IRIX® Admin: Resource Administration （日本語版）

007–3700–014JP

制作スタッフ

著作 Terry Schultz

編集 Susan Wilkening

イラスト Chris Wengelski

製作 Glen Traefald

協力エンジニア Tom Goozen、Sharif Islam、Marlys Kohnke、Tina Liang、Dennis Parker、Michael Sanford、Dan Stekloff、Sam Watters

カバー・デザイン Sarah Bolles (Sarah Bolles Design)、Dany Galgani (SGI Technical Publications)

このマニュアルでの新機能

『IRIX Admin: Resource Administration』の本改訂版では、IRIX オペレーティング・システムのリリース 6.5.20 をサポートしています。

新機能の説明

55 ページの「起動 cpuset」に、cpuset で XThread Control Interface (XTCI) を使用する場合の情報が追加されました。

ドキュメントの主な変更

ユーザ出口を使用する場合の情報を 80 ページの「`/usr/lib/acct` ディレクトリ」に、ユーザ出口スクリプトの例を107ページの「ユーザ出口の設定」にそれぞれ追加しました。

76 ページの図5-1および 92 ページの図5-2を更新しました。

25 ページの「ジョブ制限のインストール」から、ジョブ制限は IRIX フィーチャ・ストリームのみでサポートされるという記述を削除しました。このリリースから、ジョブ制限は、IRIX メンテナンス・ストリームとフィーチャ・ストリームの両方でサポートされるようになりました。

69 ページの 第5章「完全システム・アカウンティング」に、このリリースから、完全システム・アカウンティングが IRIX メンテナンス・ストリームとフィーチャ・ストリームの両方でサポートされるようになったことを示すメモを追加しました。

改訂情報

バージョン	説明
001	1999 年 7 月 初版
002	2000 年 1 月 IRIX 6.5.7 リリースをサポート
003	2000 年 4月 IRIX 6.5.8 リリースをサポート
004	2000 年 8 月 IRIX 6.5.9 リリースをサポート
005	2000 年 11 月 IRIX 6.5.10 リリースをサポート
006	2001 年 2 月 IRIX 6.5.11 リリースをサポート
007	2001 年 5 月 IRIX 6.5.12 リリースをサポート
008	2001 年 8 月 IRIX 6.5.13 リリースをサポート
009	2001 年 11 月 IRIX 6.5.14 リリースをサポート
010	2002 年 2 月 IRIX 6.5.15 リリースをサポート
011	2002 年 5 月 IRIX 6.5.16 リリースをサポート
012	2002 年 8 月 IRIX 6.5.17 リリースをサポート

013	2002 年 11 月 IRIX 6.5.18 リリースをサポート
014	2003 年 5 月 IRIX 6.5.20 リリースをサポート

目次

図一覧

表一覧

サンプル一覧

このマニュアルについて

このマニュアルでは、SGI サーバ・システムで実行される IRIX 6.5.20 オペレーティング・システムについて説明します。

このガイドは、IRIX オペレーティング・システムが実行されている SGI コンピュータ・システムの管理者の方向けのリファレンス・ドキュメントで、多様なシステム・リソース管理機能の利用に必要な情報が含まれています。

このマニュアルは、以下の章で構成されています。

関連ドキュメント

このガイドは IRIX Admin マニュアル・セットの一部で、サーバ、複合システム、およびユーザのホーム・ディレクトリや作業ディレクトリ直下以外のファイル構造に対して責任を持つ管理者の方向けに作成されています。ほかのユーザのためにシステムを保守する場合や、IRIX についてエンドユーザ・マニュアルよりも詳しい情報が必要な場合は、これらのガイドが役に立ちます。IRIX Admin ガイドは、IRIS InSight オンライン参照システムを通じて利用できます。このセットは、以下のマニュアルで構成されます。

- 『IRIX Admin: Software Installation and Licensing』- IRIX（SGI による UNIX オペレーティング・システムの実装）で実行されるソフトウェアのインストールおよびライセンスの方法について説明します。このマニュアルには、miniroot の実行方法や、`inst`(1M)（IRIX インストール・ユーティリティに対するコマンド・ライン・インタフェース）を使用したライブ・インストールの説明が含まれています。また、IRIX で実行される制限されたアプリケーションへのアクセスを制御するライセンス製品を識別し、ライセンス製品のドキュメントの参照先を示します。

- 『IRIX Admin: System Configuration and Operation』- 一般的なシステム管理作業をリストし、システム管理タスクについて説明します。これらの作業およびタスクには、オペレーティング・システムの設定、ユーザ・アカウント/ユーザ・プロセス/ディスク・リソースの管理、PROM モニタ内でのシステムとの対話、およびシステム・パフォーマンスの調整が含まれます。
- 『IRIX Admin: Disks and Filesystems』- ディスク、ファイルシステム、および論理ボリュームの概念について説明します。SCSI ディスク、XFS ファイルシステムと EFS (Extent File System) ファイルシステム、XLV 論理ボリューム、および I/O レート保証に関するシステム管理手順も記述されています。
- 『IRIX Admin: Networking and Mail』- ネットワークとメール・システムの計画、設定、使用、および保守の方法について説明します。これには、sendmail、UUCP、SLIP、および PPP の解説が含まれます。
- 『IRIX Admin: Backup, Security and Accounting』- ファイルをバックアップおよび復元する方法、システムとネットワークのセキュリティを保護する方法、およびユーザごとにシステムの使用状況を追跡する方法について説明します。
- 『IRIX Admin: Resource Administration』- システム・リソース管理の概要と、さまざまな IRIX リソース管理機能（IRIX プロセス制限、IRIX ジョブ制限、Miser バッチ処理システム、cpuset システム、完全システム・アカウンティング (CSA: Comprehensive System Accounting)、IRIX メモリの使用状況、array サービスなど）を使用および管理する方法について説明します。
- 『IRIX Admin: Peripheral Devices』- 端末、モデム、プリンタ、CD-ROM ドライブやテープ・ドライブなどの周辺機器用のソフトウェアを設定および保守する方法について説明します。
- 『IRIX Admin: Selected Reference Pages』–（InSight にはありません）– システム・ダウン時に必要になる可能性のあるコマンドの使い方について、マン・ページの情報を簡潔に紹介します。通常、個々のマン・ページは 1 つのコマンドの説明を示しますが、密接に関連する複数のコマンドについて説明するマン・ページもあります。マン・ページは、`man`(1) コマンドを使用してオンラインで参照できます。

出版物の入手方法

SGI のドキュメントは、以下の方法で入手できます。

- SGI Technical Publications Library `http://docs.sgi.com` を参照します。さまざまな形式が用意されています。このライブラリには、オンライン・ブック、リリース・ノート、マン・ページ、およびその他の情報の最新セットが幅広く含まれています。
- InfoSearch が SGI システムにインストールされている場合は、InfoSearch を使用できます。InfoSearch は、オンライン・ブック、リリース・ノート、およびマン・ページの限定されたセットを提供するオンライン・ツールです。IRIX システムで、Toolchest から「ヘルプ(Help)」->「InfoSearch」を選択するか、またはコマンド・ラインに「`infosearch`」と入力できます。

- リリース・ノートは、コマンド・ラインに「grelnotes」または「relnotes」と入力しても参照できます。
- マン・ページは、コマンド・ラインに「man *title*」と入力しても参照できます。

表記規則

このドキュメントでは、以下の表記規則が使用されています。

表記規則	意味
command	この固定スペース・フォントは、コマンド、ファイル、ルーチン、パス名、シグナル、メッセージ、プログラミング言語の構造などのリテラル項目を示します。
variable	イタリック体は、変数のエントリ、および定義される語や概念を示します。
user input	この太字体の固定スペース・フォントは、対話型セッションでユーザが入力するリテラル項目を示します。出力は、太字体ではない固定スペース・フォントで示されます。
[ ]	コマンドやディレクティブ行のオプション部分は、角括弧で囲みます。
...	省略記号は、先行する要素が繰返し可能であることを示します。

ご意見とお問合わせ先

このマニュアルの技術的正確性、内容、または構成についてご意見等ございましたら、SGI までお問合わせください。コメントいただくマニュアルのタイトルとドキュメント番号も必ず一緒にお聞かせいただくようお願いいたします。オンラインの場合、ドキュメント番号はマニュアルの最初の部分に記載されています。印刷マニュアルの場合は、各ページの下部にドキュメント番号が記載されています。

ご連絡の際は、以下のいずれかの方法をご利用いただけます。

- 電子メールの場合は、以下のアドレスまでお送りください。

  techpubs@sgi.com

- 次の Technical Publications Library の Web ページからの場合は、「Feedback」オプションをクリックしてください。

  http://docs.sgi.com

- お客様相談窓口までご連絡いただき、SGI の問題追跡システムへの入力をお申付けください。
- 郵送の場合は、次の住所までお送りください。

Technical Publications
SGI
1600 Amphitheatre Parkway, M/S 535
Mountain View, California 94043–1351, USA.

- FAX の場合は、+1 650 932 0801 の「Technical Publications」宛にお送りください。

いただいたコメントには迅速確実に対応いたします。

プロセス制限

標準のシステム・リソース制限を設定することにより、プロセス作成時にプロセスベースの同じ制限を各ログイン・プロセスに適用できます。この章では、プロセス制限について説明します。この章は、以下の節で構成されています。

プロセス制限の概要

IRIX オペレーティング・システムでは、プロセスごとの制限をサポートします。プロセスと、そのプロセスで作成される各プロセス(子プロセス)による、さまざまなシステム・リソース消費量の制限は、getrlimit(2) システム・コールで取得し、setrlimit(2) システム・コールで設定できます。

getrlimit または setrlimit の呼出しでは、操作の対象となるリソースと、リソース制限を指定します。リソースの制限は、値のペアです。1 つの値は、現在の(ソフト)制限を指定し、もう 1 つは、最大(ハード)制限を指定します。ソフト制限は、プロセスによってハード制限以下の任意の値に変更できます。また、プロセスによって、ハード制限をソフト制限以上の任意の値に下げることもできます(この操作は元に戻せません)。

csh と sh を使用したリソース消費量の制限

csh または sh で limit -h *resource max-use* というコマンドを使用すると、現在のプロセスまたはその子プロセスによって消費されるリソースを制限できます。

このコマンドは、現在のプロセスとその各子プロセスが消費するリソースの総量を制限し、それぞれが特定のリソースの最大使用量を超えないようにします。最大使用量を指定しなかった場合、現在の制限が表示されます。リソースを指定しなかった場合、すべての制限が示されます。-h フラグを指定すると、現在の制限の代わりに、ハード(最大)制限が使用されます。ハード制限は、現在の制限に対する上限値を示します。最大(ハード)制限を上げるには、CAP_PROC_MGT capability が必要です。

詳細については、csh(1) および sh(1) のマン・ページを参照してください。プロセスの操作権に対する細かな調整を可能にする capability についての詳細は、capability(4) および capabilities(4) のマン・ページを参照してください。

systune を使用したプロセス制限の表示と設定

表1-1 に、IRIX オペレーティング・システムでサポートされるプロセス制限を示します。

表1-1 プロセス制限

制限名	シンボリック ID	単位	説明	違反時の動作
rlimit_cpu_cur rlimit_cpu_max	RLIMIT_CPU	秒	プロセスで許可される CPU 時間（秒）の最大値	SIGXCPU シグナルによるプロセスの終了
rlimit_fsize_cur rlimit_fsize_max	RLIMIT_FSIZE	バイト	プロセスによって作成可能なファイルの最大サイズ	errno が EFBIG に設定されている場合、書込みまたは拡張の試行が失敗する
rlimit_data_cur rlimit_data_max	RLIMIT_DATA	バイト	プロセスの最大ヒープ・サイズ	errno が ENOMEM に設定されている場合、brk(2) の呼出しが失敗する
rlimit_stack_cur rlimit_stack_max	RLIMIT_STACK	バイト	プロセスの最大スタック・サイズ	SIGSEGV シグナルによるプロセスの終了
rlimit_ core_cur rlimit_core_max	RLIMIT_CORE	バイト	プロセスによって作成可能なコア・ファイルの最大サイズ	制限に到達した時点でコア・ファイルの書込みが終了する
rlimit_nofile_cur rlimit_nofile_max	RLIMIT_NOFILE	ファイル記述子	プロセスで同時に保持できる、開いているファイル記述子の最大数	errno が EMFILE に設定されている場合、open(2) の試行が失敗する

制限名	シンボリック ID	単位	説明	違反時の動作
rlimit_vmem_cur rlimit_vmem_max	RLIMIT_VMEM	バイト	プロセスの最大アドレス空間	errno が ENOMEM に設定されている場合、brk(2) と mmap(2) の呼出しが失敗する
rlimit_rss_cur rlimit_rss_max	RLIMIT_RSS	バイト	プロセスの常駐セット・サイズの最大サイズ	制限を超えた常駐ページが最初のスワップ候補になる
rlimit_pthread_cur rlimit_pthread_max	RLIMIT_PTHREAD	スレッド	プロセスで作成できるスレッドの最大数	errno が EAGAIN に設定されている場合、スレッド作成が失敗する

systune *resource* コマンドを使用して、プロセス制限のシステム全体のデフォルト値を表示および設定できます。*resource* グループには、以下の変数が含まれます。

```
rlimit_cpu_cur
rlimit_cpu_max
rlimit_fsize_cur
rlimit_fsize_max
rlimit_data_cur
rlimit_data_max
rlimit_stack_cur
rlimit_stack_max
rlimit_core_cur
rlimit_core_max
rlimit_nofile_cur
rlimit_nofile_max
rlimit_vmem_cur
rlimit_vmem_max
rlimit_rss_cur
```

```
rlimit_rss_max
rlimit_pthread_cur
rlimit_pthread_max
```

詳細については、systune(1M) のマン・ページを参照してください。

ジョブ制限ソフトウェアをシステムにインストールし、実行している場合は、ユーザ制限データベース (ULDB: User Limits Database) にユーザベースのプロセス制限を設定することもできます。rlimit_cpu_cur と rlimit_cpu_max のように、現在の値と最大値の両方を指定できます。ULDB の値は、systune(1M) コマンドによって設定されるシステムのデフォルトをオーバーライドします。

ULDB についての詳細は、12 ページの「ユーザ制限データベース (ULDB: User Limits Database)」を参照してください。

追加のプロセス制限パラメータ

IRIX には、特定のシステム制限を設定するためのパラメータが用意されています。たとえば、各プロセス(コアまたはファイル・サイズ)、ユーザあたりのグループの数、常駐ページの数などの最大値を設定できます。以下に、maxup および cpulimit_gracetime について説明します。すべてのパラメータは、/var/sysgen/mtune で設定および定義されています。

maxup	ユーザあたりのプロセスの最大数
cpulimit_gracetime	プロセス制限およびジョブ制限の猶予期間

maxup パラメータやその他の「システム制限パラメータ」についての詳細は、『IRIX Admin: System Configuration and Operation』を参照してください。

cpulimit_gracetime パラメータは、CPU 時間の制限を超えたプロセスの猶予期間を指定します。猶予期間は、制限を超えた後にプロセスで実行が許可される秒数に設定する必要があります。cputlimit_gracetime が設定されていない場合(つまり、ゼロの場合)、プロセスまたはジョブの CPU 制限を超えたプロセスに、SIGXCPU シグナルが送信されます。そのプロセスが実行し続けるかぎり、カーネルによって SIGXCPU シグナルが定期的に送信されます。プロセスでは、SIGXCPU シグナルを独自に処理するよう登録できるため、CPU 制限が無視されることがあります。

systune(1M) コマンドを使用して、cpulimit_gracetime パラメータをゼロ以外の値に設定した場合は、異なる処理が行われます。プロセスが CPU 制限を超えると、カーネルによって SIGXCPU シグナルが 1 回だけプロセスに送信されます。プロセスでは、このシグナルを登録し、必要なクリーンアップやシャットダウンの操作を行うことができます。cpulimit_gracetime で設定された CPU 時間が経過してもなおプロセスが実行中の場合、カーネルでは、SIGKILL シグナルを使用してそのプロセスを終了します。

ジョブ制限

標準のシステム・リソース制限を設定することにより、プロセス作成時にプロセスベースの同じ制限を各プロセスに適用できます。プロセスを個々に制限する方法は便利ですが、個々のユーザが使用するリソースを任意に制限できるわけではありません。IRIX カーネルのジョブ制限機能では、特定のログイン・セッションやバッチ実行に関連する全プロセスが、「ジョブ」と呼ばれる単一の論理単位にカプセル化されます。ジョブは、プロセスをログイン・セッションごとにグループ化するのに使用されるコンテナです。リソース使用量の制限は、特定のジョブに対してユーザごとに適用され、この制限はカーネルによって強制的に適用されます。すべてのプロセスは特定のジョブに関連付けられ、一意のジョブ識別子（ジョブ ID）で識別されます。特定のジョブに属する複数のプロセスは、1 つの単位として制限、制御、照会、および考慮できます。これにより、システム管理者は、CPU 時間、メモリ、ファイル容量、およびその他のシステム・リソースに対してジョブ固有の制限を設定できます。ユーザ制限データベース (ULDB: User Limits Database) を使用すると、ユーザ固有のジョブ制限を設定できます。ULDB が未定義の場合、ジョブ制限はすべてのジョブで同じです。ジョブ制限ソフトウェアを使用することで、マルチユーザ環境の大規模システムの利用率を向上できます。

メモ: ジョブ制限の値 (`rlim_t`) は、n32 バイナリと n64 バイナリの両方で 64 ビットです。そのため、n32 バイナリで 64 ビット制限を設定できます。`rlim_t` は o32 バイナリでは 32 ビットなので、o32 バイナリで 64 ビット制限を設定することはできません。IRIX では、o32、n64、および n32 の 3 つのアプリケーション・バイナリ・インタフェース (ABI: Application Binary Interface) をサポートします（ABI についての詳細は、`abi`(5) のマン・ページを参照してください）。

`rlimit_*` の値についての詳細は、2 ページの「systune を使用したプロセス制限の表示と設定」および 18 ページの「`showlimits`」を参照してください。

この章は、以下の節で構成されています。

- 27 ページの「エラー・メッセージ」

最初にお読みください

この章の各節では、お使いのシステムにジョブ制限ソフトウェアをインストールする方法について説明します。以下の順序で参照してください。

1. ジョブおよびジョブ制限の一般的な説明については、6 ページの「ジョブ制限の概要」および 9 ページの「サポートされているジョブ制限」を参照してください。
2. ジョブ制限パッケージのインストール方法については、25 ページの「ジョブ制限のインストール」を参照してください。
3. ユーザ制限命令入力ファイル *infile* の記述およびユーザ制限データベース (ULDB: User Limits Database) の作成についての詳細は、13 ページの「ユーザ制限命令入力ファイルの作成」および 12 ページの「ユーザ制限データベースの作成」をそれぞれ参照してください。

   ジョブ制限に関連するマン・ページのリストについては、26 ページの「ジョブ制限に関するマン・ページ」を参照してください。
4. `systune` *joblimits* コマンドを使用したシステム全体のジョブ制限のデフォルト値の設定についての詳細は、17 ページの「 `systune` を使用したジョブ制限の表示と設定」を参照してください。
5. システムのジョブ制限の表示についての詳細は、18 ページの「ジョブ制限の表示と設定用のユーザ・コマンド」を参照してください。
6. ジョブ制限のインストールに関するトラブルシューティングについての詳細は、26 ページの「ジョブ制限のトラブルシューティング」を参照してください。
7. アプリケーション・プログラミング・インタフェース (API: Application Programming Interface) についての詳細は、169ページの「ジョブ制限のアプリケーション・プログラミング・インタフェース (API: Application Programming Interface)」および173ページの「ULDB の API」を参照してください。

ジョブ制限の概要

ジョブ制限ソフトウェアを使用すると、各ユーザが適切な量のシステム・リソース（CPU 時間やメモリなど）にアクセスでき、それぞれの割当て量を超えないようにするといった設定が容易に行えます。ジョブ制限ソフトウェアで各ユーザのマシン使用量を制限することにより、システムのスループットや利用率を向上できます。IRIX でサポートされているユーザベースのジョブ制限についての詳細は、9 ページの「サポートされているジョブ制限」を参照してください。

システムの処理はさまざまな方法で起動されます。たとえば、端末からのログイン、ワークロード管理システムからの実行、cron ジョブ、または rsh、rcp、array サービスといったリモート・アクセスなどです。これらの各エントリ・ポイントによって元のシェル・プロセスが作成され、その元のエントリ・ポイントから複数のプロセスが生じます。カーネル・ジョブは、エントリ・ポイントで生じるすべてのプロセスのリソース使用量を制限する方法を提供します。ジョブはエントリ・ポイントのプロセスを祖先とするすべての関連プロセスのグループで、一意なジョブ ID で識別されます。ジョブは複数のプロセス・グループ、セッション、または array セッションから構成され、これらのいずれかのサブグループに含まれるプロセスは、すべて 1 つのジョブ内に常に含まれます。7 ページの図2-1 に、ジョブの作成を開始するエントリ・ポイントのプロセスを示します。

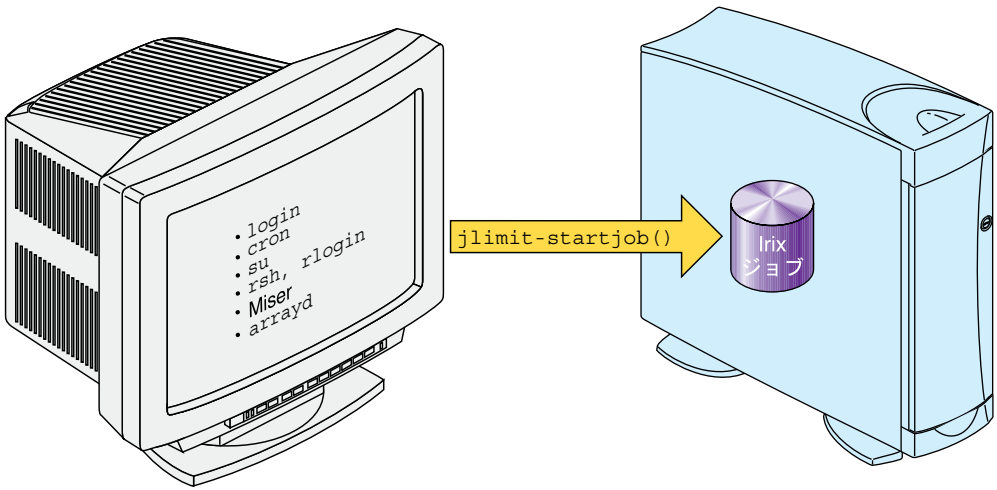

図2-1 エントリ・ポイントのプロセス

IRIX ジョブ制限には以下の特性があります。

- ジョブは、プロセスを囲込むためのコンテナです。プロセスは、ジョブの外部に出られません。また、明示的なアクション（root 特権でのシステム・コール）なしにジョブの外部に新しいプロセスを作成できません。
- 各新規プロセスでは、親プロセスからジョブ ID と制限が継承されます。
- すべてのエントリ・ポイントのプロセス（ジョブ・イニシエータ）では、新規ジョブが作成され、ジョブ制限が適切に設定されます。
- ユーザは、システム管理者が指定する最大値以下でそれぞれのジョブ制限を上下させることができます。

- ジョブ・イニシエータでは、認証とセキュリティ・チェックが行われます。

プロセス制御初期化プロセス (`init`(1M)) と、`init` によって呼出される起動スクリプトは、ジョブの一部ではなく、ジョブ ID はゼロになります。

メモ: ジョブ ID の上位ビットはマシン ID を示します。ジョブ ID には、array サービスのマシン ID (`asmchid`) が含まれます。array サービスは init プロセスによって開始され、大きなジョブ ID が作成されます。管理者にとっては、自分でマシン ID を設定していないので、大きなジョブ ID 値が何の説明もなく表示されるように思われます。`asmchid` パラメータについての詳細は、『IRIX Admin: System Configuration and Operation』の付録 A「IRIX Kernel Tunable Parameters」と、`arsctl`(2) および `newarraysess`(2) のマン・ページを参照してください。

メモ: 既存の IRIX コマンドである `jobs`(1)、`fg`(1)、および `bg`(1) のマン・ページは、シェルの「ジョブ」に関するもので、IRIX カーネルのジョブ制限とは関係ありません。

メモ: SGI 以外によって開発された SSH (Secure Shell) などのジョブ・イニシエータでは、IRIX カーネル・ジョブが起動されない場合があります。

図2-2 に 2 つの制限ドメインを示します。制限ドメインは、作業を分類するための 1 つの手法です。 7 ページの図2-1 に示すジョブ・イニシエータは、対話型プロセスまたはバッチ・プロセスに分類されます。制限ドメイン名は、ユーザ制限データベース (ULDB: User Limits Database) の作成時にシステム管理者が定義します。ULDB を使用してジョブ制限情報を取得するアプリケーションでは、特定の名前で制限情報を検索できることが想定されます。これらの名前は、規則に基づいて定義されます。制限ドメインおよび ULDB についての詳細は、12 ページの「ユーザ制限データベース (ULDB: User Limits Database)」を参照してください。

