プログラミングインタフェースガイド

Part No: E38879
2013 年 1 月

ORACLE

130221@25097

目次

はじめに

『プログラミングインタフェースガイド』では、アプリケーション開発者が使用する SunOS 5.10 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 5.10 は Solaris 10 オペレーティングシステム (Solaris OS) のコアであり、System V Interface Description (SVID) および Single UNIX Specification, version 3 (SUSv3) に準拠しています。SunOS 5.10 は UNIX System V, Release 4 (SVR4) と完全な互換性があり、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。

注 – この Solaris リリースでは、SPARC および x86 系列のプロセッサアーキテクチャー (UltraSPARC、SPARC64、AMD64、Pentium、Xeon、および Intel 64) を使用するシステムをサポートします。サポートされるシステムについては、Solaris OS: Hardware Compatibility List (`http://www.sun.com/bigadmin/hcl/`) を参照してください。本書では、プラットフォームにより実装が異なる場合は、それを特記します。

本書の x86 に関連する用語については、以下を参照してください。

- 「x86」は、64 ビットおよび 32 ビットの x86 互換製品系列を指します。
- 「x64」は、AMD64 または EM64T システムに関する 64 ビット特有の情報を指します。
- 「32 ビット x86」は、x86 をベースとするシステムに関する 32 ビット特有の情報を指します。

サポートされるシステムについては、Solaris OS: Hardware Compatibility List を参照してください。

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

次に示す章で、Solaris OS 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のサービスおよび機能について説明します。

第 1 章「メモリーと CPU の管理」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「リモート共有メモリー API (Solaris クラスタ用)」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「セッション記述プロトコル API」では、セッション記述プロトコル (SDP) の Solaris 実装用の 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 4 章「プロセススケジューラ」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 5 章「近傍性グループ API」では、近傍性グループの動作と構造、およびこれらのグループ内のスレッドに対するリソース優先順位を制御する、インタフェースについて説明します。

第 6 章「入出力インタフェース」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 7 章「プロセス間通信」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 8 章「ソケットインタフェース」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 9 章「XTI と TLI を使用したプログラミング」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 10 章「パケットフィルタリングフック」では、セキュリティ (パケットフィルタリングやファイアウォール) ソリューション、ネットワークアドレス変換 (Network Address Translation、NAT) ソリューションなど、カーネルレベルでネットワークソリューションを開発するインタフェースについて説明します。

第 11 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」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ます。

第 12 章「リアルタイムプログラミングと管理」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 13 章「Solaris ABI と ABI ツール」では、Solaris Application Binary Interface (ABI) と、アプリケーションが Solaris ABI に準拠していることを確認するためのツールである `appcert` および `apptrace` について説明します。

付録 A「UNIX ドメインソケット」では、UNIX ドメインソケットについて説明しています。

マニュアル、サポート、およびトレーニング

Sun の Web サイトでは、次の追加のリソースに関する情報を提供しています。

- ドキュメント (`http://www.sun.com/documentation/`)
- サポート (`http://www.sun.com/support/`)
- トレーニング (`http://www.sun.com/training/`)

Sun へのご意見

Sun はドキュメントの品質向上のために、お客様のご意見やご提案をお待ちしています。ご意見を投稿するには、`http://www.oracle.com/technetwork/indexes/documentation/index.html` で「Feedback」をクリックしてください。

表記上の規則

The following table describes the typographic conventions that are used in this book.

表 P–1 表記上の規則

字体	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。 `machine_name% you have mail.`
`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。	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: 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。

表 P–1 表記上の規則 (続き)

字体	意味	例
AaBbCc123	書名、新しい単語、および強調する単語を示します。	『ユーザーズガイド』の第 6 章を参照してください。 キャッシュは、ローカルに格納されるコピーです。 ファイルを保存しないでください。 注: いくつかの強調された項目は、オンラインでは太字で表示されます。

コマンド例のシェルプロンプト

次の表に、C シェル、Bourne シェル、および Korn シェルのデフォルトの UNIX システムプロンプト、およびスーパーユーザープロンプトを示します。

表 P–2 シェルプロンプト

シェル	プロンプト
C シェル	`machine_name%`
C シェルのスーパーユーザー	`machine_name#`
Bourne シェルおよび Korn シェル	`$`
Bourne シェルおよび Korn シェル(superuser)	`#`

◆ ◆ ◆ 第 1 章

1 メモリーと CPU の管理

この章では、アプリケーション開発者から見た Solaris オペレーティングシステムの仮想メモリーと CPU の管理について説明します。

- 15 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。
- 18 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。
- 20 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。
- 21 ページの「CPU パフォーマンスカウンタ」では、CPU 性能カウンタ (CPC) の使用方法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。このセクションでは、このようなインタフェースの要約について説明します。このセクションではまた、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。そのあと、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、マッピングされた領域に最初に書き込みが行われた時に、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あと、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ーデバイスなどです。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。キャッシュのステータス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 の使用法

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定します。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ステータス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ステータス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 の使用法

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、15 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 および munlockall の使用法

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 の使用法

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

もっと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)
- watchmalloc(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い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、watchmalloc、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削除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 変数の値で決定されます。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間だけが報告されますが、この動作は、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すると、プログラムの終了時に自動的にリークが報告されます。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、*n* の最大値は 16 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) の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` によって制御されます。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。

次のセクションでは、`-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

このセクションでは、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。

sysconf の使用法

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 の使用法

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 の使用法

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取得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。

CPU パフォーマンスカウンタ

このセクションでは、CPU 性能カウンタ (CPC) を使用するための開発者インタフェースについて説明します。Solaris アプリケーションは、配下のカウンタアーキテクチャーに依存しない CPC を使用できます。

libcpc に追加された API

このセクションでは、libcpc(3LIB) ライブラリに最近追加された API について説明します。以前のインタフェースについては、マニュアルページの libcpc を参照してください。

初期化インタフェース

アプリケーションが CPC 機能を使用できるように準備するには、cpc_open() 関数を呼び出してライブラリを初期化します。この関数は、ほかのインタフェースで使用される cpc_t * パラメータを返します。cpc_open() 関数の構文は、次のとおりです。

```
cpc_t*cpc_open(intver);
```

ver パラメータの値は、アプリケーションが使用中のインタフェースのバージョンを識別します。配下のカウンタがアクセス不可または使用不可の場合、cpc_open() 関数は失敗します。

ハードウェア照会インタフェース

```
uint_t cpc_npic(cpc_t *cpc);
uint_t cpc_caps(cpc_t *cpc);
void cpc_walk_events_all(cpc_t *cpc, void *arg,
          void (*action)(void *arg, const char *event));
void cpc_walk_events_pic(cpc_t *cpc, uint_t picno, void *arg,
          void(*action)(void *arg, uint_t picno, const char *event));
void cpc_walk-attrs(cpc_t *cpc, void *arg,
          void (*action)(void *arg, const char *attr));
```

cpc_npic() 関数は、配下のプロセッサ上の物理カウンタ数を返します。

cpc_caps() 関数は、*uint_t* パラメータを返します。そのパラメータ値は、配下のプロセッサがサポートする機能に対して実行されたビット単位の論理和操作の結果です。この関数には 2 つの機能があります。CPC_CAP_OVERFLOW_INTERRUPT 機能により、カウンタのオーバーフロー時に、プロセッサは割り込みを発生できます。CPC_CAP_OVERFLOW_PRECISE 機能により、プロセッサはどのカウンタがオーバーフローの割り込みを発生したかを判別できます。

カーネルは、配下のプロセッサがサポートするイベントのリストを管理します。単一チップ上の異なる物理カウンタが同じイベントのリストを使用する必要はありません。cpc_walk_events_all() 関数は、物理カウンタに関係なく、プロセッサがサポートするイベントごとにaction() ルーチンを呼び出します。cpc_walk_events_pic() 関数は、特定の物理カウンタで、プロセッサがサポートするイベントごとにaction() ルーチンを呼び出します。どちらの関数も、 *arg* パラメータを非解釈で呼び出し元から各 action() 関数の呼び出しに渡します。

プラットフォームは、配下のプロセッサがサポートする属性のリストを管理します。これらの属性によって、性能カウンタのプロセッサ固有の拡張機能にアクセスできます。cpc_walk_attrs() 関数は、属性名ごとにアクションルーチンを呼び出します。

構成インタフェース

```
cpc_set_t *cpc_set_create(cpc_t *cpc);
int cpc_set_destroy(cpc_t *cpc, cpc_set_t *set);
int cpc_set_add_request(cpc_t *cpc, cpc_set_t *set, const char *event,
          uint64_t preset, uint_t flags, uint_t nattrs,
          const cpc_attr_t *attrs);
int cpc_set_request_preset(cpc_t *cpc, cpc_set_t *set, int index,
          uint64_t preset);
```

不透明なデータ型 cpc_set_t は要求のコレクションを表します。これらのコレクションはセットと呼ばれます。cpc_set_create() 関数は空のセットを作成します。cpc_set_destroy() 関数はセットを削除して、セットが使用していたメモリーをすべて解放します。セットを削除すると、そのセットが使用していたハードウェアリソースが解放されます。

cpc_set_add_request() 関数は要求をセットに追加します。要求のパラメータは次のとおりです。

event　カウントのイベント名を指定する文字列。

preset　カウンタの初期値に使用される 64 ビットの符号なし整数。

flags　要求フラグのグループに適用される論理和操作の結果。

nattrs　attrs が指す配列内の属性の数。

attrs　cpc_attr_t 構造体の配列へのポインタ。

有効な要求フラグは次のとおりです。

CPC_COUNT_USER　このフラグを使用すると、CPU がユーザーモードで実行している間に発生するイベントをカウントできます。

CPC_COUNT_SYSTEM　このフラグを使用すると、CPU が特権モードで実行している間に発生するイベントをカウントできます。

CPC_OVF_NOTIFY_EMT　このフラグは、ハードウェアのカウンタオーバーフローの通知を要求します。

CPC インタフェースは、cpc_attr_t 構造体の配列として属性を渡します。

cpc_set_add_request() 関数が正常終了して戻った場合、この関数はインデックスを返します。インデックスは、cpc_set_add_request() 関数の呼び出しにより追加された要求が生成したデータを参照します。

cpc_set_request_preset() 関数は、事前に設定された要求の値を変更します。これによって、オーバーフローしたセットを新しい事前設定で再構築できます。

cpc_walk_requests() 関数は、ユーザーが提供した action() ルーチンを cpc_set_t 内の要求ごとに呼び出します。*arg* パラメータの値は、非解釈でユーザーのルーチンに渡されます。cpc_walk_requests() 関数を使用すると、アプリケーションはセット内の要求ごとに構成を出力できます。cpc_walk_requests() 関数の構文は、次のとおりです。

```
void cpc_walk_requests(cpc_t *cpc, cpc_set_t *set, void *arg,
void (*action)(void *arg, int index, const char *event,
uint64_t preset, uint_t flags, int nattrs,
            const cpc_attr_t *attrs));
```

バインド

このセクションのインタフェースは、セット内の要求を物理ハードウェアにバインドして、カウンタを開始位置に設定します。

```
int cpc_bind_curlwp(cpc_t *cpc, cpc_set_t *set, uint_t flags);
int cpc_bind_pctx(cpc_t *cpc, pctx_t *pctx, id_t id, cpc_set_t *set,
          uint_t flags);
int cpc_bind_cpu(cpc_t *cpc, processorid_t id, cpc_set_t *set,
          uint_t flags);
int cpc_unbind(cpc_t *cpc, cpc_set_t *set);
```

cpc_bind_curlwp() 関数は、セットを呼び出し元の LWP にバインドします。セットのカウンタはこの LWP に仮想化され、呼び出し元の LWP の実行中に CPU で発生したイベントをカウントします。cpc_bind_curlwp() ルーチンで有効なフラグは CPC_BIND_LWP_INHERIT だけです。

cpc_bind_pctx() 関数は、セットを libpctx(3LIB) を使って得られたプロセス内の LWP にバインドします。この関数に有効なフラグはありません。

cpc_bind_cpu() 関数は、セットを *id* パラメータで指定されたプロセッサにバインドします。セットを CPU にバインドすると、システム上にある既存の性能カウンタのコンテキストが無効になります。この関数に有効なフラグはありません。

cpc_unbind() 関数は性能カウンタを停止して、バインドされたセットに関連付けられたハードウェアを解放します。セットが CPU にバインドされている場合、cpc_unbind() 関数は、CPU から LWP をバインド解除して、CPC 仮想デバイスを解放します。

抽出

このセクションで説明するインタフェースを使用すると、カウンタからアプリケーションにデータを返すことができます。カウンタデータは、cpc_buf_t という不

透明なデータ構造体内にあります。このデータ構造体は、バインドされたセットが使用しているカウンタの状態についてのスナップショットを取得し、次の情報を含みます。

- 各カウンタの 64 ビットの値
- 最新のハードウェアスナップショットの時間表示
- バインドされたセットでプロセッサが使用した CPU サイクルの数をカウントする累積 CPU サイクルカウンタ

```
cpc_buf_t *cpc_buf_create(cpc_t *cpc, cpc_set_t *set);
int cpc_buf_destroy(cpc_t *cpc, cpc_buf_t *buf);
int cpc_set_sample(cpc_t *cpc, cpc_set_t *set, cpc_buf_t *buf);
```

`cpc_buf_create()` 関数は、 `cpc_set_t` で指定されたセットからデータを格納するバッファーを作成します。`cpc_buf_destroy()` 関数は、指定した `cpc_buf_t` に関連付けられたメモリーを解放します。`cpc_buf_sample()` 関数は、指定されたセットの代わりにカウントしているカウンタのスナップショットを取得します。`cpc_buf_sample()` 関数を呼び出す前に、指定されたセットはあらかじめバインドされ、バッファーが作成されている必要があります。

バッファーへの抽出では、そのセットに関連付けられた要求の事前設定を更新しません。`cpc_buf_sample()` 関数を使ってバッファーが抽出され、次にバインド解除してから再度バインドすると、`cpc_set_add_request()` 関数の元の呼び出し内の要求の事前設定からカウンタが開始します。

バッファー操作

次のルーチンにより、`cpc_buf_t` 構造体内のデータにアクセスできます。

```
int cpc_buf_get(cpc_t *cpc, cpc_buf_t *buf, int index, uint64_t *val);
int cpc_buf_set(cpc_t *cpc, cpc_buf_t *buf, int index, uint64_t *val);
hrtime_t cpc_buf_hrtime(cpc_t *cpc, cpc_buf_t *buf);
uint64_t cpc_buf_tick(cpc_t *cpc, cpc_buf_t *buf);
int cpc_buf_sub(cpc_t *cpc, cpc_buf_t *result, cpc_buf_t *left
      cpc_buf_t *right);
int cpc_buf_add(cpc_t *cpc, cpc_buf_t *result, cpc_buf_t *left,
      cpc_buf_t *right);
int cpc_buf_copy(cpc_t *cpc, cpc_buf_t *dest, cpc_buf_t *src);
void cpc_buf_zero(cpc_t *cpc, cpc_buf_t *buf);
```

`cpc_buf_get()` 関数は、*index* パラメータで特定したカウンタの値を取得します。*index* パラメータは、セットがバインドされる前に `cpc_set_add_request()` 関数で返された値です。`cpc_buf_get()` 関数は、*val* パラメータが示す位置のカウンタを格納します。

`cpc_buf_set()` 関数は、*index* パラメータで特定したカウンタの値を設定します。*index* パラメータは、セットがバインドされる前に `cpc_set_add_request()` 関数で返された値です。`cpc_buf_set()` 関数は、*val* パラメータが示す位置の値にカウンタ

の値を設定します。cpc_buf_get() 関数または cpc_buf_set() 関数のどちらも、対応する CPC 要求の事前設定を変更しません。

cpc_buf_hrtime() 関数は、いつハードウェアが抽出されたかを示す高精度な時間表示を返します。cpc_buf_tick() 関数は、LWP の実行中に経過した CPU クロックサイクルの数を返します。

cpc_buf_sub() 関数は、*left* と *right* パラメータで指定されたカウンタとクロック刻み値の違いを計算します。cpc_buf_sub() 関数は、*result* に結果を格納します。cpc_buf_sub() 関数の呼び出しでは、cpc_buf_t 値がすべて同じ cpc_set_t 構造体から由来する必要があります。*result* インデックスは、バッファー内の要求インデックスごとの *left* - *right* の計算結果を含みます。また、結果のインデックスは *tick* の違いも含みます。cpc_buf_sub() 関数は、出力先バッファーについての高精度な時間表示を、*left* または *right* バッファーの最新時間に設定します。

cpc_buf_add() 関数は、*left* と *right* パラメータで指定されたカウンタとクロック刻み値の合計を計算します。cpc_buf_add() 関数は、*result* に結果を格納します。cpc_buf_add() 関数の呼び出しでは、cpc_buf_t 値がすべて同じ cpc_set_t 構造体から由来する必要があります。*result* インデックスは、バッファー内の要求インデックスごとの *left* + *right* の計算結果を含みます。また、結果のインデックスは *tick* の合計も含みます。cpc_buf_add() 関数は、の出力先バッファーについての高精度な時間表示を、*left* または *right* バッファーの最新時間に設定します。

cpc_buf_copy() 関数は、*src* と同じ *dest* を作成します。

cpc_buf_zero() 関数は、*buf* 内のすべてを 0 に設定します。

起動インタフェース

このセクションでは、CPC の起動インタフェースについて説明します。

```
int cpc_enable(cpc_t *cpc);
int cpc_disable(cpc_t *cpc);
```

この 2 つのインタフェースはそれぞれ、既存の LWP にバインドされたセットのカウンタを有効および無効にします。これらのインタフェースを使用すると、libpctx を使ってカウンタ構成を制御プロセスに任せながら、アプリケーションは対象のコードを指定できます。

エラー処理インタフェース

このセクションでは、CPC のエラー処理インタフェースについて説明します。

```
typedef void (cpc_errhndlr_t)(const char *fn, int subcode, const char *fmt,
          va_list ap);
void cpc_seterrhndlr(cpc_t *cpc, cpc_errhndlr_t *errhndlr);
```

この 2 つのインタフェースによって、cpc_t ハンドルを渡すことができます。cpc_errhndlr_t ハンドルには、文字列のほかに整数のサブコードも指定します。整数の *subcode* は、*fn* 引数が参照する関数によって発生した特定のエラーを示します。整数の *subcode* により、アプリケーションはエラー状況を簡単に認識できます。*fmt* 引数の文字列は、エラーサブコードについての国際化された説明を含み、出力に適しています。

◆ ◆ ◆ 第 2 章

2 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コンポーネントで構成できます。このようなコンポーネントは、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。

- 30 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。
- 31 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。

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発行されます。発行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、発行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あと、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あと、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メカニズムを提供します。このメカニズムのことを「*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ると、エクスポートプロセスはデータを処理します。

API フレームワーク

RSM アプリケーションサポートコンポーネントは次のソフトウェアパッケージで配信されます。

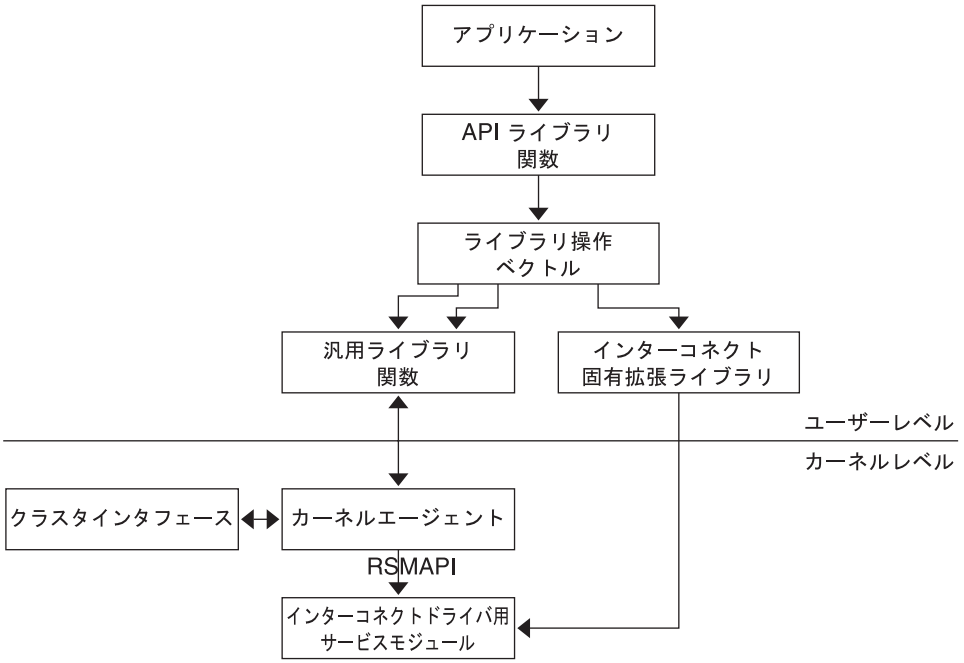
- `SUNWrsm`
  - RSMAPI 関数をエクスポートする共有ライブラリ (`/usr/lib/librsm.so`)。
  - ユーザーライブラリの代わりに RSMAPI インタフェースを介してメモリー相互接続ドライバとのインタフェースを行う Kernel Agent (KA) 仮想ドライバ (`/usr/kernel/drv/rsm`)。
  - 相互接続トポロジを取得するためのクラスタインタフェースモジュール。
- `SUNWrsmop`

  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リソースが解放されます。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コンポーネントは、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。アプリケーションコン

ポーネントは、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コンポーネントは、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コンポーネント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`	メモリーが不足しています
`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発行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あと、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、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あと、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あと、RSMAPI ライブラリインタフェースはセグメントを発行して、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発行および発行解除
- 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あと、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 プロパティー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	アクセス権が拒否されました
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_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。

セグメントの発行

```
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;    /* remote node id allowed to access resource */
    rsm_permission_t    ae_permissions;    /* mode of access allowed */
} 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発行されています
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	セグメントが発行されていません
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_MEMF	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

セグメントの発行解除

`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	セグメントが発行されていません
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あと、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	アクセス権が拒否されました
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_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あと、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	アクセス権が拒否されました
RSMERR_SEG_NOT_PUBLISHED_TO_NODE	セグメントがノードに発行されていません
RSMERR_SEG_NOT_PUBLISHED	セグメントが発行されていません
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あと、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のデータを、ハンドルとオフセットで指定された対応するインポートされたセグメントの場所に書き込みます。

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行します。

`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あと、呼び出し元は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`	アクセス権が拒否されました
`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`) の入出力ベクトルコンポーネントを使用すると、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。通知イベントの送信を指定するには、`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`	アクセス権が拒否されました
`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 */

    int        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;
    size_t     local_offset;             /* offset from handle base vaddr */
    size_t     import_segment_offset;    /* offset from segment base vaddr */
```

例 2–1 プライマリデータ構造体 (続き)

```
    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ーの相互接続 (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,    /* system will choose available virtual address */
    RSM_MAP_FIXED = 0x1,    /* map segment at specified virtual address */
} 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`	アクセス権が無効です
`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。

- 初期化
- 開く
- 閉じる
- 順番の決定

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。バリアを開いたあと、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 に設定する前に、 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() 操作および rsm_intr_signal_wait() 操作を使用した場合、カーネルへの ioctl 呼び出しを処理する必要があります。

シグナルの送信

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

void 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*);

void 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	タイマーが期限切れです
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発行されていない場合、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、セグメントを発行解除、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法

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 プロパティー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。このプロパティーの値が 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。このプロパティーの値が 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。このプロパティーのデフォルトの値は 1 です。

◆ ◆ ◆ 第 3 章

3 セッション記述プロトコル API

セッション記述プロトコル (SDP) は、マルチメディアセッションを記述します。この章で説明する SDP API には、アプリケーションに SDP の機能を追加するために使用できる関数呼び出しが含まれます。

セッション記述 API の概要

SDP API を構成する関数呼び出しは、共有オブジェクト `libcommputil.so.1` によって提供されます。この共有オブジェクト内の関数は、SDP 記述を解析し、記述の構文を確認します。

`sdp.h` ヘッダーファイルは、`sdp_session_t` 構造体を定義します。この構造体には次のメンバーが含まれます。

```
typedef    struct sdp_session {
   int             sdp_session_version; /* SDP session verstion */
   int             s_version;           /* SDP version field */
   sdp_origin_t    *s_origin;           /* SDP origin field */
   char            *s_name;             /* SDP name field */
   char            *s_info;             /* SDP info field */
   char            *s_uri;              /* SDP uri field */
   sdp_list_t      *s_email;            /* SDP email field */
   sdp_list_t      *s_phone;            /* SDP phone field */
   sdp_conn_t      *s_conn;             /* SDP connection field */
   sdp_bandwidth_t *s_bw;               /* SDP bandwidth field */
   sdp_time_t      *s_time;             /* SDP time field */
   sdp_zone_t      *s_zone;             /* SDP zone field */
   sdp_key_t       *s_key;              /* SDP key field */
   sdp_attr_t      *s_attr;             /* SDP attribute field */
   sdp_media_t     *s_media;            /* SDP media field */
} sdp_session_t;
```

`sdp_session_version` メンバーは、構造体のバージョンを追跡します。`sdp_session_version` メンバーの初期値は `SDP_SESSION_VERSION_1` です。

`sdp_origin_t` 構造体には次のメンバーが含まれます。

```
typedef struct sdp_origin {
   char        *o_username;  /* username of the originating host */
   uint64_t    o_id;         /* session id */
   uint64_t    o_version;    /* version number of this session */
                             /* description */
   char        *o_nettype;   /* type of network */
   char        *o_addrtype;  /* type of the address */
   char        *o_address;   /* address of the machine from which */
                             /* session was created */
} sdp_origin_t;
```

sdp_conn_t 構造体には次のメンバーが含まれます。

```
typedef struct sdp_conn {
   char    *c_nettype;         /* type of network */
   char    *c_addrtype;        /* type of the address */
   char    *c_address;         /* unicast-address or multicast */
                               /* address */
   int     c_addrcount;        /* number of addresses (case of */
                               /* multicast address with layered */
                               /* encodings */
   struct  sdp_conn *c_next;   /* pointer to next connection */
                               /* structure; there could be several */
                               /* connection fields in SDP description */
   uint8_t c_ttl;              /* TTL value for IPV4 multicast address */
} sdp_conn_t;
```

sdp_bandwidth_t 構造体には次のメンバーが含まれます。

```
typedef struct sdp_bandwidth {
   char                 *b_type; /* info needed to interpret b_value */
   uint64_t             b_value; /* bandwidth value */
   struct sdp_bandwidth  *b_next; /* pointer to next bandwidth structure*/
                                  /* (there could be several bandwidth */
                                  /* fields in SDP description */
} sdp_bandwidth_t;
```

sdp_list_t 構造体は、void ポインタのリンクリストです。この構造体は SDP フィールドを保持します。email や phone などの SDP 構造体フィールドの場合は、void ポインタは文字バッファーを指します。offset フィールドが繰り返される場合のように、要素の数が事前に定義されていない場合は、この構造体を使用して情報を保持します。その場合、void ポインタは整数値を保持します。

sdp_list_t 構造体には次のメンバーが含まれます。

```
typedef struct sdp_list {
   void            *value; /* string values in case of email, phone and */
                           /* format (in media field) or integer values */
                           /* in case of offset (in repeat field) */
   struct sdp_list *next;  /* pointer to the next node in the list */
} sdp_list_t;
```

sdp_repeat_t 構造体は、時間構造体 sdp_time_t に必ず含まれます。repeat フィールドが SDP 記述に単独で現れることはなく、常に time フィールドに関連付けられています。

sdp_repeat_t 構造体には次のメンバーが含まれます。

```
typedef struct sdp_repeat {
   uint64_t          r_interval; /* repeat interval, e.g. 86400 seconds */
                                 /* (1 day) */
   uint64_t          r_duration; /* duration of session, e.g. 3600 */
                                 /* seconds (1 hour) */
   sdp_list_t        *r_offset;  /* linked list of offset values; each */
                                 /* represents offset from start-time */
                                 /* in the SDP time field */
   struct sdp_repeat *r_next;    /* pointer to next repeat structure; */
                                 /* there could be several repeat */
                                 /* fields in the SDP description */
```

sdp_time_t 構造体には次のメンバーが含まれます。

```
typedef struct sdp_time {
   uint64_t         t_start;   /* start-time for a session */
   uint64_t         t_stop;    /* end-time for a session */
   sdp_repeat_t     *t_repeat; /* points to the SDP repeat field */
   struct sdp_time  *t_next;   /* pointer to next time field; there */
                               /* could there could be several time */
                               /* fields in SDP description */
} sdp_time_t;
```

sdp_zone_t 構造体には次のメンバーが含まれます。

```
typedef struct sdp_zone {
   uint64_t        z_time;    /* base time */
   char            *z_offset; /* offset added to z_time to determine */
                              /* session time; mainly used for daylight */
                              /* saving time conversions */
   struct sdp_zone *z_next;   /* pointer to next zone field; there */
                              /* could be several <adjustment-time> */
                              /* <offset> pairs within a zone field */
} sdp_zone_t;
```

sdp_key_t 構造体には次のメンバーが含まれます。

```
typedef struct sdp_key {
   char   *k_method;   /* key type */
   char   *k_enckey;   /* encryption key */
 } sdp_key_t;
```

sdp_attr_t 構造体には次のメンバーが含まれます。

```
typedef struct sdp_attr {
   char            *a_name;  /* name of the attribute */
   char            *a_value; /* value of the attribute */
   struct sdp_attr *a_next;  /* pointer to the next attribute */
                             /* structure; there could be several */
                             /* attribute fields within SDP description */
} sdp_attr_t;
```

sdp_media_t 構造体には次のメンバーが含まれます。

```
typedef struct sdp_media {
   char              *m_name;     /* name of the media such as "audio", */
                                  /* "video", "message" */
   uint_t            m_port;      /* transport layer port information */
   int               m_portcount; /* number of ports in case of */
                                  /* hierarchically encoded streams */
   char              *m_proto;    /* transport protocol */
   sdp_list_t        *m_format;   /* media format description */
   char              *m_info;     /* media info field */
   sdp_conn_t        *m_conn;     /* media connection field */
   sdp_bandwidth_t   *m_bw;       /* media bandwidth field */
   sdp_key_t         *m_key;      /* media key field */
   sdp_attr_t        *m_attr;     /* media attribute field */
   struct sdp_media  *m_next;     /* pointer to next media structure; */
                                  /* there could be several media */
                                  /* sections in SDP description */
   sdp_session_t     *m_session;  /* pointer to the session structure */
} sdp_media_t;
```

SDP 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。

- SDP セッション構造体の作成
- SDP セッション構造体内の検索
- SDP セッション構造体のシャットダウン
- ユーティリティー関数

SDP セッション構造体の作成

新しい SDP セッション構造体を作成するには、最初に `sdp_new_session()` 関数を呼び出して新しい構造体用のメモリーを割り当てます。この関数は新しいセッション構造体へのポインタを返します。このセクションに示すほかの関数は、このポインタを使用して新しいセッション構造体を作成します。新しいセッション構造体を作成したら、`sdp_session_to_str()` 関数を使用してセッション構造体を文字列表現に変換します。

新しい SDP セッション構造体の作成

```
sdp_session_t *sdp_new_session();
```

`sdp_new_session()` 関数は、`session` パラメータで指定された新しい SDP セッション構造体用のメモリーを割り当て、新しい構造体にバージョン番号を割り当てます。セッション構造体に割り当てられたメモリーは、`sdp_free_session()` 関数を呼び出すことで解放できます。

戻り値: `sdp_new_session()` 関数は、関数が正常に完了したときに、新しく割り当てられた SDP セッション構造体を返します。エラー発生時には NULL を返します。

SDP セッション構造体への発信元フィールドの追加

```
int sdp_add_origin(sdp_session_t *session, const char *name, uint64_t id,
uint64_t ver, const char *nettype, const char *addrtype, const char *address);
```

sdp_add_origin() 関数は、*name*、*id*、*ver*、*nettype*、*addrtype*、*address* の各パラメータを使用して、session パラメータの値で指定されたセッション構造体 (sdp_session_t) に Origin (o=) SDP フィールドを追加します。

戻り値: sdp_add_origin() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への名前フィールドの追加

```
int sdp_add_name(sdp_session_t *session, const char *name);
```

sdp_add_name() 関数は、*name* パラメータを使用して、session パラメータの値で指定されたセッション構造体 (sdp_session_t) に SessionName (s=) SDP フィールドを追加します。

戻り値: sdp_add_name() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への情報フィールドの追加

```
int sdp_add_information(char **information, const char *value);
```

sdp_add_information() 関数は、*value* パラメータを使用して、セッション構造体 (sdp_session_t) またはメディア構造体 (sdp_media_t) に Info (i=) SDP フィールドを追加します。このフィールドは、SDP 記述のメディアセクションまたはセッションセクションに格納されます。最初の引数として &session->s_info または &media->m_info を渡すことにより、セクションを指定する必要があります。

戻り値: sdp_add_information() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への URI フィールドの追加

```
int sdp_add_uri(sdp_session_t *session, const char *uri);
```

sdp_add_uri() 関数は、uri パラメータを使用して、session パラメータの値で指定されたセッション構造体 (sdp_session_t) に URI (*u*=) SDP フィールドを追加します。

戻り値: sdp_add_uri() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への電子メールフィールドの追加

```
int sdp_add_email(sdp_session_t *session, const char *email);
```

sdp_add_email() 関数は、email パラメータを使用して、session パラメータの値で指定されたセッション構造体 (sdp_session_t) に Email (*e*=) SDP フィールドを追加します。

戻り値: sdp_add_email() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への電話フィールドの追加

