



Solaris 7 64 ビット 開発ガイド

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

Part Number 806-0018-10
1999 年 4 月

Copyright 1999 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. All rights reserved.

本製品およびそれに関連する文書は著作権法により保護されており、その使用、複製、頒布および逆コンパイルを制限するライセンスのもとにおいて頒布されます。日本サン・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。

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

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

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

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

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

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

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ぬものとします。

本製品が、外国為替および外国貿易管理法 (外為法) に定められる戦略物資等 (貨物または役務) に該当する場合、本製品を輸出または日本国外へ持ち出す際には、日本サン・マイクロシステムズ株式会社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 7 64-bit Developer's Guide

Part No: 805-6250-10

Revision A

© 1999 by Sun Microsystems, Inc.



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 19
 - 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 20
 - 特定のシステム制約の解消 20
 - 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 21
 - 64 ビットライブラリ 21
 - 変換作業 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
 - 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	24
	デバイスドライバ	24
	どの Solaris オペレーティング環境を実行するか	25
	プログラミング例	27
4.	アプリケーションの変換	29
	データ型モデル	29
	単一ソースコードの実装	31
	派生型	31
	<sys/types.h>	32
	<inttypes.h>	32
	ツール	35
	lint(1)	36
	LP64 への変換のためのガイドライン	37
	int と ポインタが同じサイズであると仮定しない	37
	int と long が同じサイズであると仮定しない	38
	符号の拡張	38
	アドレス演算の代わりにポインタ演算を使う	40
	構造体の再構成	40
	共用体	41
	定数の型指定	41
	暗黙的宣言について	42
	sizeof() は符号なし long である	43
	意図を示すためにキャストを使う	43
	書式文字列の変換をチェックする	43
	その他の検討事項	44
	サイズが拡大した派生型	44

	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	45
	呼び出し規約の変更	45
	アルゴリズムの変更	46
	チェックリスト (64 ビットに変換する前に)	46
5.	開発環境	47
	構築環境	47
	ヘッダー	47
	コンパイラ	49
	ライブラリ	50
	リンク処理	50
	LD_LIBRARY_PATH	50
	\$ORIGIN	51
	パッケージ処理	52
	ライブラリとプログラムの配置	52
	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
	ラッパー	53
	/usr/lib/isaexec	54
	isaexec(3C)	55
	デバッグ処理	55
6.	上級者向けトピック	57
	アプリケーションに関する新しい情報	57
	汎用 64 ビット ABI の特徴	57
	SPARC V9 ABI の特徴	58
	アドレス空間の配置	59
	テキストおよびデータの配置	60
	プロセス間通信	60

ELF とシステム生成ツール	61
/proc	62
libkvm と /dev/ksyms	62
libkstat	63
stdio	64
パフォーマンスの問題	64
64 ビットアプリケーションの長所	64
64 ビットアプリケーションの短所	64
システムコールの問題	65
Eoverflow の意味	65
ioctl(2) に関する注意	65
A. 派生型の変更	67
B. よく尋ねられる質問 (FAQ)	73
C. 64 ビット 開発ガイドの更新内容	77
索引	79

はじめに

Solaris™ オペレーティング環境の機能は、顧客の需要を満たすために拡大し続けています。Solaris 7 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、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。

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、他の基本データ型は 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`、`/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
- `/proc` を利用して 64 ビットプロセスをデバッグする
- 64 ビット版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

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。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も、64 ビットに変換することがあります。

このマニュアルの付録に、変更された派生型と、よく尋ねられる質問を掲載してあります。さらに、参考として次の文献をお薦めします。

- 『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: 96115-001)
- 該当するコマンドのマニュアルページ
- 『Writing Device Drivers』 (Part No: 805-3024)
- 『C ユーザーズガイド』 (Part No: 805-7885)

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が 32 ビットまたは 64 ビットであるかを判定する方法について説明しています。また、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、および 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について説明しています。

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 ビットと 64 ビットのアプリケーションおよびインタフェース間の類似性について説明しています。
- 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、頻繁に尋ねられる質問とその回答をまとめてあります。
- 付録 C では、このマニュアルでの更新情報を説明します

マニュアルの注文方法

SunDocs プログラムでは、米国 Sun Microsystems™, Inc. (以降、Sun™ とします) の 250 冊以上のマニュアルを扱っています。このプログラムを利用して、マニュアルのセットまたは個々のマニュアルをご注文いただけます。

マニュアルのリストと注文方法については、米国 SunExpress™, Inc. のインターネットホームページ <http://www.sun.com/sunexpress> にあるカタログセクションを参照してください。

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、または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は区別して示します。	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 <i>filename</i> と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、または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越える場合、バックスラッシュは継続を示します。	sun% grep `^#define \ XV_VERSION_STRING`

