システムインタフェース

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

Part No: 805-5856–10
1998 年 11 月

原典: *System Interface Guide*

Part No: 805–5141–10

Revision A

Adobe PostScript

目次

はじめに

目的

このマニュアルは、SunOS™ ライブラリが提供するシステムインタフェースについて記してあります。このマニュアルでは、プログラムの書き方ではなく、プログラムを動作させるために必要なその他の要素について重点的に説明しています。

対象読者と前提条件

このマニュアルは一般のプログラマを対象としています。システムソフトウェアの開発に従事しているような専門的なプログラマの方で、このマニュアルでは情報が不十分と思われる場合は、*Solaris 7 Reference Collection* または *Solaris 7* リファレンスマニュアル *Collection* を参照してください。

このマニュアルでは、端末の使用方法、UNIX システムのエディタ、UNIX システムのディレクトリとファイル構造についての知識を前提としています。これらの基礎知識については、『*OpenWindows* ユーザーズガイド』を参照してください。

C 言語との関係

SunOS システムは多くのプログラミング言語をサポートしていますが、特に C 言語とは、非常に密接な関係があります。

SunOS のコードの大部分は C 言語で書かれています。そのため、このマニュアルでは C 言語でのプログラミング例を多く取り上げていますが、他の言語を使用するプログラマの方にもこのマニュアルはご利用いただけます。

ハードウェアとソフトウェアの依存関係

アドレスなどのハードウェア固有の情報を除き、このマニュアルの大部分は、Solaris 7 オペレーティングシステムが動作するあらゆるコンピュータに適用できます。

ご使用中のシステム環境でコマンドの動作が異なる場合は、動作しているシステムのバージョンが異なっている可能性があります。また、コマンドが存在しないような場合は、そのコマンドが収められているパッケージがシステムにインストールされていない可能性があります。使用できるコマンドについては、システム管理者に確認してください。

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 *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、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

- 「x86」という用語は、一般に Intel 8086 ファミリ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。
- 「SunOS 5」という場合、本書では SunOS 5.0 から SunOS 5.7 までを意味します。

第 1 章

API の概要

Sun の目標の 1 つに、Solaris の Architectual Interfaces を明確にするということがあります。これには次の 2 つの理由があります。

- システムインタフェースは、顧客との実際上の「契約」です。Sun は顧客に対し、何を提供して使ってもらうかを正確に伝え、公開もしくは公開予定のインタフェースのみを使用できるような方策を提供します。
- この定義を使用して、この契約を尊重することを保証しています。製品の特定のバージョンで提供するインタフェースは、将来のバージョンでも維持されます。したがって、Solaris の今後のリリースでも上位互換性が保たれます。

プログラミングインタフェース

Solaris は、プログラミングインタフェース、ユーザインタフェースの各要素、プロトコル、ファイルシステム内のオブジェクトの命名規則や位置など、さまざまな「インタフェース」を提供しています。システムにとって最も重要なインタフェースの 1 つが、開発者に提供するプログラミングインタフェースです。プログラミングインタフェースは、次の 2 つの部分に分かれています。1 つは、アプリケーション開発者に関係する部分で、API (アプリケーションプログラミングインタフェース) と呼ばれます。もう 1 つは、デバイスドライバやプラットフォームサポートモジュールのようにシステムコンポーネントの開発者に関係するもので、SPI (システムプログラミングインタフェース) と呼ばれます。

開発者は Solaris の各プログラミングインタフェースを、ソースレベルとバイナリレベルという 2 つのレベルで「見る」ことができます。API や SPI という頭字語を使

用する場合は、システムのソースレベルのプログラミングインタフェースを示します。アプリケーションバイナリインタフェース (ABI)、システムバイナリインタフェース (SBI) という用語を使用して、それぞれのソースレベルのプログラミングインタフェースに対応するバイナリインタフェースを示します (「ABI」という用語は他のバイナリインタフェースと混同しがちなので、「Solaris ABI」という用語のみを使用します)。

インタフェース関数

このマニュアルで説明する SunOS 5 の関数は、カーネルとアプリケーションプログラムから提供されるサービス間のインタフェースです。*Solaris 7 Reference Collection* の『*man Pages(2): System Calls*』および『*man Pages(3): Library Routines*』に掲載されている関数は、SunOS 5 オペレーティングシステムとのアプリケーションのインタフェースです。これらの関数により、アプリケーションでファイルシステム、プロセス間通信のプリミティブ、マルチタスク機構などの機能を使用できます。この『システムインタフェース』は、API の重要部分を説明するマニュアルセットの中の 1 つです。その他に、このセットには『*STREAMS Programming Guide*』、『マルチスレッドのプログラミング』、『*Transport Interfaces Programming Guide*』などのマニュアルが含まれています。

Solaris 7 Reference Collection の『*man Pages(2): System Calls*』と『*man Pages(3): Library Routines*』に掲載されているライブラリルーチンを使用すると、プログラム作成時はその実装の詳細を意識する必要がなくなります。たとえば、標準 C ライブラリ内の `fread` 関数は `read` をベースに実装されています。

C プログラムは、プログラムのコンパイル時に呼び出す関数に自動的にリンクされます。この手順は、他の言語で作成されたプログラムで異なることがあります。詳細は、『リンカーとライブラリ』を参照してください。

ライブラリ

Solaris は、静的バージョンと動的バージョンのライブラリを提供しています。静的ライブラリはインタフェースを提供せず、関数の実装だけを提供します。開発者は、共用ライブラリ (共用オブジェクトともいいます) を通じて、Solaris のアプリケーションプログラミングインタフェースを利用できます。実行環境では、動的にリンクされた実行可能オブジェクトと共用オブジェクトが実行時リンカによって

処理され、実行可能プロセスが生成されます。システムとの公開 API は、アプリケーションと動的共用ライブラリ間のインタフェースです。

静的ライブラリ

ライブラリ (.a ファイルまたはアーカイブ) の従来の静的実装では、アプリケーションプログラミングインタフェースはその実装 (ライブラリの内容) に依存しています。アプリケーションを静的ライブラリにリンクすると、そのライブラリを実装するオブジェクトコードは構築された実行可能オブジェクトに組み込まれます。ライブラリとのソースレベルのプログラミングインタフェースを保持することもできますが、将来リリースされるオペレーティングシステムの新バージョンで動作する実行可能オブジェクトを生成するには、アプリケーションをリンクし直さなければなりません。将来のバイナリ互換性は、共用ライブラリを使用する場合にのみ保証されます。

静的ライブラリは歴史的な経緯があって残されているため、その実装から独立した形でインタフェースを定義するメカニズムはありません。このため、新しいアプリケーションでは静的ライブラリを使用しないようにしてください。

動的ライブラリ

静的ライブラリと違って、共用ライブラリではアプリケーションプログラミングインタフェースが実装に依存しません。インタフェースは、実行時にのみライブラリの実装にバインドされます。このため、Sun は API を管理し、それに対して構築されたアプリケーションとのバイナリ互換性を保ちつつ、内部インタフェースの変更など、ライブラリの実装を進めることができます。

インタフェースの分類

インタフェースは、そのインタフェースの使用者や使用方法によって次のように分類されて定義されます。

公開仕様	Sun が公開するインタフェース仕様で、顧客は無償で使用 (Sun によるインタフェースの実装を使用する製品を構築) できる。顧客以外は代替実装を自由に提供でき、ライセンスや法的制限は適用されない
非公開仕様	Sun が公開しないインタフェース仕様。Sun は、このインタフェース仕様に基づいて顧客が製品を構築したり、顧客以外が代替実装を構築することを望んでいない
互換性のある変更	インタフェースやその実装に対して、それまでの有効なプログラムに影響しない方法で行う変更
互換性のない変更	インタフェースの変更や、その実装に対して、それまでの有効なプログラムが無効になるような方法で行う変更。これには、バグの修正や性能の低下が含まれる。定義されていない「実装の成果」に依存するプログラムは含まない

標準クラス

仕様	公開
互換性のない変更	メジャーなリリース番号 (X.0)
例	POSIX、ANSI-C、Solaris ABI、SCD、SVID、XPG、X11、DKI、VMEbus、Ethernet

標準インタフェースとは、Sun 以外のグループによって仕様が管理されているインタフェースのことです。これには、POSIX、ANSI C などの規格や、X/Open、MIT X コンソーシアム、OMG などのグループによる業界仕様が含まれます。

公開クラス

仕様	公開
互換性のない変更	メジャーなリリース番号 (X.0)
例	Sun DDI、XView™、ToolTalk™、NFS™ プロトコル、Sbus、OBP

上記のインタフェースの仕様は、Sun が全面的に管理しています。これらのインタフェースの仕様書も公開していて、互換性を保つことを確約しています。

破棄されたクラス

仕様	なし
互換性のない変更	マイナーなリリース番号 (X.0)
例	RFS

一般に使用されなくなったインタフェースです。使用形態の変更を顧客に伝えるために、標準の移行用プログラムによって既存のインタフェースを他の状態 (公開や標準など) から「破棄」に格下げすることがあります。

使用形態を変更するには、インタフェースの差し替え意図を 1 年前から各顧客と Sun 製品開発コミュニティに通知する必要があります。現状のインタフェースと互換性のない変更を含む製品を出荷するまでに、1 年が経過しなければなりません。

顧客への通知手段には、サポート契約、リリースノートまたは製品マニュアルに関する問い合わせ、問題のインタフェースに該当する顧客フォーラムへの発表などが含まれます。

差し替え通知は、顧客が自由に入手できる「公開」情報と見なされます。プレス発表や類似の広報など、特定の処置によって情報を「公開」する予定はありません。

第 2 章

Java プログラミング

Java とは

Java™ は、次の特徴を持つ最近開発されたコンカレントなクラスをベースとしたオブジェクト指向のプログラミング言語です。

- 簡単。C++ 言語に似ていますが、C および C++ 言語のより複雑な機能の大部分は取り除かれています。Java は、以下の機能は提供していません。
  - プログラマが制御する動的メモリ
  - ポインタ演算
  - 構造体
  - Typedefs
  - #define
- オブジェクト指向。Java は、C++ 言語の基本オブジェクト技術を提供していますが、一部は強化し、一部は削除しています。
- アーキテクチャとして独立。Java のソースコードは、アーキテクチャに依存しないオブジェクトコードにコンパイルされます。オブジェクトコードは、Java 実行時システムによって解釈されます。
- 移植性。Java は、最新の移植性標準を実装しています。たとえば、`int` は常に 32 ビットの 2 の補数となる整数です。ユーザインタフェースは、Solaris およびその他のオペレーティング環境に容易に実装される抽象化されたウィンドウシステムを通して構築されます。

- 分散型。Java には広範な TCP/IP ネットワーキング機能があります。ライブラリルーチンは、ハイパーテキスト転送プロトコル (HTTP) やファイル転送プロトコル (FTP) などのプロトコルをサポートしています。
- 堅牢。Java コンパイラと Java インタプリタの両方とも、広範なエラーチェックをサポートしています。Java がすべての動的メモリを管理し、配列境界をチェックし、その他の例外をチェックします。
- 安全。不当なメモリアクセスとなることも多い C および C++ 言語の機能は、Java 言語にはありません。インタプリタは、コンパイルされたコードにもいくつかのテストを実施し、不当なコードがないかチェックします。これらのテスト後、コンパイルされたコードは、オペランドスタックのオーバーフローやアンダーフローを発生させず、不当なデータを変換しないで、正当なオブジェクトフィールドのアクセスだけを実行します。さらに、すべてのオペコードパラメタの型は正当です。
- 高性能。アーキテクチャに依存しないマシン語に似た言語にプログラムをコンパイルするため、Java プログラムのインタプリタは小さく効率的になります。将来的には Java 環境は、実行時に Java バイトコードをネイティブなマシンコードにアセンブルもします。
- マルチスレッド化。マルチスレッドは、Java 言語の中に組み込まれています。ユーザの処置を処理している間にイメージのロードなどを操作できるようにし、対話型処理の性能を向上させることができます。
- 動的。Java は、実行時まで呼び出されるモジュールをリンクしません。

Java プログラミング環境

Java のプログラミングは、テキストエディタ、make(1S)、および以下のものによって Solaris Java Virtual Machine (以降 Solaris Java VM とします) でサポートされています。

`javac`	Java コンパイラ。Java ソースコードファイル (*name*`.java`) インタプリタ (`java(1)`) が処理できるバイトコードファイル (*name*`.class`) に翻訳する。Java アプリケーションと Java アプレットの両方がコンパイルされる。
`javald`	ラッパージェネレータ。Java アプリケーションのコンパイルと実行に必要な環境を取り込むラッパーを作成する。ラッパーが呼び出されるまで指定されたパスが結合されないので、ラッパーは `JAVA_HOME` および `CLASSPATH` パスの再配置を考慮している。
`java`	Java インタプリタ。コマンドとして呼び出して Java アプリケーションを実行したり、HTML コードによってブラウザから呼び出してアプレットを実行したりできる。
`appletviewer`	Java アプレットビューワ。このコマンドは、指定されたドキュメントまたは資源を表示し、ドキュメントが参照している各アプレットを実行する。
`javap`	Java クラスファイルの逆アセンブラ。`javac` によってコンパイルされたバイトコードのクラスファイルを逆アセンブルし、結果を標準出力に出力する。
`javah`	C ヘッダおよびスタブファイルジェネレータ。指定されたクラスごとに、*classname*`.h` という名前のヘッダファイルを作成してカレントディレクトリに入れる。またオプションで、C ソーススタブファイルを生成する。

`make(1S)` の使用方法については、『プログラミングユーティリティ』を参照してください。

通常の Java 環境変数は、次のとおりです。

変数	説明
JAVA_HOME	Java ソフトウェアのベースディレクトリのパス。たとえば、javac、java、appletviewer、javap、javah は、すべて $JAVA_HOME/bin にある。Solaris Java VM を使用するために設定する必要はない。
CLASSPATH	アプリケーションおよびアプレットで使用するためのコンパイル済み *.class ファイルを持つディレクトリへのパスがコロン (:) で区切られたリスト。javac、java、javap、javah によって使用される。設定しないと、すべての Solaris Java VM 実行可能ファイルは、/usr/java/lib/classes.zip をデフォルトとする。Solaris Java VM を使用するために設定する必要はない。
PATH	通常の実行可能ファイル検索リストに、$JAVA_HOME/bin を入れることができる。

注 - Java VM ツールは /usr/java/bin にインストールされ、各実行可能ファイルへのシンボリックリンクは、/usr/bin に格納されます。これは、新たにインストールされる Java VM パッケージを使用するためにユーザの PATH 変数に何も追加する必要がないことを意味します。また、すべての Java VM 実行可能ファイルは、パス /usr/java/lib/classes.zip をデフォルトとして標準 Java クラスライブラリを見つけます。

ベース Java プログラミング環境は、デバッガを提供していません。デバッガは、別売りの Java WorkShop™ パッケージに含まれています。

Java プログラム

Java プログラムは、アプリケーションとアプレットという 2 つの形式で作成されます。

Java アプリケーションを実行するには、コマンド行から Java インタプリタを呼び出し、コンパイル済みアプリケーションを持つファイルを指定します。

Java アプレットは、ブラウザから呼び出します。ブラウザによって解釈される HTML コードは、コンパイル済みアプレットを持つファイル名を指定します。こうすると、ブラウザは Java インタプリタを呼び出し、インタプリタがアプレットをロードして実行します。

アプリケーション

コード例 2–1 は、“Hello World” を標準出力に表示するだけのアプリケーションのソースです。メソッドは、呼び出し時に引数を受け入れますが、それらを使用して何も行いません。

コード例 2–1 Java アプリケーション

```
//
// HelloWorld Application
//
class HelloWorldApp{
 public static void main (String args[]) {
  System.out.println (``Hello World'');
 }
}
```

C と同様に最初に実行されるメソッド、すなわち関数は `main` として識別されます。キーワード `public` によって、メソッドは誰でも実行できます。`static` は `main` に `HelloWorldApp` クラスを参照させ、クラスの他のどのインスタンスも参照させません。`void` は `main` が何も戻さないことを示し、`args[]` は `String` 型の配列を宣言しています。

アプリケーションは、次のようにコンパイルします。

```
$ javac HelloWorldApp.java
```

また、次のように実行します。

```
$ java HelloWorldApp arg1 arg2 ...
```

アプレット

コード例 2–2 は、コード例 2–1 のアプリケーションと同等のアプレットのソースです。

コード例 2–2 Java アプレット

```
//
// HelloWorld Applet
//
import java.awt.Graphics;
import java.applet.Applet;

public class HelloWorld extends Applet {
 public void paint (Graphics g) {
  g.drawstring (``Hello World'', 25, 25);
 }
```

```
}
```

アプレットでは、参照されているすべてのクラスを明示的に取り込ま (import) なければなりません。キーワード public と void は、アプリケーションの場合と同じことを意味しています。extends は、HelloWorld クラスが Applet クラスから継承することを示しています。

アプレットは、次のようにコンパイルします。

```
$ javac HelloWorld.java
```

アプレットは、HTML コードによってブラウザで呼び出されます。アプレットを実行するための最小限の HTML ページは次のとおりです。

```
<title>Test</title>
<hr>
<applet code=''HelloWorld.class'' width=100 height=50>
</applet>
<hr>
```

javald と再配置可能アプリケーション

Java アプリケーションを正しく実行するには、JAVA_HOME、CLASSPATH、および LD_LIBRARY_PATH 環境変数を正しく設定する必要があります。これらの環境変数の値は、各ユーザによって制御されるため、パスが通常とは異なるように任意のパスに設定できます。また、通常、アプリケーションは CLASSPATH 変数に独自の値が必要です。

javald(1) は、Java アプリケーションのためのラッパーを生成するコマンドです。ラッパーは、JAVA_HOME、CLASSPATH、および LD_LIBRARY_PATH 環境変数のいずれか、またはすべての正しいパスを指定できます。パスを指定しても、これらの環境変数のユーザの値には影響しません。Java アプリケーションの実行中は、これらの環境変数のユーザの値を無効にします。さらにラッパーは、Java アプリケーションが実際に実行されるまで指定されたパスが結合されないことを保証するので、アプリケーションの再配置性が最大になります。

Solaris 上での Java スレッド

Java プログラミング言語は、マルチスレッド化されたプログラムのサポートを想定しています。すべての Java インタプリタは、マルチスレッド化されたプログラミング環境を提供しています。しかし、これらのインタプリタの多くは、マルチスレッ

ドの単一プロセッサ形式しかサポートしていません。そのため、マルチプロセッサ上で実行する従来の Java インタプリタでの Java プログラムのスレッドは、完全に並行には実行されず、実際には一度に 1 つのスレッドしか実行していません。

Solaris Java VM インタプリタは、複数プロセッサのコンピューティングシステムをすべて利用しています。このインタプリタは、単一プロセスの複数スレッドを複数の CPU に同時にスケジュールできるイントリンシクス Solaris マルチスレッド機能を使用します。このため、マルチスレッド化された Java プログラムは、Solaris Java VM の元で実行すると、その並行性の程度が著しく向上します。

図 2–1 は、Java スレッドが Solaris Java VM の元でどのように動作するかを示しています。Solaris スレッドの動作については、『マルチスレッドのプログラミング』を参照してください。

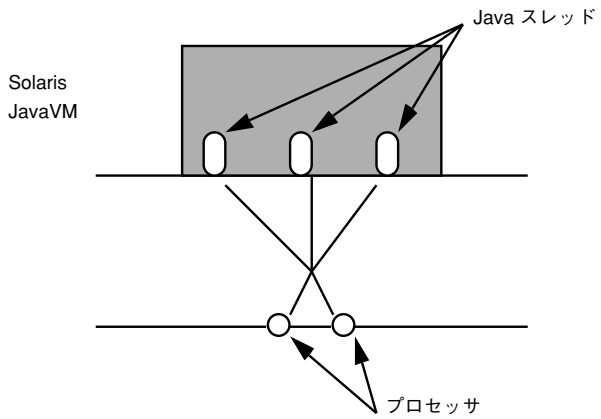

図 2–1　Solaris 上での Java スレッド

マルチスレッド化されたアプリケーションのチューニング

並行化されたマトリックス乗算などの計算のアプリケーションなどで、Solaris スレッドを十分に活用するには、Solaris のネイティブ関数 `thr_setconcurrency(3T)` を使用します。これにより Solaris 本来のマルチスレッド機能が Java アプリケーションで使用でき、複数プロセッサを完全に利用することが保証されます。これは、ほとんどの Java アプリケーションおよびアプレットには必要ありません。次のコードは、これをどのように行うかの例です。

最初の要素は、MPtest_NativeTSetconc() を使用する Java アプリケーション MPtest.java です。このアプリケーションは 10 個のスレッドを生成し、各スレッドは識別行を表示してから 10,000,000 回ループし、計算の多い活動のシミュレーションを行います。

コード例 **2–3**　MPtest.java

```
import java.applet.*;
import java.io.PrintStream;
import java.io.*;
import java.net.*;

class MPtest {
   static native void NativeTSetconc();

   static public int        THREAD_COUNT = 10;

   public static void main (String args[]) {
      int i;
      //
 set concurrency on Solaris - sets it to sysconf(_SC_NPROCESSORS_ONLN);
      NativeTSetconc();
      // start threads
      client_thread clients[] = new client_thread[ THREAD_COUNT ];
      for ( i = 0; i < THREAD_COUNT; ++i ){
         clients[i] = new client_thread(i, System.out);
         clients[i].start();
      }
   }

   static { System.loadLibrary(``NativeThreads''); }
}

class client_thread extends Thread {
   PrintStream out;
   public int        LOOP_COUNT   = 10000000;
   client_thread(int num, PrintStream out){
      super( ``Client Thread'' + Integer.toString( num ) );
      this.out = out;
      out.println(``Thread `` + num);
   }
   public void run () {
      for( int i = 0; i < this.LOOP_COUNT  ; ++i; ) {
    ;
     }
   }
}
```

2 番目の要素は、javah(1) ユーティリティによって MPtest.java から生成される C スタブファイル MPtest.c です。これを行うには、次のように入力します。

```
% javah -stubs MPtest.java
```

3 番目の要素は、同様に javah(1) ユーティリティによって MPtest.java から生成される C ヘッダファイル MPtest.h です。これを行うには、次のように入力します。

```
% javah MPtest.java
```

4 番目の要素は、C ライブラリインタフェースの呼び出しを実行する C 関数 NativeThreads.c です。

コード例 **2–4**　NativeThreads.c

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

void MPtest_NativeTSetconc(void *this) {
  thr_setconcurrency(sysconf(_SC_NPROCESSORS_ONLN));
}
```

最後に、4 つのファイルを Java アプリケーション MPtest.class に結合するには、次のような makefile を使用して非常に簡単に実行できます。

コード例 **2–5**　MPtest の makefile

```
# make は、2 つのステージで行わなければならない。
# まず、``make MPtest'' を実行する。
# 次に、NativeThreads.c を作成してネイティブ関数
# ``thr_setconcurrency(sysconf(_SC_NPROCESSORS_ONLN))
# の呼び出しを組み込んでから ``make lib'' を実行する。
# その後、LD_LIBRARY_PATH および CLASSPATH を ``.'' に
# 設定して ``java MPtest'' を実行できる。

JH_INC1=/usr/java/include
JH_INC2=/usr/java/include/solaris
CLASSPATH=.; export CLASSPATH

MPtest:
 javac MPtest.java
 (CLASSPATH=.; export CLASSPATH; javah MPtest)
 (CLASSPATH=.; export CLASSPATH; javah -stubs MPtest)
 cc -G -I${JH_INC1} -I${JH_INC2} MPtest.c NativeThreads.c \
  -o libNativeThreads.so
clean:
 rm -rf *.class MPtest.c MPtest.o libNativeThreads.so \
  NativeThreads.o *.h
```

Java WorkShop とは

Java WorkShop (JWS) は、SunSoft DevPro の別パッケージ製品です。JWS は Java で実装されており、独自の Java インタプリタを使用します。JWS は、Portfolio Manager、Project Manager、Source Editor、Build Manager、Source Browser、Debugger、Project Tester、オンラインヘルプ、および Visual Java の 9 個のアプリケーションから構成されています。

Portfolio Manager	Java プロジェクトのポートフォリオの作成とカスタマイズを行う。このアプリケーションは、新しいアプレットとアプリケーションを作成する元となるオブジェクトとアプレットの集合を管理する。
Project Manager	プロジェクトの設定を変更し、ディレクトリを設定する。コンポーネントへのパスの位置と設定の編成と保存を行う。
Source Editor	ソースコードの作成と編集を行うためのポイント&クリックツール。Java WorkShop のコンポーネントは、プロセスの作成、コンパイル、およびデバッグで頻繁に Source Editor を呼び出す。
Build Manager	Java ソースコードを Java バイトコードにコンパイルし、ソースにあるエラーを見つける。Source Editor を起動すると、Build Manager はソースコードの訂正箇所にリンクするので、訂正やコンパイルを非常に素早く実行できる。
Source Browser	プロジェクトにあるすべてのオブジェクトのクラス継承を示すツリーダイアグラムを表示する。また、プロジェクトにあるすべてのコンストラクタと通常のメソッドをリストし、文字列とシンボルを検索できるようにする。Source Browser は、Source Editor にリンクしてコードを表示する。
Debugger	デバッグプロセスの制御と管理を行うためのツール群を提供する。コントロールパネルの元でアプリケーションまたはアプレットを実行すると、スレッドの停止と再開、ブレークポイントの設定、例外のトラップ、スレッドのアルファベット順表示、およびメッセージの参照を行うことができる。
Project Tester	アプレットビューワと同様に、Project Tester を使用して、アプレットの実行とテストを行うことができる。Build Manager を使用してアプレットをコンパイルしてから、Project Tester を使用して実行する。

オンラインヘルプ	「目次」、「索引」、「チュートリアル」、「プロジェクトの管理」、「ソースコードの編集」、「プロジェクトの構築」「Visual Java」、「ソースの表示」、および「プロジェクトのデバッグ」で構成されている。
Visual Java	カスタマイズ可能なプレビルト GUI 画像ウィジェットのパレットがあるポイント&クリックインタフェースを持つ統合 Java GUI ビルダ

第 3 章

プロセス

この章では、プロセスとそれらを操作するライブラリ関数を説明します。

概要

コマンドを実行すると、オペレーティングシステムによって番号が付けられ、追跡されるプロセスが開始されます。オペレーティングシステムには、あるプロセスが常に他のプロセスによって生成されるという柔軟な機能があります。たとえば、システムにログインしてシェルから vi(1) などのエディタを使用します。次に、vi(1) からシェルを起動します。その後 ps(1) コマンドを実行すると、次のように表示されます (ps -f コマンドの実行結果を示します)。

UID	PID	PPID	C	STIME	TTY	TIME	COMD
abc	24210	1	0	06:13:14	tty29	0:05	sh
abc	24631	24210	0	06:59:07	tty29	0:13	vi c2
abc	28441	28358	80	09:17:22	tty29	0:01	ps −f
abc	28358	24631	2	09:15:14	tty29	0:01	sh −i

この例では、ユーザ abc は 4 つのプロセスを起動しています。プロセス ID (PID) と親プロセス ID (PPID) のカラムは、ユーザ abc がログオンしたときに起動された

シェルがプロセス 24210 であり、その親が初期化プロセス (プロセス ID は 1) であることを示しています。プロセス 24210 はプロセス 24631 の親プロセスで、以下も同様です。

プログラムは、その実行状態によっては他の 1 つ以上のプログラムの実行が必要な場合があります。1 つの実行可能プログラムのサイズを大きくするのは実用的ではありません。その理由は次のとおりです。

- 2 つ以上のモジュールを並行に実行させる必要がある場合がある
- ロードモジュールが大きくなりすぎて、システムの最大プロセスサイズを超えてしまう
- 組み込みたい他のモジュールのオブジェクトコードをすべて使用するとは限らない

`fork(2)` 関数と `exec(2)` 関数を使用すると、新しいプロセス (生成しようとするプロセスのコピー) を生成し、実行中のプログラムの代わりに新しい実行可能プログラムを起動できます。

関数

表 3–1 の関数は、ユーザプロセスの制御に使用します。

表 3–1 プロセス関数

関数名	目的
`fork(2)`	新しいプロセスを生成する。
`exec(2)`	プログラムを実行する。
`execl(2)`	
`execv(2)`	
`execle(2)`	
`execve(2)`	
`execlp(2)`	
`execvp(2)`	

表 3–1 プロセス関数 続く

関数名	目的
exit(2)	プロセスを終了する。
_exit(2)	
wait(2)	子プロセスが停止または終了するのを待つ。
dladdr(3X)	アドレスをシンボルの情報に変換する。
dlclose(3X)	共用オブジェクトを閉じる。
dlerror(3X)	診断情報を取得する。
dlopen(3X)	共用オブジェクトを開く。
dlsym(3X)	共用オブジェクト内のシンボルのアドレスを取得する。
setuid(2)	ユーザ ID とグループ ID を設定する。
setgid(2)	
setpgrp(2)	プロセスグループ ID を設定する。
chdir(2)	作業用ディレクトリを変更する。
fchdir(2)	
chroot(2)	ルートディレクトリを変更する。
nice(2)	プロセスの優先順位を変更する。
getcontext(2)	現在のユーザコンテキストを取得または設定する。
setcontext(2)	
getgroups(2)	補助グループ ID のリストを取得または設定する。
setgroups(2)	

表 3–1　プロセス関数　続く

関数名	目的
getpid(2)	プロセス ID、プロセスグループ ID、および親プロセス ID を取得する。
getpgrp(2)	
getppid(2)	
getpgid(2)	
getuid(2)	実ユーザ ID、実効ユーザ ID、実グループ ID、および実効グループ ID を取得する。
geteuid(2)	
getgid(2)	
getegid(2)	
pause(2)	シグナルを受信するまでプロセスを一時停止する。
priocntl(2)	プロセススケジューラを制御する。
setpgid(2)	プロセスグループ ID を設定する。
setsid(2)	セッション ID を設定する。
waitid(2)	子プロセスの状態が変化するまで待つ。

新しいプロセスの生成

fork(2)

fork(2) を呼び出すと、呼び出しプロセスを正確にコピーして新しいプロセスを生成します。この新しいプロセスを子プロセスといい、呼び出し側を親プロセスといいます。子プロセスは、新しい固有のプロセス ID を取得します。fork(2) は、正常終了すると子プロセスに 0 を戻し、親プロセスに子のプロセス ID を戻します。戻り値によって、それが親プロセスか子プロセスかがわかります。

fork(2) 関数または exec(2) 関数によって生成される新しいプロセスは、3 つの標準ファイル stdin、stdout、stderr を含めた開いているすべてのファイル記述子を親から継承します。親プロセスが、子プロセスの出力よりも先に表示しなければならない出力をバッファリングしている場合、fork(2) の前にバッファをフラッシュしなければなりません。

次のコード例は、fork および後続のアクションを呼び出します。

```
pid_t  pid;

  pid = fork;
  switch (pid) {
   case -1:   /* fork が失敗した */
    perror (``fork'');
    exit (1);
   case 0:   /* 新しい子プロセスで */
    printf (``In child, my pid is: %d\n'', getpid(); );
    do_child_stuff();
    exit (0);
   default:   /* 親では、pid は子の PID を持つ */
    printf (``In parent, my pid is %d, my child is %d\n'', getpid(), pid);
    break;
  }

  /* 親プロセスコード */
  ...
```

また、親プロセスと子プロセスの両方がストリームから入力を読み込む場合、一方のプロセスが読み込んだ情報を、もう一方のプロセスは読み込むことができません。これは、入力バッファから、あるプロセスに情報が送られると、読み込みポインタが移動してしまうからです。

注 - 従来は、fork(2) と exec(2) を使用して別の実行プロセスを起動し、新しいプロセスが終了するまで待つという方法が使われていました。実際には、第 2 のプロセスはサブルーチンを呼び出すために作成されます。サブルーチンを一時的にメモリに常駐させるには、24ページの「実行時リンク」で説明するように、dlopen(3X)、dlsym(3X)、dlclose(3X) を使用する方が効率的です。

---

exec(2)

exec(2)
は、exec1(2)、execV(2)、execle(2)、execve(2)、exec(2)、execvp(2) を含む関数ファミリ名です。これらの関数はいずれも、呼び出しプロセスを新しいプロセスで置き換えますが、引数のまとめ方と表し方が異なります。たとえば、exec1(2) は次のように使用できます。

