないプログラムのプロファイルを行うことができます。

monitor および monstartup を使用すると、静的な実行可能ファイルだけでなく個々のシェアド・ライブラリ内のアドレス範囲をプロファイルすることができます。

これらの関数の詳細については、`monitor(3)` を参照してください。

省略時の設定では、プログラムの実行が開始した直後にプロファイルが開始されます。この動作を変更するには、次に示す C シェルの例のように、`PROF_FLAGS` 環境変数に `-disable_default` を設定します。

```
setenv PROFFLAGS "-disable_default"
```

次に、monitor ルーチンを使用すると、monitor または monstartup の最初の呼び出しの後にプロファイルを開始するように設定できます。

例 8-11 では、monstartup ルーチンと monitor ルーチンをプログラム内で使用して、プロファイルの開始と終了を行う方法を示します。

例 8-11: monstartup() と monitor() の使用

```
/* Profile the domath() routine using monstartup.
 * This example allocates a buffer for the entire program.
 * Compile command: cc -p foo.c -o foo -lm
 * Before running the executable, enter the following
 * from the command line to disable default profiling support:
 * setenv PROFFLAGS -disable_default
 */

#include <stdio.h>
#include <sys/syslimits.h>

char dir[PATH_MAX];

extern void __start();
extern unsigned long _etext;

main()
{
    int i;
    int a = 1;

    /* Start profiling between __start (beginning of text)
     * and _etext (end of text). The profiling library
     * routines will allocate the buffer.
     */

    monstartup(__start, &_etext);

    for(i=0; i<10; i++)
        domath();

    /* Stop profiling and write the profiling output file. */

    monitor(0);
}
domath()
{
    int i;
    double d1, d2;
```

例 8-11: monstartup() と monitor() の使用 (続き)

```
    d2 = 3.1415;
    for (i=0; i<1000000; i++)
        d1 = sqrt(d2)*sqrt(d2);
}
```

外部名 `_etext` がすべてのプログラム・テキストの直前にあります。詳細については、`end(3)` を参照してください。

`PROFFLAGS` 環境変数を `-disable_default` に設定すると、省略時のプロファイル・バッファ・サポートが無効になるため、例 8-12 に示すように、バッファをプログラム内に割り当てることができます。

例 8-12: プログラム内のプロファイル・バッファの割り当て

```
/* Profile the domath routine using monitor().
 * Compile command: cc -p foo.c -o foo -lm
 * Before running the executable, enter the following
 * from the command line to disable default profiling support:
 * setenv PROFFLAGS -disable_default
 */

#include <sys/types.h>
#include <sys/syslimits.h>

extern char *calloc();

void domath(void);
void nextproc(void);

#define INST_SIZE 4          /* Instruction size on Alpha */
char dir[PATH_MAX];

main()
{
    int i;
    char *buffer;
    size_t bufsize;

    /* Allocate one counter for each instruction to
     * be sampled. Each counter is an unsigned short.
     */

    bufsize = (((char *)nextproc - (char *)domath)/INST_SIZE)
```


例 8-12: プログラム内のプロファイル・バッファの割り当て (続き)

```
* sizeof(unsigned short);

/* Use calloc() to ensure that the buffer is clean
 * before sampling begins.
 */

buffer = calloc(bufsize,1);

/* Start sampling. */
monitor(domath,nextproc,buffer,bufsize,0);
for(i=0;i<10;i++)
    domath();

/* Stop sampling and write out profiling buffer. */
monitor(0);
}
void domath(void)
{
    int i;
    double d1, d2;

    d2 = 3.1415;
    for (i=0; i<1000000; i++)
        d1 = sqrt(d2)*sqrt(d2);
}

void nextproc(void)
{}
```

`monitor_signal()` ルーチンを使用すると、終了しないプログラムのプロファイルを行うことができます。プログラム内でこのルーチンを単一のハンドラとして宣言し、`prof` または `gprof` プロファイル用にプログラムを作成します。プログラムの実行中に、`kill` コマンドを使用してシェルから信号を送信します。

プログラムが信号を受け取ると、`monitor_signal` が呼び出され、データ・ファイルにプロファイル・データを書き込みます。プログラムが別の信号を受け取ると、データ・ファイルは上書きされます。

例 8-13 に、`monitor_signal` ルーチンの使用方法を示します。

`PROF_FLAGS` 環境変数を `-sigdump SIGNAME` に設定すると、新しいプログラム・コードを追加することなく同じ機能を得ることができます。

例 8-13: monitor_signal() を使用した , 終了しないプログラムのプロファイル

```
/* From the shell, start up the program in background.
 * Send a signal to the process, for example: kill -30 <pid>
 * Process the [g]mon.out file normally using gprof or prof
 */

#include <signal.h>

extern int monitor_signal();

main()
{
    int i;
    double d1, d2;

    /*
     * Declare monitor_signal() as signal handler for SIGUSR1
     */
    signal(SIGUSR1, monitor_signal);
    d2 = 3.1415;
    /*
     * Loop infinitely (absurd example of non-terminating process)
     */
    for (;;)
        d1 = sqrt(d2)*sqrt(d2);
}
```

Atom ツールの使用および開発

プログラム分析ツールは、コンピュータ設計者およびソフトウェア・エンジニアにとってきわめて重要です。プログラム分析ツールを使用して、コンピュータ設計者は新しいアーキテクチャ設計のテストと測定を行い、ソフトウェア・エンジニアは、プログラム内のコードのクリティカルな部分を特定したり、分岐予測や命令スケジューリング・アルゴリズムがどの程度機能しているかを検査します。プログラム分析ツールは、基本ブロックのカウントから、データのキャッシュ・シミュレーションに至るまでの問題に必要です。これらのタスクを実行するツールは異なるように思われますが、コードの計測によって、それぞれを簡単に効率的に実現できます。

Atom では、さまざまなツールを作成できる柔軟なコード計測インタフェースを提供します。Atom は、計測およびオブジェクト・コード操作のための機構を提供し、ツール設計者がプログラムの計測するポイントを特定できるようにして、それぞれの問題に固有の部分とすべての問題に共通の部分とを分離します。Atom は、完全なプログラムを構成するオブジェクト・モジュール上で動作するので、どのコンパイラおよび言語からも独立しています。

この章では、次の 2 つの項目について説明します。

- インストール済みの Atom ツールおよび現在開発中の新しい Atom ツールの実行方法 (9.1 節)
- 専用 Atom ツールの開発方法 (9.2 節)

9.1 Atom ツールの実行

以降の各項では、次の項目について説明します。

- インストール済みの Atom ツールの使用 (9.1.1 項)
- 開発中のテスト用 Atom ツール (9.1.2 項)

9.1.1 インストール済みの Atom ツールの使用

Tru64 UNIX オペレーティング・システムでは、表 9-1 に示す多くの Atom ツールをサンプルとして提供しており、独自にカスタム設計された Atom ツールの開発に役立てることができます。これらのツールは、実際の運用を目的としたものではなく、Atom のプログラミング・インタフェースを説明するために、ソース形式で配布されます。9.2 節では、いくつかのツールについて詳細に説明しています。

表 9-1: サンプルのインストール済み Atom ツール

ツール	説明
branch	条件付き分岐をすべて計測して、正しく予測されている数を確認します。
cache	アプリケーションが 8 KB の直接マップ・キャッシュで実行される場合の、キャッシュ・ミスの割合を確認します。
dtb	アプリケーションが 8 KB のページおよびそれに完全に対応する変換バッファを使用する場合の、dtb (データ変換バッファ) のミスの数を確認します。
dyninst	命令、ロード、ストア、ブロック、およびプロシージャの、基本的な動的カウンタを提供します。
inline	インライン化の潜在的候補を特定します。
iprof	各プロシージャが呼び出された回数、および、各プロシージャによって実行された命令の数をプリントします。
malloc	malloc 関数の各呼び出しを記録し、アプリケーションの割り当てられたメモリの要約をプリントします。
prof	pthread プログラムで各プロシージャによって実行された命令の数をプリントします。
ptrace	各プロシージャの呼び出し時に、その名前をプリントします。
replace	分析ルーチンからアプリケーションの置換したエントリ・ポイントを呼び出します。
trace	実行時に、アドレス・トレースを生成して、すべてのロードとストア操作の実効アドレスのログ、およびすべての基本ブロックの先頭のアドレスのログを取ります。

サンプルのツールは、`/usr/lib/cmplrs/atom/examples` ディレクトリにあります。各サンプルには、次の 3 つのファイルが含まれます。

9-2 Atom ツールの使用および開発

- 計測ファイル — Atom の API を使用して、アプリケーション・プログラムを修正し、ツールが提供した追加のルーチンをプログラム実行中の特定の時間に起動するようにする、C ソース・ファイルです。
- 分析ファイル — 修正されたプログラムの実行時に起動されたルーチンを含む C ソース・ファイルです。これらの分析ルーチンは、ツールによって報告される実行時データを収集します。
- 記述ファイル (*toolname.desc*) — Atom にツールの計測ファイルおよび分析ファイルの名前を通知するテキスト・ファイルです。また、ツールの実行時に Atom が使用する任意のオプションも通知します。

実際の運用を目的とする Atom ツール、または製品としてカスタマに配布される Atom ツールには、通常、固有のソースの代わりに .o オブジェクト・モジュールがインストールされています。Tru64 UNIX の *hiprof(1)*、*pixie(1)*、*third(1)* の各コマンド、および Visual Thread 製品には、このようにして配布、インストール、および実行される Atom ツールが含まれています。規約により、計測ファイル、分析ファイル、および記述ファイルは、*/usr/lib/cmplrs/atom/tools* ディレクトリに格納されます。

インストール済みの Atom ツールまたはサンプルをアプリケーション・プログラム上で実行するには、次の形式で *atom(1)* コマンドを使用します。

atom *application_program* *-tool toolname* [*-env environment*] [*options...*]

この形式の *atom* コマンドでは、*-tool* オプションは必須であり、*-env* オプションを受け入れます。

-tool オプションは、使用するインストール済みの Atom ツールを識別します。省略時の設定では、Atom はインストール済みのツールを */usr/lib/cmplrs/atom/tools* ディレクトリおよび */usr/lib/cmplrs/atom/examples* ディレクトリ内で検索します。コロンで区切った追加のディレクトリのリストを *ATOMTOOLPATH* 環境変数に追加して、検索パスにディレクトリを追加することができます。

-env オプションは、ツールの代替バージョンが必要であることを示します。たとえば、Tru64 UNIX のツールには、*-env threads* でスレッド・セーフ・バージョンの実行を要求するものがあります。

atom(1) コマンドは、省略時の設定の *toolname.desc* ファイルではなく *toolname.env.desc* ファイルを検索します。このコマンドは、

指定した環境の記述ファイルが見つからない場合にエラー・メッセージを表示します。

9.1.2 開発中のテスト用ツール

atom(1) コマンドの 2 つ目のコマンド形式は、開発中の新しい Atom ツールのコンパイルと実行を容易にする目的で提供されています。次のように、コマンド行で計測および分析ファイルの名前を直接指定するだけです。

atom application_program instrumentation_file [analysis_file] [options...]

このコマンド形式は、*instrumentation_file* パラメータを必要とし、*analysis_file* パラメータを受け付けます。ただし、`-tool` または `-env` オプションは受け付けません。

instrumentation_file パラメータには、C ソース・ファイルの名前または Atom ツールの計測プロシージャを含むオブジェクト・モジュールを指定します。計測プロシージャが複数のファイルに記述されている場合は、`ld` コマンドで `-r` オプションを指定すると、各ファイルの `.o` が 1 つのファイルと一緒にリンクされます。規約により、ほとんどの計測ファイルには接尾語 `.inst.c` または `.inst.o` が付けられています。

このパラメータ用にオブジェクト・モジュールを渡す場合は、モジュールを `-gl` または `-g` オプションのいずれかを指定してコンパイルすることを検討してください。計測プロシージャにエラーが存在する場合、計測プロシージャがこのようにコンパイルされていると、Atom はより完全な診断メッセージを発行できます。

analysis_file パラメータには、Atom ツールの分析プロシージャを含む C のソース・ファイルまたはオブジェクト・モジュールの名前を指定します。分析ルーチンが複数のファイルに存在する場合は、`ld` コマンドで `-r` オプションを指定すると、各ファイルの `.o` が 1 つのファイルと一緒にリンクされます。計測ファイルが、計測するアプリケーションに対して分析プロシージャを呼び出さない場合には、分析ファイルを指定する必要はありません。規約により、ほとんどの分析ファイルには接尾語 `.anal.c` または `.anal.o` が付けられます。

分析ルーチンを単一のコンパイル単位としてコンパイルすると、性能が向上する場合があります。

計測ソース・ファイルおよび分析ソース・ファイルは、複数指定することができます。次の例では、複数のソース・ファイルから計測および分析の複合オブジェクトを作成します。

```
% cc -c file1.c file2.c
% cc -c file7.c file8
% ld -r -o tool.inst.o file1.o file2.o
% ld -r -o tool.anal.o file7.o file8.o
% atom hello tool.inst.o tool.anal.o -o hello.atom
```

注意

分析プロシージャは、C++ で記述することもできます。アプリケーションから呼び出せるようにするには、各プロシージャに `extern "C"` タイプを割り当てる必要があります。また、`atom` コマンドを入力する前に、分析ファイルのコンパイルとリンクも行わなければなりません。たとえば、次のようになります。

```
% cxx -c tool.a.C
% ld -r -o tool.anal.o tool.a.o -lcxx -lexc
% atom hello tool.inst.c tool.anal.o -o hello.atom
```

9.1.3 Atom オプション

`-tool` および `-env` オプションを除いて、`atom` コマンドの2つの形式はいつでも、`atom(l)` に記述されている残りのオプションをすべて受け付けます。特に注意を必要とするオプションは次のとおりです。

`-A1`

セーブおよびリストアする必要のあるレジスタ数を減少させることにより、Atom が分析ルーチンの呼び出しを最適化できるようにします。いくつかのツールでは、このオプションを指定すると、計測機構付きアプリケーションの性能が2倍向上します(その代わり、アプリケーションのサイズが多少増加します)。省略時の動作では、Atom はこれらの最適化を適用しません。

`-all`

共用実行可能ファイル内の、静的にロードしたシェアード・ライブラリをすべて計測します。

-debug

計測ルーチンを dbx デバッガでデバッグできるようにします。Atom は計測ルーチンの起動時に、シンボリック・デバッガへ制御を移します。次の例では、ptrace サンプル・ツールが、dbx デバッガのもとで実行されます。計測は 12 行目で停止され、プロシージャ名が出力されます。

```
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace -debug
dbx version 3.11.8
Type 'help' for help.
Stopped in InstrumentAll
(dbx) stop at 12
[4] stop at "/udir/test/scribe/atom.user/tools/ptrace.inst.c":12
(dbx) c
[3] [InstrumentAll:12 ,0x12004dea8] if (name == NULL) name = "UNKNOWN";
(dbx) p name
0x2a391 = "__start"
```

-ladebug

計測ルーチンをオプションの ladebug デバッガでデバッグできるようにします。Atom は、計測ルーチンの起動時に ladebug に制御を移します。計測ルーチンがスレッド化されている場合、または内部に C++ のコードが含まれている場合には、ladebug を使用します。詳細は、『*Ladebug Debugger Manual*』を参照してください。

-excobj

指定したシェアード・ライブラリを計測から除きます。-excobj オプションを複数回使用すると、複数のシェアード・ライブラリを指定できます。

-fork

fork のサポートが必要であることを指定します。このオプションを使用すると、マルチスレッド・アプリケーションでデッドロックを回避します。

-ga (-g)

デバッグ情報を持った計測機構付きプログラムを作成します。このオプションを使用すると、分析ルーチンをシンボリック・デバッガでデバッグできるようになります。-ga (または -g) では、できるだけ省略時の -A0 オプション (-A1 ではない) を使用するようにしてください。たとえば、次のようになります。

```
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace -ga
% dbx hello.ptrace
```



```
dbx version 3.11.8
Type 'help' for help.
(dbx) stop in ProcTrace
[2] stop in ProcTrace
(dbx) r
[2] stopped at [ProcTrace:5 ,0x120005574] fprintf (stderr,"%s\n",name);
(dbx) n
__start
[ProcTrace:6 ,0x120005598] }
```

-gap

デバッグ情報を持った計測機構付きプログラムを作成します。このオプションを使用すると、分析ルーチンおよびアプリケーション・ルーチンのデバッグが可能になります。アプリケーション内の変数およびプロシージャの名前すべてに、接頭語の “_APP_” が付加されます。-gpa を使用する場合は、省略時の -A0 オプション (-A1 ではない) をお勧めします。

-gp

デバッグ情報を持った計測機構付きプログラムを作成します。このオプションを使用すると、アプリケーション・ルーチンをシンボリック・デバッグでデバッグできるようになります。

-gpa

デバッグ情報を持った計測機構付きプログラムを作成します。このオプションを使用すると、分析ルーチンおよびアプリケーション・ルーチンのデバッグが可能になります。分析オブジェクト内の変数およびプロシージャの名前すべてに、接頭語の “_ANA_” が付加されます。-gpa を使用する場合は、省略時の -A0 オプション (-A1 ではない) をお勧めします。

-heapbase

分析ヒープのベースを変更します。省略時の分析ヒープの位置が、アプリケーション・プログラムが使用するアドレス範囲と競合する場合は -heapbase オプションを使用します。新しいベースとして、新しい 16 進数のアドレス位置か、省略時の 31 ビットのアドレス位置か、アプリケーションの bss セグメントの末尾の次のページのいずれかを選択できます。

-ii

以前に計測したシェアード・ライブラリの再使用 (または増分の計測) をできるようにします。

-incobj

指定したシェアード・ライブラリを計測します。 `-incobj` オプションを複数回使用して、複数のシェアード・ライブラリを指定することもできます。

-keep

Atom が作成する一時ファイルを、現行の作業ディレクトリに置き、計測が完了しても削除しないことを指定します。

-L

ライブラリのディレクトリでシェアード・オブジェクト・ライブラリを検索する順序を変更し、Atom が省略時のライブラリ・ディレクトリをまったく検索しないようにします。このオプションは、省略時のディレクトリを検索せず、`-Ldir` が指定したディレクトリのみを検索するようにする場合に使用します。

`-Ldir`

ライブラリのディレクトリでシェアード・オブジェクト・ライブラリを検索する順序を変更し、Atom が `dir` ディレクトリを検索した後で省略時のライブラリ・ディレクトリを検索するようにします。複数の `-Ldir` オプションを指定すると、複数のディレクトリ名を指定できます。

-map

計測機構付き実行可能ファイルのセクションの開始アドレスのリストを作成します。

-o

実行可能な出力ファイルの名前を指定します。

-pthread

スレッド・セーフ・サポートが必須であることを指定します。このオプションは、スレッド・アプリケーションの計測を行う際に使用します。

`-shlib`*dir*

Atom が計測機構付きシェアード・ライブラリを書き込む既存のディレクトリを指定します。

`-suffix`

Atom が計測機構付きバージョンを書き込む際に、各オブジェクト名に付加するファイル名の接尾語を指定します。

`-toolargs`

Atom ツールの計測ルーチンに引数を渡します。Atom は、C プログラムに渡す場合と同様に、引数 *argc* および *argv* を使用して、main プログラムに引数を渡します。たとえば、次のようになります。

```
#include <stdio.h>
unsigned InstrumentAll(int argc, char **argv) {
    int i;
    for (i = 0; i < argc; i++) {
        printf(stderr, "argv[%d]: %s\n", argv[i]);
    }
}
```

次の例は、Atom が引数 `-toolargs` を渡す方法を示しています。

```
% atom hello args.inst.c -toolargs="8192 4"
argv[0]: hello
argv[1]: 8192
argv[2]: 4
```

`-v`

Atom が計測機構付きプログラムを作成する際に行う各ステップを表示します。

`-version`

Atom のバージョン番号を表示します。

`-w0`

通常は出力されないような警告も含め、すべての警告メッセージを表示します。

`-w1`

無視してかまわない警告メッセージは出力しません。これは省略時の動作です。

-w2

分析ルーチンを処理する際に出力される警告メッセージは出力しません。

-w3

シェアード・ライブラリエラーの処理に関する警告メッセージは出力しません。

-Wla および -Wca

指定したオプションを、それぞれ分析ファイルのリンクとコンパイルのフェーズに渡します。

-Wli および -Wci

指定したオプションを、それぞれ計測ファイルのリンクとコンパイルのフェーズに渡します。

9.2 Atom ツールの開発

この節では、Atom ツールの開発方法について説明します。

9.2.1 Atom によるアプリケーションの表示

Atom は、アプリケーションをコンポーネントの階層として表示します。

1. プログラム

実行可能プログラムとすべてのシェアード・ライブラリを含みます。

2. オブジェクトの集合

オブジェクトは、メインの実行可能プログラムまたは任意のシェアード・ライブラリのいずれかです。オブジェクトは、独自の属性 (名前など) のセットを持ち、プロシージャの集合から構成されます。

3. プロシージャの集合

各プロシージャはエントリ・ポイントと基本ブロックの集合から構成されます。

4. 基本ブロックの集合

各基本ブロックは命令の集合から構成されます。

5. 命令の集合

Atom ツールは、アプリケーション・プログラム内のプロシージャ、エントリ・ポイント、基本ブロック、または命令の境界に、計測ポイントを挿入します。たとえば、基本ブロック・カウント・ツールは、各基本ブロックの先頭を計測し、データ・キャッシュ・シミュレータは、ロードおよびストアの各命令を計測し、分岐予測アナライザは、各条件付き分岐命令を計測します。

Atom では、どの計測ポイントでも、ツールによって、分析ルーチンへの呼び出しを挿入できます。ツールでは、呼び出しが、オブジェクト、プロシージャ、エントリ・ポイント、基本ブロック、または命令の、前に行われるか後に行われるかを指定できます。

9.2.2 Atom の計測ルーチン

ツールの計測ルーチンには、アプリケーションのオブジェクト、プロシージャ、エントリ・ポイント、基本ブロック、および命令をトラバースして、計測ポイントを探索したり、分析プロシージャの呼び出しを追加したり、アプリケーションの計測機構付きバージョンを作成するコードが含まれています。

`atom_instrumentation_routines(5)` で説明されているように、計測ルーチンは、ツールの必要に応じて、次のインタフェースのうちのいずれか 1 つを利用できます。

```
Instrument (int iargc, char **iargv, Obj *obj)
```

Atom は、アプリケーション・プログラムの各オブジェクトに対して `Instrument` ルーチンを呼び出します。その結果、`Instrument` ルーチンは、オブジェクト・ナビゲーション・ルーチン (`GetFirstObj` など) を使用する必要がなくなります。Atom は、`Instrument` ルーチンに次のオブジェクトを渡す前に、変更された各オブジェクトを自動的にコーディングするので、`Instrument` ルーチンは、`BuildObj`、`WriteObj`、または `ReleaseObj` ルーチンを呼び出す必要がありません。`Instrument` インタフェースを使用する場合には、`InstrumentInit` ルーチンを定義して、Atom が最初のオブジェクトに対して `Instrument` を呼び出す前に、必要なタスク (分析ルーチン・プロトタイプの実装、プログラム・レベルの計測呼び出しの追加、グローバルな初期化など) を実行することができます。また、`InstrumentFini` ルーチンを定義して、Atom が最後のオブジェクトに

対して `Instrument` を呼び出した後で、必要なタスク (グローバルなクリーンアップなど) を実行することもできます。

```
InstrumentAll (int iargc, char **iargv)
```

`Atom` は、アプリケーション・プログラム全体に対して一度 `InstrumentAll` ルーチン呼び出しします。これは、ツールの計測コード自体がアプリケーションのオブジェクトをトラバースする方法を決定できるようにします。このメソッドでは、`InstrumentInit` または `InstrumentFini` ルーチンは存在しません。`InstrumentAll` ルーチンは、`Atom` のオブジェクト・ナビゲーション・ルーチン呼び出し、`BuildObj`、`WriteObj`、または `ReleaseObj` ルーチンを使用して、アプリケーションのオブジェクトを管理する必要があります。

計測ルーチン・インタフェースに関係なく、`Atom` は、`-toolargs` オプションで指定された引数をルーチンに渡します。`Instrument` インタフェースの場合、`Atom` は、現在のオブジェクトを指すポインタも渡します。

9.2.3 `Atom` の計測インタフェース

`Atom` は、アプリケーションを計測するための総合的なインタフェースを提供します。このインタフェースでは、次のタイプの処理をサポートしています。

- プログラムのオブジェクト、プロシージャ、エントリ・ポイント、基本ブロック、および命令の間のナビゲーション (9.2.3.1 項を参照)。
- オブジェクトの作成、解放、およびコーディング (9.2.3.2 項を参照)。
- アプリケーションのさまざまな構成要素についての情報の取得 (9.2.3.3 項を参照)。
- 名前および呼び出しターゲットの解決 (9.2.3.4 項を参照)。
- プログラム内の目的とする位置への分析ルーチンの呼び出しの追加 (9.2.3.5 項を参照)。
- プログラム内のエントリ・ポイント呼出への介入。 9.2.3.6 項を参照してください。

9.2.3.1 プログラム内のナビゲーション

`atom_application_navigation(5)` で説明しているように、`Atom` のアプリケーション・ナビゲーション・ルーチンは、`Atom` ツールの計測ルー

チンが、次のように、分析プロシージャの呼び出しを追加するアプリケーション内の位置を見つけられるようにします。

- ルーチン `GetFirstObj` , `GetLastObj` , `GetNextObj` , および `GetPrevObj` は、プログラムのオブジェクト間をナビゲートします。非共用プログラムの場合には、オブジェクトは 1 つだけです。呼び出し共用プログラムの場合、最初のオブジェクトがメイン・プログラムに対応します。残りのオブジェクトは、動的にリンクされた各シェアード・ライブラリです。
- ルーチン `GetFirstObjProc` および `GetLastObjProc` はそれぞれ、指定されたオブジェクト内の最初のプロシージャまたは最後のプロシージャへのポインタを返します。ルーチン `GetNextProc` および `GetPrevProc` は、オブジェクトのプロシージャ間をナビゲートします。
- ルーチン `GetFirstEntry` および `GetLastEntry` はそれぞれ、指定されたプロシージャについて、最初または最後のエントリ・ポイントへのポインタを返します。ルーチン `GetNextEntry` および `GetPrevEntry` は、プロシージャのエントリ・ポイント間をナビゲートします。
- ルーチン `GetFirstBlock` , `GetLastBlock` , `GetNextBlock` , および `GetPrevBlock` は、プロシージャの基本ブロック間をナビゲートします。
- ルーチン `GetFirstInst` , `GetLastInst` , `GetNextInst` , および `GetPrevInst` は、基本ブロックの命令の間をナビゲートします。
- `GetInstBranchTarget` ルーチンは、指定された分岐命令のターゲットである命令を指すポインタを返します。
- `GetProcObj` ルーチンは、指定されたプロシージャを含むオブジェクトを指すポインタを返します。
- ルーチン `GetEntryProc` および `GetBlockProc` は、それぞれ指定されたエントリ・ポイントまたは基本ブロックを含むプロシージャを指すポインタを返します。
- `GetEntryBlock` ルーチンは、指定したエントリ・ポイントの最初の基本ブロックを指すポインタを返します。`GetInstBlock` ルーチンは、指定した命令を含む基本ブロックを指すポインタを返します。

9.2.3.2 オブジェクトの作成

atom_object_management(5) に説明があるように，Atom のオブジェクト管理ルーチンによって，Atom ツールの InstrumentAll ルーチンは，オブジェクトの作成，コーディング，および解放を行うことができます。

BuildObj ルーチンは，Atom がオブジェクトを処理するために必要とする内部データ構造体を作成します。InstrumentAll ルーチンは，オブジェクト内のプロシージャをトラバースして，分析ルーチン呼び出しをオブジェクトに追加する前に，BuildObj ルーチンを呼び出さなければなりません。

WriteObj ルーチンは，指定されたオブジェクトの計測機構付きバージョンをコーディングして，以前に BuildObj ルーチンが作成した内部データ構造体の割り当てを解除します。ReleaseObj ルーチンは，指定のオブジェクトの内部データ構造体の割り当てを解除しますが，オブジェクトの計測機構付きバージョンは書き出しません。

IsObjBuilt ルーチンは，指定したオブジェクトが BuildObj ルーチンによって作成されているが，まだ WriteObj ルーチンによってコーディングされていないか，あるいは ReleaseObj ルーチンによって解放されていない場合，非ゼロの値を返します。

9.2.3.3 アプリケーションの構成要素に関する情報の取得

atom_application_query(5) に説明があるように，Atom のアプリケーション照会ルーチンによって，計測ルーチンは，プログラムおよびそのオブジェクト，プロシージャ，エントリ・ポイント，基本ブロック，命令に関する静的情報を取得できます。

表 9-2 に，プログラムに関する情報を提供するルーチンを示します。

表 9-2: Atom のプログラム照会ルーチン

ルーチン	説明
GetAnalName	atom コマンドに渡される分析ファイルの名前を返します。このルーチンは，1 つの計測ファイルと複数の分析ファイルを使用するツールで役に立ちます。
GetErrantShlibName	Atom による処理が不可能なシェアード・ライブラリの名前を返します。

表 9-2: Atom のプログラム照会ルーチン (続き)

ルーチン	説明
GetErrantShlibErr	Atom シェアード・ライブラリを処理できない理由を説明するエラー・コードを返します。
GetProgInfo	プログラムのオブジェクト数, ツール・ライタが計測を要求するツールのオブジェクト数, Atom による処理が不可能なシェアード・ライブラリ数のいずれかを返します。

表 9-3 に, プログラムのオブジェクトに関する情報を提供するルーチンを示します。

表 9-3: Atom のオブジェクト照会ルーチン

ルーチン	説明
GetObjInfo	オブジェクトのテキスト, データ, bss セグメントについての情報, オブジェクトに含まれているプロシージャ, エントリ・ポイント, 基本ブロック, または命令の数, そのオブジェクト ID, オブジェクトのリンク方法に関する情報, または指定のオブジェクトを計測から除外する必要があるかどうかについての論理値のヒントのいずれかを返します。
GetObjInstArray	オブジェクトに含まれる 32 ビット命令から構成されている配列を返します。
GetObjInstCount	GetObjInstArray ルーチンによって返される配列に含まれる配列内の命令の数を返します。
GetObjName	指定されたオブジェクトの元のファイル名を返します。
GetObjOutName	計測機構付きオブジェクトの名前を返します。

次の計測ルーチンは, プログラムのオブジェクトに関する統計情報をプリントするものであり, Atom のオブジェクト照会ルーチンの使用方法を示しています。

```
1  #include <stdio.h>
2  #include <cmplrs/atom.inst.h>
3  unsigned InstrumentAll(int argc, char **argv)
4  {
5      Obj *o; Proc *p;
6      const unsigned int *textSection;
7      long textStart;
8      for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {
9          BuildObj(o);
10         textSection = GetObjInstArray(o);
```

```

11     textStart = GetObjInfo(o, ObjTextStartAddress);
12     printf("Object %d\n", GetObjInfo(o, ObjID));
13     printf("    Object name: %s\n", GetObjName(o));
14     printf("    Text segment start: 0x%x\n", textStart);
15     printf("    Text size: %ld\n", GetObjInfo(o, ObjTextSize));
16     printf("    Second instruction: 0x%x\n", textSection[1]);
17     ReleaseObj(o);
18 }
19 return(0);
20 }

```

この計測ルーチンは、プロシージャを実行可能プログラムに追加しないので、分析プロシージャは必要ありません。次の例に、このツールを使用して、プログラムをコンパイルおよび計測する処理を示します。計測機構付きプログラムのサンプル実行では、オブジェクト識別子、テキスト・セグメントのコンパイル時の開始アドレス、テキスト・セグメントのサイズ、および2番目の命令のバイナリがプリントされます。逆アセンブラによって、対応する命令を見つけるための便利な方法が提供されます。

```

% cc hello.c -o hello
% atom hello.info.inst.c -o hello.info
Object 0
  Object Name: hello
  Start Address: 0x120000000
  Text Size: 8192
  Second instruction: 0x239f001d
Object 1
  Object Name: /usr/shlib/libc.so
  Start Address: 0x3ff80080000
  Text Size: 901120
  Second instruction: 0x239f09cb
% dis hello | head -3
0x120000fe0: a77d8010    ldq t12, -32752(gp)
0x120000fe4: 239f001d    lda at, 29(zero)
0x120000fe8: 279c0000    ldah at, 0(at)
% dis /usr/shlib/libc.so | head -3
0x3ff800bd9b0: a77d8010    ldq t12, -32752(gp)
0x3ff800bd9b4: 239f09cb    lda at, 2507(zero)
0x3ff800bd9b8: 279c0000    ldah at, 0(at)

```

表 9-4 に、オブジェクトのプロシージャに関する情報を提供するルーチンを示します。

表 9-4: Atom のプロシージャ照会ルーチン

ルーチン	説明
GetProcInfo	『 <i>Calling Standard for Alpha Systems</i> 』 マニュアルおよび『 <i>Assembly Language Programmer's Guide</i> 』に定義されているように、プロシージャのスタック・フレーム、レジスタの保存、レジスタの使用法、およびプロログ特性に関する情報を返します。これらの値は、Third Degree のような、初期化されていない変数にアクセスするためのスタックを監視するツールにとっては重要です。プロシージャに含まれているエントリ・ポイントや基本ブロックや命令の数、プロシージャ ID、シンボルの解決のタイプ、最小または最大のソース行番号、最初の命令へのプロシージャのオフセット、代替エントリ・ポイントの有無、プロシージャ間の分岐やジャンプの有無、あるいはそのアドレスが取得されたかどうかの指示などの、プロシージャについての情報を返すこともできます。
ProcGP	プロシージャのグローバル・ポインタ (GP) を返します。
ProcFileName	プロシージャを含むソース・ファイル名を返します。
ProcName	プロシージャ名を返します。
ProcPC	プロシージャ内にある最初の命令のコンパイル時のプログラム・カウンタ (PC) を返します。

表 9-5 に、オブジェクトのメインおよび代替エントリ・ポイントに関する情報を提供するルーチンを示します。

表 9-5: Atom エントリ・ポイント照会ルーチン

ルーチン	説明
EntryName	エントリ・ポイント名を返します。
EntryPC	エントリ・ポイントの最初の命令の、コンパイル時のプログラム・カウンタ (PC) を返します。
GetEntryInfo	エントリ・ポイントに関する情報 (冗長かどうかなど) を返します。
GetEntryProc	エントリ・ポイントを囲む (中に含む) プロシージャを返します。

表 9-6 に、プロシージャの基本ブロックに関する情報を提供するルーチンを示します。

表 9-6: Atom の基本ブロック照会ルーチン

ルーチン	説明
BlockPC	基本ブロック内にある最初の命令のコンパイル時のプログラム・カウンタ (PC) を返します。
GetBlockInfo	基本ブロック内の命令数またはブロック ID を返します。
IsBranchTarget	ブロックが分岐命令のターゲットであるかどうかを示します。

表 9-7 に、基本ブロックの命令に関する情報を提供するルーチンを示します。

表 9-7: Atom の命令照会ルーチン

ルーチン	説明
GetInstBinary	アセンブラ言語命令の 32 ビット・バイナリ表現を返します。
GetInstClass	『Alpha Architecture Reference Manual』に定義されているように、命令クラス (浮動小数点のロードや整数のストアなど) を返します。
GetInstInfo	32 ビット命令全体を解析し、その命令の全部または一部を取得します。
GetInstRegEnum	GetInstInfo ルーチンによって返されたとき、命令フィールドからレジスタ・タイプ (浮動小数点または整数) を返します。
GetInstRegUsage	可能性のある各ソース・レジスタに対して 1 ビット・セット、可能性のある各デスティネーション・レジスタに対して 1 ビット・セットのビット・マスクを返します。
InstLineNo	命令のソース行番号を返します。
InstPC	命令のコンパイル時のプログラム・カウンタ (PC) を返します。
IsInstType	命令が指定されたタイプ (ロード命令、ストア命令、条件付き分岐、または無条件分岐) であるかどうかを示します。

9.2.3.4 名前および呼び出しターゲットの解決

Atom のシンボル解決ルーチンにより、atom_application_symbols(5) に記載したように、Atom ツールの計測ルーチンはアプリケーション・オブジェクト、プロシージャ、エントリ・ポイント、命令を探索します。これらは、指定されるか、または呼び出しサイトのターゲットになっています。

- FindObj and FindObjDepthFirst ルーチンは、全オブジェクトの中で指定されたエントリ・ポイントを検索し(それぞれ幅と深さを優先)、指定されたエントリ・ポイントを含むオブジェクトを返します。
- FindProc ルーチンは、メインまたは代替のエントリ・ポイントを含み、オブジェクト内で指定された名前を持つプロシージャを返します。
- FindEntry ルーチンは、オブジェクト内で指定された名前を持つメインまたは代替のエントリ・ポイントを返します。
- FindInst ルーチンは、オブジェクト内で指定された名前を持つ、解決されたシンボルのアドレスにある最初の命令を返します。
- GetTargetName ルーチンは、プロシージャ呼出のターゲットになっているエントリ・ポイントの名前を返します。
- GetTargetObj, GetTargetProc, GetTargetEntry ルーチンは、それぞれ、プロシージャ呼び出しのターゲットを含むオブジェクト、プロシージャ、エントリ・ポイントを返します。
- GetTargetInst ルーチンは、指定した命令がジャンプまたは分岐する先の命令を返します。

9.2.3.5 分析ルーチン呼び出しの追加

atom_application_instrumentation(5) で説明しているように、Atom のアプリケーション・計測ルーチンは、次のように、アプリケーションのさまざまな位置に任意のプロシージャ呼び出しを追加します。

- AddCallProto ルーチンを使用して、プログラムに追加する各分析プロシージャのプロトタイプを指定する必要があります。つまり、AddCallProto 呼び出しは、AddCallProgram, AddCallObj, AddCallProc, AddCallEntry, AddCallBlock, AddCallInst の呼び出しに使用される各分析プロシージャの手続型インタフェースを定義しなければなりません。Atom には、アプリケーションデータと分析データを、定数、レジスタの内容、アドレス解釈構造、計算値(実効アドレスや分岐条件など)として、追加したプロシージャに渡すための機能を備えています。
- プログラムが実行を開始する前または実行を終了した後で、分析プロシージャの呼び出しを追加するには、計測ルーチン内で AddCallProgram ルーチンを使用します。通常、このような分析プロシージャは、出力

ファイルのオープンやコマンド行オプションの解析など、プログラム全体に適用される処理を行います。

- オブジェクトが実行を開始する前または実行を終了した後で、分析プロセスの呼び出しを追加するには、計測ルーチン内で `AddCallObj` ルーチンを使用します。通常、このような分析プロセスは、そのプロセスのデータの初期化など、1つのオブジェクトに適用される処理を行います。
- プロセスが実行を開始する前または実行を終了した後で、分析プロセスの呼び出しを追加するには、計測ルーチン内で `AddCallProc` ルーチンを使用します。
- メインまたは代替エントリ・ポイントが実行を始める前に、分析プロセスの呼び出しを追加するには、計測ルーチン内で `AddCallEntry` ルーチンを使用します。
- 基本ブロックが実行を開始する前または実行を終了した後で、分析プロセスの呼び出しを追加するには、計測ルーチン内で `AddCallBlock` ルーチンを使用します。
- 指定の命令が実行される前または実行された後で、分析プロセスの呼び出しを追加するには、計測ルーチン内で `AddCallInst` ルーチンを使用します。
- 計測機構付きプログラム内のプロセスを置換するには、`ReplaceProcedure` ルーチンを使用します。たとえば、`Third Degree Atom` ツールは、`malloc` や `free` などのメモリ割り当て関数を、独自のバージョンに置換して、誤ったメモリ・アクセスおよびメモリ・リークをチェックできるようにします。

9.2.3.6 エントリ・ポイント呼び出しへの介入

`Atom` の呼び出し介入ルーチンは、`atom_application_instrumentation(5)` で述べているように、メインまたは代替エントリ・ポイントへのアプリケーションの呼び出しに、次のように置換分析ルーチンを呼び出すことで介入します。

- `ReplaceProto` ルーチンを使用して、置換されたエントリ・ポイントの代わりに呼び出される、各置換分析ルーチンのプロトタイプを指定する必要があります。言い換えると、`ReplaceProto` の呼び出しで、`ReplaceEntry` ルーチンへの呼び出しの中で参照される各分析ルーチン

に対するプロシージャのインタフェースを定義していなければなりません。Atom の機能では、アプリケーションデータと分析データを、定数、レジスタの内容、アドレス変換構造体、計算された値 (置換エントリの実行時アドレスなど) のいずれかの形式で、置換分析ルーチンに渡すことができます。

- ReplaceEntry ルーチンを使用して、計測機構付きプログラム内のメインまたは代替エントリ・ポイントを、置換分析ルーチンへの呼び出しに置き換えます。ReplaceEntry の呼び出しは、置き換えるエントリ・ポイント、置換分析ルーチン、プロシージャのインタフェース引数 (ReplaceProto でプロトタイプ化) を指定する必要があります。指定したエントリ・ポイントのみが置き換えられます。同じアプリケーション内の他のエントリ・ポイントは、別の ReplaceEntry 呼び出しによって置き換えることができます。置換ルーチンは分析を実行し、計算済みの ReplAddrValue 値を使用して置換エントリ・ポイントのエミュレートができます。このエミュレーションの例については、9.2.6 項 を参照してください。

たとえば、次のような ReplaceProto と ReplaceEntry の呼び出しのペアは、memcpy(3) のライブラリ呼び出しに介入して、代わりに my_memcpy を、3 つのアプリケーション引数と 2 つの分析引数 (アプリケーションの memcpy() の戻りアドレスと、置き換えたエントリ・ポイントの実行時アドレス) を指定して呼び出します。

```
ReplaceProto ("my_memcpy(VALUE, VALUE, VALUE, REGV, VALUE)");
ReplaceEntry (FindEntry(obj,"memcpy"), /* entry point to replace */
              "my_memcpy", /* replacement analysis routine */
              ArgValue, /* application argument */
              ArgValue, /* application argument */
              ArgValue, /* application argument */
              REG_RA, /* analysis argument */
              ReplAddrValue)); /* analysis argument */
```

関連付けられた置換分析ルーチンは、3 つのアプリケーション引数と 2 つの分析引数で宣言されます。

```
void * my_memcpy (void * s1, const void * s2, size_t n, long call_address,
                  void * (*memcpy_ptr) (void *,const void *,size_t));
```

9.2.4 Atom の記述ファイル

atom_description_file(5) で説明しているように、Atom ツールの記述ファイルは、ツールの計測および分析ファイルを識別して記述します。また、コンパイル、リンク、および起動時に、cc、ld、および atom コマンド

によって使用されるオプションを指定することもできます。各 Atom ツールは、最低 1 つの記述ファイルを提供しなければなりません。

Atom の記述ファイルには、2 つのタイプがあります。

- ツールの一般的な使用のための環境を提供する記述ファイル
ツールが提供できる汎用環境は 1 つだけです。このタイプの記述ファイルの名前は、次の形式になります。

tool.desc

- マルチスレッド・アプリケーションやカーネル・モードなど、特定のコンテキストでツールを使用するための環境を提供する記述ファイル
ツールは、それぞれ独自の記述ファイルを持つ、いくつかの専用環境を提供できます。このタイプの記述ファイルの名前は、次の形式になります。

tool.environment.desc

上記の記述ファイル名で *tool* および *environment* 部分に指定する名前は、ユーザがツールの起動時に `atom` コマンドの `-tool` および `-env` オプションに指定する値に対応します。

Atom の記述ファイルは、一連のタグと値を含むテキスト・ファイルです。ファイルの構文についての詳細は、`atom_description_file(5)` を参照してください。

9.2.5 分析プロシージャの作成

計測機構付きアプリケーションは、分析プロシージャを呼び出して、Atom ツールによって定義された特定の関数を実行します。分析プロシージャは、アプリケーション内部で同じ呼び出し (関数) が計測されている場合でも、システム・コールまたはライブラリ関数を使用できます。分析ルーチンおよび計測機構付きアプリケーションによって使用されるルーチンは、物理的に区別されます。分析ルーチンによる呼び出しが可能および不可能なライブラリ・ルーチンは、次のとおりです。

- 標準 C ライブラリ (`libc.a`) ルーチン (システム・コールを含む) は呼び出すことができますが、次のルーチンを除きます。
 - `unwind(3)` ルーチンおよび他の例外処理ルーチン
 - `tis(3)` ルーチン

また、9.2.5.1 項で説明しているように、標準 I/O ルーチンの動作で異なるものがあります。

- `pthread_atfork(3)` ルーチンは、プログラムの計測の際に `-fork` オプションを使用した場合に限り呼び出すことができます。
- 算術ライブラリ (`libm.a`) ルーチンは呼び出すことができます。
- マルチスレッドまたは例外処理に関するその他のルーチンは呼び出しではなりません (たとえば、`pthread(3)`、`exc_*`、および `libmach` ルーチン)。
- 特定の環境 (たとえば、X や Motif) を想定するその他のルーチンは、Atom 分析環境では、有効でなかったり、正確ではない可能性があります。

TLS (Thread Local Storage) は、分析ルーチンではサポートされていません。

9.2.5.1 入出力

ルーチン分析のために提供されている標準 I/O ライブラリは、計測機構付きプログラムが終了したときに、自動的にストリームのフラッシュおよびクローズを行わないため、すべての出力が完了すると、分析コードは、明示的にそれらをフラッシュおよびクローズする必要があります。また、ルーチン分析のために提供されている `stdout` および `stderr` ストリームは、アプリケーションが `exit()` を呼び出したときにクローズされるため、アプリケーションが終了した後これらのストリームを使用する必要がある場合には、分析コードはそれらのストリームの一方または両方の複製を作成しておく必要があります (`ProgramAfter` または `ObjAfter` 分析ルーチンなど)。入出力のために他のストリームをオープンする方法については、9.1.1 項に記載した `prof` ツールを参照してください。

`stderr` (または `stderr` の複製) への出力が直ちに表示されるようにするため、分析コードは、`setbuf(stream, NULL)` を呼び出してストリームのバッファを解除するか、または `fprintf` 呼び出しの各セットの後に `fflush` を呼び出す必要があります。同様に、C++ ストリームを使用する分析ルーチンは、`cerr.flush()` を呼び出すことができます。

9.2.5.2 fork および exec システム・コール

プロセスが `fork` 関数を呼び出すが `exec` 関数は呼び出さない場合、そのプロセスのクローンが作成され、親の状態の正確なコピーが子に継承されます。多くの場合、Atom ツールはこの動作を予期しています。たとえば、命令アドレス・トレース・ツールは、参照が発生した順序で混合された、親と子の両方の参照を調べます。

命令プロファイル・ツール (たとえば表 9-1 で参照されている `trace` ツール) の場合には、ファイルは `ProgramBefore` 計測ポイントでオープンされ、結果として、出力ファイル記述子は親プロセスと子プロセスの間で共用されます。 `ProgramAfter` 計測ポイントで結果がプリントされる場合、出力ファイルには親のデータが含まれ、その後に子のデータが続きます (親プロセスが最初に終了したと仮定した場合)。

イベントをカウントするツールの場合 (表 9-1 の `prof` ツールなど)、イベントは子ではなく親で発生するため、回数を保持するデータ構造体は、`fork` 呼び出しの後、子プロセスでは 0 に戻さなければなりません。このタイプの Atom ツールは、`fork` ライブラリ・プロシージャを計測し、`fork` ルーチンのリターン値を引数として分析プロシージャを呼び出すことにより、`fork` の正しい処理をサポートできます。分析プロシージャは引数に 0 (ゼロ) のリターン値を渡されると、子プロセスから呼び出されたと分かります。すると、カウント変数またはその他のデータ構造体をリセットして、子プロセスだけの統計情報を集計できるようにします。

9.2.6 置換された呼び出し側アプリケーションのエントリ・ポイント

置換されたエントリ・ポイントは、`ReplaceEntry` の呼び出し中に渡された `ReplAddrValue` パラメータを使用して置換分析ルーチンから呼び出されることがあります。 `ReplAddrValue` 引数には、9.2.3.6 項で述べたような、置換されたエントリ・ポイントのアドレスが含まれています。このアドレスによって、置換分析ルーチンを用いて、呼び出すだけで置換したエントリ・ポイントをエミュレートできるようになります。

次の例では、`memcpy()` 関数は分析関数 `my_memcpy()` に置き換えられ、これはその代わりに、置き換えられた `memcpy()` 関数を呼び出します。

次のソース・リストには、計測コードが含まれています。

```
#include <string.h>
#include <cmplrs/atom.inst.h>
```

```

unsigned InstrumentAll (int argc, char **argv)
{
    Xlate *    px;
    Obj *      o;
    Entry *    e;

    /*
     * Prototype the replacement routine.
     */
    ReplaceProto("my_memcpy(VALUE, VALUE, VALUE, REGV, VALUE)");

    /*
     * Resolve the object that contains memcpy().
     */
    o = FindObj("memcpy");
    if (o) {
        /*
         * Build the object containing memcpy so the memcpy entry point
         * can be resolved and replaced.
         */
        if (BuildObj(o)) return(1);

        /*
         * Resolve the memcpy entry point.
         */
        e = FindEntry(o, "memcpy");

        /*
         * Prefix the memcpy entry point with atom-generated code to
         * call the analysis routine my_memcpy instead.
         */
        ReplaceEntry(e, "my_memcpy",
                    ArgValue, ArgValue, ArgValue, REG_RA, ReplAddrValue);

        /*
         * Write the instrumented object.
         */
        WriteObj(o);
    }
    return (0);
}

```

次のソース・リストには分析コードが含まれています。

```

#include <stdio.h>
#include <stdlib.h>
#include <cmplrs/atom.anal.h>

/*
 * Replacement routine for memcpy();
 */
void * my_memcpy (void * s1, const void * s2, size_t n, long call_address,
                 void * (*memcpy_ptr) (void *, const void *, size_t))
{
    void * ptr = 0;

    /*
     * Report the call.
     */
    printf ("memcpy called from %lx\n", call_address);

    /*
     * Call the original memcpy().
     */
}

```

```

    */
    ptr = (*memcpy_ptr) (s1, s2, n);

    return (ptr);
}

```

9.2.7 分析ルーチンからの計測機構付き PC の決定

Xlate(5) で説明するように、Atom のアドレス変換ルーチンは、選択された命令に対して計測機構付き PC (プログラム・カウンタ) を決定できるようにします。これらの関数を使用して、計測機構付きアプリケーション内の命令の PC を、計測機構のないアプリケーション内の PC に変換するテーブルを作成できます。

命令のアドレスをアドレス変換バッファに渡すために、計測ルーチンは、まず CreateXlate ルーチン呼び出します。アドレス変換バッファが作成された後に、計測ルーチンは、AddXlateAddress ルーチン呼び出すことで命令のアドレスを追加します。エントリ・ポイントのアドレスは、AddXlateEntry ルーチン呼び出すことでアドレス変換バッファに追加することもできます。アドレス変換バッファに保持できるのは、単一オブジェクトからのアドレスだけです。

Atom ツールの計測ルーチンは、AddCallProto 呼び出しでの分析ルーチンのプロトタイプ定義に示されるように、XLATE * タイプのパラメータとして、アドレス変換バッファを分析ルーチンに渡します。

計測機構付き PC を決定するもう 1 つの方法は、分析ルーチンのプロトタイプで REGV の仮パラメータ・タイプを指定して、REG_IPC 値を渡す方法です。

Atom ツールの分析ルーチンは、次の分析インタフェースを使用して、渡されたアドレス変換バッファにアクセスします。

- XlateNum ルーチンは、指定されたアドレス変換バッファ内のアドレスの数を返します。
- XlateInstTextStart ルーチンは、指定されたアドレス変換バッファに対応する計測機構付きオブジェクトのテキスト・セグメントの開始アドレスを返します。
- XlateInstTextSize ルーチンは、テキスト・セグメントのサイズを返します。

- `XlateLoadShift` ルーチンは、指定されたアドレス変換バッファに対応するオブジェクト内の実行時アドレスとコンパイル時アドレスの差を返します。
- `XlateAddr` ルーチンは、指定されたアドレス変換バッファ内の指定された位置にある命令の計測機構付き実行時アドレスを返します。シェアード・ライブラリ内の命令の実行時アドレスは、必ずしもそのコンパイル時アドレスと同じわけではありません。

次の例に、`Xlate` ルーチンを使用するツールの計測および分析ファイルによる `Xlate` ルーチンの使用を示します。このツールは、すべての飛び越し命令のターゲット・アドレスをプリントします。これを使用するには、次のコマンドを実行します。

```
% atom progname xlate.inst.c xlate.anal.c -all
```

次のソース・リスト (`xlate.inst.c`) には、`xlate` ツールの計測が含まれています。

```
#include <stdlib.h>
#include <stdio.h>
#include <alpha/inst.h>
#include <cmplrs/atom.inst.h>

static void      address_add(unsigned long);
static unsigned  address_num(void);
static unsigned long * address_paddr(void);
static void      address_free(void);

void InstrumentInit(int iargc, char **iargv)
{
    /* Create analysis prototypes. */
    AddCallProto("RegisterNumObjs(int)");
    AddCallProto("RegisterXlate(int, XLATE *, long[0])");
    AddCallProto("JmpLog(long, REGV)");

    /* Pass the number of objects to the analysis routines. */
    AddCallProgram(ProgramBefore, "RegisterNumObjs",
        GetProgInfo(ProgNumberObjects));
}

Instrument(int iargc, char **iargv, Obj *obj)
{
    Proc *      p;
    Block *     b;
    Inst *      i;
    Xlate *     pxlt;
    union alpha_instruction bin;
    ProcRes     pres;
    unsigned long pc;
    char        proto[128];

    /*
     * Create an XLATE structure for this Obj. We use this to translate
     * instrumented jump target addresses to pure jump target addresses.
     */

```

```

pxlt = CreateXlate(obj, XLATE_NOSIZE);

for (p = GetFirstObjProc(obj); p; p = GetNextProc(p)) {
    for (b = GetFirstBlock(p); b; b = GetNextBlock(b)) {
        /*
         * If the first instruction in this basic block has had its
         * address taken, it's a potential jump target. Add the
         * instruction to the XLATE and keep track of the pure address
         * too.
         */
        i = GetFirstInst(b);
        if (GetInstInfo(i, InstAddrTaken)) {
            AddXlateAddress(pxlt, i);
            address_add(InstPC(i));
        }

        for (; i; i = GetNextInst(i)) {
            bin.word = GetInstInfo(i, InstBinary);
            if (bin.common.opcode == op_jsr &&
                bin.j_format.function == jsr_jump)
            {
                /*
                 * This is a jump instruction. Instrument it.
                 */
                AddCallInst(i, InstBefore, "JumpLog", InstPC(i),
                    GetInstInfo(i, InstRB));
            }
        }
    }
}

/*
 * Re-prototype the RegisterXlate() analysis routine now that we
 * know the size of the pure address array.
 */
sprintf(proto, "RegisterXlate(int, XLATE *, long[%d]", address_num());
AddCallProto(proto);

/*
 * Pass the XLATE and the pure address array to this object.
 */
AddCallObj(obj, ObjBefore, "RegisterXlate", GetObjInfo(obj, ObjID),
    pxlt, address_paddr());

/*
 * Deallocate the pure address array.
 */
address_free();
}

/*
** Maintains a dynamic array of pure addresses.
*/
static unsigned long * pAddr;
static unsigned      maxAddr = 0;
static unsigned      nAddr = 0;

/*
** Add an address to the array.
*/
static void address_add(
    unsigned long    addr)
{
    /*
     * If there's not enough room, expand the array.

