SN SYSTEMS
Sony Computer Entertainment Group

ユーザー ガイド

ProDG リンカー for PlayStation®3

SN Systems Limited
バージョン 310.1
2009年11月24日

目次

1: はじめに

ドキュメント変更履歴

Ver.	日付	変更
310.1	2009年11月	-comment を --comment へ訂正。 --package-file-name スイッチを追加。 ONLY_IF_RO と ONLY_IF_RW スクリプト命令を「警告なく無視」のリストに移動(ドキュメントの誤り)。 Bz73545: 「--sn-no-write-buffer」スイッチを削除。 Bz77567: 「--no-keep-memory」スイッチを削除。 Bz78517: 「—compress-output」 を更新。
300.1	2009年6月	Bz67408: -md=<type> スイッチへの注記を追加。 Bz71141: --temp-dir=<path> スイッチを追加。 Bz71316: --external-make-fself を削除。 Bz71368: --pad-debug-line スイッチの説明を更新。 Bz71822: 「無視されるリンカー スイッチ」 を更新。 Bz71830: --sn-compress-debug=<value> スイッチを削除。 Bz72833: --zgc-sections スイッチを追加。 Bz72922: 「PRX 生成」(デッドストリップ スイッチ) を更新。 Bz74636: 「SNC コンパイラ -Xnotocrestore コントロール変数」 を更新。
280.1	2009年4月	Bz68814:「入力パッケージャー」 を追加。「リンカー スイッチ」を更新。 Bz70338: 細部を修正。 Bz70494/70496: 「リンカー スイッチ」及び「FSELF の作成」を更新。 Bz70601: 「重複削除とデバッギング」 を追加。 Bz70704: 「「L0280 Definition of symbol ... overrides definition from ...」警告」及び「シンボルの上書き」を追加。 Bz70338/71353: 「エラーメッセージ」を削除。その他細かい修正。
270.1	2009年3月	「リンカー出力におけるコマンドライン順の影響」 を追加。 Bz67110/67316/67406: 「リンカー スイッチ」を更新。 Bz67453: 細部を修正。 Bz69709: 「リンカー スイッチ」を更新。
250.3	2008年12月	「リンカー スイッチ」及び「LIB_SEARCH_PATHS」を更新。
250.1	2008年10月	章の大幅な再編成。「リンカー スイッチ」を更新。「セクション シンボル」、「デッドストリッピング」、「リンケージコードの生成」、「TOC 情報」、「リンク後処理のステップ」、「トラブルシューティング」を追加。
220.1	2008年4月	「リンカー スイッチ」と「リンカー スクリプト」を更新。「PRX の自動修正」セクションを新規追加。

リンカーの概要

SN リンカーは、高性能なリンカーとして設計されています。本リンカーにより、SNC あるいは GCC ツールチェーンで作成されたオブジェクト ファイルやアーカイブのリンクが行えます。

以下が、リンカーの主要な機能です。

- デバッガー と互換性のある ELF イメージとデバッグ情報を生成。
- テンプレート、グローバルなオブジェクトの構築・破壊、例外などを含むC++をサポート。
- イメージから未使用および重複する関数を削除。
- イメージから未使用のデータを削除。
- 使用されないデストラクタをイメージから削除。
- メッセージおよびMAPファイルに含まれるシンボル名を再構築。

パフォーマンス

リンク プロセスはほぼすべてが I/O バウンドです。すなわち、時間の大半が入力 ELF ファイルからのセクション読み込みや、最終ELF または SELF 出力ファイルへの書き込みに費やされます (詳細は 49 ページの「リンク後処理のステップ」を参照)。リンカーでは、使用可能なプロセス アドレス領域を最大限に活用するように試みられます (Windows XP では最大 3GB に制限され、32ビット Windows マシンでのプロセス仮想アドレス領域は Boot.ini ファイルで「/3GB」スイッチが使用されない限り 2GB に制限される—詳細は、http://msdn2.microsoft.com/en-us/library/ms791558.aspxを参照)。

リンク時間の向上を試みる場合、ホスト マシンの指定時に考慮すべき要素が 3 つあります。

- システムに十分な RAM を確保。リンク自体は I/O バウンドであるため、仮想メモリ スワップ ファイルの使用を絶対最低限にとどめておくことが重要です。
- スタティック ライブラリの過剰使用を避ける。スタティック ライブラリはシステム コンポーネントや、ライブラリの配布時には欠かせないものの、大きなビルド内での「サブプロジェクト」に対する便利なパッケージング メカニズムとして使用されることもあります。ただし、スタティック ライブラリは、適切なオブジェクト ファイルがリンクに組み込まれていることを確認するため、リンカーによって繰り返しスキャンされる必要があり、このスキャンのせいで実行時間は長くなります。

必要メモリ

リンカー スワッピングを防ぐため、ご使用のシステムには最低2 GBのRAMを確保してください。また、無理のないリンクのため経験則として、メモリはリンク中のオブジェクト、およびライブラリファイルのトータルサイズの最低半分に該当する量が必要と言えます。PS3プロジェクトのリンクには、基本メモリ サイズとして 2 GB が推奨されます。

2: リンカーのコマンドライン書式

コマンドライン書式

以下が、リンカー コマンドライン書式です:

```
ps3ppuld <switches> <files>
```

<switches>	以下のセクションで解説する全てのコマンドライン スイッチ。これらのスイッチの前には、ハイフン(「-」)を付ける必要がある(メモを参照)。
<files>	オブジェクトファイルの場合と、ライブラリの場合がある。リンカーは、リンカー スクリプトで指定されたファイルに加えて、これらのファイルを使用する。

メモ: 1 つの文字から成るスイッチには、1 つのハイフンを前に付ける必要があります。複数の文字から成るスイッチには、1 つまたは 2 つのハイフンを連続して前に付けます (曖昧性を避けるため、2 つのハイフンの使用を推奨)。

多くのスイッチには引数が必要です。

- 1 つの文字から成るスイッチの場合、引数はスイッチに直接付加 (例:-l<library>)、または間にスペースを入れた後付加します (例:-e <entry>)。
- 複数の文字から成るスイッチの場合、引数は等号で区切る (例:--entry=<entry>)、または間にスペースを入れる (例:--Map <mapfile>) 必要があります。

コマンドラインに、コメント (「/*」で始め「*/」で終わる) を挿入することもできます。コメント区切り文字間のスイッチと引数はすべて無視されます。

スイッチ処理の順番

スイッチは、以下のように段階ごとに処理されます。

1. 後続段階 (2 から 5) で処理されないスイッチが、順番に処理されます。
2. 明示的ライブラリ パスが追加されます。
3. リンカー スクリプトが開かれ、解釈されます。
4. リンカー スクリプトからの値を上書きするオプション(--entry=<symbol>)が処理されます。
5. 入力ファイルが開かれます。

   リンカーがオブジェクトまたはアーカイブ ファイルとして認識できないリンカー入力ファイルを指定した場合、そのファイルはリンカー スクリプトとして読み込まれます。このファイルがリンカー スクリプトとして解析不可能な場合は、リンカーによってエラーが報告されます。

入力パッケージャー

SN Systems ではリンカーのサポートや、発生し得る問題の解決を支援するために、入力パッケージング ツールが使用されます。このツールでは、--package スイッチをリンカーに渡すことにより、使われたコマンドライン オプションと共にすべての入力が、1 つの Zip ファイルにパッケージされます。--package-file-name スイッチを使うと、パッケージのファイル名を指定できます。この Zip フ

ァイルを使用すると、SN Systems では過去とまったく同じ方法でリンカーの再実行が行えるため、バグを正確に再現することができます。

以下は、生成されるパッケージ内に含まれる各種ファイルのリストです。

オブジェクト ファイル

リンカーへの入力であるすべてのオブジェクト ファイルが、パッケージ ファイルに収められます。これには、プログラムからのオブジェクト ファイル、使用される任意のライブラリ、および PS3 SDK から取られる必須ファイルが含まれます。なお、入力パッケージャーでは、コードとデバッギング情報がそのまま残されます。

リンク レスポンス ファイル

このファイルには、実行時に行われるリンクの複製に必要とされるコマンドライン情報が含まれます。これは、実際に使用したコマンドラインとまったく同じというわけではありませんが、挙動は複製されます。

ファイル名マッピング

パッケージ内のオブジェクト ファイル名は、可能な限り元のファイル名に類似したものとなります。ただし、名前が重複する場合は、固有の数字が該当ファイル名の 1 つに付加されます。このため、元のパス情報はこの新しいファイル名のせいで無効となります。これを解決するため、全ファイルの情報を確認できるように、元の名前とパス間のマッピング、およびパッケージ中に現れる新しい名前が出力されます。

バージョン ファイル

このファイルには、使用されるリンカーのバージョン情報が含まれます。

スクリプト ファイル リスト

このファイルには、リンカーで使用されるスクリプト ファイルのリストが含まれます。これは、パッケージ ファイルを入力としてリンカーを実行する際に、内部で使用されます。

リンカー スイッチ

多くのスイッチはコンソールプラットフォームには適していないため、GNUリンカーのldがサポートしている全てのスイッチをリンカーはサポートしません。リンカーによってサポートされないスイッチの全リストについては、15 ページの「実装されないリンカー スイッチ」と 16 ページの「無視されるリンカー スイッチ」を参照してください。

コマンドライン スイッチは以下のいずれかになります。

スイッチ	詳細
`-( | --start-group`	グループを開始する。
`-) | --end-group`	グループを終了する。
`--callprof`	プロファイリング コードを最終出力に追加する (SN Tuner と共に使用 ― 詳細は『*Tuner ユーザーガイド*』)を参照)。
`--comment`	コメント セクションを保存する。
`--comment-report=<file>`	リンクの全ファイルに対するコメントを含むレポートを <file> に書き込む。
`--compress-output`	FSELF 出力を圧縮する。--oformat=fselfまたは --oformat=fself_npdrmと共に使用しなければならない。

-d | --dc | --dp	リンカーに、部分リンクであっても共通データを割り当てるよう指令。
--debug file	指定したファイルのデバッグ ログを生成。
--deep-search	すべてのライブラリとオブジェクト ファイルを、--start-group と --end-group 内に含める。
--default-paths	リンカー スクリプト LIB_SEARCH_PATHS エレメントからデフォルト パスを追加する。
--defsym=<symbol>=<value>	シンボル <symbol> を値 <value> で定義する。
--disable-warning=<value>	警告メッセージを無効にする。引数には、先導する「L」を除いたエラー番号を指定。たとえば、「--disable-warning=95」では警告「L0095」が無効化される。 エラー番号のリストをコンマで区切ることにより、複数のメッセージを 1 つのスイッチで無効化できる。たとえば、「--disable-warning=207,24」では警告「L0207」と「L0024」の両方が無効化される。
--discard-all	全てのローカルシンボルを破棄する。
--discard-locals	一時的なローカル シンボルを破棄する。
--dont-strip-section=X	指定されたセクションのデッドストリッピングを差し止める。KEEP リンカー スクリプト命令を使用するのと同等。複数セクションのデッドストリッピングを差し止めるには、複数の --dont-strip-section スイッチを使用する。 例: `--dont-strip-section=.dont_strip1 --dont-strip-section=.dont_strip2` 上記スイッチにより、「.dont_strip1」、「.dont_strip2」と名付けられたセクション内のデータはストリップされません。
--enable-warning	無効化された警告メッセージを有効にする。引数には、先導する「L」を除いたエラー番号を指定。たとえば、「--enable-warning=95」では警告「L0095」が有効化される。 エラー番号のリストをコンマで区切ることにより、複数のメッセージを 1 つのスイッチで有効化できる。たとえば、「--enable-warning=207,24」では警告「L0207」と「L0024」の両方が有効化される。
-e<symbol> | --entry=<symbol>	開始アドレスを設定する。
--exceptions	例外処理ありのライブラリ用のリンカー デフォルト パスを追加する。(LIB_SEARCH_PATHS "exceptions" キーから)
--external-prx-fixup	内部メカニズムではなく、SDK ppu-lv2-prx-fixup ツールを使用する。(デフォルトでは、パフォーマンス向上のため、PRX 修正を内部で実行する。)49ページの「PRX の自動修正」を参照。
--gc-sections	GNU ld のデッドストリッピング スイッチ。2.7.2782.0 より前のバージョンの SN リンカーでは、このスイッチには対応していないと単に警告が発せられる。より後のバージョンでは自動的に --strip-unused-data が選択され、警告が発せられる。 `warning: L0153: --gc-sections is deprecated: using --strip-unused-data for`

