リンカーとライブラリ

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

Part No: 819–0391–17
2009 年 4 月

原典: Linker and Libraries Guide

Part No: 817-1984–18

Revision A

090409@21990

目次

図目次

表目次

はじめに

Solaris™ オペレーティングシステム (Solaris OS) では、アプリケーション開発者はリンカー 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オブジェクトを実行します。このマニュアルは、Solaris OS リンカーの使用に関係する概念をより十分に理解することを望むエンジニアを対象にしています。

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

本書の x86 に関連する用語については、以下を参照してください。

- 「x86」は、64 ビットおよび 32 ビットの x86 互換製品系列を指します。
- 「x64」は、AMD64 または EM64T システムに関する 64 ビット特有の情報を指します。
- 「32 ビット x86」は、x86 をベースとするシステムに関する 32 ビット特有の情報を指します。

サポートされるシステムについては、Solaris OS: Hardware Compatibility List を参照してください。

お読みになる前に

このマニュアルでは、Solaris OS 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。動的実行可能ファイルと共有オブジェクトの生成および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 OS 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

ほとんどのプログラマは、このマニュアルの最初から最後までを通読する必要はありません。

内容の紹介

第 1 章「Solaris OS リンカーの紹介」では、Solaris OS でのリンク処理の概要を紹介します。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「リンカー」では、リンカーの機能について説明します。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「実行時リンカー」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「共有オブジェクト」では、共有オブジェクトの定義について記載し、そのメカニズム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「アプリケーションバイナリインタフェースとバージョン管理」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展開の管理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「サポートインタフェース」では、リンカーと実行時リンカーの処理を監視し、場合によっては修正する、インタフェース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「オブジェクトファイル形式」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「スレッド固有領域 (TLS)」では、スレッド固有領域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。

第 9 章「mapfile のオプション」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 `mapfile` 指令について説明します。この章は、上級プログラマを対象としています。

付録 A「リンカーのクイックリファレンス」では、もっと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「バージョン管理の手引き」は、共有オブジェクトのバージョン管理のための命名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「動的ストリングトークンによる依存関係の確立」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「リンカーとライブラリのアップデートおよび新機能」では、リンカーに追加された新機能と更新の概要をリリースごとに記載しています。

このマニュアルを通して、すべてのコマンド行の例は、sh(1) の構文を使用しています。すべてのプログラム例は、C 言語で記述されています。

マニュアル、サポート、およびトレーニング

Sun の Web サイトでは、次のサービスに関する情報も提供しています。

- マニュアル (http://jp.sun.com/documentation/)
- サポート (http://jp.sun.com/support/)
- トレーニング (http://jp.sun.com/training/)

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:

表 P–1 表記上の規則 *(続き)*

字体または記号	意味	例
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、`rm` *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	`sun% grep '^#define \` `XV_VERSION_STRING'`

コード例は次のように表示されます。

- C シェル

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

- C シェルのスーパーユーザー

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

- Bourne シェルおよび Korn シェル

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

- Bourne シェルおよび Korn シェルのスーパーユーザー

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

[ ] 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、英語環境での画面イメージを使っています。このため、実際に日本語環境で表示される画面イメージとこのマニュアルで使っている画面イメージが異なる場合があります。本文中で画面イメージを説明する場合には、日本語のメニュー、ボタン名などの項目名と英語の項目名が、適宜併記されています。

◆ ◆ ◆ 第 1 章

Solaris OS リンカーの紹介

このマニュアルは、Solaris OS リンカーと実行時リンカーの動作に加え、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 OS リンカーの基本的な動作には、オブジェクトの結合が含まれます。この結合によって、接続されるオブジェクトから別のオブジェクト内のシンボル定義へシンボル参照されるようになります。

このマニュアルは次の領域を扱っています。

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のデータを連結および解釈します。入力ファイルは、再配置可能オブジェクト、共有オブジェクト、またはアーカイブライブラリです。これら入力ファイルから 1 つの出力ファイルが作成されます。出力ファイルは、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です。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。

実行時リンカー
: 実行時リンカー `ld.so.1(1)` は、動的実行可能ファイルと共有オブジェクトを実行時に処理し、実行可能ファイルと共有オブジェクトを結合して、実行可能プロセスを作成します。

共有オブジェクト
: 共有オブジェクトとは、リンク編集フェーズからの出力の形式の 1 つです。共有オブジェクトを「共有ライブラリ」と呼ぶこともあります。共有オブジェクトは、強力で柔軟な実行時環境を作成する上で重要です。

オブジェクトファイル
: Solaris OS リンカーは、実行可能かつリンク可能なフォーマット (executable and linking format、ELF) に準拠したファイルを処理します。

これらの領域は、それぞれのトピックに分割できますが、重複する部分も多数あります。このマニュアルでは、相互に参照させながら、各領域について説明しています。

リンク編集

リンク編集では、一般に、コンパイラ、アセンブラ、または ld(1) によって生成されたさまざまな入力ファイルを受け取ります。リンカーは、これら入力ファイル内のデータを連結および解釈して、1 つの出力ファイルを生成します。リンカーにはさまざまなオプションを使用できますが、出力ファイル (入力再配置可能オブジェクトの連結) は次のいずれかの形式になります。

- 「再配置可能オブジェクト」– 後続のリンク編集フェーズで使用可能な、入力再配置可能オブジェクトの連結。
- 「静的実行可能ファイル」– すべてのシンボル参照を解決する入力再配置可能オブジェクトの連結。この実行可能ファイルは、実行準備が整ったプロセスを表します。25 ページの「静的実行可能ファイル」を参照してください。
- 「動的実行可能ファイル」– 実行可能プロセスを生成するときに、実行時リンカーによる割り込みを必要とする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には、実行時に結合されるシンボル参照も必要です。動的実行可能ファイルは、通常共有オブジェクトの形で 1 つ以上の依存関係を持っています。
- 「共有オブジェクト」– 実行時に動的実行可能ファイルに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、ほか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、しばしばまとめて「動的オブジェクト」と呼びます。このマニュアルでは、この動的オブジェクトに焦点を当てて説明します。

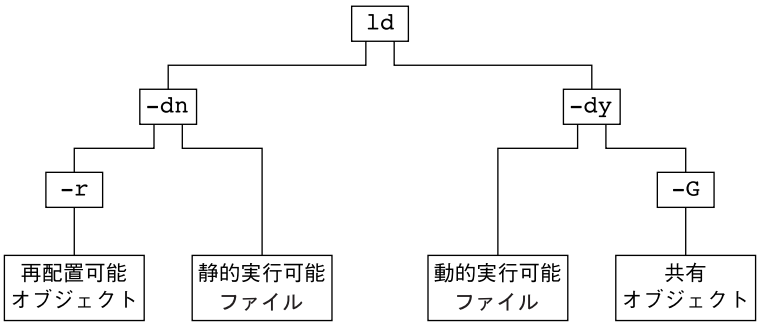


図 1–1 静的または動的リンク編集

静的実行可能ファイル

静的実行可能ファイルは、多くのリリースで作成しないように勧められています。実際、64 ビットシステムアーカイブライブラリが提供されたことはありません。静的実行可能ファイルは、システムアーカイブライブラリに反して構築されるので、実行可能ファイルにはシステム実装の詳細が含まれます。この自己内包には、多数の欠点があります。

- この実行可能ファイルは、共有オブジェクトとして提供されるシステムパッチの恩恵を受けることができません。したがって、多くのシステムの改良を利用するには、この実行可能ファイルを再構築する必要があります。
- 将来のリリースでこの実行可能ファイルを実行できなくなる可能性があります。
- システム実装の詳細を複製すると、システムのパフォーマンスに悪影響を与えます。

Solaris 10 リリースでは、32 ビットシステムアーカイブライブラリは提供されません。これらのライブラリ (特に `libc.a`) が提供されないため、特別なシステムに関する知識を持っていないかぎり、静的実行可能ファイルは作成できなくなりました。なお、リンカーの静的リンクオプションを処理する機能とアーカイブライブラリの処理に変更はありません。

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。リンカーによってこれらのオブジェクトが生成されている間、確認済みの結合要件を表す適切な記帳情報が生成されます。この情報によって、実行時リンカーは読み込み、再配置し、結合プロセスを完了できます。

プロセス実行中、実行時リンカーの機能が使用できるようになります。これらの機能は、必要に応じて共有オブジェクトを追加することによって、プロセスのアドレス領域を拡張するために使用できます。実行時リンクに組み込まれたコンポーネントのうち、もっとも一般的なのは、「動的実行可能ファイル」と「共有オブジェクト」の 2 つです。

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。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。一般的に、実行可能プロセスを作成するには、1 つまたは複数の共有オブジェクトを動的実行可能ファイルに結合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、その構造上の観点は、共有性、バージョン管理およびパフォーマンスに直接影響します。

リンカーまたは実行時リンカーによる共有オブジェクトの処理は、共有オブジェクトが使用される環境によって次のように区別されます。

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ほか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

動的リンクという言葉は、しばしば、いくつかのリンク概念を含めて使用されます。動的リンクは、リンカープロセスの動的実行可能ファイルおよび共有オブジェクトを生成する部分を指します。動的リンクは、実行可能プロセスを生成するこれらのオブジェクトの実行時リンクも指します。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実装が独立しているため、アプリケーションの安定性を維持しながら、システムを更新することができます。動的リンクは、ABI (アプリケーションバイナリインタフェース) を利用するときに必要不可欠な要素で、Solaris OS アプリケーションに適したコンパイル方式です。

アプリケーションバイナリインタフェース

システムコンポーネントとアプリケーションコンポーネントの間に定義されたバイナリインタフェースを利用すれば、これらのコンポーネントを非同期的に更新できます。Solaris OS リンカーは、これらのインタフェース上で稼動し、アプリケーションを実行できるように組み合わせます。Solaris OS リンカーによって処理さ

れるどのコンポーネントにもバイナリインタフェースがありますが、Solaris システムが提供するバイナリインタフェースを総称して、「Solaris ABI」と言います。

Solaris ABI は、「*System V* アプリケーションバイナリインタフェース」によって始まった ABI の成果の技術上の子孫です。この成果は、SPARC International, Inc.® によって行われた SPARC プロセッサ向けの追加により発展し、「SPARC Compliance Definition (SCD)」と呼ばれます。

32 ビットおよび 64 ビット環境

リンカーは 32 ビットアプリケーションおよび 64 ビットアプリケーションとして提供されています。各リンカーは 32 ビットオブジェクトおよび 64 ビットオブジェクトで動作可能です。64 ビット環境を実行するシステムでは、両方のバージョンのリンカーを実行できます。32 ビット環境を実行するシステムでは、32 ビットバージョンのリンカーのみを実行できます。詳細については、31 ページの「32 ビットリンカーと 64 ビットリンカー」を参照してください。

実行時リンカーは 32 ビットオブジェクトおよび 64 ビットオブジェクトとして提供されています。32 ビットオブジェクトは 32 ビットプロセスを実行するために使用され、64 ビットオブジェクトは 64 ビットプロセスを実行するために使用されます。

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操作に違いはありません。このマニュアルでは、多くの場合、32 ビットオブジェクトでの操作の例を使用します。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

リンカーは、たとえば `LD_LIBRARY_PATH` など、`LD_` から始まる環境変数を多数サポートしています。これらの環境変数は、この汎用形式でも使用できますが、`_32` または `_64` を接尾辞として指定することもできます (`LD_LIBRARY_PATH_64` など)。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す。またこの接尾辞は、接尾辞の付いていない汎用形式の環境変数が有効な場合でも、それに優先します。

注 – Solaris 10 よりも前のリリースでは、リンカーは値なしで指定された環境変数を無視していました。したがって、次の例では、汎用環境変数設定である /opt/lib が 32 ビットアプリケーション prog の依存関係の検索に使われていました。

```
$ LD_LIBRARY_PATH=/opt/lib  LD_LIBRARY_PATH_32=  prog
```

Solaris 10 リリースでは、接尾辞 _32 または _64 を持つ、値なしで指定された環境変数も処理されるようになりました。これらの環境変数は、関連する汎用環境変数設定を事実上取り消します。このため、前の例で、32 ビットアプリケーション prog の依存関係を検索するために、/opt/lib が使われることはありません。

このマニュアルでは、リンカーの環境変数を記述する場合は、接尾辞の付いていない汎用形式を使用します。サポートされているすべての環境変数は、ld(1) および ld.so.1(1) に定義されています。

サポートするツール

Solaris OS では、いくつかのサポートツールとライブラリも提供しています。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処理の分析や検査が行えます。これらのツールには、elfdump(1)、lari(1)、nm(1)、dump(1)、ldd(1)、pvs(1)、elf(3ELF)、およびリンカーデバッグサポートライブラリが含まれます。これらのツールについては、例を使用して詳しく説明します。

◆ ◆ ◆ 第 2 章

2 リンカー

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、リンカーのオプションによって指示され、入力部分は入力ファイルによって提供されます。

ファイルはすべて、「実行可能リンク形式」(ELF) で表現されます。ELF 書式の詳細については、第 7 章「オブジェクトファイル形式」を参照してください。この概要として、「セクション」と「セグメント」という 2 つの ELF 構造を紹介します。

セクションとは、ELF ファイル内で処理できる、分割できない最小単位のことです。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) でメモリーイメージに対応付けできる最小単位 (これ以上分割できない単位) です。

ELF セクションには多くのタイプがありますが、リンク編集フェーズに関して次の 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。
- *.symtab* や .strtab に含まれるシンボルテーブル情報や .rela.text などの再配置情報など、「リンク編集情報」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、その他のセクションを修正するためにリンカーによって解釈されます。情報セクションは、後で行われる出力ファイル処理で使用される新しい出力情報セクションの生成にも使用されます。

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。

- すべての提供オプションの検証と整合性チェック。
- 入力再配置可能オブジェクトの同じ特性を持つセクションを連結することによる出力ファイル内での新しいセクションの形成。これらの連結されたセクションは、次に、出力セグメントへと連結できます。

- 定義の参照を検証およびまとめるための再配置可能オブジェクトおよび共有オブジェクトのシンボルテーブル情報の処理。出力ファイル内での新しいシンボルテーブルまたはテーブルの生成。
- 入力再配置可能オブジェクトの再配置情報の処理、および出力ファイルを構成するセクションへのこの情報の適用。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ます。
- 作成されたすべてのセグメントを記述するプログラムヘッダーの生成。
- 必要に応じた、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の生成。

「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では、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。ただしこれらのデフォルトは、対応する `-mapfile` を指定した `M` オプションを使用して操作できます。第 9 章「mapfile のオプション」を参照してください。

リンカーの起動

リンカーは、コマンド行から直接実行することもコンパイラドライバから呼び出すようにすることもできます。次の 2 つの節では、この両方の方法を詳しく説明します。ただし、通常は、コンパイラドライバを使用することをお勧めします。コンパイル環境は、多くの場合、コンパイラドライバだけが認識し、頻繁に変化する複雑な操作の連続によって構成されています。

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の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、次のコマンドは、入力ファイル `test.o` のみを使って `a.out` という名前の動的実行可能ファイルを作成するように、リンカーに命令します。

```
$ ld test.o
```

通常、動的実行可能ファイルには、特殊な起動コードおよび終了処理コードが必要です。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コードおよび終了コードも指定できます。このコードは、実行時リンカーで正確に認識され、使用できるようにするために、正確にカプ

セル化およびラベル付けを行う必要があります。このカプセル化とラベル付けも、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ときは、コンパイラドライバを使ってリンカーを起動する必要があります。リンカーの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

リンカーを利用する一般的な方法は、言語固有のコンパイラドライバを使用する方法です。アプリケーションを構成する入力ファイルとともに、cc(1)、CC(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、次のようにコンパイルの呼び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。

---

32 ビットリンカーと 64 ビットリンカー

リンカーは 32 ビットアプリケーションおよび 64 ビットアプリケーションとして提供されています。各リンカーは 32 ビットオブジェクトおよび 64 ビットオブジェクトで動作可能です。ただし、リンク編集に 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。32 ビットリンカーも 64 ビットオブジェクトを生成できますが、生成されるオブジェクトのサイズは、.bss を除いて、2G バイトに制限されます。

デフォルトでは、コンパイラドライバにより 32 ビットリンカーが実行されます。このリンカーは、コマンド行をチェックして、リンク編集を完了するために 64 ビットリンカーを実行すべきかどうかを判別します。

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際、通常はコマンド行オプションは必要ありません。リンカーの操作モードは、コマンド行の最初の入力再配置可能オブジェクトの ELF クラスによって制御されます。mapfile またはアーカイブライブラリからだけ行われるリンクなどの特別なリンク編集は、コマンド行オブ

ジェクトの影響を受けません。これらのリンク編集はデフォルトで 32 ビットモードなので、64 ビットリンク編集を行うときはコマンド行オプションを指定する必要があります。

64 ビットリンカーは、次のいずれかの条件の下で実行されます。

- -64 オプションが使用された。
- -z altexec64 オプションが使用された。
- コマンド行の最初の再配置オブジェクトが 64 ビットである。

きわめて大規模な 32 ビットオブジェクトを作成すると、32 ビットリンカーで使用可能な仮想メモリーを使い果たしてしまうことがあります。-z altexec64 オプションを指定することで、対応する 64 ビットリンカーの使用を強制できます。64 ビットリンカーは、32 ビットオブジェクトの構築に、より大きな仮想アドレス空間を提供します。

注 – LD_ALTEXEC 環境変数を使用して、代替リンカーを指定することもできます。

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の大部分は、コンパイラドライバのコマンド行経由で渡すことができます。コンパイラオプション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。また、LD_OPTIONS 環境変数を設定して、リンカーにオプションを渡すこともできます。

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

-R オプションと -L オプションは、リンカーによって解釈されます。これらのオプションは、コンパイラドライバから受け取るコマンド行オプションより優先されます。

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調べます。どちらかの無効なオプションが検索された場合は、該当するエラーメッセージが生成されます。致命的なエラーの場合は、リンカーは強制終了します。次の例では、リンカーの検査により、不当なオプション -X と -z オプションに不当な引数が検出されています。

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

1 つの引数を必要とするオプションが 2 回指定された場合、リンカーは適切な警告を生成したあと、リンク編集を継続します。

```
$ 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の検査も行なって重大な不一致も検出します。

```
$ 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重大なエラー状態が検出されなかった場合は、リンカーは次に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「リンカーのクイックリファレンス」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、開かれ、そのファイルの ELF 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」モードでは、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」モードでは、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もっと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連結されます。「リンク編集情報」のセクションは、後で使用するために整理されます。新しい情報セクションが生成され、取って代わられるので、情報セクションは出力ファイルイメージの一部にはなりません。シンボルは、内部シンボルテーブルに集められ、検査および解決されます。このテーブルを使用して、出力イメージ内に 1 つ以上のシンボルテーブルが作成されます。

入力ファイルはリンク編集コマンド行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 オプションを使って指定します。36 ページの「追加ライブラリとのリンク」を参照してください。リンク編集時のアーカイブライブラリと共有オブジェクトの解釈は、かなり違います。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ます。アーカイブは通常、アーカイブシンボルテーブルを持つ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、アーカイブメンバーを選択して抽出できます。リンカーは、未解決のシンボル参照を使用して、アーカイブから結合プロセスの完了に必要なオブジェクトを選択します。1 つのアーカイブのすべてのメンバーを明示的に抽出することもできます。

リンカーがアーカイブから再配置可能オブジェクトを抽出するのは、次のような場合です。

- 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、シンボル参照を満たすシンボル定義が入っている場合。この参照は、「未定義」シンボルと呼ばれる場合もあります。
- 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、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ます。
- アーカイブのメンバーに、隠された可視性または保護された可視性を必要とする参照に一致するシンボル定義が含まれる場合。表 7–20 を参照してください。
- 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メンバーがすべて抽出されます。

選択式アーカイブ抽出において、ウィークシンボル参照では、-z weakextract オプションが有効になっていないかぎり、アーカイブからのオブジェクト抽出は実行されません。詳細は、45 ページの「単純な解決」を参照してください。

注 – オプション -z weakextract、 -z allextract、および -z defaultextract を使用すると、複数のアーカイブ間でアーカイブ抽出メカニズムを切り替えることができます。

選択的なアーカイブ抽出によって、リンカーは 1 つのアーカイブで複数のパスを作成します。必要に応じて、リンカー内部のシンボルテーブルに累積されているシンボル情報を満たすために、再配置可能オブジェクトが抽出されます。リンカーが、再配置可能オブジェクトを抽出せずに、アーカイブを通るフルパスを作成すると、次の入力ファイルが処理されます。

アーカイブが検出されたときに必要な再配置可能オブジェクトだけを抽出することから、コマンド行でのアーカイブの位置が重要であることがわかります。37 ページの「コマンド行上のアーカイブの位置」を参照してください。

注 – リンカーはアーカイブで複数のパスを作成してシンボルを解決しますが、このメカニズムはかなり負担が大きいものです。特に再配置可能オブジェクトのランダムな組織を含む大きなアーカイブでは、負担が大きくなります。この場合は、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整理してください。オブジェクトを整理することで、リンカーが実行するパスの数を減らすことができます。

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。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。

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は、リンカーでは使用されません。これらのセクションは、共有オブジェクトが結合されて実行可能プロセスが生成されるときに、実行時リンカーによって解釈されます。ただし、共有オブジェクトの生成は記憶されます。このオブジェクトが実行時に利用可能にしなければならない依存関係であることを示す情報が、出力ファイルイメージに保存されます。

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参照シンボルを生成するかどうかに関係なく実行されます。実行時のリンクのオーバーヘッドを最小限に抑えるために、構築中のオブジェクトからのシンボル参照を解決する依存関係だけを指定してください。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。リンカーの -z ignore オプションは、使用されていない共有オブジェクトの依存関係の記録を抑制するために使用できます。

共有オブジェクトに、ほか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できます。この処理は、すべてのコマンド行入力ファイルが処理されたあと、シンボル解決プロセスを完了するために実行されます。ただし、生成される出力ファイルイメージ内に、共有オブジェクト名は依存関係として記録されません。

コマンド行での共有オブジェクトの位置は、アーカイブ処理の場合ほど重要ではありませんが、その位置は広範囲に影響を及ぼす可能性があります。同じ名前の複数のシンボルを、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。44 ページの「シンボル解決」を参照してください。

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。遅延読み込みがない場合、実行時リンカーは指定された共有オブジェクトを同じ順序で読み込みます。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 – 多重シンボル定義は、`-m`オプションを使用して生成されるロードマップ出力で報告されます。

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が確認されますが、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、リンカーに対して必要な入力ファイルの名前を明示的につけることで指定できます。ただし、より一般的で柔軟性が高いのは、リンカーの`-l`オプションを使用する方法です。

ライブラリの命名規約

ライブラリの命名規約規約によると、共有オブジェクトは通常、接頭辞`lib`と接尾辞`.so`によって指定されます。アーカイブは、接頭辞`lib`と接尾辞`.a`によって指定されます。たとえば、`libfoo.so`は、コンパイル環境に使用できる「foo」実装の共有オブジェクトバージョンです。`libfoo.a`は、ライブラリ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ます。リンカーが`libfoo.so`を検索できない場合は、`libfoo.a`を検索してから次の検索ディレクトリに移動します。

```
$ 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。122ページの「命名規約」 および173ページの「バージョン管理ファイル名の管理」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 -l オプションを使用する場合、リンカーはまず指定された名前と一致する共有オブジェクトの指定ディレクトリを検索します。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

ライブラリ検索メカニズムにより動的モードで共有オブジェクトを検索する場合、指定したディレクトリがまず検索され、次にアーカイブライブラリが検索されます。検索タイプをより詳細に制御するには、-B オプションを使用します。

コマンド行に -B dynamic オプション、-B static オプションを指定することによって、ライブラリの検索対象をそれぞれ共有オブジェクト、アーカイブに切り替えることができます。たとえば、アプリケーションをアーカイブ libfoo.a と共有オブジェクト libbar.so にリンクするには、次のコマンドを発行します。

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

オプション -B static と -B dynamic は、正確には対称ではありません。-B static を指定すると、リンカーは、次の-B dynamic 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。しかし、-B dynamic を指定すると、リンカーは、指定されたディレクトリ内で、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。

前述の例の詳しい説明として、リンカーはまず libfoo.a を探します。次にリンカーは libbar.so を探し、見つからない場合に libbar.a を探します。

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。リンカーはアーカイブを検索して、以前に参照したことのある定義されていない仮の外部参照だけを解決します。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た後で、リンカーはコマンド行上の次の入力ファイルに移動します。

このためデフォルトでは、コマンド行上で先行するアーカイブを、後続の入力ファイルからの新しい参照の解決に使用することはありません。たとえば、次のコマンドでは、file1.c で得たシンボル参照を解決するためだけに、libfoo.a を検索するように、リンカーに指示しています。libfoo.a アーカイブは、file2.c または file3.c のシンボル参照を解決するためには、使用されません。

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

あるアーカイブからのメンバーの抽出をほかのアーカイブからのメンバーの抽出で解決しなければならないというように、アーカイブ間に相互依存関係が存在する場

合があります。依存関係が循環している場合は、前方の参照を解決するために、コマンド行上でアーカイブを繰り返し指定する必要があります。

```
$ cc -o prog .... -lA -lB -lC -lA -lB -lC -lA
```

アーカイブの繰り返し指定の決定と管理はやっかいなものです。-z rescan オプションを指定すれば、この処理は簡単になります。すべての入力ファイル処理の後、このオプションはアーカイブリスト全体を再処理します。この処理は、シンボル参照を解決する追加のアーカイブメンバーの位置を特定しようとします。アーカイブ全体を走査しても新しい再配置可能オブジェクトが抽出されないと、アーカイブの再走査は終了します。前述の例は、次のように単純化できます。

```
$ cc -o prog -z rescan .... -lA -lB -lC
```

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、複数の定義が衝突するために必要となる場合は除きます。

リンカーが検索するディレクトリ

これまでの例はすべて、コマンド行に指定されライブラリの検索場所をリンカーが認識していることを前提にしています。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、3 つの標準的なディレクトリ /usr/ccs/lib、/lib、および /usr/lib だけです。64 ビットオブジェクトをリンクする場合は、2 つの標準的なディレクトリ /lib/64 と /usr/lib/64 だけを使用します。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方法と、環境変数を使用する方法があります。

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、ライブラリ検索パス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、検索パスを変更します。たとえば、次のコマンドは、path1、/usr/ccs/lib、/lib、/usr/lib の順に libfoo を検索します。このコマンドは、path1、path2、/usr/ccs/lib、/lib、/usr/lib の順に libbar 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、リンカー専用です。これらのパス名は、作成される出力ファイルイメージには記録されません。したがって、実行時リンカーはこれらのパス名を使用できません。

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 (.) で現在のディレクトリを表すことができます。

-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。多くの場合、コンパイラドライバには、コンパイラ固有の検索パスを指定するための -Y オプションがあります。

環境変数の使用

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に付加できます。一般に、LD_LIBRARY_PATH には、コロンで区切られたディレクトリリストをとります。LD_LIBRARY_PATH のもっとも一般的な書式は、セミコロンで区切られた 2 つのディレクトリリストです。これらのリストは、コマンド行で提供される -Y リストの前後に検索されます。

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び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、dir1:dir2:path1:path2... pathn:dir3:/usr/ccs/lib:/lib:/usr/lib です。

LD_LIBRARY_PATH 定義の一部にセミコロンが指定されてない場合は、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。次の例で有効な検索パスは、path1:path2... pathn:dir1:dir2:/usr/ccs/lib:/lib:/usr/lib です。

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

注 – この環境変数は、実行時リンカーの検索パスを増強する場合にも使用できます。79 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

実行時リンカーは、デフォルトでは 2 つの場所で依存関係を検索します。32 ビットオブジェクトを処理する場合、デフォルトでは /lib と /usr/lib が検索されます。64 ビットオブジェクトを処理する場合、デフォルトでは /lib/64 と /usr/lib/64 が検索されます。このほか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。

動的実行可能ファイルまたは共有オブジェクトが別の共有オブジェクトとリンクされるとき、これらの共有オブジェクトは依存関係として記録されます。このような依存関係は、プロセスの実行中に実行時リンカーによって再配置される必要があります。動的なオブジェクトをリンクする場合は、出力ファイルに 1 つ以上の検索パスを記録できます。この検索パスは、「実行パス」と呼ばれます。実行時リンカーは、オブジェクトの実行パスを使用して、オブジェクトの依存関係を特定します。

-z nodefaultlib オプションを使用すると、実行時にデフォルトの場所を検索しない特別なオブジェクトを作成できます。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。このオプションがないと、実行時リンカーの検索パスをどのように拡張しても、最後に使用された検索パスが常にデフォルトの場所になります。

注 – デフォルトの検索パスは、実行時構成ファイルを使って管理できます。81 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。ただし、動的オブジェクトの作成者はこのファイルの存在に依存するべきではありません。常に、「実行パス」またはデフォルトの場所だけでオブジェクトの依存関係を検索できるようにしてください。

コロンで区切られたディレクトリリストを指定する -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オブジェクト内に「実行パス」を記録できます。次の例では、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。

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

共有オブジェクトの依存関係を取得するとき、実行時リンカーはまず上記パスを検索してから、デフォルトの場所を検索します。この場合、この「実行パス」は、libfoo.so.1 と libbar.so.1 の検索に使用されます。

リンカーには複数の -R オプションを指定できます。複数指定された場合は、コロンで区切って連結されます。したがって、上記の例は次のように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。390 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方法として、環境変数 LD_RUN_PATH を設定し、それをリンカーが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まったく同じですが、この両方を指定した場合は、-R が LD_RUN_PATH より優先されます。

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処理のためのコードを提供することができます。動的オブジェクトの初期設定コードは、処理中に動的オブジェクトが読み込まれるたびに、1 回ずつ実行されます。動的オブジェクトの終了コードは、動的オブジェクトが処理から読み取り解除されるか、または処理の終了のたびに 1 回ずつ実行されます。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 つのセクションタイプで組み込まれます。どちらのセクションタイプも、入力再配置可能オブジェクトの同類のセクションを連結して構築されます。

セクション .preinitarray、.initarray、および .finiarray 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、リンカーはこれらの配列を .dynamic タグペアである DT_PREINIT_[ARRAY/ARRAYSZ]、DT_INIT_[ARRAY/ARRAYSZ]、および DT_FINI_[ARRAY/ARRAYSZ] でそれぞれ識別します。これらのタグは関連するセクションを識別して、セクションを実行時リンカーによって呼び出されるようにします。「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。

注 – これらの配列に割り当てられる関数は、作成されるオブジェクトから提供する必要があります。

.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。通常コンパイラドライバは、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を使用して .init と .fini セクションを供給します。コンパイラが提供するこれらのファイルには、再配置可能オブジェクトの .init コードと .fini

コードをそれぞれ独立した関数内にカプセル化する効果があります。これらの関数は、予約シンボル名 _init と _fini によりそれぞれ識別されます。動的オブジェクトを作成する際、リンカーはこれらのシンボルを .dynamic タグの DT_INIT と DT_FINI でそれぞれ識別します。これらのタグは関連するセクションを識別して、実行時リンカーによって呼び出されるようにします。

初期設定および終了コードの実行時の詳細は、95 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。

初期設定と終了関数の登録をリンカーから直接実行するには、-z initarray オプションと -z finiarray オプションを使用します。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 .finiarray 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");
}

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

$ cc -o main -zinitarray=foo -zfiniarray=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。しかし、ほとんどのコンパイラは、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ます。たとえば、上記のコード例は、次に示す #pragma 定義を使用して書き直すことができます。これらの定義の結果、foo() に対する呼び出しが .init セクション内に配置され、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()
```

初期設定コードと終了コードが複数の再配置可能オブジェクトに分散されると、アーカイブライブラリと共有オブジェクトに組み込まれた場合とで、異なる動作をする可能性があります。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、アーカイブ内の一部オブジェクトしか抽出しない可能性があります。これらのオブジェクトは、アーカイブのメンバー全体に分散されている初期設定と終了コードの一部しか提供しない可能性があります。そして実行時に、コードのこの部分だけが実行されます。同じアプリケーションを共有オブジェクトを使用して構築した場合は、実行時に依存先が読み込まれると、累積された初期設定コードと終了コードのすべてが実行されます。

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。この分析を簡単にするには、初期設定および終了コードの内容を制限します。自己完結型の初期設定および終了コードを単純化すると、実行時の動作が予想可能になります。詳細については、96 ページの「初期設定と終了の順序」を参照してください。

初期設定コードが、dldump(3C) を使っ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、データの初期設定だけを別個に行なってください。

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すべての「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。入力再配置可能オブジェクトから渡される大域シンボルはすべて、共有オブジェクト依存関係から渡される大域シンボルとともにリンカーの内部に蓄積されます。

入力ファイルから大域シンボルが供給されると、この内部シンボルテーブル内が検索されます。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場合には、シンボル解決プロセスが呼び出されます。この解決プロセスは、再配置可能オブジェクトから渡される 2 つのエントリのうちどちらを保持するかを決定します。この解決プロセスは、共有オブジェクト依存関係への外部参照を確立する方法も決定します。

入力ファイルの処理が完了し、致命的なシンボル解決エラーが発生していない場合、リンカーは未解決のシンボル参照が残っていないかどうかを判断します。未解決のシンボル参照があると、リンク編集は強制終了します。

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、シンボル解決と未定義シンボルの処理について詳しく説明します。

シンボル解決

シンボル解決は、簡単で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。ほとんどのシンボル解決は、リンカーによって自動的に実行されます。ただし、警告診断を伴う再配置や、致命的なエラー状態の原因となる再配置もあります。

もっとも一般的で単純な解決は、あるオブジェクトから別のオブジェクト内部のシンボル定義へのシンボル参照の結合です。この結合は、2 つの再配置可能オブジェクト間、および再配置可能オブジェクトと共有オブジェクト依存関係内で検出された最初の定義の間で発生する場合があります。通常、複雑な解決は、2 つ以上の再配置可能オブジェクトの間で発生します。

2 つのシンボルの解決は、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、267 ページの「シンボルテーブルセクション」を参照してください。ただし、次の説明では、次の 3 つのシンボルタイプが特定されます。

- 「未定義シンボル」– ファイル内で参照されたが、記憶領域アドレスが割り当てられていないシンボル。
- 「一時的シンボル」– ファイル内で作成されたが、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ないシンボル。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」– 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ているシンボル。

簡単な形式では、シンボル解決で優先関係が使用されます。この関係では、定義シンボルが一時的シンボルより優先され、一時的シンボルは未定義シンボルより優先されます。

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。未定義シンボルの接頭辞は、`u_` です。一時的シンボルの接頭辞は、`t_` です。定義シンボルの接頭辞は、`d_` です。

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

int             t_bar;
int             d_bar = 1;

int 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
```

単純な解決

単純なシンボル解決は、もっとも一般的です。この場合、類似する特徴を持ち、どちらかが優先される 2 つのシンボルが検出されます。このシンボル解決は、リンカーによって自動的に実行されます。たとえば、同じ結合を持つシンボルがあり、1 つのファイルからのシンボル参照が、別のファイルの定義または一時的シンボル定義に結合されているとします。あるいは、あるファイルからの一時的シンボル定義は、ほかのファイルからの定義シンボルの定義に結合されます。この解決は、2 つの再配置可能オブジェクト間、および再配置可能オブジェクトと共有オブジェクト依存関係内で検出された最初の定義の間で発生する場合があります。

解決されるシンボルは、大域結合またはウィーク結合されます。再配置可能オブジェクト内では、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。異なる結合を伴う再配置可能オブジェクトシンボルは、わずかに変更された基本規則に従って解決されます。

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。このメカニズムでは、`#pragma` 定義を使用します。

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

int             bar = 1;
```

```
int _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。シンボル解決においては、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。

単純なシンボル解決のもう 1 つの形式である「割り込み」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で発生します。この場合、シンボルが複数回定義されていれば、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リンカーによって暗黙のうちに採用されます。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、ほか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して、別の共有オブジェクトが提供する機能を無効にすることができます。再配置可能オブジェクトと共有オブジェクトの間、または複数の共有オブジェクト間で発生する複数回定義されたシンボルは、同一に扱われます。シンボルのウィーク結合や大域結合は、これとは無関係です。最初の定義を解決することにより、シンボルの結合に関係なく、リンカーと実行時リンカーの両方が一貫して動作します。

共有オブジェクト内部で定義されたウィークシンボルを、同じ共有オブジェクトに対するシンボル割り込みと組み合わせることにより、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、システムに必要な一連の標準サービスを定義します。厳格な適合プログラムでは、これらのサービスを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。システム関数 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) が、標準 C ライブラリ内に fread(3C) を実装する基盤になることです。このため、read(2) を再定義するプログラムは、fread(3C) の実装を混乱させる可能性があります。この混乱を避けるために、ANSI C では、実装に予約され

ていない名前を実装に使用できないように定めています。次の #pragma 指令で、このような予約名だけを定義します。この名前を使用して、関数 read(2) の別名を生成します。

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

こうすることにより、ユーザーは、_read() 関数を使用している fread(3C) の実装を危険にさらすことなく、自分専用の read() 関数を自由に定義できます。

このリンカーでは、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、read() を再定義できます。前者の場合には、割り込みによって方法が決められます。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 オプションを使用して、割り込みされるすべてのシンボル参照のリストを、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んでください。

複雑な解決

複雑な解決は、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。これらの場合、リンカーは警告メッセージを生成し、もっとも適切なシンボルを選択します。このメッセージは、シンボル、相反する属性、シンボル定義の元になるファイルの識別情報を示します。次の例では、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、サイズの必要条件が異なっています。

```
$ 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
```

シンボルの整列要件が異なっている場合も、同様の診断が生成されます。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、診断を抑制できます。

異なる属性のもう 1 つの形式は、シンボルのタイプの違いです。次の例では、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。

```
$ cat foo.c
int bar()
{
        return (0);
```

```
}
$ cc -o libfoo.so -G -K pic foo.c
$ cat main.c
int     bar = 1;

int 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
```

注 – この文脈では、シンボルのタイプは ELF で使用されるタイプです。このシンボルタイプは、単純な形式であることを除けば、プログラミング言語で使用されるデータ型には関連していません。

前の例のような場合、解決が再配置可能オブジェクトと共有オブジェクト間で行われる場合に再配置可能オブジェクトの定義が使用されます。または、2 つの共有オブジェクト間で解決が行われる場合は、最初の定義が使用されます。ウィーク結合または大域結合のシンボル間でこのような解決を行うと、警告も発せられます。

リンカーの `-t` 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。

重大な解決

解決できないシンボルの矛盾は、致命的なエラー状態や該当エラーメッセージの原因となります。このメッセージは、シンボルを提供したファイルの名前とともに、シンボル名を示します。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、重大な解決エラーをすべて識別できます。

もっとも一般的な致命的エラー状態は、2 つの再配置可能オブジェクト両方が、同じ名前のウィーク以外のシンボルを定義した場合に起こります。

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

$ cat bar.c
int 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
```

foo.c と bar.c に含まれるシンボル bar の定義が、互いに矛盾しています。リンカーは、どちらを優先すべきか判別できないため、通常はエラーメッセージを出力して終了します。リンカーの -z muldefs を使用すると、エラー状態を抑制できます。このオプションによって、最初のシンボル定義が使用されます。

未定義シンボル

すべての入力ファイルを読み取り、シンボル解決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参照の内部シンボルテーブルを検索します。これらのシンボル参照は、未定義シンボルと呼ばれます。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、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出力ファイルを生成する際のデフォルト動作は、「未定義のままのシンボルが存在するかぎり、適切なエラーメッセージを出力して処理を終了する」というものです。次のように、再配置可能オブジェクト内のシンボル参照が、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();

int 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
```

同様に、共有オブジェクトを使って動的実行可能ファイルを作成する場合、未解決のままのシンボル定義が存在していると、未定義シンボルエラーが発生します。

```
$ cat foo.c
extern int bar;
int 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
```

前の例のように未定義シンボルを許可するには、リンカーの `-z nodefs` オプションを使用して、デフォルトエラー条件を抑制します。

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参照が要求されると、重大な実行時再配置エラーが発生します。このエラーは、アプリケーションをはじめて実行およびテストした際に検出される場合があります。しかし、実行パスがより複雑であるとエラー状態の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参照が、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` に次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
```

`prog` は、*libbar.so* に対する「明示的」な参照に基づいて構築されます。`libbar.so` は、`libfoo.so` に依存しています。したがって、`libfoo.so` への暗黙的参照が `prog` から確立されます。

`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参照を実行するため、`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。

このため、このタイプの結合は致命的とみなされます。暗黙的参照は、prog のリンク編集中に直接ライブラリを参照することで明示的に行います。この例で示した重大なエラーメッセージ内に必要な参照のヒントがあります。

共有オブジェクト出力ファイルの生成

リンカーが共有オブジェクト出力ファイルを生成する場合、未定義シンボルをリンク編集の後も残すことができます。このデフォルト動作により、共有オブジェクトが、依存関係として共有オブジェクトを定義する動的実行可能ファイルからシンボルを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勧めします。アプリケーションからシンボルを参照する共有オブジェクトは、extern mapfile 指令でシンボルを定義するとともに、-z defs オプションを使用できます。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。

自己完結型の共有オブジェクトは、外部シンボルへのすべての参照は指定され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する手間をユーザーにかけることなく、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、未解決のウィークシンボル参照によって重大なエラー状態は発生しません。

静的実行可能プログラムを生成中の場合は、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参照として残され、値には 0 が割り当てられます。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ます。一致が検出されない場合、実行時リンカーは重大な実行時再配置エラーを生成する代わりに、その参照がゼロのアドレスに結合されます。

従来は、これらの定義されていないウィーク参照シンボルは、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)
{
        void (* fptr)(char *);

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

libfoo.so.1 を参照するアプリケーションを構築すると、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、機能アドレスがゼロ以外をテストすると、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。

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。また、実行時シンボル結合メカニズムは、このテクニックの使用にほかの制限を課します。これらの制限によって、すべての動的オブジェクトでは一致モデルを利用できなくなります。

注 – このような未定義ウィーク参照は推奨されていません。代わりに dlsym(3C) を RTLD_DEFAULT と使用するか、または RTLD_PROBE ハンドルを使用して、シンボルの有無をテストします。112 ページの「機能のテスト」を参照してください。

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、その解決が完了するまで完全に定義されません。解決が再配置可能オブジェクトから定義シンボルに対して行われる場合、シンボルは定義順に表示されます。

シンボルグループの順序を制御する必要がある場合には、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らの相対順序が出力ファイル内に引き継がれます。

```
$ 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

入力ファイルから提供されるシンボルのほかに、追加の大域シンボル参照や大域シンボル定義をリンク編集に対して提供することができます。もっとも簡単な形式で、シンボル参照は、リンカーの `-u` オプションを使用して作成できます。リンカーの `-M` オプションと関連 `mapfile` を使用すると柔軟性が高まります。この mapfile を使用すると、大域シンボル参照およびさまざまな大域シンボル定義を定義できます。

u オプションを使用した追加シンボルの定義

`-u` オプションを指定すると、リンク編集コマンド行から大域シンボル参照を作成するためのメカニズムが使用できます。このオプションを使用して、リンク編集を完全にアーカイブから実行することができます。このオプションは、複数のアーカイブから抽出するオブジェクトを選択する際の柔軟性も高めます。アーカイブの抽出については、34 ページの「アーカイブ処理」を参照してください。

たとえば、動的実行可能プログラムを、シンボル `foo` と `bar` への参照を実行する再配置可能オブジェクト `main.o` から生成するとします。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。この再配置可能オブジェクトは、`lib2.a` に提供されたものとは機能的に異なると想定します。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
```

-u オプションは、シンボル foo への参照を生成します。この参照によって、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 への最初の参照は lib1.a が処理されてから生じる main.o 内で実行されます。このため、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。lib1.a が bar を参照する場合、処理中に再配置可能オブジェクト bar.o も 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マルチパスについては、34 ページの「アーカイブ処理」を参照してください。

---

mapfile を使用した追加シンボルの定義

広範囲な大域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。シンボル定義 mapfile のエントリの構文は次のとおりです。

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

name
このシンボル定義のセットのラベルが存在する場合は、それによってイメージ内の「バージョン定義」を識別できます。第 5 章「アプリケーションバイナリインタフェースとバージョン管理」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。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。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、同じ名前のほかの大域シンボルに対して解決されます。次の定義と別名は、作成されるオブジェクト内におけるシンボルの可視性を定義します。

`default` / `global`
このスコープの大域シンボルは、すべての外部オブジェクトに対して可視となります。このタイプのシンボルに対するオブジェクト内からの参照は実行時に結合されるため、介入が可能となります。この可視性スコープがデフォルトになりますが、これは、ほかのシンボル可視性テクニックを使って降格または削除することができます。このスコープ定義には、シンボルに STV_DEFAULT 可視性が指定された場合と同じ効果があります。表 7–20 を参照してください。

`protected` / `symbolic`
: このスコープの大域シンボルは、すべての外部オブジェクトに対して可視となります。これらのシンボルに対するオブジェクト内からの参照はリンク編集時に結合されるため、実行時の介入は防止されます。この可視性スコープは、ほかのシンボル可視性テクニックを使って降格または削除することができます。このスコープ定義には、シンボルに `STV_PROTECTED` 可視性が指定された場合と同じ効果があります。表 7–20 を参照してください。

  オブジェクトが単一のシンボルスコープを使って定義される場合、リンク編集時に、オブジェクト内のすべての再配置がそのオブジェクトに結合されます。この単一スコープでは、予約済みのシンボルさえもシンボルスコープに縮小されます。予約シンボル名のリストについては、67 ページの「出力ファイルの生成」を参照してください。

`hidden` / `local`
: このスコープの大域シンボルは、ローカル結合を持つシンボルに縮小されます。このスコープのシンボルは、ほかの外部オブジェクトから見えません。このスコープ定義には、シンボルに `STV_HIDDEN` 可視性が指定された場合と同じ効果があります。表 7–20 を参照してください。

`eliminate`
: このスコープの大域シンボルは `hidden` です。これらのシンボルテーブルのエントリは削除されます。リンカーの `-z redlocsym` オプションを使用して、ローカルシンボルを削除することもできます。

注 – `STV_` シンボル可視性属性は、コンパイラの処理するソースコードに埋め込まれたシンボル宣言に由来します。

---

symbol
: シンボル名。この名前により、修飾属性に応じて、シンボル定義またはシンボル参照が生成されます。修飾属性のないもっとも簡潔な形式で、シンボル参照が作成されます。この参照は、53 ページの「u オプションを使用した追加シンボルの定義」で説明した `-u` オプションを使用して生成する参照とまったく同じものです。 通常、このシンボル名に修飾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` スコープが定義される場合、このシンボル名を特別な「自動縮小 (auto-reduction)」指令「`*`」として定義できます。可視性が明示的に定義されていないシンボルは、生成される動的オブジェクト内のローカル結合に降格されます。明示的な可視性の定義は、`mapfile` 定義、再配置可能オブジェクト内にカプセル化された可視性定義のいずれかに起因します。

  同様に、`eliminate` スコープが定義されている場合、シンボル名を特別な「自動削除 (auto-elimination)」指令「`*`」として定義できます。可視性が明示的に定義されていないシンボルは、生成される動的オブジェクトから削除されます。

type
シンボルのタイプ属性を示します。この属性は、COMMON、data、function のいずれかです。 COMMON 属性の結果は、一時的シンボル定義になります。data および function 属性の結果は、セクションシンボル定義または絶対的なシンボル定義になります。267 ページの「シンボルテーブルセクション」を参照してください。

data 属性の結果として、OBJT シンボルが作成されます。size を伴うが value を伴わない data 属性では、このシンボルを ELF セクションに関連付けることでセクションシンボルが作成されます。このセクションは、ゼロで埋められます。

function 属性の結果として、FUNC シンボルが作成されます。size を伴うが value を伴わない function 属性では、このシンボルを ELF セクションに関連付けることにより、セクションシンボルが作成されます。このセクションには、空の関数戻り値 void (*)(void) が割り当てられます。

data または function 属性とともに value が指定されると、絶対値を表す ABS セクションインデックスを伴う適切なシンボルタイプが生成されます。

セクションデータシンボルの作成は、フィルタの作成時に役立ちます。実行可能ファイルからフィルタのセクションデータシンボルへの外部参照により、生成中のコピーが適切に再配置されます。147 ページの「コピー再配置」を参照してください。

value
値の属性を示します。この属性は、v数字の形式をとります。この属性により、シンボル定義が作成されます。

size
サイズの属性を示します。この属性は、s数字の形式をとります。この属性により、シンボル定義が作成されます。

information
このキーワードは、シンボルに次の追加情報を提供します。

AUXILIARY *name*
このシンボルが共有オブジェクト *name* に対する補助フィルタであることを示します。131 ページの「補助フィルタの生成」を参照してください。

DIRECT
このシンボルを直接結合する必要があることを示します。このキーワードをシンボル定義で使用すると、参照が、構築中のオブジェクト内から定義に直接結合されます。このキーワードをシンボル参照で使用すると、定義を提供する依存関係に直接結合されます。86 ページの「直接結合」を参照してください。このキーワードを PARENT キーワードとともに使用して、実行時に任意の親への直接結合を確立することもできます。

EXTERN
: シンボルが、作成されるオブジェクトの外部で定義されていることを示します。通常、このキーワードは、コールバックルーチンへのラベル付けで定義されます。このキーワードを使用して、`-z defs` オプションで示される未定義シンボルを抑制できます。

  このキーワードは、シンボル参照を生成する場合にのみ有効です。このシンボルの定義が、リンク編集時に結合されるオブジェクト内部で生成された場合には、暗黙的に無視されます。

FILTER *name*
: このシンボルが共有オブジェクト *name* のフィルタであることを示します。128 ページの「標準フィルタの生成」を参照してください。フィルタシンボルは、入力再配置可能オブジェクトから提供される補助実装を必要としません。したがって、シンボルの種類を定義してこの指令を使用し、絶対シンボルテーブルエントリを作成します。

NODIRECT
: このシンボルを直接結合してはならないことを示します。この状態は、作成されるオブジェクト内からの参照と外部参照に適用されます。86 ページの「直接結合」を参照してください。このキーワードを PARENT キーワードとともに使用して、実行時に任意の親への直接結合を回避することもできます。

PARENT
: シンボルが作成中のオブジェクトの親で定義されることを示します。親とは、実行時にこのオブジェクトを明示的な依存関係として参照するオブジェクトです。親は、`dlopen(3C)` を使用して、このオブジェクトを実行時に参照することもできます。通常、このキーワードは、コールバックルーチンへのラベル付けで定義されます。このキーワードを DIRECT または NODIRECT キーワードとともに使用して、親への直接的または間接的な参照を個別に確立することもできます。このキーワードを使用して、`-z defs` オプションで示される未定義シンボルを抑制できます。

  このキーワードは、シンボル参照を生成する場合にのみ有効です。このシンボルの定義が、リンク編集時に結合されるオブジェクト内部で生成された場合には、暗黙的に無視されます。

dependency
: この定義が継承するバージョン定義を示します。第 5 章「アプリケーションバイナリインタフェースとバージョン管理」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令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プログラム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ます。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「アプリケーションバイナリインタフェースとバージョン管理」に記載されています。

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。

次の節では、mapfile 構文を使用した例をいくつか示します。

シンボル参照の定義

次の例では、3つのシンボル参照を定義する方法を示します。これらの参照を使用して、アーカイブのメンバーを抽出します。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
void foo()
{
        (void) printf("foo: called from lib.a\n");
}
$ cat bar.c
void bar()
{
        (void) printf("bar: called from lib.a\n");
}
$ cat main.c
extern  void    foo(), bar();

void main()
{
        foo();
        bar();
}
$ ar -rc lib.a foo.o bar.o main.o
$ cat mapfile
{
        local:
                foo;
                bar;
        global:
                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、61 ページの「シンボル範囲の縮小」で説明しています。

絶対シンボルの定義

次の例では、2 つの絶対シンボル定義を定義する方法を示します。そして、これらの定義を使用して、入力ファイル `main.c` からの参照を解決します。

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

void 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。`size` が関連付けられるが、`value` は関連付けられない単純な `mapfile` 定義では、データ記憶域が作成されます。この場合、シンボル定義にはセクションインデックスが伴います。ただし、`mapfile` 定義に `value` を関連付けると、絶

対シンボルが作成されます。シンボルが共有オブジェクト内で定義される場合、絶対定義は避けるようにしてください。61 ページの「シンボル定義の増強」を参照してください。

一時的シンボルの定義

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。ほか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、ほかのシンボルタイプとは異なり、その「値」の属性によって、その配列要件が示される点です。そのため、リンク編集の入力ファイルから入手される一時的定義の再配列に mapfile 定義を使用できます。

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];

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

シンボル定義の増強

共有オブジェクト内での絶対データシンボルの作成は避けるべきです。通常、動的実行可能ファイルから、共有オブジェクト内のデータ項目への外部参照には、コピー再配置の作成が必要になります。147 ページの「コピー再配置」を参照してください。このような再配置を行う場合は、データ項目をデータ記憶領域と関連付けるべきです。この関連付けは、再配置可能なオブジェクトファイル内にシンボルを定義することで行うことができます。この関連付けは、mapfile 内でシンボルを size 宣言あり、value 宣言なしで定義しても行うことができます。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。

データシンボルにはフィルタを適用できます。127 ページの「フィルタとしての共有オブジェクト」を参照してください。このようなフィルタ適用を行うため、オブジェクトファイル定義は mapfile 定義で増強できます。次の例では、関数定義とデータ定義を含むフィルタを作成します。

```
$ cat mapfile
{
        global:
                foo = FUNCTION FILTER filtee.so.1;
                bar = DATA S0x4 FILTER filtee.so.1;
        local:
                *;
};
$ cc -o filter.so.1 -G -Kpic -h filter.so.1 -M mapfile -R.
$ nm -x filter.so.1 | egrep "foo|bar"
[39]    |0x000102b0|0x00000004|OBJT |GLOB |0    |12     |bar
[45]    |0x00000000|0x00000000|FUNC |GLOB |0    |ABS    |foo
$ elfdump -y filter.so.1 | egrep "foo|bar"
       [1]  F        [0] filtee.so.1        bar
       [7]  F        [0] filtee.so.1        foo
```

実行時に、外部オブジェクトからこれらのシンボルのいずれかへの参照は、「フィルティー」内の定義に解決されます。

シンボル範囲の縮小

mapfile 内のローカル範囲を持つようにシンボル定義を定義するとシンボルの最終的な結合を縮小できます。このメカニズムによって、入力の一部として生成ファイルを使用する将来のリンク編集でシンボルが表示されなくなります。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、ほか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、ほか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。これらのファイルを使用して共有オブジェクトを作成すると、通常、次のように大域範囲が指定された 3 つのシンボルが作成されます。

```
$ 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 libfoo.so.1 -G foo.c bar.c
$ nm -x libfoo.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
```

これで、libfoo.so.1 により提供された機能を、別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 への参照は、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 への直接参照も可能です。ただし、この可視性は危険な結果を招く場合があります。関数 foo の基礎となるインプリメンテーションは、後から変更することがあるためです。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す可能性があります。

また、シンボル bar と str を大域結合すると、同じ名前のシンボルによって割り込まれる可能性があります。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、45 ページの「単純な解決」 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できます。次の例では、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利用できなくなっています。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。

```
$ cat mapfile
{
        local:
                bar;
                str;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ nm -x libfoo.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
```

このようなシンボル範囲の縮小には、このほか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。シンボル再配置のオーバーヘッドの詳細は、146 ページの「再配置が実行されるとき」を参照してください。

リンク編集の間に処理されるシンボル数が多くなると、`mapfile` 内で各ローカル範囲の縮小を定義するのが困難になります。代わりとなる、より柔軟なメカニズムを使用すると、維持しなければならない大域シンボルの点で共有オブジェクトのインタフェースを定義できます。大域シンボルを定義すると、リンカーはその他のシンボルすべてをローカル結合にすることができます。このメカニズムは、特別な自動縮小指令の「*」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、生成される出力ファイル内で必要な唯一の大域シンボルとして `foo` を定義します。

```
$ cat mapfile
ISV_1.1 {
        global:
                foo;
        local:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ nm -x libfoo.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 `libfoo.so.1.1` も `mapfile` 指令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。バージョン定義はできるだけ作成してくださ

い。バージョン定義によって、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。第5章「アプリケーションバイナリインタフェースとバージョン管理」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。

バージョン名を指定する場合は必ず、「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
ISV_1.1 {
        global:
                foo;
};
$ cc -o libfoo.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 libfoo.so.1
```

`-B local` オプションを使用して、コマンド行から自動縮小指令「`*`」を表明することができます。前の例は、次のようにコンパイルすることもできます。

```
$ cc -o libfoo.so.1 -M mapfile -B local -G foo.c bar.c
```

実行可能ファイルまたは共有オブジェクトを生成すると、シンボルの縮小によって、出力イメージ内にバージョン定義が記録されます。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、自動縮小指令が指定された前の `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。

```
$ cat mapfile
ISV_1.1 {
        global:
                foo;
        local:
                *;
};
$ ld -o libfoo.o -M mapfile -r foo.o bar.o
```

```
$ nm -x libfoo.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
```

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 からバージョン管理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 libfoo.so.1 -G libfoo.o
$ nm -x libfoo.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、通常、実行可能ファイルまたは共有オブジェクトが作成されたときに行う必要があります。ただし、再配置可能オブジェクトが作成されたときは、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 libfoo.o -M mapfile -B reduce -r foo.o bar.o
$ nm -x libfoo.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
```

シンボル削除

シンボル縮小の拡張の 1 つは、オブジェクトのシンボルテーブルから特定のシンボルエントリを削除することです。局所シンボルは、オブジェクトの .symtab シンボルテーブルだけで管理されます。このテーブルは、リンカーの -s オプションまたは strip(1) を使用して、オブジェクトからすべて削除できます。しかし、.symtab シンボルテーブルは削除しないで、特定の局所シンボルだけを削除したいこともあります。

シンボル削除は、mapfile キーワード ELIMINATE を使用して実行できます。local 指令と同様に個別にシンボルを定義することも、特殊な自動削除指令「*」としてシンボル名を定義することもできます。次の例では、前述のシンボル縮小の例で使用したシンボル bar を削除しています。

