

リンカーとライブラリ

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

Part Number 816-1255-10
2001 年 8 月

Copyright 2001 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. All rights reserved.

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

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

Federal Acquisitions: Commercial Software-Government Users Subject to Standard License Terms and Conditions.

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

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

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

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

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

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

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

「ATOK」は、株式会社ジャストシステムの登録商標です。

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

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

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

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

Unicode は、Unicode, Inc. の商標です。

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

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

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

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

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

原典: *Linker and Libraries Guide*

Part No: 816-0559-10

Revision A



目次

	はじめに	11
1.	序章	17
	概要	17
	リンク編集	18
	実行時リンク	19
	動的実行可能ファイル	20
	共有オブジェクト	20
	関連情報	21
	動的リンク	21
	アプリケーションのバイナリインタフェース	21
	32 ビットおよび 64 ビット環境	22
	環境変数	22
	サポートするツール	23
	更新された機能	23
2.	リンカー	25
	概要	25
	リンカーの起動	26
	直接起動	27
	コンパイラドライバを使用する	27

リンカーオプションの指定	28
入力ファイルの処理	29
アーカイブ処理	30
共有オブジェクトの処理	31
追加ライブラリとのリンク	32
初期設定および終了セクション	38
シンボルの処理	41
シンボル解析	41
未定義シンボル	46
出力ファイル内の一時的シンボル順序	50
追加シンボルの定義	51
シンボル範囲の縮小	56
外部結合	60
出力イメージの生成	61
再配置処理	63
ディスプレイメント再配置	63
デバッグングエイド	65
3. 実行時リンカー	69
概要	69
共有オブジェクトの依存関係の配置	70
実行時リンカーによって検索されるディレクトリ	70
デフォルトの検索パスの設定	73
動的ストリングトークン	74
再配置処理	75
シンボルの検索	76
再配置が実行される時	78
再配置エラー	80
追加オブジェクトの読み込み	81

動的依存関係のレイジーローディング	82
初期設定および終了ルーチン	84
セキュリティ	87
実行時リンクのプログラミングインタフェース	88
追加オブジェクトの読み込み	90
再配置処理	91
新しいシンボルの入手	98
機能チェック	102
デバッグングエイド	103
4. 共有オブジェクト	107
概要	107
命名規約	108
共有オブジェクト名の記録	109
依存関係を持つ共有オブジェクト	113
依存関係の並べ変え	114
フィルタとしての共有オブジェクト	115
標準フィルタの生成	116
補助フィルタの生成	118
フィルターの処理	120
性能に関する考慮事項	120
役に立つツール	121
基本システム	123
動的依存関係のレイジーローディング	124
位置に依存しないコード	124
共有可能性の最大化	126
ページング回数の削減	128
再配置	129
-Bsymbolic の使用	134

5. バージョンアップ 139

概要 139

インタフェースの互換性 140

内部バージョンアップ 141

バージョン定義の作成 142

バージョン定義への結合 148

バージョン結合の指定 152

バージョンの安定性 157

再配置可能オブジェクト 157

外部バージョンアップ 158

バージョンアップファイル名の管理 158

プロセス内での互換性 161

6. サポートインタフェース 163

概要 163

リンカーのサポートインタフェース 163

サポートインタフェースの呼び出し 164

サポートインタフェース関数 165

サポートインタフェースの例 167

実行時リンカーの監査インタフェース 169

名前空間の確立 170

監査ライブラリの作成 171

監査インタフェースの呼び出し 172

ローカル監査の記録 173

監査インタフェースの関数 174

監査インタフェースの例 179

監査インタフェースのデモンストレーション 180

監査インタフェースの制限 181

実行時リンカーのデバッガインタフェース	181
制御プロセスとターゲットプロセス間の対話	182
デバッガインタフェースのエージェント	184
デバッガエクスポートインタフェース	185
デバッガインポートインタフェース	195
7. オブジェクトファイル	199
概要	199
ファイル形式	200
データ表現	201
ELF ヘッダー	203
ELF 識別	208
セクション	212
セクショングループ	226
特殊セクション	228
文字列テーブル	235
シンボルテーブル	237
Syminfo テーブル	247
再配置	248
Comdat セクション	267
バージョン情報	268
注釈セクション	274
移動セクション	276
動的リンク	279
プログラムヘッダー	279
プログラムの読み込み (プロセッサ固有)	287
実行時リンカー	294
ハッシュテーブル	323
初期設定関数および終了関数	324

- 8. **mapfile** のオプション 325
 - 概要 325
 - mapfile オプションの使い方 326
 - mapfile の構造と構文 326
 - セグメントの宣言 327
 - 対応付け指示 331
 - サイズシンボル宣言 334
 - ファイル制御指示 334
 - 対応付けの例 334
 - mapfile オプションの初期値 336
 - 内部対応付け構造 337
 - エラーメッセージ 340
 - 警告 340
 - 致命的エラー 340
 - 提供される mapfile 341
- A. リンカーのクイックリファレンス 343
 - 概要 343
 - 静的方法 344
 - 再配置可能オブジェクトの作成 344
 - 静的実行プログラムの作成 344
 - 動的方法 344
 - 共有オブジェクトの作成 345
 - 動的実行プログラムの作成 346
- B. バージョンアップの手引き 349
 - 概要 349
 - 命名規約 350
 - 共有オブジェクトのインタフェースの定義 351
 - 共有オブジェクトのバージョンアップ 352

	既存の (バージョンアップされていない) 共有オブジェクトのバージョンアップ	353
	バージョンアップ共有オブジェクトの更新	354
	新しいシンボルの追加	354
	内部実装の変更	355
	新しいシンボルと内部実装の変更	356
	標準インタフェースへのシンボルの併合	356
C.	動的ストリングトークンによる依存関係の確立	361
	概要	361
	命令セット固有の共有オブジェクト	362
	補助検索の縮小	363
	プラットフォーム固有の共有オブジェクト	364
	関連する依存関係の配置	364
	バンドルされていない製品間の依存関係	366
	セキュリティ	368
D.	『リンカーとライブラリ』の更新箇所	371
	新しい機能および更新された機能	371
	索引	377

はじめに

Solaris™ では、アプリケーション開発者がリンカー ld(1)を使用してアプリケーションやライブラリを構築し、実行時リンカー ld.so.1(1)を使用してこれらのユーティリティを実行できる環境を提供します。多くのアプリケーション開発者にとっては、リンカーがコンパイルシステムを通じて呼び出され、実行時リンカーがアプリケーションを実行できるというこの機能は、興味深いものでしょう。このマニュアルは、製品に組み込まれた概念を、より完全に理解しようとする人を対象にしています。

お読みになる前に

このマニュアルでは、Solaris リンカーおよび実行時リンカーの操作について説明しています。共有オブジェクトの生成と使用方法に関しては、動的実行環境において重要であるため、特に重点を置いて説明しています。

対象読者

このマニュアルは、次のような、Solaris リンカーに興味を持つ、意欲的な初心者から上級ユーザーまでのプログラマを対象としています。

- 初心者は、リンカーと実行時リンカーの操作の原理を学ぶ
- 中級プログラマは、有効なカスタムライブラリの作成と使用方法を学ぶ

- 言語ツール開発者などの上級プログラマは、オブジェクトファイルの変換と生成方法を学ぶ

すべてのプログラマがこのマニュアルの最初から最後までを通読する必要があるわけではありません。

内容の紹介

第 1 章では、Solaris におけるリンクプロセスの概要と、このリリースから追加された新しい機能の紹介を記載しています。この章は、すべてのプログラマを対象としています。

第 2 章では、リンカーの機能、その接続の(「静的」および「動的」な)2つの方法、入力の有効範囲と書式、出力書式についての概要を記載しています。この章は、すべてのプログラマを対象としています。

第 3 章では、実行環境とプログラム制御によるコードおよびデータの実行時の結び付きについて記載しています。この章は、すべてのプログラマを対象としています。

第 4 章では、共有オブジェクトの定義について記載し、その機構と作成方法および使用方法について説明しています。この章は、すべてのプログラマを対象としています。

第 5 章では、動的オブジェクトによって提供されたインタフェースの管理および展開方法について記載しています。この章は、すべてのプログラマを対象としています。

第 6 章では、監視用のインタフェース、場合によっては修正用のインタフェース、リンカーと実行時リンカーの処理について記載しています。この章は、上級プログラマを対象としています。

第 7 章は、ELF ファイル用のリファレンスです。この章は、上級プログラマを対象としています。

第 8 章では、出力ファイルのレイアウトを指定する、リンカーへの対応付けファイル命令について説明しています。この章は、上級プログラマを対象としています。

付録 A は、最も一般に使用されるリンカーオプションの概要を記載しています。この付録は、すべてのプログラマを対象としています。

付録 B は、バージョンの共有オブジェクトごとの命名の規約と手引きを記載しています。この付録は、すべてのプログラマを対象としています。

付録 C では、予約動的ストリングトークンを使用して、動的依存関係を定義する方法の例を記載しています。この付録は、すべてのプログラマを対象としています。

付録 D では、これまでの各リリースでこのマニュアルに追加された新しい機能と更新事項の概要をリリースごとに記載しています。

このマニュアル全体を通して、コマンド行の例ではすべて `sh(1)` 構文を使用し、プログラミングの例はすべて C 言語で書かれています。

Sun のマニュアルの注文方法

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

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

Sun のオンラインマニュアル

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

表記上の規則

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

表 P-1 表記上の規則

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

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

コード例は次のように表示されます。

■ C シェル

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

■ C シェルのスーパーユーザー

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

■ Bourne シェルおよび Korn シェル

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

- Bourne シェルおよび Korn シェルのスーパーユーザー

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

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

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

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

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

一般規則

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

序章

概要

このマニュアルは、Solaris リンカーと実行時リンカーに加え、これらが動作するオブジェクトについて説明しています。Solaris リンカーの基本的な動作には、オブジェクトの結合、1つのオブジェクトから別のオブジェクト内にあるシンボル定義へのシンボルリファレンスの接続があります。この動作は、通常、結合と呼ばれます。

このマニュアルは次の部分に展開されます。

「リンカー」

リンカー ld(1) は、1つまたは複数の入力ファイル (再配置可能なオブジェクト、共有オブジェクト、またはアーカイブライブラリのいずれか) を連結して、1つの出力ファイルを作成します (再配置可能オブジェクト、実行可能アプリケーション、または共有オブジェクトのいずれか)。リンカーは、通常、コンパイル環境の一環として呼び出されます (cc(1) を参照)。

「実行時リンカー」

実行時リンカー ld.so.1(1) は、実行時に動的実行可能ファイルと共有オブジェクトを処理し、これらを結合して実行可能なプロセスを作成します。

注 - ld.so.1 は共有オブジェクトの特殊ケースなのでバージョンの更新が可能です。ここではバージョンナンバー 1 が使われていますが、Solaris の今後のリリースによってバージョンアップされてゆく可能性があります。

「共有オブジェクト」

共有オブジェクト(「共有ライブラリ」と呼ぶ場合もある)とは、リンク編集フェーズからの出力の書式の1つです。ただし、パワフルでフレキシブルな実行時環境を作成する上でのこれらの重要性は、それぞれの節で説明しています。

「オブジェクトファイル」

Solaris リンカーは、実行可能なリンク書式(ELF)に適合するファイルを使用して稼動します。

これらの領域は、それぞれのトピックに分割できますが、重複する部分も多いため、このマニュアルでは、各領域について説明する場合には、接続の原理と設計を同時に説明しています。

リンク編集

リンク編集では、さまざまな入力ファイルを `cc(1)`、`as(1)` または `ld(1)` から入手し、これらの入力ファイル内のデータを連結し、1つの出力ファイルの形式に変換します。リンカーでは、多数のオプションが使用できますが、作成された出力ファイルは、次の4つの基本タイプのいずれかになります。

「再配置可能オブジェクト」

入力再配置可能オブジェクトの連結。これは、後続のリンク編集フェーズ内で使用されます。

「静的実行可能ファイル」

実行可能ファイルに結合されたすべてのシンボル参照を持ち、このようにして実行可能プロセスを示す、入力再配置可能オブジェクトの連結。

「動的実行可能ファイル」

実行時リンカーが実行可能プロセスを作成する場合に必要な、入力再配置可能オブジェクトの連結。そのシンボルリファレンスは、実行時に結合される必要があり、さらに1つまたは複数の共有オブジェクト形式の依存関係を持ちます。

「共有オブジェクト」

実行時に動的実行可能ファイルと結合される可能性があるサービスを提供する入力再配置可能オブジェクトの連結。また、共有オブジェクトの中にも、他の共有オブジェクトに依存する依存関係がある場合もあります。

これらの出力ファイルと、出力ファイルを作成する場合に使用するキーリンカーオプションを、図 1-1 に示します。

「動的実行可能ファイル」と「共有オブジェクト」を、通常、2つ合わせて「動的オブジェクト」と呼びます。このマニュアルでは、この2つに焦点をあてて説明しています。

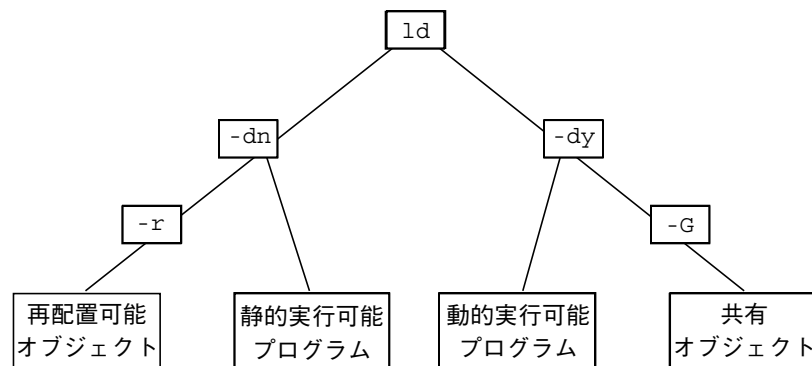


図 1-1 静的または動的リンク編集

実行時リンク

実行時リンクには、通常、過去のリンク編集から生成された1つまたは複数のオブジェクトの結び付けが組み込まれ、実行可能プロセスを生成します。リンカーによるこれらのオブジェクトの生成中に、結び付けの必要条件が検証され、該当する登録情報が各オブジェクトに追加され、実行時リンカーの対応付け、再配置、結合プロセスの完了が可能になります。

プロセスの実行中に、実行時リンカーの機能も使用可能になり、この機能を使用すると、要求に応じて追加共有オブジェクトを追加して、プロセスのアドレススペースを拡張できます。実行時リンクに組み込まれたコンポーネントのうち、最も一般的なのは、「動的実行可能ファイル」と「共有用オブジェクト」の2つです。

動的実行可能ファイル

動的実行可能ファイルとは、実行時リンカーの制御下で実行されるアプリケーションのことです。これらのアプリケーションは、通常、共有オブジェクト形式の依存関係を持ち、これらは、実行時リンカーによって配置および結合されて、実行可能プロセスが作成されます。動的実行可能ファイルは、リンカーによって生成されるデフォルトの出力ファイルになります。

共有オブジェクト

共有オブジェクトは、動的にリンクされたシステムに対し、キー構築ブロックを提供します。基本的には、共有オブジェクトは動的実行可能ファイルに類似していますが、共有オブジェクトには、仮想アドレスが割り当てられていません。

動的実行可能ファイルは、通常、1つまたは複数の共有オブジェクトに依存する依存関係を持ちます。つまり、共有オブジェクトは、動的実行可能ファイルに結合され、実行可能プロセスを作成する必要があります。共有オブジェクトは多くのアプリケーションで使用できるため、その構造上の観点からは、共有性、バージョンアップおよびパフォーマンスに直接影響します。

共有オブジェクトが使用する「環境」を参照することにより、リンカーまたは実行時リンカーのいずれかによって共有オブジェクトの処理を識別することができます。

「コンパイル環境」

共有オブジェクトは、リンカーによって処理され、動的実行可能ファイルまたは他の共有オブジェクトを生成します。共有オブジェクトは、生成される出力ファイルの依存関係になります。

「実行時環境」

共有オブジェクトは、動的実行可能ファイルとともに実行時リンカーによって処理され、実行可能プロセスを作成します。

関連情報

動的リンク

動的リンクとは、通常、実行可能プロセスを生成する際に、動的実行可能ファイルと共有オブジェクトの実行時リンクとともに、これらのオブジェクトを生成するリンク編集プロセスの一部分を受け入れる場合に使用する用語です。動的リンクを使用すると、実行時にアプリケーションを共有オブジェクトへ結合できるようにすることによって、共有オブジェクトが提供するコードを複数のアプリケーションで使用できます。

標準ライブラリのサービスからアプリケーションを切り離すことにより、動的リンクも、アプリケーションの移植性および拡張性を向上させることができます。サービスの「インタフェース」とその「実現」を切り離すことにより、システムが、アプリケーションの安定性を維持しながら展開することが可能になります。これは、「アプリケーションのバイナリインタフェース」(ABI)を提供する場合に、非常に重要な要素になります。動的リンクは、Solaris アプリケーションのコンパイルメソッドよりも優先されます。

アプリケーションのバイナリインタフェース

システムとアプリケーションコンポーネントの非同期展開を可能にするには、これらの装置間のバイナリインタフェースを定義します。Solaris リンカーは、これらのインタフェース上で稼動し、実行できるようにアプリケーションを組み合わせます。Solaris リンカーが処理するコンポーネントにはすべてバイナリインタフェースがありますが、このようなインタフェースファミリーの1つで、特にアプリケーションライターに有用なものとして、「System V アプリケーションバイナリインタフェース」があります。

「System V アプリケーションのバイナリインタフェース」、または ABI は、コンパイルされたアプリケーションプログラム用にシステムインタフェースを定義します。その目的は、標準バイナリインタフェースを「System V Interface Definition, Third Edition」を実現するシステム上のアプリケーションプログラム用に文書化することです。Solaris では、ABI 準拠アプリケーションを生成し、実行できます。SPARC™ システムでは、ABI は「SPARC Compliance Definition」(SCD) サブセットとして組み込まれています。

この後の章で説明するトピックの多くは、ABI の影響を受けています。詳細については、該当する ABI マニュアルを参照してください。

32 ビットおよび 64 ビット環境

リンカーは 32 ビットのオブジェクト上で動作し、SPARCV9 システム上では 64 ビットのオブジェクト上でも動作します。実際には、SPARC システムでは 64 ビットのリンカー (ld(1)) は 32 ビットのオブジェクトを、32 ビットのリンカーは 64 ビットのオブジェクトを生成できます。ただし、32 ビットのリンカーの場合、生成されるオブジェクトのサイズが、.bss を含めずに 2G バイトまでに制限されます。

一般に、32 ビットのリンク編集と 64 ビットのリンク編集を区別するために必要となるコマンドラインオプションはありません。リンカーは、最初に見つけた ELF オブジェクトの ELF クラスを使用して動作モードを管理します。mapfile やアーカイブライブラリのみからのリンクなどの、特別なリンク編集では、それらの入力ファイルの影響を受けず、デフォルトで 32 ビットモードになります。これらのケースでは、-64 オプションで 64 ビットのリンク編集を行うことができます。32 ビットオブジェクトと 64 ビットオブジェクトを混在させることはできません。

通常、32 ビットオブジェクト上および 64 ビットオブジェクト上のリンカーの動作は同じですが、このマニュアルでは 32 ビットオブジェクトでの動作を例として使用します。64 ビットの処理が 32 ビットの処理と異なる場合には説明します。

64 ビットアプリケーションについては、『Solaris 64 ビット 開発ガイド』を参照してください。

環境変数

リンカーでは、LD_ という文字で始まる環境変数を多数サポートしています。各環境変数は、この一般形式、またはこれに _32 か _64 の接尾辞を追加した形式が可能です。この接尾辞は、環境変数をそれぞれ 32 ビットまたは 64 ビットプロセス固有のものにし、また影響のある一般の、接尾辞の付いていない環境変数のバージョンをオーバーライドします。

このマニュアルでは、リンカーの環境変数と記述してある場合は、一般の、接尾辞の付いていない形式を使用するものとします。サポートされているすべての環境変数のリストは、ld(1) または ld.so.1(1) を参照してください。

サポートするツール

上記のオブジェクトとともに、サポートツールとライブラリもいくつか揃っています。これらのツールを使用すると、これらのオブジェクトとリンクプロセスの分析や検査が行えます。これらのツールに

は、`elfdump(1)`、`nm(1)`、`dump(1)`、`ldd(1)`、`pvs(1)`、`elf(3ELF)`、およびリンカーデバッグのサポートライブラリがあります。これらのツールについては、例を用いて詳しく説明します。

更新された機能

これまでの各リリースでこのマニュアルに追加された新しい機能と更新事項の全リストは、付録 D に記載されています。

リンカー

概要

リンク編集プロセスにより、1 つまたは複数の入力ファイルから出力ファイルが作成されます。出力ファイルの作成は、入力ファイルによって提供される入力セクションとともに、リンカーに提供されたオプションによって指示されます。

ファイルはすべて、「実行可能なリンク書式」(ELF) で表示されます。ELF 書式の詳細については、第 7 章を参照してください。ただし、ここでの概要説明では、まず、2 つの ELF 構造、「セクション」と「セグメント」について紹介する必要があります。

セクションとは、ELF ファイル内で処理できる、最も小さな、分割できない単位のことです。セグメントとは、セクションの集合で、`exec(2)` または実行時リンカー `ld.so.1(1)` によってメモリーイメージに対応付けできる (`mmap(2)` のマニュアルページを参照) 最小の一単位を表します。

ELF セクションには多くのタイプがありますが、これらはすべて、リンク編集を基準にして次の 2 つのカテゴリに分類されます。

- プログラム命令 `.text` およびその関連データ `.data` や `.bss` など、その解釈がアプリケーションそのものに対してだけ意味のある「プログラムデータ」を含むセクション
- 「リンク編集情報」`.symtab` と `.strtab` から検出されるシンボルテーブル情報など、および `.rela.text` などの再配置情報を含むセクション

基本的には、リンカーにより、「プログラムデータセクション」が連結されて出力ファイルになります。「リンク編集情報セクション」は、リンカーによって解釈されて、別のセクションに修正されるか、またはこの後処理される出力ファイルで使用される新しい出力情報セクションが生成されます。

リンカーの、次のような単純な機能の内訳については、この章で説明します。

- リンカーを通過するすべてのオプションの整合性を検証し、検査する
- 入力再配置可能オブジェクトから、同じ特性(たとえば、型、属性、名前など)のセクションを結合し、出力ファイル内に新しい出力ファイルを形成する。これらの結合されたセクションは、次に、出力セグメントへと連結できる
- リンカーは、再配置可能オブジェクトと共有オブジェクトの両方からシンボルテーブル情報を読み取り、リファレンスを定義と検証し、統一させる。さらに、通常、出力ファイル内に新しいシンボルテーブルまたはテーブルを作成する
- リンカーは、入力再配置可能オブジェクトから再配置情報を読み取り、他の入力セクションを更新してこの情報を出力ファイルに適用する。さらに、実行時リンカーが使用するために出力再配置セクションも生成される
- リンカーは、作成したすべてのセグメントを記述した「プログラムヘッダー」を生成する
- リンカーは、必要に応じて、共有オブジェクトの依存関係やシンボルの結合などの情報を実行時リンカーに提供する、動的リンク情報セクションを生成する

「セクション」と関連する「セクション」を連結して「セグメント」にするといった連結プロセスは、リンカー内のデフォルト情報を使用して実行されます。通常、ほとんどのリンク編集の場合、リンカーによって提供されるデフォルトの「セクション」と「セグメント」の処理で十分ですが、これらのデフォルトは、関連する `mapfile` を指定した `-M` オプションを使用して操作できます(詳細は、第 8 章を参照)。

リンカーの起動

リンカーは、コマンド行から直接実行できます。また、コンパイラドライバを使用して実行することができます。以下の 2 つの項では、この両方の方法を詳しく説明していますが、コンパイル環境は、通常、複雑で、コンパイラドライバだけが認識する一連の操作の変更が行われる場合もあるため、後者の方法を選択されることをお勧めします。

直接起動

リンカーを直接的に起動させる場合は、出力を作成するために必要なすべてのオブジェクトファイルとライブラリを提供する必要があります。リンカーは、出力の作成に使用するつもりオブジェクトモジュールまたはライブラリに関して、仮説を立てることをしません。たとえば、次のコマンドを発行するとリンカーは、入力ファイル `test.o` だけを使用して、`a.out` という動的実行可能ファイルを作成します。

```
$ ld test.o
```

`a.out` を有用な実行可能ファイルにするためには、これに初期設定および終了処理コードを組み込む必要があります。このコードは、言語またはオペレーティングシステム固有のもので、通常、コンパイラドライバによって提供されるファイルを通じて提供されます。

また、自分専用の初期設定および終了コードも指定できます。このコードは、実行時リンカーにより正確に認識され、使用できるようにするために、適切に暗号化およびラベル付けを行う必要があります。この暗号化とラベル付けも、コンパイラドライバによって提供されたファイルを通じて提供されます。

実際問題として、実行可能ファイルや共有オブジェクトなどの実行時オブジェクトを作成する場合は、コンパイラドライバを使用してリンカーを起動することをお勧めします。リンカーを直接起動をお勧めするのは、`-r` オプションを使用して、中間再配置可能オブジェクトを作成する場合だけです。

コンパイラドライバを使用する

リンカーを使用する従来の方法は、言語固有のコンパイラドライバを使用する方法です。入力ファイルとともに、`cc(1)`、`f77(1)` などのコンパイラドライバを指定します。すると、コンパイラドライバは、追加ファイルとデフォルトライブラリを追加して、リンク編集を完了させます。これらの追加ファイルは、たとえば、以下のようにコンパイルの呼出しを拡張することによって参照できます。

```
$ cc -# -o prog main.o
/usr/ccs/bin/ld -dy /opt/COMPILER/crti.o /opt/COMPILER/crt1.o \
/usr/ccs/lib/values-Xt.o -o prog main.o \
-YP,/opt/COMPILER/lib:/usr/ccs/lib:/usr/lib -Qy -lc \
/opt/COMPILER/crtn.o
```

注 - この例は、コンパイラドライバによって組み込まれた実際のファイルの例ですが、リンカー起動の表示に使用されるメカニズムによって異なる場合があります。

リンカーオプションの指定

リンカーに対するオプションは、通常、コンパイラドライバのコマンド行を通じて渡されます。コンパイラとリンカーオプションは、ほとんど重複する部分はありません。重複が発生した場合には、通常、特定のオプションをリンカーに渡すことを許可するコマンド行構文がコンパイラドライバによって提供されますが、この代わりに、LD_OPTIONS 環境変数を設定して、リンカーにオプションを提供することもできます。次に例を示します。

```
$ LD_OPTIONS="-R /home/me/libs -L /home/me/libs" cc -o prog main.c -lfoo
```

ここでは、-R および -L オプションがリンカーによって変換され、コンパイラドライバから受信したコマンド行オプションに付加されます。

リンカーは、オプションリスト全体を構文解析し、無効なオプションまたは関連する引数が無効であるオプションを検索します。これらの無効であるオプションのどちらかが検出された場合は、該当するエラーメッセージが生成され、さらにこのエラーが深刻なものである場合には、リンカーは自動的に終了します。次に例を示します。

```
$ ld -X -z sillydefs main.o
ld:illegal option -- X
ld: fatal: option -z has illegal argument `sillydefs'
```

この例では、リンカーの検査により、不当なオプション -X が認識され、-z オプションに不当な引数が検出されました。1つの引数を必要とするオプションが、誤って2回指定されている場合には、リンカーは該当する警告を表示しますが、リンク編集は継続します。次に例を示します。

```
$ ld -e foo ..... -e bar main.o
ld: warning: option -e appears more than once, first setting taken
```

また、リンカーはオプションリストを調べて重大な不一致も検出します。次に例を示します。

```
$ ld -dy -a main.o
ld: fatal: option -dy and -a are incompatible
```

すべてのオプションを処理しても、エラー状態が検出されなかった場合は、次にリンカーは、入力ファイルの処理を行います。

通常使用されるリンカーオプションについては、付録 A を参照してください。また、全リンカーオプションの詳細については、ld(1) のマニュアルページを参照してください。

入力ファイルの処理

リンカーは、入力ファイルをコマンド行上に表示された順番に読み取ります。各ファイルは、オープンされ、その ELF ファイルタイプを判別するために検査され、どのように処理する必要があるかが決定されます。リンク編集に必要な入力に適用するファイルタイプは、リンク編集の結合モード、「静的」または「動的」のいずれかによって決定されます。

「静的」方法では、リンカーが入力ファイルとして受け入れるのは、再配置可能オプションまたはアーカイブライブラリだけです。「動的」方法では、リンカーは、共有オブジェクトも受け入れます。

再配置可能オブジェクトは、リンク編集プロセスへの最も基本的な入力ファイルタイプを示しています。これらのファイル内の「プログラムデータ」のセクションは、生成される出力ファイルイメージに結合されます。「リンク編集情報」のセクションは、今後の利用のために構成されますが、出力ファイルイメージには組み込まれません。それは、これを配置する場所として、新しいセクションが生成されるからです。シンボルは、その検査および解析を考慮して、最終的に出力イメージ内に 1 つまたは複数のシンボルテーブルを作成できるように、特定の内部シンボルテーブル内に集められます。

入力ファイルは、リンク編集のコマンド行上に直接指定できますが、アーカイブライブラリと共有オブジェクトは、通常、`-l` オプションを使用して指定します (このメカニズムについて、およびそれが 2 種類のリンクモードにどのように関係するかについては、32ページの「追加ライブラリとのリンク」を参照)。ただし、通常、共有オブジェクトが共有ライブラリと呼ばれ、さらに、これら 2 つのオブジェクトを同じオプションを使用して指定したとしても、共有オブジェクトとアーカイブライブラリは全く別のものです。次の 2 つの項で、この違いについて説明します。

アーカイブ処理

アーカイブは、`ar` (1) を使用して構築され、通常、アーカイブシンボルテーブルとともに再配置可能オブジェクトの集合で構成されます。このシンボルテーブルにより、これらの定義の提供するオブジェクトとシンボル定義との関係がわかります。デフォルトでは、リンカーを使用すると、アーカイブ構成要素を選択して抽出できます。リンカーがアーカイブを読み取る場合は、結合処理を完了させるために必要なアーカイブから、オブジェクトだけを選択するように作成された内部シンボルテーブル内の情報を使用します。さらに、1 つのアーカイブのすべての構成要素を明示的に抽出することも可能です。

次のような場合に、リンカーはアーカイブから再配置可能オブジェクトを抽出します。

- アーカイブに、現在リンカーの内部シンボルテーブル内に保持されている、シンボルリファレンス (「未定義」シンボルと呼ぶ場合もある) を満たすシンボル定義が入っている場合
- アーカイブに、現在リンカーの内部シンボルテーブル内に保持されている、「未確認」シンボル定義を満たす「データ」シンボル定義が入っている場合。この例としては、FORTRAN COMMON ブロック定義があります。この定義により、同じ DATA シンボルを定義する再配置可能オブジェクトが抽出される
- リンカーの `-z allextract` が実行された場合。このオプションにより、選択式のアーカイブ抽出は中止され、処理中のアーカイブからアーカイブ構成要素がすべて抽出される

選択式アーカイブ抽出においては、ウィークシンボルリファレンスでは、`-z weakextract` オプションが発効されていない限り、アーカイブからの抽出は実行されません。ウィークシンボルについては、42ページの「単純な解析」の項で詳しく説明しています。

注・オプション `-z weakextract`、`-z alleextract`、および `-z defaultextract` により、複数のアーカイブ間でアーカイブメカニズムを切り替えることができます。

選択式アーカイブ抽出の場合、リンカーは、リンカーの内部シンボルテーブル内に蓄積されたシンボル情報を満たすために必要な、再配置可能オブジェクトを抽出するアーカイブを通る複数のパスを作成します。リンカーが、再配置可能オブジェクトを抽出せずに、アーカイブを通る完全なパスを作成すると、リンカーは次の入力ファイルの処理に移ります。

アーカイブに遭遇した時点で、必要な再配置可能オブジェクトだけをアーカイブから抽出することにより、メカニズムは入力ファイルリスト内のアーカイブの位置が重要であることを意味しています (詳細は、34ページの「コマンド行上のアーカイブの位置」を参照)。

注・リンカーは、アーカイブを通る複数のパスを作成し、シンボルを解析しますが、このメカニズムは、ランダムに構成された再配置可能オブジェクトが組み込まれた大容量のアーカイブの場合には、非常にコストがかかります。このような場合は、`lorder(1)` や `tsort(1)` などのツールを使用してアーカイブ内の再配置可能オブジェクトを配列し、リンカーが実行しなければならないパスの数を削減することができます。

共有オブジェクトの処理

共有オブジェクトは、分割不可能な、1つまたは複数の入力ファイルの以前の編集によって生成された総体単位です。リンカーが共有オブジェクトを処理すると、共有オブジェクトの全内容は、その結果作成された出力ファイルイメージの「論理的」な部分になります。共有オブジェクトは、リンク編集中には物理的にコピーされないため、プロセスが実行されるまで実際には組み込まれません。この論理的な組み込みは、リンク編集プロセスにとって共有オブジェクト内に定義された「すべての」シンボルエントリが利用可能になることを意味しています。

リンカーは、共有オブジェクトの「プログラムデータ」セクションと、ほとんどの「リンク編集情報」セクションを使用しません。そのため、これらのセクションは、共有オブジェクトが結合された実行可能プロセスに生成されるときに、実行時リンカーによって変換されます。ただし、共有オブジェクトのエントリが記憶さ

れ、情報は出力ファイルイメージ内に格納されて、このオブジェクトには依存関係があり、実行時に使用可能にする必要があるかどうかが表示されます。

デフォルトでは、リンク編集の一部として指定された共有オブジェクトはすべて、作成されるオブジェクト内に依存関係として記録されます。この記録は、そのオブジェクトが、共有オブジェクトによって提供された実際のリファレンスシンボルを生成するかどうかに関係なく実行されます。実行時リンクのオーバーヘッドを最小限にするには、作成されたオブジェクトからシンボルリファレンスを解析するために必要な依存関係だけを、リンク編集の一部として指定します。リンカーのデバッグ機能および `-u` オプションを指定した `ldd(1)` を使用して、使用されない依存関係を確認することができます。または、リンカーの `-z ignore` オプションを使用すると、使用しない共有オブジェクトの依存関係の記録を抑制できます。

共有オブジェクトに、他の共有オブジェクトに対する依存関係がある場合、この依存関係も処理されます。この処理は、すべてのコマンド行入力ファイルの処理が終了した後で実行されます。このような共有オブジェクトは、シンボル解析処理を完了させるために使用されますが、生成される出力ファイルイメージ内に、その名前は依存関係として記録されません。

リンク編集コマンド行上の共有オブジェクトの位置は、アーカイブ処理のための位置に比べるとそれほど重要ではありませんが、大域な効力を持たせることができます。複数のシンボルに同じ名前を付けると、再配置可能オブジェクトと共有オブジェクト間や複数の共有オブジェクト間に出現させることができます (詳細は、41 ページの「シンボル解析」を参照)。

リンカーによって処理される共有オブジェクトの順序は、出力ファイルイメージ内に格納された従属情報に保持されます。実行時リンカーがこの情報を読み取るため、指定された共有オブジェクトは同じ順序で読み込まれます。そのため、リンカーと実行時リンカーは、多重に定義された一連のシンボルのうち、1つのシンボルの最初のエントリを選択します。

注 - 複数のシンボル定義と、他のシンボル用に1つのシンボル定義の割り込みを説明した情報は、`-m` オプションを使用して生成されたロードマップ出力内に報告されます。

追加ライブラリとのリンク

通常、コンパイラドライバによって、適切なライブラリがリンカーに指定されているかどうかを確認されますが、ほとんどの場合、自分独自のライブラリを指定する

ことが必要です。共有オブジェクトとアーカイブは、リンカーに必要な入力ファイルに明示的に命名することによって指定できますが、より一般的で柔軟性のある方法として、リンカーの `-l` オプションを使用する方法があります。

ライブラリの命名規約

規則により、通常、共有オブジェクトは接頭辞 `lib` と接尾辞 `.so` で指定され、アーカイブは接頭辞 `lib` と接尾辞 `.a` で指定されます。たとえば、`libc.so` とは、コンパイル環境で使用可能になった標準 C ライブラリの共有オブジェクトバージョンで、`libc.a` とは、そのアーカイブバージョンです。

これらの規則は、リンカーの `-l` オプションによって認識されます。このオプションは、通常、追加ライブラリをリンク編集に供給する場合に使用します。次の例では、リンカーに `libfoo.so` を検索するように指示し、これが検出できない場合には、`libfoo.a` を検索してから次の検索ディレクトリに移動するように指示しています。

```
$ cc -o prog file1.c file2.c -lfoo
```

注 - 命名規約には、共有オブジェクトの「コンパイル」環境での使用に関するものと、共有オブジェクトの実行時環境での使用に関するものがあります。コンパイル環境では、単に `.so` 接尾辞を使用するのに対し、実行時環境では、通常、追加のバージョン番号を指定した接尾辞を使用します。詳細については、108ページの「命名規約」および 158ページの「バージョンアップファイル名の管理」を参照してください。

動的モードでリンク編集を行う場合、共有オブジェクトとアーカイブとを組み合わせたものへのリンクを選択できます。静的モードでリンク編集を行う場合、入力を受け入れるのはアーカイブライブラリだけです。

動的モードで、ライブラリの検索を可能にする `-l` オプションを使用すると、リンカーは、まず、指定されたディレクトリ内で、指定された名前と一致する共有オブジェクトを検索します。一致するものが見つからない場合、リンカーは、次に同じディレクトリ内でアーカイブライブラリを検索します。静的モードで `-l` オプションを使用する場合は、アーカイブライブラリだけが検索されます。

共有オブジェクトとアーカイブとの混合体へのリンク

動的モードでのライブラリ検索メカニズムでは、指定されたディレクトリを調べて共有オブジェクトを検索し、次にアーカイブライブラリを検索しますが、`-B` オプションを使用すると必要な検索タイプのより適切なコントロールを保存できます。

コマンド行上に `-Bdynamic` と `-Bstatic` オプションを必要な回数だけ指定することによって、ライブラリ検索は共有オブジェクトまたはアーカイブをそれぞれ切り替えることができます。たとえば、アーカイブ `libfoo.a` と共有オブジェクト `libbar.so` とリンクするには、次のコマンドを発行します。

```
$ cc -o prog main.o file1.c -Bstatic -lfoo -Bdynamic -lbar
```

キーワード `-Bstatic` と `-Bdynamic` は、正確には対称ではありません。`-Bstatic` を指定すると、リンカーは、次の `-Bdynamic` の発生まで入力として共有オブジェクトを受け入れませんが、`-Bdynamic` を指定すると、リンカーは、指定されたディレクトリ内で、まず最初に共有オブジェクトを検索し、次にアーカイブを検索します。

上記の例をより正確に説明すると、リンカーは、最初に `libfoo.a` を検索し、次に `libbar.so` を検索します。そしてこれに失敗すると `libbar.a` を検索します。最後に、`libc.so` を検索し、これに失敗すると `libc.a` を検索します。

コマンド行上のアーカイブの位置

コマンド行上のアーカイブの位置は、作成される出力ファイルに影響を及ぼします。リンカーがアーカイブを検索する唯一の目的は、以前に参照したことのある定義されていない仮の外部リファレンスを解析することです。この検索が完了し、必要な再配置可能オブジェクトが抽出されると、このアーカイブは、コマンド行上のアーカイブに続く入力ファイルから入手した新しいシンボルの解析には使用できません。たとえば、次のコマンドでは、`file1.c` から入手したことのあるシンボルリファレンスを解析するためだけに、`libfoo.a` を検索するように、リンカーに指示しています。

```
$ cc -o prog file1.c -Bstatic -lfoo file2.c file3.c -Bdynamic
```

`libfoo.a` は、`file2.c` または `file3.c` のシンボルリファレンスを解析するためには、使用できません。

注・原則として、コマンド行の最後にアーカイブを指定するのが最善の方法です。ただし、最後にアーカイブを配置するには、複数の定義を重複させる必要がある場合は除きます。

リンカーが検索するディレクトリ

ここまでの例はすべて、リンカーが、コマンド行上にリストされたライブラリを検索する場所を認識していることを前提としています。デフォルトでは、32 ビットオブジェクトをリンクする場合、リンカーがライブラリを検索するディレクトリとして認識しているのは、2つの標準的なディレクトリ `/usr/ccs/lib` と `/usr/lib` です。64 ビットオブジェクトのリンクの場合は、1つの標準的なディレクトリ `/usr/lib/64` のみを使用されます。これ以外のディレクトリを検索させたい場合には、リンカーの検索パスに明示的に付加する必要があります。

リンカー検索パスを変更するには、コマンド行オプションを使用するか、環境変数を使用するという2種類の方法があります。

コマンド行オプションの使用

`-L` オプションを使用すると、ライブラリ検索に新しいパス名を追加できます。このオプションは、コマンド行上で遭遇したその地点で、検索パスに影響を与えます。たとえば、次のコマンドは、`path1` (次に `/usr/ccs/lib` と `/usr/lib`) を検索し、`libfoo` を検出しますが、`libbar` を検出する場合は、`path1` と `path2` (次に `/usr/ccs/lib` と `/usr/lib`) を検索します。

```
$ cc -o prog main.o -Lpath1 file1.c -lfoo file2.c -Lpath2 -lbar
```

`-L` オプションを使用して定義されたパス名は、リンカー専用で、実行時リンカーが使用するために作成される出力ファイルイメージ内には記録されません。

注・カレントディレクトリ内のライブラリの検索にリンカーを使用する場合は、`-L` を指定する必要があります。ピリオド (`.`) を使用して、カレントディレクトリを示すことができます。

`-Y` オプションを使用すると、リンカーが検索するデフォルトのディレクトリを変更できます。このオプションに指定する引数は、ディレクトリのリストをコロンで区

切った書式で示します。たとえば、次のコマンドは、ディレクトリ
/opt/COMPILER/lib と /home/me/lib 内だけを調べて libfoo を検索します。

```
$ cc -o prog main.c -YP,/opt/COMPILER/lib:/home/me/lib -lfoo
```

-Y オプションを使用して指定したディレクトリは、-L オプションを使用して補足
できます。

環境変数の使用

コロンで区切られたディレクトリリストをとる環境変数 LD_LIBRARY_PATH を使用
しても、リンカーのライブラリ検索パスを付加できます。この最も一般的な書式
LD_LIBRARY_PATH では、セミコロンで区切られた2つのディレクトリリストをと
ります。最初のリストは、コマンド行上に指定されたリストよりも前に検索され、2
番目のリストはコマンド行上のリストよりも後に検索されます。

ここでは、LD_LIBRARY_PATH の設定と、いくつかの -L オプションを指定したリ
ンカーの呼出しを組み合わせています。

```
$ LD_LIBRARY_PATH=dir1:dir2;dir3  
$ export LD_LIBRARY_PATH  
$ cc -o prog main.c -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

有効な検索パスは、次のとおりです。

```
dir1:dir2:path1:path2... pathn:dir3:/usr/ccs/lib:/usr/lib
```

LD_LIBRARY_PATH 定義の一部にセミコロンが指定されていない場合は、指定された
ディレクトリリストは、-L オプションの後で解釈されます。次に例を示します。

```
$ LD_LIBRARY_PATH=dir1:dir2  
$ export LD_LIBRARY_PATH  
$ cc -o prog main.c -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

ここでは、有効な検索パスは次のとおりです。

```
path1:path2... pathn:dir1:dir2:/usr/ccs/lib:/usr/lib
```

注 - この環境変数は、実行時リンカーの検索パスを拡張する場合にも使用できます (詳細は、70ページの「実行時リンカーによって検索されるディレクトリ」を参照)。この環境変数がリンカーに影響しないようにするには、`-i` オプションを使用します。

実行時リンカーが検索するディレクトリ

デフォルトでは、実行時リンカーがライブラリを検出する場所として認識する標準的な場所は1つだけで、32ビットオブジェクトを処理する場合は`/usr/lib`、64ビットオブジェクトを処理する場合は`/usr/lib/64`です。この他のディレクトリを検索する場合は、実行時リンカーの検索パスに明示的に追加する必要があります。

動的な実行可能ファイルまたは共有オブジェクトは、付加された共有オブジェクトとリンクされ、これらの共有オブジェクトは、実行時リンカーによるプロセスの実行中に再び配置される必要がある従属物として記録されます。リンク編集中には、1つまたは複数のパス名を出力ファイル内に記録できます。これらのパス名は、実行時リンカーが共有オブジェクトの従属物を検索する場合に使用されます。この記録されたパス名は、「実行パス」と呼ばれます。

`-z nodefaultlib` オプションを使用してオブジェクトを作成すると、実行時に、標準的な場所 (`/usr/lib` あるいは `/usr/lib/64`) を検索しないようにできます。このオプションを使用すると、オブジェクトのすべての依存関係はその実行パスを使用して検索されます。多くの場合はこのオプションを使用せず、ユーザーが実行時リンカーのライブラリ検索パスをどのように修正しても、最後の構成要素は必ず、32ビットオブジェクトの場合は`/usr/lib`、64ビットオブジェクトの場合は`/usr/lib/64`です。

注 - デフォルトの検索パスは、実行時構成ファイルを使用して管理できます (73ページの「デフォルトの検索パスの設定」を参照)。ただし、オブジェクト作成者はこのファイルの存在に頼らず、実行パスあるいは標準的なシステムのデフォルト設定のみを使用して、オブジェクトがその依存関係を検索できるようにしてください。

コロンの区切られたディレクトリリストを伴う、`-R` オプションを使用すると、動的実行可能ファイルまたは共有ライブラリ内に実行パスを記録できます。次に例を示します。

```
$ cc -o prog main.c -R/home/me/lib:/home/you/lib -Lpath1 \  
-Lpath2 file1.c file2.c -lfoo -lbar
```

上記の例では、動的実行可能ファイル prog 内に、実行パス /home/me/lib:/home/you/lib が記録されます。実行時リンカーは、共有オブジェクトの依存関係を配置する場合に、これらのパスを使用してから、デフォルトのロケーション /usr/lib を使用します。この場合、この実行パスは、libfoo.so.1 と libbar.so.1 の配置に使用されます。

リンカーは、複数の -R オプションを受け取り、コロンで区切られたこれらの指定内容をそれぞれ結合します。そのため、上記の例は、次のようにも示すこともできます。

```
$ cc -o prog main.c -R/home/me/lib -Lpath1 -R/home/you/lib \  
-Lpath2 file1.c file2.c -lfoo -lbar
```

さまざまな場所にインストールされる可能性のあるオブジェクトについては、\$ORIGIN 動的のストリングトークンを使用して、柔軟に実行パスを記録できます。364ページの「関連する依存関係の配置」を参照してください。

注 - 以前は、-R オプションの指定に代わるものとして、環境変数 LD_RUN_PATH を設定してリンカーがこれを使用できるようにする方法がありました。LD_RUN_PATH および -R の適用範囲と機能は全く同じですが、この両方を指定した場合は、-R によって LD_RUN_PATH は上書きされます。

初期設定および終了セクション

動的オブジェクトは、実行時の初期設定と終了プロセスのためのコードを提供することができます。このコードは、関数ポインタの配列、または単一コードブロックのうちいずれか1つの方法で使用されます。これらの各セクションタイプは、入力再配置可能オブジェクトの同類のセクションの連結から構築されます。

.preinit_array、.init_array、および.fini_array セクションは、それぞれ実行時の初期設定前、初期設定、および終了関数の配列を提供します。動的オブジェクトを作成する際、実行時リンカーによって呼び出されるように、リンカーはこれらの配列を .dynamic タグの DT_PREINIT_ARRAY と

DT_PREINIT_ARRAYSZ、DT_INIT_ARRAY と DT_INIT_ARRAYSZ、DT_FINI_ARRAY と DT_FINI_ARRAYSZ でそれぞれ識別します。初期設定前の配列は、動的実行可能ファイルにのみ適用可能です。詳細は、84ページの「初期設定および終了ルーチン」を参照してください。

.init と .fini セクションは、それぞれ実行時の初期設定と終了時のコードブロックを提供します。ただし、通常コンパイラドライバからも、入力ファイルリストの冒頭部分と末尾に付加するファイルの一部として、.init と .fini セクションを指定できます。これらのファイルには、.init および .fini コードを、それぞれ予約シンボル名 `_init` と `_fini` により識別される個別の機能にカプセル化する効果があります。動的オブジェクトを作成する際、実行時リンカーによって呼び出されるように、リンカーはこれらのシンボルをそれぞれ `.dynamic` タグの DT_INIT および DT_FINI で識別します。詳細は、84ページの「初期設定および終了ルーチン」を参照してください。

初期設定および終了関数の登録は、`-z initarray` および `-z finiarrray` オプションを使用してリンカーで直接実行できます。たとえば、次のコマンドの結果、関数 `foo` のアドレスが `.initarray` 要素に配置され、関数 `bar` のアドレスが `.finiarrray` 要素に配置されます。

```
$ cat main.c
#include <stdio.h>

void foo()
{
    (void) printf("initializing: foo()\n");
}

void bar()
{
    (void) printf("finalizing: bar()\n");
}

main()
{
    (void) printf("main()\n");
    return (0);
}

$ cc -o main -zinitarray=foo -zfiniarrray=bar main.c
$ main
initializing: foo()
main()
finalizing: bar()
```

初期設定および終了セクションの作成は、アセンブラを使用して直接実行できます。あるいは、コンパイラの中にはその宣言を単純化するための特別なプリミティ

ブを提供しているものもあります。たとえば、次の #pragmas を含んだ同じ例の結果、関数 foo に対する「呼び出し」が .init セクション内に配置され、関数 bar に対する「呼び出し」が .fini セクション内に配置されます。

```
$ cat main.c
#include <stdio.h>

#pragma init (foo)
#pragma fini (bar)

.....
$ cc -o main main.c
$ main
initializing: foo()
main()
finalizing: bar()
```

共有オブジェクトとアーカイブラリ両方に組み込むことができる、初期設定コードと終了コードの設計をするときには注意が必要です。このコードが、アーカイブラリ内のいくつかの再配置可能オブジェクト全体に分散された場合、このアーカイブを使用したアプリケーションのリンク編集は、オブジェクトの一部しか抽出しない可能性があるため、初期設定コードと終了コードの一部だけしか抽出できません。そして実行時に、コードのこの部分だけが実行されます。共有オブジェクトがアプリケーションの依存関係の一つとして読み込まれている場合は、共有オブジェクトに対して構築された同じアプリケーションには、実行時に実施される、累積した初期設定コードと終了コードがすべて入っています。

実行時にプロセス内で初期設定および終了コードをどのような順序で実行すべきかを判断することは、依存関係の分析を伴う複雑な問題を含んでいます。初期設定および終了コードが外部の大域シンボルを参照していると、この判断はさらに困難なものになり、コードは循環的な依存関係に陥ることがあります。最も柔軟性が高い初期設定および終了コードは、常駐オブジェクト内の要素だけを参照するコードです。

また、初期設定コードが、`dldump(3DL)` を使用してメモリーをダンプできる動的オブジェクトとともに組み込まれている場合は、データの初期設定は独立させることをお勧めします。

シンボルの処理

入力ファイルの処理中に、入力再配置可能オブジェクトからローカルシンボルが出力ファイルイメージに渡されます。大域シンボルはすべて、リンカーの内部に蓄積されます。この内部のシンボルテーブルは、処理される新しい大域シンボルのエントリごとに検索され、過去の入力ファイルで、同じ名前のシンボルに遭遇したことがあるかどうか判別されます。同じ名前のシンボルに遭遇したことがある場合には、2つのシンボルのうちどちらを保持するかを決定するために、シンボル解析プロセスが呼び出されます。

入力ファイル処理の完了時に、シンボル解析中に深刻なエラー状態が発生しなかった場合は、リンカーは、リンク編集の失敗を招く、結合されていないシンボルリファレンス (未定義シンボル) が残っていないかどうか判別します。

最後に、リンカーの内部シンボルテーブルが、作成されるイメージのシンボルテーブルに追加されます。

次の項では、シンボル解析と未定義シンボルの処理について詳しく説明します。

シンボル解析

シンボル解析は、簡単に直感的に分かるものから、複雑で当惑するようなものまで、すべての範囲を実行します。解析は、リンカーによって自動的に実行されるか、警告診断プログラムを伴って表示されるか、またはその結果、重大なエラー状態になります。

2つのシンボルの解析は、シンボルの属性、シンボルを入手したファイルのタイプおよび生成されるファイルのタイプによって異なります。シンボルの属性についての詳細は、237ページの「シンボルテーブル」を参照してください。ただし、以下に説明するシンボルタイプは、識別する価値のある3つの基本的なシンボルタイプです。

未定義シンボル

このシンボルは、ファイル内で参照されましたが、記憶領域アドレスが割り当てられていません。

一時的シンボル

このシンボルは、ファイル内で作成されましたが、まだサイズが決められていないか、または記憶領域内に割り当てられていません。このようなシンボルは、初期化されていない C シンボル、または FORTRAN COMMON ブロックとしてファイル内に表示されます。

定義シンボル

このシンボルは、作成されてからファイル内の記憶領域アドレスおよびスペースが割り当てられています。

この最も単純な形式においては、シンボル解析には優先度関係の使用が伴いません。つまり、定義シンボルは一時的シンボルよりも優先され、次に一時的シンボルは、未定義シンボルよりも優先されます。

次の C コードの例では、これらのシンボルタイプがどのようにして生成されるかを示しています (未定義シンボルには接頭辞 `u_` が、一時的シンボルには接頭辞 `t_` が、定義シンボルには接頭辞 `d_` がそれぞれ付いています)。

```
$ cat main.c
extern int      u_bar;
extern int      u_foo();

int             t_bar;
int             d_bar = 1;

d_foo()
{
    return (u_foo(u_bar, t_bar, d_bar));
}
$ cc -o main.o -c main.c
$ nm -x main.o
```

[Index]	Value	Size	Type	Bind	Other	Shndx	Name
[8]	0x00000000	0x00000000	NOTY	GLOB	0x0	UNDEF	u_foo
[9]	0x00000000	0x00000040	FUNC	GLOB	0x0	2	d_foo
[10]	0x00000004	0x00000004	OBJT	GLOB	0x0	COMMON	t_bar
[11]	0x00000000	0x00000000	NOTY	GLOB	0x0	UNDEF	u_bar
[12]	0x00000000	0x00000004	OBJT	GLOB	0x0	3	d_bar

単純な解析

このシンボル解析は、群をぬいて最もよく使用されるもので、類似した特徴を持つ 2 つのシンボルが削除された場合や、1 つのシンボルが他のシンボルよりも優先される場合に実行されます。このシンボル解析は、リンカーによって自動的に実行されま

す。たとえば、同じ結びつきを伴う複数のシンボルでは、あるファイルから未定義シンボルへのリファレンスは結合されるか、または他のファイルからの定義シンボルまたは一時的シンボル定義によって満たされます。あるいは、あるファイルからの一時的シンボル定義は、他のファイルからの定義シンボルの定義に結合されます。

解析を受けるシンボルは、大域結合またはウィーク結合されます。ウィーク結合の方が、大域結合よりも優先度が低くなります。そのため、異なる結びつきを伴うシンボルは、基本規則のわずかな変更に従って解析されます。しかし、まず初めに、シンボルがどのように作成されるかを紹介します。

ウィークシンボルは通常、個別にあるいは大域シンボルの別名として、コンパイラによって定義されます。この機構では、`pragma` 定義を使用します。

```
$ cat main.c
#pragma weak    bar
#pragma weak    foo = _foo

int             bar = 1;

_foo()
{
    return (bar);
}
$ cc -o main.o -c main.c
$ nm -x main.o

[Index]  Value      Size      Type Bind  Other Shndx  Name
.....
[7]      |0x00000000|0x00000004|OBJT |WEAK |0x0 | 3      |bar
[8]      |0x00000000|0x00000028|FUNC |WEAK |0x0 | 2      |foo
[9]      |0x00000000|0x00000028|FUNC |GLOB |0x0 | 2      |_foo
```

ウィークの別名 `foo` に、大域シンボル `_foo` と同じ属性が割り当てられていることに注意してください。この関係は、リンカーによって保持され、その結果、シンボルには出力イメージ内の同じ値が割り当てられます。シンボル解析においては、ウィーク定義シンボルは、同じ名前の大域定義によって自動的に上書きされます。

単純なシンボル解析のこの他の形式は、再配置可能オブジェクトと共有オブジェクト間、または複数の共有オブジェクト間に発生し、割り込み (*interposition*) と呼ばれます。このような場合、シンボルが複数回定義されている場合、リンカーにより、再配置可能オブジェクト、または複数の共有オブジェクト間の最初の定義が自動的に採用されます。再配置可能オブジェクトの定義、または最初の共有オブジェクトの定義は、他のすべての定義上に割り込みを行うといわれます。この割り込みを使用すると、1つの共有オブジェクト、動的実行可能ファイル、または他の共有オブジェクトによって提供された機能を上書きできます。

ウィークシンボルと割り込みを組み合わせることにより、非常に有用なプログラミングテクニックを使用できます。たとえば、標準 C ライブラリは、再定義可能ないくつかのサービスを提供していますが、ANSI C は、システム上になければならない一連の標準サービスを定義し、厳密に適合するプログラム内に置き換えることはできません。

たとえば、関数 `fread(3C)` は、ANSI C ライブラリの関数ですが、関数 `read(2)` は、ANSI C ライブラリの関数ではありません。適合する ANSI C プログラムは、`read(2)` を再定義でき、予測できる方法で `fread(3C)` を使用できなければなりません。

ここでの問題は、`read(2)` は、標準 C ライブラリ内に `fread(3C)` を実装する基盤になるため、`read(2)` を再定義するプログラムは、`fread(3C)` の実装を間違える可能性があるのではないかと思います。このような間違いをなくすためには、ANSI C は、実装には、そこに予約されていない名前は使用できないように指定します。さらに、以下に示す `pragma` 指示語を使用することにより、この予約名を定義できます。

```
#pragma weak read = _read
```

また、この予約名から、関数 `read(2)` の別名が生成されます。こうすることにより、ユーザーは、`fread(3C)` の実装に妥協することなく、自分専用の `read()` 関数を自由に定義できます。

標準 C ライブラリの、共有オブジェクトまたはアーカイブバージョンのどちらかにリンクしている場合でも、ユーザーによる `read()` の再定義に対するリンカーからのクレームはありません。前者の場合には、割り込みによって方法が決められ、後者の場合には、`read(2)` の C ライブラリの定義をウィークにすることにより、自動的に上書き可能になります。

リンカーの `-m` オプションを使用すると、割り込みされるすべてのシンボルリファレンスのリストが、セクションの読み込みアドレス情報とともに標準出力に書き込まれます。

複雑な解析

複雑な解析は、同じ名前を持つ 2 つのシンボルが、異なる属性とともに検出された場合に発生します。この場合、リンカーは最も適切なシンボルを選択し、そのシンボル、対立する属性、およびそのシンボル定義を取り出したファイルの ID を示す警告メッセージを生成します。次に例を示します。

```

$ cat foo.c
int array[1];

$ cat bar.c
int array[2] = { 1, 2 };

$ cc -dn -r -o temp.o foo.c bar.c
ld: warning: symbol `array' has differing sizes:
      (file foo.o value=0x4; file bar.o value=0x8);
      bar.o definition taken

```

ここで、データ項目の配列の定義が指定された2つのファイルでは、サイズの必要条件が異なります。シンボル配列の必要条件が異なる場合には、同様の診断プログラムが作成されます。この2つのケースの場合、リンカーの `-t` オプションを使用すると、診断プログラムを抑制できます。

この他の属性形式の違いに、シンボルの「タイプ」があります。次に例を示します。

```

$ cat foo.c
bar()
{
    return (0);
}
$ cc -o libfoo.so -G -K pic foo.c
$ cat main.c
int    bar = 1;

main()
{
    return (bar);
}
$ cc -o main main.c -L. -lfoo
ld: warning: symbol `bar' has differing types:
      (file main.o type=OBJT; file ./libfoo.so type=FUNC);
      main.o definition taken

```

ここで、シンボル `bar()` は、データ項目と関数の両方として定義されています。

注 - このコンテキスト内の型とは、ELF で表されたシンボルタイプのことです。このシンボルタイプは、まったく手を加えられていない場合を除いて、プログラミング言語が使用するデータ型には関連していません。

このような場合には、再配置可能オブジェクトと共有オブジェクト間で解析が発生したときには、再配置可能オブジェクトの定義がとられます。または、2つの共有オブジェクト間で解析が発生した場合には、最初の共有オブジェクトの定義がと

られます。このような解析が異なる結合間(「ウィーク」または「大域」)で発生すると、警告メッセージも同時に作成されます。

リンカーの `-c` オプションを使用しても、シンボルタイプ間の不一致は抑制できません。

重大な解析

解析できないシンボルの重複によって、重大なエラー状態が発生します。この場合、シンボルを入手したファイルの名前と同時に、そのシンボル名を指摘した適切なエラーメッセージが表示されて、出力ファイルは生成されません。この重大なエラー状態によってリンカーは停止しますが、すべての入力ファイルの処理が、まず最初に完了します。この要領で、重大な解析エラーをすべて識別できます。

最も一般的な、重大エラー状態は、2つの再配置可能オブジェクトが両方とも同じ名前のシンボルを定義していて、どちらのシンボルもウィーク定義ではない場合に発生します。

```
$ cat foo.c
int bar = 1;

$ cat bar.c
bar()
{
    return (0);
}

$ cc -dn -r -o temp.o foo.c bar.c
ld: fatal: symbol `bar' is multiply defined:
      (file foo.o and file bar.o);
ld: fatal: File processing errors. No output written to int.o
```

ここで、`foo.c` と `bar.c` には、シンボル `bar` に対する重複する定義があります。リンカーは、どちらを優先すべきか判別できないため、通常、この処理は放棄されます。ただし、リンカーの `-z muldefs` オプションを使用すると、このエラー状態を防ぐことができ、リンカーが、最初のシンボル定義をとるように設定できます。

未定義シンボル

すべての入力ファイルを読み取り、シンボル解析がすべて完了すると、リンカーは、シンボル定義に結合されていないシンボルリファレンスの内部シンボルテーブルを検索します。これらのシンボルリファレンスは、未定義シンボルと呼ばれま

す。これらの未定義シンボルがリンク編集処理に及ぼす影響は、生成される出力ファイルのタイプや、場合によってはシンボルのタイプによって異なります。

実行可能ファイルの作成

リンカーが実行可能ファイルを作成しているときは、リンカーのデフォルトの動作は、シンボルを定義されないままにする必要がある適切なエラーメッセージを表示して、リンク編集を終了させることです。次のように、再配置可能オブジェクト内のシンボル「リファレンス」が、シンボル「定義」と絶対に一致しない場合に、シンボルは定義されないままの状態になります。

```
$ cat main.c
extern int foo();

main()
{
    return (foo());
}
$ cc -o prog main.c
Undefined      first referenced
 symbol         in file
foo             main.o
ld: fatal: Symbol referencing errors. No output written to prog
```

これと同様の方法で、共有オブジェクトが動的実行可能ファイルを作成するために使用されているときに、共有オブジェクト内のシンボルリファレンスが、シンボル定義と絶対に一致しない場合は、このシンボルリファレンスも未定義シンボルになります。

```
$ cat foo.c
extern int bar;
foo()
{
    return (bar);
}

$ cc -o libfoo.so -G -K pic foo.c
$ cc -o prog main.c -L. -lfoo
Undefined      first referenced
 symbol         in file
bar             ./libfoo.so
ld: fatal: Symbol referencing errors. No output written to prog
```

上記の例のような場合に、未定義シンボルを許可するには、リンカーの `-z nodefs` オプションを使用することにより、重大なエラー状態を防ぐことができます。

注 -z nodefs オプションを使用する場合は、注意が必要です。処理の実行中に使用できないシンボルリファレンスが要求されると、重大な実行時再配置エラーが発生します。このエラーは、初期化の実行中とアプリケーションのテスト中に検出できますが、実行パスが複雑であるために起こったこのエラー状態は、検出に時間がかかり、時間とコストが浪費される場合があります。

シンボルは、再配置可能オブジェクト内のシンボルリファレンスが、暗黙の内に定義された共有オブジェクト内のシンボル定義に結合されている場合にも、未定義シンボルのままになる場合があります。たとえば、上記の例で使用したファイル main.c および foo.c に以下のように続く場合です。

```
$ cat bar.c
int bar = 1;

$ cc -o libbar.so -R. -G -K pic bar.c -L. -lfoo
$ ldd libbar.so
    libfoo.so =>      ./libfoo.so

$ cc -o prog main.c -L. -lbar
Undefined          first referenced
 symbol            in file
foo                main.o (symbol belongs to implicit \
                  dependency ./libfoo.so)
ld: fatal: Symbol referencing errors. No output written to prog
```

ここで、prog は、libbar.so への明示的なリファレンスを使用して構築されます。また、libbar.so には libfoo.so への依存性があるため、prog から libfoo.so への暗黙的なリファレンスが確立します。

main.c は、libfoo.so によって作成されたインタフェースへの特定のリファレンスを実行するため、prog は、実際に libfoo.so に依存性を持つことになります。ただし、生成される出力ファイル内に記録されるのは、明示的な共有オブジェクトの依存関係だけです。そのため、libbar.so の新しいバージョンが開発され、libfoo.so への依存性がなくなった場合、prog は実行に失敗します。

この理由から、このタイプのバインディングは重大であると考えられ、暗黙的な参照は、prog のリンク編集にライブラリを直接参照することにより、明示的に実行される必要があります (この例で示した重大なエラーメッセージ内に必要なリファレンスのヒントがあります)。

共有オブジェクトの生成

リンカーが共有オブジェクトを生成する場合、デフォルトにより、未定義シンボルをリンク編集の末尾に残すことができます。これにより、共有オブジェクトはシンボルを、動的実行可能ファイルの作成に使用する場合、再配置可能オブジェクトまたは他の共有オブジェクトのどちらからでもインポートできます。

リンカーの `-z defs` オプションを使用すると、未定義シンボルが残っていた場合に、強制的に重大エラーにすることができます。共有オブジェクトを作成するときには、このオプションの使用をお奨めします。アプリケーションのシンボルを参照する共有オブジェクトでは、`-z defs` オプションを使用すれば、`extern mapfile` 指示文を使ってアプリケーションシンボルを定義できます。これについては、51 ページの「追加シンボルの定義」を参照してください。

通常、自己組み込み共有オブジェクトは、外部シンボルへのすべてのリファレンスは名前の付いた依存関係によって満たされ、最大の柔軟性が提供されます。この場合の共有オブジェクトは、共有オブジェクトの必要条件を満たす依存関係を判別し、確立したユーザー以外の、多数のユーザーによって使用されます。

ウィークシンボル

生成中の出力ファイルタイプがどのようなタイプであっても、リンク編集に結合されないウィークシンボルリファレンスにより、重大なエラー状態が発生します。

静的実行可能ファイルを生成中の場合は、シンボルは絶対シンボルに変換され、ゼロの値が割り当てられます。

動的実行可能ファイルまたは共有オブジェクトの作成中の場合は、シンボルは定義されていないウィークリファレンスとして残されます。プロセスの実行中に、実行時リンカーがこのシンボルを検索し、一致が検出されない場合は、重大な実行時再配置エラーを生成する代わりに、そのリファレンスをゼロのアドレスに結合します。

従来は、これらの定義されていないウィークリファレンスシンボルは、機能の存在をテストするためのメカニズムとして使用されていました。たとえば、次の C コードフラグは、共有オブジェクト `libfoo.so.1` 内で次のように使用されていました。

```
#pragma weak    foo

extern void    foo(char *);

void bar(char * path)
{
```

(続く)

```
void (* fptr) ();

if ((fptr = foo) != 0)
    (* fptr)(path);
}
```

アプリケーションがリファレンス `libfoo.so.1` で構築されると、シンボル `foo` の定義が検出されたかどうかに関係なく、リンク編集は、正常に完了します。アプリケーションの実行中に、機能アドレスが非ゼロをテストすると、その機能が呼び出されます。ただし、シンボル定義が検出されない場合には、機能アドレスはゼロをテストするため、その機能は呼び出されません。

ただし、コンパイルシステムは、定義されないセマンティクスを保持しながら、このアドレスの比較テクニックを参照します。その結果、テストステートメントは最適化処理によって削除されます。さらに、実行時シンボルの結合メカニズムでは、このテクニックの使用にこれ以外の制限も加え、これにより、すべての動的オブジェクトが整合性のあるモデルを使用できる状態ではなくなります。

ユーザーは、この方法で定義されていないウィークリファレンスを使用することをやめ、その代わりに `RTLD_DEFAULT` フラグを指定した `dlsym(3DL)` を使用してシンボルの存在テストを行うことをお勧めします (100ページの「機能のテスト」を参照)。

出力ファイル内の一時的シンボル順序

入力ファイルの追加は、通常、その追加の順に出力ファイルに表示されます。ただし、一時的シンボルとそれに関連する記憶領域を処理するときに、例外が発生します。一時的シンボルは、その解析が完了するまで完全に定義されません。再配置可能オブジェクトからの定義シンボルに遭遇すると、解析が実行されます。すると、表示される順序は、定義を調べるために実行された結果になります。

シンボルグループの順序を制御したい場合には、一時的定義は、ゼロで初期化されたデータ項目に再定義する必要があります。たとえば、次のような一時的定義をすると、出力ファイル内のデータ項目が、ソースファイル `foo.c` に記述された元の順序と比較されて再配列されます。

```

$ cat foo.c
char A_array[0x10];
char B_array[0x20];
char C_array[0x30];

$ cc -o prog main.c foo.c
$ nm -vx prog | grep array
[32] | 0x00020754 | 0x00000010 | OBJT | GLOB | 0x0 | 15 | A_array
[34] | 0x00020764 | 0x00000030 | OBJT | GLOB | 0x0 | 15 | C_array
[42] | 0x00020794 | 0x00000020 | OBJT | GLOB | 0x0 | 15 | B_array

```

これらのシンボルを、初期化されたデータ項目として定義することにより、入力ファイル内のこれに関連したシンボルの配列が、出力ファイル内にも持ち越されます。

```

$ cat foo.c
char A_array[0x10] = { 0 };
char B_array[0x20] = { 0 };
char C_array[0x30] = { 0 };

$ cc -o prog main.c foo.c
$ nm -vx prog | grep array
[32] | 0x000206bc | 0x00000010 | OBJT | GLOB | 0x0 | 12 | A_array
[42] | 0x000206cc | 0x00000020 | OBJT | GLOB | 0x0 | 12 | B_array
[34] | 0x000206ec | 0x00000030 | OBJT | GLOB | 0x0 | 12 | C_array

```

追加シンボルの定義

シンボルを入力ファイルから提供することに加えて、ユーザーは、リンク編集に、追加のシンボルリファレンスまたは定義を指定できます。最も簡単な形式で、シンボルリファレンスは、リンカーの `-u` オプションを使用して作成できます。より柔軟性の高いものは、リンカーの `-M` オプションと、それに関連した、シンボルリファレンスと種々のシンボル定義を定義できる `mapfile` を使用して作成できます。

`-u` オプションを指定すると、リンク編集コマンド行からシンボルリファレンスを作成するためのメカニズムが使用できます。このオプションは、リンク編集をすべてアーカイブから実行する場合に使用でき、また、複数のアーカイブから抽出するオブジェクトの選択における柔軟性を向上させることができます (アーカイブの抽出については、30ページの「アーカイブ処理」の項を参照)。

たとえば、動的実行可能ファイルの生成を、シンボル `foo` と `bar` へのリファレンスを実行する再配置可能オブジェクト `main.o` から抽出してみましょう。この場合、`lib1.a` 内に組み込まれた再配置可能オブジェクト `foo.o` からシンボル定義 `foo` を入手し、さらに `lib2.a` 内に組み込まれた再配置可能オブジェクト `bar.o` からシンボル定義 `bar` を入手します。

ただし、アーカイブ `lib1.a` にも、シンボル `bar` を定義する再配置可能オブジェクトが組み込まれています (`lib2.a` に提供されたものとは機能的に異なると仮定した場合)。必要なアーカイブ抽出を指定する場合は、次のようなリンク編集を使用できます。

```
$ cc -o prog -L. -u foo -l1 main.o -l2
```

ここで、`-u` オプションは、シンボル `foo` へのリファレンスを生成します。このリファレンスによって、再配置可能オブジェクト `foo.o` がアーカイブ `lib1.a` から抽出されます。シンボル `bar` への最初のリファレンスは、`lib1.a` が処理されてから生じる `main.o` 内で実行されるため、再配置可能オブジェクト `bar.o` はアーカイブ `lib2.a` から入手されます。

注 - この単純な例では、`lib1.a` からの再配置可能オブジェクト `foo.o` は、シンボル `bar` の直接的または間接的な参照は行いません。この参照を行なった場合、再配置可能オブジェクト `bar.o` は、その処理中に、`lib1.a` から抽出されます (アーカイブを処理するリンカーの多重パスについては、30ページの「アーカイブ処理」を参照)。

より広範囲なシンボル定義のセットは、リンカーの `-M` オプションと関連する `mapfile` を使用して入手できます。これらの `mapfile` エントリの構文は次のとおりです。

```
[ name ] {  
    scope:  
        symbol [ = [ type ] [ value ] [ size ] [ extern ] ] ;  
} [ dependency ] ;
```

name

このシンボル定義のセットのラベルは、もしあれば、イメージ内のバージョン定義を識別できます。詳細については、第5章を参照してください。

scope

生成される出力ファイル内のシンボルのバインディングの可視性を示しています。これには、値 `local` (ローカル) または `global` (大域) のいずれかが入ります。 `mapfile` で定義されたすべてのシンボルは、リンク編集プロセス中に、`scope` (スコープ) 内で `global` (大域) として処理されます。つまり、これらのシンボルは、入力ファイルのいずれかから入手された、同じ名前の他のシンボルに対して解

析されます。ただし、`local` (ローカル) `scope` として定義シンボルは、生成される実行可能ファイルまたは共有オブジェクトのファイル内のローカル結合が指定されたシンボルに変更されます。

symbol

要求されたシンボルの名前です。この名前の後に、シンボル属性 (*type*、*value*、*size* のいずれか) が付いていない場合には、シンボルリファレンスの作成になります。このリファレンスは、この項の最初に説明した `-u` オプションを使用して生成するリファレンスとまったく同じものです。このシンボル名にシンボル属性が付いている場合には、シンボル定義は、関連する属性を使用して生成されます。

`local` スコープ内では、このシンボル名は、特別な「`auto-reduction`」(自動縮小) 指示語「`*`」として定義できます。この指示語を使用すると、すべての大域シンボル (`mapfile` 内に `global` と明示的に定義されていないもの) が、生成される実行可能ファイルまたは共有オブジェクトファイル内で、ローカル結合を受け取ります。

type

シンボルのタイプ属性を示します。また、ここには、`data`、`function`、`common` のいずれかが入ります。最初の 2 つのタイプ属性の結果は、絶対的なシンボル定義になります (237 ページの「シンボルテーブル」を参照)。後者のタイプ属性の結果は、一時的シンボル定義になります。

value

シンボルの値属性を示し、`vnumber` の書式をとります。

size

シンボルのサイズ属性を示し、`snumber` の書式をとります。

extern

シンボルが、作成されているオブジェクトに外部的に定義されていることを示します。このオプションを使用して、`-z defs` オプションで示された未定義シンボル (49 ページの「共有オブジェクトの生成」を参照) を抑制できます。

dependency

この定義が継承する `version definition` (バージョン定義) を示します。詳細については、第 5 章を参照してください。

バージョン定義または自動縮小のいずれかの指示語が指定されている場合、バージョン情報が作成されるイメージ内に記録されます。このイメージが実行可能ファイルまたは共有オブジェクトである場合には、シンボル縮小も適用されます。

作成されるイメージが再配置可能オブジェクトである場合は、デフォルトにより、シンボル縮小は適用されません。この場合、シンボル縮小はバージョン情報の一部として記録され、これらの縮小は、再配置可能オブジェクトが最終的に実行可能ファイルまたは共有オブジェクトの生成に使用されるときに適用されます。リンカーの `-B reduce` オプションを使用すると、再配置可能オブジェクトを生成するときに、強制的にシンボル縮小を実行できます。

バージョン情報の詳細については、第 5 章に記載してあります。

注 - インタフェース定義を確実に安定させるためには、シンボル名の定義に対しワイルドカードによる拡張を行わないようにします。

この項では、この後、この `mapfile` 構文を使用した例をいくつか示します。

以下の例は、3つのシンボルリファレンスを定義する方法と、これらを使用してアーカイブから構成要素を抽出する方法を示しています。このアーカイブ抽出は、複数の `-u` オプションをリンク編集に指定することにより実現できますが、この例では、最終的なシンボルの範囲を、ローカルに縮小する方法も示しています。

```
$ cat foo.c
foo()
{
    (void) printf("foo: called from lib.a\n");
}
$ cat bar.c
bar()
{
    (void) printf("bar: called from lib.a\n");
}
$ cat main.c
extern void    foo(), bar();

main()
{
    foo();
    bar();
}
$ ar -rc lib.a foo.o bar.o main.o
$ cat mapfile
{
    local:
        foo;
        bar;
    global:
```

(続く)

```

        main;
    };
$ cc -o prog -M mapfile lib.a
$ prog
foo: called from lib.a
bar: called from lib.a
$ nm -x prog | egrep "main$|foo$|bar$"
[28] | 0x00010604 | 0x00000024 | FUNC | LOCL | 0x0 | 7 | foo
[30] | 0x00010628 | 0x00000024 | FUNC | LOCL | 0x0 | 7 | bar
[49] | 0x0001064c | 0x00000024 | FUNC | GLOB | 0x0 | 7 | main

```

大域からローカルへのシンボル範囲の縮小の重要性については、56ページの「シンボル範囲の縮小」で説明しています。

次の例では、2つの絶対シンボル定義を定義し、これらを使用して入力ファイル `main.c` からのリファレンスを解析する方法を示しています。

```

$ cat main.c
extern int    foo();
extern int    bar;

main()
{
    (void) printf("&foo = %x\n", &foo);
    (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
    global:
        foo = FUNCTION V0x400;
        bar = DATA V0x800;
};
$ cc -o prog -M mapfile main.c
$ prog
&foo = 400 &bar = 800
$ nm -x prog | egrep "foo$|bar$"
[37] | 0x00000800 | 0x00000000 | OBJT | GLOB | 0x0 | ABS | bar
[42] | 0x00000400 | 0x00000000 | FUNC | GLOB | 0x0 | ABS | foo

