リンカーとライブラリガイド

Part No: E38860
2013 年 1 月

ORACLE

130221@25097

目次

図目次

表目次

はじめに

Oracle Solaris オペレーティングシステム (Oracle 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およびライブラリを作成し、実行時リンカー ld.so.1(1) の支援でこれらのオブジェクトを実行できます。このマニュアルは、Oracle Solaris リンカー、実行時リンカー、および関連ツールの使用方法に関する概念を、より完全に理解したいエンジニアを対象としています。

注 – この Oracle Solaris 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を使用するシステムをサポートしています。サポートされるシステムは、Oracle Solaris OS: Hardware Compatibility Lists に記載されています。このドキュメントでは、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。

このドキュメントの x86 に関連する用語については、次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 は特に 64 ビット x86 互換 CPU 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、Oracle Solaris OS: Hardware Compatibility Listsを参照してください。

このドキュメントの x86 に関連する用語については、次を参照してください。

- 「x86」は、64 ビットおよび 32 ビットの x86 互換オブジェクト系列を指します。
- 「x64」は 64 ビットの x86 固有のオブジェクトに関連します。
- 「32 ビット x86」は 32 ビットの x86 固有のオブジェクトに関連します。

お読みになる前に

このマニュアルでは、Oracle Solaris リンカーおよび実行時リンカーの操作について説明しています。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、Oracle Solaris リンカー、実行時リンカー、および関連ツールに興味を持つ、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

このドキュメントを通して、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。

このマニュアルは次の部分に分かれています。

Oracle Solaris リンカーおよび実行時リンカーの使用

パート 1 では Oracle Solaris リンカーの使用方法について記載しています。この情報は、すべてのプログラマを対象としています。

第 1 章「Oracle Solaris リンカーの紹介」では、Oracle Solaris OS でのリンク処理の概要を紹介します。

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。

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。

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。

クイックリファレンス

パート 2 では、新規ユーザーがすぐに開始できるようにするためのクイックリファレンス情報を提供します。この情報は、すべてのプログラマを対象としています。

第 5 章「リンカーのクイックリファレンス」では、もっとも一般的に使用されるリンカーオプションの概要を提供します。

詳細情報

パート 3 では特殊なトピックについて扱います。この情報は、上級プログラマを対象としています。

第 6 章「直接結合」では、直接結合に関連する実行時シンボル検索モデルについて説明します。

第 7 章「システムのパフォーマンスを最適化するオブジェクトの構築」では、動的オブジェクトの実行時の初期設定と処理を調べ、それらの実行時パフォーマンスに影響を与える手法について説明します。

第 8 章「mapfile」では、リンカーに対するバージョン 2 の `mapfile` 指令について説明します。

第 9 章「インタフェースおよびバージョン管理」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を管理する方法について説明します。

第 10 章「動的ストリングトークンによる依存関係の確立」では、動的依存関係を定義するための予約された動的ストリングトークンを使用する方法の例を提供します。

第 11 章「拡張性メカニズム」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。

Oracle Solaris ELF アプリケーションバイナリインタフェース

パート 4 では、Oracle Solaris ELF アプリケーションバイナリインタフェース (ABI) について記載します。この情報は、上級プログラマを対象としています。

第 12 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の章です。

第 13 章「プログラムの読み込みと動的リンク」では、実行時の ELF ファイルの読み込みと管理方法について説明します。

第 14 章「スレッド固有ストレージ (TLS)」では、スレッド固有ストレージについて説明しています。

付録

付録 A「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、新しい機能と、リンカー、実行時リンカー、および関連ツールへの更新内容の概要を、変更が行われたリリースを示しながら説明します。

付録 B「System V Release 4 (バージョン 1) Mapfile」では、リンカーに対するバージョン 1 の `mapfile` 指令について説明します。この付録は、古い構文で記述された既存の `mapfiles` へのサポートを必要とするプログラマを対象としています。すべての新しいアプリケーションについては、第 8 章「mapfile」で説明されているバージョン 2 の `mapfile` 構文をお勧めします。

Oracle サポートへのアクセス

Oracle のお客様は、My Oracle Support を通じて電子的なサポートを利用することができます。詳細は、http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info を参照してください。聴覚に障害をお持ちの場合は、http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs を参照してください。

表記上の規則

次の表では、このドキュメントで使用される表記上の規則について説明します。

表 P–1　表記上の規則

字体	説明	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。 `machine_name% you have mail.`
`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。	`machine_name%` **`su`** `Password:`
aabbcc123	プレースホルダ: 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。
AaBbCc123	書名、新しい単語、および強調する単語を示します。	『ユーザーズガイド』の第 6 章を参照してください。 キャッシュは、ローカルに格納されるコピーです。 ファイルを保存しないでください。 注: いくつかの強調された項目は、オンラインでは太字で表示されます。

コマンド例のシェルプロンプト

Oracle Solaris OS に含まれるシェルで使用する、UNIX のシステムプロンプトとスーパーユーザープロンプトを次に示します。コマンド例のシェルプロンプトから、通常ユーザーと特権ユーザーのどちらがコマンドを実行すべきかがわかります。

表 P–2 シェルプロンプト

シェル	プロンプト
Bash シェル、Korn シェル、および Bourne シェル	`$`
Bash シェル、Korn シェル、および Bourne シェルのスーパーユーザー	`#`
C シェル	`machine_name%`
C シェルのスーパーユーザー	`machine_name#`

パート I

リンカーおよび実行時リンカーの使用

◆ ◆ ◆ 第 1 章

Oracle Solaris リンカーの紹介

このマニュアルは、Oracle Solaris リンカーと実行時リンカーの動作に加え、これらのユーティリティーが動作するオブジェクトについて説明しています。Oracle Solaris リンカーと実行時リンカーの基本動作は、オブジェクトを組み合わせることです。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。このファイルは、再配置可能オブジェクト、動的実行可能プログラム、共有オブジェクトのいずれかです。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
: Oracle Solaris リンカー、実行時リンカー、および関連ツールは、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。このドキュメントでは、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

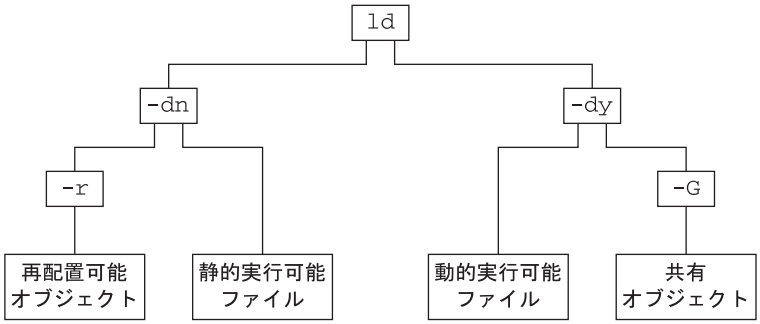
リンク編集では、一般に、コンパイラ、アセンブラ、または ld(1) によって生成されたさまざまな入力ファイルを受け取ります。リンカーは、これら入力ファイル内のデータを連結および解釈して、1 つの出力ファイルを生成します。リンカーにはさまざまなオプションを使用できますが、出力ファイル (入力再配置可能オブジェクトの連結) は次のいずれかの形式になります。

- 「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。29 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。このドキュメントでは、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。

Oracle Solaris 10 リリース以降、この OS に 32 ビット版のシステムアーカイブライブラリは含まれていません。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を処理する機能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、Oracle Solaris 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ると、これらのコンポーネントを非同期的に更新できます。Oracle Solaris リンカーと実行時リンカーはこれらのインタフェース上で動作して、実行用にアプリケーションをアセンブルします。Oracle Solaris リンカーと実行時リンカーによって処理されるすべてのコンポーネントにはバイナリインタフェースがありますが、Oracle 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、「*Oracle Solaris ABI*」と言います。

Oracle 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。

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と実行時リンカーの操作に違いはありません。このドキュメントでは、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

リンカーおよび実行時リンカーは、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、それをオーバーライドします。

注 – Oracle Solaris 10 よりも前のリリースでは、リンカーおよび実行時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
```

Oracle Solaris 10 リリース以降、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。

このドキュメントでは、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

Oracle 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、入力セクション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 12 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個別の最小単位です。

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。付録 B 「System V Release 4 (バージョン 1) Mapfile」 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 つのセクションでは、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。

クロスリンク編集

このリンカーは SPARC または x86 を対象としたクロスリンカーで、32 ビットオブジェクトまたは 64 ビットオブジェクトにリンクできます。32 ビットオブジェクトと 64 ビットオブジェクトを混在させることはできません。同様に、1 つの機械タイプのオブジェクトのみが許可されます。

通常、コマンド行オプションではリンク編集のターゲットを区別する必要はありません。リンカーは、コマンド行の最初の入力再配置可能オブジェクトの ELF 機械タイプを使用して、操作するモードを制御します。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のデフォルトは、32 ビットのネイティブターゲットです。リンク編集ターゲットを明示的に定義するには、-z target オプションを使用します。

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指定することもできます。

```
$ 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’
```

関連する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、リンク編集を続行します。

```
$ 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致命的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。

もっとも標準的に使用されるリンカーオプションについては、第 5 章「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、すべてのリンカーオプションについての詳しい説明については、ld(1) 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、あと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。

入力ファイルは直接リンク編集コマンド行に指定できますが、アーカイブライブラリと共有オブジェクトは、一般に -l オプションを使用して指定します。40 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 つのセクションで、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) を使用して構築します。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。

リンカーは、次の条件で、アーカイブから再配置可能なオブジェクトを抽出します。

- アーカイブメンバーに、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含まれている場合。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。これにより、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 12–20 を参照してください。
- リンカーの -z allextract が有効になっている場合。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では、-z weakextract オプションが有効になっていないかぎり、ウィークシンボル参照はアーカイブからオブジェクトを抽出しません。詳細は、49 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。41 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、構築中の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)` を使用すると、未使用の依存関係を確認できます。リンカーの `-z discard-unused=dependencies` オプションを使用すると、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。

コマンド行の共有オブジェクトの位置は、アーカイブ処理よりも重要性が低いですが、位置がグローバルな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。48 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 – 複数のシンボル定義は、`-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

規約により、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。140 ページの「命名規約」 および 259 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できません。

```
$ 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
```

アーカイブの指定を繰り返し決定したり管理したりすることは面倒な作業です。-z rescan-now オプションを指定すると、この処理は簡単になります。-z rescan-now オプションは、コマンド行でこのオプションが検出されると、すぐにリンカーで処理されます。このオプションより前にコマンド行で処理されたすべてのアーカイブは、すぐに再処理されます。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 .... -lA -lB -lC -z rescan-now
```

また、-z rescan-start および -z rescan-end のオプションを使用すると、相互に依存するアーカイブを 1 つのアーカイブグループにまとめることができます。結びの区切り文字がコマンド行に書かれていると、これらのグループはリンカーですぐに再処理されます。グループ内で見つかったアーカイブは再処理され、シンボル参照を解決する追加アーカイブメンバーを検出しようとします。このアーカイブ再走査は、渡されたアーカイブグループに新しいメンバーが検出されなくなるまで続けられます。アーカイブグループを使用すると、前の例は次のように書くことができます。

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

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た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、/lib、/usr/lib の順に libfoo を検索します。このコマンドは、path1、path2、/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取ります。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 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:/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 オプションのあとで解釈されます。次の例では、有効な検索パスは path1:path2.... pathn:dir1:dir2:/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。97 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。99 ページの「デフォルトの検索パスの構成」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すると、柔軟に実行パスを記録できます。268 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していました。LD_RUN_PATH と -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 .pre_initarray、.init_array 、および .fini_array 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 セクションを提供します。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、112 ページの「初期設定および終了ルーチン」を参照してください。

初期設定および終了関数の登録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、foo() のアドレスが .init_array 要素に配置され、bar() のアドレスが .fini_array 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、113 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は、データの初期設定を別途行うようにしてください。

シンボルの処理

シンボルは、局所と大域に分類できます。47 ページの「シンボルの可視性」を参照してください。

局所シンボルは、入力ファイルの処理中に入力再配置可能オブジェクトファイルから構築中の出力オブジェクトに検査なしでコピーされます。

すべての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と外部依存関係から渡される大域シンボルは、シンボル解決と呼ばれるプロセスで分析および結合されます。リンカーは、各シンボルを検出された順に内部シンボルテーブルに配置します。同じ名前のシンボルが以前のオブジェクトによって提供され、シンボルテーブル内にすでに存在する場合は、シンボル解決プロセスによって 2 つのシンボルのどちらを保持するかが決定されます。このプロセスの副作用として、リンカーは外部オブジェクトの依存関係への参照を確立する方法を決定します。

入力ファイルの処理が正常終了すると、リンカーはシンボル可視性の調整を適用し、未解決のシンボル参照が残っているかどうかを判定します。シンボル解決の致命的エラーが発生した場合や、未解決のシンボル参照が残っている場合は、リンク編集が終了します。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。

以降のセクションでは、シンボルの可視性、シンボル解決、および未定義シンボルの処理について詳しく説明します。

シンボルの可視性

シンボルは、局所と大域に分類できます。局所シンボルは、シンボル定義が含まれるオブジェクト以外のオブジェクトから参照できません。デフォルトでは、局所シ

ンボルは入力再配置可能オブジェクトファイルから構築中の出力オブジェクトにコピーされます。代わりに、局所シンボルを出力オブジェクトから削除できます。63 ページの「シンボル削除」を参照してください。

大域シンボルは、シンボル定義が含まれるオブジェクト以外のオブジェクトからも参照できます。大域シンボルは、収集と解決のあとで、出力オブジェクト内に作成されるシンボルテーブルに追加されます。すべての大域シンボルがまとめて処理および解決されますが、それらの最終的な可視性は調整できます。大域シンボルには、追加の可視性属性を定義できます。表 12–20 を参照してください。さらに、`mapfile` シンボル指令を使用して、リンク編集中にシンボルの可視性を割り当てることもできます。表 8–8 を参照してください。これらの可視性属性 (および指令) により、出力オブジェクトへの書き込み時に可視性が調整された大域シンボルを生成できます。

再配置可能オブジェクトを作成すると、すべての可視性属性および指令が出力オブジェクトに記録されます。ただし、これらの属性によって暗黙的に定義された可視性の変更は適用されません。代わりに、これらのオブジェクトを入力として読み取る動的オブジェクトの次のリンク編集まで、可視性の処理が延期されます。特殊なケースでは、`-B reduce` オプションを使用して、可視性属性および指令をただちに強制的に解釈できます。

動的実行可能ファイル (または共有オブジェクト) を作成すると、シンボルがシンボルテーブルに書き込まれる前に、シンボル可視性の属性および指令が適用されます。可視性属性により、シンボルが大域のままであり、シンボル縮小の手法による影響を受けないことを保証できます。可視性の属性および指令によって、局所に降格される大域シンボルを生成することもできます。この後者の手法がもっともよく使用されるのは、オブジェクトのエクスポートされたインタフェースを明示的に定義する場合です。60 ページの「シンボル範囲の縮小」を参照してください。

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、371 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、ストレージ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、またはストレージ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ストレージ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
$ elfdump -s main.o

Symbol Table Section:  .symtab
     index    value      size      type bind oth ver shndx          name
     ....
       [7]  0x00000000 0x00000000  FUNC GLOB  D    0 UNDEF          u_foo
       [8]  0x00000010 0x00000040  FUNC GLOB  D    0 .text          d_foo
       [9]  0x00000004 0x00000004  OBJT GLOB  D    0 COMMON         t_bar
      [10]  0x00000000 0x00000004  NOTY GLOB  D    0 UNDEF          u_bar
      [11]  0x00000000 0x00000004  OBJT GLOB  D    0 .data          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の処理中は、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。ウィークシンボル定義は同じ名前の大域定義によって暗黙のうちにオーバーライドされます。

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オーバーライド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。

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 };

$ ld -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
```

シンボルの配置要件が異なる場合に、同様の診断が生成されます。この両方の場合、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されません。

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);
}

$ ld -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 は、シンボル bar の定義が競合しています。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、暗黙的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、前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 への明示的参照によって構築されます。libbar.so は libfoo.so への依存関係があります。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 では libfoo.so によって提供されるインタフェースへの特定の参照を作成したため、prog は実際に libfoo.so への依存性を持ちます。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。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」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

ウィークシンボルは歴史的に、割り込みを回避したり、オプション機能をテストしたりするために使用されてきました。ただし、近年のプログラミング環境ではウィークシンボルは脆弱で信頼性が低いことが経験によって示されており、使用は推奨されません。

ウィークシンボル別名はシステム共有オブジェクト内で頻繁に使用されてきました。その意図は、代替のインタフェース名を提供することであり、典型的には「_」文字が前に付いたシンボル名です。このエイリアス名は、アプリケーションがシンボル名の独自の実装をエクスポートすることによる割り込みの問題を回避するために、他のシステム共有オブジェクトから参照できます。現実的には、この手法は複雑すぎることが証明され、整合性を持たずに使用されていました。最近の Oracle Solaris バージョンでは、システムオブジェクト間の明示的な結合を直接結合によって確立します。第 6 章「直接結合」を参照してください。

ウィークシンボル参照は、実行時でのインタフェースの存在をテストするためにしばしば使用されていました。この手法は、構築環境および実行環境に制限を設け、コンパイラ最適化によって回避できます。`dlsym(3C)` を `RTLD_DEFAULT` または `RTLD_PROBE` ハンドルと一緒に使用することで、シンボルの存在をテストするための一貫性のある堅牢な手段が提供されます。128 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。一時的シンボルはこの規則の例外であり、シンボルが完全に解決されるまで完全には定義されません。出力ファイル内の一時的シンボルの順番は、追加順にならない場合があります。

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 One_array[0x10];
char Two_array[0x20];
char Three_array[0x30];

$ cc -o libfoo.so -G -Kpic foo.c
$ elfdump -sN.dynsym libfoo.so | grep array | sort -k 2,2
      [11]  0x00010614 0x00000020  OBJT GLOB  D    0 .bss          Two_array
       [3]  0x00010634 0x00000030  OBJT GLOB  D    0 .bss          Three_array
       [4]  0x00010664 0x00000010  OBJT GLOB  D    0 .bss          One_array
```

シンボルをアドレス順にソートすると、その出力順序はソース内で定義された順序と異なることがわかります。反対に、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と、入力ファイル内のこれらのシンボルの相対順序は、確実に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 libfoo.so -G -Kpic foo.c
$ elfdump -sN.dynsym libfoo.so | grep array | sort -k 2,2
       [4]  0x00010614 0x00000010  OBJT GLOB  D    0 .data         One_array
      [11]  0x00010624 0x00000020  OBJT GLOB  D    0 .data         Two_array
       [3]  0x00010644 0x00000030  OBJT GLOB  D    0 .data         Three_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指定できます。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。可視性や型などのシンボルの属性を指定できます。使用可能なオプションの詳細な説明については、225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」を参照してください。

-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、37 ページの「アーカイブ処理」を参照してください。

たとえば、動的実行可能ファイルを、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 内に含まれる再配置可能オブジェクト foo.o からシンボル定義 foo を取得し、さらに lib2.a 内に含まれる再配置可能オブジェクト bar.o からシンボル定義 bar を取得します。

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 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、37 ページの「アーカイブ処理」を参照してください。

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
#include <stdio.h>
void foo()
{
        (void) printf("foo: called from lib.a\n");
}
$ cat bar.c
#include <stdio.h>
void bar()
```

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

void main()
{
        foo();
        bar();
}
$ cc -c foo.c bar.c main.c
$ ar -rc lib.a foo.o bar.o main.o
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        local:
                foo;
                bar;
        global:
                main;
};
$ cc -o prog -M mapfile lib.a
$ prog
foo: called from lib.a
bar: called from lib.a
$ elfdump -sN.symtab prog | egrep 'main$|foo$|bar$'
      [29]  0x00010f30 0x00000024  FUNC LOCL  H    0 .text          bar
      [30]  0x00010ef8 0x00000024  FUNC LOCL  H    0 .text          foo
      [55]  0x00010f68 0x00000024  FUNC GLOB  D    0 .text          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、60 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
#include <stdio.h>
extern  int     foo();
extern  int     bar;

void main()
{
        (void) printf("&foo = 0x%p\n", &foo);
        (void) printf("&bar = 0x%p\n", &bar);
}
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=FUNCTION; VALUE=0x400 };
                bar     { TYPE=DATA;     VALUE=0x800 };
};
```

```
$ cc -o prog -M mapfile main.c
$ prog
&foo = 0x400
&bar = 0x800
$ elfdump -sN.symtab prog | egrep 'foo$|bar$'
      [45]  0x00000800 0x00000000  OBJT GLOB  D    0 ABS            bar
      [69]  0x00000400 0x00000000  FUNC GLOB  D    0 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、データストレージの要素に関連しています。mapfile 定義は、このデータストレージ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、データストレージが作成されます。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。59 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、ファイル内のストレージを占有しませんが、実行時に割り当てるストレージの定義は行います。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ストレージ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、新しいストレージ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
#include <stdio.h>
extern  int     foo;
int             bar[0x10];

void main()
{
        (void) printf("&foo = 0x%p\n", &foo);
        (void) printf("&bar = 0x%p\n", &bar);
}
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=COMMON; VALUE=0x4;   SIZE=0x200 };
                bar     { TYPE=COMMON; VALUE=0x102; SIZE=0x40 };
};
```

```
$ cc -o prog -M mapfile main.c
ld: warning: symbol 'bar' has differing alignments:
    (file mapfile value=0x102; file main.o value=0x4);
    largest value applied
$ prog
&foo = 0x21264
&bar = 0x21224
$ elfdump -sN.symtab prog | egrep 'foo$|bar$'
      [45]  0x00021224 0x00000040  OBJT GLOB  D    0 .bss           bar
      [69]  0x00021264 0x00000200  OBJT GLOB  D    0 .bss           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。196 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ストレージ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。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」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。145 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
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                foo     { TYPE=FUNCTION;       FILTER=filtee.so.1 };
                bar     { TYPE=DATA; SIZE=0x4; FILTER=filtee.so.1 };
        local:
                *;
};
$ cc -o filter.so.1 -G -Kpic -h filter.so.1 -M mapfile -R.
$ elfdump -sN.dynsym filter.so.1 | egrep 'foo|bar'
       [1]  0x000105f8 0x00000004  OBJT GLOB  D    1 .data          bar
       [7]  0x00000000 0x00000000  FUNC GLOB  D    1 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 が含まれています。ファイル bar.c には、共有オブジェクトのベースとなる実装を提供するシンボル bar と str が含まれています。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
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
      [41]  0x00000560 0x00000018  FUNC GLOB  D    0 .text          bar
      [44]  0x00000520 0x0000002c  FUNC GLOB  D    0 .text          foo
      [45]  0x000106b8 0x00000004  OBJT GLOB  D    0 .data          str
```

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実装に結合されます。

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、49 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
$mapfile_version 2
SYMBOL_SCOPE {
        local:
                bar;
                str;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
      [24]  0x00000548 0x00000018  FUNC LOCL  H    0 .text          bar
      [25]  0x000106a0 0x00000004  OBJT LOCL  H    0 .data          str
      [45]  0x00000508 0x0000002c  FUNC GLOB  D    0 .text          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、195 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 を定義できます。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
        local:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
      [26]  0x00000570 0x00000018  FUNC LOCL  H    0 .text          bar
      [27]  0x000106d8 0x00000004  OBJT LOCL  H    0 .data          str
      [50]  0x00000530 0x0000002c  FUNC GLOB  D    0 .text          foo
```

この例では、mapfile 指令の一部としてバージョン名 libfoo.so.1.1 も定義しています。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 9 章「インタフェースおよびバージョン管理」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
$mapfile_version 2
SYMBOL_VERSION 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、シンボル縮小がない中間再配置可能オブジェクトが作成されます。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
        local:
                *;
};
$ ld -o libfoo.o -M mapfile -r foo.o bar.o
```

```
$ elfdump -s libfoo.o | egrep 'foo$|bar$|str$'
     [28]  0x00000050 0x00000018  FUNC GLOB  H    0 .text          bar
     [29]  0x00000010 0x0000002c  FUNC GLOB  D    2 .text          foo
     [30]  0x00000000 0x00000004  OBJT GLOB  H    0 .data          str
```

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。

```
$ ld -o libfoo.so.1 -G libfoo.o
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
     [24]  0x00000508 0x00000018  FUNC LOCL  H    0 .text          bar
     [25]  0x00010644 0x00000004  OBJT LOCL  H    0 .data          str
     [42]  0x000004c8 0x0000002c  FUNC GLOB  D    0 .text          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
$ elfdump -sN.symtab libfoo.o | egrep 'foo$|bar$|str$'
     [20]  0x00000050 0x00000018  FUNC LOCL  H    0 .text          bar
     [21]  0x00000000 0x00000004  OBJT LOCL  H    0 .data          str
     [30]  0x00000010 0x0000002c  FUNC GLOB  D    2 .text          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。このテーブル全体は、リンカーの -z strip-class オプションを使用して、またはリンク編集後に strip(1) を使用してオブジェクトから取り除くことができます。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。

シンボルの削除は、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
$mapfile_version 2
SYMBOL_VERSION ISV_1.1 {
        global:
                foo;
```

```
        local:
                str;
        eliminate:
                *;
};
$ cc -o libfoo.so.1 -M mapfile -G foo.c bar.c
$ elfdump -sN.symtab libfoo.so.1 | egrep 'foo$|bar$|str$'
      [26]  0x00010690 0x00000004  OBJT LOCL  H    0 .data          str
      [44]  0x000004e8 0x0000002c  FUNC GLOB  D    0 .text          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。第 6 章「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ます。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。

動的実行可能ファイルまたは共有オブジェクトを作成すると、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) 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` テーブルと、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 `-z strip-class` 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ができます。

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。186 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。186 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とき、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。

機能要件の特定

機能によって、コードを実行するために必要なシステム属性が決まります。使用できる機能は、優先順に次のとおりです。

- プラットフォーム機能 - 特定のプラットフォームを名前で指定します。
- マシン機能 - 特定マシンのハードウェアを名前で指定します。
- ハードウェア機能 - 命令セットの拡張やほかのハードウェアの詳細情報を、機能フラグを使用して指定します。
- ソフトウェア機能 - ソフトウェア環境の属性を、機能フラグを使用して反映します。

これらの各機能は個々に定義したり、組み合わせて機能グループを作成したりできます。

ある機能が利用可能な場合にだけ実行できるコードは、関連する ELF オブジェクト内の機能セクションで、これらの要件を指定する必要があります。オブジェクト内に機能要件を記録すると、関連コードの実行を試みる前に、システムがそのオブジェクトを検証できます。これらの要件によって、オブジェクトファミリの中からもっとも適したオブジェクトをシステムが選択できるフレームワークを規定することもできます。ファミリは同じオブジェクトの派生から構成されます。ただし、各派生は異なる機能を要求します。

動的オブジェクトに加えて、オブジェクト内の各関数または初期化済みデータ項目を、機能要件に関連付けることができます。理想的には、機能要件はコンパイラによって作成される再配置可能オブジェクトに記録され、コンパイル時に指定されたオプションまたは最適化を反映します。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。347 ページの「機能セクション」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` を使用して定義された機能は、入力再配置可能オブジェクト内で指定された機能を強化したり、オーバーライドしたりすることができます。通常 `mapfile` は、必要な機能情報を生成しないコンパイラを補強するために使用されます。

システム機能は、実行中のシステムを記述する機能です。プラットフォーム名とマシンハードウェア名は、`uname(1)` でそれぞれ `-i` オプションと `-m` オプションを使用すると表示できます。システムのハードウェア機能は、`isainfo(1)` で `-v` オプションを使用すると表示できます。実行時、オブジェクトの機能要件はシステムの機能と比較され、オブジェクトを読み込むことができるかどうか、またはオブジェクト内のシンボルを利用できるかどうかを判断します。

オブジェクト機能は、オブジェクトに関連する機能です。これらの機能はオブジェクト全体の要件を定義して、実行時にオブジェクトを読み込むことができるかどうかを制御します。あるオブジェクトが、システムで満たすことができない機能を要求している場合、そのオブジェクトは実行時に読み込めません。機能を使用すると任意のオブジェクトから複数のインスタンスを作成でき、各インスタンスはオブジェクト要件に一致するシステムに最適化されます。実行時リンカーは、オブジェクトの機能要件をシステムが提供する機能と比較することで、このようなオブジェクトインスタンスファミリの中から最適なインスタンスを透過的に選択します。

シンボル機能は、オブジェクト内の各関数または初期化されたデータ項目に関連する機能です。これらの機能は、オブジェクト内の 1 つまたは複数のシンボルの要件を定義し、実行時にそのシンボルを使用できるかどうかを制御します。シンボル機能によって、1 つの関数の複数のインスタンスが 1 つのオブジェクト内に存在できます。関数の各インスタンスは、別々の機能を持つシステムに対して最適化できます。また、1 つの初期化されたデータ項目の複数のインスタンスも 1 つのオブジェクト内に存在できます。データの各インスタンスはシステム固有のデータを定義できます。シンボルのインスタンスが、システムによって満たすことができない機能を要求している場合、そのシンボルインスタンスは実行時に使用できません。代わりに、同じシンボル名の代替インスタンスを使用する必要があります。シンボル機能では、単一のオブジェクトをさまざまな機能のシステムで使用できるようにオブジェクトを構築できます。機能ファミリは、機能に対応できるシステム向けの最適化されたインスタンスと、機能が比較的低いほかのシステム向けのより汎用的なインスタンスを提供できます。初期化されたデータ項目のファミリはシステム固有データを提供できます。実行時リンカーは、シンボルの機能要件をシステムが提供

する機能と比較することで、このようなシンボルインスタンスファミリの中から最適なインスタンスを透過的に選択します。

オブジェクトとシンボルの機能は、現在動作中のシステムに対して最適なオブジェクト、およびオブジェクト内の最適なシンボルを選択できます。オブジェクトとシンボルの機能はオプションであり、互いに独立しています。ただし、シンボル機能を定義しているオブジェクトがオブジェクト機能を定義してもかまいません。この場合、機能シンボルファミリには、オブジェクト機能を満たすシンボルの 1 つのインスタンスが伴われることになります。オブジェクトの機能が存在しない場合、機能シンボルファミリには、機能を要求しないシンボルの 1 つのインスタンスが伴われることになります。指定されたシステムに適合できる機能インスタンスがない場合、このシンボルインスタンスによってデフォルト実装が提供されます。

次の x86 の例は、`foo.o` のオブジェクト機能を示します。これらの機能はオブジェクト全体に適用されます。この例には、シンボル機能がありません。

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x840  [ SSE  MMX ]
```

次の x86 の例は、`bar.o` のシンボル機能を示します。これらの機能は個々の関数 `foo` および `bar` に適用されます。各シンボルのインスタンスは 2 つ存在し、それぞれのインスタンスは別々の機能セットに割り当てられています。この例には、オブジェクト機能がありません。

```
$ elfdump -H bar.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
      [25]  0x00000000 0x00000021  FUNC LOCL  D    0 .text    foo%mmx
      [26]  0x00000024 0x0000001e  FUNC LOCL  D    0 .text    bar%mmx

 Symbol Capabilities:
     index  tag               value
       [3]  CA_SUNW_HW_1      0x800  [ SSE ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
      [33]  0x00000044 0x00000021  FUNC LOCL  D    0 .text    foo%sse
      [34]  0x00000068 0x0000001e  FUNC LOCL  D    0 .text    bar%sse
```

注 – この例では、機能シンボルは、機能識別子を汎用のシンボル名に追加する命名規則に従っています。リンカーはこの規則に従ってオブジェクト機能をシンボル機能に変換できます。あとで説明する81 ページの「オブジェクト機能のシンボル機能への変換」を参照してください。

機能定義には、オブジェクトの要件、またはオブジェクト内の各シンボルの要件を特定できる多くの組み合わせが用意されています。ハードウェア機能には、非常に高い柔軟性が備わっています。ハードウェア機能でハードウェア要件を定義するときは、特定のマシンハードウェア名またはプラットフォーム名を指定しません。しかし、ベースとなるシステムの属性の中には、マシンのハードウェア名またはプラットフォーム名からしか判断できないものもあります。機能名を指定すると、特定のシステム機能向けにコーディングできますが、指定したオブジェクトの利用が制限される可能性があります。あるオブジェクトに対して新しいマシンハードウェア名またはプラットフォーム名が利用できるようになったときに、その新しい機能名を特定するために、オブジェクトの再作成が必要になります。

次のセクションでは、機能の定義方法とリンカーによる使用方法について説明します。

プラットフォーム機能の特定

オブジェクトのプラットフォーム機能では、オブジェクトまたはオブジェクト内の特定のシンボルが実行できるシステムのプラットフォーム名を指定します。複数のプラットフォーム機能を定義できます。これはもっとも具体的であり、ほかの機能タイプに優先します。

システムのプラットフォーム名は、ユーティリティー `uname(1)` で `-i` オプションを使用すると表示できます。

プラットフォーム機能要件は次の `mapfile` 構文を使用すると定義できます。

```
$mapfile_version 2
CAPABILITY {
        PLATFORM  = platform_name...;
        PLATFORM += platform_name...;
        PLATFORM -= platform_name...;
};
```

`PLATFORM` 属性は 1 つまたは複数のプラットフォーム名で修飾されます。「`+=`」形式の代入を使用すると、入力オブジェクトで指定されたプラットフォーム機能に追加できます。「`=`」形式はこれらをオーバーライドします。「`-=`」形式の代入を使用すると、出力オブジェクトからプラットフォーム機能を削除できます。次の SPARC の例では、オブジェクト `foo.so.1` を `SUNW,SPARC-Enterprise` プラットフォームに固有であることを指定します。

```
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        PLATFORM = 'SUNW,SPARC-Enterprise';
};
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_PLAT     SUNW,SPARC-Enterprise
```

再配置可能オブジェクトはプラットフォーム機能を定義できます。これらの機能がまとめられて、作成中のオブジェクトの最終的な機能要件が定義されます。

オブジェクトのプラットフォーム機能は、「=」形式の代入を使用すると mapfile から明示的に制御でき、入力の再配置可能オブジェクトから指定される可能性のあるプラットフォーム機能をオーバーライドできます。「=」形式の代入を使用して空の PLATFORM 属性を指定すると、作成中のオブジェクトからプラットフォームのすべての機能要件が削除されます。

動的オブジェクトで定義されたプラットフォーム機能要件は、システムのプラットフォーム名に照らして、実行時リンカーによって検証されます。オブジェクトが使用されるのは、オブジェクトに記録されたプラットフォーム名の 1 つがシステムのプラットフォーム名に一致した場合だけです。

コードを特定のプラットフォーム用に限定することは、場合によって有効となります。しかし、ハードウェア機能ファミリを作成する方が柔軟性を高めることができるため、この方法を推奨します。ハードウェア機能ファミリは、コードを幅広いシステムで実行できるように最適化できます。

マシン機能の指定

オブジェクトのマシン機能では、オブジェクトまたはオブジェクト内の特定のシンボルが実行できるシステムのマシンハードウェア名を指定します。複数のマシン機能を定義できます。この指定はプラットフォーム機能定義に比べて優先度が低いですが、ほかの機能タイプより優先します。

システムのマシンハードウェア名は、ユーティリティー uname(1) で -m オプションを使用すると表示できます。

マシン機能要件は次の mapfile 構文を使用すると定義できます。

```
$mapfile_version 2
CAPABILITY {
        MACHINE  = machine_name...;
        MACHINE += machine_name...;
        MACHINE -= machine_name...;
};
```

MACHINE 属性は 1 つまたは複数のマシンハードウェア名で修飾されます。「+=」形式の代入を使用すると、入力オブジェクトで指定されたマシン機能に追加できます。「=」形式はこれらをオーバーライドします。「-=」形式の代入を使用すると、出力オブジェクトからマシン機能を削除できます。次の SPARC の例では、オブジェクト foo.so.1 を sun4u マシンハードウェア名に固有であることを指定します。

```
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        MACHINE = sun4u;
};
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_MACH     sun4u
```

再配置可能オブジェクトはマシン機能を定義できます。これらの機能がまとめられて、作成中のオブジェクトの最終的な機能要件が定義されます。

オブジェクトのマシン機能は、「=」形式の代入を使用すると mapfile から明示的に制御でき、入力の再配置可能オブジェクトから指定される可能性のあるマシン機能をオーバーライドできます。「=」形式の代入を使用して空の MACHINE 属性を指定すると、作成中のオブジェクトからマシンのすべての機能要件が削除されます。

動的オブジェクトで定義されたマシン機能要件は、システムのマシンハードウェア名に照らして、実行時リンカーによって検証されます。オブジェクトが使用されるのは、オブジェクトに記録されたマシン名の 1 つがシステムのマシン名に一致した場合だけです。

コードを特定のマシン用に限定することは、場合によって有効となります。しかし、ハードウェア機能ファミリを作成する方が柔軟性を高めることができるため、この方法を推奨します。ハードウェア機能ファミリは、コードを幅広いシステムで実行できるように最適化できます。

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、オブジェクトまたは特定のシンボルを正しく実行するために必要なシステ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 の機能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。

```
$mapfile_version 2
CAPABILITY {
        HW  = hwcap_flag... ;
```

```
                HW += hwcap_flag...;
                HW -= hwcap_flag...;
        };
```

CAPABILITY 指令に対する HW 属性は、ハードウェア機能のシンボル表現である 1 つまたは複数のトークンで修飾されます。「+=」形式の代入を使用すると、入力オブジェクトで指定されたハードウェア機能に追加できます。「=」形式はこれらをオーバーライドします。「-=」形式の代入を使用すると、出力オブジェクトからハードウェア機能を削除できます。

SPARC システムでは、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 システムでは、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
$mapfile_version 2
CAPABILITY {
        HW += SSE MMX;
};
$ cc -o foo.so.1 -G -K pic -Mmapfile foo.c -lc
$ elfdump -H foo.so.1

Capabilities Section:  .SUNW_cap

 Object Capabilities:
    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。

オブジェクトのハードウェア機能要件は、「=」形式の代入を使用すると mapfile から明示的に制御でき、入力の再配置可能オブジェクトから指定される可能性のあるハードウェア機能をオーバーライドできます。「=」形式の代入を使用して空の HW 属性を指定すると、作成中のオブジェクトからハードウェアのすべての機能要件が削除されます。

次の例では、入力の再配置可能オブジェクト foo.o で定義されたハードウェア機能のデータが出力ファイル bar.o に含まれないように隠されています。

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap
```

```
 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x840  [ SSE  MMX ]
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        HW = ;
};
$ ld -o bar.o -r -Mmapfile foo.o
$ elfdump -H bar.o
$
```

動的オブジェクトが定義したハードウェア機能要件は、システムが提供するハードウェア機能に照らして、実行時リンカーによって検証されます。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 ]
        ....
```

別々のハードウェア機能を利用する動的オブジェクトの複数のバリエーションは、フィルタを使用することによって、柔軟な実行時環境を提供できます。263 ページの「機能固有の共有オブジェクト」を参照してください。

ハードウェア機能は、1 つのオブジェクト内の個々の関数の機能を指定するためにも使用できます。この場合、実行時リンカーは現在のシステム機能に基づいて使用する最適な関数を選択できます。75 ページの「シンボル機能関数ファミリの作成」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。ソフトウェア機能はプロセスの実行にも影響を与えることができます。現在のところ、オブジェクトによるフレームポインタの使用、およびプロセスアドレス空間の制限に関連したソフトウェア機能のみが存在します。

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。

64 ビットのオブジェクトは、実行時に 32 ビットのアドレス空間内で実行しなければならないことを指定できます。

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。

```
$mapfile_version 2
CAPABILITY {
        SF  = sfcap_flags...;
        SF += sfcap_flags...;
        SF -= sfcap_flags...;
};
```

CAPABILITY 指令に対する SF 属性は、トークン FPKNWN および FPUSED のいずれにも割り当てることができます。

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。

オブジェクトのソフトウェア機能要件は、「=」形式の代入を使用すると mapfile から明示的に制御でき、入力の再配置可能オブジェクトから指定される可能性のあるソフトウェア機能をオーバーライドできます。「=」形式の代入を使用して空の HW 属性を指定すると、作成中のオブジェクトからソフトウェアのすべての機能要件が削除されます。

次の例では、入力の再配置可能オブジェクト foo.o で定義されたソフトウェア機能のデータが出力ファイル bar.o に含まれないように隠されています。

```
$ elfdump -H foo.o

Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_SF_1     0x3  [ SF1_SUNW_FPKNWN  SF1_SUNW_FPUSED ]
$ cat mapfile
$mapfile_version 2
CAPABILITY {
        SF = ;
};
$ ld -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。

シンボル機能関数ファミリの作成

開発者は、関数の複数のインスタンス (それぞれは特定の機能セット向けに最適化) を 1 つのオブジェクト内に用意したいと考えることがよくあります。インスタンスの選択と使用が消費者に対して透過的であることが望まれます。外部インタフェースとして、汎用的なフロントエンドの関数を作成できます。この汎用インスタンスと最適化されたインスタンスを 1 つのオブジェクトに結合できます。タスクを処理するために汎用インスタンスは getisax(2) を使用してシステムの機能を判断し、最適化された適切な関数インスタンスを呼び出すことができます。このモデルは適切に機能しますが、汎用性に欠け、実行時のオーバーヘッドが発生します。