```

```

    */
    if (nAddrs >= maxAddrs) {
        maxAddrs = (nAddrs + 100) * 2;
        pAddrs = realloc(pAddrs, maxAddrs * sizeof(*pAddrs));
        if (!pAddrs) {
            fprintf(stderr, "Out of memory\n");
            exit(1);
        }
    }

    /*
     * Add the address to the array.
     */
    pAddrs[nAddrs++] = addr;
}

/*
** Return the number of elements in the address array.
*/
static unsigned address_num(void)
{
    return(nAddrs);
}

/*
** Return the array of addresses.
*/
static unsigned long *address_paddrs(void)
{
    return(pAddrs);
}

/*
** Deallocate the address array.
*/
static void address_free(void)
{
    free(pAddrs);
    pAddrs = 0;
    maxAddrs = 0;
    nAddrs = 0;
}

```

次のソース・リスト (xlate.anal.c) には, xlate ツールの分析ルーチンが含まれています。

```

#include <stdlib.h>
#include <stdio.h>
#include <cmplrs/atom.anal.h>

/*
 * Each object in the application gets one of the following data
 * structures. The XLATE contains the instrumented addresses for
 * all possible jump targets in the object. The array contains
 * the matching pure addresses.
 */
typedef struct {
    XLATE *          pXlt;
    unsigned long *  pAddrsPure;
} ObjXlt_t;

```

```

/*
 * An array with one ObjXlt_t structure for each object in the
 * application.
 */
static ObjXlt_t *      pAllXlts;
static unsigned        nObj;
static int             translate_addr(unsigned long, unsigned long *);
static int             translate_addr_obj(ObjXlt_t *, unsigned long,
                                         unsigned long *);

/*
** Called at ProgramBefore. Registers the number of objects in
** this application.
*/
void RegisterNumObjs(
    unsigned    nobj)
{
    /*
     * Allocate an array with one element for each object. The
     * elements are initialized as each object is loaded.
     */
    nObj = nobj;
    pAllXlts = calloc(nobj, sizeof(pAllXlts));
    if (!pAllXlts) {
        fprintf(stderr, "Out of Memory\n");
        exit(1);
    }
}

/*
** Called at ObjBefore for each object. Registers an XLATE with
** instrumented addresses for all possible jump targets. Also
** passes an array of pure addresses for all possible jump targets.
*/
void RegisterXlate(
    unsigned        iobj,
    XLATE *         pxlt,
    unsigned long * paddr_pure)
{
    /*
     * Initialize this object's element in the pAllXlts array.
     */
    pAllXlts[iobj].pXlt = pxlt;
    pAllXlts[iobj].pAddrPure = paddr_pure;
}

/*
** Called at InstBefore for each jump instruction. Prints the pure
** target address of the jump.
*/
void JumpLog(
    unsigned long    pc,
    REGV            targ)
{
    unsigned long    addr;

    printf("0x%lx jumps to - ", pc);
    if (translate_addr(targ, &addr))
        printf("0x%lx\n", addr);
    else
        printf("unknown\n");
}

/*
** Attempt to translate the given instrumented address to its pure

```



```

** equivalent. Set '*paddr_pure' to the pure address and return 1
** on success. Return 0 on failure.
**
** Will always succeed for jump target addresses.
*/
static int translate_addr(
    unsigned long    addr_inst,
    unsigned long *   paddr_pure)
{
    unsigned long    start;
    unsigned long    size;
    unsigned          i;

    /*
     * Find out which object contains this instrumented address.
     */
    for (i = 0; i < nObj; i++) {
        start = XlateInstTextStart(pAllXlts[i].pXlt);
        size = XlateInstTextSize(pAllXlts[i].pXlt);
        if (addr_inst >= size && addr_inst < start + size) {
            /*
             * Found the object, translate the address using that
             * object's data.
             */
            return(translate_addr_obj(&pAllXlts[i], addr_inst,
                                     paddr_pure));
        }
    }

    /*
     * No object contains this address.
     */
    return(0);
}

/*
** Attempt to translate the given instrumented address to its
** pure equivalent using the given object's translation data.
** Set '*paddr_pure' to the pure address and return 1 on success.
** Return 0 on failure.
*/
static int translate_addr_obj(
    ObjXlt_t *       pObjXlt,
    unsigned long    addr_inst,
    unsigned long *   paddr_pure)
{
    unsigned          num;
    unsigned          i;

    /*
     * See if the instrumented address matches any element in the XLATE.
     */
    num = XlateNum(pObjXlt->pXlt);
    for (i = 0; i < num; i++) {
        if (XlateAddr(pObjXlt->pXlt, i) == addr_inst) {
            /*
             * Matches this XLATE element, return the matching pure
             * address.
             */
            *paddr_pure = pObjXlt->pAddrsPure[i];
            return(1);
        }
    }
}

/*

```

```

        * No match found, must not be a possible jump target.
        */
    return(0);
}

```

9.2.8 サンプル・ツール

この項では、プロシージャ・トレース、命令プロファイル、データ・キャッシュ・シミュレーションという3つの簡単な例を使用して、基本的なツール作成インタフェースについて説明します。

9.2.8.1 プロシージャ・トレース

ptrace ツールは、プロシージャが実行される順番にその名前をプリントします。実行では、アプリケーションの各プロシージャの呼び出しが追加されます。慣例により、ptrace ツールの計測は、ptrace.inst.c ファイルに入れられます。次の例を参照してください。

```

1  #include <stdio.h>
2  #include <cmplrs/atom.inst.h> [1]
3
4  unsigned InstrumentAll(int argc, char **argv) [2]
5  {
6      Obj *o; Proc *p;
7      AddCallProto("ProcTrace(char *)"); [3]
8      for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) { [4]
9          if (BuildObj(o)) return 1; [5]
10         for (p = GetFirstObjProc(o); p != NULL; p = GetNextProc(p)) { [6]
11             const char *name = ProcName(p); [7]
12             if (name == NULL) name = "UNKNOWN"; [8]
13             AddCallProc(p, ProcBefore, "ProcTrace", name); [9]
14         }
15         WriteObj(o); [10]
16     }
17     return(0);
18 }

```

- [1] Atom 計測ルーチンおよびデータ構造体の定義を取り込みます。
- [2] InstrumentAll プロシージャを定義します。この計測ルーチンは、各分析プロシージャへのインタフェースを定義して、計測するアプリケーションの正しい位置にそれらのプロシージャの呼び出しを挿入します。
- [3] AddCallProto ルーチンを呼び出して、ProcTrace 分析プロシージャを定義します。ProcTrace は、タイプ char * の単一の引数をとります。
- [4] アプリケーション内の各オブジェクトを循環するルーチン GetFirstObj および GetNextObj を呼び出します。プログラムが非共用としてリンクされた場合は、単一のオブジェクトしか存在しません。プログラムが呼び出し共用としてリンクされた場合には、複数のオブジェクト

(メインの実行可能プログラムに 1 つと、動的にリンクされたシェアド・ライブラリごとに 1 つ) を含んでいます。メイン・プログラムは常に最初のオブジェクトです。

- ⑤ 最初のオブジェクトを構築します。オブジェクトは、まず構築しなければ使用できません。非常にまれですが、オブジェクトを構築できないことがあります。InstrumentAll ルーチンは、非ゼロ値を返すことによってこの状態を Atom に報告します。
- ⑥ ルーチン GetFirstObjProc および GetNextProc を呼び出して、アプリケーション・プログラム内の各プロシージャ内を 1 ステップずつ実行します
- ⑦ 各プロシージャに対して、プロシージャ名を探索する ProcName プロシージャを呼び出します。アプリケーションで利用可能なシンボル・テーブル情報の量によって、static として定義されているものなど、いくつかのプロシージャ名が利用できないことがあります。-g1 オプションを指定してアプリケーションをコンパイルすると、このレベルのシンボル情報が提供されます。このような場合、Atom は NULL を返します。
- ⑧ NULL プロシージャ名文字列を UNKNOWN に変換します。
- ⑨ AddCallProc ルーチンを呼び出して、p によってポイントされるプロシージャの呼び出しを追加します。ProcBefore 引数は、分析プロシージャがプロシージャ内の他のすべての命令の前に追加されることを示します。この計測ポイントで呼び出される分析プロシージャ名は、ProcTrace です。最後の引数は分析プロシージャに渡されるものです。この場合は、11 行目で取得されたプロシージャ名です。
- ⑩ 計測機構付きオブジェクト・ファイルをディスクに書き込みます。

計測ファイルは ProcTrace 分析プロシージャの呼び出しを追加しました。このプロシージャは、次の例に示すように、ptrace.anal.c 分析ファイルに定義します。

```
1 #include <stdio.h>
2
3 void ProcTrace(char *name)
4 {
5     fprintf(stderr, "%s\n", name);
6 }
```

ProcTrace 分析プロシージャは、引数として渡された文字列を stderr にプリントします。分析プロシージャは、値を返すことができません。

計測ファイルと分析ファイルを指定すると、ツールは完成します。このツールの応用例を示すために、以下のように、次のアプリケーションをコンパイルおよびリンクします。

```
#include <stdio.h>
main()
{
    printf("Hello world!\n");
}
```

次の例では、共用されない実行可能プログラムを作成して、ptrace ツールを適用し、計測機構付き実行可能プログラムを実行します。この単純なプログラムでは、約 30 個のプロシージャを呼び出します。

```
% cc -non_shared hello.c -o hello
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace
% hello.ptrace
__start
main
printf
_doprnt
__getmbcurmax
strchr
strlen
memcpy
.
.
.
```

次の例は、呼び出し共用としてリンクされたアプリケーションとともにこのプロセスを繰り返します。主な違いは、LD_LIBRARY_PATH 環境変数を現在のディレクトリに設定しなければならないことです。これは、Atom がローカル・ディレクトリに libc.so シェアード・ライブラリの計測機構付きバージョンを作成するためです。

```
% cc hello.c -o hello
% atom hello ptrace.inst.c ptrace.anal.c -o hello.ptrace -all
% setenv LD_LIBRARY_PATH `pwd`
% hello.ptrace
__start
__call_add_gp_range
__exc_add_gp_range
malloc
cartesian_alloc
cartesian_growheap2
__getpagesize
__sbrk
.
.
```

アプリケーションの呼び出し共用バージョンは、非共用バージョンが呼び出すプロシージャ数の約 2 倍を呼び出します。

計測されるのは、元のアプリケーション・プログラム内の呼び出しだけです。ProcTrace 分析プロシージャの呼び出しは元のアプリケーションには存在しなかったため、計測機構付きアプリケーション・プロシージャのトレースには現れません。同様に、各プロシージャ名をプリントする標準ライブラリ呼び出しも含まれません。アプリケーションと分析プログラムがともに `printf` 関数を呼び出す場合、Atom はこの関数の 2 つのコピーを計測機構付きアプリケーションにリンクします。アプリケーション・プログラム内のコピーだけが計測されます。Atom は、複数のエントリ・ポイントを持つプロシージャも正しく計測します。

9.2.8.2 プロファイル・ツール

`iprof` サンプル・ツールは、プログラムが実行する命令数をカウントします。これは、コードのクリティカル・セクションを見つけるのに有用です。アプリケーションが実行されるたびに、`iprof` は、各プロシージャで実行される命令数と各プロシージャの呼び出し回数のプロファイルを含む `iprof.out` というファイルを作成します。

命令数を計算する最も効率的な場所は、各基本ブロックの内部です。基本ブロックが実行されるたびに、一定数の命令が実行されます。次の例は、`iprof` ツールの計測プロシージャ (`iprof.inst.c`) がこれらのタスクをどのように実行するか示したものです。

```
1 #include <stdio.h>
2 #include <cmplrs/atom.inst.h>

3 static int n = 0;
4
5 static const char *      SafeProcName(Proc *);
6
7 void InstrumentInit(int argc, char **argv)
8 {
9     AddCallProto("OpenFile(int)"); [1]
10    AddCallProto("ProcedureCalls(int)");
11    AddCallProto("ProcedureCount(int,int)");
12    AddCallProto("ProcedurePrint(int,char*)");
13    AddCallProto("CloseFile()");
14    AddCallProgram(ProgramAfter,"CloseFile"); [2]
15 }
16
17 Instrument(int argc, char **argv, Obj *obj)
18 {
19     Proc *p; Block *b;
20 }
```

```

21     for (p = GetFirstObjProc(obj); p != NULL; p = GetNextProc(p)) { 3
22         AddCallProc(p, ProcBefore, "ProcedureCalls", n);
23         for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b)) { 4
24             AddCallBlock(b, BlockBefore, "ProcedureCount", 5
25                 n, GetBlockInfo(b, BlockNumberInsts));
26         }
27         AddCallObj(obj, ObjAfter, "ProcedurePrint", n, SafeProcName(p)); 6
28         n++; 7
29     }
30 }
31
32 void InstrumentFini(void)
33 {
34     AddCallProgram(ProgramBefore, "OpenFile", n); 8
35 }
36
37 static const char *SafeProcName(Proc *p)
38 {
39     const char *    name;
40     static char      buf[128];
41
42     name = ProcName(p); 9
43     if (name)
44         return(name);
45     sprintf(buf, "proc_at_0x%lx", ProcPC(p));
46     return(buf);
47 }

```

- 1 分析プロシージャへのインタフェースを定義します。
- 2 CloseFile 分析プロシージャへの呼び出しをプログラムの最後に追加します。
- 3 オブジェクト内の各プロシージャをループします。
- 4 プロシージャ内の各基本ブロックをループします。
- 5 この基本ブロック内の命令が実行される前に、ProcedureCount 分析プロシージャの呼び出しを追加します。ProcedureCount の引数タイプは、11 行目のプロトタイプで定義されます。最初の引数は int タイプのプロシージャ・インデックスであり、2 番目の引数も int で、基本ブロック内の命令の数です。ProcedureCount 分析プロシージャは、基本ブロック内の命令の数をプロシージャごとのデータ構造体に追加します。同様に、ProcedureCalls 分析プロシージャは、各呼び出しが呼び出されたプロシージャの実行を開始する前に、プロシージャの呼び出しカウントに加算します。
- 6 ProcedurePrint 分析プロシージャの呼び出しをプログラムの最後に追加します。ProcedurePrint 分析プロシージャは、このプロシージャの命令の使用と呼び出しカウントを要約した 1 行をプリントします。
- 7 プロシージャ・インデックスを増分します。

- ⑧ OpenFile 分析プロシーダの呼び出しをプログラムの先頭に追加して、アプリケーション内のプロシーダ数を表す `int` をそのプロシーダに渡します。OpenFile プロシーダは、命令を集計するプロシーダごとのデータ構造体を割り当てて、出力ファイルをオープンします。
- ⑨ プロシーダ名を決定します。

iprof ツールによって使用される分析プロシーダは、次の例に示すように、`iprof.anal.c` ファイルで定義されます。

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 long instrTotal = 0;
7 long *instrPerProc;
8 long *callsPerProc;
9
10 FILE *OpenUnique(char *fileName, char *type)
11 {
12     FILE *file;
13     char Name[200];
14
15     if (getenv("ATOMUNIQUE") != NULL)
16         sprintf(Name, "%s.%d", fileName, getpid());
17     else
18         strcpy(Name, fileName);
19
20     file = fopen(Name, type);
21     if (file == NULL)
22     {
23         fprintf(stderr, "Atom: can't open %s for %s\n", Name, type);
24         exit(1);
25     }
26     return(file);
27 }
28
29 static FILE *file;
30 void OpenFile(int number)
31 {
32     file = OpenUnique("iprof.out", "w");
33     fprintf(file, "%30s %15s %15s %12s\n", "Procedure", "Calls",
34         "Instructions", "Percentage");
35     instrPerProc = (long *) calloc(sizeof(long), number); ①
36     callsPerProc = (long *) calloc(sizeof(long), number);
37     if (instrPerProc == NULL || callsPerProc == NULL) {
38         fprintf(stderr, "Malloc failed\n");
39         exit(1);
40     }
41 }
42
43 void ProcedureCalls(int number)
44 {
45     callsPerProc[number]++;
46 }
47
48 void ProcedureCount(int number, int instructions)
49 {
50     instrTotal += instructions;
51     instrPerProc[number] += instructions;
```

```

52 }
53
54
55 void ProcedurePrint(int number, char *name)
56 {
57     if (instrPerProc[number] > 0) { ❷
58         fprintf(file,"%30s %15ld %15ld %12.3f\n",
59             name, callsPerProc[number], instrPerProc[number],
60             100.0 * instrPerProc[number] / instrTotal);
61     }
62 }
63
64 void CloseFile() ❸
65 {
66     fprintf(file,"\n%30s %15s %15ld\n", "Total", "", instrTotal);
67     fclose(file);
68 }

```

- ❶ カウントのデータ構造体を割り当てます。calloc 関数はカウント・データにゼロ詰めします。
- ❷ 呼び出されることのないプロシージャをフィルタにかけます。
- ❸ 出力ファイルをクローズします。分析プロシージャ内でオープンされたファイルは、ツールで明示的にクローズする必要があります。

計測ファイルと分析ファイルを指定すると、ツールは完成します。このツールの応用例を示すために、次のように、"Hello" というアプリケーションをコンパイルしてリンクします。

```

#include <stdio.h>
main()
{
    printf("Hello world!\n");
}

```

次の例は、呼び出し共用実行可能プログラムを作成して、iprof ツールを適用し、計測機構付き実行可能ファイルを実行します。9.2.8.1 項で説明した ptrace ツールとは対照的に、iprof ツールは stdout ではなくファイルにその出力を送ります。

```

% cc hello.c -o hello
% atom hello iprof.inst.c iprof.anal.c -o hello.iprof -all
% setenv LD_LIBRARY_PATH `pwd`
% hello.iprof
Hello world!
% more iprof.out

```

Procedure	Calls	Instructions	Percentage
__start	1	92	1.487
main	1	15	0.242
.			


```

.
.
printf          1          81          0.926
.
.
.

Total          8750
% unsetenv LD_LIBRARY_PATH

```

9.2.8.3 データ・キャッシュ・シミュレーション・ツール

命令およびデータ・アドレスのトレースは、長年、キャッシュ動作を収集して分析するための技術として使用されています。残念なことに、現在のマシン速度ではこれはますます難しくなっています。たとえば、Alvinn SPEC92 ベンチマークは、合計で 2,603,010,614 個の Alpha 命令について、961,082,150 回のロード、260,196,942 回のストア、73,687,356 個の基本ブロックを実行しています。各基本ブロックのアドレスと、すべてのロードとストアの実効アドレスを格納しようとする、10 GB 以上が必要になり、アプリケーションは 100 倍以上遅くなります。

cache ツールは、オン・ザ・フライ (on-the-fly) シミュレーションを使用して、8 KB の直接マップされたキャッシュで実行されるアプリケーションのキャッシュ・ミス率を決定します。次の例は、その計測ルーチンを示したものです。

```

1  #include <stdio.h>
2  #include <cmplrs/atom.inst.h>
3
4  unsigned InstrumentAll(int argc, char **argv)
5  {
6      Obj *o; Proc *p; Block *b; Inst *i;
7
8      AddCallProto("Reference(VALUE)");
9      AddCallProto("Print()");
10     for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {
11         if (BuildObj(o)) return (1);
12         for (p=GetFirstProc(); p != NULL; p = GetNextProc(p)) {
13             for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b)) {
14                 for (i = GetFirstInst(b); i != NULL; i = GetNextInst(i)) { [1]
15                     if (IsInstType(i,InstTypeLoad) || IsInstType(i,InstTypeStore)) {
16                         AddCallInst(i,InstBefore,"Reference",EffAddrValue); [2]
17                     }
18                 }
19             }
20         }
21         WriteObj(o);
22     }
23     AddCallProgram(ProgramAfter,"Print");
24     return (0);

```

```
25 }
```

- ❶ 現在の基本ブロック内の各命令を調べます。
- ❷ 命令がロードまたはストアの場合、Reference 分析プロシージャの呼び出しを追加して、データ参照の実効アドレスを渡します。

cache ツールによって使用される分析プロシージャは、次の例に示すように、`cache.anal.c` ファイルに定義します。

```
1 #include <stdio.h>
2 #include <assert.h>
3 #define CACHE_SIZE 8192
4 #define BLOCK_SHIFT 5
5 long tags[CACHE_SIZE >> BLOCK_SHIFT];
6 long references, misses;
7
8 void Reference(long address) {
9     int index = (address & (CACHE_SIZE-1)) >> BLOCK_SHIFT;
10    long tag = address >> BLOCK_SHIFT;
11    if tags[index] != tag) {
12        misses++;
13        tags[index] = tag;
14    }
15    references++;
16 }
17 void Print() {
18     FILE *file = fopen("cache.out", "w");
19     assert(file != NULL);
20     fprintf(file, "References: %ld\n", references);
21     fprintf(file, "Cache Misses: %ld\n", misses);
22     fprintf(file, "Cache Miss Rate: %f\n", (100.0 * misses) / references);
23     fclose(file);
24 }
```

計測ファイルと分析ファイルを指定すると、ツールは完成します。このツールの応用例を示すために、次のように "Hello" というアプリケーションをコンパイルして、リンクします。

```
#include <stdio.h>
main()
{
    printf("Hello world!\n");
}
```

次の例は、cache ツールを適用して、アプリケーションの非共用バージョンと呼び出し共用バージョンの両方を計測する例です。

```
% cc hello.c -o hello
% atom hello cache.inst.c cache.anal.c -o hello.cache -all
% setenv LD_LIBRARY_PATH `pwd`
% hello.cache
Hello world!
% more cache.out
References: 1091
```

```
Cache Misses: 225
Cache Miss Rate: 20.623281
% cc -non_shared hello.c -o hello
% atom hello cache.inst.c cache.anal.c -o hello.cache -all
% hello.cache
Hello world!
% more cache.out
References: 382
Cache Misses: 93
Cache Miss Rate: 24.345550
```



プログラムの最適化

アプリケーションの最適化には、作成プロセスの変更、ソース・コードの変更、またはその両方を含みます。

多くの場合、アプリケーションを最適化すると、主に実行時性能が改善されます。ただし、アプリケーション・プログラムの実行時性能を計測して、性能の改善方法を分析する前に、次の2つの点について確認してください。

- システムのソフトウェアをチェックして、アプリケーション・プログラムの作成にコンパイラとオペレーティング・システムの最新バージョンを使用していることを確認してください。通常、新バージョンのコンパイラの方がより効果的な最適化を実現し、新バージョンのオペレーティング・システムの方が効率的に動作します。
- アプリケーション・プログラムをテストして、エラーなしで実行できることを確認してください。アプリケーションを32ビット・システムから Tru64 UNIX にポータリングした場合でも、新規に開発した場合でも、アプリケーションを十分にデバッグしてテストするまでは、最適化を行わないでください。C で記述されたアプリケーションをポータリングする際には、C コンパイラの `-message_enable questcode` オプションを用いてコンパイルするか、あるいは、`-Q` オプションを指定して `lint` コマンドを使用することにより、解決する必要のある移植性の問題がないか調べます。

これらの事項について確認してから、最適化処理を開始してください。

アプリケーションの最適化プロセスは、補完的ではあるが、2つの別個のアクティビティに分けることができます。

- アプリケーションの作成プロセスを調整して、たとえば、自動前処理の最適なセットやコンパイル時の最適化を使用するようにします (10.1 節を参照)。
- アプリケーションのソース・コードを分析して、ソース・コードで効果的なアルゴリズムを使用し、性能を低下させる可能性のあるプログラミ

ング言語構造を使用していないことを確認します (10.2 節 を参照)。この手動フェーズには、第 8 章 で説明しているような、性能を分析するプロファイリング・ツールの使用も含まれます。

以降の各節で、チューニング・プロセスのこれら 2 つのアспектに関して詳細に説明します。

10.1 アプリケーション・プログラム作成のガイドライン

アプリケーションの実行時性能の自動的な改善は、作成プロセスのあらゆるフェーズにおいて行うことができます。以降の項で、コンパイル、リンクとロード、前処理と後処理、および、ライブラリ選択において行うことができる主な改善について説明します。特に効果のある手法は、`spike` ツールを用いたプロファイル主導の最適化です (10.1.3 項)。

10.1.1 コンパイルに関する考慮事項

アプリケーションのコンパイルを、設定できる最高の最適化レベル、つまり、最高の性能と正しい結果が得られるレベルで行います。一般に、言語使用の標準に準拠しているアプリケーションは、最高の最適化レベルでコンパイルすることができますが、そのような標準に準拠していないアプリケーションは、低い最適化レベルでコンパイルしなければならないことがあります。詳細については、`cc(1)`、または第 2 章を参照してください。

アプリケーションを最高のレベルでコンパイルできる場合には、すべてのソース・ファイルを 1 回のコンパイルと一緒にコンパイルしてください。複数のソース・ファイルをコンパイルすると、コンパイラが最適化のために調査するコードの量が増加します。これにより、次のような効果が得られます。

- より多くのプロシージャのインライン化
- より完全なデータ・フロー分析
- リンク時に解決される外部参照数の減少

これらの最適化を行うには、コンパイル・オプションの `-ifo` と `-O3` または `-O4` を使用します。

最高レベルの最適化が特定のプログラムで効果があるかどうかを判断するには、プログラムを 2 回コンパイルして、それぞれの結果を比較します。1 回目は、最高の最適化レベルでコンパイルし、2 回目には、1 つ下の最適化レベルでコンパイルします。いくつかのルーチンは最高の最適化レベルで

コンパイルできないことがあるため、そのようなルーチンは別々にコンパイルする必要があります。

実行時性能に重大な影響を及ぼす可能性があるその他のコンパイル時の考慮事項には、次の事柄があります。

- 多数の浮動小数点演算を行う C アプリケーションでは、結果がわずかに異なっても問題ない場合は、`-fp_reorder` オプションの使用を考慮してください。
- C アプリケーションでループ内に多数の `char`、`short`、または `int` データ項目を使用している場合には、C コンパイラの最高レベルの最適化オプションを使用して、性能を改善できます。最高レベルの最適化オプション (`-O4`) は、Alpha システムに対し、とりわけバイトのベクトル化による最適化を実現します。
- 1 回または数回のサンプル実行で性能を調べることができる C および Fortran アプリケーションでは、`-feedback` オプションの使用を検討してください。このオプションは、10.1.3.2 項で説明しているように `-spike` と併用したり、また `-ifo` オプションと併用すると、よりよい結果を得ることができます。
- 完全にデバッグして、例外を生じない C アプリケーションに対しては、`-speculate` オプションの使用を考慮してください。このオプションを使用してコンパイルしたプログラムを実行すると、さまざまな実行パスに関連する値が前もって計算されるため、必要な場合はすぐに使用できます。この先行操作は、アイドル状態のマシン・サイクルを利用します。そのため性能が低下することはなく、通常、前もって計算された値が使用されるたびに向上します。

`-speculate` オプションは、次の 2 つの形式で使用できます。

```
-speculate all
-speculate by_routine
```

どちらのオプションも、例外を破棄します。`-speculate all` オプションは、プログラムのすべてのコンパイル単位で生成された例外を破棄し、`-speculate by_routine` オプションは、そのオプションを適用されたコンパイル単位で生成された例外だけを破棄します。テスト実行で、膨大な数の例外が破棄された場合は、性能が低下します。`-speculate all` オプションは、特に浮小数点演算を行うプログラムに対して、`-speculate by_routine` オプションよりもアグレッシブ

で、より性能を向上させる可能性があります。プログラム内に例外処理を行うルーチンがある場合には、`-speculate all` オプションは使用できません。ただし、`-speculate by_routine` オプションは、そのオプションが使用されるコンパイル単位の外に例外処理がある場合には、使用することができます。デバッグの最中は、いずれの `-speculate` オプションも使用しないでください。

プログラムが正常終了した場合に、破棄された例外数のカウントをプリントするには、次の環境変数を指定します。

```
% setenv _SPECULATE_ARGS -stats
```

現在、`-speculate all` オプションでは統計機能は使用できません。

`-speculate all` および `-speculate by_routine` の各オプションを使用すると、境界合わせのフィックスアップについてのメッセージがすべて抑制されます。テストまたは本番の境界合わせフィックスアップに関するメッセージを生成するには、次の環境変数を指定します。

```
% setenv _SPECULATE_ARGS -alignmsg
```

両方のオプションを指定するには、次のようにします。

```
% setenv _SPECULATE_ARGS -stats -alignmsg
```

- 次のコンパイル・オプションと一緒に、または個別に使用することにより、実行時性能が改善できます (詳細は、`cc(1)` を参照してください)。

オプション	説明
<code>-arch</code>	命令を生成する Alpha アーキテクチャのバージョンを指定する。 <code>-arch</code> と <code>-tune</code> の違いについては、 <code>cc(1)</code> の <code>-arch</code> を参照。
<code>-ansi_alias</code>	ソース・コードが ANSI C の別名化規則に準拠しているかどうかを指定する。ANSI C 別名化規則では、よりアグレッシブな最適化が許可されている。
<code>-ansi_args</code>	ソース・コードが引数に関して ANSI C の規則に準拠しているかどうかを指定する。ANSI C の規則に準拠している場合には、特別な引数クリーニング・コードを生成する必要がない。

オプション	説明
-fast	性能向上のため、次の最適化オプションをオンにする。 <div> -ansi_alias -ansi_args -assume trusted_short_alignment -D_FASTMATH -float -fp_reorder -ifo -D_INLINE_INTRINSICS -D_INTRINSICS -intrinsics -O3 -readonly_strings </div>
-feedback	最適化の際に、コンパイラが指定されたファイル内のプロファイル情報を使用することを指定する。詳細については、10.1.3.2 項を参照。
-fp_reorder	浮動小数点演算に影響する特定のコード変換が許可されるかどうかを指定する。
-G	スモール・データ・セクション(sbss または sdata) におけるデータ項目の最大バイト・サイズを指定する。
-inline	関数のインライン展開を実行するかどうかを指定する。
-ifo	ファイルが別々にコンパイルされた場合、可能な限り、ファイル境界を超えて、最適化の向上 (ファイル間の最適化) とコード生成を行う。
-O	コンパイルにより到達可能な最適化レベルを指定する。
-om	さまざまなポストリンク・コード最適化を実行する。-non_shared オプションを用いてコンパイルしたプログラムで、最も効果がある (付録 F を参照)。このオプションは、-spike オプションに置き換えられている (10.1.3 項を参照)。
-preempt_module	モジュールごとのシンボルの優先使用をサポートする。
-speculate	実行パスが取られる前に、その実行パスで実行中のプログラム内で動作 (たとえば、ロードや演算操作) が行われるようにする。
-spike	さまざまなポストリンク・コード最適化を実行する (10.1.3 項を参照)。

オプション	説明
-tune	Alpha アーキテクチャ固有の処理系に対して、プロセッサ固有の命令のチューニングを選択する。-tune と -arch の違いについては、cc(1) の -arch を参照。
-unroll	レベル -O2 以上の最適化プログラムで行われるループの展開を制御する

上記のオプションを使用すると、正確さと標準に対する準拠が損なわれることがあります。

- C アプリケーションでは、浮動小数点の例外処理に有効なコンパイル・オプションは、次のように実行時間に重大な影響を及ぼすことがあります。
 - 省略時の例外処理 (特別なコンパイル・オプションはない)

省略時の例外処理モードでは、オーバフロー、ゼロによる除算、および無効時処理の例外で、必ず SIGFPE 例外ハンドラがシグナル通知されます。また、IEEE 無限大、IEEE NaN (not-a-number)、または IEEE 非正規化数のいずれかを使用すると、SIGFPE 例外ハンドラがシグナル通知されます。省略時の設定では、アンダフローはサイレントに結果がゼロになりますが、コンパイラは、アンダフローで SIGFPE ハンドラがシグナル通知されるようにする別のオプションをサポートしています。

省略時の例外処理モードは、特定の浮動小数点フォーマットの特殊な特性に依存していない移植可能なプログラムに適しています。省略時のモードで最適な例外処理性能が提供されます。
 - 移植可能な **IEEE** 例外処理 (-ieee)

移植可能な IEEE 例外処理モードでは、特別な呼び出しを行って、フォールトを可能にしておかない限り、浮動小数点例外はシグナル通知されません。このモードは、IEEE 無限大、IEEE NaN、および IEEE 非正規化数を正確に生じさせて、処理します。また、このモードは IEEE 浮動小数点のほとんどの移植不可能なアспектに対して次のようなサポートを提供しています。つまり、厳密でない例外を除き、すべての状態オプションとトラップ使用をサポートします。厳密でない例外機能 (-ieee_with_inexact) についての詳細は、ieee(3) を参照してください。この機能を使用すると、

プログラムがこの機能を必要とする場合に、浮動小数点の計算の速度が約 100 ファクタ以上遅くなります。

移植可能な IEEE 例外処理モードは、IEEE 浮動小数点規格の移植可能なアスペクトに依存しているプログラムに適しています。このモードでは、通常、プログラム内の浮動小数点のコンパイル量に応じて、省略時のモードより 10 ~ 20 パーセント処理速度が遅くなります。状況によっては、このモードで実行時間を 3 ファクタ以上速くすることができます。

10.1.2 リンクおよびロードに関する考慮事項

アプリケーションで多数の大きなライブラリを使用しない場合には、それを共用しないでリンクすることを考慮してください。このようにすると、リンクがライブラリへの呼び出しを最適化できるため、(呼び出しが頻繁に行われる場合には) アプリケーションの起動時間が短縮されて、実行時性能が改善されます。ただし、共用されないアプリケーションは呼び出し共用のアプリケーションより多くのシステム資源を使用することがあります。多数のアプリケーションを同時に実行しているとき、アプリケーションに共通なライブラリのセットがある (たとえば、libX11 や libc) 場合には、それらのライブラリを呼び出し共用としてリンクすると、システム全体の性能を向上させることができます。詳細については、第 4 章を参照してください。

シェアード・ライブラリを使用するアプリケーションでは、それらのライブラリがクイックスタートできることを確認してください。クイックスタートは、アプリケーションのロード時間を大幅に減少させる Tru64 UNIX の機能です。アプリケーションの数が多い場合には、アプリケーションの起動と実行に必要な全時間のうち、かなりの割合をロード時間が占めます。オブジェクトをクイックスタートできない場合には、実行はできますが、起動時間が遅くなります。詳細については、4.7 節を参照してください。

10.1.3 spike およびプロファイル主導の最適化

この項では、ポストリンク最適化プログラム spike について説明します。

10.1.3.1 spike の概要

spike ツールは、リンク後のコードの最適化を行います。プログラム全体を操作できるため、spike ではコンパイラにはできない最適化を行うことがで

きます。 10.1.3.2 項 で述べているように、`spike` は、最適化を先導するプロファイル情報を使用すると、最も効果があります。

`spike` は Tru64 UNIX Version 5.1 から提供されるようになったツールであり、`om` および `cord` に代わるものです。制御や最適化の効率が改善されており、実行可能プログラムとシェアード・ライブラリのどちらでも使用できます。`spike` は、`om` や `cord` とは併用できません。`om` および `cord` についての詳細は、付録 F を参照してください。

`spike` が実行する最適化には、コード・レイアウト、実行されないコードの削除、アドレス計算の最適化などがあります。

`spike` は、Tru64 UNIX の V4.0 以降のシステムでリンクされたバイナリを処理できます。V5.1 以降のシステムでリンクされたバイナリには、`spike` がさらに最適化を行えるようにする情報が含まれています。

注意

`spike` は、Tru64 UNIX V5.1 以降のイメージに対して一部のアドレス最適化のみを行います。 `om` は V4 イメージに対して最適化を行います。V5.1 より前のバイナリに対して `spike` を使用し、リンクの最適化を有効にしている (リンクの段階で `cc` に `-O` を渡す) 場合、`om` と `spike` の性能の違いはありません。

`spike` は、2 通りの方法で使用できます。

- コンパイルの後に、`spike` コマンドをバイナリ・ファイルに適用する方法
- `cc` コマンド (または、対応するコンパイラがシステムにインストールされている場合は、`cxx`、`f77`、`f90` のいずれか) に `-spike` オプションを付けて、コンパイル処理の一部として使用する方法

この項と 10.1.3.2 項 の例は、`spike` のこの 2 つの使用方法を示しています。実行可能ファイルを再リンクしたくない場合 (Example 1) や、コンパイルの後にプロファイル情報を使用する場合 (Example 5 および Example 6) は、`spike` コマンドが便利です。プロファイル情報を使用しない場合 (Example 2) や、コンパイラでもプロファイル情報を使用する場合 (Example 3 および Example 4) は、`-spike` オプションの方が便利です。

Example 1 および Example 2 は、最適化を先導するプロファイル情報を使用せずに `spike` を使用する方法を示しています。10.1.3.2 項では、`pixie` プロファイラからのフィードバック情報を用いて `spike` を使用する方法を示しています。

例 1

この例では、`spike` はバイナリ `my_prog` に適用され、最適化された出力ファイル `prog1.opt` を生成します。

```
% spike my_prog -o prog1.opt
```

例 2

この例では、`cc` コマンドの `-spike` オプションを使って、コンパイル時に `spike` を適用しています。

```
% cc -c file1.c
% cc -o prog3 file1.o -spike
```

最初のコマンド行は、オブジェクト・ファイル `file1.o` を作成します。2 番目のコマンド行は `file1.o` をリンクして実行可能ファイルを作成し、`spike` を用いて実行可能ファイルを最適化します。

`spike` コマンドのオプションはすべて、`(cc) -WS` オプションを用いて、`cc` コマンドの `-spike` オプションに直接渡すことができます。次の例に構文を示します。

```
% cc -spike -feedback prog -o prog *.c \
    -WS,-splitThresh,.999,-noaggressiveAlign
```

`spike` コマンドのオプションと `spike` を使用する際の制限についての詳細は、`spike(1)` を参照してください。

10.1.3.2 プロファイル主導の最適化での `spike` の使用

自動最適化は、前項で述べた `-O`、`-fast`、`-inline` などのコンパイラの自動最適化オプションを使用すると、ある程度までは達成できます。これらのオプションは、CPU アーキテクチャとキャッシュ・メモリを最善の方法で使用するための最小限の命令シーケンスを生成する際に役立ちます。

ただし、プログラムが通常の入力データおよび通常的环境のもとで実行された場合に、どの命令が最も多く実行されるかという情報を指定すると、コンパイラとリンカは、この最適化を改善できます。Tru64 UNIX では、プロ

ファイラの結果を再コンパイルにフィードバックすることで、この情報を指定できます。このカスタマイズされたプロファイル主導の最適化は、自動最適化と組み合わせて使用できます。

以下の例では、spike を pixie プロファイラと併用し、さまざまなフィードバック手法を用いてプログラムから生成される命令シーケンスを調整する方法を示しています。

例 3

この例は、spike を用いたプロファイル主導最適化の基本的な 3 つの手順、すなわち、(1) プログラムの最適化を準備する、(2) 計測機構付きのプログラムを作成して実行し、プロファイリング統計情報を収集する、(3) その情報をコンパイラとリンカにフィードバックして、実行可能コードの最適化に役立てる、という手順を示しています。後の例では、これらの手順を詳細化して、開発中に行われた変更と複数のプロファイリング実行で得たデータを合わせて調節する方法を示しています。

```
% cc -feedback prog -o prog -O3 *.c [1]
% pixie -update prog [2]
% cc -feedback prog -o prog -spike -O3 *.c [3]
```

- [1]** プログラムが最初に `-feedback` オプション付きでコンパイルされたときに、拡張された特別な実行可能ファイルが作成されます。これには、コンパイラが実行可能ファイルをソース・ファイルに対応させるために使用する情報が含まれています。また、コンパイラへのプロファイルのフィードバック情報を格納するために後で使用するセクションも含まれています。このセクションは、最初にコンパイルしたときは、pixie プロファイラ (手順 2) がフィードバック情報をまだ生成していないため、空のままです。`-feedback` オプションに指定するファイル名は、必ず実行可能ファイルと同じ名前にしてください。この例では `prog` (`-o` オプションで指定) です。特に指定しないかぎり、`-feedback` オプションでは `-g1` オプションが適用され、プロファイルに最適なシンボルが付けられます。`-On` オプションを試して、対象のプログラムとコンパイラで実行時性能が最高になる最適化のレベルを調べてください。最初のコンパイルの際には、まだフィードバック情報がないので、コンパイラは次のメッセージを出力します。

```
cc: Info: Feedback file prog does not exist (nofbfil)
cc: Info: Compilation will proceed without feedback optimizations (nofbopt)
```

- ② `pixie` コマンドは、計測機構付きのプログラム (`prog.pixie`) を作成して、それを実行します (`prof` オプション、`-update` が指定されているため)。実行の統計情報とアドレスのマッピング・データは自動的に命令カウント・ファイル (`prog.Counts`) と命令アドレス・ファイル (`prog.Addrs`) に収集されます。`-update` オプションにより、このプロファイル情報は拡張された実行可能ファイルに格納されます。
- ③ `-feedback` オプションを指定した 2 度目のコンパイルでは、拡張された実行可能ファイル内のプロファイル情報がコンパイラと (`-spike` オプションを通じて) ポストリンク最適化プログラムを先導します。このカスタマイズされたフィードバックは、`-O3` および `-spike` オプションによる自動最適化よりも優れています。コンパイラの最適化は、`-ifo` オプションや `-assume whole_program` オプションを `-feedback` オプションと組み合わせて使用すると、さらに効果が上がります。ただし、10.1.1 項で述べているように、大きいプログラムはソース・ファイルが 1 つしかない場合と同じようにコンパイルすることができないことがあります。

詳細は、`pixie(1)` および `cc(1)` を参照してください。

拡張された実行可能ファイルは、内部にプロファイル情報があるため、通常の実行可能ファイルよりも大きくなります (通常は 3 ~ 5 パーセント)。開発が完了したら、`strip` コマンドを用いてプロファイル情報とシンボル・テーブル情報を削除できます。たとえば次のようにします。

```
% strip prog
```

`spike` コマンドは、`strip` を実行したイメージは処理できません。

例 4

一般的な開発過程では、Example 3 の手順 2 と 3 を必要な回数だけ繰り返して、ソース・コードの変更による影響が反映されるようにします。たとえば次のようにします。

```
% cc -feedback prog -o prog -O3 *.c
% pixie -update prog
% cc -feedback prog -o prog -O3 *.c
[modify source code]
% cc -feedback prog -o prog -O3 *.c
.....
[modify source code]
% cc -feedback prog -o prog -O3 *.c
% pixie -update prog
% cc -feedback prog -o prog -spike -O3 *.c
```

拡張された実行可能ファイル内のプロファイル情報は、コンパイル操作では失われないので、情報を更新する `pixie` の処理手順は、ソース・モジュールが変更されて再コンパイルされるたびに繰り返す必要はありません。ただし、変更のたびに、変更内容に応じて、実際のコードと古いフィードバック情報の違いが大きくなるため、最適化の有効度が低下します。最後の変更および再コンパイルの後に `pixie` 処理手順を行うと、最後に行ったコンパイルに合わせてフィードバック情報が正確に更新された状態になります。

例 5

プロファイルの正確な情報を得るために、計測機構付きプログラムを異なる入力で複数回実行したい場合があります。この例では、プログラム `prog` の実行を 2 回計測して、そのプロファイル統計情報をマージすることでプログラムを最適化する方法を示しています。このプログラムの出力は、異なる入力で実行するたびに異なります。

```
% cc -feedback prog -o prog *.c[1]
% pixie -pids prog[2]
% prog.pixie[3]
(input set 1)
% prog.pixie
(input set 2)
% prof -pixie -update prog prog.Counts.*[4]
% spike prog -feedback prog -o prog.opt[5]
```

- [1] 最初のコンパイルでは、Example 3 で説明したように拡張された実行可能ファイルが生成されます。
- [2] 特に指定しなければ、計測機構付きプログラム (`prog.pixie`) を実行するたびに、`prog.Counts` という名前のプロファイル・データ・ファイルが作成されます。 `-pids` オプションを指定すると、計測機構付きプログラムを実行するたびに、作成されるプロファイル・データ・ファイルの名前にプロセス ID が付加されます (`prog.Counts.pid`)。したがって、後の実行で生成されるデータ・ファイルで上書きされることはありません。
- [3] 計測機構付きプログラムは 2 回実行され、そのたびに一意の名前でデータ・ファイルが生成されます。たとえば、`prog.Counts.371` と `prog.Counts.422` のようになります。
- [4] `prof -pixie` コマンドは、2 つのデータ・ファイルをマージします。 `-update` オプションにより、この結合された情報で実行可能ファイル `prog` が更新されます。

- ⑤ `-feedback` オプション付きの `spike` コマンドは、最適化を先導するために 2 回のプログラム実行で得られたプロファイル情報を結合したものを使用し、最適化された出力ファイル `prog.opt` を生成します。

この例の最後の手順は、次のように変更できます。

```
% cc -spike -feedback prog -o prog -O3 *.c
```

`-spike` オプションでは、プログラムを再リンクする必要があります。
`spike` コマンドを使用した場合は、`spike` を実行するためにプログラムをリンクし直す必要はありません。

例 6

この例は、通常の (拡張されていない) 実行可能ファイルが作成され、`spike` コマンドの `-fb` オプションが使用される (`-feedback` オプションではない) 点で、Example 5 と異なります。

```
% cc prog -o prog *.c
% pixie -pids prog
% prog.pixie
(input set 1)
% prog.pixie
(input set 2)
% prof -pixie -merge prog.Counts prog prog.Addrs prog.Counts.*
% spike prog -fb prog -o prog.opt
```

`prof -pixie -merge` コマンドは、2 回の計測実行で得られた 2 つのデータ・ファイルを 1 つの `prog.Counts` ファイルにマージします。この形式のフィードバックでは、`-g1` オプションを明示的に指定して、プロファイリングに最適なシンボルを付ける必要があります。

`spike -fb` コマンドは `prog.Addrs` および `prog.Counts` の情報を使用して、最適化された出力ファイル `prog.opt` を生成します。

Example 5 の方法をお勧めします。Example 6 の方法は互換性のためにサポートされているので、実行可能ファイルに格納されたフィードバック情報を使用する `-feedback` オプションを指定してコンパイルすることができない場合にのみ使用してください。

10.1.4 前処理と後処理に関する考慮事項

性能に影響を及ぼす前処理オプションと後処理 (実行時) オプションには、次のようなものがあります。

- Kuck & Associates Preprocessor (KAP) ツールを使用して、さらに最適化を行うことができます。プリプロセッサは最終的なソース・コードを入力として使用し、最適化されたソース・コードを出力として生成します。

KAP は、シンメトリック・マルチプロセッシング・システム (SMP) と単一プロセッサ・システムの両方で実行される次のような特性を持つアプリケーションに対して特に有効です。

- ループが多数含まれるプログラムや大きなループ結合のあるループ
- 大きなデータ・セットに対して動作するプログラム
- かなりのデータを再使用するプログラム
- 多数のプロシージャを呼び出すプログラム
- 多数の浮動小数点演算を行うプログラム

SMP システムの並列処理機能を利用するため、KAP プリプロセッサは C プログラムに対して自動および指示による分解をサポートしています。KAP の自動分解機能では、既存のプログラムを分析して、並列実行の候補となるループを探索します。その後、ループを分解して、必要なすべての同期ポイントへ挿入します。さらに制御が必要な場合には、プログラマが手操作で指示文を挿入して、個々のループの並列化を制御することができます。Tru64 UNIX システムでは、KAP は POSIX Threads Library を使用して、並列処理を実現します。

C プログラムでは、KAP は `kapc` コマンド (単独の KAP 処理を起動する) または `kcc` (KAP 処理と HP C コンパイルを組み合わせで起動する) で起動します。C プログラムで KAP を使用方法については、『KAP for C for Tru64 UNIX』を参照してください。

Tru64 UNIX システムでは、KAP は別注文のレイヤード・プロダクトとして入手可能です。

- 特にプロファイル主導のフィードバックでは、ポストリンク最適化とプロシージャの再編成のために次のツールを使用します。
 - `spike` (10.1.3 項を参照)
 - `om` (付録 F を参照)
 - `cord` (付録 F を参照)

10.1.5 ライブラリ・ルーチンの選択

性能に影響を及ぼすライブラリ・ルーチンのオプションには、次のものがあります。

- 数値演算を集中して行うアプリケーションに対しては、CXML (Compaq Extended Math Library, 以前の DXML: Digital Extended Math Library) を使用します。CXML は Alpha システム (SMP システムと単一プロセッサ・システムの両方) 用に最適化された算術ルーチンの集まりです。CXML に格納されているルーチンは、次の 4 つのライブラリに編成できます。

- BLAS

基本的な線形代数サブルーチンで構成されるライブラリ。

- LAPACK

線形システムと固有システムの問題を解決するための線形代数のパッケージ。

- スパース線形システム・ソルバ

直接的な反復型スパース・ソルバのライブラリ。

- シグナル処理

1 次元、2 次元、および 3 次元の高速フーリエ変換 (FFT)、グループ FFT、正弦/余弦変換、重畳関数、相関関数、およびデジタル・フィルタを含むシグナル処理関数の基本セット。

CXML を使用することにより、数値演算を集中して行うアプリケーションは Tru64 UNIX システム上での実行速度がかなり向上します。KAP と一緒に使用している場合は特に向上します。CXML ルーチンは、ユーザのプログラムから明示的に呼び出すことができ、場合によっては、KAP から (つまり、KAP が CXML ルーチンを使用できると認識した場合) 呼び出すことができます。コンパイル・コマンド行に `-ldxml` オプションを指定して、CXML にアクセスします。

CXML についての詳細は、『Compaq Extended Math Library Reference Guide』を参照してください。

CXML ルーチンは Fortran で書かれています。C プログラムから Fortran ルーチンを呼び出す方法については、Tru64 UNIX についての Compaq Fortran (以前の Digital Fortran) のユーザ・マニュアルを

参照してください。C プログラムから CXML ルーチンを呼び出す方法については、『TechAdvantage C/C++ Getting Started Guide』にも説明があります。

- アプリケーションで拡張精度が必要ない場合には、精度は多少落ちますが、実行速度の速い算術ライブラリ・ルーチンを使用することができます。コンパイル・コマンド行で `-D_FASTMATH` オプションを指定すると、コンパイラは、浮動小数点精度の 3 ビットを犠牲にして、実行速度の速い浮動小数点ルーチンを使用するようになります。詳細については、`cc(1)` を参照してください。
- `-D_INTRINSICS` および `-D_INLINE_INTRINSICS` オプションを使用して、C プログラムをコンパイルすることを検討してください。このようにすると、コンパイラは特定の標準 C のライブラリ・ルーチンへの呼び出しをインライン化します。

10.2 アプリケーションのコーディング上のガイドライン

アプリケーションを修正したい場合には、プロファイラ・ツールを使用して、アプリケーションの実行に最も時間がかかっている部分を判断します。多くのアプリケーションでは、実行時間のほとんどは特定のルーチンで費やされています。このような頻繁に使用されているルーチンについて、特に改善の努力を注ぐようにしてください。

Tru64 UNIX では、C および他の言語で作成されたプログラムに対して動作するいくつかのプロファイリング・ツールを提供しています。詳細については、第 7 章、第 8 章、第 9 章、`prof_intro(1)`、`gprof(1)`、`hiprof(1)`、`pixie(1)`、`prof(1)`、`third(1)`、`uprofile(1)` および `atom(1)` を参照してください。

アプリケーションで頻繁に使用されているルーチンが識別できたならば、そのコードが使用しているアルゴリズムについて検討してください。より効率的なアルゴリズムと置換が可能なものはありますか。処理速度の遅いアルゴリズムは、既存のアルゴリズムに手を加えるより、処理速度の速いものと置換する方が、性能が大幅に改善されることがよくあります。

アルゴリズムの効率に問題がない場合は、コードを変更して、アプリケーションのオブジェクト・コードを生成するコンパイラが最適化を行いやすくすることを検討してください。コンパイラによる最適化が最大限に行われるようなソース・コードの記述方法については、『High Performance Computing』

(Kevin Dowd 著, O'Reilly & Associates, Inc., ISBN 1-56592-032-5) に、一般的な説明があります。

以降の各項で、性能改善について考慮の対象となるデータ型、入出力処理、キャッシュ使用とデータの境界合わせ、および言語固有の問題について説明します。

10.2.1 データ型についての考慮事項

性能に影響を及ぼすデータ型に関する考慮事項には、次のものがあります。

- Alpha システム上で効率的なアクセスを行う最小単位は 32 ビットです。32 ビットまたは 64 ビットのデータ項目は、単一の効率的な機械語命令でアクセスすることが可能です。Alpha アーキテクチャの旧処理系 (プロセッサが EV56 より前のもの) でアプリケーションの性能が重要である場合には、次の点を考慮してください。
 - 特に頻繁に使用されるスカラに対しては、32 ビットより小さい整数および論理データ型の使用は避けてください。
 - C プログラムでは、`char` および `short` 宣言を、`int` および `long` 宣言で置換することを検討してください。
- 整数値の除算は、浮動小数点数値の除算より、演算速度が遅くなります。可能な場合には、そのような整数値演算を、浮動小数点演算と置換することを考慮してください。

整数の除算演算は Alpha プロセッサにネイティブなものではなく、ソフトウェアでエミュレートする必要があるため、演算速度が遅くなります。その他のネイティブでない演算には、超越演算 (たとえば、正弦と余弦) と平方根があります。

10.2.2 AdvFS ファイルでの直接入出力の使用

直接入出力を使用すると、ファイル管理、オンライン・バックアップ、およびオンライン・リカバリなどの AdvFS (Advanced File System) が提供するファイル・システム機能を利用できるようになります。しかも、ユーザ・データを AdvFS キャッシュにコピーすることによるオーバーヘッドはありません。直接入出力では、直接メモリ・アクセス (DMA) コマンドを使用して、アプリケーションのバッファとディスク間でユーザ・データを直接コピーします。

通常のファイル・システム入出力では、ファイル・ページをキャッシュ内に保持します。これにより、入出力が非同期に完了します。いったんデータがキャッシュ内に蓄えられて、入出力のスケジュールが設定されると、アプリケーションはデータがディスクに転送されるのを待つ必要はありません。さらに、データが既にキャッシュ内にあるため、その後このページにアクセスする際には、ディスクからデータを読み込む必要はありません。ほとんどのアプリケーションは通常のファイル・システム入出力を使用します。

通常のファイル・システム入出力は、ディスク上のデータに稀にしかアクセスせずに、スレッド間の競合を管理するアプリケーションには適しません。この種のアプリケーションでは、直接入出力による低いオーバーヘッドの利点を利用できます。ただし、データはキャッシュされないため、指定されたページへのアクセスを、競合するスレッド間で直列化する必要があります。これを行うため、省略時の設定では、直接入出力は同期入出力を実行します。これは、`read()` ルーチンがアプリケーションに制御を戻す時点で、入出力が完了しており、データはディスク上にあることを意味します。その後、そのデータを取得する場合は、常にディスクからデータを取得するための入出力操作が行われます。

アプリケーションは、非同期入出力 (AIO) を利用することもできますが、その場合でも `aio_read()` および `aio_write()` システム・ルーチンを使用することにより、基礎となる直接入出力のメカニズムを使用しています。これらのルーチンは、データがディスクに転送される前にアプリケーションに制御を戻します。また `aio_error()` ルーチンは、アプリケーションが入出力の完了に対してポーリングを行うことを可能にします (カーネルは、ファイル・ページへのアクセスの同期処理を行って、2 つのスレッドが同時に同じページへ書き込みを行わないようにします)。

直接入出力を使用して指定されたファイルへアクセスする複数のスレッドは、同じページ範囲にアクセスしなければ、その作業を同時に行うことができます。たとえば、スレッド A がページ 10 から 19 までに書き込みを行い、スレッド B がページ 20 から 39 までに書き込みを行う場合、これらの操作は同時に実行されます。これに続いて、スレッド B がページ 15 から 39 まで単一の直接入出力転送で書き込みを行う場合には、ページ範囲が重複しているため、スレッド A の書き込みが完了するまでスレッド B は待機させられます。

直接入出力を使用するときには、要求された転送がディスク・セクタ境界に合っていて、転送サイズが基本セクタ・サイズの偶数倍である場合に、最高

の性能が得られます。最適な転送サイズはデータ格納用のハードウェアに依存しますが、一般に転送サイズが大きいほど、転送効率は向上します。

注意

直接入出力モードとマップされたファイル領域 (mmap) の使用は、排他的な操作です。マップされたファイル領域を使用するファイルに対して、直接入出力モードを設定することはできません。直接入出力用に既にオープンされているファイルに対してマッピングを行う場合も、操作は失敗します。

直接入出力およびアトミック・データ・ロギング・モードも、相互に排他的です。ファイルがこれらのいずれかのモードでオープンされている場合に、同じファイルを他のモードでオープンしようとすると、操作は失敗します。

AIO アプリケーションおよび非 AIO アプリケーションの両方について、直接入出力機能を有効にして、AdvFS ファイルで使用できます。この機能を利用できるようにするには、`O_DIRECTIO` ファイル・アクセス・フラグを設定し、アプリケーションで `open` 関数を使用しています。次に例を示します。

```
open ("file", O_DIRECTIO | O_RDWR, 0644)
```

直接入出力モードは、すべてのユーザによりファイルがクローズされるまで有効です。

`fcntl()` 関数にパラメータ `F_GETCACHPOLICY` を指定して使用すると、ファイルのキャッシング・ポリシー (FCACHE または FDIRECTIO モード) を返すことができます。次に例を示します。

```
int fcntlarg = 0;
ret = fcntl( filedescriptor, F_GETCACHPOLICY, &fcntlarg );
if ( ret != -1 && fcntlarg == FDIRECTIO ) {
    .
    .
    .
}
```

直接入出力および AdvFS の使用法についての詳細は、`fcntl(2)` および `open(2)` を参照してください。

10.2.3 キャッシュ使用とデータの境界合わせに関する考慮事項

キャッシュ使用パターンは、次のように、性能にクリティカルな影響を及ぼします。

- アプリケーションにいくつか頻繁に使用されるデータ構造体がある場合は、そのデータ構造体を 2 次キャッシュ内のキャッシュ・ライン境界に割り当てるようにしてください。このようにすることにより、ユーザのアプリケーションがキャッシュを効率的に使用するように改善することができます。詳細については、『*Alpha Architecture Reference Manual*』を参照してください。
- 頻繁に使用されるデータ構造体間の潜在的なデータ・キャッシュ衝突を捜してください。メモリ内に割り当てられている 2 つのデータ構造体間の距離が一次 (内部) データ・キャッシュのサイズに等しい場合、このような衝突が起こります。データ構造体が小さい場合は、それらをメモリ内に連続して割り当てることにより、これを回避することができます。uprofile ツールを使用すると、キャッシュ衝突数とその場所を判断することができます。データ・キャッシュ衝突についての詳細は、『*Alpha Architecture Reference Manual*』を参照してください。

データの境界合わせも性能に影響を及ぼします。省略時の設定では、C コンパイラは、各データ項目を自然境界に合わせます。つまり、各データ項目を、その開始アドレスが、宣言に使用されたデータ型のサイズの倍数になるように位置付けます。自然境界に位置合わせされていないデータを境界合わせの間違ったデータと呼びます。境界合わせの間違ったデータは、実行時に、ソフトウェアで必要な調整を行うため、性能が落ちることがあります。

C プログラムでは、ポインタ変数を、あるデータ型からサイズの大きなデータ型へ型キャストした場合に、境界合わせ間違いが起こることがあります。たとえば、char ポインタ (1 バイトの境界合わせ) を int ポインタ (4 バイトの境界合わせ) へ型キャストした後、新しいポインタを逆参照すると、境界合わせされていないアクセスが発生することがあります。また、C では、`#pragma pack` 指示文を使用してパック構造体を作成しても、境界合わせされていないアクセスが行われることがあります。`#pragma pack` 指示文についての詳細は、第 3 章を参照してください。

C プログラムでの境界合わせの問題を修正するには、`-misalign` オプション (または `-assume noaligned_objects`) を使用するか、ソース・コードに必要な変更を行います。何らかの理由によって、ユーザ・プログラムで境界合わせ間違いのインスタンスを必要とする場合には、境界合わせの間違ったデータと呼び出すすべてのポインタ宣言で `__unaligned` データ型識別子を使用します。`__unaligned` として宣言されたポインタを使用してデータがアクセスされると、コンパイラは、境界合わせエラーを発生

させずにデータのコピーや格納を行うために必要な追加コードを生成します。境界合わせエラーは、性能に対して、生成される追加コードよりもはるかに重大な影響を与えます。

C プログラムのコンパイル中には、境界合わせの間違ったデータを示す警告メッセージは表示されません。ただし、プログラムの実行中には、境界合わせが間違っているデータがあると、カーネルが警告メッセージ ("unaligned access") を表示します。メッセージには、境界合わせ間違いを引き起こした命令のアドレスを示すプログラム・カウンタ (PC) 値が示されます。

次のいずれかの方法を使用して、unaligned access の問題を引き起こすコードにアクセスすることができます。

- デバッガを使用して "unaligned access" メッセージに表示されている PC 値を調べることで、境界合わせ間違いを引き起こしている命令のあるルーチン名と行番号を見つけることができます。("unaligned access" メッセージは、呼び出しルーチンから渡されたポインタが原因で表示される場合もあります。リターン・アドレス・レジスタ (ra) の中身が呼び出されたルーチンによって変更されていない場合、そのレジスタには呼び出し側ルーチンのアドレスが入っています。)
- コマンド行で `-align` オプションをオフにして、デバッガ・セッションでプログラムを実行すると、unaligned access のためにデバッガが停止した位置で、プログラムのスタックと変数を調べることができます。

データの境界合わせについての詳細は、『*Alpha Architecture Reference Manual*』を参照してください。コンパイル・コマンド行に指定できる境界合わせを制御するオプションについての詳細は、`cc(1)` を参照してください。

10.2.4 一般的なコーディングに関する考慮事項

一般的なコーディング上の考慮事項には、次のものがあります。

- `libc` 関数 (たとえば、`strcpy`, `strlen`, `strcmp`, `bcopy`, `bzero`, `memset`, `memcpy`) を使用し、同様のルーチンやユーザ独自のループを書かない。

これらの関数は、効率を良くするためハード・コードされています。

- 可能な場合には、変数には `unsigned` データ型を使用する。

これは、次の 2 つの理由に因ります。

- 変数は必ずゼロ以上であるため、コンパイラは、変数がゼロより小さい場合には適用できない最適化を実行する。
- 符号なしの除算演算では、コンパイラの生成する命令が少なくなる。

次の例を見てください。

```
int long i;
unsigned long j;
.
.
return i/2 + j/2;
```

この例では、 $i/2$ は不経済な式ですが、 $j/2$ は経済的です。

コンパイラは符号付きの $i/2$ 演算に対しては、次の 3 つの命令を生成します。

```
addq $1, 1, $28
cmovge $1, $1, $28
sra $28, 1, $2
```

一方、符号なし $j/2$ 演算に対しては、1 つの命令だけを生成します。

```
srl $3, 1, $4
```

-unsigned オプションを使用して、すべての char 宣言を unsigned char として処理することも考慮してください。

- アプリケーションが一時的に大量のデータを必要とする場合は、データを静的に宣言するのではなく、malloc 関数または alloca 組み込み関数を使用することを考慮する。

alloca 関数は、スタックからメモリを割り当てます。そのメモリは、割り当てた関数が戻ると自動的に解放されます。初めて alloca を使用するコードでは、alloca.h をインクルードしてください。そうしないと、コードが正常に動作しない場合があります。

アプリケーションで、特定の関数呼び出しのコンテキストを越えて存在するメモリが必要な場合は、malloc 関数の使用を考慮します。malloc 関数はプロセスのヒープからメモリを割り当てます。このメモリは、free を呼び出して明示的に解放するまで使用可能のままになります。

これらの関数を使用すると、物理メモリが不足しているアプリケーションで性能が改善されます。

マルチスレッド・アプリケーションでは、alloca が呼び出し元スレッドのスタックからメモリを割り当てることに注意してください。すなわち、このメモリを割り当てたり解放したりしても、争奪は起こりませ

ん。malloc 関数 (および関連する関数) は、ロックおよびアトミック・オペレーションを用いて共通プールからメモリを割り当て、同時アクセスを制御します。malloc を使用するシングル・スレッドおよびマルチスレッド・アプリケーションの性能を簡単に改善する方法については、malloc(3) の「Tuning Memory Allocation」の節を参照してください。

また、マルチスレッド・アプリケーションの場合は、arena malloc (amalloc) 機構を用いて、マルチスレッド・アプリケーションの各スレッドに対して別々のヒープをセットアップすることも考慮してください。

- 型キャスト、特に、整数から浮動小数点、および、小さなデータ型から大きなデータ型への型キャストを最小限にする。
- キャッシュ・ミスを回避するため、多次元配列が自然の記憶順にトラバースされるようにする。

つまり、行優先順で、右端の添字が 1 ずつ最も速く変化するようにします。列優先順 (Fortran で使用) は避けてください。

- アプリケーションが 32 ビット・アドレス空間に適合し、多数のポインタを含む構造体を割り当てることによって、大量の動的メモリを割り当てる場合は、-xtaso オプションを使用すると、かなりの量のメモリを節約することができる。このオプションを使用する場合には、ポインタのサイズ割り当てを制御する C 言語プリディレクタを使用して、ソース・コードを修正する必要がある。詳細については、cc(1) および第 2 章を参照。
- C プログラムでは、間接呼び出し (つまり、ルーチンまたは関数へのポインタを引数として使用する呼び出し) を使用しない。

間接呼び出しを使用すると、グローバル変数を変更する可能性があります。そのため、最適化プログラムによって安全に行われる最適化が少なくなります。

- 参照パラメータではなく値を返す関数を使用する。
- できる限り、while や for よりも do while を使用する。

do while を使用すれば、最適化プログラムは、コードをループの内側から外側に移動するために、ループ条件を複写する必要がなくなります。

- ローカル変数を使用し、グローバル変数は使用しない。

関数の外部変数は、別のソース・ファイルによって参照されない限り、static として宣言します。グローバル変数の使用を少なくすれば、コンパイラによって最適化しやすくなります。

- 参照パラメータやグローバル変数を使用しないで、値パラメータを使用する。

参照パラメータを使用すると、ポインタと同様に、最適化しにくくなります。

- 簡潔なコードを使用する。

たとえば、式の中では ++ や -- はなるべく使用しないようにします。これらのコードがもたらす副作用を考慮しないでその有用性の目的で使用する、コード全体に悪影響を及ぼします。たとえば、なるべく次のようなコードは避けます。

```
while (n--)  
{  
    .  
    .  
}
```

なるべく次のようなコードを使用してください。

```
while (n != 0)  
{  
    n--;  
    .  
    .  
}
```

- & 値を使用したアドレスの受け渡しは避ける。

& を使用すると別名が生成され、最適化プログラムがレジスタから省略時の格納ディレクトリに変数を格納し、最適化が困難になります。

- 引数の数が可変である関数は使用しない。

このような関数を使用すると、最適化プログラムは入力されるすべてのパラメータ・レジスタを保存します。

- 他のソース・モジュールから参照されない場合は関数を静的に宣言する。

静的な関数を使用すると、より効果的な呼び出しシーケンスが使用されます。

また、逆参照の結果を格納する場合にはローカル変数を使用し、できるだけ別名を使用しないようにしてください。逆参照の結果とは、指定されたアドレスから得られる値のことです。逆参照の結果は、間接参照の演算および呼び出しによって影響を受けますが、ローカル変数は影響を受けません。これは、ローカル変数は、レジスタに保持できるためです。例 10-1 は、ポ

インタを適切に配置し、別名を削除することによって、コンパイラがより良いコードを生成することを示したものです。

例 10-1: ポインタと最適化

```
int len = 10;
char a[10];

void
zero()
{
    char *p;
    for (p = a; p != a + len; ) *p++ = 0;
}
```

例 10-1 のポインタの使用方法について考慮してみてください。文 `*p++ = 0` は `len` を変更することがあるので、コンパイラはループの外側でレジスタを使用して `a + len` を 1 度だけ演算するのではなく、ループを 1 回実行するたびにメモリから `len` をロードして、`a` のアドレスに加算しなければなりません。

次の 2 つの方法を使用すると、例 10-1 のコードの効率を改善することができます。

- ポインタの代わりに添字を使用する。次の例に示すように、`azero` プロシージャで添字を使用すれば、別名を指定しなくてもすみます。コンパイラは、`len` の値をレジスタに保持し、2 つの命令をセーブしています。ポインタがソース・コードに指定されていなくても、コンパイラは、ポインタを使用して `a` を効率的にアクセスします。

ソース・コード

```
char a[10];
int len;
void
azero()
{
    int i;
    for (i = 0; i != len; i++) a[i] = 0;
}
```

- ローカル変数の使用。次の例に示すように `len` をローカル変数または仮引数として指定すると、別名が使用不能になるため、コンパイラが `len` をレジスタに置くことができます。

ソース・コード

```
char a[10];  
void  
lpzero(len)  
    int len;  
    {  
        char *p;  
        for (p = a; p != a + len; ) *p++ = 0;  
    }
```

例外条件の処理

例外は、現在実行しているスレッドで起こる特別な状態であり、その状態を認識して適切なアクションをとる実行コードが必要です。このコードは、例外ハンドラと呼ばれます。

終了ハンドラは、制御フローが特定のコード本体を出るときに実行されるコードから構成されます。終了ハンドラは、コード本体から出ることにより、メモリ・バッファの解放やロックのリリースなどのタスクを実行して、設定されたコンテキストをクリーンアップする場合に有効です。

この章では、次の項目について説明します。

- 例外処理の概要 (11.1 節)
- ユーザ・プログラムで起こす例外 (11.2 節)
- 構造化例外ハンドラの作成 (11.3 節)
- 終了ハンドラの作成 (11.4 節)

11.1 例外処理の概要

Tru64 UNIX システムでは、『Alpha Architecture Reference Manual』で説明しているように、ハードウェアが例外をトラップして、それをオペレーティング・システムのカーネルに引き渡します。カーネルは、不良メモリ・アクセスや算術トラップなどのハードウェア例外を、シグナルに変換します。プロセスは、シグナルの引き渡しを可能にし、シグナル・ハンドラを設定して、プロセス内でシグナルを処理します。

『Calling Standard for Alpha Systems』では、Tru64 UNIX システムにおける例外的なイベントの処理を可能にする特別な構造とメカニズムについて、詳しく定義しています。この規格に定義されている処理には、次のようなものがあります。

- 例外ハンドラが設定される方法
- 例外が引き起こされる方法

- 例外システムがハンドラを探索して呼び出す方法
- ハンドラが例外システムに戻る方法
- 例外システムがスタックをトラバースして、プロシージャ・コンテキストを維持する方法

Tru64 UNIX の C コンパイラの構造化例外処理機能をサポートする実行時例外ディスパッチャは、規格に記述されているフレーム・ベースの例外ハンドラの 1 例です。構造化例外処理の説明については、11.3 節を参照してください。

以降の各項で、『Calling Standard for Alpha Systems』に定義されている例外処理メカニズムをサポートする Tru64 UNIX の構成要素について簡単に説明します。

11.1.1 C コンパイラ構文

Tru64 UNIX の C コンパイラが提供する構文では、ユーザ定義およびシステム定義の例外条件に対して、コード領域を保護することができます。このメカニズムは、構造化例外処理と呼ばれ、ユーザが例外ハンドラおよび終了ハンドラを定義するとともに、保護するコード領域を示すことができます。

`c_except.h` ヘッダ・ファイルで定義しているシンボルと関数を使用すると、ユーザ例外処理コードは、現在の例外コードおよび例外について説明しているその他の情報を取得できます。

11.1.2 libexc ライブラリ・ルーチン

例外サポート・ライブラリ `/usr/ccs/lib/cmplrs/cc/libexc.a` では、次の機能を持つルーチンを提供しています。

- ユーザ定義の例外を起こしたり、UNIX シグナルを例外に変換する機能
このようなルーチンには、次のものがあります。

```
exc_raise_status_exception  
exc_raise_signal_exception  
exc_raise_exception  
exc_exception_dispatcher  
exc_dispatch_exception
```

これらの例外管理ルーチンは、例外を適切なハンドラへディスパッチするメカニズムも提供しています。11.3 節で説明する C 言語の構造化例外処理の場合、C 固有のハンドラは、ユーザ作成のコードを含む

ルーチン呼び出して、実行するアクションを決定します。ユーザ作成のコードは、例外を処理するか、または別のプロシージャを起動してその例外を処理します。

- タスクからのプロシージャ起動レベルの仮想および実際の展開、およびハンドラまたはその他のユーザ・コードにおいて実行の継続を行う機能
このようなルーチンには、次のものがあります。