	dead stripping
--gnu-mode	GNU 互換機能を有効にする。このオプションは、GCC コンパイラ ドライバがリンカーを呼び出す際に使用されることを意図したもの。
--help	オプション ヘルプを表示する。
--just-symbols=<file> | -R<file>	<file> からのシンボルのみをリンクする。
--keep=<file>	<file>にリストされているすべてのシンボルを保存。これによって、ライブラリで定義されているシンボルであっても、これらのシンボルを含むようにリンカーに指示する。コマンドラインに'--strip-unused' または '--strip-unused-data' が含まれる場合は、<file> にリストされているシンボルは、ストリップされない。ストリッピング機能モジュールによる開始ポイントのストリッピングを防ぐために、このスイッチを使用する。
--keep-eh-data	最終出力ファイルから例外処理データを削除しない。(--gnu-mode が指定されており、--exceptions または --no-exceptions のいずれも指定されていない場合、これがデフォルトの挙動となる。)
--keeptemp	リンク中に作成された任意の一時ファイルを削除しないよう、リンカーに指示する。
-L<path> | --library-path=<path>	<path> をライブラリ検索パスに追加。
-l<name> | --library=<name>	ライブラリファイル「lib<name>.a」をリンクに含める。たとえば-lcと入力すると、リンカーはlibc.aを検索する。
--linkonce-size-error	別モジュールからの linkonce セクションのサイズが異なる場合、エラーを発する。--linkonce-size-warningに優先する。
--linkonce-size-warning	別モジュールからの linkonce セクションのサイズが異なる場合、警告する。
--Map=<mapfile>	<mapfile>内にマップファイルを生成する。
--md=<type>	「ミニダンプ」クラッシュ診断を実行する。<type> は診断のタイプ。 **メモ**: このスイッチは Windows でのみサポートされます。
-mprx	--oformat=prx と同等。GCC との互換用。
-mprx-with-runtime	--oformat=prx --prx-with-runtime と同等。GCC との互換用。
--multi-toc	--multi-toc は、64 KB より多い TOC データの使用を可能にするスイッチ (これがデフォルト の挙動)。 別の TOC 領域を使用するモジュール間でのコール時、リンカーは TOC 領域調整を行う リンケージコードを自動的に生成する。詳細は、33 ページの「TOC を切り替える場合」を参照。
--no-default-script	コマンドラインでリンカスクリプトを指定しなかった場合に、デフォルトのリンカー スクリプトを検索しない。
--no-default-paths	リンカー スクリプト LIB_SEARCH_PATHS エレメントからのデフォルト パスを追加しない。

`--no-demangle`	エラー メッセージやその他出力に含まれるシンボル名をデマングルしない。
`--no-exceptions`	例外処理なしのライブラリ用のリンカー デフォルト パスを追加 (LIB_SEARCH_PATHS "no_exceptions" キーから)。最終出力ファイルから例外処理データの削除を行う。
`--no-keep-eh-data`	--exceptions が指定されている場合を除き、最終出力ファイルから例外処理データを削除する。(--gnu-mode が指定されている場合を除き、--exceptions または --no-exceptions のいずれも指定されていない場合、これがデフォルトの挙動となる。)
`--no-multi-toc`	--no-multi-toc は、複数 TOC 領域をサポートするために使用される リンケージコードをリンカーが作成しないようにする。また、プログラムに 64KB より多い TOC データが含まれる場合、エラーを発する。このスイッチは GNU ld にも存在。 `error: L0154: there is too much TOC data (>64kB) for a single TOC region (consider removing both --no-multi-toc and --no-toc-restore)`
`--no-ppuguid`	PPU GUID の生成を無効にする (デフォルト)。
`--no-prx-fixup`	PRX 修正ステップを実行しない。49ページの「PRX の自動修正」を参照。
`--no-remove-duplicate-inputs`	通常、グループの開始と終了の内側で重複する入力ファイルは無視される。このスイッチでは重複の削除を行わない。
`--no-required-files`	リンカー スクリプト REQUIRED_PATHS エレメントから必要なファイルを追加しない (--required-files も参照)。
`--no-sn-dwarf-string-pool`	オブジェクト ファイル内の不必要な文字列を削除する、重複文字列 チェックを無効にする。
`--no-standard-libraries`	リンカー スクリプト STANDARD_LIBRARIES エレメントによって指定される標準ライブラリを追加しない。
`--notocrestore` | `--no-toc-restore`	SNC PPU C/C++ コンパイラの -Xnotocrestore=2 スイッチと合わせて使用する。 --notocrestore スイッチにより、リンカーでは PRX スタブライブラリ (OS などの PRX 関数へのコールを行うために使用される) の再記述が行われる。この機能は複数 TOC 領域の存在には対応していないため、--notocrestore は --no-multi-toc を意味する。 --no-toc-restore は --notocrestore の別名。
`--no-warn-mismatch`	入力ファイル不適合に関する警告をしない。
`--no-whole-archive`	--whole-archive の効果を無効にする。
`--noinhibit-exec`	エラーが発生した場合でも出力ファイルを作成する。
`-o<file>` | `--output=<file>`	出力ファイルとして <file> を使用。
`--oformat=<format>`	出力ファイルのフォーマットを指定する。リストされた以外のフォーマットとプラットホームの組み合わせは、「unrecognised output format (認識されない出力フォーマット)」エラーの原因となる。

	PS3で使用できるフォーマット: elf、fself、fself_npdrm、prx、fsprx
--package	すべての入力とコマンドライン オプションを 1 つの Zip ファイルに収める。詳細は、7 ページの「入力パッケージャー」を参照。
--package-file-name=<file>	<file> をパッケージ作成時に使われる名前として使用する。
--pad-debug-line=<value>	各 .debug_line データを <value> バイトだけパッドし、再エンコーディング中の拡張を可能にする。デッドストリップが有効な場合 <value> はデフォルトで 1 になり、それ以外の場合はデフォルトで 0 になる。
--ppuguid	PPU GUID の生成を可能にする。これは PPU ELF ファイルを特定する固有の値で、SDK 240 で導入された。また、PRX モジュールのデバッグ実行を簡略化するため、Debugger で使用される場合もある。
--print-toc-info	TOC セクションの要件、各入力オブジェクト ファイルの割当て、TOC セクションの合計数とそのサイズに関する要旨のダンプを生成する。47 ページの「TOC 使用レポート」を参照。
--prx-fixup	PRX 修正ステップを実行する (デフォルト)。49ページの「PRX の自動修正」を参照。
--prx-with-runtime	コンパイラのランタイム ライブラリと PRX 出力をリンクする。(--oformat が「prx」または「fsprx」の場合のみ有効。)
-r | --relocatable | --relocateable	部分リンク実行をリンカーに指令する。
--report-unused	ストリップしたシンボルをファイルstatcov.txtに書き込む。
--required-files	リンカー スクリプト REQUIRED_PATHS エレメントから必要なファイルを追加 (--no-required-files を参照)。
--retain-symbols-file=<file>	<file> にリストされるシンボルのみを保存する。
-S | --strip-debug	すべてのデバッグ情報を出力からストリップする。
--S-lib	ライブラリからのデバッグ情報を出力からストリップする。
-s | --strip-all	すべてのシンボルとデバッグ情報を出力からストリップする。
--s-lib	ライブラリからのシンボルとデバッグ情報を出力からストリップする。
--script=<file> | -T<file>	<file> をリンカー スクリプトとして使用する。
--show-messages	全ての可能なエラーと警告メッセージの一覧を表示します。
--sn-best	あるセクションに対して、リンカー スクリプト内に可能な場所が複数ある場合、そのセクションは最適マッチの場所に配置される。(下記の) --sn-first と比較すること。51 ページの「リンカー スクリプト内でのセクションの場所の不確定性を解決する」を参照。
--sn-first	リンカーは、リンカー スクリプト内で最初マッチした場所にセクションを配置する。(上記の) --sn-best と比較すること。51ページの「リンカー スクリプト内でのセクションの場所の不確定性を解決する」を参照。

--sn-full-map	マップファイル中に追加の情報（たとえばスタティック変数）を入れる。
--sn-no-dtors	リンカーはデストラクタを未使用としてマークする。--strip-unused または --strip-unused-data と合わせて使用された場合は、未使用のデストラクタコードが出力ファイルから削除される。
--sort-common	共通シンボルをサイズで並べ替える。
--strip-duplicates	重複除去プロセスを有効にする。これはデッドストリッピング オプションのいずれか (--strip-unused または --strip-unused-data) と合わせて使用する必要がある。 詳細は、27 ページの「デッドストリッピング」を参照。
--strip-report=<file>	プログラム内のコードや参照を示すファイルを作成する。このレポートは、デッドストリップされている関数やデータを発見するために使用する。 例： --strip-report=stripreport.txt 詳細は、29 ページの「ストリップ レポート」を参照。
--strip-unused	コードのデッドストリッピングを有効にする。リンカーにより、プログラムの完全なコールツリーを構築するため、オブジェクトファイルとアーカイブがスキャンされる。不必要と判断された関数はすべて最終 ELF ファイルから削除される。 詳細は、27 ページの「デッドストリッピング」を参照。
--strip-unused-data	暗黙的に --strip-unused を有効にする。デッド コードのスキャンに加え、リンカーでは、使用されていないデータ オブジェクトを検出し、ELF ファイルからこれらを削除する。 詳細は、27 ページの「デッドストリッピング」を参照。
--sysroot	「sysroot」ディレクトリのプリフィクスを設定する。これは、ライブラリ検索パス (--library-path/-L オプションで設定) が「=」で始まる場合に使用される。「=」はsysroot ディレクトリ プリフィクスで置換される。
--Tbss=<addr>	<addr>をbssセクションのアドレスとして設定する。
--Tdata=<addr>	<addr>をdataセクションのアドレスとして設定する。
--Ttext=<addr>	<addr>をtextセクションのアドレスとして設定する。
-t | --trace	ファイルが開かれると同時に名前を表示する。
--temp-dir=<path>	一時ファイルに使用される一時ディレクトリを指定する。
--Ur	再配置可能な出力ファイルに、グローバル コンストラクタ/デコンストラクタ テーブルを構築する。
-u<symbol> | --undefined=<symbol>	<symbol> への未定義参照から開始する。このスイッチはリンカースクリプト命令 EXTERN と同じ効果があります。
--use-libcs	リンク時に libc.a を libcs.a に置換する。
-V | -v | --version	バージョン情報を表示する。
--verbose	リンク中に多量の情報を出力する。
--Wall	すべての警告を表示する。
--warn-built-before	リンカーによって読み込まれるファイルの最終変更日付が、指定された日付や時刻よりも前の場合に警告する。日

	付は、「yyyy/mm/dd:hh:mm:ss」のように指定される。日付の後ろ部分が省略されている場合、ゼロと見なされる(例:「2007/06/21」は時刻「00:00:00」となる)。
--warn-common	共通オブジェクトで問題が発生した場合は警告する。
--warn-if-debug-found	デバッグ情報を含むことで知られるセクションが検出された場合、警告する。以下のセクション名によって警告が発せられる。 .debug .debug_abbrev .debug_aranges .debug_frame .debug_funcnames .debug_info .debug_line .debug_loc .debug_macinfo .debug_pubnames .debug_pubtypes .debug_ranges .debug_sfnames .debug_srcinfo .debug_str .debug_typenames .debug_varnames .debug_weaknames .line
--warn-once	未定義シンボルについて 1 度だけ警告をする。
--Werror	警告をエラーとして処理する。
--whole-archive	後続するアーカイブからのオブジェクト全てをインクルードします。 --whole-archive オプション後のコマンドライン上で定義されたそれぞれのアーカイブに対し、必要なオブジェクトファイルをアーカイブで検索するのではなく、リンクされたアーカイブ内の全オブジェクトファイルをインクルードします。このオプションは複数回の使用が可能です。この設定を無効にするには --no-whole-archive スイッチを使用します。このスイッチの後に定義されたライブラリは通常どおりリンクされます。

--wrap=<symbol>	<symbol> に対してラッパー関数を使用する。リンカーに引数「-wrap=foo」が渡されると、シンボル foo への参照は __wrap_foo に解決される。さらに、オリジナル foo を解決する、新しいシンボル __real_foo が作成される。たとえば、任意のラッピング論理を実装し、__real_foo シンボルを通じて実際の fopen() コールを呼び出す、関数 __wrap_fopen() を定義することにより、リンカーに引数「--wrap=fopen」を渡し、fopen() へのコールを簡単にインターセプトできるようになる。
	`extern FILE *__real_fopen (const char * restrict filename, const char * restrict mode);`
	`FILE *__wrap_fopen (const char * restrict filename, const char * restrict mode)`
	`{`
	`  printf ('open %s\n', filename);`
	`  return __real_fopen (filename, mode);`
	`}`