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

コード例は次のように表示されます。

■ C シェルプロンプト

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

■ Bourne シェルおよび Korn シェルのプロンプト

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

■ スーパーユーザーのプロンプト

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

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

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

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

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

一般規則

- このマニュアルでは、英語環境での画面イメージを使っています。このため、実際に日本語環境で表示される画面イメージとこのマニュアルで使っている画面イメージが異なる場合があります。本文中で画面イメージを説明する場合には、日本語のメニュー、ボタン名などの項目名と英語の項目名が適宜、併記されています。
- 「x86」という用語は、一般に Intel 8086 ファミリーに属するマイクロプロセッサを意味します。これには、Pentium、Pentium Pro の各プロセッサ、および AMD と Cyrix が提供する互換マイクロプロセッサチップが含まれます。このマニュアルでは、このプラットフォームのアーキテクチャ全体を指すときに「x86」という用語を使用し、製品名では「Intel 版」という表記で統一しています。

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。たとえば、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、および科学技術計算などがあります。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) の関係を表す曲線を示します。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。

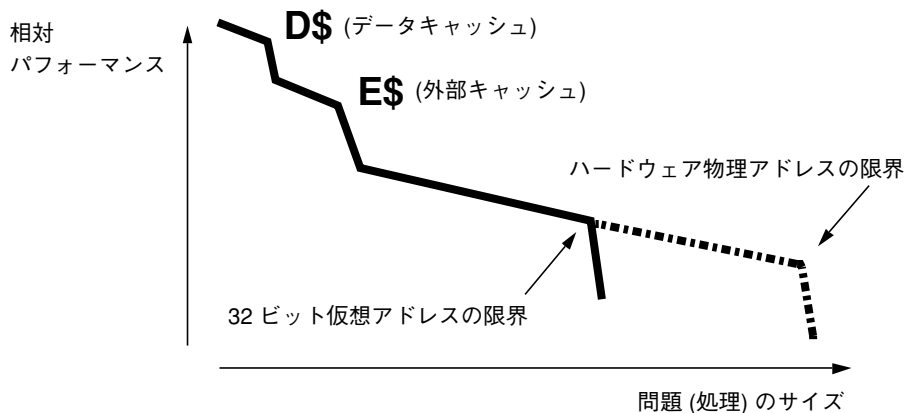


図 1-1 典型的なパフォーマンス対問題サイズ曲線

32 ビット仮想アドレスの限界を超えても、一般的には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することによって、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。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数桁以上も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 1 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 E (エクサ) 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

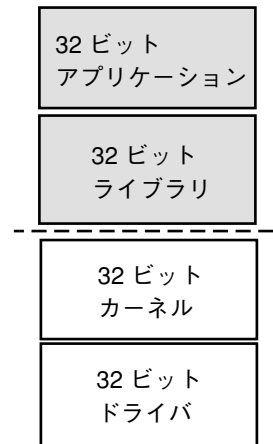
64 ビットアプリケーションを作成する理由として、次のことを挙げることができます。

- 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実行し、パフォーマンスを向上させる必要がある。
- システムインター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ーフェースが拡張されたり制約が解消された。

64 ビットの使用方法

図 2-1 に、Solaris オペレーティング環境において 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。

32 ビット Solaris



64 ビット Solaris

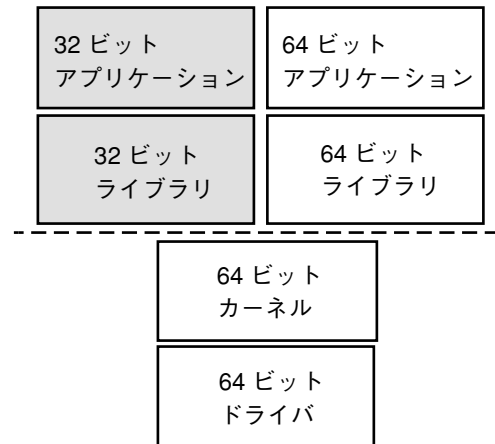


図 2-1 Solaris 7 オペレーティング環境のアーキテクチャ

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環境間の主な相違は、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、int、long、ポインタが 32 ビット長である ILP32 データ型モデルに基づいています。これらのデータ型モデルについては、第 4 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もよく、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勧めします。

次のような特性を持っているアプリケーションは、変換が必要な場合があります。

- 4G バイト以上の仮想アドレス空間を必要とす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 ビットレジスタが必要なアプリケーション

次のような特性を持っているアプリケーションは、変換が必要です。

- 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使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があります。

主な特徴

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 ビット計算
- 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 ビットのままでよく、大規模ファイルインタフェースを使用できます。ただし、移植性が重要でない場合には、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群とともに 64 ビット機能を活用することをお勧めします。