```
int sdp_add_phone(sdp_session_t *session, const char *email);
```

sdp_add_phone() 関数は、phone パラメータを使用して、session パラメータの値で指定されたセッション構造体 (sdp_session_t) に Phone (*p*=) SDP フィールドを追加します。

戻り値: sdp_add_phone() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への接続フィールドの追加

```
int sdp_add_connection(sdp_conn_t **conn, const char *nettype, const char
*addrtype, const char *address, uint8_t ttl, int addrcount);
```

sdp_add_connection() 関数は、*nettype*、*addrtype*、*address*、*ttl*、*addrcount* の各パラメータを使用して、セッション構造体 (sdp_session_t) またはメディア構造体 (sdp_media_t) に Connection (c=) SDP フィールドを追加します。IPv4 または IPv6 のユニキャストアドレスの場合は、*ttl* パラメータと *addrcount* パラメータの値をゼロに設定します。マルチキャストアドレスの場合は、*ttl* パラメータの値を 0 - 255 の範囲に設定します。マルチキャストアドレスの場合は、addrcount パラメータの値をゼロに設定できません。

このフィールドは、SDP 記述のメディアセクションまたはセッションセクションに格納されます。最初の引数として &session->s_info または &media->m_info を渡すことにより、セクションを指定する必要があります。

戻り値: sdp_add_connection() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への帯域幅フィールドの追加

```
int sdp_add_bandwidth(sdp_bandwidth_t **bw, const char *type, uint64_t value);
```

sdp_add_bandwidth() 関数は、*type* パラメータと *value* パラメータを使用して、セッション構造体 (sdp_session_t) またはメディア構造体 (sdp_media_t) に Bandwidth (b=) SDP フィールドを追加します。

このフィールドは、SDP 記述のメディアセクションまたはセッションセクションに格納されます。最初の引数として &session->s_info または &media->m_info を渡すことにより、セクションを指定する必要があります。

戻り値: sdp_add_bandwidth() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への時間フィールドの追加

```
int sdp_add_time(sdp_session_t *session, uint64_t starttime, uint64_t stoptime,
sdp_time_t **time);
```

sdp_add_time() 関数は、*starttime* パラメータと *stoptime* パラメータの値を使用して、セッション構造体に Time (t=) SDP フィールドを追加します。この関数は、新しい時間構造体を作成し、*time* パラメータにその構造体へのポインタを返します。

戻り値: sdp_add_time() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への繰り返しフィールドの追加

```
int sdp_add_repeat(sdp_time_t *time, uint64_t interval, uint64_t duration, const
char *offset);
```

sdp_add_repeat() 関数は、*interval*、*duration*、*offset* の各パラメータの値を使用して、セッション構造体に RepeatTime (r=) SDP フィールドを追加します。*offset* パラメータの値は、1 つ以上のオフセット値を保持する文字列 (60、60 1d 3h など) です。*time* パラメータの値は、sdp_add_time() 関数が作成する時間構造体へのポインタです。

戻り値: sdp_add_repeat() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体へのゾーンフィールドの追加

```
int sdp_add_zone(sdp_session_t *session, uint64_t time, const char *offset);
```

sdp_add_zone() 関数は、*time* パラメータと *offset* パラメータを使用して、session パラメータの値で指定されたセッション構造体 (sdp_session_t) に TimeZoneAdjustment (z=) SDP フィールドを追加します。1 つのゾーンフィールドに対して複数の時間およびオフセットの値を追加するには、時間/オフセットのペアごとにこの関数を呼び出します。

戻り値: sdp_add_zone() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体へのキーフィールドの追加

```
int sdp_add_key(sdp_key_t **key, const char *method, const char *enckey);
```

sdp_add_key() 関数は、*method* パラメータと *enckey* パラメータを使用して、セッション構造体 (sdp_session_t) またはメディア構造体 (sdp_media_t) に Key (k=) SDP フィールドを追加します。このフィールドは、SDP 記述のメディアセクションまたはセッションセクションに格納されます。最初の引数として &session->s_info または &media->m_info を渡すことにより、セクションを指定する必要があります。

戻り値: sdp_add_key() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体への属性フィールドの追加

```
int sdp_add_attribute(sdp_attr_t **attr, const char *name, const char *value);
```

sdp_add_attribute() 関数は、*name* パラメータと *value* パラメータを使用して、セッション構造体 (sdp_session_t) またはメディア構造体 (sdp_media_t) に Attribute (a=) SDP フィールドを追加します。このフィールドは、SDP 記述のメディアセクションまたはセッションセクションに格納されます。最初の引数として &session->s_info または &media->m_info を渡すことにより、セクションを指定する必要があります。

戻り値: sdp_add_attribute() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP セッション構造体へのメディアフィールドの追加

int **sdp_add_media**(sdp_session_t **session*, const char **name*, uint_t *port*, int *portcount*, const char **protocol*, const char **format*, sdp_media_t ***media*);

sdp_add_media() 関数は、*name*、*port*、*portcount*、*protocol*、*format* の各パラメータの値を使用して、session パラメータの値で指定されたセッション構造体 (sdp_session_t) に Media (m=) SDP フィールドを追加します。*format* パラメータは、1 つ以上の値を保持する文字列 (0 32 97 など) です。

この関数は、新しいメディア構造体を作成し、*media* パラメータにその構造体へのポインタを返します。メディア構造体に SDP フィールドを追加する関数は、このポインタを使用します。

戻り値: sdp_add_media() 関数は、関数が正常に完了したときに 0 を返します。必須のパラメータがなかった場合は、EINVAL を返します。メモリーの割り当てに失敗した場合は、ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

コード例: SDP セッション構造体の作成

この例では、このセクションに示した関数を使用して、新しい SDP セッション構造体を作成し、構造体にフィールドを追加し、完成した構造体を文字列表現に変換します。この例では、最後にプログラムから sdp_free_session() 関数を呼び出して、セッションを解放します。

例 3–1 SDP セッション構造体の作成

```
/* SDP Message we will be building
"v=0\r\n\
o=Alice 2890844526 2890842807 IN IP4 10.47.16.5\r\n\
s=-\r\n\
i=A Seminar on the session description protocol\r\n\
u=http://www.example.com/seminars/sdp.pdf\r\n\
e=alice@example.com (Alice Smith)\r\n\
p=+1 911-345-1160\r\n\
c=IN IP4 10.47.16.5\r\n\
b=CT:1024\r\n\
t=2854678930 2854679000\r\n\
r=604800 3600 0 90000\r\n\
z=2882844526 -1h 2898848070 0h\r\n\
a=recvonly\r\n\
m=audio 49170 RTP/AVP 0\r\n\
i=audio media\r\n\
b=CT:1000\r\n\
```

例 3–1 SDP セッション構造体の作成 *(続き)*

```
k=prompt\r\n\
m=video 51372 RTP/AVP 99 90\r\n\
i=video media\r\n\
a=rtpmap:99 h232-199/90000\r\n\
a=rtpmap:90 h263-1998/90000\r\n"
*/

#include stdio.h>
#include string.h>
#include errno.h>
#include sdp.h>

int main ()
{
   sdp_session_t  *my_sess;
   sdp_media_t    *my_media;
   sdp_time_t     *my_time;
   char *b_sdp;

   my_sess = sdp_new_session();
   if (my_sess == NULL) {
return (ENOMEM);
   }
   my_sess->version = 0;
   if (sdp_add_name(my_sess, "-") != 0)
goto err_ret;
   if (sdp_add_origin(my_sess, "Alice", 2890844526ULL, 2890842807ULL,
 "IN", "IP4", "10.47.16.5") != 0)
goto err_ret;
   if (sdp_add_information(&my_sess->s_info, "A Seminar on the session"
 "description protocol") != 0)
goto err_ret;
   if (sdp_add_uri (my_sess, "http://www.example.com/seminars/sdp.pdf")
 != 0)
goto err_ret;
   if (sdp_add_email(my_sess, "alice@example.com (Alice smith)") != 0)
goto err_ret;
   if (sdp_add_phone(my_sess, "+1 911-345-1160") != 0)
goto err_ret;
   if (sdp_add_connection(&my_sess->s_conn, "IN", "IP4", "10.47.16.5",
0, 0) != 0)
goto err_ret;
   if (sdp_add_bandwidth(&my_sess->s_bw, "CT", 1024) != 0)
goto err_ret;
   if (sdp_add_time(my_sess, 2854678930ULL, 2854679000ULL, &my_time)
!= 0)
goto err_ret;
   if (sdp_add_repeat(my_time, 604800ULL, 3600ULL, "0 90000") != 0)
goto err_ret;
   if (sdp_add_zone(my_sess, 2882844526ULL, "-1h") != 0)
goto err_ret;
   if (sdp_add_zone(my_sess, 2898848070ULL, "0h") != 0)
goto err_ret;
   if (sdp_add_attribute(&my_sess->s_attr, "sendrecv", NULL) != 0)
goto err_ret;
   if (sdp_add_media(my_sess, "audio", 49170, 1, "RTP/AVP",
```

例 3–1 SDP セッション構造体の作成 (続き)

```
"0", &my_media) != 0)
goto err_ret;
   if (sdp_add_information(&my_media->m_info, "audio media") != 0)
goto err_ret;
   if (sdp_add_bandwidth(&my_media->m_bw, "CT", 1000) != 0)
goto err_ret;
   if (sdp_add_key(&my_media->m_key, "prompt", NULL) != 0)
goto err_ret;
   if (sdp_add_media(my_sess, "video", 51732, 1, "RTP/AVP",
 "99 90", &my_media) != 0)
goto err_ret;
   if (sdp_add_information(&my_media->m_info, "video media") != 0)
goto err_ret;
   if (sdp_add_attribute(&my_media->m_attr, "rtpmap",
      "99 h232-199/90000") != 0)
goto err_ret;
   if (sdp_add_attribute(&my_media->m_attr, "rtpmap",
      "90 h263-1998/90000") != 0)
goto err_ret;
   b_sdp = sdp_session_to_str(my_sess, &error);

   /*
    * b_sdp is the string representation of my_sess structure
    */

   free(b_sdp);
   sdp_free_session(my_sess);
   return (0);
err_ret:
   free(b_sdp);
   sdp_free_session(my_sess);
   return (1);
}
```

SDP セッション構造体の検索

このセクションに示す関数は、SDP セッション構造体から特定の値を検索し、見つかった値へのポインタを返します。

SDP セッション構造体内の属性の検索

`sdp_attr_t *`**`sdp_find_attribute`** `(sdp_attr_t *`*`attr`*`, const char *`*`name`*`);`

`sdp_find_attribute()` 関数は、*attr* パラメータで指定された属性リストから、*name* パラメータで指定された属性名を検索します。

戻り値: `sdp_find_attribute()` 関数は、関数が正常に完了したときに、*name* パラメータで指定された属性へのポインタ (`sdp_attr_t *`) を返します。それ以外の場合、`sdp_find_attribute()` 関数は NULL 値を返します。

例 3–2 sdp_find_attribute() 関数の使用

この例では、不完全な SDP 記述にオーディオセクションが含まれています。

```
m=audio 49170 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=sendonly
a=ptime:10000
a=maxptime:20000

/*
* Assuming that above description is parsed using sdp_parse and that
* the parsed structure is in "session" sdp_session_t structure.
*/

sdp_attr_t *ptime;
sdp_attr_t *max_ptime;
sdp_media_t *media = session->s_media;

if ((ptime = sdp_find_attribute(media->m_attr, "ptime")) == NULL)
/* ptime attribute not present */
else if((max_ptime = sdp_find_attribute(media->m_attr,
 "maxptime")) == NULL)
/* max_ptime attribute not present */
```

SDP セッション構造体内のメディアの検索

sdp_media_t ***sdp_find_media**(sdp_media_t **media*, const char **name*);

sdp_find_media() 関数は、*media* パラメータで指定されたメディアリストから、*name* パラメータで指定されたメディアエントリを検索します。

戻り値: sdp_find_media() 関数は、関数が正常に完了したときに、name パラメータで指定されたメディアリストエントリへのポインタ (*sdp_media_t* *) を返します。それ以外の場合、sdp_find_media() 関数は NULL 値を返します。

例 3–3 sdp_find_media() 関数の使用

この例では、不完全な SDP 記述に 2 つのセクション (オーディオセクションとビデオセクション) が含まれています。

```
m=audio 49170 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
m=video 51372 RTP/AVP 31 32
a=rtpmap:31 H261/90000
a=rtpmap:32 MPV/90000

/*
* Assuming that above description is parsed using sdp_parse() and that
* the parsed structure is in "session" sdp_session_t structure.
*/

sdp_media_t     *my_media;
```

例 3–3 sdp_find_media() 関数の使用 (続き)

```
my_media = sdp_find_media(session->s_media, "video");

/*
* my_media now points to the structure containg video media section
* information
*/
```

SDP セッション構造体内のメディア形式の検索

sdp_attr_t ***sdp_find_media_rtpmap*** (sdp_media_t **media*, const char **format*);

sdp_find_media_rtpmap() 関数は、*media* パラメータで指定されたメディア構造体の属性リストから、*format* パラメータで指定された形式エントリを検索します。

戻り値: sdp_find_media_rtpmap() 関数は、関数が正常に完了したときに、name パラメータで指定された形式エントリへのポインタ (*sdp_attr_t* *) を返します。それ以外の場合、sdp_find_media() 関数は NULL 値を返します。

例 3–4 sdp_find_media_rtpmap() 関数の使用

この例では、不完全な SDP 記述に 2 つのセクション (オーディオセクションとビデオセクション) が含まれています。

```
m=audio 49170 RTP/AVP 0 8
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
m=video 51372 RTP/AVP 31 32
a=rtpmap:31 H261/90000
a=rtpmap:32 MPV/90000

/*
* Assuming that above description is parsed using sdp_parse() and that
* the parsed structure is in "session" sdp_session_t structure.
*/

sdp_media_t     *video;
sdp_attr_t      *mpv;

video = sdp_find_media(session->s_media, "video);
mpv = sdp_find_media_rtpmap(video, "32");

/*
* Now the attribute structure sdp_attr_t, mpv will be having
* values from the attribute field "a=rtpmap:32 MPV/90000"
*/
```

SDP セッション構造体のシャットダウン

このセクションに示す関数は、次の機能を実行します。

- SDP セッション構造体からのフィールドの削除
- SDP セッション構造体の解放

SDP セッション構造体からのフィールドの削除

```
int sdp_delete_all_field(sdp_session_t *session, const char field);
```

sdp_delete_all_field() 関数は、*field* パラメータで指定された SDP フィールドのすべての出現箇所を SDP 構造体から削除します。たとえば、SDP 構造体に 3 つの帯域幅 (b=) フィールドがある場合に、field パラメータに SDP_BANDWIDTH_FIELD という値を指定してこの関数を呼び出すと、セッション構造体から 3 つの 帯域幅フィールドがすべて削除されます。

戻り値: sdp_delete_all_field() 関数は、関数が正常に完了したときに 0 を返します。session 引数が NULL であるか、フィールドタイプが不明の場合は、EINVAL を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP メディア構造体からのフィールドの削除

```
int sdp_delete_all_media_field(sdp_media_t *media, const char field);
```

sdp_delete_all_media_field() 関数は、*field* パラメータで指定された SDP フィールドのすべての出現箇所を SDP メディア構造体から削除します。

戻り値: sdp_delete_all_media_field() 関数は、関数が正常に完了したときに 0 を返します。session 引数が NULL であるか、フィールドタイプが不明の場合は、EINVAL を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP メディア構造体からのメディアの削除

```
int sdp_delete_media(sdp_media_t **l_media, sdp_media_t *media);
```

sdp_delete_media() 関数は、*media* パラメータで指定されたメディアエントリをメディアリストから削除します。この関数は、sdp_find_media() 関数を使用して、指定されたメディアエントリを検索します。この関数は、メディアエントリを削除したあとで、メディア構造体に割り当てられていたメモリーを解放します。

戻り値: sdp_delete_media() 関数は、関数が正常に完了したときに 0 を返します。session 引数が NULL であるか、必須の引数が存在しない場合は、EINVAL を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP メディア構造体からの属性の削除

```
int sdp_delete_attribute(sdp_attr_t **l_attr, sdp_attr_t *attr);
```

sdp_delete_attribute() 関数は、*attr* パラメータで指定された属性をメディアリストから削除します。この関数は、sdp_find_media_rtpmap() 関数または sdp_find_attribute() 関数を呼び出して、指定された属性を検索します。この関数は、属性を削除したあとで、属性構造体に割り当てられていたメモリーを解放します。

戻り値: sdp_delete_attribute() 関数は、関数が正常に完了したときに 0 を返します。session 引数が NULL であるか、必須の引数が存在しない場合は、EINVAL を返します。errno の値は、エラーが発生した場合でも変化しません。

SDP メディア構造体からの属性の削除

void **sdp_free_session**(sdp_session_t **session*);

sdp_free_session() 関数は、*session* パラメータで指定されたセッションを破棄し、その構造体に関連付けられたリソースを解放します。

SDP API ユーティリティー関数

このセクションに示す関数は、SDP セッション構造体の解析と生成、既存セッションの複製、および既存セッションの文字列表現への変換を行います。

SDP セッション構造体の解析

int **sdp_parse**(const char **sdp_info*, int *len*, int *flags*, sdp_session_t ***session*, uint_t **p_error*);

sdp_parse() 関数は、*sdp_info* パラメータの SDP 記述を解析し、*sdp_session_t* 構造体を生成します。*len* パラメータは、文字バッファー *sdp_info* の長さを指定します。この関数は、*sdp_session_t* 構造体に必要なメモリーを割り当てます。このメモリーを解放するには、sdp_free_session() 関数を呼び出します。

flags パラメータの値は、ゼロに設定する必要があります。*flags* パラメータの値がゼロでない場合、sdp_parse() 関数は失敗し、戻り値として EINVAL を返し、*session の値を NULL に設定します。

p_error パラメータは、解析エラーが発生したフィールドの値を取ります。このパラメータの値が NULL になることはありません。*p_error* パラメータの取り得る値は、次のリストのとおりです。

```
SDP_VERSION_ERROR             0x00000001
SDP_ORIGIN_ERROR              0x00000002
SDP_NAME_ERROR                0x00000004
SDP_INFO_ERROR                0x00000008
SDP_URI_ERROR                 0x00000010
SDP_EMAIL_ERROR               0x00000020
SDP_PHONE_ERROR               0x00000040
```

```
SDP_CONNECTION_ERROR          0x00000080
SDP_BANDWIDTH_ERROR           0x00000100
SDP_TIME_ERROR                0x00000200
SDP_REPEAT_TIME_ERROR         0x00000400
SDP_ZONE_ERROR                0x00000800
SDP_KEY_ERROR                 0x00001000
SDP_ATTRIBUTE_ERROR           0x00002000
SDP_MEDIA_ERROR               0x00004000
SDP_FIELDS_ORDER_ERROR        0x00008000
SDP_MISSING_FIELDS            0x00010000
```

SDP 構造体のフィールドの順序が誤っており、RFC 4566 に違反している場合、sdp_parse() 関数は *p_error* パラメータの値を SDP_FIELDS_ORDER_ERROR に設定します。SDP 構造体に必須のフィールドがなく、RFC 4566 に違反している場合、sdp_parse() 関数は *p_error* パラメータの値を SDP_MISSING_FIELDS に設定します。

sdp_parse() 関数は、解析エラーがあるフィールドを処理したあとも解析を続行しますが、解析エラーがあったフィールドは、結果として得られる sdp_session_t 構造体に含まれません。

戻り値: sdp_parse() 関数は、関数が正常に完了したときに 0 を返します。session 引数が無効である場合、sdp_parse() 関数は EINVAL を返します。sdp_info の解析中にメモリー割り当てに失敗した場合、sdp_parse() 関数は ENOMEM を返します。errno の値は、エラーが発生した場合でも変化しません。

例 3–5 例: SDP セッション構造体の解析

この例で使用する SDP セッション構造体は次のとおりです。

```
v=0\r\n
o=jdoe 23423423 234234234 IN IP4 192.168.1.1\r\n
s=SDP seminar\r\n
i=A seminar on the session description protocol\r\n
e=test@host.com
c=IN IP4 156.78.90.1\r\n
t=2873397496 2873404696\r\n
```

sdp_parse_t() 関数を呼び出したあとで、結果として得られる sdp_session_t 構造体は次のとおりです。

```
session {
        sdp_session_version = 1
        s_version = 0
        s_origin {
                o_username = "jdoe"
                o_id = 23423423ULL
                o_version = 234234234ULL
                o_nettype = "IN"
                o_addrtype = "IP4"
                o_address = "192.168.1.1"
        }
        s_name = "SDP seminar"
        s_info = "A seminar on the session description protocol"
```

例 3–5 例: SDP セッション構造体の解析 *(続き)*

```
        s_uri =  (nil)
        s_email {
                value = "test@host.com"
                next = (nil)
        }
        s_phone = (nil)
        s_conn {
                c_nettype = "IN"
                c_addrtype = "IP4"
                c_address = "156.78.90.1"
                c_addrcount = 0
                c_ttl = 0
                c_next = (nil)
        }
        s_bw = (nil)
        s_time {
                t_start = 2873397496ULL
                t_stop = 2873404696ULL
                t_repeat = (nil)
                t_next = (nil)
        }
        s_zone = (nil)
        s_key = (nil)
        s_attr = (nil)
        s_media = (nil)
}
```

既存の SDP セッション構造体の複製

```
sdp_session_t sdp_clone_session(const sdp_session_t *session);
```

`sdp_clone_session()` 関数は、*session* パラメータで識別される SDP セッション構造体と同一の新しい SDP セッション構造体を作成します。`sdp_clone_session()` 関数は、正常完了時に、複製されたセッション構造体を返します。`sdp_clone_session()` 関数は、失敗時に NULL を返します。

SDP セッション構造体の文字列への変換

```
char *sdp_session_to_str(const sdp_session_t *session, int *error);
```

`sdp_session_to_str()` 関数は、*session* パラメータで指定された SDP セッション構造体の文字列表現を返します。`sdp_session_to_str()` 関数は、各 SDP フィールドに文字列を追加する前に、フィールドの最後にキャリッジリターン/改行を追加します。

戻り値: `sdp_session_to_str()` 関数は、正常完了時に SDP セッション構造体の文字列表現を返します。それ以外の場合、`sdp_session_to_str()` 関数は NULL を返します。入力が NULL だった場合、`sdp_session_to_str()` 関数は EINVAL へのエラーポインタを返します。メモリー割り当てエラーが発生した場合、`sdp_session_to_str()` 関数は ENOMEM へのエラーポインタを返します。`errno` の値は、エラーが発生した場合でも変化しません。

◆ ◆ ◆ 第 4 章

4 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。

- 73 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。
- 78 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。
- 81 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。
- 82 ページの「スケジューリングとシステム性能」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もっとも低いスケジューリングクラスです。

SunOS 5.10 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もっとも高く、プロセスが実行可能になり次第 CPU を取得できます。

SunOS 5.10 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 12 章「リアルタイムプログラミングと管理」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します。
- `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す。

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。

デフォルトの構成では、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もっと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、またはより高い優先順位を持つ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構成内容は、`priocntl(1)` コマンドと `dispadmin(1M)` コマンドで表示できます。

以降のセクションでは、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

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を提供し、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より高い優先順位を持つプロセスが実行可能になると、より高い優先順位を持つプロセスが実行中のプロセスを横取りします。

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、構成されている値の、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構成します。たとえば、デフォルトの構成では、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 種類まで構成できます。

スケジューラは、タイムシェアリングのパラメータテーブル `ts_dptbl(4)` 内の構成可能なパラメータを使用して、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)` 内の構成可能なパラメータを使用して、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

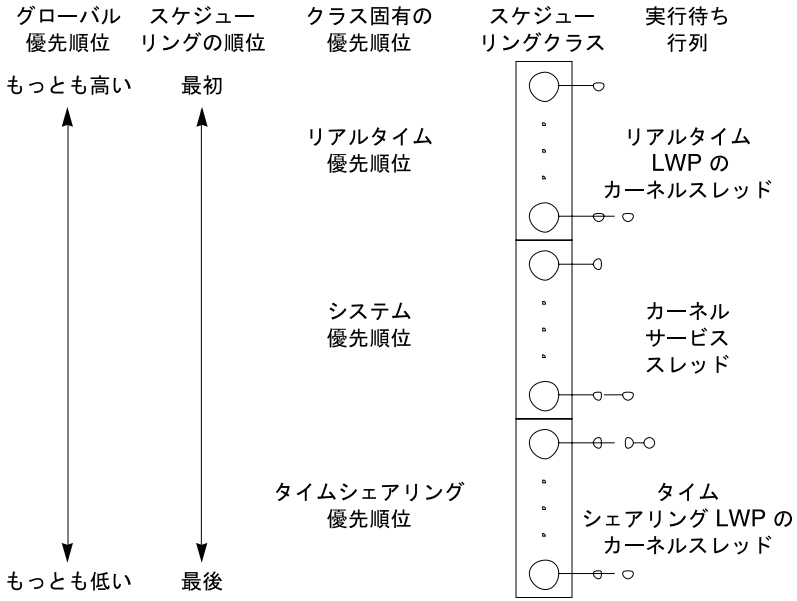
は、FX クラスは TS クラス、IA クラス、および FSS クラスと同じ優先順位の範囲を使用します。FX クラスを使用すると、ユーザーまたはアプリケーションはこのクラス内のプロセスに割り当てられたユーザー指定の優先順位値を使用して、スケジューリング優先順位を制御できます。このようなユーザー指定の優先順位値は、固定優先順位プロセスのスケジューリング優先順位をそのクラス内のほかのプロセスと相対的に決定します。

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 4–1　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構成する場合、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、『Oracle Solaris の管理: 基本管理』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ータを設定または取得するために使用されます。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ータを指定する。
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) ユーティリティーは、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` 構成情報を表示します

`priocntl -d` プロセスのスケジューリングパラメータを表示します

`priocntl -s` プロセスのスケジューリングパラメータを設定します

`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します

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構成が出力されます。

```
$ 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 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。前述の例は、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) 呼び出しにより管理できるのは、LWP、単独のプロセス、またはプロセスのグループです。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、構成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。このセクションでは、スケジューリングの変更がどのようにほか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 の使用法

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ータは、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 の使用法

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。

スケジューリングとシステム性能

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。priocntl(2) 呼び出しによってのみ、プロセスのクラスが変更できます。

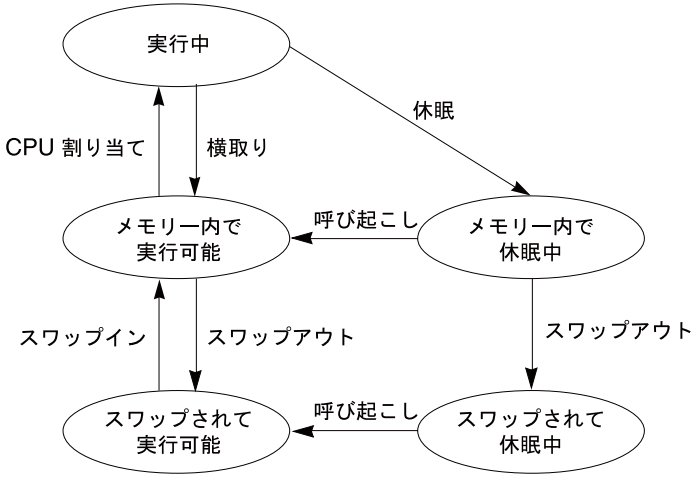
リアルタイムプロセス優先順位は、どのタイムシェアリングプロセスよりも優先順位が高くなっています。リアルタイムプロセスが実行可能である間、タイムシェアリングプロセスやシステムプロセスは実行できません。CPU の制御に失敗することがあるリアルタイムアプリケーションは、その他のユーザーや重要なカーネルハウスキーピングを完全にロックアウトする可能性があります。

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。性能にもっとも影響する要因は、CPU、プライマリメモリー量、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、すべての性能要因を表示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セカンダリ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 4–2 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す。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ます。元の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である場合も、プロセスは横取りされます。
- プロセスがプライマリメモリー内にあり、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す。
- プロセスがプライマリ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す。たとえば、入出力操作の完了、ロックされているリソースの解放、またはタイマの終了を待っている間、プロセスは休眠します。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ます。休眠の原因が解消されると、プロセスは実行可能になります。
- 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セカンダリ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、スワップされています。

- 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セカンダリ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、スワップされています。

  動作中のプロセスをすべて保留するために十分なプライマリメモリーがマシンにない場合は、アドレス空間の一部をセカンダリ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。
- システムのプライマリ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セカンダリメモリーに書き込まれますが、そのプロセスは実行可能なままです。実行中のプロセスがそのページにアクセスする場合、ページがプライマリメモリー内に読み戻されるまでプロセスは休眠します。
- システムのプライマリ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セカンダリメモリーに書き込まれます。このようにセカンダリメモリーに書き込まれたページは「スワップされた」とマークされます。このようなプロセスがスケジューリング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す。

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プライマリ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構成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、283 ページの「ディスパッチ応答時間」を参照してください。

◆ ◆ ◆ 第 5 章

5 近傍性グループ API

この章では、近傍性グループとやりとりするために、アプリケーションで使用できる API について説明します。

この章では、次の内容について説明します。

- 86 ページの「近傍性グループの概要」では、近傍性グループによる抽象化について説明します。
- 88 ページの「インタフェースバージョンの確認」では、インタフェースに関する情報を提供する関数について説明します。
- 89 ページの「近傍性グループインタフェースの初期化」では、近傍性グループ階層を探索し、内容を検索するために使用するインタフェース部分の初期化およびシャットダウンを実行する関数呼び出しについて説明します。
- 90 ページの「近傍性グループ階層」では、近傍性グループ階層を参照し、階層の特性に関する情報を取得する関数呼び出しについて説明します。
- 92 ページの「近傍性グループの内容」では、近傍性グループの内容に関する情報を取り出す関数呼び出しについて説明します。
- 94 ページの「近傍性グループの特性」では、近傍性グループの特性に関する情報を取り出す関数呼び出しについて説明します。
- 95 ページの「近傍性グループ、スレッド、およびメモリー配置」では、スレッドのメモリー配置を制御する方法とその他のメモリー管理手法について説明します。
- 103 ページの「API の使用例」では、この章で説明した API を使用して、タスクを実行するコーディング例を示します。

近傍性グループの概要

メモリー共有型マルチプロセッサマシンには、複数の CPU が搭載されています。それぞれの CPU は、そのマシンのすべてのメモリーにアクセスできます。メモリー共有型マルチプロセッサには、CPU ごとに特定のメモリー領域に対して、より高速なアクセスを可能にするメモリーアーキテクチャーを採用しているものがあります。

そのようなメモリーアーキテクチャーのマシンで Solaris ソフトウェアを実行した場合、特定の CPU による特定のメモリー領域への最短アクセス時間に関するカーネル情報が提供されると、システムのパフォーマンスを向上させることができます。この情報を処理するために近傍性グループ (lgroup) による抽象化が導入されています。lgroup による抽象化は、メモリー配置の最適化 (MPO) 機能の一部です。

lgroup は CPU およびメモリーを模したデバイスの集合です。それぞれの集合内のデバイスは、決められた応答時間の間隔範囲で集合内の任意のデバイスにアクセスできます。応答時間間隔の値は、その lgroup 内のすべての CPU とすべてのメモリー間の最小の共通応答時間を表します。lgroup を定義する応答時間範囲は、その lgroup のメンバー間の最大応答時間を制限しません。応答時間範囲の値は、そのグループ内の CPU とメモリーのあらゆる組み合わせに共通する最小の応答時間です。

lgroup は階層構造になっています。lgroup 階層は、Directed Acyclic Graph (DAG) です。ツリー構造と似ていますが、lgroup は複数の親を持つことができます。ルート lgroup にはシステムのすべてのリソースが含まれており、子 lgroup を持つことができます。さらに、ルート lgroup にはシステムでもっとも高い応答時間値を持たせることができます。すべての子 lgroup の応答時間値は、ルートよりも低くなります。応答時間値はルートに近いほど高く、葉に近いほど低くなります。

すべての CPU がどのメモリー領域に対しても同じ時間でアクセスするコンピュータは、単一の lgroup として表すことができます (図 5–1 参照)。特定の CPU が特定の領域に対してほかの領域より高速なアクセスが可能となるコンピュータは、複数の lgroup を使用して表すことができます (図 5–2 参照)。

図 5–1 単一近傍性グループの模式図

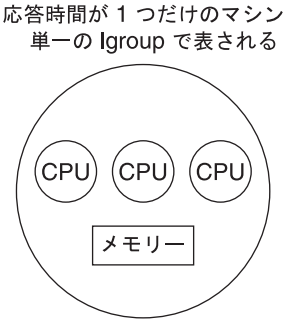
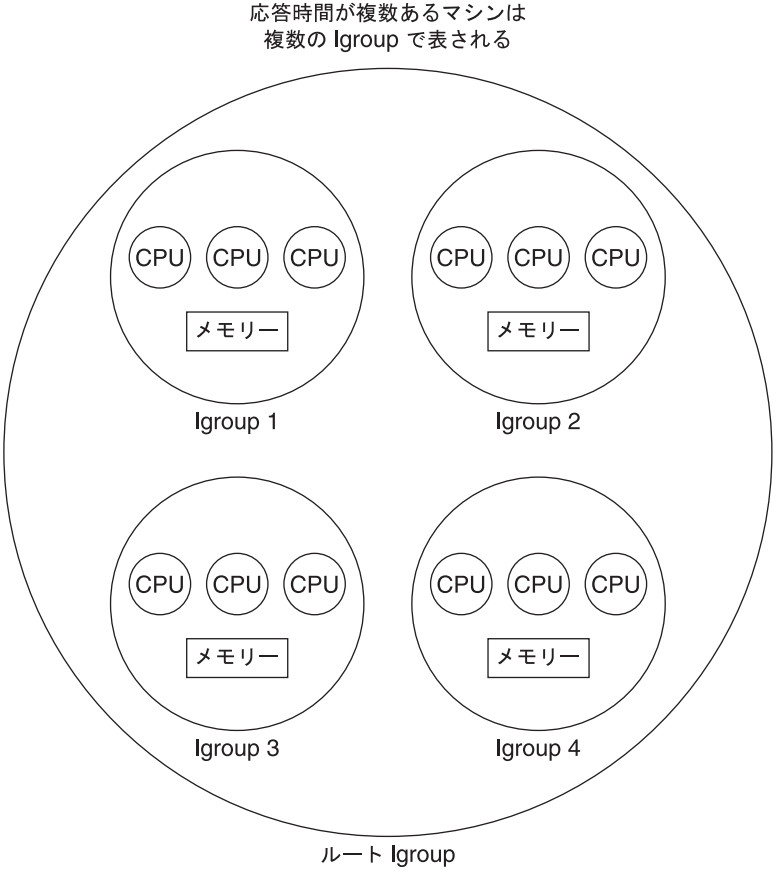