--write-fself-digest	SELF ヘッダーに SHA-1要約を作成する。このスイッチを使

	用すると、リンク時間が増加する。(デフォルトでは無効で、--oformat=fself または --oformat=fself_npdrm が使用される場合にのみ有効となる。)
-X	一時的なローカル シンボルを破棄する。
-x	全てのローカルシンボルを破棄する。
--zgc-sections	PRX 出力に対するデッドストリップを有効にする。GCC との互換性を目的とする。リンカーでは --strip-unused-data が選択され、以下の警告が発行される。 warning: L0153: --zgc-sections is deprecated: using --strip-unused-data for dead-stripping
--zgenentry	PRX 出力の作成時に使用する。Cell OS Lv-2 PRX プログラミング ガイドを参照。
--zgenprx	PRX 出力の作成時に使用する。Cell OS Lv-2 PRX プログラミング ガイドを参照。
--zgenstub	リンカーに最初のリンクを実行させ、「--stub-archive」を libgen ツールに渡す。Cell OS Lv-2 PRX プログラミング ガイドを参照。
--zlevel=<openlevel>	ppu-lv2-prx-libgen ツールに直接渡される。

実装されないリンカー スイッチ

以下の GNU ld リンカー コマンドライン スイッチでは、スイッチが使用されると、「option not implemented」エラーが発生します。

スイッチ	詳細
-A<arch> | --architecture=<arch>	アーキテクチャを設定する
--aarchive	HP/UX 互換性に関する共有ライブラリを制御する (-Bstatic と同等)
--adefault | --ashared	HP/UX 互換性に関する共有ライブラリを制御する (-Bdynamic と同等)
-assert	
-b<target> | --format=<target>	次に来る入力ファイルのターゲットを指定する
-Bdynamic | -call_shared | -dy	共有ライブラリをリンクする
-Bshareable | -shared	共有ライブラリを作成する
-Bstatic | -non_shared | -dn	共有ライブラリをリンクしない
-Bsymbolic	グローバルなレファレンスをローカルでバインドする
-c<file> | --mri-script=<file>	MRI形式のリンカー スクリプトを読み込む
--dynamic-linker=<file>	動的リンカーを使用できるよう設定する
-EB	ビッグエンディアンオブジェクトをリンクする

-EL	リトルエンディアン オブジェクトをリンクする
--embedded-relocs	埋め込み再配置可能ファイルを生成する
-F<name> | --filter=<name>	共有のオブジェクトシンボル表をフィルタリングする
-f<file> | --auxiliary=<file>	共有オブジェクト シンボル テーブルを補助的にフィルタする
--force-exe-suffix	.exe拡張子がついたファイルを強制的に生成する
-h<name> | -soname=<name>	共有ライブラリの内部名を設定する
-i	インクリメンタルにリンクする
-q | --emit-relocs	再配置セクションとコンテンツを、完全にリンクされた実行可能形式で残す
--relax	特定の出力先の分岐を広げる
-rpath=<dir>	ランタイムライブラリの検索パスにディレクトリを追加する
-rpath-link=<dir>	指定ディレクトリにある必要な共有ライブラリ の場所を検索する
--split-by-file	出力セクションを各ファイルに分割する
--split-by-reloc=<count>	出力セクションを各<count>再配置可能ファイルに分割する
--task-link	タスクレベルのリンクを実行する
--traditional-format	ネイティブのリンカーと同じフォーマットを使用する
--version-script=<scriptfile>	<scriptfile> からバージョン情報スクリプトを読み込む

無視されるリンカー スイッチ

以下の GNU ld リンカー コマンドライン スイッチでは、これらのスイッチが使用されると、「option ignored」警告が発せられます。

スイッチ	詳細
-g	
-O	
-qmagic	Linux 互換用
-Y<path>	Solaris 互換に対するデフォルトの検索パス

GNU コンパイラ ドライバから渡されるスイッチ「--eh-frame-hdr」と「 -Qy」は、暗黙のうちに無視されます。

リンカー出力におけるコマンドライン順の影響

SN リンカーと GNU リンカーでは、同様のコマンドライン構文が使用されますが、コマンドライン引数の順序とリンカー出力に関しては違いが見られます。

もっとも重要な違いは、アーカイブの処理にあります。SN リンカーでは、コマンドラインにおけるアーカイブの配置によってシンボル解決が影響を受けることはありません。特定のインプットによって解

決可能なシンボルはすべてリンク中に検出されます。結果として、SN リンカーでは --start-group と --end-group アーカイブ分類演算子に対応しているものの、これらを使用する必要がありません。アーカイブ分類演算子は、最終出力のレイアウトに影響を与え、分類されたアーカイブからの情報を互いに近くに配置する役割を果たします。一方、GNU リンカーはデフォルトではコマンドラインで指定されたアーカイブを左から右への1回のパスで処理するため、未定義シンボルを参照するオブジェクトやアーカイブは、該当シンボルを定義するアーカイブの左側に配置する必要があります。

関連する差異として、コマンドラインにオブジェクトとアーカイブを交互に指定しても、SN リンカーの出力レイアウトに影響を与えないことが挙げられます。そうではなく、すべてのオブジェクト ファイルが一緒に処理され、その後アーカイブが、未解決シンボルに対応するため必要に応じて処理されます。従って、GNU リンカーを使用した場合には可能であるとしても、オブジェクトとアーカイブの部分がリンカー出力内で混ざることに依存してはいけません。上級ユーザーは、必要に応じてカスタム リンカー スクリプトを使用することにより、求めるレイアウトを強制的に実現することも可能です。

3: リンカー スクリプト

デフォルトのリンカー スクリプト

コマンドライン オプション処理が --script オプションに出会うことなく完了すると、--no-default-script が使用されない限り、リンカーはリンカー スクリプトのデフォルトの場所を合成し、ここで検出されるファイルの処理を試行します。使用されるデフォルトのパスは以下の通りです。

- --relocatable が指定された場合:「$CELL_SDK/target/ppu/lib/prx32.sn」
- --relocatable が指定されない場合:「$CELL_SDK/target/ppu/lib/elf64_lv2_prx.sn」

リンカー スクリプト命令

リンカーは、ほとんどの「ld format」リンカー スクリプト命令に対応しています。リンカースクリプトのフォーマットに関する詳細情報は、GNU のWeb サイトhttp://www.gnu.org/ で、GNU のリンカー ld のドキュメントを参照してください。

GNU のリンカー ld で使用できて、このリンカーで使用できないスクリプト ファイル命令の詳細は、21 ページの「対応しないスクリプト ファイル命令」を参照してください。

キーワード	詳細
ABSOLUTE(<exp>)	式 <exp> の絶対値 (負でないという意味ではなく、再配置不可能という意味) を戻します。主に、シンボル値が通常セクションに関連する場合に、セクション定義内のシンボルに絶対値を割り当てるのに役立ちます。
ADDR(<section>)	指名されたセクションの絶対アドレス (VMA) を戻します。スクリプトでは、事前にこのセクションの場所を定義しておく必要があります。
AFTER	
ALIGN	
ASSERT(<exp>, <message>)	<exp> がゼロ以外であることを保証します。ゼロの場合、エラーコードと共にリンカーが終了され、<message> が印刷されます。
AT	
BLOCK	
BYTE	
COPY	
CREATE_OBJECT_SYMBOLS	各入力ファイルに対するシンボルを作成するよう、リンカーに指示するコマンドです。各シンボルの名前は、対応する入力ファイルの名前になります。各シンボルのセクションは、CREATE_OBJECT_SYMBOLS コマンドに含まれる出力セクションになります。これは、a.out オブジェクト ファイル フォーマット

	の標準です。通常これは、他のオブジェクト ファイル フォーマットに使用されません。
DEFINED	
DSECT	
END	
ENTRY	
EXCLUDE_FILE	
EXTERN(<symbol> <symbol> ...)	未定義シンボルとして <symbol> を強制的に出力ファイルに含めます。これを行うことにより、標準ライブラリから追加モジュールのリンクを引き起こす場合などがあります。各 EXTERN に対して複数のシンボルをリストでき、EXTERN を複数回使用することもできます。このコマンドは、-u コマンドライン オプションと同じ働きをします。
FILEHDR	
FILL	
FLAGS	
FORCE_COMMON_ALLOCATION	このコマンドは、-d コマンドライン オプションと同じ働きをします。すなわち、再配置可能な出力ファイルが指定された場合でさえも (-r)、共通シンボルへのスペース割当てをリンカーで実行します。
GLOBAL	
GROUP (<file> <file> ...)	GROUP コマンドは、指定されるファイルがすべてアーカイブであることを除き、INPUT と同様で、新しい未定義参照が作成されなくなるまで、繰り返し検索を行います。「--start-group」コマンドライン オプションの詳細を参照してください。また、SN リンカーではコンマがファイル名の一部と見なされるため、コンマ区切りのファイル名には対応していません。
INCLUDE <filename>	この時点で、リンカー スクリプト <filename> がインクルードされます。ファイルは現在のディレクトリと、-L コマンドライン オプションで指定された任意のディレクトリで検索されます。
INFO	
INPUT (<file> <file> ...)	コマンド ラインで指定された場合と同様に、INPUT コマンドでは、指定されたファイルをリンクに含むよう、リンカーに指示します。たとえば、リンクを実行するたびに subr.o をインクルードしたいものの、毎回リンク コマンド ラインで設定するのは面倒な場合、リンカー スクリプトに「INPUT (subr.o)」を入れることができます。実際、すべての入力ファイルをリンカー スクリプトにリストし、そのリンカーを -T オプション 1 つで呼び出せます。リンカーでは、該当ファイルを現在のディレクトリで開こうとします。見つからない場合は、リンカーによってアーカイブ ライブラリ検索パスを通じた検索が行われます。-L コマンドライン オプションの詳細を参照してください。INPUT (-lfile) を使用すると、リンカーでは --library コマンドライン スイッチと同様に、その名前を libfile.a に変換します。INPUT コマンドを間接的なリンカー スクリプトで使用すると、リンカー スクリプト ファイルがインクルードされる時点において、該当ファイルがリンクにインクルードされます。これは、アーカイブ検索に影響することがあります。SN リンカーでは、コンマをファイル名の一部として認識するため、コンマ区切りのファイル名には対応して

	いません。
KEEP	
l	
len	
LENGTH	
LIB_SEARCH_PATHS	22 ページの「LIB_SEARCH_PATHS」を参照してください。
LOADADDR	
LOCAL	
LONG	
MAP	
MAX(<exp1>, <exp2>)	<exp1> と <exp2> の最大を戻します。
MEMORY	
MIN(<exp1>, <exp2>)	<exp1> と <exp2> の最小を戻します。
NEXT	
NOCROSSREFS(<section> <section> ...)	このコマンドは、特定の出力セクション間における任意の参照に関するエラーを、リンカーで発行するために使用されます。特定タイプのプログラム、特にオーバーレイを使用する埋め込み式システムにおいて、1 つのセクションがメモリにロードされる際、別のセクションはロードされません。2 つのセクション間における直接参照はすべてエラーとなります。たとえば、あるセクションで定義される関数を、別のセクションでコールするとエラーとなります。NOCROSSREFS コマンドでは、出力セクション名のリストが使用されます。セクション間の相互参照がリンカーによって検出された場合、エラーが報告され、ゼロ以外の終了ステータスが返されます。NOCROSSREFS コマンドでは、入力セクション名ではなく、出力セクション名が使用されます。
NOLOAD	
NONE	
o	
org	
ORIGIN	
OUTPUT(<filename>)	OUTPUT コマンドでは、出力ファイルを指定します。リンカー スクリプトでのOUTPUT(<filename>) 使用は、コマンドラインで「-o<filename>」を使用するのとまったく同じです。両方が使用される場合、コマンド ライン オプションが優先します。OUTPUT コマンドを使用すると、出力ファイルの通常のデフォルト名「a.out」以外のデフォルト名を定義できます。
OVERLAY	
PHDRS	
PROVIDE	
PT_DYNAMIC	
PT_INTERP	
PT_LOAD	