```
$ cat mapfile
ISV_1.1 {
        global:
```

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

-B eliminate オプションを使用して、コマンド行から「自動削除」指令「*」を表明することもできます。

外部結合

作成するオブジェクトのシンボル参照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ます。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。一般にオブジェクトは 1 つずつ検索されますが、その際、動的実行可能プログラムから、オブジェクトが読み込まれた順に各依存関係が処理されます。

オブジェクトは、直接結合を使用するように作成することもできます。この方法では、シンボル参照と、シンボル定義を提供するオブジェクトとの関係は、作成され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、参照先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できます。86 ページの「直接結合」を参照してください。

文字列テーブルの圧縮

リンカーは、重複したエントリと末尾部分文字列を削除することによって、文字列テーブルを圧縮します。この圧縮により、どのような文字列テーブルでもサイズが相当小さくなります。たとえば、.dynstr テーブルを圧縮すると、テキストセグメントが小さくなるため、実行時のページング作業が減ります。このような利点があるため、文字列テーブルの圧縮はデフォルトで有効に設定されています。

非常に多くのシンボルを提供するオブジェクトによって、文字列テーブルの圧縮のためにリンク編集時間が延びる可能性があります。開発時にこの負担を避けるには、リンカーの -z nocompstrtab オプションを使用してください。リンク編集時に行われる文字列テーブルの圧縮は、リンカーのデバッグトークン -D strtab,detail を使用して表示できます。

出力ファイルの生成

入力ファイルの処理とシンボル解決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、リンカーは出力ファイルを生成します。リンカーは、出力ファイルの完成に必要な追加セクションをまず生成します。これらのセクションには、すべての入力ファイルから解決済みの大域およびウィークシンボル情報とともに局所シンボル定義を含むシンボルテーブルが含まれます。

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。すべての出力セクション情報が設定された後、出力ファイルサイズの合計が計算されます。次に出力ファイルイメージが適宜作成されます。

動的実行可能プログラム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` とその関連文字列テーブル `.dynstr` には、レジスタ、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、実行時処理イメージの一部としてマッピングされる `text` セグメントの一部となります。`mmap(2)` のマニュアルページを参照してください。このマッピングにより、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` とその関連文字列テーブル `.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られません。これらのセクションは、リンカーの `-s` オプションを使用して、または、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。これらのシンボルは、リンクプロセスに対して特別な意味を持っています。コードでは、これらのシンボルを定義しないでください。

`_etext`
: すべての読み取り専用情報のあとの最初の場所は、一般にテキストセグメントと呼ばれます。

`_edata`
: 初期化されたデータのあとの最初の位置。

`_end`
: すべてのデータのあとの最初の位置。

`_DYNAMIC`
: `.dynamic` 情報セクションのアドレス。

`_END_`
: `_end` と同じ。このシンボルは、`_START_` シンボルとともに、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簡単な手段を提供します。

_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) への位置独立の参照。このテーブルは、-K pic オプションを指定してコンパイルしたオブジェクトで発生する、位置独立のデータ参照から構築されます。139 ページの「位置独立のコード」を参照してください。

_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、位置独立の参照。このテーブルは、-K pic オプションを指定してコンパイルしたオブジェクトで発生する、位置独立の関数参照から構築されます。139 ページの「位置独立のコード」を参照してください。

START
: テキストセグメント内の最初の位置。このシンボルは、_END_ シンボルとともに、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簡単な手段を提供します。

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、そのシンボルが使用されます。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。

ハードウェアとソフトウェア機能の特定

再配置可能オブジェクトのハードウェア機能とソフトウェア機能は、一般的にコンパイル時に記録されます。リンカーは入力再配置可能オブジェクトの機能を組み合わせて、出力ファイルの最終機能セクションを作成します。245 ページの「ハードウェアおよびソフトウェア機能に関するセクション」を参照してください。

さらに、リンカーが出力ファイルを作成するときにも機能を定義できます。これらの機能は、mapfile とリンカーの -M オプションを使用して特定します。mapfile を使用して定義した機能は、入力再配置可能オブジェクトから提供される機能を増強したり、無効にしたりすることができます。

次の節では、mapfile を使用して機能を定義する方法を説明します。

ハードウェア機能の特定

オブジェクトのハードウェア機能は、オブジェクトを正しく実行するために必要なプラットフォームのハードウェア要件を特定します。この要件の例としては、一部の x86 アーキテクチャーで利用できる MMX または SSE を必要とするコードの特定があります。

ハードウェア機能要件は、次の mapfile 構文を使用して特定できます。

```
hwcap_1 = TOKEN | Vval [ OVERRIDE ];
```

hwcap_1 宣言は 1 つ以上のトークンで修飾されます。これはハードウェア機能のシンボル表現です。さらに、別の方法として、より多くの機能の 1 つを表す数値に V という接頭辞をつけて指定できます。SPARC プラットフォームでは、ハードウェア機能は sys/auxv_SPARC.h の AV_ の値として定義されます。x86 プラットフォームでは、ハードウェア機能は sys/auxv_386.h の AV_ の値として定義されます。

次の x86 の例では、オブジェクト foo.so.1 に必要なハードウェア機能として MMX と SSE が宣言されています。

```
$ egrep "MMX|SSE" /usr/include/sys/auxv_386.h
#define AV_386_MMX    0x0040
#define AV_386_SSE    0x0800
$ cat mapfile
hwcap_1 = SSE MMX;
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x840  [ SSE  MMX ]
```

再配置可能オブジェクトには、ハードウェア機能の値を含めることができます。リンカーは、複数の入力再配置可能オブジェクトからのハードウェア機能値を組み合わせます。この結果生じる CA_SUNW_HW_1 の値は、関連入力値のビット単位の OR となります。デフォルトでは、これらの値は、mapfile で指定されたハードウェア機能と組み合わせられます。

出力ファイルのハードウェア機能要件は、OVERRIDE キーワードを使用して mapfile から明示的に制御できます。OVERRIDE キーワードは、ハードウェア機能値 0 とともに、構築中のオブジェクトからハードウェア機能要件を事実上削除します。

```
$ elfdump -H foo.o

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x840  [ SSE  MMX ]
$ cat mapfile
hwcap_1 = V0x0 OVERRIDE;
$ cc -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
$
```

オブジェクトが定義したハードウェア機能要件は、実行時リンカーによってプロセスで利用できるハードウェア機能に対して検証されます。ハードウェア機能要件の一部を満足できない場合、そのオブジェクトは実行時に読み込みされません。たとえば、SSE 機能がプロセスで利用できない場合、ldd(1) は次のエラーを示します。

```
$ ldd prog
        foo.so.1 =>      ./foo.so.1  - hardware capability unsupported: \
                         0x800 [ SSE ]
        ....
```

異なるハードウェア機能を利用する動的オブジェクトは、フィルタを使用して柔軟な実行時環境を提供できます。385 ページの「ハードウェア機能固有の共有オブジェクト」を参照してください。

ソフトウェア機能の特定

オブジェクトのソフトウェア機能は、プロセスのデバッグまたは監視にとって重要なことがあるソフトウェアの特徴を特定します。現在、認識されているソフトウェア機能だけが、オブジェクトによるフレームポインタ使用に関係します。

オブジェクトは、フレームポインタ使用を認識することを示せます。この状態は、フレームポインタを使用中または未使用として宣言することで、修飾されます。

ソフトウェア機能フラグは、`sys/elf.h` で定義されています。

```
#define  SF1_SUNW_FPKNWN    0x001
#define  SF1_SUNW_FPUSED    0x002
```

これらのソフトウェア機能要件は、次の `mapfile` 構文を使用して特定できます。

```
sfcap_1 = TOKEN | Vval [ OVERRIDE ];
```

`sfcap_1` の宣言は、トークン `FPKNWN` と `FPUSED` で修飾できます。または、これらの状態を表す数値を代わりに使用することもできます。

再配置可能オブジェクトには、ソフトウェア機能の値を含めることができます。リンカーは、複数の入力再配置可能オブジェクトからのソフトウェア機能値を組み合わせます。ソフトウェア機能は、`mapfile` も提供されます。デフォルトでは、`mapfile` のすべての値が、再配置可能オブジェクトで提供される値と組み合わせられます。

出力ファイルのソフトウェア機能要件は、`OVERRIDE` キーワードを使用して `mapfile` から明示的に制御できます。`OVERRIDE` キーワードは、ソフトウェア機能値 `0` とともに、構築中のオブジェクトからソフトウェア機能要件を事実上削除します。

```
$ elfdump -H foo.o

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_SF_1     0x3  [ SF1_SUNW_FPKNWN  SF1_SUNW_FPUSED ]
$ cat mapfile
```

```
sfcap_1 = V0x0 OVERRIDE;
$ cc -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
$
```

ソフトウェア機能フレームポインタの処理

2 つのフレームポインタ入力値からの CA_SUNW_SF_1 値は、次のように計算されます。

表 2–1 CA_SUNW_SF_1 フレームポインタフラグ組み合わせ状態テーブル

入力ファイル 1	入力ファイル 2		
	SF1_SUNW_FPKNWN **SF1_SUNW_FPUSED**	**SF1_SUNW_FPKNWN**	**<unknown>**
SF1_SUNW_FPKNWN **SF1_SUNW_FPUSED**	SF1_SUNW_FPKNWN SF1_SUNW_FPUSED	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN SF1_SUNW_FPUSED
SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN	SF1_SUNW_FPKNWN
<unknown>	SF1_SUNW_FPKNWN SF1_SUNW_FPUSED	SF1_SUNW_FPKNWN	<unknown>

この計算は、再配置可能オブジェクト値と mapfile 値にそれぞれ適用されます。.SUNW_cap セクションが存在しない場合や、このセクションに CA_SUNW_SF_1 の値が含まれない場合、SF1_SUNW_FPKNW フラグも SF1_SUNW_FPUSED フラグも設定されていない場合は、オブジェクトのフレームポインタソフトウェア機能は不明になります。

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が、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ます。1 つは、位置に依存するコードによって発生するテキスト再配置です。この状態の詳細については、139 ページの「位置独立のコード」を参照してください。もう 1 つは、ディスプレイスメント再配置に関連して発生します。ディスプレイスメント再配置については、次の項で詳しく説明します。

ディスプレイスメント再配置

データ項目 (コピー再配置で使用可能) にディスプレイス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、147 ページの「コピー再配置」を参照してください。

ディスプレイスメント再配置は、再配置されるオフセットと再配置ターゲットが両方とも同じ位置だけ離れているかぎり有効です。コピー再配置では、共有オブジェクト内の大域データ項目が実行可能ファイルの .bss にコピーされます。このコピーは、実行可能ファイルの参照専用テキストセグメントを保持します。コピーされるデータにディスプレイス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スメントであったりすると、ディスプレイスメント再配置は無効になります。

ディスプレイスメント再配置の問題を検知するために、次の 2 つの領域で検証が試みられます。

- 最初は、共有オブジェクトの生成時に行われます。コピー再配置可能なデータ項目がディスプレイス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、それらに対しフラグが立てられます。リンカーが共有オブジェクトを構築する際には、データ項目に対しどのような外部参照がされるかは不明です。したがって、フラグが立てられたデータ項目は、エラーを引き起こす可能性があります。
- 次の検証は、実行可能ファイルの生成時に行われます。コピー再配置のデータがディスプレイスメント再配置を伴う場合は、コピー再配置の作成に対しフラグが立てられます。

  しかし、リンク編集で共有オブジェクトを作成するときに、共有オブジェクトに適用されたディスプレイスメント再配置が完了することがあります。これらのディスプレイスメント再配置には、フラグが立てられていない可能性があります。フラグの立てられていない共有オブジェクトを参照する実行可能ファイルのリンク編集では、コピー再配置のデータで有効になっているディスプレイスメントは不明となります。

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ス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 7–34 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、ディスプレイスメント再配置が適用される大域データ項目 bar[] を持つ共有オブジェクトを作成するとします。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、この状態に対する警告を出します。

```
$ cc -G -o libfoo.so.1 -z verbose -K pic 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
```

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、コピー再配置が作成されます。このコピーは、無効なディスプレイスメント再配置の原因となります。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
```

注 – ldd(1) で -d、-r のいずれかのオプションを指定すると、ディスプレイス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。

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。静的な定義を使用するか、リンカーの範囲指定を使用してください。61 ページの「シンボル範囲の縮小」を参照してください。この種の再配置の問題は、機能インタフェースを使用して共有オブジェクト内のデータにアクセスすれば、回避することができます。

デバッグ支援

Solaris OS リンカーには、デバッギングライブラリが付いています。このライブラリを使用すると、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、ユーザーのアプリケーションおよびライブラリのリンク編集を理解およびデバッグする場合に役立ちます。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。ただし、この情報の正確な形式は、リリースごとに若干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、デバッギング出力の中には見慣れないものがあるかもしれません。しかし、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。作成されるすべての出力は、標準エラーに送られます。このオプションは、1 つまたは複数のトークンで増強し、必要なデバッギン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、コマンド行で -D help を入力すれば表示できます。

```
$ ld -Dhelp
............
debug: files      display input file processing (files and libraries)
............
```

ほとんどのコンパイラドライバは、前処理フェーズ中に -D オプションを解釈します。このため、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。この構文は、リンクを編集するときにどのライブラリが使用されているかを判別するときに利用できます。アーカイブから抽出されたオブジェクトもこの構文で表示されます。

```
$ 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 のリンク編集を満たすために、メンバー 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 回検索されていることに注意してください。診断内に「(again)」が複数個含まれていることから、このアーカイブが lorder(1) や tsort(1) による並べ替えの候補であることがわかります。

symbols 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メンバーが抽出されたか、また、最初のシンボル参照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 ]
............
```

シンボル foo は、main.o によって参照されます。このシンボルは、リンカーの内部シンボルテーブルに追加されます。このシンボル参照によって、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、symbols 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決を監視できます。

```
$ 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 ]
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
.............
```

main.o からの、オリジナルの未定義シンボル foo が、アーカイブメンバー 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、デバッギン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関するアクティビティーを監視するには、リンク編集コマンド行に直接 -D オプションを配置します。このオプションはオンとオフを切り替えることができます。次の例では、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。

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

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。31 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。

◆ ◆ ◆ 第 3 章

3 実行時リンカー

「動的実行可能プログラム」を初期化および実行するとき、アプリケーションとその依存関係を結合させるために、「インタプリタ」が呼び出されます。Solaris OS では、このインタプリタを実行時リンカーと呼びます。

動的実行可能プログラムのリンク編集中に、特殊な `.interp` セクションとそれに関連するプログラムヘッダーが作成されます。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るデフォルトの名前は 実行時リンカーの名前で、32 ビットの実行プログラムの場合は `/usr/lib/ld.so.1`、 64 ビットの実行プログラムの場合は `/usr/lib/64/ld.so.1` となります。

注 – `ld.so.1` は、共有オブジェクトの特殊なケースです。ここではバージョン番号 1 が使われています。しかし、Solaris OS の今後のリリースによってバージョンアップされる可能性があります。

動的オブジェクトの実行プロセス中に、カーネルはファイルを読み込んで、プログラムのヘッダー情報を読み取ります。283 ページの「プログラムヘッダー」を参照してください。この情報を使って、カーネルは必要なインタプリタの名前を検出します。カーネルは、このインタプリンタを読み込んで制御を移し、インタプリタがアプリケーションの実行を続行するために十分な量の情報を転送します。

アプリケーションの初期化に加え、実行時リンカーは、アプリケーションが自分のアドレス空間を拡張できるようにするサービスも提供します。この処理には、追加のオブジェクトの読み込みとこれらのオブジェクトが提供するシンボルへの結合が含まれます。

実行時リンカーは次の処理を実行します。

- 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読み込みを行い、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。
- 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。
- アプリケーションに制御を渡す。
- アプリケーションの実行中に、遅延された関数の結合を実行するよう要求される。
- アプリケーションが実行時リンカーサービスに、`dlopen(3C)` によって追加のオブジェクトを入手するよう要求し、`dlsym(3C)` 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よう要求する。

共有オブジェクトの依存性

実行時リンカーがプログラムのメモリーセグメントを作成するとき、依存性は、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かを示します。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。

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。

共有オブジェクトの依存関係の検索

動的実行可能プログラムのリンク中に、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可能プログラム内に記録されます。

実行時リンカーはこの依存情報を使用して、関連オブジェクトを検索して読み込みます。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。

動的実行可能プログラムの依存関係がすべて読み込まれると、各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。

実行時リンカーが検索するディレクトリ

実行時リンカーは、デフォルトでは 2 つの場所で依存関係を検索します。32 ビットオブジェクトを処理する場合、デフォルトでは /lib と /usr/lib が検索されます。64 ビットオブジェクトを処理する場合、デフォルトでは /lib/64 と /usr/lib/64 が検索されます。単純なファイル名で指定された依存関係の前には、このデフォルトのディレクトリ名が付きます。このパス名を使用して、実際のファイルを見つけます。

動的実行可能プログラムまたは共有オブジェクトの依存関係は、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 =>     /lib/libc.so.1
        libm.so.2 =>     /lib/libm.so.2
```

ファイル /usr/bin/cat には依存関係があり、ファイル libc.so.1 と libm.so.2 を必要としています。

オブジェクトに記録されている依存関係は、elfdump(1) を使用して調べることができます。このコマンドを使用すると、ファイルの .dynamic セクションを表示して、NEEDED タグがついているエントリを探すことができます。次の例では、前の ldd(1) の例に示されていた依存関係 libm.so.2 は、ファイル /usr/bin/cat に記録されません。ldd(1) が、指定されたファイルの依存関係の全体を示し、libm.so.2 は実際には /lib/libc.so.1 の依存関係となります。

```
$ elfdump -d /usr/bin/cat

Dynamic Section:  .dynamic:
     index  tag                value
       [0]  NEEDED            0x211               libc.so.1
       ...
```

上記の elfdump(1) の例では、依存関係は単純なファイル名として表示されています。つまり、ファイル名に「/」が含まれていません。実行時リンカーが一連のデフォルト検索規則に従ってパス名を生成するためには、単純なファイル名を使用する必要があります。「/」が組み込まれたファイル名は、そのまま使用されます。

単純なファイル名の記録は、標準的でもっとも柔軟性の高い、依存関係を記録するメカニズムです。リンカーに -h オプションを指定すると、依存関係内の単純な名前が記録されます。122 ページの「命名規約」および 123 ページの「共有オブジェクト名の記録」を参照してください。

通常、依存関係は、/lib と /usr/lib または /lib/64 と /usr/lib/64 以外のディレクトリに配布されます。動的実行可能プログラムまたは共有オブジェクトが、ほかの

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。

追加の検索パスは、オブジェクトごとに、オブジェクトのリンク編集中に「実行パス」を記録して指定できます。この情報の記録方法の詳細については、40 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。

実行パスの記録は、elfdump(1) を使用して表示できます。RUNPATH タグの付いた .dynamic エントリを例示します。次の例では、 prog は libfoo.so.1 上に依存関係を持っています。実行時リンカーは、デフォルトの場所を調べる前に、ディレクトリ /home/me/lib と /home/you/lib を検索しなければなりません。

```
$ elfdump -d prog | egrep "NEEDED|RUNPATH"
       [1]  NEEDED            0x4ce              libfoo.so.1
       [3]  NEEDED            0x4f6              libc.so.1
      [21]  RUNPATH           0x210e             /home/me/lib:/home/you/lib
```

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 群のひとつを設定することです。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。次の例では、上記の例のファイル prog は、現在のカレントディレクトリ内で検出された libfoo.so.1 に結合されます。

```
$ LD_LIBRARY_PATH=. prog
```

環境変数 LD_LIBRARY_PATH の使用は、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は有用ですが、製品版ソフトウェアの場合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可能プログラムは、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、39 ページの「環境変数の使用」 と 40 ページの「実行時リンカーが検索するディレクトリ」 で説明しているとおり、LD_LIBRARY_PATH はリンカーに影響を及ぼします。

環境変数で検索パスを指定した場合、64 ビットの実行プログラムが、目的の名前と一致する 32 ビットのライブラリが格納されているパスを検索することもありえます。あるいはその逆もありえます。このような場合、実行時リンカーは一致しない 32 ビットのライブラリを拒否し、検索を続行して目的の名前と一致する有効な 64 ビットのライブラリを探します。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。この拒否を詳細に監視するには、LD_DEBUG 環境変数を設定して、files のトークンを取り込みます。114 ページの「デバッギングライブラリ」を参照してください。

```
$ LD_LIBRARY_PATH=/lib/64 LD_DEBUG=files /usr/bin/ls
...
00283: file=libc.so.1;  needed by /usr/bin/ls
00283:
00283: file=/lib/64/libc.so.1  rejected: ELF class mismatch: 32–bit/64–bit
00283:
00283: file=/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=/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す。アプリケーションを実行しようとすると、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
        libfoo.so.1 =>   (file not found)
        libc.so.1 =>     /lib/libc.so.1
        libm.so.2 =>     /lib/libm.so.2
$ 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

実行時リンカーが使用するデフォルトの検索パスは、32 ビットアプリケーションの場合、/lib と /usr/lib です。64 ビットアプリケーションの場合、デフォルト検索パスは /lib/64 と /usr/lib/64 です。このような検索パスを管理するには、crle(1) ユーティリティーで作成する実行時構成ファイルを使用します。このファイルは、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。

構成ファイルが作成されるデフォルトの場所は、32 ビットアプリケーションの場合は /var/ld/ld.config、64 ビットアプリケーションの場合は /var/ld/64/ld.config です。このファイルは、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、実行時リンカーの 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、さまざまな動的ストリングトークンを展開できます。このようなトークンは、フィルタ、「実行パス」、および依存関係の定義に利用できます。

- $HWCAP – 異なるハードウェア機能を提供するオブジェクトを配置できるディレクトリを示します。385 ページの「ハードウェア機能固有の共有オブジェクト」を参照してください。
- $ISALIST – 当該プラットフォームで実行できるネイティブな命令セットに展開します。387 ページの「命令セット固有の共有オブジェクト」を参照してください。
- $ORIGIN – 現在のオブジェクトのディレクトリの場所を示します。390 ページの「関連する依存関係の配置」を参照してください。
- $OSNAME – オペレーティングシステムの名前に展開します。389 ページの「システム固有の共有オブジェクト」を参照してください。
- $OSREL – オペレーティングシステムのリリースレベルに展開します。389 ページの「システム固有の共有オブジェクト」を参照してください。
- $PLATFORM – 当該マシンのプロセッサタイプに展開します。389 ページの「システム固有の共有オブジェクト」を参照してください。

再配置処理

アプリケーションが要求する依存関係をすべて読み込んだ後、実行時リンカーは各オブジェクトを処理し、必要な再配置すべてを実行します。

オブジェクトのリンク編集中に、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、動的実行可能ファイルまたは共有オブジェクトを作成している場合、リンク編集時には再配置の多くを完了できません。これらの再配置には、オブジェクトをメモリーに読み込むときにだけわかる論理アドレスが必要です。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ます。実行時リンカーは、新しい再配置レコードを処理する必要があります。

再配置のさまざまなタイプの詳細については、254 ページの「再配置型 (プロセッサ固有)」を参照してください。再配置には基本的に 2 つの種類があります。

- 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、`elfdump(1)` を使用して表示できます。次の例では、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。

```
$ elfdump -r libbar.so.1

Relocation Section:  .rel.got:
    type                            offset          section       symbol
  R_SPARC_RELATIVE                  0x10438         .rel.got
  R_SPARC_GLOB_DAT                  0x1043c         .rel.got      foo
```

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、シンボルが参照されていないことからわかります。この再配置では、オブジェクトがメモリーに読み込まれる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可能ファイルと依存関係のいずれかを使用して、このシンボルを検出する必要があります。

再配置シンボルの検索

実行時リンカーには、オブジェクトが必要とするシンボルを実行時に検索する責任があります。一般にユーザーは、動的実行可能ファイルやその依存関係および dlopen(3C) によって取得されたオブジェクトに適用される、デフォルトの検索モデルを理解するようになります。しかし、オブジェクトのシンボル属性や特定の結合要件が原因で、より複雑なシンボル検索が行われることもあります。

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。最初の属性は、要求元オブジェクトのシンボルの検索範囲です。2 つ目の属性は、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です。

これらの属性は、オブジェクトを読み込む際、デフォルトとして適用できます。これらの属性は、dlopen(3C) の特定のモードとしても提供できます。場合によっては、これらの属性をオブジェクトの構築時にオブジェクト内に記録することができます。

オブジェクトは、*world* 検索範囲または *group* 検索範囲、あるいはその両方を定義できます。

ワールド (world)
: オブジェクトは、プロセス内のほかの任意の大域オブジェクト内でシンボルを検索できます。

group
: オブジェクトは、同じグループのオブジェクト内のシンボルを検索できます。dlopen(3C) 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。

オブジェクトは、オブジェクトのエクスポートされたシンボルがグローバルに参照可能か、ローカルに参照可能かを定義できます。

大域 (global)
　オブジェクトのエクスポートされたシンボルは、ワールド検索範囲を持つ任意のオブジェクトから参照できます。

ローカル (local)
　オブジェクトのエクスポートされたシンボルは、同じグループを構成するほかのオブジェクトからのみ参照できます。

もっとも単純な形のシンボル検索については、次の節84 ページの「デフォルトのシンボル検索」で説明します。一般に、シンボル属性はさまざまな形の dlopen(3C) によって利用されます。これらのシナリオについては、105 ページの「シンボルの検索」に記載されています。

動的なオブジェクトで直接結合を行うと、別のシンボル検索モデルが提供されます。このモデルでは、実行時リンカーは、リンク編集時に結合されたオブジェクトからシンボルを直接検索します。86 ページの「直接結合」を参照してください。

デフォルトのシンボル検索

動的実行可能プログラムと、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。動的実行可能ファイルや、それとともに読み込まれた依存関係を対象としたデフォルトのシンボル検索では、各オブジェクトが検索されます。まず動的実行可能プログラムから検索してから、 オブジェクトが読み込まれた順番に依存関係を検索します。

ldd(1) を使用すると、動的実行可能ファイルの依存関係は読み込まれた順にリストされます。たとえば、動的実行可能ファイル prog で、依存関係として libfoo.so.1 と libbar.so.1 が指定されているとします。

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

再配置を実行するためにシンボル bar が必要な場合、実行時リンカーはまず bar を動的実行可能ファイル prog の中で検索します。シンボルが見つからない場合、実行時リンカーは共有オブジェクト /home/me/lib/libfoo.so.1 の中を検索し、最後に共有オブジェクト /home/me/lib/libbar.so.1 の中を検索します。

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。この性能についての詳細は、135 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、86 ページの「直接結合」を参照してください。

デフォルトの再配置処理モデルでは、遅延読み込み環境の遷移も提供します。現在読み込まれているオブジェクト内でシンボルが見つからない場合は、そのシンボル

を特定するために、保留となっている遅延読み込みオブジェクトが処理されます。この読み込みによって、依存関係を完全には定義していないオブジェクトを補います。ただし、これにより遅延読み込みの利点が失われることがあります。

実行時割り込み

デフォルトで、実行時リンカーはまず動的実行可能プログラム内でシンボルを検索したあと、それぞれの依存関係を検索します。このモデルでは、必要なシンボルが最初に現れた時点で検索条件が満たされます。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、ほかのすべてのインスタンスに割り込みます。

シンボル解決がどのように割り込みの影響を受けるかの概要については、45 ページの「単純な解決」で説明しています。シンボルの可視性を変更し、偶発的な割り込みの可能性を低くするメカニズムは、61 ページの「シンボル範囲の縮小」で説明しています。

オブジェクトが割り込み処理として明示的に識別されている場合、割り込みをオブジェクト単位で行えます。環境変数 LD_PRELOAD を使ってオブジェクトを読み込むか、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。実行時リンカーがシンボルを検索する場合、割り込むものとして識別されたオブジェクトはアプリケーションよりもあとで検索されますが、その他の依存関係よりは前に検索されます。

割り込み処理により提供されるすべてのインタフェースの使用が保証されるのは、プロセス再配置が行われる前に割り込み処理が読み込まれる場合のみです。環境変数 LD_PRELOAD を使用して提供される割り込み処理、またはアプリケーションの非遅延読み込み依存関係として確立される割り込み処理は、再配置処理が始まる前に読み込まれます。再配置が始まったあとでプロセスに挿入される割り込み処理は、通常の依存関係に降格されます。割り込み処理を降格できるのは、割り込み処理が遅延読み込みされた場合、または dlopen(3C) を使用した結果として読み込まれた場合です。前者のカテゴリは ldd(1) を使用して検出できます。

```
$ ldd -Lr prog
        libc.so.1 =>     /lib/libc.so.1
        foo.so.2 =>      ./foo.so.2
        libmapmalloc.so.1 =>     /usr/lib/libmapmalloc.so.1
        loading after relocation has started: interposition request \
                (DF_1_INTERPOSE) ignored: /usr/lib/libmapmalloc.so.1
```

注 – 遅延読み込みを行うために依存関係を処理している間に、明示的に定義された割り込み処理をリンカーが検出した場合、その割り込み処理は非遅延読み込み可能依存関係として記録されます。

直接結合

直接結合を使用するオブジェクトは、特定のシンボル参照とその定義を提供する依存関係との間の関係を保持します。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。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。

シンボル参照からシンボル定義への直接結合を確立するには、次のいずれかのメカニズムを使用します。

- `-B direct` オプションを使用する。このオプションは、構築されるオブジェクトとそのすべての依存関係との間に直接結合を確立します。また、このオプションは、構築されるオブジェクト内の任意のシンボル参照とそのシンボル定義との間にも、直接結合を確立します。

  `-B direct` オプションを使用すると、遅延読み込みも有効になります。これは、リンク編集のコマンド行の先頭にオプション `-z lazyload` を追加するのと同じことです。91 ページの「動的依存関係の遅延読み込み」を参照してください。
- `-z direct` オプションを使用する。このオプションは、構築されるオブジェクトから、コマンド行上でこのオプションに続いて指定される依存関係への直接結合を確立します。このオプションは、`-z nodirect` オプションと併用して、依存関係との間の直接結合を使用するかどうかを切り替えることができます。このオプションは、構築されるオブジェクト内の任意のシンボル参照とそのシンボル定義との間に直接結合を確立しません。
- `DIRECT mapfile` キーワードを使用する。このキーワードは、直接結合する個別のシンボルに対応します。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。

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。

注 – 環境変数 `LD_NODIRECT` をヌル以外の値に設定すれば、実行時に直接結合を無効にできます。

デフォルトのシンボル検索モデルでは、1 つのシンボルへのすべての参照を、1 つの定義に結合することができます。直接結合はデフォルトの検索モデルをバイパスするため、暗黙的な割り込みシンボルを迂回します。ただし、割り込み処理として明示的に識別されているオブジェクトは、シンボル定義を供給するオブジェクトの前に検索されます。明示的な割り込み処理には、環境変数 `LD_PRELOAD` を使用して読み

込まれたオブジェクトや、リンカーの -z interpose オプションを使用して作成されたオブジェクトが含まれます。85 ページの「実行時割り込み」を参照してください。

デフォルト手法の代替実装を提供するインタフェースがいくつか存在します。これらのインタフェースは、その実装が、プロセス内におけるその手法の唯一のインスタンスであることを前提とします。この例の 1 つに malloc(3C) ファミリがあります。さまざまな malloc() ファミリの実装が存在しますが、各ファミリは、プロセス内で使用される唯一の実装であると想定されています。このようなファミリ内の特定のインタフェースへの直接結合は避けるべきであり、これは、同じプロセスによってその手法のインスタンスが複数参照される可能性があるためです。たとえば、プロセス内の 1 つの依存関係が libc.so.1 に直接結合し、一方で別の依存関係が libmapmalloc.so.1 に直接結合する可能性があります。malloc() と free() の異なる 2 つの実装を一貫性のない方法で使用すると、エラーが発生しやすくなります。

プロセス内の唯一のインスタンスであると想定されているインタフェースを提供するオブジェクトは、インタフェースへの直接結合を避けるべきです。特定のインタフェースを、どの呼び出し元からも直接結合できないようにラベル付けするには、次のメカニズムのいずれかを使用します。

- -B nodirect オプションを使用する。このオプションは、オブジェクトが提供するすべてのインタフェースへの直接結合を禁止します。
- NODIRECT mapfile キーワードを使用する。このキーワードを使えば、個別のシンボルへの直接結合を禁止できます。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。

非直接ラベル付けを行うと、どのシンボル参照も特定の実装に直接結合できなくなります。参照を解決するためのシンボル検索では、デフォルトのシンボル検索モデルが使用されます。非直接ラベル付けは、Solaris OS に付属するさまざまな malloc() ファミリ実装を構築する目的で採用されました。

注 – NODIRECT mapfile キーワードは、コマンド行オプション -B direct や -z direct と組み合わせることができます。シンボルに NODIRECT を明示的に定義しない場合は、そのシンボルはコマンド行の指令に従います。同様に、DIRECT mapfile キーワードは、コマンド行オプション -B nodirect と組み合わせることができます。シンボルに DIRECT を明示的に定義しない場合は、そのシンボルはコマンド行の指令に従います。

再配置が実行されるとき

再配置は、再配置が実行されるタイミングで 2 つのタイプに区別できます。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、次のように区別されます。

- 即時参照
- 遅延参照

「即時参照」とは、オブジェクトが読み込まれたときにただちに決定しなければならない再配置のことです。この参照は、一般にオブジェクトコードで使用されるデータ項目、関数ポインタ、および位置依存共有オブジェクトからの関数呼び出しに対するものです。即時参照では、再配置された項目が参照されたことを実行時リンカーは認識できません。このため、すべての即時参照は、オブジェクトが読み込まれたら、アプリケーションが制御を獲得または再獲得する前に、再配置が完了する必要があります。

「遅延参照」とは、オブジェクトの実行時に決定できる再配置のことです。通常は、位置独立共有オブジェクトから大域関数への呼び出しか、動的実行可能ファイルから外部関数への呼び出しです。遅延参照を行う動的モジュールをコンパイルおよびリンク編集しているときに、関連付けられた関数呼び出しは、プロシージャーリンクテーブルのエントリへの呼び出しに変換されます。これらのエントリは、`.plt` セクションを構成します。プロシージャーリンクテーブルの各エントリは、関連付けられた再配置を伴う遅延参照になります。

プロシージャーリンクテーブルの特定のエントリに対する最初の呼び出しの実行中に、制御が実行時リンカーに渡されます。実行時リンカーは、関連付けられたオブジェクト内で必要なシンボルを検索し、エントリ情報を書き換えます。その後のプロシージャーリンクテーブルのエントリへの呼び出しは、直接関数に対して行われます。遅延参照では、関数が最初に呼び出されるまで、再配置を遅延させることができます。この処理は、「遅延」結合と呼ばれることがあります。

実行時リンカーのデフォルトモードは、プロシージャーリンクテーブルの再配置が行われるたびに遅延結合を実行する、というものです。デフォルトモードを無効にするには、環境変数 `LD_BIND_NOW` にヌル以外の任意の値を設定します。この環境変数の設定により、実行時リンカーは、オブジェクトが読み込まれた時点で、即時参照と遅延参照を両方とも再配置します。これらの再配置は、アプリケーションが制御を獲得または再獲得するまでの間に行われます。たとえば、環境変数を次のように設定して、ファイル `prog` とその依存関係内のすべての再配置が行われるとします。これらの再配置は、制御がアプリケーションに移る前に行われます。

```
$ LD_BIND_NOW=1 prog
```

オブジェクトへのアクセスは、`RTLD_NOW` として定義されたモードを指定して dlopen(3C) を使用することによっても行えます。リンカーの `-z now` オプションを使用してオブジェクトを構築すれば、オブジェクトが読み込まれたときに再配置処理を完了させる必要があることを示すことができます。この再配置要件は、実行時に指定したオブジェクトの依存先すべてに波及します。

注 – 前述の即時参照と遅延参照の例は、標準的なものです。ただし、プロシージャーリンクテーブルのエントリの作成は、リンク編集の入力として使用する再配置可能オブジェクトファイルが提供する再配置情報によって、最終的に制御されます。R_SPARC_WPLT30 や R_386_PLT32 などの再配置レコードには、プロシージャーリンクテーブルのエントリの作成が指定されています。こうした再配置は、位置独立のコードで共通です。

ただし、通常、動的実行可能ファイルは位置に依存するコードから作成されるため、プロシージャーリンクテーブルのエントリが必要であることを示さない場合があります。動的実行可能ファイルの位置は固定されているため、参照が外部関数定義に結合された時点で、リンカーはプロシージャーのリンクテーブルを作成できます。元の再配置レコードに関係なく、このプロシージャーリンクテーブルのエントリを作成できます。

再配置エラー

もっとも一般的な再配置エラーは、シンボルを検出できないときに発生します。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。次の例では、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。

```
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libc.so.1 =>     /lib/libc.so.1
        libbar.so.1 =>   ./libbar.so.1
        libm.so.2 =>     /lib/libm.so.2
$ prog
ld.so.1: prog: fatal: relocation error: file ./libfoo.so.1: \
    symbol bar: referenced symbol not found
$
```

動的実行可能プログラムのリンク編集中に、この種の潜在的な再配置エラーは、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。例については、49 ページの「実行可能ファイルの作成」を参照してください。ただし、実行時再配置エラーが発生するのは、実行時に配置される依存関係が、リンク編集の一部として参照される元の依存関係と互換性がない場合です。上記の例では、bar のシンボル定義を含む共有オブジェクト libbar.so.1 のバージョンに対して prog が構築されています。

リンク編集時に -z nodefs オプションを使用すると、オブジェクトの実行時再配置要件の検査が抑制されます。この抑制は、実行時再配置エラーになる可能性があります。

即時参照として使用されたシンボルが検出できないために再配置エラーが発生した場合、そのエラー状態は、プロセスの初期設定中、ただちに発生します。遅延結合のデフォルトモードにより、遅延参照として使用されるシンボルを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ます。

ldd(1) 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、すべての即時参照の再配置処理が実行されます。参照を解決できない場合には、診断メッセージが作成されます。上記の例から -d オプションを使用すると、次のエラー診断が作成されます。

```
$ ldd -d prog
        libfoo.so.1 =>   ./libfoo.so.1
        libc.so.1 =>     /lib/libc.so.1
        libbar.so.1 =>   ./libbar.so.1
        libm.so.2 =>     /lib/libm.so.2
        symbol not found: bar           (./libfoo.so.1)
```

ldd(1) に -r オプションを指定すると、すべての即時参照と遅延参照の再配置が処理されます。また、このどちらかの再配置が解決できない場合には、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

実行時リンカーでは、環境変数 LD_PRELOAD を使用することにより、プロセスの初期設定中に新しいオブジェクトを取り込めるという、一歩進んだ柔軟性も提供しています。この環境変数は、特定の共有オブジェクトまたは再配置可能オブジェクトのファイル名に初期設定することも、複数のファイル名を空白で区切った文字列に初期設定することもできます。これらのオブジェクトは、動的実行可能プログラムの後で、依存関係よりも前に読み込まれます。これらのオブジェクトには、「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。

次の例では、動的実行可能プログラム prog が読み込まれ、そのあとに共有オブジェクト newstuff.so.1 が続きます。続いて、prog 内で定義された依存関係が読み込まれます。

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

これらのオブジェクトが処理される順序は、ldd(1) を使用して表示できます。

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

次の例では、事前読み込みは少し複雑で時間がかかります。

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

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` の事前読み込みと同じ方法で、動的実行可能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序は、`ldd(1)` を使用して表示できます。

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

動的実行可能ファイルのあとにオブジェクトを挿入するこれらのメカニズムにより、割り込み機能が提供されます。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存在する関数の、新しい実装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、このオブジェクトは元のオブジェクトに割り込みます。そして、元の機能は、事前読み込みされた新しいバージョンによって完全に隠されてしまいます。

このほか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。新しいシンボルが元のシンボルに割り込むことで、新しい関数はいくつかの追加処理を実行できます。新しい関数は元の関数を呼び出すこともできます。このメカニズムでは通常、`dlsym(3C)` と特別なハンドル `RTLD_NEXT` を使って元のシンボルのアドレスを取得します。

動的依存関係の遅延読み込み

メモリーに動的オブジェクトが読み込まれる際、その動的オブジェクトに追加の依存関係がないか検査されます。デフォルトでは、存在する依存関係がただちに読み込まれます。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続けられます。最終的に、再配置で指定されたオブジェクト間のデータ参照すべてが解決されます。この処理は、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、行われます。

遅延読み込みモデルでは、遅延読み込み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。関数呼び出しの遅延結合を利用して、関数が最初に参照されるまで、依存関係の読み込みを延期することができます。結果として、参照されないオブジェクトは読み込まれません。

再配置参照は、即時か遅延です。即時参照はオブジェクトが初期化された時に解決される必要があるため、この参照を満たすすべての依存関係はすぐに読み込まれる必要があります。そのため、そういった依存関係を遅延読み込み可能として示すことは、あまり効果がありません。詳細は、87 ページの「再配置が実行されるとき」を参照してください。動的オブジェクト間の即時参照は、概してあまり推奨されません。

遅延読み込みは、デバッグライブラリ `liblddbg` への参照のためにリンカーが使用します。デバッギン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。このライブラリを遅延読み込みできるように指定することにより、ライブラリの処理コストをデバッギング出力を必要とする読み込みに使うことができます。

遅延読み込み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。このモデルは、`dlsym()` 参照の数が少ない場合に最適です。またこのモデルは、リンク編集時に依存関係の名前あるいは位置がわからない場合にも適しています。名前や位置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、依存関係を遅延読み込みに指定する方が簡単です。

特定のオブジェクトを遅延読み込み、通常読み込みとして指定するには、リンカーオプション `-z lazyload`、`-z nolazyload` をそれぞれ使用します。これらのオプションは、リンク編集コマンド行の位置に依存します。このオプションよりあとに指定される依存関係には、このオプションで指定されている読み込み属性が適用されます。デフォルトでは、`-z nolazyload` オプションが有効です。

次の単純なプログラムでは、`libdebug.so.1` に対する依存関係が指定されています。動的セクション (`.dynamic`) では、`libdebug.so.1` に対して遅延読み込みが指定されています。シンボル情報セクション (`.SUNW_syminfo`) では、`libdebug.so.1` の読み込みをトリガーするシンボル参照が指定されています。

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

Dynamic Section:  .dynamic
     index  tag           value
       [0]  POSFLAG_1     0x1          [ LAZY ]
       [1]  NEEDED        0x123        libdebug.so.1
       [2]  NEEDED        0x131        libelf.so.1
       [3]  NEEDED        0x13d        libc.so.1
       [4]  RUNPATH       0x147        $ORIGIN
       ...
$ elfdump -y prog

Syminfo section: .SUNW_syminfo
     index  flgs        bound to        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 が遅延読み込みされることを示しています。libelf.so.1 は前に LAZY フラグがないため、このライブラリはプログラムの初期始動時に読み込まれます。

注 – libc.so.1 には、ファイルが遅延読み込みされてはならないという特別なシステム要件があります。libc.so.1 が処理される時点で -z lazyload が有効である場合、フラグは実際には無視されます。

遅延読み込みを使用するには、アプリケーションで使用されるオブジェクト全体に渡り依存関係と「実行パス」を正確に宣言しなければならない場合があります。たとえば、libX.so 内のシンボルを参照する 2 つのオブジェクト libA.so と libB.so があるとします。libA.so は libX.so を依存関係として宣言しますが、libB.so は宣言しません。通常、libA.so と libB.so が併用される場合、libB.so は libX.so を参照できます。これは、libA.so によってこの依存関係が利用可能になっているためです。しかし、libX.so が遅延読み込みされるように libA.so で宣言した場合、libB.so がこの依存関係を参照するときに libX.so を読み込めない可能性があります。libB.so で libX.so を依存関係として宣言していても、その依存関係の特定に必要な「実行パス」を指定しなかった場合には、同様のエラーが発生する可能性があります。

遅延読み込みに関わらず、動的オブジェクトは、すべての依存関係と依存関係の特定方法を宣言しなければなりません。遅延読み込みでは、この依存情報がより重要な意味合いを持ちます。

注 – 環境変数 LD_NOLAZYLOAD をヌル以外の値に設定すれば、実行時に遅延読み込みを無効にできます。

---

dlopen() の代替手段の提供

遅延読み込みは、dlopen(3C) と dlsym(3C) を使用する代わりになります。101 ページの「実行時リンクのプログラミングインタフェース」を参照してください。たとえば、libfoo.so.1 の次のコードは、オブジェクトが読み込まれることを確認し、そのオブジェクトのインタフェースを呼び出します。

```
void foo()
{
    void * handle;

    if ((handle = dlopen("libbar.so.1", RTLD_LAZY)) != NULL) {
        int (* fptr)();
```

```
        if ((fptr = (int (*)())dlsym(handle, "bar1")) != NULL)
            (*fptr)(arg1);
        if ((fptr = (int (*)())dlsym(handle, "bar2")) != NULL)
            (*fptr)(arg2);
        ....
}
```

このコードは、必要なインタフェースを提供するオブジェクトが次の条件を満たす場合に、単純化できます。

- オブジェクトが、リンク編集時の依存関係として構築できる。
- オブジェクトが、常に利用できる。

遅延読み込みを活用することで、`libbar.so.1` の同じ据え置き読み込みをアーカイブできます。この場合、関数 `bar1()` を参照すると、関連依存関係が遅延読み込みされます。さらに、標準関数呼び出しを使用すると、コンパイラまたは `lint(1)` 確認が提供されます。

```
void foo()
{
    bar1(arg1);
    bar2(arg2);
    ....
}
$ cc -G -o libfoo.so.1 foo.c -L. -zlazyload -zdefs -lbar -R'$ORIGIN'
```

ただし、必要なインタフェースを提供するオブジェクトが常に利用できるとは限らない場合、このモデルは失敗します。この場合、依存関係の名前なしで依存関係の有無をテストする機能が必要です。関数参照を満足する依存関係を使用できるかどうかをテストする手段が必要になります。

`RTLD_PROBE` ハンドルのある `dlsym(3C)` は、依存関係の有無および読み込みの確認に使用できます。たとえば、`bar1()` への参照は、リンク編集時に確立された遅延依存関係を利用できるかどうかを確認します。このテストは、`dlopen(3C)` が使用されたのと同じ方法で、依存関係が提供する関数への参照を制御するために使用できます。

```
void foo()
{
    if (dlsym(RTLD_PROBE, "bar1")) {
        bar1(arg1);
        bar2(arg2);
        ....
}
```

このテクニックは、標準関数呼び出しを使用するとともに、記録された依存関係の安全な据え置き読み込みを実現します。

注 – この特別なハンドル RTLD_DEFAULT は、RTLD_PROBE の使用時と同じようなメカニズムを提供します。ただし、RTLD_DEFAULT を使用すると、存在しないシンボルを特定しようとして、処理中の遅延読み込みオブジェクトが保留されることがあります。この読み込みによって、依存関係を完全には定義していないオブジェクトを補います。ただし、これにより遅延読み込みの利点が失われることがあります。

遅延読み込みを使用するオブジェクトの作成には、-z defs オプションの使用をお勧めします。

初期設定および終了ルーチン

動的オブジェクトは、実行時の初期設定と終了処理のためのコードを提供することができます。動的オブジェクトの初期設定コードは、処理中に動的オブジェクトが読み込まれるたびに、1 回ずつ実行されます。動的オブジェクトの終了コードは、動的オブジェクトが処理から読み取り解除されるか、または処理の終了のたびに 1 回ずつ実行されます。

実行時リンカーは、制御をアプリケーションに移す前に、アプリケーション内および読み込まれたすべての依存関係内で見つかったすべての初期設定セクションを処理します。プロセス実行中に新しい動的オブジェクトが読み込まれた場合、その初期設定セクションはオブジェクトの読み込みの一部として処理されます。初期設定セクションである .preinitarray、.initarray、および .init は、動的オブジェクトの構築時にリンカーによって作成されます。

実行時リンカーは、.preinitarray セクションと .initarray セクションにアドレスが指定されている関数を実行します。これらの関数は、配列内でアドレスが出現する順序で実行されます。実行時リンカーは、.init セクションを独立した関数として実行します。1 つのオブジェクトに .init セクションと .initarray セクションの両方が含まれている場合は、そのオブジェクトに関しては、.initarray セクションによって定義されている関数よりも前に .init セクションが処理されます。

動的実行可能ファイルは、.preinitarray 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、かつほか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 .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、アプリケーションから呼び出されます。動的実行可能ファイルの .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あとで、最後に呼び出されます。

動的オブジェクトは、終了セクションも提供できます。終了セクションである .finiarray および .fini 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ます。

終了セクションはすべて、atexit(3C) に転送されます。これらの終了ルーチンは、プロセスが exit(2) を呼び出したときに呼び出されます。また終了セクションは、dlclose(3C) を持つ実行プロセスからオブジェクトが除去されたときにも呼び出されます。

実行時リンカーは、.finiarray セクションにアドレスが指定されている関数を実行します。これらの関数は、配列内でアドレスが出現する順序とは逆に実行されます。実行時リンカーは、.fini セクションを独立した関数として実行します。オブジェクトに .fini セクションと .finiarray セクションの両方が含まれている場合は、そのオブジェクトに関しては、.fini セクションによって定義されている関数よりも前に .finiarray セクションが処理されます。

注 – 動的実行可能プログラム内の .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から呼び出されます。動的実行可能プログラムの .fini セクションは、そのすべて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。

リンカーによる初期設定セクションと終了セクションの作成についての詳細は、41 ページの「初期設定および終了セクション」を参照してください。

初期設定と終了の順序

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。この処理は、初期設定セクションと終了セクションの導入以来、大きく発展してきました。この処理は、最新の言語と現在のプログラミング手法の期待を実現しようとするものです。しかし、ユーザーの期待にこたえるのが難しい状況もあります。これらの状況を理解し、初期設定および終了コードの内容を制限することで、柔軟で予測可能な実行時動作が得られます。

初期設定セクションの目的は、同一オブジェクト内のほかのコードが参照される前に小さなコードを実行することです。終了セクションの目的は、オブジェクトの実行完了後に小さなコードを実行することです。自己完結型の初期設定セクションや終了セクションは、これらの要件を容易に満たすことができます。

しかしながら、初期設定セクションは通常それよりも複雑であり、ほかのオブジェクトが提供する外部のインタフェースを参照します。したがって、ほかのオブジェクトからの参照が発生する前にオブジェクトの初期設定セクションを実行する必要がある場合には、依存関係が発生することになります。アプリケーションが大

規模な依存関係の階層を確立する可能性があります。さらに、依存関係がその階層内で循環を形成する可能性もあります。こうした状況が、追加のオブジェクトを読み込んだりすでに読み込まれたオブジェクトの再配置モードを変更したりする初期設定セクションによって、さらに複雑になる可能性があります。これらの問題を解決するためにさまざまなソート手法や実行手法が開発されてきましたが、それらもすべて、これらのセクションの本来の目的を達成するためでした。

Solaris 2.6 より前のリリースでは、依存関係の初期設定ルーチンが呼び出される順序は、読み込まれた順序の「逆」、つまり ldd(1) を使用して表示される依存関係の順序とは逆でした。同様に、依存関係の終了ルーチンが呼び出される順序は、読み込まれた順序と同じでした。しかし、依存関係の階層が複雑化するにつれ、この単純な順序付け手法は適切とは言えなくなりました。

Solaris 2.6 リリースでは、実行時リンカーは、読み込まれたオブジェクトを位相的にソートしてリストを作成するようになりました。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、示された依存関係の外部で発生したシンボル結合から構成されます。

注意 – Solaris 8 10/00 より前のリリースでは、環境変数 LD_BREADTH をヌル以外の値に設定できていました。この設定により、実行時リンカーで初期設定セクションと終了セクションを強制的に「Solaris 2.6 リリースより前」の順序で実行することができました。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。

初期設定セクションは、依存関係の位相的な順序とは逆に実行されます。循環性のある依存関係が検出された場合、循環の原因であるオブジェクトは、位相的にソートされません。循環性のある依存関係の初期設定セクションは、読み込まれた順序の逆に実行されます。同様に、終了セクションは、依存関係の位相的な順序で呼び出されます。循環性のある依存関係の終了セクションは、読み込まれた順序で実行されます。

オブジェクトの依存関係の初期設定順序に関する静的解析を取得するには、ldd(1) で -i オプションを指定します。たとえば、次の動的実行可能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。

```
$ elfdump -d B.so.1 | grep NEEDED
    [1]     NEEDED      0xa9    C.so.1
$ elfdump -d C.so.1 | grep NEEDED
    [1]     NEEDED      0xc4    B.so.1
$ elfdump -d main | grep NEEDED
    [1]     NEEDED      0xd6    A.so.1
    [2]     NEEDED      0xc8    B.so.1
    [3]     NEEDED      0xe4    libc.so.1
```

```
$ ldd -i main
        A.so.1 =>        ./A.so.1
        B.so.1 =>        ./B.so.1
        libc.so.1 =>     /lib/libc.so.1
        C.so.1 =>        ./C.so.1
        libm.so.2 =>     /lib/libm.so.2

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

   init object=/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
```

この解析結果は単純に、明示的な依存関係を位相的にソートすることによって得られたものです。ただし、自身が必要とする依存関係を定義していないオブジェクトも頻繁に作成されます。このため、シンボル結合も依存関係解析の一部として組み込まれます。明示的な依存関係を持つシンボル結合を組み込むと、より正確な依存関係の構築に役立ちます。初期設定順序のより正確な静的解析を取得するには、ldd(1) で -i オプションと -d オプションを指定してください。

オブジェクトの読み込みに使用されるもっとも一般的なモデルは遅延結合です。このモデルの場合、初期設定処理の前に処理されるのは「即時参照」シンボル結合だけです。「遅延参照」からのシンボル結合は保留されている場合があります。これらの結合は、それまでに確立された依存関係を拡張できます。すべてのシンボル結合が組み込まれた初期設定順序の静的解析を取得するには、ldd(1) で -i オプションと -r オプションを指定してください。

実際には、ほとんどのアプリケーションが遅延結合を使用します。したがって、初期設定順序を計算する前に実行された依存関係解析は、ldd -id を使用した静的解析に従います。ただし、この依存関係解析は不完全である可能性があり、また循環依存関係が存在する可能性もあるため、実行時リンカーでは動的初期設定も使用できるようになっています。

動的初期設定は、あるオブジェクトの初期設定セクションを、その同じオブジェクト内の関数が呼び出される前に実行しようとします。実行時リンカーは、遅延シンボル結合の際に、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ます。呼び出されていなければ、実行時リンカーは、シンボル結合手順から戻る前にその初期設定セクションを実行します。

動的な初期設定は、ldd(1) では確認できません。しかし、LD_DEBUG 環境変数を設定してトークン init を含めることにより、実行時に初期設定呼び出しの正確な手順を確認できます。114 ページの「デバッギングライブラリ」を参照してください。デ

バッグ用のトークン detail を追加すると、広範な初期設定情報や終了情報を取得できます。この情報には、依存関係の一覧、位相的な処理、循環依存関係の特定などが含まれます。

動的初期設定を使用できるのは、遅延参照を処理する場合だけです。この動的な初期設定を迂回するには、次の手段があります。

- 環境変数 LD_BIND_NOW の使用。
- -z now オプションを使用して構築されたオブジェクト。
- RTLD_NOW モードを使用して dlopen(3C) によって読み込まれたオブジェクト。

これまでに説明した初期設定手法だけでは、いくつかの動的な活動に対処できない可能性があります。初期設定セクションが、dlopen(3C) を使って明示的に、あるいは遅延読み込みやフィルタ使用によって暗黙的に、追加のオブジェクトを読み込む可能性があります。また、初期設定セクションが既存オブジェクトの再配置を促進する可能性もあります。遅延結合を採用するために読み込まれたオブジェクトがモード RTLD_NOW を指定した dlopen(3C) を使って参照された場合、その同じオブジェクトの結合が解決されます。この再配置促進によって結果的に、関数呼び出しの動的解決に使用可能な動的初期設定機能が抑制されます。

新しいオブジェクトが読み込まれるたびに、あるいは既存オブジェクトの再配置が促進されるたびに、それらのオブジェクトの位相的なソートが起動されます。その結果、元の初期設定の実行が中断されるとともに、新しい初期設定要件が確立され、関連する初期設定セクションが実行されます。このモデルの意図は、新しく参照されたオブジェクトを適切に初期設定し、それを元の初期設定セクションが確実に使用できるようにすることです。ところが、この並行処理が不要な再帰の原因となる可能性があります。

実行時リンカーは、遅延結合を採用したオブジェクトを処理する際に、特定レベルの再帰を検出できます。この再帰を表示するには、LD_DEBUG=init と設定します。たとえば、foo.so.1 の初期設定セクションを実行すると、別のオブジェクトが呼び出される可能性があります。そして、そのオブジェクトが foo.so.1 内のいずれかのインタフェースを参照していた場合、循環が形成されます。実行時リンカーは、遅延関数参照を foo.so.1 に結合する過程で、この再帰を検出できます。

```
$ LD_DEBUG=init prog
00905: .......
00905: warning: calling foo.so.1 whose init has not completed
00905: .......
```

実行時リンカーは、すでに再配置された参照を通じて発生した再帰を検出することはできません。

再帰は、多くのコストと問題を発生させる可能性があります。再帰が発生しないよう、初期設定セクションによって起動される可能性のある外部参照や動的読み込み活動の数を減らしてください。

初期設定処理は、dlopen(3C) を指定して実行しているプロセスに追加された、すべてのオブジェクトに対して繰り返されます。また、dlclose(3C) に対する呼び出しの結果としてプロセスから読み込み解除されるすべてのオブジェクトに対して、終了処理も行われます。

これまでの節では、ユーザーの期待に応えようとする方法で、初期設定セクションと終了セクションを実行するために使用されるさまざまな手法を説明してきました。しかし、依存関係同士の初期設定と終了の関係を単純化するためには、コーディングスタイルとリンク編集の助けも必要です。この単純化は、初期設定と予測可能な終了処理を助け、予期しない依存関係順序付けによる副作用の影響を受けにくくします。

初期設定セクションと終了セクションの内容は最小限に抑えてください。実行時にオブジェクトを初期化することによって、大域的なコンストラクタを避けてください。ほかの依存関係に対する初期設定および終了コードの依存を減らしてください。すべての動的オブジェクトについて依存関係の要件を定義してください。51 ページの「共有オブジェクト出力ファイルの生成」を参照不要な依存関係を定義しないでください。35 ページの「共有オブジェクトの処理」を参照。依存関係の循環を避けてください。初期設定または終了の順序に頼らないでください。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変更される場合があるからです。126 ページの「依存関係の順序」を参照してください。

セキュリティー

セキュリティー保護されたプロセスには、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、issetugid(2) システム呼び出しがプロセスに対して true を返した場合、そのプロセスを安全と分類します。

32 ビットオブジェクトの場合、実行時リンカーが認識しているデフォルトのトラストディレクトリは、/lib/secure と /usr/lib/secure です。64 ビットオブジェクトの場合、実行時リンカーが認識しているデフォルトのトラストディレクトリは、/lib/secure/64 と /usr/lib/secure/64 です。ユーティリティー crle(1) を使用すれば、セキュリティー保護された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。

あるセキュリティー保護されたプロセスに対して LD_LIBRARY_PATH ファミリ環境変数が有効になっている場合、実行時リンカーの検索規則を拡張するために使用されるのは、この変数に指定されたトラストディレクトリだけです。79 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。

セキュリティー保護されたプロセスでは、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」の指定が使用されます。ただし、「実行パス」はフルパス名である、つまりパス名は「/」から始まる必要があります。

セキュリティー保護されたプロセスでは、$ORIGIN 文字列の拡張は、その文字列がトラストディレクトリに拡張されるときにかぎり許可されます。393 ページの「セキュリティー」を参照してください。ただし、$ORIGIN を展開することですでに依存関係を提供したディレクトリに一致する場合、そのディレクトリは暗黙にセキュリティー保護されます。このディレクトリは、追加の依存関係を提供するために使用できます。

セキュリティー保護されたプロセスでは、LD_CONFIG は無視されます。セキュリティー保護されたプロセスは、デフォルト構成ファイルが存在する場合、この構成ファイルを使用します。crle(1) のマニュアルページを参照してください。

