Solaris 64 ビット 開発ガイド

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

Part No: 819–0389–10
2005 年 1 月

Copyright 2005 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

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

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

U.S. Government Rights Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

本製品に含まれる HG-MinchoL、HG-MinchoL-Sun、HG-PMinchoL-Sun、HG-GothicB、HG-GothicB-Sun、および HG-PGothicB-Sun は、株式会社リコーがリョービイマジクス株式会社からライセンス供与されたタイプフェースマスタをもとに作成されたものです。 HeiseiMin-W3H は、株式会社リコーが財団法人日本規格協会からライセンス供与されたタイプフェースマスタをもとに作成されたものです。フォントとして無断複製することは禁止されています。

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

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

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

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

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

Wnn6 は、オムロン株式会社、オムロンソフトウェア株式会社で共同開発されたソフトウェアです。© Copyright OMRON Co., Ltd. 1995-2000. All Rights Reserved. © Copyright OMRON SOFTWARE Co.,Ltd. 1995-2002 All Rights Reserved.

「ATOK」は、株式会社ジャストシステムの登録商標です。

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

「ATOK Server/ATOK12」に含まれる郵便番号辞書 (7 桁/5 桁) は日本郵政公社が公開したデータを元に制作された物です (一部データの加工を行なっています)。

「ATOK Server/ATOK12」に含まれるフェイスマーク辞書は、株式会社ビレッジセンターの許諾のもと、同社が発行する『インターネット・パソコン通信フェイスマークガイド』に添付のものを使用しています。

Unicode は、Unicode, Inc. の商標です。

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

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

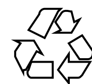
本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含みそれに限定されない、明示的であるか黙示的であるかを問わない、なんらの保証も行われないものとします。

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

原典: *Solaris 64-bit Developer's Guide*

Part No: 816-5138–10

Revision A



050106@10536

目次

はじめに **7**

1 **64** ビットコンピューティング **13**
4G バイト境界を超える 13
「大きなアドレス空間」以外の利点 15

2 **64** ビットの使用方法 **17**
主な特徴 18
大容量仮想アドレス空間 19
大規模ファイル 19
64 ビット演算 19
特定のシステム制約の解消 19
相互運用性の問題 20
カーネルメモリーを参照するプログラム 20
`/proc` の制約 20
64 ビットライブラリ 20
変換作業の見積もり 21

3 **32** ビットと **64** ビットインタフェースの比較 **23**
アプリケーションプログラミングインタフェース 23
アプリケーションバイナリインタフェース 24
32 ビットアプリケーションと 64 ビットアプリケーション間の互換性 24
アプリケーションバイナリ 24
アプリケーションソースコード 25
デバイスドライバ 25

どちらの Solaris オペレーティングシステムが実行されているか 25

4 アプリケーションの変換 **27**
データ型モデル 27
単一ソースコードの実装 30
機能テストマクロ 30
派生型 30
<sys/types.h> ファイル 30
<inttypes.h> ファイル 31
ツールのサポート 34
32 ビットと 64 ビット環境の lint 34
LP64 への変換のためのガイドライン 36
int とポインタが同じサイズであると仮定しない 36
int と long が同じサイズであると仮定しない 37
符号の拡張 37
アドレス演算の代わりにポインタ演算を使う 39
構造体の再構成 39
共用体のチェック 40
定数の型指定 40
暗黙的宣言について 41
sizeof は unsigned long である 41
意図を示すためにキャストを使う 42
書式文字列の変換をチェックする 42
その他の考慮事項 43
サイズが拡大した派生型 43
明示的な 32 ビット 対 64 ビット プロトタイプのために #ifdef を使う 44
アルゴリズムの変更 44
チェックリスト (64 ビットに変換する前に) 44
プログラミング例 45

5 開発環境 **47**
構築環境 47
ヘッダーファイル 47
コンパイラの環境 49
32 ビット ライブラリと 64 ビット ライブラリ 49
オブジェクトファイルのリンク処理 50
LD_LIBRARY_PATH 環境変数 50

$ORIGIN キーワード　50
32 ビット アプリケーションと 64 ビット アプリケーションのパッケージ処理　51
ライブラリとプログラムの配置　51
パッケージ処理のガイドライン　52
アプリケーション命名規則　52
シェルスクリプトラッパー　52
/usr/lib/isaexec バイナリファイル　53
isaexec(3c) インタフェース　54
64 ビットアプリケーションのデバッグ処理　54

6 上級者向けトピック　57
SPARC V9 ABI の特徴　57
スタックバイアス　58
SPARC V9 ABI のアドレス空間の配置　59
SPARC V9 ABI のテキストおよびデータの配置　60
SPARC V9 ABI のコードモデル　61
AMD64 ABI の特徴　62
amd64 アプリケーションのアドレス空間の配置　63
整列の問題　64
プロセス間通信　64
ELF とシステム生成ツール　65
/proc インタフェース　66
sysinfo(2) の拡張　66
libkvm と /dev/ksyms　67
libkstat カーネル統計情報　68
stdio への変更　68
パフォーマンスの問題　69
64 ビットアプリケーションの長所　69
64 ビットアプリケーションの短所　69
システムコールの問題　69
EOVERFLOW の意味　69
ioctl() に関する注意　70

A 派生型の変更 **71**

B よく尋ねられる質問 **(FAQ)** **75**

索引 **79**

はじめに

Solaris™ オペレーティングシステムの機能は、顧客のニーズを満たすために拡大し続けています。Solaris オペレーティングシステムは、32 ビットおよび 64 ビットのアーキテクチャを完全にサポートするように設計されており、大規模ファイルおよび大容量仮想アドレス空間を利用できる 64 ビットアプリケーションを構築し、実行するための環境を提供します。同時に 32 ビットアプリケーションについて、引き続き最大限のソースおよびバイナリの互換性と相互運用性を提供します。実際、Solaris の 64 ビット実装上で実行、および構築されているシステムコマンドの大部分は、32 ビットプログラムです。

注 – このリリースでは、SPARC® および x86 系列のプロセッサアーキテクチャ (UltraSPARC®、SPARC64、AMD64、Pentium、Xeon EM64T) を使用するシステムをサポートします。サポートされるシステムについては、*Solaris 10 Hardware Compatibility List* (`http://www.sun.com/bigadmin/hcl`) を参照してください。 本書では、プラットフォームにより実装が異なる場合は、それを特記します。

本書では、「x86」という用語は AMD64 あるいは Intel Xeon/Pentium 製品系列と互換性のあるプロセッサを使用して製造された 32 ビットおよび 64 ビットシステムを意味します。サポートされるシステムについては、*Solaris 10 Hardware Compatibility List* を参照してください。

---

32 ビットと 64 ビットのアプリケーションの開発環境間の主な相違点は、32 ビットアプリケーションは、`int`、`long`、およびポインタが 32 ビットである ILP32 データ型モデルに基づいているのに対し、64 ビットアプリケーションは、LP64 データ型モデルに基づいているということです。LP64 データ型モデルでは、`long` とポインタは 64 ビットで、ほかの基本データ型は ILP32 と同じです。

大部分のアプリケーションは、32 ビットプログラムのままで使用することができます。次の要件のうち 1 つ以上に該当するアプリケーションのみ、64 ビットプログラムに変換する必要があります。

- 4G バイトを超える仮想アドレス空間を必要とする

- `libkvm` ライブラリ、`/dev/mem` または `/dev/kmem` ファイルを使用してカーネルメモリーを読み込み、解釈する
- `/proc` を利用して 64 ビットプロセスをデバッグする
- 64 ビットバージョンのみで構成されるライブラリを利用する
- 64 ビット演算を効率的に行うために完全な 64 ビットレジスタが必要

このほか特定の相互運用性の問題により、コードの変更が必要になる場合があります。たとえば、アプリケーションが 2G バイトより大きいファイルを使用する場合、64 ビットに変換することがあります。

性能上の理由から、アプリケーションを 64 ビットに変換した方が望ましいことがあります。これには、64 ビット演算を効率良く実行するために 64 ビットレジスタを必要とする場合や、64 ビット命令セットにより提供されるその他の改善された機能を利用する場合などが考えられます。

対象読者

このマニュアルは、C および C++ の開発者を対象読者としています。あるアプリケーションを 32 ビットにするか 64 ビットにするかを判定する方法について説明しています。このマニュアルで示す内容は、次のとおりです。

- 32 ビットと 64 ビットアプリケーション環境の類似性と相違点
- 両環境間で移植可能なコードの書き方
- 64 ビットアプリケーションを開発するための、オペレーティングシステムに含まれているツール

内容の紹介

このマニュアルは次の章で構成されています。

- 第 1 章では、64 ビットコンピューティングがなぜ必要なのか、また 64 ビットアプリケーションの特長について概要を説明します。
- 第 2 章では、構築および実行環境について、32 ビット Solaris と 64 ビット Solaris との違いを説明します。この章は、どのような場合にコードを 64 ビット安全に変換するのが適当であるかを、アプリケーション開発者が判断するための参考となるように書かれています。
- 第 3 章では、32 ビットアプリケーションと 64 ビットアプリケーションの類似性、および 64 ビットインタフェースについて説明します。

- 第 4 章では、既存の 32 ビットコードを 64 ビット安全なコードへ変換する方法と、その変換を簡単に行うためのツールについて説明します。この章では主に、移植性のあるコードの書き方について説明しています。この章の内容は、既存の 32 ビットアプリケーションを 64 ビットに変換、または 32 ビットと 64 ビットの両環境で実行可能なアプリケーションを作成する際に適用できます。
- 第 5 章では、ヘッダー、コンパイラ、ライブラリについて、およびパッケージ方法とデバッグツールなどの構築環境について説明します。
- 第 6 章では、64 ビットシステムプログラミングと ABI の概要、およびパフォーマンスの問題について説明します。
- 付録 A では、64 ビットアプリケーション開発環境で変更された派生型について説明します。
- 付録 B では、64 ビット実装とアプリケーション開発環境に関して、よく尋ねられる質問とその回答をまとめてあります。

関連マニュアル

参考として、次のマニュアルをお薦めします。

- 『*American National Standard for Information Systems Programming Language-C, ANSI X3.159-1989*』
- 『*SPARC Architecture Manual, Version 9*』SPARC International
- 『*SPARC Compliance Definition, Version 2.4*』SPARC International
- 『*Large Files in Solaris: A White Paper*』(Part No: A White Paper』(Part No: 96115-001)
- *Solaris 10 Reference Manual Collection*
- 『*Writing Device Drivers*』 (Part No: 816-4854)
- 『*Sun Studio 10: C* ユーザーズガイド』(Part No: 819-0494-10)

Sun のオンラインマニュアル

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

表記上の規則

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

表 P–1 表記上の規則

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

コード例は次のように表示されます。

- C シェル

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

- C シェルのスーパーユーザー

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

- Bourne シェルおよび Korn シェル

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

- Bourne シェルおよび Korn シェルのスーパーユーザー

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

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

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

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

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

第 1 章

64 ビットコンピューティング

アプリケーションが高機能かつ複雑になり、またデータセットのサイズが大きくなるにつれて、既存のアプリケーションが必要とするアドレス空間のサイズが大きくなっています。今日、32 ビットシステムの 4G バイトアドレス空間の限界を超えるアプリケーションもあります。その例として、次のようなものがあります。

- さまざまなデータベースアプリケーション、特にデータの取り出しを行うデータベースアプリケーション
- Web キャッシュ、Web 検索エンジン
- CAD/CAE のシミュレーションおよびモデリングツールの構成要素
- 科学技術計算

このような大規模アプリケーションを効率的に実行したいという要求によって、64 ビットコンピューティングが開発されてきました。

4G バイト境界を超える

図 1–1 に、大容量の物理メモリーを持つコンピュータ上で実行されるアプリケーションの、典型的なパフォーマンスと問題サイズ (対象とする処理の大きさ) の関係を示します。非常に小さな問題サイズの場合には、プログラム全体がデータキャッシュ (D$) または外部キャッシュ (E$) 中に収まりますが、プログラムのデータ領域が大きくなってくると、32 ビットアプリケーションが利用できる 4G バイト仮想アドレス空間全体をプログラムが占有するようになります。

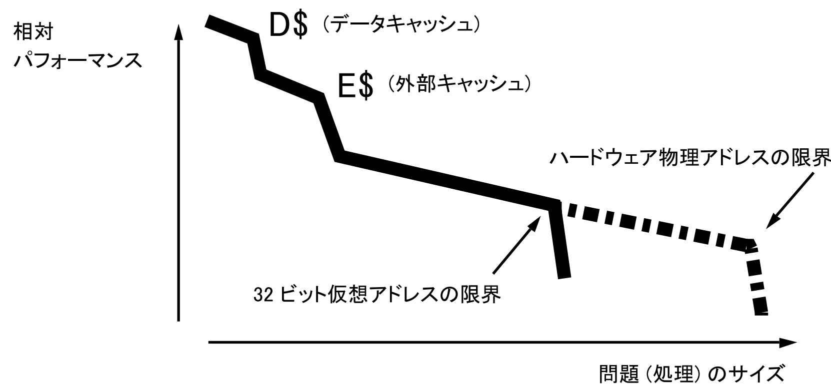


図 **1–1** 典型的なパフォーマンスと問題サイズ曲線

32 ビット仮想アドレスの限界を超えても、アプリケーションはより大きな問題サイズを取り扱うことができます。一般的には 32 ビット仮想アドレスの限界を超えると、一次メモリーと二次メモリー (たとえばディスク) に、アプリケーションデータセットを分割します。 ただし、ディスクドライブにデータを転送するのは、メモリーどうしの転送に比べて非常に長く時間がかかります。

今日では、多くのサーバーが 4 G バイト以上の物理メモリーを取り扱うことができます。高性能のデスクトップコンピュータも同様の傾向にあります。32 ビットプログラムは一度に 4 G バイト以上を直接アドレス指定することはできません。64 ビットアプリケーションでは、64 ビット仮想アドレス空間を使用して 18 エクサバイト (1 エクサバイトは約 10 の 18 乗バイト) まで直接アドレス指定することができます。こうして、サイズが大きな問題を一次メモリー内で直接取り扱うことができます。アプリケーションがマルチスレッド化されていてスケーラブルな場合は、プロセッサを追加してアプリケーションのスピードをさらに上げることができます。そのようなアプリケーションのパフォーマンスは、コンピュータ上の物理メモリーの量によってのみ制限されます。

以下のように、広範囲の種類のアプリケーションにとって、大きな問題を物理メモリー内で直接取り扱うことができるというのは、64 ビットマシンの主要な性能上の特長です。

- データベースの大部分を一次メモリー内に置くことができる。
- 大規模な CAD/CAE モデルとシミュレーションを一次メモリー内に格納できる。
- 大規模な科学技術計算の問題を一次メモリー内に格納できる。
- Web キャッシュはメモリー内により多くを記憶できるので、その結果呼び出しにかかる時間を短縮できる。

「大きなアドレス空間」以外の利点

64 ビットアプリケーションを作成する理由として、次のことを挙げることができます。

- 性能を向上させるために 64 ビットプロセッサのより広いデータパスを使用して、64 ビット整数の演算を大量に行う場合。
- システムインタフェースに使用される基本的データ型が大きくなったので、いくつかのシステムインタフェースが拡張されたり制約が解消された。
- 改善された呼び出し規約や、レジスタセットの完全利用など、64 ビット命令セットの性能上の利点を利用できる。

