プログラミングインタフェース

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

Part No: 816–3977–11
2002 年 9 月

原典: *Programming Interfaces Guide*

Part No: 806-4125-11

Revision A

020716@4333

目次

はじめに

このマニュアルでは、アプリケーション開発者が使用する SunOS™ 5.9 のネットワークとシステムのインタフェースについて説明します。

SunOS 5.9 は UNIX® System V の Release 4 (SVR4) と完全な互換性があり、System V Interface Description (SVID) の第 3 版に準拠しています。SunOS 5.9 は、System V のすべてのネットワークサービスをサポートします。

このマニュアルで紹介するユーティリティとそのオプション、およびライブラリ機能はすべて、SunOS リリース 5.8 のものです。

対象読者

このマニュアルは、初めて SunOS プラットフォームを使用するプログラマや、提供されるインタフェースをより詳細に知りたいプログラマを対象にしています。ネットワーク化されたアプリケーションに追加するインタフェースや機能については、『*ONC+ 開発ガイド*』を参照してください。

このマニュアルでは、読者がプログラミングを基本的に理解しており、C プログラミング言語を熟知して作業し、UNIX オペレーティングシステム (特に、ネットワーク関係の概念) に精通していることを前提としています。UNIX のネットワークの基本については、1998 年に Prentice Hall 社 (Upper Saddle River) から発行された W. Richard Stevens 著の『*UNIX Network Programming*』(第 2 版) を参照してください。

内容の紹介

以下に示す章で、SunOS 5.9 プラットフォームの基本的なシステムインタフェースと基本的なネットワークインタフェースに含まれるサービスおよび特性について説明します。

第 1 章では、メモリーマッピングを作成および管理するインタフェース、高性能なファイル入出力を行うインタフェース、およびその他のメモリー管理関連を制御するインタフェースについて説明します。

第 2 章では、リモート共有メモリー用のアプリケーションプログラミングインタフェース (API) のフレームワークとライブラリ関数について説明します。

第 3 章では、SunOS プロセススケジューラの動作、スケジューラの動作の変更、スケジューラのプロセス管理インタフェースとの対話、および性能について説明します。

第 4 章では、以前の形式の基本的なバッファー付きファイル入出力や、その他の入出力の要素について説明します。

第 5 章では、以前の形式のネットワーク化されていないプロセス間通信について説明します。

第 6 章では、ネットワーク化通信の基本モードであるソケットを使用する方法について説明します。

第 7 章では、XTI と TLI を使用して、トランスポートに依存しないネットワーク化通信を行う方法について説明します。

第 8 章では、ネットワークトランスポートとその構成を選択するためにアプリケーションが使用するネットワーク選択メカニズムについて説明しています。

第 9 章では、SunOS 環境におけるリアルタイムプログラミング機能とその使用方法について説明します。

第 10 章では、Solaris™ アプリケーションバイナリインタフェース (ABI) についてと、アプリケーションが Solaris ABI に準拠しているかを確認するツール、`appcert` および `apptrace` について説明します。

付録 A では、UNIX ドメインソケットについて説明しています。

Sun のオンラインマニュアル

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

表記上の規則

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

表 **P–1** 表記上の規則

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

コード例は次のように表示されます。

- C シェル

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

- C シェルのスーパーユーザー

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

- Bourne シェルおよび Korn シェル

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

- Bourne シェルおよび Korn シェルのスーパーユーザー

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

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

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

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

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

一般規則

- このマニュアルでは、「IA」という用語は、Intel 32 ビットのプロセッサアーキテクチャを意味します。これには、Pentium、Pentium Pro、Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon の各プロセッサ、および AMD、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。

第 1 章

メモリー管理

この章では、アプリケーション開発者から見た SunOS の仮想メモリーについて説明します。

- 13 ページの「メモリー管理インタフェース」では、インタフェースとキャッシュ制御について説明します。
- 16 ページの「ライブラリレベルの動的メモリー」では、ライブラリレベルの動的メモリーの割り当てとデバッグについて説明します。
- 19 ページの「その他のメモリー制御インタフェース」では、その他のメモリー制御インタフェースについて説明します。

メモリー管理インタフェース

仮想メモリー機能を使用するとき、アプリケーションはいくつかのインタフェースを使用します。この節では、このようなインタフェースの要約について説明します。この節ではまた、このようなインタフェースの使用例も示します。

マッピングの作成と使用

mmap(2) は、名前付きファイルシステムオブジェクトのプロセスアドレス空間へのマッピングを確立します。 名前付きファイルシステムオブジェクトは部分的にもプロセスアドレス空間にマッピングできます。この基本的なメモリー管理インタフェースはとても簡潔です。 open(2) を使用してファイルを開いてから、mmap(2) を使用して適切なアクセスオプションと共有オプションを持つマッピングを作成します。その後、ユーザーのアプリケーションを処理します。

mmap(2) でマッピングを確立すると、指定されたアドレス範囲にあった以前のマッピングは置き換えられます。

MAP_SHARED フラグと MAP_PRIVATE フラグはマッピングのタイプを指定します。これらのフラグはどちらか 1 つを指定する必要があります。 MAP_SHARED を設定すると、書き込みが行われたときに、マッピングされたオブジェクト下が変更されます。オブジェクトを変更するとき、これ以外の操作は必要ありません。 MAP_PRIVATE を設定すると、マッピングされたオブジェクトに書き込みが行われたときに、最初の書き込み時にページのコピーが作成されます。以降の書き込みではコピーが参照されます。コピーが作成されるのは、変更されたページだけです。

fork(2) を行なっても、マッピングのタイプは保持されます。

mmap(2) でマッピングを確立した後、呼び出しで使用されたファイル記述子は二度と使用されません。ファイルを閉じても、munmap(2) でマッピングを取り消すまで、マッピングは有効です。新しいマッピングを作成すると、既存のマッピングは失われます。

切り捨ての呼び出しを行うと、マッピングされたファイルが短くなることがあります。(短くなって) 失われた領域にアクセスしようとすると、SIGBUS シグナルが発生します。

/dev/zero をマッピングすると、0 で初期化された仮想メモリーブロックが呼び出し元プログラムに提供されます。ブロックのサイズは、mmap(2) への呼び出しに指定します。次のコードは、このテクニックを使用して、0 で初期化された記憶領域のブロックをプログラム内に作成する例を示しています。このブロックのアドレスはシステムが選択します。

```
removed to fr.ch4/pl1.create.mapping.c
```

デバイスまたはファイルの中には、マッピングによってアクセスされるときだけ使用できるものもあります。たとえば、ビットマップ形式のディスプレイをサポートするときに使用するフレームバッファデバイスなどです。ディスプレイのアドレスを直接操作する場合、ディスプレイ管理アルゴリズムはより簡単に実装できます。

マッピングの削除

munmap(2) は、呼び出し元プロセスの指定されたアドレス範囲にあるページのマッピングをすべて削除します。munmap(2) は、マッピングされていたオブジェクトにはまったく影響しません。

キャッシュ制御

SunOS の仮想メモリーシステムは、プロセッサのメモリーがファイルシステムオブジェクト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調査するために、次のようなインタフェースが提供されています。

mincore

mincore(2) インタフェースは、指定された範囲内のマッピングが示すアドレス空間にメモリーページが存在するかどうかを判断します。mincore がページをチェックしてからデータを返すまでの間にページの状態が変わっている可能性もあるので、mincore が返す情報は最新の状態を示していない場合があります。メモリーに残っていると保証されるのは、ロックされたページだけです。

mlock と munlock

mlock(3C) は、指定されたアドレス範囲内にあるページを物理メモリーにロックします。当該プロセスまたはほかのプロセスでロックされたページを参照しても、入出力操作が必要になるページフォルトは発生しません。このような入出力操作は仮想メモリーの通常の動作を妨害し、ほかのプロセスを遅くするので、mlock の使用はスーパーユーザーだけに制限されます。メモリーにロックできるページ数の制限はシステム構成によって異なります。この制限を超えると、mlock の呼び出しは失敗します。

munlock は、物理ページ上にロックされたページを解放します。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 でロックを解放できます。ただし、mlock で同じページを異なるマッピングで処理した場合、このページのロックを解除するには、すべてのマッピングを解放する必要があります。

マッピングを削除することによってもロックを解放できます。つまり、mmap(2) でマッピングを置き換えるか、munmap(2) でマッピングを削除することで可能です。

前述の MAP_PRIVATE マッピングに関連する書き込み時コピーイベントは、コピー元ページからコピー先ページにロックを転送します。したがって、書き込み時コピー先を変更しても、MAP_PRIVATE マッピングを含むアドレス範囲上のロックは透過的に保持されます。この変更については、13 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 と munlockall

mlockall(3C) と munlockall(3C) は mlock と munlock に似ていますが、mlockall と munlockall はアドレス空間全体に対して動作します。mlockall はアドレス空間にあるすべてのページにロックを設定し、munlockall はアドレス空間にある (mlock または mlockall で確立された) すべてのページのロックを解除します。

msync

msync(3C) は、指定されたアドレス範囲内にある変更されたページのすべてを、これらのアドレスでマッピングされているオブジェクトにフラッシュ (実際に書き込み) します。このコマンドは fsync(3C) に似ていますが、こちらはファイルに対して動作します。

ライブラリレベルの動的メモリー

ライブラリレベルの動的メモリー割り当ては、動的メモリー割り当てに使いやすいインタフェースを提供します。

動的メモリー割り当て

最もよく使用されるインタフェースは次のとおりです。

- `malloc`(3C)
- `free`(3C)
- `calloc`(3C)
- `cfree`(3MALLOC)

その他の動的メモリー割り当てインタフェースは `memalign` (3C)、`valloc`(3C)、および `realloc` (3C) です。

- `malloc` は、最低でも要求されたメモリー量のメモリーブロックへのポインタを返します。この (メモリー) ブロックは、任意のタイプのデータを格納できるように境界整列されます。
- `free` は、`malloc`、`calloc`、`realloc`、`memalign`、または `valloc` で取得したメモリーをシステムメモリーに返します。動的メモリー割り当てインタフェースによる予約をしてないブロックを解放しようとするとエラーが発生し、プロセスがクラッシュする可能性があります。
- `calloc` は、0 で初期化されたメモリーブロックへのポインタを返します。`calloc` で予約されたメモリーをシステムに返すには、`cfree`、`free` いずれでも可能です。このメモリー (ブロック) は、指定されたサイズの要素数からなる配列を格納できるように割り当ておよび境界整列されます。
- `memalign` は、指定されたバイト数を指定された境界上に割り当てます。境界は 2 のべき乗である必要があります。
- `valloc` は、指定されたバイト数をページ境界上に整列して割り当てます。
- `realloc` は、プロセスに割り当てられているメモリーブロックのサイズを変更します。`realloc` は、割り当てられているメモリーブロックのサイズを増減するのに使用できます。`realloc` は、問題を起こさずにメモリー割り当てを減らすことができる唯一の方法です。再割り当てされたメモリーブロックの位置は変更される可能性がありますが、その内容はメモリー割り当てのサイズが変更されるまで変更されません。

動的メモリーデバッグ

Sun™ WorkShop ツールパッケージを使用すると、動的メモリーの使用中に発生するエラーを発見して除外することができます。Sun WorkShop の Run Time Checking (RTC) 機能は、動的メモリーの使用中に発生するエラーを発見します。

-g オプションを付けてプログラムをコンパイルしなくても、RTC はすべてのエラーを発見できます。しかし、 特に初期化されていないメモリーから読み取る場合、エラーの正確性を保証するために、(-g で入手できる) シンボリック情報が必要になることもあります。したがって、シンボリック情報が入手できないと、ある種のエラーは抑制されます。このようなエラーには、a.out の rui や共有ライブラリの rui + aib + air があります。この動作を変更するには、suppress と unsuppress を使用します。

check -access

-access オプションは、アクセス権のチェックをオンにします。RTC は次のようなエラーを報告します。

baf　不正な解放

duf　重複する解放

maf　整列されていない解放

mar　整列されていない読み取り

maw　整列されていない書き込み

oom　メモリー不足

rua　割り当てられていないメモリーからの読み取り

rui　初期化されていないメモリーからの読み取り

rwo　読み取り専用メモリーへの書き込み

wua　割り当てられていないメモリーへの書き込み

デフォルトの動作は、アクセス権エラーを発見するたびにプロセスを停止します。この動作を変更するには、rtc_auto_continue dbxenv 変数を使用します。on に設定した場合、RTC はアクセス権エラーをファイルに記録します。このファイル名は rtc_error_log_file_name dbxenv 変数で決定されます。デフォルトでは、一意なアクセス権エラーごとにエラーが発生した最初の時刻だけが報告されますが 、rtc_auto_suppress dbxenv 変数を使用して変更できます。この変数のデフォルト設定は on です。

check -leaks [-frames *n*] [-match *m*]

-leaks オプションは、リークのチェックをオンにします。RTC は次のようなエラーを報告します。

aib　メモリーリークの可能性 – 唯一のポインタがブロックの真ん中を指しています。

air　メモリーリークの可能性 – ブロックへのポインタがレジスタだけに存在します。

mel　メモリーリーク – ブロックへのポインタが存在しません。

リークのチェックをオンにした場合、プログラムが終了したとき、リークレポートが自動的に報告されます。このとき、潜在的なリークを含むすべてのリークが報告されます。デフォルトでは、簡易レポートが生成されます。このデフォルトは dbxenv rtc_mel_at_exit を使用すると変更できます。リークレポートはいつでも要求できます。

-frames *n* 変数を使用した場合、リークが報告されるとき、*n* 個までのスタックフレームが個別に表示されます。-match *m* 変数を使用した場合、リークは結合されて表示されます。複数のリークが発生した割り当て時に、呼び出しスタックが *m* 個のフレームに一致した場合、これらのリークは結合されて、単一のリークレポートとして報告されます。*n* のデフォルト値は 8 または *m* の大きい方ですが、16 が上限です。 *m* のデフォルト値は 2 です。

check -memuse [-frames *n*] [-match *m*]

-memuse オプションはメモリー (ブロック) 使用状況のチェック (memuse) をオンにします。check -memuse を使用すると、check -leaks も自動的に使用されます。つまり、プログラムが終了したとき、リークレポートに加えて、(メモリー) ブロック使用状況レポート (biu) が報告されます。デフォルトでは、簡易 (メモリー) ブロック使用状況レポートが生成されます。このデフォルトは dbxenv rtc_biu_at_exit を使用すると変更できます。プログラムの実行中はいつでも、プログラム内のメモリーがどこに割り当てられているかを参照できます。

次の節では、-frames *n* と -match *m* 変数の機能について説明します。

check -all [-frames *n*] [-match *m*]

check -access; check -memuse [-frames *n*] [-match *m*] と同じです。rtc_biu_at_exit dbxenv 変数の値は check -all では変更されません。そのため、デフォルトでは、プログラムが終了したとき、メモリー (ブロック) 使用状況レポートは作成されません。

check [*funcs*] [*files*] [*loadobjects*]

check -all; suppress all; unsuppress all in *funcs files loadobjects* と同じです。このオプションを使用すると、気になる場所に RTC を集中させることができます。

その他のメモリー制御インタフェース

sysconf

sysconf(3C) は、メモリーページのシステム依存サイズを返します。移植性のため、アプリケーションはページのサイズを指定する定数を埋め込まないでください。同じ命令セットの実装においても、ページのサイズが異なることは特に珍しいことではありません。

mprotect

mprotect(2) は、指定されたアドレス範囲内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。保護は、配下のオブジェクトに許可されたアクセス権を超えることはできません。

brk と sbrk

break は、スタック内には存在しないプロセスイメージにおいて最大の有効なデータアドレスです。プログラムが実行を開始するとき、ブレーク値は通常、execve(2) によって、プログラムとそのデータ記憶領域によって定義される最大のアドレスに設定されます。

brk(2) を使用すると、さらに大きなアドレスにブレークを設定できます。また、sbrk(2) を使用すると、プロセスのデータセグメントに記憶領域の増分を追加できます。getrlimit(2) の呼び出しを使用すると、データセグメントの可能な最大サイズを取得できます。

```
caddr_t
 brk(caddr_t addr);

 caddr_t
 sbrk(intptr_t incr);
```

brk は、呼び出し元プログラムが使用していないデータセグメントの最低の位置を *addr* に設定します。この位置は、システムページサイズの次の倍数に切り上げられます。

sbrk (代替のインタフェース) は、呼び出し元プログラムのデータ空間に *incr* バイトを追加して、新しいデータ領域の開始場所へのポインタを返します。

第 2 章

リモート共有メモリー API (Solaris クラスタ用)

Solaris Cluster OS™ システムは、メモリーベース相互接続 (Dolphin-SCI など) と階層化システムソフトウェア構成要素で構成できます。このような構成要素は、リモートノード上に存在するメモリーへの直接アクセスに基づいて、ユーザーレベルのノード間メッセージング用メカニズムを実装します。このメカニズムのことを「リモート共有メモリー (RSM)」と呼びます。この章では、RSM アプリケーションプログラミングインタフェース (RSMAPI) について説明します。

- 22 ページの「API フレームワーク」では、RSMAPI フレームワークについて説明します。
- 23 ページの「API ライブラリ関数」では、RSMAPI ライブラリ関数について説明します。
- 47 ページの「RSMAPI の使用例」では、RSMAPI の使用例について説明します。

共有メモリーモデルの概要

共有メモリーモデルでは、まず、あるアプリケーションプロセスがプロセスのローカルアドレス空間から RSM エクスポートセグメントを作成します。次に、1 つまたは複数のリモートアプリケーションプロセスが相互接続上のエクスポートセグメントとインポートセグメント間の仮想接続を使用して、RSM インポートセグメントを作成します。共有セグメントのメモリー参照を行うときには、どのアプリケーションプロセスもローカルなアドレス空間のアドレスを使用します。

アプリケーションプロセスは、ローカルでアドレス可能なメモリーをエクスポートセグメントに割り当てることによって、RSM エクスポートセグメントを作成します。この割り当てには、System V Shared Memory、`mmap`(2)、`valloc`(3C) などの標準の Solaris インタフェースの 1 つを使用します。次に、アプリケーションプロセスはセグメントを作成する RSMAPI を呼び出して、割り当てられたメモリーに参照ハンドルを

提供します。RSM セグメントは 1 つまたは複数の相互接続コントローラを通じて公開されます。公開されたセグメントは、リモートからアクセスできるようになります。セグメントをインポートすることが許可されたノードのアクセス権リストも公開されます。

エクスポートされるセグメントにはセグメント ID が割り当てられます。このセグメント ID (および、作成するプロセスのクラスタノード ID) を使用すると、インポートしているプロセス (インポータ) はエクスポートセグメントを一意に指定できます。エクスポートセグメントが正常に作成されると、後続のセグメント操作で使用するための RSM エクスポートセグメントハンドルがプロセスに返されます。

アプリケーションプロセスは RSMAPI を使用して、公開されたセグメントへのアクセス権を取得し、インポートセグメントを作成します。インポートセグメントを作成した後、アプリケーションプロセスは相互接続間に仮想接続を確立します。インポートセグメントが正常に作成されると、後続のセグメントインポート操作で使用するための RSM インポートセグメントハンドルがアプリケーションプロセスに返されます。メモリーマッピングが相互接続によってサポートされている場合、仮想接続を確立した後、アプリケーションは RSMAPI を要求して、ローカルアクセス用にメモリーマップを提供できます。メモリーマッピングがサポートされていない場合、アプリケーションは RSMAPI が提供するメモリーアクセスプリミティブを使用できます。

RSMAPI は、リモートアクセスエラー検出をサポートし、書き込み順番メモリーモデルに関する問題を解決するための機構を提供します。この機構のことを「*barrier*」と呼びます。

RSMAPI が提供する通知メカニズムを使用すると、ローカルアクセスとリモートアクセスの同期をとることができます。つまり、インポートプロセスがデータ書き込み操作を終了するまで、エクスポートプロセスはデータの処理をブロックする関数を呼び出すことができます。書き込み操作が終了すると、インポートプロセスはシグナル関数を呼び出してエクスポートプロセスのブロックを解除します。ブロックが解除された後、エクスポートプロセスはデータを処理できます。

API フレームワーク

RSM アプリケーションサポート構成要素は次のソフトウェアパッケージで配信されます。

- `SUNWrsm`
  - RSMAPI 関数をエクスポートする共有ライブラリ (`/usr/lib/librsm.so`)
  - ユーザーライブラリの代わりに RSMAPI インタフェースを通じてメモリー相互接続ドライバとインタフェースをとる Kernel Agent (KA) 仮想ドライバ (`/usr/kernel/drv/rsm`)
  - 相互接続トポロジを取得するためのクラスタインタフェースモジュール
- `SUNWrsmop`

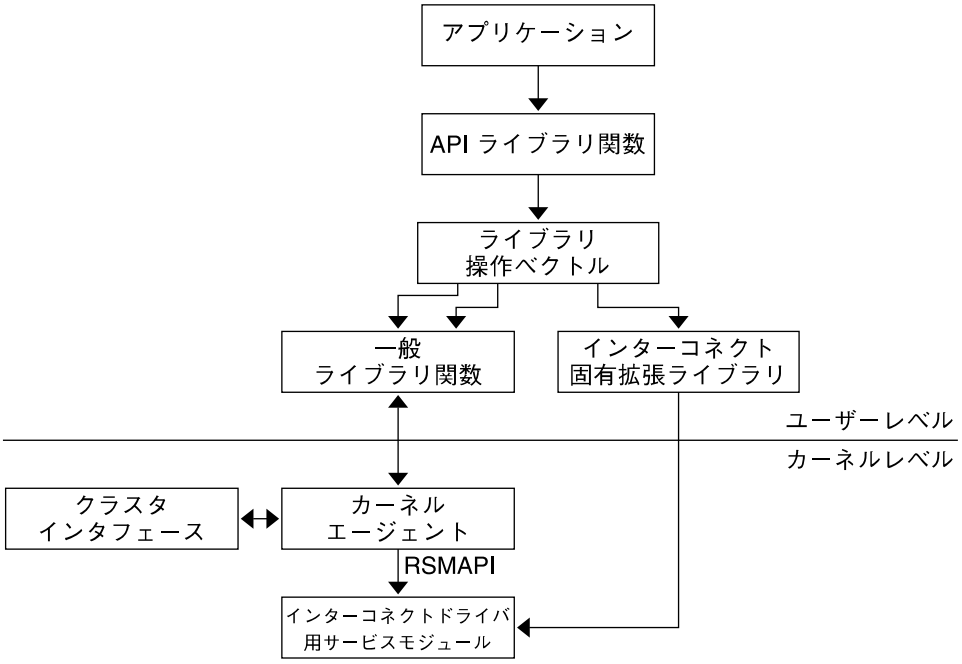
相互接続ドライバサービスモジュール (`/kernel/misc/rsmops`)

- `SUNWrsmdk`

  API 関数とデータ構造体のプロトタイプを提供するヘッダーファイル (`/opt/SUNWrsmdk/include`)

- `SUNW`*interconnect*

  システムに構成されている固有な相互接続の RSM サポートを提供する `librsm.so` へのオプション拡張。拡張はライブラリ (`librsm` *interconnect* `.so`) の形式で提供されます。



API ライブラリ関数

API ライブラリ関数は次の操作をサポートします。

- 相互接続コントローラ操作
- クラスタトポロジ操作
- メモリーセグメント操作 (セグメント管理とデータアクセスを含む)
- バリア操作
- イベント操作

相互接続コントローラ操作

コントローラ操作は、コントローラへのアクセスを取得するメカニズムを提供します。コントローラ操作はまた、配下の相互接続の特性も決定します。相互接続コントローラ操作には、次のような操作が含まれます。

- コントローラの取得
- コントローラ属性の取得
- コントローラの解放

rsm_get_controller

int **rsm_get_controller**(*char *name, rsmapi_controller_handle_t *controller*);

rsm_get_controller は、指定されたコントローラのインスタンス (sci0 や loopback など) のコントローラハンドルを取得します。返されるコントローラハンドルは後続の RSM ライブラリ呼び出しに使用されます。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_CTLR_HNDL	コントローラハンドルが無効です。
RSMERR_CTLR_NOT_PRESENT	コントローラが存在しません。
RSMERR_INSUFFICIENT_MEM	メモリーが不足しています。
RSMERR_BAD_LIBRARY_VERSION	ライブラリのバージョンが無効です。
RSMERR_BAD_ADDR	アドレスが不正です。

rsm_release_controller

int **rsm_release_controller**(*rsmapi_controller_handle_t chdl*);

この関数は、指定されたコントローラハンドルに関連するコントローラを解放します。rsm_release_controller の呼び出しごとに対応する rsm_get_controller が存在する必要があります。つまり、コントローラに関連付けられたコントローラハンドルをすべて解放すると、コントローラに関連付けられたシステム資源が解放されます。コントローラハンドルにアクセスしたり、解放されたコントローラハンドル上のインポートセグメントまたはエクスポートセグメントにアクセスしたりすることは不正です。このような場合の結果は定義されていません。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_CTLR_HNDL	コントローラハンドルが無効です。

rsm_get_controller_attr

int **rsm_get_controller_attr**
(*rsmapi_controller_handle_t chdl, rsmapi_controller_attr_t *attr*);

この関数は、指定されたコントローラハンドルの属性を取得します。この関数に現在定義されている属性は次のとおりです。

```
typedef struct {
     uint_t       attr_direct_access_sizes;
     uint_t       attr_atomic_sizes;
     size_t       attr_page_size;
     size_t       attr_max_export_segment_size;
     size_t       attr_tot_export_segment_size;
     ulong_t      attr_max_export_segments;
     size_t       attr_max_import_map_size;
     size_t       attr_tot_import_map_size;
     ulong_t      attr_max_import_segments;
 } rsmapi_controller_attr_t;
```

戻り値**:** 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_CTLR_HNDL	コントローラハンドルが無効です。
RSMERR_BAD_ADDR	アドレスが不正です。

クラスタトポロジ操作

エクスポート操作とインポート操作に必要な鍵となる相互接続データは次のとおりです。

- エクスポートクラスタのノード ID
- インポートクラスタのノード ID
- コントローラ名

基本的な制約として、インポートセグメント用に指定されたコントローラは、関連するエクスポートセグメント用に使用されるコントローラと物理的に接続されている必要があります。このインタフェースが定義する相互接続トポロジによって、アプリケーションは効率的なエクスポートポリシーとインポートポリシーを確立できます。提供されるデータには、各ローカルコントローラのローカルノード ID、ローカルコントローラインスタンス名、およびリモート接続指定が含まれます。

メモリーをエクスポートするアプリケーション構成要素は、インタフェースが提供するデータを使用して、既存のローカルコントローラセットを発見します。インタフェースが提供するデータはまた、セグメントを作成および発行するためのコントローラを正しく割り当てるために使用できます。アプリケーション構成要素は、ハードウェア相互接続とアプリケーションソフトウェアディストリビューションに整合性があるコントローラセットを使用して、エクスポートされたセグメントを効率的に分散できます。

メモリーをインポートするアプリケーション構成要素は、メモリーのエクスポートで使用されるセグメント ID とコントローラを通知する必要があります。この情報は通常、事前定義されているセグメントとコントローラのペアによって伝達されます。メモリーをインポートしている構成要素はトポロジデータを使用して、セグメントインポート操作に適切なコントローラを決定できます。

rsm_get_interconnect_topology

int **rsm_get_interconnect_topology**(*rsm_topology_t **topology_data*);

この関数は、アプリケーションポインタによって指定された場所にあるトポロジデータへのポインタを返します。トポロジデータ構造体は次のように定義されます。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_TOPOLOGY_PTR	トポロジポインタが無効です。
RSMERR_INSUFFICIENT_MEM	メモリーが足りません。
RSMERR_BAD_ADDR	メモリーが不足しています。

rsm_free_interconnect_topology

void **rsm_free_interconnect_topology**(*rsm_topology_t *topology_data*);

rsm_free_interconnect_topology 操作は、rsm_get_interconnect_topology で割り当てられたメモリーを解放します。

戻り値: ありません。

データ構造体

rsm_get_topology_data から返されるポインタは rsm_topology_t structure を参照します。この構造体は、各ローカルコントローラのローカルノード ID と connections_t 構造体へのポインタの配列を提供します。

```
typedef struct rsm_topology {
   rsm_nodeid_t    local_nodeid;
   uint_t          local_cntrl_count;
   connections_t   *connections[1];
} rsm_topology_t;
```

管理操作

RSM セグメント ID はアプリケーションが指定するか、システムが rsm_memseg_export_publish() 関数を使用して生成します。セグメント ID を指定するアプリケーションは、予約されたセグメント ID の範囲を使用する必要があります。セグメント ID の範囲を予約するには、rsm_get_segmentid_range 関数を使用して、予約されたセグメント ID の範囲をセグメント ID 構成ファイル /etc/rsm/rsm.segmentid に定義します。rsm_get_segmentid_range 関数を使用すると、アプリケーションは自分用に予約されたセグメントID の範囲を取得できます。この関数は、指定されたアプリケーション ID の /etc/rsm/rsm.segmentid ファイルに定義されているセグメント ID の範囲を読み取ります。

アプリケーション ID はアプリケーションを識別するための NULL で終了する文字列です。アプリケーションは baseid 以上で baseid+length 未満の値を使用できます。baseid または length が変更された場合、アプリケーションに返されるセグメント ID は予約された範囲内ではない場合がありますので、セグメント ID を取得するときには、予約されたセグメント ID の範囲内のオフセットを使用してください。

/etc/rsm/rsm.segmentid ファイル内のエントリは次のような形式です。

```
#keyword      appid      baseid       length
reserve       SUNWfoo    0x600000     100
```

エントリを構成する文字列は、タブまたは空白で区切ることができます。この文字列は先頭から、キーワード reserve、アプリケーション識別子 (空白を含まない文字列)、baseid (予約された範囲の開始セグメントID (16 進数))、および、length (予約されたセグメントID の数) から構成されます。コメント行には、最初の列に # を指定します。このファイルには、空白 (空白の行) があってはなりません。システムに予約されたセグメントID は/usr/include/rsm/rsm_common.h ヘッダーファイルに定義されています。アプリケーションはシステムに予約されたセグメント ID を使用できません。

成功した場合、rsm_get_segmentid_range 関数は 0 を返します。失敗した場合、この関数は次のエラー値のうちの 1 つを返します。

RSMERR_BAD_ADDR	渡されたアドレスが無効です。
RSMERR_BAD_APPID	アプリケーション ID が /etc/rsm/rsm.segmentid ファイルに定義されていません。
RSMERR_BAD_CONF	構成ファイル /etc/rsm/rsm.segmentid が存在しないか、読み取ることができません。構成ファイルの書式が正しくありません。

メモリーセグメント操作

RSM セグメントは、連続する仮想アドレスの範囲にマッピングされた (一般的に) 連続しない物理メモリーページセットを表します。RSM セグメントのエクスポート操作とインポート操作によって、相互接続のシステム間で物理メモリー領域を共有できるようになります。物理メモリーページが存在するノードのプロセスのことをメモリーの「エクスポータ」と呼びます。リモートアクセス用に公開するためにエクスポートされたセグメントは、指定されたノードに固有なセグメント識別子を持ちます。セグメント ID はエクスポータが指定するか、RSMAPI フレームワークが割り当てます。

エクスポートされたメモリーへのアクセスを取得するために、相互接続のノードのプロセスは RSM インポートセグメントを作成します。この RSM インポートセグメントは、ローカルの物理ページではなく、エクスポートされたセグメントと接続しています。相互接続がメモリーマッピングをサポートする場合、インポータはインポートセグメントのメモリーマッピングされたアドレスを使用して、エクスポートされたメモリーを読み書きできます。相互接続がメモリーマッピングをサポートしない場合、インポートしているプロセス (インポータ) はメモリーアクセスプリミティブを使用します。

エクスポート側のメモリーセグメント操作

メモリーセグメントをエクスポートするとき、アプリケーションはまず、通常のオペレーティングシステムインタフェース (System V Shared Memory Interface、`mmap`、または `valloc` など) を使用して、自分の仮想アドレス空間にメモリーを割り当てます。メモリーを割り当てた後、アプリケーションは RSMAPI ライブラリインタフェースを呼び出して、セグメントを作成して、ラベルを付けます。セグメントにラベルを付けた後、RSMAPI ライブラリインタフェースは割り当てた仮想アドレスの範囲に物理ページをバインドします。物理ページをバインドした後、RSMAPI ライブラリインタフェースはセグメントを公開して、インポートしているプロセス (インポータ) がアクセスできるようにします。

注 – `mmap` を使用して仮想アドレス空間を取得した場合、マッピングは `MAP_PRIVATE` である必要があります。

エクスポート側のメモリーセグメント操作には、次のような操作が含まれます。

- メモリーセグメントの作成および破壊
- メモリーセグメントの公開および公開解除
- メモリーセグメント用のバッキングストアの再バインド

メモリーセグメントの作成と破壊

rsm_memseg_export_create を使用して新しいメモリーセグメントを確立すると、セグメントを作成するときに物理メモリーを関連付けることができます。この操作は、エクスポート側のメモリーセグメントハンドルを新しいメモリーセグメントに戻します。セグメントは作成するプロセスが動作している間、または、rsm_memseg_export_destroy を使用して破壊するまで存在します。

注 – インポート側が切断する前に破壊操作が行われた場合、切断が強制的に行われます。

セグメントの作成

int **rsm_memseg_export_create**(*rsmapi_controller_handle_t controller*, *rsm_memseg_export_handle_t *memseg*, *void *vaddr*, *size_t size*, *uint_t flags*);

この関数はセグメントハンドルを作成します。セグメントハンドルを作成した後、この関数はセグメントハンドルを指定された仮想アドレス範囲 [vaddr..vaddr+size] にバインドします。この範囲は有効であり、コントローラの alignment プロパティ上に整列している必要があります。flags 引数はビットマスクで、次の操作を有効にします。

- セグメント上のバインド解除
- セグメント上の再バインド
- RSM_ALLOW_REBIND の flags への引き渡し
- ロック操作のサポート
- RSM_LOCK_OPS の flags への引き渡し

注 – RSM_LOCK_OPS フラグは RSMAPI の初期リリースには含まれません。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_CTLR_HNDL
  コントローラハンドルが無効です。

RSMERR_CTLR_NOT_PRESENT
  コントローラが存在しません。

RSMERR_BAD_SEG_HNDL
  セグメントハンドルが無効です。

RSMERR_BAD_LENGTH
  コントローラの長さが 0 あるいは、制限を超えています。

RSMERR_BAD_ADDR
  アドレスが無効です。

RSMERR_PERM_DENIED
アクセス権がありません。

RSMERR_INSUFFICIENT_MEM
メモリーが不足しています。

RSMERR_INSUFFICIENT_RESOURCES
リソースが不足しています。

RSMERR_BAD_MEM_ALIGNMENT
アドレスがページ境界に整列されていません。

RSMERR_INTERRUPTED
シグナルによって操作が割り込まれました。

セグメントの破壊

int **rsm_memseg_export_destroy**(*rsm_memseg_export_handle_t memseg*);

この関数はセグメントとその空きリソースの割り当てを解除します。インポートしているプロセス (インポータ) はすべて強制的に切断されます。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　セグメントハンドルが無効です。

RSMERR_POLLFD_IN_USE　pollfd は使用中です。

メモリーセグメントの発行、再発行、および発行解除

発行操作によって、相互接続上にあるほかのノードがメモリーセグメントをインポートできます。エクスポートセグメントは複数の相互接続アダプタ上で発行できます。

セグメント ID は承認された範囲または 0 を指定した場合、有効なセグメント ID が RSMAPI フレームワークによって生成され、返されます。

セグメントアクセス制御リストはノード ID とアクセス権のペアから構成されます。リストでは、指定したノード ID ごとに、Solaris のファイルアクセス権とともに所有者、グループ、およびその他のユーザーの 3 つの 8 進数によって関連する読み取り権と書き込み権が示されます。アクセス制御リストでは、各 8 進数は次の値を持ちます。

2　書き込みアクセス

4　読み取り専用アクセス

6　読み取りおよび書き込みアクセス

たとえば、0624 というアクセス権は次のことを意味します。

- エクスポータと同じ uid を持つインポータは、読み取りと書き込み両方のアクセス権を持つ
- エクスポータと同じ gid を持つインポータは、書き込みアクセス権だけを持つ

■ その他すべてのインポータは、読み取り専用アクセス権だけを持つ

アクセス制御リストが提供される場合、リストに含まれないノードはセグメントをインポートできません。ただし、アクセス制御リストが NULL の場合は、すべてのノードがセグメントをインポートできます。すべてのノードのアクセス権は、エクスポートするプロセス (エクスポータ) の所有者、グループ、およびその他のファイル作成権と同じになります。

注 – ノードアプリケーションはセグメント識別子の割り当てを管理し、エクスポートするノード上で一意性を保証する義務があります。

セグメントの公開

int **rsm_memseg_export_publish**(*rsm_memseg_export_handle_t memseg*, *rsm_memseg_id_t *segment_id*, *rsmapi_access_entry_t access_list[]*, *uint_t access_list_length*);

```
typedef struct {
rsm_node_id_t    ae_node;    /* 資源へのアクセスを許可されたリモートノード ID */
rsm_permission_t ae_permissions;    /* 許可されたアクセスモード */
}rsmapi_access_entry_t;.
```

戻り値**:** 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL
セグメントハンドルが無効です。

RSMERR_SEG_ALREADY_PUBLISHED
セグメントはすでに公開されています。

RSMERR_BAD_ACL
アクセス制御リストが無効です。

RSMERR_BAD_SEGID
セグメント ID が無効です。

RSMERR_SEGID_IN_USE
セグメント ID は使用中です。

RSMERR_RESERVED_SEGID
セグメント ID は予約されています。

RSMERR_NOT_CREATOR
セグメントの作成者ではありません。

RSMERR_BAD_ADDR
アドレスが不正です。

RSMERR_INSUFFICIENT_MEM
メモリーが不足しています。

RSMERR_INSUFFICIENT_RESOURCES
リソースが不足しています。

認可されたセグメント **ID** の範囲**:**

```
#define RSM_DRIVER_PRIVATE_ID_BASE
  0

#define RSM_DRIVER_PRIVATE_ID_END
  0x0FFFFF

#define RSM_CLUSTER_TRANSPORT_ID_BASE
  0x100000

#define RSM_CLUSTER_TRANSPORT_ID_END
  0x1FFFFF

#define RSM_RSMLIB_ID_BASE
  0x200000

#define RSM_RSMLIB_ID_END
  0x2FFFFF

#define RSM_DLPI_ID_BASE
  0x300000

#define RSM_DLPI_ID_END
  0x3FFFFF

#define RSM_HPC_ID_BASE
  0x400000

#define RSM_HPC_ID_END
  0x4FFFFF
```

次に示す範囲は、公開値が 0 の場合、システムによる割り当て用に予約されています。

```
#define RSM_USER_APP_ID_BASE    0x80000000

#define RSM_USER_APP_ID_END     0xFFFFFFF
```

セグメントの再発行

int **rsm_memseg_export_republish**(*rsm_memseg_export_handle_t memseg*, *rsmapi_access_entry_t access_list[]*, *uint_t access_list_length*);

この関数は、ノードのアクセス (制御) リストとセグメントのアクセスモードを新たに確立します。これらの変更は将来のインポート呼び出しだけに影響し、すでに許可されているインポート要求は取り消しません。

戻り値**:** 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL
セグメントハンドルが無効です。

RSMERR_SEG_NOT_PUBLISHED
  セグメントが公開されていません。

RSMERR_BAD_ACL
  アクセス制御リストが無効です。

RSMERR_NOT_CREATOR
  セグメントの作成者ではありません。

RSMERR_INSUFFICIENT_MEM
  メモリーが不足しています。

RSMERR_INSUFFICIENT_RESOURCES
  リソースが不足しています。

RSMERR_INTERRUPTED
  シグナルによって操作が割り込まれました。

セグメントの公開解除

int **rsm_memseg_export_unpublish**(*rsm_memseg_export_handle_t memseg*);

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL	セグメントハンドルが無効です。
RSMERR_SEG_NOT_PUBLISHED	セグメントが公開されていません。
RSMERR_NOT_CREATOR	セグメントの作成者ではありません。
RSMERR_INTERRUPTED	シグナルによって操作が割り込まれました。

メモリーセグメントの再バインド

再バインド操作は、エクスポートセグメントの現在のバッキングストアを解放します。現在のバッキングストアを解放した後、再バインド操作は、新しいバッキングストアを割り当てます。まず始めにアプリケーションは、セグメント用の新しい仮想メモリー割り当てを取得する必要があります。この操作はセグメントのインポータに透過的です。

注 – アプリケーションは、再バインド操作が完了するまで、セグメントデータへアクセスしてはいけません。再バインド中にセグメントからデータを取得しようとしてもシステムエラーにはなりませんが、このような操作の結果は定義されていません。

セグメントの再バインド

int **rsm_memseg_export_rebind**(*rsm_memseg_export_handle_t memseg*, *void *vaddr*, *offset_t off*, *size_t size*);

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL
セグメントハンドルが無効です。

RSMERR_BAD_LENGTH
長さが無効です。

RSMERR_BAD_ADDR
アドレスが無効です。

RSMERR_REBIND_NOT_ALLOWED
再バインドは許可されていません。

RSMERR_NOT_CREATOR
セグメントの作成者ではありません。

RSMERR_PERM_DENIED
アクセス権がありません。

RSMERR_INSUFFICIENT_MEM
メモリーが不足しています。

RSMERR_INSUFFICIENT_RESOURCES
リソースが不足しています。

RSMERR_INTERRUPTED
シグナルによって操作が割り込まれました。

インポート側のメモリーセグメント操作

インポート側の操作には、次の操作が含まれます。

- メモリーセグメントの接続および切断
- インポートされたメモリーセグメントへのアクセス
- バリア操作を使用したデータアクセス操作の順番の決定、およびアクセスエラーの検出

接続操作は、RSM インポートセグメントを作成して、エクスポートされたセグメントとの論理的な接続を形成するときに使用します。

インポートされたセグメントメモリーへのアクセスは、次の 3 つのインタフェースカテゴリによって実現されます。

- セグメントアクセス
- データ転送
- セグメントメモリーマッピング

メモリーセグメントの接続と切断

セグメントへの接続

int **rsm_memseg_import_connect**(*rsmapi_controller_handle_t controller*, *rsm_node_id_t node_id*, *rsm_memseg_id_t segment_id*, *rsm_permission_t perm*, *rsm_memseg_import_handle_t *im_memseg*);

この関数は、指定されたアクセス権 *perm* を使用してリモートノード *node_id* 上にあるセグメント *segment_id* に接続します。セグメントに接続した後、この関数はセグメントハンドルを返します。

引数 *perm* は、当該接続のインポータによって要求されるアクセスモードを指定します。接続を確立するとき、エクスポータが指定したアクセス権とインポータが使用するアクセスモード、ユーザー要求されるアクセスモードが無効な場合、接続要求は拒否されます。なお、*perm* 引数は次の 8 進数値に制限されます。

0400　読み取りモード

0200　書き込みモード

0600　読み取りおよび書き込みモード

指定されたコントローラは、セグメントのエクスポートに使用されるコントローラと物理的に接続されている必要があります。

戻り値**:** 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_CTLR_HNDL
　コントローラハンドルが無効です。

RSMERR_CTLR_NOT_PRESENT
　コントローラが存在しません。

RSMERR_BAD_SEG_HNDL
　セグメントハンドルが無効です。

RSMERR_PERM_DENIED
　アクセス権がありません。

RSMERR_SEG_NOT_PUBLISHED_TO_NODE
　セグメントがノードに公開されていません。

RSMERR_SEG_NOT_PUBLISHED
　セグメントが公開されていません。

RSMERR_REMOTE_NODE_UNREACHABLE
　リモートノードに到達できません。

RSMERR_INTERRUPTED
　接続が割り込まれました。

RSMERR_INSUFFICIENT_MEM
　メモリーが不足しています。

RSMERR_INSUFFICIENT_RESOURCES
　リソースが不足しています。

RSMERR_BAD_ADDR
　アドレスが不正です。

セグメントからの切断

int **rsm_memseg_import_disconnect**(*rsm_memseg_import_handle_t im_memseg*);

この関数はセグメントを切断します。セグメントを切断した後、この関数はセグメントのリソースを解放します。切断されたセグメントへの既存のマッピングはすべて削除されます。ハンドル im_memseg は解放されます。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL	セグメントハンドルが無効です。
RSMERR_SEG_STILL_MAPPED	セグメントがマッピングされたままになっています。
RSMERR_POLLFD_IN_USE	pollfd は使用中です。

メモリーアクセスプリミティブ

次のインタフェースは、8 ビットから64 ビットまでのデータを転送するためのメカニズムを提供します。get インタフェースは、プロセスがメモリー上の連続するデータから読みとるべき、与えられたサイズのデータ項目の数を示すリピートカウント (*rep_cnt*) を使用します。メモリー上の連続するデータは、インポートされたセグメントのオフセット (*offset*) から始まります。データは *datap* から始まる連続する場所に書き込まれます。put インタフェースは、リピートカウント (*rep_cnt*) を使用して、プロセスが読み取るべきデータ項目数を指定します。連続する場所は、*datap* から始まります。データは次に、インポートされたセグメントの *offset* から始まる連続する場所に書き込まれます。