シンボル機能はこのようなオブジェクトを作成するために別の方法を提供します。この方法はより簡単で効率が良く、さらにフロントエンドのコードを書く必要がありません。1 つの関数の複数のインスタンスを作成できます。また、さまざまな機能に関連付けることができます。これらのインスタンスと、いずれのシステムにも適した関数のデフォルトのインスタンスは、1 つの動的オブジェクトに結合できます。このファミリのシンボルの中でもっとも適したメンバーが、シンボルの機能情報を使用して、実行時リンカーによって選択されます。

次の例では、x86 オブジェクト foobar.mmx.o および foobar.sse.o が同じ関数 foo() と bar() を保持しており、それぞれ MMX と SSE の命令を使用するようにコンパイルされています。

```
$ elfdump -H foobar.mmx.o

Capabilities Section:  .SUNW_cap
```

```
 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID        mmx
       [2]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
      [10]  0x00000000 0x00000021  FUNC LOCL  D    0 .text    foo%mmx
      [16]  0x00000024 0x0000001e  FUNC LOCL  D    0 .text    bar%mmx

$ elfdump -H foobar.sse.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID        sse
       [2]  CA_SUNW_HW_1      0x800  [ SSE ]

  Capabilities symbols:
     index    value      size      type bind oth ver shndx    name
      [16]  0x00000000 0x0000002f  FUNC LOCL  D    0 .text    foo%sse
      [18]  0x00000048 0x00000030  FUNC LOCL  D    0 .text    bar%sse
```

これらの各オブジェクトは機能関数 `foo%*()` および `bar%*()` を指定する局所シンボルを含んでいます。また、各オブジェクトは関数 `foo()` および `bar()` への大域参照も定義します。`foo()` または `bar()` への内部参照はすべて、これらの大域参照を介して、外部インタフェースのように再配置されます。

これら 2 つのオブジェクトは、`foo()` および `bar()` のデフォルトのインスタンスと結合できるようになりました。これらのデフォルトインスタンスは大域参照を満たし、どのオブジェクト機能とも互換性を持つ実装を提供します。これらのデフォルトインスタンスは各機能ファミリの先頭になります。オブジェクトの機能が存在しない場合、このデフォルトインスタンスも機能を要求してはいけません。実質的に、`foo()` と `bar()` の 3 つのインスタンスが存在し、大域インスタンスはデフォルトを提供し、局所インスタンスは関連機能が使用できる場合に、実行時に使用される実装を提供します。

```
$ cc -o libfoobar.so.1 -G foobar.o foobar.sse.o foobar.mmx.o
$ elfdump -sN.dynsym libfoobar.so.1 | egrep "foo|bar"
       [2]  0x00000700 0x00000021  FUNC LOCL  D    0 .text    foo%mmx
       [4]  0x00000750 0x0000002f  FUNC LOCL  D    0 .text    foo%sse
       [8]  0x00000784 0x0000001e  FUNC LOCL  D    0 .text    bar%mmx
       [9]  0x000007b0 0x00000030  FUNC LOCL  D    0 .text    bar%sse
      [15]  0x000007a0 0x00000014  FUNC GLOB  D    1 .text    foo
      [17]  0x000007c0 0x00000014  FUNC GLOB  D    1 .text    bar
```

動的オブジェクトの機能情報には機能シンボルが表示され、利用できる機能ファミリがわかります。

```
$ elfdump -H libfoobar.so.1

Capabilities Section:  .SUNW_cap
```

```
 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID        mmx
       [2]  CA_SUNW_HW_1      0x40  [ MMX ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
       [2]  0x00000700 0x00000021  FUNC LOCL  D    0 .text    foo%mmx
       [8]  0x00000784 0x0000001e  FUNC LOCL  D    0 .text    bar%mmx

 Symbol Capabilities:
     index  tag               value
       [4]  CA_SUNW_ID        sse
       [5]  CA_SUNW_HW_1      0x800  [ SSE ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
       [4]  0x00000750 0x0000002f  FUNC LOCL  D    0 .text    foo%sse
       [9]  0x000007b0 0x00000030  FUNC LOCL  D    0 .text    bar%sse

Capabilities Chain Section:  .SUNW_capchain

 Capabilities family: foo
  chainndx  symndx      name
         1  [15]        foo
         2  [2]         foo%mmx
         3  [4]         foo%sse

 Capabilities family: bar
  chainndx  symndx      name
         5  [17]        bar
         6  [8]         bar%mmx
         7  [9]         bar%sse
```

実行時、`foo()` と `bar()` へのすべての参照は、まず大域シンボルに結合されます。しかし、実行時リンカーはこれらの関数が機能ファミリの先頭のインスタンスであることを認識しています。実行時リンカーは各ファミリメンバーを検査して、より適した機能関数が使用できるかどうかを判断します。この処理には 1 回限りのコストがかかり、関数の最初の呼び出し時に発生します。`foo()` および `bar()` への後続の呼び出しは、最初の呼び出しで選択された関数のインスタンスに直接結合されます。この関数の選択は実行時リンカーのデバッグ機能を使用すると確認できます。

次の例では、ベースとなるシステムが `MMX` または `SSE` をサポートしていません。`foo()` の先頭のインスタンスには特別な機能のサポートは必要ないため、どの再配置参照も満たします。

```
$ LD_DEBUG=symbols main
....
debug: symbol=foo;  lookup in file=./libfoo.so.1  [ ELF ]
debug:   symbol=foo[15]:  capability family default
debug:   symbol=foo%mmx[2]:  capability specific (CA_SUNW_HW_1):  [ 0x40  [ MMX ] ]
debug:   symbol=foo%mmx[2]:  capability rejected
debug:   symbol=foo%sse[4]:  capability specific (CA_SUNW_HW_1):  [ 0x800  [ SSE ] ]
debug:   symbol=foo%sse[4]:  capability rejected
debug:   symbol=foo[15]:  used
```

次の例では、MMX は使用できますが、SSE は使用できません。MMX が使用できる foo() のインスタンスは、どの再配置参照も解決します。

```
$ LD_DEBUG=symbols main
....
debug: symbol=foo;  lookup in file=./libfoo.so.1  [ ELF ]
debug:   symbol=foo[15]:  capability family default
debug:   symbol=foo[2]:  capability specific (CA_SUNW_HW_1):  [ 0x40  [ MMX ] ]
debug:   symbol=foo[2]:  capability candidate
debug:   symbol=foo[4]:  capability specific (CA_SUNW_HW_1):  [ 0x800  [ SSE ] ]
debug:   symbol=foo[4]:  capability rejected
debug:   symbol=foo[2]:  used
```

複数の機能インスタンスが同じシステムで実行できる場合、一連の優先規則を利用して 1 つのインスタンスを選択します。

- プラットフォーム名を定義している機能グループは、プラットフォーム名を定義していないグループに優先します。
- マシンのハードウェア名を定義している機能グループは、マシンのハードウェア名を定義していないグループに優先します。
- ハードウェア機能の値は、大きい値が小さい値より優先します。

機能関数インスタンスのファミリはプロシージャーのリンクテーブルエントリからアクセスできる必要があります。425 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。このプロシージャーリンクの参照には、実行時リンカーが関数を解決する必要があります。このプロセス中、実行時リンカーは関連のシンボル機能情報を処理して、使用できる関数インスタンスファミリから最適な関数を選択できます。

シンボル機能が使用されないときに、リンカーがプロシージャーのリンクテーブルエントリを必要としないでコードへの参照を解決できる場合があります。たとえば、動的実行可能ファイル内では、実行可能ファイル内に存在する関数への参照が、リンク編集時に内部的に結合できます。共有オブジェクト内に隠されて保護されている関数も、リンク編集時に内部的に結合できます。この場合、一般的に、実行時リンカーはこれらの関数への参照の解決に関与する必要はありません。

しかし、シンボル機能が使用された場合、関数はプロシージャーのリンクテーブルエントリから解決される必要があります。このエントリが必要なのは、実行時リンカーが読み取り専用のテキストセグメントを維持しながら、適切な関数を選択することに関与するためです。このメカニズムでは、機能関数へのすべての呼び出しが、プロシージャーのリンクテーブルエントリを介した間接参照になります。この間接参照は、シンボル機能が使用されない場合は必要ない可能性があります。このため、機能関数を呼び出すコストと、機能関数を使用して得られるデフォルトに対する性能の改善との間に小さいトレードオフがあります。

注 – 機能関数はプロシージャーのリンクテーブルエントリを介してアクセスする必要がありますが、この関数は隠された、または保護されたものとして定義できます。実行時リンカーはこれらの可視性に従って、関数への結合を制限します。この動作により、シンボル機能が関数と関連付けられていないときに作成される結合と同じ結合になります。隠された関数は外部オブジェクトから結合できません。保護された関数へのオブジェクト内からの参照は、同じオブジェクト内でのみ結合されます。

シンボル機能データ項目のファミリの作成

初期化されたデータの複数のインスタンス (各インスタンスはシステムに固有) を同じオブジェクト内で提供できます。しかし、このようなデータは、関数インタフェースを介して提供する方法が簡単で、お勧めです。75 ページの「シンボル機能関数ファミリの作成」を参照してください。実行可能ファイル内に初期化データの複数のインスタンスを提供するには、特別な配慮が必要です。

次の例では、`foo.c` 内のデータ項目 `foo` を初期化して、マシン名の文字列を指しています。このファイルはさまざまなマシン用にコンパイルでき、各インスタンスはマシン機能で特定されます。このデータ項目への参照は、ファイル `bar.c` の `bar()` から行われます。次に共有オブジェクト `foobar.so.1` が、`foo` の 2 つの機能インスタンスと `bar()` を結合することで作成されます。

```
$ cat foo.c
char *foo = MACHINE;
$ cat bar.c
#include    <stdio.h>

extern char *foo = MACHINE;

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

$ elfdump -H foobar.so.1

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID       sun4u
       [2]  CA_SUNW_MACH     sun4u

  Symbols:
     index    value      size      type bind oth ver shndx          name
       [1]  0x000108d4 0x00000004  OBJT LOCL  D    0 .data          foo%sun4u

 Symbol Capabilities:
     index  tag               value
       [4]  CA_SUNW_ID       sun4v
```

```
      [5]  CA_SUNW_MACH     sun4v

  Symbols:
     index    value      size      type bind oth ver shndx          name
       [2]  0x000108d8 0x00000004  OBJT LOCL  D    0 .data          foo%sun4v
```

アプリケーションは bar() を参照できます。実行時リンカーはベースとなるシステムに関連する foo のインスタンスに結合します。

```
$ uname -m
sun4u
$ main
machine: sun4u
```

このコードが適切に動作するには、コードが位置独立になるようにコンパイルされる必要があります。共有可能なオブジェクト内のコードではこれは一般的です。186 ページの「位置独立のコード」を参照してください。位置独立のデータ参照は間接参照であるため、実行時リンカーは必要な参照を検索して、データセグメントの要素を更新できます。データセグメントのこの再配置更新では、テキストセグメントが読み取り専用として保持されます。

しかし、実行可能ファイル内のコードは位置依存であることが一般的です。また、実行可能ファイル内のデータ参照はリンク編集時に結合されます。実行可能ファイル内では、実行時リンカーがシンボル機能ファミリから選択できるように、シンボル機能のデータ参照は、大域データ項目を介して解決されない状態のままである必要があります。前の例の bar.c にある bar() からの参照が位置依存コードとしてコンパイルされた場合、実行可能ファイルのテキストセグメントは実行時に再配置される必要があります。デフォルトでは、この状態は重大なリンク時エラーとなります。

```
$ cc -o main main.c bar.c foo.o foo.1.o foo.2.o ...
warning: Text relocation remains                referenced
    against symbol                  offset      in file
foo                                 0x0         bar.o
foo                                 0x8         bar.o
```

このエラー状態を解決する 1 つの方法は、bar.c を位置独立としてコンパイルすることです。ただし、この方法を正常に動作させるには、シンボル機能データ項目への実行可能ファイル内からの参照をすべて位置独立でコンパイルする必要がある点に注意してください。

データはシンボル機能メカニズムを使用してアクセスできますが、データ項目をオブジェクトへの公開インタフェースの一部にすることは問題となる可能性があります。より柔軟な別のモデルは、シンボル機能関数に各データ項目をカプセル化することです。この関数が、データをアクセスするための唯一の手段を提供します。シンボル機能関数にデータを隠すことによって、データを静的に定義し、非公開のままに維持できるという重要なメリットが得られます。前の例は、シンボル機能関数を使用するようにコーディングできます。

```
$ cat foobar.c
cat bar.c
#include    <stdio.h>

static char *foo = MACHINE;

void bar()
{
        (void) printf("machine: %s\n", foo);
}
$ elfdump -H main

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID       sun4u
       [2]  CA_SUNW_MACH     sun4u

  Symbols:
     index    value      size      type bind oth ver shndx          name
       [1]  0x0001111c 0x0000001c  FUNC LOCL  D    0 .text          bar%sun4u

 Symbol Capabilities:
     index  tag               value
       [4]  CA_SUNW_ID       sun4v
       [5]  CA_SUNW_MACH     sun4v

  Symbols:
     index    value      size      type bind oth ver shndx          name
       [2]  0x00011138 0x0000001c  FUNC LOCL  D    0 .text          bar%sun4v

$ uname -m
sun4u
$ main
machine: sun4u
```

オブジェクト機能のシンボル機能への変換

理想的には、コンパイラはシンボル機能で特定されたオブジェクトを生成できます。コンパイラがシンボル機能を解決できない場合は、リンカーが解決します。

オブジェクト機能を定義している再配置可能オブジェクトは、リンカーを使用してシンボル機能を定義する再配置可能オブジェクトに変換できます。リンカーの `-z symbolcap` オプションを使用すると、機能データセクションがすべて変換されて、シンボル機能が定義されます。オブジェクト内のすべての大域関数は局所関数に変換され、シンボル機能に関連付けられます。初期化された大域データ項目は局所データ項目に変換され、シンボル機能に関連付けられます。変換されたこれらのシンボルには、オブジェクト機能グループの一部として指定された機能識別子が追加されます。機能識別子が定義されていない場合は、デフォルトのグループ名が追加されます。

オリジナルの大域関数または初期化されたデータ項目のそれぞれに対して、大域参照が作成されます。この参照は再配置要件に関連付けられており、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ためにこのオブジェクトを最終的に結合するときに、デフォルトの大域シンボルに結合できます。

注 – -z symbolcap オプションは、オブジェクト機能セクションを含んでいるオブジェクトに適用されます。このオプションは、すでにシンボル機能を含んでいる再配置可能オブジェクト、またはオブジェクトおよびシンボルの機能を含む再配置可能オブジェクトに影響しません。この設計により、複数のオブジェクトがリンカーによって結合でき、オブジェクト機能を含むオブジェクトだけがこのオプションの影響を受けます。

次の例では、x86 再配置可能オブジェクトは 2 つの大域関数 foo() および bar() を含んでいます。このオブジェクトは MMX と SSE のハードウェア機能を必要とするようにコンパイルされています。これらの例では、機能グループは機能識別子エントリを使用して名前が付けられました。この識別子名は変換されたシンボル名に追加されます。この明示的な識別子がないと、リンカーはデフォルトの機能グループ名を追加します。

```
$ elfdump -H foo.o

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_ID        sse,mmx
       [1]  CA_SUNW_HW_1      0x840  [ SSE MMX ]

$ elfdump -s foo.o | egrep "foo|bar"
      [25]  0x00000000 0x00000021  FUNC GLOB  D    0 .text    foo
      [26]  0x00000024 0x0000001e  FUNC GLOB  D    0 .text    bar

$ elfdump -r foo.o | fgrep foo
  R_386_PLT32               0x38       .rel.text             foo
```

これで、この再配置可能オブジェクトはシンボル機能の再配置可能オブジェクトに変換できます。

```
$ ld -r -o foo.1.o -z symbolcap foo.o
$ elfdump -H foo.1.o

Capabilities Section:  .SUNW_cap

 Symbol Capabilities:
     index  tag               value
       [1]  CA_SUNW_ID        sse,mmx
       [2]  CA_SUNW_HW_1      0x840  [ SSE MMX ]

  Symbols:
     index    value      size      type bind oth ver shndx    name
      [25]  0x00000000 0x00000021  FUNC LOCL  D    0 .text    foo%sse,mmx
      [26]  0x00000024 0x0000001e  FUNC LOCL  D    0 .text    bar%sse,mmx
```

```
$ elfdump -s foo.1.o | egrep "foo|bar"
      [25]  0x00000000 0x00000021  FUNC LOCL  D    0 .text    foo%sse,mmx
      [26]  0x00000024 0x0000001e  FUNC LOCL  D    0 .text    bar%sse,mmx
      [37]  0x00000000 0x00000000  FUNC GLOB  D    0 UNDEF    foo
      [38]  0x00000000 0x00000000  FUNC GLOB  D    0 UNDEF    bar

$ elfdump -r foo.1.o | fgrep foo
  R_386_PLT32               0x38       .rel.text             foo
```

このオブジェクトは、同じ関数 (別のシンボル機能に関連) のインスタンスを含む別のオブジェクトと結合でき、実行可能ファイルまたは共有オブジェクトを作成できるようになりました。また、シンボル機能と関連付けられていない、各機能ファミリの先頭となる各関数のデフォルトインスタンスが提供される必要があります。このデフォルトのインスタンスはすべての外部参照に対応しており、どのシステムでも関数のインスタンスが確実に使用できるようになります。

実行時、`foo()` と `bar()` へのすべての参照は、先頭のインスタンスに向けられます。ただし、システムが適切な機能に対応している場合、実行時リンカーは最適なシンボル機能インスタンスを選択します。

機能ファミリの実行

通常、オブジェクトは特定のアーキテクチャーのすべてのシステム上で実行できるように設計され、構築されています。しかし、各システムが特別の機能を持っていると、多くの場合、最適化の対象となります。最適化されたコードは、前のセクションで説明したメカニズムを使用して、そのコードを実行するために必要とする機能で特定できます。

最適化されたインスタンスを実行したり、テストしたりするには、必要な機能を備えたシステムを使用する必要があります。システムごとに、実行時リンカーは使用可能な機能を判断してから、機能にもっとも対応できるインスタンスを選択します。テストと実験を支援するため、実行時リンカーに対してシステムに備わった機能ではなく機能の代替セットを使用するように指示できます。また、これらの代替機能に対して特定のファイルだけが検証されるようにも指定できます。

機能の代替セットはシステム機能から派生したもので、再度初期化したり、機能を追加または削除したりできます。

環境変数のファミリを使用すると、機能の代替セットを作成したり、その使用対象を設定したりできます。

`LD_PLATCAP={`*name*`}`
: 代替プラットフォームの名前を識別します。

`LD_MACHCAP={`*name*`}`
: 代替マシンハードウェアの名前を識別します。

LD_HWCAP=[+-]{*token* | *number*},...
代替ハードウェア機能の値を識別します。

LD_SFCAP=[+-]{*token* | *number*},...
代替ソフトウェア機能の値を識別します。

LD_CAP_FILES=*file*,...
代替機能に対して検証すべきファイルを指定します。

機能環境変数 LD_PLATCAP および LD_MACHCAP は、プラットフォーム名とマシンのハードウェア名をそれぞれ定義する文字列を受け入れます。69 ページの「プラットフォーム機能の特定」および70 ページの「マシン機能の指定」を参照してください。

機能環境変数 LD_HWCAP および LD_SFCAP は、機能のシンボル表現としてコンマ区切りのトークンリストを受け入れます。71 ページの「ハードウェア機能の特定」および73 ページの「ソフトウェア機能の特定」を参照してください。トークンには数値も指定できます。

「+」接頭辞を付けると、次に続く機能が代替機能に追加されます。「-」接頭辞を付けると、次に続く機能が代替機能から削除されます。「+-」がないと、次に続く機能が代替機能と置き換わります。

機能を削除すると、エミュレートされる機能環境がより制限されます。一般的に、機能インスタンスのファミリが存在する場合、汎用的で機能に固有でないインスタンスも提供されます。このため、より制限された機能環境を使用すると、機能が少ない、または汎用のコードのインスタンスを使用するように強制できます。

機能を追加すると、エミュレートされる機能環境がより向上します。この環境は慎重に構築する必要がありますが、機能ファミリのフレームワークを実行するために使用できます。たとえば、mapfile を使用すると、期待される機能を定義する関数ファミリを作成できます。これらの関数は printf(3C) を使用すると、その実行を確認できます。関連オブジェクトの作成を検証したり、さまざまな機能を組み合わせて実行したりできるようになります。関数の実際の機能要件をコーディングする前に、このように機能ファミリをプロトタイピングすると有益です。しかし、ファミリインスタンス内のコードが適切に動作するために特定機能を必要とし、この機能がシステムに備わっていないが代替機能として設定されている場合、このコードインスタンスは正しく動作しません。

LD_CAP_FILES も使用しないで一連の代替機能を作成すると、プロセスのすべての機能固有オブジェクトが代替機能に対して検証されます。この方法も注意深く実行してください。多くのシステムオブジェクトは、正常に動作するために、システム機能を必要とするためです。機能を変更すると、システムオブジェクトが正常に動作しなくなる場合があります。

機能の実験を行うために最適な環境は、オブジェクトで使用するすべての機能を備えたシステムを使用することです。LD_CAP_FILES も使用して、実験するオブジェク

トを分離してください。次に、「-」構文を使用して機能を無効にすると、ユーザーのさまざまな機能ファミリのインスタンスを実行できます。各インスタンスは、システムの本物の機能で完全にサポートされます。

たとえば、x86 の 2 つの機能オブジェクト libfoo.so および libbar.so があるとします。これらのオブジェクトには、SSE2 命令を使用するように最適化された機能関数、MMX 命令を使用するように最適化された関数、および機能を必要としない汎用の関数が含まれています。ベースとなるシステムには SSE2 と MMX が両方ともに備わっています。デフォルトでは、完全に最適化された SSE2 関数が使用されます。

LD_HWCAP 定義を使用することによって、SSE2 機能を削除して、MMX 命令用に最適化された関数を使用するように、libfoo.so および libbar.so を制限できます。LD_CAP_FILES を定義するもっとも柔軟な方法は、必要なファイルのベース名を使用することです。

```
$ LD_HWCAP=-sse2 LD_CAP_FILES=libfoo.so,libbar.so ./main
```

SSE2 および MMX の機能を削除して、汎用の関数だけを使用するように libfoo.so および libbar.so をさらに制限できます。

```
$ LD_HWCAP=-sse2,mmx LD_CAP_FILES=libfoo.so,libbar.so ./main
```

注 – アプリケーションで使用できる機能および設定されたすべての代替機能は、実行時リンカーの診断機能を使用すると確認できます。

```
$ LD_DEBUG=basic LD_HWCAP=-sse2,mmx,cx8 ./main
....
02328: hardware capabilities (CA_SUNW_HW_1) - 0x5c6f  \
    [ SSE3 SSE2 SSE FXSR MMX CMOV SEP CX8 TSC FPU ]
02328: alternative hardware capabilities (CA_SUNW_HW_1) - 0x4c2b  \
    [ SSE3 SSE FXSR CMOV SEP TSC FPU ]
....
```

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、186 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、次のセクション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、196 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 つ以上の動的 `DT_FLAGS_1` フラグを立てます (表 13–10 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が生成されます。

```
$ 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。60 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。

スタブオブジェクト

スタブオブジェクトは共有オブジェクトで、mapfiles からすべて作成されます。また、コードやデータを持ちませんが、本物のオブジェクト同じリンクインタフェースを提供します。スタブオブジェクトは、実行時には使用できません。しかし、スタブオブジェクトに対してアプリケーションを作成できます。スタブオブジェクトによって、実行時に使用される実際のオブジェクト名が提供されます。

スタブオブジェクトの作成時には、リンカーはコマンド行で指定されているオブジェクトとライブラリファイルをすべて無視するため、スタブを作成するためにこれらのファイルが存在している必要はありません。コンパイル手順が省略でき、リンカーの作業が比較的少ないため、スタブオブジェクトはすぐに作成できます。

スタブオブジェクトは、構築時のさまざまな問題点の解決に利用できます。

- 速度

  最新のマシンで、処理を並列化できるバージョンの make ユーティリティーを使用すると、多数のオブジェクトを同時にコンパイルしてリンクできるため、大幅なスピードアップが図れます。しかし、あるオブジェクトがほかのオブジェクトに依存したり、ほかのほぼすべてのオブジェクトが依存する中核となるオブジェクトセットが存在したりするのが一般的です。すべてのオブジェクトが別のオブジェクトで使用される前に作成されるように、構築の順番を付ける必要があります。この順序付けによってボトルネックが生じ、並列処理できる分量が減ったり、コードを作成できる全体の速度が制限されたりします。

- 複雑さと正確さ

  大規模なコードの場合、さまざまなオブジェクト間に大量の依存関係が存在するときがあります。これらのオブジェクトに関する makefiles などの構築の記述が非常に複雑になり、把握と維持が困難になる場合があります。システムの変化に伴って、依存関係が変わる場合があります。これにより、ある makefiles のセットが徐々に不正確になり、競合状態が発生したり、不可解でまれに起こる構築障害に繋がったりします。

- 依存関係のサイクル

  協力し合う共有オブジェクトを作成し、各オブジェクトがリソースを提供して互いに利用し合うようにコードを設計することが望ましい場合もあります。ただし、あるオブジェクトが、そのオブジェクトを使用するオブジェクトの前に作成されなければならない環境では、このようなサイクルはサポートできません。実行時リンカーがこのようなオブジェクト (作成できるとしても) の読み込みと使用に完全に対応していますが、サポートできません。

スタブの共有オブジェクトは、上記の問題点を回避する別のコード作成方法を提供します。ビルドによって作成されるすべての共有オブジェクトに対して、スタブオブジェクトをすぐに作成できます。その後、実際の共有オブジェクトと実行可能ファイルはすべて、リンク時に実際のオブジェクトの代わりを務めるスタブオブジェクトを使用して、平行してどのような順序でも作成できます。実行可能ファイルと実際の共有オブジェクトは保持され、スタブの共有オブジェクトは破棄されます。

スタブオブジェクトは 1 つまたは複数の mapfile から作成され、次の要件を集合的に満たす必要があります。

- 少なくとも、1 つの mapfile で STUB_OBJECT 指令を指定する必要があります。225 ページの「STUB_OBJECT 指令」を参照してください。
- オブジェクトへの外部インタフェースを構成する関数とデータのすべてのシンボルは、mapfile 内に明示的に列挙される必要があります。
- mapfile はシンボル範囲の縮小 ('*') を使用して、明示的に列挙されていないすべてのシンボルを外部インタフェースから削除する必要があります。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」を参照してください。
- オブジェクトからエクスポートされるすべての大域データには、シンボルのタイプとサイズを指定するために、mapfile に ASSERT シンボル属性が必要です。同じデータを参照するシンボルが複数ある場合、いずれかのシンボルの ASSERT で TYPE と SIZE の属性を指定し、その他では ALIAS 属性を使用して、このプライマリシンボルを参照する必要があります。228 ページの「ASSERT 属性」を参照してください。

このような mapfile を使用すると、共有オブジェクトのスタブと実オブジェクトそれぞれの作成時に同じコマンド行を使用できます。-z stub オプションは、スタブオブジェクトのリンク編集に追加され、実オブジェクトのリンク編集から削除されます。

これらの考えを示すため、次のコードで名前が idx5 の共有オブジェクトを実装します。このオブジェクトは 5 つの要素を持つ整数の配列からデータをエクスポートします。各要素は初期化され、ゼロから始まる配列インデックスを格納します。このデータは、大域配列として、また結合が弱い代替の別名データシンボルとして、関数インタフェースを介して公開されます。

```
$ cat idx5.c
int _idx5[5] = { 0, 1, 2, 3, 4 };
#pragma weak idx5 = _idx5

int
idx5_func(int index)
{
        if ((index < 0) || (index > 4))
                return (-1);
        return (_idx5[index]);
}
```

mapfile では、この共有オブジェクトで提供されるインタフェースを記述する必要があります。

```
$ cat mapfile
$mapfile_version 2
STUB_OBJECT;
SYMBOL_SCOPE {
        _idx5   {
                        ASSERT { TYPE=data; SIZE=4[5] };
                };
        idx5    {
                        ASSERT { BINDING=weak; ALIAS=_idx5 };
                };
        idx5_func;
    local:
        *;
};
```

次のメインプログラムは、idx5 共有オブジェクトから利用可能なすべてのインデックス値を出力するために使用されます。

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

extern int      _idx5[5], idx5[5], idx5_func(int);

int
main(int argc, char **argv)
{
        int     i;
        for (i = 0; i < 5; i++)
                (void) printf("[%d] %d %d %d\n",
                    i, _idx5[i], idx5[i], idx5_func(i));
        return (0);
}
```

次のコマンドでは、この共有オブジェクトのスタブ版を、stublib という名前のサブディレクトリに作成します。elfdump コマンドは、作成されるオブジェクトがスタブであることを検証するために使用されます。スタブの作成に使用されるコマンドは、-z stub オプションを追加する点、および異なる出力ファイル名を使用する点だけが、実オブジェクトのコマンドと異なります。これは、スタブ作成が既存コードに容易に追加できることを示します。

```
$ cc -Kpic -G -M mapfile -h libidx5.so.1 idx5.c -o stublib/libidx5.so.1 -zstub
$ ln -s libidx5.so.1 stublib/libidx5.so
$ elfdump -d stublib/libidx5.so | grep STUB
      [11]  FLAGS_1           0x4000000           [ STUB ]
```

これで、実際の共有オブジェクトの代わりになるスタブオブジェクトを使用して、メインプログラムを構築できるようになりました。スタブオブジェクトは、実行時に実オブジェクトを検索する「実行パス」を設定します。しかし、実オブジェクトはまだ作成されていないため、このプログラムはまだ動作しません。システムがスタブオブジェクトを読み込もうとする試みは拒否されます。実行時リンカーは、実オブジェクトにある実際のコードとデータがスタブオブジェクトにないために、実行できないことをわかっているためです。

```
$ cc main.c -L stublib -R '$ORIGIN/lib' -lidx5 -lc
$ ./a.out
ld.so.1: a.out: fatal: libidx5.so.1: open failed: \
    No such file or directory
Killed
$ LD_PRELOAD=stublib/libidx5.so.1 ./a.out
ld.so.1: a.out: fatal: stublib/libidx5.so.1: stub shared object \
    cannot be used at runtime
Killed
```

実オブジェクトの作成には、スタブオブジェクトを作成するために使用したコマンドと同じものを使用します。-z stub オプションは省かれ、実際の出力ファイルのパスが指定されます。

```
$ cc -Kpic -G -M mapfile -h libidx5.so.1 idx5.c -o lib/libidx5.so.1
```

実オブジェクトが lib サブディレクトリに作成されたあとは、このプログラムを実行できます。

```
$ ./a.out
[0] 0 0 0
[1] 1 1 1
[2] 2 2 2
[3] 3 3 3
[4] 4 4 4
```

デバッグ支援

このリンカーは、リンク編集プロセスを詳細にトレースできるデバッグ機能を備えています。この機能は、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。この機能によって表示される情報の種類は変更されない予定です。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。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。

-D で使用できるトークンは、コマンド行に -D help を入力すると表示できます。

```
$ ld -Dhelp
```

デフォルトでは、すべてのデバッグ出力が標準エラー出力ファイルである stderr に送られます。output トークンを使用すると、ファイルにデバッグを出力できます。たとえば、名前が ld-debug.txt のファイルにヘルプテキストを取り込むことができます。

```
$ ld -Dhelp,output=ld-debug.txt
```

ほとんどのコンパイラドライバは -D オプションに別の意味を割り当てており、多くの場合、前処理用マクロを定義します。LD_OPTIONS 環境変数を使用すると、コンパイラドライバをバイパスして、-D オプションを直接リンカーに指定できます。

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、このドキュメント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オーバーライドされます。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。35 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設定および実行するときは、アプリケーションとその依存関係の結合を完了するためにインタプリタが呼び出されます。Oracle 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、Oracle 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。393 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。

- 実行可能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します。

- 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します。

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 が必要です。

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。140 ページの「命名規約」および 141 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、44 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 1 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、43 ページの「環境変数の使用」 と 44 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。131 ページの「機能のデバッグ」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 ]
...
```

依存関係が見つからない場合、ldd(1) はオブジェクトを検出できないことを示します。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構成

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。

- $CAPABILITY – オブジェクトが提供するさまざまな機能を読み込めるディレクトリを示します。263 ページの「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。265 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。268 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。267 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。267 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。267 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、358 ページの「SPARC: 再配置」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。

`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。

`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。

`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。

実行時のシンボル検索は、シンボルの可視性によって指定することもできます。`STV_SINGLETON` 可視性を割り当てたシンボルは、すべてのシンボル検索範囲から

除外されます。シングルトンシンボルへのすべての参照は、プロセス内で最初に定義されているシングルトンにバインドされます。表 12–20 を参照してください。

もっとも単純な形のシンボル検索については、次のセクション102 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、121 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。第 6 章「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
```

シンボル `bar` の再配置が要求された場合、実行時リンカーは最初に動的実行可能ファイル `prog` で `bar` を検索します。シンボルが見つからない場合は、次に共有オブジェクト `/home/me/lib/libfoo.so.1` を検索し、最後に共有オブジェクト `/home/me/lib/libbar.so.1` 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。この性能については、第 7 章「システムのパフォーマンスを最適化するオブジェクトの構築」で詳しく説明しています。これに代わる検索モデルについては、第 6 章「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、49 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、60 ページの「シンボル範囲の縮小」で説明しています。

注 – STV_SINGLETON 可視性を割り当てたシンボルでは、一種の割り込みが行われます。シングルトンシンボルへのすべての参照は、プロセス内で最初に定義されているシングルトンにバインドされます。表 12–20 を参照してください。

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。

動的実行可能ファイル内の個々のシンボルは、INTERPOSE mapfile キーワードを使って割り込み処理として定義できます。このメカニズムは -z interpose オプションを使用する方法よりも優先されるので、依存関係が展開されていくときに発生する可能性のある逆割り込みの影響を受けにくくなります。177 ページの「明示的な割り込みの定義」を参照してください。

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。デフォルトモードは、環境変数 LD_BIND_NOW にヌル以外の任意の値を設定することでオーバーライドできます。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、52 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証が抑制されます。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、動的実行可能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) を使用して検証できます。

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" ldd 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、104 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、リンカーのオプション `-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。118 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);
        ....
}
```

dlopen() と dlsym() を使用するこのモデルは、非常に柔軟ですが、不自然なコーディングスタイルであり、欠点がいくつかあります。

- シンボルが終了すると予想されるオブジェクトは既知である必要があります。
- 関数ポインタを介した呼び出しには、コンパイラまたは lint(1) で検証する手段がありません。

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。

関数参照が遅延読み込みをトリガーできることを利用すると、libbar.so.1 と同じ遅延読み込みが実現できます。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。このコーディングの方が自然です。また、標準関数呼び出しを使用することによって、コンパイラまたは lint(1) で検証できるようになります。

```
void foo()
{
    bar1(arg1);
    bar2(arg2);
    ....
}
$ cc -G -o libfoo.so.1 foo.c -L. -zdefs -zlazyload -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、依存関係の名前がわからなくても、その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。

関数の有無をテストするための堅牢なモデルは、明示的に定義された遅延依存関係を使用することによって、また RTLD_PROBE ハンドルを指定して dlsym(3C) を使用することによって実現できます。

明示的に定義された遅延依存関係は、遅延読み込み可能な依存関係の拡張です。遅延依存関係に関係付けられたシンボル参照は、遅延シンボルと呼ばれます。このシンボルに対する再配置が処理されるのは、シンボルが最初に参照されるときだけです。これらの再配置は、LD_BIND_NOW 処理の一部としても、また RTLD_NOW フラグ付きの dlsym(3C) を介しても処理されません。

遅延依存関係は、リンク編集時にリンカーの -z deferred オプションを使用して確立されます。

```
$ cc -G -o libfoo.so.1 foo.c -L. -zdefs -zdeferred -lbar -R'$ORIGIN'
```

遅延依存関係として libbar.so.1 を確立した場合、bar1() への参照によって、その依存関係が利用できることを検証できます。このテストを使用すると、dlsym(3C) を使用した場合と同じ方法で、依存関係によって提供される関数への参照を制御できます。次に、このコードは bar1() および bar2() を自然に呼び出すことができます。これらの呼び出しは判読しやすく、コーディングも容易であるため、コンパイラが呼び出しシーケンスでのエラーを見つけられるようになります。

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

遅延依存関係によって柔軟性が向上します。依存関係がまだ読み込まれていない場合、その依存関係は実行時に変更できます。このメカニズムにより、dlopen(3C) と同じ程度の柔軟性が得られます。つまり、呼び出し元によって、異なるオブジェクトを読み込ませたり、異なるオブジェクトに結合させたりできるようになります。

元の依存関係名が既知の場合、RTLD_DI_DEFERRED 引数を指定して dlinfo(3C) を使用すると、元の依存関係を新しい依存関係と交換できます。また、依存関係に関連する遅延シンボルを使用すると、RTLD_DI_DEFERRED_SYM 引数を持つ dlinfo(3C) を使用して遅延依存関係を特定できます。

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 .preinit_array、.init_array、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、.preinit_array セクションと .init_array 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。オブジェクトに .init と .init_array の両方のセクションが含まれている場合、そのオブジェクトの .init_array セクションで定義された関数の前に、.init セクションが処理されます。

動的実行可能ファイルは、.preinit_array セクション内で「初期設定前」関数を提供することができます。これらの関数は、実行時リンカーがプロセスイメージを構築して再配置を実行し終わった後で、かつ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 .fini_array 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、.fini_array 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 と .fini_array の両方のセクションが含まれている場合、.fini_array セクションで定義された関数が、そのオブジェクトの .fini セクションの前に処理されます。

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、45 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。131 ページの「機能のデバッグ」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。この再配置の促進によって、関数呼び出しを動的に解決するときに使用可能な動的初期設定機能が実質的に抑制されます。

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。54 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。38 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。144 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

セキュアなプロセスには、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、そのプロセスをセキュアとして分類します。

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、セキュアな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。

あるセキュアなプロセスに対して `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。97 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。

セキュアなプロセスでは、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。

セキュアなプロセスでは、`$ORIGIN` 文字列の拡張は、その文字列がトラストディレクトリに拡張されるときにかぎり許可されます。271 ページの「セキュリティー」を参照してください。ただし、`$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アです。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。

セキュアなプロセスでは、`LD_CONFIG` は無視されます。ただし、セキュアなアプリケーションで記録された構成ファイルは使用されます。`ld(1)` の `-c` オプションを参照してください。記録済み構成ファイルは、完全パス名、つまり「/」で始まるパス名である必要があります。記録される構成ファイルが `$ORIGIN` 文字列を使用する場合、そのファイルは既知のトラストディレクトリに制限されます。セキュアなアプリケーション内の構成ファイルを記録する開発者は、構成ファイル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。記録済み構成ファイルが存在せず、デフォルトの構成ファイルが存在する場合は、セキュアなプロセスはこのデフォルト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。

セキュアなプロセスでは、LD_SIGNAL は無視されます。

セキュアな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。

セキュアなプロセスでは、単純ファイル名を構成する依存関係は、前述のパス名の制約を使用して処理されます。フルパス名または相対パス名で表示された依存関係は、そのまま使用されます。そのため、セキュアな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。

セキュアな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) を使用すると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 – Oracle 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。97 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、259 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が見つからない場合、dlopen(3C) は `NULL` ハンドルを返します。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ときに 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。第 6 章「直接結合」を参照してください。

注 – STV_SINGLETON 可視性を割り当てたシンボルは、dlopen(3C) 属性とは関係なく、デフォルトのシンボル検索を使ってバインドされます。表 12–20 を参照してください。

dlopen(3C) を使用して入手したオブジェクトには、デフォルトでワールドシンボル検索範囲と局所シンボル可視性が割り当てられます。122 ページの「デフォルトのシンボル検索モデル」セクションでは、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。125 ページの「大域オブジェクトの定義」、126 ページの「グループの分離」、および 126 ページの「オブジェクト階層」セクションでは、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。

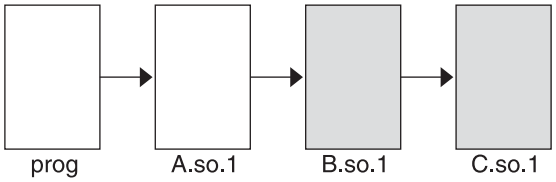
デフォルトのシンボル検索モデルは、遅延読み込み環境の遷移も行います。現在読み込まれているオブジェクト内でシンボルが見つからない場合は、そのシンボルを特定するために、保留となっている遅延読み込みオブジェクトが処理されます。この読み込みによって、依存関係を完全には定義していないオブジェクトを補います。ただし、これにより遅延読み込みの利点が失われることがあります。

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、動的実行可能プログラム 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) が実行されたあとで、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。