PT_NOTE	
PT_NULL	
PT_PHDR	
PT_SHLIB	
PT_TLS	
QUAD	
REQUIRED_FILES	23 ページの「REQUIRED_FILES」を参照してください。
SEARCH_DIR(<path>)	SEARCH_DIR コマンドでは、リンカーによるアーカイブ ライブラリの検索場所となるパスのリストに、パスを追加できます。SEARCH_DIR(<path>) は、コマンドラインで「-L<path>」を使用した場合とまったく同じ働きをします。両方が使用される場合、リンカーでは両方のパスが検索されます。コマンドライン オプションを使用して指定されたパスが、最初に検索されます。23 ページの「REQUIRED_FILES」も参照してください。
SECTIONS	
SHORT	
SINGLE_TOC	
SIZEOF	
SIZEOF_HEADERS	
sizeof_headers	
SORT	
SQUAD	
STARTUP	STARTUP コマンドは、ファイル名が最初にリンクされる入力ファイルになる点を除き、最初にコマンドラインで指定されたかのように、INPUT コマンドとまったく同じように機能します。これは、エントリ ポイントが常に最初のファイルの始めであるシステムを使用する場合に便利です。
STANDARD_LIBRARIES	24 ページの「STANDARD_LIBRARIES」を参照してください。
STRING	

対応しないスクリプト ファイル命令

以下のスクリプトファイル命令を使用すると、警告が表示されます。

```
OUTPUT_ARCH
OUTPUT_FORMAT
```

以下のスクリプト ファイル命令は、リンカーで実行されず、使用された場合はエラーが発生します。

```
HLL
INHIBIT_COMMON_ALLOCATION
SYSLIB
TARGET
VERSION
```

以下の命令はリンカーによって受け入れられますが、警告なく無視されます。

```
CONSTRUCTORS
FLOAT
NOFLOAT
```

```
ONLY_IF_RO
ONLY_IF_RW
```

セクション

コンパイラの出力によく出現するセクションの完全なリストは以下の通りです。

セクション	用途
.text	プログラムコード
.data	初期化された変数
.rodata	文字列などの読み出し専用データ
.bss	初期化されていない変数
.sdata	初期化された変数 (小データ)
.sbss	初期化されていない変数 (小データ)

リンカー スクリプトでファイルを参照する

リンカー スクリプト内のオブジェクトファイル名にワイルドカードがない場合、リンクコマンドラインに現れなくてもリンクに必要であるとみなされます。従って、もしリンカーがオブジェクトを見つけられない場合はリンクが失敗します。例えば、

```
mysection :
{
     foo.o(.text)
}
```

この場合、foo.oにワイルドカード('*'または'?')がないのでリンクに付加されます。もしリンカーがfoo.oを見つけられない場合はリンクが失敗します。

コマンドラインに明示的にリストされているfoo.oをリンクだけに足したい場合は、スクリプトに若干手を加え、ワイルドカードを含むようにします。例えば、

```
mysection :
{
     foo.o*(.text)
}
```

LIB_SEARCH_PATHS

2 つのリンカー スイッチ (--default-paths、--no-default-paths) により、デフォルトのライブラリ検索パスが、リンカー スクリプトのLIB_SEARCH_PATHS エレメントから取られたパスによって補足されたものであるかどうかが制御されます。

メモ:--default-paths/--no-default-paths スイッチに関係なく、現在のディレクトリは常にライブラリ検索パスに含まれます。

デフォルトのリンカー スクリプトにおける LIB_SEARCH_PATHS 命令は、以下のようになります。

```
LIB_SEARCH_PATHS
```

```
{
    exceptions :
    {
        "$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.1.1"
        "$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.0.2"

        "$THIS_CELL_SDK/target/ppu/lib"

        "$THIS_CELL_SDK/host-win32/sn/ppu/lib/eh"
        "$THIS_CELL_SDK/host-win32/sn/ppu/lib"
        "$SN_PS3_PATH/ppu/lib/sn"
    }
    no_exceptions :
    {
        "$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.1.1/fno-
exceptions"
        "$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.1.1/noeh"
        "$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.1.1"
        "$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.0.2/fno-
exceptions"
        "$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.0.2/noeh"
        "$THIS_CELL_SDK/host-win32/ppu/lib/gcc/ppu-lv2/4.0.2"
        "$THIS_CELL_SDK/host-win32/ppu/lib/gcc"
        "$THIS_CELL_SDK/host-win32/ppu/ppu-lv2/lib"

        "$THIS_CELL_SDK/target/ppu/lib/fno-exceptions"
        "$THIS_CELL_SDK/target/ppu/lib/noeh"
        "$THIS_CELL_SDK/target/ppu/lib"

        "$THIS_CELL_SDK/host-win32/sn/ppu/lib"
        "$SN_PS3_PATH/ppu/lib/sn"
    }
}
```

REQUIRED_FILES

REQUIRED_FILES エレメントには、ターゲットに正しくリンクするために通常必要な、ファイル名の配列が含まれます。これは、システム ライブラリで必要とされる GCC 起動関連ファイルをユーザーが忘れた場合に警告を発するため、PS3 リンカーで使用されてきました。また、これらのファイルのいずれかが欠けている場合に、ユーザーがクラッシュの診断に時間を費やさないようにデザインされました。この機能により、必要なファイルがコマンドラインに明示的にリストされていない場合、リンカーでは自動的にこれらをリンクに含めるようになります。

デフォルトのリンカー スクリプトにおける REQUIRED_PATHS 命令は、以下のようになります。

```
REQUIRED_FILES
{
    ecrti.o
    crt0.o
    crt1.o
    crtbegin.o
    crtend.o
    ecrtn.o
}
```

STANDARD_LIBRARIES

STANDARD_LIBRARIES エレメントは、GROUP コマンドの機能と同じですが、これは、--no-standard-libraries コマンドライン スイッチによって無効にすることができます。

デフォルトのリンカー スクリプトにおける STANDARD_LIBRARIES 命令は以下のようになります。

```
STANDARD_LIBRARIES (-lc -lgcc -lstdc++ -lsupc++ -lm
-lsyscall -llv2_stub -lsnc)
```

4: セクション シンボル

セクションの開始と終了擬似シンボル

リンカーでは、ELF セクションの始まりと終わりに対する GNU ldスタイルの擬似シンボルがサポートされており、これは未定義の参照が検出された場合にインスタンス化されます。

__start_XXX または __stop_XXX と名付けられたシンボルを使用する場合、リンカーでは、「XXX」と名付けられたセクションのアドレスが合成されます。セクション名は、C 名称として表せなければなりません (すなわち、英数字とアンダーライン)。

C 識別子として表せる名前を持つセクションがリンカーで認識されると、それぞれセクションの始まりと終わりを示す「__start_NAME」と「__stop_NAME」という名前のシンボルが推測的に生成されます。これらのシンボルへの未解決参照がリンカー インプット内で検出された場合、生成されるシンボルはリンカー アウトプット内で個別に定義され、参照はこれらを指すように解決されます。この機能は、GNU リンカーによって提供される拡張機能を模倣したものです。

```
int foo __attribute__ ((section ('bar')));
extern const unsigned char __start_bar [];
extern const unsigned char __stop_bar [];
const unsigned char * start_of_bar (void)
{
    return &__start_bar [0];
}
const unsigned char * end_of_bar (void)
{
    return &__stop_bar [0];
}
```

ドット セクション

さらなる拡張機能として、リンカーでは「.」(ドット) で始まるセクションに対し、同様のプロセスを実行します。この場合シンボル名は、先導する「.」をシーケンス「_Z」に置換することによって作成されます。たとえば、「.text」セクションの始まりは「__start__Ztext」と示されます。続いて未解決参照に対して同様のチェックが行われ、シンボルを定義するべきかどうかが決定されます。

メモ:先導するドット後のセクション名の残りは、有効な C 識別子であることが求められます。このため、複数のドットを含むセクション名は無視されます。

セクション名が「.」で始まるものの、それ以外は有効な C 識別子であるとリンカーで認識された場合、該当セクションの始まりと終わりを示すシンボルが推測的に生成されます。リンカー入力内でこれらシンボルへの未解決参照が検出された場合、生成されたシンボルはリンカー出力で個別に定義され、参照はそれらをポイントするように解決されます。シンボル名は先頭の「.」をシーケンス「_Z」に置換することによって作成され、変更されたセクション名の接頭辞に「__start_」または「__stop_」が付けられます。たとえば、.text セクションの始まりは「__start__Ztext」で表されます。

例 1:C 識別子として適切な文字を含む名前を持つセクションの開始アドレスと終了アドレスを検出するために、セクション シンボルを使用。

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

#define SECTION(x)  \
    __attribute__ ((section (x)))
int bar_var_1 SECTION ("bar");
int bar_var_2 SECTION ("bar");

/* リンカーによって生成されるシンボル */
extern void const * __start_bar;
extern void const * __stop_bar;

int main ()
{
    printf ( "__start_bar=%p, __stop_bar=%p\n",
               &__start_bar,
               &__stop_bar
           );
    printf ( "&bar_var_1=%p, &bar_var_2=%p\n",
               &bar_var_1,
               &bar_var_2
           );
    return EXIT_SUCCESS;
}
```

例 2:ドットで始まる名前のセクションにアクセスするために、セクション シンボルを使用。

```
#include <stddef.h>

/* リンカーによって生成されるシンボル */
extern unsigned char const
    __start__Ztext [];
extern unsigned char const
    __stop__Ztext [];

ptrdiff_t size_of_dot_text (void)
{
    return   &__stop__Ztext  [0]
           - &__start__Ztext [0];
}
```

Pragma comment

SN リンカーでは、入力オブジェクト ファイルに含まれる「.linker_cmd」セクションの処理に対応しています。このセクションは、Microsoftスタイルの #pragma comment ("lib","xxx") の使用に対応して SNC 240.1 以降で出力されます。この機能の説明は、Microsoft の Web サイト (http://msdn.microsoft.com/en-us/library/7f0aews7(VS.71).aspx) に掲載されています。

このプラグマは、リンカーのコマンドラインにファイルを自動追加するために使用できます。

5: デッドストリッピング

使用されないコードやデータのストリッピング

GNU リンカーのデッドストリピングは、各セクションの参照カウントを元に処理を行います。再配置エントリがシンボルを参照している場合、そのシンボルを含むセクションは参照済みとして記録 (マーク) されます。処理の終了時点でマークされていないセクションは必要ないと認識され、出力に書き込まれません。このアプローチのマイナス面は、これがセクション全体においてのみ機能する点であり、これは、特殊な GCC の -ffunction-sections スイッチと -fdata-sections スイッチを使用してコードがコンパイルされている必要があることを意味します。これらスイッチの使用によって生ずる ELF セクションの増加は、リンク時間に悪影響を与える可能性があります。

SN リンカーのデッドストリッピングはまったく異なります。SN リンカーのデッドストリッピングは、リンクするコードやデータの参照を正確に決定するために、再配置エントリ自体をスキャンします。この方法の利点は、リンクするオブジェクトが特殊なコンパイラ スイッチを用いてコンパイルされていない場合でも、セクションの中からコードやデータをストリップできることにあります。

コマンドライン スイッチ

SN リンカーのデッドストリッピングの性質上、その制御は、GNU リンカーで使用されるものとは異なる一連のスイッチで行われます。リンクするオブジェクトは、GCC の -ffunction-sections や -fdata-sections スイッチを用いてコンパイルされている必要はありません。

スイッチ	詳細
`--dont-strip-section=X`	指定されたセクションのデッドストリッピングを行わない。リンカー スクリプト命令 KEEP を使用した場合と同等。複数セクションに対してデッドストリッピングを行わないようにするには、複数の --dont-strip-section スイッチを使用する。 例: `--dont-strip-section=.dont_strip1 --dont-strip-section=.dont_strip2` 上記スイッチにより、「.dont_strip1」および「.dont_strip2」と名付けられたセクション内のデータはストリップされません。
`--gc-sections`	GNU リンカーのデッドストリッピング スイッチ。2.7.2782.0 より前のバージョンの SN リンカーでは、このスイッチに対応していないと単に警告が発せられる。より後のバージョンでは、自動的に --strip-unused-data が選択され、警告が発せられる。 `warning: L0153: --gc-sections is deprecated: using --strip-unused-data for dead-stripping`
`--strip-duplicates`	重複除去を有効にする。これはデッドストリッピング オプションのいずれか (--strip-unused または --strip-unused-data) と合わせて使用しなければならない。
`--strip-report=<file>`	プログラム内のコードと参照を示すファイルが作成される。このレポートには、デッドストリップされたコードやデータの情報等が列挙される。 例: --strip-report=stripreport.txt 詳細は、29 ページの「ストリップ レポート」を参照。

`--strip-unused`	コードのデッドストリッピングを有効にする。リンカーは、オブジェクト ファイルをスキャンし、プログラムの完全なコール ツリーを作成する。不必要と判断された関数は出力ファイルからすべて削除される。
`--strip-unused-data`	暗黙的に --strip-unused を有効にする。デッド コードのスキャンに加え、リンカーは、使用されていないデータ オブジェクトを検出し、出力 ファイルからこれらを削除する。

未定義のシンボル

デッドストリッピングが有効な場合でも、参照されるシンボルはすべて定義しなければなりません。仮に、あるシンボルが最終的に呼び出し側で参照されない場合、つまりデッドストリッピングによって出力ファイルに含まれないとリンク前に判断できる場合でも、そのシンボルの定義はリンク時に必要となります。

例:

```
extern void bar (void);
static void foo (void)
{
       bar ();
}

int main ()
{
}
```

bar() がプログラム内の他の場所で定義されておらず、foo() 以外の関数からコールされない場合、つまりこれがデッドストリップされるコードによってのみコールされる場合でも、ストリッピングの有効/無効に関わらず、リンク エラーが発生します。

```
error: L0039: reference to undefined symbol .bar in file main.o
```

重複除去

重複除去はリンカーの機能で、同一コードや読み込み専用データの重複コピーを除去することにより、最終実行ファイル イメージのサイズをさらに削減することを目的としています。プログラムには、多量の重複コードやデータが含まれていることがあります。コンテンツが読み込み専用である場合、リンカーは重複したコンテンツを削除し、削除されたコンテンツへの参照を、残された単一コピーへの参照へ適切に変更することが出来ます。

重複除去を有効にするには、--strip-unused または --strip-unused-data と合わせて、--strip-duplicates スイッチを使用します。

重複削除とデバッギング

重複削除機能を使用する際には、Debugger でプログラムの一部を表示できなくなる可能性が高くなります。残念ながら、DWARF デバッギング フォーマットの制限により、重複削除の使用から生ずるこの副作用は回避することができません。特に、2 つの関数が重複削除される場合を考えてみると、同一実行コードから成る 2 つ以上のソース コード表示が存在することになります。

一般に、重複削除されていないプログラム部分においては Debuggerが使用できます。しかし、もし重複削除の使用時に必要なソース コードが Debugger で表示できない場合、重複削除を無効にすると、デバッギング機能が再び使えるようになります。

ストリップ レポート

ストリップ レポート には、各種デッドストリッピング オプションの適用効果が詳しく記載されます。デッドストリッピング オプションと合わせて指定された場合は、デッドストリッピングの結果も表示されますが、それ以外の場合はデッドストリッピングの予想結果が表示されます。

このレポートは 4 つのセクションに分けられ、それぞれにデッドストリッピング処理に関する情報が提供されます。

使用されないオブジェクト

最初のセクションには、使用されない、または余分と判断されたオブジェクト (重複コードやデータなど) がリストされます。

- デッドストリッピングが有効な場合、ストリップされたシンボルのバイト数 ([ストリップ] 列)、および適切にアラインメントするために残されたパディング バイト数 ([パッド] 列)、およびそのシンボルの名前が一覧表示されます。
- デッドストリッピングが無効な場合、ストリップ可能なシンボルの仮想アドレス ([アドレス] 列)、およびストリップした場合に削減できるバイト数 ([ストリップ] 列)、およびそのシンボルの名前が一覧表示されます。

表に含まれるオブジェクト ファイルやアーカイブ名に続く角括弧内の値は、シンボル テーブル内のオブジェクトのインデックスを表します。また、デッドストリッピングの要約が表に続いて記載されます。

例:

```
  212      0 .memset (.text) in ...\target\ppu\lib\fno-exceptions\
libc.a(memset.o)[8]
    8      0 memset (.opd) in ...\target\ppu\lib\fno-exceptions\
libc.a (memset.o)[7]
    8      0 main (.opd) in ...\test.o[9]

A total of 13836 bytes can be saved by stripping 180 unused objects.
116 padding bytes will be left.
```

"ストリップできないオブジェクトから参照されるオブジェクト"

2 番目のセクションには、何らかの理由でストリップが認められないオブジェクトがリストされます。

このセクションの最初には凡例が記載され、その下の表で使用されるフラグの意味が説明されます。リストには、オブジェクトがストリップできない理由を説明したフラグと、オブジェクト名、そして出力ファイル内でこれが属するセクションが表示されます。ここでもっとも一般的なフラグは「G」で、これはオブジェクトが、ストリップされない他のグローバル シンボルによって参照されていることを意味します。一般的に、これは参照場所を特定のオブジェクトに帰属されられなかったことを意味します (サイズ0のシンボルに関連する位置が起源の場合など)。

例:

```
Flags  Object name (Section name)

-G--    ._start (.text)
```

"ストリップされていないオブジェクトと、参照するオブジェクト"

3 番目のセクションには、出力ファイルに対して決定される参照グラフが含まれます。参照グラフ内のリンクは多くのプログラムにとって視覚化が難しいため、グラフ内の各エントリは、これが直接参照するオブジェクトと共にリストされます。

- エントリの前には、一連のダッシュ(-)と入力ファイル名 (括弧に囲まれている場合もある。オブジェクトがアーカイブに帰する場合は、その後にアーカイブ名) が記載されます。オブジェクトの名前は、括弧内に表示されるターゲット セクションと共にリストされます。
- オブジェクトがグラフ内のリーフでない場合、「requires...」が表示され、そのオブジェクトを参照するオブジェクトのリストが合わせて表示されます (1 行に 1 つの参照オブジェクト)。
- 参照オブジェクトの後に「[symbol]」が表示される場合、それは、これ以上解決できないシンボルへの参照を表します (このため、デッドストリッピングには適さない)。

例:

```
--------
...\target\ppu\lib\fno-exceptions\crt0.o
._start (.text)  requires...
    _start [symbol]
    ._initialize

--------
...\test.o
.main (.text)  requires...
    <.toc.0>
    .puts
```

"デッドストリッピングに準拠したツールチェーンでビルドされていないオブジェクト モジュール"

4 番目のセクションには、SN リンカーで使用されるデッドストリッピング メカニズムとは互換性のない、オブジェクト ファイルのリストが表示されます (括弧に入っている場合もあり、続いて親アーカイブを表示)。

SDK 200 で使用されているものより前の GNU アセンブラでは、分岐元と分岐先が同一のセクションに含まれる場合に、あらかじめ分岐距離を計算し、その値を分岐命令で使用するという最適化が含まれていました。この最適化により、アセンブラの出力には、同一セクション内の分岐に対する再配置エントリが含まれていませんでした。SN リンカーのデッドストリッピングは再配置エントリを元に処理を行うため、上記のような最適化をされたオブジェクトに対しては、デッドストリッピングが正しく処理を行うことが出来ませんでした。

SDK200 以降の GNU アセンブラの出力には必要なすべての再配置エントリが含まれ、かつ、必要な再配置エントリが存在していることを示すビットが ELF のファイル ヘッダーに設定されるため、安全にデッドストリップが行えます。

デッドストリップが安全に行えるとマークされていないオブジェクト ファイルがリンカーで検出されると、警告が発行されます。ここで警告されたオブジェクトに対してデッドストリップを行うためには、SDK200 以降の GNU アセンブラ、または SN アセンブラを用いて、そのオブジェクトを再ビルドする必要があります。

```
warning: L0134: 11 of 67 files were not dead-stripped because they were not
built with a dead-stripping compatible toolchain (for details, see the strip
report [--strip-report <file>]).
```

関数「ゴースト」

リンカーでは、ストリップされた関数の「ゴースト」が出力ファイルに残されるような状況が存在します。

デッドストリッピングや重複除去の際に、リンカーは入力ファイル内で定義されたシンボルを元に処理を行います。これらのシンボルは、ある領域の名前や、場所、アドレスといった情報を持ちます。ただし、シンボルは、対応する領域に関するすべての情報が含まれているわけではなく、シンボルとして定義された範囲の外側にも、その領域に関する情報は存在します。例えばコードのアラインメントを保証するためにアセンブラが追加するパディングのサイズは、コードのシンボルのサイズには含まれません。デッドストリッピングでは、シンボルの情報を元に処理を行うため、削除されるコードがパディングを含んでいる場合に、元のコードの一部が残る場合があります。これを関数の「ゴースト」と呼びます。

以下は、問題が発生する可能性がある例です。

```
        .section .text

        # アラインメントは 2^3 (すなわち 8 バイト)。
        .align   3

        # 「.foo」シンボルを宣言し、.text セクション内で
        # これにアドレスを渡す。
.foo:
        stdu     %sp, -112(%sp)
        mflr     %r0
        std      %r0, 128(%sp)
        ...
        ld       %r0, 128(%sp)
        mtlr     %r0
        addi     %sp, %sp, 112
        bclr     20, 0

        # .foo シンボルのサイズを提供する。
        .size    . - .foo

        # シンボル bar の 8 バイト アラインメントを保証。
        # アセンブラがパディングを挿入する場合がある。
        .align   3
.bar:
```

このオブジェクトに含まれるシンボルfoo および barには、アラインメント属性 8 が指定されています。fooに 8 の倍数バイトのコードが含まれていない場合は、アセンブラが bar の直前に nop 命令 (パディング) を追加することによってアラインメントを保証します。オブジェクト ファイルに含まれる関数のシンボルサイズには、このパディングのサイズが含まれません。

デッドストリッピングを行う場合、参照されないコードやデータはリンカーによって削除されます。ただし、これを行う際には、セクションのアラインメントを順守し、残りのオブジェクトが引き続き適切にアラインされるようにしなければなりません。また、パディングのような、シンボルに含まれない領域は、その領域が不必要かどうかを判断できないため、デッドストリッパーが処理することはできません。

上記の例では、foo も bar も 8 バイトアラインメントが指定されており、これ以外の関数が、8 バイトよりも大きいアラインメントを指定していない場合は、.text セクションのアラインメントは 8 となります。ここで関数 foo をストリップする場合、foo のサイズにはパティングのサイズは含まれないので、(パディングが必要であった場合は) 最終的な結果として、リンカーでパディングの nop をストリップすることができません。また、8バイト アラインメントを維持しなければならないため、nop で占

有される 4 バイトを単純に残すこともできません。結果として、上記の例では、bclr とパディングとして挿入された nop 命令の両方が、ストリップされた関数 foo の「ゴースト」として残ることになります。

GCC の場合、可能な回避策として -falign-functions=4 スイッチを渡します。これにより、関数が 4 バイト境界でアラインされ、不要な nop 命令が削除されます。パフォーマンスへの影響はほとんどあるいはまったくありません。アラインされていない関数の存在により、関数の初回命令発行時に余分な 1 CPU サイクルが追加される可能性があります。ただし、この種の発行ストールが PPU において障害となることはめったにありません。

6: リンケージコードの生成

「リンケージコード」とは、リンク時に作成されるコードの断片を指します。これらは、TOC などの ABI 仕様をサポートするために PS3 で使用されます。

リンケージコードは呼び出し元の近くに挿入され、オリジナルのコール命令がリンケージコードに確実にアクセスできるようにします。

TOC を切り替える場合

TOC を切り替える際に生成されるリンケージコードではTOC レジスタの値が変更されます。このリンケージコードには、呼び出される側の関数アドレスと、呼び出し側の TOC 領域からの相対距離の両方が含まれます。

メモ: 合計 64KB 未満の TOC データをプログラム内で使用することにより、このリンケージコードの生成を阻止し、結果として生じるパフォーマンスへの影響を避けることができます。これを実行する方法には、リンカーの --no-multi-toc または --no-toc-restore オプションの使用が挙げられます。36 ページの「TOC オーバーヘッドの除去」セクションに記載されているテクニックを使用すると、さらに TOC オーバーヘッドを削減することも可能です。

分岐距離が短い場合

分岐距離が相対分岐命令で許可される最大値を越えない場合、以下のリンケージコードがリンカーによって挿入されます (R_PPC64_REL24 再配置に相当)。

```
std   %rtoc, 28(%sp)
addis %rtoc, %rtoc, toc_difference (hi)
addi  %rtoc, %rtoc, toc_difference (lo)
b     R_PPC64_REL24 (callee)
```

分岐距離が長い場合

分岐距離が相対分岐命令で許可される最大値を越える場合、以下のリンケージコードがリンカーによって挿入されます。

呼び出し側では、リンケージコードへの相対的な分岐を実行し、続いてリンケージコードが32ビットで表現されるアドレスに対して分岐を実行します。

TOC レジスタに加え、このリンケージコードによって %r11 と %ctr レジスタが変更されます。これらは、ABI で volatile として定義されています。