```

入力ファイルから入手される場合、関数のシンボル定義またはデータ項目は、通常、データ記憶域の要素に関連しています。mapfile 定義は、このデータ記憶域を構成するためには不十分であるため、これらのシンボルは、絶対値として残しておく必要があります。

ただし、mapfile は、common または一時的シンボルを定義する場合にも使用できます。他のタイプのシンボル定義とは違って、一時的シンボルは、ファイル内の記

憶域を占有しませんが、実行時に割り当てる記憶域の定義は行います。そのため、このタイプのシンボル定義は、作成される出力ファイルの記憶域割り当ての一因となります。

一時的シンボルの特徴は、他のシンボルタイプとは異なり、その値の属性によって、その配列条件が示される点です。mapfile 定義は、リンク編集の入力ファイルから入手された一時的定義の再配列に使用されます。

次の例では、2つの一時的シンボルの定義を示しています。シンボル foo は、新しい記憶領域を定義しているのに対し、シンボル bar は、実際に、ファイル main.c 内の同じ一時的定義の配列を変更するために使用されます。

```
$ cat main.c
extern int    foo;
int          bar[0x10];

main()
{
    (void) printf("&foo = %x\n", &foo);
    (void) printf("&bar = %x\n", &bar);
}
$ cat mapfile
{
    global:
        foo = COMMON V0x4 S0x200;
        bar = COMMON V0x100 S0x40;
};
$ cc -o prog -M mapfile main.c
ld: warning: symbol `bar' has differing alignments:
(file mapfile value=0x100; file main.o value=0x4);
largest value applied
$ prog
&foo = 20940
&bar = 20900
$ nm -x prog | egrep "foo$|bar$"
[37]  |0x00020900|0x00000040|OBJT |GLOB |0x0 |16 |bar
[42]  |0x00020940|0x00000200|OBJT |GLOB |0x0 |16 |foo
```

注 - このシンボル解析の診断は、リンカーの `-t` オプションを使用すると表示されません。

シンボル範囲の縮小

前の項では、mapfile 内のローカル範囲を持つようにシンボル定義を定義して、シンボルの最終的な結合を縮小する方法を示しました。このメカニズムは、入力の一

部として生成されたファイルを使用する、将来のリンク編集に対するシンボルの可視性を削減するという重要な役割を果たします。実際、このメカニズムは、ファイルのインタフェースの厳密な定義をするために提供されているため、他のユーザーに対して、機能の使用を制限できます。

たとえば、簡単な共有オブジェクトを、ファイル `foo.c` と `bar.c` から生成するとします。ファイル `foo.c` には、他のユーザーも使用できるように設定するサービスを提供する大域シンボル `foo` が組み込まれています。ファイル `bar.c` には、共有オブジェクトの根底となるインプリメンテーションを提供するシンボル `bar` と `str` が組み込まれています。簡単に作成された共有オブジェクトは、通常、これらの3つのシンボルすべてに、次のように大域範囲が指定されています。

```
$ cat foo.c
extern const char *   bar();

const char * foo()
{
    return (bar());
}
$ cat bar.c
const char * str = "returned from bar.c";

const char * bar()
{
    return (str);
}
$ cc -o lib.so.1 -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[29] | 0x000104d0 | 0x00000004 | OBJT | GLOB | 0x0 | 12 | str
[32] | 0x00000418 | 0x00000028 | FUNC | GLOB | 0x0 | 6 | bar
[33] | 0x000003f0 | 0x00000028 | FUNC | GLOB | 0x0 | 6 | foo
```

このようにすると、この共有オブジェクトが提供する機能を、他のアプリケーションのリンク編集の一部として使用できます。シンボル `foo` へのリファレンスは、共有オブジェクトによって提供されたインプリメンテーションに結合されます。

大域結合により、シンボル `bar` と `str` への直接リファレンスも可能ですが、これは、危険な結果を招く場合があります。それは、ユーザーが、関数 `foo` の基礎となるインプリメンテーションを後から変更すると、それが原因で知らないうちに、`bar` または `str` に結合された既存のアプリケーションが失敗または誤作動を起こします。

この他の、シンボル `bar` と `str` の大域結合の結果、同じ名前のシンボルによって、割り込みすることが可能になります(共有オブジェクト内へのシンボルの割り込みについては、42ページの「単純な解析」の項で説明しています)。この割り込みは、意図的に行うことができ、これを使用することにより、共有オブジェクトが提

供する目的の機能を取り囲むことができます。また反対に、この割り込みは、同じ共通のシンボル名をアプリケーションと共有オブジェクトの両方に使用した結果として、知らないうちに実行される場合もあります。

共有オブジェクトを開発する場合は、シンボル `bar` と `str` の範囲をローカル結合に縮小して、このような事態から保護することができます。次に例を示します。

```
$ cat mapfile
{
    local:
        bar;
        str;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[27] 0x000003dc|0x00000028|FUNC|LOCL|0x0|6|bar
[28] 0x00010494|0x00000004|OBJT|LOCL|0x0|12|str
[33] 0x000003b4|0x00000028|FUNC|GLOB|0x0|6|foo
```

ここでは、シンボル `bar` と `str` は、共有オブジェクトのインタフェースの一部としては使用できません。そのため、これらのシンボルは、外部のオブジェクトによって参照されることができないか、割り込みはできません。ユーザーは、インタフェースをこの共有オブジェクト用に効果的に定義できます。インプリメンテーションの基礎となる詳細を隠している間は、このインタフェースを管理できます。

このようなシンボル範囲の縮小には、この他にもパフォーマンスにおける利点があります。実行時に必要だったシンボル `bar` と `str` に対するシンボルの再配置は、現在は、関連する再配置に縮小されます。これにより、実行時の、共有オブジェクトの初期設定と処理のオーバーヘッドが削減されます (シンボル再配置のオーバーヘッドの詳細は、130ページの「再配置を実行する場合」を参照)。

リンク編集間で処理されるシンボル数が増えると、`mapfile` 内で各ローカル範囲への縮小を定義する能力の維持が困難になります。その代替りとなる、より柔軟性のあるメカニズムを使用すると、共有オブジェクトインタフェースを、保持する必要がある大域シンボルとして定義でき、他のシンボルはすべてローカル結合に縮小するようにリンカーに指示できます。このメカニズムは、特別な自動縮小指示語の「*」を使用して実行します。たとえば、前の `mapfile` 定義を書き換えて、`foo` を、生成される出力ファイル内で必要な大域シンボルとしてのみ定義します。

```
$ cat mapfile
lib.so.1.1
{
    global:
        foo;
    local:
        *;
```

```

};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[30] | 0x00000370 | 0x00000028 | FUNC | LOCL | 0x0 | 6 | bar
[31] | 0x00010428 | 0x00000004 | OBJT | LOCL | 0x0 | 12 | str
[35] | 0x00000348 | 0x00000028 | FUNC | GLOB | 0x0 | 6 | foo

```

この例では、バージョン名 `lib.so.1.1` も `mapfile` 指示語の一部として定義しています。このバージョン名により、ファイルのシンボルインタフェースを定義する、内部バージョン定義が確立されます。バージョン定義の作成は推奨されています。バージョン定義により、ファイルの展開全体を通して使用できる、内部バージョンメカニズムの基礎が形成されます。この詳細については、第5章を参照してください。

注 - バージョン名が指定されていないと、出力ファイル名がバージョン定義のラベル付けに使用されます。出力ファイル内に作成されたバージョン情報は、リンカーの `-z noversion` オプションを使用して表示しないようにできます。

バージョン名が指定されている場合は必ず、すべての大域シンボルをバージョン定義に割り当てる必要があります。バージョン定義に割り当てられていない大域シンボルが残っていると、リンカーにより重大なエラー状態が発生します。

```

$ cat mapfile
lib.so.1.1 {
    global:
        foo;
};
$ cc -o lib.so.1 -M mapfile -G foo.c bar.c
Undefined      first referenced
 symbol          in file
str              bar.o (symbol has no version assigned)
bar              bar.o (symbol has no version assigned)
ld: fatal: Symbol referencing errors. No output written to lib.so.1

```

実行可能ファイルまたは共有オブジェクトを作成するときに、シンボルの縮小処理が行われると、該当するシンボルが縮小されると同時に、出力イメージ内にバージョン定義が記録されます。デフォルトにより、再配置可能オブジェクトの生成時にバージョン定義は作成されますが、シンボルの縮小処理は行われません。その結果、シンボル縮小のシンボルエントリは、大域のまま残されます。たとえば、「自動縮小」指示語が指定された、前の `mapfile` と関連する再配置可能オブジェクト

を使用して、シンボル縮小が表示されていない中間再配置可能オブジェクトが作成されます。

```
$ ld -o lib.o -M mapfile -r foo.o bar.o
$ nm -x lib.o | egrep "foo$|bar$|str$"
[17] 0x00000000|0x00000004|OBJT|GLOB|0x0|3|str
[19] 0x00000028|0x00000028|FUNC|GLOB|0x0|1|bar
[20] 0x00000000|0x00000028|FUNC|GLOB|0x0|1|foo
```

ただし、このイメージ内に作成された「バージョン定義」は、シンボル縮小が要求されたという事実を記録します。再配置可能オブジェクトが、最終的に、実行可能ファイルまたは共有オブジェクトの生成に使用されるときに、シンボル縮小が実行されます。すなわち、リンカーは、mapfile からデータを処理するのと同じ方法で、再配置可能オブジェクト内に組み込まれたシンボル縮小を読み取り、解釈します。

そのため、上記の例で作成された中間再配置可能オブジェクトは、ここで、共有オブジェクトの生成に使用されます。

```
$ cc -o lib.so.1 -M mapfile -G lib.o
$ nm -x lib.so.1 | egrep "foo$|bar$|str$"
[22] 0x000104a4|0x00000004|OBJT|LOCL|0x0|14|str
[24] 0x000003dc|0x00000028|FUNC|LOCL|0x0|8|bar
[36] 0x000003b4|0x00000028|FUNC|GLOB|0x0|8|foo
```

シンボル縮小は、再配置可能オブジェクトが作成されるときに、リンカーの `-B reduce` オプションを使用して強制的に実行されます。

```
$ ld -o lib.o -M mapfile -B reduce -r foo.o bar.o
$ nm -x lib.o | egrep "foo$|bar$|str$"
[15] 0x00000000|0x00000004|OBJT|LOCL|0x0|3|str
[16] 0x00000028|0x00000028|FUNC|LOCL|0x0|1|bar
[20] 0x00000000|0x00000028|FUNC|GLOB|0x0|1|foo
```

外部結合

作成するオブジェクトのシンボルリファレンスが共有オブジェクト内の定義によって解決されると、そのシンボルは未定義のまま残り、シンボルに対応する再配置情報が実行時の検索で使用されます。定義を提供する共有オブジェクトは、通常、1つの依存条件になります。

実行時リンカーは、実行時にデフォルト検索モデルを使ってこの定義を見つけます。一般にリンカーはオブジェクトを1つずつ検索しますが、その際、動的実行可

能ファイルから始め、オブジェクトがロードされた順に各依存関係を処理します (76ページの「シンボルの検索」を参照)。

オブジェクトは、リンカーの `-Bdirect` オプションを使って作成することもできます。このオプションでは、参照されるシンボルとその定義を提供するオブジェクトとの関係は、作成するオブジェクト内に維持されます。この情報を使えば、実行時リンカーは、リファレンスと、シンボルを定義するオブジェクトを直接結合し、デフォルトのシンボル検索モデルをバイパスすることができます。直接結合情報は、リンク編集で指定される依存関係に対してのみ確立されます (したがって、`-z defs` オプションを推奨します)。直接結合では、実行時のシンボル検索処理が大幅に減ることがあります。この実行時結合のモデルに関する詳細は、78ページの「直接結合」を参照してください。

出力イメージの生成

入力ファイルの処理とシンボル解析がすべて重大なエラーが発生することもなく完了すると、リンカーは出力イメージファイルの生成を開始します。リンカーは、出力ファイルイメージを完成させるために生成する必要がある追加セクションを確立します。これには、入力ファイルからのローカルシンボル定義が組み込まれたシンボルテーブルとともに、その内部シンボルテーブルから収集された大域およびウィークシンボル情報が組み込まれます。

また、実行時リンカーが必要とする、出力の再配置および動的情報セクションも組み込まれます。出力セクション情報が確立されると、出力ファイルの合計サイズが算出され、それに従って出力ファイルイメージが作成されます。

動的実行可能ファイルまたは共有オブジェクトを作成するときに、通常、2つのシンボルテーブルが生成されます。`.dynsym` とその関連ストリングテーブル `.dynstr` には、レジスタ (これらがローカルであっても)、大域シンボル、ウィークシンボル、およびセクションシンボルが組み込まれます。これらのセクションは、実行時にプロセスイメージの一部として対応付けされる (mmap(2) のマニュアルページを参照) `text` セグメントの一部になります。これにより、実行時リンカーは、これらのセクションを読み取り、必要な再配置を実行できます。

`.symtab` とその関連ストリングテーブル `.strtab` には、入力ファイル処理から収集された「すべての」シンボルが含まれています。これらのセクションは、プロセスイメージの一部として対応付けされず、リンカーの `-s` オプションを使用するか、リンク編集後に `strip(1)` を使用して、イメージから取り除くことさえ可能です。

予約シンボルは、シンボルテーブルの生成中に作成されます。予約シンボルは、リンクプロセスに対する特別な意味を持ち、ユーザーのコードでは定義できません。

`_etext`

テキストセグメントの後の最初のロケーション

`_edata`

初期化されたデータの最初のロケーション

`_end`

すべてのデータの後の最初のロケーション

`__DYNAMIC`

動的情報セクション (`.dynamic` セクション) のアドレス

`__END__`

`_end` と同じですが、このシンボルにはローカル範囲があります (`「_START_」` を参照)。

`__GLOBAL_OFFSET_TABLE__`

リンカーが提供するアドレステーブル (`.got` セクション) への、ポジション固有のリファレンス。このテーブルは、ポジション固有のデータリファレンスから構成されます。このデータリファレンスは、`-K pic` オプションを使用してコンパイルされたオブジェクト内で発生します (詳細は、124ページの「位置に依存しないコード」を参照)。

`__PROCEDURE_LINKAGE_TABLE__`

リンカーが提供するアドレステーブル (`.plt` セクション) への、ポジション固有のリファレンス。このテーブルは、ポジション固有の関数リファレンスから構成されます。このデータリファレンスは、`-K pic` オプションを使用してコンパイルされたオブジェクト内で発生します (詳細は、124ページの「位置に依存しないコード」を参照)。

`_START_`

テキストセグメント内の最初のロケーション。このシンボルは、`_END_` とともに、ローカル範囲を持ち、オブジェクトのアドレス範囲を確立する手段を提供します。

リンカーは、実行可能ファイルを生成する場合、追加シンボルを検出して実行可能ファイルのエントリポイントを定義します。シンボルがリンカーの `-e` オプションを使用して指定された場合、これが使用されます。それ以外の場合は、リンカーは予約シンボル名 `_start` と `main` を検出します。これらのシンボルが存在しない場合には、テキストセグメントの最初のアドレスが使用されます。

再配置処理

出力ファイルを作成すると、入力ファイルからのすべてのデータセクションは新しいイメージにコピーされます。入力ファイル内に指定された再配置は、出力イメージに適用されます。生成する必要がある追加の再配置情報も、新しいイメージに書き込まれます。

再配置処理には、通常、大きな問題はありません。ただし、特定のエラーメッセージを伴うエラー状態が発生することがあります。ここでは、2つの状態について説明しておく必要があります。1つは、位置に依存するコード(124ページの「位置に依存しないコード」を参照)によって発生するテキスト再配置で、もう1つはディスプレイメント再配置に関連して発生します。ディスプレイメント再配置については、次の項で詳しく説明します。

ディスプレイメント再配置

データ項目(それ自体はコピー再配置で使用可能)にディスプレイメント再配置が適用されていると、エラー状態が発生することがあります。コピー再配置の詳細については、131ページの「コピー再配置」を参照してください。

ディスプレイメント再配置は、再配置されるオフセットと再配置される先のターゲットが両方とも同じ位置だけ離れている限り有効です。コピー再配置とは、共有オブジェクト内の大域データ項目を実行可能ファイルの `.bss` にコピーし、実行可能ファイルの読み取り専用テキストセグメントを同じ状態に保つことをいいます。コピーされるデータにディスプレイメント再配置が適用されていたり、外部

再配置がコピーされるデータへのディスプレイメントであったりすると、ディスプレイメント再配置は無効になります。

このようなエラーを検知するために、次の点に注意してください。

- コピー再配置可能なデータ項目がディスプレイメント再配置を伴うと問題が発生する可能性がある場合は、共有オブジェクトを生成するときにそれらに対しフラグを立てる。
- コピー再配置のデータがディスプレイメント再配置を伴う場合は、実行可能ファイルを生成するときにコピー再配置の作成に対しフラグを立てる。

前者の項目に関する問題は、大域データに対してディスプレイメント再配置を使用する共有オブジェクトはすでに多数あるため、このデータがコピー再配置される可能性があるという点です。リンカーが共有オブジェクトを構築する際には、共有オブジェクトに対し実行時にどのような参照がされるかは明らかではありません。したがって、フラグが立ったデータ項目は、エラーを引き起こす可能性があります。

後者の項目に関する問題は、共有オブジェクトに適用されるディスプレイメント再配置が、リンク編集で共有オブジェクトを作成する時点で完了することがあるという点です。したがって、この共有オブジェクトを参照するアプリケーションのリンク編集では、コピー再配置されたデータでどのようなディスプレイメントが有効になるのかはわかりません。

このような問題の診断を助けるため、リンカーは、動的オブジェクトに対してディスプレイメント再配置が使用されていると、1つまたは複数の動的 DT_FLAGS_1 フラグを立てます (表 7-45 を参照)。さらに、その可能性のある再配置をリンカーの `-z verbose` オプションを使って表示することもできます。

たとえば、大域データ項目 `bar[]` を持つ共有オブジェクトを作成し、このデータ項目にディスプレイメント再配置が適用されているとします。この項目は、動的実行可能ファイルから参照されると、コピー再配置される可能性があります。リンカーは、この状態に対し次の警告を出します。

```
$ cc -G -o libfoo.so.1 -z verbose -Kpic foo.o
ld: warning: relocation warning: R_SPARC_DISP32: file foo.o: symbol foo: \
  displacement relocation to be applied to the symbol bar: at 0x194: \
  displacement relocation will be visible in output image
```

次にデータ項目 `bar[]` を参照するアプリケーションを作成すると、コピー再配置が行われ、ディスプレイメント再配置が無効にされる可能性があります。リンカー

はこの状況を明示的に検出できるため、`-z verbose` オプションが使用されていなくても、次のエラーメッセージを生成します。

```
$ cc -o prog prog.o -L. -lfoo
ld: warning: relocation error: R_SPARC_DISP32: file foo.so: symbol foo: \
displacement relocation applied to the symbol bar at: 0x194: \
the symbol bar is a copy relocated symbol
```

このようなエラー状態は、再配置するシンボル定義(オフセット)と再配置のシンボルターゲットを両方ともローカルに置くことによって避けることができます。それには、静的な定義を使用するか、リンカーの範囲指定を使用します(56ページの「シンボル範囲の縮小」を参照)。共有オブジェクト内のデータにアクセスするには、機能インタフェースの使用が最も適しています。

注・リンカーの `-d` か `-r` オプションとともに `ldd(1)` を使用すると、ディスプレイメント動的フラグによって同じような再配置警告が生成されます。

デバッグガイド

Solaris リンカーには、デバッグライブラリが付いていて、これを使用するとリンク編集プロセスをより詳細に監視できます。このライブラリは、ユーザー自身のアプリケーションまたはライブラリのリンク編集を理解またはデバッグする場合に役立ちます。これは、ビジュアルエイドで、このライブラリを使用して表示される情報のタイプは、定数のままであると予期されますが、この情報の正確な形式は、リリースごとにわずかに変更される場合があります。

ELF フォーマットを熟知していないと、デバッグ出力の中には見慣れないものがあるかもしれませんが、多くのものが一般的な関心を惹くものでしょう。

デバッグは、`-D` オプションを使用して実行できます。また、作成された出力はすべて標準エラーに直接送信されます。このオプションは、1つまたは複数のトークンで拡張し、必要なデバッグのタイプを指示する必要があります。使用できるトークンは、`-Dhelp` を使用して表示できます。次に例を示します。

```
$ ld -Dhelp
debug:
debug:          For debugging the link-editing of an application:
debug:          LD_OPTIONS=-Dtoken1,token2 cc -o prog ...
debug:          or,
```

```

debug:          ld -Dtoken1,token2 -o prog ...
debug:          where placement of -D on the command line is significant
debug:          and options can be switched off by prepending with '!'.
debug:
debug:
debug: args      display input argument processing
debug: basic     provide basic trace information/warnings
debug: detail    provide more information in conjunction with other
debug:           options
debug: entry     display entrance criteria descriptors
debug: files     display input file processing (files and libraries)
debug: help      display this help message
debug: libs      display library search paths; detail flag shows actual
debug:           library lookup (-l) processing
debug: map       display map file processing
debug: move      display move section processing
debug: reloc     display relocation processing
debug: sections  display input section processing
debug: segments  display available output segments and address/offset
debug:           processing; detail flag shows associated sections
debug: support   display support library processing
debug: symbols   display symbol table processing;
debug:           detail flag shows resolution and linker table addition
debug: versions  display version processing
debug: got       display GOT symbol information

```

注 - このリストは一例で、リンカーに有用なオプションを表示しています。正確なオプションは、リリースごとに異なる場合があります。

ほとんどのコンパイラドライバは、`-D` オプションの解釈をその前処理フェーズ中に行うため、リンカーにこのオプションを渡すためには、`LD_OPTIONS` 環境変数のメカニズムが適しています。

次の例では、入力ファイルの監視方法を示しています。これは、特に、リンク編集中に、配置されたライブラリ、またはアーカイブから抽出された再配置可能オブジェクトを判別する場合に有用です。

```

$ LD_OPTIONS=-Dfiles cc -o prog main.o -L. -lfoo
.....
debug: file=main.o [ ET_REL ]
debug: file=./libfoo.a [ archive ]
debug: file=./libfoo.a(foo.o) [ ET_REL ]
debug: file=./libfoo.a [ archive ] (again)
.....

```

ここでは、prog のリンク編集を満足させるために、構成要素 foo.o がアーカイブライブラリ libfoo.a から抽出されています。foo.o の抽出が、その他の再配置可能オブジェクトの抽出を認めていないことを検証するために、このアーカイブが 2 回 (again) 検索されていることに注意してください。(again) が複数表示されていることによって、このアーカイブが lorder(1) と tsort(1) を使用した並び替えの候補になっていることがわかります。

シンボルトークンを使用することにより、どのシンボルによってアーカイブ構成要素が抽出されたか、また、最初のシンボルリファレンスを実行したオブジェクトを判別できます。

```
$ LD_OPTIONS=-Dsymbols cc -o prog main.o -L. -lfoo
.....
debug: symbol table processing; input file=main.o [ ET_REL ]
.....
debug: symbol[7]=foo (global); adding
debug:
debug: symbol table processing; input file=./libfoo.a [ archive ]
debug: archive[0]=bar
debug: archive[1]=foo (foo.o) resolves undefined or tentative symbol
debug:
debug: symbol table processing; input file=./libfoo(foo.o) [ ET_REL ]
.....
```

ここでは、シンボル foo は、main.o によって参照され、リンカーの内部シンボルテーブルに付加されます。このシンボルリファレンスによって、再配置可能オブジェクト foo.o が、アーカイブ libfoo.a から抽出されます。

注 - この出力は、このマニュアル用に簡素化したものです。

detail トークンを、シンボルトークンとともに使用すると、入力ファイル処理中のシンボル解析を監視できます。

```
$ LD_OPTIONS=-Dsymbols,detail cc -o prog main.o -L. -lfoo
.....
debug: symbol table processing; input file=main.o [ ET_REL ]
.....
debug: symbol[7]=foo (global); adding
debug:   entered  0x000000 0x000000 NOTY GLOB UNDEF REF_REL_NEED
debug:
debug: symbol table processing; input file=./libfoo.a [ archive ]
debug: archive[0]=bar
debug: archive[1]=foo (foo.o) resolves undefined or tentative symbol
debug:
debug: symbol table processing; input file=./libfoo.a(foo.o) [ ET_REL ]
```

(続く)

```

debug: symbol[1]=foo.c
.....
debug: symbol[7]=bar (global); adding
debug:   entered 0x000000 0x000004 OBJT GLOB 3      REF_REL_NEED
debug: symbol[8]=foo (global); resolving [7][0]
debug:   old 0x000000 0x000000 NOTY GLOB UNDEF main.o
debug:   new 0x000000 0x000024 FUNC GLOB 2      ./libfoo.a(foo.o)
debug: resolved 0x000000 0x000024 FUNC GLOB 2      REF_REL_NEED
.....

```

ここでは、main.o からの、オリジナルの未定義シンボル foo が、アーカイブ構成要素 foo.o から抽出されたシンボル定義で上書きされます。このシンボルの詳細情報は、各シンボルの属性に反映されます。

上記の例からわかるように、デバッグトークンのいくつかを使用すると、豊富な出力が作成されます。入力ファイルのサブセットにかかわるアクティビティだけに興味がある場合には、-D オプションを、リンク編集コマンド行内に直接配置し、オンとオフを切り替えます。次に例を示します。

```

$ ld .... -o prog main.o -L. -Dsymbols -lbar -D!symbols ....

```

ここでは、シンボル処理の表示がオンになるのは、ライブラリ libbar の処理中だけです。

注 - リンク編集コマンド行を入手するには、使用しているドライバからコンパイル行を拡張する必要があります。詳細については、27ページの「コンパイラドライバを使用する」を参照してください。

実行時リンカー

概要

「動的実行プログラム」の初期設定と実行の一部として、インタプリタは、アプリケーションのその依存関係への結合を完了させるために呼び出されます。Solaris では、このインタプリタを実行時リンカーと呼びます。

動的実行プログラムのリンク編集に、特別な `.interp` セクションが、関連するプログラムのヘッダーとともに作成されます。このセクションには、プログラムのインタプリタを指定するパス名が組み込まれています。リンカーによって提供されたデフォルトの名前は、32 ビットの実行プログラムの場合は実行時リンカー `/usr/lib/ld.so.1`、64 ビット実行プログラムの場合は `/usr/lib/64/ld.so.1` となります。

動的実行プログラムの実行プロセス中に (`exec(2)` を参照)、カーネルはファイル `mmap(2)` を参照) を対応付けして、プログラムのヘッダー情報 (279 ページの「プログラムヘッダー」を参照) を使って、必要なインタプリタの名前を検出します。カーネルは、このインタプリタの対応付けと、インタプリタへの制御の転送を行い、同時に重要な情報を渡して、インタプリタがアプリケーションの結合を続行してからそのアプリケーションを実行できるようにします。

またアプリケーションの初期設定に加えて、実行時リンカーは、サービスを提供します。このサービスを使用すると、追加オブジェクトを対応付けして、シンボルをその中に結合することにより、アプリケーションはそのアドレススペースを拡張できます。

次に、この章で説明する、実行時リンカーの機能を簡単に紹介します。

- 実行プログラムの動的情報セクション (.dynamic) を分析し、必要な依存関係を判定する
- これらの依存関係内に配置および対応付けを行い、動的情報セクションを分析して、追加の依存関係が必要かどうか判定する
- すべての依存関係が配置され読み込まれると、実行時リンカーは必要な再配置を実行し、これらのオブジェクトをプロセスの実行に備えて結合する
- 依存関係によって作成された初期設定関数を呼び出す
- アプリケーションへの制御を渡す
- アプリケーションの実行中に、実行時リンカーは、遅延された関数の結合を実行するよう要求される
- アプリケーションも、実行時リンカーサービスに、`dlopen(3DL)` によって追加のオブジェクトを入手するように要求し、`dlsym(3DL)` を使用してこれらのオブジェクト内のシンボルに結合する

共有オブジェクトの依存関係の配置

通常、動的実行プログラムのリンク編集中に、1つまたは複数の共有オブジェクトが明示的に参照されます。これらのオブジェクトは、依存関係として動的実行プログラム内に記録されます (詳細については、31ページの「共有オブジェクトの処理」を参照)。

実行時リンカーは、まず最初にこの依存情報を配置し、これを使用して関連オブジェクトの配置および対応付けを行います。これらの依存関係は、実行プログラムのリンク編集中に参照された順番で処理されます。

動的実行プログラムの依存関係がすべて読み込まれると、これらの依存関係も読み込まれた順番に検査され、追加の依存関係が配置されます。この処理は、すべての依存関係の配置と読み込みが完了するまで続きます。この技術の結果、すべての依存関係が幅優先順になります。

実行時リンカーによって検索されるディレクトリ

デフォルトでは、実行時リンカーが、依存関係の検出場所として認識しているのは、32 ビットの依存関係の場合は `/usr/lib`、64 ビットの依存関係の場合は

/usr/lib/64 という標準的なディレクトリだけです。単純なファイル名で指定された依存関係には、このディレクトリ名が接頭辞として付き、この接頭辞が付いたパス名は、実際のファイルを配置する場合に使用されます。

動的実行プログラムまたは共有オブジェクトの実際の依存関係は、`ldd(1)` を使用して表示できます。たとえば、ファイル `/usr/bin/cat` には次のような依存関係があります。

```
$ ldd /usr/bin/cat
    libc.so.1 => /usr/lib/libc.so.1
    libdl.so.1 => /usr/lib/libdl.so.1
```

ここでは、ファイル `/usr/bin/cat` は、依存関係を持つか、またはファイル `libc.so.1` と `libdl.so.1` が必要です。

ファイル内に実際に記録された依存関係は、ファイルの `.dynamic` セクションと `NEEDED` タグの付いたエントリの参照を表示する `dump(1)` コマンドを使用して検査できます。次に例を示します。

```
$ dump -Lvp /usr/bin/cat

/usr/bin/cat:
[INDEX] Tag      Value
[1]      NEEDED    libc.so.1
.....
```

上記の `ldd(1)` の例で表示された依存関係 `libdl.so.1` は、ファイル `/usr/bin/cat` 内に記録されないことに注意してください。これは、つまり `ldd(1)` が、指定されたファイルのすべての依存関係を示していて、`libdl.so.1` は、実際に `/usr/lib/libc.so.1` の依存関係であるためです。

上記の `dump(1)` の例では、依存関係は、`'/'` が含まれていない単純なファイル名で表示されています。単純なファイル名を使用することは、実行時リンカーが、一連の規則に従って必要なパス名を生成する場合に必要です。`'/'` が組み込まれたファイル名は、そのまま使用されます。

単純なファイル名の記録は、標準的な、依存関係を記録する最も柔軟性の高いメカニズムで、依存関係内の単純な名前を記録する、リンカーの `-h` オプションを使用して実行されます(このトピックの詳細については、108ページの「命名規約」と109ページの「共有オブジェクト名の記録」を参照)。

通常、依存関係は、`/usr/lib` あるいは `/usr/lib/64` 以外に配布されません。動的実行プログラムまたは共有オブジェクトが、他のディレクトリに依存関係を配置す

る必要がある場合、実行時リンカーは、明示的に、このディレクトリを検索するように指示されます。

追加の検索パスを実行時リンカーに指示する場合は、動的実行プログラムまたは共有オブジェクトのリンク編集に、「実行パス」を記録する方法をお勧めします(この情報の記録方法の詳細については、37ページの「実行時リンカーが検索するディレクトリ」を参照)。

実行パスの記録は、`dump(1)` を使用して表示し、`RPATH` タグの付いたエントリを参照できます。次に例を示します。

```
$ dump -lvp prog

prog:
[INDEX] Tag      Value
[1]      NEEDED    libfoo.so.1
[2]      NEEDED    libc.so.1
[3]      RPATH     /home/me/lib:/home/you/lib
.....
```

ここでは、`prog` は `libfoo.so.1` 上に依存関係を持っていて、実行時リンカーのデフォルトロケーション `/usr/lib` を調べる前に、ディレクトリ `/home/me/lib` と `/home/you/lib` を検索するように要求します。

実行時リンカーの検索パスに追加するもう 1 つの方法は、環境変数 `LD_LIBRARY_PATH` を設定する方法です。この環境変数は、プロセスの始動時に 1 度分析され、コロンで区切られたディレクトリのリストに設定できます。実行時リンカーは、このリストに設定したディレクトリを、指定された実行パスまたはデフォルトのディレクトリよりも前に検索します。

これらの環境変数は、アプリケーションを強制的にローカルな依存関係に結合するといったデバッグの目的に適しています。次に例を示します。

```
$ LD_LIBRARY_PATH=. prog
```

ここで、上記の例のファイル `prog` は、現在の作業ディレクトリ内で検出された `libfoo.so.1` に結合されます。

この環境変数の使用は、実行時リンカーの検索パスに影響する一時的なメカニズムとしては有用ですが、作成ソフトウェアの場合は、環境変数を使用するには大きな支障があります。この環境変数を参照できる動的実行プログラムは、その検索パスを拡張させます。これにより、全体のパフォーマンスが低下する場合があります。

また、36ページの「環境変数の使用」と37ページの「実行時リンカーが検索するディレクトリ」で示したように、この環境変数はリンカーに影響を及ぼします。

64ビットの実行プログラムが、検出する名前と一致する32ビットのライブラリを組み込んだ検索パスを持つように、またはその逆の環境を継承できます。このような場合、実行時リンカーは一致しない32ビットライブラリを拒否し、有効な64ビットと一致するライブラリを検出して、検索パスの処理を続けます。一致するものが見つからない場合には、エラーメッセージが表示されます。このエラーは、LD_DEBUG 環境変数 (103ページの「デバッグエイド」を参照) を設定してファイルを組み合わせることによって、詳細に監視できます。

```
$ LD_LIBRARY_PATH=/usr/bin/64 LD_DEBUG=files /usr/bin/ls
...
00283: file=libc.so.1; needed by /usr/bin/ls
00283:
00283: file=/usr/lib/64/libc.so.1 rejected: ELF class mismatch: \
00283:                               32-bit/64-bit
00283:
00283: file=/usr/lib/libc.so.1 [ ELF ]; generating link map
00283:   dynamic: 0xef631180 base: 0xef580000 size:      0xb8000
00283:   entry:    0xef5a1240 phdr: 0xef580034 phnum:      3
00283:   lmid:     0x0
00283:
00283: file=/usr/lib/libc.so.1; analyzing [ RTLD_GLOBAL RTLD_LAZY ]
...
```

依存関係が配置できない場合は、ldd(1)により、オブジェクトが検出できないことが表示され、また、アプリケーションを実行しようとする、実行時リンカーから該当するエラーメッセージが表示されます。

```
$ ldd prog
      libc.so.1 => (file not found)
      libfoo.so.1 => /usr/lib/libfoo.so.1
      libdl.so.1 => /usr/lib/libdl.so.1
$ prog
ld.so.1: prog: fatal: libfoo.so.1: open failed: No such file or directory
```

デフォルトの検索パスの設定

実行時リンカー (/usr/lib または /usr/lib/64) によって使用されるデフォルトの検索パスは、実行時構成ファイルを使用して管理できます。このファイルは、crle(1) ユーティリティによって作成され、正しい実行パスで作成されなかったアプリケーションについて検索パスを設定する場合に便利です。

構成ファイルのデフォルトの場所は、`/var/ld/ld.config` (32 ビットアプリケーションの場合) または `/var/ld/64/ld.config` (64 ビットアプリケーションの場合) で、システム上の、それぞれのタイプのアプリケーションすべてに影響します。構成ファイルはこれ以外の場所にも作成でき、実行時リンカー `LD_CONFIG` 環境変数を使用してこれらのファイルを選択できます。後者の方法は、構成ファイルをデフォルトの場所にインストールする前にテストする場合に便利です。

動的ストリングトークン

実行時リンカーは、実行パス (`DT_RUNPATH` または `DT_RPATH`)、フィルタ (`DT_FILTER`)、または補助フィルタ (`DT_AUXILIARY`) 内で使用される場合には、次のストリングトークンが置換されます。

`$ISALIST`

このプラットフォームで実行可能なネイティブの命令セット (`isalist(1)` を参照) に展開する。このトークンを含むパス名は、使用可能な各命令セットに置き換えられる。362ページの「命令セット固有の共有オブジェクト」を参照。

上記で指定されたパス内あるいは依存関係 (`DT_NEEDED`) エントリ内で実行時リンカーが使用されると、その実行時リンカーは次のストリングトークンを置き換えます。

`$ORIGIN`

オブジェクトが読み込まれるディレクトリを指定する。通常は、単独のバンドルされていないパッケージ内に依存関係を配置する場合に使用される。364ページの「関連する依存関係の配置」を参照

`$OSNAME`

オペレーティングシステムの名前に展開 (`uname(1) -s` を参照) する。364ページの「プラットフォーム固有の共有オブジェクト」を参照

`$OSREL`

オペレーティングシステムのリリースに展開 (`uname(1) -r` を参照) する。364ページの「プラットフォーム固有の共有オブジェクト」を参照

\$PLATFORM

現在のマシンの現在のプロセッサタイプに展開 (uname (1) -i を参照) する。364 ページの「プラットフォーム固有の共有オブジェクト」を参照

注 - \$PLATFORM は「Solaris 2.5」に付加されましたが、「Solaris 2.6」より前のものでは、補助フィルタ (DT_AUXILIARY) を表現するためだけに使用できます。118 ページの「補助フィルタの生成」を参照してください。

再配置処理

実行時リンカーは、アプリケーションが要求する依存関係がすべて配置され、読み込まれると、各オブジェクトを処理し、必要な再配置すべてを実行します。

オブジェクトのリンク編集に、入力再配置の可能なオブジェクトとともに提供された再配置の情報が、出力ファイルに適用されます。ただし、動的実行プログラムまたは共有オブジェクトを作成している場合には、ほとんどの再配置は、リンク編集時には完了できません。それは、再配置には、オブジェクトがメモリー内に読み込まれる時だけ入手することができる、論理アドレスが必要だからです。このような場合、リンカーは新しい再配置を出力ファイルイメージの一部として記録し、これが、実行時リンカーが現在処理する必要がある情報になります。

再配置のさまざまなタイプの詳細については、251ページの「再配置型(プロセッサ固有の)」を参照してください。ここでは、再配置を1つか2つのタイプに分けて説明します。

- 非シンボル再配置
- シンボル再配置

オブジェクトの再配置記録は、dump (1) を使用して表示できます。次に例を示します。

```
$ dump -rvp libbar.so.1

libbar.so.1:

.rela.got:
Offset      Symndx          Type          Addend
```

(続く)

0x10438	0	R_SPARC_RELATIVE	0
0x1043c	foo	R_SPARC_GLOB_DAT	

ここでは、ファイル `libbar.so.1` には、「大域オフセットテーブル」(`.got` セクション)が更新される必要があることを示す、2つの再配置記録が組み込まれています。

最初の再配置は、単純な相対再配置です。このことは、再配置タイプとシンボルインデックス (`Symndx`) フィールドがゼロであることから分かります。この再配置では、オブジェクトがメモリーに読み込まれたベースアドレスを使用して、関連する `.got` オフセットを更新する必要があります。

2番目の再配置では、シンボル `foo` のアドレスが必要です。この再配置を完了させるには、実行時リンカーが、これまでに読み込まれた動的実行プログラムと依存関係からこのシンボルを配置する必要があります。

シンボルの検索

オブジェクトがシンボルを必要とする場合、実行時リンカーはそのシンボルを、オブジェクトのシンボル要求の検索範囲と、プロセス内の各オブジェクトによって提供されるシンボルの可視性に基づいて検索します。これらの属性は、読み込まれる時に、オブジェクトのデフォルトとして使用され、`dlopen(3DL)` の特別なモードとしても使用されます。さらに、場合によっては、オブジェクトの構築時に、オブジェクト内に記録されます。

平均的なユーザーであれば、動的実行プログラムとその依存関係、および `dlopen(3DL)` を通じて入手したオブジェクトに適用されるデフォルトのシンボル検索モードが、理解できるようになります。前者の検索モードについては、次の項、77ページの「デフォルトの検索」で概要を説明します。種々のシンボル検索に活用できる後者については、92ページの「シンボル検索」で説明しています。

リンカーの `-Bdirect` オプションを使って動的オブジェクトを作成すると、別のシンボル検索モデルが使用されます (60ページの「外部結合」を参照)。この場合、実行時リンカーは、シンボルを検索する際に、リンク編集時にそのシンボルを指定したオブジェクトから直接探します。このモデルの詳細は、78ページの「直接結合」を参照してください。

デフォルトの検索

動的実行プログラムと、ともに読み込まれるすべての依存関係には、ワールド検索範囲と、大域シンボル可視性が割り当てられます (92ページの「シンボル検索」を参照)。これにより、実行時リンカーが、動的実行プログラムまたはこの実行プログラムとともに読み込まれた依存関係すべてを調べてシンボルを検出する場合、各オブジェクトの検索は、動的実行プログラムから開始され、次にそのオブジェクトが対応付けされた順番でそれぞれの依存関係を検索していきます。

前の項で説明したように、`ldd(1)` を使用すると、動的実行プログラムの依存関係は読み込まれた順番にリストされます。そのため、共有オブジェクト `libbar.so.1` がシンボル `foo` の再配置を行うためにそのアドレスを必要とし、かつ共有オブジェクトが動的実行プログラム `prog` の依存関係である場合には、実行時リンカーは、`foo` の検索を、最初に動的実行プログラム `prog` 内で実行し、次に共有オブジェクト `/home/me/lib/libfoo.so.1` 内で、最後に共有オブジェクト `/home/me/lib/libbar.so.1` 内で実行します。

```
$ ldd prog
  libfoo.so.1 => /home/me/lib/libfoo.so.1
  libbar.so.1 => /home/me/lib/libbar.so.1
```

注 - シンボル検索は、シンボル名のサイズが増大し依存関係の数が増加すると、特にコストのかかる処理になる可能性があります。このパフォーマンスについての詳細は、120ページの「性能に関する考慮事項」で説明しています。これに代わる検索モデルについては、78ページの「直接結合」を参照してください。

割り込み (interposition)

最初に動的実行プログラム内でシンボルの検索を行い、次に各依存関係内で検索を行うという実行時リンカーのデフォルトのメカニズムは、要求されたシンボルの最初の出現が、この検索を満足させることを意味しています。そのため、同じシンボルの複数のインスタンスが存在する場合は、最初のインスタンスが、他のすべてのインスタンスに割り込みされます (31ページの「共有オブジェクトの処理」も参照)。

直接結合

リンカーの `-Bdirect` オプションを使ってオブジェクトを作成すると、参照されるシンボルと、定義を提供する依存条件との関係は、オブジェクトに記録されます。実行時リンカーは、デフォルトのシンボル検索モデルを使用する代わりに、この情報を使って関連するオブジェクトから直接シンボルを検索します。

注 - `-Bdirect` オプションを使用すると、レイジーローディングも有効になります。これは、リンク編集のコマンド行の先頭に `-z lazyload` オプションを指定すると同じことです (82ページの「動的依存関係のレイジーローディング」を参照)。

この直接結合モデルでは、多数のシンボル再配置や依存関係を伴う動的プロセスでのシンボル検索オーバーヘッドを大幅に削減できます。さらに、このモデルでは、同じ名前の複数のシンボルを、それらが直接結合されている個々のオブジェクトから見つけることができます。

しかし、直接結合ではデフォルトの検索モデルがバイパスされるため、従来から使用されている割り込みシンボルが迂回される可能性があります。デフォルトのモデルでは必ず、1つのシンボルへのすべてのリファレンスは同じ1つの定義に結合されます。

直接結合環境でも、オブジェクト単位で、割り込みを行うことができます。それには、オブジェクトが割り込み処理として識別される必要があります。環境変数 `LD_PRELOAD` を使ってオブジェクトを読み込むか (81ページの「追加オブジェクトの読み込み」を参照)、リンカーの `-z interpose` オプションを使ってオブジェクトを作成すると、オブジェクトは割り込み処理として識別されます。実行時リンカーは、直接結合されたシンボルを検索する際に、割り込み処理として識別されたオブジェクトを最初に探してから、シンボル定義を指定するオブジェクトを探します。

注 - 直接結合を実行時に無効にするには、環境変数 `LD_NODIRECT` にヌル以外の値を設定します。

再配置が実行されるとき

再配置処理を、非シンボルとシンボルの2つのタイプに分けて簡素化し、簡単に説明してきましたが、さらに再配置処理の実行時に、これを区別すると便利です。このような、再配置されたオフセットに対して行われる参照のタイプによって、区別が実行されます。参照のタイプは、次のいずれかになります。

- データ参照

- 関数参照

データ参照とは、アプリケーションコードによってデータ項目として使用されるアドレスを参照することです。実行時リンカーは、アプリケーションコードに関する知識がないため、このデータ項目がいつ参照されるか認識できません。そのため、データ再配置はすべて、アプリケーションが制御を入手する前の、処理の初期設定中に実行されます。

関数参照とは、アプリケーションコードによって呼び出された関数のアドレスを参照するものです。動的モジュールのコンパイルおよびリンク編集中に、大域関数の呼び出しは再配置されて、プロシージャのリンクテーブルエントリの呼び出しになります(これらのエントリにより、`.plt` セクションが構成されます)。

プロシージャのリンクテーブルのエントリは、最初に呼び出された制御が実行時リンカーに渡されたときに構成されます(317ページの「プロシージャのリンクテーブル(プロセッサ固有)」を参照)。実行時リンカーは、要求されたシンボルを検索しアプリケーション内の情報を書き換えます。そのため、後に発生する `.plt` エントリへの呼び出しは、関数に直接送信されます。このメカニズムを使用すると、このタイプの再配置を、関数の最初のインスタンスが呼び出されるまで延期することができます。この処理をレイジー結合と呼びます。

レイジー結合を実行する、実行時リンカーのデフォルトモードは、空文字以外の値に環境変数 `LD_BIND_NOW` を設定することにより上書きされます。この環境変数の設定を行うと、実行時リンカーは、プロセスの初期設定中にデータ参照と関数参照の両方の再配置を実行してから、アプリケーションに制御を転送します。次に例を示します。

```
$ LD_BIND_NOW=yes prog
```

ここでは、ファイル `prog` 内とその依存関係の中のすべての再配置は制御がアプリケーションに移る前に処理されることになります。

個々のオブジェクトも、オブジェクトが読み込み時に再配置処理が完了している必要があることを指示するために、リンカーの `-z now` オプションを使用して構築されます。この再配置の必要条件は、実行時にマークされたオブジェクトの依存関係にも伝達されます。

再配置エラー

最も一般的な再配置エラーは、シンボルが検出されるときに発生します。この状態になると、適切な実行時リンカーのエラーメッセージが表示され、アプリケーションは終了します。次に例を示します。

```
$ ldd prog
  libfoo.so.1 => ./libfoo.so.1
  libc.so.1 => /usr/lib/libc.so.1
  libbar.so.1 => ./libbar.so.1
  libdl.so.1 => /usr/lib/libdl.so.1
$ prog
ld.so.1: prog: fatal: relocation error: file ./libfoo.so.1: \
symbol bar: referenced symbol not found
```

ここでは、ファイル `libfoo.so.1` 内で参照されたシンボル `bar` は配置できません。

注 - 動的実行プログラムのリンク編集に、このソートの再配置エラーが起こる可能性は、定義されていない重大なシンボルとしてフラグが付けられます (この例については、47ページの「実行可能ファイルの作成」を参照)。この実行時の再配置エラーは、`main` のリンク編集が、`bar` のシンボル定義が組み込まれた共有オブジェクト `libbar.so.1` の異なったバージョンを使用した場合、または `-z nodefs` オプションがリンク編集の一部として使用された場合に発生します。

データ参照として使用されたシンボルが配置できないために、このタイプの再配置エラーが発生した場合は、そのエラー状態は、プロセスの初期設定の直後にも発生します。ただし、レイジー結合のデフォルトモードが原因で、関数参照として使用されるシンボルが検出できない場合は、このエラー状態は、アプリケーションが制御を受け取ってから発生します。

後者の場合、コードを実行する実行パスによって、エラー状態が発生するまでに数分または数ヶ月かかる場合もあり、あるいは発生しない場合もあります。この種のエラーを防ぐためには、動的実行プログラムまたは共有オブジェクトの再配置の必要条件を、`ldd(1)` を使用して有効にしておきます。

`ldd(1)` とともに `-d` オプションを指定すると、すべての依存関係が出力され、すべてのデータ参照の再配置処理が実行されます。データ参照が解析できない場合には、診断メッセージが作成されます。上記の例から、次のように明示されます。


```
$ ldd -d prog
libfoo.so.1 => ./libfoo.so.1
libc.so.1 => /usr/lib/libc.so.1
libbar.so.1 => ./libbar.so.1
libdl.so.1 => /usr/lib/libdl.so.1
symbol not found: bar (.libfoo.so.1)
```

ldd(1) とともに `-r` オプションを指定すると、すべてのデータと関数参照の再配置が処理されます。また、この両方ともが解析されない場合には、診断メッセージが作成されます。

追加オブジェクトの読み込み

前の項では、実行時リンカーが、動的実行プログラムとその依存関係からのプロセスを、各モジュールのリンク編集集中に定義されたように初期設定する方法について説明しました。また、実行時リンカーでは、プロセスの初期設定中に新しいオブジェクトを採用できるという、一歩進んだ柔軟性も提供しています。

環境変数 `LD_PRELOAD` は、共有オブジェクトまたは再配置可能なオブジェクトのフレーム名、あるいは複数のフレーム名を空白で区切ったストリングに初期設定できます。これらのオブジェクトは、動的実行プログラムの後で、依存関係よりも前に読み込まれ、ワールド検索範囲と大域シンボル可視性が割り当てられます (92ページの「シンボル検索」を参照)。次に例を示します。

```
$ LD_PRELOAD=./newstuff.so.1 prog
```

ここでは、動的実行プログラム `prog` が読み込まれ、次に共有オブジェクト `newstuff.so.1` が続き、その次に `prog` 内に定義された依存関係が続きます。これらのオブジェクトが処理される順番は、`ldd(1)` を使用して表示できます。

```
$ LD_PRELOAD=./newstuff.so.1 ldd prog
./newstuff.so.1 => ./newstuff.so
libc.so.1 => /usr/lib/libc.so.1
```

次に別の例を示します。

```
$ LD_PRELOAD="./foo.o ./bar.o" prog
```

この例の事前読み込みは、少し複雑で時間がかかります。実行時リンカーは、最初に再配置可能オブジェクト `foo.o` と `bar.o` をリンク編集し、メモリー内に保持されていた共有オブジェクトを生成します。次にこのメモリーイメージは、この前の例で示した共有オブジェクト `newstuff.so.1` の事前読み込みとまったく同じ方法で、動的実行プログラムとその依存関係との間に挿入されます。ここでも、これらのオブジェクトが処理される順番は、`ldd(1)` を使用して表示できます。

```
$ LD_PRELOAD="./foo.o ./bar.o" ldd prog
./foo.o => ./foo.o
./bar.o => ./bar.o
libc.so.1 => /usr/lib/libc.so.1
```

オブジェクトを動的実行プログラムの後に挿入するこれらのメカニズムは、77ページの「割り込み (**interposition**)」で説明した割り込みの概念を別のレベルに受け継いだものです。これらのメカニズムを使用すると、標準的な共有オブジェクト内に常駐する関数の、新しいインプリメンテーション (実現) を試すことができます。この関数が組み込まれたオブジェクトをあらかじめ読み込むことにより、この関数は元のオブジェクトにも割り込みされます。そのため、古い関数は、新しく読み込まれたバージョンによって完全に隠れてしまいます。

この他にも事前読み込みは、標準的な共有オブジェクト内に常駐する関数を補強するために使用できます。これが、新しいシンボルを元のシンボルに割り込みさせる目的です。これにより、新しい関数は、元の関数への呼び出し機能も保持しながら、この他の処理も追加実行できます。このメカニズムには、元の関数に関連したシンボルエイリアスか (42ページの「単純な解析」を参照)、または元のシンボルのアドレスを検索する機能が必要です (101ページの「割り込み (**interposition**) の使用」を参照)。

動的依存関係のレイジーローディング

動的依存関係を読み込むデフォルトのモデルは、動的依存関係をメモリーに対応付けて、追加の依存関係があるかどうか調べます。もし依存関係がある場合には、これらは順番に、すぐに読み込まれます。このサイクルは、依存関係のツリー全体を使い果たすまで、つまりすべての内部オブジェクトの参照 (再配置) を解決するまで続けられます。

このモデルでは、これらの依存関係内のコードが実行中にアプリケーションによって実際に参照されるかどうかに関係なく、アプリケーションの依存関係がすべてメモリー内に読み込まれ、すべてのデータの再配置が実行されます。

レイジーローディングモデルでは、レイジーローディングのラベルが付いた依存関係は、明示的に参照が行われるまで読み込まれません。関数呼び出しレイジー結合 (317ページの「プロシージャのリンクテーブル (プロセッサ固有)」を参照) を利用して、依存関係の読み込みを最初に参照されるまで延期することができます。実際に参照されないオブジェクトは読み込まれません。

注・オブジェクトへの参照は、データ項目または関数に対して行うことができます (78ページの「再配置が実行される時」を参照)。データへの参照はオブジェクトが初期設定された時に解決される必要があるため、1つのデータへの参照を満たすすべての依存関係はすぐにロードされなければなりません。そのため、そういった依存関係をレイジーロード可能として示すことは、あまり効果がありません。動的オブジェクト間のデータへの参照は、概してあまり推奨されません。

レイジーローディングの例として、`liblddbg.so.4` というデバッグライブラリを参照するリンカーそのものがあります。デバッグを呼び出すことはまれなので、リンカーを呼び出すたびにこのライブラリを読み込むことは不要で、コストがかさみます。このライブラリをレイジー読み込みできるように指定することにより、その処理コストをデバッグ出力を必要とする読み込みに使うことができます。

このレイジーローディングモデルを実行するための代替メソッドは、必要に応じて依存関係に `dlopen()` または `dlsym()` を実行することです。これは、`dlsym()` を通じた参照の数が少ない場合、あるいはリンク編集時に依存関係の名前あるいはロケーションがわからない場合に最適です。ただし、名前やロケーションがわかっている依存関係のより複雑な相互作用については、通常のシンボル参照のコードを使用し、依存関係をレイジー読み込みに指定する方が簡単です。

レイジー読み込みあるいは通常の読み込みが実行されるようにオブジェクトに指定するにはそれぞれ、リンカーオプション `-z lazyload` と `-z nolazyload` を使用します。これらのオプションはリンク編集コマンド行の位置によって決まり、フラグを持つ依存関係が見つかったら、そのフラグで指定された読み込み属性が採用されます。

ここでは、`libdebug.so.1` に依存する単純なプログラムをレイジー読み込みにする例を示します。動的セクション (`.dynamic`) を検査すると `libdebug.so.1` にレイジー読み込みのマークが付けられていることが示され、シンボル情報セクショ

ン (.SUNW_syminfo) を検査すると、その読み込みをトリガーするシンボルの参照が示されます。

```
$ cc -o prog prog.c -L. -zlazyload -ldebug -znolazyload -R'$ORIGIN'
$ elfdump -d prog

Dynamic Section: .dynamic
  index  tag          value
  [1]    POSFLAG_1    0x1          [ LAZY ]
  [2]    NEEDED       0x123        libdebug.so.1
  [3]    NEEDED       0x131        libc.so.1
  [6]    RPATH        0x13b        $ORIGIN
  ...
$ elfdump -y prog

Syminfo section: .SUNW_syminfo
  index flgs boundto          symbol
  ...
  [52] DL          [1] libdebug.so.1  debug
```

値に LAZY が指定された POSFLAG_1 は、次の NEEDED エントリ libdebug.so.1 がレイジー読み込みされるように指定することに注意してください。ただし、libc.so.1 は前に LAZY フラグが無いので、prog の初期始動時に読み込まれます。

注 - 実行時にレイジーローディングを無効にするには、環境変数 LD_NOLAZYLOAD をヌル以外の値に設定します。

初期設定および終了ルーチン

制御をアプリケーションに転送する前に、実行時リンカーは、アプリケーションおよびその依存関係内で検出された初期設定セクションを処理します。これらの .preinit_array、.init_array、.init セクションは、動的オブジェクトが構築される際にリンカーによって作成され、それぞれ .dynamic タグの DT_PREINIT_ARRAY、DT_INIT_ARRAY、DT_INIT でラベル付けされます (38ページの「初期設定および終了セクション」を参照)。

DT_PREINIT_ARRAY および DT_INIT_ARRAY で指定された配列内に含まれたアドレスを持つ関数は、配列内でアドレスが出現する順と同じ順序で実行時リンカーによって実行されます。1つのオブジェクトに DT_INIT エントリと DT_INIT_ARRAY エントリの両方が含まれる場合は、そのオブジェクトについて

DT_INIT エントリによって参照される関数は、DT_INIT_ARRAY エントリによって参照される関数より前に処理されます。

動的実行可能ファイルは、`.preinit_array` セクション内で初期設定前関数を提供することができます。これらの関数は、実行時リンカーがプロセスイメージを構築して再配置を実行し終わった後で、かつ他の初期設定関数の前に実行されます。初期設定前関数は、共有オブジェクト内では許可されません。

注 - 動的実行可能ファイル内のすべての DT_INIT セクションは、コンパイラドライバから供給されるプロセスの起動メカニズムによって、アプリケーション自体から呼び出されます。動的実行可能ファイルの DT_INIT セクションは、そのすべての依存関係の初期設定セクションが実行された後で、最後に呼び出されます。

Solaris 2.6 より前のリリースでは、依存関係からの初期設定ルーチンは、読み込まれた順序の降順で、つまり、`ldd(1)` を使用して表示された依存関係の順と逆の順序で呼び出されていました。

Solaris 2.6 リリースからは、実行時リンカーは、読み込まれた依存関係から、初期設定ルーチンの依存関係の配列リストを作成します。このリストは、各オブジェクトが表す依存関係の相関関係に加えて、表示された依存関係の外部で発生した結合から構成されます。

初期設定セクションは、依存関係が配列された順序の逆の順序で実行されます。循環性のある依存関係が検出された場合は、その周期を形成するオブジェクトは、トポロジカルソートは実行できません。そのため、この初期設定セクションは、読み込まれた順序の逆の順序で実行されることになります。

`-i` オプションを指定した `ldd(1)` を使用すると、オブジェクトの依存関係の初期設定の順番を表示できます。たとえば、次の動的実行プログラムとその依存関係は、循環性のある依存関係を示しています。

```
$ dump -Lv B.so.1 | grep NEEDED
[1]   NEEDED      C.so.1
$ dump -Lv C.so.1 | grep NEEDED
[1]   NEEDED      B.so.1
$ dump -Lv main | grep NEEDED
[1]   NEEDED      A.so.1
[2]   NEEDED      B.so.1
[3]   NEEDED      libc.so.1
$ ldd -i main
A.so.1 =>      ./A.so.1
B.so.1 =>      ./B.so.1
libc.so.1 =>   /usr/lib/libc.so.1
C.so.1 =>      ./C.so.1
```

(続く)

```

libdl.so.1 => /usr/lib/libdl.so.1

cyclic dependencies detected, group [1]:
./libC.so.1
./libB.so.1

init object=/usr/lib/libc.so.1
init object=./A.so.1
init object=./C.so.1 - cyclic group [1], referenced by:
./B.so.1
init object=./B.so.1 - cyclic group [1], referenced by:
./C.so.1

```

注 - Solaris 8 10/00 より前のリリースでは、環境変数 LD_BREADTH に空文字以外の値を設定すると、リンカーで初期設定セクションを Solaris 2.6 より前の順序で実行することができました。しかし、多数のアプリケーションを初期化すると、依存関係が複雑になり、位相的な並び替えが必要になるため、この機能は Solaris 8 10/00 から無効にされています。LD_BREADTH の設定は無視され、メッセージは表示されません。

初期設定処理は、`dlopen(3DL)` が指定された実行中のプロセスに追加されたオブジェクトごとに繰り返されます。

循環的な依存関係が存在することはよくあるため、実行時リンカーはさらに、初期設定を動的に呼び出して、そのオブジェクトのコードが呼び出される前に初期設定セクションを実行しようとしています。実行時リンカーは、シンボルの結合の際に、結合する先のオブジェクトの初期設定セクションがすでに呼び出されているかどうかを判定し、呼び出されていないならば、シンボル結合手順を終る前に、それを呼び出します (78ページの「再配置が実行されるとき」を参照)。初期設定呼び出しの正確な順序は、環境変数 LD_DEBUG に *basic* を指定すれば見ることができます (103ページの「デバッグエイド」を参照)。

動的オブジェクトは、終了セクションも提供できます。これらの `.fini_array` および `.fini` セクションは、動的オブジェクトが構築され、`.dynamic` タグの `DT_FINI_ARRAY` および `DT_FINI` でラベル付けされる際に、リンカーによって作成されます (38ページの「初期設定および終了セクション」を参照)。

終了ルーチンは、`atexit(3C)` によって記録できるように構成されます。これらのルーチンは、プロセスが `exit(2)` を呼び出したとき、またはオブジェクト

が、`dlclose(3DL)` が指定された実行プロセスから除去されたときに呼び出されま
す。

`DT_FINI_ARRAY` によって指定された配列内に含まれたアドレスを持つ関数は、その
配列内でアドレスが出現する順序と逆の順序で、実行時リンカーに実行されます。1
つのオブジェクトに `DT_FINI` エントリと `DT_FINI_ARRAY` エントリの両方が含ま
れる場合は、そのオブジェクトについて `DT_FINI_ARRAY` エントリによって参照さ
れる関数は、`DT_FINI` エントリによって参照される関数よりも前に処理されます。

注 - 動的実行プログラム内の `.fini` セクションは、コンパイラドライバから提供さ
れるプロセスの終了メカニズムによってアプリケーション自体から呼び出されま
す。動的実行プログラムの `.fini` セクションは、その依存関係の終了セクションが
実行される前に、最初に呼び出されます。

Solaris 2.6 リリースからは、終了ルーチンが依存関係の配列順に呼び出されます。
Solaris 2.6 より前のリリースの場合、終了ルーチンは読み込まれた順に呼び出され
ていました。

この初期設定および終了の呼び出し順序は、簡単明瞭に見えますが、この順序を強
調しすぎないように注意が必要です。オブジェクトの順序は、共有オブジェクト
とアプリケーションの開発によって左右される場合があるからです (詳細について
は、114ページの「依存関係の並べ替え」を参照)。

セキュリティ

セキュアプロセスには、その依存関係と実行パスを評価し、不当な依存関係の置換
またはシンボルの割り込みを防ぐために使用されるいくつかの制約があります。

実行時リンカーは、そのユーザーがスーパーユーザーではなく、実際のユーザーと
実効ユーザー ID が同じではない場合、または実際のグループと実効グループ ID が
同じではない場合、プロセスをセキュアとして分類します
(`getuid(2)`、`geteuid(2)`、`getgid(2)` および `getegid(2)` を参照)。

実行時リンカーが認識するデフォルトのトラストディレクトリは、32 ビットのオブ
ジェクトの場合は `/usr/lib/secure`、64 ビットオブジェクトの場合は
`/usr/lib/secure/64` です。ユーティリティ `crle(1)` を使用すれば、セキュアア
プリケーション向けに追加のトラストディレクトリを指定できます。この方法を使

用する場合には、管理者は、ターゲットディレクトリを悪意のある侵入から適切に保護する必要があります。

LD_LIBRARY_PATH ファミリの環境変数がセキュアプロセス用に有効である場合は(70ページの「実行時リンカーによって検索されるディレクトリ」を参照)、この変数によって指定されたトラストディレクトリのみが、実行時リンカーの検索規則を補強するために使用されます。

セキュアプロセスでは、実行パスがフルパスである場合は、アプリケーションまたはその依存関係によって指定された実行パスの指定(70ページの「実行時リンカーによって検索されるディレクトリ」を参照)が使用されます。つまり、パス名の冒頭部分に / が付く場合です。

セキュアプロセスでは、\$ORIGIN 文字列の拡張は(368ページの「セキュリティ」を参照)、トラストディレクトリに拡張される場合のみ有効です。

追加オブジェクトは、LD_PRELOAD 環境変数(81ページの「追加オブジェクトの読み込み」を参照)、または LD_AUDIT 環境変数(169ページの「実行時リンカーの監査インタフェース」を参照)を使用したセキュアプロセスで読み込まれます。これらのオブジェクトはフルパス名または単純ファイル名で指定しなければなりません。フルパス名は、既知のトラストディレクトリに限定されます。単純ファイル名(名前に / がついていない)は、前述した検索パスの制約に従って配置され、既知のトラストディレクトリにのみ解決されることとなります。

セキュアプロセスでは、単純ファイル名を構成する依存関係は、前述のパス名の制約を使用して処理されます。フルパス名または相対パス名で表示された依存関係は、そのまま使用されます。そのため、セキュアプロセスの開発者は、フルパス名または相対パス名の依存関係として参照されるターゲットディレクトリを、不当な侵入から確実に保護する必要があります。

セキュアプロセスを作成する場合には、依存関係の表示や、dlopen(3DL)パス名の構築に、相対パス名は使用しないことをお勧めします。この制約は、アプリケーションと依存関係すべてに適用されます。

実行時リンクのプログラミングインタフェース

ここまでは、アプリケーションのリンク編集中に指定された依存関係が、プロセスの初期設定中に実行時リンカーによってどのように処理されるかについて説明してきました。このメカニズムに加えて、アプリケーションは、追加オブジェクトと結合することにより、その実行中にアドレススペースを拡張できます。この拡張は、ア

アプリケーションのリンク編集に指定された依存関係の処理と同じ実行時リンカーのサービスを、アプリケーションが要求できるようにすることで提供されます。

この遅延オブジェクトの結合処理には、いくつかの利点があります。

- アプリケーションの初期設定中ではなく、オブジェクトが要求された時点でオブジェクトを処理することにより、起動時間を大幅に削減できる。実際、ヘルプや情報のデバッグといったアプリケーションの特定の動作中に、そのサービスが必要とされない場合は、オブジェクトが要求されないことがある
- アプリケーションは、ネットワークングプロトコルなどの、必要なサービスによって決まる、いくつかの異なるオブジェクト間で選択される
- 実行時にオブジェクトに追加されたプロセスのアドレススペースは、使用後には解放される

以下は、アプリケーションが追加の共有オブジェクトにアクセスするために実行する典型的な手順を示しています。これについては次の項で説明します。

- 共有オブジェクトは、`dlopen(3DL)` を使用して実行中のアプリケーションのアドレススペースに配置され、追加される。この共有オブジェクトが所有する依存関係は、この時点で配置されて追加される
- 追加された共有オブジェクトとその依存関係は、再配置され、これらのオブジェクト内の初期設定セクションが呼び出される
- アプリケーションは、追加されたオブジェクト内のシンボルを、`dlsym(3DL)` を使用して配置する。次に、アプリケーションはデータを参照するか、またはこの新しいシンボルによって定義された関数を呼び出す
- オブジェクトによってアプリケーションが終了した後で、`dlclose(3DL)` を使用してアドレススペースを解放できる。解放されたオブジェクト内の終了セクションは、この時点で呼び出される
- これらの実行時リンカーのインタフェースルーチンを使用した結果発生したエラー状態は、`dLError(3DL)` を使用して表示できる

実行時リンカーのサービスは、ヘッダーファイル `dlfcn.h` 内に定義され、共有オブジェクト `libdl.so.1` によってアプリケーションで使用できるようになります。次に例を示します。

```
$ cc -o prog main.c -ldl
```

ここでは、ファイル `main.c` は、ルーチンの `dlopen(3DL)` ファミリのどれでも参照でき、アプリケーション `prog` は、実行時にこれらのルーチンと結合できます。

追加オブジェクトの読み込み

追加オブジェクトは、`dlopen(3DL)` を使用して、実行プロセスのアドレススペースに追加できます。この関数は、引数としてファイル名と結合モードを入手し、アプリケーションにハンドルを戻します。このハンドルを使用すると、アプリケーションは、`dlsym(3DL)` を使用することによってシンボルを配置できます。

ファイル名が、単純ファイル名で指定されている、つまり名前の中に '/' が組み込まれていない場合、実行時リンカーは一連の規則を使用して、適切なパス名を生成します。 '/' が組み込まれたファイル名は、そのまま使用されます。

これらの検索パスの規則は、最初の依存関係の配置に使用された規則と全く同じものです (70ページの「実行時リンカーによって検索されるディレクトリ」を参照)。たとえば、ファイル `main.c` は、以下のようなコードフラグメントが組み込まれている場合、共有オブジェクト `foo.so.1` を配置するために、実行時リンカーは、プロセスの初期設定時に表示された `LD_LIBRARY_PATH` 定義に、リンク編集 `prog` 中に指定された実行パスを続けて入力し、最後にデフォルトのロケーション `/usr/lib` を入力して使用します。

```
#include <stdio.h>
#include <dlfcn.h>

main(int argc, char ** argv)
{
    void * handle;
    .....

    if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
        (void) printf("dlopen: %s\n", dlerror());
        exit (1);
    }
    .....
}
```

ファイル名が次のように指定されている場合、実行時リンカーは、プロセスの現在の作業ディレクトリ内でこのファイルだけを検索します。

```
if ((handle = dlopen("./foo.so.1", RTLD_LAZY)) == NULL) {
```

注 - `dlopen(3DL)` を使用して指定された共有オブジェクトは、そのバージョンのファイル名で参照することをお勧めします (バージョンについての詳細は、158ページの「バージョンアップファイル名の管理」を参照)。

必要なオブジェクトが配置されていない場合は、`dlopen(3DL)` によって `NULL` ハンドルが戻されます。この場合、`dlerror(3DL)` を使用すると、失敗した真の理由を表示できます。次に例を示します。

```
$ cc -o prog main.c -ldl
$ prog
dlopen: ld.so.1: prog: fatal: foo.so.1: open failed: No such \
file or directory
```

`dlopen(3DL)` によって追加されたオブジェクトに、他のオブジェクトに依存する関係がある場合、その依存関係もプロセスのアドレススペースに配置されます。このプロセスは、指定されたオブジェクトの依存関係がすべて読み込まれるまで継続されます。この依存関係のツリーをグループと呼びます。

`dlopen(3DL)` によって指定されたオブジェクト、または依存関係がすでにプロセスイメージの一部である場合は、そのオブジェクトはこれ以上処理されません。しかし、この場合でも有効なハンドルは、アプリケーションに戻されます。このメカニズムにより、同じオブジェクトが複数回読み込まれることを防ぐことができます。また、このメカニズムを使用すると、アプリケーションは専用のハンドルを入手できます。たとえば、前述した `main.c` の例には、次のような `dlopen()` 呼び出しが組み込まれている場合、`dlopen(3DL)` から戻されたハンドルは、アプリケーションそのものの中、プロセスの初期設定の一部として読み込まれた依存関係の中、または `RTLD_GLOBAL` フラグが指定された `dlopen(3DL)` を使用してプロセスのアドレススペースに追加されたオブジェクトの中のシンボルを配置できます。

```
if ((handle = dlopen((const char *)0, RTLD_LAZY)) == NULL) {
```

再配置処理

69ページの「概要」で説明したように、オブジェクトの配置および対応付け後、実行時リンカーは各オブジェクトを処理し、必要な再配置を実行する必要があります。`dlopen(3DL)` を使用してプロセスのアドレススペースに配置されたオブジェクトは、同じ方法で再配置する必要もあります。

単純なアプリケーションの場合には、このプロセスはまったく関係ないこともありますが、多くのオブジェクトを含む多数の `dlopen(3DL)` 呼び出しと、共通の依存関係も伴う複雑なアプリケーションを所有するユーザーにとって、このことは非常に重要です。

再配置は、実行された時間によって分類されます。実行時リンカーのデフォルトの動作では、初期設定時にデータ参照の再配置がすべて処理され、通常レイジー結合と呼ばれるプロセスの実行時に関数参照がすべて処理されます。

この同じメカニズムは、モードが `RTLD_LAZY` として定義されているときに、`dlopen(3DL)` を使用して追加されたオブジェクトに適用されます。この代わりとしては、オブジェクトが追加されたときに、オブジェクトの再配置すべてをすぐに実行する必要があります。これは、モード `RTLD_NOW` を使用することによって、またはリンカーの `-z now` オプションを使用して作成されたときに、オブジェクト内のこの必要条件を記録することによって実現されます。この再配置の必要条件は、オープン状態のオブジェクトの依存関係に伝達されます。

また、再配置は、非シンボリックおよびシンボリックにも分類できます。このセクションの後半では、シンボル再配置がいつ発生するかに関係なく、この再配置に関連した問題について、シンボル検索の詳細に焦点をあてて説明します。

シンボル検索

`dlopen(3DL)` によって取得したオブジェクトが大域シンボルを参照する場合は、実行時リンカーは、プロセスを作成したオブジェクトのプールからこのシンボルを配置する必要があります。直接結合が無い場合は、`dlopen(3DL)` によって入手されたオブジェクトには、次の節で記述されているようにデフォルトのシンボル検索モデルが適用されます。ただし、プロセスを作成したオブジェクトの属性と結合される `dlopen(3DL)` のモードは、代わりにシンボル検索のモデルに提供されます。

直接結合を指定されたオブジェクトでは、それに対応する依存関係から直接、シンボルが検索されます (78ページの「直接結合」を参照)。ただし、この後で述べるすべての属性はそのまま有効です。

オブジェクトの2つの属性は、シンボル検索に影響を与えます。1つ目は、オブジェクトシンボルの検索範囲の要求で、2つ目は、プロセス内の各オブジェクトが提供するシンボルの可視性です。オブジェクトの検索範囲には次のものがあります。

ワールド (`world`)

オブジェクトは、プロセス内の他の大域オブジェクト内で検索されます。

グループ (`group`)

オブジェクトは、同じグループ内のオブジェクト内でのみ検索されます。`dlopen(3DL)` を使用して入手されたオブジェクトから作成された依存関係ツリー、またはリンカーの `-B group` オプションを使用して構築されたオブジェクトから作成された依存関係ツリーは、固有のグループを形成します。

オブジェクトからのシンボルの可視性には、次のものがあります。

大域 (global)

オブジェクトのシンボルは、ワールド検索範囲を持つオブジェクトから参照できます。

ローカル (local)

オブジェクトのシンボルは、同じグループを構成する他のオブジェクトからのみ参照されます。

デフォルトにより、`dlopen(3DL)` を使用して入手したオブジェクトには、ワールドシンボル検索範囲とローカルシンボル可視性が割り当てられます。93ページの「デフォルトのシンボル検索モデル」では、このデフォルトモデルを使用して、典型的なオブジェクトグループのインタラクションについて説明しています。96ページの「大域オブジェクトの定義」、97ページの「グループの分離」、および97ページの「オブジェクト階層」では、デフォルトのシンボル検索の展開に `dlopen(3DL)` モードとファイル属性を使用する例を示しています。

デフォルトのシンボル検索モデル

`dlopen(3DL)` によって追加された各オブジェクトでは、実行時リンカーは、最初に動的実行プログラム内でシンボルを検索し、次に、プロセスの初期設定中に提供されたそれぞれのオブジェクト内を検索します。ただし、シンボルが検出されない場合には、実行時リンカーは、`dlopen(3DL)` によって入手されたオブジェクト内と、その依存関係内の検索を続行します。

たとえば、動的実行プログラム `prog` と共有オブジェクト `B.so.1` を入手してみます。この2つには、それぞれ以下の単純な依存関係が付いています。

```
$ ldd prog
   A.so.1 =>          ./A.so.1
$ ldd B.so.1
   C.so.1 =>          ./C.so.1
```

`prog` が、`dlopen(3DL)` を使用して共有オブジェクト `B.so.1` を入手した場合、共有オブジェクト `B.so.1` と `C.so.1` の再配置に必要なシンボルが、最初に `prog` 内で検索され、`A.so.1`、`B.so.1`、`C.so.1` の順番に検索されます。このような単純

なケースでは、`dlopen(3DL)` によって入手された共有オブジェクトは、アプリケーションの元のリンク編集の末尾に追加されたと考える方が簡単な場合があります。たとえば、上記のオブジェクトの参照を図示すると、次のようになります。

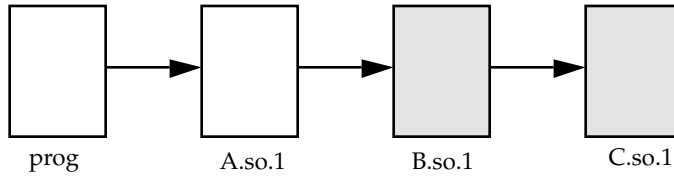


図 3-1 単一の `dlopen()` 要求

`dlopen(3DL)` から入手されたオブジェクトによって要求されたシンボル検索は、影付きのブロックで示しています。このシンボル検索は、動的実行プログラム `prog` から、最後の共有オブジェクト `C.so.1` へと進みます。

このシンボル検索は、読み込まれたオブジェクトに割り当てられた属性によって確立されます。動的実行プログラムとそれと同時に読み込まれたすべての依存関係には、大域シンボル可視性が割り当てられ、新しいオブジェクトにはワールドシンボルの検索範囲が割り当てられることを思い出してください。これによって、新しいオブジェクトは元のオブジェクト内を調べてシンボルを検索できます。また、新しいオブジェクトは、固有のグループを形成し、このグループ内では、各オブジェクトはローカルシンボル可視性を持ちます。そのため、グループ内の各オブジェクトは、他のグループ構成要素内でシンボルを検索できます。

これらの新しいオブジェクトは、アプリケーションまたはその最初のオブジェクトの依存関係によって要求される、通常のシンボル検索には影響を与えません。たとえば、上記の `dlopen(3DL)` が実行された後で、`A.so.1` に関数再配置が必要な場合、実行時リンカーの再配置シンボルの通常の検索は、`prog` と `A.so.1` で実施されますが、その後で `B.so.1` または `C.so.1` は検索されません。

このシンボル検索は、読み込まれたときにオブジェクトに割り当てられた属性によって実行されます。動的実行プログラムとこれとともに読み込まれた依存関係に割り当てられたワールドシンボルの検索範囲では、ローカルシンボル可視性だけを提供する新しいオブジェクト内を検索できません。