```
unwind
exc_virtual_unwind
RtlVirtualUnwind
exc_resume
exc_longjmp
exc_continue
exc_unwind
RtlUnwindRfp
```

展開ルーチンのいくつかは、展開時のハンドラ呼び出しもサポートして、特定のプロシージャ起動において、言語またはユーザがアイテムをクリーンアップできるようにします。

- プロシージャ固有の情報にアクセスして、ルーチン内の任意のアドレスを対応するプロシージャ情報にマップする機能

この情報は展開を引き起こすのに十分なデータを含んでいるか、またはルーチンが例外を処理するかどうかを決定します。このようなルーチンには、次のものがあります。

```
exc_add_pc_range_table
exc_remove_pc_range_table
exc_lookup_function_table_address
exc_lookup_function_entry
find_rpd
exc_add_gp_range
exc_remove_gp_range
exc_lookup_gp
```

C 言語の構造化例外ハンドラは、最後の2つのカテゴリのルーチン呼び出して、ユーザ・コードが例外を処理して実行を再開し、ユーザ定義の例外ハンドラを探索してディスパッチできるようにします。この処理については、11.3 節で説明しています。/usr/ccs/lib/cmplrs/cc/libexc.a で提供するルーチンについての詳細は、そのルーチンのリファレンス・ページを参照してください。

11.1.3 例外処理をサポートするヘッダ・ファイル

さまざまなヘッダ・ファイルで、例外処理システムおよびプロシージャ・コンテキストの操作をサポートする構造体を定義しています。表 11-1 に、このようなヘッダ・ファイルの一覧を示します。

表 11-1: 例外処理をサポートするヘッダ・ファイル

ファイル	説明
excpt.h	例外コードの構造体および Tru64 UNIX の例外コードの数を定義する。システム例外、コンテキスト・レコード、関連するフラグ、シンボリック定数、実行時プロシージャ・タイプ、および libexc.a で提供される関数用のプロトタイプも定義する。詳細は excpt(4) を参照。
c_excpt.h	C 言語の構造化例外ハンドラおよび終了ハンドラが使用するシンボルを定義する。また、例外コードを返す例外情報構造体と関数、その他の例外情報、および終了ハンドラが呼び出される状態に関する情報も定義する。詳細は c_excpt(4) を参照。
machine/fpu.h	IEEE 浮動小数点例外の引き渡しとその出現を記録する情報を探索するルーチン、ieee_set_fp_control および ieee_get_fp_control 用のプロトタイプを定義する。また、これらのルーチンをサポートする構造体と定数も定義する。詳細は ieee(3) を参照。
pdsc.h	実行時プロシージャ記述子およびコード範囲記述子などの構造体を定義する。これらは、『 <i>Calling Standard for Alpha Systems</i> 』に記述されているプロシージャ・タイプおよびフロー制御メカニズムに対して、実行時コンテキストを提供する。詳細は pdsc(4) を参照。

11.2 ユーザ・プログラムで起こす例外

ユーザ・プログラムは、通常、次の 2 つの方法で例外を起こします。

- プログラムは、exc_raise_exception または exc_raise_status_exception 関数を呼び出すことによって、アプリケーション固有の例外を明示的に開始できる。

これらの関数を使用すると、呼び出し側のプロシージャは、例外について説明する情報を指定できます。

- プログラムは、POSIX シグナルを例外に変換する特殊なシグナル・ハンドラ exc_raise_signal_exception をインストールできる。

exc_raise_signal_exception 関数は例外ディスパッチャを呼び出して、実行時スタックを探索し、現在または以前のスタック・フレームで

設定された例外ハンドラを探します。この場合、ハンドラに報告されるコードには、機能フィールドに `EXC_SIGNAL`、およびコード・フィールドにシグナル値が入ります。コードのデータ構造体についての詳細は、`except(4)` および `except.h` ヘッダ・ファイルを参照してください。

注意

算術例外およびソフトウェア生成例外の正確な例外コードは、`signal.h` ヘッダ・ファイルに定義されており、`code` 引数でシグナル・ハンドラに渡されます。特殊なシグナル・ハンドラ `exc_raise_signal_exception` は、例外ディスパッチャを呼び出す前に、このコードを `ExceptionRecord.ExceptionInfo[0]` に移動します。

11.3 節の例は、例外を明示的に起こして、シグナルを例外に変換する方法を示しています。

11.3 構造化例外ハンドラの作成

Tru64 UNIX の C コンパイラが提供する構造化例外処理機能を使用すると、特定のコード・シーケンスにおいて特定の例外条件が発生したときの処理を記述できます。これらの機能は常に有効です (`cc` コマンドの `-ms` オプションは不要です)。構造化例外ハンドラを設定する構文は、次のとおりです。

```
try {  
    try-body  
  
}  
except ( exception-filter ) {  
    exception-handler  
  
}
```

`try-body` は、例外ハンドラが保護する文または文のブロックです。`try` 本体を実行中に例外が起きた場合、C 固有の実行時ハンドラは `exception-filter` を評価して、制御を関連する `exception-handler` に移すか、外部レベルの `try` 本体でハンドラの探索を継続するか、あるいは、例外が起きた場所から通常の実行を継続するかを決定します。

exception-filter は、try 本体を保護する例外ハンドラに関連する式です。これは、単純式でも、式を評価する関数を呼び出しても構いません。例外フィルタは、例外ディスパッチャが例外処理を終了するために、次のいずれかの整数値に評価されなければなりません。

- `< 0` (`EXCEPTION_CONTINUE_EXECUTION`)

例外ディスパッチャは例外を無視し、例外によって中断された実行スレッドを再開します。継続不可能な例外の場合、ディスパッチャは `STATUS_NONCONTINUABLE_EXCEPTION` 例外を起こします。

- `0` (`EXCEPTION_CONTINUE_SEARCH`)

例外ディスパッチャはハンドラの探索を継続します。まず、現在のハンドラがネストされている可能性のある `try...except` ブロックを探索し、次に、実行時スタックにおいて、現在のフレームの前のプロシージャ・フレームで定義されている `try...except` ブロックを探索します。フィルタが例外を処理しないことを選択した場合、通常この値が返されます。

- `> 0` (`EXCEPTION_EXECUTE_HANDLER`)

例外ディスパッチャは制御を例外ハンドラに移し、ハンドラが見つかった実行時スタックのフレームで実行を継続します。このプロセスは、例外処理と呼ばれ、現在のフレームの下すべてのプロシージャ・フレームを展開して、それらのフレーム内で設定された終了ハンドラを実行します。

例外フィルタにおいて、次の2つの intrinsic 関数が、フィルタされている例外に関する情報にアクセスできます。

```
long          exception_code ();
Exception_info_ptr exception_info ();
```

`exception_code` 関数は、例外コードを返します。`exception_info` 関数は、`EXCEPTION_POINTERS` 構造体へのポインタを返します。このポインタを使用すると、例外が起きたときのマシン状態 (たとえば、システム例外やコンテキスト・レコード) にアクセスできます。詳細は、`except(4)` および `c_except(4)` を参照してください。

`exception_code` 関数は、例外フィルタまたは例外ハンドラで使用できます。しかし、`exception_info` 関数は、例外フィルタ内でのみ使用可能です。例外ハンドラにおいて、`exception_info` 関数から返された情報を使

用する必要がある場合は、その関数をフィルタ内で呼び出して、情報をローカルに格納しなければなりません。フィルタ外で例外構造体を参照する必要がある場合は、同時にそれらをコピーしておかなければなりません。これは、それらのストレージが、フィルタの実行中のみ有効なためです。

例外が起こると、例外ディスパッチャは、ハンドラが設定されているフレームに到達するまで、仮想的に実行時スタックを展開します。ディスパッチャは最初に、例外が起きたときに現在のスタック・フレームであったスタック・フレームで例外ハンドラを探索します。

ハンドラがこのスタック・フレームにない場合には、ディスパッチャは、現在のスタック・フレームおよび介在するスタック・フレームをそのままにして、例外ハンドラを設定しているフレームに到達するまで、仮想的にスタックを(それ自身のコンテキストにおいて)展開します。その後、そのハンドラに関連する例外フィルタを実行します。

この例外ディスパッチのフェーズでは、ディスパッチャは実行時スタックを仮想的にのみ展開することに注意してください。つまり、スタック上に存在している呼び出しフレームは依然その場所にあります。例外ハンドラを見つけれない場合、またはすべてのハンドラで例外が再度起こる場合は、例外ディスパッチャはシステムのラスト・チャンス・ハンドラを呼び出します。ラスト・チャンス・ハンドラの設定方法については、`exc_set_last_chance_handler(3)` を参照してください。

例外フィルタを Pascal スタイルのネストしたプロシージャのように処理することによって、例外処理コードはフィルタ式を `try...except` ブロックを含むプロシージャの有効範囲内で評価します。これにより、そのフィルタを含むプロシージャのスタック・フレームまで、スタックが実際に展開されていなくても、フィルタ式は、そのフィルタを含むプロシージャのローカル変数にアクセスできます。

例外ハンドラを実行する前に(たとえば、例外フィルタが `EXCEPTION_EXECUTE_HANDLER` を返す場合)、例外ディスパッチャは実行時スタックを実際に展開して、制御を例外ハンドラに移した結果として終了した `try...finally` ブロックに対して設定された終了ハンドラを実行します。ディスパッチャが例外ハンドラを呼び出すのは、その後だけです。

`exception-handler` は、例外条件を処理する複合文です。これは、`try...except` 構文を含むプロシージャの有効範囲内で実行され、そのローカル変数にアクセスできます。ハンドラは、例外の種類によって、さまざま

な方法で例外に対応できます。たとえば、エラーのログを取ったり、例外が生じる状況を修正することができます。

例外フィルタまたは例外ハンドラのどちらも、取得した例外情報を修正したり拡張し、C 言語の例外ディスパッチャに依頼して、外部の try 本体または以前の呼び出しフレームで設定された例外コードに新しい情報を引き渡すことができます。この処理は、例外フィルタ内で行う方が簡単であり、最後に実行されているプロシージャのフレームで行なわれ、例外コンテキストは実行時スタックにそのまま残ります。フィルタは、ディスパッチャに 0 を返すだけで処理を完了し、そのディスパッチャに次のハンドラの探索を継続することを要求します。

例外ハンドラが以前に設定されたハンドラを呼び出すためには、例外ハンドラは自分のコンテキストから、以前に設定されたハンドラで処理する別の例外を起こさなければなりません。

例 11-1 は、簡単な例外ハンドラを設定し、exc_raise_signal_exception シグナル・ハンドラによって、例外に変換されたセグメンテーション違反シグナル (SIGSEGV) を処理する方法を示しています。

例 11-1: 構造化例外としての SIGSEGV シグナルの処理

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <except.h>

void main(void)
{
    struct sigaction    act, oldact;
    char                *x=0;

    /*
     * Set up things so that SIGSEGV signals are delivered. Set
     * exc_raise_signal_exception as the SIGSEGV signal handler
     * in sigaction.
     */
    act.sa_handler = exc_raise_signal_exception;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (sigaction(SIGSEGV, &act, &oldact) < 0)
        perror("sigaction:");
    /*
     * If a segmentation violation occurs within the following try
     * block, the run-time exception dispatcher calls the exception

```

例 11-1: 構造化例外としての SIGSEGV シグナルの処理 (続き)

```
* filter associated with the except statement to determine
* whether to call the exception handler to handle the SIGSEGV
* signal exception.
*/
try {
    *x=55;
}
/*
 * The exception filter tests the exception code against
 * SIGSEGV. If it tests true, the filter returns 1 to the
 * dispatcher, which then executes the handler; if it tests
 * false, the filter returns -1 to the dispatcher, which
 * continues its search for a handler in the previous run-time
 * stack frames. Eventually the last-chance handler executes.
 * Note: Normally the printf in the filter would be replaced
 * with a call to a routine that logged the unexpected signal.
 */
except(exception_code() == EXC_VALUE(EXC_SIGNAL,SIGSEGV) ? 1 :
        (printf("unexpected signal exception code 0x%lx\n",
                exception_code()), 0))
{
    printf("segmentation violation reported: handler\n");
    exit(0);
}
printf("okay\n");
exit(1);
}
```

次は、このプログラムの実行例です。

```
% cc -std0 segfault_ex.c -lexc
% a.out
segmentation violation reported in handler
```

例 11-2 は例 11-1 と同様に、シグナル例外の処理方法を示していますが、この場合は SIGFPE を処理します。この例では、さらに、IEEE 浮動小数点例外であるゼロによる浮動除算を、`ieee_set_fp_control()` への呼び出しにより使用可能にする方法、およびハンドラがシステム例外レコードを読み取ることでより詳細な情報を取得する方法を示します。

例 11-2: 構造化例外としての IEEE 浮動小数点 SIGFPE の処理

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <except.h>
#include <machine/fpu.h>
#include <errno.h>

int main(void)
{
    Exception_info_ptr    except_info;
    system_exrec_type     exception_record;
    long                  code;
    struct sigaction       act, oldact;
    unsigned long         float_traps=IEEE_TRAP_ENABLE_DZE;
    double                 temperature=75.2, divisor=0.0, quot, return_val;

/*
 * Set up things so that IEEE DZO traps are reported and that
 * SIGFPE signals are delivered.  Set exc_raise_signal_exception
 * as the SIGFPE signal handler.
 */
    act.sa_handler = exc_raise_signal_exception;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (sigaction(SIGFPE, &act, &oldact) < 0)
        perror("sigaction:");
    ieee_set_fp_control(float_traps);

/*
 * If a floating divide-by-zero FPE occurs within the following
 * try block, the run-time exception dispatcher calls the
 * exception filter associated with the except statement to
 * determine whether the SIGFPE signal exception is to be
 * handled by the exception handler.
 */
    try {
        printf("quot = IEEE %.2f / %.2f\n",temperature,divisor);
        quot = temperature / divisor;
    }

/*
 * The exception filter saves the exception code and tests it
 * against SIGFPE. If it tests true, the filter obtains the
 * exception information, copies the exception record structure,
 * and returns 1 to the dispatcher which then executes the handler.
 * If the filter's test of the code is false, the filter
 * returns 0 to the handler, which continues its search for a
 * handler in previous run-time frames. Eventually the last-chance
 * handler executes. Note: Normally the filter printf is replaced
 * with a call to a routine that logged the unexpected signal.
 */
    except((code=exception_code()) == EXC_VALUE(EXC_SIGNAL,SIGFPE) ?
        (except_info = exception_info(),
         exception_record = *(except_info->ExceptionRecord), 1) :
        (printf("unexpected signal exception code 0x%lx\n",
                exception_code()), 0))

/*
 * The exception handler follows and prints out the signal code,
 * which has the following format:
 */
    * 0x          8          0ffe          0003
    * |          |          |          |
```


例 11-2: 構造化例外としての IEEE 浮動小数点 SIGFPE の処理 (続き)

```
* hex          SIGFPE          EXC_OSF facility  EXC_SIGNAL
*/
{ printf("Arithmetic error\n");
  printf("exception_code() returns 0x%lx\n", code);
  printf("EXC_VALUE macro in excpt.h generates 0x%lx\n",
        EXC_VALUE(EXC_SIGNAL, SIGFPE));
  printf("Signal code in the exception record is 0x%lx\n",
        exception_record.ExceptionCode);
}
/*
 * To find out what type of SIGFPE this is, look at the first
 * optional parameter in the exception record. Verify that it is
 * FPE_FLTDIV_TRAP).
 */
    printf("No. of parameters is %lu\n",
           exception_record.NumberParameters);
    printf("SIGFPE type is 0x%lx\n",
           exception_record.ExceptionInformation[0]);
/*
 * Set return value to IEEE_PLUS_INFINITY and return.
 */
    if (exception_record.ExceptionInformation[0] ==
        FPE_FLTDIV_TRAP)
    {
        *((long*)&return_val) = IEEE_PLUS_INFINITY;
        printf("Returning 0x%f to caller\n", return_val);
        return 0;
    }
/*
 * If this is a different kind of SIGFPE, return gracefully.
 */
    else
        return -1;
}
/*
 * We get here only if no exception occurred in the try block.
 */
    printf("okay: %f\n", quot);
    exit(1);
}
```

次は、このプログラムの実行例です。

```
% cc -std0 sigfpe_ex.c -lexc
% a.out
quot = IEEE 75.20 / 0.00
Arithmetic error
exception_code() returns 0x80ffe0003
The EXC_VALUE macro in excpt.h generates 0x80ffe0003
The signal code in the exception record is 0x80ffe0003
No. of parameters is 1
SIGFPE type is 0x4
Returning 0xINF to caller
```

プロシージャ (または相互に関係のあるプロシージャのグループ) は、`try...except` 構造をいくつでも含むことができ、また、これらの構造はネストしても構いません。 `try...except` ブロック内で例外が起こると、システムはそのブロックに関連する例外ハンドラを呼び出します。

例 11-3 は、2 つのプライベートな例外コードによって定義され、最も内側の `try` ブロック内でこれら 2 つの例外のどちらかを起こす、複数の `try...except` ブロックの動作を示しています。

例 11-3: 複数の構造化例外ハンドラ

```
#include <excpt.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define EXC_NOTWIDGET EXC_VALUE(EXC_C_USER, 1)
#define EXC_NOTDECWIDGET EXC_VALUE(EXC_C_USER, 2)

void getwidgetbyname(char *);

/*
 * main() sets up an exception handler to field the EXC_NOTWIDGET
 * exception and then calls getwidgetbyname().
 */
void main(int argc, char *argv[])
{
    char widget[20];
    long code;
    try {
        if (argc > 1)
            strcpy(widget, argv[1]);
        else
        {
            printf("Enter widget name: ");
            gets(widget);
        }
        getwidgetbyname(widget);
    }
    except((code=exception_code()) == EXC_NOTWIDGET)
    {
        printf("Exception 0x%x: %s is not a widget\n",
            code, widget);
        exit(0);
    }
}
/*
 * getwidgetbyname() sets up an exception handler to field the
```

例 11-3: 複数の構造化例外ハンドラ (続き)

```
* EXC_NOTDECWIDGET exception. Depending upon the data it is
* passed, its try body calls exc_raise_status_exception() to
* generate either of the user-defined exceptions.
*/
void
getwidgetbyname(char* widgetname)
{
    long code;
    try {
        if (strcmp(widgetname, "foo") == 0)
            exc_raise_status_exception(EXC_NOTDECWIDGET);
        if (strcmp(widgetname, "bar") == 0)
            exc_raise_status_exception(EXC_NOTWIDGET);
    }
    /*
    * The exception filter tests the exception code against
    * EXC_NOTDECWIDGET. If it tests true, the filter returns 1
    * to the dispatcher; if it tests false, the filter returns -1
    * to the dispatcher, which continues its search for a handler
    * in the previous run-time stack frames. When the generated
    * exception is EXC_NOTWIDGET, the dispatcher finds its handler
    * in main()'s frame.
    */
    except((code=exception_code()) == EXC_NOTDECWIDGET)
    {
        printf("Exception 0x%x: %s is not "
               "a Hewlett-Packard-supplied widget\n",
               code, widgetname);
        exit(0);
    }
    printf("widget name okay\n");
}
```

次は、このプログラムの実行例です。

```
% cc raise_ex.c -lexc
% a.out
Enter widget name: foo
Exception 0x20ffe009: foo is not a Hewlett-Packard-supplied widget
% a.out
Enter widget name: bar
Exception 0x10ffe009: bar is not a widget
```

11.4 終了ハンドラの作成

cc コンパイラは、保護されたコード本体から制御が渡されると、指定された終了コードのブロックが必ず実行されるようにします。終了コードは、制御フローが保護されたコードを出る方法にかかわらず実行されます。たとえば、保護されたコード本体の実行中に、例外またはその他のエラーが生じて、終了ハンドラは、クリーンアップ・タスクが確実に実行されるようにします。

終了ハンドラの構文は、次のとおりです。

```
try {  
    try-body  
  
}  
finally {  
    termination-handler  
  
}
```

`try-body` は、複合文として表現されたコードであり、終了ハンドラが保護します。try 本体は、文のブロックまたはネストしたブロックの集まりでも構いません。これには、次の文を含むことができます。この文は、ブロックから直ちに出て、終了ハンドラを実行します。

`leave;`

注意

Tru64 UNIX の `longjmp()` ルーチンは、展開操作を使用しません。したがって、フレーム・ベースの例外処理がある場合には、`try-body` または `termination-handler` から `longjmp()` を使用しないでください。代わりに、展開操作を介してインプリメントされる `exc_longjmp()` を使用してください。

`termination-handler` は、try 本体が正常終了したか異常終了したかにかかわらず、制御フローが保護された try 本体を出ると実行される複合文です。ブロック内の最後の文が実行された（つまり、本体の `}` に到達した）とき、保護された本体は正常終了したとみなされます。 `leave` 文を使用し

ても、正常終了します。制御フローがその他の方法で保護された本体を出ると、その本体は異常終了します。たとえば、例外や、`return`、`goto`、`break`、`continue` などの制御文で保護された本体を出た場合です。

終了ハンドラは次の `intrinsic` 関数を呼び出して、保護された本体が正常終了したか異常終了したかを判断できます。

```
int abnormal_termination ();
```

`try` 本体がシーケンシャルに ("}") に到達することによって) 完了した場合、`abnormal_termination` 関数は 0 を返し、そうでない場合は 1 を返します。

終了ハンドラ自体は、シーケンシャルに終了することも、ハンドラの外に制御を渡して終了することもできます。シーケンシャルに ("}") に到達することによって) 終了する場合、その後の制御フローは、次のように `try` 本体の終了方法に依存します。

- `try` 本体が正常に終了した場合、完全な `try...finally` ブロックの次の文から実行が継続される。
- 本体の外への明示的な飛び越しによって `try` 本体が異常終了した場合、その飛び越しが完了する。

ただし、1 つ以上の `try...finally` 文を含む本体にその飛び越しがある場合は、制御が最後に飛び越しのターゲットに渡される前に、終了ハンドラが呼び出されます。

- `try` 本体が展開により異常終了した場合、例外ハンドラへの飛び越し、または `exc_longjmp` 呼び出しにより、制御は C 実行時例外ハンドラへ戻る。

この例外ハンドラは、展開のターゲットへ飛び越す前に、要求によって終了ハンドラの呼び出しを継続します。

例外フィルタと同様に、終了ハンドラは Pascal スタイルのネストしたプロシージャとして処理され、実行時スタックからフレームを削除せずに実行されます。終了ハンドラは、プロシージャで宣言されているローカル変数に、このようにしてアクセスできます。

異常終了 (および例外) は、ほとんどのプログラムにとって、通常の制御フロー外と考えられるため、異常終了の処理には実行コストがかかります。`try` 本体外への明示的な飛び越しは、異常終了とみなされることを覚えて

おいてください。正常終了は単純な場合であり、実行時にかかるコストが少なくてすみます。

場合によっては、try 本体外への飛び越しを leave 文 (制御を最も内側の try 本体の終わりに移す) に置き換えて、try...finally ブロックを完了した後に状態変数を検査すると、このコストを回避できます。

終了ハンドラ自体は、制御の引き渡し (たとえば、goto、break、continue、return、exc_longjmp、または例外の発生) によって、シーケンシャルでない方法で終了する (たとえば、展開の異常終了) ことがあります。この制御の引き渡しが別の try...finally ブロックに存在する場合、終了ハンドラが実行されます。

例 11-4 は、例外によって最も内側の try 本体が終了するとき、終了ハンドラおよび例外ハンドラが実行される順序を示しています。

例 11-4: 例外による try 本体の異常終了

```
#include <stdio.h>
#include <signal.h>
#include <except.h>
#include <errno.h>

#define EXC_FOO EXC_VALUE(EXC_C_USER, 1)

signed int
foo_except_filter(void)
{
    printf("2. The exception causes the exception filter "
           "to be evaluated.\n");
    return 1;
}

void main (void)
{
    try {
        try {
            printf("1. The main body executes.\n");
            exc_raise_status_exception(EXC_FOO);
        }
        finally {
            printf("3. The termination handler executes "
                   "because control will leave the "
                   "try...finally block to \n");
        }
    }
    except(foo_except_filter()) {
        printf("4. execute the exception handler.\n");
    }
}
```

次に示すのは、このプログラムの実行例です。

```
% cc segfault_ex2.c -lexc
% a.out
1. The main body executes.
2. The exception causes the exception filter to be evaluated.
3. The termination handler executes because control will leave the
   try...finally block to
4. execute the exception handler.
```



スレッド・セーフなライブラリの開発

マルチスレッド・アプリケーションの開発をサポートするために、Tru64 UNIX オペレーティング・システムでは、POSIX Threads Library (Compaq Multithreading Run-Time Library) を提供しています。POSIX Threads Library インタフェースは、いくつかの拡張機能を加えた、IEEE 規格 1003.1c-1995 スレッド (POSIX P1003.1C スレッドとも呼ぶ) の Tru64 UNIX のインプリメンテーションです。

実際のスレッド・インタフェースのほかに、オペレーティング・システムでは TIS (Thread-Independent Services: スレッド独立型サービス) を提供しています。TIS ルーチンは、独自のスレッドを作成しない効率的なスレッド・セーフ・ライブラリの作成を支援します (TIS ルーチンについての詳細は、12.4.1 項を参照)。

この章では、次の項目について説明します。

- Tru64 UNIX におけるマルチスレッド・サポートの概要 (12.1 節)
- POSIX 準拠のための実行時ライブラリの変更 (12.2 節)
- スレッド・セーフおよびスレッド・リエントラント・ルーチンの特性 (12.3 節)
- スレッド・セーフ・コードの作成方法 (12.4 節)
- マルチスレッド・アプリケーションの作成方法 (12.5 節)

12.1 スレッド・サポートの概要

スレッドとは、プログラム内における単一のシーケンシャルな制御の流れのことです。複数のスレッドが同時に実行され、アドレス空間をはじめ、所有しているプロセスのほとんどのリソースを共用します。省略時の設定では、プロセスには最初に 1 つのスレッドがあります。

複数のスレッドは、次のような目的で使用されます。

- マルチプロセッサ・システム上で実行するアプリケーションの性能改善

- 特定のプログラミング・モデル (たとえば, クライアント/サーバ・モデル) の実現
- 低速デバイス処理のカプセル化と分離

また, 複数のスレッドを特定のイベント管理の代替方法として使用することもできます。たとえば, `select()` あるいは `poll()` システム・コールを使用して, 複数のファイル記述子への同時 I/O オペレーションを管理する代わりに, 各プロセスのファイル記述子ごとに 1 つのスレッドを使用することもできます。

Tru64 UNIX システムに対するマルチスレッド開発環境の構成要素は次のとおりです。

- コンパイラ・サポート
cc または c89 コマンドで `-pthread` オプションを使用してコンパイルします。
- スレッド・パッケージ
`libpthread.so` ライブラリはスレッド制御のためのインタフェースを提供するものです。
- スレッド・セーフ・サポート・ライブラリ
スレッド・セーフ・サポート・ライブラリには, `libaio`, `libcfs`, `liblmf`, `libm`, `libmsfs`, `libpruplist`, `libpthread`, `librt`, および `libsys5` が含まれます。
- ladebug デバッガ
- prof プロファイラ および gprof プロファイラ
`libprof1_r.a` プロファイリング・ライブラリを使用するためには, `prof` に対しては `-p` および `-pthread` オプションを, `gprof` に対しては `-pg` および `-pthread` オプションを指定してコンパイルします。
- atom ベースの `pixie`, `hiprof`, `third` ツール

マルチスレッド・アプリケーションのプロファイリングについては, 8.8 節を参照してください。

ロジックおよび性能の潜在的な問題について, マルチスレッド・アプリケーションを分析するには, Visual Threads (「Associated Products Volume 1」CD-ROM から利用可能) を使用することができます。Visual Threads は,

POSIX Threads Library アプリケーションおよび Java アプリケーションで
使用することができます。

12.2 POSIX 準拠のための実行時ライブラリの変更

DEC OSF/1 オペレーティング・システム (DIGITAL UNIX Version 4.0 より古いバージョン) のリリースでは、別個のリエントラント・ルーチン (*_r ルーチン) を多数提供して、C 実行時ライブラリ内の静的データの問題 (最初の 2 つの問題については 12.3.1 項を参照) を解決していました。Tru64 UNIX オペレーティング・システムのリリースでは、静的データをスレッド固有のデータと置き換えることによって、リエントラントでないルーチンの静的データの問題を解決しています。POSIX 1003.1c で指定する数個のルーチンを除き、Tru64 UNIX システムでは代替ルーチンは必要なく、バイナリ互換性のみのために保持されています。

POSIX 1003.1c で指定されている代替スレッド・セーフ・ルーチンは次の関数だけであり、スレッド・セーフなコードを記述する際には必要です。

asctime_r*	ctime_r*	getgrgid_r*
getgrnam_r*	getpwnam_r*	getpwuid_r*
gmtime_r*	localtime_r*	rand_r*
readdir_r*	strtok_r	

DIGITAL UNIX バージョン 4.0 からは、前述の一覧でアスタリスク (*) の付いているインタフェースは、POSIX 1003.1c に準拠する新しい定義になっています。これらのルーチンの旧バージョンは、プリプロセッサ・シンボル `_POSIX_C_SOURCE` に値 `199309L` (POSIX 1003.1b 準拠を示す — ただし、これを行うと POSIX 1003.1c スレッドが無効になる) を指定して定義することにより取得できます。これらのルーチンの新バージョンは、DIGITAL UNIX バージョン 4.0 以降でコードをコンパイルしたときの省略時の設定ですが、各ルーチンのリファレンス・ページで指定されているヘッダ・ファイルをインクルードすると確実です。

スレッドを使用するプログラミングについては、『*Guide to the POSIX Threads Library*』および `cc(1)`、`monitor(3)`、`prof(1)`、`gprof(1)` を参照してください。

12.3 スレッド・セーフ・ルーチンおよびリエントラント・ルーチンの特性

ライブラリ内のルーチンは、スレッド・セーフであっても、スレッド・セーフでなくても構いません。スレッド・セーフなルーチンは、複数のスレッドから、スレッド間の望ましくない相互作用なしで、同時に呼び出すことができるルーチンです。ルーチンは、次のいずれかの理由によってスレッド・セーフのことがあります。

- 本質的にリエントラントである。
- ミューテックスでロックやスレッド固有のデータを使用する。

ミューテックスは、複数のスレッドが共用データへのアクセスを直列化できるようにするために使用される同期オブジェクトです。

リエントラント関数は、複数のスレッドからの同時呼び出しで、状態を共用しません。リエントラント・ルーチンは理想的なスレッド・セーフのルーチンですが、すべてのルーチンがリエントラントとして作成できるわけではありません。

DIGITAL UNIX バージョン 4.0 より前では、C 実行時ライブラリ (libc) ルーチンの多数はスレッド・セーフではなく、これらのルーチンの代替バージョンが `libc_r` で提供されていました。DIGITAL UNIX バージョン 4.0 からは、以前 `libc_r` で提供されていたすべての代替バージョンは、`libc` にマージされました。スレッド・セーフ・ルーチンとそれに対応するスレッド・セーフでないルーチンが同じ名前を持っている場合は、スレッド・セーフでないルーチンが置き換えられました。スレッド・セーフのルーチンは TIS ルーチンを使用するように変更されています (12.4.1 項を参照)。これは、シングル・スレッドの場合に広範囲にわたるオーバーヘッドなしで、シングル・スレッドおよびマルチスレッドの両方の環境で動作します。

12.3.1 スレッド・セーフでないコーディング例

コードをスレッド・セーフにしないようにする方法は、DIGITAL UNIX バージョン 4.0 より前にスレッド・セーフでなかった `libc` 関数のいくつかを調べるとわかります。

- ポインタを単一の静的に割り当てられたバッファに返す

この問題の一例として、`ctime(3)` インタフェースがあります。

```
char *ctime(const time_t *timer);
```

12-4 スレッド・セーフなライブラリの開発

この関数は出力引数をとらず、ポインタを静的に割り当てられたバッファに返します。このバッファには、関数の単一パラメータに指定された時間の ASCII 表現による文字列が含まれています。単一の静的に割り当てられたバッファがこの目的に使用されるため、この関数を呼び出す他のスレッドは、以前に呼び出したスレッドに返す文字列を重ね書きします。

`ctime()` ルーチンをスレッド・セーフにするため、POSIX P1003.1c 規格では代替バージョン `ctime_r()` を定義して、それが追加の出力引数をとるようにしました。この引数は呼び出し側で割り当てられるユーザ提供のバッファです。`ctime_r()` 関数は、次のような ASCII 時間文字列をバッファに書き込みます。

```
char *ctime_r(const time_t *timer, char *buf);
```

この関数のユーザは、`buf` 引数によって占有されているストレージが、別のスレッドで使用されないように確認する必要があります。

- 内部状態の維持

この問題の一例は、`rand()` 関数です。

```
void srand(unsigned int seed);  
int rand(void);
```

この関数は、単純な擬似乱数ジェネレータです。`srand()` 関数で設定した任意の開始 `seed` 値に対して、擬似乱数の同一シーケンスを生成します。これを行うには、各呼び出しで更新された状態値を保持しておきます。別のスレッドでこの関数が呼び出されると、任意の開始シードに対して 1 つのスレッド内で返される数のシーケンスが、決定論的でなくなります。これは望ましくありません。

この問題を回避するため、2 番目のインタフェース `rand_r()` が POSIX 1003.1c で指定されました。このインタフェースは追加の引数を取り、`rand_r()` が乱数ジェネレータの状態を保持するために使用するユーザ提供の整数へのポインタを指定できるようにします。

```
int rand_r(unsigned int *seed);
```

この関数のユーザは、`seed` 引数が別のスレッドによって使用されることのないように確認する必要があります。スレッド固有データを使用することは、これを行う 1 つの方法です (12.4.1.2 項を参照)。

- スレッド間で共用する読み取り/書き込みデータ項目に対するオペレーション

読み取り/書き込みデータ共用の問題は、ミューテックスを使用することにより解決できます。この場合、ルーチンはリエントラントとはみなされませんが、スレッド・セーフです。スレッド固有のデータと同様、ミューテックス・ロックはルーチンのユーザに対して透過的です。

ミューテックスはいくつかの libc ルーチンで使用されますが、最も一般的なものは `stdio` ルーチン、たとえば、`printf()` です。`stdio` ルーチンのミューテックス・ロックはストリームによって行われますが、これは、2つのプロセスが同時に1つのストリーム・バッファに充てんしようとした場合に、ストリーム上での同時オペレーションが衝突するのを防ぎます。ミューテックス・ロックはまた、`fopen()` や `fclose()` などのオペレーション中に、C 実行時ライブラリにある特定の内部データ・テーブルに対しても行われます。これらのルーチンの代替バージョンは、アプリケーション・プログラミング・インタフェース (API) の変更が必要ないため、元のバージョンと同じ名前です。

ミューテックスの使用例については、12.4.3 項を参照してください。

12.4 スレッド・セーフ・コードの作成

シングル・スレッドおよびマルチスレッド・アプリケーションの両方で使用できるコードを作成するときには、スレッド・セーフな方法でコーディングする必要があります。次のコーディング方法に従ってください。

- 静的な読み取り/書き込みデータは、削除するか、スレッド固有のデータに変換するか、あるいはミューテックスにより保護する。

C 言語では、静的読み取り専用データを `const` 型修飾子で宣言すると、データの誤使用を減らすことができます。

- グローバルな読み取り/書き込みデータは、削除するか、あるいはミューテックス・ロックにより保護する。
- ファイル記述子のようなプロセスごとのシステム・リソースは、すべてのスレッドからアクセス可能であるため、注意して使用する。
- グローバルな `errno` セルへの参照は、`geterrno()` および `seterrno()` への呼び出しと置換する。

ソース・ファイルで `errno.h` が取り込まれ、次の条件のいずれかが当てはまる場合には、置換は必要ありません。

- `-pthread` オプション (`cc` または `c89` コマンド) を使用して、ファイルがコンパイルされている。
- ソース・ファイルの先頭で `pthread.h` ファイルが取り込まれている。
- `errno.h` ファイルを取り込む前に、`_REENTRANT` プリプロセッサ・シンボルが明示的に設定されている。
- 他のスレッド・セーフでないライブラリまたはオブジェクト・ファイルへの依存要素の発生を防止する必要がある。

12.4.1 スレッド固有データに対する TIS の使用

以下の項では、スレッド固有データに対して TIS (Thread Independent Services) を使用方法について説明します。

12.4.1.1 TIS の概要

TIS (Thread Independent Services) は、C 実行時ライブラリによって提供されるルーチンのパッケージであり、シングル・スレッドおよびマルチスレッド・アプリケーションの両方に対して、効率的なコードを作成するために使用されます。TIS ルーチンは、ミューテックスの処理、スレッド固有データの処理、およびその他のさまざまな目的で 사용할ことができます。

シングル・スレッド・アプリケーションで使用されると、これらのルーチンは単純化された意味規則を使用して、シングル・スレッド用のスレッド・セーフ・オペレーションを実行します。POSIX Threads Library が存在する場合は、ルーチン本体がより複雑なアルゴリズムで置き換えられて、マルチスレッド用に動作が最適化されます。

TIS を `libc` 自体の内部で使用する、1つのバージョンのC 実行時ライブラリが、シングル・スレッドおよびマルチスレッド・アプリケーションの両方で使用できるようになります。この機能の使用方法についての詳細は、『*Guide to the POSIX Threads Library*』および `tis(3)` を参照してください。

12.4.1.2 スレッド固有データの使用

例 12-1 は、シングル・スレッドおよびマルチスレッド・アプリケーションの両方で使用できる関数でスレッド固有のデータを使用する方法を示しています。簡潔にするため、ほとんどのエラー・チェックは省いています。

例 12-1: スレッド・プログラム例

```
#include <stdlib.h>
#include <string.h>
#include <tis.h>

static pthread_key_t key;

void __init_dirname()
{
    tis_key_create(&key, free);
}

void __fini_dirname()
{
    tis_key_delete(key);
}

char *dirname(char *path)
{
    char *dir, *lastslash;
    /*
     * Assume key was set and get thread-specific variable.
     */
    dir = tis_getspecific(key);
    if(!dir) { /* First time this thread got here. */
        dir = malloc(PATH_MAX);
        tis_setspecific(key, dir);
    }

    /*
     * Copy dirname component of path into buffer and return.
     */
    lastslash = strrchr(path, '/');
    if(lastslash) {
        memcpy(dir, path, lastslash-path);
        dir[lastslash-dir+1] = '\0';
    } else
        strcpy(dir, path);
    return dir;
}
```

次の TIS ルーチンが前述の例で使用されています。

tis_key_create

一意なデータ・キーを生成します。

`tis_key_delete`

データ・キーを削除します。

`tis_getspecific`

指定されたキーに関連するデータを取得します。

`tis_setspecific`

指定されたキーに関連するデータ値を設定します。

`__init_` および `__fini_` ルーチンは、この例ではスレッド固有のデータ・キーを初期化して破壊するために使用されています。このオペレーションは1度だけ行われ、これらのルーチンは、ライブラリが `dlopen()` でロードされている場合にも、このことを確実に示す便利な方法を提供します。`__init_` および `__fini_` ルーチンの使用方法についての説明は、`ld(1)` を参照してください。

スレッド固有のデータ・キーは、実行時に POSIX Threads Library によって提供される限定リソースです。多数のデータ・キーを使用する必要があるライブラリは、1つのデータ・キーだけを作成して、別々のデータ項目をすべて、構造体あるいは、そのキーで指し示されるポインタの配列として保存するようにライブラリをコーディングします。

12.4.2 TLS (Thread Local Storage) の使用

C コンパイラでは、TLS (Thread Local Storage) のサポートは常に有効になっています (cc コマンドの `-ms` オプションは不要です)。C++ では、TLS は `-ms` オプションを指定したときのみ認識され、指定していないときはエラーとして処理されます。

TLS は、マルチスレッド・プロセスのスレッドが存在する期間に静的エクステントを持ち (スタック上ではない)、スレッドごとに割り当てられるデータ領域です。

標準のマルチスレッド・プログラムでは、静的エクステント・データは、プロセスのすべてのスレッドで共有されますが、TLS は各スレッドごとに割り当てられ、各スレッドにはそれぞれ独自にデータのコピーがあり、スレッドがそのデータを変更しても、プロセス内の他のスレッドから見える値には影響を与えないようになっています。スレッドについての詳細は、『*Guide to the POSIX Threads Library*』を参照してください。

TLS の主要な機能は、POSIX (POSIX Threads Library) の `pthread_key_create()`、`pthread_setspecific()`、`pthread_getspecific()`、`pthread_key_delete()` のようなアプリケーション・プログラミング・インタフェース (API) によって提供されてきました。

これらの API は POSIX 準拠のプラットフォーム間での移植性がありますが、使いにくく間違いが起こりやすくなることがあります。また、適切な `static` および `extern` 変数宣言とその使用をすべて、スレッド・ローカル API の呼び出しに置き換えて、既存のシングルスレッド・コードをスレッド・セーフにするには、通常、かなりの技術的作業が必要になります。さらに、Windows-32 プラットフォームでは API のセット (`TlsAlloc()`、`TlsGetValue()`、`TlsSetValue()`、`TlsFree()`) が少し異なっており、POSIX API の場合と同じような使用上の問題があります。

これに対して、TLS の言語機能はいずれの API よりも使い方が簡単で、シングルスレッド・コードをマルチスレッド・コードに変換する際は特に便利です。これは、`static` または `extern` 変数がスレッド固有の値を持つように変更するには、宣言に記憶クラス修飾子を追加するだけでよいからです。コンパイラ、リンカ、プログラム・ローダ、デバッガは、この修飾子で宣言された変数に対して、複雑な API 呼び出しを自動的に効率良く実現します。API によるコーディングとは異なり、変数の使用をすべて探して変更したり、明示的に割り当ておよび割り当て解除コードを追加する必要はありません。この言語機能は、正式なプログラミング標準では一般に移植性はありませんが、Tru64 UNIX と Windows-32 プラットフォームの間では移植性があります。

12.4.2.1 `__thread` 属性

Tru64 UNIX の C および C++ コンパイラには、拡張記憶クラス属性、`__thread` が含まれます。

スレッド変数を宣言するには、`__thread` 属性を `__declspec` キーワードとともに使用しなければなりません。たとえば、次のコードは整数のスレッド・ローカル変数を宣言し、それを値で初期化しています。

```
__declspec( __thread ) int tls_i = 1;
```

12.4.2.2 ガイドラインと制限

スレッド・ローカルのオブジェクトおよび変数を宣言する際は、以下のガイドラインと制限を守らなければなりません。

12-10 スレッド・セーフなライブラリの開発

- 記憶クラス属性の `__thread` は、データの宣言および定義にのみ適用できます。関数の宣言や定義には使用できません。たとえば、次のコードではコンパイラ・エラーが発生します。

```
#define Thread __declspec( __thread )
Thread void func(); // Error
```

- `__thread` 属性は、記憶域の存続期間が静的なデータにのみ指定できます。これには、グローバル・データ・オブジェクト (static と extern の両方) と、ローカルな static オブジェクト、C++ クラスの static データ・メンバがあります。自動またはレジスタのデータ・オブジェクトには `__thread` 属性は宣言できません。たとえば、次のコードではコンパイラ・エラーが発生します。

```
#define Thread __declspec( __thread )
void func1()
{
    Thread int tls_i; // Error
}

int func2( Thread int tls_i ) // Error
{
    return tls_i;
}
```

- スレッド・ローカル・オブジェクトの宣言と定義では、その宣言と定義が同じファイルにあるか異なるファイルにあるかにかかわらず、`__thread` 属性を使用しなければなりません。たとえば、次のコードではエラーが発生します。

```
#define Thread __declspec( __thread )
extern int tls_i; // This generates an error, because the
int Thread tls_i; // declaration and the definition differ.
```

- `__thread` 属性は、タイプ修飾子としては使用できません。たとえば、次のコードではコンパイル時にエラーが発生します。

```
char __declspec( __thread ) *ch; // Error
```

- スレッド・ローカル・オブジェクトのアドレスは、リンク時の定数とは見なされず、このようなアドレスを含む式は定数式とは見なされません。標準 C では、静的またはスレッド・ローカルなエクステントを持つオブジェクトの初期値式として、スレッド・ローカル変数のアドレスを使用できないという影響があります。たとえば、ファイルの範囲で出現した場合、次のコードは C コンパイラではエラーになります。

```
#define Thread __declspec( __thread )
Thread int tls_i;
int *p = &tls_i; // ERROR
```

- 標準の C では、オブジェクトまたは変数を、自分への参照を含む式で初期化することが許されていますが、静的でないエクステンツのオブジェクトに限られています。通常、C++ では、自分への参照を含む式で動的にオブジェクトを初期化することが許されていますが、このような初期化は、スレッド・ローカル・オブジェクトでは許されていません。たとえば次のようになります。

```
#define Thread __declspec( __thread )
Thread int tls_i = tls_i;           // C and C++ error
int j = j;                          // Okay in C++; C error
Thread int tls_i = sizeof( tls_i ) // Okay in C and C++
```

現在初期化しようとしているオブジェクトを含む `sizeof` 式は、自分への参照とは見なされないため、C および C++ では許されることに注意してください。

12.4.3 スレッド間でデータを共用するためのミューテックス・ロックの使用

場合によっては、静的データをスレッド・セーフ・コードに変換するために、スレッド固有のデータを使用することは有効ではありません。たとえば、データ・オブジェクトがスレッド間で共用される (libc 内の `stdio` ストリームのように) 場合には、スレッド固有のデータは使用すべきではありません。プロセス毎のリソースの操作も、スレッド固有データが不適切な場合の例です。次の例は、スレッド・セーフな方法でプロセス毎のリソースを操作する方法を示しています。

```
#include <pthread.h>
#include <tis.h>

/*
 * NOTE: The putenv() function would have to set and clear the
 * same mutex lock before it accessed the environment.
 */

extern char **environ;
static pthread_mutex_t environ_mutex = PTHREAD_MUTEX_INITIALIZER;

char *getenv(const char *name)
{
    char **s, *value;
    int len;
    tis_mutex_lock(&environ_mutex);
    len = strlen(name);
    for(s=environ; value=*s; s++)
        if(strncmp(name, value, len) == 0 &&
```

12-12 スレッド・セーフなライブラリの開発

```

        value[len] == '=') {
            tis_mutex_unlock(&environ_mutex);
            return &(value[len+1]);
        }
    tis_mutex_unlock(&environ_mutex);
    return (char *) 0L;
}

```

この例では、環境にアクセスする前にロックが 1 度設定され (tis_mutex_lock)、リターンする前に 1 度だけロックが解除されている (tis_mutex_unlock) ことに注意してください。マルチスレッドの場合には、最初のスレッドがロックを保持している間に他のスレッドがその環境にアクセスしようとする、最初のスレッドがロック解除を実行するまで、他のスレッドはブロックされます。シングル・スレッドの場合には、ロックとロック解除のシーケンスにコーディング・エラーが存在しない限り、競合は起こりません。

マルチスレッド・アプリケーションで、ロック状態を fork() システム・コールの呼び出し中にも有効なままにしておく必要がある場合は、pthread_atfork() ハンドラ関数を作成および登録して、fork() 呼び出しの前にそのロックを設定し、fork() 呼び出しの後で子および親の両方でそのロックを解除する方法が有効です。これにより、別のスレッドがロックを保持している間に、別のスレッドがフォーク操作を行うことがなくなります。ロックが別のスレッドによって保持されている場合には、フォーク操作によって 1 つのスレッドだけを持つ子が作成されるため、子で永久にロックされることとなります。独立したライブラリの場合には、pthread_atfork() への呼び出しは、そのライブラリの __init_ ルーチンで行われます。ほとんどの Pthread ルーチンと異なり、pthread_atfork ルーチンは libc で使用可能であり、シングル・スレッドおよびマルチスレッド・アプリケーションの両方で使用することができます。

12.5 マルチスレッド・アプリケーションの作成

マルチスレッド・アプリケーションのコンパイルおよびリンクは、シングル・スレッド・アプリケーションのコンパイルおよびリンクとは多少異なります。以下の項では、この違いについて説明します。

12.5.1 マルチスレッド C アプリケーションのコンパイル

アプリケーションがシングル・スレッドかあるいはマルチスレッドかによって、多くのシステム・ヘッダ・ファイルは、アプリケーションのコンパイル

でインクルードされる際に、異なる定義のセットを提供します。コンパイラが、シングル・スレッドあるいはスレッド・セーフのどちらの動作を生成するかは、プリプロセッサ・シンボル `_REENTRANT` が定義されているかどうかによって決まります。cc または c89 コマンドに `-pthread` オプションを指定すると、`_REENTRANT` シンボルが自動的に定義されます。また、`pthread.h` ヘッダ・ファイルがインクルードされている場合も定義されます。Pthread ライブラリ `libpthread.so` を使用するアプリケーションでは、このヘッダ・ファイルを最初にインクルードする必要があります。

`-pthread` オプションは、C プログラムのコンパイルに対してはその他の影響は与えません。C コンパイラによって作成されるコードのリエントラント性は、特定のオプションではなく、プログラマによる適切なリエントラント・コーディングの使用、スレッド・セーフ・サポート・ルーチンおよび関数のみの使用によって決まります。

12.5.2 マルチスレッド C アプリケーションのリンク

マルチスレッド C アプリケーションをリンクする場合は、`-pthread` オプションを指定して cc または c89 コマンドを使用します。`-pthread` オプションは、リンク時に次の方法でライブラリ探索パスの修正に影響を与えます。

- Pthread ライブラリをリンクにインクルードする。
- 例外ライブラリをリンクにインクルードする。
- `-l` オプションで指定したライブラリに対しては、対応するスレッド・セーフ・ルーチンの名前の末尾に `_r` が付いたライブラリの配置および探索を行う。

`-pthread` オプションは、リンクの動作に対してはその他の影響は与えません。リンクされたコードのリエントラント性は、元のコードにおける適切なリエントラント・コーディングの使用、あるいは適切なヘッダ・ファイルあるいはライブラリによるコンパイルおよびリンクによって決まります。

12.5.3 その他の言語のマルチスレッド・アプリケーションの作成

すべてのコンパイラがリエントラント・コードを生成するとは限りません。言語によっては困難な場合もあります。また、アプリケーションにリンクされる実行時ライブラリがすべてスレッド・セーフであることも必要となります。詳細については、使用するコンパイラのマニュアルおよび実行時ライブラリのマニュアルを参照してください。

OpenMP 並列処理

Compaq C コンパイラは、OpenMP C APIに準拠した共用メモリ並列処理アプリケーションの開発をサポートします。この API は、コンパイラ指示文、ライブラリ関数、環境変数の集合を定義し、コンパイラ、リンカ、実行時環境に指示を与えて、アプリケーションの複数の部分を並行して実行できるようにします。

OpenMP 指示文を使用すると、複数のプロセッサで並行して実行できるコードを、通常の直列的な ANSI C のソース・コードの構造を変更することなく作成できます。これらの指示文を正しく使用すれば、マルチプロセッサ・マシンの別々のプロセッサでそのコードが同時に実行できるため、ユーザ・コードの経過時間に関する性能を大幅に向上できます。同じソース・コードをコンパイルするときに、並列処理の指示文を無視するようにすれば、OpenMP でのコンパイルと同じ機能を実行する直列的な C 言語プログラムになります。

『*OpenMP C and C++ Application Programming Interface*』仕様書は、インターネット上の <http://www.openmp.org/specs/> で参照できます。

この章では、次の項目について説明します。

- コンパイル・オプション (13.1 節)
- 環境変数 (13.2 節)
- 実行時性能のチューニング (13.3 節)
- プログラミング上の一般的な問題 (13.4 節)
- インプリメンテーション固有の動作 (13.5 節)
- デバッグ (13.6 節)

13.1 コンパイル・オプション

次の cc コマンド行オプションは、並列処理をサポートします。

-mp

コンパイラが OpenMP の手動分解プラグマと従来の手動分解指示文の両方を認識するようにしま

す。libots3 がリンクに含まれるようにします。
従来の手動分解指示文の詳細については、付録 D
を参照してください。

-omp	コンパイラが OpenMP の手動分解プラグマだけを認識し、従来の手動分解指示文を無視するようにします。従来の手動分解指示文の処理を除いて、-mp と -omp スイッチは同じです。つまり、-mp は従来の指示文を認識し、-omp は認識しません。
-granularity <i>size</i>	異なるスレッドから安全にアクセスできるメモリ内の共用データのサイズを制御します。 <i>size</i> の有効な値は、byte、longword、および quadword です。
byte	1 バイト以上のすべてのデータに、メモリ内のデータを共用する別々のスレッドからアクセスできるようにします。このオプションを使用すると、実行時の性能が低下します。
longword	自然に位置合わせされている 4 バイト以上のデータに、メモリ内のデータへのアクセスを共用する別々のスレッドから安全にアクセスできるようにします。3 バイト以下のデータ項目および位置合わせされていないデータにアクセスすると、複数のスレッドから書き込まれたデータ項目の更新に一貫性が保たれない場合があります。
quadword	自然に位置合わせされている 8 バイトのデータに、メモリ

内のデータを共用する別々のスレッドから安全にアクセスできるようにします。7 バイト以下のデータ項目および位置合わせされていないデータにアクセスすると、複数のスレッドから書き込まれたデータ項目の更新に一貫性が保たれない場合があります。これは、省略時の値です。

-check_omp

特定の OpenMP 構造の実行時検査が行われるようにします。これには、無効なネストおよびその他の無効な OpenMP の事例の実行時検出が含まれます。実行時に無効なネストが検出された場合にこのスイッチが設定されていると、実行可能プログラムは Trace/BPT トラップで異常終了します。このスイッチが設定されていないときに無効なネストが検出されると、その場合の動作は予測できません。たとえば、実行可能プログラムが停止することがあります。コンパイラでは、次の無効なネスト状態を検出します。

- ワーク・シェアリング構造、critical セクション、または master 内に、for、single、または sections 指示文を入れる。
- ワーク・シェアリング構造、critical セクション、または master 内で、barrier 指示文を実行する。
- ワーク・シェアリング構造内で、master 指示文を実行する。
- critical セクション内で、ordered 指示文を実行する。
- ordered for 内でないのに、ordered 指示文を実行する。

省略時の設定では，実行時検査は行われません。

13.2 環境変数

コンパイラおよび実行時システムは，OpenMP 仕様に概要が示されている環境変数に加えて，次の環境変数を認識します。

MP_THREAD_COUNT	実行時システムが作成するスレッドの数を指定します。省略時の設定は，プロセスで利用できるプロセッサの数です。OMP_NUM_THREADS 環境変数は，この変数に優先します。
MP_STACK_SIZE	各スレッドについて，実行時システムが割り当てるスタック空間のバイト数を指定します。0 を指定した場合，実行時システムでは非常に小さい，省略時の設定を使用します。したがって，プログラムで大きな配列を PRIVATE として宣言したときは，これらを割り当てられるだけの値を指定してください。この環境変数を使用しない場合，実行時システムは 5 MB を割り当てます。
MP_SPIN_COUNT	条件が真になるのを待つ間に，実行時システムが何回スピンするかを指定します。省略時の設定は 16,000,000 で，これは CPU 時間の約 1 秒に相当します。
MP_YIELD_COUNT	スレッド条件変数を待つスリープに入るまでに，実行時システムが sched_yield の呼び出しと条件の検査を何回交互に実行できるかを指定します。省略時の設定は 10 です。

13.3 実行時性能のチューニング

OpenMP 仕様では，parallel for 構造で，使用できるスレッドに処理を分散する各種の方法を用意しています。以降の各項では，これらの方法について説明します。

13.3.1 スケジュール・タイプとチャンクサイズの設定

スケジュール・タイプとチャンクサイズの設定の選択は、並列化されたアプリケーションの最終的な性能に良くも悪くも影響することがあります。スケジュール・タイプとチャンクサイズに不適切な設定を選択すると、並列化されたアプリケーションの性能が、直列化の場合と同じか、またはそれ以下に低下する可能性があります。

一般的な指針を次に示します。

- 一般的に、チャンクサイズは値が小さい方が処理速度が速くなります。チャンクサイズの値は、反復数を使用可能なスレッド数で割ることにより得られる値と同じか、またはそれ以下にしてください。
- スケジュール・タイプ `dynamic` および `guided` は、並列化されたアプリケーション以外の、各種負荷を抱えるターゲット・マシンに適した動作をします。これらのタイプでは、スレッドが使用可能になるときにスレッドに反復を割り当てます。他のアプリケーションが1つまたは複数のプロセッサを使用している場合は、使用可能なスレッドが次の反復を処理します。
- スケジュール・タイプ `runtime` は、スケジュール・タイプの実行時のチューニングを容易にしますが、実行時オーバーヘッドの点では若干の性能の低下が生じます。
- スケジュール・タイプとチャンクサイズが適切に設定されているかどうかを判断するには、スケジュール・タイプを `runtime` に設定したうえで、`OMP_SCHEDULE` 環境変数を使用してスケジュール・タイプとチャンクサイズの各種組み合わせを試してみる方法が効果的です。試行の後、最も良い性能が得られたスケジュール・タイプとチャンクサイズを明示的に設定します。

スケジュール・タイプとチャンクサイズの設定は、アプリケーションの性能に影響する数多くの要因のうちの2つに過ぎません。性能に影響を及ぼす可能性のあるその他の要因には、次のものがあります。

- システム・リソースの使用可能度: ターゲット・マシンで他のアプリケーションを処理している CPU は、並列化されたアプリケーションでは使用できません。
- 並列化されたコードの構造: 並列領域のスレッドが、不均衡な量の処理を実行している場合。

- 暗黙のまたは明示的なバリアの使用: このような明示的または暗黙のポイントですべてのスレッドの同期を強制する並列領域があると、1 つまたは複数のスレッドを待つ間、アプリケーションが一時停止する場合があります。
- `critical` セクションの使用と `atomic` 文の使用: `critical` セクションを使用すると、`atomic` よりもオーバーヘッドが大きくなります。スケジュール・タイプとチャンクサイズの設定の詳細については、『*OpenMP C and C++ Application Programming Interface*』仕様書を参照してください。

13.3.2 その他の制御

あるスレッドが、他のスレッドによって発生するイベントを待つ必要がある場合は、次の 3 段階のプロセスが開始されます。

1. そのスレッドは、特定の反復数だけスピンしながら、イベントの発生を待ちます。
2. プロセッサを何回も他のスレッドに譲りながら、イベントが発生していないかどうかを検査します。
3. 起こすように要求を送ってからスリープに入ります。

他のスレッドによりイベントが発生すると、スリープしているスレッドが起こされます。

環境変数 `MP_SPIN_COUNT` と `MP_YIELD_COUNT`、または `mpc_destroy` ルーチンを使用してスレッド環境をチューニングすると、性能が向上することがあります。

- `MP_SPIN_COUNT` — アプリケーションがスタンドアロンで実行される場合は、省略時の設定で良い性能が得られます。しかし、アプリケーションが他のアプリケーションとプロセッサを共用しなければならない場合は、`MP_SPIN_COUNT` の値を減らした方がよいでしょう。このことによりスレッドがスピンに浪費する時間が削減されて、プロセッサを明け渡す時間が早くなります。ただし、スレッドをスリープさせた後に再び起こすための時間がかかります。このような共用環境では、`MP_SPIN_COUNT` は 1000 程度に設定するとよいでしょう。
- `mpc_destroy` — 実行時に余分なスレッドが存在すると問題が起こる可能性がある処理 (`fork` など) を実行する場合は、`mpc_destroy` ルーチン

が役に立ちます。mpc_destroy ルーチンは、並列領域を実行するために作成されたワーカ・スレッドを破壊します。通常、mpc_destroy ルーチンは、並列領域の外からだけ呼び出します。mpc_destroy ルーチンは libots3 ライブラリに定義されています。

13.4 プログラミング上の一般的な問題

以降の各項では、並列化されたプログラムでよく発生するエラーについて説明します。

13.4.1 範囲指定

OpenMP の parallel 構造は、そのすぐ後の構造化ブロックに適用されます。複数の文を並列に実行する場合は、構造化ブロックを必ず中カッコ内に入れます。次に例を示します。

```
#pragma omp parallel
{
    pstatement one
    pstatement two
}
```

この構造化ブロックは、次に示す、OpenMP の parallel 構造が最初の文にのみ適用される構造化ブロックとは全く異なります。

```
#pragma omp parallel
    pstatement one
    pstatement two
```

中カッコを使用して後続のブロックの範囲を明示的に定義することを強くお勧めします。

13.4.2 デッドロック

どのマルチスレッド・アプリケーションにも言えることですが、プログラマは実行時のデッドロック状態を回避するよう注意する必要があります。多くの OpenMP 構造には最後に暗黙のバリアがあるため、すべてのスレッドがその構造にアクティブに加わらない場合、アプリケーションでデッドロックが生じます。このような状態は、アプリケーションの動的なエクステントを並列化する場合により多く生じます。次に例を示します。

```
worker ()
{
    #pragma omp barrier
}

main ()
{
```

```
#pragma omp parallel sections
{
    #pragma omp section
    worker();
}
```

この例では、すべてのスレッドが `worker` ルーチンに行くわけではないのに、バリアがすべてのスレッドを待つため、(複数のスレッドがアクティブなまま) デッドロックが生じます。このような状態の検出には、`-check_omp` オプション (13.1 節を参照) が役に立ちます。

有効および無効な指示文のネストの詳細については、『*OpenMP C and C++ Application Programming Interface*』仕様書を参照してください。

13.4.3 `threadprivate` ストレージ

`threadprivate` 指示文は、ファイルの有効範囲を持ちながらも各スレッドにプライベートな変数を識別します。スレッド数が一定である場合、これらの変数の値は維持されます。プログラム内でスレッド数を明示的に増減したときの `threadprivate` 変数の値への影響は定義されていません。

13.4.4 ロックの使用

ロック制御ルーチン (『*OpenMP C and C++ Application Programming Interface*』仕様書を参照) を使用するには、特定の順序で呼び出す必要があります。

1. まず、ロック変数と関連付けるロックを初期化します。
2. 関連付けられたロックが実行スレッドで使用できるようにします。
3. 実行スレッドのロックの所有を解除します。
4. 終了したら、ロックとロック変数の関連付けを解除します。

これ以外の順序でロックを使用しようとすると、デッドロック状態などの予期しない動作が生じることがあります。

13.5 インプリメンテーション固有の動作

OpenMP 仕様では、いくつかの機能と省略時の値をインプリメンテーション固有のものとしています。この節では、このような事例と Compaq C で選択されているインプリメンテーションを示します。

ネストされた並列領域に対するサポート

ネストされた並列領域があると、1つのスレッドで構成されるチームがその領域の実行のために作成されます。

OMP_SCHEDULE の省略時の値

省略時の値は `dynamic,1` です。この値は、アプリケーションが実行時スケジュールを使用するときに、OMP_SCHEDULE が定義されていない場合に使用されます。

OMP_NUM_THREADS の省略時の値

省略時の値は、マシンのプロセッサの数と同じです。

OMP_DYNAMIC の省略時の値

省略時の値は 0 です。このインプリメンテーションでは、スレッド・カウントの動的な調整はサポートされません。 `omp_set_dynamic` を 0 以外の値で使用しようとしても、実行時環境には何も作用しません。

省略時のスケジュール

`for` または `parallel for` ループにスケジュール句が含まれていない場合、スケジュール・タイプには DYNAMIC が使用され、チャンクサイズは 1 に設定されます。

`flush` 指示文

`flush` 指示文は、指示文に 1 つまたは複数の変数が指定されていても、すべての変数をフラッシュします。

13.6 デバッグ

以下の節には、OpenMP アプリケーション・プログラミング・インタフェース (API) を使用するアプリケーションの動作を診断し、デバッグする方法についてのヒントを示します。

13.6.1 デバッグに必要な背景知識

`-mp` または `-omp` オプションを使用すると、コンパイラは OpenMP 指示文を認識し、コードの指定された部分を並列処理リージョンに変換します。コンパイラは、リージョン内のコードを取り出し、それをコンパイラが作成した別のルーチンに入れることで、並列処理リージョンを実現します。この処理

は、ルーチンが呼び出された所でソース・コードをルーチンに入れるインライン化の逆なので、アウトライン化と呼ばれます。

注意

デバッガと他のアプリケーション分析ツールを効率良く使用するためには、並列処理リージョンをアウトライン化する方法を理解しなければなりません。

並列処理リージョンの場所に、コンパイラは実行時ライブラリ・ルーチンへの呼び出しを挿入します。実行時ライブラリ・ルーチンは、チームの中にスレーブ・スレッドを作成し(まだ作成されていない場合)、チーム内のすべてのスレッドを開始し、アウトライン・ルーチンを呼び出します。アウトライン・ルーチンからスレッドが戻ると、スレッドは実行時ライブラリに戻ります。ライブラリはすべてのスレッドが完了するまで待ち、その後マスタ・スレッドが呼び出し元のルーチンに戻ります。マスタ・スレッドが非並列実行を継続している間、スレーブ・スレッドは、新しい並列処理リージョンに遭遇するか、または環境変数によって決まる待ち時間(MP_SPIN_COUNT)が経過するまで待機(つまりスピン)します。待ち時間が経過すると、スレーブ・スレッドは次に並列処理リージョンに遭遇するまでスリープ状態になります。

次のソース・コードに含まれる並列処理リージョンでは、変数 *id* は各スレッドにプライベートです。並列処理リージョンの前にあるコードは、並列処理リージョンで使用するスレッドの数を明示的に2としています。次に、並列処理リージョンは、実行しているスレッドのスレッド番号を取得し、それを `printf` 文で表示します。