セキュリティー保護されたプロセスでは、LD_SIGNAL は無視されます。

セキュリティー保護されたプロセスで追加オブジェクトを読み込むには、LD_PRELOAD、LD_AUDIT のいずれかの環境変数を使用します。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ます。単純ファイル名は、既知のトラストディレクトリにのみ解決されることになります。

セキュリティー保護されたプロセスでは、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、セキュリティー保護されたプロセスの開発者は、これらの依存関係の 1 つ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べきです。

セキュリティー保護されたプロセスを作成する場合には、依存関係の表示や、dlopen(3C) パス名の構築に、相対パス名は使用しないでください。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

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は、プロセスの初期設定中に実行時リンカーによって処理されます。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。アプリケーションは、アプリケーションの標準的な依存関係の処理に使用される、実行時リンカーの同じサービスを実際に使用します。

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ます。アプリケーションの特定の実行中に、オブジェクトにより提供されるサービスが必要でない場合、オブジェクトは要求されません。このような状態は、ヘルプやデバッグ情報を提供するオブジェクトに対して発生する可能性があります。
- アプリケーションは、ネットワーキ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ます。
- 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ます。

アプリケーションは、次の典型的な手順を使用して、追加の共有オブジェクトにアクセスできます。

- 共有オブジェクトは、dlopen(3C) 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、追加されます。この共有オブジェクトの依存関係は、この時点で配置されて追加されます。
- 追加された共有オブジェクトとその依存関係は、再配置されます。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ます。
- 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(3C) を使用して配置します。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します。
- オブジェクトによってアプリケーションが終了したあとで、dlclose(3C) を使用してアドレススペースを解放できます。解放されたオブジェクト内に存在する終了セクションは、すべてこの時点で呼び出されます。
- 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(3C) を使用して表示できます。

実行時リンカーのサービスは、ヘッダーファイル `dlfcn.h` 内に定義されており、共有オブジェクト `libc.so.1` 経由でアプリケーションから使用できます。次の例では、ファイル `main.c` は、ルーチンのいずれの dlopen(3C) ファミリでも参照でき、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。

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

注 – Solaris OS の以前のリリースでは、動的リンクインタフェースは、共有オブジェクト `libdl.so.1` によって使用可能にされていました。`libdl.so.1` は既存の依存関係をサポートするために今も使用できます。ただし、`libdl.so.1` から提供されていた動的リンクインタフェースは、現在は `libc.so.1` から使用できるようになりました。このため、`-ldl` によるリンクは必要なくなりました。

追加オブジェクトの読み込み

追加オブジェクトは、dlopen(3C) 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(3C) を使用することによってシンボルを配置できます。

パス名が、単純ファイル名で指定されている (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。「/」が組み込まれたパス名は、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。79 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。たとえば、ファイル main.c には、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。

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

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

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

実行時リンカーは、共有オブジェクト foo.so.1 を検索するために、プロセスの初期設定時に存在する任意の LD_LIBRARY_PATH 定義を使用します。続いて、実行時リンカーは prog のリンク編集時に指定された「実行パス」を使用します。最後に、実行時リンカーは 32 ビットオブジェクトの場合は /lib と /usr/lib、64 ビットオブジェクトの場合は /lib/64 と /usr/lib/64 のデフォルト位置を使用します。

パス名が次のように指定されているとします。

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

実行時リンカーは、プロセスの現在のカレントディレクトリ内でこのファイルだけを検索します。

注 – dlopen(3C) 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。バージョン管理の詳細については、173 ページの「バージョン管理ファイル名の管理」を参照してください。

必要なオブジェクトを配置できない場合は、dlopen(3C) によって NULL ハンドルが戻されます。この場合、dlerror(3C) 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
$ prog
dlopen: ld.so.1: prog: fatal: foo.so.1: open failed: No such \
    file or directory
```

dlopen(3C) によって追加されたオブジェクトに、ほか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(3C) によって指定されたオブジェクト、またはその依存関係が、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。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、前述の例で main.c に次の dlopen() 呼び出しが含まれている場合があります。

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

この dlopen(3C) から返されたハンドルを使用すれば、アプリケーション自身の内部、プロセスの初期設定中に読み込まれたすべての依存関係内、またはプロセスのアドレススペースに追加されたすべてのオブジェクト内で、シンボル検索を行えます。それには、dlopen(3C) で RTLD_GLOBAL フラグを指定します。

再配置処理

実行時リンカーは、オブジェクトを検出して読み込んだあと、各オブジェクトを処理し、必要な再配置を実行します。また、dlopen(3C) 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があります。

単純なアプリケーションの場合には、このプロセスはそれほど重要な意味を持ちません。しかし、多くのオブジェクトを含む多数の dlopen(3C) 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、このプロセスは非常に重要です。

再配置は、その実行タイミングに基づいて分類できます。実行時リンカーのデフォルトの動作では、初期設定時に即時参照の再配置がすべて処理され、プロセスの実行時に遅延参照がすべて処理されます。後者の処理は通常、遅延結合と呼ばれます。

モードが RTLD_LAZY として定義されている時に dlopen(3C) を使って追加されたすべてのオブジェクトに対して、これと同じメカニズムが適用されます。代替手段は、オブジェクトが追加されるときにただちに実行されるオブジェクトのすべての再配置を要求することです。RTLD_NOW のモードを使用するか、またはリンカーの -z now オプションを使用して構築した際のオブジェクトにこの要件を記録できます。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。

シンボルの検索

dlopen(3C) 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、プロセスを作成する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。直接結合がない場合は、dlopen() によって入手されたオブジェクトには、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、プロセスを作成するオブジェクトの属性と結合される dlopen() のモードは、代わりのシンボル検索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。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。86 ページの「直接結合」を参照してください。

デフォルトにより、dlopen(3C) 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。105 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。109 ページの「大域オブジェクトの定義」、109 ページの「グループの分離」、および 109 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モデルの展開に dlopen(3C) 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

基本的な dlopen(3C) 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、最初に動的実行可能ファイル内でシンボルを検索します。次に実行時リンカーは、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。シンボルが見つからない場合、実行時リンカーは検索を続けます。実行時リンカーは次に dlopen(3C) によって取得されたオブジェクトおよびそのオブジェクトと依存関係にあるオブジェクトを検索します。

デフォルトのシンボル検索モデルは、遅延読み込み環境の遷移も行います。現在読み込まれているオブジェクト内でシンボルが見つからない場合は、そのシンボルを特定するために、保留となっている遅延読み込みオブジェクトが処理されます。この読み込みによって、依存関係を完全には定義していないオブジェクトを補います。ただし、これにより遅延読み込みの利点が失われることがあります。

次の例の動的実行可能プログラム prog と共有オブジェクト B.so.1 には、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(3C) 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 の順に検索されます。このような単純なケースでは、dlopen(3C) 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。

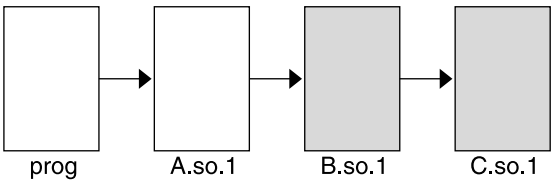


図 3–1　単一の dlopen() 要求

dlopen(3C) 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。動的実行可能ファイル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、各オブジェクトは局所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、ほかのグループメンバー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アプリケーションの最初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(3C) 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、順に prog と A.so.1 で実施されます。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。ワールドシンボルの検索範囲が、動的実行可能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ます。この検索範囲では、局所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、オブジェクト間の関係を保持します。これらの関係は、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います。指定された dlopen(3C) に関連したオブジェクトを固有のグループに割り当てることにより、同じ dlopen(3C) と関連したオブジェクトだけが、グループ内のシンボルと、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(3C) 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も dlopen(3C) を使って読み込んだとします。次の図は、オブジェクト間のシンボル検索の関係を示しています。

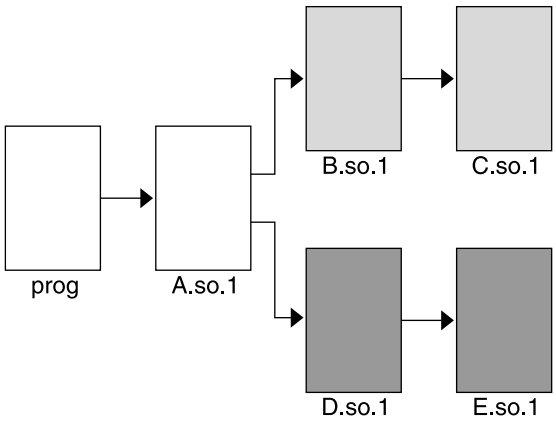


図 3–2 複数の dlopen() 要求

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 にこのシンボルを必要とする再配置が組み込まれているとします。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(3C) への複数の呼び出しにより入手されたオブジェクトのもっと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(3C) が実施された順序は、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(3C) 呼び出しが実行された順序の影響を受けます。

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(3C) 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、Z.so.1 は 2 つの dlopen(3C) 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。この依存関係を次の図に示します。

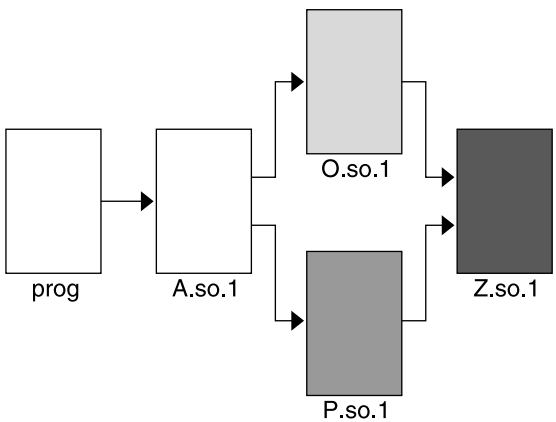


図 3–3　共通依存関係を伴う複数の dlopen() 要求

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(3C)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期することはできません。それは、この結び付きが dlopen(3C) 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 で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(3C) 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(3C) で取得されるオブジェクトにデフォルトで割り当てられる「ローカル」シンボル可視性を「大域」に拡張するには、モード引数に RTLD_GLOBAL フラグを指定します。このモードでは、dlopen(3C) 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が指定されたほか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(3C) によって入手されたオブジェクトは、dlopen() (値 0 のパス名を指定) を使用したシンボル検索にも使用できます。

注 – 局所シンボルの可視性を持つグループのメンバーが、ほか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(3C) で取得されるオブジェクトにデフォルトで割り当てられる「ワールド」シンボル検索範囲を「グループ」に縮小するには、モード引数に RTLD_GROUP フラグを指定します。このモードでは、dlopen(3C) 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。

リンカーの -B group オプションを使用して構築したオブジェクトには、グループのシンボル検索範囲を割り当てることができます。

注 – グループ検索機能を持つグループのメンバーが、ワールド検索機能を必要とするほか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連結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

最初のオブジェクトが dlopen(3C) によって入手され、dlopen() を使用して 2 番目のオブジェクトを開いた場合、両方のオブジェクトは同じ固有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。
- 2 番目のオブジェクトに対する dlopen(3C) で 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。

多くのオブジェクトが dlopen(3C) を使って 2 番目のオブジェクトを開くことができ、かつそれらの初期オブジェクトが 2 番目のオブジェクトの再配置を満たすために同じシンボルをエクスポートしなければならない場合、その 2 番目のオブジェクトに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(3C) 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。

これら 2 つの手法の間には、小さな相違点が 1 つ存在しています。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(3C) 依存関係ツリーの一部になるため、dlsym(3C) 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、dlsym(3C) 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。

「大域」シンボル可視性を持つ初期オブジェクトが dlopen(3C) を使って 2 番目のオブジェクトを取得する場合、RTLD_PARENT モードは冗長かつ無害になります。このような状態は、dlopen(3C) 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(3C) 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ハンドルが dlopen(3C) から返される。このハンドルを使用すると、指定したオブジェクトとその依存ツリーを構成するオブジェクト群からシンボルを入手できます。RTLD_FIRST モードを使用して戻されたハンドルの場合は、指定したオブジェクトだけからシンボルを入手できます。
- 値が 0 のパス名のハンドルが dlopen(3C) から返される。このハンドルを使用すると、関連づけられたリンクマップの開始オブジェクトと、その依存ツリーを構成するオブジェクト群から、シンボルを入手できます。通常、開始オブジェクトは動的実行可能ファイルです。このハンドルを使用すると、関連づけられたリンクマップ上の、dlopen(3C) で RTLD_GLOBAL モードを使用して読み込まれたすべての

オブジェクトからも、シンボルを入手できます。RTLD_FIRST モードを使用して戻されたハンドルの場合は、関連づけられたリンクマップ上の開始オブジェクトだけからシンボルを入手できます。

- 特別なハンドル RTLD_DEFAULT と RTLD_PROBE を使えば、関連付けられたリンクマップの開始元オブジェクトやその依存関係ツリーを定義するオブジェクトから、シンボルを取得できる。このハンドルを使用すると、呼び出し元と同じグループに属する、dlopen(3C) を使用して読み込まれたオブジェクトからも、シンボルを入手できます。RTLD_DEFAULT または RTLD_PROBE の使用は、呼び出し側のオブジェクトからのシンボル再配置の解決に使用するのと同じモデルに従います。
- 特別なハンドル RTLD_NEXT を使えば、呼び出し元のリンクマップリスト上に存在する次の関連オブジェクトから、シンボルを取得できる。

次に、一般的なケースを示します。この例では、アプリケーションはそのアドレス空間に追加オブジェクトを追加します。続いてアプリケーションは、dlsym(3C) を使用して関数シンボルまたはデータシンボルを見つけます。次に、アプリケーションは、これらのシンボルを使用して、これらの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。ファイル main.c には、次のコードが含まれます。

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

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

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

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

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

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。

上記のファイル main.c を使用して構築されたアプリケーション prog には、次のような依存関係があります。

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

dlopen(3C) で指定されたファイル名の値が 0 の場合、シンボル foo と bar が、まず prog で検索され、次に /lib/libc.so.1 で検索されます。

ハンドルは、シンボル検索を開始するルートを示します。このルートから、検索メカニズムは 83 ページの「再配置シンボルの検索」 で説明されているモデルと同じモデルに従います。

要求されたシンボルが配置されていない場合は、dlsym(3C) は、NULL 値を戻します。この場合、dlerror(3C) を使用すると、失敗した真の理由を表示できます。次の例では、アプリケーション prog はシンボル bar を配置できません。

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

機能のテスト

特別なハンドル RTLD_DEFAULT と RTLD_PROBE を使用すると、ほかのシンボルの有無を確認するためにアプリケーションをテストできます。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じモデルに従います。105 ページの「デフォルトのシンボル検索モデル」 を参照してください。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 が、まず prog で、次に /lib/libc.so.1 で検索されます。このコードフラグメントが、図 3–1 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、この順に継続して行われます。

このメカニズムによって、51 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参照の代わりに使用できる、堅牢かつ柔軟性のある代替機能が提供されます。

割り込みの使用

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の場所を見つけることができます。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());
        return (1);
}

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

この場合、foo は、prog に関連する共有オブジェクト (この例では /lib/libc.so.1) 内で検索されます。このコード部分が図 3–1 に示されている例からファイル B.so.1 に含まれている場合、foo は C.so.1 でのみ検索されます。

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、オブジェクト内の関数は、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。たとえば、次のコードフラグメントを共有オブジェクト malloc.so.1 内に配置します。

```
#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 (NULL);
                }
        }

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

malloc.so.1 は、malloc(3C) が通常存在するシステムライブラリ /lib/libc.so.1 の前に割り込ませることができます。こうすれば、malloc() に対するすべての呼び出しは、本来の関数が呼ばれて割り当てを行う前に、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、次のコマンドを使っても、上記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。前術の例では、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の書式設定を行なっていますが、これは、printf(3C) が使用する可能性のある malloc(3C) に起因する再帰を回避するためです。

動的実行可能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。

デバッグ支援

Solaris OS リンカーには、デバッギングライブラリとデバッギング mdb(1) モジュールが組み込まれています。デバッギングライブラリを使用すると、実行時のリンクプロセスをより詳細に監視できます。mdb(1) モジュールを使用すると、プロセスのデバッグを対話形式で行うことができます。

デバッギングライブラリ

デバッギングライブラリは、アプリケーションの実行と依存関係を理解したり、デバッグする場合に役立ちます。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。ただし、この情報の正確な形式は、リリースごとに若干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、デバッギン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。しかし、多くのものが一般的な関心を惹くものでしょう。

デバッグを有効にするには、環境変数 LD_DEBUG を使用します。すべてのデバッギングの出力は、接頭辞としてプロセス識別子を持っていて、デフォルトでは、標準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、必要なデバッギングタイプを示す必要があります。

LD_DEBUG で使用可能なトークンを表示するには、LD_DEBUG=help を使用します。どのような動的実行可能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。これは、情報が表示されたあとでプロセスが終了するためです。

```
$ LD_DEBUG=help prog
......
```

```
11693: files     display input file processing (files and libraries)
......
```

環境変数 LD_DEBUG_OUTPUT を使用すれば、標準エラーの代わりに使用する出力ファイルを指定できます。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。

セキュリティー保護されたアプリケーションのデバッギン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、もっとも有効なデバッギングオプションの 1 つです。次の例では、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可能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 -K pic -G bar.c

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

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

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

実行時シンボル結合は、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: .......
```

即時再配置で必要とされるシンボル bar は、アプリケーションが制御を取得する「前」に結合されます。これに対して、遅延再配置で要求されたシンボル foo は、アプリケーションが制御を受け取った後、関数が最初に呼び出されたときに結

合されます。この再配置は、遅延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。

LD_DEBUG=bindings,detail と設定すると、実際の結合位置の実アドレスと相対アドレスに関する追加情報が表示されます。

LD_DEBUG を使用すれば、検索パスの使用状況を表示できます。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:.  (RUNPATH/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:.  (RUNPATH/RPATH from file prog)
11775:  trying path=/tmp/bar.so.1
11775:  trying path=./bar.so.1
11775: .......
```

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。symbols と bindings を組み合わせれば、シンボル再配置処理の全容を把握できます。

```
$ LD_DEBUG=bindings,symbols prog
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: .......
```

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。このようにデータ参照検索が省略される原因は、コピーの再配置時に使用される最適化にあります。この再配置タイプの詳細については、147 ページの「コピー再配置」を参照してください。

デバッガモジュール

デバッガモジュールは、mdb(1) に読み込むことができる一群の dcmds および walkers を提供します。デバッガモジュールを使用すると、実行時リンカーのさまざまな内部データ構造を検査できます。このデバッグ情報の多くを理解するには、実行時リンカー内部に関する知識が必要です。こうした内部の情報は、リリースごとに異なる可能性があります。しかし、これらの情報は、動的にリンクされたプロセスの基本的な構成要素を明らかにし、さまざまなデバッグを助けます。

次の例に、mdb(1) とデバッガモジュールを使用したいくつかのシナリオを示します。

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

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

        if ((handle = dlopen("foo.so.1", RTLD_LAZY)) == NULL)
                return (1);

        if ((fptr = (void (*)())dlsym(handle, "foo")) == NULL)
                return (1);

        (*fptr)();
        return (0);
}
$ cc -o main main.c -R.
```

mdb(1) がデバッガモジュール ld.so を自動的に読み込まなかった場合は、明示的に読み込んでください。それにより、デバッガモジュールの機能を確認できます。

```
$ mdb main
> ::load ld.so
> ::dmods -l ld.so

ld.so
------------------------------------------------------------------
  dcmd Bind                 - Display a Binding descriptor
  dcmd Callers              - Display Rt_map CALLERS binding descriptors
  dcmd Depends              - Display Rt_map DEPENDS binding descriptors
  dcmd ElfDyn               - Display Elf_Dyn entry
  dcmd ElfEhdr              - Display Elf_Ehdr entry
  dcmd ElfPhdr              - Display Elf_Phdr entry
  dcmd Groups               - Display Rt_map GROUPS group handles
  dcmd GrpDesc              - Display a Group Descriptor
```

```
  dcmd GrpHdl               - Display a Group Handle
  dcmd Handles              - Display Rt_map HANDLES group descriptors
  ....
> ::bp main
> :r
```

プロセス内の動的オブジェクトは、リンクマップ Rt_map として表現され、このリンクマップは、リンクマップリスト上で管理されています。プロセスのすべてのリンクマップは、Rt_maps を使用して表示できます。

```
> ::Rt_maps
Link-map lists (dynlm_list): 0xffbfe0d0
-----------------------------------------------
  Lm_list: 0xff3f6f60  (LM_ID_BASE)
  ---------------------------------------------
    lmco        rtmap       ADDR()     NAME()
    ---------------------------------------------
    [0xc]       0xff3f0fdc 0x00010000 main
    [0xc]       0xff3f1394 0xff280000 /lib/libc.so.1
  ---------------------------------------------
  Lm_list: 0xff3f6f88  (LM_ID_LDSO)
  ---------------------------------------------
    [0xc]       0xff3f0c78 0xff3b0000 /lib/ld.so.1
```

個々のリンクマップは、Rt_map を使用して表示できます。

```
> 0xff3f9040::Rt_map
Rt_map located at: 0xff3f9040
     NAME: main
 PATHNAME: /export/home/user/main
     ADDR: 0x00010000         DYN: 0x000207bc
     NEXT: 0xff3f9460        PREV: 0x00000000
      FCT: 0xff3f6f18    TLSMODID:          0
     INIT: 0x00010710        FINI: 0x0001071c
   GROUPS: 0x00000000     HANDLES: 0x00000000
  DEPENDS: 0xff3f96e8     CALLERS: 0x00000000
    .....
```

オブジェクトの .dynamic セクションは、ElfDyn dcmd を使用して表示できます。次の例は、最初の 4 つのエントリを表示しています。

```
> 0x000207bc,4::ElfDyn
Elf_Dyn located at: 0x207bc
    0x207bc  NEEDED       0x0000010f
Elf_Dyn located at: 0x207c4
    0x207c4  NEEDED       0x00000124
Elf_Dyn located at: 0x207cc
    0x207cc  INIT         0x00010710
```

```
Elf_Dyn located at: 0x207d4
    0x207d4  FINI         0x0001071c
```

mdb(1) は、遅延ブレークポイントを設定するときにとても有用です。この例では、関数 foo() に対するブレークポイントが有用です。foo.so.1 に対して dlopen(3C) が実行されるまでは、このシンボルはデバッガにとって未知であるにもかかわらず、遅延ブレークポイントを設定すると、動的オブジェクトが読み込まれたときに、実ブレークポイントが設定されます。

```
> ::bp foo.so.1‘foo
> :c
> mdb: You’ve got symbols!
> mdb: stop at foo.so.1‘foo
mdb: target stopped at:
foo.so.1‘foo:   save      %sp, -0x68, %sp
```

この時点で、新しいオブジェクトが読み込まれました。

```
> *ld.so‘lml_main::Rt_maps
lmco    rtmap       ADDR()     NAME()
-----------------------------------------------
[0xc]   0xff3f0fdc 0x00010000 main
[0xc]   0xff3f1394 0xff280000 /lib/libc.so.1
[0xc]   0xff3f9ca4 0xff380000 ./foo.so.1
[0xc]   0xff37006c 0xff260000 ./bar.so.1
```

foo.so.1 のリンクマップは、dlopen(3C) から返されたハンドルを示しています。Handles を使用すると、ハンドルの構造体を展開できます。

```
> 0xff3f9ca4::Handles -v
HANDLES for ./foo.so.1
-----------------------------------------------
  HANDLE: 0xff3f9f60 Alist[used 1: total 1]
    ---------------------------------------------
    Group Handle located at: 0xff3f9f28
    ---------------------------------------------
        owner:               ./foo.so.1
        flags: 0x00000000    [ 0 ]
       refcnt:          1    depends: 0xff3f9fa0 Alist[used 2: total 4]
        ---------------------------------------------
        Group Descriptor located at: 0xff3f9fac
           depend: 0xff3f9ca4    ./foo.so.1
            flags: 0x00000003    [ AVAIL-TO-DLSYM,ADD-DEPENDENCIES ]
        ---------------------------------------------
        Group Descriptor located at: 0xff3f9fd8
           depend: 0xff37006c    ./bar.so.1
            flags: 0x00000003    [ AVAIL-TO-DLSYM,ADD-DEPENDENCIES ]
```

ハンドルの依存関係は、dlsym(3C) 要求を満たすハンドルのオブジェクトを表現するリンクマップのリストです。この例では、依存関係は foo.so.1 と bar.so.1 です。

注– 上の例は、デバッガモジュールの機能の基礎的な紹介になっていますが、正確なコマンド、使用方法、および出力は、リリースごとに異なる可能性があります。お使いのシステムで利用できる正確な機能については、mdb(1) の使用方法およびヘルプを参照してください。

◆ ◆ ◆ 第 4 章

4 共有オブジェクト

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。次の例では、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。

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

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分割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ほか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。このリンク編集中、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。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。

入力された共有オブジェクトはすべて、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で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、25 ページの「実行時リンク」を参照してください。

命名規約

リンカーも実行時リンカーも、ファイル名によるファイルの解釈は行いません。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 (214 ページの「ELF ヘッダー」 を参照)。この情報から、リンカーはファイルの処理条件を推定します。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` オプションを使用する方が一般的です。35 ページの「共有オブジェクトの処理」を参照。

このリンカー処理に適用する共有オブジェクトには、接頭辞 `lib` と接尾辞 `.so` を指定する必要があります。たとえば、`/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、`/lib/libc.so.1` の64 ビット版は、`/lib/64/libc.so.1` です。

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。幾世代にも渡って公開される共有オブジェクトのインタフェースを変更できるようにするには、共有オブジェクトをバージョン番号の付いたファイル名にします。

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、`/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、慣習的な `lib` 接頭辞をその名前に付けないことがあります。このカテゴリに属する共有オブジェクトの例には、`dlopen(3C)` 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、接頭辞 `.so` は付けることを推奨します。また、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。バージョン番号の付け方については、第 5 章「アプリケーションバイナリインタフェースとバージョン管理」を参照してください。

注 – `dlopen(3C)` で使用される共有オブジェクト名は通常、名前に「/」が付かない「単純」ファイル名として表されます。実行時リンカーは、この規則を使用して、実際のファイルを検索できます。詳細は、90 ページの「追加オブジェクトの読み込み」を参照してください。

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の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
$ elfdump -d prog | grep NEEDED
       [1]  NEEDED        0x123         libfoo.so.1

$ cc -o prog main.o ../tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
       [1]  NEEDED        0x123         ../tmp/libfoo.so

$ cc -o prog main.o /usr/tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
       [1]  NEEDED        0x123         /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、リンク編集中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中、-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。次の例では、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*」と呼ばれます。

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

次の例は、elfdump(1) を使用して SONAME タグを持つエントリを参照し、soname の記録を表示する方法を示しています。

```
$ elfdump -d ../tmp/libfoo.so | grep SONAME
       [1]  SONAME        0x123         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。

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
$ elfdump -d prog | grep NEEDED
       [1]  NEEDED        0x123         libfoo.so

$ cc -o prog main.o ../tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
       [1]  NEEDED        0x123         libfoo.so

$ cc -o prog main.o /usr/tmp/libfoo.so
$ elfdump -d prog | grep NEEDED
       [1]  NEEDED        0x123         libfoo.so
```

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。この規約では、実行時リンカーが規則を使用して実際のファイルを検索できます。詳細は、78 ページの「共有オブジェクトの依存関係の検索」を参照してください。

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、リンク編集の要件を満たすためにアーカイブから抽出できます。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。アーカイブ抽出の条件の詳細については、34 ページの「アーカイブ処理」を参照してください。

アーカイブメンバー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
$ elfdump -d main | grep NEEDED
       [1]  NEEDED        0x123         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、この例では、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します。

記録名の衝突

共有オブジェクトが動的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ます。これらの検査により、出力ファイル内に記録される依存関係名すべてが一意となります。

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」を含んでいる場合、依存関係名の衝突が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: recording name conflict: file ‘./libfoo.so’ and \
    file ‘./libbar.so’ provide identical dependency names: libsame.so.1
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、同じリンク編集中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、同様のエラー状態が発生します。

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。

```
$ cc -o libbar.so -G -K pic -h libsame.so.1 bar.c -L. -lfoo
ld: fatal: recording name conflict: file ‘./libfoo.so’ and \
    -h option provide identical dependency names: libsame.so.1
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

共有オブジェクトは独自の依存関係を持つことができます。79 ページの「実行時リンカーが検索するディレクトリ」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検索ディレクトリの中にない場合、実行時リンカーに検索場所を明示的に指示する必要があります。32 ビットオブジェクトの場合、デフォルト検索ディレクトリは `/lib` と `/usr/lib` です。64 ビットオブジェクトの場合、デフォルト検索ディレクトリは `/lib/64` と `/usr/lib/64` です。デフォルト以外の検索パスが必要なことを示すには、依存関係のあるオブジェクトに実行パスを記録する方法がお勧めです。「実行パス」は、リンカーの `-R` オプションで記録できます。

次の例では、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にあるものと予期します。

```
$ cc -o libbar.so -G -K pic bar.c
$ cc -o libfoo.so -G -K pic foo.c -R/home/me/lib -L. -lbar
$ elfdump -d libfoo.so | egrep "NEEDED|RUNPATH"
      [1]  NEEDED        0x123         libbar.so.1
      [2]  RUNPATH       0x456         /home/me/lib
```

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。これらの「実行パス」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。

LD_LIBRARY_PATH ファミリの環境変数の範囲は、よりグローバルです。これら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。これらの環境変数は、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、79 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。

依存関係の順序

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
```

共有オブジェクト libfoo.so.1 の依存関係に対して指定した処理順序の要件は、動的実行可能ファイル prog を構築した場合、保証されません。

シンボルの割り込みと .init セクションの処理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

共有オブジェクトは、「フィルタ」として機能するように定義できます。この手法には、フィルタが提供するインタフェースと、代替共有オブジェクトとの関連付けが含まれます。代替共有オブジェクトは実行時に、「フィルタ」により提供される 1 つまたは複数のインタフェースを供給します。この代替共有オブジェクトは「フィルティー」と呼ばれます。「フィルティー」は、共有オブジェクトと同じように構築されます。

フィルタ処理は、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。リンク編集時には、フィルタインタフェースに結合するシンボル参照は、フィルタシンボル定義に解決されます。実行時には、フィルタインタフェースに結合するシンボル参照は代替共有オブジェクトにリダイレクトできます。

mapfile のキーワードである FILTER または AUXILIARY を使用することで、共有オブジェクト内に定義された個別インタフェースをフィルタとして定義できます。また、特定の共有オブジェクトが提供するすべてのインタフェースをフィルタとして定義することもできます。それには、リンカーの -F または -f フラグを使用します。これらの手法は、一般に個別に使用されますが、同じ共有オブジェクトの中で組み合わせることもできます。

フィルタ処理には、次に示す 2 つの形があります。

標準フィルタ処理
: このフィルタ処理で必要となるのは、フィルタ処理対象のインタフェースのシンボルテーブルエントリだけです。実行時には、「フィルティー」からフィルタシンボル定義の実装を提供する必要があります。

  リンカーの mapfile キーワード FILTER またはリンカーの -F フラグを使用すると、インタフェースは標準フィルタとして機能するように定義されます。この mapfile キーワードまたはフラグは、実行時にシンボル定義を提供する必要がある 1 つ以上の「フィルティー」の名前で修飾されます。

  実行時に処理できない「フィルティー」はスキップされます。「フィルティー」内に標準フィルタシンボルが見つからない場合も、「フィルティー」はスキップされます。どちらの場合も、フィルタにより提供されるシンボル定義は、このシンボル検索を満たすためには使用されません。

補助フィルタ処理
このフィルタ処理は標準フィルタ処理と類似したメカニズムを提供しますが、補助フィルタインタフェースに対応するフォールバック実装がフィルタに含まれる点が異なります。実行時には、「フィルティー」からシンボル定義の実装を提供できます。

リンカーの mapfile キーワード AUXILIARY またはリンカーの -f フラグを使用すると、インタフェースは補助フィルタとして機能するように定義されます。この mapfile キーワードまたはフラグは、実行時にシンボル定義を提供できる 1 つ以上の「フィルティー」の名前で修飾されます。

実行時に処理できない「フィルティー」はスキップされます。「フィルティー」内に補助フィルタシンボルが見つからない場合も、「フィルティー」はスキップされます。どちらの場合も、フィルタにより提供されるシンボル定義は、このシンボル検索を満たすために使用されます。

標準フィルタの生成

標準フィルタを生成するには、まずフィルタ処理を適用する「フィルティー」を定義する必要があります。次の例では、シンボル foo と bar を提供する「フィルティー」 filtee.so.1 を構築します。

```
$ cat filtee.c
char * bar = "defined in filtee";

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

標準フィルタ処理は、2 つの方法のいずれかで実行できます。共有オブジェクトによって提供されるすべてのインタフェースをフィルタとして宣言するには、リンカーの -F フラグを使用します。フィルタとなる共有オブジェクトの個々のインタフェースを宣言するには、リンカーの mapfile と FILTER キーワードを使用します。

次の例では、共有オブジェクト filter.so.1 がフィルタとして定義されています。filter.so.1 はシンボル foo と bar を提供し、それ自体が「フィルティー」 filtee.so.1 のフィルタです。この例では、コンパイラドライバが -F オプションを解釈しないように、環境変数 LD_OPTIONS が使用されています。

```
$ cat filter.c
char * bar = NULL;

char * foo()
```

```
{
    return (NULL);
}
$ LD_OPTIONS='-F filtee.so.1' \
cc -o filter.so.1 -G -K pic -h filter.so.1 -R. filter.c
$ elfdump -d filter.so.1 | egrep "SONAME|FILTER"
    [2]  SONAME           0xee     filter.so.1
    [3]  FILTER           0xfb     filtee.so.1
```

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、リンカーは標準フィルタ filter.so.1 を依存関係として参照できます。リンカーは、フィルタのシンボルテーブルの情報を使用してシンボル解決を行います。しかし、実行時にフィルタのシンボルを参照すると、必ず「フィルティー」filtee.so.1 がさらに読み込まれます。実行時リンカーはこの「フィルティー」を使用して、filter.so.1 によって定義されたシンボルを解決します。この「フィルティー」が見つからないか、あるいは「フィルティー」内にフィルタシンボルが見つからない場合は、このシンボル検索でそのフィルタはスキップされます。

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 filter.so.1 からのリンク編集中に解決されます。prog を実行すると、foo と bar が、フィルタ filter.so.1 からではなく、「フィルティー」 filtee.so.1 から取得されます。

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

void main()
{
        (void) printf("foo is %s: bar is %s\n", foo(), bar);
}
$ cc -o prog main.c -R. filter.so.1
$ prog
foo is defined in filtee: bar is defined in filtee
```

次の例では、共有オブジェクト filter.so.2 は、インタフェースの 1 つである foo をフィルティー filtee.so.1 上のフィルタとして定義します。

注 – foo() にはソースコードが提供されていないので、mapfile のキーワードである FUNCTION を使用して、foo のシンボルテーブルエントリが作成されることを確認します。

```
$ cat filter.c
char * bar = "defined in filter";
$ cat mapfile
{
    global:
```

```
        foo = FUNCTION FILTER filtee.so.1;
};
$ cc -o filter.so.2 -G -K pic -h filter.so.2 -M mapfile -R. filter.c
$ elfdump -d filter.so.2 | egrep "SONAME|FILTER"
    [2]  SONAME           0xd8     filter.so.2
    [3]  SUNW_FILTER      0xfb     filtee.so.1
$ elfdump -y filter.so.2 | egrep "foo|bar"
    [1]  F    [3] filtee.so.1      foo
   [10]  D        <self>           bar
```

実行時にフィルタのシンボル `foo` を参照すると、必ず「フィルティー」`filtee.so.1` がさらに読み込まれます。実行時リンカーは、「フィルティー」を使用して、`filter.so.2` が定義したシンボル `foo` だけを解決します。シンボル `bar` への参照は、 `filter.so.2` からのシンボルを常に使用し、このシンボルに対して「フィルティー」処理は定義されません。

たとえば、次の動的実行可能ファイル `prog` は、フィルタ `filter.so.2` からのリンク編集中に解決されるシンボル `foo` と `bar` を参照します。`prog` の実行により、`foo` が「フィルティー」`filtee.so.1` から取得され、`bar` がフィルタ `filter.so.2` から取得されます。

```
$ cc -o prog main.c -R. filter.so.2
$ prog
foo is defined in filtee: bar is defined in filter
```

これらの例では、「フィルティー」`filtee.so.1` がフィルタ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ほかのオブジェクトからのシンボル参照を満たすために、「フィルティー」を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を定義するための便利なメカニズムを提供します。標準フィルタは、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作成します。標準フィルタはまた、インタフェースをその実装にリダイレクトする手段も提供します。一部の標準フィルタは、Solaris OS で使用されます。

`/usr/lib/libsys.so.1` フィルタは、標準 C ライブラリ `/usr/lib/libc.so.1` のサブセットを提供します。このサブセットは、ABI に準拠するアプリケーションが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します。

`/lib/libxnet.so.1` フィルタは、複数の「フィルティー」を使用します。このライブラリは、`/lib/libsocket.so.1`、 `/lib/libnsl.so.1`、および `/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。

`libc.so.1` は、実行時リンカーへのインタフェースフィルタを定義します。これらのインタフェースは、`libc.so.1` のコンパイル環境で参照されるシンボルと `ld.so.1(1)` の実行時環境内で作り出される実際の実装結合間の抽象化を提供します。

libnsl.so.1は、標準フィルタ gethostname(3C) を libc.so.1 に対して定義します。以前は、libnsl.so.1 も libc.so.1 もこのシンボルの同じ実装を提供していました。libnsl.so.1 をフィルタとして設定することで、gethostname() の実装は 1 つだけ必要となります。libnsl.so.1 は継続して gethostname() をエクスポートするため、このライブラリインタフェースも以前のリリースと互換性があります。

標準フィルタ内のコードは実行時に参照されることはないため、フィルタとして定義された関数に内容を追加しても意味がありません。どのようなフィルタコードでも再配置を必要とする場合があり、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。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。

フィルタ内にデータシンボルを生成するときは、常にデータをセクションに関連付けてください。この関連付けは、再配置可能なオブジェクトファイル内にシンボルを定義することで行うことができます。この関連付けは、mapfile 内でシンボルを size 宣言あり、 value 宣言なしで定義しても行うことができます。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。このようにデータを定義することで、動的実行可能ファイルからの参照が正しく確立されます。

リンカーによって実行される、より複雑なシンボル解決の中には、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。このため、フィルタ内のシンボルの属性が「フィルティー」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必要があります。属性の一貫性を維持することで、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。44 ページの「シンボル解決」を参照してください。

注 – リンカーは、処理される最初の再配置可能ファイルの ELF クラスを使用して、作成するオブジェクトのクラスを管理します。64 ビットフィルタを mapfile だけから作成するには、リンカーの -64 オプションを使用します。

補助フィルタの生成

補助フィルタを生成するには、まずフィルタ処理を適用する「フィルティー」を定義する必要があります。次の例では、シンボル foo を提供する「フィルティー」filtee.so.1 を構築します。

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

補助フィルタ処理は、2つの方法のいずれかで提供できます。共有オブジェクトによって提供されるすべてのインタフェースを補助フィルタとして宣言するには、リンカーの -f フラグを使用します。共有オブジェクトの個々のインタフェースを補助フィルタとして宣言するには、リンカーの mapfile と AUXILIARY キーワードを使用します。

次の例では、共有オブジェクト filter.so.1 が補助フィルタとして定義されています。filter.so.1 はシンボル foo と bar を提供し、それ自体が「フィルティー」 filtee.so.1 の補助フィルタです。この例では、コンパイラドライバが -f オプションを解釈しないように、環境変数 LD_OPTIONS が使用されています。

```
$ cat filter.c
char * bar = "defined in filter";

char * foo()
{
        return ("defined in filter");
}
$ LD_OPTIONS='-f filtee.so.1' \
cc -o filter.so.1 -G -K pic -h filter.so.1 -R. filter.c
$ elfdump -d filter.so.1 | egrep "SONAME|AUXILIARY"
    [2]  SONAME           0xee     filter.so.1
    [3]  AUXILIARY        0xfb     filtee.so.1
```

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、リンカーは補助フィルタ filter.so.1 を依存関係として参照できます。リンカーは、フィルタのシンボルテーブルの情報を使用してシンボル解決を行います。しかし、実行時にフィルタのシンボルを参照すると、「フィルティー」 filtee.so.1 が検索されます。この「フィルティー」が見つかると、実行時リンカーは、この「フィルティー」を使用して、filter.so.1 によって定義されたすべてのシンボルを解決します。この「フィルティー」が見つからないか、あるいは「フィルティー」内にフィルタからのシンボルが見つからない場合は、フィルタ内の元のシンボルが使用されます。

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 filter.so.1 からのリンク編集中に解決されます。prog を実行すると、foo が、フィルタ filter.so.1 からではなく、「フィルティー」 filtee.so.1 から取得されます。しかし、bar はフィルタ filter.so.1 から取得されます。これは、「フィルティー」 filtee.so.1 内にこのシンボルの代替定義が存在しないためです。

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

void main()
{
        (void) printf("foo is %s: bar is %s\n", foo(), bar);
}
```

```
$ cc -o prog main.c -R. filter.so.1
$ prog
foo is defined in filtee: bar is defined in filter
```

次の例では、共有オブジェクト filter.so.2 は、インタフェース foo をフィルティーfiltee.so.1 上の補助フィルタとして定義します。

```
$ cat filter.c
char * bar = "defined in filter";

char * foo()
{
        return ("defined in filter");
}
$ cat mapfile
{
    global:
        foo = AUXILIARY filtee.so.1;
};
$ cc -o filter.so.2 -G -K pic -h filter.so.2 -M mapfile -R. filter.c
$ elfdump -d filter.so.2 | egrep "SONAME|AUXILIARY"
    [2]  SONAME           0xd8     filter.so.2
    [3]  SUNW_AUXILIARY   0xfb     filtee.so.1
$ elfdump -y filter.so.2 | egrep "foo|bar"
    [1]  A    [3] filtee.so.1      foo
   [10]  D        <self>           bar
```

実行時にフィルタのシンボル foo を参照すると、必ず「フィルティー」filtee.so.1 が検索されます。「フィルティー」が見つかると、「フィルティー」が読み込まれます。「フィルティー」は filter.so.2 によって定義されたシンボル foo の解決に使用されます。「フィルティー」が検索されなかった場合、filter.so.2 によって定義されたシンボル foo が使用されます。シンボル bar への参照は、 filter.so.2 からのシンボルを常に使用し、このシンボルに対して「フィルティー」処理は定義されません。

たとえば、次の動的実行可能ファイル prog は、フィルタ filter.so.2 からのリンク編集中に解決されるシンボル foo と bar を参照します。「フィルティー」filtee.so.1 が存在する場合、prog の実行により foo が「フィルティー」filtee.so.1 から、bar がフィルタ filter.so.2 から取得されます。

```
$ cc -o prog main.c -R. filter.so.2
$ prog
foo is defined in filtee: bar is defined in filter
```

「フィルティー」 filtee.so.1が存在しない場合、prog を実行すると、foo と bar がフィルタ filter.so.2 から取得されます。

```
$ prog
foo is defined in filter: bar is defined in filter
```

これらの例では、「フィルティー」filtee.so.1がフィルタ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ほかのオブジェクトからのシンボル参照を満たすために、「フィルティー」を使用することができません。

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、ハードウェア機能での最適な機能およびプラットフォーム固有の共有オブジェクトを提供するためにSolaris OSで使用されます。例は、385ページの「ハードウェア機能固有の共有オブジェクト」、387ページの「命令セット固有の共有オブジェクト」、および389ページの「システム固有の共有オブジェクト」を参照してください。

注 – 環境変数LD_NOAUXFLTRを設定すれば、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」の使用およびそれらの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。

フィルタ処理の組み合わせ

標準フィルタを定義する個別インタフェースと補助インタフェースを定義する個別インタフェースを、同一の共有オブジェクト内に定義することができます。こうしたフィルタ定義の組み合わせを実現するには、mapfileのキーワードであるFILTERとAUXILIARYを使って、必要な「フィルティー」を割り当てます。

-Fまたは-fオプションを使用して自身のインタフェースのすべてをフィルタとして定義する共有オブジェクトは、標準フィルタか補助フィルタのどちらかです。

共有オブジェクトでは、個々のインタフェースをフィルタとして機能するように定義するとともに、そのオブジェクトのすべてのインタフェースをフィルタとして機能するように定義することができます。その場合、特定のインタフェースに対して定義された個別フィルタ処理が、まず処理されます。個別インタフェースフィルタに対する「フィルティー」を確立できなかった場合は、フィルタのすべてのインタフェースに対して定義された「フィルティー」が必要に応じてフォールバックを提供します。

たとえば、フィルタfilter.so.1があるとします。このフィルタでは、すべてのインタフェースが「フィルティー」filtee.so.1に対する補助フィルタとして機能するように、リンカーの-fフラグを使って定義されています。さらにfilter.so.1では、個別インタフェースfooが「フィルティー」foo.so.1に対する標準フィルタとなるように、mapfileのキーワードFILTERを使って定義されています。さらに

filter.so.1 では、個別インタフェース bar が「フィルティー」bar.so.1 に対する補助フィルタとなるように、mapfile のキーワード AUXILIARY を使って定義されています。

foo への外部参照が発生すると、「フィルティー」foo.so.1 が処理されます。foo が foo.so.1 で見つからなかった場合、このフィルタに対する処理はそれ以上実行されません。この場合にフォールバック処理が実行されない理由は、foo が標準フィルタとして定義されているからです。

bar への外部参照が発生すると、「フィルティー」bar.so.1 が処理されます。bar が bar.so.1 で見つからなかった場合、「フィルティー」filtee.so.1 によるフォールバック処理が実行されます。この場合にフォールバック処理が実行される理由は、bar が補助フィルタとして定義されているからです。bar が filtee.so.1 で見つからなかった場合、最終的にはフィルタ filter.so.1 内の bar の定義に基づいて外部参照が解決されます。

「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が参照されるまで、フィルティーの読み込みを遅延します。この実装は、必要に応じてモード RTLD_LOCAL を使用して各「フィルティー」に対して dlopen(3C) 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作成される、依存関係の報告における違いの原因となるものです。

実行時に-「フィルティー」の即時処理を起動するフィルタを作成する場合には、リンカーの z loadfltr オプションを使用できます。さらに、LD_LOADFLTR 環境変数を任意の値に設定することで、プロセス内のすべての「フィルティー」の即時処理を開始できます。

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。共有オブジェクトの性能は、共有オブジェクトを使用するアプリケーションだけでなく、システム全体に影響します。

共有オブジェクト内のコードは、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の実行時処理に関連した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。

ファイルの解析

ELF ファイルの内容を解析するときに、さまざまなツールを利用できます。ファイルのサイズを表示するには、size(1) コマンドを使用します。

```
$ 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」として知ら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一部がメモリーにどのようにマップされているかを記述しています。mmap(2) のマニュアルページを参照してください。これらの読み込み可能セグメントは、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。

```
$ elfdump -p -NPT_LOAD libfoo.so.1

Program Header[0]:
    p_vaddr:      0           p_flags:    [ PF_X PF_R ]
    p_paddr:      0           p_type:     [ PT_LOAD ]
    p_filesz:     0x59c       p_memsz:    0x59c
    p_offset:     0           p_align:    0x10000

Program Header[1]:
    p_vaddr:      0x10630     p_flags:    [ PF_X PF_W PF_R ]
    p_paddr:      0           p_type:     [ PT_LOAD ]
    p_filesz:     0x10c       p_memsz:    0x12c
    p_offset:     0x630       p_align:    0x10000
```

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がマップされ、その内容 PF_X PF_R の読み込みと実行が可能になります。データセグメントもマップされ、その内容 PF_W が変更できるようになります。データセグメントのメモリーサイズ (p_memsz) は、ファイルサイズ (p_filesz) とは異なります。この違いは、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。

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

[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) の例による出力は、セクション .init、.text、および .fini に対する関数 _init、foo、および _fini の関連付けを示しています。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。

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ほかのユーザーに対して何も影響しません。ただし、このページはほか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。

メモリーにマッピングされた共有オブジェクトのセグメントは、参照専用の「テキスト」セグメントと読み書きが可能な「データ」セグメントに分類されます。ELF ファイルからこの情報を取得する方法については、135 ページの「ファイルの解析」を参照してください。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。

動的依存関係の遅延読み込み

オブジェクトを遅延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。91 ページの「動的依存関係の遅延読み込み」を参照してください。

小さいアプリケーションの場合、典型的な 1 つの実行スレッドでアプリケーションのすべての依存関係を参照することができます。この場合、依存関係が遅延読み込み可に設定されているかどうかに関係なく、アプリケーションはすべての依存関係を読み込みます。しかし、遅延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ります。

多くの依存関係を持つアプリケーションの場合、遅延読み込みを使用すると、一部の依存関係がまったく読み込まれないことがあります。特定の実行スレッドで参照されない依存関係は読み込まれません。

位置独立のコード

動的実行可能ファイル内のコードは通常「位置に依存」し、メモリー内の固定アドレスに結び付けられています。一方、共有オブジェクトは、異なるプロセス内の異なる位置に読み込むことができます。位置独立のコードは、特定のアドレスに結び付けられていません。このように独立しているため、コードは、そのコードを使用する各プロセス内の異なるアドレスで実際に実行できます。位置独立のコードは、共有オブジェクトを作成する場合に推奨します。

コンパイラは、`-K pic` オプションによって、位置独立のコードを生成できます。

共有オブジェクトが位置に依存するコードで構築されている場合、テキストセグメントには実行時に変更が必要となる場合があります。このような変更により、再配置可能な参照を、オブジェクトが読み込まれている位置に割り当てることができます。テキストセグメントの再配置を行うには、セグメントを書き込み可能として再度マッピングする必要があります。このような変更にはスワップ空間の予約が必要で、またプロセスのテキストセグメントの非公開コピーが行われます。テキストセグメントは複数のプロセス間では共有できなくなります。位置に依存するコードは、通常、対応する位置独立のコードよりも多くの実行時再配置を必要とします。概して、テキスト再配置を処理するオーバーヘッドは、重大な性能の低下の原因になる可能性があります。

位置独立のコードから構築された共有オブジェクトでは、そのデータセグメント内のデータを介した間接参照として、再配置可能な参照が生成されます。テキストセグメント内のコードは変更する必要はありません。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。特定の間接参照のテクニックの詳細については、311 ページの「大域オフセットテーブル (プロセッサ固有)」と 312 ページの「プロシージャーのリンクテーブル (プロセッサ固有)」を参照してください。

このような再配置が存在する場合、実行時リンカーはテキスト再配置を処理しようとします。ただし、一部の再配置は実行時に処理できません。

x64 の位置に依存するコードのシーケンスは、下位 32 ビットのメモリーにのみ読み込み可能なコードを生成します。上位 32 ビットのアドレスはすべてゼロである必要があります。通常、共有オブジェクトはメモリーの最上位に読み込まれるため、上位 32 ビットのアドレスが必要になります。そのため、x64 共有オブジェクト内の、位置に依存するコードは、再配置の要件に対処するのに不十分です。共有オブジェクト内でそのようなコードを使用すると、実行時再配置エラーが発生する可能性があります。

```
$ prog
ld.so.1: prog: fatal: relocation error: R_AMD64_32: file \
    libfoo.so.1: symbol (unknown): value 0xfffffd7fff0cd457 does not fit
```

位置独立のコードはメモリー内の任意の場所に読み込めるため、x64 の共有オブジェクトの要件を満たします。

このような状況は、64 ビット SPARCV9 コードに使用されるデフォルトの ABS64 モードとは異なります。位置に依存するこのコードは通常、完全な 64 ビットアドレス範囲と互換性があります。したがって、位置に依存するコードのシーケンスは、SPARCV9 共有オブジェクト内に存在できます。64 ビット SPARCV9 コードに ABS32 モードまたは ABS44 モードのいずれかを使用しても、実行時に解決できない再配置が生じる可能性があります。ただし、これらの各モードでは、実行時リンカーがテキストセグメントを再配置する必要があります。

実行時リンカーの機能や、再配置要件の違いに関係なく、共有オブジェクトは位置独立のコードを使用して構築するべきです。

共有オブジェクトのうち、テキストセグメントに対して再配置を必要とするものを識別できます。次の例では、elfdump(1) を使用して、TEXTREL エントリという動的エントリが存在するかどうかを判別します。

```
$ cc -o libfoo.so.1 -G -R. foo.c
$ elfdump -d libfoo.so.1 | grep TEXTREL
       [9]  TEXTREL       0
```

注 – TEXTREL エントリの値は関係ありません。共有オブジェクトにこのエントリが存在する場合は、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が作成されるのを防ぐには、リンカーの -z text フラグを使用します。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するすべてのコードの出所を指摘する診断を生成します。次の例に、位置に依存するコードが、共有オブジェクトの生成にどのように失敗するかを示します。

```
$ 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
```

ファイル foo.o から位置依存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。

手書きのアセンブラコードが含まれ、その中に、位置独立の適切なプロトタイプが含まれていない場合、共有オブジェクト内ではテキスト再配置が発生する可能性もあります。

注 – いくつかの単純なソースファイルをテストしながら、位置に依存しないコードを決定することもできます。中間アセンブラ出力を生成するコンパイラ機能を使用してください。

SPARC: -K pic と -K PIC オプション

SPARC バイナリでは、-K pic オプションと代替の -K PIC オプションの動作がわずかに違っており、大域オフセットテーブルエントリの参照方法が異なります。311 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。

大域オフセットテーブルはポインタの配列で、エントリのサイズは、32 ビット (4 バイト) および 64 ビット (8 バイト) に固定です。次のコード例は、-K pic を使用して生成されるエントリを参照します。

```
        ld    [%l7 + j], %o0    ! load &j into %o0
```

%l7 には、あらかじめ計算された参照元オブジェクトのシンボル _GLOBAL_OFFSET_TABLE_ の値が代入されます。

このコード例は、大域オフセットテーブルのエントリ用に 13 ビットの変位定数を提供します。つまり、この変位は、32 ビットのオブジェクトの場合は 2048 個の一意のエントリを提供し、64 ビットのオブジェクトの場合は 1024 個の一意のエントリを提供します。返されるエントリ数より多くのエントリを要求するオブジェクトの場合、リンカーは致命的なエラーを生成します。

```
$ cc -K pic -G -o lobfoo.so.1 a.o b.o ... z.o
ld: fatal: too many symbols require ‘small’ PIC references:
        have 2050, maximum 2048 -- recompile some modules -K PIC.
```

このエラー状態を解決するには、入力再配置可能オブジェクトの一部をコンパイルするときに、-K PIC オプションを指定します。このオプションは、32 ビットの定数を大域オフセットテーブルエントリに使用します。

```
        sethi %hi(j), %g1
        or    %g1, %lo(j), %g1    ! get 32–bit constant GOT offset
        ld    [%l7 + %g1], %o0    ! load &j into %o0
```

elfdump(1) を -G オプションとともに使用すれば、オブジェクトの大域オフセットテーブルの要件を調べることができます。リンカーのデバッグトークン -D got,detail を使用すれば、リンク編集中のこれらのエントリの処理を確認することもできます。

頻繁にアクセスするデータ項目に対しては、-K pic を使用する方法が有利です。どちらの方法でもエントリを参照することはできます。しかし、再配置可能オブジェクトをどちらの方法でコンパイルしたらいいのか決めるのには時間がかかる

上、性能はわずかしか改善されません。すべての再配置型オブジェクトを -K PIC オプションを指定して再コンパイルする方が一般には簡単です。

使用されない対象物の削除

構築中のオブジェクトによって使用されない関数やデータを保持することは無駄です。この対象物によってオブジェクトが肥大し、不必要な再配置のオーバーヘッドやそれに関連したページング動作が生じます。使用されない依存関係への参照も無駄です。これらの参照によって、ほかの共有オブジェクトの不必要な読み込みと処理が生じます。

リンカーのデバッギングトークン -D unused を使用すると、リンク編集中に使用されないセクションが表示されます。未使用として特定されるセクションは、リンク編集から削除してください。未使用セクションは、リンカーの -z ignore オプションを使用して削除できます。

リンカーは、次の条件のときに再配置可能オブジェクトのセクションを未使用であると判断します。

- セクションが割り当て可能
- このセクションに結合 (再配置) するほかのセクションがない
- セクションが大域シンボルを提供しない

共有オブジェクトの外部インタフェースを定義することによって、セクションを削除するリンカーの機能を改善することができます。インタフェースを定義することによって、インタフェースの一部として定義されなかった大域シンボルは局所シンボルになります。ほかのオブジェクトから参照されていない局所シンボルは、排除の候補であると明確に識別されます。

関数やデータ変数が独自のセクションに割り当てられている場合、リンカーはこのような関数やデータ変数を個別に排除できます。このセクションの細分化は、-xF などのコンパイラオプションを使用して行います。以前のコンパイラには、関数を独自のセクションに割り当てる機能しかありませんでした。最近のコンパイラでは、-xF 構文が拡張されて、データ変数を独自のセクションに割り当てることができます。以前のコンパイラでは、-xF を使用するときには、C++ 例外処理を無効にする必要がありました。最近のコンパイラでは、この制限はなくなりました。

再配置可能オブジェクトの割り当て可能なセクションすべてが排除可能な場合、そのファイル全体がリンク編集から削除されます。

入力ファイルの排除に加えて、リンカーは使用されていない依存関係を判断できます。構築しているオブジェクトによって結合されていない場合、その依存関係は使用されていないと判断されます。未使用の依存関係の記録を排除するために、-z ignore オプションでオブジェクトを使用できます。

-z ignore オプションが適用されるのは、リンカーのコマンド行上でこのオプションのあとに指定したファイルだけです。-z ignore オプションは -z record によって取り消されます。

共有可能性の最大化

137 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ます。オブジェクトの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、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、const 宣言を使用して、テキストセグメントに移動する必要があります。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。ただし、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" };
```

次の定義を使用するほうが良いと思われるかもしれません。

```
const char * const rdstrs[] = { ..... };
```

この定義により、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。配列を次のように表現してみます。

```
const char * rdstrs[] = { ..... };
```

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。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。

注 – コンパイラによっては、位置独立のコードを生成するときに、実行時に再配置を行う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。このようなコンパイラは、このような項目を書き込み可能なセグメントに配置します。たとえば、.picdata です。

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、ほか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。次の例では、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。

```
$ strings -10 libfoo.so.1 | sort | uniq -c | sort -rn
```

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。

バッファーの動的割り当て

大きなデータバッファーは、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で必要なバッファー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ー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

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す。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る可能性があります。

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、使用頻度の高いほか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) などのツールは、コードカバレージとプロファイリングを判定するために役立ちます。

関連する機能は、各自の共有オブジェクトに分離してください。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ました。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、実際に使用頻度のもっとも高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してください。これにより、参照の近傍性が改善するだけでなく、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

82 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。83 ページの「再配置シンボルの検索」と 87 ページの「再配置が実行されるとき」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。

シンボルの検索

実行時リンカーは、特定のシンボルを検索する必要が生じると、デフォルトでは各オブジェクト内でそのシンボルを検索します。まず動的実行可能プログラムから検索してから、 オブジェクトが読み込まれた順番に共有オブジェクトを検索します。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。

この状況では、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。詳細は、61 ページの「シンボル範囲の縮小」を参照してください。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ければならなくなります。

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。61 ページの「シンボル範囲の縮小」を参照してください。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。

直接結合を使用すると、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も大幅に削減できます。86 ページの「直接結合」を参照してください。

再配置が実行されるとき

すべての即時参照再配置は、アプリケーションが制御を取得する前の、プロセスの初期設定中に実行する必要があります。これに対して、遅延参照は、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。即時参照は通常、データ参照によって行われます。このため、データ参照の数を少なくすることによって、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、データ参照を関数参照に変換して延期することもできます。たとえば、機能インタフェースによってデータ項目を返すことができます。この変換を行うと、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ため、性能は明らかに向上します。いくつかの機能インタ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。

機能インタフェースを使用した場合の利点については、147 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説明します。また、この再配置によるオーバーヘッドを回避する方法の例も示します。

再配置セクションの結合

再配置可能オブジェクト内の再配置セクションは通常、再配置の適用対象となるセクションとの 1 対 1 の関係が維持されます。ただし、実行可能ファイルまたは共有オ

ブジェクトを -z combreloc オプションによって構築すると、プロシージャーリンクテーブル再配置を除くすべてが、.SUNW_reloc という単一の共通セクションに置かれます。

この方法で再配置レコードを結合すると、すべての RELATIVE 再配置を 1 つにグループ化できます。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ます。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、位置独立の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。詳細は、139 ページの「位置独立のコード」を参照してください。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。これらの更新されたテーブルによって、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、位置独立のコードからは作成されません。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、回避する必要があります。コピー再配置という再配置手法が、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とします。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すべての参照を満たすために使用されます。動的実行可能ファイルは、データ項目を継承します。この項目を参照するプロセス内のほか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使用されなくなります。

このメカニズムの次の例では、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 つめ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]);
}
```

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します。

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
..........
```

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。新しいエラーメッセージが追加されるなど、この配列のサイズが変わったとします。この配列を参照する動的実行可能ファイルすべてで、新しいエラーメッセージにアクセスするための新しいリンク編集を行う必要がありま

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、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します。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。

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コピー再配置があるかどうかを検査できます。

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
```

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
```

この動的実行可能ファイルに対して ldd(1) を実行すると、次の結果が返されます。

```
$ 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
```

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することができるデータすべてをコピーする一方で、その割り当てスペースで許容できる量しか受け付けないということを知らせています。

位置独立の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。139 ページの「位置独立のコード」を参照してください。

-B symbolic オプションの使用

リンカーの -B symbolic 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、実行時リンカーそのものを作成するために設計されたという意味で、長い歴史があるといえます。

-B symbolic オプションを使用するときは、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必要があります。61 ページの「シンボル範囲の縮小」を参照してください。 -B symbolic を使用すると直感的にはわからない副産物ができることがあります。

シンボリックに結合されたシンボルが割り込まれた場合、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、同じ割り込み状態を作成します。147 ページの「コピー再配置」を参照してください。

注 – シンボリックに結合された共有オブジェクトは、.dynamic フラグ DF_SYMBOLIC で表されます。このタグは情報を提供するだけです。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ほか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。実行時リンカーは、共有オブジェクトをアプリケーションに結合しなくてはならないため、すべての大域関数結合を横取りすることができます。これらの結合は、.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、87 ページの「再配置が実行されるとき」を参照してください。

LD_PROFILE 環境変数には、プロファイル対象となる共有オブジェクトの名前を指定します。この環境変数を使用すると、単一の共有オブジェクトを解析できます。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 の使用が解析されます。

```
$ LD_PROFILE=libc.so.1  ls -l
```

次の例では、環境変数の設定は構成ファイルに記録されます。この設定によって、アプリケーションが libc を使用するたびに、解析情報が蓄積されます。

```
# crle  -e LD_PROFILE=libc.so.1
$ ls -l
$ make
$ ...
```

プロファイル処理が有効化されると、プロファイルデータファイルがまだ存在していない場合にはそれが作成されます。このファイルは、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を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せん。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) の上にも構築されます。

このプロファイリングメカニズムのもっとも強力な機能の 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 /lib/libc.so.1 /var/tmp/libc.so.1.profile
.....

granularity: each sample hit covers 4 byte(s) ....