これらのインタフェースはまた、読み取り元と書き込み先のエンディアン特性に互換性がない場合にバイトを交換するメカニズムも提供します。

関数のプロトタイプ:

int **rsm_memseg_import_get8**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint8_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_get16**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint16_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_get32**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint32_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_get64**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint64_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_put8**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint8_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_put16**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint16_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_put32**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint32_t *datap*, *ulong_t rep_cnt*);

int **rsm_memseg_import_put64**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset*, *uint64_t *datap*, *ulong_t rep_cnt*);

次のインタフェースは、セグメントアクセス操作がサポートするデータよりも大きなデータを転送するときに使用します。

セグメントの書き込み

int **rsm_memseg_import_put**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset* , *void *src_addr*, *size_t length*);

この関数は、*src_addr* と *length* で指定されたローカルメモリーから、対応するインポートされたセグメントのハンドルとオフセットで指定された場所にデータを書き込みます。

セグメントの読み取り

int **rsm_memseg_import_get**(*rsm_memseg_import_handle_t im_memseg*, *off_t offset* , *void *dst_addr*, *size_t length*);

この関数は rsm_memseg_import_put() と似ていますが、データはインポートされたセグメントから *dest_vec* で定義されたローカル領域に移動します。

put ルーチンと get ルーチンは、引数 *offset* で指定したバイトオフセット位置から、指定された量のデータを書き込みまたは読み込みます。これらのルーチンはセグメントのベースから開始します。オフセットは適切な境界に整列している必要があります。たとえば、rsm_memseg_import_get64() の場合、*offset* と *datap* はダブルワード境界に整列している必要がありますが、rsm_memseg_import_put32() の場合、*offset* はワード境界に整列している必要があります。

デフォルトでは、セグメントのバリアモード属性は暗黙的 (implicit) です。暗黙的なバリアモードは、操作から戻ってきたときにはデータ転送が完了または失敗していると呼び出し元が仮定していることを意味します。デフォルトのバリアモードは暗黙的であるため、アプリケーションはバリアを初期化する必要があります。デフォルトのバリアモードを使用するとき、put ルーチンまたは get ルーチンを呼び出す前に、アプリケーションは rsm_memseg_import_init_barrier() 関数を使用してバリアを初期化します。明示的な操作モードを使用するには、呼び出し元はバリア操作を使用して転送を強制的に完了させる必要があります。転送を強制的に完了させた後、結果としてエラーが発生したかどうかを判断する必要があります。

注 – オフセットを rsm_memseg_import_map() ルーチンに渡すことによって、インポートセグメントは部分的にマッピングできます。インポートセグメントを部分的にマッピングする場合、put ルーチンまたは get ルーチンの *offset* 引数はセグメントのベースからです。ユーザーは、正しいバイトオフセットが put ルーチンまたは get ルーチンに渡されていることを確認する必要があります。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL
セグメントハンドルが無効です。

RSMERR_BAD_ADDR
アドレスが不正です。

RSMERR_BAD_MEM_ALIGNMENT
メモリー整列が無効です。

RSMERR_BAD_OFFSET
オフセットが無効です。

RSMERR_BAD_LENGTH
長さが無効です。

RSMERR_PERM_DENIED
アクセス権がありません。

RSMERR_BARRIER_UNINITIALIZED
バリアが初期化されていません。

RSMERR_BARRIER_FAILURE
入出力完了エラー

RSMERR_CONN_ABORTED
接続が中断されました。

RSMERR_INSUFFICIENT_RESOURCES
リソースが不足しています。

Scatter-Gather アクセス

rsm_memseg_import_putv() と rsm_memseg_import_getv () 関数を使用すると、単一の読み取り元アドレスや単一の書き込み先アドレスではなく、入出力要求のリストを使用できます。

関数のプロトタイプ:

int **rsm_memseg_import_putv**(*rsm_scat_gath_t *sg_io*);

int **rsm_memseg_import_getv**(*rsm_scat_gath_t *sg_io*);

Scatter-Gather リスト (sg_io) の入出力ベクトル構成要素を使用すると、ローカル仮想アドレスまたは local_memory_handles を指定できます。ハンドルはローカルアドレス範囲を繰り返して使用するための効率的な方法です。割り当てられたシステムリソース (ロックダウンされたローカルメモリーなど) はハンドルが解放されるまで保持されます。ハンドルをサポートする関数は rsm_create_localmemory_handle() と rsm_free_localmemory_handle() です。

仮想アドレスやハンドルは、ベクトルに集めて、単一のリモートセグメントに書き込むことができます。この結果はまた、単一のリモートセグメントから読み取って、仮想アドレスまたはハンドルのベクトルに分散できます。

ベクトル全体の入出力は関数が返る前に初期化されます。インポートセグメントのバリアモード属性は、関数が返る前に入出力が完了しているかどうかを判断します。バリアモード属性を implicit (暗黙的) に設定すると、ベクトルに入った順番でデータ転送が完了することが保証されます。リストの各エントリは、暗黙的なバリアの開く操作と閉じる操作によって囲まれます。エラーが検出された場合、ベクトルの入出力は中断され、関数はすぐに返ります。残りのカウントは、入出力が完了または初期化されなかったエントリの数を示します。

putv 操作または getv 操作が正常に完了した場合に通知イベントをターゲットセグメントに送信することを指定できます。通知イベントを送信することを指定するには、rsm_scat_gath_t 構造体の flags エントリに RSM_IMPLICIT_SIGPOST 値を指定します。また、flags エントリに RSM_SIGPOST_NO_ACCUMULATE を指定しておくと、RSM_IMPLICIT_SIGPOST が設定されたときに、この値がシグナルポスト操作に渡されます。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SGIO
Scatter-Gather 構造体ポインタが無効です。

RSMERR_BAD_SEG_HNDL
セグメントハンドルが無効です。

RSMERR_BAD_CTLR_HNDL
コントローラハンドルが無効です。

RSMERR_BAD_ADDR
アドレスが不正です。

RSMERR_BAD_OFFSET
オフセットが無効です。

RSMERR_BAD_LENGTH
長さが無効です。

RSMERR_PERM_DENIED
アクセス権がありません。

RSMERR_BARRIER_FAILURE
入出力完了エラー

RSMERR_CONN_ABORTED
接続が中断されました。

RSMERR_INSUFFICIENT_RESOURCES
リソースが不足しています。

RSMERR_INTERRUPTED
シグナルによって操作が割り込まれました。

ローカルハンドルの取得

int **rsm_create_localmemory_handle**(*rsmapi_controller_handle_t cntrl_handle*, *rsm_localmemory_handle_t *local_handle*, *caddr_t local_vaddr* , *size_t length*);

この関数は、後続の putv または getv への呼び出しの入出力ベクトルで使用するためのローカルハンドルを取得します。ロックダウンの可能性があるので、メモリーがローカルハンドルによってスパンされている場合は特に、可能な限りハンドルを解放して、システムリソースを節約してください。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_CTLR_HNDL	コントローラハンドルが無効です。
RSMERR_BAD_LOCALMEM_HNDL	ローカルメモリーハンドルが無効です。
RSMERR_BAD_LENGTH	長さが無効です。
RSMERR_BAD_ADDR	アドレスが無効です。
RSMERR_INSUFFICIENT_MEM	メモリーが不足しています。

ローカルハンドルの解放

rsm_free_localmemory_handle(*rsmapi_controller_handle_t cntrl_handle*, *rsm_localmemory_handle_t handle*);

この関数は、ローカルハンドルに関連するシステムリソースを解放します。プロセスが終了するときにはプロセスに属するすべてのハンドルが解放されますが、この関数を呼び出すことでシステムリソースを節約できます。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_CTLR_HNDL	コントローラハンドルが無効です。
RSMERR_BAD_LOCALMEM_HNDL	ローカルメモリーハンドルが無効です。

次の例に、プライマリデータ構造体の定義を示します。

例 **2–1** プライマリデータ構造体

```
typedef void *rsm_localmemory_handle_t
typedef struct {
   ulong_t    io_request_count;    number of rsm_iovec_t entries
   ulong_t    io_residual_count;    rsm_iovec_t entries not completed

   in    flags;
   rsm_memseg_import_handle_t        remote_handle; opaque handle for
                                                    import segment
   rsm_iovec_t                            *iovec;          pointer to
                                                    array of io_vec_t
} rsm_scat_gath_t;

typedef struct {
   int    io_type;        HANDLE or VA_IMMEDIATE
   union {
        rsm_localmemory_handle_t    handle;            used with HANDLE
        caddr_t                        virtual_addr;        used with
                                                      VA_IMMEDIATE
    } local;
```

例 2–1 プライマリデータ構造体　(続き)

```
   size_t          local_offset;              offset from handle base vaddr
   size_t          import_segment_offset;    offset from segment base vaddr
   size_t          transfer_length;
} rsm_iovec_t;
```

セグメントのマッピング

マッピング操作は、ネイティブなアーキテクチャの相互接続 (Dolphin-SCI や NewLink など) だけで利用できます。セグメントをマッピングすることによって CPU メモリー操作がそのセグメントにアクセスできるようになるので、メモリーアクセスプリミティブを呼び出すオーバーヘッドを省くことができます。

インポートされたセグメントのマッピング

int **rsm_memseg_import_map**(*rsm_memseg_import_handle_t im_memseg*, *void **address*, *rsm_attribute_t attr*, *rsm_permission_t perm*, *off_t offset*, *size_t length*);

この関数は、インポートされたセグメントを呼び出し元のアドレス空間にマッピングします。属性 RSM_MAP_FIXED が指定されている場合、この関数は ***address* に指定された値にあるセグメントをマッピングします。

```
typedef enum {
RSM_MAP_NONE = 0x0,    /* システムは使用できる仮想アドレスを選択する */
RSM_MAP_FIXED = 0x1,   /* セグメントを指定された仮想アドレスにマッピングする */
} rsm_map_attr_t;
```

戻り値**:** 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL	セグメントハンドルが無効です。
RSMERR_BAD_ADDR	アドレスが無効です。
RSMERR_BAD_LENGTH	長さが無効です。
RSMERR_BAD_OFFSET	オフセットが無効です。
RSMERR_BAD_PERMS	アクセス権がありません。
RSMERR_SEG_ALREADY_MAPPED	セグメントはすでにマッピングされています。
RSMERR_SEG_NOT_CONNECTED	セグメントは接続されていません。
RSMERR_CONN_ABORTED	接続が中断されました。
RSMERR_MAP_FAILED	マッピング中にエラーが発生しました。
RSMERR_BAD_MEM_ALIGNMENT	アドレスがページ境界に整列されていません。

セグメントのマッピング解除

int **rsm_memseg_import_unmap**(*rsm_memseg_import_handle_t im_memseg*);

この関数は、ユーザーの仮想アドレス空間からインポートされたセグメントをマッピング解除します。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　セグメントハンドルが無効です。

バリア操作

バリア操作は、書き込みアクセス順番メモリーモデルに関する問題を解決するときに使用します。バリア操作は、リモートメモリーアクセスエラーを検出することもできます。

バリアメカニズムには、次のような操作が含まれます。

- バリア初期化
- バリアを開く
- バリアを閉じる
- 書き込みの順番を決定する

開く操作と閉じる操作は、エラーの検出と順番の決定を行う期間 (span-of-time) を定義します。初期化操作は、インポートされたセグメントごとにバリアの作成とバリアのタイプの指定を可能にします。現在サポートされるバリアのタイプだけが、セグメントごとに期間 (span-of-time) を持っています。タイプ引数には RSM_BAR_DEFAULT を使用してください。

閉じる操作を正常に実行することによって、バリアを開いてから閉じるまでの間に発生するアクセス操作が正常に完了することが保証されます。バリアを開いた後、個々のデータアクセス操作 (読み取りと書き込みの両方) が失敗しても、バリアを閉じるまでは報告されません。

バリアの有効範囲内で書き込みの順番を決定するには、明示的なバリア順番決定操作を使用します。バリア順番決定操作の前に発行された書き込み操作は、バリア順番決定操作後に発行された操作よりも前に完了します。あるバリアの有効範囲内の書き込み操作の順番は別のバリアの有効範囲を基準にして決定されます。

バリアの初期化

int **rsm_memseg_import_init_barrier**
(*rsm_memseg_import_handle_t im_memseg* , *rsm_barrier_type_t type* ,
*rsmapi_barrier_t *barrier*) ;

注 – 現在のところ、サポートされるタイプは RSM_BAR_DEFAULT だけです。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　セグメントハンドルが無効です。

RSMERR_BAD_BARRIER_PTR　バリアポインタが無効です。

RSMERR_INSUFFICIENT_MEM　メモリーが不足しています。

バリアを開く

int **rsm_memseg_import_open_barrier**（*rsmapi_barrier_t *barrier*）;

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　セグメントハンドルが無効です。

RSMERR_BAD_BARRIER_PTR　バリアポインタが無効です。

バリアを閉じる

int **rsm_memseg_import_close_barrier**（*rsmapi_barrier_t *barrier*）;

この関数はバリアを閉じて、すべてのストアバッファーをフラッシュします。この関数は、rsm_memseg_import_close_barrier() の呼び出しが失敗した場合、最後の rsm_memseg_import_open_barrier 呼び出しまで、呼び出し元プロセスがすべてのリモートメモリー操作を再試行することを前提にして呼び出されます。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　セグメントハンドルが無効です。

RSMERR_BAD_BARRIER_PTR　バリアポインタが無効です。

RSMERR_BARRIER_UNINITIALIZED　バリアが初期化されていません。

RSMERR_BARRIER_NOT_OPENED　バリアが開かれていません。

RSMERR_BARRIER_FAILURE　メモリーアクセスエラー

RSMERR_CONN_ABORTED　接続が中断されました。

バリアの順番決定

int **rsm_memseg_import_order_barrier**（*rsmapi_barrier_t *barrier*）;

この関数は、すべてのストアバッファーをフラッシュします。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　セグメントハンドルが無効です。

RSMERR_BAD_BARRIER_PTR　バリアポインタが無効です。

RSMERR_BARRIER_UNINITIALIZED　バリアが初期化されていません。

RSMERR_BARRIER_NOT_OPENED　バリアが開かれていません。

RSMERR_BARRIER_FAILURE　メモリーアクセスエラー

RSMERR_CONN_ABORTED　接続が中断されました。

バリアの破壊

int **rsm_memseg_import_destroy_barrier**(*rsmapi_barrier_t *barrier*);

この関数は、すべてのバリアリソースの割り当てを解除します。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　　セグメントハンドルが無効です。

RSMERR_BAD_BARRIER_PTR　　バリアポインタが無効です。

モードの設定

int **rsm_memseg_import_set_mode** (*rsm_memseg_import_handle_t im_memseg*, *rsm_barrier_mode_t mode*);

この関数は、put ルーチンで利用できるオプションの明示的なバリアの有効範囲決定をサポートします。有効なバリアモードは、RSM_BARRIER_MODE_EXPLICIT と RSM_BARRIER_MODE_IMPLICIT の 2 つです。バリアモードのデフォルト値は RSM_BARRIER_MODE_IMPLICIT です。暗黙モードでは、put 操作ごとに暗黙的なバリアの開く操作と閉じる操作が適用されます。バリアモードを RSM_BARRIER_MODE_EXPLICIT に設定する前は、rsm_memseg_import_init_barrier ルーチンを使用して、インポートされたセグメント im_memseg 用のバリアを初期化する必要があります。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　　セグメントハンドルが無効です。

モードの取得

int **rsm_memseg_import_get_mode** (*rsm_memseg_import_handle_t im_memseg* , *rsm_barrier_mode_t *mode*);

この関数は、put ルーチンにおける現在のバリアの有効範囲決定のモード値を取得します。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　　セグメントハンドルが無効です。

イベント操作

イベント操作によって、プロセスはメモリーアクセスイベントと同期をとることができます。rsm_intr_signal_wait() 関数を使用できない場合、 rsm_memseg_get_pollfd() でポーリング記述子を取得し、poll システムコールを使用することによって、プロセスはイベント待機を多重送信できます。

注 – `rsm_intr_signal_post()` 操作と `rsm_intr_signal_wait()` 操作を使用すると、カーネルへの `ioctl` 呼び出しを処理する必要が生まれます。

シグナルの送信

`int` **`rsm_intr_signal_post`** (*void *memseg, uint_t flags*);

ボイドポインタ `*memseg` を使用すると、インポートセグメントハンドルまたはエクスポートセグメントハンドルのどちらでもタイプキャスト (型変換) できます。`*memseg` がインポートセグメントハンドルを参照している場合、この関数はエクスポートしているプロセス (エクスポータ) にシグナルを送信します。`*memseg` がエクスポートセグメントハンドルを参照している場合、この関数はそのセグメントのすべてのインポータにシグナルを送信します。`flags` 引数に `RSM_SIGPOST_NO_ACCUMULATE` を設定すると、あるイベントがすでにターゲットセグメントに対して保留中である場合、当該イベントを破棄します。

戻り値: 成功した場合、`0` を返します。そうでない場合、エラー値を返します。

`RSMERR_BAD_SEG_HNDL`
セグメントハンドルが無効です。

`RSMERR_REMOTE_NODE_UNREACHABLE`
リモートノードに到達できません。

シグナルの待機

`int` **`rsm_intr_signal_wait`** (*void * memseg, int timeout*);

ボイドポインタ `*memseg` を使用すると、インポートセグメントハンドルまたはエクスポートセグメントハンドルのどちらでもタイプキャスト (型変換) できます。プロセスは *timeout* ミリ秒まで、あるいは、イベントが発生するまでブロックされます。値が -1 の場合、プロセスはイベントが発生するまで、あるいは、割り込みが発生するまでブロックされます。

戻り値: 成功した場合、`0` を返します。そうでない場合、エラー値を返します。

`RSMERR_BAD_SEG_HNDL`	セグメントハンドルが無効です。
`RSMERR_TIMEOUT`	タイマーが満了しました。
`RSMERR_INTERRUPTED`	待機中に割り込みが発生しました。

`pollfd` の取得

`int` **`rsm_memseg_get_pollfd`** (*void *memseg, struct pollfd *pollfd*);

この関数は、指定された `pollfd` 構造体を、指定されたセグメントの記述子と `rsm_intr_signal_post()` で生成された単一固定イベントで初期化します。`pollfd` 構造体を `poll` システムコールで使用すると、`rsm_intr_signal_post` によってシ

グナル送信されるイベントを待機します。メモリーセグメントがまだ公開されていない場合、poll システムコールは有効なpollfd を返しません。呼び出しが成功するたびに、指定されたセグメントの pollfd 参照カウントがインクリメントします。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　セグメントハンドルが無効です。

pollfd の解放

int **rsm_memseg_release_pollfd**(*oid *memseg*);

この呼び出しは、指定されたセグメントの pollfd 参照カウントをデクリメントします。参照カウントが 0 以外の場合、セグメントを公開解除、破壊、またはマッピング解除する操作は失敗します。

戻り値: 成功した場合、0 を返します。そうでない場合、エラー値を返します。

RSMERR_BAD_SEG_HNDL　セグメントハンドルが無効です。

RSMAPI を使用するときの一般的な注意点

この節では、共有メモリー操作のエクスポート側とインポート側における一般的な注意点について説明します。この節ではまた、セグメント、ファイル記述子、および RSM 構成可能パラメータに関する一般的な情報についても説明します。

セグメントの割り当てとファイル記述子の使用

システムはエクスポート操作またはインポート操作ごとにファイル記述子を割り当てますが、メモリーをインポートまたはエクスポートしているアプリケーションはこの記述子にアクセスできません。プロセスごとのファイル記述子割り当てのデフォルトの制限は 256 です。インポートまたはエクスポートしているアプリケーションは割り当ての制限を適切に調節する必要があります。アプリケーションがファイル記述子の制限値を 256 より大 きく設定した場合、エクスポートセグメントとインポートセグメントに割り当てられるファイル記述子は 256 から始まります。このようなファイル記述子の値が選択されるのは、アプリケーションが通常のファイル記述子を割り当てるのを妨害しないようにするためです。この動作によって、256 より小さなファイル記述子を処理できない 32 ビットアプリケーションが特定の libc 関数を使用できるようになります。

エクスポート側の注意点

アプリケーションは、再バインド操作が完了するまで、セグメントデータにアクセスしないようにする必要があります。再バインド中にセグメントからデータを取得しようとしてもシステムエラーにはなりませんが、このような操作の結果は定義されていません。仮想アドレス空間はすでにマッピングされており、有効である必要があります。

インポート側の注意点

インポートセグメント用に指定されたコントローラは、セグメントのエクスポートに使用されるコントローラと物理的に接続されている必要があります。

RSM 構成可能パラメータ

SUNWrsm ソフトウェアパッケージには rsm.conf ファイルがあります。このファイルは /usr/kernel/drv にあります。このファイルは RSM 用の構成ファイルです。rsm.conf ファイルを使用すると、特定の構成可能な RSM プロパティの値を指定できます。現在 rsm.conf ファイルに定義されている構成可能なパラメータには max-exported-memory と enable-dynamic-reconfiguration があります。

max-exported-memory
: エクスポート可能なメモリー量の上限を指定します。この上限は、利用可能なメモリーの合計に対するパー センテージで表現されます。このプロパティの値が 0 の場合、エクスポート可能なメ モリーに上限がないことを示します。

enable-dynamic-reconfiguration
: 動的再構成が有効であるかどうかを示します。このプロパティの値が 0 の場合、動的再構成が無効であることを示します。1 の場合、動的再構成が有効であることを示します。このプロパティのデフォルトの値は 1 です。

RSMAPI の使用例

この節では、簡単なプログラムを使用して RSMAPI の使用例を示します。このプログラムはエクスポータノードとインポータノードの 2 つのノード上で動作します。エクスポータノードはメモリーセグメントを作成および公開し、メッセージがセグメントに書き込まれるまで待機します。インポータノードはエクスポートされたセグメントに接続し、メッセージを書き込み、次に、エクスポータにシグナルを送信します。

```
/*
 * Copyright (c) 1998 by Sun Microsystems, Inc.
 * All rights reserved.
```

```
 */
#include <stdio.h>
#include <rsm/rsmpai.h>
#include <errno.h>


/*
        このプログラムを実行するには次の手順を行う:

        1 つ目のノード (ノード ID = 1):
                rsmtest -e -n 2

        2 つ目のノード (ノード ID = 2):
                rsmtest -i -n 1

       プログラムはコンソールでインポータにメッセージをプロンプト
       表示する。任意のメッセージを入力してリターンキーを入力する。
       エクスポートコンソールにメッセージが表示される。
*/

typedef struct {
        char    out;
        char    in;
        char    data[1];
}msg_t;

#define SEG_ID 0x400000
#define EXPORT  0
#define IMPORT  1

#define BUFSIZE (1024 * 8)
#define DEFAULT_SEGSZ BUFSIZE

#define RSM_PERM_READ                       0400
#define RSM_PERM_WRITE                      0200
#define RSM_PERM_RDWR                       (RSM_PERM_READ|RSM_PERM_WRITE)
#define
RSM_ACCESS_TRUSTED


0666

rsm_topology_t *tp;

int iterations = 10;

int mode = EXPORT;
int test = 0;

char *buf;
int buflen = BUFSIZE;
int offset = 0;
volatile char *iva;

int status;
```

```
rsm_memseg_id_t segid;
rsmapi_controller_handle_t ctrl;
rsmapi_controller_attr_t attr;
rsm_memseg_export_handle_t seg;
rsm_memseg_import_handle_t imseg;
rsm_access_entry_t list[2];
rsm_node_id_t dest;

extern void *valloc(size_t);
extern void exit();
extern void sleep();
extern int atoi(const char *);

/* 次の関数はセグメントをエクスポートし、それを発行する
 */
static int
export()
{
        int i;

        /* メモリーを割り当て、クリアする */
        buf = (char *)valloc(buflen);


if (!buf) {


(void) fprintf(stderr, "Unable to allocate memory\n");


exit (1);

}
        for (i = 0; i < buflen; i++)
                buf[i] = 0;


/* エクスポートメモリーセグメントを作成する */
        status = rsm_memseg_export_create(ctrl, &seg, (void *)buf, buflen);
        if (status != 0) {
                (void) fprintf(stderr,
                      "unable to create an exported segment %d\n", status);
                exit(1);
        }

        /* ノード 1 とノード 2 に発行するアクロスリストを設定する */
        list[0].ae_node = tp->topology_hdr.local_nodeid ;

/* 読み取り権と書き込み権を許可する */
        list[0].ae_permission = RSM_ACCESS_TRUSTED;
        list[1].ae_node = tp->topology_hdr.local_nodeid + 1;

/* 読み取り権とアクセス権を許可する */
        list[1].ae_permission = RSM_ACCESS_TRUSTED;
```

```
/* 作成されたエクスポートセグメントを発行する */
        status = rsm_memseg_export_publish(seg, &segid, list, 0);
        if (status != 0) {
              (void) fprintf(stderr, "unable to pub segment %d\n", status);
              exit(1);
        }
        return (0);
}

/* 次の関数はエクスポートされたメモリーセグメントへの接続に使用される */
static void
import()
{
        /* ローカルの仮想アドレスを介してアクセスするために
         * エクスポートされたセグメントに接続してマッピングを設定する
         */
again:
        status = rsm_memseg_import_connect(ctrl, dest, segid, RSM_PERM_RDWR,
            &imseg);
        if (status != 0) {
                (void) fprintf(stderr,
                                "unable to conect to segment %x err %x\n",
                                segid, status);
                sleep(1);
                goto again;
        }

        iva = NULL;
        status = rsm_memseg_import_map(imseg, (void **)&iva,
                                  RSM_MAP_NONE, RSM_PERM_RDWR, 0, buflen);
        if (status != 0) {
             (void) fprintf(stderr, "unable to mmap segment %d\n", status);
             exit(1);
        }
}

/* エクスポートセグメントの発行を取り消して破壊する */
static void
export_close()
{
again:
        status = rsm_memseg_export_unpublish(seg);
        if (status != 0) {
                (void) fprintf(stderr,
                        "unable to create an unpub segment %d\n", status);
                sleep(10);
                goto again;
        }

        status = rsm_memseg_export_destroy(seg);
        if (status != 0) {
                (void) fprintf(stderr, "unable to destroy segment %d\n",
                                status);
                exit(1);
```

```
        }
}

/* 仮想アドレスのマッピングを取り消してセグメントの接続を解除する */
static void
import_close()
{
        status = rsm_memseg_import_unmap(imseg);
        if (status != 0) {
            (void) fprintf(stderr, "unable to unmap segment %d\n", status);


                exit(1);
        }

        status = rsm_memseg_import_disconnect(imseg);
        if (status != 0) {
                (void) fprintf(stderr,
                        "unable to disconnect segment %d\n", status);
                exit(1);
        }
}


static void
test0()
{
        volatile msg_t  *mbuf;
        /* 報告エラーへのバリア */
        rsmapi_barrier_t bar;
        int i;

        if (mode == EXPORT) {
                (void) export();
                mbuf = (msg_t *)(buf + offset);
                mbuf->in = mbuf->out = 0;
        } else {
             import();
             mbuf = (msg_t *)(iva + offset);
             rsm_memseg_import_init_barrier(imseg, RSM_BARRIER_NODE, &bar);
        }

        (void) printf("Mbuf is %x\n", (uint_t)mbuf);
        while (iterations--> 0) {


int e;

                switch (mode) {
                case EXPORT:
                    while (mbuf->out == mbuf->in) {
                            (void) rsm_intr_signal_wait(seg, 1000);
                    }
                    (void) printf("msg [0x%x %d %d] ",
                            (uint_t)mbuf, (int)mbuf->out, mbuf->in);
```

```
                        for (i = 0; mbuf->data[i] != '\0' && i < buflen; i++) {
                                (void) putchar(mbuf->data[i]);
                                mbuf->data[i] = '?';
                        }
                        (void) putchar('\n');

                        mbuf->out++;
                        break;
                    case IMPORT:
                        (void) printf("Enter msg [0x%x %d]: ",
                                        (uint_t)mbuf, mbuf->out, mbuf->in);
retry:
                        e = rsm_memseg_import_open_barrier(&bar);
                        if (e != 0) {
                            (void) printf("Barrier open failed %x\n", e);
                            exit(1);
                        }
                        for (i = 0; (mbuf->data[i] = getchar()) != '\n'; i++)
                                ;
                        mbuf->data[i] = '\0';
                        rsm_memseg_import_order_barrier(&bar);
                        mbuf->in++;

                        e = rsm_memseg_import_close_barrier(&bar);
                        if (e != 0) {
                            (void) printf("Barrier close failed, %d\n", e);
                            goto retry;
                        }



(void)rsm_intr_signal_post(imseg);
                        break;
                }
        }

        if (mode == IMPORT) {
                import_close();
        } else {
                export_close();
        }
}


void
main(int argc, char *argv[])
{
        int unit = 0;
        char *device = "sci0";
        int i;

        segid = SEG_ID;
        buflen = DEFAULT_SEGSZ;
        while ((i = getopt(argc, argv, "OCGeid:b:sl:n:k:t:c:u:v")) != -1) {
```

```
                switch (i) {
                case 'e':
                        mode = EXPORT;
                        break;
                case 'i':
                        mode = IMPORT;
                        break;
                case 'n':
                        dest = atoi(optarg);
                        if ((int)dest < 0) dest = 0;
                        break;
                default:


  (void) fprintf(stderr, "Usage: %s -ei  -n dest\n",




 argv[0]);



  exit(1);
                }
        }

        status = rsm_get_controller(device, &ctrl);
        if (status != 0) {
                (void) fprintf(stderr, "Unable to get controller\n");
                exit(1);
        }

        status = rsm_get_controller_attr(ctrl, &attr);


status = rsm_get_interconnect_topology(&tp);

if (status != 0) {


(void) fprintf(stderr, "Unable to get topology\n");


exit(1);

} else {

    (void) printf("Local node id = %d\n",



  tp->topology_hdr.local_nodeid);
```

```
}

if (dest == 0) {

  dest = tp->topology_hdr.local_nodeid;

(void) printf("Dest is adjusted to %d\n", dest);

}

        switch (test) {
        case 0:
                test0();
                break;
        default:
                (void) printf("No test executed\n");
                break;
        }
}
```

第 3 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とスケジューリングの変更方法について説明します。

- 55 ページの「スケジューラの概要」では、スケジューラとタイムシェアリングスケジューリングクラスの概要について説明します。その他のスケジューリングクラスについても簡単に説明します。
- 60 ページの「コマンドとインタフェース」では、スケジューリングを変更するためのコマンドとインタフェースについて説明します。
- 63 ページの「その他のインタフェースとの関係」では、スケジューリングを変更したときにカーネルプロセスや特定のインタフェースに与える影響について説明します。
- 64 ページの「性能」では、スケジューリングのコマンドやインタフェースを使用するときに考慮すべき性能の問題について説明します。

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開発者を対象としています。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。

スケジューラの概要

生成されたプロセスには 1 つの軽量プロセス (LWP) がシステムによって割り当てられます。プロセスがマルチスレッド化されている場合、複数の LWP がそのプロセスに割り当てられる可能性もあります。LWP とは、UNIX システムスケジューラによってスケジューリングされ、プロセスをいつ実行するかを決定するオブジェクトのことです。スケジューラは、構成パラメータ、プロセスの動作、およびユーザーの要求に基

づいてプロセスの優先順位を管理します。スケジューラはこれらの優先順位を使用して、次に実行するプロセスを判断します。優先順位には、リアルタイム、システム、対話型(IA)、固定優先順位(FX)、公平共有(FSS)、およびタイムシェアリング(TS) の 6 つのクラスがあります。

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を調節します。この方式はまた、プロセスの優先順位を動的に調整して、CPU 時間を多く使用するプロセスのスループットを調整します。タイムシェアリングは優先順位が最も低いスケジューリングクラスです。

SunOS 5.9 のスケジューラでは、リアルタイムスケジューリング方式も使用できます。リアルタイムスケジューリングによって、ユーザーは特定のプロセスに固定優先順位を割り当てることができます。リアルタイムスケジューリングのユーザープロセスは優先順位が最も高く、プロセスが実行可能になり次第 CPU を取得できます。

SunOS 5.9 スケジューラでは、固定優先順位スケジューリング方式も使用できます。固定優先順位スケジューリングによって、ユーザーは特定のプロセスに固定優先順位を割り当てることができます。デフォルトでは、固定優先順位スケジューリング方式はタイムシェアリングスケジューリングクラスと同じ優先順位の範囲を使用します。

リアルタイム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。

リアルタイムスケジューリングによるプロセススケジューリングを制御する必要はほとんどありません。ただし、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、リアルタイムプロセスがそれらの制約を満たす唯一の方法となることがあります。

注意 – リアルタイム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、スケジューラの動作に影響す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影響を与えるインタフェースは次のとおりです。

- `dispadmin`(1M) は、実行中のシステムのスケジューラ構成を表示または変更する。
- `ts_dptbl`(4) と `rt_dptbl`(4) 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リアルタイムのパラメータを含むテーブルである。

作成されたプロセスは、そのクラス内のスケジューリングクラスや優先順位を含むスケジューリングパラメータを継承します。ユーザーの要求によってのみプロセスのスケジューリングクラスが変更されます。システムは、ユーザーの要求とそのプロセスのスケジューリングクラスに関連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、すべてのユーザーログインシェルは、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、プロセスをいつ実行するかを判断します。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。優先順位の高いプロセスが先に実行されます。CPU に割り当てられたプロセスは、プロセスが休眠するか、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循環方式で順番に実行されます。

リアルタイムプロセスは、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 – シングルプロセッサシステムにおいては、実行可能なリアルタイムプロセスが存在している間、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。

管理者はデフォルトのタイムスライスを構成テーブルで指定します。ユーザーはプロセスごとのタイムスライスをリアルタイムプロセスに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(1)` コマンドと `dispadmin(1M)` コマンドで表示できます。

以降の節では、6 つのスケジューリング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 (TS 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最適な応答性能と、CPU 時間を多く使用するプロセスには最適な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、応答性能を高めます。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では、優先順位が動的に変更され、異なる長さのタイムスライスが割り当てられます。CPU をほんの少しだけ使用したあとで休眠しているプロセスの優先順位はスケジューラによって上げられます。たとえば、あるプロセスは端末やディスクの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、休眠せずに CPU を長時間使用す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。ほかのプロセスが CPU を先に取得しても、優先順位の低いプロセスが CPU を取得すると、そのプロセスは長いタイムスライスを取得します。ただし、タイムスライス中に優先順位がより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の優先順位とユーザー指定の優先順位は、昇順になります。優先順位の高いプロセスが先に実行されます。ユーザー指定の優先順位は、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。プロセスはユーザー指定の優先順位を継承します。ユーザー指定の優先順位のデフォルトの初期値は 0 です。

「ユーザー指定の優先順位限界」は、構成によって決まったユーザー指定の優先順位の最大値です。ユーザー指定の優先順位は、この限界値より低い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、ユーザー指定の優先順位限界を上げることができます。ユーザー優先順位限界のデフォルト値は 0 です。

プロセスのユーザー指定の優先順位を下げると、プロセスに与える CPU へのアクセス権を減らすことができます。あるいは、適当なアクセス権をもちいてユーザー指定の優先順位を上げるとサービスを受けやすくできます。ユーザー指定の優先順位はユーザー指定の優先順位限界より高くには設定できません。このどちらの値もデフォルト値の 0 である場合は、ユーザー指定の優先順位を上げる前に、ユーザー指定の優先順位限界を上げる必要があります。

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まったく別にユーザー指定の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、ユーザーはユーザー指定の優先順位を –20 から+20 までの範囲で設定できますしかし、タイムシェアリングのグローバル優先順位は 60 種類まで設定できます。

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(4)` 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固有の情報が含まれます。

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、サーバー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います。ユーザーはシステムクラスにプロセスを追加できません。ユーザーはまた、システムクラスからプロセスを削除できません。システムクラスのプロセスの優先順位はカーネルコードに設定されています。設定されたシステムプロセスの優先順位は変わりません。カーネルモードで動作しているユーザープロセスはシステムクラスではありません。

リアルタイムクラス

リアルタイムクラスでは、固定優先順位スケジューリング方式を使用しているため、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。リアルタイム優先順位は、ユーザーが変更しない限り変更されません。特権ユーザーは、`priocntl`(1) コマンドまたは `priocntl`(2) インタフェースを使用して、リアルタイム優先順位を割り当てることができます。

スケジューラは、リアルタイムパラメータテーブル `rt_dptbl`(4) 内の設定可能なパラメタを使用して、リアルタイムプロセスを管理します。このテーブルには、リアルタイムクラス固有の情報が含まれています。

対話型クラス (IA クラス)

IA クラスは TS クラスにとてもよく似ています。ウィンドウイングシステムと組み合わせて使用すると、プロセスの優先順位は入力フォーカスがあるウィンドウ内で動作している間だけより高くなります。システムがウィンドウイングシステムを実行している場合、デフォルトのクラスは IA クラスです。そうでない場合、IA クラスは TS クラスと同じであり、 2 つのクラスは同じ `ts_dptbl` ディスパッチパラメータテーブルを共有します。

公平共有クラス (FSS クラス)

FSS クラスは、Fair-Share Scheduler (`FSS`(7)) がアプリケーション性能を管理する (つまり、CPU リソースの共有をプロジェクトに明示的に割り当てる) ときに使用されます。共有は、プロジェクトが CPU リソースを利用できる権利を意味します。システムはリソースの使用率を時間の経過とともに監視します。使用率が高い場合、システムは権利を減らします。使用率が低い場合、システムは権利を増やします。FSS は複数のプロセスに CPU 時間をスケジューリングするとき、各プロジェクトが所有するプロセスの数とは無関係に、プロセスの所有者の権利に従います。FSS クラスは、TS クラスおよび IA クラスと同じ優先順位の範囲を使用します。詳細は、`FSS` のマニュアルページを参照してください。

固定優先順位クラス(FX クラス)

FX クラスは、優先順位が固定された横取りのスケジューリング方式です。この方式は、ユーザーまたはアプリケーションがスケジューリング優先順位を制御する必要があるが、システムが動的に調節してはならないプロセス向けです。デフォルトでは、FX クラスは TS クラス、IA クラス、および FSS クラスと同じ優先順位の範囲を使用します。FX クラスを使用すると、ユーザーまたはアプリケーションはこのクラス内のプロセスに割り当てられたユーザー指定の優先順位値を使用して、スケジューリング優先順位を制御できます。このようなユーザー指定の優先順位値は、固定優先順位プロセスのスケジューリング優先順位をそのクラス内のほかのプロセスと相対的に決定します。

スケジューラは固定優先順位ディスパッチパラメータテーブル `fx_dptbl`(4) の構成可能なパラメータを使用して、固定優先順位プロセスを管理します。このテーブルには、固定優先順位クラス固有の情報が収められています。

コマンドとインタフェース

次の図に、デフォルトのプロセス優先順位を示します。

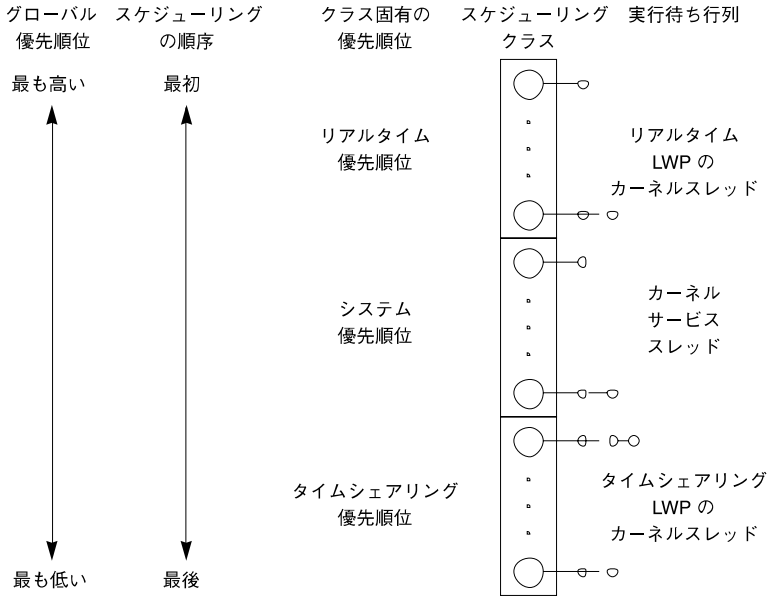


図 **3–1** プロセス優先順位 (プログラマから見た場合)

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、クラスとクラス固有の優先順位の値を指定します。クラスとクラス固有の値は、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ーまたはプログラマから見た場合とで異なります。スケジューリングクラスを設定する場合、システム管理者はグローバル優先順位を直接取り扱います。システムでは、ユーザー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 9 のシステム管理 (基本編)』を参照してください。

ps(1) コマンドで -cel オプションを指定すると、動作中のすべてのプロセスのグローバル優先順位が表示されます。priocntl(1) コマンドを指定した場合、ユーザーとプログラマが使用するクラス固有の優先順位が表示されます。

priocntl(1) コマンド、priocntl(2) インタフェース、および priocntlset(2) インタフェースは、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するのに使用されます。優先順位を設定するときには、これらのコマンドおよびインタフェースいずれの場合でも、次の同じ手順に従います。

1. ターゲットプロセスを指定する。
2. 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。
3. プロセスにパラメータを設定するコマンドまたはインタフェースを実行する。

プロセス ID は UNIX プロセスの基本設定項目です。詳細は、Intro(2) 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(2) 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 の使用法

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 つの制御インタフェースを実行します。

priocntl –l	構成情報を表示する
priocntl –d	プロセスのスケジューリングパラメータを表示する
priocntl –s	プロセスのスケジューリングパラメータを設定する
priocntl –e	指定したスケジューリングパラメータでコマンドを実行する

次に、priocntl(1) を使用したいくつかの例を示します。

- -l オプションを使用すると、次のようにデフォルトの設定が出力されます。

```
$ priocntl -l
CONFIGURED CLASSES
==================

SYS (System Class)

TS (Time Sharing)
Configured TS User Priority Range -60 through 60

RT (Real Time)
Maximum Configured RT Priority: 59
```

- すべてのプロセスの情報を表示するには、次のように指定します。

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

- すべてのタイムシェアリングプロセスの情報を表示するには、次のように指定します。

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

- ユーザーID が 103 または 6626 のすべてのプロセスの情報を表示するには、次のように指定します。

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

- ID 24668 のプロセスをデフォルトのパラメータでリアルタイムプロセスに設定するには、次のように指定します。

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

- ID 3608 のプロセスを、優先順位 55、タイムスライス 5 分の 1 秒のリアルタイムプロセスに設定するには、次のように指定します。

```
$ 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には、次のように指定します。