図 5–2 複数近傍性グループの模式図



組織化された lgroup 階層の導入によって、システムでもっとも近いリソースを検索するタスクを簡略化します。各スレッドは、作成時にホーム lgroup に割り当てられます。オペレーティングシステムは、スレッドにホーム lgroup からリソースを割り当てようとデフォルトで試みます。たとえば、Solaris カーネルが、あるスレッドのホーム lgroup にある CPU 上でそのスレッドを実行し、デフォルトでスレッドのホーム lgroup にそのスレッドのメモリーを割り当てるスケジュールを設定しようと試みます。必要なリソースがスレッドのホーム lgroup で利用可能ではない場合には、ホーム lgroup の親から順番に lgroup 階層を探索し、次に近い位置にあるリソースを検索します。必要なリソースがホーム lgroup の親で利用可能でない場合には、カーネルは引き続き lgroup 階層を探索し、そのホーム lgroup のさらに上位ノードの lgroup を順番に探索します。マシン内のすべての lgroup の最上位ノードに位置するのがルート lgroup で、これにはそのマシンのすべてのリソースが含まれます。

lgroup API は、監視と性能チューニングを目的とするアプリケーションのために lgroup の抽象化をエクスポートします。新しい API は、新規ライブラリ `liblgrp` に含まれています。アプリケーションは API を使用して、次のタスクを実行できます。

- グループ階層の検索
- 指定された lgroup の内容と特性の検出
- lgroup でのスレッドとメモリー配置の操作

インタフェースバージョンの確認

lgroup API を使用する前に、`lgrp_version(3LGRP)` 関数を使用して、lgroup インタフェースがサポートされていることを確認する必要があります。`lgrp_version()` 関数には、次の構文があります。

```
#include <sys/lgrp_user.h>
int lgrp_version(const int version);
```

`lgrp_version()` 関数は、lgroup インタフェースのバージョン番号を引数に使用し、システムがサポートしているバージョンを返します。現在の lgroup API の実装で `version` 引数で指定したバージョン番号がサポートされているときは、`lgrp_version()` 関数はそのバージョン番号を返します。サポートされていない場合は、`lgrp_version()` 関数は `LGRP_VER_NONE` を返します。

例 5–1 `lgrp_version()` の使用例

```
#include <sys/lgrp_user.h>
if (lgrp_version(LGRP_VER_CURRENT) != LGRP_VER_CURRENT) {
    fprintf(stderr, "Built with unsupported lgroup interface %d\n",
        LGRP_VER_CURRENT);
    exit (1);
}
```

近傍性グループインタフェースの初期化

アプリケーションは、API を使用して lgroup 階層を参照し、内容を検出するために、lgrp_init(3LGRP) を呼び出す必要があります。lgrp_init() を呼び出すことにより、アプリケーションは lgroup 階層のスナップショットを取得できます。アプリケーション開発者は、特に呼び出したスレッドが利用可能なリソースだけをスナップショットに含めるか、あるいはオペレーティングシステム全般で利用可能なリソースを含めるかを指定できます。lgrp_init() 関数は cookie を返します。cookie は、次のタスクで使用されます。

- lgroup 階層の参照
- lgroup の内容の判定
- スナップショットが最新であるかどうかの判定

lgrp_init() の使用法

lgrp_init() 関数は、 lgroup インタフェースを初期化し、 lgroup 階層のスナップショットを取得します。

```
#include <sys/lgrp_user.h>
lgrp_cookie_t lgrp_init(lgrp_view_t view);
```

引数 view に LGRP_VIEW_CALLER を指定して lgrp_init() 関数を呼び出す場合には、呼び出したスレッドで利用可能なリソースだけがスナップショットとして返されます。引数 view に LGRP_VIEW_OS を指定して lgrp_init() 関数を呼び出す場合は、オペレーティングシステムで利用可能なリソースがスナップショットとして返されます。スレッドが lgrp_init() 関数の呼び出しに成功すると、関数は lgroup 階層とやりとりするすべての関数で使用される cookie を返します。スレッドに cookie が必要でなくなったときは、cookie を引数に指定して lgrp_fini() 関数を呼び出します。

lgroup 階層は、マシン上のすべての CPU とメモリーリソースを含むルート lgroup 階層によって構成されています。ルート lgroup は、応答時間がより短く制限された複数の近傍性グループを持つことができます。

lgrp_init() 関数が返すエラーは 2 種類あります。無効な引数 view が指定された場合、EINVAL を返します。lgroup 階層のスナップショットを割り当てる十分なメモリーがない場合には、ENOMEM を返します。

lgrp_fini() の使用法

lgrp_fini(3LGRP) 関数は、取得した cookie の使用を終了し、対応する lgroup 階層のスナップショットを解放します。

```
#include <sys/lgrp_user.h>
int lgrp_fini(lgrp_cookie_t cookie);
```

lgrp_fini() 関数は、前に lgrp_init() を呼び出して作成した lgroup 階層のスナップショットを表す cookie を引数に使用します。lgrp_fini() 関数は、そのスナップショットに割り当てられたメモリーを解放します。lgrp_fini() を呼び出したあとでは、cookie は無効になります。その cookie を使用することはできません。

lgrp_fini() 関数に渡した cookie が無効な場合、lgrp_fini() は EINVAL を返します。

近傍性グループ階層

このセクションで説明する API を使用することにより、呼び出しスレッドから lgroup 階層を参照できます。lgroup 階層は、Directed Acyclic Graph (DAG) です。ツリー構造と似ていますが、ノードは複数の親を持つことができます。ルート lgroup はマシン全体を表し、ルート lgroup にはそのマシンのすべてのリソースが含まれます。ルート lgroup はシステム全体でもっとも高い応答時間値を持っています。それぞれの子 lgroup はルート lgroup にあるハードウェアのサブセットで構成されています。それぞれの子 lgroup の応答時間値はより低く制限されています。ルートに近い近傍性グループほど、多くのリソースと高い応答時間が与えられています。葉に近い近傍性グループは、リソースも少なく、応答時間も低くなります。lgroup には、その応答時間の範囲内で直下にリソースが含まれる場合があります。また、lgroup に葉 lgroup が含まれ、その葉 lgroup に自身のリソースセットが含まれる場合もあります。葉 lgroup のリソースは、その葉 lgroup をカプセル化する lgroup から利用可能です。

lgrp_cookie_stale() の使用法

lgrp_cookie_stale(3LGRP) 関数は、指定の cookie で表された lgroup 階層のスナップショットが最新のものであるかどうかを判定します。

```
#include <sys/lgrp_user.h>
int lgrp_cookie_stale(lgrp_cookie_t cookie);
```

lgrp_init() 関数が返す cookie は、そのスナップショットを取得した view 引数の種類によって、さまざまな理由により無効になる場合があります。view 引数に LGRP_VIEW_OS を指定して呼び出した lgrp_init() 関数が返す cookie では、動的再構成や CPU のオンラインステータスの変化などが原因で lgroup 階層に変更があったときに、無効になる場合があります。view 引数に LGRP_VIEW_CALLER を指定した lgrp_init() 関数が返す cookie では、呼び出しスレッドのプロセッサセットの変更または lgroup 階層の変化が原因で、無効になる場合があります。無効になった cookie は、その cookie で lgrp_fini() 関数を呼び出し、次に新規 cookie を生成する lgrp_init() 関数を呼び出すことによってリフレッシュされます。

無効な cookie を指定した場合、lgrp_cookie_stale() 関数は EINVAL を返します。

lgrp_view() の使用法

lgrp_view(3LGRP) 関数は、指定した lgroup 階層のスナップショットがどのビューで取得されたかを判別します。

```
#include <sys/lgrp_user.h>
lgrp_view_t lgrp_view(lgrp_cookie_t cookie);
```

lgrp_view() 関数は、lgroup 階層のスナップショットを表す cookie を引数に使用し、そのスナップショットの view を返します。LGRP_VIEW_CALLER を view に指定して取得したスナップショットには、呼び出しスレッドで使用可能なリソースのみが含まれます。LGRP_VIEW_OS で取得したスナップショットには、オペレーティングシステムで使用可能なすべてのリソースが含まれます。

無効な cookie を指定した場合、lgrp_view() 関数は EINVAL を返します。

lgrp_nlgrps() の使用法

lgrp_nlgrps(3LGRP) 関数は、システムに存在する近傍性グループの数を返します。近傍性グループが 1 つしかないシステムでは、メモリー配置の最適化による効果はありません。

```
#include <sys/lgrp_user.h>
int lgrp_nlgrps(lgrp_cookie_t cookie);
```

lgrp_nlgrps() 関数は、lgroup 階層のスナップショットを表す cookie を引数に使用し、階層で使用可能な lgroup の数を返します。

無効な cookie を指定した場合、lgrp_nlgrps() 関数は EINVAL を返します。

lgrp_root() の使用法

lgrp_root(3LGRP) 関数は、ルート lgroup ID を返します。

```
#include <sys/lgrp_user.h>
lgrp_id_t lgrp_root(lgrp_cookie_t cookie);
```

lgrp_root() 関数は、lgroup 階層のスナップショットを表す cookie を引数に使用し、ルート lgroup ID を返します。

lgrp_parents() の使用法

lgrp_parents(3LGRP) 関数は、lgroup 階層のスナップショットを表す cookie を引数に使用し、指定した lgroup の親 lgroups の数を返します。

```
#include <sys/lgrp_user.h>
int lgrp_parents(lgrp_cookie_t cookie, lgrp_id_t child,
                 lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

lgrp_array が NULL ではなく、lgrp_array_size の値がゼロでないとき、lgrp_parents() 関数は、配列の要素数の上限まで、またはすべての親 lgroup ID を配列に入れて返します。ルート lgroup には親はありません。ルート lgroup に対して lgrp_parents() 関数が呼び出された場合は、lgrp_array には何も返されません。

無効な cookie を指定した場合、lgrp_parents() 関数は EINVAL を返します。指定した lgroup ID が存在しない場合は、lgrp_parents() 関数は ESRCH を返します。

lgrp_children() の使用法

lgrp_children(3LGRP) 関数は、呼び出しスレッドの lgroup 階層のスナップショットを表す cookie を引数に使用し、指定した lgroup の子 lgroup の数を返します。

```
#include <sys/lgrp_user.h>
int lgrp_children(lgrp_cookie_t cookie, lgrp_id_t parent,
                  lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

lgrp_array が NULL ではなく、lgrp_array_size の値がゼロでないとき、lgrp_children() 関数は、要素数の上限まで子 lgroup ID を配列に入れるか、またはすべての子 lgroup ID を配列に入れて返します。

無効な cookie を指定した場合、lgrp_children() 関数は EINVAL を返します。指定した lgroup ID が存在しない場合は、lgrp_children() 関数は ESRCH を返します。

近傍性グループの内容

次に示す API では、指定した lgroup の内容に関する情報を取り出します。

lgroup 階層は、ドメインのリソースを組織化し、もっとも近いリソースを検索するプロセスを簡素化します。葉 lgroup は、もっとも応答時間の短いリソースで定義されます。葉 lgroup の上位ノードの各 lgroup には、その子 lgroup にもっとも近いリソースが含まれます。ルート lgroup には、そのドメインにあるすべてのリソースが含まれます。

lgroup のリソースは、lgroup の直下に含まれるか、またはその lgroup にカプセル化された葉 lgroup 内に含まれます。葉 lgroup は、直下にリソースを含み、ほかの lgroup をカプセル化することはありません。

lgrp_resources() の使用法

lgrp_resources() 関数は、指定した lgroup に含まれるリソースの数を返します。

```
#include <sys/lgrp_user.h>
int lgrp_resources(lgrp_cookie_t cookie, lgrp_id_t lgrp, lgrp_id_t *lgrpids,
                   uint_t count, lgrp_rsrc_t type);
```

lgrp_resources() 関数は、lgroup 階層のスナップショットを表す cookie を引数に使用します。その cookie は lgrp_init () 関数から取得されます。lgrp_resources() 関数は、lgrp 引数の値で指定した ID を持つ lgroup にあるリソースの数を返します。lgrp_resources() 関数は、CPU またはメモリーのリソースを直下に含む lgroup のセットを持つリソースを表します。lgrp_rsrc_t 引数には、次の 2 つの値が指定できます。

LGRP_RSRC_CPU　lgrp_resources() 関数は CPU リソースの数を返します。

LGRP_RSRC_MEM　lgrp_resources() 関数はメモリーリソースの数を返します。

lgrpids[] 引数として渡された値が NULL でなく、かつ count 引数に渡された値がゼロでない場合、lgrp_resources() 関数は lgroup ID を lgrpids[] 配列に格納します。lgrpids[] 配列に格納できる lgroup ID の最大数は count 引数の値です。

指定した cookie、lgroup ID、またはタイプが無効な場合、lgrp_resources() 関数は EINVAL を返します。指定した lgroup ID が見つからない場合、lgrp_resources () 関数は ESRCH を返します。

lgrp_cpus() の使用法

lgrp_cpus(3LGRP) 関数は、lgroup 階層のスナップショットを表す cookie を引数に使用し、指定した lgroup にある CPU の数を返します。

```
#include <sys/lgrp_user.h>
int lgrp_cpus(lgrp_cookie_t cookie, lgrp_id_t lgrp, processorid_t *cpuids,
              uint_t count, int content);
```

cpuid[] 引数が NULL ではなく、CPU 数がゼロでないとき、lgrp_cpus() 関数は、配列の要素数の上限まで、またはすべての CPU ID を配列に入れて返します。

content 引数には、次の 2 つの値が指定できます。

LGRP_CONTENT_ALL　lgrp_cpus() 関数は、この lgroup と下位ノードにある CPU の ID を返します。

LGRP_CONTENT_DIRECT　lgrp_cpus() 関数は、この lgroup にある CPU の ID だけを返します。

指定した cookie、lgroup ID、またはフラグのいずれか 1 つが無効な場合、lgrp_cpus() 関数は EINVAL を返します。指定した lgroup ID が存在しない場合は、lgrp_cpus() 関数は ESRCH を返します。

lgrp_mem_size() の使用法

lgrp_mem_size(3LGRP) 関数は、lgroup 階層のスナップショットを表す cookie を引数に使用し、指定した lgroup にインストールされたメモリー、または空きメモリー領域のサイズを返します。lgrp_mem_size() 関数は、メモリーサイズをバイト数で返します。

```
#include <sys/lgrp_user.h>
lgrp_mem_size_t lgrp_mem_size(lgrp_cookie_t cookie, lgrp_id_t lgrp,
                              int type, int content)
```

type 引数には、次の値が指定できます。

LGRP_MEM_SZ_FREE　lgrp_mem_size() 関数は、空きメモリー領域のサイズをバイト数で返します。

LGRP_MEM_SZ_INSTALLED　lgrp_mem_size() 関数は、インストールされたメモリーのサイズをバイト数で返します。

content 引数には、次の 2 つの値が指定できます。

LGRP_CONTENT_ALL　lgrp_mem_size() 関数は、この lgroup と下位ノードのメモリーサイズの総量を返します。

LGRP_CONTENT_DIRECT　lgrp_mem_size() 関数は、この lgroup のメモリーのサイズのみを返します。

指定した cookie、lgroup ID、またはフラグのいずれか 1 つが無効な場合、lgrp_mem_size() 関数は EINVAL を返します。指定した lgroup ID が存在しない場合は、lgrp_mem_size() 関数は ESRCH を返します。

近傍性グループの特性

次に示す API では、指定した lgroup の特性に関する情報を取り出します。

lgrp_latency_cookie() の使用法

lgrp_latency(3LGRP) 関数は、ある lgroup の CPU が別の lgroup のメモリーにアクセスするときの応答時間を返します。

```
#include <sys/lgrp_user.h>
int lgrp_latency_cookie(lgrp_cookie_t cookie, lgrp_id_t from, lgrp_id_t to.
                        lat_between_t between);
```

lgrp_latency_cookie() 関数は、lgroup 階層のスナップショットを表す cookie を引数に使用します。lgrp_init() 関数がこの cookie を作成します。lgrp_latency_cookie() 関数は、「*from*」引数の値で指定された lgroup にあるハードウェアリソースと「*to*」引数の値で指定された lgroup にあるハードウェアリソースの間の応答時間を表す値を返します。2 つの引数が同じ lgroup を指している場合 lgrp_latency_cookie() 関数は、同一 lgroup での応答時間値を返します。

注 – lgrp_latency_cookie() 関数が返す応答時間値は、オペレーティングシステムで定義されたもので、プラットフォーム固有の数値です。この値は、実際のハードウェアデバイスの応答時間を表しているとは限りません。あるドメイン内部での比較のためにのみ使用してください。

「*between*」引数の値が LGRP_LAT_CPU_TO_MEM である場合、lgrp_latency_cookie() 関数は CPU リソースからメモリーリソースまでの応答時間を測定します。

無効な lgroup ID を指定した場合、lgrp_latency_cookie() 関数は EINVAL を返します。lgrp_latency_cookie() 関数で指定された lgroup ID が見つからないとき、「from」引数で指定された lgroup に CPU が存在しないとき、または「to」引数で指定された lgroup にメモリーが存在しないときには、lgrp_latency_cookie() 関数は ESRCH を返します。

近傍性グループ、スレッド、およびメモリー配置

このセクションでは、各 lgroup のスレッドとメモリー配置を検出し、制御するために使用する API について説明します。

- lgrp_home(3LGRP) 関数は、スレッドの配置を検出するために使用します。
- メモリー配置の検出には、meminfo(2) システムコールを使用します。
- lgroup 間でのメモリー配置を制御するには、madvise(3C) 関数に MADV_ACCESS フラグを設定して使用します。
- lgrp_affinity_set(3LGRP) 関数では、指定した lgroup のスレッドのアフィニティーを設定することにより、スレッドとメモリー配置を制御できます。
- ある lgroup のアフィニティーが、リソースをどの lgroup から割り当てるか優先順位を決定するのに影響を与える可能性があります。
- カーネルが効率的なメモリーの割り当てを行うには、アプリケーションのメモリー使用パターンについての情報が必要になります。
- madvise() 関数およびその共有オブジェクト版である madv.so.1 により、この情報をカーネルに提供します。

- 実行中のプロセスが meminfo() システムコールを使用して自分自身のメモリー使用に関する情報を収集できます。

lgrp_home() の使用法

lgrp_home() 関数は、指定したプロセスまたはスレッドのホーム lgroup を返します。

```
#include <sys/lgrp_user.h>
lgrp_id_t lgrp_home(idtype_t idtype, id_t id);
```

無効な ID タイプを指定した場合、lgrp_home() 関数は EINVAL を返します。呼び出しプロセスの実効ユーザーがスーパーユーザーではなく、実ユーザー ID または実効ユーザー ID が指定したスレッドの実ユーザー ID または実効ユーザー ID とも一致しない場合、lgrp_home() 関数は EPERM を返します。指定したプロセスまたはスレッドが存在しない場合は、lgrp_home() 関数は ESRCH を返します。

madvise() の使用法

madvise() 関数は、ユーザーの仮想メモリー領域の *addr* で指定された開始アドレスから *len* パラメータの値で示される長さの範囲について特定の使用パターンに従うように、カーネルに対してアドバイス情報を与えます。カーネルはこの情報を使用して、指定された範囲に関連付けられたリソースの操作と管理の手順を最適化します。メモリーに対するアクセスパターンに関する適切な情報を持つプログラムで madvise() 関数を使用できれば、システム性能を向上させることができます。

```
#include <sys/types.h>
#include <sys/mman.h>
int madvise(caddr_t addr, size_t len, int advice);
```

madvise() 関数では、複数 lgroup に対するスレッドのメモリーの割り当て方法を操作するために、次のフラグが提供されています。

MADV_ACCESS_DEFAULT	このフラグは、指定範囲に関して期待されるカーネルのアクセスパターンを取り消してデフォルトに戻します。
MADV_ACCESS_LWP	このフラグは、指定のアドレス範囲に次回アクセスする LWP が、その領域にもっとも頻繁にアクセスする LPW であることをカーネルに指定します。それに応じて、カーネルはメモリーやほかのリソースをこの領域と LWP に割り当てます。
MADV_ACCESS_MANY	このフラグは、多くのプロセスまたは LPW が、ランダムにシステム全域から指定の領域にアクセスしていることをカーネルに指定します。それに応じて、カーネルはメモリーやほかのリソースをこの領域に割り当てます。

madvise() 関数は、次の値を返します。

EAGAIN　*addr* から *addr+len* までのアドレス範囲で指定されたマッピング領域の全体または一部が、入出力操作によりロックされている場合。

EINVAL　*addr* パラメータの値が sysconf(3C) で返されるページサイズの倍数ではない、指定したアドレス範囲の長さがゼロ以下、またはアドバイスが無効の場合。

EIO　ファイルシステムに対する読み取りや書き込み中に入出力エラーが発生した場合。

ENOMEM　指定したアドレス範囲のアドレスが、プロセスの有効なアドレス空間の範囲外、またはマップされていないページが指定されている場合。

ESTALE　NFS ファイルハンドルが無効の場合。

madv.so.1 の使用法

共有オブジェクト madv.so.1 は、起動されたプロセスやその子プロセスに対して選択された仮想メモリーの構成を実現します。共有オブジェクトを使用するには、環境変数に次の文字列を指定する必要があります。

```
LD_PRELOAD=$LD_PRELOAD:madv.so.1
```

madv.so.1 共有オブジェクトは、 MADV 環境変数の値に従ってメモリーのアドバイス情報を適用します。MADV 環境変数は、プロセスのアドレス空間におけるすべてのヒープ、共有メモリー、および mmap 領域のために使用する仮想メモリーのアドバイス情報を指定します。この情報は生成されたすべてのプロセスに適用されます。次に示す MADV 環境変数値は、複数の lgroup 間でのリソースの割り当てに影響を与えます。

access_default　この値は、カーネルに期待されるアクセスパターンをデフォルトに戻します。

access_lwp　この値は、アドレス範囲に次回アクセスする LWP が、その領域にもっとも頻繁にアクセスする LPW であることをカーネルに指定します。それに応じて、カーネルはメモリーやほかのリソースをこの領域と LWP に割り当てます。

access_many　この値は、多くのプロセスまたは LPW が、ランダムにシステム全域からメモリーにアクセスしていることをカーネルに指定します。それに応じて、カーネルはメモリーやほかのリソースを割り当てます。

MADVCFGFILE 環境変数の値は、1 つまたは複数のメモリーのアドバイス情報構成エントリが *exec-name*: *advice-opts* の書式で記述されているテキストファイルの名前です。

exec-name の値は、アプリケーションまたは実行プログラムの名前です。*exec-name* には、フルパス名、基本名、またはパターン文字列による指定が可能です。

advice-opts の値は、*region=advice* の書式で表します。*advice* の値は、MADV 環境変数の値と同じです。*region* には、次のいずれかの規定された値を指定します。

madv	プロセスのアドレス空間のすべてのヒープ、共有メモリー、および mmap(2) 領域に、アドバイスが適用されます。
heap	ヒープは、brk(2) 領域として定義されます。アドバイス情報は、既存のヒープにも将来割り当てられる追加ヒープメモリーにも適用されます。
shm	アドバイス情報は、共有メモリーセグメントに適用されます。共有メモリー操作に関する詳細は、shmat(2) を参照してください。
ism	アドバイス情報は、SHM_SHARE_MMU フラグを使用している共有メモリーセグメントに適用されます。ism オプションは、shm より優先されます。
dsm	アドバイス情報は、SHM_PAGEABLE フラグを使用している共有メモリーセグメントに適用されます。dsm オプションは、shm より優先されます。
mapshared	アドバイス情報は、MAP_SHARED フラグを使用した mmap() システムコールにより作成されたマッピングに適用されます。
mapprivate	アドバイス情報は、MAP_PRIVATE フラグを使用した mmap() システムコールにより作成されたマッピングに適用されます。
mapanon	アドバイス情報は、MAP_ANON フラグを使用した mmap() システムコールにより作成されたマッピングに適用されます。複数のオプションが指定された場合は、mapanon オプションが優先されます。

MADVERRFILE 環境変数値は、エラーメッセージが記録されるパスの名前です。MADVERRFILE による指定がない場合は、madv.so.1 共有オブジェクトは syslog(3C) を使用してエラーを記録します。重要度は LOG_ERR、機能記述子は LOG_USER になります。

メモリーに関するアドバイス情報は継承されます。子プロセスには親と同じアドバイスが適用されます。madv.so.1 共有オブジェクトにより異なるレベルのアドバイスを構成しない限り、exec(2) が呼び出されたあとには、アドバイスはシステムデフォルトの設定に戻されます。アドバイスは、ユーザープログラムによって作成された mmap() 領域にのみ適用されます。実行時リンカーまたはシステムライブラリによって作成された直接システムコールを呼び出す領域は影響を受けません。

madv.so.1 の使用例

次の例では、madv.so.1 共有オブジェクトの機能について個別に説明します。

例 5–2 アプリケーションセットへのアドバイスの設定

この構成では、exec が foo で始まるすべてのアプリケーションの ISM セグメントにアドバイス情報を適用しています。

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        foo*:ism=access_lwp
```

例 5–3 特定のアプリケーションセットに対するアドバイスの除外

この構成では、ls を除くすべてのアプリケーションにアドバイスを適用しています。

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADV=access_many
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADV MADVCFGFILE
$ cat $MADVCFGFILE
        ls:
```

例 5–4 構成ファイルでのパターンマッチの使用

MADVCFGFILE に指定された構成は MADV の設定値より優先されるので、ファイルの最後の構成エントリで *exec-name* に指定した * は、MADV を指定した場合と同じ意味になります。この例は、前述の例と同じ結果になります。

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        ls:
        *:madv=access_many
```

例 5–5 複数の領域に対するアドバイス

この構成では、あるアドバイスのタイプを mmap() 領域に適用し、さらに別のアドバイスのタイプを exec() の名前が foo で始まるアプリケーションのヒープおよび共有メモリー領域に対して適用しています。

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        foo*:madv=access_many,heap=sequential,shm=access_lwp
```

`meminfo()` の使用法

`meminfo()` 関数は、システムにより割り当てられた仮想メモリーおよび物理メモリーに関する情報を、呼び出しプロセスに提供します。

```
#include <sys/types.h>
#include <sys/mman.h>
int meminfo(const uint64_t inaddr[], int addr_count,
    const uint_t info_req[], int info_count, uint64_t outdata[],
    uint_t validity[]);
```

`meminfo()` 関数は、次に示す情報を返します。

`MEMINFO_VPHYSICAL`	指定した仮想アドレスに対応する物理メモリーのアドレス
`MEMINFO_VLGRP`	指定した仮想アドレスに対応する物理ページがある lgroup
`MEMINFO_VPAGESIZE`	指定した仮想アドレスに対応する物理ページのサイズ
`MEMINFO_VREPLCNT`	指定した仮想アドレスに対応する物理ページの複製の数
`MEMINFO_VREPL|n`	指定した仮想アドレスの *n* 番目の物理ページの複製
`MEMINFO_VREPL_LGRP|n`	指定した仮想アドレスの *n* 番目の物理ページの複製がある lgroup
`MEMINFO_PLGRP`	指定した物理アドレスがある lgroup

`meminfo()` 関数には、次のパラメータを指定できます。

inaddr	入力アドレスの配列。
addr_count	`meminfo()` 関数に渡されるアドレスの数。
info_req	要求される情報のタイプをリストする配列。
info_count	*inaddr* 配列にある各アドレスについて要求された情報の数。
outdata	`meminfo()` 関数が結果を返す配列。配列のサイズは、*info_req* と *addr_count* パラメータに指定した値の積になります。
validity	*addr_count* パラメータの値と同じサイズの配列。*validity* 配列にはビット単位の結果コードが返されます。結果コードの *0* 番目のビットでは、対応する入力アドレスの有効性が評価されています。続くビットでは、*info_req* 配列のメンバーへの応答の有効性が順番に評価されています。

outdata または *validity* 配列が指しているメモリー領域に書き込めない場合、`meminfo()` 関数は `EFAULT` を返します。*info_req* または *inaddr* 配列が指しているメモリー領域から読み込めない場合は、`meminfo()` 関数は `EFAULT` を返しま

す。*info_count* の値が 31 より大きいか、または 1 より小さい場合は、`meminfo()` 関数は `EINVAL` を返します。`addr_count()` の値がゼロより小さい場合は、`meminfo` 関数は *EINVAL* を返します。

例 5–6 仮想アドレスのセットに対応する物理ページとページサイズを出力する `meminfo()` の使用法

```
void
print_info(void **addrvec, int how_many)
{
        static const int info[] = {
                MEMINFO_VPHYSICAL,
                MEMINFO_VPAGESIZE};
        uint64_t * inaddr = alloca(sizeof(uint64_t) * how_many);
        uint64_t * outdata = alloca(sizeof(uint64_t) * how_many * 2;
        uint_t * validity = alloca(sizeof(uint_t) * how_many);

        int i;

        for (i = 0; i < how_many; i++)
                inaddr[i] = (uint64_t *)addr[i];

        if (meminfo(inaddr, how_many,  info,
                    sizeof (info)/ sizeof(info[0]),
                    outdata, validity) < 0)
                ...

        for (i = 0; i < how_many; i++) {
                if (validity[i] & 1 == 0)
                        printf("address 0x%llx not part of address
                                        space\n",
                                inaddr[i]);

                else if (validity[i] & 2 == 0)
                        printf("address 0x%llx has no physical page
                                        associated with it\n",
                                inaddr[i]);

                else {
                        char buff[80];
                        if (validity[i] & 4 == 0)
                                strcpy(buff, "<Unknown>");
                        else
                                sprintf(buff, "%lld", outdata[i * 2 +
                                                1]);
                        printf("address 0x%llx is backed by physical
                                        page 0x%llx of size %s\n",
                                        inaddr[i], outdata[i * 2], buff);
                }
        }
}
```

近傍性グループのアフィニティー

スレッドの軽量プロセス (LWP) が作成されるとき、カーネルはスレッドに近傍性グループを割り当てます。そのような lgroup は、スレッドのホーム *lgroup* と呼ばれます。カーネルは、スレッドをホーム lgroup の CPU で実行し、可能な限り lgroup からメモリーを割り当てます。ホーム lgroup のリソースが使用可能でない場合は、ほかの lgroup からリソースを割り当てます。スレッドに 1 つまたは複数の lgroup に対するアフィニティーがある場合は、オペレーティングシステムはアフィニティーの強さに応じて lgroup から順にリソースを割り当てます。lgroup は、次の 3 つのアフィニティーレベルのうち 1 つを持つことができます。

1. `LGRP_AFF_STRONG` は強いアフィニティーを示します。この lgroup がスレッドのホーム lgroup であるとき、オペレーティングシステムは、そのスレッドのホーム lgroup を変更する再ホーミングをできるだけ避けようとします。その場合でも、動的再構成、プロセッサのオフライン化、プロセッサ割り当て、プロセッサセットの割り当ておよび操作などのイベントは、スレッドの再ホーミングにつながる可能性があります。
2. `LGRP_AFF_WEAK` は、弱いアフィニティーを示します。この lgroup がスレッドのホーム lgroup であるとき、オペレーティングシステムは、負荷均衡の必要に応じてスレッドの再ホーミングを行います。
3. `LGRP_AFF_NONE` はアフィニティーを持たないことを示します。スレッドがどの lgroup にもアフィニティーを持たないとき、オペレーティングシステムはそのスレッドにホーム lgroup を割り当てます。

指定されたスレッドにリソースを割り当てるときに、オペレーティングシステムは lgroup のアフィニティーをアドバイスとして使用します。このアドバイスはほかのシステム制限とともに考慮されます。プロセッサ割り当ておよびプロセッサセットによって lgroup のアフィニティーが影響を受けることはありませんが、スレッドが実行される lgroup を制限する場合があります。

`lgrp_affinity_get()` の使用法

`lgrp_affinity_get(3LGRP)` 関数は、指定した lgroup に対して LWP が持つアフィニティーを返します。

```
#include <sys/lgrp_user.h>
lgrp_affinity_t lgrp_affinity_get(idtype_t idtype, id_t id, lgrp_id_t lgrp);
```

idtype および *id* 引数により、`lgrp_affinity_get()` 関数で検証する LWP を指定します。*idtype* の値に `P_PID` を指定した場合、`lgrp_affinity_get()` 関数は、*id* 引数の値と一致するプロセス ID を持つプロセスにある LWP のいずれかについて lgroup アフィニティーを取得します。*idtype* に `P_LWPID` を指定した場合、`lgrp_affinity_get()` 関数は、実行中のプロセスにある、*id* 引数の値と一致する LWP ID を持つ LWP につ