これらのシンボル検索とシンボル可視性の属性は、そのプロセスのアドレススペースへの投入とオブジェクト間の依存の関係に基づいて、オブジェクト間の関係を保持します。指定された `dlopen(3DL)` に関連したオブジェクトを割り当てることにより、固有のグループでは、同じ `dlopen(3DL)` と関連したオブジェクトだけが、グループ内のオブジェクトと関連する依存関係の中の検索ができます。

このオブジェクト間の関係を定義するという概念は、複数の `dlopen(3DL)` を実行するアプリケーション内では、より明確になります。たとえば、共有オブジェクト `D.so.1` に次の依存関係があるとします。

```
$ ldd D.so.1
   E.so.1 =>          ./E.so.1
```

このとき `prog` アプリケーションが、共有オブジェクト `B.so.1` に加えて、この共有オブジェクトにも `dlopen(3DL)` を実行した場合は、オブジェクト間のシンボル検索の関係は、図 3-2 のように表すことができます。

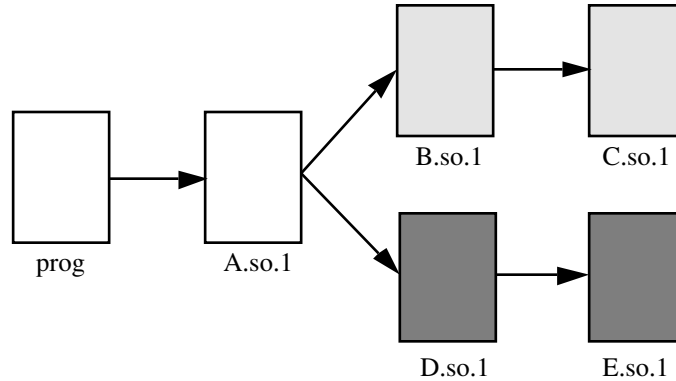


図 3-2 複数の `dlopen()` 要求

`B.so.1` と `D.so.1` の両方にシンボル `foo` の定義が組み込まれ、`C.so.1` と `E.so.1` にこのシンボルが必要とする再配置が組み込まれている場合、固有のグループに対するオブジェクトの関係によって、`C.so.1` は `B.so.1` の定義に結合され、`E.so.1` は `D.so.1` の定義に結合されます。このメカニズムは、`dlopen(3DL)` への複数の呼び出しにより入手されたオブジェクトの最も直感的な結合を提供するためのものです。

オブジェクトが、前述した処理の進行の中で使用される場合、それぞれの `dlopen(3DL)` が実施された順番は、結果として発生するシンボル結合には影響しません。ただし、複数のオブジェクトに共通の依存関係がある場合は、結果の結び付きは、`dlopen(3DL)` 呼び出しが実行された順番による影響を受けます。

次に、同じ共通依存関係を持つ共有オブジェクト `O.so.1` と `P.so.1` の例を示します。

```

$ ldd O.so.1
    Z.so.1 =>      ./Z.so.1
$ ldd P.so.1
    Z.so.1 =>      ./Z.so.1

```

この例では、prog アプリケーションは、各共有オブジェクトに `dlopen(3DL)` を使用しています。共有オブジェクト `Z.so.1` が、`O.so.1` と `P.so.1` 両方の共通依存関係であるため、図 3-3 のようにこの依存関係は 2 つの `dlopen(3DL)` 呼び出しに関連する両方のグループに割り当てられます。

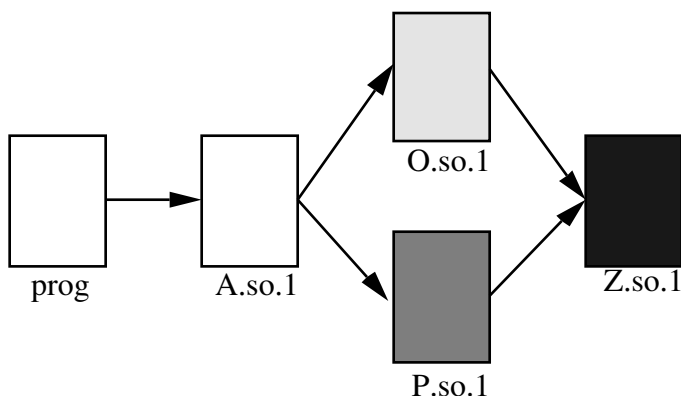


図 3-3 共通依存関係を伴う複数の `dlopen()` 要求

この結果、`O.so.1` と `P.so.1` の両方がシンボルの検索に `Z.so.1` を使用できます。ここで重要なのは、`dlopen(3DL)` の順序に限って言えば、`Z.so.1` も `O.so.1` と `P.so.1` の両方の中でシンボルを検索できることです。

そのため、`O.so.1` と `P.so.1` の両方に、`Z.so.1` の再配置に必要なシンボル `foo` の定義が組み込まれている場合、実際に発生する結び付きを予測することはできません。それは、この結び付きが `dlopen(3DL)` 呼び出しの順序の影響を受けるからです。シンボル `foo` の機能が、シンボルが定義されている 2 つの共有オブジェクト間で異なる場合、`Z.so.1` コードを実行したすべての結果は、アプリケーションの `dlopen(3DL)` の順序によって異なる可能性があります。

大域オブジェクトの定義

`dlopen(3DL)` によって入手されたオブジェクトへのデフォルトのローカルシンボルの可視性の割り当ては、モード引数に `RTLD_GLOBAL` フラグを指定することによって、大域に拡大ができます。このモードでは、`dlopen(3DL)` によって入手

されたオブジェクトは、シンボルを配置するためのワールドシンボルの検索範囲の指定された他のオブジェクトによって使用することができます。

また、`RTLD_GLOBAL` フラグが指定された `dlopen(3DL)` によって入手されたオブジェクトは、`dlopen(0)` を使用したシンボル検索にも使用できます (81ページの「追加オブジェクトの読み込み」を参照)。

注・ローカルシンボルの可視性を持つグループの構成要素が、他の大域シンボルの可視性を必要とするグループによって参照される場合、オブジェクトの可視性はローカルと大域の両方を連結したものになります。この後大域グループの参照が削除されても、この格上げされた属性はそのまま残ります。

グループの分離

`dlopen(3DL)` によって入手されたオブジェクトへのデフォルトのワールドシンボルの検索範囲の割り当ては、モード引数に `RTLD_GROUP` フラグを指定することによって、グループに縮小することができます。このモードでは、`dlopen(3DL)` によって入手されたオブジェクトは、そのオブジェクト固有のグループ内でしかシンボルの検索ができません。

オブジェクトがリンカーの `-B group` オプションを使用して構築された場合、オブジェクトには、割り当てられたグループシンボルの検索範囲があります。

注・グループ検索機能を持つグループの構成要素が、ワールド検索機能を必要とする他のグループによって参照された場合、オブジェクトの検索機能はグループとワールドが結合したものになります。この後ワールドグループの参照が削除されても、この格上げされた属性はそのまま残ります。

オブジェクト階層

`dlopen(3DL)` によって入手された最初のオブジェクトが、2番目のオブジェクトに `dlopen(3DL)` を使用した場合、両方のオブジェクトは1つのグループに割り当てられます。これにより、オブジェクトが互いにシンボルを配置し合うことを防ぐことができます。

実装の中には、最初のオブジェクトの場合、シンボルを2番目のオブジェクトの再配置用にエクスポートする必要がある場合もあります。この必要条件は、次の2つのメカニズムのいずれかによって満たすことができます。

- 最初のオブジェクトを 2 番目のオブジェクトの明示的な依存関係にする
- `dlopen(3DL)` を使用した 2 番目のオブジェクトに `RTLD_PARENT` モードフラグを使用する

最初のオブジェクトを 2 番目のオブジェクトの明示的な依存関係にした場合、これは 2 番目のオブジェクトのグループにも割り当てられます。そのため、最初のオブジェクトは、2 番目のオブジェクトの再配置に必要なシンボルも提供できます。

ただし、ほとんどのオブジェクトが 2 番目のオブジェクトに `dlopen(3DL)` を実行し、これらの最初のオブジェクトが、それぞれ 2 番目のオブジェクトの再配置を満足させる同じシンボルをエクスポートする必要がある場合、2 番目のオブジェクトには明示的な依存関係を割り当てることはできません。この場合、2 番目のオブジェクトの `dlopen(3DL)` モードは、`RTLD_PARENT` フラグを使用して補強できます。このフラグによって、2 番目のオブジェクトのグループが、明示的な依存関係が伝達されたのと同じ方法で、最初のオブジェクトに伝達されます。

上記の 2 つのテクニックには、1 つ異なる点があります。明示的な依存関係を指定する場合、その依存関係そのものは、2 番目のオブジェクトの `dlopen(3DL)` 依存関係ツリーの一部になるため、`dlopen(3DL)` を使用したシンボル検索が可能になります。`RTLD_PARENT` を使用して 2 番目のオブジェクトを入手する場合、最初のオブジェクトは、`dlopen(3DL)` を使用したシンボルの検索に使用できるようにはなりません。

注 - `dlopen(3DL)` によって 2 番目のオブジェクトが、大域シンボル可視性が指定された最初のオブジェクトから入手された場合、`RTLD_PARENT` モードは冗長で他に影響を与えることはありません。このような状態は、`dlopen(3DL)` がアプリケーションから呼び出されたとき、またはアプリケーションの中の依存関係の 1 つから呼び出されたときに多く発生します。

新しいシンボルの入手

プロセスは、`dlsym(3DL)` を使用して特定のシンボルのアドレスを入手できます。この関数は、ハンドルとシンボルをとり、呼び出し元にそのシンボルのアドレスを戻します。ハンドルは、次の方法でシンボルの検索を指示します。

- 指定されたオブジェクトの `dlopen(3DL)` から戻されたハンドルを使用すると、オブジェクトの依存関係ツリーからシンボルを入手できる

- 値が0のファイルの `dlopen(3DL)` から戻されたハンドルを使用すると、動的実行プログラム、任意の初期設定の依存関係、または `RTLD_GLOBAL` モードの `dlopen(3DL)` によって入手されたオブジェクトから、シンボルを入手できる
- 特別なハンドル `RTLD_DEFAULT` を使用すると、動的実行プログラム、任意の初期設定の依存関係、または呼び出し元と同じグループに属する `dlopen(3DL)` によって入手されたオブジェクトから、シンボルを入手できる (100ページの「機能のテスト」を参照)
- 特別なハンドル `RTLD_NEXT` を使用すると、次に関連するオブジェクトからシンボルを入手できる (101ページの「割り込み (interposition) の使用」を参照)

最初の例は、最も一般的なものです。ここでは、アプリケーションは、追加オブジェクトをそのアドレススペースに追加し、さらに `dlsym(3DL)` を使用して関数またはデータシンボルを配置します。次に、アプリケーションは、これらのシンボルを使用して、新しいオブジェクト内で提供されるサービスを呼び出します。たとえば、次のコードが組み込まれた `main.c` ファイルを取り上げてみます。

```

#include <stdio.h>
#include <dlfcn.h>

main()
{
    void * handle;
    int * dptr, (* fptr)();

    if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL) {
        (void) printf("dlopen: %s\n", dlerror());
        exit (1);
    }

    if (((fptr = (int (*)())dlsym(handle, "foo")) == NULL) ||
        ((dptr = (int *)dlsym(handle, "bar")) == NULL)) {
        (void) printf("dlsym: %s\n", dlerror());
        exit (1);
    }

    return ((*fptr)(*dptr));
}

```

ここで、シンボル `foo` と `bar` は、ファイル `foo.so.1` 内で検索された後で、このファイルに関連した依存関係が検索されます。次に、関数 `foo` は、単一の引数 `bar` によって `return()` ステートメントの一部として呼び出されます。

アプリケーション `prog` が、上記のファイル `main.c` を使用して構築された場合は、その最初の依存関係は次のものになります。

```
$ ldd prog
  libdl.so.1 => /usr/lib/libdl.so.1
  libc.so.1 => /usr/lib/libc.so.1
```

dlopen(3DL) 内に指定されたファイル名に値 0 がある場合、シンボル foo と bar は、prog、/usr/lib/libdl.so.1、/usr/lib/libc.so.1 の順番で検索されます。

ハンドルがシンボル検索を開始するルートを指示している場合は、この検索メカニズムは、76ページの「シンボルの検索」で説明したものと同一モデルに従います。

要求されたシンボルが配置されていない場合は、dlsym(3DL) は、NULL 値を返します。この場合、dlerror(3DL) を使用すると、失敗の真の理由を示すことができます。次に例を示します。

```
$ prog
dlsym: ld.so.1: main: fatal: bar: can't find symbol
```

ここでは、アプリケーション prog は、シンボル bar を配置できませんでした。

機能のテスト

特別なハンドル RTLD_DEFAULT を使用すると、アプリケーションは他のシンボルの存在をテストできます。シンボル検索は、呼び出しオブジェクトを再配置する場合に使用されるものと同一モデルに従います (87ページの「セキュリティ」を参照)。たとえば、アプリケーション prog に次のようなコードフラグメントが組み込まれているとします。

```
if ((fptr = (int (*)())dlsym(RTLD_DEFAULT, "foo")) != NULL)
    (*fptr)();
```

この場合 foo は、prog、/usr/lib/libdl.so.1、/usr/lib/libc.so.1 の順番で検索されます。このコードフラグメントが、図 3-2 の例で示すようにファイル B.so.1 に組み込まれていた場合、foo の検索は B.so.1 と C.so.1 でも継続して行われます。

このメカニズムによって、49ページの「ウィークシンボル」で説明した定義されていないウィークリファレンスの代わりに使用できる、パワフルで柔軟性のある代替機能が提供されます。

割り込み (interposition) の使用

特別なハンドル `RTLD_NEXT` を使用すると、アプリケーションは、シンボルの範囲内で次のシンボルを配置できます。たとえば、アプリケーション `prog` に次のようなコードフラグメントが組み込まれているとします。

```
if ((fptr = (int (*)())dlsym(RTLD_NEXT, "foo")) == NULL) {
    (void) printf("dlsym: %s\n", dlerror());
    exit (1);
}

return ((*fptr)());
```

この場合 `foo` は、`prog` に関連した共有オブジェクト内で、この場合は `/usr/lib/libdl.so.1` の次に `/usr/lib/libc.so.1` が検索されます。このコードフラグメントが、図 3-2 の例で示すように、ファイル `B.so.1` に組み込まれている場合は、`foo` は関連する共有オブジェクト `C.so.1` の中だけで検索されます。

`RTLD_NEXT` を使用することによって、シンボル割り込みを活用できます。たとえば、オブジェクト関数は、オブジェクトの前に付けて割り込みでき、これにより、元の関数の処理を補強できます。次のコードフラグメントが共有オブジェクト `malloc.so.1` 内にある場合、以下のようになります。

```
#include <sys/types.h>
#include <dlfcn.h>
#include <stdio.h>

void *
malloc(size_t size)
{
    static void * (* fptr)() = 0;
    char          buffer[50];

    if (fptr == 0) {
        fptr = (void * (*)())dlsym(RTLD_NEXT, "malloc");
        if (fptr == NULL) {
            (void) printf("dlopen: %s\n", dlerror());
            return (0);
        }
    }

    (void) sprintf(buffer, "malloc: %#x bytes\n", size);
    (void) write(1, buffer, strlen(buffer));
    return ((*fptr)(size));
}
```

この共有オブジェクトを、`malloc(3C)` が常駐するシステムライブラリ `/usr/lib/libc.so.1` の間に割り込むことにより、元の関数呼び出されて配置が完了する前に、この関数への呼び出しが次のように割り込まれます。

```
$ cc -o malloc.so.1 -G -K pic malloc.c
$ cc -o prog file1.o file2.o ..... -R. malloc.so.1
$ prog
malloc: 0x32 bytes
malloc: 0x14 bytes
.....
```

あるいは、次のように入力しても、上記のものと同じ割り込みを実行できます。

```
$ cc -o malloc.so.1 -G -K pic malloc.c
$ cc -o prog main.c
$ LD_PRELOAD=./malloc.so.1 prog
malloc: 0x32 bytes
malloc: 0x14 bytes
.....
```

注 - 割り込みテクニックを使用する場合、反復する可能性がある処理には注意が必要です。前の例では、`printf(3C)` を直接使用する代わりに `sprintf(3C)` を使用して診断メッセージを形成し、`printf(3C)` が `malloc(3C)` を使用する可能性があることによる反復を防いでいます。

動的実行プログラムまたはあらかじめ読み込まれたオブジェクト内で `RTLD_NEXT` を使用することにより、予測可能で有用な割り込みテクニックが使用できます。ただし、このテクニックを汎用オブジェクトの依存関係内で使用する場合には、実際に読み込まれる順番が必ず予測できるとは限らないため、注意が必要です (114ページの「依存関係の並べ替え」を参照)。

機能チェック

リンカーによって構築された実行プログラムファイルまたは共有オブジェクトファイルは、新しい実行時リンカー機能を要求できます。実行時リンカーが実行に必要な実行時機能をサポートしているかどうかを検査するために、アプリケーション `.init` セクションによって関数 `_check_rtld_feature()` が呼び出されます。現在チェックされる機能については、表 7-47 を参照してください。

デバッグガイド

Solaris リンカーには、デバッグライブラリが付いていて、これを使用すると実行時のリンクプロセスをより詳細に監視できます。このライブラリは、アプリケーションと依存関係の実行を理解したり、デバッグする場合に役立ちます。これはビジュアルエイドで、このライブラリを使用して表示される情報のタイプは定数のままであると予期されますが、この情報の正確な形式は、リリースごとにわずかに変更される場合があります。

実行時リンカーをよく理解していないと、デバッグ出力のなかには理解できないものがある可能性がありますが、一般的には、興味深い面が多いといえます。

デバッグは、環境変数 `LD_DEBUG` を使用して実行します。すべてのデバッグの出力は、接頭辞としてプロセス識別子を持っていて、デフォルトごとに、標準的なエラーに対して送信されます。この環境変数は、1 つまたは複数のトークンを使用して、必要なデバッグタイプを示す必要があります。

このデバッグオプションとともに使用できるトークンは、`LD_DEBUG=help` を使って表示できます。どの動的実行プログラムを使用しても、この情報を要求することができます。この場合、プロセス自体が終了した後でこの情報が表示されます。次に例を示します。

```
$ LD_DEBUG=help prog
11693:
11693:      For debugging the runtime linking of an application:
11693:          LD_DEBUG=token1,token2 prog
11693:      enables diagnostics to the stderr.  The additional
11693:      option:
11693:          LD_DEBUG_OUTPUT=file
11693:      redirects the diagnostics to an output file created
11593:      using the specified name and the process id as a
11693:      suffix.  All diagnostics are prepended with the
11693:      process id.
11693:
11693:
11693: basic      provide basic trace information/warnings
11693: bindings   display symbol binding; detail flag shows
11693:             absolute:relative addresses
11693: detail     provide more information in conjunction with other
11693:             options
11693: files      display input file processing (files and libraries)
11693: help       display this help message
11693: libs       display library search paths
11693: move       display move section processing
11693: reloc     display relocation processing
11693: symbols    display symbol table processing;
```

(続く)

```

11693:          detail flag shows resolution and linker table addition
11693: versions  display version processing
11693: audit      display runtime link-audit processing

```

注 - これは一例で、実行時リンカーに有効なオプションを示しています。正確なオプションについては、リリースごとに異なる場合があります。

環境変数 `LD_DEBUG_OUTPUT` を使用すると、出力ファイルを標準エラーの代わりに使用するように指定できます。出力ファイルの名前には、接尾辞としてプロセス ID が付きます。

セキュアアプリケーションのデバッグは実行できません。

実行時に発生するシンボル結合の表示機能は、最も有効なデバッグオプションの 1 つです。たとえば、2 つのローカル共有オブジェクト上に依存関係を持つ、非常に単純な動的実行プログラムを取り上げてみます。

```

$ cat bar.c
int bar = 10;
$ cc -o bar.so.1 -Kpic -G bar.c

$ cat foo.c
foo(int data)
{
    return (data);
}
$ cc -o foo.so.1 -Kpic -G foo.c

$ cat main.c
extern int    foo();
extern int    bar;

main()
{
    return (foo(bar));
}
$ cc -o prog main.c -R/tmp:. foo.so.1 bar.so.1

```

実行時シンボルは、`LD_DEBUG=bindings` を設定することによって表示されます。


```
$ LD_DEBUG=bindings prog
11753: .....
11753: binding file=prog to file=./bar.so.1: symbol bar
11753: .....
11753: transferring control: prog
11753: .....
11753: binding file=prog to file=./foo.so.1: symbol foo
11753: .....
```

ここでは、データの再配置に必要なシンボル `bar` が、アプリケーションが制御を受け取る前に結合されます。これに対して、関数の再配置に必要なシンボル `foo` は、関数が最初に呼び出されたときに、アプリケーションが制御を受け取った後で結合されます。これは、レイジー結合のデフォルトモードを示しています。環境変数 `LD_BIND_NOW` が設定されている場合、シンボル結合はすべて、アプリケーションが制御を受け取る前に実行されます。

現在の結合ロケーションの実際の相対アドレスに関する追加情報は、`LD_DEBUG=bindings,detail` を設定すると入手できます。

実行時リンカーが、関数の再配置を実行すると、関数 `.plt` に関連したデータも書き換えられるため、この後に発生する呼び出しは、関数に直接送信されます。環境変数 `LD_BIND_NOT` は、あらゆる値に設定されて、このデータの更新を防ぐことができます。この変数を、詳細結合に対するデバッグ要求とともに使用すると、関数結合すべての完全な実行時アカウントを入手できます。この組み合わせによる出力は、膨大なものになるため、アプリケーションのパフォーマンスは低下します。

この他にも、実行時のさまざまな検索パスの使用状況が表示できます。たとえば、依存関係の配置に使用される検索パスのメカニズムは、次のように `LD_DEBUG=libs` を設定して表示できます。

```
$ LD_DEBUG=libs prog
11775:
11775: find object=foo.so.1; searching
11775:  search path=/tmp:. (RPATH from file prog)
11775:  trying path=/tmp/foo.so.1
11775:  trying path=./foo.so.1
11775:
11775: find object=bar.so.1; searching
11775:  search path=/tmp:. (RPATH from file prog)
11775:  trying path=/tmp/bar.so.1
11775:  trying path=./bar.so.1
11775: .....
```

ここでは、アプリケーション `prog` 内に記録された実行パスは、2つの依存関係 `foo.so.1` と `bar.so.1` の検索に影響を与えます。

これと同様の方法で、各シンボルを検索する検索パスは、LD_DEBUG=symbols を設定して表示できます。これを bindings 要求と結合すると、次のように、シンボル再配置プロセスが完全に表示されます。

```
$ LD_DEBUG=bindings,symbols
11782: .....
11782: symbol=bar; lookup in file=./foo.so.1 [ ELF ]
11782: symbol=bar; lookup in file=./bar.so.1 [ ELF ]
11782: binding file=prog to file=./bar.so.1: symbol bar
11782: .....
11782: transferring control: prog
11782: .....
11782: symbol=foo; lookup in file=prog [ ELF ]
11782: symbol=foo; lookup in file=./foo.so.1 [ ELF ]
11782: binding file=prog to file=./foo.so.1: symbol foo
11782: .....
```

注 - 上記の例では、シンボル bar は、アプリケーション prog 内では検索されません。これは、コピーの再配置を処理するときに行なわれる最適化が原因です (この再配置タイプの詳細については、131ページの「コピー再配置」を参照)。

共有オブジェクト

概要

共有オブジェクトは、リンカーによって作成される出力形式の 1 つであり、`-G` オプションを指定して生成されます。次に例を示します。

```
$ cc -o libfoo.so.1 -G -K pic foo.c
```

ここで、共有オブジェクト `libfoo.so.1` は、入力ファイル `foo.c` から生成されます。

注 - これは、共有オブジェクトを生成する非常に簡単な例です。通常は、追加オプションを使用することをお勧めします。これらのオプションについては、以下の節で説明します。

共有オブジェクトとは、1 つまたは複数の再配置可能なオブジェクトから生成される表示できないユニットです。共有オブジェクトは、動的実行可能ファイルと結合して実行可能プロセスを形成することができます。共有オブジェクトは、その名前が示すように、複数のアプリケーションによって共有できます。このように共有オブジェクトの影響力は非常に大きくなる可能性があるため、この章では、リンカーのこの出力形式について前の章よりも詳しく説明します。

共有オブジェクトを動的実行可能ファイルや他の共有オブジェクトに結合するには、まず共有オブジェクトが必要な出力ファイルのリンク編集に使用可能でなければなりません。このリンク編集で、入力共有オブジェクトはすべて、作成中の出力

ファイルの論理アドレス空間に追加された場合のように解釈されます。つまり、共有オブジェクトのすべての機能が、出力ファイルにとって使用可能になります。

これらの共有オブジェクトは、この出力ファイルの依存関係になります。出力ファイル内には、この依存関係を記述するための少量の登録情報が保持されます。実行時リンカーは、この情報を解釈し、実行可能プロセス作成の一部として、これらの共有オブジェクトの処理を完了します。

次の節では、コンパイル環境と実行時環境内の共有オブジェクトの使用法について詳しく説明します (これらの環境については、20ページの「共有オブジェクト」を参照)。ここでは、共有オブジェクトの効率を最大にするための手法とともに、これらの環境内における共有オブジェクトの使用の調整に役立つ事項について説明します。

命名規約

リンカーも実行時リンカーも、ファイル名によってファイルを解釈しません。ファイルはすべて検査されて、その ELF タイプが判定されます (203ページの「ELF ヘッダー」を参照)。この情報から、ファイルの処理条件が推定されます。ただし、共有オブジェクトは通常、コンパイル環境または実行時環境のどちらの一部として使用されるかによって、2つの命名規約のうちどちらかに従います。

共有オブジェクトは、コンパイル環境の一部として使用される場合、リンカーによって読み取られて処理されます。これらの共有オブジェクトは、リンカーに渡されるコマンド行の一部で明示的なファイル名によって指定できますが、リンカーのライブラリ検索機能を利用するために `-l` オプションを使用の方が一般的です (31ページの「共有オブジェクトの処理」を参照)。

共有オブジェクトをこのリンカー処理に使用できるようにするには、接頭辞 `lib` と接尾辞 `.so` によって共有オブジェクトを指定する必要があります。たとえば、`/usr/lib/libc.so` は、コンパイル環境に使用できる標準 C ライブラリの共有オブジェクト表現です。規則によって、64 ビットの共有オブジェクトは、64 と呼ばれる `lib` ディレクトリのサブディレクトリに置かれます。たとえば、`/usr/lib/libc.so.1` がある場合、64 ビットに対応するオブジェクトは、`/usr/lib/64/libc.so.1` にあります。

共有オブジェクトは、実行時環境の一部として使用される場合、実行時リンカーによって読み取られて処理されます。一連のソフトウェアリリースでは、共有オブジェクトのエクスポートされたインタフェースでの変更が可能でなければならない

場合があります。このインタフェースの変更は、バージョンアップされたファイル名として共有オブジェクトを提供することによって予測可能にして、サポートできます。

バージョン付きファイル名は、通常、`.so` 接尾辞の後にバージョン番号が続くという形式をとります。たとえば、`/usr/lib/libc.so.1` は、実行時環境で使用可能な標準 C ライブラリのバージョン 1 の共有オブジェクト表示です。

共有オブジェクトが、コンパイル環境内での使用をまったく目的としていない場合は、従来の `lib` 接頭辞がその名前から削除されることがあります。このカテゴリに属する共有オブジェクトの例には、`dlopen(3DL)` だけに使用されるオブジェクトがあります。実際のファイルタイプを示すために、接尾辞 `.so` は付けた方が望ましく、一連のソフトウェアリリースで共有オブジェクトの正しい結合を行うためにはバージョン番号も必要です。

注 - `dlopen(3DL)` で使用される共有オブジェクト名は通常、単純ファイル名として表わされます。つまり、名前に `'/'` が付きません。この規則によって、実行時リンカーは、実際のファイルを検索するための規則を自由に使用できます (詳細は、81 ページの「追加オブジェクトの読み込み」を参照)。

第 5 章では、一連のソフトウェアリリースで共有オブジェクトインタフェースのバージョンアップという概念についてさらに詳しく説明します。また、コンパイル環境と実行時環境の両方で使用される共有オブジェクト間の命名規約を調整するためのメカニズムについても説明します。ただし最初に、共有オブジェクトが各自の実行時名を記録するためのメカニズムを説明しておきます。

共有オブジェクト名の記録

動的実行可能ファイルまたは共有オブジェクトでの「依存関係」の記録は、デフォルトでは、関連する共有オブジェクトがリンカーによって参照されるときにファイル名になります。たとえば、次の動的実行可能ファイルは、同じ共有オブジェクト `libfoo.so` に対して構築されますが、同じ依存関係の解釈は異なります。

```
$ cc -o ../tmp/libfoo.so -G foo.o
$ cc -o prog main.o -L../tmp -lfoo
$ dump -Lv prog | grep NEEDED
[1]      NEEDED   libfoo.so

$ cc -o prog main.o ../tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
```

(続く)

```
[1]      NEEDED    ../tmp/libfoo.so

$ cc -o prog main.o /usr/tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]      NEEDED    /usr/tmp/libfoo.so
```

上記の例が示すように、依存関係を記録するこのメカニズムでは、コンパイル手法の違いによって不一致が生じる可能性があります。また、リンク編集に参照される共有オブジェクトの位置が、インストールされたシステムでの共有オブジェクトの最終的な位置と異なる場合があります。

依存関係を指定するより一貫した手法として、共有オブジェクトは、それぞれの内部にファイル名を記録できます。共有オブジェクトは、このファイル名によって実行時に参照されます。

共有オブジェクトのリンク編集で、`-h` オプションを使用すると、その実行時名を共有オブジェクト自体に記録できます。次に例を示します。

```
$ cc -o ../tmp/libfoo.so -G -K pic -h libfoo.so.1 foo.c
```

ここで、共有オブジェクトの実行時名 `libfoo.so.1` は、ファイル自体に記録されます。この識別名は `soname` と呼ばれ、その記録は `dump(1)` を使用し、`SONAME` タグを持つエントリを参照して表示できます。次に例を示します。

```
$ dump -Lvp ../tmp/libfoo.so

../tmp/libfoo.so:
[INDEX] Tag      Value
[1]      SONAME   libfoo.so.1
.....
```

リンカーが `soname` を含む共有オブジェクトを処理する場合、生成中の出力ファイル内に依存関係として記録されるのはこの名前です。

したがって、前の例から動的実行可能ファイル `prog` を作成しているときに、この新しいバージョンの `libfoo.so` が使用されると、実行可能ファイルを作成するための3つの方式すべてによって同じ依存関係が記録されます。

```

$ cc -o prog main.o -L../tmp -lfoo
$ dump -Lv prog | grep NEEDED
[1]      NEEDED   libfoo.so.1

$ cc -o prog main.o ../tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]      NEEDED   libfoo.so.1

$ cc -o prog main.o /usr/tmp/libfoo.so
$ dump -Lv prog | grep NEEDED
[1]      NEEDED   libfoo.so.1

```

上記の例では、`-h` オプションは、単純 (simple) ファイル名を指定するために使用されます。つまり、名前に `'/'` が付きません。この規則では、実行時リンカーが実際のファイルを検索するための規則を自由に使用できるため、これを使用することをお勧めします (詳細は、70ページの「共有オブジェクトの依存関係の配置」を参照)。

アーカイブへの共有オブジェクトの取り込み

共有オブジェクトに `soname` を記録するメカニズムは、共有オブジェクトがアーカイブライブラリから処理される場合に重要です。

アーカイブは、1つまたは複数の共有オブジェクトから構築し、動的実行可能ファイルまたは共有オブジェクトを生成するために使用できます。共有オブジェクトは、リンク編集の条件を満たすためにアーカイブから抽出できます (アーカイブ抽出条件の詳細は、30ページの「アーカイブ処理」を参照)。ただし、作成中の出力ファイルに連結される再配置可能オブジェクトの処理とは違って、アーカイブから抽出された共有オブジェクトは、すべて依存関係として記録されます。

アーカイブ構成要素の名前はリンカーによって構築されて、アーカイブ名とアーカイブ内のオブジェクトの連結になります。次に例を示します。

```

$ cc -o libfoo.so.1 -G -K pic foo.c
$ ar -r libfoo.a libfoo.so.1
$ cc -o main main.o libfoo.a
$ dump -Lv main | grep NEEDED
[1]      NEEDED   libfoo.a(libfoo.so.1)

```

この連結名を持つファイルが実行時に存在することはほとんどないため、共有オブジェクト内に `soname` を与える方法が、依存関係の有意な実行時ファイル名を生成する唯一の手段です。

注 - 実行時リンカーは、アーカイブからオブジェクトを抽出しません。したがって、上記の例では、必要な共有オブジェクト依存関係をアーカイブから抽出して、実行時環境で使用できるようにする必要があります。

記録名の衝突

共有オブジェクトが実行可能ファイルまたは別の共有オブジェクトを作成するために使用される場合、リンカーは、いくつかの整合性検査を実行して、出力ファイル内に記録される依存関係名すべてが一意になるように保証します。

依存関係名の衝突は、リンク編集への入力ファイルとして使用される 2 つの共有オブジェクトがどちらも同じ `soname` を含む場合に発生する可能性があります。次に例を示します。

```
$ cc -o libfoo.so -G -K pic -h libsame.so.1 foo.c
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c
$ cc -o prog main.o -L. -lfoo -lbar
ld: fatal: file ./libbar.so: recording name 'libsame.so.1' \
      matches that provided by file ./libfoo.so
ld: fatal: File processing errors. No output written to prog
```

記録された `soname` を持たない共有オブジェクトのファイル名が、同じリンク編集に使用された別の共有オブジェクトの `soname` に一致する場合にも同様にエラー条件が発生します。

生成中の共有オブジェクトの実行時名が、その依存関係の 1 つに一致する場合にも、リンカーは名前の衝突を報告します。次に例を示します。

```
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c -L. -lfoo
ld: fatal: file ./libfoo.so: recording name 'libsame.so.1' \
      matches that supplied with -h option
ld: fatal: File processing errors. No output written to libbar.so
```


依存関係を持つ共有オブジェクト

これまでこの章で示した例のほとんどは、共有オブジェクトの依存関係が動的実行可能ファイル内でどのように維持されるかを示していますが、共有オブジェクトが独自の依存関係を持つことは非常に一般的です (詳細は、31ページの「共有オブジェクトの処理」を参照)。

70ページの「実行時リンカーによって検索されるディレクトリ」では、共有オブジェクトの依存関係を検索するために実行時リンカーが使用する検索規則について説明しています。共有オブジェクトがデフォルトディレクトリの `/usr/lib` (32ビットオブジェクトの場合)、または `/usr/lib/64` (64ビットオブジェクトの場合) がないときは、実行時リンカーに対して検索場所を明示的に指示する必要があります。この種の条件を指示するために優先されるメカニズムは、リンカーの `-R` オプションを使用して、依存関係を持つオブジェクトに「実行パス」を記録するというものです。次に例を示します。

```
$ cc -o libbar.so -G -K pic bar.c
$ cc -o libfoo.so -G -K pic foo.c -R/home/me/lib -L. -lbar
$ dump -Lv libfoo.so

libfoo.so:

**** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag      Value
[1]      NEEDED   libbar.so
[2]      RPATH    /home/me/lib
.....
```

ここで、共有オブジェクト `libfoo.so` は、`libbar.so` に対する依存関係を持ちます。これは、実行時にディレクトリ `/home/me/lib` にあるものと予期されますが、ない場合はデフォルト位置の `/usr/lib` にあるものと予期します。

共有オブジェクトでは、依存関係を検索するために必要な実行パスすべてを指定する必要があります。動的実行可能ファイルに指定された実行パスはすべて、動的実行可能ファイルの依存関係を検索するためにだけ使用されます。これは、共有オブジェクトの依存関係を検索するために使用されることはありません。

これに対して、環境変数 `LD_LIBRARY_PATH` は、より大域的な適用範囲を持ちます。この変数を使用して指定されたパス名はすべて、実行時リンカーによって、すべての共有オブジェクト依存関係を検索するために使用されます。この環境変数

は、実行時リンカーの検索パスに影響を与える一時的なメカニズムとして便利ですが、製品版ソフトウェアではできるだけ使用しないようにしてください (詳細は、70ページの「実行時リンカーによって検索されるディレクトリ」を参照)。

依存関係の並べ換え

このマニュアルで示すほとんどの例では、動的実行可能ファイルと共有オブジェクトの依存関係は、一意の比較的単純なものとして描かれています (依存共有オブジェクトの幅優先順については、70ページの「共有オブジェクトの依存関係の配置」を参照)。これらの例では、共有オブジェクトがプロセスのアドレス空間に取り込まれたときの並べ換えが、非常にわかりやすく予測可能なものに見えるかもしれません。

しかし、動的実行可能ファイルと共有オブジェクトが同じ共通の共有オブジェクトに対して依存関係を持つ場合は、オブジェクトが処理される順序が予測困難になる可能性があります。

たとえば、共有オブジェクトの開発者が、次の依存関係を持つ `libfoo.so.1` を生成したものと想定します。

```
$ ldd libfoo.so.1
    libA.so.1 =>      ./libA.so.1
    libB.so.1 =>      ./libB.so.1
    libC.so.1 =>      ./libC.so.1
```

この共有オブジェクトを使用して動的実行可能ファイル `prog` を作成し、`libC.so.1` に対してさらに明示的な依存関係を定義すると、共有オブジェクトの順序は次のようになります。

```
$ cc -o prog main.c -R. -L. -lc -lfoo
$ ldd prog
    libC.so.1 =>      ./libC.so.1
    libfoo.so.1 =>    ./libfoo.so.1
    libA.so.1 =>      ./libA.so.1
    libB.so.1 =>      ./libB.so.1
```

したがって、共有オブジェクト `libfoo.so.1` の開発者がその依存関係の処理順序にある条件を設定しても、動的実行可能ファイル `prog` を構築した場合には、設定した条件は処理順序に影響を与えません。

シンボルの割り込み (76ページの「シンボルの検索」、92ページの「シンボル検索」、および 101ページの「割り込み (interposition) の使用」を参照) と `.init` セクションの処理 (103ページの「デバッグエイド」を参照) を特に重要視する開発者は、共有オブジェクトの処理順序でのこのような変更の可能性に注意する必要があります。

フィルタとしての共有オブジェクト

フィルタとは、代替共有オブジェクトへの間接参照を提供するために使用される特殊な形式の共有オブジェクトのことをいいます。2つの形式の共有オブジェクトフィルタがあります。

- 標準フィルタ
- 補助フィルタ

標準フィルタは、基本的に単一のシンボルテーブルからなり、実行時環境からコンパイル環境を抽象化するメカニズムを提供します。このフィルタを使用するリンク編集は、フィルタ自体によって提供されるシンボルを参照しますが、シンボル参照の解釈は、実行時に代替ソースから提供されます。

標準フィルタは、リンカーの `-F` フラグによって識別されます。このフラグは、実行時にシンボル参照を与える共有オブジェクトを示す関連ファイル名をとります。この共有オブジェクトは、フィルティー (フィルタ対象) と呼ばれます。`-F` フラグを複数回使用すると、複数のフィルティーを記録できます。

フィルティーを実行時理できないか、またはフィルタによって定義されたシンボルがフィルティー内に見つからない場合、そのフィルタは無視されて、シンボル解決は次に関連する依存関係に続けられます。

補助フィルタも同様のメカニズムを備えていますが、フィルタ自体にそのシンボルに対応する実装が含まれます。フィルタを使用するリンク編集ではフィルタ自体によって提供されたシンボルを参照しますが、シンボル参照の実装は、実行時に代替ソースから提供できます。

補助フィルタは、リンカーの `-f` フラグを使用して識別されます。このフラグは、実行時にシンボルを与えるために使用できる共有オブジェクトを示す関連ファイル名をとります。この共有オブジェクトは、フィルティーと呼ばれます。`-f` フラグを複数回使用すると、複数のフィルティーを記録できます。

フィルティーを実行時に処理できないか、またはフィルティー内にフィルタが見つからないと、フィルタ内のシンボルの実装が使用されます。

標準フィルタの生成

標準フィルタを生成するには、まずフィルティー `libbar.so.1` を定義し、それに対してこのフィルタ手法を適用します。このフィルティーは、いくつかの再配置可能オブジェクトから構築される場合があります。これらのオブジェクトの1つは、ファイル `bar.c` から発生し、シンボル `foo` と `bar` を与えます。

```
$ cat bar.c
char * bar = "bar";

char * foo()
{
    return("defined in bar.c");
}
$ cc -o libbar.so.1 -G -K pic .... bar.c ....
```

標準フィルタ `libfoo.so.1` は、シンボル `foo` と `bar` に対して生成されて、フィルティー `libbar.so.1` への関連付けを示します。次に例を示します。

```
$ cat foo.c
char * bar = 0;

char * foo(){

$ LD_OPTIONS='-F libbar.so.1' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ ln -s libfoo.so.1 libfoo.so
$ dump -Lv libfoo.so.1 | egrep "SONAME|FILTER"
[1] SONAME libfoo.so.1
[2] FILTER libbar.so.1
```

注 - ここで、環境変数 `LD_OPTIONS` は、このコンパイラドライバが `-F` オプションをそれ自体のオプションの1つとして解釈しないようにするために使用されています。

リンカーは、標準フィルタ `libfoo.so.1` を参照して動的実行可能ファイルまたは共有オブジェクトを作成する場合、シンボル解決中にフィルタシンボルテーブルからの情報を使用します (詳細は、41ページの「シンボル解析」を参照)。

実行時に、フィルタのシンボルを参照すると、必ずフィルタ libbar.so.1 がさらに読み込まれます。実行時リンカーは、このフィルタを使用し、libfoo.so.1 によって定義されたシンボルを解釈処理します。

たとえば、次の動的実行可能ファイル prog は、シンボル foo と bar を参照します。これらは、リンク編集時にフィルタ libfoo.so.1 から解釈処理されます。

```
$ cat main.c
extern char * bar, * foo();

main(){
    (void) printf("foo() is %s: bar=%s\n", foo(), bar);
}
$ cc -o prog main.c -R. -L. -lfoo
$ prog
foo() is defined in bar.c: bar=bar
```

動的実行可能ファイル prog を実行すると、関数 foo() とデータ項目 bar が、フィルタ libfoo.so.1 からではなく、フィルタ libbar.so.1 から取得されます。

注 - この例では、フィルタ libbar.so.1 がフィルタ libfoo.so.1 に一意に関連付けられています。このため、prog を実行した結果読み込まれる可能性がある他のオブジェクトからのシンボル参照を満たすために使用することができません。

標準フィルタは、既存の共有オブジェクトのサブセットインタフェース、または多数の既存の共有オブジェクトに及ぶインタフェースグループを定義するためのメカニズムとなります。Solaris で使用されるフィルタには、/usr/lib/libsys.so.1 と /usr/lib/libdl.so.1 の 2 つがあります。

最初のフィルタは、標準 C ライブラリ /usr/lib/libc.so.1 のサブセットになります。このサブセットは、準拠するアプリケーションによってインポートしなければならない C ライブラリ内の ABI に準拠する関数とデータ項目を表わします。

2 つめのフィルタは、実行時リンカー自体へのユーザーインタフェースを定義します。このインタフェースは、コンパイル環境で (libdl.so.1 から) 参照されるシンボルと、実行時環境内で (ld.so.1 から) 作成される実際の実装結合間の抽象化を提供します。

複数のフィルタを使用するフィルタの一例として、/usr/lib/libxnet.so.1 があります。このライブラリは、/usr/lib/libsocket.so.1、/usr/lib/libnsl.so.1、および /usr/lib/libc.so.1 から、ソケットと XTI インタフェースを提供します。

標準フィルタ内のコードは実行時に参照されないため、フィルタ内に定義された関数に内容を加えても意味がありません。フィルタコードが再配置を必要とする場合がありますが、実行時にそのフィルタを処理すると不要なオーバーヘッドが生じます。関数は空のルーチンとして定義するか、直接 `mapfile` から定義してください (51ページの「追加シンボルの定義」を参照)。

フィルタ内にデータシンボルを生成するときにも注意が必要です。データ項目は必ず初期設定して、動的実行可能ファイルから参照されるように保証する必要があります。

リンカーによって実行されるより複雑なシンボル解釈処理の中には、シンボルサイズを含むシンボルの属性に関する知識を必要とするものがあります (詳細については、41ページの「シンボル解析」を参照)。このため、フィルタ内のシンボルの属性がフィルタ内のシンボルの属性と一致するようにシンボルを生成することをお勧めします。これにより、リンク編集処理では、実行時に使用されるシンボル定義と互換性のある方法でフィルタが解析されます。

注・リンカーは、最初に入力された再配置可能ファイルの ELF クラスを使って、作成するオブジェクトのクラスを管理します (22ページの「32 ビットおよび 64 ビット環境」を参照)。64 ビットフィルタをマップファイルだけから作成するには、リンカーに `-64` オプションが必要です。

補助フィルタの生成

補助フィルタの作成方法は、標準フィルタの場合と基本的に同じです (詳細については、116ページの「標準フィルタの生成」を参照)。まず、このフィルタ手法を適用するフィルタ `libbar.so.1` を定義します。このフィルタは、いくつかの再配置可能オブジェクトから構築される場合があります。これらのオブジェクトの 1 つは、ファイル `bar.c` から発生し、シンボル `foo` を提供します。

```
$ cat bar.c
char * foo()
{
    return("defined in bar.c");
}
$ cc -o libbar.so.1 -G -K pic .... bar.c ....
```

補助フィルタ `libfoo.so.1` が、シンボル `foo` と `bar` に対して生成されて、フィルタ `libbar.so.1` への関連付けを示します。次に例を示します。

```
$ cat foo.c
char * bar = "foo";

char * foo()
{
    return ("defined in foo.c");
}
$ LD_OPTIONS='-f libbar.so.1' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ ln -s libfoo.so.1 libfoo.so
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"
[1] SONAME libfoo.so.1
[2] AUXILIARY libbar.so.1
```

注 - ここで、環境変数 `LD_OPTIONS` は、このコンパイラドライバが `-f` オプションをそれ自体のオプションの1つとして解釈しないようにするために使用されています。

リンカーは、補助フィルタ `libfoo.so.1` を参照して動的実行可能ファイルまたは共有オブジェクトを作成する場合、シンボル解決中、フィルタシンボルテーブルの情報を使用します (詳細については、41ページの「シンボル解析」を参照)。

実行時にフィルタのシンボルを参照すると、フィルタ `libbar.so.1` が検索されます。このフィルタが見つかったら、実行時リンカーは、このフィルタを使用して、`libfoo.so.1` によって定義されたすべてのシンボルを解釈処理します。このフィルタが見つからないか、またはフィルタにフィルタからのシンボルがない場合は、フィルタ内のシンボルの元の値が使用されます。

たとえば、次の動的実行可能ファイル `prog` は、シンボル `foo` と `bar` を参照します。これらのシンボルは、フィルタ `libfoo.so.1` からのリンク編集集中に解釈処理されます。

```
$ cat main.c
extern char * bar, * foo();

main(){
    (void) printf("foo() is %s: bar=%s\n", foo(), bar);
}
$ cc -o prog main.c -R. -L. -lfoo
$ prog
```

(続く)

```
foo() is defined in bar.c: bar=foo
```

動的実行可能ファイル `prog` を実行すると、関数 `foo()` は、フィルタ `libfoo.so.1` からではなく、フィルティー `libbar.so.1` から取得されます。ただし、データ項目 `bar` は、フィルタ `libfoo.so.1` から取得されます。このシンボルは、フィルティー `libbar.so.1` に代替定義を持たないためです。

補助フィルタは、既存の共有オブジェクトの代替インタフェースを定義するメカニズムとなります。このメカニズムは Solaris で使用されて、プラットフォーム固有の共有オブジェクト内に最適化された機能を提供します。例は、362ページの「命令セット固有の共有オブジェクト」および 364ページの「プラットフォーム固有の共有オブジェクト」を参照してください。

フィルティーの処理

実行時リンカーによるフィルタ処理は、フィルタ内のシンボルへの参照が生じるまで、フィルティーの読み込みを延期します。この実装は、各フィルティーに対して必要に応じて `dlopen(3DL)` を実行するフィルタに似ています。この実装は、`ldd(1)` などのツールによって生じる可能性がある、依存関係の報告における違いの原因となるものです。

フィルタを作成して、そのフィルティーを実行時に即時処理する場合は、リンカーの `-z loadfltr` オプションを使用できます。また、プロセス内のフィルタすべての即時処理は、どの値にも環境変数 `LD_LOADFLTR` を設定することによってトリガーすることもできます。

性能に関する考慮事項

共有オブジェクトは、同じシステム内の複数のアプリケーションで使用できます。したがって、共有オブジェクトの性能の影響は、それを使用するアプリケーションだけでなく、システム全体に至るほどに大きくなる可能性があります。

共有オブジェクト内の実際のコードは、実行中のプロセスの性能に直接影響しますが、ここでは共有オブジェクト自体の実行時処理に焦点を絞って性能の問題を説明

します。次の節では、再配置によるオーバーヘッドとともに、テキストサイズや純度 (purity) などの面についても見ながら、この処理について詳しく説明します。

役に立つツール

性能について述べる前に、使用可能なツールと ELF ファイルの内容の解析に対するその使用法を理解しておく役立ちます。

ELF ファイルに定義されているセクションまたはセグメントのいずれかのサイズに対する参照は、頻繁に行われます (ELF 形式の詳細については、第7章を参照)。ファイルのサイズは、`size(1)` コマンドを使用して表示できます。次に例を示します。

```
$ size -x libfoo.so.1
59c + 10c + 20 = 0x6c8

$ size -xf libfoo.so.1
..... + 1c(.init) + ac(.text) + c(.fini) + 4(.rodata) + \
..... + 18(.data) + 20(.bss) .....
```

最初の例は、SunOS™ オペレーティングシステムの以前のリリースから引き続き使用されてきたカテゴリである、共有オブジェクトテキスト、データ、および bss のサイズを示します。ELF 形式は、データをセクションに編成することによって、ファイル内のデータを表現するためのより精密な方法を提供します。2 番目の例は、ファイルの読み込み可能な各セクションのサイズを表示しています。

セクションは、セグメントと呼ばれる単位に割り当てられます。セグメントの一部は、ファイルの部分がメモリーにどのように割り当てられるかを記述します (`mmap(2)` のマニュアルページを参照)。これらの読み込み可能なセグメントは、`dump(1)` コマンドを使用して、LOAD エントリを調べることによって表示できます。次に例を示します。

```
$ dump -ov libfoo.so.1

libfoo.so.1:
***** PROGRAM EXECUTION HEADER *****
Type      Offset      Vaddr      Paddr
Filesz    Memsz      Flags      Align

LOAD      0x94        0x94        0x0
0x59c     0x59c      r-x        0x10000
```

(続く)

LOAD	0x630	0x10630	0x0
0x10c	0x12c	rwX	0x10000

ここで、共有オブジェクト `libfoo.so.1` には、一般にテキストセグメントおよびデータセグメントと呼ばれる 2 つの読み込み可能なセグメントがあります。テキストセグメントは、その内容の読み取りと実行 (`r-x`) も可能になるように割り当てられます。これに対して、データセグメントは、その内容の変更 (`rwX`) も可能になるように割り当てられます。データセグメントのメモリーサイズ (`Memsz`) が、ファイルサイズ (`Filesz`) とは異なることに注意してください。この違いは、実際にはデータセグメントの一部であり、セグメントが読み込まれると動的に作成される `.bss` セクションを示すものです。

ただし、通常プログラマは、関数とデータ要素をそのコード内に定義するシンボルの点からファイルについて考えます。これらのシンボルは、`nm(1)` を使用して表示できます。次に例を示します。

```
$ nm -x libfoo.so.1

[Index]  Value          Size          Type Bind Other Shndx  Name
.....
[39]     |0x00000538|0x00000000|FUNC |GLOB |0x0 |7      |_init
[40]     |0x00000588|0x00000034|FUNC |GLOB |0x0 |8      |foo
[41]     |0x00000600|0x00000000|FUNC |GLOB |0x0 |9      |_fini
[42]     |0x00010688|0x00000010|OBJT |GLOB |0x0 |13     |data
[43]     |0x0001073c|0x00000020|OBJT |GLOB |0x0 |16     |bss
.....
```

シンボルを含むセクションは、シンボルテーブルのセクションインデックス (`Shndx`) フィールドを参照し、`dump(1)` を使用してファイル内のセクションを表示することによって判定できます。次に例を示します。

```
$ dump -hv libfoo.so.1

libfoo.so.1:
**** SECTION HEADER TABLE ****
[No]  Type  Flags  Addr      Offset     Size      Name
.....
[7]   PBIT  -AI    0x538     0x538     0x1c     .init
[8]   PBIT  -AI    0x554     0x554     0xac     .text
```

(続く)

```

[9]      PBIT  -AI    0x600    0x600    0xc     .fini
.....
[13]     PBIT  WA-    0x10688  0x688    0x18    .data
.....
[16]     NOBI  WA-    0x1073c  0x73c    0x20    .bss
.....

```

前出の `nm(1)` および `dump(1)` の例による出力を使用すると、セクション `.init`、`.text`、および `.fini` に対する関数 `_init`、`foo`、および `_fini` の関連付けが見られます。これらのセクションは読み取り専用であるため、テキストセグメントの一部です。

同様に、データ配列 `data` と `bss` は、それぞれセクション `.data` と `.bss` に関連付けられています。これらのセクションは書き込み可能であるため、データセグメントの一部です。

注 - 前出の `dump(1)` の表示は例のために簡素化されています。

ここで説明したツール情報を利用すると、生成するすべての ELF ファイル内でのコードおよびデータの位置を解析できます。この知識は、次の節での説明を理解する上で役立ちます。

基本システム

アプリケーションがある共有オブジェクトを使用して構築される場合、そのオブジェクトの読み込み可能な内容全体が、実行時にそのプロセスの仮想アドレス空間に割り当てられます。共有オブジェクトを使用する各プロセスは、まずメモリー内にある共有オブジェクトの単一のコピーを参照します。

共有オブジェクト内の再配置は処理されて、シンボリック参照を該当する定義に結合します。これにより、共有オブジェクトがリンカーによって生成されたときには得られなかった真の仮想アドレスが計算されます。通常、これらの再配置によって、プロセスのデータセグメント内のエントリが更新されます。

メモリー管理スキーマは、プロセス間で共有オブジェクトの共有メモリーをページ細部のレベルで動的リンクするときの基本となります。メモリーページは、実行時に変更されていなければ共有できます。プロセスは、データ項目の書き込み時、または共有オブジェクトへの参照の再配置時に共有オブジェクトのページに書き込む

場合、そのページの専用コピーを生成します。この専用コピーは共有オブジェクトの他のユーザーに対して何も影響しませんが、このページは、他のプロセス間での共有に伴う利点をすべて失います。この方法で変更されたテキストページは、「純粋でない」(impure)と呼ばれます。

メモリーに割り当てられた共有オブジェクトのセグメントは、2つの基本的なカテゴリに分類されます。これは、読み取り専用の「テキスト」セグメントと、読み書き可能な「データ」セグメントです(ELFファイルからこの情報を取得する方法については、121ページの「役に立つツール」を参照)。共有オブジェクトを開発するときの主要目的は、テキストセグメントを最大化して、データセグメントを最小化することにあります。これにより、共有オブジェクトの初期設定と使用に必要な処理の量を削減しながら、コード共有の量を最適化できます。次の節では、この目的を達成するために役立つメカニズムを示します。

動的依存関係のレイジーローディング

オブジェクトをレイジー読み込みするように設定すると、共有オブジェクトの依存関係の読み込みは、最初に参照されるまで延期できます(82ページの「動的依存関係のレイジーローディング」を参照)。

依存関係の数が少ないアプリケーションの場合、アプリケーションを実行すると、レイジー読み込み可に設定されているかどうかに関係なく、すべての依存関係が読み込まれることがあります。ただし、レイジー読み込みでは依存関係の処理が処理の起動時から延期され、処理の実行期間全体にわたって広がるので、全体的な処理性能は向上します。

多くの依存関係を持つアプリケーションの場合、レイジー読み込みを使用すると、実行の特定スレッドに参照されないためにまったく読み込まれない依存関係もあります。

位置に依存しないコード

実行時に必要なページ変更が最も少ないプログラムを作成するために、コンパイラは、`-k pic` オプションによって、位置に依存しないコードを生成します。動的実行可能ファイル内のコードは、通常、メモリー内の固定アドレスに結合されていますが、位置に依存しないコードは、プロセスのアドレス空間内にある任意の場所に読み込みできます。このコードは、特定のアドレスに結合されていないため、それを使用する各プロセスの異なるアドレスでページ変更を行わなくても、正しく実行されます。

位置に依存しないコードを使用すると、再配置可能な参照が、共有オブジェクトのデータセグメント内のデータを使用する間接参照の形式で生成されます。この結果、テキストセグメントコードは読み取り専用のままになり、すべての再配置更新がデータセグメント内の対応するエントリに適用されます。これらの2つのセクションの使用方法については、316ページの「大域オフセットテーブル (プロセッサ固有)」と317ページの「プロシージャのリンクテーブル (プロセッサ固有)」を参照してください。

共有オブジェクトが位置に依存しないコードから構築される場合、テキストセグメントでは通常、実行時に大量の再配置を実行する必要があります。この再配置を処理するために実行時リンカーが用意されていますが、この処理によるシステムオーバーヘッドによって深刻な性能低下が生じるおそれがあります。

テキストセグメントに対して再配置を必要とする共有オブジェクトは、`dump(1)` を使用して、`TEXTREL` エントリの出力を調べることによって識別できます。次に例を示します。

```
$ cc -o libfoo.so.1 -G -R. foo.c
$ dump -Lv libfoo.so.1 | grep TEXTREL
[9]      TEXTREL  0
```

注 - `TEXTREL` エントリの値は不適切です。共有オブジェクトにこの値が存在する場合は、テキスト再配置があることを示しています。

テキスト再配置を含む共有オブジェクトの作成を防止するには、リンカーの `-z text` フラグを使用することをお勧めします。このフラグを使用すると、リンカーは、入力として使用された、位置に依存しないコード以外のコードのソースを示す診断を生成し、意図した共有オブジェクトの生成は失敗に終わります。次に例を示します。

```
$ cc -o libfoo.so.1 -z text -G -R. foo.c
Text relocation remains          referenced
  against symbol                offset      in file
foo                             0x0        foo.o
bar                             0x8        foo.o
ld: fatal: relocations remain against allocatable but \
non-writable sections
```

ここでは、ファイル `foo.o` から位置に依存しないコード以外のコードが生成されたために、テキストセグメントに対して2つの再配置が生成されています。これらの

診断は、可能な場合、再配置の実行に必要なシンボリック参照すべてを示します。この場合、再配置はシンボル `foo` と `bar` に対するものです。

`-Kpic` オプションを使用しないという原因以外で、共有オブジェクトの生成時にテキスト再配置が作成される最も一般的な原因は、位置に依存しない適切なプロトタイプによって符号化されていない手書きアセンブラコードを含めているというものです。

注 - 中間アセンブラファイルを生成するコンパイラ機能を使用すると、いくつかの単純なテストケースソースファイルを試すことによって、位置からの独立性を有効にするために使用されるコーディング手法がわかります。

プロセッサによっては、位置に依存しないフラグの2つ目の形式として `-K PIC` も使用できます。このフラグを使用すると、追加コードによるオーバーヘッドと引き換えに、より多数の再配置を処理できます (詳細については、`cc(1)` を参照)。

共有可能性の最大化

123ページの「基本システム」で説明したように、共有オブジェクトのテキストセグメントだけがそれを使用するすべてのプロセスによって共有され、データセグメントは通常共有されません。共有オブジェクトを使用する各プロセスは、通常、そのデータセグメント全体の専用メモリーコピーをそのセグメント内に書き込まれるデータ項目として生成します。データセグメントを削減するにはテキストセグメントに書き込まれることがないデータ要素を移動するか、またはデータ項目を完全に削除します。

次の節では、データセグメントのサイズを削減するために使用できるいくつかのメカニズムについて説明します。

テキストへの読み取り専用データの移動

読み取り専用のデータ要素はすべて、テキストセグメントに移動する必要があります。これは、`const` 宣言を使用して達成できます。たとえば、次の文字列は、書き込み可能なデータセグメントの一部である `.data` セクションにあります。

```
char * rdstr = 'this is a read-only string';
```

これに対して、次の文字列は、テキストセグメント内にある読み取り専用データセクションである `.rodata` セクション内にあります。

```
const char * rdstr = 'this is a read-only string';
```

読み取り専用要素をテキストセグメントに移動することによるデータセグメントの削減は目的に沿うものですが、再配置を必要とするデータ要素を移動すると、逆効果になるおそれがあります。たとえば、次の文字列配列があるとします。

```
char * rdstrs[] = { "this is a read-only string",  
                  "this is another read-only string" };
```

最初は、次の定義の方が良く思われるかもしれません。

```
const char * const rdstrs[] = { ..... };
```

これにより、文字列とこれらの文字列へのポインタ配列は、確実に `.rodata` セクションに置かれます。ただし、ユーザーがアドレス配列を読み取り専用と認識しても、実行時にはこれらのアドレスを再配置しなければなりません。したがって、この定義では再配置が作成されます。この定義は次のように表わします。

```
const char * rdstrs[] = { ..... };
```

これにより、配列文字列は読み取り専用のテキストセグメント内に保持されますが、配列ポインタは、再配置できる書き込み可能なデータセグメント内に保持されます。

注 - コンパイラによっては、位置に依存しないコードを生成するときに、実行時再配置を起こして、このような項目を書き込み可能セグメントに置くことになる読み取り専用割り当てを検出できるものがあります (たとえば、`.picdata`)。

多重定義されたデータの短縮

多重定義されたデータを短縮すると、データを削減できます。同じエラーメッセージが複数回発生するプログラムの場合は、1つの大域なデータを定義し、他のインスタンスすべてにこれを参照させると効率が良くなります。次に例を示します。

```
const char * Errmsg = "prog: error encountered: %d";  
  
foo()  
{  
    .....  
    (void) fprintf(stderr, Errmsg, error);  
    .....  
}
```

この種のデータ削減に適した対象は文字列です。共有オブジェクトでの文字列の使用は、`strings(1)` を使用して調べることができます。次に例を示します。

```
$ strings -10 libfoo.so.1 | sort | uniq -c | sort -rn
```

上記のコードは、ファイル `libfoo.so.1` 内に、データ文字列のソートされたリストを生成します。このリスト内の各項目には、文字列の出現回数を示す接頭辞が付いています。

自動変数の使用

データ項目用の常時記憶領域は、関連する機能が自動 (スタック) 変数を使用するように設計できる場合、完全に削除することができます。常時記憶領域を少しでも削除すると、通常これに対応して、必要な実行時再配置の数も減ります。

バッファの動的割り当て

大きなデータバッファは、通常、常時記憶領域を使用して定義するのではなく、動的に割り当てる必要があります。これにより、アプリケーションの現在の呼び出しに必要なバッファだけが割り当てられるため、メモリー全体を節約できます。動的割り当てを行うと、互換性に影響を与えることなくバッファのサイズを変更できるため、柔軟性も増します。

ページング回数の削減

126ページの「共有可能性の最大化」で説明したメカニズムの多くは、共有オブジェクトを使用するときに生じるページングを削減するために役立ちます。ここでは、一般的なソフトウェア性能に関する考慮事項のいくつかについてさらに説明します。

新しいページにアクセスするすべてのプロセスでページフォルトが発生します。これはコストのかかる操作であり、また共有オブジェクトは多数のプロセスで使用できるため、共有オブジェクトへのアクセスによって生成されるページフォルトの数を減らすと、プロセスおよびシステム全体の効率が改善されます。

使用頻度の高いルーチンとそのデータを隣接するページの集合として編成すると、参照の効率が良くなるため、性能は通常向上します。あるプロセスがこれらの関数の1つを呼び出すとき、この関数がすでにメモリー内にある場合があります。これは、この関数が、使用頻度の高い他の関数のすぐ近くに存在するためです。同様に、相互に関連する関数をグループ化すると、参照効率が向上します。たとえば、

関数 `foo()` への呼び出しによって、常に関数 `bar()` が呼び出される場合は、これらの関数を同じページ上に置きます。`cflow(1)`、`tcov(1)`、`prof(1)`、および `gprof(1)` は、コードカバレッジとプロファイリングを判定するために役立ちます。

関連する機能を各自の共有オブジェクトに分離することもお勧めします。標準 C ライブラリは従来、関連しない多数の関数を含んで構築されていて、たとえば単一の実行可能ファイルがこのライブラリ内のすべてを使用することはほとんどありません。このライブラリは広範囲に使用されるため、実際に使用頻度の最も低い関数がどれかを判定することもかなり困難です。これに対して、共有オブジェクトを最初から設計する場合は、関連する関数だけを共有オブジェクト内に保持することをお勧めします。これにより、参照効率が向上するだけでなく、オブジェクト全体のサイズを減らすという効果も得られます。

再配置

75ページの「再配置処理」では、実行時リンカーが動的実行可能ファイルと共有オブジェクトを再配置して、実行可能プロセスを作成するためのメカニズムについて説明しました。76ページの「シンボルの検索」と78ページの「再配置が実行される時」は、この再配置処理を2つの領域に分類して、関連のメカニズムを簡素化して説明しています。これらの2つのカテゴリは、再配置による性能への影響を考慮するためにも最適です。

シンボル検索

実行時リンカーは、シンボルを検索する必要がある場合、デフォルトでは各オブジェクトを検索して検索を行います。実行時リンカーは、まず動的実行可能ファイルから始めて、オブジェクトが読み込まれるのと同じ順序で各共有オブジェクトへと進みます。ほとんどの場合、シンボル再配置を必要とする共有オブジェクトは、シンボル定義の提供者になります。

この場合、この再配置に使用されるシンボルが共有オブジェクトのインタフェースの一部として必要ではない場合、このシンボルは静的変数または自動変数に変換される可能性が高くなります。シンボル削減は、共有オブジェクトのインタフェースから削除されたシンボルにも適用できます(詳細については、56ページの「シンボル範囲の縮小」を参照)。これらの変換を行うことによって、リンカーは、共有オブジェクトの作成中にこれらのシンボルに対するシンボル再配置を処理しなくてはなりません。

共有オブジェクトから表示できなければならない唯一の大域データ項目は、そのユーザーインターフェースに関するものです。しかし、大域データは異なる複数のソースファイルにある複数の関数から参照できるように定義されていることが多いため、これは歴史的に達成が困難です。シンボルの縮小 (56ページの「シンボル範囲の縮小」を参照) を適用することによって、不要な大域シンボルを削除できます。共有オブジェクトからエクスポートされた大域シンボルの数を少しでも減らせば、再配置のコストを削減し、性能全体を向上させることができます。

直接結合では、多数のシンボル再配置や依存関係を伴う動的プロセスでのシンボル検索のオーバーヘッドを大幅に削減できます (78ページの「直接結合」を参照)。

再配置を実行する場合

すべてのデータ参照再配置は、アプリケーションが制御を取得する前に、プロセス初期設定段階で実行する必要があります。これに対して、関数参照は、関数の最初のインスタンスが呼び出されるまで延期できます。データ再配置の数を減らすことによって、プロセスの実行時初期設定も削減されます。

初期設定再配置コストは、たとえば機能インターフェースによってデータ項目を返すなど、データ再配置を関数再配置に変換して延期することもできます。この変換によって、初期設定再配置コストがプロセスの実行期間中に効率的に分配されると、性能は明らかに向上します。いくつかの機能インターフェースはプロセスの特定の呼び出しでは決して呼び出されない可能性もあるため、それらの再配置オーバーヘッドもすべてなくなります。

機能インターフェースを使用した場合の利点については、131ページの「コピー再配置」で説明します。この節では、動的実行可能ファイルと共有オブジェクトの間で使用される特殊でコストのかかる再配置メカニズムについて述べ、この再配置によるオーバーヘッドを回避する方法の例を示します。

再配置セクションの結合

再配置は、デフォルトでは、適用対象のセクションによってグループ化されます。ただし、オブジェクトを `-z combrelloc` オプションによって構築すると、プロシージャリンクテーブル再配置を除くすべてが、単一の共通セクション `.SUNW_reloc` に置かれます。

この方法で再配置レコードを結合すると、すべての `RELATIVE` 再配置が1つにグループ化され、すべてのシンボルの再配置はシンボル名によって並べ替えられます。 `RELATIVE` 再配置をグループ化すると `DT_RELACOUNT/DT_RELCOUNT`

.dynamic エントリを使用した最適な実行時処理が行われ、シンボルのエントリを並べ替えると実行時にシンボルを検索する時間を削減できます。

コピー再配置

共有オブジェクトは、通常、位置に依存しないコードによって構築されます。このタイプのコードから外部データ項目への参照は、1組のテーブルによる間接アドレス指定を使用します (詳細については、124ページの「位置に依存しないコード」を参照)。これらのテーブルは、データ項目の実アドレスによって実行時に更新されます。この実アドレスによって、コード自体を変更することなくデータにアクセスすることができます。

ただし、動的実行可能ファイルは通常、位置に依存しないコードからは作成されません。したがって、これらのファイルが作成する外部データへの参照は、その参照を行うコードを変更することによって実行時にしか実行できないように見えます。読み取り専用のテキストセグメントの変更は回避する必要があるため、コピー再配置と呼ばれる再配置手法が、この参照を解決するために使用されます。

動的実行可能ファイルを作成するためにリンカーが使用され、データ項目への参照が依存共有オブジェクトのどれかに常駐する場合は、動的実行可能ファイルの .bss で、共有オブジェクト内のデータ項目のサイズに等しいスペースが割り当てられます。このスペースには、共有オブジェクトに定義されているのと同じシンボリック名も割り当てられます。リンカーは、このデータ割り当てとともに特殊なコピー再配置レコードを生成して、実行時リンカーに対し、共有オブジェクトから動的実行可能ファイル内のこの割り当てスペースへデータをコピーするように指示します。

このスペースに割り当てられたシンボルは大域であるため、すべての共有オブジェクトからの参照をすべてを満たすために使用されます。この結果、動的実行可能ファイルはデータ項目を継承し、この項目を参照するプロセス内の他のオブジェクトすべてがこのコピーに結合されます。コピーの元となるデータは未使用になります。

このメカニズムを明確に示す例があります。この例では標準 C ライブラリ内で保持されるシステムエラーメッセージの配列を使用します。SunOS オペレーティングシステムの以前のリリースでは、この情報へのインタフェースが、2つの大域変数 `sys_errlist[]` および `sys_nerr` によって提供されました。最初の変数はエラーメッセージ文字列を提供し、2つ目の変数は配列自体のサイズを示しました。これらの変数はアプリケーション内で、通常次のように使用されていました。

```
$ cat foo.c
extern int      sys_nerr;
extern char *   sys_errlist[];
```

```

char *
error(int errnumb)
{
    if ((errnumb < 0) || (errnumb >= sys_nerr))
        return (0);
    return (sys_errlist[errnumb]);
}

```

ここで、アプリケーションは、関数 `error` を使用して、番号 `errnumb` に対応するシステムエラーメッセージを取得するために提供しています。

このコードを使用して作成された動的実行可能ファイルを調べると、コピー再配置の実装が更に詳細に示されます。

```

$ cc -o prog main.c foo.c
$ nm -x prog | grep sys_
[36] |0x00020910|0x00000260|OBJT |WEAK |0x0 |16 |sys_errlist
[37] |0x0002090c|0x00000004|OBJT |WEAK |0x0 |16 |sys_nerr
$ dump -hv prog | grep bss
[16] NOBI WA- 0x20908 0x908 0x268 .bss
$ dump -rv prog

**** RELOCATION INFORMATION ****

.rela.bss:
Offset      Symndx      Type      Addend
0x2090c     sys_nerr    R_SPARC_COPY 0
0x20910     sys_errlist R_SPARC_COPY 0
.....

```

ここで、リンカーは、動的実行可能ファイルの `.bss` にスペースを割り当てて、`sys_errlist` および `sys_nerr` によって表わされるデータを受け取っています。これらのデータは、プロセス初期設定時に、実行時リンカーによって C ライブラリからコピーされます。このため、これらのデータを使用する各アプリケーションは、データの専用コピーを各自のデータセグメントで取得します。

この手法には、実際には 2 つの欠点があります。まず、各アプリケーションでは、実行時のデータコピーによるオーバーヘッドによって性能が低下します。もう 1 つは、データ配列 `sys_errlist` のサイズが、C ライブラリのインタフェースの一部になるという点です。この配列のサイズが変わると、新しいエラーメッセージが追加されて、この配列を参照する動的実行可能ファイルすべてで新しいエラーメッ

ページにアクセスするための新しいリンク編集が行われます。この新しいリンク編集が行われないと、動的実行可能ファイル内の割り当てスペースが不足して、新しいデータを保持できません。

このような欠点は、動的実行可能ファイルに必要なデータが機能インタフェースによって提供されればなくなります。ANSI C 関数 `strerror(3C)` は、この点を示すものです。この関数は、提示されたエラー番号に基づいて該当するエラー文字列へのポインタを返すように実装されます。この関数の実装状態は次のようになります。

```
$ cat strerror.c
static const char * sys_errlist[] = {
    "Error 0",
    "Not owner",
    "No such file or directory",
    .....
};
static const int sys_nerr =
    sizeof (sys_errlist) / sizeof (char *);

char *
strerror(int errnum)
{
    if ((errnum < 0) || (errnum >= sys_nerr))
        return (0);
    return ((char *)sys_errlist[errnum]);
}
```

`foo.c` のエラールーチンは、ここではこの機能インタフェースを使用するように単純化できます。これによって、プロセス初期設定時に元のコピー再配置を実行する必要がなくなります。

また、データは共有オブジェクト限定のものであるため、そのインタフェースの一部ではなくなります。したがって、共有オブジェクトは、データを使用する動的実行可能ファイルに悪影響を与えることなく、自由にデータを変更できます。共有オブジェクトのインタフェースからデータ項目を削除すると、一般に共有オブジェクトのインタフェースとコードが維持しやすくなるとともに、性能も向上します。

コピー再配置は回避する必要がありますが、`ldd(1)` に `-d` オプションまたは `-r` オプションのどちらかをつけて使用すると、動的実行可能ファイル内にそれがいないかを検査できます。

たとえば、動的実行可能ファイル `prog` が当初、次の 2 つのコピー再配置が記録されるように、共有オブジェクト `libfoo.so.1` に対して構築されている場合を考えます。

```

$ nm -x prog | grep _size_
[36] | 0x000207d8|0x40|OBJT |GLOB |15 | _size_gets_smaller
[39] | 0x00020818|0x40|OBJT |GLOB |15 | _size_gets_larger
$ dump -rv size | grep _size_
0x207d8      _size_gets_smaller      R_SPARC_COPY      0
0x20818      _size_gets_larger       R_SPARC_COPY      0

```

さらに、これらのシンボルについて異なるサイズを含む、この共有オブジェクトの新しいバージョンが提供されているとします。

```

$ nm -x libfoo.so.1 | grep _size_
[26] | 0x00010378|0x10|OBJT |GLOB |8 | _size_gets_smaller
[28] | 0x00010388|0x80|OBJT |GLOB |8 | _size_gets_larger

```

この場合、動的実行可能ファイルに対して `ldd(1)` を実行すると、次のように表示されます。

```

$ ldd -d prog
libfoo.so.1 => ./libfoo.so.1
.....
copy relocation sizes differ: _size_gets_smaller
(file prog size=40; file ./libfoo.so.1 size=10);
./libfoo.so.1 size used; possible insufficient data copied
copy relocation sizes differ: _size_gets_larger
(file prog size=40; file ./libfoo.so.1 size=80);
./prog size used; possible data truncation

```

ここで、`ldd(1)` は、動的実行可能ファイルが、共有オブジェクトが提供しなければならぬデータすべてをコピーするけれども、その割り当てスペースで許容できる量しか受け付けられないということを知らせています。

注 - 位置に依存しないコードだけでアプリケーションを作成すれば、コピー再配置を完全に排除することができます (124ページの「位置に依存しないコード」を参照)。

-Bsymbolic の使用

リンカーの `-Bsymbolic` オプションを使用すると、シンボルの参照を共有オブジェクト内の大域定義に結合できます。このオプションは本来、実行時リンカーそのものを作成するために設計されたもので、長い歴史があるといえます。

オブジェクトのインタフェースを定義し、非公開シンボルをローカルに縮小することは、`-Bsymbolic` オプションを使用した実行時再配置のコストを削減する望ましい仕組みです (56ページの「シンボル範囲の縮小」を参照)。実際に、`-Bsymbolic` を使用すると直感的にはわからない副産物のできる可能性があります。

シンボリックに結合されたシンボルが割り込み (*interposition*) された場合、シンボリックに結合されたオブジェクトの外からのそのシンボルへの参照は、その割り込みに結合します。オブジェクトそのものはすでに内部的に結合されています。本質的に、同じ名前を持つ2つのシンボルは、プロセス内から参照されます。シンボリックに結合されたデータシンボルは、コピーを再配置し (131ページの「コピー再配置」を参照)、同じ割り込み状態を作成します。

注 - シンボリックに結合された共有オブジェクトは、`.dynamic` エントリ `DT_SYMBOLIC` で表されます。このタグは情報を提供するだけで、実行時リンカーは、これらのオブジェクトからのシンボルの検索を他のオブジェクトからの場合と同じ方法で処理します。シンボリック結合はリンカーフェーズで作成されたものと想定されます。

共有オブジェクトのプロファイリング

実行時リンカーは、アプリケーションの実行中に処理された共有オブジェクトすべてのプロファイリング情報を生成できます。これが可能となるのは、実行時リンカーが、共有オブジェクトをアプリケーションに結合しなければならないためであり、したがって、実行時リンカーはすべての大域関数割り当てを傍受できます (これらの割り当ては、`.plt` エントリによって起こります。このメカニズムの詳細は、78ページの「再配置が実行される時」を参照)。

共有オブジェクトのプロファイルは、環境変数 `LD_PROFILE` でその名前を指定することによって有効になります。この環境変数を使用すると、一度に1つの共有オブジェクトを解析できます。ただし、環境変数の設定は、1つまたは複数のアプリケーションによる共有オブジェクトの使用を解析するために使用できます。次の例では、コマンド `ls(1)` の1回の呼び出しによる `libc` の使用が解析されています。

```
$ LD_PROFILE=libc.so.1 ls -l
```

次の例では、環境変数の設定によって、アプリケーションが `libc` を使用するたびに、環境変数の設定期間に関する解析情報が蓄積されます。

```
$ LD_PROFILE=libc.so.1; export LD_PROFILE
$ ls -l
$ make
$ ...
```

プロファイリングが有効な場合、プロファイルデータファイルがなければ、ファイルが作成されて、実行時リンカーに割り当てられます。上記の例で、このデータファイルは `/var/tmp/libc.so.1.profile` です。64 ビットライブラリは、拡張プロファイル形式を必要とし、`.profilex` 接尾辞を使用して書かれます。代替ディレクトリを指定して、環境変数 `LD_PROFILE_OUTPUT` によってプロファイルデータを格納することもできます。

このプロファイルデータファイルは、`profil(2)` データを保存して、指定の共有オブジェクトの使用に関連するカウント情報呼び出すために使用されます。このプロファイルデータは、`gprof(1)` によって直接調べることができます。

注 - `gprof(1)` は通常、`cc(1)` の `-xpg` オプションにを使用してコンパイルされた実行可能ファイルにより作成された `gmon.out` プロファイルデータを解析するために使用されます。実行時リンカーのプロファイル解析では、このオプションによってコードをコンパイルする必要はありません。依存共有オブジェクトがプロファイルされるアプリケーションは、`profil(2)` に対して呼び出しを行うことができます。これは、このシステム呼び出しでは、同じプロセス内で複数の呼び出しが行われないためです。同じ理由から、`cc(1)` の `-xpg` オプションによって、これらのアプリケーションをコンパイルすることもできません。このコンパイラによって生成されたプロファイリングのメカニズムが `profil(2)` の上にも構築されるためです。

このプロファイリングメカニズムの最も強力な機能の 1 つに、複数のアプリケーションに使用される共有オブジェクトの解析があります。通常、プロファイリング解析は、1 つまたは 2 つのアプリケーションを使用して実行されます。しかし共有オブジェクトは、その性質上、多数のアプリケーションで使用できます。これらのアプリケーションによる共有オブジェクトの使用方法を解析すると、共有オブジェクトの全体の性能を向上させるには、どこに注意すべきかを理解できます。

次の例は、ソース階層内でいくつかのアプリケーションを作成したときの `libc` の性能解析を示しています。


```

$ LD_PROFILE=libc.so.1 ; export LD_PROFILE
$ make
$ gprof -b /usr/lib/libc.so.1 /var/tmp/libc.so.1.profile
.....

granularity: each sample hit covers 4 byte(s) ....

index  %time      self descents    called/total    parents
        called+self  name          index
        called/total  children
.....
-----
          0.33      0.00      52/29381      _gettxt [96]
          1.12      0.00      174/29381     _tzload [54]
          10.50     0.00      1634/29381    <external>
          16.14     0.00      2512/29381    _opendir [15]
          160.65    0.00      25009/29381   _endopen [3]
[2]     35.0  188.74      0.00      29381         _open [2]
-----
.....
granularity: each sample hit covers 4 byte(s) ....

% cumulative  self
time  seconds  seconds  calls  ms/call  ms/call  name
35.0   188.74   188.74   29381   6.42     6.42     _open [2]
13.0   258.80    70.06   12094   5.79     5.79     _write [4]
9.9    312.32    53.52   34303   1.56     1.56     _read [6]
7.1    350.53    38.21   1177    32.46    32.46    _fork [9]
.....