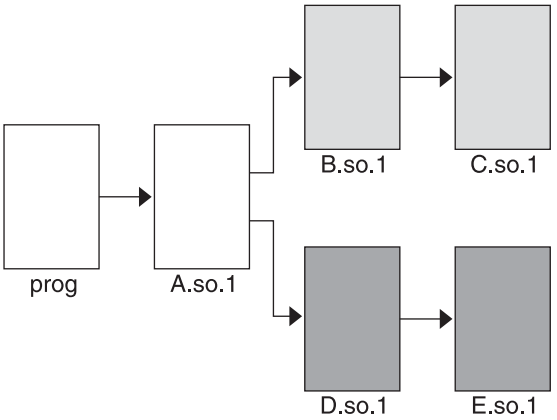
これらのシンボル検索とシンボル可視性の属性は、オブジェクト間の関係を保持します。これらの関係は、そのプロセスのアドレススペースへの投入とオブジェクト間の依存の関係に基づいています。指定された dlopen(3C) に関連したオブジェクトを固有のグループに割り当てることにより、同じ dlopen(3C) と関連したオブジェクトだけが、グループ内のシンボルと、関連する依存関係の中の検索ができます。

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() リクエスト



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。

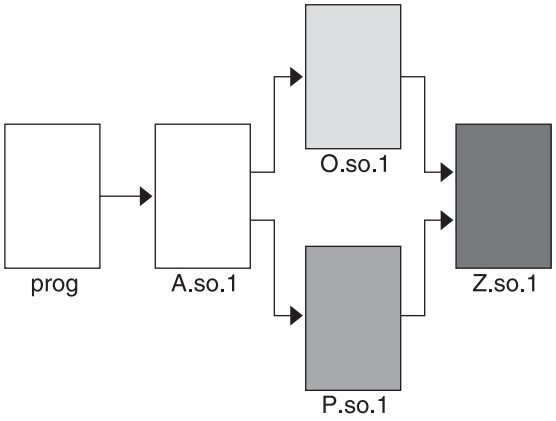
オブジェクトが、前述した処理の進行の中で使用される場合、それぞれの dlopen(3C) が実施された順序は、結果として発生するシンボル結合には影響しません。ただし、複数のオブジェクトに共通の依存関係がある場合は、結果の結び付きは、dlopen(3C) 呼び出しが実行された順序の影響を受けます。

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局所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されることができます。

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 – グループのメンバーが、局所シンボルの可視性を定義しており、かつ大域シンボルの可視性を定義するほかのグループによって参照されている場合、オブジェクトの可視性は局所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 – グループのメンバーがグループの検索要件を定義しており、かつワールド検索要件を定義する別のグループによって参照されている場合、オブジェクトの検索要件はグループとワールドの両方を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() を使用してセカンダリ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、シンボルをセカンダリ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。
- セカンダリオブジェクトに対する dlopen(3C)で RTLD_PARENT モードフラグを使用します。

最初のオブジェクトをセカンダリオブジェクトの明示的な依存関係にした場合、これはセカンダリ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、セカンダリ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セカンダリ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、そのセカンダリ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、セカンダリオブジェクトの 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、セカンダリ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、セカンダリオブジェクトの 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 を使用してセカンダリ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) を使ってセカンダリ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) を使用すると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名を取り、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) から返されます。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) から返されます。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、関連付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ます。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ます。

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 101 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。

必要なシンボルが見つからない場合、dlsym(3C) は NULL 値を返します。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、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。

RTLD_DEFAULT ハンドルは、実行時リンカーで使用される規則と同じものを使用して、呼び出し元オブジェクトからのすべての参照を解決します。122 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。このモデルの 2 つの特徴に注意をしてください。

- 動的実行可能ファイルからの同じシンボル参照に一致するシンボル参照は、実行可能ファイルからの参照に関連するプロシージャーリンクテーブルのエントリに結合されます。425 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。動的リンクのこの動作によって、プロセス内のすべてのコンポーネントが、ある関数に対して 1 つのアドレスを参照することが保証されます。
- プロセス内に現在読み込まれているオブジェクトの中でウィーク以外のシンボル参照を満たすシンボル定義を検出できない場合は、遅延読み込みのフォールバックが開始されます。このフォールバックは、読み込まれる動的オブジェクトごとに繰り返され、保留中の遅延読み込み可能オブジェクトを読み込んでシンボルの解決を試みます。このモデルは、依存関係を完全には定義していなかった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メリットが損なわれることがあります。再配置シンボルが見つからない場合に、不必要なオブジェクトが読み込まれたり、すべての遅延読み込み可能オブジェクトが完全に読み込まれたりする可能性があります。

RTLD_PROBE は RTLD_DEFAULT と同様のモデルに従いますが、RTLD_DEFAULT で説明した 2 つの点が異なります。RTLD_PROBE は明示的なシンボル定義に結合するだけであり、実行可能ファイル内のプロシージャーリンクテーブルのエントリに結合されません。また、RTLD_PROBE では完全な遅延読み込みフォールバックは起動されません。既存プロセス内でシンボルの有無を検出するには、RTLD_PROBE フラグを使用するのが最適です。

RTLD_DEFAULT と RTLD_PROBE はともに明示的な遅延読み込みを起動できます。オブジェクトは関数を参照でき、その参照は遅延読み込み可能な依存関係を介して確立できます。この関数を呼び出す前に、RTLD_DEFAULT または RTLD_PROBE を使用すると関数の有無をテストできます。オブジェクトはこの関数を参照しているため、関連する遅延依存関係を読み込む試みが最初に行われます。次に、RTLD_DEFAULT と RTLD_PROBE の規則に従って関数に結合されます。次の例では、RTLD_PROBE の呼び出しを使用して、遅延読み込みのトリガーと、依存関係が存在する場合に、読み込まれた依存関係に結合します。

```
void foo()
{
    if (dlsym(RTLD_PROBE, "foo1")) {
        foo1(arg1);
        foo2(arg2);
        ....
}
```

機能性をテストするモデルを、堅牢で柔軟なモデルにするには、関連する遅延依存関係に明示的に *deferred* とタグ付けしてください。110 ページの「dlopen() の代替手段の提供」を参照してください。このタグ付けによって、実行時に遅延依存関係を変更する手段も提供されます。

54 ページの「ウィークシンボル」で説明したように、RTLD_DEFAULT または RTLD_PROBE を使用すると、未定義のウィーク参照の使用に代わる、より堅牢な手段が提供されます。

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 に示されている例のファイル 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

Oracle Solaris 実行時リンカーには、デバッグライブラリとデバッグを行う mdb(1) モジュールが備わっています。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。

機能のデバッグ

実行時リンカーには、アプリケーションの実行時リンクとその依存関係を詳細に追跡できるデバッグ機能が備わっています。この機能によって表示される情報の種類は変更されない予定です。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。デバッグのすべての出力の前に、処理識別子が追加されます。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。

```
$ LD_DEBUG=help prog
```

prog は任意の動的実行可能ファイルです。制御が prog に移る前に、このプロセスはヘルプ情報を表示して終了します。実行可能ファイルの選択は重要ではありません。

デフォルトでは、すべてのデバッグ出力が標準エラー出力ファイルである stderr に送られます。output トークンを使用すると、ファイルにデバッグを出力できます。たとえば、名前が rtld-debug.txt のファイルにヘルプテキストを取り込むことができます。

```
$ LD_DEBUG=help,output=rtld-debug.txt prog
```

また、デバッグ出力は環境変数 LD_DEBUG_OUTPUT を設定することによってリダイレクトできます。LD_DEBUG_OUTPUT が使用された場合、処理識別子が接尾辞として出力ファイル名に追加されます。

LD_DEBUG_OUTPUT と output のトークンが両方指定された場合、LD_DEBUG_OUTPUT が優先されます。LD_DEBUG_OUTPUT と output のトークンが両方指定された場合、LD_DEBUG_OUTPUT が優先されます。fork(2) を呼び出すプログラムで output トークンを使用すると、それぞれのプロセスで同じファイルにデバッグ出力を書き込むことになります。デバッグ出力は順序に規則性がなく、不完全なものになります。このような場合には、LD_DEBUG_OUTPUT を使用して、個別のファイルに各プロセスのデバッグ出力を指定してください。

セキュアな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、196 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コンポーネントを明らかにし、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。これでデバッガモジュールの機能を検査できます。

```
$ 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。ただし、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す。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) リクエスト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。使用している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。

次のセクションでは、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、29 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 (312 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の一部として、明示的なファイル名で指定できますが、多くの場合 `-l` オプションを使用してリンカーのライブラリ検索機能を利用します。38 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 9 章「インタフェースおよびバージョン管理」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、106 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、96 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、37 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。97 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、97 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。

共有オブジェクト内で定義される各インタフェースは、`mapfile` キーワードの `FILTER` または `AUXILIARY` を使用することで、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` オプションを使用します。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` オプションを使用すると、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` キーワードまたはオプションは、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` を使用したり、リンカーの `-f` オプションを使用したりすると、補助ファイルとして機能するようにインタフェースを定義できます。この `mapfile` キーワードまたはオプションは、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` オプションを使用します。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ため、mapfile キーワード FUNCTION を使用して foo 用のシンボルテーブルエントリが確実に作成されるようにします。

```
$ cat filter.c
char *bar = "defined in filter";
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
         global:
                foo     { TYPE=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。いくつかの標準フィルタが、Oracle Solaris OS で使用されています。

/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。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」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。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」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。48 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 オプションを使用します。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
$mapfile_version 2
SYMBOL_SCOPE {
        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 Oracle Solaris OS で使用され、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します。たとえば、263 ページの「機能固有の共有オブジェクト」、265 ページの「命令セット固有の共有オブジェクト」、および267 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している各インタフェースと、補助フィルタを定義している各インタフェースは、同じ共有オブジェクト内に定義できます。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 オプション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。

パート II

クイックリファレンス

◆ ◆ ◆ 第 5 章

リンカーのクイックリファレンス

以降のセクションには、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、28 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。35 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。37 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。40 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。42 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。

高品質のオブジェクトを作成するために、長い期間をかけて、多くの機能がリンカーに追加されてきました。これらの機能により、さまざまな実行環境で効率的かつ確実にオブジェクトを使用できるようになります。ただし、既存の構築環境との下位互換性を確保するために、デフォルトではこれらの機能の多くが有効になっていません。たとえば、直接結合や遅延読み込みなどの機能は、明示的に有効にする必要があります。リンカーは、どの機能を適用するかを選択するプロセスを簡略化できる、-z guidance オプションを備えています。ガイダンスが要求されると、リンカーは警告ガイダンスメッセージを発行できます。使用するオプションやほかの関連する変更を提示するこれらのメッセージを使用すると、さらに高品質なオブジェクトを作成できます。新しい機能がリンカーに追加されたときや、高品質オブジェクトを作成するためにさらに適した方法が見つかったときなど、ガイダンスメッセージが徐々に変わる可能性があります。ld(1) 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には、-r オプションを使用します。

```
$ ld -r -o temp.o file1.o file2.o file3.o .....
```

静的実行プログラムの作成

注 – 静的実行プログラムの使用は制限されています。29 ページの「静的実行可能ファイル」を参照してください。一般的に静的実行可能ファイルには、別のプラットフォームまたはオペレーティングシステムの別のバージョンで実行できないようにするために、プラットフォーム固有の実装の詳細情報が含まれています。Oracle Solaris 共有オブジェクトの多くの実装は、dlopen(3C) や、dlsym(3C) などの動的リンク機能に依存しています。106 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、静的実行可能ファイルでは使用できません。

静的実行可能ファイルを作成するには、-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。40 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。
- リンカーの -z guidance オプションの使用をお勧めします。ガイダンスメッセージは、リンカーオプションに関して、また作成されたオブジェクトを改善できるほかの操作に関して提案します。
- 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。186 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 discard-unused=sections オプションを使用します。189 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。
- アプリケーションレジスタは、エンドユーザーが使用できる SPARC アーキテクチャーの予約済みの機能です。外部で使用することを目的としている SPARC 共有オブジェクトでは、共有オブジェクトがアプリケーションレジスタを使用しないようにするため、C コンパイラに -xregs=no%appl オプションを使用する必要があります。これによって、共有オブジェクトの実装を損なうことなく、すべての外部ユーザーがアプリケーションレジスタを使用できるようになります。
- 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。第 9 章「インタフェースおよびバージョン管理」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。259 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。54 ページの「共有オブジェクト出力ファイルの生成」を参照
- 不要な依存関係を回避します。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。38 ページの「共有オブジェクトの処理」を参照。または、`-z discard-unused=dependencies` 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関係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。108 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。143 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。
- 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。第 6 章「直接結合」を参照してください。

次の例は、上記のポイント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 discard-unused=sections -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。141 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。259 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、上記のポイント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 discard-unused=sections -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(192 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(194 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 (60 ページの「シンボル範囲の縮小」を参照)、関数インタフェースを経由して、データにアクセスできるようにします (196 ページの「コピー再配置」を参照)。

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。
- リンカーの -z guidance オプションの使用をお勧めします。ガイダンスメッセージは、リンカーオプションに関して、また作成されたオブジェクトを改善できるほかの操作に関して提案します。
- -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。108 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。
- 不要な依存関係を回避します。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。38 ページの「共有オブジェクトの処理」を参照。または、-z discard-unused=dependencies 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関係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。44 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。第 6 章「直接結合」を参照してください。

次の例は、上記のポイントを組み合わせたものです。

```
$ cc -o prog -R /home/lib -z discard-unused=dependencies -z lazyload -B direct -L. \
-lfoo file1.o file2.o file3.o .....
```

パート III

詳細情報

◆ ◆ ◆ 第 6 章

6 直接結合

実行時リンカーは、動的実行可能ファイルと多くの依存関係からプロセスを構築する一貫として、シンボル定義にシンボル参照を結合する必要があります。デフォルトでは、シンボル定義は簡単な検索モデルを使用して検出されます。一般にオブジェクトは 1 つずつ検索され、動的実行可能ファイルから始まって、オブジェクトが読み込まれる順に各依存関係が処理されます。このモデルは、最初に動的リンクが導入されて以来、有効でした。一般的に、このような簡単なモデルでは、すべてのシンボル参照が 1 つの定義に結合されることになります。結合される定義は、読み込まれた一連の依存関係の中で検出された最初の定義です。

動的実行可能ファイルは、動的リンクが初期段階であったときに作成された実行可能ファイルより、さらに複雑な処理に発展してきました。依存関係の数は何十から何百の単位に増加しました。動的オブジェクト間で参照されるシンボルインタフェースの数も大幅に増加しました。シンボル名のサイズは、C++ などの言語のサポートに使用される名前の符号化などの技術によってかなり増加しました。これらの要因によって、シンボル参照がシンボル定義に結合されるために、多くのアプリケーションで起動時間が増加しました。

プロセス内のシンボル数の増加は、名前空間の汚染の増加にもつながっています。同じ名前のシンボルのインスタンスが複数存在することは、より一般的になっています。同じシンボルの複数のインスタンスが存在することによって予期しない結合や誤った結合が発生したときに、処理の障害を診断することは多くの場合で困難です。

また現在では、プロセスの個々のオブジェクトが、同じ名前で多重定義されたシンボルの別々のインスタンスに対して結合する必要のあるプロセスが存在します。

シンボルの結合の柔軟性を向上させながら、デフォルトの検索モデルのオーバーヘッドに対応するため、別のシンボル検索モデルが作成されました。このモデルは直接結合と呼ばれます。

直接結合では、プロセスのオブジェクト間で確立される厳密な結合関係を可能にします。直接結合関係によって、意図しない結合から関連オブジェクトを分離するこ

とで、名前空間の不測の競合を回避できます。この保護によってプロセス内のオブジェクトの堅牢性が増すため、予期しない、診断が困難な結合の状況を回避できます。

直接結合は割り込みに影響を及ぼすことがあります。直接結合を採用することで、意図しない割り込みを回避できます。しかし、直接結合によって意図する割り込みも回避される場合があります。

この章では、直接結合モデルと、このモデルを使用するようにオブジェクトを変換するときに考慮する必要がある割り込みの問題について説明します。

シンボル結合の確認

デフォルトのシンボル検索モデルを理解して、このモデルを直接結合と比較するために、次のコンポーネントを使用してプロセスを構築します。

```
$ cat main.c
extern int W(), X();

int main() { return (W() + X()); }

$ cat W.c
extern int b();

int a() { return (1); }
int W() { return (a() - b()); }

$ cat w.c
int b() { return (2); }

$ cat X.c
extern int b();

int a() { return (3); }
int X() { return (a() - b()); }

$ cat x.c
int b() { return (4); }

$ cc -o w.so.1 -G -Kpic w.c
$ cc -o W.so.1 -G -Kpic W.c -R. w.so.1
$ cc -o x.so.1 -G -Kpic x.c
$ cc -o X.so.1 -G -Kpic X.c -R. x.so.1
$ cc -o prog1 -R. main.c W.so.1 X.so.1
```

アプリケーションのコンポーネントは次の順番で読み込まれます。

```
$ ldd prog1
        W.so.1 =>        ./W.so.1
        X.so.1 =>        ./X.so.1
        w.so.1 =>        ./w.so.1
        x.so.1 =>        ./x.so.1
```

W.so.1 と X.so.1 の両方のファイルで、a() という名前の関数を定義します。w.so.1 と x.so.1 の両方のファイルで、b() という名前の関数を定義します。さらに、W.so.1 と X.so.1 の両方のファイルで、関数 a() および b() を参照します。

実行時のシンボル検索 (デフォルトの検索モデルを使用) と最終的な結合は、LD_DEBUG 環境変数を設定することで確認できます。実行時リンカーの診断によって、関数 a() および b() への結合が確認できます。

```
$ LD_DEBUG=symbols,bindings prog1
.....
17375: symbol=a;  lookup in file=prog1  [ ELF ]
17375: symbol=a;  lookup in file=./W.so.1  [ ELF ]
17375: binding file=./W.so.1 to file=./W.so.1: symbol ‘a’
.....
17375: symbol=b;  lookup in file=prog1  [ ELF ]
17375: symbol=b;  lookup in file=./W.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./X.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./w.so.1  [ ELF ]
17375: binding file=./W.so.1 to file=./w.so.1: symbol ‘b’
.....
17375: symbol=a;  lookup in file=prog1  [ ELF ]
17375: symbol=a;  lookup in file=./W.so.1  [ ELF ]
17375: binding file=./X.so.1 to file=./W.so.1: symbol ‘a’
.....
17375: symbol=b;  lookup in file=prog1  [ ELF ]
17375: symbol=b;  lookup in file=./W.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./X.so.1  [ ELF ]
17375: symbol=b;  lookup in file=./w.so.1  [ ELF ]
17375: binding file=./X.so.1 to file=./w.so.1: symbol ‘b’
```

関数 a() または b() のいずれかを参照するたびに、アプリケーション prog1 で始まる関連シンボルを検索することになります。a() への各参照は、W.so.1 で検出されたシンボルの最初のインスタンスに結合します。b() への各参照は、w.so.1 で検出されたシンボルの最初のインスタンスに結合します。この例では、W.so.1 および w.so.1 の関数定義が、X.so.1 および x.so.1 の関数定義にどのように割り込むかを示します。割り込みの存在は、直接結合の使用を検討するときに重要な要素となります。割り込みについては、次のセクションで詳細に説明します。

この例は簡潔であり、関連する診断は容易に理解できます。しかし、ほとんどのアプリケーションでは、多数の動的コンポーネントから構成されているため、この例よりかなり複雑です。これらのコンポーネントは多くの場合、異なるソースベースから構築され、別々に配布されます。

複雑なプロセスの診断を解析することは、非常に困難な場合があります。動的オブジェクトのインタフェースを解析するには、ほかに lari(1) ユーティリティーを使用する方法があります。lari は、プロセスの結合情報と、各オブジェクトが提供するインタフェース定義を組み合わせて解析します。この情報によって、lari はプロセスのシンボル依存関係に関する興味深い情報を簡潔に伝えることができます。この情報は、直接結合に関連する割り込みを解析するときに、非常に有用です。

デフォルトでは、lari は興味深いと見なされた情報を伝えます。この情報はシンボル定義の複数のインスタンスから生じます。lari は prog1 に関する次の情報を表示します。

```
$ lari prog1
[2:2ES]: a(): ./W.so.1
[2:0]: a(): ./X.so.1
[2:2E]: b(): ./w.so.1
[2:0]: b(): ./x.so.1
```

この例では、prog1 から確立されたプロセスに、多重定義された a() と b() の 2 つのシンボルが含まれています。出力診断の最初の要素 (角括弧で囲まれた要素) は関連するシンボルを示しています。

最初の 10 進値は関連するシンボルのインスタンスの数を示しています。a() と b() の 2 つのインスタンスが存在します。2 番目の 10 進値は、このシンボルに対して解決された結合の数を示しています。W.so.1 のシンボル定義 a() は、この依存関係に対して 2 つの結合が確立されたことを表しています。同様に、w.so.1() のシンボル定義 b は、この依存関係に対して 2 つの結合が確立されたことを表しています。結合数に続く文字は結合を修飾します。文字「E」は、結合が外部オブジェクトから確立されたことを示しています。文字「S」は、結合が同じオブジェクトから確立されたことを示しています。

これらのコンポーネントから作成された LD_DEBUG、lari、およびプロセスの例は、次のセクションで直接結合についてさらに説明するために使用されます。

直接結合の有効化

直接結合を使用するオブジェクトは、シンボル参照と、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。

動的オブジェクトの直接結合情報はリンク編集時に記録されます。この情報は、該当オブジェクトのリンク編集時に指定された依存関係に対してのみ確立できます。-z defs オプションを使用すると、必要なすべての依存関係がリンク編集の一部として確実に提供されます。

直接結合を使用するオブジェクトは、直接結合を使用しないオブジェクトがあるプロセス内に存在できます。直接結合を使用しないオブジェクトは、デフォルトのシンボル検索モデルを使用します。

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、次のいずれかのリンク編集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、構築中のオブジェクト内の任意のシンボル参照とシンボル定義との間に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、リンカーのコマンド行の先頭に `-z lazyload` オプションを追加するのと同じです。この属性は108 ページの「動的依存関係の遅延読み込み」で紹介されました。
- `-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、依存関係の間で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、構築中のオブジェクト内のシンボル参照と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。このキーワードは、225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」で説明されています。

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。この環境変数を設定することで、プロセス内のすべてのシンボル結合は、デフォルトの検索モデルを使用して実行されます。

次のセクションでは、直接結合の各メカニズムの使用方法について説明します。

`-B direct` オプションの使用方法

`-B direct` オプションは、どの動的オブジェクトに対しても直接結合を有効にする、もっとも簡単なメカニズムを提供します。このオプションは、どの依存関係にも、また作成中のオブジェクト内に、直接結合を確立します。

前の例で使用されたコンポーネントから、直接結合されたオブジェクトである `W.so.2` を作成できます。

```
$ cc -o W.so.2 -G -Kpic W.c -R. -Bdirect w.so.1
$ cc -o prog2 -R. main.c W.so.2 X.so.1
```

直接結合の情報は、`W.so.2` 内にあるシンボル情報セクション (`.SUNW_syminfo`) に保持されます。このセクションは `elfdump(1)` を使用すると参照できます。

```
$ elfdump -y W.so.2
    [6]  DB          <self>         a
    [7]  DBL     [1] w.so.1         b
```

文字「DB」は、関連シンボルに対する直接結合が記録されたことを示しています。関数 a() は、格納するオブジェクト W.so.2 に結合されています。関数 b() は、依存関係 w.so.1 に直接結合されています。文字「L」は、依存関係 w.so.1 が遅延読み込みされることも示しています。

W.so.2 に確立された直接結合は、LD_DEBUG 環境を使用すると確認できます。detail トークンは追加情報を結合診断に追加します。W.so.2 の場合、このトークンは結合が直接的であることを示します。detail トークンは、結合アドレスに関する追加情報も提供します。単純化するため、このアドレス情報は、次の例から生成される出力から省略されました。

```
$ LD_DEBUG=symbols,bindings,detail prog2
.....
18452: symbol=a;  lookup in file=./W.so.2  [ ELF ]
18452: binding file=./W.so.2 to file=./W.so.2: symbol ‘a’  (direct)
18452: symbol=b;  lookup in file=./w.so.1  [ ELF ]
18452: binding file=./W.so.2 to file=./w.so.1: symbol ‘b’  (direct)
```

lari(1) ユーティリティーは直接結合情報を表示することもできます。

```
$ lari prog2
[2:2ESD]: a(): ./W.so.2
[2:0]: a(): ./X.so.1
[2:2ED]: b(): ./w.so.1
[2:0]: b(): ./x.so.1
```

文字「D」は、W.so.2 によって定義された関数 a() が直接結合されていることを示しています。同様に、w.so.1 で定義された関数 b() は直接結合されています。

注 – 関数 a() で W.so.2 を W.so.2 に直接結合すると、W.so.2 を構築するときに -B symbolic オプションを使用した場合と類似の効果が生まれます。ただし、-B symbolic オプションでは、内部的に解決できる a() などの参照がリンク編集時に終了処理されることになります。このシンボルの解決では、実行時に解決する結合が残りません。

-B symbolic 結合とは異なり、-B direct では実行時に解決する結合が残ります。このため、この結合は明示的な割り込みによってオーバーライドしたり、環境変数 LD_NODIRECT を非ヌル値に設定することによって無効化したりすることができます。

複雑なオブジェクトを読み込むときに発生する実行時再配置のオーバーヘッドを削減するために、多くの場合、シンボリック結合が採用されてきました。直接結合は、まったく同じシンボル結合を確立するために使用できます。ただし、各直接結合を作成するために実行時再配置が引き続き必要です。直接結合にはシンボリック結合より多くのオーバーヘッドが発生しますが、柔軟性は向上します。

-z direct オプションの使用方法

-z direct オプションを使用すると、リンク編集コマンド行のオプションに従うどのような依存関係に対しても直接結合を確立できるメカニズムが提供されます。-B direct オプションとは異なり、構築中のオブジェクト内に、直接結合は確立されません。

このオプションは、割り込まれるように設計されたオブジェクトを作成する場合に適しています。たとえば、共有オブジェクトは、いくつかのデフォルト (フォールバック) インタフェースを持つように設計される場合があります。アプリケーションは、アプリケーション定義を実行時に結合するために、自由にこれらのインタフェースを独自に定義できます。アプリケーションが共有オブジェクトのインタフェースに割り込めるようにするには、-B direct オプションではなく、-z direct オプションを使用して共有オブジェクトを作成します。

-z direct オプションは、1 つまたは複数の依存関係に対する直接結合を選択する場合にも有効です。-z nodirect オプションを使用すると、リンク編集で指定される依存関係の間で、直接結合の使用を切り替えることができます。

前の例で使用されたコンポーネントから、直接結合されたオブジェクトである X.so.2 を作成できます。

```
$ cc -o X.so.2 -G -Kpic X.c -R. -zdirect x.so.1
$ cc -o prog3 -R. main.c W.so.2 X.so.2
```

直接結合情報は、elfdump(1) で表示できます。

```
$ elfdump -y X.so.2
    [6]  D           <self>         a
    [7]  DB      [1] x.so.1         b
```

関数 b() は、依存関係 x.so.1 に直接結合されています。関数 a() は、オブジェクト X.so.2 と潜在的な直接結合 (「D」) を持つように定義されていますが、直接結合は確立されません。

実行時の結合を確認するために、LD_DEBUG 環境変数を使用できます。

```
$ LD_DEBUG=symbols,bindings,detail prog3
.....
06177: symbol=a;  lookup in file=prog3  [ ELF ]
06177: symbol=a;  lookup in file=./W.so.2  [ ELF ]
06177: binding file=./X.so.2 to file=./W.so.2: symbol ‘a’
06177: symbol=b;  lookup in file=./x.so.1  [ ELF ]
06177: binding file=./X.so.2 to file=./x.so.1: symbol ‘b’  (direct)
```

lari(1) ユーティリティーは直接結合情報を表示することもできます。

```
$ lari prog3
[2:2ESD]: a(): ./W.so.2
```

```
[2:0]: a(): ./X.so.2
[2:1ED]: b(): ./w.so.1
[2:1ED]: b(): ./x.so.1
```

W.so.2 で定義された関数 a() は、X.so.2 で作成されたデフォルトのシンボル参照を引き続き満たします。しかし、x.so.1 で定義された関数 b() は、X.so.2 で作成された参照から直接結合されています。

DIRECT mapfile キーワードの使用方法

DIRECT mapfile キーワードは、個々のシンボルに対して直接結合を確立する手段を提供します。このメカニズムは、特別なリンク編集のシナリオを対象としています。

前の例で使用されたコンポーネントから、関数 main() は外部関数 W() と X() を参照します。これらの関数の結合はデフォルトの検索モデルに従います。

```
$ LD_DEBUG=symbols,bindings prog3
.....
18754: symbol=W;  lookup in file=prog3  [ ELF ]
18754: symbol=W;  lookup in file=./W.so.2  [ ELF ]
18754: binding file=prog3 to file=./W.so.2: symbol `W'
.....
18754: symbol=X;  lookup in file=prog3  [ ELF ]
18754: symbol=X;  lookup in file=./W.so.2  [ ELF ]
18754: symbol=X;  lookup in file=./X.so.2  [ ELF ]
18754: binding file=prog3 to file=./X.so.2: symbol `X'
```

直接結合が関数 W() と X() に対して確立されるように、prog3 は DIRECT mapfile キーワードを使用して再構築できます。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                W       { FLAGS = EXTERN DIRECT };
                X       { FLAGS = EXTERN DIRECT };
};
$ cc -o prog4 -R. main.c W.so.2 X.so.2 -Mmapfile
```

実行時の結合を確認するために、LD_DEBUG 環境変数を使用できます。

```
$ LD_DEBUG=symbols,bindings,detail prog4
.....
23432: symbol=W;  lookup in file=./W.so.2  [ ELF ]
23432: binding file=prog4 to file=./W.so.2: symbol `W'  (direct)
23432: symbol=X;  lookup in file=./X.so.2  [ ELF ]
23432: binding file=prog4 to file=./x.so.2: symbol `X'  (direct)
```

lari(1) ユーティリティーは直接結合情報を表示することもできます。しかしこの場合、関数 W() と X() は多重定義されていません。このためデフォルトでは、lari にはこれらの関数が興味深いものであることはわかりません。すべてのシンボル情報を表示するために、-a オプションを使用する必要があります。

```
$ lari -a prog4
....
[1:1ED]: W(): ./W.so.2
.....
[2:1ED]: X(): ./X.so.2
.....
```

注 – W.so.2 と X.so.1 に対する同じ直接結合は、-B direct オプションまたは -z direct オプションを使用して prog4 を構築することで作成できます。この例の唯一の目的は、mapfile キーワードがどのように使用できるかを示すことです。

直接結合と割り込み

割り込みが発生する可能性があるのは、あるシンボルの複数のインスタンス (名前は同じ) が、プロセスに読み込まれた別々の動的オブジェクトに存在する場合です。デフォルト検索モデルでは、シンボル参照は、読み込まれた一連の依存関係で検出された最初の定義に結合されます。この場合、最初のシンボルが、同じ名前の別のシンボルに「割り込む」と言います。

直接結合は暗黙の割り込みを回避できます。参照と関連する依存関係の中で直接結合された参照を検索するとき、割り込みを有効にするデフォルトのシンボル検索モデルはバイパスされます。直接結合の環境では、同じ名前を持つ、シンボルの別々の定義に対して結合を確立できます。

同じ名前を持つ、シンボルの別々の定義に結合できるという点は、直接結合の有用な機能の 1 つです。ただし、アプリケーションが割り込みのインスタンスに依存していると、直接結合を使用した場合に、アプリケーションの期待動作が阻害される可能性があります。既存のアプリケーションで直接結合の使用を決める前に、割り込みが存在するかどうかを判断するために、アプリケーションを分析する必要があります。

アプリケーション内で割り込みが可能かどうかを判断するには、lari(1) を使用します。デフォルトでは、lari は興味深い情報を表示します。この情報は、シンボル定義の複数のインスタンスから生じているため、割り込みに繋がる可能性があります。

割り込みは、シンボルの 1 つのインスタンスが結合されたときにのみ発生します。シンボルの複数のインスタンスが lari によって表示された場合には、割り込みが発生しない可能性があります。別の複数のインスタンスシンボルが存在する場合がありますが、参照されない可能性があります。これらの未参照のシンボルは引き続き割り込みの候補であり、今後のコード開発でこれらのシンボルが参照される可能性があります。多重定義されたシンボルのすべてのインスタンスについて、直接結合の使用を検討するときに分析する必要があります。

同じ名前のシンボルのインスタンスが複数存在する場合で、特に割り込みが確認された場合は、次の処理のいずれかを実行する必要があります。

- シンボルのインスタンスをローカライズして名前空間の衝突を取り除きます。
- 複数のインスタンスを削除して 1 つのシンボル定義を残します。
- 割り込み要件を明示的に定義します。
- シンボルが直接結合されないように、割り込みが発生する可能性があるシンボルを特定します。

次のセクションでは、これらの処理を詳細に説明します。

シンボルインスタンスのローカライズ

別々の実装を提供する、同じ名前の多重定義シンボルは、偶発的な割り込みを回避するために分離する必要があります。オブジェクトによってエクスポートされたインタフェースからシンボルを削除するもっとも簡単な方法は、シンボルを局所に限定することです。シンボルを局所に限定するには、シンボル「static」を定義することで、またはコンパイラが提供するシンボル属性を使用することでも、実現できます。

シンボルは、リンカーと mapfile を使用しても、局所に限定できます。次の例では、local スコープ指令を使用して大域関数 error() を局所シンボルに限定する mapfile を示します。

```
$ cc -o A.so.1 -G -Kpic error.c a.c b.c ...
$ elfdump -sN.symtab A.so.1 | fgrep error
    [36]  0x000002d0 0x00000014  FUNC GLOB  D    0 .text      error
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        local:
                error;
};
$ cc -o A.so.2 -G -Kpic -M mapfile error.c a.c b.c ...
$ elfdump -sN.symtab A.so.2 | fgrep error
    [24]  0x000002c8 0x00000014  FUNC LOCL  H    0 .text      error
```

mapfile の明示的な定義を使用して個々のシンボルを局所に限定することはできますが、シンボルのバージョン管理を介して全体のインタフェースファミリを定義することをお勧めします。第 9 章「インタフェースおよびバージョン管理」を参照してください。

バージョン管理は、主に共有オブジェクトからエクスポートされたインタフェースを特定するために採用された、有用な技術です。同じように、動的実行可能ファイルをバージョン管理すると、エクスポートされたインタフェースを定義できます。動的実行可能ファイルに必要なことは、結合するオブジェクトの依存関係に対して利用できるようにしなければならないインタフェースをエクスポートするだけです。多くの場合、動的実行可能ファイルに追加するコードは、インタフェースをエクスポートする必要がありません。

動的実行可能ファイルからエクスポートされたインタフェースを削除するには、コンパイラドライバによって確立されたすべてのシンボル定義を考慮する必要があります。これらの定義は、コンパイラドライバが最後のリンク編集に追加する補助ファイルが元になっています。35 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。

次の例の mapfile では、コンパイラドライバが確立する可能性がある、共通のシンボル定義セットをエクスポートする一方で、ほかのすべての大域定義を局所に限定します。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                __Argv;
                __environ_lock;
                _environ;
                _lib_version;
                environ;
        local:
                *;
};
```

コンパイラドライバが確立するシンボル定義を決定する必要があります。動的実行可能ファイル内で使用されるすべての定義は大域のままにします。

動的実行可能ファイルからエクスポートされたインタフェースを削除することで、オブジェクトの依存関係が変わっていくにつれて今後発生する可能性がある割り込みの問題から、この実行可能ファイルが保護されます。

同じ名前の多重定義シンボルの削除

同じ名前の多重定義シンボルは、シンボルに関連付けられた実装が状態を保持していると、直接結合された環境内で問題となる可能性があります。典型的にはデータシンボルが違反していますが、状態を保持する関数も問題になる場合があります。

直接結合された環境では、同じシンボルの複数のインスタンスに結合できます。このため、もともとプロセス内ではインスタンスが 1 つであることを意図していた別々の状態変数を、結合している別々のインスタンスが操作できます。

たとえば、2 つの共有オブジェクトが同じデータ項目 errval を含むものとします。また、2 つの関数 action() と inspect() が別々の共有オブジェクトに存在するものとします。これらの関数は、値 errval をそれぞれ読み書きします。

デフォルトの検索モデルの場合、errval の 1 つの定義がその他の定義に割り込みます。action() と inspect() の両方の関数は、errval の同じインスタンスに結合されます。このため、action() によって errval にエラーコードが書き込まれた場合、inspect() がこのコードを読み込んで、このエラー条件に従って動作します。

しかし、action() および inspect() を含むオブジェクトが、errval をそれぞれで定義した別々の依存関係に結合されたものと仮定します。直接結合された環境では、これらの関数は errval の別々の定義に結合されます。エラーコードは action() によって errval の 1 つのインスタンスに書き込まれ、その一方で inspect() は errval のもう一方の初期化されていない定義を読み込みます。その結果、inspect() では、処理対象となるエラー条件を検出しません。

一般的に、データシンボルの複数のインスタンスは、シンボルがヘッダーで宣言された場合に発生します。

```
int bar;
```

このデータ宣言では、ヘッダーを含むコンパイル単位ごとに、データ項目が作成されることになります。その結果生じる一時的なデータ項目によって、シンボルの複数のインスタンスが別々の動的オブジェクトに定義される場合があります。

しかし、データ項目を明示的に外部として定義することで、データ項目への参照は、ヘッダーを含むコンパイル単位ごとに作成されます。

```
extern int bar;
```

その結果、これらの参照を実行時に 1 つのデータインスタンスに解決できます。

シンボル実装を削除するときに、そのインタフェースの保持が必要な場合があります。同じインタフェースの複数のインスタンスは、既存のすべてのインタフェースを保持しながら、1 つの実装にすることができます。このモデルは、FILTER mapfile キーワードを使用して個々のシンボルフィルタを作成することで実現できます。このキーワードは、225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」で説明されています。

個々のシンボルフィルタを作成することが有効なのは、シンボルの実装が削除されたオブジェクト内で、依存関係がそのシンボルの検出を期待している場合です。

たとえば、関数 error() が A.so.1 と B.so.1 の 2 つの共有オブジェクトに存在するものとします。シンボルの重複を削除するには、A.so.1 から実装を削除する必要があります。しかし、別の依存関係が A.so.1 から得られる error() に依存しています。次の例は、A.so.1 での error() の定義を示しています。mapfile を使用すると、B.so.1 に指定されたこのシンボルのフィルタを残したまま、error() の実装を削除できます。

```
$ cc -o A.so.1 -G -Kpic error.c a.c b.c ...
$ elfdump -sN.dynsym A.so.1 | fgrep error
    [3]  0x00000300 0x00000014  FUNC GLOB  D    0 .text      error
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                error { TYPE=FUNCTION; FILTER=B.so.1 };
```

```
};
$ cc -o A.so.2 -G -Kpic -M mapfile a.c b.c ...
$ elfdump -sN.dynsym A.so.2 | fgrep error
    [3]  0x00000000 0x00000000  FUNC GLOB  D    0 ABS        error
$ elfdump -y A.so.2 | fgrep error
    [3]  F       [0] B.so.1         error
```

関数 error() は大域であり、A.so.2 のエクスポートされたインタフェースのままです。しかし、このシンボルへの実行時結合はフィルティー B.so.1 になります。文字「F」はこのシンボルのフィルタ特性を表しています。

既存のインタフェースを保持したまま 1 つの実装にするこのモデルは、複数の Oracle Solaris ライブラリで使用されています。たとえば、以前に libc.so.1 に定義された多くの数値演算インタフェースは、現在は libm.so.2 の推奨される関数実装を指しています。

明示的な割り込みの定義

デフォルトの検索モデルでは、同じ名前のシンボルのインスタンスが、同じ名前の後のインスタンスに割り込む可能性があります。明示的なラベル付けをしなくても割り込みは発生するため、1 つのシンボル定義がすべての参照から結合されます。この暗黙の割り込みはシンボル検索の結果として起こるもので、実行時リンカーに指定された明示的な命令によるものではありません。この暗黙の割り込みは直接結合によって回避できます。

直接結合は、関連するシンボル定義に対する直接のシンボル参照を解決する働きをしますが、明示的な割り込みは直接結合の検索の前に処理されます。このため、直接結合の環境であっても、割り込み処理を設計でき、直接結合の関連付けに割り込みを期待できます。割り込み処理は、次の方法を利用して明示的に定義できます。

- LD_PRELOAD 環境変数を使用。
- リンカーの -z interpose オプションを使用。
- INTERPOSE mapfile キーワードを使用。
- singleton シンボル定義の結果として。

LD_PRELOAD 環境変数の割り込み機能と -z interpose オプションが、しばらく前から利用可能です。103 ページの「実行時割り込み」を参照してください。これらのオブジェクトは割り込み処理であると明示的に定義されているため、実行時リンカーは直接結合を処理する前にこれらのオブジェクトを検査します。

共有オブジェクトに対して確立された割り込みは、その動的オブジェクトのすべてのインタフェースに適用されます。LD_PRELOAD 環境変数を使用してオブジェクトが読み込まれたときに、このオブジェクトの割り込みが確立されます。オブジェクトの割り込みは、-z interpose オプションで作成されたオブジェクトが読み込まれたときにも確立されます。このオブジェクトモデルが重要なのは、特別なハンドル RTLD_NEXT を持つ dlsym(3C) などの技術が使用されたときです。割り込むオブジェクトは、次のオブジェクトの一貫したビューを常に持つべきです。

動的実行可能ファイルは、INTERPOSE mapfile キーワードを使用して個々の割り込みシンボルを定義できるという点で柔軟性が増します。動的実行可能ファイルはプロセスにロードされる最初のオブジェクトであるため、次のオブジェクトの実行可能ファイルのビューは常に一貫したものになります。