第 **2** 章

64 ビットの使用方法

アプリケーション開発者にとって、64 ビットおよび 32 ビットの Solaris オペレーティングシステム間の主な相違は、使用されている C データ型モデルです。64 ビット Solaris では、`long` とポインタが 64 ビットである LP64 データ型モデルを使用します。他のすべての基本データ型は、32 ビット実装の場合と同じで、 ILP32 データ型モデルに基づいています。ILP32 データ型モデルでは、 `int`、 `long` とポインタが 32 ビット長になります。これらのデータ型モデルについては、第 3 章でさらに詳しく説明しています。

変換が必要なアプリケーションはあまりありません。ほとんどのアプリケーションは、32 ビットアプリケーションのままでよく、コード変換や再コンパイルせずに 64 ビットオペレーティングシステム上で実行できます。64 ビット機能を必要としない 32 ビットアプリケーションは、移植性を維持するために 32 ビットのままにしておくことをお薦めします。

次のような特性を持っているアプリケーションは、変換した方が有利な場合があります。

- 4G バイト以上の仮想アドレス空間を利用できるアプリケーション
- 32 ビットインタフェースの限界によって制約されるアプリケーション
- 64 ビット演算を効率的に実行するために完全 64 ビットレジスタを利用できるアプリケーション
- 64 ビットの命令セットが提供する、改善された性能を利用できるアプリケーション

次のような特性を持っているアプリケーションは、変換が必要な場合があります。

- `libkvm`、`/dev/mem`、または `/dev/kmem` を使用してカーネルメモリーを読み込み、解釈するアプリケーション
- 64 ビットプロセスをデバッグするために `/proc` を使用するアプリケーション
- 64 ビットバージョンのみで構成されるライブラリを利用するアプリケーション

いくつかの特定の相互運用性のために、コードの変更が必要になります。2 G バイトより大きいファイルを使用しているアプリケーションは、大規模ファイル API を直接使用するのではなく、64 ビットアプリケーションに変換する場合もあります。

これらの項目については以降の節で説明します。

主な特徴

次の図に、Solaris オペレーティングシステムにおいて 32 ビットと 64 ビットの両方がサポートされているしくみを示します。左側のシステムは、32 ビットデバイスドライバを使用した 32 ビットカーネル上で、32 ビットライブラリとアプリケーションのみをサポートします。右側のシステムは、左側と同じ 32 ビットのアプリケーションとライブラリをサポートしますが、64 ビットデバイスドライバを使用した 64 ビットカーネル上で、64 ビットのライブラリとアプリケーションも同時にサポートします。

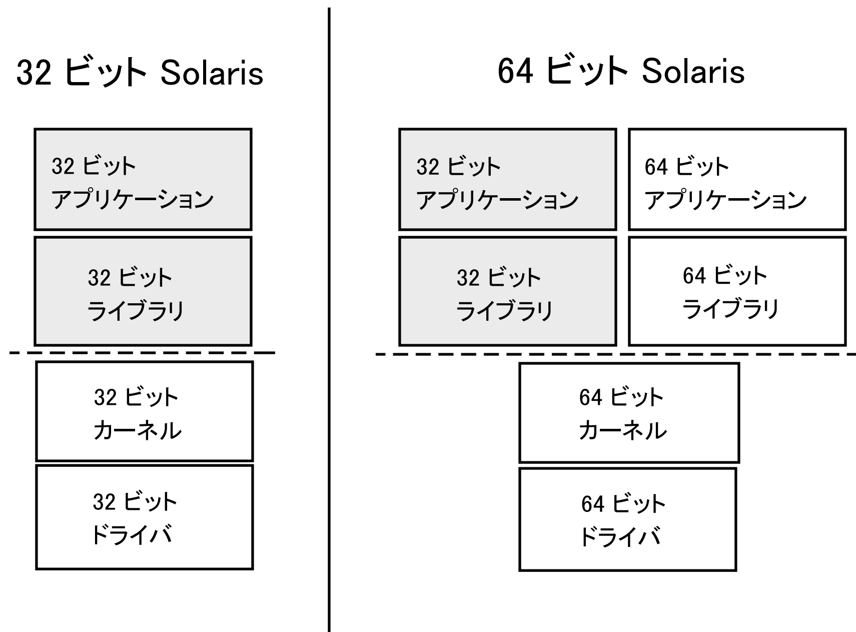


図 **2–1** Solaris オペレーティングシステムのアーキテクチャ

64 ビット環境の主な特徴は、次のとおりです。

- 大容量仮想アドレス空間
- 大規模ファイル
- 64 ビット演算
- 特定のシステム制約の解除

大容量仮想アドレス空間

64 ビット環境では、1 つのプロセスは 64 ビット、すなわち 18 エクサバイトまでの仮想アドレス空間を持つことができます。これは、32 ビットプロセスの現在の最大値のおよそ 40 億倍になります。

注 – ハードウェア上の制約のため、完全な 64 ビットアドレス空間をサポートしていないプラットフォームもあります。

大規模ファイル

アプリケーションが大規模ファイルに対するサポートのみを必要とする場合は、32 ビットのままで、大規模ファイルインタフェースを使用することができます。ただし、移植性がそれほど問題にならない場合には、アプリケーションを 64 ビットプログラムに変換して、整合性のあるインタフェースのセットを備えた 64 ビット機能を活用することもできます。

64 ビット演算

64 ビット演算は、以前の 32 ビット Solaris のリリースでも利用できましたが、64 ビット実装では、完全 64 ビットハードウェアレジスタを整数計算およびパラメータ渡しに利用しています。このため、アプリケーションは 64 ビット CPU ハードウェアの機能を最大限に活用することができます。

特定のシステム制約の解消

64 ビットシステムインタフェースは、本質的に 32 ビットシステムインタフェースの一部のものより機能が優れています。2038 年問題 (32 ビットの `time_t` が時間を使い果すこと) の影響を受けると考えられるアプリケーションのプログラミングでは、64 ビットの `time_t` を利用することができます。2038 年はずっと先の話と思われるでしょうが、抵当ローンのように将来のことがらに関する計算を行うアプリケーションでは、この拡張された時間機能が必要となる場合があります。

相互運用性の問題

アプリケーションを 64 ビット安全にしたり 32 ビットまたは 64 ビットプログラムと相互運用できるように変更する必要が生じるというような相互運用性の問題には、次のものがあります。

- クライアントとサーバーとの転送
- 永続的なデータを操作するプログラム
- 共有メモリー

カーネルメモリーを参照するプログラム

カーネルは 64 ビットデータ構造を内部的に使用する LP64 データ型モデルのオブジェクトであるため、`libkvm`、`/dev/mem`、または `/dev/kmem` を使用する既存の 32 ビットアプリケーションは正しく動作しません。64 ビットプログラムに変換する必要があります。

`/proc` の制約

`/proc` を使用する 32 ビットプログラムは、32 ビットプロセスの情報を認識することができますが、64 ビットプロセスのすべての属性を理解することはできません。プロセスを記述する既存のインタフェースおよびデータ構造は、関係する 64 ビット情報を包含できるほど大きくありません。32 ビットプロセスと 64 ビットプロセスの両方と一緒に動作させるためには、プログラムを 64 ビットプログラムとして再コンパイルする必要があります。この問題をもつ典型的なプログラムはデバッガです。

64 ビットライブラリ

32 ビットアプリケーションは 32 ビットライブラリとリンクする必要があり、64 ビットアプリケーションは 64 ビットライブラリとリンクする必要があります。旧式のライブラリを除いて、すべてのシステムライブラリには 32 ビットと 64 ビットのバージョンがあります。

変換作業の見積もり

アプリケーションを完全に 64 ビットプログラムに変換することを決定した後、ほとんどのアプリケーションにおいて必要な作業はわずかです。以降の章で、アプリケーションおよび変換に関連する作業を確定する方法について説明します。

第 **3** 章

32 ビットと 64 ビットインタフェースの比較

13 ページの「4G バイト境界を超える」で説明したように、ほとんどの 32 ビットアプリケーションは、変更しなくても Solaris 64 ビットオペレーティングシステムで動作します。アプリケーションによっては、64 ビットアプリケーションとして再コンパイルのみが必要なものや、変換する必要があるものもあります。この章では、アプリケーションを再コンパイルしたり、64 ビットに変換する必要がある開発者向けに、13 ページの「4G バイト境界を超える」で述べた項目に基づいて説明します。

アプリケーションプログラミングインタフェース

64 ビットオペレーティング環境でサポートされている 32 ビットアプリケーションプログラミングインタフェース (API) は、32 ビットオペレーティング環境でサポートされている API と同じです。したがって、32 ビットアプリケーションを 32 ビットと 64 ビットの環境間で変更する必要はありません。ただし、64 ビットアプリケーションとして再コンパイルする場合には修正作業が必要な場合があります。64 ビットアプリケーション用にコードを修正するためのガイドラインについては、第 4 章を参照してください。

デフォルトの 64 ビット API とは、基本的に UNIX 98 ファミリーの API です。その仕様は、派生型を使って書かれています。64 ビットバージョンは、それら派生型のいくつかを 64 ビットに拡張することで作られています。これらの API を使って正しく書かれたアプリケーションは、32 ビットと 64 ビット間でソースを移植できます。Solaris 10 では UNIX 2001 API ファミリーも使用できます (`standards`(5) を参照)。

アプリケーションバイナリインタフェース

SPARC™ V8 ABI は、既存のプロセッサ固有のアプリケーションバイナリインタフェース (ABI) で、32 ビット SPARC バージョンの Solaris 実装はこのインタフェースに基づいています。SPARC V9 ABI は、SPARC V8 ABI を拡張して 64 ビット動作をサポートし、拡張アーキテクチャの新しい機能を定義しています。詳細は、57 ページの「SPARC V9 ABI の特徴」を参照してください。

i386 ABI はプロセッサ固有の ABI で、32 ビットバージョンの Solaris (x86 プラットフォーム版) はこのインタフェースに基づいています。

Solaris 10 リリースでは、x86 システム上の 64 ビットバージョンの Solaris はプロセッサ固有の ABI である amd64 ABI に基づいています。amd64 ABI は 64 ビット動作をサポートし、新しいアーキテクチャの新しい機能を定義しています。64 ビット ABI を使用するプログラムは、32 ビットのプログラムより動作が優れることがあります。amd64 ABI をサポートするプロセッサは i386 ABI もサポートします。詳細は、62 ページの「AMD64 ABI の特徴」を参照してください。

32 ビットアプリケーションと 64 ビットアプリケーション間の互換性

以降の節では、32 ビットと 64 ビットアプリケーション間のさまざまなレベルの互換性について説明します。

アプリケーションバイナリ

既存の 32 ビットアプリケーションは、32 ビットまたは 64 ビットのどちらのオペレーティング環境でも実行できます。唯一の例外は、`libkvm`、`/dev/mem`、`/dev/kmem`、または `/proc` を使用するアプリケーションです。詳細は、13 ページの「4G バイト境界を超える」を参照してください。

アプリケーションソースコード

32 ビットアプリケーションに対しては、ソースレベルの互換性が維持されています。64 ビットアプリケーションについては、アプリケーションプログラミングインタフェースに使用される派生型に変更が加えられています。派生型およびインタフェースを正しく使用しているアプリケーションは、32 ビットに対してソースレベルで互換性があり、より容易に 64 ビットに移行することができます。

デバイスドライバ

32 ビットデバイスドライバは、64 ビットオペレーティングシステムでは使用できないため、32 ビットデバイスドライバは、64 ビットオブジェクトとして再コンパイルしなければなりません。さらに、64 ビットドライバは、32 ビットと 64 ビットの両方のアプリケーションをサポートしなければなりません。64 ビットオペレーティング環境に提供されているドライバはすべて、32 ビットと 64 ビットの両方のアプリケーションをサポートします。ただし、基本ドライバモデル、および DDI (Device Driver Interface) でサポートされているインタフェースに変更はありません。必要な作業は、LP64 データ型モデルの環境で適切となるようにコードを修正することです。詳細は、『*Writing Device Drivers*』を参照してください。

どちらの Solaris オペレーティングシステムが実行されているか

Solaris オペレーティングシステムは、2 つのアプリケーションバイナリインタフェース (ABI) を同時にサポートしています。言い換えれば、2 つの独立した、完全に機能するシステムコールパスが、64 ビットカーネルに接続されており、2 組のライブラリがアプリケーションをサポートします。

64 ビットオペレーティングシステムは、64 ビット CPU ハードウェア上でのみ動作しますが、32 ビットオペレーティングシステムは、32 ビットと 64 ビットのどちらの CPU ハードウェア上でも動作します。Solaris の 32 ビットと 64 ビットのオペレーティング環境は同じように見えるので、特定のハードウェアプラットフォーム上でどちらのバージョンが動作しているかすぐに判断できないことがあります。

`isainfo` コマンドを使用すると、システムで実行されているバージョンを簡単に確認できます。この新しいコマンドは、システムでサポートされているアプリケーション環境に関する情報を出力します。

次に示すのは、64 ビットオペレーティングシステムが実行されている UltraSPARC™ システムで実行した `isainfo` コマンドの出力例です。

```
% isainfo -v
64-bit sparcv9 applications
```

```
32-bit sparc applications
```

32 ビット Solaris オペレーティングシステムが実行されている x86 システムで同じコマンドを実行した場合は、次のように出力されます。

```
% isainfo -v
32-bit i386 applications
```

64 ビット Solaris オペレーティングシステムが実行されている x86 システムで同じコマンドを実行した場合は、次のように出力されます。

```
% isainfo -v
64-bit amd64 applications
32-bit i386 applications
```

注 – 64 ビットカーネルを実行できない x86 システムもあります。そのシステムで Solaris オペレーティングシステムを実行する場合、カーネルは 32 ビット モードで実行されます。

---

isainfo(1) コマンドの便利なオプションとして -n があり、実行中のプラットフォームのネイティブ命令セットを出力できます。

```
% isainfo -n
sparcv9
```

-b オプションを使用すると、対応するネイティブのアプリケーション環境のアドレス空間のビット幅を、次のように出力できます。

```
% echo "Welcome to "`isainfo -b`"-bit Solaris"
Welcome to 64-bit Solaris
```

64 ビット機能があるかどうかを判定することによって、以前のバージョンの Solaris オペレーティングシステムで実行しなければならないアプリケーションであるかどうかを判断することができます。uname(1) で OS のバージョンを調べるか、または /usr/bin/isainfo が存在するかどうかを調べます。

関連コマンドの isalist(1) は、シェルスクリプトで使用するのに適しており、プラットフォームでサポートされている命令セットをすべて出力するのに使うことができます。ただし、命令セットの拡張の数が増加するに従い、すべてのサブセットを一覧表示することの限界が明らかになっています。今後、このインタフェースに依存しないようにしてください。

命令セットの拡張に依存するライブラリを作成するには、ダイナミックリンカーのハードウェア機能を使用します。isainfo コマンドを使用して、現在のプラットフォーム上の命令セットの拡張を確認します。

```
% isainfo -x
amd64: sse2 sse fxsr amd_3dnowx amd_3dnow amd_mmx mmx cmov amd_sysc cx8 tsc fpu
i386: sse2 sse fxsr amd_3dnowx amd_3dnow amd_mmx mmx cmov amd_sysc cx8 tsc fpu
```