```

特殊名 <external> は、プロファイル中の共有オブジェクトのアドレス範囲外からの参照を示しています。したがって、上記の例では、1634 は、動的実行可能ファイル、または プロファイル解析の進行中に libc によって結合された他の共有オブジェクトから発生した libc 内の関数 open(2) を呼び出しています。

注 - 共有オブジェクトのプロファイルは、マルチスレッド化に対し安全です。ただし、あるスレッドがプロファイルデータ情報を更新しているときに、もう 1 つのスレッドが fork(2) を呼び出す場合は例外です。fork1(2) を使用すると、この制限はなくなります。

バージョンアップ

概要

リンカーによって処理されるオブジェクトには、他のオブジェクトを結合できる多数の大域シンボルがあります。これらのシンボルは、オブジェクトのアプリケーションバイナリインタフェース (ABI) を記述するものです。オブジェクトの展開中、このインタフェースは、大域シンボルの追加または削除が原因で変更されることがあります。また、オブジェクト展開には、内部実装の変更が関与することがあります。

バージョンアップとは、インタフェースや実装状態の変更を示すためにオブジェクトに適用できるいくつかの手法のことをいいます。これらの手法を使用すると、下位互換性を保ちながらオブジェクト制御による展開を行うことができます。

この章では、オブジェクトの ABI の定義方法について説明し、このインタフェースに対する変更によって下位互換性が受ける影響を分類します。また、インタフェースと実装状態の変更を新しいリリースのオブジェクトに組み込むためのモデルを示します。

この章では、動的実行可能ファイルと共有オブジェクトの実行時インタフェースを中心に説明します。これらの動的オブジェクト内での変更を記述して管理するために使用される手法は、一般的な用語で説明してあります。共有オブジェクトに適用される命名規約とバージョンアップシナリオの共通セットは、付録 B に示してあります。

動的オブジェクトの開発者は、インタフェース変更の結果に注意し、特に以前のオブジェクトとの下位互換性を維持するという点で、これらの変更の管理方法を理解する必要があります。

動的オブジェクトによって使用可能になった大域シンボルは、オブジェクトの公共インタフェースを表わします。リンク編集の最後でオブジェクトに残る大域シンボルの数は、公共にしたいと望む数を超える場合がよくあります。これらの大域シンボルは、そのオブジェクトの構築に使用された再配置可能オブジェクトの間で必要な相互関係から引き出されるものであり、オブジェクト自体の専用インタフェースを表わします。

オブジェクトのバイナリインタフェースを定義するには、まず、作成中のオブジェクトから公共に使用できるようにする大域シンボルだけを定義します。これらの公共シンボルは、リンカーの `-M` オプションと関連の `mapfile` を最終リンク編集の一部として使用することによって確立できます。この手法は、56ページの「シンボル範囲の縮小」に説明されています。この公共インタフェースは、1つまたは複数のバージョン定義を作成中のオブジェクト内に確立し、オブジェクトの展開時に新しいインタフェースを追加するための基礎を作ります。

次の節は、この初期公共インタフェースに基づいて説明されています。最初に、インタフェースへの各種の変更をどのように分類すると、適切な管理を行えるかを理解しておくと便利です。

インタフェースの互換性

オブジェクトにはさまざまな変更を加えることができます。これらの変更は、単純に次の2つのグループに分類することができます。

- 互換性のある変更。これらの変更は、今まで使用できたインタフェースがすべてそのままの状態に残されるという点で付加的なもの
- 互換性のない変更。これらの変更は既存インタフェースを変更したため、そのインタフェースの既存ユーザーはそれを使用できないか、または動作が異なる

次のリストは、共通のオブジェクト変更のいくつかを上記のカテゴリのどちらかに分類しています。

- シンボルの追加 - 互換性のある変更
- シンボルの削除 - 互換性のない変更
- 非 `varargs` (3HEAD) 関数への引数の追加 - 互換性のない変更

- 関数からの引数の削除 - 互換性のない変更
- 関数への、または外部定義としてのデータ項目のサイズまたは内容の変更 - 互換性のない変更
- バグ修正または関数の内部拡張 - オブジェクトの意味プロパティを変更しない場合は互換性のある変更。変更する場合は互換性のない変更

注 - シンボルを追加すると、割り込みが原因で新しいシンボルがアプリケーションによるそのシンボルの使用法と矛盾するという互換性のない変更が生じる可能性があります。ただし、通常ソースレベル名前空間の管理が使用されるため、実際にはこの可能性はめったにありません。

互換性のある更新は、バージョン定義を生成されたオブジェクトの「内部」に維持することにより調整できます。互換性のない変更は、新しい「外部」バージョンアップ名によって新しいオブジェクトを作成することにより調整できます。これらのバージョンアップ手法を使用すると、実行時の正しいバージョン割り当てを検査できるだけでなく、アプリケーションの選択的割り当てを行うことができます。これらの2つの手法については、次の節でさらに詳しく説明します。

内部バージョンアップ

動的オブジェクトには、1つまたは複数の内部バージョン定義を関連付けることができます。各バージョン定義は通常、1つまたは複数の名前に関連付けられます。シンボル名は、1つのバージョン定義にしか関連付けられませんが、バージョン定義は他のバージョン定義からシンボルを継承できます。したがって、1つまたは複数の独立した、または関連するバージョン定義を作成中のオブジェクト内に定義するための構造が存在します。オブジェクトに新しい変更が加えられたら、新しいバージョン定義を追加してこれらの変更を表現することができます。

共有オブジェクト内にバージョン定義を与えた場合、次の2つの結果が得られます。

- この共有オブジェクトに対して構築された動的オブジェクトは、それらが結合されているバージョン定義への依存関係を記録できる。これらのバージョンの依存関係は、アプリケーションの正しい実行に適切なインタフェースまたは機能を使用できるかどうかを確認するため、実行時に検査される
- 動的オブジェクトは、そのリンク編集集中に結合先として必要な共有オブジェクトのバージョン定義だけを選択できる。このメカニズムを使用すると、開発者は、

最も自由な操作を可能にするインタフェースまたは機能に合わせて、共有オブジェクトへの依存関係を調整することができる

バージョン定義の作成

バージョン定義は、一般にシンボル名と一意のバージョン名との関連付けからなります。これらの関連付けは、mapfile 内に確立され、リンカーの `-M` オプションを使用して、オブジェクトの最終リンク編集に与えられます (この手法については、56 ページの「シンボル範囲の縮小」を参照)。

バージョン定義は、バージョン名が mapfile 命令の一部として指定されている場合は必ず確立されます。次の例では、2 つのソースファイルが mapfile 命令とともに結合されて、定義済み公共インタフェースを持つオブジェクトを作成しています。

```
$ cat foo.c
extern const char * _foo1;

void fool()
{
    (void) printf(_foo1);
}

$ cat data.c
const char * _foo1 = "string used by fool()\n";

$ cat mapfile
SUNW_1.1 {
    global:
        fool;
    local:
        *;
};

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ nm -x libfoo.so.1 | grep "foo.$"
[33]  |0x0001058c|0x00000004|OBJT |LOCL |0x0 |17 |_foo1
[35]  |0x00000454|0x00000034|FUNC |GLOB |0x0 |9 |foo1
```

ここで、シンボル `foo1` は共有オブジェクトの公共インタフェースを提供するために定義された唯一の大域シンボルです。特殊な自動縮小命令「*」は、他の大域シンボルすべてを縮小することによって、生成中のオブジェクト内にローカル結合が生じるようにします (この命令については、51 ページの「追加シンボルの定義」を参照)。関連バージョン名 `SUNW_1.1` は、バージョン定義を生成させます。したがって、共有オブジェクトの公共インタフェースは、大域シンボル `foo1` に関連付けられた内部バージョン定義 `SUNW_1.1` からなります。

バージョン定義、または自動縮小命令によってオブジェクトが生成されると、基本バージョンも必ず作成されます。この基本バージョンは、ファイル自体の名前を使用して定義され、リンカーによって生成された予約シンボルすべてを関連付けるために使用されます (予約シンボルのリストについては 61 ページの「出力イメージの生成」を参照)。

オブジェクト内に含まれるバージョン定義は、`pvs(1)` に `-d` オプションを付けて使用して表示できます。

```
$ pvs -d libfoo.so.1
    libfoo.so.1;
    SUNW_1.1;
```

ここで、オブジェクト `libfoo.so.1` には、基本バージョン定義 `libfoo.so.1` とともに、`SUNW_1.1` という名前の内部バージョン定義があります。

注 - リンカーの `-z noversion` オプションを使用すると、`mapfile` 指示のシンボル縮小を実行できますが、バージョン定義の作成は抑制されます。

この初期バージョン定義から始めて、新しいインタフェースと更新された機能を追加することによって、オブジェクトを展開させることができます。たとえば、新機能 `foo2` は、それがサポートするデータ構造とともに、ソースファイル `foo.c` および `data.c` を更新することによってオブジェクトに追加することができます。

```
$ cat foo.c
extern const char * _foo1;
extern const char * _foo2;

void foo1()
{
    (void) printf(_foo1);
}

void foo2()
{
    (void) printf(_foo2);
}

$ cat data.c
const char * _foo1 = "string used by foo1()\n";
const char * _foo2 = "string used by foo2()\n";
```

新しいバージョン定義 SUNW_1.2 を作成すると、シンボル foo2 を表わす新しいインタフェースを定義できます。また、この新しいインタフェースは、元のバージョン定義 SUNW_1.1 を継承するように定義できます。

この新しいインタフェースを作成すると、オブジェクトの展開が識別され、ユーザーは結合先のインタフェースを検査して選択できるため重要です。これらの概念については、148ページの「バージョン定義への結合」と152ページの「バージョン結合の指定」で詳しく説明します。

次の例は、これらの1つのインタフェースを作成する `mapfile` 命令を示しています。

```
$ cat mapfile
SUNW_1.1 {                                # Release X
    global:
        foo1;
    local:
        *;
};

SUNW_1.2 {                                # Release X+1
    global:
        foo2;
} SUNW_1.1;

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ nm -x libfoo.so.1 | grep "foo.$"
[33]  |0x00010644|0x00000004|OBJT|LOCL|0x0|17|_foo1
[34]  |0x00010648|0x00000004|OBJT|LOCL|0x0|17|_foo2
[36]  |0x000004bc|0x00000034|FUNC|GLOB|0x0|9|foo1
[37]  |0x000004f0|0x00000034|FUNC|GLOB|0x0|9|foo2
```

ここで、foo1 と foo2 は、いずれも共有オブジェクトの公共インタフェースの一部として定義されています。ただし、これらの各シンボルは、別々のバージョン定義に割り当てられています。foo1 は SUNW_1.1 に、foo2 は SUNW_1.2 に割り当てられています。

これらのバージョン定義、その継承、およびそのシンボル関連付けは、`pvs(1)` に `-d`、`-v`、および `-s` オプションをつけて表示できます。

```
$ pvs -dsv libfoo.so.1
libfoo.so.1:
    _end;
    _GLOBAL_OFFSET_TABLE_;
    _DYNAMIC;
    _edata;
    _PROCEDURE_LINKAGE_TABLE_;
```

(続く)


```

        _etext;
SUNW_1.1:
    foo1;
    SUNW_1.1;
SUNW_1.2:                {SUNW_1.1}:
    foo2;
    SUNW_1.2

```

ここで、バージョン定義 SUNW_1.2 は、バージョン定義 SUNW_1.1 に対する依存関係を持っています。

あるバージョン定義から別のバージョン定義への継承は、バージョン依存関係に結合するオブジェクトによって最終的に記録されるバージョン情報を減らすために便利な手法です。バージョン継承については、148ページの「バージョン定義への結合」で詳しく説明します。

どの内部バージョン定義にも、対応するバージョン定義シンボルが作成されています。pvs (1) の例で示したように、これらのシンボルは `-v` オプションを使用して表示されます。

ウィークバージョン定義の作成

オブジェクトに対する新しいインタフェース定義の照会を必要としない内部変更は、ウィークバージョン定義を作成することによって定義できます。このような変更の例としては、バグ修正や性能の改善があります。

こういったバージョン定義は、大域インタフェースシンボルが関連付けられていないという点で空です。

たとえば、以前の例で使用したデータファイル `data.c` が、次のようにより詳しい文字列定義を提供するように更新されたとします。

```

$ cat data.c
const char * _foo1 = "string used by function foo1()\n";
const char * _foo2 = "string used by function foo2()\n";

```

この場合、ウィークバージョン定義を照会すると、この変更を次のように識別できます。

```

$ cat mapfile
SUNW_1.1 {                # Release X
    global:
        foo1;
    local:
        *;
};

SUNW_1.2 {                # Release X+1
    global:
        foo2;
} SUNW_1.1;

SUNW_1.2.1 { } SUNW_1.2; # Release X+2

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ pvs -dv libfoo.so.1
    libfoo.so.1;
    SUNW_1.1;
    SUNW_1.2:          {SUNW_1.1};
    SUNW_1.2.1 [WEAK]: {SUNW_1.2};

```

ここで、空のバージョン定義は、ウィークラベルによって示されます。これらのウィークバージョン定義を使用すると、アプリケーションは、その機能に関連するバージョン定義に結合することによって、特定の実装状態の存在を検査できます。148ページの「バージョン定義への結合」では、これらの定義を使用する方法について詳しく説明します。

関連のないインタフェースの定義

以前の例は、オブジェクトに追加された新しいバージョン定義は、既存のバージョン定義をどのように継承するかを示しています。一意の依存しないバージョン定義を作成することもできます。次の例では、2つの新しいファイル `bar1.c` と `bar2.c` がオブジェクト `libfoo.so.1` に追加されています。これらのファイルは、2つの新しいシンボル `bar1` と `bar2` をそれぞれ提供します。

```

$ cat bar1.c
extern void foo1();

void bar1()
{
    foo1();
}

$ cat bar2.c
extern void foo2();

```

(続く)

```
void bar2()
{
    foo2();
}
```

これらの2つのシンボルは、2つの新しい公共インタフェースの定義を目的としています。新しいインタフェースはどちらも相互に関連がありませんが、それぞれ元の SUNW_1.2 インタフェースへの依存関係を表わします。

次の mapfile 定義は、必要な関連付けを作成します。

```
$ cat mapfile
SUNW_1.1 {                # Release X
    global:
        foo1;
    local:
        *;
};
SUNW_1.2 {                # Release X+1
    global:
        foo2;
} SUNW_1.1;
SUNW_1.2.1 { } SUNW_1.2; # Release X+2
SUNW_1.3a {              # Release X+3
    global:
        bar1;
} SUNW_1.2;
SUNW_1.3b {              # Release X+3
    global:
        bar2;
} SUNW_1.2;
```

ここでも、この mapfile を使用して libfoo.so.1 に作成されたバージョン定義とそれらに関連する依存関係は、pvs(1) を使用して検査できます。

```
$ cc -o libfoo.so.1 -M mapfile -G foo.o bar1.o bar2.o data.o
$ pvs -dv libfoo.so.1
libfoo.so.1;
SUNW_1.1;
SUNW_1.2: {SUNW_1.1};
SUNW_1.2.1 [WEAK]: {SUNW_1.2};
SUNW_1.3a: {SUNW_1.2};
SUNW_1.3b: {SUNW_1.2};
```

次の節では、これらのバージョン定義記録を使用して、実行時結合の条件を検査し、オブジェクトの作成中にその結合を制御する方法について説明します。

バージョン定義への結合

動的実行可能ファイルまたは共有オブジェクトが、他の共有オブジェクトに対して構築される場合、これらの依存関係は結果オブジェクトに記録されます (詳細は、31ページの「共有オブジェクトの処理」と109ページの「共有オブジェクト名の記録」を参照)。これらの共有オブジェクトの依存関係にバージョン定義も含まれる場合、関連のバージョン依存関係は結果オブジェクトに記録されます。

次の例は、前述のデータファイルを取り上げて、コンパイル時環境に適した共有オブジェクトを生成しています。この共有オブジェクト `libfoo.so.1` は、次の結合例で使用されます。

```
$ cc -o libfoo.so.1 -h libfoo.so.1 -M mapfile -G foo.o bar.o \ data.o
$ ln -s libfoo.so.1 libfoo.so
$ pvs -dsv libfoo.so.1
libfoo.so.1:
    _end;
    _GLOBAL_OFFSET_TABLE_;
    _DYNAMIC;
    _edata;
    _PROCEDURE_LINKAGE_TABLE_;
    _etext;
SUNW_1.1:
    foo1;
    SUNW_1.1;
SUNW_1.2:
    foo2;
    SUNW_1.2;
SUNW_1.2.1 [WEAK]:
    SUNW_1.2.1;
SUNW_1.3a:
    bar1;
    SUNW_1.3a;
SUNW_1.3b:
    bar2;
    SUNW_1.3b
```

実際には、この共有オブジェクトによって提供される6つの公共インタフェースがあります。これらのインタフェースのうち4つ (`SUNW_1.1`、`SUNW_1.2`、`SUNW_1.3a`、および `SUNW_1.3b`) は1組の関数を定義し、1つ (`SUNW_1.2.1`) は共有オブジェクトに対する内部実装の変更を記述し、もう1つ (`libfoo.so.1`) はいくつかの予約ラベルを定義します。このオブジェク

トによって作成される動的オブジェクトは、それらが結合するインタフェースがどれかを記録します。

次の例では、両方のシンボル `foo1` と `foo2` を参照するアプリケーションを作成しています。アプリケーションに記録されるバージョンアップ依存関係に関する情報は、`pvs(1)` に `-r` オプションを付けて使用して調べることができます。

```
$ cat prog.c
extern void foo1();
extern void foo2();

main()
{
    foo1();
    foo2();
}
$ cc -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
    libfoo.so.1 (SUNW_1.2, SUNW_1.2.1);
```

この例では、アプリケーション `prog` は、実際に2つのインタフェース `SUNW_1.1` と `SUNW_1.2` に結合されています。これらのインタフェースが、大域シンボル `foo1` と `foo2` をそれぞれ提供したためです。

ただし、バージョン定義 `SUNW_1.1` はバージョン定義 `SUNW_1.2` から継承されたものとして `libfoo.so.1` 内に定義されているため、後者のバージョン依存関係だけを記録する必要があります。バージョン定義依存関係のこの正規化によって、オブジェクト内に保持して、実行時に処理する必要があるバージョン情報は削減されます。

アプリケーション `prog` は、ウィークバージョン定義 `SUNW_1.2.1` を含む共有オブジェクトの実装状態に対して構築されるため、この依存関係も記録されます。このバージョン定義は、バージョン定義 `SUNW_1.2` を継承するように定義されていますが、バージョンのウィーク性は `SUNW_1.1` によるその正規化を阻害するため、依存関係は別々に記録されます。

相互に継承される複数のウィークバージョン定義がある場合、これらの定義は、ウィークでないバージョン定義と同じ方法で正規化されます。

注 - バージョン依存関係の記録は、リンカーの `-z noversion` オプションによって抑制できます。

これらのバージョン定義依存関係の記録を終えると、実行時リンカーは、アプリケーションの実行時に結合されたオブジェクト内に必要なバージョン定義がある

かどうかを検査します。この検査は、`ldd(1)` に `-v` オプションを付けて使用して表示できます。たとえば、アプリケーション `prog` に対して、`ldd(1)` を実行すると、バージョン定義依存関係は、共有オブジェクト `libfoo.so.1` で正しく検出されることがわかります。

```
$ ldd -v prog

find object=libfoo.so.1; required by prog
  libfoo.so.1 => ./libfoo.so.1
find version=libfoo.so.1;
  libfoo.so.1 (SUNW_1.2) => ./libfoo.so.1
  libfoo.so.1 (SUNW_1.2.1) => ./libfoo.so.1
....
```

注 - `ldd(1)` に `-v` オプションを付けると、詳細出力が暗黙のうちに指定されます。この出力では、すべての依存関係の再帰的なリストが、すべてのバージョンアップ条件とともに生成されます。

ウィークでないバージョン定義依存関係を検出できないと、アプリケーションの初期設定中に重大なエラーが起こります。検出できないウィークバージョン定義依存関係は、暗黙の内に無視されます。たとえば、`libfoo.so.1` がバージョン定義 `SUNW_1.1` だけを含む環境で、アプリケーション `prog` が実行された場合は、次の重大なエラーが生じます。

```
$ pvs -dv libfoo.so.1
  libfoo.so.1;
  SUNW_1.1;
$ prog
ld.so.1: prog: fatal: libfoo.so.1: version 'SUNW_1.2' not \
found (required by file prog)
```

アプリケーション `prog` がバージョン定義依存関係を記録しなかった場合は、必要なインタフェースシンボル `foo2` が存在しないこと自体が、アプリケーションの実行中に重大な再配置エラーとして現われます (80ページの「再配置エラー」を参照)。この再配置エラーは、プロセス初期設定中またはプロセス実行中に生じる可能性があります。また、アプリケーションの実行パスが関数 `foo2` を呼び出さなかった場合には、まったく生じないこともあります。

バージョン定義依存関係を記録すると、アプリケーションによって必要なインタフェースが使用可能かどうかはすぐに示されます。

libfoo.so.1 がバージョン定義 SUNW_1.1 と SUNW_1.2 だけを含む環境内でアプリケーション prog が実行された場合、ウィークでないバージョン定義条件はすべて満たされます。ウィークバージョン定義 SUNW_1.2.1 の不在は重大ではないエラーと見なされるため、実行時エラー条件は生成されません。ただし、ldd(1) を使用すると、検出できないすべてのバージョン定義が表示されます。

```
$ pvs -dv libfoo.so.1
    libfoo.so.1;
    SUNW_1.1;
    SUNW_1.2:                {SUNW_1.1};
$ prog
string used by foo1()
string used by foo2()
$ ldd prog
    libfoo.so.1 =>    ./libfoo.so.1
    libfoo.so.1 (SUNW_1.2.1) =>    (version not found)
    .....
```

注 - オブジェクトが指定の依存関係からのバージョン定義を必要としている場合、実行時にその依存関係の実装状態にバージョン定義情報が含まれていないことがわかると、依存関係のバージョン検査は暗黙の内に無視されます。この方針は、非バージョンアップ共有オブジェクトからバージョンアップ共有オブジェクトへの移行が行われるときに、下位互換性レベルを提供するものです。ただし、ldd(1) は、バージョン条件の違いを表示するために引き続き使用できます。

追加オブジェクトのバージョンの検査

バージョン定義シンボルも、dlopen(3DL) によって取得されたオブジェクトのバージョン条件を検査するメカニズムとなるものです。この関数を使用してプロセスのアドレス空間に追加されたオブジェクトに対しては、実行時リンカーによる自動バージョン依存関係検査が行われません。このため、この関数の呼び出し元が、バージョンアップ条件が適合しているかどうかを検査する必要があります。

必要なバージョン定義があるかどうかは、dlsym(3DL) を使用して、関連のバージョン定義シンボルを調べることによって検査できます。次の例は、dlopen(3DL) によってプロセスに追加され、SUNW_1.2 が使用可能かどうかを確認される共有オブジェクト libfoo.so.1 を示しています。

```

#include      <stdio.h>
#include      <dlfcn.h>

main()
{
    void *      handle;
    const char * file = "libfoo.so.1";
    const char * vers = "SUNW_1.2";
    ....

    if ((handle = dlopen(file, RTLD_LAZY)) == NULL) {
        (void) printf("dlopen: %s\n", dlerror());
        exit (1);
    }

    if (dlsym(handle, vers) == NULL) {
        (void) printf("fatal: %s: version '%s' not found\n",
            file, vers);
        exit (1);
    }
    ....
}

```

バージョン結合の指定

バージョン定義を含む共有オブジェクトに対して動的オブジェクトを作成する場合、リンカーに対して、特定のバージョン定義への結合を制限するように指示することができます。リンカーを使用すると、特定インタフェースへのオブジェクトの結合を効果的に制御することができます。

オブジェクトの結合条件は、ファイル制御命令によって制御できます。この命令は、リンカーの `-M` オプションと関連の `mapfile` を使用して提供されます。これらのファイル制御 `mapfile` 命令の構文は、次のとおりです。

```
name - version [ version ... ] [ $ADDVERS=version ];
```

- **Name** — 共有オブジェクト依存関係の名前を表す。この名前は、リンカーによって使用される共有オブジェクトのコンパイル環境名と一致しなければなりません (33ページの「ライブラリの命名規約」を参照)
- **Version** — 結合に使用可能でなければならない共有オブジェクト内のバージョン定義名を表わす。複数のバージョン定義を指定できる
- **\$ADDVERS** — 追加バージョン定義を記録できるようにする

次のように、この結合制御が役立つシナリオがいくつかあります。

- 共有オブジェクトが、一意の依存しないバージョンを定義するようにバージョンアップされていて、異なる標準インタフェースを定義する可能性が高い場

合、アプリケーションでは、その結合が特定インタフェースの条件を満たすように保証できる

- 共有オブジェクトがいくつかのソフトウェアリリースに対してバージョンアップされている場合、アプリケーション開発者は、前のソフトウェアリリースで使用可能であったインタフェースだけを使用するように制限できる。したがって、アプリケーションインタフェースの条件が共有オブジェクトの以前のリリースと一致するという知識を前提として、最新リリースの共有オブジェクトを使用してアプリケーションを構築することができる

次に、バージョン制御機構の使用例を示します。この例では、次のバージョンインタフェース定義を含む共有オブジェクト `libfoo.so.1` を使用しています。

```
$ pvs -dsv libfoo.so.1
libfoo.so.1:
    _end;
    _GLOBAL_OFFSET_TABLE_;
    _DYNAMIC;
    _edata;
    _PROCEDURE_LINKAGE_TABLE_;
    _etext;
SUNW_1.1:
    foo1;
    foo2;
    SUNW_1.1;
SUNW_1.2:      {SUNW_1.1}:
    bar;
```

バージョン定義 `SUNW_1.1` および `SUNW_1.2` は、ソフトウェア Release X および Release X+1 で使用可能な `libfoo.so.1` 内のインタフェースをそれぞれ表わします。

アプリケーションは、次のバージョン制御 `mapfile` 命令を使用して、Release X で使用可能なインタフェースだけに結合するように構築できます。

```
$ cat mapfile
libfoo.so - SUNW_1.1;
```

たとえば、アプリケーション `prog` を開発する場合、アプリケーションが Release で実行されるようにすると、アプリケーションではそのリリースで使用可能なインタフェースだけを使用できます。アプリケーションがシンボル `bar` を間違えて参照すると、そのアプリケーションが必要なインタフェースに準拠していないことが、未定義のシンボルエラーとして、リンカーによって通知されます。

```

$ cat prog.c
extern void foo1();
extern void bar();

main()
{
    foo1();
    bar();
}
$ cc -o prog prog.c -M mapfile -L. -R. -lfoo
Undefined      first referenced
 symbol        in file
bar             prog.o (symbol belongs to unavailable \
              version ./libfoo.so (SUNW_1.2))
ld: fatal: Symbol referencing errors. No output written to prog

```

SUNW_1.1 インタフェースに準拠するには、bar への参照を削除する必要があります。これは、アプリケーションを再処理して bar に対する条件を削除するか、または bar の実装をアプリケーションの作成に追加することによって行います。

追加バージョン定義への結合

通常のオブジェクトシンボル結合からバージョン依存関係を作成するよりも、追加バージョン依存関係を記録した方がよい場合があります。これは、\$ADDVERS ファイル制御命令を使用して実行できます。この節では、この追加結合が役に立ついくつかのシナリオについて説明します。

libfoo.so.1 の例に続いて、Release X+2 において、バージョン定義 SUNW_1.1 が 2 つの標準リリース STAND_A と STAND_B に分割される場合を想定します。互換性を維持するには、SUNW_1.1 バージョン定義を維持する必要がありますが、ここでは、2 つの標準定義を継承するものとして表わされています。

```

$ pvs -dsv libfoo.so.1
libfoo.so.1:
    _end;
    _GLOBAL_OFFSET_TABLE_;
    _DYNAMIC;
    _edata;
    _PROCEDURE_LINKAGE_TABLE_;
    _etext;
SUNW_1.1:      {STAND_A, STAND_B}:
    SUNW_1.1;
SUNW_1.2:      {SUNW_1.1}:
    bar;
STAND_A:
    foo1;

```

(続く)

```

        STAND_A;
STAND_B:
        foo2;
        STAND_B;

```

アプリケーション `prog` の唯一の条件がインタフェースシンボル `foo1` であるようにその作成を続けた場合、このアプリケーションはバージョン定義 `STAND_A` に対して単一の依存関係を持ちます。このことは、インタフェース `foo1` が以前のリリースで存在したけれども、バージョン定義 `STAND_A` は存在しなかったために `libfoo.so.1` が Release `X+2` よりも小さいシステムでの `prog` の実行を阻害します。

したがって、アプリケーション `prog` は、次のファイル制御命令を使用して `SUNW_1.1` に対する依存関係を作成することによって、その条件を以前のリリースに合わせて構築できます。

```

$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.1;
$ cat prog
extern void foo1();

main()
{
    foo1();
}
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
    libfoo.so.1 (SUNW_1.1);

```

この明示的な依存関係は、真の依存関係の条件をカプセル化し、旧リリースとの互換性を保つのに十分なものです。

145ページの「ウィークバージョン定義の作成」では、ウィークバージョン定義を使用して、内部実装の変更をマークする方法について説明しました。これらのバージョン定義は、オブジェクトに対して行われたバグ修正と性能の改善に適しています。アプリケーションを正しく実行するためにウィークバージョンが必要な場合は、このバージョン定義への明示的な依存関係を生成できます。

バグ修正や性能の改善がアプリケーションを正しく機能させるために重要な場合は、このような依存関係の確立も重要になります。

引き続き libfoo.so.1 の例で、バグ修正がウィークバージョン定義 SUNW_1.2.1 としてソフトウェア Release X+3 に組み込まれている場合を想定します。

```
$ pvs -dsv libfoo.so.1
libfoo.so.1:
    _end;
    _GLOBAL_OFFSET_TABLE_;
    _DYNAMIC;
    _edata;
    _PROCEDURE_LINKAGE_TABLE_;
    _etext;
SUNW_1.1:      {STAND_A, STAND_B}:
    SUNW_1.1;
SUNW_1.2:      {SUNW_1.1}:
    bar;
STAND_A:
    foo1;
    STAND_A;
STAND_B:
    foo2;
    STAND_B;
SUNW_1.2.1 [WEAK]: {SUNW_1.2}:
    SUNW_1.2.1;
```

通常、アプリケーションは、この共有オブジェクトに対して構築されている場合、バージョン定義 SUNW_1.2.1 に対するウィークである依存関係を記録します。この依存関係は情報提供だけを目的とするものであり、バージョン定義が実行時に使用される libfoo.so.1 に見つからなくても、アプリケーションを終了させません。

ファイル制御命令 \$ADDVERS を使用すると、バージョン定義に対する明示的な依存関係を生成できます。この定義がウィークである場合、この明示的参照によって、バージョン定義が強い依存関係に高められます。

したがって、アプリケーション prog は、次のファイル制御命令を使用し、SUNW_1.2.1 インタフェースを実行時に使用できるという条件を実施するように構築できます。

```
$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.2.1;
$ cat prog
extern void fool();

main()
{
    fool();
}
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
```

(続く)

```
libfoo.so.1 (SUNW_1.2.1);
```

ここで、`prog` は、インタフェース `STAND_A` に対する明示的な依存関係によって構築されています。バージョン定義 `SUNW_1.2.1` は、強いバージョンに高められているため、依存関係 `STAND_A` によっても正規化されます。実行時にバージョン定義 `SUNW_1.2.1` が見つからないと、重大なエラーが生成されます。

注 - 1 つまたは 2 つの依存関係を処理する場合、バージョン定義への明示的な結合は、バージョン定義シンボルを参照することによっても達成できます。この処理は、リンカーの `-u` オプションを使用すると、最も簡単に実行できます。ただし、シンボル参照は非選択的であり、類似の名前を持つ複数のバージョン定義を含む可能性がある複数の依存関係を処理する場合は、この手法で明示的な結合を作成することはできません。

バージョンの安定性

この章のこれまでの情報から、明らかに、個々のバージョンの定義がそのオブジェクトの使用期間にわたって変更されなければ、1 つのオブジェクト内の複数のバージョンに結合する各種のモデルは損なわれません。

一度オブジェクトに対してバージョン定義を作成して公開したら、そのバージョン定義は、そのオブジェクトの次のリリースでも変更されずに存在していなければなりません。バージョン名およびそれに関連するシンボルは両方とも変更しないでください。このため、バージョン定義内で定義されるシンボル名には、ワイルドカードによる拡張はサポートされていません。これは、ワイルドカードに当てはまるシンボルの数が、オブジェクトが発展する過程で異なる場合があるからです。

再配置可能オブジェクト

前の節では、動的オブジェクト内でバージョン情報を記録して使用方法について説明しました。再配置可能オブジェクトは、同様の方法でバージョンアップ情報を保持できますが、これらの情報の使用方法に多少違いがあります。

再配置可能オブジェクトのリンク編集に提供されるバージョン定義はすべて、前の例で説明したものとまったく同じ形式で記録されます。ただしデフォルトにより、作成中のオブジェクトに対するシンボル削減は実行されません。代わりに、再配置可能オブジェクトが動的オブジェクトの生成に対して最終的に入力として使用されると、バージョン記録自体が、適用するシンボル削減を判定するために使用されます。

また、再配置可能オブジェクトで検出されたバージョン定義はすべて、動的オブジェクトに伝達されます。再配置可能オブジェクトでのバージョン処理の例については、56ページの「シンボル範囲の縮小」を参照してください。

外部バージョンアップ

共有オブジェクトへの実行時参照は、常にファイルのバージョンファイル名を参照しなければなりません。通常、これはバージョン番号が接尾辞として付いたファイル名として表わされます。共有オブジェクトのインタフェースが互換性のない方法で(古いアプリケーションを破壊するような方法で)変更される場合は、新しい共有オブジェクトを新しいバージョンアップファイル名によって配布する必要があります。また、元のバージョンアップファイル名も配布して、古いアプリケーションに必要なインタフェースを提供する必要があります。

共有オブジェクトを個別のバージョンアップファイル名として実行時環境内に提供すると、一連のソフトウェアリリースに対して構築されたアプリケーションで、それらを構築する基本となったインタフェースを実行中の結合に使用できるようにすることができます。

次の節では、コンパイル環境と実行時環境間でのインタフェースの結合を同期する方法について説明します。

バージョンアップファイル名の管理

108ページの「命名規約」では、リンク編集に共有オブジェクトを入力するための最も一般的な手法は、`-l` オプションを使用する方法であると述べました。このオプションは、リンカーのライブラリ検索機構を使用して接頭辞 `lib` と接尾辞 `.so` が付いた共有オブジェクトを探します。

ただし、実行時に、共有オブジェクト依存関係は、そのバージョンアップファイル名形式で存在していなければなりません。これらのオブジェクトのバージョンを管理するための最も一般的な手法は、これらの命名規約に従う 2 つの異なる共有オブ

ジェクトを維持する方法ではなく、2つのファイル名間にファイルシステムリンクを作成する方法です。

実行時共有オブジェクト `libfoo.so.1` をコンパイル環境で使用できるようにするには、コンパイルファイル名から実行時ファイル名にシンボルリンクを与える必要があります。次に例を示します。

```
$ cc -o libfoo.so.1 -G -K pic foo.c
$ ln -s libfoo.so.1 libfoo.so
$ ls -l libfoo*
lrwxrwxrwx 1 usr grp          11 1991 libfoo.so -> libfoo.so.1
-rwxrwxr-x 1 usr grp        3136 1991 libfoo.so.1
```

注 - シンボルリンクまたはハードリンクを使用できます。ただしマニュアルおよび診断ツールとしては、シンボルリンクの方が有効です。

ここで、共有オブジェクト `libfoo.so.1` は、実行時環境に対して生成されています。シンボルリンク `libfoo.so` の生成は、コンパイル環境でのこのファイルの使用も有効にしています。次に例を示します。

```
$ cc -o prog main.o -L. -lfoo
```

ここで、リンカーは、シンボルリンク `libfoo.so` を追って見つける共有オブジェクト `libfoo.so.1` によって記述されたインタフェースを使用して、再配置可能オブジェクト `main.o` を処理します。

一連のソフトウェアリリースに対して、この共有オブジェクトの新しいバージョンが変更されたインタフェースによって配布される場合、コンパイル環境はシンボルリンクを変更することで適用可能なインタフェースを使用するように構築することができます。次に例を示します。

```
$ ls -l libfoo*
lrwxrwxrwx 1 usr grp          11 1993 libfoo.so -> libfoo.so.3
-rwxrwxr-x 1 usr grp        3136 1991 libfoo.so.1
-rwxrwxr-x 1 usr grp        3237 1992 libfoo.so.2
-rwxrwxr-x 1 usr grp        3554 1993 libfoo.so.3
```

ここでは、共有オブジェクトの3つの主要バージョンが使用できます。これらの共有オブジェクトのうち、`libfoo.so.1` と `libfoo.so.2` の2つは、既存アプリケーションに対する依存関係を提供します。`libfoo.so.3` は、新しいアプリケーションを作成して実行するための最新主要リリースを提供します。

このシンボルリンク機構自体を使用するだけでは、コンパイル環境での使用から実行時環境での条件に合わせて共有オブジェクトを正しく結合することはできません。例が示しているように、リンカーは、動的実行可能ファイル `prog` に、それが処理した共有オブジェクトのファイル名を記録します。この場合、これはコンパイル環境のファイル名です。

```
$ dump -Lv prog

prog:
**** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag      Value
[1]      NEEDED   libfoo.so
.....
```

つまり、アプリケーション `prog` が実行されると、実行時リンカーは、依存関係 `libfoo.so` を検索し、結果として、これはこのシンボルリンクが指すすべてのファイルに結合されます。

依存関係として記録される正しい実行時名を指定するには、共有オブジェクト `libfoo.so.1` を `soname` 定義によって構築する必要があります。この定義は、共有オブジェクトの実行時名を識別するもので、この共有オブジェクトに対してリンクするすべてのオブジェクトによって、依存関係名として使用されます。この定義は、共有オブジェクト自体のリンク編集に `-h` オプションを使用して与えることができます。次に例を示します。

```
$ cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 foo.c
$ ln -s libfoo.so.1 libfoo.so
$ cc -o prog main.o -L. -lfoo
$ dump -Lv prog

prog:
**** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag      Value
[1]      NEEDED   libfoo.so.1
.....
```

このシンボルリンクと `soname` 機構は、コンパイル環境と実行時環境の共有オブジェクト命名規約の間に強固な同期を確立しました。これによって、リンク編集に処理されたインタフェースは、生成された出力ファイルに正確に記録されます。この記録によって、意図したインタフェースが実行時に提供されます。

プロセス内での互換性

注意: 外部的にバージョン定義された共有オブジェクトを新しく作成することは重大な変更であり、この共有オブジェクトを使用するすべての処理の完全な依存関係を理解するために注意が必要です。

たとえば、あるアプリケーションが、`libfoo.so.1` および外部から記述されたオブジェクト `libISV.so.1` と依存関係があるとします。この後者のオブジェクトは `libfoo.so.1` とも依存関係があるとします。外部オブジェクト `libISV.so.1` の使用には何の変更も加えずに、`libfoo.so.2` 内の新しいインタフェースを使用するためにアプリケーションを再設計すると、`libfoo.so` の主要なバージョンが両方とも実行プロセスに含まれることとなります。`libfoo.so` のバージョンを変更する理由は互換性のない変更をマークすることだけなので、プロセス内にオブジェクトの両方のバージョンを持つことは、不正なシンボル結合が発生する原因となり、そのために望ましくない相互作用を引き起こすことがあります。

サポートインタフェース

概要

リンカーには、リンカーおよび実行時リンカーの処理を監視し、場合によって変更するための多数のサポートインタフェースがあります。これらのインタフェースでは、通常、前の章で説明したよりもさらに詳しくリンク編集の概念を理解する必要があります。この章では、次のインタフェースについて説明します。

- リンカーのサポートインタフェース(「ld-サポート」)
- 実行時リンカーの監査インタフェース(「rtld-監査」)
- 実行時リンカーのデバuggインタフェース(「rtld-デバugg」)

リンカーのサポートインタフェース

第 2 章で説明したように、リンカーは、ファイルのオープンやこれらのファイルからのセクションの連結を含む多数の操作を実行します。これらの操作の監視、および場合によっては変更は、コンパイルシステムのコンポーネントにとって有益なことがよくあります。

このセクションでは、入力ファイル検査、および場合によってはリンク編集を構成するファイルの入力ファイルデータ変更用にサポートされているインタフェースについて説明します。このインタフェースは、ld-サポートインタフェースと呼ばれます。このインタフェースを使用する 2 つのアプリケーションは、このインタ

フェースを使用して再配置可能オブジェクト内の情報デバッグを処理するリンカーそのものと、このインタフェースを使用して状態情報を保存する `make(1S)` ユーティリティです。

`ld`-サポートインタフェースは、1つまたは複数のサポートインタフェースルーチンを提供するサポートライブラリから構成されています。このライブラリはリンク編集プロセスの一部として読み込まれ、検出されたサポートルーチンはすべてリンク編集の各段階で呼び出されます。

このインタフェースを使用する場合は、`elf(3ELF)` 構造とファイル形式に詳しくなければなりません。

サポートインタフェースの呼び出し

リンカーは、`SGS_SUPPORT` 環境変数またはリンカーの `-S` オプションのどちらかによって提供される1つまたは複数のサポートライブラリを受け入れます。環境変数は、コロンで区切られたサポートライブラリのリストから構成されています。

```
$ SGS_SUPPORT=./support.so.1:libldstab.so.1 cc ...
```

`-S` オプションは、単一のサポートライブラリを指定します。次のように複数の `-S` オプションを指定できます。

```
$ LD_OPTIONS="-S./support.so.1 -Slibldstab.so.1" cc ...
```

サポートライブラリは、共有オブジェクトの1つです。リンカーは、各サポートライブラリに対してライブラリが指定された順序で、`dlopen(3DL)` を実行します。環境変数と `-S` オプションの両方がある場合は、環境変数によって指定されたサポートライブラリが最初に処理されます。各サポートライブラリは、`dlsym(3DL)` によって、サポートインタフェースルーチンがないかどうかをさらに検索されます。これらのサポートルーチンは、リンク編集の各段階で呼び出されます。

注 - デフォルトごとに、Solaris サポートライブラリ `libldstab.so.1` は、リンカーを使用して、入力再配置可能オブジェクト内に提供されるコンパイラ生成デバッグ情報を処理、圧縮します。このデフォルト処理は、`-S` オプションを使用して指定されたサポートライブラリでリンカーを呼び出すと抑止されます。各サポートライブラリサービスだけでなく `libldstab.so.1` のデフォルト処理も必要な場合は、リンカーに提供されたサポートライブラリのリストに `libldstab.so.1` を明示的に追加する必要があります。

32 ビットおよび 64 ビット環境

22ページの「32 ビットおよび 64 ビット環境」で説明しているように、64 ビットリンカー (ld(1)) は 32 ビットのオブジェクトを生成でき、32 ビットリンカーは 64 ビットのオブジェクトを生成できます。これらのオブジェクトはそれぞれ、定義されているサポートインタフェースに関連付けられています。

64 ビットオブジェクトのサポートインタフェースは 32 ビットオブジェクトのサポートインタフェースと似ていますが、末尾に「64」という接尾辞が付きます。たとえば、ld_start() および ld_start64() のようになります。この規則により、サポートインタフェースの両方の実装状態を、libldstab.so.1 という単一の共有オブジェクトの 32 ビットと 64 ビットの各クラスに常駐させることができます。

サポートインタフェース関数

ld-サポートインタフェースはすべて、ヘッダファイル link.h に定義されています。インタフェース引数はすべて、基本的な C タイプまたは ELF タイプです。ELF データタイプは、ELF アクセスライブラリ libelf によって調べることができます (libelf の内容については、elf(3ELF) を参照)。次のインタフェース関数は、ld-サポートインタフェースによって提供されており、予定の使用順序に従って記述されています。

ld_start()

```
void ld_start(const char * name, const Elf32_Half type,
              const char * caller);

void ld_start64(const char * name, const Elf64_Half type,
                const char * caller);
```

この関数は、リンカーコマンド行の初期妥当性検査の後に呼び出されて、入力ファイル処理の開始を示します。

name は、作成される出力ファイル名を示します。type は出力ファイルタイプであり、ET_DYN、ET_REL、ET_EXEC のどれかです (sys/elf.h に定義)。caller は、インタフェースを呼び出すアプリケーションを示します。これは通常、/usr/ccs/bin/ld です。

ld_file()

```

void ld_file(const char * name, const Elf_Kind kind, int flags,
             Elf * elf);

void ld_file64(const char * name, const Elf_Kind kind, int flags,
              Elf * elf);

```

この関数は、ファイルデータの処理が実行される前に、各入力ファイルに対して呼び出されます。

name は処理される入力ファイルを示します。**kind** は入力ファイルのタイプを示し、ELF_K_AR または ELF_K_ELF のどちらかになります (`libelf.h` に定義)。**flags** は、リンカーによるファイルの取得方法を示し、次の 1 つまたは複数の定義にすることができます。

- LD_SUP_DERIVED — ファイル名はコマンド行に明示的に指定されていない。-1 の拡張から派生するか、または抽出されたアーカイブ構成要素を示す
- LD_SUP_INHERITED — ファイルは、コマンド行共有オブジェクトの依存関係として取得される
- LD_SUP_EXTRACTED — ファイルは、アーカイブから抽出される

flags 値が指定されていない場合、入力ファイルはコマンド行に明示的に指定されています。**elf** は、ファイル ELF 記述子へのポインタです。

`ld_section()`

```

void ld_section(const char * name, Elf32_Shdr * shdr,
               Elf32_Word sndx, Elf_Data * data, Elf * elf);

void ld_section64(const char * name, Elf64_Shdr * shdr,
                 Elf64_Word sndx, Elf_Data * data, Elf * elf);

```

この関数は、セクションデータの処理が実行される前に、入力ファイルの各セクションに対して呼び出されます。

name は、入力セクション名を示します。**shdr** は、関連のセクションヘッダへのポインタを示します。**sndx** は、入力ファイル内のセクションインデックスです。**data** は、関連データバッファへのポインタを示します。**elf** は、ファイル ELF 記述子へのポインタです。

データの変更は、データ自体を再割り当てして、Elf_Data バッファに d_buf ポインタを再割り当てすることによって可能になります。データへの変更はすべて、Elf_Data バッファの d_size 要素を保証しなければなりません。出力イメージの一部になる入力セクションでは、d_size 要素をゼロに設定すると、出力イメージからデータが効率的に削除されます。

注・リンカーの `-s` オプションによって取り除かれるセクションや、SHT_SUNW_COMDAT 処理 (267ページの「Comdat セクション」を参照) によって破棄されるセクションは、`ld_section()` に報告されません。

`ld_atexit()`

```
void ld_atexit(int status);  
void ld_atexit64(int status);
```

この関数は、リンク編集の終了時に呼び出されます。

`status` は、リンカーによって返される `exit(2)` コードであり、`EXIT_FAILURE` または `EXIT_SUCCESS` のどちらかです (`stdlib.h` に定義)。

サポートインタフェースの例

次の例では、32 ビットリンク編集の一部として処理される再配置可能オブジェクトファイルのセクション名 (表 7-17 を参照) を出力するサポートライブラリを作成しています。

```
$ cat support.c  
#include <link.h>  
  
static int indent = 0;  
  
void  
ld_start(const char * name, const Elf32_Half type,  
          const char * caller)  
{  
    (void) printf("output image: %s\n", name);  
}  
  
void  
ld_file(const char * name, const Elf_Kind kind, int flags,  
        Elf * elf)  
{
```

(続く)

```

        if (flags & LD_SUP_EXTRACTED)
            indent = 4;
        else
            indent = 2;

        (void) printf("%*sfile: %s\n", indent, "", name);
    }

void
ld_section(const char * name, Elf32_Shdr * shdr, Elf32_Word sndx,
           Elf_Data * data, Elf * elf)
{
    Elf32_Ehdr * ehdr = elf32_getehdr(elf);

    if (ehdr->e_type == ET_REL)
        (void) printf("%*s section [%ld]: %s\n", indent,
                    "", sndx, name);
}

```

このサポートライブラリは、libelf に依存して、入力ファイルタイプを判定するために使用される ELF アクセス関数 elf32_getehdr(3ELF) を提供します。サポートライブラリは、次の行によって構築されます。

```
$ cc -o support.so.1 -G -K pic support.c -lelf -lc
```

次の例は、再配置可能オブジェクトおよびローカル範囲アーカイブライブラリによる簡易アプリケーションの構築の結果生じたセクション診断を示しています。-S オプションを使用すると、デフォルトデバッグ情報処理だけでなく、サポートライブラリの呼び出しも行われます。

```

$ LD_OPTIONS="-S./support.so.1 -Slibldstab.so.1" \
cc -o prog main.c -L. -lfoo

output image: prog
  file: /opt/COMPILER/crti.o
    section [1]: .shstrtab
    section [2]: .text
    .....
  file: /opt/COMPILER/crt1.o
    section [1]: .shstrtab
    section [2]: .text
    .....
  file: /opt/COMPILER/values-xt.o
    section [1]: .shstrtab
    section [2]: .text
    .....

```

(続く)


```
file: main.o
  section [1]: .shstrtab
  section [2]: .text
  .....
file: ./libfoo.a
  file: ./libfoo.a(foo.o)
  section [1]: .shstrtab
  section [2]: .text
  .....
file: /usr/lib/libc.so
file: /opt/COMPILER/crt.o
  section [1]: .shstrtab
  section [2]: .text
  .....
file: /usr/lib/libdl.so.1
```

注 - この例で表示されるセクションの数は、出力を簡素化するために減らされています。また、コンパイラドライバによって取り込まれるファイルも異なる場合があります。

実行時リンカーの監査インタフェース

第3章で説明したように、実行時リンカーは、メモリーへのオブジェクトの割り当てや記号の結合を含む多数の操作を実行します。これらの操作を同じプロセス内から監視、および場合によっては変更すると強力なプロセス監視ツールになります。

このセクションでは、プロセスに関する実行時リンク情報にアクセスするために、そのプロセスにサポートされているインタフェースについて説明します。このインタフェースは `rtld`-監査インタフェースと呼ばれます。この機構の使用例としては、135ページの「共有オブジェクトのプロファイリング」で説明している共有オブジェクトの実行時プロファイルがあります。

`rtld`-監査インタフェースは、1つまたは複数の監査インタフェースルーチンを提供する監査ライブラリとして実装されます。このライブラリがプロセスの一部として読み込まれている場合は、プロセス実行の各段階で、実行時リンカーによって監査ルーチンが呼び出されます。これらのインタフェースを使用すると、監査ライブラリは次のものにアクセスできます。

- 依存関係の検索。検索パスは監査ライブラリによって置き換えることができる
- 読み込まれているオブジェクトに関する情報
- 読み込まれているこれらのオブジェクト間で発生するシンボル結合。これらの結合は、監査ライブラリによって変更できる
- 関数呼び出しとその戻り値の監査を可能にするために、プロシージャリンカーテーブルエントリ (317ページの「プロシージャのリンクテーブル (プロセッサ固有)」を参照) によって提供されるレイジー結合機構の開発。関数の引数とその戻り値は、監査ライブラリによって変更できる

これらの機能のいくつかは、特殊な共有オブジェクトを事前に読み込むことによって実現できます (81ページの「追加オブジェクトの読み込み」を参照)。ただし、事前に読み込まれたオブジェクトは、プロセスのオブジェクトと同じ名前空間内にあります。このことは、通常、事前に読み込まれた共有オブジェクトの実装を制限したり、複雑化したりします。rtld-監査インタフェースは、ユーザーに対して、監査ライブラリを実行するための固有の名前空間を提供します。これにより、監査ライブラリがプロセス内で発生する通常の結合を妨害することはなくなります。

名前空間の確立

実行時リンカーは、動的実行可能なプログラムをその依存関係と結合すると、リンクマップのリンクリストを生成して、プロセスを記述します。リンクマップ構造は、プロセス内の各オブジェクトを記述し、`/usr/include/sys/link.h` に定義されます。アプリケーションのオブジェクトを結合するために必要な記号検索機構は、このリンクマップリストを検索します。このリンクマップリストは、プロセス記号解決用の名前空間を提供します。

実行時リンカー自体もリンクマップによって記述されると、このリンクマップは、アプリケーションオブジェクトのリストとは異なるリストに維持されます。この結果、実行時リンカーは各自の固有名前空間に常駐することとなるため、実行時リンカー内のサービスにアプリケーションが直接結合されることはなくなります (アプリケーションは、フィルタ `libdl.so.1` を介してのみ、実行時リンカーの公共サービスを要求できます)。

rtld-監査インタフェースは、すべての監査ライブラリを保持するための各自のリンクマップリストを使用します。この結果、監査ライブラリは、アプリケーションの記号結合条件から分離されます。アプリケーションリンクマップリストの検査は、`dlopen (3DL)` によって実行できます。これは、`RTLD_NOLOAD` フラグとともに

に使用すると、監査ライブラリで、オブジェクトを読み込むことなくその存在を照会することができます。

アプリケーションと実行時リンカーのリンクマップリストを定義するために、2つの識別子が `/usr/include/link.h` に定義されています。

```
#define LM_ID_BASE      0      /* application link-map list */
#define LM_ID_LDSO     1      /* runtime linker link-map list */
```

各 `rtld`-監査サポートライブラリには、固有の空きリンクマップ識別子が割り当てられています。

監査ライブラリの作成

監査ライブラリは他の共有オブジェクトと同様に構築されますが、プロセス内の固有の名前空間には、いくつかの注意が必要です。

- すべての依存関係の条件を提供しなければならない
- プロセス内のインタフェースに複数のインスタンスを提供しないシステムインタフェースは、使用できない

監査ライブラリが `printf(3C)` を呼び出す場合、その監査ライブラリは、`libc` への依存関係を定義しなければなりません (49ページの「共有オブジェクトの生成」を参照)。監査ライブラリには、固有の名前空間があるため、監査中のアプリケーションに存在する `libc` によって記号参照を満たすことはできません。監査ライブラリに `libc` への依存関係がある場合は、2つのバージョンの `libc.so.1` がプロセスに読み込まれます。1つはアプリケーションのリンクマップリストの結合条件を満たし、もう1つは監査リンクマップリストの結合条件を満たします。

すべての依存関係が記録された状態で監査ライブラリが構築されるようにするには、リンカーの `-z defs` オプションを使用します (49ページの「共有オブジェクトの生成」を参照)。

システムインタフェースには、プロセスにおける実装状態の唯一のインスタンスであると想定して存在するものもあります。たとえば、スレッド、シグナル、および `malloc(3C)` などです。このようなインタフェースを使用すると、アプリケーションの動作が不正に変更されるおそれがあるため、監査ライブラリでは、このようなインタフェースの使用を避ける必要があります。

注 - `mapmalloc(3MALLOC)` を使用した監査ライブラリによるメモリー割り当ては受け入れられます。これは、アプリケーションによって通常使用される割り当てスキーマとこの割り当てが共存可能なためです。

監査インタフェースの呼び出し

`rtld_audit` インタフェースは、次のいずれかの方法によって有効になります。それぞれの方法は、監視対象のオブジェクトの範囲を意味します。

- 「大域」監査は、実行時リンカーの環境変数 `LD_AUDIT` を使用することにより有効になる。この方法により使用可能になる監査ライブラリには、プロセスが使用するすべての動的オブジェクトに関する情報が提供される
- 「ローカル」監査は、オブジェクトの作成時にオブジェクト内に記録された動的エントリによって有効になる (173ページの「ローカル監査の記録」の「ローカル監査の記録」を参照)。この方法によって使用可能になる監査ライブラリには、監査する動的オブジェクトに関する情報が提供される

それぞれの呼び出し方法は、`dlmopen(3DL)` によって読み込まれる共有オブジェクトをコロんで区切ったリストを含む文字列で構成されています。各オブジェクトは、各自の監査リンクマップリストに読み込まれます。また、各オブジェクトは、`dlsym(3DL)` によって、監査ルーチンがないか検索されます。検出された監査ルーチンは、アプリケーション実行中に各段階で呼び出されます。

`rtld_audit` インタフェースを使用すると、複数の監査ライブラリを与えることができます。この方法で使用される監査ライブラリは、通常実行時リンカーによって返される結合を変更することはできません。もし変更すると、後に続く監査ライブラリで予期しない結果が生じます。

安全なアプリケーション (82ページの「動的依存関係のレイジーローディング」を参照) は、トラストディレクトリから監査ライブラリだけを取得できます。現在監査ライブラリに使用できるトラストディレクトリは、32ビットオブジェクトの場合は `/usr/lib/secure` と、64ビットオブジェクトの場合は `/usr/lib/secure/64` だけです。

ローカル監査の記録

ローカル監査要求は、オブジェクトがリンカーオプション `-p` あるいは `-P` を使用して作成された場合に確立できます。開発者が、共有オブジェクト `libfoo.so.1` の使用状況を、監査ライブラリ `audit.so.1` を使用して監査したい場合は、リンク編集時に `-p` オプションを使用してこの要求を記録すれば、その監査を行うことができます。

```
$ cc -G -o libfoo.so.1 -Wl,-paudit.so.1 -Kpic foo.c
$ dump -Lv libfoo.so.1 | fgrep AUDIT
[3]  AUDIT      audit.so.1
```

実行時には、この監査識別子があることにより監査ライブラリが読み込まれ、識別するオブジェクトに関する情報がその監査ライブラリに渡されます。

ただし、この仕組みだけでは、識別するオブジェクトの検索などの情報は監査ライブラリが読み込まれる前に発生してしまいます。できるだけ多くの監査情報を提供するため、ローカル監査を要求するオブジェクトの存在は、そのオブジェクトのユーザーに広く知らされます。たとえば、`libfoo.so.1` に依存するアプリケーションを作成すると、そのアプリケーションは、その依存関係の監査が必要であることを示すよう認識されます。

```
$ cc -o main main.c libfoo.so.1
$ dump -Lv main | fgrep AUDIT
[5]  DEPAUDIT   audit.so.1
```

この仕組みによって使用できる監査の結果、監査ライブラリが読み込まれ、アプリケーションの明白な依存関係すべてに関する情報が渡されます。この依存関係の監査は、リンカーの `-P` オプションを使用することにより、オブジェクトの作成時に直接記録することもできます。

```
$ cc -o main main.c -Wl,-Paudit.so.1
$ dump -Lv main | fgrep AUDIT
[5]  DEPAUDIT   audit.so.1
```

注 - 環境変数 LD_NOAUDIT を NULL 以外の値に設定すると、実行時に監査を無効にすることができます。

監査インタフェースの関数

次の関数が rtld-監査インタフェースによって提供されており、予定の使用順序で記述されています。

注 - アーキテクチャあるいはオブジェクトクラス固有のインタフェースの参照は、説明を簡潔にするために、通常は総称に省略します。たとえば、la_symbind32() および la_symbind64() は la_symbind() で表します。

la_version()

```
uint_t la_version(uint_t version);
```

この関数は、実行時リンカーと監査ライブラリの間で初期接続を提供します。このインタフェースを読み込むには、監査ライブラリによってこれを提供する必要がある場合があります。

実行時リンカーは、サポート可能な最上位バージョンの rtld-監査によって、このインタフェースを呼び出します。監査ライブラリは、このバージョンが十分に使用できるかどうかを確認して、使用する予定のバージョンを返すことができます。このバージョンは、通常、/usr/include/link.h に定義されている LAV_CURRENT です。

監査ライブラリがゼロのバージョン、または実行時リンカーがサポートする rtld-監査インタフェースよりも大きい値を返す場合は、監査ライブラリは使用されません。

la_activity()

```
void la_activity(uintptr_t * cookie, uint_t flag);
```

この関数は、リンク対応付けアクティビティが行われていることを監査プログラムに知らせます。

cookie は、リンク対応付けの先頭のオブジェクトを指します。flags は、/usr/include/link.h に定義されているものと同じタイプのアクティビティを指します。

- LA_ACT_ADD — リンク対応付けリストにオブジェクトが追加される
- LA_ACT_DELETE — リンク対応付けリストからオブジェクトが削除される
- LA_ACT_CONSISTENT — オブジェクトのアクティビティが完了した

la_objsearch()

```
char * la_objsearch(const char * name, uintptr_t * cookie, uint_t flag);
```

この関数は、オブジェクトの検索が行われていることを監査プログラムに知らせます。

`name` は、検索しているファイルあるいはパス名を指します。`cookie` は、検索を開始しているオブジェクトを指します。`flags` は、`/usr/include/link.h` に定義されているものと同じ `name` の起因を指します。

- LA_SER_ORIG — 最初の検索名。通常、DT_NEEDED エントリとして記録されたファイル名あるいは `dlopen(3DL)` に与えられた引数を指す
- LA_SER_LIBPATH — 名前が `LD_LIBRARY_PATH` エントリから作成されている
- LA_SER_RUNPATH — 名前が「実行パス」エントリから作成されている
- LA_SER_CONFIG — 名前が、構成ファイルで指定されたデフォルトの検索パスエントリから作成されている (`crle(1)` のマニュアルページを参照)
- LA_SER_DEFAULT — 名前がデフォルトの検索パスエントリから作成されている
- LA_SER_SECURE — `LA_SER_CONFIG` および `LA_SER_DEFAULT` に追加され、デフォルトのパスエントリがセキュアオブジェクトに適用されることを示す

戻り値は、実行時リンカーが処理を継続する必要がある検索パス名を示し、値 0 は、このパスが無視されることを示します。検索パスを監視する監査ライブラリは、`name` を返します。

la_objopen()

```
uint_t la_objopen(Link_map * lmp, Lmid_t lmid, uintptr_t * cookie);
```

この関数は、新しいオブジェクトが実行時リンカーによって読み込まれるたびに呼び出されます。

`lmp` は、新しいオブジェクトを記述するリンクマップ構造を提供します。`lmid` は、オブジェクトが追加されているリンクマップリストを特定します (170ページの「名前空間の確立」を参照)。`cookie` は、識別子へのポインタを提供します。この識別子は、オブジェクト `lmp` に初期設定されますが、監査ライブラリによって、オブジェクトを他の `rtdl`-監査インタフェースルーチンに対して特定するように変更できます。

この関数は、このオブジェクトで問題になるシンボル結合を示す値を返し、後に `la_symbind()` を呼び出します。この結果の値は、`/usr/include/link.h` に定義された次の値のマスクです。

- `LA_FLG_BINDTO` — このオブジェクトに対する監査シンボル結合
- `LA_FLG_BINDFROM` — このオブジェクトからの監査シンボル結合

これらの2つのフラグの使用法については、`la_symbind()` を参照してください。

ゼロの戻り値は、結合情報がこのオブジェクトで問題にならないことを示します。

`la_preinit()`

```
void la_preinit(uintptr_t * cookie);
```

この関数は、すべてのオブジェクトがアプリケーションに読み込まれた後で、アプリケーションへの制御の譲渡が発生する前に一度呼び出されます。

`cookie` は、プロセスを開始したプライマリオブジェクト、通常は動的実行可能プログラムを表します。

`la_symbind()`

```
uintptr_t la_symbind32(Elf32_Sym * sym, uint_t ndx,
                      uintptr_t * refcook, uintptr_t * defcook, uint_t * flags);

uintptr_t la_symbind64(Elf64_Sym * sym, uint_t ndx,
                      uintptr_t * refcook, uintptr_t * defcook, uint_t * flags,
                      const char * sym_name);
```

この関数は、結合通知のタグが付けられた2つのオブジェクト間で結合が発生すると呼び出されます (`la_objopen()` を参照)。

`sym` は、構築された記号構造 (`/usr/include/sys/elf.h` を参照) であり、`sym->st_value` は結合中の記号定義のアドレスを示しま

す。la_symbind32() は、sym->st_name を調整して実際の記号名を指していますが、la_symbind64() は sym->st_name を結合オブジェクトの文字列テーブルのインデックスのままにしています。

ndx は、結合オブジェクト動的記号テーブル内の記号インデックスを示します。refcook は、この記号への参照を行うオブジェクトを記述します。この識別子は、LA_FLG_BINDFROM を返した la_objopen() に渡されたものと同じです。defcook は、この記号を定義するオブジェクトを記述します。この識別子は、LA_FLG_BINDTO を返した la_objopen() に渡されるものと同じです。

flags は、結合に関する情報を伝達し、プロシージャのリンクテーブルシンボルエントリの連続監査を変更するために使用することができるデータ項目を指します。この値は、/usr/include/link.h に定義された次のフラグのマスクです。

- LA_SYMB_NOPLTENTER — la_pltenter() 関数は、この記号に対しては呼び出されない
- LA_SYMB_NOPLTEXTIT — la_pltexit() 関数は、この記号に対しては呼び出されない
- LA_SYMB_DLSYM — dlsym(3DL) を呼び出した結果発生したシンボル結合
- LA_SYMB_ALTVALUE (LAV_VERSION2) — la_symbind() への以前の呼び出しによって、記号値に対して代替値が返される

デフォルトにより、la_pltenter() または la_pltexit() 関数が監査ライブラリ内に存在する場合、シンボルが参照されるたびにこれらはプロシージャのリンクテーブル記号に対して la_symbind() の後で呼び出されます (181ページの「監査インタフェースの制限」も参照)。

戻り値は、この呼び出しに続いて制御を渡す必要があるアドレスを示します。シンボル結合を監視するだけの監査ライブラリは、sym->st_value の値を返すため、制御は結合記号定義に渡されます。監査ライブラリは、異なる値を返すことによって、シンボル結合を意図的にリダイレクトできます。

sym_name は、la_symbind64() のみに適用されますが、処理されるシンボルの名前を含みます。これは、32 ビットインタフェースから sym->st_name フィールドで使用できます。

la_pltenter()

```

uintptr_t la_sparcv8_pltenter(Elf32_Sym * sym, uint_t ndx,
                             uintptr_t * refcook, uintptr_t * defcook,
                             La_sparcv8_regs * regs, uint_t * flags);

uintptr_t la_sparcv9_pltenter(Elf64_Sym * sym, uint_t ndx,
                             uintptr_t * refcook, uintptr_t * defcook,
                             La_sparcv9_regs * regs, uint_t * flags,
                             const char * sym_name);

uintptr_t la_i86_pltenter(Elf32_Sym * sym, uint_t ndx,
                         uintptr_t * refcook, uintptr_t * defcook,
                         La_i86_regs * regs, uint_t * flags);

```

これらの関数は、結合通知のタグが付けられた2つのオブジェクト間のプロシージャのリンクシンボルエントリが呼び出されると、SPARC および IA システムでそれぞれ呼び出されます (`la_objopen()` と `la_symbind()` を参照)。

`sym`、`ndx`、`refcook`、`defcook`、および `sym_name` は、`la_symbind()` に渡されたものと同じ情報を提供します。

`regs` は、`/usr/include/link.h` に定義されているように、SPARC システム上の `out` レジスタと、IA システム上の `stack` および `frame` レジスタを指します。

`flags` は、結合に関する情報を伝達し、プロシージャのリンクテーブルエントリの連続監査を変更するために使用することができるデータ項目を指します。このデータ項目は、`la_symbind()` から `flags` によって指されるものと同じです。この値は、`/usr/include/link.h` に定義された次のフラグのマスクです。

- `LA_SYMB_NOPLTENTER` — `la_pltenter()` は、この記号では再び呼び出されることはない
- `LA_SYMB_NOPLTEXIT` — `la_pltexit()` は、この記号では呼び出されない

戻り値は、この呼び出しに続いて制御を渡す必要があるアドレスを示します。シンボル結合を監視するだけの監査ライブラリは、`sym->st_value` の値を返すため、制御は結合記号定義に渡されます。監査ライブラリは、異なる値を返すことによって、シンボル結合を意図的にリダイレクトできます。

`la_pltexit()`

```

uintptr_t la_pltexit(Elf32_Sym * sym, uint_t ndx, uintptr_t * refcook,
                    uintptr_t * defcook, uintptr_t retval);

uintptr_t la_pltexit64(Elf64_Sym * sym, uint_t ndx, uintptr_t * refcook,
                      uintptr_t * defcook, uintptr_t retval, const char * sym_name);

```

この関数は、結合通知のタグが付けられた2つのオブジェクト間のプロシージャのリンク記号項目 (`la_objopen()` と `la_symbind()` を参照) が返されて、制御が呼び出し側に到達するまでの間に呼び出されます。

`sym`、`ndx`、`refcook`、`defcook`、および `sym_name` は、`la_symbind()` に渡されるものと同じ情報を提供します。`retval` は結合関数からの戻りコードです。シンボル結合を監視する監査ライブラリは、`retval` を返します。監査ライブラリは意図的に異なる値を返すことができます。

注 - このインタフェースは実験的なものです。181ページの「監査インタフェースの制限」を参照してください。

`la_objclose()`

```
uint_t la_objclose(uintptr_t * cookie);
```

この関数はオブジェクトに対する終了コードが実行されてから、オブジェクトが読み込みを解除されるまでに呼び出されます (103ページの「デバ깅エイド」を参照)。

`cookie` は、以前の `la_objopen()` から取得されていて、オブジェクトを特定します。戻り値は、ここではすべて無視されます。

監査インタフェースの例

次の単純な例では、動的実行可能プログラム `date(1)` によって読み込まれた各共有オブジェクトの依存関係の名前を出力する、監査ライブラリを作成しています。

```

$ cat audit.c
#include <link.h>
#include <stdio.h>

uint_t
la_version(uint_t version)
{

```

```

        return (LAV_CURRENT);
    }

    uint_t
    la_objjopen(Link_map * lmp, Lmid_t lmid, uintptr_t * cookie)
    {
        if (lmid == LM_ID_BASE)
            (void) printf("file: %s loaded\n", lmp->l_name);
        return (0);
    }
}
$ cc -o audit.so.1 -G -K pic -z defs audit.c -lmapmalloc -lc
$ LD_AUDIT=./audit.so.1 date
file: date loaded
file: /usr/lib/libc.so.1 loaded
file: /usr/lib/libdl.so.1 loaded
file: /usr/lib/locale/en_US/en_US.so.2 loaded
Thur Aug 10 17:03:55 PST 2000

```

監査インタフェースのデモンストレーション

/usr/demo/link_audit の SUNWosdem パッケージには、rtld-監査インタフェースを使用する多数のデモアプリケーションが用意されています。

sotruss

このデモアプリケーションは、指定アプリケーションの動的オブジェクト間でのプロセス呼び出しを追跡します。

whocalls

このデモアプリケーションは、指定アプリケーションに呼び出されるたびに、指定関数のスタックと追跡を行います。

perfcnt

このデモアプリケーションは、指定アプリケーションの各関数で費やされた時間を追跡します。

symbindrep

このデモアプリケーションは、指定アプリケーションを読み込むために実行されたすべてのシンボル結合を報告します。

sotruss(1) と whocalls(1) は、SUNWtoo パッケージにも組み込まれています。perfcnt と symbindrep はサンプルプログラムであり、実際の環境での使用を目的としていません。

監査インタフェースの制限

la_pltexit() 系列の使用にはいくつかの制限があります。これらの制限は、呼び出し側と呼び出し先の間で余分なスタックフレームを挿入して、la_pltexit() 戻り値を獲得する方法を提供するための必要から生じたものです。la_pltenter() ルーチンだけを呼び出す場合には、妨害となるスタックを整理してから宛先関数に制御を譲渡できるため、上記は問題になりません。

これらの制限が原因で、la_pltexit() は、上記の制限を改善するために今後のリリースで変更される可能性がある、実験的インタフェースとみなされます。問題がある場合には、la_pltexit() ルーチンを避けるようにしてください。

スタックを直接検査する関数

スタックを直接検査するか、またはその状態について仮定をたてる少数の関数があります。これらの関数の例としては、setjmp(3C) ファミリ、vfork(2)、および構造へのポインタではなく構造を返す関数があります。これらの関数は、la_pltexit() をサポートするために作成される余分なスタックによって調整されます。

実行時リンカーは、このタイプの関数を検出できないため、監査ライブラリの作成元が、このようなルーチンの la_pltexit() を無効にする必要があります。

実行時リンカーのデバッグインタフェース

第3章で説明したように、実行時リンカーは、メモリーへのオブジェクトの割り当てやシンボルの結合を含む多数の操作を実行します。デバッグプログラムは、通常、これらの実行時リンカーの操作をアプリケーション解析の一部として記述する情報にアクセスする必要があります。これらのデバッグプログラムは、解析対象のアプリケーションに対する独立したプロセスとして実行されます。

このセクションでは、他のプロセスから動的にリンクされたアプリケーションを監視、変更するためにサポートされているインタフェースを説明します。このインタフェースは、`rtld`-デバッガインタフェースと呼ばれます。このインタフェースのアーキテクチャは、`libthread_db(3THR)` で使用されるモデルに従っています。

`rtld`-デバッガインタフェースを使用する場合は、少なくとも次の2つのプロセスが関与します。

- 1つまたは複数のターゲットプロセス。ターゲットプロセスは動的にリンクし、実行時リンカー `/usr/lib/ld.so.1` (32ビットプロセスの場合)、または `/usr/lib/64/ld.so.1` (64ビットプロセスの場合) を使用する必要がある
- 制御プロセスは、`rtld`-デバッガのインタフェースライブラリとリンクし、それを使用してターゲットプロセスの動的側面を検査する。64ビット制御プロセスは、64ビットおよび32ビットの両方のターゲットをデバッグできる。ただし、32ビット制御プロセスは32ビットターゲットに制限される

`rtld`-デバッガは、制御プロセスがデバッガであり、そのターゲットが動的実行可能なプログラムの場合に、最もよく使用されます。

`rtld`-デバッガインタフェースは、ターゲットプロセスに対して次のものを有効にします。

- 実行時リンカーとの最初の認識
- 動的オブジェクトの読み込みと読み込み解除の通知
- 読み込まれたオブジェクトすべてに関する情報の検索
- プロシージャのリンクテーブルエントリのステップオーバー
- オブジェクトパッドの有効化

制御プロセスとターゲットプロセス間の対話

ターゲットプロセスを検査して操作できるようにするために、`rtld`-デバッガインタフェースは、エクスポートされたインタフェース、インポートされたインタフェース、およびエージェントを使用して、これらのインタフェース間で通信を行います。

制御プロセスは、`librtld_db.so.1` によって提供される `rtld`-デバッガインタフェースにリンクされて、このライブラリからエクスポートされたインタフェースを要求します。このインタフェースは、`/usr/include/rtld_db.h` に定義されています。次に、`librtld_db.so.1` は制御プロセスからインポートされたインタフェースを要求します。この対話によって、`rtld`-デバッガインタフェースは、次のことを行うことができます。

- ターゲットプロセス内のシンボルの検索
- ターゲットプロセスのメモリーの読み取りと書き込み

インポートされたインタフェースは多数の `proc_service` ルーチンから構成されています (195ページの「デバグインポートインタフェース」を参照)。ほとんどのデバグは、このルーチンをすでに使用してプロセスを解析しています。

`rtld`-デバグインタフェースは、`rtld`-デバグインタフェースの要求により解析中のプロセスが停止することを前提としています。停止しない場合は、ターゲットプロセスの実行時リンカー内にあるデータ構造が、検査時に一貫した状態にない可能性があります。

図 6-1 は、`librtld_db.so.1`、制御プロセス (デバグ)、およびターゲットプロセス (動的実行可能プログラム) 間の情報の流れを示しています。