```
execl("/usr/bin/prog2", ``prog2'', progarg1, progarg2, (char (*)0));
```

exec1(2) 引数リストは、次のとおりです。

/usr/bin/prog2	新しいプロセスファイルのパス名
prog2	新しいプロセスが argv[0]に取り込む名前
progarg1 progarg2	prog2 への char (*) 型の引数
(char (*)0)	引数の終わりを示す NULL の char ポインタ

exec(2) が正常に実行されるといかなる時でも、復帰はありません。新しいプロセスが exec(2) を呼び出したプロセスを上書きしてしまうためです。新しいプロセスは、古いプロセスのプロセス ID やその他の属性を引き継ぎます。exec(2) の呼び出しが失敗すると、制御は呼び出しプログラムに戻され、戻り値 –1 が返されます。errno をチェックすれば、失敗した理由がわかります。

実行時リンク

アプリケーションは、実行中に他の共用オブジェクトにバインドして、アドレス空間を拡張できます。このような共用オブジェクトの遅延バインディングには、次のような長所があります。

- アプリケーションの初期化中ではなく、必要なときに共用オブジェクトを処理すると、起動時間を大幅に短縮できる。また、ヘルプやデバッグ情報を含むオブジェクトなどのアプリケーションを実行する場合には、共用オブジェクトが不要になることもある。
- アプリケーションでは、ネットワークプロトコルなど、必要なサービスだけを多数の異なる共用オブジェクトから選択できる。
- 実行中にプロセスのアドレス空間に追加された共用オブジェクトを使用後に解放できる。

  アプリケーションが共用オブジェクトにアクセスする場合の典型的な例は次のとおりです。

- dlopen(3X) を使用して共用オブジェクトを配置し、実行中のアプリケーションのアドレス空間に追加する。共有オブジェクトの依存関係も、この時点で見つけて追加する。たとえば、次のようになる。

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

   main(int argc, char ** argv)
    {
    void *  handle;
    .....
    if ((handle = dlopen(``foo.so.1'', RTLD_LAZY)) == NULL) {
     (void) printf(``dlopen: %s\n'', dlerror());
     exit (1);
    }
    .....
```

- 追加した共用オブジェクト (1 つまたは複数) は再配置され、新しい共用オブジェクト (1 つまたは複数) 内の初期化セクションが呼び出される。

- アプリケーションは dlsym(3X) を使用して、追加された共用オブジェクト (1 つまたは複数) 内でシンボルの位置を見つける。これにより、アプリケーションはこの新しいシンボルで定義されたデータを参照したり、関数を呼び出したりできる。上記の例の続きは次のようになる。

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

- アプリケーションが共用オブジェクト (1 つまたは複数) の使用を終了すると、dlclose(3X) を使用してアドレス空間が解放される。解放される共用オブジェクト (1 つまたは複数) 内の終了セクションが、この時点で呼び出される。たとえば、次のようになる。

```
if (dlcose (handle) != 0) {
    (void) printf(``dlclose: %s\n'', dlerror());
     exit (1);
   }
```

- これらの実行時リンカインタフェースルーチンを使用した結果として発生するエラー状態は、dlerror(3X) を使用して表示できる。

  実行時リンカのサービスは、ヘッダファイル <dlfcn.h> 内で定義されていて、共用ライブラリ libdl.so.1 を介してアプリケーションで利用できます。たとえば、次のようになります。

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

この場合、ファイル `main.c` は `dlopen(3X)` ファミリの任意のルーチンを参照でき、アプリケーション `prog` は実行時にこれらのルーチンにバインドされます。

アプリケーションが指定する実行時リンクの詳細は、『リンカーとライブラリ』の第 3 章を参照してください。使用方法については、`dladdr(3X)`、`dlclose(3X)`、`dlerror(3X)`、`dlopen(3X)`、`dlsym(3X)` の各マニュアルページを参照してください。

プロセスのスケジューリング

UNIX システムのスケジューラは、プロセスをいつ実行するかを決定します。このスケジューラは、構成パラメタ、プロセスの動作、およびユーザの要求に基づいてプロセスの優先順位を管理し、これらの優先順位を使用して CPU にプロセスが割り当てられます。

スケジューラ関数を使用すると、特定のプロセスの実行順序と、各プロセスが他のプロセスに CPU を明け渡すまでに CPU を使用できる時間をすべてユーザが制御できます。

デフォルトでは、スケジューラはタイムシェアリング方式を使用します。タイムシェアリング方式では、プロセスの優先順位を動的に調整して、対話型プロセスには適切な応答性能、CPU を多く使用するプロセスには良いスループットを提供します。

スケジューラは、実時間スケジューリング方式も提供します。実時間スケジューリングにより、ユーザはプロセスごとに優先順位を固定できます。固定優先順位とは、システムによって変更されない優先順位のことです。最も優先順位の高い実時間ユーザプロセスは、他のシステムプロセスが実行可能な状態になっても、実行可能になりしだい常に CPU を使用できます。したがって、プログラムでプロセスの正確な実行順序を指定できます。また、実時間プロセスの応答時間がシステムによって保証されるようなプログラムも作成できます。

ほとんどの SunOS 5 のバージョンでは、デフォルトのスケジューラの構成で十分に機能するため、実時間プロセスは必要ありません。管理者が構成パラメタを変更したり、ユーザが各自のプロセスの優先順位を変更したりする必要はありません。ただし、タイミングの制約が厳しいプログラムでは、実時間プロセスでなければ制約を満たせないことがあります。

詳細は、`priocntl(1)`、`priocntl(2)`、`dispadmin(1M)` の各マニュアルページを参照してください。このテーマの詳細な説明は、第 4 章を参照してください。

エラー処理

関数が正常に終了しなかった場合は、いくつかの例外を除いて、常に -1 の値がプログラムに戻されます (『*man Pages(2): System Calls*』で説明している関数では、戻り値が定義されていないものもありますが、これは例外です)。この場合、プログラムに -1 が戻されるだけでなく、外部宣言されている変数の errno に整数値が設定されます。C プログラムでは、次の文をプログラムに入れておけば errno の値を判定できます。

```
#include <errno.h>
```

関数が正常終了すると、errno の値はクリアされません。したがって、この値を検査するのは、関数が -1 を戻した場合だけにしてください。一部の関数は -1 を戻しますが、errno を設定しません。errno が有効な値を持つことが確かである関数については、マニュアルページを参照してください。エラーについては、`Intro(2)` のマニュアルページを参照してください。

C 言語の `perror(3C)` 関数を使用すれば、errno の値に対応するエラーメッセージを stderr に出力でき、`strerror(3C)` 関数を使用すれば、対応する印字可能文字列を取得できます。

第 4 章

プロセススケジューラ

この章では、プロセスのスケジューリングについて説明します。マルチスレッド化されたスケジューリングについては、『マルチスレッドのプログラミング』を参照してください。この章は、プロセスの実行順序についてデフォルトのスケジューリングが提供する以上の制御を行う必要があるプログラマを対象としています。

スケジューラの概要

プロセスは、生成されると 1 つの軽量プロセス (LWP) を割り当てられます。(マルチスレッド化されているプロセスは、さらに多くの LWP を割り当てられることがあります。) LWP は、UNIX システムのスケジューラが実際にスケジューリングするオブジェクトです。スケジューラは、プロセスが実行する時期を決定し、構成パラメタ、プロセスの特性、およびユーザ要求に基づいてプロセス優先順位を維持管理します。スケジューラは、これらの優先順位を使用してプロセスを実行します。

デフォルトでは、タイムシェアリング方式を使用します。この方式は、プロセスの優先順位を動的に調整して、対話型プロセスの応答時間と CPU 時間を多く使用するプロセスのスループットとを調整します。

SunOS 5 のスケジューラでは、実時間スケジューリング方式も使用できます。実時間スケジューリング方式では、ユーザがプロセスごとに優先順位を固定できます。最も優先順位の高い実時間ユーザプロセスは、システムプロセスが実行できる場合でも、そのプロセスが実行可能なときはいつでも CPU を利用できます。

実時間プロセスがシステムからの応答時間を保証されるように、プログラムを作成できます。詳細は、第 9 章を参照してください。

実時間スケジューリングによるプロセススケジューリングの制御が必要なことはほとんどなく、それが解決するよりも多くの問題をはらんでいます。しかし、プログラムの要件に厳しいタイミングの制約が含まれるときは、実時間プロセスがそれらの制約を満たす唯一の方法となる場合があります。

注 - 実時間プロセスを不用意に使用すると、タイムシェアリングプロセスの性能が極めて悪くなることがあります。

スケジューラ管理を変更すると、スケジューラの特性に影響を与える可能性があるため、プログラマもスケジューラ管理について多少理解しておく必要があります。スケジューラ管理に関連するマニュアルページは次のとおりです。

- `dispadmin(1M)` では、実行中のシステムでスケジューラの構成を変更する方法を説明しています。
- `ts_dptbl` と `rt_dptbl` では、スケジューラの構成に使用するタイムシェアリングと実時間のパラメタテーブルを説明しています。

図 4–1 に SunOS 5 のスケジューラの働きを示します。

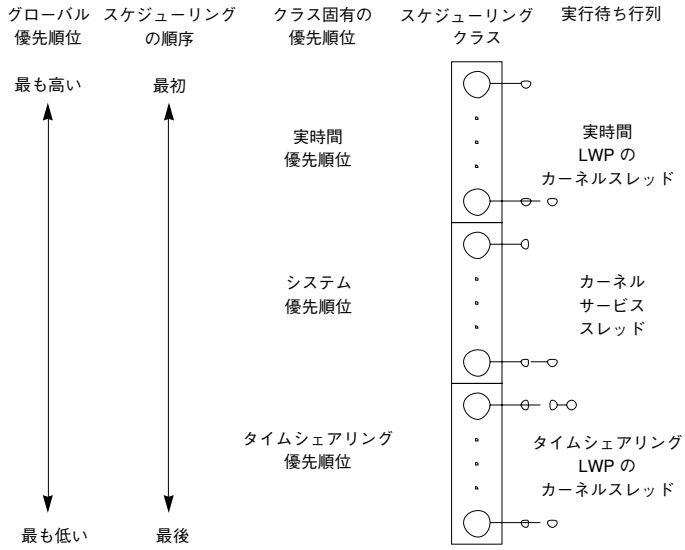

図 4–1　SunOS 5 プロセスのスケジューラ

作成されたプロセスは、スケジューリングクラスやそのクラス内の優先順位を含むスケジューリングパラメタを継承します。プロセスは、ユーザ要求によってだけクラスを変更します。システムは、ユーザの要求とそのプロセスのスケジューリングクラスに対応する方針に基づいて、プロセスの優先順位を管理します。

デフォルトの設定では、初期化プロセスはタイムシェアリングクラスに属します。そのため、すべてのユーザログインシェルは、タイムシェアリングプロセスとして開始します。

スケジューラは、クラス固有優先順位をグローバル優先順位に変換します。プロセスのグローバル優先順位は、そのプロセスの実行時期を決定します。つまり、スケジューラは常に、グローバル優先順位が最も高い実行可能なプロセスを実行します。優先順位の数値が大きいプロセスが先に実行されます。スケジューラがプロセスを CPU に割り当てると、プロセスは休眠するか、そのタイムスライスを使い切るか、または優先順位がさらに高いプロセスによって横取りされるまで実行されます。優先順位が同じプロセスは、優先順位を競い合い実行されます。

実時間プロセスは、どのカーネルプロセスよりも優先順位が高く、カーネルプロセスは、どのタイムシェアリングプロセスよりも優先順位が高くなっています。

注 - 実行可能な実時間プロセスがある限り、カーネルプロセスやタイムシェアリングプロセスは実行されません。

管理者は構成テーブルでデフォルトのタイムスライスを指定しますが、ユーザは実時間プロセスにプロセスごとのタイムスライスを割り当てることができます。

プロセスのグローバル優先順位は、`ps(1)` コマンドの `–cl` オプションで表示できます。クラス固有の優先順位についての設定内容は、`priocntl` コマンドと `dispadmin(1M)` コマンドで表示できます。

以降の節では、3 つのデフォルトクラスのスケジューリング方式について説明します。

タイムシェアリングクラス

タイムシェアリング方式の目的は、対話型プロセスには良い応答性能、CPU 時間を多く使用するプロセスには良いスループットを提供することです。スケジューラは、切り替え自体に時間がかかりすぎない頻度で CPU の割り当てを切り替え、応答性能を良くします。タイムスライスは通常、数百ミリ秒です。

タイムシェアリング方式は、優先順位がダイナミックに変更され、割り当てられるタイムスライスの長さは異なります。スケジューラは、CPU をほんのわずかだけ使用して休眠しているプロセスの優先順位を上げます (プロセスは、端末からの読み取りやディスク読み取りなどの入出力操作を開始すると休眠します) 。頻繁に休眠する

のは、編集や簡単なシェルコマンドの実行など、対話型タスクの特性です。一方、休眠しないで CPU を長時間使用しているプロセスの優先順位は下げられます。

デフォルトのタイムシェアリング方式では、優先順位が低いプロセスに長いタイムスライスが与えられます。優先順位が低いプロセスは、CPU を長時間使用する傾向があるからです。他のプロセスが CPU を先に取得しても、優先順位の低いプロセスが CPU を取得すれば長時間使用できます。ただし、タイムスライス中に優先順位が高いプロセスが実行可能になると、そのプロセスが CPU を横取りします。

グローバルプロセス優先順位とユーザ指定優先順位は、昇順になります。つまり、優先順位が低い数値のプロセスが最初に実行されます。ユーザ優先順位は、設定されている値の、負の最大値から正の最大値までの値になります。プロセスはユーザ優先順位を継承します。ユーザ優先順位のデフォルト初期値は 0 (ゼロ) です。

「ユーザ優先順位限界」は、設定されているユーザ優先順位の最大値です。ユーザ優先は、この限界値以下の任意の値に設定できます。適当なアクセス権を持っていると、ユーザ優先順位限界を上げることができます。ユーザ優先順位限界のデフォルト値は 0 (ゼロ) です。

プロセスのユーザ優先順位を下げて、CPU の使用率を減少させたり、適当なアクセス権を持っている場合は、ユーザ優先順位を上げてサービスを受けやすくしたりできます。ユーザ優先順位はユーザ優先順位限界より大きく設定できないので、この値がどちらもデフォルト値の 0 になっている場合は、ユーザ優先順位を上げる前にユーザ優先順位限界を上げなければなりません。

管理者は、グローバルなタイムシェアリング優先順位とは独立にユーザ優先順位の最大値を設定します。たとえば、デフォルトの設定では、ユーザは –20 から +20 までの範囲だけにユーザ優先順位を設定できますが、60 種類のタイムシェアリングのグローバル優先順位が設定できます。

スケジューラは、タイムシェアリングのパラメタテーブル `ts_dptbl` 内の設定可能なパラメタを使用して、タイムシェアリングプロセスを管理します。このテーブルには、タイムシェアリングクラスに固有の情報が収められています。

システムクラス

システムクラスでは、固定優先順位方式を使用して、サーバなどのカーネルプロセスや、ページングデーモンなどのハウスキーピングプロセスを実行します。システムクラスは、カーネルが使用するために予約されており、ユーザはシステムクラスに対してプロセスの追加や削除はできません。システムクラスのプロセスの優先順位は、以上のようなプロセスのカーネルコードで設定されていて、いったん設定さ

れるとシステムプロセスの優先順位は変わりません (カーネルモードで実行中のユーザプロセスは、システムクラスにはありません)。

実時間クラス

実時間クラスでは、固定優先順位スケジューリング方式を使用しており、クリティカルなプロセスがあらかじめ設定された順序で実行されます。実時間優先順位は、ユーザが変更を要求しない限り変更されません。特権ユーザは、`priocntl` コマンドまたは `priocntl` 関数を使用して、実時間優先順位を割り当てることができます。

スケジューラは、実時間パラメタテーブル `rt_dptbl(4)` 内の設定可能なパラメタを使用して、実時間プロセスを管理します。このテーブルには、実時間クラスに固有の情報が収められています。

コマンドと関数

図 4–2 は、デフォルトのプロセス優先順位を示しています。

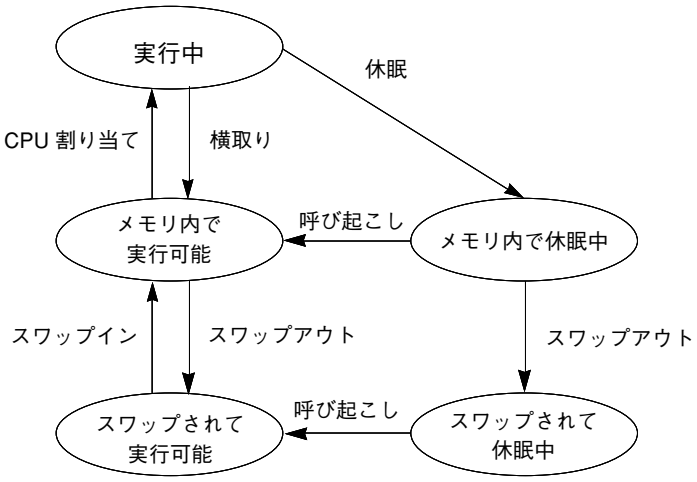

図 4–2　プロセスの優先順位 (プログラマから見た場合)

プロセス優先順位が意味を持つのは、スケジューリングクラスについてだけです。プロセス優先順位を指定するには、クラスとクラスに固有の優先順位の値を指定します。クラスとクラスに固有の値は、システムによってグローバル優先順位に割り当てられ、この値を使用してプロセスがスケジューリングされます。

優先順位は、システム管理者から見た場合と、ユーザやプログラマから見た場合とは異なります。スケジューリングクラスを設定する場合、管理者はグローバル優先順位を直接扱います。システムでは、ユーザが指定した優先順位は、このグローバル優先順位に割り当てられます。優先順位の詳細は、『*Solaris* のシステム管理 *(*第 *1* 巻*)*』を参照してください。

`ps -cel` コマンドは、有効なすべてのプロセスのグローバル優先順位を報告します。`priocntl` コマンドは、ユーザとプログラマが使用するクラス固有の優先順位を報告します。

`priocntl(1)` コマンド、`priocntl(2)` 関数、および `priocntlset(2)` 関数は、プロセスのスケジューリングパラメタを設定または検索します。優先順位を設定する場合の基本的な方法はどれも同じで、次のようになります。

- ターゲットプロセスを指定する。
- そのプロセスに希望するスケジューリングパラメタを指定する。
- プロセスにパラメタを設定するコマンドまたは関数を実行する。

以上の ID は UNIX のプロセスの基本的な特性です (詳細は、`Intro(2)` のマニュアルページを参照してください)。クラス ID は、プロセスのスケジューリングクラスです。`priocntl` は、タイムシェアリングクラスと実時間クラスだけに有効で、システムクラスには使用できません。

`priocntl(1)` コマンド

`priocntl(1)` ユーティリティは、プロセスをスケジューリングする際に、次の 4 つの制御機能を実行します。

`priocntl -l`	設定内容が表示される。
`priocntl -d`	プロセスのスケジューリングパラメタが表示される。
`priocntl -s`	プロセスのスケジューリングパラメタが設定される。
`priocntl -e`	指定したスケジューリングパラメタでコマンドが実行される。

以下に、`priocntl` を使用した例をいくつか示します。

デフォルト設定について `-l` オプションを使用すると、次のように出力されます。

```
$ priocntl -d -i all
CONFIGURED CLASSES ==================  SYS (System Class)
TS (Time Sharing) Configured TS User Priority Range -20 through 20
T (Real Time) Maximum Configured RT Priority: 59
```

すべてのプロセスについての情報を表示する例

```
$ priocntl -d -i all
```

すべてのタイムシェアリングプロセスについての情報を表示する例

```
$ priocntl -d -i class TS
```

ユーザ ID が 103 または 6626 のすべてのプロセスについての情報を表示する例

```
$ priocntl -d -i uid 103 6626
```

ID 24688 のプロセスに実時間プロセスのデフォルトパラメタを設定する例

```
$ priocntl -s -c RT -i pid 24668
```

ID 3608 のプロセスに優先順位 55 の実時間プロセスとして設定する例

```
$ priocntl -s -c RT -p 55 -t 1 -r 5 -i pid 3608
```

すべてのプロセスをタイムシェアリングプロセスに変更する例

```
$ priocntl -s -c RT -p 55 -t 1 -r 5 -i pid 3608
```

ユーザ ID 1122 のプロセスに対して、タイムシェアリングユーザ優先順位とユーザ優先順位限界を -10 に減少する例

```
$ priocntl -s -c TS -p -10 -m -10 -i uid 1122
```

デフォルトの実時間優先順位で実時間シェルを開始する例

```
$ priocntl -e -c RT /bin/sh
```

タイムシェアリングユーザ優先順位を -10 にして make を実行する例

```
$ priocntl -e -c TS -p -10 make bigprog
```

priocntl コマンドは、nice の機能を包んでいます。nice は、タイムシェアリングプロセスについてだけ有効で、数値が大きいほど優先順位が低くなります。上の例は、「増分」に -10 を指定して nice を実効するのと同じです。

```
$ nice -10 make bigprog
```

priocntl(2) 関数

priocntl(2) 関数は、priocntl(1) ユーティリティがプロセスに対して行うのと同様、1 つのプロセスまたは 1 組のプロセスのスケジューリングパラメタを取得または設定します。priocntl(2) 呼び出しは LWP に、1 つのプロセスだけに、またはプロセスのグループに働かせることができます。プロセスのグループは、親プロセス、プロセスグループ、セッション、ユーザ、グループ、クラス、またはアクティブなすべてのプロセスによって識別できます。使用方法については、マニュアルページを参照してください。

priocntl(2) を使用して priocntl –l と同等のことを行う例が、付録 Aにあります。

PC_GETCLINFO コマンドは、クラス ID を与えるとスケジューリングクラス名とパラメタを取得します。このコマンドは、どのクラスが設定されているかについて想定しないプログラムの作成を容易にします。PC_GETCLINFO と一緒に priocntl(2) を使用して、プロセス ID を元にしてプロセスのクラス名を取得する例は、コード例 A–2 にあります。

PC_SETPARMS コマンドは、1 組のプロセスのスケジューリングクラスとパラメタを設定します。*idtype* と *id* の入力引数は、変更するプロセスを指定します。コード例 A–3 は、PC_SETPARMS コマンドと一緒に priocntl(2) を使用して、タイムシェアリングプロセスを実時間プロセスに変換する例を示しています。

priocntlset(2) 関数

priocntlset(2) 関数は、priocntl(2) と同様に、1 組のプロセスのスケジューリングパラメタを変更します。priocntlset(2) には、priocntl(2) と同じコマンドセットがあり、*cmd* と *arg* の入力引数は同じです。ただし、priocntl(2) は 1 組の *idtype* と *id* のペアだけによって指定される 1 組のプロセスに適用されるのに対し、priocntlset(2) は 2 組の *idtype* と *id* のペアを論理的に結合した結果の 1 組のプロセスに適用されます。詳細は、マニュアルページを参照してください。

コード例 A–4では、priocntlset(2) を使用して、同じユーザ ID のタイムシェアリングプロセスを実時間プロセスに変更しないで、実時間プロセスの優先順位を変更する例を示します。

他の関数との関係

カーネルプロセス

カーネルは、デーモンやハウスキーピングプロセスをシステムのスケジューリングクラスに割り当てます。ユーザは、このクラスにプロセスを追加または削除したり、これらのプロセスの優先順位を変更したりできません。ps –cel コマンドによって、すべてのプロセスのスケジューリングクラスが示されます。システムクラスのプロセスは、CLS カラムに SYS と表示されます。

fork(2) と exec(2)

スケジューリングクラス、優先順位、その他のスケジューリングパラメタは、fork(2) 関数や exec(2) 関数を実行した場合も継承されます。

nice(2)

nice(1) コマンドと nice(2) 関数は、UNIX システムの以前のバージョンと同じ働きをします。 これらは、タイムシェアリングプロセスの優先順位を変更します。これらの関数でも、数値が小さいほどタイムシェアリング優先順位が高くなります。

プロセスのスケジューリングクラスや実時間優先順位を変更するには、priocntl 関数の 1 つを使用しなければなりません。priocntl 関数では、数値が大きいほど優先順位が高くなります。

init(1M)

init(1M) プロセスは、スケジューラによって特殊な場合として扱われます。init のスケジューラの設定項目を変更するには、*idtype* と *id*、または procset 構造体で、init だけが指定されていなければなりません。

性能

スケジューラは、プロセスをいつどれだけの時間実行するかを決定するので、システムの性能と見かけの性能に重要な影響を与えます。

デフォルトでは、プロセスはすべてタイムシェアリングプロセスです。プロセスがクラスを変更するのは、`priocntl()` 関数呼び出しによってだけです。

実時間プロセス優先順位は、どのタイムシェアリングプロセスよりも優先順位が高くなっています。したがって、実行可能な実時間プロセスが存在する限り、タイムシェアリングプロセスやシステムプロセスは実行されません。このため、実時間アプリケーションは注意して作成しないと、ユーザや基本的なカーネルのハウスキーピングが完全にロッキングされてしまうことがあります。

実時間アプリケーションは、プロセスのクラスと優先順位を制御する以外にも、性能に影響を与えるいくつかの他の要因も制御しなければなりません。性能にとって最も重要な要因は、CPU パワー、一次メモリ量、入出力スループットです。これらの要因は相互に複雑に関連しています。`sar(1)` コマンドには、すべての性能要因について報告するオプションがあります。

プロセスの状態変移

厳しい実時間制約を持つアプリケーションは、プロセスがスワップされたり二次メモリにページアウトされたりしないようにする必要がある場合があります。UNIX のプロセスの状態と状態間の変移の概要を図 4–3 に示します。

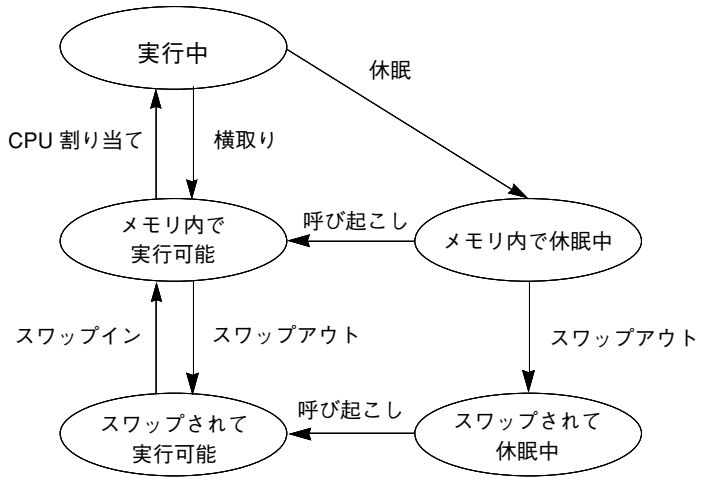

図 4–3　プロセス状態の変移図

有効なプロセスは、通常、上記の図の 5 つの状態のうちの 1 つにあります。矢印は状態が変わる方向を示します。

- プロセスは、CPU に割り当てられていれば実行中である。優先順位が高いプロセスが実行可能になると、実行中のプロセスはスケジューラによって横取りされる(実行状態から削除される)。プロセスがタイムスライスをすべて使用して、同じ優先順位のプロセスが実行可能な場合にも、プロセスは横取りされる。
- プロセスは一次メモリ内にあり、実行準備ができているがCPU には割り当てられていない場合、メモリ内で実行可能である。
- プロセスは一次メモリ内にあるが、実行を継続するために特定のイベントを待っている場合、メモリ内で休眠中である。たとえば、入出力操作の完了、ロッキングされている資源の解放、タイマの終了を待っている場合は休眠中である。イベントが発生すると、プロセスに呼び起こしが送信される。休眠の原因が解消されれば、プロセスは実行可能になる。
- プロセスが特定のイベントを待っているのではなく、一次メモリに他のプロセスのための空間を空けるために、アドレス空間がすべて二次メモリに書き込まれている場合、プロセスは実行可能でスワップされている。
- プロセスが特定のイベントを待っており、一次メモリに他のプロセスのための空間を空けるために、アドレス空間がすべて二次メモリに書き込まれている場合、プロセスは休眠中でスワップされている。

  有効なプロセスをすべて保持するための十分な一次メモリがマシンにない場合は、次のようにして、アドレス空間の一部を二次メモリにページングするかスワップしなければなりません。

- システムの一次メモリが不足した場合は、いくつかのプロセスの個々のページが二次メモリに書き込まれるが、そのプロセスは実行可能なままである。プロセスを実行する際にそのページにアクセスする場合は、ページが一次メモリ内に読み戻されるまでプロセスは休眠しなければならない。
- システムの一次メモリ不足がさらに深刻になると、いくつかのプロセスの全ページが二次メモリに書き込まれ、そのプロセスがスワップされたとマークされる。このようなプロセスは、システムスケジューラのデーモンプロセスによって選択されてメモリ内に読み戻された場合だけスケジュール可能な状態に戻る。

プロセスが再度実行可能になったときに、ページングとスワップの両方により、遅延が発生します。タイミング要求の厳しいプロセスにとっては、この遅延は受け入れられないものです。

実時間プロセスにすれば、プロセスの一部がページングされることがあっても、スワップはされないためスワップによる遅延を避けることができます。また、プログラムは、テキストとデータを一次メモリ内にロッキングすれば、ページングとスワップを避けることができます。詳細は、`memcntl(2)` のマニュアルページを参照してください。ロッキングできる量はメモリ設定によって制限されます。また、ロッキングが多すぎると、テキストやデータをメモリ内にロッキングしていないプロセスが大幅に遅れます。

実時間プロセスの性能とその他のプロセスの性能の兼ね合いは、ローカルなニーズによって異なります。システムによっては、必要な実時間応答を保証するためにプロセスのロッキングが必要な場合もあります。

ソフトウェアの潜在的な時間

実時間アプリケーションの潜在的な時間については、94ページの「ディスパッチ中の潜在的な時間」を参照してください。

第 5 章

シグナル

この章では、シグナルがアプリケーションに対してできることと、シグナルをどのようにして使用できるようにするかについて説明します。

概要

シグナルは、イベントが発生したときにプロセスに送られるソフトウェア生成割り込みです。シグナルは、SIGFPE や SIGSEGV など、アプリケーション内のエラーによって同期して生成されることもありますが、ほとんどは非同期です。システムで、別のプロセスからユーザが「割り込み」、「停止」、または「終了」要求を入れるなどのソフトウェアイベントを検出すると、シグナルをプロセスに送信できます。バスエラーや不当な命令などのハードウェアイベントが検出されると、カーネルから直接シグナルを送ることもあります。

プロセスに配信できる一連のシグナルは、システムで定義されています。シグナルの配信は、ハードウェア割り込みの発生に似ています。つまり、シグナルをそれ以降に発生したシグナルからブロックされ配信されないようにできます。配信先プロセスがシグナルに対応して動作を起こさないと、ほとんどのシグナルはそのプロセスを終了します。配信先プロセスを停止させるシグナルや、無視してもよいシグナルもあります。各シグナルのデフォルトの動作は、次のいずれか 1 つです。

- シグナルは受信後に廃棄される。
- シグナルの受信後にプロセスが終了する。
- コアファイルが書かれた後、プロセスが終了する。

- シグナルの受信後にプロセスを停止する。

システムによって定義されるシグナルは、次の 5 種類に分類できます。

- ハードウェア条件
- ソフトウェア条件
- 入出力通知
- プロセス制御
- 資源制御

シグナルのセットは、<signal.h> ヘッダに定義されています。

シグナル処理

シグナルはプロセスに配信されると、そのプロセスに対して保留状態にある一連のシグナルに加えられます。シグナルは、プロセスに対してブロッキングされていなければ配信されます。シグナルが配信されるとプロセスの現在の状態が保存され、新しいシグナルマスクが設定され、シグナルハンドラが呼び出されます。

BSD シグナルセマンティクスでは、シグナルがプロセスに配信されると、プロセスのシグナルハンドラの持続期間 (またはマスクを修正するインタフェースが呼び出されるまで)、新しいシグナルマスクがインストールされます。シグナルハンドラが実行している間は、このマスクは配信されたばかりのシグナルが、またプロセスに割り込まないようにブロッキングします。

System V シグナルセマンティクスでは、この保護を提供していないので、同じシグナルがそのシグナルを処理しているハンドラに割り込むことができます。このため、シグナルハンドラは再入可能であることが必要です。

すべてのシグナルは、同じ優先順位です。シグナルハンドラが呼び出したシグナルをブロッキングしても、他のシグナルをプロセスに配信できます。

シグナルはスタックされません。シグナルハンドラが、1 つのシグナルが実際に配信された回数を記録する方法はありません。

ブロッキング

グローバルシグナル「マスク」は、プロセスへの配信をブロッキングされる一連のシグナルを定義します。プロセスのシグナルマスクは、その初期状態を親プロセスのシグナルマスクからコピーしま

す。sigaction(2)、sigblock(3B)、sigsetmask(3B)、sigprocmask(2)、sigsetops(3C)、sighold(3C)、または sigrelse(3C) を呼び出すと、マスクを変更できます (詳細は、signal(3C) を参照してください)。

ハンドリング

アプリケーションプログラムは、特定のシグナルが受信されると呼び出される関数「シグナルハンドラ」を指定できます。シグナルを受信してシグナルハンドラが呼び出されることを、シグナルを「キャッチする」と言います。プロセスは、次のいずれか 1 つの方法でシグナルを処理できます。