```
$ priocntl -s -c TS -i all
```

- ユーザー ID が 1122 のプロセスのタイムシェアリングユーザー指定の優先順位とユーザー指定の優先順位制限を -10 に減らすには、次のように指定します。

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

- リアルタイムシェルをデフォルトのリアルタイム優先順位で起動するには、次のように指定します。

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

- make をタイムシェアリングユーザー優先順位 -10 で実行するには、次のように指定します。

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

priocntl(1) には、nice(1) のインタフェースが含まれます。nice(1) 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。上記の例は、nice(1) を使用してインクリメントを 10 に設定するのと同じです。

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

priocntl インタフェース

priocntl(2) は、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ータを管理します。priocntl(2) 呼び出しは LWP、単独のプロセス、またはプロセスのグループに関して動作することができます。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、ユーザー、グループ、クラス、または動作中のすべてのプロセスによって識別できます。詳細は、priocntl のマニュアルページを参照してください。

クラス ID を指定した場合、PC_GETCLINFO コマンドはスケジューリングクラス名とパラメータを取得します。このコマンドを使用すると、設定するクラスを想定しないプログラムを作成できます。

PC_SETXPARMS コマンドは、一組のプロセスのスケジューリングクラスとパラメータを設定します。*idtype* と *id* の入力引数は、変更するプロセスを指定します。

その他のインタフェースとの関係

あるタイムシェアリングクラスのプロセスの優先順位を変更すると、そのタイムシェアリングクラスのほかのプロセスの動作に影響する可能性があります。この節では、スケジューリングの変更がどのように他のプロセスに影響するかについて説明します。

カーネルプロセス

カーネルのデーモンやハウスキーピングプロセスはシステムスケジューリングクラスのメンバーです。ユーザーは、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することはできません。ps –cel コマンドを実行すると、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。ps(1) コマンドで -f オプションを指定すると、システムクラスのプロセスには、CLS カラムの SYS と表示されます。

fork と exec

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) インタフェースや exec(2) インタフェースを実行した場合も継承されます。

nice

nice(1) コマンドと nice(2) インタフェースは、UNIX システムの以前のバージョンと同じ動作になります。これらのコマンドは、タイムシェアリングプロセスの優先順位を変更します。これらのインタフェースでも、数値が小さいほどタイムシェアリング優先順位は高くなります。

プロセスのスケジューリングクラスを変更したり、リアルタイム優先順位を指定したりするには、priocntl(2) を使用します。数値が大きいほど優先順位は高くなります。

init(1M)

init(1M) プロセスは、スケジューラに対しては特殊なケースとして動作します。init(1M) 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 だけをプロセスに指定する必要があります。

性能

スケジューラは、プロセスをいつどのくらいの時間実行するかを決定します。したがって、スケジューラの動作はシステム性能に大きな影響を与えます。

デフォルトでは、ユーザー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(2) 関数呼び出しによってだけです。

リアルタイム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。リアルタイムプロセスが実行可能である間、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できません。CPU の制御に失敗することがあるリアルタイムアプリケーションは、その他のユーザーや重要なカーネルハウスキーピングを完全にロックアウトする可能性があります。

プロセスのクラスと優先順位を制御する以外に、リアルタイムアプリケーションは、性能に影響するほかの要因も制御する必要があります。性能に最も影響する要因は、CPU、一次メモリー量、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

リアルタイム制約が厳しい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ーにページアウトされたりしないようにする必要があります。次の図では、UNIX のプロセスの状態と状態間の変移の概要を示します。

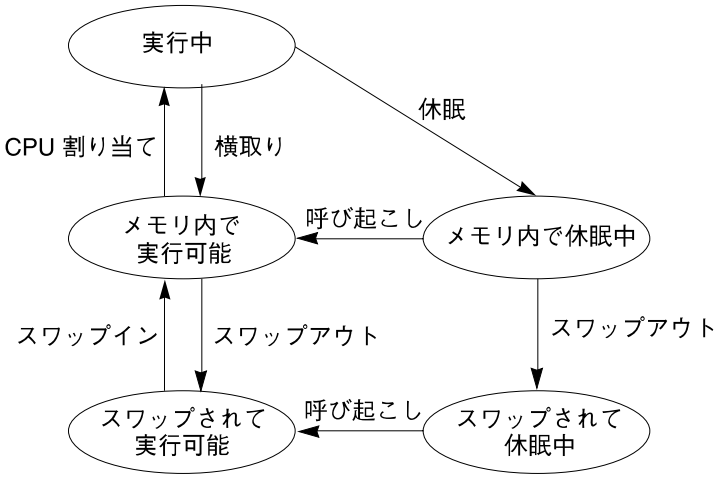


図 **3–2** プロセス状態の変移図

動作中のプロセスは、通常、上記の図の 5 つのうち 1 つの状態にあります。矢印は、プロセスの状態の変化を示します。

- 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、優先順位が低い実行中のプロセスはスケジューラによって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たときに、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。
- プロセスが一次メモリー内にあり、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、メモリー内で実行可能である。
- プロセスが一次メモリー内にあるが、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、メモリー内で休眠中である。たとえば、入出力操作の完了、ロックされている資源の解放、またはタイマの終了を待っている間、プロセスは休眠する。イベントが発生すると、ウェイクアップコールがプロセスに送信される。休眠の原因が解消されると、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おらず、そのアドレス空間全体が二次メモリーに書き込まれている場合、そのプロセスは実行可能な状態であり、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、そのアドレス空間全体が二次メモリーに書き込まれている場合、そのプロセスは休眠中であり、スワップされている。

  動作中のプロセスをすべて保留するために十分な一次メモリーがマシンにない場合は、アドレス空間の一部を二次メモリーにページングするかスワップする必要があります。
- システムの一次メモリーが不足した場合は、いくつかのプロセスの個々のページが二次メモリーに書き込まれるが、そのプロセスは実行可能なままである。実行中のプロセスがそのページにアクセスする場合、ページが一次メモリー内に読み戻され

るまでプロセスは休眠する。

- システムの一次メモリー不足がさらに深刻になると、いくつかのプロセスのすべてのページが二次メモリーに書き込まれる。このように二次メモリーに書き込まれたページは「スワップされた」とマークされる。このようなプロセスがスケジュール可能な状態に戻るのは、システムスケジューラデーモンがこれらのプロセスをメモリー内に読み戻すように選択した場合だけである。

プロセスが再度実行可能になった場合、ページングとスワップの両方により、遅延が発生します。タイミング要求が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。

リアルタイムプロセスにすれば、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。プログラムは、テキストとデータを一次メモリー内にロックして、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。ロックできる量はメモリー設定によって制限されます。また、ロックが多すぎると、テキストやデータをメモリー内にロックしていないプロセスが大幅に遅れる可能性があります。

リアルタイムプロセスの性能とその他のプロセスとの性能の兼ね合いは、ローカルな必要性によって異なります。システムによっては、必要なリアルタイム応答を保証するためにプロセスのロックが必要な場合もあります。

注 – リアルタイムアプリケーションの応答時間については、187 ページの「ディスパッチ応答時間」を参照してください。

第 4 章

入出力インタフェース

この章では、仮想メモリーサービスのないシステムに対するファイル入出力操作を紹介します。この章ではまた、仮想記憶機能によって向上した入出力方式についても説明します。70 ページの「ファイルとレコードのロックの使用」では、ファイルとレコード のロックの以前の方法について説明します。

ファイルと入出力インタフェース

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」と呼びます。通常ファイルには、ASCII テキスト、ほか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。

通常ファイルの構成要素は次のとおりです。

- 「*i* ノード」と呼ばれる制御データ。この制御データには、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シーケンス。

Solaris オペレーティングシステムでは、次のような基本的なファイル入出力インタフェースが提供されています。

- 従来の raw スタイルのファイル入出力については、68 ページの「基本ファイル入出力」を参照してください。
- 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、仮想メモリーのないシステムで実行するアプリケーションの効率を改善できます。SunOS™ オペレーティングシステムのような仮想メモリー環境で動作しているアプリケーションでは、標準のファイル入出力は利用しなくなっています。
- メモリーマッピングインタフェースについては、13 ページの「メモリー管理インタフェース」を参照してください。マッピングファイルは、SunOS プラットフォームで動作するほとんどのアプリケーションに最も効率的なファイル入出力形式です。

基本ファイル入出力

次のインタフェースは、ファイルとキャラクタ入出力デバイス上で基本的な操作を実行します。

表 **4–1** 基本的なファイル入出力インタフェース

インタフェース名	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)	読み取りまたは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、実際に転送されたバイト数が返されます。read(2) では、ファイルの終わりは戻り値が 0 になります。

例 **4–1** 基本的なファイル入出力インタフェース

```
#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);
}
```

ファイルの読み取りまたは書き込みが完了した後は必ず、そのファイルに対して close(2) を呼び出してください。open(2) の呼び出しが完了していないファイル記述子に対しては close(2) を呼び出してはなりません。

開いたファイルへのファイルポインタオフセットを変更するには、read(2) または write(2) を使用するか、lseek(2) を呼び出します。次の例では、lseek の使い方を示します。

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

 /* 現在のオフセットの位置を start にセットする */
 start = lseek (fd, 0L, SEEK_CUR);

 /* start に戻る */
 n = lseek (fd, -start, SEEK_SET);
 read (fd, &rec, sizeof (rec));

 /* 前のレコードを書き換える */
 n = lseek (fd, -sizeof (rec), SEEK_CUR);
 write (fd, (char *&rec, sizeof (rec));
```

高度なファイル入出力

次の表に、高度なファイル入出力インタフェースが実行するタスクの一覧を示します。

表 **4–2** 高度なファイル入出力インタフェース

インタフェース名	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断する
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
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)	ファイルの状態を取得する
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

表 **4–2** 高度なファイル入出力インタフェース　(続き)

インタフェース名	目的
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取る
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

次の表にあるファイルシステム制御インタフェースを用いて、ファイルシステムに対してさまざまな制御を行うことができます。

表 **4–3** ファイルシステム制御インタフェース

インタフェース名	目的
ustat(2)	ファイルシステムの統計情報を取得する
sync(2)	スーパーブロックを更新する
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クの使用

ファイル要素をロックするために従来のファイル入出力を使用する必要はありません。マッピングされたファイルには、より軽量な同期メカニズム (『マルチスレッドのプログラミング』を参照) を使用します。

ファイルをロックすると、複数のユーザーが同時にファイルを更新しようとした場合に生じるエラーを防止できます。ファイルの一部だけもロックできます。

ファイルをロックすると、そのファイル全体へのアクセスがブロックされます。レコードをロックすると、そのファイルの指定されたセグメントへのアクセスがブロックされます。SunOS 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。

ロックタイプの選択

強制ロックでは、要求されたファイルセグメントが解放されるまで、プロセスは保留されます。アドバイザリロックでは、ロックが取得されたかどうかを示す結果だけが返されます。プロセスはアドバイザリロックの結果を無視できます。同一のファイルに強制ロックとアドバイザリロックを同時に適用することはできません。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クが強制ロックとして処理されるか、アドバイザリロックとして処理されるかが決まります。

2 つの基本的なロック呼び出しのうち、`fcntl`(2) は `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 で規格化されています。`lockf`(3C) は、ほか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。

アドバイザリロックと強制ロックの選択

強制ロックの場合、対象のファイルはグループ ID の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。どちらかの条件が欠けていると、すべてのレコードロックはアドバイザリロックになります。

次のように強制ロックを設定します。

```
#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;
     /* グループの実行権をモードから削除する */
     mode &= ~(S_IEXEC>>3);
     /* グループ ID の設定ビットをモードに設定する */
     mode |= S_ISGID;
     if (chmod(filename, mode) < 0) {
         perror("program");
         exit(2);
     }
     ...
```

ファイルを実行するとき、オペレーティングシステムはレコードロックを無視します。レコードロックが適用されるファイルには実行権を設定しないでください。

ファイルに強制ロックを設定するには、次のように `chmod`(1) コマンドも可能です。

```
$ chmod +l file
```

このコマンドはファイルモード内に O2On0 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クを示します。*n* が偶数の場合、そのビットは強制ロックを有効にすると解釈され、*n* が奇数の場合、そのビットは「実行時グループ ID 設定」として解釈されます。

この設定を表示するには、ls(1) コマンドに –l オプション (ロングリスト形式) を指定して実行します。

```
$ ls -l file
```

すると、次のような情報が表示されます。

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

アクセス権の文字「l」は、グループ ID の設定ビットがオンであることを示します。グループ ID の設定ビットがオンであるので、強制ロックは有効です。グループ ID の設定ビットの通常の意味論も有効です。

強制ロックについての注意事項

ロックについては、次の点について注意してください。

- 強制ロックは、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、強制ロックはサポートされません。
- 強制ロックは、ファイル内のロック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。
- 不可分のトランザクションに多重の読み取りや書き込みが必要な場合は、入出力を開始する前に、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ク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クで十分です。
- レコードロックが使用されるファイルについては、全プログラムに無制限のアクセス権を与えてはいけません。
- 入出力要求のたびにレコードロック検査を実行する必要がないため、アドバイザリロックの方が効率的です。

サポートされるファイルシステム

次の表に、アドバイザリロックと強制ロックの両方がサポートされるファイルシステムの一覧を示します。

表 **4–4** サポートされるファイルシステム

ファイルシステム	説明
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の先頭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ク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ク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、ファイルロックはサポートされません。

ロック用にファイルを開く

ロックを要求できるのは、有効な開いたファイル記述子を持つファイルだけです。読み取りロックの場合は、少なくとも読み取りアクセスを設定してファイルを開く必要があります。書き込み用ロックの場合は、書き込みアクセスも設定してファイルを開く必要があります。次の例では、ファイルは読み取りと書き込みの両方の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);
     }
     ...
```

ファイルロックの設定

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

ファイルをロックする方法はいくつかあります。どの方法を選択するかは、ロックとプログラムのほかの部分との関係、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) インタフェースを使用します。fcntl(2) インタフェースは、次のいずれかの状況が発生するまでファイルをロックしようとします。

- ファイルロックが正常に設定された
- 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クを中止した

```
#include <fcntl.h>

 ...
     struct flock lck;

 ...
     lck.l_type = F_WRLCK;     /* 書き込みロックを設定する */
     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) を使用すると、構造体の変数を設定し、ロック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は flock(3UCB) ではロックできません。ただし、マルチスレッド指向の同期メカニズムを使用すると、マッピングされたファイルをロックできます。このような同期メカニズムは POSIX スタイルと Solaris スタイルのどちらでも使用できます。mutex(3THR)、condition(3THR)、semaphore (3THR)、mmap(2)、および rwlock(3THR) のマニュアルページを参照してください。

レコードロックの設定と解除

レコードをロックする場合、ロックセグメントの開始位置と長さを 0 に設定してはなりません。それ以外、レコードのロックはファイルのロックと同じです。

レコードロックを使用するのは、データが競合するためです。したがって、必要なすべてのロックを設定できない場合に備えて、次のような対処方法を用意しておく必要があります。

- 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
- ロック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

次の例に、fcntl(2) を使用してレコードをロックする方法を示します。

```
{
     struct flock lck;
       ...
```

```
    lck.l_type = F_WRLCK;     /* 書き込みロックを設定する*/
    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 に書き込みロックを設定する */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* this の書き込みロックが失敗 */
        return (-1);
 ...
}
```

次の例に、lockf(3C) インタフェースを示します。

```
#include <unistd.h>

{
 ...
    /* this をロックする */
    (void) lseek(fd, this, SEEK_SET);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* this のロックが失敗。here のロックを解除する */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);
}
```

ロックの解除は設定と同じように行います。ロックタイプが異なるだけです (F_ULOCK)。ロックの解除は別のプロセスによってブロックされず、そのプロセスが設定したロックに対してだけ有効です。ロック解除は、前のロック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。

ロック情報の取得

どのプロセスがロックを保留しているかを判断できます。ロックは上記の例のように設定され、fcntl(2) で F_GETLK が使用されます。

次の例では、ファイル内でロック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。

例 4–2 ファイル内でロックされているセグメントの出力

```
struct flock lck;

    lck.l_whence = 0;
    lck.l_start = 0L;
    lck.l_len = 0L;
    do {
        lck.l_type = F_WRLCK;
```

例 **4–2** ファイル内でロックされているセグメントの出力　(続き)

```
        (void) fcntl(fd, F_GETLK, &lck);
        if (lck.l_type != F_UNLCK) {
            (void) printf("%d %d %c %8ld %8ld\n", lck.l_sysid, lck.l_pid,
           (lck.l_type == F_WRLCK) ? 'W' : 'R', lck.l_start, lck.l_len);
            /* このロックがアドレス空間の終わりまで続いている場合、
             * 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、見つかったロックの後方にあるロックを探す */
                    lck.l_start += lck.l_len;
            }
        }
    } while (lck.l_type != F_UNLCK);
```

F_GETLK コマンドを指定すると、fcntl(2) はサーバーが応答するまで待機および休眠できます。fcntl(2) はまた、クライアントまたはサーバー側の資源が不足すると失敗して、ENOLCK を返すことがあります。

F_TEST コマンドを指定すると、 lockf(3C) はプロセスがロックを保留しているかどうかを検査できます。このインタフェースは、ロックの位置と所有権についての情報を返しません。

例 **4–3** lockf によるプロセスの検査

```
(void) lseek(fd, 0, 0L);
 /* ファイルの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;
     }
```

プロセスのフォークとロック

プロセスがフォークを行うと、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、ロック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。親プロセスと子プロセスは、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ク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。レコードのロックを保留しているプログラムがフォークを行う場合、子プロセスはまず、そのファイルを閉じる必要があります。ファイルを閉じた後、子プロセスはそのファイルを開き直して、新しい異なるファイルポインタを設定する必要があります。

デッドロック処理

UNIX のロック機能を使用すると、デッドロックを検出および防止できます。デッドロックが発生する可能性があるのは、システムがレコードロックインタフェースを休眠させようとするときだけです。(このとき)、2 つのプロセスがデッドロック状態であるかどうかを判断する検索が行われます。潜在的なデッドロックが検出されると、ロックインタフェースは失敗し、デッドロックを示す値が `errno` に設定されます。`F_SETLK` を使用してロックを設定するプロセスは、ロックがすぐに取得できなくてもそれを待たないので、デッドロックは発生しません。

端末入出力インタフェース

端末入出力インタフェースは、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。詳細は、`termios`(3C) と `termio`(7I) のマニュアルページを参照してください。

表 **4–5** 端末入出力インタフェース

インタフェース名	目的
`tcgetattr(3C),tcsetattr(3C)`	端末属性を取得または設定する
`tcsendbreak`(3C)、`tcdrain`(3C)、`tcflush`(3C)、`tcflow`(3C)	回線制御インタフェースを実行する
`cfgetospeed(3C),cfgetispeed(3C), cfsetispeed(3C),cfsetospeed(3C)`	ボーレートを取得または設定する
`tcsetpgrp(3C)`	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(3C)`	端末のセッション ID を取得する

次の例に、`DEBUG` 以外の操作モードにおいて、サーバーがどのようにその呼び出し元の制御端末との関連付けを解除するかを示します。

例 **4–4** 制御端末との関連付けを解除する

```
(void) close(0);
(void) close(1);
(void) close(2);
```

例 **4–4** 制御端末との関連付けを解除する　(続き)

```
(void) open("/", O_RDONLY);
(void) dup2(0, 1);
(void) dup2(0, 2);
setsid();
```

この操作モードでは、サーバーは制御端末のプロセスグループからシグナルを受信しません。サーバーが関連付けを解除した後、サーバーはエラーレポートを端末に送信できません。したがって、このサーバーは `syslog`(3C) を使用してエラーを記録する必要があります。

第 5 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。

SunOS 5.9 およびその互換オペレーティングシステムは、並行プロセスがデータを交換し、実行の同期をとるためのさまざまなメカニズムを持っています。この章では、これらのメカニズムについて説明します (ただし、マッピングされたメモリーを除く)。

- 80 ページの「プロセス間のパイプ」では、パイプ (匿名のデータ待ち行列) について説明します。
- 81 ページの「名前付きパイプ」では、名前付きパイプ (ファイル名を持つデータ待ち行列) について説明します。
- 84 ページの「System V IPC」では、System V のメッセージ待ち行列、セマフォ、および共有メモリーについて説明します。
- 82 ページの「POSIX プロセス間通信」では、POSIX のメッセージ待ち行列、セマフォ、および共有メモリーについて説明します。
- 81 ページの「ソケット」では、ソケットを使用したプロセス間通信について説明します。
- 13 ページの「メモリー管理インタフェース」では、マッピングされたメモリーとファイルについて説明されています。

プロセス間のパイプ

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」と呼びます。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接続したり、あるいは 1 本のパイプでその子プロセスに関連する親プロセスと接続したり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返します。フォーク後、両方のプロセスは p[0] から読み取り、p[1] に書き込みます。実際には、これらのプロセスが読み取りまたは書き込みを行うのは循環バッファーに対してであり、この循環バッファーを管理することによって、プロセスの代わりにパイプとの読み取りまたは書き込みを行うことができます。

fork(2) を使用すると各プロセスの開いているファイルテーブルが複写されるので、各プロセスは 2 つのリーダー (読み取り用パイプ) と 2 つのライター (書き込み用パイプ) を持つことになります。パイプを適切に機能させるには、余分なリーダーとライターを閉じる必要があります。たとえば、同じプロセスが片方のリーダーを書き込み用に開いたまま、もう一方のリーダーから読み取ろうとすると、EOF (ファイルの終わり) は返されません。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:                          /* 子プロセス */
             close( p[0] );
             dup2( p[1], 1);
             close P[1] );
             exec( ... );
             exit(1);
         default:                          /* 親プロセス */
             close( p[1] );
             dup2( P[0], 0 );
             close( p[0] );
             break;
     }
     ...
```

ある条件下で、パイプからの読み取りパイプへの書き込みを行うと、次の表のようになります。

表 5–1 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、ライター接続	読み取りはブロックされる
書き込み	フルのパイプ、リーダー接続	書き込みはブロックされる
読み取り	空のパイプ、接続ライターなし	EOF が戻される
書き込み	リーダーなし	SIGPIPE

fcntl(2) を記述子に呼び出して FNDELAY を設定すると、ブロックを阻止でき、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 が返されます。

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クします。

ブロックせずに名前付きパイプを開くには、open(2) を呼び出すときに、(sys/fcntl.h にある) O_NDELAY マスクと選択したファイルモードマスクの論理和を取ります。open(2) を呼び出したときにほか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 が返されます。

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、プロセス間通信とシステム間通信の基本的な構成要素です。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 つの形式と 1 つまたは複数の関連プロセスを持ちます。

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。ソケットドメインは 23 個ありますが (`sys/socket.h` を参照)、Solaris 9 およびその互換オペレーティングシステムでは通常、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、ほかの形態の IPC と同様に、単一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 パスで名前付けされます。UNIX ドメインのソケットについての詳細は、第 6 章を参照してください。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。インターネットドメインソケットについては、第 6 章を参照してください。

POSIX プロセス間通信

POSIX プロセス間通信 (IPC) は System V プロセス間通信の変形です。 POSIX プロセス間通信は Solaris 7 で導入されました。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が、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。POSIX IPC には、次のような機能が含まれます。

- 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るメッセージ
- プロセスが実行の同期を取ることができるセマフォ
- 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有できる共有メモリー

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

次の表に、POSIX メッセージ待ち行列インタフェースの一覧を示します。

表 5–2 POSIX メッセージ待ち行列インタフェース

インタフェース名	種類
`mq_open(3RT)`	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

表 **5–2** POSIX メッセージ待ち行列インタフェース　(続き)

インタフェース名	種類
mq_close(3RT)	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(3RT)	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(3RT)	メッセージを待ち行列に入れる
mq_receive(3RT)	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(3RT)	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(3RT)、mq_getattr (3RT)	メッセージ待ち行列属性を設定または取得する

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。

次の表に、POSIX セマフォインタフェースの一覧を示します。

表 **5–3** POSIX セマフォインタフェース

sem_open(3RT)	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(3RT)	名前なしセマフォ構造体を初期化する (呼び出し元プログラムの内部で行われるのため、名前付きセマフォではない)
sem_close(3RT)	開いているセマフォへの接続を終了する
sem_unlink(3RT)	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(3RT)	名前なしセマフォ構造体を初期化する (呼び出し元プログラムの内部で行われるのため、名前付きセマフォではない)
sem_getvalue(3RT)	セマフォの値を指定された整数にコピーする
sem_wait(3RT)、sem_trywait(3RT)	セマフォがほかのプロセスによって保持されている場合に、ブロックするかエラーを返す
sem_post(3RT)	セマフォの数を増やす

POSIX 共有メモリー

POSIX 共有メモリーは、実際にはマッピングされているメモリーの変形です (詳細は、13 ページの「マッピングの作成と使用」を参照)。主な違いは、以下のとおりです。

- 共有メモリーオブジェクトを開くには、open(2) を呼び出すのではなく、shm_open(3RT) を使用する。
- オブジェクトを閉じるまたは削除するには、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(3RT) を使用する。

shm_open(3RT) 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

SunOS 5.9 およびその互換オペレーティングシステムは、System V のプロセス間通信 (IPC) パッケージも提供します。System V IPC は事実上 POSIX IPC に置き換えられましたが、以前のアプリケーションをサポートするために現在も提供されています。

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。

メッセージ、セマフォ、および共有メモリーのアクセス権

メッセージ、セマフォ、および共有メモリーは、通常ファイルと同じように、所有者、グループ、およびその他のユーザーのための読み取り権と書き込み権を持っています (実行権はない)。ファイルと同じ点は、作成元プロセスがデフォルトの所有者を識別することです。ファイルとは異なる点は、作成者は機能の所有権を別のユーザーに割り当てたり、所有権割り当てを取り消したりすることができる点です。

IPC インタフェース、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その機能を識別できる必要があります。アクセス権を要求する IPC 機能をプロセスが識別できるようにするために、IPC 機能へのアクセスを初期化または提供するインタフェースは key_t というキー引数を使

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。このようなキーは、ftok(3C) を使用して、ファイル名をシステム内で一意のキー値に変換することで導くこともできます。

メッセージ、セマフォ、または共有メモリーへのアクセスを初期化または取得するインタフェースは int 型の ID 番号を返します。IPC インタフェースの読み取り、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。

呼び出しに適切なフラグ引数として IPC_CREAT フラグを指定した場合、IPC 機能が存在していなければ、インタフェースはその IPC 機能を新たに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インタフェースを呼び出した場合、IPC 機能がすでに存在していれば、インタフェースは失敗します。この動作は複数のプロセスが 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。サーバープロセスがすべて 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。

IPC_CREAT と IPC_EXCL 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、インタフェースはその機能のID を返して、アクセスを取得できるようにします。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。

論理 (ビット単位) OR を使用すると、IPC_CREAT と IPC_EXCL を 8 進数のアクセス権モードと組み合わせることによって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「'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) を使用して待ち行列を初期化する必要があります。待ち行列の所有者または作成者は msgctl(2) を使用して、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 msgctl(2) を使用して、操作を制御できます。

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、サーバープロセスはクライアントプロセス ID をメッセージタイプとして使用することによって、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、複数のサーバープロセスは、共有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。

メッセージを送受信する操作はそれぞれ msgsnd(2) と msgrcv(2) 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) と msgrcv(2) は、ブロック操作としても非ブロック操作としても実行できます。ブロック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) 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) を返すこともできます。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 進数の整数である必要があります。

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(2) 関数は失敗します。

次のコードに、msgget(2) 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)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 である必要があります。cmd 引数は、次のいずれか 1 つです。

IPC_STAT　待ち行列の状態の情報を 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つ必要がある。

IPC_SET　所有者のユーザー 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ーの有効なユーザー ID を持つ必要がある。

IPC_RMID　msgid 引数で指定したメッセージ待ち行列を削除する。

次のコードに、さまざまなフラグをすべて指定した msgctl(2) の使用例を示します。

```
#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) はそれぞれメッセージを送信および受信します。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む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 引数は、さまざまな制御フラグを渡します。

次のコードに、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;     /* メッセージフラグ */
     struct msgbuf      *msgp;     /* メッセージバッファのポインタ */
```

```
size_t            msgsz;    /* メッセージサイズ */
   size_t            maxmsgsize;
long                msgtyp;    /* メッセージタイプ */
int                 msqid        /* メッセージ待ち行列 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、セマフォは共有メモリー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します。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、極めて重いセマフォです。より軽量なセマフォはスレッドライブラリ (`semaphore`(3THR) のマニュアルページを参照) で利用できます。また、POSIX セマフォは System V セマフォの最新の実装です (83 ページの「POSIX セマフォ」を参照)。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ーで使用する必要があります (13 ページの「メモリー管理インタフェース」を参照)。

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できており、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) を使用して初期化する必要があります。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、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) によって行います。このインタフェースは、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。

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、プロセスはブロックし、次のいずれかになるまでブロックされたままです。

- 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、操作の取り消しまたはセマフォの解放に失敗した場合、セマフォはメモリー内にロッキングされたままになります。この現象を防ぐには semop(2) に SEM_UNDO 制御フラグを指定して、各セマフォ操作に undo 構造体を割り当て、セマフォを以前の状態に戻すことができるようにします。プロセスが異常終了すると、undo 構造体内の操作がシステムによって適用されます。これにより、プロセスが異常終了しても、セマフォの整合性が保たれます。

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有する場合は、SEM_UNDO を有効にしてセマフォに対する操作を行わないでください。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るようにする必要があります。

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 を有効にしておく必要があります。プロセスが正常に実行されると、取り消し操作は undo 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、undo 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。undo 構造体は 0 になると削除されます。

SEM_UNDO を正しく使用しないと、割り当てられた undo 構造体がシステムをリブートするまで解放されないため、メモリーリークが発生する可能性があります。

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) を返します。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と呼び出しは失敗します。正しい数がわからない場合は、nsems 引数を 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。ただし、セマフォのセット間の断片化のため、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。

次のコードに、semget(2) の使用例を示します。

```
#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) では、有効なセマフォ ID を指定して呼び出してください。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、arg は int 値の 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、arg は unsigned short 値の配列へのポインタである arg.array と解釈される

SETALL　セット内のすべてのセマフォに値を設定する。この場合、arg は unsigned short 値の配列へのポインタである 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、semid_ds 型のバッファーへのポインタ arg.buf が指すデータ構造体に入れる

IPC_SET　有効なユーザー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ーとして有効なユーザー識別子を持つ必要があります。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。

次のコードに、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、以前の semget(2) 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む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。*SEMOPM* オプションは単一の semop(2) 呼び出しで指定できる最大操作数で、デフォルトでは 10 に設定されています。

実行する操作は、次のように判断されます。

- 正の整数の場合は、セマフォの値をその数だけ増加する

- 負の整数の場合は、セマフォの値をその数だけ減少する。セマフォを 0 未満の値に設定しようとすると、IPC_NOWAIT が有効であるかどうかによって、失敗するかブロックされる
- 値が 0 の場合は、セマフォの値が 0 になるのを待つ

semop(2) で使用できる制御フラグは IPC_NOWAIT と SEM_UNDO の 2 つです。

IPC_NOWAIT	配列内のどの操作についても設定できる。IPC_NOWAIT が設定されている操作を実行できなかった場合、セマフォの値を変更せずにインタフェースを戻す。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 0 でないときに 0 かどうか検査しようとするとインタフェースは失敗する
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

次のコードに、semop(2) 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 共有メモリー

SunOS 5.9 オペレーティングシステムで共有メモリーアプリケーションを実装するには、mmap(2) とシステムの内蔵仮想メモリー機能を利用する方法が最も効率的です。詳細は、第 1 章を参照してください。

SunOS 5.9 は System V 共有メモリーもサポートしますが、物理メモリーのセグメントを複数のプロセスの仮想アドレス空間に接続する方法としては最適ではありません。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メカニズムを使用して、不整合や衝突などを回避できます。

プロセスは、shmget(2) を使用して共有メモリーセグメントを作成します。この呼び出しは、既存の共有セグメントの 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。

共有メモリーセグメントの元の所有者は、shmctl(2) を使用して所有権をほかのユーザーに割り当てることができます。所有者はこの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、ほかのプロセスも shmctl(2) を使用して共用メモリーセグメントにさまざまな制御機能を実行できます。

共有メモリーセグメントを作成した後は、shmat(2) を使用してプロセスのアドレス空間にセグメントを接続できます。切り離すには shmdt(2) を使用します。プロセスを接続するには shmat(2) に対しての適当なアクセス権を持つ必要があります。接続すると、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。共有セグメントは、同じプロセスによって何回でも接続できます。

共有メモリーセグメントは、物理メモリー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 ID は shmid と呼びます。共有メモリーセグメントの制御構造体は sys/shm.h に定義されています。

共有メモリーセグメントのアクセス

shmget(2) を使用して、共有メモリーセグメントへアクセスします。成功すると、共有メモリーセグメント ID (*shmid*) を返します。次のコードに、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() に渡すサイズ */
     ...
     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);
     }
 ...
```

共有メモリーセグメントの制御

shmctl(2) を使用して、共有メモリー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	指定したメモリー内の共有メモリーセグメントをロックする。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ーの ID を持つ必要がある
SHM_UNLOCK	共有メモリーセグメントのロックを解除する。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ーの ID を持つ必要がある
IPC_STAT	制御構造体にある状態情報を取得して、buf が指すバッファーに入れる。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つ必要がある
IPC_SET	有効なユーザー ID およびグループ ID 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ーの有効な ID を持つ必要がある
IPC_RMID	共有メモリー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ーの有効な ID を持つ必要がある

次のコードに、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有メモリーデータ構造体 */
     ...
     shmid = ...
     cmd = ...
     if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
         perror("shmctl: shmctl failed");
         exit(1);
     ...
```

共有メモリーセグメントの接続と切り離し

共有メモリーセグメントの接続と切り離しを行うには、shmat() と shmdt() を使用します(shmop(2) のマニュアルページを参照)。shmat(2) は、共有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有メモリーセグメントを切り離します。次のコードに、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];            /* 現在接続されているセグメントの状態*/
     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 6 章

ソケットインタフェース

この章では、ソケットインタフェースについて説明します。また、プログラム例を使用して重要なポイントを示します。この章の内容は次のとおりです。

- 97 ページの「SunOS 4 のバイナリ互換性」では、SunOS 4 とのバイナリ互換性について説明します。
- 101 ページの「ソケットの基本的な使用」では、ソケットの作成、コネクション、および終了について説明します。
- 121 ページの「クライアントサーバープログラム」では、クライアントサーバーアーキテクチャについて説明します。
- 125 ページの「ソケットの拡張機能」では、マルチキャストや非同期ソケットなどの拡張機能について説明します。

注 – この章で説明するインタフェースは、マルチスレッドに対して安全です。ソケットインタフェースの呼び出しを含むアプリケーションは、マルチスレッド対応のアプリケーションで自由に使用できます。ただし、アプリケーションに有効な多重度は指定されていません。

SunOS 4 のバイナリ互換性

SunOS 4 以降で行われた 2 つの主な変更は SunOS 5.9 リリースでも継承されています。パッケージにバイナリ互換性があるため、動的にリンクされた SunOS 4 ベースのソケットアプリケーションは SunOS 5.9 でも実行できます。

- コンパイル行で、ソケットライブラリ (`-lsocket` または `libsocket`) を明示的に指定する必要があります。
- 場合によっては `libnsl` もリンクする必要があります (`-lnsl -lsocket` ではなく `-lsocket -lnsl` と指定する)。

- SunOS 5.9 で実行するには、ソケットライブラリを使用して SunOS 4 のソケットベースアプリケーションをすべてコンパイルし直す必要があります。

ソケットの概要

ソケットは、1981 年以来 SunOS リリースに不可欠な部分となっています。ソケットは、名前を結合できる通信の終端です。ソケットにはタイプがあり、関連プロセスが 1 つ存在します。ソケットは、次のようなプロセス間通信のためのクライアントサーバーモデルを実装するために設計されました。

- ネットワークプロトコルのインタフェースが、TCP/IP、Xerox インターネットプロトコル (XNS)、UNIX ファミリのような複数の通信プロトコルを提供する必要がある
- ネットワークプロトコルのインタフェースが、コネクションを待機するサーバーコードとコネクションを開始するクライアントコードを提供する必要がある
- 通信がコネクション型であるかコネクションレス型であるかによって操作を変える必要がある
- アプリケーションプログラムが、`open`(2) 呼び出しを使用してアドレスをバインドするのではなく、配信しようとしているデータグラムの着信先アドレスを指定する必要がある

ソケットは、UNIX ファイルのように動作し、ネットワークプロトコルが使用できるようにします。アプリケーションは、必要に応じてソケットを作成します。ソケットは、`close`(2)、`read`(2)、`write`(2)、`<ioctl`(2)、および `fcntl`(2) インタフェースと連携して動作します。オペレーティングシステムは、ファイルの ファイル記述子とソケットのファイル記述子を区別します。

ソケットライブラリ

ソケットインタフェースルーチンは、アプリケーションとリンクが必要なライブラリ内に存在します。ライブラリ `libsocket.so` は、他のシステムサービスライブラリとともに `/usr/lib` にあります。`libsocket.so` は動的リンクに使用されます。

ソケットタイプ

ソケットタイプには、ユーザーが認識できる通信プロパティを定義します。インターネットファミリソケットは、TCP/IP トランスポートプロトコルへのアクセスを提供します。インターネットファミリは、IPv6 と IPv4 の両方で通信できるソケットの場合、`AF_INET6` という値で識別されます。また、以前のアプリケーションとのソース互換、および IPv4 に対する raw アクセスを目的とした値 `AF_INET` もサポートされています。

次に、SunOS 環境がサポートする 3 つのタイプのソケットを示します。

- 「ストリームソケット」は、プロセスが TCP を使用して通信できるようにします。ストリームソケットは、信頼性の高い、順序付けされた、重複のない双方向データフローをレコード境界なしで提供します。コネクションが確立されたあと、これらのソケットからのデータの読み取り、およびこれらのソケットに対するデータの書き込みがバイトストリームとして行えます。ソケットタイプは SOCK_STREAM です。
- 「データグラムソケット」は、プロセスが UDP を使用して通信できるようにします。データグラムソケットは、メッセージの双方向フローをサポートします。データグラムソケット側のプロセスは、送信シーケンスから順序を変えてメッセージを受信できます。データグラムソケット側のプロセスはまた、重複したメッセージも受信できます。データ内のレコード境界は保持されます。ソケットタイプは SOCK_DGRAM です。
- 「raw ソケット」は、ICMP へのアクセスを提供します。このタイプのソケットは、通常、データグラム型ですが、実際の特性はプロトコルが提供するインタフェースに依存します。raw ソケットは、ほとんどのアプリケーションには使用されません。raw ソケットは、新しい通信プロトコルの開発をサポートしたり、既存プロトコルの難解な機能にアクセスしたりするために提供されています。raw ソケットを使用できるのは、スーパーユーザープロセスだけです。ソケットタイプは SOCK_RAW です。

詳細については、131 ページの「特定のプロトコルの選択」を参照してください。

インタフェースセット

SunOS 5.9 プラットフォームは 2 つのソケットインタフェースセットを提供します。BSD ソケットインタフェース (SunOS バージョン 5.7 およびそれ以降のリリースで提供される) と XNS 5 (Unix98) ソケットインタフェースです。XNS 5 インタフェースは、BSD インタフェースとわずかに異なります。

XNS 5 ソケットインタフェースについては、次のマニュアルページを参照してください。

- `accept`(3XNET)
- `bind`(3XNET)
- `connect`(3XNET)
- `endhostent`(3XNET)
- `endnetent`(3XNET)
- `endprotoent`(3XNET)
- `endservent`(3XNET)
- `gethostbyaddr`(3XNET)
- `gethostbyname`(3XNET)
- `gethostent`(3XNET)
- `gethostname`(3XNET)
- `getnetbyaddr`(3XNET)
- `getnetbyname`(3XNET)

- getnetent(3XNET)
- getpeername(3XNET)
- getprotobyname(3XNET)
- getprotobynumber(3XNET)
- getprotoent(3XNET)
- getservbyname(3XNET)
- getservbyport(3XNET)
- getservent(3XNET)
- getsockname(3XNET)
- getsockopt(3XNET)
- htonl(3XNET)
- htons(3XNET)
- inet_addr(3XNET)
- inet_lnaof(3XNET)
- inet_makeaddr(3XNET)
- inet_netof(3XNET)
- inet_network(3XNET)
- inet_ntoa(3XNET)
- listen(3XNET)
- ntohl(3XNET)
- ntohs(3XNET)
- recv(3XNET)
- recvfrom(3XNET)
- recvmsg(3XNET)
- send(3XNET)
- sendmsg(3XNET)
- sendto(3XNET)
- sethostent(3XNET)
- setnetent(3XNET)
- setprotoent(3XNET)
- setservent(3XNET)
- setsockopt(3XNET)
- shutdown(3XNET)
- socket(3XNET)
- socketpair(3XNET)

従来の BSD ソケットの動作については、対応する 3N のマニュアルページを参照してください。さらに、マニュアルページのセクション 3N には、次のような新しいインタフェースが追加されました。

- freeaddrinfo(3SOCKET)
- freehostent(3SOCKET)
- getaddrinfo(3SOCKET)
- getipnodebyaddr(3SOCKET)
- getipnodebyname(3SOCKET)
- getnameinfo(3SOCKET)
- inet_ntop(3SOCKET)
- inet_pton(3SOCKET)

XNS 5 (Unix98) ソケットインタフェースを使用するアプリケーションを構築する方法については、standards(5) のマニュアルページを参照してください。

ソケットの基本的な使用

この節では、基本的なソケットインタフェースの使用について説明します。

ソケットの作成

socket(3SOCKET) 呼び出しは、指定されたファミリに指定されたタイプのソケットを作成します。

```
s = socket(family, type, protocol);
```

プロトコルが指定されない場合、システムは要求されたソケットタイプをサポートするプロトコルを選択します。ソケットハンドルが返されます。ソケットハンドルはファイル記述子です。

ファミリは、sys/socket.h に定義されている定数の 1 つで指定します。AF_ *suite* という名前の定数は、名前を解釈するときに使用されるアドレス形式を指定します。

AF_APPLETALK	Apple Computer, Inc. の Appletalk ネットワーク
AF_INET6	IPv6 と IPv4 用のインターネットファミリ
AF_INET	IPv4 専用のインターネットファミリ
AF_PUP	Xerox Corporation の PUP インターネット
AF_UNIX	UNIX ファイルシステム

ソケットタイプは、sys/socket.h で定義されています。ソケットタイプの SOCK_STREAM、SOCK_DGRAM、または SOCK_RAW は、AF_INET6、AF_INET、および AF_UNIX でサポートされます。インターネットファミリでストリームソケットを作成する例です。

```
s = socket(AF_INET6, SOCK_STREAM, 0);
```

この呼び出しの結果、ストリームソケットが作成されます。(このストリームソケットでは) TCP プロトコルが基本的な通信を提供します。ほとんどの場合、*protocol* 引数はデフォルトの 0 に設定します。125 ページの「ソケットの拡張機能」で説明しているように、デフォルト以外のプロトコルも指定できます。

ローカル名のバインド

ソケットは、その作成時には名前がありません。アドレスがソケットにバインドされるまで、リモートプロセスはソケットを参照できません。通信プロセスは、アドレスを介して接続されます。インターネットファミリでは、コネクションはローカルアドレス、リモートアドレス、ローカルポート、およびリモートポートから構成されます。順番が重複しているセット、たとえば `protocol`、`local address`、`local port`、`foreign address`、`foreign port` は指定できません。ほとんどのファミリでは、コネクションは一意である必要があります。

`bind`(3SOCKET) インタフェースを使用すると、プロセスはソケットのローカルアドレスを指定できます。このインタフェースは `local address`、`local port` というセットになります。`connect`(3SOCKET) と `accept`(3SOCKET) は、アドレス組のリモート側を固定することにより、ソケットの関連付けを完了します。`bind`(3SOCKET) 呼び出しは次のように使用します。

`bind` (*s, name, namelen*);

s は、ソケットハンドルです。バインド名は、バイト文字列で、サポートするプロトコル (複数も可) がこれを解釈します。インターネットファミリ名には、インターネットアドレスとポート番号が含まれます。

次の例では、インターネットアドレスのバインドを示します。

```
#include <sys/types.h>
#include <netinet/in.h>
...
        struct sockaddr_in6 sin6;
...
        s = socket(AF_INET6, SOCK_STREAM, 0);
        bzero (&sin6, sizeof (sin6));
        sin6.sin6_family = AF_INET6;
        sin6.sin6_addr.s6_addr = in6addr_arg;
        sin6.sin6_port = htons(MYPORT);
        bind(s, (struct sockaddr *) &sin6, sizeof sin6);
```

アドレス `sin6` の内容は、インターネットアドレスのバインドについて説明した 131 ページの「アドレスのバインド」で説明しています。

コネクションの確立

コネクションの確立は、通常、クライアントの役割を果たすプロセスと、サーバーの役割を果たすプロセスによって非同期で行われます。サーバーは、サービスに関連付けられた既知のアドレスにソケットをバインドし、コネクション要求のためにソケットをブロックします。これで、無関係のプロセスがサーバーに接続できます。クライ

アントは、サーバーのソケットへのコネクションを起動することでサーバーにサービスを要求します。クライアント側では、connect(3SOCKET) 呼び出しでコネクションを起動します。インターネットファミリの場合、このコネクションは次のようになります。

```
struct sockaddr_in6 server;
 ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

接続呼び出しの時点でクライアントのソケットがバインドされていない場合、システムは自動的に名前を選択し、ソケットにバインドします。詳細は、131 ページの「アドレスのバインド」を参照してください。これは、クライアントのソケットにローカルアドレスを バインドする一般的な方法です。