次の例は、exit() 関数に明示的に割り込みを行うアプリケーションを示しています。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                exit    { FLAGS = INTERPOSE };
};
$ cc -o prog -M mapfile exit.c a.c b.c ...
$ elfdump -y prog | fgrep exit
    [6]  DI          <self>          exit
```

文字「I」は、このシンボルの割り込み特性を表しています。おそらく、この exit() 関数の実装では、システム関数 _exit() が直接参照されるか、または RTLD_NEXT ハンドルを持つ dlsym() を使ってシステム関数 exit() に達するまで呼び出しが続けられると考えられます。

最初は、-z interpose オプションを使ってこのオブジェクトを特定することを考えてはどうでしょうか。ただし、この手法は、アプリケーションによってエクスポートされたすべてのインタフェースが割り込み処理として機能するため、かなりの重量になります。より適切な代替方法は、-z interpose オプションを使用するとともに、割り込み処理を除く、アプリケーションで提供されたすべてシンボルをローカライズすることです。

しかし、INTERPOSE mapfile キーワードを使用すると、柔軟性が向上します。このキーワードを使用すると、アプリケーションでいくつかのインタフェースをエクスポートしながら、それらのインタフェースから割り込み処理として機能させるものを選択できます。

STV_SINGLETON 可視性が割り当てられているシンボルには、効果的な割り込みの形態が備わっています。表 12–20 を参照してください。これらのシンボルは、コンパイルシステムによって、プロセス内の多数のオブジェクトで多重インスタンス化状態になる可能性がある実装に割り当てることができます。singleton シンボルへのすべての参照は、プロセス内で最初に定義された singleton に結合されます。

シンボルの直接結合の回避

直接結合は明示的な割り込みでオーバーライドできます。177 ページの「明示的な割り込みの定義」を参照してください。しかし、明示的な割り込みの確立を制御できない場合があります。

たとえば、直接結合を使用する共有オブジェクトファミリを配布するとします。また、このファミリの共有オブジェクトによって提供されるシンボルに顧客が割り込むことがわかっています。これらの顧客が自身の割り込み要件を明示的に定義しないと、直接結合を使用する共有オブジェクトの再配布によって、その割り込みが損なわれる場合があります。

ユーザーが独自の割り込みルーチンを提供することを期待して、いくつかのデフォルトインタフェースを提供する共有オブジェクトを設計することもできます。

既存のアプリケーションが混乱を起こさないようにするため、1 つまたは複数のインタフェースに明示的に直接結合しない共有オブジェクトを配布できます。

動的オブジェクトへの直接結合は、次のいずれかのオプションを使用すると回避できます。

- `-B nodirect` オプションを使用する。このオプションを使用すると、作成中のオブジェクトが提供するインタフェースに直接結合されません。
- `NODIRECT mapfile` キーワードを使用する。このキーワードを使用すると、個別のシンボルへの直接結合を避けられます。このキーワードは、225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」で説明されています。
- `singleton` シンボル定義の結果として。

`nodirect` とラベルが付けられたインタフェースは、外部オブジェクトから直接結合できません。また、`nodirect` とラベルが付けられたインタフェースは、同じオブジェクト内から直接結合できません。

次のセクションでは、直接結合回避メカニズムの各使用方法について説明します。

`-B nodirect` オプションの使用方法

`-B nodirect` オプションは、どの動的オブジェクトからの直接結合も回避する、もっとも簡単なメカニズムを提供します。このオプションはほかのオブジェクトから、また作成中のオブジェクト内からの直接結合を回避します。

次のコンポーネントを使用して、3 つの共有オブジェクト、`A.so.1`、`O.so.1`、および`X.so.1` を作成します。`-B nodirect` オプションを使用すると、`A.so.1` は `O.so.1` に直接結合されません。しかし、`O.so.1` は、`-z direct` オプションを使用して `X.so.1` への直接結合を引き続き確立できます。

```
$ cat a.c
extern int o(), p(), x(), y();

int a() { return (o() + p() - x() - y()); }

$ cat o.c
extern int x(), y();

int o() { return (x()); }
int p() { return (y()); }

$ cat x.c
int x() { return (1); }
int y() { return (2); }

$ cc -o X.so.1 -G -Kpic x.c
$ cc -o O.so.1 -G -Kpic o.c -Bnodirect -zdirect -R. X.so.1
$ cc -o A.so.1 -G -Kpic a.c -Bdirect -R. O.so.1 X.so.1
```

A.so.1 および O.so.1 のシンボル情報は、elfdump(1) で参照できます。

```
$ elfdump -y A.so.1
    [1]  DBL      [3] X.so.1            x
    [5]  DBL      [3] X.so.1            y
    [6]  DL       [1] O.so.1            o
    [9]  DL       [1] O.so.1            p
$ elfdump -y O.so.1
    [3]  DB       [0] X.so.1            x
    [4]  DB       [0] X.so.1            y
    [6]  N                              o
    [7]  N                              p
```

文字「N」は、関数 o() および p() に直接結合が許可されていないことを示しています。A.so.1 が -B direct オプションによって直接結合を要求したとしても、関数 o() および p() に直接結合は確立されません。O.so.1 は引き続き、-z direct オプションを使用して依存関係 X.so.1 への直接結合を要求できます。

Oracle Solaris ライブラリ libproc.so.1 は、-B nodirect オプションを使って作成されます。このライブラリのユーザーは、多くの libproc 関数に対して独自のコールバックインタフェースを用意することが求められています。libproc の依存関係から libproc 関数への参照は、いずれかのユーザー定義 (このような定義がある場合) に結合する必要があります。

NODIRECT mapfile キーワードの使用方法

NODIRECT mapfile キーワードは、個々のシンボルへの直接結合を回避する手段を提供します。このキーワードは、-B nodirect オプションよりも詳細に直接結合の回避を制御できます。

前の例で使用されたコンポーネントから、O.so.2 は、関数 o() に直接接続しないように作成できます。

```
$ cat mapfile
$mapfile_version 2
SYMBOL_SCOPE {
        global:
                o       { FLAGS = NODIRECT };
};
$ cc -o O.so.2 -G -Kpic o.c -Mmapfile -zdirect -R. X.so.1
$ cc -o A.so.2 -G -Kpic a.c -Bdirect -R. O.so.2 X.so.1
```

A.so.2 および O.so.2 のシンボル情報は、elfdump(1) で参照できます。

```
$ elfdump -y A.so.2
    [1]  DBL      [3] X.so.1            x
    [5]  DBL      [3] X.so.1            y
    [6]  DL       [1] O.so.1            o
    [9]  DBL      [1] O.so.1            p
$ elfdump -y O.so.1
    [3]  DB       [0] X.so.1            x
    [4]  DB       [0] X.so.1            y
    [6]  N                              o
    [7]  D            <self>            p
```

O.so.1 は、関数 o() が直接結合できないことを宣言するだけです。このため A.so.2 は、O.so.1 内の関数 p() に直接結合できます。

Oracle Solaris ライブラリ内にあるいくつかの個別インタフェースは、直接結合できないように定義されています。データ項目 errno はその 1 例です。このデータ項目は libc.so.1 で定義されています。このデータ項目は、ヘッダーファイル stdio.h をインクルードすることで参照できます。しかし、多くのアプリケーションでは独自の errno を定義するように指導されているのが一般的でした。libc.so.1 で定義された errno に直接結合するシステムライブラリファミリが配布された場合、これらのアプリケーションの機能が損なわれることになります。

直接結合しないように定義された別のインタフェースファミリには、malloc(3C) ファミリがあります。malloc() ファミリは、ユーザーアプリケーション内に頻繁に実装される別のインタフェースセットです。これらのユーザー実装はシステム定義に割り込むことを意図しています。

注 – Oracle Solaris OS には、代替の malloc() 実装を提供する、さまざまなシステム割り込みライブラリが備わっています。また各実装は、プロセス内で使用される唯一の実装であることが期待されます。すべての malloc() 割り込みライブラリは -z interpose オプションを指定して作成されています。libc.so.1 内の malloc() ファミリは、直接結合しないようにラベル付けされているので、このオプションは必ずしも必要ではありません。

しかし、割り込み処理を作成するための先例とするため、割り込みライブラリは -z interpose を指定して作成されました。この明示的な割り込みには、libc.so.1 内に確立された直接結合の回避定義に対する不都合な相互作用はありません。

STV_SINGLETON 可視性が割り当てられているシンボルは、直接結合できません。表 12–20 を参照してください。これらのシンボルは、コンパイルシステムによって、プロセス内の多数のオブジェクトで多重インスタンス化状態になる可能性がある実装に割り当てることができます。singleton シンボルへのすべての参照は、プロセス内で最初に定義された singleton に結合されます。

◆ ◆ ◆ 第 7 章

7 システムのパフォーマンスを最適化するオブジェクトの構築

動的実行可能ファイルと共有オブジェクトについては、実行時の処理でこれらのオブジェクトを使用するプロセスを確立する必要があります。プロセスの複数のインスタンスが常にアクティブになることができ、同時に複数のプロセスが共有オブジェクトを使用できます。動的オブジェクトの構築は、実行時の初期設定、プロセス間でオブジェクトが共有される可能性、およびシステム全体のパフォーマンスに影響を与えます。

以降のセクションでは、動的オブジェクトの実行時の初期設定と処理、およびそれらの実行時パフォーマンスに影響を与える要因 (テキストのサイズと純度、再配置のオーバーヘッドなど) について説明します。

elfdump を使用したファイルの解析

ELF ファイルの内容を解析するために、標準の UNIX ユーティリティー dump(1)、nm(1)、size(1) など、さまざまなツールが使用できます。Oracle Solaris では、これらのツールは大部分が elfdump(1) に置き換えられました。

以降のセクションで説明するさまざまなパフォーマンスの問題を調べるには、eldump を使用して ELF オブジェクトの内容を診断すると役に立つことがあります。

ELF 形式では、データがセクションにまとめられます。次に、セクションはセグメントと呼ばれる単位に割り当てられます。セグメントには、ファイルの一部がどのようにメモリーにマップされるかが記述されます。mmap(2) 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、elfdump(1) コマンドを使用して PT_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:     0x53c       p_memsz:    0x53c
    p_offset:     0           p_align:    0x10000

Program Header[1]:
    p_vaddr:      0x1053c     p_flags:    [ PF_X PF_W PF_R ]
    p_paddr:      0           p_type:     [ PT_LOAD ]
    p_filesz:     0x114       p_memsz:    0x13c
    p_offset:     0x53c       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 と 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。

プログラマがファイルについて考えるとき、多くの場合、そのコード内の関数とデータ要素を定義するシンボルの点から考えます。これらのシンボルは、elfdump の -s オプションを使用すると表示できます。

```
$ elfdump -sN.symtab libfoo.so.1

Symbol Table Section:  .symtab
     index    value      size      type bind oth ver shndx          name
      ....
      [36]  0x00010628 0x00000028  OBJT GLOB  D    0 .data          data
      ....
      [38]  0x00010650 0x00000028  OBJT GLOB  D    0 .bss           bss
      ....
      [40]  0x00000520 0x0000000c  FUNC GLOB  D    0 .init          _init
      ....
      [44]  0x00000508 0x00000014  FUNC GLOB  D    0 .text          foo
      ....
      [46]  0x0000052c 0x0000000c  FUNC GLOB  D    0 .fini          _fini
```

elfdump で表示されるシンボルテーブル情報には、シンボルに関連するセクションが含まれます。elfdump の -c オプションを使用すると、これらのセクションに関する情報を表示できます。

```
$ elfdump -c libfoo.so.1
....
Section Header[6]:  sh_name: .text
    sh_addr:      0x4f8           sh_flags:   [ SHF_ALLOC SHF_EXECINSTR ]
    sh_size:      0x28            sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x4f8           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x8

Section Header[7]:  sh_name: .init
    sh_addr:      0x520           sh_flags:   [ SHF_ALLOC SHF_EXECINSTR ]
    sh_size:      0xc             sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x520           sh_entsize: 0
    sh_link:      0               sh_info:    0
```

```
    sh_addralign: 0x4

Section Header[8]:  sh_name: .fini
    sh_addr:      0x52c           sh_flags:   [ SHF_ALLOC SHF_EXECINSTR ]
    sh_size:      0xc             sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x52c           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x4
....
Section Header[12]:  sh_name: .data
    sh_addr:      0x10628         sh_flags:   [ SHF_WRITE SHF_ALLOC ]
    sh_size:      0x28            sh_type:    [ SHT_PROGBITS ]
    sh_offset:    0x628           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x4
....
Section Header[14]:  sh_name: .bss
    sh_addr:      0x10650         sh_flags:   [ SHF_WRITE SHF_ALLOC ]
    sh_size:      0x28            sh_type:    [ SHT_NOBITS ]
    sh_offset:    0x650           sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x4
....
```

上記の例の elfdump(1) からの出力は、関数 _init、foo、および _fini と、セクション .init、.text、および .fini との関連を示します。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。

基本システム

アプリケーションは、動的実行可能ファイルと 1 つ以上の共有オブジェクトの依存関係から構築されます。実行時には、動的実行可能ファイルと共有オブジェクトの読み込み可能な内容の全体がそのプロセスの仮想アドレス空間にマップされます。各プロセスは、最初にメモリー内の動的実行可能ファイルと共有オブジェクトの単一コピーを参照します。

動的オブジェクト内の再配置が処理され、シンボリック参照が該当する定義に結合されます。これにより、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。

オブジェクトの動的リンクのベースとなるメモリー管理スキームでは、プロセス間のメモリー共有がページの粒度で行われます。メモリーページは、実行時に変更されていなければプロセス間で共有できます。プロセスが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オブジェクトのページに書き込むと、そのページの専用コピーが生成されます。この専用コピーは、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。

メモリーにマップされた動的オブジェクトのセグメントは、読み取り専用のテキストセグメントと読み書き可能なデータセグメントに分類されます。183 ページの「elfdump を使用したファイルの解析」 ファイルからこの情報を取得する方法については、Analyzing Files With elfdumpを参照してください。動的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。この分割により、動的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。次のセクションでは、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。108 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、一般に位置依存であり、メモリー内の固定アドレスに結合されます。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、424 ページの「大域オフセットテーブル (プロセッサ固有)」と 425 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できます。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。

`-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。424 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

入力再配置可能オブジェクトファイルの関数やデータが構築中のオブジェクトによって使用されない場合は、この対象物を取り込んでも無駄です。この使用されない対象物によってオブジェクトが必要以上に肥大し、実行時にそのオブジェクトを使用するときのオーバーヘッドが増加します。

使用されない共有オブジェクト依存関係への参照も無駄です。特に遅延読み込みがない場合は、これらの参照によって実行時にこれら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。

未使用のセクション、未使用の再配置可能オブジェクトファイル、および未使用の共有オブジェクト依存関係は、リンク編集時にリンカーの -D unused デバッグオプションを使用して診断できます。

未使用のファイルと依存関係は、-z guidance オプションを使用したときにも診断されます。

未使用のセクション、未使用のファイル、および未使用の依存関係は、リンク編集から削除すべきです。この削除によって、リンク編集のコストが減少し、構築中のオブジェクトを実行時に使用するコストも減少します。ただし、これらの項目の削除に問題がある場合は、-z discard-unused オプションを使用すると、未使用の対象物を構築中のオブジェクトから破棄できます。

未使用セクションの削除

入力再配置可能オブジェクトファイルに含まれる ELF セクションは、3 つの条件が真である場合に未使用と判定されます。

- そのセクションは、大域シンボルを提供しません。
- そのセクションは、割り当て可能なセグメントで使用されます。
- そのセクションは、リンク編集で使用される (任意のオブジェクトに含まれる) ほかの使用済みセクションによって参照されません。

未使用セクションをリンク編集から破棄するには、-z discard-unused=sections オプションを使用します。

動的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り、未使用セクションを診断して破棄するリンカーの機能を向上させることができます。第 9 章「インタフェースおよびバージョン管理」を参照してください。インタフェースを定義すると、インタフェースの一部として定義されていない大域シンボルが局所に限定されます。ほかのオブジェクトから参照されない限定されたシンボルは、破棄の対象として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、リンカーはこれらの項目を個別に破棄できます。このセクション改良は、-xF コンパイラオプションを使用して実現できます。

未使用ファイルの削除

入力再配置可能オブジェクトファイルは、再配置可能オブジェクトによって提供されるすべての割り当て可能セクションが未使用である場合に、未使用と判定されます。

未使用ファイルは、-z guidance オプションで診断され、-z discard-unused=files オプションを使用してリンク編集から破棄できます。

-z discard-unused オプションは、-z guidance の処理を補完するために、未使用セクションと未使用ファイルを独立して制御します。-z guidance では、未使用と判定されるファイルが識別されます。未使用ファイルは、多くの場合、リンク編集から簡

単に削除できます。しかし、未使用と判定されるセクションは -z guidance の処理では識別されません。未使用セクションは、削除するためにより多くの調査と作業が必要であり、ユーザーが制御できないコンパイラ動作の結果として生じる可能性があります。

-z guidance オプションとともに -z discard-unused=sections オプションを使用すると、未使用セクションは自動的に削除されますが、未使用ファイルはリンク編集から削除する対象として識別されます。

未使用の依存関係の削除

明示的な共有オブジェクト依存関係は、コマンド行でパス名または (より一般的には) -l オプションを使用して定義されます。明示的な依存関係には、コンパイラドライバ (-lc など) によって提供された可能性があるものも含まれます。明示的な依存関係は、2 つの条件が真である場合に未使用と判定されます。

- その依存関係によって提供される大域シンボルが、構築中のオブジェクトから参照されていません。
- その依存関係は、暗黙の依存関係の要件を補いません。

未使用の依存関係は、-z guidance オプションで診断され、-z discard-unused=dependencies オプションを使用してリンク編集から破棄できます。

暗黙の依存関係は、明示的な依存関係の依存関係です。暗黙の依存関係は、すべてのシンボル解決の封じ込めを完成させるため、リンク編集の一部として処理できます。このシンボルの封じ込めにより、構築中のオブジェクトが自己完結型であり、未参照のシンボルが残っていないことが保証されます。

すべての動的オブジェクトは、必要な依存関係を定義すべきです。この要件は、動的実行可能ファイルを構築するときはデフォルトで適用されますが、共有オブジェクトを構築するときは -z defs オプションを使用した場合にのみ適用されます。残念ながら共有オブジェクトにそのオブジェクトが必要とする依存関係が定義されていない場合は、それらのオブジェクトのために明示的な依存関係を提供する必要がある可能性があります。このような依存関係は、補完依存関係と呼ばれます。補完依存関係は、計画的に -z defs オプションを使用してすべての動的オブジェクトを構築することで不要になります。

自身の依存関係を定義しない動的オブジェクトは修正するべきです。ただし、これらのオブジェクトが有効なプロセスを作成するために補完依存関係を必要とする場合があるため、-z discard-unused=dependencies オプションでは未使用の補完依存関係は削除されません。

-z ignore および -z record オプションは、-z discard-unused=dependencies オプションと組み合わせて使用できる定位置オプションです。これらの定位置オプションは、対象となるオブジェクトを選択して、破棄機能をオンまたはオフにします。

共有可能性の最大化

185 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。

次のセクションでは、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

データ項目用の永続ストレージは、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。永続ストレージ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、永続ストレージ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

100 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。101 ページの「再配置シンボルの検索」と195 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、60 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。60 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。第 6 章「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、196 ページの「コピー再配置」セクションで説明します。このセクションでは、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、リンカーエディタによって実行可能ファイルまたは共有オブジェクトが作成するときに、プロシージャーリンクテーブルの再配置を除くすべての再配置が .SUNW_reloc という名前の 1 つの共通セクションに配置されます。

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、186 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
$ elfdump -sN.dynsym prog | grep ' sys_'
      [24]  0x00021240 0x00000260  OBJT GLOB  D    1 .bss           sys_errlist
      [39]  0x00021230 0x00000004  OBJT GLOB  D    1 .bss           sys_nerr
$ elfdump -c prog
....
Section Header[19]:  sh_name: .bss
    sh_addr:      0x21230         sh_flags:   [ SHF_WRITE SHF_ALLOC ]
    sh_size:      0x270           sh_type:    [ SHT_NOBITS ]
    sh_offset:    0x1230          sh_entsize: 0
    sh_link:      0               sh_info:    0
    sh_addralign: 0x8
....
$ elfdump -r prog

Relocation Section:  .SUNW_reloc
    type                        offset     addend  section        symbol
....
  R_SPARC_COPY                0x21240          0  .SUNW_reloc    sys_errlist
  R_SPARC_COPY                0x21230          0  .SUNW_reloc    sys_nerr
....
```

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。

```
$ cat foo.c
int _size_gets_smaller[16];
int _size_gets_larger[16];
```

```
$ cc -o libfoo.so -G foo.c
$ cc -o prog main.c -L. -R. -lfoo
$ elfdump -sN.symtab prog | grep _size
      [49]  0x000211d0 0x00000040  OBJT GLOB  D    0 .bss           _size_gets_larger
      [59]  0x00021190 0x00000040  OBJT GLOB  D    0 .bss           _size_gets_smaller
$ elfdump -r prog | grep _size
  R_SPARC_COPY               0x211d0          0  .SUNW_reloc    _size_gets_larger
  R_SPARC_COPY               0x21190          0  .SUNW_reloc    _size_gets_smaller
```

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。

```
$ cat foo2.c
int _size_gets_smaller[4];
int _size_gets_larger[32];
$ cc -o libfoo.so -G foo2.c
$ elfdump -sN.symtab libfoo.so | grep _size
      [37]  0x000105cc 0x00000010  OBJT GLOB  D    0 .bss           _size_gets_smaller
      [41]  0x000105dc 0x00000080  OBJT GLOB  D    0 .bss           _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
    ....
      relocation R_SPARC_COPY sizes differ: _size_gets_larger
          (file prog size=0x40; file ./libfoo.so size=0x80)
          prog size used; possible data truncation
      relocation R_SPARC_COPY sizes differ: _size_gets_smaller
          (file prog size=0x40; file ./libfoo.so size=0x10)
          ./libfoo.so size used; possible insufficient data copied
....
```

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。186 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。60 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。
196 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、195 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 8 章

8 mapfile

mapfile を使用すると、リンカーの操作と、結果として得られる出力オブジェクトを詳細に制御できます。

- 出力セグメントの作成または変更、あるいはその両方を行います。
- 入力セクションをセグメントに割り当てる方法と、それらのセクションの相対的な順序を定義します。
- シンボル範囲の指定またはバージョン管理、あるいはその両方を行うことによって、共有可能なオブジェクト用の安定した、下位互換性を持つインタフェースを作成します。
- 共有可能なオブジェクトの依存関係から、使用対象のバージョンを定義します。
- 出力オブジェクトにヘッダーオプションを設定します。
- 動的実行可能ファイルのプロセススタック属性を設定します。
- ハードウェアおよびソフトウェア機能を設定またはオーバーライドします。

注 – `mapfile` を使用せずにリンカーを使用すると、有効な ELF 出力ファイルが常に生成されます。`mapfile` オプションによって、出力オブジェクトに対する高い柔軟性と高性能の制御機能がユーザーに提供されますが、一部のオプションによって、無効または使用できないオブジェクトが生成される可能性があります。ユーザーは、ELF 形式を制御する規則および規約についての知識を持っていることが期待されます。

`-M` コマンド行オプションは、使用される `mapfile` を指定するために使用されます。単一のリンク操作内で複数の mapfile を使用できます。複数の `mapfile` が指定されると、リンカーは、それらがあたかも単一の論理 `mapfile` であるかのように、各ファイルを指定された順序で処理します。この処理は、あらゆる入力オブジェクトが処理される前に実行されます。

システムの `/usr/lib/ld` ディレクトリ内に、一般的な問題を解決するためのサンプル mapfile が提供されています。

mapfile の構造と構文

mapfile 指令は複数の行に渡ることができ、改行を含む空白文字を任意の数だけ含めることができます。

すべての構文説明について、次の表記が適用されます。

- 空白文字または改行は、名前または値の中を除いてどこにでも入れられます。
- ハッシュ文字 (#) 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ます。コメントはリンカーによって解釈されず、ドキュメント化のためのみに使用されます。
- すべての指令はセミコロン (;) で終了します。{ . . . } セクション内の最後のセミコロンは省略できます。
- 固定幅のすべての文字エントリ、すべてのコロン (:)、セミコロン (;)、代入 (=, +=, -=)、および括弧 { . . . } は、そのままの文字を入力します。
- 「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。
- [. . .] 括弧は省略可能な構文を示すために使用されます。括弧は入力しません。実際の指令内には現れません。
- 名前は大文字と小文字の区別がある文字列です。表 8–2 には、mapfile 内で一般的に見られる名前およびその他の文字列のリストが含まれています。名前は 3 つの異なる形式で指定できます。
  - 引用符なし

    引用符なしの名前は英字および数字のシーケンスです。最初の文字は英字である必要があり、0 個以上の英字または数字が後に続きます。パーセント (%)、斜線 (/)、ピリオド (.)、および下線 (_) は英字としてカウントされます。ドル記号 ($)、およびハイフン (-) は数字としてカウントされます。
  - 単一引用符

    単一引用符 (') で囲んだ名前は、単一引用符または改行以外のすべての文字を含むことができます。すべての文字はリテラル文字として解釈されます。この引用形式は、引用符なしの名前では許可されない通常のプリント可能文字を含むファイルパスやその他の名前を指定する場合に便利です。
  - 二重引用符

    二重引用符 (") で囲んだ名前は、二重引用符または改行以外のすべての文字を含むことができます。バックスラッシュ (\) はエスケープ文字です。C プログラミング言語の文字列リテラル内で使用される場合と同じように機能します。表 8–1 で示すように、前にバックスラッシュが付いた文字は、それらが表す文字に置換されます。表 8–1 に示す文字以外の文字がバックスラッシュのあとに続くと、エラーになります。

- *value* は数値を表し、整数定数について C 言語で使用される規則に従って、16 進数、10 進数、または 8 進数が使えます。すべての値は符号なしの整数値で、32 ビット出力オブジェクトの場合は 32 ビット、64 ビット出力オブジェクトの場合は 64 ビットです。
- *segment_flags* には、表 8–3 に示す 1 つ以上の値の空白区切りのリストとしてメモリーアクセス権を指定します。これらの値は `<sys/elf.h>` 内に定義されている PF_ 値に対応します。

表 8–1　二重引用符テキストのエスケープシーケンス

エスケープシーケンス	意味
`\a`	警告 (ベル)
`\b`	バックスペース
`\f`	用紙送り
`\n`	改行
`\r`	return
`\t`	水平タブ
`\v`	垂直タブ
`\\`	バックスラッシュ
`\'`	単一引用符
`\"`	二重引用符
`\ooo`	8 進数の定数で、ooo は 1 つから 3 つまでの 8 進数 (0...7) です

表 8–2　mapfile 内で一般的に使用される名前およびその他の文字列

名前	目的
segment_name	ELF セグメントの名前
section_name	ELF セクションの名前
symbol_name	ELF シンボルの名前
file_path	ELF オブジェクトまたは ELF オブジェクトを含むアーカイブを参照するために使用される、スラッシュ (/) で区切られた複数の名前から構成される UNIX のファイルパス
file_basename	*file_path* の最後のコンポーネント (basename(1))
objname	*file_basename*、またはアーカイブ内に含まれているオブジェクトの名前

表 8–2 mapfile 内で一般的に使用される名前およびその他の文字列 (続き)

名前	目的
soname	共有可能オブジェクトの SONAME に使用される共有可能オブジェクト名 (`libc.so.1` など)
version_name	ELF バージョン管理セクション内で使用されるシンボルバージョンの名前
inherited_version_name	別のシンボルバージョンによって継承されたシンボルバージョンの名前

表 8–3 セグメントフラグ

フラグの値	意味
`READ`	セグメントは読み込み可能です
`WRITE`	セグメントは書き込み可能です
`EXECUTE`	セグメントは実行可能です
`0`	すべてのアクセス権フラグがクリアされます
`DATA`	ターゲットプラットフォーム上のデータセグメントに適した READ、WRITE、および EXECUTE フラグの組み合わせ
`STACK`	プラットフォーム ABI によって定義される、ターゲットプラットフォームに適した READ、WRITE、および EXECUTE フラグの組み合わせ

mapfile のバージョン

コメントまたは空白でない mapfile 内の先頭行には、mapfile のバージョン宣言が期待されます。この宣言によって、ファイルの残りの部分で使用される mapfile 言語のバージョンが確立されます。このマニュアルに記載されている mapfile 言語はバージョン 2 です。

```
$mapfile_version 2
```

バージョン宣言で開始されない mapfile は、AT&T によって System V Release 4 Unix (SVR4) 用に定義されたオリジナルの mapfile 言語で記述されているとみなされます。リンカーはこのような mapfile を処理する能力を保持しています。この構文は、付録 B 「System V Release 4 (バージョン 1) Mapfile」に記載されています。

条件付き入力

mapfile 内の行は、特定の ELFCLASS (32 ビットまたは 64 ビット) あるいは機械タイプにのみ適用されるように条件付けできます。

```
$if expr
...
[$elif expr]
...
[$else]
...
$endif
```

条件付き入力式は、論理的な *true* または *false* の値に評価されます。それぞれの指令 (`$if`、`$elif`、`$else`、および `$endif`) は 1 行に 1 つ記述します。`$if` 行と後続の `$elif` 行の中の式は、*true* と評価される式が見つかるまで順番に評価されます。*false* の値を持つ行に続くテキストは、破棄されます。true の指令の行に続くテキストは、通常どおり処理されます。この説明のテキストとは、条件文の一部でない任意の内容を表します。true となる `$if` または `$elif` が見つかり、そのテキストが処理されると、後続の `$elif` および `$else` 行に加え、それらのテキストも破棄されます。すべての式がゼロで、`$else` がある場合、`$else` に続くテキストが通常どおり処理されます。

`$if` 指令の範囲は、複数の mapfile を超えることはできません。`$if` 指令は、`$if` 指令を使用する `mapfile` 内で、対応する `$endif` によって終了する必要があり、終了しない場合はリンカーはエラーを生成します。

リンカーは、`$if` および `$elif` によって評価される論理式に使用できる名前の内部テーブルを保持します。このテーブルは起動時に、次の表内のそれぞれの名前で初期化され、作成中の出力オブジェクトに適用されます。

表 8–4　定義済みの条件式の名前

名前	意味
`_ELF32`	32 ビットオブジェクト
`_ELF64`	64 ビットオブジェクト
`_ET_DYN`	共有オブジェクトファイル
`_ET_EXEC`	実行可能オブジェクト
`_ET_REL`	再配置可能オブジェクト
`_sparc`	SPARC マシン (32 ビットまたは 64 ビット)
`_x86`	x86 マシン (32 ビットまたは 64 ビット)
`true`	常に定義済み

名前は大文字と小文字を区別して、示されたとおりの名前を使用する必要があります。たとえば、true は定義されていますが、TRUE は定義されていません。これらのすべての名前は、それ自体を論理式として使用できます。次に例を示します。

```
$if _ELF64
...
$endif
```

出力オブジェクトが 64 ビットの場合、この例は *true* に評価され、リンカーは囲まれたテキストを処理します。これらの論理式では数値は許可されませんが、特殊な例外として、値 1 は *true* に評価され、0 は *false* に評価されます。

未定義のすべての名前は *false* に評価されます。一般的に、無条件にスキップする必要がある入力行を指定するために、未定義の名前 false を使用します。

```
$if false
...
$endif
```

次の表に示す演算子を使用すると、さらに複雑な論理式を記述できます。

表 8–5 条件式の演算子

演算子	意味
&&	論理積
||	論理和
(式)	式の一部
!	後続の式のブール値を否定します

式は左から右に評価されます。式の一部は、それを囲む式の前に評価されます。

たとえば、x86 プラットフォーム用の 64 ビットオブジェクトを構築するとき、次の構造構文が評価されます。

```
$if _ELF64 && _x86
...
$endif
```

$add 指令を使用すると、リンカーの既知の名前テーブルに新しい名前を追加できます。前の例を使用すると、$if 指令を簡素化するために、64 ビット x86 オブジェクトを表す amd64 という名前を定義すると便利な場合もあります。

```
$if _ELF64 && _x86
$add amd64
$endif
```

これを使用すると、前の例を簡素化できます。

```
$if amd64
...
$endif
```

$clear 指令は $add 指令の逆です。これは内部テーブルから名前を削除するために使用されます。

```
$clear amd64
```

$add 指令の効果は、$add を使用する mapfile の終わりを超えて持続し、同じリンク操作内でリンカーによって処理される後続の mapfile でも使用できます。この動作を望まない場合は、$add が含まれる mapfile の最後で $clear を使用して定義を削除します。

最後に、$error 指令を使用すると、リンカーは行の残りのすべてのテキストを重大なエラーとして出力し、リンク操作を停止します。$error 指令は、プログラマがオブジェクトを新しい機械タイプに移植する際、必要な mapfile 定義が欠落した正しくないオブジェクトを暗黙のうちに構築できないようにするために使用できます。

```
$if _sparc
...
$elif _x86
...
$else
$error unknown machine type
$endif
```

C 言語のプログラマであれば、mapfile の条件付き入力に使用される構文は、C プリプロセッサのマクロ言語の構文と似ていることがわかります。この類似性は意図的なものです。ただし、mapfile の条件付き入力の指令は、設計上、C プリプロセッサによって提供されるものよりも、ずっと機能が劣ります。クロスプラットフォーム環境でのリンク操作をサポートするために必要なもっとも基本的な機能のみが提供されます。

2 つの言語の大きな違いは次のとおりです。

- C プリプロセッサは完全なマクロ言語を定義し、マクロはソーステキストと、#if および #elif プリプロセッサステートメントによって評価される式の両方に適用されます。リンカーの mapfile にはマクロ機能が実装されていません。
- C プリプロセッサによって評価される式には、数値の型と豊富な演算子セットが含まれます。mapfile 論理式には、ブール型の *true* および *false* 値と、限定された演算子セットが含まれます。
- C プリプロセッサ式は、マクロとして定義されることがある任意の数値を含み、指定のマクロが定義されたものかどうかを評価するために defined() が使用されます。これによって、*true* (ゼロ以外) または *false* (ゼロ) 値が生成されます。mapfile 論理式ではブール値のみを操作でき、名前は defined() 操作を使用せずに直接使用されます。指定された名前はリンカーの既知の名前テーブルにある場合は *true* と見なされ、そうでない場合は *false* と見なされます。

高度なマクロ処理が必要な場合、m4(1) などの外部マクロプロセッサの使用を検討する必要があります。

指令の構文

mapfile 指令は、出力オブジェクトのさまざまな側面を指定するために存在します。これらの指令は共通の構文を共有しています。属性に名前と値のペアを使用し、階層およびグループを表すために {...} 構文を使用します。

mapfile 指令の構文は、次の一般的な形式に基づきます。

もっとも簡単な形式は、値を持たない指令の名前です。

```
directive;
```

次の形式は、指令の名前に、値または空白区切りの値のリストが付いたものです。

```
directive = value...;
```

示されている「=」代入演算子のほかに、「+=」および「-=」形式の代入も使用できます。「=」演算子は、指定された指令を、指定された値または値リストに設定します。「+=」演算子は、右側の値を現在の値に追加するために使用され、「-=」演算子は値を削除するために使用されます。

さらに複雑な指令では、複数の属性を {...} 括弧で囲んで、複数の属性を 1 つの単位としてグループ化して操作します。

```
directive [name] {
        attribute [directive = value];
        ...
} [name];
```

開き括弧 ({) の前に名前を置くことができ、指定されたステートメントの結果に名前を付けるために使用されます。同様に、閉じ括弧 (}) の後ろで終端のセミコロン (;) の前に、1 つ以上のオプションの名前を置くことができます。これらの名前は、定義される項目が、ほかの名前付き項目と関係があることを表すために使用されます。

グループ内の属性の形式は、上記で説明した同じ構文を使用します。つまり、単純な指令に、値が付いたものか、代入演算子 (=、+=、 -=) の後に値が付いたものか、または空白区切りの値のリストが付いたもので、セミコロン (;) で終了します。

指令が持つ属性にはサブ属性を持たせることができます。そのような場合、サブ属性も階層を示すために、入れ子の {...} 括弧でグループ化されます。

```
directive [name] {
        attribute {
                subatribute [= value];
```

```
                    ...
            };
} [name...];
```

mapfile 構文の文法では、入れ子が許可される深さに制限がありません。入れ子の深さは、指令の要件のみに依存します。

mapfile 指令

次の指令がリンカーで受け入れられます。

表 8–6 mapfile 指令

指令	目的
CAPABILITY	ハードウェア、ソフトウェア、およびプラットフォーム機能
DEPEND_VERSIONS	共有オブジェクトの依存関係から許可されるバージョンを指定します
HDR_NOALLOC	ELF ヘッダーおよびプログラムヘッダーは割り当て不可です
LOAD_SEGMENT	読み込み可能な新しいセグメントを作成するか、既存の読み込みセグメントを変更します
NOTE_SEGMENT	注釈セグメントを作成するか、既存の注釈セグメントを変更します
NULL_SEGMENT	ヌルセグメントを作成するか、既存のヌルセグメントを変更します
PHDR_ADD_NULL	ヌルプログラムヘッダーエントリを追加します
SEGMENT_ORDER	出力オブジェクトおよびプログラムヘッダー配列内のセグメントの順序を指定します
STACK	プロセススタック属性
STUB_OBJECT	オブジェクトがスタブオブジェクトとして構築できることを指定します
SYMBOL_SCOPE	命名されていない大域バージョン内でシンボルの属性およびスコープを設定します
SYMBOL_VERSION	明示的に命名されたバージョン内でシンボルの属性およびスコープを設定します

サポートされる各 mapfile 指令の具体的な構文については、後続のセクションで示します。

CAPABILITY 指令

再配置可能オブジェクトのハードウェア、ソフトウェア、マシン、およびプラットフォームの機能は通常、コンパイル時にオブジェクト内部に記録されます。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。mapfile 内で機能を定義して、入力再配置可能オブジェクトから指定される機能に追加したり、完全に置き換えたりすることができます。

```
CAPABILITY [capid] {
        HW  = [hwcap_flag...];
        HW += [hwcap_flag...];
        HW -= [hwcap_flag...];

        HW_1  = [value...];
        HW_1 += [value...];
        HW_1 -= [value...];

        HW_2  = [value...];
        HW_2 += [value...];
        HW_2 -= [value...];

        MACHINE  = [machine_name...];
        MACHINE += [machine_name...];
        MACHINE -= [machine_name...];

        PLATFORM  = [platform_name...];
        PLATFORM += [platform_name...];
        PLATFORM -= [platform_name...];

        SF  = [sfcap_flag...];
        SF += [sfcap_flag...];
        SF -= [sfcap_flag...];

        SF_1  = [value...];
        SF_1 += [value...];
        SF_1 -= [value...];
};
```

オプションの *capid* の名前が存在する場合、この名前はオブジェクト機能のシンボリック名を提供し、結果として出力オブジェクト内に CA_SUNW_ID 機能エントリができます。複数の CAPABILITY 指令が存在する場合、最後の指令の *capid* が使用されます。

空の CAPABILITY 指令を使用すると、機能の値を何も指定せずにオブジェクト機能の *capid* を指定できます。

```
CAPABILITY capid;
```

各タイプの機能について、リンカーは現在の値 (*value*) と、除外する一連の値 (*exclude*) を保持します。ハードウェアおよびソフトウェア機能の場合、これらの値はビットマスクです。マシンおよびプラットフォームの機能の場合、これらは名前のリストです。mapfile を処理する前に、すべての機能の *value* および *exclude* の値がクリアされます。代入演算子の機能を次に示します。

- 「+=」演算子を使用すると、指定された値はその機能の現在の *value* に追加され、その機能の *exclude* 値から削除されます。

- 「-=」演算子を使用すると、指定された値はその機能の *exclude* 値に追加され、その機能の現在の *value* から削除されます。
- 「=」演算子を使用すると、以前の *value* は指定された値に置き換わり、*exclude* が 0 にリセットされます。また、「=」を使用すると、入力ファイル処理から収集されたすべての機能がオーバーライドされます。

入力オブジェクトは mapfile が読み込まれたあとに処理されます。入力オブジェクトによって指定された機能の値は、mapfile からの値とマージされます。ただし、「=」演算子が使用された場合は、入力オブジェクト内に見つかった機能は無視されます。つまり、「=」演算子は入力オブジェクトをオーバーライドし、「+=」演算子は機能を追加します。

結果の機能の値を出力オブジェクトに書き込む前に、リンカーは「-=」演算子で指定されたすべての機能の値を削除します。

指定された機能を出力オブジェクトから完全に除去するには、「=」演算子と空の値リストを使用すれば十分です。たとえば次の例は、入力オブジェクトによって提供されるすべてのハードウェア機能を抑制します。

```
$mapfile_version 2
CAPABILITY {
        HW = ;
};
```

ELF オブジェクト内では、ハードウェアとソフトウェアの機能は、オブジェクトの機能セクションから検出される 1 つ以上のビットマスク内でのビット割り当てとして表現されます。HW および SF mapfile 属性は、この実装の抽象的なビューを提供し、リンカーによって適切なマスクおよびビットに変換される、空白区切りのシンボリック機能名のリストを受け入れます。番号付き属性 (HW_1 、HW_2、SF_1) は、ベースとなる機能ビットマスクへの直接数値アクセスを可能にするために存在します。これらは正式に定義されていない機能ビットを指定するために使用できます。可能な場合、HW および SF 属性を使用することをお勧めします。