- プロセスはデフォルトの動作を起動できる。
- プロセスはシグナルをブロッキングできる (無視できないシグナルもある)。
- プロセスはハンドラでシグナルをキャッチできる。

シグナルハンドラは、通常はプロセスの現スタック上で実行します。こうするとシグナルハンドラは、プロセスで実行が割り込まれた位置に戻ることができます。シグナルハンドラが特定のスタックに実行するように、これをシグナルごとに変更できます。割り込まれたコンテキスト以外のコンテキストで再開しなければならない場合は、プロセスは以前のコンテキスト自体を復元しなければなりません。

シグナルハンドラのインストール

signal(3C)、sigset(3C)、signal(3B)、および sigvec(3B) すべてを使用して、シグナルハンドラをインストールできます。これらのすべては、シグナルに対する前の動作を戻します。4 つのインタフェースには、1 つの重要な違いがあります。signal(3C) は System V シグナルセマンティクスとなります (同じシグナルがそのハンドラに割り込むことができます)。 sigset(3C)、signal(3B)、および sigvec(3B) は、すべて BSD シグナルセマンティクスになります (シグナルはそのハンドラが復帰するまでブロッキングされます)。さらに、signal(3C) と signal(3B) の両方とも、シグナルを無視するかデフォルトの動作を復元するかを制御できます。コード例 5–1 は、簡単なハンドラとそのインストールを示しています。

コード例 5–1　シグナルハンドラのインストール

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

void sigcatcher()
```

```
{
  printf (``PID %d caught signal.\n''. getpid());
}

main()
{
  pid_t ppid;

  signal (SIGINT, sigcatcher);
  if (fork() == 0) {
   sleep( 5 );
   ppid = getppid();
   while( TRUE )
    if (kill( ppid, SIGINT) == -1 )
     exit( 1 );
  }
  pause();
}
```

シグナル SIGKILL または SIGSTOP に対して、ハンドラのインストールまたは SIG_IGN の設定を試みるとエラーになります。シグナル SIGCONT に対して SIG_IGN の設定を試みてもエラーになります。これはデフォルトで無視されているためです。

シグナルハンドラをインストールすると、signal(3C)、sigset(3C)、signal(3B)、または sigvec(3B) をもう一度呼び出して明示的に置き換えるまで、インストールされたままになります。

SIGCHILD のキャッチ

子プロセスが停止または終了すると、SIGCHILD が親プロセスに送られます。このシグナルへのデフォルトの応答は無視することです。このシグナルをキャッチでき、すぐに wait(2) および wait3(3C) を呼び出して、子プロセスからの終了ステータスを得ることができます。こうすると、ゾンビプロセスのエントリをできるだけ素早く削除できます。コード例 5–2 は、SIGCHILD をキャッチするハンドラのインストールを示しています。

コード例 **5–2**　SIGCHILD をキャッチするハンドラのインストール

```
#include <stdio.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/resource.h>

void proc_exit()
{
  int wstat;
  union wait wstat;
  pid_t pid;
```

```
  while (TRUE) {
   pid = wait3 (&wstat, WNOHANG, (struct rusage *)NULL );
   if (pid == 0)
    return;
   else if (pid == -1)
    return;
   else
    printf ("Return code: %d\n", wstat.w_retcode);
  }
}
main ()
{
  signal (SIGCHLD, proc_exit);
  switch (fork()) {
   case -1:
    perror ("main: fork");
    exit (0);
   case 0:
    printf ("I'm alive (terporarily)\n");
    exit (rand());
   default:
    pause();
  }
}
```

SIGCHILD をキャッチャするハンドラは、通常はプロセス初期化の一部として設定します。これらは、子プロセスをフォークする前に設定しなければなりません。典型的な SIGCHILD ハンドラは、子プロセスの終了状態を検索するだけです。

第 6 章

入出力インタフェース

この章では、仮想記憶サービスを提供していないシステムに提供されるファイル入出力操作を紹介します。仮想記憶機能によって提供される向上した入出力方式についても言及します。また、ファイルとレコードをロッキングする旧式の重量方式も説明します。

ファイルと入出力

一連のデータとして構成されたファイルを「通常」ファイルと言います。このようなファイルには、ASCII テキスト、他の符号化バイナリデータによるテキスト、実行可能コード、またはテキスト、データ、コードの組み合わせが入っています。ファイルは、次の 2 つのコンポーネントに分かれています。

- 「i ノード」と呼ばれる制御データ。これらのデータには、ファイルタイプ、アクセス権、所有者、ファイルサイズ、データブロックの位置が含まれる。
- ファイルの内容。区切れのないバイトのシーケンス

Solaris は、次の 3 つの基本的なファイル入出力インタフェースを用意しています。

- 第 1 の形式は、伝統的な様式のファイル入出力。詳細は、48ページの「基本ファイル入出力」を参照してください。
- 第 2 の形式は、「標準のファイル入出力」。標準の入出力バッファリングによって、インタフェースが容易になり、仮想メモリのないシステム上で実行されるアプリケーションの効率を改善できます。Solaris 7 オペレーティング環境など、仮

想メモリ環境で動作するアプリケーションの場合、標準のファイル入出力はきわめて効率の悪い形式です。

- 第 3 のファイル入出力形式は、63ページの「メモリ管理インタフェース」で説明するメモリマッピングインタフェースによって提供されます。マッピングファイルは、Solaris 7 環境で実行されるほとんどのアプリケーションに最も効率的で高性能のファイル入出力形式です。

基本ファイル入出力

表 6–1に示している関数は、ファイルで基本操作を実行します。

表 6–1 基本的なファイル入出力関数

関数名	目的
`open(2)`	読み取りまたは書き込み用にファイルを開く。
`close(2)`	ファイル記述子を閉じる。
`read(2)`	ファイルから読み取る。
`write(2)`	ファイルに書き込む。
`creat(2)`	新しいファイルを作成するか、既存のファイルに上書きする。
`unlink(2)`	ディレクトリエントリを削除する。
`lseek(2)`	読み取り / 書き込み用のファイルポインタを移動する。

次のコード例は、基本的なファイル入出力インタフェースの使用方法を示します。`read(2)` と `write(2)` はどちらも、現在のファイルのオフセットから指定された数を超えないバイト数を転送します。実際に転送されたバイト数が戻されます。ファイルの終わりでは、 `read(2)` の戻り値が 0 になります。

```
#include   <fcntl.h>
#define   MAXSIZE   256

main()
{
  int  fd;
```

```
  ssize_t n;
  char  array[MAXSIZE];


  fd = open ("/etc/motd", O_RDONLY);
  if (fd == -1) {
   perror ("open");
   exit (1);
  }
  while ((n = read (fd, array, MAXSIZE)) > 0)
   if (write (1, array, n) != n)
    perror ("write");
  if (n == -1)
   perror ("read");
  close (fd);
}
```

読み取りまたは書き込みを完了したら、必ずファイルを閉じてください。

開いているファイル内のオフセットは、read(2)、write(2)、または lseek(2) を呼び出すことによって変更されます。次に lseek(2) の使用例を示します。

```
off_t  start, n;
 struct  record  rec;

 /* record current offset in start */
 start = lseek (fd, 0L, SEEK_CUR);

 /* go back to start */
 n = lseek (fd, -start, SEEK_SET);
 read (fd, &rec, sizeof (rec));

 /* rewrite previous record */
 n = lseek (fd, -sizeof (rec), SEEK_CUR);
 write (fd, (char *&rec, sizeof (rec));
```

高度なファイル入出力

高度なファイル入出力関数は、ディレクトリとファイルの作成と削除、既存のファイルへのリンクの作成、ファイル状態情報の取得または変更を行います。

表 6–2 高度なファイル入出力関数

関数名	目的
link(2)	ファイルにリンクする。
access(2)	ファイルのアクセス可能性を判定する。
mknod(2)	特殊ファイルまたは通常のファイルを作成する。
chmod(2)	ファイルのモードを変更する。
chown(2),lchown(2),fchown(2)	ファイルの所有者とグループを変更する。

表 6–2　高度なファイル入出力関数　続く

utime(2)	ファイルのアクセス時刻や変更時刻を設定する。
stat(2)、 lstat(2)、fstat(2)	ファイルのステータスを取得する。
fcntl(2)	ファイル制御機能を実行する。
ioctl(2)	デバイスを制御する。
fpathconf(2)	設定可能なパス名変数を取得する。
opendir(3C),readdir(3C), closedir(3C)	ディレクトリを操作する。
mkdir(2)	ディレクトリを作成する。
readlink(2)	シンボリックリンクの値を読みとる。
rename(2)	ファイル名を変更する。
rmdir(2)	ディレクトリを削除する。
symlink(2)	ファイルへのシンボリックリンクを作成する。

ファイルシステム制御

ファイルシステム制御関数を使用して、ファイルシステムを制御できます。

表 6–3　ファイルシステム情報を取得する。

関数名	目的
ustat(2)	ファイルシステムの統計情報を取得する。
sync(2)	スーパーブロッキングを更新する。
mount(2)	ファイルシステムをマウントする。
statvfs(2),fstatvfs(2)	ファイルシステム情報を取得する。
sysfs(2)	ファイルシステムの種類の情報を取得する。

ファイルとレコードのロッキング

従来のファイル入出力を使用して、ファイル要素をロッキングする必要はありません。『マルチスレッドのプログラミング』で説明している軽量の同期メカニズムは、マッピングされたファイルと一緒に効果的に使用できるため、この節で説明する旧式のファイルロッキングよりも効率的です。

ファイルまたはその一部をロッキングすると、2 人以上のユーザがファイルの情報を同時に更新しようとした場合に生じるエラーを防止できます。

ファイルロッキングとレコードロッキングは、実際には同じものです。ただし、ファイルロッキングではファイル全体へのアクセスをブロッキングするのに対し、レコードロッキングではファイルの指定されたセグメントへのアクセスだけをブロッキングします。(SunOS 5 システムでは、すべてのファイルはデータのバイトシーケンスであり、レコードはファイルを使用するプログラムの概念です。)

サポートするファイルシステム

次のタイプのファイルシステム上では、アドバイザリロッキングと強制ロッキングがサポートされます。

- `ufs` — ディスクベースのデフォルトのファイルシステム
- `fifofs` — プロセスが共通の方法でデータにアクセスできるようにする名前付きパイプファイルからなる疑似ファイルシステム
- `namefs` — ファイル記述子をファイル上に動的にマウントするために、主に STREAMS によって使用される疑似ファイルシステム
- `specfs` — 特殊なキャラクタ型デバイスやブロッキング型デバイスにアクセスするための疑似ファイルシステム

  NFS 上では、アドバイザリファイルロッキングのみがサポートされます。

  `proc` ファイルシステムと `fd` ファイルシステム上では、ファイルロッキングはサポートされません。

ロッキングタイプの選択

強制ロッキングでは、要求されたファイルセグメントが解放されるまで、プロセスは待機します。アドバイザリロッキングでは、ロッキングが取得されたかどうかを

示す結果だけが返されます。プロセスは、その結果を無視して入出力を続けることができます。同一のファイルに強制ロッキングとアドバイザリロッキングを同時には適用できません。開いたときのファイルのモードによって、そのファイル上の既存のロッキングが強制ロッキングとして処理されるか、アドバイザリロッキングとして処理されるかが決まります。

2 つの基本的なロッキング呼び出しのうち、`fcntl(2)` の方が `lockf(3C)` よりも移植性が高く高性能ですが、より複雑です。`fcntl(2)` は POSIX 1003.1 で規格されています。 `lockf(3C)` は、他のアプリケーションとの互換性を保つために用意されています。

用語

この節の残りの部分を読むために役立つ定義の一部を示します。

用語	定義
レコード	ファイル内のバイトの連続したセット。UNIX オペレーティングシステムでは、ファイルのレコード構造は定義されていない。ファイルを使用するプログラムで、必要なレコード構造を定義できる。
共同プロセス	共用リソースのアクセスを規制するいくつかのメカニズムを使用する 2 つ以上のプロセス
読み取りロッキング	他のプロセスにも読み取りロッキングの適用や読み取りの実行をさせ、他のプロセスの書き込みまたは書き込みロッキングの適用をブロッキングする。
書き込みロッキング	他のすべてのプロセスの読み取り、書き込み、またはロッキングの適用をブロッキングする。
アドバイザリロッキング	ロッキングを保持していないプロセスをブロッキングしないでエラーを返す。アドバイザリロッキングは、`creat(2)`、`open(2)`、`read(2)`、および `write(2)` の処理には適用されない。
強制ロッキング	ロッキングを保持していないプロセスの実行をブロッキングする。ロッキングされているレコードへのアクセスは、`creat(2)`、`open(2)`、`read(2)`、および `write(2)` の処理に適用される。

ロッキング用にファイルを開く

ロッキングは、有効な記述子を持つファイルに対してだけ要求できます。読み取りロッキングの場合は、少なくとも読み取りアクセスを設定してファイルを開かなければなりません。書き込みロッキングの場合は、書き込みアクセスも設定してファイルを開かなければなりません。次の例では、ファイルは読み取りと書き込みアクセス用に開かれます。

```
...
  filename = argv[1];
  fd = open (filename, O_RDWR);
  if(fd < 0) {
   perror(filename);
   exit(2);
  }
  ...
```

ファイルロッキングの設定

ファイル全体をロッキングするには、オフセットを 0 に設定し、サイズを 0 に設定します。

ファイルをロッキングするには、いくつかの方法があります。どの方法を選択するかは、ロッキングとプログラムの他の部分との関係、または性能や移植性によって決まります。次の例では、POSIX 標準互換の `fcntl(2)` 関数を使用します。このプログラムは、次のいずれかの状況が発生するまでファイルをロッキングしようとします。

- ファイルが正常にロッキングされた
- エラーが発生した
- MAX_TRY 回数を超えたため、プログラムがファイルのロッキングを中止した

```
#include <fcntl.h>

 ...
  struct flock lck;

 ...
  lck.l_type = F_WRLCK; /* 書き込みロッキングを設定する */
  lck.l_whence = 0; /* ファイルの先頭からのオフセットは l_start */
  lck.l_start = (off_t)0;
  lck.l_len = (off_t)0; /* ファイルの最後まで */
  if (fcntl(fd, F_SETLK, &lck) <0) {
   if (errno == EAGAIN || errno == EACCES) {
    (void) fprintf(stderr, "File busy try again later!\n");
    return;
   }
   perror("fcntl");
   exit (2);
  }
  ...
```

fcntl(2) を使用して、いくつかの構造体変数を設定し、ロッキング要求の型と開始を設定できます。

注 - マッピングされたファイルを flock(3B) でロッキングできません。詳細は、mmap(2) を参照してください。しかし、(POSIX スタイルまたは Solaris スタイルでの) マルチスレッド指向同期メカニズムをマッピングされたファイルで使用できます。 mutex(3T)、condition(3T)、semaphore(3T)、および rwlock(3T) のマニュアルページを参照してください。

レコードロッキングの設定と解除

レコードのロッキングは、ロッキングセグメントの開始位置と長さが 0 に設定されないこと以外は、ファイルのロッキングと同様に行います。

必要なすべてのロッキングを設定できない場合に備えて、対処方法を用意しておいてください。レコードロッキングを使用するのは、レコードへのアクセスが競合するためです。実行される動作は、プログラムによって次のように異なります。

- 一定時間待ってから再試行する
- 手順を中止してユーザに警告する
- ロッキングが解除されたことを示すシグナルを受信するまでプロセスを休眠させておく
- 上記のいくつかを組み合わせて実行する

次の例は、fcntl(2)を使用してレコードをロッキングする方法を示します。

```
{
    struct flock lck;
      ...
    lck.l_type = F_WRLCK; /* 書き込みロッキングを設定する */
    lck.l_whence = 0; /* ファイルの先頭からのオフセットは l_start */
    lck.l_start = here;
    lck.l_len = sizeof(struct record);

    /* this に書き込みロッキングを設定する */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
     /* this の書き込みロッキングが失敗 */
     return (-1);
   ...
  }
```

次の例は、lockf(3C) 関数を示します。

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

```
 ...
  /* this をロッキングする */
  (void) lseek(fd, this, SEEK_SET);
  if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
   /* this のロッキングが失敗。here のロッキングを解除する */
   (void) lseek(fd, here, 0);
   (void) lockf(fd, F_ULOCK, sizeof(struct record));
   return (-1);
}
```

各ロッキングはタイプが異なるだけで、設定されたときと同じ方法で解除されます (F_ULOCK)。ロッキングの解除は別のプロセスによってブロッキングされず、そのプロセスが設定したロッキングに対してだけ有効です。ロッキング解除は、前のロッキング呼び出しで指定されたファイルのセグメントに対してだけ有効です。

ロッキング情報の取得

ロッキングを設定できないようにブロッキングしているプロセスがあれば、それがどのプロセスかを判定できます。これは簡単なテストとして、またはファイル上のロッキングを見つけるために使用できます。ロッキングは上記の例のように設定され、`fcntl()` 呼び出しで `F_GETLK` が使用されます。次の例では、ファイル内でロッキングされているすべてのセグメントについてのデータを検索して出力します。

```
struct flock lck;

  lck.l_whence = 0;
  lck.l_start = 0L;
  lck.l_len = 0L;
  do {
   lck.l_type = F_WRLCK;
   (void) fcntl(fd, F_GETLK, &lck);
   if (lck.l_type != F_UNLCK) {
    (void) printf("%d %d %c %8d %8d\n", lck.l_sysid, lck.l_pid,
      (lck.l_type == F_WRLCK) ? 'W' : 'R', lck.l_start, lck.l_len);
    /* このロッキングがアドレス空間の終わりまで続いている場合は、
     * それ以上探す必要がないのでループは終了する */
    if (lck.l_len == 0) {
    /* それ以外の場合は、見つかったロッキングの後方のロッキングを探す */
      lck.l_start += lck.l_len;
    }
   }
  } while (lck.l_type != F_UNLCK);
```

`F_GETLK()` コマンドを伴う `fcntl(2)` 関数は、サーバの応答を待つ間に休眠することがあり、クライアント側またはサーバ側の資源が不足すると失敗する場合もあります (`ENOLCK` を戻します)。

F_TEST コマンドを伴う lockf(3C) 関数は、プロセスがロッキングを 保持しているかどうかの検査にも使用できます。ただしこの関数は、ロッキングの位置とそのロッキングを 所有しているプロセスについての情報は戻しません。

```
(void) lseek(fd, 0, 0L);
 /* ファイルアドレス空間の終わりまで検索するため、
    テスト領域の大きさを 0 に設定する */
 if (lockf(fd, (off_t)0, SEEK_SET) < 0) {
  switch (errno) {
   case EACCES:
   case EAGAIN:
    (void) printf("file is locked by another process\n");
    break;
   case EBADF:
    /* lockf に渡された引数が不正 */
    perror("lockf");
    break;
   default:
    (void) printf("lockf: unexpected error <%d>\n", errno);
    break;
  }
```

ロッキングの継承

プロセスが fork を行うと、子プロセスは親プロセスが開いたファイル記述子のコピーを受け取ります。ただし、ロッキングは特定のプロセスによって所有されるので、子プロセスに継承されません。親と子は、ファイルごとに共通のファイルポインタを共有します。両方のプロセスが、同じファイル内の同じ位置にロッキングを設定しようとすることがあります。この問題は、lockf(3C) と fcntl(2) でも発生します。レコードをロッキングするプログラムが fork を行う場合、子プロセスはファイルを閉じてから開き直して、新しく別のファイルポインタを設定する必要があります。

デッドロッキング処理

UNIX のロッキング機能を使用すると、デッドロッキングの検出と防止ができます。デッドロッキングが発生する可能性があるのは、システムがレコードロッキング機能を休眠させようとするときだけです。プロセス A がプロセス B に保持されたロッキングを待つと同時に、プロセス B がプロセス A に保持されたロッキングを待つような状況が発生していないかが検査されます。潜在的なデッドロッキングが検出されると、ロッキング関数は失敗し、デッドロッキングを示す値が errno に設定されます。F_SETLK を使用してロッキングを設定するプロセスは、ロッキングが即座に取得できなくても取得できるまで待たないので、デッドロッキングは発生しません。

アドバイザリロッキングと強制ロッキングの選択

強制ロッキングの場合、対象のファイルは set-group 識別 (setgid) グループ ID 設定ビットがオンになっており、グループの実行権がオフになっている通常ファイルでなければなりません。どちらかの条件が満たされなければ、すべてのレコードロッキングはアドバイザリロッキングになります。強制ロッキングを使用するには、次の例のようにします。

```
#include <sys/types.h>
#include <sys/stat.h>

 int mode;
 struct stat buf;
  ...
  if (stat(filename, &buf) < 0) {
   perror("program");
   exit (2);
  }
  /* 現在設定されているモードを取得する */
  mode = buf.st_mode;
  /* グループの実効権をモードから削除する */
  mode &= ~(S_IEXEC>>3);
   /* set-group 識別 (setgid) ビットをモードに設定する */
  mode |= S_ISGID;
  if (chmod(filename, mode) < 0) {
   perror("program");
   exit(2);
  }
  ...
```

レコードロッキングが適用されるファイルの場合は、実行権を設定しないでください。これは、オペレーティングシステムはファイルの実行時にレコードロッキングを無視するからです。

ファイルに強制ロッキングを設定するには、chmod(1) コマンドも使用できます。このコマンドを使用すると次のようになります。

```
$ chmod +l file
```

このコマンドはファイルモード内に O20n0 アクセス権ビットを設定します。これはファイルの強制ロッキングを示します。n が偶数の場合、そのビットは強制ロッキングを有効にすると解釈されます。n が奇数の場合、そのビットは実行時に set-group 識別 (setgid) がオンになると解釈されます。

ls(1) コマンドで –l オプションを使用してロングリスト形式を指定すると、 この設定が表示されます。

```
$ ls -l file
```

この場合、次のような情報が表示されます。

```
-rw---l--- 1 user group size mod_time file
```

アクセス権の英字 "l" は、set-group 識別 (setgid) ビットがオンになっていることを示します。したがって、通常の set-group 識別 (setgid) ビットがオンになるだけでなく強制ロッキングが有効になっています。

強制ロッキングについての注意事項

- 強制ロッキングは、ローカルファイルだけで利用できます。NFS を介してファイルにアクセスするときは利用できません。
- 強制ロッキングは、ファイル内のロッキングされているセグメントだけを保護します。ファイルの残りの部分には、通常のファイルアクセス権に従ってアクセスできます。
- 原子操作的トランザクションのために多重の読み取りや書き込みが必要な場合は、入出力が始まる前に、対象となるすべてのセグメントについてプロセスが明示的にロッキングする必要があります。このように動作するプログラムの場合は、いずれもアドバイザリロッキングで十分です。
- レコードロッキングが使用されるファイルについては、どんなプログラムでも無制限のアクセス権を与えないでください。
- 入出力要求のたびにレコードロッキング検査を実行する必要がないので、アドバイザリロッキングの方が効率的です。

端末入出力

端末入出力関数は、次の表に示すように、非同期通信ポートを制御する一般的な端末インタフェースを処理します。`termios(3)` と `termio(7I)` のマニュアルページも参照してください。

表 **6–4** 端末入出力関数

	目的
`tcgetattr(3),tcsetattr(3)`	端末属性を取得または設定する。
`tcsendbreak(3),tcdrain(3), tcflush(3),tcflow(3)`	回線制御関数を実行する。
`cfgetospeed(3),cfgetispeed(3), cfsetispeed(3),cfsetospeed(3)`	ボーレートを取得または設定する。
`tcsetpgrp(3)`	端末のフォアグラウンドプロセスのグループ ID を取得または設定する。
`tcgetsid(3)`	端末のセッション ID を取得する。

第 7 章

メモリ管理

この章では、アプリケーション開発者の観点から SunOS 5 仮想記憶を説明し、アプリケーション開発者が利用できる、固定記憶のシステムにはない仮想記憶の機能を示します。また、これらの機能を使用し、制御するために SunOS 5 で提供されているインタフェースを説明します。

仮想記憶の概要

(仮想記憶ではない) 固定記憶のシステムでは、プロセスの実アドレス空間は、システム主記憶のある部分を占有し、それに限定されます。

SunOS 5 の仮想記憶では、プロセスの実アドレス空間は、ディスク記憶装置のスワップパーティションにあるファイルを占有します (このファイルをバッキングストアと言います)。主記憶のページは、プロセスアドレス空間のアクティブな (または最近アクティブだった) 部分をバッファリングし、CPU が実行するコードとプログラムが処理するデータを提供します。

現在メモリにないアドレスが CPU によってアクセスされ、「ページフォルト」が発生すると、アドレス空間のページが読み込まれます。参照されているアドレスセグメントがメモリに読み込まれてページフォルトが解決されるまで実行は継続できないので、ページが読み込まれるまでプロセスは休眠します。

アプリケーション開発者にとって 2 つのメモリシステムの最も明瞭な違いは、仮想記憶によって、アプリケーションはより大きいアドレス空間を占有できることです。また、仮想記憶を使用すると、ファイル入出力が簡単で効率的になり、プロセス間のメモリ共用が非常に効率的になります。

アドレス空間とマッピング

バッキングストアファイル (プロセスのアドレス空間) は、スワップ空間だけに存在するため、UNIX 名前付きファイル空間に含まれていません。(このため、他のプロセスはバッキングストアファイルにアクセスできません。) しかし、1 つ以上の名前付きファイルの全部または一部をバッキングストアに論理的に挿入できるようにしたり、その結果を単一のアドレス空間として扱えるようにしたりするために簡単に拡張できます。このことを「マッピング」と言います。

マッピングを使用すると、読み取り可能または書き込み可能なファイルの任意の部分を論理的にプロセスのアドレス空間に含めることができます。プロセスのアドレス空間の他の部分と同様に、ページフォルトによって強制されるまで、ファイルのどのページも実際にはメモリに読み込まれません。その内容が修正された場合だけ、メモリのページはファイルに書き込まれます。そのため、ファイルの読み取りと書き込みは、完全に自動的かつ効率的に行われます。

複数のプロセスを 1 つの名前付きファイルにマッピングできます。こうすることで、効率的にプロセス間のメモリを共用できます。他のファイルの全部または一部をプロセス間で共用することもできます。

マッピングできない名前付きファイルシステムのオブジェクトもあります。たとえば、端末やネットワークデバイスファイルなどの記憶領域として扱うことができないデバイスなどです。

プロセスのアドレス空間は、アドレス空間にマッピングされるファイルの全部 (またはファイルの一部) によって定義されます。各マッピングは、プロセスが実行されるシステムのページ境界に合うように大きさと境界割り当てが調整されます。プロセスに関連付けられているメモリはありません。

プロセスのページは、一度に 1 つのオブジェクトだけにマッピングされますが、オブジェクトのアドレスは、多くのプロセスのマッピングの対象になることがあります。「ページ」という概念は、マッピングされるオブジェクトの属性ではありません。オブジェクトのマッピングは、プロセスがオブジェクトの内容を読み取るまたは書き込むための潜在的な機能だけを提供します。

マッピングによって、プロセスはオブジェクトの内容を直接アドレス指定できます。アプリケーションは、読み取りと書き込みによって間接的ではなく、使用する記憶領域の資源に直接アクセスできると便利な場合があります。たとえば、効率化 (不要なデータコピーの省略) と単純化 (読み取り、バッファの変更、書き込みではなく 1 つの操作で更新) などの利点があります。オブジェクトにアクセスし、アクセス中その識別情報を保つ機能は、このアクセス方式に固有で、この機能により共通のコードとデータを共用できます。

ファイルシステムの名前空間は、NFS を介して他のシステムから接続されているディレクトリツリーを含むので、ネットワークに接続されたファイルをプロセスのアドレス空間にマッピングすることもできます。

一貫性

複数のプロセスが 1 つのファイルを同時にマッピングするときに、メモリまたはファイルに含まれているデータを共用するかに関係なく、データ要素に同時にアクセスすると問題が発生することがあります。そのようなプロセスは、SunOS 5 で提供される同期メカニズムのいずれかによって協調できます。SunOS 5 で最も効率的な同期メカニズムは非常に軽量なので、スレッドライブラリに組み込まれています。その使用方法については、`mutex(3T)`、`condition(3T)`、`rwlock(3T)`、および `semaphore(3T)` の各マニュアルページを参照してください。

メモリ管理インタフェース

仮想記憶機能は、いくつかの関数の組み合わせによって使用し、制御します。この節は、これらの呼び出しを要約して説明し、それらの使用例を示します。

マッピングの作成と使用

`mmap(2)` は、プロセスのアドレス空間への名前付きファイルシステムオブジェクト (またはその一部) のマッピングを確立します。これは、基本的なメモリ管理機能で、非常に簡単です。まず、ファイルを `open(2)` し、次に適当なアクセスオプションと共用オプションで `mmap(2)` し、後は必要な作業を行います。

`mmap` によって確立されるマッピングは、指定されたアドレス範囲の以前のマッピングを置き換えます。

`MAP_SHARED` および `MAP_PRIVATE` フラグは、マッピングのタイプを指定します。いずれか 1 つを指定しなければなりません。`MAP_SHARED` を指定すると、書き込みがマッピングされているオブジェクトを修正します。変更を有効にするための追加操作は必要ありません。`MAP_PRIVATE` を指定すると、マッピングされている領域への最初の書き込みでページのコピーが作成され、すべての書き込みはそのコピーを参照します。実際に修正されたページだけがコピーされます。

マッピングのタイプは、`fork(2)` を実行した場合も保持されます。

mmap(2) 呼び出しに使用するファイル記述子は、マッピングが確立した後は開いたままにする必要はありません。ファイル記述子を閉じても、マッピングが munmap(2) によって取り消されるまで、または新しいマッピングで置き換えられるまで、このマッピングは有効です。

切り捨ての呼び出しによってファイルが短くされて、マッピングがファイルの終わりを越えてしまった場合は、ファイルの存在しない領域にアクセスすると、SIGBUS シグナルが発生します。

/dev/zero をマッピングすると、mmap で指定した大きさの仮想記憶のブロッキングに 0 を設定します。次のプログラムは、システムが選択したアドレスにスクラッチ記憶領域のブロッキングを作成する例を示しています。

```
int fd;
  caddr_t result;

  if ((fd = open("/dev/zero", O_RDWR)) == -1)
   return ((caddr_t)-1);
  result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  (void) close(fd);
```

デバイスまたはファイルは、マッピングによってアクセスした場合だけ役立つことがあります。この例としては、ビットマップ表示をサポートするフレームバッファデバイスがあります。この場合、表示管理アルゴリズムは、表示アドレスに直接ランダムアクセスできる場合に効率的に機能します。

マッピングの削除

munmap(2) は呼び出しプロセスから、指定したアドレス範囲内のページのすべてのマッピングを削除します。munmap は、マッピングされたオブジェクトには影響しません。

キャッシュ制御

SunOS 5 仮想記憶システムは、プロセッサのメモリがファイルシステムのオブジェクトからのデータをバッファリングするキャッシュシステムです。キャッシュの状態を制御または照会するためのインタフェースがあります。

mincore(2)

mincore(2) は、メモリページが指定された範囲のマッピングの対象となるアドレス空間にあるかどうか調べます。mincore がチェックした後でも、データを戻すまでの間にページの状態が変わる可能性があるため、戻された情報は古くなっていることがあります。ロッキングされたページだけがメモリに残っていることを保証されます。

mlock(3C) と munlock(3C)

mlock(3C) は、指定されたアドレス範囲にあるページを物理メモリでロッキングします。(このプロセスまたは他のプロセスで) ロッキングされたページを参照しても、入出力操作を必要とするページフォルトが生じることはありません。この操作は物理資源を拘束し、通常のシステム操作を破壊する可能性があるため、mlock(3C) を使用できるのはスーパーユーザだけに制限されています。設定に依存するページ制限だけがメモリ内にロッキングされます。この制限を越えると、mlock(3C) 呼び出しは失敗します。

munlock(3C) は、物理ページのロッキングを解放します。1 つのマッピングのアドレス範囲で複数の mlock 呼び出しを行なっている場合も、1 回の munlock 呼び出しだけでロッキングが解放されます。ただし、同じページの異なったマッピングが mlock で処理されている場合は、すべてのマッピングへのロッキングが解放されるまでページのロッキングは解除されません。

ロッキングは、マッピングが mmap 操作で置き換えられるか、munmap で削除された場合にも解放されます。

MAP_PRIVATE マッピングに伴う「書き込み時コピー」イベントの際に、ロッキングはページ間を移転されるため、MAP_PRIVATE マッピングを含むアドレス範囲のロッキングは、「書き込み時コピー」のリダイレクトに合わせて透過的に保持されます (リダイレクトについては、63ページの「マッピングの作成と使用」を参照してください)。

mlockall(3C) と munlockall(3C)

mlockall(3C) と munlockall(3C) は、mlock(3C) と munlock(3C) に似ていますが、アドレス空間全体を操作します。mlockall(3C) はアドレス空間にあるすべてのページにロッキングを設定し、munlockall(3C) は mlock(3C) と mlockall(3C) のどちらで設定されたかに関係なく、アドレス範囲にあるすべてのページのロッキングを解除します。

msync(3C)

msync(3C) は、指定されたアドレス範囲にある修正されたすべてのページを、そのアドレスによってマッピングされているオブジェクトにフラッシュします。これは、ファイルに対する fsync(3C) 操作と同様です。

その他のメモリ制御機能

sysconf(3C)

sysconf(3C) は、システムに依存するメモリページの大きさを戻します。移植性を保つため、アプリケーションにページの大きさを指定する定数を組み込まないでください。同じ命令セットの実装においてもページの大きさが異なることは珍しくありません。

mprotect(2)

mprotect(2)は、指定されたアドレス範囲にあるすべてのページに指定された保護を割り当てます。割り当てられた保護は、対象となるオブジェクトに認められている権限を越えることはできません。

brk(2) と sbrk(2)

brk(2) と sbrk(2) は、記憶領域をプロセスのデータセグメントに追加するために呼び出されます。

プロセスは brk(2) と sbrk(2) を呼び出して、この領域を操作できます。