```
1
2 main()
3 {
4     int id;
5     omp_set_num_threads(2);
6     # pragma omp parallel private (id)
7     {
8         id = omp_get_thread_num();
9         printf ("Hello World from OpenMP Thread %d\n", id);
10    }
11 }
```

`dis` コマンドを用いて、上記のソース・コードから生成されたオブジェクト・モジュールを逆アセンブルすると、次のような結果が出力されます。

```
__main_6: 1
0x0: 27bb0001 ldah gp, 1(t12)
0x4: 2ffe0000 ldq_u zero, 0(sp)
```



```

0x8: 23bd8110 lda gp, -32496(gp)
0xc: 2ffe0000 ldq_u zero, 0(sp)
0x10: 23defff0 lda sp, -16(sp)
0x14: b75e0000 stq ra, 0(sp)
0x18: a2310020 ldl a1, 32(a1)
0x1c: f620000e bne a1, 0x58
0x20: a77d8038 ldq t12, -32712(gp)
0x24: 6b5b4000 jsr ra, (t12), omp_get_thread_num
0x28: 27ba0001 ldah gp, 1(ra)
0x2c: 47e00411 bis zero, v0, a1
0x30: 23bd80e8 lda gp, -32536(gp)
0x34: a77d8028 ldq t12, -32728(gp)
0x38: a61d8030 ldq a0, -32720(gp)
0x3c: 6b5b4000 jsr ra, (t12), printf
0x40: 27ba0001 ldah gp, 1(ra)
0x44: 23bd80d0 lda gp, -32560(gp)
0x48: a75e0000 ldq ra, 0(sp)
0x4c: 63ff0000 trapb 0x50: 23de0010 lda sp, 16(sp)
0x54: 6bfa8001 ret zero, (ra), 1
0x58: 221ffff4 lda a0, -12(zero)
0x5c: 000000aa call_pal gentrap
0x60: c3ffffef br zero, 0x20
0x64: 2ffe0000 ldq_u zero, 0(sp)
0x68: 2ffe0000 ldq_u zero, 0(sp)
0x6c: 2ffe0000 ldq_u zero, 0(sp)
main:
0x70: 27bb0001 ldah gp, 1(t12)
0x74: 2ffe0000 ldq_u zero, 0(sp)
0x78: 23bd80a0 lda gp, -32608(gp)
0x7c: 2ffe0000 ldq_u zero, 0(sp)
0x80: a77d8020 ldq t12, -32736(gp)
0x84: 23defff0 lda sp, -16(sp)
0x88: b75e0000 stq ra, 0(sp)
0x8c: 47e05410 bis zero, 0x2, a0
0x90: 6b5b4000 jsr ra, (t12), omp_set_num_threads
0x94: 27ba0001 ldah gp, 1(ra)
0x98: 47fe0411 bis zero, sp, a1
0x9c: 2ffe0000 ldq_u zero, 0(sp)
0xa0: 23bd807c lda gp, -32644(gp)
0xa4: 47ff0412 bis zero, zero, a2
0xa8: a77d8010 ldq t12, -32752(gp)
0xac: a61d8018 ldq a0, -32744(gp)
0xb0: 6b5b4000 jsr ra, (t12), _OtsEnterParallelOpenMP [2]
0xb4: 27ba0001 ldah gp, 1(ra) : a75e0000 ldq ra,
0(sp)
0xbc: 2ffe0000 ldq_u zero, 0(sp)
0xc0: 23bd805c lda gp, -32676(gp)
0xc4: 47ff0400 bis zero, zero, v0
0xc8: 23de0010 lda sp, 16(sp)
0xcc: 6bfa8001 ret zero, (ra), 1

```

① `__main_6` は、リストの 6 行目の、ルーチン `main` で始まる並列処理リージョンに対して、コンパイラが作成したアウトライン・ルーチンです。コンパイラが生成するアウトライン・ルーチンの名前は、次のような形式になっています。

`__original-routine-name_listing-line-number`

② `_OtsEnterParallelOpenMP` への呼び出しがコンパイラによって挿入され、スレッド作成と並列処理リージョンの実行の整合がとられます。

実行の制御は、すべてのスレッドが並列処理リージョンを終了するまで、`_OtsEnterParallelOpenMP` の内部に留まります。

13.6.2 デバッグおよびアプリケーション分析のツール

OpenMP アプリケーションをデバッグするための主要なツールは Ladebug デバッガです。その他のツールには、Visual Threads、Atom ツールの `pixie` と `third`、および OpenMP ツール `ompc` があります。

13.6.2.1 Ladebug

この項では、OpenMP アプリケーションで Ladebug デバッガを使用する方法について説明します。ここでは、従来のマルチスレッド・アプリケーションと比較して、OpenMP アプリケーションに特有の事項について説明します。13.6.1 項でのプログラム例を使用して、OpenMP アプリケーションのデバッグのコンセプトを示します。マルチスレッド・プログラムのデバッグについての詳細は、『*Ladebug Debugger Manual*』を参照してください。

OpenMP アプリケーションはマルチスレッドなので、通常のマルチスレッド・プログラムの場合と同じ方法でデバッグできます。ただし、考慮すべきことがいくつかあります。

- 最適化されたコードでは、コンパイラはソース・モジュールを変更して OpenMP をサポートできるようにしています。したがって、デバッガに表示されるソース・モジュールでは、プログラムの実際の実行が反映されません。たとえば、コンパイラが実行するアウトライン化処理で生成されるルーチンは、独立したルーチンとしては表示されません。デバッグ・セッションの前には、出力リストまたはオブジェクト・モジュールの逆アセンブルにより、これらのルーチンの名前がわかります。これらのルーチンは、通常のルーチンと同じように Ladebug セッションで分析できます。
- OpenMP 標準では、スレッド 0 (マスタ・スレッド) から始まるスレッド番号を定義しています。Ladebug は OpenMP のスレッド番号は扱いません。代わりに、番号が 1 から始まる `pthread` でのスレッド番号を扱います。
- OpenMP スレーブ・スレッドの呼び出しスタックは、`thdBase` という `pthread` ライブラリ・ルーチンから始まり、`slave_main` という `libots3` のルーチンまで続きます。

- 並列処理リージョンにプライベートな変数は、各スレッドにプライベートです。明らかにプライベートな変数 (firstprivate, lastprivate, private または reduction で修飾) には、スレッドごとに異なるメモリ位置があります。

並列処理リージョンをデバッグする場合、アウトライン・ルーチン名にブレークポイントを設定します。次の例は、Ladebug セッションの開始、並列処理リージョンへのブレークポイントの設定、実行の継続を示しています。ユーザ・コマンドは脚注で説明しています。

```
> ladebug example 1
Welcome to the Ladebug Debugger Version 4.0-48
-----
object file name: example
Reading symbolic information ...done
(ladebug) stop in __main_6 2
[#1: stop in void __main_6(int, int) ]
(ladebug) run 3
[l1] stopped at [void __main_6(int, int):6 0x1200014e0]
      6 # pragma omp parallel private (id)
(ladebug) thread 4
Thread Name          State      Substate    Policy      Pri
-----
>*> 1 default thread  running
(ladebug) cont 5
Hello World from OpenMP Thread 0
[l1] stopped at [void __main_6(int, int):6 0x1200014e0]
      6 # pragma omp parallel private (id)
(ladebug) thread 6
Thread Name          State      Substate    Policy      Pri
-----
>*> 2 <anonymous>    running
(ladebug) cont 7
Hello World from OpenMP Thread 1
Process has exited with status 0
```

- 1 例のアプリケーションを指定して Ladebug セッションを開始します。
- 2 アウトライン・ルーチン __main_6 の開始点で止めるためにブレークポイントを設定します。
- 3 プログラムを起動します。 __main_6 の先頭で制御が止まります。
- 4 並列処理リージョンをアクティブに実行しているスレッドを表示します (この例では pthread 1, OpenMP スレッド 0)。
- 5 この地点から実行を継続し、再びブレークポイントで止まる前に、OpenMP スレッド 0 の並列処理リージョンが終了して、適切な OpenMP スレッド番号で「Hello World」というメッセージが出力されることを確認します。

- ⑥ 並列処理リージョンをアクティブに実行している次のスレッドを表示します (pthread 2, OpenMP スレッド 1)。
- ⑦ この地点から実行を継続すると、次のメッセージが出力され、プログラムの実行が終了します。

次の例では、pthread 2 (OpenMP スレッド 1) が並列処理リージョンの実行を開始したときに、アウトライン・ルーチンの先頭にブレークポイントを設定する方法を示します。

```
> ladebug example
Welcome to the Ladebug Debugger Version 4.0-48
-----
object file name: example
Reading symbolic information ...done
(ladebug) stop thread 2 in __main_6
[#1: stop thread (2) in void __main_6(int, int) ]
(ladebug) r
Hello World from OpenMP Thread 0
[1] stopped at [void __main_6(int, int):6 0x1200014e0]
      6 # pragma omp parallel private (id)
(ladebug) thread
  Thread Name      State      Substate      Policy      Pri
  -----
>*      2 <anonymous>      running      SCHED_OTHER  19
(ladebug) c
Hello World from OpenMP Thread 1
Process has exited with status 0
```

- ① 並列処理リージョンの開始点で OpenMP スレッド 1 (pthread 2) を止めます。

OpenMP 組込みのワーク・シェアリング構造 (for および sections 指示文) のデバッグは、前の例と同じように行います。

注意

Ladebug デバッガは、OpenMP デバッグをまだ完全にはサポートしていません。threadprivate として宣言された変数は、Ladebug では認識されず、表示できません。

13.6.2.2 Visual Threads

OpenMP が組み込まれたプログラムは、Compaq Visual Threads (dxthreads) 製品でモニタリングできます。この製品は、「Associated Products Volume 1」CD-ROM にあります。詳細は、Visual Threads のオンライン・ヘルプを参照してください。

13.6.2.3 Atom および OpenMP ツール

OpenMP アプリケーションは、OpenMP アプリケーションをモニタリングするために作成された特殊なツールの `ompc`、Atom ベースのツール `pixie` (実行のプロファイル用)、Third Degree (メモリ・アクセスとリークをモニタリングする `third`) を用いて計測できます。

`ompc` ツールは、関連する環境変数設定をキャプチャし、OpenMP に関する実行時ライブラリ・ルーチンの呼び出しをカウントします。開発者の意図しない状況では、警告およびエラー・メッセージを生成して注意を促します。最後に、環境変数の設定に基づいて、これらの実行時ライブラリ・ルーチンへの呼び出しをすべてトレースし、ルーチンが呼び出された順番を (OpenMP スレッド番号で) 通知します。詳細は `ompc(5)` を参照してください。

Atom ベースの `pixie` ツールは、アプリケーションでの効率の良くないスレッド使用を検出するために使用します。13.6.1 項で説明したように、スレーブ・スレッドは、新しい並列処理リージョンの実行が始まるか、または `MP_SPIN_COUNT` が経過するまで待機 (つまりスピン) します。アプリケーションの並列処理リージョン間の時間が長い場合、スレッドはスリープ状態になるまでスピンします。`pixie` でアプリケーションを計測すると、アプリケーションがどこで多くの時間を費やしているかを確認できます。アプリケーションが `slave_main` で CPU 時間を大量に費やしている場合は、スレッドがスピンに費やす時間が長すぎることを示しています。このようなアプリケーションでは、`MP_SPIN_COUNT` の値 (省略時の値は 16000000) を小さくすると、全体の性能が良くなります。`pixie` についての詳細は、第 8 章と `pixie(1)` を参照してください。

Third Degree ツールについては、第 7 章と `third(1)` を参照してください。

13.6.2.4 その他のデバッグ支援機能

その他のデバッグ支援機能には、次のようなものがあります。

- コンパイル時オプションの `-check_omp`。デッドロックや競合条件を検出するための実行時チェックが組み込まれる (13.1 節を参照)。
- プログラム内のアクティブなスレッド数を変更可能にする
`omp_set_num_threads` および `mpc_destroy` 関数。

`omp_set_num_threads` または `mpc_destroy` を呼び出すと、プログラム内でアクティブなスレッドの数を変更できます。いずれの場合も、

`threadprivate` で宣言されたデータや、スレーブ・スレッドに関するデータは、アプリケーションの起動時の値で再度初期化されます。たとえば、アクティブなスレッドの数が 4 のときに、`omp_set_num_threads` を呼び出してこの数を 2 に設定すると、OpenMP スレッドの 1, 2, 3 に対応する `threadprivate` データがリセットされます。マスタ・スレッド (OpenMP スレッド 0) に対応する `threadprivate` データは変更されません。

`mpc_destroy` についての詳細は、13.3.2 項を参照してください。

EVM イベントの発信と受信

この章では、Tru64 UNIX Event Manager (EVM) のユーザ・レベル・プログラミング・インタフェースについて説明します。この章に記載した主な問題は次のとおりです。

- イベントとして扱う状態の変化の決定方法
- イベントの内容の設計
- EVM の API 関数とユーティリティを使用した、イベントの発信、登録、およびアクセス

この章では、イベントとは、次のいずれかが関心を持つ可能性のある現象が起きたことを示します。

- システム管理者、アプリケーション管理者、またはその他のユーザ
- システム監視ソフトウェア
- オペレーティング・システム
- アプリケーション・プログラム

これらは、ローカル・システムまたはリモート・システムに存在します。

この章では、ユーザ・レベルのイベント処理についてのみ説明します。カーネル内部でのイベントの発信と登録については、`kevm(7)` を参照してください。

この章では、以下の項目について説明します。

- イベントとイベント管理 (14.1 節)
- EVM イベント処理の概要 (14.2 節)
- EVM の起動と停止 (14.3 節)
- イベントの発信とアクセスの権限 (14.4 節)
- EVM イベントの内容 (14.5 節)
- イベント・セットの設計 (14.6 節)

- EVM プログラミング・インタフェース (14.7 節)
- EVM へのイベント・チャンネルの追加 (14.8 節)

EVM プログラミング・インタフェースがサポートする関数の詳細については、オンライン・リファレンス・ページに記載しています。使用頻度の高い EVM 関数の概要と例については 14.7 節 を参照してください。

イベント・ビューアとコマンド行インタフェースについては、この章では説明していません。ビューアについては、オンライン・ヘルプを参照してください。コマンド行での操作については、『システム管理ガイド』マニュアルを参照してください。

14.1 イベントとイベント管理

EVM では、イベント情報の発信、配信、保存、および表示を集中管理することができます。このとき、各イベント発信者が使用するイベント・チャンネルには依存せず、発信者と現在のチャンネルの関係を変更する必要もありません。EVM を使用すると、従来のバージョンの Tru64 UNIX システムと比較して、システム管理者は簡単にイベント情報にアクセスできます。また、柔軟なインフラストラクチャが提供されているので、次の配信元でイベント配信チャンネルとして使用することができます。

- Tru64 UNIX の開発グループ
- 独立ソフトウェア・ベンダ
- ユーザ・アプリケーションの開発者
- その他のイベント・チャンネル

イベント情報を渡すためのメカニズムは、イベント (またはイベント通知) と呼ばれ、イベントを生成するコンポーネントはイベント発信者 (**event poster**) と呼ばれます。EVM のイベント発信メカニズムは、単方向の通信チャンネルです。これは、発信者が、イベントのアクセスを希望する任意エンティティに対して情報を通信できるようにしたものです。発信者は、どのエンティティが発信イベントのアクセスを希望しているかを把握している必要はありません。

イベント情報を受信するエンティティは、イベント受信者 (**event subscriber**) と呼ばれます。イベントによって、受信者の種類は異なり、システム管理者や、他のソフトウェア・コンポーネント、または一般ユー

ザなどが含まれることがあります。また、イベントによっては受信者が存在しないこともあります。

イベントは、任意のプロセスで発信および受信を行うことができます。また、同一のプロセスで発信と受信の両方を行うこともできます。ただし、特定のイベントの発信および受信は、セキュリティ権限によって管理されます (14.4 節)。

Tru64 UNIX システムの以前のバージョンでは、イベント処理のためにさまざまな種類のチャンネルがサポートされており、標準のものもあれば専用のものもありました。最も単純なイベント・チャンネルは、静的な ASCII ログ・ファイルです。このログ・ファイルには、単一のソースからのイベント情報が格納され、ユーザは標準の UNIX ツール (`more` など) を使用して表示できます。動的なチャンネルとしては、システム・ロガー (`syslog`) やバイナリ・イベント・ロガー (`binlog`) などがあります。どちらのチャンネルでも、デーモン・プロセスを使用して、複数のソースに関するイベント情報の受信、記録、および転送を行います。`syslog` と `binlog` についての詳細は、`syslogd(8)` および `binlogd(8)` を参照してください。

EVM では、複数のイベント・チャンネルを集中管理するために、すべてのソースのイベントが 1 つのイベント・ストリームに結合されます。イベントに関心のあるユーザは、結合されたストリームをリアル・タイムで監視したり、イベントの履歴をストレージから取得して表示したりできます。EVM の表示機能には、グラフィカルなイベント・ビューアと、完全なセットのコマンド行ユーティリティがあります。イベント・ビューアは、SysMan アプリケーションに統合されています。コマンド行ユーティリティでは、イベントのフィルタ、ソート、およびフォーマットをさまざまな方法で行うことができます。選択した状態が自動的に通知されるように EVM を構成することもできます。

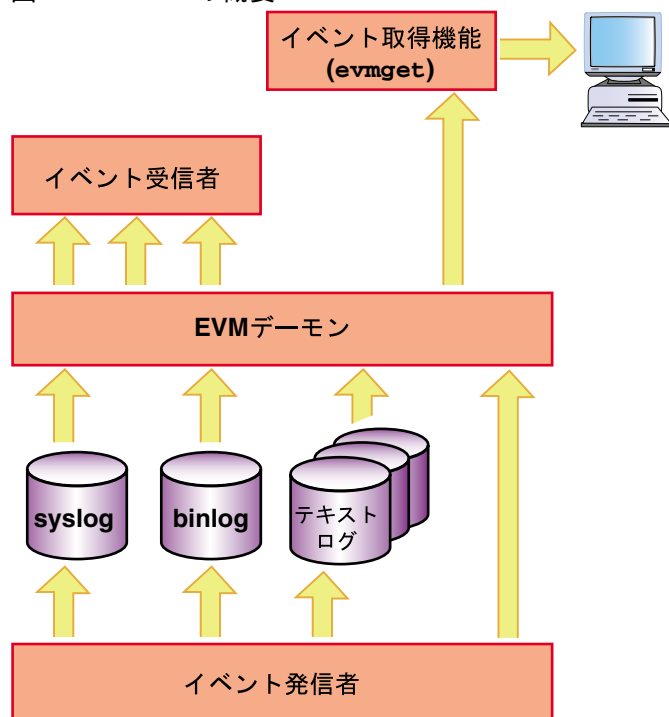
注意

EVM は、メッセージをブロードキャストするための機能です。2 つのプロセス間に二地点間専用通信チャンネルをインプリメントする場合には使用しないでください。このような目的で EVM を使用すると、システムの性能が低下する可能性があります。別のプロセスと通信を確立する必要があるときに、送信する情報がシステ

ム管理者やその他のプロセスに関係がない場合は、ソケットまたはパイプなどの通信チャネルを使用して直接接続してください。

図 14-1 は、発信、受信、および取得操作の概要です。イベント取得操作については、`evmget(1)` を参照してください。

図 14-1: EVM の概要



ZK-1458U-AIJ

14.2 EVM イベント処理の概要

EVM イベントはデータのパッケージであり、さまざまなソフトウェア・コンポーネント間で受渡したり、後で表示するためにデータを保存できます。

イベントを発信するためには、EVM データベースに保存されているイベント・テンプレートと一致している必要があります。イベントのデータ項目は、発信するイベントまたは一致するテンプレートに設定できます。発信したイベントのデータ項目とイベント・テンプレートのデータ項目をマージした結果が、イベント受信者が受信するイベントの内容になります。イベント

14-4 EVM イベントの発信と受信

の一致およびマージについての詳細は、14.6.3.2 項 および 14.6.3.3 項を参照してください。

標準的なイベントのライフ・サイクルには、次の操作が含まれます。

1. プロセスから発信される可能性のあるすべてのイベントに対して、テンプレートが作成されます。テンプレートは、製品またはサブシステムのインストール時に EVM データベースに格納されます。
2. イベントの受信に関係するプロセスでは、EVM デーモンに対する接続が確立されて、受信要求が発行されます。受信要求には、関心のあるイベントの識別に使用するフィルタが指定されています。
3. プロセスまたはカーネル構成要素でイベントに値する状態変更が検出されると、EVM への接続が行われて、状態変更に対応するイベントが発信されます。

また、EVM 以外のイベント・チャンネルでイベントが発信された場合、そのイベントはチャンネルによって独自の方法で処理されます (ログの記録など)。次に、EVM フォーマットに変換されて、EVM に渡されます。

4. EVM では、イベントの発信要求の有効性が検査されます。特に、EVM テンプレート・データベースに対応するテンプレートが存在するかどうかと、発信者にイベントを発信する権限があるかどうかを検査されます。有効である場合は、発信されたイベントとテンプレートのデータ項目がマージされたイベントが作成されます。
5. マージされたイベントが、EVM からイベントを受信するすべてのプロセスに渡されます。
6. 受信者によって、イベントが適切に処理されます。たとえば、保存や、システム管理者へのメール送信、アプリケーションのフェールオーバーの開始などが行われます。

14.3 EVM の起動と停止

EVM は、システムのスタートアップ時に自動的に起動し、システムのシャットダウン時に自動的に停止します。

EVM デーモンの起動と停止については、『システム管理ガイド』を参照してください。

起動時に EVM がイベント・テンプレート・データベースを設定する手順，および起動後にイベント・テンプレート・データベースを修正する方法については，14.6.3.4 項を参照してください。

14.4 イベントの発信とアクセスの権限

イベントの発信とアクセスを行う場合は，セキュリティに関して考慮することが重要です。

- 特定のイベント情報に対するアクセスを管理しなかった場合，システム操作に関する重要な情報が，その権限のないユーザに与えられることがあります。
- 特定のイベントの発信を管理しなかった場合，システムのシャットダウンのような重要なシステム操作が，その権限のないユーザによって実行されるおそれがあります。

システム管理者は，EVM 権限ファイル `/etc/evm.auth` を変更することによって，どのユーザがイベントのアクセスや発信を行えるかということを制御できます。

アクセス制御についての詳細は，『システム管理ガイド』マニュアルを参照してください。

14.5 EVM イベントの内容

イベントによって送信されるデータのフォーマット，タイプ，および目的には，さまざまな種類があります。単純なイベントの場合は，単純なテキスト・メッセージです。複雑なイベントの場合は，複雑なバイナリ・データの集合で構成されます。この場合，データの変換時にフォーマット情報が必要になります。2つのアプリケーションが協調して動作する場合は，簡単なバイナリ・フラグまたはカウントの場合もあります。ただし，イベントにとってデータは必須ではありません。たとえば，特定のイベント・タイプを送信するだけで，特定の状態変更が発生したことを受信者が認識できることもあります。

イベント・データは，変換方法に応じて複雑になります。イベントをわかりやすくするために，数値データと説明テキストを組み合わせることもできます。また，受信プロセスによっては，バイナリ・データとして表示する必要があります。さらに，複数の言語で表示しなければならないこともあります。

EVM のデータ・メカニズムでは、イベントに適した任意の方法でデータを送信できます。この機能を使用できる構造体には、EVM フォーマット・データ項目 (EVM の標準データ項目の 1 つ) および EVM 変数データ項目があります。

イベント・データの構造体には、次の 2 種類のデータ項目があります。

- 標準データ項目 — 定義済みの名前を持った、一定数の項目
- 変数データ項目 — 多くの場合、イベントの設計者が名前と型を定義する

イベントを作成する場合は、任意の数のデータ項目を含めることができます。イベントを発信すると、ホスト名やタイムスタンプなど、一定の環境標準データ項目が EVM によって自動的に追加されます。

14.5.1 標準データ項目

標準データ項目は、イベントで必須の共通データ項目で、EVM によって認識または処理されます。標準データ項目の名前は、列挙型定数として指定されています。したがって、名前自体の領域は、イベントの主な領域ではありません。

一部の標準データ項目は、イベントを発信するアプリケーションまたはイベントのテンプレートによってイベントに挿入されます。その他のデータ項目は、必要に応じて、EVM コンポーネントにより、自動的に挿入されます。任意のデータ項目をイベントから抽出することができます。

標準データ項目は、`EvmEvent(5)` で説明しています。特に注意が必要な項目は、表 14-1 にリストし、以下の節で説明しています。

表 14-1: 標準データ項目

データ項目	説明
イベント名 (14.5.1.1 項)	イベントの名前を指定する。
フォーマット (14.5.1.2 項)	イベントのメッセージ文字列を指定する。
優先度 (14.5.1.3 項)	受信者にとってのイベントの重要度。イベント配信の順序には影響しない。
I18N カタログ (14.5.1.4 項)	多国語対応イベント用の I18N カタログ・ファイルの名前。
I18N メッセージ・セット ID (14.5.1.4 項)	I18N メッセージ・カタログ内のメッセージ・セットを識別する。

表 14-1: 標準データ項目 (続き)

データ項目	説明
I18N メッセージ ID (14.5.1.4 項)	イベント・フォーマットの I18N メッセージ ID。
クラスタ・イベント (14.5.1.5 項)	TruCluster 環境では、この項目によりイベントはクラスタ内の全ノードに配信される。
参照 (14.5.1.6 項)	イベントの説明テキストに対する参照。

以降の各項で、表 14-1 の標準データ項目について詳しく説明します。

14.5.1.1 イベント名データ項目

イベント名データ項目は、イベントを識別する文字列で、通常はイベントの発信元と何が起こったかを知らせます。イベント名は、イベントの型を一意に識別するので、同じイベントの異なるインスタンスに同じ名前を付けることができます。

EVM は、特定のイベントを発信またはアクセスする権限がユーザにあるかどうかを判断する場合や、イベントに対応するテンプレート情報を検索する場合に、イベント名を使用します。クライアント・アプリケーションは、イベント名をイベント・フィルタと組み合わせて使用し (14.7.11 項)、受信するイベントを選択したり、イベントの受信時に必要な動作を判断したりします。システム管理者は、イベント名を使用して、特に関心のあるイベントのイベント・ログを検索できます。

イベント名には次のような特徴があります。

- 名前は、1 つ以上のコンポーネントをドットで区切って続けて記述する。
- コンポーネントは英字、数字、下線を順不同で 1 つ以上使用して構成する。
- 名前に含まれているコンポーネントの数に制限はない。
- 名前の先頭や末尾にドットを使用することはできない。

コンポーネントを 3 つ以上含んだ名前を持たないイベントは、発信できません。

イベント名のコンポーネントは、最初 (左端) のコンポーネントで始まるイベントの階層構造を表します。最初のコンポーネントは最も一般的で、イベントの発信を行うエンティティをグローバルなレベルで区別し

ます。その他のコンポーネントは、レベルの詳細さが増していく順番に並べます。まずイベントを発信すべきと判断したコンポーネント、何が起こったかという情報、最後に最も詳細な情報を置きます。たとえば、`sys.unix.fs.filesystem_full` というイベント名があるとします。

- `sys.unix` は、イベントがオペレーティング・システムのコンポーネントから発信されたシステム・イベントであることを示します。
- `fs` は、イベントがファイル・システム・コンポーネントから発信されたことを示します。
- `filesystem_full` は、報告している出来事が、ファイル・システムの満杯であることを示す。

何が起こったかを示す基本イベント名は、イベントが発信されるときにコンポーネントを追加して拡張し、より詳しい情報を示すようにすることができます。前述したイベント名の例では、追加のコンポーネントとしてファイル・システム名を追加して(`sys.unix.fs.filesystem_full.usr` など)、どのファイル・システムが満杯になったかという情報による拡張が行えます。名前を拡張しても、短い方の名前の各コンポーネントはすべて拡張した名前に一致するので、このイベントが `sys.unix.fs.filesystem_full` イベントであることに変わりはありません。`sys.unix.fs.filesystem_full` という名前のイベントを検索すると、拡張した名前のイベントも見つかります。

この命名規則では、オープンエンドな方法でイベントを識別できます。つまり、どのレベルでも詳細な情報を記述できるようになっています。注意深く命名すると、受信者が、特定のイベントやイベント・クラスを選んで表示したり監視したりするのが容易になります。イベント名を詳しく指定すると、指定する基準も正確になります。たとえば、イベント名を `myco.myprod.env.temp.ok` および `myco.myprod.env.temp.high` とすると、システム管理者は `myco.myprod.env.temp` と指定して温度に関するイベントをすべて監視したり、`myco.myprod.env.temp.high` と指定して、監視するイベントを高温に関するイベントのみに制限したりすることができます。

システム管理者やモニタリング・ソフトウェアの混乱や誤動作（誤操作）を避けるため、新しいイベントに一意の名前を付けるようにしてください。次の規則に従って命名すれば、すでに使用されている名前との重複を避けることができます。

- サードパーティの製品またはアプリケーションが発信するイベントでは、名前のコンポーネントのうち最初の3つが、イベントを発信する製品のベンダ (vendor)、製品 (product)、プログラムまたは製品コンポーネント (program or product component) を表すようにします。 vendor コンポーネントがグローバルに一意になるようにするには、このコンポーネントに2～3文字の頭文字ではなく、会社のフルネームや他社が使用しないような略称を使用します。
- ユーザ・アプリケーションの開発者や管理者は、製品のベンダに関して推奨される規則に従って、最初のコンポーネントを部門を識別する名前にします。ただし、接頭語 local を選ぶこともできます。この接頭語は、ユーザがインストールする外部で開発された製品から発信されるイベントと衝突しないためです。

2番目のコンポーネントは、そのイベントを発信するアプリケーションまたはグループを識別する名前にします。たとえば、イベント名の先頭を myco.payroll や myco.admin にすることができます。
- Tru64 UNIX が発信するイベントでは、最初の2つのコンポーネントが sys.unix であり、3番目のコンポーネントでイベントを発信するサブシステムを識別します。最初のコンポーネントの sys は、システム・イベント用に予約されています。

次の表に、特殊な意味を持つ名前をリストします。 名前の衝突を防ぐために、これらの名前は指定以外のイベントの最初のコンポーネントには使用しないようにしてください。

名前	用途	例
sys	オペレーティング・システムのイベント	sys.unix.evm.logger.config_err
local	ローカルなユーザ組織用	local.app.payroll.started local.app.payroll.started
test	ローカルなテスト・イベントの発信用	

イベント名は、選択して公開した後は変更しないでください。 アプリケーションが発信するイベントは、外部インタフェースの一部であり、他のアプリケーションや製品コンポーネントがそのイベントの受信に依存している可能性があるためです。

14.5.1.1.1 予約コンポーネント名

前項では、イベント名が詳細になれば、管理者は EVM フィルタを用いて特定のタイプのイベントを探す操作がやりやすくなると説明しました。たとえば、`sys.unix.hw.registered.cpu` というイベントは、オペレーティング・システムのハードウェア管理サブシステムが、プロセッサを検出して登録したことを示します。このイベントでは、変数データ項目 (14.5.2 項) にプロセッサのハードウェア ID が含まれているので、イベント・ビューアまたは `evmshow` でイベントを表示させれば、ハードウェア ID が分かります。そのプロセッサに関連する問題をトレースしている場合は、イベント・フィルタを使用して、そのハードウェア ID を含むイベントのみを選択することができます。イベント自体に含まれる変数の値でイベントをフィルタリングすることはできないので、この選択は、選択に使用する情報がイベント名にも含まれている場合にのみ使用できます。この場合、ハードウェア管理サブシステムは、登録した各デバイスに割り当てたハードウェア ID によって名前を拡張することで、選択を可能にします。ハードウェア ID が 2 の場合、次のコマンドで関連するイベントを検索できます。

```
evmget -A -f "[name sys.unix.hw.*.2]"
```

この例では、アスタリスク (*) がワイルドカード文字になり、ID 2 を含むハードウェア・サブシステムが発信したイベントがすべて、報告内容にかかわらず見つけられるようになります。ただし、ハードウェア・サブシステム以外のサブシステムも、プロセッサに関する情報を通知する可能性があるため、このフィルタでは、関心のあるデバイスのイベントをすべて選択できないことがあります。このようなイベントを検索に含めたい場合もあります。`[name *.2]` と指定すると検索範囲を広げることができますが、ハードウェア ID に関係のない 2 という数値を含む他のイベントがログに入っていると、それも結果に含まれてしまいます。

予約コンポーネント名の規約により、発信したサブシステムやアプリケーションにかかわらず、関連イベントを選択できるようになります。規約では、予約コンポーネント名は下線文字 (_) で始まり、常に特定タイプのエンティティを識別します。発信サブシステムやアプリケーションは、イベントを発信する前に予約コンポーネント名を基本イベント名に付加し、識別するエンティティに応じて、それに続くコンポーネントが特定のインスタンスを識別します。

予約コンポーネント名は、一般に特定のサブシステムやアプリケーションの代わりとして定義されますが、名前が定義された後の使用は制限されません。

そのエンティティに関して報告する情報を持つものはすべて、発信するイベント内に、予約名とエンティティ ID を含めることができます。たとえば、予約コンポーネント名 `_hwid` は、ハードウェア管理サブシステムで定義されているとおり、ハードウェア・デバイス ID を指定します。その後にはデバイス ID が必要です。したがって、前述の例を続けると、プロセッサの登録を報告するイベントの名前は、`sys.unix.hw.registered.cpu._hwid.2` になります。次のコマンドで、すべてのハードウェア・デバイスに関連するすべてのイベントが検索できます。

```
evmget -A -f "[name *._hwid]"
```

次のコマンドを使用すると、最も関心のあるプロセッサまで検索の幅を狭めることができます。

```
evmget -A -f "[name *._hwid.2]"
```

発信者が規約に従っているという前提で、ハードウェア ID を除くすべてのコンポーネントに対するフィルタにワイルドカードを使用すると、発信したサブシステムやアプリケーションにかかわらず、関連するイベントがすべて確実に検索されます。

予約コンポーネント名の規約では、予約コンポーネント名を使用する際は、発信されるイベントに同じ名前と値の変数 (先頭の下線を含む) が含まれていなければなりません。この規約により、受信クライアントは、イベント名を解析して値を検索することなく、EVM の通常の API 関数を通じてイベントから値を取得できます。それ以外の場合、この規約では同じ情報をイベントに 2 回追加しなければならないので、`EvmEventPost()` 関数は名前が下線で始まる変数をイベント内で自動的に検索し、呼び出し側のプログラムが対応するコンポーネントをイベント名にまだ含めていない場合、イベントを発信する前にそれを付加します。名前の自動拡張については、`EvmEventPost(1)` を参照してください。

イベントには、任意の数の予約コンポーネント名を、イベント基本名の後の任意のコンポーネント位置に順不同で付加できます。各予約コンポーネントのすぐ後には、対応する値コンポーネントが必要です。

EVM の名前照合方式では、発信イベントと一致する部分の最も多い名前が検索されるため、イベントを登録するときに、イベントのテンプレート名に予約コンポーネントを指定する必要はありません。この節で説明した例では、テンプレート (およびイベントの基本名) は `sys.unix.hw.registered` にな

ります。予約コンポーネントを名前に追加すると、イベントの発信時に、より詳細なレベルを指定できます。

先頭に下線が 1 つある予約コンポーネント名を定義する規約は、システム用に予約されています。ローカル・サイトとサードパーティ製品のベンダは、`__prod_code` のように、予約コンポーネント名の先頭に下線を 2 つ使用して、独自の規約を規定しなければなりません。

Tru64 UNIX が使用する予約済みのコンポーネント名のリストは、`EvmEvent(5)` を参照してください。

14.5.1.1.2 イベント名の比較

イベント名を検索対象の名前と比較するには、EVM の API 関数を使用します。ただし、`strcmp()` は使用しないでください。着信イベントの名前には、予期しない数のコンポーネントが含まれることがあります。イベント名の照合についての詳細は、14.7.12.9 項を参照してください。

14.5.1.2 イベントのフォーマット・データ項目

フォーマット・データ項目には、分かりやすい文字列でイベントの概要が説明されています。フォーマット行は、次のような場合に展開されて表示されます。

- イベントが `evmshow` コマンドで表示されている場合
- イベントがイベント・ビューアで表示されている場合
- プログラムが `EvmEventFormat()` などの表示 API 関数を呼び出した場合

フォーマット・データ項目には、イベント内の他のデータ項目への参照が含まれることもあります。次の例について説明します。

```
Application close-down has started
```

```
Close-down of application $app has started
```

2 つ目の例では、`$app` はアプリケーションの名前に置き換わります。このとき、イベントにはこの名前の変数の値が含まれているとします。したがって、この行は次のように表示されます。

```
Close-down of application payroll has started
```

変数については、14.5.2 項で説明しています。

フォーマット行は、発信コンポーネントに関する簡潔な一貫性のある識別情報から開始して、発生した現象が簡単にわかるように記述します。次のガイドラインを考慮してください。

- 完全な文章である必要はないが、はっきりとわかりやすく記述する。
- 複数の文を記述しない。メッセージの最後にピリオドを付けない。発生した現象について詳細で文法的に正しい説明を記述する場合は、イベントの説明テキストを使用する。説明テキストの使用方法については、14.6.2 項を参照。
- 最も重要な情報 (発生した現象を管理者に伝える部分) を行の先頭に記述して、この情報を表示するときに、管理者がビューア・ウィンドウを水平方向にスクロールする必要がないようにする。変数データは、多くの場合、行の最後に記述した方がよい。
- フォーマット文字列には、すべての変数への参照を含める必要はない。管理者は、イベントの詳細な説明を要求することによって簡単に変数データを取得できる。ただし、ファイル・システムやデバイスの名前などの主要なデータは記述する。

代入を行う変数名を指定するには、`$app` のように、前に `$` を付けます。必要に応じて、`@host_name` のように、標準データ項目の前に `@` を付けることもできます。標準データ項目名では大文字および小文字は区別されませんが、変数名では大文字と小文字が区別されます。

フォーマット・テキストで `$` と `@` の特別な意味を無効にするには、前にバックスラッシュ (`\`) を付けます。テキストに定数のバックスラッシュを含めるには、バックスラッシュを 2 つ (`\\`) 使用します。隣接するテキストからデータ項目または変数を区別するには、`${app}` のように、名前を中カッコで囲みます。

変数やデータ項目の表示方法は、名前の後にパーセント記号 (%) とフォーマット指定子を続けることで制御できます。フォーマット指定子では、表示フィールドの最小幅と表示フォーマットを指定します。たとえば、次のフォーマット文字列は、`device_id` という変数が 16 進数形式で表示され、`temperature` が浮動小数点、小数点以下 2 桁で表示されることを示します。

```
The cabinet temperature for device 0x$device_id%x is $temperature%.2 degrees
```

これに対応するイベントは次のように表示されます。

The cabinet temperature for device 0x7a is 62.4 degrees

有効なフォーマット指定のリストは、evmshow(1) を参照してください。

表 14-2 は、フォーマットされたイベントを作成するときに、イベントのフォーマット・テキストに変数とデータ項目が代入される場合の例です。

表 14-2: イベント・テキストへの変数代入

フォーマット・データ項目	変数データ項目	生成されたテキスト
Application started	なし	Application started
Debug message: \$msg	msg (string) = "Check-point reached"	Debug message: Checkpoint reached
Temperature too high: \$temp	temp (float) = 87.2	Temperature too high: 87.2
This event was posted by @user_name	なし	This event was posted by jem

14.5.1.3 イベントの優先度データ項目

イベントの優先度は、イベントのログ記録，ソート，表示，および対応を手動または自動で行うときの、イベント選択基準として使用されます。この優先度は、受信者の EVM クライアントにイベントを配信する順序の決定には使用されません。イベントは、EVM デーモンが受信した順に配信されます。優先度は 0 ~ 700 の整数値で、0 が最も低い重要度になります。優先度レベルについての詳細は、EvmEvent(5) を参照してください。

システム管理者は、優先度に応じて次のような対応を行います。

- 優先度が 200 (通知レベル) 以上のすべてのイベントのログが記録されるように、ロガーを構成する。
- 優先度が 600 (警報レベル) 以上のイベントが発信されたときに、メールの送信または警報の発行 (ポケベルで呼び出す、など) が行われるように、ロガーを構成する。
- ビューアを使用して、優先度が 300 (警告レベル) 以上のイベントをすべて検索する。
- ビューアを使用して、ログに記録されたすべての優先度のイベントを調べ、問題を分析したり、システムが正しく動作していることを確認する。

優先度は、発信コード内にハードコードしないでください。優先度はテンプレート・ファイルに設定します。

イベントの優先度を選択する場合は、同じアプリケーション内、およびその他の類似したアプリケーション内のほかのイベントと一貫性を保つようにします。アプリケーションに適用できない優先度がある場合は、使用する優先度を制限してください。EvmEvent(5)の優先度情報をガイドラインとして使用し、既存のテンプレートを参照して、同様のイベントで使用されている優先度を確認します。

イベントの優先度は慎重かつ客観的に選択してください。発生する結果ではなく、通知する内容を考慮します。たとえば、アプリケーションの障害には重要なものもありますが、多くはそうではありません。アプリケーションの重要度は、上位のコンポーネントで判断します。上位のコンポーネントでは、エラー・イベントを受信して、重要度が認識されているアプリケーション・エラーが通知されたときに、クリティカル・レベルのイベントを発行します。

これらの考慮事項を基準にして、エラーは発生しているがクリティカルの基準に達していないイベントの優先度は、500 (クリティカル・レベル)ではなく400 (エラー・レベル)に指定します。一方、システムの温度が高すぎることを通知するイベントは、ほとんどの場合クリティカルです。

相互に関連するイベントの優先度は、イベントごとに設定します。たとえば、優先度が500の障害イベントを発信してアプリケーションに障害が発生したことを通知してから、関連するイベントを発信してアプリケーションがリストアされたことを通知する場合、2つ目のイベントの優先度は、たとえば200など、最初のイベントより低く設定します。リストア・イベントに優先度500を設定した場合は、イベントに対して不適切なアクションがとられることがあります。

14.5.1.4 I18N カタログ名、メッセージ・セットID、およびメッセージID データ項目

イベントに含まれているテキストが、さまざまな国の管理者に通知される可能性がある場合は、イベントの多国語対応を検討してください。イベントの多国語対応を行うには、すべてのイベントに対するフォーマット文字列が格納されたI18N カタログ・ファイルが必要です。ただし、イベント・テキストを表示するときにカタログ・ファイルが使用できない場合は、イベントにテキストを記述してください。

カタログ・ファイルは、通常の I18N の規則に従って配置し、イベントの仕様にそのファイル名を含める必要があります。言語ロケールごとに、独自のカタログ・ファイルを指定することもできます。詳細は、『国際化ソフトウェア・プログラミング・ガイド』を参照してください。

カタログ・ファイルを複数のメッセージ・セットに分割し、イベントにメッセージ・セット ID を指定することもできます。1 つのイベントに関連するメッセージは、すべて同じメッセージ・セットに属していなければなりません。

データ項目 `I18N_catalog` および `I18N_msgset_id` はカタログ名を定義します。また、該当する場合は、フォーマット・テキストのメッセージ・セットと、イベントに含まれる文字列変数のうち関連するメッセージ ID が指定されているものをすべて定義します。データ項目 `I18N_format_msg_id` には、フォーマット・テキストのメッセージ ID を定義します。カタログまたはメッセージ ID が指定されていない場合は、イベント概要を表示するときに、フォーマット・データ項目に指定されたフォーマット・テキストが使用されます。

イベント・テキストの翻訳の設定方法については、14.6.4 項を参照してください。

14.5.1.5 クラスタ・イベント・データ項目

クラスタ・イベント・データ項目が `true` に設定され、TruCluster システムのメンバであるシステムにイベントが発信されると、EVM はクラスタのアクティブなノードすべての受信者にイベントを配信します。項目が設定されていないか、`false` に設定されている場合、イベントはローカル・ノードの受信者にのみ配信されます。イベントがスタンドアロンのシステムに発信される場合、この項目に効果はありません。クラスタ・イベント・フラグを `true` に設定しているイベントは、クラスタワイド・イベントと呼ばれます。

テンプレートまたは `evmpost` へのイベント指定の中で使用すると、クラスタ・イベント・データ項目の値は `true` または `false` になります。API 関数のデータ項目として使用されると、値は `EvmTRUE` または `EvmFALSE` のいずれかになります。

クラスタワイド・イベントは、イベントがユーザ・レベルとカーネルのどちらから発信されたかによって、異なります。

- ユーザ・レベルでは、クラスタワイド・イベントは他のイベントと同じようにローカルの EVM デモンへ発信され、次にデモンはそのイベントを自分の受信者と、他のアクティブなノードで実行中の EVM デモンに配信します。event_id データ項目はローカル・デモンによって設定され、受信者に配信される前に他のデモンによって変更されることはありません。これは、次のようなことを意味します。
 - 全ノードの受信者はそのイベントの同一のコピーを受信する。
 - 発信ノードではないノード上で受信したコピーの event_id データ項目は、これらのノードでローカルに発信されるイベントとは順番が違っていることが多い。
- カーネルから発信されるクラスタワイド・イベントは、最初にローカル・デモンを通すことなく他のノードへ配信されるので、その結果、ローカルの event_id の値を与えられていません。この場合、各ノードで実行されるデモンは、event_id が設定されていないことを認識し、自分で値を設定します。これは、次のようなことを意味します。
 - クラスターの異なるノードの受信者は、同じカーネルが発信したクラスタワイド・イベントを異なる event_id の値で受信する。
 - 発信ノードではないノード上で受信したコピーの event_id データ項目は、これらのノードでローカルに発信されるイベントと連続している。

カーネルレベル・イベントの発信と受信についての詳細は、kevm(7) を参照してください。

EVM は、クラスタ別名を使用してクラスタに接続されている受信者が、クラスタワイド・イベントのコピーを 1 つだけ受信するようにします。ただし EVM logger は、どのクラスタ・メンバが発信したかに関係なく、フィルタに一致するイベントをすべて受信して記録するため、各ノードでコピーが別々に記録されます。その結果、格納されたイベントを、クラスタ別名を使用した接続により evmget コマンドを使用して取得する場合、イベントを記録した各ノードからイベントのコピーが 1 つずつ返されることになります。

クラスタベース・アプリケーションを開発する場合、どのイベント(存在する場合)をクラスタワイド・イベントに指定する必要があるかを、注意して判断してください。目的の受信者が、イベントの各インスタンスをどのノードがポストしたかを認識して適切な動作を行うように設計することが特に重要です。member_id データ項目を標準ライブラリ関数(clu_get_info()) など

と組み合わせて使用すると、発信とローカル・ノードについての詳細が得られます。詳細は、`EvmEvent(5)` および `clu_get_info(3)` を参照してください。

14.5.1.6 参照データ項目

参照データ項目は、イベントの詳細表示のために、イベントの説明テキストを検索するときに使用されます。このデータ項目の値は、イベント名とともにイベント・チャンネルの説明スクリプトに渡されて、イベントに関連付けられている説明テキストが識別できるようにします。説明スクリプトはチャンネルごとに異なるため、フィールドのフォーマットはチャンネルごとに異なります。ただし、EVM ログ (`evmlog` チャンネル) に対して格納/取得されるイベントの場合は、参照データ項目には次のフォーマットの文字列を指定します。

```
cat:catalog_name[:set_number]
```

`catalog_name` は、イベントの説明テキストが格納されている I18N カタログの名前です。説明スクリプトで適切なメッセージを検索できるようにするには、カタログ内の各説明メッセージの先頭に、中カッコで囲まれたイベントの名前がなければなりません。

オプションの `set_number` は、説明テキストが格納されたカタログのメッセージ・セットの番号です。セット番号を指定しない場合は、カタログ全体が検索されます。

サードパーティ製品のベンダおよびローカル・アプリケーションの場合は、説明テキストを Tru64 UNIX の説明テキスト・カタログ `evmexp.cat` には追加しないで、別のカテゴリを提供します。この項目の値は、通常、テンプレート・ファイルにグローバル・データ項目として設定します。

イベントの説明テキストの作成方法についての詳細は、14.6.2 項を参照してください。

14.5.2 変数データ項目

イベントの変数データ項目は、イベントのインスタンスごとに異なる情報を指定するときに使用します。たとえば、高温が検出されたときにイベントを発信する場合は、実際の温度を浮動小数点値として変数データ項目に指定することができます。

変数データ項目には、次のプロパティがあります。

- 名前

- 型
- 値
- サイズ (ほとんどの型で暗黙に指定される)
- I18N メッセージ ID (オプション — 文字列変数の場合にのみ適用)

変数名は、大文字または小文字の英数字および下線 (_) を任意に組み合わせて指定できます。名前はわかりやすく指定する必要があります。ただし、変数名はイベントに含めて送信されるため、名前が長くなるほどイベントの物理的な長さも大きくなります。

変数データ項目は、受信者が直接抽出および使用したり、表示用のフォーマット済みイベントを作成するために、イベント・フォーマットのテキスト文字列と結合することができます。

表 14-3 に、EVM でサポートされる変数の型を示します。

表 14-3: EVM の変数データの型

型識別子	サイズと型
EvmTYPE_BOOLEAN	8 ビットの整数
EvmTYPE_CHAR	8 ビットの文字
EvmTYPE_INT16	16 ビットの符号付き整数
EvmTYPE_INT32	32 ビットの符号付き整数
EvmTYPE_INT64	64 ビットの符号付き整数
EvmTYPE_UINT8	8 ビットの符号なし整数
EvmTYPE_UINT16	16 ビットの符号なし整数
EvmTYPE_UINT32	32 ビットの符号なし整数
EvmTYPE_UINT64	64 ビットの符号なし整数
EvmTYPE_FLOAT	32 ビットの浮動小数点値
EvmTYPE_DOUBLE	64 ビットの浮動小数点値
EvmTYPE_STRING	ヌルで終了する文字列
EvmTYPE_OPAQUE	サイズを明示的に指定しなければならないバイナリ・データ

変数にはイベントのインスタンス固有の情報が含まれるため、通常は発信者が指定します。ただし、文書化の目的では、変数の名前と型、およびダ

ミーの値をイベントのテンプレートに指定しておくくと便利です。テンプレートについては、14.6.3 項を参照してください。

14.6 イベント・セットの設計

アプリケーションまたはサブシステムを設計する場合は、関連するイベント・セットも設計する必要があります。

EVM イベントは、慎重に設計する必要があります。イベントは、ヒューマン・スタイル (読みやすいテキスト) とプログラム・スタイル (バイナリ・データ) の 2 種類のインタフェース要件を満たす必要があります。イベントが発信された後は、テキスト形式またはバイナリ・データ形式のいずれかで表示および処理することができます。

イベントを設計するときは、次の事項を考慮します。

1. 関連するイベント・セットに対して、ファミリ名を決定します (詳細は、14.5.1.1 項を参照)。
2. 監視しているエンティティが関心のある状態変更の一覧を作成し、各イベントの名前を選択します (詳細は、14.6.1 項を参照)。
3. 各イベントの内容を決定します。すべてのイベントには固有の名前が必要です。ほとんどの場合、フォーマット文字列と優先度を指定する必要があります。また、多くの場合、変数も必要です。各変数について、型および指定できる値を決定します (詳細は、14.5 節を参照)。
4. 文書化のために、各イベントの詳細な説明を記述します。イベントの意味、発生するタイミング、ユーザまたは責任のある受信者の対応方法、およびイベントの内容 (特に、すべての変数データ項目) の詳細を記述します。説明テキストは、通常、カタログ・ファイル内に保存され、アクセスして表示することができます (詳細は、14.6.2 項を参照)。
5. イベントごとに、テンプレートに指定する項目、および発信者が指定する項目を決定します。イベント名を除いて、発信コードおよびテンプレートの項目はすべてオプションです。オプション項目が発信コードおよびテンプレートの両方で宣言されている場合は、発信者の宣言が優先されます (テンプレートについての詳細は 14.6.3 項を、一般に発信されるイベント項目についての詳細は 14.5 節を、テンプレートと発信イベントのデータ項目のマージ方法についての詳細は 14.6.3.3 項をそれぞれ参照)。

6. イベントを多国語対応にするかどうかを決定します。多国語対応にする場合は、I18N カタログ・ファイルに対して名前を選択し、必要なメッセージ・セットをカタログに設定します (詳細は、14.6.4 項を参照)。

設計者は、EVM イベントは、ほかのプログラムまたはサブシステムが依存するインタフェースであることを認識する必要があります。このため、設定後は、通常変更しないようにします。

14.6.1 イベントに値する状態変更の決定

イベントの重要性は、アプリケーションごとに異なります。場合によっては、コードで認識された状態変更が、ほかのコードに通知が必要な重要性を持つかどうかを決定するのが困難なこともあります。

次の現象が発生した場合は、イベントを発信することをお勧めします。

- コンポーネントによってシステムが変更された場合。たとえば、システム管理プログラムが、システムに新しいユーザを追加した場合や、ネットワーク構成を変更した場合。
- 潜在的に重大なエラーが発生した場合。たとえば、システム管理プログラムによって主要なシステム・ファイルが存在しないことが検出された場合や、デバイス・ドライバによってディスク障害が検出された場合。
- 障害が発生する可能性を示す境界値を超えた場合。たとえば、ファイル・システム管理ソフトウェアでは、ファイル・システムが境界値を超過し、95% 以上使用されていることが検出されたときに、警告イベントが発信される。ただし、状態が境界値の前後を上下する場合は、このようなイベントが繰り返し発信されないように注意する。通常は、あらかじめ設定した時間が経過しても同じ状態が続いている場合にだけ、イベントが再発信されるようにする。この場合、経過期間中に、何度か境界値を下回り回復したとしても影響しない。
- ユーザに特権が与えられた場合、またはシステムの操作に影響するアクションが実行された場合。たとえば、システム管理プログラムによってディスクのマウント/アンマウントが行われた場合、またはシステムがクローズされている場合。

次の現象が発生した場合は、イベントを発信しないでください。

- ユーザが制御しているセッションで、あるユーザによってエラーが発生したときに、そのユーザと直接通信できる場合。たとえば、ユーザ

がプロンプトに間違って入力した場合は、構成プログラムからイベントを発信しない。

- 次の基準を満たす「エラー」を処理している場合。つまり、予期されている通常の動作で、発生の原因がわかっているため、システム管理者の対応が必要ないエラーの場合。
- 検出された状態が既に通知されているため、再度通知する必要がない場合。

イベントを発信するときは、特定の状態が繰り返し発生する場合などに、短期間に同じイベントを繰り返し発信しないようにします。場合によっては、指定の期間内に同じ現象が発生した回数を通知するなど、要約イベントを一定間隔で発信することも有効な方法です。

イベント処理が頻繁に発生すると、アプリケーションがメッセージの負荷に対応できなくなる可能性があり、イベントが失われる原因になります。失われたイベントの処理方法については、14.7.12.10 項を参照してください。

14.6.2 イベントの説明テキストの作成

説明テキストは、すべての EVM イベントに必要です。説明テキストは、発信時にはイベントに含まれていませんが、カタログ・ファイルに保存されており、イベントの参照データ項目と名前データ項目の内容によって参照されます (14.5.1.6 項を参照)。sys.unix イベントの説明テキストは、evmexp.cat というカタログ・ファイルに物理的に格納されています。イベントの説明テキストを表示するには、evmshow の -x または -d オプションを使用するか、または SysMan イベント・ビューアを使用してイベントの詳細な表示を要求します。

説明には、イベントの名前および意味を記述します。コンテキスト (一定時間内の発生回数やほかのイベントの存在など) によって意味が異なってくるイベントの場合は、この内容および例をいくつか記述します。イベントに対してアクションが必要な場合は、そのアクションを記述します。コンテキストによってアクションが異なってくる場合は、その内容と例を記述します。イベントに対するユーザのアクションが必要ない場合は、その点を明示的に記述することをお勧めします。

例 14-1 は、システム・イベントの説明テキストの例です。

例 14-1: イベントの説明テキストの例

Example 1:

```
EVENT sys.unix.evm.daemon.event_activity
```

Explanation:

This high-priority event is posted by the EVM daemon when it detects a high number of events occurring over several minutes.

Action: Use the EVM event viewer or the `evmget(1)` command to review the event log for the source of the activity. If the log does not show high activity around the time at which this event was posted, it is likely that the events were low priority, and hence were not logged. You can monitor low-priority events by running the `evmwatch(1)` command with an appropriate filter, or by temporarily reconfiguring the EVM logger to log low-priority events.

Note: You can change the parameters that control the posting of this event by modifying the daemon configuration file, `/etc/evmdaemon.conf`.

14.6.3 イベント・テンプレートの設計

各発信イベントには、テンプレートが必要です。1つのテンプレートに複数のイベントを割り当てることもできます。テンプレートには、イベント名およびイベントの定数のデータ項目を定義します。

イベント・テンプレートは、次の2つの目的で使用されます。

- EVM へのイベントの登録。登録されていないイベントは発信できません。
- イベントのほとんど、またはすべてのインスタンスに対して共通のデータ項目 (メッセージ・カタログ情報など) をイベント・テンプレートに設定することができます。テンプレートに定数のデータ項目を設定しておく、これらの情報の更新が容易になり、イベント発信者が発信要求にこれらの情報を繰り返し指定する必要がなくなります。イベント・テンプレートに設定する項目の決定方法については、14.6.3.1 項を参照してください。

イベント・テンプレートの内容と発信イベントの内容をマージする方法についての説明は、14.6.3.3 項を参照してください。

テンプレートは、集中管理されたデータベース内のファイルに保存されます (14.6.3.4 項を参照)。テンプレート・ファイルには、任意の数のイベント・テンプレートを含めることができます。

14.6.3.1 イベント・テンプレートに設定する項目の決定

イベント発信者が設定する項目およびテンプレートで設定する項目は、設計時に決定します。定数データ項目は、通常、発信プログラムでイベントにハードコードするのではなく、イベント・テンプレートに設定します。

イベント・テンプレート方式の主な利点は、アプリケーションのすべてのイベントの定数属性を集中管理できるため、開発サイクル時に容易に変更ができることと、アプリケーションの開発後に属性情報の検索先が一箇所になることです。

原則として、発信アプリケーション・プログラムにハードコードするイベント情報は最小限にし、できるだけ多くの情報をイベント・テンプレートに設定します。通常、アプリケーションでは次の情報だけを設定します。

- イベント名 (必須 — 3 つ以上のコンポーネントで構成され、対応するテンプレートのイベント名と同じ長さまたはそれ以上でなくてはならない)
- 変数の値

通常、テンプレートには次の情報を設定します。

- イベント名 (必須 — 2 つ以上のコンポーネントが必要)
- 優先度
- フォーマット・テキスト
- I18N メッセージ・カタログ情報 (多国語対応のイベントの場合)
- Cluster event フラグ (該当する場合)
- 変数 (通常はゼロまたは空の文字列に初期化する)

変数データ項目の値は、通常、イベント発信時に発信アプリケーションで設定しますが、テンプレートにも変数を設定しておくと、文書化の点から、テンプレートの重要性が高まります。テンプレートの変数には、通常、ゼロ (文字列変数の場合は空文字列) を設定します。テンプレートに、実際の省略

時の値 (発信者が変更できる) を設定しておくとも便利です。この場合は、テンプレート・ファイルにコメントとして記述します。

次のソース・ファイルは、1つのイベントを含むイベントのテンプレート・ファイルの例です。