64 ビット演算

64 ビット計算は、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` が時間を使い果たすこと) 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 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 64 ビット安全にしたり 64 ビットプログラム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
- 連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 ビットプロセスとを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。これは通常デバッガを使用する時に問題となります。

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。古いライブラリ除いて、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。ただし、静的ライブラリとして提供されている 64 ビットライブラリはありません。

変換作業

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、多くのアプリケーションの場合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。

32 ビットと 64 ビットインタフェースの比較

第 2 章で説明したように、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 64 ビット Solaris 環境で動作します。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したり、完全 64 ビットプログラムに変換する必要があります。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、第 2 章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 ビットおよび 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になります。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。

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 ビットバージョンを作成するには、その派生型のいくつかを 64 ビットに拡張します。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。

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。追加情報については、57ページの「アプリケーションに関する新しい情報」を参照してください。

互換性

32 ビットおよび 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 SPARC アプリケーションは、バイナリの互換性があり、32 ビットまたは 64 ビットのオペレーティングシステム環境で実行できます。唯一の例外は、libkvm、/dev/kmem、または /proc を使用するアプリケーションです。詳細は、第 2 章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。正しい派生型およびインタフェースを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システム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ています。ただし、基本ドライバモデル、つまり 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。

どの Solaris オペレーティング環境を実行するか

Solaris は、2 つの first class 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。図 2-1 を参照してください。

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。32 ビット Solaris と 64 ビット Solaris の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をすぐには判断できません。isainfo(1) コマンドを使用すると、これを簡単に確認することができます。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(1) コマンドの出力例です。

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

同じコマンドを古い SPARC システム、または 32 ビットオペレーティングシステムが実行されている UltraSPARC システム上で実行した場合は、次のように出力されます。

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

同じコマンドを x86 システム上で実行すると次のようになります。

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

isainfo(1) コマンドのオプションの 1 つに `-n` オプションがあります。このオプションは、動作中のプラットフォームにネイティブな命令セットの名前を出力します。

```
% isainfo -n
sparcv9
```

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は、次のようにして確認できます。

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

`uname -r` 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て、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 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。

関連コマンドの `isalist(1)` は、シェルスクリプトに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。`isalist(1)` で出力される命令セットアーキテクチャのなかには、かなりプラットフォームに固有なものもありますが、`isainfo(1)` ではシステム上で最も移植性のあるアプリケーション環境の属性だけを出力します。これら 2 つのコマンドとも、`sysinfo(2)` システムコールの `SI_ISALIST` サブオプションを使って構築されています。詳細は `isalist(1)` のマニュアルページを参照してください。

次の例は、64 ビットオペレーティングシステムが動作している UltraSPARC システム上で `isalist(1)` コマンドを実行したときの出力です。

```
% isalist
sparcv9+vis sparcv9 sparcv8plus+vis sparcv8plus sparcv8
sparcv8-fsmuld sparcv7 sparc
```

サポートされる命令セットのリストは、パフォーマンスおよび機能のレベル順にソートされて出力されます。このコマンドの使用法についての詳細は、`isalist(1)` のマニュアルページを参照してください。また、`optisa(1)` のマニュアルページも参照してください。

注 - 上記の `isalist(1)` コマンドの出力例中にある `sparcv9+vis` および `sparcv8+vis` は、システムが UltraSPARC-1 拡張命令セットをサポートしていることを示しています。この拡張命令セットはすべての SPARC V9 システムでサポートされているわけではないので、移植できるアプリケーションはこの拡張命令セットに依存しないようにしてください。また、UltraSPARC に固有の機能に合わせてコーディングしないようにしてください。

プログラミング例

次に示すプログラミング例では、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。

注 - デフォルトのコンパイル環境は、移植性が維持されるように 32 ビットアプリケーションを作成するように設定されています。

```
% cat foo.c
#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例

```
% 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例

```
% cc -xarch=v9 -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
```

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、`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。またこの LP64 データ型モデルは、業界の企業コンソーシアムで合意を得ています。C のデータ型の `int`、`short`、`char` は、ILP32 データ型モデルと同じです。

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 の基本データ型と、それらの ILP32 および LP64 のデータ型モデルでのサイズをビット単位で示します。

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
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

32 ビットアプリケーションにおいては、しばしば int、ポインタ、および long が同じサイズであると仮定します。LP64 データ型モデルでは、long とポインタのサイズが変更されています。この点から、32 ビットから 64 へ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せん。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

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、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。

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必要があるからです。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う便利です。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。

システムインクルードファイルの `<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、および派生型が格納されています。これらについての詳細は、このマニュアルでは説明していませんが、一部についてはこの章の以降の節および付録 A で説明しています。

<sys/types.h>

<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 `_LP64` と `_ILP32` の定義を利用できるようになります。このヘッダーには、それぞれ必要に応じて適切な箇所で使用される基本派生型が多数含まれています。特に次のものは重要です。