```
caddr_t
 brk(caddr_t addr);

caddr_t
 sbrk(int incr);
```

brk(2) は、呼び出し側によって使用されていない最小値のデータセグメント位置を *addr*に設定します (システムのページサイズの最小倍数に丸められます)。

代替関数である sbrk(2) は、*incr* バイトを呼び出し側のデータ空間に追加し、新しいデータ領域の開始位置へのポインタを戻します。

第 8 章

プロセス間通信

この章は、マルチプロセスアプリケーションを開発するプログラマを対象としています。

Solaris 7 およびその互換性オペレーティング環境には、並行プロセスがデータの交換と実行の同期を行うための非常に広範なメカニズムがあります。これらのメカニズムには、次のものがあります。

- パイプ — 匿名のデータ待ち行列
- 名前付きパイプ — ファイル名の付いたデータ待ち行列
- System V メッセージ待ち行列、セマフォ、および共用メモリ
- POSIX メッセージ待ち行列、セマフォ、および共用メモリ
- シグナル — ソフトウェア生成割り込み
- ソケット
- マッピングメモリとマッピングファイル (詳細は、63ページの「メモリ管理インタフェース」を参照)

この章では、マッピングメモリを除いたこれらのメカニズムについて説明します。

パイプ

2 つのプロセスの間のパイプは、親プロセスで作成されているファイルのペアです。パイプは、親プロセスがフォークしたときの結果のプロセスを接続します。パイプは、ファイル名空間には存在しないため、「匿名」と言います。パイプは通

常、2 つのプロセスだけを接続しますが、任意の数の子プロセスを相互に、または親プロセスと 1 つのパイプだけで接続することもできます。

パイプは、親プロセスで pipe(2) 呼び出しを使用し作成されます。pipe(2) は 引数の配列に 2 つのファイル記述子を戻します。フォーク後、両方のプロセスは p[0] を使用して読み取り、p[1] を使用して書き込みます。プロセスは、実際にはそれらのために管理されている循環バッファとの間で読み取りと書き込みを行います。

fork(2) を使用すると、プロセスごとの開かれているファイルのテーブルが複写されるので、 各プロセスは 2 つのリーダと 2 つのライタを持ちます。パイプを適切に機能させるには、余分なリーダとライタを閉じなければなりません。 たとえば、同じプロセスによる書き込みのためにリーダのもう一方の終わりも開いていると、ファイルの終わり指標は戻されません。 次のコードは、パイプの作成、フォーク、および重複したパイプの終わりのクリアを示しています。

```
#include <stdio.h>
#include <unistd.h>
 ...
  int p[2];
 ...
  if (pipe(p) == -1) exit(1);
  switch( fork() )
  {
   case 0:      /* in child */
    close( p[0] );
    dup2( p[1], 1);
    close P[1] );
    exec( ... );
    exit(1);
   default:      /* in parent */
    close( p[1] );
    dup2( P[0], 0 );
    close( p[0] );
    break;
  }
  ...
```

表 8–1 は、一定の条件下でのパイプからの読み取りとパイプへの書き込みの結果を示しています。

表 8–1 パイプでの読み取りと書き込みの結果

実行	条件	結果
読み取り	空のパイプ、ライタ接続	読み取りはブロッキングされる
書き込み	フルのパイプ、ライタ接続	書き込みはブロッキングされる

表 8–1 パイプでの読み取りと書き込みの結果 続く

実行	条件	結果
読み取り	空のパイプ、接続ライタなし	EOF が戻される
書き込み	リーダなし	SIGPIPE

fcntl(2) を記述子に呼び出して FNDELAY を設定すると、ブロッキングを阻止できます。この状態で入出力関数の呼び出しを行うと、errno に EWOULDBLOCK が設定され、エラー (-1) が戻されます。

名前付きパイプ

名前付きパイプは、パイプとほぼ同じように機能しますが、名前の付いた実体としてファイルシステムに作成されます。こうすると、フォークによって関係付けられた任意のプロセスでパイプを無条件に開くことができます。名前付きパイプは、mknod(2) の呼び出し によって作成されます。その後、適当なアクセス権を持つ任意のプロセスで、名前付きパイプの読み取りと書き込みを実行できます。

open(2) の呼び出しでは、パイプを開くプロセスは、もう 1 つのプロセスもパイプを開くまでブロッキングします。

ブロッキングせずに名前付きパイプを開くために、open() の呼び出し時に (<sys/fcntl.h> にある) O_NDELAY マスクを、選択されているファイルモードマスクと論理和を取ることができます。 open(2) を呼び出したときに他のどのプロセスもパイプと接続していない場合は、errno に EWOULDBLOCK が設定され -1 が戻されます。

ソケット

ソケットは、2 つのプロセス間のポイントツーポイントの双方向通信を提供します。ソケットは、非常に多くの目的に使用でき、プロセス間およびシステム間通信

の基本構成要素です。ソケットは、名前を結合できる通信の終端です。ソケットは、1 つの型と 1 つ以上の関連プロセスを持ちます。

ソケットのアドレス空間

ソケットは通信ドメインに存在します。ソケットドメインは、アドレッシング構造と一連のプロトコルを提供する抽象的なものです。ソケットは、同じドメイン内のソケットとだけ接続します。23 個のソケットドメインが識別されていて (<sys/socket.h> を参照)、Solaris 7 およびその互換オペレーティング環境では通常は UNIX ドメインとインターネットドメインだけが使用されます。

ソケットは、IPC の他の形態と同様に、1 つのシステム上のプロセス間の通信に使用できます。UNIX ドメイン (AF_UNIX) は、1 つのシステム上のソケットアドレス空間を提供します。UNIX ドメインのソケットは、UNIX パスで名前を指定されます。

ソケットは、異なるシステムにあるプロセス間の通信に使用することもできます。接続されているシステム間のソケットアドレス空間をインターネットドメイン (AF_INET) と言います。インターネットドメイン通信は、TCP/IP インターネットプロトコルを使用します。

ソケットのタイプ

ソケットのタイプは、アプリケーションに見える通信属性を定義します。プロセスは、同じタイプのソケット間だけで通信します。ソケットには次の 5 種類のタイプがあります。

- ストリームソケットは、レコード境界のない双方向の逐次で信頼でき重複しないデータフローを提供します。ストリームは、電話の会話とほとんど同じように働きます。ソケットタイプは SOCK_STREAM であり、インターネットドメインでは伝送制御プロトコル (TCP) を使用します。
- データグラムソケットは、双方向のメッセージフローをサポートします。データグラムソケットは、メッセージが送られた順序とは異なる順番でメッセージを受け取ることができます。データの中のレコード境界は保たれます。データグラムソケットは、郵便でやりとりする手紙の受け渡しとほとんど同じように働きます。ソケットタイプは SOCK_DGRAM であり、インターネットドメインではユーザデータグラムプロトコル (UDP) を使用します。

- 逐次パケットソケットは、固定した最大長のデータグラムに双方向で逐次的な信頼できる接続を提供します。ソケットタイプは、SOCK_SEQPACKET です。このタイプのプロトコルは、プロトコルファミリとしては実装されていません。
- ローソケットは、基礎となる通信プロトコルへのアクセスを提供します。このソケットは、通常はデータグラムが中心ですが、その正確な特性はプロトコルが提供するインタフェースによって異なります。

ソケットの作成と名前の指定

socket(3N) を呼び出して、指定したドメインに指定したタイプのソケットを作成します。プロトコルを指定しないと、システムは指定されたソケットタイプをサポートしているプロトコルをデフォルトとして使用します。ソケットハンドル (記述子) が戻されます。

リモートプロセスは、アドレスが結合されるまでソケットを識別する方法を持ちません。通信するプロセスは、アドレスによって接続します。UNIX ドメインでは、接続は通常は 1 つまたは 2 つのパス名から構成されます。インターネットドメインでは、接続はローカルアドレス、リモートアドレス、ローカルポート、リモートポートから構成されます。ほとんどのドメインでは、接続は一意でなければなりません。

bind(3N) を呼び出して、パスまたはインターネットアドレスをソケットに結合します。 bind(3N) を呼び出すには、ソケットのドメインに応じて、3 つの異なる方法があります。パスが 14 文字以下の UNIX ドメインソケットでは、次のようにします。

```
#include <sys/socket.h>
 ...
  bind (sd, (struct sockaddr *) &addr, length);
```

UNIX ドメインソケットのパスが 14 文字よりも多くの文字を必要とする場合は、次のようにします。

```
#include <sys/un.h>
 ...
  bind (sd, (struct sockaddr *) &addr, length);
```

インターネットドメインソケットでは、次のようにします。

```
#include <netinet/in.h>
 ...
  bind (sd, (struct sockaddr *) &addr, length);
```

UNIX ドメインで名前を結合すると、名前付きソケットがファイルシステムに作成されます。ソケットを削除するには、unlink(2) または rm(1) を使用します。

ストリームソケットの接続

ソケットの接続は、通常は対称的ではありません。1 つのプロセスが通常はサーバとして動作し、もう 1 つのプロセスはクライアントとして動作します。サーバは、以前に合意しているパスまたはアドレスにソケットを結合します。その後、ソケットでブロッキングします。SOCK_STREAM ソケットでは、サーバは listen(3N) を呼び出し、待ち行列に並べられる接続要求の個数を指定します。

クライアントは connect(3N) を呼び出して、サーバのソケットへの接続を開始します。UNIX ドメインの呼び出しは、次のようになります。

```
struct sockaddr_un server;
 ...
  connect (sd, (struct sockaddr_un *)&server, length);
```

インターネットドメインの呼び出しは、次のようになります。

```
struct sockaddr_in;
 ...
  connect (sd, (struct sockaddr_in *)&server, length);
```

クライアントのソケットが接続呼び出しの時点で結合されていないと、自動的に名前が選択されてソケットに結合されます。

SOCK_STREAM ソケットでは、サーバは accept(3N) を呼び出して接続を完了します。accept(3N) は、特定の接続だけに有効である新しいソケット記述子を戻します。サーバは、一度に複数の SOCK_STREAM 接続をアクティブにできます。

ストリームデータ転送

SOCK_STREAM ソケットとの間でデータを送受信する関数には、write(2)、read(2)、send(3N)、および recv(3N) があります。send(3N) と recv(3N) は、read(2) と write(2) によく似ていますが、いくつかの動作フラグが追加されています。

SOCK_STREAM ソケットは、close(2) の呼び出しによって廃棄されます。

データグラムソケット

データグラムソケットは、接続の確立を必要としません。各メッセージが受信先アドレスを運びます。特定のローカルアドレスが必要な場合は、bind(3N) の呼び出しをデータ転送の前に常に実行しなければなりません。データは、sendto(3N) または sendmsg(3N)(send(3N) のマニュアルページを参照) の呼び出しによって送

られます。sendto(3N) の呼び出しは send(3N) の呼び出しと同様ですが、受信先アドレスも指定します。

データグラムソケットメッセージを受信するには、recvfrom(3N) または recvmsg(3N)(recv(3N) のマニュアルページを参照) を呼び出します。recv(3N) では着信データ用に 1 つのバッファが必要ですが、recvfrom(3N) では着信メッセージ用と発信元アドレスを受け取るため用に 2 つのバッファが必要です。

データグラムソケットは、connect(3N) を使用しても、ソケットを指定された受信先ソケットに接続できます。接続するときは、 send(3N) と recv(3N) を使用してデータを送受信します。

accept(3N) と listen(3N) は、データグラムソケットでは使用しません。

ソケットオプション

ソケットは、getsockopt(3N) でフェッチして setsockopt(3N) で設定できるいくつかのオプションを持っています。これらの関数は、固有のソケットレベル (*level* = SOL_SOCKET) で使用できますが、ソケットオプション名を指定しなければなりません。その他のレベルでオプションを操作するには、オプションを制御するプロトコルの番号を指定しなければなりません (詳細は、getprotoent(3N) についての記述を参照してください)。

POSIX IPC

POSIX プロセス間通信は、System V プロセス間通信の変形です。これは Solaris 2.6 で新しく追加されました。System V オブジェクトと同様に、POSIX IPC オブジェクトは、所有者、所有者のグループ、およびその他に読み取り権と書き込み権があります (実行権はありません)。POSIX IPC オブジェクトの所有者が、そのオブジェクトの所有者を変更する方法はありません。

System V IPC インタフェースとは異なり、POSIX IPC インタフェースはすべてマルチスレッドに対して安全です。

POSIX メッセージ

POSIX メッセージ待ち行列インタフェースは、次のとおりです。

mq_open(3R)	名前付きメッセージ待ち行列に接続する。指定によっては作成する。
mq_close(3R)	開いているメッセージ待ち行列への接続を終了する。
mq_unlink(3R)	開いているメッセージ待ち行列への接続を終了し、最後のプロセスが待ち行列を閉じるときに待ち行列を削除する。
mq_send(3R)	メッセージを待ち行列に入れる。
mq_receive(3R)	最も古い最高優先順位メッセージを待ち行列から受け取る(削除する)。
mq_notify(3R)	メッセージが待ち行列で使用できることをプロセスまたはスレッドに通知する。
mq_setattr(3R)	メッセージ待ち行列属性を設定または取得する。
mq_getattr(3R)	

POSIX セマフォ

POSIX セマフォは、System V セマフォより軽量です。POSIX セマフォ構造体は 25 個までのセマフォの配列ではなく、1 つのセマフォだけを定義します。

POSIX セマフォインタフェースは、次のとおりです。

sem_open(3R)	名前付きセマフォに接続する。指定によっては作成する。
sem_init(3R)	名前なしセマフォを初期化する。
sem_close(3R)	開いているセマフォへの接続を終了する。
sem_unlink(3R)	開いているセマフォへの接続を終了し、最後のプロセスがセマフォを閉じるときにセマフォを削除する。
sem_destroy(3R)	sem_init(3R) で初期化された名前なしセマフォを破棄する。
sem_getvalue(3R)	セマフォの値を指定された整数にコピーする。

sem_wait(3R)	セマフォが他のプロセスによって保持されている場合に、ブロッキングするかエラーを戻す。
sem_trywait(3R)	
sem_post(3R)	セマフォのカウントを増やす。

POSIX 共用メモリ

POSIX 共用メモリは、実際にはマッピングされているメモリの変形です (詳細は、63 ページの「マッピングの作成と使用」を参照) してください。主な違いは、(open(2) を呼び出す代わりに) shm_open(3R) を使用して共用メモリオブジェクトを開いて、(オブジェクトを削除しない close(2) を呼び出す代わりに) shm_unlink(3R) を使用してオブジェクトを閉じて削除することです。shm_open(3R) のオプションは、open(2) で提供されているオプションの数よりかなり少なくなっています。

System V IPC

Solaris 7 およびその互換オペレーティング環境では、パイプや名前付きパイプよりも多目的に使用できる 3 種類のプロセス間通信をサポートしているプロセス間通信 (IPC) パッケージが提供されています。

- メッセージでは、プロセスが書式付きデータを任意のプロセスに送信できます。
- セマフォでは、プロセスが実行の同期を取ることができます。
- 共用メモリでは、複数のプロセスがそれぞれの仮想アドレス空間の一部を共用できます。

System V IPC の詳細は、ipcrm(1)、ipcs(1)、Intro(2)、msgctl(2)、msgget(2)、msgrcv(2)、msgsnd(2)、semget(2)、semctl(2)、semop(2)、shmget(2)、shmctl(2)、shmop(2)、および ftok(3C) のマニュアルページを参照してください。

アクセス権

メッセージ、セマフォ、および共用メモリは、通常ファイルと同じように、所有者、グループ、およびその他のための読み取り権と書き込み権を持っています (実行

権は持っていません)。ファイルと同様に、作成元プロセスはデフォルトの所有者を識別します。ファイルとは異なり、作成者は機能の所有権を別のユーザに割り当てることができます。また、所有権割り当てを取り消すこともできます。

IPC 機能、キー引数、および作成フラグ

IPC 機能へのアクセスを要求するプロセスは、識別できなければなりません。これを行うために、IPC 機能を初期化する関数または IPC 機能へのアクセスを提供する関数は、`key_t` キー引数を使用します。キーは、任意の値または実行時に共通の元になる値から導き出すことができる値です。1 つの方法は、`ftok(3C)` を使用することです。`ftok(3C)` は、ファイル名をシステムで固有のキー値に変換します。

メッセージ、セマフォ、および共用メモリの初期化やアクセスを行う関数は、`int` 型の ID 番号を戻します。IPC 機能の読み取り、書き込み、および制御操作を行う関数は、この ID を使用します。

キー引数に `IPC_PRIVATE` を指定して関数を呼び出すと、作成プロセス専用の IPC 機能のインスタンスが新しく初期化されます。

フラグ引数として `IPC_CREAT` フラグを指定すると、関数はその IPC 機能が存在していない場合は新しく作成しようとします。

`IPC_CREAT` と `IPC_EXCL` の両方のフラグを指定して関数を呼び出した場合、その関数は IPC 機能がすでに存在していると失敗します。これは 2 つ以上のプロセスが IPC 機能を初期化する可能性がある場合に便利です。たとえば、複数のサーバプロセスが同じ IPC 機能にアクセスしようとする場合です。サーバプロセスがすべて `IPC_EXCL` を指定して IPC 機能を作成しようとすると、最初のプロセスだけが成功します。

この 2 つのフラグをどちらも指定しない場合、IPC 機能がすでに存在していれば、アクセスした関数は単にその機能の ID を戻します。`IPC_CREAT` を指定しない場合、該当する機能がまだ初期化されていなければ、呼び出しは失敗します。

これらの制御フラグは、論理 (ビット単位) `OR` を使用して 8 進数値アクセス権モードと組み合わせるとフラグ引数を作成できます。たとえば、次の例では、メッセージ待ち行列が存在していない場合は新しい待ち行列を初期化します。

```
msqid = msgget(ftok("/tmp", 'A'), (IPC_CREAT | IPC_EXCL | 0400));
```

最初の引数は、文字列 (`"/tmp"`) に基づいてキー (次の 'A') と評価されます。2 番目の引数は、アクセス権と制御フラグが組み合わされたものと評価されます。

System V メッセージ

プロセスがメッセージを送受信できるようにするには、msgget(2) 関数によって待ち行列を初期化しなければなりません。待ち行列の所有者または作成者は、msgctl(2) を使用して、その所有権またはアクセス権を変更できます。また、そうするためのアクセス権を持つプロセスは、制御操作のために msgctl(2) を使用することもできます。

IPC メッセージを使用すると、プロセスはメッセージを送受信し、メッセージを任意の順序で処理待ち行列に入れることができます。パイプで使用されるファイルバイトストリームのモデルによるデータフローとは異なり、IPC メッセージでは長さが明示されます。

メッセージには特定のタイプを割り当てることができます。このため、サーバプロセスは、クライアント PID をメッセージタイプとして使用して、その待ち行列上のクライアント間にメッセージトラフィックを振り向けることができます。単一メッセージトランザクションでは、複数のサーバプロセスは、共用メッセージ待ち行列に送られるトランザクション群に対して、並行して働くことができます。

メッセージの送受信操作は、それぞれ msgsnd(2) 関数と msgrcv(2) 関数によって実行されます。メッセージが送信されると、そのテキストがメッセージ待ち行列にコピーされます。msgsnd(2) 関数と msgrcv(2) 関数は、ブロッキング操作としても非ブロッキング操作としても実行できます。ブロッキングされたメッセージ操作は、次の条件のどれかが生じるまで中断されます。

- 呼び出しが成功した
- プロセスがシグナルを受信した
- 待ち行列が削除された

メッセージ待ち行列の初期化

msgget(2) 関数は、新しいメッセージ待ち行列を初期化します。また、key 引数に対応する待ち行列のメッセージ待ち行列 ID (msgid) を戻します。msgflg 引数として渡される値は、待ち行列アクセス権と制御フラグを設定する 8 進整数でなければなりません。

MSGMNI カーネル構成オプションは、カーネルがサポートする固有のメッセージ待ち行列の最大数を指定します。この制限を越えると、msgget() 関数は失敗します。次に、msgget() 関数の使用例を示します。

```
#include <sys/ipc.h>
 #include <sys/msg.h>
```

```
...
 key_t key;  /* msgget() に渡す key */
 int msgflg, /* msgget() に渡す msgflg */
   msqid; /* msgget() からの戻り値 */  ...
 key = ...
 msgflg = ...
 if ((msqid = msgget(key, msgflg)) == -1)
 {
  perror("msgget: msgget failed");
  exit(1);
 } else
  (void) fprintf(stderr, "msgget succeeded");
 ...
```

メッセージ待ち行列の制御

msgctl(2) 関数は、メッセージ待ち行列のアクセス権やその他の特性を変更します。msgid 引数は、既存のメッセージ待ち行列の ID でなければなりません。cmd 引数は、次のいずれか 1 つです。

IPC_STAT	待ち行列の状態についての情報を buf が指すデータ構造体に入れる。この呼び出しを行うには、プロセスが読み取り権を持っていなければならない。
IPC_SET	所有者のユーザ ID とグループ ID、アクセス権、およびメッセージ待ち行列の大きさ (バイト数) を設定する。この呼び出しを行うには、プロセスが所有者、作成者、またはスーパーユーザの有効なユーザ ID を持っていなければならない。
IPC_RMID	msgid 引数で指定したメッセージ待ち行列を削除する。

次に、msgctl(2) 関数に各種のフラグをすべて付けて使用する例を示します。

```
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/msg.h>

  ...
  if (msgctl(msqid, IPC_STAT, &buf) == -1)  {
   perror("msgctl: msgctl failed");
   exit(1);
  }
  ...
  if (msgctl(msqid, IPC_SET, &buf) == -1) {
   perror("msgctl: msgctl failed");
   exit(1);
  }
  ...
```

メッセージの送受信

msgsnd(2) と msgrcv(2) 関数は、それぞれメッセージを送受信します。msgid 引数は、既存のメッセージ待ち行列の ID でなければなりません。msgp 引数は、メッセージのタイプとテキストを含んでいる構造体へのポインタです。msgsz 引数は、メッセージの長さをバイト数で指定します。msgflg 引数には、様々な制御フラグを渡すことができます。

次に、msgsnd(2) と msgrcv(2) の使用例を示します。

```
#include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/msg.h>

 ...
  int    msgflg; /* 操作用のメッセージフラグ */
  struct msgbuf  *msgp; /* メッセージバッファへのポインタ */
  size_t   msgsz; /* メッセージの長さ */
  size_t   maxmsgsize;
  long    msgtyp; /* 希望するメッセージタイプ */
  int    msqid  /* 使用するメッセージ待ち行列の ID */
  ...
  msgp = malloc(sizeof(struct msgbuf) - sizeof (msgp->mtext)
       + maxmsgsz);
  if (msgp == NULL) {
   (void) fprintf(stderr, "msgop: %s %ld byte messages.\n",
     "could not allocate message buffer for", maxmsgsz);
   exit(1);
   ...
   msgsz = ...
   msgflg = ...
   if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
    perror("msgop: msgsnd failed");
   ...
   msgsz = ...
   msgtyp = first_on_queue;
   msgflg = ...
   if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
    perror("msgop: msgrcv failed");
   ...
```

System V セマフォ

セマフォを使用すると、プロセスは状態情報を問い合わせたり、変更したりできます。通常、セマフォは共用メモリセグメントなどのシステム資源が利用可能かどうかを監視して制御するために使用されます。セマフォは、個々のユニットまたはセット内の要素として操作できます。

System V IPC セマフォは、大きな配列の中に存在できるため、極めて重量級です。より軽量のセマフォをスレッドライブラリ (semaphore(3T) のマニュアルページを参照) と POSIX セマフォ (74ページの「POSIX セマフォ」を参照) で使用できま

す。スレッドライブラリセマフォは、マッピングされたメモリ (63ページの「メモリ管理インタフェース」を参照) と一緒に使用しなければなりません。

セマフォのセットは、制御構造体と個々のセマフォの配列から成ります。デフォルトでは、25 個までの要素を持つことができます。セマフォのセットは、semget(2) を使用して初期化しなければなりません。セマフォ作成者は semctl(2) を使用して、その所有権またはアクセス権を変更できます。アクセス権を持つプロセスは、semctl(2) を使用して操作を制御できます。

セマフォの操作は、semop(2) 関数によって行います。この関数は、セマフォ操作構造体の配列へのポインタを受け入れます。操作配列内の各構造体は、セマフォに実行する操作についてのデータを持ちます。読み取り権を持つプロセスは、セマフォがゼロ値を持っているかどうかを検査できます。セマフォを増分または減分する操作には、書き込み権が必要です。

要求された操作が失敗すると、どのセマフォも変更されません。IPC_NOWAIT フラグが設定されている場合を除いて、プロセスはブロッキングし、次のいずれかが生じるまでブロッキングされたままです。

- セマフォ操作がすべて終了して呼び出しが成功した
- プロセスがシグナルを受信した
- セマフォのセットが削除された

セマフォを更新できるのは、一度に 1 つのプロセスだけです。異なるプロセスが同時に要求した場合は、任意の順序で処理されます。操作の配列が semop(2) 呼び出しによって与えられると、配列内のすべての操作が正常に終了できるまで更新されません。

セマフォを排他的に使用しているプロセスが異常終了した後、操作を取り消すかセマフォを解放しなければ、セマフォはメモリ内にロッキングされたままになります。これを防止するため、semop(2) は SEM_UNDO 制御フラグにセマフォ操作ごとに、それぞれアンドゥ構造体を割り当てます。この構造体には、セマフォを以前の状態に戻すために必要な情報があります。プロセスが異常終了すると、アンドゥ構造体内の操作がシステムによって適用されます。このようにすれば、プロセスが異常終了しても、セマフォが整合性のない状態になってしまうことはありません。

プロセスがセマフォによって制御される資源へのアクセスを共用する場合は、SEM_UNDO を有効にしてセマフォに対する操作を行なってはいけません。現在、資源を制御しているプロセスが異常終了すると、その資源は整合性のない状態になったと見なされます。別のプロセスがこの資源を整合性のある状態に復元するためには、そのことを認識できなければなりません。

SEM_UNDO を有効にしてセマフォ操作を実行するときは、取り消し操作を行う呼び出しについても SEM_UNDO を有効にしておかなければなりません。プロセスが正常に実行されると、取り消し操作はアンドゥ構造体に補数値を補って更新します。このため、プロセスが異常終了しない限り、アンドゥ構造体に適用された値は最終的に取り消されて 0 になります。アンドゥ構造体は 0 になると削除されます。

SEM_UNDO を整合性なく使用すると、システムが再起動されるまで割り当てられたアンドゥ構造体が削除されないことがあるので、資源の過剰消費につながります。

セマフォのセットの初期化

semget(2) は、セマフォの初期化またはセマフォへのアクセスを行います。呼び出しが成功すると、セマフォ ID (semid) を戻します。key 引数は、セマフォ ID に関連付けられた値です。nsems 引数は、セマフォ配列内の要素数を指定します。nsems が既存の配列の要素数を超える呼び出しは失敗します。正しい数がわからない場合は、この値を 0 に指定すると正しく実行されます。semflg 引数は、初期状態のアクセス権と作成の制御フラグを指定します。

SEMMNI システム構成オプションは、配列内のセマフォの最大許容数を指定します。SEMMNS オプションは、すべてのセマフォのセットを通じて個々のセマフォの最大数を指定します。セマフォのセット間のフラグメンテーションのため、利用できるすべてのセマフォを割り当てられない場合もあります。

次の例は、semget() 関数を示しています。

```
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/sem.h>
 ...
  key_t  key;   /* semget() に渡す key */
  int  semflg;   /* semget() に渡す semflg */
  int  nsems;   /* semget() に渡す nsems */
  int  semid;   /* semget() からの戻り値 */
  ...
  key = ...
  nsems = ...
  semflg = ...
  ...
  if ((semid = semget(key, nsems, semflg)) == -1) {
   perror("semget: semget failed");
   exit(1);
  } else
   exit(0);
 ...
```

セマフォの制御

semctl(2) は、セマフォのセットのアクセス権とその他の特性を変更します。semctl(2) は、 有効なセマフォ ID を指定して呼び出さなければなりません。semnum 値は、そのインデックスによって配列内のセマフォを選択します。cmd 引数は、次のいずれかの制御フラグです。

GETVAL	単一セマフォの値を戻す。
SETVAL	単一セマフォの値を設定する。この場合には、arg は int の arg.val と解釈される。
GETPID	セマフォまたは配列に対して最後に操作を実行したプロセスの PID を戻す。
GETNCNT	セマフォの値が増加するのを待っているプロセス数を戻す。
GETZCNT	特定のセマフォの値が 0 に達するのを待っているプロセス数を戻す。
GETALL	セット内のすべてのセマフォの値を戻す。この場合には、arg は unsigned short の配列へのポインタ arg.array と解釈される。
SETALL	セットにあるすべてのセマフォに値を設定する。この場合、arg は unsigned short の配列へのポインタ arg.array と解釈される。
IPC_STAT	制御構造体からセマフォのセットの状態情報を取得し、それを semid_ds 型のバッファへのポインタ arg.buf が指すデータ構造体に入れる。
IPC_SET	有効なユーザおよびグループの識別子とアクセス権を設定する。この場合、arg は arg.buf と解釈される。
IPC_RMID	指定したセマフォのセットを削除する。

IPC_SET または IPC_RMID コマンドを実行するには、所有者、作成者、またはスーパーユーザとして有効なユーザ識別子を持っていなければなりません。その他の制御コマンドには、読み取り権と書き込み権が必要です。

次に、semctl(2) の使用例を示します。

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
 ...
  register int    i;
 ...
  i = semctl(semid, semnum, cmd, arg);
```

```
 if (i == -1) {
  perror("semctl: semctl failed");
  exit(1);
...
```

セマフォの操作

semop(2) は、セマフォのセットへの操作を実行します。semid 引数は、前の semget() 呼び出しによって戻されたセマフォ ID です。sops 引数は、セマフォ操作について次のような情報を含んでいる構造体の配列へのポインタです。

- セマフォ番号
- 実行する操作
- 制御フラグ (存在する場合)

sembuf 構造体は、<sys/sem.h> に定義されているセマフォ操作を指定します。nsops 引数は配列の長さを指定します。配列の最大長は、SEMOPM 構成オプションで指定されます。 これは単一の semop(2) 呼び出しで許される最大操作数で、デフォルトでは 10 に設定されています。

実行する操作は、次のように判定されます。

- 正の整数の場合は、セマフォの値をそれだけ増加します。
- 負の整数の場合は、セマフォの値をそれだけ減少します。セマフォをゼロより小さい値に設定しようとすると、IPC_NOWAIT が有効であるかどうかに応じて、失敗するかブロッキングされます。
- 値がゼロの場合は、セマフォの値がゼロになるのを待ちます。

semop(2) では、次の 2 つの制御フラグを使用できます。

IPC_NOWAIT	配列内のどの操作についても設定できる。この関数は、IPC_NOWAIT が設定されている操作が正しく実行できなかった場合に、セマフォの値を変更せずに戻る。セマフォを現在の値より多く減らそうしたり、セマフォがゼロでないときにゼロかどうか検査しようとすると関数は失敗する。
SEM_UNDO	プロセスの終了時に配列内の個々の操作を取り消す。

次に、semop(2) 関数の使用例を示します。

```
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
 ...
  int    i;   /* 作業領域 */
  int    nsops;   /* 実行する操作数 */
```

```
  int    semid;   /* セマフォのセットの ID */
  struct sembuf    *sops;   /* 実行する操作へのポインタ */
  ...
  if ((i = semop(semid, sops, nsops)) == -1) {
   perror("semop: semop failed");
  } else
   (void) fprintf(stderr, "semop: returned %d\n", i);
 ...
```

System V 共用メモリ

Solaris 7 オペレーティングシステムで共用メモリアプリケーションを実装するには、mmap(2) 関数と固有仮想記憶管理を利用する方法が最も効率的です。第 7 章を参照してください。