いてlgroupアフィニティーを取得します。*idtype*にP_MYIDを指定した場合、lgrp_affinity_get()関数は、実行中のLPWについてlgroupアフィニティーを取得します。

指定したlgroupまたはIDタイプが有効でないとき、lgrp_affinity_get()関数はEINVALを返します。呼び出しプロセスの実効ユーザーがスーパーユーザーではなく、呼び出しプロセスのIDがどのLPWの実ユーザーIDまたは実効ユーザーIDとも一致しない場合、lgrp_affinity_get()関数はEPERMを返します。指定したlgroupまたはLPWが存在しない場合は、lgrp_affinity_get()関数はESRCHを返します。

lgrp_affinity_set()の使用法

lgrp_affinity_set(3LGRP)関数は、指定したlgroupに対してLWPまたはLWPのセットが持つアフィニティーを設定します。

```
#include <sys/lgrp_user.h>
int lgrp_affinity_set(idtype_t idtype, id_t id, lgrp_id_t lgrp,
                      lgrp_affinity_t affinity);
```

*idtype*および*id*引数により、lgrp_affinity_set()関数で検証するLWPまたはLWPのセットを指定します。*idtype*にP_PIDを指定した場合、lgrp_affinity_set()関数は、*id*引数の値が一致するプロセスIDを持つプロセスのすべてのLWPについて、lgroupアフィニティーを*affinity*引数で指定したアフィニティーレベルに設定します。*idtype*にP_LWPIDを指定した場合、lgrp_affinity_set()関数は、*id*引数の値が一致するLWP IDを持つ実行プロセス中のLWPについて、lgroupアフィニティーを*affinity*引数で指定したアフィニティーレベルに設定します。*idtype*にP_MYIDを指定した場合は、lgrp_affinity_set()関数は、実行中のLWPまたはプロセスについて、lgroupアフィニティーを*affinity*引数で指定したアフィニティーレベルに設定します。

指定したlgroup、アフィニティー、またはIDタイプが有効でないとき、lgrp_affinity_set()関数はEINVALを返します。呼び出しプロセスの実効ユーザーがスーパーユーザーではなく、呼び出しプロセスのIDがどのLPWの実ユーザーIDまたは実効ユーザーIDとも一致しない場合、lgrp_affinity_set()関数はEPERMを返します。指定したlgroupまたはLPWが存在しない場合は、lgrp_affinity_set()関数はESRCHを返します。

APIの使用例

このセクションでは、この章で説明したAPIを使用するタスクのコーディング例を示します。

例5–7 スレッドへのメモリーの移動

次のコーディング例では、*addr* と *addr+len* のアドレス範囲にあるメモリーを、その範囲に次回アクセスするスレッド付近に移動します。

```
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>

/*
 * Move memory to thread
 */
void
mem_to_thread(caddr_t addr, size_t len)
{
    if (madvise(addr, len, MADV_ACCESS_LWP) < 0)
        perror("madvise");
}
```

例5–8 メモリーへのスレッドの移動

このコーディング例では、`meminfo()` 関数を使用して、指定されたアドレスで仮想ページにバックアップする物理メモリーの lgroup を判定します。次にこの例では、現在のスレッドをそのメモリー付近に移動するために、その lgroup に強いアフィニティーを設定します。

```
#include <stdio.h>
#include <sys/lgrp_user.h>
#include <sys/mman.h>
#include <sys/types.h>

/*
 * Move a thread to memory
 */
int
thread_to_memory(caddr_t va)
{
    uint64_t    addr;
    ulong_t     count;
    lgrp_id_t   home;
    uint64_t    lgrp;
    uint_t      request;
    uint_t      valid;

    addr = (uint64_t)va;
    count = 1;
    request = MEMINFO_VLGRP;
    if (meminfo(&addr, 1, &request, 1, &lgrp, &valid) != 0) {
        perror("meminfo");
        return (1);
    }

    if (lgrp_affinity_set(P_LWPID, P_MYID, lgrp, LGRP_AFF_STRONG) != 0) {
        perror("lgrp_affinity_set");
        return (2);
```

例 5–8 メモリーへのスレッドの移動 (続き)

```
    }

    home = lgrp_home(P_LWPID, P_MYID);
    if (home == -1) {
        perror ("lgrp_home");
        return (3);
    }

    if (home != lgrp)
        return (-1);

    return (0);
}
```

例 5–9 lgroup階層の巡回

次のコーディング例では、lgroup階層を巡回し、出力します。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/lgrp_user.h>
#include <sys/types.h>

/*
 * Walk and print lgroup hierarchy from given lgroup
 * through all its descendants
 */
int
lgrp_walk(lgrp_cookie_t cookie, lgrp_id_t lgrp, lgrp_content_t content)
{
    lgrp_affinity_t    aff;
    lgrp_id_t          *children;
    processorid_t      *cpuids;
    int                i;
    int                ncpus;
    int                nchildren;
    int                nparents;
    lgrp_id_t          *parents;
    lgrp_mem_size_t    size;

    /*
     * Print given lgroup, caller's affinity for lgroup,
     * and desired content specified
     */
    printf("LGROUP #%d:\n", lgrp);

    aff = lgrp_affinity_get(P_LWPID, P_MYID, lgrp);
    if (aff == -1)
        perror ("lgrp_affinity_get");
    printf("\tAFFINITY: %d\n", aff);

    printf("CONTENT %d:\n", content);

    /*
```

例 5–9 lgroup階層の巡回 *(続き)*

```
 * Get CPUs
 */
ncpus = lgrp_cpus(cookie, lgrp, NULL, 0, content);
printf("\t%d CPUS: ", ncpus);
if (ncpus == -1) {
    perror("lgrp_cpus");
    return (-1);
} else if (ncpus > 0) {
    cpuids = malloc(ncpus * sizeof (processorid_t));
    ncpus = lgrp_cpus(cookie, lgrp, cpuids, ncpus, content);
            if (ncpus == -1) {
        free(cpuids);
                        perror("lgrp_cpus");
        return (-1);
    }
    for (i = 0; i < ncpus; i++)
        printf("%d ", cpuids[i]);
    free(cpuids);
}
printf("\n");

/*
 * Get memory size
 */
printf("\tMEMORY: ");
size = lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_INSTALLED, content);
if (size == -1) {
    perror("lgrp_mem_size");
    return (-1);
}
printf("installed bytes 0x%llx, ", size);
size = lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_FREE, content);
    if (size == -1) {
    perror("lgrp_mem_size");
    return (-1);
}
printf("free bytes 0x%llx\n", size);

/*
 * Get parents
 */
nparents = lgrp_parents(cookie, lgrp, NULL, 0);
printf("\t%d PARENTS: ", nparents);
if (nparents == -1) {
    perror("lgrp_parents");
    return (-1);
} else if (nparents > 0) {
    parents = malloc(nparents * sizeof (lgrp_id_t));
    nparents = lgrp_parents(cookie, lgrp, parents, nparents);
                if (nparents == -1) {
        free(parents);
                    perror("lgrp_parents");
        return (-1);
                }
    for (i = 0; i < nparents; i++)
```

例 5–9 lgroup階層の巡回 *(続き)*

```
            printf("%d ", parents[i]);
        free(parents);
    }
    printf("\n");

    /*
     * Get children
     */
    nchildren = lgrp_children(cookie, lgrp, NULL, 0);
    printf("\t%d CHILDREN: ", nchildren);
    if (nchildren == -1) {
        perror("lgrp_children");
        return (-1);
    } else if (nchildren > 0) {
        children = malloc(nchildren * sizeof (lgrp_id_t));
        nchildren = lgrp_children(cookie, lgrp, children, nchildren);
                  if (nchildren == -1) {
            free(children);
                        perror("lgrp_children");
            return (-1);
                  }
        printf("Children: ");
        for (i = 0; i < nchildren; i++)
            printf("%d ", children[i]);
        printf("\n");

        for (i = 0; i < nchildren; i++)
            lgrp_walk(cookie, children[i], content);

        free(children);
    }
    printf("\n");

    return (0);
}
```

例 5–10 指定されたlgroup以外で利用可能なメモリーを持つもっとも近いlgroupの検索

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/lgrp_user.h>
#include <sys/types.h>

#define    INT_MAX    2147483647

/*
 * Find next closest lgroup outside given one with available memory
 */
lgrp_id_t
lgrp_next_nearest(lgrp_cookie_t cookie, lgrp_id_t from)
{
    lgrp_id_t          closest;
    int                i;
    int                latency;
    int                lowest;
```

例 5–10 指定されたlgroup以外で利用可能なメモリーを持つもっとも近いlgroupの検索 (続き)

```
int                nparents;
lgrp_id_t          *parents;
lgrp_mem_size_t    size;

/*
 * Get number of parents
 */
nparents = lgrp_parents(cookie, from, NULL, 0);
if (nparents == -1) {
    perror("lgrp_parents");
    return (LGRP_NONE);
}

/*
 * No parents, so current lgroup is next nearest
 */
if (nparents == 0) {
    return (from);
}

/*
 * Get parents
 */
parents = malloc(nparents * sizeof (lgrp_id_t));
nparents = lgrp_parents(cookie, from, parents, nparents);
if (nparents == -1) {
    perror("lgrp_parents");
    free(parents);
    return (LGRP_NONE);
    }

/*
 * Find closest parent (ie. the one with lowest latency)
 */
closest = LGRP_NONE;
lowest = INT_MAX;
for (i = 0; i < nparents; i++) {
    lgrp_id_t    lgrp;

    /*
     * See whether parent has any free memory
     */
    size = lgrp_mem_size(cookie, parents[i], LGRP_MEM_SZ_FREE,
        LGRP_CONTENT_ALL);
    if (size > 0)
        lgrp = parents[i];
    else {
        if (size == -1)
            perror("lgrp_mem_size");

        /*
         * Find nearest ancestor if parent doesn't
         * have any memory
         */
```

例 5–10 指定されたlgroup以外で利用可能なメモリーを持つもっとも近いlgroupの検索 (続き)

```
            lgrp = lgrp_next_nearest(cookie, parents[i]);
            if (lgrp == LGRP_NONE)
                continue;
        }

        /*
         * Get latency within parent lgroup
         */
        latency = lgrp_latency_cookie(lgrp, lgrp);
        if (latency == -1) {
            perror("lgrp_latency_cookie");
            continue;
        }

        /*
         * Remember lgroup with lowest latency
         */
        if (latency < lowest) {
            closest = lgrp;
            lowest = latency;
        }
    }

    free(parents);
    return (closest);
}

/*
 * Find lgroup with memory nearest home lgroup of current thread
 */
lgrp_id_t
lgrp_nearest(lgrp_cookie_t cookie)
{
    lgrp_id_t    home;
    longlong_t    size;

    /*
     * Get home lgroup
     */
    home = lgrp_home(P_LWPID, P_MYID);

    /*
     * See whether home lgroup has any memory available in its hierarchy
     */
    size = lgrp_mem_size(cookie, home, LGRP_MEM_SZ_FREE,
        LGRP_CONTENT_ALL);
    if (size == -1)
        perror("lgrp_mem_size");

    /*
     * It does, so return the home lgroup.
     */
    if (size > 0)
        return (home);
```

例 5–10 指定されたlgroup以外で利用可能なメモリーを持つもっとも近いlgroupの検索 (続き)

```
    /*
     * Otherwise, find next nearest lgroup outside of the home.
     */
    return (lgrp_next_nearest(cookie, home));
}
```

例 5–11 空きメモリー領域を持つもっとも近いlgroupの検索

このコーディング例では、指定されたスレッドのホームlgroupにもっとも近い、空きメモリー領域を持つlgroupを検索します。

```
lgrp_id_t
lgrp_nearest(lgrp_cookie_t cookie)
{
        lgrp_id_t         home;
        longlong_t        size;

        /*
         * Get home lgroup
         */

        home = lgrp_home();

        /*
         * See whether home lgroup has any memory available in its hierarchy
         */

        if (lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_FREE,
            LGRP_CONTENT_ALL, &size) == -1)
                perror("lgrp_mem_size");

        /*
         * It does, so return the home lgroup.
         */

        if (size > 0)
                return (home);

        /*
         * Otherwise, find next nearest lgroup outside of the home.
         */

        return (lgrp_next_nearest(cookie, home));
}
```

◆ ◆ ◆ 第 6 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。114 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をロックする旧来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

一連のデータが編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。

通常ファイルのコンポーネントは次のとおりです。

- 「*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ます。
- 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、112 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、15 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もっと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 6–1 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きます
close(2)	ファイル記述子を閉じます
read(2)	ファイルから読み取ります
write(2)	ファイルに書き込みます
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します
unlink(2)	ディレクトリエントリを削除します
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します

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 6–1 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あとは必ず、そのファイルに対して close(2) を呼び出してください。close(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 open(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;

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

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

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

高度なファイル入出力

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 6–2 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)	ファイルにリンクします
access(2)	ファイルのアクセス可能性を判断します
mknod(2)	特殊ファイルまたは通常のファイルを作成します
chmod(2)	ファイルのモードを変更します
chown(2)、lchown(2)、fchown(2)	ファイルの所有者とグループを変更します
utime(2)	ファイルのアクセス時間や変更時間を設定します
stat(2)、lstat(2)、fstat(2)	ファイルのステータスを取得します
fcntl(2)	ファイル制御機能を実行します
ioctl(2)	デバイスを制御します
fpathconf(2)	構成可能なパス名変数を取得します

表 6–2 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します
mkdir(2)	ディレクトリを作成します
readlink(2)	シンボリックリンクの値を読み取ります
rename(2)	ファイル名を変更します
rmdir(2)	ディレクトリを削除します
symlink(2)	ファイルへのシンボリックリンクを作成します

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 6–3 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します
sync(2)	スーパーブロックを更新します
mount(2)	ファイルシステムをマウントします
statvfs(2)、fstatvfs(2)	ファイルシステム情報を取得します
sysfs(2)	ファイルシステムの種類の情報を取得します

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);
     }
     /* get currently set mode */
     mode = buf.st_mode;
     /* remove group execute permission from mode */
     mode &= ~(S_IEXEC>>3);
         /* set 'set group id bit' in mode */
     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)` コマンドも使用可能です。

```
$ chmod +l file
```

このコマンドはファイルモード内に `O20n0` アクセス権ビットを設定します。これはファイルの強制ロックを示します。*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 6–4 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。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;    /* setting a write lock */
    lck.l_whence = 0;    /* offset l_start from beginning of file */
    lck.l_start = (off_t)0;
    lck.l_len = (off_t)0;    /* until the end of the file */
    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、ロック要求のタイプ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。

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;    /* setting a write lock */
     lck.l_whence = 0;    /* offset l_start from beginning of file */
     lck.l_start = here;
```

```
     lck.l_len = sizeof(struct record);

     /* lock "this" with write lock */
     lck.l_start = this;
     if (fcntl(fd, F_SETLKW, &lck) < 0) {
         /* "this" lock failed. */
         return (-1);
 ...
}
```

次の例に、lockf(3C) インタフェースを示します。

```
#include <unistd.h>

{
 ...
     /* lock "this" */
     (void) lseek(fd, this, SEEK_SET);
     if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
         /* Lock on "this" failed. Clear lock on "here". */
         (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。ロックは前述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 6–2 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;
         (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);
             /* If this lock goes to the end of the address space, no
              * need to look further, so break out. */
             if (lck.l_len == 0) {
```

例 6–2 ファイル内でロックされているセグメントの出力 (続き)

```
            /* else, look for new lock after the one just found. */
                    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。また、クライアントまたはサーバー側のリソース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 6–3 lockf によるプロセスの検査

```
(void) lseek(fd, 0, 0L);
 /* set the size of the test region to zero (0). to test until the
    end of the file address space. */
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:
            /* bad argument passed to lockf */
            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。ファイルを閉じたあと、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

次の表に示すように、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) および termio(7I) のマニュアルページを参照してください。

表 6–5 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します
tcsendbreak(3C)、tcdrain(3C)、tcflush(3C)、tcflow(3C)	回線制御インタフェースを実行します
cfgetospeed(3C)、cfgetispeed(3C)、cfsetispeed(3C)、cfsetospeed(3C)	ボーレートを取得または設定します
tcsetpgrp(3C)	端末のフォアグラウンドプロセスのグループ ID を取得または設定します
tcgetsid(3C)	端末のセッション ID を取得します

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 6–4 制御端末との関連付けを解除する

```
(void) close(0);
(void) close(1);
(void) close(2);
(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あと、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 7 章

7 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 5.10 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)。

- 123 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。
- 名前付きパイプ (ファイル名を持つデータ待ち行列。)125 ページの「名前付きパイプ」では、名前付きパイプについて説明します。
- 128 ページの「System V IPC」では、System V のメッセージ待ち行列、セマフォー、および共有メモリーについて説明します。
- 126 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、セマフォー、および共有メモリーについて説明します。
- 125 ページの「ソケットの概要」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。
- 15 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します。

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。この呼び出しは引数の配列に 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:                   /* in child */
                        close( p[0] );
                        dup2( p[1], 1);
                        close P[1] );
                        exec( ... );
                        exit(1);
                default:                  /* in parent */
                        close( p[1] );
                        dup2( P[0], 0 );
                        close( p[0] );
                        break;
        }
        ...
```

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 7–1 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

表 7–1　パイプでの読み取りと書き込みの結果　　(続き)

実行	条件	結果
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ために、open(2) は呼び出されると、O_NDELAY マスク (sys/fcntl.h にある) 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)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。

ソケットの概要

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コンポーネントです。ソケットは、名前をバインド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 10 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の詳細は、付録 A 「UNIX ドメインソケット」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のソケットについては、第 8 章「ソケットインタフェース」で説明されています。

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ー。
- 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 7–2 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	目的
mq_open(3RT)	名前付きメッセージ待ち行列に接続します。指定によっては作成します
mq_close(3RT)	開いているメッセージ待ち行列への接続を終了します
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します
mq_send(3RT)	メッセージを待ち行列に入れます
mq_receive(3RT)	もっと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か、削除します
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します

表 7–2 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	目的
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します

POSIX セマフォー

POSIX セマフォーは、System V セマフォーより軽量です。POSIX セマフォー構造体は 25 個までのセマフォーの配列ではなく、1 つのセマフォーだけを定義します。

次の表に、POSIX セマフォーインタフェースの一覧を示します。

sem_open(3RT)	名前付きセマフォーに接続する。指定によっては作成します
sem_init(3RT)	名前なしセマフォー構造体を初期化します (呼び出し元プログラムの内部で行われるため、名前付きセマフォーではない)
sem_close(3RT)	開いているセマフォーへの接続を終了します
sem_unlink(3RT)	開いているセマフォーへの接続を終了し、最後のプロセスがセマフォーを閉じるときにセマフォーを削除します
sem_destroy(3RT)	名前なしセマフォー構造体を初期化します (呼び出し元プログラムの内部で行われるため、名前付きセマフォーではない)
sem_getvalue(3RT)	セマフォーの値を指定された整数にコピーします
sem_wait(3RT)、sem_trywait(3RT)	セマフォー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します
sem_post(3RT)	セマフォーの数を増やします

POSIX 共有メモリー

POSIX 共有メモリーは、実際にマッピングされているメモリーの変形です (15 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) を使用します。
- オブジェクトを閉じるか削除するには、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) を使用します。

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 5.10 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、セマフォー、および共有メモリーのアクセス権

メッセージ、セマフォー、および共有メモリーには、通常のファイルと同じように、ほかのユーザーに対する読み取り権と書き込み権 (ただし、実行権はない)、および所有者、グループがあります。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、その機能を識別する必要があります。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、セマフォー、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 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。

これら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 を指定しなし場合、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、キー引数に対応する待ち行列のメッセージ待ち行列 ID (msqid) 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。この制限を超えると、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;         /* key to be passed to msgget() */
        int      msgflg,      /* msgflg to be passed to msgget() */
                 msqid;       /* return value from msgget() */
        ...
        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　待ち行列のステータスの情報を buf が指すデータ構造体に入れます。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ります。

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) を設定します。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 を持つ必要があります。

IPC_RMID　msqid 引数で指定したメッセージ待ち行列を削除します。

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 である必要があります。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;        /* message flags for the operation */
        struct msgbuf   *msgp;         /* pointer to the message buffer */
        size_t          msgsz;         /* message size */
        size_t          maxmsgsize;    /* maximum message size */
        long            msgtyp;        /* desired message type */
        int             msqid          /* message queue ID to be used */
        ...
        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 セマフォー

セマフォーを使用すると、プロセスはステータス情報を問い合わせたり、変更したりできます。通常、セマフォーは共有メモリーセグメントなどのシステムリソース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。セマフォーは、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 セマフォーは、大きな配列の中に存在できるため、極めて重いセマフォーです。より軽量なセマフォーは、スレッドライブラリで利用できます。また、POSIX セマフォーは System V セマフォーの最新の実装です (127 ページの「POSIX セマフォー」を参照)。スレッドライブラリセマフォーは、マッピングされたメモリーで使用する必要があります (15 ページの「メモリー管理インタフェース」を参照)。

セマフォーのセットは、制御構造体と個々のセマフォー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。セマフォーのセットは、`semget(2)` を使用して初期化する必要があります。セマフォー作成者は `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。

セマフォー操作は `semop(2)` によって行います。このインタフェースは、セマフォー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、セマフォー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、セマフォーがゼロ値を持っているかどうかを検査できます。セマフォー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、どのセマフォーも変更されません。`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。

- セマフォー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。
- セマフォーのセットが削除された。

セマフォーを更新できるのは、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。

セマフォー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ーの解放に失敗した場合、セマフォーはメモリー内にロック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、各セマフォー操作に undo 構造体を割り当て、セマフォー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、セマフォーの整合性が保たれます。

プロセスがセマフォーによって制御されるリソース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ーに対する操作を行わないでください。現在、リソースを制御しているプロセスが異常終了すると、そのリソース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リソース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ー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。

セマフォーのセットの初期化

semget(2) は、セマフォーの初期化またはセマフォーへのアクセスを行います。呼び出しが成功すると、セマフォー ID (semid) を返します。key 引数は、セマフォー ID に関連付けられた値です。nsems 引数は、セマフォー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、配列内のセマフォーの最大数を指定します。SEMMNS オプションは、すべてのセマフォーのセットを通じて個々のセマフォーの最大数を指定します。ただし、セマフォー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ー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;       /* key to pass to semget() */
        int      semflg;    /* semflg to pass to semget() */
        int      nsems;     /* nsems to pass to semget() */
        int      semid;     /* return value from semget() */
        ...
        key = ...
        nsems = ...
        semflg = ...
        ...
        if ((semid = semget(key, nsems, semflg)) == -1) {
                perror("semget: semget failed");
                exit(1);
        } else
                exit(0);
...
```

セマフォーの制御

semctl(2) は、セマフォーのセットのアクセス権とその他の特性を変更します。有効なセマフォー 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ーを選択します。*cmd* 引数は、次のいずれかの制御フラグです。

GETVAL	単一セマフォーの値を戻します。
SETVAL	単一セマフォーの値を設定します。この場合、arg は int 値の arg.val と解釈されます。
GETPID	セマフォー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 を戻します。
GETNCNT	セマフォーの値が増加するのを待っているプロセス数を戻します。
GETZCNT	特定のセマフォーの値が 0 に達するのを待っているプロセス数を戻します。
GETALL	セット内のすべてのセマフォーの値を戻します。この場合、arg は unsigned short 値の配列へのポインタである arg.array と解釈されます。
SETALL	セット内のすべてのセマフォーに値を設定します。この場合、arg は unsigned short 値の配列へのポインタである arg.array と解釈されます。
IPC_STAT	制御構造体からセマフォーのセットのステータス情報を取得し、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ます。

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します。この場合、arg は arg.buf と解釈されます。
IPC_RMID	指定したセマフォーのセットを削除します。

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);
...
```

セマフォーの操作

semop(2) は、セマフォー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ー ID です。sops 引数は、セマフォー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。

- セマフォー番号
- 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ー操作を指定します。nsops 引数は配列の長さを指定します。配列の最大長は、*SEMOPM* 構成オプションで指定されます。このオプションでは、単一の semop(2) 呼び出しで使用できる最大操作数が決定され、デフォルトではその値は 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、セマフォーの値をその数だけ増加します。
- 負の整数の場合は、セマフォーの値をその数だけ減少します。セマフォーを 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ます。
- 値が 0 の場合は、セマフォーの値が 0 になるのを待ちます。

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ます。IPC_NOWAIT が設定されている操作を実行できなかった場合、セマフォーの値を変更せずにインタフェースを戻します。セマフォーを現在の値より多く減らそうしたり、セマフォーが 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します。

SEM_UNDO　プロセスの終了時に配列内の個々の操作を取り消します。

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;              /* work area */
        int             nsops;          /* number of operations to do */
        int             semid;          /* semid of semaphore set */
        struct sembuf   *sops;          /* ptr to operations to perform */
        ...
        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 5.10 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もっとも効率的です。詳細は、第 1 章「メモリーと CPU の管理」を参照してください。

SunOS 5.10 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、セマフォー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あとは、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;      /* key to be passed to shmget() */
        int       shmflg;   /* shmflg to be passed to shmget() */
        int       shmid;    /* return value from shmget() */
        size_t    size;     /* size to be passed to shmget() */
        ...
        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します。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 を持つ必要があります。
SHM_UNLOCK	共有メモリーセグメントのロックを解除します。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 を持つ必要があります。

IPC_STAT	制御構造体にあるステータス情報を取得して、buf が指すバッファーに入れます。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ります。
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します。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 を持つ必要があります。
IPC_RMID	共有メモリーセグメントを削除します。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 を持つ必要があります。

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;                /* command code for shmctl() */
int     shmid;              /* segment ID */
struct  shmid_ds  shmid_ds; /* shared memory data structure to hold results */
        ...
        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返します。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 { /* Internal record of attached segments. */
        int          shmid;        /* shmid of attached segment */
        char         *shmaddr;     /* attach point */
        int          shmflg;       /* flags used on attach */
} ap[MAXnap];                      /* State of current attached segments. */
int     nap;                       /* Number of currently attached segments. */
...
char    *addr;                     /* address work variable */
```

```
register int          i;            /* work area */
register struct state *p;           /* ptr to current state entry */
...
        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 8 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。

- 141 ページの「SunOS 4 のバイナリ互換性」では、SunOS 4 環境とのバイナリ互換について説明します。
- 145 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、および閉鎖について説明します。
- 166 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ーについて説明します。
- 171 ページの「ソケットの拡張機能」では、マルチキャスト、非同期ソケットなどの拡張機能について説明します。
- 190 ページの「Stream Control Transmission Protocol (SCTP)」では、ストリーム制御伝送プロトコル (Stream Control Transmission Protocol、SCTP) を実装するために使用するインタフェース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 5.10 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 5.10 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 5.10 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 年以降の SunOS リリースの必須部分になっています。ソケットは、名前をバインド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) インタフェースとともに動作します。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 が、その他のシステムサービスライブラリとともに /usr/lib に含まれています。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ーを定義します。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 の両方で通信可能なソケットの場合に、値 AF_INET6 によって識別されます。以前のアプリケーションとのソース互換性および IPv4 への raw アクセスのために、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 4 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、TCP を使用したプロセスの通信を可能にします。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、UDP を使用したプロセスの通信を可能にします。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、送信シーケンスとは異なる順序でメッセージを受信することがあります。また、データグラムソケット側のプロセスは、重複したメッセージを受信することがあります。データグラムソケットで送信されるメッセージは、失われる場合があります。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。raw ソケットは、ネットワーキングスタックによって直接サポートされない IP ベースのほかのプロトコルへのアクセスも提供します。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。
- *SEQ* ソケットは、1 対 N のストリーム制御伝送プロトコル (Stream Control Transmission Protocol、SCTP) コネクションをサポートします。SCTP の詳細は、190 ページの「Stream Control Transmission Protocol (SCTP)」で説明しています。

詳細については、175 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 5.10 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、および XNS 5 (Unix03) (SunOS バージョン 5.7 以降) 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 (Unix03) 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

このセクションでは、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。AF_INET6、AF_INET 、および AF_UNIX では、SOCK_STREAM、SOCK_DGRAM または SOCK_RAW のタイプがサポートされます。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。171 ページの「ソケットの拡張機能」で説明するように、デフォルト以外のプロトコルを指定することもできます。

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) インタフェースによって、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をバインドします。

```
#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する 176 ページの「アドレスのバインド」に示されています。

コネクションの確立

通常のコネクション確立は、クライアントとしてのプロセス動作とサーバーとしてのプロセス動作によって、非対称に行われます。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、176 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);                   /* Allow queue of 5 connections */
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 番目のパラメータは、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。

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、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、171 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 8–1 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	指定された時間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ステータス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ステータス情報によっても発生する。返されるステータス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

このセクションでは、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);
```

send(3SOCKET) および recv(3SOCKET) も使用できます。

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

send(3SOCKET) および recv(3SOCKET) は、read(2) および write(2) に非常に似ていますが、flags 引数が重要です。次のうちの 1 つまたは複数が必要である場合、flags 引数 (sys/socket.h で定義) は 0 以外の値として指定できます。

MSG_OOB	帯域外データを送受信する
MSG_PEEK	データを読み取らずに検索する
MSG_DONTROUTE	ルーティングパケットなしでデータを送信する

帯域外データは、ストリームソケットに固有です。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ルーティングパケットなしでデータを送信するオプションは現在、ルーティング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によって破棄できます。close(2) のあと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 ソケットを適切に閉じます。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

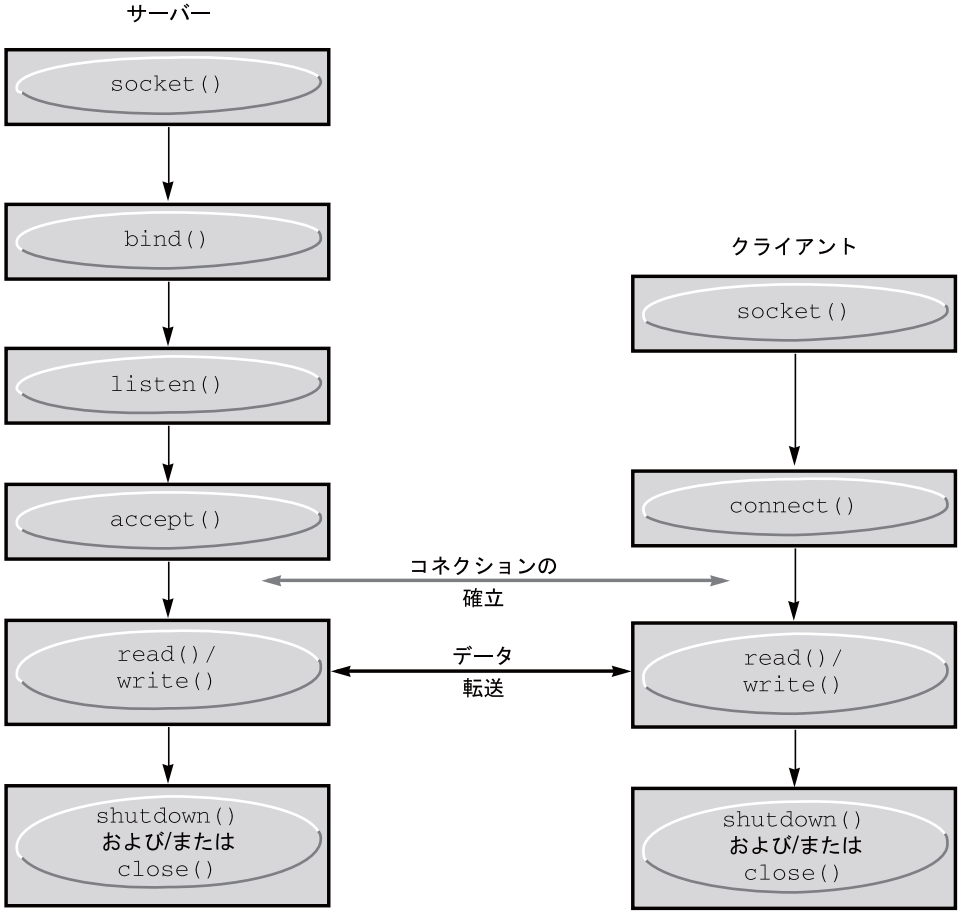
1 それ以上の送信を許可しない

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 8–1 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、176 ページの「アドレスのバインド」で説明しています。

例 8–1　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>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
/*
 * This program creates a socket and then begins an infinite loop.
 * Each time through the loop it accepts a connection and prints
 * data from it. When the connection breaks, or the client closes
 * the connection, the program accepts a new connection.
*/
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    /* Create socket. */
    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Bind socket using wildcards.*/
    bzero (&server, sizeof(server));
    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);
    }
    /* Find out assigned port number and print it out. */
    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));
    /* Start accepting connections. */
    listen(sock, 5);
    do {
```

例 8–1 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)

```
        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, sizeof(buf))) == -1)
                perror("reading stream message");
            if (rval == 0)
                printf("Ending connection\n");
            else
                /* assumes the data is printable */
                printf("-->%s\n", buf);
        } while (rval > 0);
        close(msgsock);
    } while(TRUE);
    /*
     * Since this program has an infinite loop, the socket "sock" is
     * never explicitly closed. However, all sockets are closed
     * automatically when a process is killed or terminates normally.
     */
    exit(0);
}
```