<code>clock_t</code>	システムクロック時間を表わします。
<code>dev_t</code>	デバイス番号に使用される型です。
<code>off_t</code>	ファイルサイズとオフセット用に使用される型です。
<code>ptrdiff_t</code>	2つのポインタの減算結果を示す符号付き整数型です。
<code>size_t</code>	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。
<code>ssize_t</code>	バイト数またはエラーを返す関数によって使用される型です。
<code>time_t</code>	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。

これらの型のいくつかの使用方法については、37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>

インクルードファイルの <inttypes.h> は、定数、マクロ、および派生型を定義するために Solaris 2.6 に追加されました。これによって、明示的にサイズを指定されたデータ項目について、コンパイル環境とは無関係にコードに互換性を持たせることができます。<inttypes.h> 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います。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

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`、および `uint64_t` のような、符号付き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_least16_t`、`int_least32_t`、`int_least64_t`、`uint_least8_t`、`uint_least16_t`、`uint_least32_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
- バイナリデータ構造

`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領域が確保できる符号付き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。

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)`、`INT16_C(C)`、`INT32_C(C)`、`INT64_C(c)`、`UINT8_C(c)`、`UINT16_C(C)`、`UINT32_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` の後に `ll` の場合には `ll` を、`LP64` の場合には `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、37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。

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。これには、`INT8_MIN`、`INT16_MIN`、`INT32_MIN`、`INT64_MIN`、`INT8_MAX`、`INT16_MAX`、`INT32_MAX`、`INT64_MAX`、およびそれらの符号なしのものなど、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。これらには、`INT_LEAST8_MIN`、`INT_LEAST16_MIN`、`INT_LEAST32_MIN`、`INT_LEAST64_MIN`、`INT_LEAST8_MAX`、`INT_LEAST16_MAX`、`INT_LEAST32_MAX`、`INT_LEAST64_MAX`、およびそれらの符号なしのものなどがあります。

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

書式指示子 `printf(3S)` と `scanf(3S)` 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(3S)` 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。次の例を参照してください。

例 4-1

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

同様に、`scanf(3S)` 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。

例 4-2

```
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、33 ページの「固定幅整数型」を参照してください。

ツール

Sun WorkShop Compilers C のバージョン 5.0 に `lint(1)` プログラムの新しいバージョンが Sun から提供されています。64 ビットで発生する可能性がある問題を検出できるように強化され、コードを 64 ビット安全にするのに役に立ちます。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が行われます。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。

コードを 64 ビット安全に修正する場合、64 ビット環境用の派生型とデータ構造を正しく定義しているヘッダーファイルを使用してください。つまり、Solaris 7 オペレーティング環境にあるヘッダーファイルを使用してください。

Sun WorkShop Compilers C のデバッグ機能と `lint(1)` の詳細については、『C ユーザーズガイド』を参照してください。

lint (1)

lint(1) 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` フラグは、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 環境で実行するコードに対して lint を実行する場合に使用します。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。

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

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とは、64 ビット問題を特定し、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。

次に、考慮すべき警告をいくつか例として示します。

```
warning: conversion of pointer loses bits
warning: assignment causes implicit narrowing conversion
warning: cast to pointer from different-bit integer
warning: cast from bigger-bit integer to smaller-bit integer
warning: assignment of bigger-bit integer to smaller-bit integer
warning: bigger-bit constant truncated to smaller bits by assignment
warning: passing bigger-bit integer arg, expecting smaller-bit integer:
    routine name(arg argument number)
warning: passing bigger-bit integer constant arg, expecting smaller-bit
integer: routine name(arg argument number)
warning: function argument (number) type inconsistent with format
    printf (arg argument number) type used: (format) type expected file (line)
```

(続く)

```
warning: function argument (number) type inconsistent with format:
scanf (arg argument number) type used: (format) type expected file (line)
warning: cast from bigger-bit constant expression to smaller-bit integer
```

`/*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。`lint(1)` コマンドについての詳細は、『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、ということを覚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。

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 と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。

1. `lint` は、キャストされる型に定数式を変換できない場合にだけ、警告を出します。

例 4-3

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

%
warning: conversion of pointer loses bits
```

推奨される使用法

```
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 int と long が同じサイズとして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しなければなりません。ILP32 データ型モデルでは、int と long は 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。

例 4-4

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

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

符号の拡張

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。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、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、符号付きあるいは符号なしに関わらず、int を呼び出す式の中で使用できます。int が元の型の取り得る値をすべて格納することができる場合は、その値は signed 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 4-5

```
%cat 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; /* 符号拡張あり */
    printf("addr 0x%lx\n", addr);

    addr = (unsigned int)(a.base << 13); /* 符号拡張なし */
    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。