Solaris 7 では、System V 共用メモリもサポートされます。共用メモリを使用すると、複数のプロセスが同時に物理メモリの 1 つのセグメントを自分の仮想アドレス空間に接続できますが、効率は低下します。複数のプロセスに書き込みアクセスが許可されているときは、セマフォなどの外部のプロトコルや機構を使用して、不一致や衝突などを防止できます。

プロセスは、shmget(2) を使用して共用メモリセグメントを作成します。この呼び出しは、 既存の共用セグメントの ID を取得する際にも使用できます。作成プロセスは、セグメントのアクセス権と大きさ (バイト数) を設定します。

共用メモリセグメントの元の所有者は、shmctl(2) を使用して所有権を他のユーザに割り当てることができます。この関数では割り当てを取り消すこともできます。適切なアクセス権を持っていれば、他のプロセスも shmctl(2) を使用して共用メモリセグメントに様々な制御機能を実行できます。

共用メモリセグメントを作成すると、shmat(2) を使用してプロセスのアドレス空間に接続できます。切り離すには shmdt(2) を使用します。接続するプロセスが shmat(2) のための適当なアクセス権を持っていなければなりません。接続してしまうと、プロセスは接続操作で要求されているアクセス権に従って、セグメントの読み取りまたは書き込みを実行できます。共用セグメントは、同じプロセスによって何回でも接続できます。

共用メモリセグメントは、物理メモリ内のある領域を指す一意の ID を持つ制御構造体から成ります。セグメント識別子を shmid と言います。共用メモリセグメントの制御構造体は、<sys/shm.h> にあります。

共用メモリセグメントのアクセス

shmget(2) を使用して、共用メモリセグメントへアクセスします。呼び出しが成功すると、共用メモリセグメント ID (shmid) を戻します。次に、shmget(2) の使用例を示します。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
 ...
  key_t  key;   /* shmget() に渡す key */
  int  shmflg;  /* shmget() に渡す shmflg */
  int  shmid;  /* shmget() からの戻り値 */
  size_t size;   /* shmget() に渡す大きさ */
  ...
  key = ...
  size = ...
  shmflg) = ...
  if ((shmid = shmget (key, size, shmflg)) == -1) {
   perror("shmget: shmget failed");
   exit(1);
  } else {
   (void) fprintf(stderr,
      "shmget: shmget returned %d\n", shmid);
   exit(0);
  }
  ...
```

共用メモリセグメントの制御

shmctl(2)を使用して、共用メモリセグメントのアクセス権とその他の特性を変更します。cmd 引数は、次の制御コマンドのいずれか 1 つです。

SHM_LOCK	指定したメモリ内の共用メモリセグメントをロッキングする。このコマンドを実行するプロセスは、有効なスーパーユーザの ID を持っていなければならない。
SHM_UNLOCK	共用メモリセグメントのロッキングを解除する。このコマンドを実行するプロセスは、有効なスーパーユーザの ID を持っていなければならない。
IPC_STAT	制御構造体にある状態情報を取得して、buf が指すバッファに入れる。このコマンドを実行するプロセスは、セグメントの読み取り権を持っていなければならない。

IPC_SET	有効なユーザおよびグループの識別子とアクセス権を設定する。このコマンドを実行するプロセスは、所有者、作成者、またはスーパーユーザの有効な ID を持っていなければならない。
IPC_RMID	共用メモリセグメントを削除する。このコマンドを実行するプロセスは、所有者、作成者、またはスーパーユーザの有効な ID を持っていなければならない。

次に、shmctl(2) の使用例を示します。

```
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>
 ...
  int  cmd;  /* shmctl() のためのコマンドコード */
  int  shmid;  /* セグメント ID */
  struct shmid_ds  shmid_ds;   /* 結果を保持するための共用メモリデータ構造体 */
  ...
  shmid = ...
  cmd = ...
  if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
   perror("shmctl: shmctl failed");
   exit(1);
  ...
```

共用メモリセグメントの接続と切り離し

shmat() と shmdt() (shmop(2) を参照) を使用して、共用メモリセグメントの接続と切り離しを行います。shmat(2) は、共用セグメントの先頭へのポインタを戻します。shmdt(2) は、*shmaddr* で指定されたアドレスから共用メモリセグメントを切り離します。次に、shmat(2) と shmdt(2) の呼び出しの使用例を示します。

```
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>

static struct state { /* 接続されるセグメントの内部レコード */
  int  shmid;  /* 接続されるセグメントの ID */
  char  *shmaddr; /* 接続点 */
  int  shmflg;  /* 接続時に使用されるフラグ */
  } ap[MAXnap];   /* 接続されている現在のセグメントの状態 */
  int  nap;   /* 現在接続されているセグメント数 */
 ...
  char    *addr;   /* アドレス用の作業変数 */
  register int    i;   /* 作業領域 */
  register struct state   *p;    /* 現在の状態エントリへのポインタ */
 ...
  p = &ap[nap++];
  p->shmid = ...
  p->shmaddr = ...
  p->shmflg = ...
  p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
  if(p->shmaddr == (char *)-1) {
   perror("shmat failed");
```

```
 nap--;
} else
  (void) fprintf(stderr, "shmop: shmat returned %p\n",
    p->shmaddr);
...
i = shmdt(addr);
if(i == -1) {
 perror("shmdt failed");
} else {
 (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);
 for (p = ap, i = nap; i--; p++) {
  if (p->shmaddr == addr) *p = ap[--nap];
 }
}
...
```

第 9 章

実時間プログラミングと管理

この章では、SunOS 5 で実行する実時間アプリケーションの書き方と移植方法について説明します。この章は、実時間アプリケーションを書いた経験があるプログラマと、実時間処理と Solaris システムに詳しい管理者向けに書かれています。

実時間アプリケーションの基本的な規則

実時間応答は、一定の条件を満たした場合に保証されます。この節ではその条件を明らかにし、問題を生じたりシステムを無効にしたりしてしまう重大な設計上のエラーをいくつか説明します。

ここでは、システムの応答時間を劣化させる可能性のある問題を取り上げます。その中にはワークステーションをハングさせてしまうものもあります。それほど重大ではないエラーとしては、優先順位の反転やシステムの過負荷 (処理が多すぎる) などがあります。

Solaris の実時間プロセスには、次のような特色があります。

- 93ページの「スケジューリング」で説明しているように、実時間スケジューリングクラスで動作します。
- 106ページの「メモリロッキング」で説明しているように、プロセスのアドレス空間内のすべてのメモリをロッキングします。
- 91ページの「共用ライブラリ」で説明しているように、静的にリンクされたプログラム、またはダイナミック結合が前もって完了しているプログラムから生じます。

この章では、実時間操作を単一スレッドのプロセスとして説明しますが、説明の内容はマルチスレッドプロセスにも当てはまります (マルチスレッドプロセスの詳細は、『マルチスレッドのプログラミング』を参照してください)。スレッドの実時間スケジューリングを保証するには、結合スレッドとして扱われなければならず、スレッドの LWP が実時間スケジューリングクラスで実行されなければなりません。メモリのロッキングと初期ダイナミック結合は、プロセス内のすべてのスレッドについて有効です。

プロセスが最も高い優先順位を持つ場合は、次のようになります。

- 実行可能になると保証されているディスパッチ中の潜在的な時間期間の間、プロセッサを得る (94ページの「ディスパッチ中の潜在的な時間」を参照)
- 最も優先順位が高い実行可能なプロセスである限り、実行を継続する

実時間プロセスは、システム上の他のイベントのために、プロセッサの制御を失ったり、プロセッサの制御を得られなかったりすることがあります。そのようなイベントの例としては、外部イベント (割り込みなど)、資源不足、外部イベント待ち (同期入出力)、より高い優先順位のプロセスによる横取りなどがあります。

実時間スケジューリングは通常、`open(2)` や `close(2)` など、システムの初期化と終了を行うサービスには適用されません。

応答時間の劣化

この節で説明する問題は程度は異なりますが、どれもシステムの応答時間を劣化させます。劣化が大きいと、アプリケーションがクリティカルデッドラインを逃してしまうことがあります。

実時間処理は、システム上で実時間アプリケーションを実行している他の有効なアプリケーションの操作に対しても大きな影響を及ぼします。実時間プロセスの優先順位は高いので、タイムシェアリングプロセスはかなりの時間、実行を妨げられます。表示に対してキーボードで応答するなどの対話型操作は著しく遅くなります。

システムの応答時間

SunOS 5 でのシステムの応答が、入出力イベントのタイミングを制限することはありません。これは、実行がタイムクリティカルなプログラムセグメントには、同期入出力呼び出しを入れてはいけないということです。時間制限が非常に長いプログラムセグメントでも、同期入出力を行わないでください。たとえば、大量の記憶領

域の入出力の際に読み取りや書き込み操作を行うと、その間システムはハングしてしまいます。

よくあるアプリケーションの誤りは、入出力を実行してエラーメッセージのテキストをディスクから取得することです。この処理は、独立した実時間ではないプロセスまたはスレッドから実行しなければなりません。

割り込みサービス

割り込みの優先順位は、プロセスの優先順位に左右されません。プロセスの優先順位を設定しても、プロセスの動作によって生じるハードウェア割り込みの優先順位は設定されません。つまり、実時間プロセスによって制御されるデバイスについての割り込み処理は、必ずしもタイムシェアリングプロセスによって制御される他のデバイスについての割り込み処理よりも前に実行されるとは限りません。

共用ライブラリ

タイムシェアリングプロセスでは、ダイナミックにリンクされる共用ライブラリを使用すると、メモリ量をかなり節約できます。このようなタイプのリンクは、ファイルマッピングの形で実装されます。ダイナミックにリンクされたライブラリルーチンは、暗黙の読み取りを行います。

実時間プログラムでは、プログラムを起動する際に、環境変数 `LD_BIND_NOW` を非 `NULL` に設定すると、共用ライブラリを使用してもダイナミック結合が行われないようにできます。このようにすると、プログラムの実行前にダイナミック結合が実行されます。詳細は、『リンカーとライブラリ』を参照してください。

優先順位の反転

実時間プロセスが必要とする資源を、タイムシェアリングプロセスが取得すると、実時間プロセスをブロッキングできます。優先順位の反転とは、優先順位の高いプロセスが優先順位の低いプロセスによってブロッキングされる際に生じる状態です。「ブロッキング」とは、あるプロセスが、1 つまたは複数のプロセスが資源の制御を手放すのを待っている状態を指します。このブロッキングが長引くと、たとえ低レベル資源についてでも、デッドラインを逃してしまうことがあります。

図 9–1 の場合を例にとると、優先順位の低いプロセスが共用資源を保持しているために、その共用資源を使用したい優先順位の高いプロセスがブロッキングされています。この優先順位の低いプロセスは、中間の優先順位を持つプロセスによって横

取りされます。この状態は長く続く場合があり、実際には優先順位の高いプロセスが資源を待たなければならない時間は、優先順位の低いプロセスによって危険領域が実行されている継続時間だけではなく、中間のプロセスがブロッキングされるまでの時間によって決まるので、どれだけ長く続くかわかりません。中間のプロセスは、いくつ関与していてもかまいません。

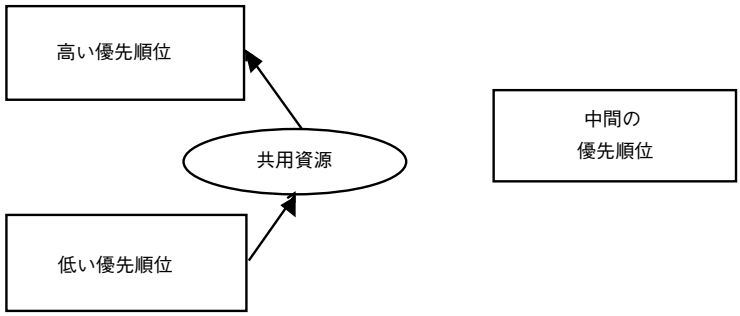

図 9–1　制限されない優先順位の反転

スティッキロッキング

ページは、ロッキングカウントが 65535 (0xFFFF) に達すると、メモリ内に永久にロッキングされます。0xFFFF の値は実装時に定義されており、将来のリリースで変更される可能性があります。このようにしてロッキングされたページのロッキングは解除できません。

ランナウェイ実時間プロセス

ランナウェイ実時間プロセスは、システムを停止させたり、システムが停止したように見えるほどシステムの応答を遅くしたりすることがあります。

注 - SPARC™ システム上にランナウェイプロセスがある場合は、Stop-A キーを押します。この手順を何回も繰り返さなければならない場合があります。この手順がうまく機能しない場合は、電源を切ってからしばらく待ち、もう一度電源を入れてください。

優先順位の高い実時間プロセスが CPU の制御を手放さない場合、無限ループを強制的に終了させなければ、システムの制御は得られません。このようなランナウェイプロセスは、Control-C キーを入力しても応答しません。ランナウェイプロセスよりも高い優先順に設定されているシェルを使用しようとしても失敗します。

入出力の特性

非同期入出力

非同期入出力操作がカーネルへの待ち行列に入った順序で行われるという保証はありません。また、非同期操作が実行された順序で呼び出し側に戻されるという保証もありません。

`aioread(3)` 呼び出しの高速シーケンスのために単一のバッファが指定されている場合、最初の呼び出しが行われてから最後の結果が呼び出し側にシグナルとして送信されるまでの間のバッファの状態については、保証されていません。

1 つの `aio_result_t` 構造体は、一度に 1 つの非同期読み取りまたは書き込みだけに使用してください。

実時間ファイル

SunOS 5 には、ファイルを確実に物理的に連続して割り当てる機能は用意されていません。

通常のファイルについては、`read(2)` と `write(2)` の操作は常にバッファリングされます。アプリケーションは `mmap(2)` と `msync(3C)` を使用して、二次記憶領域とプロセスメモリ間の入出力転送を直接実行できます。

スケジューリング

実時間スケジューリング制約は、データ取得やプロセス制御ハードウェアの管理のために必要です。実時間環境では、プロセスが制限された時間内で外部イベントに反応する必要があります。この制約は、処理する資源をタイムシェアリングプロセスのセットに「公平に」分配するように設計されているカーネルの能力を超えることがあります。

この節では、SunOS 5 の実時間スケジューラ、その優先順位待ち行列、およびスケジューリングを制御するシステムコールとユーティリティの使用方法について説明します。この節で説明している関数の詳細は、『 *man Pages(3): Library Routines*』を参照してください。

ディスパッチ中の潜在的な時間

実時間アプリケーションをスケジューリングする際に最も重要な要素は、実時間スケジューリングクラスを用意することです。標準のタイムシェアリングのスケジューリングクラスは、どのプロセスも平等に扱って優先順位の概念に制限があるので、実時間アプリケーションには適しません。実時間アプリケーションでは、プロセスの優先順位が絶対的なものとして受け取られ、アプリケーションの明示的な操作によってしか変更されないスケジューリングクラスが必要です。

ディスパッチ中の潜在的な時間とは、プロセスの操作開始の要求にシステムが応答するのにかかる時間を指します。アプリケーションの優先順位を尊重するように特別に作成されたスケジューラを使用すると、ディスパッチ中の潜在的な時間を制限した実時間アプリケーションを開発できます。

図 9–2 に、アプリケーションが外部イベントの要求に応答するのにかかる時間を示します。

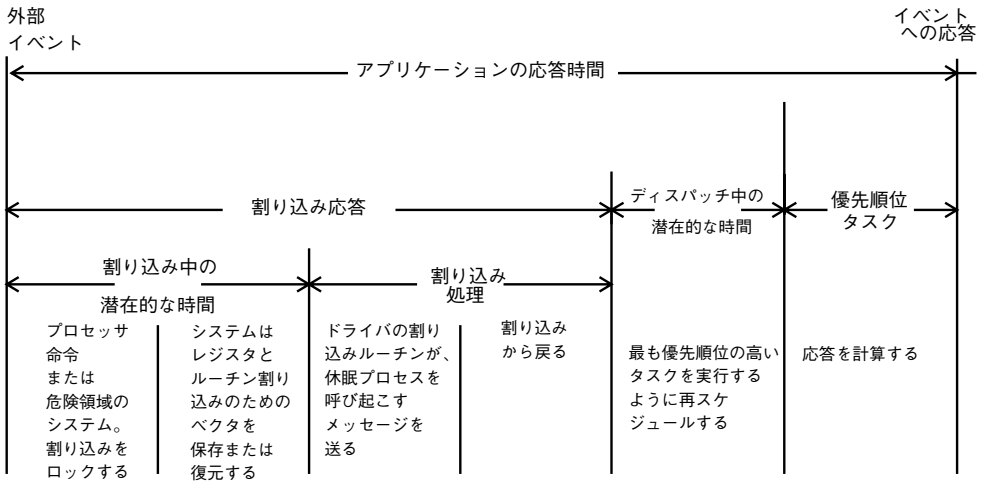

図 9–2　アプリケーションの応答時間

全体的なアプリケーションの応答時間には、割り込み応答時間、ディスパッチ中の潜在的な時間、およびアプリケーションが応答を決定するのにかかる時間が含まれます。

アプリケーションの割り込み応答時間には、システムの割り込み中の潜在的な時間とデバイスドライバの割り込み処理時間が含まれます。割り込み中の潜在的な時間は、システムが割り込みを無効にして実行しなければならない最長の間隔によって

決まります。これは SunOS 5 では、プロセッサの割り込みレベルの上昇を通常は要求しない同期プリミティブを使用して最小化されています。

割り込み処理中は、ドライバの割り込みルーチンが優先順位の高いプロセスを呼び起こして終了すると戻ります。システムでは、割り込まれたプロセスよりも高い優先順位を持つプロセスが現在ディスパッチ可能であることが検知され、そのプロセスをディスパッチするように指定されます。優先順位の低いプロセスから高いプロセスへコンテキストスイッチングする時間は、ディスパッチ中の潜在的な時間に含まれます。

図 9–3 に、システムが外部イベントに応答するのにかかる時間として定義された、システムの内部ディスパッチ中の潜在的な時間とアプリケーションの応答時間を示します。内部イベントのディスパッチ中の潜在的な時間は、あるプロセスがより高い優先順位のプロセスを呼び起こし、システムでそのプロセスがディスパッチされるのにかかる時間を表します。

アプリケーションの応答時間は、ドライバがより高い優先順位のプロセスを呼び起こし、優先順位の低いプロセスに資源を解放させ、より高い優先順位のタスクを再スケジュールして応答を計算し、タスクをディスパッチするのにかかる合計時間です。

注 - ディスパッチ中の潜在的な時間のインタバル間に割り込みが入って処理されることがあります。この処理でアプリケーションの応答時間は増えますが、ディスパッチ中の潜在的な時間の測定には影響を与えないので、ディスパッチ中の潜在的な時間の保証によって制限されることはありません。

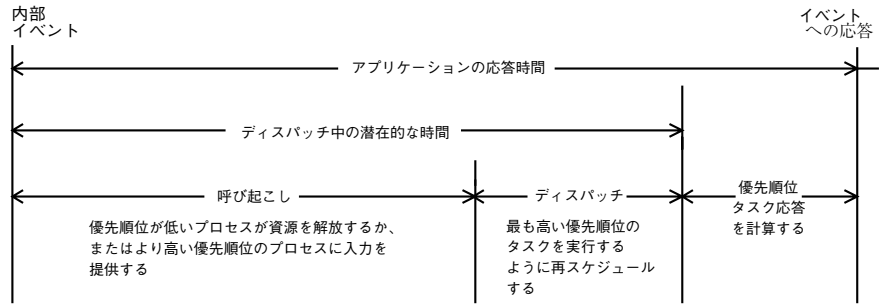

図 9–3　内部ディスパッチ中の潜在的な時間

実時間 SunOS 5 で用意されている新しいスケジューリング手法によって、システムのディスパッチ中の潜在的な時間は指定された範囲内になります。下の表に示すように、ディスパッチ中の潜在的な時間はプロセス数を制限すると向上します。

表 9–1　SunOS 5 での実時間システムのディスパッチ中の潜在的な時間

ワークステーション	制限されたプロセス数	任意のプロセス数
SPARCstation™ 2	有効なプロセスが 8 個未満の場合は、システム内で 0.5 ミリ秒未満	1.0 ミリ秒
SPARCstation 5	1.0 ミリ秒	0.3 ミリ秒
Ultra™ 1-1677	0.15 ミリ秒未満	0.15 ミリ秒

ディスパッチ中の潜在的な時間の検査と、製造業務やデータ収集業務などのクリティカルな環境での経験によって、Sun ワークステーションは実時間アプリケーション開発のための有効なプラットフォームであることが証明されています。(ただし上記の結果は、最新製品によるものではありませんのでご了承ください)。

スケジューリングクラス

SunOS 5 のカーネルは、プロセスを優先順位によってディスパッチします。スケジューラ (またはディスパッチャ) は、スケジューリングクラスの概念をサポートしています。クラスは、実時間 (RT)、システム (SYS)、またはタイムシェアリング (TS) として定義されます。各クラスには、プロセスをディスパッチするための固有のスケジューリング方式があります。

カーネルは、最も優先順位が高いプロセスを最初にディスパッチします。デフォルトでは、実時間プロセスが SYS や TS のプロセスよりも優先されますが、管理者は TS と RT のプロセスの優先順位が重なり合うように設定することもできます。

図 9–4 に SunOS 5 から見たクラスの概念を示します。

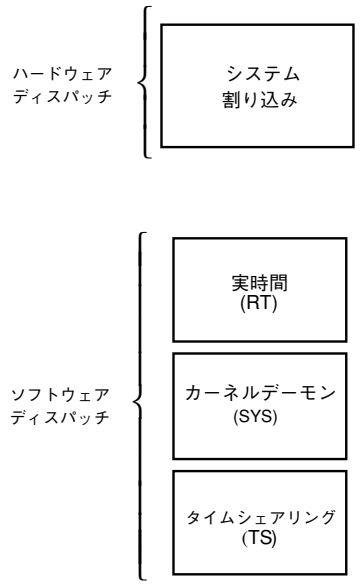

図 9–4　スケジューリングクラスのディスパッチ優先順位

最も優先順位が高いのはハードウェア割り込みで、これはソフトウェアでは制御できません。割り込みを処理するルーチンは、割り込みが生じるとただちに直接ディスパッチされ、その際には現在のプロセスの優先順位は考慮されません。

実時間プロセスは、ソフトウェアでは最も高い優先順位をデフォルトで持ちます。RT クラスのプロセスは、優先順位とタイムカンタム値を持ちます。RT プロセスは、厳密にこれらのパラメタに基づいてスケジュールされます。RT プロセスが実行可能である限り、sys や TS のプロセスは実行できません。固定優先順位スケジューリングでは、クリティカルプロセスを完了まで事前に指定した順序で実行できます。この優先順位は、アプリケーションで変更されない限り変わりません。

RT クラスのプロセスは、有限無限を問わず親プロセスのタイムカンタムを継承します。有限タイムカンタムを持つプロセスは、タイムカンタムの有効時間が切れるか、プロセスが終了するか、ブロッキングされるか (入出力イベントを待つ)、またはより高い優先順位を持つ実行可能な実時間プロセスに横取りされるまで実行されます。無限タイムカンタムを持つプロセスは、プロセスが終了するか、ブロッキングされるか、または横取りされるまで実行されます。

sys クラスは、ページング、STREAMS、スワッパなどの特殊なシステムプロセスをスケジュールするために存在します。あるプロセスのクラスを sys クラスに変更できません。プロセスの sys クラスは、プロセスの開始時にカーネルによって確立された固定優先順位を持っています。

優先順位が最も低いのは、タイムシェアリング (TS) クラスです。TS クラスのプロセスは、各タイムスライスを数百ミリ秒としてダイナミックにスケジュールされます。TS スケジューラは、頻繁にコンテキストスイッチングを行なって、各プロセスに実行する機会を平等に与えます。これは、各プロセスのタイムスライスの値とプロセスの履歴 (プロセスが最後に休眠したのはいつか) に基づき、CPU の利用率を考慮して行われます。デフォルトのタイムシェアリング方式では、優先順位の低いプロセスに長いタイムスライスを与えます。

子プロセスは `fork(2)` を通じて、親プロセスのスケジューリングクラスと属性を継承します。プロセスのスケジューリングクラスと属性は、`exec(2)` を実行しても変わりません。

各スケジューリングクラスは、異なったアルゴリズムによってディスパッチされます。クラスに依存するルーチンは、カーネルによって呼び出され、CPU のプロセススケジューリングが決定されます。カーネルはクラスに依存し、待ち行列内から最も優先順位の高いプロセスを取り出します。各クラスは、自分のクラスのプロセスの優先順位値を計算しなければなりません。この値は、そのプロセスのディスパッチ優先順位変数に入れられます。

図 9–5 に示すように各クラスのアルゴリズムは、それぞれ独自の方法によってグローバル実行待ち行列に入れる最も優先順位の高いプロセスを指定します。

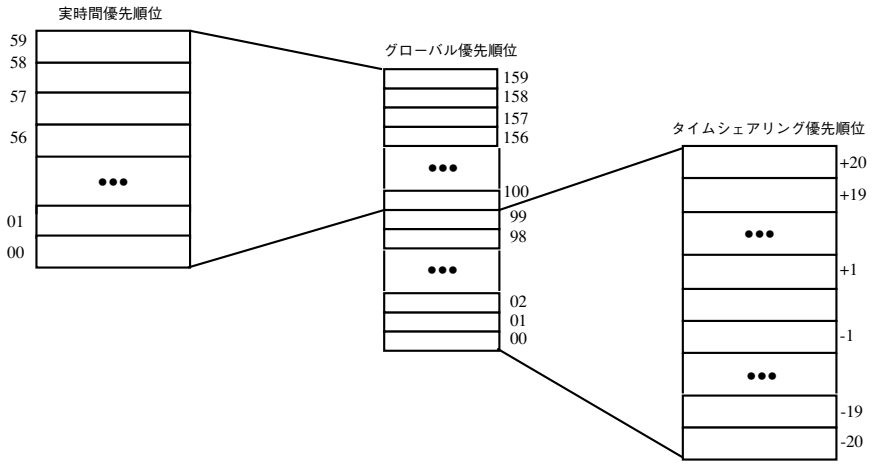

図 9–5　カーネルのディスパッチ待ち行列

各クラスには、そのクラスのプロセスに適用される優先順位レベルのセットがあります。クラス固有のマッピングによって、この優先順位がグローバル優先順位のセットに割り当てられます。グローバルスケジューリング優先順位のセットへの対応は 0 で始まったり連続したりしている必要はありません。

デフォルトでは、タイムシェアリング (TS) プロセスのグローバル優先順位の値は -20 から +20 までの範囲で、カーネルの 0 から 40 までに割り当てられており、一時的な割り当ては 99 まであります。実時間 (RT) プロセスのデフォルトの優先順位は 0 から 59 までの範囲で、カーネルの 100 から 150 までに割り当てられます。カーネルのクラスに依存しないコードは、待ち行列内のグローバル優先順位の最も高いプロセスを実行します。

ディスパッチ待ち行列

ディスパッチ待ち行列は、同じグローバル優先順位を持つプロセスが線状にリンクしたリストです。各プロセスは、それぞれに接続されているクラス固有の情報によって起動されます。プロセスは、グローバル優先順位に基づいたカーネルのディスパッチテーブルからディスパッチされます。

プロセスのディスパッチ

プロセスがディスパッチされると、プロセスのコンテキストがメモリ管理情報、レジスタ、スタックとともにメモリ内に割り当てられて実行が始まります。メモリ管理情報は、現在実行中のプロセスのために仮想記憶変換を実行する際、必要となるデータを含むハードウェアレジスタの形をとります。

横取り

より高い優先順位を持つプロセスがディスパッチ可能になると、カーネルは計算に割り込んでコンテキストスイッチングを強制し、現在実行中のプロセスを横取りします。より高い優先順位のプロセスがディスパッチ可能になったことをカーネルが見つけると、プロセスはいつでも横取りされます。

たとえば、プロセス A が周辺デバイスから読み取りを行なっているとします。プロセス A はカーネルによって休眠状態に置かれます。次に、カーネルはより優先順位の低いプロセス B が実行可能になったのに気づき、プロセス B がディスパッチされ実行が始まります。ここで周辺デバイスが割り込み、デバイスのドライバが入ります。デバイスドライバはプロセス A を実行可能にして戻ります。ここで、カーネルは割り込まれたプロセス B に戻るのではなく、B の処理を横取りして、呼び起こされたプロセス A の実行を再開します。

もう 1 つの重要な例としては、複数のプロセスがカーネル資源を奪い合う場合があります。優先順位の高い実時間プロセスが待っている資源を優先順位の低いプロセ

スが解放すると、カーネルはただちに優先順位の低いプロセスを横取りして、優先順位の高いプロセスの実行を再開します。

カーネル優先順位の反転

優先順位の反転は、優先順位の高いプロセスが 1 つまたは複数の優先順位の低いプロセスによって長時間ブロッキングされた場合に生じます。SunOS 5 のカーネルで相互排他ロッキングなどの同期プリミティブを使用すると、優先順位の反転につながることがあります。

「ブロッキング」とは、あるプロセスが 1 つまたは複数のプロセスが資源を手放すのを待たなければならない状態のことです。このブロッキングが継続すると、使用レベルが低いものでもデッドラインを逃してしまうことがあります。

相互排他ロッキングの優先順位反転の問題については、SunOS 5 のカーネルで基本的な優先順位継承方式を実装することによって対応しています。この方式では、優先順位の低いプロセスが優先順位の高いプロセスの実行をブロッキングすると、優先順位の低いプロセスが優先順位の高いプロセスの優先順位を継承することになります。このため、プロセスがブロッキングされている時間の上限が設定されます。この方式はカーネルの特性で、プログラマがシステムコールや関数の実行によって講じる解決策ではありません。ただしこの場合でも、ユーザレベルのプロセスは優先順位の反転を生じることがあります。

ユーザ優先順位の反転

他のプロセスと同期がとられているプロセスが、待っているプロセスの優先順位を自動的に継承する機構はありません。アプリケーションは、優先順位の上限をエミュレートして自分の優先順位反転を制限できます。

このモデルでは、アプリケーションは、それぞれの同期オブジェクトに優先順位を関連付けますが、通常はそのオブジェクトをブロッキングできる、あらゆるプロセス中で最も高い優先順位になります。

次に、各プロセスは次のようなシーケンスを使用して共用資源を操作します。

- プロセスの優先順位を現在のレベルと同期オブジェクトレベルの最高まで上げる
- 同期しているオブジェクトを取得する
- 危険領域を実行する
- 同期しているオブジェクトを解放する
- 以前のプロセス優先順位に戻す

スケジューリングを制御するシステムコール

priocntl(2)

有効なクラスのスケジューリング制御は、priocntl(2) で処理します。クラスの属性は、fork(2) や exec(2) を実行した場合にも、優先順位制御に必要なスケジューリングパラメタやアクセス権とともに継承されます。この特色は、RT と TS クラスのどちらにも当てはまります。

priocntl(2) 関数は、システムコールが適用される実時間プロセス、プロセスのセット、またはクラスを指定するインタフェースを提供します。priocntlset(2) のシステムコールも、システムコールを適用するプロセスのセット全体を指定する、さらに一般的なインタフェースを提供します。

priocntl のコマンド引数
は、PC_GETCID、PC_GETCLINFO、PC_GETPARMS、PC_SETPARMS のいずれかにします。呼び出しプロセスの実識別子または実効識別子は、対象となるプロセスのものと一致するか、スーパーユーザ特権を持っていなければなりません。

PC_GETCID	このコマンドは、認識可能なクラス名 (実時間なら RT、タイムシェアリングなら TS) を含む構造体の名前フィールドを受け入れます。クラス ID とクラス属性データの配列が戻されます。
PC_GETCLINFO	このコマンドは、認識可能なクラス識別子を含む構造体の ID フィールドを受け入れます。クラス名とクラス属性データの配列が戻されます。
PC_GETPARMS	このコマンドは、指定したプロセスの 1 つのスケジューリングクラス識別子またはクラス固有のスケジューリングパラメタ、あるいはその両方を戻します。idtype と id によって大きなセットが指定された場合でも、PC_GETPARMS は 1 つのプロセスのパラメタだけを戻します。どのプロセスを選択するかはクラスによって決まります。
PC_SETPARMS	このコマンドは、指定したプロセス (複数でも可) のスケジューリングクラス識別子またはクラス固有のスケジューリングパラメタ、あるいはその両方を設定します。

sched_get_priority_max(3R)

指定された方針の最大値を戻します。

sched_get_priority_min(3R)

指定された方針の最小値を戻します (詳細は、sched_get_priority_max(3R) のマニュアルページを参照してください)。

sched_rr_get_interval(3R)

指定された timespec 構造体を現在の実行時間限界に更新します (詳細は、sched_get_priority_max(3R) のマニュアルページを参照してください)。

sched_setparam(3R) と sched_getparam(3R)

指定されたプロセスのスケジューリングパラメタを設定または取得します。

sched_yield(3R)

プロセスリストの先頭に戻るまで、呼び出しプロセスをブロッキングします。

スケジューリングを制御するユーティリティ

プロセスのスケジューリングを制御する管理用ユーティリティとして、dispadmin(1M) と priocntl(1) があります。どちらのユーティリティも、互換性のあるオプションとロード可能なモジュールを伴う priocntl(2) のシステムコールをサポートします。これらのユーティリティは、実行中に実時間プロセスのスケジューリングを制御するシステム管理機能を提供します。

priocntl(1)

priocntl(1) コマンドは、プロセスのスケジューリングパラメタの設定と取り出しを行います。

dispadmin(1M)

dispadmin(1M) ユーティリティに –l コマンド行オプションを付けると、実行中に現プロセスのすべてのスケジューリングクラスが表示されます。実時間クラスを表す引数として RT を –c オプションの後ろに指定すると、プロセスのスケジューリングを変更することもできます。

表 9–2 に示しているオプションも使用できます。

表 9–2 dispadmin(1M) ユーティリティのクラスオプション

オプション	意味
–l	現在設定されているスケージュリングクラスを表示する。
–c	パラメタを表示または変更するクラスを指定する。
–g	指定したクラスのディスパッチパラメタを取得する。
–r	–g オプションを使用した場合、タイムカンタムの解像度を指定する。
–s	値が保存されているファイルを指定する。

ディスパッチパラメタが保存されているクラス固有のファイルを実行中にロードすることもできます。このファイルを使用して、起動時に確立されたデフォルトの値を新しい優先順位のセットで置き換えることができます。このクラス固有のファイルでは、–g オプションで使用される書式の引数を挿入しなければなりません。RT クラスのパラメタは rt_dptbl(4) にあり、この節の終わりに例を示します。

システムに RT クラスのファイルを追加するには、次のモジュールが存在しなければなりません。

- rt_dptbl をロードするクラスモジュール内の rt_dptbl(4) ルーチン
- ディスパッチパラメタと、config_rt_dptbl へのポインタを戻すルーチンを提供する rt_dptbl(4)
- dispadmin(1M) 実行可能ファイル

1. 次のコマンドでクラス固有のモジュールをロードします。

この場合、*module_name* はクラス固有のモジュールにします。