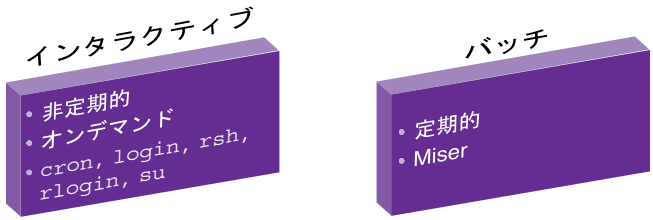

図2-2 制限ドメイン

IRIX オペレーティング・システムは、システムのメモリ使用量に関する情報を提供する多くのコマンドを備えています。`jstat`(1) ジョブ制限コマンドは、現在の使用量、およびジョブ内で同時に実行されて

いるすべてのプロセスの最大メモリ値を報告します。IRIX におけるメモリ使用量についての詳細は、125 ページの 第6章「IRIX のメモリ使用量」を参照してください。jstat(1) コマンドについての詳細は、22 ページの「jstat」を参照してください。

サポートされているジョブ制限

表2-1 に、IRIX オペレーティング・システムでサポートされているジョブ制限を示します。各制限項目を使用して、ジョブ内の全プロセスによる特定のシステム・リソースの使用を制限します。ジョブ制限ソフトウェアには、各ジョブ内のプロセス数を制御する JLIMIT_NUMPROC と呼ばれる制限も用意されています。

表2-1 ジョブ制限

制限名	シンボリック ID	単位	説明	違反時の動作
jlimit_nproc_cur jlimit_nproc_max	JLIMIT_NUMPROC	プロセス	ジョブ内のプロセスの最大数	errno が EAGAIN に設定され、ジョブによるプロセス作成が失敗する
jlimit_nofile_cur jlimit_nofile_max	JLIMIT_NOFILE	ファイル記述子	ジョブ内のすべてのプロセスで保持できる、開いているファイル記述子の合計の最大数	errno が EMFILE に設定され、ジョブによる open(2) の呼出しが失敗する
jlimit_rss_cur jlimit_rss_max	JLIMIT_RSS	バイト	ジョブ内の全プロセスに対する常駐セット・サイズの合計の最大値	制限を超えた常駐ページが最初のスワップ候補になる
jlimit_vmem_cur jlimit_vmem_max	JLIMIT_VMEM	バイト	ジョブ内の全プロセスに対するアドレス空間の合計の最大値	errno が ENOMEM に設定され、ジョブでの brk(2) と mmap(2) の呼出しが失敗する
jlimit_data_cur jlimit_data_max	JLIMIT_DATA	バイト	ジョブ内の全プロセスに対するヒープ・サイズの合計の最大値	ENOMEM が errno に設定され、ジョブによる brk(2) の呼出しが失敗する

制限名	シンボリック ID	単位	説明	違反時の動作
jlimit_cpu_cur jlimit_cpu_max	JLIMIT_CPU	秒	ジョブ内の全プロセスに許可される CPU 時間(秒)の合計の最大値	CPU 時間を消費し続けているジョブ内の全プロセスが SIGXCPU シグナルによって終了される。下記のメモを参照してください。cpulimit_gracetime パラメータを使用して、シグナルの動作を変更することもできます(11 ページの「cpulimit_gracetime」を参照してください)。
jlimit_pmem_cur jlimit_pmem_max	JLIMIT_PMEM	バイト	ジョブ内の全プロセスに対する常駐セット・サイズの合計の最大値	システム・リソースを消費し続けているジョブ内の全プロセスが SIGKILL シグナルによって終了される。下記のメモおよび 11 ページの「cpulimit_gracetime」を参照してください。

getjlimit および setjlimit

ジョブのシステム・リソース消費に関する制限(9 ページの表2-1 を参照)は、getjlimit(2) 関数で取得し、setjlimit(2) 関数で設定できます。getjlimit 関数は、指定したジョブのジョブ制限の現在の値と最大値を取得します。ほかのユーザが所有するジョブの値を取得するには、CAP_MAC_READ capability が必要です。

setjlimit(2) 関数は、指定したジョブのジョブ制限の現在の値と最大値を設定します。現在のジョブが、設定しようとするジョブと異なる場合、setjlimit 関数は CAP_MAC_WRITE capability をチェックします。最大(ハード)制限を上げようとしている場合、setjlimit 関数は CAP_PROC_MGT capability をチェックします。

詳細については、getjlimit(2) のマン・ページを参照してください。プロセスの操作権に対する細かな調整を可能にする capability についての詳細は、capability(4) および capabilities(4) のマン・ページを参照してください。

waitjob

waitjob を使用すると、バッチ処理システムで、異常終了したジョブのジョブ制限情報を検索できます。waitjob 関数は、setwaitjobpid 引数を使用して待機するよう設定された停止ジョブに関する情報を取得します。waitjob(2) と setwaitjobpid(2) の呼出しについての詳細は、169ページの「ジョブ制限のアプリケーション・プログラミング・インタフェース (API: Application Programming Interface)」および173ページの「ULDB の API」と、waitjob(2) および setwaitjobpid(2) のマン・ページをそれぞれ参照してください。

systune

systune *joblimits* コマンドを使用して、システム全体のデフォルト値を設定できます。詳細については、17 ページの「 systune を使用したジョブ制限の表示と設定」および systune(1M) のマン・ページを参照してください。

cpulimit_gracetime

cpulimit_gracetime パラメータは、CPU 時間の制限を超えたプロセスの猶予期間を指定します。ジョブ内の各プロセスには、cpulimit_gracetime が関連付けられています。cpulimit_gracetime パラメータが 10 秒に設定されていて、ジョブが 100 のプロセスを持っている場合、理論的には、JLIMIT_CPU 制限を超えた後、さらに 1000 秒ジョブを実行できます。cpulimit_gracetime パラメータは、CPU 制限に関連付けられているシグナルの動作を制御します。cpulimit_gracetime パラメータについての詳細は、4 ページの「追加のプロセス制限パラメータ」を参照してください。

cpulimit_gracetime の処理については、ジョブ制限ソフトウェアの動作はプロセス制限に似ています。プロセスが実行されると CPU 使用量が増加します。制限値に達すると、SIGXCPU シグナルが各プロセスに対してプロセス実行時に個別に送信されます。SIGXCPU がプロセスに送信されると、そのプロセスで猶予期間が有効になります。猶予期間が期限切れになった時点でまだプロセスが実行中である場合、プロセスは SIGKILL シグナルによって終了されます。ジョブ内で実行されているプロセスに対してのみ SIGXCPU シグナルが送信されます。ジョブ内の各プロセスには個々の猶予期間があります。そのため、SIGXCPU シグナルはジョブ内の全プロセスにまとめて送信されません。

メモ: SIGXCPU または SIGKILL シグナルをそれぞれ受信するプロセスは、クロック割込みが発生し、JLIMIT_CPU または JLIMIT_PMEM 制限を超えたときに CPU 時間やメモリなどのシステム・リソースを実行および消費しているジョブ内のプロセスのみです。待ち状態のジョブ内のプロセスは、制限を超えてもシグナルを受信しない可能性があります。

ユーザ制限データベース (ULDB: User Limits Database)

ユーザ制限データベース (ULDB: User Limits Database) にはジョブ制限の情報が含まれており、システム管理者はこの情報を使用してマシンへのアクセスをユーザ単位で制御できます。ジョブ・イニシエータ、つまり、新規のジョブをシステム上で起動するアプリケーション(`login`、`rsh`、`rlogin`、`cron`、または Miser などのワークロード管理システムなど)では、特定のユーザのジョブ制限値が ULDB から取得され、その情報を使用して適切に制限が設定されます。

ジョブ・イニシエータについての詳細は、6 ページの「ジョブ制限の概要」を参照してください。

ジョブ制限パッケージがインストールされている場合、ULDB を使用して、ジョブのジョブ制限とプロセス制限の値が設定されます。ジョブ制限がインストールされていない場合、現在のリソース制限機能によりプロセス制限が処理されます。

特定のユーザ用の値を記述した「user」エントリがない場合、ドメインのデフォルトがすべてのユーザに適用されます。ユーザ固有の値は、ドメインのデフォルト値に優先します。ULDB の値は、ジョブ制限とプロセス制限の両方のシステム・デフォルト値をオーバーライドします。

この節では、ULDB の内容の作成、保守、および表示に使用するコマンドと、アプリケーションから ULDB 情報にアクセスするためのライブラリの API について説明します。

メモ: `/etc/jlimits.in` ファイルに含まれている ULDB 設定ファイルには、ULDB を設定する際に利用できるテンプレートが含まれています。

`/etc` ディレクトリには、`jlimits` ファイルと `jlimits.m` ファイルも含まれています。`jlimits.in` ファイルは、ジョブ制限をローカル ULDB の `jlimits.m` ファイルや NIS のマスタ・マップに読込む際に使用されるコロン区切りの `jlimits` ファイルにパースされます。`jlimits` ファイルは、`genlimits`(1M) コマンドによって自動的に生成されます。`jlimits.m` ファイルは、ローカル ULDB の mdbm ファイルです。

ユーザ制限データベースの作成

ULDB を作成するコマンドは、次のとおりです。

```
genlimits [-i infile] [-l] [-m] [-L local_database] [-N nisfile] [-v]
```

`genlimits` コマンドは、フォーマットされた ASCII ユーザ制限命令入力ファイル (*infile*) をコロン区切りの ASCII ファイルに変換します。このファイルを使用して、以下のいずれかの出力形式を作成できます。

- NIS (Network Information Service) マスタ・サーバ・マップ(`-m` オプション)
- NIS 用、または直接使用する(NIS 用ではない)ローカル・データベース(`-l` オプション)

genlimits コマンドでは、以下のオプションを使用できます。

-i *infile*	ユーザ制限命令入力ファイルの場所を指定します。-i オプションを指定しなかった場合、デフォルトのファイルは /etc/jlimits.in です。
-l	NIS 用、または直接使用する(NIS 用ではない)ローカル・データベースを作成します。NIS が有効な場合、ローカル・データベースには、NIS サーバのエントリをオーバーライドするか、NIS サーバのエントリを補足するローカルのエントリが含まれます。NIS が有効でない場合、ローカル・データベースには、システムでの制限を設定する情報が含まれます。デフォルトでは、このデータベースは /etc/jlimits.m ファイルに含まれています。-l オプションは、-m オプションと同時には使用できません。
-m	NIS マスタ・サーバ・マップを作成します。標準の NIS マップの保存場所にマップを生成し、保存します。NIS マップ・ファイルの保存場所は変更できません。-m オプションは、-l オプションと同時には使用できません。
-L *local_database*	ローカル・データベースの別の保存場所を指定します。-L オプションは -l オプションと組合わせて使用します。
-N *nisfile*	NIS データベースの作成済みソース入力ファイルの別の保存場所を指定します。デフォルトの保存場所は、/etc/jlimits ファイルです。-N *nisfile* オプションを使用すると、既存の /etc/jlimits ファイルを上書きせずにデータベースを新規作成できます。
-v	genlimits コマンドの動作を示すメッセージを出力する冗長モードを指定します。

詳細については、genlimits(1M) のマン・ページを参照してください。

ユーザ制限命令入力ファイルの作成

ユーザ制限命令ファイルには、ULDB の生成に使用するドメイン、制限、およびユーザの情報を定義する genlimits(1M) コマンドへの入力が含まれます。この節では、ユーザ制限命令入力ファイルの記述方法について説明します。

コメント

コメント記号 (#) に続くテキストは、すべてコメントとして扱われます。

数値の制限値

以下のように、数値に文字を付加して、制限値を決定する数値に適用する乗数を表すことができます。

文字	乗数値
k（キロ）	1024 (2**10)
m（メガ）	1,048,576 (2**20)
g（ギガ）	1,073,741,824 (2**30)
t（テラ）	1,099,511,627,776 (2**40)
H（時）	3600
M（分）	60

- k、m、g、および t の乗数は、メモリの制限やその他の大きな値を定義する際に使用します。
- H と M の乗数は、時間の値を定義する際に使用します。

乗数値は、システム・インクルード・ファイル `/usr/include/uldb.h` に定義されています。

上記の方法で乗数を使用する場合の前提条件はありません。

数値の制限値には、その特定の制限タイプに上限がないことを示す「unlimited」を指定することもできます。

ULDB の作成についての詳細は、`genlimits`(1M) のマン・ページを参照してください。

domain 命令

ULDB で参照される各制限ドメインは、命令「domain」で始める必要があります。この命令には、ASCII 文字で記述されたドメイン名と、ドメインに対するデフォルトの制限値のリストを指定します。domain 命令の例を以下に示します。

```
domain domain_name {
      limit_name = value
      limit_name:machname = value
      ...
   }
```

一部のドメイン名は、ユーザのジョブ制限用に予約されています。その他のドメイン名は、特定用途のために作成および使用することが可能です。次のリストに、予約されているドメイン名を示します。

予約されているドメイン名	説明
`interactive`	`telnet` や `login` などの対話型のジョブ・イニシエータで使用

batch	すべてのワークロード管理ソフトウェアの 2 番目の選択肢として使用される、汎用のバッチ・ドメイン
miser	処理を Miser に送るときに使用されるドメイン
nqe	処理を NQE に送るときに使用されるドメイン
lsf	処理を LSF に送るときに使用されるドメイン

user 命令

「user」命令は、個々のユーザに対する制限を設定します。ユーザ名には有効なログイン・アカウントを指定する必要があります。uid 値はオプションです。uid が指定されている場合、genlimits コマンドによって、指定された uid が、genlimits が実行されたマシンのユーザに定義された uid と一致するかどうかが検証されます。domain 節では、ユーザが一意の制限値を持つ各ドメインを指定します。user 命令にリストされるドメインは、user 命令の前に domain 命令ですでに定義されている必要があります。domain 節の構造体とその意味は、domain 命令と同じです。システムの全ユーザに対して user 命令を記述する必要はありません。照会したユーザに対する user 命令がない場合や、照会したドメインに対する値がない場合、そのドメインに対するデフォルト値が返されます。user 命令の例を以下に示します。

```
user user_name[:uid] {
      domain_name {
         limit_name = value
         limit_name:machname = value
         ...
      }
      domain_name {
         ...
      }
      ...
   }
```

domain 命令と user 命令の制限指定には、オプションでマシン名を含めることができます。マシン名を含めて指定した制限値は、該当するマシンに対してのみ適用されます。マシン名のない制限は、クラスタ内のすべてのマシンに適用されます。1 つの命令入力ファイルには、マシン名が指定されていない制限に加えて、それと同じ制限を複数回、それぞれ異なるマシン名を指定して記述できます。

genlimits コマンドでは、以下のように、マシン名が関連付けられた制限値の処理方法は、生成されるデータベースのタイプ（12 ページの「ユーザ制限データベースの作成」を参照）によって異なります。

- -m オプションを使用して NIS マスタ・マップを生成する場合、マシン名が関連付けられた制限値は無視されます。マシン名が指定されていない、クラスタ全体に適用される値だけがデータベースに含まれることになります。

- -l オプションを使用してローカル・データベースを生成する場合、genlimits コマンドでは、ローカル・マシン名が指定された制限値(存在する場合)が選択されます。ローカル・マシン名が指定された制限値がない場合、genlimits コマンドでは、マシン名が指定されていない、クラスタ全体に適用される値が選択されます。ローカル・マシン名を確認するには、uname -n コマンドを実行します。uname コマンドについての詳細は、uname(1) のマン・ページを参照してください。

ユーザ制限命令入力ファイルの設定例

genlimits コマンドを実行すると ULDB は完全に再構築されるため、入力命令ファイルには、データベースに必要な情報がすべて含まれる必要があります。情報の変更が必要な場合、システム管理者はユーザ制限命令入力ファイルを編集し、データベースを再構築する必要があります。特定のユーザに対する user エントリがない場合はドメインのデフォルトが使用されるため、管理者は、必要なユーザに対してのみ名前を指定した user エントリを作成してデフォルト値を上書きする必要があります。以下に、3 つの制限タイプ、2 つのドメイン、および個々の制限のある 1 ユーザを指定した、ユーザ制限命令入力ファイルの例を示します。ULDB には、制限値のみが保存されます。値の意味とその単位は、制限を使用するアプリケーションに依存します。

メモ: ULDB のエントリを更新しても、システムにおけるジョブ制限値が変更されない場合は、ULDB 内で使用される制限名と systune *joblimits* グループで使用される制限名が完全に一致していることを確認してください。詳細については、26 ページの「ジョブ制限のトラブルシューティング」を参照してください。

```
domain interactive {              # domain for interactive logins
   jlimit_cpu_cur = 60
   jlimit_cpu_max = 120           # limit interactive jobs to 120 CPU seconds
   jlimit_vmem_cur = 2m
   jlimit_vmem_max = 4m           # limit interactive jobs to 4 megabytes of virtual memory
   jlimit_numproc_cur =10
   jlimit_numproc_max = 20        # limit interactive jobs to 20 concurrent processes
}
domain batch {                    # domain for batch submissions
   jlimit_cpu_cur = 3600
   jlimit_cpu_max = 7200          # limit batch jobs to two hours of CPU time
   jlimit_vmem_cur = 128m
   jlimit_vmem_max = 256m         # limit batch jobs to 256 megabytes of memory
   jlimit_numproc_cur = unlimited
   jlimit_numproc_max = unlimited # no limit on processes in a batch job
}

user fred:123 {                   # User "fred" gets his own interactive CPU limits
   interactive  {                 #
      jlimit_cpu_cur = 300
```

```
      jlimit_cpu_max = 600        # "fred" needs to run longer jobs in interactive mode
    }
}
```

systune を使用したジョブ制限の表示と設定

systune *joblimits* コマンドを使用して、ユーザのジョブ制限のシステム全体のデフォルト値を表示および設定できます。ULDB が存在する場合、ULDB の値はこれらの値に優先します。*joblimits* グループには、以下の変数が含まれます。

```
jlimit_cpu_cur
jlimit_cpu_max
jlimit_data_cur
jlimit_data_max
jlimit_vmem_cur
jlimit_vmem_max
jlimit_rss_cur
jlimit_rss_max
jlimit_nofile_cur
jlimit_nofile_max
jlimit_numproc_cur
jlimit_numproc_max
jlimit_pmem_cur
jlimit_pmem_max
```

systune *joblimits* コマンドからの出力を以下に示します。

```
$ systune joblimits
group: joblimits (statically changeable)
        jlimit_numproc_max = 1024 (0x400) ll
        jlimit_numproc_cur = 1024 (0x400) ll
        jlimit_nofile_max = 5000 (0x1388) ll
        jlimit_nofile_cur = 400 (0x190) ll
        jlimit_rss_max = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_rss_cur = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_vmem_max = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_vmem_cur = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_data_max = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_data_cur = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_cpu_max = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_cpu_cur = 9223372036854775807 (0x7fffffffffffffff) ll
        jlimit_pmem_max = 9223372036854775807 (0x7fffffffffffffff) ll
```

```
jlimit_pmem_cur = 9223372036854775807 (0x7fffffffffffffff) ll
```

表示情報の内容は以下のとおりです。

- jlimit_numproc - プロセス数の制限
- jlimit_nofile - ファイル数の制限
- jlimit_rss - 常駐セット・サイズ（デフォルトはバイト単位）
- jlimit_vmem - 仮想メモリの制限（デフォルトはバイト単位）
- jlimit_data – データ・サイズ（デフォルトはバイト単位）
- jlimit_cpu - CPU 時間（デフォルトは秒単位）
- jlimit_pmem - ジョブ内の全プロセスに対する常駐セット・サイズの最大値（デフォルトはバイト単位）

詳細については、systune(1M) および jlimit(1) のマン・ページを参照してください。

ジョブ制限の表示と設定用のユーザ・コマンド

この節では、ジョブ制限の表示と設定に使用する以下のユーザ・コマンドについて説明します。

- 18 ページの「showlimits」
- 21 ページの「jlimit」
- 22 ページの「jstat」

showlimits

ULDB の制限情報を表示するコマンドは、次のとおりです。

showlimits [-D] [-d] [-u *user_name*] [*domain_name*]

showlimits コマンドは、ユーザ制限データベース (ULDB: User Limits Database) の制限情報を表示します。

showlimits コマンドでは、以下のオプションを使用できます。

-D　ULDB 内に定義されたすべてのドメイン名を表示します。-D オプションを指定すると、ドメイン名とその他のオプションは無視されます。

`-d`	ドメインのデフォルトの制限を表示します。オプションが指定されていない場合、`showlimits` コマンドは、すべてのドメインに対するデフォルトの制限を表示します。
`-u` *user_name*	現在のユーザの代わりに、指定されたユーザに対する制限値を表示します。
domain_name	すべてのドメインの代わりに、指定されたドメインに対する制限値を表示します。

オプションが指定されていない場合、`showlimits` コマンドは、以下のようにすべてのドメインでの現在のユーザに対する現在の制限情報を表示します。

```
% showlimits

Domain interactive:
        jlimit_cpu_cur: unlimited
        jlimit_cpu_max: unlimited
        jlimit_data_cur: unlimited
        jlimit_data_max: unlimited
        jlimit_nofile_cur: 400
        jlimit_nofile_max: unlimited
        jlimit_vmem_cur: unlimited
        jlimit_vmem_max: unlimited
        jlimit_rss_cur: unlimited
        jlimit_rss_max: unlimited
        jlimit_pthread_cur: 2k
        jlimit_pthread_max: 65535
        jlimit_numproc_cur: 1k
        jlimit_numproc_max: 65535
        rlimit_cpu_cur: unlimited
        rlimit_cpu_max: unlimited
        rlimit_fsize_cur: unlimited
        rlimit_fsize_max: unlimited
        rlimit_data_max: unlimited
        rlimit_stack_cur: 64m
        rlimit_stack_max: unlimited
        rlimit_core_cur: unlimited
        rlimit_core_max: unlimited
        rlimit_nofile_cur: 200
        rlimit_nofile_max: unlimited
        rlimit_vmem_max: unlimited
        rlimit_rss_max: unlimited
```

```
Domain batch:
        jlimit_cpu_cur: unlimited
        jlimit_cpu_max: unlimited
        jlimit_data_cur: unlimited
        jlimit_data_max: unlimited
        jlimit_nofile_cur: 400
        jlimit_nofile_max: unlimited
        jlimit_vmem_cur: unlimited
        jlimit_vmem_max: unlimited
        jlimit_rss_cur: unlimited
        jlimit_rss_max: unlimited
        jlimit_pthread_cur: 2k
        jlimit_pthread_max: 65535
        jlimit_numproc_cur: 1k
        jlimit_numproc_max: 65535
        rlimit_cpu_cur: unlimited
        rlimit_cpu_max: unlimited
        rlimit_fsize_cur: unlimited
        rlimit_fsize_max: unlimited
        rlimit_data_max: unlimited
        rlimit_stack_cur: 64m
        rlimit_stack_max: unlimited
        rlimit_core_cur: unlimited
        rlimit_core_max: unlimited
        rlimit_nofile_cur: 200
        rlimit_nofile_max: unlimited
        rlimit_vmem_max: unlimited
        rlimit_rss_max: unlimited
```

メモ: ユーザがログインした後で ULDB が変更された場合、現在の制限は有効にはなりません。現在の制限は、新たにログインしたユーザに対して有効になります。

ジョブ制限値の説明については、9 ページの表2-1 を参照してください。プロセス制限値の説明については、2 ページの表1-1 を参照してください。

詳細については、`showlimits`(1) のマン・ページを参照してください。

jlimit

ジョブ制限の表示と設定を行うコマンドは、次のとおりです。

```
jlimit [-j job_id] [-h] [limit_name [value]]
```

jlimit コマンドは、ジョブのリソース使用量に対する制限を表示および変更します。ユーザのユーザ制限データベース (ULDB: User Limits Database) 情報に含まれる値を使用して、ジョブの開始時に現在の制限と最大(ハード)制限が設定されます。最大制限を超えない範囲で、現在の制限を上下させることができます。また、最大制限を下げることはできますが、後で元に戻すことはできません。最大制限を上げるには、CAP_PROC_MGT capability が必要です。最大制限の値にかかわらず、現在の制限に達したときには、常に制限の違反時の動作が発生します。プロセスの操作権に対する細かな調整を可能にする capability についての詳細は、capability(4) および capabilities(4) のマン・ページを参照してください。

jlimit コマンドでは、以下のオプションを使用できます。

-j *job_id*	制限を変更するジョブのジョブ ID を指定します。ほかのユーザに属するジョブのジョブ制限を変更するには、CAP_MAC_WRITE および CAP_PROC_MGT capability が必要です。ジョブ ID は、16 進で出力されます。ジョブ ID を指定するとき、プレフィックス「0x」はオプションです。
-h	ジョブの最大(ハード)制限値を表示または変更することを指定します。-h オプションを指定しなかった場合、jlimit コマンドは、現在の制限値を表示または変更します。
limit_name [*value*]	指定された制限に対する値を設定または表示します。 • 制限名が指定されていない場合、jlimit は、すべての制限の値を表示します。 • 制限名が値なしで指定されている場合、 jlimit は、制限の値を表示します。 • 制限名と値の両方が指定されている場合、 jlimit は、該当する制限の値を設定します。