コネクションを開始するため、例 8–2 のクライアントプログラムでは、ストリームソケットを作成し、コネクションのためのソケットのアドレスを指定して connect(3SOCKET) を呼び出しています。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。このプログラムに含まれる ntohl(3SOCKET)、ntohs(3SOCKET)、htons(3SOCKET)、および htonl(3XNET) などのデータ表現ルーチンの詳細は、byteorder(3SOCKET) のマニュアルページを参照してください。

例 8–2 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 . . ."
/*
 * This program creates a socket and initiates a connection with
 * the socket given in the command line. Some data are sent over the
 * connection and then the socket is closed, ending the connection.
 * The form of the command line is: streamwrite hostname portnumber
 * Usage: pgm host port
 */
main(int argc, char *argv[])
{
    int sock, errnum, h_addr_index;
```

例 8–2 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)

```
    struct sockaddr_in6 server;
    struct hostent *hp;
    char buf[1024];
    /* Create socket. */
    sock = socket( AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Connect socket using name specified by command line. */
    bzero (&server, sizeof (server));
    server.sin6_family = AF_INET6;
    hp = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &errnum);
/*
 * getipnodebyname returns a structure including the network address
 * of the specified host.
 */
    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) {
                /* Try next address */
                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);
}
```

ストリームソケットに 1 対 1 の SCTP コネクションのサポートを追加できます。次の例のコードでは、既存のプログラムに -p を追加することにより、使用するプロトコルをプログラムで指定できるようにしています。

例 8–3 ストリームソケットへの SCTP サポートの追加

```
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <errno.h>

int
main(int argc, char *argv[])
{
    struct protoent *proto = NULL;
    int c;
    int s;
    int protocol;

    while ((c = getopt(argc, argv, "p:")) != -1) {
        switch (c) {
        case 'p':
            proto = getprotobyname(optarg);
            if (proto == NULL) {
                fprintf(stderr, "Unknown protocol: %s\n",
                        optarg);
                return (-1);
            }
            break;
        default:
            fprintf(stderr, "Unknown option: %c\n", c);
            return (-1);
        }
    }

    /* Use the default protocol, which is TCP, if not specified. */
    if (proto == NULL)
        protocol = 0;
    else
        protocol = proto->p_proto;

    /* Create a IPv6 SOCK_STREAM socket of the protocol. */
    if ((s = socket(AF_INET6, SOCK_STREAM, protocol)) == -1) {
        fprintf(stderr, "Cannot create SOCK_STREAM socket of type %s: "
                "%s\n", proto != NULL ? proto->p_name : "tcp",
                strerror(errno));
        return (-1);
    }
    printf("Success\n");
    return (0);
}
```

入出力の多重化

要求は、複数のソケットまたは複数のファイルに多重化できます。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 番目、3 番目、4 番目の引数は、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、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*) がセットをクリアします。

select(3C) に 5 番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) ルーチンは通常、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ステータス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 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 8–4　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>
#include <stdio.h>
#define TRUE 1
/*
 * This program uses select to check that someone is
 * trying to connect before calling accept.
 */
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    fd_set ready;
    struct timeval to;
    /* Open a socket and bind it as in previous examples. */
    /* Start accepting connections. */
    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, sizeof(buf))) == -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。

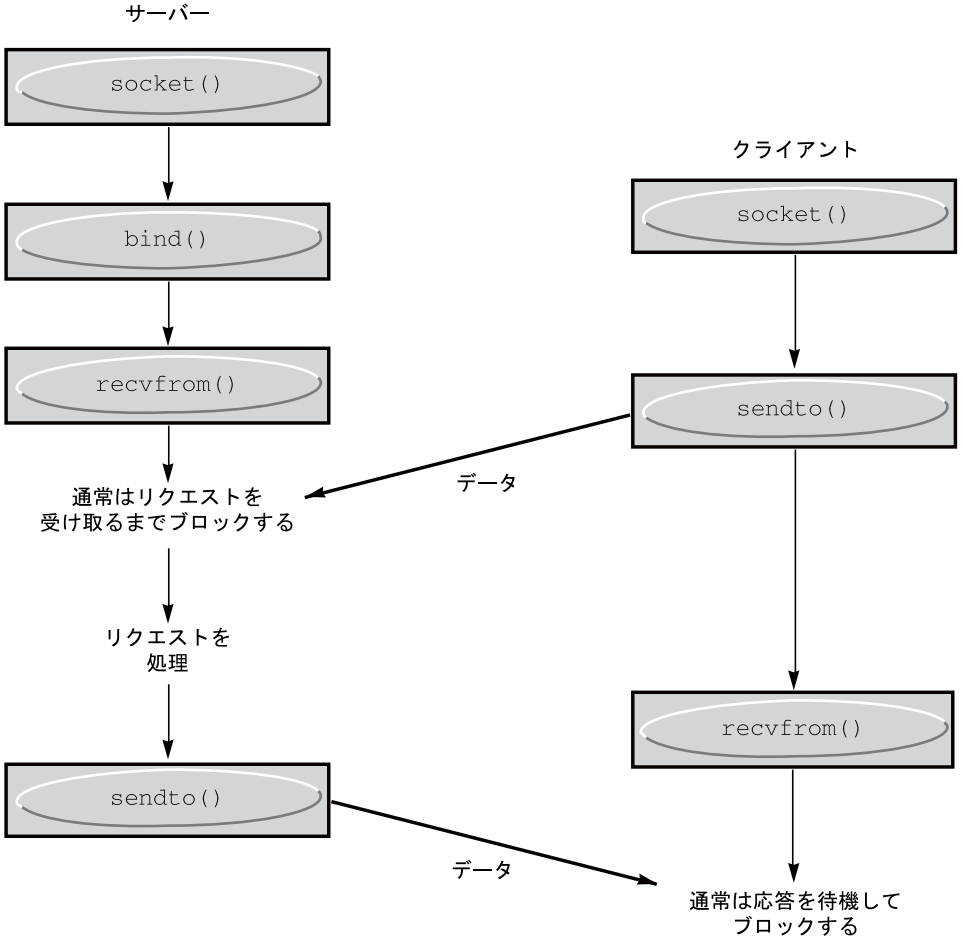
select(3C) ルーチンは同期多重化スキームを提供します。SIGIO シグナルと SIGURG シグナル (171 ページの「ソケットの拡張機能」を参照) によって、出力の完了、入力の有効性、および例外条件の非同期通知を指定できます。

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。各メッセージには着信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8–2 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



145 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。呼び出しの前に、*fromlen* が *from* バッファーのサイズに設定されます。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、ソケットを特定の着信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。着信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 番目の connect(3SOCKET) 呼び出しは、着信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、エラーステータスを問い合わせることもできます。

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 8–5 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 . . ."
/*
 * Here I send a datagram to a receiver whose name I get from
 * the command line arguments. The form of the command line is:
 * dgramsend hostname portnumber
 */
main(int argc, char *argv[])
{
    int sock, errnum;
    struct sockaddr_in6 name;
    struct hostent *hp;
    /* Create socket on which to send. */
    sock = socket(AF_INET6,SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /*
```

例 8–5 インターネットファミリデータグラムの送信 *(続き)*

```
     * Construct name, with no wildcards, of the socket to “send’’
     * to. getinodebyname returns a structure including the network
     * address of the specified host. The port number is taken from
     * the command line.
     */
    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]));
    /* Send message. */
    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 8–6 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>
/*
 * This program creates a datagram socket, binds a name to it, then
 * reads from the socket.
 */
 main()
{
    int sock, length;
    struct sockaddr_in6 name;
    char buf[1024];
    /* Create socket from which to read. */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /* Create name with wildcards. */
    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 8–6 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);
    }
    /* Find assigned port value and print it out. */
    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));
    /* Read from the socket. */
    if (read(sock, buf, 1024) == -1 )
        perror("receiving datagram packet");
    /* Assumes the data is printable */
    printf("-->%s\n", buf);
    close(sock);
    exit(0);
}
```

標準ルーチン

このセクションでは、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、このセクションで記載されているインタフェースは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ーによって異なります。

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)`、`gai_strerror(3SOCKET)`、および `freeaddrinfo(3SOCKET)` 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。これら
は、getipnodebyname(3SOCKET)、gethostbyname(3NSL)、および
getservbyname(3SOCKET) の API よりも新しいインタフェースです。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。gai_strerror(3SOCKET) を呼び出すと、getaddrinfo(3SOCKET) と getnameinfo(3SOCKET) から返される EAI_xxx コードに基づくエラーメッセージが出力されます。

次に、getaddrinfo(3SOCKET) の使用例を示します。

```
struct addrinfo        *res, *aip;
struct addrinfo        hints;
int                    error;

/* Get host address.  Any type of address will do. */
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

インターネットホスト名とアドレスのマッピングは、gethostent(3NSL) に定義するように hostent 構造体によって表現されます。

```
struct hostent {
    char  *h_name;            /* official name of host */
    char  **h_aliases;        /* alias list */
    int   h_addrtype;         /* hostaddrtype(e.g.,AF_INET6) */
    int   h_length;           /* length of address */
    char  **h_addr_list;      /* list of addrs, null terminated */
};
/*1st addr, net byte order*/
#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 で終了す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

ネットワーク名をネットワーク番号にマッピングし、netent 構造体を返すルーチンです。

```
/*
 * Assumes that a network number fits in 32 bits.
 */
struct netent {
   char     *n_name;      /* official name of net */
   char     **n_aliases;  /* alias list */
   int      n_addrtype;   /* net address type */
   int      n_net;        /* net number, host byte order */
};
```

getnetbyname(3SOCKET)、getnetbyaddr_r(3SOCKET)、および getnetent(3SOCKET) は、前述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;       /* official protocol name */
    char    **p_aliases    /* alias list */
    int     p_proto;       /* protocol number */
};
```

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。サービス名とポート番号のマッピング
は、getprotoent(3SOCKET) で定義される servent 構造体によって記述されます。

```
struct servent {
    char    *s_name;        /* official service name */
    char    **s_aliases;    /* alias list */
    int     s_port;         /* port number, network byte order */
    char    *s_proto;       /* protocol to use */
};
```

getservbyname(3SOCKET) は、サービス名、およびオプションとして修飾プロトコルを servent 構造体にマッピングします。呼び出し:

```
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。呼び出し:

```
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) によって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 8–2 実行時ライブラリルーチン

インタフェース	摘要
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への 32 ビット整数バイトオーダー変換
htons(3SOCKET)	ホストからネットワークへの 16 ビット整数バイトオーダー変換
ntohl(3SOCKET)	ネットワークからホストへの 32 ビット整数バイトオーダー変換
ntohs(3SOCKET)	ネットワークからホストへの 16 ビット整数バイトオーダー変換

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ーでは、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。このスキームでは、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。

代替スキームとして、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があります。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) の詳細は、179 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されます。既知の UNIX ファミリ名の例には、rlogin サービスがあります。例 8–7 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, 1);
(void) dup2(0, 2);
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、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。

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, &api);
```

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 8–7 サーバーのメインループ

```
/* Wait for a connection request. */
for (;;) {
    faddrlen = sizeof (faddr);
    new_sock = accept(sock, (struct sockaddr *)api->ai_addr,
            api->ai_addrlen)
    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) によって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

このセクションでは、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) {
    /*
     * Open socket.  The address type depends on what
     * getaddrinfo() gave us.
     */
    sock = socket(aip->ai_family, aip->ai_socktype,
          aip->ai_protocol);
    if (sock == -1) {
        perror("socket");
        freeaddrinfo(res);
        return (-1);
    }

    /* Connect to the host. */
    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

一部のサービスでは、データグラムソケットを使用します。rwho(1) サービスは、LAN に接続されたホストについてのステータス情報を提供します。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ステータスを取得できます。次の例に、典型的な出力例を示します。

例 8–8　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
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ステータス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ステータス情報を受信します。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、各ホストのステータス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ステータス情報を受信し、次に、このコードを実行しているホストのステータス情報を提供します。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を調べて、そのパケットが別の rwho(1) サーバープロセスから送信されたことを確認し、到着時間を記録します。次に、パケットはホストのステータス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 8–9　rwho(1) サーバー

```
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");
```

例 8–9 rwho(1) サーバー (続き)

```
        exit(1);
    }
    bind(s, (struct sockaddr *) &sin, sizeof sin);
    ...
    signal(SIGALRM, onalrm);
    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 つめのサーバータスクは、そのホストのステータスの供給です。このタスクでは、周期的にシステムステータス情報を取得し、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。

ステータス情報は、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。このセクションでは、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 ID 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 に関する 174 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) を呼び出し、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。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。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。そうでない場合は、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 フラグを使用して 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 8–10　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;

    /* flush local terminal output */
    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 8–11　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された時に発生する、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 8–12 ソケットを非同期にする

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
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 (次のセクションを参照)。

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 (173 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 8–13 入出力要求の非同期通知

```
#include <fcntl.h>
#include <sys/file.h>
 ...
signal(SIGIO, io_handler);
/* Set the process receiving SIGIO/SIGURG signals to us. */
if (fcntl(s, F_SETOWN, getpid()) < 0) {
```

例 8–13　入出力要求の非同期通知　　(続き)

```
    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、そのあと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 is the out-of-band data handling routine */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
    perror("ioctl: SIOCSPGRP");
}
```

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) は要求したタイプの返されたソケットがデフォルトのプロトコルを使用することを選択します。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。

インターネットファミリで raw ソケットを使用して新しいプロトコルを IP 上に実装すると、ソケットは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、161 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。

特定のポート上に常駐するサービスを、そのポートに bind(3SOCKET) でバインドすることができます。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 を超えるインターネットポート番号を任意に使用できる。インターネットポート番号の最大値は 65535。
- 現在、ポート番号はほかのソケットにバインドされていない。

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。

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 stdio の規定に従って提供されるバッファリングされた入出力を使用できます。

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)) {
            /* this is a IPv4-mapped IPv6 address */
            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ているのは 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リソースの検索
- 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群ごとに 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 8–14　net/if.h ヘッダーファイル

```
struct ifreq {
    #define IFNAMSIZ 16
    char ifr_name[IFNAMSIZ]; /* if name, e.g., "en0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        char ifru_oname[IFNAMSIZ]; /* other if name */
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_metric;
        char ifru_data[1]; /* interface dependent data */
        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。

```
/*
 * Do SIOCGIFNUM ioctl to find the number of interfaces
 *
 * Allocate space for number of interfaces found
 *
 * Do SIOCGIFCONF with allocated buffer
 *
 */
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。ホストに接続されているすべてのネットワークはそれぞれ、関連付けられた `ifreq` 構造体を 1 つ持っています。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群の番号順

`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 8–15 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++) {
    /*
     * Be careful not to use an interface devoted to an address
     * family other than those intended.
     */
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
        ...
    }
    /* Skip boring cases */
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
            (ifr->ifr_flags & IFF_LOOPBACK) ||
```

例 8–15 インタフェースフラグの取得 (続き)

```
            (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 8–16 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着信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、SOCK_DGRAM と SOCK_RAW タイプの AF_INET6 および AF_INET ソケット上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) 呼び出しで着信先アドレスとして 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。この場合、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ることはありません。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。これにより、マルチキャストの配信範囲を制御します。

```
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 までの着信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、ルーティング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してい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;   /* multicast group to join */
    struct in_addr imr_interface;   /* interface to join on */
}
```

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 INADDR_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。

次に示すコードが bind(3SOCKET) の前にあると、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に必要な定義は、<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) 呼び出しで着信先アドレスとして 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。これにより、マルチキャストの配信範囲を制御します。

```
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エンコード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　全世界的な配信範囲

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 multicast addr */
    unsigned int       ipv6mr_interface;    /* interface index */
}
```

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。

次に示すコードが bind(3SOCKET) の前にあると、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に必要な定義は、<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。

Stream Control Transmission Protocol (SCTP)

ストリーム制御伝送プロトコル (SCTP) は、TCP が提供するサービスと同様のサービスを提供する信頼性の高いトランスポートプロトコルです。さらに、SCTP はネットワークレベルの耐障害性を提供します。SCTP では、関連付けの両端でマルチホーミングがサポートされます。SCTP ソケット API は、TCP にならった 1 対 1 ソケットスタイルをサポートします。その他に SCTP ソケット API は、シグナリングで使用するために設計された 1 対多ソケットスタイルもサポートします。1 対多ソケットスタイルは、プロセスで使用されるファイル記述子の数を削減します。SCTP 関数の呼び出しを使用するには、`libsctp` ライブラリをリンクする必要があります。

SCTP 関連付けは 2 つの終端の間で設定します。各終端は、特定のタイプのサービス拒否 (DoS) 攻撃から保護するために、cookie による 4 ウェイハンドシェークを使用します。複数の IP アドレスによって終端を表現できます。

SCTP スタックの実装

このセクションでは、IETF 勧告の Stream Control Transmission Protocol (RFC 2960) および Stream Control Transmission Protocol Checksum Change (RFC 3309) について、Solaris における実装の詳細を表に示します。このセクションの表では、RFC 2960 勧告の実装例外について示します。Solaris オペレーティングシステムの SCTP プロトコルは、RFC 3309 のすべてと、RFC 2960 のセクションのうちこの表に明記されていないセクションのすべてを実装しています。

表 **8–3** RFC 2960 との比較における Solaris SCTP 実装の例外

RFC 2960 のセクション	**Solaris** 実装における例外
3. SCTP Packet Format	3.2 Chunk Field Descriptions: Solaris SCTP は、オプションの ECNE および CWR を実装していません。
	3.3.2: Solaris SCTP は、Initiation (INIT) の Optional ECN、Host Name Address、および Cookie Preserving パラメータを実装していません。
	3.3.3: Solaris SCTP は、Initiation Acknowledgement、Optional ECN、および Host Name Address パラメータを実装していません。
5. Association Initialization	5.1.2, Handle Address Parameters: セクション (B) の Optional Host Name は実装されていません。
6. User Data Transfer	6.8, Adler-32 Checksum Calculation: RFC 3309 によって、このセクションは廃止されています。

表 8–3　RFC 2960 との比較における Solaris SCTP 実装の例外　(続き)

RFC 2960 のセクション	Solaris 実装における例外
10. Interface with Upper Layer	Solaris SCTP は、IETF TSVWG SCTP ソケット API ドラフトを実装しています。

注 – TSVWG SCTP ソケット API の Solaris 10 実装は、Solaris 10 が最初に出荷されたときに公開されていた API ドラフトのバージョンに基づいています。

SCTP ソケットインタフェース

`socket()` 呼び出しは、`IPPROTO_SCTP` のソケットを作成するとき、SCTP 固有のソケット作成ルーチンを呼び出します。SCTP ソケットでソケットを呼び出すと、自動的に適切な SCTP ソケットルーチンが呼び出されます。1 対 1 ソケットの場合、各ソケットは 1 つの SCTP 関連付けに対応します。1 対 1 ソケットを作成するには、次の関数を呼び出します。

```
socket(AF_INET[6], SOCK_STREAM, IPPROTO_STCP);
```

1 対多スタイルソケットの場合、各ソケットは複数の SCTP 関連付けを処理します。各関連付けには、*sctp_assoc_t* と呼ばれる関連付け識別子があります。1 対多ソケットを作成するには、次の関数を呼び出します。

```
socket(AF_INET[6], SOCK_SEQPACKET, IPPROTO_STCP);
```

sctp_bindx()

```
int sctp_bindx(int sock, void *addrs, int addrcnt, int flags);
```

`sctp_bindx()` 関数は、SCTP ソケット上のアドレスを管理します。*sock* パラメータが IPv4 ソケットである場合、`sctp_bindx()` 関数に渡されるアドレスは IPv4 アドレスである必要があります。*sock* パラメータが IPv6 ソケットである場合、`sctp_bindx()` 関数に渡されるアドレスは IPv4 アドレスまたは IPv6 アドレスのどちらかになります。`sctp_bindx()` 関数に渡されるアドレスが `INADDR_ANY` または `IN6ADDR_ANY` であると、ソケットは使用可能なすべてのアドレスにバインドします。`bind(3SOCKET)` を使用して SCTP 終端をバインドします。

**addrs* パラメータの値は、1 つ以上のソケットアドレスの配列へのポインタです。各アドレスは、それぞれ該当する構造体に含まれます。アドレスが IPv4 アドレスである場合、`sockaddr_in` 構造体または `sockaddr_in6` 構造体に含まれます。IPv6 アドレスである場合、`sockaddr_in6` 構造体に含まれます。アドレスタイプ群によって、アドレス長が異なります。呼び出し元は、配列内のアドレスの数を *addrcnt* パラメータによって指定します。

`sctp_bindx()` 関数は成功すると 0 を返します。失敗すると、`sctp_bindx()` 関数は -1 を返し、`errno` の値を該当するエラーコードに設定します。

各ソケットアドレスに同じポートが指定されていない場合、sctp_bindx() 関数は失敗し、errno の値を EINVAL に設定します。

flags パラメータは、現在定義されている次のフラグの 0 以上に対してビット単位の論理和演算を実行することによって求められます。

- SCTP_BINDX_ADD_ADDR
- SCTP_BINDX_REM_ADDR

SCTP_BINDX_ADD_ADDR は、指定されたアドレスを関連付けに追加するように SCTP に指示します。SCTP_BINDX_REM_ADDR は、指定されたアドレスを関連付けから削除するように SCTP に指示します。この 2 つのフラグは相互に排他です。両方が指定された場合、sctp_bindx() は失敗して errno の値を EINVAL に設定します。

呼び出し元は、関連付けからすべてのアドレスを削除することはできません。このような試みは拒否され、sctp_bindx() 関数は失敗して errno の値が EINVAL に設定されます。アプリケーションは、bind() 関数を呼び出したあとに sctp_bindx(SCTP_BINDX_ADD_ADDR) を使用することにより、追加のアドレスを終端に関連付けることができます。またアプリケーションは、sctp_bindx(SCTP_BINDX_REM_ADDR) を使用することにより、待機しているソケットに関連付けられているアドレスを削除できます。アドレスを削除するために sctp_bindx(SCTP_BINDX_REM_ADDR) を使用したあとは、新しい関連付けを受け入れても、削除されたアドレスは再び関連付けられません。終端が動的アドレスをサポートする場合、SCTP_BINDX_REM_ADDR または SCTP_BINDX_ADD_ADDR を使用すると、ピアにメッセージが送信されてピアのアドレスリストが変更されます。接続されている関連付けにおけるアドレスの追加および削除は、オプションの機能です。この機能をサポートしない実装では、EOPNOTSUPP が返されます。

アドレス群が AF_INET または AF_INET6 ではない場合、sctp_bindx() 関数は失敗して EAFNOSUPPORT を返します。sctp_bindx() に渡される *sock* パラメータのファイル記述子が無効である場合、sctp_bindx() 関数は失敗して EBADF を返します。

sctp_opt_info()

int **sctp_opt_info**(int *sock*, sctp_assoc_id_t *id*, int *opt*, void **arg*, socklen_t **len*);

sctp_opt_info() 関数は、*sock* パラメータに記述されるソケットに関連付けられている SCTP レベルのオプションを返します。1 対多スタイルの SCTP ソケットの場合、*id* パラメータの値は特定の関連付けを指します。1 対 1 スタイルの SCTP ソケットの場合、*id* パラメータは無視されます。*opt* パラメータの値は、取得する SCTP ソケットオプションを指定します。*arg* パラメータの値は、呼び出し元プログラムによって割り当てられるオプション固有の構造体バッファーです。**len* パラメータの値は、オプションの長さです。

opt パラメータは、次の値を取ることができます。

SCTP_RTOINFO

調整可能な再伝送タイムアウト(RTO) を初期化および設定するために使用されるプロトコルのパラメータを返します。プロトコルのパラメータは次の構造体を使用します。

```
struct sctp_rtoinfo {
    sctp_assoc_t    srto_assoc_id;
    uint32_t        srto_initial;
    uint32_t        srto_max;
    uint32_t        srto_min;
};
```

srto_assoc_id　対象になる関連付けを指定するこの値は、呼び出し元プログラムが提供します。

srto_initial　この値は初期 RTO 値です。

srto_max　この値は最大 RTO 値です。

srto_min　この値は最小 RTO 値です。

SCTP_ASSOCINFO

関連付け固有のパラメータを返します。パラメータは次の構造体を使用します。

```
struct sctp_assocparams {
    sctp_assoc_t    sasoc_assoc_id;
    uint16_t        sasoc_asocmaxrxt;
    uint16_t        sasoc_number_peer_destinations;
    uint32_t        sasoc_peer_rwnd;
    uint32_t        sasoc_local_rwnd;
    uint32_t        sasoc_cookie_life;
};
```

sasoc_assoc_id
　対象になる関連付けを指定するこの値は、呼び出し元プログラムが提供します。

sasoc_assocmaxrxt
　この値は、関連付けに対する再伝送の最大数を指定します。

sasoc_number_peer_destinations
　この値は、ピアが持つアドレスの数を指定します。

sasoc_peer_rwnd
　この値は、ピアの受信ウィンドウの現在値を指定します。

sasoc_local_rwnd
　この値は、ピアの送信先の、最後に報告された受信ウィンドウを指定します。

`sasoc_cookie_life`
この値は、関連付けの cookie の存続時間を指定し、cookie を発行する際に使用されます。

時間の値を使用するすべてのパラメータはミリ秒単位です。

`SCTP_DEFAULT_SEND_PARAM` sendto(3SOCKET) 関数の呼び出しが関連付けに関して使用するパラメータのデフォルトセットを返します。パラメータは次の構造体を使用します。

```
struct sctp_sndrcvinfo {
    uint16_t        sinfo_stream;
    uint16_t        sinfo_ssn;
    uint16_t        sinfo_flags;
    uint32_t        sinfo_ppid;
    uint32_t        sinfo_context;
    uint32_t        sinfo_timetolive;
    uint32_t        sinfo_tsn;
    uint32_t        sinfo_cumtsn;
    sctp_assoc_t    sinfo_assoc_id;
};
```

`sinfo_stream`
この値は、`sendmsg()` 呼び出しのデフォルトストリームを指定します。

`sinfo_ssn`
この値は常に 0 です。

`sinfo_flags` この値は、`sendmsg()` 呼び出しのデフォルトフラグです。フラグは次の値を取ることができます。

- `MSG_UNORDERED`
- `MSG_ADDR_OVER`
- `MSG_ABORT`
- `MSG_EOF`
- `MSG_PR_SCTP`

`sinfo_ppid` この値は、`sendmsg()` 呼び出しのデフォルトのペイロードプロトコル識別子です。

`sinfo_context` この値は、`sendmsg()` 呼び出しのデフォルトコンテキストです。

`sinfo_timetolive` この値は、ミリ秒単位で期間を指定します。この期間を経過すると、伝送が開始されていないメッセージは期限切れになります。値が 0 の場合は、メッセージが期限切れにならないことを示します。`MSG_PR_SCTP` フラグ

が設定されている場合、sinfo_timetolive に指定された期間内に伝送が正常に完了しないメッセージは期限切れになります。

sinfo_tsn
この値は常に 0 です。

sinfo_cumtsn
この値は常に 0 です。

sinfo_assoc_id
この値は、呼び出し元アプリケーションによって設定され、対象になる関連付けを指定します。

SCTP_PEER_ADDR_PARAMS
指定されたピアのアドレスのパラメータを返します。パラメータは次の構造体を使用します。

```
struct sctp_paddrparams {
    sctp_assoc_t              spp_assoc_id;
    struct sockaddr_storage   spp_address;
    uint32_t                  spp_hbinterval;
    uint16_t                  spp_pathmaxrxt;
};
```

spp_assoc_id
対象になる関連付けを指定するこの値は、呼び出し元プログラムが提供します。

spp_address
この値は、対象になるピアのアドレスを指定します。

spp_hbinterval
この値は、ミリ秒単位でハートビート間隔を指定します。

spp_pathmaxrxt
この値は、あるアドレスを到達不能と見なすまでの、再伝送を試みる最大回数を指定します。

SCTP_STATUS
関連付けに関する現在のステータス情報を返します。パラメータは次の構造体を使用します。

```
struct sctp_status {
    sctp_assoc_t            sstat_assoc_id;
    int32_t                 sstat_state;
    uint32_t                sstat_rwnd;
    uint16_t                sstat_unackdata;
    uint16_t                sstat_penddata;
    uint16_t                sstat_instrms;
    uint16_t                sstat_outstrms;
    uint32_t                sstat_fragmentation_point;
    struct sctp_paddrinfo   sstat_primary;
};
```

`sstat_assoc_id`
対象になる関連付けを指定するこの値は、呼び出し元プログラムが提供します。

`sstat_state`
この値は、関連付けの現在の状態を示し、関連付けは次の状態を取ることができます。

`SCTP_IDLE`	SCTP 終端がいずれの関連付けとも関連付けられていません。`socket()` 関数が終端を開いた直後、または、終端が閉じられた直後、終端はこの状態になります。
`SCTP_BOUND`	`bind()` の呼び出しのあと、SCTP 終端が 1 つ以上のローカルアドレスにバインドされています。
`SCTP_LISTEN`	リモート SCTP 終端からの関連付け要求を終端が待機しています。
`SCTP_COOKIE_WAIT`	SCTP 終端が INIT チャンクを送信し、INIT-ACK チャンクを待機しています。
`SCTP_COOKIE_ECHOED`	SCTP 終端が、ピアの INIT-ACK チャンクから受信した cookie をピアにエコーバックしました。
`SCTP_ESTABLISHED`	SCTP 終端はピアとデータを交換できます。
`SCTP_SHUTDOWN_PENDING`	SCTP 終端が上位層から SHUTDOWN プリミティブを受信しました。この終端は上位層からデータを受け入れません。
`SCTP_SHUTDOWN_SEND`	SCTP_SHUTDOWN_PENDING 状態にあった SCTP 終端が SHUTDOWN チャンク

をピアに送信しました。SHUTDOWN チャンクが送信されるのは、終端からピアへの未処理データがすべて確認されたあとだけです。終端のピアが SHUTDOWN ACK チャンクを送信すると、終端は SHUTDOWN COMPLETE チャンクを送信し、関連付けが閉じられたと見なされます。

`SCTP_SHUTDOWN_RECEIVED` SCTP 終端がピアから SHUTDOWN チャンクを受信しました。この終端はそのユーザーから新しいデータを受け入れません。

`SCTP_SHUTDOWN_ACK_SEND` SCTP_SHUTDOWN_RECEIVED 状態の SCTP 終端がピアに SHUTDOWN ACK チャンクを送信しました。終端が SHUTDOWN ACK チャンクを送信するのは、その終端からのすべての未処理データをピアが確認したあとだけです。終端のピアが SHUTDOWN COMPLETE チャンクを送信すると、関連付けは閉じられます。

`sstat_rwnd`
この値は、関連付けピアの現在の受信ウィンドウです。

`sstat_unackdata`
この値は、未確認 DATA チャンクの数です。

`sstat_penddata`
この値は、受信を待機している DATA チャンクの数です。

`sstat_instrms`
この値は、着信ストリームの数です。

`sstat_outstrms`
この値は、発信ストリームの数です。

`sstat_fragmentation_point`
メッセージ、SCTP ヘッダー、および IP ヘッダーを含めたサイズが `sstat_fragmentation_point` の値を超える場合、メッセージは分割されます。この値は、パケットの着信先アドレスのパス最大伝送ユニット (P-MTU) と同じです。

`sstat_primary`
この値は、プライマリピアのアドレスに関する情報です。この情報は次の構造体を使用します。

```
struct sctp_paddrinfo {
    sctp_assoc_t              spinfo_assoc_id;
    struct sockaddr_storage   spinfo_address;
    int32_t                   spinfo_state;
    uint32_t                  spinfo_cwnd;
    uint32_t                  spinfo_srtt;
    uint32_t                  spinfo_rto;
    uint32_t                  spinfo_mtu;
};
```