                                called/total     parents
index  %time    self descendents  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              self    total
 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から、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ほか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。`fork(2)` を使用すると、この制限はなくなります。

◆ ◆ ◆ 第 5 章

アプリケーションバイナリインタフェースとバージョン管理

リンカーによって処理される ELF オブジェクトには、ほか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。

バージョン管理とは、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ます。また、この ABI 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についても説明します。これらの概念については、インタフェースと実装の変更を新しいリリースのオブジェクトにどのように組み込むかを示すモデルを使用して説明します。

この章では、動的実行可能プログラム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管理シナリオの共通セットは、付録 B「バージョン管理の手引き」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、オブジェクトの公開インタフェースを表します。リンク編集後にオブジェクトに残る大域シンボルの数は、公開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シンボル状態から引き出されます。これらの大域シンボルは、オブジェクト内の非公開インタフェースを表します。

オブジェクトの ABI を定義するには、まず、オブジェクトから公開して使用できるようにする大域シンボルを決定するべきです。これらの公開シンボルは、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、61 ページの「シンボル範囲の縮小」に説明されています。この公開インタフェースは、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ます。これらの定義は、オブジェクトの進化に合わせて新しいインタフェースを追加する際の基礎となります。

次の節は、この初期公開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、これらのインタフェースを適切に管理できるかを理解しておくべきです。

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。これらの変更は付加的です。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で残されます。
- 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ます。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、または動作が異なってきます。

次のリストは、共通のオブジェクト変更のいくつかを分類しています。

表 5–1 インタフェースの互換性の例

オブジェクトの変更	更新タイプ
シンボルの追加	互換性あり
シンボルの削除	互換性なし
「非」 varargs(3EXT) 関数への引数の追加	互換性なし
関数からの引数の削除	互換性なし
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	互換性なし
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ーを変更しない場合)	互換性あり
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ーを変更する場合)	互換性なし

注 – シンボルを追加すると、割り込みが原因で、互換性のない変更が生じる可能性があります。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場合があります。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、実際にはこのような互換性のない変更はめったにありません。

互換性のある変更は、生成されるオブジェクトの内部でバージョン定義を管理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管理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管理手法を使用すると、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。バージョン管理手法を使用すれば、実行時の正しいバージョン割り当てを検査することもできます。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。

内部バージョン管理

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。ただし、バージョン定義はほか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。

- バージョン定義を与えられた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ます。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、実行時に検査されます。
- 動的オブジェクトは、結合する共有オブジェクトのバージョン定義をリンク編集中に選択できます。このメカニズムを使用すると、開発者は、共有オブジェクト内のもっとも適したインタフェースまたは機能への、依存関係を制御することができます。

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、61 ページの「シンボル範囲の縮小」を参照してください。

バージョン定義は、バージョン名が mapfile 指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 指令とともに結合されて、定義済み公開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 foo1()
{
        (void) printf(_foo1);
}

$ cat data.c
const char * _foo1 = "string used by foo1()\n";

$ cat mapfile
SUNW_1.1 {                  # Release X
        global:
                foo1;
        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
```

シンボル foo1 は、共有オブジェクトの公開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。特殊な自動縮小指令「*」は、ほか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。この自動縮小指令については、54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。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開インタフェースは、内部バージョン定義 SUNW_1.1 に関連付けられた大域シンボル foo1 で構成されます。

バージョン定義または自動縮小指令によってオブジェクトが生成されると、基本バージョン定義も必ず作成されます。この基本バージョンは、作成されるオブジェクトの名前を使用して定義されます。この基本バージョンは、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、67 ページの「出力ファイルの生成」を参照してください。

オブジェクト内に含まれるバージョン定義は、-d オプションを付けた pvs(1) を使用して表示できます。

```
$ pvs -d libfoo.so.1
        libfoo.so.1;
        SUNW_1.1;
```

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 指令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、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にはオブジェクトの展開を記述できるため、このインタフェースを作成することは重要です。ユーザーはこれらのインタフェースを使って、結合先のインタフェースを検査して選択できます。これらの概念については、163 ページの「バージョン定義への結合」と167 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 指令を示しています。

```
$ 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
```

foo1 と foo2 は、いずれも共有オブジェクトの公開インタフェースの一部として定義されています。ただし、これらのシンボルはそれぞれ別のバージョン定義に割り当てられます。foo1 は、バージョン SUNW_1.1 に割り当てられます。foo2 は、バージョン SUNW_1.2 に割り当てられます。

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_;
                _etext;
        SUNW_1.1:
                foo1;
                SUNW_1.1;
        SUNW_1.2:               {SUNW_1.1}:
                foo2;
                SUNW_1.2
```

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、便利な手法です。この継承によって、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が削減されます。バージョン継承については、163 ページの「バージョン定義への結合」で詳しく説明します。

バージョン定義シンボルが作成され、バージョン定義に関連付けられます。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。このようなバージョン定義は空です。このバージョン定義には、大域インタフェースシンボルが関連付けられません。

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";
```

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};
```

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特定の実装詳細の存在を検査できます。アプリケーションは、必要とする実装詳細に関連付けられたバージョン定

義に結合できます。163 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();

void bar2()
{
        foo2();
}
```

これらの 2 つのシンボルは、2 つの新しい公開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。ただし、それぞれのインタフェースは、元の SUNW_1.2 インタフェースへの依存関係を表します。

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};
```

バージョン定義を使用して、実行時結合の要件を検査できます。また、バージョン定義を使用して、オブジェクトの作成中にオブジェクトの結合を制御することもできます。次の節では、これらのバージョン定義の使用方法について詳細に説明します。

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、ほか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、35 ページの「共有オブジェクトの処理」と 123 ページの「共有オブジェクト名の記録」を参照してください。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構築されたオブジェクトに記録されます。

次の例は、前の節のデータファイルを使用して、コンパイル時環境に適した共有オブジェクト libfoo.so.1 を生成しています。

```
$ 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:               {SUNW_1.1}:
                foo2;
                SUNW_1.2;
        SUNW_1.2.1 [WEAK]:      {SUNW_1.2}:
                SUNW_1.2.1;
        SUNW_1.3a:              {SUNW_1.2}:
                bar1;
                SUNW_1.3a;
        SUNW_1.3b:              {SUNW_1.2}:
                bar2;
                SUNW_1.3b
```

6つの公開インタフェースが、共有オブジェクト libfoo.so.1 によって提供されています。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、SUNW_1.3b) はエクスポートされたシンボル名を定義します。1つのインタフェース SUNW_1.2.1 は、オブジェクトに対する内部実装の変更を記述します。もう1つのインタフェース libfoo.so.1 は、いくつかの予約ラベルを定義します。libfoo.so.1 によって依存関係として作成される動的オブジェクトは、その動的オブジェクトが結合するインタフェースのバージョン名を記録します。

次の例では、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管理依存関係に関する情報は、-r オプションを付けた pvs(1) を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は、それぞれ大域シンボル foo1 と foo2 を提供しました。

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、記録が必要なのは 1 つの依存関係だけです。この継承によって、バージョン定義の依存関係が正規化されます。この正規化によって、オブジェクト内に保持されているバージョン情報の量は削減されます。また、この正規化によって実行時に必要なバージョン検査の処理も縮小されます。

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します。ウィークバージョン定義の依存関係は、別々に記録されることになります。

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。

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から、記録された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、-v オプションを付けた ldd(1) を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、依存関係 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管理要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、シンボル foo2 が存在しないときには、実行時に重大な再配置エラーが発生することになります。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、エラー状態がまったく生じないこともあります。89 ページの「再配置エラー」を参照してください。

バージョン定義依存関係によって、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がすぐに示されます。

たとえば、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、prog を実行するとします。この場合、ウィークでないバージョン定義要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ます。この場合、実行時エラー条件は生成されません。

```
$ 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(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。

```
$ ldd prog
        libfoo.so.1 =>   ./libfoo.so.1
        libfoo.so.1 (SUNW_1.2.1) =>          (version not found)
        ...........
```

実行時に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場合、依存関係のバージョン検査は暗黙のうちに無視されます。この方針は、非バージョン管理共有オブジェクトからバージョン管理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。ldd(1) は、バージョン要件の違いを表示するためにいつでも使用できます。

注 – 環境変数 LD_NOVERSION を使用すると、すべての実行時バージョン検査を抑制できます。

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(3C) によって取得されたオブジェクトのバージョン要件を検査するメカニズムとなるものです。dlopen(3C) 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、自動バージョン依存関係検査が行われません。このため、dlopen(3C) の呼び出し元が、バージョン管理要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(3C) 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。次の例では、dlopen(3C) を使用して共有オブジェクト libfoo.so.1 をプロセスに追加します。次に、インタフェース SUNW_1.2 が利用可能であることを確認します。

```
#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 | RTLD_FIRST))) == 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);
        }
        ....
```

注 – dlopen(3C) のフラグ RTLD_FIRST を使用すると、dlsym(3C) の検索が libfoo.so.1 に制限されます。

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要件は、「ファイル制御指令」によって制御できます。この指令は、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。ファイル制御指令には、次の構文を使用します。

name - *version* [*version* ...] [$ADDVERS=*version*];

- *name* — 共有オブジェクト依存関係の名前を表します。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。36 ページの「ライブラリの命名規約」を 参照してください。
- *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します。複数のバージョン定義を指定できます。
- $ADDVERS – 追加バージョン定義を記録できるようにします。

バージョン結合の制御は、次のような場合に役立ちます。

- 共有オブジェクトが一意の独立したバージョンを定義するとき。このバージョン管理は、異なる標準インタフェースを定義するときに使用できます。結合制御でオブジェクトを構築することによって、そのオブジェクトが特定のインタフェースだけに結合することを保証できます。
- 複数世代のソフトウェアリリースにまたがって、共有オブジェクトをバージョン管理するとき。結合制御でオブジェクトを構築することによって、以前のソフトウェアリリースで利用可能だったインタフェースだけに結合するように制限できます。したがって、最新リリースの共有オブジェクトを使用して構築したオブジェクトでも、古いリリースの共有オブジェクトを使用して実行できます。

次に、バージョン制御メカニズム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します。

アプリケーションは、次のバージョン制御 mapfile 指令を使用して、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、Release X 上で動作するアプリケーション prog を開発するとします。アプリケーションでは、Release X で使用可能なインタフェース以外は使用できません。シンボル bar を間違えて参照すると、アプリケーションは要求されるインタフェースに準拠しなくなります。リンカーはこの状態を未定義のシンボルエラーとして通知します。

```
$ 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 に対する要件を削除するか、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。

注– デフォルトでは、リンク編集の一部として検出された共有オブジェクト依存関係も、すべてのファイル制御指令に対して確認されます。環境変数 LD_NOVERSION を使用して、共有オブジェクト依存関係のバージョン検査を抑制します。

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作成されるバージョン依存関係に、追加のバージョン依存関係を記録するには、$ADDVERS ファイル制御指令を使用します。次の節では、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。

インタフェースの再定義

1 つのシナリオは、ISV 固有のインタフェースを公開標準インタフェースで使用します。

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。次の例では、このバージョン定義は 2 つの標準定義を継承するものとして表されています。

```
$ 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;
                STAND_A;
        STAND_B:
                foo2;
                STAND_B;
```

アプリケーション prog の唯一の要件がインタフェースシンボル foo1 である場合、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、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。以前のリリースでは、インタフェース foo1 が存在する場合でも、バージョン定義 STAND_A は存在しませんでした。

アプリケーション prog は、SUNW_1.1 に対する依存関係を作成することによって、その要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、真の依存関係の要件をカプセル化するのに十分です。この依存関係は古いリリースとの互換性も保ちます。

ウィークバージョンの結合

161 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。ウィークバージョンの存在が必要である場合、ウィークバージョン定義への明示的な依存関係を作成できます。オブジェクトを正しく機能させるためにバグ修正や性能の改善が重要な場合、このような依存関係の作成も重要になります。

上記の `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 `libfoo.so.1` に対して構築されている場合、バージョン定義 `SUNW_1.2.1` に対する弱い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します。実行時に使用される `libfoo.so.1` の実装にバージョン定義が見つからなくても、この依存関係によってアプリケーションが強制終了されることはありません。

ファイル制御指令 `$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。

アプリケーション `prog` は、次のファイル制御指令を使用して、`SUNW_1.2.1` インタフェースを実行時に使用できるという要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 foo1();

main()
{
        foo1();
}
$ cc -M mapfile -o prog prog.c -L. -R. -lfoo
$ pvs -r prog
        libfoo.so.1 (SUNW_1.2.1);
```

prog には、インタフェース STAND_A に対する明示的な依存関係があります。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 – 依存関係が少ない場合、リンカーの -u オプションを使用して、バージョン定義に明示的に結合できます。このオプションで、バージョン定義シンボルを参照します。ただし、シンボル参照は非選択的です。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できないことがあります。

バージョンの安定性

バージョン定義をオブジェクトの内部に結合することを説明するために、さまざまなモデルについて説明してきました。これらのモデルでは、インタフェースの要件を実行時に検査できます。この検査は、各バージョン定義がオブジェクトの使用期間内に変わらない場合にのみ有効です。

オブジェクトのバージョン定義を作成したら、ほかのオブジェクトが結合することができます。このバージョン定義は、オブジェクトの次のリリースでも存在している必要があります。バージョン名およびバージョンに関連するシンボルは両方とも変更しないでください。これらの要件を適用するために、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。数が一致しない場合には、インタフェースが突然不安定になることがあります。

再配置可能オブジェクト

ここまでの節で、バージョン情報を動的オブジェクトの内部に記録する方法を説明してきました。再配置可能オブジェクトは、同様の方法でバージョン管理情報を保持できます。ただし、こ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、オブジェクトに記録されます。これらの定義は、動的オブジェクトに記録されるバージョン定義と同じ形式で記録されます。ただしデフォルトにより、作成中の再配置可能オブジェクトに対するシンボル削減は実行されません。再配置可能オブジェクトが動的オブジェクトの生成に使用されると、バージョン情報に定義されているシンボル削減が再配置可能オブジェクトに適用されます。

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、61 ページの「シンボル範囲の縮小」を参照してください。

注 – リンカーの `-B reduce` オプションを使用すると、バージョン定義に定義されているシンボル削減を再配置可能オブジェクトに適用できます。

外部バージョン管理

共有オブジェクトへの実行時参照は、常にバージョン管理ファイル名を参照するべきです。通常、バージョン管理ファイル名は、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変更すると、その変更によって古いアプリケーションが破壊される可能性があります。このような場合は、新しい共有オブジェクトを新しいバージョン管理ファイル名によって配布するべきです。また、元のバージョン管理ファイル名も配布して、古いアプリケーションで必要なインタフェースを提供する必要があります。

一連のソフトウェアリリースに対してアプリケーションを構築しているときは、実行時環境内に共有オブジェクトを個別のバージョンファイル名で提供する必要があります。このようにすれば、アプリケーションを構築するときに基にしたインタフェースが、実行中に結合するアプリケーションで利用できることを保証できます。

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。

バージョン管理ファイル名の管理

リンク編集では、一般的にリンカーの `-l` オプションを使用して共有オブジェクトの依存関係を参照します。このオプションは、リンカーのライブラリ検索メカニズムを使用して接頭辞 `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、バージョン管理ファイル名として存在していなければなりません。2 つの命名規約に従う 2 つの異なる共有オブジェクトを維持するのではなく、2 つのファイル名間にファイルシステムリンクを作成します。

たとえば、シンボリックリンクを使用すれば、共有オブジェクト libfoo.so.1 をコンパイル環境で利用できるようにすることができます。コンパイルファイル名は、実行時ファイル名へのシンボリックリンクになります。

```
$ 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
```

シンボリックリンクまたはハードリンクを使用できます。ただし記述および診断目的としては、シンボリックリンクの方が有効です。

共有オブジェクト libfoo.so.1 は、実行時環境用に生成されています。シンボリックリンク libfoo.so 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。

```
$ cc -o prog main.o -L. -lfoo
```

リンカーは、シンボリックリンク 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わたって、libfoo.so の新しいバージョンをインタフェースを変更して配布できます。シンボリックリンクを変更することによって、適用可能なインタフェースを使用するよう、コンパイル環境を構築することができます。

```
$ 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
```

この例では、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。libfoo.so.1 と libfoo.so.2 の 2 つのバージョンは、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。

このシンボリックリンクのメカニズムを使用するだけでは、コンパイル環境での共有オブジェクトが実行時バージョン管理ファイル名と同期をとることができません。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、リンカー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、リンカーで表示されるファイル名はコンパイル環境のファイルです。

```
$ dump -Lv prog

prog:
 **** DYNAMIC SECTION INFORMATION ****
.dynamic:
[INDEX] Tag      Value
[1]     NEEDED   libfoo.so
.........
```

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 を検索します。prog は、このシンボリックリンクが指すすべてのファイルに結合されます。

正しい実行時名を依存関係として記録するには、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します。この名前は、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、共有オブジェクトの作成中に -h オプションを使用して与えることができます。

```
$ 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
.........
```

このシンボリックリンクと「soname」メカニズムは、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す。リンク編集中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。

同じプロセス内の複数の外部バージョン管理ファイル

外部的にバージョン管理された新しい共有オブジェクトを作成することは、大きな変更です。外部的にバージョン管理された一連の共有オブジェクトのメンバーを使用するすべてのプロセスの依存関係を、完全に理解している必要があります。

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。新しいインタフェース libfoo.so.2 を使

用するように、アプリケーションを再設計する可能性があります。ただし、アプリケーションが外部オブジェクト libISV.so.1 を使用することは変更しない可能性があります。実行時に読み込まれる libfoo.so 実装の可視性のスコープに応じて、主要なバージョンのファイルが両方とも実行プロセスに含まれます。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です。つまり、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。

インタフェースに互換性のない変更を行うことは避けるようにしてください。互換性のない変更は、インタフェース定義およびインタフェース定義を参照するすべてのオブジェクトを完全に制御できる場合に限って検討することをお勧めします。

◆ ◆ ◆ 第 6 章

サポートインタフェース

リンカーには、リンカーおよび実行時リンカーの処理の監視および変更を可能に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。

- 「ld-サポート」 – 177 ページの「リンカーのサポートインタフェース」
- 「rtld-監査」 – 185 ページの「実行時リンカーの監査インタフェース」
- 「rtld-デバッガ」 – 197 ページの「実行時リンカーのデバッガインタフェース」

リンカーのサポートインタフェース

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。

この節では、「ld-サポート」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、入力ファイル検査用、および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もある程度サポートされています。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、リンカーおよび `make(1S)` ユーティリティーです。リンカーは、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デバッグ情報を処理します。`make` ユーティリティーは、このインタフェースを使用して状態情報を保存します。

「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ます。ライブラリで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には、`elf(3ELF)` 構造とファイル形式に精通している必要があります。

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:support.so.2 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。複数の -S オプションを指定できます。

```
$ LD_OPTIONS="-S./support.so.1 -Ssupport.so.2" 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、dlopen(3C) を使用して、各サポートライブラリを指定された順序で開きます。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。次に、各サポートライブラリ内で、dlsym(3C) を使用してサポートインタフェースルーチンの検索が実行されます。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。

サポートライブラリは、32 ビットまたは 64 ビットのいずれの場合でも、呼び出されるリンカーの ELF クラスと一致している必要があります。詳細は、178 ページの「32 ビットおよび 64 ビット環境」を参照してください。

注 – デフォルトでは、リンカーは Solaris OS サポートライブラリ libldstab.so.1 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。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があります。その場合は、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 を明示的に追加します。

---

32 ビットおよび 64 ビット環境

27 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ます。また、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、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。

SGS_SUPPORT 環境変数は、接尾辞 _32 または _64 を使用して指定でき、また、リンカーオプション -z ld32 および -z ld64 を使用して -S オプション要件を定義できます。これらの各定義は、対応する 32 ビットまたは 64 ビットのリンカーによってのみ解釈されます。このため、リンカーの種類が不明な場合に、両方の種類のサポートライブラリを指定できます。

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、ヘッダーファイル 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 を使用して確認できます。libelf の詳細は、elf(3ELF) のマニュアルページを参照してください。次のインタフェース関数が「ld-サポート」インタフェースにより提供されます。各インタフェース関数は、使用順序に従って記載されています。

ld_version()
: この関数は、リンカーとサポートライブラリとの間の初期ハンドシェークを提供します。

```
uint_t ld_version(uint_t version);
```

リンカーは、リンカーがサポート可能な最新バージョンの「ld-サポート」インタフェースを使用して、このインタフェースを呼び出します。サポートライブラリは、このバージョンが使用するのに十分かどうかを確認できます。次に、サポートライブラリは、サポートライブラリが使用する予定のバージョンを返すことができます。通常、このバージョンは LD_SUP_VCURRENT です。

サポートライブラリがこのインタフェースを提供しない場合、初期サポートレベルは LD_SUP_VERSION1 と見なされます。

サポートライブラリがゼロのバージョン、またはリンカーがサポートする ld-サポートインタフェースよりも大きい値を返す場合、サポートライブラリは使用されません。

ld_start()
: この関数は、リンカーコマンド行の初期妥当性検査のあとに呼び出されます。この関数は、入力ファイル処理の開始を示します。

```
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);
```

*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 のいずれかで、これは sys/elf.h に定義されています。*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、これは通常、/usr/ccs/bin/ld です。

ld_open()
この関数は、リンク編集への各入力ファイルに対して呼び出されます。バージョン LD_SUP_VERSION3 で追加されたこの関数は、ld_file() 関数よりも高い柔軟性を備えています。サポートライブラリはこの関数を使用することで、ファイル記述子、ELF 記述子、およびそれらに関連付けられたファイル名を置き換えることができます。この関数は次のシナリオで使用できます。

- 既存の ELF ファイルへの新しいセクションの追加。この場合、元の ELF 記述子を、ELF ファイルの更新を可能とする記述子で置き換えるようにしてください。elf_begin(3ELF) の ELF_C_RDWR 引数を参照してください。
- 入力ファイルの全体を別のファイルで置き換え可能。この場合、元のファイル記述子と ELF 記述子を、新しいファイルに関連付けられた記述子で置き換えるようにしてください。

どちらのシナリオの場合も、パス名とファイル名を別の名前で置き換えることができ、そうした場合、それは入力ファイルが変更されたことを示します。

```
void ld_open(const char ** pname, const char ** fname, int * fd,
    int flags, Elf ** elf, Elf * ref, size_t off, Elf_Kind kind);
void ld_open64(const char ** pname, const char ** fname, int * fd,
    int flags, Elf ** elf, Elf * ref, size_t off, Elf_Kind kind);
```

pname は、処理されようとしている入力ファイルのパス名です。*fname* は、処理されようとしている入力ファイルのファイル名です。*fname* は通常、*pname* のベース名になります。*pname* と *fname* はどちらも、サポートライブラリから変更できます。

fd は、入力ファイルのファイル記述子です。サポートライブラリからこの記述子を閉じ、新しいファイル記述子をリンカーに返せます。値が -1 のファイル記述子を返せば、そのファイルを無視すべきであることを示せます。

注 – ld_open() に渡された *fd* には、リンカーが ld_open() でファイル記述子を閉じることができないと、値 -1 が設定されます。この状況が発生するもっとも一般的な理由は、アーカイブメンバーの処理の場合です。値 -1 が ld_open() に渡されると、記述子を閉じることができなくなり、代替の記述子がサポートライブラリから返されなくなります。

---

flags フィールドは、リンカーによるファイルの取得方法を示します。このフィールドには、次の定義の 1 つまたは複数を指定できます。

- LD_SUP_DERIVED – ファイル名がコマンド行に明示的に指定されませんでした。ファイルは、-l を展開して派生されました。あるいは、ファイルは、抽出されたアーカイブメンバーです。
- LD_SUP_EXTRACTED – ファイルはアーカイブから抽出されました。
- LD_SUP_INHERITED – ファイルはコマンド行の共有オブジェクトの依存関係として取得されました。

flags 値が指定されていない場合は、入力ファイルがコマンド行に明示的に指定されました。

elf は、入力ファイルの ELF 記述子です。サポートライブラリからこの記述子を閉じ、新しい ELF 記述子をリンカーに返せます。値が 0 の ELF 記述子を返すことができ、そのファイルを無視すべきであることを示せます。*elf* 記述子がアーカイブライブラリのメンバーに関連付けられている場合、*ref* 記述子はその背後のアーカイブファイルの ELF 記述子になります。*off* は、アーカイブファイル内のアーカイブメンバーのオフセットを表します。

kind は入力ファイルのタイプを示し、libelf.h に定義されているように ELF_K_AR または ELF_K_ELF のいずれかになります。

ld_file()
この関数は、リンク編集への各入力ファイルに対して呼び出されます。この関数は、ファイルデータの処理が実行される前に呼び出されます。

```
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);
```

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、libelf.h に定義されているように ELF_K_AR または ELF_K_ELF のいずれかになります。*flags* フィールドは、リンカーによるファイルの取得方法を示します。このフィールドには、ld_open() の *flags* フィールドと同じ定義を含めることができます。

- LD_SUP_DERIVED – ファイル名がコマンド行に明示的に指定されませんでした。ファイルは、-l を展開して派生されました。あるいは、ファイルは、抽出されたアーカイブメンバーです。
- LD_SUP_EXTRACTED – ファイルはアーカイブから抽出されました。
- LD_SUP_INHERITED – ファイルはコマンド行の共有オブジェクトの依存関係として取得されました。

flags 値が指定されていない場合は、入力ファイルがコマンド行に明示的に指定されました。

elf は、入力ファイルの ELF 記述子です。

`ld_input_section()`
この関数は、入力ファイルの各セクションに対して呼び出されます。この関数は、リンカーがそのセクションを出力ファイルに送信することを決定する前に呼び出されます。これは、バージョン `LD_SUP_VERSION2` で追加された関数です。これは、出力ファイルに寄与するセクションに対してのみ呼び出される、`ld_section()` 処理とは異なります。

```
void ld_input_section(const char * name, Elf32_Shdr ** shdr,
        Elf32_Word sndx, Elf_Data * data, Elf * elf, unit_t flags);

void ld_input_section64(const char * name, Elf64_Shdr ** shdr,
        Elf64_Word sndx, Elf_Data * data, Elf * elf, uint_t flags);
```

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ー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ー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。*flags* は、将来の使用のために予約されています。

セクションヘッダーの再割り当ておよび **shdr* への代入によるセクションヘッダーの変更は許されています。リンカーは、`ld_input_section()` から戻った後で、**shdr* が指し示すセクションヘッダー情報を使用して、セクションを処理します。

データを再割り当てし、`Elf_Data` バッファーの `d_buf` ポインタに代入してデータを変更できます。データを変更する場合、`Elf_Data` バッファーの `d_size` 要素を正しく設定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実際に削除されます。

flags フィールドは、初期値にゼロが設定される `uint_t` データフィールドを指します。フラグは、将来のアップデートでリンカーやサポートライブラリが割り当てできるように提供はされていますが、現在のところは割り当てられていません。

`ld_section()`
この関数は、出力ファイルに送信される入力ファイルのセクションごとに呼び出されます。この関数は、セクションデータの処理が実行される前に呼び出されます。

```
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);
```

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ー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ー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。

データを再割り当てし、Elf_Data バッファーの d_buf ポインタに代入してデータを変更できます。データを変更する場合、Elf_Data バッファーの d_size 要素を正しく設定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実際に削除されます。

注 – 出力ファイルから取り除かれるセクションは、ld_section() に報告されません。セクションは、リンカーの -s オプションを使って取り除かれます。セクションは、SHT_SUNW_COMDAT 処理や SHF_EXCLUDE の識別によって破棄されます。244 ページの「「COMDAT」セクション」と表 7–8 を参照してください。

---

ld_input_done()
: この関数は、入力ファイルの処理が完了してから、出力ファイルの配置が実行されるまでに呼び出されます。これは LD_SUP_VERSION2 で追加された関数です。

```
void ld_input_done(uint_t * flags);
```

flags フィールドは、初期値にゼロが設定される uint_t データフィールドを指します。フラグは、将来のアップデートでリンカーやサポートライブラリが割り当てできるように提供はされていますが、現在のところは割り当てられていません。

ld_atexit()
: この関数は、リンク編集の完了時に呼び出されます。

```
void ld_atexit(int status);

void ld_atexit64(int status);
```

status は、リンカーによって返される exit(2) コードであり、stdlib.h に定義されているように、EXIT_FAILURE または EXIT_SUCCESS のいずれかになります。

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を出力するサポートライブラリを作成します。

```
$ cat support.c
#include        <link.h>
#include        <stdio.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)
{
        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,
                    "", (long)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。このサポートライブラリを構築するには次のようにします。

```
$ 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 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
   .......
file: main.o
   section [1]: .shstrtab
   section [2]: .text
   .......
file: ./libfoo.a
  file: ./libfoo.a(foo.o)
     section [1]: .shstrtab
     section [2]: .text
     .......
file: /lib/libc.so
file: /opt/COMPILER/crtn.o
   section [1]: .shstrtab
   section [2]: .text
   .......
```

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

「rtld-監査」インタフェースを使用すると、プロセスがそれ自身に関する実行時リンク情報にアクセスできます。151 ページの「共有オブジェクトのプロファイリング」では、このメカニズムの使用例として、共有オブジェクトの実行時プロファイルを示します。

「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。監査ライブラリはこれらのインタフェースを使って、次の情報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ます。
- 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ます。

- 関数呼び出しとその戻り値の監査を可能にするための、プロシージャーのリンカーテーブルエントリによって提供される遅延結合メカニズムの利用。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ます。312 ページの「プロシージャーのリンクテーブル (プロセッサ固有)」を参照してください。

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。しかし、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存在します。このため、通常、事前に読み込まれた共有オブジェクトの実装は制限されるか、複雑になります。「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。この名前空間により、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ます。リンクマップ構造は、`/usr/include/sys/link.h` に定義されています。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シンボル検索メカニズムは、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、プロセスシンボル解決用の「名前空間」を提供します。

実行時リンカーも、リンクマップによって記述されます。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で管理されます。この結果、実行時リンカーが固有の名前空間内に常駐することになるため、アプリケーションが実行時リンカー内のサービスに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 `libc.so.1` または `libdl.so.1` を介して、実行時リンカーの公開サービスを呼び出すことができるだけです。

「rtld-監査」インタフェースは、監査ライブラリが保持される各自のリンクマップリストを使用します。このため、監査ライブラリは、アプリケーションのシンボル結合要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、`dlmopen(3C)` によって実行できます。`dlmopen(3C)` を `RTLD_NOLOAD` フラグとともに使用すると、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 */
```

すべての「rtld-監査」サポートライブラリには、固有の新しい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

監査ライブラリはほかの共有オブジェクトと同様に構築されます。ただし、プロセス内の監査ライブラリに固有の名前空間には、いくつかの注意が必要です。

- ライブラリは、すべての依存関係の要件を提供しなければならない。
- ライブラリは、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を使用できない。

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する必要があります。51 ページの「共有オブジェクト出力ファイルの生成」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` によってシンボル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要件を満たし、もう 1 つは監査リンクマップリストの結合要件を満たします。

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。

システムインタフェースの中には、自らがプロセス内部の実装の唯一のインスタンスであると想定しているものがあります。このような実装の例として、シグナルおよび `malloc(3C)` があります。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 – 監査ライブラリは、`mapmalloc(3MALLOC)` を使用してメモリー割り当てを行うことができます。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-監査」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。

- 「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ります。この方法によって使用可能になる監査ライブラリには、監査用に識別される動的オブジェクトに関する情報が指定されます。
- 「大域」監査は、環境変数 `LD_AUDIT` を使用することにより有効になります。また、ローカル監査動的エントリと `-z globalaudit` オプションを組み合わせることによっても、アプリケーションの大域監査を有効にすることができます。これら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指定されます。

それぞれの呼び出し方法は、dlopen(3C) によって読み込まれる共有オブジェクトをコロン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(3C) 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-監査」インタフェースを使用すると、複数の監査ライブラリを指定することができます。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。これらの結合を変更すると、後に続く監査ライブラリで予期しない結果が生じる場合があります。

安全なアプリケーションは、トラストディレクトリからだけ監査ライブラリを取得できます。デフォルトでは、32 ビットオブジェクトの実行時リンカーが認識できるトラストディレクトリは、/lib/secure と /usr/lib/secure だけです。64 ビットオブジェクトの場合、トラストディレクトリは /lib/secure/64 と /usr/lib/secure/64 です。

注 – 環境変数 LD_NOAUDIT をヌル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。

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 または -P を使用して作成された場合に確立できます。たとえば、監査ライブラリ audit.so.1 を使用して libfoo.so.1 を監査するには、リンク編集時に -p オプションを使用して、この要求を記録します。

```
$ cc -G -o libfoo.so.1 -Wl,-paudit.so.1 -K pic 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ます。次に、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。

このメカニズムだけでは、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
```

このメカニズムで監査が有効になると、アプリケーションのすべての明示的な依存関係に関する情報が監査ライブラリに渡されます。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
```

大域監査の記録

大域監査の要件は、環境変数 LD_AUDIT を設定することによって確立できます。たとえば、この環境変数を使えば、アプリケーション main とそのプロセスのすべての依存関係を、監査ライブラリ audit.so.1 を使って監査できます。

```
$ LD_AUDIT=audit.so.1 main
```

また、-z globalaudit オプションを指定することで、アプリケーション内のローカル監査を記録することによる大域監査を実現できます。たとえば、大域監査が有効になるようにアプリケーション main を構築するには、リンカーの -P オプションと -z globalaudit オプションを使用します。

```
$ cc -o main main.c -Wl,-Paudit.so.1 -z globalaudit
$ dump -Lv main | fgrep AUDIT
[5]    DEPAUDIT   audit.so.1
[26]   FLAGS_1    [ GLOBAL_AUDITING ]
```

監査がこれらのメカニズムのどちらで有効化された場合も、プロセスの「すべて」の動的オブジェクトに関する情報が監査ライブラリに渡されます。

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います。これらの関数は使用順序に従って記載されています。

注 – アーキテクチャー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では、説明を簡潔にするため、省略して一般名を使用します。たとえば、la_symbind32() および la_symbind64() は la_symbind() で表します。

---

la_version()
: この関数は、実行時リンカーと監査ライブラリの間に初期ハンドシェークを提供します。監査ライブラリが読み込まれるためには、このインタフェースが提供されている必要があります。

```
uint_t la_version(uint_t version);
```

実行時リンカーは、実行時リンカーがサポート可能な最上位バージョンの「rtld-監査」インタフェースによって、このインタフェースを呼び出します。監査ライブラリは、このバージョンが使用するのに十分かどうかを確認して、監査ライブラリが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、あるいは、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なバージョンを返す場合、監査ライブラリは破棄されます。

la_activity()
この関数は、リンクマップアクティビティーが行われていることを監査プログラムに知らせます。

```
void la_activity(uintptr_t * cookie, uint_t flags);
```

cookie は、リンクマップ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ーを指します。

- LA_ACT_ADD – リンクマップリストにオブジェクトが追加される。
- LA_ACT_DELETE – リンクマップ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ーが完了した。

la_objsearch()
この関数は、オブジェクトの検索を実行することを監査プログラムに知らせます。

```
char * la_objsearch(const char * name, uintptr_t * cookie, uint_t flags);
```

name は、検索中の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 *name* の出所および作成を示します。

- LA_SER_ORIG – 初期検索名。通常は、DT_NEEDED エントリとして記録されたファイル名、あるいは dlopen(3C) に与えられた引数を指します。
- LA_SER_LIBPATH – パス名が LD_LIBRARY_PATH コンポーネントから作成されている。
- LA_SER_RUNPATH – パス名が「実行パス」コンポーネントから作成されている。
- LA_SER_DEFAULT – パス名がデフォルトの検索パスコンポーネントから作成されている。
- LA_SER_CONFIG – パスコンポーネントの出所が構成ファイルである。crle(1) のマニュアルページを参照してください。
- LA_SER_SECURE – パスコンポーネントがセキュリティー保護されたオブジェクトに固有である。

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ます。値 0 は、このパスが無視されることを示しています。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()`
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と呼び出されます。

```
uint_t la_objopen(Link_map * lmp, Lmid_t lmid, uintptr_t * cookie);
```

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。*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。この識別子は、監査ライブラリによって、オブジェクトをほかの「rtld-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。

`la_objopen()` 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。

これらの値により、監査者は `la_symbind()` で監視するオブジェクトを選択できます。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。

たとえば、監査者は、`libfoo.so` から `libbar.so` への結合を監視できます。`libfoo.so()` の `la_objopen` は、 `LA_FLG_BINDFROM` を返します。`libbar.so` の `la_objopen()` は、`LA_FLG_BINDTO` を返します。

監査者は、`libfoo.so` と `libbar.so` 間のすべての結合を監視できます。両方のオブジェクトの `la_objopen()` は、`LA_FLG_BINDFROM` と `LA_FLG_BINDTO` を返します。

監査者は、`libbar.so` へのすべての結合も監視できます。`libbar.so()` の `la_objopen` は、`LA_FLG_BINDTO` を返します。すべての `la_objopen()` 呼び出しは、`LA_FLG_BINDFROM` を返します。

`la_objfilter()`
この関数は、フィルタが新しい「フィルティー」を読み込むと呼び出されます。127 ページの「フィルタとしての共有オブジェクト」を参照してください。

```
int la_objfilter(uintptr_t * fltrcook, const char * fltestr,
        uintptr_t * fltecook, uint_t flags);
```

fltrcook は、フィルタを特定します。*fltestr* は、フィルティー文字列を指します。*fltecook* は、フィルティーを特定します。*flags* は、現在使用されていません。`la_objfilter()` は、フィルタとフィルティーの `la_objopen()` の後に呼び出されます。

戻り値 0 は、このフィルティーが無視されることを示しています。フィルタの使用を監視する監査ライブラリは、0 以外の値を返します。

`la_preinit()`
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あとで、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。

```
void la_preinit(uintptr_t * cookie);
```

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()`
この関数は、`la_objopen()` によって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。

```
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);
```

sym は構築されたシンボル構造であり、 `sym->st_value` は結合されたシンボル定義のアドレスを示します。`/usr/include/sys/elf.h` を参照してください。`la_symbind32()` は、`sym->st_name` を調整して実際のシンボル名を指すようにしています。`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シンボルテーブル内のシンボルインデックスを示します。*refcook* は、このシンボルへの参照を行うオブジェクトを特定します。この識別子は、`LA_FLG_BINDFROM()` を返した `la_objopen` 関数に渡されたものと同じです。*defcook* は、このシンボルを定義するオブジェクトを特定します。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できるデータ項目を指します。このデータ項目を使用すると、プロシージャーのリンクテーブルエントリの連続監査も変更できます。この値は、`/usr/include/link.h` に定義されたシンボル結合フラグのマスクです。

次のフラグが `la_symbind()` に提供される場合もあります。

- `LA_SYMB_DLSYM` – dlsym(3C) を呼び出した結果、シンボル結合が発生した。
- `LA_SYMB_ALTVALUE (LAV_VERSION2)` – `la_symbind()` への以前の呼び出しによって、シンボル値に対して代替値が返された。

`la_pltenter()` または `la_pltexit()` 関数が存在する場合、これらの関数は、プロシージャーリンクテーブルエントリの `la_symbind()` の後に呼び出されます。これ

らの関数は、シンボルが参照されるたびに呼び出されます。詳細は、196 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。

次のフラグは、デフォルトの動作を変更するために `la_symbind()` から提供されます。これらのフラグは、*flags* 引数が指す値とのビット単位の OR 演算として適用されます。

- `LA_SYMB_NOPLTENTER` – このシンボルに対して `la_pltenter()` 関数は呼び出さない。
- `LA_SYMB_NOPLTEXIT` – このシンボルに対して `la_pltexit()` 関数は呼び出さない。

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、制御は結合シンボル定義に渡されます。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()` のみに適用可能であり、処理されるシンボルの名前を含みます。この名前は、32 ビットインタフェースの `sym->st_name` フィールドから得られます。

`la_pltenter()`
これらの関数はシステムに固有です。これらの関数は、結合通知のタグが付いた 2 つのオブジェクト間でプロシージャーのリンクテーブルエントリが呼び出されるときに呼び出されます。

```
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);
uintptr_t la_amd64_pltenter(Elf64_Sym * sym, uint_t ndx,
        uintptr_t * refcook, uintptr_t * defcook,
        La_amd64_regs * regs, uint_t * flags, const char * sym_name);
```

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。

`la_sparcv8_pltenter()` と `la_sparcv9_pltenter()` では、*regs* は out レジスタを指します。`la_i86_pltenter()` では、 *regs* は stack および frame レジスタを指します。`la_amd64_pltenter()` では、 *regs* は stack および frame レジスタ、および整数引数の受け渡しに使用されるレジスタを指します。*regs* は `/usr/include/link.h` に定義されています。

flags は、結合に関する情報を伝達できるデータ項目を指します。このデータ項目を使用すると、プロシージャーリンクテーブルのエントリの連続監査を変更できます。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。

次のフラグは、現在の監査動作を変更するために `la_pltenter()` から提供できます。これらのフラグは、*flags* 引数が指す値とのビット単位の OR 演算として適用されます。

- `LA_SYMB_NOPLTENTER` – `la_pltenter()` は、このシンボルでは再び呼び出されることはない。
- `LA_SYMB_NOPLTEXIT` – `la_pltexit()` は、このシンボル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、制御は結合シンボル定義に渡されます。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()`
この関数は、結合通知のタグが付いた 2 つのオブジェクト間でプロシージャーのリンクテーブルエントリが返されるときに呼び出されます。この関数は、制御が呼び出し側に到達する前に呼び出されます。

```
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);
```

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。*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 – `la_pltexit()` は実験段階のインタフェースです。詳細は、196 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()`
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。

```
uint_t la_objclose(uintptr_t * cookie);
```

cookie はオブジェクトを特定するもので、以前の `la_objopen()` から取得されています。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_objopen(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: /lib/libc.so.1 loaded
file: /lib/libm.so.2 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シージャー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 パッケージに組み込まれています。perfcnt と symbindrep はサンプルプログラムです。これらのアプリケーションは、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

「rtld-監査」 実装には制限があります。監査ライブラリを設計するときは、これらの制限をよく理解するようにしてください。

アプリケーションコードの実行

オブジェクトがプロセスに追加されると、監査ライブラリは情報を受け取ります。監査ライブラリがこのような情報を受け取るときに、監視するオブジェクトが実行できる状態でない場合があります。たとえば、監査プログラムは、読み込むオブジェクトのための la_objopen() 呼び出しを受け取ることができます。ただし、そのオブジェクト内のコードを実行するには、その前にオブジェクトの依存関係を読み込んで再配置する必要があります。監査ライブラリは、dlopen(3C) を使用してハンドルを取得して、読み込んだオブジェクトを検査しなければならない場合があります。このハンドルは、dlsym(3C) を使用してインタフェースを検索するために使用できます。ただし、この方法で取得したインタフェースは、そのオブジェクトの初期化が完了したことがわかるまで、呼び出さないようにしてください。

la_pltexit() の使用

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、この要件は問題になりません。この場合、目的の関数に制御を渡す前に、余分なスタックを整理できます。

これらの制限が原因で、la_pltexit() は、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() ルーチンの使用は避けてください。

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ガインタフェース

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、デバッガが解析するアプリケーションから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、ほか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「rtld-デバッガ」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のアーキテクチャーは、`libc_db(3LIB)` で使用されるモデルに準拠します。

「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ります。
- 「制御」プロセスは、「rtld-デバッガ」インタフェースライブラリとリンクし、そのインタフェースを使用してターゲットプロセスの動的側面を検査します。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ます。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ます。

「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、もっと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、次のアクティビティーを有効にします。

- 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。
- プロシージャー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。「rtld-デバッガ」インタフェースは、この対話によって次の処理を実行できます。

- 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ます。大半のデバッガは、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。これらのルーチンについては、208 ページの「デバッガインポートインタフェース」を参照してください。

「rtld-デバッガ」インタフェースは、「rtld-デバッガ」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。

librtld_db.so.1、制御プロセス (デバッガ)、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、次の図に示します。

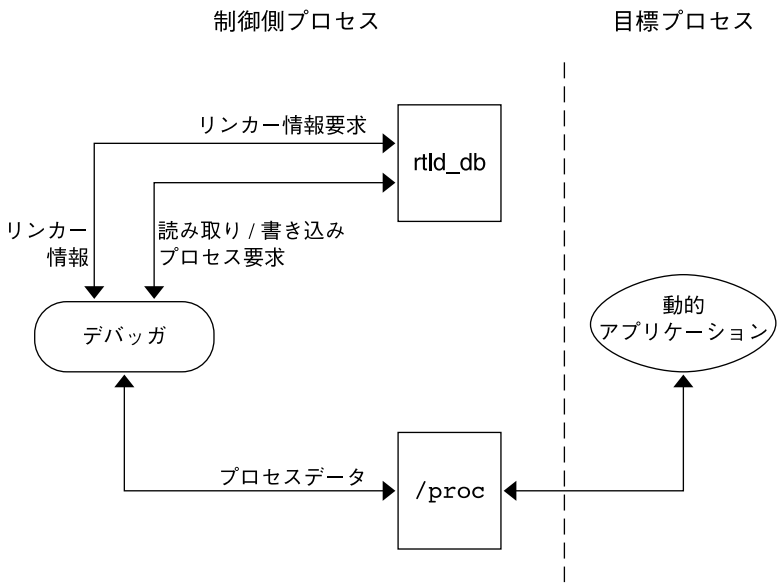


図 6–1 「rtld-デバッガ」の情報の流れ

注 – 「rtld-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。「rtld-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。

「rtld-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/librtld_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、およびすべての librtld_db.so.1 エクスポートインタフェースの必須呼び出しシーケンスを示します。次のセクションでは、「rtld-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入手することができます。

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可能な不透明なハンドルを提供します。エージェントは、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との間の通信メカニズムも提供します。「rtld-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
: 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
: 「rtld-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。

デバッガ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さまざまなインタフェースについて説明します。機能グループごとに分けて説明します。

エージェント操作インタフェース

rd_init()
: この関数は、「rtld-デバッガ」バージョン要件を確立します。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。

```
rd_err_e rd_init(int version);
```

Solaris 8 10/00 リリース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。詳細は、201 ページの「読み込み可能オブジェクトの走査」の rl_flags、 rl_bend および rl_dynamic フィールドを参照してください。

Solaris 8 01/01 リリース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。詳細は、205 ページの「プロシージャーのリンクテーブルのスキップ」の pi_baddr および pi_flags フィールドを参照してください。

制御プロセスのバージョン要件が使用可能な「rtld-デバッガ」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()
: 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。

```
rd_agent_t * rd_new(struct ps_prochandle * php);
```

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インポートされたインタフェースで使用されるものであり、「rtld-デバッガ」インタフェースに対して不透明です。

rd_reset()
: 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。

```
rd_err_e rd_reset(struct rd_agent * rdap);
```

この関数は、ターゲットプロセスが再起動されると呼び出されます。

rd_delete()
: 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放します。

```
void rd_delete(struct rd_agent * rdap);
```

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()`
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。

```
char * rd_errstr(rd_err_e rderr);
```

`rd_log()`
この関数は、ログ記録をオン (1) またはオフ (0) にします。

```
void rd_log(const int onoff);
```

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、`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ベースアドレス。

`rl_data_base`
動的オブジェクトのデータセグメントのベースアドレス。

`rl_lmident`
リンクマップ識別子 (186 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標準フィルタの場合は、「フィルティー」の名前を指定します。

`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ません。

`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ベースアドレス (207 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ベースアドレス (207 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 7–32 を参照) のエントリへの参照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。

`rd_loadobj_iter()` 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のリンクマップリストの情報にアクセスします。

`rd_loadobj_iter()`
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。

```
typedef int rl_iter_f(const rd_loadobj_t *, void *);

rd_err_e rd_loadobj_iter(rd_agent_t * rap, rl_iter_f * cb,
        void * clnt_data);
```

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` 構造へのポインタによって返されます。

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する。
- 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

制御プロセスは、実行時リンカーの適用範囲内で発生する特定のイベントを追跡できます。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します。

`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、基本実行可能プログラムに制御を渡します。

`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ます。

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;
```

イベントを追跡する関数を次に示します。

`rd_event_enable()`
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。

```
rd_err_e rd_event_enable(struct rd_agent * rdap, int onoff);
```

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()`
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。

```
rd_err_e rd_event_addr(rd_agent_t * rdap, rd_event_e event,
        rd_notify_t * notify);
```

イベントの種類によっては、制御プロセスの通知は、`notify->u.syscallno` で特定される害のない簡単なシステム呼び出しの呼び出しや、`notify->u.bptaddr` で指定されるアドレスでのブレークポイントの実行で行われます。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 の値を次に示します。

RD_NOSTATE
  使用可能な追加状態情報はありません。

RD_CONSISTANT
  リンクマップは安定した状態にあってテスト可能です。

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ではありません。リンクマップは、RD_CONSISTANT 状態に達するまでテストできません。

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ではありません。リンクマップは、RD_CONSISTANT 状態に達するまでテストできません。

rd_event_getmsg() 関数を使用して、このイベント状態情報を取得します。

rd_event_getmsg()
  この関数は、イベントに関する追加情報を提供します。

```
rd_err_e rd_event_getmsg(struct rd_agent * rdap, rd_event_msg_t * msg);
```

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

プロシージャーのリンクテーブルのスキップ

「rtld-デバッガ」インタフェースは、制御プロセスが、プロシージャーのリンクのテーブルエントリをスキップオーバーする機能を提供します。デバッガなどの制御プロセスが、関数に介入するようにとの要求をはじめて受けると、プロシージャーのリンクテーブル処理は、制御を実行時リンカーに渡して関数定義を検索します。

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ー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、プロシージャー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ます。

ターゲットプロセスは、プロシージャーのリンクのテーブルエントリに介入すると、rd_plt_resolution() インタフェースを呼び出します。

rd_plt_resolution()
: この関数は、現在のプロシージャー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。

```
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);
```

pc は、プロシージャー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* はプロシージャーのリンクテーブルのベース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ーがプロシージャー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、rtld_db.h 内の次のデータ構造に定義された、プロシージャー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
```

rd_plt_info_t 構造体の要素を次に示します。

pi_skip_method
: プロシージャー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します。rd_skip_e 値内の 1 つに設定されます。

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します。

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します。

pi_baddr
RD_VERSION3 で追加された、プロシージャー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) 宛先アドレスを示します。

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) プロシージャーのリンクエントリを示します。

次のシナリオは rd_plt_info_t 戻り値から考えられます。

- このプロシージャーのリンクテーブルによる最初の呼び出しは、実行時リンカーによって解決する必要があります。この場合、rd_plt_info_t には次のものが含まれます。

  {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、プロシージャーのリンクのテーブルエントリ処理は終了します。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ー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。
- このプロシージャーのリンクテーブル全体で Nth 番目。rd_plt_info_t には、次のものが含まれます。

  {RD_RESOLVE_STEP, M, 0, <BoundAddr>, RD_FLG_PI_PLTBOUND}

  プロシージャー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。プロシージャーのリンクのテーブルエントリと結合しているアドレスは、<BoundAddr> で、RD_FLG_PI_PLTBOUND ビットはフラグフィールドに設定されています。

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、もっと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()`
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。

```
rd_err_e rd_objpad_enable(struct rd_agent * rdap, size_t padsize);
```

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)` に対し `PROT_NONE` アクセス権と `MAP_NORESERVE` フラグを指定して、メモリー割り当てとして予約されます。実際には、ターゲットプロセスの仮想アドレス空間の、読み込み済みオブジェクトに隣接する領域が予約されます。これらの領域は、制御プロセスによってあとで使用できます。

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)` に `MAP_NORESERVE` を指定して `/dev/zero` から取得したメモリー割り当ての予約は、`proc(1)` 機能や `rd_loadobj_t` で提供されるリンクマップ情報を参照することで得られます。

デバッガ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、`rdb` デモデバッガにあります。「rtld-デバッガ」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-デバッガ」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()`
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。

```
ps_err_e ps_pauxv(const struct ps_prochandle * ph, auxv_t ** aux);
```

`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()`
この関数は、ターゲットプロセスからデータを読み取ります。

```
ps_err_e ps_pread(const struct ps_prochandle * ph, paddr_t addr,
        char * buf, int size);
```

ターゲットプロセス内のアドレス *addr* から、*size* バイトが *buf* にコピーされます。

`ps_pwrite()`
この関数は、ターゲットプロセスにデータを書き込みます。

```
ps_err_e ps_pwrite(const struct ps_prochandle * ph, paddr_t addr,
        char * buf, int size);
```

buf から *size* バイトが、ターゲットプロセスのアドレス *addr* にコピーされます。

`ps_plog()`
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。

```
void ps_plog(const char * fmt, ...);
```

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ます。`ps_plog()` の引数は、`printf(3C)` 形式に従います。

`ps_pglobal_lookup()`
この関数は、ターゲットプロセス内のシンボルを検索します。

```
ps_err_e ps_pglobal_lookup(const struct ps_prochandle * ph,
        const char * obj, const char * name, ulong_t * sym_addr);
```

ターゲットプロセス *ph* 内のオブジェクト *obj* 内で、シンボル *name* が検索されます。シンボルが検出されると、シンボルのアドレスが *sym_addr* に保存されます。

`ps_pglobal_sym()`
この関数は、ターゲットプロセス内のシンボルを検索します。

```
ps_err_e ps_pglobal_sym(const struct ps_prochandle * ph,
        const char * obj, const char * name, ps_sym_t * sym_desc);
```

ターゲットプロセス *ph* 内のオブジェクト *obj* 内で、シンボル *name* が検索されます。シンボルが検出されると、シンボルの記述子が *sym_desc* に保存されます。

「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シンボルを検出してから、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_LDSO ((const char *)0x1)  /* runtime linker id */
```