-j オプションと引数 *job_id* が指定されている場合、jlimit コマンドは、以下の情報を出力します。

```
 % jlimit -j 0x14
cputime: unlimited
datasize: unlimited
files: unlimited
vmemory: unlimited
ressetsize: unlimited
processes: 65535
```

制限値の説明については、9 ページの表2-1 を参照してください。

詳細については、jlimit(1) のマン・ページを参照してください。

jstat

アクティブなジョブのステータス情報を表示するコマンドは、次のとおりです。

```
jstat [-a] [-l] [-p]
jstat [-j job_id] [-l] [-p]
```

jstat コマンドでは、以下のオプションを使用できます。

-a	すべてのジョブの情報を表示します。
-j *job_id*	指定したジョブ ID (*job_id*) の情報のみを表示します。
-l	現在のジョブまたは指定したジョブに関する制限情報(現在の使用量、現在の制限、最大制限など)を表示します。
-p	現在のジョブまたは指定したジョブに属する各プロセスの情報(プロセス ID、状態、実行コマンドなど)を表示します。
-P	メモリ制限情報を、バイト単位ではなくページ単位で表示します。このオプションは、-l オプションと併せて使用します。

-a または -j *job_id* のいずれも使用されていない場合、jstat コマンドは、現在のジョブの情報を表示します。

-l オプションが指定されている場合、jstat コマンドは、以下のように、現在のジョブに関する現在の使用量、最大使用量、現在の制限、および最大制限の情報を表示します。

```
% jstat -l

JID              OWNER          COMMAND
---------------- -------------- --------------
0x5eac0000001bd terry             -csh

LIMIT NAME      USAGE          HIGH USAGE     CURRENT LIMIT  MAX LIMIT
--------------- -------------- -------------- -------------- --------------
cputime         1:05           1:05           unlimited      unlimited
datasize        400k           400k           unlimited      unlimited
files           10             35             400            5000
vmemory         44             201            unlimited      unlimited
ressetsize      340            357            unlimited      unlimited
processes       2              4              1024           1024
```

-l オプションと -P オプションが指定されている場合、jstat コマンドは、-l オプションだけが指定された場合と同じメモリ情報をページ単位で表示します。SGI システムでは、複数のページ・サイズがサポートされます。ページ・サイズについての詳細は、『IRIX Admin: System Configuration and Operation』の第 10 章「System Performance Tuning」の「Multiple Page Sizes」の節を参照してください。

要約情報は常に出力されます。制限値の説明については、9 ページの表2-1 を参照してください。

詳細については、jstat(1) のマン・ページを参照してください。

ジョブ制限と既存の IRIX ソフトウェア

ps -j コマンドは、プロセス ID、プロセス・グループ ID、セッション ID、およびジョブ ID を 16 進で出力します。

```
% ps -j
    PID    PGID     SID     JID TTY      TIME CMD
 253430  253430  253430  0x5eac001bd ttyq12  0:00 csh
 254563  254563  253430  0x5eac001bd ttyq12  0:00 ps
```

詳細については、ps(1) のマン・ページを参照してください。

クラスタ内のほかのマシン上でジョブの新規プロセスが開始されると、array サービス・デーモン (arrayd(1M)) によって、ジョブ ID が起動元のマシンからほかのマシンに伝えられます。

詳細については、arrayd(1M) のマン・ページを参照してください。

cpr(1) コマンドを使用して、システム再起動状態ファイルにジョブ情報を含めることができます。cpr -p オプションには、JID チェックポイント・タイプが追加されています。この JID タイプによって、ジョブ全体をチェックポイントおよび再開できます。以下に例を示します。

```
% cpr -c ckpt02 -p 0x8000000000001234:JID
```

この例では、ジョブ ID 0x8000000000001234 のジョブに含まれるすべてのプロセスを、状態ファイルのディレクトリ /ckpt02 にチェックポイントしています。

詳細については、cpr(1) のマン・ページを参照してください。

システムにジョブ制限ソフトウェアがインストールされている場合、リモート・シェル・サーバ (rshd(1M)) とリモート実行サーバ (rexecd(1M)) を介して開始されるジョブで SIGXCPU シグナルが認識されるようにするには、/etc/default/rshd ファイルと /etc/default/rexecd ファイルをそれぞれ更新する必要があります。SVR4_SIGNALS パラメータは NO に設定する必要があります。これによって、rshd サーバと rexecd サーバで SIGXCPU シグナルが認識されます。

詳細については、rsh(1M) および rexecd(1M) のマン・ページを参照してください。

MPI (Message Passing Interface) ジョブでのジョブ制限の実行

MPI (Message Passing Interface) ジョブには、多数のファイル記述子が必要です。デフォルトでは、files 制限に対するジョブの現在の制限は、400 に設定されています。-l オプションを指定して jstat コマンドを実行すると、以下のように表示されます。

```
% jstat -l

JID                 OWNER          COMMAND
------------------- -------------- --------------
0x23fc000000000035 user           -csh

LIMIT NAME          USAGE          HIGH USAGE     CURRENT LIMIT  MAX LIMIT
------------------- -------------- -------------- -------------- --------------
cputime             0              0              unlimited      unlimited
datasize            80k            208k           unlimited      unlimited
files               8              28             400            5000
vmemory             2384k          9824k          unlimited      unlimited
ressetsize          608k           2320k          unlimited      unlimited
threads             1              1              2048           2048
processes           2              6              1024           1024
physmem             608k           2320k          unlimited      unlimited
```

CPU が 16 個以上あるシステムで MPI ジョブを実行した場合、デフォルトで 400 に設定されている `files` の現在の制限にすぐに到達し、次のようなエラー・メッセージが発行されます。

```
MPI jobs fail with the error MPI: fork_slaves/fork: Resource temporarily unavailable
MPI: daemon terminated: mice1 - job aborting
```

このエラーを回避するには、MPI ジョブを実行するときに、`files` 制限のデフォルトの現在の制限を高くします。システムのジョブ制限の設定についての詳細は、12 ページの「ユーザ制限データベース (ULDB: User Limits Database)」および 17 ページの「 systune を使用したジョブ制限の表示と設定」を参照してください。

以下の表に、大規模な MPI ジョブを実行するときに推奨される `files` 制限のデフォルトの現在の制限を、システム内の CPU 数別に示します。推奨される設定は、概算される値です。

CPU の数	デフォルトの現在の制限 (この値以上を推奨)
16	351
17	380
18	410
20	472
25	648
30	848
50	4448

ジョブ制限のインストール

カーネル・ジョブ制限ソフトウェアをインストールするには、ソフトウェア・インストール・ツール `inst`(1M) またはソフトウェア管理ツール `swmgr`(1M) を使用します。`inst`(1M) および `swmgr`(1M) についての詳細は、IRIX Admin マニュアル・セットの『IRIX Admin: Software Installation and Licensing』と、それぞれのマン・ページを参照してください。

カーネル・ジョブ制限ソフトウェアを IRIX システムにインストールするには、サブシステム `eoe.sw.jlimits` をインストールします。

ジョブ制限ソフトウェアをインストールした後、`autoconfig`(1M) コマンドを実行してシステムを再起動します。

ジョブ制限を無効にするには、`eoe.sw.jlimits` ソフトウェア・モジュールをアンインストールし、システムを再起動する必要があります。

ジョブ制限のトラブルシューティング

ULDB のエントリを更新しても、システムにおけるジョブ制限値が変更されない場合は、ULDB 内で使用される制限名と `systune` *joblimits* グループで使用される制限名が完全に一致していることを確認してください。ULDB では、どのジョブ制限変数が有効であり、どのジョブ制限変数が無効であるかは判断できません。ULDB のシンボリック名が正しく入力されていない場合、`systune` *joblimits* グループからの値が適用されます。制限名についての詳細は、9 ページの表2-1 を参照してください。

ジョブ制限に関するマン・ページ

`man` コマンドは、すべてのリソース管理コマンドに関するオンライン・ヘルプを提供します。マン・ページをオンラインで表示するには、「`man` *commandname*」と入力します。

一般ユーザ用マン・ページ

ジョブ制限ソフトウェアでは、以下の一般ユーザ用マン・ページが用意されています。

一般ユーザ用マン・ページ	説明
`jlimit`(1)	リソース制限を表示および設定します。
`jstat`(1)	ジョブのステータス情報を表示します。
`showlimits`(1)	ユーザ制限データベース (ULDB: User Limits Database) の制限情報を表示します。

管理者用マン・ページ

ジョブ制限ソフトウェアでは、以下の管理者用マン・ページが用意されています。

管理者用マン・ページ	説明
`genlimits`(1M)	ユーザ制限データベースを作成します。

アプリケーション・インタフェースに関するマン・ページ

ジョブ制限ソフトウェアを使用するアプリケーションの開発者向けに、以下のオンライン・マン・ページが用意されています。

アプリケーション・インタフェースに関するマン・ページ	説明
`getjid`(2)	ジョブ ID を取得します。

`getjlimit`(2)	ジョブの最大システム・リソース消費量を制御します。
`getjusage`(2)	ジョブの使用状況に関する情報を取得します。
`killjob`(2)	指定したジョブのすべてのプロセスを終了します。
`jlimit_startjob`(3c)	新しいジョブを作成します。
`makenewjob`(2)	新しいジョブ・コンテナを作成します。
`setjusage`(2)	指定したジョブ ID のリソース使用量の値を更新します。
`setwaitjobpid`(2)	指定したプロセス ID (PID: Process ID) が `waitjob`(2) 関数を呼出すのを待つようにジョブを設定します。
`waitjob`(2)	終了したジョブに関する情報を取得します。
`uldb_get_limit_values`(3c)	ユーザ制限データベース (ULDB: User Limits Database) と対話して、ドメインまたはユーザの制限値を取得または設定する関数の集合。

エラー・メッセージ

ジョブ制限に関して返されるエラー・メッセージは、以下のとおりです。

EBUSY	要求されたジョブ ID は使用中です。
EINVAL	無効なパラメータです。
ENOATTR	ドメイン名またはドメイン名リストが指定されていません。
ENOEXIST	`jlimits` ファイルが存在しません。
ENOJOB	指定されたジョブ ID のジョブが見つかりません。
ENOMEM	十分なメモリが利用できません。
ENOPKG	ジョブ制限ソフトウェアがインストールされていません。

Miser バッチ処理システム

Miser は、時間や領域の必要量がわかっているアプリケーションの決定性バッチ・スケジュールを可能にするリソース管理機能で、システム・リソースの静的な分割が不要です。ジョブを指定すると、Miser は管理下の時間または領域のプール全体を検索し、そのジョブのリソース要件に最適な割当てを求めます。

Miser には、再起動せずにほとんどのパラメータを修正できる、管理用の拡張インタフェースがあります。Miser は、独立したトラステッド・プロセス(高信頼プロセス)として動作します。カーネルからかユーザからかを問わず、Miser への通信はすべて一連の Miser コマンドを通じて行われます。Miser は、プロセスのスケジュール、プロセスの状態変更、およびバッチ・システム設定の制御に対する要求を受信し、その要求の値とステータス情報を返します。

この章は、以下の節で構成されています。

最初にお読みください

この章の各節では、お使いのシステムに Miser ソフトウェアをインストールする方法について説明します。以下の順序で参照してください。

1. Miser の一般的な説明については、30 ページの「Miser の概要」を参照してください。
2. Miser パッケージのインストール方法については、41 ページの「Miser の有効化と無効化」を参照してください。
3. Miser キューの設定方法についての詳細は、32 ページの「Miser の設定」を参照してください。
4. Miser ジョブの指定についての詳細は、42 ページの「Miser ジョブの指定」を参照してください。
5. Miser のマン・ページについての詳細は、45 ページの「Miser のマン・ページ」を参照してください。

Miser の概要

Miser は、時間または領域のプールのセットを管理します。プールの時間コンポーネントは、Miser がどの程度先までジョブをスケジュールできるかを定義します。プールの領域コンポーネントは、ジョブをスケジュールできるリソースの集合です。領域コンポーネントは、時間によって変化します。

システム・プールは、Miser が利用できるリソース（CPU の数や物理メモリ）の集合を表します。ユーザ定義プールの集合は、ジョブをスケジュールできるリソースを表します。ユーザのプールが所有するリソースは、Miser が利用できるリソース合計を超えることはできません。Miser が管理するリソースが、スケジュールされたジョブによって使用されていないときは、Miser 以外のアプリケーションがそのリソースを利用できます。

各プールには、プール・リソースの定義、プールからリソースを割当てるジョブの集合、およびジョブのスケジュールを制御するポリシーが関連付けられています。リソース・プール、スケジュールされたジョブ、およびポリシーをまとめてキューと呼びます。

キューを使用することで、バッチ・システムをきめ細かく管理できます。キューに割当てるリソースは、時間に応じて変えることができます。たとえば、日中は 5 個、夜間は 20 個の CPU を管理するようキューを設定できます。複数のキューを使用すると、バッチ・システムのユーザ間でリソースを分割できます。たとえば、24 個の CPU があるシステム上で 2 つのキューを定義し、一方に 16 個の CPU、もう一方に 6 個の CPU を割当てることができます（2 個の CPU は Miser の管理外に置くものとします）。キューへのアクセスをシステム上の特定ユーザまたはユーザのグループに制限して、リソースの分割を強制できます。

ポリシーは、アプリケーションのリソース要求を満たす時間または領域のブロックを検索する方法を定義します。Miser には、「default」と「repack」の 2 つのポリシーがあります。default は、ファースト・フィット・ポリシーで、いったんジョブがスケジュールされると、その開始時刻と終了時刻は変更されません。先行するジョブがスケジュールよりも早く完了した場合でも、それ以後にスケジュールされているジョブの開始時刻と終了時刻には影響しません。一方 repack は、ファースト・フィット・ポリシーに加えて、スケジュールされたジョブの順序を保ちつつ、先行するジョブが早く終了した場合には、残りのジョブを前倒しするよう再スケジュールを行います。

ユーザは、`miser_submit` コマンドを使用して、ジョブをキューに入れることができます。このコマンドは、ジョブをアタッチするキューと、キューに対して実行するリソース要求を指定します。それぞれの Miser ジョブは、IRIX プロセス・グループです。リソース要求は、時間と領域の組です。時間は、単一の CPU で実行された場合の総 CPU 時間（wall-clock 時間）です。領域は論理 CPU 数と必要な物理メモリです。要求は Miser に渡され、Miser は、キューにアタッチされたポリシーを使用して、キューのリソースに対してジョブをスケジュールします。Miser は、ジョブの開始時刻と終了時刻をユーザに返します。

ジョブの開始時刻までは、ジョブはバッチ状態にあります。バッチ状態にあるジョブの優先度は、実行中のどのプロセスよりも低くなります。システムに待ち状態のリソースがあれば、バッチ状態にあるジョブを実行できます。これを「日和見的に実行される (run opportunistically)」といいます。指定した実行時刻になると、ジョブの状態はバッチ・クリティカルに変わり、その時点でジョブの優先度はどの非リアルタイム・プロセスよりも高くなります。バッチ状態の実行で消費した時間は、要求およびスケジュールされた時間にはカウントされません。プロセスがバッチ・クリティカル状態にある間、要求した物理メモリと CPU の

確保は保証されます。割当て時間を超えるか、要求した物理メモリよりも多くのメモリを使用すると、プロセスは終了します。

default ポリシーを使用してスケジュールされた、static フラグの指定があるジョブは、そのセグメントの実行がスケジュールされた時刻にのみ実行されます。待ち状態のリソースが利用可能でも、先行して実行されません。repack ポリシーを使用してスケジュールされているジョブは、先行して実行できます。

論理 CPU 数について

ジョブをスケジュールするとき、Miser は、ジョブ用にいくつかの CPU と一定量のメモリを予約するよう要求します。ジョブ用にこれらのリソースが予約された期間に達すると、Miser は、特定の CPU と一定量の論理スワップ・スペースをこのジョブ用に予約します。

ジョブへの CPU 割当てに影響を及ぼす問題がいくつかあります。ジョブがバッチ・クリティカルになると、Miser は、ノードが密集しているクラスタを探そうとします。そのようなクラスタが見つからない場合、そのジョブのスレッドは、利用可能な任意の空き CPU に割当てられます。これらの CPU は、システムの離れた位置に分散している場合もあります。

対話型プロセスにおける CPU 予約の効果

Miser の利点の 1 つは、CPU 予約の方法にあります。Miser は、物理的な CPU 数ではなく論理 CPU 数に基づいて CPU の制御と予約を行います。これにより、Miser は CPU リソースを柔軟に制御できます。

日和見的に実行される対話型とバッチ型のプロセスは、システム内で Miser のジョブ用に予約されていないすべての CPU を使用できます。新しいジョブが入ると、Miser は、利用できる論理的なリソースの量に基づいて、ジョブをスケジュールします。その結果、CPU が Miser によって予約され、対話型プロセスは新たに予約された CPU 上で実行できなくなります。ただし、リソースが Miser によって使用されていない場合、そのリソースは、ほかの任意のアプリケーションが自由に使用できます。Miser は、必要になったときにリソースを要求します。

Miser のメモリ管理について

CPU は必要に応じて Miser によって予約されますが、メモリは、必要になる前にあらかじめ予約しておく必要があります。

Miser は、起動時に、ジョブ用に予約できる CPU の数とメモリ量を通知されます。CPU の数は、論理数です。Miser のジョブがバッチ・クリティカルになると、CPU の集合が割当てられます。Miser のジョブで CPU が必要になるまでは（つまり、プロセスやスレッドが実行可能になるまでは）、システムの残りの部分がその CPU を利用できます。Miser のジョブのスレッドが実行を開始すると、現在の非 Miser スレッドはプリエンプト（実行の一時停止）され、Miser スレッドが現在実行されていない CPU 上で再開されます。

メモリ・リソースは、CPU リソースとまったく状況が異なります。Miser がジョブの予約に使用するメモリは、論理スワップ・スペースと呼ばれます。論理スワップ・スペースは、物理メモリ（カーネルによって占有される小さな領域）とすべてのスワップ・デバイスの総計として定義されます。

Miser の起動時にジョブに対してメモリを予約する必要があります。ただし、物理メモリを予約する必要はありません。非 Miser ジョブのメモリを移動するのに十分な物理メモリとスワップがあることだけが確認されます。これは、必要なメモリに等しい論理スワップを予約することで行われます。

Miser に対して指定されたジョブだけが、Miser 用に予約された論理スワップ・スペースの割当てを使用できます。Miser によって使用されていない物理メモリは、ほかの任意のアプリケーションが自由に使用できます。Miser は、必要になったときに物理メモリを要求します。

Miser の管理がユーザに及ぼす影響

ユーザが Miser にジョブを指定すると、要求された期間中、リソースの割当てがそのジョブ用に予約されます。ジョブは、システム・リソースを求めて競合する必要はありません。結果として、ジョブは、対話型ジョブとして実行する場合よりも、短時間で完了し、実行時間が安定します。ただしデメリットもあります。Miser はリソースの領域を共有するため、ジョブはスケジュールされた予約期間になってから、要求したリソースが予約されます。それよりも先に、非静的ジョブが空きを見て実行される場合があります。非静的ジョブは対話型の負荷と競合しますが、優先度は対話型の負荷よりも低くなります。

ユーザが対話的に作業している場合、そのユーザは、システム・リソースをすべて利用できるわけではありません。ユーザの対話型プロセスは、システム上で予約されていない CPU をすべて利用できますが、メモリ割当てには、限られた量の論理スワップ・スペースしか使用できません。非 Miser ジョブで利用できる論理スワップ・スペースの量は、Miser の起動時に予約されなかった量です。

Miser の設定

Miser で主として設定の対象となるのは、キューの集合です。Miser のキューは、Miser に割当てられるリソースを定義します。

Miser の設定は、以下のように行います。

- Miser システム・キュー定義ファイルを設定します。Miser システムには、必ず Miser システム・キュー定義ファイルが必要です。このファイルのベクトル定義は、ほかのキューのベクトル定義で利用可能な最大リソースを指定します。
- Miser ユーザ・キュー定義ファイルを設定してキューを定義します。
- Miser 設定ファイルを設定して Miser システムの一部となるすべてのキューを列挙します。

- Miser コマンド・ライン・オプション・ファイルを設定して、Miser が管理できる最大の CPU 数とメモリを定義します。

Miser システム・キュー定義ファイルの設定

Miser システム・キュー定義ファイル (`/etc/miser_system.conf`) は、システム・プールによって管理されるリソースを定義します。このファイルは、プールの最大持続時間を定義します。その他のすべてのキューは、システム・キュー以下でなければなりません。システム・キューは、ジョブが要求できるリソースの上限を指定します。Miser システム・キューは設定されている必要があります。

有効なトークンは、以下のとおりです。

POLICY *name*	システム・キューにはポリシーがないため、ポリシーは常に「none」です。
QUANTUM *time*	量子時間 (quantum) のサイズ。**quantum** は、任意の秒数を表す Miser の用語です。quantum を使用して、時間または領域のプールを分割する方法を指定します。これは、システム・キュー定義ファイルとユーザ・キュー定義ファイルの両方で指定し、両方のファイルで同じでなければなりません。
NSEG *number*	リソース・セグメントの数
SEGMENT	ベクトル定義の、新しいセグメントの先頭を定義します。それぞれの新しいセグメントは、*SEGMENT* トークンで始める必要があります。各セグメントには少なくとも CPU の数、メモリ、wall-clock 時間が必要です。
START *number*	セグメントが開始される量子時間。時刻の基点は、ローカル時間で 1970 年 1 月 1 日（木）の 00:00 です。 Miser は、現在の日付になるまでキューを前方に繰返すことにより、開始時刻と終了時刻を現在の時刻にマップします。たとえば、24 時間のキューは、常に現在の日付の午前零時から始まります。
END *number*	セグメントが終了する量子時間
NCPUS *number*	CPU の数
MEMORY *amount*	整数で指定するメモリの量。k（キロバイト）、m（メガバイト）、または g（ギガバイト）の単位を後に付けることができます。単位の指定がない場合は、デフォルトでバイト単位になります。

以下のシステム・キュー定義ファイルは、20 秒の量子時間と 1 つの要素をベクトル定義に持つキューを定義します。各セグメントの開始時刻と終了時刻は、秒単位ではなく量子時間単位で指定されています。

このセグメントは、1 個の CPU と 5 メガバイトのメモリがある、00:00 に開始して 00:20 に終了するリソース単位を定義しています。

```
POLICY none # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 1

SEGMENT
START 0
END 60# Number of quanta (20min*60sec) / 20
NCPUS 1
MEMORY 5m
```

Miser ユーザ・キュー定義ファイルの設定

Miser ユーザ・キュー定義ファイル (/etc/miser_default.conf) は、CPU、物理メモリ、ポリシー名、およびキューのリソース・プールを定義します。このファイルは、キューのポリシーを指定するヘッダ、リソース・セグメントの数、およびキューによって使用される量子時間で構成されます。

キューへのアクセスは、キュー定義ファイルのパーミッションによって管理されます。読込みパーミッションがあるユーザは、miser_qinfo コマンドを使用して、キューの内容を確認できます。実行パーミッションがあるユーザは、miser_submit コマンドを使用して、キューにジョブをスケジュールできます。書込みパーミッションがあるユーザは、miser_move コマンドと miser_reset コマンドを使用して、キューのリソースを変更できます。

デフォルトのユーザ・キュー定義ファイルは、ほかのユーザ・キュー定義ファイルのテンプレートとして使用できます。それぞれの Miser キューには、独立したキュー定義ファイルがあります。このファイル名は、全体的な Miser 設定ファイル (/etc/miser.conf) で指定します。

ユーザは、ユーザ・キューによって管理されるリソースに対してスケジュールを行います（システム・キューに対して行うのではありません）。ユーザ・キューによって指定される持続時間が、システム・キューによって指定される持続時間よりも短い場合、ユーザ・キューは何度も繰返されます（たとえば、システム・キューが 1 週間を指定し、ユーザ・キューが 24 時間を指定する場合など）。システム・キューがユーザ・キューで割切れない（たとえば、システム・キューが 6 で、ユーザ・キューが 5）場合は、ユーザ・キューは余りを切捨てた整数回数だけ繰返します。