HW 属性

ハードウェア機能は、空白区切りのシンボリック機能名のリストとして指定されます。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。mapfile では同じ名前を使用し、AV_ 接頭辞を付けません。たとえば、x86 の AV_SSE ハードウェア機能は mapfile 内では SSE と呼ばれます。このリストには、CA_SUNW_HW_ 機能マスク用に定義されたすべての機能名を含めることができます。

HW_1/HW_2 属性

HW_1 および HW_2 属性には、CA_SUNW_HW_1 および CA_SUNW_HW_2 機能マスクを数値として直接指定したり、そのマスクに対応するシンボリックハードウェア機能名として指定したりできます。

MACHINE 属性

MACHINE 属性は、オブジェクトが実行可能なシステムのマシンハードウェア名を指定します。システムのマシンハードウェア名は、ユーティリティー uname(1) に -m オプションを付けて実行すると表示できます。CAPABILITY 指令は複数のマシン名を指定できます。それぞれの名前は出力オブジェクト内の CA_SUNW_MACH 機能エントリになります。

PLATFORM 属性

PLATFORM 属性は、オブジェクトが実行可能なシステムのプラットフォーム名を指定します。システムのプラットフォーム名は、ユーティリティー uname(1) に -i オプションを付けて実行すると表示できます。CAPABILITY 指令は複数のプラットフォーム名を指定できます。それぞれの名前は出力オブジェクト内の CA_SUNW_PLAT 機能エントリになります。

SF 属性

ソフトウェア機能は、空白区切りのシンボリック機能名のリストとして指定されます。ソフトウェア機能は、<sys/elf.h> の SF1_SUNW_ の値として定義されます。mapfile では同じ名前を使用し、SF1_SUNW_ 接頭辞を付けません。たとえば、SF1_SUNW_ADDR32 ソフトウェア機能は mapfile 内では ADDR32 と呼ばれます。このリストには、CA_SUNW_SF_1 用に定義されたすべての機能名を含めることができます。

SF_1 属性

SF_1 属性には、CA_SUNW_SF_1 機能マスクを数値として直接指定したり、そのマスクに対応するシンボリックソフトウェア機能名として指定したりできます。

DEPEND_VERSIONS 指令

共有可能オブジェクトをリンクするとき、オブジェクトからエクスポートされるすべてのバージョンのシンボルが、通常はリンカーで使用可能になります。DEPEND_VERSIONS 指令は、指定したバージョンへのアクセスに限定するために使用されます。バージョンアクセスの制限は、古いバージョンのシステムで使用できない可能性がある新しい機能が出力オブジェクトで使用されないようにするために使用できます。

DEPEND_VERSIONS 指令には次の構文があります。

```
DEPEND_VERSIONS objname {
        ALLOW = version_name;
        REQUIRE = version_name;
        ...
};
```

objname は、コマンド行で指定された共有オブジェクトの名前です。-l コマンド行オプションを使用してオブジェクトを指定する一般的な場合、これは指定された名前に lib 接頭辞が付いたものになります。たとえば、libc はコマンド行で一般的に -lc と参照されるため、DEPEND_VERSIONS 指令内では libc.so と指定されます。

ALLOW 属性

ALLOW 属性は、指定されたバージョンと、そのバージョンによって継承されたバージョンが、出力オブジェクト内のシンボルを解決するためにリンカーによって使用可能であることを指定します。リンカーは、このバージョンを含む継承の連鎖内で使用されるもっとも高いバージョンの要件を、出力オブジェクトの要件に追加します。

REQUIRE 属性

REQUIRE は、リンク操作の要件を満たすために指定されたバージョンが実際に必要かどうかによらず、そのバージョンを出力オブジェクトの要件に追加します。

HDR_NOALLOC 指令

すべての ELF オブジェクトには、ファイル内のオフセット 0 に ELF ヘッダーがあります。実行可能ファイルおよび共有可能オブジェクトにはプログラムヘッダーもあり、これらは ELF ヘッダーを経由してアクセスされます。リンカーは通常、これらの項目が、読み込み可能な先頭セグメントの一部として組み込まれるように配置します。したがって、これらのヘッダーに含まれる情報は、マップされたイメージ内で可視となり、通常は実行時リンカーによって使用されます。HDR_NOALLOC 指令はこれを防ぎます。

```
HDR_NOALLOC;
```

HDR_NOALLOC が指定されると、ELF ヘッダーおよびプログラムヘッダー配列は、結果として生じる出力オブジェクトファイルの先頭に引き続き示されますが、読み込み可能なセグメントには含まれず、イメージの仮想アドレス計算は、ELF ヘッダーのベースでなく先頭セグメントの先頭セクションから開始されます。

PHDR_ADD_NULL 指令

PHDR_ADD_NULL 指令によって、リンカーは、タイプが PT_NULL の指定された数のプログラムヘッダーエントリを、プログラムヘッダー配列の末尾に追加します。追加の PT_NULL エントリは、事後処理ユーティリティーで使用できます。

```
PHDR_ADD_NULL = value;
```

value は正の整数値である必要があり、作成する追加の PT_NULL エントリの数を指定します。結果として生じるプログラムヘッダーエントリのすべてのフィールドは、0 に設定されます。

LOAD_SEGMENT/NOTE_SEGMENT/NULL_SEGMENT 指令

セグメントは出力オブジェクトの連続する部分で、セクションを含みます。mapfile セグメント指令では、3 つの異なるセグメントタイプの指定が可能です。

- LOAD_SEGMENT

  読み込み可能セグメントは、実行時にプロセスのアドレス空間にマップされるコードまたはデータを含みます。リンカーは割り当て可能な各セグメントに対して PT_LOAD プログラムヘッダーエントリを作成し、実行時リンカーはこれらを使用してセグメントを見つけ、マップします。

- NOTE_SEGMENT

  注釈セグメントには注釈セクションが含まれています。リンカーは、セグメントを参照する PT_NOTE プログラムヘッダーエントリを作成します。注釈セグメントは割り当てできません。

- NULL_SEGMENT

  ヌルセグメントは出力オブジェクトに含まれるセクションを保持しますが、そのセクションは実行時にオブジェクトで使用できません。そのようなセクションの一般的な例として、.symtab シンボルテーブルや、デバッガのために生成されるさまざまなセクションがあります。ヌルセグメントのプログラムヘッダーは作成されません。

セグメント指令は、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するために使用されます。既存のセグメントとは、以前に定義したものか、232 ページの「定義済みセグメント」で説明されている組み込みセグメントのいずれかです。新しいセグメントはそれぞれオブジェクトに追加され、同じタイプの最後のセグメントの後に置かれます。読み込み可能なセグメントが最初に追加され、その次が注釈セグメントで、最後がヌルセグメントです。これらのセグメントに関連付けられたプログラムヘッダーは、セグメントと同じ相対順序でプログラムヘッダー配列に配置されます。デフォルトの配置は、読み込み可能なセグメントの場合は明示的にアドレスを設定するか、または SEGMENT_ORDER 指令を使用すると変更できます。

segment_name が事前に存在するセグメントの場合、指定された属性によって既存のセグメントが変更されます。それ以外の場合、新しいセグメントが作成され、指定された属性が新しいセグメントに適用されます。リンカーは、明示的に指定されていない属性にデフォルト値を設定します。

注 – セグメント名を選択するとき、リンカーの将来のバージョンで、新しい定義済みセグメントが追加される可能性があることに留意してください。ユーザーのセグメント指令に使用する名前がこの新しい名前と一致した場合、新しい定義済みセグメントによって、ユーザーの `mapfile` は、新規セグメントの作成から既存のセグメントの変更へと意味が変わります。この状況を回避するためのもっとも良い方法は、セグメントに一般的な名前を使用することを避け、すべてのセグメント名に対して企業識別子、プロジェクト識別子、プログラムの名前などの固有の接頭辞を追加することです。たとえば、`hello_world` という名前のプログラムの場合、`hello_world_data_segment` というセグメント名を使用できます。

3 つすべてのセグメント指令は、中核となる一連の共通属性を共有します。セグメント宣言は次のようになり、*directive* は `LOAD_SEGMENT`、`NOTE_SEGMENT`、`NULL_SEGMENT` のいずれかで置き換えます。

```
        directive segment_name {
                ASSIGN_SECTION [assign_name];
                ASSIGN_SECTION [assign_name] {
                        FILE_BASENAME = file_basename;
                        FILE_OBJNAME = objname;
                        FILE_PATH = file_path;
                        FLAGS = section_flags;
                        IS_NAME = section_name;
                        TYPE = section_type;
                };

                DISABLE;

                IS_ORDER  = assign_name...;
                IS_ORDER += assign_name...;

                OS_ORDER  = section_name...;
                OS_ORDER += section_name...;
};
```

`LOAD_SEGMENT` 指令は、読み込み可能セグメントに固有の一連の追加属性を受け入れます。これらの追加属性の構文は次のとおりです。

```
        LOAD_SEGMENT segment_name {
                ALIGN = value;

                FLAGS  = segment_flags;
                FLAGS += segment_flags;
                FLAGS -= segment_flags;

                MAX_SIZE = value;

                NOHDR;

                PADDR = value;
                ROUND = value;
```

```
                SIZE_SYMBOL  = symbol_name...;
                SIZE_SYMBOL += symbol_name...;

                VADDR = value;
};
```

どのセグメント指令も空の指令として指定できます。空のセグメント指令によって新しいセグメントが作成されると、すべてのセグメント属性にデフォルト値が設定されます。空のセグメントは次のように宣言されます。

```
        LOAD_SEGMENT segment_name;

        NOTE_SEGMENT segment_name;

        NULL_SEGMENT segment_name;
```

1 つ以上のセグメント指令によって受け入れられるすべての属性について、次に説明します。

ALIGN 属性 (LOAD_SEGMENT のみ)

ALIGN 属性は、読み込み可能セグメントの配置を指定するために使用されます。指定された値は、セグメントに対応するプログラムヘッダーの p_align フィールドに設定されます。セグメント配置は、セグメントの最初の仮想アドレスを計算する際に使用されます。

指定される配置は 2 のべき乗である必要があります。デフォルトでは、リンカーによって、セグメントの配置は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、ソフトウェアのリビジョンによっても異なる場合があります。

ALIGN 属性は PADDR および VADDR 属性と相互に排他的で、これらと一緒に使用できません。PADDR または VADDR が指定されると、対応するプログラムヘッダーの p_align フィールドはデフォルト値に設定されます。

ASSIGN_SECTION 属性

ASSIGN_SECTION は、指定されたセグメントへの割り当て用にセクションを集合的に修飾する、セクション名、タイプ、およびフラグなどのセクション属性の組み合わせです。このような属性のセットは、エントランス基準と呼ばれます。セクションが一致するのは、セクション属性がこれらのエントランス基準と正確に一致するときです。ASSIGN_SECTION は、属性を何も指定しない場合、基準が比較される任意のセクションと一致します。

指定されたセグメントに対して複数の ASSIGN_SECTION 属性を指定できます。ASSIGN_SECTION 属性は、それぞれ互いに独立しています。セクションがセグメントに割り当てられるのは、そのセグメントに関連付けられているいずれかの ASSIGN_SECTION 定義とセクションが一致する場合です。セグメントに 1 つ以上の ASSIGN_SECTION 属性が存在しない場合、リンカーはセクションをセグメントに割り当てません。

リンカーはセクションをセグメントに割り当てるために、エントランス基準の内部リストを使用します。mapfile 内で見つかった ASSIGN_SECTION 宣言は、見つかった順序でこのリストに配置されます。232 ページの「定義済みセグメント」で説明した組み込みセグメントのエントランス基準は、このリスト内で最後の mapfile 定義済みエントリの直後に配置されます。

エントランス基準には、オプションの名前 (*assign_name*) を指定できます。この名前は IS_ORDER 属性と一緒に使用すると、入力セクションが出力セクションに配置される順序を指定できます。

入力セクションを配置するには、リンカーはエントランス基準リストの先頭から開始し、セクションの属性を各エントランス基準と順番に比較します。セクションは、セクション属性に正確に一致した最初のエントランス基準に関連付けられているセグメントに割り当てられます。一致がない場合、一般的なすべての割り当て不可のセクションの場合と同じように、セクションはファイルの末尾に配置されます。

ASSIGN_SECTION は次を受け入れます。

- FILE_BASENAME、FILE_OBJNAME、FILE_PATH

  これらの属性を使用すると、属性を取得したファイルのパス (FILE_PATH)、ベース名 (FILE_BASENAME)、またはオブジェクト名 (FILE_OBJNAME) に基づいてセクションを選択できます。

  ファイルパスは、UNIX 標準のスラッシュ区切りの表記規則を使用して指定されます。最後のパスセグメントはパスのベース名で、単純にファイル名 としても知られています。アーカイブの場合、ベース名はアーカイブメンバーの名前を使用して拡張でき、archive_name(component_name) という形式を使用します。たとえば、/lib/libfoo.a(bar.o) は /lib/libfoo.a という名前のアーカイブにあるオブジェクト bar.o を指定します。

  FILE_BASENAME と FILE_OBJNAME はアーカイブ以外に適用された場合は同等で、指定された名前をファイルのベース名と比較します。アーカイブに適用される場合、FILE_BASENAME はアーカイブ名のベース名を調べます。一方、FILE_OBJNAME はアーカイブ内に格納されているオブジェクトの名前を調べます。

  それぞれの ASSIGN_SECTION は、FILE_BASENAME 、FILE_PATH、および FILE_OBJNAME のすべての値のリストを保持します。これらの定義のいずれかが入力ファイルと一致すると、ファイルが一致したことになります。

- IS_NAME

  入力セクションの名前。

- TYPE

  ELF の *section_type* を指定します。<sys/elf.h> 内で定義されている任意の SHT_ 定数を指定できます。SHT_ 接頭辞は削除します (PROGBITS、SYMTAB、NOBITS など)。

- FLAGS

FLAGS 属性には、*section_flags* を使用して表 8–7 に示す値の空白区切りのリストとしてセクション属性を指定します。これらの値は <sys/elf.h> 内に定義されている SHF_ 値に対応します。個々のフラグの前に感嘆符 (!) が付いている場合は、その属性が存在してはいけないことを明示しています。次の例では、セクションは割り当て可能だが書き込み不可と定義されています。

```
ALLOC !WRITE
```

section_flags リストに明示されていないフラグは無視されます。上記の例では、指定されたフラグに対してセクションを照合するときに、ALLOC および WRITE の値のみが検査されます。ほかのセクションフラグは任意の値を持つことができます。

表 8–7 セクションフラグの値

フラグの値	意味
ALLOC	セクションは割り当て可能です
WRITE	セクションは書き込み可能です
EXECUTE	セクションは実行可能です
AMD64_LARGE	セクションは 2G バイトより大きくすることができます

DISABLE 属性

DISABLE 属性を使用すると、リンカーはセグメントを無視します。無効にされたセグメントには、セクションが割り当てられません。セグメントは後続のセグメント指令によって参照されると、自動的にふたたび有効化されます。したがって、空の参照のみで、無効にされたセクションをふたたび有効にできます。

```
segment segment_name;
```

FLAGS 属性 (LOAD_SEGMENT のみ)

FLAGS 属性は、表 8–3 のアクセス権の空白区切りのリストとして、セグメントのアクセス権を指定します。デフォルトでは、ユーザー定義のセグメントは READ、WRITE、および EXECUTE アクセス権を受け入れます。232 ページの「定義済みセグメント」で説明されている定義済みセグメント用のデフォルトフラグがリンカーによって指定され、場合によってはプラットフォーム依存になります。

3 つの形式を使用できます。

```
FLAGS  = segment_flags...;
FLAGS += segment_flags...;
FLAGS -= segment_flags...;
```

シンプルな「=」代入演算子は現在のフラグを新しいセットで置き換え、「+=」形式は既存のセットに新規フラグを追加し、「-=」形式は指定されたフラグを既存のセットから削除します。

IS_ORDER 属性

リンカーは通常、見つかった順序で出力セクションをセグメントに配置します。同様に、出力セクションを構成する入力セクションも、見つかった順序で配置されます。IS_ORDER 属性は、入力セクションのこのデフォルトの配置を変更するために使用できます。IS_ORDER は、エントランス基準名 (*assign_name*) の空白区切りのリストを指定します。これらのいずれかのエントランス基準に一致したセクションは、出力セクションの先頭に配置され、IS_ORDER で指定された順序で並べ替えられます。IS_ORDER リストに見つからないエントランス基準に一致したセクションは、並べ替えられたセクションの後方に、見つかった順序で配置されます。

「=」形式の代入を使用すると、指定されたセグメントの IS_ORDER の以前の値は破棄され、新しいリストで置き換わります。「+=」形式の IS_ORDER の場合、既存のリストの末尾に新しいリストを連結します。

IS_ORDER 属性が特に注目されるのは、コンパイラの -xF オプションと合わせて使用する場合です。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。リンカーがこれらのセクションを出力に配置するとき、% と、% の後続のすべてが削除されます。したがって、これら 3 つの関数はすべて、.text 出力セクションに配置されます。IS_ORDER 属性は、.text 出力セクション内で特定の相対的な順序で配置することを強制するために使用できます。

次のユーザー定義の mapfile を検討します。

```
$mapfile_version 2
        LOAD_SEGMENT text {
        ASSIGN_SECTION text_bar  { IS_NAME = .text%bar };
        ASSIGN_SECTION text_main { IS_NAME = .text%main };
        ASSIGN_SECTION text_foo  { IS_NAME = .text%foo };
        IS_ORDER = text_foo text_bar text_main;
};
```

これら 3 つの関数がソースコード内で検出された順序や、リンカーによって見つけられた順序に関係なく、出力オブジェクトの text セグメント内での順序は foo()、bar()、main() となります。

MAX_SIZE 属性 (LOAD_SEGMENT のみ)

リンカーはデフォルトで、セグメントの内容が必要とするサイズにまで、セグメントが大きくなることを許可します。セグメントの最大サイズを指定するために、MAX_SIZE 属性を使用できます。MAX_SIZE が設定されると、セグメントが指定されたサイズを超えて大きくなった場合にリンカーはエラーを生成します。

NOHDR 属性 (LOAD_SEGMENT のみ)

NOHDR 属性が設定されたセグメントが、出力オブジェクトの最初の読み込み可能セグメントになった場合、ELF およびプログラムヘッダーはセグメントに含まれません。

NOHDR 属性が最上位の HDR_NOALLOC 指令と異なる点は、HDR_NOALLOC はセグメントごとの値であり、セグメントが最初の読み込み可能セグメントになった場合にのみ有効だということです。この機能は主に、古い mapfile との互換性を確保するために存在します。詳細は、付録 B 「System V Release 4 (バージョン 1) Mapfile」 を参照してください。

セグメントの NOHDR 属性よりも HDR_NOALLOC 指令を優先させることをお勧めします。

OS_ORDER 属性

リンカーは通常、見つかった順序で出力セクションをセグメントに配置します。OS_ORDER 属性は、出力セクションのこのデフォルトの配置を変更するために使用できます。OS_ORDER は、出力セクション名 (*section_name*) の空白区切りのリストを指定します。リストされたセクションは、セグメントの先頭に配置され、OS_ORDER で指定された順序で並べ替えられます。OS_ORDER にリストされていないセクションは、並べ替えられたセクションの後方に、見つかった順序で配置されます。

「=」形式の代入を使用すると、指定されたセグメントの OS_ORDER の以前の値は破棄され、新しいリストで置き換わります。「+=」形式の OS_ORDER の場合、既存のリストの末尾に新しいリストを連結します。

PADDR 属性 (LOAD_SEGMENT のみ)

PADDR 属性は、セグメントの物理アドレスを明示するために使用されます。指定された値は、セグメントに対応するプログラムヘッダーの p_addr フィールドに設定されます。デフォルトでは、リンカーはセグメントの物理アドレスを 0 に設定します。この理由は、このフィールドはユーザーモードオブジェクトについては意味がなく、主にオペレーティングシステムのカーネルなど「ユーザーランド以外」のオブジェクトに関係があるためです。

ROUND 属性 (LOAD_SEGMENT のみ)

ROUND 属性は、セグメントのサイズを指定された値に丸める必要があることを指定するために使用されます。丸める値の指定は 2 のべき乗である必要があります。デフォルトでは、リンカーはセグメントの丸め係数を 1 に設定し、これはセグメントサイズが丸められないことを意味します。

SIZE_SYMBOL 属性 (LOAD_SEGMENT のみ)

SIZE_SYMBOL 属性は、リンカーによって作成される、セクションのサイズシンボル名の空白区切りのリストを定義します。サイズシンボルは、セグメントのサイズをバイト数で示す大域絶対シンボルです。これらのシンボルは、オブジェクトファイル

内で参照できます。コード内でシンボルにアクセスするには、*symbol_name* が言語内で正当な識別子であることを確認する必要があります。シンボルをほかの言語からアクセスできる可能性が高くするため、C プログラミング言語のシンボル命名規則が推奨されます。

「=」形式の代入は初期値を設定するために使用でき、リンカーのセッションにつき 1 回のみ使用できます。「+=」形式の SIZE_SYMBOL の場合、既存のリストの末尾に新しいリストを連結し、必要な回数だけ使用できます。

VADDR (LOAD_SEGMENT のみ)

VADDR 属性は、セグメントの仮想アドレスを明示するために使用されます。指定された値は、セグメントに対応するプログラムヘッダーの p_vaddr フィールドに設定されます。デフォルトでは、リンカーは出力ファイルが作成されるとき、仮想アドレスをセグメントに割り当てます。

SEGMENT_ORDER 指令

SEGMENT_ORDER 指令は、出力オブジェクト内のセグメントのデフォルト以外の順序付けを指定するために使用されます。

SEGMENT_ORDER は、セグメント名の空白区切りのリストを受け入れます。

```
SEGMENT_ORDER  = segment_name...;
SEGMENT_ORDER += segment_name...;
```

「=」形式の代入を使用すると、以前のセグメント順序リストは破棄され、新しいリストで置き換わります。「+=」形式の代入の場合、既存のリストの末尾に新しいリストを連結します。

デフォルトでは、リンカーは次の順序でセグメントを順序付けします。

1. アドレスによって並べ替えられた、LOAD_SEGMENT 指令の VADDR 属性を使用してアドレスを明示した読み込み可能セグメント。
2. SEGMENT_ORDER 指令を使用して、指定された順序で並べ替えられたセグメント。
3. アドレスが明示されていない読み込み可能セグメントで、SEGMENT_ORDER リストに見つからないもの。
4. アドレスが明示されていない注釈セグメントで、SEGMENT_ORDER リストに見つからないもの。
5. アドレスが明示されていないヌルセグメントで、SEGMENT_ORDER リストに見つからないもの。

注 – ELF には、整形オブジェクトが従う必要がある暗黙の規約があります。

- 最初の読み込み可能セグメントは、読み取り専用、割り当て可能、および実行可能であることが期待され、ELF ヘッダーとプログラムヘッダー配列を受け取ります。これは通常、定義済みのテキストセグメントです。
- 実行可能ファイル内の最後の読み込み可能セグメントは、書き込み可能であることが期待され、動的ヒープの先頭は通常、同じ仮想メモリー割り当て内のすぐ後に配置されます。

mapfile を使用すると、これらの要件に違反するオブジェクトを作成できます。そのようなオブジェクトの実行結果は定義されていないため、この操作は回避する必要があります。

HDR_NOALLOC 指令が指定されないかぎり、リンカーは、最初のセグメントが注釈セグメントやヌルセグメントでなく、読み込み可能セグメントであるという要件を強制します。HDR_NOALLOC は「ユーザーランド」のオブジェクト用に使用できないため、あまり実用的ではありません。この機能はオペレーティングシステムのカーネルを構築するときに使用されます。

STACK 指令

STACK 指令はプロセススタックの属性を指定します。

```
        STACK {
                FLAGS  = segment_flags...;
                FLAGS += segment_flags...;
                FLAGS -= segment_flags...;
};
```

FLAGS 属性は、表 8–3 に記載されている任意の値で構成される空白区切りのリストで、セグメントのアクセス権を指定します。

3 つの形式を使用できます。シンプルな「=」代入演算子は現在のフラグを新しいセットで置き換え、「+=」形式は既存のセットに新規フラグを追加し、「-=」形式は指定されたフラグを既存のセットから削除します。

デフォルトのスタックアクセス権はプラットフォーム ABI によって定義され、プラットフォームによって異なります。ターゲットプラットフォームの値はセグメントフラグ名 STACK を使用して指定します。

一部のプラットフォームでは、デフォルトのアクセス権に EXECUTE を含めることが ABI によって要求されます。EXECUTE が必要とされることはほとんどなく、一般的には潜在的なセキュリティーリスクと見なされます。EXECUTE アクセス権をスタックから削除することをお勧めします。

```
STACK {
        FLAGS -= EXECUTE;
};
```

STACK 指令は出力 ELF オブジェクト内の PT_SUNWSTACK プログラムヘッダーエントリに反映されます。

STUB_OBJECT 指令

STUB_OBJECT 指令は、mapfile によって記述されたオブジェクトがスタブオブジェクトとして作成できることをリンカーに通知します。

```
STUB_OBJECT;
```

スタブ共有オブジェクトは、コマンド行で指定された mapfile 内の情報から完全に作成されます。スタブオブジェクトを作成するために -z stub オプションを指定する場合、STUB_OBJECT 指令が mapfile に存在することが必須で、リンカーはシンボルの ASSERT 属性内の情報を使用して、実オブジェクトのシンボルと一致する大域シンボルを作成します。

SYMBOL_SCOPE/SYMBOL_VERSION 指令

SYMBOL_SCOPE および SYMBOL_VERSION 指令は、大域シンボルのスコープと属性を指定するために使用されます。SYMBOL_SCOPE は、命名されていないベースシンボルバージョンのコンテキストで動作し、SYMBOL_VERSION は、明示的に命名された大域バージョンにシンボルを収集するために使用されます。SYMBOL_VERSION 指令を使用すると、下位互換性を保ちながらオブジェクトの進化をサポートできる安定したインタフェースを作成できます。

SYMBOL_VERSION の構文は次のとおりです。

```
SYMBOL_VERSION version_name {
    symbol_scope:
        *;

        symbol_name;
        symbol_name {
                ASSERT = {
                        ALIAS = symbol_name;
                        BINDING = symbol_binding;
                        TYPE = symbol_type;

                        SIZE = size_value;
                        SIZE = size_value[count];
                };
                AUXILIARY = soname;
                FILTER = soname;
                FLAGS = symbol_flags...;
```

```
                        SIZE = size_value;
                        SIZE = size_value[count];

                        TYPE = symbol_type;
                        VALUE = value;
                };
        } [inherited_version_name...];
```

SYMBOL_SCOPE はバージョン名を受け入れませんが、それ以外は同一です。

```
        SYMBOL_SCOPE {
                ...
        };
```

SYMBOL_VERSION 指令では、*version_name* がこの一連のシンボル定義のラベルを提供します。このラベルは、出力オブジェクト内のバージョン定義を指定します。1 つ以上の継承されたバージョン (*inherited_version_name*) を空白区切りで指定できます。この場合、新しく定義されたバージョンは、指定されたバージョンを継承します。第 9 章「インタフェースおよびバージョン管理」を参照してください。

symbol_scope は SYMBOL_SCOPE または SYMBOL_VERSION 指令内でシンボルのスコープを定義します。デフォルトでは、シンボルは大域スコープを持つものと想定されます。これは *symbol_scope* の後ろにコロン (:) を付けて指定することによって変更できます。これらの行は、後続のスコープ宣言によって変更されるまで、後に続くすべてのシンボルのシンボルスコープを決定します。可能性のあるスコープ値と意味を、次の表に示します。

表 8–8　シンボルのスコープのタイプ

スコープ	意味
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 12–20 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 12–20 を参照してください。

表 8–8 シンボルのスコープのタイプ (続き)

スコープ	意味
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 12–20 を参照してください。
exported	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。ほかのどのようなシンボル可視性テクニックを使っても、このシンボル可視性を降格または削除することはできません。このスコープ定義には、シンボルに STV_EXPORTED 可視性が指定された場合と同じ効果があります。表 12–20 を参照してください。
singleton	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。オブジェクト内からそのようなシンボルへの参照は実行時にバインドされるので、シンボルの 1 つのインスタンスだけがプロセス内のすべての参照にバインドされます。ほかのどのようなシンボル可視性テクニックを使っても、このシンボル可視性を降格または削除することはできません。このスコープ定義には、シンボルに STV_SINGLETON 可視性が指定された場合と同じ効果があります。表 12–20 を参照してください。
eliminate	このスコープの大域シンボルは不可視です。これらのシンボルテーブルのエントリは削除されます。このスコープ定義には、シンボルに STV_ELIMINATE 可視性が指定された場合と同じ効果があります。表 12–20 を参照してください。

symbol_name はシンボルの名前です。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、56 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た場合、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た場合、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。

SYMBOL_VERSION 指令が指定されるか、SYMBOL_VERSION または SYMBOL_SCOPE のいずれかで自動縮小が指定された場合、作成されるイメージにバージョン情報が記録されます。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は、第 9 章「インタフェースおよびバージョン管理」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。

シンボルをバージョンに割り当てたり、そのスコープを指定したり、またはその両方を行うには、*symbol_name* 自体をリストすると簡単です。オプションのシンボル属性は {} 括弧内で指定できます。有効な属性を次に示します。

ASSERT 属性

ASSERT 属性は、シンボルの予期される特性を指定するために使用されます。リンカーは、リンク編集で得られたシンボル特性と、ASSERT 属性によって指定されたシンボル特性を比較します。表明された属性と実際の属性が一致しない場合、重大なエラーが発生し、出力オブジェクトは作成されません。

ASSERT 属性の解釈は、STUB_OBJECT 指令または -z stub コマンド行オプションが使用されるかどうかによって異なります。次の 3 つの場合が考えられます。

1. STUB_OBJECT 指令を使用しない場合、ASSERT 属性は不要です。ただし、ASSERT 属性が存在する場合、属性はリンク編集で収集された実際の値に対して検証されます。いずれかの ASSERT 属性が、関連付けられている実際の値と一致しない場合、リンク編集はエラーで終了します。
2. STUB_OBJECT 指令が使用され、-z stub コマンド行オプションが指定された場合、リンカーは ASSERT 指令を使用して、オブジェクトによって提供される大域シンボルの属性を定義します。87 ページの「スタブオブジェクト」を参照してください。
3. STUB_OBJECT 指令が使用され、-z stub コマンド行オプションが指定されない場合、リンカーは、結果として生成されるオブジェクト内のすべての大域データが、対応する ASSERT 指令を持つことを要求します。この指令では、大域データをデータとして宣言し、サイズを指定する必要があります。このモードで、TYPE

ASSERT 属性が指定されない場合、GLOBAL が想定されます。同様に、SH_ATTR が指定されない場合、デフォルト値 BITS が想定されます。これらのデフォルト値によって、スタブと実オブジェクトのデータ属性が互換性を持つことが保証されます。結果の ASSERT ステートメントは、上記の最初の場合と同じ方法で評価されます。225 ページの「STUB_OBJECT 指令」を参照してください。

ASSERT は次を受け入れます。

- ALIAS

  以前定義されたシンボルの別名を定義します。別名シンボルは、メインシンボルと同じタイプ、値、サイズを持ちます。ALIAS 属性は、TYPE、SIZE、および SH_ATTR 属性と一緒に使用できません。ALIAS が指定された場合、タイプ、サイズ、およびセクションの属性は、別名シンボルから取得されます。

- BIND

  ELF の *symbol_binding* を指定します。<sys/elf.h> 内で定義されている任意の STB_ 値を指定できます。STB_ 接頭辞は削除します。たとえば、GLOBAL または WEAK です。

- TYPE

  ELF の *symbol_type* を指定します。<sys/elf.h> 内で定義されている任意の STT_ 定数を指定できます。STT_ 接頭辞は削除します。たとえば、OBJECT、COMMON、または FUNC となります。 また、ほかの mapfile 使用法との互換性を維持するために、FUNCTION および DATA をそれぞれ STT_FUNC および STT_OBJECT に指定できます。TYPE は ALIAS と同時に使用できません。

- SH_ATTR

  シンボルに関連付けられているセクションの属性を指定します。指定できる *section_attributes* を、表 8–9 に示します。SH_ATTR は ALIAS と同時に使用できません。

- SIZE

  予想されるシンボルサイズを指定します。SIZE は ALIAS と同時に使用できません。*size_value* 引数の構文は、SIZE 属性の説明にあるとおりです。231 ページの「SIZE 属性」を参照してください。

表 8–9　SH_ATTR の値

セクション属性	意味
BITS	セクションのタイプは SHT_NOBITS ではありません
NOBITS	セクションのタイプは SHT_NOBITS です

AUXILIARY 属性

このシンボルが共有オブジェクト名 (*soname*) の補助フィルタであることを示します。149 ページの「補助フィルタの生成」を参照してください。

FILTER 属性

このシンボルが共有オブジェクト名 (*name*) のフィルタであることを示します。146 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。

FLAGS 属性

symbol_flags は、次の値が 1 つ以上含まれる空白区切りリストとしてシンボル属性を指定します。

表 8–10　シンボルフラグの値

フラグ	意味
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。このフラグ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。第 6 章「直接結合」を参照してください。このフラグを PARENT フラグとともに使用すると、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。
DYNSORT	このシンボルをソートセクションに取り込むべきであることを示します。380 ページの「シンボルソートセクション」を参照してください。シンボルタイプは STT_FUNC、STT_OBJECT、STT_COMMON、または STT_TLS である必要があります。
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。このフラグによって、-z defs オプションで示される未定義シンボルが抑制されます。このフラグは、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。
INTERPOSE	このシンボルは割り込み処理として機能することを示します。このフラグは、動的実行可能ファイルを生成するときにだけ使用できます。このフラグは、割り込みシンボルを定義する際に、-z interpose オプションを使用したときよりも詳細な制御を提供します。
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。第 6 章「直接結合」を参照してください。このフラグを PARENT フラグとともに使用すると、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。
NODYNSORT	このシンボルをソートセクションに含めてはならないことを示します。380 ページの「シンボルソートセクション」を参照してください。

表 8–10 シンボルフラグの値 (続き)

フラグ	意味
PARENT	シンボルが、作成中のオブジェクトの親で定義されていることを示します。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。このフラグは通常、コールバックルーチンへのラベル付けで定義されます。このフラグを DIRECT または NODIRECT フラグとともに使用すると、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。このフラグによって、-z defs オプションで示される未定義シンボルが抑制されます。このフラグは、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。

SIZE 属性

サイズ属性を設定します。この属性により、シンボル定義が作成されます。

size_value 引数には、数値またはシンボリック名 addrsize を指定できます。addrsize はメモリーアドレスを保持できるマシンワードのサイズを表します。リンカーは addrsize に対し、32 ビットオブジェクトを作成するときは値 4 を、64 ビットオブジェクトを構築するときは値 8 を代入します。addrsize は、条件付き入力の使用を必要としないで 32 ビットおよび 64 ビットオブジェクトに合わせて自動的に調節されるため、ポインタ変数および C 変数の long 型のサイズを表す際に使用すると便利です。

size_value 引数にはオプションで *count* 値を角括弧で囲んで接尾辞として追加できます。*count* が存在する場合、*size_value* と *count* が掛け合わされて最終的なサイズの値が取得されます。

TYPE 属性

シンボルのタイプ属性です。この属性は、COMMON、 DATA、または FUNCTION のいずれかです。COMMON を指定すると、一時的なシンボル定義になります。DATA および FUNCTION を指定すると、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。371 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。

データ属性を指定すると、OBJT シンボルが作成されます。サイズを指定し値を指定しないデータ属性を指定すると、セクションシンボルが ELF セクションに関連付けられて作成されます。このセクションは、ゼロで埋められます。関数属性を指定すると、FUNC シンボルが作成されます。

サイズを指定し値を指定しない関数属性を指定すると、セクションシンボルが ELF セクションに関連付けられて作成されます。このセクションには、リンカーによって生成される void 関数が、次のシグニチャーを使用して割り当てられます。

```
void (*)(void)
```

値が指定されたデータまたは関数属性を指定すると、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。196 ページの「コピー再配置」を参照してください。

VALUE 属性

値の属性を示します。この属性により、シンボル定義が作成されます。

定義済みセグメント

リンカーには、定義済みの出力セグメント記述子とエントランス基準のセットが提供されています。これらの定義は、ほとんどのリンクシナリオのニーズを満たし、システムによって期待される ELF レイアウト規則および規約に適合します。

text、data、および extra セグメントがもっとも重要で、その他のセグメントは次に説明する特殊な目的のために役立ちます。

- `text`

  `text` セグメントは、割り当て可能な書き込み不可のセクションを受け入れる、読み取り専用の実行可能で読み込み可能なセグメントを定義します。これには、実行可能コード、プログラムで必要な読み取り専用データ、および実行時リンカーによって使用されるリンカーによって生成される読み取り専用データ (動的シンボルテーブルなど) が含まれます。

  `text` セグメントはプロセスの最初のセグメントであり、そのため、リンカーによって ELF ヘッダーおよびプログラムヘッダー配列が割り当てられます。この動作は `HDR_NOALLOC` mapfile 指令を使用すると回避できます。

- `data`

  `data` セグメントは書き込み可能で読み込み可能なセグメントを定義します。`data` セグメントは、プログラムが必要とする書き込み可能データや、実行時リンカーが使用する書き込み可能データに使用します。たとえば、大域オフセットテーブル (GOT) や、SPARC などのアーキテクチャー上でプロシージャーのリンクテーブル (PLT) を書き込み可能にする必要があるときに PLT に使用します。

- `extra`

  extra セグメントは、どこにも割り当てられず、最後のエントランス基準レコードによってそこに指定されたすべてのセクションを取り込みます。一般的な例として、シンボルテーブル (`.symtab`) や、デバッガ用に生成されるさまざまなセクションがあります。これはヌルセグメントで、対応するプログラムヘッダーテーブルエントリはありません。

- note

  note セグメントは、SHT_NOTE タイプのすべてのセクションを取り込みます。リンカーは、note セグメントを参照するための PT_NOTE プログラムヘッダーエントリを提供します。

- lrodata/ldata

  x86–64 ABI では、小規模、中規模、および大規模なコンパイルモデルを定義します。ABI では、中規模および大規模なモデルのセクションについて、SHF_AMD64_LARGE セクションフラグを設定する必要があります。SHF_AMD64_LARGE がない入力セクションは、2G バイトのサイズを超えない出力セグメント内に配置する必要があります。lrodata および ldata の定義済みセグメントは x86–64 出力オブジェクトについてのみ存在し、SHF_AMD64_LARGE フラグが設定されているセクションを処理するために使用されます。lrodata は読み取り専用セクションを受け取り、ldata はその他のセクションを受け取ります。

- bss

  ELF では任意のセグメントに NOBITS セクションを含めることができます。リンカーはそれらのセクションを、セクションが割り当てられているセグメントの最後に配置します。これはプログラムヘッダーエントリ p_filesz および p_memsz フィールドを使用して実装され、次の規則に従う必要があります。

  ```
  p_memsz >= p_filesz
  ```

  p_memsz が p_filesz より大きい場合、余分なバイトは NOBITS になります。先頭の p_filesz バイトはオブジェクトファイルに由来し、p_memsz までの残りのバイトはすべて、使用前にシステムによってゼロ化されます。

  デフォルトの割り当て規則では、読み取り専用の NOBITS セクションを text セグメントに、書き込み可能な NOBITS セクションを data セグメントに割り当てます。リンカーは bss セグメントを、書き込み可能な NOBITS セクションを受け入れることができる代替セグメントとして定義します。このセグメントはデフォルトで無効化されており、使用するには明示的に有効化する必要があります。

  書き込み可能な NOBITS セクションは data セグメントの一部として容易に処理できるため、bss セグメントをべつに持つ利点はあまり明確ではありません。慣例的に、プロセス動的メモリーヒープは最終セグメントの末尾から開始し、書き込み可能である必要があります。これは通常はデータセグメントですが、bss が有効な場合は bss が最終セグメントになります。動的実行可能プログラムを構築するときに、適切な配置で bss セグメントを有効にすると、ヒープに大きなページを割り当てられるようになります。bss セグメントを有効にして 4M バイトの配置を設定する例を次に示します。

  ```
  LOAD_SEGMENT bss {
          ALIGN=0x400000;
  };
  ```

注 – 配置の指定はマシン固有であるため、ハードウェアプラットフォームの種類によっては同様の利点が得られない場合があります。将来のリリースでは、基本となる最適なページサイズをより柔軟に要求する方法が進展する可能性があります。

マッピングの例

ユーザー定義の mapfile の例を次に示します。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: セクションからセグメントへの割り当て