例 4-6

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

%
warning: conversion of pointer loses bits
```

推奨される使用法

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

構造体の再構成

パディングが必要な場所について、アプリケーションにおける内部データ構造をチェックする必要があります。LP64 データ型モデルでは long またはポインタフィールドは 64 ビットに拡張されるので、境界を整列するために構造体内のフィールド間にパディングを行うことができます。SPARC プラットフォーム上の 64 ビット環境では、構造体の型はすべて、少なくとも構造体内にある一番大きいサイズに

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始めの位置に移動することです。

例 4-7

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

推奨される使用法

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

共用体

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。

例 4-8

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

推奨される使用法

```
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。各整数定数の後に (u、U、l、L) を組合せたものを追加して、定数式にデータ型を明示的に指定してください。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。

例 4-9

```
int i = 32;
long j = 1 << i; /* 右辺は整数式なので、j には 0 が */
                /* 代入されます */
```

推奨される使用法

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

暗黙的宣言について

Sun WorkShop™ で提供される C 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。このように使用される long およびポインタは、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。

例 4-10

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

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

推奨される使用法

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

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

sizeof() は符号なし long である

LP64 データ型モデルでは、sizeof() は unsigned long の実効的なデータ型をもちます。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。

例 4-11

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

%
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(3S)、sprintf(3S)、scanf(3S)、sscanf(3S) が long またはポインタ引数に対して使用されている場合、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。

例 4-12

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

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

推奨される使用法

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

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。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であることを確認してください。

例 4-13

```
size_t nbytes;
u_long 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);

%
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
```

推奨される使用法

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

その他の検討事項

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 ビットを表わすために、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、utmp(4) または 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修正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。

呼び出し規約の変更

構造体を値渡しで渡し、SPARC V9 用のコードをコンパイルする場合、構造体が小さいと、その値のコピーを指すポインタとしてではなくレジスタ経由で値が渡されます。これは、C コードと手作業で記述したアセンブリコードとの間で構造体を渡す際に問題が発生します。

浮動小数点パラメータも同様に動作します。つまり、値渡しで渡された浮動小数点の値が、浮動小数点レジスタに渡される場合があります。

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ります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 37 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>` (または、少なくとも `<sys/isa_defs.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` および `-Xarch=v9` フラグを指定して `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、その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。

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。これを確認するには、第 3 章で説明した `isainfo(1)` コマンドを使用することができます。この章での説明は、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ています。32 ビットオペレーティングシステム環境を使用している場合でも、システム上に 64 ビットライブラリパッケージがインストールされていれば、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

SPARC マシンの構築環境には、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。

ヘッダー

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勧めします。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。

<code>__sparc</code>	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 <code>sparc</code> は、慣例上 <code>__sparc</code> の同義語です。
<code>__sparcv8</code>	『SPARC Architecture Manual』のバージョン 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。
<code>__sparcv9</code>	『SPARC Architecture Manual』のバージョン 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。
<code>__i386</code>	このシンボルは、Intel 386 命令セットあるいはそのスーパーセットを実装するすべてのプロセッサに対する総称です。これには、386、486、および Pentium ファミリーのプロセッサのすべてのメンバーが含まれます。

シンボル `__sparcv8` および `__sparcv9` は、相互に排他的で (両方が同時に定義されることはありません)、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。

<code>_ILP32</code>	<code>int</code> 、 <code>long</code> 、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
<code>_LP64</code>	<code>long</code> 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

シンボル `_ILP32` および `_LP64` も、SPARC マシンに固有ではありませんが、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、`__sparcv8` または `__sparcv9` ではな

く、`_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、コンパイラ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他の 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

Sun WorkShop 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 WorkShop で提供される C コンパイラの 5.0 リリースは、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、Intel マシン上で SPARC オブジェクトを作成したり、SPARC マシン上で Intel 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 `__sparc` 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。

たとえば、Sun WorkShop C コンパイラを使用して SPARC マシンで 64 ビットコンパイル環境を使用できるようにする場合は、`-xarch=v9` フラグを `cc(1)` への引数として指定する必要があります。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=v9` フラグは、ネイティブコンパイルおよびクロスコンパイルの両方のモードで有効です。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。

ライブラリ

Solaris 7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。静的ライブラリについては、32 ビットのみが含まれており、64 ビットは提供されていません。

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。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、SPARC プラットフォームでは適切な `lib` ディレクトリの `sparcv9` 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環境で構築された 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。

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。

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

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。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

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、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、オートマウント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。このようなことはもはや必要ありません。これは、\$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るからです。\$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。

パッケージ処理

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

ライブラリとプログラムの配置については、前述の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勧めします。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 (32 ビットと 64 ビットとの違いを意識する必要がない) 場所に置くようにしてください。

つまり SPARC マシンでは、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く必要があります。x86 マシン上での 32 ビットライブラリの場所は、SPARC マシンと同じです。