第 4 章

アプリケーションの変換

32 ビットから 64 ビットに変換アプリケーションを変換する際には、次の 2 つの基本的な問題があります。

- データ型の整合性および異なるデータ型モデル
- 異なるデータ型モデルを使ったアプリケーション間の相互運用

通常は、できるだけ少ない数の #ifdef を使って 1 つのソースだけを管理する方が、複数のソースツリーを管理するよりも便利です。この章では、32 ビット環境と 64 ビット環境の両方で正しく動作するコードを書くためのガイドラインを示します。既存のコードを変換するのに必要なことは、再コンパイルして、64 ビットライブラリと再リンクするだけです。コードの変更が必要になった場合のために、変換を簡単に実行するためのツールについてもこの章で説明します。

データ型モデル

すでに説明したように、32 ビットと 64 ビットの環境の大きな違いは、データ型モデルです。

32 ビットアプリケーションに使用される C データ型モデルは ILP32 で、int、long、およびポインタが 32 ビットであるためそのように呼ばれています。LP64 データ型モデルは、64 ビットアプリケーション用の C データ型モデルで、業界の企業コンソーシアムで合意を得ています。この LP64 データ型モデルは、long とポインタが 64 ビットに拡大されたためそのように呼ばれています。ほかの C データ型 int、short、char は、ILP32 データ型モデルと同じです。

次に示すプログラミング例 foo.c では、LP64 と ILP32 のデータ型モデルの効果を直接的に示しています。同じプログラムを 32 ビットまたは 64 ビットプログラムとしてコンパイルすることができます。

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

```
main(int argc, char *argv[])
{
        (void) printf("char is \t\t%lu bytes\n", sizeof (char));
        (void) printf("short is \t%lu bytes\n", sizeof (short));
        (void) printf("int is \t\t%lu bytes\n", sizeof (int));
        (void) printf("long is \t\t%lu bytes\n", sizeof (long));
        (void) printf("long long is \t\t%lu bytes\n", sizeof (long long));
        (void) printf("pointer is \t%lu bytes\n", sizeof (void *));
        return (0);
}
```

32 ビットコンパイルの結果は、次のようになります。

```
% cc -O -o foo32 foo.c
% foo32
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          4 bytes
long long is     8 bytes
pointer is       4 bytes
```

64 ビットコンパイルの結果は、次のようになります。

```
% cc -xarch=generic64 -O -o foo64 foo.c
% foo64
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          8 bytes
long long is     8 bytes
pointer is       8 bytes
```

注 – デフォルトのコンパイル環境は、移植性を最大限にするように設計されているため、32 ビットのアプリケーションを作成します。

次に示すように C の各整数データ型間の標準的な関係は変わりません。

```
sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeof (long)
```

表 4–1 に C の基本データ型、およびそれらの LP32 と LP64 のデータ型モデルのビットサイズを示します。

表 4–1 データ型サイズ (単位 : ビット)

C データ型	ILP32	LP64
char	8	変更なし
short	16	変更なし

表 4–1 データ型サイズ (単位 : ビット)　(続き)

C データ型	ILP32	LP64
`int`	32	変更なし
`long`	32	**64**
`long long`	64	変更なし
ポインタ	32	**64**
`enum`	32	変更なし
`float`	32	変更なし
`double`	64	変更なし
`long double`	128	変更なし

`int`、`long`、およびポインタ型を区別なく使用する旧式の 32 ビットアプリケーションもあります。`long` とポインタのサイズが LP64 データ型モデルで大きくなっています。この点から、32 ビットから 64 ビットへの多くの変換の問題が発生する可能性があるということを認識する必要があります。

さらに意図するプログラム処理を示すためには、宣言とキャストが重要になります。たとえば、データ型が変わると式の評価方法が影響を受ける可能性があります。データ型のサイズが変更された場合には、C の標準の変換規則は影響を受けます。意図する内容を明確に示すには、定数の型を宣言する必要があります。キャストを式に入れることによって、式を確実に意図するように評価させることも必要です。これは特に、符号拡張の場合に当てはまります。この場合、目的の処理を正しく示すには、明示的にキャストする必要があります。

その他の問題としては、組み込みの C 演算子、書式文字列、アセンブリ言語、互換性、および相互運用性の問題があります。

この章の以降の節で、次のようにこれらの問題の対処方法を紹介します。

- これまで概要を示した問題の詳細な説明
- コードを 32 ビットおよび 64 ビットに対して安全にするのに有用な、派生型とインクルードファイルの解説
- コードを 64 ビット安全にするためのツールの紹介
- 32 ビットおよび 64 ビット環境間でコードを移植可能にするための一般的規則

単一ソースコードの実装

32 ビットおよび 64 ビットコンパイルをサポートする単一ソースコードを書く際に役立つ、アプリケーション開発者向けの資源について説明します。

システムインクルードファイル `<sys/types.h>` と `<inttypes.h>` には、アプリケーションを 32 ビット および 64 ビット安全にするために使用できる定数、マクロ、および派生型が含まれています。これらの詳細についてはこのマニュアルでは説明していませんが、その一部はこの章の以降の節および付録 A で説明しています。

機能テストマクロ

`<sys/types.h>` をインクルードするアプリケーションのソースファイルでは、`<sys/isa_defs.h>` をインクルードすることによってプログラミングモデルシンボル `_LP64` と `_ILP32` の定義を利用できるようになります。

プリプロセッサシンボル (`_LP64` と `_ILP32`) およびマクロ (`_LITTLE_ENDIAN` と `_BIG_ENDIAN`) については、`types` (3HEAD) を参照してください。

派生型

システム派生型は、コードを 32 ビットおよび 64 ビット安全にするのに便利です。これは、派生型自身が ILP32 および LP64 のデータ型モデルに対して安全であるからです。一般に、変更を可能にするために派生型を使用しておくと便利です。後でデータ型モデルが変更された場合に、または異なるプラットフォームに移植する場合に、アプリケーションそのものではなく、システム派生型を変更するだけで済みます。

`<sys/types.h>` ファイル

`<sys/types.h>` ヘッダーには、必要に応じて使用される多数の基本的な派生型が含まれています。特に次のものは重要です。

`clock_t` システムの時間をクロック刻み (clock tick) で表します。

`dev_t` デバイス番号に使用される型です。

`off_t` ファイルサイズとオフセット用に使用される型です。

`ptrdiff_t` 2 つのポインタの減算結果を示す符号付き整数型です。

`size_t` メモリー内のオブジェクトのサイズ (バイト単位) 用に使用される型です。

ssize_t　バイト数またはエラーのどちらを返すこともある関数によって使用される「符号付きサイズ」型です。

time_t　秒単位の時間用に使用される型です。

これらの型はすべて、ILP32 コンパイル環境では 32 ビット、LP64 コンパイル環境では 64 ビットになります。

これらの型の一部の使用方法については、36 ページの「LP64 への変換のためのガイドライン」で詳しく説明しています。

<inttypes.h> ファイル

定数、マクロ、および派生型を定義するために、インクルードファイル <inttypes.h> が Solaris 2.6 リリースに追加されました。これにより、コンパイル環境とは無関係に、プログラマが記述したコードをサイズ指定されたデータ項目と明示的に互換性を持たせることができます。このファイルには、8 ビット、16 ビット、32 ビット、および 64 ビットのオブジェクトを操作するための機構が含まれてます。このインクルードファイルは、ANSI C の原案の一部で、ISO/JTC1/SC22/WG14 C 委員会による現在の ISO C 標準、つまり ISO/IEC 9899:1990 プログラミング言語 - C の改訂案を反映しています。

<inttypes.h> の主な機能は、次のとおりです。

- 固定幅整数型の集合
- uintptr_t とその他の有用なデータ型
- 定数マクロ
- 制限値
- 書式文字列マクロ

これらについては以降の節で説明します。

固定幅整数型

<inttypes.h> で提供される固定幅整数型には、int8_t、int16_t、int32_t、int64_t、uint8_t、uint16_t、uint32_t、uint64_t などの符号付き整数型および符号なし整数型があります。特定のビット数を格納できる最小の整数型として定義される派生型には、int_least8_t、int_least64_t、uint_least8_t、uint_least64_t があります。

これらの固定幅型を無制限に使用しないでください。たとえば、int はこれまでと同様に、ループカウンタやファイル記述子などについて使用でき、long は配列のインデックスに使用できます。固定幅型は、次に示すような明示的なバイナリ表現に使用してください。

- ディスク上のデータ

- 送受信データ
- ハードウェアレジスタ
- バイナリインタフェース仕様 (明示的にサイズの決められたオブジェクトがあるもの、または 32 ビットプログラムと 64 ビットプログラム間での共有や通信を含むもの)
- バイナリデータ構造 (32 ビットプログラムおよび 64 ビットプログラムが、共有メモリー、共有ファイルなどを介して使用するもの)

uintptr_t とその他の有用なデータ型

<inttypes.h> によって提供されるその他の型として、ポインタを格納するために十分なサイズの符号付き整数型および符号なし整数型があります。これらの型には、intptr_t と uintptr_t があります。さらに、intmax_t および uintmax_t という (ビット単位で) 最長の符号付きおよび符号なしデータ型があります。

uintptr_t 型をポインタ用の整数型として使用する方が、unsigned long のような基本データ型を使用するよりも便利です。unsigned long は、IPL32 と LP64 データ型モデルの両方でポインタと同じサイズですが、uintptr_t を使用すると、uintptr_t の定義を変更するだけで異なるデータ型モデルを使用できます。このため、他の多くのシステムに移植が可能となります。またこれによって、C プログラムコード中に意図する処理をより明確に記述することができます。

intptr_t と uintptr_t 型は、アドレス計算をする際にポインタをキャストするのに非常に役に立ちます。long または unsigned long の代わりにこれらを使用することができます。

注 – 通常は、uintptr_t を使用してキャストする方が、intptr_t を使用するよりも安全です。特に比較の場合はこの方法が安全です。

定数マクロ

マクロは、定数のサイズと符号を指定するために使用できます。マクロには、INT8_C(c)、...、INT64_C(c)、UINT8_C(c)、...、UINT64_C(c) があります。基本的にこれらのマクロは、必要な場合に定数の後ろに l、ul、ll、または ull を置きます。たとえば、INT64_C(1) は、定数 1 の後ろに ILP32 の場合は ll を、LP64 の場合は l を付加します。

定数を最大のデータ型にするためのマクロには、INTMAX_C(c) と UINTMAX_C(c) があります。これらのマクロは、36 ページの「LP64 への変換のためのガイドライン」で説明している定数の型を指定するのに非常に役に立ちます。

<inttypes.h> によって定義される制限値

<inttypes.h> に定義されている制限値は、さまざまな整数型の最小値および最大値を指定する定数です。このファイルには、INT8_MIN、…、 INT64_MIN、INT8_MAX、…、INT64_MAX、およびこれらの符号なし定数の、各固定幅型の最小値と最大値が指定されています。

最小サイズ型のそれぞれの最小値と最大値も指定されています。すなわち、INT_LEAST8_MIN、…、 INT_LEAST64_MIN、INT_LEAST8_MAX、…、INT_LEAST64_MAX、およびこれらの符号なし定数です。

サポートされている整数型のうちの最大の型の最小値と最大値も定義されています。これらには、INTMAX_MIN と INTMAX_MAX、およびそれらの符号なしのものがあります。

書式文字列マクロ

printf と scanf の書式指示子を指定するためのマクロも <inttypes.h> にあります。これらのマクロは、引数のビット数がマクロ名に組み込まれている場合に、初期指示子の先頭に l または ll を付加することによって引数を long または long long として指定します。

printf(3C) 書式指示子用のマクロは、10 進、8 進、符号なし、16 進の、8 ビット、16 ビット、32 ビット、64 ビットの整数、最小整数型と最大整数型を出力するためのものです。64 ビットの整数を 16 進表記で出力する例を、次に示します。

```
int64_t i;
printf("i =%" PRIx64 "\n", i);
```

同様に、scanf(3C) 書式指示子用のマクロが、10 進、8 進、符号なし、および 16 進の 8 ビット、16 ビット、32 ビット、64 ビットの整数、ならびに最小整数型と最大整数型の読み込み用に提供されています。符号なし 64 ビットの 10 進整数を読み込む例を、次に示します。

```
uint64_t u;
scanf("%" SCNu64 "\n", &u);
```

これらのマクロは、無制限に使用しないでください。これらは固定幅型と一緒に使用するのが最適な使用方法です。詳細は、31 ページの「固定幅整数型」を参照してください。

ツールのサポート

Sun Studio 10 コンパイラで使用できる lint プログラムは、発生する可能性のある 64 ビット関連の問題を検出することができるので、コードを 64 ビット安全にするのに便利です。また、C コンパイラの -v コンパイルオプションも便利です。このオプションを使用すると、コンパイル時に通常のチェックに加えて、より厳しい意味解析上のチェックを行うことができます。さらに、引数として指定したファイルに対して lint に似たチェックも実行します。

C コンパイラの機能と lint については、『*Sun Studio 10: C ユーザーズガイド*』を参照してください。

32 ビットと 64 ビット環境の lint

lint は、32 ビットコードおよび 64 ビットコードの両方に使用することができます。32 ビット環境および 64 ビット環境の両方で実行するコードには、-errchk=longptr64 オプションを使用します。-errchk=longptr64 オプションは、ロング整数とポインタのサイズが 64 ビットで、かつ普通の整数が 32 ビットである環境への移植性を調べるのに使用します。

-Xarch=v9 オプションは、64 ビット SPARC 環境で実行する lint コードに対して使用します。64 ビット SPARC 上で実行するコードに対して、発生する可能性がある 64 ビット関連の問題について警告を表示するようにするには、-Xarch=v9 オプションと共に -errchk=longptr64 オプションを使用します。

Solaris 10 リリースから、-Xarch=amd64 オプションは 64 ビット AMD 環境で実行する lint コードに対して使用します。

注 – lint には -D__sparcv9 オプションを使用しないでください。

---

lint オプションの説明は、『*Sun Studio 10: C ユーザーズガイド*』を参照してください。

警告がある場合、lint(1) は、エラーが発生した行の行番号、問題を説明する警告メッセージ、およびポインタが関わっているかどうかを出力します。関連する型のサイズも示されます。ポインタが関わっているかどうかおよび型のサイズを知ることは、64 ビット関連の問題を特定し、さらに 32 ビットとそれより小さい型との間の既存の問題を避けるのに役に立ちます。

注 – lint は発生する可能性がある 64 ビット関連の問題に関して警告を出すことはできますが、問題をすべて検出できるわけではありません。また lint が出力する警告の中には 64 ビット関連以外の問題が含まれていることもあります。警告が出されていても、そのコードは特定の意図に沿って記述されていてアプリケーションにとって適切なコードである、という場合がよくあります。