クライアントのコネクションを受信するには、サーバーはそのソケットをバインドした後に 2 つの処理を行う必要があります。まず、待ち行列に入れることができるコネクション要求の数を示し、続いてコネクションを受け入れます。

```
struct sockaddr_in6 from;
 ...
listen(s, 5);                 /* 5 つのコネクション待ち行列を許可する */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

ソケットハンドル *s* は、コネクション要求の送信先であるアドレスにバインドされるソケットです。listen(3SOCKET) の 2 つ目のパラメータは、待ち行列に入れることができる未処理のコネクションの最大数を指定します。from は、クライアントのアドレスを指定する構造体です。場合によって NULL ポインタが渡されます。*fromlen* は構造体の長さです。UNIX ファミリでは、 from は struct sockaddr_un として宣言されます。

accept(3SOCKET) ルーチンは通常、プロセスをブロックします。accept (3SOCKET) は、要求しているクライアントに接続される新しいソケット記述子を返します。*fromlen* の値は、アドレスの実際のサイズに変更されます。

サーバーは、特定のアドレスからのみコネクションを受け入れますが、これを表示することはできません。サーバーは accept (3SOCKET が返した from アドレスを確認し、受け入れ不可能なクライアントとのコネクションを閉じることができます。サーバーは、複数のソケット上のコネクションを受け入れることも、あるいは accept (3SOCKET) 呼び出しのブロックを避けることもできます。これらの手法については、125 ページの「ソケットの拡張機能」で説明しています。

コネクションエラー

コネクションが失敗した場合、エラーが返されますが、システムがバインドしたアドレスは残ります。コネクションが成功した場合、ソケットがサーバーに関連付けられ、データ転送を開始できます。

次の表に、コネクションが失敗したときに返される一般的なエラーの一覧を示します。

表 **6–1** ソケットコネクションエラー

ソケットエラー	エラーの説明
ENOBUFS	呼び出しをサポートするためのメモリーが足りない
EPROTONOSUPPORT	不明なプロトコルの要求
EPROTOTYPE	サポートされないソケットタイプの要求
ETIMEDOUT	指定された時刻にコネクションが確立されていない。このエラーは、宛先ホストがダウンしているか、あるいはネットワーク内の障害で伝送が中断した場合に発生する
ECONNREFUSED	ホストがサービスを拒否した。このエラーは、要求されたアドレスにサーバープロセスが存在しない場合に発生する
ENETDOWN または EHOSTDOWN	これらのエラーは、基本通信インタフェースが配信する状態情報によって発生する
ENETUNREACH または EHOSTUNREACH	この操作エラーは、ネットワークまたはホストへの経路がないために発生する。この操作エラーはまた、中間ゲートウェイまたは切り替えノードが返す状態情報によっても発生する。返される状態情報が十分でないために、ダウンしているネットワークとダウンしているホストが区別できない場合もある

データ転送

この節では、データを送受信するためのインタフェースについて説明します。メッセージの送受信は、次のように通常の `read`(2) インタフェースと `write`(2) インタフェースを使用できます。

```
write(s, buf, sizeof buf);
read(s,  buf, sizeof buf);
```

また、次のように `send`(3SOCKET) と `recv`(3SOCKET) も使用できます。

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

`send`(3SOCKET) と `recv`(3SOCKET) は `read`(2) と `write`(2) に非常によく似ていますが、`flags` 引数が重要です。`flags` 引数 (`sys/socket.h` で定義) は 0 以外の値として、次のうちの 1 つまたは複数を指定できます。

`MSG_OOB` 帯域外データを送受信する

`MSG_PEEK` 読み取らずにデータの確認だけを行う

`MSG_DONTROUTE` パケットの経路を指定せずにデータを送信する

帯域外データは、ストリームソケット固有のものです。recv(3SOCKET) 呼び出しで MSG_PEEK を指定した場合、存在するすべてのデータがユーザーに返されますが、データは読み取られていないものとして扱われます。次に、ソケット上で read(2) または recv(3SOCKET) を呼び出すと、同じデータが返されます。発信パケットに適用されるパケット経路を指定せずにデータを送信するオプションは現在、経路制御テーブルの管理プロセスだけに使用されています。

ソケットを閉じる

SOCK_STREAM ソケットは close(2) インタフェース呼び出しで破棄できます。close(2) の後でも確実な配信が見込まれるソケットの待ち行列にデータが入っている場合、プロトコルは引き続きデータを転送しようとします。期限が来てもデータが配信されない場合、データは破棄されます。

shutdown(3SOCKET) は SOCK_STREAM ソケットを正常に閉じ、両方のプロセスで送信が行われなくなっていることを認識できます。この呼び出しの形式は次のとおりです。

```
shutdown(s, how);
```

how は次のように定義されています。

0　それ以上の受信を許可しない

1　それ以上の送信を許可しない

2　それ以上の送受信を許可しない

ストリームソケットのコネクション

次の 2 つの例に、インターネットファミリのストリームコネクションの開始と受け入れを示します。

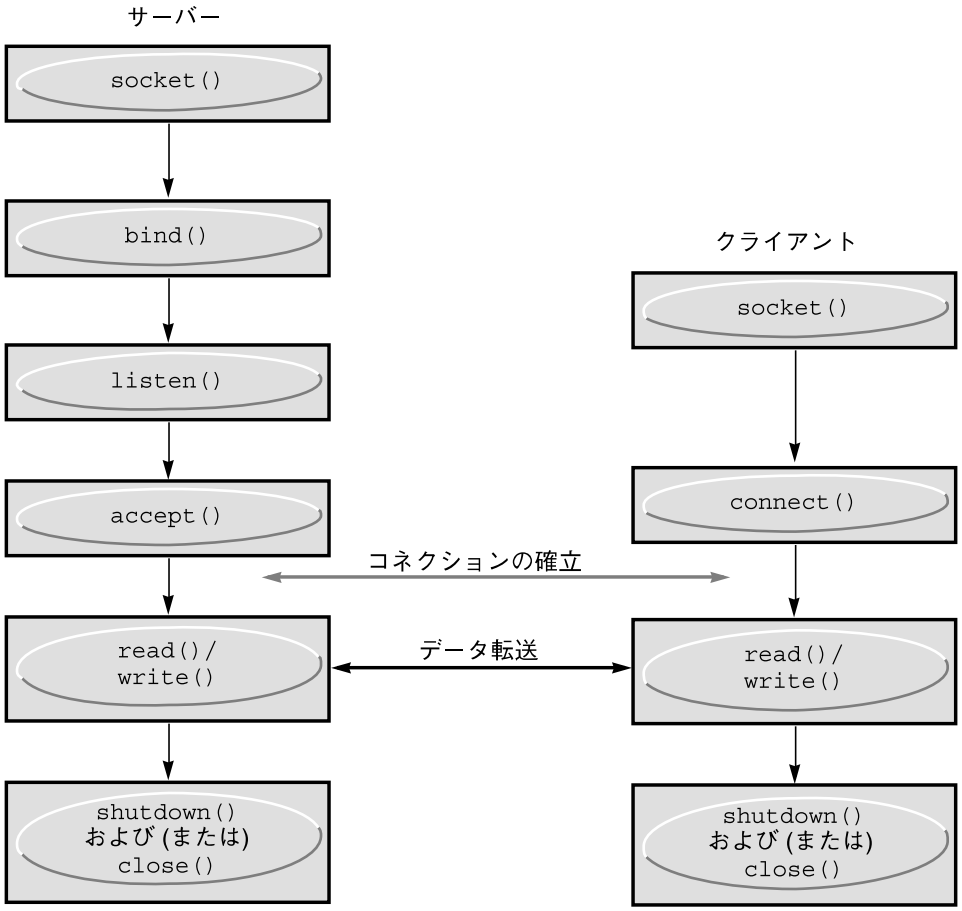


図 **6–1** ストリームソケットを使用したコネクション型の通信

次のプログラムはサーバーの例です。このサーバーは、ソケットを作成し、そのソケットに名前をバインドし、そして、ポート番号を表示します。このプログラムは `listen`(3SOCKET) を呼び出して、ソケットがコネクション要求を受け入れる用意ができていることをマークし、要求の待ち行列を初期化します。プログラムの残りの部分は無限ループです。ループの各パスは、新しいソケットを作成することによって新しいコネクションを受け入れ、待ち行列からそのコネクションを削除します。サーバーは、ソケットからのメッセージを読み取って表示し、ソケットを閉じます。`in6addr_any` の使用については、131 ページの「アドレスのバインド」で説明しています。

例 **6–1** インターネットストリームコネクションの受け入れ (サーバー)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

例 **6–1** インターネットストリームコネクションの受け入れ (サーバー)　(続き)

```
#include <netdb.h>
#include >stdio.h>
#define TRUE 1
/*
 * このプログラムは、ソケットを作成したあと無限ループを開始する。
 * ループごとにコネクションを受け入れ、そのコネクションからのデータを出力
 * するコネクションが遮断されるか、またはクライアントが コネクションを閉じた
 * 時点でプログラムは新しいコネクションを受け入れる
*/
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    /* ソケットを作成する */
    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
      perror("opening stream socket");
      exit(1);
   }
   /* ワイルドカードを使用してソケットをバインドする */
   bzero (&server, sizeof(server));
/* bzero (&sin6, sizeof (sin6)); */
   server.sin6_family = AF_INET6;
   server.sin6_addr = in6addr_any;
   server.sin6_port = 0;
   if (bind(sock, (struct sockaddr *) &server, sizeof server)
         == -1) {
      perror("binding stream socket");
      exit(1);
   }
   /* 割り当てられたポート番号を調べ、それを出力する */
   length = sizeof server;
   if (getsockname(sock,(struct sockaddr *) &server, &length)
         == -1) {
      perror("getting socket name");
      exit(1);
   }
   printf("Socket port #%d\n", ntohs(server.sin6_port));
   /* コネクションの受け入れを開始する */
   listen(sock, 5);
   do {
      msgsock = accept(sock,(struct sockaddr *) 0,(int *) 0);
      if (msgsock == -1)
         perror("accept");
      else do {
         memset(buf, 0, sizeof buf);
         if ((rval = read(msgsock,buf, 1024)) == -1)
            perror("reading stream message");
         if (rval == 0)
            printf("Ending connection\n");
```

例 **6–1** インターネットストリームコネクションの受け入れ (サーバー)　(続き)

```
            else
                /* データが出力可能であると想定する */
                printf("-->%s\n", buf);
        } while (rval > 0);
        close(msgsock);
    } while(TRUE);
    /*
     * このプログラムには無限ループが含まれるため、ソケットの
     * sock は明示的に閉じられることはない。ただし、プロセスが
     * 中断されるかまたは正常に終了する場合は自動的に閉じる
     */
    exit(0);
}
```

例 6–2 のクライアント側プログラムは、コネクションを開始するために、ストリームソケットを作成し、コネクション用のソケットのアドレスを指定して connect (3SOCKET) を呼び出します。宛先ソケットが存在し、要求が受け入れられる場合、コネクションは完了します。すると、プログラムはデータを送信できます。データは、メッセージ境界なしで順番に配信されます。コネクションは、一方のソケットが閉じられた時点で遮断されます。このプログラム内のデータ表現ルーチン (ntohl (3SOCKET)、ntohs(3SOCKET)、htons(3SOCKET)、および htonl(3XNET) など) の詳細については、byteorder(3SOCKET) のマニュアルページを参照してください。

例 **6–2** インターネットファミリのストリームコネクション(クライアント)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "Half a league, half a league . . ."
/*
 * このプログラムはソケットを作成し、コマンド行で指定されるソケットを
 * 使用してコネクションを開始する。このコネクションでいくつかのデータが
 * 送信されたあとソケットが閉じられコネクションが終了する
 * コマンドの形式 : streamwrite hostname portnumber
 * 使用法 : pgm host part
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock, errnum, h_addr_index;
    struct sockaddr_in6 server;
    struct hostent *hp;
    char buf[1024];
    /* ソケットを作成する */
    sock = socket( AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
```

例 6–2 インターネットファミリのストリームコネクション(クライアント)　(続き)

```
        exit(1);
    }
    /* コマンド行で指定される名前を使用してソケットを接続する */
    bzero (&server, sizeof (server));
    server.sin6_family = AF_INET6;
    hp = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &errnum);
/*
 * getipnodebyname が指定されたホストのネットワークアドレスを含む
 * 構造体を返す
 */
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    h_addr_index = 0;
    while (hp->h_addr_list[h_addr_index] != NULL) {
        bcopy(hp->h_addr_list[h_addr_index], &server.sin6_addr,
                    hp->h_length);
        server.sin6_port = htons(atoi(argv[2]));
        if (connect(sock, (struct sockaddr *) $server,
                    sizeof (server)) == -1) {
            if (hp->h_addr_list[++h_addr_index] != NULL) {
                /* 次のアドレスを試みる */
                continue;
            }
            perror("connecting stream socket");
            freehostent(hp);
            exit(1);
        }
        break;
    }
    freehostent(hp);
    if (write( sock, DATA, sizeof DATA) == -1)
        perror("writing on stream socket");
    close(sock);
    freehostent (hp);
    exit(0);
}
```

入出力の多重化

要求は、複数のソケットまたはファイルの間で多重化できます。多重化を行うには、`select`(3C) を使用します。

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
 ...
fd_set readmask, writemask, exceptmask;
```

```
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

select(3C) の最初の引数は、続く 3 つの引数が示すリスト内のファイル記述子の数です。

select(3C) の 2 つ目、3 つ目、4 つ目の引数は、3 つのファイル記述子セットを指します。つまり、 読み取りを行う記述子セット、書き込みを行うセット、および例外条件が認められるセットです。帯域外データは、唯一の例外条件です。これらのポインタはどれも、適切にキャストされた NULL として指定できます。各セットは、ロング整数ビットマスクの配列を含む構造体です。配列のサイズは FD_SETSIZE (select.h で定義) で設定します。配列には、各 FD_SETSIZE ファイル記述子のための 1 ビットを保持するだけの長さがあります。

マクロ FD_SET (*fd*, &*mask*) はセット mask 内のファイル記述子 *fd* を追加し、マクロ FD_CLR (*fd*, &*mask*) はこの記述子を削除します。セット mask は使用前に 0 にする必要があり、マクロ FD_ZERO (&*mask*) はセット mask をクリアします。

select(3C) に 5 つ目の引数を使用すると、タイムアウト値を指定できます。timeout ポインタが NULL の場合、ファイル記述子が選択できるようになるまで、または、シグナルが受信されるまで、select(3C) はブロックされます。timeout 内のフィールドが 0 に設定されると、select(3C) はすぐにポーリングして返されます。

select(3C) は通常、選択されたファイル記述子の数を返しますが、タイムアウト期限が過ぎていた場合は 0 を返します。エラーまたは割り込みが発生した場合、select(3C) ルーチンは、*errno* にエラー番号を指定し、ファイル記述子マスクを変更せずに、−1 を返します。成功した場合に返される 3 つのセットは読み取り可能なファイル記述子、書き込み可能なファイル記述子、または例外条件が保留されたファイル記述子を示します。

FD_ISSET (*fd*, &*mask*) マクロを使用して、選択マスク内のファイルの記述子の状態をテストしてください。セット mask 内に *fd* が存在する場合、このマクロは 0 以外の値を返します。それ以外の場合、このマクロは 0 を返します。ソケット上の待ち行列に入っているコネクション要求を確認するには、select(3C) を使用し、続いて、読み取りセット上で FD_ISSET (*fd*, &*mask*) マクロを使用します。

次の例は、読み取り用のリスニング (待機) ソケット上で select(3C) を使用することによって、accept(3SOCKET) 呼び出しでいつ新しいコネクションをピックアップできるかどうかタイミングを判定する方法を示します。このプログラムは、コネクション要求を受け入れ、データを読み取り、単一のソケットで切断します。

例 **6–3** select(3C) を使用して保留状態のコネクションを確認する

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time/h>
#include <netinet/in.h>
#include <netdb.h>
```

例 **6–3** select(3C) を使用して保留状態のコネクションを確認する　(続き)

```
#include <stdio.h>
#define TRUE 1
/*
 * このプログラムは accept を呼び出す前に、select を使用して
 * 他のユーザーがコネクションを試みていないかを確認する
 */
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    fd_set ready;
    struct timeval to;
    /* ソケットを開き、そのソケットを以前の例と同様にバインドする */
    /* コネクションの受け入れを開始する */
    listen(sock, 5);
    do {
        FD_ZERO(&ready);
        FD_SET(sock, &ready);
        to.tv_sec = 5;
        to.tv_usec = 0;
        if (select(sock + 1, &ready, (fd_set *)0,
                   (fd_set *)0, &to) == -1) {
            perror("select");
            continue;
        }
        if (FD_ISSET(sock, &ready)) {
            msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
            if (msgsock == -1)
                perror("accept");
            else do {
                memset(buf, 0, sizeof buf);
                if ((rval = read(msgsock, buf, 1024)) == -1)
                    perror("reading stream message");
                else if (rval == 0)
                    printf("Ending connection\n");
                else
                    printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
        } else
            printf("Do something else\n");
        } while (TRUE);
    exit(0);
}
```

以前のバージョンの select(3C) ルーチンでは、引数は fd_sets へのポインタではなく、整数へのポインタでした。ファイル記述子の数が整数内のビット数よりも小さい場合は、現在でもこのような呼び出しを使用できます。

select(3C) ルーチンは同期多重スキーマを提供します。SIGIO シグナルと SIGURG シグナル (125 ページの「ソケットの拡張機能」を参照) によって、出力の完了、入力の有効性、および例外条件の非同期通知を指定できます。

データグラムソケット

データグラムソケットは、コネクションの確立を要求せずに、対称型データ交換インタフェースを提供します。各メッセージには宛先アドレスが含まれます。次の図では、サーバーとクライアント間の通信の流れを示します。

次の図において、サーバー側の bind(3SOCKET) 手順は省略できます。

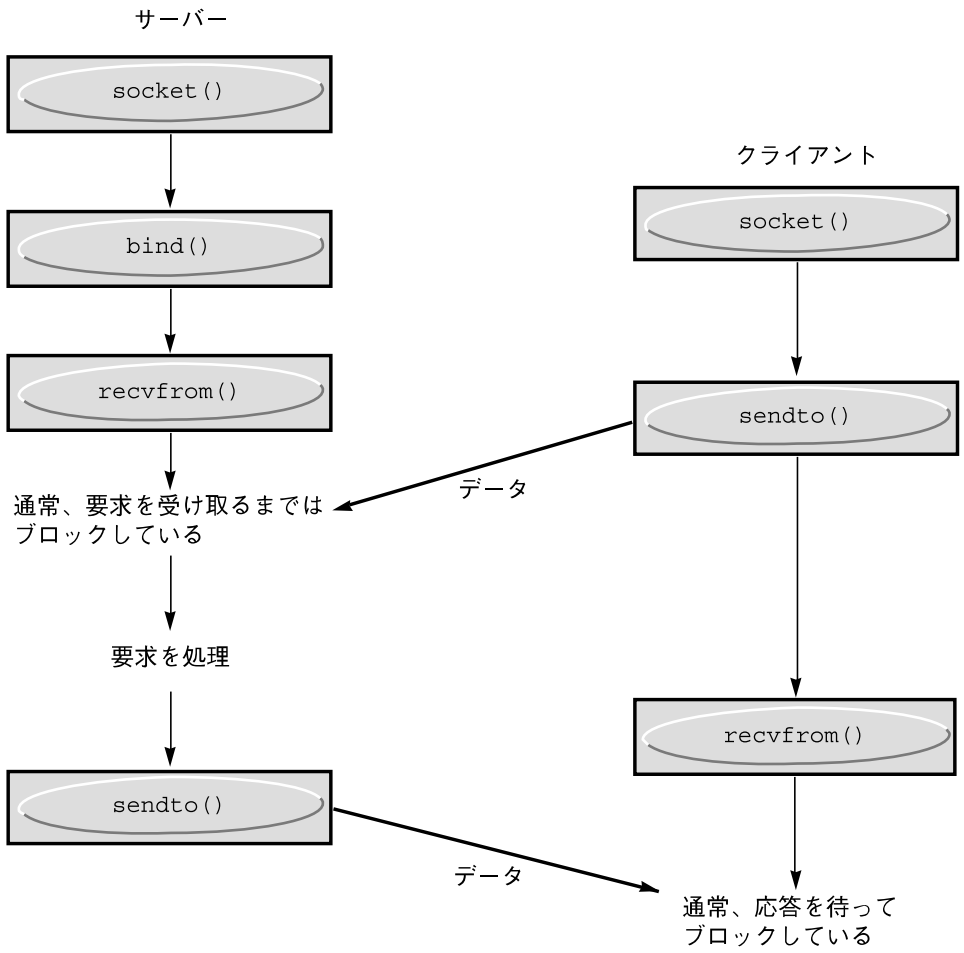


図 **6–2** データグラムソケットを使用したコネクションレス型の通信

101 ページの「ソケットの作成」で説明しているように、データグラムソケットを作成します。特定のローカルアドレスが必要な場合、`bind`(3SOCKET) 操作を最初のデータ伝送よりも先に行う必要があります。それ以外の場合、データが最初に送信される際にシステムがローカルアドレスまたはポートを設定します。データを送信するには、`sendto`(3SOCKET) を使用します。

```
sendto(s, buf, buflen, flags, (struct sockaddr *) &to, tolen);
```

s、*buf*、*buflen*、および *flags* パラメータは、コネクション型のソケットの場合と同じです。*to* と *tolen* の値は、意図するメッセージ受信者のアドレスを示します。ローカルにエラー条件 (到達できないネットワークなど) が検出されると、`–1` が返され、*errno* にエラー番号が設定されます。

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *) &from, &fromlen);
```

データグラムソケット上でメッセージを受信するには、recvfrom(3SOCKET) を使用します。呼び出す前、*fromlen* には *from* バッファーのサイズが設定されます。fromlen にはデータグラムの配信元であるアドレスのサイズが設定されて返されます。

データグラムソケットは connect(3SOCKET) 呼び出しを使用して、ソケットを特定の宛先アドレスに関連付けることもできます。これにより、ソケットは send (3SOCKET) 呼び出しを使用できます。宛先アドレスが明示的に指定されていないソケット上に送信されるデータはすべて、接続されたピアにアドレス指定されます。そして、そのピアから受信されるデータだけが配信されます。1 つのソケットに一度に接続できるのは、接続された 1 つのアドレスだけです。2 つ目の connect (3SOCKET) 呼び出しは、宛先アドレスを変更します。データグラムソケット上のコネクション要求は、すぐに返されます。システムは、ピアのアドレスを記録します。accept(3SOCKET) と listen(3SOCKET) はデータグラムソケットでは使用されません。

データグラムソケットが接続されている間、前の send(3SOCKET) 呼び出しからのエラーは非同期に返すことができます。ソケットはこれらのエラーを後続の操作で報告できます。また、getsockopt(3SOCKET) のオプションである SO_ERROR を使用して、エラー状態を問い合わすこともできます。

次のコードに、ソケットの作成、名前のバインド、ソケットへのメッセージ送信によって、インターネット呼び出しを送信する例を示します。

例 **6–4** インターネットファミリデータグラムの送信

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "The sea is calm, the tide is full . . ."
/*
 * ここで、コマンド行引数から取得する名前を持つ受信箱にデータ
 * グラムを送信する。
 * コマンド行の形式 : dgramsend hostname portnumber
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock, errnum;
    struct sockaddr_in6 name;
    struct hostent *hp;
    /* 送信するソケットを作成する */
    sock = socket(AF_INET6,SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /*
     * 送信先ソケットのワイルドカードを使用しない構造名
```

例 **6–4** インターネットファミリデータグラムの送信　(続き)

```
     * getinodebyname は、指定されたホストのネットワークアドレスを
     * 含む構造体を返す。ポート番号はコマンド行から取得される
     */
    hp = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &errnum);
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    bzero (&name, sizeof (name));
    memcpy((char *) &name.sin6_addr, (char *) hp->h_addr,
       hp->h_length);
    name.sin6_family = AF_INET6;
    name.sin6_port = htons(atoi(argv[2]));
    /* メッセージを送信する */
    if (sendto(sock,DATA, sizeof DATA ,0,
        (struct sockaddr *) &name,sizeof name) == -1)
        perror("sending datagram message");
    close(sock);
    exit(0);
}
```

次のコードに、ソケットの作成、名前のバインド、ソケットからのメッセージ読み取りによって、インターネット呼び出しを読み取る例を示します。

例 **6–5** インターネットファミリデータグラムの読み取り

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
/*
 * このプログラムはデータグラムソケットを作成し、そのソケットに名前を
 * バインドし、続いてそのソケットから読み取ります
 */
 main()
{
    int sock, length;
    struct sockaddr_in6 name;
    char buf[1024];
    /* 読み取るソケットを作成する */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
   }
   /* ワイルドカードを使用して名前を作成する */
   bzero (&name, sizeof (name));
   name.sin6_family = AF_INET6;
   name.sin6_addr = in6addr_any;
   name.sin6_port = 0;
   if (bind (sock, (struct sockaddr *)&name, sizeof (name)) == -1) {
```

例 **6–5** インターネットファミリデータグラムの読み取り　(続き)

```
        perror("binding datagram socket");
        exit(1);
    }
    /* 割り当てられたポート値を確認し、それを出力する */
    length = sizeof(name);
    if (getsockname(sock,(struct sockaddr *) &name, &length)
          == -1)     {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs(name.sin6_port));
    /* ソケットから読み取りを行う */
    if (read(sock, buf, 1024) == -1 )
        perror("receiving datagram packet");
    /* データが出力可能であると想定する */
    printf("-->%s\n", buf);
    close(sock);
    exit(0);
}
```

標準ルーチン

この節では、ネットワークアドレスを検出したり、構築したりするルーチンについて説明します。特に明記しない限り、インターネットファミリだけに適用されます。

リモートホスト上のサービスを検出するには、クライアントとサーバーが通信を行う前にさまざまなレベルの割り当てを行う必要があります。サービスには、人が使用するための名前が付いています。サービス名とホスト名は、ネットワークアドレスに変換され、そのネットワークアドレスを使用してホストを検出し、ホストへの経路を指定します。割り当ての細部は、ネットワークアーキテクチャによって異なります。望ましいのは、ホストに名前が付けられることをネットワークが必要とせず、物理的なホストの位置の識別情報を保護できることです。

標準ルーチンは、ホスト名をネットワークアドレスに、ネットワーク名をネットワーク番号に、プロトコル名をプロトコル番号に、サービス名をポート番号にマッピングします。標準ルーチンはまた、サーバープロセスとの通信で使用するために適切なプロトコルも指定します。標準ルーチンを使用する場合は、ファイル `netdb.h` を組み込む必要があります。

ホスト名とサービス名

インタフェース getaddrinfo(3SOCKET)、getnameinfo(3SOCKET)、および freeaddrinfo(3SOCKE) を使用すると、ホスト上のサービスの名前とアドレスを簡単に変換できます。IPv6 の場合、getipnodebyname(3SOCKET) と getservbyname(3SOCKET) を呼び出す代わりに、これらのインタフェースを使用できます。同様に IPv4 でも、gethostbyname(3NSL) と getservbyname(3SOCKET) の代わりに、これらのインタフェースを使用できます。IPv6 アドレスと IPv4 アドレスは、どちらも透過的に処理されます。

getaddrinfo(3SOCKET) ルーチンは、指定されたホスト名とサービス名に結合アドレスとポート番号を返します。getaddrinfo(3SOCKET) が返す情報は動的に割り当てられるので、この情報は freeaddrinfo(3SOCKET) を使用して解放し、メモリーリークを回避する必要があります。getnameinfo(3SOCKET) は、指定されたアドレスとポート番号に関連付けられたホスト名とサービス名を返します。getaddrinfo (3SOCKET) と getnameinfo(3SOCKET) が返す EAI_xxx コードに基づくエラーメッセージを出力するには、gai_strerror(3SOCKET) を呼び出す必要があります。

次に、getaddrinfo(3SOCKET) の使用例を示します。

```
struct addrinfo         *res, *aip;
struct addrinfo         hints;
int                     sock = -1;
int                     error;

/* ホストアドレスを取得する。アドレスのタイプは任意 */
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(hostname, servicename, &hints, &res);
if (error != 0) {
  (void) fprintf(stderr, "getaddrinfo: %s for host %s service %s\n",
  gai_strerror(error), hostname, servicename);
 return (-1);
}
```

res が指す構造体の getaddrinfo(3SOCKET) が返す情報を処理したあと、freeaddrinfo(res) を使用して記憶領域を解放する必要があります。

次の例に示すように、getnameinfo(3SOCKET) ルーチンはエラーの原因を識別するときに特に便利です。

```
struct sockaddr_storage faddr;
int                     sock, new_sock, sock_opt;
socklen_t               faddrlen;
int                     error;
char                    hname[NI_MAXHOST];
char                    sname[NI_MAXSERV];

 ...
```

```
faddrlen = sizeof (faddr);
new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
if (new_sock == -1) {
    if (errno != EINTR && errno != ECONNABORTED) {
        perror("accept");
    }
    continue;
}
error = getnameinfo((struct sockaddr *)&faddr, faddrlen, hname,
            sizeof (hname), sname, sizeof (sname), 0);
if (error) {
  (void) fprintf(stderr, "getnameinfo: %s\n",
              gai_strerror(error));
} else {
    (void) printf("Connection from %s/%s\n", hname, sname);
}
```

ホスト名 – hostent

gethostent(3NSL) に定義されているように、インターネットホスト名からアドレスへのマッピングは hostent 構造体で表現されます。

```
struct hostent {
       char  *h_name;            /* ホストの正式名称 */
       char  **h_aliases;        /* 別名リスト */
       int   h_addrtype;         /* ホストアドレスのタイプ (AF_INET6 など) */
       int   h_length;           /* アドレスの長さ */
       char  **h_addr_list;      /* NULL で終わるアドレスのリスト */
};
/* 最初のアドレス、ネットワークバイトオーダー */
#define h_addr h_addr_list[0]
```

getipnodebyname(3SOCKET)	インターネットホスト名を hostent 構造体にマッピングする
getipnodebyaddr(3SOCKET)	インターネットホストアドレスを hostent 構造体にマッピングする
freehostent(3SOCKET)	hostent 構造体のメモリーを解放する
inet_ntop(3SOCKET)	インターネットホストアドレスを表示可能な文字列にマッピングする

このルーチンは、ホストの名前、その別名、アドレスタイプ、および NULL で終わる可変長アドレスのリストを含む hostent 構造体を返します。このアドレスリストが必要なのは、ホストが多くのアドレスを持つことができるためです。h_addr 定義は下位互換性のためであり、この定義は hostent 構造体のアドレスリストの最初のアドレスです。

ネットワーク名 – netent

ネットワーク名を番号に割り当て、netent 構造体を返すルーチンを示します。

```
/*
 * ネットワーク番号が 32 ビットに収まると想定します。
 */
struct netent {
   char     *n_name;      /* ネットの正式名称 */
   char     **n_aliases;  /* 別名リスト */
   int      n_addrtype;   /* ネットアドレスのタイプ */
   int      n_net;        /* ネット番号、ホストバイトオーダー */
};
```

getnetbyname(3SOCKET)、getnetbyaddr_r(3SOCKET)、および getnetent (3SOCKET) は、上記のホストルーチンに対応するネットワーク側のルーチンです。

プロトコル名 – protoent

protoent 構造体は、getprotobyname(3SOCKET)、getprotobynumber (3SOCKET)、および getprotoent(3SOCKET) で使用され、getprotoent (3SOCKET) で定義されるプロトコル名マッピングを定義します。

```
struct protoent {
   char     *p_name;          /* プロトコルの正式名称 */
   char     **p_aliases       /* 別名リスト */
   int      p_proto;          /* プロトコル番号 */
};
```

サービス名 – servent

インターネットファミリサービスは、特定の既知のポートに常駐し、特定のプロトコルを使用します。サービス名からポート番号へのマッピングは、getprotoent (3SOCKET) で定義されている servent 構造体で表現されます。

```
struct servent {
   char     *s_name;         /* サービスの正式名称 */
   char     **s_aliases;     /* 別名リスト */
   int      s_port;          /* パート番号、ネットワークバイトオーダー */
   char     *s_proto;        /* 使用するプロトコル */
};
```

getservbyname(3SOCKET) はサービス名と修飾プロトコル (省略可能) を servent 構造体にマッピングします。次の呼び出しは、

```
sp = getservbyname("telnet", (char *) 0);
```

任意のプロトコルを使用する Telnet サーバーのサービス仕様を返します。次の呼び出しは、

```
sp = getservbyname("telnet", "tcp");
```

TCP プロトコルを使用するTelnet サーバーを返します。getservbyport(3SOCKET) と getservent(3SOCKET) も提供されます。getservbyport(3SOCKET) には、getservbyname(3SOCKET) で使用されるインタフェースに似たインタフェースがあります。つまり、オプションのプロトコル名を指定して、ルックアップを修飾できます。

その他のルーチン

その他にも、名前とアドレスの操作を簡易化するルーチンはいくつかあります。次の表に、可変長のバイト列、およびバイトスワッピングのネットワークアドレスと値を要約します。

表 6–2 実行時ライブラリルーチン

インタフェース	機能説明
memcmp(3C)	バイト列を比較する。 同じ場合は 0、異なる場合は 0 以外の値を返す
memcpy(3C)	*s2* の *n* バイトを *s1* にコピーする
memset(3C)	base の最初の *n* バイトの領域に値 value を割り当てる
htonl(3SOCKET)	ホストからネットワークバイトオーダーへの 32 ビット量の変換
htons(3SOCKET)	ホストからネットワークバイトオーダーへの 16 ビット量の変換
ntohl(3SOCKET)	ネットワークからホストバイトオーダーへの 32 ビット量の変換
ntohs(3SOCKET)	ネットワークからホストバイトオーダーへの 16 ビット量の変換

バイトスワッピングルーチンを使用するのは、アドレスはネットワークオーダーで供給されるとオペレーティングシステムが考えるためです。一部のアーキテクチャでは、ホストバイトオーダーがネットワークバイトオーダーと異なるため、プログラムは必要に応じて値をバイトスワップする必要があります。そのため、ネットワークアドレスを返すルーチンは、ネットワークオーダーで返します。バイトスワッピング問題が発生するのは、ネットワークアドレスを解釈する場合だけです。たとえば、次のコードは TCP ポートまたは UDP ポートをフォーマットします。

```
printf("port number %d\n", ntohs(sp->s_port));
```

これらのルーチンを必要としないマシンでは、アドレスは NULL マクロとして定義されます。

クライアントサーバープログラム

もっとも一般的な分散型アプリケーションは、クライアントサーバーモデルです。このスキーマでは、クライアントプロセスはサーバープロセスからのサービスを要求します。

代替スキーマとして、休止しているサーバープロセスを削除できるサービスサーバーがあります。たとえば、inetd(1M) というインターネットサービスデーモンです。inetd(1M) はさまざまなポートで待機しますが、起動時に構成ファイルを読み取ることによって使用するポートを決定します。inetd(1M) のサービスを受けるポートでコネクションが要求されると、inetd(1M) はクライアントにサービスを行うために適切なサーバーを生成します。クライアントは、そのコネクションで中間媒体が何らかの役割を果たすことは意識しません。inetd(1M) の詳細については、135 ページの「inetd デーモン」を参照してください。

ソケットとサービス

ほとんどのサーバーには、既知のインターネットポート番号または UNIX ファミリ名でアクセスします。既知の UNIX ファミリ名の例には、rlogin サービスがあります。例 6–6 に、リモートログインサーバーのメインループを示します。

DEBUG モードで動作していない限り、サーバーはその呼び出し元の制御端末との関連付けを解除します。

```
(void) close(0);
(void) close(1);
(void) close(2);
(void) open("/", O_RDONLY);
(void) dup2(0);
(void) dup2(0);
setsid();
```

関連付けを解除することによって、サーバーは制御端末のプロセスグループからシグナルを受信しません。制御端末との関連付けを解除したあと、サーバーはエラーレポートを制御端末に送信できません。したがって、サーバーは syslog(3C) でエラーを記録する必要があります。

サービスの定義を取得するために、サーバーは getaddrinfo(3SOCKET) を呼び出します。

```
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo(NULL, "rlogin", &hints, &aip);
```

aip に返される結果は、プログラムがサービス要求を待機するインターネットポートを定義します。標準のポート番号の一部は /usr/include/netinet/in.h で定義されています。

次に、サーバーはソケットを作成して、サービス要求を待機します。bind (3SOCKET) ルーチンを使用すると、サーバーは必ず指定された場所で待機します。リモートログインサーバーが待機するポート番号は制限されているため、サーバーはスーパーユーザーとして動作します。次のループに、サーバーのメインループ (本体) を示します。

例 **6–6** サーバーのメインループ

```
/* コネクション要求を待機する */
for (;;) {
    faddrlen = sizeof (faddr);
    new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
    if (new_sock == -1) {
        if (errno != EINTR && errno != ECONNABORTED) {
            perror("rlogind: accept");
        }
        continue;
    }
    if (fork() == 0) {
        close (sock);
        doit (new_sock, &faddr);
    }
    close (new_sock);
}
/*NOTREACHED*/
```

accept(3SOCKET) は、クライアントがサービスを要求するまでメッセージをブロックします。さらに、SIGCHLD などのシグナルによる割り込みを受けた場合、accept (3SOCKET) は失敗を示す値を返します。accept(3SOCKET) からの戻り値を調べて、エラーが発生している場合は syslog(3C) でエラーを記録します。

次に、サーバーは子プロセスをフォークし、リモートログインプロトコル処理の本体を呼び出します。コネクション要求を待ち行列に入れるために親プロセスが使用するソケットは、子プロセスで閉じられます。accept(3SOCKET) が作成したソケットは、親プロセスで閉じられます。クライアントのアドレスがサーバーアプリケーションの doit() ルーチンに渡され、クライアントが認証されます。

ソケットとクライアント

この節では、クライアントリモートログインプロセスで行われる処理について説明します。サーバー側と同様に、まずリモートログインのサービス定義の位置を確認します。

```
bzero(&hints, sizeof (hints));
    hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
    hints.ai_socktype = SOCK_STREAM;
```

```
        error = getaddrinfo(hostname, servicename, &hints, &res);
        if (error != 0) {
          (void) fprintf(stderr, "getaddrinfo: %s for host %s service %s\n",
                         gai_strerror(error), hostname, servicename);
          return (-1);
        }
```

getaddrinfo(3SOCKET) は、res にあるアドレスの一覧の先頭を返します。希望のアドレスを見つけるには、ソケットを作成し、一覧に返される各アドレスに接続して、動作するアドレスが見つかるまで繰り返します。

```
for (aip = res; aip != NULL; aip = aip->ai_next) {
        /*
         * ソケットを開く。アドレスタイプは、提供される
         * getaddrinfo() によって変わる。
         */
        sock = socket(aip->ai_family, aip->ai_socktype,
            aip->ai_protocol);
        if (sock == -1) {
            perror("socket");
            freeaddrinfo(res);
            return (-1);
        }

        /* ホストに接続する */
        if (connect(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
            perror("connect");
            (void) close(sock);
            sock = -1;
            continue;
        }
        break;
    }
```

ソケットが作成され、希望のサービスに接続されます。sock はバインド解除されているので、connect(3SOCKET) ルーチンは暗黙的に sock をバインドします。

コネクションレス型のサーバー

サービスの中にはデータグラムソケットを使用するものがあります。rwho(1) サービスは、LAN に接続されたホストについての状態情報を提供します。ネットワークトラフィックが重くなるため、in.rwhod(1M) は実行しないでください。rwho サービスは、特定のネットワークに接続されたすべてのホストに情報をブロードキャストします。rwho サービスは、データグラムソケットを使用する例の 1 つです。

rwho(1) サーバープロセスを実行するホスト上のユーザーは、ruptime(1) を使用して別のホストの現在の状態を取得できます。次の例に、典型的な出力例を示します。

例 **6–7** ruptime(1) プログラムの出力

```
itchy up 9:45, 5 users, load 1.15, 1.39, 1.31
scratchy up 2+12:04, 8 users, load 4.67, 5.13, 4.59
```

例 **6–7** ruptime(1) プログラムの出力　(続き)

```
click up 10:10, 0 users, load 0.27, 0.15, 0.14
clack up 2+06:28, 9 users, load 1.04, 1.20, 1.65
ezekiel up 25+09:48, 0 users, load 1.49, 1.43, 1.41
dandy 5+00:05, 0 users, load 1.51, 1.54, 1.56
peninsula down 0:24
wood down 17:04
carpediem down 16:09
chances up 2+15:57, 3 users, load 1.52, 1.81, 1.86
```

各ホストには、rwho(1) サーバープロセスによって状態情報が周期的にブロードキャスト送信されます。このサーバープロセスも状態情報を受信します。このサーバープロセスはまた、データベースを更新します。このデータベースは、各ホストの状態のために解釈されます。サーバーはそれぞれ個別に動作し、ローカルネットワークとそのブロードキャスト機能によってのみ結合されます。

大量のネットトラフィックが生成されるため、ブロードキャストを使用することは非効率的です。サービスが広範囲に渡り、頻繁に使用されない限り、周期的なブロードキャストに手間がかかり簡潔さが失われます。

次に、rwho(1) サーバープロセスの簡単な例を示します。このコードは、まず、ネットワーク上のほかのホストからブロードキャストされた状態情報を受信し、次に、自分が動作しているホストの状態情報を提供します。最初のタスクは、プログラムのメインループで行われます。rwho(1) ポートで受信したパケットが別の rwho(1) サーバープロセスから送信されたことを確認したあと、到着時刻を記録します。次に、パケットはホストの状態でファイルを更新します。一定の時間内にホストからの通信がない場合、データベースルーチンはホストが停止していると想定し、この情報を記録します。ホストが稼働している間にはサーバーが停止していることもあるので、このアプリケーションはよくエラーになります。

例 **6–8** rwho(1) サーバープロセス

```
main()
{
   ...
   sp = getservbyname("who", "udp");
   net = getnetbyname("localnet");
   sin.sin6_addr = inet_makeaddr(net->n_net, in6addr_any);
   sin.sin6_port = sp->s_port;
   ...
   s = socket(AF_INET6, SOCK_DGRAM, 0);
   ...
   on = 1;
   if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on)
         == -1) {
      syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
      exit(1);
   }
   bind(s, (struct sockaddr *) &sin, sizeof sin);
   ...
   signal(SIGALRM, onalrm);
```

例 **6–8** rwho(1) サーバープロセス　*(続き)*

```
   onalrm();
   while(1) {
      struct whod wd;
          int cc, whod, len = sizeof from;
      cc = recvfrom(s, (char *) &wd, sizeof(struct whod), 0,
         (struct sockaddr *) &from, &len);
      if (cc <= 0) {
      if (cc == -1 && errno != EINTR)
         syslog(LOG_ERR, "rwhod: recv: %m");
      continue;
      }
      if (from.sin6_port != sp->s_port) {
         syslog(LOG_ERR, "rwhod: %d: bad from port",
            ntohs(from.sin6_port));
         continue;
      }
      ...
      if (!verify( wd.wd_hostname)) {
         syslog(LOG_ERR, "rwhod: bad host name from %x",
            ntohl(from.sin6_addr.s6_addr));
         continue;
      }
      (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
      whod = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
      ...
      (void) time(&wd.wd_recvtime);
      (void) write(whod, (char *) &wd, cc);
      (void) close(whod);
   }
   exit(0);
}
```

2 つ目のサーバータスクは、そのホストの状態の供給です。このタスクでは、周期的にシステム状態情報を取得し、その情報をメッセージにパッケージ化し、このメッセージをローカルネットワーク上でブロードキャストして、ほかの rwho(1) サーバープロセスに知らせる必要があります。このタスクはタイマーで実行されます。このタスクはシグナルによって起動されます。

状態情報は、ローカルネットワーク上でブロードキャスト送信されます。ブロードキャストをサポートしないネットワークでは、マルチキャストを使用してください。

ソケットの拡張機能

分散型アプリケーションを構築する場合、通常は、これまでに説明したメカニズムで十分対応できます。この節では、拡張機能について説明します。

帯域外データ

ストリームソケットの抽象化には、帯域外データが含まれます。帯域外データは、接続されたストリームソケットペア間の論理的に独立した伝送チャネルです。帯域外データは通常データとは無関係に配信されます。帯域外データ機能が使用される場合、一度に 1 つ以上の帯域外メッセージが確実に配信されなければなりません。このメッセージには 1 バイト以上のデータを含むことができます。また、いつでも 1 つ以上のメッセージの配信を保留できます。

帯域内シグナリングでは、緊急データは通常データと一緒に順番どおりに配信され、メッセージは通常データストリームから抽出されます。抽出されたメッセージは個別に格納されます。したがって、ユーザーは中間のデータをバッファリングせずに、緊急データを順番どおりに受信するか、順不同で受信するかを選択できます。

MSG_PEEK を使用すると、帯域外データを先読みできます。ソケットにプロセスグループがある場合は、その存在がプロトコルに通知される時に SIGURG シグナルが生成されます。プロセスは適切な fcntl(2) 呼び出しを使用して、プロセスグループまたはプロセス ID が SIGURG を配信するように設定できます (SIGIO については、129 ページの「割り込み方式のソケット入出力」を参照)。複数のソケットに配信待ちの帯域外データがある場合は、例外状況用に select(3C) 呼び出し、どのソケットがこのようなデータを保留しているかを判断してください。

帯域外データが送信された位置のデータストリームには、論理マークが置かれます。リモートログインアプリケーションとリモートシェルアプリケーションは、この機能を使用してクライアントプロセスとサーバープロセス間にシグナルを伝達します。シグナルが受信された時点で、データストリームの論理マークまでのデータはすべて破棄されます。

帯域外メッセージデータを送信するには、MSG_OOB フラグを send(3SOCKET) または sendto(3SOCKET) に指定します。ただし、帯域外データを受信するには、MSG_OOB フラグを recvfrom(3SOCKET) または recv(3SOCKET) に指定します。ただし、帯域外データを順番どおりに取得する場合、MSG_OOB フラグは必要ありません。SIOCATMARK ioctl(2) は、読み取りポインタが現在、データストリーム内のマークを指しているかどうかを示します。

```
int yes;
ioctl(s, SIOCATMARK, &yes);
```

yes が 1 で返される場合、次の読み取りはマークのあとのデータを返します。yes が 1 でない場合は、帯域外データが到着したと想定して、次の読み取りは帯域外シグナルを送信する前にクライアントによって送信されたデータを提供します。割り込みシグナルまたは終了シグナルを受信したときに出力をフラッシュするリモートログインプロセス内のルーチンを以下に示します。このコードは通常データを破棄を示すマークまで読み取った後、帯域外バイトを読み取ります。