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下の sparcv7 または sparcv9 サブディレクトリに適宜置くことをお勧めします。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勧めします。

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、32 ビットおよび 64 ビットアプリケーションを 52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、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、53ページの「ラッパー」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。

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 7 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(1)` 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(3C)` 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isaexec(3C) ラッパーについて説明します。

`/usr/lib/isaexec`

isaexec(3C) は 32 ビット実行可能バイナリファイルです。直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、この名前で行う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、`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` 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。

isaexec(3C) ラッパーは、完全に解決されたシンボリックリンクがない自分のパス名を、`argv[0]` 引数とは別に `getexecname(3C)` を使って調べ、`sysinfo(SI_ISALIST, ...)` を使用して `isalist` 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 `execve(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)

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(4) 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。

デバッグ処理

truss(1) コマンド、/proc ツール、adb(1)、dbx、および、adbgen(1M) などの Solaris 上で実行できるデバッグツールのすべてが、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、dbx デバッグが以外はすべて Solaris に含まれています。

adbgen(1M) は、指定した adb(1) マクロを作るプログラムを生成します。64 ビットマクロを生成するために `-m lp64` オプションを指定し、64 ビットシステム上で実行する必要があります。詳細は、adbgen(1M) の マニュアルページを参照してください。

adbgen(1M) 以外の上記のデバック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、adb(1) に対しては、多数の拡張が加えられました。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、次の修飾子が新たに追加されました。

追加された `?`, `/`, `=` の修飾子:

```
g (8) 8 バイトを符号なしとして 8 進表示
G (8) 8 バイトを符号ありとして 8 進表示
e (8) 8 バイトを符号ありとして 10 進表示
E (8) 8 バイトを符号なしとして 10 進表示
J (8) 8 バイトを 16 進表示
K (n) ポインタまたは long を 16 進表示
      32 ビットプログラムでは 4 バイトを表示
      64 ビットプログラムでは 8 バイトを表示
Y (8) 8 バイトを日付表示
```

追加された `?`, `/` の修飾子

```
M <value> <mask>
<mask> でマスクして 8 バイト値を比較。
'.' を一致した位置に移動
Z (8) 8 バイトを書き込み
```


上級者向けトピック

この章では、64 ビット Solaris オペレーティング環境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、システムプログラミングに関するさまざまな情報を提供します。

アプリケーションに関する新しい情報

64 ビット環境のほとんどの新機能は、32 ビットインタフェースを拡張した汎用機能ですが、一部の 64 ビット新機能は SPARC V9 に固有の機能です。

『*SPARC Compliance Definition, Version 2.4*』には、SPARC V9 ABI の詳細が含まれます。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。この文書は、www.sparc.com にアクセスすると、SPARC International から入手できます。

汎用 64 ビット 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。

SPARC V9 ABI の特徴

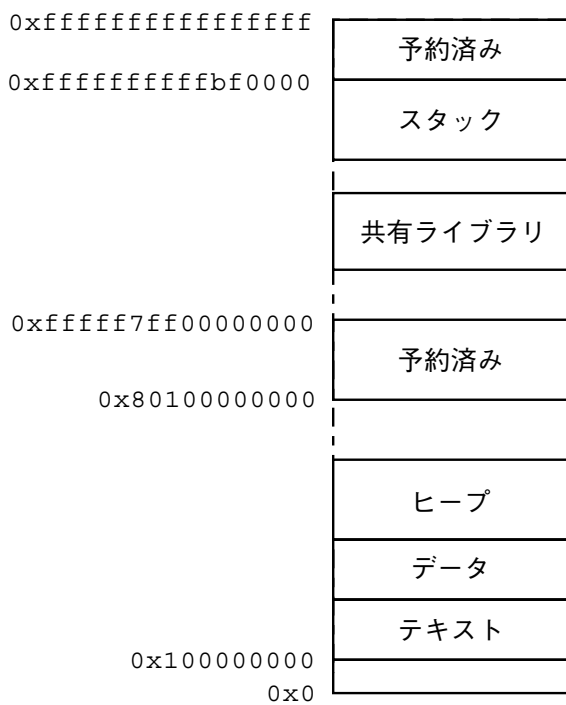
次に SPARC V9 ABI の特徴を示します。

- 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 ビットです。
- 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、アドレス空間内に分岐させるため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。

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 V8 と同様に、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。



上述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。

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 0x10000000 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、リンカーのマッピングファイルを使用し、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ピングファイルは、`/usr/lib/ld/sparcv9/map.below4G` にあります。詳細は、`ld(1)` リンカーのマニュアルページを参照してください。

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。

- システム V IPC プリミティブ、たとえば `shmop(2)`、`semop(2)`、`msgsnd(2)`
- 共有ファイル上の `mmap(2)`
- プロセス間の `pipe(2)`
- プロセス間の `door_call(3X)`
- `xdr(3N)` に説明されている外部データ表現を使用した、同じあるいは異なるマシン上のプロセス間の `rpc(3N)`