有効なトークンは、以下のとおりです。

POLICY *name*	キューに入れられるアプリケーションをスケジュールするために使用するポリシーの名前。有効なポリシーは、「default」と「repack」の 2 つです。default はファースト・フィット・ポリシーで、いったんジョブがスケジュールされると、その開始時刻と終了時刻は変更されません。repack はスケジュールされたジョブの順序を維持しつつ、先行するジョブが早く終了した場合には、残りのジョブを前倒しするよう再スケジュールを行います。最初にジョブをスケジュールするときは、どちらのポリシーもファースト・フィット方式を使用する点に注意してください。
QUANTUM *time*	量子時間 (quantum) のサイズ。**quantum** は、任意の秒数を表す Miser の用語です。quantum を使用して、時間または領域のプールを分割する方法を指定します。これは、システム・キュー定義ファイルとユーザ・キュー定義ファイルの両方で指定し、両方のファイルで同じでなければなりません。
NSEG *number*	リソース・セグメントの数
SEGMENT	ベクトル定義の、新しいセグメントの先頭を定義します。それぞれの新しいセグメントは、*SEGMENT* トークンで始める必要があります。各セグメントには少なくとも CPU の数、メモリ、wall-clock 時間が必要です。
START *number*	セグメントが開始される量子時間。時刻の基点は、ローカル時間で 1970 年 1 月 1 日（木）の 00:00 です。 Miser は、現在の日付になるまでキューを前方に繰返すことにより、開始時刻と終了時刻を現在の時刻にマップします。たとえば、24 時間のキューは、常に現在の日付の午前零時から始まります。
END *number*	セグメントが終了する量子時間
NCPUS *number*	CPU の数
MEMORY *amount*	整数で指定するメモリの量。k（キロバイト）、m（メガバイト）、または g（ギガバイト）の単位を後に付けることができます。単位の指定がない場合は、デフォルトでバイト単位になります。

以下のシステム・キュー定義ファイルは、「default」という名前のポリシーを使用してキューを定義します。このキューは、20 秒の量子時間と 3 つの要素をベクトル定義に持っています。各セグメントの開始時刻と終了時刻は、秒単位ではなく量子時間単位で指定されています。

- 最初のセグメントは、50 個の CPU と 100 MBのメモリがある、00:00 に開始して 00:50 に終了するリソース単位を定義しています。
- 2 番目のセグメントは、50 個の CPU と 100 MBのメモリがある、00:51.67 に開始して 01:00 に終了するリソース単位を定義しています。

- 3 番目のセグメントは、50 個の CPU と 100 MBのメモリがある、01:02.00 に開始して 01:03.33 に終了するリソース単位を定義しています。

```
POLICY default
QUANTUM 20
NSEG 3

SEGMENT
START 0
END 150 (50min*60sec) / 20
NCPUS 50
MEMORY 100m

SEGMENT
START 155 ((51min*60sec)+67) / 20
END 185 (1h*60min*60sec) / 20
NCPUS 50
MEMORY 100m

SEGMENT
START 186 ((1h*60min*60sec)+(2min*60sec)) / 20
END 190 ((1h*60min*60sec)+(3min*60sec)+33sec) / 20
NCPUS 50
MEMORY 100m
```

Miser 設定ファイルの設定

Miser 設定ファイル (`/etc/miser.conf`) は、Miser キューの名前と、各キューに対するキュー定義ファイルのパス名をリストします。このファイルは、すべてのキュー名とそのキュー定義ファイルを列挙します。

すべての Miser 設定ファイルには、キューの 1 つとして、システム・プールのリソースを定義する Miser システム・キューを含める必要があります。Miser システム・キューは、「system」という名前で指定します。

有効なトークンは、以下のとおりです。

QUEUE *queue_name queue_definition_file_path*

> *queue_name* には、Miser へのインタフェースで使用するキューの名前を指定します。キュー名は、1 文字以上、8 文字以下でなければなりません。キュー名「system」を使用して、Miser システム・キューを指定します。

以下に、Miser 設定ファイルのサンプルを示します。

```
# Miser config file
QUEUE system /hosts/foobar/usr/local/data/system.conf
QUEUE user /hosts/foobar/usr/local/data/usr.conf
```

Miser コマンドライン・オプション・ファイルの設定

Miser コマンド・ライン・オプション・ファイル (`/etc/config/miser.options`) は、Miser が管理できる最大の CPU 数とメモリを定義します。

`-c` フラグは、Miser が使用できる CPU の最大数を定義します。この値は、システム・キューのリソース・セグメントが予約できる CPU の最大数です。

`-m` フラグは、Miser が使用できる最大のメモリを定義します。この値は、システム・キューのリソース・セグメントが予約できる最大メモリです。Miser 用に予約されるメモリは、物理メモリから割当てられます。Miser が使用するメモリの量は、物理メモリの総量から、カーネルが使用するのに十分なメモリを除いた量よりも少なくなければなりません。また、Miser のメモリがすべて使用中の場合に、Miser の管理下にない既存プロセスを移動するのに十分なスワップ・スペースが確保されるよう、システムには少なくとも Miser が設定する量のスワップ・スペースが必要です。

以下の例では、コマンド・ライン・オプション・ファイルで `-c` と `-m` の値をそれぞれ *1* メガバイトと *5* メガバイトに設定します。

```
-f/etc/miser.conf -v -d -c 1 -m 5m
```

`-v` フラグは、冗長モードを指定します。これにより詳細情報が出力されます。

`-d` フラグは、デバッグ・モードを指定します。このモードを指定すると、アプリケーションは tty の制御を放棄しません（つまり、デーモンにはなりません）。このモードは、Miser が正しく起動しない原因を調べる際に、`-v` フラグと組合わせて使用すると便利です。

メモ: `-C` フラグを使用すると、Miser デーモンを強制終了してから再起動されるまでの間、Miser が予約したすべてのリソースを解放できます。詳細については、`miser`(1) のマン・ページを参照してください。

設定の指針

Miser の設定は、サイトによって異なります。以下の指針を参考にしてください。

- システムでは、対話型プロセスとバッチ・プロセスの使用のバランスをとる必要があります。1 つの考え方は、Miser の制御下にないプロセッサを少なくとも 1 つか 2 つ、常に残しておくということです。これらのプロセッサは、Miser が管理する CPU がすべて予約されている場合に、システムの対話型

の部分として機能します。対話型の処理では、CPU に必要な負荷の平均は、一般に 2 よりも小さくなるようにします。最適な空き CPU 数の調整にあたっては、この点に留意してください。

- 論理スワップの空き容量は、空き CPU の数とのバランスをとる必要があります。システムに *N* 個の CPU がある場合、これらの *N* 個の CPU で実行されるプロセスに適切なメモリ量も必要です。また、多くのシステム管理者は、スワップ・スペースを使用してこのメモリをバックアップします。空き CPU を独立したシステムととらえて、それに応じたメモリとスワップ・スペースを用意すれば、対話型の処理は問題なく機能します。Miser によって予約されない空きメモリは、論理スワップ・スペース（物理メモリとスワップ・デバイスの組合わせ）であることを忘れないでください。
- 仮想スワップを使用する場合は、注意が必要です。Miser アプリケーションが実行されていない場合、タイムシェア・プロセスがすべての物理メモリを消費する可能性があります。Miser が実行されると、Miser は物理メモリの返還を要求し、タイムシェア・プロセスをスワップ・アウトします。システムが仮想スワップを使用している場合は、プロセスを移動する物理スワップがない可能性があります。すると、この時点でタイムシェア・プロセスは終了します。

Miser の設定例

この節で使用する例では、ユーザ・プログラム用に 12 個の CPU と 160 MB のメモリが使用可能であるものとします。

例 1:

この例では、システムは、1 日 24 時間、1 つのキューを使用したバッチ・スケジュール専用です。

最初の手順では、システム・キューを定義します。システム・キューに指定する長さは決定する必要があります。システム・キューの長さは、システムによって管理されるジョブの最大持続時間を定義します。このシステムの場合、1 つのジョブの最大持続時間は 48 時間であるという判断から、システム・ベクトルの持続時間を 48 時間と定義します。

```
# The system queue /usr/local/miser/system.conf
POLICY none # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 1

SEGMENT
NCPUS 12
MEMORY 160m
START 0
END 8640 # Number of quanta (48h*60 min*60 sec) / 20
```

次の手順では、ユーザ・キューを定義します。

```
# The user queue /usr/local/miser/physics.conf
POLICY default # First fit, once scheduled maintains start/end time
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 1

SEGMENT
NCPUS 12
MEMORY 160m
START 0
END 8640 # Number of quanta (48h*60 min*60 sec) / 20
```

最後の手順では、Miser 設定ファイルを定義します。

```
# Miser config file
QUEUE system /usr/local/miser/system.conf
QUEUE physics /usr/local/miser/physics.conf
```

例 2:

以下の例では、システムは 1 日 24 時間、バッチ・スケジュール専用で、2 つのユーザ・グループである chemistry と physics に分かれています。システムは、physics に 75%、chemistry に 25% の比率で分割します。

システム・キューは、例 1 で示したものと同一です。

physics ユーザ・キューは、以下のようになります。

```
# The physics queue /usr/local/miser/physics
POLICY default # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 1

SEGMENT
NCPUS 8
MEMORY 120m
START 0
END 8640 # Number of quanta (48h*60min*60sec) / 20
```

次に、chemistry キューを定義します。

```
# The chemistry queue /usr/local/miser/chemistry.conf
POLICY default # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
```

```
NSEG 1

SEGMENT
NCPUS 4
MEMORY 40m
START 0
END 8640 # Number of quanta (48h*60min*60sec) / 20
```

各キューへのアクセスを制限するため、ユーザ・グループ physics と chemistry を作成します。次に、physics キュー定義ファイルには、グループ physics にのみ実行許可を設定します。chemistry キューに対しても同様の設定を行います。

physics と chemistry のキューを定義したので、Miser 設定ファイルを定義します。

```
# Miser configuration file
QUEUE system /usr/local/miser/system.conf
QUEUE physics /usr/local/miser/physics.conf
QUEUE chem /usr/local/miser/chemistry.conf
```

例 3:

この例では、システムは、昼間はタイムシェアリング専用、夜間はバッチ専用です。夜間は午後 8:00 から午前 4:00 まで、昼間は午前 4:00 から午後 8:00 までです。

最初に、システム・キューを定義します。

```
# The system queue /hosts/foobar/usr/local/data/system.conf
POLICY none # System queue has no policy
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 2

SEGMENT
NCPUS 12
MEMORY 160m
START 0
END 720 # (4h*60min*60sec) / 20

SEGMENT
NCPUS 12
MEMORY 160m
START 3600 # (8pm is 20 hours from UTC, so 20h*60min*60sec) / 20
END 4320
```

次に、バッチ・キューを定義します。

```
# User queue
POLICY repack # Repacks jobs (FIFO) if a job finishes early
QUANTUM 20 # Default quantum set to 20 seconds
NSEG 2

SEGMENT
NCPUS 12
MEMORY 160m
START 0
END 720 # (4h*60min*60sec) / 20

SEGMENT
NCPUS 12
MEMORY 160m
START 3600 # (8pm is 20 hours from 0, so 20h*60min*60sec) / 20
END 4320
```

最後の手順では、Miser 設定ファイルを定義します。

```
# Miser config file
QUEUE system /usr/local/miser/system.conf
QUEUE user /usr/local/miser/usr.conf
```

Miser の有効化と無効化

Miser バッチ処理システムをセットアップするには、以下の手順を実行する必要があります。

1. `inst`(1M) ユーティリティを使用して、IRIX 配布メディアから `eoe.sw.miser` サブシステムをインストールします。
2. Miser の設定ファイルを自分のサイトに合わせて変更します。Miser の設定ファイルについての詳細は、38 ページの「Miser の設定例」を参照してください。

   Miser の設定ファイルを適切に変更後、`chkconfig`(1) コマンドを使用して Miser をブート時の起動対象に選択し、システムを再起動します。または、コマンド `/etc/init.d/miser start` を使用して、**root** で直接 Miser を起動することもできます。再起動せずに直接 Miser を手動で起動する場合は、先に `chkconfig` コマンドを実行しなければ、Miser は起動しません。
3. Miser を手動で有効にするには、以下のコマンド・シーケンスを使用します。

   ```
   chkconfig miser on
   /etc/init.d/miser start
   ```

4. Miser は **root** でいつでも停止できます。Miser を無効にするには、以下のコマンド・シーケンスを使用します。

```
/etc/init.d/miser stop
/etc/init.d/miser cleanup
```

実行中の Miser ジョブは停止されません。現在使用中のリソースは、ジョブが終了するまでは返還要求できません。Miser を停止した後で再起動する場合は、`miser cleanup` コマンドを実行する必要はありません。

メモ: Miser の `-c` フラグを使用すると、Miser デーモンを強制終了してから再起動されるまでの間、Miser で予約されたすべてのリソースを解放できます。

Miser ジョブの指定

Miser によって管理されるようジョブを指定するコマンドは、以下のとおりです。

```
miser_submit -q queue -o c=cpus,m=memory, t=time[,static] command
miser_submit -q queue -f file command
```

`-q` *queue*	アプリケーションをスケジュールするキューの名前を指定します。
`-o c=`*cpus*`,m=`*memory*`,` `t=`*time*`[,static]`	リソースのブロックを指定します。cpus は、スケジュールの対象となるキューで利用できる CPU の最大数以下の整数です。メモリ量 memory は、**k**(キロバイト)、**m**(メガバイト)、または **g**(ギガバイト)の単位を後に付けた整数で指定します。単位の指定がない場合は、デフォルトでバイト単位になります。時刻 time は、**h**(時)、**m**(分)、または **s**(秒)のいずれかを整数の後に付けるか、*hh*:*mm*:*ss* 形式の文字列で指定します。 default ポリシーを使用してスケジュールされた、`static` フラグの指定があるジョブは、そのセグメントの実行がスケジュールされた時刻にのみ実行されます。待ち状態のリソースが利用可能でも、先行して実行されません。repack ポリシーを使用してスケジュールされているジョブは、先行して実行できます。
`-f` *file*	リソース・セグメントのリストを指定するファイル。このフラグを使用すると、ジョブのスケジュール・パラメータを詳しく制御できます。
command	スクリプトやプログラムの名前を指定します。

詳細については、`miser_submit`(1) および `miser_submit`(4) のマン・ページを参照してください。

ジョブのスケジュールまたは説明に関する Miser での照会

指定したジョブのスケジュールまたは説明について、Miser で照会を行うコマンドは、以下のとおりです。

```
miser_jinfo -j bid [-d]
```

bid は、Miser ジョブの ID であり、ジョブのプロセス・グループ ID です。-d フラグを指定すると、ジョブの所有者とコマンドを含む、ジョブの説明が表示されます。

システムが高負荷下にあるときは、Miser のスワップ処理に時間がかかる場合があります。したがって、Miser のジョブは、指定した後、即座に処理が開始しない場合があります。

詳細については、miser_jinfo(1) のマン・ページを参照してください。

キューに関する Miser での照会

Miser キューに関する情報、キューのリソース・ステータス、およびキューにスケジュールされたジョブのリストについて、Miser で照会を行うコマンドは、以下のとおりです。

```
miser_qinfo -Q|-q queue [-j]|-a
```

-Q フラグを指定すると、現在設定されている Miser のキュー名のリストが返されます。-q フラグを指定すると、指定したキュー名に対応する空きリソースが返されます。-j フラグを指定すると、そのキューに現在スケジュールされているジョブのリストが返されます。-a フラグを指定すると、設定されたすべての Miser キューにあるすべてのスケジュールされたジョブをジョブ ID の順序で並べたリストが返されます。ジョブの簡単な説明も出力されます。

詳細については、miser_qinfo(1) のマン・ページを参照してください。

リソースのブロックの移動

1 つのキューから別のキューにリソースのブロックを移動するコマンドは、以下のとおりです。

```
miser_move -s srcq -d destq -f file
miser_move -s srcq -d destq -o s=start,e=end,c=CPUs,m=memory
```

このコマンドは、開始時刻 start から終了時刻 end まで、移動元キュー srcq のベクトルから領域の組を削除し、移動先キュー dstq のベクトルに追加します。追加または削除されるリソースは、ベクトル定義を変更しません。したがって、それらは一時的なものです。このコマンドは、各リソース移動の開始時刻と終了時刻、および移動されたリソースの量を返します。

-s と -d のフラグには、任意の有効な Miser キューを指定できます。-f フラグには、リソース・ブロックの指定が含まれます。-o フラグには、移動するリソースのブロックを指定します。開始時刻 start と終了時刻 end は、現在の時刻からの相対値で指定します。cpus には、キューに関連付けられた空き CPU の

最大数までの整数が指定できます。メモリ量 memory は、**k**（キロバイト）、**m**（メガバイト）、または **g**（ギガバイト）の識別子が付いた整数です。

メモ:リソースの移動は一時的なものです。Miser が強制終了されたりクラッシュした場合、移動されたリソースは失われ、Miser は再起動できなくなります。

詳細については、`miser_move`(1) および `miser_move`(4) のマン・ページを参照してください。

Miser のリセット

新しい設定ファイルを使用して Miser をリセットするコマンドは、以下のとおりです。

```
miser_reset -f file
```

このコマンドは、新しい設定ファイル（`-f` *file* で指定）を、実行中の Miser のバージョンに対して強制的に適用します。すべてのスケジュールされたジョブが新しい設定に対して正常にスケジュールできた場合にのみ、新しい設定は成功します。

詳細については、`miser_reset`(1) のマン・ページを参照してください。

Miser ジョブの終了

`miser_kill` コマンドを使用して、Miser に対して指定されたジョブを終了します。このコマンドは、プロセスを終了させると同時に、Miser デーモンと通信して、指定済みのプロセスで現在使用されているすべてのリソースを解放します。詳細については、`miser_kill`(1) のマン・ページを参照してください。

Miser とバッチ管理システム

この節では、Miser のジョブと、バッチ管理システムのバッチ・ジョブとの違いについて説明します。バッチ管理システムには、NQE (Network Queuing Environment) や LSF (Load Share Facility) などがあります。

Miser と、NQE などのバッチ管理システムは、互いに異なる重要な特質に欠けています。Miser には、Miser セッションを保護および管理する機能がありません。これに対して、バッチ管理システムには、リソースを保証する機能がありません。ただし、バッチ管理システムが Miser のスケジューラに対応している場合、これら 2 つのシステムを一緒に使用することで、より能力の高いソリューションが実現します。

ユーザのサイトで、バッチ管理システムによって提供されるジョブの管理や保護が必要ない場合は、Miser が単独で十分なバッチ・システムとなります。ただし、製品レベルの品質が求められるほとんどの

環境では、NQE や LSF などのバッチ・システムによって提供されるサポートや保護が必要です。このようなサイトでは、Miser のスケジューラと一緒にバッチ管理システムを実行する必要があります。

Miser のマン・ページ

man コマンドは、すべてのリソース管理コマンドに関するオンライン・ヘルプを提供します。マン・ページをオンラインで表示するには、「man *commandname*」と入力します。

一般ユーザ用マン・ページ

Miser ソフトウェアでは、以下の一般ユーザ用マン・ページが用意されています。

一般ユーザ用マン・ページ	説明
miser(1)	Miser リソース・マネージャ。miser デーモンを起動します。
miser_jinfo(1)	指定したジョブのスケジュールと説明について、Miser で照会を行います。
miser_kill(1)	Miser ジョブを強制終了します。
miser_move(1)	リソースのブロックをキュー間で移動します。
miser_qinfo(1)	Miser キューに関する情報、キューのリソース・ステータス、およびキューにスケジュールされたジョブのリストについて、Miser で照会を行います。
miser_reset(1)	新しい設定ファイルを使用して Miser をリセットします。
miser_submit(1)	ジョブを Miser キューに入れます。

ファイル形式に関するマン・ページ

Miser ソフトウェアでは、以下のファイル形式に関するマン・ページが用意されています。

ファイル形式に関するマン・ページ	説明
miser(4)	Miser の設定ファイル
miser_move(4)	Miser のリソース移動リスト
miser_submit(4)	Miser のリソース・スケジュール・リスト

その他のマン・ページ

Miser ソフトウェアでは、以下のさまざまなマン・ページが用意されています。

その他のマン・ページ	説明
`miser`(5)	Miser リソース・マネージャの概要

cpuset システム

cpuset は、CPU の名前付きの集合で、制限付きまたは開放されたものとして定義できます。制限付きの cpuset では、その cpuset のメンバーのプロセスのみが、該当する CPU 集合の上で実行可能です。開放された cpuset の場合は、任意のプロセスがその CPU 上で実行できますが、cpuset のメンバーであるプロセスはその cpuset に所属する CPU 上でのみ実行できます。cpuset は、cpuset 設定ファイルと名前によって定義されます。

cpuset システムは、基本的に、1 つまたは複数のプロセスが使用できるプロセッサ数をシステム管理者が制限できるワークロード管理ツールです。cpuset で、カーネルとユーザ・メモリの両方を制限することもできます。

メモリ制限機能が有効なとき、指定された CPU のリストから、それぞれ CPU の集合を含むノードの集合が計算され、メモリ割当てを、ノードに割当てられた CPU に制限できます。割当ての制限を使用可能な物理メモリに制限するか、オーバーフローをスワップ・ファイルにスワップできます。

システム管理者は、cpuset を使用して、大規模なシステム内で CPU を分割できます。システムを分割すると、一連のプロセスが特定の CPU に含まれ、これらのプロセスと、システム上のほかの作業の間の対話と競合が減少します。制限付き cpuset の場合、その cpuset にアタッチされたプロセスは、システム上のほかの作業の影響を受けません。cpuset にアタッチされたプロセスのみを、cpuset に割当てられた CPU で実行するようスケジュールできます。解放された cpuset を使用すると、プロセスを特定の CPU の集合でのみ実行されるよう制限し、これらのプロセスがシステムのほかの部分に及ぼす制限を最小限に抑えることができます。

システム管理者は、たとえば大規模なシステムで、通常のシステム利用をマシンの一部に制限し、システムの残りの部分を特別な目的に使用できます。`boot_cpuset`(4) ツールを使用すると、通常の起動プロセス(`init`、`inetd` など)をマシンの一部に制限し、マシンの残りの部分を特定のユーザが特別な目的のアプリケーションに使用できるようにします。カーネルは、システムの 2 つの部分の間で、プロセッサとメモリの分割を管理します。管理者は、たとえばシステムを半分ずつに分け、半分で通常のシステム利用をサポートし、残りの半分を特定のアプリケーション専用にできます。この方法を物理的な再設定と比較したときの利点は、単に再起動するだけで設定を変更でき、設定をハードウェア・モジュールの境界に合わせる必要がないことです。

XThread Control Interface (XTCI) を使用することで、カーネルのシステム・スレッドと割込みスレッドを起動 cpuset に制限できます。これにより、システム・スレッドと割込みスレッドが、リソースを求めて起動 cpuset の外部のアプリケーションと競合するのを防止します。XTCI についての詳細は、`realtime`(5) のマン・ページと、『REACT Real-Time Programmer's Guide』を参照してください。起動 cpuset についての詳細は、55 ページの「起動 cpuset」を参照してください。

`cpuset -q` *`cpuset_name`* `-p` コマンドを使用して、指定した cpuset に関連付けられているプロセスや CPU の数などの特定の cpuset のプロパティを確認することができます。cpuset のプロパティについ

ての詳細は、61 ページの「cpuset に関連付けられているプロパティの取得」と、cpuset(1) のマン・ページを参照してください。

静的 cpuset は、システムが起動された後で管理者によって定義されます。ユーザは、これらの定義された静的 cpuset にプロセスをアタッチできます。cpuset は、ジョブの実行が終了しても存在し続けます。

動的 cpuset は、ジョブに必要な場合にワークロード管理者によって作成されます。ワークロード管理者は、新規に作成された cpuset にジョブをアタッチし、ジョブの実行が終了するとその cpuset を破棄します。

cpuset は、LSF (Load Sharing Facility) や PBS (Portable Batch System) などのバッチ処理システムと共にデータ・センターのリソース管理に使用して、大規模なアプリケーションのパフォーマンスを向上できます。LSF や PBS などのアプリケーションと共に cpuset を使用すると、SGI Origin システムはさらに効率的に運用されて、ジョブ間の干渉は減り、システム・ランタイムの整合性および予測可能性を大幅に向上させることができます。

cpuset ライブラリ・ルーチンである cpusetMove (3x) および cpusetMoveMigrate(3x) は、cpuset 間でプロセスを移動したり、オプションとしてメモリを移行するために使用できます。これらのルーチンを使用すると、特定のプロセスやプロセス・グループを既存の cpuset 間で移動したり、ある名前付きの cpuset から、特定の名前付きの cpuset に割当てられていない CPU のプールに移動することができます。未使用 CPU のこのプールは、グローバル cpuset と呼ばれます。

この機能を使用すると、既存の cpuset を破棄してリソースを解放し、重要ジョブを実行した後、cpuset を再構成して前のジョブを続けることが簡単になります。プロセスによって使用されるメモリは、新しい cpuset に関連付けられているノードに移行できるため、メモリの局所性は改善されます。cpusetMove(3x) ルーチンと cpusetMoveMigrate(3x) ルーチンについての詳細は、64 ページの「cpusetMove と cpusetMoveMigrate 関数の使用」および177ページの「cpuset システムの API」を参照してください。