制御プロセスは、次の擬似コードを使用して、これらのオブジェクト用の `procfs` ファイルシステムを利用できます。

```
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シンボル情報をテストできます。

◆ ◆ ◆ 第 7 章

7 オブジェクトファイル形式

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 つの種類があります。

- 「再配置可能オブジェクト」ファイルは、コードとデータが入っているセクションを保持します。このファイルは、ほかの再配置可能オブジェクトファイルとリンクして、動的実行可能ファイル、共有オブジェクトファイル、または別の再配置可能オブジェクトを作成するのに適しています。
- 「動的実行可能」ファイルは、実行可能なプログラムを保持します。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します。このファイルは、一般的に実行時に共有オブジェクトファイルと結合され、プロセスイメージを作成します。
- 「共有オブジェクト」ファイルは、追加リンクに適したコードとデータを保持します。リンカーは、共有オブジェクトファイルをほかの再配置可能オブジェクトファイルや共有オブジェクトファイルとともに処理して、別のオブジェクトファイルを作ることができます。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ほか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します。

この章の最初の節、212 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。次の節、283 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。

プログラム内からオブジェクトファイルを操作するには、ELF アクセスライブラリ `libelf` によって提供される関数を使用します。`libelf` の説明については、`elf(3ELF)` のマニュアルページ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

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。次の図にオブジェクトファイルの編成を示します。

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。

注 – 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。ELF 形式には柔軟性があるため、ヘッダーテーブル、セクション、およびセグメントの順序は特に決まっていません。この図に示したのは、Solaris OS で使用される典型的なレイアウトです。

「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、セクションの集合です。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最小単位です。

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の大部分を保持します。このデータには、命令、データ、シンボルテーブル、再配置情報などが含

まれます。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生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する必要があります。再配置可能オブジェクトでは、プログラムヘッダーテーブルは必要ありません。

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。

データ表現

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ー、および 64 ビットアーキテクチャーを持つさまざまなプロセッサをサポートしています。しかしながら、データ表現は、より大きな、またはより小さなアーキテクチャーに拡張できるように意図されています。表 7–1 と表 7–2 に、32 ビットデータタイプと 64 ビットデータタイプの一覧を示します。

オブジェクトファイルは、いくつかの制御データをマシンに依存しない形式で表現します。この形式は、オブジェクトファイルの共通の識別および解釈を規定します。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　ELF 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　ELF 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。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できます。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 メンバー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ます。同様に、Elf64_Addr メンバーが存在する構造は 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

ELF ヘッダーには実際のサイズが記録されるため、オブジェクトファイル内の制御構造は大きくなることがあります。オブジェクトファイルの形式が変更されると、プログラムは、予想より大きい、または小さい制御構造に遭遇する可能性があります。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。不足する情報の取り扱いは状況に依存し、拡張が定義されたときに定められます。

ELF ヘッダーの構造体は次のとおりです。sys/elf.h を参照してください。

```
#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;
        Elf64_Half      e_phentsize;
        Elf64_Half      e_phnum;
        Elf64_Half      e_shentsize;
        Elf64_Half      e_shnum;
        Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

`e_ident`
先頭のバイト列は、オブジェクトファイルであることを示す印です。これらのバイトには、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。詳細な説明は、218 ページの「ELF 識別」に記載されています。

`e_type`
オブジェクトファイルの種類を示します。次の種類が存在します。

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

名前	値	意味
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、プロセッサ固有のセマンティクスのために予約されています。ほかの値は、将来の使用に備えて保留されます。

e_machine
個々のファイルに必要なアーキテクチャーを指定します。関連するアーキテクチャーを、次の表に示します。

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
EM_AMD64	62	AMD 64

ほかの値は、将来の使用に備えて保留されます。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、e_flags に定義されるフラグでは、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
オブジェクトファイルのバージョンを示します。次のバージョンが存在します。

名前	値	意味
EV_NONE	0	無効バージョン
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、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
システムが制御を最初に渡す仮想アドレスを保持し、仮想アドレスが与えられると、プロセスが起動します。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、このメンバーは 0 を保持します。

e_phoff
プログラムヘッダーテーブルのファイルオフセットを保持します (単位: バイト)。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、このメンバーは 0 を保持します。

e_shoff
セクションヘッダーテーブルのファイルオフセットを保持します (単位: バイト)。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、このメンバーは 0 を保持します。

e_flags
ファイルに対応付けられた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。このメンバーは、現在 x86 に対しては 0 です。SPARC の場合のフラグを、次の表に示します。

名前	値	意味
EF_SPARC_EXT_MASK	0xffff00	ベンダー拡張マスク
EF_SPARC_32PLUS	0x000100	V8+ 共通機能
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
EF_SPARCV9_MM	0x3	メモリーモデルのマスク
EF_SPARCV9_TSO	0x0	トータルストアオーダリング (TSO)
EF_SPARCV9_PSO	0x1	パーシャルストアオーダリング (PSO)
EF_SPARCV9_RMO	0x2	リラックスメモリーオーダリング (RMO)

e_ehsize
ELF ヘッダーのサイズ (単位: バイト)。

e_phentsize
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)。すべてのエントリは同じサイズです。

e_phnum
プログラムヘッダーテーブルのエントリ数。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。

プログラムヘッダーの数が PN_XNUM (0xffff) 以上である場合、このメンバーは値 PN_XNUM (0xffff) を保持します。プログラムヘッダーテーブルエントリの実際の数は、インデックス 0 のセクションヘッダーの sh_info フィールドに含まれます。そうでない場合、初期セクションヘッダーエントリの sh_info メンバーには値 0 が入っています。表 7–6 および表 7–7 を参照してください。

e_shentsize
セクションヘッダーのサイズ (単位:バイト)。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
セクションヘッダーテーブルのエントリ数。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。

セクション数が SHN_LORESERVE (0xff00) 以上の場合、e_shnum の値は 0 になります。セクションヘッダーテーブルエントリの実際の数は、インデックス 0 の sh_size フィールドに含まれます。そうでない場合、初期セクションヘッダーエントリの sh_size メンバーには値 0 が入っています。表 7–6 および表 7–7 を参照してください。

e_shstrndx
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。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、このメンバーは値 SHN_UNDEF を保持します。

セクション名文字列テーブルセクションのインデックスが SHN_LORESERVE (0xff00) 以上の場合、このメンバーの値は SHN_XINDEX (0xffff) となり、セクション名文字列テーブルセクションの実際のインデックスはインデックス 0 のセクション ヘッダーの sh_link フィールドに入っています。そうでない場合、初期セクションヘッダーエントリの sh_link メンバーには値 0 が入っています。表 7–6 および表 7–7 を参照してください。

ELF 識別

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するため、ファイルの初期バイトによりファイルの解釈方法が

指定されます。これらの初期バイトは、問い合わせが行われるプロセッサにも、ファイルのほかの内容にも依存しません。

ELF ヘッダーおよびオブジェクトファイル の初期バイトは、e_ident メンバーに一致します。

表 7–3 ELF 識別インデックス

名前	値	目的
EI_MAG0	0	ファイルの識別
EI_MAG1	1	ファイルの識別
EI_MAG2	2	ファイルの識別
EI_MAG3	3	ファイルの識別
EI_CLASS	4	ファイルのクラス
EI_DATA	5	データの符号化
EI_VERSION	6	ファイルのバージョン
EI_OSABI	7	オペレーティングシステム / ABI の識別
EI_ABIVERSION	8	ABI のバージョン
EI_PAD	9	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、次の値を保持するバイトにアクセスします。

EI_MAG0 - EI_MAG3
: ファイルを ELF オブジェクトファイルとして識別する 4 バイトの「マジックナンバー」。次の表を参照してください。

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
: 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。次の表にファイルのクラスを示します。

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コンテナ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。オブジェクトファイルセクションに含まれるデータは、異なるプログラミングモデルに準拠する場合があります。

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。このクラスは、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、64 ビット SPARC や x64 などの 64 ビットアーキテクチャー用に予約されています。このクラスは、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 (次の表を参照)。

名前	値	意味
ELFDATANONE	0	無効な符号化
ELFDATA2LSB	1	図 7–2 を参照してください。
ELFDATA2MSB	2	図 7–3 を参照してください。

これらの符号化の詳細は、221 ページの「データの符号化」で説明します。ほかの値は、将来の使用に備えて保留されます。

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、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 ABI とともにオペレーティングシステムを識別します。ほかの 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または ABI 特有の意味を持つフラグおよび値を保持するものがあります。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 の互換性のな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、指定な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、これらの値を無視します。

データの符号化

ファイルのデータ符号化方式は、ファイルの整数タイプを解釈する方法を指定します。クラス ELFCLASS32 のファイルおよびクラス ELFCLASS64 のファイルは、1、2、4、および 8 バイトを占める整数を使用して、オフセット、アドレス、およびその他の情報を表現します。定義されている符号化方式の下では、オブジェクトは次の図の説明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。この符号化は、一般的にはよく「リトルエンディアン」と呼ばれます。

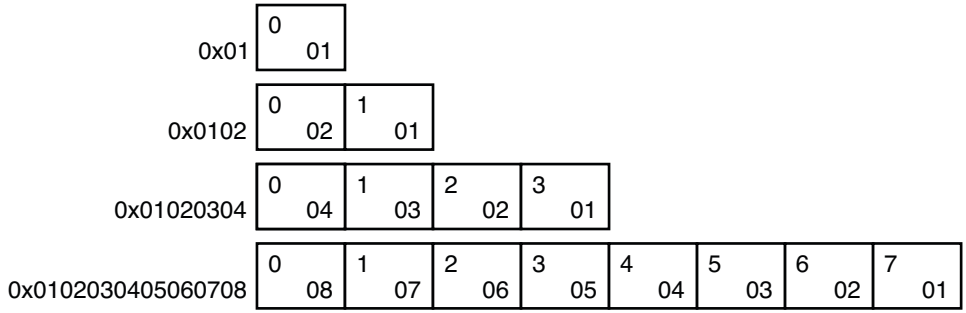


図 7–2 データの符号化方法 ELFDATA2LSB

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。この符号化は、一般的にはよく「ビッグエンディアン」と呼ばれます。

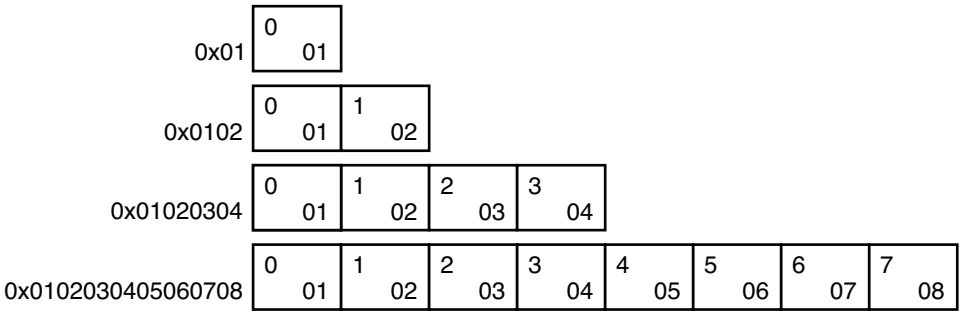


図 7–3 データの符号化方法 ELFDATA2MSB

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、ファイルのセクションすべて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、`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` メンバーは、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示します。`e_shnum` メンバーは、セクションヘッダーテーブルに含まれるエントリ数を示します。`e_shentsize` メンバーは、各エントリのバイト単位の大きさを示します。

セクション数が `SHN_LORESERVE` (`0xff00`) 以上の場合、`e_shnum` の値は `SHN_UNDEF` (`0`) になります。セクションヘッダーテーブルエントリの実際の数は、インデックス `0` の `sh_size` フィールドに含まれます。そうでない場合、初期エントリの `sh_size` メンバーには値 0 が入っています。

セクションヘッダーテーブルインデックスの中には、インデックスサイズが制限されている文脈で予約されているものがあります。たとえば、シンボルテーブルエントリの `st_shndx` メンバー、 ELF ヘッダーの `e_shnum` メンバーと `e_shstrndx` メンバーなどがそうです。このような文脈では、予約値はオブジェクトファイル内の実際のセクションを示しません。また、このような文脈では、エスケープ値は、実際のセクションインデックスがどこかもっと大きなフィールド内に存在することを示します。

表 7–4 ELF 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 7–4 ELF セクションの特殊インデックス (続き)

名前	値
SHN_BEFORE	0xff00
SHN_AFTER	0xff01
SHN_AMD64_LCOMMON	0xff02
SHN_HIPROC	0xff1f
SHN_LOOS	0xff20
SHN_LOSUNW	0xff3f
SHN_SUNW_IGNORE	0xff3f
SHN_HISUNW	0xff3f
SHN_HIOS	0xff3f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_XINDEX	0xffff
SHN_HIRESERVE	0xffff

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 メンバーが、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_UNDEF
未定義、存在しない、無関係など、無意味なセクション参照。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。

SHN_LORESERVE
予約済みインデックスの範囲の下限。

SHN_LOPROC - SHN_HIPROC
この両端を含む範囲の値は、プロセッサ固有のセマンティクスのために予約されています。

SHN_LOOS - SHN_HIOS
この両端を含む範囲の値は、オペレーティングシステム固有のセマンティクスのために予約されています。

SHN_LOSUNW - SHN_HISUNW
この両端を含む範囲の値は、Sun 固有のセマンティクスのために予約されています。

SHN_SUNW_IGNORE
このセクションインデックスは、再配置可能オブジェクト内の一時的なシンボル定義を提供します。dtrace(1M) の内部使用のため予約されています。

SHN_BEFORE, SHN_AFTER
SHF_LINK_ORDER および SHF_ORDERED セクションフラグとともに先頭および末尾のセクションの順序付けを行います。表 7–8 を参照してください。

SHN_AMD64_LCOMMON
x64 固有の共通ブロックラベル。このラベルは SHN_COMMON に似ていますが、大規模な共通ブロックの識別をサポートする点が異なります。

SHN_ABS
対応する参照の絶対値。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
このセクションに対して相対的に定義されるシンボルは、FORTRAN の COMMON や割り当てられていない C 外部変数などの共通シンボルです。これらのシンボルは、一時的シンボルと呼ばれることもあります。

SHN_XINDEX
実際のセクションヘッダーインデックスが大きすぎて格納先のフィールドに入りきらないことを示すエスケープ値。ヘッダーセクションインデックスは、このインデックスが出現する構造体に固有の別の場所に存在します。

SHN_HIRESERVE
予約済みインデックスの範囲の上限。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。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次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ちょうど 1 つ含まれます。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ります。
- 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) を占めます。
- ファイル内のセクション同士は重なりません。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ありません。

- 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ります。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ります。使用されていないデータの内容は不定です。

セクションヘッダーの構造体は、次のとおりです。sys/elf.h を参照してください。

```
typedef struct {
        elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        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
: セクション名。このメンバー値はセクションヘッダーの文字列テーブルセクションへのインデックスで、ヌル文字で終わる文字列の位置を示します。セクション名とその説明は、表 7–10 を参照してください。

sh_type
: セクションの内容とセマンティクスを分類します。セクションの種類とその説明は、表 7–5 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 7–8 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、このメンバー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、このメンバーには 0 が存在します。

sh_offset
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。SHT_NOBITS 型のセクションの場合はファイル内のスペースを占めないため、このメンバーは、ファイル内の概念的なオフセットを示します。

sh_size
セクションのサイズ (バイト)。セクションのタイプが SHT_NOBITS でないかぎり、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
セクションヘッダーテーブルのインデックスリンク。このリンクの解釈は、セクションのタイプに依存します。値については、表 7–9 を参照してください。

sh_info
追加情報。情報の解釈は、セクションのタイプに依存します。値については、表 7–9 を参照してください。このセクションヘッダーの sh_flags フィールドに属性 SHF_INFO_LINK が含まれている場合、このメンバーはセクションヘッダーテーブルインデックスを表します。

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。この場合、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メンバーは、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、このメンバーには 0 が格納されます。

セクションヘッダーの sh_type メンバーは、次の表に示すようにこのセクションのセマンティクスを示します。

表 7–5 ELFセクションタイプ、*sh_type*

名前	値
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_INIT_ARRAY	14
SHT_FINI_ARRAY	15
SHT_PREINIT_ARRAY	16
SHT_GROUP	17
SHT_SYMTAB_SHNDX	18
SHT_LOOS	0x60000000
SHT_LOSUNW	0x6ffffff4
SHT_SUNW_dof	0x6ffffff4
SHT_SUNW_cap	0x6ffffff5
SHT_SUNW_SIGNATURE	0x6ffffff6
SHT_SUNW_ANNOTATE	0x6ffffff7
SHT_SUNW_DEBUGSTR	0x6ffffff8
SHT_SUNW_DEBUG	0x6ffffff9
SHT_SUNW_move	0x6ffffffa
SHT_SUNW_COMDAT	0x6ffffffb

表 7–5 ELF セクションタイプ、*sh_type* (続き)

名前	値
SHT_SUNW_syminfo	0x6ffffffc
SHT_SUNW_verdef	0x6ffffffd
SHT_SUNW_verneed	0x6ffffffe
SHT_SUNW_versym	0x6fffffff
SHT_HISUNW	0x6fffffff
SHT_HIOS	0x6fffffff
SHT_LOPROC	0x70000000
SHT_SPARC_GOTDATA	0x70000000
SHT_AMD64_UNWIND	0x70000001
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

SHT_NULL
: セクションヘッダーが無効であることを示します。このセクションヘッダーには、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ほかのメンバーの値は不定です。

SHT_PROGBITS
: プログラムによって定義された情報を示します。その形式や意味はすべて、プログラムによって決定されます。

SHT_SYMTAB、SHT_DYNSYM
: シンボルテーブルを示します。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示します。このテーブル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、267 ページの「シンボルテーブルセクション」を参照してください。

SHT_STRTAB、SHT_DYNSTR
: 文字列テーブルを示します。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を指定できます。詳細は、265 ページの「文字列テーブルセクション」を参照してください。

SHT_RELA
: 32 ビットクラスのオブジェクトファイル用のタイプ Elf32_Rela など、明示的加数を含む再配置エントリを示します。オブジェクトファイルには、複数の再配置セクションを指定できます。詳細は、252 ページの「再配置セクション」を参照してください。

SHT_HASH
: シンボルハッシュテーブルを示します。動的にリンクされた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、247 ページの「ハッシュテーブルセクション」を参照してください。

SHT_DYNAMIC
: 動的リンク処理用の情報を示します。現在、オブジェクトファイルには動的セクションを 1 つだけ含めることができます。詳細は、296 ページの「動的セクション」を参照してください。

SHT_NOTE
: ファイルに何らかの方法で付加すべき情報を示します。詳細は、251 ページの「注釈セクション」を参照してください。

SHT_NOBITS
: ファイル内の領域を占有しないセクションを示します。このセクションは、その他の点では SHT_PROGBITS に似ています。このセクションにはデータは存在しませんが、sh_offset メンバーには概念上のファイルオフセットが存在します。

SHT_REL
: 32 ビットクラスのオブジェクトファイル用のタイプ Elf32_Rel など、明示的加数を含まない再配置エントリを示します。オブジェクトファイルには、複数の再配置セクションを指定できます。詳細は、252 ページの「再配置セクション」を参照してください。

SHT_SHLIB
: セマンティクスが定義されていない予約済みセクションを示します。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_INIT_ARRAY
: 初期設定関数へのポインタの配列を含むセクションを示します。配列内の各ポインタは、void を戻り値とする、パラメータを持たないプロシージャーと見なされます。詳細は、41 ページの「初期設定および終了セクション」を参照してください。

SHT_FINI_ARRAY
: 終了関数へのポインタの配列を含むセクションを示します。配列内の各ポインタは、void を戻り値とする、パラメータを持たないプロシージャーと見なされます。詳細は、41 ページの「初期設定および終了セクション」を参照してください。

SHT_PREINIT_ARRAY
: ほかのすべての初期設定関数の前に呼び出される関数へのポインタの配列を含むセクションを示します。配列内の各ポインタは、void を戻り値とする、パラメータを持たないプロシージャーと見なされます。詳細は、41 ページの「初期設定および終了セクション」を参照してください。

SHT_GROUP
: セクショングループを示します。セクショングループとは、関連する一連のセクションであり、リンカーは 1 つの単位として扱う必要があります。タイプが SHT_GROUP であるセクションは、再配置可能オブジェクト内にしか存在できません。詳細は、244 ページの「グループセクション」を参照してください。

SHT_SYMTAB_SHNDX
: 拡張されたセクションインデックスが入ったセクション (シンボルテーブルに関連付けられている) を示します。シンボルテーブルによって参照されているセクションヘッダーインデックスのいずれかにエスケープ値 SHN_XINDEX が含まれる場合は、関連する SHT_SYMTAB_SHNDX が必要です。

  SHT_SYMTAB_SHNDX セクションは、Elf32_Word 値の配列です。この配列には、関連するシンボルテーブルエントリごとに 1 つのエントリが存在します。これらの値は、シンボルテーブルエントリが定義されているセクションヘッダーインデックスを示します。一致する Elf32_Word に実際のセクションヘッダーインデックスが含まれるのは、対応するシンボルテーブルエントリの st_shndx フィールドにエスケープ値 SHN_XINDEX が含まれる場合だけです。そうでない場合、エントリは必ず SHN_UNDEF (0) です。

SHT_LOOS – SHT_HIOS
: この両端を含む範囲の値は、オペレーティングシステム固有のセマンティクスのために予約されています。

SHT_LOSUNW – SHT_HISUNW
: この両端を含む範囲の値は、Solaris OS 用のセマンティクスのために予約されています。

SHT_SUNW_dof
: dtrace(1M) の内部使用のため予約されています。

SHT_SUNW_cap
: ハードウェアとソフトウェアの機能要件を指定します。詳細は、245 ページの「ハードウェアおよびソフトウェア機能に関するセクション」を参照してください。

SHT_SUNW_SIGNATURE
: モジュール検証用の署名を示します。

SHT_SUNW_ANNOTATE
: 注釈セクションの処理は、デフォルトのセクション処理規則のすべてに従います。唯一の例外は、注釈セクションが割り当て不可能なメモリー内に存在する場

合に発生します。セクションのヘッダーフラグ SHF_ALLOC が設定されていないと、リンカーは、このセクションに対する未対応の再配置をすべて黙って無視します。

SHT_SUNW_DEBUGSTR、 SHT_SUNW_DEBUG
: デバッグ情報を示します。このタイプのセクションは、リンカーの -s オプションを使用するか、あるいはリンク編集後に strip(1) を使用して、オブジェクトから取り除くことができます。

SHT_SUNW_move
: 部分的に初期設定されたシンボルを処理するためのデータを示します。詳細は、248 ページの「移動セクション」を参照してください。

SHT_SUNW_COMDAT
: 同一データの複数のコピーを単一のコピーに低減することを可能にするセクションを示します。詳細は、244 ページの「「COMDAT」セクション」を参照してください。

SHT_SUNW_syminfo
: 追加のシンボル情報を示します。詳細は、276 ページの「Syminfo テーブルセクション」を参照してください。

SHT_SUNW_verdef
: このファイルで定義された細粒度のバージョンを示します。詳細は、278 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
: このファイルが必要とする細粒度の依存関係を示します。詳細は、280 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
: シンボルと、ファイルが提供するバージョン定義との関係を記述したテーブルを示します。詳細は、282 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
: この両端を含む範囲の値は、プロセッサ固有のセマンティクスのために予約されています。

SHT_SPARC_GOTDATA
: GOT からの相対アドレスを使って参照される、SPARC 固有のデータを示します。つまり、シンボル _GLOBAL_OFFSET_TABLE_ に割り当てられたアドレスに対する相対的なオフセットです。64 ビット SPARC の場合、このセクション内のデータは、リンク編集時に GOT アドレスの {+-} 2^32 バイト内の場所に結合されなければなりません。

SHT_AMD64_UNWIND
: スタックを巻き戻すための巻き戻し (unwind) 関数テーブルエントリを含む、x64 固有のデータを示します。

SHT_LOUSER
: アプリケーションプログラム用として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。

SHT_HIUSER
: アプリケーションプログラム用として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。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。

ほか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、そのインデックスが未定義セクション参照を示している場合でも、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。その値は次の表のとおりです。

表 7–6 ELF セクションヘッダーテーブルエントリ: 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

セクションまたはプログラムヘッダーの数が ELF ヘッダーデータサイズを超えた場合、セクションヘッダー 0 の構成要素を使って拡張 ELF ヘッダー属性が定義されます。その値は次の表のとおりです。

表 7–7 ELF 拡張セクションヘッダーテーブルエントリ: 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

表 7–7 ELF 拡張セクションヘッダーテーブルエントリ: インデックス 0 (続き)

名前	値	注意
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	e_shnum	セクションヘッダーテーブルのエントリ数
sh_link	e_shstrndx	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インデックス
sh_info	e_phnum	プログラムヘッダーテーブルのエントリ数
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 メンバーは、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 7–8 ELF セクションの属性フラグ

名前	値
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MERGE	0x10
SHF_STRINGS	0x20
SHF_INFO_LINK	0x40
SHF_LINK_ORDER	0x80
SHF_OS_NONCONFORMING	0x100
SHF_GROUP	0x200
SHF_TLS	0x400
SHF_MASKOS	0x0ff00000
SHF_AMD64_LARGE	0x10000000

表 7–8 ELF セクションの属性フラグ (続き)

名前	値
SHF_ORDERED	0x40000000
SHF_EXCLUDE	0x80000000
SHF_MASKPROC	0xf0000000

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
: プロセス実行中に書き込み可能にすべきセクションを示します。

SHF_ALLOC
: プロセス実行中にメモリーを占有するセクションを示します。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
: 実行可能なマシン命令を含むセクションを示します。

SHF_MERGE
: マージして重複をなくすことの可能なデータを含むセクションを示します。同時に SHF_STRINGS フラグが設定されていないかぎり、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、データ要素はヌル文字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
: ヌル文字で終わっている文字列で構成されるセクションを示します。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 フィールドには、セクションヘッダーテーブルのインデックスが格納されます。

SHF_LINK_ORDER
: このセクションは、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。このセクションを出力ファイル内のほかのセクションと結合する場合、結合対象セクションと同じ相対的な順序で現われます。同様に、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。

特殊な sh_link 値である SHN_BEFORE および SHN_AFTER (表 7–4 を参照) は、順序付けされるセット内のほか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。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。

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。

sh_link 順序付け情報が存在しない場合、出力ファイルの 1 つのセクション内にまとめられた単一入力ファイルからのセクションは、連続的になります。これらのセクションの相対順序付けは、入力ファイル内のセクションの相対順序付けと同じになります。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_OS_NONCONFORMING
: このセクションは、不適切な動作を避けるために、標準のリンク処理規則に含まれない OS 固有の特殊処理を必要とします。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、このセクションを含むオブジェクトファイルは拒否され、エラーが出力されます。

SHF_GROUP
: このセクションは、セクショングループのメンバー (おそらく唯一のメンバー) 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内に含まれるセクションに対してしか設定できません。詳細は、244 ページの「グループセクション」を参照してください。

SHF_TLS
: このセクションには、スレッド固有領域が格納されます。プロセス内の各スレッドは、このデータのインスタンスをそれぞれ別個に持ちます。詳細は、第 8 章「スレッド固有領域 (TLS)」を参照してください。

SHF_MASKOS
: このマスクに含まれるビットはすべて、オペレーティングシステム固有のセマンティクスのために予約されています。

SHF_AMD64_LARGE
: x64 用のデフォルトコンパイルモデルで使用できるのは、32 ビットのディスプレイスメントだけです。このディスプレイスメントでは、セクションのサイズ (最終的にはセグメントのサイズ) が 2G バイトに制限されます。この属性フラグは、2G バイトを超えるデータを格納できるセクションを識別します。このフラグを使えば、異なるコードモデルを使用するオブジェクトファイルのリンク処理を行えます。

  SHF_AMD64_LARGE 属性フラグを含まない x64 オブジェクトファイルセクションは、小規模コードモデルを使用するオブジェクトから自由に参照できます。この

フラグを含むセクションは、それよりも規模の大きいコードモデルを使用するオブジェクトからしか参照できません。たとえば、x64 中規模コードモデルのオブジェクトは、この属性フラグを含むセクション内のデータとこの属性フラグを含まないセクション内のデータを参照できます。ところが、x64 小規模コードモデルのオブジェクトは、このフラグを含まないセクション内のデータしか参照できません。

SHF_ORDERED
: このセクションは、同じ型のほか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 および SHN_AFTER (表 7–4 を参照) は、順序付けされるセット内のほか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。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 つのセクション内にまとめられた単一入力ファイルからのセクションは、連続的になります。これらのセクションの相対順序付けは、入力ファイル内で表示されるセクションの相対順序付けと同じになります。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外されます。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
: このマスクに含まれるビットはすべて、プロセッサ固有のセマンティクスのために予約されています。

セクションヘッダーの 2 つのメンバー 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 7–9 ELF sh_link と sh_info の解釈

sh_type	sh_link	sh_info
SHT_DYNAMIC	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
SHT_HASH	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_REL SHT_RELA	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。	sh_flags メンバーに SHF_INFO_LINK フラグが含まれている場合は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、それ以外の場合は 0。表 7–10 と 252 ページの「再配置セクション」も参照してください。
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。	最後の局所シンボルのシンボルテーブルインデックス STB_LOCAL より 1 大きい。
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します。
SHT_SYMTAB_SHNDX	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_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。

表 7–9 ELF sh_link と sh_info の解釈 (続き)

sh_type	sh_link	sh_info
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

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。次の表に示す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指定された型と属性が存在します。

表 7–10 ELF 特殊セクション

名前	種類	属性
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	None
.data、.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dynamic	SHT_DYNAMIC	SHF_ALLOC + SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.eh_frame_hdr	SHT_AMD64_UNWIND	SHF_ALLOC
.eh_frame	SHT_AMD64_UNWIND	SHF_ALLOC + SHF_WRITE
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.finiarray	SHT_FINI_ARRAY	SHF_ALLOC + SHF_WRITE
.got	SHT_PROGBITS	311 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	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.initarray	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE

表 7–10 ELF特殊セクション (続き)

名前	種類	属性
.interp	SHT_PROGBITS	295 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	SHT_NOTE	None
.lbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE
.ldata、.ldata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_AMD64_LARGE
.lrodata、.lrodata1	SHT_PROGBITS	SHF_ALLOC + SHF_AMD64_LARGE
.plt	SHT_PROGBITS	312 ページの「プロシージャーのリンクテーブル (プロセッサ固有)」を参照してください
.preinitarray	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
.rela	SHT_RELA	None
.rel*name*	SHT_REL	252 ページの「再配置セクション」を参照してください
.rela*name*	SHT_RELA	252 ページの「再配置セクション」を参照してください
.rodata、.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	None
.strtab	SHT_STRTAB	この表のあとの説明を参照してください。
.symtab	SHT_SYMTAB	267 ページの「シンボルテーブルセクション」を参照してください
.symtab_shndx	SHT_SYMTAB_SHNDX	267 ページの「シンボルテーブルセクション」を参照してください
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata、.tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.SUNW_bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.SUNW_cap	SHT_SUNW_cap	SHF_ALLOC
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_move	SHT_SUNW_move	SHF_ALLOC

表 7–10 ELF 特殊セクション (続き)

名前	種類	属性
.SUNW_reloc	SHT_REL	SHF_ALLOC
	SHT_RELA	
.SUNW_syminfo	SHT_SUNW_syminfo	SHF_ALLOC
.SUNW_version	SHT_SUNW_verdef	SHF_ALLOC
	SHT_SUNW_verneed	
	SHT_SUNW_versym	

.bss
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。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 で示しているとおり、ファイル領域を占めません。

.comment
コメント情報 (通常、コンパイルシステムのコンポーネントが使用)。このセクションは、mcs(1) により操作できます。

.data, .data1
プログラムのメモリーイメージで使用される、初期化済みのデータ。

.dynamic
動的リンク情報。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
動的リンクに必要な文字列 (もっとも一般的には、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)。

.dynsym
動的リンクシンボルテーブル。詳細は、267 ページの「シンボルテーブルセクション」を参照してください。

.eh_frame_hdr、 .eh_frame
スタックを戻すために使用する呼び出しフレーム情報。

.fini
こ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関数で使用される実行可能命令。詳細は、95 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。

.finiarray
こ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。詳細は、95 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
大域オフセットテーブル。詳細は、311 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
シンボルハッシュテーブル。詳細は、247 ページの「ハッシュテーブルセクション」を参照してください。

.init
こ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化関数で使用される実行可能命令。詳細は、95 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。

.initarray
こ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化配列に使用される関数ポインタの配列。詳細は、95 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
プログラムインタプリタのパス名。詳細は、295 ページの「プログラムインタプリタ」を参照してください。

.lbss
x64 固有の初期化されていないデータ。このデータは .bss に似ていますが、2G バイトを超えるセクションをサポートする点が異なります。

.ldata、.ldata1
x64 固有の初期化済みデータ。このデータは .data に似ていますが、2G バイトを超えるセクションをサポートする点が異なります。

.lrodata、.lrodata1
x64 固有の読み取り専用データ。このデータは .rodata に似ていますが、2G バイトを超えるセクションをサポートする点が異なります。

.note
251 ページの「注釈セクション」に記載された形式の情報。

.plt
プロシージャーのリンクテーブル。詳細は、312 ページの「プロシージャーのリンクテーブル (プロセッサ固有)」を参照してください。

.preinitarray
こ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。詳細は、95 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
特定のセクションに適用されない再配置情報。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、275 ページの「レジスタシンボル」を参照してください。

`.rel`*name*、`.rela`*name*
再配置情報 (詳細は、252 ページの「再配置セクション」を参照)。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`
読み取り専用データ (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)。詳細は、283 ページの「プログラムヘッダー」を参照してください。

`.shstrtab`
セクション名。

`.strtab`
文字列。通常は、シンボルテーブルエントリに関連付けられた名前を表す文字列です。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`
シンボルテーブル (詳細は、267 ページの「シンボルテーブルセクション」を参照)。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。

`.symtab_shndx`
このセクションには、`.symtab` による指定に従い、特別なシンボルテーブルセクションインデックス配列が保持されます。関連付けられたシンボルテーブルセクションに SHF_ALLOC ビットが含まれる場合、このセクションの属性も SHF_ALLOC ビットを含みます。そうでない場合、このビットはオフになります。

`.tbss`
このセクションには、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スレッド固有データが格納されます。データが新しい実行フロー用に具体化されると、システムはデータを 0 で初期化します。このセクションは、セクション型 SHT_NOBITS で示しているとおり、ファイル領域を占めません。詳細は、第 8 章「スレッド固有領域 (TLS)」を参照してください。

`.tdata`、`.tdata1`
これらのセクションは、プログラムのメモリーイメージで使用される、初期化されたスレッド固有データを保持します。その内容のコピーは、それぞれ新しい実行フロー用にシステムによって具体化されます。詳細は、第 8 章「スレッド固有領域 (TLS)」を参照してください。

`.text`
プログラムの「テキスト」すなわち実行可能命令。

.SUNW_bss
プログラムのメモリーイメージで使用される、共有オブジェクト用の部分的に初期化されたデータ。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 で示しているとおり、ファイル領域を占めません。

.SUNW_cap
ハードウェアとソフトウェア機能の要件。詳細は、245 ページの「ハードウェアおよびソフトウェア機能に関するセクション」を参照してください。

.SUNW_heap
dldump(3C) により作成される動的実行可能ファイルのヒープ。

.SUNW_move
部分的に初期化されたデータに関する追加情報。詳細は、248 ページの「移動セクション」を参照してください。

.SUNW_reloc
再配置情報 (詳細は、252 ページの「再配置セクション」を参照)。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
シンボルテーブルの追加情報。詳細は、276 ページの「Syminfo テーブルセクション」を参照してください。

.SUNW_version
バージョン情報。詳細は、277 ページの「バージョン管理セクション」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、システムで予約されています。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、予約されていないセクションを定義できます。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ー用に予約されるセクション名は、アーキテクチャー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ー名を入れる必要があります。たとえば、.Foo.psect は、FOO アーキテクチャーで定義される 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。

「COMDAT」セクション

「COMDAT」セクションは、セクション名 (sh_name) で一意に識別されます。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を検出すると、最初のセクションが保持され、残りのセクションは破棄されます。破棄された SHT_SUNW_COMDAT セクションに適用された再配置はすべて無視されます。破棄されたセクションで定義されたシンボルもすべて削除されます。

さらに、リンカーは、コンパイラ起動時に -xF オプションが指定された場合のセクション再順序付けで使用されるセクション命名規則をサポートします。関数が *.sectname%funcname* という名前の SHT_SUNW_COMDAT セクションに配置された場合、保持されている最終的な SHT_SUNW_COMDAT セクションは、*.sectname* という名前のセクションに結合されます。この方法を使用すると、SHT_SUNW_COMDAT セクションは最終的に .text、.data、またはほかのセクションに入れられます。

グループセクション

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以外にも、別の情報が必要になる場合もあります。この別の情報は、参照される文字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ッギング情報セクション、およびその他の情報セクションなどです。

グループセクション間では内部参照がある場合もあります。ただし、別のオブジェクトからの重複によって、これらのセクションの 1 つが削除 (あるいは、置換) されると、このような参照は意味を成さなくなります。したがって、このようなグループをリンクされたオブジェクトに組み込んだり、オブジェクトから削除したりするときは、1 つの単位として扱います。

タイプ SHT_GROUP のセクションは、そのような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、識別シンボルエントリを指定します。sh_link メンバー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 メンバー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 メンバーは、値 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、次のフラグが定義されています。

表 7–11 ELF グループセクションのフラグ

名前	値
GRP_COMDAT	0x1

GRP_COMDAT
: GRP_COMDAT は COMDAT グループであることを示します。このグループは、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ほかのオブジェクトファイル内のほかの 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ます。残りのグループのメンバーは破棄されます。

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。こ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メンバー内に設定していなければなりません。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メンバーを削除します。

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 (タイプ STT_SECTION を持つシンボルを含む) を参照できません。
- グループを形成するセクションにグループの外から非シンボル参照を行なってはいけません。たとえば、sh_link または sh_info メンバー内でのグループメンバーのセクションヘッダーインデックスは使用できません。
- グループのセクションの 1 つに関連して定義されたシンボルテーブルエントリは、グループのメンバーが破棄されると削除されることがあります。この削除が行われるのは、シンボルテーブルエントリが含まれるシンボルテーブルセクションがグループの一部ではない場合です。

ハードウェアおよびソフトウェア機能に関するセクション

SHT_SUNW_cap セクションは、オブジェクトのハードウェアとソフトウェア機能を特定します。このセクションには、次の構造の配列が含まれます。sys/link.h を参照してください。

```
typedef struct {
        Elf32_Word      c_tag;
        union {
                Elf32_Word      c_val;
                Elf32_Addr      c_ptr;
        } c_un;
} Elf32_Cap;

typedef struct {
        Elf64_Xword     c_tag;
        union {
                Elf64_Xword     c_val;
                Elf64_Addr      c_ptr;
        } c_un;
} Elf64_Cap;
```

この種の各オブジェクトに対して、c_tag は c_un の解釈を制御します。

c_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。

c_ptr
　このオブジェクトは、プログラムの仮想アドレスを表します。

次の機能タグがあります。

表 7–12　ELF 機能配列タグ

名前	値	c_un
CA_SUNW_NULL	0	無視される
CA_SUNW_HW_1	1	c_val
CA_SUNW_SF_1	2	c_val

CA_SUNW_NULL
　機能配列の終わりを示します。

CA_SUNW_HW_1
　ハードウェア機能の値を示します。c_val 要素は、関連ハードウェア機能を表す値を含みます。SPARC プラットフォームでは、ハードウェア機能は sys/auxv_SPARC.h に定義されます。x86 プラットフォームでは、ハードウェア機能は sys/auxv_386.h に定義されます。

CA_SUNW_SF_1
　ソフトウェア機能の値を示します。c_val 要素は、sys/elf.h に定義される関連ソフトウェア機能を表す値を含みます。

再配置可能オブジェクトには、機能セクションを含めることができます。リンカーは、複数の入力再配置可能オブジェクトからの機能セクションを 1 つの機能セクションに統合します。リンカーを使用すると、オブジェクトの構築時に機能を定義することもできます。68 ページの「ハードウェアとソフトウェア機能の特定」を参照してください。

ハードウェア機能情報が格納された機能セクションを含む動的オブジェクトでは、そのセクションに関連付けられた PT_SUNWCAP プログラムヘッダーが存在しています。このプログラムヘッダーにより、実行時リンカーは、プロセスで利用可能なハードウェア機能に対してオブジェクトを確認できます。

異なるハードウェア機能を利用する動的オブジェクトは、フィルタを使用して柔軟な実行時環境を提供できます。385 ページの「ハードウェア機能固有の共有オブジェクト」を参照してください。

ハッシュテーブルセクション

ハッシュテーブルは、シンボルテーブルへのアクセスを提供する Elf32_Word または Elf64_Word オブジェクトから構成されます。SHT_HASH セクションは、このハッシュテーブルを提供します。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。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次の図ではラベルを表示しますが、ラベルは仕様の一部ではありません。

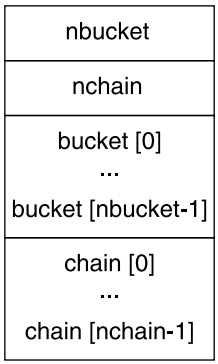


図 7–4 シンボルハッシュテーブル

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により、連鎖テーブルエントリも選択されます。

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。つまり、ハッシュ関数がある名前に対して値 *x* を返した場合、bucket [*x*% nbucket] はインデックス *y* を返します。このインデックスは、シンボルテーブルと連鎖テーブルの両方へのインデックスです。シンボルテーブルエントリが目的の名前でなかった場合、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返します。

目的の名前を持つシンボルテーブルエントリが選択されるか、chain エントリの値が STN_UNDEF になるまで、chain リンクをたどることができます。

ハッシュ関数を次に示します。

```
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;
}
```

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。

FORTRAN COMMON ブロックなど、部分的に初期化された大規模なデータ変数を含むオブジェクトは、多大なディスクスペースオーバーヘッドを引き起こす可能性があります。SHT_SUNW_move セクションは、これらのデータ変数を圧縮するメカニズムを提供します。これにより、関連するオブジェクトのディスクサイズを減らすことができます。

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を含みます。これらのエントリは、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す。これらの項目はオブジェクトファイル内のスペースは使用しません

が、実行時にはオブジェクトのメモリーイメージに反映されます。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))
```

これらの構造の要素は次のとおりです。

m_value
: 初期設定値で、この値はメモリーイメージへ移されます。

m_info
: 初期設定が適用されるものに関連するシンボルテーブルインデックス、および初期設定されるオフセットのサイズ (単位: バイト)。このメンバーの下位 8 ビットにはサイズを定義します (1、2、4、または 8)。上位ビットにはシンボルインデックスを定義します。

m_poffset
: 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。

m_repeat
: 繰り返し回数。

m_stride
: スキップの数。この値は、繰り返し初期化を行う際にスキップされる単位数を示します。1 単位は m_info で定義された初期化オブジェクトのサイズで

す。m_stride が 0 の場合、初期化が連続した単位に対して行われることを示します。

次のデータ定義は、通常、オブジェクトファイル内で 0x8000 バイトを消費します。

```
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'}
};
```

このデータを説明するために、SHT_SUNW_move セクションを使用します。データ項目は、 .bss セクションに定義されます。このデータ項目のゼロ以外の要素は、適切な移動エントリで初期化されます。

```
$ elfdump -s data | fgrep move
     [17]  0x00020868 0x00008000  OBJT GLOB 0   .bss       move
$ elfdump -m data

Move Section: .SUNW_move
    symndx offset   size repeat stride   value               with respect to
      [17]      8      4      1      1 0x00000000000000001  move
      [17]     12      4      1      1 0x00000000031000000  move
      [17]     24      4      2      1 0x0000000000000000f  move
      [17]     28      4      2      1 0x00000000046000000  move
      [17]     48      4      1      1 0x0000000000000000e  move
      [17]     52      4      1      1 0x00000000045000000  move
      [17]     64      4      1      1 0x0000000000000000e  move
      [17]     68      4      1      1 0x00000000045000000  move
```

再配置可能オブジェクトから提供される移動セクションは連結され、リンカーにより作成されるオブジェクト内に出力されます。ただし、次の条件が成り立つ場合、リンカーは移動エントリを処理します。この処理は、移動エントリの内容を従来のデータ項目に拡張します。

- 出力ファイルは、静的な実行可能ファイルである。
- 移動エントリのサイズは、移動データの拡張先のシンボルのサイズより大きくなる。
- -z nopartial オプションは有効である。

注釈セクション

ベンダーやシステムエンジニアは、オブジェクトファイルに特別な情報を付加し、ほかのプログラムからその準拠性や互換性を確認できるようにする必要があることがあります。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。

次の図に示すように、セクションとプログラムヘッダー要素内の注釈情報は、任意の数のエントリを保持します。64 ビットオブジェクトおよび 32 ビット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わかりやすくするためにラベルを図 7–6 に示しますが、ラベルは仕様の一部ではありません。

図 7–5 注釈の情報

namesz と name
: 名前の先頭 namesz バイトには、エントリの所有者または作者を示す、ヌル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メカニズムは存在しません。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 がない場合、namesz の値は 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。記述子がない場合、descsz の値は 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 はパットの長さを含みません。

type
: 注釈の解釈を示します。各エントリの作者は、自分で種類を管理します。1 つの type 値に関して複数の解釈が存在する場合があります。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。

次の図に示す注釈セグメントは、2 つのエントリを保持しています。

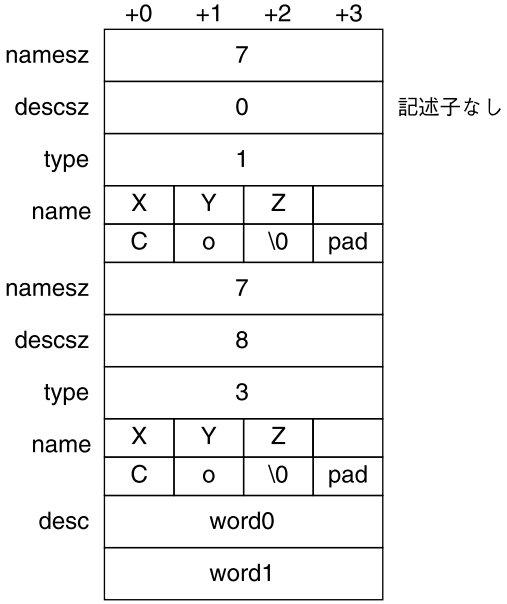


図7–6 注釈セグメントの例

注 – システムは、名前なし (`namesz == 0`) の注釈情報と、長さ 0 の名前 (`name[0] == '\0'`) を持つ注釈情報を予約していますが、現時点ではタイプは定義していません。ほかのすべての名前には、少なくとも 1 つのヌル以外の文字が存在しなければなりません。

再配置セクション

再配置は、シンボル参照をシンボル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。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。この情報により、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、次の構造体を持つことができます。`sys/elf.h` を参照してください。

```
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`
: このメンバーは、再配置処理を適用する位置を与えます。オブジェクトファイルが異なると、このメンバーの解釈が多少異なります。

  再配置可能ファイルの場合、値はセクションのオフセットを示します。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示します。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します。

  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を示します。この情報により、再配置エントリは、実行時リンカーにとって、より意味のあるものになります。

  関連するプログラムによるアクセスの効率を高めるため、メンバーの解釈はオブジェクトファイルによって異なりますが、再配置タイプの意味は同じになります。

`r_info`
: このメンバーは、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は、それぞれ `ELF32_R_TYPE` または `ELF32_R_SYM` をエントリの `r_info` メンバーに適用した結果です。

```
#define ELF32_R_SYM(info)             ((info)>>8)
#define ELF32_R_TYPE(info)            ((unsigned char)(info))
#define ELF32_R_INFO(sym, type)       (((sym)<<8)+(unsigned char)(type))

#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))
```

64 ビット SPARC `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。既存の再配置タイプの場合、データフィールドはゼロになります。これに対し、新しい再配置タイプの場合には、データビットが使用される可能性があります。

```
#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`
: このメンバーは、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。

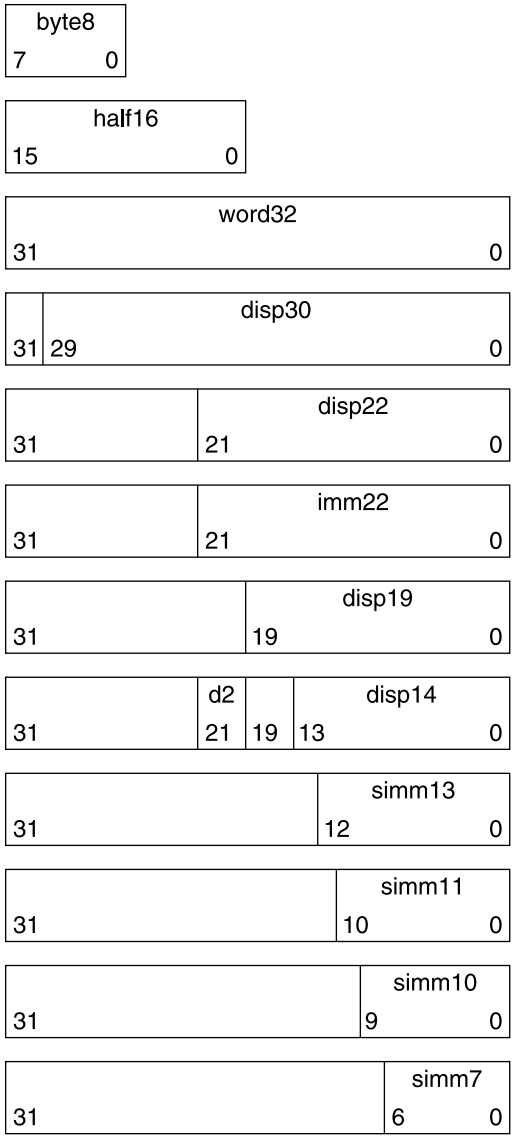
`Rela` エントリには、明示的加数が含まれます。`Rel` エントリには、変更される位置に暗黙の加数が存在します。32 ビット SPARC では、`Elf32_Rela` 再配置エントリのみを使用します。64 ビット SPARC および 64 ビット x86 では、`Elf64_Rela` 再配置エントリのみを使用します。したがって、`r_addend` メンバーは再配置加数として機能します。x86 は、`Elf32_Rel` 再配置エントリのみを使用します。再配置対象のフィールドは、加数を保持します。すべての場合において、加数と計算された結果は同じバイト順序を使用します。

再配置セクションは、ほかに 2 つのセクションを参照することがあります。1 つは `sh_info` セクションヘッダーエントリにより示されるシンボルテーブルで、もう 1 つは `sh_link` セクションヘッダーエントリにより示される変更対象のセクションです。222 ページの「セクション」 に、各セクションの関係を示します。再配置オブジェクトに再配置セクションが存在するが、実行可能ファイルや共有オブジェクトが任意の場合は、`sh_info` エントリが必要です。再配置オフセットが存在すれば、再配置を実行できます。

再配置型 (プロセッサ固有)

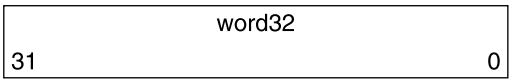
再配置エントリには、次の図に示す命令およびデータフィールドの変更方法が記述されます。ビット番号は、下隅に示されています。

SPARC プラットフォームの場合、再配置エントリはバイト (`byte8`)、ハーフワード (`half16`) またはワードに適用されます。

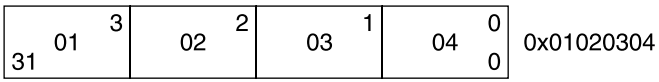


64 ビット SPARC と x64 では、再配置は拡張ワード (`xword64`) にも適用されます。

x86 の場合、再配置エントリはワード (word32) に適用されます。



word32 は、任意バイト整列が存在する 4 バイトを占める 32 ビットフィールドを指定します。これらの値は、x86 アーキテクチャーにおけるほかのワード値と同じバイト順序を使用します。



いずれの場合でも 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。

次の再配置型の計算では、操作により、再配置可能ファイルが実行可能ファイルまたは共有オブジェクトファイルに変換されることが仮定されています。概念上、リンカーは 1 つまたは複数の再配置可能ファイルを併合して出力します。リンカーは、まず入力ファイルの結合/配置方法を決めます。次に、シンボルの値を更新し、再配置を実行します。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に適用される再配置は類似しており、同じ結果を実現します。このセクションの表では、次の表記が使用されています。

A　再配置可能フィールドの値を計算するために使用される加数。

B　実行時に共有オブジェクトがメモリーに読み込まれるベースアドレス。一般的に、共有オブジェクトファイルは、ベース仮想アドレス 0 で作成されます。ただし、共有オブジェクトの実行アドレスは異なります。283 ページの「プログラムヘッダー」を参照してください。

G　実行時に再配置エントリのシンボルのアドレスが存在する大域オフセットテーブルへのオフセット。311 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。

GOT　大域オフセットテーブルのアドレス。311 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。

L　シンボルに対するプロシージャーのリンクテーブルエントリのセクションオフセットまたはアドレス。312 ページの「プロシージャーのリンクテーブル (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セクションオフセットまたはアドレス (r_offset を使用して計算)。

S　インデックスが再配置エントリ内に存在するシンボルの値。

Z　インデックスが再配置エントリ内に存在するシンボルのサイズ。

SPARC: 再配置型

次の表に示すフィールド名は、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 7–13　SPARC: ELF 再配置型

名前	値	フィールド	計算
R_SPARC_NONE	0	None	None
R_SPARC_8	1	V-byte8	S + A
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

表 7–13 SPARC: ELF 再配置型 *(続き)*

名前	値	フィールド	計算
R_SPARC_WPLT30	18	V-disp30	(L + A - P) >> 2
R_SPARC_COPY	19	None	この表のあとの説明を参照してください。
R_SPARC_GLOB_DAT	20	V-word32	S + A
R_SPARC_JMP_SLOT	21	None	この表のあとの説明を参照してください。
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) >> 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) >> 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_HH22	34	V-imm22	(S + A) >> 42
R_SPARC_HM10	35	T-simm13	((S + A) >> 32) & 0x3ff
R_SPARC_LM22	36	T-imm22	(S + A) >> 10
R_SPARC_PC_HH22	37	V-imm22	(S + A - P) >> 42
R_SPARC_PC_HM10	38	T-simm13	((S + A - P) >> 32) & 0x3ff
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
R_SPARC_5	44	V-imm5	S + A
R_SPARC_6	45	V-imm6	S + A
R_SPARC_HIX22	48	V-imm22	((S + A) ^ 0xffffffffffffffff) >> 10

表 7–13 SPARC: ELF 再配置型 (続き)

名前	値	フィールド	計算
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-word32	S + A
R_SPARC_UA16	55	V-half16	S + A
R_SPARC_GOTDATA_HIX22	80	T-imm22	((S + A - GOT) >> 10) ^ ((S + A - GOT) >> 31)
R_SPARC_GOTDATA_LOX10	81	T-imm13	((S + A - GOT) & 0x3ff) | (((S + A - GOT) >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP_HIX22	82	T-imm22	(G >> 10) ^ (G >> 31)
R_SPARC_GOTDATA_OP_LOX10	83	T-imm13	(G & 0x3ff) | ((G >> 31) & 0x1c00)
R_SPARC_GOTDATA_OP	84	Word32	この表のあとの説明を参照してください。

注 – スレッド固有領域の参照に使用できる再配置はほかにも存在します。これらの再配置については、第 8 章「スレッド固有領域 (TLS)」で説明しています。

いくつかの再配置型には、単純な計算を超えたセマンティクスが存在します。

R_SPARC_GOT10
: R_SPARC_LO10 に似ていますが、シンボルの GOT エントリのアドレスを参照する点が異なります。また、 R_SPARC_GOT10 は、大域オフセットテーブルの作成をリンカーに指示します。

R_SPARC_GOT13
: R_SPARC_13 に似ていますが、シンボルの GOT エントリのアドレスを参照する点が異なります。また、 R_SPARC_GOT13 は、大域オフセットテーブルの作成をリンカーに指示します。

R_SPARC_GOT22
: R_SPARC_22 に似ていますが、シンボルの GOT エントリのアドレスを参照する点が異なります。また、 R_SPARC_GOT22 は、大域オフセットテーブルの作成をリンカーに指示します。

R_SPARC_WPLT30
: R_SPARC_WDISP30 に似ていますが、シンボルのプロシージャーリンクテーブルエントリのアドレスを参照する点が異なります。また、R_SPARC_WPLT30 は、プロシージャーのリンクテーブル作成をリンカーに指示します。

R_SPARC_COPY
: リンカーは、この再配置型を作成して、動的実行可能ファイルが読み取り専用のテキストセグメントを保持できるようにします。この再配置型のオフセットメンバーは、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。147 ページの「コピー再配置」を参照してください。

R_SPARC_GLOB_DAT
: R_SPARC_32 に似ていますが、再配置は GOT エントリを指定されたシンボルのアドレスに設定する点が異なります。この特殊な再配置型を使うと、シンボルと GOT 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、動的オブジェクトが遅延結合を提供できるようにするため、この再配置型を作成します。この再配置型のオフセットメンバーは、プロシージャー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、プロシージャー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、動的オブジェクト用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メンバーは、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値 0 を指定する必要があります。

R_SPARC_UA32
: 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。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 (アーキテクチャー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_LM22
: 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_LM22
: 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_HIX22
: 64 ビットアドレス空間の最上位 4G バイトに限定される実行可能ファイルに対して R_SPARC_LOX10 とともに使用されます。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 とともに使用されます。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_L44
: 再配置型 R_SPARC_H44 および R_SPARC_M44 とともに使用され、44 ビット絶対アドレス指定モデルを生成します。

R_SPARC_REGISTER
: レジスタシンボルの初期化に使用されます。この再配置型のオフセットメンバーには、初期化されるレジスタ番号が存在します。このレジスタに対応するレジスタシンボルが必要です。このシンボルの種類は SHN_ABS です。

R_SPARC_GOTDATA_OP_HIX22、R_SPARC_GOTDATA_OP_LOX10 と R_SPARC_GOTDATA_OP
: これらの再配置はコード変換に対応したものです。

64 ビット SPARC: 再配置型

再配置計算に使用される次の表記は、64 ビット SPARC 固有のものです。

O
: 再配置可能フィールドの値を計算するために使用される二次的な加数。この加数は、ELF64_R_TYPE_DATA マクロを適用することにより r_info フィールドから抽出されます。

次の表に示す再配置型は、32 ビット SPARC 用に定義された再配置型を拡張または変更します。257 ページの「SPARC: 再配置型」を参照してください。

表 7–14　64 ビット SPARC: ELF 再配置型

名前	値	フィールド	計算
R_SPARC_HI22	9	V-imm22	(S + A) >> 10
R_SPARC_GLOB_DAT	20	V-xword64	S + A
R_SPARC_RELATIVE	22	V-xword64	B + A
R_SPARC_64	32	V-xword64	S + A
R_SPARC_OLO10	33	V-simm13	((S + A) & 0x3ff) + O
R_SPARC_DISP64	46	V-xword64	S + A - P
R_SPARC_PLT64	47	V-xword64	L + A

表 7–14 64 ビット SPARC: ELF 再配置型 (続き)

名前	値	フィールド	計算
R_SPARC_REGISTER	53	V-xword64	S + A
R_SPARC_UA64	54	V-xword64	S + A
R_SPARC_H34	85	V-imm22	(S + A) >> 12

次の再配置型には、単純な計算を超えたセマンティクスが存在します。

R_SPARC_OLO10
: 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。

32 ビット x86: 再配置型

次の表に、32 ビット x86 用に定義された再配置を示します。

表 7–15 32 ビット x86: ELF 再配置型

名前	値	フィールド	計算
R_386_NONE	0	None	None
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	None	この表のあとの説明を参照してください。
R_386_GLOB_DAT	6	word32	S
R_386_JMP_SLOT	7	word32	S
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
R_386_16	20	word16	S + A
R_386_PC16	21	word16	S + A - P
R_386_8	22	word8	S + A

表 7–15　32 ビット x86: ELF 再配置型　(続き)