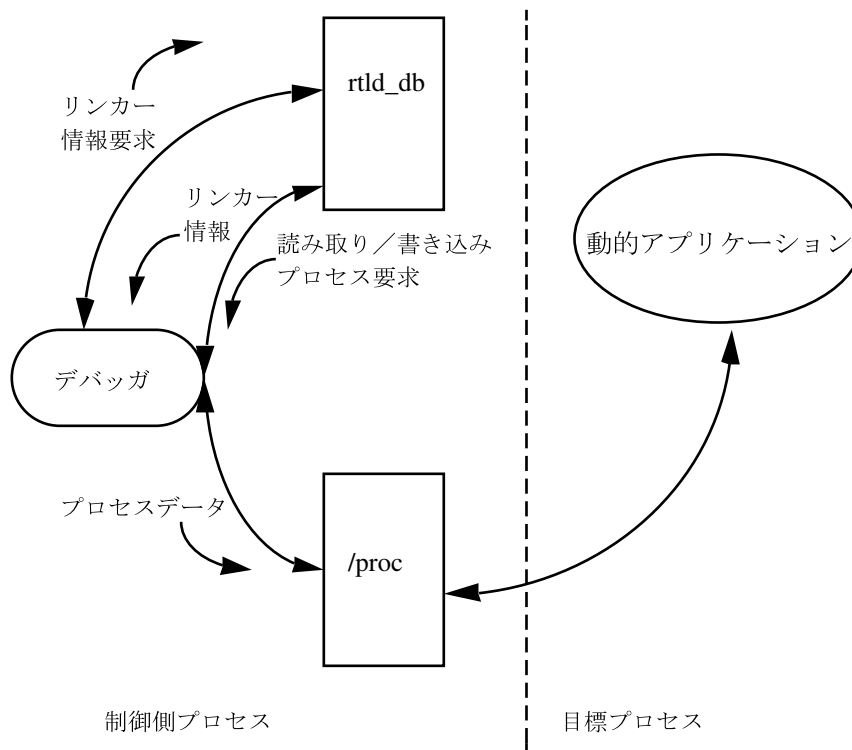


図 6-1 `rtld`-デバグの情報の流れ

注 - rtdl-デバッガインタフェースは、実験的と見なされる `proc_service` インタフェース (`/usr/include/proc_service.h`) に依存します。rtdl-デバッガインタフェースは、展開時に、`proc_service` インタフェース内の変更を追跡しなければならないことがあります。

rtdl-デバッガインタフェースを使用する制御プロセスのサンプル実装状態は、`/usr/demo/librtdl_db` の `SUNWosdem` パッケージに用意されています。このデバッガ `rdb` は、`proc_service` インポートインタフェースの使用例を提示して、すべての `librtdl_db.so.1` インポートインタフェースの使用例を提示して、すべての rtdl-デバッガインタフェースについて説明します。さらに詳しい情報は、サンプルデバッガをテストして得ることができます。

デバッガインタフェースのエージェント

エージェントは、内部インタフェース構造を記述できる不透明なハンドルを提供して、エクスポートインタフェースとインポートインタフェースの間の通信機構となるものです。rtdl-デバッガインタフェースは、いくつかのプロセスを同時に操作できるデバッガによる使用を目的としているため、これらのエージェントは、プロセスを特定するために使用されます。

`struct ps_prochandle`

```
struct ps_prochandle;
```

制御プロセスによって、エクスポートインタフェースとインポートインタフェースの間で渡されるターゲットプロセスを特定するために作成される不透明な構造です。

`struct rd_agent`

```
struct rd_agent;
```

rtdl-デバッガインタフェースによって、エクスポートインタフェースとインポートインタフェースの間で渡されるターゲットプロセスを特定するために作成される不透明な構造です。

デバッグエクスポートインタフェース

このセクションでは、`/usr/lib/librtld_db.so.1` 監査ライブラリによってエクスポートされる各インタフェースについて説明します。この節は、機能グループごとに分けられています。

エージェント操作

`rd_init()`

```
rd_err_e rd_init(int version);
```

この関数は、`rtld`-デバッグバージョン条件を確立します。ベースとなるバージョンは、`RD_VERSION1` として定義されています。現在の「バージョン」は常に `RD_VERSION` で定義されます。

Solaris 8 10/00 で追加されたバージョン `RD_VERSION2` は、`rd_loadobj_t` 構造体を拡張するものです (187ページの「読み込み可能オブジェクトの走査」の `rl_flags`、`rl_bend`、`rl_dynamic` の各フィールドを参照)。

Solaris 8 1/01 で追加されたバージョン `RD_VERSION3` は、`rd_plt_info_t` 構造体を拡張するものです (192ページの「プロシージャのリンクテーブルのスキップ」の `pi_baddr` および `pi_flags` の各フィールドを参照)。

制御プロセスのバージョン条件が使用可能な `rtld`-デバッグインタフェースよりも大きい場合は、`RD_NOCAPAB` が返されます。

`rd_new()`

```
rd_agent_t * rd_new(struct ps_prochandle * php);
```

この関数は、新しいエクスポートのインタフェースエージェントを作成します。「`php`」は、制御プロセスによってターゲットプロセスを特定するために作成された cookie です。この cookie は、制御プロセスによってコンテキストを維持するために提供される重要なインタフェースで使用されるものであり、`rtld`-デバッグインタフェースに対して不透明です。

`rd_reset()`

```
rd_err_e rd_reset(struct rd_agent * rdap);
```

この関数は、rd_new() に指定された同じ ps_prochandle 構造に基づくエージェント内の情報をリセットします。この関数は、ターゲットプロセスが再スタートされると呼び出されます。

rd_delete()

```
void rd_delete(struct rd_agent * rdap);
```

この関数は、エージェントを削除し、それに関連するすべての状態を解除します。

エラー処理

次のエラー状態は、rtld-デバッガインタフェース (rtld_db.h に定義) によって返されます。

```
typedef enum {
    RD_ERR,
    RD_OK,
    RD_NOCAPAB,
    RD_DBERR,
    RD_NOBASE,
    RD_NODYNAM,
    RD_NOMAPS
} rd_err_e;
```

次のインタフェースは、エラー情報を収集するために使用できます。

rd_errstr()

```
char * rd_errstr(rd_err_e rderr);
```

この関数は、エラーコード「rderr」を記述する記述エラー文字列を返します。

rd_log()

```
void rd_log(const int onoff);
```

この関数は、ログ記録をオン (1) またはオフ (0) にします。ログ記録がオンの場合、制御プロセスによって提供されるインポートインタフェース関数 ps_plog() は、さらに詳しい診断情報によって呼び出されます。

読み込み可能オブジェクトの走査

実行時リンカーのリンクマップで維持される各オブジェクト情報の取得は (170ページの「名前空間の確立」を参照)、`rtld_db.h` に定義された次の構造を使用して実現されます。

```
typedef struct rd_loadobj {
    psaddr_t      rl_nameaddr;
    unsigned      rl_flags;
    psaddr_t      rl_base;
    psaddr_t      rl_data_base;
    unsigned      rl_lmident;
    psaddr_t      rl_refnameaddr;
    psaddr_t      rl_plt_base;
    unsigned      rl_plt_size;
    psaddr_t      rl_bend;
    psaddr_t      rl_padstart;
    psaddr_t      rl_padend;
    psaddt_t      rl_dynamic;
} rd_loadobj_t;
```

文字列ポインタを含めて、この構造で指定されるアドレスはすべてターゲットプロセス内のアドレスであり、制御プロセス自体のアドレス空間のアドレスでないことに注意してください。

`rl_nameaddr`

動的オブジェクトの名前を含む文字列へのポインタ

`rl_flags`

リビジョン `RD_VERSION2` では、動的に読み込まれる再配置可能オブジェクトは `RD_FLG_MEM_OBJECT` で識別されます。

`rl_base`

動的オブジェクトの基本アドレス

`rl_data_base`

動的オブジェクトのデータセグメントの基本アドレス

`rl_lmident`

リンクマップ識別子 (170ページの「名前空間の確立」を参照)

rl_refnameaddr

動的オブジェクトがフィルタの場合は (115ページの「フィルタとしての共有オブジェクト」を参照)、フィルターの名前を指定する

rl_plt_base, rl_plt_size

これらの要素は、下方互換性のために存在するものであり、現在は使用されていない

rl_bend

オブジェクトのエンドアドレス (text + data + bss)。リビジョン RD_VERSION2 では、動的に読み込まれる再配置可能オブジェクトの場合、この要素は作成されたオブジェクトの最後を指します。このオブジェクトには、自身のセクションヘッダーが含まれています。

rl_padstart

動的オブジェクト前のパッドの基本アドレス (195ページの「動的オブジェクトのパッド」を参照)

rl_padend

動的オブジェクト後のパッドの基本アドレス (195ページの「動的オブジェクトのパッド」を参照)

rl_dynamic

このフィールドは RD_VERSION2 に追加されたもので、DT_CHECKSUM (表 7-43 を参照) のエントリへのリファレンスを可能にするオブジェクトの動的セクションのベースアドレスを提供します。

次のルーチンは、このオブジェクトデータ構造を使用して実行時リンカーリンクマップリストの情報にアクセスしています。

rd_loadobj_iter()

```
typedef int rl_iter_f(const rd_loadobj_t *, void *);  
rd_err_e rd_loadobj_iter(rd_agent_t * rap, rl_iter_f * cb,
```

(続く)

```
void * clnt_data);
```

この関数は、ターゲットプロセスに現在読み込まれている動的オブジェクトすべてを反復します。各反復時に、`cb` によって指定されたインポート関数が呼び出されます。`clnt_data` は、`cb` 呼び出しにデータを渡すために使用できます。各オブジェクトに関する情報は、スタックが割り当てられた `volatile rd_loadobj_t` 構造へのポインタを介して返されます。

`cb` ルーチンからの戻りコードは、`rd_loadobj_iter()` によってテストされ、次の意味を持ちます。

- 1 — リンクマップの処理を継続
- 0 — リンクマップの処理を停止して、制御プロセスに制御を返す

`rd_loadobj_iter()` は、正常だと `RD_OK` を返します。`RD_NOMAPS` が返される場合、実行時リンカーは、まだ初期リンクマップを読み込みません。

イベント通知

実行時リンカーの適用範囲内で発生し、制御プロセスが追跡できる特定のイベントがあります。これらのイベントは次のとおりです。

RD_PREINIT

実行時リンカーは、すべての動的オブジェクトを読み込んで再配置し、読み込まれた各オブジェクトの `.init` セクションの呼び出しを開始 (103ページの「デバッグングエイド」を参照)。

RD_POSTINIT

実行時リンカーは、すべての `.init` セクションの呼び出しを終了して、基本実行可能ファイルに制御を渡す。

RD_DLACTIVITY

実行時リンカーは、動的オブジェクトを読み込みまたは読み込み解除のために呼び出される (81ページの「追加オブジェクトの読み込み」を参照)。

これらのイベントは、次のインタフェース (sys/link.h と rtld_db.h に定義) を使用して監視できます。

```
typedef enum {
    RD_NONE = 0,
    RD_PREINIT,
    RD_POSTINIT,
    RD_DLACTIVITY
} rd_event_e;

/*
 * ways that the event notification can take place:
 */
typedef enum {
    RD_NOTIFY_BPT,
    RD_NOTIFY_AUTOBPT,
    RD_NOTIFY_SYSCALL
} rd_notify_e;

/*
 * information on ways that the event notification can take place:
 */
typedef struct rd_notify {
    rd_notify_e    type;
    union {
        psaddr_t    bptaddr;
        long        syscallno;
    } u;
} rd_notify_t;
```

rd_event_enable()

```
rd_err_e rd_event_enable(struct rd_agent * rdap, int onoff);
```

この関数は、イベント監視を有効 (1) または無効 (0) にします。

注・パフォーマンス上の理由から、現在、実行時リンカーはイベントの無効化を無視します。制御プロセスは、このルーチンへの最後の呼び出しが原因で指定のブレークポイントに到達しないと、想定することはできません。

rd_event_addr()

```
rd_err_e rd_event_addr(rd_agent_t * rdap, rd_event_e event,  
rd_notify_t * notify);
```

この関数は、制御プログラムへの指定イベントの通知方法を指定します。

イベントタイプに従って、制御プロセスの通知は、`notify->u.syscallno` で特定されるチープなシステム呼び出しを呼び出すか、または `notify->u.bptaddr` によって指定されたアドレスでブレークポイントを実行することで行われます。システム呼び出しの追跡または実際のブレークポイントの設定は、制御プロセスが行う必要があります。

イベントが発生した場合は、`rtld_db.h` に定義された次のインタフェースによって追加情報を取得できます。

```
typedef enum {  
    RD_NOSTATE = 0,  
    RD_CONSISTENT,  
    RD_ADD,  
    RD_DELETE  
} rd_state_e;  
  
typedef struct rd_event_msg {  
    rd_event_e      type;  
    union {  
        rd_state_e  state;  
    } u;  
} rd_event_msg_t;
```

`rd_state_e` 値の意味は次のとおりです。

`RD_NOSTATE`

使用可能な追加状態情報なし

`RD_CONSISTANT`

リンクマップは安定した状態にあってテスト可能

`RD_ADD`

動的オブジェクトは削除処理中であり、リンクマップは安定した状態にない。リンクマップは、`RD_CONSISTANT` 状態に達するまでテストできない

RD_DELETE

動的オブジェクトは削除処理中であり、リンクマップは安定した状態にない。リンクマップは、RD_CONSISTANT 状態に達するまでテストできない

rd_event_getmsg()

```
rd_err_e rd_event_getmsg(struct rd_agent * rdap, rd_event_msg_t * msg);
```

この関数は、イベントに関する追加情報を提供します。

次の表は、異なる各イベントタイプで可能な状態を示しています。

RD_PREINIT	RD_POSTINIT	RD_DLACTIVITY
RD_NOSTATE	RD_NOSTATE	RD_CONSISTANT
		RD_ADD
		RD_DELETE

プロシージャのリンクテーブルのスキップ

rtld-デバッグインタフェースは、プロシージャのリンクのテーブルエントリをスキップオーバーするための機能を提供します (317ページの「プロシージャのリンクテーブル (プロセッサ固有)」を参照)。デバッガなどの制御プロセスは、初めて関数に介入するよう要求される場合、実際のプロシージャのリンクテーブル処理をスキップしようとします。この結果、制御は、関数定義を検索するために実行時リンカーに渡されます。

次のインタフェースを使用すると、制御プロセスで実行時リンカーのプロシージャのリンクテーブル処理にステップオーバーできます。制御プロセスは、ELF ファイルで提供される外部情報に基づいて、プロシージャのリンクのテーブルエントリに遭遇する時期を判定できるものと想定されます。

ターゲットプロセスは、プロシージャのリンクのテーブルエントリに介入すると、次のインタフェースを呼び出します。

rd_plt_resolution()


```
rd_err_e rd_plt_resolution(rd_agent_t * rdap, paddr_t pc,
                          lwpid_t lwpid, paddr_t plt_base, rd_plt_info_t * rpi);
```

この関数は、現在のプロシージャのリンクテーブルエントリの解決状態と、それをスキップする方法に関する情報を返します。

`pc` は、プロシージャのリンクテーブルエントリの最初の命令を表します。`lwpid` は `lwp` 識別子を提供し、`plt_base` はプロシージャのリンクテーブルの基本アドレスを提供します。これらの3つの変数は、各種のアーキテクチャがプロシージャのリンクテーブルを処理するため十分な情報を提供します。

`rpi` は、次のデータ構造 (`rtld_db.h` に定義) に定義された、プロシージャのリンクのテーブルエントリに関する詳しい情報を提供します。

```
typedef enum {
    RD_RESOLVE_NONE,
    RD_RESOLVE_STEP,
    RD_RESOLVE_TARGET,
    RD_RESOLVE_TARGET_STEP
} rd_skip_e;

typedef struct rd_plt_info {
    rd_skip_e      pi_skip_method;
    long          pi_nstep;
    psaddr_t      pi_target;
    psaddr_t      pi_baddr;
    unsigned int  pi_flags;
} rd_plt_info_t;

#define RD_FLG_PI_PLTBOUND    0x0001
```

`pi_skip_method`

プロシージャのリンクテーブルエントリがどのように扱われるかを示す。`rd_skip_e` 値内の1つに設定される

`pi_nstep`

`RD_RESOLVE_STEP` または `RD_RESOLVE_TARGET_STEP` が返された時にステップオーバーする命令がいくつあるかを示す

`pi_target`

`RD_RESOLVE_TARGET_STEP` または `RD_RESOLVE_TARGET` が返された時にブレークポイントを設定するアドレス指定する

pi_baddr

RD_VERSION3 で追加された、プロシージャのリンクテーブルの宛先アドレス。pi_flags フィールドの RD_FLG_PI_PLTBOUND フラグが設定されると、この要素は解決された (結合された) 宛先アドレスを示す

pi_flags

RD_VERSION3 で追加されたフラグフィールド。フラグ RD_FLG_PI_PLTBOUND は、pi_baddr フィールドで取得できる宛先アドレスへ解決された (結合された) プロシージャのリンクエントリを示す

次のシナリオは rd_plt_info_t 戻り値から考えられます。

- これはこのプロシージャのリンクテーブルで最初の呼び出しであるため、実行時リンカーによって解決する必要がある。rd_plt_info_t には、次のものが含まれる

```
{RD_RESOLVE_TARGET_STEP, M, <BREAK>, 0, 0}
```

制御プロセスは、BREAK にブレークポイントを設定し、ターゲットプロセスを続けます。ブレークポイントに達すると、プロシージャのリンクのテーブルエントリ処理は終了し、制御プロセスは M 命令を宛先関数にステップできます。これはプロシージャのリンクテーブルエントリで最初の呼び出しであるため、結合アドレス (pi_baddr) が設定されていないことに注意してください。

- このプロシージャのリンクテーブル全体で N 番目。rd_plt_info_t には、次のものが含まれる

```
{RD_RESOLVE_STEP, M, 0, <BoundAddr>, RD_FLG_PI_PLTBOUND}
```

プロシージャのリンクのテーブルエントリはすでに解決されていて、制御プロセスは M 命令を宛先関数にステップできます。プロシージャのリンクテーブルエントリが結合されるアドレスは <BoundAddr> で、RD_FLG_PI_PLTBOUND ビットが flags フィールドに設定されています。

注 - 今後の実装状態では、ターゲット関数にブレークポイントを直接設定する方法として、RD_RESOLVE_TARGET を使用する可能性がありますが、この機能は、今回のバージョンの rtdl-デバッグインタフェースではまだ使用できません。

動的オブジェクトのパッド

実行時リンカーのデフォルト動作は、オペレーティングシステムに依存して、最も効率的に参照できる場所に動的オブジェクトを読み込みます。制御プロセスの中には、ターゲットプロセスのメモリーに読み込まれたオブジェクトの周りにパッドがあることによって、利益を受けるものがあります。このインタフェースを使用すると、制御プロセスは、このパッドを要求できます。

`rd_objpad_enable()`

```
rd_err_e rd_objpad_enable(struct rd_agent * rdap, size_t padsize);
```

この関数は、ターゲットプロセスによって続けて読み込まれたオブジェクトのパッドを有効または無効にします。パッドは読み込まれたオブジェクトの両側で行われます。

`padsize` は、メモリーに読み込まれたオブジェクトの前後両方で維持されるパッドのサイズをバイト数で指定します。このパッドは、`mmap(2)` に `PROT_NONE` 権と `MAP_NORESERVE` フラグをつけて使用して、メモリー割り当てとして予約できます。実行時リンカーは、読み込まれたオブジェクトに隣接するターゲットプロセスの仮想アドレス空間の領域を効果的に予約します。これらの領域は、制御プロセスによって後に利用できます。

`padsize` を 0 にすると、後のオブジェクトに対するオブジェクトパッドは無効になります。

注 - `mmap(2)` を `/dev/zero` から、`MAP_NORESERVE` によって使用して取得される予約は、`proc(1)` 機能を使用して、`rd_loadobj_t` に提供されたリンクマップ情報を参照することによって報告できます。

デバッグインポートインタフェース

制御プロセスが `librtld_db.so.1` に対して提供しなければならないインポートインタフェースは、`/usr/include/proc_service.h` に定義されています。これらの `proc_service` 関数のサンプル実装状態は、`rd` デモデバッガにあります。`rtld`-デバッグインタフェースは、使用可能な `proc_service` インタフェースのサブセットだけを使用します。`rtld`-デバッグインタフェースの今後のバージョンでは、互換性のない変更を作成することなく、追加 `proc_service` インタフェースを利用できる可能性があります。

次のインタフェースは、現在、`rtld`-デバッガインタフェースによって使用されています。

`ps_pauxv()`

```
ps_err_e ps_pauxv(const struct ps_prochandle * ph, auxv_t ** aux);
```

この関数は、`auxv` ベクトルのコピーへのポインタを返します。`auxv` ベクトル情報は、割り当てられた構造にコピーされるため、このポインタの存続期間は、`ps_prochandle` が有効な間になります。

`ps_pread()`

```
ps_err_e ps_pread(const struct ps_prochandle * ph, paddr_t addr,
                  char * buf, int size);
```

この関数は、アドレス `addr` にあるターゲットプロセスからサイズバイトを読み取って、それらを `buf` にコピーします。

`ps_pwrite()`

```
ps_err_e ps_pwrite(const struct ps_prochandle * ph, paddr_t addr,
                   char * buf, int size);
```

この関数は、サイズバイトを `buf` から、アドレス `addr` にあるターゲットプロセスに書き込みます。

`ps_plog()`

```
void ps_plog(const char * fmt, ...);
```

この関数は、`rtld`-デバッガインタフェースから追加診断情報によって呼び出されます。この診断情報をどこに記録するか、または記録するかどうかは、制御プロセスが決める必要があります。`ps_plog()` の引数は、`printf(3C)` 形式に従っています。

`ps_pglobal_lookup()`

```
ps_err_e ps_pglobal_lookup(const struct ps_prochandle * ph,
                           const char * obj, const char * name, ulong_t * sym_addr);
```

この関数は、ターゲットプロセス `ph` 内のオブジェクト `obj` 内の記号 `name` を検索します。記号が検出されると、記号のアドレスが `sym_addr` に保存されます。

`ps_pglobal_sym()`

```
ps_err_e ps_pglobal_sym(const struct ps_prochandle * ph,
                        const char * obj, const char * name, ps_sym_t * sym);
```

この関数は、ターゲットプロセス `ph` 内のオブジェクト `obj` 内の記号 `name` を検索します。記号が検出されると、記述子 `sym` は埋められます。

`rtld`-デバッガインタフェースがアプリケーションまたは実行時リンカー内の記号を検出してから、リンクマップを作成する必要があるイベントでは、`obj` に対する次の予約値を使用できます。

```
#define PS_OBJ_EXEC ((const char *)0x0) /* application id */
#define PS_OBJ_LDSD ((const char *)0x1) /* runtime linker id */
```

制御プロセスがこれらのオブジェクトの記号テーブルを検出するために使用できる機構の1つに、次の擬似コードを使用する `procfs` ファイルシステムを介するものがあります。

```
ioctl(..., PIOCNAUXV, ...)      - obtain AUX vectors
ldsoaddr = auxv[AT_BASE];
ldsofd = ioctl(..., PIOCOPENM, &ldsoaddr);

/* process elf information found in ldsofd ... */

execfd = ioctl(..., PIOCOPENM, 0);

/* process elf information found in execfd ... */
```

ファイル記述子が見つかったら、ELF ファイルは、制御プログラムによってその記号情報をテストできます。

オブジェクトファイル

概要

この章では、アセンブラとリンカーで生成されるオブジェクトファイルの実行可能リンク形式 (ELF) について説明します。オブジェクトファイルには、主に次の3つの種類が存在します。

- 再配置可能ファイルは、他のオブジェクトファイルとリンクして実行可能ファイル、共有オブジェクトファイル、または別の再配置可能ファイルを作成するのに適したコードとデータを保持する
- 実行可能ファイルは、実行可能なプログラムを保持する。実行可能ファイルは、`exec(2)` によるプログラムのプロセスイメージの作成方法を指定する
- 共有オブジェクトファイルは、次の2つのリンクに適したコードとデータを保持する。(1) リンカーは、共有オブジェクトファイルを他の再配置可能ファイルや共有オブジェクトファイルと共に処理して、別のオブジェクトファイルを作ることができる。(2) 実行時リンカーは、共有オブジェクトファイルを動的実行可能ファイルや他の共有オブジェクトファイルと組み合わせ、プロセスイメージを作成する

200ページの「ファイル形式」では、オブジェクトファイルの形式、およびこの形式がプログラム作成にどのように関係しているかに焦点を当てています。279ページの「動的リンク」では、この形式がプログラムの読み込みにどのように関係しているかに焦点を当てています。

オブジェクトファイルは、ELF アクセスライブラリ `libelf` に含まれる関数で処理できます。`libelf` の説明については、`elf(3ELF)` を参照してください。`libelf` を使用するサンプルソースコードは、`SUNWosdem` パッケージに含まれており、`/usr/demo/ELF` ディレクトリの下に置かれています。

ファイル形式

すでに述べたとおり、オブジェクトファイルはプログラムのリンクと実行の両方に関係します。利便性と効率性のため、オブジェクトファイルの形式には、リンクと実行の異なる要求に合わせて、2つの平行した見方があります。図 7-1 にオブジェクトファイルの編成を示します。

リンク	実行
ELF ヘッダー	ELF ヘッダー
プログラムヘッダー テーブル (オプション)	プログラムヘッダー テーブル
セクション 1	セグメント 1
...	
セクション n	セグメント 2
...	
...	...
セクションヘッダー テーブル	セクションヘッダー テーブル (オプション)

図 7-1 オブジェクトファイル形式

ELF ヘッダーはオブジェクトファイルの先頭に存在し、ファイル編成を記述する「ロードマップ」を保持します。

「セクション」は、ELF ファイル内で処理可能な最小単位 (これ以上分割できない単位) です。「セグメント」は、`exec(2)` または実行時リンカーでメモリーイメージに対応付けできる (`mmap(2)` のマニュアルページを参照) 最小単位 (これ以上分割できない単位) です。

セクションは、リンクの観点から見たオブジェクトファイルの情報(命令、データ、シンボルテーブル、再配置情報など)の大部分を保持します。セクションに関しては、この章の前半で説明します。セグメントとプログラムの実行の観点から見たファイルの構造に関しては、この章の後半で説明します。

プログラムヘッダーテーブル(存在する場合)は、システムにプロセスイメージの作成方法を通知します。プロセスイメージの作成に使用されるファイル(実行可能ファイルと共有オブジェクト)には、プログラムヘッダーテーブルが存在しなければなりません。再配置可能オブジェクトには、プログラムヘッダーテーブルは存在する必要はありません。

セクションヘッダーテーブルには、ファイルのセクションを記述する情報が入っています。セクションヘッダーテーブルには各セクションのエントリが存在します。各エントリは、セクション名、セクションサイズなどの情報が含まれます。リンク編集で使用されるファイルには、セクションヘッダーテーブルが存在しなければなりません。他のオブジェクトファイルには、セクションヘッダーテーブルは存在してもしなくてもかまいません。

注・図 7-1 では ELF ヘッダの直後にプログラムヘッダーテーブルが示され、セクションヘッダーテーブルがセクションの後に続いていますが、実際のファイルは異なる場合があります。また、セクションとセグメントの順序は特に決まっています。ELF ヘッダーの位置のみがファイル内で固定されています。

データ表現

ここで記述されているとおり、オブジェクトファイルの形式は、8 ビットバイト、32 ビットアーキテクチャ、および 64 ビットアーキテクチャを持つさまざまなプロセッサをサポートしていますが、オブジェクトファイルの形式は、より大きな(またはより小さな)アーキテクチャに拡張できることを意図しています。

したがって、オブジェクトファイルはマシンに依存しない形式になっているいくつかの制御データを表現し、その結果、オブジェクトファイルが識別でき、オブジェクトファイルの内容が共通した方法で解釈できます。オブジェクトファイルの残りのデータは、このオブジェクトファイルが作成されたマシンとは関係なく、対象となるプロセッサ用に符号化されています。

表 7-1 32 ビットデータタイプ

名前	サイズ	整列	目的
Elf32_Addr	4	4	符号なしプログラムアドレス
Elf32_Half	2	2	符号なし、中程度の整数
Elf32_Off	4	4	符号なしファイルオフセット
Elf32_Sword	4	4	符号付き整数
Elf32_Word	4	4	符号なし整数
unsigned char	1	1	符号なし、短い整数

表 7-2 64 ビットデータタイプ

名前	サイズ	整列	目的
Elf64_Addr	8	8	符号なしプログラムアドレス
Elf64_Half	2	2	符号なし、中程度の整数
Elf64_Off	8	8	符号なしファイルオフセット
Elf64_Sword	4	4	符号付き整数
Elf64_Word	4	4	符号なし整数
Elf64_Xword	8	8	符号なし、長い整数
Elf64_Sxword	8	8	符号付き、長い整数
unsigned char	1	1	符号なし、短い整数

オブジェクトファイルの形式で定義されるすべてのデータ構造は、該当クラスの自然なサイズと整列ガイドラインに従います。必要であれば、データ構造に明示的にパッドを入れることで、4 バイトオブジェクトに対して 4 バイト整列を保証したり

構造サイズを4の倍数に設定したりします。また、データはファイルの先頭から適切に整列されます。したがってたとえば、Elf32_Addr 構成要素が存在する構造はファイル内において4バイト境界で整列され、Elf64_Addr 構成要素が存在する構造は8バイト境界で整列されます。

注 - 移植性を考慮して、ELF ではビットフィールドを使用していません。

ELF ヘッダー

いくつかのオブジェクトファイル制御構造は大きくなる場合がありますが、そのサイズは ELF ヘッダーに記録されます。オブジェクトファイルの形式が変わった場合、ELF 形式のファイルにアクセスするプログラムは、大きくなったり小さくなったりした制御構造体を扱うことになります。大きくなった場合は、追加された部分を無視することができるかもしれませんが。小さくなった場合は、無くなった部分の扱いは状況に依存しますし、形式が変更された時に規定されるでしょう。

ELF ヘッダーの構造体 (sys/elf.h で定義されている) は、以下のとおりです。

```
#define EI_NIDENT      16

typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf32_Half      e_type;
    Elf32_Half      e_machine;
    Elf32_Word      e_version;
    Elf32_Addr      e_entry;
    Elf32_Off       e_phoff;
    Elf32_Off       e_shoff;
    Elf32_Word      e_flags;
    Elf32_Half      e_ehsize;
    Elf32_Half      e_phentsize;
    Elf32_Half      e_phnum;
    Elf32_Half      e_shentsize;
    Elf32_Half      e_shnum;
    Elf32_Half      e_shstrndx;
} Elf32_Ehdr;

typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf64_Half      e_type;
    Elf64_Half      e_machine;
    Elf64_Word      e_version;
    Elf64_Addr      e_entry;
    Elf64_Off       e_phoff;
    Elf64_Off       e_shoff;
    Elf64_Word      e_flags;
    Elf64_Half      e_ehsize;
```

(続く)

```

Elf64_Half e_phentsize;
Elf64_Half e_phnum;
Elf64_Half e_shentsize;
Elf64_Half e_shnum;
Elf64_Half e_shstrndx;
} Elf64_Ehdr;

```

e_ident

先頭のバイト列に、オブジェクトファイルであることを示す印と、機種に依存しない、ファイルの内容を復号化または解釈するためのデータが入ります。完全な記述は、208ページの「ELF 識別」で行われています。

e_type

この構成要素は、オブジェクトファイルの種類を示します。

表 7-3 ELF ファイル識別子

名前	値	意味
ET_NONE	0	ファイルタイプが存在しない
ET_REL	1	再配置可能ファイル
ET_EXEC	2	実行可能ファイル
ET_DYN	3	共有オブジェクトファイル
ET_CORE	4	コアファイル
ET_LOPROC	0xff00	プロセッサに固有
ET_HIPROC	0xffff	プロセッサに固有

コアファイルの内容は指定されていませんが、ET_CORE タイプはコアファイルを示すために予約されます。ET_LOPROC から ET_HIPROC までの値 (それぞれを含む)

は、プロセッサ固有の方法で解釈されます。他の値は予約され、必要に応じて新しいオブジェクトファイルの種類に割り当てられます。

e_machine

この構成要素の値は、個々のファイルが必要とするアーキテクチャを指定します。

表 7-4 ELF 機種

名前	値	意味
EM_NONE	0	マシンが存在しない
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_SPARC32PLUS	18	Sun SPARC 32+
EM_SPARCV9	43	SPARC V9

他の値は予約され、必要に応じて新しい機種に割り当てられます (`sys/elf.h` を参照)。プロセッサ固有の ELF 名の識別には、機種名が使用されます。たとえば、以下に述べるフラグでは接頭辞 `EF_` が使用されます。EM_XYZ マシンの WIDGET というフラグは、`EF_XYZ_WIDGET` と呼ばれます。

e_version

この構成要素は、オブジェクトファイルのバージョンを示します。

表 7-5 ELF バージョン

名前	値	意味
EV_NONE	0	無効なバージョン
EV_CURRENT	>=1	現在のバージョン

値 1 は最初のファイル形式を示し、拡張した場合は番号を大きくします。EV_CURRENT の値は、現在のバージョン番号を示すために必要に応じて変化します。

e_entry

この構成要素は、システムが制御を最初に渡す仮想アドレスを保持します。仮想アドレスが与えられると、プロセスが起動されます。ファイルに関連するエントリポイントが存在しない場合、この構成要素は 0 を保持します。

e_phoff

この構成要素は、プログラムヘッダーテーブルのファイルオフセット (単位: バイト) を保持します。ファイルにプログラムヘッダーテーブルが存在しない場合、この構成要素は 0 を保持します。

e_shoff

この構成要素は、セクションヘッダーテーブルのファイルオフセット (単位: バイト) を保持します。ファイルにセクションヘッダーテーブルが存在しない場合、この構成要素は 0 を保持します。

e_flags

この構成要素は、ファイルに対応付けられているプロセッサ固有のフラグを保持します。フラグ名は、EF_machine 「_flag」という形式をとります。この構成要素は現在、SPARC と IA に対して 0 です。

表 7-6 SPARC: ELF フラグ

名前	値	意味
EF_SPARCV9_MM	0x3	メモリーモデルのマスク
EF_SPARCV9_TSO	0x0	Total Store Ordering
EF_SPARCV9_PSO	0x1	Partial Store Ordering
EF_SPARCV9_RMO	0x2	Relaxed Memory Ordering

表 7-6 SPARC: ELF フラグ 続く

名前	値	意味
EF_SPARC_EXT_MASK	0xffff00	ベンダー拡張マスク
EF_SPARC_SUN_US1	0x000200	Sun UltraSPARC™ 1 拡張
EF_SPARC_HAL_R1	0x000400	HAL R1 拡張
EF_SPARC_SUN_US3	0x000800	Sun UltraSPARC 3 拡張

e_ehsize

この構成要素は、ELF ヘッダーのサイズ (単位: バイト) を保持します。

e_phentsize

この構成要素は、ファイルのプログラムヘッダーテーブルの 1 つのエントリのサイズ (単位: バイト) を保持します。すべてのエントリは同じサイズです。

e_phnum

この構成要素は、プログラムヘッダーテーブルのエントリ数を保持します。したがって、e_phentsize に e_phnum を掛けると、テーブルのサイズ (単位: バイト) が求められます。ファイルにプログラムヘッダーテーブルが存在しない場合、e_phnum は値 0 を保持します。

e_shentsize

この構成要素は、1 つのセクションヘッダーのサイズ (単位: バイト) を保持します。1 つのセクションヘッダーは、セクションヘッダーテーブルの 1 つのエントリです。すべてのエントリは同じサイズです。

e_shnum

この構成要素は、セクションヘッダーテーブルのエントリ数を保持します。したがって、e_shentsize に e_shnum を掛けると、セクションヘッダーテーブルの

サイズ(単位:バイト)が求められます。ファイルにセクションヘッダーテーブルが存在しない場合、`e_shnum` は値 0 を保持します。

`e_shstrndx`

この構成要素は、セクション名文字列テーブルに対応するエントリのセクションヘッダーテーブルインデックスを保持します。ファイルにセクション名文字列テーブルが存在しない場合、この構成要素は値 `SHN_UNDEF` を保持します。詳細は、212ページの「セクション」と 235ページの「文字列テーブル」を参照してください。

ELF 識別

先に述べたとおり、ELF はオブジェクトファイルの枠組みを提供し、複数のプロセッサ、複数のデータ符号化、複数のクラスのマシンをサポートします。このオブジェクトファイルファミリをサポートできるようにファイルの初期バイトは、問い合わせが行われるプロセッサに関係なくかつファイルの残りの内容にも関係なくファイルのセマンティクスを指定します。

ELF ヘッダー (とオブジェクトファイル) の初期バイトは、`e_ident` 構成要素に一致します。

表 7-7 `e_ident[]` 識別インデックス

名前	値	目的
<code>EI_MAG0</code>	0	ファイルの識別
<code>EI_MAG1</code>	1	ファイルの識別
<code>EI_MAG2</code>	2	ファイルの識別
<code>EI_MAG3</code>	3	ファイルの識別
<code>EI_CLASS</code>	4	ファイルのクラス
<code>EI_DATA</code>	5	データの符号化
<code>EI_VERSION</code>	6	ファイルのバージョン
<code>EI_OSABI</code>	7	オペレーティングシステム / ABI の識別

表 7-7 e_ident[] 識別インデックス 続く

名前	値	目的
EI_ABIVERSION	8	ABI のバージョン
EI_PAD	7	パッドバイトの開始
EI_NIDENT	16	e_ident[] のサイズ

次のインデックスは、マジックナンバーを保持するバイトをアクセスします。

EI_MAG0 - EI_MAG3

ファイルの先頭 4 バイトは、ファイルを ELF オブジェクトファイルとして識別する「マジックナンバー」を保持します。

表 7-8 マジックナンバー

名前	値	位置
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

EI_CLASS

その次のバイト e_ident[EI_CLASS] は、ファイルのクラスまたは容量を示します。

表 7-9 ファイルクラス

名前	値	意味
ELFCLASSNONE	0	無効なクラス
ELFCLASS32	1	32 ビットオブジェクト
ELFCLASS64	2	64 ビットオブジェクト

ファイル形式は、最大マシンのサイズを最小マシンに押しつけることなしにさまざまなサイズのマシン間で互換性が維持されるように設計されています。ファイルのクラスは、オブジェクトファイルそのもののデータ構造によって使用される基本タイプを定義します。

クラス ELFCLASS32 は、4 ギガバイトまでのファイルと仮想アドレス空間が存在するマシンをサポートします。これは、表 7-1 で定義される基本タイプを使用します。

クラス ELFCLASS64 は、SPARC などの 64 ビットアーキテクチャ用に予約されています。これは、表 7-2 で定義される基本タイプを使用します。

EI_DATA

バイト `e_ident[EI_DATA]` は、オブジェクトファイルのプロセッサ固有のデータの符号化を指定します。現在、以下の符号化が定義されています。

表 7-10 データの符号化

名前	値	意味
ELFDATANONE	0	無効な符号化
ELFDATA2LSB	1	図 7-2 を参照
ELFDATA2MSB	2	図 7-3 を参照

これらの符号化の詳細を以下に示します。他の値は予約され、必要に応じて新しい符号化に割り当てられます。

EI_VERSION

バイト `e_ident[EI_VERSION]` は、ELF ヘッダーバージョン番号を指定します。現在この値は、`e_version` の表 7-5 で説明しているように、`EV_CURRENT` でなければなりません。

EI_OSABI

バイト `e_ident[EI_OSABI]` は、オブジェクトのターゲット先となるオペレーティングシステムおよび ABI を識別します。他の ELF 構造体内のフィールドの中には、オペレーティングシステム特有か ABI 特有の意味、あるいはその両方に特有の意味を持つフラグおよび値を持つものがあります。これらのフィールドの解釈は、このバイトの値によって決定されます。

EI_ABIVERSION

バイト `e_ident[EI_ABIVERSION]` は、オブジェクトのターゲット先となる ABI のバージョンを識別します。このフィールドは、ABI の互換性の無いバージョンを識別するために使用します。このバージョン番号の解釈は、`EI_OSABI` フィールドで識別される ABI によって異なります。プロセッサについて `EI_OSABI` フィールドに値が何も指定されていない場合、または `EI_OSABI` バイトの特定の値によって決定される ABI についてバージョンの値が何も指定されていない場合は、「指定無し」を示すものとして値 0 が使用されます。

EI_PAD

この値は、`e_ident` の使用されていないバイトの先頭を示します。これらのバイトは保留され、0 に設定されます。オブジェクトファイルを読み取るプログラムは、これらのバイトを無視するべきです。使用されていないバイト列が使用されるようになった場合、`EI_PAD` の値は変更されます。

ファイルのデータ符号化方式は、ファイル内の基本オブジェクトを解釈する方法を指定します。クラス `ELFCLASS32` のファイルは、1、2、および 4 バイトを占めるオブジェクトを使用します。クラス `ELFCLASS64` のファイルは、1、2、4、および 8 バイトを占めるオブジェクトを使用します。定義されている符号化方式の下では、オブジェクトは以下のように表されます。バイト番号は、左上隅に示されています。

`ELFDATA2LSB` を符号化すると、最下位バイトが最低位アドレスを占める 2 の補数値が指定されます。

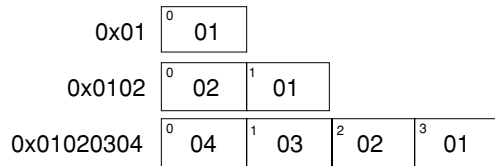


図 7-2 データの符号化方法 ELFDATA2LSB

ELFDATA2MSB を符号化すると、最上位バイトが最低位アドレスを占める 2 の補数値が指定されます。

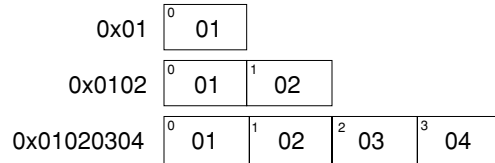


図 7-3 データの符号化方法 ELFDATA2MSB

セクション

オブジェクトファイルのセクションヘッダーテーブルを使用すると、すべてのファイルのセクションを見つけ出すことができます。セクションヘッダーテーブルは、以下に示されているとおり、Elf32_Shdr 構造体または Elf64_Shdr 構造体の配列です。セクションヘッダーテーブルインデックスは、この配列への添字です。ELF ヘッダーの e_shoff 構成要素は、ファイルの先頭からセクションヘッダーテーブルまでのバイトオフセットを与えます。e_shnum は、セクションヘッダーテーブルに存在するエントリ数を与えます。e_shentsize は、各エントリのサイズ (単位: バイト) を与えます。

いくつかのセクションヘッダーテーブルインデックスは予約されます。オブジェクトファイルには、これらの特殊インデックスのセクションは存在しません。

表 7-11 セクションの特殊インデックス

名前	値
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00

表 7-11 セクションの特殊インデックス 続く

名前	値
SHN_BEFORE	0xff00
SHN_AFTER	0xff01
SHN_HIPROC	0xff1f
SHN_ABS	0xffff1
SHN_COMMON	0xffff2
SHN_HIRESERVE	0xfffff

SHN_UNDEF

この値は、未定義の、失われた、関連のない、または無意味なセクション参照を示します。たとえば、セクション番号 SHN_UNDEF に関して「定義された」シンボルは、未定義シンボルです。

注 - インデックス 0 は未定義値として予約されますが、セクションヘッダーテーブルにはインデックス 0 のエントリが存在します。つまり、ELF ヘッダーの `e_shnum` 構成要素が、ファイルのセクションヘッダーテーブルに 6 つのエントリが存在することを示している場合、これら 6 つのエントリにはインデックス 0 から 5 ままでが与えられます。先頭のエントリの内容は、この項の末尾に記述します。

SHN_LORESERVE

この値は、予約されているインデックスの範囲の下限を指定します。

SHN_LOPROC - SHN_HIPROC

この範囲の値は、プロセッサ固有の使用方法に予約されます。

SHN_BEFORE, SHN_AFTER

これらの値は、SHF_LINK_ORDER および SHF_ORDERED セクションフラグと共に先頭および末尾セクションに順序付けを行います (表 7-14 を参照)。

SHN_ABS

この値は、対応する参照の絶対値を示します。たとえば、セクション番号 SHN_ABS からの相対で定義されたシンボルは絶対値をとり、再配置の影響を受けません。

SHN_COMMON

このセクションに関して定義されたシンボルは、共通シンボルです。たとえば、FORTRAN COMMON や割り当てられていない C 外部変数です。これらのシンボルは、ときどき一時的シンボルと呼ばれることもあります。

SHN_HIRESERVE

この値は、予約されているインデックスの範囲の上限を指定します。システムは、SHN_LORESERVE から SHN_HIRESERVE までのインデックスを予約します。値は、セクションヘッダーテーブルを参照しません。つまり、セクションヘッダーテーブルには予約されているインデックスのエントリは存在しません。

セクションには、ELF ヘッダー、プログラムヘッダーテーブル、セクションヘッダーテーブルを除く、オブジェクトファイルのすべての情報が存在します。また、オブジェクトファイルのセクションは以下の条件を満たします。

- オブジェクトファイルの各セクションには、そのセクションを記述するセクションヘッダーが必ず 1 つ存在する。対応するセクションが存在しないセクションヘッダーが存在することもある
- 各セクションは、ファイル内で連続するバイトシーケンス (空の場合もある) を占める
- ファイル内のセクション同士は重ならない。ファイル内のどのバイトも複数のセクションに属することはない
- オブジェクトファイルには、使用されていない領域が存在することがある。さまざまなヘッダーとセクションは、オブジェクトファイルのすべてのバイトをカバーしないことがある。使用されていないデータの内容は不定

セクションヘッダーの構造体 (sys/elf.h で定義されている) は、次のとおりです。

```
typedef struct {
    Elf32_Word    sh_name;
    Elf32_Word    sh_type;
    Elf32_Word    sh_flags;
    Elf32_Addr    sh_addr;
```

(続く)

```

        Elf32_Off      sh_offset;
        Elf32_Word     sh_size;
        Elf32_Word     sh_link;
        Elf32_Word     sh_info;
        Elf32_Word     sh_addralign;
        Elf32_Word     sh_entsize;
    } Elf32_Shdr;

typedef struct {
        Elf64_Word     sh_name;
        Elf64_Word     sh_type;
        Elf64_Xword    sh_flags;
        Elf64_Addr     sh_addr;
        Elf64_Off      sh_offset;
        Elf64_Xword    sh_size;
        Elf64_Word     sh_link;
        Elf64_Word     sh_info;
        Elf64_Xword    sh_addralign;
        Elf64_Xword    sh_entsize;
    } Elf64_Shdr;

```

sh_name

この構成要素は、セクション名を指定します。値はセクションヘッダーの文字列テーブルセクション (235ページの「文字列テーブル」を参照) へのインデックスで、空文字で終わっている文字列を示します。セクション名とその説明は、表 7-17 を参照してください。

sh_type

この構成要素は、セクションの内容と意味を分類します。セクションの種類とその説明は、表 7-12 を参照してください。

sh_flags

セクションは、さまざまな属性を記述する 1 ビットフラグをサポートします。フラグの定義は、表 7-14 を参照してください。

sh_addr

セクションがプロセスのメモリーイメージに現れる場合、この構成要素はセクションの先頭バイトが存在しなければならないアドレスを与えます。セクションがプロセスのメモリーイメージに現れない場合、この構成要素には 0 が存在します。

sh_offset

この構成要素は、ファイルの先頭からセクションの先頭バイトまでのバイトオフセットを与えます。以下に説明されている SHT_NOBITS 型のセクションは、ファイルのスペースを占めません。sh_offset 構成要素は、ファイル内の概念上の位置を示します。

sh_size

この構成要素は、セクションのサイズ (単位: バイト) を与えます。セクションの型が SHT_NOBITS でない限り、セクションはファイルの sh_size バイトを占めます。タイプが SHT_NOBITS のセクションは、0 以外のサイズをとることがありますが、ファイルのスペースは占めません。

sh_link

この構成要素は、セクションヘッダーテーブルインデックスリンクを保持します。このリンクの解釈は、セクションの型に依存します。値は、表 7-15 を参照してください。

sh_info

この構成要素は、追加的な情報を保持します。追加的な情報の解釈は、セクションの型に依存します。値は、表 7-15 を参照してください。

sh_addralign

いくつかのセクションには、アドレス整列制約が存在します。たとえば、あるセクションが 2 語で構成されるデータを保持している場合、システムはそのセクション全体に対して 2 語単位の整列を保証しなければなりません。つまり、sh_addr の値は、sh_addralign の値を法として 0 でなければなりません。現在、0、および 2 の非負整数累乗のみが許可されています。値 0 と 1 は、セクションに整列制約が存在しないことを意味します。

sh_entsize

いくつかのセクションは、サイズが一定のエントリのテーブル (シンボルテーブルなど) を保持します。このようなセクションに対してこの構成要素は、各エントリのサイズ (単位: バイト) を与えます。サイズが一定のエントリのテーブルをセクションが保持しない場合、この構成要素には 0 が格納されます。

セクションヘッダーの `sh_type` 構成要素は、表 7-12 に示すようにこのセクションの意味を指定します。

表 7-12 `sh_type` が保持するセクションの型

名前	値
<code>SHT_NULL</code>	0
<code>SHT_PROGBITS</code>	1
<code>SHT_SYMTAB</code>	2
<code>SHT_STRTAB</code>	3
<code>SHT_RELA</code>	4
<code>SHT_HASH</code>	5
<code>SHT_DYNAMIC</code>	6
<code>SHT_NOTE</code>	7
<code>SHT_NOBITS</code>	8
<code>SHT_REL</code>	9
<code>SHT_SHLIB</code>	10
<code>SHT_DYNSYM</code>	11
<code>SHT_SUNW_move</code>	0x6fffffff
<code>SHT_SUNW_COMDAT</code>	0x6ffffffb
<code>SHT_SUNW_syminfo</code>	0x6ffffffc
<code>SHT_SUNW_verdef</code>	0x6ffffffd
<code>SHT_SUNW_verneed</code>	0x6ffffffe

表 7-12 sh_type が保持するセクションの型 続く

名前	値
SHT_SUNW_versym	0x6fffffff
SHT_LOPROC	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

SHT_NULL

この値は、該当セクションヘッダーが使用されないものであることを示します。このセクションには、関連付けられているセクションは存在しません。セクションヘッダーの他の構成要素の値は不定です。

SHT_PROGBITS

このセクションは、プログラムで定義された情報を保持します。プログラムの形式と意味は、プログラムが独自に決定します。

SHT_SYMTAB, SHT_DYNSYM

これらのセクションは、シンボルテーブルを保持します。一般に、SHT_SYMTAB セクションはリンク編集に関するシンボルを与えます。このセクションには完全なシンボルテーブルとして、動的リンクに不要な多くのシンボルが存在することがあります。また、オブジェクトファイルには SHT_DYNSYM セクション (動的リンクシンボルの最小セットを保持して領域を節約している) が存在することがあります。詳細は、237ページの「シンボルテーブル」を参照してください。

SHT_STRTAB, SHT_DYNSTR

これらのセクションは、文字列テーブルを保持します。オブジェクトファイルには、複数の文字列テーブルセクションが存在できます。詳細は、235ページの「文字列テーブル」を参照してください。

SHT_REL

このセクションは、明示的加数が存在する再配置エントリ (32 ビットクラスのオブジェクトファイルの `Elf32_Rela` タイプなど) を保持します。オブジェクトファイルには、複数の再配置セクションが存在できます。詳細は、248ページの「再配置」を参照してください。

SHT_HASH

このセクションは、シンボルハッシュテーブルを保持します。動的にリンクされたすべてのオブジェクトファイルには、シンボルハッシュテーブルが存在しなければなりません。現在、オブジェクトファイルにはハッシュテーブルは1つしか存在できませんが、この制約は将来、緩和されるかもしれません。詳細は、323ページの「ハッシュテーブル」を参照してください。

SHT_DYNAMIC

このセクションは、動的リンクに関する情報を保持します。現在、オブジェクトファイルには動的リンクのセクションは1つしか存在できませんが、この制約は将来、緩和されるかもしれません。詳細は、296ページの「動的セクション」を参照してください。

SHT_NOTE

このセクションは、何らかの方法でファイルを示す情報を保持します。詳細は、274ページの「注釈セクション」を参照してください。

SHT_NOBITS

この型のセクションはファイルの領域を占めませんが、他の点では `SHT_PROGBITS` に類似しています。このセクションにはデータは存在しませんが、`sh_offset` 構成要素には概念上のファイルオフセットが存在します。

SHT_REL

このセクションは、明示的加数が存在しない再配置エントリ (32 ビットクラスのオブジェクトファイルの `Elf32_Rel` 型など) を保持します。オブジェクトファイルには、複数の再配置セクションが存在できます。詳細は、248ページの「再配置」を参照してください。

SHT_SHLIB

このセクション型は予約されていますが、解釈の方法は定義されていません。この型のセクションが存在するプログラムは、ABI に準拠しません。

SHT_SUNW_COMDAT

このセクションには、部分的に初期化されたシンボルを扱うデータが存在します。詳細は、267ページの「Comdat セクション」を参照してください。

SHT_SUNW_move

このセクションには、部分的に初期化されたシンボルを扱うデータが存在します。詳細は、276ページの「移動セクション」を参照してください。

SHT_SUNW_syminfo

このセクションには、追加シンボル情報を保持するテーブルが存在します。詳細は、276ページの「移動セクション」を参照してください。

SHT_SUNW_verdef

このセクションには、このファイルで定義されているきめの細かいバージョンの定義が存在します。詳細は、268ページの「バージョン定義セクション」を参照してください。

SHT_SUNW_verneed

このセクションには、イメージの実行に必要なきめの細かい依存性の記述が存在します。詳細は、272ページの「バージョン依存セクション」を参照してください。

SHT_SUNW_versym

このセクションには、シンボルとバージョン定義(ファイルが与える)の関係を記述するテーブルが存在します。詳細は、271ページの「バージョンシンボルセクション」を参照してください。

SHT_LOPROC - SHT_HIPROC

この範囲の値は、プロセッサ固有な使用方法用に予約されます。

SHT_LOUSER

この値は、アプリケーションプログラムに対して予約されるインデックスの範囲の下限を示します。

SHT_HIUSER

この値は、アプリケーションプログラムに対して予約されるインデックスの範囲の上限を示します。SHT_LOUSER から SHT_HIUSER までのセクション型は、現在の、または将来のシステム定義セクション型と競合することなくアプリケーションで使用できます。

他のセクション型の値は、保留されています。先に述べたとおり、インデックス 0 (SHN_UNDEF) のセクションヘッダーは存在します (このインデックスが未定義セクション参照を示してもです)。その値は表 7-13 の通りです。

表 7-13 セクションヘッダーのテーブルエントリ: インデックス 0

名前	値	注意
sh_name	0	名前が存在しない
sh_type	SHT_NULL	使用されない
sh_flags	0	フラグが存在しない
sh_addr	0	アドレスが存在しない
sh_offset	0	ファイルオフセットが存在しない
sh_size	0	サイズが存在しない
sh_link	SHN_UNDEF	リンク情報が存在しない
sh_info	0	補助情報が存在しない
sh_addralign	0	整列が存在しない
sh_entsize	0	エントリが存在しない

セクションヘッダーの `sh_flags` 構成要素は、セクションの属性を記述する 1 ビットフラグを保持します。

表 7-14 セクションの属性のフラグ

名前	値
<code>SHF_WRITE</code>	<code>0x1</code>
<code>SHF_ALLOC</code>	<code>0x2</code>
<code>SHF_EXECINSTR</code>	<code>0x4</code>
<code>SHF_MERGE</code>	<code>0x10</code>
<code>SHF_STRINGS</code>	<code>0x20</code>
<code>SHF_INFO_LINK</code>	<code>0x40</code>
<code>SHF_LINK_ORDER</code>	<code>0x80</code>
<code>SHF_OS_NONCONFORMING</code>	<code>0x100</code>
<code>SHF_GROUP</code>	<code>0x200</code>
<code>SHF_MASKOS</code>	<code>0x0ff00000</code>
<code>SHF_ORDERED</code>	<code>0x40000000</code>
<code>SHF_EXCLUDE</code>	<code>0x80000000</code>
<code>SHF_MASKPROC</code>	<code>0xf0000000</code>

`sh_flags` にフラグビットが設定されると、属性がセクションに対して「オン」になります。設定されない場合は、属性が「オフ」になるか、または適用されません。定義されていない属性は保留され、0 に設定されています。

SHF_WRITE

このセクションには、プロセス実行時に書き込み可能でなければならないデータが存在します。

SHF_ALLOC

このセクションは、プロセス実行時にメモリーを占めます。いくつかの制御セクションは、オブジェクトファイルのメモリーイメージに存在しません。この属性は、これらのセクションに対してオフです。

SHF_EXECINSTR

このセクションには、実行可能なマシン命令が存在します。

SHF_MERGE

このセクション内のデータは、重複を避けるためにマージすることができます。同時に SHF_STRINGS フラグが設定されていない限り、このセクション内のデータ要素は統一されたサイズになります。各要素のサイズは、セクションヘッダーの `sh_entsize` フィールドで指定されます。同時に SHF_STRINGS フラグも設定されている場合は、データ要素は NULL で終わる文字列で構成されています。各文字のサイズは、セクションヘッダーの `sh_entsize` フィールドで指定されます。

SHF_STRINGS

このセクション内のデータ要素が、NULL で終わる文字列で構成されています。各文字のサイズは、セクションヘッダーの `sh_entsize` フィールドで指定されます。

SHF_INFO_LINK

このセクションヘッダーの `sh_info` フィールドは、セクションヘッダーのテーブルインデックスを保持します。

SHF_LINK_ORDER

このフラグは、リンカーに特別な順序の要求を追加します。この要求は、このセクションのヘッダーの `sh_link` フィールドが別のセクション(リンク先のセクション)を参照する場合に適用されます。このセクションが出力ファイル内の他のセクションと結合される場合には、これらのセクションに関しては、同じ相対的な順序で現われます。すなわち、リンクされるセクションは、それが結合されるセクションに現われます。

sh_info の特殊値である SHN_BEFORE および SHN_AFTER (表 7-11 を参照) は、順序付けされるセット内の他のすべてのセクションに対して、ソートされたセクションがそれぞれ前に付くまたは後に付くことを示します。入力ファイルのリンク行の順序は、1 つの順序付けセット内の複数のセクションがこれらの特殊値のいずれかを持つ場合のために、予約されています。

このフラグを使用する場合の典型的なものとして、アドレスの順序でテキストまたはデータセクションを参照するテーブルを構築する場合があります。

SHF_OS_NONCONFORMING

このセクションは、正しくない動作を避けるために、特別な OS 固有の処理 (標準のリンク処理規則の範囲を越えるもの) を必要とするものです。このセクションが、これらのフィールドに対して sh_type 値を持つか、OS 固有の範囲内にある sh_flags ビットを含み、かつリンカーがこれらの値を認識しない場合は、リンカーはこのセクションを含むオブジェクトファイルを拒否し、エラーを出します。

SHF_GROUP

このセクションは、1 つのセクショングループの (おそらく唯一の) 構成要素です。このセクションは、タイプ SHT_GROUP のセクションに参照されなければなりません。SHF_GROUP フラグは、再配置可能オブジェクト (ET_REL に設定された ELF ヘッダーの e_type 構成要素を持つオブジェクト) に含まれたセクションに対してのみ設定することができます。詳細は、226 ページの「セクショングループ」を参照してください。

SHF_MASKOS

このマスクに含まれるすべてのビットは、オペレーティングシステム特有のセマンティクス (意味的用途) のために予約されています。

SHF_ORDERED

このセクションは、同じ型の他のセクションと順序付けられます。順序付けられるセクションは、sh_link エントリでポイントされるセクション内で結合されます。順序付けられるセクションの sh_link エントリは、自身を指し示すことがあります。

順序付けられるセクションの sh_info エントリが同一入力ファイル内の有効セクションの場合、順序付けられるセクションは、sh_info エントリでポイントされるセクションの出力ファイル内の相対順序付けに基づいて整列されます。

特殊な `sh_info` 値である `SHN_BEFORE` または `SHN_AFTER` (表 7-11 を参照) は、整列対象セクションが順序付け対象となる他のすべてのセクションの前または後に存在することを意味します。順序付けの対象となるセクションの複数にこれらの特殊値の 1 つが存在する場合、入力ファイルが指定された順序は保存されます。

`sh_info` 順序付け情報が存在しない場合、出力ファイルの 1 つのセクション内で結合される 1 つの入力ファイルからのセクションは連続的になり、入力ファイル内の相対順序付けと同じ相対順序付けになります。複数の入力ファイルからの場合は、リンクコマンドで指定された順序になります。

SHF_EXCLUDE

このセクションは、実行可能オブジェクトまたは共有オブジェクトのリンク編集への入力から除かれます。このフラグは、`SHF_ALLOC` フラグが設定されている場合、またはセクションに対する参照が存在する場合、無視されます。

SHF_MASKPROC

このマスクに存在するすべてのビットは、プロセッサ固有な使用方法に予約されません。

セクションヘッダーの 2 つの構成要素 `sh_link` と `sh_info` は、セクション型に従って特殊な情報を保持します。

表 7-15 `sh_link` と `sh_info` の解釈

<code>sh_type</code>	<code>sh_link</code>	<code>sh_info</code>
<code>SHT_DYNAMIC</code>	関連付けられている文字列テーブルのセクションヘッダーインデックス	0
<code>SHT_HASH</code>	関連付けられているシンボルテーブルのセクションヘッダーインデックス	0
<code>SHT_REL</code> <code>SHT_RELA</code>	関連付けられているシンボルテーブルのセクションヘッダーインデックス	再配置が適用されるセクションのセクションヘッダーインデックス。表 7-17 も参照

表 7-15 sh_link と sh_info の解釈 続く

sh_type	sh_link	sh_info
SHT_SYMTAB SHT_DYNSYM	関連付けられている文字列 テーブルのセクションヘッ ダーインデックス	最後のローカルシンボルのシ ンボルテーブルインデック スより 1 大きい (STB_LOCAL に対応する)
SHT_GROUP	関連付けられているシンボル テーブルのセクションヘッ ダーインデックス	関連付けられているシンボル テーブル内のエントリの、シ ンボルテーブルインデック ス。指定されたシンボルテー ブルエントリの名前は、その セクショングループのシグニ チャを提供する
SHT_SUNW_move	関連付けられているシンボル テーブルのセクションヘッ ダーインデックス	0
SHT_SUNW_COMDAT	0	0
SHT_SUNW_syminfo	関連付けられているシンボル テーブルのセクションヘッ ダーインデックス	関連付けられている .dynamic セクションのセク ションヘッダーインデックス
SHT_SUNW_verdef	関連付けられている文字列 テーブルのセクションヘッ ダーインデックス	セクション内のバージョン定 義数
SHT_SUNW_verneed	関連付けられている文字列 テーブルのセクションヘッ ダーインデックス	セクション内のバージョン依 存数
SHT_SUNW_versym	関連付けられているシンボル テーブルのセクションヘッ ダーインデックス	0

セクショングループ

セクションの中には、相互関連のあるグループがあるものがあります。たとえ
ば、インライン関数の out-of-line 定義では、実行可能命令を含むセクションに加え
て、参照定数を含む読み取り専用のデータセクション、1 つまたは複数のデバuggi

ング情報セクション、およびその他の情報セクションを必要とする場合があります。さらに、これらのセクションの中には、セクションのうちの1つが削除されたり他のオブジェクトからの重複によって置き換えられたりすると意味を成さない内部参照がある可能性があります。そのため、これらのグループは、リンクされるオブジェクトに含めるまたはそこから削除する場合は、1つのユニットとして扱う必要があります。

タイプ `SHT_GROUP` のセクションは、そういったセクションのグループ化を定義します。含んでいるオブジェクトのシンボルテーブルのうちの1つからのシンボル名が、そのセクショングループについてのシグニチャを提供します。`SHT_GROUP` セクションのセクションヘッダーが、上記のように識別シンボルエントリを指定します。`sh_link` 構成要素はそのエントリを含むシンボルテーブルセクションのセクションヘッダーインデックスを含み、`sh_info` 構成要素はその識別エントリのシンボルテーブルインデックスを含みます。そのセクションヘッダーの `sh_flags` 構成要素は、0を含みます。そのセクションの名前 (`sh_name`) は指定されません。

`SHT_GROUP` セクションのセクションデータは、`Elf32_Word` エントリの配列です。最初のエントリは、フラグです。残りのエントリは、セクションヘッダーのインデックスのシーケンスです。

現在、以下のフラグが定義されています。

表 7-16 セクショングループのフラグ

名前	値
<code>GRP_COMDAT</code>	0x1

`GRP_COMDAT`

`COMDAT` グループであることを示します。これは、同じグループシグニチャを持つものとして重複が定義されている場合には、他のオブジェクトファイル内の他の `COMDAT` グループと重複する可能性があります。その場合には、重複グループのうち1つのみがリンカーによって保持され、残りのグループの構成要素は破棄されます。

`SHT_GROUP` セクション内のセクションヘッダーインデックスは、そのグループを構成するセクションを識別します。それらの各セクションは、`SHF_GROUP` フラグを `sh_flags` セクションヘッダー構成要素内に設定していなければなりません。リ

リンカーがそのセクショングループを削除することを決めた場合、リンカーはそのグループのすべての構成要素を削除します。

未決定の参照を残すことなく、シンボルテーブルの処理を最小限にしてグループの削除を行うには、次の規則に従う必要があります。

- グループを形成するセクションへのそのグループの外のセクションからの参照は、STB_GLOBAL または STB_WEAK 結合とセクションインデックス SHN_UNDEF を伴うシンボルテーブルエントリを介して行わなければなりません。その参照を含むオブジェクト内に同じシンボルの定義がある場合は、その参照とは別のシンボルテーブルエントリを持つ必要があります。そのグループの外のセクションは、そのグループのセクション内に含まれるアドレスについて STB_LOCAL 結合を持つシンボルを参照しない (タイプ STT_SECTION を持つシンボルを含む、) 可能性があります。
- グループを形成するセクションへのそのグループの外からの非シンボル参照が無い場合もあります。たとえば、sh_link または sh_info 構成要素内でのグループ構成要素のセクションヘッダーインデックスの使用などです。
- そのグループのセクションのうちの1つに関連すると定義され、かつそのグループの一部でないシンボルテーブルセクション内に含まれるシンボルテーブルエントリは、そのグループの構成要素が破棄される場合には、削除されます。

特殊セクション

さまざまなセクションがプログラム情報と制御情報を保持します。以下の一覧表に示されているセクションはシステムで使用されますが、これらのセクションには一覧表で示されている型と属性が存在します。

表 7-17 特殊セクション

名前	型	属性
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	なし
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE

表 7-17 特殊セクション 続く

名前	型	属性
<code>.dynamic</code>	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE
<code>.dynstr</code>	SHT_STRTAB	SHF_ALLOC
<code>.dynsym</code>	SHT_DYNSYM	SHF_ALLOC
<code>.fini</code>	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
<code>.fini_array</code>	SHT_FINI_ARRAY	SHF_ALLOC + SHF_WRITE
<code>.got</code>	SHT_PROGBITS	316ページの「大域オフセットテーブル (プロセッサ固有)」を参照
<code>.hash</code>	SHT_HASH	SHF_ALLOC
<code>.init</code>	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
<code>.init_array</code>	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
<code>.interp</code>	SHT_PROGBITS	294ページの「プログラムインタプリタ」を参照
<code>.note</code>	SHT_NOTE	なし
<code>.plt</code>	SHT_PROGBITS	317ページの「プロシージャのリンクテーブル (プロセッサ固有)」を参照
<code>.preinit_array</code>	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
<code>.rela</code>	SHT_RELA	なし
<code>.relname</code>	SHT_REL	248ページの「再配置」を参照
<code>.relocname</code>	SHT_RELA	248ページの「再配置」を参照
<code>.rodata</code>	SHT_PROGBITS	SHF_ALLOC

表 7-17 特殊セクション 続く

名前	型	属性
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	なし
.strtab	SHT_STRTAB	後続の .strtab 記述を参照
.symtab	SHT_SYMTAB	237ページの「シンボルテーブル」を参照
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.SUNW_bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_move	SHT_SUNW_move	SHF_ALLOC
.SUNW_reloc	SHT_rel SHT_rela	SHF_ALLOC
.SUNW_syminfo	SHT_SUNW_syminfo	SHF_ALLOC
.SUNW_version	SHT_SUNW_verdef SHT_SUNW_verneed SHT_SUNW_versym	SHF_ALLOC

.bss

このセクションは、プログラムのメモリーイメージで使用される初期化されていないデータを保持します。システムは、プログラムが実行を開始すると 0 でデータを初期化することになっています。このセクションは、セクション型 SHT_NOBITS で示されているとおり、ファイルスペースを占めません。

`.comment`

このセクションは、コメント情報を保持し、このセクションはコメント情報を保持し、通常、コンパイルシステムのコンポーネントが使用します。このセクションは `mcs(1)` によって操作できます。

`.data`、`.data1`

これらのセクションは、プログラムのメモリーイメージに使用される初期化されているデータを保持します。

`.dynamic`

このセクションは、動的リンク情報を保持します。詳細は、296ページの「動的セクション」を参照してください。

`.dynstr`

このセクションは、動的リンクに必要な文字列 (最も一般的には、シンボルテーブルエントリに関連付けられている名前を表す文字列) を保持します。

`.dynsym`

このセクションは、動的リンクシンボルテーブルを保持します。詳細は、237ページの「シンボルテーブル」を参照してください。

`.fini`

このセクションは、プロセス終了時に使用される実行可能命令を保持します。つまり、プログラムが正常終了すると、システムはこのセクションの命令を実行できるようにします。詳細は、84ページの「初期設定および終了ルーチン」を参照してください。

`.fini_array`

このセクションは、そのセクションを含む実行可能ファイルまたは共有オブジェクトの単一の終了配列に使用される関数ポインタの配列を保持します。詳細は、84ページの「初期設定および終了ルーチン」を参照してください。

.got

このセクションは、大域オフセットテーブルを保持します。詳細は、316ページの「大域オフセットテーブル (プロセッサ固有)」を参照してください。

.hash

このセクションは、シンボルハッシュテーブルを保持します。詳細は、323ページの「ハッシュテーブル」を参照してください。

.init

このセクションは、プロセス初期化時に使用される実行可能命令を保持します。つまり、プログラムが実行を開始すると、システムはプログラムエントリポイントを呼び出す前にこのセクションの命令を実行できるようにします。詳細は、84ページの「初期設定および終了ルーチン」を参照してください。

.init_array

このセクションは、そのセクションを含む実行可能ファイルまたは共有オブジェクトの単一の初期設定配列に使用される関数ポインタの配列を保持します。詳細は、84ページの「初期設定および終了ルーチン」を参照してください。

.interp

このセクションは、プログラムインタプリタのパス名を保持します。詳細は、294ページの「プログラムインタプリタ」を参照してください。

.note

このセクションは、274ページの「注釈セクション」に記述されている形式で情報を保持します。

.plt

このセクションは、プロシージャのリンクテーブルを保持します。詳細は、317ページの「プロシージャのリンクテーブル (プロセッサ固有)」を参照してください。

`.preinit_array`

このセクションは、そのセクションを含む実行可能ファイルまたは共有オブジェクトの単一の初期設定前の配列に使用される関数ポインタの配列を保持します。詳細は、84ページの「初期設定および終了ルーチン」を参照してください。

`.rela`

このセクションは、特定のセクションに適用されないレジスタ再配置情報を保持します。このセクションの用途の1つは、レジスタの再配置です。詳細は、246ページの「レジスタシンボル」を参照してください。

`.relname`、`.relname`

これらのセクションは、再配置情報(248ページの「再配置」に記述されている)を保持します。再配置が存在する読み込み可能セグメントがファイルに存在する場合、これらのセクションの属性として `SHF_ALLOC` ビットがオンになります。そうでない場合、このビットはオフになります。慣例により、`name` は再配置が適用されるセクションの名前になります。したがって、`.text` の再配置セクションには、通常 `.rel.text` または `.rela.text` という名前が存在します。

`.rodata`、`.rodata1`

これらのセクションは、読み取り専用データを保持します。このデータは、通常はプロセスイメージの書き込み不可セグメントに使用されます。詳細は、279ページの「プログラムヘッダー」を参照してください。

`.shstrtab`

このセクションは、セクション名を保持します。

`.strtab`

このセクションは、文字列(最も一般的には、シンボルテーブルエントリに関連付けられている名前を表す文字列)を保持します。シンボル文字列テーブルが存在する読み込み可能セグメントがファイルに存在する場合、セクションの属性として `SHF_ALLOC` ビットがオンになります。そうでない場合、このビットはオフになります。

`.symtab`

このセクションは、237ページの「シンボルテーブル」に記述しているとおおり、シンボルテーブルを保持します。シンボルテーブルが存在する読み込み可能セグメントがファイルに存在する場合、セクションの属性として `SHF_ALLOC` ビットがオンになります。そうでない場合、このビットはオフになります。

`.text`

このセクションは、プログラムの「テキスト」すなわち実行可能命令を保持します。

`.SUNW_bss`

このセクションは、プログラムのメモリーイメージで使用される、共有オブジェクトの部分的に初期化されたデータを保持します。データは実行時に初期化されます。このセクションは、セクション型 `SHT_NOBITS` で示しているとおおり、ファイル領域を占めません。

`.SUNW_heap`

このセクションは、`dldump(3DL)` によって作成される動的実行可能ファイルのデータ領域(ヒープ)を保持します。

`.SUNW_move`

このセクションは、部分的に初期化されたデータに関する追加情報を保持します。276ページの「移動セクション」を参照してください。

`.SUNW_reloc`

このセクションは、248ページの「再配置」で記述しているとおおり、再配置情報を保持します。このセクションは再配置セクションが連結されたものであり、個々の再配置レコードに対するより良い参照のローカル性(局所性)を与えます。再配置レコード自身のオフセットのみが意味があり、したがってセクション `sh_info` の値は0です。

`.SUNW_syminfo`

このセクションは、シンボルテーブルの追加情報を保持します。詳細は、247ページの「Syminfo テーブル」を参照してください。

`.SUNW_version`

この名前を持つセクションは、バージョン情報を保持します。詳細は、268ページの「バージョン情報」を参照してください。

ドット (.) 接頭辞付きのセクション名はシステムにおいて予約されています。これらのセクションの既存の意味が満足できるものであれば、アプリケーションはこれらのセクションを使用できます。アプリケーションは、ドット (.) 接頭辞なしの名前を使用して、システムで予約されたセクションとの競合を回避することができます。オブジェクトファイル形式では、上記リストに記載されていないセクションが定義できます。オブジェクトファイルには、同じ名前を持つ複数のセクションが存在できます。

プロセッサアーキテクチャに対して予約されるセクション名は、アーキテクチャ名の省略形をセクション名の前に入れることで作成されます。セクション名の前に、`e_machine` に対して使用されるアーキテクチャ名を入れる必要があります。たとえば、`.Foo.psect` は、FOO アーキテクチャで定義される `psect` セクションです。

既存の拡張セクションは、従来から使用されている名前をそのまま使用しています。

既存の拡張セクション

<code>.conflict</code>	<code>.liblist</code>	<code>.lit8</code>	<code>.sdata</code>
<code>.debug</code>	<code>.line</code>	<code>.reginfo</code>	<code>.stab</code>
<code>.gptab</code>	<code>.lit4</code>	<code>.sbss</code>	<code>.tdesc</code>

文字列テーブル

文字列テーブルセクションは、空文字で終了する一連の文字 (一般に文字列と呼ばれている) を保持します。オブジェクトファイルは、これらの文字列を使用してシンボルとセクション名を表します。文字列は、文字列テーブルセクションへのインデックスとして参照されます。

先頭バイト (インデックス 0) は、空文字を保持します。同様に、文字列テーブルの最後のバイトは、空文字を保持します。したがって、すべての文字列は確実に空文字で終了します。インデックスが 0 の文字列は、名前を指定しないか空文字の名前を指定します (状況に依存する)。

空の文字列テーブルセクションが許可されており、このセクションのセクションヘッダーの `sh_size` 構成要素に 0 が入ります。0 以外のインデックスは、空の文字列テーブルに対して無効です。

セクションヘッダーの `sh_name` 構成要素は、ELF ヘッダーの `e_shstrndx` 構成要素で示されているとおり、セクションヘッダー文字列テーブルセクションへのインデックスを保持します。図 7-4 は、25 バイトの文字列テーブルと、さまざまなインデックスに関連付けられている文字列を示しています。

インデックス	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

図 7-4 文字列テーブル

表 7-18 は、上に示した文字列テーブルの文字列を示しています。

表 7-18 文字列テーブルインデックス

インデックス	文字列
0	なし
1	name
7	Variable
11	able
16	able
24	空文字列

例で示しているとおり、文字列テーブルインデックスはセクションのすべてのバイトを参照できます。文字列は 2 回以上現れることができ、部分文字列に対する参照は存在でき、単一文字列は複数回参照できます。参照されない文字列も許可されます。

シンボルテーブル

オブジェクトファイルのシンボルテーブルは、プログラムのシンボル定義または参照の探索と再配置に必要な情報を保持します。シンボルテーブルインデックスは、この配列への添字です。インデックス 0 はシンボルテーブルの先頭エントリを指定し、また未定義シンボルインデックスとして機能します。先頭エントリの内容は、このセクションの後の方で記述します。

表 7-19 シンボルテーブルの先頭エントリ

名前	値
STN_UNDEF	0

シンボルテーブルエントリの形式 (`sys/elf.h` で定義されている) は、次のとおりです。

```
typedef struct {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half    st_shndx;
} Elf32_Sym;

typedef struct {
    Elf64_Word    st_name;
    unsigned char st_info;
    unsigned char st_other;
    Elf64_Half    st_shndx;
    Elf64_Addr    st_value;
    Elf64_Xword   st_size;
} Elf64_Sym;
```

`st_name`

この構成要素は、オブジェクトファイルのシンボル文字列テーブルへのインデックス (シンボル名の文字表現を保持する) を保持します。値が 0 以外の場合、その値はシンボル名を与える文字列テーブルインデックスを表します。値が 0 の場合、シンボルテーブルエントリに名前は存在しません。