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、データが正確に同じサイズで、両プロセスで意味があることを確認することです。データ構造が `int32_t` や `int64_t` のような固定幅型を使って構成されていることを確認してください。

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が、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`) はあるが、64 ビットポインタに等価なものは 32 ビット環境にはない、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には、32 ビットプロセスは共有データのうち、4G バイトまでだけを一度に「見る」ことができるという点に注意が必要です。

XDR ルーチンの `xdr_long(3N)` は問題と思われるかもしれません。しかし、これは既存の protocol 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(3E)` API、たとえば `elf64_getehdr(3E)` を使って読むことができます。

ELF ライブラリ `libelf(4)` 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(3E) 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

/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。

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必要があ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 libkvm(4) 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(3K)、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 libkvm(4) の代わりに使用できるのなら、移植性を最大限維持するために、それらを使用してアプリケーションを 32 ビットのままとしておく

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(4)` は、64 ビットのカーネルクラッシュダンプに対して `kvm_open(3K)` 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(4)` は、32 ビットカーネルクラッシュダンプに対して `kvm_open(3K)` 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(3K)` または `kvm_write(3K)` 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(3K)` と `kvm_write(3K)` を使っている `libkvm(4)` を利用するアプリケーションはすべて、`kvm_kread(3K)`、`kvm_kwrite(3K)`、`kvm_uread(3K)`、`kvm_uwrite(3K)` のルーチンを使う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、問題が発生します。これはデータ構造のオフセットおよびサイズは、確実に 32 ビットおよび 64 ビットカーネル間で異なるためです。

libkstat

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(3K)` 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明で、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勧めします。

stdio

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、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ことをお勧めします。

`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(3S)` 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 ビット量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方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方がより効率的である

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

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(2) 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(2)` 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(2)` 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(2)` 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。

例 6-1

```
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、実行される時、どちらの `ioctl(2)` 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(2)` 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ています。リトルエンディアン実装の場合でも、コンテナには上位の 32 ビットに意味のない値が含まれます。2 番目の `ioctl(2)` 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。

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 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
<code>_ILP32</code>	<code>_ILP32</code> 機能テストマクロは、 <code>int</code> 、 <code>long</code> 、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。
<code>_LP64</code>	<code>_LP64</code> 機能テストマクロは、 <code>int</code> は 32 ビット量で <code>long</code> 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。 <code>_LP64</code> は、64 ビットモード (<code>-xarch=v9</code>) でコンパイルするときにデフォルトで定義されます。 <code><sys/types.h></code> または <code><sys/isa_defs.h></code> がソースにインクルードされて <code>_LP64</code> 定義が見えるようになっていない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

```
#if defined(_LP64)
typedef ulong_t size_t; /* size of something in bytes */
#else
typedef uint_t size_t; /* (historical version) */
#endif
```

この例の 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 ビットです。

例 A-2

```
#if defined(_LP64)
typedef int uid_t; /* UID type */
#else
typedef long uid_t; /* (historical version) */
#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れば、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が、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が、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++ アプリケーション同士の整合性と、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 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、ということを覚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
<code>blkcnt_t</code>	<code>longlong_t</code>	<code>longlong_t</code>	<code>long</code>
<code>id_t</code>	<code>long</code>	<code>long</code>	<code>int</code>
<code>major_t</code>	<code>ulong_t</code>	<code>ulong_t</code>	<code>uint_t</code>
<code>minor_t</code>	<code>ulong_t</code>	<code>ulong_t</code>	<code>uint_t</code>
<code>mode_t</code>	<code>ulong_t</code>	<code>ulong_t</code>	<code>uint_t</code>
<code>nlink_t</code>	<code>ulong_t</code>	<code>ulong_t</code>	<code>uint_t</code>
<code>paddr_t</code>	<code>ulong_t</code>	<code>ulong_t</code>	未定義
<code>pid_t</code>	<code>long</code>	<code>long</code>	<code>int</code>
<code>ptrdiff_t</code>	<code>int</code>	<code>int</code>	<code>long</code>
<code>size_t</code>	<code>uint_t</code>	<code>uint_t</code>	<code>ulong_t</code>

表 A-2 変更された派生型 — 一般 続く

派生型	Solaris 2.6	_ILP32	_LP64
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 と同じであることを注意してください。