プロセスは、初めにマークまでを読み取らずに、帯域外データの読み取りまたは先読みを行うこともできます。基底のプロトコルが通常データと一緒に帯域内にある緊急データを配信するときに、その存在だけを前もって通知する場合、このようなデータにアクセスすることはより困難になります。このようなタイプのプロトコルの例としては、TCP (インターネットファミリにソケットストリームを提供するときに使用され

るプロトコル) があります。このようなプロトコルでは、MSG_OOB フラグを指定して recv(3SOCKET) を呼び出したときに、帯域外バイトが到着していないことがあります。このような場合、呼び出しはエラー EWOULDBLOCK を返します。また、入力バッファー内の帯域内データの量によっては、ピアはバッファーが空になるまで (通常のフロー制御によって) 緊急データを送信できなくなる場合があります。この場合、プロセスが待ち行列に入ったデータを十分に読み取って入力バッファーをクリアしてからでないと、ピアは緊急データを送信できません。

例 **6–9** 帯域外データの受信時における端末入出力のフラッシュ

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
        int out = FWRITE;
        char waste[BUFSIZ];
        int mark = 0;

        /* ローカル端末出力をフラッシュする */
        ioctl(1, TIOCFLUSH, (char *) &out);
        while(1) {
            if (ioctl(rem, SIOCATMARK, &mark) == -1) {
                perror("ioctl");
                break;
            }
            if (mark)
                break;
            (void) read(rem, waste, sizeof waste);
        }
        if (recv(rem, &mark, 1, MSG_OOB) == -1) {
            perror("recv");
            ...
        }
        ...
}
```

ソケットストリームのインライン (帯域内) にある緊急データの位置を保持する機能もあります。この機能は、ソケットレベルのオプションである SO_OOBINLINE として提供されます。使用法については、getsockopt(3SOCKET) のマニュアルページを参照してください。このソケットレベルのオプションを使用すると、緊急データの位置を保持できます。ただし、MSG_OOB フラグを指定しない場合、通常データストリームにおいてマークの直後にある緊急データが返されます。複数の緊急指示を受信するとマークは移動しますが、帯域外データが消失することはありません。

非ブロックソケット

一部のアプリケーションは、ブロックしないソケットを必要とします。たとえば、要求がすぐに完了できない場合、サーバーはエラーコードを返して、その要求を実行しないことがあります。このようなエラーが発生した場合、プロセスは要求が完了するまで待ち、結果として中断されます。このようなアプリケーションではソケットを作成および接続したあと、次の例に示すように、fcntl(2) 呼び出しを発行してソケットを非ブロックに設定します。

例 **6–10** 非ブロックソケットの設定

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL, 0) == -1)
        perror("fcntl F_GETFL");
        exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY) == -1)
        perror("fcntl F_SETFL, FNDELAY");
        exit(1);
}
...
```

非ブロックソケットで入出力を行う場合は、操作が正常にブロックする場合に発生する、errno.h 内のエラー EWOULDBLOCK を確認してください。accept(3SOCKET)、connect(3SOCKET)、send(3SOCKET)、recv(3SOCKET)、read(2)、および write(2) はすべて EWOULDBLOCK を返すことができます。send(3SOCKET) などの操作を完全には実行できないが、部分的な書き込みは可能である場合 (ストリームソケットを使用する場合など)、送信できるデータはすべて処理されます。そして、戻り値は実際に送信された量になります。

非同期ソケット入出力

複数の要求を同時に処理するアプリケーションでは、プロセス間の非同期通信が必要です。非同期ソケットは SOCK_STREAM タイプである必要があります。ソケットを非同期にするには、次に示すように、 fcntl(2) 呼び出しを実行します。

例 **6–11** ソケットを非同期にする

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
```

例 **6–11** ソケットを非同期にする　*(続き)*

```
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL ) == −1)
        perror("fcntl F_GETFL");
        exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY | FASYNC) == −1)
        perror("fcntl F_SETFL, FNDELAY | FASYNC");
        exit(1);
}
...
```

ソケットを初期化および接続して、非ブロックと非同期に設定したあと、通信はファイルを非同期で読み書きする場合のように行われます。データ転送を開始するには、send(3SOCKET)、write(2)、recv(3SOCKET)、または read(2) を使用します。データ転送を完了するには、シグナル (割り込み) 方式の入出力ルーチンを使用します (次の節を参照)。

割り込み方式のソケット入出力

SIGIO シグナルは、ソケット (任意のファイル記述子) がデータ転送を終了した時点をプロセスに通知します。 SIGIO を使用する手順は次のとおりです。

1. signal(3C) 呼び出しまたは sigvec(3UCB) 呼び出しを使用して、SIGIO シグナルハンドラを設定する。
2. fcntl(2) を使用してプロセス ID またはプロセスグループ ID を設定し、シグナルの経路をそれ自体のプロセス ID またはプロセスグループ ID に指定する。ソケットのデフォルトのプロセスグループはグループ 0。
3. ソケットを非同期に変換する (128 ページの「非同期ソケット入出力」を参照)。

次のコードに、特定のプロセスがあるソケットに対して要求を行うときに、保留中の要求の情報を受信できるようにする例を示します。SIGURG のハンドラを追加すると、このコードは SIGURG シグナルを受信する目的でも使用できます。

例 **6–12** 入出力要求の非同期通知

```
#include <fcntl.h>
#include <sys/file.h>
 ...
signal(SIGIO, io_handler);
/* SIGIO または SIGURG シグナルを受信するプロセスを s に設定する。*/
if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}
```

シグナルとプロセスグループ ID

SIGURG と SIGIO の場合、各ソケットにはプロセス番号とプロセスグループ ID があります。前述の例のとおり、これらの値は 0 に初期化されますが、F_SETOWN fcntl (2) コマンドを使用すると、その後でも定義し直すことができます。fcntl(2) の 3 番目の引数が正の場合、ソケットのプロセス ID を設定します。fcntl(2) の 3 番目の引数が負の場合、ソケットのプロセスグループ ID を設定します。SIGURG シグナルと SIGIO シグナルの受信側として許可されるのは、呼び出し側のプロセスだけです。同様に、fcntl(2)、F_GETOWN は、ソケットのプロセス番号を返します。

また、ioctl(2) を使用してソケットをユーザーのプロセスグループに割り当てても、SIGURG と SIGIO を受信できるように設定できます。

```
/* oobdata はルーチンを処理する帯域外データ */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
        perror("ioctl: SIOCSPGRP");
}
```

サーバープロセスで便利なシグナルとして、ほかに SIGCHLD が挙げられます。このシグナルは、任意の子プロセスがその状態を変更した場合にプロセスに配信されます。通常、サーバーはこのシグナルを使用して、明示的に終了を待機せずに、あるいは終了状態を周期的にポーリングせずに、終了した子プロセスの「リープ (取得)」を行います。たとえば、前述の例のリモートログインサーバーループは次のように拡張できます。

例 **6–13** SIGCHLD シグナル

```
int reaper();
...
sigset(SIGCHLD, reaper);
listen(f, 5);
while (1) {
        int g, len = sizeof from;
        g = accept(f, (struct sockaddr *) &from, &len);
        if (g < 0) {
            if (errno != EINTR)
                syslog(LOG_ERR, "rlogind: accept: %m");
            continue;
        }
        ...
}

#include <wait.h>

reaper()
{
        int options;
        int error;
        siginfo_t info;

        options = WNOHANG | WEXITED;
```

例 **6–13** SIGCHLD シグナル　*(続き)*

```
        bzero((char *) &info, sizeof(info));
        error = waitid(P_ALL, 0, &info, options);
}
```

親サーバープロセスがその子プロセスのリープに失敗する場合、ゾンビプロセスが生じます。

特定のプロトコルの選択

socket(3SOCKET) 呼び出しの 3 番目の引数が 0 の場合、socket(3SOCKET) は要求されたタイプである返されたソケットにデフォルトのプロトコルを使用するように選択します。通常はデフォルトプロトコルで十分であり、ほかの選択肢はありません。raw ソケットを使用して低レベルのプロトコルやハードウェアインタフェースと直接通信を行う場合は、プロトコルの引数で非多重化を設定してください。

raw ソケットをインターネットファミリで使用して新しいプロトコルを IP 上に実装すると、このソケットは必ず、指定されたプロトコルのパケットだけを受信します。特定のプロトコルを取得するには、プロトコルファミリで定義されているようにプロトコル番号を決定します。インターネットファミリの場合、116 ページの「標準ルーチン」で説明しているライブラリルーチンの 1 つ (getprotobyname(3SOCKET) など) を使用してください。

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET6, SOCK_STREAM, pp->p_proto);
```

getprotobyname を使用すると、ソケット s はストリームベースのコネクションを使用しますが、デフォルトの tcp ではなく、newtcp というプロトコルタイプを使用します。

アドレスのバインド

アドレスを指定するとき、TCP と UDP は次の 4 つの要素を使用します。

- ローカル IP アドレス
- ローカルポート番号
- 外部 IP アドレス
- 外部ポート番号

TCP では、これらの 4 つの組は一意である必要があります。UDP にはこのような要求はありません。ホストは複数のネットワークに常駐でき、ユーザーは割り当てられているポート番号に直接アクセスできません。したがって、ユーザープログラムは必

ずしもローカルアドレスとローカルポートに使用する適切な値を認識できるとは限りません。この問題を避けるため、アドレスの一部を指定せずにおき、必要に応じてシステムにこれらの部分を適切に割り当てることができます。これらの組の各部は、ソケット API のさまざまな部分によって指定できます。

bind(3SOCKET)	ローカルアドレスまたはローカルポート (あるいはこの両方)
connect(3SOCKET)	外部アドレスと外部ポート

accept(3SOCKET) 呼び出しは外部クライアントからコネクション情報を取得します。したがって、accept(3SOCKET) の呼び出し元が何も指定していなくても、ローカルアドレスとローカルポートをシステムに指定できます。外部アドレスと外部ポートが返されます。

listen(3SOCKET) を呼び出すと、ローカルポートが選択されます。ローカル情報を割り当てる bind(3SOCKET) を明示的に指定していない場合、listen(3SOCKET) は一時的なポート番号を割り当てます。

あるポート上に常駐するサービスがローカルアドレス情報を必要としない場合、そのサービスはそのポートに bind(3SOCKET) できます。このとき、ローカルアドレスは指定しないままにしておいてもかまいません。ローカルアドレスは、<netinet/in.h> に定数値を持つ変数 in6addr_any に設定されます。ローカルポートを固定する必要がない場合、listen(3SOCKET) を呼び出すと、ポートが選択されます。アドレス in6addr_any またはポート番号 0 を指定することを「ワイルドカード (を使用する)」と呼びます。 AF_INET の場合は、in6addr_any の代わりに INADDR_ANY を使用します。

ワイルドカードアドレスは、インターネットファミリにおけるローカルアドレスのバインドを簡易化します。次のコードは、getaddrinfo(3SOCKET) の呼び出しで返された特定のポート番号をソケットにバインドし、ローカルアドレスを指定しないままにしておく例です。

```
#include <sys/types.h>
#include <netinet/in.h>
...
    struct addrinfo        *aip;
...
    if (bind(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
        perror("bind");
        (void) close(sock);
        return (-1);
    }
```

ホスト上の各ネットワークインタフェースは、通常、一意の IP アドレスを持ちます。ワイルドカードローカルアドレスを持つソケットは、指定されたポート番号に宛てたメッセージを受信できます。ワイルドカードローカルアドレスを持つソケットはまた、ホストに割り当てられている可能性のあるアドレスに送信されたメッセージを受信できます。特定のネットワーク上のホストだけにサーバーとの接続を許可するために、サーバーは適切なネットワーク上のインタフェースのアドレスをバインドします。

同様に、ローカルポート番号を指定しないままにしておくと、システムがポート番号を選択します。たとえば、特定のローカルアドレスをソケットにバインドするが、ローカルポート番号は指定しないままにしておくには、次のように bind を使用します。

```
bzero (&sin, sizeof (sin));
(void) inet_pton (AF_INET6, ":ffff:127.0.0.1", sin.sin6_addr.s6_addr);
sin.sin6_family = AF_INET6;
sin.sin6_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

システムは、次の 2 つの基準でローカルポート番号を選択します。

- 1024 未満のインターネットポート番号 (IPPORT_RESERVED) は、特権ユーザー用に予約される。非特権ユーザーは、1024 を超える任意のインターネットポート番号を使用されるインターネットポート番号の最大値は 65535。
- 現在、ほかのソケットにバインドされていないポート番号

クライアントのポート番号と IP アドレスは accept(3SOCKET) または getpeername (3SOCKET) で確認します。

関連付けが 2 段階のプロセスで作成されるため、システムがポート番号を選択するために使用するアルゴリズムがアプリケーションに適さない場合もあります。たとえば、インターネットファイル転送プロトコルでは、データコネクションは常に同じローカルポートから実行する必要があると定めています。しかし、異なる外部ポートに接続することによって、関連付けの重複を避けることができます。この場合、前のデータコネクションのソケットが存在しているとき、システムは同じローカルアドレスとローカルポート番号をソケットにバインドすることを許可しません。

デフォルトのポート選択アルゴリズムを無効にするには、次に示すようにオプション呼び出しを行なってからアドレスをバインドする必要があります。

```
 ...
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

この呼び出しを行うと、すでに使用されているローカルアドレスをバインドできます。この呼び出しは一意性という条件に違反しません。なぜなら、同じローカルアドレスとローカルポートを持つ別のソケットが同じ外部アドレスと外部ポートを持たないことをシステムがコネクション時に検証するためです。関連付けがすでに存在する場合、エラー EADDRINUSE が返されます。

ゼロコピーとチェックサム負荷解除

SunOS 5.6 およびその互換バージョンでは、TCP/IP プロトコルスタックは、ゼロコピーと TCP チェックサム負荷解除という 2 つの新しい機能をサポートするように拡張されました。

- ゼロコピーは、仮想メモリー MMU の再マッピングと、書き込み時にコピーを行う手法を使用して、アプリケーションとカーネル空間の間でデータを移動します。
- チェックサム負荷解除は、特殊なハードウェアロジックにより TCP チェックサム計算の負荷を解除します。

ゼロコピーとチェックサム負荷解除は互いに機能的には依存していませんが、最高の性能を得るには連携して動作する必要があります。チェックサム負荷解除には、ネットワークインタフェースのハードウェアサポートが必要です。このハードウェアサポートがない場合、ゼロコピーは有効になりません。

ゼロコピーには、仮想メモリーページの再マッピングを適用する前に、アプリケーションがページ型のバッファーを供給することが必要です。負荷が高い書き込み時コピーの失敗を避けるには、アプリケーションは伝送側に大きな循環バッファーを使用する必要があります。一般的なバッファー割り当ては 16 の 8K バッファーです。

ソケットオプション

`setsockopt`(3SOCKET) と `getsockopt`(3SOCKET) を使用すると、ソケットのオプションを設定および取得できます。たとえば、送信バッファー空間または受信バッファー空間を変更できます。次に、呼び出しの一般的な書式を示します。

```
setsockopt(s, level, optname, optval, optlen);
```

および

```
getsockopt(s, level, optname, optval, optlen);
```

オペレーティングシステムはいつでもこれらの値を適切に調整できます。

次に、`setsockopt`(3SOCKET) 呼び出しと `getsockopt`(3SOCKET) 呼び出しの引数を示します。

s オプションの適用先であるソケット

level `sys/socket.h` 内の記号定数 `SOL_SOCKET` が示すプロトコルレベル (ソケットレベルなど) を指定する

optname オプションを指定する、`sys/socket.h` で定義されている記号定数

optval オプションの値を示す

optlen オプションの値の長さを示す

getsockopt(3SOCKET) の場合、*optlen* は値結果の引数です。初期状態時、*optlen* 引数は *optval* が示す記憶領域のサイズに設定されます。復帰時、*optlen* 引数は使用された記憶領域の長さに設定されます。

既存のソケットのタイプを判断する必要があるとき、プログラムは SO_TYPE ソケットオプションと getsockopt(3SOCKET) 呼び出しを使用して、inetd(1M) を起動する必要があります。

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

int type, size;

size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) <0) {
     ...
}
```

getsockopt(3SOCKET) のあと、type はソケットタイプの値 (sys/socket.h で定義) に設定されます。データグラムソケットの場合、type は SOCK_DGRAM です。

inetd デーモン

inetd(1M) デーモンは起動時に呼び出され、待機するサービスを /etc/inet/inetd.conf ファイルから取得します。デーモンは /etc/inet/inetd.conf ファイルに記述されている各サービスごとに 1 つのソケットを作成し、各ソケットに適切なポート番号の割り当てを行います。inetd(1M) についての詳細は、マニュアルページを参照してください。

inetd(1M) デーモンは各ソケットをポーリングして、そのソケットに対応するサービスへのコネクション要求を待機します。SOCK_STREAM タイプのソケットの場合、inetd(1M) は待機ソケット上で受け入れ (accept(3SOCKET))、フォークし (fork(2))、新しいソケットをファイル記述子 0 および 1 (stdin および stdout) に複製し (dup(2)) 、ほかの開いているファイル記述子を閉じて、適切なサーバーを実行します (exec(2))。

inetd(1M) を使用する主な利点は、使用していないサービスがシステムのリソースを消費しない点にあります。また、コネクションの確立に関する処理の大部分を inetd (1M) が行う点も大きな利点の 1 つです。inetd(1M) によって起動されたサーバーのソケットはファイル記述子 0 と 1 上のクライアントに接続されます。したがって、サーバーはすぐに、読み取り、書き込み、送信、または受信を行うことができます。fflush(3C) を適宜使用する限り、サーバーはバッファリングされた入出力を stdio の規約に従って使用できます。

getpeername(3SOCKET) ルーチンはソケットに接続されたピア (プロセス) のアドレスを返します。このルーチンは、inetd(1M) によって起動されたサーバーで使用すると便利です。たとえば、このルーチンを使用すると、クライアントの IPv6 アドレスを表現するときに使用される fec0::56:a00:20ff:fe7d:3dd2 のようなインターネットアドレスを記録できます。次に、inetd(1M) サーバーが使用するコードの例を示します。

```
struct sockaddr_storage name;
int namelen = sizeof (name);
char abuf[INET6_ADDRSTRLEN];
struct in6_addr addr6;
struct in_addr addr;

if (getpeername(fd, (struct sockaddr *)&name, &namelen) == -1) {
    perror("getpeername");
    exit(1);
} else {
    addr = ((struct sockaddr_in *)&name)->sin_addr;
    addr6 = ((struct sockaddr_in6 *)&name)->sin6_addr;
    if (name.ss_family == AF_INET) {
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6 &&
               IN6_IS_ADDR_V4MAPPED(&addr6)) {
            /* これは IPv4 でマップされた IPv6 アドレス */
            IN6_MAPPED_TO_IN(&addr6, &addr);
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6) {
            (void) inet_ntop(AF_INET6, &addr6, abuf, sizeof (abuf));

    }
    syslog("Connection from %s\n", abuf);
}
```

ブロードキャストとネットワーク構成の判断

ブロードキャストは IPv6 ではサポートされません。ブロードキャストがサポートされるのは IPv4 のみです。

データグラムソケットにより送信されたメッセージは、接続されているネットワークのすべてのホストに届くようにブロードキャストを行うことができます。システムはブロードキャストのシミュレーションをソフトウェアで行わないため、ネットワークがブロードキャストをサポートする必要があります。ブロードキャストメッセージを使用すると、ネットワーク上のすべてのホストがブロードキャストメッセージをサービスする必要があるので、ブロードキャストメッセージはネットワークに大きな負荷をかける可能性があります。ブロードキャストは主に次の 2 つの目的に使用されます。

- アドレスが不明なローカルネットワーク上の資源の検索
- アクセス可能なすべての隣接ホストに情報を送信する必要がある機能

ブロードキャストメッセージを送信するには、次のようにインターネットデータグラムソケットを作成します。

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

次に、ポート番号をソケットにバインドします。

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

ネットワークのブロードキャストアドレスに送信することにより、データグラムは 1 つのネットワーク上のみでブロードキャストを行うことができます。また、netinet/in.h 内で定義されている特別なアドレス INADDR_BROADCAST に送信することにより、接続されているすべてのネットワークに対して、データグラムのブロードキャストを行うことができます。

システムは、システム上のネットワークインタフェースについての情報の数を判断するメカニズムを提供します。この情報には、IP アドレスおよびブロードキャストアドレスが含まれます。SIOCGIFCONF ioctl(2) 呼び出しはホストのインタフェース構成を単一の ifconf 構造体で返します。この構造体には ifreq 構造体の配列が含まれます。ifreq 構造体は、ホストに接続されているすべてのネットワークインタフェースがサポートするアドレスファミリごとに 1 つずつ存在します。

次の例では、net/if.h で定義されている ifreq 構造体を示します。

例 **6–14** net/if.h ヘッダーファイル

```
struct ifreq {
#define IFNAMSIZ 16
char ifr_name[IFNAMSIZ]; /* たとえば名前が "en0" */
union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        char ifru_oname[IFNAMSIZ]; /* 名前の場合、その他 */
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_metric;
        char ifru_data[1]; /* インタフェース依存データ */
        char ifru_enaddr[6];
} ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr
#define ifr_dstaddr ifr_ifru.ifru_dstaddr
#define ifr_oname ifr_ifru.ifru_oname
#define ifr_broadaddr ifr_ifru.ifru_broadaddr
#define ifr_flags ifr_ifru.ifru_flags
#define ifr_metric ifr_ifru.ifru_metric
#define ifr_data ifr_ifru.ifru_data
#define ifr_enaddr ifr_ifru.ifru_enaddr
};
```

インタフェース構成を取得する呼び出しは以下の通りです。

```
/*
 * インタフェースの数を検索するため SIOCGIFNUM ioctl を実行。
 *
 * 発見されたインタフェースの数に相当する空間を割り当て。
 *
 * 割り当てられたバッファーに対し SIOCGIFCONF を実行。
 *
 */
if (ioctl(s, SIOCGIFNUM, (char *)&numifs) == -1) {
        numifs = MAXIFS;
}
bufsize = numifs * sizeof(struct ifreq);
reqbuf = (struct ifreq *)malloc(bufsize);
if (reqbuf == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
}
ifc.ifc_buf = (caddr_t)&reqbuf[0];
ifc.ifc_len = bufsize;
if (ioctl(s, SIOCGIFCONF, (char *)&ifc) == -1) {
        perror("ioctl(SIOCGIFCONF)");
        exit(1);
}
...
}
```

この呼び出しの後、*buf* には ifreq 構造体の配列が含まれます。ifreq 構造体は、ホストに接続されているすべてのネットワークごとに 1 つずつ存在します。これらの構造体のソート順は次のとおりです。

- インタフェース名のアルファベット順
- サポートされるアドレスファミリの番号順 ifc.ifc_len の値は ifreq 構造体が使用したバイト数に設定されます。

各構造体は、対応するネットワークが稼働または停止しているか、ポイントツーポイントまたはブロードキャストのどちらであるか、などを示すインタフェースフラグセットを持ちます。次の例では、ifreq 構造体が指定するインタフェース用の SIOCGIFFLAGS フラグを返す ioctl(2) を示します。

例 **6–15** インタフェースフラグの取得

```
struct ifreq *ifr;
ifr = ifc.ifc_req;
for (n = ifc.ifc_len/sizeof (struct ifreq); --n>= 0; ifr++) {
   /*
    * 別の目的でアドレスファミリ用に使用されているインタフェースを
    * 使用しないよう注意する。
    */
   if (ifr->ifr_addr.sa_family != AF_INET)
      continue;
   if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
```

例 **6–15** インタフェースフラグの取得　(続き)

```
        ...
    }
      if ((ifr->ifr_flags & IFF_UP) == 0 ||
       (ifr->ifr_flags & IFF_LOOPBACK) ||
       (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
       continue;
}
```

次の例では、インタフェースのブロードキャストアドレスを取得するための SIOGGIFBRDADDR ioctl(2) コマンドを示します。

例 **6–16** インタフェースのブロードキャストアドレス

```
if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
        ...
}
memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
        sizeof ifr->ifr_broadaddr);
```

また、SIOGGIFBRDADDR ioctl(2) を使用すると、ポイントツーポイントインタフェースの宛先アドレスを取得できます。

インタフェースのブロードキャストアドレスを取得したあと、sendto(3SOCKET) を使用してブロードキャストデータグラムを送信します。

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

ホストのインタフェースがブロードキャストまたはポイントツーポイントアドレスをサポートする場合、そのホストが接続されているインタフェースごとに 1 つの sendto(3SOCKET) を使用します。

マルチキャストの使用

IP マルチキャストは、タイプ SOCK_DGRAM と SOCK_RAW のソケット AF_INET6 と AF_INET でのみサポートされます。IP マルチキャストはまた、インタフェースドライバがマルチキャストをサポートするサブネットワーク上でのみサポートされます。

IPv4 マルチキャストデータグラムの送信

マルチキャストデータグラムを送信するには、sendto(3SOCKET) 呼び出しで宛先アドレスとして 224.0.0.0 から 239.255.255.255 までの範囲の IP マルチキャストアドレスを指定します。

デフォルトでは、IP マルチキャストデータグラムは生存期間 (TTL) 1 で送信されます。この値では、データグラムは単一のサブネットワーク外には転送されません。ソケットオプション IP_MULTICAST_TTL を指定すると、後続のマルチキャストデータグラムの TTL を 0 から 255 までの任意の値に設定できます。したがって、マルチキャストの配信範囲を制御できます。

```
u_char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,sizeof(ttl))
```

TTL 0 のマルチキャストデータグラムはどのサブネット上でも伝送されませんが、送信ホストが宛先グループに属しており、送信側ソケットでマルチキャストループバックが有効な場合は、ローカルに配信できます。最初の配信先 (ホップ) となるサブネットが 1 つまたは複数のマルチキャストルーターに接続されている場合、1 より大きな TTL を持つマルチキャストデータグラムを複数のサブネットに配信できます。配信範囲の制御に意味を持たせるために、マルチキャストルーターは TTL しきい値という概念をサポートします。このしきい値は、一定の TTL より少ないデータグラムが一定のサブネットを超えることを回避します。このしきい値は、次のような初期 TTL の値を使用して、マルチキャストデータグラムの規約を実施します。

0　同じホストに制限される

1　同じサブネットに制限される

32　同じサイトに制限される

64　同じ地域に制限される

128　同じ大陸に制限される

255　配信範囲内で制限されない

サイトと地域は厳密には定義されず、サイトはローカルの事柄としてさらに小さな管理ユニットに分割できます。

アプリケーションは、上記の TTL 以外に初期 TTL を選択できます。たとえば、アプリケーションはマルチキャスト照会を送信することによって (つまり、TTL を 0 から開始して、応答を受信するまで、TTL を大きくしていく照会のこと)、ネットワークリソースの拡張リング検索を実行できます。

マルチキャストルーターは、TTL の値にかかわらず、224.0.0.0 から 224.0.0.255 までの宛先アドレスを持つマルチキャストデータグラムを転送しません。この範囲のアドレスは、経路指定プロトコルとその他の低レベルトポロジの発見または保守プロトコル (ゲートウェイ発見、グループメンバーシップ報告など) の使用に予約されています。

ホストが複数のマルチキャスト可能なインタフェースを持つ場合でも、各マルチキャスト伝送は単一のネットワークインタフェースから送信されます。ホストがマルチキャストルーターでもあり、TTL が 1 より大きい場合には、発信元以外のインタフェースにもマルチキャストを転送できます。ソケットオプションを使用すると、特定のソケットからの後続の転送用のデフォルトを変更できます。

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr))
```

addr は、希望する発信インタフェースのローカル IP アドレスです。デフォルトインタフェースに戻すには、アドレス INADDR_ANY を指定します。インタフェースのローカル IP アドレスを取得するには、SIOCGIFCONF ioctl を使用します。インタフェースがマルチキャストをサポートするかどうかを判断するには、 SIOCGIFFLAGS ioctl を使用してインタフェースフラグを取り出し、IFF_MULTICAST フラグが設定されているかどうかをテストします。このオプションは、インターネットトポロジと明確な関係があるマルチキャストルーターなどのシステムサービスを主な対象としています。

送信ホスト自体が属しているグループにマルチキャストデータグラムが送信された場合、デフォルトでは、データグラムのコピーが IP 層によってローカル配信用にループバックされます。次のように別のソケットオプションを使用すると、送信側は明示的に、後続のデータグラムがループバックされるかどうかを制御できます。

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop))
```

loop の値は、ループバックを無効にする場合は 0、ループバックを有効にする場合は 1 です。このオプションを使用すると、自分自身の伝送を受信するというオーバーヘッドを排除できるので、単一のホストに単一のインスタンスしか持たないアプリケーションの性能が上がります。単一のホストに複数のインスタンスを持つアプリケーションや送信側が宛先グループに属さないアプリケーションは、このオプションを使用してはなりません。

送信ホストが別のインタフェースの宛先グループに属している場合、1 を超える初期 TTL で送信されたマルチキャストデータグラムは、他方のインタフェース上の送信ホストに配信できます。このような配信には、ループバック制御オプションは何の効果もありません。

IPv4 マルチキャストデータグラムの受信

IP マルチキャストデータグラムを受信するためには、ホストは 1 つまたは複数の IP マルチキャストグループのメンバーになる必要があります。プロセスは、次のソケットオプションを使用して、マルチキャストグループに加わるようにホストに求めることができます。

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq))
```

mreq は次の構造体です。

```
struct ip_mreq {
    struct in_addr imr_multiaddr;   /* 加わるマルチキャストグループ */
    struct in_addr imr_interface;   /* 加わるインタフェース */
}
```

各メンバーシップは単一のインタフェースに関連付けられます。したがって、複数のインタフェース上にある同じグループに加わることができます。デフォルトのマルチキャストインタフェースを選択するには、imr_interface アドレスに in6addr_any を指定します。特定のマルチキャスト可能なインタフェースを選択するには、ホストのローカルアドレスの 1 つを指定します。

メンバーシップを取り消すには、次のコードを使用します。

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq))
```

mreq には、メンバーシップの追加に使用した値と同じ値が入ります。ソケットを閉じるか、ソケットを保持しているプロセスを停止すると、そのソケットに関連付けられたメンバーシップは取り消されます。特定のグループ内で複数のソケットがメンバーシップを要求でき、ホストは最後の要求が取り消されるまでそのグループのメンバーにとどまります。

任意のソケットがデータグラムの宛先グループのメンバーシップを要求した場合、カーネル IP 層は受信マルチキャストパケットを受け入れます。特定のソケットがマルチキャストデータグラムを受信するかどうかは、ソケットに関連付けられた宛先ポートとメンバーシップ、または、raw ソケットのプロトコルタイプによって決定されます。特定のポートに送信されたマルチキャストデータグラムを受信するには、ローカルアドレスを未指定のまま (INADDR_ANY などに指定) ローカルポートにバインドします。

bind(3SOCKET) の前に次に示すコードがあると、複数のプロセスを同じ SOCK_DGRAM UDP ポートにバインドできます。

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

この場合、共有ポートに向けられた各受信マルチキャストまたは受信ブロードキャスト UDP データグラムは、そのポートにバインドされているすべてのソケットに配信されます。下位互換性の理由から、この配信は単一キャストの受信データグラムには適用されません。データグラムの宛先ポートにバインドされているソケットの数にかかわらず、単一キャストデータグラムが複数のソケットに配信されることはありません。SOCK_RAW ソケットは、SO_REUSEADDR オプションがなくても単一の IP プロトコルタイプを共有できます。

マルチキャストに関連する新しいソケットオプションの説明は、<netinet/in.h> を参照してください。IP アドレスはすべて、ネットワークバイトオーダーで渡されます。

IPv6 マルチキャストデータグラムの送信

IPv6 マルチキャストデータグラムを送信するには、sendto(3SOCKET) 呼び出しで宛先アドレスとして ff00::0/8 という範囲内の IP マルチキャストアドレスを指定します。

デフォルトでは、IP マルチキャストデータグラムはホップ制限 1 で送信されます。この値では、データグラムは単一のサブネットワーク外には転送されません。ソケットオプション IPV6_MULTICAST_HOPS を指定すると、後続のマルチキャストデータグラムのホップ制限を 0 から 255 までの任意の値に設定できます。したがって、マルチキャストの配信範囲を制御できます。

```
uint_l;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hops,sizeof(hops))
```

ホップ制限 0 のマルチキャストデータグラムはどのサブネットにも伝送できませんが、次の場合、データグラムはローカルに配信できます。

- 送信ホストが宛先グループに属している場合
- 送信側ソケットでマルチキャストループバックが有効な場合

最初の配信先 (ホップ) となるサブネットが 1 つまたは複数のマルチキャストルーターに接続されている場合、1 より大きなホップ制限を持つマルチキャストデータグラムを複数のサブネットに配信できます。IPv4 マルチキャストアドレスと異なり、IPv6 マルチキャストアドレスには、アドレスの最初の部分にコード化された明示的な配信範囲情報が含まれます。定義されている配信範囲を次に示します (x は未指定)。

ffX1::0/16　ノード – ローカルな配信範囲 — 同じノードに制限される

ffX2::0/16　リンク – ローカルな配信範囲

ffX5::0/16　サイト – ローカルな配信範囲

ffX8::0/16　組織 – ローカルな配信範囲

ffXe::0/16　グローバルスコープ

アプリケーションは、マルチキャストアドレスの配信範囲とは個別に、異なるホップ制限値を使用できます。たとえば、アプリケーションはマルチキャスト照会を送信することによって (つまり、ホップ制限を 0 から開始して、応答を受信するまで、ホップ制限を大きくしていく照会のこと)、ネットワークリソースの拡張リング検索を実行できます。

ホストが複数のマルチキャスト可能なインタフェースを持つ場合でも、各マルチキャスト伝送は単一のネットワークインタフェースから送信されます。ホストがマルチキャストルーターでもあり、ホップ制限が 1 より大きい場合には、発信元以外のインタフェースにもマルチキャストを転送できます。ソケットオプションを使用すると、特定のソケットからの後続の転送用のデフォルトを変更できます。

```
uint_t ifindex;

ifindex = if_nametoindex )"hme3");
```

```
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_IF, &ifindex,
      sizeof(ifindex))
```

ifindex は、希望する発信インタフェースのインタフェースインデックスです。デフォルトインタフェースに戻すには、値 0 を指定します。

送信ホスト自体が属しているグループにマルチキャストデータグラムが送信された場合、デフォルトでは、データグラムのコピーが IP 層によってローカル配信用にループバックされます。別のソケットオプションを使用すると、送信側は明示的に、後続のデータグラムをループバックするかどうかを制御できます。

```
uint_t loop;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop,
        sizeof(loop))
```

loop の値は、ループバックを無効にする場合は 0、ループバックを有効にする場合は 1 です。単一のホストにインスタンスを 1 つしか持たないアプリケーション (ルーターやメールデーモンなど) では、このオプションを使用するとアプリケーション自体の伝送を受信するオーバーヘッドが排除されるため、性能が向上します。このオプションは、単一のホスト上に複数のインスタンスを持つアプリケーション (会議システムプログラムなど) や送信側が宛先グループに属さないアプリケーション (時間照会プログラムなど) に使用してはなりません。

送信ホストが別のインタフェースの宛先グループに属している場合、1 を超えるホップ制限で送信されたマルチキャストデータグラムは、他方のインタフェース上の送信ホストに配信できます。このような配信には、ループバック制御オプションは何の効果もありません。

IPv6 マルチキャストデータグラムの受信

IP マルチキャストデータグラムを受信するためには、ホストは 1 つまたは複数の IP マルチキャストグループのメンバーになる必要があります。プロセスは、次のソケットオプションを使用して、マルチキャストグループに加わるようにホストに求めることができます。

```
struct ipv6_mreq mreq;
setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, &mreq, sizeof(mreq))
```

mreq は次の構造体です。

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr;   /* IPv6 マルチキャストアドレス */
    unsigned int    ipv6mr_interface;   /* インタフェースインデックス */
}
```

各メンバーシップは単一のインタフェースに関連付けられます。したがって、複数のインタフェース上にある同じグループに加わることができます。デフォルトのマルチキャストインタフェースを選択するには、ipv6_interface に 0 を指定します。マルチキャスト可能なインタフェースを選択するには、ホストのインタフェースの 1 つのインタフェースインデックスを指定します。

グループから抜けるには、次のコードを使用します。

```
struct ipv6_mreq mreq;
setsockopt(sock, IPPROTO_IPV6, IP_LEAVE_GROUP, &mreq, sizeof(mreq))
```

mreq には、メンバーシップの追加に使用した値と同じ値が入ります。ソケットを閉じるか、ソケットを保持しているプロセスを停止すると、そのソケットに関連付けられたメンバーシップは取り消されます。複数のソケットは特定のグループ内の 1 つのメンバーシップを要求できます。このとき、ホストは最後の要求が取り消されるまでそのグループのメンバーに残ります。

任意のソケットがデータグラムの宛先グループのメンバーシップを要求した場合、カーネル IP 層は受信マルチキャストパケットを受け入れます。特定のソケットがマルチキャストデータグラムを受信するかどうかは、ソケットに関連付けられた宛先ポートとメンバーシップ、または、raw ソケットのプロトコルタイプによって決定されます。特定のポートに送信されたマルチキャストデータグラムを受信するには、ローカルアドレスを未指定のまま (INADDR_ANY などに指定) ローカルポートにバインドします。

bind(3SOCKET) の前に次に示すコードがあると、複数のプロセスを同じ SOCK_DGRAM UDP ポートにバインドできます。

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

この場合、ポートにバインドされているすべてのソケットは、共有ポートに向けられたすべての受信マルチキャスト UDP データグラムを受信します。下位互換性の理由から、この配信は単一キャストの受信データグラムには適用されません。データグラムの宛先ポートにバインドされているソケットの数にかかわらず、単一キャストデータグラムが複数のソケットに配信されることはありません。SOCK_RAW ソケットは、SO_REUSEADDR オプションがなくても単一の IP プロトコルタイプを共有できます。

マルチキャストに関連する新しいソケットオプションの説明は、<netinet/in.h> を参照してください。IP アドレスはすべて、ネットワークバイトオーダーで渡されます。

第 7 章

XTI と TLI を使用したプログラミング

この章では、トランスポート層インタフェース (TLI) と X/Open トランスポートインタフェース (XTI) について説明します。非同期実行モードなどの拡張機能については、152 ページの「XTI/TLI の拡張機能」で説明します。

分散/集中データ転送などの最近 XTI に追加された機能については、173 ページの「XTI インタフェースへの追加」で説明します。

OSI モデル (第 4 層) のトランスポート層はアプリケーションと上位層の間でエンドツーエンドのサービスを提供するモデルの最下位層です。この層は、配下のネットワークのトポロジと特性をユーザーには見えないようにします。トランスポート層はまた、現在の数多くのプロトコル群 (OSI プロトコル、TCP および TCP/IP インターネットプロトコル群、Xerox Network Systems (XNS)、システムネットワークアーキテクチャ (System Network Architecture、SNA) など) に共通する一連のサービスを定義します。

TLI は、業界標準の Transport Service Definition (ISO 8072) でモデル化されています。TLI は、TCP と UDP の両方にアクセスするために使用できます。XTI と TLI はネットワークプログラミングインタフェースを構成するインタフェースセットです。XTI は SunOS 4 プラットフォーム用の以前の TLI インタフェースを発展させたものです。Solaris オペレーティングシステムはどちらのインタフェースもサポートしますが、このインタフェースセットの将来の方向性を表しているのは XTI の方です。Solaris ソフトウェアは STREAMS 入出力メカニズムを使用して、XTI と TLI をユーザーライブラリとして実装しています。

XTI と TLI について

注 – この章で取り上げるインタフェースはマルチスレッドに対して安全です。これは XTI/TLI インタフェース呼び出しを含むアプリケーションがマルチスレッド化されたアプリケーション内で自由に使用できることを意味します。これらのインタフェース呼び出しは再入可能ではないので、スケーラビリティは直線的でありません。

注意 – XTI/TLI インタフェースの非同期環境における動作は特定されていません。これらのインタフェースはシグナルハンドラルーチンからは使用しないでください。

TLI は AT&T の System V Release 3 とともに 1986 年に導入されました。TLI は トランスポート層インタフェース API を規定しました。TLI は ISO Transport Service Definition が規定するモデルに基づいています。TLI は OSI トランスポート層とセッション層の間の API を提供します。TLI インタフェースは UNIX の AT&T System V Release 4 バージョンでさらに発展し、SunOS 5.6 オペレーティングシステムインタフェースにも取り入れられました。

XTI インタフェースは TLI インタフェースを発展させたもので、このインタフェースの将来の方向を表しています。TLI を使用するアプリケーションとの互換性が保証されています。ただちに TLI のアプリケーションを XTI のアプリケーションに移行する必要性はありませんが、新しいアプリケーションでは XTI インタフェースを使用し、必要に応じて、TLI アプリケーションを XTI に移行してください。

TLI はライブラリ (`libns1`) 内のインタフェース呼び出しセットとして実装され、そこにアプリケーションがリンクします。XTI アプリケーションは c89 フロントエンドを使用してコンパイルし、`xnet` ライブラリ (`libxnet`) とリンクする必要があります。XTI を使用するコンパイルの詳細については、`standards`(5) のマニュアルページを参照してください。

注 – XTI インタフェースを使用するアプリケーションは `xti.h` ヘッダーファイルを使用するのに対し、TLI インタフェースを使用するアプリケーションは `tiuser.h` ヘッダーファイルを使用しています。

XTI/TLI コードは第 4 章で説明されている追加のインタフェースとメカニズムと組み合わせと同時に使用することで、現在のトランスポートプロバイダに依存する必要がなくなります。SunOS 5.x はいくつかのトランスポートプロバイダ (たとえば、TCP) をオペレーティングシステムの一部として用意しています。トランスポートプロバイ

ダはサービスを実行し、トランスポートユーザーはサービスを要求します。トランスポートユーザーがトランスポートプロバイダへサービス要求を行います。たとえば、TCP や UDP 上のデータ転送要求などがそれに当たります。

XTI/TLI は次の 2 つの構成要素を利用することによっても、トランスポートに依存しないプログラミングが可能になります。

- トランスポート選択や名前からアドレスへの変換 (name-to-address) を始めとするトランスポートサービスを実行するライブラリルーチン。ネットワークサービスライブラリには ユーザープロセスで XTI/TLI を実装するインタフェースセットが用意されています。詳細は第 8 章を参照してください。

  TLI を使用するプログラムは `libnsl` ネットワークサービスライブラリにリンクする必要があります(コンパイル時に `-l nsl` オプションを使用)。

  XTI を使用するプログラムは `xnet` ライブラリにリンクする必要があります (コンパイル時に `-l xnet` オプションを使用) 。
- 状態遷移規則は、トランスポートルーチンを呼び出すシーケンスを定義します。状態遷移規則の詳細については、163 ページの「状態遷移」を参照してください。状態テーブルは状態およびイベントの処理に基づいて、ライブラリ呼び出しの正当なシーケンス定義します。これらのイベントには、ユーザー生成ライブラリ呼び出し、プロバイダ生成イベントのインジケータが含まれます。XLI/TLI のプログラマはインタフェースを使用する前にすべての状態遷移をよく理解しておく必要があります。

XTI/TLI 読み取りインタフェースと書き込み用インタフェース

ユーザーがコネクション中に受信したデータを処理するために、既存のプログラム上 (`/usr/bin/cat` など) で `exec`(2) を使用してトランスポートコネクションを確立したいとします。既存のプログラムは `read`(2) および `write`(2) を使用します。XTI/TLI は直接トランスポートプロバイダへの読み取りインタフェースと書き込みインタフェースをサポートしていませんが、これを処理することが可能です。このインタフェースを使用すると、データ転送フェーズにおいて `read`(2) および `write`(2) 呼び出しをトランスポートコネクション上で実行できます。このセクションでは XTI/TLI のコネクションモードサービスへの読み取りインタフェースと書き込みインタフェースについて説明しています。なおこのインタフェースはコネクションレスモードサービスでは使用できません。

例 **7–1** 読み取りインタフェースと書き込みインタフェース

```
#include <stropts.h>
    .
    ./*
      同一のローカル管理および接続確立手順
```

例 **7–1** 読み取りインタフェースと書き込みインタフェース　*(続き)*

```
     */
    .
    if (ioctl(fd, I_PUSH, "tirdwr") == -1) {
       perror("I_PUSH of tirdwr failed");
       exit(5);
      }
    close(0);
    dup(fd);
    execl("/usr/bin/cat", "/usr/bin/cat", (char *) 0);
    perror("exec of /usr/bin/cat failed");
    exit(6);
}
```