注 - 外部 C シンボルは、C とオブジェクトファイルのシンボルテーブルにおいて同じ名前を持ちます。

st_value

この構成要素は、関連付けられているシンボルの値を与えます。この値は、絶対値やアドレスなど(状況に依存する)を表します。245ページの「シンボル値」を参照してください。

st_size

多くのシンボルは、関連付けられているサイズを持っています。たとえば、データオブジェクトのサイズは、データオブジェクトに存在するバイト数です。この構成要素は、シンボルがサイズを持っていない場合またはサイズが不明な場合、0を保持します。

st_info

この構成要素は、シンボルの種類と結び付けられる属性を指定します。値と意味のリストを以下に示します。次のコードは、値(sys/elf.hで定義されている)の処理方法を示します。

```
#define ELF32_ST_BIND(i)          ((i) >> 4)
#define ELF32_ST_TYPE(i)         ((i) & 0xf)
#define ELF32_ST_INFO(b, t)      ((b)<<4)+((t)&0xf)

#define ELF64_ST_BIND(info)      ((info) >> 4)
#define ELF64_ST_TYPE(info)     ((info) & 0xf)
#define ELF64_ST_INFO(bind, type) ((bind)<<4)+((type)&0xf)
```

st_other

この構成要素は、現在シンボルの可視性のために使用されています。値とその意味に関しては、後述のとおりです。以下のコードは、32ビットオブジェクトと64ビットオブジェクトについて値を操作する方法を示しています。その他のビットは0を含んでおり、特に意味はありません。

```
#define ELF32_ST_VISIBILITY(o)  ((o)&0x3)
#define ELF64_ST_VISIBILITY(o)  ((o)&0x3)
```

st_shndx

すべてのシンボルテーブルエントリは、何らかのセクションに関して定義されます。この構成要素は、該当するセクションヘッダーテーブルインデックスを保持し

ます。いくつかのセクションインデックスは、特別な意味を示します。表 7-12 を参照してください。

シンボルの結び付けは、リンクの可視性と動作を決定します。

表 7-20 シンボルの結び付け (ELF32_ST_BIND、ELF64_ST_BIND)

名前	値
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOOS	10
STB_HIOS	12
STB_LOPROC	13
STB_HIPROC	15

STB_LOCAL

ローカルシンボルは、ローカルシンボルの定義が存在するオブジェクトファイルの外部では見えません。同じ名前のローカルシンボルは、互いに干渉することなく複数のファイルに存在できます。

STB_GLOBAL

大域シンボルは、結合されるすべてのオブジェクトファイルで見ることができます。あるファイルの大域シンボルの定義は、その大域シンボルへの別ファイルの未定義参照を解決します。

STB_WEAK

ウィークシンボルは大域シンボルに似ていますが、ウィークシンボルの定義の優先順位は大域シンボルの定義より低いです。

STB_LOOS - STB_HIOS

この範囲の値 (両端の値も含む) は、オペレーティングシステム特有のセマンティクス用に予約されています。

STB_LOPROC - STB_HIPROC

この範囲の値は、プロセッサ固有の使用方法に対して予約されます。

大域シンボルとウィークシンボルは、主に2つの点で異なります。

- リンカーは、いくつかの再配置可能オブジェクトファイルを結合するとき、同じ名前の STB_GLOBAL シンボルの複数の定義を許可しない。一方、定義された大域シンボルが存在している場合、同じ名前のウィークシンボルが現れてもエラーは発生しない。リンカーは大域定義を使用し、ウィーク定義を無視する。

同様に、共通シンボルが存在している場合 (つまり、SHN_COMMON を保持している st_shndx フィールドが存在するシンボルが存在している場合)、同じ名前のウィークシンボルが現れてもエラーは発生しない。リンカーは共通定義を使用し、ウィーク定義を無視する。詳細は、41ページの「シンボル解析」を参照してください。

- リンカーは、アーカイブライブラリ (30ページの「アーカイブ処理」を参照) を検索するとき、大域シンボル (未定義または一時的) の定義が存在するアーカイブ構成要素を抜き出す。構成要素の定義は、大域シンボルまたはウィークシンボル

リンカーはデフォルトでは、未定義のウィークシンボルを解決するためのアーカイブ構成要素を抜き出さない。解決されていないウィークシンボルは、値0を持つ。-z weakextract を使用すると、このデフォルトの動作が無効になり、ウィーク参照がアーカイブ構成要素を抜き出すことができる。

注 - ウィークシンボルは、主にシステムソフトウェアでの使用を意図したものです。アプリケーションプログラムでの使用は推奨されません。

各シンボルテーブルにおいて、STB_LOCAL 結び付きが存在するすべてのシンボルは、ウィークシンボルと大域シンボルの前に存在します。212ページの「セクション」に記述されているとおり、シンボルテーブルセクションの sh_info セクションヘッダー構成要素は、最初のローカルではないシンボルに対するシンボルテーブルインデックスを保持します。

シンボルの種類は、関連付けられている実体に関する一般的分類を行います。

表 7-21 シンボルの種類 (ELF32_ST_TYPE、ELF64_ST_TYPE)

名前	値
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_COMMON	5
STT_LOOS	10
STT_HIOS	12
STT_LOPROC	13
STT_SPARC_REGISTER	13
STT_HIPROC	15

STT_NOTYPE

シンボルの種類は指定されません。

STT_OBJECT

シンボルは、データオブジェクト (変数や配列など) と関連付けられています。

STT_FUNC

シンボルは、関数または他の実行可能コードに関連付けられています。

STT_SECTION

シンボルは、セクションに関連付けられています。この種類のシンボルテーブルエントリは主に再配置を行うために存在しており、通常、STB_LOCAL に結び付けられています。

STT_FILE

慣例により、シンボルの名前はオブジェクトファイルに対応するソースファイルの名前を与えます。ファイルシンボルは STB_LOCAL に結び付けられており、セクションインデックスは SHN_ABS であり、ファイルシンボル (存在する場合) はファイルの他の STB_LOCAL シンボルの前に存在します。SHT_SYMTAB のシンボルインデックス 1 は、ファイル自身を表す STT_FILE シンボルです。慣例により、この後にはファイルの STT_SECTION シンボルと、ローカルシンボルに短縮されている大域シンボルが続きます (詳細は、56ページの「シンボル範囲の縮小」と、第 5 章を参照)。

STT_COMMON

このシンボルは、初期設定されていない共通ブロックを表しますが、これは STT_OBJECT とまったく同じに扱われます。

STT_LOOS - STT_HIOS

この範囲の値 (両端の値を含む) は、オペレーティングシステム固有のセマンティクス (意味的用途) のために予約されています。

STT_LOPROC - STT_HIPROC

この範囲の値 (両端の値を含む) は、プロセッサ固有の使用方法に対して予約されています。

共有オブジェクトファイルの関数シンボル (STT_FUNC タイプが存在する) は、特別な意味を持っています。別のオブジェクトファイルが共有オブジェクトの関数を参照すると、リンカーは参照されるシンボルのプロシージャのリンクテーブルエントリを自動的に作成します。STT_FUNC 以外のタイプの共有オブジェクトシンボルは、プロシージャのリンクテーブルから自動的に参照されません。

シンボルの可視性は、(再配置可能オブジェクトの中に指定することもできますが、) シンボルがいったん実行可能ファイルまたは共有オブジェクトの一部になってからどのようにアクセスされるかを定義します。

表 7-22 シンボルの可視性

名前	値
STV_DEFAULT	1
STV_INTERNAL	2
STV_HIDDEN	3
STV_PROTECTED	4

STV_DEFAULT

STV_DEFAULT 属性を持つシンボルの可視性は、シンボルの結合タイプで指定されたものになります。つまり、大域シンボルおよびウィークシンボルは、それらの定義するコンポーネント (実行可能ファイルまたは共有オブジェクト) の外から見る事ができます。ローカルシンボルは、後述のように「隠されて」います。また、大域シンボルおよびウィークシンボルは「横取り可能な」ものです。つまり、それらは別のコンポーネント内の同じ名前の定義によってオーバーライドが可能なものです。

STV_PROTECTED

現在のコンポーネン内で定義されたシンボルは、それが他のコンポーネン内で参照可能であるが横取り可能ではない場合、保護されています。つまり、定義するコンポーネン内からのそのようなシンボルへの参照は、たとえデフォルトの規則によって間に入るような別のコンポーネンの定義があったとしても、そのコンポーネンの定義にもとづいて解決されなければなりません。STB_LOCAL 結合を持つシンボルは、STV_PROTECTED 可視性を持ちません。

STV_HIDDEN

現在のコンポーネン内で定義されたシンボルは、その名前が他のコンポーネンから参照することができない場合、「隠されて」います。そのようなシンボルは、保護される必要があります。この属性は、コンポーネンの外部インタフェースの管理に使用されます。そのようなシンボルによって指定されたオブジェクトは、そのアドレスが外部に渡された場合、他のコンポーネンから参照できることに留意してください。

再配置可能オブジェクトに含まれた「隠された」シンボルは、その再配置可能オブジェクトが実行可能ファイルまたは共有オブジェクトに含まれる時には、リンカーによって削除されるか STB_LOCAL 結合に変換されます。

STV_INTERNAL

この可視性の属性は、現在予約されています。将来使用される場合は、そのセマンティクスは少なくとも STV_HIDDEN と同じくらい制約の厳しいものとなります。

可視性の属性は、リンク編集中、実行可能ファイルまたは共有オブジェクト内のシンボルの解決には全く影響をおよぼしません。このような解決は、結合タイプによって制御されます。いったんリンカーがその解決を選択すると、これらの属性は以下の2つの必要条件を課します。どちらの必要条件も、リンクされるコード内の参照は、属性の利点を利用するために最適化されるという事実に基づくものです。

- 1つ目に、すべてのデフォルトでない可視性の属性は、シンボルの参照に適用される際、その参照を満たすための定義は現在の実行可能ファイルまたは共有オブジェクト内で提供されなければならないという条件を課します。このようなシンボルの参照がリンクされるコンポーネント内に定義を持たない場合は、その参照は STB_WEAK 結合を持つ必要があり、0 に解決されます。
- 2つ目に、名前への参照または名前の定義がデフォルトでない可視性の属性を持つシンボルである場合、その可視性の属性はリンクされるオブジェクト内の解決シンボルへ伝達されなければなりません。特定のシンボルへの参照または特定のシンボルの定義に対して異なる可視性の属性が指定されている場合は、最も制約の厳しい可視性の属性が、リンクされるオブジェクト内の解決シンボルへ伝達されなければなりません。属性は、最も制約の少ないものから最も制約の厳しいものの順に、STV_PROTECTED、STV_HIDDEN、STV_INTERNAL となります。

シンボル値がセクション内の特定位置を参照すると、セクションインデックス構成要素 st_shndx は、セクションヘッダーテーブルへのインデックスを保持します。再配置時にセクションが移動すると、シンボル値も変化し、シンボルへの参照はプログラム内の同じ位置を指し示し続けます。いくつかの特殊セクションのインデックス値は、他の意味付けがされています。

SHN_ABS

シンボルは、絶対値(再配置が行われても変化しない)を持ちます。

SHN_COMMON

シンボルは、割り当てられていない共通ブロックを示します。シンボル値は、セクションの `sh_addralign` 構成要素に類似した整列制約を与えます。つまり、リンカーは `st_value` の倍数のアドレスにシンボル記憶領域を割り当てます。シンボルのサイズは、必要なバイト数を示します。

SHN_UNDEF

このセクションテーブルインデックスは、シンボルが未定義であることを意味します。リンカーがこのオブジェクトファイルを、示されたシンボルを定義する他のオブジェクトファイルに結合すると、このシンボルに対するこのファイルの参照は実際の定義に結び付けられます。

前述したとおり、インデックス 0 (`STN_UNDEF`) のシンボルテーブルエントリは予約されています。このシンボルテーブルエントリは以下の値を保持します。

表 7-23 シンボルテーブルエントリ: インデックス 0

名前	値	注意
<code>st_name</code>	0	名前はない
<code>st_value</code>	0	値は 0
<code>st_size</code>	0	サイズはない
<code>st_info</code>	0	種類はない。ローカル結合
<code>st_other</code>	0	
<code>st_shndx</code>	<code>SHN_UNDEF</code>	セクションは存在しない

シンボル値

異なる複数のオブジェクトファイル型のシンボルテーブルエントリは、`st_value` 構成要素に対してわずかに異なる解釈を持ちます。

- 再配置可能ファイルでは、`st_value` はセクションインデックスが `SHN_COMMON` であるシンボルに対する整列制約を保持する
- 再配置可能ファイルでは、`st_value` は定義されたシンボルに対するセクションオフセットを保持する。つまり、`st_value` は、`st_shndx` が識別するセクションの先頭からのオフセット
- 実行可能オブジェクトファイルと共有オブジェクトファイルでは、`st_value` は仮想アドレスを保持する。これらのファイルのシンボルを実行時リンカーに対してより有用にするために、セクションオフセット (ファイル解釈) の代わりに、セクション番号が無関係な仮想アドレス (ファイル解釈) が使用される

シンボルテーブル値は、異なる種類のオブジェクトファイルでも似た意味を持ちますが、適切なプログラムはデータに効率的にアクセスできます。

レジスタシンボル

SPARC アーキテクチャは、レジスタシンボル (大域レジスタを初期化するシンボル) をサポートします。レジスタシンボルに対するシンボルテーブルエントリには、以下の値が入ります。

表 7-24 SPARC: シンボルテーブルエントリ: レジスタシンボル

フィールド	意味
<code>st_name</code>	シンボル名文字列テーブルへのインデックス。または 0 (スラッシュレジスタ)
<code>st_value</code>	レジスタ番号。整数レジスタの割り当てについては、ABI マニュアルを参照
<code>st_size</code>	未使用 (0)
<code>st_info</code>	結び付きは標準的には <code>STB_GLOBAL</code> 。種類は <code>STT_SPARC_REGISTER</code> でなければならない
<code>st_other</code>	未使用 (0)
<code>st_shndx</code>	<code>SHN_ABS</code> (このオブジェクトがこのレジスタシンボルを初期化する場合)。 <code>SHN_UNDEF</code> (それ以外の場合)

表 7-25 SPARC: レジスタ番号

名前	値	意味
STO_SPARC_REGISTER_G2	0x2	%g2
STO_SPARC_REGISTER_G3	0x3	%g3

特定の領域レジスタのエントリが存在しないことは、その特定の領域レジスタがオブジェクトで使用されないことを意味します。

Syminfo テーブル

このセクションには、Elf32_Syminfo 型または Elf64_Syminfo 型の複数のエントリが存在します。関連付けられているシンボルテーブルの各エントリ (sh_link) の .SUNW_syminfo セクションには、1つのエントリが存在します。

このセクションがオブジェクトに存在している場合、関連付けられているシンボルテーブルからシンボルインデックスを取り出し、このシンボルインデックスを使ってこのセクションに存在する対応する Elf32_Syminfo または Elf64_Syminfo エントリを見つけることで、追加シンボル情報を見つけます。関連付けられているシンボルテーブルと、Syminfo テーブルには、必ず同じ数のエントリが存在します。

インデックス 0 は、Syminfo テーブルの現バージョン (SYMINFO_CURRENT) を格納するために使用されます。シンボルテーブルエントリ 0 は必ず UNDEF シンボルテーブルエントリに対して予約されるので、矛盾は発生しません。

Syminfo エントリの形式は、以下のとおりです。

```
typedef struct {
    Elf32_Half    si_boundto;
    Elf32_Half    si_flags;
} Elf32_Syminfo;

typedef struct {
    Elf64_Half    si_boundto;
    Elf64_Half    si_flags;
} Elf64_Syminfo;
```

si_flags

これはビットフィールドであり、表 7-26 に示すようにフラグを設定できます。

表 7-26 Syminfo フラグ

名前	値	意味
SYMINFO_FLG_DIRECT	0x01	このシンボルは、外部オブジェクトを直接参照した
SYMINFO_FLG_COPY	0x04	このシンボルは、コピー再配置を行った結果
SYMINFO_FLG_LAZYLOAD	0x08	このシンボルは、レイジーロード可能な外部オブジェクトを参照する

si_boundto

sh_info フィールドで示される .dynamic セクションのエントリへのインデックス。このエントリは、DT_* (DT_NEEDED) エントリ (Syminfo エントリに関連付けられている動的オブジェクトを示す) を示します。表 7-27 に示すエントリは、si_boundto に対して予約されています。

表 7-27 si_boundto 予約値

名前	値	意味
SYMINFO_BT_SELF	0xffff	自己に結びつけられるシンボル
SYMINFO_BT_PARENT	0xfffe	親に結びつけられるシンボル。親は、この動的オブジェクトの読み込みを発生させる最初のオブジェクト

再配置

再配置は、記号参照を記号定義に関連付ける処理です。たとえば、プログラムが関数を呼び出すとき、関連付けられている呼び出し命令は、実行時に適切な宛先アドレスに制御を渡さなければなりません。つまり、再配置可能ファイルには、セクション内容の変更方法を示す情報が存在しなければなりません。その結果、実行可

能オブジェクトファイルと共有オブジェクトファイルは、プロセスのプログラムイメージに関する正しい情報を保持できます。再配置エントリは、これらのデータを保持します。

再配置エントリは、以下の構造体 (`sys/elf.h` で定義されている) を持つことができます。

```
typedef struct {
    Elf32_Addr    r_offset;
    Elf32_Word    r_info;
} Elf32_Rel;

typedef struct {
    Elf32_Addr    r_offset;
    Elf32_Word    r_info;
    Elf32_Sword   r_addend;
} Elf32_Rela;

typedef struct {
    Elf64_Addr    r_offset;
    Elf64_Xword   r_info;
} Elf64_Rel;

typedef struct {
    Elf64_Addr    r_offset;
    Elf64_Xword   r_info;
    Elf64_Sxword  r_addend;
} Elf64_Rela;
```

`r_offset`

この構成要素は、再配置処理を適用する位置を与えます。再配置可能ファイルの場合、値はセクションの先頭から再配置の影響を受ける領域までのオフセットです。実行可能ファイルまたは共有オブジェクトの場合、値は再配置の影響を受ける領域の仮想アドレスです。

`r_info`

この構成要素は、再配置が行われなければならないシンボルテーブルインデックスと、適用される再配置の種類を与えます。たとえば、呼び出し命令の再配置エントリは、呼び出される関数のシンボルテーブルインデックスを保持します。インデックスが `STN_UNDEF` (未定義シンボルインデックス) の場合、再配置はシンボル値として 0 を使用します。再配置の種類はプロセッサに固有です。再配置の種類は以下に記述します。以下のテキストが再配置エントリの再配置の種類またはシンボルテーブルインデックスを参照すると、それぞれ `ELF32_R_TYPE` または `ELF32_R_SYM` をエントリの `r_info` 構成要素に適用した結果が得られます。

```

#define ELF32_R_SYM(i)                ((i)>>8)
#define ELF32_R_TYPE(i)              ((unsigned char)(i))
#define ELF32_R_INFO(s, t)           (((s)<<8)+(unsigned char)(t))

#define ELF64_R_SYM(info)            ((info)>>32)
#define ELF64_R_TYPE(info)          ((Elf64_Word)(info))
#define ELF64_R_INFO(sym, type)     (((Elf64_Xword)(sym)<<32)+ \
                                       (Elf64_Xword)(type))

```

Elf64_Rel および Elf64_Rela 構造の場合、r_info フィールドはさらに 8 ビットの識別子と 24 ビットの付随的なデータに分割されます。

```

#define ELF64_R_TYPE_DATA(info)      (((Elf64_Xword)(info)<<32)>>40)
#define ELF64_R_TYPE_ID(info)       (((Elf64_Xword)(info)<<56)>>56)
#define ELF64_R_TYPE_INFO(data, type) (((Elf64_Xword)(data)<<8)+ \
                                         (Elf64_Xword)(type))

```

r_addend

この構成要素は、再配置可能フィールドに格納される値の計算に使用される定数加数を指定します。

前述したとおり、Elf*_Rela エントリにのみ明示的加数が存在します。タイプ Elf*_Rel のエントリには、変更される位置に暗黙の加数が存在します。32 ビット SPARC は Elf32_Rela エントリ、64 ビット SPARC は Elf64_Rela エントリ、IA は Elf32_Rel エントリをそれぞれ使用します。

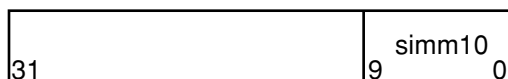
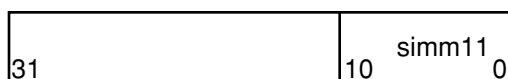
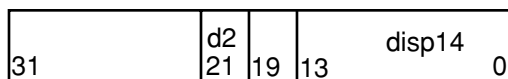
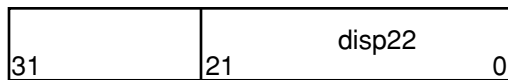
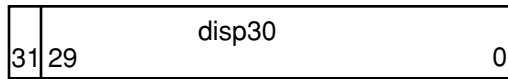
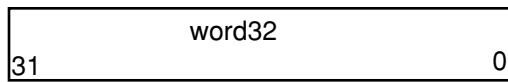
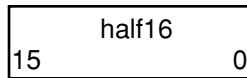
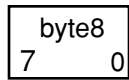
再配置セクションは、他の 2 つのセクション (シンボルテーブルと変更されるセクション) を参照します。セクションヘッダーの sh_info と sh_link 構成要素 (212 ページの「セクション」で記述している) は、これらの関係を指定します。異なる複数のオブジェクトファイルに対する再配置エントリには、r_offset 構成要素に関してわずかに異なる解釈が存在します。

- 再配置可能ファイルでは r_offset は、セクションオフセットを保持する。つまり、再配置セクション自身はファイルの別セクションの変更方法を記述する。再配置オフセットは、2 番目のセクション内の領域を指定する
- 実行可能オブジェクトファイルと共有オブジェクトファイルでは、r_offset は仮想アドレスを保持する。これらのファイルの再配置エントリを実行時リンカーに対してより有用にするために、セクションオフセット (ファイル解釈) の代わりに、仮想アドレス (メモリー解釈) が使用される

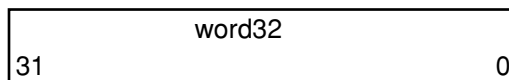
r_offset の解釈は異なる複数のオブジェクトファイルでは変化し、その結果、関連プログラムによる効率的アクセスが可能になっていますが、再配置タイプの意味は変化しません。

再配置型 (プロセッサ固有の)

SPARC の場合、再配置エントリは以下の命令/データフィールドの変更方法を記述します (ビット番号はボックスの下隅に表示される)。



IA の場合、再配置エントリは以下の命令/データフィールドの変更方法を記述します (ビット番号はボックスの下隅に表示される)。



意味します。プロシージャのリンクテーブルエントリは、関数呼び出しを適切な宛先に変更します。リンカーは初期プロシージャのリンクテーブルを作成し、実行時リンカーは実行時にエントリを変更します。詳細は、317ページの「プロシージャのリンクテーブル (プロセッサ固有)」を参照してください。

- P 再配置される領域の位置 (セクションオフセットまたはアドレス) (`r_offset` を使って計算される) を意味します。
- S インデックスが再配置エントリに存在するシンボルの値を意味します。

SPARC 再配置エントリは、バイト (`byte8`)、ハーフワード (`half16`)、またはワード (その他) に適用されます。IA 再配置エントリはワードに適用されます。どの場合も `r_offset` 値は、影響が与えられる領域の先頭バイトのオフセットまたは仮想アドレスを指定します。再配置タイプは、変更されるビットと、これらのビットの値の計算方法を指定します。

32 ビット SPARC は明示的加数が存在する `Elf32_Rela` 再配置エントリのみを使用し、64 ビット SPARC は `Elf64_Rela` を使用します。したがって、`r_addend` 構成要素は再配置加数として機能します。IA は `Elf32_Rel` 再配置エントリのみを使用し、再配置されるフィールドは加数を保持します。すべての場合において、加数と計算された結果は同じバイト順序を使用します。

SPARC: 再配置型

表 7-28 に示されているフィールド名は、再配置型がオーバーフローを検査するかどうかを通知します。計算される再配置値は意図したフィールドより大きい場合があり、再配置の型によっては値の適合を検証 (V) したり結果を切り捨てたり (T) することがあります。たとえば、`V-simm13` は、計算された値が `simm13` フィールドの外部に 0 以外の有意ビットを持つことがないことを意味します。

表 7-28 SPARC: 再配置型

名前	値	フィールド	計算
<code>R_SPARC_NONE</code>	0	なし	なし
<code>R_SPARC_8</code>	1	V-byte8	S + A

表 7-28 SPARC: 再配置型 続く

名前	値	フィールド	計算
R_SPARC_16	2	V-half16	S + A
R_SPARC_32	3	V-word32	S + A
R_SPARC_DISP8	4	V-byte8	S + A - P
R_SPARC_DISP16	5	V-half16	S + A - P
R_SPARC_DISP32	6	V-disp32	S + A - P
R_SPARC_WDISP30	7	V-disp30	(S + A - P) >> 2
R_SPARC_WDISP22	8	V-disp22	(S + A - P) >> 2
R_SPARC_HI22	9	T-imm22	(S + A) >> 10
R_SPARC_22	10	V-imm22	S + A
R_SPARC_13	11	V-simm13	S + A
R_SPARC_LO10	12	T-simm13	(S + A) & 0x3ff
R_SPARC_GOT10	13	T-simm13	G & 0x3ff
R_SPARC_GOT13	14	V-simm13	G
R_SPARC_GOT22	15	T-simm22	G >> 10
R_SPARC_PC10	16	T-simm13	(S + A - P) & 0x3ff
R_SPARC_PC22	17	V-disp22	(S + A - P) >> 10
R_SPARC_WPLT30	18	V-disp30	(L + A - P) >> 2
R_SPARC_COPY	19	なし	なし
R_SPARC_GLOB_DAT	20	V-word32	S + A

表 7-28 SPARC: 再配置型 続く

名前	値	フィールド	計算
R_SPARC_JMP_SLOT	21	なし	「R_SPARC_JMP_SLOT」を参照
R_SPARC_RELATIVE	22	V-word32	$B + A$
R_SPARC_UA32	23	V-word32	$S + A$
R_SPARC_PLT32	24	V-word32	$L + A$
R_SPARC_HIPLT22	25	T-imm22	$(L + A) \gg 10$
R_SPARC_LOPLT10	26	T-simm13	$(L + A) \& 0x3ff$
R_SPARC_PCPLT32	27	V-word32	$L + A - P$
R_SPARC_PCPLT22	28	V-disp22	$(L + A - P) \gg 10$
R_SPARC_PCPLT10	29	V-simm13	$(L + A - P) \& 0x3ff$
R_SPARC_10	30	V-simm10	$S + A$
R_SPARC_11	31	V-simm11	$S + A$
R_SPARC_WDISP16	40	V-d2/disp14	$(S + A - P) \gg 2$
R_SPARC_WDISP19	41	V-disp19	$(S + A - P) \gg 2$
R_SPARC_7	43	V-imm7	$S + A$
R_SPARC_5	44	V-imm5	$S + A$
R_SPARC_6	45	V-imm6	$S + A$

いくつかの再配置型には、単純な計算を超えた意味があります。

R_SPARC_GOT10

この再配置型は R_SPARC_LO10 に似ています。ただし、次の点が異なります。つまり、この再配置型はシンボルの大域オフセットテーブルエントリのアドレスを参照し、かつ大域オフセットテーブルを作成するようにリンカーに指示します。

R_SPARC_GOT13

この再配置型は R_SPARC_13 に似ています。ただし、次の点が異なります。つまり、この再配置型はシンボルの大域オフセットテーブルエントリのアドレスを参照し、かつ大域オフセットテーブルを作成するようにリンカーに指示します。

R_SPARC_GOT22

この再配置型は R_SPARC_22 に似ています。ただし、次の点が異なります。つまり、この再配置型はシンボルの大域オフセットテーブルエントリのアドレスを参照し、かつ大域オフセットテーブルを作成するようにリンカーに指示します。

R_SPARC_WPLT30

この再配置型は R_SPARC_WDISP30 に類似しています。ただし、次の点が異なります。つまり、この再配置型はシンボルのプロシージャのリンクテーブルエントリのアドレスを参照し、かつプロシージャのリンクテーブルを作成するようにリンカーに指示します。

R_SPARC_COPY

リンカーは、動的リンクを行うためにこの再配置型を使用します。この再配置型のオフセット構成要素は、書き込み可能セグメントの位置を参照します。シンボルテーブルインデックスは、現オブジェクトファイルと共有オブジェクトの両方に存在する必要があるシンボルを指定します。実行時、実行時リンカーは共有オブジェクトのシンボルに関連付けられているデータを、オフセットで指定されている位置にコピーします。詳細は、131ページの「コピー再配置」を参照してください。

R_SPARC_GLOB_DAT

この再配置型は R_SPARC_32 に似ています。ただし、次の点が異なります。つまり、この再配置型は大域オフセットテーブルエントリを、指定されたシンボルのアドレスに設定します。この特殊な再配置型を使うと、シンボルと大域オフセットテーブルエントリの対応付けを判定できます。

R_SPARC_JMP_SLOT

リンカーは、動的リンクを行うためにこの再配置型を使用します。この再配置型のオフセット構成要素は、プロシージャのリンクテーブルエントリの位置を与えます。実行時リンカーは、プロシージャのリンクテーブルエントリを変更して指定シンボルアドレスに制御を渡します。

R_SPARC_RELATIVE

リンカーは、動的リンクを行うためにこの再配置型を使用します。この再配置型のオフセット構成要素は、相対アドレスを表す値が存在する、共有オブジェクト内の位置を与えます。実行時リンカーは共有オブジェクトが読み込まれる仮想アドレスを相対アドレスに加算することで、対応する仮想アドレスを計算します。この型に対する再配置エントリは、シンボルテーブルインデックスに対して0を指定しなければなりません。

R_SPARC_UA32

この再配置型は R_SPARC_32 に似ていますが、整列されていないワードを参照する点が異なります。つまり、再配置されるワードは、任意整列が存在する4つの別個のバイトとして処理されなければなりません(アーキテクチャの要求に従って整列されるワードとしては処理されません)。

SPARC: 再配置型

注 - 表 7-29 に示されているフィールド名は、再配置型がオーバーフローを検査するかどうかを通知します。計算される再配置値は意図したフィールドより大きい場合があり、再配置型によっては値の適合を検証 (V) したり結果を切り捨てたり (T) することがあります。たとえば、V-simm13 は、計算された値が simm13 フィールドの外部に0以外の有意ビットを持つことがないことを意味します。

表 7-29 SPARC: 再配置型

名前	値	フィールド	計算
R_SPARC_NONE	0	なし	なし
R_SPARC_8	1	V-byte8	S + A

表 7-29 SPARC: 再配置型 続く

名前	値	フィールド	計算
R_SPARC_16	2	V-half16	S + A
R_SPARC_32	3	V-word32	S + A
R_SPARC_DISP8	4	V-byte8	S + A - P
R_SPARC_DISP16	5	V-half16	S + A - P
R_SPARC_DISP32	6	V-disp32	S + A - P
R_SPARC_WDISP30	7	V-disp30	(S + A - P) >> 2
R_SPARC_WDISP22	8	V-disp22	(S + A - P) >> 2
R_SPARC_HI22	9	V-imm22	(S + A) >> 10
R_SPARC_22	10	V-imm22	S + A
R_SPARC_13	11	V-simm13	S + A
R_SPARC_LO10	12	T-simm13	(S + A) & 0x3ff
R_SPARC_GOT10	13	T-simm13	G & 0x3ff
R_SPARC_GOT13	14	V-simm13	G
R_SPARC_GOT22	15	T-simm22	G >> 10
R_SPARC_PC10	16	T-simm13	(S + A - P) & 0x3ff
R_SPARC_PC22	17	V-disp22	(S + A - P) >> 10
R_SPARC_WPLT30	18	V-disp30	(L + A - P) >> 2
R_SPARC_COPY	19	なし	なし
R_SPARC_GLOB_DAT	20	V-xword64	S + A

表 7-29 SPARC: 再配置型 続く

名前	値	フィールド	計算
R_SPARC_JMP_SLOT	21	なし	「R_SPARC_JMP_SLOT」を参照
R_SPARC_RELATIVE	22	V-xword64	$B + A$
R_SPARC_UA32	23	V-word32	$S + A$
R_SPARC_PLT32	24	V-word32	$L + A$
R_SPARC_HIPLT22	25	T-imm22	$(L + A) \gg 10$
R_SPARC_LOPLT10	26	T-simm13	$(L + A) \& 0x3ff$
R_SPARC_PCPLT32	27	V-disp32	$L + A - P$
R_SPARC_PCPLT22	28	V-disp22	$(L + A - P) \gg 10$
R_SPARC_PCPLT10	29	V-simm13	$(L + A - P) \& 0x3ff$
R_SPARC_10	30	V-simm10	$S + A$
R_SPARC_11	31	V-simm11	$S + A$
R_SPARC_64	32	V-xword64	$S + A$
R_SPARC_OLO10	33	V-simm13	$((S + A) \& 0x3ff) + 0$
R_SPARC_HH22	34	V-imm22	$(S + A) \gg 42$
R_SPARC_HM10	35	T-simm13	$((S + A) \gg 32) \& 0x3ff$
R_SPARC_LM22	36	T-imm22	$(S + A) \gg 10$
R_SPARC_PC_HH22	37	V-imm22	$(S + A - P) \gg 42$
R_SPARC_PC_HM10	38	T-simm13	$((S + A - P) \gg 32) \& 0x3ff$

表 7-29 SPARC: 再配置型 続く

名前	値	フィールド	計算
R_SPARC_PC_LM22	39	T-imm22	(S + A - P) >> 10
R_SPARC_WDISP16	40	V-d2/disp14	(S + A - P) >> 2
R_SPARC_WDISP19	41	V-disp19	(S + A - P) >> 2
R_SPARC_7	43	V-imm7	(S + A) & 0x7f
R_SPARC_5	44	V-imm5	(S + A) & 0x1f
R_SPARC_6	45	V-imm6	(S + A) & 0x3f
R_SPARC_DISP64	46	V-xword64	S + A - P
R_SPARC_PLT64	47	V-xword64	L + A
R_SPARC_HIX22	48	V-imm22	((S + A) ^ 0xffffffffffffffff) >> 10
R_SPARC_LOX10	49	T-simm13	((S + A) & 0x3ff) 0x1c00
R_SPARC_H44	50	V-imm22	(S + A) >> 22
R_SPARC_M44	51	T-imm10	((S + A) >> 12) & 0x3ff
R_SPARC_L44	52	T-imm13	(S + A) & 0xfff
R_SPARC_REGISTER	53	V-xword64	S + A
R_SPARC_UA64	54	V-xword64	S + A
R_SPARC_UA16	55	V-half16	S + A

いくつかの再配置型には、単純な計算を超えた意味が存在します。

R_SPARC_GOT10

この再配置型は R_SPARC_LO10 に似ています。ただし、次の点が異なります。つまり、この再配置型はシンボルの大域オフセットテーブルエントリのアドレスを参照し、かつ大域オフセットテーブルを作成するようにリンカーに指示します。

R_SPARC_GOT13

この再配置型は R_SPARC_13 に似ています。ただし、次の点が異なります。つまり、この再配置型はシンボルの大域オフセットテーブルエントリのアドレスを参照し、かつ大域オフセットテーブルを作成するようにリンカーに指示します。

R_SPARC_GOT22

この再配置型は R_SPARC_22 に似ています。ただし、次の点が異なります。つまり、この再配置型はシンボルの大域オフセットテーブルエントリのアドレスを参照し、かつ大域オフセットテーブルを作成するようにリンカーに指示します。

R_SPARC_WPLT30

この再配置型は R_SPARC_WDISP30 に似ています。ただし、次の点が異なります。つまり、この再配置型はシンボルのプロシージャのリンクテーブルエントリのアドレスを参照し、かつプロシージャのリンクテーブルを作成するようにリンカーに指示します。

R_SPARC_COPY

リンカーは、動的リンクを行うためにこの再配置型を使用します。この再配置型のオフセット構成要素は、書き込み可能セグメントの位置を参照します。シンボルテーブルインデックスは、現オブジェクトファイルと共有オブジェクトの両方に存在する必要があるシンボルを指定します。実行時、実行時リンカーは共有オブジェクトのシンボルに関連付けられているデータを、オフセットで指定されている位置にコピーします。詳細は、131ページの「コピー再配置」を参照してください。

R_SPARC_GLOB_DAT

この再配置型は R_SPARC_64 に似ています。ただし、次の点が異なります。つまり、この再配置型は大域オフセットテーブルエントリを、指定されたシンボルのアドレスに設定します。この特殊な再配置型を使うと、シンボルと大域オフセットテーブルエントリの対応付けを判定できます。

R_SPARC_JMP_SLOT

リンカーは、動的リンクを行うためにこの再配置型を作成します。この再配置型のオフセット構成要素は、プロシージャのリンクテーブルエントリの位置を与えます。実行時リンカーは、プロシージャのリンクテーブルエントリを変更して指定シンボルアドレスに制御を渡します。

R_SPARC_RELATIVE

リンカーは、動的リンクを行うためにこの再配置型を使用します。この再配置型のオフセット構成要素は、相対アドレスを表す値が存在する、共有オブジェクト内の位置を与えます。実行時リンカーは共有オブジェクトが読み込まれる仮想アドレスを相対アドレスに加算することで、対応する仮想アドレスを計算します。この型に対する再配置エントリは、シンボルテーブルインデックスに対して0を指定しなければなりません。

R_SPARC_UA32

この再配置型は R_SPARC_32 に似ていますが、整列されていないワードを参照する点が異なります。つまり、再配置されるワードは、任意整列が存在する4つの別個のバイトとして処理されなければなりません(アーキテクチャの要求に従って整列されるワードとしては処理されません)。

R_SPARC_OLO10

この再配置型は R_SPARC_LO10 に似ていますが、符号付き13ビット即値フィールドを十分に使用するために余分なオフセットが追加される点が異なります。

R_SPARC_HH22

アセンブラは、"imm22-instruction ... %hh (absolute) ..." という形式の命令を認識すると、この再配置型を使用します。

R_SPARC_HM10

アセンブラは、"simm13-instruction ... %hm (absolute) ..." という形式の命令を認識すると、この再配置型を生成します。

R_SPARC_LM22

アセンブラは、"imm22-instruction ... %lm (absolute) ..." という形式の命令を認識すると、この再配置型を使用します。この再配置型は R_SPARC_HI22 に似ていますが、妥当性検査ではなく切り捨てを行う点が異なります。

R_SPARC_PC_HH22

アセンブラは、"imm22-instruction ... %hh (pc-relative) ..." という形式の命令を認識すると、この再配置型を使用します。

R_SPARC_PC_HM10

アセンブラは、"simm13-instruction ... %hm (pc-relative) ..." という形式の命令を認識すると、この再配置型を生成します。

R_SPARC_PC_LM22

アセンブラは、"imm22-instruction ... %lm (pc-relative) ..." という形式の命令を認識すると、この再配置型を使用します。この再配置型は R_SPARC_PC22 に似ていますが、妥当性検査ではなく切り捨てを行う点が異なります。

R_SPARC_7

アセンブラは7ビットソフトウェアトラップ番号に対してこの再配置型を使用します。

R_SPARC_HIX22

この再配置型は、64ビットアドレス空間の最上位4GBに限定される実行可能ファイルに対して R_SPARC_LOX10 と共に使用されます。R_SPARC_HI22 に似ていますが、リンク値の1の補数を与えます。

R_SPARC_LOX10

R_SPARC_HIX22 と共に使用されます。R_SPARC_LO10 に似ていますが、必ずリンク値のビット10からビット12までを設定します。

R_SPARC_H44

アセンブラは、"imm44-instruction ... %h44 (absolute) ..." という形式の命令を認識すると、この再配置型を使用します。

R_SPARC_M44

アセンブラは、"imm44-instruction ... %m44 (absolute) ..." という形式の命令を認識すると、この再配置型を生成します。

R_SPARC_L44

この再配置型は再配置型 R_SPARC_H44 および R_SPARC_M44 と共に使用され、44 ビット絶対アドレス指定モデルを生成します。アセンブラは、"imm44-instruction ... %l44 (absolute) ..." という形式の命令を認識すると、この再配置型を生成します。

R_SPARC_REGISTER

この再配置型は、レジスタシンボルを初期化するために使用されます。この再配置型のオフセット構成要素には、初期化されるレジスタ番号が存在します。SHN_ABS 型のこのレジスタには、対応するレジスタシンボルが存在しなければなりません。

IA: 再配置型

表 7-30 IA: 再配置型

名前	値	フィールド	計算
R_386_NONE	0	なし	なし
R_386_32	1	word32	S + A
R_386_PC32	2	word32	S + A - P
R_386_GOT32	3	word32	G + A
R_386_PLT32	4	word32	L + A - P
R_386_COPY	5	なし	なし
R_386_GLOB_DAT	6	word32	S
R_386_JMP_SLOT	7	word32	S

表 7-30 IA: 再配置型 続く

名前	値	フィールド	計算
R_386_RELATIVE	8	word32	B + A
R_386_GOTOFF	9	word32	S + A - GOT
R_386_GOTPC	10	word32	GOT + A - P
R_386_32PLT	11	word32	L + A

いくつかの再配置型には、単純な計算を超えた意味が存在します。

R_386_GOT32

この再配置型は、大域オフセットテーブルのベースからシンボルの大域オフセットテーブルエントリまでの距離を計算します。この再配置型はまた、大域オフセットテーブルを作成するようにリンカーに指示します。

R_386_PLT32

この再配置型はシンボルのプロシージャのリンクテーブルエントリのアドレスを計算し、かつプロシージャのリンクテーブルを作成するようにリンカーに指示します。

R_386_COPY

リンカーは、動的リンクを行うためにこの再配置型を使用します。この再配置型のオフセット構成要素は、書き込み可能セグメントの位置を参照します。シンボルテーブルインデックスは、現オブジェクトファイルと共有オブジェクトの両方に存在する必要があるシンボルを指定します。実行時、実行時リンカーは共有オブジェクトのシンボルに関連付けられているデータを、オフセットで指定されている位置にコピーします。詳細は、131ページの「コピー再配置」を参照してください。

R_386_GLOB_DAT

この再配置型は大域オフセットテーブルエントリを、指定されたシンボルのアドレスに設定します。この特殊な再配置型を使うと、シンボルと大域オフセットテーブルエントリの対応付けを判定できます。

R_386_JMP_SLOT

リンカーは、動的リンクを行うためにこの再配置型を使用します。この再配置型のオフセット構成要素は、プロシージャのリンクテーブルエントリの位置を与えます。実行時リンカーは、プロシージャのリンクテーブルエントリを変更して指定シンボルアドレスに制御を渡します。

R_386_RELATIVE

リンカーは、動的リンクを行うためにこの再配置型を使用します。この再配置型のオフセット構成要素は、相対アドレスを表す値が存在する、共有オブジェクト内の位置を与えます。実行時リンカーは共有オブジェクトが読み込まれる仮想アドレスに相対アドレスを加算することで、対応する仮想アドレスを計算します。この型に対する再配置エントリは、シンボルテーブルインデックスに対して0を指定しなければなりません。

R_386_GOTOFF

この再配置型は、シンボルの値と大域オフセットテーブルのアドレスの差を計算します。この再配置型はまた、大域オフセットテーブルを作成するようにリンカーに指示します。

R_386_GOTPC

この再配置型は R_386_PC32 に似ていますが、計算を行う際に大域オフセットテーブルのアドレスを使用する点が異なります。この再配置で参照されるシンボルは、通常 `_GLOBAL_OFFSET_TABLE_` です。この再配置型はまた、大域オフセットテーブルを作成するようにリンカーに指示します。

Comdat セクション

この種類のセクションは、セクション名 (sh_name) で一意に示されます。リンカーが、同じセクション名の SHT_SUNW_COMDAT 型の複数のセクションと出会うと、最初のセクションが保持され、他のすべてのセクションは捨てられます。また、捨てられた SHT_SUNW_COMDAT セクションに適用されたすべての再配置は無視され、かつ捨てられたセクションで定義されたすべてのシンボルも保持されません。

また、リンカーはセクション命名規約をサポートします。このセクション命名規約は、コンパイラが `-xF` オプションで呼び出されると、セクションの再順序付け (327ページの「セグメントの宣言」を参照) で使用されます。つまり、セクション

が、.<funcname>%<sectname> という名前のセクションに入れられると、保持された最後の SHT_SUNW_COMDAT セクションが、.<sectname> で示されるセクションに合体します。この方法を使用すると、SHT_SUNW_COMDAT セクションを最終的に .text、.data、または他のセクションに入れることができます。

バージョン情報

リンカーで作成されるオブジェクトには、2つの型のバージョン情報が存在できません。

- 「バージョン定義」は大域シンボルとの関連付けを与え、SHT_SUNW_verdef および SHT_SUNW_versym 型のセクションを使用して実現される
- 「バージョン依存性」は他のオブジェクト依存性からのバージョン定義要求を示し、SHT_SUNW_verneed 型のセクションを使用して実現される

これらのセクションを形成する構造体は、sys/link.h で定義されます。バージョン情報が存在するセクションには、.SUNW_version という名前が付けられます。

バージョン定義セクション

このセクションは、SHT_SUNW_verdef 型で定義されます。このセクションが存在する場合、SHT_SUNW_versym セクションも存在しなければなりません。これら2つの構造体を使用することで、シンボルとバージョン定義の関連付けがファイル内で維持されます(詳細は、142ページの「バージョン定義の作成」を参照)。このセクションの要素の構造体は、以下のとおりです。

```
typedef struct {
    Elf32_Half    vd_version;
    Elf32_Half    vd_flags;
    Elf32_Half    vd_ndx;
    Elf32_Half    vd_cnt;
    Elf32_Word    vd_hash;
    Elf32_Word    vd_aux;
    Elf32_Word    vd_next;
} Elf32_Verdef;

typedef struct {
    Elf32_Word    vda_name;
    Elf32_Word    vda_next;
} Elf32_Verdaux;

typedef struct {
    Elf64_Half    vd_version;
```

(続く)

```

        Elf64_Half    vd_flags;
        Elf64_Half    vd_ndx;
        Elf64_Half    vd_cnt;
        Elf64_Word    vd_hash;
        Elf64_Word    vd_aux;
        Elf64_Word    vd_next;
    } Elf64_Verdef;

typedef struct {
        Elf64_Word    vda_name;
        Elf64_Word    vda_next;
    } Elf64_Verdaux;

```

vd_version

この構成要素は、構造体自身のバージョンを示します。

表 7-31 バージョン定義構造のバージョン

名前	値	意味
VER_DEF_NONE	0	無効バージョン
VER_DEF_CURRENT	>=1	現在のバージョン

値 1 は最初のセクション形式を示し、拡張した場合は番号を大きくします。VER_DEF_CURRENT の値は、現在のバージョン番号を示すために必要に応じて変化します。

vd_flags

この構成要素は、バージョン定義に固有の情報を保持します。

表 7-32 バージョン定義セクションのフラグ

名前	値	意味
VER_FLG_BASE	0x1	ファイル自身のバージョン定義
VER_FLG_WEAK	0x2	ウィークバージョン識別子

ベースのバージョン定義は、バージョン定義またはシンボルの自動短縮簡約がファイルに適用されている場合、必ず存在します。ベースバージョンは、ファイルの予約されたシンボルに対してデフォルトのバージョンを与えます (61ページの「出力イメージの生成」を参照)。ウィークバージョン定義には、関連付けられているシンボルは存在しません (詳細は、145ページの「ウィークバージョン定義の作成」を参照)。

vd_ndx

この構成要素は、バージョンインデックスを保持します。各バージョン定義には、SHT_SUNW_versym エントリを適切なバージョン定義に関連付ける一意のインデックスが存在します。

vd_cnt

この構成要素は、Elf32_Verdaux 配列の要素数を示します。

vd_hash

この構成要素は、バージョン定義名の値を保持します (この値は、323ページの「ハッシュテーブル」に記述されているハッシング機能で生成されます)。

vd_aux

この構成要素は、この Elf32_Verdef エントリの手元からバージョン定義名の Elf32_Verdaux 配列までのバイトオフセットを保持します。配列の手元要素は存在しなければならず、この構造体が定義するバージョン定義文字列を指し示します。追加要素は存在することができ、また番号は vd_cnt 値で示されます。これらの要素は、このバージョン定義の依存関係を表します。これらの依存関係の各々は、独自のバージョン定義構造体を持っています。

vd_next

この構成要素は、この Elf32_Verdef 構造体の先頭から次の Elf32_Verdef エントリまでのバイトオフセットを保持します。

vda_name

この構成要素は、空文字で終わっている文字列への文字列テーブルオフセットを保持し、バージョン定義名を与えます。

vda_next

この構成要素は、この Elf32_Verdaux エントリ先頭から次の Elf32_Verdaux エントリまでのバイトオフセットを保持します。

バージョンシンボルセクション

このセクションは SHT_SUNW_versym 型で定義されており、以下の構造を持つ要素配列からなります。

```
typedef Elf32_Half    Elf32_Versym;  
typedef Elf64_Half    Elf64_Versym;
```

配列の要素数は、関連付けられているシンボルテーブルに存在するシンボルテーブルエントリ数 (セクション sh_link 値で決定される) に等しくなければなりません。配列の各要素には 1 つのインデックスが存在し、このインデックスは表 7-33 に示す値をとることができます。

表 7-33 バージョン依存インデックス

名前	値	意味
VER_NDX_LOCAL	0	シンボルにローカル適用範囲が存在する。
VER_NDX_GLOBAL	1	シンボルに大域適用範囲 (ベースバージョン定義に割り当てられる) が存在する
	>1	シンボルに大域適用範囲 (ユーザー定義バージョン定義に割り当てられる) が存在する

VER_NDX_GLOBAL より大きいインデックス値は、SHT_SUNW_verdef セクションのエントリの vd_ndx 値に一致しなければなりません。VER_NDX_GLOBAL より大きいインデックス値が存在しない場合、SHT_SUNW_verdef セクションが存在する必要はありません。

バージョン依存セクション

このセクションは、SHT_SUNW_verneed 型で定義されます。このセクションは、ファイルの動的依存性から要求されるバージョン定義を示すことで、ファイルの動的依存性要求を補足します。依存性にバージョン定義が存在する場合のみ、記録がこのセクションにおいて行われます。このセクションの要素の構造体は、次のとおりです。

```
typedef struct {
    Elf32_Half    vn_version;
    Elf32_Half    vn_cnt;
    Elf32_Word    vn_file;
    Elf32_Word    vn_aux;
    Elf32_Word    vn_next;
} Elf32_Verneed;

typedef struct {
    Elf32_Word    vna_hash;
    Elf32_Half    vna_flags;
    Elf32_Half    vna_other;
    Elf32_Word    vna_name;
    Elf32_Word    vna_next;
} Elf32_Vernaux;

typedef struct {
    Elf64_Half    vn_version;
    Elf64_Half    vn_cnt;
    Elf64_Word    vn_file;
    Elf64_Word    vn_aux;
    Elf64_Word    vn_next;
} Elf64_Verneed;

typedef struct {
    Elf64_Word    vna_hash;
    Elf64_Half    vna_flags;
    Elf64_Half    vna_other;
    Elf64_Word    vna_name;
    Elf64_Word    vna_next;
} Elf64_Vernaux;
```

vn_version

この構成要素は、構造体自身のバージョンを示します。

表 7-34 バージョン依存構造体のバージョン

名前	値	意味
VER_NEED_NONE	0	無効バージョン
VER_NEED_CURRENT	>=1	現在のバージョン

値 1 は最初のセクション形式を示し、拡張した場合は番号を大きくします。VER_NEED_CURRENT の値は、現在のバージョン番号を示すために必要に応じて変化します。

vn_cnt

この構成要素は、Elf32_Verneed 配列の要素数を示します。

vn_file

この構成要素は、空文字で終わっている文字列への文字列テーブルオフセットを保持し、バージョン依存性が存在するファイル名を与えます。この名前は、ファイルに存在する .dynamic 依存性 (296ページの「動的セクション」を参照) のどれかに一致します。

vn_aux

この構成要素は、この Elf32_Verneed エントリの先頭から、関連付けられているファイル依存性から要求されるバージョン定義の Elf32_Verneed 配列までのバイトオフセットを保持します。少なくとも 1 つのバージョン依存性が存在しなければなりません。追加バージョン依存性は存在することができ、また番号は vn_cnt 値で示されます。

vn_next

この構成要素は、この Elf32_Verneed エントリの先頭から次の Elf32_Verneed エントリまでのバイトオフセットを保持します。

vna_hash

この構成要素は、バージョン依存性の名前のハッシュ値を保持します (この値は、323ページの「ハッシュテーブル」に記述されているハッシュ関数で生成されます)。

vna_flags

この構成要素は、バージョン依存性に固有の情報を保持します。

表 7-35 バージョン依存構造のフラグ

名前	値	意味
VER_FLG_WEAK	0x2	ウィークバージョン識別子

ウィークバージョン依存性は、ウィークバージョン定義への最初の結び付きを示します。詳細は、142ページの「バージョン定義の作成」を参照してください。

vna_other

この構成要素は現在、使用されていません。

vna_name

この構成要素は、空文字で終わっている文字列への文字列テーブルオフセットを保持し、バージョン依存性の名前を与えます。

vna_next

この構成要素は、この Elf32_Vernaux エントリの先頭から次の Elf32_Vernaux エントリまでのバイトオフセットを保持します。

注釈セクション

ソフトウェアを開発して販売する場合、オブジェクトファイルに特別な情報を付加して、他のプログラムから準拠性や互換性などを確認できるようにしたいことがあります。SHT_NOTE 型のセクションと PT_NOTE 型のプログラムヘッダー要素は、この目的に対して使用できます。

セクションとプログラムヘッダー要素での注釈情報は任意の数のエントリを保持します。64 ビットおよび 32 ビットのオブジェクトについては、各エントリはターゲットプロセッサの形式になっている 4 バイトワードの配列です。注釈情報の構造についての説明を容易にするためにラベルを図 7-6 に示します。ただし、ラベルは仕様の一部ではありません。

namesz
descsz
type
name ...
desc ...

図 7-5 注釈の情報

namesz と name

name の先頭 namesz バイトには、エントリの所有者または作者を示す、空文字で終わっている文字列が存在します。名前の競合を回避するための正式な機構は存在しません。慣例では、ベンダーは識別子として自身の名前 (“XYZ Computer Company” など) を使用します。name が存在しない場合、namesz は 0 になります。name の領域は、パッドを使用して、4 バイトに整列します。必要であれば namesz は、パッドの長さを含みません。

descsz と desc

desc の先頭 descsz バイトは、注釈記述を保持します。注釈の記述が存在しない場合、descsz は 0 になります。desc の領域は、必要であればパッドを使用して、4 バイトに整列します。descsz はパッドの長さを含みません。

type

このワードは注釈の記述のセマンティクスを示します。各エントリの作者は、自分で種類を管理しますので、1 つの type 値に関して複数の解釈が存在することがあります。したがって、注釈の記述を認識するには、name と type の両方を認識しなければなりません。type は現在、負でない値でなければなりません。

たとえば、以下の注釈のセグメントは 2 つのエントリを保持しています。

	+0	+1	+2	+3
namesz	7			
descsz	0			
type	1			
name	X	Y	Z	
	C	o	\0	pad
namesz	7			
descsz	8			
type	3			
name	X	Y	Z	
	C	o	\0	pad
desc	word0			
	word1			

記述なし

図 7-6 注釈セグメントの例

注・システムは注釈情報(名前が存在せず (namesz == 0)、名前の長さが 0 (name[0] == '\0') を予約していますが、現在 type を定義していません。他のすべての名前には、少なくとも 1 つの空ではない文字が存在しなければなりません。

移動セクション

一般に、ELF ファイル内では、初期設定されたデータ変数はオブジェクトファイル内で維持されます。データ変数が非常に大きく、初期設定された(ゼロ以外の)要素が少数の場合でも、変数全体はやはりオブジェクトファイルで維持されます。

サイズの大きな部分的に初期設定されたデータ変数を含むオブジェクト (FORTRAN COMMON ブロックなどのような) は、多大なディスクスペースオーバーヘッドをもたらすことがあります。SHT_SUNW_move セクションは、こういったデータ変数を圧縮して関連オブジェクトのディスクサイズを減らすメカニズムを提供します。

SHT_SUNW_move セクションは、ELF32_Move または Elf64_Move 型の複数のエントリーを含みます。これらのエントリーはデータ変数を一時的項目 (.bss) として定義することが可能で、そのためオブジェクトファイル内にスペースを占めることなく、

実行時にオブジェクトのメモリーイメージに反映させることができます。移動レコードは、完全なデータ変数を構成するためにデータについてメモリーイメージがどのように初期設定されるかを確立します。

ELF32_Move および Elf64_Move エントリは次のように定義されます。

```
typedef struct {
    Elf32_Lword    m_value;
    Elf32_Word     m_info;
    Elf32_Word     m_poffset;
    Elf32_Half     m_repeat;
    Elf32_Half     m_stride;
} Elf32_Move;

#define ELF32_M_SYM(info)      ((info)>>8)
#define ELF32_M_SIZE(info)    ((unsigned char)(info))
#define ELF32_M_INFO(sym, size) ((sym)<<8)+(unsigned char)(size)

typedef struct {
    Elf64_Lword    m_value;
    Elf64_Xword    m_info;
    Elf64_Xword    m_poffset;
    Elf64_Half     m_repeat;
    Elf64_Half     m_stride;
} Elf64_Move;

#define ELF64_M_SYM(info)      ((info)>>8)
#define ELF64_M_SIZE(info)    ((unsigned char)(info))
#define ELF64_M_INFO(sym, size) ((sym)<<8)+(unsigned char)(size)
```

m_value

この構成要素は、初期設定値 (メモリーイメージへ移動される値) を提供します。

m_info

この構成要素は、初期設定が適用されるものに関するシンボルテーブルインデックスと、初期設定されるオフセットのサイズ (単位: バイト) の両方を提供します。構成要素の下位 8 ビットはサイズを定義し、1、2、4、または 8 になります。上位バイトはシンボルインデックスを定義します。

m_poffset

この構成要素は、初期設定が適用される、関連シンボルからの相対オフセットです。

m_repeat

この構成要素は、繰り返し回数です。

m_stride

この構成要素は、スキップの数を提供します。この値は、繰り返し初期設定を実行する際にスキップするユニットの数を示します。1ユニットはm_infoで定義された初期設定オブジェクトのサイズです。m_strideの値が0の場合、初期設定をm_repeatユニット連続して行うことを示します。

次のデータ定義は、オブジェクトファイル内で通常0x8000バイトを消費するものです。

```
typedef struct {
    int    one;
    char   two;
} Data

Data move[0x1000] = {
    {0, 0},
    {1, '1'},
    {0, 0},
    {0xf, 'F'}, {0xf, 'F'},
    {0, 0},
    {0xe, 'E'},
    {0, 0},
    {0xe, 'E'}
};
```

SHT_SUNW_move セクションを使用して、データ項目を .bss セクションへ移動し、関連する移動エンタリで初期設定することができます。

```
% elfdump -s data | fgrep move
[17] 0x00020868 0x00008000 OBJT GLOB 0 .bss move
% elfdump -m data
Move Section: .SUNW_move
  offset  ndx    size  repeat  stride  value  with respect to
  0x8     0x17  4     1       0       0x1    foo
  0xc     0x17  1     1       0       0x31   foo
  0x18    0x17  4     2       2       0xf    foo
  0x1c    0x17  1     2       8       0x46   foo
  0x28    0x17  4     2       4       0xe    foo
  0x2c    0x17  1     2       16      0x45   foo
```

動的リンク

このセクションは、オブジェクトファイル情報と、プログラムの実行イメージを作成するシステム動作を記述します。ここに記述されているいくつかの情報はすべてのシステムに適用され、プロセッサに固有の情報はその旨が示されたセクションに存在します。

実行可能オブジェクトファイルと共有オブジェクトファイルは、プログラムを静的に表現します。このようなプログラムを実行するためには、システムはこれらのファイルを使用して動的なプログラムの表現、すなわちプロセスイメージを作成します。プロセスイメージには、テキスト、データ、スタックなどが存在するセグメントが存在します。この項の主な細目は以下のとおりです。

- 279ページの「プログラムヘッダー」では、プログラム実行に直接関係するオブジェクトファイルの構造を記述する。重要なデータ構造体であるプログラムヘッダーテーブルは、ファイル内のセグメントイメージの位置を示す。また、このプログラムヘッダーテーブルは、プログラムのメモリーイメージの作成に必要な他の情報が存在する
- 287ページの「プログラムの読み込み(プロセッサ固有)」では、メモリーにプログラムを読み込むために使用する情報を記述する
- 294ページの「実行時リンカー」では、プロセスイメージのオブジェクトファイル間でシンボル参照を指定、解決するために使用する情報を記述する

プログラムヘッダー

実行可能オブジェクトファイルまたは共有オブジェクトファイルのプログラムヘッダーテーブルは、構造体の配列です(各構造体は、実行されるプログラムを作成するためにシステムが必要とするセグメントやその他の情報を記述します)。各オブジェクトファイルセグメントには、286ページの「セグメントの内容」に記述されているとおり、1つまたは複数のセクションが存在します。

プログラムヘッダーは、実行可能オブジェクトファイルと共有オブジェクトファイルに対してのみ意味があります。プログラムヘッダーサイズは、ELFヘッダーの `e_phentsize` 構成要素と `e_phnum` 構成要素で指定されます。詳細は、203ページの「ELFヘッダー」を参照してください。

プログラムヘッダーの構造体 (`sys/elf.h` で定義されている) は、次のとおりです。

```

typedef struct {
    Elf32_Word      p_type;
    Elf32_Off       p_offset;
    Elf32_Addr      p_vaddr;
    Elf32_Addr      p_paddr;
    Elf32_Word      p_filesz;
    Elf32_Word      p_memsz;
    Elf32_Word      p_flags;
    Elf32_Word      p_align;
} Elf32_Phdr;

typedef struct {
    Elf64_Word      p_type;
    Elf64_Word      p_flags;
    Elf64_Off       p_offset;
    Elf64_Addr      p_vaddr;
    Elf64_Addr      p_paddr;
    Elf64_Xword     p_filesz;
    Elf64_Xword     p_memsz;
    Elf64_Xword     p_align;
} Elf64_Phdr;

```

p_type

この構成要素は、この配列要素が記述するセグメント型、または配列要素の情報のセマンティクスを与えます。型の値とその意味は、表 7-36 を参照してください。

p_offset

この構成要素は、ファイルの先頭から、セグメントの先頭バイトが存在する位置までのオフセットを与えます。

p_vaddr

この構成要素は、セグメントの先頭バイトが存在するメモリーの仮想アドレスを与えます。

p_paddr

物理アドレス指定が妥当であるシステムの場合、この構成要素はセグメントの物理アドレスに対して使用されます。本システムはアプリケーションプログラムに対して物理アドレス指定を無視するので、この構成要素には実行可能ファイルと共有オブジェクトに対する指定されていない内容が存在します。

p_filesz

この構成要素は、セグメントのファイルイメージのバイト数を与えます。この構成要素は0の場合もあります。

p_memsz

この構成要素は、セグメントのメモリーイメージのバイト数を与えます。この構成要素は0の場合もあります。

p_flags

この構成要素は、セグメントに関するフラグを与えます。定義されているフラグ値は、表 7-37 のとおりです。

p_align

287ページの「プログラムの読み込み (プロセッサ固有)」に記述されているとおり、読み込み可能なプロセスセグメントには、ページサイズを法として `p_vaddr` と `p_offset` に対して同じ値が存在しなければなりません。この構成要素は、セグメントがメモリーとファイルにおいて整列される値を与えます。値 0 と 1 は、整列が必要ないことを意味します。その他の値の場合、`p_align` は 2 の正整数累乗でなければならない、また `p_vaddr` は `p_align` を法として `p_offset` に等しくなければなりません。

いくつかのエントリはプロセスセグメントを記述しますが、それ以外のエントリは補足情報を与え、プロセスイメージには関与しません。セグメントエントリが現れる順序は、以下に明確に記されている場合を除き任意です。定義されている型の値は、次のとおりです。他の値は、将来における使用に備えて保留されます。

表 7-36 セグメント型 p_type

名前	値
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3

表 7-36 セグメント型 p_type 続く

名前	値
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_LOSUNW	0x6fffffff
PT_SUNWBSS	0x6fffffff
PT_SUNWSTACK	0x6fffffff
PT_HISUNW	0x6fffffff
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

PT_NULL

この配列要素は使用されていません。他の構成要素の値は不定です。この型を使用すると、プログラムヘッダーテーブルに、無視されるエントリを入れることができます。

PT_LOAD

この配列要素は、`p_filesz` と `p_memsz` で記述される読み込み可能セグメントを指定します。ファイルのバイト列は、メモリーセグメントの先頭に対応付けされます。セグメントのメモリーサイズ (`p_memsz`) がファイルサイズ (`p_filesz`) より大きい場合、不足するバイトは、値 0 を保持しセグメントの初期化領域に続くように定義されます。ファイルサイズがメモリーサイズより大きくなることは許可されません。プログラムヘッダーテーブルの読み込み可能セグメントエントリは、`p_vaddr` 構成要素で整列され、昇順に現れます。

PT_DYNAMIC

この配列要素は、動的リンク情報を指定します。詳細は、296ページの「動的セクション」を参照してください。

PT_INTERP

この配列要素は、インタプリタとして呼び出される、空文字で終了しているパス名の位置とサイズを指定します。このセグメント型は、実行可能ファイルに対してのみ意味があります(ただし、共有オブジェクトに対しても指定されることがあります)。このセグメント型は、ファイル内で複数存在することはできません。このセグメントタイプが存在する場合、このセグメント型は読み込み可能セグメントエントリの前に存在しなければなりません。詳細は、294ページの「プログラムインタプリタ」を参照してください。

PT_NOTE

この配列要素は、補助情報の位置とサイズを指定します。詳細は、274ページの「注釈セクション」を参照してください。

PT_SHLIB

このセグメント型は予約されますが、解釈の方法は定義されていません。

PT_PHDR

この配列要素が存在する場合、この配列要素は、プログラムヘッダーテーブル自身の、ファイルとプログラムのメモリーイメージにおける位置とサイズを指定します。このセグメント型は、ファイル内で複数存在することはできません。また、このセグメント型は、プログラムヘッダーテーブルがプログラムのメモリーイメージの一部になる場合に限り存在できます。このセグメント型が存在する場合、このセグメント型は読み込み可能セグメントエントリの前に存在しなければなりません。詳細は、294ページの「プログラムインタプリタ」を参照してください。

PT_LOSUNW - PT_HISUNW

この範囲の値は、Sun 固有のセマンティクスに対して予約されます。

PT_SUNWBSS

この配列要素は、PT_LOAD 要素と同じ属性を持ち、.SUNW_bss セクションを記述するために使用されます。

PT_SUNWSTACK

この配列要素はプロセススタックを記述します。現在のところ、そういった要素は1つのみ存在し、`p_flags` フィールドで定義されているアクセスパーミッションのみが有用です。

PT_LOPROC - PT_HIPROC

この範囲の値は、プロセッサ固有のセマンティクスに対して予約されます。

注 - 他の箇所ですべてのプログラムヘッダーセグメントタイプはそれぞれ存在することもありますし、存在しないこともあります。つまり、ファイルのプログラムヘッダーテーブルには、このプログラムの内容に関する要素のみが存在できます。

ベースアドレス

実行可能オブジェクトファイルと共有オブジェクトファイルには、ベースアドレス (プログラムのオブジェクトファイルのメモリーイメージに関連付けられている最下位仮想アドレス) が存在します。ベースアドレスは、たとえば動的リンク時にプログラムのメモリーイメージを再配置するために使用されます。

実行可能オブジェクトファイルと共有オブジェクトファイルのベースアドレスは、実行時に3つの値 (プログラムの読み込み可能セグメントのメモリー読み込みアドレス、最大ページサイズ、最下位仮想アドレス) から計算されます。287ページの「プログラムの読み込み (プロセッサ固有)」に記述されているとおり、プログラムヘッダーの仮想アドレスは、プログラムのメモリーイメージの実際の仮想アドレスを表さないことがあります。

ベースアドレスを計算するには、`PT_LOAD` セグメントの最下位 `p_vaddr` 値に関連付けられているメモリーアドレスを判定します。次に、メモリーアドレスを最大ページサイズの最近倍数に切り捨てることで、ベースアドレスが求められます。メモリーに読み込まれるファイルの型によって、メモリーアドレスは `p_vaddr` 値に一致する場合もあれば、一致しない場合もあります。

セグメントへのアクセス権

システムで読み込まれるプログラムには、少なくとも1つの読み込み可能セグメントが存在しなければなりません (ただし、このことはファイル形式では要求されてい

ません)。システムは、読み込み可能セグメントのメモリーイメージを作成するとき、`p_flags` 構成要素で指定されるアクセス権を与えます。PF_MASKPROC マスクのすべてのビットは、プロセッサ固有のセマンティクスに対して予約されます。

表 7-37 セグメントフラグビット `p_flags`

名前	値	意味
PF_X	0x1	実行
PF_W	0x2	書き込み
PF_R	0x4	読み取り
PF_MASKPROC	0xf0000000	指定されていない

アクセス権ビットが 0 の場合、その種類のアクセスは拒否されます。実際のメモリーアクセス権は、メモリー管理ユニット (システムによって異なることがある) に依存します。すべてのフラグ組み合わせが有効ですが、システムは要求以上のアクセスを与えることがあります。ただしどんな場合も、特に断りが明示的に記述されていない限り、セグメントは書き込み権を持ちません。表 7-38 は、正確なフラグ解釈と許容されるフラグ解釈を示しています。

表 7-38 セグメントへのアクセス権

	値	正確なフラグ解釈	許容されるフラグ解釈
なし	0	すべてのアクセスが拒否される	すべてのアクセスが拒否される
PF_X	1	実行のみ	読み取り、実行
PF_W	2	書き込みのみ	読み取り、書き込み、実行
PF_W+PF_X	3	書き込み、実行	読み取り、書き込み、実行
PF_R	4	読み取りのみ	読み取り、実行
PF_R+PF_X	5	読み取り、実行	読み取り、実行

表 7-38 セグメントへのアクセス権 続く

	値	正確なフラグ解釈	許容されるフラグ解釈
PF_R+PF_W	6	読み取り、書き込み	読み取り、書き込み、実行
PF_R+PF_W+PF_X	7	読み取り、書き込み、実行	読み取り、書き込み、実行

たとえば、標準的なテキストセグメントは読み取り権と実行権を持っていますが、書き込み権は持っていません。データセグメントは通常、読み取り権、書き込み権、および実行権を持っています。

セグメントの内容

オブジェクトファイルセグメントは、1つまたは複数のセクションで構成されます。ただし、プログラムヘッダーはこの事実には関与しません。ファイルセグメントに1つのセクションが存在するか複数のセクションが存在するかもまた、プログラム読み込み時に重要ではありません。しかし、さまざまなデータが、プログラム実行時や動的リンク時などには存在しなければなりません。セグメント内のセクションの順序と帰属関係は、異なることがあります。また、プロセッサによっては、固有の制約により、以下の例と異なることがあります。

テキストセグメントのセクション(この章の初めのほうで記述している)には、読み取り専用の命令/データが存在します。データセグメントには、書き込み可能のデータ/命令が存在します。すべての特殊セクションの一覧については、表 7-17 を参照してください。特定の実行可能ファイルに存在しているセクションを確認するには、`dump(1)` を使用してください。

PT_DYNAMIC プログラムヘッダー要素は、296ページの「動的セクション」で説明しているとおり、`.dynamic` セクションを指し示します。さらに、`.got` セクションと `.plt` セクションには、位置に依存しないコードと動的リンクに関する情報が存在します。

`.plt` は、テキストセグメントまたはデータセグメントに存在できます(どちらのセグメントに存在するかはプロセッサに依存します)。詳細は、316ページの「大域オフセットテーブル(プロセッサ固有)」と 317ページの「プロシージャのリンクテーブル(プロセッサ固有)」を参照してください。

表 7-12 で記述しているとおり、`.bss` セクションには SHT_NOBITS 型が存在します。このセクションはファイル領域を占めませんが、セグメントのメモリーイメー

ジに与えられます。通常、これらの初期化されていないデータはセグメントの終わりに存在し、その結果、関連付けられているプログラムヘッダー要素において `p_memsz` が `p_filesz` より大きくなります。

プログラムの読み込み (プロセッサ固有)

システムは、プロセスイメージを作成または拡張するとき、ファイルのセグメントを仮想メモリーセグメントに論理的にコピーします。システムがファイルをいつ物理的に読み取るかは、プログラムの挙動やシステムの負荷などに依存します。

プロセスは実行時に論理ページを参照しない限り物理ページを必要とせず、また一般に多くのページを未参照状態のままにします。したがって、物理読み取りを遅延させると、これらの物理読み取りが不要になり、システム性能が向上します。この効率性を実際に実現するには、実行可能オブジェクトファイルと共有オブジェクトファイルには、ファイルオフセットと仮想アドレスがページサイズを法として同じであるセグメントイメージが存在しなければなりません。

32 ビットのセグメントの仮想アドレスとファイルオフセットは、64K (0x10000) を法として同じです。64 ビットのセグメントの仮想アドレスとファイルオフセットは、1 MB (0x100000) を法として同じです。セグメントを最大ページサイズに整列すると、ファイルは物理ページサイズには関係なくページング処理に対して適切になります。

デフォルトでは 64 ビット SPARC プログラムは開始アドレス (0x100000000) にリンクされます。プログラム全体 (テキスト、データ、ヒープ、スタック、共有オブジェクトの依存関係を含む) は、4G バイトより上に存在します。そうすることにより、プログラムがポインタを切り捨てると、アドレス空間の最下位 4G バイトでフォルトが発生することになるので、64 ビットプログラムが正しく作られたことを確認することがより容易になります。64 ビットプログラムは 4G バイトより上でリンクされていますが、コンパイラあるいはリンカーにマップファイルおよび `-M` オプションを使用することにより、4G バイト未満でリンクすることも可能です (/usr/lib/ld/sparcv9/map.below4G を参照)。

次に SPARC バージョンの例を示します。

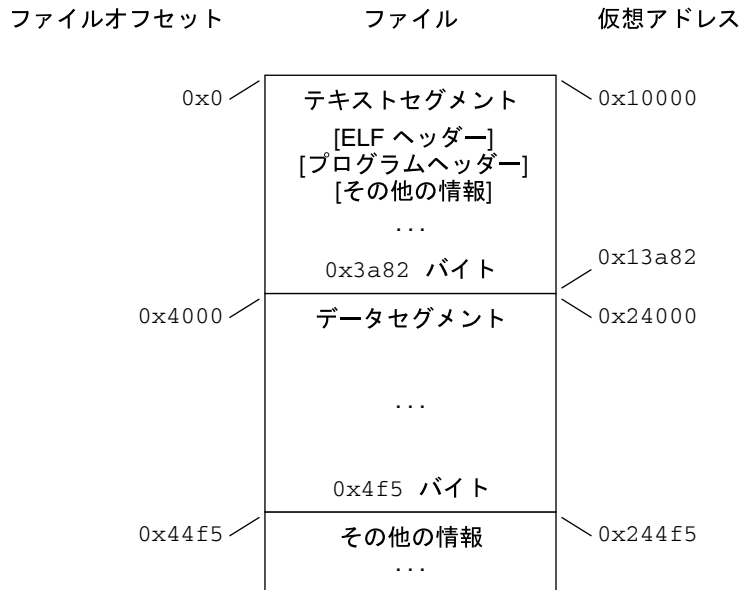


図 7-7 SPARC: 実行可能ファイル (64K に整列)

表 7-39 SPARC: プログラムヘッダーセグメント (64K に整列)

構成要素	テキスト	データ
p_type	PT_LOAD	PT_LOAD
p_offset	0x0	0x4000
p_vaddr	0x10000	0x24000
p_paddr	指定無し	指定無し
p_filesize	0x3a82	0x4f5
p_memsz	0x3a82	0x10a4
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x10000	0x10000

次に IA バージョンの例を示します。

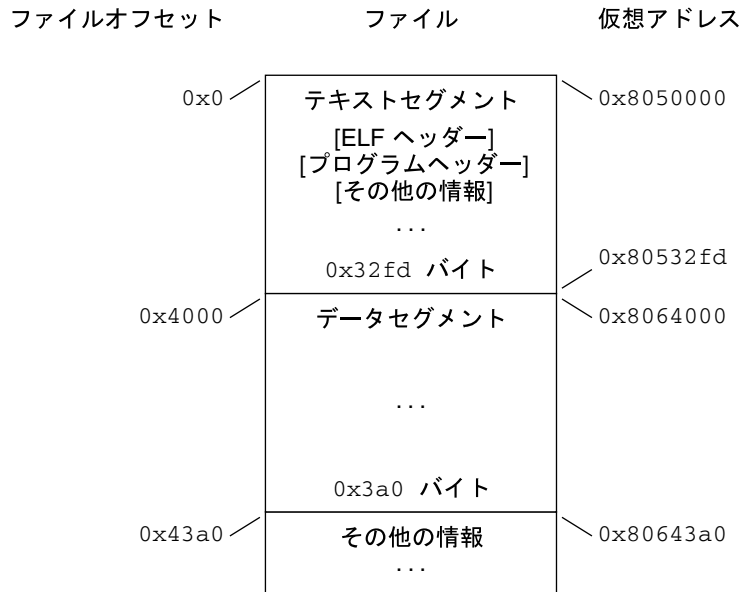


図 7-8 IA: 実行可能ファイル (64K に整列)

表 7-40 IA: プログラムヘッダーセグメント (64K に整列)

構成要素	テキスト	データ
p_type	PT_LOAD	PT_LOAD
p_offset	0x0	0x4000
p_vaddr	0x8050000	0x8064000
p_paddr	指定無し	指定無し
p_filesize	0x32fd	0x3a0
p_memsz	0x32fd	0xdc4
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x10000	0x10000

例で示しているファイルオフセットと仮想アドレスはテキストとデータの両方に対して最大ページサイズを法として同じですが、最大 4 ファイルページ (ページサイズとファイルシステムブロックサイズに依存する) のテキストページまたはデータページは純粹なものではありません (それぞれテキストまたはデータ以外の情報を含みます)。