次のサンプルプログラムと lint(1) 出力は、64 ビットクリーンコード以外のコードで発生する lint 警告のよくある例を示したものです。

```
 1    #include <inttypes.h>
 2    #include <stdio.h>
 3    #include <stdlib.h>
 4
 5    static char chararray[] = "abcdefghijklmnopqrstuvwxyz";
 6
 7    static char *myfunc(int i)
 8    {
 9        return(& chararray[i]);
10    }
11
12    void main(void)
13    {
14        int    intx;
15        long    longx;
16        char    *ptrx;
17
18        (void) scanf("%d", &longx);
19        intx = longx;
20        ptrx = myfunc(longx);
21        (void) printf("%d\n", longx);
22        intx = ptrx;
23        ptrx = intx;
24        intx = (int)longx;
25        ptrx = (char *)intx;
26        intx = 2147483648L;
27        intx = (int) 2147483648L;
28        ptrx = myfunc(2147483648L);
29    }
```

```
(19) warning: assignment of 64-bit integer to 32-bit integer
(20) warning: passing 64-bit integer arg, expecting 32-bit integer: myfunc(arg 1)
(22) warning: improper pointer/integer combination: op "="
(22) warning: conversion of pointer loses bits
(23) warning: improper pointer/integer combination: op "="
(23) warning: cast to pointer from 32-bit integer
(24) warning: cast from 64-bit integer to 32-bit integer
(25) warning: cast to pointer from 32-bit integer
(26) warning: 64-bit constant truncated to 32 bits by assignment
(27) warning: cast from 64-bit integer constant expression to 32-bit integer
(28) warning: passing 64-bit integer constant arg, expecting 32-bit integer: myfunc(arg 1)
```

```
function argument ( number ) type inconsistent with format
    scanf (arg 2)     long * :: (format) int *    t.c(18)
    printf (arg 2)     long  :: (format) int     t.c(21)
```

(このコードサンプルの 27 行目の lint 警告は、定数式がキャストされる型に当てはまらないときにのみ出力されます。)

/*LINTED*/ コメントをその前の行に置くと、任意のソース行に対する警告を抑止できます。これは、意図的に特別な動作をコード中に記述したい場合には役に立ちます。例としては、キャストや代入の場合があります。/*LINTED*/ コメントは、実際に問題がある場合にもそれを検出しないようにするので、使用する際は十分に注意してください。詳細は、『*Sun Studio 10: C* ユーザーズガイド』または lint(1) のマニュアルページを参照してください。

LP64 への変換のためのガイドライン

lint(1) を使用する際には、すべての問題が lint(1) によって警告として検出されるわけではないこと、変更が不要な点についても lint(1) によって警告として出力されることがある、ということを憶えておいてください。警告の内容は、目的と照らし合わせて調べてください。これから示す例では、コードを変換する際に遭遇する可能性が高い問題を説明します。適切な場所で、lint(1) に相当する警告が現れます。

int とポインタが同じサイズであると仮定しない

int とポインタは、ILP32 環境では同じサイズであるため、多くのコードがこの仮定に基づいています。ポインタは、アドレス計算の際に int または unsigned int にキャストされることがあります。また、ポインタは long にキャストすることもできます。long とポインタは、ILP32 および LP64 で同じサイズだからです。unsigned long を明示的に使うかわりに、uintptr_t を使用してください。uintptr_t は、意図することがより明確にわかり、コードをより移植可能なものにして、その結果、将来変更があっても影響されないようにするためです。次に例を示します。

```
char *p;
p = (char *) ((int)p & PAGEOFFSET);
```

この場合、次の警告が出ます。

```
warning: conversion of pointer loses bits
```

次のコードを使用すると、正しい結果が出ます。

```
char *p;
p = (char *) ((uintptr_t)p & PAGEOFFSET);
```

int と long が同じサイズであると仮定しない

int と long は、ILP32 では実際には区別されないため、意図的あるいは非意図的にそれらは交換可能であると仮定して、既存のコードの多くで区別することなく使用されています。このように仮定しているコードは、ILP32 および LP64 で動作するように変更する必要があります。ILP32 データ型モデルでは int と long の両方が 32 ビットですが、LP64 データ型モデルでは long は 64 ビットです。次に例を示します。

```
int waiting;
long w_io;
long w_swap;
...
waiting = w_io + w_swap;
```

この場合、次の警告が出ます。

```
warning: assignment of 64-bit integer to 32-bit integer
```

符号の拡張

意図しない符号の拡張は、64 ビットに変換する際によく発生する問題です。符号の拡張について lint(1) は警告を出さないため、実際に問題が発生する前に問題を検出するのは困難です。さらに、型変換および型昇格に関する規則には不明瞭な部分もあります。意図しない符号拡張の問題を解決するには、意図する結果を得ることができるように明示的なキャストを使用する必要があります。

ANSI C の変換規則を理解しておくと、なぜ符号の拡張が発生するかを理解するのに役立ちます。32 ビットおよび 64 ビットの整数値間において、符号拡張の問題の原因になることがある変換規則は、次のとおりです。

1. 整数の昇格

   char、short、列挙型、またはビットフィールド型は、符号付き / 符号なしに関わらず、int を必要とする式の中に使用できます。int が元の型の取り得る値をすべて格納できる場合、その値は int に変換されます。そうでない場合は、unsigned int に変換されます。

2. 符号付きおよび符号なし整数間の変換

   負の符号付き整数が、サイズが同じまたはより大きい型の符号なし整数に昇格される場合、最初に大きい型の符号付きの値に昇格され、その後符号なしの値に変換されます。

変換規則についての詳細は、ANSI C 規格を参照してください。この規格には、通常の算術変換や整数定数についての規則が規定されています。

64 ビットプログラムとしてコンパイルした場合、次の例の addr 変数は、addr および a.base が符号なしの型であっても符号付きの型に拡張されます。

例 **4–1** test.c

```
struct foo {
        unsigned int    base:19, rehash:13;
};

main(int argc, char *argv[])
{
        struct foo    a;
        unsigned long addr;

        a.base = 0x40000;
        addr = a.base << 13;        /* 符号拡張が発生する */
        printf("addr 0x%lx\n", addr);

        addr = (unsigned int)(a.base << 13);  /* 符号拡張が発生しない */
        printf("addr 0x%lx\n", addr);
}
```

このように符号拡張が発生するのは、変換規則が次のように適用されるからです。

1. `a.base` が、整数の昇格規則によって、`unsigned int` から `int` に変換されます。このため、式 `a.base << 13` は `int` 型ですが、符号拡張はまだ発生していません。
2. 式 `a.base << 13` は、`int` 型ですが、符号付きおよび符号なしの整数昇格規則によって最初に `long` へ変換され、その後 `unsigned long` へ変換された後、`addr` に代入されます。符号拡張は、この式が `int` から `long` に変換されるときに発生します。

```
% cc -o test64 -xarch=v9 test.c
% ./test64
addr 0xffffffff80000000
addr 0x80000000
%
```

同じ例題が 32 ビットプログラムとしてコンパイルされた場合、符号拡張は発生しません。

```
% cc -o test32 test.c
% ./test32
addr 0x80000000
addr 0x80000000
%
```

アドレス演算の代わりにポインタ演算を使う

一般に、ポインタ演算を使用した場合の方が、アドレス演算を使用した場合よりもうまく機能します。この理由は、ポインタ演算はデータ型モデルに依存しませんが、アドレス演算はデータ型モデルに依存するためです。さらにポインタ演算の方がコードを簡潔に記述することができます。次に例を示します。

```
int *end;
int *p;
p = malloc(4 * NUM_ELEMENTS);
end = (int *)((unsigned int)p + 4 * NUM_ELEMENTS);
```

この場合、次の警告が出ます。

```
warning: conversion of pointer loses bits
```

次のコードを使用すると、正しい結果が出ます。

```
int *end;
int *p;
p = malloc(sizeof (*p) * NUM_ELEMENTS);
end = p + NUM_ELEMENTS;
```

構造体の再構成

LP64 データ型モデルでは `long` およびポインタフィールドが 64 ビットに拡張されるため、コンパイラが構造体にパディングを追加して、境界を整列することがあります。SPARCV9 ABI と amd64 ABI のどちらでも、構造体の型はすべて、最低でも構造体内での最大サイズに整列されます。構造体を再構成するための簡単な規則は、`long` とポインタのフィールドを構造体の先頭位置に移動して、残りのフィールドを整列し直すことです。通常はサイズの大きい方から順に整列しますが、どれほどうまく詰め込めるかによって異なります。次に例を示します。

```
struct bar {
        int i;
        long j;
        int k;
        char *p;
};          /* sizeof (struct bar) = 32 */
```

より良い結果を得るには、次のコードを使用します。

```
struct bar {
        char *p;
        long j;
        int i;
        int k;
};          /* sizeof (struct bar) = 24 */
```

注 – 基本的な型の整列は、i386 と amd64 ABI 間で異なります。64 ページの「整列の問題」を参照してください。

共用体のチェック

共用体フィールドは、ILP32 と LP64 とでサイズが変更されているので、必ず確認してください。

```
typedef union {
       double   _d;
       long _l[2];
} llx_t;
```

このコードは、次のように使用してください。

```
typedef union {
       double _d;
       int _l[2];
} llx_t;
```

定数の型指定

一部の定数式では、精度が不足するためにデータが失われる可能性があります。このような問題を検出するのは非常に困難です。定数式にデータ型を明示的に指定してください。各整数定数の後に (u、U、l、L) を組合せたものを追加します。キャストを使用して定数式のデータ型を指定することもできます。次に例を示します。

```
int i = 32;
long j = 1 << i;        /* 右辺が int 型のため j には 0 が代入される */
```

このコードは、次のように使用してください。

```
int i = 32;
long j = 1L << i;
```

暗黙的宣言について

特定のコンパイルモードのコンパイラは、モジュールに使用されかつ extern として定義または宣言されていない関数または変数に対してそのデータ型を int とみなします。long およびポインタがこのように使用された場合、コンパイラの暗黙的な int 宣言によって切り捨てられます。関数または変数に対する適切な extern 宣言は、C モジュール中にではなくヘッダーに置いてください。このヘッダーは、関数または変数を使用する C モジュールがインクルードするようにしてください。これがシステムヘッダーに定義されている関数または変数であっても、適当なヘッダーをコード内にインクルードしてください。

getlogin() が宣言されていないコードの例を次に示します。

```
int
main(int argc, char *argv[])
{
        char *name = getlogin()
        printf("login = %s\n", name);
        return (0);
}
```

この場合、次の警告が出ます。

```
warning: improper pointer/integer combination: op "="
warning: cast to pointer from 32-bit integer
implicitly declared to return int
getlogin        printf
```

より良い結果を得るには、次のコードを使用します。

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

int
main(int argc, char *argv[])
{
        char *name = getlogin();
        (void) printf("login = %s\n", name);
        return (0);
}
```

sizeof は unsigned long である

LP64 環境では、sizeof は、unsigned long として実装される size_t の実効的なデータ型を持ちます。sizeof は、int 型の引数を期待する (受け取る) 関数に渡されたり、int に代入またはキャストされることがあります。このような切り捨てによってデータが失われることがあります。次に例を示します。

```
long a[50];
unsigned char size = sizeof (a);
```

この場合、次の警告が出ます。

```
warning: 64-bit constant truncated to 8 bits by assignment
warning: initializer does not fit or is out of range: 0x190
```

意図を示すためにキャストを使う

関係式には変換規則があるので、注意を要します。必要に応じてキャストを追加して、式をどのように評価するかを明示的に記述してください。

書式文字列の変換をチェックする

printf(3C)、sprintf(3C)、 scanf(3C)、 sscanf(3C) が long またはポインタ引数に対して使用される場合、それらを long またはポインタ引数用に変更する必要がある場合があります。ポインタ引数を 32 ビットおよび 64 ビット環境で動作させるためには、書式文字列に指定する変換操作を %p にしてください。次に例を示します。

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, "di%x", (void *)devi);
```

この場合、次の警告が出ます。

```
warning: function argument (number) type inconsistent with format
  sprintf (arg 3)     void *: (format) int
```

次のコードを使用すると、正しい結果が出ます。

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, "di%p", (void *)devi);
```

さらに、buf によって示される記憶領域に 16 桁を格納できる大きさが十分にあることを確認してください。long 引数の場合は、long サイズ指定子 l を書式化文字列の変換操作文字の前に追加してください。次に例を示します。

```
    size_t nbytes;
    ulong_t align, addr, raddr, alloc;
    printf("kalloca:%d%%%d from heap got %x.%x returns %x\n",
            nbytes, align, (int)raddr, (int)(raddr + alloc), (int)addr);
```

この場合、次の警告が出ます。

```
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
```

次のコードを使用すると、正しい結果が出ます。

```
size_t nbytes;
ulong_t align, addr, raddr, alloc;
printf("kalloca:%lu%%%lu from heap got %lx.%lx returns %lx\n",
        nbytes, align, raddr, raddr + alloc, addr);
```

その他の考慮事項

アプリケーションを完全に 64 ビットプログラムに変換する際によく発生する、その他の問題を取り上げます。

サイズが拡大した派生型

64 ビットアプリケーション環境で 64 ビットを表すために、多くの派生型が変更されました。この変更は、 32 ビットアプリケーションには影響を与えませんが、これらの型で記述されるデータを使用またはエクスポートする 64 ビットアプリケーションは、再評価して正しく動作することを確認する必要があります。たとえば、utmpx(4) ファイルを直接操作するアプリケーションについては再確認が必要です。64 ビットアプリケーション環境で正しく動作するように、これらのファイルへ直接アクセスしないようにしてください。代わりに getutxent(3C) および関連する関数を使用してください。

付録 A には、変更された派生型の一覧が記載されています。

変更による副作用をチェックする

ある部分で型の変更を行なった結果、別の部分で予期しない 64 ビット 変換が起きることがあります。たとえば、以前には int を戻していたが現在は ssize_t を返す関数に対しては、すべての呼び出し側をチェックする必要があります。

long を使用する意味があるかどうかチェックする

long は ILP32 データ型モデルでは 32 ビット、LP64 データ型モデルでは 64 ビットなので、以前は long として定義されたものが不適切または不要になることがあります。このような場合は、より移植性の高い派生型を使うこともできます。

上述の理由で、LP64 データ型モデルにおいて多くの派生型が変更されている場合があります。たとえば、pid_t は 32 ビット 環境では long のままですが、64 ビット 環境では int です。LP64 コンパイル環境用に変更された派生型のリストについては、付録 A を参照してください。

明示的な 32 ビット 対 64 ビット プロトタイプのために #ifdef を使う

32 ビットおよび 64 ビット用にそれぞれ固有のインタフェースが必要な場合があります。ヘッダーで _LP64 または _ILP32 という機能テストマクロを使用することによって、それぞれのインタフェースを設けることができます。同様に、32 ビットおよび 64 ビット環境で動作させるコードに、それぞれのコンパイル環境に応じて適切な #ifdef を使用する必要がある場合もあります。

アルゴリズムの変更

コードを 64 ビット安全にした後、コードを再検討して、アルゴリズムおよびデータ構造が意図どおりであることを確認してください。データ型が大きいほど、データ構造体はより大きい空間を使用します。コードのパフォーマンスも同様に変化する場合があります。これらのことを考えて、コードを適切に修正する必要があるかもしれません。

チェックリスト (64 ビットに変換する前に)

以下の各項目を確認していくことによって、コードを 64 ビットに変換する必要があるかどうかを判断することができます。

- このマニュアル全体をお読みください。特に 36 ページの「LP64 への変換のためのガイドライン」に重点を置いて読んでください。
- すべてのデータ構造体とインタフェースを再検討して、それらが 64 ビット環境でも有効であることを確認してください。
- <sys/types.h> をコードにインクルードして、_ILP32 または _LP64 の定義やその他の基本派生型を組み込んでください。
- 関数プロトタイプおよび非局所的な有効範囲を持つ外部宣言をヘッダーに移動し、それらのヘッダーをコードにインクルードしてください。

- -errchk=longptr64 を指定して lint(1) を実行し、各警告メッセージを確認してください (すべての警告どおりに変更が必要なわけではありません)。結果として必要になる変更によっては、32 ビットおよび 64 ビットモードで lint(1) を再実行する必要があります。
- アプリケーションを 64 ビット専用としてだけで提供する予定でない場合は、コードを 32 ビットおよび 64 ビットとしてコンパイルしてください。
- 32 ビットオペレーティングシステム上で 32 ビットバージョンのアプリケーションを実行し、さらに、64 ビットオペレーティングシステム上で 64 ビットバージョンを実行して、アプリケーションをテストしてください。64 ビットオペレーティングシステム上で 32 ビットバージョンのテストも実行できますが、その必要はありません。

プログラミング例

次に示すプログラミング例 foo.c では、LP64 と ILP32 のデータ型モデルの効果を直接的に示しています。同じプログラムを 32 ビットまたは 64 ビットプログラムとしてコンパイルすることができます。