システムの分割についての詳細は、『IRIX Admin: System Configuration and Operation』の第 4 章「Configuring the IRIX Operating System」を参照してください。

cpuset ライブラリ内のインタフェースを使用して、プログラマは cpuset を作成、破棄したり、既存の cpuset に関する情報を取得したり、特定の cpuset に関連付けられているプロパティを取得したり、プロセスとその子プロセスを cpuset にアタッチしたりできます。

この章は、以下の節で構成されています。

- 49 ページの「cpuset の使用」
- 51 ページの「cpuset 内の CPU に対する制限」
- 51 ページの「cpuset システムのチュートリアル」
- 55 ページの「起動 cpuset」
- 57 ページの「cpuset コマンドと設定ファイル」

cpuset の使用

この節では、cpuset および cpuset(1) コマンドを使用するための基本的な手順を示します。詳細については、51 ページの「cpuset システムのチュートリアル」を参照してください。

cpuset システム・ソフトウェアのインストール方法については、61 ページの「cpuset システムのインストール」を参照してください。

cpuset を使用するには、以下の手順を実行します。

1. cpuset 設定ファイルを作成し、名前を付けます。このファイルの形式については、58 ページの「cpuset 設定ファイル」を参照してください。cpuset に含まれる CPU に適用される制限については、51 ページの「cpuset 内の CPU に対する制限」を参照してください。
2. -f パラメータで設定ファイルを、-q パラメータで名前を指定して、cpuset を作成します。

cpuset (1) コマンドを使用して、cpuset を作成、破棄したり、既存の cpuset に関する情報を取得したり、プロセスとその子プロセスを cpuset にアタッチしたりします。cpuset コマンドの構文は、次のとおりです。

```
cpuset [-q cpuset_name[,cpuset_name_dest] [-A command]|[-c -f filename]|
[-d]|[-i]|[-l][-m]|[-M]|[-Q]|[-p]]|[-T]| -C | -Q | -h
```

cpuset コマンドでは、以下のオプションを使用できます。

-q *cpuset_name* [-A *command*]
: -q パラメータで指定された cpuset で、指定したコマンドを実行します。ユーザにアクセス・パーミッションがなかったり、cpuset が存在しない場合は、エラーが返されます。

`-q` *cpuset_name* [`-c -f` *filename*]	`-f` パラメータで設定ファイルを、 `-q` パラメータで名前を指定して、cpuset を作成します。cpuset 名がすでに存在していたり、cpuset 設定ファイルで指定されている CPU がすでに cpuset のメンバーであったり、ユーザに適切なパーミッションがない場合、操作は失敗します。
`-q` *cpuset_name* `-d`	指定した cpuset を破棄します。cpuset は、現在アタッチされているプロセスがない場合にのみ破棄できます。
`-q` *cpuset_name* `-l`	cpuset 内のプロセスをすべてリストします。
`-q` *cpuset_name* `-m`	アタッチされているプロセスをすべて cpuset 外に移動します。
`-q` *cpuset_name* `-d`	指定した cpuset を破棄します。cpuset は、現在アタッチされているプロセスがない場合にのみ破棄できます。
`-q` *cpuset_name* `-Q`	cpuset に含まれる CPU のリストを出力します。
`-q` *cpuset_name* `-p`	パーミッション、ACL、MAC ラベル、フラグ、プロセスの数、および指定した cpuset に関連付けられている CPU 番号を出力します。
`-q` *cpuset_name,cpuset_name_dest* `-M suboption`	`-M` オプションは、プロセスまたはプロセス・グループと関連メモリを、*cpuset_name* から *cpuset_name_dest* に移動します。有効なサブオプションは、`PID`、`ASH`、`JID`、`SID`、および `PGID` で、移動される ID タイプを示します。また、`-M` オプションには `-i` オプションが必要です。
`-q` *cpuset_name,cpuset_name_dest* `-T suboption`	`-T` オプションは、プロセスまたはプロセス・グループを `cpuset_name` から `cpuset_name_dest` に移動しますが、関連メモリは移動しません。有効なサブオプションは、`PID`、`ASH`、`JID`、`SID`、および `PGID` で、移動される ID タイプを示します。また、`-T` オプションには `-i` オプションが必要です。
`-q` *cpuset_name,cpuset_name_dest* [`-M` | `-T`] `suboption -i` *id*	`-i` オプションは、移動する必要のある ID をコマンドに通知します。

`-C`	プロセスが現在アタッチされている cpuset の名前を出力します。
`-Q`	現在、定義されているすべての cpuset の名前をリストします。
`-h`	コマンドの使用法に関するメッセージを出力します。

3. 次の `cpuset` コマンドを実行して、作成した cpuset でコマンドを実行します。

   ```
   cpuset -q cpuset_name -A command
   ```

cpuset の使用についての詳細は、`cpuset`(1) のマン・ページ、51 ページの「cpuset 内の CPU に対する制限」、および 51 ページの「cpuset システムのチュートリアル」を参照してください。

cpuset 内の CPU に対する制限

cpuset に含まれる CPU には、以下の制限が適用されます。

- CPU は、1 つの cpuset にのみ含めることができます。
- CPU 0 は、`EXCLUSIVE` cpuset に含めることはできません。
- CPU は制限付きかつ隔離されて（`mpadmin`(1) および `sysmp`(2) を参照）、かつ同時に cpuset のメンバーになることはできません。
- スーパー・ユーザのみが cpuset を作成または破棄できます。
- `runon`(1) コマンドは、ユーザに cpuset の設定ファイルへの書込みパーミッションまたはグループ書込みパーミッションがないかぎり、cpuset に含まれる CPU でコマンドを実行できません。

`cpuset` コマンドの引数の説明および詳細については、`cpuset`(1)、`cpuset`(4)、および `cpuset`(5) のマン・ページを参照してください。

cpuset システムのチュートリアル

この節では、cpuset を使用してシステムを分割する方法の例を示します。また、サンプル・システムを複数の cpuset に分割する簡単な手順と、詳細な説明の参照先を示します。

52 ページの図4-1 に、16 個のプロセッサと 3 個の cpuset があるシステムのブロック図を示します。この節では、設定ファイルの例と、通常のシステム利用のためにシステムの CPU の半分を含む起動 (Boot) cpuset を作成し、さらに特定の目的のために `Green` と `Blue` という 2 つの cpuset を作成するためのコマンドを示します。`Green` cpuset は、グループ `artists` のメンバーが実行する特定のアプリケーショ

ンに制限される cpuset を指定します。Blue cpuset は、グループ writers のメンバーが実行する特定のアプリケーションに制限される cpuset を指定します。

図4-1 cpuset を使用したシステム分割

プロセッサが 16 個のシステムを、52 ページの図4-1 に示すように 3 個の cpuset に分割するには、以下の手順を実行します。

1. boot_cpuset.config というファイルを作成して起動 cpuset を作成し、CPU が 16 個のシステムの半分を通常のシステム利用専用にします。起動 cpuset には、デーモン、対話型またはバックグ

ラウンドの処理、スクリプトなど、システムの標準のプロセスがすべて含まれます。このファイルの内容は、以下のとおりです。

```
# boot
MEMORY_LOCAL
MEMORY_MANDATORY

CPU 0
CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
CPU 6
CPU 7
```

メモ: このリリースでは、boot_cpuset.config ファイル内の 1 行で指定できる CPU の数は 1 個だけです。boot_cpuset.config ファイルについての詳細は、55 ページの「起動 cpuset」を参照してください。

MEMORY_LOCAL および MEMORY_MANDATORY の各フラグについては、58 ページの「cpuset 設定ファイル」を参照してください。

2. chkconfig(1M) コマンドを -f オプションと共に使用して、以下を含む /etc/config/boot_cpuset ファイルを作成します。

   ```
   chkconfig boot_cpuset on
   ```

   /etc/config/boot_cpuset ファイルについての詳細は、55 ページの「起動 cpuset」を参照してください。

   システムが再起動されると、起動 cpuset が作成されます。

3. Green という専用 cpuset を作成し、そこで実行する特定のアプリケーション（ここでは MovieMaker）を割当てます。これを行うには、以下の手順を実行します。

   a. 以下の内容で、cpuset_1 という cpuset 設定ファイルを作成します。

      ```
      # the cpuset configuration file called cpuset_1 that shows
      # a cpuset dedicated to a specific application
      EXCLUSIVE
      MEMORY_LOCAL
      MEMORY_MANDATORY
      ```

```
CPU 8
CPU 9
CPU 10
CPU 11
```

メモ: cpuset 設定ファイル内の 1 行では、複数の CPU や CPU の範囲を指定できます。この例では、CPU 8-11 のように、1 行で 8～11 の CPU を指定できます。cpuset 設定ファイルについての詳細は、58 ページの「cpuset 設定ファイル」を参照してください。

EXCLUSIVE、MEMORY_LOCAL、および MEMORY_MANDATORY の各フラグについては、58 ページの「cpuset 設定ファイル」を参照してください。

b. chmod(1) コマンドを使用して cpuset_1 設定ファイルのパーミッションを設定し、グループ artists のメンバーのみが Green cpuset で moviemaker アプリケーションを実行できるようにします。

c. cpuset(1) コマンドを使用して、Green cpuset を作成します。設定ファイル cpuset_1 は -f パラメータで指定し、名前 Green は -q パラメータで指定します。

```
cpuset -q Green -f cpuset_1
```

d. 次の cpuset コマンドを実行して、専用 cpuset で MovieMaker を実行します。

```
cpuset -q Green -A moviemaker
```

cpuset(1) コマンドについての詳細は、57 ページの「cpuset コマンド」を参照してください。

moviemaker ジョブのスレッドは、この cpuset 内の CPU でのみ実行されます。MovieMaker ジョブでは、cpuset 内の CPU が含まれるシステム・ノードのメモリが使用されます。ほかの cpuset で実行されているジョブでは、これらのノードのメモリは使用されません。この例で示す構文で cpuset コマンドを使用して、ほかのアプリケーションも同様にこの cpuset で実行できます。

4. Blue という 3 番目の cpuset ファイルを作成し、この cpuset でのみ実行するアプリケーションを指定します。これを行うには、以下の手順を実行します。

a. 以下の内容で、cpuset_2 という cpuset 設定ファイルを作成します。

```
# the cpuset configuration file called cpuset_2 that shows
# a cpuset dedicated to a specific application
EXCLUSIVE
MEMORY_LOCAL
MEMORY_MANDATORY

CPU 12
CPU 13
```

```
CPU 14
CPU 15
```

b. chmod(1) コマンドを使用して cpuset_2 設定ファイルのパーミッションを設定し、グループ writers のメンバーのみが Blue cpuset で bookmaker アプリケーションを実行できるようにします。

c. cpuset(1) コマンドを使用して、Blue cpuset を作成します。設定ファイル cpuset_2 は -f パラメータで指定し、名前は -q パラメータで指定します。

```
cpuset -q Blue -f cpuset_2
```

d. 次の cpuset(1) コマンドを実行して、Green cpuset 内の CPU で bookmaker を実行します。

```
cpuset -q Blue -A bookmaker
```

bookmaker ジョブのスレッドは、この cpuset でのみ実行されます。BookMaker ジョブでは、cpuset 内の CPU が含まれるシステム・ノードのメモリが使用されます。ほかの cpuset で実行されているジョブでは、これらのノードのメモリは使用されません。

起動 cpuset

boot_cpuset.so(4) ライブラリを使用して、init(1M) プロセスとそのすべての子孫を cpuset 内に含めることができます。標準プロセスはすべて init プロセスの子孫なので、デーモン、対話型またはバックグラウンドの処理、スクリプトなど、システム上のすべての標準プロセスがこの cpuset に制限されます。この cpuset を **boot** と呼びます。

XThread Control Interface (XTCI) を使用することで、カーネルのシステム・スレッドと割込みスレッドを起動 cpuset に制限できます。これについては、realtime(5) のマン・ページと、『REACT Real-Time Programmer's Guide』で説明されています。

メモ:

boot_cpuset.so ライブラリは、SGI 2000、SGI Origin 300、および SGI Orgin 3000 シリーズの、ccNUMA または NUMAflex アーキテクチャに基づくシステムでのみ使用できます。

SGI Origin 3000 シリーズのサーバでは、NUMAflex 相互接続構成およびモジュラー・コンポーネント（または「ブリック」と呼ばれる）を使用して、CPU とメモリ、I/O、およびストレージを、それぞれ別のブリックに分離します。C-brick と呼ばれる CPU ブリックには、4 個の CPU と、最大 8 G バイトのローカル・メモリが含まれます。SGI 2000 シリーズのサーバでは、以前の ccNUMA 相互接続構成が使用されます。スケーラブルな ccNUMA アーキテクチャの最小構築ブロックはノード・ボードで、キャッシュとメモリに関連付けられた 2 個の CPU から構成されます。このマニュアルに記載されている cpuset の説明は、NUMAflex アーキテクチャと ccNUMA アーキテクチャの両方に適用されます。

---

boot_cpuset.so ライブラリは /lib32 ディレクトリにあり、その動作は以下のファイルで制御されます。

- /etc/config/boot_cpuset
- /etc/config/boot_cpuset.config

/etc/config/boot_cpuset ファイルを作成するには、次のように chkconfig(1M) コマンドを使用します。

```
chkconfig -f boot_cpuset on
```

chkconfig(1M) コマンドを使用して、boot_cpuset.so(4) ライブラリを有効または無効に設定できます。ライブラリが有効に設定されている場合、システム起動時に init によって boot_cpuset.so ライブラリが読込まれて実行され、cpuset が作成されます。ライブラリが無効に設定されている場合、ライブラリは終了し、init で通常の処理が再開されます。

/etc/config/boot_cpuset.config ファイルは、cpuset を指定する設定ファイルです。このファイルは、cpuset(4) 設定ファイルと同じ規則に従います。

以下に、通常のシステム利用のために CPU が 8 個のシステムを半分に分割する boot_cpuset.config ファイルの例を示します。

```
# the boot_cpuset
MEMORY_LOCAL
MEMORY_MANDATORY

CPU 0
CPU 1
CPU 2
CPU 3
```

メモ: CPU 0 は、EXCLUSIVE cpuset に含めることはできません。cpuset に含まれる CPU に適用される制限については、51 ページの「cpuset 内の CPU に対する制限」を参照してください。

2 番目の設定ファイルは、特定のアプリケーション専用にできる cpuset を示します。

```
# the cpuset dedicated to a specific application
EXCLUSIVE
MEMORY_LOCAL
MEMORY_MANDATORY

CPU 4
CPU 5
CPU 6
CPU 7
```

詳細については、57 ページの「cpuset コマンドと設定ファイル」と、cpuset(4) のマン・ページを参照してください。

cpuset コマンドと設定ファイル

この節では、cpuset(1) コマンドと cpuset 設定ファイルについて説明します。

cpuset コマンド

cpuset (1) コマンドは、cpuset と呼ばれる、CPU の集合の定義および管理に使用します。cpuset は、CPU の名前付きの集合で、制限付きまたは開放されたものとして定義できます。cpuset コマンドを使用して、cpuset を作成、破棄したり、既存の cpuset に関する情報を取得したり、プロセスを cpuset にアタッチしたりできます。cpuset へのアタッチは、fork(2) システム・コールに渡されて継承されます。したがって、アタッチされたプロセスのすべての子プロセスも、同じ cpuset にアタッチされます。

メモ: cpuset コマンドでは、Miser バッチ処理システムを使用する必要はありません。

制限付きの cpuset では、その cpuset にアタッチされたプロセスのみが、該当する CPU 集合の上で実行可能です。開放された cpuset の場合は、任意のプロセスがその CPU 上で実行できますが、cpuset にアタッチされたプロセスはその cpuset に所属する CPU 上でのみ実行できます。

SGI 2000、SGI Origin 300、および SGI Origin 3000 シリーズのシステム、つまり ccNUMA アーキテクチャに基づくシステムでは、管理者はメモリ割当てを cpuset で定義された CPU を含むノード内に制限

できます。詳細については、この後の MEMORY_MANDATORY フラグの説明、および cpuset(4) のマン・ページを参照してください。

cpuset 設定ファイル

cpuset は、cpuset 設定ファイルと名前によって定義されます。ファイル形式の定義については、cpuset(4) のマン・ページを参照してください。cpuset 設定ファイルは、cpuset のメンバーとなる CPU をリストするために使用します。このファイルには、cpuset を定義するのに必要な追加の引数も含まれます。cpuset 名の長さは、3 文字以上、8 文字以下です。2 文字以下の名前は、予約されています。cpuset 設定ファイル内の 1 行では、cpuset の一部として 1 つまたは複数の CPU を指定したり、CPU の範囲を指定することができます。cpuset 内では、特定の順序で CPU を指定する必要はありません。システム上の cpuset ごとに別個の cpuset 設定ファイルが必要です。

メモ: クラスタ環境では、cpuset 設定ファイルは root ファイルシステムに存在する必要があります。cpuset 設定ファイルが root ファイルシステム以外のファイルシステムに存在している場合にファイルシステムをアンマウントしようとすると、cpuset の v ノードはアクティブの状態のままになり、unmount コマンドは失敗します。詳細については、mount(1M) のマン・ページを参照してください。

設定ファイルのパーミッションにより、cpuset へのアクセスが定義されます。パーミッションをチェックする必要がある場合、ファイルの現在のパーミッションが使用されます。したがって、特定の cpuset へのアクセスは、cpuset を破棄して再作成しなくても、単にアクセス・パーミッションを変更するだけで変更できます。ユーザは、読込みパーミッションがあれば cpuset に関する情報を取得し、実行パーミッションがあれば cpuset にプロセスをアタッチできます。

規則上、SGI システムの CPU の番号は「0〜システム上のプロセッサ数 - 1」の範囲となります。mpadmin -n コマンドを実行すると、システム上で物理的に構成されたプロセッサが報告されます。また、hinv -vm コマンドを使用して、システムのハードウェア設定を表示できます。CPU の命名規則とシステムのハードウェア設定についての詳細は、『IRIX Admin: System Configuration and Operation』の第 4 章「Configuring the IRIX Operating System」と、mpadmin(1) および hinv(1) のマン・ページを参照してください。

以下の例は、3 個の CPU を含む排他的な cpuset を記述するサンプル設定ファイルです。

```
# cpuset configuration file
EXCLUSIVE
MEMORY_LOCAL
MEMORY_EXCLUSIVE

CPU 1
CPU 5
CPU 10
```

この指定により、3 個の CPU を含む cpuset が作成されます。EXCLUSIVE フラグを設定すると、上記の CPU は、この cpuset に明示的に割当てられたスレッドの実行に制限されます。MEMORY_LOCAL フラグを設定すると、この cpuset 上で実行されるジョブは、cpuset 内の CPU を含むノードのメモリを使用します。MEMORY_EXCLUSIVE フラグを設定すると、ほかの cpuset やグローバル cpuset で実行されるジョブは、通常はこれらのノードのメモリを使用しません。

MEMORY_MANDATORY フラグを設定すると、この cpuset 上で実行されるジョブは、cpuset 内の CPU を含むノードのメモリのみを使用できます。MEMORY_LOCAL フラグは強制ではありませんが、MEMORY_MANDATORY フラグはカーネルによって強制されます。

メモ: Miser と cpuset の両方があるシステムでは、Miser キューが使用する CPU と、cpuset に割当てられた CPU との間で競合が発生する場合があります。Miser は、EXCLUSIVE フラグが設定された cpuset に所属する CPU にアクセスできません。Miser と cpuset を同じシステムで実行することは避けてください。

以下の例は、7 個の CPU を含む排他的な cpuset を記述するサンプル設定ファイルです。

```
# cpuset configuration file
EXCLUSIVE
MEMORY_LOCAL
MEMORY_EXCLUSIVE

CPU 16
CPU 17-19, 21
CPU 27
CPU 25
```

コマンドは改行で終わります。コメント区切り # に続く文字は無視されます。大文字と小文字は区別されます。トークンは空白（無視される）で区切ります。

有効なトークンは、以下のとおりです。

有効なトークン	説明
EXCLUSIVE	cpuset 内の CPU を制限付きとして定義します。ファイル内の任意の場所に記述できます。同じ行にあるそれ以外の部分は無視されます。
MEMORY_LOCAL	cpuset に割当てられたスレッドは、その cpuset 内のノードからのみメモリを割当てようとします。cpuset の外部からのメモリ割当ては、cpuset 内で空きメモリが利用できない場合にのみ発生します。cpuset の外部で実行しているスレッドへのメモリ割当てには制限は加えられません。

MEMORY_EXCLUSIVE

cpuset の外部でメモリが利用できない場合を除き、cpuset に割当てられないスレッドは、その cpuset 内のメモリを使用しません。

cpuset が作成され、その cpuset のノードですでに実行中のスレッドによってメモリが占有されると、明示的にこのメモリを移動するための試みは行われません。ページ移動が可能な場合、ページへの参照の大半が非ローカルであることがシステムで検出されると、そのページは移動されます。

MEMORY_KERNEL_AVOID

カーネルは、この cpuset に含まれるノードからのメモリ割当てを避けます。カーネルのメモリ要求がこの cpuset の外部では満たされない場合、このオプションは無視され、この cpuset 内部でのメモリ割当てが発生します。現在このオプションでは、指定したノードにシステム・バッファ・キャッシュが配置されることのみが防止されます。

注意: cpuset が実行されている大部分のサイトでは、このオプションを使用しないでください。このようなサイトでオプションを使用すると、システム・パフォーマンスが低下する可能性があります。これは、カーネル・メモリ割当てが残りのシステム・ノードに集中するためです。このオプションは特定のワークロード・パターンに対してのみ有効で、それ以外の状況ではパフォーマンス・ペナルティが深刻になる場合があります。このオプションは、SGI サポート・スタッフから指示があった場合にのみ使用してください。

このオプションは、IRIX 6.5.7 リリースで追加されました。

MEMORY_MANDATORY

カーネルは、すべてのメモリ割当てをこの cpuset に含まれるノードに限定します。メモリ要求が満たされない場合、メモリが使用可能になるまで割当てプロセスはスリープします。これ以上のメモリを割当てられない場合、プロセスは強制終了されます。

POLICY_PAGE

MEMORY_MANDATORY トークンが必要です。ポリシーが指定されていない場合のデフォルトのポリシーです。このポリシーでは、カーネルが、ユーザ・ページをスワップ・ファイル（swap(1M) を参照）に移動し、この cpuset に含まれるノード上の物理メモリを解放します。スワップ・スペースの空きがなくなった場合、プロセスは強制終了されます。

POLICY_KILL	MEMORY_MANDATORY トークンが必要です。カーネルは、カーネルのヒープからなるべく多くの領域を解放しようとしますが、ユーザ・ページをスワップ・ファイルに移動しません。cpuset に含まれるノード上のすべての物理メモリに空きがなくなった場合、プロセスは強制終了されます。
CPU	この cpuset の一部となる CPU を指定します。

cpuset システムのインストール

cpuset システムは機能的に Miser バッチ処理システムと分離していますが、現在の cpuset システムは、Miser のソフトウェア開発と関連して開発されています。cpuset システム・ソフトウェアは、Miser サブシステム・ソフトウェアに含まれます。cpuset システム・ソフトウェアのインストール方法については、41 ページの「Miser の有効化と無効化」を参照してください。

cpuset に関連付けられているプロパティの取得

`cpuset -q` *`cpuset_name`* `-p` コマンドを使用して、特定の cpuset に関連付けられているさまざまなプロパティ（以下を参照）を確認することができます。

- cpuset へのアクセス（設定ファイルのパーミッションで定義）
- アクセス・コントロール・リスト (ACL: Access Control List)
- 強制アクセス・コントロール (MAC: Mandatory Access Control) ラベル
- フラグ（MEMORY_EXCLUSIVE など）

  cpuset に関連付けられているフラグについての詳細は、58 ページの「cpuset 設定ファイル」と、`cpuset`(4) のマン・ページを参照してください。
- プロセスの数
- CPU 番号

cpuset ライブラリ内の `cpusetGetProperties`(3x) 関数を使用して、指定した cpuset のさまざまなプロパティを取得できます。`cpusetFreeProperties`(3x) 関数を使用して、`cpuset_Properties_t` 構造体に使用されているメモリを解放できます。詳細については、198ページの「取得関数」および215ページの「クリーンアップ関数」と、`cpusetGetProperties` (3x) および `cpusetFreeProperties`(3x) のマン・ページを参照してください。

cpuset システムと Trusted IRIX

この節では、Trusted IRIX 環境での cpuset の実行方法について説明します。

設定ファイルのパーミッションにより、cpuset へのアクセスが定義されます。パーミッションをチェックする必要がある場合、ファイルの現在のパーミッションが使用されます。

ユーザは、読込みパーミッションがあれば cpuset に関する情報を取得し、実行パーミッションがあれば cpuset にプロセスをアタッチできます。

IRIX 上の cpuset には、root と user という 2 つのユーザ・クラスが必要です。root クラスは、プロセスを作成、破棄、移動したり、cpuset にプロセスを追加します。user クラスは、特定の cpuset に対する設定ファイルのパーミッションによって可能な操作が異なります。