名前	値	フィールド	計算
R_386_PC8	23	word8	S + A - P

注– スレッド固有領域の参照に使用できる再配置はほかにも存在します。これらの再配置については、第 8 章「スレッド固有領域 (TLS)」で説明しています。

いくつかの再配置型には、単純な計算を超えたセマンティクスが存在します。

R_386_GOT32
: GOT のベースからシンボルの GOT 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
: シンボルのプロシージャーのリンクテーブルエントリのアドレスを計算し、かつプロシージャー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、この再配置型を作成して、動的実行可能ファイルが読み取り専用のテキストセグメントを保持できるようにします。この再配置型のオフセットメンバーは、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。147 ページの「コピー再配置」を参照してください。

R_386_GLOB_DAT
: GOT 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 GOT 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、動的オブジェクトが遅延結合を提供できるようにするため、この再配置型を作成します。この再配置型のオフセットメンバーは、プロシージャー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、プロシージャー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、動的オブジェクト用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メンバーは、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値 0 を指定する必要があります。

R_386_GOTOFF
: シンボルの値と GOT 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
: R_386_PC32 に似ていますが、計算を行う際に GOT 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。

x64: 再配置型

次の表に、x64 用に定義された再配置を示します。

表 7–16 x64: ELF 再配置型

名前	値	フィールド	計算
R_AMD64_NONE	0	None	None
R_AMD64_64	1	word64	S + A
R_AMD64_PC32	2	word32	S + A - P
R_AMD64_GOT32	3	word32	G + A
R_AMD64_PLT32	4	word32	L + A - P
R_AMD64_COPY	5	None	この表のあとの説明を参照してください。
R_AMD64_GLOB_DAT	6	word64	S
R_AMD64_JUMP_SLOT	7	word64	S
R_AMD64_RELATIVE	8	word64	B + A
R_AMD64_GOTPCREL	9	word32	G + GOT + A - P
R_AMD64_32	10	word32	S + A
R_AMD64_32S	11	word32	S + A
R_AMD64_16	12	word16	S + A
R_AMD64_PC16	13	word16	S + A - P
R_AMD64_8	14	word8	S + A
R_AMD64_PC8	15	word8	S + A - P
R_AMD64_PC64	24	word64	S + A - P

表 7–16 x64: ELF 再配置型 (続き)

名前	値	フィールド	計算
R_AMD64_GOTOFF64	25	word64	S + A - GOT
R_AMD64_GOTPC32	26	word32	GOT + A + P

注 – スレッド固有領域の参照に使用できる再配置はほかにも存在します。これらの再配置については、第 8 章「スレッド固有領域 (TLS)」で説明しています。

これらの再配置型のほとんどの特別なセマンティクスは、x86 で使用されているものと同じです。いくつかの再配置型には、単純な計算を超えたセマンティクスが存在します。

R_AMD64_GOTPCREL
: この再配置のセマンティクスは R_AMD64_GOT32 または等しい R_386_GOTPC 再配置と異なります。x64 アーキテクチャーは、命令ポインタに対して相対的なアドレス指定モードを提供します。したがって、アドレスは 1 つの命令で GOT から読み込むことができます。

  R_AMD64_GOTPCREL 再配置の計算は、シンボルのアドレスを指定した GOT 内の位置と再配置を適用する位置の間の差を提供します。

R_AMD64_32
: 計算値は 32 ビットに切り捨てられます。リンカーは、再配置のために生成された値が元の 64 ビット値にゼロ拡張されていることを確認します。

R_AMD64_32S
: 計算値は 32 ビットに切り捨てられます。リンカーは、再配置のために生成された値が元の 64 ビット値に符号拡張されていることを確認します。

R_AMD64_8、R_AMD64_16、R_AMD64_PC16、R_AMD64_PC8
: これらの再配置は x64 ABI には準拠していませんが、文書化するためにここに追加しておきます。R_AMD64_8 再配置は、計算値を 8 ビットに切り詰めます。R_AMD64_16 再配置は、計算値を 16 ビットに切り詰めます。

文字列テーブルセクション

文字列テーブルセクションは、ヌル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。文字列をインデックスに使用して、文字列テーブルセクションを参照します。

先頭バイト (インデックス 0) は、ヌル文字を保持します。同様に、文字列テーブルの最後のバイトは、ヌル文字を保持します。したがって、すべての文字列は確実にヌ

ル文字で終了します。したがって、すべての文字列は確実にヌル文字で終了します。インデックスが 0 の文字列は、名前を指定しないかヌル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、セクションヘッダーの sh_size メンバーに 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 メンバーは、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。セクションヘッダー文字列テーブルは、ELF ヘッダーの e_shstrndx メンバーで示されます。次の図は、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。

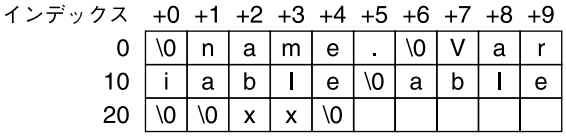


図 7–7 ELF 文字列テーブル

次の表に、上の図に示した文字列テーブルの文字列を示しています。

表 7–17 ELF 文字列テーブルインデックス

索引	文字列
0	None
1	name
7	Variable
11	able
16	able
24	ヌル文字列

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 回以上出現可能です。部分文字列に対する参照は存在可能です。単一文字列は複数回参照可能です。参照されない文字列も許可されます。

シンボルテーブルセクション

オブジェクトファイルのシンボルテーブルには、プログラムのシンボル定義とシンボル参照の検索と再配置に必要となる情報が格納されます。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。表 7–21 を参照してください。

シンボルテーブルエントリの形式は、次のとおりです。sys/elf.h を参照してください。

```
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
: 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)。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。

st_value
: 関連付けられているシンボルの値。この値は、状況に応じて絶対値またはアドレスを表します。274 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、オブジェクトに存在するバイト数です。このメンバーは、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、値 0 を保持します。

st_info
: シンボルの種類および結び付けられる属性。値と意味のリストを、表 7–18 に示します。次のコードは、値の処理方法を示します。sys/elf.h を参照してください。

```
#define ELF32_ST_BIND(info)          ((info) >> 4)
#define ELF32_ST_TYPE(info)          ((info) & 0xf)
#define ELF32_ST_INFO(bind, type)    (((bind)<<4)+((type)&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
: シンボルの可視性。値と意味のリストを、表 7–20 に示します。次のコードは、32 ビットオブジェクトと 64 ビットオブジェクトの両方の値を操作する方法を示しています。その他のビットは 0 に設定され、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。このメンバーは、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 7–4 を参照してください。

  このメンバーに SHN_XINDEX が含まれる場合は、実際のセクションヘッダーインデックスが大きすぎてこのフィールドに入りません。実際の値は、タイプ SHT_SYMTAB_SHNDX の関連するセクション内に存在します。

シンボルのバインディングは、st_info フィールドで決定されますが、これにより、リンクの可視性と動作が決定します。

表 7–18 ELF シンボルのバインディング、(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
: 局所シンボル。局所シンボルは、局所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。同じ名前の局所シンボルは、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。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。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のために予約されています。

STB_LOPROC - STB_HIPROC
: この範囲の値は、プロセッサ固有のセマンティクスのために予約されています。

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。

- リンカーが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場合は、同じ名前を持つ複数の STB_GLOBAL シンボルは定義できません。ただし、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ません。リンカーは大域定義を使用し、ウィーク定義を無視します。

  同様に、共有シンボルが存在している場合にそれと同じ名前のウィークシンボルが現れても、エラーは発生しません。リンカーは共通定義を使用し、ウィーク定義を無視します。共通シンボルは、SHN_COMMON を保持する st_shndx フィールドを持ちます。44 ページの「シンボル解決」を参照してください。
- リンカーがアーカイブライブラリを検索すると、未定義または一時的な大域シンボル定義が存在するアーカイブメンバーが抽出されます。メンバーの定義は、大域シンボルまたはウィークシンボルになります。

  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メンバーを抽出しません。解決されていないウィークシンボルは、値 0 を持ちます。-z weakextract を使用すると、このデフォルトの動作が無効になります。このオプションを使用すると、ウィーク参照がアーカイブメンバーを抽出できます。

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 結合を持つ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。222 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メンバーは、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。

シンボルのタイプは st_info フィールドで指定され、これにより、関連付けられた実体の一般的な分類が決定されます。

表 7–19 ELF シンボルのタイプ (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_TLS	6
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、関数またはほか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 です。このシンボルは、存在する場合、ファイルのほかの 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、そのオブジェクトファイルを表す STT_FILE シンボルです。慣例により、このシンボルの後にはファイル STT_SECTION シンボルが続きます。これらのセクションシンボルの後には、ローカルになった大域シンボルが続きます。

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。このシンボルは、STT_OBJECT とまったく同じに扱われます。

STT_TLS
: シンボルは、スレッド固有領域の実体を指定します。定義されている場合、実際のアドレスではなく、シンボルに割り当てられたオフセットを提供します。

  スレッドローカルストレージの再配置では、タイプが STT_TLS のシンボルしか参照できません。割り当て可能なセクションからタイプが STT_TLS のシンボルを参照するには、特別なスレッドローカルストレージ再配置を使用するしか方法がありません。詳細は、第 8 章「スレッド固有領域 (TLS)」を参照してください。割り当てができないセクションからタイプが STT_TLS のシンボルを参照する際には、この制限はありません。

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のために予約されています。

STT_LOPROC - STT_HIPROC
: この範囲の値は、プロセッサ固有のセマンティクスのために予約されています。

シンボルの可視性は、その st_other フィールドで決まります。この可視性は、再配置可能オブジェクトで指定できます。シンボルの可視性により、シンボルが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た後のシンボルへのアクセス方法が定義されます。

表 7–20 ELF シンボルの可視性

名前	値
STV_DEFAULT	0
STV_INTERNAL	1
STV_HIDDEN	2

表 7–20 ELF シンボルの可視性 (続き)

名前	値
STV_PROTECTED	3

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。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) の外から見ることができます。局所シンボルは、「隠されて」います。大域シンボルおよびウィークシンボルは、横取りすることもできます。別のコンポーネントの同じ名前の定義によってこれらのシンボルに割り込むこともできます。

STV_PROTECTED
: 現在のコンポーネント内で定義されたシンボルは、それがほかのコンポーネント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。定義コンポーネント内からシンボルへの参照など、あらゆる参照について、コンポーネント内の定義に解決する必要があります。この解決は、シンボル定義がデフォルト規則によって割り込みを行う別のコンポーネント内に存在する場合も、実行する必要があります。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ト内で定義されたシンボルは、その名前がほかのコンポーネント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、コンポーネント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でも、ほかのコンポーネントから参照可能です。

  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、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。

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まった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、これらの属性は次の 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。

- 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、「その参照を満たす定義は、リンクされているオブジェクト内で提供されなければならない」ということを暗黙の条件としています。この種のシンボルの参

照がリンクされるオブジェク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ます。この場合、参照は 0 に解決されます。

- 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てい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。シンボルの特定のインスタンスに対して異なる可視性の属性が指定されている場合は、もっと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、もっとも制約の少ないものからもっと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メンバー 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ます。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。いくつかの特別なセクションインデックス値は、ほかのセマンティクスが付けられています。

SHN_ABS
: このシンボルは、再配置が行われても変わらない絶対値を持ちます。

SHN_COMMON および SHN_AMD64_LCOMMON
: このシンボルは、まだ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 メンバーに類似した整列制約を与えます。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示します。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ほか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。このエントリは次の値を保持します。

表 7–21 ELF シンボルテーブルエントリ:インデックス 0

名前	値	注意
st_name	0	名前が存在しない
st_value	0	値は 0
st_size	0	サイズが存在しない
st_info	0	種類はない。ローカル結合
st_other	0	

表 7–21 ELF シンボルテーブルエントリ:インデックス 0 (続き)

名前	値	注意
st_shndx	SHN_UNDEF	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 メンバー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します。
- 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します。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になります。
- 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では、st_value は仮想アドレスを保持します。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) が使用されます。

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。

シンボルテーブルのレイアウトと規則

シンボルテーブル内のシンボルは、次の順序で書き込まれます。

- シンボルテーブルのインデックス 0 は、未定義のシンボルを表現するために使用されます。このシンボルテーブルの最初のエントリは常に、すべてゼロです。つまり、シンボルタイプは STT_NOTYPE です。
- シンボルテーブルに局所シンボルが含まれている場合、そのシンボルテーブルの 2 番目のエントリは、ファイルの名前を示す STT_FILE シンボルです。
- STT_SECTION タイプのセクションシンボル。
- STT_REGISTER タイプのレジスタシンボル。
- ローカルスコープに制限されている大域シンボル。
- 局所シンボルを使用する入力ファイルの場合は、入力ファイルの名前を示す STT_FILE シンボルとその局所シンボル。
- 大域シンボルのすぐあとに、シンボルテーブル内の局所シンボルが書き込まれます。最初の大域シンボルは、シンボルテーブルの sh_info 値によって識別されます。局所シンボルと大域シンボルは常にこの方法で別々に処理されるので、混在する可能性はありません。

Solaris OS には、2 つ の特別なシンボルテーブルがあります。

.symtab (SHT_SYMTAB)
このシンボルテーブルには、関連する ELF ファイルを示すあらゆるシンボルが入っています。このシンボルテーブルは、通常は割り当てることができないため、プロセスのメモリーイメージ内では使用できません。

ELIMINATE キーワードと一緒に mapfile を使用すると、.symtab から大域シンボルを削除できます。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。リンカーの -z redlocsym オプションを使用して、ローカルシンボルを削除することもできます。

.dynsym (SHT_DYNSYM)
このテーブルには、.symtab テーブルのシンボルのうち、動的リンクをサポートするために必要なシンボルだけが入っています。このシンボルテーブルは、割り当てることができるため、プロセスのメモリーイメージ内で使用できます。

.dynsym テーブルは標準 NULL シンボルで始まり、そのあとに大域シンボルが続きます。STT_FILE シンボルは通常、このシンボルテーブルにはありません。STT_SECTION シンボルは、再配置エントリが必要とする場合に存在する可能性があります。

レジスタシンボル

SPARC アーキテクチャーは、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、次の値が入ります。

表 7–22 SPARC: ELF 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
st_name	シンボル名文字列テーブルへのインデックス。または 0 (スクラッチレジスタ)。
st_value	レジスタ番号。整数レジスタの割り当てについては、ABI マニュアルを参照してください。
st_size	未使用 (0)。
st_info	結合は標準的には STB_GLOBAL です。種類は STT_SPARC_REGISTER でなければなりません。
st_other	未使用 (0)。
st_shndx	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は、SHN_ABS。それ以外の場合は、SHN_UNDEF。

定義済みの SPARC 用レジスタ値を、次に示します。

表 7–23 SPARC: ELF レジスタ番号

名前	値	意味
STO_SPARC_REGISTER_G2	0x2	%g2
STO_SPARC_REGISTER_G3	0x3	%g3

特定の大域レジスタのエントリが存在しないことは、その特定の大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 テーブルセクション

syminfo 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。.SUNW_syminfo セクションには、関連付けられているシンボルテーブル (sh_link) のエントリごとに 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 エントリまたは 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用に予約されるので、矛盾は発生しません。

Syminfo エントリの形式は、次のとおりです。sys/link.h を参照してください。

```
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_boundto
: .dynamic セクションのエントリへのインデックスで、sh_info フィールドにより示され、Syminfo フラグを増加させます。たとえば、DT_NEEDED エントリは、Syminfo エントリに関連付けられた動的オブジェクトを示します。次の表に示すエントリは、si_boundto に対して予約されています。

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です。
SYMINFO_BT_NONE	0xfffd	シンボルに特別なシンボル結合は含まれません。

si_flags
このビットフィールドでは、次の表に示すフラグを設定できます。

名前	値	意味
SYMINFO_FLG_DIRECT	0x01	シンボル参照は、定義を含むオブジェクトへ直接関連付けられます。
SYMINFO_FLG_COPY	0x04	シンボル定義はコピー再配置の結果です。
SYMINFO_FLG_LAZYLOAD	0x08	遅延読み込みの必要があるオブジェクトに対するシンボル参照です。
SYMINFO_FLG_DIRECTBIND	0x10	シンボル参照は定義に直接結合される必要があります。
SYMINFO_FLG_NOEXTDIRECT	0x20	外部参照はこのシンボル定義に直接結合できません。

バージョン管理セクション

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す。

- 「バージョン定義」は大域シンボルの関連付けを提供し、タイプ SHT_SUNW_verdef と SHT_SUNW_versym のセクションを使って実装されます。
- 「バージョン依存」はほかのオブジェクト依存関係からのバージョン定義要件を示し、タイプ SHT_SUNW_verneed のセクションを使って実装されます。

これらのセクションを形成する構造体は、sys/link.h 内で定義されています。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、タイプ SHT_SUNW_verdef によって定義されます。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 つの構造体は、ファイル内にシンボルとバージョン定義の関連付けを提供します。157 ページの「バージョン定義の作成」を参照してください。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      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;
        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
: このメンバーは、構造体のバージョンを示します (次の表を参照)。

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、より大きい番号の新しいバージョンが必要です。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
このメンバーは、バージョン定義に固有の情報を保持します (次の表を参照)。

名前	値	意味
VER_FLG_BASE	0x1	ファイルのバージョン定義。
VER_FLG_WEAK	0x2	ウィークバージョン識別子。

基本バージョン定義は、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。基本バージョンは、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。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。161 ページの「ウィークバージョン定義の作成」を参照してください。

vd_ndx
バージョンインデックス。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
Elf32_Verdaux 配列の要素数。

vd_hash
バージョン定義名のハッシュ値。この値は、247 ページの「ハッシュテーブルセクション」に記述されているのと同じハッシング機能により生成されます。

vd_aux
この Elf32_Verdef エントリの先頭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。配列の先頭要素は存在しなければなりません。この要素は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。追加要素は存在可能です。要素の番号は 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 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。

vda_name
ヌル文字で終わる文字列への文字列テーブルオフセットで、バージョン定義名を指定します。

vda_next
　このElf32_Verdaux エントリの先頭から次のElf32_Verdaux エントリまでのバイトオフセット。

バージョン依存セクション

バージョン依存セクションは、タイプSHT_SUNW_verneed によって定義されます。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
　このメンバーは、構造体のバージョンを示します (次の表を参照)。

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、より大きい番号の新しいバージョンが必要です。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
Elf32_Vernaux 配列の要素数。

vn_file
ヌル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で、バージョン依存性のファイル名を指定します。この名前は、ファイル内に存在する .dynamic 依存性のどれかに一致します。296 ページの「動的セクション」を参照してください。

vn_aux
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_Vernaux 配列までのバイトオフセット。少なくとも 1 つのバージョン依存性が存在する必要があります。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 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。

vna_hash
バージョン依存性の名前のハッシュ値。この値は、247 ページの「ハッシュテーブルセクション」に記述されているのと同じハッシング機能により生成されます。

vna_flags
バージョン依存性に固有の情報 (次の表を参照)。

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。

vna_other
現在、使用されていません。

vna_name
ヌル文字で終わる文字列への文字列テーブルオフセット。バージョン依存性の名前を与えます。

vna_next
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。

バージョンシンボルセクション

バージョンシンボルセクションは、タイプ SHT_SUNW_versym によって定義されます。このセクションは、次の構造を持つ要素配列で構成されます。

```
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に等しくなければなりません。この値は、セクションの sh_link 値で決定されます。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次の表に示す値をとることができます。

表 7–24 ELF 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します。
VER_NDX_GLOBAL	1	シンボルに大域適用範囲が存在し、ベースバージョン定義に割り当てられています。
	>1	シンボルに大域適用範囲が存在し、ユーザー定義バージョン定義に割り当てられています。

VER_NDX_GLOBAL よりも大きいインデックス値はどれも、SHT_SUNW_verdef セクション内の特定のエントリの vd_ndx 値に対応します。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。

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。ここで説明する情報の大半は、すべてのシステムに適用されます。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、アプリケーション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、スタックなどがあるセグメントが存在します。次の主な細目があります。

- 283 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します。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します。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ほかの情報が存在します。
- 289 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します。
- 295 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します。

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準備するためにシステムが必要とするセグメントなど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、289 ページの「セグメントの内容」で説明しているように、1 つ以上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` メンバーと `e_phnum` メンバーで指定されます。

プログラムヘッダーの構造体は、次のとおりです。`sys/elf.h` を参照してください。

```
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`
この配列要素が記述するセグメント型、または配列要素情報の解釈方法。型の値とその意味は、表 7–25 を参照してください。

`p_offset`
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。

`p_vaddr`
セグメントの先頭バイトが存在するメモリー内の仮想アドレス。

`p_paddr`
セグメントの物理アドレス (物理アドレス指定が適切なシステムの場合)。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、このメンバー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`
セグメントのファイルイメージのバイト数 (0 の場合もある)。

`p_memsz`
セグメントのメモリーイメージのバイト数 (0 の場合もある)。

`p_flags`
セグメントに関係するフラグ。型の値とその意味は、表 7–26 を参照してください。

`p_align`
読み込み可能なプロセスセグメントは、ページサイズを基にして、`p_vaddr` と `p_offset` に対して同じ値を保持する必要があります。このメンバーは、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ず、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。289 ページの「プログラムの読み込み (プロセッサ固有)」を参照してください。

エントリの中には、プロセスセグメントを記述するものもあります。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、明示されている場合を除き任意です。定義されている型の値を、次の表に示します。

表 7–25 ELFセグメント型

名前	値
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_TLS	7
PT_LOOS	0x60000000
PT_SUNW_UNWIND	0x6464e550
PT_LOSUNW	0x6ffffffa
PT_SUNWBSS	0x6ffffffa
PT_SUNWSTACK	0x6ffffffb
PT_SUNWDTRACE	0x6ffffffc
PT_SUNWCAP	0x6ffffffd
PT_HISUNW	0x6fffffff
PT_HIOS	0x6fffffff
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

PT_NULL
: 未使用メンバーの値は不定です。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
: p_filesz と p_memsz により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られます。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 を保持するように定義されます。これらのバイトはセグメントの初期化領域に続きます。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昇順に現れ、p_vaddr メンバーでソートされます。

PT_DYNAMIC
: 動的リンクに関する情報を指定します。296 ページの「動的セクション」を参照してください。

PT_INTERP
: インタプリタとして呼び出される、ヌル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。動的実行可能ファイルの場合、この型は必須です。共有オブジェクトの場合は、この型を指定することができます。この型は、ファイル内で複数指定することはできません。この型が存在する場合、この型はすべての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、295 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
: 補助情報の位置とサイズを指定します。詳細は、251 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、予約済みですが、セマンティクスは定義されていません。

PT_PHDR
: プログラムヘッダーテーブルの、ファイル、および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。このセグメント型を、ファイル内に複数指定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かぎり指定できます。この型が存在する場合、この型はすべての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、295 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_TLS
: スレッド固有領域のテンプレートを指定します。詳細は、327 ページの「スレッド固有領域 (TLS) セクション」を参照してください。

PT_LOOS - PT_HIOS
: この範囲の値 (両端の値を含む) は、OS 固有のセマンティクスのために予約されています。

PT_SUNW_UNWIND
: このセグメントは、スタック巻き戻し (unwind) テーブルを含んでいます。

PT_LOSUNW - PT_HISUNW
: この範囲の値 (両端の値を含む) は、Sun 固有のセマンティクスのために予約されています。

PT_SUNWBSS
PT_LOAD 要素と同じ属性で、.SUNW_bss セクションの記述に使用します。

PT_SUNWSTACK
プロセススタックを記述します。PT_SUNWSTACK 要素は 1 つのみ存在できます。p_flags フィールドで定義されたアクセス権のみが意味を持ちます。

PT_SUNWDTRACE
dtrace(1M) の内部使用のため予約されています。

PT_SUNWCAP
ハードウェア機能要件を指定します。詳細は、245 ページの「ハードウェアおよびソフトウェア機能に関するセクション」を参照してください。

PT_LOPROC - PT_HIPROC
この範囲の値 (両端の値を含む) は、プロセッサ固有のセマンティクスのために予約されています。

注 – ほかの箇所で特に要求されないかぎり、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。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係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。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。289 ページの「プログラムの読み込み (プロセッサ固有)」を参照してください。

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、この制限はファイル形式による要件ではあ

りません)。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 メンバー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のために予約されます。

表 7–26 ELF セグメントフラグ

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	指定なし

アクセス権ビットが 0 の場合、そのビット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かぎり、セグメントは書き込み権を持ちません。次の表に、正確なフラグ解釈と許容されるフラグ解釈を示します。

表 7–27 ELF セグメントへのアクセス権

フラグ	値	正確なフラグ解釈	許容されるフラグ解釈
None	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
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。次に、セグメントの内容を一般的な言葉で説明します。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。

テキストセグメントには、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 7–10 を参照してください。

`PT_DYNAMIC` プログラムヘッダー要素は、`.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、位置独立のコードと動的リンクに関係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、311 ページの「大域オフセットテーブル (プロセッサ固有)」と312 ページの「プロシージャーのリンクテーブル (プロセッサ固有)」を参照してください。

タイプ `SHT_NOBITS` のセクションは、ファイル内の領域を占有しませんが、セグメントのメモリーイメージには反映されます。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かぎり物理ページを必要としません。プロセスは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、システム性能を向上させることができます。この効率性を実現するには、実行可能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する必要があります。

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、1M バイト (`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。プログラム全体は、テキスト、データ、ヒープ、スタック、および共用オブジェクト依存関係を含めて、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いことをより簡単に確認できます。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、リンカーに mapfile および -M オプションを使用することにより、プログラムを4Gバイト未満でリンクすることも可能です。詳細は、/usr/lib/ld/sparcv9/map.below4G を参照してください。

次の図に、SPARCバージョンの実行可能ファイルの例を示します。

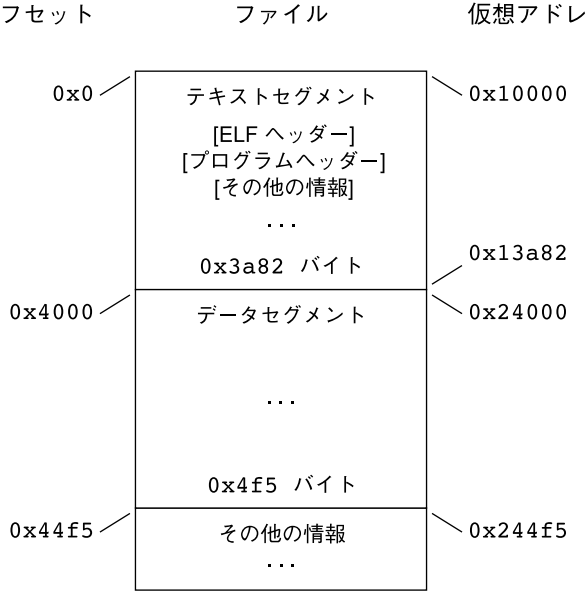


図 7–8 SPARC: 実行可能ファイル(64Kに整列)

次の表に、前の図に示した読み込み可能セグメント要素の定義を示します。

表 7–28 SPARC: ELFプログラムヘッダーセグメント(64Kに整列)

メンバー	テキスト	データ
p_type	PT_LOAD	PT_LOAD
p_offset	0x0	0x4000
p_vaddr	0x10000	0x24000

表 7–28　SPARC: ELF プログラムヘッダーセグメント (64K に整列)　(続き)

メンバー	テキスト	データ
p_paddr	指定なし	指定なし
p_filesize	0x3a82	0x4f5
p_memsz	0x3a82	0x10a4
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x10000	0x10000

次の図に、x86 バージョンの実行可能ファイルの例を示します。

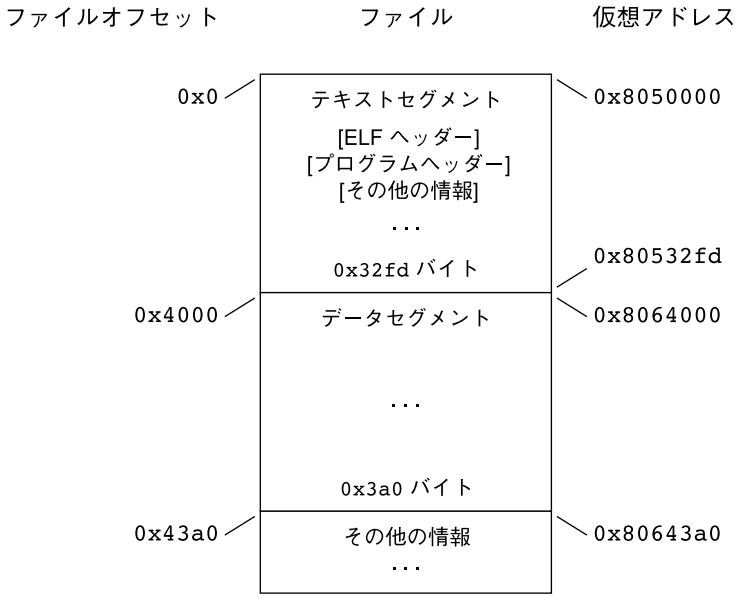


図 7–9　32 ビット x86: 実行可能ファイル (64K に整列)

次の表に、前の図に示した読み込み可能セグメント要素の定義を示します。

表 7–29　32 ビット x86: ELF プログラムヘッダーセグメント (64K に整列)

メンバー	テキスト	データ
p_type	PT_LOAD	PT_LOAD
p_offset	0x0	0x4000

表 7–29 32 ビット x86: ELF プログラムヘッダーセグメント (64K に整列) (続き)

メンバー	テキスト	データ
p_vaddr	0x8050000	0x8064000
p_paddr	指定なし	指定なし
p_filesize	0x32fd	0x3a0
p_memsz	0x32fd	0xdc4
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x10000	0x10000

例に示した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) に、純粋ではないテキストやデータが含まれます。

- 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、およびほかの情報が存在します。
- 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します。
- 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します。
- 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ます。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ーアクセス権を設定します。セグメントのアドレスは調整され、アドレス空間の各論理ページに同じアクセス権セットが確実に存在するようになります。前の例では、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ます。

注 – 前の例は、テキストセグメントを丸めた、典型的な Solaris OS のバイナリを反映したものです。

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)。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してはならない)。

ほかの 3 ページに含まれる純粋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粋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 4K バイト (0x1000) ページを使用する例を、次の図に示します。単純化するために次の図では、1 ページのサイズのみを示しています。

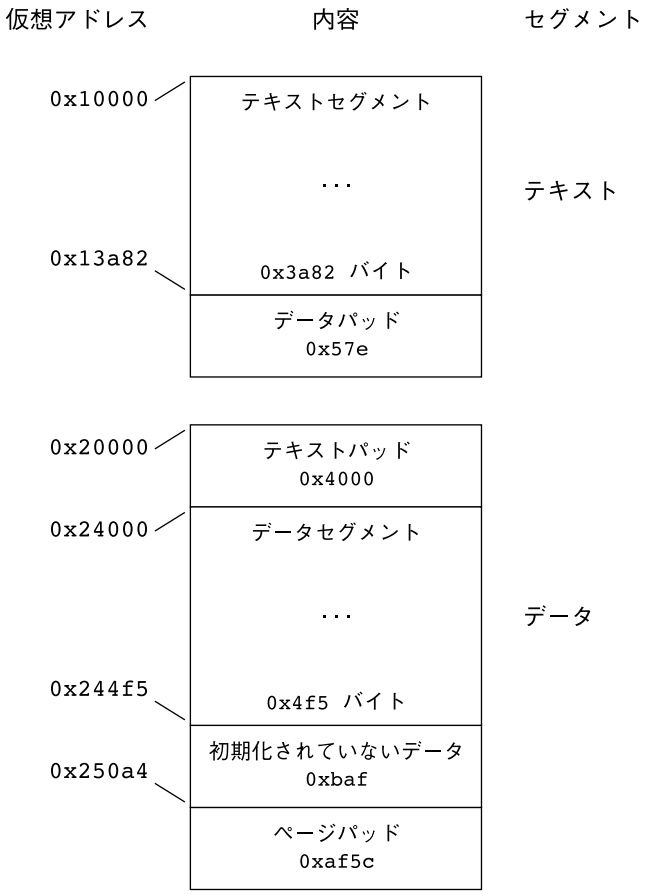


図 7–10　32 ビット SPARC: プロセスイメージセグメント

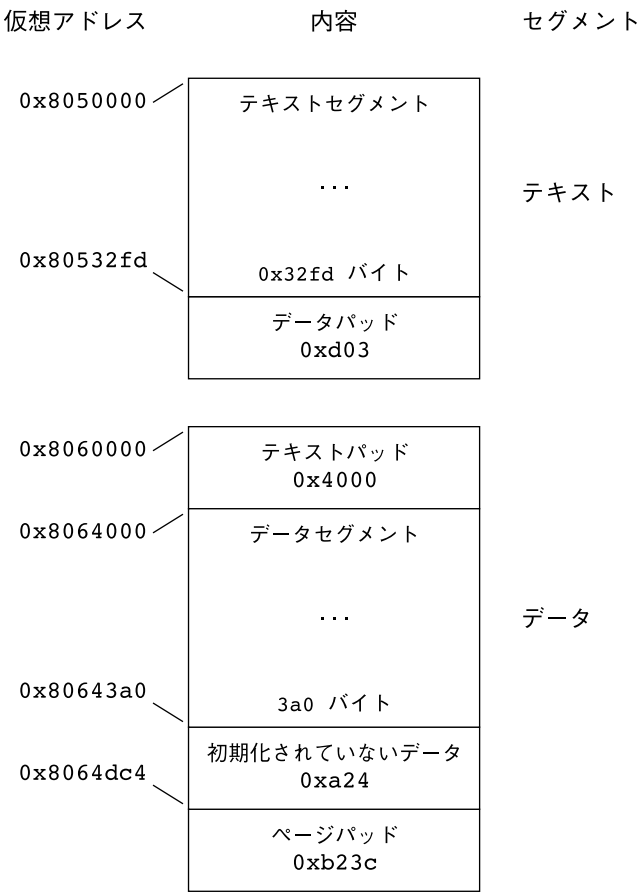


図 7–11 x86: プロセスイメージセグメント

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。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、位置独立のコードが存在します。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間で変化させることができます。

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。位置独立の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。

次の表は、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ての例で、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 7–30　32 ビット SPARC: ELF 共有オブジェクトセグメントアドレスの例

送信元	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 7–31　32 ビット x86: ELF 共有オブジェクトセグメントアドレスの例

送信元	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

動的リンクを開始する動的実行可能ファイルまたは共有オブジェクトは、1 つの PT_INTERP プログラムヘッダー要素を保持できます。システムは exec(2) の実行中に、PT_INTERP セグメントからパス名を取り出し、その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。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提供する必要があります。

Solaris OS では、インタプリタは実行時リンカー ld.so.1(1) として知られています。

実行時リンカー

リンカーは、動的リンクを開始する動的オブジェクトを作成する際、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ます。この要素は、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。

リンカーはまた、実行時リンカーを支援する、実行可能ファイルと共有オブジェクトファイル用のさまざまなデータを作成します。これらのデータは読み込み可能セグメントに存在するため、データを実行時に使用できます。これらのセグメントには、次のものが含まれます。

- タイプ SHT_DYNAMIC の .dynamic セクション。このセクションにはさまざまなデータが格納されます。このセクションの始まりに存在する構造体には、ほかの動的リンク情報のアドレスが存在します。
- タイプ SHT_PROGBITS の .got セクションと .plt セクション。このセクションには独立した次の 2 つのテーブルが格納されます。大域オフセットテーブルとプロシージャーリンクテーブルです。これ以降の節では、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および変更するかを説明します。
- タイプ SHT_HASH の .hash セクション。このセクションにはシンボルハッシュテーブルが格納されます。

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を占有することが可能です。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。

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、このセクションには、次の構造体を持つ配列が存在します。sys/link.h を参照してください。

```
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します。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。

各動的タグの値 (2 つの特別な互換性範囲を除く) で d_un union の解釈を決定します。この規則は、外部ツールによる動的タグの解釈をよりシンプルにします。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。

次の表は、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 7–32　ELF 動的配列タグ

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_NULL	0	無視される	必須	必須
DT_NEEDED	1	d_val	任意	任意
DT_PLTRELSZ	2	d_val	任意	任意
DT_PLTGOT	3	d_ptr	任意	任意
DT_HASH	4	d_ptr	必須	必須
DT_STRTAB	5	d_ptr	必須	必須
DT_SYMTAB	6	d_ptr	必須	必須
DT_RELA	7	d_ptr	必須	任意
DT_RELASZ	8	d_val	必須	任意
DT_RELAENT	9	d_val	必須	任意

表 7–32 ELF 動的配列タグ (続き)

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_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
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_ENCODING	32	指定なし	指定なし	指定なし
DT_PREINIT_ARRAY	32	d_ptr	任意	無視される
DT_PREINIT_ARRAYSZ	33	d_val	任意	無視される
DT_MAXPOSTAGS	34	指定なし	指定なし	指定なし
DT_LOOS	0x6000000d	指定なし	指定なし	指定なし
DT_SUNW_AUXILIARY	0x6000000d	d_ptr	指定なし	任意

表 7–32 ELF 動的配列タグ (続き)

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_SUNW_RTLDINF	0x6000000e	d_ptr	任意	任意
DT_SUNW_FILTER	0x6000000e	d_ptr	指定なし	任意
DT_SUNW_CAP	0x60000010	d_ptr	任意	任意
DT_HIOS	0x6ffff000	指定なし	指定なし	指定なし
DT_VALRNGLO	0x6ffffd00	指定なし	指定なし	指定なし
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_1	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
DT_VALRNGHI	0x6ffffdff	指定なし	指定なし	指定なし
DT_ADDRRNGLO	0x6ffffe00	指定なし	指定なし	指定なし
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_ADDRRNGHI	0x6ffffeff	指定なし	指定なし	指定なし
DT_RELACOUNT	0x6ffffff9	d_val	任意	任意
DT_RELCOUNT	0x6ffffffa	d_val	任意	任意
DT_FLAGS_1	0x6ffffffb	d_val	任意	任意
DT_VERDEF	0x6ffffffc	d_ptr	任意	任意
DT_VERDEFNUM	0x6ffffffd	d_val	任意	任意

表 7–32 ELF 動的配列タグ (続き)

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_VERNEED	0x6ffffffe	d_ptr	任意	任意
DT_VERNEEDNUM	0x6fffffff	d_val	任意	任意
DT_LOPROC	0x70000000	指定なし	指定なし	指定なし
DT_SPARC_REGISTER	0x70000001	d_val	任意	任意
DT_AUXILIARY	0x7ffffffd	d_val	指定なし	任意
DT_USED	0x7ffffffe	d_val	任意	任意
DT_FILTER	0x7fffffff	d_val	指定なし	任意
DT_HIPROC	0x7fffffff	指定なし	指定なし	指定なし

DT_NULL
: _DYNAMIC 配列の終わりを示します。

DT_NEEDED
: ヌル文字で終わっている文字列の DT_STRTAB 文字列テーブルオフセットであり、必要な依存性の名前を示します。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、ほか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。78 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_PLTRELSZ
: プロシージャー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)。312 ページの「プロシージャーのリンクテーブル (プロセッサ固有)」を参照してください。

DT_PLTGOT
: プロシージャー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るアドレス。312 ページの「プロシージャーのリンクテーブル (プロセッサ固有)」と311 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。

DT_HASH
: シンボルハッシュテーブルのアドレス。このテーブルは、DT_SYMTAB 要素で示されるシンボルテーブルを参照します。247 ページの「ハッシュテーブルセクション」を参照してください。

DT_STRTAB
: 文字列テーブルのアドレス。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、およびほかの文字列が存在します。265 ページの「文字列テーブルセクション」を参照してください。

DT_SYMTAB
シンボルテーブルのアドレス。267 ページの「シンボルテーブルセクション」を参照してください。

DT_RELA
再配置テーブルのアドレス。252 ページの「再配置セクション」を参照してください。

オブジェクトファイルには、複数の再配置セクションを指定できます。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独立している場合がありますが、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、DT_RELASZ 要素と DT_RELAENT 要素も存在する必要があります。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。

DT_RELASZ
DT_RELA 再配置テーブルの合計サイズ (単位: バイト)。

DT_RELAENT
DT_RELA 再配置エントリのサイズ (単位: バイト)。

DT_STRSZ
DT_STRTAB 文字列テーブルの合計サイズ (単位: バイト)。

DT_SYMENT
DT_SYMTAB シンボルエントリのサイズ (単位: バイト)。

DT_INIT
初期化関数のアドレス。41 ページの「初期設定および終了セクション」を参照してください。

DT_FINI
終了関数のアドレス。41 ページの「初期設定および終了セクション」を参照してください。

DT_SONAME
ヌル文字で終わっている文字列の DT_STRTAB 文字列テーブルオフセットで、共有オブジェクトの名前を示します。123 ページの「共有オブジェクト名の記録」を参照してください。

DT_RPATH
ヌル文字で終わっているライブラリ検索パス文字列の DT_STRTAB 文字列テーブルオフセット。この要素の使用は、DT_RUNPATH に置き換えられました。79 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。

DT_SYMBOLIC
: オブジェクトが、リンク編集中に適用されたシンボリック結合を含むことを示します。この要素の使用は、DF_SYMBOLIC フラグに置き換えられました。150 ページの「-B symbolic オプションの使用」を参照してください。

DT_REL
: DT_RELA に似ていますが、テーブルに暗黙の加数が存在する点が異なります。この要素が存在する場合、DT_RELSZ 要素と DT_RELENT 要素も存在する必要があります。

DT_RELSZ
: DT_REL 再配置テーブルの合計サイズ (単位: バイト)。

DT_RELENT
: DT_REL 再配置エントリのサイズ (単位: バイト)。

DT_PLTREL
: プロシージャーのリンクテーブルが参照する再配置エントリの型 (DT_REL または DT_RELA) を示します。1 つのプロシージャー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。312 ページの「プロシージャーのリンクテーブル (プロセッサ固有)」を参照してください。この要素が存在する場合、DT_JMPREL 要素も存在する必要があります。

DT_DEBUG
: デバッグに使用されます。

DT_TEXTREL
: 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できることを示します。この要素の使用は、DF_TEXTREL フラグに置き換えられました。139 ページの「位置独立のコード」を参照してください。

DT_JMPREL
: プロシージャー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。312 ページの「プロシージャーのリンクテーブル (プロセッサ固有)」を参照してください。これらの再配置エントリを分離しておくと、遅延結合が有効なオブジェクトの読み込み時に、実行時リンカーはこれらのエントリを無視できます。この要素が存在する場合、DT_PLTRELSZ 要素と DT_PLTREL 要素も存在する必要があります。

DT_POSFLAG_1
: 直後の DT_ 要素に適用されるさまざまな状態フラグ。表 7–35 を参照してください。

DT_BIND_NOW
: プログラムに制御を渡す前に、この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。環境または dlopen(3C) で指定された場

合、このエントリは遅延結合の使用指令よりも優先されます。この要素の使用は、DF_BIND_NOW フラグに置き換えられました。詳細は、87 ページの「再配置が実行されるとき」を参照してください。

DT_INIT_ARRAY
: 初期設定関数へのポインタの配列のアドレス。この要素が存在する場合、DT_INIT_ARRAYSZ 要素も存在する必要があります。41 ページの「初期設定および終了セクション」を参照してください。

DT_FINI_ARRAY
: 終了関数へのポインタの配列のアドレス。この要素が存在する場合、DT_FINI_ARRAYSZ 要素も存在する必要があります。41 ページの「初期設定および終了セクション」を参照してください。

DT_INIT_ARRAYSZ
: DT_INIT_ARRAY 配列の合計サイズ (単位: バイト)。

DT_FINI_ARRAYSZ
: DT_FINI_ARRAY 配列の合計サイズ (単位: バイト)。

DT_RUNPATH
: ヌル文字で終わっているライブラリ検索パス文字列の DT_STRTAB 文字列テーブルオフセット。79 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。

DT_FLAGS
: このオブジェクトに特有のフラグ値。表 7–33 を参照してください。

DT_ENCODING
: DT_ENCODING と等しいかそれより大きく、かつ DT_LOOS と等しいかそれより小さい動的タグ値は、 d_un union の解釈の規則に従います。

DT_PREINIT_ARRAY
: 「初期設定前」関数へのポインタの配列のアドレス。この要素が存在する場合、DT_PREINIT_ARRAYSZ 要素も存在する必要があります。この配列は、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、この配列は無視されます。41 ページの「初期設定および終了セクション」を参照してください。

DT_PREINIT_ARRAYSZ
: DT_PREINIT_ARRAY 配列の合計サイズ (単位: バイト)。

DT_MAXPOSTAGS
: 値が正である動的配列タグの数。

DT_LOOS - DT_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のために予約されています。このような値はすべて、d_un union の解釈の規則に従います。

DT_SUNW_AUXILIARY
ヌル文字で終わっている文字列の DT_STRTAB 文字列テーブルオフセットで、シンボル別の補助フィルティーを 1 つ以上指定します。131 ページの「補助フィルタの生成」を参照してください。

DT_SUNW_RTLDINF
実行時リンカーによる使用のために予約されています。

DT_SUNW_FILTER
ヌル文字で終わっている文字列の DT_STRTAB 文字列テーブルオフセットで、シンボル別の標準フィルティーを 1 つ以上指定します。128 ページの「標準フィルタの生成」を参照してください。

DT_SUNW_CAP
ハードウェアとソフトウェア機能セクションのアドレス。245 ページの「ハードウェアおよびソフトウェア機能に関するセクション」を参照してください。

DT_SYMINFO
シンボル情報テーブルのアドレス。この要素が存在する場合、DT_SYMINENT 要素と DT_SYMINSZ 要素も存在する必要があります。276 ページの「Syminfo テーブルセクション」を参照してください。

DT_SYMINENT
DT_SYMINFO 情報エントリのサイズ (単位: バイト)。

DT_SYMINSZ
DT_SYMINFO テーブルのサイズ (単位: バイト)。

DT_VERDEF
バージョン定義テーブルのアドレス。このテーブル内の要素には、文字列テーブル DT_STRTAB のインデックスが含まれます。この要素が存在する場合、DT_VERDEFNUM 要素も存在する必要があります。278 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。

DT_VERDEFNUM
DT_VERDEF テーブルのエントリ数。

DT_VERNEED
バージョン依存性テーブルのアドレス。このテーブル内の要素には、文字列テーブル DT_STRTAB のインデックスが含まれます。この要素が存在する場合、DT_VERNEEDNUM 要素も存在する必要があります。280 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。

DT_VERNEEDNUM
DT_VERNEEDNUM テーブルのエントリ数。

DT_RELACOUNT
すべての Elf32_Rela または Elf64_Rela 再配置の連結から生成される RELATIVE 再配置回数を示します。146 ページの「再配置セクションの結合」を参照してください。

DT_RELCOUNT
すべての Elf32_Rel 再配置の連結から生成される RELATIVE 再配置回数を示します。146 ページの「再配置セクションの結合」を参照してください。

DT_AUXILIARY
ヌル文字で終わっている DT_STRTAB 文字列テーブルオフセットで、1 つ以上の補助フィルティーを指定します。131 ページの「補助フィルタの生成」を参照してください。

DT_FILTER
ヌル文字で終わっている DT_STRTAB 文字列テーブルオフセットで、1 つ以上の標準「フィルティー」を指定します。128 ページの「標準フィルタの生成」を参照してください。

DT_CHECKSUM
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。gelf_checksum(3ELF) のマニュアルページを参照してください。

DT_MOVEENT
DT_MOVETAB 移動エントリのサイズ (単位: バイト)。

DT_MOVESZ
DT_MOVETAB テーブルの合計サイズ (単位: バイト)。

DT_MOVETAB
移動テーブルのアドレス。この要素が存在する場合、DT_MOVEENT 要素と DT_MOVESZ 要素も存在する必要があります。248 ページの「移動セクション」を参照してください。

DT_CONFIG
ヌル文字で終わっている DT_STRTAB 文字列テーブルオフセットで、構成ファイルを定義します。構成ファイルは、実行可能ファイルでのみ有効であり、通常このオブジェクトに固有のファイルです。81 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。

DT_DEPAUDIT
ヌル文字で終わっている DT_STRTAB 文字列テーブルオフセットで、1 つ以上の監査ライブラリを定義します。185 ページの「実行時リンカーの監査インタフェース」を参照してください。

DT_AUDIT
ヌル文字で終わっている DT_STRTAB 文字列テーブルオフセットで、1 つ以上の監査ライブラリを定義します。185 ページの「実行時リンカーの監査インタフェース」を参照してください。

DT_FLAGS_1
このオブジェクトに特有のフラグ値。表 7–34 を参照してください。

DT_FEATURE_1
このオブジェクト特有の機能を示す値。表 7–36 を参照してください。

DT_VALRNGLO - DT_VALRNGHI
この範囲の値 (両端の値を含む) は、動的構造体の d_un.d_val フィールドによって使用されます。

DT_ADDRRNGLO - DT_ADDRRNGHI
この範囲の値 (両端の値を含む) は、動的構造体の d_un.d_ptr フィールドによって使用されます。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
DT_SYMTAB シンボルテーブル内の STT_SPARC_REGISTER シンボルのインデックス。シンボルテーブルの各 STT_SPARC_REGISTER シンボルには、1 つの動的エントリが存在します。275 ページの「レジスタシンボル」を参照してください。

DT_LOPROC - DT_HIPROC
この範囲の値は、プロセッサ固有のセマンティクスのために予約されています。

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。

表 7–33 ELF 動的フラグ DT_FLAGS

名前	値	意味
DF_ORIGIN	0x1	$ORIGIN 処理が必要です
DF_SYMBOLIC	0x2	シンボリックシンボル解決が必要です
DF_TEXTREL	0x4	テキストの再配置が存在します
DF_BIND_NOW	0x8	非遅延結合が必要です
DF_STATIC_TLS	0x10	オブジェクトは静的なスレッド固有領域方式を使用します

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。390 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、リンク編集中に適用されたシンボリック結合を含むことを示します。150 ページの「-B symbolic オプションの使用」を参照してください。

DF_TEXTREL
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できることを示します。139 ページの「位置独立のコード」を参照してください。

DF_BIND_NOW
: プログラムに制御を渡す前に、この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。環境または dlopen(3C) で指定された場合、このエントリは遅延結合の使用指令よりも優先されます。詳細は、87 ページの「再配置が実行されるとき」を参照してください。

DF_STATIC_TLS
: 静的なスレッド固有領域方式を使用するコードがオブジェクトに含まれていることを示します。静的なスレッド固有領域は、dlopen(3C) または遅延読み込みを使用して動的に読み込まれるオブジェクトでは使用すべきではありません。

表 7–34 ELF 動的フラグ DT_FLAGS_1

名前	値	意味
DF_1_NOW	0x1	完全な再配置処理を行います。
DF_1_GLOBAL	0x2	未使用
DF_1_GROUP	0x4	オブジェクトがグループのメンバーであることを示します。
DF_1_NODELETE	0x8	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。
DF_1_LOADFLTR	0x10	フィルティーの即時読み込みを保証します。
DF_1_INITFIRST	0x20	オブジェクトの初期化を最初に実行します。
DF_1_NOOPEN	0x40	オブジェクトを dlopen(3C) で使用できません。
DF_1_ORIGIN	0x80	$ORIGIN 処理が必要です。
DF_1_DIRECT	0x100	直接結合が有効です。
DF_1_INTERPOSE	0x400	オブジェクトは割り込み処理です。
DF_1_NODEFLIB	0x800	デフォルトのライブラリ検索パスを無視します。
DF_1_NODUMP	0x1000	オブジェクトを dldump(3C) 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	「フィルティー」がフィルタの検索を終了します。
DF_1_DISPRELDNE	0x8000	ディスプレイスメント再配置が実行されました。
DF_1_DISPRELPND	0x10000	ディスプレイスメント再配置の保留。

表 7–34 ELF 動的フラグ DT_FLAGS_1 (続き)

名前	値	意味
DF_1_NODIRECT	0x20000	オブジェクトは間接的な結合を含みます。
DF_1_IGNMULDEF	0x40000	内部使用。
DF_1_NOKSYMS	0x80000	内部使用。
DF_1_NOHDR	0x100000	内部使用。
DF_1_NORELOC	0x400000	内部使用。
DF_1_GLOBAUDIT	0x1000000	大域監査を確立します。

DF_1_NOW
: プログラムに制御を渡す前に、この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。環境または dlopen(3C) で指定された場合、このフラグは遅延結合の使用指令よりも優先されます。詳細は、87 ページの「再配置が実行されるとき」を参照してください。

DF_1_GROUP
: オブジェクトがグループのメンバーであることを示します。このフラグは、リンカーの -B group オプションを使用してオブジェクトに記録されます。109 ページの「オブジェクト階層」を参照してください。

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。オブジェクトは、dlopen(3C) 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(3C) で読み込み解除できません。このフラグは、リンカーの -z nodelete 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_LOADFLTR
: フィルタに対してのみ意味があります。関連付けられているすべてのフィルティーがただちに処理されることを示します。このフラグは、リンカーの -z loadfltr オプションを使用してオブジェクトに記録されます。135 ページの「「フィルティー」の処理」を参照してください。

DF_1_INITFIRST
: 読み込まれたほかのオブジェクトよりも先に、このオブジェクトの初期化セクションが実行されることを示します。このフラグは特殊なシステムライブラリでのみ使用するもので、リンカーの -z initfirst 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_NOOPEN
: dlopen(3C) を使ってオブジェクトを実行中のプロセスに追加できないことを示します。このフラグは、リンカーの -z nodlopen 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_ORIGIN
: オブジェクトに $ORIGIN 処理が必要であることを示します。390 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
: オブジェクトが直接結合情報を使用することを示します。86 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の割り込みが、一次読み込みオブジェクト (通常は実行可能ファイル) 以外のすべてのシンボルの前で発生します。このフラグは、リンカーの -z interpose オプションを使用して記録されます。85 ページの「実行時割り込み」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際、デフォルトのライブラリ検索パスがすべて無視されることを示します。このフラグは、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。40 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。

DF_1_NODUMP
: このオブジェクトが dldump(3C) によってダンプされないことを示します。このオプションの候補には、再配置を保持しないオブジェクトが含まれ、これらのオブジェクトは、crle(1) を使用して代替オブジェクトを生成する際に含めることができます。このフラグは、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
: このオブジェクトが、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。このフラグ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
: 「フィルティー」に対してのみ意味があります。以降の「フィルティー」に対するフィルタ検索は行われません。このフラグは、リンカーの -z endfiltee オプションを使用してオブジェクトに記録されます。388 ページの「「フィルティー」検索の縮小」を参照してください。

DF_1_DISPRELDNE
: このオブジェクトにディスプレイス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レコードは破棄されるため、オブジェクト内のディスプレイスメント再配置レコードはもはや存在しません。72 ページの「ディスプレイスメント再配置」を参照してください。

DF_1_DISPRELPND
: このオブジェクトのディスプレイスメント再配置が保留されていることを示します。ディスプレイスメント再配置はオブジェクト内部で終了するため、再配置は実行時に完了できます。72 ページの「ディスプレイスメント再配置」を参照してください。

DF_1_NODIRECT
　このオブジェクトに、直接結合できないシンボルが含まれることを示します。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。

DF_1_IGNMULDEF
　カーネルの実行時リンカーによる使用のために予約されています。

DF_1_NOKSYMS
　カーネルの実行時リンカーによる使用のために予約されています。

DF_1_NOHDR
　カーネルの実行時リンカーによる使用のために予約されています。

DF_1_NORELOC
　カーネルの実行時リンカーによる使用のために予約されています。

DF_1_GLOBAUDIT
　動的実行可能ファイルで大域監査が必要であることを示します。189 ページの「大域監査の記録」を参照してください。

表 7–35　ELF 動的位置フラグ DT_POSFLAG_1

名前	値	意味
DF_P1_LAZYLOAD	0x1	遅延読み込みされた依存関係を示します。
DF_P1_GROUPPERM	0x2	グループの依存関係を示します。

DF_P1_LAZYLOAD
　後続の DT_NEEDED エントリが遅延読み込み対象のオブジェクトであることを示します。このフラグは、リンカーの -z lazyload オプションを使用してオブジェクトに記録されます。91 ページの「動的依存関係の遅延読み込み」を参照してください。

DF_P1_GROUPPERM
　後続の DT_NEEDED エントリがグループとして読み込まれるオブジェクトであることを示します。このフラグは、リンカーの -z groupperm オプションを使用してオブジェクトに記録されます。109 ページの「グループの分離」を参照してください。

表 7–36　ELF 動的機能フラグ DT_FEATURE_1

名前	値	意味
DTF_1_PARINIT	0x1	部分的な初期化機能が必要です。
DTF_1_CONFEXP	0x2	構成ファイルが必要です。

DTF_1_PARINIT
: オブジェクトが部分的な初期化を必要とすることを示します。248 ページの「移動セクション」を参照してください。

DTF_1_CONFEXP
: このオブジェクトが、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。このフラグ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。このフラグの効果は、DF_1_CONFALT と同じです。

大域オフセットテーブル(プロセッサ固有)

一般に位置独立の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は、内部で使用するデータ内に絶対アドレスを保持します。このため、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、位置独立のアドレス指定を使用して GOT を参照し、絶対値を抽出します。この方法により、位置独立の参照を、絶対位置にリダイレクトできます。

最初は、GOT は再配置エントリで要求される情報を保持します。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あと、実行時リンカーが再配置エントリを処理します。これらの再配置のいくつかは、R_xxxx_GLOB_DAT タイプで GOT を参照する場合があります。

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 GOT 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 GOT 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、すべての GOT の再配置を処理してから、プロセスイメージ内のいずれかのコードに制御を渡します。この処理により、実行時に絶対アドレスが利用可能になります。

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。このシンボルを利用することにより、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。この方法は、実行時リンカーにとって特に重要です。なぜなら、実行時リンカーはほか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かぎり、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。

GOT の形式と解釈は、プロセッサに固有です。_GLOBAL_OFFSET_TABLE_ シンボルは、テーブルをアクセスするために使用できます。このシンボルは、.got セクションの中央に存在可能であるため、負の添字と負でない添字の両方をアドレスの配列に含めることができます。シンボルタイプは、32 ビットコードの場合、Elf 32_Addr の配列で、64 ビットコードの場合、Elf 64_Addr の配列です。

```
extern  Elf32_Addr  _GLOBAL_OFFSET_TABLE_[];
extern  Elf64_Addr  _GLOBAL_OFFSET_TABLE_[];
```

プロシージャーのリンクテーブル (プロセッサ固有)

大域オフセットテーブルは位置独立のアドレスの計算を絶対位置に変換します。同様に、プロシージャーのリンクテーブルは位置独立の関数呼び出しを絶対位置に変換します。リンカーは、異なる動的オブジェクト間の実行転送 (関数呼び出しなど) を解決できません。このため、リンカーはプログラム転送制御をプロシージャーのリンクテーブルのエントリに与えます。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リダイレクトします。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ーのリンクテーブルが存在します。

32 ビット SPARC: プロシージャーのリンクテーブル

32 ビット SPARC 動的オブジェクトの場合、プロシージャーのリンクテーブルは専用データ内に存在します。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ー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。

最初の 4 つのプロシージャーのリンクテーブルエントリは、予約されています。表 7–37 に例示されてはいますが、これらのエントリの元の内容は指定されていません。テーブル内の各エントリは 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。