```
#include <stdio.h>
int
main(int argc, char *argv[])
{
        (void) printf("char is \t\t%lu bytes\n", sizeof (char));
        (void) printf("short is \t%lu bytes\n", sizeof (short));
        (void) printf("int is \t\t%lu bytes\n", sizeof (int));
        (void) printf("long is \t\t%lu bytes\n", sizeof (long));
        (void) printf("long long is \t\t%lu bytes\n", sizeof (long long));
        (void) printf("pointer is \t%lu bytes\n", sizeof (void *));
        return (0);
}
```

32 ビットコンパイルの結果は、次のようになります。

```
% cc -O -o foo32 foo.c
% foo32
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          4 bytes
long long is     8 bytes
pointer is       4 bytes
```

64 ビットコンパイルの結果は、次のようになります。

```
% cc -xarch=generic64 -O -o foo64 foo.c
% foo64
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          8 bytes
long long is     8 bytes
pointer is       8 bytes
```

注 – デフォルトのコンパイル環境は、移植性を最大限にするように設計されているため、32 ビットのアプリケーションを作成します。

第 **5** 章

開発環境

この章では 64 ビットアプリケーション開発環境について説明します。構築環境、その他ヘッダーおよびライブラリの問題、コンパイラオプション、リンク、およびデバッグツールについて説明します。パッケージ処理に関するガイドラインも示します。

まず、オペレーティングシステムのバージョンが 32 ビットであるかあるいは 64 ビットであるかを確認する必要があります。使用しているバージョンが不明な場合は、64 ビットオペレーティングシステムを使用していると仮定します。バージョンを確認するには、第 3 章で説明した `isainfo(1)` コマンドを使用します。32 ビットオペレーティング環境を使用している場合でも、システムヘッダーファイルおよび 64 ビットライブラリがシステム上にあれば、64 ビットアプリケーションを構築することができます。

構築環境

構築環境には、システムヘッダー、コンパイルシステム、およびライブラリが含まれています。これらのことについては、次の節で説明します。

ヘッダーファイル

1 組のシステムヘッダーが 32 ビットおよび 64 ビットのコンパイル環境をサポートします。64 ビットコンパイル環境用に別のインクルードパスを指定する必要はありません。

64 ビット環境をサポートするためにヘッダーに加えられた変更をより理解するには、ヘッダーの `<sys/isa_defs.h>` のさまざまな定義について理解しておくことをお薦めします。このヘッダー中には `#define` が含まれており、それらは各命令セットアーキテクチャに対して設定されています。`<sys/types.h>` をインクルードすると、自動的に `<sys/isa_defs.h>` がインクルードされます。

次の表にあるシンボルは、コンパイル環境 (コンパイラ) によって定義されます。

シンボル	説明
__sparc	SPARC ファミリーのプロセッサアーキテクチャのいずれかであることを示します。アーキテクチャには、SPARC V7、SPARC V8、および SPARC V9 があります。シンボルの sparc は、非推奨の __sparc の同義語です。
__sparcv8	『The SPARC Architecture Manual, Version 8』で定義されている 32 ビット SPARC V8 アーキテクチャを示します。
__sparcv9	『The SPARC Architecture Manual, Version 9』で定義されている 64 ビット SPARC V9 アーキテクチャを示します。
__x86	x86 ファミリーのプロセッサアーキテクチャのいずれかであることを示します。これらのアーキテクチャには 386、486、Pentium、AMD64、および EM64T プロセッサがあります。
__i386	32 ビット i386 アーキテクチャであることを示します。
__amd64	64 ビット amd64 アーキテクチャであることを示します。

注 – __i386 および __amd64 は、相互に排他的です。シンボル __sparcv8 および __sparcv9 は相互に排他的で、シンボル __sparc が定義されているときにだけ意味があります。

次に示すシンボルは、上記のシンボルのいくつかの組み合わせから派生したものです。

シンボル	説明
_ILP32	int、long、およびポインタのサイズがすべて 32 ビットであるデータ型モデル
_LP64	long およびポインタのサイズがすべて 64 ビットであるデータ型モデル

注 – シンボル _ILP32 および _LP64 も、相互に排他的です。

完全に移植性のあるコードを書くことが不可能で、32 ビットおよび 64 ビットのそれぞれに固有のコードが必要な場合は、_ILP32 または _LP64 を使って、コードを条件付きで切り替えるようにしてください。これによって、マシンに依存しないコンパイル環境にすることができ、アプリケーションをすべての 64 ビットプラットフォームに移植する際の移植性が高くなります。

コンパイラの環境

Sun Studio C、C++、および Fortran のコンパイル環境が拡張され、32 ビットおよび 64 ビットアプリケーションの両方を作成できるようになりました。Sun Studio で提供される C コンパイラのリリース 10.0 は、64 ビットコンパイルをサポートします。

ネイティブコンパイルモードおよびクロスコンパイルモードがサポートされています。デフォルトのコンパイル環境は、これまでどおり 32 ビットアプリケーションを作成します。ただし、両モードともアーキテクチャに依存します。Sun コンパイラを使用して、x86 マシン上で SPARC オブジェクトを作成したり、 SPARC マシン上で x86 オブジェクトを作成することはできません。アーキテクチャやコンパイラのモードが指定されていない場合は、適宜 `__sparcv8` または `__i386` シンボルがデフォルトで定義され、この一部として `_ILP32` も定義されます。これによって、既存のアプリケーションおよびハードウェアにおける相互運用性が高くなります。

Sun Studio 8 リリースから、`cc`(1) `-xarch=generic64` フラグを使用して、64 ビットコンパイル環境を使用できるようにします。

これによって LP64 コードが含まれる ELF64 オブジェクトが生成されます。ELF64 とは、64 ビットのプロセッサおよびアーキテクチャをサポートする 64 ビットオブジェクトファイル形式のことです。一方、デフォルトの 32 ビットモードでコンパイルしたときは、ELF32 オブジェクトファイルが生成されます。

`-xarch=generic64` フラグを使用すると、32 ビットまたは 64 ビットのシステムで 64 ビット コードを生成できます。32 ビットコンパイラを使用すると、32 ビットシステムで 64 ビットオブジェクトを作成できますが、その 64 ビットオブジェクトは 32 ビットシステムでは実行できません。64 ビットライブラリに対するライブラリパスを指定する必要はありません。`-l` または `-L` オプションを使用してライブラリまたはライブラリパスを追加指定し、そのパスが 32 ビットライブラリだけを指している場合は、リンカーはそれを検出し、エラーを出力してリンク処理を異常終了します。

コンパイラオプションの説明は、『*Sun Studio 10: C ユーザーズガイド*』を参照してください。

32 ビット ライブラリと 64 ビット ライブラリ

Solaris オペレーティングシステムには、32 ビットおよび 64 ビットの両コンパイル環境用の共有ライブラリが含まれています。

32 ビットアプリケーションは 32 ビットライブラリとリンクし、64 ビットアプリケーションは 64 ビットライブラリとリンクしなければなりません。64 ビットライブラリを使って 32 ビットアプリケーションを作成または実行することはできません。32 ビットライブラリは、これまでどおり `/usr/lib` および `/usr/ccs/lib` に置かれています。64 ビットライブラリは、適切な `lib` ディレクトリのサブディレクトリに置かれています。32 ビットライブラリの場所は変更されていないので、Solaris 2.6 以前のオペレーティングシステムで構築された 32 ビットアプリケーションとのバイナリの互換性があります。移植可能な Make ファイルは、`64` シンボリックリンクを使用してライブラリのいずれかのディレクトリを参照します。

64 ビットアプリケーションを構築するためには 64 ビットライブラリが必要です。64 ビットライブラリは、32 ビットおよび 64 ビットの両環境で利用できるため、ネイティブコンパイルもクロスコンパイルも可能です。コンパイラとその他のツール (たとえば ld、ar、as など) は 32 ビットプログラムで、32 ビットまたは 64 ビットシステム上で 64 ビットプログラムを構築する機能があります。もちろん、32 ビットオペレーティング環境上のシステムで作成された 64 ビットプログラムは、32 ビット環境では実行できません。

オブジェクトファイルのリンク処理

リンカーは、32 ビットアプリケーションのままですが、ほとんどのユーザーはこのことを意識する必要がありません。通常リンカーはコンパイラドライバ (たとえば cc (1)) から間接的に呼び出されるからです。ELF32 オブジェクトファイルをリンカーへの入力として指定すると、リンカーは ELF32 出力ファイルを生成します。同様に、入力ファイルとして ELF64 オブジェクトファイルを指定すれば、ELF64 出力ファイルを生成します。ELF32 と ELF64 の入力ファイルを混在させて指定しようとしてもリンカーに拒絶されます。

LD_LIBRARY_PATH 環境変数

SPARC。32 ビットおよび 64 ビットのアプリケーション用の動的リンカープログラムは、それぞれ /usr/lib/ld.so.1 と /usr/lib/sparcv9/ld.so.1 です。

x86。AMD64 アーキテクチャでは、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用の動的リンカープログラムは、それぞれ /usr/lib/ld.so.1 と /usr/lib/amd64/ld.so.1 です。

これらの動的リンカーは両方とも、LD_LIBRARY_PATH 環境変数で指定された、コロンで区切られたディレクトリ名のリストを実行時に検索します。32 ビット動的リンカーは 32 ビットライブラリとだけ結合し、64 ビット動的リンカーは 64 ビットライブラリとだけ結合します。したがって、必要であれば 32 ビットおよび 64 ビットライブラリの両方を格納しているディレクトリを環境変数 LD_LIBRARY_PATH で指定することができます。

64 ビット動的リンカーの検索パスは、LD_LIBRARY_PATH_64 環境変数を使って完全に上書きする (優先させる) ことができます。

$ORIGIN キーワード

アプリケーションを配布し管理するための共通の手法として、関連するアプリケーションとライブラリを 1 つのディレクトリ階層に入れる手法があります。一般的に、アプリケーションが使用するライブラリは lib サブディレクトリに置き、アプリケーションそのものはベースディレクトリの bin サブディレクトリに置きます。このベー

スディレクトリは、Sun が配布するコンピューティングファイルシステムである NFS™ によりエクスポートし、クライアントマシンにマウントできます。環境によっては、オートマウンタとネームサービスを使用して、アプリケーションを配布し、すべてのクライアントにおいてアプリケーション階層のファイルシステムの名前空間を同じにすることもできます。そのような環境では、-R オプションをリンカーに指定してアプリケーションを構築できます。このオプションは、実行時に共有ライブラリを検索するディレクトリの絶対パス名を指定します。

ただし環境によっては、ファイルシステムの名前空間がうまく制御されず、開発者がデバッグ用のツール、つまり LD_LIBRARY_PATH 環境変数を使ってラッパースクリプトにライブラリの検索パスを指定するという手法を採用していました。これは不必要であり、$ORIGIN キーワードをリンカーの -R オプションに指定されたパス名に含めることができます。$ORIGIN キーワードは、実行可能プログラムそのものがあるディレクトリ名に実行時に展開されます。つまり、$ORIGIN からの相対パス名でライブラリディレクトリのパス名を指定できるということです。この結果、LD_LIBRARY_PATH をまったく設定しなくても、アプリケーションのベースディレクトリを移動することができるようになります。

この機能は 32 ビットおよび 64 ビットの両方のアプリケーションで利用できます。新しいアプリケーションを作成する場合に、LD_LIBRARY_PATH を正しく構成するユーザーやスクリプトに対する依存性を減らしたいときに、この機能を利用できます。

詳細は、『リンカーとライブラリ』を参照してください。

32 ビット アプリケーションと 64 ビット アプリケーションのパッケージ処理

以降の節では 32 ビットおよび 64 ビットアプリケーションのパッケージ処理について説明します。

ライブラリとプログラムの配置

SPARC。新しいライブラリとプログラムの配置については、49 ページの「32 ビット ライブラリと 64 ビット ライブラリ」に記載されている標準規則に従います。32 ビットライブラリは従来どおり同じ場所に置かれますが、64 ビットライブラリは、通常のデフォルトのディレクトリ内の、アーキテクチャに依存する特定のサブディレクトリに置くことをお薦めします。32 ビットおよび 64 ビットにそれぞれ固有のアプリケーションは、ユーザーにとって透過的な場所に置くようにしてください。

つまり、32 ビットライブラリは同じライブラリディレクトリに置き、64 ビットライブラリは適切な lib ディレクトリ下の sparcv9 サブディレクトリに置きます。

32 ビットまたは 64 ビット環境に固有のバージョンを必要とするプログラムは、上記とは多少異なり、通常置かれるディレクトリ下の sparcv7 または sparcv9 サブディレクトリに適宜置くことをお薦めします。

64 ビットライブラリは、適切な lib ディレクトリ下の amd64 サブディレクトリに置く必要があります。

32 ビット または 64 ビット 環境に固有のバージョンを必要とするプログラムは、通常置かれるディレクトリの下の i86 または amd64 サブディレクトリに適宜置くことをお薦めします。

52 ページの「アプリケーション命名規則」を参照してください。

パッケージ処理のガイドライン

パッケージ処理の選択肢として、32 ビットおよび 64 ビットアプリケーション用にそれぞれパッケージを別々に作成するか、あるいは 32 ビットおよび 64 ビットバージョンを 1 つのパッケージにまとめるか、という問題があります。1 つのパッケージを作成する場合は、この章で説明しているサブディレクトリの命名規則をパッケージの内容に対して適用することをお薦めします。

アプリケーション命名規則

32 ビットおよび 64 ビットのアプリケーションに対して foo32 や foo64 のような特定の名前を付ける代わりに、51 ページの「ライブラリとプログラムの配置」で説明したように、32 ビットおよび 64 ビットアプリケーションをプラットフォーム固有の適切なサブディレクトリに置くことができます。このようにすると、次の項で説明しているラッパーを使用して、環境に応じて 32 ビットまたは 64 ビットのいずれかのアプリケーションを実行することができます。利点としては、プラットフォームに応じて適切なバージョンのアプリケーションが自動的に実行されるため、ユーザーは 32 ビットまたは 64 ビットのどちらであるかなどについて意識する必要がありません。

シェルスクリプトラッパー

32 ビットおよび 64 ビット固有のバージョンのアプリケーションが必要な場合、シェルスクリプトラッパーを使うと、ユーザーがバージョンに関して意識する必要がなくなります。32 ビットおよび 64 ビットバージョンが必要な、Solaris オペレーティングシステムの多くのツールが、この例として当てはまります。ラッパーを利用すると、特定のハードウェアプラットフォーム上で実行可能な固有の命令セットを isalist コマンドを使って調べ、それに基づいて適切なバージョンのツールを実行させることができます。

次に、ネイティブ命令セットラッパーの例を示します。