```
# modload /kernel/sched/module_name
```

2. dispadmin(1M) コマンドを起動します。

```
# dispadmin -c RT -s file_name
```

上書きされるテーブルと同じ数のエントリを持つテーブルが、ファイルに記述されていなければなりません。

スケジューリングの設定

両方のスケジューリングクラスにはパラメタテーブル rt_dptbl(4) と ts_dptbl(4) が関連づけられています。これらのテーブルは、起動時にロード可能なモジュールを使用するか、実行中に dispadmin(1M) を使用して設定できます。

ディスパッチャパラメタテーブル

実時間のための中心となるテーブルで、RT スケジューリングの設定項目を指定します。rt_dptbl(4) 構造体は、パラメタ配列の structrt_dpent_t 構造体から成り、これは n 個の優先順位レベルそれぞれに 1 つずつあります。ある優先順位レベル i の設定項目は、配列内の i 番目のパラメタ構造体 rt_dptbl[i] によって指定されます。

パラメタ構造体は次のメンバーから成ります (/usr/include/sys/rt.h ヘッダファイルにも記述されています)。

rt_globpri	この優先順位レベルに関係づけられているグローバルスケジューリング優先順位。rt_globpri の値は dispadmin(1M) では変更できません。
rt_quantum	このレベルのプロセスに割り当てられるタイムカンタムの長さを目盛で表したもの (123ページの「タイムスタンプ機能」を参照)。タイムカンタム値は、特定のレベルのプロセスのデフォルト値、つまり開始値です。実時間プロセスのタイムカンタムは、priocntl(1) コマンドまたは priocntl(2) システムコールによって変更できます。

config_rt_dptbl の再設定

実時間管理者は、いつでも config_rt_dptbl を再設定して、スケジューラの実時間部分の振る舞いを変更できます。1 つの方法は、rt_dptbl(4) のマニュアルページの「REPLACING THE RT_DPTBL LOADABLE MODULE」というセッションで説明しています。

もう 1 つの方法は、dispadmin(1M) コマンドを使用して、実行中のシステムで実時間パラメタテーブルを調査または変更する方法です。dispadmin(1M) を実時間クラスで起動すると、カーネルの中心テーブルにある現在の config_rt_dptb1 内

から現在の rt_quantum 値を取り出すことができます。現在の中心テーブルを上書きする際、dispadmin(1M) への入力に使用された設定ファイルは、rt_dptbl(4) のマニュアルページで説明されている書式に合致していなければなりません。

config_rt_dptbl[] 内にある優先順位を設定されたプロセス rtdpent_t と、関連づけられているタイムカンタム値を次に示します。

```
rtdpent_t  rt_dptbl[] = {
    /* 優先順位レベルのタイムカンタム */
            100,   100,
            101,   100,
            102,   100,
            103,   100,
            104,   100,
            105,   100,
            106,   100,
            107,   100,
            108,   100,
            109,   100,
            110,   80,
            111,   80,
            112,   80,
            113,   80,
            114,   80,
            115,   80,
            116,   80,
            117,   80,
            118,   80,
            119,   80,
            120,   60,
            121,   60,
            122,   60,
            123,   60,
            124,   60,
            125,   60,
            126,   60,
            127,   60,
            128,   60,
            129,   60,
            130,   40,
            131,   40,
            132,   40,
            133,   40,
            134,   40,
            135,   40,
            136,   40,
            137,   40,
            138,   40
            139,   40,
            140,   20,
            141,   20,
            142,   20,
            143,   20,
            144,   20,
            145,   20,
            146,   20,
            147,   20,
            148,   20,
            149,   20,
            150,   10,
            151,   10,
            152,   10,
            153,   10,
            154,   10,
            155,   10,
            156,   10,
            157,   10,
            158,   10,
            159,   10,
                                    }
```

メモリロッキング

メモリのロッキングは、実時間アプリケーションにとって最重要事項の 1 つです。実時間環境では潜在的な時間を減らし、ページングとスワッピングを防ぐために、プロセスは連続してメモリ内に常駐していることを保証される必要があります。

この節では、SunOS 5 で実時間アプリケーションに利用できるメモリロッキング機構について説明します。メモリ管理関数や呼び出しの使用方法の詳細は、『*man Pages(3): Library Routines*』を参照してください。

概要

SunOS 5 では、プロセスのメモリ常駐性は、プロセスの現在の状態、使用できる全物理記憶、有効なプロセス数、およびプロセッサのメモリに対する要求によって決まります。これは、タイムシェアリング環境には適合しますが、実時間プロセスについては受け入れられないことがよくあります。実時間環境では、プロセスは、メモリアクセスとディスパッチ中の潜在的な時間を減らすために、プロセスの全部または一部がメモリ内に常駐していることが保証される必要があります。

SunOS 5 の実時間では、メモリロッキングはスーパーユーザ特権付きで実行されているプロセスが、自分の仮想アドレス空間の指定された部分を物理記憶にロッキングできるようにするライブラリルーチンのセットによって提供されています。このようにしてロッキングされたページは、ロッキングを解除されるか、プロセスが終了するまでページングの対象になりません。

一度にロッキングできるページ数には、システム全体で共通の制限があります。これは調整可能なパラメタで、デフォルト値は起動時に計算されます。値は、ページフレーム数マイナス一定のパーセント (現在は 10 パーセントに設定) が基本になります。

ページのロッキング

`mlock(3C)`は、メモリの 1 セグメントをシステムの物理記憶にロッキングするように要求します。指定したセグメントを構成するページは、ページフォルトが発生して物理記憶に入り、各ロッキングカウント値は 1 増やされます。ロッキングのカウントが 0 より大きいページは、ページング操作から除外されます。

特定のページを異なったマッピングで複数のプロセスを使って何度もロッキングできます。2 つの異なったプロセスが同じページをロッキングすると、両方のプロセスがロッキングを解除するまで、そのページはロッキングされたままです。ただし、マッピング内でページロッキングはネストしません。同じアドレスに対して同じプロセスがロッキング関数を何度も呼び出した場合でも、ロッキングは一度のロッキング解除要求で削除されます。

ロッキングが実行されているマッピングが削除されると、そのメモリセグメントのロッキングは解除されます。ファイルを閉じるまたは切り捨てることによってページが削除された場合も、ロッキングは解除されます。

fork(2) 呼び出しが行われた後、ロッキングは子プロセスには継承されません。したがって、メモリをロッキングしているプロセスが子プロセスをフォークすると、子プロセスはページをロッキングするために、自分でメモリロッキング操作を行わなければなりません。そうしないと、プロセスをフォークした場合に通常必要となるページのコピー即時書き込みを行わなければならなくなります。

ページのロッキング解除

メモリによるページのロッキングを解除するには、プロセスは munlock(3C) 呼び出しによって、ロッキングされている仮想記憶のセグメントを解放するように要求します。こうすると、指定された物理ページのロッキングカウントが減らされます。ページのロッキングカウントが 0 まで減ると、そのページは普通にスワップされます。

全ページのロッキング

スーパーユーザプロセスは、mlockall(3C) 呼び出しによって、自身のアドレス空間内の全マッピングをロッキングするように要求できます。MCL_CURRENT フラグが設定されている場合は、既存のメモリマッピングがすべてロッキングされます。MCL_FUTURE フラグが設定されている場合は、既存のページに追加されるか、既存のマッピングを置き換えるマッピングはすべてメモリ内にロッキングされます。

スティッキロッキング

ページのロッキングカウントが 65535 (0xFFFF) に達すると、そのページは永久にロッキングされます。0xFFFF の値は実装の際に定義されているため、将来のリリースでは変更される可能性があります。このようにしてロッキングされたページは、ロッキングを解除できません。復元するにはシステムを再起動してください。

高性能入出力

この節では、実時間プロセスでの入出力について説明します。SunOS 5 では、高速で非同期の入出力操作を実行するための 2 種類のインタフェースをライブラリで提供しています。Solaris 非同期入出力インタフェースは、Solaris 1.x でも提供されています。POSIX 非同期入出力インタフェースは新しい標準です。堅牢性向上のため、SunOS は情報の消失やデータの不一致を防止するためのファイルおよびメモリ内同期操作とモードも提供しています。詳細は、『*man Pages(3): Library Routines*』を参照してください。

標準の UNIX 入出力は、アプリケーションのプログラマと同期します。`read(2)` または `write(2)` を呼び出すアプリケーションは、通常はシステムコールが終了するまで待ちます。

実時間アプリケーションは、入出力に非同期で結合された特性を必要とします。非同期入出力呼び出しを発行したプロセスは、入出力操作の完了を待たずに先に進むことができます。呼び出し側は、入出力操作が終了すると通知されます。その間、プロセスは他の動作を行います。

非同期入出力は、任意の SunOS ファイルで使用できます。ファイルは同期して開かれますが、特別なフラグ設定は必要ありません。非同期入出力転送には、呼び出し、要求、操作の 3 つの要素があります。アプリケーションは非同期入出力関数を呼び出し、入出力要求が待ち行列に置かれ、呼び出しはただちに復帰します。ある時点で、システムは要求を待ち行列から取り出し、入出力操作を開始します。

非同期入出力要求と標準入出力要求は、任意のファイル記述子で混在させることができます。システムは、読み取り要求と書き込み要求の特定の順序を維持しません。システムは、保留状態にあるすべての読み取り要求と書き込み要求の順序を任意に並べ替えます。特定の順序を必要とするアプリケーションは、前の操作の完了を確認してから従属する要求を発行しなければなりません。

POSIX 非同期入出力

POSIX 非同期入出力は、`aiocb` 構造体を使用して行います。`aiocb` 制御ブロッキングは、各非同期入出力要求を識別し、すべての制御情報を持っています。制御ブロッキングは、一度に 1 つの要求だけに使用でき、その要求が完了すると再使用できます。

一般的な POSIX 非同期入出力操作は、aio_read(3R) または aio_write(3R) 呼び出しによって開始します。ポーリングまたはシグナルを使用して、操作の完了を判断できます。シグナルを操作の完了に使用する場合は、各操作に一意にタグを付けることができます。タグは生成されたシグナルの si_value 構成要素に戻されます (詳細は、siginfo(5) のマニュアルページを参照してください)。

aio_read(3R)

aio_read(3R) は、読み取り操作の開始のために非同期入出力制御ブロッキングを使用して呼び出します。

aio_write(3R)

aio_write(3R) は、書き込み操作の開始のために非同期入出力制御ブロッキングを指定して呼び出します。

aio_return(3R) と aio_error(3R)

aio_return(3R) と aio_error(3R)は、操作が完了しているとわかった後、それぞれ戻り値とエラー値を取得するために呼び出します。

aio_cancel(3R)

aio_cancel(3R) は、保留状態の操作を取り消すために非同期入出力制御ブロッキングを指定して呼び出します。

aio_fsync(3R)

aio_fsync(3R) は、指定したファイルで保留状態のすべての入出力操作に対する非同期の fsync(3C) または fdatasync(3R) 要求を待ち行列に並べます。

aio_suspend(3R)

aio_suspend(3R) は、1 つ以上の先行する非同期入出力要求が同期して行われるかのように呼び出し側を一時停止します。

Solaris 非同期入出力

通知 (SIGIO)

非同期入出力呼び出しが正常に復帰しても、入出力操作は待ち行列に並べられただけであり、実行を待っています。実際の操作は、戻り値と潜在的なエラー識別子も持っています。これらの値は、同期呼び出しの結果として呼び出し側に戻されます。入出力が終了すると、戻り値とエラー値は、要求時点でユーザが aio_result_t へのポインタとして指定した位置に格納されます。aio_result_t 構造体は、<sys/asynch.h> に次のように定義されています。

```
typedef struct aio_result_t {
  ssize_t aio_return; /* 読み取りまたは書き込みの戻り値 */
  int    aio_errno;  /* 入出力によって生成されたエラー番号 */
 } aio_result_t;
```

aio_result_t が変更されると、入出力要求を行なったプロセスに SIGIO シグナルが配信されます。

2 つ以上の非同期入出力操作を保留状態にしている場合、どの要求によって SIGIO シグナルが生じたか、またはどちらの要求によって SIGIO シグナルが生じたのかは調べることはできません。SIGIO を受け取ったプロセスは、SIGIO を生じた原因となる条件をすべてチェックしなければなりません。

aioread(3)

aioread(3) は read(2) の非同期版です。aioread は通常の読み取り引数に加えて、ファイル位置と、システムが操作結果を格納する aio_result_t 構造体のアドレスを指定する引数を取ります。ファイル位置には、操作前にファイル内で行うシークを指定します。aioread 呼び出しが成功したか失敗したかに関係なく、ファイルポインタが更新されます。

aiowrite(3)

aiowrite(3) 関数は write(2) の非同期版です。aiowrite は通常の書き込み引数以外にファイル位置と、操作結果が格納される aio_result_t 構造体のアドレスを指定する引数を取ります。

ファイル位置には、操作の前にファイル内で行うシークを指定します。aiowrite 呼び出しが成功すると、ファイルポインタはシークと書き込みが成功した場合の位

置に変更されます。ファイルポインタは書き込みを行なった後、以降の書き込みができなくなった場合も変更されます。

aiocancel(3)

aiocancel(3) は、その aio_result_t 構造体を引数として指定した非同期要求の取り消しを試みます。aiocancel 呼び出しは、要求がまだ待ち行列にある場合だけに成功します。操作がすでに進行していると aiocancel は失敗します。

aiowait(3)

aiowait(3) を呼び出すと、少なくとも 1 つの未処理の非同期入出力操作が完了するまで呼び出しプロセスはブロッキングされます。タイムアウトパラメタは、入出力の完了を待つ最大インタバルを指します。0 のタイムアウト値は、待つ必要がないことを指定します。aiowait(3) は、完了した操作の aio_result_t 構造体へのポインタを戻します。

poll(2)

SIGIO 割り込みに依存するのではなく、デバイスをポーリングしたい場合は、poll(2) システムコールを使用します。SIGIO 割り込みの原因を調べるためにポーリングすることもできます。

close(2)

ファイルは、close(2) 呼び出しによって閉じられます。close(2) を呼び出すと、取り消すことができる未処理の非同期入出力要求はすべて取り消されます。close(2) 関数は、取り消せない操作の完了を待ちます (詳細は、111ページの「aiocancel(3)」を参照してください)。close(2) 呼び出しが戻ると、そのファイル記述子について保留状態にある非同期入出力要求はなくなります。ファイルが閉じられると、取り消されるのは指定したファイル記述子に対する待ち行列内にある非同期入出力要求だけです。他のファイル記述子について、保留状態にある入出力要求は取り消されません。

同期入出力

アプリケーションは、情報が安定した記憶領域に書き込まれたことや、ファイル変更が特定の順序で行われることを保証する必要がある場合があります。同期入出力は、このような場合のために用意されています。

同期のモード

SunOS 5 では、物理記憶領域媒体でエラーなしに読み取れることがシステムで保証されている場合、書き込まれたデータがすべて、そのファイルをあとで開いた際に (システムや電源の障害後であっても)、書き込み操作のために通常ファイルへ正しく転送されます。物理記憶領域媒体上のデータのイメージを要求側のプロセスが利用できる場合、データは読み取り操作のために正しく転送されます。入出力操作は、関連づけられているデータが正しく転送されたか、操作が失敗と診断された場合に完了します。

入出力操作は、次の場合に同期入出力データの保全を完了します。

読み取りについては、操作は完了または失敗と診断されます。読み取りが完了するのは、データのイメージが要求側のプロセスに正しく転送された場合だけです。同期読み取り操作が要求された時点で、読み取るデータに影響を与える書き込み要求が保留状態にある場合は、データを読み取る前に書き込み要求が正しく転送されます。

書き込みについては、操作は完了または失敗と診断されます。書き込みが完了するのは、書き込み要求で指定されたデータが正しく転送され、そのデータを取り出すために必要なファイルシステム情報がすべて正しく転送された場合だけです。

データの取り出しに必要のないファイル属性 (アクセス時間、変更時間、状態変更時間) は、呼び出し側プロセスに戻る前に正しく転送されているわけではありません。

同期入出力ファイルの保全の完了は、呼び出し側プロセスに戻る前に入出力操作に関連するすべてのファイル属性 (アクセス時間、修正時間、状態変更時間を含む) が正しく転送されなければならない点を除けば、同期入出力データの保全の完了と同じです。

ファイルの同期

`fsync(3C)` 関数と `fdatasync(3R)` 関数は、ファイルを二次記憶領域と明示的に同期をとります。形式は次のようになります。

fsync(3C) は、入出力ファイルの保全完了レベルで関数の同期をとることを保証し、fdatasync(3R) は、入出力データの保全完了レベルで関数の同期をとることを保証します。

アプリケーションは、操作が完了する前に各入出力操作の同期をとるように指定できます。open(2) または fcntl(2) によってファイル記述に O_DSYNC フラグを設定すると、操作が完了したと見なされる前にすべての入出力書き込み (write(2) と aiowrite(3)) が入出力データ完了に達します。ファイル記述に O_SYNC フラグを設定すると、操作が完了したと見なされる前に、すべての入出力書き込みが入出力ファイル完了に達します。ファイル記述に O_RSYNC フラグを設定すると、すべての入出力読み取り (read(2) と aio_read(3R)) が、O_DSYNC または O_SYNC を記述子に設定した書き込み要求と同じ完了レベルに達します。

プロセス間通信

この節では、SunOS 5 のプロセス間通信 (IPC) 機能を、実時間処理との関連で説明します。シグナル、パイプ、FIFO (名前付きパイプ)、メッセージ待ち行列、共用メモリ、ファイルマッピング、およびセマフォについて説明します。プロセス間通信に役立つライブラリ、関数、およびルーチンについては、第 8 章と『*man Pages(3): Library Routines*』を参照してください。

概要

実時間処理は、しばしば高速な高いバンド幅のプロセス間通信を必要とします。どの機構を使用すればよいかは機能的な要求によって決まり、相対的な性能はアプリケーションの特性に依存します。

UNIX での従来のプロセス間通信の方法はパイプですが、パイプはフレーム上の問題を生じます。複数の人がメッセージを書いた結果が混合したり、あるメッセージを複数の人が読むと分断されてしまったりすることがあります。

IPC のメッセージは、ファイルの読み取りや書き込みと似たものです。これは 2 つ以上のプロセスが 1 つの媒体によって通信しなければならない場合、パイプより使いやすいです。

IPC の共用セマフォ機能では、プロセスの同期をとることができます。共用メモリは最も高速なプロセス間通信の形式です。共用メモリの主な長所は、メッセージ

データのコピーが不要な点です。共用メモリアクセスの同期をとるには、通常はセマフォの機構を使用します。

シグナル

シグナルを使用してプロセス間で少量の情報を送信できます。次のように、送り側は sigqueue(3R) 関数を使用して、少量の情報とともにシグナルをターゲットプロセスに送信します。

ターゲットプロセスは、以降に発生した保留状態のシグナルも待ち行列に入れるため、指定されたシグナルの SA_SIGINFO ビットを設定していなければなりません (詳細は、sigaction(2) のマニュアルページを参照してください)。

ターゲットプロセスは、シグナルを同期または非同期に受信できます。シグナルをブロッキングしたまま (sigprocmask(2) のマニュアルページを参照)、sigwaitinfo(3R) または sigtimedwait(3R) を呼び出すと、シグナルは、siginfo_t 引数の si_value メンバーに格納されている、sigqueue(3R) の呼び出し側によって送信された値と同期をとって受信されます。シグナルのブロッキングを解除しておくと、シグナルは sigaction(2) によって指定されたシグナルハンドラに配信され、値はハンドラへの siginfo_t 引数の si_value に設定されます。

関連づけられた値を持つシグナルで、送信しても配信されないものの数は、1 プロセスあたり固定です。{SIGQUEUE_MAX} 個のシグナルの記憶領域は、sigqueue(3R) を最初に呼び出した時点で割り当てられます。その後 sigqueue(3R) を呼び出すと、ターゲットプロセスの待ち行列にシグナルが正常に入るか、または制限時間内で異常終了します。

パイプ

パイプは、プロセス間の一方方向の通信を提供します。プロセスがパイプで通信するには、共通の祖先を持っていなければなりません。パイプを通して渡されるデータは、通常の UNIX バイトストリームとして扱われます。パイプについては、67 ページの「パイプ」を参照してください。

名前付きパイプ

SunOS 5 では名前付きパイプ (FIFO) が用意されています。FIFO はディレクトリ内の名前付きエンティティなので、パイプに比べて柔軟性があります。FIFO が作成され

ると、適切なアクセス権を持っていればどのプロセスでも FIFO を開くことができます。プロセスは親を共用している必要はなく、親がパイプを初期化して子孫に渡す必要もありません。詳細は、69ページの「名前付きパイプ」を参照してください。

メッセージ待ち行列

メッセージ待ち行列は、プロセス間で通信するもう 1 つの手段を提供します。任意の数のプロセスが 1 つのメッセージ待ち行列だけで送受信できます。メッセージは、バイトストリームとしてではなく、任意の大きさのブロッキングとして渡されます。メッセージ待ち行列は、System V 版と POSIX 版の両方で提供されています。詳細は、77ページの「System V メッセージ」と 73ページの「POSIX メッセージ」を参照してください。

セマフォ

セマフォは、共用資源に対してアクセスの同期をとる機構です。セマフォも、System V と POSIX の両方で提供されています。System V セマフォは非常に柔軟性がありますが、重量がかなりあります。POSIX セマフォは、極めて軽量です。詳細は、79ページの「System V セマフォ」と 74ページの「POSIX セマフォ」を参照してください。

セマフォを使用すると、この章で前述した技法によって明示的に回避しない限り、優先順位の反転が生じる場合があるので注意してください。

共用メモリ

プロセスが通信するための最も高速な方法は、直接メモリの共用セグメントを使用した場合です。共通メモリ領域が共用しているプロセスのアドレス空間に追加されます。アプリケーションは、データを格納することでデータを送信し、データを取り出すことで通信データを受信します。SunOS 5 では、共用メモリのための機構として、メモリにマッピングされたファイル、System V IPC 共用メモリ、POSIX 共用メモリの 3 つの方法を提供しています。

共用メモリを使用するときの最も大きな問題点は、3 つ以上のプロセスが同時に共用メモリに読み取りや書き込みを行おうとすると結果が正しくなくなる場合があることです。詳細は、116ページの「共用メモリの同期」を参照してください。

メモリにマッピングされたファイル

mmap(2) インタフェースは、共用メモリセグメントを呼び出し側のアドレス空間に接続します。呼び出し側は、アドレスと長さによって共用セグメントを指定します。呼び出し側は、アクセス保護フラグとマッピングされたページを管理する方法も指定しなければなりません。mmap(2) を使用して、ファイルまたはファイルのセグメントをプロセスのメモリにマッピングすることもできます。この技法は、あるアプリケーションでは非常に便利ですが、マッピングされたファイルセグメントへの格納が暗黙の入出力になる場合があるということを忘れがちです。それ以外の場合では、結合されているプロセスの応答時間が予測できないものになることもあります。msync(3C) は、指定したメモリセグメントのその時のまたは最終的なコピーをパーマネント記憶領域に作成します。詳細は、63ページの「メモリ管理インタフェース」を参照してください。

ファイルなしメモリマッピング

0 の特別ファイルである /dev/zero(4S) は、名前がなく 0 で初期化されたメモリオブジェクトを作成するのに使用できます。メモリオブジェクトの長さはマッピングを含むページの最小の番号になります。オブジェクトは、共通の先祖プロセスをもつ子孫だけが共用できます。

System V IPC 共用メモリ

shmget(2) 呼び出しを使用して、共用メモリセグメントの作成と既存の共用メモリセグメントを取得できます。shmget(2) 関数は、ファイル識別子に似た識別子を戻します。shmat(2) を呼び出すと、mmap(2) とほとんど同じように共用メモリセグメントがプロセスメモリの仮想セグメントになります。詳細は、84ページの「System V 共用メモリ」を参照してください。

POSIX 共用メモリ

POSIX 共用メモリは System V 共用メモリの変形で、若干の違いはありますが同様の機能を提供します。詳細は、75ページの「POSIX 共用メモリ」を参照してください。

共用メモリの同期

共用メモリでは、メモリの一部が 1 つ以上のプロセスのアドレス空間にマッピングされます。アクセスを協調させる方法は自動的には提供されないため、2 つのプロ

セスが同時に同じ場所の共用メモリに書き込もうとすることがあります。このため共用メモリは、通常はプロセスの同期をとるセマフォやその他の機構と一緒に使用します。System V セマフォと POSIX セマフォは、両方ともこの目的のために使用できます。マルチスレッドライブラリに提供されている相互排他ロッキング、リーダロッキングとライタロッキング、セマフォ、および条件変数もこの目的のために使用できます。

IPC および同期の機構の選択

アプリケーションには特定の機能上の要求があって、それによってどの IPC 機構を使用するかが決まります。いくつかの機構が使用できる場合は、アプリケーションの作成者が、そのうちどれが最もアプリケーションに適しているかを決定します。アプリケーションの特性によって、IPC および同期の機構を選択してください。アプリケーションで使用される様々な長さのメッセージの組み合わせについて各機構のスループットを測定し、どの機構の応答が最も良いかを調べてください。

非同期ネットワーキング

この節では、実時間アプリケーションでトランスポートレベルインタフェース (TLI) を使用して非同期ネットワーク通信を行う方法を説明します。SunOS では、STREAMS の非同期機能と TLI ライブラリルーチンの非ブロッキングモードの組み合わせを使用した TLI イベントの非同期ネットワーク処理をサポートしています。

トランスポートレベルインタフェースの詳細は、『*Transport Interfaces Programming Guide*』と『*man Pages(3): Library Routines*』を参照してください。

ネットワーキングのモード

トランスポートレベルインタフェースでは、「接続モード」と「接続なしモード」という 2 つのモードサービスが用意されています。

接続モードサービス

「接続モード」は回線中心で、確立された接続上を信頼できるシーケンスでデータを伝送します。データ伝送フェーズでのアドレスの解決と伝送のオーバヘッドを避

けるための識別手続きも用意されています。このサービスは、比較的長い時間持続するデータストリーム中心の対話を必要とするアプリケーションに適しています。

接続なしモードサービス

「接続なしモード」はメッセージ中心で、複数のユニット間の論理的な関係を要求されない独立した単位でのデータ伝送をサポートします。宛先を含むデータのユニットを配信するために必要なすべての情報は、データと合わせて単一のサービス要求として送信側からトランスポートプロバイダに渡されます。接続なしモードサービスは、短い時間の要求と応答の対話を行い、データの配信の保証やシーケンスを必要としないアプリケーションに適しています。接続なしモードによる伝送は、概して信頼性が低いと言えます。

ネットワーキングプログラミングモデル

ファイルやデバイスの入出力と同様に、ネットワーク転送はプロセスサービス要求によって同期または非同期に実行できます。

同期ネットワーキング

同期ネットワーキングは、同期したファイルやデバイスの入出力と同様に進行します。`write(2)` 関数と同様に、送信要求はメッセージをバッファリングして戻りますが、バッファ空間がすぐ使用できない場合は、呼び出しプロセスを一時停止させることもあります。`read(2)` 関数と同様に、受信要求は要求を満たすデータが到着するまで、呼び出しプロセスの実行を一時停止させます。SunOS 5 では、転送サービスについて限界を保証していないので、同期ネットワーキングは他のデバイスに対して実時間特性が必要なプロセスには適していません。

非同期ネットワーキング

非同期ネットワーキングは、非ブロッキングサービス要求によって用意されます。また、アプリケーションは接続が確立またはデータが送受信されたときに非同期通知を要求することもできます。

非同期接続なしモードサービス

非同期接続なしモードのネットワーキングは、非ブロッキングサービスのエンドポイントを設定して、データ転送についての非同期通知をポーリングまたは受信することによって行われます。非同期通知を使用する場合は、実際のデータ受信はシグナルハンドラ内で行われます。

エンドポイントを非同期にする

t_open(3N) によってエンドポイントを確立して、t_bind(3N) によって識別情報を確立したら、エンドポイントを非同期サービス用に設定できます。これは、fcntl(2) 関数によってエンドポイントに O_NONBLOCK フラグを設定して行います。これ以降は、 t_sndudata(3N) を呼び出すと、バッファ空間がすぐに利用できない場合は -1 が戻され、t_errno が TFLOW に設定されます。同様に t_rcvudata(3N) を呼び出すと、データがない場合は -1 が戻され、t_errno が TNODATA に設定されます。

非同期ネットワーク転送

アプリケーションは、poll(2) 関数を使用してエンドポイントでデータの受信を定期的に調べるか、データの受信を待つことができますが、データが到着したことを知らせる非同期通知を受信する必要がある場合もあります。この場合は、ioctl(2) 関数に I_SETSIG コマンドを付けて実行し、エンドポイントにデータが到着したら SIGPOLL シグナルをプロセスに送信するように要求します。アプリケーションは、複数のメッセージによって 1 つのシグナルが生成される場合についても対処しなければなりません。

次の例で、protocol はアプリケーションが選択したトランスポートプロトコル名です。

```
#include <sys/types.h>
#include <tiuser.h>
#include <signal.h>
#include <stropts.h>

int    fd;
struct t_bind    *bind;
void    sigpoll(int);

 fd = t_open(protocol, O_RDWR, (struct t_info *) NULL);

 bind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
 ... /* バインドアドレスを設定する */
 t_bind(fd, bind, bin
```

```
 /* エンドポイントを非ブロッキングにする */
 fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);

 /* SIGPOLL のためのシグナルハンドラを確立する */
 signal(SIGPOLL, sigpoll);

 /* 受信データがあるときに SIGPOLL シグナルを要求する */
 ioctl(fd, I_SETSIG, S_INPUT | S_HIPRI);

 ...

void sigpoll(int sig)
{
 int     flags;
 struct t_unitdata     ud;

 for (;;) {
  ... /* ud を初期化する */
  if (t_rcvudata(fd, &ud, &flags) < 0) {
   if (t_errno == TNODATA)
    break; /* これ以上メッセージはない */
   ... /* その他のエラー状態を処理する */
 }
 ... /* ud 内のメッセージを処理する */
}
```

非同期接続モードサービス

接続モードサービスの場合、アプリケーションは、データ転送だけではなく接続も非同期に確立されるように設定できます。操作のシーケンスは、プロセスが他のプロセスに接続しようとしているのか、接続試行を待っているのかによって異なります。

非同期に接続を確立する

プロセスは、接続を試みてから非同期に完了できます。プロセスはまず接続エンドポイントを作成し、fcntl(2) を使用してエンドポイントを非ブロッキング操作用に設定します。接続なしデータ転送の場合と同様に、エンドポイントは接続の完了と以降のデータ転送について非同期通知を行うように設定することもできます。次に、接続プロセスは t_connect(3N) 関数を使用して転送の設定を開始します。最後に、t_rcvconnect(3N) 関数を使用して接続が確立されたことを確認します。

接続の非同期使用

接続を非同期に待つには、プロセスはまずサービスアドレスに結合した非ブロッキングエンドポイントを確立します。poll(2) の結果または非同期通知によって接続要求が到着したことがわかったら、プロセスは t_listen(3N) 関数を使用して接続要求を取得できます。接続を受け入れるには、プロセスは t_accept(3N) 関数を使

用します。応答するエンドポイントは、非同期データ転送用に別個に設定しなければなりません。

次の例は、接続を非同期に要求する方法を示しています。