```
# Example event file
priority 200                # Default priority
ref cat:myapp_exp.cat      # Global reference
event {
    name                    myco.myapp.env.temperature
    format                  "Temperature is $temperature"
    var { name temperature type FLOAT value 0,0 }
}
```

イベントには、必要に応じて任意の数の変数を指定できます。ただし、不透明な変数 (バイナリ構造) は、テンプレートではサポートされません。

14.6.3.2 イベント・テンプレート名と発信イベントの名前の照合

EVM は、イベントが発信されると、テンプレート・データベースで、発信イベントの名前とイベント名が一致するテンプレートを検索します。一致する名前が存在しない場合は、イベントを発信したプログラムにエラー・コードを返します。一致する名前が見つかった場合は、EVM はテンプレートに設定されているデータ項目を、イベントを発信したプログラムで設定された項目と結合します。この操作によりマージされたイベントが生成されて、受信者に配信されます。マージ操作についての詳細は、14.6.3.3 項を参照してください。

テンプレート照合プロセスでは、発信イベント名の左端のコンポーネントと、テンプレートのイベント名のすべてのコンポーネントが一致していることのみが検査されます。EVM では、次の規則に基づいて、データベース内で最も近い一致を検索します。

- 最も近い一致とは、テンプレート・イベントの名前と発信イベントのコンポーネントを左端から右方向に比較したときに、発信イベントのほとんどのコンポーネントと正確に一致する名前を持つテンプレート・イベントのことである。
- 発信イベントのコンポーネント数が、最も近いデータベース・エントリのコンポーネント数以上の場合は、一致していると見なされる。発信イベントのコンポーネント数が少ない場合は、一致しているとは見なされない。

- 各コンポーネントは、正確に一致しなければならない。
- テンプレート名には、2 つ以上のコンポーネントが必要である。発信イベント名には、3 つ以上のコンポーネントが必要である。

テンプレートは、アプリケーションが発信するイベントごとに用意してください。こうしておくで、イベント固有の情報を、テンプレートに格納することによって集中管理することができます。しかし、最適一致方式の利点は、発信するときにイベント名にさまざまなインスタンス情報を追加して拡張できることです。たとえば、デバイス名や温度の値を追加コンポーネントとして追加できます。インスタンス・コンポーネントを追加した場合は、イベントのフィルタおよびソートが簡単になります。イベント名の拡張方法の例については、14.5.1.1.1 項を参照してください。

表 14-4 は、イベント・テンプレートと発信イベント間のイベント名の照合の例です。

表 14-4: 名前照合の例

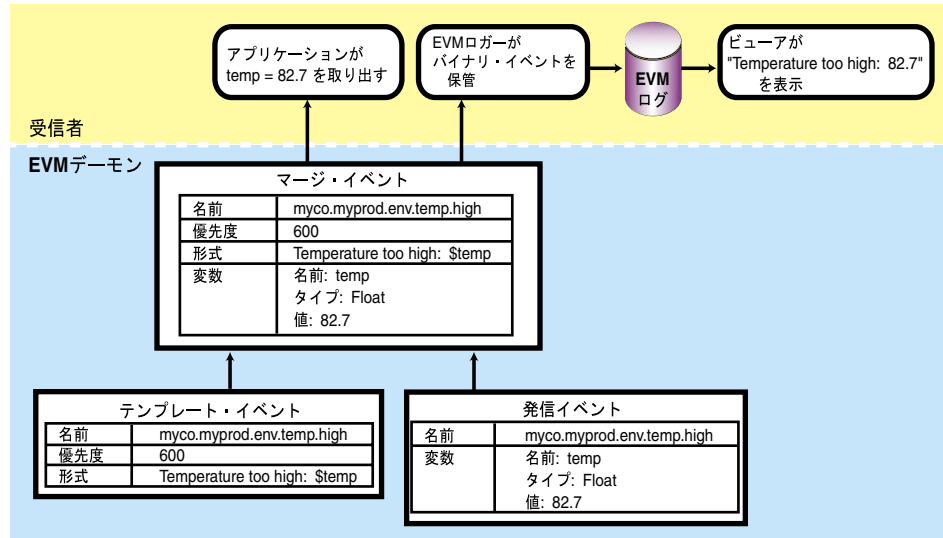
発信イベント名	テンプレートのイベント名	一致の状態
myco.myprod.env	myco.myprod.env	一致
myco.myprod.env.temp.high.70	myco.myprod.env.temp	一致
myco.myotherprod	myco.myother-prod.start	不一致。発信イベントのコンポーネント数が足りない。

14.6.3.3 テンプレートと発信イベントのデータ項目のマージ

EVM デーモンによって発信イベントの有効性が確認されると、発信イベントのデータ項目とテンプレートのデータ項目がマージされ、イベントを受信しているクライアントにマージされたイベントが配信されます。マージ・プロセスでは、テンプレートで設定するテキストおよびデータ項目と、発信者で設定するテキストとデータ項目の選択は、イベント設計者が大部分を決定することができます。

図 14-2 は、イベントのマージの概念図です。

図 14-2: 発信イベントとテンプレートのマージ



ZK-1399U-AJJ

テンプレートおよび発信イベントの両方に同じデータ項目が設定されている場合は、マージされたイベントでは発信イベントの値が使用されます。

マージ・プロセスでは、発信イベントおよびテンプレートのデータ項目が結合された、バイナリ形式のイベント構造が生成されます。マージされたイベントは、フォーマットされたメッセージとしてではなく、バイナリ形式で受信者に配信されるため、受信者は、EVM API 関数を使用して、イベントを表示できるようにフォーマットしたり、イベントから情報を抽出する必要があります。API 関数については、14.7 節を参照してください。

14.6.3.4 テンプレート・ファイルのインストール 位置，命名，所有権，および許可の要件

通常、イベントのテンプレート・ファイルは、製品またはアプリケーションをインストールするときにインストールされます。システム・テンプレートは、`/usr/share/evm/templates` の下のサブディレクトリに格納されます。サードパーティ製品およびローカル・アプリケーションのテンプレートは、ローカル・テンプレート・ディレクトリ `/var/evm/adm/templates` に格納されます。EVM デーモンが別のテンプレートの位置を指定できるように、システム・テンプレート・ディレクトリおよびローカル・ディレクトリ間には、リンクが作成されます。

製品またはローカル・アプリケーションにテンプレートを追加するには、アプリケーションをインストールするときに、ローカル・テンプレート・ディレクトリの下に適切な名前のサブディレクトリを作成し、そのディレクトリにテンプレートをインストールする必要があります。EVM のテンプレート検索ポリシーでは、シンボリック・リンクを介して検索されるため、アプリケーションと密接に関連するディレクトリにもテンプレートをインストールし、そのディレクトリとローカル・ディレクトリの間リンクを作成して接続することもできます。

テンプレート・ファイルの接尾語は、`.evt` でなければなりません。また、所有者は `root` または `bin` にし、`0600`、`0400`、`0640`、または `0440` のいずれかのアクセス許可が必要です。新しいディレクトリにも、適切な許可を割り当てます。

ファイルをインストールしたら、`root` として `evmreload -d` コマンドを実行し、EVM で新しいテンプレートを認識できるようにします。次に、エラーがないかどうかを確認します。詳細は、`evmreload(8)` を参照してください。

14.6.3.5 イベント・テンプレートの登録の確認

テンプレートが登録されたかどうかを判断するには、`-i` オプションを指定して `evmwatch` を使用します。たとえば、次のコマンドを実行すると、`myapp` アプリケーションに登録されているすべてのイベント・テンプレートの名前の一覧が表示されます。

```
evmwatch -i -f "[name myco.myprod.myapp]" | evmshow -t "@name"
```

テンプレートの詳細を表示するには、`evmshow` の `-d` オプションを使用します。`evmwatch` では、アクセス権限のないイベントのテンプレートは返されません。このため、テンプレートを表示する場合は、`root` としてログインしなければならないことがあります。

14.6.4 イベント・テキストの翻訳の設定 (I18N)

イベントのフォーマット・データ項目およびイベントに含まれる文字列変数の値を表示するときに、別の言語に変換できるようにするには、イベントを多国語対応 (I18N) にする必要があります。さまざまな国で使用される製品を開発している場合は、任意のまたはすべての項目を変換できるようにします。

格納されている同一のイベントを表示する場合、ユーザごとに異なる言語が使用される可能性があります。このため、言語の翻訳は、イベントの発

信時ではなく、イベントが表示用のフォーマットに変換されるときに同時に行う必要があります。使用されるすべての言語のテキストをイベントに設定することは実際的ではないため、イベントがフォーマットされるときに、関連するメッセージ・カタログが使用できるようにしなければなりません。製品の開発者は、メッセージ・カタログを提供し、製品と一緒にインストールされるようにメッセージ・カタログを組み込む必要があります。カタログが使用できない場合のために、多国語対応のイベントに対して省略時の母国語文字列を含めることができます。

次のデータ項目は、イベントの多国語対応化をサポートしています。

- I18N メッセージ・カタログ名
- I18N メッセージ・セット識別子 (オプション)
- フォーマット・データ項目の I18N メッセージ識別子
- 多国語対応の文字列変数ごとの I18N メッセージ識別子
- この一覧にある任意のまたはすべての項目に対する省略時の母国語文字列

イベントのメッセージ識別子は、すべて同じメッセージ・カタログに関連付けられているため、すべて同じメッセージ・セット (省略時は 1) に属していなければなりません。

イベントのフォーマット文字列のカタログ ID、セット ID、およびメッセージ ID は、通常は変更されないため、すべてイベント・テンプレートに設定してください。イベントに文字列型の変数が含まれる場合、その変数はデバイス名やアプリケーション名などの項目を参照していることが多いため、表示する言語にかかわらず、通常は翻訳する必要はありません。このため、ほとんどの場合、メッセージ ID を設定する必要はありません。ただし、文字列変数の値を翻訳しなければならない稀な場合は、発信者はメッセージ ID を設定する必要があります。

表 14-5 は、イベント例に対する多国語対応の値の例です。

表 14-5: 多国語対応のイベントに対するデータ項目値の例

イベントのデータ項目	値	メッセージ ID
名前	acme.prod.env.temp	n/a
メッセージ・カタログ	acme_prod.cat	n/a

表 14-5: 多国語対応のイベントに対するデータ項目値の例 (続き)

イベントのデータ項目	値	メッセージ ID
フォーマット文字列	Temperature of sensor \$sensor is \$temperature	541
文字列変数 “ sensor ”	S27	n/a
文字列変数 “ temperature ”	high	542

temperature.cat の英語版とフランス語版は、次のようになります。

- 英語版
 - 541:Temperature of sensor \$sensor is \$temperature
 - 542:high
 - 543:low
- フランス語版
 - 541:La temperature du senseur \$sensor est \$temperature
 - 542:haute
 - 543:basse

ビューアでイベントを表示する場合は、ビューアによってフォーマット関数が呼び出され、ユーザのロケール設定に応じて、フォーマット関数から次のいずれかの文字列が返されます。ただし、適切なカタログ・ファイルを見つけることができ、指定されたメッセージがファイルに格納されている必要があります。

```
"Temperature at sensor S27 is high"
"La temperature du senseur S27 est haute"
```

フォーマット関数でカタログからイベントが解釈できない場合は、イベントに設定されている値が使用され、ユーザのロケールにかかわらず、次のメッセージが返されます。

```
"Temperature at sensor S27 is high"
```

イベント・ファイルが別のシステムに渡されて分析される場合は、関連付けられているカタログ・ファイルが使用できないことがあります。上の例のように、イベントに省略時の値が含まれている場合は、母国語で表示することができます。イベントに省略時の値が含まれていない場合は、フォーマット

された文字列を使用して、イベント名およびすべての変数のダンプを表示することができます。たとえば、次のように表示されます。

```
Event "myco.myprod.env.temp": $sensor = "S27" $temperature = "high"
```

I18N についての詳細は、『国際化ソフトウェア・プログラミング・ガイド』を参照してください。

14.7 EVM プログラミング・インタフェース

EVM イベントは不透明なデータ構造であり、EVM のアプリケーション・プログラミング・インタフェース (API) 関数によるアクセスと操作が可能です。

Tru64 UNIX システム上で実行する Java プログラムをサポートするために、JNI (Java Native Interface) パッケージがあります。Java インタフェースの詳細は、Web ブラウザを使用して次のファイルを参照してください。

```
file:/usr/share/doc/lib/evm/java_api/help-doc.html
```

以降の各項では、EVM イベントに対して通常実行されるいくつかの操作に関するプログラミング情報について説明するとともに例を示します。

14.7.1 EVM ヘッダ・ファイル

EVM 関数を使用するプログラムには、次のヘッダ・ファイルをインクルードします。

```
#include <evm/evm.h>
```

14.7.2 EVM API ライブラリ

EVM API 関数を使用するプログラムでは、シェアード・ライブラリ `libevm.so` またはスタティック・ライブラリ `libevm.a` に対してリンクを作成する必要があります。シェアード・ライブラリはルート・パーティション・ディレクトリ `/shlib` に格納されています。このため、システムがシングルユーザ・モードで稼働しているときに、動作している必要があるプログラムから使用できます。ただし、EVM デーモンは、実行レベルが 2 になったときに起動されるため、シングルユーザ・モードのときにデーモンに接続しようとするとうまくいきません。 `/usr/shlib` から `/shlib` 内のシェアード・ライブラリに対するシンボリック・リンクが作成されているため、省略時のライブラリ検索パスを使用して、リンク先のプログラムがライブラリにアクセスすることができます。

14.7.3 戻り状態コード

EVM 関数から返された状態コードは、ヘッダ・ファイル `evm/evm.h` に列挙されます。EVM 関数から共通に返される値は、次のとおりです。

- `EvmERROR_NONE` — 操作が正常に完了した。
- `EvmERROR_INVALID_ARGUMENT` — 関数に渡された引数の 1 つが無効だった。
- `EvmERROR_INVALID_VALUE` — 構造内に無効な値があった。
- `EvmERROR_NO_MEMORY` — ヒープ・メモリを取得できなかったため、操作が失敗した。
- `EvmERROR_NOT_PRESENT` — 要求された項目がイベントに存在しない。

状態コードの詳細なリストは、`evm/evm.h` ファイルを参照してください。

14.7.4 シグナルの処理

EVM API は、通常の処理ではシグナルは使用せず、アプリケーション・プログラムでシグナルが使用されるときにも通常は干渉しません。ただし、Tru64 UNIX システムの省略時のアクションでは、データを読み込むプロセスが存在しないときに、ローカル接続に対して書き込もうとした場合、サイレントに終了します。このため、EVM デーモンが接続アクティビティの前または途中で終了した場合は、クライアント・プロセスが何も記録しないで終了する恐れがあります。

この状態が発生するのを回避するために、`EvmConnCreate()` 関数は、SIGPIPE のハンドラが呼び出し側によって設定されているかどうかをチェックし、設定されていない場合は省略時のハンドラをインストールします。シグナルが発生してもハンドラは何のアクションもとりませんが、ハンドラが存在するためにクライアントは終了しません。EVM ハンドラは、`EvmConnCreate()` 呼び出しの前または後にプログラムで独自のハンドラを設定することにより、無効にできます。プログラムで省略時のアクションが必要な場合は、`EvmConnCreate()` を呼び出してから `SIG_DFL` にアクションを設定します。

詳細は、`signal(2)` を参照してください。

14.7.5 マルチスレッド・プログラムでの EVM

EVM API 関数は、すべてスレッド・セーフです。このため、内部静的記憶域を使用しなければならない場合は、ロックを使用して、記憶域に対して各スレッドから同時にアクセスされないようにします。それでも、マルチスレッド・プログラムで EVM API 呼び出しを使用する場合は、同期エラーを回避するために一定の予防策が必要です。

1. 可能であれば、API 関数から返されたエンティティの使用を、エンティティが設定されているスレッド内に制限します。次のエンティティで制限することができます。

エンティティ	型	エンティティを返す関数
接続コンテキスト	EvmConnection_t	EvmConnCreate()
イベント	EvmEvent_t	EvmEventCreate() EvmEventCreateVa() EvmEventRead() EvmEventDup()
データ項目	EvmItemValue_t	EvmItemGet()
データ項目リスト	EvmItemList_t	EvmItemListGet()
変数	EvmVarValue_t	EvmVarGet()
変数リスト	EvmVarList_t	EvmVarListGet()
イベント・フィルタ	EvmFilter_t	EvmFilterCreate()
接続 fd	EvmFd_t	EvmConnFdGet()

2. これらのエンティティを複数のスレッドから参照する場合は、ロックを使用して、同時アクセスまたは同時更新が行われないようにする必要があります。

これらの規則に従わない場合は、予期しないエラーが発生する可能性が高くなります。

14.7.6 EVM イベントの再割り当てと複製

EVM イベントの作成、受信、または読み取りが行われた後で、そのイベントを再度割り当てる必要がある場合は、イベントがメモリに保存される方法を理解しておくと便利です。

`EvmEvent_t` 型には、制御情報を保持する短いハンドル構造体へのポインタと、イベント本体へのポインタが定義されています。 `EvmEventCreate()` または関連する関数を使用して新しいイベントを作成すると、その関数によってハンドルおよびイベント本体に対してヒープ・メモリがそれぞれ割り当てられます。次に、本体のアドレスがハンドルに格納され、ハンドルのアドレスが関数の `EvmEvent_t` 出力引数として返されます。イベントを参照する場合は、返されたポインタを使用しなければなりません。

変数を追加するなど、イベントを変更すると、多くの場合、イベント本体に対して使用されていた領域が解放されて、別の位置に再度割り当てられます。このとき、ハンドルに格納されたアドレスが自動的に更新され、新しい位置が反映されます。つまり、再割り当ては、プログラムに対して完全に透過的に行われます。ハンドルの位置は、イベントの存在期間内は変更されません。

イベントを変数間で転送する場合は、簡単な C 言語の代入文を `EvmEvent_t` 型の変数間で使用します。保持されている値は、一定の位置 (イベントのハンドル) へのポインタだけなので、必要に応じて両方の変数からイベントを参照できます。ただし、後で `EvmEventDestroy()` を使用してイベントを破壊する場合は、両方の参照を破棄しなければなりません。

イベントの再割り当てでは、イベント本体はコピーされず、イベントのハンドルへの参照だけがコピーされることに注意してください。完全に独立したイベントのコピーを作成する必要がある場合には、`EvmEventDup()` 関数呼び出しを使用します。

14.7.7 コールバック関数

EVM の発信および受信クライアントは、`EvmConnCreate()` 関数呼び出しを使用して EVM デーモンに接続します。このとき、`EvmRESPONSE_IGNORE`、`EvmRESPONSE_WAIT`、`EvmRESPONSE_CALLBACK` の 3 つの応答モードから 1 つを指定しなければなりません。これらのモードについては、`EvmConnCreate(3)` を参照してください。

受信クライアントの場合は、応答モードとして `EvmRESPONSE_CALLBACK` を指定する必要があります。着信イベントは、`EvmConnCreate()` の 4 番目の引数に指定したコールバック関数によって受信クライアントに渡されます。受信クライアントからコールバック関数を使用する場合の例については、14.7.12.6 項を参照してください。

コールバック関数を使用する場合は、シグナル・ハンドラと異なり、非同期的に呼び出されないことに注意してください。プログラムで `EvmConnWait()`、`select()`、または関連する関数を使用して入力アクティビティの接続を確認してから、`EvmConnDispatch()` を呼び出してアクティビティを処理する必要があります。 `EvmConnDispatch()` によって、接続からの着信メッセージが読み込まれ、必要に応じてコールバック関数が呼び出されます。コールバック関数が呼び出されるタイミングについては、`EvmCallback(5)` を参照してください。

ほかの関数と同様に、コールバック関数に渡される引数は、プログラム・スタックに渡され、関数の有効範囲内でのみ使用できます。このため、コールバックから戻った後で使用するために値を保存しておきたい場合は、戻る前に値をグローバル・メモリ空間にコピーする必要があります。

コールバックによって着信イベントの存在が通知されたときに、イベントをコールバック内で処理および破壊するのではなく、保存したい場合は、グローバル・アクセスできる `EvmEvent_t` 型の変数を宣言して、着信イベントを割り当てます。次に例を示します。

```
/* In global declarations */
EvmEvent_t SavedEvent;
...

/* In your callback function */
SavedEvent = cbdata->event;
```

イベントを割り当てたときは、イベント本体ではなくイベントへの参照だけがコピーされるため、コールバック関数でイベントを破壊してはなりません。イベントの処理が終了したら、後でイベントを破壊する必要があります。破壊しない場合、メモリ・リークが発生することがあります。

イベントの割り当てについては、14.7.6 項を参照してください。

14.7.8 接続ポリシーの選択

プログラムが発信クライアントの場合、EVM デーモンに接続するポリシーを選択する必要があります。次のいずれかを選択してください。

- プログラムの起動時に接続を確立し、プログラムが終了するまで維持する (永続的な接続)。
- イベントの発信が必要になったときに毎回接続を確立し、終わるとすぐに切断する (一時的な接続)。

処理中に EVM 接続を確立する場合の CPU 負荷は無視できません。セットアップと認証のプロトコルで、ファイル入出力の他にメッセージのトランザクションが複数必要になるからです。原則的には、通常のプログラム動作中に何度もイベントの発信があると予測される場合には、永続的な接続を行ってください。このオプションを選択した場合、プログラム・コードでは、EVM デーモンが終了した場合の予期しない切断を考慮しなければなりません。切断の処理については、14.7.9 項を参照してください。

イベントをあまり発信しない場合(たとえば、予期しないエラーが起きたときに限られる場合)は、一時的な接続を行ってください。一時的な接続では、複数のイベントが発信されるとセットアップ時間の負荷が大きくなりますが、永続的な接続に必要なシステム資源の消費と、予期しない切断に対するプログラム・コードでの対処が不要になります。一時的な接続でローカル・デーモンにイベントを発信するための最も簡単な方法は、`EvmEventPost()` または `EvmEventPostVa()` の接続引数として `NULL` を渡すことです。

永続的な接続は必要としないが、特定の状況で短期間に何度もイベントを発信することが予測される場合は、個別に接続してイベントを連続的に発信するよりも、一時的な接続を作成して、動作が完了した時点でそれを破棄すると良いでしょう。

受信クライアントは、イベントを確実に受け取るために永続的な接続を維持しなければなりません。

14.7.9 切断の処理

プログラムが受信クライアントの場合は、EVM デーモンからの切断を適切に処理することが特に重要です。通常の実操作では切断は発生しませんが、システムをテストしている場合や障害が検出された場合は、切断が発生することがあります。EVM デーモンが中断されると、通常数秒のうちに `Essential Services Monitor` デーモンがこれを自動的に再開させます。詳細は、`esmd(8)` を参照してください。

接続を処理する関数からの戻りコードは、必ず検査する必要がありますが、受信クライアントではこれが特に重要です。受信クライアントは、接続上でのアクティビティを待機していることが多いからです。

切断が発生した場合、プログラムは `select()` または `EvmConnWait()` 呼び出しを中断し、後続の `EvmConnCheck()` と `EvmConnDispatch()` への呼び出しで障害状態のコードが返されます。このとき、`select()` または

`EvmConnWait()` 呼び出しにすぐに戻らないでください。CPU にバインドされたループが発生します。代わりに、状態コードが接続エラーを示しているかどうかを判断し、接続エラーの場合は `EvmConnDestroy()` を使用して接続を破壊してから、再接続を試みます。再接続の最初の試行に失敗した場合は、一定の間隔が経過してから接続が再試行されるように設定して、接続が再度確立されるまで定期的に再試行を繰り返します。どのような障害でも、通常デーモンはその後自動的に再開されるため、再試行の間隔は、最初の 60 秒は 1 秒に 1 回、その後は接続が回復するまで 5 秒おきにすることをお勧めします。

接続関数から返されるエラー・コードは、切断以外の状態を示していることもあります。特に、シグナルの結果として、プログラムが `EvmConnWait()` を中断することがあります。この場合は、`EvmConnWait()` 呼び出しにすぐに戻ってください。

プログラムが発信クライアントの場合、切断を処理しなければならないのは、通常は永続的な接続を行っている場合だけです (14.7.8 項を参照)。

`EvmEventPost()` や `EvmEventPostVa()` からのリターン状態をチェックし、必要に応じて再度接続を確立し、発信動作をやり直します。

14.7.10 失われたイベント

イベントの動きが活発なシステムでは、発信されたイベントを多く受信するクライアントが、イベントの一部を受信できなくなることがあります。クライアントが長時間かけてイベントを処理する場合 (たとえば、各イベントをディスクに書き込まなければならないなど) では、起こりやすくなります。このようなクライアントに EVM logger があります。これは、イベントを失ったときにそれを認識し、特別な “missed events” を記録することで、その失敗を通知します。アプリケーションがイベントを失うおそれがある場合は、その危険が最小になるようにシステムを構成し、プログラムが適切に対処するようにしてください。

EVM デーモンから、受信クライアントにイベントを送るときに、固定サイズの通信 (送信および受信) バッファによる接続で行います。クライアントが適切な時間内に受信イベントを処理することができず、イベントの負荷が高い場合、バッファが満杯になり、デーモンはその後イベントを送信できなくなります。省略時の受信バッファ・サイズは、`sysconfig` パラメータ `sb_max` で定義されたシステムワイド・ソケット・バッファの最大値に設定

されています。送信バッファのサイズは、ソケット・バッファに対するシステムの省略時の値、32,767 に設定されています。

EVM デーモンは多くのシステム・コンポーネントとアプリケーションに対して重要なリソースなので、クライアントがバッファのクリアを待っている間にブロックすることはできません。その結果、バッファが満杯のために接続バッファに書き込めない場合、デーモンはその接続にブロックされたというマークを付け、他の動作を続けます。その後クライアントが入力を読み、バッファに空き領域がある場合、EVM デーモンは失敗した書き込みを完了し、クライアントはイベントを受信します。ただし、その間に他のイベントが到着し、ブロックされた受信者に送信する必要があっても、デーモンはそれを送信しません。その代わりに、失った回数をカウントし、接続のブロックが解除された時点でその数を受信者に通知します。受信者は適切な対処が必要ですが、どのイベントを失ったかを知る方法はありません。

省略時の受信バッファ・サイズを大きくした結果、イベントが失われる可能性は旧バージョンよりも低くなりました。しかし極端な状況では受信イベントを失う危険性はまだあります。このようなことが起こる可能性は、いくつかの要因によって決まります。

- システムの速度と負荷
- 受信対象イベント
- イベントが発信される頻度
- 接続の入力バッファ・サイズ
- 発信されるイベントのサイズ

着信イベントを失うリスクを最小限に抑えるには、着信イベントをできるだけ素早く、かつ効率的に処理できるように、また、イベントを失ったことが通知されたときには適切な対処を行うように、アプリケーションを設計します。失ったイベントを処理するプログラムの例については、14.7.12.10 項を参照してください。

アプリケーションがイベントを続けて失う場合は、システム・パラメータを変更して受信バッファ・サイズを大きくすることができます。受信バッファ・サイズは、省略時のシステム・ソケット・バッファの最大サイズに設定されています。このパラメータの現在のサイズを調べるには、次のコマンドを入力します。

```
sysconfig -q socket sb_max
```

このパラメータの実行時の値を変更して、その値が次のリブートまで有効になるようにするには、次のコマンドを入力します。

```
sysconfig -r socket sb_max=new_value
```

この変更は、新しいEVM 接続にのみ影響します。

変更を持続的にするには、`sysconfigdb` または `dxkerneltuner` を使用して変更します。詳細については、`sysconfigdb(8)` または `dxkerneltuner(8)` を参照してください。

14.7.11 イベント・フィルタの使用

イベント・フィルタは、ユーザが関心を持つイベント・セットを識別するときに使用します。イベント・フィルタを設定すると、フィルタ・エバリュエータに対して、対象となるイベントを定義する文字列が渡されます。次に、一連のイベントが渡されて、フィルタを通過できるかどうかを示す論理値が各イベントに返されます。

EVM の受信クライアント・プログラムでは、フィルタは受信対象のイベントを指定するときに使用されます。また、受信したイベントに対するアクションを決定するときにも使用されます。返されるイベントを制限するために、コマンド行ユーティリティで起動オプションとして使用することもできます。コマンド行ユーティリティでフィルタを使用する方法については、『システム管理ガイド』を参照してください。フィルタの構文についての詳細は、`EvmFilter(5)` を参照してください。

フィルタの使用方法についての詳細は、14.7.12.8 項を参照してください。

14.7.12 EVM プログラミング操作の例

以降の各項では、次の操作の例を示します。

- 簡単なイベント操作の実行 (14.7.12.1 項)
- 可変長の引数リストの使用 (14.7.12.2 項)
- 変数の追加と取得 (14.7.12.3 項)
- イベントの発信 (14.7.12.4 項)
- イベントの読み取りと書き込み (14.7.12.5 項)

- イベント通知の受信 (14.7.12.6 項)
- 複数の入出力ソースの処理 (14.7.12.7 項)
- フィルタ・エバリュエータの使用 (14.7.12.8 項)
- イベント名の照合 (14.7.12.9 項)
- 失われたイベントの処理 (14.7.12.10 項)

14.7.12.1 簡単なイベント操作の実行

すべての EVM クライアントは、標準のデータ項目および変数で構成される不透明なバイナリ構造である EVM イベントを処理する必要があります。例 14-2 は、イベントを作成して、そのイベントに項目を追加し、次にそのイベントから項目を取得する場合の例です。

この例では、次の関数について説明します。

- `EvmEventCreate` — 空のイベントを作成する (詳細は、`EvmEventCreate(3)` を参照)。
- `EvmEventDestroy` — 以前に作成されたイベントを破壊して、メモリを解放する。この関数は、イベントを解放する必要がある場合に使用する。イベントの参照は、ヒープに割り当てられている構造体をポイントしているが、その構造体にはほかの構造体への参照が含まれるため、イベント参照に直接 `free()` を使用するとメモリの損失が発生する (詳細は、`EvmEventDestroy(3)` を参照)。
- `EvmItemSet` — イベントのデータ項目に値を設定する。設定する項目と変数は、`EvmEventCreateVa` と同じ (詳細は `EvmItemSet(3)` を参照)。
- `EvmItemGet` — 指定されたイベント・データ項目の値を返す (詳細は `EvmItemGet(3)` を参照)。
- `EvmItemRelease` — `EvmItemGet()` を使用して、イベントの指定されたデータ項目を取得したときに割り当てられたメモリを解放する (詳細は、`EvmItemRelease(3)` を参照)。

例 14-2: 簡単なイベント操作の実行

```
#include <stdio.h>
#include <evm/evm.h>

main()
{
    EvmEvent_t      event;
```

例 14-2: 簡単なイベント操作の実行 (続き)

```
EvmItemValue_t    itemval;
EvmStatus_t       status;

EvmEventCreate(&event); 1

EvmItemSet(event, EvmITEM_NAME, "myco.examples.app.started"); 2
EvmItemSet(event, EvmITEM_PRIORITY, 200);

status = EvmItemGet(event, EvmITEM_NAME, &itemval); 3
if (status == EvmERROR_NONE)
{
    fprintf(stdout, "Event name: %s\n", itemval.NAME);
    EvmItemRelease(EvmITEM_NAME, itemval);
}

EvmEventDestroy(event); 4
}
```

- 1 EvmEventCreate() を使用して空のイベントを作成します。この関数を使用する場合は、イベント・ハンドルへのポインタを指定して、標準データ項目が設定されていないイベントを受信します。イベントは空ですが、メモリは使用するので、後で EvmEventDestroy() を使用して領域を解放する必要があります。
- 2 標準データ項目をイベントに追加するには、EvmItemSet() を使用します。ただし、ほとんどの場合、プログラムで追加する項目はイベントの名前だけです。その他の標準データ項目は、イベントが発信されたときに自動的に追加されます。または、イベント・テンプレートに設定してください。設定可能な項目の一覧については、EvmItemSet(3) を参照してください。
- 3 イベントの項目を取得するには、EvmItemGet() を使用します。項目の値は、イベントから EvmItemValue_t 構造体を介して参照される記憶域にコピーされるため、使い終わったら EvmItemRelease() を使用してその記憶域を解放する必要があります。項目を取得しても、その項目はイベントからは削除されません。また、コピーを受け取るため、必要なコピー数を取得できます。

このコード例では、イベントの名前 (既に追加したもの) を取得し、その値をプリントしてから、記憶域を解放します。イベント内に存在しない項目を要求していることもあるので、取得操作からの戻り状態は必ず確認します。

- 4 イベントの処理が終了したら、イベントによって使用されていた記憶域を解放します。

14.7.12.2 可変長の引数リストの使用

`varargs` (可変長引数リスト) 版の作成関数を使用してイベントの作成および項目の追加を1つのステップで行うと、コードのサイズを削減して、効率を向上させることができます。また、`varargs` 版の項目設定関数を使用すると、既存のイベントに対して効率的に項目を追加することができます。

例 14-3 では、次の関数について説明します。

- `EvmEventCreateVa` — 1 回の呼び出しでイベントを作成し、項目の名前と値を設定する (詳細は `EvmEventCreateVa(3)` を参照)。
- `EvmItemSetVa` — イベントのデータ項目の値を設定する。設定する項目と変数のリストは、`EvmEventCreateVa` と同じ (詳細は `EvmItemSetVa(3)` を参照)。

例 14-3: 可変長の引数リストの使用

```
#include <stdio.h>
#include <evm/evm.h>

main()
{
    EvmEvent_t      event;

    EvmEventCreateVa(&event,
                    EvmITEM_NAME, "myco.examples.app.started",
                    EvmITEM_PRIORITY, 200,
                    EvmITEM_NONE);

    EvmItemSetVa(event,
                 EvmITEM_NAME, "myco.examples.app.finished",
                 EvmITEM_PRIORITY, 100,
                 EvmITEM_NONE);

    EvmItemSetVa(event,
                 EvmITEM_VAR_UINT16, "exit_code", 17,
                 EvmITEM_VAR_STRING, "progname", "my_app",
                 EvmITEM_NONE);

    EvmEventDump(event, stdout);

    EvmEventDestroy(event);
}
```

- 1 `EvmEventCreateVa()` に指定する各項目には、識別子と値が必要です。引数リストの最後には、`EvmITEM_NONE` 識別子を付けます。

- ② `varargs` 版の `EvmItemSet()` には、`EvmEventCreateVa()` と同じ形式の引数リストを使用します。この例では、イベントに既に設定されている項目を追加しているため、実際には前の値が新しい値に置き換わるだけです。
- ③ 変数データ項目を `varargs` リストに入れることは、項目識別子 `EvmITEM_VAR_Xxx` を用いて簡単に実行できます。ここで `Xxx` は変数の型です。このようにして変数を含める場合、変数を記述するために、識別子の後に正しい数の引数を続けて指定することが重要です。2 つの引数を必ず付加しなければなりません。これは、変数名を含む文字列と値引数です。値引数の型は変数の型に応じて変わります。この例では、最初の変数には文字列が必要で、2 番目の変数には整数が必要です。変数の型によっては、さらに変数が必要になることがあります。詳細は `EvmItemSet(3)` を参照してください。
- ④ `EvmEventDump()` を呼び出すと、イベントがフォーマットされて標準出力に表示されるため、データ項目と変数が期待どおりに追加されたことを確認できます。
- ⑤ イベントの処理が終了したら、これが使用していた記憶域を解放します。

14.7.12.3 変数の追加と取得

上記の例では、変数項目は `EvmItemSetVa()` を用いて `varargs` リストに入れることでイベントに追加されています。変数の値は、引数として項目識別子を取る `varargs` 関数のいずれかをを用いて追加または変更できます。

例 14-4 では、既存のイベントに変数データ項目を追加するもうひとつの方法を示し、また、変数の値を取得する方法も示します。EVM ライブラリには、変数について記述するデータ構造体を使用する基本的な `get` および `set` 関数と、構造体の処理をカプセル化してプログラミングを簡単にする便利な関数のセットが含まれています。この例では、両方の型の関数の使い方を示します。

この例では、次の関数について説明します。

- `EvmVarSet` — 指定された変数データ項目の値をイベントに設定する。この関数は、変数の追加および既存の変数の値の変更に使用する（詳細は、`EvmVarSet(3)` を参照）。

- `EvmVarGet` — 指定された変数構造体内の指定されたイベント変数の詳細情報を返す。呼び出し側は、`EvmVarRelease()` を呼び出して、変数が使用したメモリを解放する必要がある (詳細は、`EvmVarGet(3)` を参照)。
- `EvmVarRelease` — `EvmVarGet()` を使用して、指定された変数をイベントから取得したときに割り当てられたメモリを解放する。指定された変数構造を解放するのではなく、構造が参照するヒープ記憶域だけを解放する (詳細は、`EvmVarRelease(3)` を参照)。

例 14-4: 変数の追加と取得

```
#include <stdio.h>
#include <evm/evm.h>
void main()
{
    EvmEvent_t      event;
    EvmStatus_t     status;
    EvmVarValue_t   varval_1, varval_2;
    EvmVarStruct_t  varinfo;
    EvmString_t     progame;
    EvmUInt16_t     exit_code;

    EvmEventCreateVa(&event,
                    EvmITEM_NAME, "myco.examples.app.finished",
                    EvmITEM_NONE);

    /*
     * Set and retrieve some values using basic set/get functions:
     */
    varval_1.STRING = "my_app";
    varval_2.UINT16 = 17;

    EvmVarSet(event, "progame", EvmTYPE_STRING, varval_1, 0, 0);
    EvmVarSet(event, "exit_code", EvmTYPE_UINT16, varval_2, 0, 0);

    status = EvmVarGet(event, "progame", &varinfo);
    if (status == EvmERROR_NONE)
    {
        fprintf(stdout, "Program name: %s\n", varinfo.value.STRING);
        EvmVarRelease(&varinfo);
    }

    status = EvmVarGet(event, "exit_code", &varinfo);
    if (status == EvmERROR_NONE)
    {
        fprintf(stdout, "Exit code: %d\n", varinfo.value.UINT16);
        EvmVarRelease(&varinfo);
    }

    /*
     * Set and retrieve the same values using convenience functions:
     */
    EvmVarSetString(event, "progame", "my_app");
    EvmVarSetUInt16(event, "exit_code", 17);

    status = EvmVarGetString(event, "progame", &progame, NULL);
    if (status == EvmERROR_NONE)
    {
        fprintf(stdout, "Program name: %s\n", progame);
        free(progame);
    }

    status = EvmVarGetUInt16(event, "exit_code", &exit_code);
    if (status == EvmERROR_NONE)
```

例 14-4: 変数の追加と取得 (続き)

```
    fprintf(stdout, "Exit code: %d\n", exit_code);  
    EvmEventDestroy(event);  
}
```

- ❶ プリミティブ関数を使用してイベントに変数を追加するには、まず `EvmVarValue_t` 型の共用体に値を設定します。共用体のメンバの名前は、EVM 変数型の名前と同じです。
- ❷ `EvmVarSet()` を使用して、イベントに変数を追加します。変数にはわかりやすい名前を付けます。最後の 2 つの引数は、不透明な変数を追加するか、または文字列変数に I18N メッセージを設定したとき以外は、0 に設定されます。
- ❸ 変数の値を取得するには、変数の名前を `EvmVarGet()` に渡します。`EvmVarGet()` は、値を `EvmVarStruct_t` 構造体にコピーします。この構造体には、変数の名前、型、およびサイズも格納されているので、汎用コードを作成して、任意の型の変数进行处理することができます。変数を取得しても変数はイベントから削除されないで、何回でも変数を取得できます。返された情報によってヒープ・メモリの空間が使用されるので、値を使い終わったら、`EvmVarRelease()` を使用してクリーンアップする必要があります。
- ❹ `EvmVarSetString()` および `EvmVarSetUint16()` 関数は、`EvmVarSetXxx()` ファミリの他の関数とともに、`EvmVarSet()` 関数のシンプルな代用品として使用できます。これらは必要な引数が少なく、値の構造体を設定する必要もありません。

文字列変数に I18N メッセージ ID を指定する場合は、`EvmVarSetStringI18N()` 関数を使用します。
- ❺ `EvmVarGetString()` および `EvmVarGetUint16()` 関数は、`EvmVarGetXxx()` ファミリの他の関数とともに、`EvmVarGet()` 関数のシンプルな代用品として使用できます。これらは必要な引数が少なく、変数情報の構造体をセットアップして `EvmVarRelease()` を呼び出す取得操作に従う必要もありません。
- ❻ 簡易関数を用いて文字列または不透明な変数を取得する場合、この関数は値をヒープ領域に割り当て、そのヒープへのポインタを返しま

す。この値を使い終えたときは、`free()` を用いてヒープ領域を解放しなければなりません。

- 7 イベントの処理が終了したら、イベントによって使用されていた記憶域を解放します。

14.7.12.4 イベントの発信

イベントを作成したら、多くの場合、発信します。イベントを発信すると、EVM デーモンによって受信者に配信されます。イベントを発信するには、デーモンに対して発信接続を作成しておく必要があります。

例 14-5 は、接続を作成し、イベントを発信して、接続を切断する方法の例です。

この例では、次の関数について説明します。

- `EvmConnCreate` — アプリケーション・プログラムと EVM 間の接続を確立し、入出力アクティビティの処理方法を定義する。発信、リッスン(受信)、またはサービスの各接続に対して、別個の接続を確立する(詳細は、`EvmConnection(5)` および `EvmConnCreate(3)` を参照)。
- `EvmEventPost` — EVM にイベントを発信する(詳細は、`EvmEventPost(3)` を参照)。
- `EvmConnDestroy` — 指定の接続を破壊する(詳細は、`EvmConnDestroy(3)` を参照)。

`EvmConnCreate()` の代わりに、シンプルな `EvmConnCreatePoster()` マクロを用いて発信接続を行うことができます。このマクロは必要な引数が `EvmConnCreate()` よりも少なく、指定できる接続オプションが少ないものの、ローカル・システムにイベントを発信する必要のあるアプリケーションのほとんどで使用できます。

プログラムで一時的な EVM 接続を使用する場合は、`EvmConnCreate()` や `EvmConnDestroy()` の呼び出しを省略して、`EvmEventPost()` への接続引数として `NULL` を渡すこともできます。接続の確立には処理時間がかかるので、プログラムに適していれば、一時的な接続のみを使用してください。詳細は、14.7.8 項 および `EvmEventPost(3)` を参照してください。

`EvmEventPostVa()` 関数を使用すると、イベントの作成、発信、廃棄を 1 回の呼び出しで行えます。詳細は `EvmEventPost(3)` を参照してください。

例 14-5: イベントの発信

```
#include <stdio.h>
#include <evm/evm.h>

void main()
{
    EvmEvent_t      event;
    EvmStatus_t     status;
    EvmConnection_t conn;

    status = EvmConnCreate(EvmCONNECTION_POST, EvmRESPONSE_WAIT, 1,
                          NULL, NULL, NULL, &conn);
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Failed to create EVM posting connection\n");
        exit(1);
    }

    EvmEventCreateVa(&event,
                    EvmITEM_NAME, "myco.examples.app.error_detected",
                    EvmITEM_NONE);

    status = EvmEventPost(conn, event);
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Failed to post event\n");
        exit(1);
    }

    EvmEventDestroy(event);
    EvmConnDestroy(conn);
}
```

- ❶ EVM デーモンへの接続を作成するには、`EvmConnCreate()` を使用します。接続は、プログラムを終了するか、または `EvmConnDestroy()` を使用して明示的に破壊するまで切断されません。`EvmConnCreate()` の最初の 2 つの引数では、接続がイベントの発信に使用されること、および EVM デーモンがイベントの受諾を肯定応答してから発信関数が戻ることを指定しています。このため、発信に失敗した場合は、対応する処置を実行できます (その他の応答オプションについては、`EvmConnCreate(3)` を参照)。3 番目の引数に対する `NULL` 値は、ローカル・システム上で動作する EVM デーモンに接続することを示します。ほとんどの場合、ここには `NULL` を指定します。リモート接続についての詳細は、`EvmConnCreate(3)` を参照してください。4 番目と 5 番目の値は、ほかの応答オプション用なので、待機モード応答の場合は `NULL` を指定します。最後の引数を使用して、接続のハンドルを取得します。この接続で今後呼び出しを実行するときは、この値を指定しなければなりません。
- ❷ イベントを作成して発信します。

- ③ イベントおよび接続を破壊して、クリーンアップします。定期的にイベントを発信する場合は接続を破壊しないで、今後のすべてのイベントに対して再使用することをお勧めします。こうしておくで、発信するたびに接続を再確立するオーバーヘッドを削減できます。

14.7.12.5 イベントの読み取りと書き込み

次のいずれかの操作を実行するプログラムを作成する場合は、EVM の読み取りおよび書き込み関数を使用する必要があります。

- ファイルにイベントを格納する
- パイプまたはソケット接続を用いて、イベントを別のプロセスに渡す
- ファイルに格納したイベントを分析する
- EVM デモン以外のプロセスからイベントを受信する

標準の UNIX 書き込み関数を使用して、イベントを直接書き込むことはできません。これは、イベント・ハンドルには、イベント本体へのポインタ以外は設定されていないうえ、イベントが変更されるたびにイベント本体の位置が変更される可能性があるためです。逆に、イベントを読み込むときは、本体を読み込むだけでは十分ではありません。ハンドルを作成し、API 関数を介してイベントを参照できるようにする必要があります。

例 14-6 は、ファイルへのイベントの書き込み、ファイルからプログラムへのイベントの読み取り、およびイベントの有効性の確認を行う方法の例です。

この例では、次の関数について説明します。

- `EvmEventWrite` — オープン・ファイル記述子にイベントを書き込む (詳細は、`EvmEventWrite(3)` を参照)。
- `EvmEventRead` — 新しいイベント構造体を作成し、ファイル記述子から読み込まれたイベントを設定する。新しいイベントは、`EvmEventDestroy()` を使用して解放する必要がある (詳細は、`EvmEventRead(3)` を参照)。
- `EvmEventValidate` — イベントについてデータの一貫性をチェックする。このチェックは、接続を介して受信したり、記憶域から取得したイベントの有効性を確認するために行う (詳細は、`EvmEventValidate(3)` を参照)。

例 14-6: イベントの読み取りと書き込み

```
#include <stdio.h>
#include <fcntl.h>
#include <evm/evm.h>

void
main()
{
    EvmEvent_t      event_in, event_out;
    EvmStatus_t     status;
    EvmItemValue_t  itemval;
    int             fd;

    EvmEventCreateVa(&event_out,                                ❶
                    EvmITEM_NAME, "myco.examples.app.saved_event",
                    EvmITEM_NONE);

    fd = open("eventlog", O_RDWR | O_CREAT | O_TRUNC,          ❷
              S_IRUSR | S_IWUSR);

    if (fd < 0)
    {
        fprintf(stderr, "Failed to open output log file\n");
        exit(1);
    }

    status = EvmEventWrite(fd, event_out);
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Failed to write event to log file\n");
        exit(1);
    }

    lseek(fd, 0, SEEK_SET);                                     ❸
    status = EvmEventRead(fd, &event_in);
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Failed to read event from log file\n");
        exit(1);
    }

    status = EvmEventValidate(event_in);                         ❹
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Event read from logfile is invalid");
        exit(1);
    }

    status = EvmItemGet(event_in, EvmITEM_NAME, &itemval);      ❺
    if (status == EvmERROR_NONE)
    {
        fprintf(stdout, "Event name: %s\n", itemval.NAME);
        EvmItemRelease(EvmITEM_NAME, itemval);
    }

    EvmEventDestroy(event_in);                                  ❻
    EvmEventDestroy(event_out);
}
```

- ❶ 名前が設定されたイベントを作成します。
- ❷ 出力ログ・ファイルを作成し、`EvmEventWrite()` を使用してイベントを書き込みます。イベントは、別のプロセスへのパイプを含め、任意のファイル記述子に書き込むことができます。ただし、イベントはバ

イナリ・データ・パッケージであるため、端末やプリンタには直接書き込まないでください。

- ③ `EvmEventRead()` を使用してイベントを読み戻します。このとき、別のイベントが作成され、イベント・ハンドル自体ではなく、イベント・ハンドルへのポインタを設定する必要があることに注意してください。
- ④ 着信イベントは、このプロセスによって制御されていないため、一貫性を確認することが重要です。ファイルからイベントを読み込むたびに、または EVM デーモン以外のプロセスからイベントを受信するたびに、`EvmEventValidate()` 関数を使用して確認します。
- ⑤ 読み込んだイベントが、既に関数に書き込んだイベントと同じものであることを確認するには、名前を取得して表示します。
- ⑥ イベントによって使用された領域を解放します。

14.7.12.6 イベント通知の受信

イベント通知を受信するプログラムでは、次の操作を実行する必要があります。

- EVM デーモンに対してリッスン接続を作成する。
- EVM デーモンにフィルタ文字列を渡して、関心のあるイベントを通知する。
- 接続上のイベント関連アクティビティを監視して、イベントが着信したらただちに処理できるようにする。

例 14-7 は、イベントの受信を待機し、イベントが着信するたびに `stdout` に表示される例です。

この例では、次の関数について説明します。

- `EvmConnSubscribe` — 指定したフィルタと一致する発信イベントの通知を要求する (詳細は、`EvmConnSubscribe(3)` を参照)。
- `EvmConnWait` — 指定の接続上で、アクティビティが検出されるまでブロックする。アクティビティが検出されると、呼び出し側のプログラムによって `EvmConnDispatch()` が呼び出され、そのアクティビティが処理される (詳細は、`EvmConnWait(3)` を参照)。

- `EvmConnDispatch` — 必要に応じてプログラムのコールバック関数を呼び出し、指定の接続上での未処理の入出力を処理する (詳細は、`EvmConnDispatch(3)` を参照)。
- `EvmEventFormat` — イベントをフォーマットする (詳細は、`EvmEventFormat(3)` を参照)。

注意

`EvmConnCreate()` の代わりに、シンプルな `EvmConnCreateSubscriber()` マクロを用いて受信接続を行うことができます。このマクロは必要な引数が `EvmConnCreate()` よりも少なく、指定できる接続オプションは少ないものの、受信アプリケーションのほとんどで使用できます。詳細は `EvmConnCreate(3)` を参照してください。

例 14-7: イベント通知の受信

```
#include <stdio.h>
#include <evm/evm.h>

void
EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
        EvmCallbackData_t *cbdata);

/*=====
 * Function: main()
 *=====*/
main()
{
    EvmConnection_t    conn;
    EvmStatus_t        status;

    status = EvmConnCreate(EvmCONNECTION_LISTEN, EvmRESPONSE_CALLBACK,
                          NULL, EventCB, NULL, &conn); 1
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Failed to create EVM listening connection\n");
        exit(1);
    }

    status = EvmConnSubscribe(conn, NULL, "[name *.evm.msg.user]"); 2
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Failed to subscribe for event notification\n");
        exit(1);
    }

    for (;;) 3
    {
        status = EvmConnWait(conn, NULL);
        if (status == EvmERROR_NONE)
        {
            fprintf(stderr, "Connection error\n");
            exit(1);
        }

        if (EvmConnDispatch(conn) != EvmERROR_NONE)
```

例 14-7: イベント通知の受信 (続き)

```
        {   fprintf(stderr, "Connection dispatch error\n");
            exit(1);
        }
    }
}

/*=====
 * Function: EventCB()
 *=====*/
void
EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg, 4
        EvmCallbackData_t *cbdata)
{   char buff[256];

    switch (cbdata->reason) { 5
    case   EvmREASON_EVENT_DELIVERED:
        EvmEventFormat(buff, sizeof(buff), cbdata->event);
        fprintf(stdout, "Event: %s\n", buff);

        EvmEventDestroy(cbdata->event); 6
        break;
    default: 7
        break;
    }
}
```

- 1 `EvmConnCreate()` を使用して EVM デモンへの接続を確立します。ここではリッスン接続を指定します。このコード例では、次の引数が指定されています。
- `EvmConnCreate()` の最初の 2 つの引数には、接続がリッスンに使用されること、およびイベントが着信した場合はコールバック関数を介して通知することが指定されている。
 - 3 番目の引数には `NULL` が指定されているが、ローカル EVM デモンに接続することを示す。リモート接続についての詳細は、`EvmConnCreate(3)` を参照。
 - 4 番目の引数には、イベントが着信した場合に呼び出すコールバック関数 (`EventCB`) が指定されている。
 - 5 番目の引数は、コールバック引数であり、この接続で使用するために呼び出されるたびにコールバック関数に渡される値である。この引数は任意の目的で使用できるが、この例では `NULL` に設定されている。
 - 最後の引数は、接続へのハンドルを受信する。

- ② 次のステップでは、`EvmConnSubscribe()` 関数を使用して EVM デーモンに受信したいイベントを通知しています。ここでは、`evmpost` で発信できるユーザ・メッセージを待機しています。これらのイベントには、`sys.unix.evm.msg.user` という名前が付いています。`EvmConnSubscribe()` を呼び出すと、`EvmConnDispatch()` を呼び出したときに、コールバック関数が `EvmREASON_SUBSCRIBE_COMPLETE` という理由コードで呼び出されます。
- ③ この例では、イベントの着信を待機する以外に処理が記述されていません。このため、`EvmConnWait()` を使用して接続上のアクティビティを監視しながら、永久にループします。`EvmConnWait()` の 2 番目の引数として渡された `NULL` は、アクティビティが存在しない場合でもタイムアウトにならないことを示します。`EvmConnWait()` は、アクティビティが発生するたびに帰りますが、イベントが着信しているとは限りません。たとえば、EVM デーモンからほかのメッセージが送信された場合、またはプロセスがシグナルによって割り込まれた場合にも帰ります。この関数が正常に戻った場合は、`EvmConnDispatch()` を呼び出してアクティビティを処理する必要があります。通常は、コールバック関数 `EventCB()` が呼び出されますが、いつもとは限りません。
- ④ `EvmConnDispatch()` では、アプリケーション・コードによる処理を要求するメッセージがデーモンから読み込まれたときに、コールバック関数が呼び出されます。このメッセージには、着信イベントも含まれます。ただし、アプリケーションでは、ほかの種類のメッセージも考慮する必要があります。このコード例では、次の引数が指定されています。
- コールバック関数の最初の引数は、接続ハンドルである。ある状況で接続を識別するときに使用すると便利であるが、用途は限られている。
 - 2 番目の引数は、接続を確立したときに指定したコールバック引数である。この値を使用して接続を識別することができ、C++ コードの場合は、オブジェクト・インスタンスにポインタを渡すときに使用することもできるが、使用しなくてもよい。
 - 最後の引数は、コールバック・データであり、コールバック情報が含まれている構造体へのポインタである。コードが呼び出された理由を確認するには、コールバック・データの構造体を検査する必要がある。構造体には、理由コード、状態値、およびイベントへのポ

インタ (イベントが配信されている場合) が含まれている。この構造体についての詳細は、`EvmCallback(5)` を参照。

この例では、理由コード `EvmREASON_EVENT_DELIVERED` に記述されているように、着信イベント・メッセージを待機しています。イベントが着信すると、フォーマット後に表示します。イベントによって使用される領域を解放するため、処理が終了したらイベントを破壊します。

- ⑤ 実行されるアクションは、コールバックの理由コードに依存しています。この場合、理由はイベントが配信されたことなので、`EvmEventFormat()` 関数を使用してイベントを表示用にフォーマットし、`stdout` に出力します。

`EvmEventFormat()` では、イベントのフォーマット・データ項目 (設定されている場合) を基にしてテキスト行が生成され、その結果が指定したバッファに格納されます。フォーマットによって、イベント本体が変更されることはありません。

- ⑥ 配信されたイベントによってヒープ領域が使用されています。イベントの処理が終了したら、アプリケーションでこの領域を解放する必要があります。
- ⑦ 既に受信要求を発行しているため、`EvmREASON_SUBSCRIBE_COMPLETE` というコールバック理由によってその関数も起動されます。この例の場合は、この理由コードおよびその他の理由コードは無視して構いません。

14.7.12.7 複数の入出力ソースの処理

イベントのリッスン以外に処理を行うプログラムを作成する場合は、イベントの着信を待機するときに `EvmEventWait()` 以外の関数を使用することもできます。たとえば、`select` システム・コールを使用して、EVM 接続など、複数のファイル記述子に対する入出力アクティビティを監視する方法があります。

例 14-8 は、EVM 接続を `stdin` からの入力とともに処理する方法の例です。

この例では、次の関数について説明します。

- `EvmConnFdGet` — 指定の接続に関連付けられたファイル記述子 (ファイル番号) を返す (詳細は、`EvmConnFdGet(3)` を参照)。
- `EvmConnCheck` — 指定の接続上で、未処理の入出力アクティビティがないかどうかを確認する (詳細は、`EvmConnCheck(3)` を参照)。

例 14-8: 複数の入出力ソースの処理

```
#include <stdio.h>
#include <sys/time.h>
#include <evm/evm.h>

void HandleInput();
void EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
             EvmCallbackData_t *cbdata);

/*=====
 * Function: main()
 *=====*/
main()
{
    EvmConnection_t    conn;
    EvmStatus_t         status;
    fd_set              read_fds;
    int                 conn_fd;
    EvmBoolean_t        io_waiting;

    status = EvmConnCreate(EvmCONNECTION_LISTEN, EvmRESPONSE_CALLBACK,
                          NULL, EventCB, NULL, &conn);
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Failed to create EVM listening connection\n");
        exit(1);
    }

    status = EvmConnSubscribe(conn, NULL, "[name sys.unix.evm.msg.user]");
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Failed to subscribe for event notification\n");
        exit(1);
    }

    EvmConnFdGet(conn, &conn_fd);

    for (;;)
    {
        FD_ZERO(&read_fds);
        FD_SET(fileno(stdin), &read_fds);
        FD_SET(conn_fd, &read_fds);

        select(FD_SETSIZE, &read_fds, NULL, NULL, NULL);

        if (FD_ISSET(fileno(stdin), &read_fds))
            HandleInput();

        status = EvmConnCheck(conn, &io_waiting);
        if (status != EvmERROR_NONE)
        {
            fprintf(stderr, "Connection error\n");
            exit(1);
        }
        if (io_waiting)
        {
            status = EvmConnDispatch(conn);
            if (status != EvmERROR_NONE)
            {
                fprintf(stderr, "Connection dispatch error\n");
                exit(1);
            }
        }
    }
}

/*=====
 * Function: HandleInput()
 *=====*/
void
```

例 14-8: 複数の入出力ソースの処理 (続き)

```
HandleInput() 4
{
    char buff[256];
    if (feof(stdin))
        exit(0);

    if (fgets(buff, sizeof(buff), stdin) == NULL)
        exit(0);

    if (buff[0] == '\n')
        exit(0);

    fprintf(stdout, buff);
}

/*****
 * Function: EventCB()
 *****/
void
EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg, 5
        EvmCallbackData_t *cbdata)
{
    char buff[256];

    switch (cbdata->reason) {
    case EvmREASON_EVENT_DELIVERED:

        EvmEventFormat(buff, sizeof(buff), cbdata->event);
        fprintf(stdout, "Event: %s\n", buff);

        EvmEventDestroy(cbdata->event);
        break;
    default:
        break;
    }
}
```

- 1 EvmConnFdGet() を使用して、接続に割り当てられたファイル記述子を検索します。
- 2 この例では、入出力アクティビティを監視するときに select を使用しています。select を使用すると、複数のファイル記述子を監視できます。select 呼び出しが戻ったときに、アクティビティが発生したファイル記述子を確認し、適切に処理します。
- 3 EvmConnCheck() を使用して、EVM 接続上で未処理のアクティビティが存在するかどうかを確認します。接続のファイル記述子がわかっているので、FD_ISSET() を使用することもできます。
- 4 HandleInput() 関数は、stdin から入力行を読み取り、stdout にエコーします。エラーが発生するか空の行を読み込んだ場合は、プログラムが終了します。

⑤ イベントのコールバック関数は、前の例で使用したものと同じです。

14.7.12.8 フィルタ・エバリュエータの使用

イベント受信者によっては、さまざまな基準を使用してイベントを監視し、着信イベントの属性に応じて異なる対応が必要になることがあります。たとえば、EVM ロガーでは、構成ファイルからフィルタ文字列を読み込んで、その文字列に記述されているすべてのイベントを受信します。このため、イベントを適切なログに書き込むには、イベントが着信したときに、イベントと一致するフィルタ文字列を識別する必要があります。

たとえば、EVM デーモンに対して複数の接続を作成し、接続ごとに別個のフィルタ文字列を使用して受信する方法があります。しかし、この方法の場合は、接続のオーバーヘッドが大きくなり、同じイベントが 2 つ以上の接続を介して送信される可能性が大きくなります。

もっと効率的な方法として、論理演算子 OR を使用して、フィルタ文字列をすべて取り込み、それらを結合して 1 つの論理文字列にする方法があります。結合された文字列は、単一の接続上で一致するイベントをすべて受信するときに使用することができます。結合された文字列は後で破棄できますが、元の文字列は保持しておく必要があります。接続の再確立またはフィルタの変更が必要になった場合は、元の文字列を使用して再受信することができます。

ただし、イベントが着信したときには、イベントの処理方法を決定するために、イベントと一致する元のフィルタ文字列を識別する必要があります。このとき、EVM フィルタ・エバリュエータを使用することができます。フィルタ・エバリュエータは、ユーザが作成できるオブジェクトで、フィルタ文字列とともにロードされ、フィルタと一致するイベントを識別する（一致が存在する場合）ために複数のイベントが渡されます。元の各フィルタ文字列に対して別個のエバリュエータを割り当てておくと、着信イベントごとにそれぞれのエバリュエータを適用して、イベントと一致するエバリュエータを識別することができます。

例 14-9 はこの方法の例です。簡単な 3 つのフィルタ文字列を使用して、一致が存在する場合は、着信イベントと一致するフィルタに応じて異なるメッセージを出力しています。

この例では、次の関数について説明します。

- `EvmFilterCreate` — フィルタ・エバリュエータのインスタンスを生成し、ハンドルを返す (詳細は、`EvmFilterCreate(3)` および `EvmFilter(5)` を参照)。
- `EvmFilterSet` — 後続の照合で使用するフィルタ文字列を、フィルタ・エバリュエータに渡す (詳細は、`EvmFilterSet(3)` および `EvmFilter(5)` を参照)。
- `EvmFilterTest` — 指定のイベントを、フィルタ・エバリュエータに現在関連付けられているフィルタ文字列と比較する。イベントとフィルタ文字列が一致した場合は、`EvmTRUE` を返し、一致しない場合は `EvmFALSE` を返す (詳細は、`EvmFilterTest(3)` および `EvmFilter(5)` を参照)。
- `EvmFilterDestroy` — フィルタ・エバリュエータを破壊し、関連付けられたリソースをすべて解放する (詳細は、`EvmFilterDestroy(3)` および `EvmFilter(5)` を参照)。

例 14-9: フィルタ・エバリュエータの使用

```
#include <stdio.h>
#include <sys/time.h>
#include <evm/evm.h>

void
EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
        EvmCallbackData_t *cbdata);

#define FILTER_1 "[name *.class_1]"
#define FILTER_2 "[name *.class_2]"
#define FILTER_3 "([name *.class_2] | [name *.class_3]) & [priority >= 300]"

/*=====
 * Function: main()
 *=====*/
main()
{
    EvmConnection_t    conn;
    Evmstatus_t        status;
    int                conn_fd;
    char                *filter_string;

    status = EvmConnCreate(EvmCONNECTION_LISTEN, EvmRESPONSE_CALLBACK,
                          NULL, EventCB, NULL, &conn);
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Failed to create EVM listening connection\n");
        exit(1);
    }

    filter_string = (char *)malloc(strlen(FILTER_1) + strlen(FILTER_2) +
                                   strlen(FILTER_3) + 30);
    sprintf(filter_string, "(%s) | (%s) | (%s)", FILTER_1, FILTER_2, FILTER_3);

    status = EvmConnSubscribe(conn, NULL, filter_string);
    if (status != EvmERROR_NONE)
```

例 14-9: フィルタ・エバリュエータの使用 (続き)

```
{    fprintf(stderr, "Failed to subscribe for event notification\n");
    exit(1);
}
free(filter_string);

for (;;)                                4
{    status = EvmConnWait(conn, NULL);
    if (status != EvmERROR_NONE)
    {    fprintf(stderr, "Connection error\n");
        exit(1);
    }
    if (EvmConnDispatch(conn) != EvmERROR_NONE)
    {    fprintf(stderr, "Connection dispatch error\n");
        exit(1);
    }
}
}

/*=====
 * Function: EventCB()
 *=====*/
void
EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
        EvmCallbackData_t *cbdata)
{    EvmBoolean_t    match;

    static EvmFilter_t  f1, f2, f3;
    static EvmBoolean_t filters_initialized = EvmFALSE;

    if (! filters_initialized)
    {    if (EvmFilterCreate(&f1) != EvmERROR_NONE)
        {    fprintf(stderr, "Failed to create filter evaluator\n");
            exit(1);
        }
        if (EvmFilterSet(f1, FILTER_1) != EvmERROR_NONE)
        {    fprintf(stderr, "Failed to set filter evaluator\n");
            exit(1);
        }

        EvmFilterCreate(&f2);
        EvmFilterSet(f2, FILTER_2);
        EvmFilterCreate(&f3);
        EvmFilterSet(f3, FILTER_3);

        filters_initialized = EvmTRUE;
    }

    switch (cbdata->reason) {
    case    EvmREASON_EVENT_DELIVERED:                                6

        EvmFilterTest(f1, cbdata->event, &match);
        if (match)
            fprintf(stdout, "Filter 1 event received\n");

        EvmFilterTest(f2, cbdata->event, &match);
        if (match)
            fprintf(stdout, "Filter 2 event received\n");

        EvmFilterTest(f3, cbdata->event, &match);
        if (match)
```

例 14-9: フィルタ・エバリュエータの使用 (続き)

```
        fprintf(stdout, "Filter 3 event received\n");

        EvmEventDestroy(cbdata->event);
        break;

    default:
        break;
    }
}
```

- ❶ 3つの簡単なフィルタ文字列を定義します。最初の2つは、名前だけを基にしてフィルタを実行します。3番目の文字列は、300以上の優先度が設定されたイベントのうち、2つの名前のいずれかと一致するイベントを選択します。
- ❷ 3つのフィルタ文字列を、1つの論理式に結合します。各部分文字列はカッコで囲まれ、論理演算子 OR で区切られています。この結果、イベントが1つ以上の部分文字列と一致したときに、EVM デモンから通知されます。
- ❸ 結合された文字列を使用してイベントを受信します。その後、文字列の使用が終了したため、その領域を解放します。何らかの理由で再受信が必要になった場合は、いつでも部分文字列を再度結合できます。
- ❹ エバリュエータを設定したら、受信するすべてのイベントに対して使用できるように保存します (つまり、静的要素にします)。もう1つの方法として、イベントを受信するたびに新しいエバリュエータを作成し、関数を終了するときに破壊することもできます。ただし、保存しておくと、セットアップおよび破棄の繰り返しによるオーバーヘッドを回避できます。
- ❺ `EvmFilterCreate()` を使用して3つのフィルタ・エバリュエータを作成し、3つのフィルタ文字列をロードします。わかりやすくするために、この例では最初のフィルタの戻り状態だけを確認していますが、製品コードでは各フィルタの戻り状態を確認してください。このコードは、イベントを最初に受信した場合にだけ実行されます。
- ❻ 前述のセクションで設定した3つのフィルタ・エバリュエータに対して、各着信イベントを検査し、イベントと一致したエバリュエータごとに異

なるメッセージを出力しています。イベントが複数のエバリュエータと一致することもあり、この場合は、複数のメッセージを出力します。

14.7.12.9 イベント名の照合

EVM の命名ポリシでは、基本イベント名の後に任意の数のコンポーネントを追加して拡張することができます。つまり、コンポーネントが追加されている可能性があるため、イベント名が元の名前と全く同じである保証はありません。

このため、イベント名を既知のイベント名と比較する場合は、通常の文字列比較関数を使用しないでください。コンポーネントが追加されている場合は、適切に名前を照合することができません。代わりに、EVM の名前照合関数を使用します。名前照合関数を使用すると、指定したパターンを基にしてイベント名が照合され、照合するイベント名にコンポーネントが追加されていても無視されます。また、名前パターンにワイルドカード文字を含めることもできます。

例 14-10 では、次の関数について説明します。

- `EvmEventNameMatch` — イベント名文字列 (ワイルドカード文字も使用可能) およびイベントを入力引数として指定できる。イベントが名前文字列と一致するかどうかを返す。

次の関数は、`EvmEventNameMatch` に関連しています。

- `EvmEventNameMatchStr` — イベント名をイベントから抽出せずに、文字列として指定する。

例 14-10: イベント名の照合