`spinfo_assoc_id` 対象になる関連付けを指定するこの値は、呼び出し元プログラムが提供します。

`spinfo_address` この値は、プライマリピアのアドレスです。

`spinfo_state` この値は、`SCTP_ACTIVE` または `SCTP_INACTIVE` の 2 つの値のいずれかを取ります。

`spinfo_cwnd` この値は、ピアアドレスの輻輳ウィンドウです。

`spinfo_srtt` この値は、ピアアドレスに関する現在の平滑化された RTT の計算値です。ミリ秒単位で示されます。

`spinfo_rto` この値は、ピアアドレスに関する現在の伝送タイムアウト値です。ミリ秒単位で示されます。

`spinfo_mtu` この値は、ピアアドレスに関する P-MTU です。

sctp_opt_info() 関数は成功すると 0 を返します。失敗すると、sctp_opt_info() 関数は -1 を返し、errno の値を該当するエラーコードに設定します。sctp_opt_info() に渡される *sock* パラメータのファイル記述子が無効である場合、sctp_opt_info() 関数は失敗して EBADF を返します。sctp_opt_info() 関数に渡される *sock* パラメータのファイル記述子がソケットではない場合、sctp_opt_info() 関数は失敗して ENOTSOCK を返します。関連付け ID が 1 対多スタイル SCTP ソケットに対して無効である場合、sctp_opt_info() 関数は失敗して errno の値を EINVAL に設定します。指定されたオプションに対して入力バッファー長が短すぎる場合、sctp_opt_info() 関数は失敗して errno の値を EINVAL に設定します。ピアのアドレスのアドレス群が AF_INET または AF_INET6 ではない場合、sctp_opt_info() 関数は失敗して errno の値を EAFNOSUPPORT に設定します。

sctp_recvmsg()

ssize_t **sctp_recvmsg**(int *s*, void **msg*, size_t *len*, struct sockaddr **from*, socklen_t **fromlen*, struct sctp_sndrcvinfo **sinfo* , int **msg_flags*);

sctp_recvmsg() 関数は、*s* パラメータによって指定される SCTP 終端からのメッセージの受信を有効にします。呼び出し元プログラムによって次の属性を指定できます。

msg
このパラメータは、メッセージバッファーのアドレスです。

len
このパラメータは、メッセージバッファーの長さです。

from
このパラメータは、送信元のアドレスを含むアドレスへのポインタです。

fromlen
from パラメータのアドレスに関連付けられたバッファーのサイズです。

sinfo
このパラメータは、呼び出し元プログラムで sctp_data_io_events が有効な場合のみ有効です。sctp_data_io_events を有効にするには、ソケットオプション SCTP_EVENTS() によって setsockopt 関数を呼び出します。sctp_data_io_events が有効である場合、アプリケーションは着信メッセージごとに sctp_sndrcvinfo 構造体の内容を受信します。このパラメータは、sctp_sndrcvinfo 構造体へのポインタです。構造体はメッセージの受信時にデータを取り込みます。

msg_flags
このパラメータは、存在するメッセージフラグを含みます。

sctp_recvmsg() 関数は、受信するバイト数を返します。エラーが発生した場合、sctp_recvmsg() 関数は -1 を返します。

s パラメータの渡されたファイル記述子が有効でない場合、sctp_recvmsg() 関数は失敗して errno の値を EBADF に設定します。*s* パラメータに渡されたファイル記述子がソケットでない場合、sctp_recvmsg() 関数は失敗して errno の値を ENOTSOCK に設定します。*msg_flags* パラメータに値 MSG_OOB が含まれる場合、sctp_recvmsg() 関数は失敗して errno の値を EOPNOTSUPP に設定します。関連付けが確立していない場合、sctp_recvmsg() 関数は失敗して errno の値を ENOTCONN に設定します。

sctp_sendmsg()

ssize_t **sctp_sendmsg**(int *s*, const void **msg*, size_t *len*, const struct sockaddr **to*, socklen_t *tolen*, uint32_t *ppid*, uint32_t *flags*, uint16_t *stream_no*, uint32_t *timetolive*, uint32_t *context*);

sctp_sendmsg() 関数は、SCTP 終端からメッセージを送信する際の拡張 SCTP 機能を有効にします。

s この値は、メッセージを送信する SCTP 終端を指定します。

msg この値は、sctp_sendmsg() 関数によって送信されるメッセージです。

len この値は、メッセージの長さで、バイト単位で表されます。

to この値は、メッセージの着信先アドレスです。

tolen この値は、着信先アドレスの長さです。

ppid この値は、アプリケーション固有のペイロードプロトコル識別子です。

stream_no この値は、メッセージのターゲットストリームです。

timetolive この値は、メッセージが正常にピアに送信されなかった場合に期限切れになるまでの期間で、ミリ秒単位で示されます。

context メッセージの送信時にエラーが発生した場合に、この値が返されます。

flags この値は、0 以上の次のフラグビットに対するビット単位の論理和演算を実行することによって求められます。

MSG_UNORDERED
: このフラグが設定されている場合 sctp_sendmsg() 関数はメッセージを順不同で配信します。

MSG_ADDR_OVER
: このフラグが設定されている場合、sctp_sendmsg() 関数は関連付けのプライマリ着信先アドレスではなく、*to* パラメータのアドレスを使用します。このフラグは 1 対多スタイル SCTP ソケットの場合にのみ使用されます。

MSG_ABORT
このフラグが設定されている場合、指定された関連付けは ABORT シグナルをピアに送信して異常終了します。このフラグは 1 対多スタイル SCTP ソケットの場合にのみ使用されます。

MSG_EOF
このフラグが設定されている場合、指定された関連付けは適切に終了します。このフラグは 1 対多スタイル SCTP ソケットの場合にのみ使用されます。

MSG_PR_SCTP
このフラグが設定されている場合、timetolive パラメータに指定された期間内に伝送が正常に完了しないメッセージは期限切れになります。

sctp_sendmsg() 関数は、送信したバイト数を返します。エラーが発生した場合、sctp_sendmsg() 関数は -1 を返します。

s パラメータに渡されたファイル記述子が有効でない場合、sctp_sendmsg() 関数は失敗して errno の値を EBADF に設定します。*s* パラメータの渡されたファイル記述子がソケットでない場合、sctp_sendmsg() 関数は失敗して errno の値を ENOTSOCK に設定します。*flags* パラメータに値 MSG_OOB が含まれる場合、sctp_sendmsg() 関数は失敗して errno の値を EOPNOTSUPP に設定します。1 対 1 スタイルソケットで *flags* パラメータに値 MSG_ABORT または MSG_EOF が含まれる場合、sctp_sendmsg() 関数は失敗して errno の値を EOPNOTSUPP に設定します。関連付けが確立していない場合、sctp_sendmsg() 関数は失敗して errno の値を ENOTCONN に設定します。ソケットが停止していてそれ以上の書込みができない場合、sctp_sendmsg() 関数は失敗して errno の値を EPIPE に設定します。ソケットが非ブロックで、伝送待ち行列がいっぱいである場合、sctp_sendmsg() 関数は失敗して errno の値を EAGAIN に設定します。

制御メッセージ長が不正である場合、sctp_sendmsg() 関数は失敗して errno の値を EINVAL に設定します。指定された着信先アドレスが関連付けに属さない場合、sctp_sendmsg() 関数は失敗して errno の値を EINVAL に設定します。*stream_no* の値が関連付けによってサポートされる発信ストリームの数以外である場合、sctp_sendmsg() 関数は失敗して errno の値を EINVAL に設定します。指定された着信先アドレスのアドレス群が AF_INET または AF_INET6 でない場合、sctp_sendmsg() 関数は失敗して errno の値を EINVAL に設定します。

sctp_send()

ssize_t **sctp_send**(int *s*, const void **msg*, size_t *len*, const struct sctp_sndrcvinfo **sinfo* , int *flags*);

sctp_send() 関数は、1 対 1 スタイルソケットと 1 対多スタイルソケットで使用できます。sctp_send() 関数は、SCTP 終端からメッセージを送信する際の拡張 SCTP 機能を有効にします。

s
この値は、socket() 関数によって作成されるソケットです。

msg
この値は、sctp_send() 関数によって送信されるメッセージです。

len
この値は、メッセージの長さで、バイト単位で表されます。

sinfo
この値は、メッセージの送信のために使用されるパラメータです。1 対多スタイルソケットの場合、メッセージの送信先の関連付け ID をこの値に指定できます。

flags
この値は、sendmsg() 関数のフラグパラメータと同じです。

sctp_send() 関数は、送信したバイト数を返します。エラーが発生した場合、sctp_send() 関数は -1 を返します。

s パラメータの渡されたファイル記述子が有効でない場合、sctp_send() 関数は失敗して errno の値を EBADF に設定します。*s* パラメータの渡されたファイル記述子がソケットでない場合、sctp_send() 関数は失敗して errno の値を ENOTSOCK に設定します。*sinfo* パラメータの *sinfo_flags* フィールドに値 MSG_OOB が含まれる場合、sctp_send() 関数は失敗して errno の値を EOPNOTSUPP に設定します。1 対 1 スタイルソケットで *sinfo* パラメータの *sinfo_flags* フィールドに値 MSG_ABORT または MSG_EOF が含まれる場合、sctp_send() 関数は失敗して errno の値を EOPNOTSUPP に設定します。関連付けが確立していない場合、sctp_send() 関数は失敗して errno の値を ENOTCONN に設定します。ソケットが停止していてそれ以上の書込みができない場合、sctp_send() 関数は失敗して errno の値を EPIPE に設定します。ソケットが非ブロックで、伝送待ち行列がいっぱいである場合、sctp_send() 関数は失敗して errno の値を EAGAIN に設定します。

制御メッセージ長が不正である場合、sctp_send() 関数は失敗して errno の値を EINVAL に設定します。指定された着信先アドレスが関連付けに属さない場合、sctp_send() 関数は失敗して errno の値を EINVAL に設定します。*stream_no* の値が関連付けによってサポートされる発信ストリームの数以外である場合、sctp_send() 関数は失敗して errno の値を EINVAL に設定します。指定された着信先アドレスのアドレス群が AF_INET または AF_INET6 ではない場合、sctp_send() 関数は失敗して errno の値を EINVAL に設定します。

分岐関連付け

アプリケーションは、1 対多スタイルソケットで確立された関連付けを別のソケットとファイル記述子に分岐できます。散発的なメッセージの送信側または受信側が多数あり、元の 1 対多スタイルソケットのままである必要があるアプリケーションの場合に、別のソケットとファイル記述子を使用すると便利です。アプリケーションは、大量のデータトラフィックを伝送する関連付けを別のソケット記述子に分岐し

ます。アプリケーションは、sctp_peeloff() 呼び出しを使用して、関連付けを別のソケットに分岐します。新しいソケットは 1 対 1 スタイルソケットです。sctp_peeloff() 関数の構文は次のとおりです。

```
int sctp_peeloff(int sock, sctp_assoc_t id);
```

sock
socket() システムコールから返される元の 1 対多スタイルソケット記述子。

id
別のファイル記述子に分岐する関連付けの識別子。

sock に渡されるソケット記述子が 1 対多スタイル SCTP ソケットではない場合、sctp_peeloff() 関数は失敗して EOPTNOTSUPP を返します。id() の値が 0 である場合、または、id の値が *sock* パラメータに渡されるソケット記述子の関連付けの最大数より大きい場合、*sctp_peeloff* 関数は失敗して *EINVAL* を返します。sctp_peeloff() 関数が新しいユーザーファイル記述子またはファイル構造体を作成できない場合、関数は失敗して EMFILE を返します。

sctp_getpaddrs()

sctp_getpaddrs() 関数は、関連付け内のすべてのピアを返します。

```
int sctp_getpaddrs(int sock, sctp_assoc_t id, void **addrs);
```

sctp_getpaddrs() 関数が正常に結果を返した場合、**addrs* パラメータの値は、動的に割り当てられるパックされた配列である、各アドレスの適切なタイプの sockaddr 構造体を指します。呼び出し元スレッドは、sctp_freepaddrs() 関数によってメモリーを解放します。**addrs* パラメータに値 NULL はありません。*sock* に指定されるソケット記述子が IPv4 ソケット用である場合、sctp_getpaddrs() 関数は IPv4 アドレスを返します。*sock* に指定されるソケット記述子が IPv6 ソケット用である場合、sctp_getpaddrs() 関数は IPv4 アドレスと IPv6 アドレスを混在して返します。1 対多スタイルソケットの場合、*id* パラメータは照会する関連付けを指定します。1 対 1 スタイルソケットの場合、sctp_getpaddrs() 関数は *id* パラメータを無視します。sctp_getpaddrs() 関数が正常に結果を返す場合、関連付け内のピアアドレスの数が返されます。ソケット上に関連付けがない場合、sctp_getpaddrs() 関数は 0 を返し、**addrs* パラメータの値は未定義です。エラーが発生した場合、sctp_getpaddrs() 関数は -1 を返し、**addrs* パラメータの値は未定義です。

sctp_getpaddrs() に渡される *sock* パラメータのファイル記述子が無効である場合、sctp_getpaddrs() 関数は失敗して EBADF を返します。sctp_getpaddrs() 関数に渡される *sock* パラメータのファイル記述子がソケットでない場合、sctp_getpaddrs() 関数は失敗して ENOTSOCK を返します。sctp_getpaddrs() 関数に渡される *sock* パラメータのファイル記述子が接続されていないソケットである場合、sctp_getpaddrs() 関数は失敗して ENOTCONN を返します。

sctp_freepaddrs()

sctp_freepaddrs() 関数は、それ以前の sctp_getpaddrs() の呼び出しによって割り当てられたすべてのリソースを解放します。sctp_freepaddrs() 関数の構文は次のとおりです。

```
void sctp_freepaddrs(void *addrs);
```

*addrs パラメータは、sctp_getpaddrs() 関数によって返されたピアのアドレスを含む配列です。

sctp_getladdrs()

sctp_getladdrs() 関数は、ソケット上のローカルでバインドされているすべてのアドレスを返します。sctp_getladdrs() 関数の構文は次のとおりです。

```
int sctp_getladdrs(int sock, sctp_assoc_t id, void **addrs);
```

sctp_getladdrs() 関数が正常に結果を返した場合、*addrs* の値は動的に割り当てられるパックされた配列の *sockaddr* 構造体を指します。*sockaddr* 構造体は各ローカルアドレスの適切なタイプです。呼び出し元アプリケーションは、sctp_freeladdrs() 関数を使用してメモリーを解放します。*addrs* パラメータの値が NULL であってはなりません。

sd パラメータによって参照されるソケットが IPv4 ソケットである場合、sctp_getladdrs() 関数は IPv4 アドレスを返します。*sd* パラメータによって参照されるソケットが IPv6 ソケットである場合、sctp_getladdrs() 関数は必要に応じて IPv4 アドレスと IPv6 アドレスを混在して返します。

1 対多スタイルソケットで sctp_getladdrs() 関数が起動されると、*id* パラメータは照会する関連付けを指定します。1 対 1 ソケットで sctp_getladdrs() 関数で動作する場合、*id* パラメータは無視されます。

id パラメータの値が 0 である場合、sctp_getladdrs() 関数は特定の関連付けに関係なくローカルでバインドされているアドレスを返します。sctp_getladdrs() 関数が正常に結果を返す場合、ソケットにバインドされているローカルアドレスの数を報告します。ソケットがバインドされていない場合、sctp_getladdrs() 関数は 0 を返し、**addrs* の値は未定義です。エラーが発生した場合、sctp_getladdrs() 関数は -1 を返し、**addrs* パラメータの値は未定義です。

sctp_freeladdrs()

sctp_freeladdrs() 関数は、それ以前の sctp_getladdrs() の呼び出しによって割り当てられたすべてのリソースを解放します。sctp_freeladdrs() 関数の構文は次のとおりです。

```
void sctp_freeladdrs(void *addrs);
```

*addrs パラメータは、sctp_getladdrs() 関数によって返されたピアのアドレスを含む配列です。

SCTP を使用したコーディング例

このセクションでは、SCTP ソケットの 2 つの使用法を示します。

例 8–17 SCTP エコークライアント

```
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/* To enable socket features used for SCTP socket. */
#define _XPG4_2
#define __EXTENSIONS__

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <netinet/sctp.h>
#include <netdb.h>

#define BUFLEN 1024

static void
usage(char *a0)
{
    fprintf(stderr, "Usage: %s <addr>\n", a0);
}

/*
 * Read from the network.
 */
static void
readit(void *vfdp)
{
    int   fd;
    ssize_t   n;
    char   buf[BUFLEN];
    struct msghdr  msg[1];
    struct iovec  iov[1];
    struct cmsghdr  *cmsg;
    struct sctp_sndrcvinfo *sri;
    char   cbuf[sizeof (*cmsg) + sizeof (*sri)];
    union sctp_notification *snp;

    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

    fd = *(int *)vfdp;

    /* Initialize the message header for receiving */
    memset(msg, 0, sizeof (*msg));
    msg->msg_control = cbuf;
    msg->msg_controllen = sizeof (*cmsg) + sizeof (*sri);
```

例 8–17 SCTP エコークライアント (続き)

```
    msg->msg_flags = 0;
    cmsg = (struct cmsghdr *)cbuf;
    sri = (struct sctp_sndrcvinfo *)(cmsg + 1);
    iov->iov_base = buf;
    iov->iov_len = BUFLEN;
    msg->msg_iov = iov;
    msg->msg_iovlen = 1;

    while ((n = recvmsg(fd, msg, 0)) > 0) {
        /* Intercept notifications here */
        if (msg->msg_flags & MSG_NOTIFICATION) {
            snp = (union sctp_notification *)buf;
            printf("[ Receive notification type %u ]\n",
                snp->sn_type);
            continue;
        }
        msg->msg_control = cbuf;
        msg->msg_controllen = sizeof (*cmsg) + sizeof (*sri);
        printf("[ Receive echo (%u bytes): stream = %hu, ssn = %hu, "
            "flags = %hx, ppid = %u ]\n", n,
            sri->sinfo_stream, sri->sinfo_ssn, sri->sinfo_flags,
            sri->sinfo_ppid);
    }

    if (n < 0) {
        perror("recv");
        exit(1);
    }

    close(fd);
    exit(0);
}

#define MAX_STREAM 64

static void
echo(struct sockaddr_in *addr)
{
    int     fd;
    uchar_t     buf[BUFLEN];
    ssize_t    n;
    int    perr;
    pthread_t   tid;
    struct cmsghdr   *cmsg;
    struct sctp_sndrcvinfo  *sri;
    char    cbuf[sizeof (*cmsg) + sizeof (*sri)];
    struct msghdr   msg[1];
    struct iovec   iov[1];
    int    ret;
    struct sctp_initmsg   initmsg;
    struct sctp_event_subscribe events;

    /* Create a one-one SCTP socket */
    if ((fd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) == -1) {
        perror("socket");
        exit(1);
```

例 8–17 SCTP エコークライアント (続き)

```
}

/*
 * We are interested in association change events and we want
 * to get sctp_sndrcvinfo in each receive.
 */
events.sctp_association_event = 1;
events.sctp_data_io_event = 1;
ret = setsockopt(fd, IPPROTO_SCTP, SCTP_EVENTS, &events,
    sizeof (events));
if (ret < 0) {
    perror("setsockopt SCTP_EVENTS");
    exit(1);
}

/*
 * Set the SCTP stream parameters to tell the other side when
 * setting up the association.
 */
memset(&initmsg, 0, sizeof(struct sctp_initmsg));
initmsg.sinit_num_ostreams = MAX_STREAM;
initmsg.sinit_max_instreams = MAX_STREAM;
initmsg.sinit_max_attempts = MAX_STREAM;
ret = setsockopt(fd, IPPROTO_SCTP, SCTP_INITMSG, &initmsg,
    sizeof(struct sctp_initmsg));
if (ret < 0) {
    perror("setsockopt SCTP_INITMSG");
    exit(1);
}

if (connect(fd, (struct sockaddr *)addr, sizeof (*addr)) == -1) {
    perror("connect");
    exit(1);
}

/* Initialize the message header structure for sending. */
memset(msg, 0, sizeof (*msg));
iov->iov_base = buf;
msg->msg_iov = iov;
msg->msg_iovlen = 1;
msg->msg_control = cbuf;
msg->msg_controllen = sizeof (*cmsg) + sizeof (*sri);
msg->msg_flags |= MSG_XPG4_2;

memset(cbuf, 0, sizeof (*cmsg) + sizeof (*sri));
cmsg = (struct cmsghdr *)cbuf;
sri = (struct sctp_sndrcvinfo *)(cmsg + 1);

cmsg->cmsg_len = sizeof (*cmsg) + sizeof (*sri);
cmsg->cmsg_level = IPPROTO_SCTP;
cmsg->cmsg_type  = SCTP_SNDRCV;

sri->sinfo_ppid   = 1;
/* Start sending to stream 0. */
sri->sinfo_stream = 0;
```

例 8–17 SCTP エコークライアント (続き)

```
    /* Create a thread to receive network traffic. */
    perr = pthread_create(&tid, NULL, (void *(*)(void *))readit, &fd);

    if (perr != 0) {
        fprintf(stderr, "pthread_create: %d\n", perr);
        exit(1);
    }

    /* Read from stdin and then send to the echo server. */
    while ((n = read(fileno(stdin), buf, BUFLEN)) > 0) {
        iov->iov_len = n;
        if (sendmsg(fd, msg, 0) < 0) {
            perror("sendmsg");
            exit(1);
        }
        /* Send the next message to a different stream. */
        sri->sinfo_stream = (sri->sinfo_stream + 1) % MAX_STREAM;
    }

    pthread_cancel(tid);
    close(fd);
}

int
main(int argc, char **argv)
{
    struct sockaddr_in addr[1];
    struct hostent *hp;
    int error;

    if (argc < 2) {
        usage(*argv);
        exit(1);
    }

    /* Find the host to connect to. */
    hp = getipnodebyname(argv[1], AF_INET, AI_DEFAULT, &error);
    if (hp == NULL) {
        fprintf(stderr, "host not found\n");
        exit(1);
    }

    addr->sin_family = AF_INET;
    addr->sin_addr.s_addr = *(ipaddr_t *)hp->h_addr_list[0];
    addr->sin_port = htons(5000);

    echo(addr);

    return (0);
}
```

例 8–18 SCTP エコーサーバー

```
/*
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
```

例 8–18 SCTP エコーサーバー (続き)

```
 */

/* To enable socket features used for SCTP socket. */
#define _XPG4_2
#define __EXTENSIONS__

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>

#define BUFLEN 1024

/*
 * Given an event notification, print out what it is.
 */
static void
handle_event(void *buf)
{
    struct sctp_assoc_change *sac;
    struct sctp_send_failed  *ssf;
    struct sctp_paddr_change *spc;
    struct sctp_remote_error *sre;
    union sctp_notification  *snp;
    char    addrbuf[INET6_ADDRSTRLEN];
    const char   *ap;
    struct sockaddr_in  *sin;
    struct sockaddr_in6  *sin6;

    snp = buf;

    switch (snp->sn_header.sn_type) {
    case SCTP_ASSOC_CHANGE:
        sac = &snp->sn_assoc_change;
        printf("^^^ assoc_change: state=%hu, error=%hu, instr=%hu "
            "outstr=%hu\n", sac->sac_state, sac->sac_error,
            sac->sac_inbound_streams, sac->sac_outbound_streams);
        break;
    case SCTP_SEND_FAILED:
        ssf = &snp->sn_send_failed;
        printf("^^^ sendfailed: len=%hu err=%d\n", ssf->ssf_length,
            ssf->ssf_error);
        break;
    case SCTP_PEER_ADDR_CHANGE:
        spc = &snp->sn_paddr_change;
        if (spc->spc_aaddr.ss_family == AF_INET) {
            sin = (struct sockaddr_in *)&spc->spc_aaddr;
            ap = inet_ntop(AF_INET, &sin->sin_addr, addrbuf,
                INET6_ADDRSTRLEN);
        } else {
            sin6 = (struct sockaddr_in6 *)&spc->spc_aaddr;
```

例 8–18 SCTP エコーサーバー (続き)

```
            ap = inet_ntop(AF_INET6, &sin6->sin6_addr, addrbuf,
                INET6_ADDRSTRLEN);
        }
        printf("^^^ intf_change: %s state=%d, error=%d\n", ap,
            spc->spc_state, spc->spc_error);
        break;
    case SCTP_REMOTE_ERROR:
        sre = &snp->sn_remote_error;
        printf("^^^ remote_error: err=%hu len=%hu\n",
            ntohs(sre->sre_error), ntohs(sre->sre_length));
        break;
    case SCTP_SHUTDOWN_EVENT:
        printf("^^^ shutdown event\n");
        break;
    default:
        printf("unknown type: %hu\n", snp->sn_header.sn_type);
        break;
    }
}

/*
 * Receive a message from the network.
 */
static void *
getmsg(int fd, struct msghdr *msg, void *buf, size_t *buflen,
    ssize_t *nrp, size_t cmsglen)
{
    ssize_t  nr = 0;
    struct iovec iov[1];

    *nrp = 0;
    iov->iov_base = buf;
    msg->msg_iov = iov;
    msg->msg_iovlen = 1;

    /* Loop until a whole message is received. */
    for (;;) {
        msg->msg_flags = MSG_XPG4_2;
        msg->msg_iov->iov_len = *buflen;
        msg->msg_controllen = cmsglen;

        nr += recvmsg(fd, msg, 0);
        if (nr <= 0) {
            /* EOF or error */
            *nrp = nr;
            return (NULL);
        }

        /* Whole message is received, return it. */
        if (msg->msg_flags & MSG_EOR) {
            *nrp = nr;
            return (buf);
        }

        /* Maybe we need a bigger buffer, do realloc(). */
        if (*buflen == nr) {
```

例 8–18 SCTP エコーサーバー (続き)

```
            buf = realloc(buf, *buflen * 2);
            if (buf == 0) {
                fprintf(stderr, "out of memory\n");
                exit(1);
            }
            *buflen *= 2;
        }

        /* Set the next read offset */
        iov->iov_base = (char *)buf + nr;
        iov->iov_len = *buflen - nr;

    }
}

/*
 * The echo server.
 */
static void
echo(int fd)
{
    ssize_t   nr;
    struct sctp_sndrcvinfo *sri;
    struct msghdr  msg[1];
    struct cmsghdr  *cmsg;
    char   cbuf[sizeof (*cmsg) + sizeof (*sri)];
    char   *buf;
    size_t   buflen;
    struct iovec  iov[1];
    size_t   cmsglen = sizeof (*cmsg) + sizeof (*sri);

    /* Allocate the initial data buffer */
    buflen = BUFLEN;
    if ((buf = malloc(BUFLEN)) == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }

    /* Set up the msghdr structure for receiving */
    memset(msg, 0, sizeof (*msg));
    msg->msg_control = cbuf;
    msg->msg_controllen = cmsglen;
    msg->msg_flags = 0;
    cmsg = (struct cmsghdr *)cbuf;
    sri = (struct sctp_sndrcvinfo *)(cmsg + 1);

    /* Wait for something to echo */
    while ((buf = getmsg(fd, msg, buf, &buflen, &nr, cmsglen)) != NULL) {

        /* Intercept notifications here */
        if (msg->msg_flags & MSG_NOTIFICATION) {
            handle_event(buf);
            continue;
        }

        iov->iov_base = buf;
```

例 8–18 SCTP エコーサーバー (続き)

```
        msg->msg_iov = iov;
        msg->msg_iovlen = 1;
        iov->iov_len = nr;
        msg->msg_control = cbuf;
        msg->msg_controllen = sizeof (*cmsg) + sizeof (*sri);

        printf("got %u bytes on stream %hu:\n", nr,
               sri->sinfo_stream);
        write(0, buf, nr);

        /* Echo it back */
        msg->msg_flags = MSG_XPG4_2;
        if (sendmsg(fd, msg, 0) < 0) {
            perror("sendmsg");
            exit(1);
        }
    }

    if (nr < 0) {
        perror("recvmsg");
    }
    close(fd);
}

int
main(void)
{
    int    lfd;
    int    cfd;
    int    onoff = 1;
    struct sockaddr_in  sin[1];
    struct sctp_event_subscribe events;
    struct sctp_initmsg  initmsg;

    if ((lfd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) == -1) {
        perror("socket");
        exit(1);
    }

    sin->sin_family = AF_INET;
    sin->sin_port = htons(5000);
    sin->sin_addr.s_addr = INADDR_ANY;
    if (bind(lfd, (struct sockaddr *)sin, sizeof (*sin)) == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(lfd, 1) == -1) {
        perror("listen");
        exit(1);
    }

    (void) memset(&initmsg, 0, sizeof(struct sctp_initmsg));
    initmsg.sinit_num_ostreams = 64;
    initmsg.sinit_max_instreams = 64;
    initmsg.sinit_max_attempts = 64;
```

例 8–18 SCTP エコーサーバー (続き)

```
    if (setsockopt(lfd, IPPROTO_SCTP, SCTP_INITMSG, &initmsg,
           sizeof(struct sctp_initmsg)) < 0) {
        perror("SCTP_INITMSG");
        exit (1);
    }

    /* Events to be notified for */
    (void) memset(&events, 0, sizeof (events));
    events.sctp_data_io_event = 1;
    events.sctp_association_event = 1;
    events.sctp_send_failure_event = 1;
    events.sctp_address_event = 1;
    events.sctp_peer_error_event = 1;
    events.sctp_shutdown_event = 1;

    /* Wait for new associations */
    for (;;) {
        if ((cfd = accept(lfd, NULL, 0)) == -1) {
            perror("accept");
            exit(1);
        }

        /* Enable ancillary data */
        if (setsockopt(cfd, IPPROTO_SCTP, SCTP_EVENTS, &events,
               sizeof (events)) < 0) {
            perror("setsockopt SCTP_EVENTS");
            exit(1);
        }
        /* Echo back any and all data */
        echo(cfd);
    }
}
```

◆ ◆ ◆ 第 9 章

9 XTI と TLI を使用したプログラミング

この章では、トランスポートレベルインタフェース (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、220 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、最近 XTI に追加された機能については、241 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ー (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、スケーラビリティー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仕様化されていません。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 1986 年に AT&T の System V, Release 3 によって導入されました。TLI はトランスポートレベルインタフェース 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性を表しています。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 (`libnsl`) 内のインタフェース呼び出しセットとして実装され、それに対して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。

第 4 章で説明している追加のインタフェースとメカニズムを組み合わせて使用することで、XTI/TLI コードを現在のトランスポートプロバイダから独立させることができます。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 つのコンポーネント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 11 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」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、231 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。XTI/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 読み取り用インタフェースと書き込み用インタフェース

ユーザーは、コネクションを介して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することを望む場合があります。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 9–1　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>

/* Same local management and connection establishment steps. */

if (ioctl(fd, I_PUSH, "tirdwr") == −1) {
```

例 9–1 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)

```
    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 をトランスポートエンドポイント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/書き込みインタフェースを呼び出します。詳細は、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) のセマンティクスを使用するようになります。 read および write のセマンティクスを使用するとき、XTI/TLI はメッセージを保持しません。トランスポートプロバイダから tirdwr をポップすると、XTI/TLI は本来のセマンティクスに戻ります (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 XTI/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 を発生させ、使用不可になったことを通知します。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などです。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 を発生させます。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あと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。

- 220 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。
- 221 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) は通常接続要求のブロックを行います。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 または O_NONBLOCK を使用することによって各 XTI/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 9–2 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 XTI/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、あるサーバーが 1 つのコネクション要求を処理する間、他のクライアントからはそのサーバーがビジーであるように見えます。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 **9–2** 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                 /* server's well known address */

int conn_fd;                       /* server connection here */
extern int t_errno;
/* holds connect requests */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
   struct pollfd pollfds[NUM_FDS];
   struct t_bind *bind;
   int i;
```

例 9–2 終端の確立 (複数コネクションへ変更可能) (続き)

```
   /*
    * Only opening and binding one transport endpoint, but more can
    * be supported
    */
   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);
   }
   /* Was the correct address bound? */
   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 9–2 を少し変更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 9–3 コネクションリクエストの処理

```
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。他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、あとでそれを処理します。

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 9–4 イベント処理ルーチン

```
do_event( slot, fd)
int slot;
int fd;
{
   struct t_discon *discon;
   int i;
```

例 9–4 イベント処理ルーチン *(続き)*

```
   switch (t_look(fd)) {
   default:
      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:
      /* since POLLIN returned, this should not happen */
      fprintf(stderr,"t_look returned no event\n");
      exit(10);
   case T_LISTEN:
      /* find free element in calls array */
      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);
      }
      /* find call ind in array and delete it */
      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 9–4 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、*fd* により指定された終端上の XTI/TLI イベントの識別を行います。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 9–5 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);
      }
      if (t_bind(conn_fd, (struct t_bind *) NULL,
            (struct t_bind *) NULL) == −1) {
         t_error("t_bind failed");
```

例 9–5 すべてのコネクション要求の処理 (続き)

```
            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て直ちに service_conn_ind が返され、現在のコネクション要求は保存されたあとで処理されます。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

このセクションでは、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の技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。送信リクエストは 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他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されます。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。これにより、使用可能なバッファー領域がすぐに確保できない場合、t_sndudata(3NSL) への呼び出しは -1 を返し、t_errno を TFLOW に設定します。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 を TNODATA に設定します。

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);
    ...     /* set up binding address */
    t_bind(fd, bind, bin

    /* make endpoint non-blocking */
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);

    /* establish signal handler for SIGPOLL */
    signal(SIGPOLL, sigpoll);

    /* request SIGPOLL signal when receive data is available */
    ioctl(fd, I_SETSIG, S_INPUT | S_HIPRI);

    ...