```
#! /bin/sh

CMD=`basename $0`
DIR=`dirname $0`
EXEC=
for isa in `/usr/bin/isalist`; do
    if [-x ${DIR}/${isa}/${CMD}]; then
        EXEC=${DIR}/${isa}/${CMD}
        break
    fi
done
if [-z "${EXEC}"]; then
        echo 1>&2 "$0: no executable for this architecture"
        exit 1

fi
exec ${EXEC} "${@}"
```

この例には問題が 1 つあります。$0 引数が、その引数自身の実行可能プログラムに対する完全パス名であることを前提にしていることです。このような理由から、汎用的なラッパーである isaexec() が作成され、32 ビットおよび 64 ビット固有のアプリケーションの問題に対処しています。引き続きこのラッパーについて説明します。

/usr/lib/isaexec バイナリファイル

isaexec(3C) は 32 ビット実行可能バイナリファイルです。1 つ前の節で説明したようなシェルスクリプトラッパー機能を実行しますが、その際引数リストも正確に保存します。実行可能プログラムの完全パス名は /usr/lib/isaexec ですが、この名前で実行するようには設計されていません。このプログラムでは、isalist(1) によって選択された複数のバージョンで存在するプログラムの主要な名前 (プログラム実行時にユーザーが使用する名前) にコピーされたり、リンク (シンボリックリンクではなくハードリンク) される可能性があります。

たとえば、SPARC 環境では、truss(1) コマンドは次の 3 つの実行可能ファイルとして存在します。

```
/usr/bin/truss
/usr/bin/sparcv7/truss
/usr/bin/sparcv9/truss
```

sparcv7 と sparcv9 サブディレクトリの実行可能プログラムは、実在する truss(1) 実行可能プログラムで、それぞれ 32 ビットおよび 64 ビットプログラムです。ラッパーファイルの /usr/bin/truss は、/usr/lib/isaexec にハードリンクされています。

x86 環境では、truss(1) コマンドは次の 3 つの実行可能ファイルとして存在します。

```
/usr/bin/truss
/usr/bin/i86/truss
/usr/bin/amd64/truss
```

isaexec(3C) ラッパーは、完全に解決された、シンボリックリンクのない自分のパス名を、argv[0] 引数とは別に getexecname(3C) を使用して調べ、sysinfo (SI_ISALIST, ...) を使用して isalist(1) を取得します。そして、その結果得られた自分自身のディレクトリのサブディレクトリリストを調べ、自分の名前がある最初の実行可能ファイルに対して exec(2) を実行します。そのとき isaexec(3C) ラッパーは、引数ベクトルと環境ベクトルを変更せずに渡します。このようにして、argv[0] は最初に指定されたとおりに最終的なプログラムイメージに渡されます。サブディレクトリ名を含むように修正された完全パス名に変換された形ではありません。

注 – ラッパーが存在する場合があるため、実行可能プログラムを他の場所に移動する際は注意が必要です。実際のプログラムではなく、ラッパーを移動してしまう可能性があります。

---

isaexec(3c) インタフェース

多くのアプリケーションでは、すでに起動ラッパープログラムを使用して、環境変数の設定、一時ファイルの消去、デーモンの起動などを行なっています。libc(3LIB) 内の isaexec(3C) インタフェースを利用すると、前述のようなシェルスクリプトラッパーの例で使用しているのと同じアルゴリズムを、カスタマイズしたラッパープログラムから直接呼び出すことができます。

64 ビットアプリケーションのデバッグ処理

Solaris 上で実行できる truss(1) コマンド、/proc ツール (proc(1))、および mdb などのすべてのデバッグツールが、64 ビットアプリケーションで動作するようにアップグレードされています。

これらのデバッグツールのうち、64 ビットアプリケーションをデバッグできる dbx デバッガは、Sun Studio ツール群の一部として入手できます。それ以外のツールはすべて Solaris リリースの中に含まれています。

これらのすべてのデバッグツールのオプションには変更がありません。64 ビットプログラムをデバッグするために、mdb では多数の拡張機能を使用できます。ポインタを間接参照するために「*」を使用すると、64 ビットプログラムに対しては 8 バイトを、32 ビットプログラムに対しては 4 バイトを参照します。さらに、次の修飾子を使用できます。

追加修飾子 ?、/、= ：

```
g        (8) 符号なし 8 進数で 8 バイト表示
G        (8) 符号付き 8 進数で 8 バイト表示
e        (8) 符号付き 10 進数で 8 バイト表示
E        (8) 符号なし 10 進数で 8 バイト表示
J        (8) 16 進数で 8 バイト表示
K        (n) pointer または long を 16 進数で印刷
          32 ビットプログラムを 4 バイト表示、
          64 ビットプログラムを 8 ビット表示する。
y        (8) 日付形式で 8 バイト印刷
```

追加修飾子 ?、/ ：

```
M <value> <mask>  8 バイトの値に <mask> を適用し比較する。
         「.」をマッチする位置へ移動する。
Z        (8) 8 バイト書き込み
```

第 **6** 章

上級者向けトピック

この章では、64 ビット Solaris オペレーティングシステムについてさらに詳しく知りたいシステムプログラマ向けに、プログラミングに関するさまざまな情報を提供します。

64 ビット環境のほとんどの新機能は、一般の 32 ビットインタフェースを拡張したものですが、一部の新機能は 64 ビット環境に固有の機能です。

SPARC V9 ABI の特徴

64 ビットアプリケーションは、ELF64 実行可能およびリンク形式 (Executable and Linking Format) によって作成されます。この形式によって、大規模なアプリケーションおよびアドレス空間を完全に記述することができます。

SPARCV9。『*SPARC Compliance Definition*, Version 2.4』には、SPARC V9 ABI の詳細が含まれます。このマニュアルでは 32 ビットの SPARC V8 ABI と 64 ビット SPARC V9 ABI について説明しています。このマニュアルは、SPARC International の www.sparc.com から入手できます。

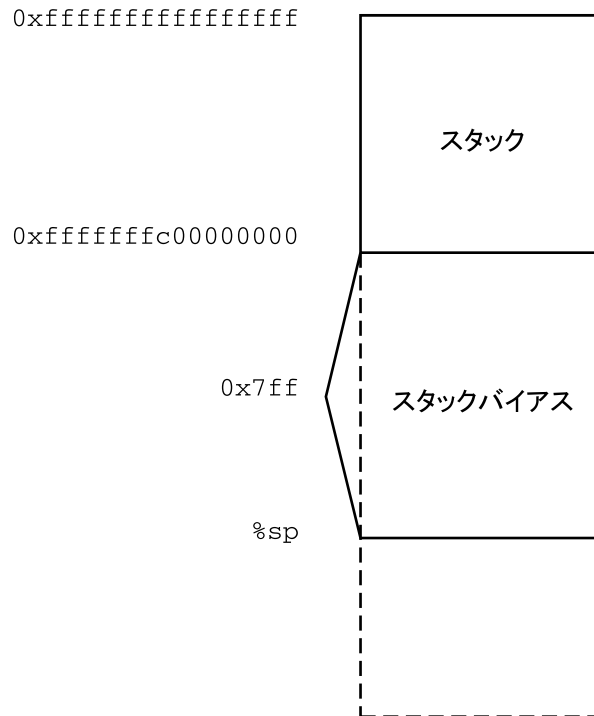
次に SPARC V9 ABI の特徴を示します。

- すべての 64 ビット SPARC 命令と 64 ビット幅のレジスタを最大限有効に活用できます。関連した新しい命令の多くは、既存の V8 命令セットの拡張版です。『*The SPARC Architecture Manual, Version 9*』を参照してください。
- 基本的な呼び出し規約は同じです。呼び出し側の最初の 6 つの引数は、出力レジスタの `%o0`-`%o5` に格納されます。SPARC V9 ABI では、関数呼び出しの動作を「軽く」するために、従来より大きいレジスタファイル上で、従来どおりレジスタウィンドウを使用しています。結果は `%o0` に格納されます。すべてのレジスタは 64 ビット量として扱われるので、64 ビットの値は、一組のレジスタにではなく 1 つのレジスタに渡されます。

- スタックの配置が変わりました。基本セルサイズが 32 ビットから 64 ビットに拡大されました。さまざまな隠れたパラメータ語が削除されました。戻りアドレスは `%o7 + 8` のままです。
- `%o6` は従来どおりスタックポインタレジスタ `%sp` として参照され、`%i6` はフレームポインタレジスタ `%fp` として参照されます。ただし、`%sp` レジスタと `%fp` レジスタは、スタックバイアスと呼ばれる定数だけ、スタックの実際のメモリー位置からオフセットされます。スタックバイアスのサイズは 2047 バイトです。
- 命令長は従来どおり 32 ビットです。したがって、アドレス定数を生成するには通常以上の命令が必要となります。CALL 命令は、アドレス空間内への分岐には使用できなくなりました。CALL 命令は、`%pc` から + 2G バイトまたは - 2G バイト以内までしか到達できないからです。
- 整数乗算機能および除算機能は、現在完全にハードウェアで実装されています。
- データ構造体を渡す方法と戻す方法は異なります。小さいデータ構造体と浮動小数点引数のいくつかは、現在はレジスタに直接渡されます。
- ユーザートラップ機能により、ユーザートラップハンドラが (シグナルを発信する代わりに) 非特権コードからのトラップのいくつかを取り扱うことができるようになりました。
- すべてのデータ型はそれぞれのサイズに境界整列されるようになりました。
- 基本派生型の多くは、従来よりサイズが大きくなりました。したがって、多くのシステムコールインタフェースのデータ構造体のサイズも変わっています。
- 2 つの異なるライブラリセット (32 ビット SPARC アプリケーション用のライブラリと 64 ビット SPARC アプリケーション用のライブラリ) が、システムに存在します。

スタックバイアス

SPARC V9。開発者にとって重要な SPARC V9 ABI の特徴の 1 つに、スタックバイアスがあります。64 ビット の SPARC プログラムでは、2047 バイトのスタックバイアスを、フレームポインタとスタックポインタの両方に追加して、スタックフレームの実際のデータを取得する必要があります。次の図を参照してください。

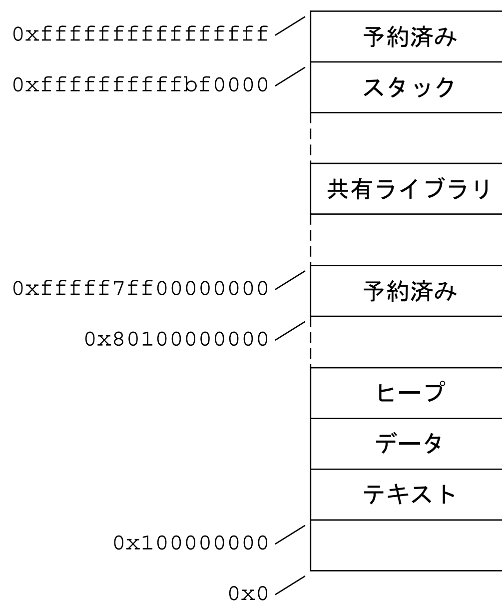


スタックバイアスについては、SPARC V9 ABI を参照してください。

SPARC V9 ABI のアドレス空間の配置

SPARC V9。64 ビットアプリケーションのアドレス空間の配置は、32 ビットアプリケーションのアドレス空間の配置に密接に関係しています。ただし、開始アドレスとアドレス指定の制限値は大きく変更されています。SPARC V8 と同様に、SPARC V9 のスタックはアドレス空間の上端から下方に広がり、ヒープは下端から上方にデータセグメントを拡張します。

以下の図は、64 ビットアプリケーションに与えられたデフォルトのアドレス空間を示します。「予約済み」となっているアドレス空間の領域は、アプリケーションからマップすることはできません。これらの制約は、将来のシステムで緩和される可能性があります。



上図の実際のアドレスは、ある特定のマシンの特定の実装を示しており、説明のためにだけ掲載してあります。

SPARC V9 ABI のテキストおよびデータの配置

デフォルトでは、64 ビット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、および共有ライブラリを含めて 4G バイトを超えるアドレスに存在します。これは、64 ビットプログラムが正しいことを検証するのに役立ちます。たとえばプログラムが関連するポインタの上位 32 ビットを切り落としてしまうと、そのプログラムはアドレス空間の下方の 4G バイトの部分へアクセスしようとして失敗します。

64 ビットプログラムは 4G バイトを超える位置でリンクされますが、リンカーのマップファイルを使用し、コンパイラまたはリンカーに `-M` オプションを指定して、4G バイト未満の位置でリンクすることも可能です。4G バイト未満で 64 ビット SPARC プログラムをリンクするためのリンカーマップファイルは、`/usr/lib/ld/sparcv9/map.below4G` にあります。

詳細は、`ld(1)` のリンカーのマニュアルページを参照してください。

SPARC V9 ABI のコードモデル

SPARC V9。コンパイラには、性能の向上や、64 ビット SPARC プログラムでのコードサイズを小さくするなど、さまざまな目的に合わせた各種のコードモデルがあります。コードモデルは以下の要素で決定します。

- 位置決め方法 (絶対コード、あるいは位置に依存しないコード)
- コードサイズ (2G バイト未満)
- 位置 (下部、中央、アドレス空間内の任意位置)
- 外部オブジェクト参照モデル (スモールまたはラージ)

次の表は、64 ビット SPARC プログラムで使用できる各種コードモデルを示したものです。

表 **6–1** コードモデルの説明 SPARC V9

コードモデル	位置決め方法	コードサイズ	位置	外部オブジェクト参照モデル
`abs32`	絶対	2G バイト未満	下部 (アドレス空間の下位 32 ビット)	なし
`abs44`	絶対	2G バイト未満	中央 (アドレス空間の下位 44 ビット)	なし
`abs64`	絶対	2G バイト未満	任意	なし
`pic`	位置に依存しないコード	2G バイト未満	任意	スモール (1024 以下の外部オブジェクト)
`PIC`	位置に依存しないコード	2G バイト未満	任意	ラージ (2**29 以下の外部オブジェクト)

スモールコードモデルを使用すると、命令シーケンスを短くできる場合があります。絶対コード内で静的データ参照を行うのに必要な命令の数は、`abs32` コードモデルの場合が最も少なく、`abs64` が最も多く、`abs44` がその中間になります。同様に、`pic` コードモデルは、`PIC` コードモデルよりも少ない命令で静的データ参照を行います。その結果、コードモデルが小さいほどコードサイズも小さくなり、ラージコードモデルのような、より完全な機能性を必要としないプログラムの性能が向上します。

使用するコードモデルを指定するには、`-xcode=<model>` コンパイラオプションを使用する必要があります。現在、コンパイラは 64 ビットオブジェクトに対し、デフォルトで `abs64` モデルを使用します。コードは、`abs44` コードモデルの使用により最適化できます。より少ない命令を使用して、現在の UltraSPARC プラットフォームがサポートする 44 ビットのアドレス空間を利用できます。

コードモデルについては、SPARC V9 ABI およびコンパイラのマニュアルを参照してください。

注 – abs32 コードモデルでコンパイルしたプログラムは、-M /usr/lib/ld/sparcv9/map.below4G オプションを使用して、4G バイトよりも下方にリンクする必要があります。

AMD64 ABI の特徴

64 ビットアプリケーションは、ELF64 実行可能およびリンク形式 (Executable and Linking Format) によって作成されます。この形式によって、大規模なアプリケーションおよびアドレス空間を完全に記述することができます。

次に AMD ABI の特徴を示します。

- すべての 64 ビット命令と 64 ビットレジスタを最大限有効に活用できます。新しい命令の多くは、既存の i386 命令セットの単純な拡張版です。汎用レジスタは 16 あります。

  7 つの汎用レジスタ (%rdi、%rsi、%rdx、%rcx、%r8、%r9、%rax) は、引数をレジスタに渡す関数呼び出しシーケンスにおける適切な役割を定義されています。
  2 つのレジスタ (%rsp、%rbp) はスタック管理のために使用されます。
  2 つのレジスタ (%r10、%r11) は一時的なものです。
  5 つのレジスタ (%r12、%r13、%r14、%r15、 %rbx) は呼び出し先保管です。
- 基本的な関数呼び出し規約は、AMD ABI では異なります。引数はレジスタに格納されます。単純な整数の引数の場合、最初の引数から順に %rdi、%rsi、%rdx、%rcx、%r8、%r9 レジスタに格納されます。
- スタックの配置は、AMD では多少異なります。具体的には、スタックは常に、CALL 命令の直前で 16 バイトの境界に整列されます。
- 命令長は従来どおり 32 ビットです。したがって、アドレス定数を生成するには通常以上の命令が必要となります。CALL 命令は、アドレス空間内への分岐には使用できなくなりました。CALL 命令は、%rip から + 2G バイトまたは - 2G バイト以内までしか到達できないからです。
- 整数乗算機能および除算機能は、現在完全にハードウェアで実装されています。
- データ構造体を渡す方法と戻す方法は異なります。小さいデータ構造体と浮動小数点引数のいくつかは、現在はレジスタに直接渡されます。
- より効率的な位置に依存しないコードを生成できる、PC 相対の新しいアドレス指定モードがあります。
- すべてのデータ型はそれぞれのサイズに境界整列されるようになりました。
- 基本派生型の多くは、従来よりサイズが大きくなりました。したがって、多くのシステムコールインタフェースのデータ構造体のサイズも変わっています。
- 2 つの異なるライブラリセット (32 ビット i386 アプリケーション用のライブラリと 64 ビット amd64 アプリケーション用のライブラリ) が、システムに存在します。

- 浮動小数点の機能が大幅に拡張されています。

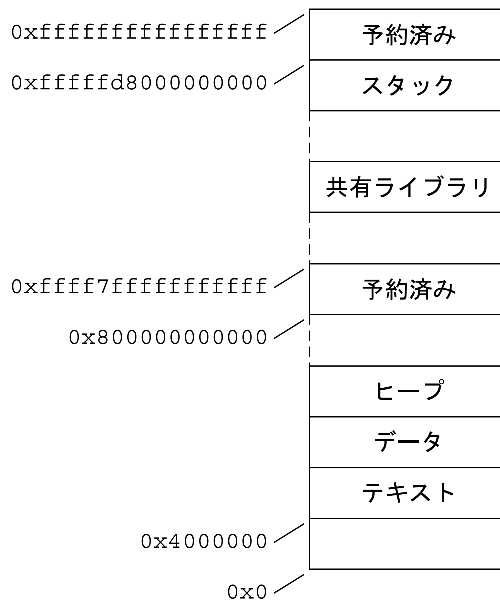
  64 ビット ABI では、x87 浮動小数点レジスタ (`%fpr0` 〜 `%fpr7` および `%mm0` 〜 `%mm7`) で動作する x87 と MMX のすべての命令を使用できます。
  さらに、128 ビット XMM レジスタ (`%xmm0` 〜 `%xmm15`) で動作する SSE と SSE2 命令のフルセットも使用できます。

amd64 psABI 草案文書『*System V Application Binary Interface, AMD64 Architecture Processor Supplement*』(草案バージョン 0.92、2004 年 9 月 9 日) を参照してください。

amd64 アプリケーションのアドレス空間の配置

64 ビットアプリケーションのアドレス空間の配置は、32 ビットアプリケーションのアドレス空間の配置に密接に関係しています。ただし、開始アドレスとアドレス指定の制限値は大きく変更されています。SPARC V9 と同様に、amd64 のスタックはアドレス空間の上端から下方に広がり、ヒープは下端から上方にデータセグメントを拡張します。

以下の図は、64 ビットアプリケーションに与えられたデフォルトのアドレス空間を示します。「予約済み」となっているアドレス空間の領域は、アプリケーションからマップすることはできません。これらの制約は、将来のシステムで緩和される可能性があります。



上図の実際のアドレスは、ある特定のマシンの特定の実装を示しており、説明のためにだけ掲載してあります。

整列の問題

データ構造体の 32 ビット long long 要素の整列に関連してもう一つの問題があります。すなわち、i386 アプリケーションは 32 ビット境界で long long 要素を整列するだけですが、amd64 ABI は long long 要素を 64 ビット境界に配置するので、データ構造体に大きな隙間が生じる可能性があります。SPARC は 32 ビットと 64 ビットの両方で long long 項目が 64 ビット境界に整列されており、この点が異なります。

次の表は、設計アーキテクチャに対するデータ型整列を示します。

表 **6–2** データ型整列

アーキテクチャ	**long long**	**double**	**long double**
i386	4	4	4
amd64	8	8	16
sparcv8	8	8	8
sparcv9	8	8	16

SPARC システムで LP64 に対し問題がないと思われるコードであっても、整列の違いのために、32 ビットと 64 ビットのプログラミング環境間でデータ構造体をコピーする際に問題が生じる可能性があります。そのようなプログラミング環境には、デバイスドライバ ioctl ルーチン、doors ルーチンや、その他の IPC メカニズムがあります。整列の問題は、これらのインタフェースを慎重にコーディングしたり、#pragma pack または _Pack 指示語を適切に使用することによって回避できます。

プロセス間通信

次に示すプロセス間通信 (IPC) プリミティブは、従来どおり 64 ビットプロセスと 32 ビットプロセスとの間で動作します。

- System V IPC プリミティブ (shmop(2)、semop(2)、 msgsnd(2) など)
- 共有ファイル上の mmap(2) への呼び出し
- プロセス間の pipe(2) の使用
- プロセス間の door_call(3DOOR) の使用
- xdr(3NSL) に説明されている外部データ表現を使用した、同一マシン上あるいは異なるマシン上でのプロセス間の rpc(3NSL) の使用

これらのすべてのプリミティブは、32 ビットプロセスと 64 ビットプロセスとの間の通信を可能にしますが、プロセス間で交換されているデータがそれらすべてのプロセスで正しく解釈されることを、明確な手順によって確認する必要がある場合がありま

す。たとえば、long 型の変数を含む C データ構造体で記述されるデータを 2 つのプロセスが実際に共有するには、32 ビットプロセスがこの変数を 4 バイト量とみなし、64 ビットプロセスはこの変数を 8 バイト量とみなすということを認識する必要があります。

この相違を取り扱う 1 つの方法は、両プロセス間で意味をなすようにデータが完全に同じサイズであることを保障することです。int32_t や int64_t のような固定幅型を使ってデータ構造を構成してください。整列に関しても配慮が必要です。共有されるデータ構造体は、パディングを追加したり、#pragma pack、_Pack などのコンパイラ指示語を使用して再パックする必要のある場合があります。64 ページの「整列の問題」を参照してください。

システムで提供される派生型に対応する派生型の一群が <sys/types32.h> にあります。これらの派生型は、32 ビットシステムの基本型と同じ符号、同じサイズですが、ILP32 および LP64 のコンパイル環境でサイズが変わらないように定義されています。

32 ビットプロセスと 64 ビットプロセスとの間でポインタを共有するのは、さらに困難です。まずポインタのサイズが異なるということがあります。またそれ以上に重要なことは、既存の C の使用法に 64 ビット整数 (long long) はありますが、64 ビットポインタには 32 ビット環境に相当するものはない、ということです。64 ビットプロセスが 32 ビットプロセスとデータを共有できるようにするため、32 ビットプロセスは共有データのうち、4G バイトまでしか一度に「見る」ことはできません。

XDR ルーチンの xdr_long(3NSL) は問題と思われるかもしれません。しかし、これは既存のプロトコルとの互換性を持たせるために従来どおり 32 ビットとして取り扱われます。64 ビットバージョンのルーチンが 32 ビットに格納できない long 値をコード化するように要求された場合、そのコード化処理は失敗します。

ELF とシステム生成ツール

64 ビットバイナリは、ELF64 形式でファイルに格納されます。この ELF64 形式は、ほとんどのフィールドが完全 64 ビットアプリケーションを格納するために拡張されていることを除いて、ELF32 形式に類似しています。ELF64 ファイルは elf(3ELF) API、たとえば elf_getarhd(3ELF) を使って読むことができます。

ELF ライブラリ elf(3ELF) の 32 ビットおよび 64 ビットのバージョンは、それぞれ ELF32 および ELF64 形式と、対応する API をサポートします。これによりアプリケーションは、32 ビットシステムまたは 64 ビットシステム (64 ビットプログラムを実行するには 64 ビットシステムが必要) から、両ファイル形式を構築、読み込み、あるいは修正ができるようになります。

さらに、Solaris では GELF (Generic ELF) インタフェースを提供し、プログラマが 1 つの共通 API を使用して両方の ELF 形式を操作できるようにしています。詳細は、elf(3ELF) のマニュアルページを参照してください。

ar(1)、nm(1)、ld(1)、および dump(1) を含む、すべてのシステム ELF ユーティリティが両方の ELF 形式を使用できるように変更されています。

/proc インタフェース

/proc インタフェースは、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の両方で利用できます。32 ビットアプリケーションは、他の 32 ビットアプリケーションの状態を調べたり制御したりできます。したがって、既存の 32 ビットデバッガを 32 ビットアプリケーションのデバッグに使用できます。

64 ビットアプリケーションは、他の 32 ビットまたは 64 ビットアプリケーションの状態を調べたり制御したりできます。ただし 32 ビットアプリケーションでは、64 ビットアプリケーションを制御できません。これは、32 ビット API では 64 ビットプロセスの完全な状態を記述することができないからです。このため、64 ビットアプリケーションをデバッグするには、64 ビットのデバッガが必要となります。

sysinfo(2) の拡張

Solaris S10 オペレーティングシステムの新しい sysinfo(2) サブコードによって、アプリケーションは使用可能な命令セットアーキテクチャに関する詳細な情報を確認できます。