再配置テーブルは、プロシージャー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、予約されていないプロシージャーのリンクテーブルエントリごとに 1 つのエントリが同じ順番で存在します。各エントリの再配置タイプ

は、R_SPARC_JMP_SLOT です。再配置オフセットは関連付けられているプロシージャー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ます。シンボルテーブルインデックスは適切なシンボルを参照します。

プロシージャーのリンクテーブル機能を説明するため、表 7–37 に 4 つのエントリが示されています。4 つのエントリのうちの 2 つは初期状態で予約されているエントリです。3 番目のエントリは name101 に対する呼び出しです。4 番目のエントリは name102 に対する呼び出しです。この例では、name102 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います。この最後のエントリの後には nop 命令が続きます。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ーリンクテーブルのエントリを変更するために使用できる命令シーケンスを示しています。

表 7–37　32 ビット SPARC: プロシージャー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:	.PLT0:
unimp	save %sp, -64, %sp
unimp	call runtime_linker
unimp	nop
.PLT1:	.PLT1:
unimp	.word identification
unimp	unimp
unimp	unimp
.PLT101:	.PLT101:
sethi (.-.PLT0), %g1	nop
ba,a .PLT0	ba,a name101
nop	nop
.PLT102:	.PLT102:
sethi (.-.PLT0), %g1	sethi (.-.PLT0), %g1
ba,a .PLT0	sethi %hi(name102), %g1
nop	jmpl %g1+%lo(name102), %g0
nop	nop

次の手順は、実行時リンカーとプログラムがプロシージャーのリンクテーブルによってシンボル参照をどのように協調して解決するかを示しています。ただし、次に記述されている手順は、単に説明のためのものです。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. プログラムのメモリーイメージが最初に作成されると、実行時リンカーはプロシージャーのリンクテーブルの初期エントリを変更します。これらのエントリは、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修正されます。実

行時リンカーはまた、識別情報 (identification) を 2 番目のエントリに格納します。実行時リンカーが制御を受け取ると、このワードは呼び出したオブジェクトを見つけるために調べられます。

2. ほかのすべてのプロシージャーのリンクテーブルエントリは、最初は先頭エントリに渡されます。これで、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制御を取得します。たとえば、プログラムが `name101` を呼び出すと、制御がラベル `.PLT101` に渡されます。
3. `sethi` 命令は、現在のプロシージャーのリンクテーブルエントリ (`.PLT101`) と最初のプロシージャーのリンクテーブルエントリ (`.PLT0`) の距離を計算します。この値は、`%g1` レジスタの最上位 22 ビットを占めます。
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します。
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) を取得します。
6. 実行時リンカーは、`%g1` 値をシフトしプロシージャーのリンクテーブルエントリのサイズで除算することで、 `name101` の再配置エントリのインデックスを計算します。再配置エントリ `101` のタイプは `R_SPARC_JMP_SLOT` です。再配置オフセットは `.PLT101` のアドレスを指定し、また、そのシンボルテーブルインデックスは `name101` を参照します。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、プロシージャー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します。

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必ずしも作成するとは限りません。ただし、作成する場合は、いくつかの点でより詳細な説明が必要です。

- コードを再入可能にするため、プロシージャーのリンクテーブルの命令に、特定の順番で変更が加えられます。実行時リンカーが関数のプロシージャー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りません。
- 実行時リンカーは、エントリを変換するために 3 つのワードを変更します。実行時リンカーは、命令を実行する際、ワード単位でのみ不可分に更新できます。このため、各ワードを逆順に更新して再入を可能にします。再入可能関数呼び出しが最後のパッチの直前に発生した場合、実行時リンカーは再度制御を取得します。実行時リンカーに対する両方の呼び出しで、同じプロシージャーのリンクテーブルエントリに変更が加えられるが、これらの変更は互いに干渉しません。
- プロシージャー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ます。`sethi` は `%g1` レジスタの値を変更するが、以前の内容を破棄しても問題はありません。
- 変換後、最後のプロシージャーのリンクテーブルエントリ (`.PLT102`) は、`jmp1` の遅延命令を必要とします。要求されている後続の `nop` は、この遅延スロットを埋めます。

注 – .PLT101 と .PLT102 の命令シーケンスの違いから、関連する宛先に合わせた最適化の方法を知ることができます。

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ヌル文字以外の場合、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 再配置エントリを処理します。

64 ビット SPARC: プロシージャーのリンクテーブル

64 ビット SPARC 動的オブジェクトの場合、プロシージャーのリンクテーブルは専用データ内に存在します。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ー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。

最初の 4 つのプロシージャーのリンクテーブルエントリは、予約されています。表 7–38 に例示されてはいますが、これらのエントリの元の内容は指定されていません。テーブル内の先頭 32,768 エントリは、それぞれ 8 ワード (32 バイト) を占め、32 バイト境界で整列する必要があります。テーブル全体は 256 バイト境界で整列する必要があります。32,768 を超えるエントリが必要な場合、残りのエントリは 6 ワード (24 バイト) および 1 つのポインタ (8 バイト) で構成されます。命令は、160 エントリのブロックにまとめられ、その次に 160 個ポインタが続きます。最後のグループのエントリとポインタは、160 未満でもかまいません。パディングの必要はありません。

注 – 32,768 および 160 という数字は、それぞれ分岐と読み込み置換の制限に基づいており、また、キャッシュの効率を向上させるために、コードとデータの間の区分を 256 バイト境界に合わせています。

再配置テーブルは、プロシージャー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、予約されていないプロシージャーのリンクテーブルエントリごとに 1 つのエントリが同じ順番で存在します。各エントリの再配置タイプは、R_SPARC_JMP_SLOT です。最初の 32,767 スロットでは、再配置オフセットは関連するプロシージャー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ます。加数フィールドはゼロになります。シンボルテーブルインデックスは適切なシンボルを参照します。32,768 以後のスロットでは、再配置オフセットは関連するポインタの先頭バイトのアドレスを指定します。加数フィールドは、再配置されていない値 -(.PLTN + 4) になります。シンボルテーブルインデックスは適切なシンボルを参照します。

プロシージャーのリンクテーブル機能を説明するため、表 7–38 に 4 つのエントリが示されています。最初の 3 つのエントリは、予約済みの初期エントリを示します。続く 3 つのエントリは、32,768 エントリの初期状態と、それぞれ、対象アドレス

がエントリの +/- 2G バイト以内の場合、アドレス空間の下位 4G バイト以内の場合、およびその他の場合に適用されると考えられる、変換された状態を示しています。最後の 2 つのエントリは、命令とポインタのペアで構成される、後のエントリの例を示します。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ーリンクテーブルのエントリを変更するために使用できる命令シーケンスを示しています。

表 7–38 64 ビット SPARC: プロシージャー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:	.PLT0:
unimp	save %sp, -176, %sp
unimp	sethi %hh(runtime_linker_0), %l0
unimp	sethi %lm(runtime_linker_0), %l1
unimp	or %l0, %hm(runtime_linker_0), %l0
unimp	sllx %l0, 32, %l0
unimp	or %l0, %l1, %l0
unimp	jmpl %l0+%lo(runtime_linker_0), %o1
unimp	mov %g1, %o0
.PLT1:	.PLT1:
unimp	save %sp, -176, %sp
unimp	sethi %hh(runtime_linker_1), %l0
unimp	sethi %lm(runtime_linker_1), %l1
unimp	or %l0, %hm(runtime_linker_1), %l0
unimp	sllx %l0, 32, %l0
unimp	or %l0, %l1, %l0
unimp	jmpl %l0+%lo(runtime_linker_0), %o1
unimp	mov %g1, %o0
.PLT2:	.PLT2:
unimp	.xword identification

表 7–38 64 ビット SPARC: プロシージャーのリンクテーブルの例 (続き)

オブジェクトファイル	メモリーセグメント

.PLT101:
sethi (.-.PLT0), %g1
ba,a %xcc, .PLT1
nop
nop
nop; nop
nop; nop
	.PLT101:
	nop
	mov %o7, %g1
	call name101
	mov %g1, %o7
	nop; nop
	nop; nop

.PLT102: sethi (.-.PLT0), %g1 ba,a %xcc, .PLT1 nop nop nop; nop nop; nop	.PLT102: nop sethi %hi(name102), %g1 jmpl %g1+%lo(name102), %g0 nop nop; nop nop; nop
.PLT103: sethi (.-.PLT0), %g1 ba,a %xcc, .PLT1 nop nop nop nop nop nop	.PLT103: nop sethi %hh(name103), %g1 sethi %lm(name103), %g5 or %hm(name103), %g1 sllx %g1, 32, %g1 or %g1, %g5, %g5 jmpl %g5+%lo(name103), %g0 nop

.PLT32768:
mov %o7, %g5
call .+8
nop
ldx [%o7+.PLTP32768 -
(.PLT32768+4)], %g1
jmpl %o7+%g1, %g1
mov %g5, %o7
 	
...
	.PLT32768:
	<unchanged>
	<unchanged>
	<unchanged>
	<unchanged>
	
	<unchanged>
	<unchanged>
	
	...

.PLT32927:
mov %o7, %g5
call .+8
nop
ldx [%o7+.PLTP32927 -
(.PLT32927+4)], %g1
jmpl %o7+%g1, %g1
mov %g5, %o7
	.PLT32927:
	<unchanged>
	<unchanged>
	<unchanged>
	<unchanged>
	
	<unchanged>
	<unchanged>

表 7–38 64 ビット SPARC: プロシージャーのリンクテーブルの例 (続き)

オブジェクトファイル	メモリーセグメント
.PLTP32768 .xword .PLT0 - (.PLT32768+4) ... .PLTP32927 .xword .PLT0 - (.PLT32927+4)	.PLTP32768 .xword name32768 - (.PLT32768+4) ... .PLTP32927 .xword name32927 - (.PLT32927+4)

次の手順は、実行時リンカーとプログラムがプロシージャーのリンクテーブルによってシンボル参照をどのように協調して解決するかを示しています。ただし、次に記述されている手順は、単に説明のためのものです。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. プログラムのメモリーイメージが最初に作成されると、実行時リンカーはプロシージャーのリンクテーブルの初期エントリを変更します。エントリは、実行時リンカー自身のルーチンに制御を渡すように修正されます。実行時リンカーはまた、識別情報 (identification) の拡張ワードを 3 番目のエントリに格納します。実行時リンカーが制御を受け取ると、このワードは呼び出したオブジェクトを見つけるために調べられます。
2. ほかのすべてのプロシージャーのリンクテーブルエントリは、最初、先頭または 2 番目のエントリに渡されます。これらのエントリは、スタックフレームを確立して、実行時リンカーを呼び出します。
3. 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) を取得します。
4. 実行時リンカーは、テーブルスロットの再配置エントリのインデックスを計算します。
5. インデックス情報に関しては、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、プロシージャーのリンクテーブルエントリを変更してから、制御を宛先に渡します。

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必ずしも作成するとは限りません。ただし、作成する場合は、いくつかの点でより詳細な説明が必要です。

- コードを再入可能にするため、プロシージャーのリンクテーブルの命令に、特定の順番で変更が加えられます。実行時リンカーが関数のプロシージャー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りません。

- 実行時リンカーは、8 ワードまで変更を加えてエントリを変換できます。実行時リンカーは、命令を実行する際、ワード単位でのみ不可分に更新できます。このため、64 ビットストアを使用している場合、再入可能性は、まず nop 命令を置換命令で上書きし、次に ba、a および sethi をパッチ適用することで実現されます。再入可能関数呼び出しが最後のパッチの直前に発生した場合、実行時リンカーは再度制御を取得します。実行時リンカーに対する両方の呼び出しで、同じプロシージャーのリンクテーブルエントリに変更が加えられるが、これらの変更は互いに干渉しません。
- 最初の sethi 命令が変更されると、この命令を変更するには nop を使用する必要があります。

エントリの 2 番目のフォームに示すように、ポインタの変更は、単一の不可分 64 ビットストアを使用して行われます。

注 – .PLT101、.PLT102、および .PLT103 の命令シーケンスの違いから、関連する宛先に合わせた最適化の方法を知ることができます。

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ヌル文字以外の場合、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 再配置エントリを処理します。

32 ビット x86: プロシージャーのリンクテーブル

32 ビット x86 動的オブジェクトの場合、プロシージャーリンクテーブルは共有テキスト内に存在しますが、非公開の大域オフセットテーブル内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リダイレクトします。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ーのリンクテーブルが存在します。

表 7–39　32 ビット x86: 絶対プロシージャー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
```

表 7–40　32 ビット x86: 位置独立のプロシージャーリンクテーブルの例

```
.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
```

注 – 前述の例が示すとおり、プロシージャーリンクテーブルの命令は、絶対コードと位置独立のコードで異なるオペランドアドレス指定モードを使用します。それでも、実行時リンカーへのインタフェースは同一です。

次の手順は、実行時リンカーとプログラムがプロシージャーのリンクテーブルおよび大域オフセットテーブルによってシンボル参照をどのように協同で解決するかを示しています。

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な値を設定します。これらの値については、次の手順で説明します。

2. プロシージャーのリンクテーブルが位置独立の場合、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りません。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ー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ーのリンクテーブルエントリに渡されます。したがって、呼び出し側関数は、プロシージャー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りません。
3. たとえば、プログラムが name1 を呼び出すと、制御が .PLT1 に渡されます。
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します。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します (name1 の実アドレスは保持しない)。
5. 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します。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です。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します。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 を取得します。
6. プログラムは、再配置オフセットをプッシュした後、.PLT0 (プロシージャーのリンクテーブルの先頭エントリ) にジャンプします。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ます。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します。
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します。
8. その後のプロシージャーのリンクテーブルエントリに対する実行は、name1 に直接渡されます (実行時リンカーの再呼び出しは行われない)。.PLT1 における jmp 命令は、pushl 命令にジャンプする代わりに、name1 にジャンプします。

LD_BIND_NOW 環境変数は、動的リンク処理の動作を変更します。この環境変数の値がヌル文字以外の場合、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 再配置エントリを処理します。

x64: プロシージャーのリンクテーブル

x64 動的オブジェクトの場合、プロシージャーリンクテーブルは共有テキスト内に存在しますが、非公開の大域オフセットテーブル内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リダイレクトします。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ーのリンクテーブルが存在します。

表 7–41 x64: プロシージャーのリンクテーブルの例

```
.PLT0:
    pushq   GOT+8(%rip)                         # GOT[1]
    jmp     *GOT+16(%rip)                       # GOT[2]
    nop;    nop
    nop;    nop
.PLT1:
    jmp     *name1@GOTPCREL(%rip)               # 16 bytes from .PLT0
    pushq   $index1
    jmp     .PLT0
.PLT2:
    jmp     *name2@GOTPCREL(%rip)               # 16 bytes from .PLT1
    pushl   $index2
    jmp     .PLT0
```

次の手順は、実行時リンカーとプログラムがプロシージャーのリンクテーブルおよび大域オフセットテーブルによってシンボル参照をどのように協同で解決するかを示しています。

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な値を設定します。これらの値については、次の手順で説明します。
2. 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ー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ーのリンクテーブルエントリに渡されます。
3. たとえば、プログラムが `name1` を呼び出すと、制御が `.PLT1` に渡されます。
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します。大域オフセットテーブルは最初は、後続の `pushq` 命令のアドレスを保持します (`name1` の実アドレスは保持しない)。
5. プログラムは再配置インデックス (`index1`) をスタックにプッシュします。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インデックスです。再配置テーブルは `DT_JUMPREL` 動的セクションエントリによって識別されます。指定された再配置エントリには `R_AMD64_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します。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` を取得します。
6. プログラムは、再配置インデックスをプッシュした後、`.PLT0` (プロシージャーのリンクテーブルの先頭エントリ) にジャンプします。`pushq` 命令は、2 番目の大域オフセットテーブルエントリ (`GOT+8`) 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ます。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+16`) のアドレスにジャンプして、実行時リンカーにジャンプします。

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します。
8. その後のプロシージャーのリンクテーブルエントリに対する実行は、name1 に直接渡されます (実行時リンカーの再呼び出しは行われない)。.PLT1 における jmp 命令は、pushq 命令にジャンプする代わりに、name1 にジャンプします。

LD_BIND_NOW 環境変数は、動的リンク処理の動作を変更します。この環境変数の値がヌル文字以外の場合、実行時リンカーは、プログラムに制御を渡す前に R_AMD64_JMP_SLOT 再配置エントリを処理します。

◆ ◆ ◆ 第 8 章

8 スレッド固有領域 (TLS)

コンパイル環境は、スレッド固有データの宣言をサポートします。このデータは、スレッド特有データやスレッド専用データと呼ばれることもありますが、一般には頭文字で TLS と呼ばれます。変数をスレッド固有として宣言すると、コンパイラは自動的にこれらの変数をスレッド単位で割り当てます。

この機能の組み込みサポートには、次に示す 3 つの目的があります。

- スレッド固有のデータを割り当てる POSIX インタフェースの構築基盤を提供する。
- アプリケーションとライブラリでスレッド固有変数を直接使用するための、便利で効果的なメカニズムを提供する。
- コンパイラが、ループ並列化による最適化をする場合に、TLS を必要なだけ割り当てることができる。

C/C++ プログラミングインタフェース

次の例に示すように、`__thread` キーワードを使用すると、変数をスレッド固有として宣言できます。

```
__thread int i;
__thread char *p;
__thread struct state s;
```

ループの最適化の際に、コンパイラは必要に応じてスレッド固有一時領域を作成することがあります。

適用性

`__thread` キーワードは任意の大域変数、ファイルスコープの静的変数、または関数スコープの静的変数に適用できます。常にスレッド固有である自動変数には影響を与えません。

初期化
C++ では、初期化に静的なコンストラクタが必要となる場合には、スレッド固有変数の初期化が行われないことがあります。静的なコンストラクタを必要としないかぎり、スレッド固有変数は通常の静的変数に有効な任意の値に初期化できます。

変数は、(スレッド固有であるかどうかにかかわらず) スレッド固有変数のアドレスに静的に初期化することはできません。

結合
スレッド固有変数の宣言と参照は外部的に行えます。スレッド固有変数は、通常のシンボルと同じ割り込み規則に従う必要があります。

動的な読み込みの制限
さまざまな TLS アクセスモデルを利用できます。331 ページの「スレッド固有領域のアクセスモデル」を参照してください。共有オブジェクトを開発するときは、オブジェクトの読み込みに関連して一部のアクセスモデルに適用される制限に注意するようにしてください。共有オブジェクトは、プロセスの起動時に動的に読み込むことができ、またプロセスの起動後には、遅延読み込み、フィルタ、または `dlopen(3C)` によって、動的に読み込むことができます。プロセスの起動が完了すると、メインスレッドのスレッドポインタが確立されます。すべての静的な TLS 領域要件は、スレッドポインタが確立される前に計算されます。

スレッド固有変数を参照する共有オブジェクトでは、その参照を含むすべての変換ユニットは、動的な TLS モデルを使ってコンパイルするようにしてください。このアクセスモデルを使用すると、共有オブジェクトをもっとも柔軟に読み込むことができます。ただし、静的な TLS モデルを使用すると、コードの速度が向上します。静的な TLS モデルを使用する共有オブジェクトは、プロセスを初期化するときに読み込むことができます。ただし、プロセスを初期化したあとは、静的な TLS モデルを使用する共有オブジェクトの読み込みは、十分なバックアップ TLS 記憶域が使用可能な場合にのみ実行できます。328 ページの「プログラムの起動」を参照してください。

アドレス演算子
スレッド固有変数には、アドレス演算子 `&` を使用できます。この演算子は、実行時に評価されて、現在のスレッド内の変数のアドレスを返します。この演算子によって取得されたアドレスは、アドレスを評価したスレッドが存在するかぎり、プロセス内のあらゆるスレッドで自由に使用できます。スレッドが終了した時点で、そのスレッド内のスレッド固有変数を指すポインタはすべて無効になります。

スレッド固有変数のアドレスを取得するために `dlsym(3C)` を使用すると、`dlsym()` を呼び出したスレッド内におけるその変数のインスタンスのアドレスが返されます。

スレッド固有領域 (TLS) セクション

コンパイル時に割り当てられたスレッド固有データのコピーは、実行される個々のスレッドに、個別に関連付けられる必要があります。このデータを提供するために、TLS セクションを使用してサイズと初期の内容を指定します。コンパイル環境は、SHF_TLS フラグで識別されるセクション内に TLS を割り当てます。これらのセクションは、領域がどのように宣言されているかにもとづき、初期化された TLS と初期化されていない TLS を提供します。

- 初期化されたスレッド固有変数は、.tdata セクション内または .tdata1 セクション内に割り当てます。この初期化は再配置を必要とする場合があります。
- 初期化されていないスレッド固有変数は、COMMON シンボルとして定義します。その結果の割り当ては、.tbss セクション内で行われます。

初期化されていないセクションは、適切な整列になるようにパッドを入れられて、初期化されたセクションの直後に割り当てられます。結合されたこれらのセクションは、新しいスレッドの作成時に TLS の割り当てに使用できる TLS テンプレートとなります。このテンプレートの初期化された部分を、TLS 初期化イメージと呼びます。初期化されたスレッド固有変数の結果として発生する再配置はすべて、このテンプレートに適用されます。その後、新しいスレッドが初期値を要求すると、再配置された値が使用されます。

TLS シンボルのシンボルタイプは STT_TLS です。これらのシンボルには、TLS テンプレートの先頭からの相対オフセットが割り当てられます。これらのシンボルに関連付けられた実際の仮想アドレスとは無関係です。このアドレスが指すのはテンプレートだけで、各データ項目のスレッドごとのコピーではありません。動的実行可能ファイルと共有オブジェクトでは、STT_TLS シンボルの st_value フィールドに、定義済みシンボルの場合は割り当てられた TLS オフセットが含まれます。未定義シンボルの場合は、このフィールドにゼロが含まれます。

TLS へのアクセスをサポートするために、再配置がいくつか定義されます。339 ページの「SPARC: スレッド固有領域の再配置のタイプ」、346 ページの「32 ビット x86: スレッド固有領域の再配置のタイプ」、および352 ページの「x64: スレッド固有領域の再配置のタイプ」を参照してください。TLS 再配置は通常、タイプ STT_TLS のシンボルを参照します。TLS 再配置は、GOT エントリに関連するローカルセクションシンボルも参照できます。この場合、割り当てられた TLS のオフセットが、関連する GOT エントリに保存されます。

動的実行可能ファイルと共有オブジェクトでは、PT_TLS プログラムエントリが TLS テンプレートを記述します。このテンプレートには、次のメンバーが含まれます。

表 8–1　ELF PT_TLS プログラムヘッダーエントリ

メンバー	値
p_offset	TLS 初期化イメージのファイルオフセット
p_vaddr	TLS 初期化イメージの仮想メモリーアドレス
p_paddr	0
p_filesz	TLS 初期化イメージのサイズ
p_memsz	TLS テンプレートの合計サイズ
p_flags	PF_R
p_align	TLS テンプレートの整列

スレッド固有領域の実行時の割り当て

TLS は、プログラムの存続中に、次の 3 つの機会に作成されます。

- プログラムの起動時。
- 新しいスレッドの作成時。
- プログラムの起動に続いて共有オブジェクトが読み込まれたあと、スレッドがはじめて TLS ブロックを参照するとき。

スレッド固有データ領域は、図 8–1 に示すように実行時に配置されます。

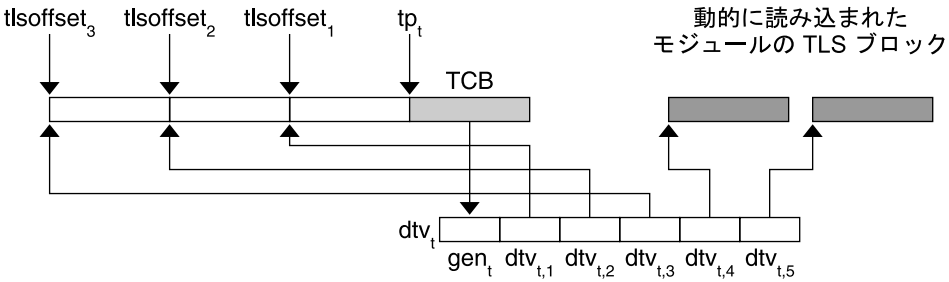


図 8–1　スレッド固有領域の実行時のレイアウト

プログラムの起動

プログラムの起動時に、実行システムはメインスレッド用の TLS を作成します。

まず実行時リンカーが、読み込まれたすべての動的オブジェクト (動的な実行可能ファイルを含む) の TLS テンプレートを論理的に結合し、単一の静的なテンプレート

としてまとめます。各動的オブジェクトの TLS テンプレートには、結合されたテンプレート内のオフセット $tlsoffset_m$ が次のように割り当てられます。

- $tlsoffset_1 = round(tlssize_1, align_1)$
- $tlsoffset_{m+1} = round(tlsoffset_m + tlssize_{m+1}, align_{m+1})$

$tlssize_{m+1}$ は動的オブジェクト m の割り当てテンプレートのサイズで、$align_{m+1}$ は整列です。ここで、1 <= m <= M であり、M は読み込まれる動的オブジェクトの合計数です。round(offset, align) 関数は、*align* の次の倍数に丸められたオフセットを返します。

次に、実行時リンカーは、起動時の TLS に必要な割り当てサイズの $tlssize_S$ を計算します。このサイズは、$tlsoffset_M$ に 512 バイトを加えた値に等しくなります。この加算により、静的な TLS 参照のバックアップ予約が得られます。静的な TLS 参照を作成し、プロセスの初期化後に読み込まれる共有オブジェクトは、このバックアップ予約に割り当てられます。ただし、この予約のサイズは固定および限定されています。また、この予約が対応しているのは、初期化されていない TLS データ項目用の記憶域の提供だけです。柔軟性を最大限高めるため、動的な TLS モデルを使用して、共有オブジェクトがスレッドローカル変数を参照するようにしてください。

計算された TLS サイズ $tlssize_S$ に関連付けられる静的な TLS 領域は、スレッドポインタ tp_t の直前に配置されます。TLS データに対するアクセスは、tp_t からの減算にもとづいて行われます。

静的な TLS 領域は、初期化レコードのリンクリストに関連付けられます。このリスト内の各レコードは、読み込まれた動的オブジェクトごとにその TLS 初期化イメージを記述するものです。各レコードには、次のフィールドが含まれています。

- TLS 初期化イメージを指すポインタ。
- TLS 初期化イメージのサイズ。
- オブジェクトの $tlsoffset_m$。
- オブジェクトが静的な TLS モデルを使用するかどうかを示すフラグ。

スレッドライブラリは、この情報を使用して初期スレッドに領域を割り当てます。この領域が初期化され、初期スレッド用に動的な TLS ベクトルが作成されます。

スレッドの作成

初期スレッドと、新しく作成されるスレッドに対して、スレッドライブラリは読み込まれる動的オブジェクトごとに新しい TLS ブロックを割り当てます。ブロックは、個別に割り当てられることも、単一の連続ブロックとして割り当てられることもあります。

各スレッド t は関連するスレッドポインタ tp_t を持ち、このポインタはスレッド制御ブロック TCB を指します。スレッドポインタ tp には、常に、現在動作しているスレッドの tp_t の値が含まれます。

続いてスレッドライブラリは、現在のスレッド t のためにポインタのベクトル $_{dtv}t$ を作成します。各ベクトルの最初の要素には、ベクトルを拡張すべきタイミングを決定するために使用される生成番号 gen_t が入ります。331 ページの「スレッド固有領域ブロックの遅延割り当て」を参照してください。

ベクトル内の残りの各要素 $dtv_{t,m}$ は、動的オブジェクト m に属する TLS 用に予約されたブロックへのポインタです。

起動後動的に読み込まれたオブジェクトについては、スレッドライブラリは TLS ブロックの割り当てを延期します。割り当ては、読み込まれたオブジェクト内で TLS 変数に対して最初の参照が行われる時に発生します。割り当てが延期されたブロックの場合、ポインタ $dtv_{t,m}$ は実装が定める特別な値に設定されます。

注 – 実行時リンカーは、ベクトル内の単一の要素 dtvt,1 を共有するために、すべての起動オブジェクトの $_{TLS}$ テンプレートをグループ化できます。このグループ化によって、前述のオフセット計算や初期化レコードのリストの作成が影響を受けることはありません。しかし、次の節では、M の値 (オブジェクトの合計数) は値 1 から始まっています。

続いて、スレッドライブラリが、新しい領域ブロック内の対応する場所に初期化イメージをコピーします。

起動後の動的読み込み

動的な TLS だけが含まれる共有オブジェクトは、プロセスの起動に続いて無制限に読み込むことができます。実行時リンカーは、初期化レコードのリストを拡張して新しいオブジェクトの初期化テンプレートを含めます。新しいオブジェクトには、インデックス m = M + 1 が与えられます。カウンタ M は 1 ずつ増えていきます。しかし、新しい TLS ブロックの割り当ては、それらが実際に参照されるまで延期されます。

動的な TLS だけが含まれる共有オブジェクトの読み込みが解除されると、その共有オブジェクトが使用している TLS ブロックは解放されます。

静的な TLS が含まれる共有オブジェクトは、プロセスの起動に続いて一定の制限内で読み込むことができます。静的な TLS 参照を満たすことができるのは、残りのバックアップ TLS 予約からだけです。328 ページの「プログラムの起動」を参照してください。この予約のサイズは制限されています。また、この予約で提供できるのは、初期化されていない TLS データ項目用の記憶域だけです。

静的な TLS を含む共有オブジェクトは、読み込み解除されません。静的な TLS 処理の結果として、共有オブジェクトには削除不可のタグが付けられます。

スレッド固有領域ブロックの遅延割り当て

動的な TLS モデルでは、スレッド t がオブジェクト m の TLS ブロックにアクセスする必要が生じた場合、コードは dtv_t を更新し、TLS ブロックの初期割り当てを行います。スレッドライブラリは、動的な TLS 割り当てが行えるように、次のインタフェースを提供します。

```
typedef struct {
    unsigned long ti_moduleid;
    unsigned long ti_tlsoffset;
} TLS_index;

extern void * __tls_get_addr(TLS_index * ti);     (SPARC and x64)
extern void * ___tls_get_addr(TLS_index * ti);    (32–bit x86)
```

注 – この関数の SPARC 定義と 64 ビットの x86 定義は、同一の関数シグニチャーを持ちます。しかし 32 ビットの x86 バージョンは、スタック上で引数を渡すデフォルトの呼び出し規約を使用しません。代わりに 32 ビットの x86 バージョンは、より効率の良い %eax レジスタによって引数を渡します。この代替呼び出し手法を使用することを示すため、32 ビットの x86 関数名にはその先頭に 3 つの下線が付いています。

`tls_get_addr()` の両バージョンとも、スレッドごとの生成カウンタ gen_t を調べ、ベクトルが更新を必要としていないかを確認します。ベクトル dtv_t が古い場合、ルーチンがベクトルを更新し、必要に応じ、追加エントリ用のスペースを確保するため再割り当てを行います。続いてこのルーチンは、$dtv_{t,m}$ に対応する TLS ブロックがすでに割り当てられているかを調べます。ベクトルが割り当てられていない場合、このルーチンはブロックの割 り当てと初期化を行います。このルーチンは、実行時リンカーが提供する初期化レコードリスト内の情報を使用します。ポインタ $dtv_{t,m}$ は、割り当てられたブロックを指すように設定されます。ルーチンは、ブロック内の指定されたオフセットへのポインタを返します。

スレッド固有領域のアクセスモデル

各 TLS 参照は、次のアクセスモデルのどれかになります。ここでは、もっとも一般的なモデル (しかし最適化の程度はもっとも低い) から順に、もっとも高速なモデル (しかし制限度は高い) へと並んでいます。

General Dynamic (GD) - 動的な TLS
このモデルでは、共有オブジェクトまたは動的実行可能ファイルから、すべての TLS 変数を参照できます。このモデルでは、TLS ブロックが特定のスレッドからはじめて参照される時まで、このブロックの割り当てを延期することもできます。

Local Dynamic (LD) - 局所シンボルの動的な TLS
このモデルは、*GD* モデルを最適化したものです。コンパイラが、構築されるオブジェクト内で変数がローカルに結合されているか、あるいは保護されていると判断することがあります。この場合、コンパイラは、動的な `tlsoffset` を静的に結合してこのモデルを使用するように、リンカーに指示します。このモデルにより、*GD* モデルを上回る性能が得られます。`dtv()0,m` のアドレスの確認は、関数ごとに $_{\text{tls\_get\_addr}}$ を 1 度呼び出すだけです。リンク編集時に結合される動的な TLS オフセットは、参照ごとに $\texttt{dtv}_{0,m}$ アドレスに追加されます。

Initial Executable (IE) - オフセットが割り当てられた静的な TLS
このモデルは、初期の静的な TLS テンプレートの一部として利用できる TLS 変数だけを参照できます。このテンプレートは、プロセスの起動時に使用できるすべての TLS ブロック、および小規模なバックアップ予約で構成されます。328 ページの「プログラムの起動」を参照してください。このモデルでは、変数 *x* の、スレッドポインタからの相対オフセットは、*x* の `GOT` エントリ内に保存されます。

このモデルでは、初期プロセスの起動後に、遅延読み込み、フィルタ、`dlopen(3C)` などによって読み込まれる共有ライブラリから、限定された数の TLS 変数を参照できます。このアクセスは、固定のバックアップ予約から満たされます。この予約で提供できるのは、初期化されていない TLS データ項目用の記憶域だけです。柔軟性を最大限高めるため、動的な TLS モデルを使用して、共有オブジェクトがスレッドローカル変数を参照するようにしてください。

注 – フィルタを使用して、静的な TLS の使用を動的に選択できます。共有オブジェクトを、動的な TLS を使用するように構築します。さらに共有オブジェクトを、静的な TLS を使用するために構築された対応物に対する補助フィルタとして動作するよう、構築することができます。静的な TLS オブジェクトの読み込みがリソースにより許可される場合に、そのオブジェクトが使用されます。それ以外の場合、動的な TLS オブジェクトへのフォールバックにより、共有オブジェクトの提供する機能が常に使用可能であることが保証されます。フィルタの詳細については、127 ページの「フィルタとしての共有オブジェクト」を参照してください。

Local Executable (LE) - 静的な TLS
このモデルは、動的実行可能ファイルの TLS ブロックの一部である TLS 変数だけを参照できます。リンカーは、動的な再配置や `GOT` の参照を別途行うことなく、スレッドポインタからの相対オフセットを静的に計算します。このモデルを

使用して動的実行可能ファイルの外部に存在する変数を参照することはできません。

リンカーは、妥当と判断する場合は、比較的一般的なアクセスモデルから、より最適化されたモデルへとコードを移行できます。この移行は、独特な TLS 再配置を使用することで行えます。これらの再配置は、更新を要求するだけでなく、どの TLS アクセスモデルが使用されているかの特定もします。

作成されるオブジェクトのタイプとともに TLS アクセスモデルを認識することで、リンカーは変換を実行できます。たとえば、*GD* アクセスモデルを使用している再配置可能オブジェクトが、動的実行可能ファイルにリンクされているとします。この場合、リンカーは *IE* アクセスモデルまたは *LE* アクセスモデルを使用して参照を適宜移行できます。そのあとで、そのモデルに必要な再配置が行われます。

次の図は、それぞれのアクセスモデルと、あるモデルから別のモデルにどのように移行するかを示しています。

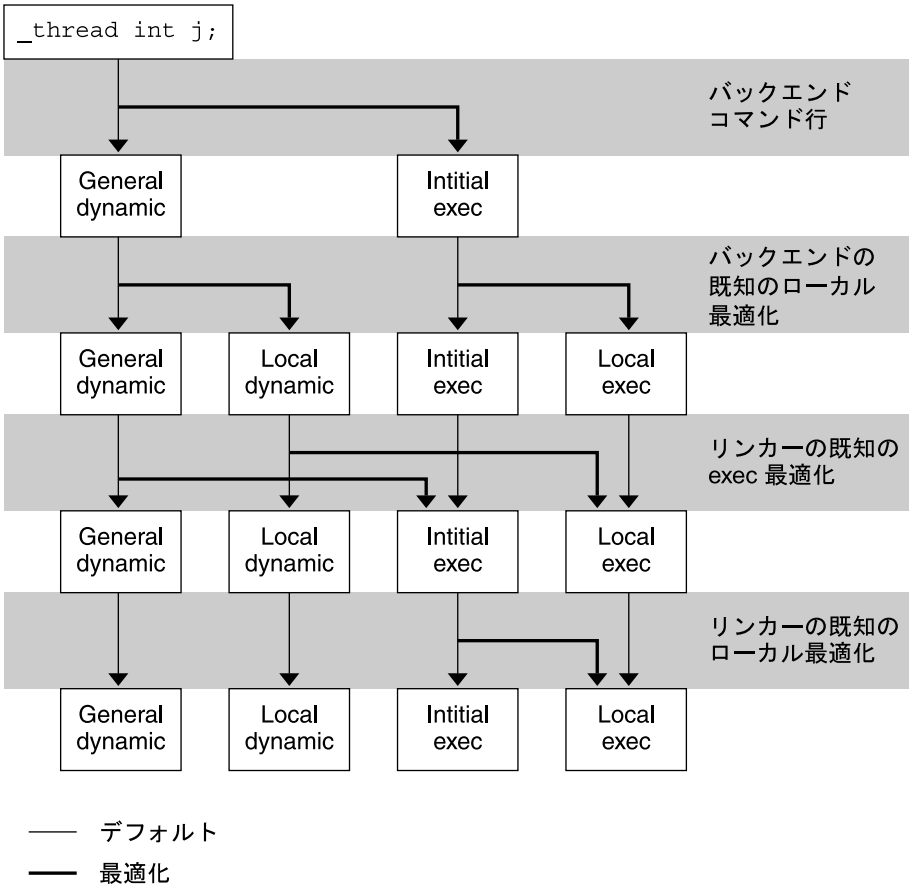


図 8–2 スレッド固有領域のアクセスモデルと移行

SPARC: スレッド固有変数へのアクセス

SPARC では、スレッド固有変数へのアクセスに次のコードシーケンスモデルを使用できます。

SPARC: General Dynamic (GD)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている GD モデルを実装します。

表 8–2 SPARC: General Dynamic スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
`# %l7 - initialized to GOT pointer`		
		x
`0x00 sethi %hi(@dtlndx(x)), %o0`	R_SPARC_TLS_GD_HI22	x
`0x04 add %o0, %lo(@dtlndx(x)), %o0`	R_SPARC_TLS_GD_LO10	x
`0x08 add %l7, %o0, %o0`	R_SPARC_TLS_GD_ADD	x
`0x0c call x@TLSPLT`	R_SPARC_TLS_GD_CALL	
`# %o0 - contains address of TLS variable`		
	未処理の再配置: 32 ビット	シンボル
GOT[n]	R_SPARC_TLS_DTPMOD32	x
GOT[n + 1]	R_SPARC_TLS_DTPOFF32	x
	未処理の再配置: 64 ビット	シンボル
GOT[n]	R_SPARC_TLS_DTPMOD64	x
GOT[n + 1]	R_SPARC_TLS_DTPOFF64	x

sethi 命令は R_SPARC_TLS_GD_HI22 再配置を生成し、add 命令は R_SPARC_TLS_GD_LO10 再配置を生成します。これらの再配置は、変数 x の TLS_index 構造体を保持する領域を GOT 内に割り当てるように、リンカーに指示します。リンカーは、この新しい GOT エントリに GOT からの相対オフセットを代入することによって、この再配置を処理します。

読み込みオブジェクトインデックスと x の TLS ブロックインデックスは実行時まで不明です。したがって、リンカーは、実行時リンカーによって処理されるように、GOT に対する R_SPARC_TLS_DTPMOD32 再配置と R_SPARC_TLS_DPTOFF32 再配置を設定します。

2 番目の add 命令は、R_SPARC_TLS_GD_ADD 再配置を生成します。この再配置が使用されるのは、リンカーによって GD コードシーケンスがほかのシーケンスに変更される場合だけです。

call 命令は特別な構文である x@TLSPLT を使用します。この call 命令は TLS 変数を参照し、R_SPARC_TLS_GD_CALL 再配置を生成します。この再配置は、__tls_get_addr() 関数の呼び出しを結合するようリンカーに指示し、call 命令を GD コードシーケンスに関連付けます。

注 – add 命令は、call 命令の前に指定する必要があります。add 命令を、呼び出しの遅延スロットに配置することはできません。これは、あとで発生するコード変換が既知の順序を必要とするためです。

R_SPARC_TLS_GD_ADD 再配置によってタグが付けられた add 命令の GOT ポインタとして使用されるレジスタは、add 命令内の最初のレジスタでなければなりません。このように指定することで、リンカーはコード変換時に GOT ポインタであるレジスタを識別できるようになります。

SPARC: Local Dynamic (LD)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている LD モデルを実装します。

表 8–3 SPARC: Local Dynamic スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
`# %l7 - initialized to GOT pointer`		
`0x00 sethi %hi(@tmndx(x1)), %o0`	R_SPARC_TLS_LDM_HI22	x1
`0x04 add %o0, %lo(@tmndx(x1)), %o0`	R_SPARC_TLS_LDM_LO10	x1
`0x08 add %l7, %o0, %o0`	R_SPARC_TLS_LDM_ADD	x1
`0x0c call x@TLSPLT`	R_SPARC_TLS_LDM_CALL	x1
`# %o0 - contains address of TLS block of current object`		
`0x10 sethi %hi(@dtpoff(x1)), %l1`	R_SPARC_TLS_LDO_HIX22	x1
`0x14 xor %l1, %lo(@dtpoff(x1)), %l1`	R_SPARC_TLS_LDO_LOX10	x1
`0x18 add %o0, %l1, %l1`	R_SPARC_TLS_LDO_ADD	x1
`# %l1 - contains address of local TLS variable x1`		
`0x20 sethi %hi(@dtpoff(x2)), %l2`	R_SPARC_TLS_LDO_HIX22	x2
`0x24 xor %l2, %lo(@dtpoff(x2)), %l2`	R_SPARC_TLS_LDO_LOX10	x2
`0x28 add %o0, %l2, %l2`	R_SPARC_TLS_LDO_ADD	x2
`# %l2 - contains address of local TLS variable x2`		
	未処理の再配置: 32 ビット	シンボル
`GOT[n]`	R_SPARC_TLS_DTPMOD32	x1
`GOT[n + 1]`	<none>	

表 8–3 SPARC: Local Dynamic スレッド固有変数のアクセスコード (続き)

	未処理の再配置: 64 ビット	シンボル
GOT[n]	R_SPARC_TLS_DTPMOD64	x1
GOT[n + 1]	<none>	

最初の sethi 命令は R_SPARC_TLS_LDM_HI22 再配置を生成し、add 命令は R_SPARC_TLS_LDM_LO10 再配置を生成します。これらの再配置は、現在のオブジェクトの TLS_index 構造体を保持する領域を GOT に割り当てるように、リンカーに指示します。リンカーは、この新しい GOT エントリに GOT からの相対オフセットを代入することによって、この再配置を処理します。

読み込みオブジェクトインデックスは実行時まで不明です。したがって、R_SPARC_TLS_DTPMOD32 再配置が作成され、TLS_index 構造体の ti_tlsoffset フィールドにゼロが埋め込まれます。

2 つめの add 命令には R_SPARC_TLS_LDM_ADD 再配置によってタグが付けられ、call 命令には R_SPARC_TLS_LDM_CALL 再配置によってタグが付けられます。

以降の sethi 命令は R_SPARC_LDO_HIX22 再配置を生成し、xor 命令は R_SPARC_TLS_LDO_LOX10 再配置を生成します。各局所シンボルの TLS オフセットはリンク編集時に認識されるため、これらの値は直接埋め込まれます。add 命令には、R_SPARC_TLS_LDO_ADD 再配置によってタグが付けられます。

手続きが複数の局所シンボルを参照する場合には、コンパイラは TLS ブロックの基底アドレスを取得するコードを 1 度だけ生成します。以後、各シンボルのアドレスの計算にはこの基底アドレスが使用され、個別にライブラリを呼び出すことはありません。

注 – R_SPARC_TLS_LDO_ADD によってタグが付けられた add 命令内の TLS オブジェクトアドレスが入ったレジスタは、命令シーケンス内の最初のレジスタでなければなりません。このように指定することで、リンカーはコード変換時にレジスタを識別できるようになります。

---

32 ビット SPARC: Initial Executable (IE)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている IE モデルを実装します。

表 8–4 32 ビット SPARC: Initial Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル

表 8–4 32 ビット SPARC: Initial Executable スレッド固有変数のアクセスコード (続き)

# %l7 - initialized to GOT pointer, %g7 - thread pointer		
0x00 sethi %hi(@tpoff(x)), %o0	R_SPARC_TLS_IE_HI22	x
0x04 or %o0, %lo(@tpoff(x)), %o0	R_SPARC_TLS_IE_LO10	x
0x08 ld [%l7 + %o0], %o0	R_SPARC_TLS_IE_LD	x
0x0c add %g7, %o0, %o0	R_SPARC_TLS_IE_ADD	x
# %o0 - contains address of TLS variable		
	未処理の再配置	シンボル
GOT[n]	R_SPARC_TLS_TPOFF32	x

sethi 命令は R_SPARC_TLS_IE_HI22 再配置を生成し、or 命令は R_SPARC_TLS_IE_LO10 再配置を生成します。これらの再配置は、シンボル x の静的な TLS オフセットを保存する領域を GOT 内に作成するように、リンカーに指示します。実行時リンカーがシンボル x の負の静的 TLS オフセットを埋め込むよう、GOT に対する R_SPARC_TLS_TPOFF32 の再配置は、未処理の状態に置かれます。ld 命令には R_SPARC_TLS_IE_LD 再配置によってタグが付けられ、add 命令には R_SPARC_TLS_IE_ADD 再配置によってタグが付けられます。

注 – R_SPARC_TLS_IE_ADD 再配置によってタグが付けられた add 命令の GOT ポインタとして使用されるレジスタは、この命令内の最初のレジスタでなければなりません。このように指定することで、リンカーはコード変換時に GOT ポインタであるレジスタを識別できるようになります。

---

64 ビット SPARC: Initial Executable (IE)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている IE モデルを実装します。

表 8–5 64 ビット SPARC: Initial Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル

表 8–5 64 ビット SPARC: Initial Executable スレッド固有変数のアクセスコード (続き)

`# %l7 - initialized to GOT pointer, %g7 - thread pointer`		
`0x00 sethi %hi(@tpoff(x)), %o0`	R_SPARC_TLS_IE_HI22	x
`0x04 or %o0, %lo(@tpoff(x)), %o0`	R_SPARC_TLS_IE_LO10	x
`0x08 ldx [%l7 + %o0], %o0`	R_SPARC_TLS_IE_LD	x
`0x0c add %g7, %o0, %o0`	R_SPARC_TLS_IE_ADD	x
`# %o0 - contains address of TLS variable`		
	未処理の再配置	シンボル
GOT[n]	R_SPARC_TLS_TPOFF64	x

SPARC: Local Executable (LE)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている LE モデルを実装します。

表 8–6 SPARC: Local Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
`# %g7 - thread pointer`		
`0x00 sethi %hix(@tpoff(x)), %o0`	R_SPARC_TLS_LE_HIX22	x
`0x04 xor %o0,%lo(@tpoff(x)),%o0`	R_SPARC_TLS_LE_LOX10	x
`0x08 add %g7, %o0, %o0`	<none>	
`# %o0 - contains address of TLS variable`		

`sethi` 命令は `R_SPARC_TLS_LE_HIX22` 再配置を生成し、`xor` 命令は `R_SPARC_TLS_LE_LOX10` 再配置を生成します。リンカーは、実行可能ファイルで定義されたシンボルの静的な TLS オフセットに、これらの再配置を直接結合します。実行時には、再配置処理は不要です。

SPARC: スレッド固有領域の再配置のタイプ

次の表に、SPARC 用に定義された TLS 再配置を示します。表内の説明では、次の表記が使用されています。

`@dtlndx(x)`
: `TLS_index` 構造体を保持するために、GOT 内に連続した 2 つのエントリを割り当てます。この情報は、`__tls_get_addr()` に渡されます。このエントリを参照する命令は、2 つの GOT エントリのうちの最初のエントリのアドレスに結合されます。

@tmndx(x)
TLS_index 構造体を保持するために、GOT 内に連続した 2 つのエントリを割り当てます。この情報は、__tls_get_addr() に渡されます。この構造体の ti_tlsoffset フィールドは 0 に設定され、ti_moduleid は実行時に埋め込まれます。__tls_get_addr () 呼び出しは、動的な TLS ブロックの開始オフセットを返します。

@dtpoff(x)
TLS ブロックからの相対 tlsoffset を計算します。

@tpoff(x)
静的な TLS ブロックからの負の相対 tlsoffset を計算します。この値は、TLS アドレスを計算するためにスレッドポインタに追加されます。

@dtpmod(x)
TLS シンボルを含むオブジェクトの識別子を計算します。

表 8–7 SPARC: スレッド固有領域の再配置のタイプ

名前	値	フィールド	計算
R_SPARC_TLS_GD_HI22	56	T-simm22	@dtlndx(S + A) >> 10
R_SPARC_TLS_GD_LO10	57	T-simm13	@dtlndx(S + A) & 0x3ff
R_SPARC_TLS_GD_ADD	58	なし	この表のあとの説明を参照してください。
R_SPARC_TLS_GD_CALL	59	V-disp30	この表のあとの説明を参照してください。
R_SPARC_TLS_LDM_HI22	60	T-simm22	@tmndx(S + A) >> 10
R_SPARC_TLS_LDM_LO10	61	T-simm13	@tmndx(S + A) & 0x3ff
R_SPARC_TLS_LDM_ADD	62	なし	この表のあとの説明を参照してください。
R_SPARC_TLS_LDM_CALL	63	V-disp30	この表のあとの説明を参照してください。
R_SPARC_TLS_LDO_HIX22	64	T-simm22	@dtpoff(S + A) >> 10
R_SPARC_TLS_LDO_LOX10	65	T-simm13	@dtpoff(S + A) & 0x3ff
R_SPARC_TLS_LDO_ADD	66	なし	この表のあとの説明を参照してください。
R_SPARC_TLS_IE_HI22	67	T-simm22	@got(@tpoff(S + A)) >> 10
R_SPARC_TLS_IE_LO10	68	T-simm13	@got(@tpoff(S + A)) & 0x3ff
R_SPARC_TLS_IE_LD	69	なし	この表のあとの説明を参照してください。
R_SPARC_TLS_IE_LDX	70	なし	この表のあとの説明を参照してください。
R_SPARC_TLS_IE_ADD	71	なし	この表のあとの説明を参照してください。
R_SPARC_TLS_LE_HIX22	72	T-imm22	(@tpoff(S + A) ^0xffffffffffffffff) >> 10

表 8–7 SPARC: スレッド固有領域の再配置のタイプ (続き)

名前	値	フィールド	計算
R_SPARC_TLS_LE_LOX10	73	T-simm13	(@tpoff(S + A) & 0x3ff) | 0x1c00
R_SPARC_TLS_DTPMOD32	74	V-word32	@dtpmod(S + A)
R_SPARC_TLS_DTPMOD64	75	V-word64	@dtpmod(S + A)
R_SPARC_TLS_DTPOFF32	76	V-word32	@dtpoff(S + A)
R_SPARC_TLS_DTPOFF64	77	V-word64	@dtpoff(S + A)
R_SPARC_TLS_TPOFF32	78	V-word32	@tpoff(S + A)
R_SPARC_TLS_TPOFF64	79	V-word64	@tpoff(S + A)

いくつかの再配置型には、単純な計算を超えたセマンティクスが存在します。

R_SPARC_TLS_GD_ADD
: この再配置は、GD コードシーケンスの add 命令にタグを付けます。GOT ポインタに使用されるレジスタは、シーケンス内の最初のレジスタです。この再配置によってタグが付けられる命令は、R_SPARC_TLS_GD_CALL 再配置によってタグが付けられる call 命令の前に置かれます。この再配置は、リンク編集時に TLS モデルを移行するために使用されます。

R_SPARC_TLS_GD_CALL
: この再配置は、__tls_get_addr() 関数を参照する R_SPARC_WPLT30 再配置と同じように処理されます。この再配置は、 GD コードシーケンスの一部です。

R_SPARC_LDM_ADD
: この再配置は、LD コードシーケンスの最初の add 命令にタグを付けます。GOT ポインタに使用されるレジスタは、シーケンス内の最初のレジスタです。この再配置によってタグが付けられる命令は、R_SPARC_TLS_GD_CALL 再配置によってタグが付けられる call 命令の前に置かれます。この再配置は、リンク編集時に TLS モデルを移行するために使用されます。

R_SPARC_LDM_CALL
: この再配置は、__tls_get_addr() 関数を参照する R_SPARC_WPLT30 再配置と同じように処理されます。この再配置は、LD コードシーケンスの一部です。

R_SPARC_LDO_ADD
: この再配置は、LD コードシーケンス内の最後の add 命令にタグを付けます。コードシーケンスの先頭で計算されるオブジェクトアドレスを含むレジスタは、この命令における最初のレジスタです。この再配置により、リンカーはコード変換時にレジスタを識別できるようになります。

R_SPARC_TLS_IE_LD
: この再配置は、32 ビットの IE コードシーケンス内の ld 命令にタグを付けます。この再配置は、リンク編集時に TLS モデルを移行するために使用されます。

R_SPARC_TLS_IE_LDX
: この再配置は、64 ビットの IE コードシーケンス内の ldx 命令にタグを付けます。この再配置は、リンク編集時に TLS モデルを移行するために使用されます。

R_SPARC_TLS_IE_ADD
: この再配置は、IE コードシーケンス内の add 命令にタグを付けます。GOT ポインタに使用されるレジスタは、シーケンス内の最初のレジスタです。

32 ビット x86: スレッド固有変数へのアクセス

x86 では、TLS へのアクセスに次のコードシーケンスモデルを使用できます。

32 ビット x86: General Dynamic (GD)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている GD モデルを実装します。

表 8–8　32 ビット x86: General Dynamic スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
0x00 leal x@tlsgd(,%ebx,1), %eax 0x07 call x@tlsgdplt # %eax - contains address of TLS variable	R_386_TLS_GD R_386_TLS_GD_PLT	x x
	未処理の再配置	シンボル
GOT[n] GOT[n + 1]	R_386_TLS_DTPMOD32 R_386_TLS_DTPOFF32	x

leal 命令は R_386_TLS_GD 再配置を生成します。この再配置は、変数 x の TLS_index 構造体を保持する領域を GOT 内に割り当てるよう、リンカーに指示します。リンカーは、この新しい GOT エントリに GOT からの相対オフセットを代入することによって、この再配置を処理します。

読み込みオブジェクトインデックスと x の TLS ブロックインデックスは実行時まで不明なため、リンカーは、実行時リンカーによって処理されるように、GOT に対して R_386_TLS_DTPMOD32 再配置と R_386_TLS_DTPOFF32 再配置を設定します。生成された GOT エントリのアドレスは、___tls_get_addr() 呼び出しのためにレジスタ %eax に読み込まれます。

call 命令は、R_386_TLS_GD_PLT 再配置を生成します。この再配置は、___tls_get_addr () 関数の呼び出しを結合するようリンカーに指示し、call 命令を GD コードシーケンスに関連付けます。

call 命令は、leal 命令の直後に配置する必要があります。この要件は、コード変換を可能にするために必要です。

x86: Local Dynamic (LD)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている LD モデルを実装します。

表 8–9　32 ビット x86: Local Dynamic スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
0x00 leal x1@tlsldm(%ebx), %eax 0x06 call x1@tlsldmplt # %eax - contains address of TLS block of current object 0x10 leal x1@dtpoff(%eax), %edx # %edx - contains address of local TLS variable x1 0x20 leal x2@dtpoff(%eax), %edx # %edx - contains address of local TLS variable x2	R_386_TLS_LDM R_386_TLS_LDM_PLT R_386_TLS_LDO_32 R_386_TLS_LDO_32	x1 x1 x1 x2
	未処理の再配置	シンボル
GOT[n] GOT[n + 1]	R_386_TLS_DTPMOD32 <none>	x

最初の leal 命令は R_386_TLS_LDM 再配置を生成します。この再配置は、現在のオブジェクトの TLS_index 構造体を保持する領域を GOT に割り当てるように、リンカーに指示します。リンカーは、この新しいリンクテーブルエントリに GOT からの相対オフセットを代入することによって、この再配置を処理します。

読み込みオブジェクトインデックスは実行時まで不明です。したがって、R_386_TLS_DTPMOD32 再配置が作成され、構造体の ti_tlsoffset フィールドにゼロが埋め込まれます。call 命令には、R_386_TLS_LDM_PLT 再配置によってタグが付けられます。

各局所シンボルの TLS オフセットはリンク編集時に認識されるため、リンカーはこれらの値を直接埋め込みます。

手続きが複数の局所シンボルを参照する場合には、コンパイラは TLS ブロックの基底アドレスを取得するコードを 1 度だけ生成します。以後、各シンボルのアドレスの計算にはこの基底アドレスが使用され、個別にライブラリを呼び出すことはありません。

32 ビット x86: Initial Executable (IE)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている IE モデルを実装します。

IE モデルには 2 つのコードシーケンスが存在します。その 1 つは、GOT ポインタを使用する、位置に依存しないコード用です。もう 1 つのシーケンスは、GOT ポインタを使用しない、位置に依存するコード用です。

表 8–10　32 ビット x86: 位置に依存しない Initial Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
0x00 movl %gs:0, %eax 0x06 addl x@gotntpoff(%ebx), %eax # %eax - contains address of TLS variable	<none> R_386_TLS_GOTIE	 x
	未処理の再配置	シンボル
GOT[n]	R_386_TLS_TPOFF	x

addl 命令は R_386_TLS_GOTIE 再配置を生成します。この再配置は、シンボル x の静的な TLS オフセットを保存する領域を GOT 内に作成するように、リンカーに指示します。このとき、GOT テーブルに対する R_386_TLS_TPOFF 再配置は、未処理の状態に置かれます。あとで、実行時リンカーがシンボル x の静的な TLS オフセットを埋め込みます。

表 8–11　32 ビット x86: 位置に依存する Initial Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
0x00 movl %gs:0, %eax 0x06 addl x@indntpoff, %eax # %eax - contains address of TLS variable	<none> R_386_TLS_IE	 x
	未処理の再配置	シンボル
GOT[n]	R_386_TLS_TPOFF	x

addl 命令は R_386_TLS_IE 再配置を生成します。この再配置は、シンボル x の静的な TLS オフセットを保存する領域を GOT 内に作成するように、リンカーに指示しま

す。このシーケンスと位置に依存しない形式との主な違いは、GOT ポインタレジスタのオフセットを使用せず、作成される GOT エントリに直接、命令が結合されることです。このとき、GOT に対する R_386_TLS_TPOFF 再配置は、未処理の状態に置かれます。あとで、実行時リンカーがシンボル x の静的な TLS オフセットを埋め込みます。

次の 2 つのシーケンスに示すように、メモリー参照にオフセットを直接埋め込むことによって、変数 x の (アドレスではなく) 内容を読み込むことができます。

表 8–12　32 ビット x86: 位置に依存しない Initial Executable 動的スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル

`0x00 movl  x@gotntpoff(%ebx), %eax`
`0x06 movl  %gs:(%eax), %eax`
 	
`# %eax - contains address of TLS variable`
	R_386_TLS_GOTIE
	<none>
		x

	未処理の再配置	シンボル
GOT[n]	R_386_TLS_TPOFF	x

表 8–13　32 ビット x86: 位置に依存しない Initial Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル

`0x00 movl  x@indntpoff, %ecx`
`0x06 movl  %gs:(%ecx), %eax`
 	
`# %eax - contains address of TLS variable`
	R_386_TLS_IE
	<none>
		x

	未処理の再配置	シンボル
GOT[n]	R_386_TLS_TPOFF	x

最後のシーケンスで、%ecx レジスタではなく %eax レジスタを使用すると、最初の命令は 5 バイト長または 6 バイト長になる可能性があります。

32 ビット x86: Local Executable (LE)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている LE モデルを実装します。

表 8–14　32 ビット x86: Local Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル

`0x00 movl  %gs:0, %eax`
`0x06 leal  x@ntpoff(%eax), %eax`
 	
`# %eax - contains address of TLS variable`
	`<none>`
	`R_386_TLS_LE`
		
		`x`

`movl` 命令は `R_386_TLS_LE_32` 再配置を生成します。リンカーは、実行可能ファイルで定義されたシンボルの静的な TLS オフセットに、この再配置を直接結合します。実行時には処理は不要です。

次の命令シーケンスを使用すると、同じ再配置により変数 `x` の (アドレスではなく) 内容にアクセスできます。

表 8–15　32 ビット x86: Local Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル

`0x00 movl  %gs:0, %eax`
`0x06 movl  x@ntpoff(%eax), %eax`
 	
`# %eax - contains address of TLS variable`
	`<none>`
	`R_386_TLS_LE`
		
		`x`

次のシーケンスを使用すると、変数のアドレスの計算ではなく、その変数からの読み込みやその変数への保存を実行できます。この例では、`x@ntpoff` による式を即値としてではなく絶対アドレスとして使用しています。

表 8–16　32 ビット x86: Local Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル

`0x00 movl %gs:x@ntpoff, %eax`
 	
`# %eax - contains address of TLS variable`
	`R_386_TLS_LE`	`x`

32 ビット x86: スレッド固有領域の再配置のタイプ

次の表に、x86 用に定義された TLS 再配置を示します。表内の説明では、次の表記が使用されています。

@tlsgd(x)
TLS_index 構造体を保持するために、GOT 内に連続した 2 つのエントリを割り当てます。この構造体は、___tls_get_addr() に渡されます。このエントリを参照する命令は、2 つの GOT エントリのうちの最初のエントリに結合されます。

@tlsgdplt(x)
この再配置は、___tls_get_addr() 関数を参照する R_386_PLT32 再配置と同じように処理されます。

@tlsldm(x)
TLS_index 構造体を保持するために、GOT 内に連続した 2 つのエントリを割り当てます。この構造体は、___tls_get_addr() に渡されます。TLS_index の ti_tlsoffset フィールドは 0 に設定され、ti_moduleid は実行時に埋められます。___tls_get_addr() 呼び出しは、動的な TLS ブロックの開始オフセットを返します。

@gotntpoff(x)
GOT に 1 つのエントリを割り当ててから、静的な TLS ブロックからの負の相対 tlsoffset を使用してこのエントリを初期化します。このシーケンスは、R_386_TLS_TPOFF 再配置を使用して実行時に行われます。

@indntpoff(x)
この式は @gotntpoff に似ていますが、位置に依存するコードで使用されます。@gotntpoff は、movl 命令内または addl 命令内の GOT の先頭からの相対 GOT スロットアドレスに解決されます。@indntpoff は、絶対 GOT スロットアドレスに解決されます。

@ntpoff(x)
静的な TLS ブロックからの負の相対 tlsoffset を計算します。

@dtpoff(x)
TLS ブロックからの相対 tlsoffset を計算します。この値は加数の即値として使用され、特定のレジスタには関連付けられません。

@dtpmod(x)
TLS シンボルを含むオブジェクトの識別子を計算します。

表 8–17　32 ビット x86: スレッド固有領域の再配置のタイプ

名前	値	フィールド	計算
R_386_TLS_GD_PLT	12	Word32	@tlsgdplt
R_386_TLS_LDM_PLT	13	Word32	@tlsldmplt
R_386_TLS_TPOFF	14	Word32	@ntpoff(S)
R_386_TLS_IE	15	Word32	@indntpoff(S)

表 8–17 32 ビット x86: スレッド固有領域の再配置のタイプ (続き)

名前	値	フィールド	計算
R_386_TLS_GOTIE	16	Word32	@gotntpoff(S)
R_386_TLS_LE	17	Word32	@ntpoff(S)
R_386_TLS_GD	18	Word32	@tlsgd(S)
R_386_TLS_LDM	19	Word32	@tlsldm(S)
R_386_TLS_LDO_32	32	Word32	@dtpoff(S)
R_386_TLS_DTPMOD32	35	Word32	@dtpmod(S)
R_386_TLS_DTPOFF32	36	Word32	@dtpoff(S)

x64: スレッド固有変数へのアクセス

x64 では、TLS へのアクセスに次のコードシーケンスモデルを使用できます。

x64: General Dynamic (GD)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている GD モデルを実装します。

表 8–18 x64: General Dynamic スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
0x00 .byte 0x66 0x01 leaq x@tlsgd(%rip), %rdi 0x08 .word 0x666 0x0a rex64 0x0b call __tls_get_addr@plt # %iax - contains address of TLS variable	<none> R_AMD64_TLSGD <none> <none> R_AMD64_PLT32	 x __tls_get_addr
	未処理の再配置	シンボル
GOT[n] GOT[n + 1]	R_AMD64_DTPMOD64 R_AMD64_DTPOFF64	x x

__tls_get_addr() 関数は、tls_index 構造体のアドレスという 1 つのパラメータを取ります。x@tlsgd(%rip) による式と関連付けられた R_AMD64_TLSGD 再配置は、tls_index 構造体を GOT 内で割り当てるように、リンカーに指示します。tls_index 構造体で必要な 2 つの要素は、連続する GOT エントリである GOT[n] および GOT[n+1] で保持されます。これらの GOT エントリは、R_AMD64_DTPMOD64 再配置と R_AMD64_DTPOFF64 再配置に関連付けられます。

アドレス 0x00 の命令は、最初の GOT エントリのアドレスを計算します。この計算により、リンク編集時に明らかになる GOT の最初の PC 相対アドレスが現在の命令のポインタに追加されます。結果は、%rdi レジスタを使用して __tls_get_addr() 関数に渡されます。

注 – leaq 命令は、最初の GOT エントリのアドレスを計算します。この計算は、リンク編集時に決定された GOT の PC 相対アドレスを現在の命令のポインタに追加して行われます。.byte、.word、および .rex64 接頭辞によって、命令シーケンス全体で 16 バイトを占めることが確実になります。接頭辞が使用されるのは、コードに悪影響を及ぼすことがないからです。

---

x64: Local Dynamic (LD)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている LD モデルを実装します。

表 8–19 x64: Local Dynamic スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
0x00 leaq x1@tlsld(%rip), %rdi 0x07 call __tls_get_addr@plt # %rax - contains address of TLS block 0x10 leaq x1@dtpoff(%rax), %rcx # %rcx - contains address of TLS variable x1 0x20 leaq x2@dtpoff(%rax), %r9 # %rcx - contains address of TLS variable x2	R_AMD64_TLSLD R_AMD64_PLT32 R_AMD64_DTOFF32 R_AMD64_DTOFF32	x1 __tls_get_addr x1 x2
	未処理の再配置	シンボル
GOT[n]	R_AMD64_DTMOD64	x1

最初の 2 つの命令は、パッドはありませんが、一般的な動的モデルに使用されるコードシーケンスと同じです。2 つの命令は必ず連続させてください。x1@tlsld(%rip) シーケンスは、シンボル x1 の tls_index エントリを生成します。このインデックスはオフセットがゼロの x1 を含む現在のモジュールを参照します。リンカーは、オブジェクト R_AMD64_DTMOD64 の再配置を作成します。

オフセットは別々に読み込まれるので、R_AMD64_DTOFF32 再配置は不要です。x1@dtpoff による式は、シンボル x1 のオフセットにアクセスするために使用さ

れます。この命令をアドレス 0x10 として使用して、完全なオフセットが読み込まれ %rax() 内の __tls_get_addr 呼び出しの結果に追加されて結果が %rcx に生成されます。x1@dtpoff による式は、R_AMD64_DTPOFF32 再配置を作成します。

次の命令を使用すると、変数のアドレスを計算するのではなく、変数の値を読み込むことができます。この命令は、元の leaq 命令と同じ再配置を作成します。