```
#include <tiuser.h>
int     fd;
struct t_call  *call;

 fd = .../* 非ブロッキングエンドポイントを確立する */

 call = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR);
 .../* call 構造体を初期化する */
 t_connect(fd, call, call);

 /* 接続要求は現在非同期に進行している */

 .../* 接続が受け付けられたという通知を受け取る */
 t_rcvconnect(fd, &call);
```

次の例は、接続を非同期に監視する方法を示しています。

```
#include <tiuser.h>
int     fd, res_fd;
struct t_call  call;

 fd = ... /* 非ブロッキングエンドポイントを確立する */

 .../* 接続要求が到着したという通知を受け取る */
 call = (struct t_call *) t_alloc(fd, T_CALL, T_ALL);
 t_listen(fd, &call);

 .../* 接続を受け入れるかどうかを決定する */
 res_fd = ... /* 応答用に非ブロッキングエンドポイントを確立する */
 t_accept(fd, res_fd, call);
```

非同期でファイルを開く

アプリケーションは、遠隔ホストからマウントされたファイルシステム内や、初期化が遅れているデバイス上の通常ファイルをダイナミックに開く必要がある場合もあります。ただし、そのようなファイルを開く状態が進行している間は、アプリケーションは他のイベントへ実時間応答できなくなります。この問題は SunOS 5 では、2 番目のプロセスが実際にファイルを開き、そのファイル記述子を実時間プロセスに渡すことによって解決できます。

ファイル記述子の転送

SunOS 5 の STREAMS インタフェースは、開いているファイルの記述子をプロセスからプロセスへ渡す機構を提供しています。開いているファイルの記述子を持つプロセスは、`ioctl(2)` 関数にコマンド引数 `I_SENDFD` を付けて使用します。2 番目

のプロセスは、ioctl() 関数にコマンド引数 I_RECVFD を付けて呼び出し、ファイル記述子を取得します。

コード例 9–1 では、親プロセスがまずテストファイルについての情報を表示し、その後パイプを作成します。次に、親プロセスは子プロセスを作成し、子プロセスはテストファイルを開いて、そのファイル記述子をパイプを通じて親に戻します。親プロセスは、新しいファイル記述子についての状態情報を表示します。

コード例 9–1 ファイル記述子の転送

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stropts.h>
#include <stdio.h>

#define TESTFILE "/dev/null"
main(int argc, char *argv[])
{
 int fd;
 int pipefd[2];
 struct stat statbuf;

 stat(TESTFILE, &statbuf);
 statout(TESTFILE, &statbuf);
 pipe(pipefd);
 if (fork() == 0) {
  close(pipefd[0]);
  sendfd(pipefd[1]);
 } else {
  close(pipefd[1])
  recvfd(pipefd[0]);
 }
}

sendfd(int p)
{
 int tfd;

 tfd = open(TESTFILE, O_RDWR);
 ioctl(p, I_SENDFD, tfd);
}

recvfd(int p)
{
 struct strrecvfd rfdbuf;
 struct stat statbuf;
 char   fdbuf[32];

 ioctl(p, I_RECVFD, &rfdbuf);
 fstat(rfdbuf.fd, &statbuf);
 sprintf(fdbuf, "recvfd=%d", rfdbuf.fd);
 statout(fdbuf, &statbuf);
}

statout(char *f, struct stat *s)
{
```

```
 printf("stat: from=%s mode=0%o, ino=%ld, dev=%lx, rdev=%lx\n",
  f, s->st_mode, s->st_ino, s->st_dev, s->st_rdev);
 fflush(stdout);
}
```

タイマ

この節では、SunOS 5 で実時間アプリケーションのために使用できるタイミング機能について説明します。実時間アプリケーションでこの機能を活用したい場合は、この節に示されているルーチンについて、『*man Pages(3): Library Routines*』を参照してください。

SunOS 5 のタイミング機能は、「タイムスタンプ」と「インタバルタイマ」という 2 つの機能に分けられます。タイムスタンプ機能は経過時間を測定して、アプリケーションが、ある状態の持続時間やイベント間の時間を測定できるようにします。インタバルタイマ機能は、アプリケーションを指定した時間に呼び起こして、アプリケーションが時間の経過に基づいて動作をスケジュールできるようにします。アプリケーションは、タイムスタンプ機能をポーリングして自分をスケジュールすることもできますが、そのようなアプリケーションは、プロセッサを独占して他のシステム関数に悪影響を与えます。

タイムスタンプ機能

タイムスタンプは 2 つの関数によって提供されます。`gettimeofday(3C)` 関数は、グリニッジ標準時間 1970 年 1 月 1 日午前 0 時からの秒数とマイクロ秒数によって時間を表し、現在の時間を `timeval` 構造体に与えます。`clock_gettime(3R)` 関数は、`CLOCK_REALTIME` のクロック ID とともに使用して、`gettimeofday(3C)` が戻すタイムインタバルと同じ時間を秒とナノ秒で表して、現在の時間を `timespec` 構造体に与えます。

SunOS 5 はハードウェア定期タイマを使用します。ある種のワークステーションでは、これが唯一の時間情報で、タイムスタンプの精度はその定期タイマの解像度までに制限されます。その他のプラットフォームでは、1 マイクロ秒の解像度を持つタイマレジスタによって、タイムスタンプ精度は 1 マイクロ秒となっています。

インタバルタイマ機能

実時間アプリケーションは、インタバルタイマを使用して活動をスケジュールすることがよくあります。インタバルタイマには、「単発」型と「周期」型の 2 種類があります。

単発タイマは、現在時間または絶対時間に相対的な有効時間に設定されるタイマです。タイマは、有効時間が終了すると解除されます。このようなタイマは、データを記憶領域に転送した後のバッファの消去や操作のタイムアウトの管理に便利です。

周期タイマには、初期有効時間 (絶対時間または相対時間) と繰り返しインタバルが設定されています。インタバルタイマの有効時間が経過するたびに、タイマは繰り返し再ロードされ、自動的に再度有効になります。このタイマはデータロギングやサーボ制御に便利です。インタバルタイマ機能を呼び出す際は、システムのハードウェア定期タイマの解像度より小さな時間値は、ハードウェア定期タイマインタバル (10 ミリ秒) の時間値より大きい最小の倍数に丸められます。

SunOS 5 には、`setitimer(2)` インタフェースと `getitimer(2)` インタフェースの 2 組のタイマインタフェースがあります。これらのインタフェースは、タイムインタバルを指定する `timeval` 構造体を使用して、BSD タイマと呼ばれる固定設定タイマを動作させます。POSIX タイマである `timer_create(3R)` は、POSIX クロック `CLOCK_REALTIME` を動作させます。POSIX タイマの動作は、`timespec` 構造体によって表されます。

`getitimer(2)` 関数と `setitimer(2)` 関数は、それぞれ指定された BSD インタバルタイマの値の取り出しと設定を行います。プロセスは `ITIMER_REAL` で指定する実時間タイマを含め、3 つの BSD インタバルタイマを利用できます。BSD タイマを使用して有効になっている場合は、システムによってタイマにふさわしいシグナルがタイマを設定したプロセスに送信されます。

`timer_create(3R)` は、{`TIMER_MAX`} 個までの POSIX タイマを生成できます。呼び出し側はタイマの有効時間が経過したときに、どのシグナルと関連値をプロセスに送るかを指定できます。`timer_settime(3R)` と `timer_gettime(3R)` は、指定された POSIX インタバルタイマの値を、それぞれ設定および検索します。必要なシグナルが保留状態にある間に POSIX タイマの有効時間が経過すると配信がカウントされ、`timer_getoverrun(3R)` はそのような有効時間切れのカウントを検索します。`timer_delete(3R)` は、POSIX タイマの割り当てを解除します。

コード例 9–2 に、`setitimer(2)` を使用して定期割り込みを発生させる方法とタイマ割り込みの到着の制御方法を示します。

コード例 **9–2**　タイマ割り込みの制御

```
#include <unistd.h>
#include <signal.h>
#include <sys/time.h>

#define TIMERCNT 8

void timerhandler();
int  timercnt;
struct  timeval alarmtimes[TIMERCNT];

main()
{
 struct itimerval times;
 sigset_t sigset;
 int  i, ret;
 struct sigaction act;
 siginfo_t si;

 /* SIGALRM をブロッキングする */
 sigemptyset(&sigset);
 sigaddset(&sigset, SIGALRM);
 sigprocmask(SIG_BLOCK, &sigset, NULL);

 /* SIGALRM のためのハンドラを設定する */
 act.sa_handler = timerhandler;
 sigemptyset(&act.sa_mask);
 act.sa_flags = SA_SIGINFO;
 sigaction(SIGALRM, &act, NULL);
 /*
  * 3 秒後に開始し、そのあとは 3 分の 1 秒おきに開始するように
  * インタバルタイマを設定する
  */
 times.it_value.tv_sec = 3;
 times.it_value.tv_usec = 0;
 times.it_interval.tv_sec = 0;
 times.it_interval.tv_usec = 333333;
 ret = setitimer(ITIMER_REAL, &times, NULL);
 printf("main:setitimer ret = %d\n", ret);

 /* 現在はアラーム待ち */
 sigemptyset(&sigset);
 timerhandler(0, si, NULL);
 while (timercnt < TIMERCNT) {
  ret = sigsuspend(&sigset);
 }
 printtimes();
}

void timerhandler(sig, siginfo, context)
 int  sig;
 siginfo_t siginfo;
 void  *context;
{
 printf("timerhandler:start\n");
 gettimeofday(&alarmtimes[timercnt], NULL);
 timercnt++;
 printf("timerhandler:timercnt = %d\n", timercnt);
}
```

```
printtimes()
{
 int i;

 for (i = 0; i < TIMERCNT; i++) {
  printf("%ld.%0l6d\n", alarmtimes[i].tv_sec,
    alarmtimes[i].tv_usec);
 }
}
```

付録 **A**

完全なコーディング例

次のプログラムは、priocntl –l ユーティリティの呼び出しと同じです。これは、タイムシェアリングと実時間のスケジューリングクラスに対して有効な優先順位の範囲の取得と印刷を行います。

コード例 **A–1** priocntl() を使用した有効な優先順位の表示

```
 /*
 * スケジューリングクラス ID と優先順位範囲を取得する
 */

#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
main ()
{
 pcinfo_t   pcinfo;
 tsinfo_t   *tsinfop;
 rtinfo_t   *rtinfop;
 short   maxtsupri, maxrtpri;

 /* タイムシェアリング */
  (void) strcpy (pcinfo.pc_clname, "TS");
  if (priocntl (0L, 0L, PC_GETCID, &pcinfo) == -1L) {
   perror ("PC_GETCID failed for time-sharing class");
   exit (1);
  }
  tsinfop = (struct tsinfo *) pcinfo.pc_clinfo;
  maxtsupri = tsinfop->ts_maxupri;
  (void) printf("Time sharing: ID %ld, priority range -%d
      through %d\n",
   pcinfo.pc_cid, maxtsupri, maxtsupri);
```

```
 /* 実時間 */
  (void) strcpy(pcinfo.pc_clname, "RT");
  if (priocntl (0L, 0L, PC_GETCID, &pcinfo) == -1L) {
   perror ("PC_GETCID failed for realtime class");
   exit (2);
  }
 rtinfop = (struct rtinfo *) pcinfo.pc_clinfo;
 maxrtpri = rtinfop->rt_maxpri;
 (void) printf("Real time: ID %ld, priority range 0 through %d\n",
   pcinfo.pc_cid, maxrtpri);
 return (0);
}
```

次に、getcid というこのプログラムの出力例を示します。

```
$ getcid Time sharing: ID 1, priority range -20 through 20 Real time: ID 2,
priority range 0 through 59
```

次の例は、PC_GETCLINFO を使用して、プロセス ID を元にしてプロセスのクラス名を取得します。

コード例 A–2 priocntl() を使用したクラス名の取得

```
/* プロセス ID を指定してスケジューリングクラス名を取得する */

#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

main (argc, argv)
 int argc;
 char *argv[];
{
 pcinfo_t   pcinfo;
 id_t   pid, classID;
 id_t   getclassID();

 if ((pid = atoi(argv[1])) <= 0) {
  perror ("bad pid");
  exit (1);
 }
 if ((classID = getclassID(pid)) == -1) {
  perror ("unknown class ID");
  exit (2);
 }
 pcinfo.pc_cid = classID;
 if (priocntl (0L, 0L, PC_GETCLINFO, &pcinfo) == -1L) {
  perror ("PC_GETCLINFO failed");
  exit (3);
 }
 (void) printf("process ID %d, class %s\n", pid,
  pcinfo.pc_clname);
```

```
}

/*
 * ID が pid のプロセスのスケジューリングクラス ID を戻す
 */

getclassID (pid)
 id_t pid;
{
 pcparms_t   pcparms;

 pcparms.pc_cid = PC_CLNULL;
 if (priocntl(P_PID, pid, PC_GETPARMS, &pcparms) == -1) {
  return (-1);
 }
 return (pcparms.pc_cid);
}
```

次のプログラムは、プロセス ID を入力として受け取り、プロセスを「最高有効優先順位 - 1」の実時間プロセスにし、その優先順位のデフォルトのタイムスライスをプロセスに与えます。プログラムは schedinfo 関数を呼び出して、実時間クラス ID と最高優先順位を取得します。

コード例 A–3　priocntl() を使用した指定プロセスの実時間への変換

```
/*
 * 入力引数はプロセス ID。
 * プロセスを最高優先順位 - 1 の実時間プロセスにする
 */
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

main (argc, argv)
 int argc;
 char *argv[];
{
 pcparms_t  pcparms;
 rtparms_t  *rtparmsp;
 id_t  pid, rtID;
 id_t  schedinfo();
 short  maxrtpri;
 if ((pid = atoi(argv[1])) <= 0) {
  perror ("bad pid");
  exit (1);
 }

 /* 実時間プロセスの優先順位の最大を取得する */
 if ((rtID = schedinfo ("RT", &maxrtpri)) == -1) {
  perror ("schedinfo failed for RT");
  exit (2);
```

```
 }

 /* プロセスを最高優先順位 - 1、デフォルトのタイムスライスの RT に変更する */
 pcparms.pc_cid = rtID;
 rtparmsp = (struct rtparms *) pcparms.pc_clparms;
 rtparmsp->rt_pri = maxrtpri - 1;
 rtparmsp->rt_tqnsecs = RT_TQDEF;

 if (priocntl(P_PID, pid, PC_SETPARMS, &pcparms) == -1) {
  perror ("PC_SETPARMS failed");
  exit (3);
 }
}

/*
 * クラス ID と最高優先順位を戻す。
 * 入力引数名はクラス名。
 * 最高優先順位は *maxpri に戻される
 */

id_t
schedinfo (name, maxpri)
 char *name;
 short *maxpri;
{
 pcinfo_t   info;
 tsinfo_t   *tsinfop;
 rtinfo_   *rtinfop;

 (void) strcpy(info.pc_clname, name);
 if (priocntl (0L, 0L, PC_GETCID, &info) == -1L) {
  return (-1);
 }
 if (strcmp(name, "TS") == 0) {
  tsinfop = (struct tsinfo *) info.pc_clinfo;
  *maxpri = tsinfop->ts_maxupri;
 } else if (strcmp(name, "RT") == 0) {
  rtinfop = (struct rtinfo *) info.pc_clinfo;
  *maxpri = rtinfop->rt_maxpri;
 } else {
  return (-1);
 }
 return (info.pc_cid);
}
```

次に、priocntlset が有用な例を示します。プログラムが 1 つのユーザ ID だけで実行する実時間プロセスとタイムシェアリングプロセスの両方を持っていると想定します。タイムシェアリングプロセスを実時間プロセスに変えずに、実時間プロセスだけの優先順位を変更したい場合、プログラムは次のようになります。

コード例 A–4　priocntlset() を使用したプロセス優先順位の変更

```
/*
 * この uid の実時間優先順位を
 * 実時間プロセスの最高優先順位 - 1 に変更する
 */
```

```
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

main (argc, argv)
 int argc;
 char *argv[];
{
 procset_t    procset;
 pcparms_t    pcparms;
 struct rtparms    *rtparmsp;
 id_t    rtclassID;
 id_t    schedinfo();
 short    maxrtpri;

 /* このプロセスと同じ uid のプロセスを選択する */
 procset.p_lidtype = P_UID;
 procset.p_lid = getuid();

 /* 実時間クラスに関する情報を取得する */
 if ((rtclassID = schedinfo ("RT", &maxrtpri)) == -1) {
  perror ("schedinfo failed");
  exit (1);
 }

...
}

/*
 * クラス ID と最高優先順位を戻す。
 * 入力引数名はクラス名。
 * 最高優先順位は *maxpri に戻される
 */

id_t
schedinfo (name, maxpri)
 char *name;
 short *maxpri;
{
 pcinfo_t   info;
 tsinfo_t   *tsinfop;
 rtinfo_   *rtinfop;

 (void) strcpy(info.pc_clname, name);
 if (priocntl (0L, 0L, PC_GETCID, &info) == -1L) {
  return (-1);
 }
 if (strcmp(name, "TS") == 0) {
  tsinfop = (struct tsinfo *) info.pc_clinfo;
  *maxpri = tsinfop->ts_maxupri;
 } else if (strcmp(name, "RT") == 0) {
  rtinfop = (struct rtinfo *) info.pc_clinfo;
  *maxpri = rtinfop->rt_maxpri;
 } else {
  return (-1);
```

```
 }
 return (info.pc_cid);
}
```

コード例 **A–5**　msgget() を使用したプログラム

```
/*
 * msgget.c: msgget() 関数の例を示す。
 * このプログラムは msgget() 関数の簡単な例である。
 * 引数を要求し、呼び出しを行い、結果を報告する
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

extern void  exit();
extern void  perror();

main()
{
 key_t  key;  /* msgget() に渡すキー */
 int  msgflg,  /* msgget() に渡す msgflg */
   msqid;  /* msgget() からの戻り値 */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);
 (void) fprintf(stderr, "\nExpected flags for msgflg argument
are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter msgflg value: ");
 (void) scanf("%i", &msgflg);

 (void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,
%#o)\n",
 key, msgflg);
 if ((msqid = msgget(key, msgflg)) == -1)
 {
  perror("msgget: msgget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
   "msgget: msgget succeeded: msqid = %d\n", msqid);
```

```
  exit(0);
 }
}
```

コード例 **A–6**　msgctl() を使用したプログラム

```
/*
 * msgctl.c:  msgctl() 関数の例を示す。
 *
 * これは msgctl() 関数の簡単な例である。
 * このプログラムは、1 つのメッセージ待ち行列に
 * 1 つの制御操作を実行できるようにする。
 * 制御操作が失敗するとただちに処理を中止するので、
 * 読み取り権のないアクセス権の設定にしないように
 * 注意が必要である。
 * このコードでアクセス権の再設定はできない
 */
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/msg.h>
#include   <time.h>

static void do_msgctl();
extern void exit();
extern void perror();
static char warning_message[] = "If you remove read permission for \
     yourself, this program will fail frequently!";

main()
{
 struct msqid_ds buf;     /* IPC_STAT コマンドと IP_SET コマンドのための
      待ち行列記述子バッファ */
 int    cmd,  /* msgctl() に指定するコマンド */
    msqid;  /* msgctl() に指定する待ち行列 ID */

 (void fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* msgctl() 呼び出しの msgid 引数と cmd 引数を取得する */
 (void) fprintf(stderr,
  "Please enter arguments for msgctls() as requested.");
 (void) fprintf(stderr, "\nEnter the msqid: ");
 (void) scanf("%i", &msqid);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\nEnter the value for the command: ");
 (void) scanf("%i", &cmd);

 switch (cmd) {
  case IPC_SET:
   /* メッセージ待ち行列制御構造体にある設定を修正する */
   (void) fprintf(stderr, "Before IPC_SET, get current values:");
```

```
  /* IPC_STAT 処理にそのまま進む */
 case IPC_STAT:
  /* 現在のメッセージ待ち行列制御構造体のコピーを取得して
   * 表示する */
  do_msgctl(msqid, IPC_STAT, &buf);
  (void) fprintf(stderr, ]
  "msg_perm.uid = %d\n", buf.msg_perm.uid);
  (void) fprintf(stderr,
  "msg_perm.gid = %d\n", buf.msg_perm.gid);
  (void) fprintf(stderr,
  "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
  (void) fprintf(stderr,
  "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
  (void) fprintf(stderr, "msg_perm.mode = %#o, ",
  buf.msg_perm.mode);
  (void) fprintf(stderr, "access permissions = %#o\n",
  buf.msg_perm.mode & 0777);
  (void) fprintf(stderr, "msg_cbytes = %d\n",
      buf.msg_cbytes);
  (void) fprintf(stderr, "msg_qbytes = %d\n",
      buf.msg_qbytes);
  (void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
  (void) fprintf(stderr, "msg_lspid = %d\n",
      buf.msg_lspid);
  (void) fprintf(stderr, "msg_lrpid = %d\n",
      buf.msg_lrpid);
  (void) fprintf(stderr, "msg_stime = %s", buf.msg_stime ?
  ctime(&buf.msg_stime) : "Not Set\n");
  (void) fprintf(stderr, "msg_rtime = %s", buf.msg_rtime ?
  ctime(&buf.msg_rtime) : "Not Set\n");
  (void) fprintf(stderr, "msg_ctime = %s",
      ctime(&buf.msg_ctime));
  if (cmd == IPC_STAT)
   break;
  /* 今度は IPC_SET で続ける */
  (void) fprintf(stderr, "Enter msg_perm.uid: ");
  (void) scanf ("%hi", &buf.msg_perm.uid);
  (void) fprintf(stderr, "Enter msg_perm.gid: ");
  (void) scanf("%hi", &buf.msg_perm.gid);
  (void) fprintf(stderr, "%s\n", warning_message);
  (void) fprintf(stderr, "Enter msg_perm.mode: ");
  (void) scanf("%hi", &buf.msg_perm.mode);
  (void) fprintf(stderr, "Enter msg_qbytes: ");
  (void) scanf("%hi", &buf.msg_qbytes);
  do_msgctl(msqid, IPC_SET, &buf);
  break;
 case IPC_RMID:
 default:
  /* メッセージ待ち行列を削除したか、未知のコマンドが指定された */
  do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
  break;
 }
 exit(0);
}

/*
 * msgctl() に渡す引数の指示を表示し、msgctl() を呼び出し、
 * 結果を報告する。msgctl() は失敗すると復帰しない。
 * この例では、エラーを処理せずに報告するだけである
 */
```

```
static void
do_msgctl(msqid, cmd, buf)
struct msqid_ds    *buf;  /* 待ち行列記述子バッファへのポインタ */
int    cmd,  /* コマンドコード */
    msqid;  /* 待ち行列 ID */
{
 register int rtrn;  /* msgctl() からの戻り値の保持領域 */

 (void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d, %s)\n",
   msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
 rtrn = msgctl(msqid, cmd, buf);
 if (rtrn == -1) {
  perror("msgctl: msgctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "msgctl: msgctl returned %d\n",
      rtrn);
 }
}
```

コード例 **A–7**　msgsnd() と msgrcv() を使用したプログラム

```
/*
 * msgop.c: msgsnd () 関数と msgrcv() 関数の例を示す。
 *
 * これは、メッセージの送信ルーチンと受信ルーチンの簡単な例である。
 * このプログラムを使用して、1 つの待ち行列との間で必要な数の
 * メッセージを送受信できる
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

static int ask();
extern void exit();
extern char *malloc();
extern void perror();

char first_on_queue[] = "-> first message on queue",
 full_buf[] = "Message buffer overflow. Extra message text\
      discarded.";

main()
{
 register int    c;  /* メッセージテキスト入力 */
 int    choice; /* ユーザが入力した操作コード */
 register int    i;  /* mtext のためのループ制御 */
 int    msgflg; /* 操作のためのメッセージフラグ */
 struct msgbuf     *msgp;  /* メッセージバッファへのポインタ */
 int    msgsz;  /* メッセージの大きさ */
 long    msgtyp;  /* 希望のメッセージタイプ */
 int    msqid,  /* 使用するメッセージ待ち行列 ID */
     maxmsgsz, /* 割り当てるメッセージバッファの大きさ */
     rtrn;  /* msgrcv または msgsnd からの戻り値 */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
```

```
(void) fprintf(stderr,
 "\t0x... is interpreted as hexadecimal,\n");
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");
/* メッセージ待ち行列 ID を取得してメッセージバッファを設定する */
(void) fprintf(stderr, "Enter msqid: ");
(void) scanf("%i", &msqid);
/*
 * <sys/msg.h> は、char mtext[1] として定義された
 * メッセージテキストの msgbuf 構造体の定義を含んでいることに
 * 注意。
 * したがって、この定義はテンプレートだけであり、
 * 0 バイトまたは 1 バイトのメッセージだけを送受信したい場合
 * 以外は、直接使用できる構造体定義ではない。
 * これを扱うには、テンプレートを含む十分大きい領域、
 * つまりメッセージテキストのテンプレートフィールドの大きさに、
 * 必要なメッセージテキストのフィールドの大きさを加えたものを
 * malloc する。
 * その後、malloc によって戻されるポインタを、
 * 必要な大きさのメッセージテキストフィールドの msgbuf 構造体
 * として使用できる。
 * msgp が何も指していなくても、
 * sizeof (msgp->mtext) は有効であることにも注意
 */
(void) fprintf(stderr,
 "Enter the message buffer size you want:");
(void) scanf("%i", &maxmsgsz);
if (maxmsgsz < 0) {
 (void) fprintf(stderr, "msgop: %s\n",
   "The message buffer size must be >= 0.");
 exit(1);
}
msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf)
   - sizeof msgp->mtext + maxmsgsz));
if (msgp == NULL) {
 (void) fprintf(stderr, "msgop: %s %d byte messages.\n",
   "could not allocate message buffer for", maxmsgsz);
 exit(1);
}
/* ユーザが終了を望むまで、
 メッセージ操作をループする */
while (choice = ask()) {
 switch (choice) {
 case 1: /*  msgsnd(): 引数を取得して呼び出しを行い、
   結果を報告する */
  (void) fprintf(stderr, "Valid msgsnd message %s\n",
   "types are positive integers.");
  (void) fprintf(stderr, "Enter msgp->mtype: ");
  (void) scanf("%li", &msgp->mtype);
  if (maxmsgsz) {
   /* scanf を使用しているので、
      メッセージテキストの読み取りを開始する前に、
        入力されたメッセージタイプ行の入力の残りを捨てるため
      下のループが必要 */
   while ((c = getchar()) != '\n' && c != EOF);
   (void) fprintf(stderr, "Enter a %s:\n",
       "one line message");
   for (i = 0; ((c = getchar()) != '\n'); i++) {
    if (i >= maxmsgsz) {
     (void) fprintf(stderr, "\n%s\n", full_buf);
```

```
    while ((c = getchar()) != '\n');
    break;
   }
   msgp->mtext[i] = c;
  }
  msgsz = i;
 } else
  msgsz = 0;
 (void) fprintf(stderr,"\nMeaningful msgsnd flag is:\n");
 (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
  IPC_NOWAIT);
 (void) fprintf(stderr, "Enter msgflg: ");
 (void) scanf("%i", &msgflg);
 (void) fprintf(stderr, "%s(%d, msgp, %d, %#o)\n",
  "msgop: Calling msgsnd", msqid, msgsz, msgflg);
 (void) fprintf(stderr, "msgp->mtype = %ld\n",
     msgp->mtype);
 (void) fprintf(stderr, "msgp->mtext = \"");
 for (i = 0; i < msgsz; i++)
  (void) fputc(msgp->mtext[i], stderr);
  (void) fprintf(stderr, "\"\n");
  rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
  if (rtrn == -1)
   perror("msgop: msgsnd failed");
  else
   (void) fprintf(stderr,
      "msgop: msgsnd returned %d\n", rtrn);
  break;
case 2: /* msgrcv(): 引数を取得して呼び出しを行い、
       結果を報告する */
 for (msgsz = -1; msgsz < 0 || msgsz > maxmsgsz;
    (void) scanf("%i", &msgsz))
  (void) fprintf(stderr, "%s (0 <= msgsz <= %d): ",
      "Enter msgsz", maxmsgsz);
 (void) fprintf(stderr, "msgtyp meanings:\n");
 (void) fprintf(stderr, "\t 0 %s\n", first_on_queue);
 (void) fprintf(stderr, "\t>0 %s of given type\n",
     first_on_queue);
 (void) fprintf(stderr, "\t<0 %s with type <= |msgtyp|\n",
     first_on_queue);
 (void) fprintf(stderr, "Enter msgtyp: ");
 (void) scanf("%li", &msgtyp);
 (void) fprintf(stderr,
     "Meaningful msgrcv flags are:\n");
 (void) fprintf(stderr, "\tMSG_NOERROR =\t%#8.8o\n",
     MSG_NOERROR);
 (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
     IPC_NOWAIT);
 (void) fprintf(stderr, "Enter msgflg: ");
 (void) scanf("%i", &msgflg);
 (void) fprintf(stderr, "%s(%d, msgp, %d, %ld, %#o);\n",
     "msgop: Calling msgrcv", msqid, msgsz,
     msgtyp, msgflg);
 rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
 if (rtrn == -1)
  perror("msgop: msgrcv failed");
 else {
  (void) fprintf(stderr, "msgop: %s %d\n",
      "msgrcv returned", rtrn);
  (void) fprintf(stderr, "msgp->mtype = %ld\n",
```

```
        msgp->mtype);
    (void) fprintf(stderr, "msgp->mtext is: \"");
    for (i = 0; i < rtrn; i++)
     (void) fputc(msgp->mtext[i], stderr);
    (void) fprintf(stderr, "\"\n");
   }
   break;
  default:
   (void) fprintf(stderr, "msgop: operation unknown\n");
   break;
  }
 }
 exit(0);
}

/*
 * 次に何を行うかをユーザに尋ねる。ユーザの選択コードを戻す。
 * ユーザが有効な選択肢を選択するまで復帰しない
 */
static
ask()
{
 int response; /* ユーザの応答 */

 do {
  (void) fprintf(stderr, "Your options are:\n");
  (void) fprintf(stderr, "\tExit =\t0 or Control-D\n");
  (void) fprintf(stderr, "\tmsgsnd =\t1\n");
  (void) fprintf(stderr, "\tmsgrcv =\t2\n");
  (void) fprintf(stderr, "Enter your choice: ");

  /* 応答を事前に設定するので "^D" は終了と解釈される */
  response = 0;
  (void) scanf("%i", &response);
 } while (response < 0 || response > 2);

 return(response);
}
```

コード例 **A–8**　semget() を使用したプログラム

```
/*
 * semget.c: semget() 関数の例を示す。
 *
 * これは semget() 関数の簡単な例である。このプログラムは、
 * 引数を要求し、呼び出しを行い、結果を報告する
*/

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/sem.h>

extern void    exit();
extern void    perror();

main()
{
```

```
 key_t  key;   /* semget() に渡すキー */
 int  semflg;   /* semget() に渡す semflg */
 int  nsems;   /* semget() に渡す nsems */
 int  semid;   /* semget() からの戻り値 */

 (void) fprintf(stderr,
  "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 (void) fprintf(stderr, "Enter nsems value: ");
 (void) scanf("%i", &nsems);
 (void) fprintf(stderr, "\nExpected flags for semflg are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n", IPC_CREAT);
 (void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner alter = \t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup alter = \t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother alter = \t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter semflg value: ");
 (void) scanf("%i", &semflg);
 (void) fprintf(stderr, "\nsemget: Calling semget(%#lx, %
     %#o)\n",key, nsems, semflg);
 if ((semid = semget(key, nsems, semflg)) == -1) {
  perror("semget: semget failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "semget: semget succeeded: semid = %d\n",
   semid);
  exit(0);
 }
}
```