```
SI_ARCHITECTURE_32
SI_ARCHITECTURE_64
SI_ARCHITECTURE_K
SI_ARCHITECTURE_NATIVE
```

たとえば、SI_ARCHITECTURE_64 サブコードを使用することによって、システム上に 64 ビット ABI があればその名前がわかります。詳細は、sysinfo(2) を参照してください。

libkvm と /dev/ksyms

64 ビットの Solaris システムは、64 ビットカーネルを使って実装されています。カーネルの内容を直接調べたり変更するアプリケーションは、64 ビットアプリケーションに変換し、64 ビットバージョンのライブラリとリンクしなければなりません。

このような変換と修正を行う前に、まずアプリケーションがカーネルのデータ構造を直接知る必要があるかどうかを検討した方がよいでしょう。プログラムが最初に移植されるかあるいは新規に作成された後に、システムコールを使って必要なデータを抽出するインタフェースが Solaris プラットフォームで利用可能になって追加された、という可能性があります。この場合は、最も一般的な代替 API として sysinfo(2)、kstat(3KSTAT)、sysconf(3C)、proc(4) を参照してください。これらのインタフェースが kvm_open(3KVM) の代わりに使用できるのなら、移植性を最大限にするために、それらを使用してアプリケーションを 32 ビットのままにしてください。さらに利点として、これらの API のほとんどは処理が速く、カーネルメモリーにアクセスするときと同じセキュリティ特権を必要としないことがあります。

32 ビットバージョンの libkvm は、64 ビットのカーネルクラッシュダンプに対して kvm_open(3KVM) を使用しようとしたときに異常終了します。同様に、64 ビットバージョンの libkvm は、32 ビットのカーネルクラッシュダンプに対して kvm_open (3KVM) を使用しようとしたときに異常終了します。

カーネルは 64 ビットプログラムなので、カーネルのシンボルテーブルを直接調べるために /dev/ksyms を開くアプリケーションは、ELF64 形式を理解するように機能を拡張する必要があります。

kvm_read() または kvm_write() へのアドレス引数がカーネルアドレスであるかユーザーアドレスであるかが曖昧であることは、64 ビットアプリケーションおよびカーネルではさらに問題となります。現在でもまだ kvm_read() と kvm_write() を使用している libkvm を利用するアプリケーションはすべて、kvm_read()、kvm_write()、kvm_uread()、kvm_uwrite() のルーチンを使用するようにする必要があります。これらのルーチンは、Solaris 2.5 から利用できるようになっています。

/dev/kmem または /dev/mem を直接読むアプリケーションは、従来どおり実行できます。ただし、これらのデバイスから読み込んだデータを解釈しようとすると、問題が発生することがあります。これは、データ構造体のオフセットおよびサイズが 32 ビットと 64 ビットのカーネル間で確実に異なるためです。

libkstat カーネル統計情報

多くのカーネル統計情報のサイズは、カーネルが 64 ビットあるいは 32 ビットプログラムのどちらであるかということとは関係ありません。名前付き kstat (kstat(3KSTAT) のマニュアルページを参照) がエクスポートするデータ型は自己記述型で、符号付きまたは符号なしの 32 ビットまたは 64 ビットカウンタデータを、適切なタグを付けてエクスポートします。したがって、libkstat を使用するアプリケーションは、64 ビットカーネル上で正常に動作させるために 64 ビットアプリケーションに変換する必要はありません。

注 – 名前付き kstats を作成および管理するデバイスドライバを修正するときは、エクスポートしようとする統計情報のサイズを、固定幅の統計データ型を使って 32 ビットおよび 64 ビットカーネル間で不変にすることをお薦めします。

---

stdio への変更

64 ビット環境では stdio 機能が拡張されて、256 を超える数のストリームを同時に開くことができるようになりました。32 ビットの stdio 機能では、従来どおり 256 を超える数のストリームを同時に開くことはできないという制限があります。

64 ビットアプリケーションが、FILE データ構造体のメンバーにアクセスできることに依存しないようにしてください。実装に固有な構造体メンバーに直接アクセスしようとすると、コンパイルエラーとなります。この変更で既存の 32 ビットアプリケーションが影響を受けることはありませんが、このように構造体のメンバーを直接使用する方法は、すべてのコードから取り除くべきです。

FILE 構造体には長い歴史があり、この構造体の中身を参照してストリームの状態に関する付加的な情報を収集するアプリケーションもあります。64 ビットのこの構造体は参照できないようになっているため、新しいルーチン群が 32 ビットの libc と 64 ビットの libc に加えられ、その結果、実装の内部に依存することなく同じ状態を調べることができるようになりました。たとえば __fbufsize(3C) のマニュアルページを参照してください。

パフォーマンスの問題

64 ビットのパフォーマンスの長所および短所について説明します。

64 ビットアプリケーションの長所

- 64 ビットに対する算術演算および論理演算がより効率的である
- 演算に、全レジスタ幅、全レジスタセット、および新しい命令が使用される
- 64 ビット量のパラメータ渡しがより効率的である
- 小さなデータ構造体および浮動小数点のパラメータ渡しがより効率的である
- 整数レジスタおよび浮動小数点レジスタが追加されている
- amd64 の場合、より効率的な位置に依存しないコードのための、PC 相対のアドレス指定モードがある

64 ビットアプリケーションの短所

- より大きいレジスタを格納するためにより大きなスタック空間を必要とする
- より大きなポインタによってより大きなキャッシュサイズを使用する
- 32 ビットのプラットフォームでは実行できない

システムコールの問題

システムコールの問題について説明します。

EOVERFLOW の意味

戻り値の EOVERFLOW は、カーネルからの情報を渡すために使うデータ構造体の 1 つまたは複数のフィールドが小さすぎて値を格納できない場合に、常にシステムコールから返されます。

現在、64 ビットカーネル上の大きなオブジェクトに遭遇したとき、多くの 32 ビットシステムコールは EOVERFLOW を返します。これまでも、大規模ファイルを扱う場合には同様でしたが、daddr_t、dev_t、time_t、およびその派生型の struct timeval と timespec_t が現在では 64 ビットを格納するため、32 ビットアプリケーションにおいては、従来よりも EOVERFLOW が返される場合が増えます。

ioctl() に関する注意

一部の ioctl(2) 呼び出しは、これまでうまく指定されていませんでした。ioctl() はコンパイル時の型検査では検出されません。そのため、ioctl() は追跡が困難なバグの原因になる可能性があります。

2 つの ioctl() 呼び出しを考えてみてください。一方は 32 ビット量（IOP32）へのポインタを操作し、もう一方は long（IOPLONG）へのポインタを操作します。

次のコード例は、32 ビットアプリケーションの一部として動作します。