```
std   %rtoc, 28(%sp)
addis %rtoc, %rtoc, toc_difference (hi)
addi  %rtoc, %rtoc, toc_difference (lo)
lis   %r11, R_PPC64_ADDR16_HA (callee)
addi  %r11, %r11, R_PPC64_ADDR16_LO (callee)
mtctr %r11
bctr
```

TOC を切り替えない場合

分岐距離が相対分岐命令で許可される最大値を越える場合、リンカーはリンケージコードを挿入します。このリンケージコードによって %r11 と %ctr レジスタが変更されます。これらは、ABI で volatile として定義されています。

```
lis   %r11, R_PPC64_ADDR16_HA (callee)
addi  %r11, %r11, R_PPC64_ADDR16_LO (callee)
mtctr %r11
bctr
```

レジスタの保存および復元をする場合

リンカーは必要に応じて、Cell OS Lv-2 PPU ABI 仕様書の「レジスタの保存および復元関数」に記載される関数を作成します。ABI で定義される名前を持つ未定義関数への参照 (_savegpr0_32 や _restvr_20 など)が存在する場合、リンカーはこのドキュメントに記載する定義で関数を作成します。

リンカーはこれら関数の複数コピー作成をできる限り回避し、かつ、これらの関数へのリンケージコードを作成しないことを保証します。既存の「レジスタの保存および復元関数」に対してアクセスが不可能である場合、リンカーは新しいコピーを生成します。

例:

```
_savegpr0_30:
       std        %r30, -16(%sp)
_savegpr0_31:
       std        %r31, -8(%sp)
       std        %r0, 16(%sp)
       blr
```

「レジスタの保存および復元関数」の全リストについては、Cell OS Lv-2 PPU ABI 仕様書を参照してください。

7: TOC 情報

バックグラウンド

Cell OS Lv-2 PPU ABI 仕様書には、コンパイラとリンカー両方の挙動に影響するTOC として知られる構造についての説明があります。

- 関数へのコールでは、リンカーがコード修正を行えるように、コール命令自体の後にスペースを設けなければならない。
- ポインタによる関数へのコールでは、中間的な構造体である「.opd」エントリを使用しなければならない。この構造体は、ターゲット コードで使用される TOC 領域のアドレスと、ターゲット コード自体のアドレスから構成される。

以下は、両方の挙動を示した例です。

```
typedef void (*func_ptr) (void);
void foo (func_ptr p)
{
     (*p) ();
}

extern void bar (void);
void qaz (void)
{
     bar ();
}
```

SNC -O3 スイッチでこのサンプルをコンパイルすると、以下のようなコードが生成されます。この大半は ABI によって規定されているため、GCC でも非常に似たコード出力が行われます。明確にするために、関数のプロローグとエピローグは削除されています。

```
.foo:
     ...関数プロローグを削除...
     lwz      %r4, 0(%r3)
     std      %rtoc, 40(%sp)
     mtctr      %r4
     lwz      %rtoc, 4(%r3)
     bctrl
     ld      %rtoc, 40(%sp)
     ...関数エピローグを削除...

.qaz:
     ...関数プロローグを削除...
     bl      .bar
     nop
     ...関数エピローグを削除...
```

この命令の大半は TOC を機能させるために存在しています。GCC でコンパイルされるコードとの互換性を確保するため、ABI で規定されているルールに注意深く従っていますが、SNC ではこの TOC にデータを配置しません。

「TOC 復元なし」モードにより、直接コールと関数ポインタによるコールの両方の効率を上げることができます。

TOC オーバーヘッドの除去

このセクションでは、SNC コンパイラの -Xnotocrestore=2 コントロール変数設定、および SN リンカーの --notocrestore (--no-toc-restore の別名) スイッチの使用による、「TOC 復元なし (no TOC restore)」モードの使用について説明します。これらのスイッチにより、TOC のオーバーヘッドをほぼ完全に削除し、全体的なコード サイズを大幅に削減できます。

- SNC の -Xnotocrestore=2 でコンパイルされたコードと、SNC でコンパイルされた他のコードやGCC でコンパイルされたコードは、リンク時に混合させることができます。ただし、アプリケーション内の TOC サイズが合計 64KB 以下である必要があります。この制限は、--no-toc-restore スイッチが使用される際に、リンカーによって適用されます。

「TOC 復元なし」モードの使用方法

1. SNC の -Xnotocrestore=2コントロール変数を設定してコンパイルする。
2. SN リンカーの --notocrestore スイッチを有効にしてリンクします。

メモ:「TOC 復元なし」モードと互換性のない方法で、PRX コードを構築することも可能です。問題が発生した場合は、39 ページの「制限」を参照してください。

SN リンカー コマンドライン スイッチ

スイッチ	詳細
`--multi-toc`	64KB を越える TOC データの使用を可能にする (これがデフォルトの挙動)。 リンカーは、異なる TOC 領域を使用する呼び出しの際に、TOC 領域の切り替えを行うリンケージコードを自動的に生成します。詳細は、33 ページの「TOC を切り替える場合」を参照してください。
`--no-multi-toc`	リンカーが複数 TOC 領域のサポートに使用されるリンケージコードを出力せず、プログラムに 64KB を越える TOC データが含まれる場合にはエラーとする。このスイッチは GNU リンカー にも存在。 `error: L0154: there is too much TOC data (>64kB) for a single TOC region (consider removing both --no-multi-toc and --no-toc-restore)`
`--notocrestore | --no-toc-restore`	SNC PPU C/C++ コンパイラの -Xnotocrestore=2 スイッチと合わせて使用する。 --notocrestore スイッチにより、リンカーは、PRX 関数 (OS など) へのコールを行う際に使用される PRX スタブ ライブラリの書き換えを行います。この機能は、複数 TOC 領域が存在する場合には使用できないため、--notocrestore 使用時は --no-multi-toc が有効になっていると仮定されます。 --no-toc-restore は --notocrestore の別名です。

SNC PPU C/C++ コンパイラコントロール変数

ここでは便宜のために、SNC PPU C/C++ コンパイラ -Xnotocrestore コントロール変数の説明をします。より厳密な説明については、『*SNC PPU C/C++ コンパイラのユーザーガイド*』を参照してください。

コントロール変数	詳細
-Xnotocrestore=0	コンパイラは、ABI 完全準拠コードを生成します。ポインタによって関数をコールするコードでは、呼び出される側の TOC レジスタの値が、呼び出し側のものと異なる可能性があると仮定されます。 呼び出される側のコードがリンク時に別の TOC 領域に存在する場合に、リンカーがTOCポインタの復元を行えるように、外部関数へのコール後に nop 命令が生成されます。 このオプションでビルドされたコードを適切に実行するために、特別なリンカー スイッチは必要ありません。 この値が notocrestore 制御のデフォルト値です。
-Xnotocrestore=1	コンパイラにより、外部関数へのコール後の nop 命令が省かれるが、ポインタを通じたコールは TOC-safe であることが保証されます。プログラムは、SN リンカーの --notocrestore スイッチをつけてリンクする必要があります。
-Xnotocrestore=2	コンパイラでは、外部関数へのコール後の nop 命令が省かれ、ポインタによるコールは常に同じ TOC 領域を使用すると仮定されます。プログラムは、SN リンカーの --notocrestore スイッチをつけてリンクする必要があります。

SNC コンパイラ -Xnotocrestore コントロール変数

-Xnotocrestore=2 を使用して「バックグラウンド」節と同じコードを SNC でコンパイルすると、以下のようなコードが出力されます。

```
.foo:
     ...関数プロローグを削除...
     lwz      %r3, 0(%r3)
     mtctr      %r3
     bctrl
     ...関数エピローグを削除...
.qaz:
     ...関数プロローグを削除...
     bl       .bar
     ...関数エピローグを削除...
```

こちらの方がはるかに優れています。ここでは、最初のケースから 1 ストアと 2 ロードを削除し、2 番目のケースからは不必要な nop を削除しています。

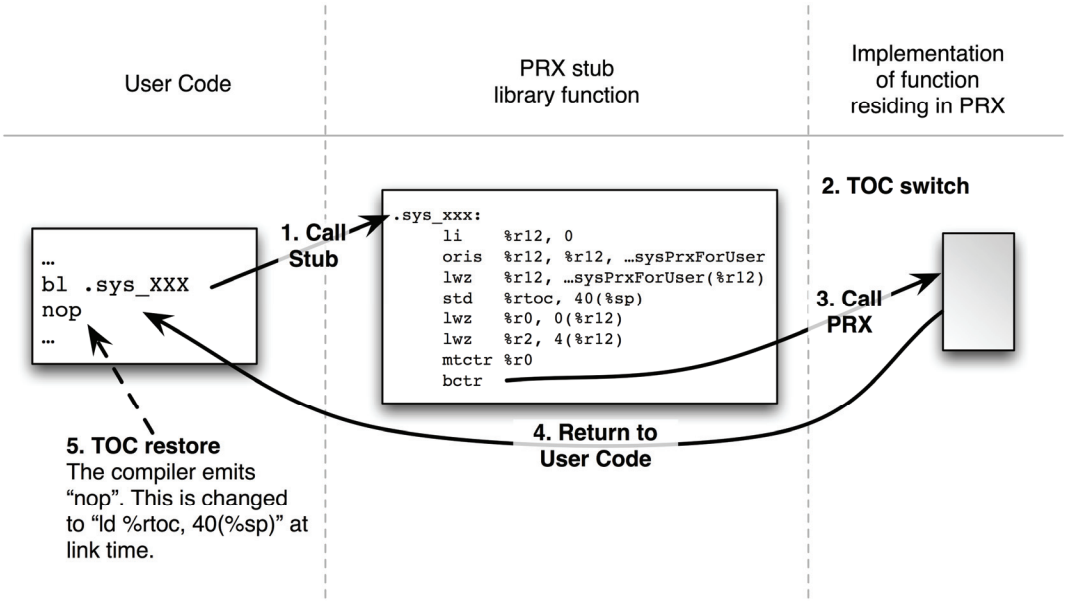
このコードでは、TOC レジスタ (%rtoc/%r2) は関数コールの実行前に変更する必要が一切ない、その後に復元される必要はないと仮定しています。この条件を満たすためには、リンカーによる支援が必要となります。

SN リンカー --notocrestore スイッチ

GCC でコンパイルされた PRX ライブラリは、そのままでは「TOC 復元なし」モデルとともに動作しません。これらの PRX ではメイン プログラムとは別 TOC 領域が必要であり、リンカーではこの挙動を継続してサポートしなければなりません。

以下の図は、PRX スタブ ライブラリ内のコードを、「TOC 復元なし」モードに対応するために別の実装に置き換える、リンカーで使用されるメカニズムを示したものです。

--notocrestore を使用しない PRX コール

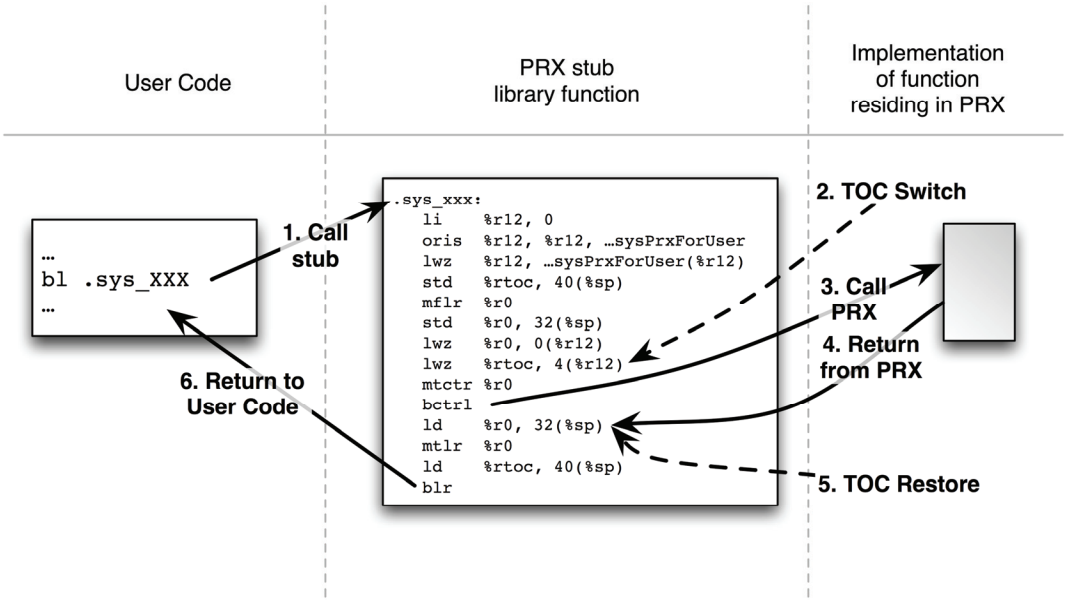


PRX のコールに使用されるデフォルトのコード シーケンスには、ターゲット関数への分岐前に、PRX の TOC データをポイントするように TOC レジスタを変更するコードが含まれます。この関数は、オリジナルのコールした場所に直接戻ります。リンカーは、オリジナル コール命令に続く nop 命令をパッチし、呼び出し側の TOC レジスタ値を復元します。

以下が、実行される処理です。

1. ユーザー コードにより、PRX スタブ コードがコールされます。これにより、PRX コールの実行に必要なセットアップが行われます。
2. PRX スタブ コードでは、TOC レジスタの現在値が保存され、PRX TOC 領域が参照されるように TOC レジスタを変更します。
3. PRX コードが呼び出されます。
4. コールが直接ユーザー コードに戻されます。
5. ユーザー コードによってオリジナルの TOC 領域が復元されます。従来の外部関数コールと同じテクニックを使用し、コンパイラの nop が TOC 復元命令に置換されます。これが「PRX の修正」プロセスです。詳細は 49 ページの「PRX の自動修正」を参照してください。

--notocrestore を使用した PRX コール



「TOC 復元なし」モードが有効な場合、リンカーは、PRX 関数コールの実行に使用されるコード シーケンスが認識すると、これを TOC 復元を直接行うバージョンに置換します (コール元のパッチに依存するのではなく)。

以下は、置換コードによって実行される処理です。

1. ユーザー コードによって PRX スタブ コードがコールされます。これにより、PRX コールの実行に必要なセットアップが行われます。
2. PRX スタブ コードは、TOC レジスタの現在値を保存し、PRX TOC 領域が参照されるように TOC レジスタを変更します。
3. PRX コードが呼び出されます。
4. コールが PRX スタブ コードに戻ります。
5. オリジナルの TOC 値が復元されます。
6. ユーザー コードに戻ります。

このアプローチの利点は、PRX 関数へのコール命令の後に続く、コンパイラによって追加される nop 命令の生成に依存しないことにあります。

この方法で PRX スタブ ライブラリの置換を有効にするには、--notocrestore スイッチを使用してください。

TOC データの合計が 64KB を越える場合、リンカーはエラーを発行します。