クライアントは tirdwr をトランスポート終端に関連付けられたストリーム内にプッシュすることにより読み取りインタフェースと書き込みインタフェースを呼び出します。詳細については、streamio(7I) のマニュアルページの I_PUSH を参照してください。tirdwr モジュールはトランスポートプロバイダより上位に位置する XTI/TLI を純粋な読み取りインタフェースと書き込みインタフェースに変換します。モジュールが設置された段階で、クライアントは close(2) および dup(2) を呼び出してトランスポート終端を標準入力ファイルとして確立し、/usr/bin/cat を使用して入力を処理します。

tirdwr をトランスポートプロバイダにプッシュすると、XTI/TLI は read(2) および write(2) の意味論を使用するようになります。read および write の意味論を使用するとき、XTI/TLI はメッセージ境界を保持しません。トランスポートプロバイダから tirdwr をポップすると、XTI/TLI は本来の意味論に戻ります (streamio(7I) のマニュアルページの I_POP を参照)。

注意 – tirdwr モジュールをストリーム上にプッシュできるのは、トランスポート終端がデータ転送フェーズ中にある場合だけです。モジュールをプッシュしたあと、ユーザーは XLI/TLI ルーチンを呼び出すことはできません。ユーザーが XTI/TLI ルーチンを呼び出した場合、tirdwr はストリーム上に重大なプロトコルエラー EPROTO を生成し、使用不可であることを通知します。このとき、tirdwr モジュールをストリーム上からポップすると、トランスポートコネクションは中止されます。詳細については、streamio(7I) のマニュアルページの I_POP を参照してください。

データの書き込み

write(2) を使用してトランスポートコネクションにデータを送信したあと、tirdwr はトランスポートプロバイダを通じてデータを渡します。メカニズム上は許可されていますが、ゼロ長のデータパケットを送った場合、tirdwr はメッセージを破棄します。トランスポートコネクションが中止された場合、ハングアップ状態がストリーム

上に生成され、それ以降の write(2) 呼び出しは失敗し、errno は ENXIO に設定されます。この問題が発生するのは、たとえば、リモートユーザーが t_snddis(3NSL) を使用してコネクションを中止した場合があります。ハングアップ後も利用できるデータの取り出しは可能です。

データの読み取り

トランスポートコネクションに着信したデータを読み取るには、read(2) を使用します。tirdwr はトランスポートプロバイダからデータを渡します。tirdwr モジュールは、トランスポートプロバイダからユーザーに渡されるその他のイベントまたは要求を次のように処理します。

- read(2) はユーザーへ送られる優先データを識別できません。read(2) が優先データ要求を受信した場合、tirdwr はストリーム上に重大なプロトコルエラー EPROTO を生成します。このエラーが発生すると、後続のシステムコールは失敗します。優先データを受信するときには、read(2) を使用しないでください。
- tirdwr は放棄型の切断要求を破棄し、ストリーム上にハングアップ状態を生成します。後続の read(2) 呼び出しには残りのデータを返し、すべてのデータを返した後の呼び出しにはファイルの終わりを示す 0 を返します。
- tirdwr は正常型解放要求を破棄し、ゼロ長のメッセージをユーザーに配信します。read(2) のマニュアルページで説明するようにファイルの終わりを示す 0 をユーザーに返します 。
- read(2) がその他の XTI/TLI 要求を受信した場合、tirdwr はストリーム上に重大なプロトコルエラー EPROTO を生成します。このエラーが発生すると、後続のシステムコールは失敗します。コネクションを確立したあと、ユーザーが tirdwr をストリーム上にプッシュした場合、tirdwr は要求を生成しません。

コネクションを閉じる

ストリーム上に tirdwr が存在する場合、コネクションの間はトランスポートコネクション上でデータの送受信が可能です。どちらのユーザーも、トランスポート終端に関連付けられたファイル記述子を閉じることにより、またはストリーム上から tirdwr モジュールをポップさせることによりコネクションを終了させることが可能です。どちらの場合も tirdwr は次の処理を行います。

- 正常型解放要求を受信した場合、tirdwr は要求をトランスポートプロバイダに渡してコネクションを正常に解放します。データ転送が完了すると、正常型解放手続きを実行したリモートユーザーは期待される結果を受信します。
- 切断要求を受信した場合、tirdwr は特別な処理を行いません。
- 正常型解放要求または切断要求のどちらも受信しない場合、tirdwr は切断要求をトランスポートプロバイダに渡してコネクションを中止します。
- ストリーム上でエラーが発生したときに切断要求を受信しない場合、tirdwr は切断要求をトランスポートプロバイダに渡します。

tirdwr をストリーム上にプッシュしたあと、プロセスは正常型解放を実行できません。トランスポートコネクションの相手側のユーザーが解放を実行した場合、tirdwr は正常型解放を処理します。このセクションのクライアントがサーバープログラムと通信している場合、サーバーは正常型解放要求を使用してデータの転送を終了します。次に、サーバーはクライアントからの対応する要求を待ちます。この時点でクライアントは、トランスポート終端を終了して閉じます。ファイル記述子を閉じたあと、tirdwr はコネクションのクライアント側から正常解放型要求を実行します。この解放によって、サーバーをブロックする要求が生成されます。

データがそのままで配信されることを保証するために、この正常型解放を必要とするTCP などのプロトコルもあります。

XTI/TLI の拡張機能

このセクションでは高度な XTI/TLI の概念を説明します。

- 152 ページの「非同期実行モード」では、いくつかのライブラリ呼び出しで使用するオプションの非ブロッキング (非同期) モードについて説明します。
- 153 ページの「 XTI/TLI の高度なプログラミング例」では、複数の未処理コネクション要求をサポートし、イベント方式で動作するサーバーのプログラム例を示します。

非同期実行モード

多くの XTI/TLI ライブラリルーチンは受信イベントの発生を待機するブロックを行います。ただし、処理時間の条件が高いアプリケーションではこれを使用しないでください。アプリケーションは、非同期 XTI/TLI イベントを待機する間にローカル処理が行えます。

アプリケーションが XTI/TLI イベントの非同期処理にアクセスするには、XTI/TLI ライブラリルーチンの非同期機能と非ブロッキングモードを組み合わせて使用する必要があります。poll(2) システムコールと I_SETSIG ioctl(2) コマンドを使用してイベントを非同期的に処理する方法については、『ONC+ 開発ガイド』を参照してください。

イベントが発生するまでブロックする各 XTI/TLI ルーチンは特別な非ブロッキングモードで実行できます。たとえば、t_listen(3NSL) は通常、コネクション要求を受信するまでブロックします。サーバーは t_listen(3NSL) を非ブロッキング (または非同期) モードで呼び出すことによって、トランスポート終端を定期的にポーリングして、コネクション要求が待ち行列に入っているかを確認できます。非同期モードを有効にするには、ファイル記述子に O_NDELAY または O_NONBLOCK を設定します。これらのモードは、t_open(3NSL) を使用してフラグとして設定するか、また

は、XTI/TLI ルーチンを呼び出す前に fcntl(2) を呼び出して設定することになります。fcntl(2) を使用すると、このモードをいつでも有効または無効にできます。なおこの章のすべてのプログラム例ではデフォルトの同期処理モードを使用しています。

O_NDELAY と O_NONBLOCK を使用することによって各 XLI/TLI ルーチンに与える影響はそれぞれ異なります。特定のルーチンへの影響を知るには、O_NDELAY と O_NONBLOCK の正確な意味論を認識する必要があります。

XTI/TLI の高度なプログラミング例

例 7–2 に、2 つの重要な概念を示します。1 つ目はサーバーにおける複数の未処理のコネクション要求に対する管理能力。2 つ目はイベント方式の XLI/TLI の使用法およびシステムコールインタフェースです。

XTI/TLI を使用すると、サーバーは複数の未処理のコネクション要求を管理できます。複数のコネクション要求を同時に受信する理由の 1 つは、クライアントを順位付けることです。複数のコネクション要求を受信した場合、サーバーはクライアントの優先順位に従ってコネクション要求を受け付けることが可能です。

複数の未処理コネクション要求を同時に処理する理由の 2 つ目、シングルスレッド処理の限界です。トランスポートプロバイダによっては、サーバーが 1 つのコネクション要求を処理する間、ほかのクライアントにはサーバーがビジーであると見えます。複数のコネクション要求を同時に処理する場合、サーバーがビジーになるのは、サーバーを同時に呼び出そうとするクライアントの数が最大数を超える場合だけです。

次のサーバーの例はイベント方式です。プロセスはトランスポート終端をポーリングして、XTI/TLI 受信イベントが発生しているかを確認し、受信したイベントに適切な処理を行います。複数のトランスポート終端をポーリングして、受信イベントが発生しているか確認する例を示します。

例 **7–2** 終端の確立 (複数コネクションへ変更可能)

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS 1
#define MAX_CONN_IND 4
#define SRV_ADDR 1                  /*サーバー既知アドレス*/

int conn_fd;                        /*サーバーの接続*/
extern int t_errno;
/*接続要求を格納*/
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
```

例 **7–2** 終端の確立 (複数コネクションへ変更可能)　(続き)

```
   struct pollfd pollfds[NUM_FDS];
   struct t_bind *bind;
   int i;

   /*
    * 1 つのトランスポート終端をオープンし、バインドする
    * 複数の指定も可能
    */
   if ((pollfds[0].fd = t_open("/dev/tivc", O_RDWR,
         (struct t_info *) NULL)) == -1) {
      t_error("t_open failed");
      exit(1);
   }
   if ((bind = (struct t_bind *) t_alloc(pollfds[0].fd, T_BIND,
         T_ALL)) == (struct t_bind *) NULL) {
      t_error("t_alloc of t_bind structure failed");
      exit(2);
   }
   bind->qlen = MAX_CONN_IND;
   bind->addr.len = sizeof(int);
   *(int *) bind->addr.buf = SRV_ADDR;
   if (t_bind(pollfds[0].fd, bind, bind) == -1) {
      t_error("t_bind failed");
      exit(3);
   }
   /*正しいアドレスがバインドされたかどうか */
   if (bind->addr.len != sizeof(int) ||
      *(int *)bind->addr.buf != SRV_ADDR) {
      fprintf(stderr, "t_bind bound wrong address\n");
      exit(4);
   }
}
```

t_open(3NSL) によって返されるファイル記述子は pollfd 構造体に格納され、トランスポート終端にデータの受信イベントが発生しているかを確認するポーリングを制御するときに使用されます。poll(2) のマニュアルページを参照してください。この例では 1 つのトランスポート終端だけが確立されます。ただし、例の残りの部分は複数のトランスポート終端を管理するために書かれています。例 7–2 を少し変更をすることにより複数のトランスポート終端をサポートできるようになります。

このサーバーは t_bind(3NSL) 用に qlen を 1 より大きな値に設定します。この値は、サーバーが複数の未処理のコネクション要求を待ち行列に入れる必要があるということを指定します。サーバーは現在のコネクション要求の受け付けを行なってから、別のコネクション要求を受け付けます。この例では、MAX_CONN_IND 個までのコネクション要求を待ち行列に入れることができます。MAX_CONN_IND 個の未処理のコネクション要求をサポートできない場合、トランスポートプロバイダはネゴシエーションを行なって qlen の値を小さくすることができます。

アドレスをバインドし、コネクション要求を処理できるようになったあと、サーバーは次の例に示すように動作します。

例 **7–3** コネクション要求の処理

```
pollfds[0].events = POLLIN;

while (TRUE) {
    if (poll(pollfds, NUM_FDS, −1) == −1) {
   perror(“poll failed”);
   exit(5);
    }
    for (i = 0; i < NUM_FDS; i++) {
   switch (pollfds[i].revents) {
      default:
         perror(“poll returned error event”);
      exit(6);
      case 0:
         continue;
      case POLLIN:
         do_event(i, pollfds[i].fd);
         service_conn_ind(i, pollfds[i].fd);
       }
   }
}
```

pollfd 構造体の events フィールドは POLLIN に設定され、XTI/TLI 受信イベントをサーバーに通知します。次にサーバーは無限ループに入り、トランスポート終端をポーリングして、イベントが発生している場合はイベントを処理します。

poll(2) 呼び出しは受信イベントが発生するまで無期限にブロックします。応答時に、サーバーは トランスポート終端ごとに 1 つずつあるエントリごとに revents の値を確認し、新しいイベントが発生しているかを確認します。revents が 0 の場合、この終端上ではイベントが生成されていないので、サーバーは次の終端に進みます。revents が POLLIN の場合は終端上にイベントがあるため、do_event を呼び出してイベントを処理します。revents がそれ以外の値の場合は、終端上のエラーを通知し、サーバーは終了します。終端が複数ある場合、サーバーはこのファイル記述子を閉じて、処理を継続します。

サーバーはループを繰り返すごとに service_conn_ind を呼び出して、未処理のコネクション要求を処理します。ほかのコネクション要求が保留状態の場合、service_conn_ind は新しいコネクション要求を保存してあとで処理します。

次に、サーバーが do_event を呼び出して受信イベントを処理する例を示します。

例 **7–4** イベント処理ルーチン

```
do_event( slot, fd)
int slot;
int fd;
{
   struct t_discon *discon;
   int i;

   switch (t_look(fd)) {
   default:
```

例 **7–4** イベント処理ルーチン　(続き)

```
      fprintf(stderr, "t_look: unexpected event\n");
      exit(7);
   case T_ERROR:
      fprintf(stderr, "t_look returned T_ERROR event\n");
      exit(8);
   case −1:
      t_error("t_look failed");
      exit(9);
   case 0:
      /*POLLIN の戻りのため、本来起きるべきではない*/
      fprintf(stderr,"t_look returned no event\n");
      exit(10);
   case T_LISTEN:
      /*calls 配列内の未使用要素を捜す*/
      for (i = 0; i < MAX_CONN_IND; i++) {
         if (calls[slot][i] == (struct t_call *) NULL)
            break;
      }
      if ((calls[slot][i] = (struct t_call *) t_alloc( fd, T_CALL,
               T_ALL)) == (struct t_call *) NULL) {
         t_error("t_alloc of t_call structure failed");
         exit(11);
      }
      if (t_listen(fd, calls[slot][i] ) == −1) {
         t_error("t_listen failed");
         exit(12);
      }
      break;
   case T_DISCONNECT:
      discon = (struct t_discon *) t_alloc(fd, T_DIS, T_ALL);
      if (discon == (struct t_discon *) NULL) {
         t_error("t_alloc of t_discon structure failed");
         exit(13)
      }
      if(t_rcvdis( fd, discon) == −1) {
         t_error("t_rcvdis failed");
         exit(14);
      }
      /*配列内から切断要求エントリを見つけ、削除*/
      for (i = 0; i < MAX_CONN_IND; i++) {
         if (discon->sequence == calls[slot][i]->sequence) {
            t_free(calls[slot][i], T_CALL);
            calls[slot][i] = (struct t_call *) NULL;
         }
      }
      t_free(discon, T_DIS);
      break;
   }
}
```

例 7–4 の引数は番号の *slot* とファイル記述子の *fd* です。*slot* は各トランスポート終端のエントリを持つグローバル配列 calls のインデックスです。各エントリは終端で受信されるコネクション要求を保持する t_call 構造体の配列です。

do_event モジュールは t_look (3NSL) を呼び出して、*fd* が指す終端上の XTI/TLI イベントを識別します。イベントがコネクション要求 (T_LISTEN イベント) あるいは切断要求 (T_DISCONNECT イベント) する場合、イベントは処理されます。それ以外の場合、サーバーはエラーメッセージを出力して終了します。

コネクション要求の場合、do_event は最初の未使用エントリを検索するため未処理のコネクション要求配列を走査します。エントリには t_call 構造体が割り当てられ、コネクション要求は t_listen(3NSL) によって受信されます。配列は未処理コネクション要求の最大数を保持するのに十分な大きさを持っています。コネクション要求の処理は延期されます。

切断要求は事前に送られたコネクション要求と対応している必要があります。要求を受信するために、do_event モジュールは t_discon 構造体を割り当てます。この構造体には次のフィールドが存在します。

```
struct t_discon {
     struct netbuf udata;
     int reason;
     int sequence;
}
```

udata 構造体には、切断要求によって送信されたユーザーデータが含まれます。reason の値には、プロトコル固有の切断理由コードが含まれます。sequence の値は、切断要求に一致するコネクション要求を識別します。

サーバーは切断要求を受信するために、t_rcvdis(3NSL) を呼び出します。次に、コネクション要求の配列を走査して、切断要求の sequence 番号と一致するコネクション要求があるかどうかを走査します。一致するコネクション要求が見つかった場合、サーバーはその構造体を解放して、エントリを NULL に設定します。

トランスポート終端上にイベントが見つかった場合、サーバーは終端上の待ち行列に入っているすべてのコネクション要求を処理するために service_conn_ind を呼び出します。

例 **7–5** すべてのコネクション要求の処理

```
service_conn_ind(slot, fd)
{
   int i;

    for (i = 0; i < MAX_CONN_IND; i++) {
      if (calls[slot][i] == (struct t_call *) NULL)
         continue;
      if((conn_fd = t_open( "/dev/tivc", O_RDWR,
            (struct t_info *) NULL)) == -1) {
         t_error("open failed");
         exit(15);
      }
```

例 **7–5** すべてのコネクション要求の処理　(続き)

```
        if (t_bind(conn_fd, (struct t_bind *) NULL,
              (struct t_bind *) NULL) == -1) {
           t_error("t_bind failed");
           exit(16);
        }
        if (t_accept(fd, conn_fd, calls[slot][i]) == -1) {
           if (t_errno == TLOOK) {
              t_close(conn_fd);
              return;
           }
           t_error("t_accept failed");
           exit(167);
        }
        t_free(calls[slot][i], T_CALL);
        calls[slot][i] = (struct t_call *) NULL;
        run_server(fd);
     }
}
```

それぞれのトランスポート終端について、未処理のコネクション要求の配列が走査されます。サーバーは要求ごとに応答用のトランスポート終端を開いて、終端にアドレスをバインドして、終端上で接続を受け入れます。現在の要求を受け入れる前に別のコネクション要求または切断要求を受信した場合、`t_accept`(3NSL) は失敗して、`t_errno` に `TLOOK` を設定します。保留状態のコネクション要求イベントまたは切断要求イベントがトランスポート終端に存在する場合は、未処理のコネクション要求を受け入れることはできません。

このエラーが発生した場合、応答用のトランスポート終端は閉じられ、`service_conn_ind` がすぐに返されます。 (現在のコネクション要求はあとで処理するために保存されます)。この動作によって、サーバーのメイン処理ループに入り、もう一度 `poll`(2) を呼び出すことによって、新しいイベントを発見できます。このように、ユーザーは複数のコネクション要求を待ち行列に入れることができます。

結果的にすべてのイベントが処理され、`service_conn_ind` はそれぞれのコネクション要求を順に受け取ることができます。

非同期ネットワーキング

この節では、XTI/TLI を使用して非同期ネットワーク通信を行う、リアルタイムアプリケーション用の手法について説明します。SunOS プラットフォームは、STREAMS の非同期機能と XTI/TLI ライブラリルーチンの非ブロッキングモードを組み合わせることによって、XTI/TLI イベントの非同期ネットワーク処理をサポートします。

ネットワークプログラミングモデル

ネットワーク転送はファイルやデバイスの入出力と同様に、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は、ファイルやデバイスの同期入出力と似ています。送信要求は `write`(2) インタフェースと同様に、メッセージをバッファーに入れたあとに返りますが、バッファー領域をすぐに確保できない場合、呼び出し元プロセスの実行を保留する可能性もあります。受信要求は `read`(2) インタフェースと同様に、必要なデータが到着するまで呼び出し元プロセスの実行を保留します。トランスポートサービスには保証された境界が存在しないため、同期ネットワーキングはほかのデバイスと関連しながらリアルタイムで動作する必要があるプロセスには不適切です。

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実現できます。コネクションが確立されるとき、データが送信されるとき、またはデータが受信されるとき、アプリケーションは非同期通知を要求できます。

非同期コネクションレスモードサービス

非同期コネクションレスモードネットワーキングを行うには、終端を非ブロッキングサービス向けに構成して、次に、非同期通知をポーリングするか、データが転送されたときに非同期通信を受信します。非同期通知が使用された場合、実際のデータの受信は通常シグナルハンドラ内で行われます。

終端の非同期化

終端を非同期サービス向けに構成するには、`t_open`(3NSL) を使用して終端を確立したあと、`t_bind`(3NSL) を使用してその識別情報を確立します。次に、`fcntl`(2) インタフェースを使用して、終端に `O_NONBLOCK` フラグを設定します。すると、バッファー領域をすぐに確保できない場合でも、`t_sndudata` (3NSL) への呼び出しは `-1` を返し、`t_errno` に `TFLOW` を設定します。同様に、データが存在しない場合でも、`t_rcvudata`(3NSL) への呼び出しは `-1` を返し、`t_errno` に `TNODATA` を設定します。

非同期ネットワーク転送

アプリケーションは `poll`(2) を使用して終端にデータが着信したかどうかを定期的に確認したり、終端がデータを受信するまで待機したりできますが、データが着信したときには非同期通知を受信する必要があります。`I_SETSIG` を指定して `ioctl`(2) コマンドを使用すると、終端にデータが着信したときに `SIGPOLL` シグナルがプロセスに送信されるように要求できます。アプリケーションは複数のメッセージが単一のシグナルとして送信されないように確認する必要があります。

次の例で、アプリケーションによって選択されたトランスポートプロトコル名は `protocol` 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);
    ...     /* バインディングアドレスを設定 */
    t_bind(fd, bind, bin

    /* 終端を非ブロッキングにする */
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);

    /* SIGPOLL 用のシグナルハンドラを確立 */
    signal(SIGPOLL, sigpoll);

    /* 受信データが使用できるときは SIGPOLL シグナルを要求 */
    ioctl(fd, I_SETSIG, S_INPUT | S_HIPRI);

    ...

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

    for (;;) {
        ... /* ud を初期化 */
        if (t_rcvudata(fd, &ud, &flags) < 0) {
            if (t_errno == TNODATA)
                break;  /* これ以上メッセージなし */
            ... /* ほかのエラー状態を処理 */
    }
    ... /* ud でメッセージを処理 */
}
```

非同期コネクションモードサービス

コネクションモードサービスでは、アプリケーションはデータ転送だけではなく、コネクションの確立そのものを非同期的に行うように設定できます。操作手順は、プロセスがほかのプロセスに接続しようとしているかどうか、または、プロセスがコネクションを待機しているかどうかによって異なります。

非同期的なコネクションの確立

プロセスはコネクションを非同期的に確立できます。プロセスはまず、接続用の終端を作成し、fcntl(2) を使用して、作成した終端を非ブロッキング操作向けに構成します。この終端はまた、コネクションレスデータ転送と同様に、コネクションが完了したときや以降のデータが転送されるときに非同期通知が送信されるようにも構成できます。次に、接続元プロセスは t_connect(3NSL) を使用して、転送設定を初期化します。それから、 t_rcvconnect(3NSL) を使用して、コネクションの確立を確認します。

非同期的なコネクションの使用

非同期的にコネクションを待機する場合、プロセスはまず、サービスアドレスにバインドされた非ブロッキング終端を確立します。poll(2) の結果または非同期通知によってコネクション要求の着信が伝えられた場合、プロセスは t_listen(3NSL) を使用してコネクション要求を取得します。コネクションを受け入れる場合、プロセスは t_accept(3NSL) を使用します。応答用の終端を別に非同期的にデータを転送するように構成する必要があります。

次の例に、非同期的にコネクションを要求する方法を示します。

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

    fd = .../* 非ブロッキング終端を確立 */

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

    /* コネクション要求は現在非同期に行われている */

    .../* コネクションが許可されたことの通知を受信 */
    t_rcvconnect(fd, &call);
```

次の例に、非同期的にコネクションを待機する方法を示します。

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

    fd = ... /* 非ブロッキング終端を確立 */

    .../* コネクション要求が到着したことの通知を受信 */
    call = (struct t_call *) t_alloc(fd, T_CALL, T_ALL);
    t_listen(fd, &call);

    .../* コネクションを許可するかどうかを決定 */
    res_fd = ... /* 応答のため非ブロッキング終端を確立 */
    t_accept(fd, res_fd, call);
```

非同期的に開く

アプリケーションはリモートホストからマウントされたファイルシステムや初期化に時間がかかっているデバイスにある通常ファイルを動的に開く必要がある場合があります。しかし、このようなファイルを開く要求を処理している間、アプリケーションはほかのイベントにリアルタイムで応答できません。この問題を解決するために、SunOS ソフトウェアはファイルを実際に開く作業を別のプロセスに任せて、ファイル記述子をリアルタイムプロセスに渡します。

ファイル記述子の転送

SunOS プラットフォームが提供する STREAMS インタフェースには、開いたファイル記述子をあるプロセスから別のプロセスに渡すメカニズムが用意されています。開いたファイル記述子を渡したいプロセスは、 コマンド引数 `I_SENDFD` を指定して `ioctl`(2) を使用します。ファイル記述子を取得したいプロセスは、コマンド引数 `I_RECVFD` を指定して `ioctl`(2) を使用します。

次の例では、親プロセスはまず、テストファイルについての情報を出力し、パイプを作成します。親プロセスは次に、テストファイルを開いて、開いたファイル記述子をパイプ経由で親プロセスに返すような子プロセスを作成します。そのあと、親プロセスは新しいファイル記述子の状態情報を表示します。

例 **7–6** 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)
{
```

例 7–6 ファイル記述子の転送　(続き)

```
    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);
}
```

状態遷移

次の表は、XTI/TLI 関連のすべての状態遷移を説明します。

XTI/TLI 状態

次の表に、XTI/TLI の状態遷移で使用される状態とサービスタイプを定義します。

表 7–1 XTI/TLI 状態遷移とサービスタイプ

状態	説明	サービスタイプ
T_UNINIT	初期化が行われていない - インタフェースの初期状態と終了状態	T_COTS、T_COTS_ORD、T_CLTS
T_UNBND	初期化されているが、バインドされていない	T_COTS 、T_COTS_ORD、T_CLTS

表 **7–1** XTI/TLI 状態遷移とサービスタイプ　(続き)

状態	説明	サービスタイプ
T_IDLE	コネクションが確立されていない	T_COTS 、T_COTS _ORD、T_CLTS
T_OUTCON	クライアントに対する送信コネクションが保留中	T_COTS、T_COTS_ORD
T_INCON	サーバーに対する受信コネクションが保留中	T_COTS、T_COTS_ORD
T_DATAXFER	データ転送	T_COTS、T_COTS_ORD
T_OUTREL	送信正常型解放 (正常型解放要求待ち)	T_COTS_ORD
T_INREL	受信正常型解放 (正常型解放要求の送信待ち)	T_COTS_ORD

送信イベント

次の表に示す送信イベントは、指定されたトランスポートルーチンから返される状態に対応しており、このようなイベントにおいて、これらのルーチンはトランスポートプロバイダに要求または応答を送信します。この表で示すイベントの一部 (accept など) は、発生した時点におけるコンテキストによって意味が変わります。これらのコンテキストは、次の変数の値に基づきます。

- *ocnt* – 未処理のコネクション要求の数
- *fd* – 現在のトランスポート終端のファイル記述子
- *resfd* – コネクションが受け入れられるトランスポート終端のファイル記述子

表 **7–2** 送信イベント

イベント	説明	サービスタイプ
opened	t_open(3NSL) が正常に終了した	T_COTS、 T_COTS_ORD、T _CLTS
bind	t_bind(3NSL) が正常に終了した	T_COTS、 T_COTS_ORD、T _CLTS
optmgmt	t_optmgmt(3NSL) が正常に終了した	T_COTS、 T_COTS_ORD、T _CLTS
unbind	t_unbind(3NSL) が正常に終了した	T_COTS、 T_COTS_ORD、T _CLTS
closed	t_close(3NSL) が正常に終了した	T_COTS、 T_COTS_ORD、T _CLT
connect1	同期モードの t_connect(3NSL) が正常に終了した	T_COTS 、T_COTS_ORD

表 7–2 送信イベント　(続き)

イベント	説明	サービスタイプ
connect2	非同期モードの t_connect(3NSL) で TNODATA エラーが発生したか、あるいは、切断要求がトランスポート終端に着信したことにより TLOOK エラーが発生した	T_COTS、 T_COTS_ORD
accept1	ocnt == 1、fd == resfd を指定した t_accept(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
accept2	ocnt== 1、fd!= resfd を指定した（t_accept 3NSL）が正常に終了した	T_COTS、 T_COTS_ORD
accept3	ocnt> 1を指定した t_accept(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
snd	t_snd(3NSL) が正常に終了した	T_COTS、 T_COTS_ORD
snddis1	ocnt <= 1 を指定した t_snddis (3NSL) が正常に終了した	T_COTS、T_COTS_ORD
snddis2	ocnt> 1 を指定した t_snddis(3NSL) が正常に終了した	T_COTS、T_COTS_ORD
sndrel	t_sndrel(3NSL) が正常に終了した	T_COTS_ORD
sndudata	t_sndudata(3NSL) が正常に終了した	T_CLTS

受信イベント

受信イベントは、指定されたルーチンが正常に終了したときに発生します。これらのルーチンは、トランスポートプロバイダからのデータやイベント情報を返します。ルーチンから返された値に直接関連付けられていない唯一の受信イベントは pass_conn であり、このイベントはコネクションがほかの終端に移行するときに発生します。終端で XTI/TLI ルーチンを呼び出さなくても、コネクションを渡している終端ではこのイベントが発生します。

次の表では、rcvdis イベントは、終端上の未処理のコネクション要求の数を示す ocnt の値によって区別されます。

表 7–3 受信イベント

イベント	説明	サービスタイプ
listen	t_listen(3NSL) が正常に終了した	T_COTS、 T_COTS_ORD
rcvconnect	t_rcvconnect(3NSL) が正常に終了した	T_COTS、 T_COTS_ORD

表 **7–3** 受信イベント　(続き)

イベント	説明	サービスタイプ
rcv	t_rcv(3NSL) が正常に終了した	T_COTS、 T_COTS_ORD
rcvdis1	onct <= 0 を指定した t_rcvdis (3NSL) rcvdis1t_rcvdis() が正常に終了した	T_COTS、 T_COTS_ORD
rcvdis2	ocnt == 1 を指定した t_rcvdis (3NSL) が正常に終了した	T_COTS、T_COTS_ORD
rcvdis3	ocnt> 1 を指定した t_rcvdis (3NSL) が正常に終了した	T_COTS、T_COTS_ORD
rcvrel	t_rcvrel(3NSL) が正常に終了した	T_COTS_ORD
rcvudata	t_rcvudata(3NSL) が正常に終了した	T_CLTS
rcvuderr	t_rcvuderr(3NSL) が正常に終了した	T_CLTS
pass_conn	渡されたコネクションを受信した	T_COTS、T_COTS_ORD

状態テーブル

状態テーブルは、XTI/TLI の状態遷移を示します。状態テーブルの列には現在の状態を、行には現在のイベントを、行と列の交差する部分では次に発生する状態を示しています。次に発生する状態が空の場合は、状態とイベントの組み合わせが無効であることを意味します。また次に発生する状態には、動作一覧が示されている場合もあります。動作は、指定された順序で実行する必要があります。

状態テーブルを見る場合は、次の点を理解してください。

- t_close(3NSL) はコネクション型トランスポートプロバイダ用に確立されたコネクションを終了します。コネクションの終了が正常型または放棄型のどちらで行われるかは、トランスポートプロバイダがサポートするサービスタイプによって決まります。詳細は、t_getinfo(3NSL) のマニュアルページを参照してください。
- トランスポートユーザーがシーケンス外のインタフェース呼び出しを発行すると、そのインタフェースは失敗し、t_errno は TOUTSTATE に設定されます。この状態は変更できません。
- t_connect(3NSL) のあとにエラーコード TLOOK または TNODATA が返されると、状態が変化する可能性があります。次の状態テーブルでは、XTI/TLI を正しく使用していることを前提としています。
- インタフェースのマニュアルページに特に指定されていない限り、ほかのトランスポートエラーによって状態が変化することはありません。
- サポートインタフェース t_getinfo(3NSL)、t_getstate(3NSL)、t_alloc (3NSL)、t_free(3NSL)、t_sync(3NSL)、t_look(3NSL)、および t_error (3NSL) は状態に影響しないため、この状態テーブルから除外されています。

次の表の状態遷移には、トランスポートユーザーが行う必要がある動作が記載されているものもあります。各動作は、次のリストから求められた数字によって表現されます。

- 未処理のコネクション要求の数に 0 を設定する
- 未処理のコネクション要求の数を 1 だけ増やす
- 未処理のコネクション要求の数を 1 だけ減らす
- 別のトランスポート終端にコネクションを渡す (t_accept(3NSL) のマニュアルページを参照)

次の表に、終端の確立の状態を示します。

表 **7–4** コネクション確立時における状態

イベント/状態	**T_UNINIT**	**T_UNBND**	**T_IDLE**
opened	T_UNBND		
bind		T_IDLE[1]	
optmgmt (TLI のみ)			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

次の表に、コネクションレスモードにおけるデータの転送の状態を示します。

表 **7–5** コネクションモードにおける状態 — その 1

イベント/状態	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
connect1	T_DATAXFER			
connect2	T_OUTCON			
rcvconnect		T_DATAXFER		
listen	T_INCON [2]		T_INCON [2]	
accept1			T_DATAXFER [3]	
accept2			T_IDLE [3] [4]	
accept3			T_INCON [3] [4]	
snd				T_DATAXFER
rcv				T_DATAXFER
snddis1		T_IDLE	T_IDLE [3]	T_IDLE

表 **7–5** コネクションモードにおける状態 — その 1　*(続き)*

イベント/状態	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
snddis2			T_INCON [3]	
rcvdis1		T_IDLE		T_IDLE
rcvdis2			T_IDLE [3]	
rcvdis3			T_INCON [3]	
sndrel				T_OUTREL
rcvrel				T_INREL
pass_conn	T_DATAXFER			
optmgmt	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
closed	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT

次の表に、コネクションモードにおけるコネクションの確立、コネクションの解放、およびデータの転送の状態を示します。

表 **7–6** コネクションモードにおける状態 — その 2

イベント/状態	T_OUTREL	T_INREL	T_UNBND
connect1			
connect2			
rcvconnect			
listen			
accept1			
accept2			
accept3			
snd		T_INREL	
rcv	T_OUTREL		
snddis1	T_IDLE	T_IDLE	
snddis2			
rcvdis1	T_IDLE	T_IDLE	
rcvdis2			
rcvdis3			
sndrel		T_IDLE	

表 7–6 コネクションモードにおける状態 — その 2　(続き)

イベント/状態	T_OUTREL	T_INREL	T_UNBND
rcvrel	T_IDLE		
pass_conn			T_DATAXFER
optmgmt	T_OUTREL	T_INREL	T_UNBND
closed	T_UNINIT	T_UNINIT	

次の表に、コネクションレスモードにおける状態を示します。

表 7–7 コネクションレスモードにおける状態

イベント/状態	T_IDLE
snudata	T_IDLE
rcvdata	T_IDLE
rcvuderr	T_IDLE

プロトコルに依存しない処理に関する指針

XTI/TLI が提供する一連のサービスは、多くのトランスポートプロトコルに共通であり、XTI/TLI を使用すると、アプリケーションはプロトコルに依存しない処理が可能になります。ただし、すべてのトランスポートプロトコルが XTI/TLI をサポートしているわけではありません。ソフトウェアをさまざまなプロトコル環境で実行する必要がある場合は、共通のサービスだけを使用してください。

次に示すサービスはすべてのトランスポートプロトコルに共通とは限らないので、注意してください。

- コネクションモードのサービスでは、すべてのトランスポートプロバイダで転送サービスデータユニット (TSDU) がサポートされるとは限りませんので、コネクションの際に論理的なデータ境界が保たれることを前提としないでください。
- プロトコルおよび実装に固有なサービスの制限は、t_open(3NSL) および t_getinfo(3NSL) のルーチンによって返されます。これらの制限に基づいてバッファーを割り当て、プロトコルに固有なトランスポートアドレスおよびオプションを格納してください。
- ユーザーデータを送信するときには、t_connect(3NSL) や t_snddis(3NSL) などのコネクション要求や切断要求を使用しないでください。これは、すべてのトランスポートプロトコルがこの方法を使用できるわけではないためです。

- t_listen(3NSL) で使用される t_call 構造体のバッファーには、コネクション確立時にクライアントが送信するデータを格納できるだけの大きさが必要です。現在のトランスポートプロバイダのアドレス、オプション、およびユーザーデータを格納できるように、t_alloc(3NSL) に T_ALL 引数を使用して最大のバッファーサイズを設定してください。
- クライアント側の終端では、t_bind(3NSL) のプロトコルアドレスを指定しないでください。トランスポートプロバイダがトランスポート終端に適切なプロトコルアドレスを割り当て、サーバーは、トランスポートプロバイダの名前空間を知らなくても、t_bind(3NSL) のプロトコルアドレスを取得できなければなりません。
- トランスポートアドレスの形式を仮定しないでください。また、トランスポートアドレスをプログラム内定数にしないでください。トランスポート選択の詳細については、第 8 章を参照してください。
- t_rcvdis(3NSL) に関連付けられた理由コードはプロトコルに依存します。プロトコルに依存しないことが重要である場合、これらの理由コードを使用しないでください。
- t_rcvuderr(3NSL) のエラーコードはプロトコルに依存します。プロトコルに依存しないことが重要である場合、これらのエラーコードを使用しないでください。
- プログラム内にデバイス名をコーディングしないでください。デバイスノードは、特定のトランスポートプロバイダを指定し、プロトコルに依存します。トランスポート選択の詳細については、第 8 章を参照してください。
- 複数のプロトコル環境で実行予定のプログラムでは、t_sndrel(3NSL) および t_rcvrel(3NSL) が提供するコネクションモードサービスの正常型解放機能 (オプション) を使用しないでください。正常型解放機能は、すべてのコネクション型トランスポートプロトコルでサポートされているわけではありません。この機能を使用すると、解放型システムと正常に通信できなくなることがあります。

XTI/TLI とソケットインタフェース

XTI/TLI とソケットとでは、同じタスクでも処理方法が異なります。どちらも機能的に似ているメカニズムとサービスを提供しますが、ルーチンや低レベルのサービスには 1 対 1 の互換性があるわけではありません。アプリケーションを移植しようとする場合は、XTI/TLI インタフェースとソケットベースのインタフェースとの間の類似点や相違点をよく知る必要があります。

トランスポートの独立性に関しては、次の問題があります。これらの問題は、RPC アプリケーションにも関係があります。

- 特権ポート (Privileged ports) – 特権ポートは、TCP/IP インターネットプロトコルのバークレー版ソフトウェア配布(BSD) を実装するための機能です。特権ポートは移植可能ではありません。特権ポートの概念は、トランスポートに依存しない環境ではサポートされません。

- 隠されたアドレス (Opaque addresses) – トランスポートに依存しない形態では、ホストを指定するアドレス部分とそのホスト上でサービスを指定するアドレス部分とを区別できません。ネットワークサービスのホストアドレスを認識できることを前提としたコードは必ず変更してください。
- ブロードキャスト(Broadcast) – トランスポートに依存しない形態では、ブロードキャストアドレスはありません。

ソケット関数と XTI/TLI 関数との対応関係

次の表に、XTI/TLI インタフェースとソケットインタフェースのおおまかな対応関係を示します。コメント列には、相違点を示します。コメントがない場合、インタフェースがほとんど同じであるか、または一方のインタフェースに相当する関数が存在しないことを意味します。

表 **7–8** TLI 関数とソケット関数の対応表

TLI インタフェース	ソケットインタフェース	説明
t_open(3NSL)	socket(3SOCKET)	
–	socketpair (3SOCKET)	
t_bind(3NSL)	bind(3SOCKET)	t_bind(3NSL) は、受信ソケットの待ち行列の深さを設定するが、bind(3SOCKET) は設定しない。ソケットの場合、待ち行列の長さは listen(3SOCKET) への呼び出しに指定する
t_optmgmt(3NSL)	getsockopt (3SOCKET) setsockopt (3SOCKET)	t_optmgmt(3NSL) はトランスポート層のオプションだけを管理する getsockopt (3SOCKET) および setsockopt (3SOCKET) は、トランスポート層のオプションだけではなく、ソケット層および任意のプロトコル層のオプションも管理する
t_unbind(3NSL)	–	
t_close(3NSL)	close(2)	
t_getinfo(3NSL)	getsockopt (3SOCKET)	t_getinfo(3NSL) はトランスポートに関する情報を返す。getsockopt(3SOCKET) はトランスポートおよびソケットに関する情報を返すことができる

表 **7–8** TLI 関数とソケット関数の対応表　(続き)

TLI インタフェース	ソケットインタフェース	説明
t_getstate(3NSL)	-	
t_sync(3NSL)	-	
t_alloc(3NSL)	-	
t_free(3NSL)	-	
t_look(3NSL)	-	SO_ERROR オプションを指定した getsockopt(3SOCKET) は t_look(3NSL) t_look() と同じ種類のエラー情報を返す
t_error(3NSL)	perror(3C)	
t_connect(3NSL)	connect(3SOCKET)	connect(3SOCKET) を呼び出す前に、ローカルの終端をバインドする必要はない。t_connect(3NSL) を呼び出す前には、終端をバインドする。connect (3SOCKET) をコネクションレス終端で実行すると、データグラムのデフォルト宛先アドレスを設定できる。connect(3SOCKET) を使用すると、データを送信できる
t_rcvconnect(3NSL)	-	
t_listen(3NSL)	listen(3SOCKET)	t_listen(3NSL) は接続指示を待つ。listen(3SOCKET) は待ち行列の深さを設定する
t_accept(3NSL)	accept(3SOCKET)	
t_snd(3NSL)	send(3SOCKET)	
	sendto(3SOCKET)	
	sendmsg(3SOCKET)	sendto(3SOCKET) および sendmsg (3SOCKET) はデータグラムモードでもコネクションモードでも機能する
t_rcv(3NSL)	recv(3SOCKET)	
	recvfrom(3SOCKET)	
	recvmsg(3SOCKET)	recvfrom(3SOCKET) および recvmsg (3SOCKET) はデータグラムモードでもコネクションモードでも機能する。
t_snddis(3NSL)	-	
t_rcvdis(3NSL)	-	
t_sndrel(3NSL)	shutdown(3SOCKET)	
t_rcvrel(3NSL)	-	

表 7–8 TLI 関数とソケット関数の対応表　(続き)

TLI インタフェース	ソケットインタフェース	説明
t_sndudata(3NSL)	sendto(3SOCKET)	
	recvmsg(3SOCKET)	
t_rcvuderr(3NSL)	-	
read(2)、write(2)	read(2)、 write(2)	XTI/TLI では、read(2) または write(2) を呼び出す前に tirdwr(7M) モジュールをプッシュしておく必要がある。ソケットでは、read(2) または write(2) を呼び出すだけでよい

XTI インタフェースへの追加

XNS 5 (Unix98) 標準に新規の XTI インタフェースが導入されました。これらの XTI インタフェースについて、次に簡単に説明します。詳細については、関連するマニュアルページを参照してください。なお、TLI ユーザーはこれらのインタフェースを使用できません。分散および集中データ転送インタフェースは次のとおりです。

t_sndvudata(3NSL)	1 つまたは複数の非連続バッファー上のデータユニットを送信する
t_rcvvudata(3NSL)	1 つまたは複数の非連続バッファーにデータユニットを受信する
t_sndv(3NSL)	コネクション時に、1 つまたは複数の非連続バッファー上のデータまたは優先データを送信する
t_rcvv(3NSL)	コネクションを経由して受信したデータまたは優先データを、1 つまたは複数の非連続バッファーに格納する

XTI ユーティリティインタフェース t_sysconf(3NSL) は構成可能な XTI 変数を取得します。t_sndreldata(3NSL) インタフェースは、ユーザーデータを使用して正常型解放を発行したり、正常解放に応答したりします。t_rcvreldata(3NSL) は、正常型解放の指示やユーザーデータが含まれる確認を受信します。

注 – 追加のインタフェースである t_sndreldata(3NSL) および t_rcvreldata (3NSL) は「最小 OSI」と呼ばれる特定のトランスポートだけで使用されますが、最小 OSI は Solaris プラットフォームではサポートされません。これらのインタフェースは、インターネットトランスポート (TCP または UDP) と併用することはできません。

第 8 章

トランスポート選択と名前からアドレスへのマッピング

この章では、トランスポートの選択およびネットワークアドレスの解決方法を示します。また、アプリケーションが使用できる通信プロトコルを指定できるようにするインタフェースについて説明します。さらに、名前からネットワークアドレスに直接マッピングする追加機能についても取り上げます。

注 – この章では、「ネットワーク」と「トランスポート」という用語はどちらも同じ意味で使用されます。 この 2 つの用語は、OSI 参照モデルのトランスポート層に準拠するプログラム可能なインタフェースを指します。「ネットワーク」という用語は、何らかの電子媒体を介して接続できる物理的なコンピュータの集まりを指す場合にも使用されます。

トランスポート選択

注意 – この章で取り上げるインタフェースはマルチスレッドに対して安全です。「マルチスレッドに対して安全」ということは、トランスポート選択機能インタフェース呼び出しを行うアプリケーションをマルチスレッド対応アプリケーション内で自由に使用できることを意味します。これらのインタフェース呼び出しは再入可能ではないので、スケーラビリティは直線的でありません。

分散アプリケーションを各種のプロトコルに移植可能にするには、分散アプリケーションでトランスポートサービスの標準インタフェースを使用する必要があります。トランスポート選択サービスが提供するインタフェースを使用すると、アプリケーションは、使用するプロトコルを選択できます。このインタフェースによって、プロトコルと媒体に依存しないアプリケーションが実現されます。

トランスポート選択により、クライアントアプリケーションは、クライアントがサーバーとの通信を確立するまでに、どのトランスポートを使用できるかを簡単に試すことができます。トランスポート選択を使用すると、サーバーアプリケーションは複数のトランスポート上で要求を受け入れることができ、複数のプロトコルを経由して通信できます。どのトランスポートが使用できるかは、ローカルなデフォルトシーケンスで指定された順序、またはユーザーが指定した順序で試すことができます。

使用可能なトランスポートのうち、どれを選択するかを決定するのは、アプリケーションの役割です。トランスポート選択メカニズムを使用すると、選択が統一的な方法で簡単に行えます。

名前からアドレスへのマッピング

名前からアドレスへのマッピングを行うと、使用されるトランスポートに関係なく、アプリケーションは指定のホスト上で実行されるサービスのアドレスを取得できます。名前からアドレスへのマッピングでは、次のインタフェースを使用します。

`netdir_getbyname`(3NSL)	ホスト名およびサービス名を一連のアドレスに対応づける
`netdir_getbyaddr`(3NSL)	アドレスを、ホスト名およびサービス名に対応づける
`netdir_free`(3NSL)	名前からアドレスへの変換ルーチンによって割り当てられた構造体を解放する
`taddr2uaddr`(3NSL)	アドレスを変換し、トランスポートに依存しないアドレスの文字表現を返す
`uaddr2taddr`(3NSL)	汎用アドレスを `netbuf` 構造体に変換する
`netdir_options`(3NSL)	ブロードキャストアドレス、TCP や UDP の予約ポート機能など、トランスポート固有の機能へのインタフェースをとる
`netdir_perror`(3NSL)	名前からアドレスにマッピングするルーチンの 1 つが失敗した理由を示すメッセージを `stderr` に表示する
`netdir_sperror`(3NSL)	名前からアドレスにマッピングするルーチンの 1 つが失敗した理由を示すエラーメッセージを含む文字列を返す

各ルーチンの最初の引数では、トランスポートを示す netconfig(4) 構造体を指定します。これらのルーチンは、netconfig(4) 構造体内にあるディレクトリルックアップ用のライブラリパスの配列を使用して、変換が正常終了するまで各パスを呼び出します。

表 8–1 に、名前からアドレスへのマッピング用ライブラリを示します。178 ページの「名前からアドレスへのマッピングルーチンの使用」で説明しているルーチンは、netdir(3NSL) のマニュアルページに定義されています。

注 – tcpip.so、switch.so、および nis.so というライブラリは、Solaris 環境ではすでに廃止されました。 この変更の詳細については、 nsswitch.conf(4) のマニュアルページおよび gethostbyname(3NSL) マニュアルページの NOTES セクションを参照してください。

表 8–1 名前からアドレスへのマッピングを行うライブラリ

ライブラリ	トランスポートファミリ	説明
-	inet	プロトコルファミリ inet のネットワークでは、名前からアドレスへのマッピングはファイル nsswitch.conf(4) 内にある *hosts* と *services* のエントリに基づくネームサービス切り替えによって行われる。inet 以外のファミリを使用するネットワークでは、「-」を指定すると、名前からアドレスへのマッピング機能が存在しないことを示す
straddr.so	loopback	ループバックトランスポートのように、文字列をアドレスとして受け入れる任意のプロトコルの、名前からアドレスにマッピングするルーチンが含まれる

straddr.so ライブラリ

straddr.so ライブラリで使用される名前からアドレスへの変換ファイルは、システム管理者が作成します。システム管理者はまた、このような変換ファイルを保守します。straddr.so ファイルには、/etc/net/*transport-name*/hosts と /etc/net/*transport-name*/services があります。*transport-name* は、文字列アドレスを受け入れるトランスポートのローカル名であり、/etc/netconfig ファイルの *network ID* フィールドに指定されています。たとえば、ticlts のホストファイルは、/etc/net/ticlts/hosts となり、ticlts のサービスファイルは、/etc/net/ticlts/services となります。

ほとんどの文字列アドレスは「ホスト」と「サービス」を区別しません。しかし、文字列をホスト部分とサービス部分とに分けると、ほかのトランスポートとの間で一貫性が保たれます。/etc/net/*transport-name*/hosts ファイルには、次のようにホストアドレスと見なされるテキスト文字列に続いて、ホスト名を定義します。