この例では、セグメントを定義し、セグメントに入力セクションを割り当てる方法について説明します。

例 8–1 セクションからセグメントへの基本的な割り当て

```
 1    $mapfile_version 2
 2    LOAD_SEGMENT elephant {
 3            ASSIGN_SECTION {
 4                    IS_NAME=.data;
 5                    FILE_PATH=peanuts.o;
 6            };
 7            ASSIGN_SECTION {
 8                    IS_NAME=.data;
 9                    FILE_OBJNAME=popcorn.o;
10           };
11   };
12
13   LOAD_SEGMENT monkey {
14           VADDR=0x80000000;
15           MAX_SIZE=0x4000;
16           ASSIGN_SECTION {
17                   TYPE=progbits;
18                   FLAGS=ALLOC EXECUTE;
19           };
20           ASSIGN_SECTION {
21                   IS_NAME=.data
22           };
23   };
24
25   LOAD_SEGMENT donkey {
26           FLAGS=READ EXECUTE;
27           ALIGN=0x1000;
28           ASSIGN_SECTION {
29                   IS_NAME=.data;
30           };
31   };
32
```

例 8–1 セクションからセグメントへの基本的な割り当て (続き)

```
33   LOAD_SEGMENT text {
34           VADDR=0x80008000
35   };
```

4 つの別々のセグメントがこの例では扱われています。1 行目に示すように、すべての mapfile は $mapfile_version 宣言で始まります。elephant セグメント (2 行目から 11 行目) は、peanuts.o または popcorn.o ファイルからすべてのデータセクションを受け取ります。オブジェクト popcorn.o はアーカイブから取得でき、その場合、アーカイブファイルは任意の名前を持つことができます。あるいは、popcorn.o はベース名が popcorn.o の任意のファイルから取得できます。これに対して、 peanuts.o は同一の名前のファイルからのみ取得できます。たとえば、/var/tmp/peanuts.o がリンク編集時に指定された場合、peanuts.o と一致しません。

monkey セグメント (13 行目から 23 行目) には、仮想アドレス 0x80000000 および最大長 0x4000 が指定されています。このセグメントは、PROGBITS および割り当て可能な実行可能プログラムの両方であるすべてのセクション、さらに elephant セグメントに含まれていない .data という名前のすべてのセクションを受け取ります。monkey セグメントに入る .data セクションは、PROGBITS でも割り当て可能な実行可能プログラムである必要もありません。これは、.data セクションは 16 行目でなく 20 行目のエントランス基準に一致するためです。このことは、ASSIGN_SECTION 属性内のサブ属性間には *and* 関係が存在し、単一セグメントの異なる ASSIGN_SECTION 属性間には *or* 関係が存在することを示しています。

donkey セグメント (25 行目から 31 行目) は、デフォルト以外のアクセス権フラグと配置を指定し、.data という名前のすべてのセクションを受け入れます。ただし、このセグメントにはどのセクションも割り当てられず、その結果、donkey セグメントは出力オブジェクトに含まれません。この理由は、リンカーは mapfile に現れる順序でエントランス基準を検証するためです。この mapfile では、elephant セグメントが一部の .data セクションを受け入れ、monkey セグメントが残りを受け取るため、donkey には何も残りません。

33 行目から 35 行目では、text セグメントの仮想アドレスを 0x80008000 に設定します。text セグメントは、232 ページの「定義済みセグメント」で説明されているように、標準の定義済みセグメントの 1 つであるため、このステートメントは新しいセグメントを作成するのではなく、既存のセグメントを変更します。

例: 定義済みセクションの変更

次の mapfile の例では、定義済みの text セグメントと data セグメント、ヘッダーオプション、およびセグメント内のセクションの順序付けを操作します。

例 8–2 定義済みセクションの操作とセクションからセグメントへの割り当て

```
 1    $mapfile_version 2
 2    HDR_NOALLOC;
 3
 4    LOAD_SEGMENT text {
 5            VADDR=0xf0004000;
 6            FLAGS=READ EXECUTE;
 7            OS_ORDER=.text .rodata;
 9            ASSIGN_SECTION {
10                   TYPE=PROGBITS;
11                   FLAGS=ALLOC !WRITE;
12           };
13   };
14
15   LOAD_SEGMENT data {
16           FLAGS=READ WRITE EXECUTE;
17           ALIGN=0x1000;
18           ROUND=0x1000;
19   };
```

先頭行は常に、使用する mapfile 言語のバージョンを宣言します。HDR_NOALLOC 指令 (2 行目) では、結果オブジェクトは、オブジェクトの最初に割り当て可能なセグメント (定義済み text セグメント) の内部に ELF ヘッダーまたはプログラムヘッダー配列を含んではならないということが指定されています。

4 行目から 13 行目のセグメント指令は、text セグメントに仮想アドレスとアクセス権フラグを設定します。この指令では、.text という名前のセクションがセグメントの先頭に配置され、.rodata という名前のすべてのセクションがその後に続き、その他のすべてのセクションはこれらの後に続く必要があることが指定されます。最後に、割り当てることができる書き込み不可の PROGBITS セクションがセグメントに割り当てられます。

15 行目から 19 行目のセグメント指令は、data セグメントを 0x1000 境界に配置する必要があることを指定します。これは、セグメント内の先頭セクションを同じ配置方法で整列する効果があります。セグメントの長さは、配置方法と同じ値の倍数値に切り上げられます。セグメントのアクセス権は、読み取り、書き込み、および実行に設定されます。

リンカー内部情報: セクションおよびセグメント処理

ここでは、セクションを出力セグメントに割り当てるためにリンカーによって使用される内部処理について説明します。この情報は、mapfile を使用するためには必要ありません。この情報は主に、リンカーの内部情報に関心を持ち、セグメントの mapfile 指令がリンカーによってどのように解釈されて実行されるかについて深く理解することが必要な読者を対象としています。

セクションからセグメントへの割り当て

入力セクションを出力セグメントに割り当てるプロセスには、次のデータ構造が関与します。

- 入力セクション

  入力セクションは再配置可能オブジェクト入力からリンカーに読み取られます。一部の情報はリンカーによって検査および処理されますが、その他の情報は内容が検査されることなく単純に出力に渡されます (PROGBITS など)。

- 出力セクション

  出力セクションは、出力オブジェクトに書き込まれるセクションです。一部のセクションは、入力オブジェクトから渡されたセクションの連結によって形成されます。シンボルテーブルや再配置セクションなどのその他のセクションは、一般的には入力オブジェクトから読み込んだ情報を組み込んで、リンカー自体によって生成されます。

  リンカーが入力セクションを通過させて出力セクションにした場合、そのセクションは通常、入力セクションの名前を保持します。ただし、リンカーは特定の状況で名前を変更できます。たとえば、リンカーは name%XXX という形式の入力セクション名を変換し、% 文字とその後に続く文字を出力セクションから削除します。

- セグメント記述子

  リンカーは既知のセグメントのリストを保持しています。このリストは最初は、232 ページの「定義済みセグメント」に記載されている定義済みセグメントが格納されています。LOAD_SEGMENT、NOTE_SEGMENT、または NULL_SEGMENT の mapfile 指令が使用されて新しいセグメントが作成されると、新しいセグメント用の追加のセグメント記述子がこのリストに追加されます。新しいセグメントは、仮想アドレスの設定 (LOAD_SEGMENT) や、SEGMENT_ORDER 指令を使用して明示的に順序付けされないかぎり、このリストの末尾、同じタイプのほかのセグメントの後に配置されます。

  出力オブジェクトを作成するとき、リンカーはセクションを受け取るセグメントについてのみプログラムヘッダーを作成します。空のセグメントは自動的に無視されます。したがって、リンカーのリストからセグメント定義を削除するための明示的な機能はありませんが、ユーザー指定のセグメント定義を使用すると、定義済みセグメントの定義の使用を完全に置き換えることができます。

- エントランス基準

  セクションを所定のセグメントに配置するために必要な一連のセクション属性を、そのセグメントのエントランス基準と呼びます。セグメントは任意の数のエントランス基準を持つことができます。

  リンカーは、定義済みのすべてのエントランス基準の内部リストを保持しています。このリストは、次に説明するように、セクションをセグメントに配置するために使用されます。各 mapfile は、LOAD_SEGMENT、NOTE_SEGMENT、または NULL_SEGMENT の mapfile 指令の ASSIGN_SECTION 属性によって作成されたエントラン

ス基準を、mapfile 内で見つかった順序でこのリストの上部に挿入します。232 ページの「定義済みセグメント」で説明されている組み込みセグメントのエントランス基準は、このリストの最後に配置されます。したがって、mapfile で定義されるエントランス基準は組み込み規則よりも優先され、コマンド行の最後にある mapfile は、最初に検出されたものよりも優先されます。

出力オブジェクトに書き込まれる各セクションについて、リンカーは次の手順を実行して、セクションを出力セグメントに配置します。

1. セクションの属性は、内部エントランス基準リストの先頭から各レコードと比較され、各エントランス基準が順番に検証されます。エントランス基準内のすべての属性が完全に一致したときにエントランス基準と一致したことになります。そのエントランス基準に関連付けられたセグメントは無効化されません。検索は、一致した最初のエントランス基準で停止し、セクションは関連付けられたセグメントに指定されます。

   エントランス基準に一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。デバッグセクションなどの割り当て不能なセクションは、ほとんどがこの領域に配置されることになります。
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の属性値と完全に一致する場合、セクションはその出力セクションに対応するセクションの列挙の最後に置かれます。

   一致する出力セクションが見つからない場合、配置されるセクションの属性を使用して新しい出力セクションが作成され、新しい出力セクション内に入力セクションが配置されます。この新しい出力セクションは、セグメント内で同じセクションタイプを持つほかの出力セクションの後ろに配置されますが、ほかの出力セクションが存在しない場合はセグメントの末尾に配置されます。

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、このセクションは PROGBITS セクションとして処理されます。mapfile でこのセクションタイプ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 (セクションフラグ、セクション名) を使って付け直すことができます。

定義済みセグメントとエントランス基準のための mapfile 指令

リンカーには、232 ページの「定義済みセグメント」で説明されているように、定義済みの出力セグメント記述子とエントランス基準のセットが提供されています。リンカーはこれらのセクションについて認識しているため、mapfile 指令でこれらを作

成する必要はありません。これらの作成に使用できる mapfile 指令は、説明のため、または比較的複雑な mapfile を指定する例として示されています。mapfile セグメント指令は、これらの組み込み定義を変更または拡張するために使用できます。

通常、セクションからセグメントへの割り当ては、単一のセグメント指令の内部で行われます。しかし、定義済みのセクションにはより複雑な要件があるため、セグメントがメモリーに展開されている順序と異なる順序でエントランス基準を処理する必要があります。これを実現するには 2 つの方法が使用されます。1 つ目は、すべてのセグメントを望ましい順序で定義する方法、2 つ目は、望ましい結果が得られる順序でエントランス基準を設定する方法です。ユーザーの mapfile でこの方法が必要になることはまれです。

```
# Predefined segments and entrance criteria for the Oracle Solaris
# link-editor
$mapfile_version 2

# The lrodata and ldata segments only apply to x86-64 objects.
# Establish amd64 as a convenient token for conditional input
$if _ELF64 && _x86
$add amd64
$endif

# Pass 1: Define the segments and their attributes, but
# defer the entrance criteria details to the 2nd pass.
LOAD_SEGMENT text {
        FLAGS = READ EXECUTE;
};
LOAD_SEGMENT data {
        FLAGS = READ WRITE EXECUTE;
};
LOAD_SEGMENT bss {
        DISABLE;
        FLAGS=DATA;
};
$if amd64
        LOAD_SEGMENT lrodata {
                FLAGS = READ
        };
        LOAD_SEGMENT ldata {
                FLAGS = READ WRITE;
        };
$endif
NOTE_SEGMENT note;
NULL_SEGMENT extra;

# Pass 2: Define ASSIGN_SECTION attributes for the segments defined
   # above, in the order the link-editor should evaluate them.

# All SHT_NOTE sections go to the note segment
NOTE_SEGMENT note {
        ASSIGN_SECTION {
                TYPE = NOTE;
        };
};
$if amd64
```

```
        # Medium/large model x86-64 readonly sections to lrodata
        LOAD_SEGMENT lrodata {
                ASSIGN_SECTION {
                        FLAGS = ALLOC AMD64_LARGE;
                };
        };
$endif

# text receives all readonly allocable sections
LOAD_SEGMENT text {
        ASSIGN_SECTION {
                FLAGS = ALLOC !WRITE;
        };
};

# If bss is enabled, it takes the writable NOBITS sections
# that would otherwise end up in ldata or data.
LOAD_SEGMENT bss {
        DISABLE;
        ASSIGN_SECTION {
                FLAGS = ALLOC WRITE;
                TYPE = NOBITS;
        };
};

$if amd64
        # Medium/large model x86-64 writable sections to ldata
        LOAD_SEGMENT ldata {
                ASSIGN_SECTION {
                        FLAGS = ALLOC WRITE AMD64_LARGE;
                };
                ASSIGN_SECTION {
                        TYPE = NOBITS;
                        FLAGS = AMD64_LARGE
                };
        };
$endif

# Any writable allocable sections not taken above go to data
LOAD_SEGMENT data {
        ASSIGN_SECTION {
                FLAGS = ALLOC WRITE;
        };
};

# Any section that makes it to this point ends up at the
# end of the object file in the extra segment. This accounts
# for the bulk of non-allocable sections.
NULL_SEGMENT extra {
        ASSIGN_SECTION;
};
```

◆ ◆ ◆ 第 9 章

9 インタフェースおよびバージョン管理

リンカーおよび実行時リンカーで処理される 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用意されています。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。

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、60 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。

次のセクションは、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 9–1 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
非可変引数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。ただし多くの場合、ソースレベルの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、次のセクション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でバージョン定義を行うと、次の 2 つの機能が利用できます。

- 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、60 ページの「シンボル範囲の縮小」セクション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
#include <stdio.h>
extern  const char *_foo1;

void foo1()
{
        (void) printf(_foo1);
}

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

$ cat mapfile
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                  # Release X
        global:
                foo1;
        local:
                *;
};
$ cc -c -Kpic foo.c data.c
$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ elfdump -sN.symtab libfoo.so.1 | grep 'foo.$'
      [32]  0x0001074c 0x00000004  OBJT LOCL  H    0 .data          _foo1
      [53]  0x00000560 0x00000038  FUNC GLOB  D    0 .text          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。自動縮小指令については、225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」で説明されます。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、65 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
#include <stdio.h>
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、249 ページの「バージョン定義への結合」と253 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
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                   # Release X
        global:
                foo1;
        local:
                *;
};
```

```
SYMBOL_VERSION SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;

$ cc -o libfoo.so.1 -M mapfile -G foo.o data.o
$ elfdump -sN.symtab libfoo.so.1 | grep 'foo.$'
      [28]  0x000107a4 0x00000004  OBJT LOCL  H    0 .data          _foo1
      [29]  0x000107a8 0x00000004  OBJT LOCL  H    0 .data          _foo2
      [48]  0x000005e8 0x00000020  FUNC GLOB  D    0 .text          foo1
      [51]  0x00000618 0x00000020  FUNC GLOB  D    0 .text          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、249 ページの「バージョン定義への結合」セクション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
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                   # Release X
        global:
                foo1;
        local:
                *;
};

SYMBOL_VERSION SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;

SYMBOL_VERSION 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。249 ページの「バージョン定義への結合」セクションでは、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
$mapfile_version 2
SYMBOL_VERSION SUNW_1.1 {                   # Release X
        global:
                foo1;
        local:
                *;
};

SYMBOL_VERSION SUNW_1.2 {                   # Release X+1
        global:
                foo2;
} SUNW_1.1;

SYMBOL_VERSION SUNW_1.2.1 { } SUNW_1.2;     # Release X+2

SYMBOL_VERSION SUNW_1.3a {                  # Release X+3
        global:
                bar1;
} SUNW_1.2;

SYMBOL_VERSION 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。次のセクションでは、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、38 ページの「共有オブジェクトの処理」と 141 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、前のセクション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。この検証は、ldd(1) で -v オプション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。105 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());
                return (1);
        }

        if (dlsym(handle, vers) == NULL) {
                (void) printf("fatal: %s: version ‘%s’ not found\n",
                    file, vers);
                return (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リンクされた動的オブジェクトを作成する場合、特定のバージョン定義に対する結合を制限するように、リンカーに指示できます。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 DEPEND_VERSIONS mapfile 指令で制御できます。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。DEPEND_VERSIONS 指令は次の構文を使用します。

```
$mapfile_version 2
DEPEND_VERSIONS objname {
        ALLOW   = version_name;
        REQUIRE = version_name;
        ...
};
```

- *objname* は共有オブジェクトの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。40 ページの「ライブラリの命名規約」を参照してください。
- ALLOW 属性を使用して、結合に利用できるようにすべき共有オブジェクト内のバージョン定義名を指定します。複数の ALLOW 属性を指定できます。
- REQUIRE 属性を使用すると、記録されるバージョン定義を追加できます。複数の REQUIRE 属性を指定できます。

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
$mapfile_version 2
DEPEND_VERSIONS libfoo.so {
        ALLOW = 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

オブジェクトの標準のシンボル結合から作成されるバージョン依存関係よりも多くのバージョン依存関係を記録するには、DEPEND_VERSIONS mapfile 指令に REQUIRE 属性を使用します。次のセクションでは、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
$mapfile_version 2
DEPEND_VERSIONS libfoo.so {
        ALLOW = SUNW_1.1;
        REQURE = 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

246 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。

REQUIRE 属性を DEPEND_VERSIONS mapfile 指令に使用すると、バージョン定義に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
$mapfile_version 2
DEPEND_VERSIONS libfoo.so {
        ALLOW = SUNW_1.1;
        REQUIRE = 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証できます。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

ここまでのセクションで、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、60 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。

次のセクションでは、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。

```
$ elfdump -d prog | grep libfoo
       [0]  NEEDED           0x1b7              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
$ elfdump -d prog | grep libfoog
       [0]  NEEDED           0x1b7              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 10 章

10 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。

- `$CAPABILITY` (`$HWCAP`)
- `$ISALIST`
- `$OSNAME`、`$OSREL`、`$PLATFORM`、および `$MACHINE`
- `$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。

機能固有の共有オブジェクト

動的トークン `$CAPABILITY` を使用すると、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。

注 – 元の機能はハードウェア機能のみに基づいて実装されました。トークン `$HWCAP` は、この機能処理を選択するために使用されました。それ以降、ハードウェア機能を超えて機能が拡張されたため、`$HWCAP` トークンは `$CAPABILITY` トークンで置き換えられました。互換性については、`$HWCAP` トークンは `$CAPABILITY` トークンの別名として解釈されます。

パス名の指定は、$CAPABILITY トークンで終わるフルパス名で構成する必要があります。$CAPABILITY 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、機能の要件を示す必要があります。66 ページの「機能要件の特定」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機能に対して検証されます。プロセスで使用できるオブジェクトは、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。

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/cap/$CAPABILITY' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ elfdump -d libfoo.so.1 | egrep 'SONAME|AUXILIARY'
       [2]  SONAME            0x1                libfoo.so.1
       [3]  AUXILIARY         0x96               /opt/ISV/lib/cap/$CAPABILITY
$ elfdump -H /opt/ISV/lib/cap/*

/opt/ISV/lib/cap/filtee.so.3:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x1000  [ SSE2 ]

/opt/ISV/lib/cap/filtee.so.1:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x40  [ MMX ]

/opt/ISV/lib/cap/filtee.so.2:

Capabilities Section:  .SUNW_cap

 Object Capabilities:
     index  tag               value
       [0]  CA_SUNW_HW_1     0x800  [ SSE ]
```

フィルタ libfoo.so.1 を MMX と SSE のハードウェア機能を使用できるシステムで処理した場合、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=cap/filtee.so.2  [ ELF ]
01233: symbol=foo;  lookup in file=cap/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 $CAPABILITY 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 $CAPABILITY 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たり、プロセスに適したものをソートしたり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=cap/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、Oracle 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、263 ページの「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
$ elfdump -d libfoo.so.1 | egrep 'SONAME|AUXILIARY'
       [2]  SONAME            0x1                libfoo.so.1
       [3]  AUXILIARY         0x96               /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
$ elfdump -d libfoo.so.1 | egrep 'RUNPATH|AUXILIARY'
       [3]  AUXILIARY         0x96               libbar.so.1
       [4]  RUNPATH           0xa2               /opt/ISV/lib/$ISALIST
```

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` 構成の `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、$PLATFORM、および $MACHINE は実行時に展開され、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。$PLATFORM は、uname -i を実行した場合に表示されるように、ベースとなるプラットフォーム名を反映して展開されます。$MACHINE は、uname -m を実行した場合に表示されるように、ベースとなるマシンのハードウェア名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。

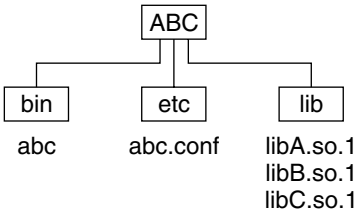
```
$ LD_OPTIONS='-f /platform/$PLATFORM/lib/libbar.so.1' \
cc -o libfoo.so.1 -G -K pic -h libfoo.so.1 -R. foo.c
$ elfdump -d libfoo.so.1 | egrep 'SONAME|AUXILIARY'
       [2]  SONAME            0x1                 libfoo.so.1
       [3]  AUXILIARY         0x96                /platform/$PLATFORM/lib/libbar.so.1
```

このメカニズムは、共有オブジェクト /lib/libc.so.1 にプラットフォーム固有の拡張を行うために、Oracle 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 10–1 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、PATH に /opt/ABC/bin を追加して、製品のバイナリの位置を特定します。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
$ elfdump -d abc | egrep 'NEEDED|RUNPATH'
       [0]  NEEDED            0x1b5              libA.so.1
       ....
       [4]  RUNPATH           0x1bf              /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
$ elfdump -d libA.so.1 | egrep 'NEEDED|RUNPATH'
       [0]  NEEDED            0x96               libB.so.1
       [4]  RUNPATH           0xa0               /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
$ elfdump -d abc | egrep 'NEEDED|RUNPATH'
       [0]  NEEDED            0x1b5              libA.so.1
```

```
      ....
      [4]  RUNPATH           0x1bf               $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
$ elfdump -d lib/libA.so.1 | egrep 'NEEDED|RUNPATH'
      [0]  NEEDED            0x96                libB.so.1
      [4]  RUNPATH           0xa0                $ORIGIN
```

次に、この製品が /usr/local/ABC 内にインストールされ、またユーザーの 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を検索することになります。

```
$ 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
```

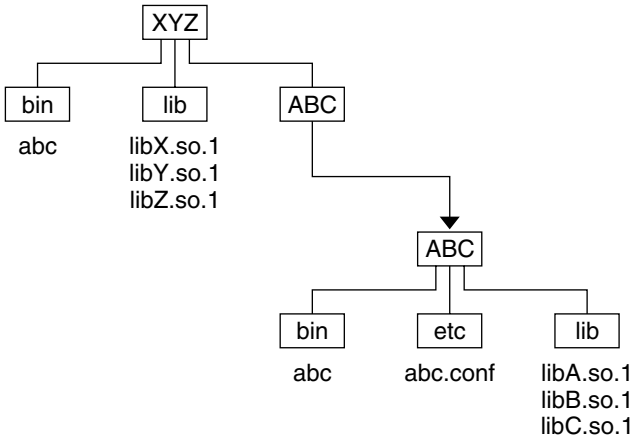
注 – $ORIGIN トークンを含むオブジェクトは、シンボリックリンクを使用すると参照できます。この場合、オブジェクトの真の起点を判断するために、シンボリックリンクは完全に解決されます。

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 10–2 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
$ elfdump -d xyz | egrep 'NEEDED|RUNPATH'
       [0]  NEEDED            0x1b5              libX.so.1
       [1]  NEEDED            0x1bf              libA.so.1
       ....
       [2]  NEEDED            0x18f              libc.so.1
       [5]  RUNPATH           0x1c9              $ORIGIN/../lib:$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
$ elfdump -d libX.so.1 | egrep 'NEEDED|RUNPATH'
       [0]  NEEDED            0x96               libY.so.1
       [1]  NEEDED            0xa0               libC.so.1
       [5]  RUNPATH           0xaa               $ORIGIN:$ORIGIN/../ABC/lib
```

次にこの製品が /usr/local/XYZ 内にインストールされた場合、インストール後のスクリプトによってシンボリックリンクを確立する必要があります。

```
$ ln -s ../ABC /usr/local/XYZ/ABC
```

ユーザーの PATH に /usr/local/XYZ/bin が追加されると、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検索されます。

```
$ 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
```

注 – オブジェクトの起点は、実行時に RTLD_DI_ORIGIN フラグが付いた dlinfo(3C) を使用すると取得できます。この起点のパスを使用すると、関連製品の階層から追加ファイルにアクセスできます。

セキュリティー

セキュアなプロセスでは、$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 を使用するセキュアな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 がセキュアな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、セキュアな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。

◆ ◆ ◆ 第 11 章

11 拡張性メカニズム

リンカーおよび実行時リンカーは、リンカーおよび実行時リンカーの処理を監視および変更できるインタフェースを提供しています。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」 – 273 ページの「リンカーのサポートインタフェース」
- 「rtld-監査」 – 281 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」 – 294 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。

このセクションでは、「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="-Ssupport.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、274 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。

32 ビットおよび 64 ビット環境

31 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バージョン LD_SUP_VNONE を戻す場合、リンカーは暗黙のうちにサポートライブラリの読み込みを解除して、このライブラリを使用しないで処理を進めます。戻されたバージョンが、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大きいと、致命的エラーが発生し、リンカーは実行を終了します。それ以外の場合、指定された ld-サポートインタフェースバージョンのサポートライブラリを使用して、実行は継続します。

ld_start()
: この関数は、リンカーコマンド行の初期妥当性検証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 `-z strip-class` 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。345 ページの「COMDAT セクション」と表 12–8 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、プロセスの実行時リンクに関する情報にアクセスできます。「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。
- プロシージャーのリンク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。関数呼び出しとその戻り値を監査できます。425 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。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。

この情報のいくつかは、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取得できます。しかし、事前に読み込まれたオブジェクトは、アプリケーション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、監査ライブラリがアプリケーション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。

rtld-監査インタフェースの使用例として、200 ページの「共有オブジェクトのプロファイリング」で説明した共有オブジェクトの実行時プロファイリングがあります。

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、アプリケーションを記述します。リンクマップ構造は、アプリケーション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の結合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全体を検索します。このリンクマップリストは、アプリケーション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、実行時リンカーは固有の名前空間に存在するため、アプリケーションは実行時

リンカー内のサービスを参照したり、直接アクセスしたりできません。したがって、アプリケーションは libc.so.1 または libdl.so.1 が提供するフィルタを介してのみ実行時リンカーにアクセスできます。

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 */
```

実行時リンカーを使用すると、これら 2 つの標準リンクマップリストに加えて、任意の数の追加リンクマップリストを作成できます。これらの追加リンクマップリストでは、それぞれに固有の名前空間が提供されます。rtld 監査インタフェースは、監査ライブラリを保持するために独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。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。

監査ライブラリでは、dlmopen(3C) を使用するとアプリケーションのリンクマップリストを検査できます。RTLD_NOLOAD フラグを指定して dlmopen() を使用すると、監査ライブラリはオブジェクトを読み込まずにそのオブジェクトの有無を問い合わせることができます。

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。

監査ライブラリが外部インタフェースを参照している場合、監査ライブラリではインタフェースを定義する依存関係を定義する必要があります。たとえば、監査ライブラリが printf(3C) を呼び出す場合、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。54 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ム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 1 つまたは複数の監査プログラムを定義することで有効になります。284 ページの「ローカル監査の記録」を参照してください。この方法で実行時に使用可能になる監査ライブラリには、ローカル監査を要求した動的オブジェクトに関する情報が提供されます。
- 大域監査は、環境変数 LD_AUDIT を使用して 1 つまたは複数の監査プログラムを定義することで有効になります。また、ローカル監査定義と -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。284 ページの「大域監査の記録」を参照してください。これらの方法により実行時に使用可能になる監査ライブラリには、アプリケーションが使用するすべての動的オブジェクトに関する情報が提供されます。

監査プログラムを定義する両方のメソッドでは、dlmopen(3C) によって読み込まれる、コロン区切りの共有オブジェクトリストから構成される文字列を使用します。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。

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
$ elfdump -d libfoo.so.1 | grep AUDIT
       [2]  AUDIT            0x96               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
$ elfdump -d main | grep AUDIT
       [4]  DEPAUDIT         0x1be              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
$ elfdump -d main | grep AUDIT
       [3]  DEPAUDIT         0x1b2              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、この環境変数を使用すると、アプリケーションの main とそのアプリケーション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
$ elfdump -d main | grep AUDIT
       [3]  DEPAUDIT          0x1b2              audit.so.1
      [26]  FLAGS_1           0x1000000          [ 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、アプリケーション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対話

監査ルーチンには 1 つまたは複数の *cookie* が設定されます。cookie とは、個々の動的オブジェクトを記述するデータ項目です。最初の cookie が `la_objopen()` ルーチンに設定されるのは、動的オブジェクトが最初に読み込まれたときです。この cookie は、読み込まれた動的オブジェクトの関連した `Link_map` に対するポインタです。しかし、`la_objopen()` ルーチンは自由に別の cookie を割り当てたり、実行時リンカーに返したりできます。このメカニズムにより、監査プログラムは各動的オブジェクトとともに独自のデータを保持し、後続のすべての監査ルーチン呼び出しでこのデータを受け取れます。

「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。この場合、ある監査プログラムからの戻り情報は次の監査プログラムの同じ監査ルーチンに渡されます。同様に、ある監査プログラムが作成した cookie は、次の監査プログラムに渡されます。ほかの監査ライブラリとの共存が想定される監査ライブラリを設計する際には注意が必要です。安全に実行するために、実行時リンカーによって通常返される結合または cookie を変更しないでください。これらのデータを変更すると、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。変更する場合、結合または cookie の情報を変更しても監査プログラムが安全に連携できるように、監査プログラムを設計する必要があります。

監査インタフェースの関数

次のルーチンが rtld-監査インタフェースによって提供されます。これらのルーチンは予想される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()`
: このルーチンは、実行時リンカーと監査ライブラリの間の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。

残りの監査ルーチンには 1 つ以上の *cookie* が渡されます。285 ページの「監査インタフェースの対話」を参照してください。

la_version() の呼び出しに続いて、la_objopen() ルーチンが 2 回呼び出されます。最初の呼び出しでは動的実行可能ファイルのリンクマップ情報が渡され、2 回目の呼び出しでは実行時リンカーのリンクマップ情報が渡されます。

la_objopen()
このルーチンは、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きに呼び出されます。

```
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。オブジェクトをほかの rtld-監査インタフェースルーチンで識別しやすくするために、監査ライブラリでこの識別子を再割り当てすることができます。

la_objopen() ルーチンは、このオブジェクトで関心があ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。

監査バージョン LAV_VERSION5 を指定すると、動的実行可能ファイルを表す la_objopen() 呼び出しがローカル監査プログラムで行われます。この場合、監査プログラムはシンボル結合フラグを返さないでください。監視プログラムの読み込みが遅すぎるために、動的実行可能ファイルに関連するシンボル結合を監視できない場合があるためです。監査プログラムによって返されるフラグはすべて無視されます。la_objopen() 呼び出しは、ローカル監査プログラムに、後続の la_preinit() または la_activity() 呼び出しに必要な初期 cookie を提供します。

la_activity()
このルーチンは、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。

動的実行可能ファイルおよび実行時リンカーの la_objopen() 呼び出しに続いて、LA_ACT_ADD アクティビティーがプロセス起動時に呼び出されて、新しい依存関係が追加されることが示されます。このアクティビティーは、遅延読み込みと dlopen(3C) イベントの場合にも呼び出されます。LA_ACT_DELETE アクティビティーは、dlclose(3C) でオブジェクトが削除されたときにも呼び出されます。

LA_ACT_ADD および LA_ACT_DELETE アクティビティーは、後に続くことが予想されるイベントのヒントです。実際のイベントが異なる場合が多数あります。たとえば、新しいオブジェクトが追加された場合にそれらのオブジェクトを完全に再配置できないときは、新しいオブジェクトのいくつかが削除されることがあります。.fini 実行可能ファイルによって新しいオブジェクトが遅延読み込みになった場合、オブジェクトの削除によって新しいオブジェクトが追加されることもあります。オブジェクトの追加または削除の後に、アプリケーションのリンクマップリストが一貫していることを示すために LA_ACT_CONSISTENT アクティビティーが発生します。このアクティビティーは信頼できます。監査プログラムは、無条件に LA_ACT_ADD および LA_ACT_DELETE のヒントを信じるのではなく、実際の結果を注意深く検証する必要があります。

監査バージョン LAV_VERSION1 から LAV_VERSION4 の場合、la_activity() は大域監査でのみ呼び出されていました。監査バージョンが LAV_VERSION5 の場合、ローカル監査によってアクティビティーのイベントを取得できます。アクティビティーのイベントは、アプリケーションのリンクマップを表す cookie を提供しま

す。このアクティビティーを準備し、監査プログラムがこの cookie の内容を制御できるようにするため、最初にローカル監査の `la_objopen()` が呼び出されます。`la_objopen()` 呼び出しによって、アプリケーションのリンクマップを表す初期 cookie が提供されます。285 ページの「監査インタフェースの対話」を参照してください。

`la_objsearch()`
このルーチンは、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アな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_objfilter()`
このルーチンは、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。145 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()`
このルーチンは、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。

監査バージョン `LAV_VERSION1` から `LAV_VERSION4` の場合、`la_preinit()` は大域監査でのみ呼び出されていました。監査バージョンが `LAV_VERSION5` の場合、ローカル監査によって preinit イベントを取得できます。preinit イベントは、アプリケーションのリンクマップを表す cookie を提供します。この preinit を準備し、監査プログラムがこの cookie の内容を制御できるようにするため、最初にローカル監査の `la_objopen()` が呼び出されます。`la_objopen()` 呼び出しによって、アプリケーションのリンクマップを表す初期 cookie が提供されます。285 ページの「監査インタフェースの対話」を参照してください。

`la_symbind()`
このルーチンは、`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したときに呼び出されます。

```
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()` ルーチン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() ルーチンが存在する場合、これらのルーチンは、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。これらのルーチンは、シンボルが参照されるたびに呼び出されます。詳細は、293 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() ルーチンを呼び出さない。
- LA_SYMB_NOPLTEXIT – このシンボルに対して la_pltexit() ルーチンを呼び出さない。

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()
これらのルーチンはシステムに固有です。これらのルーチンは、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()
このルーチンは、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。このルーチンは、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、293 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()
このルーチンは、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

rtld-audit インタフェースを使用するデモアプリケーションは、`/usr/demo/link_audit` の下の `SUNWosdem` パッケージにあります。

`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)` および `whocalls(1)` は `SUNWtoo` パッケージに含まれています。`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。

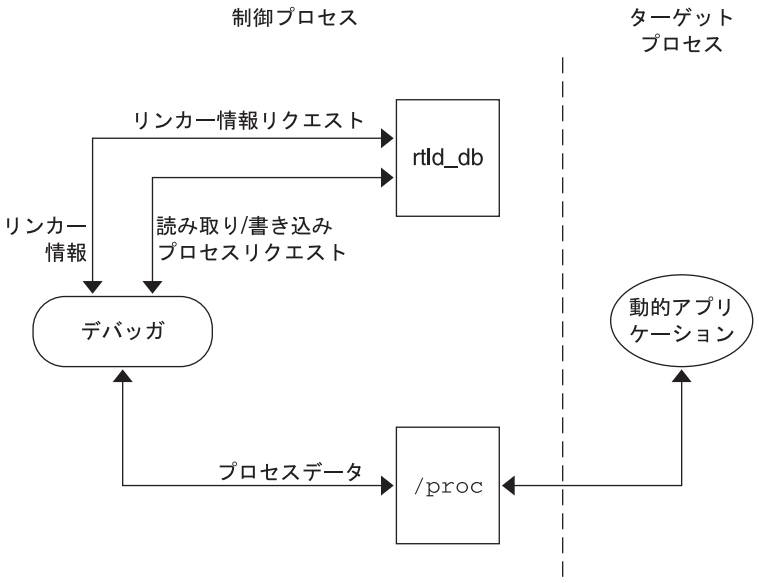
- ターゲットプロセス内のシンボルの検索。
- ターゲットプロセスのメモリーの読み取りと書き込み。

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、305 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 11–1 「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-debugger 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は、/usr/demo/librtld_db の下の SUNWosdem パッケージにあります。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、298 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、302 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;
        unsigned long   rl_tlsmodid;
} 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 (281 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 (304 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 (304 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 13–8 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。

rl_tlsmodid
: このフィールドは、RD_VERSION4 に追加されたもので、スレッド固有ストレージ (TLS) 参照のモジュール識別子を提供します。このモジュール識別子は、オブジェクトに固有の短い整数です。この識別子は、問題となるオブジェクトのスレッドの TLS ブロックの基底アドレスを取得するために、libc_db の関数 td_thr_tlsbase() に渡すことができます。 td_thr_tlsbase(3C_DB) を参照してください。

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) を PROT_NONE アクセス権と MAP_NORESERVE フラグで使用して、メモリー割り当てとして予約されています。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 – /dev/zero から mmap(2) を MAP_NORESERVE で使用して取得した予約は、proc(1) 機能を使用し、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レポートできます。

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。

パート IV

ELF アプリケーションバイナリインタフェース

汎用 ELF 形式は、*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定義されます。この参照ドキュメントには、汎用バージョンの情報と、Oracle Solaris の拡張機能によって拡張された情報が含まれています。

◆ ◆ ◆ 第 12 章

12 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。

プログラムは、ELF アクセスライブラリ `libelf` によって提供されている関数を含むオブジェクトファイルを操作できます。`libelf` の内容の説明については、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 12–1 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、Oracle 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 12–1 と表 12–2 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エンコードされています。

表 12–1　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 12–2　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、316 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
ET_CORE	4	コアファイル
ET_LOPROC	0xff00	プロセッサ固有の範囲の開始
ET_HIPROC	0xffff	プロセッサ固有の範囲の終了

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 12–6 および表 12–7 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 12–6 および表 12–7 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 12–6 および表 12–7 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、複数のデータエンコード、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 12–3　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	データのエンコード
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

名前	値	意味
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 12–1 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 12–2 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エンコードを指定します (次の表を参照)。

名前	値	意味
ELFDATANONE	0	無効なエンコード
ELFDATA2LSB	1	図 12–2 を参照してください。
ELFDATA2MSB	2	図 12–3 を参照してください。

これらのエンコードの詳細は、319 ページの「データのエンコード」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。

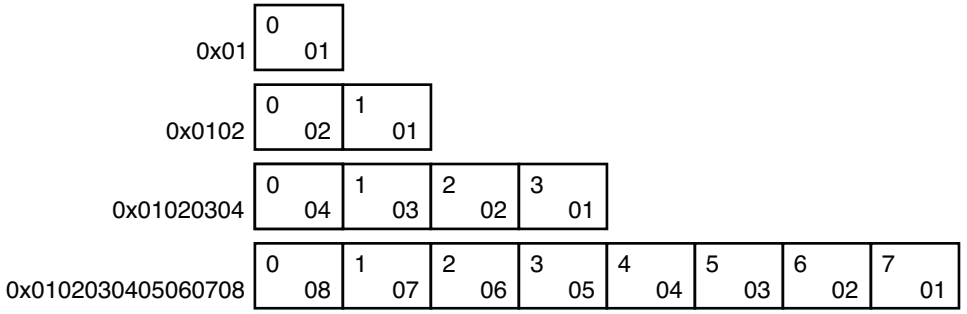
EI_PAD
: この値は、e_ident の使用されていないバイトの先頭を示します。これらのバイトは保留され、0 に設定されます。オブジェクトファイルを読み取るプログラムは、これらの値を無視します。

データのエンコード

ファイルのデータエンコード方式は、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。定義されているエンコード方式の下では、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。

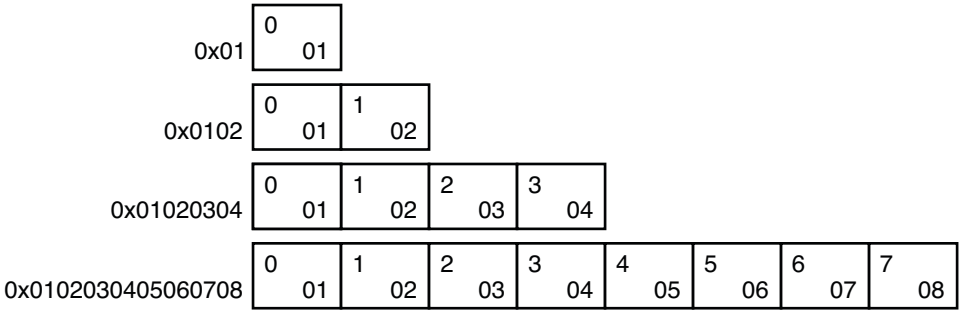
ELFDATA2LSB をエンコードすると、最下位バイトが最低位アドレスを占める 2 の補数値が指定されます。このエンコードは、一般的にはよく「リトルエンディアン」と呼ばれます。

図 12–2 データのエンコード方法 ELFDATA2LSB



ELFDATA2MSB をエンコードすると、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。このエンコードは、一般的にはよく「ビッグエンディアン」と呼ばれます。

図 12–3 データのエンコード方法 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 12–4 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
SHN_BEFORE	0xff00

表 12–4 ELF セクションの特殊インデックス (続き)

名前	値
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、このセクション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 12–8 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 12–10 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 12–5 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 12–8 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 12–9 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 12–9 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 12–5 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

表 12–5 ELFセクションタイプ、*sh_type* (続き)