設定ファイルに以下のパーミッションがある場合を考えます。

```
パーミッション      所有者    グループ   サイズ   ファイル名
-----------------------------------------------------------

-rwxr-x---      root    cpuset   512    cpuset.test
```

グループ読込みパーミッションがあるので、グループ cpuset に属するユーザは、`cpuset.test` ファイルによって定義された cpuset 内の cpuset をすべてリストし、この cpuset 内のすべてのプロセスのリストを取得できます。 `cpuset.test` ファイルによって管理される cpuset にユーザがプロセスを追加できるようにするには、パーミッションを次のように変更する必要があります。

```
パーミッション      所有者    グループ   サイズ   ファイル名
-----------------------------------------------------------

-rwxr-x---      root    cpuset   512    cpuset.test
```

Trusted IRIX 環境では、パーミッションは `/etc/capability` ファイルによって管理されます。プロセスの操作権に対する細かな調整を可能にする capability についての詳細は、`capability`(4) および `capabilities`(4) のマン・ページを参照してください。`capability` ファイル内の各ユーザには、最小パーミッションと最大パーミッションのセットがあります。そのため、root には特殊な権限はありません（ただし、`suattr`(1M) 呼出しを使用すると、すべての capability とパーミッションが容認されます）。また、capability とパーミッションは、強制アクセス・コントロール (MAC: Mandatory Access Control) ラベルとアクセス・コントロール・リスト (ACL: Access Control List) を使用することによって制限されます。

Trusted IRIX において、グループ `cpuset` に属するユーザが、 `cpuset.test` ファイルによって定義された cpuset 内の cpuset をすべてリストし、この cpuset 内のすべてのプロセスのリストを取得できるようにするには、以下を実行する必要があります。

- userlow という MAC ラベルをユーザに割当てます。
- `/etc/capability` ファイルに `cpuuser1:all=:all=` というエントリを入力します。

ユーザに対して、すべての capability に関する、有効で、継承された、許可を与える権限 (+eip) を与えないで下さい。+eip を与えた場合、ユーザの操作権は、cpuset システムによって許可される範囲を上回ってしまいます。

/etc/capability ファイル内に cpuuser1:all=:all= というエントリが入力されている Trusted IRIX ユーザは、IRIX の user クラスと同じパーミッションを持ちます。

Trusted IRIX の root クラスには、cpuset を作成、破棄したり、cpuset からプロセスを移動したりするために CAP_SCHED_MGT+eip capability が必要です。

Trusted IRIX では、ACL を使用してグループ・パーミッションを制御できます。ACL を使用すると、cpuset にプロセスを追加できるユーザをグループ内から簡単に選択できます。cpuset への特定ユーザのアクセスは、このユーザを設定ファイルのグループ・オーナーに含めなくても、ACL を使用して制御できます。

cpuset ライブラリの使用

cpuset ライブラリ内のインタフェースを使用して、プログラマは cpuset を作成、破棄したり、既存の cpuset に関する情報を取得したり、既存の cpuset に関連付けられているプロパティを取得したり、プロセスとその子プロセスを cpuset にアタッチしたりできます。

cpuset ライブラリの使用についての詳細は、177ページの「cpuset システムの API」を参照してください。

この節では、以下のトピックについて説明します。

- 63 ページの「cpusetAttachPID と cpusetDetachPID 関数の使用」
- 64 ページの「cpusetMove と cpusetMoveMigrate 関数の使用」

cpusetAttachPID と cpusetDetachPID 関数の使用

cpuset ライブラリ内の cpusetAttachPID(3x) 関数を使用して、PID 値によって識別される特定のプロセスを cpuset にアタッチできます。また、新しい cpusetDetachPID 関数を使用して、PID 値によって識別される特定のプロセスを cpuset からデタッチできます。特定のプロセスを cpuset にアタッチしたり、cpuset からデタッチできる機能は、cpuset 設定ファイルのパーミッションと、関連するプロセスの所有権によって制御されます。cpuset 設定ファイルについての詳細は、58 ページの「cpuset 設定ファイル」を参照してください。

cpusetAttachPID(3x) および cpusetDetachPID(3x) 関数は、メモリ・レーテンシの問題を避けるために設定される MEMORY_MANDATORY フラグと一緒には使用しないでください。cpuset では元の処理ノードからのみメモリを使用するため、cpusetAttachPID および cpusetDetachPID 関数は以下のように使用します。

64 ページの図4-2 に、CPU が 4 個ずつ含まれる、合計 2 つの cpuset で実行されているいくつかのジョブを示します。ある重要ジョブが、8 個の CPU をすべて使用する新しい cpuset を必要としているとします。新しい cpuset を作成するには、以下の手順を実行します。

1. `cpusetDetachPID` 関数を使用して、cpuset A と cpuset B からすべてのジョブを移動します。
2. cpuset A と B で実行されているジョブを中断します。
3. `cpusetDestroy` (3x) 関数を使用して、cpuset A と cpuset B を破棄します。
4. `cpusetCreate` (3x) 関数を使用して、重要ジョブに新しい cpuset を作成します。
5. 新しい cpuset 内の重要ジョブを実行します。
6. 重要ジョブの実行が完了したら、新しい cpuset を破棄します。
7. cpuset A と B を、以前とまったく同じように再作成します。
8. 中断したジョブを再開します。
9. `cpusetAttachPID` 関数を使用して、各ジョブを対応する cpuset に再アタッチします。

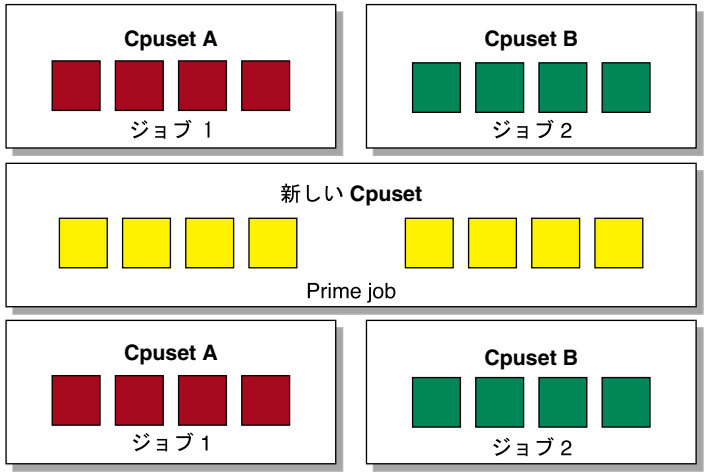

図4-2 `cpusetAttachPID` と `cpusetDetachPID` 関数の使用

`cpusetMove` と `cpusetMoveMigrate` 関数の使用

65 ページの図4-3 に、`cpusetMove`(3x) および `cpusetMoveMigrate` (3x) 関数の使用の例を示します。

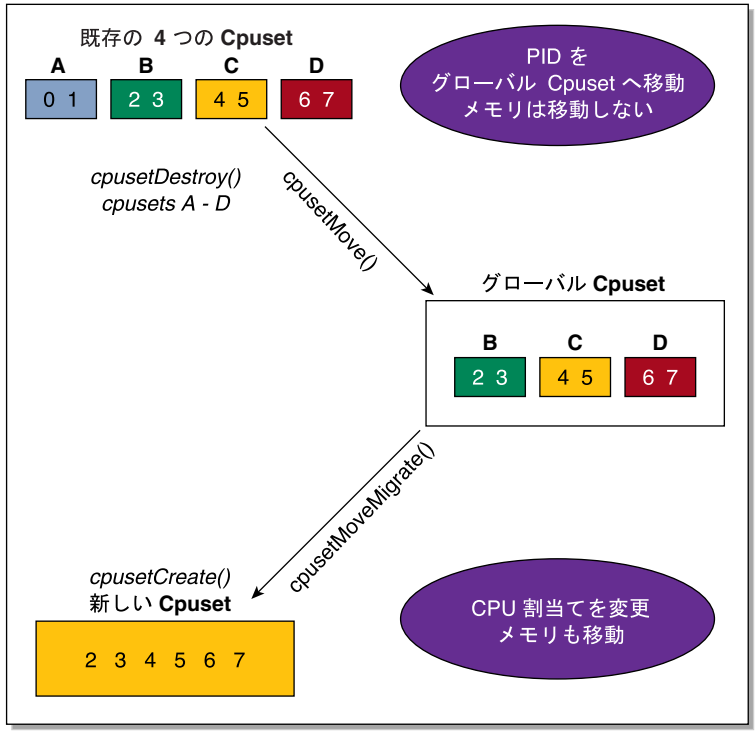

図4-3 特定の cpuset から別の cpuset へのプロセスの移動

`cpusetMoveMigrate` 関数を使用して、特定のプロセスとそれに関連付けられているメモリ(プロセス ID (PID)、プロセス・グループ ID (PGID)、ジョブ ID (JID)、セッション ID (SID)、またはアレイ・サービス・ハンドル (ASH: Array Services Handle) によって識別される)を指定した cpuset に直接移動できます。`cpusetMove` 関数は、プロセス(PID、PGID、JID、または ASH によって識別される)を、指定した cpuset から別の cpuset またはグローバル cpuset に一時的に移動します。この場合、メモリは移行(移動)されません。グローバル cpuset とは、cpuset 内には含まれないすべての CPU を説明するために使用される用語です。`cpusetMoveMigrate` 関数とは異なり、`cpusetMove` 関数は、プロセスに関連付けられているメモリを移動しません。65 ページの図4-3 に、これらの関数の使用例を示します。破棄する予定の cpuset からグローバル cpuset にプロセスを移動するには、`cpusetMove` 関数を使用して、移動先を NULL と指定します。その後、`cpusetMoveMigrate` 関数を使用して、グローバル cpuset から、新規に作成された cpuset にプロセスを移動します。

cpuset システムのマン・ページ

man コマンドは、すべてのリソース管理コマンドに関するオンライン・ヘルプを提供します。マン・ページをオンラインで表示するには、「man *commandname*」と入力します。cpuset ライブラリのマン・ページの印刷版については、付録 A の177ページの「cpuset システムの API」を参照してください。

一般ユーザ用マン・ページ

cpuset システム・ソフトウェアでは、以下の一般ユーザ用マン・ページが用意されています。

一般ユーザ用マン・ページ	説明
cpuset(1)	CPU の集合を定義および管理します。

cpuset ライブラリのマン・ページ

cpuset システム・ソフトウェアでは、以下の cpuset ライブラリのマン・ページが用意されています。

cpuset ライブラリのマン・ページ	説明
cpusetAllocQueueDef(3x)	cpuset_QueueDef_t 構造体を割当てます。。
cpusetAttach(3x)	現在のプロセスを cpuset にアタッチします。
cpusetAttachPID(3x)	特定のプロセスを cpuset にアタッチします。
cpusetCreate(3x)	cpuset を作成します。
cpusetDestroy(3x)	cpuset を破棄します。
cpusetDetachAll(3x)	cpusetからすべてのスレッドをデタッチします。
cpusetDetachPID(3x)	特定のプロセスをcpusetからデタッチします。
cpusetFreeCPUList(3x)	cpuset_CPUList_t 構造体で使用されているメモリを解放します。
cpusetFreeNameList(3x)	cpuset_NameList_t 構造体で使用されているメモリを解放します。
cpusetFreePIDList(3x)	cpuset_PIDList_t 構造体で使用されているメモリを解放します。
cpusetFreeProperties(3x)	cpuset_Properties_t 構造体で使用されているメモリを解放します。
cpusetFreeQueueDef(3x)	cpuset_QueueDef_t 構造体で使用されているメモリを解放します。

`cpusetGetCPUCount`(3x)	システム上で構成されている CPU の数を取得します。
`cpusetGetCPUList`(3x)	cpuset に割当てられたすべての CPU のリストを取得します。
`cpusetGetName`(3x)	プロセスがアタッチされている cpuset の名前を取得します。
`cpusetGetNameList`(3x)	定義されたすべての cpuset の名前のリストを取得します。
`cpusetGetPIDList`(3x)	cpuset に割当てられたすべての PID のリストを取得します。
`cpusetGetProperties`(3x)	cpuset に関連付けられているさまざまなプロパティを取得します。
`cpusetMove`(3x)	プロセス(PID、PGID、JID、SID、または ASH によって識別される)を、指定した cpuset から一時的に移動します。
`cpusetMoveMigrate`(3x)	特定のプロセス(PID、PGID、JID、SID、または ASH によって識別される)とそれに関連付けられているメモリを、特定の cpuset から別の cpuset に移動します。

ファイル形式に関するマン・ページ

cpuset システム・ソフトウェアでは、以下のファイル形式に関するマン・ページが用意されています。

ファイル形式に関するマン・ページ	説明
`cpuset`(4)	cpuset 設定ファイル

その他のマン・ページ

cpuset システム・ソフトウェアでは、以下のさまざまなマン・ページが用意されています。

その他のマン・ページ	説明
`cpuset`(5)	cpuset システムの概要を示します。

完全システム・アカウンティング

IRIX システムには、基本アカウンティング、拡張アカウンティング、および完全システム・アカウンティング (CSA: Comprehensive System Accounting) の 3 種類のアカウンティングがあります。サイトにおけるアカウンティングのニーズに応じて、これらのアカウンティングのうちいずれか 1 つまたはこれらを組合わせて使用できます。この章では、CSA について詳しく説明します。

IRIX の 3 種類のアカウンティングを使用することで、特定の種類のシステム・アクティビティをログに記録し、課金できます。アカウンティング・データを使用することによって、システム・リソースの使用状況や、ユーザが妥当な量を超えるリソースを使用していないかどうかを判断したり、特定のユーザが特定の時刻に呼出したすべてのプロセスのリストを検査することによってセキュリティ違反などの重要なシステム・イベントをトレースしたり、あるいはシステム・リソースの使用料をログイン・アカウントに対して請求する課金システムを設定することが可能です。

基本アカウンティングは、標準的な UNIX アカウンティング機能で構成されます。基本アカウンティングはプロセス指向です。つまり、実行されたプロセスごとに新しいアカウンティング・レコードが生成されます。このレコードには、個々のプロセスによって使用されるリソースに関する統計情報が含まれます。`runacct`(1M) コマンドは、通常 `cron`(1M) によって起動される、主要な日別アカウンティング用シェル・スクリプトです。`runacct`(1M) コマンドは、プロセスのアカウンティング・データ・ファイルに書込まれるアカウンティング・レコードを処理します。

拡張アカウンティングは IRIX の機能で、プロジェクトと array セッションのアカウンティング機能に併せて、プロセスの拡張アカウンティング機能を備えています。アカウンティング・データをアカウンティング・データ・ファイルに直接書込む基本アカウンティングや CSA とは異なり、拡張アカウンティングではシステム監査トレール (SAT: System Audit Trail) 機能を使用してデータがデータ・ファイルに書込まれます。監査データは、`satd`(1M) プログラムによってカーネルから直接収集されます。拡張アカウンティングのデータは、基本アカウンティングによって収集、報告されるデータのスーパーセットです。

CSA は、より詳細で正確なアカウンティング・データをジョブごとに提供する付加機能を持っています。また、一部のデーモンのデータも保有します。`csarun`(1M) コマンドは通常 `cron`(1M) コマンドによって起動され、CSA の日別アカウンティング・ファイルの処理を指示します。`csarun`(1M) コマンドは、CSA のアカウンティング・データ・ファイルに書込まれるアカウンティング・レコードを処理します。

基本アカウンティングおよび拡張アカウンティングについての詳細は、『IRIX Admin: Backup, Security and Accounting』の第 7 章「System Accounting」の「About the Process Accounting System」および「IRIX Extended Accounting」をそれぞれ参照してください。

メモ: このリリースから、CSA は、IRIX フィーチャ・ストリームとメンテナンス・ストリームの両方でサポートされるようになりました。

この章は、以下の節で構成されています。

最初にお読みください

この章の各節では、お使いのシステムに CSA ソフトウェアをインストールする方法について説明します。以下の順序で参照してください。

1. CSA の一般的な説明については、71 ページの「CSA の概要」を参照してください。
2. CSA パッケージと、CSA で使用されるジョブ制限パッケージをインストールする方法については、74 ページの「CSA の有効化と無効化」を参照してください。
3. CSA のディレクトリとファイルについての詳細は、75 ページの「CSA のファイルとディレクトリ」を参照してください。
4. システムでの CSA の設定、日常の操作、システムに合わせた CSA のカスタマイズなど、CSA についての詳細は、82 ページの「完全システム・アカウンティングに関する詳細」を参照してください。
5. CSA のマン・ページのリストについては、121ページの「CSA のマン・ページ」を参照してください。
6. CSA を使用して生成できるレポートのタイプについての詳細は、115ページの「CSA レポート」を参照してください。

CSA の概要

完全システム・アカウンティング (CSA: Comprehensive System Accounting) は、C プログラムとシェル・スクリプトのセットです。ほかのアカウンティング・パッケージと同様に、プロセスごとのリソース使用状況データの収集、ディスク使用量のモニタ、および特定のログイン・アカウントに対して料金を請求するための方法を提供します。CSA の特長を以下に示します。

- ジョブごとのアカウンティング
- デーモンのアカウンティング（テープ、NQS、およびワークロード管理システム）
- 柔軟なアカウンティング期間（日別または定期（月別）アカウンティング・レポートをいつでも作成でき、1 日に 1 回または月に 1 回に制限されません）
- 柔軟なシステム課金単位 (SBU: System Billing Unit)
- アカウンティング・データのオフライン・アーカイブ
- 日別および定期（月別）アカウンティングについてサイト固有のカスタマイズを行うためのユーザ出口
- `/etc/csa.conf` ファイル内の設定可能パラメータ
- ユーザ・ジョブのアカウンティング（`ja`(1) コマンド）

CSA は、システム起動期間内にプロセスごとのアカウンティング情報を収集し、それをジョブ識別子 (`jid`) 別に結合します。ジョブの CSA アカウンティングは、特定のジョブ識別子の、1 回のシステム起動期間中におけるすべてのアカウンティング・データで構成されます。ただし、NQS ジョブやワークロード管理ジョブは、複数の再起動にまたがり、複数のジョブ識別子で構成される場合があります。したがって、このようなジョブの CSA アカウンティングには、NQS 識別子やワークロード管理識別子に関連付けられたアカウンティング・データが含まれます。

デーモンのアカウンティング・レコードは、デーモン固有のイベントが終了するときに書込まれます。これらのレコードは、同じジョブに関連付けられたプロセスごとのアカウンティング・レコードと結合されます。

CSA は、デフォルトで、終了したジョブのアカウンティング・データのみを報告します。対話型ジョブ、`cron` ジョブ、および `at` ジョブは、ジョブ内の最後のプロセスが終了すると、通常はログイン・シェルで終了します。NQS ジョブまたはワークロード管理ジョブは、そのジョブのデーモン・アカウンティング・レコードおよび EOJ (end-of-job) レコードに基づいて、終了が認識されます。アクティブなジョブは、次のアカウンティング期間にリサイクルされます。この動作は、`csarun` コマンドの `-A` オプションを使用して変更できます。

システム課金単位 (SBU: System Billing Unit) は、マシン・リソースの使用量を表す測定単位です。SBU は CSA 設定ファイル `/etc/csa.conf` で定義され、デフォルトで `0.0` に設定されます。CSA のアカウンティング・レコード内の各フィールドに関連付けられた重み係数を変更することによって、個々のサイトにとって適切な SBU 値が得られます。SBU についての詳細は、101ページの「システム課金単位 (SBU)」を参照してください。

CSA のアカウンティング・レコードは、基本アカウンティングの pacct ファイルではなく、CSA 用の独立した /var/adm/acct/day/pacct ファイルに書込まれます。CSA のコマンドは、CSA で生成されるアカウンティング・レコードに対してのみ使用できます。同様に、基本アカウンティングのコマンドは、基本アカウンティングで生成されるレコードに対してのみ使用できます。

csarun(1M) 日別アカウンティング・スクリプトで使用できるユーザ出口は 4 つあります。csaperiod(1M) 月別アカウンティング・スクリプトで使用できるユーザ出口は 1 つです。これらのユーザ出口を使用することによって、追加処理の実行や、アカウンティング・データのアーカイブを行うユーザ出口スクリプトを作成して、毎日または毎月のアカウンティングの実行を各サイトのニーズに応じてカスタマイズできます。詳細については、csarun(1M) および csaperiod(1M) のマン・ページを参照してください。

CSA では、2 つのユーザ・アカウンティング・コマンドである csacom(1) と ja(1) が用意されています。csacom コマンドは、CSA の pacct ファイルを読取り、選択されたアカウンティング・レコードを標準出力に書込みます。csacom コマンドは、基本アカウンティングの acctcom(1) コマンドに非常に良く似ています。ja コマンドは、呼出し元の現在のジョブについてのジョブ・アカウンティング情報を提供します。この情報は、カーネルが書込みを行う、別個のユーザ・ジョブ・アカウンティング・ファイルから取得されます。詳細については、csacom(1) および ja(1) のマン・ページを参照してください。

/etc/csa.conf ファイルには、CSA 設定変数が記述されます。これらの変数は CSA コマンドによって使用されます。

ほかのアカウンティング・パッケージやモニタリング・パッケージと同様に、CSA の機能によって全体的なシステム・オーバーヘッドが高くなります。そのため、CSA はデフォルトでカーネル内で無効になっています。CSA を有効にする方法については、74 ページの「CSA の有効化と無効化」を参照してください。

概念と用語

以下の概念と用語は、アカウンティング機能を使用する上で理解しておくことが重要です。

用語	説明
日別アカウンティング	日別アカウンティングとは、raw アカウンティング・データの処理、整理、および報告を、通常 1 日 1 回実行することです。 基本アカウンティングの場合、日別アカウンティングは 1 日 1 回しか実行できません。CSA では、必要に応じて 1 日に何回でも実行できますが、そのような場合でも、この機能は日別アカウンティングと呼ばれます。

ジョブ

ジョブとは、システムが独立した1つの実体として処理するプロセスのグループで、一意のジョブ識別子(ジョブ ID)によって識別されます。

CSA は、アカウンティング・データをジョブ単位および起動回数単位で整理し、データを `sorted pacct` ファイルに書込む唯一のアカウンティング手法です。

NQS またはワークロード管理以外のジョブの場合、ジョブは、特定のジョブ ID の、1 回の起動期間中におけるすべてのアカウンティング・データで構成されます。

NQS ジョブは、ジョブの NQS シーケンス番号に関連付けられたすべてのジョブ ID のアカウンティング・データで構成され、ワークロード管理ジョブは、ワークロード管理要求 ID に関連付けられたすべてのジョブ ID のアカウンティング・データで構成されます。NQS ジョブまたはワークロード管理ジョブは、複数の起動期間にまたがることがあります。ジョブが再開される場合、そのジョブには、実行されるすべての起動期間を通じて、同じジョブ ID が関連付けられます。繰返し実行される NQS ジョブやワークロード管理ジョブには、複数のジョブ ID が割当てられます。CSA では、NQS ジョブまたはワークロード管理ジョブのすべての段階が、同一ジョブ内にあるものとして処理されます。

定期アカウンティング

定期(月別)アカウンティングは、日別アカウンティング・レポートを詳細に処理、報告、および要約して、システムの使用状況をさらに高いレベルから分析します。

基本アカウンティングの場合、定期アカウンティングとは月単位で実行するアカウンティングを指します。CSA の場合は、システム管理者がアカウンティング期間を指定でき、月単位または累積日数単位で実行します。そのため、定期アカウンティングは 1 か月に 1 回以上実行することもできますが、その場合でも月別アカウンティングと呼ばれます。

デーモンのアカウンティング

デーモンのアカウンティングとは、raw アカウンティング・データの処理、整理、および報告を、デーモン固有のイベントの終了時に実行することです。

リサイクル・データ

リサイクル・データとは、raw アカウンティング・データ・ファイル内に残されたデータのことで、アカウンティング・レポートの次の実行時まで保存されます。

デフォルトでは、アクティブなジョブのアカウンティング・データはジョブが終了するまでリサイクルされます。csarun コマンドに -A オプションが指定されないかぎり、CSA は終了したジョブのデータのみを報告します。csarun は、リサイクル・データを /var/adm/acct/day/pacct0 データ・ファイル内に格納します。

この章では、全体を通して以下の略語と定義を使用します。

略語	定義
MMDD	月、日
hhmm	時、分

CSA の有効化と無効化

CSA ジョブ・アカウンティングを設定するには、以下の手順を実行する必要があります。

1. inst(1M) ユーティリティを使用して、IRIX 配布メディアから eoe.sw.csaacct サブシステムをインストールします。CSA をインストールするには、eoe.sw.acct と eoe.sw.jlimits の各サブシステムもインストールされている必要があります。

2. systune(1M) ユーティリティを使用して do_csaacct を 0 以外の値に設定することによって、CSA をカーネル内で有効にします。システムは、この手順を終了した後で再起動する必要があります。