```
joyluckaddr      joyluck

carpediemaddr    carpediem

thehopaddr       thehop

pongoaddr        pongo
```

ルックバックトランスポートは自分が含まれているホスト以外では実行できないため、ほかのホストを記述しても意味がありません。

/etc/net/*transport-name*/services には、サービス名に続いて、サービスアドレスを特定する文字列を定義します。

```
rpcbind    rpc
listen    serve
```

このルーチンは、ホストアドレス、ピリオド (.)、およびサービスアドレスを結合して、完全な文字列アドレスを作成します。たとえば、pongo での listen サービスのアドレスは、pongoaddr.serve になります。

このライブラリを使用するトランスポート上で、あるアプリケーションが特定のホスト上のサービスアドレスを要求するとき、ホスト名が /etc/net/*transport*/hosts に定義されていなければなりません。また、サービス名も /etc/net/*transport*/services に定義されていなければなりません。どちらか一方でも欠けると、名前からアドレスへの変換が失敗します。

名前からアドレスへのマッピングルーチンの使用

この節では、使用できるマッピングルーチンについて簡単に説明します。ルーチンは、ネットワーク名を返すか、または対応するネットワークアドレスにネットワーク名を変換します。netdir_getbyname(3NSL)、netdir_getbyaddr(3NSL)、および taddr2uaddr(3NSL) はデータへのポインタを返しますが、これらのポインタは netdir_free(3NSL) 呼び出しで解放する必要があります。

```
int netdir_getbyname(struct netconfig *nconf,
        struct nd_hostserv *service, struct nd_addrlist **addrs);
```

netdir_getbyname(3NSL) は *service* に指定されたホスト名とサービス名を、*nconf* で指定されたトランスポートに一致するアドレスセットに対応づけます。nd_hostserv と nd_addrlist の各構造体は、netdir(3NSL) のマニュアルページに定義されています。アドレスへのポインタは、*addrs* に返されます。

使用可能なすべてのトランスポート上で、ホストおよびサービスのすべてのアドレスを取得するには、getnetpath(3NSL) または getnetconfig(3NSL) のいずれかで返される各 netconfig(4) 構造体を使用して netdir_getbyname(3NSL) を呼び出します。

```
int netdir_getbyaddr(struct netconfig *nconf,
        struct nd_hostservlist **service, struct netbuf *netaddr);
```

netdir_getbyaddr(3NSL) は、アドレスをホスト名とサービス名に対応づけます。このインタフェースは、*netaddr* に指定されたアドレスを使用して呼び出され、ホスト名とサービス名のペアのリストを *service* に返します。nd_hostservlist 構造体は、netdir(3NSL) に定義されています。

```
void netdir_free(void *ptr, int struct_type);
```

netdir_free(3NSL) ルーチンは、名前からアドレスへの変換ルーチンによって割り当てられた構造体を解放します。次の表に、パラメータに使用できる値を示します。

表 **8–2** netdir_free(3NSL) ルーチン

struct_type	ptr
ND_HOSTSERV	nd_hostserv 構造体へのポインタ
ND_HOSTSERVLIST	nd_hostservlist 構造体へのポインタ
ND_ADDR	netbuf 構造体へのポインタ
ND_ADDRLIST	nd_addrlist 構造体へのポインタ

```
char *taddr2uaddr(struct netconfig *nconf, struct netbuf *addr);
```

taddr2uaddr(3NSL) は、*addr* が指すアドレスを変換し、アドレスのトランスポートに依存しない文字列表現を返します。この文字列表現のことを「汎用アドレス」と呼びます。*nconf* には、アドレスが有効なトランスポートを指定します。汎用アドレスは、free(3C) で解放できます。

```
struct netbuf *uaddr2taddr(struct netconfig *nconf, char *uaddr);
```

uaddr が指す汎用アドレスは、netbuf 構造体に変換されます。*nconf* には、アドレスが有効なトランスポートを指定します。

```
int netdir_options(const struct netconfig *config,
          const int option, const int fildes, char *point_to_args);
```

netdir_options(3NSL) は、ブロードキャストアドレス、TCP や UDP の予約ポート機能など、トランスポート固有の機能へのインタフェースを提供します。 *nconf* にはトランスポートを指定し、*option* にはトランスポート固有の動作を指定します。*fd* の値は *option* の値によって指定するかどうかが決まります。4 つ目の引数は、操作固有のデータを指します。

次の表に、option に使用できる値を示します。

表 **8–3** netdir_options に指定できる値

オプション	説明
ND_SET_BROADCAST	ブロードキャスト用のトランスポートを設定する (トランスポートがブロードキャスト機能をサポートしている場合)

表 **8–3** netdir_options に指定できる値　(続き)

オプション	説明
ND_SET_RESERVEDPORT	アプリケーションが予約ポートにバインドできるようにする (トランスポートがそのようなバインドを許可している場合)
ND_CHECK_RESERVEDPORT	アドレスが予約ポートに対応しているかどうかを検証する (トランスポートが予約ポートをサポートしている場合)
ND_MERGEADDR	ローカルに意味のあるアドレスを、クライアントホストが接続できるアドレスに変換する

netdir_perror(3NSL) ルーチンは、名前からアドレスにマッピングするルーチンの 1 つが失敗した理由を示すメッセージを stderr に表示します。

```
void netdir_perror(char *s);
```

netdir_sperror(3NSL) ルーチンは、名前からアドレスにマッピングするルーチンの 1 つが失敗した理由を示すエラーメッセージが含む文字列を返します。

```
char *netdir_sperror(void);
```

次の例に、ネットワーク選択および名前からアドレスへのマッピングを示します。

例 **8–1** ネットワーク選択および名前からアドレスへのマッピング

```
#include <netconfig.h>
#include <netdir.h>
#include <sys/tiuser.h>

struct nd_hostserv nd_hostserv;   /* ホストとサービスの情報 */
struct nd_addrlist *nd_addrlistp; /* サービスのアドレスリスト */
struct netbuf *netbufp;           /* サービスのアドレス */
struct netconfig *nconf;          /* トランスポート情報 */
int i;                            /* アドレスの数 */
char *uaddr;                      /* サービスの汎用アドレス */
void *handlep;                    /* ネットワーク選択用のハンドル */
/*
 * 「gandalf」というホスト上の「日付」サービスを参照する
 * ホスト構造体の設定
 */
nd_hostserv.h_host = "gandalf";
nd_hostserv.h_serv = "date";
/*
 * ネットワーク選択機構の初期化
 */
if ((handlep = setnetpath()) == (void *)NULL) {
   nc_perror(argv[0]);
   exit(1);
}
/*
 * トランスポートプロバイダ間のループ
```

例 **8–1** ネットワーク選択および名前からアドレスへのマッピング　(続き)

```
 */
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL)
{
   /*
    * netconfig 構造体で指定したトランスポートプロバイダに
    * 関連付けられた情報を出力する。
    */
   printf("Transport provider name: %s\n", nconf->nc_netid);
   printf("Transport protocol family: %s\n", nconf->nc_protofmly);
   printf("The transport device file: %s\n", nconf->nc_device);
   printf("Transport provider semantics: ");
       switch (nconf->nc_semantics) {
   case NC_TPI_COTS:
      printf("virtual circuit\n");
      break;
   case NC_TPI_COTS_ORD:
      printf("virtual circuit with orderly release\n");
      break;

   case NC_TPI_CLTS:
      printf("datagram\n");
      break;
   }
   /*
    * netconfig 構造体で指定したトランスポートプロバイダ
    * を経由して、「gandalf」というホスト上の「日付」
    * サービスのアドレスの取得
    */
   if (netdir_getbyname(nconf, &nd_hostserv, &nd_addrlistp) != ND_OK) {
      printf("Cannot determine address for service\n");
      netdir_perror(argv[0]);
      continue;
   }
   printf("<%d> addresses of date service on gandalf:\n",
      nd_addrlistp->n_cnt);
   /*
    * 現在のトランスポートプロバイダ上で、「gandalf」
    * というホスト上にある「日付」サービスの全アドレスの出力
    */
   netbufp = nd_addrlistp->n_addrs;
   for (i = 0; i < nd_addrlistp->n_cnt; i++, netbufp++) {
      uaddr = taddr2uaddr(nconf,netbufp);
      printf("%s\n",uaddr);
      free(uaddr);
   }
       netdir_free( nd_addrlistp, ND_ADDRLIST );

}
endnetconfig(handlep);
```

第 9 章

リアルタイムプログラミングと管理

この章では、SunOS で実行するリアルタイム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、リアルタイムアプリケーションを書いた経験があるプログラマや、リアルタイム処理と Solaris システムに詳しい管理者を対象として書かれています。

次の内容について説明します。

- 187 ページの「スケジューリング」では、リアルタイムアプリケーションのスケジューリングの必要性について説明します。
- 198 ページの「メモリーのロック」
- 206 ページの「非同期ネットワーキング」

リアルタイムアプリケーションの基本的な規則

リアルタイム応答は、一定の条件を満たした場合に保証されます。この節では、その条件を明確にし、設計上の重大な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遅くす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が動かなくなるものもあります。それほど重大ではないエラーには、優先順位の逆転やシステムの過負荷などがあります。

Solaris のリアルタイムプロセスには、次のような特長があります。

- 187 ページの「スケジューリング」で説明しているように、リアルタイムスケジューリングクラスで動作します。
- 198 ページの「メモリーのロック」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ーをロックします。

- 185 ページの「共有ライブラリ」で説明しているように、静的にリンクされたプログラム、または動的バインド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、リアルタイム処理を単一スレッド化プロセスとして説明していますが、マルチスレッド化プロセスにも当てはまります。マルチスレッド化プロセスについての詳細は、『マルチスレッドプログラミング』を参照してください。スレッドのリアルタイムスケジューリングを保証するには、スレッドはバインドされたスレッドとして作成される必要があります。さらに、スレッドの LWP は `RT` (リアルタイム) スケジューリングクラスで実行される必要があります。メモリーのロックと初期の動的バインドは、プロセス内のすべてのスレッドについて有効です。

あるプロセスが最も高い優先順位を持つとき、このプロセスは、ディスパッチ応答時間内にプロセッサを取得して実行できることが保証されます。詳細は、187 ページの「ディスパッチ応答時間」を参照してください。優先順位が最も高い実行可能なプロセスである限り、このプロセスは実行を継続します。

リアルタイムプロセッサは、システム上のほかのイベントのために、プロセッサの制御を失うことがあります。リアルタイムプロセッサはまた、システム上のほかのイベントのために、プロセッサの制御を取得できないこともあります。例として、割り込みなどの外部イベント、資源不足、同期入出力などの外部イベント待ち、より優先順位が高いプロセスによる横取りなどのイベントが挙げられます。

リアルタイム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。

応答時間を低下させる要因

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低下させます。応答時間の低下が大きいと、アプリケーションがクリティカルなデッドラインに間に合わないことがあります。

リアルタイム処理は、システム上でリアルタイムアプリケーションを実行しているほかの有効なアプリケーションの操作を損なうこともあります。リアルタイムプロセスの優先順位は高いため、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。この状況では、表示やキーボードの応答時間など、対話型の動作性が極端に低下することがあります。

同期入出力呼び出し

SunOS 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を意味します。時間制限が非常に長いプログラムセグメントでも、同期入出力は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アップしてしまいます。

よくあるアプリケーションの誤りは、エラーメッセージのテキストをディスクから取得するときに、入出力を実行することです。エラーメッセージのテキストをディスクから取得するには、独立したプロセスまたはスレッドから入出力を実行する必要があります。また、この独立したプロセスまたはスレッドはリアルタイムで動作していないものにしてください。

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。あるプロセスのアクションが原因で発生したハードウェア割り込みサービスは、そのプロセスのグループに設定されている優先順位を継承しません。したがって、優先順位が高いリアルタイムプロセスを制御しているデバイスに必ずしも、優先順位が高い割り込み処理が割り当てられるとは限りません。

共有ライブラリ

タイムシェアリングプロセスでは、動的にリンクされる共有ライブラリを使用すると、メモリー量を大幅に節約できます。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。動的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。

リアルタイムプログラムは、プログラムを起動するときに、環境変数 `LD_BIND_NOW` に `NULL` 以外の値を設定できます。環境変数 `LD_BIND_NOW` に `NULL` 以外の値を設定すると、共有ライブラリを使用しても動的バインドは行われません。また、すべての動的リンクは、プログラムの実行前にバインドが行なわれます。詳細は、『リンカーとライブラリ』を参照してください。

優先順位の逆転

リアルタイム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、リアルタイムプロセスをブロックできます。優先順位の逆転は、優先順位が高いプロセスが優先順位が低いプロセスによってブロックされることで起こります。「ブロック化」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たなければならない状態のことです。ブロック化に時間がかかると、リアルタイムプロセスはデッドラインに間に合わないことがあります。

次の図に、優先順位が高いプロセスが共有資源を要求する例を示します。優先順位が低いプロセスが保持している共有資源を優先順位が中間のプロセスが横取りしているので、優先順位が高いプロセスはブロック化されています。中間のプロセスは、いくつ関与していてもかまいません。優先順位が中間のプロセスはすべて、優先順位が低いプロセスのクリティカルな部分と同様に、実行を終了する必要があります。すべての実行が終了するまでには、しばらく時間がかかることがあります。

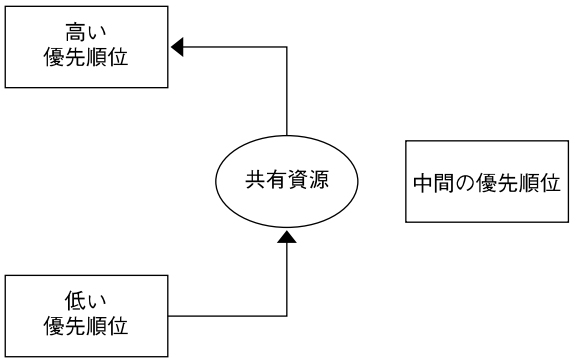


図 **9–1** 制限されない優先順位の逆転

この問題とその対処方法については、『マルチスレッドのプログラミング』の「相互排他ロック属性」の節で説明しています。

スティッキロック

ページのロックカウントが 65535 (0xFFFF) に達すると、そのページは永久にロックされます。値 0xFFFF は実装によって定義されており、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クされたページのロックは解除できません。

ランナウェイリアルタイムプロセス

ランナウェイリアルタイムプロセスは、システムを停止させることがあります。ランナウェイリアルタイムプロセスはまた、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もあります。

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 を押します。Stop-A は何回も押す必要があることもあります。Stop-A を押してもランナウェイプロセスが停止しない場合、電源を切ってからしばらく待ち、もう一度電源を入れ直してください。ランナウェイプロセスが SPARC 以外のシステム上にある場合も、電源を切ってからしばらく待ち、もう一度電源を入れ直してください。

優先順位が高いリアルタイムプロセスが CPU の制御を放棄しない場合、無限ループを強制的に終了させないと、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 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位が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。

非同期入出力の動作

非同期入出力操作は必ずしも、カーネルの待ち行列に入った順番で実行されるとは限りません。非同期入出力操作はまた、実行された順序で呼び出し側に返されるとも限りません。

aioread(3AIO) を繰り返して高速に呼び出すことができるように単一のバッファを指定している場合、バッファの状態は確定されません。バッファの状態が確定されないのは、最初の呼び出しが行われてから最後の呼び出しの結果が呼び出し側にシグナル送信されるまでの間です。

個々の aio_result_t 構造体は一度に1 つの非同期操作だけに使用できます。この非同期操作は読み取りでも書き込みでもかまいません。

リアルタイムファイル

SunOS 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) の操作が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

リアルタイム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。リアルタイム環境では、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 のリアルタイムスケジューラ、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。

ディスパッチ応答時間

リアルタイム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、リアルタイム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い、優先順位の概念に制限があります。したがって、リアルタイムアプリケーションには適しません。リアルタイムア

プリケーションは、プロセスの優先順位が絶対的なものとして受け取られるスケジューリングクラスを必要とします。リアルタイムアプリケーションはまた、プロセスの優先順位がアプリケーションの明示的な操作でしか変更されないスケジューリングクラスを必要とします。

「ディスパッチ応答時間」とは、プロセスの操作開始の要求にシステムが応答するまでの時間を指します。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、ディスパッチ応答時間を制限したリアルタイムアプリケーションを開発できます。

次の図に、あるアプリケーションが外部イベントからの要求に応答するまでの時間を示します。

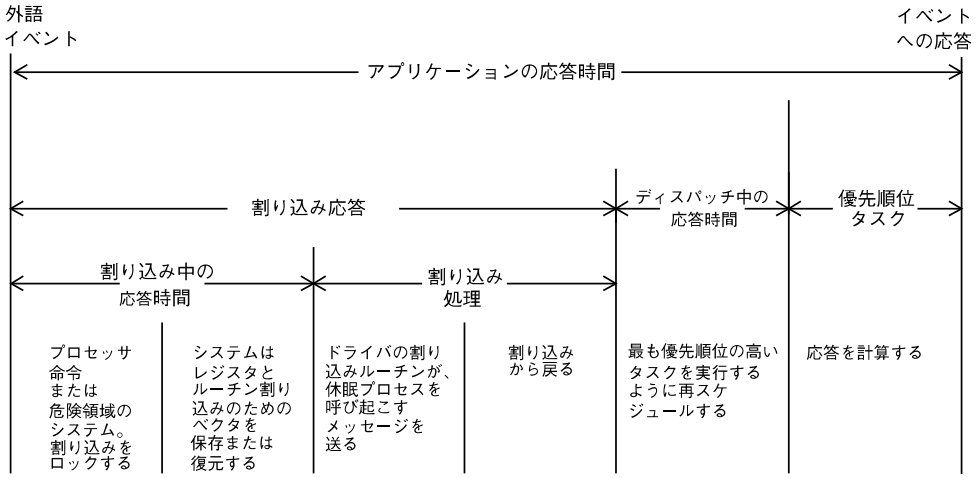


図 **9–2** アプリケーション応答時間

全体のアプリケーション応答時間には、割り込み応答時間、ディスパッチ応答時間、およびアプリケーションの応答時間が含まれます。

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応答時間とデバイスドライバ独自の割り込み処理時間が含まれます。割り込み応答時間は、システムが割り込みを無効にして実行する必要がある最長の間隔によって決まります。SunOS では、この時間を最小限にするために、通常はプロセッサの割り込みレベルを上げる必要がない同期プリミティブを使用しています。

割り込み処理中、ドライバの割り込みルーチンはまず、優先順位が高いプロセスを呼び起こし、そのプロセスが終了すると戻ります。割り込まれたプロセスよりも優先順位が高いプロセスが現在ディスパッチ可能であることを検出すると、システムは (優先順位が高い) プロセスをディスパッチします。優先順位が低いプロセスから高いプロセスへコンテキストを切り換える時間は、ディスパッチ応答時間に含まれます。

図 9–3に、システムの内部イベントのディスパッチ応答時間とアプリケーション応答時間を示します。アプリケーション応答時間は、システムが内部イベントに応答するまでに必要な時間のことです。内部イベントのディスパッチ応答時間とは、あるプロセスが優先順位がより高いプロセスを呼び起こすまでに必要な時間のことです。このディスパッチ応答時間には、システムが優先順位がより高いプロセスをディスパッチするまでに必要な時間も含まれます。

アプリケーション応答時間とは、ドライバが次の作業を完了するまでに必要な時間のことです。つまり、優先順位がより高いプロセスを呼び起こし、優先順位が低いプロセスから資源を解放し、優先順位がより高いタスクをスケジュールし直し、応答を計算し、タスクをディスパッチすることです。

ディスパッチ応答時間のインターバルの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応答時間は増えますが、ディスパッチ応答時間の測定には影響を与えません。したがって、この処理はディスパッチ応答時間の保証には制限されません。

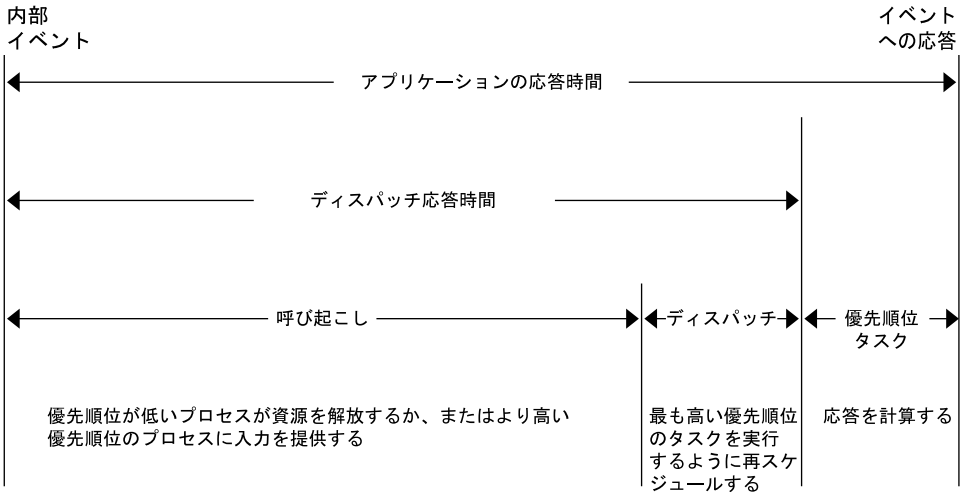


図 **9–3** 内部ディスパッチ応答時間

リアルタイム SunOS に用意されている新しいスケジューリング手法を使用すると、システムのディスパッチ応答時間を指定された範囲に限定できます。次の表に示すように、プロセス数を制限するとディスパッチ応答時間が改善されます。

表 **9–1** リアルタイムシステムディスパッチ応答時間

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	動作中のプロセスが 16 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

表 **9–1** リアルタイムシステムディスパッチ応答時間　*(続き)*

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 5	0.3 ミリ秒未満	0.3 ミリ秒
Ultra™ 1-167	0.15 ミリ秒未満	0.15 ミリ秒未満

スケジューリングクラス

SunOS 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、リアルタイム (RT)、システム (sys)、および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、リアルタイムプロセスが sys や TS のプロセスよりも優先されます。システム管理者は、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。

次の図に、SunOS カーネルから見たクラスの概念を示します。

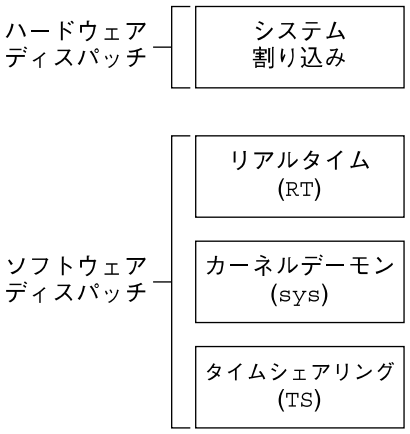


図 **9–4** スケジューリングクラスのディスパッチ優先順位

ハードウェア割り込みは優先順位が最も高いので、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。

リアルタイムプロセス (RT) 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 (time quantum) 値を持ちます。RT プロセスは、厳密にこれらのパラメー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され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期限が切れるまで実行されます。有限タイムカンタムを持つプロセスはまた、入出力イベントを待つ間ブロックされるか、またはより高い優先順位を持つ実行可能なリアルタイム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、ブロックされるか、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、スワッピング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。通常のプロセスのクラスは SYS クラスには変更できません。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動的にスケジュールされます。TS スケジューラは、次の値に基づいて、ラウンドロビン方式でコンテキストを切り換えることによって、すべてのプロセスに平等な機会を提供します。

- タイムスライス値
- プロセス履歴 (プロセスが最後に休眠状態に入ったときを記録する)
- CPU 使用率

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。

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。exec(2) を実行しても、プロセスのスケジューリングクラスと属性は変わりません。

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。カーネルはクラスから独立しており、優先順位が最も高いプロセスを待ち行列内から取り出します。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。

次の図に示すように、各クラスのアルゴリズムは独自の方法で優先順位が最も高いプロセスを選択して、グローバル実行待ち行列に入れます。

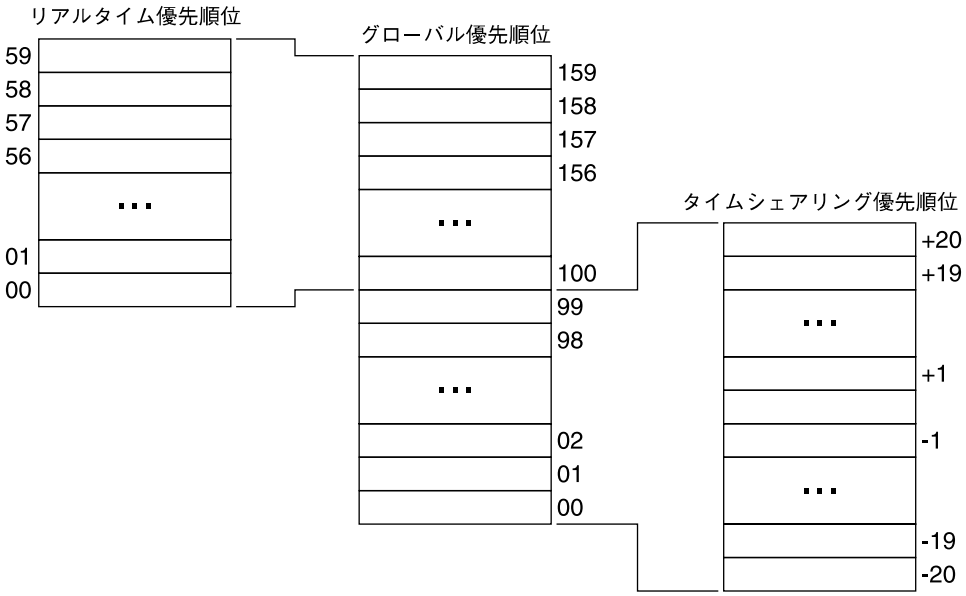


図 **9–5** カーネルディスパッチ待ち行列

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マッピングセットは、0 で始まっていたり、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 までの範囲です。このようなグローバル優先順位の値はカーネルの 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。リアルタイム (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直線的にリンクしたリストです。各プロセスには起動時に、クラス固有な情報が付けられます。プロセスは、グローバル優先順位に基づく順番で、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、メモリー管理情報、レジスタ、スタックとともに、プロセスのコンテキストがメモリー内に割り当てられます。コンテキストマッピングが完了したあと、実行が始まります。メモリー管理情報はハードウェアレジスタの形式をしており、現在実行中のプロセスのために仮想メモリー変換を実行するときに必要となるデータが入っています。

プロセスの横取り

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、カーネルはコンピュータ操作に割り込んで強制的にコンテキストを切り換え、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ます。すると、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を送信し、デバイスドライバの処理に入ります。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争奪する場合があります。たとえば、優先順位の高いリアルタイムプロセスが優先順位の低いプロセスが持っているリソースを待っていると仮定します。このとき、優先順位の低いプロセスがそのリソースを解放すると、カーネルは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逆転

優先順位の逆転は、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クされた場合に生じます。SunOS のカーネルで相互排他ロックなどの同期プリミティブを使用すると、優先順位の逆転につながることがあります。

「ブロック化」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。このブロック化が継続すると、使用レベルが低いものでもデッドラインに間に合わないことがあります。

相互排他ロックによって優先順位が逆転する問題については、SunOS 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クすると、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。この継承のため、プロセスがブロック化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動作であり、プログラマがシステムコールやインタフェース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ーレベルのプロセスは優先順位の逆転を生じることがあります。

ユーザー優先順位の逆転

ユーザー優先順位が逆転する問題とその対処方法については、『マルチスレッドのプログラミング』の「相互排他ロック属性 」の項を参照してください。

スケジューリングを制御するインタフェース呼び出し

次に、プロセスのスケジューリングを制御するインタフェース呼び出しを説明します。

priocntl

動作中のクラスのスケジューリング制御は、priocntl(2) で処理します。fork(2) や exec(2) を実行した場合、クラスの属性は、 優先順位の制御に必要なスケジューリングパラメータやアクセス権とともに継承されます。この継承は、RT クラスと TS クラスの両方に当てはまります。

priocntl(2) は、システムコールが適用されるリアルタイムプロセス、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です。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(2) 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。 呼び出し側プロセスの実 ID または実効 ID は、対象となるプロセスの実 ID または実効 ID と一致するか、あるいは、スーパーユーザー特権を持つ必要があります。

PC_GETCID　このコマンドは、認識可能なクラス名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返されます。

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返されます。

PC_GETPARMS　このコマンドは、指定されたプロセスのうち、1 つのプロセス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ータを返します。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ータだけを返します。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、指定されたプロセス (複数可) のスケジューリングクラスまたはクラス固有のスケジューリングパラメータを設定します。

その他のインタフェース呼び出し

`sched_get_priority_max`
　指定された方針の最大値を返します。

`sched_get_priority_min`
　指定された方針の最小値を返します。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`
　指定された `timespec` 構造体を現在の実行時間制限に更新します。詳細は、`sched_get_priority_max`(3RT) のマニュアルページを参照してください。

`sched_setparam`, `sched_getparam`
　指定されたプロセスのスケジューリングパラメータを設定または取得します。

`sched_yield`
　呼び出し側プロセスがプロセスリストの先頭に戻るまで、呼び出し側プロセスをブロックします。

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システム管理用ユーティリティには、`dispadmin`(1M) および `priocntl`(1) があります。どちらのユーティリティも `priocntl`(2) システムコールをサポートし、オプションに互換性があり、モジュールもロード可能です。これらのユーティリティは、実行時にリアルタイム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ータの設定および取得を行います。

`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、実行時に現在の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。リアルタイム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。

`dispadmin`(1M) のクラスオプションは次のとおりです。

`-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`　パラメータを表示または変更するクラスを指定する

`-g`　指定されたクラスのディスパッチパラメータを取得する

-r　–g オプションと共に使用し、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ータが保存されているクラス固有のファイルは実行時にもロードできます。このファイルを使用して、ブート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は、-g オプションで使用される書式で引数を表明する必要があります。RT クラスのパラメータについては rt_dptbl(4) を参照し、例については例 9–1 を参照してください。

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する必要があります。

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

次の手順で、RT クラスのディスパッチテーブルのインストールを行います。

1. 次のコマンドでクラス固有のモジュールをロードする。*module_name* にはクラス固有のモジュールを指定してください。

   # **modload /kernel/sched/***module_name*

2. dispadmin コマンドを起動する。

   # **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る必要があります。

スケジューリングの設定

RT と TS の両方のスケジューリングクラスには、パラメータテーブル rt_dptbl(4) および ts_dptbl(4) が関連づけられています。これらのテーブルは、ブート時にロード可能なモジュールを使用するか、実行時に dispadmin(1M) を使用することで設定されます。

ディスパッチャパラメータテーブル

リアルタイム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) 構造体はパラメータの配列 struct rt_dpent_t から構成されます。*n* 個の優先順位レベルはそれぞれ 1 つのパラメータを持っています。ある優先順位レベルの設定項目は、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。

パラメータ構造体は次のメンバーから構成されます (これらは、/usr/include/sys/rt.h ヘッダファイルでも説明されています)。

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。dispadmin(1M) では、rt_globpri の値は変更できない

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。詳細は、208 ページの「タイムスタンプインタフェース」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。リアルタイムプロセスのタイムカンタムを変更するには、priocntl(1) コマンドまたは priocntl(2) システムコールを使用する

config_rt_dptbl の再設定

リアルタイムのシステム管理者は、いつでも config_rt_dptbl を再設定して、スケジューラのリアルタイム部分の動作を変更できます。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」という節で説明されています。

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リアルタイムパラメタテーブルを調査または変更する方法です。dispadmin(1M) をリアルタイム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_dptbl 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とき、dispadmin(1M) への入力に使用される設定ファイルは rt_dptbl(4) のマニュアルページで説明されている書式に準拠する必要があります。

次に、config_rt_dptbl[] 内の優先順位を設定されたプロセス rtdpent_t とそれに関連づけられたタイムカンタム値の例を示します。

例 **9–1** RT クラスのディスパッチパラメータ

```
rtdpent_t  rt_dptbl[] = {             129,    60,
     /* 優先順位レベルのタイムカンタム */                                  130,    40,
        100,    100,                                                     131,    40,
        101,    100,                                                     132,    40,
        102,    100,                                                     133,    40,
        103,    100,                                                     134,    40,
        104,    100,                                                     135,    40,
        105,    100,                                                     136,    40,
        106,    100,                                                     137,    40,
        107,    100,                                                     138,    40
        108,    100,                                                     139,    40,
        109,    100,                                                     140,    20,
        110,    80,                                                      141,    20,
        111,    80,                                                      142,    20,
        112,    80,                                                      143,    20,
        113,    80,                                                      144,    20,
        114,    80,                                                      145,    20,
```

例 **9–1** RT クラスのディスパッチパラメータ　*(続き)*