- 先頭テキストページには、ELF ヘッダー、プログラムヘッダーテーブル、および他の情報が存在する
- 最終テキストページには、データの始まりのコピーが存在する
- 先頭データページには、テキストの終わりのコピーが存在する
- 最後のデータページには、実行中プロセスに関連していないファイル情報が存在できる。論理的にはシステムは、あたかも各セグメントが完全であり分離されているようにメモリアクセス権を設定する。セグメントのアドレスは、アドレス空間の各論理ページに同じ許可が確実に存在できるよう調整される。上の例では、テキストの終わりとデータの始まりを保持しているファイル領域が 2 回対応付けされます (1 回はテキストに関して 1 つの仮想アドレスで対応付けされ、もう 1 回はデータに関して別の仮想アドレスで対応付けされる)

注 - 上記の例は、テキストセグメントを丸めた、典型的な Solaris のシステムバイナリを反映したものです (表 8-1 を参照)。

データセグメントの終わりは、初期化されていないデータに対して特別な処理を必要とします (初期値が 0 になるようにシステムで定義されている)。したがって、ファイルの最後のデータページに、論理メモリーページに存在しない情報が存在する場合、これらのデータは 0 に設定しなければなりません (実行可能ファイルの未知の内容のままにはしてはならない)。

他の 3 ページに存在する純粹でないテキストまたはデータは、論理的にはプロセスイメージの一部ではありません。システムがこれらの純粹でないテキストまたはデータを除去するかどうかについては、規定されていません。このプログラムのメモリーイメージが後に続き、4 キロバイト (0x1000) ページを使用します。単純化するために次の例では、1 ページのサイズのみを示しています。

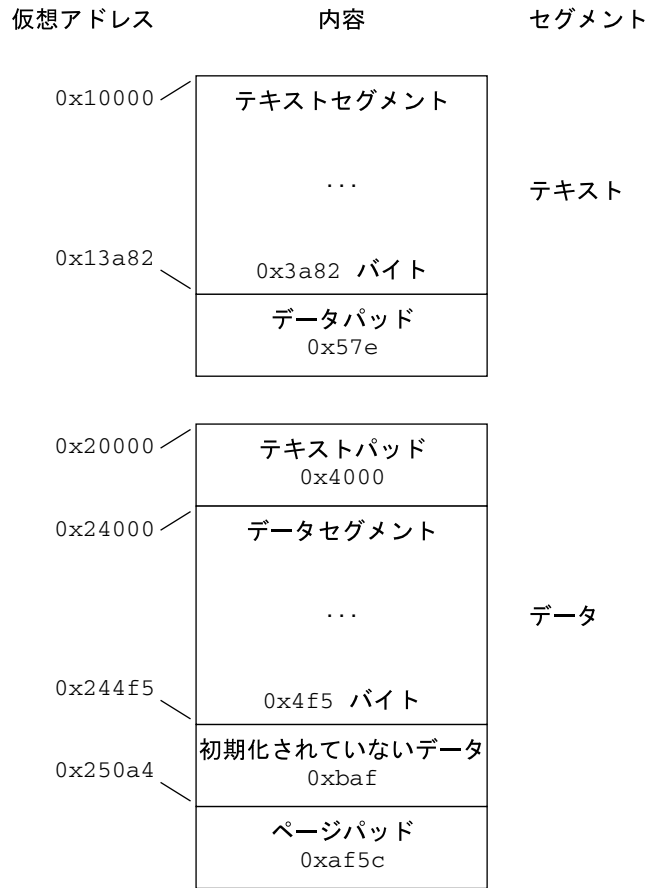


図 7-9 SPARC: プロセスイメージセグメント

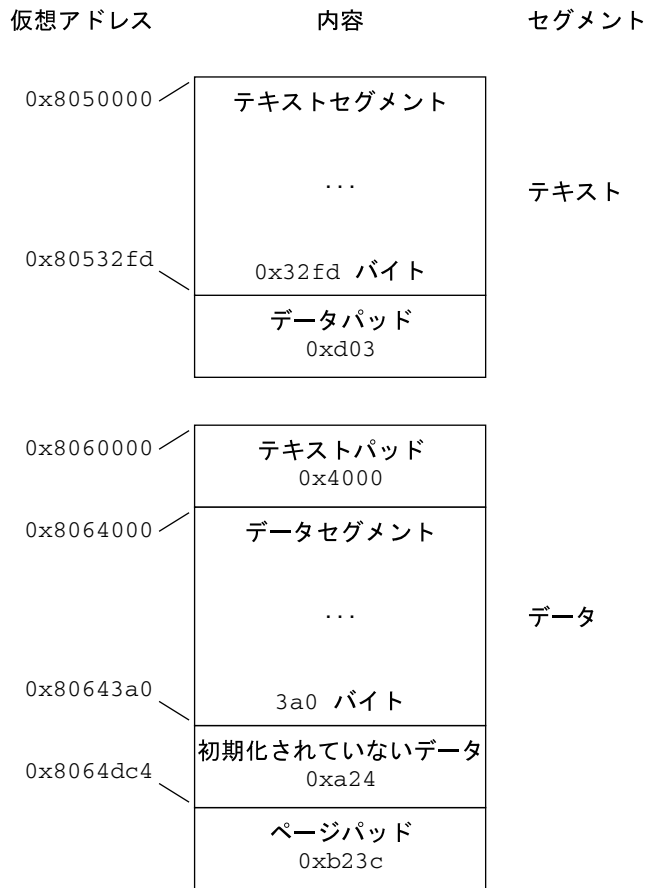


図 7-10 IA: プロセスイメージセグメント

セグメント読み込みは、実行可能ファイルと共有オブジェクトでは異なる側面が 1 つ存在します。実行可能ファイルのセグメントには、標準的には絶対コードが存在します。プロセスを正しく実行するには、セグメントは実行可能ファイルを作成するために使用された仮想アドレスに存在しなければなりません。したがって、システムは変化しない `p_vaddr` 値を仮想アドレスとして使用します。

一方、通常は共有オブジェクトのセグメントには、位置に依存しないコードが存在します。(背景については、第 2 章を参照してください。)したがって、セグメントの仮想アドレスは、実行動作を無効にすることなくプロセスによって異なることができます。

システムは個々のプロセスごとに仮想アドレスを選択しますが、セグメントの相対位置は維持します。位置に依存しないコードはセグメント間で相対アドレス指定を

使用するので、メモリーの仮想アドレス間の差は、ファイルの仮想アドレス間の差に一致しなければなりません。

以下の表は、いくつかのプロセスに対する共有オブジェクト仮想アドレスの割り当の例で、一定の相対位置になることを示しています。これらの表は、ベースアドレスの計算も示しています。

表 7-41 SPARC: 共有オブジェクトのセグメントアドレスの例

ソース	テキスト	データ	ベースアドレス
ファイル	0x0	0x4000	0x0
プロセス 1	0xc0000000	0xc0024000	0xc0000000
プロセス 2	0xc0010000	0xc0034000	0xc0010000
プロセス 3	0xd0020000	0xd0024000	0xd0020000
プロセス 4	0xd0030000	0xd0034000	0xd0030000

表 7-42 IA: 共有オブジェクトセグメントアドレスの例

ソース	テキスト	データ	ベースアドレス
ファイル	0x0	0x4000	0x0
プロセス 1	0x8000000	0x8004000	0x80000000
プロセス 2	0x80081000	0x80085000	0x80081000
プロセス 3	0x900c0000	0x900c4000	0x900c0000
プロセス 4	0x900c6000	0x900ca000	0x900c6000

プログラムインタプリタ

動的リンクに関与する実行可能ファイルは、1つの PT_INTERP プログラムヘッダー要素を持つことができます。exec(2) 時、システムは PT_INTERP セグメントからパス名を取り出し、インタプリタファイルのセグメントから初期プロセスイメージを作成します。つまり、システムは元の実行可能ファイルのセグメントイメージを使用する代わりに、インタプリタのメモリーイメージを作成します。したがって、インタプリタはシステムから制御を受け取り、アプリケーションプログラムに対して環境を与えなければなりません。

インタプリタは、次の2つの方法のどちらかで制御を受け取ります。方法1: インタプリタは、実行可能ファイルを読み込むためのファイル記述子(先頭に位置付けられている)を受け取ることができます。インタプリタは、このファイル記述子を使用して、実行可能ファイルのセグメントをメモリーに読み込んだり対応付けしたりできます。方法2: 実行可能ファイルの形式によっては、システムはインタプリタにファイル記述子を与える代わりに実行可能ファイルをメモリーに読み込みます。

ファイル記述子が異なる可能性があることを除き、インタプリタの初期プロセス状態は、実行可能ファイルの初期プロセス状態に一致します。インタプリタ自身が、2番目のインタプリタを必要とすることは許可されません。インタプリタは、共有オブジェクトファイルまたは実行可能ファイルです。

- 共有オブジェクト(一般的な場合)は位置に依存しないものとして読み込みされ、アドレスはプロセスによって異なる。システムは、mmap(2) および関連するサービスで使用される動的セグメント領域にセグメントを作成する。したがって、一般的に、共有オブジェクトインタプリタは元の実行可能ファイルの元のセグメントアドレスと競合しない
- 実行可能ファイルは、固定アドレスに読み込まれる。システムは、プログラムヘッダーテーブルからの仮想アドレスを使用して実行可能ファイルのセグメントを作成する。したがって、実行可能ファイル(インタプリタ)の仮想アドレスは、最初に実行された実行可能ファイルと競合することがある。競合を解決するのはインタプリタの役目

実行時リンカー

リンカーは、動的リンクを使用する実行可能ファイルを作成するとき、PT_INTERP 型のプログラムヘッダー要素を実行可能ファイルに付加し、実行時リンカーをプログラムインタプリタとして呼び出すようにシステムに指示します。exec(2) と実行

時リンカーは、協調してプログラムのプロセスイメージを作成します。この処理には以下の動作が伴います。

- 実行可能ファイルのメモリーセグメントをプロセスイメージに付加する
- 共有オブジェクトメモリーセグメントをプロセスイメージに付加する
- 実行可能ファイルと共有オブジェクトに対して再配置を行う
- 実行可能ファイルの読み取りに使用されたファイル記述子を閉じる(このファイル記述子が実行時リンカーに与えられている場合)
- 読み込まれるオブジェクトで提供される初期設定セクションを呼び出す。324ページの「初期設定関数および終了関数」を参照
- プログラムがあたかも `exec(2)` から直接制御を受け取ったように見えるように、プログラムに制御を渡す

リンカーはまた、実行時リンカーが実行可能オブジェクトファイルと共有オブジェクトファイルを処理するときの手助けになるさまざまなデータを作成します。先出のプログラムヘッダーセグメントで示されているとおり、これらのデータは読み込み可能セグメントに存在します。したがって、これらのデータは実行時に使用可能です。(正確なセグメント内容はプロセッサに固有であることを思い出してください。)

- `SHT_DYNAMIC` 型の `.dynamic` セクションには、さまざまなデータが存在する。このセクションの始まりに存在する構造体には、他の動的リンク情報のアドレスが存在する。
- `SHT_HASH` 型の `.hash` セクションには、シンボルハッシュテーブルが存在する
- `SHT_PROGBITS` 型の `.got` セクションと `.plt` セクションには、2つの別個のテーブルが存在する。つまり、大域オフセットテーブルとプロシージャのリンクテーブルが存在する。次の項では、オブジェクトファイルのメモリーイメージを作成するために実行時リンカーがテーブルをどのように使用するかを説明する

287ページの「プログラムの読み込み(プロセッサ固有)」で説明しているとおり、共有オブジェクトは、ファイルのプログラムヘッダーテーブルに記録されているアドレスとは異なる仮想メモリーアドレスに置かれることができます。実行時リンカーは、アプリケーションが制御を取得する前に、メモリーイメージを再配置して絶対アドレスを更新します。プログラムヘッダーテーブルに指定されているアドレスにライブラリが読み込まれた場合、絶対アドレス値は正しいのですが、通常、このようなことは起きません。

プロセス環境 (`exec(2)` を参照) に、空ではない値を持つ `LD_BIND_NOW` という変数が存在する場合、実行時リンカーは、プログラムに制御を渡す前にすべての再配置を処理します。たとえば、以下の各環境エンタリは、この動作を指定します。

```
LD_BIND_NOW=1
LD_BIND_NOW=on
LD_BIND_NOW=off
```

実行時リンカーは、プロシージャのリンクテーブルエントリを遅延評価できます。その場合、呼び出されない関数の解決/再配置に費やされる無駄を回避できます。詳細は、317ページの「プロシージャのリンクテーブル (プロセッサ固有)」を参照してください。

動的セクション

オブジェクトファイルが動的リンクに関係している場合、このオブジェクトファイルのプログラムヘッダーテーブルには、PT_DYNAMIC 型の要素が存在します。このセグメントには、.dynamic セクションが存在します。特殊なシンボル _DYNAMIC は、このセクションを示します。このセクションには、以下の構造体 (sys/link.h で定義される) の配列が存在します。

```
typedef struct {
    Elf32_Sword d_tag;
    union {
        Elf32_Word    d_val;
        Elf32_Addr    d_ptr;
        Elf32_Off     d_off;
    } d_un;
} Elf32_Dyn;

typedef struct {
    Elf64_Xword d_tag;
    union {
        Elf64_Xword    d_val;
        Elf64_Addr     d_ptr;
    } d_un;
} Elf64_Dyn;
```

このタイプの各オブジェクトの場合、d_tag は d_un の解釈に影響します。

d_val

このオブジェクトは、さまざまに解釈される整数値を表します。

d_ptr

このオブジェクトは、プログラムの仮想アドレスを表わします。すでに述べたように、ファイルの仮想アドレスは、実行時にメモリーの仮想アドレスに一致しないこ

とがあります。実行時リンカーは、動的構造体に存在するアドレスを解釈するとき、元のファイル値とメモリーのベースアドレスに基づいて実際のアドレスを計算します。整合性のため、ファイルには動的構造体内のアドレスを補正するための再配置エントリは存在しません。

ツールによる動的セクションエントリの内容の解釈をシンプルにするために、各タグの値 (2 つの特別な互換性の範囲は除く) で `d_un union` の解釈を決定するとします。偶数の値を持つタグは、`d_ptr` を使用する動的セクションのエントリを示します。奇数の値を持つタグは、`d_val` を使用する動的セクションのエントリ、または `d_ptr` と `d_val` のどちらも使用しない動的セクションのエントリを示します。特殊値 `DT_ENCODING` より小さい値を持つタグ、および `DT_HIOS` と `DT_LOPROC` 間の範囲に入る値を持つタグは、これらの規則には従いません。

表 7-43 は、実行可能オブジェクトファイルと共有オブジェクトファイルのタグ要求についてまとめています。タグに「必須」という印が付いている場合、動的リンク配列にはその型のエントリが存在しなければなりません。また、「任意」は、タグのエントリが現れてもよいですが必須ではないことを意味します

表 7-43 動的配列タグ `d_tag`

名前	値	<code>d_un</code>	実行可能 ファイル	共有オブジェ クトファイル
<code>DT_NULL</code>	0	無視される	必須	必須
<code>DT_NEEDED</code>	1	<code>d_val</code>	任意	任意
<code>DT_PLTRELSZ</code>	2	<code>d_val</code>	任意	任意
<code>DT_PLTGOT</code>	3	<code>d_ptr</code>	任意	任意
<code>DT_HASH</code>	4	<code>d_ptr</code>	必須	必須
<code>DT_STRTAB</code>	5	<code>d_ptr</code>	必須	必須
<code>DT_SYMTAB</code>	6	<code>d_ptr</code>	必須	必須
<code>DT_RELA</code>	7	<code>d_ptr</code>	必須	任意
<code>DT_RELASZ</code>	8	<code>d_val</code>	必須	任意

表 7-43 動的配列タグ d_tag 続く

名前	値	d_un	実行可能 ファイル	共有オブジェ クトファイル
DT_RELAENT	9	d_val	必須	任意
DT_STRSZ	10	d_val	必須	必須
DT_SYMENT	11	d_val	必須	必須
DT_INIT	12	d_ptr	任意	任意
DT_FINI	13	d_ptr	任意	任意
DT_SONAME	14	d_val	無視される	任意
DT_RPATH	15	d_val	任意	任意
DT_SYMBOLIC	16	無視される	無視される	任意
DT_REL	17	d_ptr	必須	任意
DT_RELSZ	18	d_val	必須	任意
DT_RELENT	19	d_val	必須	任意
DT_PLTREL	20	d_val	任意	任意
DT_DEBUG	21	d_ptr	任意	無視される
DT_TEXTREL	22	無視される	任意	任意
DT_JMPREL	23	d_ptr	任意	任意
DT_BIND_NOW	24	無視される	任意	任意
DT_INIT_ARRAY	25	d_ptr	任意	任意
DT_FINI_ARRAY	26	d_ptr	任意	任意

表 7-43 動的配列タグ d_tag 続く

名前	値	d_un	実行可能 ファイル	共有オブジェ クトファイル
DT_INIT_ARRAYSZ	27	d_val	任意	任意
DT_FINI_ARRAYSZ	28	d_val	任意	任意
DT_RUNPATH	29	d_val	任意	任意
DT_FLAGS	30	d_val	任意	任意
DT_ENCODINGS	32	未指定	未指定	未指定
DT_PREINIT_ARRAY	32	d_ptr	任意	任意
DT_PREINIT_ARRAYSZ	33	d_val	任意	任意
DT_LOOS	0x6000000d	未指定	未指定	未指定
DT_HIOS	0x6ffff000	未指定	未指定	未指定
DT_VALRNGLO	0x6ffffd00	未指定	未指定	未指定
DT_CHECKSUM	0x6ffffdf8	d_val	任意	任意
DT_PLTPADSZ	0x6ffffdf9	d_val	任意	任意
DT_MOVEENT	0x6ffffdfa	d_val	任意	任意
DT_MOVESZ	0x6ffffdfb	d_val	任意	任意
DT_FEATURE	0x6ffffdfc	d_val	任意	任意
DT_POSFLAG_1	0x6ffffdfd	d_val	任意	任意
DT_SYMINSZ	0x6ffffdfe	d_val	任意	任意
DT_SYMINENT	0x6ffffdff	d_val	任意	任意

表 7-43 動的配列タグ d_tag 続く

名前	値	d_un	実行可能 ファイル	共有オブジェ クトファイル
DT_VALRNGHI	0x6ffffdfff	未指定	未指定	未指定
DT_ADDRNGLO	0x6ffffe00	未指定	未指定	未指定
DT_CONFIG	0x6ffffefa	d_ptr	任意	任意
DT_DEPAUDIT	0x6ffffefb	d_ptr	任意	任意
DT_AUDIT	0x6ffffefc	d_ptr	任意	任意
DT_PLTPAD	0x6ffffefd	d_ptr	任意	任意
DT_MOVETAB	0x6ffffefe	d_ptr	任意	任意
DT_SYMINFO	0x6ffffeff	d_ptr	任意	任意
DT_ADDRNGHI	0x6ffffeff	未指定	未指定	未指定
DT_RELACOUNT	0x6fffff9	d_val	任意	任意
DT_RELCOUNT	0x6fffffa	d_val	任意	任意
DT_FLAGS_1	0x6fffffb	d_val	任意	任意
DT_VERDEF	0x6fffffc	d_ptr	任意	任意
DT_VERDEFNUM	0x6fffffd	d_val	任意	任意
DT_VERNEED	0x6fffffe	d_ptr	任意	任意
DT_VERNEEDNUM	0x6ffffff	d_val	任意	任意
DT_AUXILIARY	0x7fffffd	d_val	未指定	任意
DT_USED	0x7fffffe	d_val	任意	任意

表 7-43 動的配列タグ d_tag 続く

名前	値	d_un	実行可能 ファイル	共有オブジェ クトファイル
DT_FILTER	0x7fffffff	d_val	未指定	任意
DT_LOPROC	0x70000000	未指定	未指定	未指定
DT_SPARC_REGISTER	0x70000001	d_val	任意	任意
DT_HIPROC	0x7fffffff	未指定	未指定	未指定

DT_NULL

DT_NULL タグが付けられたエントリは、_DYNAMIC 配列の終わりを示します。

DT_NEEDED

この要素は、空文字で終わっている文字列の文字列テーブルオフセットを保持し、必要な依存性の名前を与えます。オフセットは、DT_STRTAB エントリに記録されるテーブルへのインデックスです。これらの名前については、316ページの「共有オブジェクトの依存性」を参照してください。動的配列には、この型の複数のエントリが存在できます。これらのエントリの相対順序は意味がありますが、他の型のエントリに対するこれらのエントリの相対順序には意味がありません。

DT_PLTRELSZ

この要素は、プロシージャのリンクテーブルに関連付けられている再配置エントリの合計サイズ (単位: バイト) を保持します。DT_JMPREL 型のエントリが存在する場合、DT_PLTRELSZ 型のエントリも必要です。

DT_PLTGOT

この要素は、プロシージャのリンクテーブルまたは大域オフセットテーブルに関連付けられているアドレスを保持します。

DT_HASH

この要素は、シンボルハッシュテーブル (323ページの「ハッシュテーブル」に記述されている) を指し示します。ハッシュテーブルは、DT_SYMTAB 要素で示されるシンボルテーブルを参照します。

DT_STRTAB

この要素は、文字列テーブル (この章の前半部分で記述している) のアドレスを保持します。文字列テーブルには、実行時リンカーが必要とするシンボル名、依存性名、および他の文字列が存在します。

DT_SYMTAB

この要素は、再配置テーブル (この章の前半部分で記述している) のアドレスを保持します。32 ビットクラスのファイルに対しては Elf32_Sym エントリ、64 ビットクラスのファイルに対しては ELF 64_sym エントリが存在します。

DT_RELA

この要素は、再配置テーブル (この章の前半部分で記述している) のアドレスを保持します。再配置テーブルのエントリ (32 ビットファイルクラス用の Elf32_Rel、64 ビットファイルクラス用の ELF 64_Rel など) には、明示的加数が存在します。

オブジェクトファイルには、複数の再配置セクションが存在できます。リンカーは、実行可能オブジェクトファイルまたは共有オブジェクトファイルの再配置テーブルを作成するとき、これらのセクションを連結して単一のテーブルを作成します。これらの各セクションはオブジェクトファイル内で個々に独立したままですが、実行時リンカーは単一のテーブルとして扱います。実行時リンカーは、実行可能ファイルのプロセスイメージを作成したり、またはプロセスイメージに共有オブジェクトを付加したりするとき、再配置テーブルを読み取り、関連付けられている動作を実行します。

この要素が存在する場合、動的構造体には DT_RELASZ 要素と DT_RELAENT 要素も存在しなければなりません。再配置がファイルに対して「必須」の場合、DT_RELA または DT_REL が使用可能です (これらは両方とも使用可能ですが、必要ではありません)。

DT_RELASZ

この要素は、DT_RELA 再配置テーブルの合計サイズ (単位: バイト) を保持します。

DT_RELAMENT

この要素は、DT_RELA 再配置エントリのサイズ (単位: バイト) を保持します。

DT_STRSZ

この要素は、文字列テーブルのサイズ (単位: バイト) を保持します。

DT_SYMENT

この要素は、シンボルテーブルエントリのサイズ (単位: バイト) を保持します。

DT_INIT

この要素は、初期化関数のアドレスを保持します (初期化関数については、324ページの「初期設定関数および終了関数」で記述しています)。

DT_FINI

この要素は、終了関数のアドレスを保持します (終了関数については、324ページの「初期設定関数および終了関数」に記述されています)。

DT_SONAME

この要素は、空文字で終わっている文字列の文字列テーブルオフセットを保持し、共有オブジェクトの名前を与えます。このオフセットは、DT_STRTAB エントリに記録されているテーブルへのインデックスです。これらの名前については、316ページの「共有オブジェクトの依存性」を参照してください。

DT_RPATH

この要素は、ライブラリ検索パス (空文字で終わっている文字列) 文字列テーブルオフセットを保持します (ライブラリ検索パスについては、113ページの「依存関係を持つ共有オブジェクト」で記述しています)。このオフセットは、DT_STRTAB エントリに記録されているテーブルへのインデックスです。この使用は、DT_RUNPATH に置き換えられました。

DT_SYMBOLIC

この要素は、オブジェクトが、リンク編集時に適用されたシンボル結合を含むかどうかを示します。134ページの「-Bsymbolic の使用」も参照してください。この使用は、DF_SYMBOLIC に置き換えられました。

DT_REL

この要素は DT_RELA に似ていますが、テーブルに暗黙の加数 (32 ビットファイルクラス用の Elf32_Rel など) が存在する点が異なります。この要素が存在する場合、動的構造体には DT_RELSZ 要素と DT_RELENT 要素も存在しなければなりません。

DT_RELSZ

この要素は、DT_REL 再配置テーブルの合計サイズ (単位: バイト) を保持します。

DT_RELENT

この要素は、DT_REL 再配置エントリのサイズ (単位: バイト) を保持します。

DT_PLTREL

この要素は、プロシージャのリンクテーブルが参照する再配置エントリの型を指定します。d_val 要素は、必要に応じて DT_REL または DT_RELA を保持します。1 つのプロシージャのリンクテーブルでは、すべての再配置は、同じ再配置を使用しなければなりません。

DT_DEBUG

この要素は、デバッグ時に使用されます。

DT_TEXTREL

この構成要素が存在しない場合、再配置エントリが書き込み不可セグメント (プログラムヘッダーテーブルのセグメントのアクセス権で指定されます) に対する変更を発生させてはならないことを意味します。この構成要素が存在する場合、1 つまたは複数の再配置エントリが書き込み不可セグメントに対する変更を要求する可能性があります。実行時リンカーはそれに応じて対応することができます。この使用は、DF_TEXTREL フラグに置き換えられました。

DT_JMPREL

このエントリが存在する場合、このエントリの d_ptr 要素は、プロシージャのリンクテーブルにのみ関連付けられている再配置エントリのアドレスを保持します。これらの再配置エントリを分離しておく、遅延結合が有効の場合、実行時リンカーはプロセス初期化時にこれらの再配置エントリを無視します。このエントリが存在する場合、DT_PLTRELSZ 型と DT_PLTREL 型の関連エントリも存在しなければなりません。

DT_POSFLAG_1

このエントリが存在する場合、このエントリの `d_val` 要素は、さまざまな状態フラグを保持します。これらのフラグは、`.dynamic` セクション内の直後の `DT_*` エントリに適用されます。

DT_BIND_NOW

このエントリが共有オブジェクトまたは実行可能ファイル内に存在する場合、このエントリは、プログラムに制御を渡す前に、このエントリを含むオブジェクトについてのすべての再配置を処理するよう実行時リンカーに指示します。環境で指定された、または `dlopen(3DL)` を介した場合、このエントリの存在は、このオブジェクトについてレイジー結合を使用するようという指示よりも優先されます。この使用は、`DF_BIND_NOW` フラグに置き換えられました。

DT_INIT_ARRAY

この要素は、324ページの「初期設定関数および終了関数」に記述されている、初期設定関数へのポインタの配列のアドレスを保持します。

DT_FINI_ARRAY

この要素は、324ページの「初期設定関数および終了関数」に記述されている、終了関数へのポインタの配列のアドレスを保持します。

DT_RUNPATH

この要素は、113ページの「依存関係を持つ共有オブジェクト」に記述されている、`NULL` で終わる検索ライブラリの検索パス文字列の文字列テーブルオフセットを保持します。このオフセットは、`DT_STRTAB` エントリに記録されたテーブルに対するインデックスです。

DT_FLAGS

この要素は、読み込まれるオブジェクトに特有のフラグ値を保持します。各フラグ値は、`DF_flag_name` という名前を持ちます。定義された値とその意味は後述されています。その他のすべての値は予約されています。

DT_ENCODING

`DT_ENCODING` と等しいかそれより大きく、かつ `DT_LOOS` より小さい値は、前述の `d_un union` の解釈の規則に従います。

DT_PREINIT_ARRAY

この要素は、324ページの「初期設定関数および終了関数」に記述されている、初期設定前関数へのポインタの配列のアドレスを保持します。DT_PREINIT_ARRAY テーブルは実行可能ファイル内でのみ処理されます。共有オブジェクト内に含まれている場合は無視されます。

DT_PREINIT_ARRAYSZ

この要素は、DT_PREINIT_ARRAY エントリにポイントされた初期設定前関数の配列のサイズ (単位: バイト) を保持します。1つのオブジェクトが DT_PREINIT_ARRAY エントリを持つ場合は、このオブジェクトは DT_PREINIT_ARRAYSZ エントリも持つ必要があります。DT_PREINIT_ARRAY と同様、このエントリは共有オブジェクト内に現れた場合は無視されます。

DT_LOOS - DT_HIOS

この範囲内の値 (両端の値も含む) は、オペレーティングシステム特有のセマンティクスのために予約されています。このような値はすべて、前述の `d_un union` の解釈の規則に従います。

DT_SYMINFO

この要素は、SHT_SUNW_syminfo セクションのアドレスを保持します。

DT_SYMINENT

この要素は、SHT_SUNW_syminfo テーブルエントリのサイズ (単位: バイト) を保持します。

DT_SYMINSZ

この要素は、SHT_SUNW_syminfo セクションのサイズ (単位: バイト) を保持します。

DT_VERDEF

この要素は、バージョン定義テーブル (この章の前半部分で記述している) のアドレスを保持します。32 ビットクラスのファイルに対しては `Elf32_Verdef` エントリ、64 ビットクラスのファイルに対しては `Elf64_Verdef` エントリが存在します。詳細は、268ページの「バージョン定義セクション」を参照してください。これ

らのエン트리内の要素には、DT_STRTAB エントリに記録されているテーブルへのインデックスが存在します。

DT_VERDEFNUM

この要素は、バージョン定義テーブルのエントリ数を指定します。

DT_VERNEED

この要素は、バージョン依存性テーブル (この章の前半部分で記述している) のアドレスを保持します。32 ビットクラスのファイルに対しては Elf32_Verneed エントリ、64 ビットクラスのファイルに対しては Elf64_Verneed エントリが存在します。詳細は、272ページの「バージョン依存セクション」を参照してください。これらのエン트리内の要素には、DT_STRTAB エントリに記録されているテーブルへのインデックスが存在します。

DT_VERNEEDNUM

この要素は、バージョン依存性テーブルのエントリ数を指定します。

DT_RELACOUNT

すべての Elf32_Rela あるいは Elf64_Rela R_*_RELATIVE 再配置は単一のブロックに入っており、このエント리는そのブロックのエントリ数を指定します。これにより、実行時リンカーは RELATIVE 再配置の処理を合理化できます。

DT_RELCOUNT

すべての Elf32_Rel R_*_RELATIVE 再配置は単一のブロックに入っており、このエント리는そのブロックのエントリ数を指定します。これにより、実行時リンカーは RELATIVE 再配置の処理を合理化できます。

DT_AUXILIARY

この要素は、空文字で終わっている文字列 (オブジェクトに命名している) の文字列テーブルオフセットを保持します。このオフセットは、DT_STRTAB エントリに記録されているテーブルへのインデックスです。補助オブジェクト内のシンボルは、このオブジェクト内のシンボルに優先して使用されます。

DT_FILTER

この要素は、オブジェクトを示す文字列 (空文字で終わっている文字列) の文字列テーブルオフセットを保持します。このオフセットは、DT_STRTAB エントリに記録されているテーブルへのインデックスです。このオブジェクトのシンボルテーブルは、命名されたオブジェクトのシンボルテーブルのフィルタとして使用されます。

DT_CHECKSUM

この要素は、オブジェクトの選択されたセクションの簡単なチェックサムを保持します。gelf_checksum(3ELF) を参照してください。

DT_MOVEENT

この要素は、SHT_SUNW_move で定義されるセクションの合計サイズを保持します。276ページの「移動セクション」を参照してください。

DT_MOVESZ

この要素は、DT_MOVETAB 移動エントリのサイズ (単位: バイト) を保持します。

DT_MOVETAB

この要素は、移動テーブル (この章の前半部分で記述している) のアドレスを保持します。移動テーブルのエントリには、32 ビットクラスのファイルに対する Elf32_Move エントリ、64 ビットクラスのファイルに対する Elf64_Move エントリが存在します。

DT_CONFIG

この要素は、構成ファイルを定義する文字列 (ヌル文字で終端している文字列) の文字列テーブルオフセットを保持します。このオフセットは DT_STRTAB エントリに記録されているテーブルへのインデックスです。構成ファイルは、検索パス、ディレクトリキャッシュ、または代替オブジェクトを提供し、通常 DT_CONFIG エントリを指定するアプリケーションに固有です。crle(1) およびリンカーの -c オプションを参照してください。

DT_DEPAUDIT

この要素は、1 つあるいは複数の監査ライブラリを定義する文字列 (ヌル文字で終端している文字列) の文字列テーブルオフセットを保持します。このオフセットは DT_STRTAB エントリに記録されているテーブルへのインデックスです。監査ライブ

ラリは実行時に、このオブジェクトを監査するために使用されます。リンカーの `-p` オプションを参照してください。

DT_AUDIT

この要素は、1つあるいは複数の監査ライブラリを定義する文字列 (ヌル文字で終了している文字列) の文字列テーブルオフセットを保持します。このオフセットは `DT_STRTAB` エントリに記録されているテーブルへのインデックスです。監査ライブラリは実行時に、このオブジェクトを監査するために使用されます。リンカーの `-p` オプションを参照してください。

DT_FLAGS_1

このエントリが存在する場合、このエントリの `d_val` 要素は、さまざまな状態フラグを保持します。表 7-45 を参照してください。

DT_FEATURE_1

このエントリが存在する場合、このエントリの `d_val` 構成要素は実行時機能要求を定義する `DTF_1_` フラグを保持します。関数 `_check_rtld_feature()` は、通常、コンパイラ供給の起動コードから実行時に呼び出され、要求された機能が存在することを確認します。102ページの「機能チェッカ」、および表 7-47 を参照してください。

DT_VALRNGLO - DT_VALRNGHI

この範囲の値は、動的構造体内の `d_un.d_val` フィールドで使用されます。

DT_ADDRNGLO - DT_ADDRNGHI

この範囲の値は、動的構造体内の `d_un.d_ptr` フィールドで使用されます。ELF オブジェクトが作成後に調整された場合、これらのエントリも更新する必要があります。

DT_SPARC_REGISTER

この要素には、`STT_SPARC_REGISTER` シンボルのインデックスが存在します。シンボルテーブルの各 `STT_SPARC_REGISTER` シンボルテーブルエントリには、これらのエントリの1つが存在します。

この範囲の値は、プロセッサに固有の使用方法に対して予約されています。

動的配列の最後にある DT_NULL 要素と、DT_NEEDED と DT_POSFLAG_1 要素の相対的な順序を除くと、エントリはどの順序で現れてもかまいません。表に示されていないタグ値は予約されています。

次の表に示す動的状態フラグ DT_FLAGS が現在、使用可能です。

表 7-44 動的タグ DT_FLAGS

名前	値	意味
DF_ORIGIN	0x1	\$ORIGIN 処理が必要
DF_SYMBOLIC	0x2	シンボリックシンボル解決が必要
DF_TEXTREL	0x4	テキストの再配置が存在する
DF_BIND_NOW	0x8	非レイジー結合が必要

DF_ORIGIN

オブジェクトに \$ORIGIN 処理が必要であることを示します。364ページの「関連する依存関係の配置」を参照してください。

DF_SYMBOLIC

オブジェクトが、リンク編集に適用されたシンボリック結合を含むことを示します。134ページの「-Bsymbolic の使用」も参照してください。

DF_TEXTREL

このフラグが無いことは、再配置エントリが書き込み不可セグメント (プログラムヘッダーテーブル内のセグメントのアクセス権で指定される) に対する変更を発生させてはならないことを示します。このフラグが存在する場合は、1つまたは複数の再配置エントリが書き込み不可セグメントに対する変更を要求する可能性があり、実行時リンカーはそれに応じて対応することができます。

DF_BIND_NOW

共有オブジェクトまたは実行可能ファイル内で設定された場合、このフラグは、プログラムに制御を渡す前にこのエントリを含むオブジェクトについてすべての再配置を処理するよう動的リンカーに指示します。環境で指定された、または `dlopen(3DL)` を介した場合、このエントリの存在は、このオブジェクトについてレイジー結合を使用するようという指示よりも優先されます。この状態は、リンカーの `-z now` オプションを使用してオブジェクト内に記録されます。

次の表に示す動的状態フラグ `DT_FLAGS_1` が現在、使用可能です。

表 7-45 動的タグ `DT_FLAGS_1`

名前	値	意味
<code>DF_1_NOW</code>	<code>0x1</code>	完全な再配置処理を行います
<code>DF_1_GLOBAL</code>	<code>0x2</code>	未使用
<code>DF_1_GROUP</code>	<code>0x4</code>	オブジェクトがグループの構成要素であることを示します
<code>DF_1_NODELETE</code>	<code>0x8</code>	オブジェクトがプロセスから削除できないことを示します
<code>DF_1_LOADFLTR</code>	<code>0x10</code>	フィルテーター (1 つまたは複数) の即時読み込みを保証します
<code>DF_1_INITFIRST</code>	<code>0x20</code>	オブジェクトの初期化を最初に実行します
<code>DF_1_NOOPEN</code>	<code>0x40</code>	オブジェクトを <code>dlopen(3DL)</code> で使用できません
<code>DF_1_ORIGIN</code>	<code>0x80</code>	<code>\$ORIGIN</code> 処理が必要です
<code>DF_1_DIRECT</code>	<code>0x100</code>	直接結合が有効です
<code>DF_1_INTERPOSE</code>	<code>0x400</code>	オブジェクトは割り込み処理です
<code>DF_1_NODEFLIB</code>	<code>0x800</code>	デフォルトのライブラリ検索パスを無視します

表 7-45 動的タグ DT_FLAGS_1 続く

名前	値	意味
DF_1_NODUMP	0x1000	オブジェクトを dldump (3DL) でダンプできません
DF_1_CONFALT	0x2000	オブジェクトは代替構成です
DF_1_ENDFILTEE	0x4000	フィルティアーがフィルタの検索を終了します
DF_1_DISPREDNE	0x8000	ディスプレイメント再配置の終了
DF_1_DISPREDPND	0x10000	ディスプレイメント再配置の保留

DF_1_NOW

オブジェクトが読み込まれると、すべての再配置処理が行われます。78ページの「再配置が実行される時」を参照してください。この状態フラグは、リンカーの `-z now` オプションを使うことによって有効になります。

DF_1_GROUP

オブジェクトがグループの構成要素であることを示します。92ページの「シンボル検索」を参照してください。この状態フラグは、リンカーの `-B group` オプションを使うことによって有効になります。

DF_1_NODELETE

オブジェクトがプロセスから削除できないことを示します。したがってオブジェクトは、`dlopen (3DL)` で直接または依存性としてプロセスに読み込まれた場合、`dlclose (3DL)` で読み込み解除できません。この状態フラグは、リンカーの `-z nodelete` オプションを使うことによってオブジェクトに有効になります。

DF_1_LOADFLTR

この状態フラグは、「フィルタ」に対してのみ意味があります (115ページの「フィルタとしての共有オブジェクト」を参照)。フィルタが読み込まれると、関連付けられているすべてのフィルティアーが直ちに処理されます (120ページの「フィルティアー

の処理」を参照)。この状態フラグは、リンカーの `-z loadfltr` オプションを使うことによってオブジェクトに有効になります。

DF_1_INITFIRST

オブジェクトが読み込まれると、このオブジェクトと共に読み込まれた他のオブジェクトより先に、このオブジェクトの初期化セクションが実行されます。103ページの「デバッグングエイド」を参照してください。この特殊な状態フラグは、`libthread.so.1` で使用されることを意図しています。この状態フラグは、リンカーの `-z initfirst` オプションを使うことによって有効になります。

DF_1_NOOPEN

`dlopen(3DL)` を使ってオブジェクトを実行中プロセスに追加できないことを示します。この状態フラグは、リンカーの `-z nodlopen` オプションを使うことによってオブジェクトに有効になります。

DF_1_ORIGIN

オブジェクトに対し `$ORIGIN` 処理が必要なことを表します。364ページの「関連する依存関係の配置」を参照してください。

DF_1_DIRECT

オブジェクトには、直接結合情報が存在します。78ページの「直接結合」を参照してください。

DF_1_INTERPOSE

オブジェクトシンボルテーブルは、起動性実行可能プログラム以外のすべてのシンボルの前に割り込みされます。この状態フラグは、リンカーの `-z interpose` オプションを使うことによって有効になります。78ページの「直接結合」を参照してください。

DF_1_NODEFLIB

このオブジェクトの依存関係を検索する場合、デフォルトのライブラリ検索パスを無視します。37ページの「実行時リンカーが検索するディレクトリ」を参照してください。この状態は、リンカーの `-z nodefaultlib` オプションを使用してオブジェクトに記録されます。

DF_1_NODUMP

このオブジェクトは `dldump(3DL)` によってダンプできません。このオプションの候補には、`libdl.so.1` などのフィルタが含まれます。`libdl.so.1` は、`crle(1)` を使用して構成ファイルを生成する際に含めることができます。この状態は、リンカーの `-z nodump` オプションを使用してオブジェクトに記録されます。

DF_1_CONFALT

`crle(1)` によって生成された代替構成オブジェクトであることを示します。この状態により実行時リンカーがトリガーされ、構成ファイル `$ORIGIN/ld.config.app-name` が検索されます。

DF_1_ENDFILTEE

この状態はフィルTEEに対してのみ意味があります (115ページの「フィルタとしての共有オブジェクト」を参照)。このフラグが設定されていると、それ以後のフィルTEEに対するフィルタ検索は行われません (363ページの「補助検索の縮小」を参照)。この状態をオブジェクトに記録するには、リンカーの `-z endfiltee` オプションを使用します。

DF_1_DISPREDNE

このオブジェクトにディスプレイメント再配置が適用されたことを示します。再配置が適用されると再配置レコードは破棄されるため、オブジェクト内の再配置レコードはもはや存在しません。63ページの「ディスプレイメント再配置」を参照してください。

DF_1_DISPREDPND

このオブジェクトのディスプレイメント再配置が保留されていることを示します。再配置レコードはオブジェクト内に存在しているため、再配置は実行時に完了することができます。63ページの「ディスプレイメント再配置」を参照してください。

表 7-46 動的タグ DT_POSFLAG_1

名前	値	意味
DF_P1_LAZYLOAD	0x1	後続のオブジェクトは、レイジーロードされる
DF_P1_GROUPEPERM	0x2	後続のオブジェクトは、グループパーミッションの状態でもロードされる

DF_P1_LAZYLOAD

後続の DT_* エントリは、読み込まれるオブジェクトを示します。このオブジェクトのローディングは、Syminfo テーブルエントリで示されるシンボル結合がこのオブジェクトを明確に参照するまで遅延されます。この状態は、リンカーの `-z lazyload` オプションを使用してオブジェクト内に記録されます。

DF_P1_GROUPEPERM

後続の DT_* エントリは、読み込まれるオブジェクトを示します。このオブジェクト内のシンボルは、一般的なシンボル解決に対しては使用できません。このオブジェクト内のシンボルは、このオブジェクトを読み込ませたオブジェクト (1 つまたは複数) に対してのみ使用可能です。この状態は、リンカーの `-z grouperpm` オプションを使用してオブジェクト内に記録されます。

表 7-47 に示す動的機能フラグが現在、使用可能です。

表 7-47 動的機能フラグ DT_FEATURE

名前	値	意味
DTF_1_PARINIT	0x1	部分的な初期化機能が必要
DTF_1_CONFEXP	0x2	構成ファイルが必要 (代替オブジェクトについては、 <code>crle(1)</code> によって設定される。DTF_1_CONFALT と同じ効果がある)

共有オブジェクトの依存性

実行時リンカーがオブジェクトファイルのメモリーセグメントを作成するとき、依存性 (動的構造体の DT_NEEDED エントリに記録される) は、プログラムのサービスを提供するためにどの共有オブジェクトが必要であることを示します。参照された共有オブジェクトとそれが依存するものを繰り返し結合することによって、実行時リンカーは完全なプロセスイメージを生成します。

実行時リンカーは、記号参照を解決するとき、幅優先検索を使用してシンボルテーブルを調べます。つまり、実行時リンカーはまず実行可能プログラム自身のシンボルテーブルを参照し、次に DT_NEEDED エントリのシンボルテーブルを (順番に) 参照し、次に第 2 レベルの DT_NEEDED エントリといった具合に参照します。

注 - 共有オブジェクトが依存性リストにおいて複数回参照されるときでも、実行時リンカーはこの共有オブジェクトをプロセスに 1 回だけ結合します。

依存リストに示されている名前は、オブジェクトファイルの作成に使用される共有オブジェクトの DT_SONAME 文字列またはパス名です。

大域オフセットテーブル (プロセッサ固有)

一般に位置に依存しないコードには絶対仮想アドレスは存在できません。大域オフセットテーブルには内部で使用するデータの絶対アドレスが存在しており、したがって、位置からの独立性とプログラムのテキストの共有性を低下させることなくアドレスが使用可能になります。プログラムは、位置に依存しないアドレス指定を使用して大域オフセットテーブルを参照し、絶対値を抜き出すことで位置に依存しない参照を、絶対位置に向け直します。

最初は、大域オフセットテーブルには再配置エントリで要求される情報が存在しません (詳細は、248ページの「再配置」を参照)。システムが読み込み可能オブジェクトファイルのメモリーセグメントを作成した後、実行時リンカーが再配置エントリを処理します。これらの再配置エントリのいくつかは、R_SPARC_GLOB_DAT 型 (SPARC の場合) または R_386_GLOB_DAT 型 (IA の場合) であり、大域オフセットテーブルを参照します。

実行時リンカーは、関連付けられているシンボル値を判定し、絶対アドレスを計算し、適切なメモリーテーブルエントリに正しい値を設定します。リンカーがオブジェクトファイルを作成するとき、絶対アドレスは認識されていませんが、実行時リンカーはすべてのメモリーセグメントのアドレスを認識しており、したがって、これらのメモリーセグメントに存在するシンボルの絶対アドレスを計算できます。

プログラムがシンボルの絶対アドレスへの直接アクセスを必要とする場合、このシンボルには大域オフセットテーブルエントリが存在します。実行可能ファイルと共有オブジェクトには別個の大域オフセットテーブルが存在するので、シンボルのアドレスはいくつかのテーブルに現れることがあります。実行時リンカーは、プロセスイメージのコードに制御を与える前に大域オフセットテーブルのすべての再配置を処理します。したがって、実行時に絶対アドレスが使用可能になります。

テーブルのエントリ 0 は、`_DYNAMIC` シンボルで参照される動的構造体のアドレスを保持するために予約されています。したがって、実行時リンカーなどのプログラムは、再配置エントリを処理していなくても自身の動的構造体を見つけることができます。このことは、実行時リンカーにとって特に重要です。なぜなら、実行時リンカーは他のプログラムに頼ることなく自身を初期化してメモリーイメージを再配置しなければならないからです。

システムは、異なるプログラムの同じ共有オブジェクトに対して、異なるメモリーセグメントアドレスを与えることがあります。さらに、システムはプログラムを実行するごとに異なるライブラリアドレスを与えることさえあります。しかし、プロセスイメージがいったん作成されると、メモリーセグメントのアドレスは変更されません。プロセスが存在する限り、そのプロセスのメモリーセグメントは固定仮想されたアドレスに存在します。

大域オフセットテーブルの形式と解釈は、プロセッサに固有です。SPARC プロセッサと IA プロセッサの場合、`_GLOBAL_OFFSET_TABLE_` シンボルは、テーブルにアクセスするために使用できます。64 ビットコードの場合、シンボルタイプは `Elf64_Addr` の配列です。

```
extern Elf32_Addr _GLOBAL_OFFSET_TABLE_[];
```

シンボル `_GLOBAL_OFFSET_TABLE_` は、`.got` セクションの中央に存在でき、その場合は、負の添字と負でない添字の両方がアドレスの配列になることができます。

プロシージャのリンクテーブル (プロセッサ固有)

大域オフセットテーブルは位置に依存しないアドレスの計算を絶対位置に変換するので、プロシージャのリンクテーブルは位置に依存しない関数呼び出しを絶対位置に変換します。リンカーは、ある 1 つの実行可能オブジェクトまたは共有オブジェクトから別の実行可能オブジェクトまたは共有オブジェクトへの実行転送 (関数呼び出しなど) を解決できません。したがって、リンカーはプログラム転送制御をプロシージャのリンクテーブルのエントリに与えます。

SPARC: プロシージャのリンクテーブル

SPARC アーキテクチャの場合、プロシージャのリンクテーブルは私用 (**private**) データに存在します。実行時リンカーは、宛先の絶対アドレスを判定し、これらの絶対アドレスに従ってプロシージャのリンクテーブルのメモリーイメージに変更を加えます。このようにして実行時リンカーは、位置からの独立性とプログラムのテキストの共有性を低下させることなくエントリを向け直します。実行可能ファイルと共有オブジェクトファイルには、別個のプロシージャのリンクテーブルが存在します。

最初の 4 つのプロシージャのリンクテーブルエントリは、予約されます。テーブルの各エントリは 3 ワード (12 バイト) を占めており、最後のテーブルエントリの後には `nop` 命令が続きます。64 ビット SPARC オブジェクトの場合、各エントリは 8 つの命令 (32 バイト) を占めており、32 バイト境界で整列されなければなりません (テーブル全体は 256 バイト境界で整列されなければなりません)。

再配置テーブルは、プロシージャのリンクテーブルに関連付けられています。 `_DYNAMIC` 配列の `DT_JMP_REL` エントリは、最初の再配置エントリの位置を与えます。再配置テーブルには、各プロシージャのリンクテーブルエントリに対して 1 つのエントリが同じ順番で存在します。最初の 4 つのエントリを除き、再配置タイプは `R_SPARC_JMP_SLOT` であり、再配置オフセットは関連付けられているプロシージャのリンクテーブルエントリの先頭バイトのアドレスを指定し、シンボルテーブルインデックスは該当するシンボルを参照します。

プロシージャのリンクテーブルの説明のため、表 7-48 は 4 つのエントリを示しています。最初の 2 つのエントリは予約されている最初の 4 つのエントリの内の 2 つであり、3 番目のエントリは `name1` に対する呼び出しであり、4 番目のエントリは `name2` に対する呼び出しです。この例では、`name2` のエントリがテーブルの最後のエントリであることを前提としており、後に続く `nop` 命令が示されています。左欄は、動的リンクが行われる前のオブジェクトファイルの命令を示しています。右欄は、実行時リンカーがプロシージャのリンクテーブルエントリを修正することによりとり得る方法を示しています。

表 7-48 SPARC: プロシージャのリンクテーブルの例

オブジェクトファイル	メモリーセグメント
.PLT0: unimp unimp unimp	.PLT0: save %sp,-64,%sp call runtime-linker nop
.PLT1: unimp unimp unimpPLT1: .word identification unimp unimp ...
...	...
.PLT101: sethi (-.PLT0),%g1 ba,a .PLT0 nop	.PLT101: sethi (-.PLT0),%g1 sethi %hi(name1),%g1 jmp1 %g1+%lo(name1),%g0
.PLT102: sethi (-.PLT0),%g1 ba,a .PLT0 nop	.PLT102: sethi (-.PLT0),%g1 sethi %hi(name2),%g1 jmp1 %g1+%lo(name2),%g0
nop	nop

実行時リンカーとプログラムは、以下の手順に従ってプロシージャのリンクテーブル内のシンボル参照を協調して解決します。ただし、以下に記述されている手順は、単に説明用のためのものです。実行時リンカーの正確な実行時動作については、記述されていません。

1. 実行時リンカーは、プログラムのメモリーイメージを最初に作成するとき、プロシージャのリンクテーブルの初期エンタリに、実行時リンカー自身のルーチンの1つに制御を渡すように変更を加える。実行時リンカーはまた、識別情報 (identification) を2番目のエンタリに格納する。リンカー自身のルーチンが制御を受け取った時、このワードを調べることで、このルーチンを呼び出したオブジェクトを見つけることができる
2. 他のすべてのプロシージャのリンクテーブルエンタリは、最初は先頭エンタリに渡される。これで、実行時リンカーは各テーブルエンタリの最初の実行時に制御を取得する。たとえば、プログラムが name1 を呼び出すと、制御が .PLT101 に渡される
3. sethi 命令は、現在のプロシージャのリンクテーブルエンタリと最初のプロシージャのリンクテーブルエンタリの距離を計算する。つまり、.PLT101 と .PLT0 の距離を計算する。この値は、%g1 レジスタの最上位 22 ビットを占める。この例では、実行時リンカーが制御を受け取ると、%g1 には 0x12f000 が格納される

4. 次に、`ba,a` 命令が `.PLT0` にジャンプして、スタックフレームを作成し、実行時リンカーを呼び出す
5. 実行時リンカーは、識別情報の値を使うことによってオブジェクトのデータ構造体 (再配置テーブルを含む) を取得する
6. 実行時リンカーは、`%1` 値をシフトしプロシージャのリンクテーブルエントリのサイズで除算することで、`name1` の再配置エントリのインデックスを計算する。再配置エントリ 101 には `R_SPARC_JMP_SLOT` が存在し、オフセットは `.PLT101` のアドレスを指定し、シンボルテーブルインデックスは `name1` を参照。したがって、実行時リンカーはシンボルの実際の値を取得し、スタックを戻し、プロシージャのリンクテーブルエントリに変更を加え、本来の宛先に制御を渡す

実行時リンカーは、メモリーセグメント欄に示された命令シーケンスを作成しなければならないということはないのですが、作成することがあります。もし作成した場合は、いくつかの点についてより詳しい説明が必要です。

- コードを再入可能にするためには、プロシージャのリンクテーブルの命令は、特定の順番で変更が加えられる。実行時リンカーが関数のプロシージャのリンクテーブルエントリを修正中に信号が到達した場合、信号処理コードは、予想可能であり (かつ正しい) 結果を与える元の関数を呼び出すことができなければならない
- 実行時リンカーは、2つのワードに変更を加えてエントリを変換する。実行時リンカーは、各ワードを自動的に更新する。再入可能性は、まず `nop` を `jmp1` 命令で上書きし、次に `ba,a` を `sethi` 命令で上書きすることで実現される。再入可能関数呼び出しがワードの2つの更新間に発生した場合、`jmp1` は `ba,a` 命令の遅延スロットに存在し、遅延命令を取り消す。したがって、実行時リンカーは再び制御を取得す。実行時リンカーに対する両方の呼び出しで、同じプロシージャのリンクテーブルエントリに変更が加えられますが、これらの変更は互いに干渉しない
- プロシージャのリンクテーブルエントリの最初の `sethi` 命令は、1つ前のエントリの `jmp1` 命令の遅延スロットを埋める。`sethi` は `%1` レジスタの値を変更するが、以前の内容を捨てても問題はない
- 変換後、最後のプロシージャのリンクテーブルエントリ (前述した `.PLT102`) は、`jmp1` に対して遅延命令を必要とする。要求されている後続の `nop` は、この遅延スロットを埋める

`LD_BIND_NOW` 環境変数は、動的リンクの動作を変更します。この環境変数の値が空文字列以外の場合、実行時リンカーは、プログラムに制御を渡す前に `R_SPARC_JMP_SLOT` 再配置エントリ (プロシージャのリンクテーブルエントリ) を

処理します。LD_BIND_NOW が空文字列の場合、実行時リンカーは、各テーブルエントリの最初の実行時にリンクテーブルエントリを評価します。

IA: プロシージャのリンクテーブル

IA アーキテクチャの場合、プロシージャのリンクテーブルは共有テキストに存在しますが、私用大域オフセットテーブルのアドレスを使用します。実行時リンカーは、宛先の絶対アドレスを判定し、これらの絶対アドレスに従って大域オフセットテーブルのメモリーイメージに変更を加えます。このようにして実行時リンカーは、位置からの独立性とプログラムのテキストの共有性を低下させることなくエントリを向け直します。実行可能ファイルと共有オブジェクトファイルには、別個のプロシージャのリンクテーブルが存在します。

表 7-49 IA: プロシージャのリンクテーブルの例

```
.PLT0: pushl got_plus_4
      jmp  *got_plus_8
      nop; nop
      nop; nop
.PLT1: jmp  *name1_in_GOT
      pushl $offset
      jmp  .PLT0@PC
.PLT2: jmp  *name2_in_GOT
      pushl $offset
      jmp  .PLT0@PC
...

.PLT0: pushl 4(%ebx)
      jmp  *8(%ebx)
      nop; nop
      nop; nop
.PLT1: jmp  *name1@GOT(%ebx)
      pushl $offset
      jmp  .PLT0@PC
.PLT2: jmp  *name2@GOT(%ebx)
      pushl $offset
      jmp  .PLT0@PC
...
```

実行時リンカーとプログラムは、以下の手順に従ってプロシージャのリンクテーブル内と大域オフセットテーブル内のシンボル参照を協調して解決します。

1. 実行時リンカーは、プログラムのメモリーイメージを最初に作成するとき、大域オフセットテーブルの 2 番目と 3 番目のエントリに特殊値を設定する。これらの値については、以下の手順で説明する

2. プロシージャのリンクテーブルが位置に依存していない場合、大域オフセットテーブルのアドレスは、`%ebx` に存在しなければならない。プロセスイメージにおける各共有オブジェクトファイルには自身のプロシージャのリンクテーブルが存在しており、制御は同じオブジェクトファイル内からのみプロシージャのリンクテーブルエントリに渡される。したがって、呼び出し側関数は、プロシージャのリンクテーブルエントリを呼び出す前に、大域オフセットテーブルベースレジスタをセットしなければならない
3. たとえば、プログラムが `name1` を呼び出すと、制御が `.PLT1` に渡される
4. 最初の命令は、`name1` の大域オフセットテーブルエントリのアドレスにジャンプする。大域オフセットテーブルは最初は、後続の `pushl` 命令のアドレスを保持する (`name1` の実アドレスは保持しない)
5. したがって、プログラムは再配置オフセット (`offset`) をスタックにプッシュする。再配置オフセットは、再配置テーブルへの 32 ビットの負ではないバイトオフセット。指定された再配置エントリには `R_386_JMP_SLOT` が存在しており、オフセットは、前の `jmp` 命令で使用された大域オフセットテーブルエントリを指定する。再配置エントリにはシンボルテーブルインデックスも存在しており、実行時リンカーはこれを使って参照されたシンボル `name1` を取得する
6. プログラムは、再配置オフセットをプッシュした後、`.PLT0` (プロシージャのリンクテーブルの先頭エントリ) にジャンプする。`pushl` 命令は、2 番目の大域オフセットテーブルエントリ (`got_plus_4` または `4(%ebx)`) の値をスタックにプッシュして、実行時リンカーに 1 ワードの識別情報を与える。プログラムは次に、3 番目の大域オフセットテーブルエントリ (`got_plus_8` または `8(%ebx)`) のアドレスにジャンプして、実行時リンカーにジャンプする
7. 実行時リンカーはスタックを戻し、指定された再配置エントリを調べ、シンボルの値を取得し、`name1` の実際のアドレスを大域オフセットテーブルエントリに格納し、そして宛先にジャンプする
8. その後のプロシージャのリンクテーブルエントリに対する実行は、`name1` に直接渡される (実行時リンカーの再呼び出しは行われない)。これは、`.PLT1` における `jmp` 命令は、`pushl` 命令にはジャンプする代わりに、`name1` にジャンプするため

`LD_BIND_NOW` 環境変数は、動的リンク動作を変更します。この環境変数の値が空文字列以外の場合、実行時リンカーは、プログラムに制御を渡す前に

`R_386_JMP_SLOT` 再配置エントリ (プロシージャのリンクテーブルエントリ) を処理します。`LD_BIND_NOW` が空文字列の場合、実行時リンカーは、各テーブルエントリの最初の実行時にリンクテーブルエントリを評価します。

ハッシュテーブル

Elf32_Word オブジェクトまたは Elf64_Word オブジェクトのハッシュテーブルは、シンボルテーブルアクセスをサポートしています。ハッシュが関連付けられているシンボルテーブルは、ハッシュテーブルのセクションヘッダーの `sh_link` エントリに指定されます (表 7-15 を参照)。ハッシュテーブルの構造についての説明をわかりやすくするためにラベルを図 7-11 に示します。ただし、ラベルは仕様の一部ではありません。

<code>nbucket</code>
<code>nchain</code>
<code>bucket [0]</code>
<code>...</code>
<code>bucket [nbucket - 1]</code>
<code>chain [0]</code>
<code>...</code>
<code>chain [nchain - 1]</code>

図 7-11 シンボルハッシュテーブル

`bucket` 配列には `nbucket` 個のエントリが存在し、`chain` 配列には `nchain` 個のエントリが存在します。インデックスは 0 から始まります。`bucket` と `chain` には、シンボルテーブルインデックスを保持します。連鎖テーブルエントリは、シンボルテーブルに対応しています。シンボルテーブルエントリ数は、`nchain` に等しくなければなりません。したがって、シンボルテーブルインデックスは、連鎖テーブルエントリも適用できます。

ハッシュ関数はシンボル名を受け取り、`bucket` インデックスの計算に使用できる値を返します。したがって、ハッシュ関数が名前に対して値 x を返すと、`bucket [x%nbucket]` は、シンボルテーブルと連鎖テーブルの両方にインデックス y を与えます。シンボルテーブルエントリが、求めるシンボルテーブルエントリでなかった場合、`chain[y]` は、同じハッシュ値が存在する次のシンボルテーブルエントリを与えます。

`chain` リンクから、求める名前のシンボルテーブルエントリを探すことができます。`chain` リンクの最後には `STN_UNDEF` が設定されます。

```
unsigned long
elf_Hash(const unsigned char *name)
{
```

```
unsigned long h = 0, g;

while (*name)
{
    h = (h << 4) + *name++;
    if (g = h & 0xf0000000)
        h ^= g >> 24;
    h &= -g;
}
return h;
}
```

初期設定関数および終了関数

実行時リンカーがプロセスイメージを作成し、再配置を行なった後、各共有オブジェクトと実行可能ファイルにはいくつかの初期設定関数を実行する機会があります。共有オブジェクトの初期設定はすべて、実行可能ファイルが制御を得る前に実行されます。

同様に、動的オブジェクトには終了関数が存在できます。終了関数は、元となるプロセスが終了手順を開始した後、またはそのオブジェクトが `dlclose(3DL)` により実行プロセスから削除された場合、`atexit(3C)` の機構で実行されます。詳細は、`atexit(3C)` のマニュアルページを参照してください。

動的オブジェクトは、表 7-43 に記述されているように、動的構造内の `DT_INIT_ARRAY/DT_INIT` および `DT_FINI_ARRAY/DT_FINI` エントリを介して初期設定関数および終了関数を指定します。

初期設定および終了についての詳細は、38ページの「初期設定および終了セクション」および 84ページの「初期設定および終了ルーチン」を参照してください。

注 - `atexit(3C)` 終了処理は通常は行われますが、プロセス終了時に実行されることは保証されません。たとえば、プロセスが `_exit()` を呼び出した場合、またはプロセスが捕捉もされず無視もされない信号を受け取って終了した場合、プロセスは終了処理を実行しません。

mapfile のオプション

概要

リンカーは、入力配置可能オブジェクトの入力セクションを出力ファイルオブジェクトのセグメントに、自動的にかつ効率的に対応付けします。対応する mapfile を `-M` オプションに指定すれば、リンカーの初期値の対応付けの方法を変更できます。

この mapfile オプションを使えば、以下のことができます。

- セグメントを宣言し、セグメント型、アクセス権、アドレス、長さ、および整列などセグメント属性の値を指定する
- 特定のセグメントに対応付けするのにセクションに必要な属性値を指定することにより (属性とは、セクション名、セクション型、およびアクセス権)、また必要ならば、入力セクションをどのオブジェクトファイルから取ってくるかを指定することにより、入力セクションのセグメントへの対応付けを制御する
- 指定したセグメントのサイズに等しい値に (リンカーが) 割り当てた、オブジェクトファイルから参照できる大域絶対シンボルを宣言する

mapfile オプションによって、ifiles (以前のリンカーで使用できた機能で、コマンド言語による指示を使用する) のユーザーは、mapfiles に移行できます。ただし、以前の ifiles で利用できた機能で、前述の説明に含まれていないものすべては、mapfile オプションでは利用できません。

注 - `mapfile` オプションを使う場合、実行できない `a.out` ファイルを簡単に作れてしまうことに留意してください。リンカーは、`mapfile` オプションなしでも、正しい `a.out` を作成することができます。`mapfile` オプションは、システムプログラム用のもので、アプリケーションプログラムでの使用を意図したものではありません。

mapfile オプションの使い方

`mapfile` オプションを使う場合、次のことを行います。

- `mapfile` 指示をファイル、たとえば `mapfile` に入力する
- リンカーのコマンド行に、`-M mapfile` を指定する

`mapfile` が現在のディレクトリにない場合、フルパス名を入れます。初期値の検索パスは存在しません。

mapfile の構造と構文

以下の 4 つの基本的な指示を `mapfile` に入力できます。

- セグメント宣言
- 対応付け指示
- サイズシンボル宣言
- ファイル制御指示

それぞれの指示は複数の行にまたがることができ、最後にセミコロンを付ければ、いくつでも空白 (改行を含む) を入れることができます。`mapfile` に指示をまったく入れないことも、複数入れることもできます (指示をまったく入れない場合、リンカーは `mapfile` を無視し、自らの初期値を使います)。

通常、セグメント宣言の後には、対応付け指示がきます。つまり、セグメントを宣言し、次にセクションがそのセグメントの一部になる条件を定義するわけです。対応付け先のセグメントを最初に宣言しないで、対応付け指示あるいはサイズシンボル宣言を入力した場合、後で説明する組み込みのセグメント以外のセグメントに

は、以下で説明するように初期値の属性が与えられます。この場合このようなセグメントは「暗示的に宣言されたセグメント」になります。

サイズシンボル宣言、およびファイル制御指示は、`mapfile` のどこにでも入れることができます。

以後の節では、それぞれの指示について説明します。すべての構文説明について、次の表記が適用されます。

- 固定幅の文字のエントリ、すべてのコロン、セミコロン、等符号、@ 記号は、そのままの文字を入力する
- 「斜体文字」で示されたエントリはすべて、適切なもので置き換える
- `{... }*` は、「無しまたはそれ以上」を意味する
- `{... }+` は、「1つまたはそれ以上」を意味する
- `[...]` は、「任意指定」を意味する
- `section_names` と `segment_names` は、C 識別子と同じ規則に従い、ピリオド (.) は文字として扱われる (たとえば、`.bss` は正当な名前)
- `section_names`、`segment_names`、`file_names`、および `symbol_names` には大文字と小文字の区別があり、それ以外のものには大文字と小文字の区別は無い
- 空白文字 (あるいは改行文字) は、数字の前および名前や値の間以外はどこにでも入れられる
- # で始まり改行で終わるコメントは、空白文字を入れることができる場所であれば、どこにでも入れられる

セグメントの宣言

セグメントの宣言により、`a.out` に新しいセグメントを作ったり、既存のセグメントの属性値を変更したりできます (既存のセグメントとは、以前に定義したもの、あるいは以下に述べる 3 つの組み込みセグメントの 1 つのことです)。

セグメントの宣言は以下の構文で行います。

```
segment_name = {segment_attribute_value}*;
```

各 `segment_names` について、任意の数の `segment_attribute_values` を任意の順序で指定でき、それぞれは空白文字で区切ります (セグメント属性ごとに、1 つの値だけ指定できます)。セグメント属性と、それらの有効な値は表 8-1 に示すとおりです。

表 8-1 mapfile セグメント属性

属性	値
<code>segment_type</code>	LOAD NOTE STACK
<code>segment_flags</code>	?[E] [N] [O] [R] [W] [X]
<code>virtual_address</code>	<i>Vnumber</i>
<code>physical_address</code>	<i>Pnumber</i>
長さ	<i>Lnumber</i>
丸め	<i>Rnumber</i>
整列	<i>Anumber</i>

3つの組み込みセグメントが存在し、以下のような初期値の属性値を持っています。

- テキスト (LOAD、?RX、`virtual_address` と `physical_address` と長さは指定なし、CPU 固有の整列値)
- データ (LOAD、?RWX、`virtual_address` と `physical_address` と長さは指定なし、CPU 固有の整列値)
- 注釈 (NOTE)

リンカーは、`mapfile` を読み取る前に、これらのセグメントが宣言されたように動作します。詳細は、336ページの「`mapfile` オプションの初期値」を参照してください。

セグメント宣言を入力する場合、以下のことに注意してください。

- 数字には、C 言語と同じ形式で、16 進数、10 進数、あるいは 8 進数が使えます
- V、P、L、R、あるいは A と数字の間には空白文字を入れてはいけません
- `segment_type` 値は、LOAD、NOTE、または STACK のいずれか
- `segment_type` の初期値は、LOAD

- `segment_flags` 値は、R は読み取り可能、w は書き込み可能、x は実行可能、o は順番です。疑問符 ? と `segment_flags` 値を構成する個々のフラグの間には、空白文字を入れてはいけない
- LOAD セグメントの `segment_flags` 値は、初期値で RWX になる
- NOTE セグメントには、`segment_type` 以外のセグメント属性値は割り当てられない
- タイプ STACK の 1 つの `segment_type` がアクセス許可され、`segment_flags` から選択されたセグメントのアクセス要求のみが指定できる
- 暗示的に宣言されたセグメントでは、`segment_type` 値は LOAD、`segment_flags` 値は RWX、`virtual_address` と `physical_address` と整列値は初期値、そして長さは無制限になる

注 - リンカーは、1 つ前のセグメントの属性値に基づいて、現在のセグメントのアドレスや長さを計算します。また、暗示的に宣言されたセグメントがデフォルトで「長さ制限なし」になっていても、マシンのメモリー限界は適用されます。

- LOAD セグメントには、`virtual_address` 値、`physical_address` 値、最大セグメント長値のいずれかまたはすべてを明示的に指定できる
- セグメントに ? の `segment_flags` 値があつて後に何も無い場合、値は読み取り不可、書き込み不可、および実行不可になる
- 整列値は、セグメントの最初の仮想アドレスを計算する際に使われる。この整列は、整列指定されたセグメントにだけ影響する。その他のセグメントは、その整列が変更されない限り、デフォルトの整列が使われる
- `virtual_address`、`physical_address`、あるいは長さの属性値のいずれかが設定されていない場合、リンカーは `a.out` を作成する際に、これらの値を計算する
- セグメントに対して整列値が指定されていない場合、組み込みの初期値に設定される (初期値は CPU により異なり、カーネルのバージョンによっても異なることがある。これらの数字については、適切な資料で確認のこと)
- `virtual_address` と整列値の両方がセグメントに対して指定されている場合、`virtual_address` の方が優先される
- `virtual_address` 値がセグメントに対して指定されている場合、プログラムヘッダーの整列フィールドには、初期値の整列値が設定される

- 丸め値がセグメントに対して設定されている場合、そのセグメントの仮想アドレスは与えられた値に一致する次のアドレスに丸められる。この値は、指定の対象となるセグメントにしか効力はない。値が入力されないと、丸めは行われない

?E フラグにより、空のセグメントが作れます。これは、関連するセクションがないセグメントです。このセグメントは実行プログラムについてのみ指定でき、LOAD 型で、長さおよび整列が指定されていなければなりません。この種類のセグメントは複数定義することもできます。

?N フラグにより、ELF ヘッダー、および任意のプログラムヘッダーを最初の読み込み可能なセグメントの一部分として含めるかどうかを制御できます。初期値では、ELF ヘッダーおよびプログラムヘッダーは、これらのヘッダーの情報が対応付けられたイメージ内 (一般に実行時リンカーによって対応付けられる) で使われるため、最初のセグメントに含まれます。?N オプションを使用すると、イメージの仮想アドレス計算が最初のセグメントの最初の「セクション」で開始します。

?O フラグにより、最終的な再配置可能オブジェクト、実行可能ファイル、あるいは共有オブジェクトのセクションの順序を制御できます。このフラグは、コンパイラの -xF オプションと合わせて使うようになっています。ファイルを -xF オプションを使ってコンパイルすると、そのファイル内の各関数が、.text セクションと同じ属性を持つ別個のセクションに置かれます。これらのセクションは、.text%function_name (function_name は関数名) という名前になります。

たとえば、main()、foo()、および bar() の 3 つの関数を持つファイルを、-xF オプションを使ってコンパイルすると、3 つの関数のテキストが

.text%main、.text%foo、および .text%bar という名前のセクションにあるオブジェクトファイルが作成されます。-xF オプションは 1 つのセクションに 1 つの関数を割り当てるので、セクションの順番を制御するために ?O フラグを使うと、実際には関数の順番を制御することになります。

次のユーザー定義の mapfile を検討します。

```
text = LOAD ?RXO;  
text: .text%foo;  
text: .text%bar;  
text: .text%main;
```

ソースファイルの関数定義の順序が、main、foo、および bar の場合、最終的な実行プログラムには foo、bar、および main の順序で関数が含まれます。

同じ名前の静的関数を対象とする場合、ファイル名も指定する必要があります。?O フラグを指定すると、mapfile で指定されたとおりにセクションの順序付けが行わ

れます。たとえば、静的関数 `bar()` がファイル `a.o` および `b.o` にあって、ファイル `a.o` の関数 `bar()` を、ファイル `b.o` の関数 `bar()` の前に置く場合、`mapfile` のエントリは次のようになります。

```
text: .text%bar: a.o;
text: .text%bar: b.o;
```

次の構文を使用することも可能ですが、ファイル `a.o` の関数 `bar()` が、ファイル `b.o` の関数 `bar()` の前に置かれることが保証されません。2 番目の形式は、結果が予測できないため、お勧めしません。

```
text: .text%bar: a.o b.o;
```

注 - `virtual_address` が指定されている場合、セグメントはその仮想アドレスに置かれます。これは、システムカーネルに対して、正しい結果を出します。 `exec(2)` を介して開始するファイルの場合、この方法では、セグメントにそのページ境界に対応する正しいオフセットがないため、間違った `a.out` ファイルが作成されることとなります。

対応付け指示

対応付け指示は、入力セクションをどのように出力セグメントに対応付けするかをリンカーに伝えます。基本的には、対応付け先のセグメントの名前を指定し、名前を指定したセグメントに対応付けするためにセクションの属性をどうすべきかを指定します。特定のセグメントに対応付けするためにセクションが持っていなければならないセクション属性値 (`section_attribute_values`) は、そのセグメントの「エントランス基準」と呼ばれます。`a.out` の指定したセグメントにセクションを置くには、セクションはセグメントのエントランス基準に正確に合致していなければなりません。

対応付け指示には以下のような構文があります。

```
segment_name : {section_attribute_value}* [: {file_name}+];
```

セグメント名 (`segment_name`) に対して、任意の数のセクション属性値 (`section_attribute_values`) を任意の順序で指定し、それぞれは空白文字で区切ります (セクション属性ごとに 1 つの値だけ指定できます)。ファイル名 (`file_name`) を指定して、特定の `.o` ファイルからセクションを持ってこなければ

ならないというように指定することもできます。セクション属性とその有効値は表 8-2 に示すとおりです。

表 8-2 セクション属性

セクション属性	値
<code>section_name</code>	任意の有効なセクション名
<code>section_type</code>	<code>\$PROGBITS</code> <code>\$SYMTAB</code> <code>\$STRTAB</code> <code>\$REL</code> <code>\$RELA</code> <code>\$NOTE</code> <code>\$NOBITS</code>
<code>section_flags</code>	?[!A][!W][!X]

対応付け指示を入力する場合、以下の点に注意してください。

- 上に挙げた `section_types` から 1 つの値を選択する。上記の `section_types` は組み込まれているものです。`section_types` の詳細については、212ページの「セクション」を参照
- `section_flags` 値は、A は割り当て可能、W は書き込み可能、X は実行可能。個々のフラグの前に感嘆符 (!) がついている場合、リンカーは、フラグが設定されていないことを確認する。`section_flags` 値を構成する疑問符、感嘆符、および個々のフラグの間には空白文字を入れてはいけない
- ファイル名には、ファイル名として正当な名前を指定できる。また、`/usr/lib/usr/libc.a(printf.o)` のように「アーカイブ名(要素名)」の形式で指定することもできます。さらに、ファイル名に `*filename` の形式で指定することもできる
- ファイル名が `*filename` の形式になっている場合、リンカーはコマンド行で与えられた行からファイル名に `basename(1)` と同等の処理を行い、得られたファイル名を使って指定された `filename` との一致を調べる。言い換えれ

ば、mapfile で指定する filename は、コマンドラインで与えられるファイル名の最後の部分だけが合致すればよい(334ページの「対応付けの例」を参照)

- リンク編集の際に -l オプションを使っていて、-l オプションの後のライブラリが現在のディレクトリにある場合、mapfile 内のライブラリと一致させるために mapfile 内のライブラリの前に ./ (あるいはパス名全体) を付ける必要がある
- 特定の出力セグメントについて複数の指示行を指定できる。たとえば、以下のような一連の指示を行うことができる

```
S1 : $PROGBITS;  
S1 : $NOBITS;
```

1つのセグメントに対して複数の対応付け命令行を指示することは、複数のセクション属性値を指定するための唯一の方法です。

- 1つのセクションは複数のエントランス基準に合致することがある。その場合、mapfile で最初にエントランス基準が合致したセグメントが使われる。たとえば、mapfile が以下のようにになっているとする

```
S1 : $PROGBITS;  
S2 : $PROGBITS;
```

この場合、\$PROGBITS セクションは、セグメント S1 に対応付けられます。

セグメント内セクションの順序

以下のような表記を使うことにより、セグメント内にセクションを配置する順序を指定できます。

```
segment_name | section_name1;  
segment_name | section_name2;  
segment_name | section_name3;
```

上記の形式で指定されたセクションは、すべての名前なしセクションの前に、mapfile に列挙されている順序で配置されます。

サイズシンボル宣言

サイズシンボル宣言を使って、指定したセグメントのサイズをバイトで示す大域絶対シンボルを定義できます。このシンボルは、オブジェクトファイル内で参照できます。サイズシンボルは次の構文で宣言します。

```
segment_name @ symbol_name;
```

symbol_name (シンボル名) には、任意の正当な C 識別子が入ります。ただし、リンカーは、symbol_name の構文の確認は行いません。

ファイル制御指示

ファイル制御指示により、共有オブジェクト内のどのバージョンの定義をリンク編集時に使用可能にするかを指定できます。ファイル制御構文で宣言します。

```
shared_object_name - version_name [ version_name ... ];
```

version_name (バージョン名) には、指定した shared_object_name (共有オブジェクト名) 内のバージョン定義名が入ります。バージョン制御の詳細については、152ページの「バージョン結合の指定」を参照してください。