名前	値
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	0x6fffffef
SHT_SUNW_capchain	0x6fffffef
SHT_SUNW_capinfo	0x6ffffff0
SHT_SUNW_symsort	0x6ffffff1
SHT_SUNW_tlssort	0x6ffffff2
SHT_SUNW_LDYNSYM	0x6ffffff3
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 12–5 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、SHT_SUNW_LDYNSYM
: 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) が含まれていることがあります。

  SHT_DYNSYM は、SHT_SUNW_LDYNSYM セクションで拡張することもできます。この追加セクションは、局所関数シンボルを実行時環境に提供しますが、動的リンクには必要ありません。このセクションを追加することで、SHT_SYMTAB を割り当てることができないために、テーブルが使用できないまたはファイルから削除されたときでも、デバッガは実行時状況で正確なスタックトレースを行うことができます。また、このセクションは、dladdr(3C) が使用する追加シンボリック情報を実行時環境に提供します。

SHT_SUNW_LDYNSYM セクションと SHT_DYNSYM セクションの両方があるときは、リンカーはそれらのデータ領域を並べて配置します。SHT_SUNW_LDYNSYM セクションは SHT_DYNSYM セクションの前に配置されます。このように配置されることで、SHT_SYMTAB の追加シンボルを含めて、これらの 2 つのテーブルを大きな 1 つの連続したシンボルテーブルとして表示することができます。

詳細は、371 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、369 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、356 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、351 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、407 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、354 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、356 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、45 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、45 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、45 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、346 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、Oracle 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_capchain
機能ファミリメンバーを集めたインデックスの配列です。配列の最初の要素は連鎖バージョンメンバーです。この要素の後に来るのは、0 で終了する一連の機能シンボルインデックスです。インデックスの 0 で終了する各グループは機能ファミリを表します。各ファミリの最初の要素は機能の先頭のシンボルです。次の要素はファミリメンバーを指します。詳細については、347 ページの「機能セクション」を参照してください。

SHT_SUNW_capinfo
シンボルテーブルエントリを機能要件に関連付けるインデックスの配列と、その先頭の機能シンボルです。シンボル機能を定義するオブジェクトには SHT_SUNW_cap セクションが含まれています。SHT_SUNW_cap セクションのヘッダー情報は、関連する SHT_SUNW_capinfo セクションを指します。SHT_SUNW_capinfo セクションのヘッダー情報は、関連するシンボルテーブルセクションを指します。詳細については、347 ページの「機能セクション」を参照してください。

SHT_SUNW_symsort
並んで配置される SHT_SUNW_LDYNSYM セクションと SHT_DYNSYM セクションによって作成される動的なシンボルテーブルへのインデックスの配列。これらのインデックスは、SHT_SUNW_LDYNSYM セクションの開始位置からの相対値です。インデックスはメモリーアドレスを含み、上記のシンボルを参照します。インデックスは、アドレスの昇順にシンボルを参照するようにソートされます。

SHT_SUNW_tlssort
並んで配置される SHT_SUNW_LDYNSYM セクションと SHT_DYNSYM セクションによって作成される動的なシンボルテーブルへのインデックスの配列。これらのインデックスは、SHT_SUNW_LDYNSYM セクションの開始位置からの相対値です。インデックスは、スレッド固有ストレージシンボルを参照します。第 14 章「スレッド固有ストレージ (TLS)」を参照してください。インデックスは、オフセットの昇順にシンボルを参照するようにソートされます。

SHT_SUNW_LDYNSYM
非大域シンボル用の動的シンボルテーブル。すでに説明した「SHT_SYMTAB、SHT_DYNSYM、SHT_SUNW_LDYNSYM」を参照してください。

SHT_SUNW_dof
dtrace(1M) の内部使用のため予約されています。

SHT_SUNW_cap
機能要件を指定します。詳細については、347 ページの「機能セクション」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 -z strip-class オプションを使用するか、あるいはリンク編集後に strip(1) を使用すると、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、352 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、345 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、383 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、385 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、387 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、389 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 12–6 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 12–7 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 12–7 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 12–8 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_SUNW_NODISCARD	0x00100000

表 12–8 ELF セクションの属性フラグ (続き)

名前	値
SHF_MASKPROC	0xf0000000
SHF_AMD64_LARGE	0x10000000
SHF_ORDERED	0x40000000
SHF_EXCLUDE	0x80000000

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、リンク先のセクションは、それが結合されるセクションに現われます。リンク先のセクションは順不同でなければならず、その結果 SHF_LINK_ORDER または SHF_ORDERED を指定することはできません。

特殊な sh_link 値である SHN_BEFORE および SHN_AFTER (表 12–4 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またはあと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、346 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 14 章「スレッド固有ストレージ (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_SUNW_NODISCARD
: このセクションは、リンカーによって破棄されず、常に出力オブジェクトにコピーされます。リンカーには、使用されない入力セクションをリンク編集から破棄する機能が用意されています。SHF_SUNW_NODISCARD セクションフラグは、このような最適化からセクションを除外します。

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。

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
: SHF_ORDERED は SHF_LINK_ORDER が提供する機能の古いバージョンであり、SHF_LINK_ORDER に置き換えられました。 SHF_ORDERED は 2 つの異なる機能を提供します。1 つは、出力セクションを指定できる機能、もう 1 つは、リンカーから特別な順序付け要件を要求する機能です。

  SHF_ORDERED セクションの sh_link フィールドはセクションのリンクリストを形成します。このリストは、自分自身を指す sh_link を持つ最終セクションで終了します。このリストにあるすべてのセクションは、リストの最終セクションの名前を使用して、出力セクションに割り当てられます。

  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。sh_info エントリによって指されたセクションは順不同でなければならず、その結果、SHF_LINK_ORDER または SHF_ORDERED を指定することができません。

  特殊な sh_info 値である SHN_BEFORE および SHN_AFTER (表 12–4 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またはあと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。

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 12–9 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 12–10 と 356 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_cap	シンボル機能が存在する場合、関連する SHT_SUNW_capinfo テーブルのセクションヘッダーインデックス。そうでない場合は 0。	機能が名前付きの文字列を参照する場合、関連する文字列テーブルのセクションヘッダーインデックス。そうでない場合は 0。

表 12–9 ELF sh_link と sh_info の解釈 (続き)

sh_type	sh_link	sh_info
SHT_SUNW_capinfo	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。	動的オブジェクトの場合、関連する SHT_SUNW_capchain テーブルのセクションヘッダーインデックス。そうでない場合は 0。
SHT_SUNW_symsort	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_tlssort	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_LDYNSYM	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。このインデックスは、SHT_DYNSYM セクションで使用される文字列テーブルと同じです。	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_SUNW_LDYNSYM には局所シンボルしか含まれないので、sh_info はテーブル内のシンボル数に等しくなります。
SHT_SUNW_move	関連付けられているシンボルテーブルのセクションヘッダーインデックス。	0
SHT_SUNW_COMDAT	0	0
SHT_SUNW_syminfo	関連付けられているシンボルテーブルのセクションヘッダーインデックス。	関連付けられている .dynamic セクションのセクションヘッダーインデックス。
SHT_SUNW_verdef	関連付けられている文字列テーブルのセクションヘッダーインデックス。	セクション内のバージョン定義数。
SHT_SUNW_verneed	関連付けられている文字列テーブルのセクションヘッダーインデックス。	セクション内のバージョン依存数。
SHT_SUNW_versym	関連付けられているシンボルテーブルのセクションヘッダーインデックス。	0

セクションのマージ

SHF_MERGE セクションフラグを使って、再配置可能オブジェクト内の SHT_PROGBITS セクションにマークすることができます。表 12–8 を参照してください。このフラグによって、そのセクションがほかのオブジェクトの互換セクションとマージできることがわかります。これらの再配置可能オブジェクトをマージすることによって、それらから構築される実行可能ファイルや共有オブジェクトのサイズを減らせる可能性があります。サイズを減らすことによって、最終オブジェクトの実行時パフォーマンスが向上することもあります。

セクションに SHF_MERGE フラグが付いていることは、次のような特性があることを示します。

- セクションは読み取り専用です。このセクションを含むプログラムは、実行時にセクションデータを変更することはできません。
- セクション内の各項目はそれぞれの再配置レコードからアクセスされます。プログラムコードは、セクション内の項目にアクセスするコードを生成するときに、それらの項目の相対位置についていかなる仮定を行なってはいけません。
- セクションに SHF_STRINGS フラグも設定されている場合は、そのセクションには NULL で終わっている文字列だけを含めることができます。NULL 文字は文字列の最後にしか使用できません。文字列の途中に NULL 文字が現れてはいけません。

SHF_MERGE は、最適化できる可能性があることを示す、省略可能なフラグです。リンカーは、最適化を実行するか最適化を無視するかを選択できます。リンカーは、いずれの場合も有効な出力オブジェクトを作成します。リンカーは現在のところ、SHF_STRINGS フラグがマークされた文字列データを含むセクションだけに、セクションのマージを実装します。

SHF_STRINGS セクションフラグと一緒に SHF_MERGE フラグも設定されているときは、そのセクション内の文字列はほかの互換セクション内の文字列とマージできます。リンカーは、SHT_STRTAB 文字列テーブル .strtab と .dynstr の圧縮に使用される文字列圧縮アルゴリズムを使用して、このようなセクションをマージします。

- 重複する文字列は 1 つだけがコピーされます。
- 末尾の文字列は削除されます。たとえば、入力セクションに文字列「bigdog」と「dog」が含まれる場合は、短いほうの「dog」文字列が削除され、長い文字列の末尾を使って短い文字列が表現されます。

リンカーは現在のところ、バイトサイズ文字から構成され、特殊な整列制約のない文字列だけに、文字列のマージを実装します。具体的には、セクションは次の特性を備えている必要があります。

- sh_entsize は 0 または 1 でなければいけません。ワイド文字を含むセクションはサポートされません。

- sh_addralign が 0 または 1 であるバイト整列のセクションのみがマージされます。

注 – リンカーの -z nocompstrtab オプションを使って文字列テーブルの圧縮を抑制できます。

特殊セクション

さまざまなセクションがプログラム情報と制御情報を保持します。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 12–10 ELF 特殊セクション

名前	タイプ	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
.fini_array	SHT_FINI_ARRAY	SHF_ALLOC + SHF_WRITE
.got	SHT_PROGBITS	424 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
.init_array	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
.interp	SHT_PROGBITS	406 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

表 12–10 ELF特殊セクション (続き)

名前	タイプ	属性
.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	425 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
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
.rela	SHT_RELA	None
.rel*name*	SHT_REL	356 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	356 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	371 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	371 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_capchain	SHT_SUNW_capchain	SHF_ALLOC
.SUNW_capinfo	SHT_SUNW_capinfo	SHF_ALLOC
.SUNW_heap	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.SUNW_ldynsym	SHT_SUNW_LDYNSYM	SHF_ALLOC

表 12–10 ELF 特殊セクション (続き)

名前	タイプ	属性
.SUNW_dynsymsort	SHT_SUNW_symsort	SHF_ALLOC
.SUNW_dymtlssort	SHT_SUNW_tlssort	SHF_ALLOC
.SUNW_move	SHT_SUNW_move	SHF_ALLOC
.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、407 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、371 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、112 ページの「初期設定および終了ルーチン」を参照してください。

.fini_array
このセクション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、112 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、424 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、351 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、112 ページの「初期設定および終了ルーチン」を参照してください。

.init_array
このセクション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、112 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、406 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
354 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、425 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。

.preinit_array
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、112 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、382 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、356 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、393 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、371 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 14 章「スレッド固有ストレージ (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 14 章「スレッド固有ストレージ (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
機能要件。詳細については、347 ページの「機能セクション」を参照してください。

.SUNW_capchain
機能連鎖テーブル。詳細については、347 ページの「機能セクション」を参照してください。

.SUNW_capinfo
機能シンボル情報。詳細については、347 ページの「機能セクション」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_dynsymsort
.SUNW_ldynsym と .dynsym の結合シンボルテーブル内のシンボルへのインデックスの配列。インデックスは、アドレスの昇順にシンボルを参照するようにソートされます。変数を表現しないシンボルまたは関数を表現しないシンボルは取り込まれません。大域シンボルとウィークシンボルが冗長な場合には、ウィークシンボルのみが残ります。詳細は、380 ページの「シンボルソートセクション」を参照してください。

.SUNW_dyntlssort
.SUNW_ldynsym と .dynsym の結合シンボルテーブル内のスレッド固有ストレージシンボルへのインデックスの配列。インデックスは、オフセットの昇順にシンボルを参照するようにソートされます。TLS 変数を表現しないシンボルは取り込まれません。大域シンボルとウィークシンボルが冗長な場合には、ウィークシンボルのみが残ります。詳細は、380 ページの「シンボルソートセクション」を参照してください。

.SUNW_ldynsym
.dynsym セクションを拡張します。このセクションには、完全な .symtab セクションを使用できない状況で使用される局所関数シンボルが含まれます。リンカーは常に、.SUNW_ldynsym セクションのデータを .dynsym セクションの直前に並べて配置します。両方のセクションは常に同じ .dynstr 文字列テーブルセクションを使用します。この配置と構成により、両方のシンボルテーブルを 1 つの

大きなシンボルテーブルとして処理することができます。371 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。

`.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、352 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、356 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、383 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、385 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 12–11　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。

機能セクション

SHT_SUNW_cap セクションはオブジェクトの機能要件を指定します。これらの機能はオブジェクト機能と呼ばれます。このセクションは、オブジェクト内の関数の機能要件または初期化されたデータ項目も指定できます。これらの機能はシンボル機能と呼ばれます。このセクションには、次の構造の配列が含まれます。sys/elf.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 12–12　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_HW_2	3	c_val
CA_SUNW_PLAT	4	c_ptr
CA_SUNW_MACH	5	c_ptr
CA_SUNW_ID	6	c_ptr

CA_SUNW_NULL
　機能グループの最後にマークを付けます。

CA_SUNW_HW_1、CA_SUNW_HW_2
　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。

CA_SUNW_PLAT
　プラットフォーム名を指定します。c_ptr 要素は、プラットフォーム名を定義する、ヌル文字で終わる文字列の文字列テーブルオフセットを含みます。

CA_SUNW_MACH
　マシン名を指定します。c_ptr 要素は、マシンハードウェア名を定義する、ヌル文字で終わる文字列の文字列テーブルオフセットを含みます。

CA_SUNW_ID
: 機能識別子名を指定します。c_ptr 要素は、識別子名を定義する、ヌル文字で終わる文字列の文字列テーブルオフセットを含みます。この要素は機能を定義しませんが、機能グループの参照に使用できる固有のシンボリック名を機能グループに割り当てます。この識別子名は、リンカーの -z symbolcap 処理の一部として局所シンボルに変換される、すべての大域シンボル名に追加されます。81 ページの「オブジェクト機能のシンボル機能への変換」を参照してください。

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。66 ページの「機能要件の特定」を参照してください。

CA_SUNW_NULL で終了する複数の機能グループがオブジェクト内に存在できます。最初のグループ (インデックス 0 から開始) はオブジェクト機能を指定します。オブジェクト機能を定義する動的オブジェクトには、セクションに関連した PT_SUNWCAP プログラムヘッダーがあります。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システム機能に対してオブジェクトを確認できます。異なるオブジェクト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環境を提供できます。263 ページの「機能固有の共有オブジェクト」を参照してください。

追加の機能グループはシンボル機能を指定します。シンボル機能によって、同じシンボルの複数のインスタンスがオブジェクト内に存在できます。各インスタンスは、そのインスタンスを使用するために利用できなければならない一連の機能に関連付けられています。シンボル機能が存在する場合、SHT_SUNW_cap セクションの sh_link 要素は関連の SHT_SUNW_capinfo テーブルを指します。シンボル機能を利用する動的オブジェクトでは、特定のシステム向けに最適化された関数を柔軟に有効化できます。75 ページの「シンボル機能関数ファミリの作成」を参照してください。

SHT_SUNW_capinfo テーブルは関連のシンボルテーブルに相当します。SHT_SUNW_capinfo セクションの sh_link 要素は関連のシンボルテーブルを指します。機能に関連付けられた関数は、SHT_SUNW_cap セクション内の機能グループを指定する SHT_SUNW_capinfo テーブル内にインデックスを持ちます。

動的オブジェクト内では、SHT_SUNW_capinfo セクションの sh_info 要素は機能連鎖テーブルである SHT_SUNW_capchain を指します。このテーブルは、実行時リンカーが機能ファミリのメンバーの位置を特定するために使用されます。

SHT_SUNW_capinfo 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 Elf32_Word    Elf32_Capinfo;
typedef Elf64_Xword   Elf64_Capinfo;
```

このテーブル内の要素は、次のマクロを使用して解釈されます。sys/elf.h を参照してください。

```
#define ELF32_C_SYM(info)       ((info)>>8)
#define ELF32_C_GROUP(info)     ((unsigned char)(info))
#define ELF32_C_INFO(sym, grp)  (((sym)<<8)+(unsigned char)(grp))

#define ELF64_C_SYM(info)       ((info)>>32)
#define ELF64_C_GROUP(info)     ((Elf64_Word)(info))
#define ELF64_C_INFO(sym, grp)  (((Elf64_Xword)(sym)<<32)+(Elf64_Xword)(grp))
```

SHT_SUNW_capinfo エントリの group 要素には、このシンボルに関連する SHT_SUNW_cap テーブルのインデックスが含まれています。このようにして、この要素はシンボルを機能グループに関連付けます。予約済みのグループインデックスである CAPINFO_SUNW_GLOB は、機能インスタンスファミリの先頭のシンボルを識別して、デフォルトのインスタンスを指定します。

名前	値	意味
CAPINFO_SUNW_GLOB	0xff	デフォルトのシンボルを特定します。このシンボルは特定の機能に関連付けられていませんが、シンボル機能ファミリの先頭になります。

SHT_SUNW_capinfo エントリの symbol 要素には、このシンボルに関連する先頭のシンボルのインデックスが含まれます。グループとシンボルの情報を使用すると、リンカーは再配置可能オブジェクトから機能シンボルファミリを処理し、すべての出力オブジェクトに必要な機能情報を作成できます。動的オブジェクトでは、グループ CAPINFO_SUNW_GLOB を使用してタグが付けられた先頭のシンボルのシンボル要素が、SHT_SUNW_capchain テーブル内へのインデックスです。このインデックスを使用すると、実行時リンカーは機能連鎖テーブルをこのインデックスからたどり、0 エントリが検出されるまでエントリを次々に検査できます。この連鎖エントリには、機能ファミリメンバーごとにシンボルインデックスが含まれています。

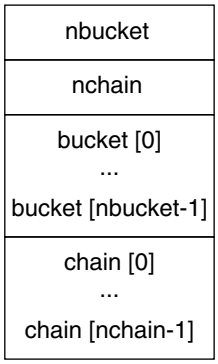
シンボル機能を定義している動的オブジェクトには、DT_SUNW_CAP 動的エントリと DT_SUNW_CAPINFO 動的エントリがあります。これらのエントリは、それぞれ SHT_SUNW_cap セクションと SHT_SUNW_capinfo セクションを特定します。オブジェクトには、SHT_SUNW_capchain セクション、セクションのエントリサイズ、および合計サイズを指定する DT_SUNW_CAPCHAIN、DT_SUNW_CAPCHAINENT、および DT_SUNW_CAPCHAINSZ の各エントリも含まれます。これらのエントリを使用すると、実行時リンカーはシンボル機能インスタンスファミリから、使用に最適なシンボルを確立できます。

オブジェクトは、オブジェクト機能のみ、シンボル機能のみ、または両方のタイプの機能を定義できます。オブジェクト機能グループはインデックス 0 から開始します。シンボル機能グループは 0 以外のインデックスから開始します。オブジェクトでシンボル機能が定義されているが、オブジェクト機能が定義されていない場合、シンボル機能の開始を示すためにインデックス 0 に 1 つの CA_SUNW_NULL エントリが存在する必要があります。

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 12–4 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]       0x44      4      1      1       0x45000000  move
```

```
[17]        0x40      4       1       1              0xe  move
[17]        0x34      4       1       1       0x45000000  move
[17]        0x30      4       1       1              0xe  move
[17]        0x1c      4       2       1       0x46000000  move
[17]        0x18      4       2       1              0xf  move
[17]         0xc      4       1       1       0x31000000  move
[17]         0x8      4       1       1              0x1  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 12–6 に示しますが、ラベルは仕様の一部ではありません。

図 12–5　注釈の情報

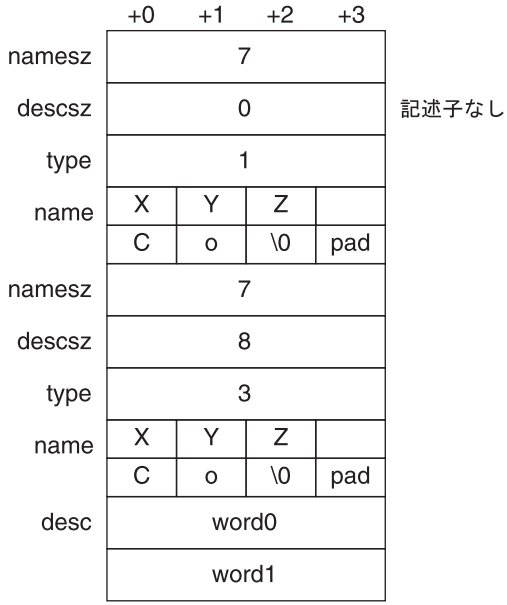
namesz と name
: 名前の先頭 namesz バイトには、エントリの所有者または作者を示す、ヌル文字で終わっている文字列が存在します。名前の競合を回避するための正式なメカニズムは存在しません。慣例では、ベンダーは識別子として自身の名前 (“XYZ Computer Company” など) を使用します。name がない場合、namesz の値は 0 になります。name の領域は、パッドを使用して、4 バイトに整列します。必要であれば namesz は、パッドの長さを含みません。

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 12–6 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ストレージを指定します。

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ストレージユニット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。320 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オプションの場合は、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。

いずれの場合でも r_offset 値は、影響を受けるストレージユニット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。

再配置計算

再配置計算を記述するために次の表記が使用されます。

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。393 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。424 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。424 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。425 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。

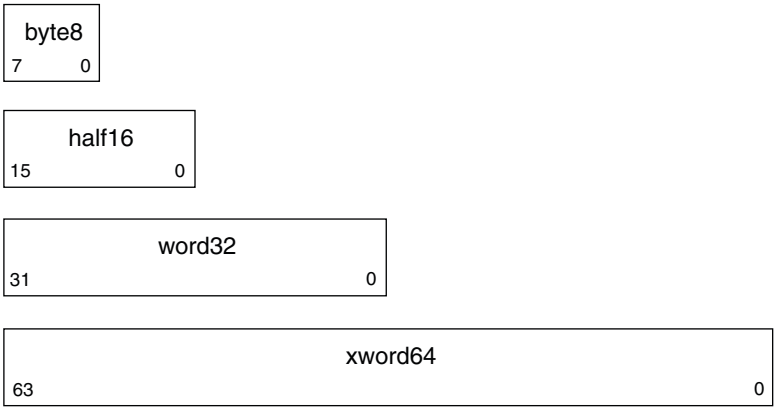
P　再配置されるストレージユニットのセクションオフセットまたはアドレス (r_offset を使用して計算)。

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。

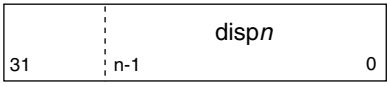
Z　インデックスが再配置エントリ内に存在するシンボルのサイズ。

SPARC: 再配置

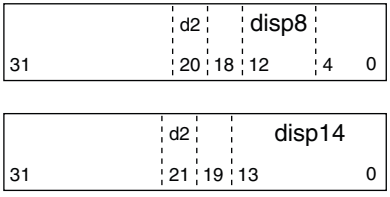
SPARC プラットフォームでは、再配置エントリはバイト (byte8)、ハーフワード (half16)、ワード (word32)、および拡張ワード (xword64) に適用されます。



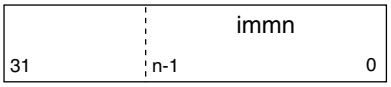
再配置フィールドの `disp`*n* ファミリ (`disp19`、`disp22`、`disp30`) は、ワード整列された符号拡張の PC 相対ディスプレイスメントです。すべてワードの位置 0 の最下位ビットで値をエンコードし、値に割り当てられたビット数についてのみ異なります。



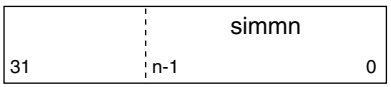
`d2/disp8` および `d2/disp14` バリアントは、連続しない 2 つのビットフィールド `d2` および `disp` *n* を使用して、16 および 10 ビットのディスプレイスメント値をエンコードします。



再配置フィールドの `imm`*n* ファミリ (`imm5`、`imm6`、`imm7`、`imm10`、`imm13`、`imm22`) は、符号なしの整数定数を表します。すべてワードの位置 0 の最下位ビットで値をエンコードし、値に割り当てられたビット数についてのみ異なります。



再配置フィールドの simm*n* ファミリ (simm10、simm11、simm13、simm22) は、符号付きの整数定数を表します。すべてワードの位置 0 の最下位ビットで値をエンコードし、値に割り当てられたビット数についてのみ異なります。



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 12–13 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

表 12–13 SPARC: ELF 再配置型 (続き)

名前	値	フィールド	計算
R_SPARC_GOT13	14	V-simm13	G
R_SPARC_GOT22	15	T-simm22	G >> 10
R_SPARC_PC10	16	T-simm13	(S + A - P) & 0x3ff
R_SPARC_PC22	17	V-disp22	(S + A - P) >> 10
R_SPARC_WPLT30	18	V-disp30	(L + A - P) >> 2
R_SPARC_COPY	19	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

表 12–13 SPARC: ELF 再配置型 (続き)

名前	値	フィールド	計算
.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
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	V-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。
R_SPARC_SIZE32	86	V-word32	Z + A
R_SPARC_WDISP10	88	V-d2/disp8	(S + A - P) >> 2

注 – スレッド固有ストレージ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 14 章「スレッド固有ストレージ (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。196 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、妥当性検証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、妥当性検証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。360 ページの「SPARC: 再配置型」 を参照してください。

表 12–14　64 ビット SPARC: ELF 再配置型

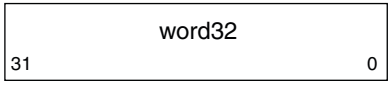
名前	値	フィールド	計算
R_SPARC_HI22	9	V-imm22	(S + A) >> 10
R_SPARC_GLOB_DAT	20	V-xword64	S + A
R_SPARC_RELATIVE	22	V-xword64	B + A
R_SPARC_64	32	V-xword64	S + A
R_SPARC_OLO10	33	V-simm13	((S + A) & 0x3ff) + O
R_SPARC_DISP64	46	V-xword64	S + A - P
R_SPARC_PLT64	47	V-xword64	L + A
R_SPARC_REGISTER	53	V-xword64	S + A
R_SPARC_UA64	54	V-xword64	S + A
R_SPARC_H34	85	V-imm22	(S + A) >> 12
R_SPARC_SIZE64	87	V-xword64	Z + A

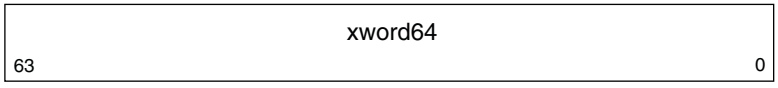
次の再配置型には、単純な計算を超えたセマンティクスが存在します。

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。

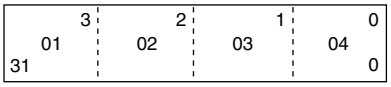
x86: 再配置

x86 では、再配置エントリはワード (word32) および拡張ワード (xword64) 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。



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 12–15　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
R_386_PC8	23	word8	S + A - P
R_386_SIZE32	38	word32	Z + A

注 – スレッド固有ストレージ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 14 章「スレッド固有ストレージ (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。196 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 12–16 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
R_AMD64_GOTOFF64	25	word64	S + A - GOT
R_AMD64_GOTPC32	26	word32	GOT + A + P

表 12–16 x64: ELF 再配置型 (続き)

名前	値	フィールド	計算
R_AMD64_SIZE32	32	word32	Z + A
R_AMD64_SIZE64	33	word64	Z + A

注 – スレッド固有ストレージ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 14 章「スレッド固有ストレージ (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、ドキュメント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

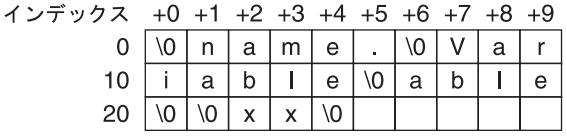
文字列テーブルセクションは、ヌル文字で終了する一連の文字 (一般に文字列と呼ばれている) を保持します。オブジェクトファイルは、これらの文字列を使用してシンボルとセクション名を表します。文字列をインデックスに使用して、文字列テーブルセクションを参照します。

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 12–7 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 12–17 ELF 文字列テーブルインデックス

インデックス	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 12–21 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。378 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 12–18 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 12–20 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 12–4 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 12–18 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。48 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をオーバーライドします。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。320 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 12–19 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、スレッド固有ストレージ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 14 章「スレッド固有ストレージ (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 12–20 ELF シンボルの可視性

名前	値
STV_DEFAULT	0
STV_INTERNAL	1
STV_HIDDEN	2
STV_PROTECTED	3
STV_EXPORTED	4
STV_SINGLETON	5
STV_ELIMINATE	6

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
: この可視性属性は STV_HIDDEN と同じものとして解釈されます。

STV_EXPORTED
: この可視性属性によって、シンボルのスコープが大域に維持されます。ほかのどのようなシンボル可視性テクニックを使っても、この可視性を降格または削除することはできません。STB_LOCAL 結合を持つシンボルは、STV_EXPORTED 可視性を持ちません。

STV_SINGLETON
: この可視性属性によって、シンボルのスコープが大域に維持され、プロセス内のすべての参照はシンボル定義の 1 つのインスタンスだけにバインドされます。ほかのどのようなシンボル可視性テクニックを使っても、この可視性を降格または削除することはできません。STB_LOCAL 結合を持つシンボルは、STV_SINGLETON 可視性を持ちません。STV_SINGLETON に直接バインドすることはできません。

STV_ELIMINATE
: この可視性属性は STV_HIDDEN を継承します。現在のコンポーネント内でこの属性が定義されたシンボルは、ほかのコンポーネントから見えません。このシンボル

は、そのコンポーネントを使用する動的実行可能ファイルまたは共有オブジェクトのシンボルテーブルには書き込まれません。

STV_SINGLETON 可視性属性は、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影響する可能性があります。プロセス内の参照には、シングルトンの 1 つのインスタンスだけをバインドできます。

STV_SINGLETON は STV_DEFAULT 可視性属性と一緒に使用できますが、STV_SINGLETON が優先されます。STV_EXPORT は STV_DEFAULT 可視性属性と一緒に使用できますが、STV_EXPORT が優先されます。STV_SINGLETON または STV_EXPORT 可視性は、それ以外の可視性属性とは一緒に使用できません。そのような場合、リンク編集にとって致命的とみなされます。

ほかの可視性の属性は、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用のストレージを割り当てます。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 12–21 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	
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。

Oracle Solaris OS には、3 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。63 ページの「シンボル削除」および 225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」を参照してください。

.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。

.SUNW_ldynsym (SHT_SUNW_LDYNSYM)
: .dynsym テーブル内で見つかる情報を拡張する省略可能なシンボルテーブル。.SUNW_ldynsym テーブルには局所関数シンボルが含まれます。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。このセクションを追加することで、.symtab を割り当てることができないために、テーブルが使用できないまたはファイルから削除されたときでも、デバッガは実行時状況で正確なスタックトレースを行うことができます。また、このセクションは、dladdr(3C) が使用する追加シンボリック情報を実行時環境に提供します。

.SUNW_ldynsym テーブルが存在するには、.dynsym テーブルが存在している必要があります。.SUNW_ldynsym セクションと .dynsym セクションの両方があるときは、リンカーはそれらのデータ領域を並べて配置します (.SUNW_ldynsym が最初)。このように配置されることで、2 つのテーブルを大きな 1 つの連続したシンボルテーブルとして表示することができます。このシンボルテーブルは、すでに説明した標準レイアウト規則に従います。

.SUNW_ldynsym テーブルを削除するには、リンカーの -z noldynsym オプションを使用します。

シンボルソートセクション

並んで配置される .SUNW_ldynsym セクションと .dynsym セクションによって作成される動的なシンボルテーブルを使って、メモリーアドレスを対応するシンボルにマッピングできます。このマッピングを使って、特定のアドレスがどの関数または変数を表現するかを判断できます。ただし、シンボルテーブルを解析してマッピングを判断することは、シンボルがシンボルテーブルに書き込まれる順番が原因で、複雑な作業になります。378 ページの「シンボルテーブルのレイアウトと規則」を参照してください。このレイアウトによって、アドレスをシンボル名に関連付ける作業は複雑になります。

- シンボルがアドレスでソートされていないため、テーブル全体を地道に上から順番に検索する必要があります。
- 特定のアドレスを複数のシンボルが参照していることがあります。これらのシンボルは有効で正しいのですが、それらの等価の名前のうち、デバッグツールがどれを使用するかを選択するかが明確でないことがあります。ツールごとに異なる名前が使用されることもあります。こうした問題によって、ユーザーが混乱する可能性があります。
- 多くのシンボルがアドレス以外の情報を提供しています。それらのシンボルを検索に含めるべきではありません。

これらの問題を解決するために、シンボルソートセクションを使用します。シンボルソートセクションは、Elf32_Word または Elf64_Word オブジェクトの配列です。この配列の各要素は、.SUNW_ldynsym と .dynsym の結合シンボルテーブルへのインデックスです。この配列の要素は、参照されるシンボルがソート順に提供されるようにソートされます。関数または変数を表現するシンボルのみが取り込まれます。ソート配列に関連付けられたシンボルは、 -S オプション付きで elfdump(1) に使用することで表示できます。

通常のシンボルとスレッド固有ストレージシンボルを一緒にソートすることはできません。通常のシンボルの値は、そのシンボルが参照している関数または変数のアドレスです。スレッド固有ストレージシンボルの値は、変数のスレッドオフセットです。したがって、通常のシンボルとスレッド固有ストレージシンボルでは、異なる 2 つのソートセクションが使用されます。

.SUNW_dynsymsort
SHT_SUNW_SYMSORT タイプのセクション。.SUNW_ldynsym と .dynsym の結合シンボルテーブル内の通常のシンボルへのインデックスが含まれます (アドレスでソート)。変数または関数を表現しないシンボルは取り込まれません。

.SUNW_dyntlssort
SHT_SUNW_TLSSORT タイプのセクション。.SUNW_ldynsym と .dynsym の結合シンボルテーブル内の TLS シンボルへのインデックスが含まれます (オフセットでソート)。このセクションは、オブジェクトファイルに TLS シンボルが含まれる場合にだけ作成されます。

リンカーは、ソートセクションがどのシンボルを参照するかを選択するために、次の規則を記載順に使用します。

- シンボルは関数タイプまたは変数タイプである必要がある: STT_FUNC、STT_OBJECT、STT_COMMON、または STT_TLS。
- 次のシンボルは常に取り込まれる (存在する場合): _DYNAMIC、_end、_fini、_GLOBAL_OFFSET_TABLE_、_init、_PROCEDURE_LINKAGE_TABLE_、および _start。
- 同じ項目を参照する大域シンボルとウィークシンボルが見つかった場合は、ウィークシンボルが取り込まれ、大域シンボルは除外される。
- シンボルは未定義であってはいけない。
- シンボルはゼロ以外のサイズである必要がある。

これらの規則によって、コンパイラとリンカーが自動的に生成するシンボルは除外されます。選択されるシンボルは、ユーザーに関係するものです。ただし、次の 2 つの場合には、選択処理を改善するために手動による介入が必要になる場合があります。

- 規則によって、必要とする特殊シンボルが選択されなかった場合。たとえば、サイズがゼロの特殊シンボルなど。
- 不要で余分なシンボルが選択される場合。たとえば、共有オブジェクトには、同じアドレスを参照し、同じサイズのシンボルを複数定義できます。これらの別名シンボルは同じ項目を参照することになります。ソートセクション内で、複数のシンボルファミリの 1 つだけを取り込みたい場合があります。

mapfile のキーワード DYNSORT および NODYNSORT により、シンボルをきめ細かく選択できます。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」を参照してください。

DYNSORT
ソートセクションに含める必要があるシンボルを指定します。シンボルタイプは STT_FUNC、STT_OBJECT、STT_COMMON、または STT_TLS である必要があります。

NODYNSORT
ソートセクションに含める必要があるシンボルを指定します。

たとえば、あるオブジェクトのシンボルテーブル定義が次のようになっているとします。

```
$ elfdump -sN.symtab foo.so.1 | egrep "foo$|bar$"
      [37]  0x000004b0 0x0000001c  FUNC GLOB  D   0 .text      bar
      [38]  0x000004b0 0x0000001c  FUNC WEAK  D   0 .text      foo
```

シンボル foo と bar は別名ペアを表現しています。デフォルトでは、ソートされた配列を作成するときに、シンボル foo だけが表現されます。

```
$ cc -o foo.so.1 -G foo.c
$ elfdump -S foo.so.1 | egrep "foo$|bar$"
      [13]  0x000004b0 0x0000001c  FUNC WEAK  D   0 .text      foo
```

リンカーによって同じ項目を参照する大域シンボルとウィークシンボルが検出された場合は、通常はウィークシンボルが選択されます。ウィークシンボル foo に関連付けられたので、シンボル bar はソートされた配列から除外されます。

次の mapfile を実行すると、シンボル bar がソートされた配列内で表現されています。シンボル foo は表示されません。