```
#include <stdio.h>
#include <evm/evm.h>

/*=====
 * Function: main()
 *=====*/
main()
{
    EvmStatus_t      status;
    EvmEvent_t        event;
    EvmBoolean_t      match;
    char              buff[80];

    while (EvmERROR_NONE == EvmEventRead(fileno(stdin), &event)) {
        {
            EvmEventNameMatch("*.msg", event, &match);
            if (match)
            {
                EvmEventFormat(buff, sizeof(buff), event);
                fprintf(stdout, "%s\n", buff);
            }
        }
    }
}
```

例 14-10: イベント名の照合 (続き)

```
}  
}
```

- 1 `stdin` からイベントを読み取り、ワイルドカード文字列 `*.msg` と一致するイベントだけを表示します。この種類のイベントの名前は通常 `sys.unix.evm.msg.user` または `sys.unix.evm.msg.admin` ですが、照合は機能します。

14.7.12.10 失われたイベントの処理

開発中のアプリケーションが、短期間に大量に発信されるイベントのセットを受信する場合、プログラムが EVM デーモンにより、1 つまたは複数のイベントを失ったと通知することがあります。これは、プログラムの処理量がひじょうに多く、新しくイベントが到着してもすぐに対処できず、未処理のイベントで通信バッファが満杯になってしまうような場合に起こりやすくなります。失われたイベントについての詳細は、14.7.10 項を参照してください。

例 14-11 では、イベントをすべて受信し、各イベントを到着時に `stdout` に表示します。これは、コールバック関数に `EvmREASON_EVENTS_MISSED` という理由を入れることで失われたイベントの通知を監視し、失ったイベントの数を含むメッセージを表示します。

例 14-11: 失われたイベントの処理

```
#include <stdio.h>  
#include <evm/evm.h>  
  
void EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,  
             EvmCallbackData_t *cbdata);  
  
/*=====*/  
* Function: main()  
*=====*/  
main()  
{  
    EvmConnection_t    conn;  
    EvmStatus_t        status;  
    int                conn_fd;  
  
    status = EvmConnCreate(EvmCONNECTION_LISTEN, EvmRESPONSE_CALLBACK,  
                          NULL, EventCB, NULL, &conn);  
    if (status != EvmERROR_NONE)  
    {  
        fprintf(stderr, "Failed to create EVM listening connection\n");  
        exit(1);  
    }  
}
```

例 14-11: 失われたイベントの処理 (続き)

```
status = EvmConnSubscribe(conn, NULL, "[name *]");
if (status != EvmERROR_NONE)
{
    fprintf(stderr, "Failed to subscribe for event notification\n"); ❶
    exit(1);
}

for (;;)
{
    status = EvmConnWait(conn, NULL);
    if (status != EvmERROR_NONE)
    {
        fprintf(stderr, "Connection error\n");
        exit(1);
    }
    if (EvmConnDispatch(conn) != EvmERROR_NONE)
    {
        fprintf(stderr, "Connection dispatch error\n");
        exit(1);
    }
}

}

/*****
 * Function: EventCB()
 *****/
void
EventCB(EvmConnection_t conn, EvmCallbackArg_t cbarg,
        EvmCallbackData_t *cbdata)
{
    char buff[256];

    switch (cbdata->reason) {
    case EvmREASON_EVENT_DELIVERED:
        EvmEventFormat(buff, sizeof(buff), cbdata->event);
        fprintf(stdout, "Event: %s\n", buff);

        EvmEventDestroy(cbdata->event);

        sleep(1); ❷
        break;

    case EvmREASON_EVENTS_MISSED: ❸
        fprintf(stdout, "*** Missed %d incoming events\n",
                cbdata->extension.eventMissedData.missedCount);
        break;

    default:
        break;
    }
}
```

- ❶ この例では、すべてのイベントの通知を受信しています。
- ❷ イベントが失われた状態を示すために、各イベントを受信した後に1秒間スリープしています。これは、負荷の超過が発生した状況をシミュ

レートしており、イベントの負荷が大きい場合は、入力接続バッファがいっぱいになることが確認されます。

- ③ イベントが失われたため、EVM デーモンがイベントを送信できない場合は、コールバック関数が理由コード `EvmREASON_EVENTS_MISSED` で呼び出されます。コールバック・データ・メンバ `extension.eventMissedData.missedCount` には、失われたイベント数が格納されます。

14.8 EVM へのイベント・チャネルの追加

イベント・チャネルという言葉は、イベント情報を公開または取得するために使用する機能を表します。また、次のいずれかを表す場合もあります。

- 単純なログ・ファイル
- イベント管理システム
- ステータス情報のスナップショットを取得するために実行されるプログラム

イベント・チャネルには、アクティブ・チャネルとパッシブ・チャネルがあります。アクティブ・チャネルでは、イベントが発生すると直ちに固有のイベント情報が EVM に発信されます。パッシブ・チャネルでは、イベント情報はチャネル内に蓄積され、検索するには EVM からのアクションが必要です。

イベント・チャネルは、正式なイベント通知メカニズムである必要はありません。情報の格納、あるいは状態または状態変更の表示を行うことができるものであれば、何でも構いません。次に例を示します。

- アプリケーションまたはシステム・コンポーネントで独自のログ・ファイルに状態情報を書き込む場合は、そのログ・ファイルの各エントリをイベントと見なすことができます。新しいエントリがあるかどうかについてログ・ファイルを定期的に確認し、新しいエントリが検出されるたびにイベントを発信することができます。ログ・ファイル自体も、過去のイベントを取得するための場所として機能します。
- 一部のアプリケーションおよびシステム・コンポーネントには、状態を照会する機能があります。これらのアプリケーションおよびシステム・コンポーネントでは、状態を定期的に監視し、重大な変更が検出された場合にイベントを発信することができます。このようなイベント・チャ

ネルには、通常、イベント情報を格納する手段はないため、EVM のロギング機能を使用してイベントのログを記録することができます。

- アプリケーションの中心部分でそのアプリケーションのイベント情報をすべて処理する場合は、イベント情報を EVM に転送できるように処理コードを変更することができます。これは、アクティブ・イベント・チャンネルの例です。

既存のイベント・チャンネルを EVM を介してアクセスできるようにする処理は、カプセル化と呼ばれます。

EVM イベント・チャンネルは、チャンネル構成ファイルを使用して構成します。このファイルは、EVM の起動時に EVM チャンネル・マネージャによって読み込まれ、また、チャンネル情報が必要な場合は、コマンド行ユーティリティでも使用されます。このファイルを変更する場合は、次のコマンドを入力して、チャンネル・マネージャに変更を通知する必要があります。

```
evmreload -c
```

チャンネル構成ファイルの構文については、`evmchannel.conf(4)` を参照してください。イベント・チャンネルをカプセル化するには、さまざまなチャンネル関数を処理する実行可能プログラムが必要です。次のようなチャンネル関数があります。

- 取得関数 — `evmget` が実行されたときに、チャンネルから過去のイベントを取得する。
- 詳細関数 — `evmshow` が `-d` オプションを指定して実行されたときに、イベント・ストリームを詳細に表示する。
- 説明関数 — `evmshow` が `-x` オプションを指定して実行されたときに、イベント・ストリームの説明テキストを提供する。この関数は、一般にチャンネルの詳細関数でも起動される。
- 監視関数 — EVM チャンネル・マネージャによって定期的に実行され、チャンネル状態をチェックし、必要に応じてイベントの発信を行う。
- クリーンアップ関数 — EVM チャンネル・マネージャによって毎日実行され、チャンネルに必要なクリーンアップ処理を行う。

これらの関数はすべてオプションです。チャンネル構成ファイルのチャンネル定義に適切なエントリを追加することにより、チャンネルに対して定義されま

す。チャンネル関数には任意の種類の実行可能ファイルを使用できますが、この後の各項で説明するとおり動作する必要があります。

一時ファイルを使用するチャンネル関数では、終了前に必ずクリーンアップを実行する必要があります。また、割り込みが発生した場合でも、クリーンアップを実行できなければなりません。

また、次の操作が必要です。

- 新しいイベント・チャンネルを介して発信または取得するイベントに対するイベント・テンプレートの追加。イベント・テンプレートの追加については、14.6.3 項を参照。
- 必要に応じて EVM 権限ファイルを変更することによる、イベントに適切な発信およびアクセス権限が割り当てられていることの確認。イベントに対するアクセスの制御方法についての詳細は、『システム管理ガイド』を参照。

14.8.1 取得関数

チャンネルの取得関数は、EVM の `get_server` によって実行されます。`get_server` は、EVM デーモンによって実行され、`evmget` を介して行われたイベント取得要求を処理します。この関数は常に `root` として実行されるので、適切なセキュリティ対策が必要です。

この関数では、次の呼び出し構文を使用します。

```
function-name [-f filter-string]
```

必要に応じて、その他の引数も関数に渡すことができます。チャンネル構成ファイルの該当する関数の行に、引数を指定してください。

取得関数を実行すると、イベントがチャンネルのログ・ファイルから取得されて、EVM のイベント・フォーマットに変換され、変換後の EVM イベントが `stdout` ストリームに書き込まれます。*filter-string* を指定すると、フィルタと一致するイベントだけが `stdout` に書き込まれます。エラー・メッセージは、`stderr` に書き込まれ、`evmget` に渡されてから、その `stderr` ストリームに出力されます。このため、エラー・メッセージがこの関数で発生したことを明示的に示す必要があります。`stdout` には、EVM イベント以外は書き込まないでください。

取得スクリプトの形式は、元のイベントの格納形式に大きく依存します。標準では、次のステップが実行されます。

1. `grep`、`awk` および `sed` などの UNIX の標準ツールか、または `perl` などのプログラミング言語を使用して、イベント行の選択および空白行とコメントの削除を行い、次のステップに必要な形式にフォーマットします。イベントが単一行テキストで構成され、各行のフォーマットが一定しており、タイムスタンプ、ホスト名、およびメッセージなどの項目がすべての行で同じ位置に配置されている場合、この作業は簡単に行うことができます。
2. 各行を EVM イベントに変換します。これを行うには、UNIX ツールを使用して、各行を `evmpost` への入力に適した形式にフォーマットし、発信するのではなく、`-r` オプションを使用して、EVM イベントを `stdout` に出力します。迅速に変換する方法として、EVM チャネル・ユーティリティの `/usr/share/evm/channels/bin/text2evm` を使用することもできます。このツールを使用する場合は、現在、次の形式で入力する必要があります。

```
evm-event-name date time host user message
```

各項目の意味は以下のとおりです。

- `evm-event-name` はツールによって作成される EVM イベントの NAME です。特定のチャンネルを通じて渡されるすべてのイベントは、イベントをチャンネルに対応づけることができるように、名前のコンポーネントのうち、最初の数個が同じでなければなりません。
- `date` および `time` は、イベントの `TIMESTAMP` 項目を構成します。日付のコンポーネントは基本フォーマット `year/month/day` でなければなりません。ここでスラッシュ (/) 文字は次の文字のいずれかで置き換えることができます。

ハイフン (-)、コロン (:)、ピリオド (.)

ログファイルのフォーマットにバリエーションを許し、シェル・スクリプトで実行する変換の作業量を最小限にするために、値のフォーマットにはある程度の柔軟性もあります。

- 年は 2 桁あるいは 4 桁にするか、または疑問符 (?) 文字に置き換えることができます。4 桁の数値を指定すると、年は変更されません。2 桁の数値を指定すると、値が 90 以上であれば 1900 が加算されて年になります。値が 90 未満であれ

ば、2000 が加算されます。年の代わりに ? 文字を指定すると、指定されたタイムスタンプの月と日を現在の日付と比較して、省略された年をツールが計算します。タイムスタンプの月と日が現在より後の日付を示していれば、年の値は前年になり、それ以外の場合は今年になります。

- 月は 1 ~ 12 までの 2 桁の数値か、月の名前で指定します。月の名前は、省略形 (Feb など) と省略していない形式 (February) のいずれでも指定できます。月の名前は、英語またはシステムの省略時ロケールの言語で指定します。

時刻は `hours:minutes:seconds` というフォーマットで指定します。

- `host` は `HOST_NAME` 項目であり、`hostname` コマンドで調べることができます。
 - `user` は `USER_NAME` 項目です。ログ内の項目がすべて 1 つのアプリケーション・プログラムまたはサブシステムによって記録される場合は、ログファイルの所有者を `user` フィールドに指定します。
 - `message` はイベントのメッセージ・テキストであり、`FORMAT` データ項目としてイベントに挿入されます。
3. `filter-string` を指定した場合は、`-f` および `-r` オプションを指定した `evmshow` を介してその出力を渡すと、フィルタで要求されている文字列だけに出力が制限されます。
 4. 最後に、取得したイベントに対してイベント・テンプレートに含まれるデータ項目を設定したい場合は、EVM チャンネル・ユーティリティの `/usr/share/evm/channels/bin/merge_template` を介して出力をパイプすることもできます。このプログラムは `stdin` の EVM イベントを読み取り、各イベントに対応するテンプレートを EVM デーモンから取得します。次にテンプレート情報をイベントにマージし、処理結果の展開済みイベントを `stdout` に書き込みます。

チャンネルのログ・ファイルを EVM フォーマットに変換するのが困難な場合、たとえば、各エントリが構造化されていない複数のテキスト行で構成されているため、簡単に解析できない場合は、取得関数を使用しないで、イベントが発信されたときに EVM ロガーでログが記録されるようにすることをお勧めします。この場合、イベントが 2 つの場所に格納されるた

め、記憶域の使用量が多くなりますが、取得時間およびプログラミング工数が大幅に削減されます。

チャンネルに対して独自の取得関数を使用する場合は、EVM ロガーの構成ファイルのフィルタ文字列を変更して、イベントが EVM ログ内で重複しないようにする必要があります。EVM ロガーの構成ファイルを変更する方法については、『システム管理ガイド』を参照してください。

14.8.2 詳細関数

詳細関数は、`evmshow` に `-d` オプションを指定して呼び出したときに実行されます。現在は `evmshow` を実行するユーザの権限で実行されますが、今後のリリースでは変更される可能性があるため、適切なセキュリティ対策が重要になります。

この関数では、次の呼び出し構文を使用します。

```
function-name [arguments]
```

チャンネル構成ファイルで、詳細関数の行に引数が指定されると、これらは実行時に直接関数に渡されます。詳細関数は、`stdin` を介して EVM のイベント・ストリームを受信し、各イベントの内容を説明するテキスト・ストリームを `stdout` に表示する必要があります。出力を作成するときは、さまざまな形式の `evmshow` (特に `-x` と `-D`) が役に立ちますが、`-d` オプションは使用しないように注意してください。再帰的なループが発生する可能性があります。

`stderr` に書き込まれたメッセージは、リダイレクトされない限り、`evmshow` の `stderr` ストリームに出力されます。必要な場合は、この関数によって書き込まれたことを明示的に記述してください。

新しいチャンネルに対する詳細スクリプトを開発するためのモデルとして、`evmlog` チャンネル関数、`/usr/share/evm/channels/evmlog/evm-log_details` が使用できます。詳細関数を指定していないときには、`evmshow` とイベント・ビューアは、詳細表示が要求されたときに、チャンネルに属するイベントのダンプをフォーマットして表示します。

14.8.3 説明関数

説明関数は、`evmshow` に `-x` オプションを指定して呼び出したときに実行されます。現在は `evmshow` を実行するユーザの権限で実行されますが、今後

のリリースでは変更される可能性があるため、適切なセキュリティ対策が重要になります。この関数では、次の呼び出し構文を使用します。

```
function-name event-name [reference]
```

説明関数は、説明が必要なイベントの名前およびオプションの参照値を指定して呼び出します。参照値を指定した場合は、イベントの参照データ項目の内容が参照されます。参照値が利用できない場合は、`evmshow` はこの引数としてハイフン (-) を渡しますが、この引数を省略することもできます。

説明関数は、引数を使用して、指定されたイベント名に対応するイベントの説明をフォーマットし、テキスト行として `stdout` に書き込みます。説明が見つからない場合は、代わりに適切なメッセージを `stdout` に書き込みます。`stderr` に書き込まれたメッセージは、リダイレクトされない限り、`evmshow` の `stderr` ストリームに出力されます。このため、必要な場合は、この関数によって書き込まれたことを明示的に記述してください。

説明関数では、次の基準を満たす場合にだけ、`evmlog` 説明関数の `/usr/share/evm/channels/evmlog/evmlog_explain` を呼び出すことができます。

- `event-name` 引数の他に、`cat:catalog-name[:set_number]` という形式の参照引数を渡す場合。このとき、`catalog-name` は、チャンネルのイベントの説明が格納されている I18N カタログ・ファイルの名前であり、オプションの `set_number` は、説明が含まれるメッセージ・セットの番号です (たとえば `cat:myprod.cat:3` のようになります)。
- カタログ内の各説明が、中カッコで囲まれたイベント名で開始している場合。たとえば、`{myco.myprod.myapp.startup}`。

メッセージ・カタログは、通常の I18N 規則に従って配置されていなければなりません。検索時間を最小限に抑えるには、説明をいくつかのセットにグループ化し、イベントの参照データ項目にセット番号を指定します。カタログ・ファイルを生成する手順については、`mkcatdefs(1)` および `gencat(1)` を参照してください。

14.8.4 監視関数

監視関数は、EVM チャンネル・マネージャによって実行されます。チャンネル・マネージャの起動時、およびチャンネル・マネージャが `evmreload` によって再構成されたときは、引数 `-init` が必要ですが、その後は引数

-init を指定しないで定期的に実行されます。実行期間は、チャンネル値 `mon_period` で制御されます。この関数は、常に root として実行されるので、適切なセキュリティ対策が必要です。

この関数では、次の呼び出し構文を使用します。

function-name [init]

引数 `init` が指定されているかどうかによって、この関数が管理している作業ファイルの初期化が必要かどうかが決まります。必要な場合には、チャンネル構成ファイルの該当するコマンド行に引数を設定することにより、追加の引数を渡すことができます。ただし、引数 `init` は常に最後の引数として渡されることに注意してください。

監視関数によって実行されるアクションについて制限はありませんが、通常、このジョブでは、状態をチェックして、状態の変化を検出した場合はイベントを発信します。

この関数は、`stdout` を指定せず、チャンネル・マネージャのログ・ファイルに `stderr` を指定して呼び出します。関数が `stderr` の再割り当てを行わない場合、ここに書き込まれるメッセージはすべて、チャンネル・マネージャのログ・エントリと同じフォーマットになるので、そのチャンネルから送られたこと明確に分かるようにする必要があります。または、エラー・メッセージを EVM イベントとして送信し、関数が起動されるたびに同じ状態を不必要に通知することのないようにします。

次の監視スクリプトの例では、ログ・ファイルの行数をカウントし、そのカウントを状態ファイルに保存することにより、初期化を行っています。後続の呼び出しでは、ファイルの行数を古い行数と比較し、新しい行を UNIX の `tail` コマンドで抽出してから、`evmpost` を使用して EVM イベントとして発信します。

```
#!/bin/sh
INIT=$1
STATE=/tmp/mylog.state
LOG=/tmp/mylog

EVENT_NAME=myco.admin.mylog.line_added

# No log? Create one!
if [ ! -f $LOG ]
then
    touch $LOG
fi

# If we're initializing then save the current logfile
# state and exit:
```

```

if [ "$INIT" != "" ]
then
    # Count the lines in the demolog, and save the count
    # in the state file:
    wc -l $LOG | awk '{print $1}' > $STATE
    exit
fi

# Find out how many lines there were in the file last time
# we checked:
OLDCOUNT=`cat $STATE`

# How many now?
NEWCOUNT=`wc -l $LOG | awk '{print $1}'`
if [ $NEWCOUNT > $OLDCOUNT ]
then
    # Save the new line count to the state file:
    echo $NEWCOUNT > $STATE

    # What's the difference between the old and new counts?
    diff=`expr $NEWCOUNT - $OLDCOUNT | awk '{print $1}'`

    # Post an event for each new line:
    tail -$diff $LOG | while read LINE
    do
        echo 'event { name '${EVENT_NAME}' \
            ' var {name msg type STRING value "'$LINE'" } }' | evmpost
    done
fi

```

14.8.5 クリーンアップ関数

クリーンアップ関数は、EVM チャンネル・マネージャによって毎日実行されます。チャンネル構成ファイルに指定されている時間に、チャンネルのログ・ファイルの保管や削除などのハウスキープینگ処理が行われます。この関数は常に root として実行されるので、適切なセキュリティ対策が必要です。

クリーンアップ関数は、コマンド行として指定し、そのまま実行されます。このため、必要に応じて、コマンド行から引数を渡すことができます。この関数では、任意の必要なアクションを実行できます。この関数は、stdout を指定せず、チャンネル・マネージャのログ・ファイルに stderr を指定して実行します。このため、状態メッセージが必要な場合は、stderr には書き込まずに、通常は evmpost を使用して EVM イベントの形式で発信します。関数が stderr の再割り当てを行わない場合、ここに書き込まれるメッセージはすべて、チャンネル・マネージャのログ・エントリと同じフォーマットになるので、そのチャンネルから送られたことが明確に分かるようにする必要があります。stdout には何も書き込まないでください。

この関数は、実行する時間にかかわらず同じ結果が得られるように作成します。たとえば、find コマンドの -mtime オプションを使用して、保管するログ・ファイルを特定することがあります。

14.8.6 チャンネルのセキュリティ

ほとんどの場合，チャンネル関数は EVM デーモンの子プロセスによって実行されるため，完全な root 権限で実行されることになります。このため，システムの一貫性を保持するために，次の点に従ってください。

- 制限された書き込み権限のあるディレクトリに配置する。
- 関数自体にも，制限された書き込みおよび実行権限を割り当てる。
- この関数から，適切な権限が割り当てられていないほかのプログラムを呼び出さない。

Tru64 UNIX システムにおける 32 ビット・ポインタの使用

Tru64 UNIX システムでは、省略時のポインタ型のサイズは 64 ビットで、すべてのシステム・インタフェースは 64 ビット・ポインタを使用します。Compaq C コンパイラは、64 ビット・ポインタをサポートするだけでなく、32 ビット・ポインタの使用もサポートします。

ほとんどの場合、64 ビット・ポインタがプログラム内で問題を起こすことはなく、32 ビット・ポインタの問題はまったく無視しても構いません。ただし、次の場合は 32 ビット・ポインタの問題について考慮する必要があります。

- `int` 型へのポインタ代入を含むプログラムを移植する場合。
- プログラムに 64 ビットのポインタ・サイズが必要でなく、ポインタがメモリを使いすぎる原因となっている場合。たとえば、プログラムでポインタ・フィールドから構成される大きな構造体を使用している場合に、それらのフィールドが 32 ビットのアドレス範囲を超える必要がない場合などです。
- 32 ビット・システムと 64 ビット・システムが混在する環境に対するプログラムを開発しており、プログラムのメモリ内構造体が両方のシステムからアクセスされる場合。

アプリケーションで 32 ビット・ポインタを使用する場合には、特殊なオプションを使用したアプリケーションのコンパイルおよびリンクが必要であり、コード固有の性質により、ソース・コードの変更も必要となる場合があります。

この付録では、次の種類のポインタについて説明します。

- `short` ポインタ: 32 ビット・ポインタ。`short` ポインタを宣言すると 32 ビットが割り当てられます。

- long ポインタ: 64 ビット・ポインタ。long ポインタを宣言すると 64 ビットが割り当てられます。Tru64 UNIX システムでは、このポインタが省略時のポインタ型です。
- 単純ポインタ: 非ポインタ・データ・タイプへのポインタ。たとえば、`int *num_val` のようになります。厳密に言うと、ポイントされている型にはポインタが含まれていないため、ポイントされている型のサイズは、ポインタの型のサイズに依存しません。
- 複合ポインタ: ポインタのサイズによってサイズが決まるデータ型へのポインタ。たとえば、`char **FontList` のようになります。

A.1 コンパイラ・システムと 32 ビット・ポインタの言語サポート

次のメカニズムにより、32 ビット・ポインタの使用が制御されます。

- cc のオプション `-xtaso` および `-xtaso_short` は、ポインタを 32 ビット・データ型として使用するプログラムをコンパイルする場合に必要です。
 - `-xtaso`

`#pragma pointer_size` 指示文が認識されるようにして、リンクの場合には `-taso` (切り捨てアドレス・サポート・オプション) がリンクに渡されるようにします。
 - `-xtaso_short`

`-xtaso_short` では、short ポインタを使用するように初期コンパイラ状態を設定しますが、それ以外は `-xtaso` と同じです。すべてのシステム・ルーチンは、64 ビット・ポインタを使用し続けるため、この方法でコンパイルする場合、ほとんどのアプリケーションではソースの変更が必要になります。ただし、`protect_headers_setup` (A.3.3 項を参照) を使用すると、ソース・コードを変更する必要性を大幅に減少させることができます。

通常、short ポインタの使用には対象となるアプリケーションに関する十分な知識が必要であるため、移植のための支援ツールとして `-xtaso_short` を使用することは推奨できません。特に、コーディングのよくないプログラム (ポインタと `int` 値が数多く混在するプログラム) を移植するときには使用しないでください。

- `ld` のオプション `-taso` は、実行可能ファイルと関連シェアド・ライブラリが下位 31 ビットでアドレス可能な仮想アドレス空間に配置されるようにします。 `-taso` オプションは、アドレス値が 32 ビットの変数に格納できることを想定したプログラム (ポインタが `int` 型変数と同じ長さであることを想定したプログラム) を移植する場合に役に立ちます。 `-taso` オプションはポインタのデータ型のサイズには影響しませんが、プログラム内のポインタの値が、32 ビット表現でも 64 ビット表現でも同一であることを保証します。

リンカ・オプション `-taso` は、実行時環境とライブラリの使用方法を制限します。 `-taso` オプションについての詳細は、A.2 節を参照してください。

- `#pragma pointer_size` 指示文は、C 言語プログラム内のポインタ型のサイズを制御します。これらのプリグマは、`cc` コマンド行で `-xtaso` あるいは `-xtaso_short` オプションが指定されている場合にのみ、コンパイラによって認識されます。これらのオプションがいずれも指定されていない場合、プリグマは無視されます。

`#pragma pointer_size` 指示文の詳細については、3.12 節を参照してください。

次は、`short` ポインタと `long` ポインタの両方を使用するための例です。

```
main ()
{
    int *a_ptr;
    printf ("A pointer is %ld bytes\n", sizeof (a_ptr));
}
```

省略時の設定または `-xtaso` オプションでコンパイルした場合、サンプル・プログラムは次のようなメッセージを出力します。

```
A pointer is 8 bytes
```

`-xtaso_short` オプションでコンパイルした場合、サンプル・プログラムは次のようなメッセージを出力します。

```
A pointer is 4 bytes
```

A.2 `-taso` オプションの使用

`-taso` オプションは、プログラム内のすべての 64 ビット・ポインタ内で 32 ビットのアドレッシングが行われるようにします。これにより、32 ビットの

アドレッシングに関する問題はほとんど解決されます。ただし、一部のポインタの物理サイズを 32 ビットに制限することを要求することは除きます (これは `-xtaso` または `-xtaso_short` オプション、および `pointer_size` プラグマによって処理されます)。

`-taso` オプションは、プログラム内の `int` 型へのポインタ代入によって発生するアドレッシングの問題を処理するために頻繁に使用されます。多くの C 言語プログラム、特に現在認められているプログラミング手法に準拠していない旧式のプログラムでは、`int` 型変数にポインタを代入しています。このような代入は推奨できませんが、ポインタと `int` 型変数が同一サイズのシステムの場合は正しい結果が得られます。一方、Tru64 UNIX システムの場合、64 ビットのポインタを 32 ビットの `int` 型変数に代入するときにアドレスの上位 32 ビットが失われてしまうため、誤った結果が発生する可能性があります。次のコード例で、この問題を示します。

```
{
char *x;    /* 64-bit pointer */
int z;      /* 32-bit int variable */
.
.
.
    x = malloc(1024); /* get memory and store address in 64 bits */
    z = x;           /* assign low-order 32 bits of 64-bit pointer to
                      32-bit int variable */
}
```

`int` 型へのポインタ代入による問題を処理する際に最も移植性の高い方法は、コードを修正してこのような代入を取り除くことです。ただし、大規模なアプリケーションの場合は、時間のかかる作業になる可能性があります。ソース・コード内の `int` 型へのポインタ代入を検索する場合は、`lint -Q` コマンドを使用します。

この問題の解決に、`-taso` オプションを使用する方法もあります。`-taso` オプションを使用すると、`int` 型へのポインタ代入を修正する必要がなくなります。`-taso` オプションは、プログラムが実行を開始するときに、プログラム内のすべての位置が 64 ビット・アドレスの下位 31 ビットで表現できるようにプログラムのアドレス空間を配置します。これには、シェアード・ライブラリからのルーチンおよびデータのアドレスも含まれます。

`-taso` オプションは、システムがサポートするどのデータ型のサイズにもまったく影響しません。データ型への影響は、ポインタ内のアドレスを 31 ビットに制限するという点だけです。つまり、ポインタのサイズは 64 ビットのままでありますが、アドレスが使用するのは下位 31 ビットだけとなります。

A.2.1 -taso オプションの使用と効果

-taso オプションは、オブジェクト・モジュールの作成に使用する cc または ld のコマンド行に指定します。cc コマンド行に指定した場合、-taso オプションは ld リンカに渡されます。-taso オプションは、リンカがオブジェクト・モジュールにフラグを設定するようにします。このフラグが設定されていると、ローダはモジュールを 31 ビットのアドレス空間にロードします。

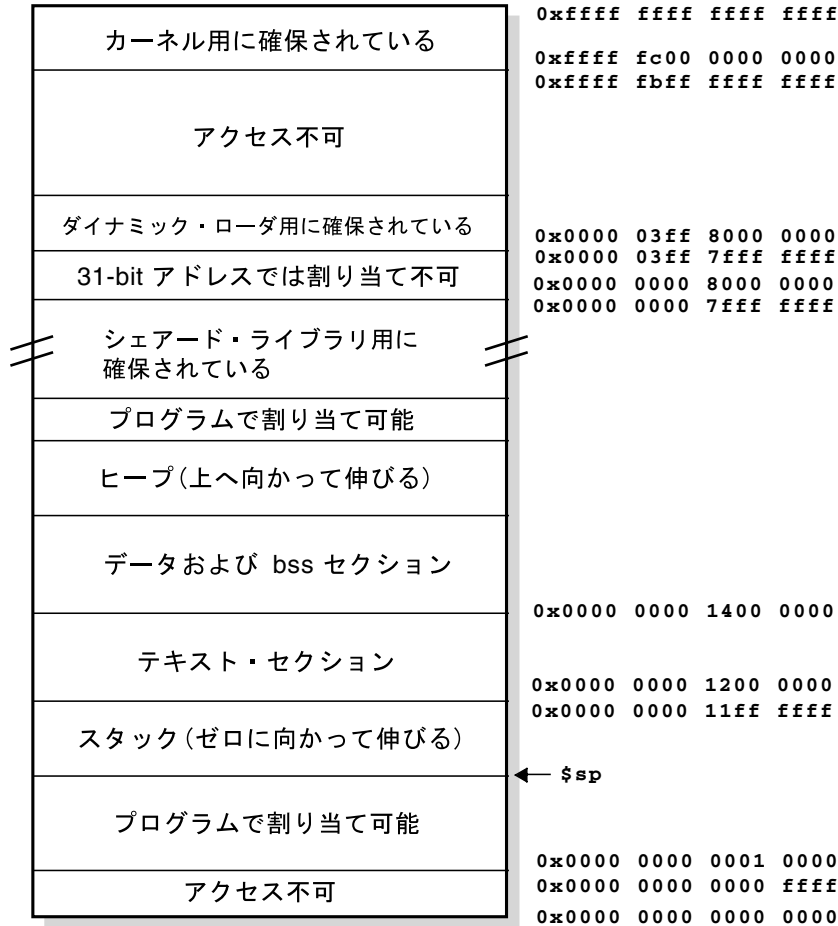
31 ビットのアドレス制限は、int 型へのポインタ代入を行うときに 32 ビットの int 型変数の符号ビット (ビット 31) を設定する可能性を回避するために使用されます。int 型へのポインタ代入によって int 型変数の符号ビットが設定できるようにした場合、符号拡張により問題が発生する可能性があります。次にその例を示します。

```
{
char *x;    /* 64-bit pointer */
int z;      /* 32-bit int variable */
.
.
.
        /* address of named_obj = 0x0000 0000 8000 0000 */
x = &named_obj; /* 0x0000 0000 8000 0000 = original pointer
                value */
z = x; /* 0x8000 0000 = value created by pointer-to-int
        assignment */
x = z; /* 0xffff ffff 8000 0000 = value created by pointer-
        to-int-to-pointer or pointer-to-int-to-long
        assignment (32 high-order bits set to ones by
        sign extension) */
}
```

-taso オプションは、アプリケーションのテキスト・セグメントとデータ・セグメントが 31 ビットのアドレスでアクセスできるメモリにロードされるようにします。したがって、int 型変数にポインタが代入される場合、64 ビットのポインタと 32 ビットの変数の値は常に等しくなります (A.2.2 項で説明する特殊な場合を除きます)。

図 A-1 は、-taso および -call_shared オプションを使用するプログラムのメモリの図です。cc コマンドを実行してリンカを起動した場合の省略時の設定は -call_shared で、ld を直接起動した場合の省略時の設定は -non_shared です。

図 A-1: **-taso** オプションを使用した場合のメモリのレイアウト



ZK-0876U-AIJ

スタックとヒープのアドレスも 31 ビット内に入ることに注意してください。スタックはテキスト・セグメントの最下部から下に向かって伸びて、ヒープはデータ・セグメントの最上部から上に向かって伸びます。

アプリケーションのテキスト・セグメントとデータ・セグメントが下位メモリにロードされるようにするには、**-T** および **-D** オプション (それぞれテキスト・セグメントとデータ・セグメントのアドレスを設定するリンク・オプション) を使用することもできます。一方、**-taso** オプションはテキスト・セグメントとデータ・セグメントの省略時のアドレス設定に加え、31 ビットのアドレス空間外にリンクされているシェアード・ライブラリもロードによって適切に再配置されるようにします。

テキスト・セグメントとデータ・セグメントに使用される省略時のアドレス設定は、cc コマンド行で指定するオプションによって決まります。

- -taso オプションとともに -non_shared または -call_shared オプションを指定すると、省略時のアドレス設定は次のようになります。

0x0000 0000 1200 0000 (テキスト・セグメントの開始アドレス)

0x0000 0000 1400 0000 (データ・セグメントの開始アドレス)

- -taso オプションとともに -shared オプションを指定すると、省略時のアドレス設定は次のようになります。

0x0000 0000 7000 0000 (テキスト・セグメントの開始アドレス)

0x0000 0000 8000 0000 (データ・セグメントの終了アドレス)

ほとんどのアプリケーションの場合、これらの省略時の設定値を使用することによってテキスト・セグメントとデータ・セグメントに十分な空間が作成されます (テキスト・セグメントとデータ・セグメントの内容の詳細については、『*Object File/Symbol Table Format Specification*』を参照してください)。この省略時の設定値により、アプリケーションでは大量の mmap 空間を割り当てることができます。

-taso オプションを指定し、同時に -T と -D を使用してテキスト・セグメントとデータ・セグメントのアドレス値を指定した場合、-taso の省略時のアドレス設定は指定した値に置き換えられます。

31 ビットのアドレス空間内でプログラムの作成に成功したかどうかを確認する場合は、odump ユーティリティを使用することができます。テキスト、データ、および bss セグメントの開始アドレスを表示する場合は、次のコマンドを入力します。

```
% odump -ov obj_file_x.o
```

どのアドレスの 31 ~ 63 ビットは設定されず、0 ~ 30 ビットだけが設定されていなければなりません。

-taso を使用して作成した共用オブジェクトは、-taso を使用しないで作成した共用オブジェクトとリンクすることはできません。taso 共用オブジェクトを nontaso 共用オブジェクトとリンクしようとする、次のエラー・メッセージが表示されます。

```
Cannot mix 32 and 64 bit shared objects without -taso option
```

A.2.2 -taso オプションの効果に対する制限事項

-taso オプションは、プログラムが 31 ビットの制限を超えたアドレスをマップすることを妨げず、これが行われても警告メッセージを通知しません。このようなアドレスは、次のいずれかのメカニズムを使用して作成することができます。

- **-T および -D オプション**

前述したように、-T オプションと -D オプションを -taso オプションとともに使用した場合、-taso オプションの省略時の設定値が指定された値に置き換えられます。したがって、-taso オプションの目的を無効にしないために、-T オプションと -D オプションに指定するアドレスを 31 ビットのアドレス範囲内から選択する必要があります。

- **malloc() 関数**

taso アプリケーションで malloc を使用する場合のアドレッシングの問題を回避するには、libc の `__taso_mode` 変数 (グローバルな読み取り専用 “unsigned long” 変数) を置き換える変数をアプリケーション内で宣言して、静的に値 1 で初期化します。この値は、malloc ルーチンに対して taso アプリケーションで動作していることを通知し、malloc は割り当てられたメモリが 31 ビットの範囲内にあることをチェックして保証します。

- **mmap システム・コール**

mmap システム・コールを使用する taso アプリケーションでは、mmap にジャケッ・ルーチンを使用し、アドレスのマップによって 31 ビットの範囲を超えることがないようにする必要があります。この場合、次の手順に従う必要があります。

1. mmap によって 31 ビットのアドレス範囲を超える空間が割り当てられないようにするには、すべてのモジュール (または最低でも mmap を参照するすべてのモジュール) に対して、cc コマンド行で次のコンパイル・オプションを指定します。

```
-Dmmap=_mmap_32_
```

このオプションでは、mmap 関数の名前がジャケッ・ルーチンの名前 (`_mmap_32_`) に等しいことを定義しています。その結果、ソース・プログラム内で mmap 関数が参照されると、このジャケッ・ルーチンが起動されます。

2. `mmap` 関数が 1 つのソース・モジュールだけで呼び出される場合は、ジャケッ・ルーチンとそのモジュールに取り込むか、`mmap_32.o` オブジェクト・モジュールを作成して `cc` コマンド行に指定します。ジャケッ・ルーチンのファイル指定は `/usr/opt/alt/usr/lib/support/mmap_32.c` であり、ファイルはオプションのソフトウェア・サブセット `CMPDEVENHnnn` にあります。

`mmap` 関数が複数のソース・ファイルから呼び出される場合は、`mmap_32.o` オブジェクト・モジュールを作成して `cc` コマンド行に指定するメソッドを使用する必要があります。これは、ジャケッ・ルーチンを複数のモジュールに入れた場合、リンカ・エラーが発生するためです。

A.2.3 `taso` 環境での `malloc` の動作

`call-shared` の `taso` アプリケーション (図 A-1 を参照) に対するメモリ領域をローダがセットアップする方法に起因して、特別の手順を実行しなければ、`malloc()` 関数で 256 M バイトを超えるメモリを割り当ててはできません。

この問題を解決するために、割り当てのための領域を増やす必要があります。これは、次の方法で行います。

- `-non_shared` でのリンク

`taso` アプリケーションで大きいメモリを割り当てするための、最も効果的な方法は、シェアード・ライブラリなしでアプリケーションをリンクする方法です。`-non_shared` でリンクすることができれば、`malloc()` は `bss` の末尾から `0x7fff ffff` までのメモリを割り当てることができます。このとき、プログラムが `mmap` を用いてその範囲内のメモリを割り当てないということを前提とします。`mmap` による割り当てで生じる競合の問題については、このリストの次の項で説明します。

- `malloc` 関数の `__sbrk_override` チューニング変数の使用

`malloc` ルーチン (`malloc`, `calloc`, `realloc`, `valloc`) は通常、`sbrk` システム・コールによってメモリを割り当てます。ただし、`sbrk` がシェアード・ライブラリのテキスト・セクションに到達するか、または `mmap` によって割り当てられた領域に到達すると、`sbrk` による割り当ては失敗します。

`__sbrk_override` チューニング変数を使用すると、割り当ては、メモリの `0x0001 0000` から `0x7fff ffff` までの間で行われます。この変数を設定すると、`malloc` は `sbrk` の呼び出しの失敗を検出したときに、`mmap` の使用へと切り替わり、`mmap` は前に示したアドレス範囲の中に十分な領域を見つけると、割り当て要求を満たします。

A.3 `-xtaso` または `-xtaso_short` オプションの使用

`-xtaso` または `-xtaso_short` オプションを使用すると、プログラム内で `short` (32 ビット) と `long` (64 ビット) の両方のポインタ型を使用することができます。ただし、`-xtaso` と `-xtaso_short` のどちらにも `-taso` オプションが関係するため、64 ビットのデータ型が 31 ビットのアドレッシングに制限されることに注意してください。

`-xtaso` または `-xtaso_short` オプションは、プログラム内で `short` ポインタが必要になる場合にだけ使用します。32 ビットのアドレッシングを使用し、`short` ポインタが必要でない場合は、`-taso` オプションを使用します。

Tru64 UNIX は 64 ビット・システムであり、オペレーティング・システムあるいはシステム提供のライブラリでポインタ・データを交換する場合には 64 ビット・ポインタを使用しなければならないため、`short` ポインタを使用するほとんどのプログラムでは、`long` ポインタも使用する必要があります。通常のアプリケーションは、標準のシステム・データ型を使用するので、ポインタの変換は必要ありません。`short` ポインタを使用するアプリケーションでは、`pointer_size` プラグマを使用して `short` ポインタを `long` ポインタに明示的に変換しなければならない場合があります (3.12 節を参照)。

注意

新規アプリケーションで `short` ポインタの使用を検討している場合は、最初に `long` ポインタで開発しておき、後に `short` ポインタの方が有用かどうかを分析して判断してください。

A.3.1 ポインタ・サイズの変更に関するコーディング上の注意事項

ソース・コードでのポインタの扱いに関連するコーディング上の注意事項を次に示します。

- 定義の一部としてポインタを含む `typedef` で使用されるポインタのサイズは、`typedef` を使用するときではなく宣言するときに決定されます。このため、`typedef` 定義の一部として `short` ポインタが宣言されている場合、その `typedef` を使用して宣言されている変数については、`long` ポインタが宣言されたコンテキストでコンパイルされたとしても、すべて `short` ポインタが使用されます。
- マクロ内のポインタのサイズは、マクロが展開されるコンテキストによって決定されます。`typedef` を使用して必要なポインタ・サイズを宣言する以外に、マクロの一部としてポインタ・サイズを指定する方法はありません。
- 一般的に、`short` および `long` 単純ポインタ間での変換は安全であり、代入でのキャストあるいは関数呼び出しを行う必要はなく、暗黙に行われます。一方、複合ポインタでは通常、`short` および `long` のポインタ表現間での変換を正しく行うためにソース・コードの変換が必要です。
たとえば、引数ベクトル `argv` は複合 `long` ポインタであり、そのように宣言する必要があります。多くの X11 ライブラリ関数は、複合 `long` ポインタを返します。これらの関数の戻り値を正しく宣言しないと、誤動作が発生します。関数が `short` ポインタだけを使用するようにコンパイルされており、そのようなベクトルにアクセスする必要がある場合には、それを関数に渡す前に、`long` ポインタ・ベクトルから `short` ポインタ・ベクトルに値をコピーするコードを追加する必要があります。
- `short` ポインタの使用をサポートしているのは C および C++ コンパイラだけです。C および C++ ルーチンから他の言語で作成されたルーチンへ `short` ポインタを渡してはなりません。
- `pointer_size` プラグマおよび `-xtaso_short` オプションは、`main()` の 2 番目の引数 (通常 `argv` と呼ばれる) のサイズには影響を与えません。この引数は、`pointer_size` プラグマが他のポインタ・サイズを 4 バイトに設定するために使用されている場合であっても、常に 8 バイトのサイズとなります。

A.3.2 32 ビット・ポインタの使用に関する制限事項

Tru64 UNIX システム上のほとんどのアプリケーションは、32 ビットで表現できないアドレスを使用しています。したがって、通常のアプリケーションによって呼び出される可能性のあるライブラリに、`short` ポインタを含めるこ

とはできません。ソフトウェア・ライブラリを作成するベンダは、short ポインタを使用してはなりません。

A.3.3 システム・ヘッダ・ファイルに関する問題の回避

システム・ライブラリを作成する場合、コンパイラは、ポインタが 64 ビットであり、構造体メンバの自然境界合わせが行われていることを想定しています。これは、C および C++ コンパイラの省略時の設定です。システム・ライブラリに対するインタフェース (/usr/include ツリーにあるヘッダ・ファイル) では、この想定を明示的にコード化していません。

ポインタ・サイズ (-xtaso_short) および構造体メンバの境界合わせ (-zpn [この場合 nl=8] または -nomember_alignment) に関するコンパイラの想定を変更することができます。これらのオプションのいずれかを使用し、アプリケーションで /usr/include ツリーからヘッダ・ファイルをインクルードして、ライブラリ関数を呼び出したり、そのヘッダ・ファイルで宣言されている型を使用している場合には、問題が発生する可能性があります。特に、システム・ヘッダ・ファイル宣言を処理する場合に、コンパイラで計算されたデータ・レイアウトは、コンパイルされてシステム・ライブラリに格納されたレイアウトと異なる可能性があります。このような状態では、予測できない結果が発生する可能性があります。

この項で説明して状況下での、ポインタ・サイズおよびデータの境界合わせで問題が発生する可能性を取り除くため、システムにコンパイラをインストールした直後にスクリプト `protect_headers_setup.sh` を実行してください。これを実行する理由と方法についての詳細は、`protect_headers_setup(8)` を参照してください。

コンパイル・コマンド行で `-protect_headers` オプションのバリエーションを使用することにより、`protect_headers_setup` スクリプトで設定したプロテクションを有効にしたり、無効にすることができます。`-protect_headers` オプションの詳細については、`cc(1)` を参照してください。

B

System V 実行環境における相違点

付録 B では、System V 実行環境の C 言語プログラムに対するソース・コード互換性を得る方法について説明します。また、オペレーティング・システムの省略時のシステム・コールおよびライブラリ関数と、System V 実行環境におけるシステム・コールおよびライブラリ関数との違いについても要約します。

B.1 ソース・コードの互換性

C 言語プログラムに対するソース・コード互換性を得るには、使用するシェルの `PATH` 環境変数を変更してアプリケーションのコンパイルおよびリンクを行います。

`PATH` 環境変数を変更する場合、System V 実行環境へのアクセスは次の 2 つのレベルで行なわれます。

- 第 1 のレベルでは、変更した `PATH` 環境変数によって、オペレーティング・システムの省略時のバージョンのコマンドの代わりに System V バージョンの多くのコマンドが実行されます。
- 第 2 のレベルでは、System V の `cc` コマンドあるいは `ld` コマンドが実行されます。

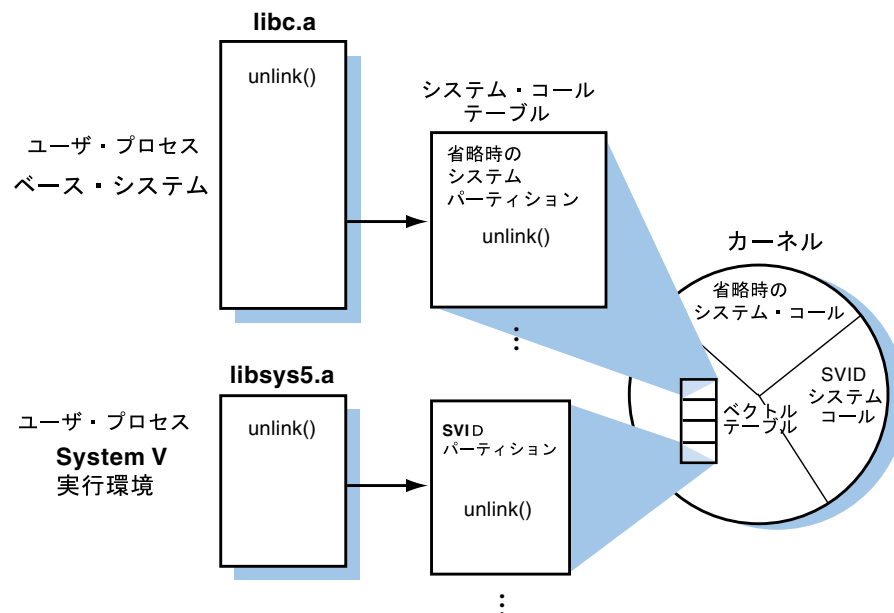
System V バージョンの `cc` コマンドおよび `ld` コマンドを実行すると、システム・コールおよびサブルーチンに対するソース・コード参照は、System V 実行環境のライブラリによって解決されます。System V 実行環境でそのサブルーチンあるいはシステム・コールが見つからない場合、ソース・コード参照は標準のライブラリおよびコマンド行で指定したその他のライブラリで解決されます。また、標準のヘッダ・ファイルの代わりに System V バージョンのヘッダ・ファイル (`/usr/include` ファイルなど) を使用するように、インクルード・ファイルの探索パスが変更されます。

システム・コールを起動するライブラリ関数は、システム・コール・テーブルを使用してシステムのプリミティブをカーネルに配置します。基本オペレー

ティング・システムには、System V 用のものを含め、いくつかのシステム・テーブルが用意されています。System V 動作を示すシステム・コールは、システム・コール・テーブルの System V の部分にエントリされています。

PATH 環境変数を System V 実行環境に設定してプログラムをリンクすると、システム・コールの参照を解決するために libsys5 が探索されます。図 B-1 に示すように、libsys5 によって起動される unlink() システム・コールは、システム・コール・テーブルの System V 部分のエントリをポイントします。このマッピングは、システムの省略時の unlink() システム・コールのマッピングとは別のカーネル領域に対して行われます。

図 B-1: システム・コールの解決



ZK-0814U-AIJ

System V 実行環境の cc および ld コマンドは (指定されている場合)、コマンド実行前に、システムの省略時の cc および ld コマンドにいくつかのオプションを追加するシェル・スクリプトです。

cc コマンドは、SVID バージョンのヘッダ・ファイルを使用するために -Ipath オプションをコマンド行に自動的に挿入します。たとえば、省略時のヘッダ・ファイルの代わりに /usr/include ファイルが使用されます。SVID バージョンと違いのないシステム・ヘッダ・ファイルについては、省略時のファイルが使用されます。

B-2 System V 実行環境における相違点

cc および ld コマンドは、自動的に次のオプションを使用します。

- `-Lpath`

System V ライブラリのパスを提供します。

- `-lsys5`

システム・コールおよびサブルーチンの参照を解決する際に、標準の C ライブラリの前に `libsys5.a` ライブラリを探索するよう指定します。

- `-D__SVID__`

省略時のシステムから SVID 固有の動作を選択的に有効にします。

省略時の設定では、cc コマンドはシェアード・ライブラリを使用してプログラムを動的にリンクします。System V 実行環境は、この機能をサポートするために、`libsys5.a` に加えて `libsys5.so` をサポートしています。

System V バージョンの cc および ld コマンドは、省略時のバージョンの cc および ld コマンドにユーザが指定したコマンド行オプションを渡します。このため、ユーザはライブラリの階層構造を作成することができます。たとえば、`PATH` 環境変数に System V 実行環境が設定されていて、算術ライブラリ関数および `/local/lib` ディレクトリの `libloc.a` 関数への参照がプログラムに含まれている場合、次のようにプログラムをコンパイルします。

```
% cc -non_shared -L/local/lib src.c -lm -lloc
```

System V cc コマンドは、このコマンド行を読み取り、省略時のライブラリより優先して System V 実行環境ライブラリを探索するのに必要なオプションを追加します。また、このコマンドは `/usr/include` の標準のヘッダ・ファイルの代わりに、System V のヘッダ・ファイルをインクルードします。ご使用の環境に SVID 2 が設定されている場合は、前のコマンドは次のように処理されます。

```
/bin/cc -D__SVID__ -I$SVID2PATH/usr/include -L$SVID2PATH/usr/lib \
-non_shared -L/local/lib src.c -lm -lloc -lsys5
```

このコマンドを使用すると、次の順番でライブラリが探索されます。

1. `/usr/lib/libm.a`
2. `/local/lib/libloc.a`
3. `SVID2PATH/usr/lib/libsys5.a`
4. `/usr/lib/libc.a`

探索されるライブラリおよびその順序は実行する関数によって異なります。
詳細については、cc(1) および ld(1) を参照してください。

B.2 システム・コールとライブラリ・ルーチンの要約

表 B-1 に、System V 実行環境におけるシステム・コールの動作について示します。表 B-2 には、System V 実行環境におけるライブラリ関数の動作について示します。

これらのシステム・コールおよびライブラリ関数の詳細については、それぞれのリファレンス・ページを参照してください。

表 B-1: システム・コールの要約

システム・コール	System V 動作
longjmp(2) および setjmp(2)	スタックのみをリストア/保管する。
mknod(2)	ディレクトリ，通常ファイル，特殊ファイル の作成機能を提供する。
mount(2sv) および umount(2sv)	省略時のバージョンとは異なる引数を取り， <sys/types.h> ヘッダ・ファイルをインクルードする 必要がある。
<div>注意</div> <div>System V バージョンの mount リファレンス・ページにアクセスするためには，man コマンドでセクション番号 2sv を指定する 必要があります。</div>	
open(2)	省略時の設定として O_NOCTTY フラグを設定しないように 指定する。この設定によって，ある条件を満たせば， 端末装置に対するオープン呼び出しで，そのデバイスが そのプロセスの制御端末となることが可能になる。
pipe(2)	STREAMS ベースのファイル記述子に対するパイプ 操作をサポートする。
sigaction(2) およ び signal(2)	カーネルがシグナル・ハンドラに追加情報を渡すことを 指定する。この情報には，siginfo 構造体にシグナル を発行した理由，ucontext 構造体にシグナルを発行し たときの呼び出しプロセスのコンテキストを含みます。

表 B-1: システム・コールの要約 (続き)

システム・コール	System V 動作
sigpause(2)	呼び出しプロセスのシグナル・マスクから指定したシグナルのブロックを解除し、シグナルを受信するまで呼び出しプロセスを停止する。SIGKILL および SIGSTOP のシグナルはリセットできない。
sigset(2)	SIGCHLD のディスポジションを SIG_IGN に設定した場合、呼び出しプロセスの子プロセスは、プロセス終了時にゾンビに変わることができないことを指定する。子プロセスの終了を親プロセスが待つ場合、すべての子プロセスが終了するまで親プロセスはブロックされる。この操作は、値 -1 を返し errno を [ECHILD] に設定する。
unlink(2)	スーパーユーザを含め、ユーザは空でないディレクトリのリンクを解除することができない。errno を ENOTEMPTY に設定する。ディレクトリが空の場合、スーパーユーザはリンクを解除できる。

表 B-2: ライブラリ関数の要約

ライブラリ関数	System V 動作
getcwd(3)	現在のディレクトリを取得する。 <code>char *getcwd (char *buffer, int size);</code>
mkfifo(3)	STREAMS ベースの FIFO の作成をサポートし、 <code>/dev/streams/pipe</code> を使用する。
mktemp(3)	getpid 関数を使用して、一意な名前の <code>pid</code> の部分を取得する。
ttyname(3)	端末が擬似端末装置の場合、 <code>/dev/pts/</code> で始まるパス名で文字列のポインタを返す。



C

動的に構成可能なカーネル・サブシステムの作成

新しいカーネル・サブシステムの作成，あるいは既存のカーネル・サブシステムの変更を行う場合，それらを動的に構成可能なように設定することができます。付録 C では，以下の情報を提供することにより，動的に構成可能なカーネル・サブシステムの作成方法について説明します。

- 動的に構成可能なサブシステムについての概要 (C.1 節)
- 属性テーブルについての概要と属性テーブルの例 (C.2 節)
- 構成ルーチンの作成方法と構成ルーチンの例 (C.3 節)
- オペレーティング・システムとサブシステムとで互換性があるかどうかを確認するためのオペレーティング・システムのバージョンのチェック方法 (C.4 節)
- テストのために，ロード可能なサブシステムをカーネルに組み込む方法 (C.5 節)
- テストのために，実行時に属性の変更を可能にするスタティック・サブシステムをカーネルに組み込む方法 (C.6 節)
- 動的に構成可能なサブシステムのデバッグに関する情報 (C.7 節)

Tru64 UNIX オペレーティング・システムが動的に構成可能なサブシステムをサポートする以前は，それらのシステムの構成ファイルをシステム管理者が編集することによりカーネル・サブシステムを管理していました。この方法では，サブシステムの追加/変更，あるいはサブシステム・パラメータの変更の際に，場合によっては複雑で時間のかかる，カーネルの再構築が必要でした。複数のシステムを管理しているシステム管理者は，それらのシステムの各構成ファイルに対して変更を加え，それぞれのカーネルを再構築する必要がありました。

動的に構成可能なサブシステムのサポートによって，システム管理者は，ファイルの編集やカーネルの再構築をせずにシステム・パラメータの変更や

サブシステムのロード/アンロードを行うことができます。システム管理者は、`sysconfig` コマンドを使用することによりカーネルのサブシステムを構成することができます。このコマンドを使用することにより、システム管理者は、サブシステムのロードおよび構成、アンロードおよび構成除外、再構成 (変更)、および照会をローカル・システムあるいはリモート・システムで行なうことができます。

ロード可能なデバイス・ドライバを作成する場合は、デバイス・ドライバ固有の問題について考慮する必要があります。ロード可能なデバイス・ドライバの作成方法については、『*Writing Device Drivers*』を参照してください。

C.1 動的に構成可能なサブシステムの概要

多くの Tru64 UNIX カーネル・サブシステムはスタティック・サブシステムであるため、構築時にカーネルとリンクされています。カーネルを構築した後は、これらのサブシステムをロードあるいはアンロードすることはできません。スタティック・サブシステムの例としては、`vm` (virtual memory: 仮想メモリ) サブシステムが挙げられます。このサブシステムはシステムが正しく動作するために必要なものです。

カーネル・サブシステムによってはロード可能なものもあります。ロード可能なサブシステムは、カーネルを再構築せずにカーネルに対してサブシステムを追加/削除することができます。ロード可能なサブシステムの例としては、`presto` サブシステムが挙げられます。`presto` サブシステムは、Prestoserve ソフトウェアを使用している場合のみロードされます。

スタティック・サブシステムおよびロード可能なサブシステムのどちらも、動的に構成することができます。

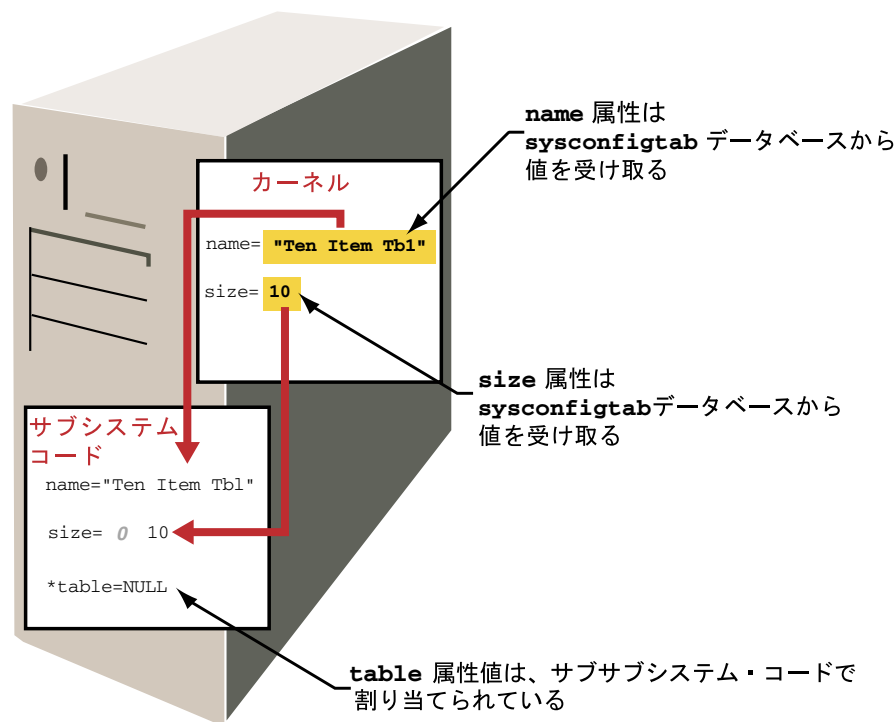
- スタティック・サブシステムの場合、動的に構成可能であるということは、カーネルを再構築しなくても選択したサブシステムの属性が変更できることを意味します。このタイプのサブシステムは、属性値に関する照会に応答することが可能で、使用中でなければ構成除外することもできます (ただし、アンロードすることはできません)。
- ロード可能なサブシステムの場合、動的に構成可能であるということは、ロード時にサブシステムがカーネルに構成され、カーネルを再構築せずにサブシステムの変更が可能で、また、アンロードする前にサブシステムの構成が除外されることを意味します。このタイプのサブシステムは、属性に関する照会に応答することができます。

C-2 動的に構成可能なカーネル・サブシステムの作成

従来のカーネル・サブシステムと同じように、動的に構成可能なサブシステムはパラメータを持っています。これらのパラメータは属性と呼ばれます。サブシステムの属性の例としては、タイムアウト値、テーブル・サイズ、メモリ位置、サブシステム名などがあります。サブシステムの属性は属性テーブルに定義します。属性テーブルについては C.2 節を参照してください。

ロード可能サブシステムを最初に構成する前に、システム管理者は属性に対する値を `sysconfigtab` データベースに保管することができます。このデータベースは `/etc/sysconfigtab` ファイルに保管され、ブート時にカーネル・メモリにロードされます。このデータベースに保管された値は、サブシステムがその属性の初期値を供給しているかどうかにかかわらず、サブシステムの属性の初期値になります。図 C-1 は、どのようにして `sysconfigtab` データベースから初期属性値が得られるかを示しています。

図 C-1: システム属性値の初期化
カーネル・メモリ空間



ZK-0973U-AIJ

図 C-1 で、サブシステムが `size` 属性の値を 0 (ゼロ) に初期化しているにもかかわらず、`size` 属性の初期値として `sysconfigtab` データベースの値が割り当てられていることに注意してください。

サブシステム・コードで宣言されている属性テーブルを使用するには、最初の構成時にどちらのサブシステム属性値を設定できるかを制御します。`sysconfigtab` データベースに設定できる属性を制御する方法については、C.2 節を参照してください。

最初の構成時のための属性値を保管できることに加えて、システム管理者は、サブシステムがカーネルに構成されている場合はいつでもその属性値の照会や再構成が可能です。照会要求に対しては、システム管理者に属性値が返されます。再構成要求の場合は、属性値は変更されます。これらの処理は、属性がサブシステム・コードでどのように宣言されているかに依存します。

- サブシステムの属性テーブルが属性のアドレスをカーネルに供給している場合、カーネルがその属性値の変更あるいは照会に対する応答を行うことができます。カーネルにアドレスを供給し、カーネルが属性値を処理できるようにすることは、属性値を保守する最も効果的な方法です。
- カーネルがその属性値へアクセスできない場合は、サブシステムが属性値の変更あるいは照会への応答を行う必要があります。カーネルが属性値を処理できるようにするのが属性値を保守する最も効果的な方法ですが、サブシステムが属性値を保守せざるを得ない場合もあります。たとえば、カーネルは属性値を計算することはできませんので、計算が必要な値の保守はサブシステムが行なう必要があります。

サブシステムのどの属性値に対する照会あるいは再構成を可能にするかについては、C.2 節に示す方法で制御します。

動的に構成可能な各サブシステムは、属性テーブルに加えて構成ルーチンも使用します。このルーチンは、サブシステムで保守する属性値の計算などのタスクを行います。また、たとえばテーブルの大きさの決定、あるいはサブシステムが使用するローカル変数のメモリ位置の保管などのサブシステム固有のタスクも実行します。C.3 節で、構成ルーチンの作成方法について説明します。カーネルは、サブシステムが構成、照会、再構成、あるいは構成除外されるたびにサブシステムの構成ファイルを呼び出します。

カーネルに構成できるサブシステムは、カーネル構成から除外することもできます。システム管理者がカーネル構成からサブシステムを除外すると、

そのサブシステムに占有されていたカーネル・メモリは、サブシステムがロード可能な場合には解放されます。構成除外要求の際にカーネルがサブシステム構成ルーチンを呼び出すことにより、サブシステムはサブシステム固有の構成除外タスクを実行することができます。サブシステム固有の構成除外タスクの例としては、サブシステム・コードによって割り当てられたメモリの解放があります。

C.2 属性テーブルの概要

動的に構成可能な効果的なサブシステムを作成するポイントは、効果的な属性テーブルを定義することです。属性テーブルはサブシステムの属性を定義します。属性の例としては、タイムアウト値、テーブル・サイズ、メモリ位置などがあります。属性テーブルは、定義属性テーブルと通信属性テーブルの2つのフォーマットで存在します。

- 定義属性テーブルはサブシステム・コードに含まれています。この属性テーブルは、そのサブシステムの属性を定義します。各属性の定義は、属性テーブル構造体の1つの要素です。この定義には、属性名、属性のデータ型、システム管理者がその属性に対して行うことができる要求リストが含まれます。各属性の定義には、最小値、最大値、また、オプションでそれらの保管領域も含まれます。カーネルは、構成、再構成、照会、構成除外などのシステム管理者からの要求に応答するために属性の定義を使用します。
- 通信属性テーブルはカーネルとサブシステム・コードの通信に使用されます。この属性テーブル構造体の各要素は、1つの属性の情報を媒介します。この情報には、次のものが含まれます。
 - 属性の名前とデータ型
 - その属性に対して行われた要求
 - 要求の状態
 - 属性の値

この属性テーブルは、システム管理者が構成、再構成、照会、あるいは構成除外などの要求を行ったときにカーネルからサブシステムへ渡されます。

2つのタイプの属性テーブルを使用するのは、カーネル・メモリを節約するためです。定義属性テーブルの情報のいくつか(たとえば、属性名、属性

のデータ型など)は通信属性テーブルの情報と同じですが、情報の多くはそれぞれのテーブルで異なります。たとえば、属性定義時には要求は発生しないので、定義属性テーブルには要求の状態は保管しません。同様に、通信属性テーブルには、各属性に対する要求のサポート・リストは含まれません。カーネル・メモリを節約するために、各属性テーブルには必要な情報のみが含まれます。

注意

サブシステム属性テーブルに定義されている属性名は、文字列 `method` で始まっていてはなりません。この文字列は、ロード可能デバイス・ドライバ・メソッドで使用されるネーミング属性のために予約されています。デバイス・ドライバ・メソッドについての詳細は、『*Writing Device Drivers*』を参照してください。

以降の各項で、`/sys/include/sys/sysconfig.h` における宣言を示しながら、両方のタイプの属性テーブルについて説明します。

C.2.1 定義属性テーブル

定義属性テーブルには、データ型 `cfg_subsys_attr_t` が含まれています。これは、`/sys/include/sys/sysconfig.h` ファイルで次のように宣言されている属性の構造体です。