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	blkcnt_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、デフォルトでは `_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。これらの規則はヘッダー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	<code>_ILP32_FILE_OFFSET_BITS == 32</code>	<code>_ILP32_FILE_OFFSET_BITS == 64</code>	<code>_LP64_FILE_OFFSET_BITS == 64</code>
<code>ino_t</code>	<code>ulong_t</code>	<code>u_longlong_t</code>	<code>ulong_t</code>
<code>blkcnt_t</code>	<code>long</code>	<code>longlong_t</code>	<code>long</code>
<code>fsblkcnt_t</code>	<code>ulong_t</code>	<code>u_longlong_t</code>	<code>ulong_t</code>
<code>fsfilcnt_t</code>	<code>ulong_t</code>	<code>u_longlong_t</code>	<code>ulong_t</code>
<code>off_t</code>	<code>long</code>	<code>longlong_t</code>	<code>long</code>

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。**32** ビットオペレーティングシステムは **64** ビットハードウェア上で実行できますが、**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 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(4) 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。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ので、64 ビットプロセスを認識できないからです。

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、C コンパイラまたはアセンブラの `-xarch=v9` オプションを使用して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？

できます。ただし 64 ビットライブラリパッケージがインストールされている必要があります。また、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ることはできますか？

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、2000 年および 2038 年問題に関して安全です。

64 ビット libc(4) では sys_errlist[] と sys_nerr はどうなりましたか？

これらのシンボルは 64 ビットアプリケーションには見えません。ただし、32 ビットバイナリとの互換性を保つために、これらのシンボルは 32 ビットの libc(4) に残されています。

過去においては、公式に文書に記載されていない sys_errlist[] を使用して、アプリケーションはシステムが返した errno 値を印字可能な文字列に変換してきました。公式文書に記載されていないこのインターフェースを使用すると、互換性の問題が発生し、将来拡張された Solaris オペレーティングシステムを使用する際に制約が発生してしまいます。

このような問題を回避するには、32 ビットおよび 64 ビットアプリケーションの両方で、公式に記載されている標準 API の strerror(3C) を使用してください。また strerror(3C) には、自動的に現在のロケールの文字列を返すようにするという利点もあります。

malloc(3C) はなぜ 2G バイトを超える値を返さないのですか？

アプリケーションのデータセグメントのサイズを制御する資源のデフォルトのソフト制限値は、2G バイトです。アプリケーションがこれ以上の空間を割り当てる必要がある場合は、シェルを使ってこの制限値を大きくするか、あるいは制限値をなくすることができます。csh の場合は、次のコマンドを使用します。

```
% limit datasize unlimited
```

sh または ksh の場合は、次のコマンドを使用します。

```
$ ulimit -d unlimited
```

64 ビットの libc.a はどこにありますか？

64 ビットのアーカイブライブラリは提供されていません。

64 ビット Solaris オペレーティング環境を実行中の UltraSPARC 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。つまり、sparc と出力されません。

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古いライブラリである **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` はユーザーに役立つ情報を追加します。

64 ビット 開発ガイドの更新内容

『Solaris 7 64 ビット 開発ガイド』のこのバージョンでは、次の情報が追加または更新されました。

- 『SPARC Compliance Definition, Version 2.4』は、www.sparc.com にアクセスすれば、SPARC International から入手できます。57ページの「アプリケーションに関する新しい情報」を参照してください。
- 64 ビット版の `ls` に関する情報が付録 B に追加されました。
- コンパイラコマンドの参照情報が更新されました

Solaris 7 のリリース以降、Sun WorkShop™ Compilers C および添付される『C ユーザーズガイド』が更新されました。lint プログラムと C コンパイラに関するすべての参照指示が、正しいマニュアルページエントリ、つまり、`lint(1)` と `cc(1)` を参照するように更新されました。第 4 章 と 第 5 章 を参照してください。

索引

数字

64 ビット演算 20

64 ビットライブラリ 21

D

/dev/ksyms 62

E

ELF 61

E_OVERFLOW 65

G

GELF 61

I

ILP32

定義 30

ioctl(2) 65

isainfo(1) 25

isalist(1) 26

L

LD_LIBRARY_PATH 50

libkstat 63

libkvm 62

lint 36

LP64

定義 30

<inttypes.h> 32

<sys/types.h> 32

O

optisa(1) 27

\$ORIGIN 51

P

/proc 62

/proc

/proc の制約

定義 21

S

sizeof() 43

SPARC V9 ABI

特徴 58

stdio 64

U

uintptr_t 33

か

カーネルメモリーリーダー 20

こ

互換性 24

アプリケーションソースコード 24
アプリケーションバイナリ 24
デバイスドライバ 24
コンパイラ 49

そ

相互運用性の問題 20

た

大規模ファイル
定義 19
大容量仮想アドレス空間
定義 19

て

デバッグ処理 55

は

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 52

ひ

表記上の規則

ふ

符号の拡張 38
整数への昇格 39
変換 39

へ

ヘッダー 47

ほ

ポインタ計算 40

ら

ライブラリ 50
ラッパー
isaexec(3C) 55
/usr/lib/isaexec 54