```
        115,    80,                                     146,    20,
        116,    80,                                     147,    20,
        117,    80,                                     148,    20,
        118,    80,                                     149,    20,
        119,    80,                                     150,    10,
        120,    60,                                     151,    10,
        121,    60,                                     152,    10,
        122,    60,                                     153,    10,
        123,    60,                                     154,    10,
        124,    60,                                     155,    10,
        125,    60,                                     156,    10,
        126,    60,                                     157,    10,
        126,    60,                                     158,    10,
        127,    60,                                     159,    10,
        128,    60,                                 }
```

メモリーのロック

メモリーのロックは、リアルタイムアプリケーションにとって最重要事項の 1 つです。リアルタイム環境では、応答時間を減らし、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ー内に常駐することが保証されなければなりません。

この節では、SunOS において、リアルタイムアプリケーションが利用できるメモリーロックのメカニズムについて説明します。

SunOS では、プロセスがメモリーに常駐するかどうかは、プロセスの現在の状態、使用できる全物理メモリー、動作中のプロセス数、およびプロセッサのメモリーに対するデマンドによって決まります。このような方法は、タイムシェアリング環境には適合します。しかし、リアルタイム環境には受け入れられないことがよくあります。リアルタイム環境では、プロセスのメモリーアクセスとディスパッチ応答時間を減らすために、プロセスはメモリー常駐を保証する必要があります。

SunOS のリアルタイムメモリーロックは、ライブラリルーチンのセットで提供されます。このようなライブラリルーチンを使用すると、スーパーユーザー特権で実行しているプロセスは、自分の仮想アドレス空間の指定された部分を物理メモリーにロックできます。このようにしてロックされたページは、ロックが解除されるか、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。

オペレーティングシステムには、一度にロックできるページ数にシステム全体の制限があります。この制限は調整可能なパラメータであり、デフォルト値はブート時に計算されます。デフォルト値は、ページフレーム数から一定の割合 (現在は 10 %に設定) を引いた値が基本になります。

ページのロック

mlock(3C) への呼び出しは、メモリの 1 セグメントをシステムの物理メモリーにロックするように要求します。たとえば、指定されたセグメントを構成するページにページフォルトが発生したと仮定します。すると、各ページのロックカウントが 1 だけ増えます。ロックのカウントが 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るマッピングで複数のプロセスを使って何度もロックできます。2 つの異なったプロセスが同じページをロックすると、両方のプロセスがロックを解除するまで、そのページはロックされたままです。ただし、マッピング内でページロックは入れ子にしません。同じプロセスが同じアドレスに対してロックインタフェースを何度も呼び出した場合でも、ロックは一度のロック解除要求で削除されます。

ロックが実行されているマッピングが削除されると、そのメモリーセグメントのロックは解除されます。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、ロックは暗黙的に解除されます。

fork(2) 呼び出しが行われたあと、ロックは子プロセスには継承されません。したがって、メモリーをロックしているプロセスが子プロセスをフォークした場合、子プロセスは自分でメモリーロック操作を行い、自分のページをロックする必要があります。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。

ページのロック解除

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。munlock により、指定された仮想ページのロックカウントは 1 だけ減ります。ページのロックカウントが 0 まで減ると、そのページは普通にスワップされます。

全ページのロック

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、既存のメモリーマッピングがすべてロックされます。MCL_FUTURE フラグが設定されている場合は、既存のマッピングに追加されるマッピングまたは既存のマッピングを置き換えるマッピングはすべて、メモリー内にロックされます。

スティッキロック

ページのロックカウントが 65535 (0xFFFF) に達すると、そのページは永久にロックされます。値 0xFFFF は実装によって定義されています。この値は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クされたページは、ロックを解除できません。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、リアルタイムプロセスでの入出力について説明します。SunOS では、高速で非同期的な入出力操作を実行するために、インタフェースと呼び出しの 2 つのセットをライブラリで提供しています。POSIX 非同期入出力インタフェースは最新の標準です。SunOS 環境は、情報の消失やデータの不一致を防止するために、ファイルおよびメモリー内の同期操作とモードを提供します。

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。

リアルタイムアプリケーションは、入出力に非同期でバインド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。

非同期入出力は、任意の SunOS ファイルで使用できます。ファイルは同期的に開かれますが、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インタフェースを呼び出し、入出力要求は待ち行列に置かれ、呼び出しはすぐに戻ります。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 制御ブロックは、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。制御ブロックを使用できる要求は一度に1 つだけです。その要求が完了すると、制御ブロックはまた使用できるようになります。

一般的な POSIX 非同期入出力操作は、aio_read(3RT) または aio_write(3RT) 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すると、操作の完了を判断できます。操作の完了にシグナルを使用する場合は、各操作に一意のタグを付けることができます。この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。siginfo(3HEAD) のマニュアルページを参照してください。

aio_read	aio_read(3RT) 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クを使用して呼び出します。
aio_write	aio_write(3RT) 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クを指定して呼び出します。
aio_return, aio_error	aio_return(3RT) および aio_error(3RT) はそれぞれ、操作が完了していると判明したあとに、戻り値とエラー値を取得するために呼び出します。
aio_cancel	aio_cancel(3RT) 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クを指定して呼び出します。aio_cancel は、制御ブロックによって要求が指定されている場合、指定された要求を取り消します。aio_cancel はまた、制御ブロックによって指定されたファイル記述子に対して保留されているすべての要求を取り消します。
aio_fsync	aio_fsync(3RT) は、指定されたファイル上で保留されているすべての入出力操作に対する非同期的な fsync(3C) または fdatasync(3RT) 要求を待ち行列に入れます。
aio_suspend	aio_suspend(3RT) は、1 つ以上の先行する非同期入出力要求が同期的に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

この節では、Solaris オペレーティング環境における非同期入出力について説明します。

通知 (SIGIO)

非同期入出力呼び出しが正常に返ったとき、入出力操作は単に待ち行列に入って、実行が行われるのを待ちます。実際の操作は、戻り値と潜在的なエラー識別子を持っています。呼び出しが同期的に行われた場合、この戻り値と潜在的なエラー識別子は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両方とも、ユーザーが要求時に 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る errno */
 } 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しているプロセスは、SIGIO シグナルの原因を特定できません。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確認する必要があります。

aioread

aioread(3AIO) ルーチンは read(2) の非同期バージョンです。通常の読み取り引数に加えて、aioread(3AIO) はファイルの位置を指定する引数と aio_result_t 構造体のアドレスを指定する引数を取ります。aio_result_t 構造体には、操作の結果情報が格納されます。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。aioread(3AIO) 呼び出しが成功したか失敗したかに関係なく、ファイルポインタは更新されます。

aiowrite

aiowrite(3AIO) ルーチンは write(2) の非同期バージョンです。通常の書き込み引数に加えて、aiowrite(3AIO) はファイルの位置を指定する引数と aio_result_t 構造体のアドレスを指定する引数を取ります。aio_result_t 構造体には、操作の結果情報が格納されます。

ファイル位置は、この操作が行われる前に、ファイル内でシーク操作が実行されることを指定します。aiowrite(3AIO) 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あと、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

aiocancel(3AIO) ルーチンは、aio_result_t 構造体を引数として指定した非同期要求を取り消そうとします。aiocancel(3AIO) 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場合、aiocancel(3AIO) は失敗します。

aiowait

aiowait(3AIO) 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、呼び出し側プロセスはブロックされます。タイムアウトパラメータは、入出力の完了を待つ最大インターバルを指します。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(3AIO) 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

非同期入出力イベントの完了を同期的に決定するには、SIGIO 割り込みに依存するのではなく、poll(2) を使用します。ポーリングを使用すると、SIGIO 割り込みの原因を調べることもできます。

あまり多くのファイルで poll(2) を使用すると、処理が遅くなります。この問題は、poll(7D) で解決します。

poll

/dev/poll を使用すると、多数のファイル記述子のポーリングを高いスケーラビリティで行うことができます。このスケーラビリティは、新しい API のセットと新しいドライバ /dev/poll によって実現されます。/dev/poll API は poll(2) を置き換えるものではなく、どちらかを選択して使用するものです。poll(7D) を使用すると、/dev/poll API の詳細と例を提供できます。適切に使用すると、/dev/poll API は poll(2) よりも高いスケーラビリティを提供します。この API は、特に、次の条件を満たすアプリケーションに適します。

- 多数のファイル記述子のポーリングを繰り返し行うアプリケーション
- ポーリングが行われたファイル記述子が比較的安定している、つまりひんぱんに開閉が行われない
- ポーリングイベントの総数に比較して、保留が少ないファイル記述子のセット

close

ファイルを閉じるには、close(2) を呼び出します。close(2) を呼び出すと、未処理の非同期入出力要求のうち、閉じることができるものを取り消します。close(2) は、取り消せない操作の完了を待ちます。詳細は、202 ページの「aiocancel」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。ほか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。

同期モード

SunOS では、データが書き込み操作用としてファイルに正常に転送されるには、システムがファイルを開いたときには、以前に書き込まれたデータを読み取ることができることを保証している必要があります。この確認は、物理的な記憶媒体に障害がないことを想定しています。また、データが読み取り操作用として正常に転送されるには、要求側プロセスが物理記憶媒体上にあるデータのイメージを利用できる必要があります。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整合性を保証します。

- 読み取りの場合、操作は完了するか、失敗して原因究明されます。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とき、読み取るべきデータに保留状態の書き込み要求が影響を与える場合、この書き込み要求はデータを読み取る前に正常に終了します。
- 書き込みの場合も、操作は完了するか、失敗して原因究明されます。書き込みが正常に終了するのは、書き込み要求で指定されたデータが正しく転送され、さらに、そのデータを取得する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整合性の保証は、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が正常に転送されている必要があります。さもないと、同期入出力ファイルの整合性の保証は、同期入出力データの整合性の保証と同等です。

ファイルの同期

`fsync`(3C) および `fdatasync`(3RT) は明示的にファイルと二次記憶領域の同期をとります。

`fsync`(3C) ルーチンは、入出力ファイルの整合性の保証レベルでインタフェースの同期をとることを保証します。`fdatasync`(3RT) は、入出力データの整合性の保証レベルでインタフェース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) を使用して、ファイル記述子に 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は入出力データ完了に達します。ファイル記述子に 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は入出力ファイル完了に達します。ファイル記述子に 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(3RT)) はファイル記述子に設定したのと同じ完了レベルに達します。ファイル記述子に設定するのは、O_DSYNC または O_SYNC のどちらでもかまいません。

プロセス間通信

この節では、リアルタイム処理と関連する、SunOS のプロセス間通信 (IPC™) インタフェースについて説明します。また、シグナル、パイプ、FIFO、メッセージ待ち行列、共有メモリー、ファイルマッピング、およびセマフォについても説明します。プロセス間通信に役立つライブラリ、インタフェース、およびルーチンについては、第 5 章を参照してください。

シグナルの処理

次のように、送信側は sigqueue(3RT) 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。

以降に発生する保留状態のシグナルも待ち行列に入るので、ターゲットプロセスは指定されたシグナルの SA_SIGINFO ビットを設定する必要があります。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通常、シグナルを非同期的に受信します。シグナルを同期的に受信するには、シグナルをブロックして、sigwaitinfo(3RT) または sigtimedwait(3RT) を呼び出します。詳細は、sigprocmask(2) のマニュアルページを参照してください。この手順によって、シグナルは同期的に受信されるようになります。このとき、sigqueue(3RT) の呼び出し側が送信した値は siginfo_t 引数の si_value メンバーに格納されます。シグナルのブロックを解除しておくと、シグナルは siginfo_t 引数の si_value に格納された値とともに、sigaction(2) によって指定されたシグナルハンドラに配信されます。

あるプロセスが送信できるシグナルの数は関連する値で指定されており、残りは送信されないままになります。sigqueue(3RT) を最初に呼び出したとき、{SIGQUEUE_MAX } 個のシグナルが入る記憶領域が割り当てられます。次に、sigqueue(3RT) 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、制限時間内で失敗します。

パイプ、名前付きパイプ、およびメッセージ待ち行列

パイプ、名前付きパイプ、およびメッセージ待ち行列は、文字入出力デバイスと同様に動作します。ただし、これらのインタフェースは接続には異なる方法を使用します。パイプについての詳細は、80 ページの「プロセス間のパイプ」を参照してください。名前付きパイプについての詳細は、81 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についての詳細は、85 ページの「System V メッセージ」および 82 ページの「POSIX メッセージ」を参照してください。

セマフォ

セマフォも System V と POSIX の両方のスタイルで提供されます。詳細は、88 ページの「System V セマフォ」および 83 ページの「POSIX セマフォ」を参照してください。

この章で前述した手法によって優先順位の逆転を明示的に回避しない限り、セマフォを使用すると、優先順位の逆転が発生する場合があるので注意してください。

共有メモリー

プロセスが最も高速に通信する方法は、メモリーの共有セグメントを直接使用する方法です。3 つ以上のプロセスが同時に共有メモリーに読み書きしようとすると、メモリーの内容が正確でなくなる可能性があります。これは、共有メモリーを使用する場合の最も大きな問題です。

非同期ネットワーキング

この節では、ソケットまたはリアルタイムアプリケーション用のトランスポートレベルインタフェース (TLI) を使用した非同期ネットワーク通信について説明します。ソケットを使用した非同期ネットワーキングを行うには、`SOCK_STREAM` タイプのオープンソケットを非同期および非ブロックに設定します。非同期ソケットについての詳細は、125 ページの「ソケットの拡張機能」を参照してください。TLI イベントの非同期ネットワーク処理は、STREAMS 非同期機能と TLI ライブラリルーチンの非ブロックモードの組み合わせによってサポートされます。

TLI (Transport-Level Interface) についての詳細は、第 7 章を参照してください。

ネットワーキングのモード

ソケットと Transport-Level Interface (TLI) はどちらも 2 つのサービスモード、「コネクションモード」と「コネクションレスモード」を提供します。

「コネクションモード」サービスは回線指向です。このサービスを使用すると、データは、確立されたコネクション経由で、信頼できる順序付け方法で伝送されます。このサービスはまた、データ転送フェーズ中のアドレス解決および転送のオーバーヘッドを回避する識別処理を提供します。このサービスは、比較的長時間持続するデータストリーム指向の対話を必要とするアプリケーションに適しています。

「コネクションレスモード」サービスはメッセージ指向であり、複数のユニット間の論理的な関係が要求されない、独立したユニットでのデータ伝送をサポートします。単一のサービス要求は、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を送信側からトランスポートプロバイダに渡します。このサービス要求には、宛先アドレスや配信されるデータが含まれます。コネクションレスモードサービスは、対話が短時間で済み、保証された順番どおりの方法でデータを配信する必要がないアプリケーションに適しています。一般的に、コネクションレスモードによる伝送は信頼性が低いと言えます。

タイミング機能

この節では、SunOS におけるリアルタイムアプリケーションで利用できるタイミング機能について説明します。このようなメカニズムをリアルタイムアプリケーションで使用するには、この節で説明する各ルーチンのマニュアルページの詳細な情報が必要です。

SunOS のタイミングインタフェースは「タイムスタンプ」と「インターバルタイマー」の 2 つの機能に分類できます。「タイムスタンプインタフェース」は経過時間を測定する方法を提供します。したがって、タイムスタンプインタフェースを使用すると、アプリケーションはある状態の持続時間やイベント間の時間を測定できます。「インターバルタイマー」を使用すると、アプリケーションはさまざまな活動を指定された時刻に呼び起こし、時間の経過に基づいてスケジューリングできます。

タイムスタンプインタフェース

タイムスタンプは 2 つのインタフェースによって提供されます。gettimeofday(3C) 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* 構造体に提供します。clock_gettime は、CLOCK_REALTIME の clockid を使用して、gettimeofday(3C) が戻すタイムインターバルと同じ時間を秒とナノ秒で表し、現在の時間を *timeval* 構造体に提供します。

SunOS はハードウェア定期タイマーを使用します。このハードウェア定期タイマーが唯一の時間情報源であるワークステーションもあります。ハードウェア定期タイマーが唯一の時間情報源である場合、タイムスタンプの精度はハードウェア定期タイマーの解像度に制限されます。その他のプラットフォームでは、タイマーレジスタの解像度が 1 マイクロ秒である場合、そのタイムスタンプの精度は 1 マイクロ秒であることを意味します。

インターバルタイマーインタフェース

リアルタイムアプリケーションは、インターバルタイマーを使用して動作をスケジュールすることがよくあります。インターバルタイマーには「単発」型と「周期」型の 2 つの種類があります。

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

周期タイマーには、初期有効期限 (絶対時間または相対時間) と繰り返しインターバルが設定されています。インターバルタイマーの有効期限が経過するたびに、インターバルタイマーは繰り返して再ロードされます。そして、インターバルタイマーは自動的に再設定されます。このタイマーはデータの記録やサーボの制御に便利です。インターバルタイマー機能を呼び出すとき、システムのハードウェア定期タイマーの解像度より小さな時間値は、ハードウェア定期タイマーのインターバルの次の倍数に丸められます。このインターバルは通常 10 ミリ秒です。

SunOS には、2 種類のタイマーインタフェースがあります。setitimer(2) および getitimer(2) インタフェースは「BSD タイマー」という固定設定タイマーを動作させ、timeval 構造体を使用して、時間インターバルを指定します。timer_create (3RT) で作成される POSIX タイマーは POSIX クロック CLOCK_REALTIME を動作させます。POSIX タイマーの動作は timespec 構造体によって表されます。

getitimer(2) および setitimer(2) 関数はそれぞれ、指定された BSD インターバルタイマーの値を取得および確立します。プロセスは 3 つの BSD インターバルタイマーを利用できます (ITIMER_REAL で指定されるリアルタイムタイマーを含む)。BSD タイマーが設定されており、有効になっている (期限切れになることが許可されている) 場合、システムはタイマーを設定したプロセスに適切なシグナルを送信します。

timer_create(3RT) ルーチンは TIMER_MAX 個までの POSIX タイマーを作成できます。呼び出し側はタイマーの有効期限が経過したときに、どのシグナルとそれに関連する値をプロセスに送信するかを指定できます。timer_settime(3RT) および timer_gettime(3RT) ルーチンはそれぞれ、指定された POSIX インターバルタイマーの値を取得および確立します。必要なシグナルの配信が保留状態の間でも、POSIX タイマーは期限切れになることがあります。タイマーの有効期限がカウントされるので、timer_getoverrun(3RT) でそのカウントを取得します。timer_delete(3RT) で POSIX タイマーの割り当てを解除します。

次の例に、setitimer(2) を使用して、定期割り込みを生成する方法と、タイマー割り込みの到着を制御する方法を示します。

例 **9–2** タイマー割り込みの制御

```
#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 をブロックする */
    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_action = 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ーバルタイマーを設定する
     */
    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);
```

例 **9–2** タイマー割り込みの制御　(続き)

```
    /* 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);
    }
}
```

第 10 章

Solaris ABI と ABI ツール

Solaris Application Binary Interface (ABI) はアプリケーション開発者用のインタフェースを定義します。この ABI に準拠することによって、アプリケーションのバイナリの安定性が強化されます。この章では、Solaris ABI についてと、アプリケーションが Solaris ABI に準拠しているかどうかを確認するためのツールについて説明します。

- Solaris ABI の定義と目的については、212 ページの「Solaris ABI の定義」を参照してください。
- 2 つの ABI ツール `appcert` および `apptrace` の使用方法については、215 ページの「Solaris ABI ツール」を参照してください。

Solaris ABI とは ?

Solaris ABI とは、アプリケーションが Solaris オペレーティングシステムで利用できる (つまり、サポートされる) 実行時インタフェースセットのことです。ABI の最も重要な構成要素は次のとおりです。

- Solaris システムライブラリが提供するインタフェース (マニュアルページの 3 章を参照)
- Solaris カーネルシステムコールが提供するインタフェース (マニュアルページの 2 章を参照)
- さまざまなシステムファイルとシステムディレクトリの場所と形式 (マニュアルページの 4 章を参照)
- Solaris ユーティリティの入出力用の構文と意味論 (マニュアルページの 1 章を参照)

Solaris ABI の中心となる構成要素はシステムライブラリインタフェースセットです。この章では、「*ABI*」という用語はこのような構成要素だけを指します。Solaris オペレーティングシステムがインタフェースを提供するのは C 言語だけであるので、この ABI が持っているのも C 言語用のインタフェースだけです。

Solaris API (Application Programming Interface) 向けに作成された C ソースコードは C コンパイラによってバイナリに変換されます。このとき、バイナリはプラットフォームによって、3 つの ABI バージョン (32 ビット SPARC、64 ビットSPARC、または 32 ビットIntel) のうちの1 つになります。ABI は API とよく似ていますが、ソースをコンパイルするプロセスにいくつかの違いがあります。次に、違いについて説明します。

- コンパイラ指令 (`#define` など) はソースレベルの構成を変更または置換する可能性があります。結果として、ソースに存在していたシンボルがバイナリに存在しなかったり、ソースに存在していなかったシンボルがバイナリに存在することがあります。
- コンパイラはプロセッサ固有のシンボル (算術命令など) を生成することがあり、ソースレベルの構成を変更または置換する可能性があります。
- コンパイラのバイナリレイアウトは、そのコンパイラと、コンパイラが受け入れるソース言語のバージョンに固有になることがあります。このような場合、同じコードを異なるコンパイラでコンパイルすると、互換性のないバイナリが生成される可能性があります。

このような理由のため、異なる Solaris リリースでコンパイルした場合、ソースレベル (API) では互換性があっても、バイナリレベルでは十分な互換性を得られません。

Solaris ABI は、オペレーティングシステムが提供する、サポートされるインタフェースから構成されます。システムで利用できるインタフェースの中には、オペレーティングシステムが排他的に使用することを目的としているインタフェースもあります。このような排他的なインタフェースは、アプリケーションでは使用できません。SunOS 5.6 より前のリリースでは、アプリケーション開発者は Solaris ライブラリのすべてのインタフェースを利用できていました。Solaris リンクエディタのライブラリシンボル有効範囲の手法を使用すると、ライブラリの外では使用する予定がないインタフェースの有効範囲をライブラリのローカルだけに縮小できます。詳細については、『リンカーとライブラリ』 を参照してください。ただし、システム要件のため、必ずしもすべての非公開インタフェースがこのように有効範囲を縮小できるわけではありません。このようなインタフェースには「*private*」というラベルが付いてあり、Solaris ABI には含まれていません。

Solaris ABI の定義

Solaris ABI は Solaris のマニュアルページに説明されていますが、この ABI は Solaris ライブラリで定義されています。このような定義は、リンクエディタと実行時リンカーで使用されるライブラリバージョンの手法と方針によって行われます。

Solaris ライブラリにおけるシンボルバージョン管理

Solaris リンクエディタと実行時リンカーは、ファイルバージョン管理とシンボルバージョン管理の 2 種類のライブラリバージョン管理を使用します。ファイルバージョン管理では、ライブラリの名前にバージョン番号が追加されます (たとえば、`libc.so.1`)。このようなライブラリにある 1 つまたは複数の公開インタフェースに互換性のない変更を行ったとき、バージョン番号はインクリメントされます (たとえば、`libc.so.2`)。動的にリンクされるアプリケーションでは、構築時にバインドしたシンボルが実行時にライブラリに存在しないことがあります。シンボルのバージョンを管理するために、Solaris リンカーはシンボルのセットに名前を関連付けます。Solaris リンカーは次に、実行時のリンク中、その名前がライブラリに存在するかどうかを確認して、関連付けたシンボルが存在することを検証します。

ライブラリシンボルバージョン管理では、シンボルのセットにシンボルバージョン名を関連付け、その名前に番号付けスキーマがある場合は、マップファイルを使用して番号を関連付けます。次の例は、架空の Sun ライブラリ `libfoo.so.1` のマップファイルです。

```
SUNW_1.2 {
    global:
        symbolD;
        symbolE
} SUNW_1.1;

SUNW_1.1 {
    global:
        symbolA;
        symbolB;
        symbolC;
};

SUNWprivate {
    global:
        __fooimpl;
};

local: *
```

このマップファイルでは、`symbolA`、`symbolB`、および `symbolC` がバージョン `SUNW_1.1` に関連付けられ、`symbolD` および `symbolE` が `SUNW_1.2` に関連付けられています。そして、`SUNW_1.2` が `SUNW_1.1` に関連付けられたすべてのシンボルを継承することを意味します。シンボル `__fooimpl` は、継承チェーンを持たない異なる名前付きセット *SUNWprivate* に関連付けられています。

構築時、リンクエディタはアプリケーションが使用しているシンボルを調査します。リンクエディタは次に、これらのシンボルが依存しているアプリケーションにセット名を記録します。(番号付け継承) チェーンを持っているセットの場合、リンクエディタは、アプリケーションが使用するすべてのシンボルが含まれる最小限の名前付きセットを記録します。例えばアプリケーションが `symbolA` および `symbolB` だけを使

用する場合、リンクエディタは SUNW_1.1 への依存関係を記録します。また、アプリケーションが symbolA、symbolB、および symbolD を使用する場合、SUNW_1.2 は SUNW_1.1 を取り込んでいるためリンクエディタは SUNW_1.2 への依存関係を記録します。

実行時、リンカーは、依存関係としてアプリケーションに記録されたバージョン名がリンクされるライブラリに存在しているかどうかをチェックします。このプロセスによって、必要なシンボルが存在しているかどうかをすばやく確認できます。詳細については、『リンカーとライブラリ』を参照してください。

注 – マップファイル内の「*local: **」指令は、ライブラリ内のシンボルのうち、明示的に名前付きセットに関連付けられていないシンボルの有効範囲はライブラリにローカルであることを意味します。つまり、このように有効範囲がローカルに制限されたシンボルはライブラリの外からは見えないことを意味します。この規約は、シンボルが見えるのは、シンボルバージョン管理名に関連付けられているときだけであることを保証します。

シンボルバージョン管理による Solaris ABI へのラベル付け

ライブラリ内のシンボルのうち、表示できるシンボルはなんらかの名前付きセットに属しているので、名前付けスキーマを使用すると、シンボルの ABI 状態にラベルを付けることができます。このラベル付けを行うには、すべての非公開インタフェースを *SUNWprivate* から始まるセット名に関連付けます。公開インタフェースはほかの名前から始まり、特に、次のような名前があります。

- *SYSVABI* - System V ABI 定義で定義されたインタフェース用
- *SISCD* - SPARC International の『*SPARC Compliance Definition*』で定義されたインタフェース用
- *SUNW* - Sun Microsystems で定義されたインタフェース用

このような公開名前付きセットには *major.minor* 番号付けスキーマによって番号が付けられます。minor 番号は、新しいシンボルが名前付きセットに追加されるたびにインクリメントされます。major 番号は、あまりありませんが、既存のシンボルに互換性のない変更が行われたときにインクリメントされます。既存のシンボルに互換性のない変更が行われたときは、ライブラリのファイル名のバージョン番号もインクリメントされます。

したがって、Solaris ライブラリ ABI の定義はライブラリに含まれており、*SUNWprivate* から始まらないシンボルバージョン名に関連付けられたシンボルセットから構成されます。pvs コマンドは、ライブラリ内にあるシンボルの一覧を表示します。

Solaris ABI ツール

Solaris インタフェースを使用するアプリケーションが Solaris ABI に準拠しているかどうかを確認するために、Solaris オペレーティング環境は 2 つのツールを提供します。appcert ユーティリティは、ELF バイナリが使用している Solaris ライブラリインタフェースの非公開インタフェースの使用状況について静的に調査します。 次に、appcert ユーティリティは潜在的なバイナリ安定性の問題について、要約レポートと詳細レポートを生成します。apptrace ツールは実行時リンカーのリンク監査機能を使用して、アプリケーションを実行しながら動的に Solaris ライブラリルーチン呼び出しをトレースします。この機能によって、開発者は、アプリケーションが Solaris システムインタフェースを使用しているかどうかを調査できます。

ABI ツールを使用すると、特定の Solaris リリースで互換性の問題が存在するバイナリをすばやく簡単に識別できます。バイナリ安定性を確認するには、次のようにします。

- 現在の **Solaris** リリース上で **appcert** を使用して、問題のあるバイナリを選別します。バイナリが、問題のあるインタフェースを使用しているか否かを識別できます。
- ターゲットの **Solaris** リリース上で **apptrace** を使用して、問題が存在するかどうかを確認します。インタフェースを使用しながら動的に観察することによって、インタフェース互換性の問題が存在するかどうかを確認できます。

appcert ユーティリティ

appcert ユーティリティは、ELF バイナリを静的に調査して、使用されているライブラリシンボルを特定の Solaris リリースにおける公開インタフェースまたは非公開インタフェースのモデルに対して比較する Perl スクリプトです。appcert ユーティリティは SPARC またはIntel のどちらのプラットフォーム上でも動作します。appcert ユーティリティはさらに、SPARC と Intel の32 ビットインタフェースだけではなく、SPARC の 64 ビットインタフェースの使用状況も確認できます。appcert ユーティリティは C 言語インタフェースだけを調査することに注意してください。

新しい Solaris リリースが入手可能になると、いくつかのライブラリインタフェースは動作が変わったり、完全になくなったりします。すると、このようなインタフェースに依存するアプリケーションの性能に影響を与えることがあります。Solaris ABI は、アプリケーションが安全かつ安定して使用できる実行時ライブラリインタフェースを定義します。appcert ユーティリティは、アプリケーションが Solaris ABI に準拠していることを開発者が確認できるように設計されています。

appcert の確認項目

アプリケーションを調査するとき、appcert ユーティリティは次のことを確認します。

- 非公開シンボルの使用
- 静的なリンク
- 非結合シンボル

非公開シンボルの使用

非公開シンボルとは、Solaris ライブラリがお互いを呼び出すときに使用する関数またはデータのことです。非公開シンボルの意味論的な動作は変更されることがあり、ときには、削除されることもあります。このようなシンボルのことを「降格シンボル」と呼びます。非公開シンボルは変更されやすいので、非公開シンボルに依存しているアプリケーションには潜在的に安定性に問題があります。

静的なリンク

Solaris ライブラリ間で非公開シンボルを呼び出すとき、その意味論はリリースごとに変わる可能性があります。したがって、アーカイブに静的にリンクすると、アプリケーションのバイナリ安定性が低下します。この問題を回避するためには、このようなアーカイブに対応する共有オブジェクトファイルに動的にリンクすることが必要です。

非結合シンボル

アプリケーションを調査するとき、`appcert` ユーティリティは動的リンカーを使用してアプリケーションが使用するライブラリシンボルを解決します。このとき、動的リンカーが解決できないシンボルのことを「非結合シンボル」と呼びます。非結合シンボルの原因には、`LD_LIBRARY_PATH` 環境変数が正しく設定されていないなどの環境の問題があります。非結合シンボルの原因にはまた、コンパイル時に `-l` *lib* または `-z` のスイッチの定義を省略したなどの構築時の問題もあります。ただしこのような例はまれで、多くの場合 `appcert` ユーティリティが非結合シンボルを報告するときは、より深刻な問題が発生していることになります。

`appcert` の非確認項目

`appcert` ユーティリティで調査しているオブジェクトファイルがライブラリに依存する場合、このような依存関係をオブジェクトに記録しておく必要があります。このような依存関係をオブジェクトに記録するには、コードをコンパイルするときに、コンパイラの `-l` スイッチを使用します。オブジェクトファイルがほかの共有ライブラリに依存する場合、`appcert` ユーティリティを実行するときに、このような共有ライブラリには `LD_LIBRARY_PATH` または `RPATH` 経由でアクセスできる必要があります。

マシンが 64 ビットの Solaris カーネルを実行していなければ、`appcert` ユーティリティは 64 ビットアプリケーションを確認できません。64 ビットアプリケーションを確認しているとき、`appcert` ユーティリティは静的リンクを確認しません。

appcert は次のことを確認できません。

- 完全または部分的に静的にリンクされているオブジェクトファイル。完全に静的にリンクされているオブジェクトは「unstable (安定していない)」と報告される
- 実行権が設定されていない実行可能ファイル。appcert ユーティリティはこのような実行可能ファイルをスキップする。実行権が設定されていない共有オブジェクトは通常どおりに確認する
- ユーザー ID が root に設定されているオブジェクトファイル
- ELF 以外の実行可能ファイル (シェルスクリプトなど)
- C 言語以外の言語の Solaris インタフェース。コードは C 言語である必要はないが、Solaris ライブラリへの呼び出しは C 言語を使う必要がある

appcert の使用方法

appcert を使用して、利用しているアプリケーションを確認するには、次のように入力します。

appcert *object* | *directory*

object | *directory* は次のどちらかです。

- appcert で調査したいオブジェクトの完全な一覧
- このようなオブジェクトが格納されているディレクトリの完全な一覧

注 – appcert ユーティリティは、アプリケーションを実行する環境とは異なる環境で実行する場合もあります。このような環境では、appcert ユーティリティは Solaris ライブラリインタフェースへの参照を正しく解決できないことがあります。

---

appcert は Solaris 実行時リンカーを使用して、実行可能ファイルまたは共有オブジェクトファイルごとにインタフェース依存関係のプロファイルを構築します。このプロファイルを使用すると、アプリケーションが依存している Solaris システムインタフェースを判断できます。このプロファイルに記述されている依存関係を Solaris ABI と比較すると、Solaris ABI への準拠を確認できます。このプロファイルには、非公開インタフェースが見つかってはなりません。

appcert はディレクトリを再帰的に検索して、ELF 以外を無視しながら、オブジェクトファイルを探します。アプリケーションの確認が終了すると、appcert は長いレポートを標準出力 (通常は画面) に出力します。このレポートは、作業用ディレクトリ (通常は /tmp/appcert.*pid*) のReport という名前のファイルに書き込まれます。このサブディレクトリ名の *pid* は 1 から 6 桁の数字であり、appcert の当該インスタンスのプロセス ID を示します。appcert が出力ファイルを書き込むディレクトリ構造の詳細については、220 ページの「appcert の結果」を参照してください。

appcert のオプション

次のオプションで、appcert の動作を変更できます。次のオプションは、コマンド行において appcert コマンドから *object* | *directory* オペランドの間のどこにでも入力できます。

`-B`
: appcert ユーティリティをバッチモードで実行します。

  バッチモードでは、appcert は確認するバイナリごとに 1 行をレポートに書き込みます。

  PASS で始まる行は、その行が示すバイナリには appcert の警告が発行されなかったことを意味します。

  FAIL で始まる行は、その行が示すバイナリに問題が見つかったことを意味します。

  INC で始まる行は、その行が示すバイナリが完全には確認できなかったことを意味します。

`-f` *infile*
: ファイル *infile* には、確認すべきファイルの一覧がファイル名ごとに 1 行ずつ書き込まれている必要があります。このファイルで指定されたファイルは、コマンド行に指定されたファイルに追加されます。このスイッチを使用する場合、オブジェクトまたはディレクトリをコマンド行に指定する必要はありません。

`-h`
: appcert の使用方法を出力します。

`-L`
: デフォルトでは、appcert はアプリケーション内にあるすべての共有オブジェクトを記して、このような共有オブジェクトが入っているディレクトリを LD_LIBRARY_PATH に追加しますが、-L スイッチはこの動作を無効にします。

`-n`
: デフォルトでは、ディレクトリを検索して確認すべきバイナリを探すとき、appcert はシンボリックリンクに従います。-n スイッチはこの動作を無効にします。

`-S`
: Solaris ライブラリディレクトリ /usr/openwin/lib および /usr/dt/lib を LD_LIBRARY_PATH に追加します。

`-w` *working_dir*
: ライブラリ構成要素を実行するディレクトリを指定します。このスイッチを指定した場合、appcert は一時ファイルをこのディレクトリに作成します。このスイッチを指定しない場合、appcert は一時ファイルを /tmp ディレクトリに作成します。

appcert によるアプリケーションの選択

appcert を使用すると、指定されたセットの中でどのアプリケーションが潜在的に安定性の問題を持っているかをすばやく簡単に識別できます。appcert が何も安定性の問題を報告しなかった場合、そのアプリケーションは、以降の Solaris リリースでもバイナリ安定性問題は検証されにくいと考えられます。次の表に、よくあるバイナリ安定性問題の一覧を示します。

表 10–1 よくあるバイナリ安定性問題

問題	回避方法
ある非公開シンボルを使用しているが、変更されることがわかっている	このようなシンボルの使用をすぐに停止する
ある非公開シンボルを使用しているが、まだ変更されていない	今のところアプリケーションは動作しているが、このようなシンボルの使用をできるだけ早く停止する
あるライブラリに静的にリンクしているが、同等な共有オブジェクトを入手できる	代わりに、同等な共有オブジェクトを使用する
あるライブラリに静的に共有しているが、同等な共有オブジェクトを入手できない	可能であれば、ld -z allextract を使用して、.a ファイルを .so ファイルに変換する。変換できない場合、共有オブジェクトが利用できるようになるまで、静的なライブラリを使用し続ける
ある非公開シンボルを使用しているが、同等な公開シンボルを入手できない	Sun に連絡して、公開インタフェースを要求する
あるシンボルを使用しているが、評判が悪いか、削除されることが計画されている	今のところアプリケーションは動作しているが、このようなシンボルの使用をできるだけ早く停止する
ある公開シンボルを使用しているが、すでに変更されている	コンパイルし直す

リリースによっては、非公開インタフェースを使用することによる潜在的な安定性の問題が発生しないこともあります。なぜなら、リリースが変わっても、非公開インタフェースの動作が変更されるとは限らないためです。ターゲットリリースで非公開インタフェースの動作が変更されているかどうかを確認するには、apptrace を使用します。apptrace の使用方法については、222 ページの「apptrace によるアプリケーションの確認」を参照してください。

appcert の結果

appcert ユーティリティがアプリケーションのオブジェクトファイルを解析した結果は、appcert ユーティリティの作業用ディレクトリ (通常は /tmp) にあるいくつかのファイルに書き込まれます。作業用ディレクトリの下にあるメインサブディレクトリは appcert.*pid* です。このとき、*pid* は appcert の当該インスタンスのプロセス ID です。appcert ユーティリティの結果は、次のファイルに書き込まれます。

Index	確認されたバイナリ間のマッピングと、当該バイナリに固有な appcert の出力が格納されているサブディレクトリ名が書き込まれる
Report	appcert を実行したときに stdout に出力されたレポートのコピーが書き込まれる
Skipped	appcert が確認しようとしたが強制的にスキップされたバイナリの一覧と、各バイナリがスキップされた理由が書き込まれる。スキップされる理由には次のようなものが挙げられる ■ ファイルがバイナリオブジェクトではない ■ 当該ユーザーではファイルを読み取ることができない ■ ファイルにメタキャラクタが含まれている ■ ファイルの実行ビットが設定されていない
objects/*object_name*	objects サブディレクトリの下には、appcert が確認するオブジェクトごとのサブディレクトリが作成される。サブディレクトリごとに、次のようなファイルが格納される

check.demoted.symbols	Solaris 降格シンボルであると appcert が疑っているシンボルの一覧
check.dynamic.private	オブジェクトが直接バインドされている Solaris 非公開シンボルの一覧
check.dynamic.public	オブジェクトが直接バインドされている Solaris 公開シンボルの一覧
check.dynamic.unbound	ldd -r を実行したときに、動的リンカーによってバインドされなかったシンボルの一覧。ldd が戻す行には、「file not found」も含まれる

`summary.dynamic`	`appcert` が調査したオブジェクト内にある動的バインドの要約がプリンタ形式で書き込まれる (各 Solaris ライブラリから使用される公開シンボルと非公開シンボルのテーブルも含まれる)

`appcert` は終了するときに、次の 4 つのうちの 1 つを返します。

0 `appcert` はバイナリ安定性問題の潜在的な原因を見つけなかった。

1 `appcert` は正常に実行されなかった。

2 `appcert` が確認した一部のオブジェクトにバイナリ安定性問題が見つかった。

3 `appcert` が確認すべきバイナリオブジェクトが見つからなかった。

`appcert` が報告した問題の修正

- 非公開シンボルの使用 – 開発したときの Solaris リリースとは異なる Solaris リリース上で実行しようとすると、非公開シンボルに依存するアプリケーションは動作しない可能性があります。これは、非公開シンボルはSolaris リリース間で変更または削除される可能性があるためです。アプリケーション内で非公開シンボルが使用されていることを `appcert` が報告した場合は、非公開シンボルを使用しないようにアプリケーションを再作成してください。
- 降格シンボル – 降格シンボルとは、後の Solaris リリースにおいて削除された、あるいは、有効範囲がローカルに制限された Solaris ライブラリの関数またはデータ変数のことです。このようなシンボルを直接呼び出すアプリケーションは、ライブラリが当該シンボルをエクスポートしないリリース上では動作できません。
- 非結合シンボル – 非結合シンボルとは、アプリケーションが参照するライブラリシンボルのうち、`appcert` によって呼び出されたときに動的リンカーが解決できなかったライブラリシンボルのことです。非結合シンボルは必ずしも常にバイナリ安定性が低いことを示す指標ではありませんが、降格シンボルへの依存関係など、より深刻な問題が発生していることを示す場合もあります。
- 廃止ライブラリ – 廃止ライブラリとは、将来のリリースで Solaris オペレーティング環境から削除される可能性があるライブラリのことです。`appcert` ユーティリティは廃止ライブラリのすべての使用に対して警告を発します。廃止ライブラリに依存するアプリケーションは、将来のリリースでサポートされなくなり、機能しなくなる可能性があります。廃止ライブラリのインタフェースを使用しないでください。
- `sys_errlist` または `sys_nerr` の使用 – `sys_errlist` シンボルおよび `sys_nerr` シンボルを使用すると、バイナリ安定性が低下する可能性があります。これは、`sys_errlist` 配列の終わりを越えた参照が行われる可能性があるためです。代わりに `strerror` を使用してください。

- 強いシンボルと弱いシンボルの使用 – 将来の Solaris リリースで動作が変更される可能性があるので、弱いシンボルに関連付けられた強いシンボルは非公開シンボルとして予約されます。アプリケーションは弱いシンボルに直接参照する必要があります。強いシンボルの例としては、弱いシンボル `socket` に関連付けられた `_socket` があります。

apptrace によるアプリケーションの確認

`apptrace` は、アプリケーションを実行しながら動的に Solaris ライブラリルーチンへの呼び出しをトレースする C 言語のプログラムです。`apptrace` ユーティリティは SPARC またはIntel のどちらのプラットフォーム上でも動作します。`apptrace` ユーティリティは、SPARC と Intel の32 ビットインタフェースだけではなく、SPARC の 64 ビットインタフェースへのインタフェース呼び出しをトレースできます。ただし `appcert` と同様に、`apptrace` が調査するのは C 言語のインタフェースだけです。

アプリケーションの確認

`appcert` を使用してアプリケーションのバイナリ安定性低下の危険性を判断した後は、`apptrace` を使用して各ケースの危険度を評価します。`apptrace` を使用すると、アプリケーションが 各インタフェースを正しく使用しているかどうかを確認し、特定のリリースとのバイナリ互換性を判断できます。

`apptrace` を用いると、アプリケーションが公開インタフェースを正しく使用しているかどうかを確認できます。たとえば、システム管理ファイル `/etc/passwd` を開くとき、アプリケーションは `open()` を使用するのではなく、適切なプログラマティックインタフェースを使用する必要があります。このような Solaris ABI を正しく使用しているかどうかを検査 できる機能を使用すると、潜在的なインタフェースの問題をすばやく簡単に識別できます。

apptrace の実行

`apptrace` を実行するとき、トレースするアプリケーションは何も変更する必要がありません。`apptrace` を使用するには、まず `apptrace` を入力し、次に希望のオプションを入力、最後に対象となるアプリケーションを実行するコマンド行に入力します。`apptrace` は実行時リンカーのリンク監査機能を使用して、アプリケーションによる Solaris ライブラリインタフェースへの呼び出しを遮断します。次に、`apptrace` は呼び出しをトレースして、呼び出しの引数と戻り値についての名前と値を出力します。トレースは単一の行に出力することも、読みやすさのために複数の行に分けて出力することも可能です。公開インタフェースは人が読める形式で出力されます。非公開インタフェースは 16 進数で出力されます。

apptrace は、個々のインタフェースとライブラリの両方のレベルで、トレースする呼び出しを選択できます。たとえば、apptrace は libnsl から printf() への呼び出しをトレースできるし、特定のライブラリ内にあるすべての printf() 呼び出しをトレースすることもできます。apptrace はまた、ユーザーが指定した呼び出しを詳細にもトレースできます。apptrace の動作を規定する仕様は、truss(1) の使用方法と整合性がある構文によって規定されます。-f オプションを指定すると、apptrace はフォークされた子プロセスもトレースします。-o オプションを指定すると、apptrace は -o に指定されたファイルに結果を出力します。

apptrace がトレースするのはライブラリレベルの呼び出しだけであり、また、実行中のアプリケーションのプロセスにロードされるので、truss よりも性能が上がります。しかし、printf は例外ですが apptrace は可変引数リストを受け入れたり、スタックなどの呼び出し側の情報を調査したりする関数への呼び出しをトレースできません (たとえば、setcontext、 getcontext、setjmp、longjmp、および vfork)。

apptrace 出力の解釈

次は、apptrace で単純な 1 バイナリアプリケーション ls をトレースしたときの出力例です。

例 **10–1** デフォルトのトレース動作

```
% apptrace ls /etc/passwd
ls       -> libc.so.1:atexit(func = 0xff3cb8f0) = 0x0
ls       -> libc.so.1:atexit(func = 0x129a4) = 0x0
ls       -> libc.so.1:getuid() = 0x32c3
ls       -> libc.so.1:time(tloc = 0x23918) = 0x3b2fe4ef
ls       -> libc.so.1:isatty(fildes = 0x1) = 0x1
ls       -> libc.so.1:ioctl(0x1, 0x540d, 0xffbff7ac)
ls       -> libc.so.1:ioctl(0x1, 0x5468, 0x23908)
ls       -> libc.so.1:setlocale(category = 0x6, locale = "") = "C"
ls       -> libc.so.1:calloc(nelem = 0x1, elsize = 0x40) = 0x23cd0
ls       -> libc.so.1:lstat64(path = "/etc/passwd", buf = 0xffbff6b0) = 0x0
ls       -> libc.so.1:acl(pathp = "/etc/passwd", cmd = 0x3, nentries = 0x0,
             aclbufp = 0x0) = 0x4
ls       -> libc.so.1:qsort(base = 0x23cd0, nel = 0x1, width = 0x40,
             compar = 0x12038)
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:strlen(s = "") = 0x0
ls       -> libc.so.1:strlen(s = "/etc/passwd") = 0xb
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:strlen(s = "") = 0x0
ls       -> libc.so.1:printf(format = 0x12ab8, ...) = 11
ls       -> libc.so.1:printf(/etc/passwd
format = 0x12abc, ...) = 1
ls       -> libc.so.1:exit(status = 0)
```

上記例は、ls /etc/passwd というコマンド上で、すべてのライブラリ呼び出しをトレースしたときのデフォルトのトレース動作を示しています。apptrace はシステムコールごとに、次のような情報を含む 1 行を出力します。

- システムコールの名前
- システムコールが属するライブラリ
- システムコールの引数と戻り値 ls の出力は apptrace の出力に混合されます。

例 **10–2** 選択的なトレース

```
% apptrace -t \*printf ls /etc/passwd
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:printf(format = 0x12ab8, ...) = 11
ls       -> libc.so.1:printf(/etc/passwd
format = 0x12abc, ...) = 1
```

上記例は、正規表現を使用することによって、apptrace がトレースする呼び出しを選択する方法を示しています。この例では、前の例と同じ ls コマンド上で、printf で終わるインタフェース (sprintf も含まれる) への呼び出しをトレースします。 この結果、apptrace は printf および sprintf への呼び出しだけをトレースします。

例 **10–3** 詳細なトレース

```
% apptrace -v sprintf ls /etc/passwd
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
  buf =    (char *) 0x233d0 ""
  format = (char *) 0x12af8 "%s%s%s"
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
  buf =    (char *) 0x233d0 ""
  format = (char *) 0x12af8 "%s%s%s"
/etc/passwd
```

上記例は、詳細トレースモードを示しており、読みやすさのために、sprintf への引数が複数の行に出力されています。最後に、apptrace は ls コマンドの出力を表示します。

付録 A

UNIX ドメインソケット

UNIX ドメインのソケットは、UNIX パスで名前付けされます。たとえば、ソケット名には `/tmp/foo` などがあります。UNIX ドメインソケットは、単一ホスト上のプロセス間でだけ交信します。UNIX ドメイン上のソケットは、単一ホスト上のプロセス間の交信にしか使用できないため、ネットワークプロトコルの一部とは見なされません。

ソケットタイプには、ユーザーが認識できる通信プロパティを定義します。インターネットドメインソケットを使用すると、TCP/IP トランスポートプロトコルにアクセスできます。インターネットドメインは、`AF_INET` という値で識別します。ソケットは、同じドメイン内にあるソケットとだけデータをやりとりします。

ソケットの作成

`socket`(3SOCKET) 呼び出しは、指定されたファミリに指定されたタイプのソケットを作成します。

```
s = socket(family, type, protocol);
```

プロトコルが指定されないと (値が `0`)、システムは要求されたソケットタイプをサポートするプロトコルを選択します。ソケットハンドル (ファイル記述子) が返されます。

ファミリは、`sys/socket.h` に定義されている定数の 1 つで指定します。`AF_` *suite* という定数は、名前を解釈するときに使用するアドレス形式を指定します。

次のコードでは、マシン内部で使用されるデータグラムソケットを作成します。

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

通常 *protocol* 引数には `0` (デフォルトのプロトコル) を設定します。

ローカル名のバインド

ソケットは、その作成時には名前がありません。アドレスがソケットにバインドされるまで、リモートプロセスはソケットを参照できません。通信プロセスは、アドレスを介して接続されます。UNIX ファミリでは、接続は、通常 1 つまたは 2 つのパス名からなります。UNIX ファミリのソケットは、必ずしも名前にバインドされる必要はありません。バインドされると、 `local pathname` や `foreign pathname` などの順序セットは重複して存在できません。パス名では、既存のファイルを参照できません。

`bind`(3SOCKET) 呼び出しを使用すると、プロセスはソケットのローカルアドレスを指定できます。これによって、`local pathname` 順序セットが作成され、一方、`connect`(3SOCKET) および `accept`(3SOCKET) はアドレスのリモート側を固定することによってソケットの関連付けを完了します。`bind`(3SOCKET) は次のように使用します。

```
bind (s, name, namelen);
```

s は、ソケットハンドルです。バインド名は、バイト文字列で、サポートするプロトコル (複数も可) がこれを解釈します。UNIX ファミリ名には、パス名とファミリが含まれます。例では、UNIX ファミリソケットに `/tmp/foo` という名前をバインドしています。

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind (s, (struct sockaddr *) &addr,
        strlen(addr.sun_path) + sizeof (addr.sun_family));
```

この例では、`AF_UNIX` ソケットアドレスの大きさを判断するときには NULL バイトがカウントされないので、`strlen`(3C) を使用しています。

`addr.sun_path` で参照されるファイル名は、システムファイルの名前空間でソケットとして作成されます。呼び出し側は、`addr.sun_path` が作成されるディレクトリに書き込み許可を持っている必要があります。このファイルは、不要になったときに呼び出し側が削除してください。`AF_UNIX` ソケットを削除するには、`unlink`(1M) を使用します。

コネクションの確立

通常、コネクションの確立は非対称に行われます。1 つのプロセスは、クライアントとして動作し、もう一方のプロセスはサーバーとして動作します。サーバーは、サービスに関連付けられた既知のアドレスにソケットをバインドし、コネクション要求のためにソケットをブロックします。これで、無関係のプロセスがサーバーに接続できます。クライアントは、サーバーのソケットへのコネクションを起動することでサーバーにサービスを要求します。クライアント側では、connect(3SOCKET) 呼び出しでコネクションを起動します。UNIX ファミリでは、これを次のように表現します。

```
struct sockaddr_un server;
        server.sun.family = AF_UNIX;
         ...
        connect(s, (struct sockaddr *)&server, strlen(server.sun_path)
         + sizeof (server.sun_family));
```

コネクションエラーについては、103 ページの「コネクションエラー」を参照してください。104 ページの「データ転送」では、データの転送方法を、105 ページの「ソケットを閉じる」では、ソケットを閉じる方法が説明されています。

索引

L

M

N

O

P

R

S

T

U

V

X

あ

い

う

お

か

き

は

ひ

ふ

ほ