```
typedef struct cfg_attr {
    char        name[CFG_ATTR_NAME_SZ]; 1
    uint        type; 2
    uint        operation; 3
    whatever    address; 4
    uint        min; 5
    uint        max;
    uint        binlength; 6
}cfg_subsys_attr_t;
```

- 1** name フィールドに属性の名前が保管されます。2 文字から `CFG_ATTR_NAME_SZ` 定数に保管されている値まで長さの英文字を選択します。 `CFG_ATTR_NAME_SZ` 定数は、`/sys/include/sys/sysconfig.h` ファイルで定義されます。

- ② このフィールドに、表 C-1 に示すいずれか 1 つの属性データ型を指定します。

表 C-1: 属性データ型

データ型名	説明
CFG_ATTR_STRTYPE	ヌルで終わる文字の配列 (char*)
CFG_ATTR_INTTYPE	32 ビットの符号付きの数値 (int)
CFG_ATTR_UINTTYPE	32 ビットの符号なしの数値 (unsigned)
CFG_ATTR_LONGTYPE	64 ビットの符号付きの数値 (long)
CFG_ATTR_ULONGTYPE	64 ビットの符号なしの数値
CFG_ATTR Bintype	バイトの配列

- ③ operation フィールドには、その属性に対して実行できる要求を指定します。表 C-2 に示す要求コードから 1 つあるいは複数を指定します。
- 単一の属性の構成除外はできないため、CFG_OP_UNCONFIGURE 要求コードは、個々の属性に対しては意味を持ちません。
- したがって、operation フィールドに CFG_OP_UNCONFIGURE は指定できません。

表 C-2: 属性に対して許可される要求を指定するコード

要求コード	意味
CFG_OP_CONFIGURE	サブシステムが最初に構成されるときに属性の値を設定できる。
CFG_OP_QUERY	サブシステムが構成されていれば、いつでも属性の値を表示できる。
CFG_OP_RECONFIGURE	サブシステムが構成されていれば、いつでも属性の値を変更できる。

- ④ address フィールドは、カーネルが属性の値にアクセスするかどうかを決定します。

このフィールドにアドレスを指定すると、カーネルは属性値の読み取りおよび変更を行うことができます。sysconfig コマンドからカーネルが照会要求を受信すると、カーネルはこのフィールドに指定されている

位置から値を読み取り、その値を返します。構成要求あるいは再構成要求に対して、カーネルは次の条件をチェックします。

- 新しい値のデータ型がその属性に対して適切であるか。
- その値が属性の最小値と最大値の範囲内であるか。

その値がこれらの条件に当てはまる場合、カーネルは新しい値を属性値として保管します (最大値と最小値は属性定義の次の 2 つのフィールドで指定します)。

サブシステム・コードの属性に対する照会、構成、再構成などの要求に対してユーザが応答することもできます。この場合、このフィールドに NULL を指定します。属性値の制御方法についての詳細は、C.3 節を参照してください。

- ⑤ min および max フィールドは、属性値の最小値と最大値を定義します。

カーネルは、これらの 2 つのフィールドを属性のデータ型に従って別々に解釈します。属性のデータ型が整数型である場合、これらのフィールドには最小値および最大値を示す整数値が含まれます。属性のデータ型が `CFG_ATTR_STRTYPE` である場合、これらのフィールドには文字列の最小長および最大長が含まれます。属性のデータ型が `CFG_ATTR_BINTYPE` である場合は、これらのフィールドには、ユーザが修正することのできる最小バイト数および最大バイト数が含まれます。

- ⑥ カーネルがバイナリ属性の内容を読み取ったり変更したりできるように設定するには、`binlength` フィールドを使用してそのバイナリ・データの現在の値を指定します。その属性の保管されているバイナリ・データの長さをカーネルが変更すると、カーネルはこのフィールドの内容も修正します。

属性のデータ型が整数あるいは文字列であるか、バイナリ属性に対する照会要求および再構成要求に対して構成ルーチンで応答したい場合には、このフィールドは使用しません。

C.2.2 定義属性テーブルの例

例 C-1 に示すのは、定義属性テーブルの例です。この属性テーブルは、架空のカーネル・サブシステム `table_mgr` のものです。この架空のサブシステムの構成ルーチンについては、C.3 節で説明しています。

例 C-1: 定義属性テーブルの例

```
#include <sys/sysconfig.h>
#include <sys/errno.h>

/*
 *   Initialize attributes
 */
static char          name[] = "Default Table";
static int           size = 0;
static long          *table = NULL;

/*
 *   Declare attributes in an attribute table
 */

cfg_subsys_attr_t table_mgr_attributes[] = {
/*
 *   "name" is the name of the table
 */
{"name", 1          CFG_ATTR_STRTYPE, 2
 CFG_OP_CONFIGURE | CFG_OP_QUERY | CFG_OP_RECONFIGURE, 3
 (caddr_t) name, 42, sizeof(name), 506},
/*
 *   "size" indicates how large the table should be
 */
{"size",          CFG_ATTR_INTTYPE,
 CFG_OP_CONFIGURE | CFG_OP_QUERY | CFG_OP_RECONFIGURE,
 NULL, 1, 10, 0},
/*
 *   "table" is a binary representation of the table
 */
{"table",          CFG_ATTR_BINTYPE,
 CFG_OP_QUERY,
 NULL, 0, 0, 0},
/*
 *   "element" is a cell in the table array
 */
{"element",          CFG_ATTR_LONGTYPE,
 CFG_OP_QUERY | CFG_OP_RECONFIGURE,
 NULL, 0, 99, 0},
{0,0,0,0,0,0} /* required last element */
};
```

このテーブルの最後のエントリ `{,0,0,0,0,0,0}` は空の属性です。この属性は属性テーブルの終わりを示すもので、すべての属性テーブルで必要なものです。

この属性テーブルの最初の行は、テーブルの名前を定義しています。この属性テーブルの名前は `table_mgr_attributes` です。name 属性のフィールドについて以下に説明します。

- ❶ 属性名は name フィールドに保管されます。name は、属性テーブルの最初にあるデータ宣言によって "Default Table" に初期化されています。
- ❷ 属性データ型は `CFG_ATTR_STRTYPE` です。これはヌルで終了する文字列配列です。
- ❸ このフィールドは、対象となる属性に対して実行できる操作を指定します。この例の場合、この属性に対して構成、照会、再構成が可能です。
- ❹ このフィールドは、カーネルがユーザの属性値にアクセスできるかどうかを決定します。

この例のように、このフィールドにアドレスを指定した場合、カーネルはユーザの属性値の読み取りおよび変更を行うことが可能です。カーネルが `sysconfig` コマンドから照会要求を受け取った場合、このフィールドに指定した場所の値を読み取り、その値を返します。構成要求および再構成要求に対しては、カーネルは、新しい値のデータ型がその属性に対して適当かどうかを確認しその値が属性の最小値と最大値の範囲内に収まるかどうかをチェックします。この値がこれらの要件に該当する場合、カーネルはその属性に対する新しい値を保管します (最小値および最大値は、属性定義の次の 2 つのフィールドで指定します)。

- ❺ これらの 2 つのフィールドは、属性に対する最小値 (この例の場合 2) および最大値 (この例の場合 `sizeof(name)`) を定義します。

属性の最小値および最大値として、システムで定義されている値を使用したい場合は、`/usr/include/limits.h` ファイルで定義されている定数の 1 つを使用することができます。

- ❻ バイナリ属性の内容をカーネルが読み取り/変更できるように設定するには、このフィールドを使用してバイナリ・データの現在のサイズを指定します。この属性に保管されているバイナリ・データの長さをカーネルが変更すると、このフィールドの内容も変更されます。

属性が整数または文字列であるか、バイナリ属性に対する照会あるいは再構成要求に構成ルーチンで応答する場合には、このフィールドは使用されません。

C.2.3 通信属性テーブル

/sys/include/sys/sysconfig.h ファイルで定義される通信属性テーブルは、cfg_attr_t データ型を持ちます。例 C-2 で示すように、このデータ型は属性の構造体になっています。

例 C-2: 通信属性テーブル

```
typedef struct cfg_attr {
    char      name[CFG_ATTR_NAME_SZ]; 1
    uint      type; 2
    uint      status; 3
    uint      operation; 4
    long      index; 5
    union { 6
        struct {
            caddr_t val;
            ulong   min_len;
            ulong   max_len;
            void     (*disposal)();
        } str;
        struct {
            caddr_t val;
            ulong   min_size;
            ulong   max_size;
            void     (*disposal)();
            ulong   val_size;
        } bin;
        struct {
            caddr_t val;
            ulong   min_len;
            ulong   max_len;
        } num;
    } attr;
}cfg_attr_t;
```

- 1** name フィールドには属性の名前を指定します。属性名の命名規則は、定義属性テーブルの name フィールドの命名規則と同じです。

- ② type フィールドには属性データ型を指定します。表 C-1 に指定できるデータ型を示します。
- ③ status フィールドには、表 C-3 に示す定義済み状態コードを指定します。

表 C-3: 属性状態コード

状態コード	意味
CFG_ATTR_EEXISTS	属性が存在しない。
CFG_ATTR_EINDEX	属性インデックスが正しくない。
CFG_ATTR_ELARGE	属性値あるいはサイズが大きすぎる。
CFG_ATTR_EMEM	その属性に対して有効なメモリがない。
CFG_ATTR_EOP	その属性は、要求された操作をサポートしていない。
CFG_ATTR_ESMALL	属性値あるいはサイズが小さすぎる。
CFG_ATTR_ESUBSYS	カーネルによるサブシステムの構成/再構成およびサブシステムに対する照会への応答が許可されない。サブシステム・コードで応答を行なう必要がある。
CFG_ATTR_ETYPE	属性タイプが正しくない。あるいは、属性タイプが一致しない。
CFG_ATTR_SUCCESS	操作が正しく行なわれた。

- ④ operation フィールドには、表 C-2 に示すコードを指定します。
- ⑤ index フィールドには、構造化された属性に対するインデックスを指定します。
- ⑥ 共用体 attr には、属性値とその最小値および最大値が含まれます。

CFG_ATTR_STRTYPE データ型の属性の場合、val 変数には文字列データが含まれます。最小値および最大値は、文字列長の最小値および最大値となります。disposal ルーチンは、アプリケーションによるカーネル・メモリの使用が終わったら、ユーザがカーネル・メモリに自由に書き込むことができるルーチンです。

CFG_ATTR_BINTYPE データ型の属性の場合、val フィールドにはバイナリ値が含まれます。最小値および最大値は、ユーザが変更できるバイト数の最小値および最大値となります。disposal ルーチンは、アプリケーションによるカーネル・メモリの使用が終わったら、ユーザがカーネ

ル・メモリに自由に書き込むことができるルーチンです。val_size 変数にはバイナリ・データの現在のサイズが含まれます。

数値データ型の場合、val 変数には整数値が含まれ、最小値および最大値も整数値となります。

C.2.4 通信属性テーブルの例

この項では、通信属性テーブルの例を示します。この属性テーブルは、架空のカーネル・サブシステム table_mgr のものです。この架空のサブシステムの構成ルーチンについては、C.3 節で説明しています。

```
table_mgr_configure(  
  cfg_op_t      op,          /*Operation code*/[1]  
  caddr_t       indata,      /*Data passed to the subsystem*/[2]  
  ulong         indata_size, /*Size of indata*/  
  caddr_t       outdata,     /*Data returned to kernel*/[3]  
  ulong         outdata_size) /*Count of return data items*/  
{
```

次の示すのは、table_mgr_configure 通信属性テーブルのフィールドの説明です。

[1] op 変数には、次のいずれかが含まれます。

```
CFG_OP_CONFIGURE  
CFG_OP_QUERY  
CFG_OP_RECONFIGURE  
CFG_OP_UNCONFIGURE
```

[2] indata 構造体は、構成ルーチンに対して indata_size のデータを提供します。オペレーション・コードが CFG_OP_CONFIGURE あるいは CFG_OP_QUERY の場合、このデータは、構成あるいは照会される属性名のリストです。オペレーション・コードが CFG_OP_RECONFIGURE の場合、データは属性名と値で構成されます。オペレーション・コードが CFG_OP_UNCONFIGURE の場合には、構成ルーチンに値は渡されません。

[3] outdata 構造体および outdata_size 変数は、構成サブシステムの将来の拡張機能のためのプレースホルダです。

C.3 構成ルーチンの作成

サブシステムを構成可能にするためには、構成ルーチンを定義する必要があります。このルーチンは定義属性テーブルとともに機能し、サブシステ

ムの構成/再構成，サブシステムの照会に対する応答，サブシステムの構成除外を行ないます。

サブシステムでの必要性に依存して，構成ルーチンは単純であったり複雑であったりします。構成ルーチンの目的は，カーネルが実行できないタスクを実行することです。定義属性テーブルによって属性のロケーションをカーネルに知らせることができるので，カーネルは属性に関するすべての構成要求，再構成要求，照会要求を処理することができます。ただし，これらの要求が発生している間，カーネルにおける処理量は制限されます。たとえば，カーネルによる属性値の計算などは行えないので，属性値を計算する必要があるような属性については構成ファイルで処理する必要があります。

以降の各項で，構成ルーチンの例を示します。ここで示すサンプル・ルーチンは，バイナリ・データを保守するための架空のサブシステム `table_mgr` のための構成ルーチンです。この構成ルーチンは，次のタスクを行います。

- 最初の構成時に，テーブルに対してカーネル・メモリを割り当てます。
- テーブル属性に関する照会を処理します。
- システム管理者が要求した際に，テーブルのサイズを変更します。
- 構成除外の際にカーネル・メモリを解放します。
- カーネルに制御を戻します。

このサブシステムのソース・コードは，`/usr/examples/cfgmgr` ディレクトリに含まれています。このサブシステムの定義属性テーブルについては C.2.2 項を，通信属性テーブルについては C.2.4 項を参照してください。

C.3.1 初期構成の実行

最初の構成時に，`table_mgr` サブシステムは，それが保守するテーブルを作成します。例 C-1 に示すように，システム管理者は，初期構成時にテーブルの名前とサイズを設定することができます。これらの値を設定するために，システム管理者は，`sysconfigtab` データベースに希望する値を保管しておきます。

サブシステムのコードで定義されているこのテーブルの省略時の名前は，`Default Table` です。また，このテーブルの省略時のサイズはゼロです。

例 C-3 に示すのは，`table_mgr` サブシステムの初期構成時に実行されるコードです。

例 C-3: 初期構成の実行

```
...
switch(op){[1]
case CFG_OP_CONFIGURE:

    attributes = (cfg_attr_t*)indata;[2]

    for (i=0; i<indata_size; i++){[3]
        if (attributes[i].status == CFG_ATTR_ESUBSYS) {[4]

            if (!strcmp("size", attributes[i].name)){[5]
                /* Set the size of the table */
                table = (long *) kalloc(attributes[i].attr.num.val*sizeof(long));[6]

                /*
                 * Make sure that memory is available
                 */

                if (table == NULL) {[7]
                    attributes[i].status = CFG_ATTR_EMEM;
                    continue;
                }

                /*
                 * Success, so update the new table size and attribute status
                 */

                size = attributes[i].attr.num.val; [8]
                attributes[i].status = CFG_ATTR_SUCCESS;
                continue;
            }
        }
    }
    break;
...

```

- [1] 構成ルーチンには switch 文が含まれます。この switch 文によって、サブシステムは多くの操作に対して応答することができます。サブシステムは op 変数の値に従って、異なるタスクを実行します。
- [2] この文は、attributes ポインタを初期化します。これにより、indata 構造体で渡されたデータを構成ルーチンが操作できるようになります。
- [3] for ループは、構成ルーチンに渡される各属性の状態を確認します。
- [4] 属性の状態フィールドに CFG_ATTR_ESUBSYS が含まれている場合、構成ルーチンはその属性を構成しなければなりません。

- ⑤ 最初の構成時に操作する必要のある唯一の属性は `size` です。 `size` 属性が現在の属性である場合のみ、 `if` 文内のコードが実行されます。
- ⑥ 状態フィールドに `CFG_ATTR_ESUBSYS` が含まれ、属性名フィールドに `size` が含まれる場合、ローカル変数 `table` はカーネル・メモリ領域のアドレスを受け取ります。カーネル・メモリ領域は、 `attributes[i].attr.num.val` で指定されているテーブルのサイズを保管するのに十分な大きさでなければなりません。
`attributes[i].attr.num.val` に指定される値は、テーブル内のロングワード数を指定する整数です。カーネルは `sysconfigtab` データベースから整数値を読み取り、 `attr` 共用体で構成ルーチンに渡します。
- ⑦ カーネル・メモリを割り当てることができない場合、 `kalloc` ルーチンは `NULL` を返します。テーブルに対してメモリが割り当てられていない場合、構成ルーチンは、メモリを使用できないことを示す `CFG_ATTR_EMEM` を返します。この場合、カーネルはエラー・メッセージを表示します。サブシステムはカーネルに構成されますが、システム管理者は、 `sysconfig` コマンドを使用してテーブルのサイズを再設定する必要があります。
- ⑧ カーネル・メモリが正しく割り当てられると、 `sysconfigtab` ファイルのテーブル・サイズが静的な外部変数 `size` に保管されます。これによりサブシステムは、テーブルのサイズを必要とする操作に対してこの値を使用することができるようになります。

C.3.2 照会要求に対する応答

`table_mgr` サブシステムのユーザは、照会要求によって次の項目を表示することができます。

- テーブルの名前
- テーブルのサイズ
- テーブルそのもの
- テーブルの要素の 1 つ

例 C-1 に示すように、 `name` 属性の宣言には、カーネルがそのテーブルの名前に直接アクセスできるようにアドレス (`caddr_t`) `name` が含まれています。このため、構成ルーチンには、テーブル名に関する照会に応答するためのコードは不要です。

例 C-4 に示すのは、照会要求の一部として実行されるコードの例です。

例 C-4: 照会要求に対する応答

```
switch (op):
:
:

    case CFG_OP_QUERY:
/*
 * indata is a list of attributes to be queried, and
 * indata_size is the count of attributes
 */
attributes = (cfg_attr_t *) indata; [1]

for (i = 0; i < indata_size; i++) { [2]
    if (attributes[i].status == CFG_ATTR_ESUBSYS) { [3]

/*
 * We need to handle the query for the following
 * attributes.
 */

if (!strcmp(attributes[i].name, "size")) { [4]

    /*
     * Fetch the size of the table.
     */
    attributes[i].attr.num.val = (long) size;
    attributes[i].status = CFG_ATTR_SUCCESS;
    continue;
}

if (!strcmp(attributes[i].name, "table")) { [5]

    /*
     * Fetch the address of the table, along with its size.
     */
    attributes[i].attr.bin.val = (caddr_t) table;
    attributes[i].attr.bin.val_size = size * sizeof(long);
    attributes[i].status = CFG_ATTR_SUCCESS;
    continue;
}

if (!strcmp(attributes[i].name, "element")) { [6]

    /*
     * Make sure that the index is in the right range.
     */
    if (attributes[i].index < 1 || attributes[i].index > size) {
        attributes[i].status = CFG_ATTR_EINDEX;
        continue;
    }

    /*
     * Fetch the element.
     */
    attributes[i].attr.num.val = table[attributes[i].index - 1];
    attributes[i].status = CFG_ATTR_SUCCESS;
    continue;
}
}
}
```

例 C-4: 照会要求に対する応答 (続き)

```
break;  
:  
:
```

- ① この文は、`attributes` ポインタを初期化します。これにより、`indata` 構造体で渡されたデータを構成ルーチンが操作できるようになります。
- ② `for` ループは、構成ルーチンに渡される各属性の状態を確認します。
- ③ 属性の状態フィールドに `CFG_ATTR_ESUBSYS` が含まれている場合、構成ルーチンはその属性に関する照会に応答しなければなりません。
- ④ 現在の属性が `size` の場合、このルーチンは `size` 変数の値を `attr` 共用体の `val` フィールド (`attributes[i].attr.num.val`) にコピーします。 `size` 変数の値は整数なので、共用体の `num` の部分が使用されます。

次にこのルーチンは、状態フィールド `attributes[i].status` に `CFG_ATTR_SUCCESS` を保管します。

- ⑤ 現在の属性が `table` の場合、このルーチンはテーブルのアドレスを `attr` 共用体の `val` フィールドに保管します。この属性はバイナリなので、共用体の `bin` の部分が使用され、テーブルのサイズは `val_size` フィールドに保管されます。テーブルのサイズは、現在のテーブル・サイズ `size` と、このサイズのロングワードを乗算することにより計算されます。

`status` フィールドは `CFG_ATTR_SUCCESS` に設定されて、処理が成功したことを示します。

- ⑥ 現在の属性が `element` の場合、このルーチンはテーブルの要素の値を `attr` 共用体の `val` フィールドに保管します。各フィールドはロングワードなので、`attr` 共用体の `num` の部分が使用されます。

`sysconfig` コマンド行で指定したインデックスが範囲外の場合、ルーチンは `CFG_ATTR_EINDEX` を状態フィールドに保管します。この場合、カーネルはエラー・メッセージを表示します。システム管理者は別のインデックスで再処理する必要があります。

インデックスが範囲内の場合、status フィールドには
CFG_ATTR_SUCCESS が設定されて、処理が成功したことを示します。

C.3.3 再構成要求に対する応答

再構成要求は、table_mgr サブシステムの属性を変更します。例 C-1 に示す定義属性テーブルでは、次の table_mgr 属性をシステム管理者が再構成することができます。

- テーブルの名前
- テーブルのサイズ
- テーブルの要素の内容

例 C-1 に示すように、name 属性宣言には、カーネルがテーブル名に直接アクセスできるようにアドレス ((caddr_t) name) が含まれます。このため構成ルーチンには、テーブル名の再構成要求に応答するためのコードは不要です。

例 C-5 に示すのは、再構成要求の際に実行されるコードの例です。

例 C-5: 再構成要求に対する応答

```
switch(op){
:
:

case CFG_OP_RECONFIGURE:
/*
 * The indata parameter is a list of attributes to be
 * reconfigured, and indata_size is the count of attributes.
 */
attributes = (cfg_attr_t *) indata; [1]

for (i = 0; i < indata_size; i++) { [2]
    if (attributes[i].status == CFG_ATTR_ESUBSYS) { [3]

/*
 * We need to handle the reconfigure for the following
 * attributes.
 */

    if (!strcmp(attributes[i].name, "size")) { [4]

        long  *new_table;
        int   new_size;

/*
 * Change the size of the table.
 */
        new_size = (int) attributes[i].attr.num.val; [5]
        new_table = (long *) kalloc(new_size * sizeof(long));

/*
 * Make sure that we were able to allocate memory.
```

例 C-5: 再構成要求に対する応答 (続き)

```
        */
        if (new_table == NULL) { ❹
            attributes[i].status = CFG_ATTR_EMEM;
            continue;
        }

        /*
        * Update the new table with the contents of the old one,
        * then free the memory for the old table.
        */
        if (size) { ❺
            bcopy(table, new_table, sizeof(long) *
                ((size < new_size) ? size : new_size));
            kfree(table);
        }

        /*
        * Success, so update the new table address and size.
        */
        table = new_table; ❻
        size = new_size;
        attributes[i].status = CFG_ATTR_SUCCESS;
        continue;
    }

    if (!strcmp(attributes[i].name, "element")) { ❸

        /*
        * Make sure that the index is in the right range.
        */
        if (attributes[i].index < 1 || attributes[i].index > size) ❿
            attributes[i].status = CFG_ATTR_EINDEX;
            continue;
        }

        /*
        * Update the element.
        */
        table[attributes[i].index - 1] = attributes[i].attr.num.val; ⓫
        attributes[i].status = CFG_ATTR_SUCCESS;
        continue;
    }
}

break;

:
:
```

- ❶ この文は、attributes ポインタを初期化します。これにより、indata 構造体で渡されたデータを構成ルーチンが操作できるようになります。

- ② `for` ループは、構成ルーチンに渡される各属性の状態を確認します。
- ③ 属性の状態フィールドに `CFG_ATTR_ESUBSYS` が含まれている場合、構成ルーチンはその属性を再構成しなければなりません。
- ④ 現在の属性が `size` の場合、再構成を行なうとテーブルのサイズが変更されます。サブシステムは、カーネル・メモリを使用することができ、既存のテーブルのデータを損失しないことを保証するため、2つの変数が新たに宣言されます。`new_table` および `new_size` 変数は、新しいテーブルの定義と新しいテーブル・サイズを保管します。
- ⑤ `new_size` 変数は、`attributes[i].attr.num.val` フィールドで渡される新しいサイズを受け取ります。この値は `sysconfig` コマンド行で指定されます。

`new_table` 変数は、新しいテーブル・サイズとして適切なバイト数を含んでいるメモリ領域をポイントするアドレスを受け取ります。新しいテーブル・サイズは、`new_size` 変数の値とバイト数のロングワード (`sizeof(long)`) の乗算で計算されます。
- ⑥ カーネル・メモリを割り当てることができない場合、`kalloc` ルーチンは `NULL` を返します。テーブルに対してメモリが割り当てられていない場合、構成ルーチンは、メモリを使用できないことを示す `CFG_ATTR_EMEM` を返します。この場合、カーネルはエラー・メッセージを表示します。システム管理者は、適切な値を指定して `sysconfig` コマンドを再入力する必要があります。
- ⑦ この `if` 文は、テーブルが存在するかどうかを判断します。テーブルが存在する場合、サブシステムは既存テーブルから新しいテーブルヘデータをコピーし、既存テーブルが占有していたメモリを解放します。
- ⑧ 最後に、カーネル・メモリが割り当てられ、既存テーブルのデータが保管されたことをサブシステムが確認した後、サブシステムは `new_table` に保管されているアドレスを `table` に移します。また、サブシステムは新しいテーブル・サイズを `new_size` から `size` へ移します。

`CFG_ATTR_SUCCESS` が `status` フィールドに設定されて、操作が正しく行われたことを示します。
- ⑨ 現在の属性が `element` の場合、サブシステムは新しいテーブル要素をテーブルに保管します。

- 10 値を保管する前に、指定されたインデックスが有効な範囲内にあるかどうかをルーチンが確認します。インデックスが範囲外の場合、ルーチンは状態フィールドに `CFG_ATTR_EINDEX` を保管します。この場合、カーネルはエラー・メッセージを表示します。システム管理者は、別のインデックスで再操作する必要があります。
- 11 インデックスが範囲内の場合、サブシステムは、`attr` 共用体の `val` フィールドをテーブルの要素に保管します。各要素はロングワードなので、`attr` 共用体の `num` の部分が使用されます。

操作が正しく行われたことを示す `CFG_ATTR_SUCCESS` が `status` フィールドに設定されます。

C.3.4 サブシステムで定義した操作の実行

`table_mgr` サブシステムは、テーブル内のすべてのフィールドの値を 2 倍にするアプリケーション固有の演算を定義します。

サブシステムで独自の演算を定義する場合、`<sys/sysconfig.h>` ファイルで定義しているように、演算コードは `CFG_OP_SUBSYS_MIN` から `CFG_OP_SUBSYS_MAX` までの範囲でなければなりません。カーネルは、この範囲の演算コードを受け取ると、直ちにサブシステム・コードに制御を移します。サブシステム定義の演算に対しては、カーネルは動作しません。

サブシステムに制御が移ると、サブシステムは、要求で渡されたデータの操作を含め、演算を実行します。

次の例は、`CFG_OP_SUBSYS_MIN` 値を持つ要求に対する応答で実行されたコードです。

```
switch (op) {
:

    case CFG_OP_SUBSYS_MIN:

        /*
         * Double each element of the table.
         */
        for (i=0; ((table != NULL) && (i < size)); i++)
            table[i] *= 2;

        break;
:
}
```



```
}
```

このコードは、テーブル内の各要素の値を 2 倍にします。

C.3.5 サブシステムの構成除外

table_mgr サブシステムを構成除外すると、カーネル・メモリが解放されます。次に示すのは、構成除外要求に応答する際に実行されるコード例です。

```
switch(op) {
:

    case CFG_OP_UNCONFIGURE:
/*
 * Free up the table if we allocated one.
 */
    if (size)
        kfree(table, size*sizeof(long));
    size = 0;
    break;
}

return ESUCCESS;
}
```

この構成ルーチンのコードは、テーブルに対してメモリが割り当てられているかどうかを判断します。メモリが割り当てられている場合、ルーチンは kfree 関数を使用してメモリを解放します。

C.3.6 構成ルーチンの終了

次に示すのは、return 文の例です。

```
switch(op) {
:

    size = 0;
    break;
}

return ESUCCESS;
```

構成要求、再構成要求、照会要求、あるいは構成除外要求が完了すると、サブシステム構成ルーチンは ESUCCESS を返します。このサブシステムが設計された方法では、構成、照会、再構成、あるいは構成除外要求の全体が失敗

することはありません。 C.3.1 項および C.3.3 項に示すように、個々の属性についての操作が失敗することはありません。

たとえば、1 つ以上のキー属性の構成に失敗したときにサブシステムの構成全体を失敗させたいときのように、状況によっては、サブシステムの構成、再構成、あるいは構成除外を失敗させたい場合があるかもしれません。次の例は、エラー値でルーチンを終了する例を示しています。

```
switch(op) {
:

    if (table == NULL) {
        attributes[i].status = CFG_ATTR_EMEM;
        return ENOMEM;          /*Return message from errno.h*/
    }
```

この例の if 文は、テーブルに対してメモリが割り当てられているかどうかをテストします。メモリが割り当てられていない場合、サブシステムはエラー状態を返しサブシステムの構成は失敗します。

/sys/include/sys/sysconfig.h および /usr/include/errno.h ファイルで定義されているように、次のメッセージが表示されます。

```
No memory available for the attribute
Not enough core
```

この場合、システム管理者は、再度 sysconfig コマンドを実行してサブシステムの構成を行う必要があります。

0 以外の戻り値は、エラー状態と解釈されます。サブシステムがエラー状態を返した場合は、次のような状況が発生します。

- 最初の構成でエラーが返されると、サブシステムはカーネルに構成されません。
- 照会要求でエラーが発生した場合、データの表示は行われません。
- 構成除外要求でエラーが発生した場合、サブシステムはカーネルから除外されません。

C.4 オペレーティング・システムのリビジョンの確認

ロード可能なサブシステムを作成する場合、オペレーティング・システムのバージョン番号を確認するためのコードをサブシステムに追加すべきです。このコードにより、作成したサブシステムとは互換性のないバージョンのオペレーティング・システムにそのサブシステムがロードされるのを防ぎます。

新たにリリースされたオペレーティング・システムが前のバージョンと比較して大きな違いがある場合、このリリースをメジャー・リリースと呼びます。メジャー・リリースの場合、オペレーティング・システムに対して行われた変更が原因でサブシステムが正しく動作しないことがあるので、オペレーティング・システムのメジャー・リリースごとにテストを行い、必要ならサブシステムを修正する必要があります。メジャー・リリースでオペレーティング・システムに新たに追加された機能を使用したい場合も同様です。

新たにリリースされたオペレーティング・システムが、前のバージョンと比較して大きな違いがない場合、このリリースをマイナー・リリースと呼びます。一般に、メジャー・リリースのオペレーティング・システム上では、サブシステムに何も変更を加えないで実行できるはずです。ただし、新しいオペレーティング・システムでのテストは行うべきです。オペレーティング・システムに新たに追加された機能を使用したい場合も同様です。

オペレーティング・システムのバージョン番号を調べたい場合は、Tru64 UNIX システムが提供するグローバル・カーネル変数 `version_major` および `version_minor` を使用します。次に示すのは、オペレーティング・システムのバージョンを調べるためのコード例です。

```
⋮
extern int version_major;
extern int version_minor;

if (version_major != 5 && version_minor != 0)
    return EVERSION;
```

この例では、バージョン 5.0 のオペレーティング・システムでサブシステムを実行するように指定しています。

C.5 ロード可能なサブシステムの構築とロード

ロード可能なサブシステムを作成した後、テストのため、それを構築して、カーネルに組み込む必要があります。この節では、ロード可能なサブシステムの構築およびロード方法について説明します。実行時の属性構成が可能なスタティック・サブシステムの構築方法については、C.6 節を参照してください。

動的にロード可能なサブシステムを構築するための、次に示すプロシージャでは、ファイル `table_mgr.c` と `table_data.c` に含まれている

table_mgr という名前のサブシステムを構築しているものとします。このサブシステムを構築するには、次の手順に従います。

1. サブシステムのソース・ファイルを /usr/sys 領域のディレクトリに移動します。

```
# mkdir /usr/sys/mysubsys
# cp table_mgr.c /usr/sys/mysubsys/table_mgr.c
# cp table_data.c /usr/sys/mysubsys/table_data.c
```

ディレクトリ名 mysubsys は、好みのディレクトリ名に置き換えることができます。

2. 好みのテキスト・エディタを使用して、/usr/sys/conf/files ファイルを編集し、次の行を挿入します。

```
#
# table_mgr subsystem
#
MODULE/DYNAMIC/table_mgr          optional table_mgr Binary
mysubsys/table_mgr.c              module table_mgr
mysubsys/table_data.c             module table_mgr
```

files ファイルのエントリは、config プログラムに対するサブシステムを記述します。このエントリの最初の行には、次の情報が入っています。

- MODULE/DYNAMIC/table_mgr トークンは、サブシステムが table_mgr という名前の動的カーネル・モジュール (オブジェクト・グループ) であることを示します。
- optional キーワードは、サブシステムをカーネルに組み込む必要がないことを示します。
- table_mgr 識別子は、sysconfig および autosysconfig コマンド行のサブシステムを識別するトークンです。この名前を選択する際には、他のサブシステム名に関して、確実に一意であるように十分に注意してください。サブシステムに対して複数の名前をリストすることができます。
- Binary キーワードは、サブシステムがすでにコンパイルされ、オブジェクト・ファイルがターゲット・カーネル内にリンクできることを示します。

files ファイル・エントリに続く行には、各モジュールを構成するソース・ファイルへのパス名を指定します。

3. 次のコマンドを入力して、makefile と関連するヘッダ・ファイルを生成します。

```
# /usr/sys/conf/sourceconfig BINARY
```

4. /usr/sys/BINARY ディレクトリに変更して、次のようにモジュールを構築します。

```
# cd /usr/sys/BINARY
# make table_mgr.mod
```

5. エラーなしでモジュールが構築できると、それを /subsys ディレクトリに移動して、システムがロードできるようにします。

```
# cp table_mgr.mod /subsys/
```

6. /sbin/sysconfig コマンドまたは /sbin/init.d/autosysconfig コマンドを使用して、サブシステムをロードします。

次に示すコマンド行は、table_mgr サブシステムのロードおよび構成に使用できます。

```
# /sbin/sysconfig -c table_mgr
```

システムをリブートするたびに、そのサブシステムをカーネルに組み込みたい場合は、次のコマンドを入力します。

```
# /sbin/init.d/autosysconfig add table_mgr
```

autosysconfig コマンドは、table_mgr サブシステムを自動的にカーネルに組み込むサブシステムのリストに追加します。

C.6 静的な構成可能サブシステムのカーネルへの構築

実行時に属性構成が可能なスタティック・サブシステムを作成すると、テストのため、それをカーネルに組み込む必要があります。この節では、属性の動的構成をサポートするスタティック・サブシステムの構築方法について説明します。

動的にロード可能なサブシステムを構築するための、次に示すプロシージャでは、ファイル table_mgr.c に格納されている table_mgr という名前のサブシステムを構築しているものとします。

1. サブシステムのソース・ファイルを /usr/sys 領域のディレクトリに移動します。

```
# mkdir /usr/sys/mysubsys
# cp table_mgr.c /usr/sys/mysubsys/table_mgr.c
```

```
# cp table_data.c /usr/sys/mysubsys/table_data.c
```

ディレクトリ名 mysubsys は、好みのディレクトリ名に置き換えることができます。

2. 好みのテキスト・エディタを使用して、/usr/sys/conf/files ファイルを編集し、次の行を挿入します。

```
#
# table_mgr subsystem
#
MODULE/STATIC/table_mgr          optional table_mgr Binary
mysubsys/table_mgr.c             module table_mgr
mysubsys/table_data.c            module table_mgr
```

files ファイルのエントリは、config プログラムに対するサブシステムを記述します。このエントリの最初の行には、次の情報が入っています。

- MODULE/STATIC/table_mgr トークンは、サブシステムが table_mgr という名前の静的カーネル・モジュール (オブジェクト・グループ) であることを示します。
- optional キーワードは、サブシステムをカーネルに組み込む必要がないことを示します。
- table_mgr 識別子は、システム構成ファイルのサブシステムを識別するトークンです。この名前を選択する際には、他のサブシステム名に関して、確実に一意であるように十分に注意してください。サブシステムに対して複数の名前をリストすることができます。
- Binary キーワードは、サブシステムがすでにコンパイルされ、オブジェクト・ファイルがターゲット・カーネル内にリンクできることを示します。

files ファイル・エントリに続く行には、各モジュールを構成するソース・ファイルへのパス名を指定します。

3. /usr/sbin/doconfig プログラムを実行して、カーネルを再構築します。

```
# /usr/sbin/doconfig
```

4. 次のプロンプトに対して、構成ファイルの名前を入力します。

```
*** KERNEL CONFIGURATION AND BUILD PROCEDURE ***
Enter a name for the kernel configuration file. [MYSYS]: MYSYS.TEST
```

カーネル・サブシステムをテストするため、構成ファイルの新しいファイル名を入力します。たとえば、`MYSYS.TEST` と入力します。`doconfig` プログラムで新しい構成ファイル名を指定すると、システム上の既存の構成ファイルは変更されません。このようにしておくと、後で既存の構成ファイルを使用して、テスト中のサブシステムを除いて、システムを構成することができます。

5. Kernel Option Selection メニューからオプション 15 を選択します。オプション 15 は、新しいカーネル・オプションを追加していることを示します。
6. 次のプロンプトに対して応答すると、構成ファイルを編集することを示します。

```
Do you want to edit the configuration file? (y/n) [n] yes
```

すると、`doconfig` プログラムはエディタを起動します (`doconfig` で起動するエディタを制御するには、`EDITOR` 環境変数を定義します)。サブシステムの識別子 (この場合には `table_mgr`) を構成ファイルに追加します。

```
options    TABLE_MGR
```

エディタを終了すると、`doconfig` プログラムが新しい構成ファイルと新しいカーネルを構築します。

7. 新しいカーネルをルート (`/`) ディレクトリにコピーします。

```
# cp /usr/sys/MYSYS_TEST/vmunix /vmunix
```

8. システムをシャットダウンしてリブートします。

```
# shutdown -r now
```

注意

`optional` キーワードを `standard` に置き換えると、カーネルでそのモジュールが必須であることを指定できます。 `standard` キーワードを使用すると、システム構成ファイルを編集する手間を省くことができます。次に示す `files` ファイル・エントリは、必須カーネル・モジュールのためのものであり、システム構成ファイルに含まれているかどうかに関係なく、カーネルに組み込まれます。

```
#
# table_mgr subsystem
#
MODULE/STATIC/table_mgr          standard Binary
mysubsys/table_mgr.c             module table_mgr
mysubsys/table_data.c            module table_mgr
```

前述のようなエントリを files ファイルに記述する場合には、MYSYS: という名前のシステム上で、次のコマンドを入力することにより、サブシステムをカーネルに追加します。

```
# /usr/sbin/doconfig -c MYSYS
```

このコマンドの MYSYS は、使用しているシステム構成ファイル名と置き換えてください。

このコマンドを実行すると、テストしているサブシステム、この場合には table_mgr サブシステムを加えて、既存のシステム構成ファイルに記述されている vmunix カーネルを構築します。

C.7 サブシステムのテスト

サブシステムにおける構成、照会、再構成、構成除外要求のテストは、sysconfig コマンドを使用して行うことができます。サブシステムをテストしている場合は、-v オプションを指定して sysconfig コマンドを入力します。-v オプションを指定すると、sysconfig コマンドは通常より多くの情報を表示します。このコマンドは、cfgmgr 構成管理サーバと kloadsrv カーネル・ローディング・ソフトウェアからの情報を /dev/console スクリーンに表示します。kloadsrv からの情報は、サブシステムのロードが失敗する原因となる未解決シンボルの名前を調べるのに特に便利です。

多くの場合、カーネル・サブシステムのデバッグには、他のカーネル・プログラムをテストするのと同じように dbx、kdebug、および kdbx を使用することができます。dbx -remote コマンドを使用して kdebug デバッガを使用する場合、サブシステムの .mod ファイルは、dbx を実行しているシステムとリモート・テスト・システムで同じ位置になければなりません。サブシステムのソース・コードは、dbx を実行しているシステム上の同じ位置になければなりません。kdebug デバッガを使用するために必要な設定については、『*Kernel Debugging*』を参照してください。

dbx 起動時に動的にロード可能なサブシステムがロードされていない場合は、dbx addobj コマンドを実行することによりデバッガがサブシステムの開始アドレスを決定することができます。サブシステムの開始アドレスへデバッガがアクセスできない場合は、サブシステムのデータを調べサブシステムのコードにブレークポイントを設定するために、dbx addobj コマンドを使用することはできません。次の手順は、dbx デバッガを起動して、table_mgr.mod サブシステムを構成し、addobj コマンドを入力する方法を示しています。

1. dbx デバッガを起動します。

```
# dbx -k /vmunix
dbx version 3.11.4
Type 'help' for help.

stopped at [thread_block:1542 ,0xffffffffc00002f5334]

(dbx)
```

2. sysconfig コマンドを入力して、サブシステムの最初の構成を行います。

```
# sysconfig -c table_mgr
```

3. 次のように addobj コマンドを入力します。

```
(dbx) addobj /subsys/table_mgr.mod
(dbx) p &table_mgr_configure
0xffffffff895aa000

addobj コマンドにはサブシステムの絶対パス名を指定してください
(dbx セッションを開始する前にサブシステムがロードされている場合は、addobj コマンドを実行する必要はありません)。
```

初めて構成するサブシステムのサブシステム・コードにブレークポイントを設定するには、サブシステムのロードの後、カーネルが構成ルーチンを呼び出す前に、addobj コマンドを入力します。サブシステムのロードと構成ファイルの呼び出しの間で実行を停止するには、特別なルーチン subsys_preconfigure にブレークポイントを設定します。次の手順は、このブレークポイントの設定方法を示しています。

1. dbx デバッガを起動して、subsys_preconfigure ルーチンにブレークポイントを設定します。次のように入力してください。

```
# dbx -remote /vmunix
dbx version 3.11.4
```

Type 'help' for help.

stopped at [thread_block:1542 ,0xfffffc00002f5334]

(dbx) **stop in subsys_preconfigure**

(dbx) **run**

2. sysconfig コマンドを入力して table_mgr サブシステムを初期構成します。

sysconfig -c table_mgr

3. addobj コマンドを入力して、構成ルーチンにブレークポイントを設定します。

[5] stopped at [subsys_preconfigure:1546
,0xfffffc0000273c58]

(dbx) **addobj /subsys/table_mgr.mod**

(dbx) **stop in table_mgr_configure**

[6] stop in table_mgr_configure

(dbx) **continue**

[6] stopped at [table_mgr_configure:47 ,0xfffffffff895aa028]
(dbx)

4. subsys_preconfigure ルーチンで実行が停止したら dbx スタック・トレース・コマンド trace を実行することにより、構成要求がテスト中のサブシステムに対するものであることを確認できます。次に、サブシステム構成ルーチンにブレークポイントを設定します。

並列処理 — 従来の方法

Tru64 UNIX システムでは、Compaq C プログラムの並列処理は、次の 2 つの方法でサポートされます。

- OpenMP インタフェース — OpenMP Architecture Review Board により定義された並列処理インタフェース
- 従来の並列処理インタフェース — OpenMP インタフェース以前に開発された並列処理インタフェース

この付録では、従来 of 並列処理インタフェースについて説明します。つまり、OpenMP インタフェースの実装以前にサポートされた言語機能について説明します。OpenMP インタフェースについての詳細は、第 13 章を参照してください。

注意

従来のインタフェースを使用しているプログラマは、業界標準である OpenMP インタフェースへの変換を考慮する必要があります。

アプリケーションを並列処理に変換する場合、または新しい並列処理アプリケーションを作成する場合は、OpenMP インタフェースを使用する必要があります。

この付録を理解するには、スレッドとは何か、データ・アクセスがスレッド・セーフであるとは何を意味するか、といったマルチプロセッシングの基本的な概念を理解している必要があります。

並列処理の指示文には、ANSI C の `#pragma` 前処理指示文を使用します。これは、実装で定義された動作を C 言語に追加するための標準的な C のメカニズムです。このため、この付録では並列処理指示文 (または並列指示文) および並列処理プラグマという用語は置き換えて使用することが可能です。

この付録には、次のトピックに関する情報が含まれます。

- 並列処理プラグマの使用に適用される一般的なコード記述規則 (D.1 節)
- 並列処理プラグマの構文 (D.2 節)
- 実行時にスレッド・リソースの割り当てを制御するために使用可能な環境変数 (D.3 節)

D.1 並列処理プラグマの使用

この節では、すべての並列処理プラグマに適用される一般的なコーディング規則、およびプラグマの一般的な使用方法の概要を説明します。

D.1.1 一般的なコーディング規則

多くの場合、並列処理プラグマのコーディング規則は、Compaq C で記述された他のほとんどのプラグマの規則に従います。たとえば、マクロ置換はプラグマでは実行されません。その他の場合、並列処理プラグマは Compaq C の他のいかなるプラグマとも異なっています。これは、他のプラグマが一般にコンパイラ状態の設定 (メッセージまたはアラインメント) などの機能を実行するのに対し、これらのプラグマはステートメントであるためです。次に例を示します。

- プラグマは、関数本体の内部になければなりません。
- プラグマは、プログラムの実行に影響を与えます (付録で説明)。
- ほとんどのプラグマは、次に続く文に適用されます。プラグマを複数の文に適用する場合、複合文を使用する必要があります (つまり、文を中カッコで囲む必要があります)。

いくつかのプラグマには、補足情報を指定する修飾子を続けて指定することができます。これらの修飾子の使用を容易にするため、修飾子を含む行が `#pragma` で始まる場合、各修飾子は並列処理プラグマに続く行に指定することができます。次に例を示します。

```
#pragma parallel if(test_function()) local(var1, var2, var2)
```

上の文は、次のように記述することもできます。

```
#pragma parallel
#pragma if(test_function())
#pragma local(var1, var2, var2)
```

修飾子自体は複数行に分割できないことに注意してください。たとえば、上のコードは次のように記述することはできません。

```
#pragma parallel
#pragma if(test_function()) local(var1,
#pragma var2, var2)
```

D.1.2 一般的な使用法

`#pragma parallel` 指示文は、並列領域の開始に使用されます。

`#pragma parallel` 指示文に続く文は、並列領域の範囲を限定する働きをします。一般的にこれは、C の通常文 (他の並列処理指示文を含む場合も含めない場合もある) を含む複合文、または別の並列処理指示文 (この場合、並列領域はその 1 文で構成される) になります。並列領域を制限する複合文の中では、他の並列処理指示文により制御されない通常の C 文は、単に各スレッド上で実行されるだけです。他の並列処理指示文により制御される並列領域内の C 文は、指示文のセマンティクスに従って実行されます。

他のすべての並列処理プラグマは、`#pragma critical` を除き、構文的に並列領域内にある必要があります。並列領域内の最も一般的なコードは、スレッドにより実行される `for` ループに関するものです。このような `for` ループでは、その前に `#pragma pfor` を記述する必要があります。この構造により、異なるスレッドが `for` ループの異なる繰り返しを実行することが可能になり、プログラムの実行速度が速くなります。次の例は、ループの並列実行で利用できるプラグマです。

```
#pragma parallel local(a)
#pragma pfor iterate(a = 1 ; 1000 ; 1)
for(a = 0 ; a < 1000 ; a++) {
<loop code>
}
```

並列に実行されるループは、特定の特性に従う必要があります。次の事項が含まれます。

- インデックス変数は、`for` 文の 3 番目の式を除き、ループによって変更されることはありません。さらに、`for` 文の 3 番目の式は、インデックス変数を常に同じ量だけ修正します。
- ループの繰り返しは、それぞれ独立していなければなりません。つまり、ループのある繰り返しにより実行された計算は、他の繰り返しの結果に依存してはなりません。
- ループの繰り返しの数は、ループの開始前に確定していなければなりません。

プログラマは、並列ループがこれらの制限に従うことを確認する責任があります。

また、並列処理には、いくつかの異なるコード・ブロックを並列に実行するという使用方法もあります。 `#pragma psection` および `#pragma sections` 指示文が、この目的で使用されます。 次のコードは、これらの指示文の使用法を示します。

```
#pragma parallel
#pragma psection
{
    #pragma section
    { <code block>
    }
    #pragma section
    { <code block>
    }
    #pragma section
    { <code block>
    }
}
```

コード・ブロックには、さらに特定の制限が適用されます。たとえば、あるコード・ブロックは、他のコード・ブロックで実行された計算結果に依存してはなりません。

並列領域に指定可能なコードには、シリアル・コードもあります。シリアル・コードは、`#pragma pfor` と `#pragma psection` のいずれにも記述されません。この場合、並列領域の実行用に作成されたすべてのスレッドが同一のコードを実行します。これはリソースの浪費に見えますが、シリアル・コードを2つの `#pragma pfor` ループまたは2つの `#pragma psection` ブロックの間に配置するのは、多くの場合有用な方法です。シリアル・コードはすべてのスレッドにより実行されますが、この構造は、最初の `pfor` または `psection` の後に並列領域をクローズし、2番目の `pfor` または `psection` の前に別の並列領域をオープンするよりも効率的です。これは、並列領域の作成およびクローズの際に実行時のオーバーヘッドが発生するためです。

シリアル・コードを並列領域に配置する際には注意する必要があります。次の文は予想できない結果となる可能性があります。

```
a++;
b++
```

予想できない結果が発生するのは、すべてのスレッドがこの文を実行するために、変数 a および b が何回も増分されるためです。この問題を避けるためには、シリアル・コードを `#pragma one processor` ブロックで囲みます。次に例を示します。

```
#pragma one processor
{
a++;
b++;
}
```

このコードが実行されるまで、スレッドはこのコード以降を実行できないことに注意してください。

D.1.3 並列指示文のネスト

Compaq C は、ネストした並列領域を現在サポートしていません。構文上、並列領域が別の並列領域に含まれる場合、コンパイラはエラーを発生します。ただし、並列領域内で実行されるルーチンが別のルーチンを呼び出し、呼び出されたルーチンが並列領域に入ろうとする場合、2 番目の並列領域は直列的に実行されるため、エラーは発生しません。

`#pragma parallel` を除き、ほとんどの並列構造は他の並列構造を実行することはできません。たとえば、`#pragma pfor`、`#pragma one processor`、`#pragma section`、または `#pragma critical code block` コードの実行中に実行可能な他の並列処理構造は `#pragma critical` だけです。1 つの並列処理プラグマが構文的に別の並列処理プラグマ内にネストしている場合、コンパイラは不正な操作を示すエラーを発生します。ただし、コード・ブロック内で実行中のコードがあるルーチンに制御を移し、続いてそのルーチンがこれらの指示文のいずれかを実行する場合、その動作は予測できません。

(この付録の最初に説明したように、`#pragma critical` を除くすべての並列処理プラグマは、構文的に `#pragma parallel` 領域内にある必要があります。)

D.2 並列処理プラグマの構文

ここでは、各並列処理プラグマの構文について説明します。

次の並列処理プラグマは、従来の並列処理インタフェースによりサポートされます。

- `#pragma parallel` — コードの並列領域を示します (D.2.1 項)。

- `#pragma pfor` — 並列に実行されるループをマークします (D.2.2 項)。
- `#pragma psection` — 大量のコード・セクションを開始します。各コード・セクションは、他のコード・セクションと並列に実行されます (D.2.3 項)。
- `#pragma section` — `psection` 領域内の各コード・セクションを指定します (D.2.3 項)。
- `#pragma critical` — 一度に 1 つのスレッドだけを実行可能にするためにコードのクリティカルな領域へのアクセスを保護します (D.2.4 項)。
- `#pragma one processor` — 1 つのスレッドだけで実行されるコード・セクション (D.2.5 項)。
- `#pragma synchronize` — すべてのスレッドがこのポイントに達するまでスレッドを停止します (D.2.6 項)。
- `#pragma enter gate` および `#pragma exit gate` — より複雑な同期形式。すべてのスレッドが入力ゲートを通過するまで、終了ゲートを出ることは許可されません (D.2.7 項)。

D.2.1 `#pragma parallel`

`#pragma parallel` 指示文は、並列領域のコードをマークします。このプリグマの構文は、次のとおりです。

```
#pragma parallel [parallel-modifiers...] statement-or-code_block
```

`#pragma parallel` の *parallel-modifiers* は、次のとおりです。

```
local(variable-list)
byvalue(variable-list)
shared(variable-list)
if (expression) [[no]ifinline]
numthreads(numthreads-option)
```

local, byvalue,
shared 修飾子

local, byvalue, および shared 修飾子に対する引数 *variable-list* は、プログラムですでに宣言された変数をコンマで区切ってリストにしたものです。任意の数の local, byvalue, および shared 修飾子を指定できます。多数の変数を必要とする修飾子がある場合、これは有用です。

shared および byvalue 修飾子に続く変数は、各スレッドにより共有されます。

local 修飾子に続く変数は、各スレッドに対する固有の変数となります。領域外の変数値が、領域内に渡されることはありません。領域内では、local 修飾子の変数値は未定義です。local リストに変数を配置すると、並列領域内で変数を定義する場合と同様の効果があります。

これらの修飾子は、他の C コンパイラとの互換性を維持する目的のみで提供されています。Compaq C では、並列領域外で宣言されたアクセス可能なすべての変数は、並列領域内でアクセスおよび変更が可能であり、またすべてのスレッドから共有されます (変数が local 修飾子内で指定されていない限り)。次に例を示します。

```
int a,b,c;
#pragma parallel local(a) shared(b) byvalue(c)
{
<code that references a, b, and c>
}
```

上の文は、次のように記述することもできます。

```
int a,b,c;
#pragma parallel
{
int a;
<code that references a, b, and c>
}
```

if 修飾子

if 修飾子に続く式には、並列領域内のコードが、多くのスレッドにより実際に並列に実行されるか、それとも単一のスレッドにより直列に実行されるかを決定する条件を指定します。条件がゼロ以外の場合、コードは並列に実行されます。並列実行を行うかどうかの決定を実行時まで延期する場合にも、この修飾子を使用します。

少量のコードを並列に実行すると、直列に実行する場合よりも時間がかかることに注意してください。これは、コードの並列実行に必要なスレッドの生成と削除に関係したオーバーヘッドのためです。

noifinline 修飾子

noifinline 修飾子は、if 修飾子が存在する場合にだけ使用できます。省略時の値である ifinline

は、コンパイラに対し並列領域内に 2 つのバージョンのコードを生成するよう指示します。1 つは `if` 式がゼロでない場合に並列処理を行うため、もう 1 つは `if` 式がゼロの場合に直列処理を行うためです。`noifinline` 修飾子は、コンパイラに対し、1 つの形式のコードだけを生成するよう指示します。`noifinline` 修飾子を使用すると、生成されるコードの量は少なくなりますが、`if` 式の値がゼロの場合、コードの実行速度は低下します。

`numthreads` 修飾子

`numthreads-option` は、次のいずれかになります。

```
min=expr1, max=expr2
percent=expr
expr
```

どの場合にも、式が評価する値は正の整数値でなければなりません。`numthreads(expr)` は、`numthreads(min=0, max=expr)` と等価になります。`min` が指定されると、領域の実行に `expr1` スレッド (またはそれ以上) が利用可能な場合にのみコードは並列に実行されます。`max` が指定された場合、並列領域は `expr2` スレッド以外では実行されません。`percent` が指定された場合、並列領域は実行可能なスレッドの `expr` パーセントで実行されます。

並列領域の例は、次のとおりです。

```
#pragma parallel local(a,b) if(func()) numthreads(x)
{
  code
}
```

`func` がゼロ以外の値を返す場合に、このコード領域は実行されます。並列に実行される場合、最大 `x` のスレッドが使用されます。並列領域内では、各スレッドは変数 `a` と `b` のローカル・コピーを保持します。他の変数はすべて、複数のスレッドにより共有されます。

D.2.2 #pragma pfor

#pragma pfor 指示文は、ループに並列実行用のマークを付けます。構文上、#pragma pfor は並列領域内にのみ記述できます。このプラグマの構文は、次のとおりです。

```
#pragma pfor iterate(iterate-expressions) [pfor-options] for-statement
```

上に示したように、#pragma pfor に続いて *iterate* 修飾子を指定する必要があります。*iterate-expressions* の書式は、for ループと同様です。

```
index-variable = expr1 ; expr2 ; expr3
```

- *index-variable* = *expr1* は、#pragma pfor に続く for 文の最初の式と一致している必要があります。正しい動作を保証するために、*index-variable* は並列領域に対してローカルに存在しなければなりません。
- *expr2* 式には、ループが実行される回数を指定します。
- *expr3* 式には、ループの各繰り返しでインデックス変数に追加される値を指定します。

iterate-expressions は、#pragma pfor に続く for 文内の式と密接に関連しています。*iterate-expressions* 内で提供される情報に、for ループの実行方法を正確に反映させるのは、プログラマの責任です。

pfor-options は次のようになります。

```
schedtype(schedule-type)  
chunksize(expr)
```

schedtype オプションは、実行時スケジューラに対し、利用可能なスレッド間で繰り返しを分割する方法を指定します。有効な *schedule-type* 値は次のとおりです。

- *simple* — スケジューラは、実行可能なすべてのスレッド間で繰り返しを均等に分割します。省略時には、この値が使用されます。
- *dynamic* — スケジューラは各スレッドに対し、*chunksize* 式で指定された繰り返し数を指定します。
- *interleave* — これは、作業が交互にスレッドに割り当てられることを除き、*dynamic*と同じです。

- `gss` — スケジューラは、各スレッドにさまざまな繰り返し数を割り当てます。これは `dynamic` と類似していますが、各スレッドに対して `chunksize` で指定された固定値を割り当ててではなく、大きな数で始まり小さな数で終わるように繰り返し数を割り当てます。

`dynamic` または `interleave` の `schedtype` には `chunksize` オプションを指定する必要があります。このオプションは、繰り返しの回数を指定するために使用されます。

D.2.3 `#pragma psection` および `#pragma section`

`#pragma psection` および `#pragma section` 指示文では、相互に並行して実行されるコード・セクションを指定します。これらの指示文は、並列領域内でのみ記述できます。これらのプリAGMAの構文は、次のとおりです。

```
#pragma psection
{
    #pragma section
    stmt1
    #pragma section
    stmt2 . . .
    #pragma section
    stmtn
}
```

これらのプリAGMAには、修飾子はありません。 `#pragma psection` に続いて、カッコで囲まれたコード・ブロックを記述します。コード・ブロックは、 `#pragma section` 指示文およびそれに続く単独の文またはカッコで囲まれた文のグループで構成されます。 `psection` コード・ブロック内には、任意の数の `#pragma section` 指示文を指定できます。

D.2.4 `#pragma critical`

`#pragma critical` 指示文では、同時に複数のスレッドにより実行されるコード・セクションを指定します。このプリAGMAの構文は、次のとおりです。

```
#pragma critical [lock-option] statement-or-code-block
```

`lock-option` は、次のいずれかの値になります。

- `block` — ロックは、このクリティカル・セクションに固有です。このクリティカル・セクションの実行中に、他のスレッドは他のクリティカル・セクションを実行できますが、このクリティカル・セクションを実行できるのは1つのスレッドだけです。このオプションは、並列領域内のクリティカル・セクションにのみ指定可能です。
- `region` — ロックは、この並列領域に固有です。並列領域外のコードを実行する他のスレッドは、他のクリティカル・セクションを実行できますが、並列領域内の他のクリティカル・セクションを実行することはできません。このオプションは、並列領域内のクリティカル・セクションのみに指定可能です。
- `global` — グローバル・ロックです。このクリティカル・セクションの実行中は、他のクリティカル・セクションを実行することはできません。省略時はこの値が使用されます。
- `expr` — ユーザによるロック変数を指定する式です。この場合、式は 32 ビットまたは 64 ビットの整数変数である必要があります。

D.2.5 `#pragma one processor`

`#pragma one processor` 指示文は、単独のスレッドにより実行されるコード・セクションを示します。この指示文は、並列領域内にのみ記述できます。このプラグマの構文は、次のとおりです。