```
movq  x1@dtpoff(%rax), %r11
```

TLS ブロックのベースアドレスがレジスタ内で保持されていると、保護されているスレッド固有変数のアドレスの読み込み、保存、または計算には 1 つの命令が必要となります。

固有の動的モデルを使用した場合には、一般的な動的モデルを使用した場合よりも利点があります。すべての追加スレッド固有変数アクセスに必要なのは、3 つの新しい命令だけです。さらに、GOT エントリの追加や実行時の再配置は必要ありません。

x64: Initial Executable (IE)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている IE モデルを実装します。

表 8–20 x64: Initial Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル
0x00 movq %fs:0, %rax 0x09 addq x@gottpoff(%rip), %rax # %rax - contains address of TLS variable	<none> R_AMD64_GOTTPOFF	 x
	未処理の再配置	シンボル
GOT[n]	R_AMD64_TPOFF64	x

シンボル x の R_AMD64_GOTTPOFF 再配置は、リンカーに GOT エントリおよび関連する R_AMD64_TPOFF64 再配置の生成を要求します。その後、この命令は x@gottpoff(%rip) 命令の最後からの GOT エントリの相対オフセットを使用します。R_AMD64_TPOFF64 再配置は、現在読み込まれているモジュールから決定されるシンボル x の値を使用します。オフセットは GOT エントリに書き込まれたあとで、addq 命令によって読み込まれます。

x のアドレスではなく x の内容を読み込む場合は、次のシーケンスを使用できます。

表 8–21 x64: Initial Executable スレッド固有変数のアクセスコード II

コードシーケンス	初期の再配置	シンボル

`0x00 movq  x@gottpoff(%rip), %rax`
`0x06 movq  %fs:(%rax), %rax`
 	
`# %rax - contains contents of TLS variable`
	`R_AMD64_GOTTPOFF`
	`<none>`
		x

	未処理の再配置	シンボル
`GOT[n]`	`R_AMD64_TPOFF64`	x

x64: Local Executable (LE)

このコードシーケンスは、331 ページの「スレッド固有領域のアクセスモデル」で説明されている LE モデルを実装します。

表 8–22 x64: Local Executable スレッド固有変数のアクセスコード

コードシーケンス	初期の再配置	シンボル

`0x00 movq %fs:0, %rax`
`0x06 leaq  x@tpoff(%rax), %rax`
 	
`# %rax - contains address of TLS variable`
	`<none>`
	`R_AMD64_TPOFF32`
		x

TLS 変数のアドレスではなく TLS 変数の内容を読み込む場合は、次のシーケンスを使用できます。

表 8–23 x64: Local Executable スレッド固有変数のアクセスコード II

コードシーケンス	初期の再配置	シンボル

`0x00 movq %fs:0, %rax`
`0x06 movq  x@tpoff(%rax), %rax`
 	
`# %rax - contains contents of TLS variable`
	`<none>`
	`R_AMD64_TPOFF32`
		x

次のシーケンスはより短いものです。

表 8–24 x64: Local Executable スレッド固有変数のアクセスコード III

コードシーケンス	初期の再配置	シンボル
0x00 movq %fs:x@tpoff, %rax # %rax - contains contents of TLS variable	R_AMD64_TPOFF32	x

x64: スレッド固有領域の再配置のタイプ

次の表に、x64 用に定義された TLS 再配置を示します。表内の説明では、次の表記が使用されています。

@tlsgd(%rip)
: TLS_index 構造体を保持するために、GOT 内に連続した 2 つのエントリを割り当てます。この構造体は、__tls_get_addr() に渡されます。この命令は、正確で一般的な動的コードシーケンス内でのみ使用できます。

@tlsld(%rip)
: TLS_index 構造体を保持するために、GOT 内に連続した 2 つのエントリを割り当てます。この構造体は、__tls_get_addr() に渡されます。実行時、オブジェクトの ti_offset オフセットフィールドはゼロに設定され、ti_module オフセットは初期化されます。__tls_get_addr() 関数の呼び出しは、動的な TLS ブロックの開始オフセットを返します。この命令は、正確なコードシーケンス内で使用できます。

@dtpoff
: 変数を含む TLS ブロックの最初からの変数の相対オフセットを計算します。計算値は加数の即値として使用され、特定のレジスタには関連付けられません。

@dtpmod(x)
: TLS シンボルを含むオブジェクトの識別子を計算します。

@gottpoff(%rip)
: 最初の TLS ブロック内の変数のオフセットを保持する GOT 内のエントリを割り当てます。このオフセットは TLS ブロックの端、%fs:0 から計算されます。この演算子は、movq または addq 命令とだけ使用できます。

@tpoff(x)
: TLS ブロックの端、%fs:0 からの変数の相対オフセットを計算します。GOT エントリは作成されません。

表 8–25 x64: スレッド固有領域の再配置のタイプ

名前	値	フィールド	計算
R_AMD64_DPTMOD64	16	Word64	@dtpmod(s)

表 8–25 x64: スレッド固有領域の再配置のタイプ (続き)

名前	値	フィールド	計算
R_AMD64_DTPOFF64	17	Word64	@dtpoff(s)
R_AMD64_TPOFF64	18	Word64	@tpoff(s)
R_AMD64_TLSGD	19	Word32	@tlsgd(s)
R_AMD64_TLSLD	20	Word32	@tlsld(s)
R_AMD64_DTPOFF32	21	Word32	@dtpoff(s)
R_AMD64_GOTTPOFF	22	Word32	@gottpoff(s)
R_AMD64_TPOFF32	23	Word32	@gottpoff(s)

◆ ◆ ◆ 第 9 章

mapfile のオプション

リンカーは、再配置可能オブジェクトの入力セクションを、作成中の出力ファイル内のセグメントに、自動的にかつ効率的に対応付けします。-M オプションで関連するマップファイル (mapfile) を指定すると、リンカーがデフォルトで行う対応付けを変更することができます。また、mapfile を使用して、新規セグメントの作成、属性の変更、およびシンボルのバージョン情報管理情報の指定を実行できます。

注 – mapfile オプションを使用すると、実行されない出力ファイルを簡単に作成できます。リンカーは、mapfile オプションなしでも、正しい出力ファイルを作成できます。

---

mapfiles のサンプルは、/usr/lib/ld ディレクトリにあります。

mapfile の構造と構文

次の基本的な指令を mapfile に入力できます。

- セグメント宣言。
- 対応付け指令。
- セクションからセグメントへの順序付け。
- サイズシンボル宣言。
- ファイル制御指令。

それぞれの指令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。

通常、セグメント宣言の後に、対応付け指令を記述します。セグメントを宣言してから、セクションがそのセグメントの一部分になる条件を定義します。対応付け先のセグメントを最初に宣言せずに、対応付け指令あるいはサイズシンボル宣言を入

力した場合、組み込みのセグメント以外のセグメントには、デフォルト属性が付与されます。この種のセグメントは、「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令は、mapfile のどこにでも入れることができます。

以後の節では、それぞれの指令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「なしまたはそれ以上」を意味する。
- { ... }+ は「1 つまたはそれ以上」を意味する。
- [...] は「省略可能」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 (.) は文字として処理される。たとえば、.bss は正当な名前です。
- 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る。それ以外のものには大文字と小文字の区別はない。
- 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、出力ファイルに新しいセグメントを作成したり、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、あるいは次に述べる 4 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次の構文で行います。

```
segment_name = {segment_attribute_value}*;
```

各 segment_name に対して、任意の数の segment_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。セグメント属性ごとに 1 つの値だけを指定できます。セグメント属性および有効な値を、次の表に示します。

表 9–1　Mapfile セグメント属性

属性	値
segment_type	LOAD | NOTE STACK
segment_flags	? [E] [N] [O] [R] [W] [X]
virtual_address	V *number*
physical_address	P*number*
length	L*number*
rounding	R*number*
alignment	A*number*

4 つの組み込みセグメントが存在し、次のデフォルト属性値を保持します。

- text – LOAD、?RX、virtual_address と physical_address と length は指定なし。alignment 値は CPU タイプごとにデフォルトに設定。
- data – LOAD、?RWX、virtual_address と physical_address と length は指定なし。alignment 値は CPU タイプごとにデフォルトに設定。
- bss – 無効、LOAD、?RWX、virtual_address と physical_address と length は指定なし。alignment 値は CPU タイプごとにデフォルトに設定。
- note – NOTE.

デフォルトでは、bss セグメントは無効に設定されています。この唯一の入力部分である SHT_NOBITS タイプのセクションは、データセグメント内でキャプチャーされます。表 7–5 セクションの詳細は、Table 7–5 を参照してください。bss セグメントの作成を有効にするときは、もっとも単純な bss 宣言だけでかまいません。

```
bss =;
```

すべての SHT_NOBITS セクションは、データセグメントではなく、このセグメントによりキャプチャーされるようになります。もっとも単純な場合、ほかのセグメントにも適用されるデフォルトを使用してこのセグメントの整列が行われます。宣言を実行してセグメントの追加属性を指定することにより、セグメント作成を有効にしたり、指定した属性を付与したりすることもできます。

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。364 ページの「mapfile オプションのデフォルト」を参照してください。

セグメント宣言を入力する場合、次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 のいずれかです。未指定の場合、セグメントタイプはデフォルトの LOAD に設定されます。
- 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 は順番を表します。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ません。
- LOAD セグメントの segment_flags 値は、デフォルトで RWX になります。
- 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ません。
- STACK 値の segment_type が 1 つ許可されます。segment_flags から選択されたセグメントのアクセス要求だけを指定できます。
- 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 と整列値はデフォルト、そして長さは無制限になります。

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。

- LOAD セグメントには、virtual_address 値または physical_address 値、および最大セグメント長値を明示的に指定できます。
- セグメントに ? の segment_flags 値があって後に何もない場合、値は読み取り不可、書き込み不可、および実行不可になります。
- alignment 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ます。この整列は、整列の指定されたセグメントにだけ影響します。その他のセグメントは、その alignment 値が変更されないかぎり、デフォルトの整列が使われます。
- 属性値 virtual_address、physical_address、length のいずれかが設定されていない場合、リンカーは出力ファイルの作成時にこれらの値を計算します。
- セグメントに対して alignment 値が指定されていない場合、整列が組み込みのデフォルトに設定されます。デフォルトは CPU により異なり、ソフトウェアのバージョンによっても異なる場合があります。
- 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 の方が優先されます。
- 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、デフォルトの整列値が設定されます。
- rounding 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ます。この値は、値の指定対象のセグメントにしか効力はありません。値が入力されないと、丸めは行われません。

注 – virtual_address 値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。システムカーネルの場合、この方法で正しい結果が生成されます。exec(2) を介して開始するファイルの場合、この方法では、セグメントがページ境界に対応する正しいオフセットを保持しないため、不正な出力ファイルが作成されることになります。

?E フラグにより、空のセグメントが作れます。この空のセグメントには、関連付けられたセクションが存在しません。このセグメントは、LOAD セグメントにできます。空の LOAD セグメントは、実行可能ファイルに対してのみ指定できます。これらのセグメントは、指定されたサイズと整列を保持する必要があります。これらのセグメントにより、プロセスの起動時にメモリー予約が作成されます。LOAD セグメントは複数定義できます。

?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。デフォルトでは、 ELF ヘッダーおよびプログラムヘッダーは、最初のセグメントに含まれます。これらのヘッダー内の情報は、対応付けられたイメージ内で (通常は実行時リンカーにより) 使用されます。?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されます。

?O フラグを使用すると、出力ファイル内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) という名前です。

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、再配置可能オブジェクトファイルが作成され、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配置されます。-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;
```

最初の宣言により、?O フラグがデフォルトのテキストセグメントに関連付けられます。

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;
```

次の構文を入力することもできます。

```
text: .text%bar: a.o b.o;
```

しかし、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は保証されません。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。

対応付け指令

対応付け指令は、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) のセットは、そのセグメントの「エントランス基準」と呼ばれます。出力ファイル内の指定されたセグメントにセクションを置くには、セクションがセグメントのエントランス基準に正確に合致している必要があります。

対応付け指令には次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 つの値だけを指定できます。file_name 宣言を使用して、特定の .o ファイルに由来するセクションだけに限定することも可能です。セクション属性とその有効値は次の表に示すとおりです。

表 9–2 セクション属性

セクション属性	値
section_name	任意の有効なセクション名

表 9–2 セクション属性 (続き)

セクション属性	値
section_type	$PROGBITS
	$SYMTAB
	$STRTAB
	$REL
	$RELA
	$NOTE
	$NOBITS
section_flags	? [[!]A] [[!]W] [[!]X]

対応付け指令を入力する場合、次の点に注意してください。

- 前記の section_types から 1 つの値を選択します。前記の section_types は組み込まれています。section_types の詳細については、222 ページの「セクション」を参照してください。
- section_flags 値は、A は割り当て可能、W は書き込み可能、X は実行可能です。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します。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ません。
- file_name には、ファイル名として正当な名前を *filename または archive_name(component_name) の形式で指定できます (例、/lib/libc.a(printf.o))。リンカーは、ファイル名の構文をチェックしません。
- file_name が *filename の形式になっている場合、リンカーはコマンド行からファイルの basename(1) を判断します。このベース名を使って、指定された filename との一致が実行されます。言い換えれば、mapfile で指定する filename は、コマンド行で指定されたファイル名の最後の部分だけが合致する必要があります。363 ページの「対応付けの例」を参照してください。
- リンク編集で -l オプションを使用するときに、-l オプションのあとのライブラリがカレントディレクトリ内にある場合は、そのライブラリを検出させるため、名前の前に ./ を付けるか、またはパス名全体を mapfile 内で記述する必要があります。
- 特定の出力セグメントについて複数の指令行を指定できます。たとえば、次に示す一連の指令を行うことができます。

```
S1 : $PROGBITS;
S1 : $NOBITS;
```

1 つのセグメントに対して複数の対応付け指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ります。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ます。たとえば、mapfile が次のようになっているとします。

```
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

次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 には、任意の正当な C 識別子を指定できます。リンカーは、symbol_name の構文をチェックしません。

ファイル制御指令

ファイル制御指令により、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。

対応付けの例

次にユーザー定義の mapfile の例を示します。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 9–1 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 からすべての .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。

同じ行の属性の間には「and」関係があります。たとえば、2 行目の $PROGBITS と ?AX は「and」関係です。同じセグメントの属性が複数行にある場合は、それらの間に「or」関係があります。たとえば、2 行目の $PROGBITS ?AX と 3行目の .data は「or」関係です。

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、デフォルトとして 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 に設定されています。segment_type 属性は変わらないため、警告は出ません。virtual_address 値は 0x80000000 に、最大 length 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。エントランス基準は、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ます。実際には、3 行目の monkey のエント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、alignment 値は 0x1000 に設定されています。これらの属性値の両方が変化するため、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。次の例では指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;
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。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分として 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 セクションのあとにきます。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 オプションのデフォルト

リンカーは、デフォルトの segment_attribute_values を持つ 4 つの組み込みセグメント (text、data、 bss、note)、および対応するデフォルトの対応付け指令を定義します。リンカーはデフォルトを提供するのに実際の mapfile は使いませんが、ここではデフォルトの内容を 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。

次に示す例は、リンカーのデフォルトを 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、あるいはデフォルトへの追加や変更を行なったりします。

```
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、次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ます。
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 値が次の順序になるように追加されます。

   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ます。

mapfile の各対応付け指令が読み取られる際、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令の後 (かつ、そのセグメントのデフォルトの対応付け指令の前) に、指令が追加されます。

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。デフォルト mapfile に対応するデフォルト対応付け構造が、リンカーにより使用されます。ユーザーの mapfile はすべて、デフォルト対応付け構造内に特定の値を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 9–1 に示します。「エントランス基準」ボックスは、デフォルトの対応付け指令内の情報に対応しています。「セグメント属性記述子」ボックスは、デフォルトのセグメント宣言内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。

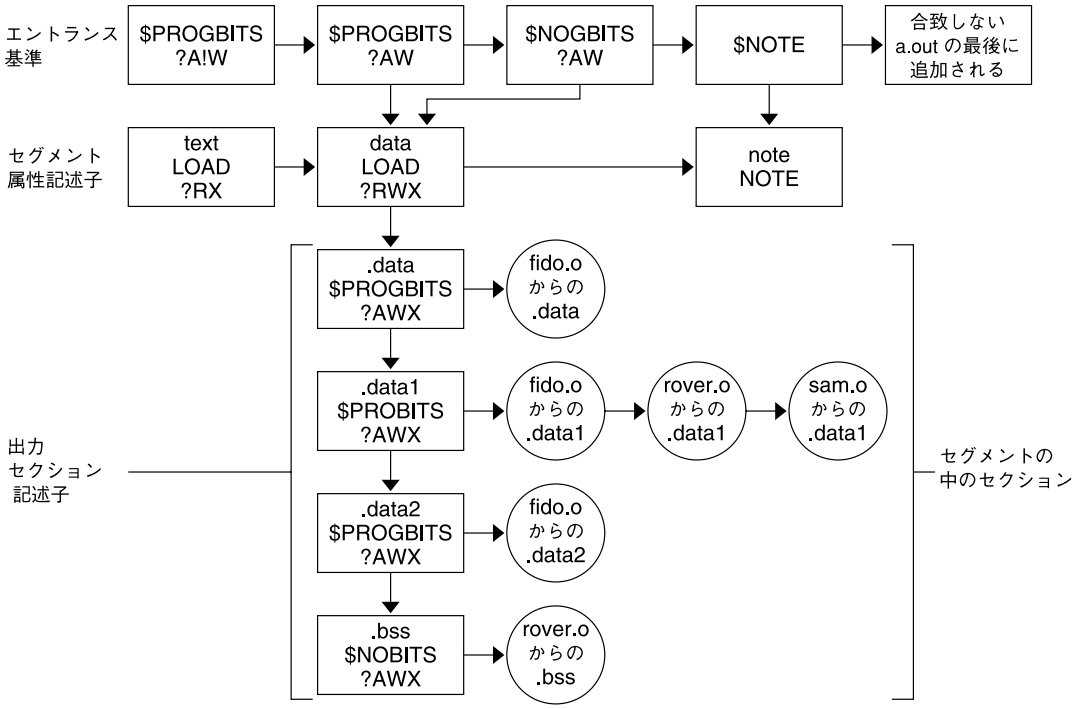


図 9–1 簡単な対応付け構造

リンカーはセクションをセグメントに対応付ける際に、次の手順で行います。

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 9–1 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。

   エント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出力ファイルの最後に置かれます。この情報に関するプログラムヘッダーエントリは作成されません。

2. セクションがセグメントの中に入る際に、リンカーは次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 $PROGBITS、section_flags が ?AWX のセクションは、図 9–1 の 2 番目のエントランス基準ボックスにあてはまり、data セグメントに置かれます。セクションは 2 番目の出力セクション記述子ボックスにちょうど一致し (.data1、$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ほかに存在する場合、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、同じセクションタイプの最後の出力セクション記述子の後に置かれます。Figure 9–1 の 図 9–1 セクションはこの方法で配置されました。

指定されたセクションタイプの出力セクション記述子がほかに存在しない場合、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 – 入力セクションが、SHT_LOUSER と SHT_HIUSER の間にユーザー定義のセクションタイプ値を保持する場合、このセクションタイプ値は $PROGBITS セクションとして処理されます。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、エント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。これらのセクションタイプを参照する指令は、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。

◆ ◆ ◆ 付 録 A

リンカーのクイックリファレンス

以降の節には、リンカーでもっとも一般的に使用するシナリオの概要が記載してあります。これは、実際に操作を行う際の「虎の巻」として利用できます。リンカーによって生成される出力モジュールの種類については、24 ページの「リンク編集」を参照してください。

記載された例には、コンパイラドライバに指定するリンカーのオプションが示されています。リンカーを起動するには、これらのオプションを使用するのがもっとも一般的です。例の中では、cc(1) を使用しています。31 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。

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、検査が行われて、必要な処理の種類が判別されます。33 ページの「入力ファイルの処理」を参照してください。

lib*x*.so の命名規約に従って命名された共有オブジェクトと、lib*x*.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。36 ページの「ライブラリの命名規約」を 参照してください。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。38 ページの「リンカーが検索するディレクトリ」を参照してください。

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

静的方法は、-d n オプションが使用された場合に選択され、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

- 再配置可能オブジェクトを作成する場合、-d n および -r オプションを使用します。

```
$ 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。25 ページの「静的実行可能ファイル」を参照してください。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、ほか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。Solaris OS ライブラリの多くの実装は、dlopen(3C) や、dlsym(3C) などの動的リンク機能に依存しています。90 ページの「追加オブジェクトの読み込み」を参照してください。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。

- 静的実行プログラムを作成するには、-r オプションを指定せずに -d n オプションを使用します。

```
$ cc -dn -o prog file1.o file2.o file3.o .....
```

-a オプションを使用して、静的実行可能プログラムの作成を指示できます。-r オプションを指定せずに -d n を使用した場合は、-a オプションと同じです。

動的方法

これは、リンカーのデフォルトの動作方法です。-d y オプションで明示的に指定することもできますが、-d n オプションを使用しない場合には、暗黙のうちに指定されます。

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、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ます。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。37 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」を参照してください。

共有オブジェクトの作成

- 共有オブジェクトを作成する場合、-G オプションを使用します。-d y は省略時にはデフォルトで暗黙指定されるため、指定する必要はありません。

- 入力再配置可能オブジェクトは、位置独立のコードから作成する必要があります。たとえば、C コンパイラは -K pic オプションで位置独立のコードを生成します。139 ページの「位置独立のコード」を参照してください。この要件を強制するには、-z text オプションを使用します。
- 使用されない再配置可能オブジェクトを含めないようにします。または、参照されていない ELF セクションを削除するようにリンカーに指示する -z ignore オプションを使用します。142 ページの「使用されない対象物の削除」を参照してください。
- 共有オブジェクトが外部での使用を意図しているものである場合、アプリケーションレジスタを使用しないことを確認します。アプリケーションレジスタを使用しない場合、外部ユーザーは、共有オブジェクトの実装を気にすることなくこれらのレジスタを自由に使用できます。たとえば、SPARC C コンパイラは、-xregs=no%appl オプションを指定すると、アプリケーションレジスタを使用しません。
- 共有オブジェクトの公開インタフェースを確立します。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限定します。これは、-mapfile とともに M オプションを指定することにより定義できます。付録 B「バージョン管理の手引き」を参照してください。
- 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します。173 ページの「バージョン管理ファイル名の管理」を参照してください。
- 自己完結型の共有オブジェクトは、もっとも柔軟性が高いです。これは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ます。自己完結を強制する場合は、-z defs オプションを指定します。51 ページの「共有オブジェクト出力ファイルの生成」を参照
- 不要な依存性を排除します。不要な依存性を検出および排除するには、-u オプションとともに ldd を使用します。35 ページの「共有オブジェクトの処理」を参照。または、-z ignore オプションを使用して、参照されるオブジェクトに対する依存性だけを記録するようにリンカーに指示します。
- 生成される共有オブジェクトがほか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遅延読み込みを指定します。91 ページの「動的依存関係の遅延読み込み」を参照してください。
- 生成中の共有オブジェクトがほかの共有オブジェクトに依存していて、これらの依存関係がデフォルトの検索場所にはない場合は、-R オプションを使用して出力ファイルにパス名を記録します。125 ページの「依存関係を持つ共有オブジェクト」を参照してください。
- 複数の再配置セクションを単一の .SUNW_reloc セクションに結合することによって、再配置処理を最適化します。-z combreloc オプションを使用します。
- 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します。86 ページの「直接結合」を参照してください。

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 -c -o foo.o -K pic -xregs=no%appl foo.c
$ cc -M mapfile -G -o libfoo.so.1 -z text -z defs -B direct -z lazyload \
-z combreloc -z ignore -R /home/lib foo.o -L. -lbar -lc
```

- 生成される共有オブジェクトを、ほかのリンク編集への入力として使用する場合は、-h オプションを使用して、内部に共有オブジェクトの実行名を記録します。123 ページの「共有オブジェクト名の記録」を参照してください。
- 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します。173 ページの「バージョン管理ファイル名の管理」を参照してください。

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 combreloc -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します。 共有性を最大限にし (143 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します (145 ページの「ページング回数の削減」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 (61 ページの「シンボル範囲の縮小」)、 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します (147 ページの「コピー再配置」)

動的実行可能プログラムの作成

- 動的実行可能プログラムを作成する場合、-G と -d n オプションは使用しません。
- -z lazyload オプションを使用して、動的実行可能プログラムの依存関係の遅延読み込みを指定します。91 ページの「動的依存関係の遅延読み込み」を参照してください。
- 不要な依存性を排除します。不要な依存性を検出および排除するには、-u オプションとともに ldd を使用します。35 ページの「共有オブジェクトの処理」を参照。または、-z ignore オプションを使用して、参照されるオブジェクトに対する依存性だけを記録するようにリンカーに指示します。
- 動的実行可能ファイルの依存関係がデフォルトの検索位置にない場合は、-R オプションを使用して出力ファイルにパス名を記録します。40 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。
- -B direct を使用して直接結合情報を確立します。86 ページの「直接結合」を参照してください。

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 .....
```

◆ ◆ ◆ 付録 B

バージョン管理の手引き

ELF オブジェクトでは、大域シンボルはほか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」の提供者として機能します。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ります。そのため、変更点を識別することは重要です。

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。内部バージョン管理の詳細は、第 5 章「アプリケーションバイナリインタフェースとバージョン管理」を参照してください。

内部バージョン管理がもっとも利用されるのは、共有オブジェクトです。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 (163 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 (167 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。この付録では、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メカニズム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メカニズムを示しています。

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。122 ページの「命名規約」を参照してください。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 (「System V アプリケーションバイナリインタフェース」など) を定義する。
- ベンダー固有の公開インタフェースを定義する。
- ベンダー固有の非公開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。157 ページの「バージョン定義の作成」を参照してください。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。161 ページの「ウィークバージョン定義の作成」を参照してください。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。この点で、しばしば利用されるのが、企業の株式銘柄のシンボルです。

非公開バージョン定義は、使用方法が制限されているかまたは確定されていないシンボルを示します。特に、「private」という語を明確に示す必要があります。

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、一意の名前およびマイナー (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方法を示しています。

SVABI.1
「System V アプリケーションバイナリインタフェース」標準インタフェースを定義します。

SUNW_1.1
Solaris OS 公開インタフェースを定義します。

SUNWprivate_1.1
Solaris OS 非公開インタフェースを定義します。

SUNW_1.1.1
Solaris OS 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どれに属するかを判別します。

- 業界標準インタフェースシンボルの規約は、公開されたヘッダー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ます。また、対応する標準の文献にも記述されています。
- ベンダーの公開インタフェースシンボルの規約は、公開されたヘッダー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ます。
- ベンダーの非公開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ません。

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、ほか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ります。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、ほか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。

ベンダーの非公開インタフェースは、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。61 ページの「シンボル範囲の縮小」を参照してください。

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。mapfile 構文の詳細は、54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。

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ます。

```
$ cat mapfile
SUNW_1.1 {                  # Release X.
        global:
                foo2;
                foo1;
        local:
                *;
};
$ 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、共有オブジェクトの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ほかの大域シンボルはすべて、自動縮小指令「*」によってローカル範囲に縮小されています。61 ページの「シンボル範囲の縮小」を参照してください。

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、ほか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、ほかのユーザーが発見して別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要件を検査するために役立ちます。114 ページの「デバッギングライブラリ」を参照してください。ただし、この既存結合要件の判定は、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要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。173 ページの「バージョン管理ファイル名の管理」を参照してください。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。ただし、元の共有オブジェクトを更新する必要がある場合は、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適切な場合があります。実装は新しいプラットフォームとの互換性を保つ必要があるので、更新はパッチで行う必要があります。

バージョンアップ共有オブジェクトの更新

内部バージョンアップによって吸収できる共有オブジェクトに対しては、互換性のある変更しか行うことができません。156 ページの「インタフェースの互換性」を参照してください。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。173 ページの「バージョン管理ファイル名の管理」を参照してください。

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 {                  # Release X+1.
        global:
                foo3;
} SUNW_1.1;

SUNW_1.1 {                  # Release X.
        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が維持され

ていることを確認してください。新しい標準インタフェースおよび元のインタフェースの定義を構築可能な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;
                            # Subversion - providing for
STAND.0.1 {                 # SUNW_1.2 and STAND.1 interfaces.
        global:
                foo3;
};
                            # Subversion - providing for
STAND.0.2 {                 # SUNW_1.1 and STAND.1 interfaces.
        global:
                foo1;
};
```

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。

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すことが求められます。この要件は、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と、実行時バージョンアップエラーが発生します163 ページの「バージョン定義への結合」を参照してください。

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できます。169 ページの「追加バージョン定義への結合」を参照してください。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 指令を使用して、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);
```

実際には、この方法でバージョン結合を行う必要はほとんどありません。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、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ファミリから参照します。

◆ ◆ ◆ 付録 C

動的ストリングトークンによる依存関係の確立

動的オブジェクトは、明示的に、またはフィルタを通して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。フィルタ、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、次の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。

- `$HWCAP`
- `$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。

ハードウェア機能固有の共有オブジェクト

動的トークン `$HWCAP` を使用して、ハードウェア機能固有の共有オブジェクトがあるディレクトリを指定することができます。このトークンは、フィルタまたは依存関係に対して使用できます。このトークンは複数のオブジェクトに拡張できるので、依存関係と使用する場合も管理できます。`dlopen(3C)` で取得された依存関係は、`RTLD_FIRST` モードでこのトークンを使用できます。このトークンを使用する明示的な依存関係は、最初に見つかった適切な依存関係を読み込みます。

パス名の指定は、`$HWCAP` トークンで終わるフルパス名で構成する必要があります。`$HWCAP` トークンで指定されたディレクトリ内の共有オブジェクトは、実行時に検査されます。これらのオブジェクトは、ハードウェア機能の要件を示す必要があります。68 ページの「ハードウェアとソフトウェア機能の特定」を参照してください。各オブジェクトは、プロセスで使用可能なハードウェア機能に対して検証されます。プロセスと使用できるオブジェクトは、ハードウェア機能値の降順で保存されます。これらのソートされたフィルティーは、フィルタ内で定義されたシンボルを解決するために使用されます。

ハードウェア機能ディレクトリ内のフィルティーには、命名に関する制限はありません。次の例で、ハードウェア機能フィルティーにアクセスするために補助フィルタ libfoo.so.1 をどのように設計するかを示します。

```
$ LD_OPTIONS='-f /opt/ISV/lib/hwcap/$HWCAP' \
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/hwcap/$HWCAP
$ elfdump -H /opt/ISV/lib/hwcap/*

/opt/ISV/lib/hwcap/filtee.so.3:

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x1000  [ SSE2 ]

/opt/ISV/lib/hwcap/filtee.so.1:

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x40  [ MMX ]

/opt/ISV/lib/hwcap/filtee.so.2:

Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag               value
       [0]  CA_SUNW_HW_1     0x800  [ SSE ]
```

フィルタ libfoo.so.1 を MMX と SSE 機能を使用できるプラットフォームで処理した場合、次のフィルティー検索順序が採用されます。

```
$ cc -o prog prog.c -R. -lfoo
$ LD_DEBUG=symbols prog
.....
01233: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
01233: symbol=foo;  lookup in file=hwcap/filtee.so.2  [ ELF ]
01233: symbol=foo;  lookup in file=hwcap/filtee.so.1  [ ELF ]
.....
```

なお、filtee.so.2 の機能値は filtee.so.1 の機能値より大きくなります。SSE2 機能を使用できないので、filtee.so.3 がシンボル検索に含まれる可能性はありません。

「フィルティー」検索の縮小

フィルタ内で $HWCAP を使用すると、1 つまたは複数の「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実装を提供できます。

指定された $HWCAP ディレクトリ内のすべての共有オブジェクトは、使用可能性を検証し、プロセスに適したものをソートするために、点検されます。ソートされると、すべてのオブジェクトは使用準備のため読み込まれます。

リンカーの -z endfiltee オプションを使用して「フィルティー」を作成して、これが使用可能な最後の「フィルティー」であることを示します。このオプションで特定されたフィルティーは、そのフィルタのフィルティーのソートリストを終了します。このフィルティーをフィルタに対して読み込んだ後は、いかなるオブジェクトもソートされません。前の例で filter.so.2 フィルティーに -z endfiltee のタグが付けられている場合、フィルティー検索は次のようになります。

```
$ LD_DEBUG=symbols prog
.....
01424: symbol=foo;  lookup in file=libfoo.so.1  [ ELF ]
01424: symbol=foo;  lookup in file=hwcap/filtee.so.2  [ ELF ]
.....
```

命令セット固有の共有オブジェクト

動的トークン $ISALIST は、実行時に展開され、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ー isalist(1) によって表されます。このトークンは、フィルタ、実行パス定義、および依存関係に対して使用できます。このトークンは複数のオブジェクトに拡張できるので、依存関係と使用する場合も管理できます。dlopen(3C) で取得された依存関係は、RTLD_FIRST モードでこのトークンを使用できます。このトークンを使用する明示的な依存関係は、最初に見つかった適切な依存関係を読み込みます。

注 – このトークンは廃止されたため、Solaris の今後のリリースで削除される可能性があります。命令セットの拡張を処理するために推奨されるテクニックについては、385 ページの「ハードウェア機能固有の共有オブジェクト」を参照してください。

$ISALIST トークンに組み込まれたストリング名はすべて、複数の文字列に効率良く複製されます。各文字列には、使用可能な命令セットの 1 つが割り当てられます。

次の例では、命令セット固有の「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 /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 "RUNPATH|AUXILIARY"
  [1]    RUNPATH   /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
      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
```

「フィルティー」検索の縮小

フィルタ内で $ISALIST を使用すると、1 つまたは複数の「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実装を提供できます。

フィルタ内でどのようなインタフェースを定義しても、目的のインタフェースを探すために、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になり得ます。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、徹底的な「フィルティー」の検索は逆効果になるかもしれません。

リンカーの -z endfiltee オプションを使用して「フィルティー」を作成して、これが使用可能な最後の「フィルティー」であることを示します。このオプションによって、該当するフィルタに対してそれ以上の「フィルティー」検索を行わないようにできます。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
```

システ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。これらのトークンは、フィルタ、「実行パス」、または依存関係の定義に利用できます。

$OSNAME は、ユーティリティー uname(1) にオプション -s を付けて実行した場合に表示されるように、オペレーティングシステムの名前を反映して展開されます。$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して展開されます。$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して展開されます。

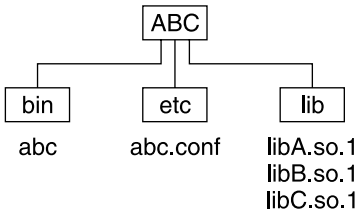
次の例は、プラットフォーム固有の「フィルティー」libfoo.so.1 にアクセスするように補助フィルタ libbar.so.1 を設計する方法を示します。

```
$ LD_OPTIONS='-f /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 /platform/$PLATFORM/lib/libbar.so.1
```

このメカニズムは、共有オブジェクト /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 OS で使用されます。

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、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製品の相互依存関係

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、この「実行パス」は次のようになります。

```
$ cc -o abc abc.c -R/opt/ABC/lib -L/opt/ABC/lib -lA
$ dump -Lv abc
  [1]    NEEDED  libA.so.1
  [2]    RUNPATH /opt/ABC/lib
```

libA.so.1 の依存関係でも同様に、実行パスは次のようになります。

```
$ cc -o libA.so.1 -G -Kpic A.c -R/opt/ABC/lib -L/opt/ABC/lib -lB
$ dump -Lv libA.so.1
  [1]    NEEDED  libB.so.1
  [2]    RUNPATH /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。

動的トークン $ORIGIN は、オブジェクトが存在するディレクトリに展開されます。このトークンは、フィルタ、「実行パス」、または依存関係の定義に利用できます。この機構を使用すると、バンドルされていないアプリケーションを再定義して、$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。

```
$ cc -o abc abc.c '-R$ORIGIN/../lib' -L/opt/ABC/lib -lA
$ dump -Lv abc
```

```
  [1]    NEEDED  libA.so.1
  [2]    RUNPATH $ORIGIN/../lib
```

$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。

```
$ cc -o libA.so.1 -G -Kpic A.c '-R$ORIGIN' -L/opt/ABC/lib -lB
$ dump -Lv libA.so.1
  [1]    NEEDED  libB.so.1
  [2]    RUNPATH $ORIGIN
```

この製品が /usr/local/ABC 内にインストールされ、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。

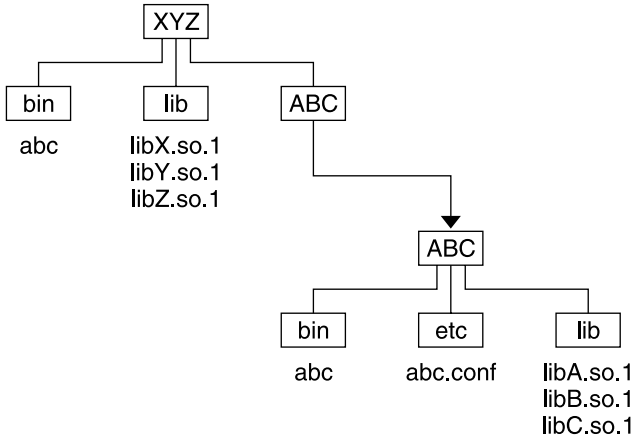
```
$ ldd -s abc
.....
  find object=libA.so.1; required by abc
    search path=$ORIGIN/../lib  (RUNPATH/RPATH from file abc)
      trying path=/usr/local/ABC/lib/libA.so.1
        libA.so.1 =>     /usr/local/ABC/lib/libA.so.1

  find object=libB.so.1; required by /usr/local/ABC/lib/libA.so.1
    search path=$ORIGIN  (RUNPATH/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もう 1 つの問題は、バンドルされていない製品同士の依存関係を表現するモデルを、どのようにして確立するかです。

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。この依存関係を確立するには、ホストパッケージインストールスクリプトを使用します。このスクリプトは ABC 製品のインストール位置へのシンボリックリンクを生成します (次の図を参照)。



図C–2 バンドルされていない製品の「相互依存関係」

XYZ 製品のバイナリおよび共有オブジェクトは、シンボリックリンクを使用して、ABC 製品との依存関係を表現できます。このリンクはその時点で、安定した参照点になります。アプリケーション xyz の場合、この「実行パス」は次のようになります。

```
$ cc -o xyz xyz.c '-R$ORIGIN/../lib:$ORIGIN/../ABC/lib' \
-L/opt/ABC/lib -lX -lA
$ dump -Lv xyz
  [1]    NEEDED  libX.so.1
  [2]    NEEDED  libA.so.1
  [3]    RUNPATH $ORIGIN/../lib:$ORIGIN/../ABC/lib
```

libX.so.1 の依存関係でも同様に、この実行パスは次のようになります。

```
$ cc -o libX.so.1 -G -Kpic X.c '-R$ORIGIN:$ORIGIN/../ABC/lib' \
-L/opt/ABC/lib -lY -lC
$ dump -Lv libX.so.1
  [1]    NEEDED  libY.so.1
  [2]    NEEDED  libC.so.1
  [3]    RUNPATH $ORIGIN:$ORIGIN/../ABC/lib
```

この製品が /usr/local/XYZ 内にインストールされている場合は、次のシンボリックリンクを確立するために、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  (RUNPATH/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  (RUNPATH/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 \
                (RUNPATH/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 \
                (RUNPATH/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  (RUNPATH/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
```

セキュリティー

セキュリティー保護されたプロセスでは、$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かぎり許可されます。ほかの相対パス名は、セキュリティーリスクを伴います。

$ORIGIN/../lib のようなパスは一定の場所 (実行可能プログラムの場所で特定される) を指しているように見えますが、それは正しくありません。同じファイルシステム内の書き込み可能なディレクトリにより、$ORIGIN を使用するセキュリティー保護されたプログラムが不当に利用される可能性があります。

次の例は、$ORIGIN がセキュリティー保護されたプロセス内で任意に拡張された場合、セキュリティー侵入が生じる可能性があることを示しています。

```
$ 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
```

ユーティリティー crle(1) を使用すれば、セキュリティー保護されたアプリケーションによる $ORIGIN の使用を可能にす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。

◆ ◆ ◆ 付 録 D

リンカーとライブラリのアップデートおよび新機能

この付録は、Solaris OS に追加されたアップデートと新機能の概要を提供します。

Solaris 10 5/08 リリース

- リンカーの `-z globalaudit` オプションを指定することで、アプリケーション内の監査を記録することによる大域監査を有効化できるようになりました。189 ページの「大域監査の記録」を参照してください。
- リンカーの新しいサポートインタフェース `ld_open()` と `ld_open64()` が追加されました。179 ページの「サポートインタフェース関数」を参照してください。

Solaris 10 8/07 リリース

- リンカーの `-z altexec64` オプションおよび `LD_ALTEXEC` 環境変数を使用して代替リンカーを実行する際の柔軟性が向上しました。31 ページの「32 ビットリンカーと 64 ビットリンカー」を参照してください。
- `mapfiles` を使用して生成されるシンボル定義を、ELF セクションに関連付けることができるようになりました。54 ページの「`mapfile` を使用した追加シンボルの定義」を参照してください。
- リンカーが、共有オブジェクト内で静的な TLS を作成できるようになりました。また、起動後の共有オブジェクト内部で静的な TLS の限定的な使用を可能にするバックアップ TLS 予約が規定されました。328 ページの「プログラムの起動」を参照してください。

Solaris 10 1/06 リリース

- x64 中規模コードモデルのサポートが提供されました。表 7–4、表 7–8、および表 7–10 を参照してください。
- dlinfo(3C) のフラグ RTLD_DI_ARGSINFO を使用すれば、コマンド行引数、環境変数、およびプロセスの補助ベクトル配列を取得できます。
- リンカーの -B nodirect オプションにより、外部参照からの直接結合をより柔軟に禁止できるようになりました。86 ページの「直接結合」を参照してください。

Solaris 10 リリース

- x64 がサポートされるようになりました。表 7–5、238 ページの「特殊セクション」、264 ページの「x64: 再配置型」、348 ページの「x64: スレッド固有変数へのアクセス」、および 352 ページの「x64: スレッド固有領域の再配置のタイプ」を参照してください。
- ファイルシステムの再構成により、多数のコンポーネントが /usr/lib から /lib に移動されました。これにより、リンカーと実行時リンカー両方のデフォルト検索パスが変更されました。38 ページの「リンカーが検索するディレクトリ」、79 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 100 ページの「セキュリティー」を参照してください。
- システムアーカイブライブラリが提供されなくなりました。したがって、静的にリンクされた実行可能ファイルは作成できなくなりました。25 ページの「静的実行可能ファイル」を参照してください。
- crle(1) の -A オプションにより、代替依存関係をより柔軟に定義できるようになりました。
- リンカーは、値なしで指定された環境変数を処理できるようになりました。27 ページの「環境変数」を参照してください。
- dlopen(3C) とともに、明示的な依存関係の定義として使用されるパス名は、すべての予約トークンを使用できるようになりました。付録 C「動的ストリングトークンによる依存関係の確立」を参照してください。予約トークンを使用するパス名は、新ユーティリティー moe(1) で評価されます。
- dlsym(3C) と新しいハンドル RTLD_PROBE によって、特定のインタフェースの存在有無を確認するための最適な方法が実現されました。93 ページの「dlopen() の代替手段の提供」を参照してください。

Solaris 9 9/04 リリース

- リンカーによって、ELF オブジェクトのハードウェアおよびソフトウェア要件をより柔軟に定義できるようになりました。245 ページの「ハードウェアおよびソフトウェア機能に関するセクション」を参照してください。
- 実行時リンク監査インタフェース `la_objfilter()` が追加されました。189 ページの「監査インタフェースの関数」を参照してください。
- 共有オブジェクトのフィルタ処理が拡張され、シンボルごとのフィルタ処理が可能になりました。127 ページの「フィルタとしての共有オブジェクト」を参照してください。

Solaris 9 4/04 リリース

- 新しいセクションタイプ `SHT_SUNW_ANNOTATE`、`SHT_SUNW_DEBUGSTR`、`SHT_SUNW_DEBUG`、および `SHT_SPARC_GOTDATA` がサポートされるようになりました。表 7–5 を参照してください。
- 新しいユーティリティー `lari(1)` により、実行時インタフェースの分析が簡単になりました。
- リンカーオプション `-z direct`、`-z nodirect` や、`DIRECT` および `NODIRECT` `mapfile` 指令などにより、直接結合をより細かく制御できるようになりました。54 ページの「`mapfile` を使用した追加シンボルの定義」と86 ページの「直接結合」を参照してください。

Solaris 9 12/03 リリース

- `ld(1)` の性能の向上によって、大規模なアプリケーションのリンク編集時間を大幅に短縮できます。

Solaris 9 8/03 リリース

- `RTLD_FIRST` フラグを使って作成された `dlopen(3C)` ハンドルを使用することで、`dlsym(3C)` のシンボル処理時間を短縮できます。110 ページの「新しいシンボルの入手」を参照してください。
- 実行時リンカーが不正プロセスを終了させるために使用する信号は、`dlinfo(3C)` フラグである `RTLD_DI_GETSIGNAL` と `RTLD_DI_SETSIGNAL` を使用して管理できるようになりました。

Solaris 9 12/02 リリース

- 文字列テーブルの圧縮がリンカーにより提供されます。これにより、 .dynstr セクションおよび .strtab セクションが縮小することがあります。このデフォルト処理は、リンカーの -z nocompstrtab オプションで無効にできます。66 ページの「文字列テーブルの圧縮」を参照してください。
- -z ignore オプションが、リンク編集時に参照されないセクションを排除するように拡張されました。142 ページの「使用されない対象物の削除」を参照してください。
- 参照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 オプションを参照してください。
- リンカーにより、拡張 ELF セクションが提供されます。214 ページの「ELF ヘッダー」、表 7–5、222 ページの「セクション」、表 7–10、および 267 ページの「シンボルテーブルセクション」を参照してください。
- protected mapfile 指令により、シンボルの可視性をより柔軟に定義できるようになりました。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。

Solaris 9 リリース

- スレッド固有領域 (TLS) のサポートが提供されます。第 8 章「スレッド固有領域 (TLS)」を参照してください。
- -z rescan オプションにより、アーカイブライブラリをリンク編集に指定する際の柔軟性が向上しました。37 ページの「コマンド行上のアーカイブの位置」を参照してください。
- -z ld32 および -z ld64 オプションにより、リンカーサポートインタフェースを使用する際の柔軟性が向上しました。178 ページの「32 ビットおよび 64 ビット環境」を参照してください。
- 補助リンカーサポートインタフェース ld_input_done()、ld_input_section()、ld_input_section64()、および ld_version() が追加されました。179 ページの「サポートインタフェース関数」を参照してください。
- 実行時リンカーにより解釈される環境変数を、構成ファイル内で指定することにより、複数のプロセスに対応させることができるようになりました。crle(1) のマニュアルページの -e および -E オプションを参照してください。
- 64 ビット SPARC オブジェクト内部で、32,768 以上のプロシージャーリンクテーブルエントリがサポートされるようになりました。315 ページの「64 ビット SPARC: プロシージャーのリンクテーブル」を参照してください。
- mdb(1) デバッガモジュールを使用することで、実行時リンカーのデータ構造の検査を、デバッグプロセスの一部として実行できます。117 ページの「デバッガモジュール」を参照してください。

- bss セグメント宣言指令により、bss セグメントをより簡単に作成できます。356 ページの「セグメントの宣言」を参照してください。

Solaris 8 07/01 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。詳細は、-u オプションを参照してください。
- さまざまな ELF ABI 拡張が追加されました。41 ページの「初期設定および終了セクション」、95 ページの「初期設定および終了ルーチン」、表 7–3、表 7–8、表 7–9、244 ページの「グループセクション」、表 7–10、表 7–20、表 7–32、表 7–33、および 289 ページの「プログラムの読み込み (プロセッサ固有)」を参照してください。
- リンカー固有の環境変数に _32 および _64 の 2 つの接尾辞が使用可能になりました。これにより、環境変数がより柔軟に使用できます。27 ページの「環境変数」を参照してください。

Solaris 8 01/01 リリース

- dladdr1() の導入により、dladdr(3C) から入手可能なシンボリック情報が拡張されました。
- 動的オブジェクトの $ORIGIN を、dlinfo(3C) から入手可能になりました。
- crle(1) で作成された実行時構成ファイルの管理が、簡単になりました。構成ファイルを検査することで、ファイル作成に使用されたコマンド行オプションが表示されます。-u オプションを指定すると、更新機能を利用できます。
- 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、プロシージャー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。Agent Manipulation Interfaces() の 199 ページの「エージェント操作インタフェース」を参照してください。この更新により rd_plt_info_t 構造体が機能拡張されます。205 ページの「プロシージャーのリンクテーブルのスキップ」の rd_plt_resolution() を参照してください。
- 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ファイルに定義することができます。356 ページの「セグメントの宣言」を参照してください。

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。95 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。
- 実行時リンカーおよびそのデバッガインタフェースが拡張され、実行時解析とコアファイル解析の性能が向上しました。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。Agent Manipulation Interfaces()の 199 ページの「エージェント操作インタフェース」を参照してください。この更新により rd_loadobj_t 構造体が拡張されます。201 ページの「読み込み可能オブジェクトの走査」を参照してください。
- ディスプレイス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できるようになりました。72 ページの「ディスプレイスメント再配置」を参照してください。
- 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 mapfile から単独で構築できます。128 ページの「標準フィルタの生成」を参照してください。
- 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(3C) を使って調べることができます。
- 新しいハンドル RTLD_SELF により、dlsym(3C) と dlinfo(3C) の検索セマンティクスが拡張されました。
- 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。86 ページの「直接結合」を参照してください。

Solaris 8 リリース

- ファイルを前もって読み込むことのできるセキュリティー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。100 ページの「セキュリティー」を参照してください。
- リンカーの -z nodefaultlib オプションおよび新ユーティリティー 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。40 ページの「実行時リンカーが検索するディレクトリ」と 81 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 指令により、-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。54 ページの「mapfile を使用した追加シンボルの定義」を参照してください。
- 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。82 ページの「動的ストリングトークン」を参照してください。

- 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。188 ページの「ローカル監査の記録」を参照してください。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。189 ページの「監査インタフェースの関数」を参照してください。
- 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 7–32 を参照してください。

索引

E

G

L

U

あ

い

う

え

お

す

せ

わ