3. chkconfig(1M) ユーティリティで以下のように指定して、CSA を有効にします。この設定は、システムを再起動しても保持されます。

   ```
   chkconfig csaacct on
   ```

4. /etc/csa.conf 内の CSA 設定変数を必要に応じて変更します。

5. csaswitch(1M) コマンドを以下のように実行して、/etc/csa.conf で定義されたアカウンティング・レコードのタイプとしきい値を有効にします。

   ```
   csaswitch -c on
   ```

   chkconfig(1M) ユーティリティによって CSA が有効に設定されている場合、この後でシステムを再起動すると、この手順は自動的に実行されます。

   cron(1M) コマンドで日別アカウンティングが自動的に実行されるよう crontabs ファイルにエントリを追加する方法についての詳細は、83 ページの「CSA の設定」を参照してください。

CSA ジョブ・アカウンティングを無効にするには、以下の手順を実行する必要があります。

1. CSA を停止するには、csaswitch(1M) コマンドを以下のように指定します。

   ```
   csaswitch -c halt
   ```

2. システム再起動の後に CSA が起動しないようにするには、chkconfig(1M) コマンドを以下のように指定します。

   ```
   chkconfig csaacct off
   ```

3. systune(1M) ユーティリティを使用して do_csaacct を 0 に設定することによって、CSA をカーネル内で無効にします。システムは、この手順を終了した後で再起動する必要があります。

CSA のファイルとディレクトリ

以下の節では、CSA のファイルとディレクトリについて説明します。

/var/adm/acct ディレクトリ内のファイル

/var/adm/acct ディレクトリには、さまざまなサブディレクトリに分かれて CSA のデータ・ファイルとレポート・ファイルが含まれています。/var/adm/acct には、CSA で使用されるデータ収集ファイルが含まれています。CSA と IRIX の基本アカウンティングは、別個の pacct ファイルにアクセスします。以下の図に、CSA のディレクトリとファイルの構造を示します。

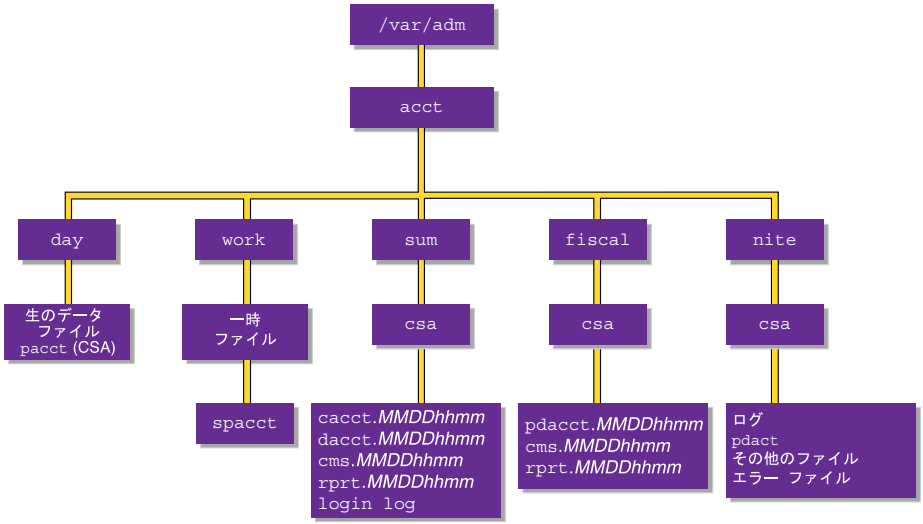

図5-1 /var/adm/acct ディレクトリ

CSA のデータ・ファイルとレポート・ファイルには、それぞれ「月-日-時-分」形式の接尾辞が付けられます。

警告: IRIX セキュリティ強化システムでは、csacom(1) コマンドは保護チャンネルとして扱われます。このコマンドの使用は adm グループにのみ制限することをお薦めします。

/var/adm/acct/ ディレクトリ内のファイル

/var/adm/acct ディレクトリには、以下のサブディレクトリがあります。

ディレクトリ	説明
day	現在の raw アカウンティング・データ・ファイルが pacct 形式で含まれています。
work	CSA で一時作業領域として使用されます。CSA 日別アカウンティング実行の開始時に /var/adm/acct/day から移動された raw データ・ファイルと spacct ファイルが含まれています。

sum/csa	csarun(1M) によって作成される、累積的な日別アカウンティングの集計ファイルとレポートが含まれています。ASCII 形式のデータは以下のファイルにあります。/var/adm/acct/sum/csa/rprt.*MMDDhhmm* バイナリ・データは以下のファイルにあります。 /var/adm/acct/sum/csa/cacct.*MMDDhhmm* /var/adm/acct/sum/csa/cms.*MMDDhhmm*、および /var/adm/acct/sum/csa/dacct.*MMDDhhmm*
fiscal/csa	csaperiod(1M) によって作成される、定期アカウンティングの集計ファイルとレポートが含まれています。ASCII 形式のデータは以下のファイルにあります。/var/adm/acct/fiscal/csa/rprt.*MMDDhhmm* バイナリ・データは以下のファイルにあります。 /usr/adm/acct/fiscal/csa/cms.*MMDDhhmm* および /usr/adm/acct/fiscal/csa/pdacct.*MMDDhhmm*
nite/csa	ログ・ファイル、csarun の状態ファイル、および実行日時記録ファイルが含まれています。

/var/adm/acct/day ディレクトリ内のファイル

/var/adm/acct/day ディレクトリには、以下のファイルがあります。

ファイル	説明
dodiskerr	ディスク・アカウンティング・エラー・ファイル
pacct	プロセスとデーモンのアカウンティング・データ
pacct0	プロセスとデーモンのリサイクル・アカウンティング・データ
dtmp	dodisk によって作成されるディスク・アカウンティング・データ (ASCII)

/var/adm/acct/work ディレクトリ内のファイル

/var/adm/acct/work/*MMDD/hhmm* ディレクトリには、以下のファイルがあります。

ファイル	説明
BAD.Wpacct*	無効なレコード(csaverify(1M) によって検証)が保存されている、未処理のアカウンティング・データ
Ever.tmp1	データ検証作業ファイル
Ever.tmp2	データ検証作業ファイル
Rpacct0	アカウンティングの次の実行時にリサイクルされる、プロセスとデーモンのアカウンティング・データ

Wdiskcacct	dodisk(1M) によって作成されるディスク・アカウンティング・データ (cacct.h 形式)。dodisk(1M) のマン・ページを参照してください。
Wdtmp	dodisk(1M) によって作成されるディスク・アカウンティング・データ (ASCII)
Wpacct*	プロセスとデーモンの raw アカウンティング・データ
spacct	sorted pacct ファイル

/var/adm/acct/sum/csa ディレクトリ内のファイル

/var/adm/acct/sum/csa ディレクトリには、以下のデータ・ファイルがあります。

ファイル	説明
cacct.*MMDDhhmm*	cacct.h 形式の統合日別データ。csaperiod コマンドで -r オプションを指定した場合、このファイルは削除されます。
cms.*MMDDhhmm*	コマンド集計 (cms) レコード形式の、コマンドの使用状況に関する日別データ。csaperiod コマンドで -r オプションを指定した場合、このファイルは削除されます。
dacct.*MMDDhhmm*	cacct.h 形式の、ディスク使用量に関する日別データ。csaperiod コマンドで -r オプションを指定した場合、このファイルは削除されます。
loginlog	lastlogin によって作成されるログイン・レコード・ファイル
rprt.*MMDDhhmm*	日別アカウンティング・レポート

/var/adm/acct/fiscal/csa ディレクトリ内のファイル

/var/adm/acct/fiscal/csa ディレクトリには、以下のファイルがあります。

ファイル	説明
cms.*MMDDhhmm*	コマンド集計 (cms) レコード形式の、コマンドの使用状況に関する定期データ
pdacct.*MMDDhhmm*	統合定期データ
rprt.*MMDDhhmm*	定期アカウンティング・レポート

/var/adm/acct/nite/csa ディレクトリ内のファイル

/var/adm/acct/nite/csa ディレクトリには、以下のファイルがあります。

ファイル	説明
active	csarun(1M) コマンドによって、コマンドの実行状況の記録や、警告およびエラー・メッセージの出力に使用されます。csarun がエラーを検出すると、active*MMDDhhmm* は active と同じになります。
clastdate	*MMDDhhmm* 形式の、csarun が最後に実行されたときの日時記録（2 回分）
dk2log	dodisk の実行中に作成される診断出力。83 ページの「CSA の設定」で、dodisk の cron のエントリを参照してください。
diskcacct	dodisk によって作成される、cacct.h 形式のディスク・アカウンティング・レコード
Eaddc*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csaaddc(1M) コマンドによって生成されるエラーまたは警告メッセージ
Earc1*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csa.archive1(1M) コマンドによって生成されるエラーまたは警告メッセージ
Earc2*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csa.archive2(1M) コマンドによって生成されるエラーまたは警告メッセージ
Ebld.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csabuild(1M) コマンドによって生成されるエラーまたは警告メッセージ
Ecmd.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、ASCII ファイルを生成するときに csacms(1M) コマンドによって生成されるエラーまたは警告メッセージ
Ecms.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、バイナリ・データ・ファイルを生成するときに csacms(1M) コマンドによって生成されるエラーまたは警告メッセージ
Econ.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csacon(1M) コマンドによって生成されるエラーまたは警告メッセージ
Ecrep.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csacrep(1M) コマンドによって生成されるエラーまたは警告メッセージ
Ecrpt.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csacrep(1M) コマンドによって生成されるエラーまたは警告メッセージ

Edrpt.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csadrep(1M) コマンドによって生成されるエラーまたは警告メッセージ
Erec.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csarecy(1M) コマンドによって生成されるエラーまたは警告メッセージ
Euser.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csa.user(1M) ユーザ出口によって生成されるエラーまたは警告メッセージ
Epuser.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csa.puser(1M) ユーザ出口によって生成されるエラーまたは警告メッセージ
Ever.tmp1*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csaverify(1M) コマンドによって生成される無効なレコード・オフセットの出力ファイル
Ever.tmp2*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csaverify(1M) コマンドによって生成されるエラーまたは警告メッセージ
Ever.*MMDDhhmm*	*MMDD* の *hhmm* に実行されたアカウンティングについて、csaedit(1M) および csaverify(1M) コマンド（Ever.tmp2 ファイル）によって生成されるエラーまたは警告メッセージ
fd2log	csarun の実行中に作成される診断出力。83 ページの「CSA の設定」で、csarun の cron のエントリを参照してください。
lock lock1	csarun(1M) コマンドの連続使用を制御します。
pd2log	csaperiod の実行中に作成される診断出力。83 ページの「CSA の設定」で、csaperiod の cron のエントリを参照してください。
pdact	csaperiod の実行状況とステータス。csaperiod がエラーを検出すると、pdact.*MMDDhhmm* は pdact と同じになります。
statefile	csarun コマンドの実行中に現在の状態を記録します。

/usr/lib/acct ディレクトリ

/usr/lib/acct ディレクトリには、CSA で使用される以下のコマンドとシェル・スクリプトが含まれています。

コマンド	説明
csaaddc	複数の *cacct* レコードを結合します。

csabuild	アカウンティング・レコードをジョブ・レコードにまとめます。
csachargefee	ユーザに対して料金を請求します。
csackpacct	CSA プロセス・アカウンティング・ファイルのサイズをチェックします。
csacms	プロセスごとのアカウンティング・レコードから、コマンドの使用状況を集計します。
csacon	sorted pacct ファイルのレコードを圧縮します。
csacrep	統合アカウンティング・データについて報告します。
csadrep	デーモンの使用状況について報告します。
csaedit	アカウンティング情報を表示、編集します。
csagetconfig	アカウンティング設定ファイル内で、指定した引数を検索します。
csajrep	sorted pacct ファイルのジョブ・レポートを出力します。
csaperiod	定期アカウンティングを実行します。
csarecy	未完了のジョブのレコードをアカウンティングの次の実行時にリサイクルします。
csarun	日別アカウンティング・ファイルを処理し、レポートを生成します。
csaswitch	異なるタイプの 完全システム・アカウンティング (CSA: Comprehensive System Accounting) のステータスをチェックしたり、有効または無効に切替えたり、アカウンティング・ファイルを保守の目的で交換したりします。
csaverify	アカウンティング・レコードが有効であることを確認します。

/usr/bin ディレクトリには、CSA 関連のユーザ・コマンドが含まれています。

コマンド	説明
ja	ユーザ・ジョブのアカウンティング情報の取得を開始、停止します。
csacom	CSA のプロセス・アカウンティング・ファイルを検索、出力します。

ユーザ出口を利用して、日別アカウンティング時にサイト固有の追加処理を実行するスクリプトを作成することで、csarun または csaperiod プロシージャをサイトの固有のニーズに合わせてカスタマイズできます。サイトでアカウンティング・ユーザ出口を使用する場合は、実行パーミッションを持つ adm が所有するユーザ出口ファイルを作成する必要があります。システムをアップグレードした場合は、ユーザ出口を再作成する必要があります。サイト固有のユーザ出口の設定の詳細、およびユーザ出口スクリプトの例については、107ページの「ユーザ出口の設定」を参照してください。

サイトでアカウンティング・ユーザ出口を使用する場合、/usr/lib/acct ディレクトリには以下のスクリプトも含まれる場合があります。

スクリプト	説明
csa.archive1	サイトで生成される csarun のユーザ出口
csa.archive2	サイトで生成される csarun のユーザ出口
csa.fef	サイトで生成される csarun のユーザ出口
csa.user	サイトで生成される csarun のユーザ出口
csa.puser	サイトで生成される csaperiod のユーザ出口

/etc ディレクトリ

/etc ディレクトリには、CSA ソフトウェアで使用されるパラメータのラベルと値を含む csa.conf ファイルがあります。

/etc/config ディレクトリ

/etc/config ディレクトリには、chkconfig(1M) コマンドによって使用される csaacct ファイルがあります。csaacct.options には、csaswitch(1M) コマンドに渡されるオプションがあります。テキスト・エディタを使用して、システム起動時にのみ csaswitch に渡される任意の csaswitch(1M) オプションを追加します。

完全システム・アカウンティングに関する詳細

この節では、CSA についての詳細と、以下のトピックについて説明します。

- 83 ページの「日別操作の概要」
- 83 ページの「CSA の設定」
- 87 ページの「csarun コマンド」
- 91 ページの「データ・ファイルの確認と編集」
- 91 ページの「CSA のデータ処理」
- 95 ページの「データのリサイクル」
- 100ページの「CSA のカスタマイズ」

日別操作の概要

IRIX オペレーティング・システムがマルチユーザ・モードで実行しているとき、アカウンティングは以下のように動作します。ただし、各サイトで CSA をカスタマイズできるので、以下の動作は個々のサイトにおける実際の動作とは異なる場合もあります。

1. CSA アカウンティングが有効になっていると、システムがマルチユーザ・モードに切替わるときに /usr/lib/acct/csaswitch コマンド（csaswitch(1M) のマン・ページを参照）が /etc/rc2 によって呼出されます。

2. デフォルトでは、csa、メモリ、および I/O レコードのタイプが、/etc/csa.conf で有効になっています。ただし、NQS、ワークロード管理、またはテープ・デーモンのアカウンティングを実行するには、/etc/csa.conf ファイルと、該当するサブシステムに変更を加える必要があります。詳細については、83 ページの「CSA の設定」を参照してください。

3. 各ユーザが使用しているディスク領域の量は、定期的にチェックされます。/usr/lib/acct/dodisk コマンド（dodisk(1M) を参照）が cron コマンドによって定期的に実行され、各ユーザが使用しているディスク領域量のスナップショットが生成されます。dodisk コマンドの実行は、/usr/lib/acct/csarun の 1 回の実行につき 1 回までです（csarun(1M) を参照してください）。同じアカウンティング期間中に dodisk が複数回起動されると、前の dodisk 出力は上書きされます。

4. 課金ファイルが作成されます。特定のユーザに対して料金を請求したいサイトでは、/usr/lib/acct/csachargefee を呼出すことによって課金できます（csachargefee(1M) を参照してください）。各アカウンティング期間の課金ファイル (/var/adm/acct/day/fee) は、/usr/lib/acct/csaperiod（csaperiod(1M) を参照）によって、統合アカウンティング・レコードに結合されます。

5. 日別アカウンティングが実行されます。1 日の指定された時刻に、csarun が cron コマンドによって実行され、現在のアカウンティング・データが処理されます。csarun からの出力は、日別アカウンティング・ファイルと ASCII 形式のレポートです。

6. 定期（月別）アカウンティングが実行されます。1 日の特定の時刻または 1 か月の特定の日に、/usr/lib/acct/csaperiod（csaperiod を参照）が cron コマンドによって実行され、前のアカウンティング期間からの統合アカウンティング・データが処理されます。csaperiod からの出力は、定期（月別）アカウンティング・ファイルと ASCII 形式のレポートです。

7. アカウンティングが無効化されます。システムを正常にシャットダウンすると、csaswitch(1M) コマンドが実行され、CSA のすべてのプロセスとデーモンのアカウンティングが停止されます。

CSA の設定

ここでは、CSA の設定方法について簡単に説明します。サイトごとの変更方法については、100ページの「CSA のカスタマイズ」で詳しく説明します。この節で説明するように、CSA はスーパー・ユーザのパーミッ

ションを持つユーザが実行します。CSA は、adm グループに所属し、CAP_ACCT_MGT capability を持っているユーザが実行することもできます。プロセスの操作権に対する細かな調整を可能にする capability についての詳細は、capability(4) および capabilities(4) のマン・ページを参照してください。必要な変更については、113ページの「スーパー・ユーザ以外による CSA の実行許可」を参照してください。

1. 必要に応じて、システム課金単位 (SBU: System Billing Unit) のデフォルトの重み係数を変更します。デフォルトでは、SBU は計算されません。サイトで SBU を報告したい場合は、設定ファイル /etc/csa.conf を変更する必要があります。

2. /etc/csa.conf ファイルで、必要なパラメータを変更します。このファイルには、アカウンティング・システム用に設定可能なパラメータが記述されています。

3. デーモンのアカウンティングが必要な場合は、以下の手順を実行して、システム起動時にデーモンのアカウンティングを有効にしてください。

   a. デーモンのアカウンティングを有効にするサブシステムについて、/etc/csa.conf の変数が on になっていることを確認します。NQS_START を on に設定すると、NQS のアカウンティングが有効になります。WKMG_START を on に設定すると、ワークロード管理のアカウンティングが有効になります。TAPE_START を on に設定すると、テープのアカウンティングが有効になります。

   b. 必要な場合は、デーモンの側からアカウンティングを有効にします。特に、NQS、ワークロード管理、およびテープのアカウンティングは、関連デーモンからも有効にする必要があります。qmgr set accounting on コマンドを使用して、NQS のアカウンティングを有効にします。テープ・デーモンのアカウンティングを有効にするには、tmdaemon に -c オプションを指定して実行します。tmdaemon コマンドについての詳細は、『TMF Administrator's Guide』を参照してください。ワークロード管理のアカウンティングを有効にする方法については、お使いのシステム用のワークロード管理ガイドを参照してください。

4. root で crontab(1) コマンドに -e オプションを指定して実行し、以下のようなエントリを追加します。

メモ: crontab(1) コマンドを使用せずに crontab ファイルを更新する場合は（vi(1) エディタを使用する場合など）、ファイルを更新した後に cron(1M) にシグナルを送信する必要があります。crontab コマンドを使用してファイルを編集した場合、ファイルを保存してエディタを終了したときに自動的に crontab ファイルが更新され、cron(1M) にシグナルが送信されます。crontab コマンドについての詳細は、crontab(1) のマン・ページを参照してください。

```
0 4 *  * 1-6  if /etc/chkconfig csaacct; then /usr/lib/acct/csarun 2> /var/adm/acct/nite/csa/fd2log; fi
0 2 *  * 4    if /etc/chkconfig csaacct; then /usr/lib/acct/dodisk -c > /var/adm/acct/nite/csa/dk2log; fi
5 * *  * 1-6  if /etc/chkconfig csaacct; then /usr/lib/acct/csackpacct; fi
0 5 1  * *    if /etc/chkconfig csaacct; then /usr/lib/acct/csaperiod -r  \
2> /var/adm/acct/nite/csa/pd2log; fi
```

これらのエントリについては、以下の手順で説明します。

a. たいていのインストールでは、/var/spool/cron/crontabs/root に以下のようなエントリを追加して、日別アカウンティングが cron(1M) によって自動的に実行されるようにします。

```
0 4 *  * 1-6  if /etc/chkconfig csaacct; then /usr/lib/acct/csarun 2> /var/adm/acct/nite/csa/fd2log; fi
0 2 *  * 4    if /etc/chkconfig csaacct; then /usr/lib/acct/dodisk -c > /var/adm/acct/nite/csa/dk2log; fi
```

csarun(1m) コマンドは、dodisk が完了するのに十分な時間を置いてから実行する必要があります。csarun が実行される前に dodisk が完了しなかった場合、ディスク・アカウンティング情報が見つからないか、不完全になる可能性があります。

dodisk コマンドは -c オプションを指定して呼出す必要があります。詳細については、dodisk(1M) のマン・ページを参照してください。

b. pacct ファイルのサイズを定期的にチェックします。/var/spool/cron/crontabs/root 内に以下のようなエントリを追加する必要があります。

```
5 * *  * 1-6  if /etc/chkconfig csaacct; then /usr/lib/acct/csackpacct; fi
```

この cron コマンドで csackpacct(1m) シェル・スクリプトを定期的に実行する必要があります。pacct ファイルが 4000 個の 1K ブロック(デフォルト)より大きくなると、csackpacct は、コマンド /usr/lib/acct/csaswitch -c switch を呼出して新しい pacct ファイルを作成します。csackpacct コマンドは、/var/adm/acct(デフォルトで /var ディレクトリに格納されている)があるファイルシステム上に最低 2000 個の 1K の空きブロックがあることも確認します。十分な空きブロックがない場合、CSA アカウンティングは無効になります。csackpacct の次回実行時に十分な空きブロックがあれば、CSA アカウンティングが有効になります。

MIN_BLKS 変数が /etc/csa.conf 設定ファイル内で正しく設定されていることを確認します。MIN_BLKS は、var/adm/acct ディレクトリが存在するファイルシステムで必要な未使用の 1K ブロックの最低数です。デフォルトは 2000 です。

アカウンティング・ファイルシステム(デフォルトで /var ディレクトリに格納されている)のディスク領域が不足したときに管理者に通知が送られるようにするため、csackpacct を定期的に実行することは非常に重要です。ファイルシステムのクリーンアップ後は、csackpacct を次に呼出したときに、プロセスとデーモンのアカウンティングが有効になります。csaswitch -c on を呼出すことによって、アカウンティングを手動で再有効化することもできます。

csackpacct を定期的に実行しなかった場合、アカウンティング・ファイルシステムのディスク領域が不足すると、書込みエラーが発生してアカウンティングが無効になったことを知らせるエラー・メッセージがコンソールに書込まれます。ディスク領域を直ちに解放しないと、大量のアカウンティング・データが不必要に失われる可能性もあります。また、アカウンティング・データが失われたことが原因で、csarun が異常終了したり、誤った情報が報告される可能性もあります。

c. 月別アカウンティングを実行するには、/var/spool/cron/crontabs/root に以下のようなエントリを追加する必要があります。このコマンドは、/var/adm/acct/sum/csa/* で見つかったすべての統合データ・ファイルについて月別レポートを生成し、これらのデータ・ファイルを削除します。

```
0 5 1  * *    if /etc/chkconfig csaacct; then /usr/lib/acct/csaperiod -r \
2> /var/adm/acct/nite/csa/pd2log; fi
```

このエントリは、csarun の完了に十分な時間があるときに実行されます。この例では、各月の第 1 日目に定期アカウンティング・ファイルが作成され、報告されます。これらのファイルには、前の月のアカウンティングに関する情報が含まれます。

5. Trusted IRIX システムでは、以下の手順を実行します。

a. user adm が CAP_ACCT_MGT capability を持っていることを確認します。

b. 以下のユーザ出口が存在する場合は、user adm によって読取り可能および実行可能であることを確認します。

- /usr/lib/acct/csa.archive1
- /usr/lib/acct/csa.archive2
- /usr/lib/acct/csa.fef
- /usr/lib/acct/csa.puser

c. /var/spool/cron/crontabs/root に以下を参考にしてエントリを追加します。

```
 2 * * 4 suattr -M dbadmin -C CAP_DAC_READ_SEARCH,CAP_DAC_WRITE,
CAP_FOWNER,CAP_MAC_READ+eip -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/dodisk -c 2> /var/adm/acct/nite/csa/dk2log; fi"
```

d. /var/spool/cron/crontabs/adm に以下を参考にしてエントリを追加します。

```
0 4 * * 1-6 su adm -C CAP_ACCT_MGT+pi -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/csarun 2> /var/adm/acct/nite/csa/fd2log; fi"
5 * * * 1-6 su adm -C CAP_ACCT_MGT+pi -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/csackpacct; fi"
0 5 1 * * if /etc/chkconfig csaacct;
then /usr/lib/acct/csaperiod -r 2> /var/adm/acct/nite/csa/pd2log; fi
```