対応付けの例

以下はユーザー定義の mapfile の例です。左の数字は、説明のためにつけたものです。実際には、数字の右の情報だけが、mapfile に含まれます。

例 8-1 ユーザー定義の mapfile

```
1.   elephant : .data : peanuts.o *popcorn.o;
2.   monkey : $PROGBITS ?AX;
3.   monkey : .data;
4.   monkey = LOAD V0x80000000 L0x4000;
5.   donkey : .data;
6.   donkey = ?RX A0x1000;
7.   text = V0x80008000;
```

4つの別々のセグメントがこの例では扱われています。暗黙の内に宣言されたセグメント elephant (1行目) は、ファイル peanuts.o および popcorn.o からすべ

での .data セクションを受け取ります。*popcorn.o は、リンカーに与えられるすべての popcorn.o ファイルと一致します。ファイルは、現在のディレクトリにある必要はありません。他方、/var/tmp/peanuts.o がリンカーに提供された場合、これには * が前に付かないため peanuts.o とは一致しません。

暗黙の内に宣言されたセグメント monkey (2 行目) は \$PROGBITS および割当て可能な実行プログラム (?AX) の両方になっているすべてのセクション、さらに .data (3 行目) という名前のすべてのセクション (セグメント elephant に入らなかったもの) を受け取ります。別の行で section_name に section_type と section_flags の値が指定されているので、monkey セグメントに入る .data セクションが、\$PROGBITS あるいは割当て可能な実行プログラムである必要はありません (and の関係は、2 行目の \$PROGBITS and ?AX で示されるように、同じ行の属性の間に存在します。or の関係は、2 行目の \$PROGBITS ?AX or 3 行目の .data のように、複数の行にまたがる同じセグメントの属性の間に存在します)。

monkey セグメントは、segment_type が LOAD、segment_flags が RWX、virtual_address と physical_address は無指定、長さや整列の値は、初期値として 2 行目で暗黙の内に宣言されています。4 行目では、monkey の segment_type 値は LOAD に (segment_type 属性は変わっていないので、警告は出ません)、virtual_address 値は 0x80000000 に、最大長値は 0x4000 に設定されています。

5 行目では、donkey セグメントを暗黙の内に宣言しています。エンタランス基準は、このすべての .data セクションをこのセグメントに振り向けるようになっていきます。実際には、3 行目の monkey のエンタランス基準はこれらのセクションのすべてを取り込むので、このセグメントに入るセクションはありません。6 行目では、segment_flags 値は ?RX に設定され、整列値は 0x1000 に設定されています (これらの属性値の両方が変わったので警告が出ます)。

7 行目では、テキストセグメントの virtual_address 値を 0x80008000 に設定します。

ユーザー定義の mapfile の例は、説明のために警告を出すように設計されています。次の例では命令の順序を変更して警告を避けています。

```
1.    elephant : .data : peanuts.o *popcorn.o;
4.    monkey = LOAD V0x80000000 L0x4000;
2.    monkey : $PROGBITS ?AX;
3.    monkey : .data;
6.    donkey = ?RX A0x1000;
5.    donkey : .data;
```

(続く)

```
7.    text = V0x80008000;
```

次の mapfile の例では、セグメント内セクションの順序付けを使っています。

```
1.    text = LOAD ?RXN V0xf0004000;
2.    text | .text;
3.    text | .rodata;
4.    text : $PROGBITS ?A!W;
5.    data = LOAD ?RWX R0x1000;
```

text セグメントおよび data セグメントが、この例では扱われています。1 行目では、text セグメントの virtual_address が 0xf0004000 になるように、またこのセグメントのアドレス計算の一部として ELF ヘッダーやプログラムヘッダーを含めないように宣言しています。2 行目と 3 行目では、セグメント内セクションの順序付けを有効にし、このセグメントでは .text セクションと .rodata セクションが最初の 2 つのセクションになるように指定しています。この結果、.text セクションは仮想アドレスが 0xf0004000 になり、その直後に .rodata セクションがきます。

text セグメントを構成するその他の \$PROGBITS セクションは、.rodata セクションの後にきます。5 行目では data セグメントを指定し、その仮想アドレスは 0x1000 バイト境界で始まらなければならないと指定しています。data を構成する最初のセクションも、ファイルイメージ内の 0x1000 バイト境界に存在します。

mapfile オプションの初期値

リンカーは、327ページの「セグメントの宣言」で説明したように、初期値の segment_attribute_values で 3 つの組み込みセグメント (text、data、および note)、および対応する初期値の対応付け命令を定義します。リンカーは初期値を提供するのに実際の mapfile は使いませんが、ここでは初期値の内容を mapfile に書かれているものとして、リンカーがユーザーの mapfile を処理する際にどのようなことが起こるかを説明します。

以下に示す例は、リンカーの初期値を mapfile として表現したものです。リンカーは、mapfile をすでに読み取ったかのように実行を開始します。その後でリ

ンカーは、mapfile を読み取ったり、あるいは初期値への追加や変更を行ったりします。

```
text = LOAD ?RX;
text : ?A!W;
data = LOAD ?RWX;
data : ?AW;
note = NOTE;
note : $NOTE;
```

mapfile の各セグメント宣言は読み取られる際、以下のようにセグメント宣言の既存リストと比較されます。

1. セグメントが mapfile に存在しておらず、同じ segment-type 値の別のセグメントが存在する場合、そのセグメントは、すべての同じ segment_type の既存セグメントの前に追加される
2. セグメントが読み取られたとき、既存の mapfile に同じ segment_type のセグメントがない場合、セグメントは、segment_type 値が以下の順序になるように追加される

INTERP

LOAD

DYNAMIC

NOTE

3. セグメントの segment_type が LOAD で、この LOAD 可能なセグメントに virtual_address 値を定義していた場合、セグメントは、virtual_address 値が定義されていない、あるいはより高い virtual_address 値の LOAD が指定されたセグメントの前、そしてそれよりも低い virtual_address 値のセグメントの後に置かれる

mapfile の各対応付け指示が読み取られる際、同じセグメントに対してすでに指定されたその他の対応付け指示の後 (かつ、そのセグメントのデフォルトの対応付け指示の前) に、指示が追加されます。

内部対応付け構造

ELF ベースのリンカーのもっとも重要なデータ構造の 1 つとして対応付け構造があります。上記で述べた初期値 mapfile に対応する初期対応付け構造は、コマンド

実行時にリンカーで使われます。mapfile オプションが使われる場合、リンカーは mapfile を解析して、特定の値を初期対応付け構造に追加または上書きします。

一般的な (ただし多少簡略化した) 対応付け構造を図 8-1 に示します。「エントランス基準」ボックスは初期値の対応付け指示の情報に対応しており、「セグメント属性記述子」ボックスは、初期値のセグメント宣言の情報に対応しています。「出力記述子」ボックスは、各セグメントの下に来るセクションの詳細な属性を示します。セクション自体は丸で囲んであります。

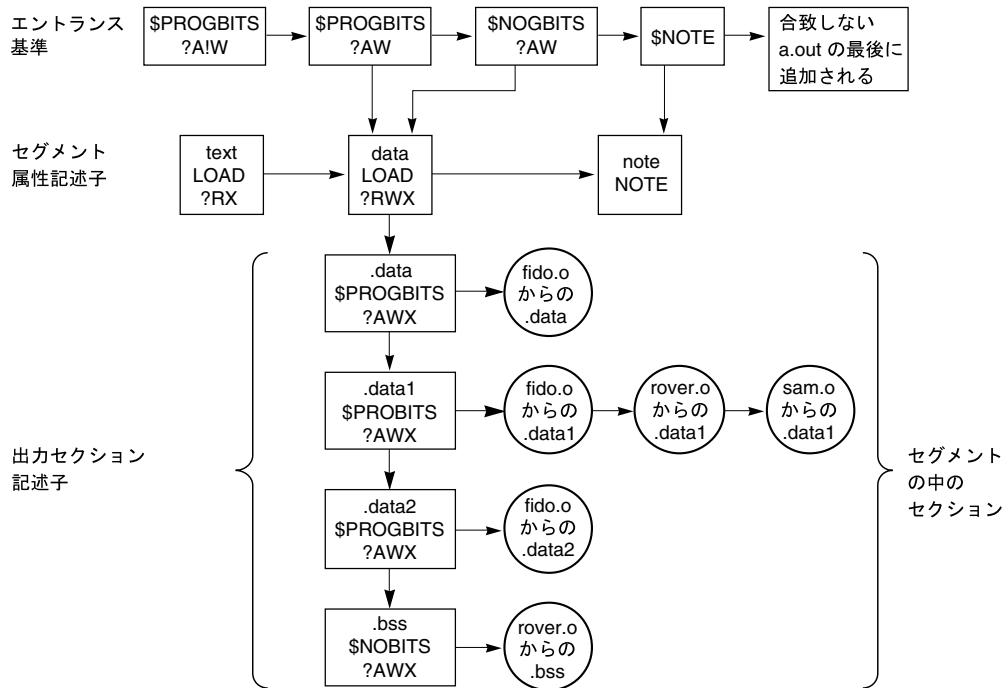


図 8-1 簡単な対応付け構造

リンカーはセクションをセグメントに対応付ける際に、以下の手順で行います。

1. セクションを読み取る際に、リンカーは合致するものを見つけるために、一連のエントランス基準を検査します。指定したすべての条件が合致する必要があります。

図 8-1 では、text セグメントに入るセクションの場合、section_type は \$PROGBITS、section_flags は ?A!W でなければなりません。エントランス基準では名前は指定されていないので、名前 .text は必要ありません。エントランス基準では実行ビットは何も指定されていないので、セクションは (section_flags 値の) X または !X のどちらかでもかまいません。

エンタランス基準に合致するものが見つからない場合、セクションはその他すべてのセグメントの後の a.out ファイルの最後に置かれます (この情報については、プログラムヘッダーエントリは作られません。詳細は、279ページの「プログラムヘッダー」を参照)。

2. セクションがセグメントの中に入る際に、リンカーは以下のようにそのセグメントの既存の一連の出力セクション記述子を検査します。

セクションの属性値が既存の出力セクション記述子の属性値とまったく合致する場合、セクションは出力セクション記述子に対応するセクションの列挙の最後に置かれます。

たとえば、section_name が .data1、section_type が \$PROGBITS、および section_flags が ?AWX のセクションは、図 8-1 の 2 番目のエンタランス基準ボックスにあてはまり、data セグメントに置かれます。セクションは 2 番目の出力セクション記述子ボックスにちょうど一致し

(.data1、\$PROGBITS、?AWX)、このボックスに対応する一連のセクションの最後に追加されます。fido.o、rover.o、および sam.o の .data1 セクションは、このことを示しています。

一致する出力セクション記述子が見つからず、同じ section_type の出力セクション記述子が他に存在する場合、セクションとして同じ属性値で新しい出力セクション記述子が作られ、そのセクションは新しい出力セクション記述子と対応づけられます。出力セクション記述子 (およびセクション) は、同じ section_type の最後の出力セクション記述子の後に置かれます。図 8-1 の .data2 セクションはこのやり方で置かれました。

示された section_type の出力セクション記述子が他に存在しない場合、新しい出力セクション記述子が作られ、セクションはそのセクション内に置かれます。

注 - 入力セクションにユーザー定義の section_type が設定されている場合 (つまり、212ページの「セクション」で説明しているように、SHT_LOUSER と SHT_HIUSER の間)、これは \$PROGBITS セクションとして扱われま
す。mapfile でこの section_type 値に名前を付ける方法はありませんが、これらのセクションは、エンタランス基準でその他の属性値指定 (section_flags、section_name) を使って付け直すことができます。

3. すべてのコマンド行で指定されたオブジェクトファイルとライブラリが読み取られた後、セグメントがセクションをまったく含まない場合、そのセグメントに対してはプログラムヘッダーエントリは作られません。

注 - \$SYMTAB、\$STRTAB、\$REL、および \$RELA 型の入力セクションは、リンカーにより内部で使用されます。これらの `section_type` を参照する指示は、リンカーで作った出力セクションしかセグメントに対応付けできません。

エラーメッセージ

警告

このカテゴリのエラーメッセージが出力される場合は、リンカーが停止したり、リンカーが実行可能な `a.out` を作成できないといったことはありません。次のような条件で、警告が発せられます。

- `physical_address`、`virtual_address`、あるいは長さの値が、LOAD セグメント以外のすべてのセグメントに対して指示されている場合 (指示は無視される)
- すでに存在するセグメントに対して、属性値を変更する宣言が行なわれた場合 (2 番目の宣言で指示した属性値が有効になる)
- 属性値 (`text` や `data` に対しては `segment_type` と `segment_flags`。note に対しては `segment_type`) が組み込みセグメントのいずれかに対して変更された場合
- 暗黙の内に宣言により作成されたセグメントの属性値 (`segment_type`、`segment_flags`、長さ、整列の一部または全部) が変更された場合。?O フラグが追加されただけの場合、属性値変更の警告は発せられない
- エントランス基準に合わない場合。?O フラグが有効になっていて、エントランス基準に合う入力セクションがない場合、警告が発せられる

致命的エラー

このカテゴリのエラーが発生した場合、その時点でリンカーの実行を停止します。以下の条件で致命的エラーが発生します。

- 対応付けファイルを開いたり読んだりできない場合
- 構文エラーが `mapfile` で発見された場合

注 - `file_name`、`section_name`、`segment_name`、あるいは `symbol_name` が 326ページの「`mapfile` の構造と構文」の規則に合わない場合、リンカーはこの条件が構文エラーを作成しない限り、エラーを返しません。たとえば、名前が特殊な文字で始まり、この名前が指示行の最初にある場合、リンカーはエラーを返します。名前が `section_name` の場合 (指示内に表示されます)、リンカーはエラーを返しません。

- 1つの命令行に複数の `segment_type`、`segment_flags`、`virtual_address`、`physical_address`、長さあるいは整列の値がある場合
 - `mapfile` で `interp` セグメントまたは `dynamic` セグメントを操作しようとした場合
-

注 - `interp` セグメントおよび `dynamic` セグメントは特殊な組み込みのセグメントで、これはどのようなやり方を使っても変更できません。

- セグメントが、長さ属性値で指定した長さよりも大きくなった場合
- ユーザー定義の `virtual_address` が原因で、セグメントが前のセグメントを上書きする場合
- 1つの命令行に複数の `section_name`、`section_type`、あるいは `section_flags` 値がある場合
- フラグおよびその要素 (たとえば、`A` と `!A`) が1つの指示行にある場合

提供される `mapfile`

`mapfile` のサンプルは、`/usr/lib/ld` ディレクトリにあります。

リンカーのクイックリファレンス

概要

この項では、簡単な概要を記載してあります。すなわち、この項は、最も一般的に使用されるリンカーのシナリオの「虎の巻」です (リンカーによって生成される出力モジュールの種類の詳細は、18ページの「リンク編集」を参照)。

ここに記載された例では、コンパイラドライバ `cc(1)` に提供されるリンカーのオプションを示しています。これは、リンカーを起動させる、最も一般的な機構です (27ページの「コンパイラドライバを使用する」を参照)。

リンカーは、入力ファイルの名前によって動作を変えることはありません。各ファイルは、開かれ、ファイルが必要とする処理の種類を判別するために検査が行われます (29ページの「入力ファイルの処理」を参照)。

`libx.so` の命名規約に従って命名された共有オブジェクトと、`libx.a` の命名規約に従って命名されたアーカイブライブラリは、`-l` オプションを使用して指定できます (33ページの「ライブラリの命名規約」を参照)。これにより、`-L` オプションを使用して指定できる検索パスに、より柔軟性を持たせることができます (35ページの「リンカーが検索するディレクトリ」を参照)。

リンカーは、基本的には、「静的」または「動的」の2つの方法のうちのいずれかで稼動します。

静的方法

この方法は、`-dn` オプションが使用された場合に選択されます。また、このモードを使用すると、再配置可能オブジェクトと静的実行プログラムを作成できます。この場合、再配置可能オブジェクトとアーカイブライブラリの入力形式だけが受け入れられます。`-l` オプションを使用すると、アーカイブライブラリが検索されます。

再配置可能オブジェクトの作成

- 次のように、`-dn` オプションと `-r` オプションを使用する

```
$ cc -dn -r -o temp.o file1.o file2.o file3.o .....
```

静的実行プログラムの作成

静的実行プログラムの使用は制限されています。静的実行プログラムには、通常、プラットフォーム固有な実装に依存した情報などが組み込まれ、これにより、他のプラットフォーム上で実行プログラムを実行することが制限されます。また、ほとんどの Solaris ライブラリは、`dlopen(3DL)` や `dlsym(3DL)` などの動的リンク機能に依存しています (81ページの「追加オブジェクトの読み込み」を参照)。これらの機能は、静的実行プログラムでは使用できません。

- `-dn` オプションは、次のように `-r` オプションを「指定せずに」使用する

```
$ cc -dn -o prog file1.o file2.o file3.o .....
```

注 `-a` オプションを使用して、静的実行プログラムの作成を指示できます。そして、`-dn` を指定して、`-r` を指定しない場合、`-a` が暗黙指定されます。

動的方法

これは、リンカーの標準の動作方法です。`-dy` オプションで明示的に指定することもできますが、`-dn` オプションを使用しない場合には、暗黙指定されます。

この場合、再配置可能オブジェクト、共有オブジェクト、およびアーカイブライブラリを指定できます。`-l` オプションを使用すると、ディレクトリ検索が実行されま

す。ここで、各ディレクトリは、共有オブジェクトを見つけるために検索され、そのディレクトリで共有オブジェクトが見つからない場合は、次にアーカイブライブラリが検索されます。`-B static` オプションを使用すると、アーカイブライブラリの検索だけに限定されます。(34ページの「共有オブジェクトとアーカイブとの混合体へのリンク」を参照)。

共有オブジェクトの作成

- `-G` オプションを使用する (`-dy` は省略時には暗黙指定されるため、指定する必要はない)
- 入力再配置可能オブジェクトは、位置非依存のコードから作成する必要がある。`-z text` オプションを使用して、この必要条件を実行する (124ページの「位置に依存しないコード」を参照)
- 共有オブジェクトの公開インタフェースを確立する。共有オブジェクトの外から見える大域シンボルを定義し、それ以外のすべてのシンボルはローカル範囲に隠蔽する。これは、`-M` オプションとともに `mapfile` を指定することによって定義できる。詳細は、付録 B を参照
- 将来アップグレードに対応できるように、共有オブジェクトにはバージョンを含む名前を使用する (158ページの「バージョンアップファイル名の管理」を参照)
- 生成される共有オブジェクトが他の共有オブジェクトに依存している場合は、`-z lazyload` オプションを使用してレイジー読み込みを指定する (82ページの「動的依存関係のレイジーローディング」を参照)
- 生成される共有オブジェクトが他の共有オブジェクトと依存関係があり、これらの共有オブジェクトが 32 ビットオブジェクト用 `/usr/lib` または 64 ビットオブジェクト用 `/usr/lib/64` に存在しない場合は、`-R` オプションを使用して、そのパス名を出力ファイル内に記録する (113ページの「依存関係を持つ共有オブジェクト」を参照)
- 自己完結型の共有オブジェクトは、最も柔軟性が高く、オブジェクトが必要とするものすべてを自身が提供している場合に作成される。`-z defs` オプションを使用して、自己完結していることを確認する (49ページの「共有オブジェクトの生成」を参照)
- すべての再配置を 1 つの `.SUNW_reloc` セクションに配置することによって、再配置処理を最適化する。`-z combrelloc` オプションを使用する
- 不必要な依存性を排除する。`-z ignore` オプションを使用するとリンカーは、参照されたオブジェクトに対する依存性だけを記録する

- このオブジェクトや関連する依存関係で割り込みシンボルが使用されない場合は、`-B direct` を使って直接結合情報を確立する (60ページの「外部結合」を参照)。

次に、説明したオプションを組み合わせた例を示します。

```
$ cc -c -o foo.o -Kpic foo.c
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs -B direct -z lazyload \
-z combrelloc -z ignore -R /home/lib foo.o -L. -lbar -lc
```

- 生成される共有オブジェクトが、他のリンク編集への入力として使用される場合、`-h` オプションを使用して、その中に共有オブジェクトの実行名を記録する (109ページの「共有オブジェクト名の記録」を参照)
- 共有オブジェクトを、バージョンを含まない名前のファイルにシンボリックリンクして、その共有オブジェクトをコンパイル環境でも使用できるようにする (158ページの「バージョンアップファイル名の管理」を参照)

次の例は、説明したオプションを組み合わせたものです。

```
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs -B direct -z lazyload \
-z ignore -R /home/lib -h libfoo.so.1 foo.o -L. -lbar -lc
$ ln -s libfoo.so.1 libfoo.so
```

- 共有オブジェクトのパフォーマンスへの影響を考慮する。共有性を最大限にし (126ページの「共有可能性の最大化」を参照) ページング回数を最小にする (を参照)。特にシンボル再配置を最小限にすることにより、再配置の無駄を削減し (135ページの「共有オブジェクトのプロファイリング」を参照)、関数インタフェースを経由して、データにアクセスできるようにする (131ページの「コピー再配置」を参照)

動的実行プログラムの作成

- `-G` オプションと `-dn` オプションを使用しない
- 生成される共有オブジェクトが他の共有オブジェクトに依存している場合は、`-z lazyload` オプションを使用してレイジー読み込みを指定する (82ページの「動的依存関係のレイジーローディング」を参照)

- 生成される動的実行プログラムが他の共有オブジェクトと依存関係があり、これらの共有オブジェクトが 32 ビットオブジェクト用 /usr/lib または 64 ビットオブジェクト用 /usr/lib/64 に存在しない場合は、`-R` オプションを使用してパス名を出力ファイル内に記録する (37 ページの「実行時リンカーが検索するディレクトリ」を参照)
- `-Bdirect` を使って直接結合情報を確立する (60 ページの「外部結合」を参照)。

次の例は、説明したオプションを組み合わせたものです。

```
$ cc -o prog -R /home/lib -z ignore -z lazyload -B direct -L. \  
-lfoo file1.o file2.o file3.o .....
```


バージョンアップの手引き

概要

ELF オブジェクトでは、大域シンボルは他のオブジェクトから結合できます。これらの大域シンボルのいくつかは、オブジェクトの「公開インタフェース」を提供するものであると言えます。それ以外のシンボルは、オブジェクトの内部実装の一部であり、外部使用を目的としていません。オブジェクトのインタフェースは、ソフトウェアのリリースごとに変更されることがあるので、この変更を識別する方法が必要です。

また、ソフトウェアリリースごとのオブジェクトの内部実装の変更を識別する方法も必要とされます。

インタフェースと実装状態の識別情報はいずれも、オブジェクト内にバージョン定義として記録できます (内部バージョンアップの詳しい説明については、139ページの「概要」を参照)。

内部バージョンアップは、共有オブジェクトで最も使用されます。これは、変更を記録して、実行中にインタフェースの妥当性検査 (148ページの「バージョン定義への結合」を参照) を行えるようにし、さらにアプリケーションによる選択的結合 (152ページの「バージョン結合の指定」を参照) を可能にするからです。この章では、共有オブジェクトを例として使用します。

以後の節では、共有オブジェクトに適用されるリンカーが提供する内部バージョンアップ機構の簡単な概要を示します。これらの例では、共有オブジェクトの初期

構築からいくつかの一般的な更新の筋書きを通して、共有オブジェクトをバージョンアップするための規約と機構を示しています。

命名規約

共有オブジェクトは、メジャー (major) ナンバーファイル接尾辞を含むという命名規約に従います。(108ページの「命名規約」を参照)。この共有オブジェクト内では、1つまたは複数のバージョン定義を作成できます。各バージョン定義は、次のいずれかに分類できます。

- 業界標準インタフェースへ準拠したインタフェースを定義する (たとえば、「System V Application Binary Interface」)
- ベンダー特定の「公開」インタフェースを定義する
- ベンダー特定の「専用」インタフェースを定義する
- オブジェクトの内部実装に対する変更(ベンダー特定)を定義する

次の「バージョン定義」命名規約は、定義がどの分類に属するのかを示すために役立ちます。

最初の3つの分類は、インタフェース定義を示します。これらの定義は、インタフェースを構成する大域シンボル名とバージョン定義名の関連付けからなります(142ページの「バージョン定義の作成」を参照)。共有オブジェクト内のインタフェースの変更は、しばしば「マイナーリビジョン (minor revision)」と呼ばれます。このため、このタイプのバージョン定義には、マイナー (minor) バージョンナンバーの接尾辞を付けます。これは、ファイル名のメジャー (major) バージョンナンバーの接尾辞をベースとしたものです。

最後の分類は、オブジェクト内の変更を示します。この定義は、ラベルとして機能するバージョン定義からなり、関連するシンボル名はありません。この定義は、ウィーク (weak) バージョン定義と呼ばれます(145ページの「ウィークバージョン定義の作成」を参照)。共有オブジェクト内の実装の変更は、しばしば「マイクロリビジョン (micro revision)」と呼ばれます。このため、このタイプのバージョン定義には、マイクロ (micro) バージョンナンバーの接尾辞を付けます。これは、元になったもののマイナーナンバーをベースとしたものです。

業界標準インタフェースは、この標準を反映するバージョン定義名を使用しなければなりません。ベンダーインタフェースは、そのベンダー固有のバージョン定義名を使用する必要があります(企業の株式銘柄のシンボルが適していることがあります)。

私用バージョン定義は、使用方法が制限されているかまたは保証されていないシンボルを示します。「private」という語を明確に示すべきです。

バージョン定義を行うと、関連するバージョンシンボル名が必ず作成されます。したがって、一意の名前の使用とマイナー (minor) / マイクロ (micro) の接尾辞の使用という規則を使用すると、構築されるオブジェクト内でシンボルが衝突する可能性を減らすことができます。

次の定義例は、これらの命名規約の使用を示しています。

SVABI.1

「System V Application Binary Interface」標準インタフェースを定義します。

SUNW_1.1

Solaris 公開インタフェースを定義します。

SUNWprivate_1.1

Solaris 私用インタフェースを定義します。

SUNW_1.1.1

Solaris 内部実装の変更を示します。

共有オブジェクトのインタフェースの定義

共有オブジェクトのインタフェースを確立するには、まず、共有オブジェクトによって提供される大域シンボルが、3つのインタフェースバージョン定義分類のうちどれに属するのかを判別します。

- 業界標準インタフェースシンボルの規約は、公開されたヘッダファイルとベンダーによって提供される関連のマニュアルページに定義されている。また、対応する標準の文献にも記述されている
- ベンダーの公開インタフェースシンボルの規約は、公開されたヘッダファイルとベンダーによって提供される関連のマニュアルページに定義されている
- ベンダーの私用インタフェースシンボルの定義は、ほとんど、またはまったく公開されていない

これらのインタフェースを定義することによって、ベンダーは、共有オブジェクトの各インタフェースの保証の程度を示します。業界標準およびベンダーが公開している各インタフェースは、リリースが替わっても安定して使用できます。リリースが替わってもアプリケーションが引き続き正しく機能することを知っていれば、これらのインタフェースを自由に安全に結合することができます。

業界標準インタフェースは、他のベンダーによって提供されたシステムでも使用できる可能性があるため、これらのインタフェースを使用するようにアプリケーションを制限することによって、バイナリ互換性を高めることができます。

ベンダー公開インタフェースは、他のベンダーによって提供されたシステムでは使用できない場合がありますが、これらのインタフェースはそれらが提供されたシステムがバージョンアップしても、安定して使用できます。

ベンダー私有インタフェースは、非常に不安定であり、リリースが替わると変更されたり、削除されたりすることもあります。これらのインタフェースが提供する機能は保証されていないか、または実験的なものです。あるいは、ベンダー特定のアプリケーションに対するアクセスだけを提供することを目的としています。いかなる程度のバイナリ互換性を実現したい場合でも、これらのインタフェースの使用を避けるようにしてください。

上記のどれにも分類されない大域シンボルは、ローカルな適用範囲に限定して、結合では参照できないようにする必要があります (56ページの「シンボル範囲の縮小」を参照)。

共有オブジェクトのバージョンアップ

共有オブジェクトの使用可能インタフェースを決定して、`mapfile` とリンカーの `-M` オプションを使用すれば対応するバージョン定義を作成できます (この `mapfile` 構文の説明については、51ページの「追加シンボルの定義」を参照)。

次の例は、共有オブジェクト `libfoo.so.1` にベンダー公開インタフェースを定義しています。

```
$ cat mapfile
SUNW_1.1 {
    global:
        foo2;
        foo1;
    local:
```

(続く)


```
};
    *;
$ cc -G -o libfoo.so.1 -h libfoo.so.1 -z text -M mapfile foo.c
```

ここで、大域シンボル `foo1` と `foo2` は、共有オブジェクト公開インタフェース `SUNW_1.1` に割り当てられています。入力ファイルから提供された他の大域シンボルはすべて、自動縮小命令「*」によってローカル範囲に縮小されています (56ページの「シンボル範囲の縮小」を参照)。

注 - 各バージョン定義の `mapfile` エントリには、更新のリリースまたは日付を反映するコメントを付けてください。この情報は、たとえば複数の開発者によって行なわれた1つのオブジェクトに対する複数の変更を1つのバージョン定義にまとめて、ソフトウェア製品の一部として共有オブジェクトを配布するのに適切なものに調整するときに役立ちます。

既存の (バージョンアップされていない) 共有オブジェクトのバージョンアップ

既存のバージョンアップされていない共有オブジェクトをバージョンアップするには、特に注意が必要です。これは、以前のソフトウェアリリースで配布された共有オブジェクトが、その大域シンボルのすべてを、他のものと結合できるようにしているためです。共有オブジェクトの意図したインタフェースを判定できる可能性はありますが、ソフトウェア開発者が発見して他のシンボルに結合した可能性もあります。したがって、シンボルを削除すると、新しくバージョンアップされた共有オブジェクトの配布時にアプリケーションに障害が生じる場合があります。

既存のバージョンアップされていない共有オブジェクトの内部バージョンアップは、既存アプリケーションを破壊することなく、インタフェースを判定して適用できる場合に実現できます。実行時リンカーのデバッグ機能は、各種のアプリケーションの結合条件を検査するために役立ちます (103ページの「デバッグングエイド」を参照)。ただし、この既存結合条件の判定は、共有オブジェクトを使用するすべてのプログラムがわかっているということを前提としています。

既存のバージョンアップされていない共有オブジェクトの結合条件を判定できない場合は、新しいバージョンアップ名を使用して、新しい共有オブジェクトファイルを作成する必要があります (158ページの「バージョンアップファイル名の管理」を参照)。すべての既存アプリケーションの依存関係を満たすには、この新しい共有オブジェクトだけでなく、元の共有オブジェクトも配布する必要があります。

元の共有オブジェクトの実装を一切変更しない場合は、共有オブジェクトのバイナリをそのまま配布するだけで十分でしょう。しかし、元の共有オブジェクトを更新する必要がある場合 (たとえば、パッチや、新しいプラットフォームとの互換性を保つための実装の変更など) は、代替ソースツリーから共有オブジェクトを作り直した方がいいでしょう。

バージョンアップ共有オブジェクトの更新

共有オブジェクトに対する互換性のある変更は、内部バージョンアップによって吸収できます (140ページの「インタフェースの互換性」を参照)。すべての互換性のない変更では、新しい外部バージョンアップ名によって、新しい共有オブジェクトを作成する必要があります (158ページの「バージョンアップファイル名の管理」を参照)。

内部バージョンアップによって収容できる互換性のある更新は、次の3つの基本分類に属します。

- 新しいシンボルの追加
- 既存のシンボルに対して新しいインタフェースの作成
- 内部実装の変更

最初の2つは、インタフェースバージョン定義に適切なシンボルを関連付けることによって実現されます。最後のカテゴリは、関連のシンボルを持たないウィークバージョン定義を作成することによって実現されます。

新しいシンボルの追加

新しい大域シンボルを含む、互換性のある、新しいリリースの共有オブジェクトは、これらのシンボルを新しいバージョン定義に割り当てる必要があります。この新しいバージョン定義は、以前のバージョン定義を継承しなければなりません。

次の mapfile の例では、新しいシンボル foo3 を新しいインタフェースバージョン定義 SUNW_1.2 に割り当てています。この新しいインタフェースは、元のインタフェース SUNW_1.1 を継承します。

```
$ cat mapfile
SUNW_1.2 {
    global:
        foo3;
} SUNW_1.1;

SUNW_1.1 {
    global:
        foo2;
        foo1;
    local:
        *;
};
```

バージョン定義の継承によって、共有オブジェクトのユーザーすべてに記録する必要があるバージョン情報の量が減ります。

内部実装の変更

オブジェクトの実装に対する更新からなる、互換性のある新しいリリースの共有オブジェクト (たとえばバグ修正や性能の改善) にはすべて、ウィークバージョン定義を付ける必要があります。この新しいバージョン定義は、更新の発生時に存在する最新のバージョン定義を継承しなければなりません。

次の mapfile の例では、ウィークバージョン定義 SUNW_1.1.1 を生成しています。この新しいインタフェースは、以前のインタフェース SUNW_1.1 によって提供された実装に対して、内部変更が加えられたことを示します。

```
$ cat mapfile
SUNW_1.1.1 { } SUNW_1.1;      # Release X+1.

SUNW_1.1 {                    # Release X.
    global:
        foo2;
        foo1;
    local:
        *;
};
```

新しいシンボルと内部実装の変更

同じリリースで内部変更と新しいシンボルの追加が同時に発生した場合は、ウィークバージョンとインタフェースバージョン定義の両方を作成する必要があります。次の例は、バージョン定義 SUNW_1.2 と、同じリリース期間中に追加されたインタフェース変更 SUNW_1.1.1 を示しています。どちらのインタフェースも元のインタフェース SUNW_1.1 を継承します。

```
$ cat mapfile
SUNW_1.2 {                    # Release X+1.
    global:
        foo3;
} SUNW_1.1;

SUNW_1.1.1 { } SUNW_1.1;     # Release X+1.

SUNW_1.1 {                    # Release X.
    global:
        foo2;
        foo1;
    local:
        *;
};
```

注 - SUNW_1.1 と SUNW_1.1.1 の各バージョン定義へのコメントは、これらが両方とも同じリリースに適用されていることを示しています。

標準インタフェースへのシンボルの併合

場合によっては、ベンダーインタフェースによって提供されたシンボルが、新しい業界標準に組み込まれることがあります。新しい業界標準インタフェースを作成する場合、共有オブジェクトによって提供された元のインタフェース定義を維持するこ

とが重要です。これを実現するには、新しい標準インタフェースおよび元ののインタフェースの定義を構築できる、中間バージョン定義を作成する必要があります。

次の mapfile の例は、新しい業界標準インタフェース STAND.1 の追加を示しています。このインタフェースには、新しいシンボル foo4 と既存のシンボル foo3 および foo1 が含まれます。これらは当初、インタフェース SUNW_1.2 および SUNW_1.1 によって提供されたものです。

```
$ cat mapfile
STAND.1 {                                # Release X+2.
    global:
        foo4;
} STAND.0.1 STAND.0.2;

SUNW_1.2 {                                # Release X+1.
    global:
        SUNW_1.2;
} STAND.0.1 SUNW_1.1;

SUNW_1.1.1 { } SUNW_1.1;                 # Release X+1.

SUNW_1.1 {                                # Release X.
    global:
        foo2;
    local:
        *;
} STAND.0.2;

STAND.0.1 {                                # Subversion - providing for
    global:                                # SUNW_1.2 and STAND.1 interfaces.
        foo3;
};

STAND.0.2 {                                # Subversion - providing for
    global:                                # SUNW_1.1 and STAND.1 interfaces.
        foo1;
};
```

ここで、シンボル foo3 と foo1 は、元のインタフェース定義および新しいインタフェース定義を作成するために使用される自身の中間インタフェース定義に取り込まれます。

注 - SUNW_1.2 インタフェースの新しい定義は、各自のバージョン定義シンボルを参照しています。この参照がないと、SUNW_1.2 インタフェースには即時シンボル参照が含まれないため、ウィークバージョン定義として分類されます。

シンボル定義を標準インタフェースに併合する場合、元のインタフェース定義が引き続き同じシンボル列を表わすことが求められます。これは、pvs(1) を使用して

検査できます。次の例は、SUNW_1.2 インタフェースがソフトウェアリリース X+1 に存在する場合のシンボル列を示しています。

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1
SUNW_1.2:
    foo3;
SUNW_1.1:
    foo2;
    foo1;
```

ソフトウェアリリース X+2 での新しい標準インタフェースの導入によって、使用可能なインタフェースバージョン定義は変更されますが、元の各インタフェースによって提供されたシンボル列はそのままです。次の例は、インタフェース SUNW_1.2 が引き続きシンボル foo1、foo2、および foo3 を提供することを示しています。

```
$ pvs -ds -N SUNW_1.2 libfoo.so.1
SUNW_1.2:
STAND.0.1:
    foo3;
SUNW_1.1:
    foo2;
STAND.0.2:
    foo1;
```

アプリケーションが、新しいサブバージョンの 1 つだけを参照する場合があります。この場合、以前のリリースでこのアプリケーションを実行しようとする、実行時バージョンアップエラーが生じます (148 ページの「バージョン定義への結合」を参照)。

この場合、アプリケーションバージョン結合は、既存のバージョン名を直接参照することによって昇格することができます (154 ページの「追加バージョン定義への結合」を参照)。

たとえば、アプリケーションが、共有オブジェクト libfoo.so.1 からシンボル foo1 だけを参照する場合、そのバージョン参照は STAND.0.2 に対して行われます。このアプリケーションを以前のリリースで実行できるようにするには、バージョン制御 mapfile 指示を使用して、バージョン結合を SUNW_1.1 に昇格します。

```
$ cat prog.c
extern void foo1();

main()
{
    foo1();
}
$ cc -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
    libfoo.so.1 (STAND.0.2);

$ cat mapfile
libfoo.so - SUNW_1.1 $ADDVERS=SUNW_1.1;
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
    libfoo.so.1 (SUNW_1.1);
```

新しい標準バイナリインタフェースの導入はめったになく、またほとんどのアプリケーションが多くのシンボルをインタフェースバージョンに関係なく参照するため、実際には、この方法でバージョン結合を行う必要はめったにありません。

動的ストリングトークンによる依存関係の 確立

概要

動的オブジェクトは、明示的に (70ページの「共有オブジェクトの依存関係の配置」を参照) あるいはフィルタを通して (115ページの「フィルタとしての共有オブジェクト」を参照)、依存関係を確立できます。それぞれの仕組みは「実行パス」で拡張できます。「実行パス」は実行時リンカーに、要求された依存関係を検索させ、読み込ませる指示を送ります。74ページの「動的ストリングトークン」で概説しているように、依存関係および「実行パス」の情報を記録するストリング名は、予約された動的ストリングトークンによって拡張できます。

- \$ISALIST
- \$OSNAME、\$OSREL、および \$PLATFORM
- \$ORIGIN

以降のセクションでは、これらのトークンの使用方法について具体的な例を示します。

命令セット固有の共有オブジェクト

動的トークン `$ISALIST` は、実行時に拡張され、このプラットフォームで実行可能なネイティブの命令セットを反映します。この様子はユーティリティ `isalist(1)` によって表されます。

`$ISALIST` トークンを持つあらゆるストリング名は、それぞれ使用できる命令セットの1つを割り当てられた複数の文字列に効率良くコピーされます。このトークンは、「フィルタ」あるいは「実行パス」指定でのみ使用できます。

次の例では、命令セット固有のフィルティアー `libfoo.so.1` にアクセスするように補助フィルタ `libbar.so.1` を設計する方法を示します。

```
$ LD_OPTIONS='-f /opt/ISV/lib/$ISALIST/libbar.so.1' \  
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c  
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"  
[1] SONAME libfoo.so.1  
[2] AUXILIARY /opt/ISV/lib/$ISALIST/libbar.so.1
```

あるいは、代わりに「実行パス」を使用することができます。

```
$ LD_OPTIONS='-f libbar.so.1' \  
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R'/opt/ISV/lib/$ISALIST' foo.c  
$ dump -Lv libfoo.so.1 | egrep "RPATH|AUXILIARY"  
[1] RPATH /opt/ISV/lib/$ISALIST  
[2] AUXILIARY libbar.so.1
```

どちらの場合でも、実行時リンカーはプラットフォームで使用可能な命令リストを使用して、複数の検索パスを構成します。たとえば、次のアプリケーションは `libfoo.so.1` に依存関係があり、SUNW,Ultra-2 上で実行されます。

```
$ ldd -ls prog  
.....  
find object=libbar.so.1; required by ./libfoo.so.1  
  search path=/opt/ISV/lib/$ISALIST (RPATH from file ./libfoo.so.1)  
  trying path=/opt/ISV/lib/sparcv9+vis/libbar.so.1  
  trying path=/opt/ISV/lib/sparcv9/libbar.so.1  
  trying path=/opt/ISV/lib/sparcv8plus+vis/libbar.so.1  
  trying path=/opt/ISV/lib/sparcv8plus/libbar.so.1  
  trying path=/opt/ISV/lib/sparcv8/libbar.so.1  
  trying path=/opt/ISV/lib/sparcv8-fsmuld/libbar.so.1  
  trying path=/opt/ISV/lib/sparcv7/libbar.so.1
```

(続く)

```
trying path=/opt/ISV/lib/sparc/libbar.so.1
```

また、同じ依存関係を持つアプリケーションは、MMX 設計の Pentium Pro で実行されます。

```
$ ldd -ls prog
.....
find object=libbar.so.1; required by ./libfoo.so.1
  search path=/opt/ISV/lib/$ISALIST (RPATH from file ./libfoo.so.1)
    trying path=/opt/ISV/lib/pentium_pro+mmx/libbar.so.1
    trying path=/opt/ISV/lib/pentium_pro/libbar.so.1
    trying path=/opt/ISV/lib/pentium+mmx/libbar.so.1
    trying path=/opt/ISV/lib/pentium/libbar.so.1
    trying path=/opt/ISV/lib/i486/libbar.so.1
    trying path=/opt/ISV/lib/i386/libbar.so.1
    trying path=/opt/ISV/lib/i86/libbar.so.1
```

補助検索の縮小

補助フィルタ内で \$ISALIST を使用すると、1 つあるいは複数のフィルティーが、フィルタ内で定義されたインタフェースの代替の実装を提供できます。

ただし、フィルタで定義されたインタフェースにフィルティーで定義された代替の実装がない場合は、要求されたインタフェースの場所を探すために、可能性のあるフィルティーすべてを徹底的に検索する結果となります。性能が重要となる機能を提供するためにフィルティーを使用する場合には、徹底的にフィルティーを検索することによって生産性が下がることがあります。

リンカーの `-z endfiltee` オプションを使用してフィルティーを作成し、それが使用可能な最後のフィルティーであると指定し、そのフィルタについてそれ以上のフィルティー検索を行わないようにできます。たとえば、前の SPARC の例で、`sparcv9` フィルティーが存在し、`-z endfiltee` のタグが付いている場合、フィルティー検索は次のようになります。

```
$ ldd -ls prog
.....
find object=libbar.so.1; required by ./libfoo.so.1
  search path=/opt/ISV/lib/$ISALIST (RPATH from file ./libfoo.so.1)
    trying path=/opt/ISV/lib/sparcv9+vis/libbar.so.1
    trying path=/opt/ISV/lib/sparcv9/libbar.so.1
```

プラットフォーム固有の共有オブジェクト

動的トークン `$OSNAME`、`$OSREL`、および `$PLATFORM` は実行時に展開されて、システム固有の情報を提供します。`$OSNAME` は、ユーティリティ `uname(1)` にオプション `-s` を付けて実行した場合に表示されるように、オペレーティングシステムの名前を反映するよう展開されます。`$OSREL` は、`uname -r` を実行した場合に表示されるように、オペレーティングシステムのリリースレベルを反映するよう展開され、`$PLATFORM` は、`uname -i` を実行した場合に表示されるように、基礎としているハードウェア実装を反映するよう展開されます。

次の例は、プラットフォーム固有のフィルティー `libfoo.so.1` にアクセスするように補助フィルタ `libbar.so.1` を設計する方法を示します。

```
$ LD_OPTIONS='-f /usr/platform/$PLATFORM/lib/libbar.so.1' \  
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c  
$ dump -Lv libfoo.so.1 | egrep "SONAME|AUXILIARY"  
[1] SONAME libfoo.so.1  
[2] AUXILIARY /usr/platform/$PLATFORM/lib/libbar.so.1
```

この仕組みは、共有オブジェクト `/usr/lib/libc.so.1` に対してプラットフォーム固有の拡張を提供するために、Solaris 上で使用されます。

注 - 環境変数 `LD_NOAUXFLTR` を設定して、実行時リンカーの「補助」フィルタ処理を無効にすることができます。補助フィルタはプラットフォーム固有の最適化に使用されることが多いので、フィルティーを使用する場合および性能インパクトを評価する場合にこのオプションが便利です。

関連する依存関係の配置

通常、バンドルされていない製品は、独立した固有の場所にインストールされるように設計されています。この製品は、バイナリ、共有オブジェクト、および関連構成ファイルからなります。たとえば、バンドルされていない製品 `ABC` は、次の配置をとる場合があります。

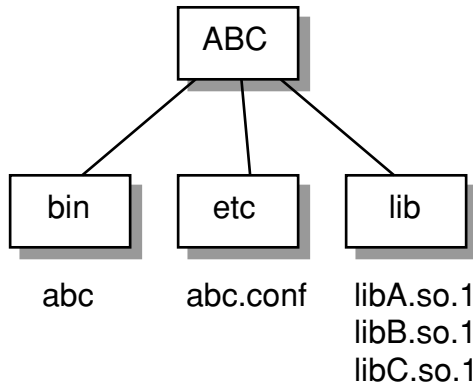


図 C-1 バンドルされていない依存関係

製品が、`/opt` の元にインストールされるように設計されていると想定します。通常、ユーザーは、`PATH` に製品バイナリの位置を示す `/opt/ABC/bin` を追加する必要があります。各バイナリは、バイナリ内に直接記録された実行パスを使用して、その依存する相手を探します。アプリケーション `abc` の場合、これは次のようになります。

```

% dump -Lv abc

[1]  NEEDED  libA.so.1
[2]  RPATH   /opt/ABC/lib
  
```

また、`libA.so.1` の依存関係でも同様にこれは次のようになります。

```

% dump -Lv libA.so.1

[1]  NEEDED  libB.so.1
[2]  RPATH   /opt/ABC/lib
  
```

この依存関係の表現は、製品が推奨されているデフォルト以外のディレクトリにインストールされるまで正常に作動します。異なるインストール環境が作成された場合、ユーザーは `LD_LIBRARY_PATH` を使用して、製品アプリケーションを実行しなければなりません。通常、これは、バイナリごとにラッパーを作って対応しますが、場合によっては、適切なオブジェクト内の実行パスを変更しようとするユーザーもいます。

動的トークン `$ORIGIN` は、オブジェクトが存在するディレクトリを表わします。この機能は、カーネルによってプロセス開始時に実行時リンカーに提供された新しい補助ベクトルに対応しています (`getexecname(3C)` のマニュアルページも参

照)。この機構を使用すると、バンドルされていないアプリケーションを再定義して、\$ORIGIN との相対位置で依存対象の位置を示すことができます。

```
% dump -Lv abc
[1]  NEEDED  libA.so.1
[2]  RPATH   $ORIGIN/../lib
```

また、\$ORIGIN との関係で libA.so.1 の依存関係を定義することもできます。

```
% dump -Lv libA.so.1
[1]  NEEDED  libB.so.1
[2]  RPATH   $ORIGIN
```

したがって、この製品が /usr/local/ABC の元にインストールされて、ユーザーの PATH に /usr/local/ABC/bin が追加された場合、アプリケーション abc を呼び出すと次のように、パス名検索でその依存関係が探されます。

```
% ldd -s abc
find object=libA.so.1; required by abc
  search path=$ORIGIN/../lib (RPATH from file abc)
  trying path=/usr/local/ABC/lib/libA.so.1
    libA.so.1 => /usr/local/ABC/lib/libA.so.1

find object=libB.so.1; required by /usr/local/ABC/lib/libA.so.1
  search path=$ORIGIN (RPATH from file /usr/local/ABC/lib/libA.so.1)
  trying path=/usr/local/ABC/lib/libB.so.1
    libB.so.1 => /usr/local/ABC/lib/libB.so.1
```

バンドルされていない製品間の依存関係

依存関係の場所に関する次の問題としては、バンドルされていない製品が別のバンドルされていない製品の共有オブジェクトに対して依存関係を持つときの、基本となるモデルを作成する方法が挙げられます。

たとえば、バンドルされていない製品 XYZ は製品 ABC に対して依存関係を持つ場合があります。この依存関係は、図 C-2 に示すようにホストパッケージインストールスクリプトによって ABC 製品のインストール位置へのシンボルリンクを生成することで確立できます。

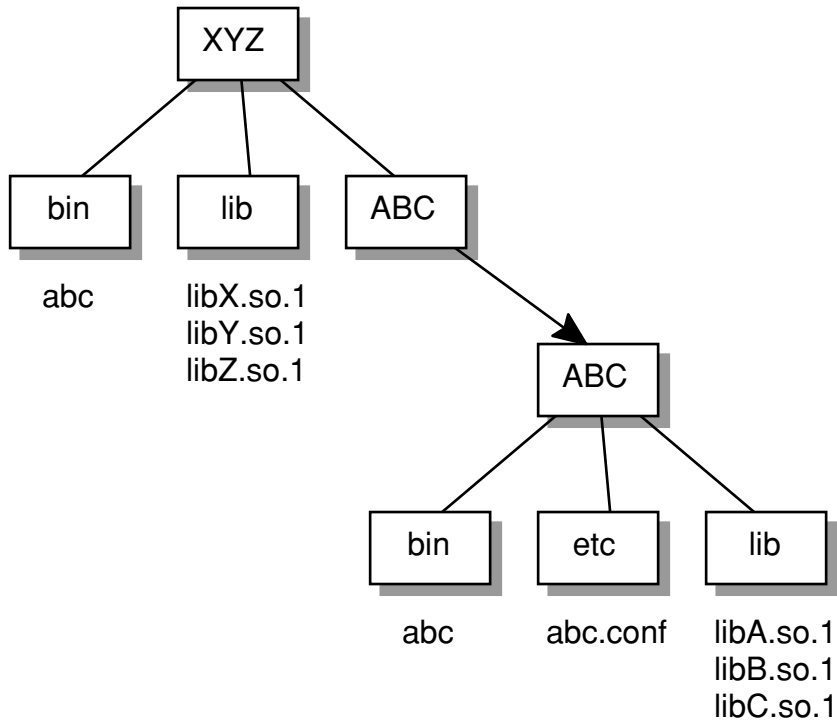


図 C-2 バンドルされていない製品の相互依存関係

XYZ 製品のバイナリと共有オブジェクトは安定した参照位置としてシンボルリンクを使用して、ABC 製品への依存関係を表わします。アプリケーション xyz の場合、次のようになります。

```

% dump -Lv xyz
[1]  NEEDED  libX.so.1
[2]  NEEDED  libA.so.1
[3]  RPATH   $ORIGIN/../../lib:$ORIGIN/../../ABC/lib
  
```

libX.so.1 の依存関係でも同様に、次のようになります。

```

% dump -Lv libX.so.1
[1]  NEEDED  libY.so.1
[2]  NEEDED  libC.so.1
[3]  RPATH   $ORIGIN:$ORIGIN/../../ABC/lib
  
```

したがって、この製品が /usr/local/XYZ の元にインストールされている場合は、次のシンボルリンクを確立するために、そのインストール後実行スクリプトが必要です。

```
% ln -s ../ABC /usr/local/XYZ/ABC
```

ユーザーの PATH に /usr/local/XYZ/bin が追加される場合、アプリケーション xyz の呼び出しによって、次のようにパス名検索でその依存関係が探されます。

```
% ldd -s xyz
find object=libX.so.1; required by xyz
search path=$ORIGIN/../lib:$ORIGIN/../ABC/lib (RPATH from file xyz)
trying path=/usr/local/XYZ/lib/libX.so.1
libX.so.1 => /usr/local/XYZ/lib/libX.so.1

find object=libA.so.1; required by xyz
search path=$ORIGIN/../lib:$ORIGIN/../ABC/lib (RPATH from file xyz)
trying path=/usr/local/XYZ/lib/libA.so.1
trying path=/usr/local/ABC/lib/libA.so.1
libA.so.1 => /usr/local/ABC/lib/libA.so.1

find object=libY.so.1; required by /usr/local/XYZ/lib/libX.so.1
search path=$ORIGIN:$ORIGIN/../ABC/lib \
(RPATH from file /usr/local/XYZ/lib/libX.so.1)
trying path=/usr/local/XYZ/lib/libY.so.1
libY.so.1 => /usr/local/XYZ/lib/libY.so.1

find object=libC.so.1; required by /usr/local/XYZ/lib/libX.so.1
search path=$ORIGIN:$ORIGIN/../ABC/lib \
(RPATH from file /usr/local/XYZ/lib/libX.so.1)
trying path=/usr/local/XYZ/lib/libC.so.1
trying path=/usr/local/ABC/lib/libC.so.1
libC.so.1 => /usr/local/ABC/lib/libC.so.1

find object=libB.so.1; required by /usr/local/ABC/lib/libA.so.1
search path=$ORIGIN (RPATH from file /usr/local/ABC/lib/libA.so.1)
trying path=/usr/local/ABC/lib/libB.so.1
libB.so.1 => /usr/local/ABC/lib/libB.so.1
```

セキュリティ

セキュアプロセス (87ページの「セキュリティ」を参照) では、\$ORIGIN 文字列の拡張は、それがトラストディレクトリに拡張されるときに限り許可されます。他の相対パス名は、セキュリティリスクを伴います。

一見すると、\$ORIGIN/../lib のようなパスは一定の場所 (実行可能プログラムの場所で特定される) を指しているように見えますが、それは正しくありません。\$ORIGIN を使ってセキュアプログラムを活用するために必要なものは、同じファイルシステム内の書き込み可能なディレクトリです。

次の例は、\$ORIGIN がセキュアプロセス内で任意に拡張された場合、上記の事例がどのようにして起こるかを示しています。


```
% cd /worldwritable/dir/in/same/fs
% mkdir bin lib
% ln $ORIGIN/bin/program bin/program
% cp ~/crooked-libc.so.1 lib/libc.so.1
% bin/program
..... using crooked-libc.so.1
```

ユーティリティ `crle(1)` を使用すれば、`$ORIGIN` を使用するセキュアアプリケーションに適用されるトラストディレクトリを指定できます。この方法を使用する場合には、管理者は、ターゲットディレクトリを悪意のある侵入から適切に保護する必要があります。

『リンカーとライブラリ』の更新箇所

新しい機能および更新された機能

この付録では、このマニュアルに追加された新機能や更新された事項、およびそれらがどのリリースで追加されたかの概要を示します。

Solaris 8 7/01 リリース

- 使用されない依存関係を、`ldd(1)` を使用して特定できるようになりました (`-u` オプションを参照)。
- さまざまな ELF ABI の拡張が追加され、関連する記述が更新されました。38ページの「初期設定および終了セクション」、84ページの「初期設定および終了ルーチン」、表 7-4、表 7-7、表 7-14、表 7-15、226ページの「セクショングループ」、表 7-17、表 7-22、表 7-43、表 7-44、287ページの「プログラムの読み込み (プロセッサ固有)」を参照。
- `_32` および `_64` の形式が追加されたことにより、リンカー特有の環境変数の使用における柔軟性が大幅に向上しました。22ページの「環境変数」を参照。

Solaris 8 1/01 リリース

- `dladdr(3DL)` から入手できるシンボリック情報が、`dladdr1()` の導入により拡張されました。
- 動的オブジェクトの `$ORIGIN` (364ページの「関連する依存関係の配置」を参照) が `dlinfo(3DL)` から入手できます。

- `crle(1)` で作成された実行時構成ファイルの管理が、構成ファイルの作成に使用されたコマンドラインオプションの表示、および機能の更新によって簡単になりました (`-u` オプションを参照)。
- 実行時リンカーおよびデバッガインタフェースが拡張され、プロシージャリンクテーブルエントリの解決を検出できるようになりました。この拡張は、新しいバージョンナンバーで識別することができます。185ページの「エージェント操作」の `rd_init()` を参照。この更新により `rd_plt_info_t` 構造体が機能拡張されます。192ページの「プロシージャのリンクテーブルのスキップ」の `rd_plt_resolution()` を参照。
- 新しい `mapfile` セグメント記述子 `STACK` を使用してアプリケーションスタックを非実行可能プログラムに定義することができます。327ページの「セグメントの宣言」を参照。

Solaris 8 10/00 リリース

- 実行時リンカーが、環境変数 `LD_BREADTH` を無視します。84ページの「初期設定および終了ルーチン」を参照。
- 実行時リンカーとそのデバッガインタフェースが拡張され、実行時分析とコアファイル分析が改善されました。この更新は、新しいバージョンナンバーで識別することができます。185ページの「エージェント操作」の `rd_init()` を参照。この更新により `rd_loadobj_t` 構造体が拡張されます。187ページの「読み込み可能オブジェクトの走査」を参照。
- ディスプレイメント再配置されたデータがコピー再配置で使用されるか、使用される可能性があることを検査する機能が提供されます。63ページの「ディスプレイメント再配置」を参照。
- 64 ビットフィルタが、リンカーの `-64` オプションを使用してマップファイルから単独で構築できます。116ページの「標準フィルタの生成」を参照。
- `$ORIGIN` 動的文字列トークンの拡張がなぜセキュアアプリケーション内に限定されるのかの説明が追加されました。368ページの「セキュリティ」を参照。
- 動的オブジェクトの依存関係の検索に使用される検索パスを、`dlinfo(3DL)` を使って調べることができます。
- `dlsym(3DL)` と `dlinfo(3DL)` 検索の方法が新しいハンドル `RTLD_SELF` によって拡張されました。

- 動的オブジェクトの再配置に使用される実行時シンボル検索メカニズムを、各動的オブジェクト内に直接結合情報を確立することによって、大幅に削減することができます。60ページの「外部結合」および78ページの「直接結合」を参照。

Solaris 8 リリース

- ファイルを前もってロードすることのできるセキュリティ保護されたディレクトリが、32ビットオブジェクトの場合は `/usr/lib/secure`、64ビットオブジェクトの場合は `/usr/lib/secure/64` となりました。87ページの「セキュリティ」を参照。
- リンカーの `-z nodefaultlib` オプションおよび新ユーティリティ `crle(1)` によって作成される実行時構成ファイルを使用することにより、実行時リンカーの検索パスを変更する柔軟性が向上しました。37ページの「実行時リンカーが検索するディレクトリ」および73ページの「デフォルトの検索パスの設定」を参照してください。
- 新しい `extern mapfile` 指示文により、`-z defs` の使用に外部的に定義されたシンボルを提供します。51ページの「追加シンボルの定義」を参照。
- 新しい `$ISALIST`、`$OSNAME`、および `$OSREL` 動的ストリングトークンにより、命令セット固有およびシステム固有の依存関係を確立する際の柔軟性が向上しました。74ページの「動的ストリングトークン」を参照。
- リンカーの `-Bsymbolic` オプションの使用に関する注意が、134ページの「`-Bsymbolic` の使用」に追加されました。
- リンカーの `-p` および `-P` オプションにより、実行時リンク監査ライブラリを呼び出す方法が追加されました。173ページの「ローカル監査の記録」を参照。実行時リンク監査インタフェース `la_activity()` および `la_objsearch()` が追加されました。174ページの「監査インタフェースの関数」を参照。
- 新しい動的セクションタグ `DT_CHECKSUM` により、ELF ファイルとコアイメージとの統合が可能になりました。表 7-43 を参照。

Solaris 7 リリース

- 64ビットELFオブジェクト形式がサポートされるようになりました。詳細は、200ページの「ファイル形式」を参照してください。64ビット処理のためのリンカーの拡張機能および相違点には、以下のものがあります。`/usr/lib/64` の使用 (35ページの「リンカーが検索するディレクトリ」、37ページの「実行時リンカーが検索するディレクトリ」、および108ページの「命名規約」を参照)、環

境変数 `LD_LIBRARY_PATH_64` (36ページの「環境変数の使用」および70ページの「実行時リンカーによって検索されるディレクトリ」を参照)、および実行時リンカー `/usr/lib/64/ld.so.1` (69ページの「概要」を参照)。

- リンカーの `-z combrelloc` オプションを使用することによって、最適化された再配置セクションを使用して共有オブジェクトを構築できます。130ページの「再配置セクションの結合」を参照。
- 新しい `$ORIGIN` 動的ストリングトークンにより、バンドルされていないソフトウェア内に依存関係を確立する際の柔軟性が向上しました。74ページの「動的ストリングトークン」を参照。
- 共有オブジェクトの読み込みは、実行プログラムが実際にそのオブジェクトを参照するまで延期することができます。124ページの「動的依存関係のレイジーローディング」を参照。
- 重複して定義されたシンボルの除去に対処するために、`SHT_SUNW_COMDAT` セクションが追加されました。267ページの「Comdat セクション」を参照。
- シンボルの部分的な初期化に対処するため、`SHT_SUNW_move` セクションが追加されました。276ページの「移動セクション」を参照。
- 実行時リンク監査のインタフェース
`la_symbind64()`、`la_sparcv9_pltenter()`、および `la_pltexit64()` が、新しいリンク監査フラグ `LA_SYMB_ALTVALUE` とともに付加されました。174ページの「監査インタフェースの関数」を参照。

Solaris 2.6 リリース

- ウィークシンボルリファレンスは、リンカーの `-z weakextract` オプションを使用することにより、アーカイブ構成要素の抽出をトリガーできます。すべてのアーカイブ構成要素の抽出は、`-z allextract` オプションを使用して実行できます。30ページの「アーカイブ処理」を参照。
- 作成されるオブジェクトが参照しないリンク編集の一部として指定された共有オブジェクトは、リンカーの `-z ignore` オプションを使用して無視することができます。したがって、その共有オブジェクトの依存関係の記録も減らすことができます。31ページの「共有オブジェクトの処理」を参照。
- リンカーが、予約シンボル `_START_` および `_END_` を生成し、オブジェクトのアドレス範囲を確立する方法を提供します。61ページの「出力イメージの生成」を参照。

- 初期設定および終了コードの実行時の順序に変更が加えられ、依存関係の要件をより良く満たすようになりました。84ページの「初期設定および終了ルーチン」を参照。
- シンボル解析の方法が、`dlopen(3DL)` 用に拡張されました。92ページの「シンボル検索」、97ページの「グループの分離」の `RTLD_GROUP`、および 97ページの「オブジェクト階層」の `RTLD_PARENT` を参照。
- シンボル検索の方法が、`RTLD_DEFAULT` を処理する新しい `dlsym(3DL)` により拡張されました。93ページの「デフォルトのシンボル検索モデル」を参照。
- フィルタ処理が拡張され、複数のフィルタが定義できるようになり、さらに強制的に読み込まれるフィルタが使用できるようになりました。プラットフォーム固有のフィルタを作成する場合の例も提供されました。115ページの「フィルタとしての共有オブジェクト」を参照。
- `mapfile` ファイル制御指示語 `$ADDVERS` を使用すると、追加されたバージョン依存関係の記録が実行できます。154ページの「追加バージョン定義への結合」を参照。
- 実行時リンカーの監査インタフェースで、プロセスの内部から動的にリンクされたアプリケーションの監視および変更のサポートを提供します。169ページの「実行時リンカーの監査インタフェース」を参照。
- 実行時リンカーのデバグインタフェースにより、外部プロセスから動的にリンクされたアプリケーションの監視および修正のサポートが提供されます。169ページの「実行時リンカーの監査インタフェース」を参照。
- 追加のセクションタイプがサポートされました。`SHN_BEFORE` と `SHN_AFTER` については表 7-11、`SHF_ORDERED` と `SHF_EXCLUDE` については表 7-14 を参照してください。
- 新しい動的セクションタグ `DT_1_FLAGS` がサポートされました。種々のフラグ値については、表 7-45 を参照。
- ELF のデモプログラムパッケージが提供されました。第 7 章を参照。
- リンカーが国際化メッセージをサポートするようになりました。システムエラーはすべて、`strerror(3C)` を使用して報告されます。

索引

数字

32/64 ビット 22, 35, 37, 69, 71 - 73, 88, 108,
113, 165, 172, 182, 201, 203,
210, 212, 247, 249, 345

A

ABI 21
\$ADDVERS 152
ar(1) 30
as(1) 18
atexit(3C) 87, 324

C

cc(1) 17, 18, 27
COMDAT 167, 268
COMMON 42, 53, 56, 214

D

dladdr(3DL) 371
dladdr1(3DL) 371
dlclose(3DL) 87, 89, 324
dldump(3DL) 40
dlerror(3DL) 89
dlfcn.h 89
dldlinfo(3DL) 372
dlmopen(3DL) 171

dlopen(3DL) 70, 89, 90, 97, 120, 151, 164
共有オブジェクト命名規約 109
グループ 91, 92
順番の影響 95
動的実行プログラム 91, 97
モード 91, 92, 97, 98, 171
dlsym(3DL) 70, 89, 98, 102, 152, 164
特別なハンドル 50, 99, 373
dump(1) 23, 71, 75, 122, 125

E

ELF 18, 25, 121, 199
elf(3E) 23
environment variables 23
exec(2) 25, 69, 200

F

f77(1) 27

I

\$ISALIST 362

L

ld(1) 17
ld.so.1(1) 17, 25, 69

ldd(1) 23, 71, 73, 77, 80, 120, 150, 151

ldd(1) オプション 85

-d 65, 80, 133

-i 85

-r 65, 133

-u 32

-v 150

LD_AUDIT 88, 172

LD_BIND_NOT 105

LD_BIND_NOW 79, 105, 296, 321, 322

LD_BREADTH 86

LD_DEBUG 103

LD_DEBUG_OUTPUT 104

LD_LIBRARY_PATH 72, 88, 90, 114, 172

LD_LOADFLTR 120

LD_NOAUDIT 174

LD_NOAUXFLTR 364

LD_NODIRECT 78

LD_NOLAZYLOAD 84

LD_OPTIONS 28, 66

LD_PRELOAD 81, 88

LD_PROFILE 135

LD_PROFILE_OUTPUT 136

LD_RUN_PATH 38

libdl.so.1 89

libelf.so.1 165, 200

libldstab.so.1 164

lorder(1) 31, 67

M

mapfile 325, 341

エラーメッセージ 340

サイズシンボル宣言 334

構造 326

構文 326

初期値 336

セグメントの宣言 327

対応付け構造 338

対応付け指示 331

使い方 326

例 334

mmap(2) 69

N

NEEDED 71, 109

nm(1) 23, 122

O

\$OSNAME 364

\$OSREL 364

P

\$PLATFORM 364

profil(2) 136

pvs(1) 23, 143, 144, 147, 149

R

RPATH 72

RTLD_DEFAULT 50, 99

RTLD_GLOBAL 91, 97

RTLD_GROUP 97

RTLD_LAZY 92

RTLD_NEXT 99

RTLD_NOLOAD 171

RTLD_NOW 92

RTLD_PARENT 98

S

SCD 21

SGS_SUPPORT 164

size(1) 121

SONAME 110

SPARC Compliance Definition 21

strings(1) 128

strip(1) 61

SUNWosdem 180, 184, 200

SUNWtoo 181

SYMBOLIC 135

System V アプリケーションバイナリインタ
フェース 350

T

TEXTREL 125

tsort(1) 31, 67

U

/usr/ccs/bin/ld 165

/usr/ccs/lib 35

/usr/lib 37, 71, 72, 90, 172

/usr/lib/64 35, 37, 71, 72

/usr/lib/64/ld.so.1 69, 182
/usr/lib/ld.so.1 69, 182
/usr/lib/secure 88, 172
/usr/lib/secure/64 88, 172

あ

アーカイブ 33
共有オブジェクトの取り込み 111
命名規約 33
リンカー処理 30
を通る複数のパス 31
アプリケーションバイナリインタフェース 21, 117, 139

い

依存関係
グループ 91, 92
依存関係の順序 114
一時的シンボル 30, 42, 53, 56
位置に依存しないコード 124, 292, 304, 316
インタフェース
公的 350
私的 140
インタプリタ 69

う

ウィークシンボル 43, 239, 240
未定義 49

え

エラーメッセージ
実行時リンカー 65, 73, 80, 91, 100, 133, 150
リンカー 28, 29, 45 - 47, 59, 112, 125, 154

お

オブジェクトの事前読み込み 81
オブジェクトファイル 18
再配置 248, 258, 316
実行時の事前読み込み 81
シンボルテーブル 237, 246
セクショングループのフラグ 227

セクションタイプ 217, 235
セクションの整列 216
セクションの属性 222, 235
セクションヘッダー 212, 235
セクション名 235
セグメントタイプ 280, 284
セグメントの内容 286, 287
セグメントへのアクセス権 285, 286
大域オフセットテーブル 316
注釈セクション 274, 276
データ表現 201
プログラムインタプリタ 294
プログラムの読み込み 287
プログラムヘッダー 279, 283, 284
プロシージャのリンクテーブル 317, 321
ベースアドレス 284
文字列テーブル 235, 236

か

仮想アドレス 287
環境変数
LD_AUDIT 88, 172
LD_BIND_NOT 105
LD_BIND_NOW 79, 105, 295, 321, 322
LD_BREADTH 86
LD_DEBUG 103
LD_DEBUG_OUTPUT 104
LD_LIBRARY_PATH 36, 72, 88, 90, 114, 172
LD_LOADFLTR 120
LD_NOAUDIT 174
LD_NOAUXFLTR 364
LD_NODIRECT 78
LD_NOLAZYLOAD 84
LD_OPTIONS 28, 66
LD_PRELOAD 81, 88
LD_PROFILE 135
LD_PROFILE_OUTPUT 136
LD_RUN_PATH 38
SGS_SUPPORT 164

き

共有オブジェクト 18 - 20, 70
暗黙的定義 48
依存関係 113

依存関係の順序 114
実行時名の記録 109
実装 248, 258, 290
フィルタとしての 115
明示的定義 48
命名規約 33, 108
リンカー処理 31
共有オブジェクトの生成 49
共有ライブラリ 18

け

結合 17
依存関係の順序 114
ウィークバージョン定義への 155
共有オブジェクトの依存関係への 109,
148
直接 61, 76, 130
バージョン定義への 148
レイジー 79, 91, 105
検索パス
実行時リンカー 37, 70, 362, 364
リンク編集 35

こ

コンパイル環境 17, 20, 33, 108

さ

再配置 75, 129, 134, 248, 258
関数参照 79
コピー 63, 131
実行時リンカー 77, 79, 91, 105
シンボリック 75, 129
ディスプレイイメント 63
データ参照 79
非シンボリック 75, 129
再配置可能オブジェクト 18
サポートインタフェース
実行時リンカー (rtld-デバugga) 163, 181
実行時リンカー (rtld-監査) 163, 169
リンカー (ld-サポート) 163

し

システム固有の補助フィルタ 364
実行可能なリンク書式 18
実行可能ファイルの作成 47

実行時環境 21, 33, 108
実行時リンカー 17, 19, 69, 294
共有オブジェクトの処理 70
検索パス 37, 70
再配置処理 75
初期設定と終了ルーチン 84
セキュリティ 87
直接結合 76, 78, 92, 130
追加オブジェクトの読み込み 81
名前空間 170
バージョン定義の検査 150
リンクマップ 170
レイジー結合 79, 91, 105
実行時リンカーのサポートインタフェース
(rtld-デバugga) 163, 181
ps_global_sym 196
ps_pglobal_sym 197
ps_plog 196
ps_pread 196
ps_pwrite 196
rd_delete 186
rd_errstr 186
rd_event_addr 191
rd_event_enable 190
rd_event_getmsg 192
rd_init 185
rd_loadobj_iter 189
rd_log 187
rd_new 185
rd_objpad_enable 195
rd_plt_resolution 193
rd_reset 186
実行時リンカーのサポートインタフェース
(rtld-監査) 163, 169
la_activity 174
la_i86_pltenter 178
la_objclose 179
la_objopen 176
la_objseach 175
la_pltexit 179
la_preinit 176
la_sparcv8_pltenter 178
la_symbind32 176
la_symbind64 176
la_version 174
実行時リンク 19
実行パス 37, 72, 88, 90, 105, 113

出力ファイルイメージの生成 61
初期設定と終了 27, 38, 84
シンボル
COMMON 42, 53, 56, 214
auto-reduction (自動縮小) 53
アーカイブの抽出 30
一時的 30, 42, 53, 56, 214
ウィーク 30, 43, 49, 240
公的インタフェース 140
実行時検索 79, 91, 92, 102, 105
自動縮小 142, 353
絶対 53, 214
専用インタフェース 140
存在テスト 49
大域 41, 43, 140, 239, 240
定義 42, 47
定義された 30
範囲 92, 97
未定義 30, 42, 46, 47, 49
リファレンス 30, 47
ローカル 41, 239, 240
シンボル解析 41, 61
検索範囲 92
重大な 46
シンボルの可視性 93
単純な 42
複雑な 44
複数の定義 32
割り込み 77
シンボルの予約名 62
_DYNAMIC 62
END 62
_GLOBAL_OFFSET_TABLE_ 62, 317
_PROCEDURE_LINKAGE_TABLE_ 63
START 63
_edata 62
_end 62
_etext 62
_fini 39
_init 39
_start 63
main 63

せ

静的実行可能ファイル 18
性能
位置に依存しないコード 124

基本システム 123
共有可能性の最大化 126
再配置 129, 135
参照のローカリティの改善 129, 135
自動変数の使用 128
多重定義の短縮 127
データセグメントの最小化 126
バッファの動的割り当て 128
セキュリティ 87
セクション 25, 121
セクションタイプ 39, 71
SHT_SUNW_COMDAT 167, 268
.SUNW_reloc 131
.SUNW_version 268
.bss 25, 131
.data 25, 126
.dynamic 62, 70, 135
.dynstr 61
.dysym 61
.fini 39, 86
.fini_array 39, 86
.got 62, 76
.init 39, 84
.init_array 39, 84
.interp 69
.picdata 127
.plt 63, 79, 135
.preinit_array 39, 84
.rela.text 26
.rodata 126
.strtab 26, 61
.symtab 26, 61
.text 25
セグメント 25, 121
データ 122, 124
テキスト 122, 124

た

大域オフセットテーブル 62, 76, 125, 258,
316, 317
大域シンボル 41, 140, 239, 240

ち

直接結合 61, 130

て

データ表現 201
デバ깅エイド
 実行時リンク編集 103
 リンク編集 65
デモンストレーション
 prefcnt 180
 sotruss 180
 sybindrep 181
 whocalls 180

と

動的実行可能ファイル 18, 20
動的情報タグ
 NEEDED 71, 109
 RPATH 72
 SONAME 110
 SYMBOLIC 135
 TEXTREL 125
動的リンク 21
 実装 248, 258, 290

な

名前空間 170

に

入力ファイルの処理 29

は

バージョンアップ 139
 イメージ内の定義の生成 52, 59, 141
 概要 139
 公的インタフェースの定義 59, 142
 実行時の検査 150, 151
 正規化 149
 定義 140, 141, 148
 定義への結合 148, 152
 ファイル制御命令 152
 ファイル名 141, 354
 ベースのバージョン定義 143
パッケージ
 SUNWosdem 180, 184, 200
 SUNWtoo 181

ひ

標準フィルタ 115, 116

ふ

フィルタ 115
 標準 115, 116
 補助 115, 118, 364
複数定義のシンボル 61, 239, 240
プラットフォーム固有の補助フィルタ 364
プログラムインタプリタ 69, 294
プロシージャのリンクテーブル 63, 79, 125,
 258, 317, 321
 SPARC 318, 321

へ

ページング 287, 290
ベースアドレス 284

ほ

補助フィルタ 115, 118

み

未定義シンボル 46

め

命名規約
 アーカイブ 33
 共有オブジェクト 33, 108
 ライブラリ 33

ら

ライブラリ
 アーカイブ 33
 共有 248, 258, 290
 命名規約 33

り

リンカー 17, 25
 エラーメッセージ 17
 オプションの指定 28
 概要 25

- コンパイラドライバを使用して起動する 27
- セクション 25
- セグメント 25
- 直接起動する 26
- 直接結合 61
- デバッグgingエイド 65
- リンカーオプション
 - 64 118
 - B direct 61, 346, 347
 - B dynamic 34
 - B group 312
 - B reduce 54, 60
 - B static 34, 345
 - B group 92, 97
 - D 65
 - F 115
 - G 107
 - L 35, 343
 - M 26, 51, 52, 140, 142, 152, 325, 345, 352
 - P 173, 309
 - R 38, 113, 345, 347
 - S 164
 - Y 36
 - a 344
 - c 308
 - d 344
 - e 63
 - f 115
 - h 71, 110, 160, 346
 - i 37
 - l 30, 33, 108, 158, 343
 - m 32, 44
 - p 173, 309
 - r 27, 344
 - s 61
 - t 45, 46
 - u 51
 - z allextact 30
 - z defaultextract 31
 - z defs 49, 53, 171, 345, 346
 - z endfiltee 314
 - z finiarray 39
 - z groupperm 315
 - z ignore 32
 - z initarray 39
 - z initfirst 313
 - z interpose 78, 313

- z lazyload 83, 315, 345, 346
- z loadfltr 120, 313
- z muldefs 46
- z nodefaultlib 37, 314
- z nodefs 47, 80
- z nodelete 312
- z nodlopen 313
- z nodump 314
- z nolazyload 83, 345, 347
- z noversion 59, 143, 149
- z now 79, 92, 311, 312
- z text 125, 345
- z weakextract 30, 240
- z verbose 64
- リンカー出力
 - 共有オブジェクト 19
 - 再配置可能オブジェクト 18
 - 静的実行可能ファイル 18
 - 動的実行可能ファイル 19
- リンカーのサポートインタフェース (ld-サポート) 163
 - ld_atexit 167
 - ld_file 166
 - ld_section 166
 - ld_start 165
- リンク編集 18, 237, 258, 290
 - アーカイブ処理 30
 - 共有オブジェクトとアーカイブの混合 34
 - 共有オブジェクトの処理 31
 - 検索パス 35
 - コマンド行上のファイルの位置 34
 - 追加ライブラリの追加 32
 - 動的 248, 258, 290
 - 入力ファイルの処理 29
 - バージョン定義への結合 148, 152
 - 複数定義のシンボル 240
 - ライブラリ入力処理 30
 - ライブラリのリンクオプション 30

れ

レイジー結合 79, 91, 105

ろ

ローカルシンボル 41, 239, 240

わ

割り込み 43, 44, 58, 77, 82, 101, 141