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

    for (;;) {
        ... /* initialize ud */
        if (t_rcvudata(fd, &ud, &flags) < 0) {
            if (t_errno == TNODATA)
                break;  /* no more messages */
            ... /* process other error conditions */
    }
    ... /* process message in ud */
}
```

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 = /* establish a non-blocking endpoint */

call = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR);
/* initialize call structure */
t_connect(fd, call, call);

/* connection request is now proceeding asynchronously */

/* receive indication that connection has been accepted */
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 = /* establish non-blocking endpoint */

/*receive indication that connection request has arrived */
call = (struct t_call *) t_alloc(fd, T_CALL, T_ALL);
t_listen(fd, &call);

/* determine whether or not to accept connection */
res_fd = /* establish non-blocking endpoint for response */
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の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他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ステータス情報を表示します。

例 9–6 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)
{
    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);
```

例 9–6 ファイル記述子の転送 (続き)

```
    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

次のセクションの表は、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 9–1 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
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ステータスに対応しており、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 9–2 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
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他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 9–3 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
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) 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、他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 9–4 コネクション確立時の状態

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、コネクションモードにおけるデータの転送の状態を示します。

表 9–5 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
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 9–6 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			

表 9–6 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
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	
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 9–7 コネクションレスモードでの状態

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)` 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。
- `t_connect(3NSL)` や `t_snddis(3NSL)` などの接続要求や切断要求を使用してユーザーデータを送信してはなりません。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容量が必要です。`t_alloc(3NSL)` の `T_ALL` 引数を使用して、最大バッファーサイズを設定し、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します。
- 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り込むことが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は、第 11 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」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)` 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は、第 11 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」を参照してください。
- 複数のプロトコル環境で実行する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。
- 不透明なアドレス(Opaque address) – 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。

ソケットと XTI/TLI 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 9–8　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	コメント
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) への呼び出しで指定する
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
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)	

表 9–8 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	コメント
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着信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)	-	
t_sndudata(3NSL)	sendto(3SOCKET)	
	recvmsg(3SOCKET)	
t_rcvuderr(3NSL)	-	

表 9–8 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	コメント
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 (UNIX03) 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 ユーティリティーインタフェース 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、正常型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 10 章

10 パケットフィルタリングフック

パケットフィルタリングフックインタフェースにより、セキュリティ (パケットフィルタリングやファイアウォール) ソリューション、ネットワークアドレス変換 (Network Address Translation、NAT) ソリューションなど、カーネルレベルの付加価値ネットワークソリューションの開発が容易になります。

パケットフィルタリングフックインタフェースは、次の機能を提供します。

- フックポイントのいずれかにパケットが到達するたびの通知
- IP の排他インスタンスを必要とする新しいゾーンブートをサポートするために IP の新しいインスタンスが作成されるたびの通知
- インタフェースの名前やアドレスなど、その他の基本的なネットワークインタフェース情報へのカーネルのアクセス
- ループバックインタフェースでのパケット傍受

ループバックパケット傍受では、IP の共有インスタンスを使用しているゾーン間でパケットが移動するときに、それらのパケットにアクセスすることもできます。これはデフォルトのモデルです。

パケットフィルタリングフックインタフェース

パケットフィルタリングフックインタフェースには、カーネル関数とデータ型の定義が含まれます。

パケットフィルタリングフックのカーネル関数

パケットフィルタリングフックのカーネル関数は、パケットフィルタリングをサポートする `misc/neti` カーネルモジュールおよび `misc/hook` カーネルモジュールから

エクスポートされます。これらの関数を使用するためには、カーネルモジュールを -Nmisc/neti と -Nmisc/hook にリンクして、関数がカーネルによって正しく読み込まれるようにします。

hook_alloc(9F)	hook_t データ構造体を割り当てます。
hook_free(9F)	hook_alloc によって最初に割り当てられた hook_t() 構造体を解放します。
net_event_notify_register(9F)	指定されたイベントに対する変更が発生したときに呼び出される関数を登録します。
net_event_notify_unregister(9F)	指定されたコールバック関数の呼び出しによる、指定されたイベントに対する変更の通知をこれ以上受け取らないことを示します。
net_getifname(9F)	指定のネットワークインタフェースに指定された名前を取得します。
net_getlifaddr(9F)	指定された各論理インタフェースのネットワークアドレス情報を取得します。
net_getmtu(9F)	指定されたネットワークインタフェースの現在の MTU に関する情報を取得します。
net_getpmtuenabled(9F)	指定されたネットワークプロトコルに対してパス MTU (Path MTU、PMTU) 検出が有効かどうかを示します。
net_hook_register(9F)	指定されたネットワークプロトコルに属するイベントにコールバックを登録できるようにするフックを追加します。
net_hook_unregister(9F)	net_hook_register() によって登録されたコールバックフックを無効にします。
net_inject(9F)	ネットワーク層のパケットをカーネルまたはネットワークに配信します。
net_inject_alloc(9F)	net_inject_t 構造体を割り当てます。
net_inject_free(9F)	net_inject_alloc によって最初に割り当てられた net_inject_t() 構造体を解放します。
net_instance_alloc(9F)	net_instance_t 構造体を割り当てます。
net_instance_free(9F)	net_instance_alloc によって最初に割り当てられた net_instance_t() 構造体を解放します。

net_instance_notify_register(9F)	指定のネットワークインスタンスに対して新しいインスタンスが追加または削除されたときに呼び出される指定の関数を登録します。
net_instance_notify_unregister(9F)	指定されたコールバック関数の呼び出しによる、指定されたインスタンスに対する変更の通知をこれ以上受け取らないことを示します。
net_instance_register(9F)	IP インスタンスの保守に関連するイベントが発生するときに呼び出される関数のセットを記録します。
net_instance_unregister(9F)	net_instance_register() によって以前に登録されたインスタンスのセットを削除します。
net_ispartialchecksum(9F)	指定されたパケットに、部分チェックサム値のみを持つヘッダーが含まれるかどうかを示します。
net_isvalidchecksum(9F)	指定されたパケットのレイヤー 3 チェックサム、および場合によってはレイヤー 4 チェックサムを検査します。
net_kstat_create(9F)	IP の指定されたインスタンスの新しい kstat(9S) 構造体を割り当てて初期化します。
net_kstat_delete(9F)	IP の指定されたインスタンスの kstat をシステムから削除します。
net_lifgetnext(9F)	物理ネットワークインタフェースに関連付けられているすべての論理インタフェースを検索します。
net_phygetnext(9F)	ネットワークプロトコルが「所有」するすべてのネットワークインタフェースを検索します。
net_phylookup(9F)	ネットワークプロトコルの指定されたインタフェース名の取得を試行します。
net_protocol_lookup(9F)	ネットワーク層プロトコルの実装を検出します。

net_protocol_notify_register(9F)	指定のプロトコルに対する変更が発生するときに呼び出される指定の関数を登録します。
net_protocol_notify_unregister(9F)	呼び出す関数のリストから、指定された関数を削除します。
net_protocol_release(9F)	指定されたネットワークプロトコルへの参照が必要なくなったことを示します。
net_routeto(9F)	送信されるネットワークインタフェースパケットを示します。

パケットフィルタリングフックのデータ型

次の型が前述の関数をサポートしています。

hook_t(9S)	ネットワークイベントに挿入されるコールバック。
hook_nic_event(9S)	発生した、ネットワークインタフェースに属するイベント。
hook_pkt_event(9S)	フックに渡されるパケットイベント構造体。
net_inject_t(9S)	パケットの転送方法に関する情報。
net_instance_t(9S)	関連イベントが IP 内で発生するときに呼び出されるインスタンスのコレクション。

パケットフィルタリングフックインタフェースの使用

この API では IP スタックの複数のインスタンスを同じカーネルで同時に実行できるので、パケットフィルタリングフックインタフェースを操作するためには、ある程度の量のプログラミングが必要になります。IP スタックでは、ゾーンに対してその IP スタック自体の複数のインスタンスを使用でき、フレームワークの複数のインスタンスは、IP でのパケット傍受をサポートしています。

このセクションでは、パケットフィルタリングフック API を使用してインバウンド IPv4 パケットを受信するコード例を示します。

IP インスタンス

この API を使用する場合は、まず、IP の複数のインスタンスを 1 つのカーネルで実行できるようにするか、大域ゾーンとやりとりするだけにするかを決定する必要があります。

IP インスタンスの存在がわかるように、インスタンスの作成、破棄、および終了時に起動されるコールバック関数を登録します。これら 3 つの関数ポインタを格納する net_instance_t パケットイベント構造体を割り当てるには、net_instance_alloc() を使用します。コールバックや構造体が必要なくなったときにリソースを解放するには、net_instance_free() を使用します。構造体のインスタンスに名前を指定するには、nin_name を指定します。少なくとも、nin_create() コールバックと nin_destroy() コールバックを指定します。IP の新しいインスタンスが作成されると nin_create() 関数が呼び出され、IP のインスタンスが破棄されると nin_destroy() 関数が呼び出されます。

nin_shutdown() の指定は、kstat に情報をエクスポートする場合を除いてオプションです。インスタンス単位で kstat を使用するには、作成コールバック時に net_kstat_create() を使用します。kstat 情報のクリーンアップは、破棄コールバックではなく、終了コールバック時に行います。kstat 情報をクリーンアップするには、net_kstat_delete() を使用します。

```
extern void *mycreate(const netid_t);

net_instance_t *n;

n = net_instance_alloc(NETINFO_VERSION);
if (n != NULL) {
    n->nin_create = mycreate;
    n->nin_destroy = mydestroy;
    n->nin_name = "my module";
    if (net_instance_register(n) != 0)
        net_instance_free(n);
}
```

net_instance_alloc() が呼び出される場合に IP のインスタンスが 1 つ以上あるときは、現在有効なインスタンスごとに作成コールバックが呼び出されます。コールバックをサポートするこのフレームワークでは、特定のインスタンスに対して一度に有効になるのは、作成関数、破棄関数、または終了関数のいずれか 1 つだけです。また、作成コールバックが呼び出されると、作成コールバックが完了するまで終了コールバックは呼び出されません。同様に、破棄コールバックは、終了コールバックが完了するまで開始されません。

次の例の mycreate() 関数は、作成コールバックの簡単な例です。mycreate() 関数は、ネットワークインスタンス識別子を独自の非公開のコンテキスト構造体に記録し、新しいプロトコル (IPv4 や IPv6 など) がこのフレームワークに登録されるときに呼び出される新しいコールバックを登録します。

ゾーンが実行されていないために大域ゾーン以外のインスタンスがない場合、net_instance_register() を呼び出すと、大域ゾーンに対して作成コールバックが実行されます。あとで net_instance_unregister() が呼び出されるように、破棄コールバックを指定してください。nin_create() フィールドまたは nin_destroy フィールドを NULL に設定して net_instance_register を呼び出すと、失敗します。

```
void *
mycreate(const netid_t id)
{
    mytype_t *ctx;

    ctx = kmem_alloc(sizeof(*ctx), KM_SLEEP);
    ctx->instance_id = id;
    net_instance_notify_register(id, mynewproto, ctx);
    return (ctx);
}
```

mynewproto() 関数は、ネットワークインスタンスに対してネットワークプロトコルが追加または削除されるたびに呼び出されることになります。登録したネットワークプロトコルが特定のインスタンス内ですでに動作中の場合、作成コールバックは、すでに存在するプロトコルごとに呼び出されます。

プロトコルの登録

このコールバックでは、proto 引数のみ呼び出し元によって指定されます。この時点では、有意なイベント名もフック名も指定できません。この例の関数では、IPv4 プロトコルの登録を通知するイベントのみ検索されます。

この関数では、次に、net_protocol_notify_register() インタフェースを使用して mynewevent() 関数を登録することによって IPv4 プロトコルにイベントが追加されたことを検出します。

```
static int
mynewproto(hook_notify_cmd_t cmd, void *arg, const char *proto,
    const char *event, const char *hook)
{
    mytype_t *ctx = arg;

    if (strcmp(proto, NHF_INET) != 0)
        return (0);

    switch (cmd) {
        case HN_REGISTER :
            ctx->inet = net_protocol_lookup(s->id, proto);
            net_protocol_notify_register(s->inet, mynewevent, ctx);
            break;
        case HN_UNREGISTER :
        case HN_NONE :
            break;
    }
    return (0);
}
```

次の表に、mynewproto() コールバックで使用されることがある 3 つのプロトコルを示します。今後、新しいプロトコルが追加される可能性があるので、不明なプロトコルは確実にエラー (戻り値 0) にしてください。

プログラミング記号	プロトコル
NHF_INET	IPv4
NHF_INET6	IPv6
NHF_ARP	ARP

イベントの登録

インスタンスやプロトコルの処理が動的であるのと同様に、各プロトコル下で行われるイベントの処理も動的です。この API では、ネットワークインタフェースイベントとパケットイベントの 2 種類のイベントがサポートされています。

次の関数では、IPv4 のインバウンドパケットのイベントが存在することの通知についてチェックされます。通知があると、`hook_t` 構造体が割り当てられ、インバウンド IPv4 パケットごとに呼び出される関数が記述されます。

```
static int
mynewevent(hook_notify_cmd_t cmd, void *arg, const char *parent,
    const char *event, const char *hook)
{
    mytype_t *ctx = arg;
    char buffer[32];
    hook_t *h;

    if ((strcmp(event, NH_PHYSICAL_IN) == 0) &&
        (strcmp(parent, NHF_INET) == 0)) {
            sprintf(buffer, "mypkthook_%s_%s", parent, event);
            h = hook_alloc(HOOK_VERSION);
            h->h_hint = HH_NONE;
            h->h_arg = s;
            h->h_name = strdup(buffer);
            h->h_func = mypkthook;
            s->hook_in = h;
            net_hook_register(ctx->inet, (char *)event, h);
    } else {
            h = NULL;
    }
    return (0);
}
```

`mynewevent()` 関数は、追加および削除されるイベントごとに呼び出されます。次のイベントを使用できます。

イベント名	データ構造体	コメント
NH_PHYSICAL_IN	hook_pkt_event_t	このイベントは、ネットワークプロトコルに到達し、ネットワークインタフェースドライバから受信されたパケットごとに生成されます。

イベント名	データ構造体	コメント
NH_PHYSICAL_OUT	hook_pkt_event_t	このイベントは、ネットワークプロトコル層から送信するために、ネットワークインタフェースドライバに配信する前に、パケットごとに生成されます。
NH_FORWARDING	hook_pkt_event_t	このイベントは、システムで受信されて別のネットワークインタフェースに送信されるすべてのパケットに対して生成されます。このイベントが発生するのは NH_PHYSICAL_IN の後と NH_PHYSICAL_OUT の前です。
NH_LOOPBACK_IN	hook_pkt_event_t	このイベントは、ループバックインタフェースで受信されるパケット、またはネットワークインスタンスを大域ゾーンと共有しているゾーンで受信されるパケットに対して生成されます。
NH_LOOPBACK_OUT	hook_pkt_event_t	このイベントは、ループバックインタフェースで送信されるパケット、またはネットワークインスタンスを大域ゾーンと共有しているゾーンで送信されているパケットに対して生成されます。
NH_NIC_EVENTS	hook_nic_event_t	このイベントは、ネットワークインタフェースの状態の特定の変更に対して生成されます。

パケットイベントの場合、IP スタックの特定のポイントごとに固有のイベントが 1 つあります。これは、パケットのフローにおいてパケットを傍受する正確な位置を選択できるようにするためであり、カーネル内で発生するすべてのパケットイベントを検査して過負荷状態になるのを避けることができます。ネットワークインタフェースイベントの場合、モデルは異なります。これは、1 つにはイベントの量が少ないためであり、また、必要とされるイベントが 1 つではなく複数であることが多いためです。

ネットワークインタフェースイベントは、次のイベントのいずれかを通知します。

- インタフェースが作成 (NE_PLUMB) または破棄 (NE_UNPLUMB) されます。
- インタフェースの状態がアップ (NE_UP) またはダウン (NE_DOWN) に変更します。
- インタフェースにアドレスの変更 (NE_ADDRESS_CHANGE) があります。

今後、新しいネットワークインタフェースイベントが追加される可能性があるので、コールバック関数が受信した不明なイベントや認識できないイベントに対しては、常に 0 を返してください。

パケットフック

パケットフック関数は、パケットが受信されると呼び出されます。この場合、mypkthook() 関数は、物理ネットワークインタフェースからカーネルに受信するインバウンドパケットごとに呼び出されることになります。共有 IP インスタンスモデルを使用するゾーン間またはループバックインタフェース上を流れる、内部的に生成されたパケットは、対象になりません。

パケットを受け取ることと、パケットのドロップに必要なものを関数が正常に返すようにすることとの違いを示すために、次のコードでは、パケット 100 個ごとの発信元アドレスと宛先アドレスを出力し、パケットをドロップして、パケットロスを 1% にします。

```
static int
mypkthook(hook_event_token_t tok, hook_data_t data, void *arg)
{
    static int counter = 0;
    mytupe_t *ctx = arg;
    hook_pkt_event_t *pkt = (hook_pkt_event_t)data;
    struct ip *ip;
    size_t bytes;

    bytes = msgdsize(pkt->hpe_mb);

    ip = (struct ip *)pkt->hpe_hdr;

    counter++;
    if (counter == 100) {
        printf("drop %d bytes received from %x to %x\n", bytes,
            ntohl(ip->ip_src.s_addr), ntohl(ip->ip_dst.s_addr));
        counter = 0;
        freemsg(*pkt->hpe_mp);
        *pkt->hpe_mp = NULL;
        pkt->hpe_mb = NULL;
        pkt->hpe_hdr = NULL;
        return (1);
    }
    return (0);
}
```

この関数で受信されたパケットと、パケットイベントからコールバックとして呼び出されるすべての要素は、1 つずつ受信されます。パケットとこのインタフェースは連鎖していないので、呼び出しごとにパケットは 1 個だけであり、b_next は常に NULL になります。ほかのパケットはありませんが、1 個のパケットが、b_cont と連鎖した複数の mblk_t 構造体で構成されることがあります。

パケットフィルタリングフックの例

コンパイルしてカーネルに読み込むことができる完全な例を次に示します。

64 ビットシステムで動作中のカーネルモジュールにこのコードをコンパイルするには、次のコマンドを使用します。

```
# gcc -D_KERNEL -m64 -c full.c
# ld -dy -Nmisc/neti -Nmisc/hook -r full.o -o full
```

例 10–1 パケットフィルタリングフックのプログラム例

```
/*
 * This file is a test module written to test the netinfo APIs in OpenSolaris.
 * It is being published to demonstrate how the APIs can be used.
 */
#include <sys/param.h>
#include <sys/sunddi.h>
#include <sys/modctl.h>
#include <sys/ddi.h>
#include "neti.h"

/*
 * Module linkage information for the kernel.
 */
static struct modldrv modlmisc = {
        &mod_miscops,           /* drv_modops */
        "neti test module",     /* drv_linkinfo */
};

static struct modlinkage modlinkage = {
        MODREV_1,               /* ml_rev */
        &modlmisc,              /* ml_linkage */
        NULL
};

typedef struct scratch_s {
        int             sentinel_1;
        netid_t         id;
        int             sentinel_2;
        int             event_notify;
        int             sentinel_3;
        int             v4_event_notify;
        int             sentinel_4;
        int             v6_event_notify;
        int             sentinel_5;
        int             arp_event_notify;
        int             sentinel_6;
        int             v4_hook_notify;
        int             sentinel_7;
        int             v6_hook_notify;
        int             sentinel_8;
        int             arp_hook_notify;
        int             sentinel_9;
        hook_t          *v4_h_in;
        int             sentinel_10;
```

例 10–1 パケットフィルタリングフックのプログラム例 *(続き)*

```
        hook_t          *v6_h_in;
        int             sentinel_11;
        hook_t          *arp_h_in;
        int             sentinel_12;
        net_handle_t    v4;
        int             sentinel_13;
        net_handle_t    v6;
        int             sentinel_14;
        net_handle_t    arp;
        int             sentinel_15;
} scratch_t;

#define MAX_RECALL_DOLOG        10000
char    recall_myname[10];
net_instance_t *recall_global;
int     recall_inited = 0;
int     recall_doing[MAX_RECALL_DOLOG];
int     recall_doidx = 0;
kmutex_t        recall_lock;
int     recall_continue = 1;
timeout_id_t    recall_timeout;
int     recall_steps = 0;
int     recall_alloced = 0;
void    *recall_alloclog[MAX_RECALL_DOLOG];
int     recall_freed = 0;
void    *recall_freelog[MAX_RECALL_DOLOG];

static int recall_init(void);
static void recall_fini(void);
static void *recall_create(const netid_t id);
static void recall_shutdown(const netid_t id, void *arg);
static void recall_destroy(const netid_t id, void *arg);
static int recall_newproto(hook_notify_cmd_t cmd, void *arg,
    const char *parent, const char *event, const char *hook);
static int recall_newevent(hook_notify_cmd_t cmd, void *arg,
    const char *parent, const char *event, const char *hook);
static int recall_newhook(hook_notify_cmd_t cmd, void *arg,
    const char *parent, const char *event, const char *hook);
static void recall_expire(void *arg);

static void recall_strfree(char *);
static char *recall_strdup(char *, int);

static void
recall_add_do(int mydo)
{
        mutex_enter(&recall_lock);
        recall_doing[recall_doidx] = mydo;
        recall_doidx++;
        recall_steps++;
        if ((recall_steps % 1000000) == 0)
                printf("stamp %d %d\n", recall_steps, recall_doidx);
        if (recall_doidx == MAX_RECALL_DOLOG)
                recall_doidx = 0;
        mutex_exit(&recall_lock);
}
```

例 10–1 パケットフィルタリングフックのプログラム例 *(続き)*

```
static void *recall_alloc(size_t len, int wait)
{
        int i;

        mutex_enter(&recall_lock);
        i = recall_alloced++;
        if (recall_alloced == MAX_RECALL_DOLOG)
                recall_alloced = 0;
        mutex_exit(&recall_lock);

        recall_alloclog[i] = kmem_alloc(len, wait);
        return recall_alloclog[i];
}

static void recall_free(void *ptr, size_t len)
{
        int i;

        mutex_enter(&recall_lock);
        i = recall_freed++;
        if (recall_freed == MAX_RECALL_DOLOG)
                recall_freed = 0;
        mutex_exit(&recall_lock);

        recall_freelog[i] = ptr;
        kmem_free(ptr, len);
}

static void recall_assert(scratch_t *s)
{
        ASSERT(s->sentinel_1 == 0);
        ASSERT(s->sentinel_2 == 0);
        ASSERT(s->sentinel_3 == 0);
        ASSERT(s->sentinel_4 == 0);
        ASSERT(s->sentinel_5 == 0);
        ASSERT(s->sentinel_6 == 0);
        ASSERT(s->sentinel_7 == 0);
        ASSERT(s->sentinel_8 == 0);
        ASSERT(s->sentinel_9 == 0);
        ASSERT(s->sentinel_10 == 0);
        ASSERT(s->sentinel_11 == 0);
        ASSERT(s->sentinel_12 == 0);
        ASSERT(s->sentinel_13 == 0);
        ASSERT(s->sentinel_14 == 0);
        ASSERT(s->sentinel_15 == 0);
}

int
_init(void)
{
        int error;

        bzero(recall_doing, sizeof(recall_doing));
        mutex_init(&recall_lock, NULL, MUTEX_DRIVER, NULL);
```

例 10–1 パケットフィルタリングフックのプログラム例 (続き)

```
        error = recall_init();
        if (error == DDI_SUCCESS) {
                error = mod_install(&modlinkage);
                if (error != 0)
                        recall_fini();
        }

        recall_timeout = timeout(recall_expire, NULL, drv_usectohz(500000));

        return (error);
}

int
_fini(void)
{
        int error;

        recall_continue = 0;
        if (recall_timeout != NULL) {
                untimeout(recall_timeout);
                recall_timeout = NULL;
        }
        error = mod_remove(&modlinkage);
        if (error == 0) {
                recall_fini();
                delay(drv_usectohz(500000));    /* .5 seconds */

                mutex_destroy(&recall_lock);

                ASSERT(recall_inited == 0);
        }

        return (error);
}

int
_info(struct modinfo *info)
{
        return(0);
}

static int
recall_init()
{
        recall_global = net_instance_alloc(NETINFO_VERSION);

        strcpy(recall_myname, "full_");
        bcopy(((char *)&recall_global) + 4, recall_myname + 5, 4);
        recall_myname[5] = (recall_myname[5] & 0x7f) | 0x20;
        recall_myname[6] = (recall_myname[6] & 0x7f) | 0x20;
        recall_myname[7] = (recall_myname[7] & 0x7f) | 0x20;
        recall_myname[8] = (recall_myname[8] & 0x7f) | 0x20;
        recall_myname[9] = '\0';

        recall_global->nin_create = recall_create;
        recall_global->nin_shutdown = recall_shutdown;
```

例 10–1 パケットフィルタリングフックのプログラム例 (続き)

```
        recall_global->nin_destroy = recall_destroy;
        recall_global->nin_name = recall_myname;

        if (net_instance_register(recall_global) != 0)
                return (DDI_FAILURE);

        return (DDI_SUCCESS);
}

static void
recall_fini()
{

        if (recall_global != NULL) {
                net_instance_unregister(recall_global);
                net_instance_free(recall_global);
                recall_global = NULL;
        }
}

static void
recall_expire(void *arg)
{

        if (!recall_continue)
                return;

        recall_fini();

        if (!recall_continue)
                return;

        delay(drv_usectohz(5000));      /* .005 seconds */

        if (!recall_continue)
                return;

        if (recall_init() == DDI_SUCCESS)
                recall_timeout = timeout(recall_expire, NULL,
                    drv_usectohz(5000));        /* .005 seconds */
}

static void *
recall_create(const netid_t id)
{
        scratch_t *s = kmem_zalloc(sizeof(*s), KM_SLEEP);

        if (s == NULL)
                return (NULL);

        recall_inited++;

        s->id = id;

        net_instance_notify_register(id, recall_newproto, s);
```

例 10–1 パケットフィルタリングフックのプログラム例 (続き)

```
        return s;
}

static void
recall_shutdown(const netid_t id, void *arg)
{
        scratch_t *s = arg;

        ASSERT(s != NULL);
        recall_add_do(__LINE__);
        net_instance_notify_unregister(id, recall_newproto);

        if (s->v4 != NULL) {
                if (s->v4_h_in != NULL) {
                        net_hook_unregister(s->v4, NH_PHYSICAL_IN,
                            s->v4_h_in);
                        recall_strfree(s->v4_h_in->h_name);
                        hook_free(s->v4_h_in);
                        s->v4_h_in = NULL;
                }
                if (net_protocol_notify_unregister(s->v4, recall_newevent))
                        cmn_err(CE_WARN,
                            "v4:net_protocol_notify_unregister(%p) failed",
                            s->v4);
                net_protocol_release(s->v4);
                s->v4 = NULL;
        }

        if (s->v6 != NULL) {
                if (s->v6_h_in != NULL) {
                        net_hook_unregister(s->v6, NH_PHYSICAL_IN,
                            s->v6_h_in);
                        recall_strfree(s->v6_h_in->h_name);
                        hook_free(s->v6_h_in);
                        s->v6_h_in = NULL;
                }
                if (net_protocol_notify_unregister(s->v6, recall_newevent))
                        cmn_err(CE_WARN,
                            "v6:net_protocol_notify_unregister(%p) failed",
                            s->v6);
                net_protocol_release(s->v6);
                s->v6 = NULL;
        }

        if (s->arp != NULL) {
                if (s->arp_h_in != NULL) {
                        net_hook_unregister(s->arp, NH_PHYSICAL_IN,
                            s->arp_h_in);
                        recall_strfree(s->arp_h_in->h_name);
                        hook_free(s->arp_h_in);
                        s->arp_h_in = NULL;
                }
                if (net_protocol_notify_unregister(s->arp, recall_newevent))
                        cmn_err(CE_WARN,
                            "arp:net_protocol_notify_unregister(%p) failed",
                            s->arp);
```

例 10–1 パケットフィルタリングフックのプログラム例 (続き)

```
                net_protocol_release(s->arp);
                s->arp = NULL;
        }
}

static void
recall_destroy(const netid_t id, void *arg)
{
        scratch_t *s = arg;

        ASSERT(s != NULL);

        recall_assert(s);

        ASSERT(s->v4 == NULL);
        ASSERT(s->v6 == NULL);
        ASSERT(s->arp == NULL);
        ASSERT(s->v4_h_in == NULL);
        ASSERT(s->v6_h_in == NULL);
        ASSERT(s->arp_h_in == NULL);
        kmem_free(s, sizeof(*s));

        ASSERT(recall_inited > 0);
        recall_inited--;
}

static int
recall_newproto(hook_notify_cmd_t cmd, void *arg, const char *parent,
    const char *event, const char *hook)
{
        scratch_t *s = arg;

        s->event_notify++;

        recall_assert(s);

        switch (cmd) {
        case HN_REGISTER :
                if (strcmp(parent, NHF_INET) == 0) {
                        s->v4 = net_protocol_lookup(s->id, parent);
                        net_protocol_notify_register(s->v4, recall_newevent, s);
                } else if (strcmp(parent, NHF_INET6) == 0) {
                        s->v6 = net_protocol_lookup(s->id, parent);
                        net_protocol_notify_register(s->v6, recall_newevent, s);
                } else if (strcmp(parent, NHF_ARP) == 0) {
                        s->arp = net_protocol_lookup(s->id, parent);
                        net_protocol_notify_register(s->arp,recall_newevent, s);
                }
                break;

        case HN_UNREGISTER :
        case HN_NONE :
                break;
        }

        return 0;
```

例 10–1 パケットフィルタリングフックのプログラム例 (続き)

```
}

static int
recall_do_event(hook_event_token_t tok, hook_data_t data, void *ctx)
{
        scratch_t *s = ctx;

        recall_assert(s);

        return (0);
}

static int
recall_newevent(hook_notify_cmd_t cmd, void *arg, const char *parent,
    const char *event, const char *hook)
{
        scratch_t *s = arg;
        char buffer[32];
        hook_t *h;

        recall_assert(s);

        if (strcmp(event, NH_PHYSICAL_IN) == 0) {

                sprintf(buffer, "%s_%s_%s", recall_myname, parent, event);
                h = hook_alloc(HOOK_VERSION);
                h->h_hint = HH_NONE;
                h->h_arg = s;
                h->h_name = recall_strdup(buffer, KM_SLEEP);
                h->h_func = recall_do_event;
        } else {
                h = NULL;
        }

        if (strcmp(parent, NHF_INET) == 0) {
                s->v4_event_notify++;
                if (h != NULL) {
                        s->v4_h_in = h;
                        net_hook_register(s->v4, (char *)event, h);
                }
                net_event_notify_register(s->v4, (char *)event,
                    recall_newhook, s);

        } else if (strcmp(parent, NHF_INET6) == 0) {
                s->v6_event_notify++;
                if (h != NULL) {
                        s->v6_h_in = h;
                        net_hook_register(s->v6, (char *)event, h);
                }
                net_event_notify_register(s->v6, (char *)event,
                    recall_newhook, s);

        } else if (strcmp(parent, NHF_ARP) == 0) {
                s->arp_event_notify++;
                if (h != NULL) {
                        s->arp_h_in = h;
```

例 10–1 パケットフィルタリングフックのプログラム例 (続き)

```
                        net_hook_register(s->arp, (char *)event, h);
                }
                net_event_notify_register(s->arp, (char *)event,
                    recall_newhook, s);
        }
        recall_assert(s);

        return (0);
}

static int
recall_newhook(hook_notify_cmd_t cmd, void *arg, const char *parent,
    const char *event, const char *hook)
{
        scratch_t *s = arg;

        recall_assert(s);

        if (strcmp(parent, NHF_INET) == 0) {
                s->v4_hook_notify++;

        } else if (strcmp(parent, NHF_INET6) == 0) {
                s->v6_hook_notify++;

        } else if (strcmp(parent, NHF_ARP) == 0) {
                s->arp_hook_notify++;
        }
        recall_assert(s);

        return (0);
}

static void recall_strfree(char *str)
{
        int len;

        if (str != NULL) {
                len = strlen(str);
                recall_free(str, len + 1);
        }
}

static char* recall_strdup(char *str, int wait)
{
        char *newstr;
        int len;

        len = strlen(str);
        newstr = recall_alloc(len, wait);
        if (newstr != NULL)
                strcpy(newstr, str);

        return (newstr);
}
```

例 10–2　net_inject のプログラム例

```
* Copyright (c) 2012, Oracle and/or its affiliates.
 All rights reserved.
 */

* PAMP driver - Ping Amplifier enables Solaris to send two ICMP echo
* responses for every ICMP request.
* This example provides a test module of the Oracle Solaris PF-hooks
* (netinfo(9f)) API.This example discovers ICMP echo
* implementation by intercepting inbound packets using
* physical-in‘ event hook.
* If the intercepted packet happens to be a ICMPv4 echo request,
* the module will generate a corresponding ICMP echo response
* which will then be sent to the network interface card using
* the net_inject(9f) function. The original ICMPv4 echo request will be
* allowed to enter the the IP stack so that the request can be
* processed by the destination IP stack.
* The destination stack in turn will send its own ICMPv4 echo response.
* Therefore there will be two ICMPv4 echo responses for a single
* ICMPv4 echo request.

*
* The following example code demonstrates two key functions of netinfo(9f) API:
*
* Packet Interception
*
* Packet Injection
*
* In order to be able to talk to netinfo(9f), the driver must allocate and
* register its own net_instance_t - ‘pamp_ninst‘. This happens in the
* pamp_attach() function, which imlements ‘ddi_attach‘ driver operation.The
* net_instance_t registers three callbacks with netinfo(9f) module:
*     _create
*    _shutdown
*    _destroy
* The netinfo(9f) command uses these functions to request the driver to
* create, shutdown, or destroy the driver context bound to a particular IP instance.
* This will enable the driver to handle packets for every IP stack found in
* the Oracle Solaris kernel. For purposes of this example, the driver is always
* implicitly bound to every IP instance.
*/

/* Use the following makefile to build the driver::
/* Begin Makefile */
ALL = pamp_drv pamp_drv.conf

pamp_drv = pamp_drv.o

pamp_drv.conf: pamp_drv
echo ’name="pamp_drv" parent="pseudo" instance=0;’ > pamp_drv.conf

pamp_drv: pamp_drv.o
ld -dy -r -Ndrv/ip -Nmisc/neti -Nmsic/hook -o pamp_drv pamp_drv.o
pamp_drv.o: pamp_drv.c
cc -m64 -xmodel=kernel -D_KERNEL -c -o $@ $<

install:
cp pamp_drv /usr/kernel/drv/‘isainfo -k‘/pamp_drv
```

例 10–2 net_inject のプログラム例 (続き)

```
cp pamp_drv.conf /usr/kernel/drv/pamp_drv.conf

uninstall:
rm -rf /usr/kernel/drv/‘isainfo -k‘/pamp_drv
rm -rf /usr/kernel/drv/pamp_drv.conf

clean:
    rm -f pamp_drv.o pamp_drv pamp_drv.conf

*End Makefile */

 *
* The Makefile shown above will build a pamp_drv driver binary
* and pamp_drv.conf file for driver configuration. If you are
* building on a test machine, use ‘make install‘ to place
* driver and configuration files in the specified location.
* Otherwise copy the pamp_drv binary and the pamp_drv.conf
files to your test machine manually.
*
* Run the following command to load the driver to kernel:

    add_drv pam_drv
* Run the following command to unload the driver to kernel:

    rem_drv pamp_drv
*
* To check if your driver is working you need to use a snoop
* and ‘ping‘ which will be running
* on a remote host. Start snoop on your network interface:

    snoop -d netX icmp

 * Run a ping on a remote host:

ping -ns <test.box>
* test.box refers to the system where the driver is installed.