```
$ cat mapfile
{
    global:
        bar = DYNSORT;
        foo = NODYNSORT;
};
$ cc -M mapfile -o foo.so.2 -Kpic -G foo.c
$ elfdump -S foo.so.2 | egrep "foo$|bar$"
      [13]  0x000004b0 0x0000001c  FUNC GLOB  D   0 .text      bar
```

.SUNW_dynsymsort セクションと .SUNW_dyntlssort セクションには、.SUNW_ldynsym セクションの存在が必要です。したがって、- z noldynsym オプションを使用すると、すべてのソートセクションが作成されなくなります。

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 12–22　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。

表 12–22 SPARC: ELF シンボルテーブルエントリ: レジスタシンボル (続き)

フィールド	意味
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 12–23 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。
SYMINFO_BT_EXTERN	0xfffc	シンボル定義は外部です。

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_FILTER	0x02	シンボル定義は、標準フィルタとして機能します。
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。

名前	値	意味
SYMINFO_FLG_AUXILIARY	0x40	シンボル定義は、補助フィルタとして機能します。
SYMINFO_FLG_INTERPOSE	0x80	シンボル定義は割り込み処理として機能します。この属性は動的実行可能ファイルにのみ適用できます。
SYMINFO_FLG_CAP	0x100	シンボルは、機能と関連付けられています。
SYMINFO_FLG_DEFERRED	0x200	シンボルを BIND_NOW 再配置に含めません。

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。
- バージョン依存関係は、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ており、タイプ SHT_SUNW_verneedSHT_SUNW_versym 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。243 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。246 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、351 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。407 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、351 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。
VER_FLG_INFO	0x4	SHT_SUNW_versym 参照は情報提供のために存在しており、実行時に検証の必要はありません。

: 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
: ゼロでない場合、この依存関係バージョンに割り当てられたバージョンインデックス。SHT_SUNW_versym 内でこのインデックスを使用して、大域シンボル参照をこのバージョンに割り当てます。

  Oracle Solaris 10 のリリースまでの Solaris のバージョンでは、バージョンシンボルインデックスは依存関係バージョンに割り当てられていませんでした。これらのオブジェクトでは、vna_other の値は 0 です。

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 12–24 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	シンボルは大域適用範囲を持ち、ユーザー定義のバージョン定義 SHT_SUNW_verdef、またはバージョン依存関係 SHT_SUNW_verneed に割り当てられています。

シンボルが、予約された特別なインデックス 0 に割り当てられる場合があります。このインデックスには次のいずれかの理由のために割り当てられます。

- 非大域シンボルは常に VER_NDX_LOCAL に割り当てられます。しかし、これは実際にはほとんどありません。多くの場合、バージョン管理セクションは、大域シンボルだけを含む動的シンボルテーブル .dynsym と合わせてのみ作成されます。
- SHT_SUNW_verdef バージョン定義セクションがないオブジェクト内に定義された大域シンボル。
- SHT_SUNW_verneed バージョン依存関係セクションがないオブジェクト内に定義された未定義大域シンボル。または、バージョン依存関係のセクションがバージョンインデックスを割り当てないオブジェクト内に定義された未定義大域シンボル。
- シンボルテーブルの最初のエントリは常に NULL です。このエントリは常に VER_NDX_LOCAL を受け取りますが、その値には特別な意味はありません。

オブジェクトで定義されたバージョンは、1 から始まるバージョンインデックスが割り当てられ、バージョンごとに 1 増加します。インデックス 1 は最初の大域バージョンに予約されています。オブジェクトに SHT_SUNW_verdef バージョン定義セクションがない場合、そのオブジェクトで定義されたすべての大域シンボルはインデックス 1 を受け取ります。オブジェクトにバージョン定義セクションがない場合、VER_NDX_GLOBAL は前述の最初のバージョンを参照するだけです。

ほかの SHT_SUNW_verneed 依存関係からオブジェクトが要求したバージョンには、最終のバージョン定義インデックスの後に 1 を加えた値から始まるバージョンインデックスが割り当てられます。これらのインデックスもバージョンごとに 1 増加し

ます。インデックス 1 は常に VER_NDX_GLOBAL に予約されているため、依存関係のバージョンが取り得るの最初のインデックスは 2 となります。

Oracle Solaris 10 リリースまでの Solaris のバージョンでは、バージョンインデックスは SHT_SUNW_verneed 依存関係バージョンに割り当てられていませんでした。このようなオブジェクトでは、いずれのシンボル参照のバージョンインデックスも、バージョン管理情報が該当のシンボルで使用できないことを表す 0 になります。

◆ ◆ ◆ 第 13 章

13 プログラムの読み込みと動的リンク

この章は、オブジェクトファイル情報と、実行中プログラム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。次の主なセクションがあります。

- 393 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。
- 400 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。
- 407 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、399 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 13–1 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 13–2 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。400 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 13–1 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_SUNW_EH_FRAME	0x6474e550
PT_LOSUNW	0x6ffffffa
PT_SUNWBSS	0x6ffffffa
PT_SUNWSTACK	0x6ffffffb
PT_SUNWDTRACE	0x6ffffffc
PT_SUNWCAP	0x6ffffffd
PT_HISUNW	0x6fffffff
PT_HIOS	0x6fffffff

表 13–1 ELF セグメント型 (続き)

名前	値
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。407 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、406 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、354 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、406 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
スレッド固有ストレージのテンプレートを指定します。詳細は、439 ページの「スレッド固有ストレージ (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_SUNW_EH_FRAME
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_SUNW_EH_FRAME は PT_SUNW_EH_UNWIND に相当します。

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
機能要件を指定します。詳細については、347 ページの「機能セクション」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。400 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 13–2　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 13–3　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 12–10 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、424 ページの「大域オフセットテーブル (プロセッサ固有)」と 425 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で 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。

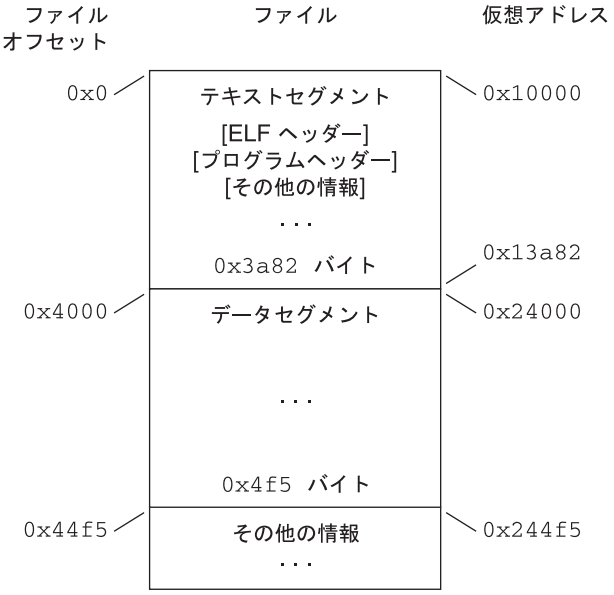
プロセスは実行時に論理ページを参照しないかぎり物理ページを必要としません。プロセスは一般に多くのページを未参照状態のままにします。したがって、物理読み取りを遅延させると、システム性能を向上させることができます。この効率性を実現するには、実行可能ファイルと共有オブジェクトファイルには、ファイルオフセットと仮想アドレスがページサイズを法として同じであるセグメントイメージが存在する必要があります。

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 13–1　SPARC: 実行可能ファイル (64K に整列)



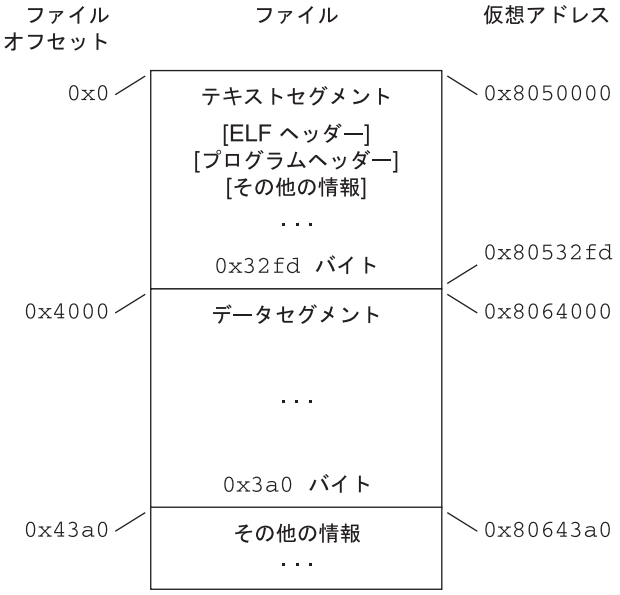
次の表に、前の図に示した読み込み可能セグメント要素の定義を示します。

表 13–4　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
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 13–2　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 13–5　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
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 Oracle 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 13–3　32 ビット SPARC: プロセスイメージセグメント

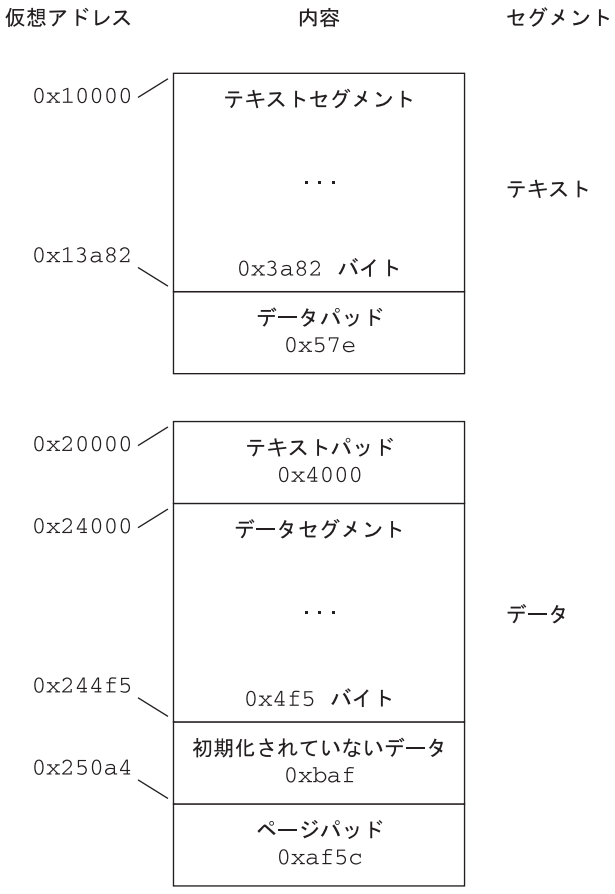
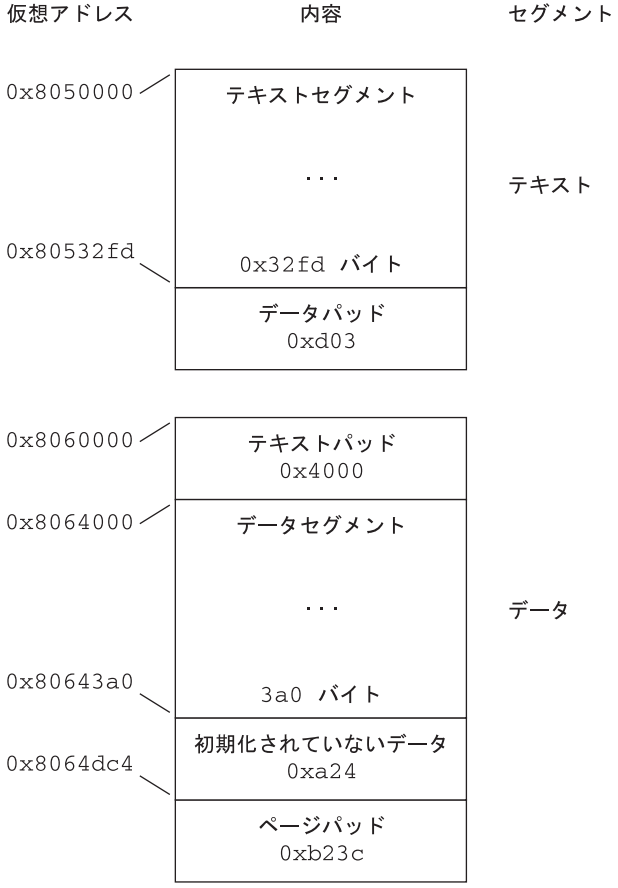


図 13–4 x86: プロセスイメージセグメント



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 13–6　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 13–7　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。

Oracle 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。これらのセクションには、2 つの独立したテーブル (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) が格納されます。これ以降のセクションでは、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。

一般的に、各動的タグ値によって d_un union の解釈が決まります。この規則は、他社製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。互換性のために次のような特別な範囲の値を持つタグは、これらの規則に従いません。他社製ツールは、これらの例外的な範囲に項目ごとに明示的に対応する必要があります。

- 特別な値 DT_ENCODING より小さい値を持つタグ。
- DT_LOOS - DT_SUNW_ENCODING の値を持つタグ。
- DT_HIOS - DT_LOPROC の値を持つタグ。

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、オプションは、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 13–8　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	オプション	オプション
DT_PLTRELSZ	2	d_val	オプション	オプション
DT_PLTGOT	3	d_ptr	オプション	オプション

表 13–8 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_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	オプション
DT_RELASZ	8	d_val	必須	オプション
DT_RELAENT	9	d_val	必須	オプション
DT_STRSZ	10	d_val	必須	必須
DT_SYMENT	11	d_val	必須	必須
DT_INIT	12	d_ptr	オプション	オプション
DT_FINI	13	d_ptr	オプション	オプション
DT_SONAME	14	d_val	無視される	オプション
DT_RPATH	15	d_val	オプション	オプション
DT_SYMBOLIC	16	無視される	無視される	オプション
DT_REL	17	d_ptr	必須	オプション
DT_RELSZ	18	d_val	必須	オプション
DT_RELENT	19	d_val	必須	オプション
DT_PLTREL	20	d_val	オプション	オプション
DT_DEBUG	21	d_ptr	オプション	無視される
DT_TEXTREL	22	無視される	オプション	オプション
DT_JMPREL	23	d_ptr	オプション	オプション
DT_BIND_NOW	24	無視される	オプション	オプション
DT_INIT_ARRAY	25	d_ptr	オプション	オプション
DT_FINI_ARRAY	26	d_ptr	オプション	オプション
DT_INIT_ARRAYSZ	27	d_val	オプション	オプション
DT_FINI_ARRAYSZ	28	d_val	オプション	オプション
DT_RUNPATH	29	d_val	オプション	オプション
DT_FLAGS	30	d_val	オプション	オプション

表 13–8 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_ENCODING	32	指定なし	指定なし	指定なし
DT_PREINIT_ARRAY	32	d_ptr	オプション	無視される
DT_PREINIT_ARRAYSZ	33	d_val	オプション	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	オプション
DT_SUNW_RTLDINF	0x6000000e	d_ptr	オプション	オプション
DT_SUNW_FILTER	0x6000000e	d_ptr	指定なし	オプション
DT_SUNW_CAP	0x60000010	d_ptr	オプション	オプション
DT_SUNW_SYMTAB	0x60000011	d_ptr	オプション	オプション
DT_SUNW_SYMSZ	0x60000012	d_val	オプション	オプション
DT_SUNW_ENCODING	0x60000013	指定なし	指定なし	指定なし
DT_SUNW_SORTENT	0x60000013	d_val	オプション	オプション
DT_SUNW_SYMSORT	0x60000014	d_ptr	オプション	オプション
DT_SUNW_SYMSORTSZ	0x60000015	d_val	オプション	オプション
DT_SUNW_TLSSORT	0x60000016	d_ptr	オプション	オプション
DT_SUNW_TLSSORTSZ	0x60000017	d_val	オプション	オプション
DT_SUNW_CAPINFO	0x60000018	d_ptr	オプション	オプション
DT_SUNW_STRPAD	0x60000019	d_val	オプション	オプション
DT_SUNW_CAPCHAIN	0x6000001a	d_ptr	オプション	オプション
DT_SUNW_LDMACH	0x6000001b	d_val	オプション	オプション
DT_SUNW_CAPCHAINENT	0x6000001d	d_val	オプション	オプション
DT_SUNW_CAPCHAINSZ	0x6000001f	d_val	オプション	オプション
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	オプション	オプション
DT_PLTPADSZ	0x6ffffdf9	d_val	オプション	オプション

表 13–8 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_MOVEENT	0x6ffffdfa	d_val	オプション	オプション
DT_MOVESZ	0x6ffffdfb	d_val	オプション	オプション
DT_POSFLAG_1	0x6ffffdfd	d_val	オプション	オプション
DT_SYMINSZ	0x6ffffdfe	d_val	オプション	オプション
DT_SYMINENT	0x6ffffdff	d_val	オプション	オプション
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	オプション	オプション
DT_DEPAUDIT	0x6ffffefb	d_ptr	オプション	オプション
DT_AUDIT	0x6ffffefc	d_ptr	オプション	オプション
DT_PLTPAD	0x6ffffefd	d_ptr	オプション	オプション
DT_MOVETAB	0x6ffffefe	d_ptr	オプション	オプション
DT_SYMINFO	0x6ffffeff	d_ptr	オプション	オプション
DT_ADDRRNGHI	0x6ffffeff	指定なし	指定なし	指定なし
DT_RELACOUNT	0x6ffffff9	d_val	オプション	オプション
DT_RELCOUNT	0x6ffffffa	d_val	オプション	オプション
DT_FLAGS_1	0x6ffffffb	d_val	オプション	オプション
DT_VERDEF	0x6ffffffc	d_ptr	オプション	オプション
DT_VERDEFNUM	0x6ffffffd	d_val	オプション	オプション
DT_VERNEED	0x6ffffffe	d_ptr	オプション	オプション
DT_VERNEEDNUM	0x6fffffff	d_val	オプション	オプション
DT_LOPROC	0x70000000	指定なし	指定なし	指定なし
DT_SPARC_REGISTER	0x70000001	d_val	オプション	オプション
DT_AUXILIARY	0x7ffffffd	d_val	指定なし	オプション
DT_USED	0x7ffffffe	d_val	オプション	オプション
DT_FILTER	0x7fffffff	d_val	指定なし	オプション
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。96 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)。425 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。425 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424 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。351 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。369 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。371 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。356 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。45 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。45 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。141 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。97 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。199 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。425 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。186 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。425 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 13–11 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、195 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。45 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。45 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。97 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 13–9 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。45 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。149 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。146 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
: 機能セクションのアドレス。347 ページの「機能セクション」を参照してください。

DT_SUNW_SYMTAB
シンボルテーブルのアドレス。局所関数シンボルが含まれ、DT_SYMTAB から提供されるシンボルを拡張します。これらのシンボルは常に、DT_SYMTAB から提供されるシンボルの直前に並んで配置されます。371 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_SUNW_SYMSZ
DT_SUNW_SYMTAB と DT_SYMTAB から提供されるシンボルテーブルを結合したサイズ。

DT_SUNW_ENCODING
DT_SUNW_ENCODING と等しいかそれより大きく、かつ DT_HIOS 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_SUNW_SORTENT
DT_SUNW_SYMSORT および DT_SUNW_TLSSORT シンボルソートエントリのサイズ (単位: バイト)。

DT_SUNW_SYMSORT
シンボルテーブルインデックスの配列のアドレス。DT_SUNW_SYMTAB が参照するシンボルテーブル内の関数シンボルと変数シンボルに、ソートキーに基づいてアクセスできます。380 ページの「シンボルソートセクション」を参照してください。

DT_SUNW_SYMSORTSZ
DT_SUNW_SYMSORT 配列の合計サイズ (単位: バイト)。

DT_SUNW_TLSSORT
シンボルテーブルインデックスの配列のアドレス。DT_SUNW_SYMTAB が参照するシンボルテーブル内のスレッド固有シンボルに、ソートキーに基づいてアクセスできます。380 ページの「シンボルソートセクション」を参照してください。

DT_SUNW_TLSSORTSZ
DT_SUNW_TLSSORT 配列の合計サイズ (単位: バイト)。

DT_SUNW_CAPINFO
シンボルをその機能要件に関連付けることができるシンボルテーブルインデックスの配列のアドレス。347 ページの「機能セクション」を参照してください。

DT_SUNW_STRPAD
動的文字列テーブルの末尾に予約されている未使用領域の合計サイズ (単位: バイト)。オブジェクト内に DT_SUNW_STRPAD が存在しない場合は、予約されている領域はありません。

DT_SUNW_CAPCHAIN
機能ファミリインデックスの配列のアドレス。インデックスの各ファミリは 0 エントリで終了します。

DT_SUNW_LDMACH
このオブジェクトを生成したリンカーのマシンアーキテクチャー。DT_SUNW_LDMACH では、ELF ヘッダーの e_machine フィールドに使用される

EM_ 整数値と同じ値が使用されます。312 ページの「ELF ヘッダー」を参照してください。DT_SUNW_LDMACH は、オブジェクトを構築するリンカーのクラス (32 ビットまたは 64 ビット) とプラットフォームを識別するために使用されます。この情報は、実行時リンカーでは使用されず、ドキュメントのためだけに使用されます。

DT_SUNW_CAPCHAINENT
: DT_SUNW_CAPCHAIN エントリのサイズ (バイト単位)。

DT_SUNW_CAPCHAINSZ
: 合計サイズ (バイト単位)、または DT_SUNW_CAPCHAIN 連鎖。

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。383 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。385 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。387 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。196 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。196 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。149 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。146 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。352 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。99 ページの「デフォルトの検索パスの構成」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。281 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。281 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 13–10 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。382 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 13–9 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ストレージスキームを使用します

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。268 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。199 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。186 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、195 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ストレージスキーム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ストレージは、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 13–10 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。
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。

表 13–10 ELF 動的フラグ DT_FLAGS_1 (続き)

名前	値	意味
DF_1_EDITED	0x200000	オブジェクトが最初に構築されてから変更されています。
DF_1_NORELOC	0x400000	内部使用。
DF_1_SYMINTPOSE	0x800000	個別の割り込みシンボルが存在します。
DF_1_GLOBAUDIT	0x1000000	大域監査を確立します。
DF_1_SINGLETON	0x2000000	シングルトンシンボルが存在します。

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、195 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。126 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。152 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。268 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。第 6 章「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、プライマリ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。103 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。44 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。266 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。86 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。86 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。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」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_EDITED
: このオブジェクトがリンカーによって最初に構築されたあとに編集または変更されたことを示します。このフラグは、オブジェクトが最初に構築されたあとになんらかの変更が加えられたことをデバッガに警告するために使用されます。

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_SYMINTPOSE
: プライマリ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に割り込むべき個々のシンボルが、オブジェクトに含まれることを示します。このフラグは、`mapfile` と `INTERPOSE` キーワードを使ってオブジェクトが構築されるときに記録されます。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」を参照してください。

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。284 ページの「大域監査の記録」を参照してください。

DF_1_SINGLETON
: このオブジェクトに `singleton` シンボルが定義されている、またはオブジェクトがこのシンボルを参照していることを示します。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」を参照してください。

表 13–11 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。108 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。126 ページの「グループの分離」を参照してください。

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、Elf32_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 13–12 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を説明するため、表 13–12 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あとには 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 13–12　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 13–13 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を説明するため、表 13–13 にいくつか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 **13–13** 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 13–13 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 13–13 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 13–14　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 13–15　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あと、.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 13–16 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あと、.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 14 章

スレッド固有ストレージ (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。443 ページの「スレッド固有ストレージのアクセスモデル」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 ストレージ要件は、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 ストレージが使用可能な場合にのみ実行できます。440 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。

スレッド固有ストレージ(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、ストレージ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。451ページの「SPARC: スレッド固有ストレージの再配置のタイプ」、458ページの「32ビットx86: スレッド固有ストレージの再配置のタイプ」、および464ページの「x64: スレッド固有ストレージの再配置のタイプ」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。

静的TLS項目に対する再配置の場合、再配置アドレスは、静的TLSテンプレートの最後からの負のオフセットとしてエンコードされます。このオフセットの計算は、最初に、32ビットオブジェクトの場合はもっとも近い8バイト境界に、64ビットオブジェクトの場合はもっとも近い16バイト境界にテンプレートサイズを丸めます。この丸めによって、静的なTLSテンプレートがどのような利用にも適切に配置されることが保証されます。

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 14–1　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

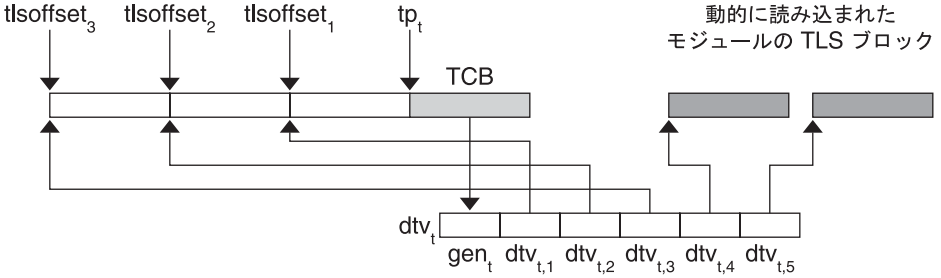
スレッド固有ストレージの実行時の割り当て

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ストレージは、図 14–1 に示すように実行時に配置されます。

図 14–1　スレッド固有ストレージ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 データ項目用のストレージの提供だけです。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ストレージを割り当てます。このストレージが初期化され、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。443 ページの「スレッド固有ストレージ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、次のセクションでは、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、新しいストレージ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。440 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 データ項目用のストレージだけです。

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。

スレッド固有ストレージ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。

スレッド固有ストレージのアクセスモデル

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 $_{\texttt{tls\_get\_addr}}$ 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 $\texttt{dtv}_{0,\texttt{m}}$ 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。
440 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 データ項目用のストレージだけです。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、145 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

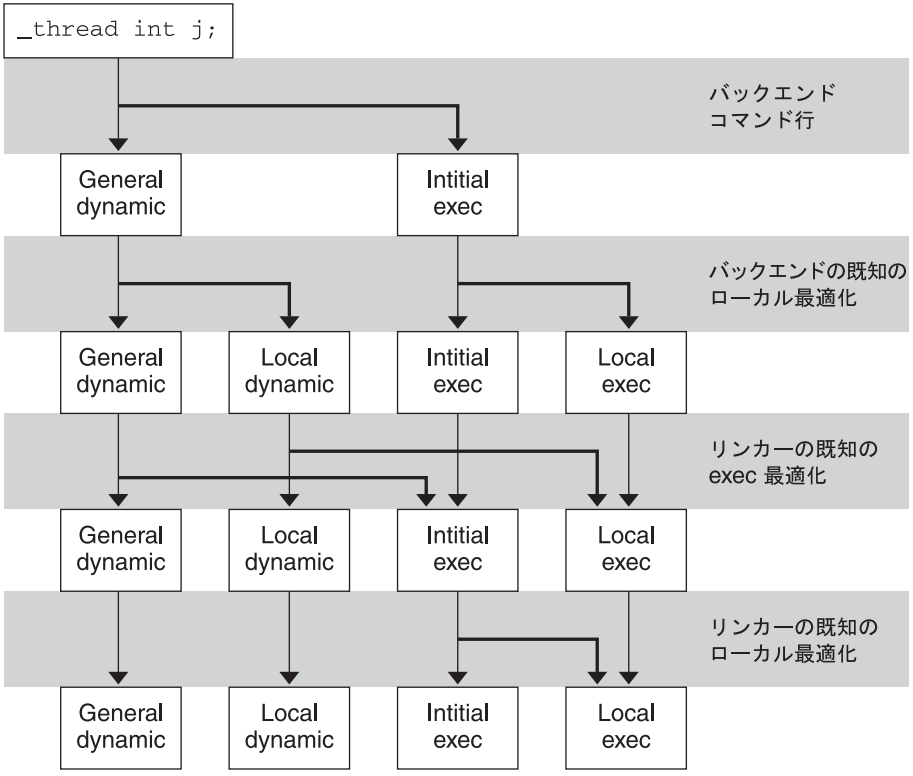
使用して動的実行可能ファイルの外部に存在する変数を参照することはできません。

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 14–2　スレッド固有ストレージ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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている GD モデルを実装します。

表 14–2　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

表 14–2 SPARC: General Dynamic スレッド固有変数のアクセスコード (続き)

# %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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている LD モデルを実装します。

表 14–3 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`
`0x04 add %o0, %lo(@tmndx(x1)), %o0`
`0x08 add %l7, %o0, %o0`
`0x0c call x@TLSPLT`
 	
`# %o0 - contains address of TLS block of current object`
 	
`0x10 sethi %hi(@dtpoff(x1)), %l1`
`0x14 xor %l1, %lo(@dtpoff(x1)), %l1`
`0x18 add %o0, %l1, %l1`
 	
`# %l1 - contains address of local TLS variable x1`
 	
`0x20 sethi %hi(@dtpoff(x2)), %l2`
`0x24 xor %l2, %lo(@dtpoff(x2)), %l2`
`0x28 add %o0, %l2, %l2`
 	
`# %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]` `GOT[n + 1]`	`R_SPARC_TLS_DTPMOD32` `<none>`	x1
	未処理の再配置: **64** ビット	シンボル

表 14–3 SPARC: Local Dynamic スレッド固有変数のアクセスコード (続き)

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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている IE モデルを実装します。

表 14–4 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 14–4 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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている IE モデルを実装します。

表 14–5 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
# %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		

表 14–5 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)

	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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている LE モデルを実装します。

表 14–6 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: スレッド固有ストレージの再配置のタイプ

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 14–7 SPARC: スレッド固有ストレージの再配置のタイプ

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
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)

表 14–7 SPARC: スレッド固有ストレージの再配置のタイプ (続き)

名前	値	フィールド	計算
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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている GD モデルを実装します。

表 14–8　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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている LD モデルを実装します。

表 14–9　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、443 ページの「スレッド固有ストレージのアクセスモデル」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 14–10　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 14–11　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 14–12　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 14–13　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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている LE モデルを実装します。

表 14–14　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 14–15　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 14–16　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: スレッド固有ストレージの再配置のタイプ

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 14–17　32 ビット x86: スレッド固有ストレージの再配置のタイプ

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)
R_386_TLS_GOTIE	16	Word32	@gotntpoff(S)

表 14–17　32 ビット x86: スレッド固有ストレージの再配置のタイプ　(続き)

名前	値	フィールド	計算
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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている GD モデルを実装します。

表 14–18　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 0x6666 0x0a rex64 0x0b call __tls_get_addr@plt # %rax - 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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている LD モデルを実装します。

表 14–19 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 # %r9 - 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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている IE モデルを実装します。

表 14–20 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 14–21 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`
`0x07 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、443 ページの「スレッド固有ストレージのアクセスモデル」で説明されている LE モデルを実装します。

表 14–22 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`
`0x09 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 14–23 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`
`0x09 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 14–24 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: スレッド固有ストレージの再配置のタイプ

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 14–25 x64: スレッド固有ストレージの再配置のタイプ

名前	値	フィールド	計算
R_AMD64_DPTMOD64	16	Word64	@dtpmod(s)

表 14–25 x64: スレッド固有ストレージの再配置のタイプ (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)

パート V

付録

◆ ◆ ◆ 付 録 A

リンカーとライブラリのアップデートおよび新機能

この付録では、Oracle Solaris OS のリリースに追加された更新内容と新機能の概要について説明します。

Oracle Solaris 10 1/13 リリース

- リンカーの `-z discard-unused` オプションにより、使用されない対象物をリンク編集からより柔軟に破棄できるようになりました。189 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。
- リンカーの `-z strip-class` オプションにより、重要でないセクションをオブジェクトから柔軟に取り除くことができるようになりました。`-z strip-class` オプションは古い `-s` オプションに優先し、取り除くセクションをさらにきめ細かく制御します。

Oracle Solaris 10 8/11 リリース

- リンカーはスタブオブジェクトを作成できます。スタブオブジェクトとは、完全に `mapfile` から作成される、コードとデータを含まずに実際のオブジェクトと同じリンクインタフェースを提供する共有オブジェクトです。スタブオブジェクトは、非常に簡単にリンカーで作成できます。また、構築の並列処理を増やしたり、構築の複雑さを軽減したりできます。87 ページの「スタブオブジェクト」を参照してください。
- リンカーは、`-z guidance` オプションを使用して高品質のオブジェクトを作成するためのガイドを提供できます。`ld(1)` を参照してください。
- アーカイブ処理では、サイズが 4G バイトより大きいアーカイブを作成できるようになりました。
- ローカルの監査者が `la_preinit()` と `la_activity()` のイベントを受け取ることができるようになりました。281 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。

- 遅延依存関係により、機能が存在するかどうかを検査するモデルがより堅牢になりました。128 ページの「機能のテスト」および 110 ページの「dlopen() の代替手段の提供」を参照してください。
- 新しい mapfile 構文が利用できるようになりました。第 8 章「mapfile」を参照してください。この構文は、元の System V Release 4 の言語の構文に比べて、可読性と拡張性が向上しています。元の mapfile を処理するためのサポート機能はリンカー内に完全に保持されています。元の mapfile 構文と使用方法については、付録 B 「System V Release 4 (バージョン 1) Mapfile」を参照してください。
- 個々のシンボルは機能要件と関連付けることができます。66 ページの「機能要件の特定」を参照してください。この機能は、動的オブジェクト内の最適化された機能ファミリの作成に対応しています。75 ページの「シンボル機能関数ファミリの作成」および 347 ページの「機能セクション」を参照してください。
- リンカーで作成され、Oracle Solaris 固有の ELF データを含むオブジェクトは、e_ident[EI_OSABI] ELF ヘッダー内で ELFOSABI_SOLARIS のタグが付けられます。これまで、ELFOSABI_NONE がすべてのオブジェクトに使用されてきました。この変更は主に情報を充実させるためであり、実行時リンカーは引き続き ELFOSABI_NONE と ELFOSABI_SOLARIS は同じであると見なします。しかし、elfdump(1) および類似の診断ツールは、この ABI 情報を使用して、特定のオブジェクトに関するより正確な情報を作成できます。
- elfdump(1) は拡張され、e_ident[EI_OSABI] ELF ヘッダーの値、または新しい -O オプションを利用して、特定の ABI に固有の ELF データの型と値を特定したり、この情報を使用してオブジェクトの内容をより正確に表示したりできるようになりました。Linux オペレーティングシステムからのオブジェクト内の ABI 固有情報を表示する機能が大幅に拡張されました。
- プロセスで読み込まれたオブジェクトのセグメント対応付け情報は、dlinfo(3C) のフラグ RTLD_DI_MMAPCNT および RTLD_DI_MMAPS を使用すると取得できます。
- リンカーは多くの GNU リンカーオプションを認識します。ld(1) を参照してください。
- リンカーに SPARC および x86 ターゲットのためのクロスリンクが提供されます。35 ページの「クロスリンク編集」を参照してください。
- リンカーに SHF_MERGE | SHF_STRING 文字列セクションをマージする機能が提供されます。338 ページの「セクションのマージ」を参照してください。
- 実行可能プログラムと共有オブジェクトの作成時に、デフォルトで再配置セクションがマージされるようになりました。196 ページの「再配置セクションの結合」を参照してください。この動作を使用するには、リンカーの -z combreloc オプションが必要でした。このデフォルトの動作を無効にして、再配置を適用する必要のあるセクションで 1 対 1 関係を保持するために、-z nocombreloc が用意されています。
- 新しいユーティリティー elfedit(1) を使って ELF オブジェクトを編集できます。
- 新しいユーティリティー elfwrap(1) を使用すると、任意のデータファイルを ELF 再配置可能オブジェクト内にカプセル化できます。

- 追加のシンボル可視性属性が提供されています。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」および表 12–20 にある、エクスポート済み、シングルトン、および削除属性の記述を参照してください。
- リンカーとその関連 ELF ユーティリティーが、/usr/ccs/bin から /usr/bin に移動されました。34 ページの「リンカーの起動」を参照してください。
- シンボルソートセクションが追加され、メモリーアドレスとシンボリック名の関連付けを簡単にできるようになりました。380 ページの「シンボルソートセクション」を参照してください。
- 動的オブジェクトで使用できるシンボルテーブル情報が、新しい .SUNW_ldynsym セクションの追加によって拡張されています。371 ページの「シンボルテーブルセクション」および表 12–5 を参照してください。
- crle(1) によって管理される構成ファイルの形式が拡張され、ファイルの識別機能が向上しています。改善された形式により、実行時リンカーが互換性のないプラットフォームで生成された構成ファイルを使用しないようになります。
- 関連付けたシンボルのサイズを再配置の計算で使用するために、新しい再配置タイプが追加されました。358 ページの「SPARC: 再配置」を参照してください。
- -z rescan-now、-z recan-start、および -z rescan-end のオプションでは、アーカイブライブラリをリンク編集に容易に指定できるようになりました。41 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。

廃止機能

次の機能が廃止されました。これらの機能は内部機能、またはほとんど使用されない機能です。既存の関連する ELF 定義はすべて無視されますが、elfdump(1) などのツールによって、定義を引き続き表示できます。

DT_FEATURE_1
: この動的セクションタグは実行時の機能要件を特定しました。407 ページの「動的セクション」を参照してください。このタグは機能フラグ DTF_1_PARINIT および DTF_1_CONVEXP を提供しました。DT_FEATURE_1 タグおよび関連フラグは、今後、リンカーによって作成されたり、実行時リンカーによって処理されたりしません。

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。284 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。275 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。
- 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。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」を参照してください。
- リンカーおよび実行時リンカーは、共有オブジェクト内の静的な TLS の作成に対応しています。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。440 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 12–4、表 12–8、および表 12–10 を参照してください。
- 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は、`dlinfo(3C)` のフラグ `RTLD_DI_ARGSINFO` を使用すると取得できます。
- 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。第 6 章「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 12–5、339 ページの「特殊セクション」、368 ページの「x64: 再配置型」、460 ページの「x64: スレッド固有変数へのアクセス」、および464 ページの「x64: スレッド固有ストレージの再配置のタイプ」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。42 ページの「リンカーが検索するディレクトリ」、97 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 117 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。29 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。
- リンカーおよび実行時リンカーは、値が指定されていない環境変数を処理します。32 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。第 10 章「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、インタフェースが存在するかどうかを確認する最適な方法が提供されました。110 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

- リンカーおよび実行時リンカーによって、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。347 ページの「機能セクション」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。285 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。145 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 12–5 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 と -z nodirect、および DIRECT と NODIRECT の mapfile 指令により、直接結合をより細かく制御できるようになりました。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」、および第 6 章「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。127 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) のフラグである 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

- リンカーに文字列テーブルの圧縮機能が追加されたため、.dynstr セクションおよび .strtab セクションが縮小されることがあります。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。64 ページの「文字列テーブルの圧縮」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。
- リンカーは拡張された ELF セクションに対応しています。312 ページの「ELF ヘッダー」、表 12–5、320 ページの「セクション」、表 12–10、および371 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。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」を参照してください。

Solaris 9 リリース

- スレッド固有ストレージ (TLS) のサポートが提供されます。第 14 章「スレッド固有ストレージ (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。41 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。274 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。275 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って解釈される環境変数は、構成ファイル内にこれらの変数を指定することで、複数のプロセス用に確立できます。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。428 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。134 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。480 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。45 ページの「初期設定および終了セクション」、112 ページの「初期設定および終了ルーチン」、表 12–3、表 12–8、表 12–9、346 ページの「グループセクション」、表 12–10、表 12–20、表 13–8、表 13–9、400 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。32 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 296 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。302 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。480 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。112 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 296 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。298 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かを検証できます。86 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。146 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。
- dlsym(3C) および dlinfo(3C) の検索セマンティクスが、新しいハンドル RTLD_SELF を使用して拡張されました。
- 動的オブジェクトの再配置に使用される実行時シンボル検索メカニズムを、各動的オブジェクト内に直接結合情報を確立することによって、大幅に削減することができます。第 6 章「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アなディレクトリは、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 です。117 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、および新しいユーティリティー 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ときの柔軟性が向上しました。44 ページの「実行時リンカーが検索するディレクトリ」と 99 ページの「デフォルトの検索パスの構成」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。225 ページの「SYMBOL_SCOPE/SYMBOL_VERSION 指令」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。99 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。284 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。285 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 13–8 を参照してください。

◆ ◆ ◆ 付 録 B

System V Release 4 (バージョン 1) Mapfile

注 – この付録は、オリジナルの System V Release 4 mapfile 言語 (バージョン 1) について説明しています。この mapfile 構文は継続してサポートされていますが、新しいアプリケーションについては、第 8 章「mapfile」で説明されているバージョン 2 の mapfile 言語をお勧めします。

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

次の基本タイプの指令を `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。

以後のセクションでは、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「オプション」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 B–1 Mapfile セグメント属性

属性	値
segment_type	LOAD | NOTE | NULL | 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。SHT_NOBITS セクションの詳細は、表 12–5 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。488 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、NULL、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、ソフトウェアのリビ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 セグメントまたは NULL 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。空の NULL セグメントは、事後処理ユーティリティーで使用できるプログラムヘッダーエントリを追加するためのものです。これらのセグメントに追加属性を指定すべきではありません。LOAD セグメントと NULL セグメントは複数定義してもかまいません。

?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 B–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 B–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 つ以下の section_type を選択する必要があります。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、320 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 file name 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。487 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 B–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。

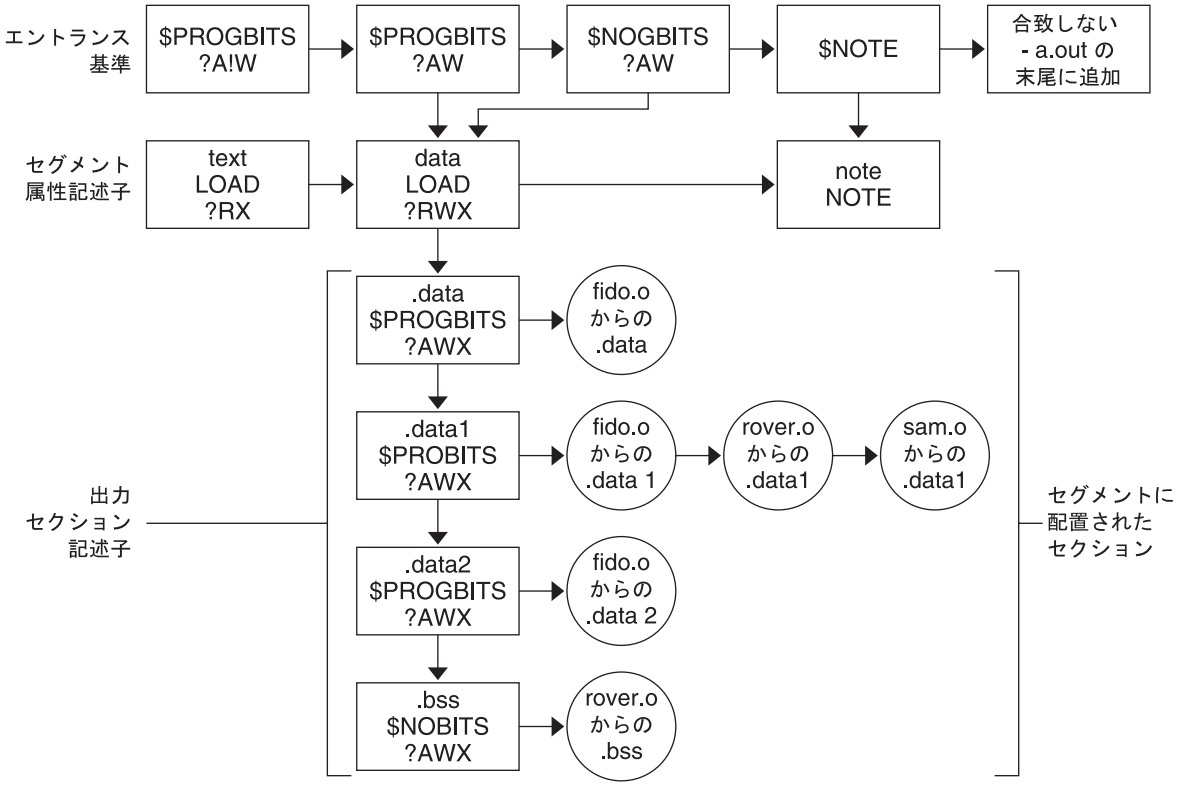
mapfile の各対応付け指令が読み取られる際、同じセグメントに対してすでに指定されたその他の対応付け指令の後 (かつ、そのセグメントのデフォルトの対応付け指令の前) に、指令が追加されます。

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オーバーライドします。

一般的な (ただし多少簡略化した) 対応付け構造を図 B–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 B–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 B–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 $PROGBITS、section_flags が ?AWX のセクションは、図 B–1 の 2 番目のエントランス基準ボックスにあてはまり、data セグメントに置かれます。セクションは 2 番目の出力セクション記述子ボックスにちょうど一致し (.data1、$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。図 B–1 の .data2 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。

索引

E

G

L

M

N

O

P

R

S

T

U

あ

い

う

え

こ

さ

し

た

ち

て

わ