```
int a, d;
long b;
...
if (ioctl(d, IOP32, &b) == -1)
        return (errno);
if (ioctl(d, IOPLONG, &a) == -1)
        return (errno);
```

このコードが 32 ビットアプリケーションの一部としてコンパイルされ、実行されるとき、どちらの ioctl(2) 呼び出しも正しく動作します。

このコードが 64 ビットアプリケーションとしてコンパイルされ、実行されるとき、どちらの ioctl() 呼び出しも正常終了しますが、正しく動作しません。最初の ioctl() は、大きすぎるコンテナ (データを格納するメモリー領域) を渡します。その結果、ビッグエンディアン実装の場合は、カーネルは 64 ビットワードの誤った部分へ、あるいは誤った部分からコピーしようとします。リトルエンディアン実装の場合でも、コンテナには上位 32 ビットに意味のない値が含まれます。2 番目の ioctl() は、コピー量が多すぎるため、正しくない値を読み込むか、あるいはユーザースタック内の隣接する変数を破壊してしまいます。

付録 A

派生型の変更

デフォルトの 32 ビットコンパイル環境は、派生型およびサイズに関して従来の Solaris オペレーティングシステムリリースと同じです。64 ビットコンパイル環境では、派生型をいくつか変更する必要があります。これらの派生型について、次に示す表で説明します。

コンパイル環境は 32 ビットと 64 ビットとで相違がありますが、両方の環境で同じヘッダー群が使用され、それぞれをコンパイルオプションで適切に定義することができます。アプリケーション開発者がどの選択肢を利用できるかをよりよく理解するには、`_ILP32` および `_LP64` という機能テストマクロを理解することが役に立ちます。

表 **A–1** 機能テストマクロ

機能テストマクロ	説明
`_ILP32`	`_ILP32` 機能テストマクロは、`int`、`long`、およびポインタが 32 ビット量である ILP32 データ型モデルを指定するのに使用します。このマクロを使用すれば、自動的に従来の Solaris の実装と同じ派生型およびサイズが見えるようになります。これは、32 ビットアプリケーションを構築するときのデフォルトのコンパイル環境です。これによって C および C++ アプリケーションに対するバイナリおよびソースの互換性が保証されます。
`_LP64`	`_LP64` 機能テストマクロは、`int` は 32 ビット量で `long` とポインタは 64 ビット量である _LP64 データ型モデルを指定するのに使用します。_LP64 は、64 ビットモードでコンパイルするときにデフォルトで定義されます。`<sys/types.h>` または `<sys/feature_tests.h>` がソースにインクルードされて `_LP64` 定義が見えるようになっていることを確認する以外、開発者は何もする必要がありません。

次のコード例で、コンパイル環境に応じた正しい定義が見えるようになる機能テストマクロを使用する例を示します。

例 **A–1** `_LP64` に定義された `size_t`

```
#if defined(_LP64)
typedef ulong_t size_t;          /* サイズ (バイト数) */
#else
```

例 **A–1** `_LP64` に定義された `size_t`　（続き）

```
typedef uint_t size_t;          /* （従来のバージョン） */
#endif
```

この例にある定義で 64 ビットアプリケーションを構築する場合、`size_t` は `ulong_t` または `unsigned long` となり、これは LP64 データ型モデルでは 64 ビット量です。32 ビットアプリケーションを構築する場合、`size_t` は `uint_t` または `unsigned int` として定義されます。これは、ILP32 および LP64 のどちらのデータ型モデルでも 32 ビット量です。

例 **A–2** `_LP64` に定義された `uid_t`

```
#if defined(_LP64)
typedef int     uid_t;          /* UID 型 */
#else
typedef long    uid_t;          /* （従来のバージョン） */
#endif
```

ILP32 データ型モデルの表現が LP64 データ型モデルの表現と同じであったとしても、これらの例のどちらの場合でも同じ最終結果が得られたはずです。たとえば、`size_t` を `ulong_t` に、`uid_t` を`int` に変換したとしても、32 ビットアプリケーション環境ではどちらも 32 ビット量であったはずです。しかし、従来の型表現を維持することによって、32 ビットの C および C++ アプリケーションどうしの整合性と、Solaris の以前のバージョンとのバイナリおよびソースの互換性が保証されています。

表 A–2 に、変更された派生型を示します。`_ILP32` 機能テストマクロの欄の型は、Solaris ソフトウェアに 64 ビットサポートが追加される前の Solaris 2.6 の型と同じであることに注意してください。32 ビットアプリケーションを作成する場合に利用できる派生型は `_ILP32` 欄の型です。64 ビットアプリケーションを作成する場合に利用できる派生型は `_LP64` 欄の型です。これらの型はすべて `<sys/types.h>` に定義されています。例外は `wchar_t` と `wint_t` 型で、これらは `<wchar.h>` に定義されています。

これらの表を再確認するときには、32 ビット環境では `int`、`long`、およびポインタは 32 ビット量である、ということを憶えておいてください。64 ビット環境では、`int` は 32 ビット量ですが、`long` およびポインタは 64 ビット量です。

表 **A–2** 変更された派生型 — 一般

派生型	**Solaris 2.6**	`_ILP32`	`_LP64`
blkcnt_t	longlong_t	longlong_t	long
id_t	long	long	int
major_t	ulong_t	ulong_t	uint_t

表 **A–2** 変更された派生型 — 一般　(続き)

派生型	**Solaris 2.6**	**_ILP32**	**_LP64**
minor_t	ulong_t	ulong_t	uint_t
mode_t	ulong_t	ulong_t	uint_t
nlink_t	ulong_t	ulong_t	uint_t
paddr_t	ulong_t	ulong_t	未定義
pid_t	long	long	int
ptrdiff_t	int	int	long
size_t	uint_t	uint_t	ulong_t
ssize_t	int	int	long
uid_t	long	long	int
wchar_t	long	long	int
wint_t	long	long	int

表 A–3 に、大規模ファイルのコンパイル環境に固有の派生型を示します。これらの型は、機能テストマクロの _LARGEFILE64_SOURCE が定義されている場合にのみ定義されます。ILP32 コンパイル環境が Solaris 2.6 と同じであることに注意してください。

表 **A–3** 変更された派生型 — 大規模ファイルの場合のみ

派生型	**Solaris 2.6**	**_ILP32**	**_LP64**
blkcnt64_t	longlong_t	longlong_t	blkcnt_t
fsblkcnt64_t	u_longlong_t	u_longlong_t	fsblkcnt_t
fsfilcnt64_t	u_longlong_t	u_longlong_t	fsfilcnt_t
ino64_t	u_longlong_t	u_longlong_t	ino_t
off64_t	longlong_t	longlong_t	off_t

表 A–4 に、_FILE_OFFSET_BITS の値に関する派生型を示します。_LP64 を定義し _FILE_OFFSET_BITS==32 に設定して、アプリケーションをコンパイルすることはできません。_LP64 が定義された場合、デフォルトで _FILE_OFFSET_BITS==64 です。_ILP32 が定義されて _FILE_OFFSET_BITS が定義されない場合、デフォルトで _FILE_OFFSET_BITS==32 となります。これらの規則は <sys/feature_tests.h> ヘッダーファイルに定義されています。

表 **A–4** 変更された派生型 — FILE_OFFSET_BITS 値

派生型	_ILP32 _FILE_ OFFSET_BITS ==32	_ILP32 _FILE_ OFFSET_BITS ==64	_LP64 _FILE_ OFFSET_BITS==64
ino_t	ulong_t	u_longlong_t	ulong_t
blkcnt_t	long	longlong_t	long
fsblkcnt_t	ulong_t	u_longlong_t	ulong_t
fsfilcnt_t	ulong_t	u_longlong_t	ulong_t
off_t	long	longlong_t	long

付録 B

よく尋ねられる質問 (FAQ)

32 ビットバージョンと **64** ビットバージョンのどちらのオペレーティングシステムが動作しているかは、どのようにしたらわかりますか？

`isainfo -v` コマンドを使用すると、オペレーティングシステムが実行できるアプリケーションを調べることができます。 詳細は、`isainfo(1)` のマニュアルページを参照してください。

32 ビットのハードウェア上で **64** ビットのオペレーティングシステムを実行できますか？

できません。64 ビットオペレーティングシステムを 32 ビットハードウェア上で実行することはできません。64 ビットオペレーティングシステムには、64 ビットの MMU と CPU ハードウェアが必要です。

32 ビットアプリケーションを **32** ビットオペレーティング環境のシステム上で実行する場合、アプリケーションを変更する必要がありますか？

いいえ。32 ビットオペレーティング環境のシステム上でのみアプリケーションを実行する予定ならば、変更も再コンパイルも不要です。

32 ビットアプリケーションを **64** ビットオペレーティング環境のシステム上で実行する場合、アプリケーションを変更する必要がありますか？

ほとんどのアプリケーションは、コード変更も再コンパイルもせずに 32 ビットのまま、64 ビットオペレーティング環境のシステム上で従来どおり実行することができます。64 ビット機能を必要としない 32 ビットアプリケーションは 32 ビットのままにしておくと、移植性を保つことができます。

`libkvm(3LIB)` を使用しているアプリケーションを 64 ビットオペレーティング環境のシステムで実行するには、アプリケーションを 64 ビットとして再コンパイルする必要があります。 また、アプリケーションが `/proc` を使用している場合も、64 ビットとして再コンパイルする必要があります。そうしなければ、64 ビットプロセスを認識できません。これは、プロセスを記述している既存のインタフェースとデータ構造体が 64 ビット量を格納できる程大きくないからです。

64 ビット機能を利用するにはどんなプログラムを起動する必要がありますか？

特に 64 ビット機能を起動するためのプログラムはありません。64 ビットオペレーティング環境上のシステムの 64 ビット機能を利用するためには、アプリケーションを再構築する必要があります。

64 ビットオペレーティング環境のシステム上で **32** ビットアプリケーションを構築できますか？

できます。ネイティブコンパイルモードおよびクロスコンパイルモードがサポートされています。実行しているオペレーティングシステムが 32 ビットと 64 ビットのどちらであるかとは無関係に、デフォルトのコンパイルモードは 32 ビットです。

32 ビットオペレーティング環境のシステム上で **64** ビットアプリケーションを構築できますか？

システムヘッダーと 64 ビットライブラリをインストールしてあれば可能です。ただし、32 ビットオペレーティング環境のシステム上では、64 ビットアプリケーションを実行することはできません。

アプリケーションを構築しリンクするときに **32** ビットライブラリと **64** ビットライブラリを組み合わせて使うことはできますか？

できません。32 ビットアプリケーションは 32 ビットライブラリとリンクし、64 ビットアプリケーションは 64 ビットライブラリとリンクしなければなりません。異なるバージョンのライブラリと一緒に構築あるいはリンクしようとすると、エラーになります。

64 ビット実装上の浮動小数点データ型のサイズは？

32 ビットと 64 ビットとでサイズが変更されたデータ型は `long` と ポインタのみです。表 4–1 を参照してください。

`time_t` のサイズはどうですか **?**

`time_t` 型は `long` のままです。64 ビット環境では、これは 64 ビット量に拡張されます。したがって、64 ビットアプリケーションは 2038 年問題に関して安全です。

64 ビット **Solaris** オペレーティングシステムを実行中のマシンでは `uname(1)` の出力値はどのようになりますか？

`uname -p` コマンドの出力は変更されていません。

64 ビットの **XView** アプリケーションまたは **OLIT** アプリケーションを作成することはできますか？

できません。32 ビット環境において旧式のライブラリである XView または OLIT のライブラリは 64 ビット環境に対応していません。

`/usr/bin/sparcv9/ls` に `ls` の **64** ビット版があるのはなぜですか？

通常の処理では 64 ビット版の `ls` は必要ありません。ただし、32 ビット版の ls が認識するには大きすぎるファイルシステムオブジェクトを `/tmp` と `/proc` に作成できるため、64 ビット版の `ls` を使用することによってそれらのオブジェクトを確認できます。

索引

数字・記号
64 ビット演算, 19
64 ビットライブラリ, 20

A
ABI,「SPARC V9 ABI」を参照
amd64 ABI, アドレス空間の配置, 63
API, 23

D
/dev/ksyms, 67

E
ELF, 65-66
EOVERFLOW, 69

G
GELF, 65-66

I
ILP32, 7
<inttypes.h>, 31-33
ioctl(2), 70
isainfo(1), 25
isalist(1), 26

L
LD_LIBRARY_PATH, 50
libkstat, 68
libkvm, 67
lint, 34-36
LP64, 7
　変換のためのガイドライン, 36-43

O
$ORIGIN, 50-51

P
/proc, 8, 66
/proc の制約, 定義, 20

S
sizeof, 41-42
SPARC V9 ABI
　アドレス空間の配置, 59-60
　スタックバイアス, 58-59
stdio, 変更, 68
sysinfo(2), 拡張, 66

<sys/types.h>, 30-31

U
uintptr_t, 32

か
カーネルメモリーリーダー, 20

こ
コードモデル, 61-62
互換性
アプリケーションソースコード, 25
アプリケーションバイナリ, 24
デバイスドライバ, 25
コンパイラ, 49

し
書式文字列マクロ, 33

せ
制限値, 33

そ
相互運用性の問題, 20

た
大規模ファイル, 7
定義, 19
大容量仮想アドレス空間, 7
定義, 19

て
定数マクロ, 32
データ型モデル
「ILP32」を参照
「LP64」も参照
デバッグ処理, 54-55

は
派生型, 30
パッケージ処理
アプリケーション命名規則, 52
パッケージ処理のガイドライン, 52
ライブラリとプログラムの配置, 51-52

ふ
符号の拡張, 37-38
整数の昇格, 37
変換, 37
プロセス間通信, 64-65

へ
ヘッダー, 47-48

ほ
ポインタ演算, 39

ら
ライブラリ, 49-50
ラッパー
isaexec(3C), 54
/usr/lib/isaexec, 53-54

り
リンク処理, 50-51