コード例 **A–9**　semctl() を使用したプログラム

```
/*
 * semctl.c: semctl() 関数の例を示す。
 *
 * これは semctl() 関数の簡単な例である。
 * このプログラムによって、1 つの制御操作を 1 つのセマフォのセットに
 * 実行できる。制御操作が失敗するとただちに処理を中止するため、
 * アクセス権を読み取り権なしの設定にしないように注意が必要である。
 * このコードでアクセス権を再設定できない
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#include    <time.h>
```

```
struct semid_ds semid_ds;

static void   do_semctl();
static void   do_stat();
extern char   *malloc();
extern void   exit();
extern void   perror();

char    warning_message[] = "If you remove read permission\
    for yourself, this program will fail frequently!";

main()
{
 union semun    arg;    /* semctl() に渡す共用体 */
 int    cmd,    /* semctl() に指定するコマンド */
     i,    /* 作業領域 */
     semid,    /* semctl() に渡す semid */
     semnum;    /* semctl() に渡す semnum */

 (void) fprintf(stderr,
    "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "Enter semid value: ");
 (void) scanf("%i", &semid);

 (void) fprintf(stderr, "Valid semctl cmd values are:\n");
 (void) fprintf(stderr, "\tGETALL = %d\n", GETALL);
 (void) fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
 (void) fprintf(stderr, "\tGETPID = %d\n", GETPID);
 (void) fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
 (void) fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\tSETALL = %d\n", SETALL);
 (void) fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
 (void) fprintf(stderr, "\nEnter cmd: ");
 (void) scanf("%i", &cmd);

 /* 複数のコマンドで必要な設定操作を行う */
 switch (cmd) {
  case GETVAL:
  case SETVAL:
  case GETNCNT:
  case GETZCNT:
   /* これらのコマンドにセマフォ番号を取得する */
   (void) fprintf(stderr, "\nEnter semnum value: ");
   (void) scanf("%i", &semnum);
   break;
  case GETALL:
  case SETALL:
   /* セマフォ値のためのバッファを割り当てる */
   (void) fprintf(stderr,
    "Get number of semaphores in the set.\n");
   arg.buf = &semid_ds;
   do_semctl(semid, 0, IPC_STAT, arg);
   if (arg.array =
```

```
    (ushort *)malloc((unsigned)
     (semid_ds.sem_nsems * sizeof(ushort)))) {
    /* 必要なものを得た場合は switch からぬける */
    break;
   }
   (void) fprintf(stderr,
    "semctl: unable to allocate space for %d values\n",
    semid_ds.sem_nsems);
   exit(2);
}

/* 指定されたコマンドに必要な引数の残りを取得する */
switch (cmd) {
 case SETVAL:
  /* 1 つのセマフォの値を設定する */
  (void) fprintf(stderr, "\nEnter semaphore value: ");
  (void) scanf("%i", &arg.val);
  do_semctl(semid, semnum, SETVAL, arg);
  /* そのまま進んで結果を検証する */
  (void) fprintf(stderr,
   "Do semctl GETVAL command to verify results.\n");
 case GETVAL:
  /* 1 つのセマフォの値を取得する */
  arg.val = 0;
  do_semctl(semid, semnum, GETVAL, arg);
  break;
 case GETPID:
  /* セマフォに対して semctl(SETVAL)、semctl(SETALL)、
     または semop() の呼び出しが正常に終了した最後の
     プロセスの PID を取得する */
  arg.val = 0;
  do_semctl(semid, 0, GETPID, arg);
  break;
 case GETNCNT:
  /* セマフォ値が増加するのを待っている
     プロセス数を取得する */
  arg.val = 0;
  do_semctl(semid, semnum, GETNCNT, arg);
  break;
 case GETZCNT:
  /* セマフォ値がゼロになるのを待っている
     プロセス数を取得する */
  arg.val = 0;
  do_semctl(semid, semnum, GETZCNT, arg);
  break;
 case SETALL:
  /* セットにあるすべてのセマフォの値を設定する */
  (void) fprintf(stderr,
      "There are %d semaphores in the set.\n",
      semid_ds.sem_nsems);
  (void) fprintf(stderr, "Enter semaphore values:\n");
  for (i = 0; i < semid_ds.sem_nsems; i++) {
   (void) fprintf(stderr, "Semaphore %d: ", i);
   (void) scanf("%hi", &arg.array[i]);
  }
  do_semctl(semid, 0, SETALL, arg);
  /* そのまま進んで結果を検証する */
  (void) fprintf(stderr,
   "Do semctl GETALL command to verify results.\n");
 case GETALL:
```

```
    /* セットにあるすべてのセマフォの値を取得して
       表示する */
    do_semctl(semid, 0, GETALL, arg);
    (void) fprintf(stderr,
        "The values of the %d semaphores are:\n",
        semid_ds.sem_nsems);
    for (i = 0; i < semid_ds.sem_nsems; i++)
     (void) fprintf(stderr, "%d ", arg.array[i]);
    (void) fprintf(stderr, "\n");
    break;
   case IPC_SET:
    /* モードや所有権を修正する */
    arg.buf = &semid_ds;
    do_semctl(semid, 0, IPC_STAT, arg);
    (void) fprintf(stderr, "Status before IPC_SET:\n");
    do_stat();
    (void) fprintf(stderr, "Enter sem_perm.uid value: ");
    (void) scanf("%hi", &semid_ds.sem_perm.uid);
    (void) fprintf(stderr, "Enter sem_perm.gid value: ");
    (void) scanf("%hi", &semid_ds.sem_perm.gid);
    (void) fprintf(stderr, "%s\n", warning_message);
    (void) fprintf(stderr, "Enter sem_perm.mode value: ");
    (void) scanf("%hi", &semid_ds.sem_perm.mode);
    do_semctl(semid, 0, IPC_SET, arg);
    /* そのまま進んで変更を検証する */
    (void) fprintf(stderr, "Status after IPC_SET:\n");
   case IPC_STAT:
    /* 現在のステータスを取得して表示する */
    arg.buf = &semid_ds;
    do_semctl(semid, 0, IPC_STAT, arg);
    do_stat();
    break;
   case IPC_RMID:
    /* セマフォのセットを削除する */
    arg.val = 0;
    do_semctl(semid, 0, IPC_RMID, arg);
    break;
   default:
    /* 未知のコマンドを semctl に渡す */
    arg.val = 0;
    do_semctl(semid, 0, cmd, arg);
    break;
  }
  exit(0);
}

/*
 * semctl() に渡す引数の指示を表示し、semctl() を呼び出し、
 * 結果を報告する。semctl() は失敗すると復帰しない。
 * この例は、エラーを処理しないで
 * 報告するだけである
 */
static void
do_semctl(semid, semnum, cmd, arg)
union semun  arg;
int  cmd,
   semid,
   semnum;
{
 register int      i;   /* 作業領域 */
```

```
 void) fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d,",
    semid, semnum, cmd);
 switch (cmd) {
  case GETALL:
   (void) fprintf(stderr, "arg.array = %#x)\n",
       arg.array);
   break;
  case IPC_STAT:
  case IPC_SET:
   (void) fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
   break;
  case SETALL:
   (void) fprintf(stderr, "arg.array = [", arg.buf);
   for (i = 0;i < semid_ds.sem_nsems;) {
    (void) fprintf(stderr, "%d", arg.array[i++]);
    if (i < semid_ds.sem_nsems)
      (void) fprintf(stderr, ", ");
   }
   (void) fprintf(stderr, "])\n");
   break;
  case SETVAL:
  default:
   (void) fprintf(stderr, "arg.val = %d)\n", arg.val);
   break;
 }
 i = semctl(semid, semnum, cmd, arg);
 if (i == -1) {
  perror("semctl: semctl failed");
  exit(1);
 }
 (void) fprintf(stderr, "semctl: semctl returned %d\n", i);
 return;
}

/*
 * ステータス構造体の一般的に使用される部分の内容を表示する
 */
static void
do_stat()
{
 (void) fprintf(stderr, "sem_perm.uid = %d\n",
       semid_ds.sem_perm.uid);
 (void) fprintf(stderr, "sem_perm.gid = %d\n",
       semid_ds.sem_perm.gid);
 (void) fprintf(stderr, "sem_perm.cuid = %d\n",
       semid_ds.sem_perm.cuid);
 (void) fprintf(stderr, "sem_perm.cgid = %d\n",
       semid_ds.sem_perm.cgid);
 (void) fprintf(stderr, "sem_perm.mode = %#o, ",
       semid_ds.sem_perm.mode);
 (void) fprintf(stderr, "access permissions = %#o\n",
       semid_ds.sem_perm.mode & 0777);
 (void) fprintf(stderr, "sem_nsems = %d\n",
       semid_ds.sem_nsems);
 (void) fprintf(stderr, "sem_otime = %s", semid_ds.sem_otime ?
       ctime(&semid_ds.sem_otime) : "Not Set\n");
 (void) fprintf(stderr, "sem_ctime = %s",
       ctime(&semid_ds.sem_ctime));
}
```

コード例 **A–10**　semop() を使用したプログラム

```
/*
 * semop.c: semop() 関数の例を示す。
 *
 * これは、semop() 関数の簡単な例である。
 * このプログラムによって、
 * semop() の引数の設定と呼び出しを行うことができる。
 * その後、1 つのセマフォのセットに関して繰り返して結果を報告する。
 * セマフォのセットに読み取り権を持っていなければならない。
 * 持っていないと失敗する。
 * (セットにあるセマフォ数を得るため、および semop() の呼び出しの
 * 前後に値を報告するため、読み取り権が必要)
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

static int      ask();
extern void     exit();
extern void     free();
extern char     *malloc();
extern void     perror();

static struct semid_ds  semid_ds;    /* セマフォのセットの状態 */

static char    error_mesg1[] = "semop: Can't allocate space for %d\
        semaphore values. Giving up.\n";
static char   error_mesg2[] = "semop: Can't allocate space for %d\
        sembuf structures. Giving up.\n";

main()
{
 register int    i;   /* 作業領域 */
 int    nsops;   /* 操作数 */
 int    semid;   /* セマフォのセットの semid */
 struct sembuf    *sops;   /* 実行する操作へのポインタ */

 (void) fprintf(stderr,
    "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 /* 呼び出し側が何もしたくなくなるまでループする */
 while (nsops = ask(&semid, &sops)) {
  /* 実行する操作の配列を初期化する */
  for (i = 0; i < nsops; i++) {
   (void) fprintf(stderr,
     "\nEnter values for operation %d of %d.\n",
       i + 1, nsops);
   (void) fprintf(stderr,
     "sem_num(valid values are 0 <= sem_num < %d): ",
     semid_ds.sem_nsems);
   (void) scanf("%hi", &sops[i].sem_num);
   (void) fprintf(stderr, "sem_op: ");
   (void) scanf("%hi", &sops[i].sem_op);
   (void) fprintf(stderr,
```

```
      "Expected flags in sem_flg are:\n");
    (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",
      IPC_NOWAIT);
    (void) fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",
      SEM_UNDO);
    (void) fprintf(stderr, "sem_flg: ");
    (void) scanf("%hi", &sops[i].sem_flg);
   }

   /* 呼び出し時の内容を表示する */
   (void) fprintf(stderr,
      "\nsemop: Calling semop(%d, &sops, %d) with:",
      semid, nsops);
   for (i = 0; i < nsops; i++)
   {
    (void) fprintf(stderr, "\nsops[%d].sem_num = %d, ", i,
       sops[i].sem_num);
    (void) fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
    (void) fprintf(stderr, "sem_flg = %#o\n",
       sops[i].sem_flg);
   }

   /* semop() 呼び出しを実行して結果を報告する */
   if ((i = semop(semid, sops, nsops)) == -1) {
    perror("semop: semop failed");
   } else {
    (void) fprintf(stderr, "semop: semop returned %d\n", i);
   }
  }
}

/*
 * ユーザに続けたいかを尋ねる。
 *
 * 最初の呼び出しで
 * 処理する semid を取得して呼び出し側に返す。
 * 各呼び出しで
 *  1. 現在のセマフォ値を表示する。
 *  2. 次の semop の呼び出しで何回操作を実行するかをユーザに尋ねる。
 *      ジョブに十分な sembuf 構造体の配列を割り当て、
 *      呼び出し側が指定したポインタをその配列に設定する。
 *      (配列が十分に大きい場合は、後続の呼び出しで再使用する。
 *      十分に大きくない場合は配列を解放して、より大きい配列
 *      を割り当てる)
 */
static
ask(semidp, sopsp)
int   *semidp;  /* semid へのポインタ (最初だけ使用する) */
struct sembuf   **sopsp;
{
 static union semun     arg;  /* semctl への引数 */
 int     i;  /* 作業領域 */
 static int     nsops = 0;  /* 現在割り当てられている
           sembuf 配列の大きさ */
 static int     semid = -1;   /* ユーザが指定した semid */
 static struct sembuf    *sops;     /* 割り当てられている配列へのポインタ */

 if (semid < 0) {
  /* 最初の呼び出し: ユーザからの semid と
     セマフォのセットの現在の状態を取得する */
```

```
 (void) fprintf(stderr,
   "Enter semid of the semaphore set you want to use: ");
 (void) scanf("%i", &semid);
 *semidp = semid;
 arg.buf = &semid_ds;
 if (semctl(semid, 0, IPC_STAT, arg) == -1) {
  perror("semop: semctl(IPC_STAT) failed");
  /* semctl が失敗すると、semid_ds はゼロのままであるため、
     セマフォ数の後での検査はゼロになることに注意 */
  (void) fprintf(stderr,
    "Before and after values are not printed.\n");
 } else {
  if ((arg.array = (ushort *)malloc(
   (unsigned)(sizeof(ushort) * semid_ds.sem_nsems)))
     == NULL) {
   (void) fprintf(stderr, error_mesg1,
     semid_ds.sem_nsems);
   exit(1);
  }
 }
}
/* 現在のセマフォ値を表示する */
if (semid_ds.sem_nsems) {
 (void) fprintf(stderr,
     "There are %d semaphores in the set.\n",
     semid_ds.sem_nsems);
 if (semctl(semid, 0, GETALL, arg) == -1) {
  perror("semop: semctl(GETALL) failed");
 } else {
  (void) fprintf(stderr, "Current semaphore values are:");
  for (i = 0; i < semid_ds.sem_nsems;
   (void) fprintf(stderr, " %d", arg.array[i++]));
  (void) fprintf(stderr, "\n");
 }
}
/* 次の呼び出しで行われる操作がいくつになるかを見つけ出し、
   十分なスペースを割り当てる */
(void) fprintf(stderr,
    "How many semaphore operations do you want %s\n",
    "on the next call to semop()?");
(void) fprintf(stderr, "Enter 0 or control-D to quit: ");
i = 0;
if (scanf("%i", &i) == EOF || i == 0)
 exit(0);
if (i > nsops) {
 if (nsops)
  free((char *)sops);
 nsops = i;
 if ((sops = (struct sembuf *)malloc((unsigned)(nsops *
  sizeof(struct sembuf)))) == NULL) {
  (void) fprintf(stderr, error_mesg2, nsops);
  exit(2);
 }
}
*sopsp = sops;
return (i);
}
```

コード例 **A–11**　shmget() を使用したプログラム

```
/*
 * shmget.c: shmget() 関数の例を示す。
 *
 * これは shmget() 関数の簡単な例である。
 * このプログラムは、引数を要求し、呼び出しを行い、結果を報告する
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

extern void    exit();
extern void    perror();

main()
{
 key_t  key;   /* shmget() に渡すキー */
 int  shmflg;   /* shmget() に渡す shmflg */
 int  shmid;   /* shmget() からの戻り値 */
 int  size;   /* shmget() に渡す大きさ */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
    "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* キーを取得する */
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);

 /* セグメントの大きさを取得する */
 (void) fprintf(stderr, "Enter size: ");
 (void) scanf("%i", &size);

 /* shmflg 値を取得する */
 (void) fprintf(stderr,
    "Expected flags for the shmflg argument are:\n");
 (void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n", IPC_CREAT);
 (void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter shmflg: ");
 (void) scanf("%i", &shmflg);

 /* 呼び出しを行い、結果を報告する */
 (void) fprintf(stderr,
     "shmget: Calling shmget(%#lx, %d, %#o)\n",
     key, size, shmflg);
 if ((shmid = shmget (key, size, shmflg)) == -1) {
  perror("shmget: shmget failed");
```

```
  exit(1);
 } else {
  (void) fprintf(stderr,
     "shmget: shmget returned %d\n", shmid);
  exit(0);
 }
}
```

コード例 **A–12**　shmctl() を使用したプログラム

```
/*
 * shmctl.c: shmctl() 関数の例を示す。
 *
 * これは shmctl() 関数の簡単な例である。
 * このプログラムによって、1 つの制御操作を
 * 1 つの共用メモリセグメントに実行できる。
 * (一部の操作はユーザの要求に関係なく行われる。
 * 制御操作が失敗すると、すぐに処理を中止する。
 * 読み取り権のないアクセス権を設定しないように注意。
 * このコードでアクセス権の再設定はできない)
*/

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>
#include   <time.h>
static void   do_shmctl();
extern void   exit();
extern void   perror();

main()
{
 int  cmd;  /* shmctl() のコマンドコード */
 int  shmid;  /* セグメント ID */
 struct shmid_ds  shmid_ds;     /* 結果を保持するための
          共用メモリのデータ構造体 */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
     "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* shmid と cmd を取得する */
 (void) fprintf(stderr,
     "Enter the shmid for the desired segment: ");
 (void) scanf("%i", &shmid);
 (void) fprintf(stderr, "Valid shmctl cmd values are:\n");
 (void) fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
 (void) fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
 (void) fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
 (void) fprintf(stderr, "Enter the desired cmd value: ");
 (void) scanf("%i", &cmd);
```

```
 switch (cmd) {
  case IPC_STAT:
   /* 共用メモリセグメントの状態を取得する */
   break;
  case IPC_SET:
   /* 所有者の UID および GID とアクセス権を設定する */
   /* 現在値を取得して表示する */
   do_shmctl(shmid, IPC_STAT, &shmid_ds);
   /* 読み込む UID、GID、およびアクセス権を設定する */
   (void) fprintf(stderr, "\nEnter shm_perm.uid: ");
   (void) scanf("%hi", &shmid_ds.shm_perm.uid);
   (void) fprintf(stderr, "Enter shm_perm.gid: ");
   (void) scanf("%hi", &shmid_ds.shm_perm.gid);
   (void) fprintf(stderr,
    "Note: Keep read permission for yourself.\n");
   (void) fprintf(stderr, "Enter shm_perm.mode: ");
   (void) scanf("%hi", &shmid_ds.shm_perm.mode);
   break;
  case IPC_RMID:
   /* 最後の接続点が切り離されたらセグメントを削除する */
   break;
  case SHM_LOCK:
   /* 共用メモリセグメントをロッキングする */
   break;
  case SHM_UNLOCK:
   /* 共用メモリセグメントのロッキングを解除する */
   break;
  default:
   /* 未知のコマンドが shmctl に渡される */
   break;
 }
 do_shmctl(shmid, cmd, &shmid_ds);
 exit(0);
}

/*
 * shmctl() に渡す引数を表示し、shmctl() を呼び出し、
 * 結果を報告する。shmctl() は、失敗すると復帰しない。
 * この例は、エラーを処理しないで報告するだけである
 */
static void
do_shmctl(shmid, cmd, buf)
int    shmid,    /* 接続点 */
    cmd;    /* コマンドコード */
struct shmid_ds    *buf;  /* 共用メモリのデータ構造体へのポインタ */
{
 register int    rtrn;  /* 保持領域 */

 (void) fprintf(stderr, "shmctl: Calling shmctl(%d, %d, buf)\n",
  shmid, cmd);
 if (cmd == IPC_SET) {
  (void) fprintf(stderr, "\tbuf->shm_perm.uid == %d\n",
     buf->shm_perm.uid);
  (void) fprintf(stderr, "\tbuf->shm_perm.gid == %d\n",
     buf->shm_perm.gid);
  (void) fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n",
     buf->shm_perm.mode);
 }
 if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
  perror("shmctl: shmctl failed");
```

```
 exit(1);
 } else {
  (void) fprintf(stderr,
      "shmctl: shmctl returned %d\n", rtrn);
 }
 if (cmd != IPC_STAT && cmd != IPC_SET)
  return;

 /* 現在の状態を表示する */
 (void) fprintf(stderr, "\nCurrent status:\n");
 (void) fprintf(stderr, "\tshm_perm.uid = %d\n",
      buf->shm_perm.uid);
 (void) fprintf(stderr, "\tshm_perm.gid = %d\n",
      buf->shm_perm.gid);
 (void) fprintf(stderr, "\tshm_perm.cuid = %d\n",
      buf->shm_perm.cuid);
 (void) fprintf(stderr, "\tshm_perm.cgid = %d\n",
      buf->shm_perm.cgid);
 (void) fprintf(stderr, "\tshm_perm.mode = %#o\n",
      buf->shm_perm.mode);
 (void) fprintf(stderr, "\tshm_perm.key = %#x\n",
      buf->shm_perm.key);
 (void) fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
 (void) fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
 (void) fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
 (void) fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
 (void) fprintf(stderr, "\tshm_atime = %s",
      buf->shm_atime ? ctime(&buf->shm_atime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_dtime = %s",
      buf->shm_dtime ? ctime(&buf->shm_dtime) : "Not Set\n");
 (void) fprintf(stderr, "\tshm_ctime = %s",
      ctime(&buf->shm_ctime));
}
```

コード例 **A–13**　shmat() と shmdt() を使用したプログラム

```
/*
 * shmop.c: shmat() 関数と shmdt() 関数の例を示す。
 *
 * これは、shmat() および shmdt() システムコールの簡単な例である。
 * このプログラムによって、セグメントの接続と切り離しを行い、
 * 接続セグメントとの間で文字列の書き込みと読み取りを行うことができる
 */

#include   <stdio.h>
#include   <setjmp.h>
#include   <signal.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>

#define   MAXnap  4 /* 並列接続の最大数 */

static   ask();
static void  catcher();
extern void  exit();
static   good_addr();
extern void  perror();
```

```
extern char  *shmat();

static struct state     { /* 現在接続されているセグメントの
         内部レコード */
 int  shmid;   /* 接続されているセグメントの shmid */
 char  *shmaddr;   /* 接続点 */
 int  shmflg;   /* 接続で使用されるフラグ */
} ap[MAXnap];      /* 現在接続されているセグメントの状態 */

static int    nap;  /* 現在接続されているセグメント数 */
static jmp_buf   segvbuf;   /* SIGSEGV キャッチングのための
         プロセス状態の保存領域 */

main()
{
 register int    action;   /* 実行するアクション */
 char    *addr;   /* アドレス作業領域 */
 register int    i;   /* 作業領域 */
 register struct state   *p;    /* 現在の状態エントリへのポインタ */
 void   (*savefunc)();  /* SIGSEGV 状態の保持領域 */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 while (action = ask()) {
  if (nap) {
   (void) fprintf(stderr,
      "\nCurrently attached segment(s):\n");
   (void) fprintf(stderr, " shmid address\n");
   (void) fprintf(stderr, "------- ----------\n");
   p = &ap[nap];
   while (p-- != ap) {
    (void) fprintf(stderr, "%6d", p->shmid);
    (void) fprintf(stderr, "%#11x", p->shmaddr);
    (void) fprintf(stderr, " Read%s\n",
     (p->shmflg & SHM_RDONLY) ?
     "-Only" : "/Write");
   }
  } else
   (void) fprintf(stderr,
    "\nNo segments are currently attached.\n");
  switch (action) {
  case 1:   /* 要求する shmat */
   /* 別の接続のための余地があることを検証する */
   if (nap == MAXnap) {
    (void) fprintf(stderr, "%s %d %s\n",
       "This simple example will only allow",
       MAXnap, "attached segments.");
    break;
   }
   p = &ap[nap++];
   /* 引数を取得し、呼び出しを行い、結果を報告し、
    * 現在の状態の配列を更新する */
   (void) fprintf(stderr,
    "Enter shmid of segment to attach: ");
   (void) scanf("%i", &p->shmid);

   (void) fprintf(stderr, "Enter shmaddr: ");
```

```
 (void) scanf("%i", &p->shmaddr);
 (void) fprintf(stderr,
  "Meaningful shmflg values are:\n");
 (void) fprintf(stderr, "\tSHM_RDONLY = \t%#8.8o\n",
  SHM_RDONLY);
 (void) fprintf(stderr, "\tSHM_RND = \t%#8.8o\n",
  SHM_RND);
 (void) fprintf(stderr, "Enter shmflg value: ");
 (void) scanf("%i", &p->shmflg);

 (void) fprintf(stderr,
  "shmop: Calling shmat(%d, %#x, %#o)\n",
  p->shmid, p->shmaddr, p->shmflg);
  p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
 if(p->shmaddr == (char *)-1) {
  perror("shmop: shmat failed");
  nap--;
 } else {
  (void) fprintf(stderr,
   "shmop: shmat returned %#8.8x\n", p->shmaddr);
 }
 break;

case 2:   /* 要求する shmdt */
 /* アドレスを取得し、呼び出しを行い、結果を報告し、
  * 内部状態の一致を行う */
 (void) fprintf(stderr,
  "Enter detach shmaddr: ");
 (void) scanf("%i", &addr);

 i = shmdt(addr);
 if(i == -1) {
  perror("shmop: shmdt failed");
 } else {
  (void) fprintf(stderr,
   "shmop: shmdt returned %d\n", i);
  for (p = ap, i = nap; i--; p++) {
   if (p->shmaddr == addr)
    *p = ap[--nap];
  }
 }
 break;
case 3: /* 要求されたセグメントからの読み取り */
 if (nap == 0)
  break;

 (void) fprintf(stderr, "Enter address of an %s",
  "attached segment: ");
 (void) scanf("%i", &addr);

 if (good_addr(addr))
  (void) fprintf(stderr, "String @ %#x is `%s'\n",
   addr, addr);
 break;

case 4: /* 要求されたセグメントへの書き込み */
 if (nap == 0)
  break;

 (void) fprintf(stderr, "Enter address of an %s",
```

```
    "attached segment: ");
   (void) scanf("%i", &addr);

   /* 読み取り専用の接続セグメントへの書き込みの試みをトラップする
    * SIGSEGV キャッチルーチンを設定する */
   savefunc = signal(SIGSEGV, catcher);

   if (setjmp(segvbuf)) {
    (void) fprintf(stderr, "shmop: %s: %s\n",
     "SIGSEGV signal caught",
     "Write aborted.");
   } else {
    if (good_addr(addr)) {
     (void) fflush(stdin);
     (void) fprintf(stderr, "%s %s %#x:\n",
      "Enter one line to be copied",
      "to shared segment attached @",
      addr);
     (void) gets(addr);
    }
   }
   (void) fflush(stdin);

   /* SIGSEGV を前の状態に復元する */
   (void) signal(SIGSEGV, savefunc);
   break;
  }
 }
 exit(0);
 /* 達していない */
}
/* 次のアクションを求める */
static
ask()
{
 int  response;   /* ユーザの応答 */
 do {
   (void) fprintf(stderr, "Your options are:\n");
   (void) fprintf(stderr, "\t^D = exit\n");
   (void) fprintf(stderr, "\t 0 = exit\n");
   (void) fprintf(stderr, "\t 1 = shmat\n");
   (void) fprintf(stderr, "\t 2 = shmdt\n");
   (void) fprintf(stderr, "\t 3 = read from segment\n");
   (void) fprintf(stderr, "\t 4 = write to segment\n");
   (void) fprintf(stderr,
    "Enter the number corresponding to your choice: ");

   /* 応答を事前設定するので "^D" は終了と解釈される */
   response = 0;
   (void) scanf("%i", &response);
 } while (response < 0 || response > 4);
 return (response);
}
/*
 * SHM_RDONLY フラグセットに接続された共用メモリセグメントへの
 * 書き込みの試みによって発生するキャッチシグナル
 */
/* 使用されている引数 */
static void
catcher(sig)
```

```
{
 longjmp(segvbuf, 1);
 /* 達していない */
}
/*
 * 指定されたアドレスが接続されているセグメントの
 * アドレスであることを検証する。
 * アドレスが有効な場合は 1 を戻し、無効な場合は 0 を戻す
 */
static
good_addr(address)
char *address;
{
 register struct state *p;  /* 接続されているセグメントへのポインタ */

 for (p = ap; p != &ap[nap]; p++)
   if (p->shmaddr == address)
    return(1);
 return(0);
}
```

コード例 A–14 は、リスト要素レコードとしてファイルに格納される二重リンクトリストへのエントリの挿入例を示しています。たとえば、その後に新しいレコードが挿入されるレコードがすでに読み取りロッキングを持っていると想定します。このレコードへのロッキングは、そのレコードを編集できるようにロッキングを変更するか、書き込みロッキングに強化しなければなりません。

ロッキングの強化 (通常は読み取りロッキングから書き込みロッキング) は、他のプロセスがファイルの同じセクションに読み取りロッキングを保持していない場合に認められます。保留状態の書き込みロッキングを持つプロセスがファイルの同じセクションで休眠しているときは、ロッキングの強化は成功し、他の (休眠している) ロッキングは待ちます。書き込みロッキングから読み取りロッキングへの変更には制約はありません。いずれの場合にも、ロッキングは新しいロッキングタイプで再設定されるだけです。lockf 関数は読み取りロッキングを持っていないので、ロッキングの強化はその呼び出しには適用されません。

これら 3 つのレコードへのロッキングは、他のプロセスがその設定をブロッキングしている場合に待ち (休眠) に設定されます。これは、F_SETLKW コマンドで行います。代わりに F_SETLK コマンドを使用すると、fcntl 関数はブロッキングされていると失敗します。その後プログラムは、各エラー復帰セクションでブロッキングされた状態を処理するように変更しなければなりません。

コード例 A–14　ロッキング強化したレコードのロッキング

```
 struct record {
...     /* レコードのデータ部分 */
 off_t prev;     /* リスト内の前のレコードへのインデックス */
 off_t next;     /* リスト内の次のレコードへのインデックス */
};
```

```
/* fcntl(2) を使用したロッキングの強化
 * このルーチンに入るときは、
 *      "here" と "next" に読み取りロッキングがあると想定されている。
 *      "here" と "next" に書き込みロッキングが得られると、
 *           書き込みロッキングを "this" に設定する。
 *           インデックスを "this" レコードに戻す。
 * 書き込みロッキングが得られない場合は、
 *      読み取りロッキングを "here" と "next" に復元する。
 *      その他のすべてのロッキングを解除する。
 *      -1 を戻す
 */

off_t
set3lock (this, here, next)
off_t this, here, next;
{
 struct flock lck;
 lck.l_type = F_WRLCK; /* 書き込みロッキングの設定 */
 lck.l_whence = 0; /* ファイルの先頭からのオフセット l_start */
 lck.l_start = here;
 lck.l_len = sizeof(struct record);

 /* "here" のロッキングを書き込みロッキングに強化する */
 if (fcntl(fd, F_SETLKW, &lck) < 0) {
      return (-1);
 }
 /* "this" を書き込みロッキングでロッキングする */
 lck.l_start = this;
 if (fcntl(fd, F_SETLKW, &lck) < 0) {
  /* "this" のロッキングの失敗で "here" のロッキングを読み取りロッキングに下げる */
  lck.l_type = F_RDLCK;
  lck.l_start = here;
  (void) fcntl(fd, F_SETLKW, &lck);
  return (-1);
 /* "next" のロッキングを書き込みロッキングに強化する */
 lck.l_start = next;
 if (fcntl(fd, F_SETLKW, &lck) < 0) {
  /* "next" のロッキングの失敗で "here" のロッキングを読み取りロッキングに下げ、 */
  lck.l_type = F_RDLCK;
  lck.l_start = here;
  (void) fcntl(fd, F_SETLK, &lck);
  /* "this" のロッキングを解除する */
  lck.l_type = F_UNLCK;
  lck.l_start = this;
  (void) fcntl(fd, F_SETLK, &lck);
  return (-1); /* ロッキングを設定できず、再試行するか終了する */
 }

 return (this);
}
```

コード例 **A–15**　lockf() を使用したレコードの書き込みロッキング

```
/* lockf(3C)
 * このルーチンに入るときは、"here" と "next" にロッキングはないと
 * 想定されている。ロッキングが得られると、"this" にロッキングを設定し、
 * インデックスを "this" レコードに戻す。ロッキングが得られないと、
```

```
 * 他のすべてのロッキングを解除し、-1 を戻す
 */
#include <unistd.h>

long
set3lock (this, here, next)
long this, here, next;
{
 /* "here" をロッキングする */
 (void) lseek(fd, here, 0);
 if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
  return (-1);
 }
 /* "this" をロッキングする */
 (void) lseek(fd, this, SEEK_SET);
 if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
  /* "this" のロッキングが失敗した。"here" のロッキングを解除する */
  (void) lseek(fd, here, 0);
  (void) lockf(fd, F_ULOCK, sizeof(struct record));
  return (-1);
 }
 /* "next" をロッキングする */
 (void) lseek(fd, next, 0);
 if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
  /* "next" のロッキングが失敗した。"here" のロッキングを解除する */
  (void) lseek(fd, here, 0);
  (void) lockf(fd, F_ULOCK, sizeof(struct record));
  /* "this" のロッキングを解除する */
  (void) lseek(fd, this, 0);
  (void) lockf(fd, F_ULOCK, sizeof(struct record));
  return (-1);    /* ロッキングを設定できず、再試行するか終了する */
 }
 return (this);
}
```

索引