```
#pragma one processor statement-or-code-block
```

D.2.6 `#pragma synchronize`

`#pragma synchronize` 指示文は、すべてのスレッドがこのポイントに達するまで、次の文が実行されないようにします。この指示文は、並列領域内にのみ記述できます。このプラグマの構文は、次のとおりです。

```
#pragma synchronize
```

D.2.7 `#pragma enter gate` および `#pragma exit gate`

`#pragma enter gate` および `#pragma exit gate` 指示文を使用すると、`#pragma synchronize` よりも柔軟な同期が可能になります。これらの指示文は、並列領域内にのみ記述できます。領域内の各 `#pragma enter gate`

は、対になる `#pragma exit gate` が必要です。これらのプラグマの構文は次のとおりです。

```
#pragma enter gate (name)
```

```
#pragma exit gate (name)
```

name は、各ゲートを示す識別子です。ゲート名は、独自の名前空間内にあります。たとえば、ゲート名 `foo` は変数名 `foo` とは明確に区別されています。ゲート名は、使用前に宣言されることはありません。

このタイプの同期は、次のように機能します。`#pragma exit gate` 以降の文は、すべてのスレッドが対応する `#pragma enter gate` に達するまで、いかなるスレッドも実行することはできません。

D.3 環境変数

特定の場面での並列コード実行は、プログラム開始時のプロセス内の環境変数の値によって制御可能です。プログラムで最初に並列実行が開始される際にチェックされる環境変数は、現在のところ次のとおりです。

- `MP_THREAD_COUNT` — 作成するスレッドの数を実行時システムに示します。省略時には、作成するスレッド数としてシステムのプロセッサ数を使用します。
- `MP_CHUNK_SIZE` — 使用するチャンクサイズを示します。ユーザが `RUNTIME` スケジュール・タイプを要求した場合、またはチャンクサイズを必要とする別のスケジュール・タイプを要求する際にチャンクサイズを省略した場合に、使用するチャンクサイズを実行時システムに示します。
- `MP_STACK_SIZE` — 実行時システムに対し、スレッド作成時に各スレッドに何バイトのスタック・スペースを割り当てるかを示します。省略時の値は非常に小さいため、大きな配列をローカルとして宣言する場合には、それらの配列を割り当てられるほど大きなスタックを指定する必要があります。
- `MP_SPIN_COUNT` — 実行時システムに対し、条件が真になるのを待つ間、スピンする回数を示します。
- `MP_YIELD_COUNT` — Pthread 条件変数を待ちながらスリープ状態に入る前に、`sched_yield` の呼び出しと条件のテストを交互に何回実行するかを実行時システムに示します。

コマンド行シェルの規則に従って、これらの環境変数を整数値に設定することができます。環境変数が設定されていない場合、実行時システムは適正と思われる規定値を選択します (一般的に、経過時間が最小となるようにリソースを割り当てます)。



デバイス特殊ファイル名の処理

デバイスおよびデバイス特殊ファイルの命名および構成方法は、Tru64 UNIX オペレーティング・システムのバージョン 5.0 で変更されています。変更点についての詳細は、`dsfmgr(8)` および『システム管理ガイド』を参照してください。また、各種のインストレーション操作が新しいスタイルおよび古いスタイルのデバイス特殊ファイル名の使用方法に与える影響については、『インストレーション・ガイド』を参照してください。

デバイス処理に影響を与える変更点をサポートするために、オペレーティング・システムは新しいスタイルの名前と古いスタイルの名前の変換を容易にするための変換ルーチンを提供します。ここでは、その変換ルーチンについて説明します。これらのルーチンは、システムが古いスタイルの名前をサポートしなくなるまで、継続して提供されます。

デバイス特殊ファイルの名前を参照するシステム・コンポーネントまたはアプリケーションは、そのソース・コードを変更する必要があります。変更は、必要に応じて、すべてのデバイス特殊ファイル名を新しいスタイルの名前に置き換えるか、新しいスタイルの名前を導き出す変換ルーチンを使用して行います。

次の規則は、すべての変換ルーチンに適用されます。

- 関数呼び出しのポインタがヌルでない場合、引数が返されます。
- 必須なのは最初の引数だけです。ただし、他の引数が指定されない場合、関数は、状態情報を除き、有用な情報を返しません。

各ルーチン (`dsfcvt_btoc()`, `dsfcvt_ctob()`, `dsfcvt_ntoo()`, `dsfcvt_oton()`, `dsfcvt_noro()`, および `dsfcvt_cdevtoname()`) については、以下で説明します。ルーチンの説明に続いて、パラメータについて説明します。

<code>dsfcvt_btoc()</code>	ブロック名をキャラクタ名に変換します。ブロック・デバイス名が、デバイス状態データベース内で検索され、見つかると、対応するキャラクタ・
----------------------------	--

デバイス名が検索されます。キャラクタ・デバイス名が見つかったと、その名前がハードウェア ID とともに返されます。

構文:

```
int dsfcvt_btoc(  
    const char *blk_name,  
    char *chr_name,  
    char *blk_path,  
    char *chr_path,  
    long *hardware_id);
```

リターン値: ESUCCESS (両方のデバイス名が検出された), ENOENT (ブロック・デバイス名が検出されなかった), ENODEV (キャラクタ・デバイス名が検出されなかった)。

dsfcvt_ctob()

キャラクタ名をブロック名に変換します。キャラクタ名をデバイス状態データベース内で検索し、見つかったと、対応するブロック・デバイスを検索します。ブロック・デバイス名が見つかったと、その名前をハードウェア ID とともに返します。

構文:

```
int dsfcvt_ctob(  
    const char *chr_name,  
    char *blk_name,  
    char *chr_path,  
    char *blk_path,  
    long *hardware_id);
```

リターン値: ESUCCESS (両方のデバイス名が検出された), ENOENT (キャラクタ・デバイス名が検出されなかった), ENODEV (ブロック・デバイス名が検出されなかった)。

dsfcvt_ntoo()

新しいスタイルの名前を古いスタイルの名前に変換します。新しいスタイルの名前をデバイス状態データベース内で検索し、見つかったと、古いスタイルの名前 (存在する場合) をハードウェア ID とともに返します。

構文:

```
int dsfcvt_ntoo(  
    const char *new_name,
```

```
char *old_name,
char *new_path,
char *old_path,
long *hardware_id);
```

リターン値: ESUCCESS (両方のデバイス名が検出された), ENOENT (新しいデバイス名が検出されなかった), ENODEV (古いスタイルのデバイス名は検出されなかった)。

dsfcvt_oton()

古いスタイルの名前を新しいスタイルの名前に変換します。古いスタイルの名前をデバイス状態データベース内で検索し、見つかると、新しいスタイルの名前をハードウェア ID とともに返します。

構文:

```
int dsfcvt_oton(
    const char *old_name,
    char *new_name,
    char *old_path,
    char *new_path,
    long *hardware_id);
```

リターン値: ESUCCESS (両方のデバイス名が検出された), ENOENT (古いスタイルのデバイス名は検出されなかった), ENODEV (新しいデバイス名が検出されなかった)。

dsfcvt_noro()

名前を古いスタイルの名前または新しいスタイルの名前に変換します。入力名 (新しいスタイルまたは古いスタイル) をデバイス状態データベース内で検索します。一致した最初の古いスタイルの名前または新しいスタイルの名前が適切なフィールド内に返され、入力名が適切なフィールド内に返されます。ハードウェア ID も返されます。

検索名は、一方または両方の名前引数に指定できます。両方の場合、どちらも同じ名前でなければならず、同一の文字列値をポイントしてはなりません。

構文:

```
int dsfcvt_noro(
    char *new_name,
    char *old_name,
    char *new_path,
```

```
char *old_path,
long *hardware_id);
```

リターン値: ESUCCESS (両方のデバイス名が検出された), ENOENT (入力デバイス名は, 新しいスタイルまたは古いスタイルのいずれの場合にも検出されなかった), ENODEV (他のデバイス名は, 新しいスタイルまたは古いスタイルのいずれの場合にも検出されなかった)。

dsfcvt_cdevtoname()

デバイスに固有の数値識別子 (*cdev*) を新しいスタイルの名前およびパスに変換します。 *cdev* をデバイス状態データベース内で検索し, 見つかると, 新しい名前とパスを返します。

構文:

```
int dsfcvt_cdevtoname(
    dev_t cdev,
    char *new_name,
    char *new_path);
```

リターン値: ESUCCESS (デバイスが検出された), ENOENT (デバイスは検出されなかった)。

次のリストに, 関数の引数についての情報を示します。

blk_name | *chr_name* ブロックまたはキャラクタ・デバイスの名前へのポインタ。たとえば, それぞれ *dsk1a*, *rz4a*, または *rrz4a*。

blk_path | *chr_path* *blk_name* (例, /dev/disk/dsk1a) または *chr_name* (例, /dev/rdisk/dsk1a) により識別されるデバイス・パスへのポインタ。

new_name | *old_name* デバイス名へのポインタ。引数 *new_name* は新しいスタイルの命名規則に従い, 引数 *old_name* は古いスタイルの命名規則に従います。

new_path | *old_path* デバイス名のパスへのポインタ。引数 *new_path* は新しいスタイルの命名規則に従い, 引数 *old_path* は, 古いスタイルの命名規則に従います。たとえ

ば、`/dev/disk/dsk1a` および `/dev/rdisk/dsk1a` となり、古いスタイルの規則では、それぞれ `/dev/rz4a` および `/dev/rrz4a` となります。

hardware_id

デバイスに固有の数値識別子へのポインタ (この識別子は、カーネルによる割り当ておよび保守が行われます)。

cdev

カーネルによりデバイスに割り当てられる固有の主デバイス番号および副デバイス番号。



F

-om および cord によるプログラムの最適化

この付録では、cc コマンドの -om および -cord オプションを使用してプログラムを最適化する方法について説明します。-om はポストリンク最適化を実行します。-cord は、実行可能プログラムやシェアード・ライブラリ内のプロシージャを再編成して、キャッシュの効率を改善します。

注意

cc コマンドの -om および -cord オプションは、制御しやすく最適化の効率も良い spike ツール (10.1.3 項を参照) に置き換えられています。最適化とプロファイリング技術についての詳細は、それぞれ第 10 章と第 8 章を参照してください。

この付録の内容は、次のとおりです。

- -om ポストリンク最適化プログラムの使用 (F.1 節)
- -cord およびフィードバック・ファイルによるプロシージャの再編成 (F.2 節)

F.1 -om ポストリンク最適化プログラムの使用

cc コマンドの -om オプションを使用することによって、ポストリンク最適化を実行することができます。-om の概要を F.1.1 項で説明します。フィードバック・ファイルと一緒に -om を使用してプロファイル主導の最適化を行う方法については、F.1.2 項で説明します。

F.1.1 概要

-om ポストリンク最適化プログラムは、以下のコード最適化を実行します。

- nop (no operation) 命令、つまり、マシン状態に影響を及ぼさない命令の削除

- `.lita` データ、つまり、実行可能イメージのデータ・セクションの中で、64 ビット・アドレス指定用のアドレス・リテラルを保持している部分の削除。利用可能なオプションを使用して、最適化を実行した後、未使用の `.lita` エントリを削除して、`.lita` セクションを圧縮できます。
- ユーザが決定したサイズに従った共通シンボルの再割り当て

`-om` オプションを単独で使用すると、全範囲にわたるポストリンク最適化を実行することができます。 `-om` の後に次のいずれかのオプションを付けて指定すると、特定のポストリンク最適化を実行できます (詳細は `cc(1)` を参照してください)。

```
-WL, -om_compress_lita
-WL, -om_dead_code
-WL, -om_feedback
-WL, -om_Gcommon, num
-WL, -om_ireorg_feedback, file
-WL, -om_no_inst_sched
-WL, -om_no_align_labels
-WL, -om_split_procedures
```

`-om` オプションは、`-non_shared` オプションと併用した場合、すなわち実行可能プログラムの場合に最も効率が良くなります。たとえば次のように指定します。

```
% cc -non_shared -O3 -om prog.c
```

`-om` オプションは、最終的なリンクを行うときに指定しなければなりません。

F.1.2 `-om` によるプロファイル主導の最適化

`pixie` プロファイラ (`pixie(1)` を参照) は、プロファイル情報を生成します。このプロファイル情報は、`cc` コマンドの `-om` オプションと `-feedback` オプションで、特定の入力データ・セットによってプログラムに与えられた要求に合わせて、生成される命令シーケンスを調整するために使用されます。この手法は、実行可能ファイルの場合に最も効果があります。シェアード・ライブラリの場合は、F.2 節で説明しているように `cord` も使用できます。または `-om` オプションを省略します。

次の例は、この処理に必要な基本的な 3 つの手順、すなわち、(1) プロファイル主導の最適化に向けてプログラムを準備する、(2) 計測機構付きプログラムを作成して実行し、プロファイリング統計情報を収集する、(3) その情報をコンパイラとリンクにフィードバックして、実行可能コードの最適化に役立て

る，という手順を示しています。後の例では，これらの手順を詳細化して，開発中に行われた変更と複数のプロファイリング実行で得たデータを合わせて調節する方法を示しています。

```
% cc -feedback prog -o prog -non_shared -O3 *.c [1]
% pixie -update prog [2]
% cc -feedback prog -o prog -non_shared -om -O3 *.c [3]
```

- [1] プログラムが最初に `-feedback` オプション付きでコンパイルされたときに，拡張された特別な実行可能ファイルが作成されます。これには，コンパイラが実行可能ファイルをソース・ファイルに対応させるために使用する情報が含まれています。また，コンパイラへのプロファイルのフィードバック情報を格納するために後で使用するセクションも含まれています。このセクションは，最初にコンパイルしたときは `pixie` プロファイラ (手順 2) がフィードバック情報をまだ生成していないため，空のままです。`-feedback` オプションに指定するファイル名は，実行可能ファイルと同じ名前にしてください。この例では `prog` (`-o` オプションで指定) です。特に指定しないかぎり，`-feedback` オプションでは `-g1` オプションが適用され，プロファイルに最適なシンボルが付けられます。`-On` オプションを試して，対象のプログラムとコンパイラで実行時性能が最高になる最適化のレベルを調べてください。
- [2] `pixie` コマンドは，計測機構付きプログラム (`prog.pixie`) を作成して，それを実行します (`prof` オプション，`-update` が指定されているため)。実行の統計情報とアドレスのマッピング・データは自動的に命令カウント・ファイル (`prog.Counts`) と命令アドレス・ファイル (`prog.Addrs`) に収集されます。`-update` オプションにより，このプロファイル情報は拡張された実行可能ファイルに格納されます。
- [3] `-feedback` オプションを指定した 2 度目のコンパイルでは，拡張された実行可能ファイル内のプロファイル情報がコンパイラと (`-om` オプションを通じて) ポストリンク最適化プログラムを先導します。このカスタマイズされたフィードバックは，`-O3` および `-om` オプションによる自動最適化よりも優れています。コンパイラの最適化は，`-ifo` オプションや `-assume whole_program` オプションを `-feedback` オプションと組み合わせて使用すると，さらに効果が上がります。ただし，10.1.1 項で述べているように，大きいプログラムはソース・ファイルが 1 つしかない場合と同じようにコンパイルすることができないことがあります。

一般的な開発過程では、上記の手順 2 と 3 を必要な回数だけ繰り返して、ソース・コードの変更による影響が反映されるようにします。たとえば次のようにします。

```
% cc -feedback prog -o prog -non_shared -O3 *.c
% pixie -update prog
% cc -feedback prog -o prog -non_shared -O3 *.c
[modify source code]
% cc -feedback prog -o prog -non_shared -O3 *.c
.....
[modify source code]
% cc -feedback prog -o prog -non_shared -O3 *.c
% pixie -update prog
% cc -feedback prog -o prog -non_shared -om -O3 *.c
```

拡張された実行可能ファイル内のプロファイル情報は、コンパイル操作では失われないので、情報を更新する `pixie` の処理手順は、ソース・モジュールが変更されて再コンパイルされるたびに繰り返す必要はありません。ただし、変更のたびに、変更内容に応じて、実際のコードと古いフィードバック情報の違いが大きくなるため、最適化の有効度が低下します。最後の変更および再コンパイルの後に `pixie` 処理手順を行うと、最後に行ったコンパイルに合わせてフィードバック情報が正確に更新された状態になります。

拡張された実行可能ファイルは、内部にプロファイル情報があるため、通常の実行可能ファイルよりも大きくなります (通常は 3 ~ 5 パーセント)。開発が完了したら、`strip` コマンドを用いてプロファイル情報とシンボル・テーブル情報を削除できます。たとえば次のようにします。

```
% strip prog
```

プロファイルの正確な情報を得るために、計測機構付きプログラムを異なる入力で複数回実行したい場合があります。次の例では、プログラム `prog` の実行を 2 回計測して、そのプロファイル統計情報をマージする方法を示しています。このプログラムの出力は、異なる入力で実行するたびに異なります。

```
% cc -feedback prog -o prog -non_shared -O3 *.c[1]
% pixie -pids prog[2]
% prog.pixie[3]
(input set 1)
% prog.pixie
(input set 2)
% prof -pixie -update prog prog.Counts.*[4]
% cc -feedback prog -o prog -non_shared -om -O3 *.c[5]
```

- ① 最初のコンパイルでは、上記の例で説明したように拡張された実行可能ファイルが生成されます。
- ② 特に指定しなければ、計測機構付きプログラム (`prog.pixie`) を実行するたびに、`prog.Counts` という名前のプロファイル・データ・ファイ

ルが作成されます。-pids オプションを指定すると、計測機構付きプログラムを実行するたびに、作成されるプロファイル・データ・ファイルの名前にプロセス ID が付加されます (つまり、prog.Counts.pid になります)。したがって、後の実行で生成されるデータ・ファイルで上書きされることはありません。

- ③ 計測機構付きプログラムは 2 回実行され、そのたびに一意の名前でデータ・ファイルが生成されます。たとえば、prog.Counts.371 と prog.Counts.422 のようになります。
- ④ prof -pixie コマンドは、2 つのデータ・ファイルをマージします。-update オプションにより、この結合された情報で実行可能ファイル prog が更新されます。
- ⑤ 2 回目のコンパイルでは、2 回のプログラム実行で得られたプロファイル情報を結合したものを使用してプロファイル主導の最適化を実行します。

F.2 -cord によるプロファイル主導の再編成

cc コマンドの -cord オプションは cord ユーティリティを起動します。このユーティリティは、実行可能プログラムまたはシェアード・ライブラリ内のプロシージャを再編成して、命令キャッシュの効率を改善します。アプリケーションを実際に実行して得られたデータを含むフィードバック・ファイルを -cord への入力として使用できますが、これは F.1.2 項で述べたフィードバック・ファイルとは種類が異なるフィードバック・ファイル (pixie または prof -pixie コマンドで作成) です。次の例では、フィードバック・ファイルを作成し、そのフィードバック・ファイルを入力として、-cord オプションを使用して実行可能ファイルをコンパイルする方法を示します。

```
% cc -O3 -o prog *.c
% pixie -feedback prog.fb prog①
% cc -O3 -cord -feedback prog.fb -o prog *.c ②
```

- ① pixie コマンドは計測機構付きプログラムを作成して、それを実行します (prof のオプション -feedback が指定されているため)。-feedback オプションは、実行の統計情報を収集したフィードバック・ファイル (prog.fb) を作成し、そのファイルは次のステップでコンパイラが使用します。
- ② cc コマンドの -feedback オプションは、フィードバック・ファイルを入力として受け入れます。-cord オプションは cord ユーティリティを呼び出します。

フィードバックを使用してシェアード・ライブラリをコンパイルする方法も同様です。最適化を最も必要とするライブラリ・コードを動作させる 1 つ以上のプログラムでシェアード・ライブラリのプロファイルを行います。たとえば、次のようにコマンドを実行します。

```
% cc -o libexample.so -shared -g1 -O3 lib*.c[1]
% cc -o exerciser -O3 exerciser.c -L. -lexample[2]
% pixie -L. -incobj libexample.so -run exerciser[3]
% prof -pixie -feedback libexample.fb libexample.so exerciser.Counts[4]
% cc -cord -feedback libexample.fb -o libexample.so -shared -g1 -O3 lib*.c[5]
```

- ^[1] -g1 オプションを使用してシェアード・ライブラリをコンパイルし、ソース行ごとのフィードバック・データが得られるようにします。
- ^[2] ライブラリの重要な部分を動作させるプログラムを作成します。
- ^[3] シェアード・ライブラリとプログラムに計測機構が付加され、プロファイリングするために実行されます。
- ^[4] シェアード・ライブラリについてのみ、フィードバック・ファイルが作成されます。
- ^[5] プロファイルによって最も多く使用されていることが示されたコードの性能を最適化するために、シェアード・ライブラリの再コンパイル、再リンク、再編成を実行します。

同じ最適化レベルで作成されたフィードバック・ファイルを使用してください。

フィードバック・ファイルを作成して、`-non_shared` オプションを用いてプログラムをコンパイルする場合は、フィードバック・ファイルに対して `-cord` オプションよりも `-om` オプションを使用する方が有効です。

また、`cord` を `runcord` ユーティリティとともに使用することもできます。詳細は、`pixie(1)`、`prof(1)`、`cord(1)`、`runcord(1)` を参照してください。

索引

数字および記号

- /
(スラッシュを参照)
- ?
(疑問符を参照)
- 32 ビット・アプリケーション
メモリ使用の削減..... 10-23

A

- a.out** 2-24
情報の表示..... 2-30
省略時の実行可能ファイル... 2-4,
2-14
- abnormal_termination** 関数 11-15
- AdvFS**
直接入出力の使用..... 10-17
- AES**
アプリケーション・レベル・インタ
フェース要件..... 1-3
- AIO** ルーチン 10-18
- alias** コマンド (**dbx**)..... 5-27
- __align** 記憶クラス修飾子 2-9
- alloca** 関数 10-22
- Alpha** 命令セット
ネイティブでない命令の使用 10-17
- amalloc** 関数 10-23
- ANSI**
名前空間のクリーンアップ... 2-32

- ansi_args** オプション (**cc**)..... 10-4
- ansi_alias** オプション (**cc**) 10-4
- ANSI** 規格
アプリケーション開発の際の留意事
項 1-2, 1-3
- arch** オプション (**cc**)..... 10-4
- ASM** 2-33
- assert** 指示文
pragma assert 指示文 3-2
- assign** コマンド (**dbx**)..... 5-45
- assume noaccuracy_sensitive** オ
プション (**cc**)
(**-fp_reorder** オプションを参照)
- assume noaligned_objects** 10-20
- as** コマンド 2-3
コンパイル済みファイルのリン
ク 2-24
- Atom** ツール 9-1
Atom ツールの実行 9-1
インストール済みツールの使用 9-2
開発 9-10
開発中のツールのテスト 9-4
コマンド構文 9-3
例..... 9-2

B

- binlog** と **syslog**
EVM との相互作用..... 14-5

C

c_except.h ヘッダ・ファイル ... 11-4
call コマンド (**dbx**) 5-46
catch コマンド (**dbx**) 5-55
cc コマンド
 -pg オプション 8-5, 8-11
 -p オプション 8-13, 8-18
 System V 実行環境での使用... B-1
 -taso オプション A-4
 関数のインライン展開の指定 3-16
 コンパイル制御オプション... 2-14
 省略時の境界合わせの設定... 3-33
 省略時の動作 2-14
 追加のライブラリの指定 2-23
 デバッグ・オプション 5-8
 プロファイルの -g および -O オプション 8-4
 ライブラリの探索パスの指定 . 2-4
 リンクの起動 2-24
CFG_ATTR_BINTYPE データ型 C-7
CFG_ATTR_INTTYPE データ型 C-7
CFG_ATTR_LONGTYPE データ型 C-7
CFG_ATTR_STRTYPE データ型 C-7
CFG_ATTR_UINTTYPE データ型 C-7
CFG_ATTR_ULONGTYPE データ型 C-7
CFG_OP_CONFIGURE 要求コード C-7
CFG_OP_QUERY 要求コード .. C-7

CFG_OP_RECONFIGURE 要求コード C-7
cfg_subsys_attr_t データ型 C-6
-check_omp オプション (**cc**) .. 13-3
cma_debug() コマンド (**dbx**) .. 5-69
Compaq Extended Math Library
 アクセス方法 10-15
conti コマンド (**dbx**) 5-44
cont コマンド (**dbx**) 5-44
cord ユーティリティ F-5
coredump コマンド 5-6, 5-10
cpp コマンド
 定義済みマクロ 2-11
critical 指示文 D-10
Ctrl/Z
 dbx でのシンボル名の補完 ... 5-18
CXML
 アクセス方法 10-15
-c オプション (**cc**)
 多言語プログラムのコンパイル 2-19
-c オプション (**dbx**) 5-10
C 言語, プログラム検査
 移行 6-11
 移植性 6-12
 外部名 6-13
 関数および変数 6-7
 関数定義 6-6
 構造体, 共用体 6-6
 初期化されていない変数の使用方法 6-11
 データ型 6-5
 変数の初期化 6-11
 文字の使用法 6-12
C コンパイラ 2-1

C プリプロセッサ	2-10
各国語対応インクルード・ファイ	
ル	2-12
共通ファイルの取り込み	2-11
処理系固有の指示文	2-13
定義済みマクロ	2-10
C プログラム	
最適化に関する考慮事項	10-1

D

-D_FASTMATH オプション	
(cc)	10-16
-D_INLINE_INTRINSICS オプシ	
ョン (cc)	10-16
-D_INTRINSICS オプション	
(cc)	10-16
.dbxinit ファイル	5-9
dbx コマンド	5-1
(dbx デバッガ も参照)	
alias	5-27
args	5-40
assign	5-45
call	5-46
catch	5-55
cma_debug()	5-69
cont	5-44
conti	5-44
delete	5-30
disable	5-30
down	5-34
dump	5-58
edit	5-38
enable	5-30

file	5-35
func	5-34
goto	5-43
ignore	5-55
list	5-36
listobj	5-31
next	5-41
nexti	5-41
patch	5-45
playback input	5-61
playback output	5-63
print	5-56
printregs	5-58
quit	5-11
record input	5-61
record output	5-63
rerun	5-40
return	5-43
run	5-40
set	5-19
setenv	5-47
sh	5-32
source	5-61
status	5-29
step	5-41
stepi	5-41
stop	5-49
stopi	5-49
tlist	5-68
trace	5-52
tracei	5-52
tset	5-68
tstack	5-33, 5-68

unalias 5-28
 unset 5-19
 up 5-34
 use 5-32
 whatis 5-39
 when 5-53
 where 5-33
 whereis 5-38
 which 5-38
 / および? 5-37
dbx でのコマンドの探索 (/ および
 ?) 5-37
dbx デバッガ 1-6, 5-1
 (dbx コマンド も参照)
 dbx からのシェルの呼び出し . 5-32
 EDITMODE オプション 5-16
 EDITOR オプション 5-16
 -g オプション (cc) 5-8
 LINEEDIT オプション 5-16, 5-18
 エディタの呼び出し 5-38
 演算子の優先順位 5-12
 組み込みデータ型 5-13
 コマンド行オプション 5-9
 コマンド行編集 5-16
 コマンドの繰り返し 5-15
 コンパイル・コマンド・オプション
 (-g) 5-8
 初期化ファイル (dbxinit) 5-9
 シンボル名の補完 (Ctrl/Z) 5-18
 定義済みの変数 5-20
 デバッグの技法 5-5
 複数のコマンドの入力 5-18
dbx におけるイメージ起動 5-50
delete コマンド (dbx) 5-30

dis (オブジェクト・ファイル・ツ
 ル) 2-30
disable コマンド (dbx) 5-30
DMA
 直接入出力による使用 10-17
down コマンド (dbx) 5-34
dsfcvt* ルーチン E-1
dump コマンド (dbx) 5-58
-D オプション
 -taso オプションとの使用 A-7

E

EDITMODE 変数
 dbx コマンド行編集 5-16
EDITOR 変数
 dbx コマンド行編集 5-16
edit コマンド (dbx) 5-38
enable コマンド (dbx) 5-30
enter gate 指示文 D-11
environment 指示文
 pragma environment 指示文 .. 3-7
Event Manager
 (EVM を参照)
EVM 14-1
 (EVM 関数; イベント (EVM) も
 参照)
 binlog と syslog との相互作用 14-5
 イベント受信者
 定義 14-3
 イベント受信フィルタ 14-40
 イベント発信者
 定義 14-2
 起動と停止 14-5
 コールバック関数 14-5

プログラミング・インタフェース 14-32
ヘッダ・ファイルの必要条件 14-32
戻り状態コード 14-33

EVM 関数

EvmConnCheck..... 14-56e
EvmConnCreate 14-48e
EvmConnDestroy 14-48e
EvmConnDispatch..... 14-52e
EvmConnFdGet..... 14-56e
EvmConnSubscribe..... 14-52e
EvmConnWait 14-52e
EvmEventCreate 14-41e
EvmEventCreateVa..... 14-43e
EvmEventDestroy 14-41e
EvmEventFormat..... 14-52e
EvmEventNameMatch 14-62e
EvmEventNameMatchStr. 14-62e
EvmEventPost 14-48e
EvmEventRead 14-50e
EvmEventValidate..... 14-50e
EvmEventWrite..... 14-50e
EvmFilterCreate..... 14-59e
EvmFilterDestroy..... 14-59e
EvmFilterSet..... 14-59e
EvmFilterTest..... 14-59e
EvmItemGet 14-41e
EvmItemRelease..... 14-41e
EvmItemSet..... 14-41e
EvmItemSetVa..... 14-43e
EvmVarGet..... 14-45e
EvmVarRelease 14-45e
EvmVarSet..... 14-45e

EVM コマンド

evmget..... 14-4
exception_code 関数..... 11-6
exception_info 関数..... 11-6
exit gate 指示文 D-11
extern_model 指示文
 pragma extern_model 指示文 . 3-9
extern_prefix 指示文
 pragma extern_prefix 指示文. 3-14

F

-fast オプション (**cc**)..... 10-4
-feedback オプション (**cc**) 10-4,
 10-9, F-2
-feedback オプション (**pixie**)... F-5
file (オブジェクト・ファイル・ツール) 2-29
file コマンド (**dbx**)..... 5-35
fixso ユーティリティ 4-17
-fp_reorder オプション (**cc**) ... 10-4
fpu.h ヘッダ・ファイル..... 11-4
function 指示文
 pragma function 指示文 3-17
func コマンド (**dbx**) 5-34

G

goto コマンド (**dbx**) 5-43
gprof..... 8-1, 8-5, 8-11
 (プロファイル も参照)
-granularity オプション (**cc**).. 13-2
-g オプション (**cc**)..... 5-8, 8-4
-G オプション (**cc**) 10-4

H

hiprof .. 8-1, 8-6, 8-13, 8-17, 9-3
(プロファイル も参照)

I

-ieee オプション (**cc**) 10-6

IEEE 浮動小数点

(浮動小数点の範囲と処理 を参
照)

ifdef 指示文

各国語対応インクルード・ファイル
用 2-12

-ifo オプション (**cc**) 10-2, 10-4

ignore コマンド (**dbx**) 5-55

-inline オプション (**cc**) 10-4

inline 指示文

pragma inline 指示文 3-15

intrinsic 指示文

pragma intrinsic 指示文 3-17

ISO 規格

アプリケーション開発の際の留意事
項 1-3

-i オプション (**dbx**) 5-10

-I オプション (**dbx**) 5-10

K

KAP

推奨する使用方法 10-14

Kuck & Associates Preproces-

sor 10-14

(KAP も参照)

-k オプション (**dbx**) 5-10

L

ladebug デバッガ 5-1

ld コマンド 2-24

System V 実行環境での使用... B-1

-taso オプションの指定 A-5

taso 共用オブジェクトとのリン

ク A-7

ld リンカ

アセンブリ言語ファイルの処

理 2-24

オブジェクト・ファイルのリン

ク 1-5

シェアード・ライブラリとのリン

ク 4-8

直接または cc を介した使用.. 2-22

leave 文 11-14

libc.so 省略時の C ライブラリ . 2-24

libexc 例外ライブラリ 11-1

libpthread.so 12-2

limits.h ファイル C-10

LINEEDIT 変数

dbx コマンド行編集 5-16

dbx シンボル名の補完 5-18

linkage 指示文

pragma linkage 指示文 3-19

lint 6-2

lint ライブラリの作成 6-16

移行検査 6-11

移植性検査 6-12

エラー・メッセージ 6-18

オプション 6-2

警告クラス 6-24

コマンド構文 6-2

コーディング・エラーの検査 6-14

コーディング上の問題のチェック
 力 6-1
 データ型検査 6-5
 テーブル・サイズの増加 6-15
 プログラム・フロー検査 6-4
 変数および関数の検査 6-7
listobj コマンド (**dbx**) 5-31
list コマンド (**dbx**) 5-36
long ポインタ A-2

M

malloc 関数
 -taso との使用 A-8
 チューニング・オプション 10-22
member_alignment 指示文
 pragma member_alignment 指示
 文 3-22
message 指示文
 pragma message 指示文 3-24
-misalign 10-20
mmap システム・コール
 -taso オプションとの使用 A-9
 シェアード・ライブラリ 4-20
-module_path オプション
 (**dbx**) 5-10
-module_verbose オプション
 (**dbx**) 5-10
moncontrol ルーチン 8-36
 サンプル・コード 8-37
monitor_signal ルーチン 8-36
 サンプル・コード 8-39
monitor ルーチン
 プロファイルの制御 8-36

monstartup ルーチン 8-36
 サンプル・コード 8-37
MP_CHUNK_SIZE 変数 D-12
MP_SPIN_COUNT 変数 13-4,
 D-12
MP_STACK_SIZE 変数 13-4, D-12
MP_THREAD_COUNT 変数 13-4,
 D-12
MP_YIELD_COUNT 変数 13-4,
 D-12
mpc_destroy ルーチン 13-6
-mp オプション (**cc**) 13-1

N

nexti コマンド (**dbx**) 5-41
next コマンド (**dbx**) 5-41
nm (オブジェクト・ファイルのツ
 ール) 2-29
nm コマンド 2-29
-noaccuracy_sensitive オプション
 (**cc**)
 (-fp_reorder オプション を参照)

O

odump (オブジェクト・ファイルの
 ツール) 2-28
odump コマンド A-7
-Olimit オプション (**cc**) 10-4
-om
 ポストリンク最適化プログラム F-1
omp barrier 指示文 13-7
omp parallel 指示文 13-7

-omp オプション (cc)..... 13-2
-om オプション (cc)..... 10-4
one processor 指示文 D-11
OpenMP 指示文 13-1
optimize 指示文
 pragma optimize 指示文 3-29
-O オプション (cc) 2-23,
 8-4, 10-2, 10-3
 lint メッセージ抑制のための使
 用 6-4
 シェアード・ライブラリの問
 題 4-36

P

pack 指示文
 pragma pack 指示文 3-32
parallel 指示文 D-6
patch コマンド (dbx)..... 5-45
pdsc.h ヘッダ・ファイル 11-4
pfor 指示文 D-9
-pg オプション (cc) 8-5, 8-11
pixie 8-1, 8-13, 8-21,
 8-27, 9-3, 10-9, F-2
 (プロファイル も参照)
-pixstats オプション (prof) 8-23
playback input コマンド (dbx) 5-61
playback output コマンド
 (dbx) 5-63
pointer_size 指示文
 pragma pointer_size 指示文 .. 3-34
POSIX 規格
 アプリケーション開発の際の留意事
 項 1-2, 1-3

pragma
 unroll..... 3-36
-preempt_module オプション
 (cc) 10-4
-preempt_symbol オプション
 (cc) 10-4
printregs コマンド (dbx) 5-58
print コマンド (dbx) 5-56
prof..... 8-1, 8-13, 8-18, 8-21
 (プロファイル も参照)
PROFFLAGS
 環境変数..... 8-35

profiling
 プロファイル主導の最適化.... F-5
protect_headers_setup スクリプ
 ト..... A-12
-protect_headers オプション A-12
psection 指示文 D-10
-p オプション (cc)..... 8-13, 8-18

Q

quit コマンド (dbx)..... 5-11

R

RCS コード管理システム 1-7
record input コマンド (dbx) .. 5-61
record output コマンド (dbx) 5-63
rerun コマンド (dbx)..... 5-40
return コマンド (dbx) 5-43
run コマンド (dbx) 5-40
-r オプション (dbx) 5-10

S

SCCS (ソース・コード制御システム) 1-7
section 指示文 D-10
setenv コマンド (dbx) 5-47
 デバッガへの影響 5-18
 デバッガへの効果 5-16
setld ユーティリティ 1-7
set コマンド (dbx) 5-19
short ポインタ A-1
sh コマンド (dbx) 5-32
size (オブジェクト・ファイル・ツール) 2-30
SMP
 KAPにおける分解のサポート 10-14
source コマンド (dbx) 5-61
-speculate オプション (cc) 10-3
-speculate オプション (cc) 10-4
spike
 ポストリンク最適化プログラム
 ム 10-7
-spike オプション (cc) 10-4, 10-7, 10-9
spike コマンド 10-7, 10-9
status コマンド (dbx) 5-29
stdump (オブジェクト・ファイル・ツール) 2-32
stepi コマンド (dbx) 5-41
step コマンド (dbx) 5-41
\$stop_on_exec 変数 (dbx) 5-49
stopi コマンド (dbx) 5-49
stop コマンド (dbx) 5-49

STREAMS 1-8
strings コマンド 2-28
synchronize 指示文 D-11
sysconfigtab データベース C-3
sysconfig コマンド C-1, C-30
System V IPC 1-8
System V 実行環境 B-1
 cc コマンドの使用 B-1
 ld コマンドの使用 B-1
 アプリケーションのコンパイルとリンク B-1
 システム・コールの要約 B-4

T

-taso オプション
 cc コマンド A-4
 -T および -D オプションの効果 A-8
-testcoverage オプション
 (pixie) 8-27
Third Degree. 7-1, 8-1, 8-25, 9-3
 (プロファイルも参照)
threadprivate 指示文 13-8
TIS (Thread Independent Services) 12-7
tlist コマンド (dbx) 5-68
tracei コマンド (dbx) 5-52
trace コマンド (dbx) 5-52
Tru64 UNIX でサポートされるプログラム言語 1-5
try...except 文
 例外ハンドラでの使用 11-5
try...finally 文
 終了ハンドラでの使用 11-14

tset コマンド (**dbx**) 5-68
tstack コマンド (**dbx**) .. 5-33, 5-68
-tune オプション (**cc**)..... 10-4
-T オプション
 -taso オプションとの使用 A-8

U

unalias コマンド (**dbx**)..... 5-28
-unroll オプション (**cc**)..... 10-4
unroll 指示文
 pragma unroll 指示文 3-36
unset コマンド (**dbx**)..... 5-19
-update オプション (**pixie**) ... 10-9,
 F-2
-update オプション (**prof**) ... 10-12,
 F-4
uprofile 8-1, 8-13
 (プロファイル も参照)
up コマンド (**dbx**) 5-34
use_linkage 指示文
 pragma use_linkage 指示文 .. 3-36
use コマンド (**dbx**) 5-32
/usr/shlib ディレクトリ
 シェアード・ライブラリ 4-2

V

Visual Threads..... 5-1, 8-34

W

weak 指示文
 pragma weak 指示文 3-37
whatis コマンド (**dbx**) 5-39
when コマンド (**dbx**)..... 5-53

whereis コマンド (**dbx**)..... 5-38
where コマンド(**dbx**) 5-33
which コマンド (**dbx**) 5-38

X

X/Open 規格

アプリケーション開発の際の留意事
 項 1-2, 1-3

X/Open 転送インタフェース

 (**XTI**) 1-8
excpt.h ヘッダ・ファイル 11-4
-xtaso_short オプション (**cc**)... A-2
-xtaso オプション (**cc**).. 10-23, A-2
XTI..... 1-8

あ

アクティブ化レベル

dbx でのアクティブ化レベルについ
 ての情報の表示..... 5-58
 dbx での変更..... 5-34
 アクティブ化レベルにローカルな変
 数の値の表示 5-58
 スタック・トレースでの定義 . 5-4
 スタック・トレースによる確
 認 5-33

アプリケーション開発

 フェーズ..... 1-1

アプリケーション環境仕様

 (AES を参照)

アプリケーション・プログラム

 System V 実行環境でのコンパイル
 とリンク B-1
 -xtaso によるメモリ使用の削
 減 10-23

移植	6-13
移植性	1-2
コーディング上のガイドライ ン	10-16
最適化	10-1
作成のガイドライン.....	10-2
安全なプログラム	1-4
アーカイブ・ファイル	
セクション・サイズの決定...	2-30
選択した部分のダンプ	2-28

い

移植性	
外部名	6-13
規格	1-2
ビット・フィールド.....	6-13
位置合わせ	
ビット・フィールドの位置合 せ	2-7
イベント (EVM).....	14-1
(EVM も参照)	
イベント受信フィルタ	14-40
イベントの優先度	14-15
イベント名.....	14-8
設計	14-21
データ項目.....	14-7
内容	14-6
発信とアクセスの権限	14-6
フォーマット・データ項目.	14-13
イベント受信者 (EVM)	
定義	14-3
イベント・テンプレート	
作成方法	14-24

イベント発信者 (EVM)	
定義	14-2
意味規則	
名前の解決.....	4-6
インクルード・ファイル	
(ヘッダ・ファイルを参照)	
インストール・ツール.....	1-7
インライン・アセンブリ・コード	
(ASM)	2-33
インライン化, プロシージャ	
-D_INLINE_INTRINSICS オプショ ン (cc)	10-16
コンパイルの最適化.....	10-2

え

エディタ	
dbx からの呼び出し	5-38
演算子	
dbx 式の優先順位	5-12
エンディアン・バイト順	
Tru64 UNIX によるサポート...	2-5

お

大きなデータ・セット	
効率的な処理	10-14
オブジェクト・ファイル	
機械語への逆アセンブル	2-30
シンボル・テーブル情報の表	
示	2-29
セクション・サイズの決定...	2-30
選択した部分のダンプ	2-28
ファイル・タイプの決定	2-29

オブジェクト・ファイルのツール..... 2-27
 dis 2-30
 file 2-29
 nm 2-29
 odump 2-28
 size 2-30
 オプション, cc コンパイラ 2-14

か

下位互換性
 シェアード・ライブラリ 4-22
 開発ツール, ソフトウェア (**Tru64 UNIX**) 1-5
 外部参照
 リンク時における解決の減少 10-2
 外部名..... 6-13
 各国語対応プログラム
 インクルード・ファイル 2-12
 型キャスト
 lint による検査 6-7
 回避する時期 10-23
 型宣言
 dbx での表示..... 5-39
 環境変数
 dbx での設定..... 5-47
 EDITMODE..... 5-16
 EDITOR..... 5-16
 LINEEDIT 5-16
 MP_CHUNK_SIZE D-12
 MP_SPIN_COUNT ... 13-4, D-12
 MP_STACK_SIZE 13-4, D-12
 MP_THREAD_COUNT..... 13-4, D-12

MP_YIELD_COUNT . 13-4, D-12
 PROFFLAGS..... 8-35
 関数
 lint による検査 6-7
 カーネル・デバッグ
 -k オプション 5-10

き

記憶クラス修飾子
 __align 2-9
 規格..... 1-3
 (ANSI 規格; POSIX 規格;
 X/Open 規格 も参照)
 プログラミングの際の留意事項 1-2
 起動時間
 短縮 10-7
 疑問符 (?)
 dbx でのコマンドの探索 5-37
 キャッシュ使用
 コーディング上の提案 10-19
 キャッシュ・スラッシング
 防止 10-20
 キャッシュの使用
 cord による改善..... F-5
 キャッシュの衝突, データ
 回避 8-13, 10-20
 キャッシュのスラッシング
 回避 8-13
 キャッシュ・ミス
 回避 10-23
 キャッシュ・ミス, データ
 プロファイル 8-13
 境界合わせ
 データ型の境界合わせ 2-6
 境界合わせが誤っているデータ

(境界合わせされていないデータ
を参照)
境界合わせされていないデータ
回避 10-19
境界合わせ, データ
境界合わせ誤りの回避 10-19
共通ファイル
(ヘッダ・ファイルを参照)
共用オブジェクト 4-11
共用体
lintによる検査 6-6
共用メモリ
(System V IPCを参照)
切り捨てアドレス・サポート・オブ
ション A-2, A-4
(-taso オプションも参照)

く

クイックスタート
シェアード・ライブラリのロード時
間の短縮 10-7
使用 4-11
問題の解決
fixso 4-17
手動 4-14
組み込み関数
プラグマの同義語 3-18
組み込みデータ型
dbx コマンドでの使用 5-13

け

警告クラス 6-24

こ

コア・スナップショット
dbx の指定 5-10
dbx への指定 5-6
コア・ダンプ・ファイル
dbx の指定 5-5, 5-10
dbx への指定 5-6
シェアード・ライブラリの位置の指
定 5-31
ネーミング 5-64
構造化例外処理 11-5
構造体
lintによる検査 6-6
メンバの位置合わせ 2-6
構造体の境界合わせ
pragma member_alignment 指示
文 3-22
国際化
アプリケーション開発 1-3
コマンド行編集 (dbx) 5-16
コンパイラ・オプション (cc) ... 2-14
コンパイラ・コマンド
リンカの呼び出し 1-5
コンパイラ・システム 2-1
ANSI 名前空間のクリーンアッ
プ 2-31
C コンパイラ環境 2-14
C プリプロセッサ 2-10
オブジェクト・ファイルのツ
ール 2-27
ドライバ・プログラム 2-2
リンカ 2-22
コンパイラの最適化

-O オプションの使用 (cc) 10-3
推奨する最適化レベル 10-2
フィードバック・ファイルによる改善 10-9, F-2, F-5
コンパイル, アプリケーション
System V 実行環境 B-1
コーディング・エラー
lint による検査 6-14
コーディング上の提案
C 固有の考慮事項 10-21
キャッシュ使用パターン ... 10-19
データ型 10-17
データの境界合わせ 10-19
符号についての考慮事項 ... 10-22
ライブラリ・ルーチンの選択 10-15
コールバック関数 (EVM) 14-5

さ

最適化

-om および cord による F-1
-O オプションの使用 (cc) 10-3
spike の使用 10-7
技術 10-1
コンパイラ・オプション 2-23
コンパイル・オプション 10-2
フィードバック・ファイルによる改善 10-9, F-2, F-5
プロファイル時 8-4
プロファイル主導 10-9, F-2
ポストリンク 10-7, F-1

し

シェアード・ライブラリ
C プログラムとのリンク 4-8

dbx での表示 5-31
mmap システム・コール 4-20
下位互換性 4-22
概要 4-2
コア・ダンプのための位置の指定 5-31
作成 4-9
指定解除 4-8
使用不可能なアプリケーション 4-9
シンボルの解決 4-4
性能に関する考慮事項 10-7
探索パス 4-4
バイナリ非互換 4-22
バージョン管理 4-21
バージョン識別子 4-23
複数バージョン従属 4-27
部分バージョン 4-26
プロファイル 8-6, 8-30
マイナー・バージョン 4-25
命名規則 4-3
メジャー・バージョン 4-25
利点 4-2
を使用したプログラムのデバッグ 4-19
シェアード・ライブラリのバージョン管理
定義 4-22
式
dbx での値の表示 5-44, 5-56
dbx の演算子の優先順位 5-12
シグナル 1-8
dbx で停止する 5-55
指示による分解
KAP の使用 10-14
指示文

pragma extern_model 指示文	3-9	自動分解	
ifdef	2-12	KAP の使用	10-14
pragma assert 指示文	3-2	終了処理	11-5
pragma environment	3-7	終了ハンドラ	
pragma extern_prefix	3-14	コーディング方法	11-14
pragma function	3-17	受信者	
pragma inline	3-15	(イベント受信者 (EVM) を参照)	
pragma intrinsic	3-17	出力エラー	
pragma linkage	3-19	dbx による特定	5-6
pragma member_alignment	3-22	条件コード	
pragma message	3-24	dbx での記述	5-53
pragma optimize	3-29	シンボル	
pragma pack	3-32	解決	4-6
pragma pointer_size	3-34	シェアード・ライブラリのシンボル	
pragma unroll	3-36	の解決	4-4
pragma use_linkage	3-36	探索パス	4-4
pragma weak	3-37	名前の解決の意味規則	4-6
取り込む	2-12	未解決のシンボル処理のオプション	4-7
システム・コール		割り当て	4-35
System V 実行環境での相違点	B-2	シンボル, 強い	2-32
システム・ライブラリ	4-1	シンボル・テーブル	
実行可能なディスク・ファイル		ANSI 名前空間のクリーンアップ	2-32
dbx でのパッチ	5-45	表示	2-29
実行可能プログラム		シンボルの解決	
作成	2-4, 2-24	シェアード・ライブラリ	4-4
実行	2-26	シンボル名	
実行時		dbx での Ctrl/Z を使用した補完	5-18
改善のためのコーディング上のガイドライン	10-16	シンボル, 弱い	2-32
実行時に影響を及ぼす作成オプション	10-2	シンメトリック・マルチプロセッシング (SMP を参照)	
実行時エラー			
dbx による特定	5-5		

す

- スタック・トレース
 - dbx での取得..... 5-33
 - アクティブ化レベルの確認に使用 5-33
 - アクティブ化レベルの定義に使用 5-4
- スタティック・サブシステム
 - 定義 C-2
- スラッシュ (/)
 - dbx でのコマンドの探索 5-37
- スレッド 1-8, 12-1
 - Visual Threads..... 5-1
 - 直接入出力の使用 10-18
 - マルチスレッド・アプリケーションのデバッグ 5-68
 - マルチスレッド・アプリケーションのプロファイル..... 8-34
- スレッド・セーフ・コード
 - 作成方法 12-6
- スレッド・セーフ・ルーチン
 - 特性 12-4

せ

- 整数の乗算
 - 浮動小数点の乗算との置換. 10-17
- 整数の除算
 - 浮動小数点の除算との置換. 10-17
- 性能
 - プロファイルによる性能の向上 8-1
- 接尾語, ファイル名
 - プログラミング言語ファイル用 2-4
- セマフォ
 - (System V IPC を参照)

そ

- 属性
 - 初期値の割り当て..... C-3
 - 定義 C-3
 - 定義の例..... C-10
- 属性テーブル
 - 内容 C-5
- 属性のデータ型 C-7
- 属性要求コード C-7
- ソケット 1-8
- ソフトウェア開発ツール (Tru64 UNIX)..... 1-5
- ソース行と命令のプロファイル. 8-13
- ソース・コード
 - dbx での探索..... 5-37
 - dbx でのリスト 5-36
 - lint によるチェック 6-2
- ソース・コード制御システム
 - SCCS..... 1-7
- ソース・コードの互換性
 - System V 実行環境..... B-1
- ソース・ディレクトリ
 - dbx での指定..... 5-32
- ソース・ファイル
 - dbx での指定..... 5-35
 - アクセスの制御 1-6

た

- 多言語プログラム
 - コンパイル..... 2-19
- 探索する順序
 - リンカ・ライブラリ..... 2-25
- 探索パス
 - 限定 4-8

シェアード・ライブラリ	4-4
ローダ	4-5
探索パスの限定	4-8
単純ポインタ	A-2

ち

直接入出力	
性能改善のための使用	10-17
直接メモリ・アクセス (DMA を参照)	

つ

強いシンボル	2-32
ツール	
ソフトウェア開発用の主なツ ール	1-5

て

定義済みの変数	
dbx	5-20
ディスク・ファイル, 実行可能	
dbx でのパッチ	5-45
ディレクトリ	
リンクが探索する順序	2-25
ディレクトリ, ソース	
dbx での指定	5-32
テキスト・セグメント	
-taso オプションの効果	A-7
デバイス特殊ファイル名	
古いスタイルと新しいスタイルの変 換	E-1
デバugga	

(dbx デバugga を参照)	
デバugga	
一般的概念	5-3
カーネル・デバugga (-k オプショ ン)	5-10
シェアード・ライブラリを使用して いるプログラム	4-19
デバugga・ツール	1-6
(Third Degree; dbx デバugga; lint も参照)	
Tru64 UNIX でのサポート	1-6
テンプレート (EVM)	
(イベント・テンプレートを参 照)	
データ型	
-O オプションの効果 (cc)	10-3
Tru64 UNIX でサポートされる 型	2-4
位置合わせの修飾	2-9
キャスト	6-7
構造体の位置合わせ	2-6
混在	6-5
コーディング上の提案	10-17
サイズ	2-5
属性用	C-7
配列	6-6
配列ポインタ	6-6
浮動少数点の範囲と処理	2-5
データ型 (組み込み)	
dbx コマンドでの使用	5-13
データ・キャッシュの衝突	
回避	8-13, 10-20
データ・キャッシュ・ミス	
プロファイル	8-13

データ構造体
 割り当て上の提案 10-20
データ・セグメント
 -taso オプションの効果 A-7
データ・セット, 大きな
 効率的な処理 10-14
データの型, 変数 (EVM) 14-20
データの境界合わせ
 コーディング上の提案 10-19
データの再使用
 効率的な処理 10-14
データ・フロー分析
 コンパイルの最適化 10-2
データ割り当て
 コーディング上の提案 10-22

と

動的に構成可能なサブシステム
 作成 C-1
 定義 C-2
ドライバ・プログラム
 コンパイラ・システム 2-2
トラステッド・プログラム 1-4

な

名前空間
 クリーンアップ 2-31
名前の解決
 意味規則 4-6

は

バイト順
 Tru64 UNIX によるサポート .. 2-5

バイナリ非互換
 シェアード・ライブラリ 4-22
パイプ 1-8
配布メディア
 アプリケーションのロード 1-8
配列境界
 実行時検査の有効化 2-19
配列の使用
 C における最適化 10-23
 割り当て上の考慮事項 10-20
発信者
 (イベント発信者 (EVM) を参照)
パラメータ
 (属性 を参照)
バージョン管理
 シェアード・ライブラリ 4-21

ひ

ビット・フィールド 6-13
ヒープ・メモリの解析
 プロファイル 8-25

ふ

ファイル
 (アーカイブ・ファイル, オブ
 ジェクト・ファイル, ソース・
 ファイル, ヘッダ・ファイル,
 実行可能なディスク・ファイ
 ル を参照)
ファイル共用
 性能に対する効果 10-7
ファイル名
 プログラミング言語ファイルの接尾
 語 2-4

フィードバック・ファイル	
cord によるプロファイル主導の再編成	F-5
プロファイル主導の最適化...	10-9, F-2, F-5
フォーマット・データ項目	
(EVM)	14-13
複合ポインタ	A-2
符号付き変数	
性能に与える影響	10-22
符号なし変数	
性能に与える影響	10-22
浮動小数点演算	
-fp_reorder オプション (cc) ...	10-3
KAPの使用	10-14
例外処理	10-6
浮動小数点演算 (複雑な場合)	
CXML の使用	10-15
浮動小数点の範囲と処理	
IEEE 標準	2-5
プラグマ	
assert 指示文	3-2
critical	D-10
enter gate	D-11
environment	3-7
exit gate	D-11
extern_model 指示文	3-9
extern_prefix	3-14
function	3-17
inline	3-15
intrinsic	3-17
linkage	3-19
member_alignment	3-22
message	3-24
omp barrier	13-7
omp parallel	13-7
one processor	D-11
optimize	3-29
pack	3-32
parallel	D-6
pfor	D-9
pointer_size	3-34
psection	D-10
section	D-10
synchronize	D-11
threadprivate	13-8
use_linkage	3-36
weak	3-37
ブリプロセッサ, C	
(C ブリプロセッサ を参照)	
ブレイクポイント	
条件指定によるブレイクポイントの設定	5-50
設定	5-49
続行	5-44
プロシージャでの設定	5-50
フレーム・ベースの例外処理 ...	11-5
プログラムのインストール・ツール	1-7
プログラムのチェック	
C プログラム	6-2
プログラムのプロファイル	8-1
プロシージャのインライン化	
-D_INLINE_INTRINSICS オプション (cc)	10-16
コンパイルの最適化	10-2
プロシージャ呼び出し	
効率的な処理	10-14

プロセス間通信	
STREAMS.....	1-8
System V IPC	1-8
X/Open 転送インタフェース	
(XTI)	1-8
シグナル.....	1-8
スレッド.....	1-8
ソケット.....	1-8
パイプ.....	1-8
プロファイリング	
pixie	10-9
spike.....	10-9
最適化のためのフィードバック・	
ファイル	10-9
プロファイル主導の最適化...	10-9
プロファイル.....	8-1
Atom ツールの使用	9-1
cord によるプロファイル主導の再編	
成	F-5
gprof.....	8-5, 8-11
-g オプション (cc)	8-4
hiprof.....	8-5, 8-13, 8-17, 9-3
hiprof を使用した PC サンプリン	
グ	8-17
moncontrol ルーチン	8-36
monitor_signal ルーチン	8-36
monitor ルーチンを使用した .	8-36
monstartup ルーチン	8-36
-O オプション (cc).....	8-4
-pg オプション (cc).....	8-5, 8-11
pixie	8-13, 8-21, 8-27, 9-3, F-2
prof.....	8-13, 8-18, 8-21
-p オプション (cc)	8-13, 8-18
-testcoverage オプション	
(pixie)	8-27
Third Degree.....	7-1, 8-25, 9-3
uprofile	8-13
行ごとの表示の制限.....	8-30
最適化オプションの使用	8-4
最適化用のフィードバック・ファイ	
ル	F-2, F-5
サンプル・プログラム	8-2
シェアード・ライブラリ	8-6, 8-30
システム・モニタの使用	8-24
システム・リソースの使用の最小	
化	8-23
手動による設計とコードの最適	
化	8-5
ソース行と命令	8-13
ソース行と命令の , CPU 時間また	
はイベントのプロファイル	8-13
テスト・ケースの重要性の確	
認	8-27
データ・ファイルのマージ..	8-31,
10-12, F-4	
表示する情報の制限.....	8-28
表示する情報の選択.....	8-28
ヒープ・メモリの解析の使用	8-25
プロファイル主導の最適化....	F-2
マルチスレッド・アプリケーショ	
ン	8-34
命令とソース行	8-13
メモリのリーク	8-25
呼び出しグラフを使用した CPU 時	
間のプロファイル	8-5
分解	
KAP の使用.....	10-14

へ

並列処理

OpenMP 以前の方法 D-1

OpenMP 指示文 13-1

ヘッダ・ファイル

c_except.h 11-4

except.h 11-4

fpu.h 11-4

pdsc.h 11-4

各国語対応 2-12

規格対応 1-3

取り込み 2-11

編集

dbx でのコマンド行編集 5-16

変数 5-1

(環境変数 も参照)

dbx での名前の表示 5-38

dbx の定義済みの変数 5-20

アクティブ化レベルの値の調

査 5-58

値の割り当て 5-45

型宣言の表示 5-39

トレース 5-52

有効範囲の決定 5-52

変数データの型 (EVM) 14-20

変数, 符号付きまたは符号なし

性能に与える影響 10-22

ほ

ポインタ

32 ビット A-2

long A-2

short A-1

単純 A-2

複合 A-2

ポインタ用のメモリ使用の削減

(-xtaso) 10-23

ポインタ・サイズ

変換 A-1

ま

マクロ

cpp コマンド 2-11

定義済み 2-10

マジック・ナンバ 2-29

マルチスレッド・アプリケーション

作成 12-13

プロファイル 8-34

ライブラリの開発 12-1

マルチプロセッシング, シンメトリック

(SMP を参照)

み

未解決のシンボル

ld コマンドのオプション 4-7

シェアード・ライブラリ 4-4

ミス, キャッシュ

回避 10-23

め

命名規則

シェアード・ライブラリ 4-3

命令セット, **Alpha**

ネイティブでない命令の使用 10-17

命令とソース行のプロファイル. 8-13
メッセージ, **IPC**
 (System V IPC を参照)
メモリ
 dbx での内容の表示 5-59
 メモリ使用のチューニング. 10-22
 リークの検出 7-1, 8-25
メモリ・アクセス
 初期化されていないまたは誤った
 データの検出 7-1

も

文字
 C プログラムでの使用方法 ... 6-12
モニタリング・ツール 8-1

ゆ

有効範囲 5-1
 (アクティブ化レベル も参照)
 dbx 変数の有効範囲の指定 ... 5-12
 アクティブ化レベルの決定 5-4
 変数の有効範囲の決定 5-52

よ

呼び出し
 (プロシージャ呼び出し を参照)
呼び出しグラフを使用したプロファイ
 ル 8-5
弱いシンボル 2-32

ら

ライブラリ

 シェアード 4-1
 実行時 2-24
 プログラムとのリンク 2-24
ライブラリ定義ファイル (**lint**) . 6-16
ライブラリの選択
 性能に対する影響 10-15

り

リンカ
 (ld リンカ を参照)
リンク, アプリケーション
 System V 実行環境 B-1
 コンパイラ・コマンドの使用 2-23
リンク・オプション
 ファイル共有の効果 10-7

る

ルーチン
 dbx の制御のもとでの呼び出
 し 5-46
ループ
 KAP 最適化 10-14
 lint の分析 6-4

れ

例外
 構造化 11-5
 定義 11-1
 フレーム・ベース 11-5
例外コード 11-6
例外処理 11-1
 アプリケーション開発時の考慮事
 項 11-1

浮動小数点演算
性能に関する考慮事項 10-6
ヘッダ・ファイル 11-4
例外ハンドラ
実行時スタック上への配置... 11-7
例外フィルタ 11-6
レジスタ
dbx での値の表示 5-58
列挙データ型 6-7

ろ

ローダ

探索パス 4-5
ロード可能なサブシステム
定義 C-2
ロード時間
シェアード・ライブラリのロード時
間の短縮 10-7

わ

割り当て、データ
コーディング上の提案 10-22



Tru64 UNIX ドキュメントの購入方法

Tru64 UNIX ドキュメントのご購入については、弊社担当営業または日本ヒューレット・パッカートの各営業所/代理店にお問い合わせください。

各ドキュメント・キットの注文番号は以下のとおりです。ドキュメント・キットに含まれるマニュアルの内容については『ドキュメント概要』を参照してください。

キット名	注文番号
Tru64 UNIX Documentation CD-ROM	QA-6ADAA-G8
Tru64 UNIX Documentation Kit	QA-6ADAA-GZ
End User Documentation Kit	QA-6ADAB-GZ
- Startup Documentation Kit	QA-6ADAC-GZ
- General User Documentation Kit	QA-6ADAD-GZ
- System and Network Management Documentation Kit	QA-6ADAE-GZ
Developer's Documentation Kit	QA-6ADAF-GZ
Reference Pages Documentation Kit	QA-6ADAG-GZ
TruCluster Server Documentation Kit	QA-6BRAA-GZ
Tru64 UNIX 日本語ドキュメント・キット	QA-6ADJB-GZ
スタートアップ・ドキュメント・キット	QA-6ADJC-GZ
一般ユーザ・ドキュメント・キット	QA-6ADJD-GZ
システム/ネットワーク管理ドキュメント・キット	QA-6ADJE-GZ
プログラミング・ドキュメント・キット	QA-6ADJF-GZ
CDE 翻訳ドキュメント・キット	QA-6ADJG-GZ
TruCluster Server 日本語ドキュメント・キット	QA-05SJA-GZ
Advanced Server for UNIX 日本語ドキュメント・キット	QA-5U2JA-GZ



マニュアルに対するご意見

Tru64 UNIX
プログラミング・ガイド
AA-RK3MD-TE

弊社のマニュアルに関して、ご意見、ご要望、または内容の不明確な部分など、お気づきの点がございましたら、下記にご記入の上、弊社社員にお渡しくださるようお願い申し上げます。

マニュアルの採点：

	大変良い	良い	普通	良くない
正確さ(説明どおりに動作するか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
情報量(十分か)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
分かり易さ	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
マニュアルの構成	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
図(役立つか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
例(役立つか)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
索引(項目の検索性)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ページ・レイアウト(情報の検索性)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

内容の不明確な部分がありましたら、以下にご記入ください：

ペー ジ

その他お気づきの点がございましたら、以下にご記入ください：

ご使用のソフトウェアのバージョン： _____

貴社名/部課名 _____

御名前 _____

記入日 _____

(注) 当用紙を受け取った弊社社員は、すみやかに下記にお送りください。

ビジネスクリティカルシステム統括本部 **BCS** 技術本部 **Alpha** ソフトウェア技術部