```
error: L0154: there is too much TOC data (>64kB) for a single TOC region
(consider removing both --no-multi-toc and --no-toc-restore)
```

GCC では、-mminimal-toc や -mbase-toc スイッチなどを使用し、TOC の領域使用を強制的に少なくすることが可能です。SNC は TOC データを生成することはありません。

制限

「TOC 復元なし」手法は、PRX モジュール内関数へのコールをインターセプトすることに依存しています。直接的なコールの場合、リンカーでは、前述したようにスタブ コードの修正が行えます。

以下の例では、これが C と C++ プログラムの両方で示されており、それぞれで関数ポインタと仮想メソッドが個々に使用されています。

メモ:これらの例では、TOC への参照があることを確実にするために、PRX ライブラリ ソース コードのコンパイルには GCC を使用する必要があります。PRX のコンパイルに SNC が使用される場合、問題は発生しません。

TOC 復元なし C サンプル (prx1)

最初の例は、PRX 常駐ライブラリ (prx1.h、prx1.c) と、PRX を使用するアプリケーション (app1.c) で構成されます。PRX では、関数のアドレスを戻す関数 get_callback() がエクスポートされます。メイン プログラムでは、get_callback() を使用して関数ポインタを取得し、その後にこれをコールします。予想される結果では、PPU stderr チャンネルに「in callback」というテキストが表示されます。

prx1.h:

```
#ifndef PRX1_H
#define PRX1_H

typedef void (*callback_ptr) (void);
callback_ptr get_callback (void);

#endif /* PRX1_H */
```

prx1.c:

```
#include "prx1.h"
#include <sys/prx.h>
#include <sys/tty.h>
SYS_MODULE_INFO (prx1, 0, 1, 0);
SYS_LIB_DECLARE (prx1, SYS_LIB_AUTO_EXPORT |
     SYS_LIB_WEAK_IMPORT);
/* get_callback 関数をエクスポート。 */
SYS_LIB_EXPORT (get_callback,  prx1);

static void write_message (char const * message)
{
    unsigned int write_length;
    char const * end;
    for (end = message; *end != '\0'; ++end)
        ;

    sys_tty_write (SYS_TTYP_PPU_STDERR, message,
                   end - message, &write_length);
}
void callback (void)
{
    write_message ("in callback");
}
callback_ptr get_callback (void)
{
    return &callback;
}
```

app1.c:

```
#include <stdlib.h>
#include <cell/error.h>
#include <sys/prx.h>
#include <sys/paths.h>
#include "prx1.h"
```

```
/* PRX のコール準備 */
static sys_prx_id_t load_start (char const * path)
{
    int module_result;
    sys_prx_id_t id =
              sys_prx_load_module (path, 0, NULL);
    sys_prx_start_module (id, 0, NULL,
              &module_result, 0, NULL);
    return id;
}

/* PRX の後処理 */
static void stop_unload (sys_prx_id_t id)
{
    int module_result;
    sys_prx_stop_module (id, 0, NULL,
              &module_result, 0, NULL);
    sys_prx_unload_module (id, 0, NULL);
}

int main ()
{
    char const * path = SYS_APP_HOME "/prx1.sprx";
    sys_prx_id_t prx_id = load_start (path);

    /* PRX からコールバックを取得し、それをコール */
    callback_ptr cb = get_callback ();
    (*cb) ();

    stop_unload (prx_id);
    return EXIT_SUCCESS;
}
```

prx1 sample Makefile:

```
# Makefile for prx1 example
CC       = ps3ppusnc
CFLAGS   = -g -O0
LD       = ps3ppuld
LDFLAGS  =
RUN      = ps3run
RUNFLAGS = -p -q -r -f . -h .
# 「TOC 復元なし」モードを試みる場合は、
# 以下の 2 行のコメント外す
#CFLAGS += -Xnotocrestore=2
#LDFLAGS += --notocrestore

.PHONY : all
all : prx1.sprx app1.self

.PHONY : clean
clean :
     -rm -f prx1.o prx1.sprx
     -rm -f prx1_stub.a prx1_verlog.txt
     -rm -f app1.o app1.self

.PHONY : run
run : app1.self
     $(RUN) $(RUNFLAGS) $^
```

```
app1.o : app1.c prx1.h

# 問題を再現させるため、TOC が確実に使用
# されるようprx1.o は GCC で
# コンパイルしなければならない。
prx1.o : prx1.c prx1.h
     ppu-lv2-gcc -o $@ -c -g -O0 prx1.c

prx1.sprx prx1_stub.a : prx1.o
     $(LD) --oformat=fsprx -o prx1.sprx \
         $(LDFLAGS) $^
app1.self : app1.o prx1_stub.a
     $(LD) --oformat=fself -o app1.self \
         $(LDFLAGS) $^
```

「TOC 復元なし」モードが使用されるとこの例は失敗します。リンカーは、ポインタによってコールされる場合、callback() で認識されるTOCポインタの値を正しいものにするために行う、コールバック関数のインターセプトができません。

TOC 復元なし C++ サンプル (prx2)

2 番目の例も、PRX 常駐ライブラリ (prx2.h と prx2.cpp 内のソース コード) と、PRX を使用するアプリケーション (app2.cpp) で構成されます。PRX では、クラス foo のインスタンスを戻す関数 get_foo() がエクスポートされます。続いてメイン プログラムでは、クラスの仮想メソッドの 1 つが呼び出されます。予想される結果では、stdout に「in member_function」というテキストが出力されます。

prx2.h:

```
#ifndef PRX2_H
#define PRX2_H

class foo
{
public:
    virtual ~foo ();
    virtual void member_function () const;
};

extern "C" foo * get_foo ();

#endif // PRX2_H
```

prx2.cpp:

```
#include "prx2.h"
#include <cstdio>
#include <sys/prx.h>

SYS_MODULE_INFO (prx2, 0, 1, 1);
SYS_LIB_DECLARE (prx2, SYS_LIB_AUTO_EXPORT |
    SYS_LIB_WEAK_IMPORT);
SYS_LIB_EXPORT (get_foo, prx2);

foo::~foo ()
{
}
void foo::member_function () const
{
    std::puts ("in foo::member_function");
```

```
}

extern "C" foo * get_foo ()
{
    return new foo;
}
```

app2.cpp:

```
#include <cstdlib>
#include <sys/prx.h>
#include <sys/paths.h>
#include "prx2.h"

class prx_loader
{
public:
    prx_loader (char const * path)
    {
        id_ = sys_prx_load_module (path, 0, NULL);
        int module_result;
        sys_prx_start_module (id_, 0, NULL,
            &module_result, 0, NULL);
    }

    ~prx_loader ()
    {
        int module_result;
        sys_prx_stop_module (id_, 0, NULL,
             &module_result, 0, NULL);
        sys_prx_unload_module (id_, 0, NULL);
    }
private:
    sys_prx_id_t id_;
};

extern "C" int sys_libc;
extern "C" int sys_libstdcxx;

int main ()
{
    sys_prx_register_library (&sys_libc);
    sys_prx_register_library (&sys_libstdcxx);
    prx_loader loader (SYS_APP_HOME "/prx2.sprx");

    foo * f = get_foo ();
    f->member_function ();

    return EXIT_SUCCESS;
}
```

prx2 sample makefile:

```
# Makefile for prx2 example
CXX      = ps3ppusnc
CXXFLAGS = -g -O0
LD       = ps3ppuld
LDFLAGS  =
RUN      = ps3run
RUNFLAGS = -p -q -r -f . -h .
```

```
APP_LIBRARIES = libc_libent.o libstdc++_libent.o
PRX_LIBRARIES = -lc_stub -lstdc++_stub

# 「TOC 復元なし」モードを試みる場合は、
# 以下の 2 行のコメントを外す
#CFLAGS += -Xnotocrestore=2
#LDFLAGS += --notocrestore

.PHONY : all
all : prx2.sprx app2.self

.PHONY : clean
clean :
     -rm -f prx2.o prx2.sprx
     -rm -f prx2_stub.a prx2_verlog.txt
     -rm -f app2.o app2.self

.PHONY : run
run : app2.self
     $(RUN) $(RUNFLAGS) $^

prx2.o : prx2.cpp prx2.h
     ppu-lv2-g++ -o $@ -c -g -O0 \
         -fno-exceptions -fno-rtti prx2.cpp

app2.o : app2.cpp prx2.h

# ライブラリ PRX をリンクし、対応する
# スタブ ライブラリを作成。
prx2.sprx prx2_stub.a prx2_verlog.txt : prx2.o
     $(LD) --oformat=fsprx -o prx2.sprx \
         $(LDFLAGS) $^ $(PRX_LIBRARIES)

app2.self : app2.o prx2_stub.a
     $(LD) --oformat=fself -o app2.self \
         $(LDFLAGS) $^ $(APP_LIBRARIES)
```

この例は、前述の C サンプルよりも一般的ではないでしょう。C++ クラス インターフェイスをライブラリから露出することは、「脆弱な基底クラス」問題 (http://en.wikipedia.org/wiki/Fragile_base_class) を参照) による悪影響を受けるため、めったに行われません。とはいえ、この例も同様な問題が発生します。この場合、リンカーでは仮想関数コールのインターセプトができず、TOC 値に必要な調整が行えないため、PRX ライブラリで定義される仮想メソッドへのコールは失敗となる可能性があります。

解決策

この問題を解決するには、2 つのアプローチがあります。最初のアプローチでは、コードを変更する必要はないものの、「TOC 復元なし」モードの利点のいくつかを妥協しなければなりません。2 つめのアプローチではその妥協は削減されるものの、コード変更が必要となります。

SNC コンパイラ -Xnotocrestore=1

最初のアプローチでは TOC 復元モデルを使用します。このモデルではコンパイラにより、間接的な関数コールを実現するためにTOCを意識したコードが生成されます。

これまでの全例において、コンパイラの -Xnotocrestore=2 モードが使用されてきました。このスイッチによってコンパイラでは、外部関数へのコールとポインタによるコールの両方から、TOC 関連の機構が削除されます。

リンカーでは、リンク時にスタブ ライブラリ コードを置換することにより、呼び出される側の TOC レジスタ値がエントリ時に補正されると言う意味では、外部関数のコールが「安全である」と保証できます。ただし、ポインタによるコールについてはこの保証を行えません。

-Xnotocrestore=1 を使用すると、SNC コンパイラでは外部関数へのコール後の nop 命令が省略されますが、ポインタによるコールの実行には、TOCを意識したコードが引き続き使用されます。

この妥協により、上記された両方の例が正しく機能します。残念ながら、不必要な TOC コードが通常より除去されないことにより、「TOC 復元なし」モードの利点は減少します。

#pragma control notocrestore=0

2 つ目のアプローチでは、PRX ライブラリで実装される関数への間接的なコールを特定し、そのコールを実行する小さい関数を記述します。そしてこの新しい関数を、「TOC 復元なし」オプションがコンテキストで無効にされるべきであることを示す、コンパイラ プラグマでマークします。

上記 app1.c の例でのコード:

```
int main ()
{
    ...
    callback_ptr cb = get_callback ();
    (*cb) ();
    ...
}
```

これが以下のようになります。

```
#pragma control %push notocrestore=0
#pragma noinline
void invoke_callback (callback_ptr cb)
{
    (*cb) ();
}
#pragma control %pop notocrestore

int main ()
{
    ...
    callback_ptr cb = get_callback ();
    invoke_callback (cb);
    ...
}
```

「TOC 復元なし」モードを無効にすること、および invoke_callback 関数がインライン化されないようにすることの両方が重要です。最適化制御は関数ごとに機能するため、invoke_callback() がインライン化された場合、「TOC 復元なし」モードの制御値を変更しても効果はありません。

このため、2 つのプラグマを使用する必要があります。1 つは「TOC 復元なし」モードを無効にするもの

```
#pragma control %push notocrestore=0
```

2 つ目は関数のインライン化を防ぐものです。

```
#pragma noinline
```

最後に、「TOC 復元なし」モードの前のステータスを復元します。

```
#pragma control %pop notocrestore
```

C++ 例の変更も同じ方針で行えます。ただし、今回は foo の「プロキシ」クラス (以下の例では foo_proxy と命名) と、スマート ポインタ クラス ntr_ptr (no-toc-restore ポインタ) が導入できるため、foo の仮想メソッドの各コールへの変更数を最小限に抑えることが可能です。

```
// アプリケーションから PRX へのコールに対し、
// プロキシとして作用するクラスを宣言。
class foo_proxy
{
public:
    typedef foo  proxied_type;
    explicit foo_proxy (foo * f) : f_ (f) { }

    void member_function () const;

private:
    foo * f_;
};

// foo::member_function をコールするスタブ関数。
// ABIに準拠した TOC 処理を使用するために、notocrestore 制御を 0 に
// 設定するプラグマと、この関数がインライン化されるのを防ぐための
// プラグマを使用。
#pragma control %push notocrestore=0
#pragma control noinline
void foo_proxy::member_function () const
{
    f_->member_function ();
}
#pragma control %pop notocrestore
```

```
// メンバ関数コールの実行時に、対応するプロキシ クラスが
// 確実に使用されるようにする、「スマート ポインタ」タイプの
// テンプレート クラス。
template <class Proxy>
    class ntr_ptr
    {
    public:
        typedef typename Proxy::proxied_type
            proxied_type;

        explicit ntr_ptr (proxied_type * p)
            : proxy_ (p) { }
        Proxy const * operator-> () const
        {
            return &proxy_;
        }
        Proxy * operator-> ()
        {
            return &proxy_;
        }

    private:
        Proxy proxy_;
    };
```

上記のプロキシ クラスとテンプレート クラスを使用するため、main() の実装を以下のように多少変更します。変更前:

```
int main ()
{
    ...
    foo * f = get_foo ();
```

```
    f->member_function ();
    ...
}
```

変更後:

```
int main ()
{
    ...
    ntr_ptr<foo_proxy> f (get_foo ());
    f->member_function ();
    ...
}
```

TOC 使用レポート

リンカーはTOC の使用状況等をオプションを指定することにより出力することが出来ます。これは、TOC を切り替える際に挿入されるリンケージコードがパフォーマンスやコード サイズにあたえる影響を把握する際に役立ちます (詳細は、33 ページの「TOC を切り替える場合」を参照)。

TOC 使用レポートは 3 つのセクションで構成されます。

- 「TOC モジュール サイズ」セクション。各モジュールと、そこに含まれる TOC データの量が示されます。
- 「TOC 割当て」セクション。リンクされているオブジェクト モジュール内の各セクションに割り当てられている TOC 領域が示されます。また、リンカーでは、別の TOC 領域を使用する関数（つまりリンケージコードが必要となる関数）の名前を発見するため、静的分析が行われます。これらは各セクション内にリストされます。
- 「TOC ステータス」セクション。ここには、それぞれの TOC 領域がリストされ、そのサイズとアドレスが表示されます。

このレポートは、他ツールによる直接的な分析を可能にするため、タブ区切りテキストとして記述されます。

コマンドライン スイッチ

スイッチ	詳細
`--print-toc-info`	標準出力 に TOC 割当てレポートを出力する。

8: PRX ファイルのビルド

PRX 生成

コマンドライン スイッチ

スイッチ	詳細
-mprx	--oformat=prx と同等。GCC との互換用。
-mprx-with-runtime	--oformat=prx --prx-with-runtime と同等。GCC との互換用。
--oformat=prx	PRX出力を作成。
--oformat=fsprx	サイン付き PRX 出力を作成。
--prx-with-runtime	コンパイラ ランタイム ライブラリと PRX 出力をリンク。(--oformat が「prx」または「fsprx」の場合のみ有効」。)
--strip-unused	コードのデッドストリッピングを有効にする。リンカーにより、プログラムの完全なコールツリーを構築するため、オブジェクトファイルとアーカイブがスキャンされる。不必要と判断された関数はすべて最終 PRX ファイルから削除される。 詳細は、27 ページの「デッドストリッピング」を参照。
--strip-unused-data	暗黙的に --strip-unused を有効にする。デッド コードのスキャンに加え、リンカーでは、使用されていないデータ オブジェクトを検出し、PRX ファイルからこれらを削除する。 詳細は、27 ページの「デッドストリッピング」を参照。
--zgc-sections	PRX 出力に対するデッドストリップを有効にする。GCC との互換性を目的とする。リンカーでは --strip-unused-data が選択され、以下の警告が発行される。 `warning: L0153: --zgc-sections is deprecated: using --strip-unused-data for dead-stripping`
--zgenentry	PRX 出力の作成時に使用する。Cell OS Lv-2 PRX プログラミング ガイドを参照。
--zgenprx	PRX 出力の作成時に使用する。Cell OS Lv-2 PRX プログラミング ガイドを参照。
--zgenstub	リンカーに最初のリンクを実行させ、「--stub-archive」を libgen ツールに渡す。Cell OS Lv-2 PRX プログラミング ガイドを参照。

9: リンク後処理のステップ

PS3 でのプログラム実行を可能にするには、2 つのリンク後処理を実行する必要があります。

- ppu-lv2-prx-fixup --stub-fix-only
- make_fselfまたはmake_fself_npdrm

リンクの実行に GCC を使用すると、2 つの処理のうち最初の 1 つがコンパイラ ドライバによって実行され (つまりユーザーにはこの処理が見えない)、make_fself は手動で実行する必要があります。

SN リンカーでは、これらの両ステップをリンキング プロセスの一部として実行できます。ELF ファイルの後処理の必要性、また、make_fself の場合は完全なコピー作成の必要性をなくすことにより、ディスク I/O の量を大幅に削減します。結果として、開発の反復時間が大幅に削減されます (当社テストでは平均 30% の削減)。

PRX の自動修正

有効な PS3 プログラムを作成するために必要な「ppu-lv2-prx-fixup --stub-fix-only」は、リンカーに実装されています。これにより、リンカーの ELF ファイルの後処理を行うプログラムによって実行されねばならない、付加的なディスク アクセスを避けられるため、リンク処理にかかる時間が改善されます。この処理を制御するスイッチがいくつか用意されています。

スイッチ	詳細
`--prx-fixup`	PRX-fixup処理を実行する (デフォルト)。リンカーに PRX-fixup処理の実行を指示する。--gnu-mode が指定されない限りはこれがデフォルト。
`--no-prx-fixup`	PRX-fixup処理を実行しない。これにより、自動「PRX スタブ修正」段階が無効化される。--gnu-mode が指定されている場合、これがデフォルト。
`--external-prx-fixup`	内部メカニズムではなく、外部 「ppu-lv2-prx-fixup --stub-fix-only」ツールが使用される。(パフォーマンス向上のため、デフォルトでは PRX-fixup処理が内部で実行される。)

FSELF の作成

最高のパフォーマンスを得るためには、--oformat=fself スイッチを使用します (コンパイラ ドライバの使用時にはこれがデフォルトです)。

スイッチ	詳細
`--compress-output`	FSELF出力を圧縮する。--oformat-fselfまたは --oformat=fself_npdrmと共に使用しなければならない。
`--oformat=elf`	ELF 出力を作成する (デフォルト)。
`--oformat=fself`	SELF (サイン付きELF) 出力を作成する。

`--oformat=fself_npdrm`	ネットワーク SELF 出力を作成する。
`--write-fself-digest`	SELF ヘッダーに SHA-1 要約を作成する。このスイッチを使用すると、リンク時間が増加する。(デフォルトでは無効で、--oformat=fself または --oformat=fself_npdrm が使用される場合にのみ有効となる。)

10: トラブルシューティング

トラブルシューティング

リンカーのエラーコードおよびその説明は、--show-messages コマンドライン スイッチをお使い下さい。

「L0065 Another location found for ...」警告

この警告は、リンクするセクションに対して、リンカー スクリプト内に可能な場所が複数見つかった場合に生成されます。リンカー スクリプトを編集するか、-sn-first または -sn-best のコマンドラインスイッチを使ってこの問題に対処してください。詳細は、51 ページの「リンカー スクリプト内でのセクションの場所の不確定性を解決する」を参照してください。

「L0280 Definition of symbol ... overrides definition from ...」警告

この警告は、1 シンボルに対する多重定義がリンカーで検出されたものの、その定義の 1 つが他よりも高い優先度を保持していることを示します。リンカーでは、優先度の高いシンボル定義を使用してリンクが継続されます。

ELF 仕様に従うと、多重定義はエラーとなります。ただし、SN リンカーでは 1 つの定義が明らかに優先されるという特定の状況において、多重定義が許可されます。

詳細は、52 ページの「シンボルの上書き」を参照してください。

リンカー スクリプト内でのセクションの場所の不確定性を解決する

不確定性は、リンカー スクリプト内に、リンク済みのいずれかのオブジェクトからのセクションに対して複数の場所が可能である場合に発生します。そのような場合、次のスタイルの警告が生成されます。

```
Command line : warning: L0065:Another location found for .text section from
file C:\so\o1.o
Command line : warning: First location '*(.text)', second 'o1.o(.text)'
Command line : warning: Linker will use the best location for this section
Command line : warning: Use -sn-best to remove warnings. Use -sn-first to
use first location in linker script
```

この場合、ファイル o1.o からのテキストをどこに置くかについて不確定性が生じます。リンカー スイッチの -sn-first と -sn-best を使用すると、リンカー スクリプトをどのように解釈するか制御できます。-sn-first は、リンカー スクリプト内で最初にマッチした場所を使用します。-sn-best は、そのセクションに対する最適な場所にマッチし、警告を生成せずにこれを行います。デフォルト動作では最適な場所にマッチします。この場所がスクリプト内の最初の場所でない場合は、警告が生成されます。

シンボルの上書き

一般的に、多重定義は ELF 仕様で定められるところにより、エラー L0019 の原因となります。ただし、SN リンカーでは、1 つの定義がオブジェクト ファイル内で検出され、別の定義がアーカイブで検出される場合において、多重定義が許可されます。この場合、オブジェクト ファイルからのシンボルに、より高い優先度が与えられ、エラーは発生しません。ただし、警告 L0280 が発行されます。たとえば以下のようになります。

```
Command line : warning: L0280: definition of symbol `.foo' from
"foo_replace.o" overrides definition from "libx.a(foo.o)"
```

この振る舞いを利用すれば、標準ライブラリからの関数を置換するなど、アーカイブからのシンボルを選択的に上書きできます。たとえば、標準ライブラリ版の malloc と free を上書きできます。

メモ:標準ライブラリからの関数を置換するには、注意が必要です。上書きされる関数への全コール (標準ライブラリ内からのものでさえも) がリダイレクトされるため、結果が予測不可能となる場合があります。特に、free を置換せずに malloc を置換すると、メモリ割り当て問題が発生する可能性が高くなります。

多重定義がオブジェクト ファイル内で検出された場合、またはアーカイブ内にのみ検出された場合、エラー L0019 が発生します。

11: インデックス

レ

使

入

分

制

実

対

必

未

無

解

重

関