*
* The snoop should show there are two ICMP echo replies for every ICMP echo
 * request. The expected output should be similar to the snoop output shown below:
 * 172.16.1.2 -> 172.16.1.100 ICMP Echo request (ID: 16652 Sequence number: 0)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 0)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 0)
 * 172.16.1.2 -> 172.16.1.100 ICMP Echo request (ID: 16652 Sequence number: 1)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 1)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 1)
 * 172.16.1.2 -> 172.16.1.100 ICMP Echo request (ID: 16652 Sequence number: 2)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 2)
 * 172.16.1.100 -> 172.16.1.2   ICMP Echo reply (ID: 16652 Sequence number: 2)
 */
#include <sys/atomic.h>
#include <sys/ksynch.h>
#include <sys/ddi.h>
#include <sys/modctl.h>
#include <sys/random.h>
#include <sys/sunddi.h>
```

例 10–2　net_inject のプログラム例　　(続き)

```
#include <sys/stream.h>
#include <sys/devops.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/neti.h>
#include <sys/hook.h>
#include <sys/hook_event.h>
#include <sys/synch.h>
#include <inet/ip.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h
#include <netinet/ip_icmp.h>


/*
 * This is a context for the driver. The context is allocated by
 * pamp_nin_create() callback for every IP instance found in kernel.
 */
typedef struct pamp_ipstack
{
    hook_t *pamp_phyin;
    int pamp_hook_ok;
    net_handle_t    pamp_ipv4;
} pamp_ipstack_t;
static kmutex_t    pamp_stcksmx;
/*
 * The netinstance, which passes driver callbacks to netinfo module.
 */
static net_instance_t    *pamp_ninst = NULL;
/*
 * Solaris kernel driver APIs.
 */
static int pamp_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int pamp_attach(dev_info_t *, ddi_attach_cmd_t);
static int pamp_detach(dev_info_t *, ddi_detach_cmd_t);static dev_info_t    *pamp_dev_info = NULL;
/*
 * Driver does not support any device operations.
 */

extern struct cb_ops no_cb_ops;

static struct dev_ops pamp_ops = {
    DEVO_REV,
    0,
    pamp_getinfo,
    nulldev,
    nulldev,
    pamp_attach,
    pamp_detach,
    nodev,
 &no_cb_ops,
    NULL,
    NULL,
};
```

例 10–2 net_inject のプログラム例 (続き)

```
static struct modldrv    pamp_module = {
&mod_driverops,
    "ECHO_1",
    &pamp_ops
};
static struct modlinkage pamp_modlink = {
    MODREV_1,
    &pamp_module,
    NULL
};

/*
 * Netinfo stack instance create/destroy/shutdown routines.
 */
static void *pamp_nin_create(const netid_t);
static void pamp_nin_destroy(const netid_t, void *);
static void pamp_nin_shutdown(const netid_t, void *);

/*
 * Callback to process intercepted packets delivered by hook event
 */
static int pamp_pkt_in(hook_event_token_t, hook_data_t, void *);

/*
 * Kernel driver getinfo operation
 */
static int
pamp_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void * arg, void **resultp)
{
    int    e;

    switch (cmd) {
        case DDI_INFO_DEVT2DEVINFO:
            *resultp = pamp_dev_info;
            e = DDI_SUCCESS;
            break;
        case DDI_INFO_DEVT2INSTANCE:
            *resultp = NULL;
            e = DDI_SUCCESS;
            break;
        default:
            e = DDI_FAILURE;
    }

    return (e);
}
/*
 * Kernel driver attach operation. The job of the driver is to create a net
 * instance for our driver and register it with netinfo(9f)
 */
static int pamp_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int    rc;
#define    RETURN(_x_)
    do    {
        mutex_exit(&pamp_stcksmx);
```

例 10–2 net_inject のプログラム例 (続き)

```
    return (_x_);
} while (0)

/*
 * Fail for all commands except DDI_ATTACH.
 */
if (cmd != DDI_ATTACH) {
    return (DDI_FAILURE);
}
mutex_enter(&pamp_stcksmx);
/*
 * It is an error to apply attach operation on a driver which is already
 * attached.
 */
if (pamp_ninst != NULL) {
    RETURN(DDI_FAILURE);
}
/*
 * At most one driver instance is allowed (instance 0).
 */
if (ddi_get_instance(dip) != 0) {
    RETURN(DDI_FAILURE);
}

rc = ddi_create_minor_node(dip, "pamp", S_IFCHR, 0, DDI_PSEUDO, 0);
if (rc != DDI_SUCCESS) {
    ddi_remove_minor_node(dip, NULL);
    RETURN(DDI_FAILURE);
}

/*
 * Create and register pamp net instance.  Note we are assigning
 * callbacks _create, _destroy, _shutdown. These callbacks will ask
 * our driver to create/destroy/shutdown our IP driver instances.
 */
pamp_ninst = net_instance_alloc(NETINFO_VERSION);
if (pamp_ninst == NULL) {
    ddi_remove_minor_node(dip, NULL);
    RETURN(DDI_FAILURE);
}

pamp_ninst->nin_name = "pamp";
pamp_ninst->nin_create = pamp_nin_create;
pamp_ninst->nin_destroy = pamp_nin_destroy;
pamp_ninst->nin_shutdown = pamp_nin_shutdown;
pamp_dev_info = dip;
mutex_exit(&pamp_stcksmx);

/*
 * Although it is not shown in the following example, it is
 * recommended that all mutexes/exclusive locks be released before *
 * calling net_instance_register(9F) to avoid a recursive lock
 * entry.  As soon as pamp_ninst is registered, the
 * net_instance_register(9f) will call pamp_nin_create() callback.
 * The callback will run in the same context as the one in which
 * pamp_attach() is running. If pamp_nin_create() grabs the same
```

例 10–2 net_inject のプログラム例 (続き)

```
     * lock held already by pamp_attach(), then such a lock is being
     * operated on recursively.
     */
    (void) net_instance_register(pamp_ninst);

    return (DDI_SUCCESS);
#undef    RETURN
}

/*
 * The detach function will unregister and destroy our driver netinstance. The same rules
 * for exclusive locks/mutexes introduced for attach operation apply to detach.
 * The netinfo will take care to call the shutdown()/destroy() callbacks for
 * every IP stack instance.
 */
static int
pamp_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    pamp_ipstack_t    *pamp_ipstack;
    net_instance_t    *ninst = NULL;

    /*
     * It is an error to apply detach operation on driver, when another
     * detach operation is running (in progress), or when detach operation
     * is complete (pamp_ninst).
     */
    mutex_enter(&pamp_stcksmx);
    if (pamp_ninst == NULL) {
        mutex_exit(&pamp_stcksmx);
        return (DDI_FAILURE);
    }

    ninst = pamp_ninst;
    pamp_ninst = NULL;
    mutex_exit(&pamp_stcksmx);

    /*
     * Calling net_instance_unregister(9f) will invoke pamp_nin_destroy()
     * for every pamp_ipstack instance created so far.  Therefore it is advisable
     * to not hold any mutexes, because it might get grabbed by pamp_nin_destroy() function.
     */
    net_instance_unregister(ninst);
    net_instance_free(ninst);

    (void) ddi_get_instance(dip);
    ddi_remove_minor_node(dip, NULL);


    return (DDI_SUCCESS);
}

/*
 * Netinfo callback, which is supposed to create an IP stack context for our
 * ICMP echo server.
 *
 * NOTE: NULL return value is not interpreted as a failure here. The
```

例 10–2　net_inject のプログラム例　　(続き)

```
 * pamp_nin_shutdown()/pamp_nin_destroy() will receive NULL pointer for IP stack
 * instance with given ‘netid‘ id.
 *
 */
static void *
pamp_nin_create(const netid_t netid)
{
    pamp_ipstack_t    *pamp_ipstack;

    pamp_ipstack = (pamp_ipstack_t *)kmem_zalloc(
        sizeof (pamp_ipstack_t), KM_NOSLEEP);

    if (pamp_ipstack == NULL) {
        return (NULL);
    }

    HOOK_INIT(pamp_ipstack->pamp_phyin, pamp_pkt_in, "pkt_in",
        pamp_ipstack);

    pamp_ipstack->pamp_ipv4 = net_protocol_lookup(netid, NHF_INET);
    if (pamp_ipstack->pamp_ipv4 == NULL) {
        kmem_free(pamp_ipstack, sizeof (pamp_ipstack_t));
        return (NULL);
    }

    pamp_ipstack->pamp_hook_ok = net_hook_register(
        pamp_ipstack->pamp_ipv4, NH_PHYSICAL_IN, pamp_ipstack->pamp_phyin);
    if (pamp_ipstack->pamp_hook_ok != 0) {
        net_protocol_release(pamp_ipstack->pamp_ipv4);
        hook_free(pamp_ipstack->pamp_phyin);
        kmem_free(pamp_ipstack, sizeof (pamp_ipstack_t));
        return (NULL);
    }

    return (pamp_ipstack);
}

/*
 * This event is delivered right before the particular stack instance is
 * destroyed.
 */
static void
pamp_nin_shutdown(const netid_t netid, void *stack)
{
    return;
}

/*
 * Important to note here that the netinfo(9f) module ensures that no
 * no pamp_pkt_in() is "running" when the stack it is bound to is being destroyed.
 */


static void
pamp_nin_destroy(const netid_t netid, void *stack)
{
```

例 10–2 net_inject のプログラム例 (続き)

```
    pamp_ipstack_t    *pamp_ipstack = (pamp_ipstack_t *)stack;

    /*
     * Remember stack can be NULL! The pamp_nin_create() function returns
     * NULL on failure. The return value of pamp_nin_create() function will
     * be ‘kept‘ in netinfo module as a driver context for particular IP
     * instance. As soon as the instance is destroyed the NULL value
     * will appear here in pamp_nin_destroy(). Same applies to
     * pamp_nin_shutdown(). Therefore our driver must be able to handle
     * NULL here.
     */
    if (pamp_ipstack == NULL)
        return;

    /*
     * If driver has managed to initialize packet hook, then it has to be
     * unhooked here.
     */
    if (pamp_ipstack->pamp_hook_ok != -1) {
        (void) net_hook_unregister(pamp_ipstack->pamp_ipv4,
            NH_PHYSICAL_IN, pamp_ipstack->pamp_phyin);
        hook_free(pamp_ipstack->pamp_phyin);
        (void) net_protocol_release(pamp_ipstack->pamp_ipv4);
    }

    kmem_free(pamp_ipstack, sizeof (pamp_ipstack_t));
}

/*
 * Packet hook handler
 *
 * Function receives intercepted IPv4 packets coming from NIC to IP stack.  If
 * inbound packet is ICMP ehco request, then function will generate ICMP echo
 * response and use net_inject() to send it to network.  Function will also let
 * ICMP echo request in, so it will be still processed by destination IP stack,
 * which should also generate its own ICMP echo response. The snoop should show
 * you there will be two ICMP echo responses leaving the system where the pamp
 * driver is installed
 */

static int
pamp_pkt_in(hook_event_token_t ev, hook_data_t info, void *arg)
{
    hook_pkt_event_t    *hpe = (hook_pkt_event_t *)info;
    phy_if_t        phyif;
    struct ip        *ip;

    /*
     * Since our pamp_pkt_in callback is hooked to PHYSICAL_IN hook pkt.
     * event only, the physical interface index will always be passed as
     * hpe_ifp member.
     *
     * If our hook processes PHYSICAL_OUT hook pkt event, then
     * the physical interface index will be passed as hpe_ofp member.
     */
     phyif = hpe->hpe_ifp;
```

例 10–2　net_inject のプログラム例　　(続き)

```
ip = hpe->hpe_hdr;
if (ip->ip_p == IPPROTO_ICMP) {
    mblk_t    *mb;

    /*
     * All packets are copied/placed into a continuous buffer to make
     * parsing easier.
     */
    if ((mb = msgpullup(hpe->hpe_mb, -1)) != NULL) {
        struct icmp    *icmp;
        pamp_ipstack_t    *pamp_ipstack = (pamp_ipstack_t *)arg;

        ip = (struct ip *)mb->b_rptr;
        icmp = (struct icmp *)(mb->b_rptr + IPH_HDR_LENGTH(ip));

        if (icmp->icmp_type == ICMP_ECHO) {
            struct in_addr    addr;
            uint32_t    sum;
            mblk_t    *echo_resp = copymsg(mb);
            net_inject_t    ninj;

            /*
             * We need to make copy of packet, since we are
             * going to turn it into ICMP echo response.
             */
            if (echo_resp == NULL) {
                return (0);
            }
            ip = (struct ip *)echo_resp->b_rptr;
            addr = ip->ip_src;
            ip->ip_src = ip->ip_dst;
            ip->ip_dst = addr;
            icmp = (struct icmp *) (echo_resp->b_rptr + IPH_HDR_LENGTH(ip));
            icmp->icmp_type = ICMP_ECHO_REPLY;
            sum = ~ntohs(icmp->icmp_cksum) & 0xffff;
            sum += (ICMP_ECHO_REQUEST - ICMP_ECHO_REPLY);
            icmp->icmp_cksum =
                htons(~((sum >> 16) + (sum & 0xffff)));

            /*
             * Now we have assembled an ICMP response with
             * correct chksum.  It's time to send it out.
             * We have to initialize command for
             * net_inject(9f) --  ninj.
             */
            ninj.ni_packet = echo_resp;
            ninj.ni_physical = phyif;
            /*
             * As we are going use NI_QUEUE_OUT to send
             * our ICMP response, we don't need to set up
             * .ni_addr, which is required for NI_DIRECT_OUT
             * injection path only. In such case packet
             * bypasses IP stack routing and is pushed
             * directly to physical device queue. Therefore
             * net_inject(9f) requires as to specify
```

例 10–2 net_inject のプログラム例 (続き)

```
                 * next-hop IP address.
                 *
                 * Using NI_QUEUE_OUT is more convenient for us
                 * since IP stack will take care of routing
                 * process and will find out ‘ni_addr‘
                 * (next-hop) address on its own.
                 */
                (void) net_inject(pamp_ipstack->pamp_ipv4,
                    NI_QUEUE_OUT, &ninj);
            }
        }
    }

    /*
     * 0 as return value will let packet in.
     */
    return (0);
}

/*
 * Kernel module handling.
 */
int init()
{
    mutex_init(&pamp_stcksmx, "pamp_mutex", MUTEX_DRIVER, NULL);
    return (mod_install(&pamp_modlink));
}

int fini()
{
    int rv;

    rv = mod_remove(&pamp_modlink);
    return (rv);
}

int info(struct modinfo *modinfop)
{
    return (mod_info(&pamp_modlink, modinfop));
}
```

◆ ◆ ◆ 第 11 章

11 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、スケーラビリティー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 11–1 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。274 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 11–1　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
```

ループ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
```

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

このセクションでは、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 11–2 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 11–3 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)
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を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 11–1 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;   /* host and service information */
struct nd_addrlist *nd_addrlistp; /* addresses for the service */
struct netbuf *netbufp;           /* the address of the service */
struct netconfig *nconf;          /* transport information*/
int i;                            /* the number of addresses */
char *uaddr;                      /* service universal address */
void *handlep;                    /* a handle into network selection */
/*
 * Set the host structure to reference the "date"
 * service on host "gandalf"
 */
nd_hostserv.h_host = "gandalf";
nd_hostserv.h_serv = "date";
/*
 * Initialize the network selection mechanism.
 */
if ((handlep = setnetpath()) == (void *)NULL) {
```

例 11–1 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)

```
   nc_perror(argv[0]);
   exit(1);
}
/*
 * Loop through the transport providers.
 */
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL)
{
   /*
    * Print out the information associated with the
    * transport provider described in the "netconfig"
    * structure.
    */
   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;
   }
   /*
    * Get the address for service "date" on the host
    * named "gandalf" over the transport provider
    * specified in the netconfig structure.
    */
   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);
   /*
    * Print out all addresses for service "date" on
    * host "gandalf" on current transport provider.
    */
   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 12 章

12 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。

この章では、次の内容について説明します。

- 283 ページの「リアルタイムスケジューラ」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。
- 294 ページの「メモリーのロック」
- 302 ページの「非同期ネットワーク通信」

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。このセクションでは、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。

- 283 ページの「リアルタイムスケジューラ」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。
- 294 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。
- 281 ページの「共有ライブラリ」で説明しているように、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、『マルチスレッドのプログラミング』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。

あるプロセスがもっと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、283 ページの「ディスパッチ応答時間」を参照してください。優先順位がもっと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。これらのイベントには、割り込みなどの外部イベント、リソース不足、同期入出力などの外部イベント待機、より高い優先順位を持つプロセスによる横取りなどが含まれます。

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

このセクションで説明する問題は、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、同期入出力は決して行わないでください。たとえば、大量のストレージ入出力の際に読み取りまたは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。

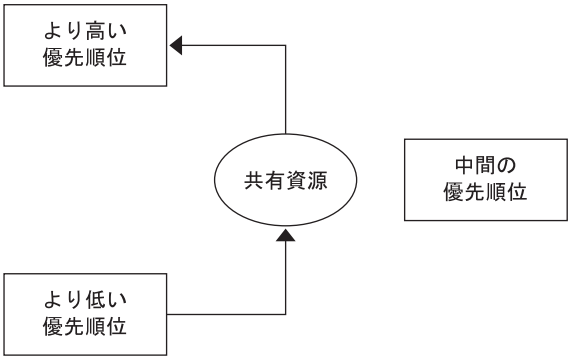
リアルタイムプログラムは、プログラムを起動するときに、環境変数 `LD_BIND_NOW` に `NULL` 以外の値を設定できます。環境変数 LD_BIND_NOW に NULL 以外の値を設定すると、共有ライブラリを使用しても動的バインドは行われません。また、すべての動的リンクは、プログラムの実行前にバインドが行なわれます。詳細は、『リンカーとライブラリ』を参照してください。

優先順位の逆転

リアルタイムプロセスが必要とするリソースを、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リソース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リソースを要求する例を示します。低い優先順位を持つプロセスが保持しているリソースを中間の優先順位を持つプロセスが横取りしているので、高い優先順位を持つプロセスはブロックされています。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セクションと同様に、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 12–1 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」のセクションで説明しています。

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。場合によっては、Stop-A を何度も押す必要があります。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 を押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ーを指定している場合、バッファーの状態は確定されません。バッファー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) および msync(3C) を使用して、セカンダリストレージとプロセスメモリー間の入出力転送を直接実行できます。

リアルタイムスケジューラ

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、処理するリソース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。

このセクションでは、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ーの使用方法について説明します。

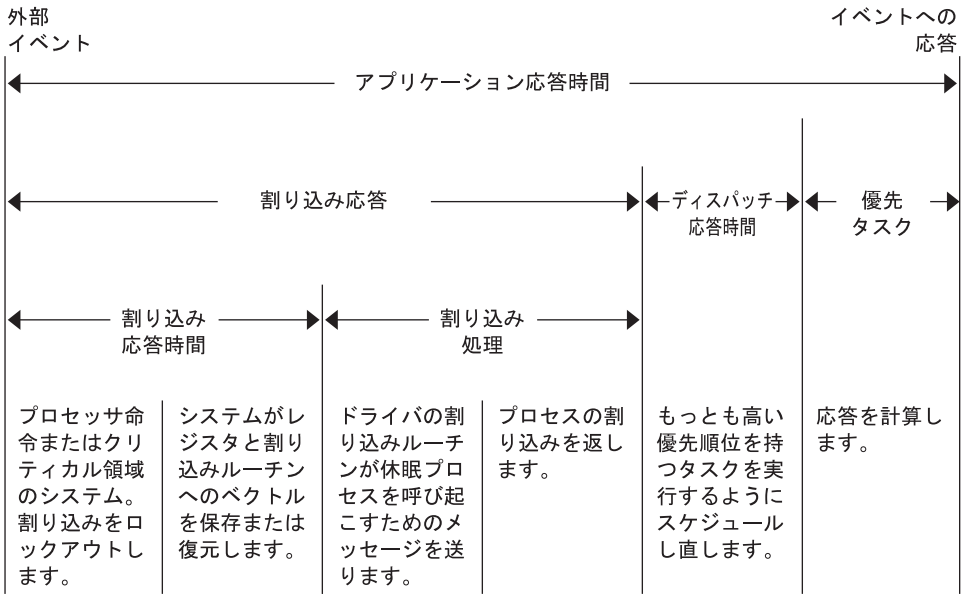
ディスパッチ応答時間

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もっとも重要な要素は、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12–2 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。

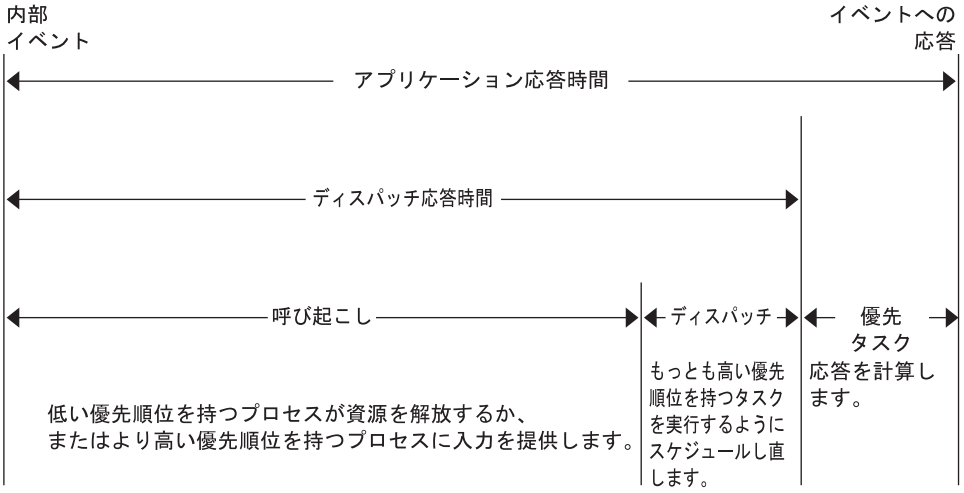
割り込み処理中、ドライバの割り込みルーチンはまず、優先順位が高いプロセスを呼び起こし、そのプロセスが終了すると戻ります。割り込まれたプロセスよりも優先順位が高いプロセスが現在ディスパッチ可能であることを検出すると、システムは(優先順位が高い)プロセスをディスパッチします。優先順位が低いプロセスから高いプロセスへコンテキストを切り換える時間は、ディスパッチ応答時間に含まれます。

図 12–3 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、ドライバが次のタスク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リソースを解放し、優先順位がより高いタスクをスケジューリングし直し、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 12–3　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 12–1 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
SPARCstation 5	0.3 ミリ秒未満	0.3 ミリ秒

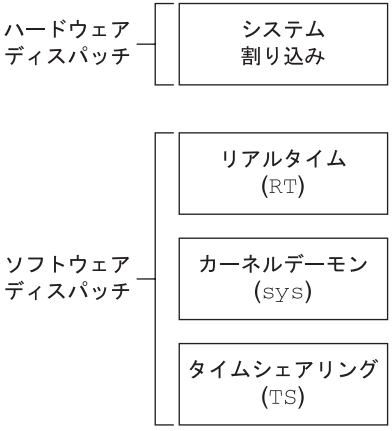
スケジューリングクラス

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ー) 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、もっと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構成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 12–4 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もっとも高いので、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、ソフトウェアではもっと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リングされます。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、入出力イベントを待つ間ブロックされた場合や、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た場合にも、実行を停止します。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、または横取りされた場合にのみ、実行を停止します。

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リング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。

優先順位がもっとも低いのは、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リングされます。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

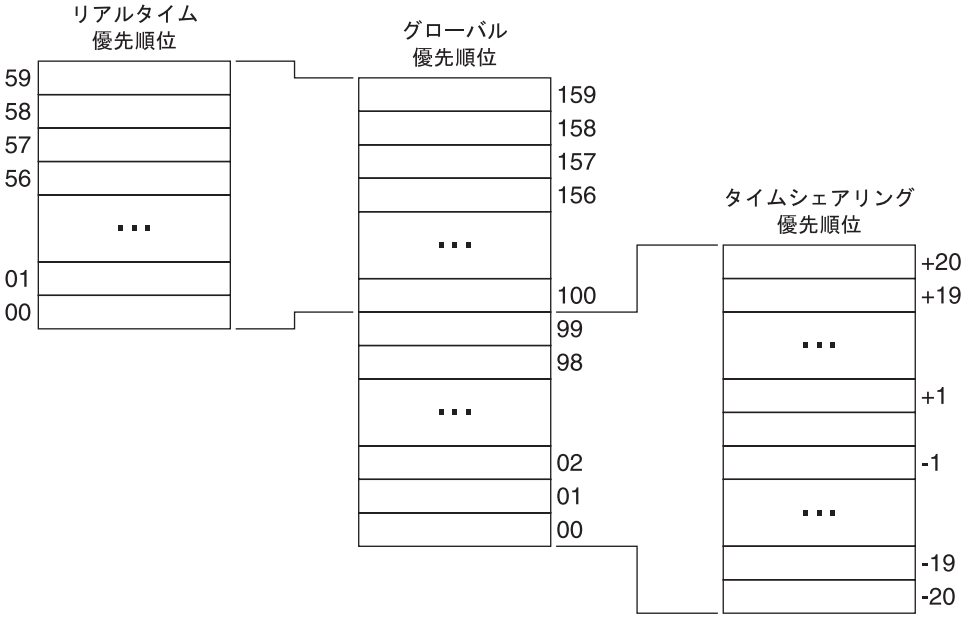
デフォルトのタイムシェアリング方式では、優先順位が低いプロセスに長いタイムスライスが与えられます。

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、優先順位がもっと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もっと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12–5 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 159 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もっと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替え、現在実行中のプロセスを横取りします。より高い優先順位を持つ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る可能性があります。

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リソース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。低い優先順位を持つプロセスがそのリソースを解放すると、カーネルはそのプロセスを横取りして、高い優先順位を持つ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リソース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」のセクション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` の使用法

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)` 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、より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(3RT)` 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ー

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ーには、`dispadmin(1M)` および `priocntl(1)` があります。どちらのユーティリティーも `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ーは、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)` ユーティリティーに `-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` 現在構成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 オプションとともに使用し、タイムカンタムの精度を指定する

`-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 12–1 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)` 実行可能モジュール

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する。

   ```
   # 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構成

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)` と `ts_dptbl(4)` が関連付けられています。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)` を使用することで構成されます。

ディスパッチャー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* 番目のパラメータ構造体 `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付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、303 ページの「タイムスタンプインタフェース」を参照してください。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ウンタを変更するには、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 の再構成

リアルタイムのシステム管理者は、いつでも config_rt_dptbl を再構成して、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」というセクション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ー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構成ファイルは 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[] に表示されるプロセスとして、優先順位を設定されたプロセス rtdpent_t とそれに関連付けられたタイムカンタムの config_rt_dptbl[] 値の例を示します。

例 12–1　RT クラスのディスパッチパラメータ

```
rtdpent_t  rt_dptbl[] = {             129,    60,
     /* prilevel Time quantum */                                  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 12–1 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。

このセクションでは、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要求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、メモリーの 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

メモリーによるページのロックを解除するために、プロセスは munlock(3C) 呼び出しによって、ロック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

スーパーユーザープロセスは、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クするように要求できます。MCL_CURRENT フラグが設定されている場合は、既存のメモリマッピングがすべてロックされます。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。

スティッキロックの復元

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リブートしてください。

高性能入出力

このセクションでは、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 コンポーネントに戻されます。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

このセクションでは、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; /* return value of read or write */
     int         aio_errno;  /* errno generated by the IO */
 } 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 の使用法

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 の使用法

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 の使用法

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にのみ成功します。操作がすでに進行している場合、aiocancel(3AIO) は失敗します。

aiowait の使用法

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返します。

poll() の使用法

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 ドライバの使用法

/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ーで行うことができます。このスケーラビリティーは、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ーを提供します。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 の使用法

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、298 ページの「aiocancel の使用法」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、関連付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セカンダリストレージの同期をとります。

`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

このセクションでは、リアルタイム処理と関連する、SunOS 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、およびセマフォー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 7 章「プロセス間通信」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情報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、123 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、125 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、129 ページの「System V メッセージ」および126 ページの「POSIX メッセージ」を参照してください。

セマフォーの使用法

セマフォーも System V と POSIX の両方のスタイルで提供されます。詳細は、132 ページの「System V セマフォー」および127 ページの「POSIX セマフォー」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、セマフォーを使用すると、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

プロセスがもっと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もっとも大きな問題です。

非同期ネットワーク通信

このセクションでは、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、171 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。

トランスポートレベルインタフェース (TLI) についての詳細は、第 9 章「XTI と TLI を使用したプログラミング」を参照してください。

ネットワーキングのモード

ソケットとトランスポートレベルインタフェース (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、着信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

このセクションでは、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、このセクション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間に呼び起こし、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 timespec 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精度に制限されます。その他のプラットフォームでは、タイマーレジスタの精度が 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リング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あと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ロギング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精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 12–2　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;

    /* block SIGALRM */
    sigemptyset (&sigset);
    sigaddset (&sigset, SIGALRM);
    sigprocmask (SIG_BLOCK, &sigset, NULL);

    /* set up handler for SIGALRM */
    act.sa_action = timerhandler;
    sigemptyset (&act.sa_mask);
    act.sa_flags = SA_SIGINFO;
    sigaction (SIGALRM, &act, NULL);
    /*
     * set up interval timer, starting in three seconds,
     *    then every 1/3 second
     */
    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);

    /* now wait for the alarms */
    sigemptyset (&sigset);
    timerhandler (0, si, NULL);
    while (timercnt < TIMERCNT) {
        ret = sigsuspend (&sigset);
```

例 12–2 タイマー割り込みの制御 *(続き)*

```
    }
    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 13 章

13 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、309 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` と `apptrace` については、311 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 のもっとも重要なコンポーネントは次のとおりです。

- 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セクション 3 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セクション 2 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セクション 4 を参照)
- Solaris ユーティリティーの入出力用の構文と意味論 (マニュアルページのセクション 1 を参照)

Solaris ABI の中心となるコンポーネント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コンポーネントだけを指します。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 4 つの ABI バージョンのうちのいずれかのバイナリに変換されます。バージョンは次のとおりです。

- 32 ビット SPARC
- 64 ビット SPARC
- 32 ビット x86
- 64 ビット x86 (Opteron)

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重要な違いがあります。

- 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 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、2 種類のライブラリ管理 (ファイルバージョン管理とシンボルバージョン管理) 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 (`libc.so.1` のように) 追加されます。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うと、バージョン番号は (`libc.so.2` のように) インクリメントされます。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。シンボルバージョン管理では、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ムがある場合は、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、`SUNW_1.2` は

SUNW_1.1 に関連付けられたすべてのシンボルを継承しています。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。たとえば、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、明示的に名前付きセットに関連付けられていないライブラリ内のシンボルは、その有効範囲が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、名前付けスキームを使用すると、シンボルの ABI ステータス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* の番号付けスキーム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。既存のシンボルに互換性のない変更が行われたときにそ

のシンボルが含まれるセットの major 番号が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 オペレーティングシステムは 2 つのツールを提供します。`appcert` ユーティリティーは、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のインスタンスについて静的な調査を行います。次に、`appcert` ユーティリティーは潜在的なバイナリ安定性の問題について、サマリー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視機能を使用して、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** を使用して選別します。これは、どの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を識別します。
- ターゲットの **Solaris** リリース上で **apptrace** を使用して検証します。これは、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します。

appcert ユーティリティー

`appcert` ユーティリティーは、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。このユーティリティーは SPARC または x86 のどちらのプラットフォーム上でも動作します。SPARC と x86 の 32 ビットインタフェースだけでなく、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` ユーティリティーは 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 ユーティリティーは、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 ユーティリティー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 ユーティリティー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 が非結合シンボルを報告するときは、存在しない非公開シンボルに依存しているなど、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 ユーティリティー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 ユーティリティーを実行するときに、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 ユーティリティーは 64 ビットアプリケーションを確認できません。Solaris では 64 ビットの静的ライブラリが提供されていないため、appcert は 64 ビットアプリケーションの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 ユーティリティー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。コードが C 言語である必要はありませんが、Solaris ライブラリの呼び出しには C 言語を使う必要があります。

appcert の使用法

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 ユーティリティーは、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 ユーティリティーは 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 は検索結果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 が出力ファイルに書き込むディレクトリ構造の詳細については、316 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 ユーティリティー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*	ライブラリコンポーネント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抱え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の問題が報告されることは起こり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 13–1 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

表 13–1 よくあるバイナリ安定性問題 (続き)

問題	回避方法
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` の使用法については、318 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` ユーティリティー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` ユーティリティー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` ユーティリティーの結果は、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
  - ファイル名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` が返す行には、「`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サマリー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、あとの 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 ユーティリティー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ことがあります。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 ユーティリティーは SPARC または x86 のどちらのプラットフォーム上でも動作します。apptrace ユーティリティーは、SPARC と x86 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 と入力し、次に希望のオプションを入力し、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を入力します。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 からの printf() の呼び出しをトレースすることができ、特定のライブラリ内のさまざまな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トレースできます。apptrace の動作を命令する仕様は、truss(1) の使用法と整合する構文によって制御されます。-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、スタックまたは呼び出し元の情報を調査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 13–1　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"
```

例 13–1 デフォルトのトレース動作 *(続き)*

```
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 13–2 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 13–3 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"
```

例 13–3 詳細なトレース (続き)

/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ーを定義します。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、148 ページの「コネクションエラー」を参照してください。149 ページの「データ転送」では、データの転送方法が、149 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

ま

め

ゆ

り

れ

ろ