6. holidays ファイルを更新します。ファイル /usr/lib/acct/holidays には、アカウンティング・システムのプライム・テーブルと非プライム・テーブルが含まれます。このテーブルを編集して、その年のサイトの休日スケジュールを反映させる必要があります。テーブルの形式は以下の 3 種類のエントリで構成されます。

- コメント行。行の先頭文字はアスタリスクにします。ファイル内の任意の場所に配置できます。
- 年指定行。この行はファイル内の（コメント行を除く）先頭のデータ行である必要があり、一度だけ指定します。この行は、それぞれ 4 桁の 3 つのフィールドで構成されます（先頭の空白は無視されます）。たとえば、年を 1992、プライム時刻を 9:00 a.m、非プライム時刻を 4:30 p.m にそれぞれ指定するには、以下のようなエントリが適切です。

  ```
  1992 0900 1630
  ```

  時刻フィールドでは、時刻 2400 は 0000 に自動的に変換されます。
- 営業休日行。この行は年指定行の後に続き、以下の一般的な形式を持ちます。

  ```
  day-of-year Month Day Description of Holiday
  ```

  年の通算日フィールドには 1～366 の範囲の数値が入り、対応する休日を示します（先頭の空白は無視されます）。その他の 3 つのフィールドはコメント用に用意されているもので、現在はほかのプログラムで使用されません。

csarun コマンド

/usr/lib/acct/csarun コマンドは、通常 cron(1) によって起動され、日別アカウンティング・ファイルの処理を指示します。csarun は、pacct ファイルに書込まれたアカウンティング・レコードを処理します。このコマンドは、通常、非プライム時間に cron によって起動されます。

csarun コマンドには、4 つのユーザ出口ポイントもあります。これによって、サイトのニーズに応じて毎日のアカウンティングの実行をカスタマイズできます。

csarun コマンドは、エラーが発生した場合でもファイルを壊しません。このコマンドには一連の保護メカニズムが組込まれており、エラーを認識し、インテリジェント診断を行って、csarun を最低限の操作で再開できるような方法で処理を終了します。

毎日の呼出し

csarun コマンドは、cron によって定期的に呼出されます。新しいアカウンティング期間について csarun を呼出すときは、前回の csarun 呼出しが正常に完了したことを確認することが非常に重要です。そうしなかった場合、未完了のジョブに関する情報が不正確になります。

以下のコマンドを実行することによって、新しいアカウンティング期間のデータを対話的に処理することもできます。

```
nohup csarun 2> /var/adm/acct/nite/csa/fd2log &
```

この方法で csarun を実行するときは、前回の呼出しが正常に完了していることを確認します。完了の確認は、/var/adm/acct/nite/csa にある active と statefile の各ファイルで行います。ど

ちらのファイルも、前回の呼出しが正常に完了していることを示している必要があります。89 ページの「csarun の再開」を参照してください。

エラー・メッセージとステータス・メッセージ

csarun によって生成されるエラー・メッセージとステータス・メッセージは、/var/adm/acct/nite/csa ディレクトリに格納されます。実行の進行状況は、状況説明のメッセージをファイル active に書込むことによって追跡されます。csarun 実行中の診断出力は、fd2log に書込まれます。lock ファイルと lock1 ファイルによって、csarun の多重呼出しが防止されます。呼出し時にこの 2 つのファイルがあった場合、csarun は異常終了します。clastdate ファイルには、csarun の最後の 2 回の実行の月、日、および時刻が記録されます。

csarun から呼出されるプログラムのエラーと警告メッセージは、ファイルに書込まれます。ファイルの名前は、E で始まり、現在の日付と時刻で終わります。たとえば、Ebld.11121400 は、11 月 12 日の 14:00 に csabuild によって呼出された csarun のエラー・ファイルです。

csarun は、エラーを検出すると、SYSLOG ファイルにメッセージを書込み、ロックを削除します。次に、診断ファイルを保存し、実行を終了します。また、csarun は、設定ファイル /etc/csa.conf に定義されているように、致命的エラーの場合は MAIL_LIST にメールを送信し、警告メッセージの場合は WMAIL_LIST にメールを送信します。

状態

csarun を再開できるよう、処理は複数の独立した再入可能状態に分割されています。各状態が完了すると、/var/adm/acct/nite/csa/statefile が更新され、次の状態が反映されます。csarun は CLEANUP 状態に達すると、さまざまなデータ・ファイルとロックを削除し、終了します。

以下に、各状態で発生するイベントについて説明します。*MMDD* は csarun が呼出された月と日、*hhmm* は時と分をそれぞれ指します。

状態	説明
SETUP	現在のアカウンティング・ファイルが csaswitch によって切替えられます。次に、アカウンティング・ファイルが /var/adm/acct/work/*MMDD*/*hhmm* ディレクトリに移動されます。ファイル名の先頭には W が付けられます。/var/adm/acct/nite/csa/diskcacct も同じディレクトリに移動されます。
VERIFY	アカウンティング・ファイルが有効なデータを含んでいるかどうかチェックされます。無効なデータを持つレコードは削除されます。不正なデータ・ファイルの名前には、/var/adm/acct/work/*MMDD*/*hhmm* ディレクトリで BAD. というプレフィックスが付けられます。修正されたファイルには、このプレフィックスは付けられません。

ARCHIVE1　csarun スクリプトの最初のユーザ出口。/usr/lib/acct/csa.archive1 という名前のスクリプトが存在する場合、そのスクリプトがシェルの .(ドット)コマンドで実行されます。(ドット)コマンドではコンパイルされたプログラムは実行できませんが、ユーザ出口スクリプトの中から実行できます。このユーザ出口を使用して、${WORK} 内のアカウンティング・ファイルをアーカイブできます。

BUILD　pacct アカウンティング・データが、sorted pacct ファイル内にまとめられます。

ARCHIVE2　csarun スクリプトの 2 番目のユーザ出口。/usr/lib/acct/csa.archive2 という名前のスクリプトが存在する場合、そのスクリプトがシェルの .(ドット)コマンドで実行されます。(ドット)コマンドではコンパイルされたプログラムは実行できませんが、ユーザ出口スクリプトの中から実行できます。このユーザ出口を使用して、sorted pacct ファイルをアーカイブできます。

CMS　コマンド集計ファイルを cms.h 形式で生成します。cms ファイルは、csaperiod によって使用できるよう /var/adm/acct/sum/csa/cms.*MMDDhhmm* に書込まれます。

REPORT　日別アカウンティング・レポートを生成して、/var/adm/acct/sum/csa/rprt.*MMDDhhmm* に格納します。統合データ・ファイル /var/adm/acct/sum/csa/cacct.*MMDDhhmm* も、sorted pacct ファイルから生成されます。また、未完了ジョブのアカウンティング・データはリサイクルされます。

DREP　sorted pacct ファイルに基づいて、デーモンの使用状況に関するレポートを生成します。このレポートは、日別アカウンティング・レポート /var/adm/acct/sum/csa/rprt.*MMDDhhmm* に追加されます。

FEF　csarun スクリプトの 3 番目のユーザ出口。/var/lib/acct/csa.fef という名前のスクリプトが存在する場合、そのスクリプトがシェルの .(ドット)コマンドで実行されます。(ドット)コマンドではコンパイルされたプログラムは実行できませんが、ユーザ出口スクリプトの中から実行できます。csarun 変数は、エクスポートされていなくても、ユーザ出口スクリプトで利用できます。この出口を使用して、sorted pacct ファイルをフロントエンド・システムにとって適切な形式に変換できます。

USEREXIT　csarun スクリプトの 4 番目のユーザ出口。/usr/lib/acct/csa.user という名前のスクリプトが存在する場合、そのスクリプトがシェルの .(ドット)コマンドで実行されます。(ドット)コマンドではコンパイルされたプログラムは実行できませんが、ユーザ出口スクリプトの中から実行できます。csarun 変数は、エクスポートされていなくても、ユーザ出口スクリプトで利用できます。この出口を使用して、ローカルなアカウンティング・プログラムを実行できます。

CLEANUP　一時ファイルをクリーンアップし、ロックを削除してから、終了します。

csarun の再開

csarun が引数なしで実行された場合、前回の呼出しは正常に完了したものと解釈されます。

csarun を再開する場合は、以下のオペランドが必要です。

```
csarun [MMDD [hhmm [state]]]
```

MMDD は月と日、*hhmm* は時と分、*state* は csarun のエントリ状態をそれぞれ示します。

csarun を再開するには、以下の手順に従います。

1. 以下のコマンド・ラインを使用して、すべてのロック・ファイルを削除します。

   ```
   rm -f /var/adm/acct/nite/csa/lock*
   ```

2. 以下の例を参考にして、適切な csarun 再開コマンドを実行します。

   a. clastdate と statefile で指定された時刻と状態を使用して csarun を再開するには、以下のコマンドを実行します。

      ```
      nohup csarun 0601 2> /var/adm/acct/nite/csa/fd2log &
      ```

      この例の場合、csarun は、clastdate と statefile で指定された時刻と状態を使用して、6 月 1 日の実行が再開されます。

   b. statefile で指定された状態を使用して csarun を再開するには、以下のコマンドを実行します。

      ```
      nohup csarun 0601 0400 2> /var/adm/acct/nite/csa/fd2log &
      ```

      この例の場合、csarun は、statefile で見つかった状態を使用して、6 月 1 日の午前 4 時に呼出された実行が再開されます。

   c. 指定された日付、時刻、および状態を使用して csarun を再開するには、以下のコマンドを実行します。

      ```
      nohup csarun 0601 0400 BUILD 2> /var/adm/acct/nite/csa/fd2log &
      ```

      この例の場合、csarun は、6 月 1 日の午前 4 時に開始された呼出しが、状態 BUILD で再開されます。

csarun を再開する前に、適切なディレクトリを復元する必要があります。ディレクトリを復元しなかった場合、それ以上の処理は実行できません。復元する必要のあるディレクトリは、以下のとおりです。

```
/var/adm/acct/work/MMDD/hhmm
/var/adm/acct/sum/csa
```

ARCHIVE2、CMS、REPORT、DREP、または FEF という状態で再開する場合、sorted pacct ファイルは /var/adm/acct/work/*MMDD*/*hhmm* 内にある必要があります。このファイルがない場合、csarun は、自動的に BUILD 状態で再開します。サイト固有の USEREXIT 状態にあるときに実行されたタスクによっては、sorted pacct ファイルが必要な場合と必要でない場合があります。この実行は、受入れられる場合と受入れられない場合があります。

データ・ファイルの確認と編集

この節では、さまざまなアカウンティング・ファイルから不正なデータを削除する方法について説明します。

csaverify(1M) コマンドは、アカウンティング・レコードが有効であるかどうかを確認し、無効なレコードを特定します。アカウンティング・ファイルは、pacct ファイルまたは sorted pacct ファイルです。csaverify は、無効なレコードを検出すると、レコードの開始バイト・オフセットと長さを報告します。この情報は、標準出力以外に、ファイルにも書込むことができます。長さ -1 は、ファイルの終わりを示します。生成された出力ファイルは csaedit(1M) への入力として使用して、pacct レコードまたは sorted pacct レコードを削除できます。

1. pacct ファイルは、以下のコマンド・ラインを使用して確認します。また、以下の出力が生成されます。

```
$  /usr/lib/acct/csaverify -P pacct -o offsetfile
acct.cat-330 /usr/lib/acct/csaverify: CAUTION
   readacctent(): An error was returned from the
'readpacct()' routine.
```

2. 以下に示すとおり、csaverify で生成されたファイル offsetfile を csaedit への入力として使用し、無効なレコードを削除します（残りの有効なレコードは pacct.NEW に書込まれます）。

```
/usr/lib/acct/csaedit -b offsetfile -P pacct -o pacct.NEW
```

3. 新しい pacct ファイルを以下のように再確認し、すべての不正レコードが削除されたことを確認します。

```
/usr/lib/acct/csaverify -P pacct.NEW
```

csaedit -A オプションを使用すると、省略化された ASCII バージョンの pacct ファイルまたは sorted pacct ファイルを生成できます。

CSA のデータ処理

この節では、さまざまな CSA プログラム間におけるデータの流れについて説明します。このデータの流れの様子を図5-2 に示します。

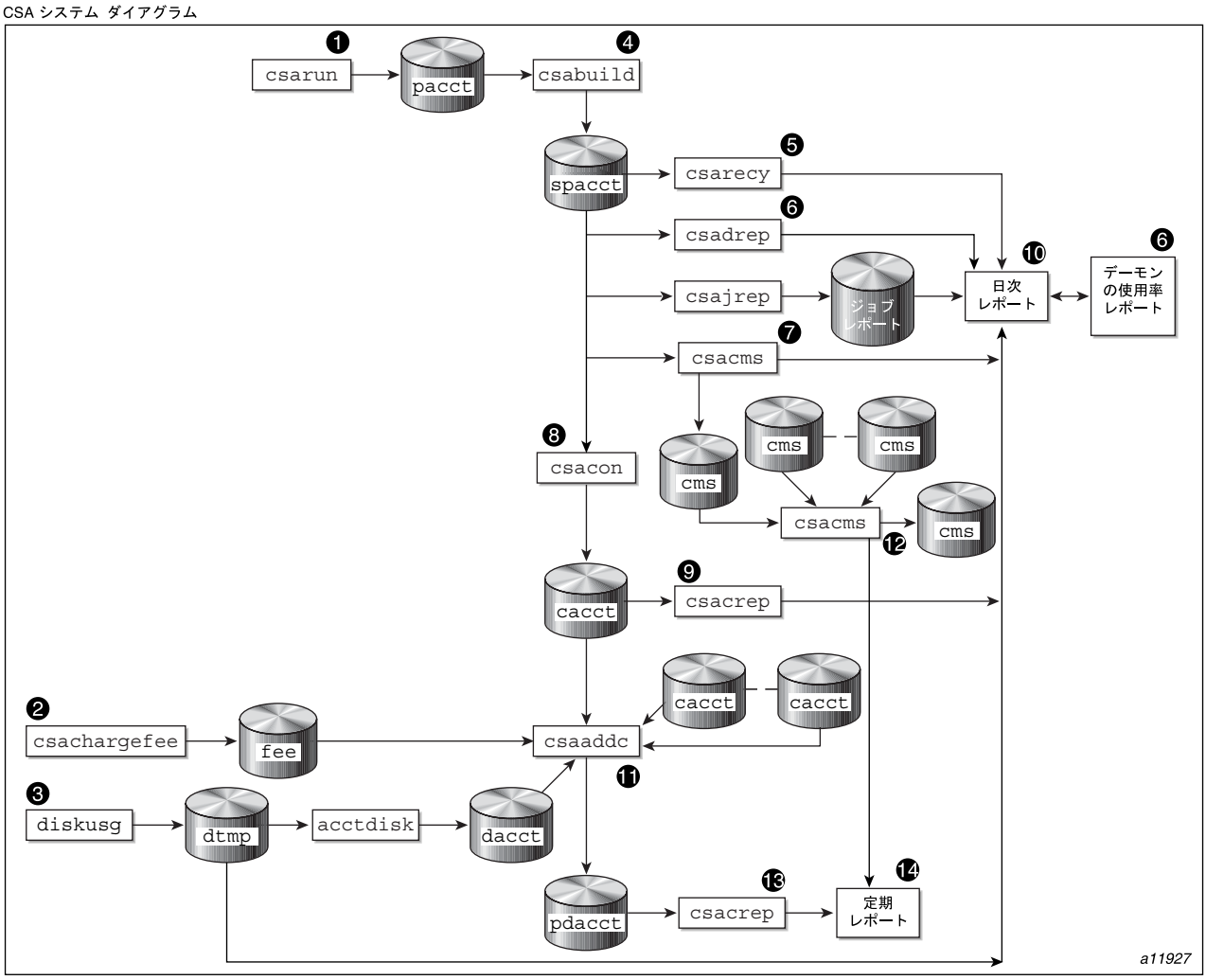

図5-2 CSA のデータ処理

1. raw アカウンティング・ファイルを生成します。さまざまなデーモンやシステム・プロセスが raw `pacct` アカウンティング・ファイルに書込みを行います。

2. 課金ファイルを作成します。特定のユーザに対して料金を請求したいサイトでは、`csachargefee`(1m) コマンドを使用して課金できます。`csachargefee` コマンドは、`csaaddc`(1m) によって処理される課金ファイルを作成します。

3. ディスク使用量の統計情報を生成します。dodisk(1m) シェル・スクリプトを使用することによって、サイトのディスク使用量のスナップショットを取ることができます。dodisk は、使用量を動的に報告することは行わず、コマンドが実行された時点でのディスクの使用量のみを報告します。ディスク使用量が csaaddc によって処理されます。

4. アカウンティング・レコードをジョブ・レコードにまとめます。csabuild(1M) コマンドが、CSA の pacct ファイルからアカウンティング・レコードを読込んで、ジョブ ID および起動時刻別にジョブ・レコードにまとめます。また、これらのジョブ・レコードを sorted pacct ファイルに書込みます。この sorted pacct ファイルには、各ジョブで使用できるすべてのアカウンティング・データが含まれます。pacct ファイルの設定レコードは、各起動期間内のジョブ ID が 0 のジョブ・レコードに関連付けられます。sorted pacct ファイル内の情報は、レポートの生成または課金の目的でほかのコマンドによって使用されます。

5. 未完了のジョブに関する情報をリサイクルします。csarecy(1M) コマンドが、現在のアカウンティング期間の sorted pacct ファイルからジョブ情報を取得し、未完了のジョブのレコードを pacct0 ファイルに書込んで、次のアカウンティング期間用にリサイクルします。csabuild(1M) は、未完了のアカウンティング・ジョブ（EOJ (end-of-job) レコードのないジョブ）にマークを付けます。csarecy は、それらのレコードを sorted pacct ファイルから取出し、次回の期間のアカウンティング・ファイル・ディレクトリに格納します。この過程は、ジョブが完了するまで繰返されます。

   終了したジョブのデータが引続きリサイクルされる場合もあります。この処理は、アカウンティング・データが失われた場合に発生する可能性があります。データが永久にリサイクルされることを防ぐには、csarun を編集して、csabuild が -o *nday* オプション付きで実行されるようにします。このオプションによって、*nday* 日より古いジョブはすべて終了します。適切な日数を示す *nday* 値を指定します（詳細については、csabuild のマン・ページおよび 95 ページの「データのリサイクル」を参照してください）。

6. デーモンの使用状況レポートを生成します。このレポートは、日別レポートに追加されます。csadrep(1m) が、NQS、ワークロード管理、およびテープのデーモンの使用状況を報告します。入力は、csabuild(1M) によって作成された sorted pacct ファイル、または csadrep に -o オプションを指定して作成されたバイナリ・ファイルから取得されます。files オペランドでバイナリ・ファイルを指定します。

7. プロセスごとのアカウンティング・レコードから、コマンドの使用状況を集計します。csacms(1m) コマンドが、sorted pacct ファイルを読込みます。同じ名前のコマンドを実行したプロセスのすべてのレコードを追加し、ソートした上で、cms 形式を使用して var/adm/acct/sum/csa/cms.*MMDDhhmm* に書込みます。csacms(1m) コマンドは、ASCII ファイルも作成できます。

8. sorted pacct ファイルのレコードを圧縮します。csacon(1M) コマンドが、sorted pacct ファイルのレコードを圧縮し、統合レコードを cacct 形式で var/adm/acct/sum/csa/cacct.*MMDDhhmm* に書込みます。

9. 統合データに基づいて、アカウンティング・レポートを生成します。csacrep(1m) コマンドが、csacon(1M) コマンドの出力など、cacct 形式のデータからレポートを生成します。レポートの形

式は、/etc/csa.conf ファイル内の CSACREP の値によって決まります。この値を変更しないかぎり、CPU 時間、KCORE 合計時間（分）、KVIRTUAL 合計時間（分）、ブロック I/O 待ち時間、および raw I/O 待ち時間が報告されます。レポートは、最初にユーザ ID、次に 2 番目のキーであるプロジェクト ID 別にソートされ、ヘッダが出力されます。

10. 日別アカウンティング・レポートを作成します。日別アカウンティング・レポートには、以下の項目が含まれます。

    - 統合情報レポート（手順 11）
    - 未完了のリサイクル・ジョブ（手順 5）
    - ディスク使用状況レポート（手順 3）
    - 日別コマンド集計（手順 7）
    - 最終ログイン情報
    - デーモン使用状況レポート（手順 6）

11. cacct レコードを結合します。csaaddc(1M) コマンドが、指定された統合オプションを使用して cacct レコードを結合し、統合レコードを cacct 形式で作成します。

12. プロセスごとのアカウンティング・レコードから、コマンドの使用状況を集計します。csacms(1m) コマンドが、手順 7 で作成された cms ファイルを読込みます。ASCII ファイルとバイナリ・ファイルの両方が作成されます。

13. 統合アカウンティング・レポートを生成します。csacrep(1m) を使用して、定期アカウンティング・ファイルに基づいてレポートが作成されます。

14. 定期アカウンティング・レポートのレイアウトを以下に示します。

    - 統合情報レポート
    - コマンド集計レポート

手順 4～11 は、各アカウンティング期間中に csarun(1m) によって実行されます。定期（月別）アカウンティング（手順 12～14）は、csaperiod(1m) コマンドによって起動されます。日別アカウンティング、定期アカウンティング、および課金とディスク使用量に関するレポートの生成は（手順 2～3）、cron(1m) を使用して、定期的に実行されるようスケジュールできます。詳細については、83 ページの「CSA の設定」を参照してください。

データのリサイクル

システム管理者は、正確なアカウンティング・レポートが生成されるよう、リサイクル・データを正しく保守する必要があります。以下の節では、データのリサイクルと、管理者による不要なリサイクル・アカウンティング・データのパージ(除去)方法について説明します。

データのリサイクルによって、CSA は、複数のアカウンティング期間にまたがってアクティブなジョブに対して正しく課金できます。csarun は、デフォルトで、現在のアカウンティング期間中に終了したジョブについてのみデータを報告します。データのリサイクルによって、CSA は、アクティブなジョブのデータをジョブが終了するまで保持します。

sorted pacct ファイルでは、ジョブがアクティブであるか終了しているかどうかを示すフラグを csabuild が各ジョブに付けます。csarecy は、sorted pacct ファイルを読込み、アクティブなジョブ用のデータをリサイクルします。csacon は、終了したジョブのデータを統合します。このデータは、後で csaperiod によって使用されます。csabuild、csarecy、および csacon は、すべて csarun によって呼出されます。

csarun は、リサイクル・データを /var/adm/acct/day/pacct0 ファイルに格納します。

通常は、管理者がリサイクル・アカウンティング・データを手動でパージする必要はありません。しかし、アカウンティング・データが失われた場合に限って手動でパージする必要があります。データが失われると、ジョブは永久にリサイクルされ、貴重な CPU サイクルとディスク領域が浪費される場合があります。

ジョブの終了方法

対話型のジョブ、cron ジョブ、および at ジョブは、ジョブ内の最後のプロセスが終了したときに終了します。通常、終了する最後のプロセスはログイン・シェルです。ジョブが終了すると、カーネルは EOJ (end-of-job) レコードを pacct ファイルに書込みます。

NQS デーモンまたはワークロード管理デーモンが NQS 要求またはワークロード管理要求の出力を送信すると、その要求は終了します。次に、デーモンは、NQS の NQ_DISP レコード・タイプまたはワークロード管理の WM_TERM レコード・タイプを pacct アカウンティング・ファイルに書込みます。カーネルは、EOJ レコードを pacct ファイルに書込みます。

対話型のジョブとは異なり、NQS 要求またはワークロード管理要求には、複数の EOJ レコードが関連付けられる場合があります。要求の EOJ レコードに加え、パイプ・クライアント(NQS のみ)、ネット・クライアント、および要求のチェックポイント済み部分の EOJ レコードもあります。パイプ・クライアントとネット・クライアントは、要求に代わって、NQS またはワークロード管理の処理を実行します。LSF (Load Sharing Faciliry) システムは、現在ネット・クライアントをサポートしていません。

csabuild コマンドは、以下のいずれかの条件が満たされる場合、sorted pacct ファイル内のジョブに終了のフラグを付けます。

- ジョブが対話型、cron、または at で、ジョブの EOJ レコードが pacct ファイルにある場合

- ジョブがNQS要求で、要求のEOJレコードとNQ_DISPレコード・タイプがpacctファイルにある場合
- ジョブがワークロード管理要求で、要求のEOJレコードとWM_TERMレコード・タイプがpacctファイルにある場合
- ジョブが対話型、cron、またはatで、システムのクラッシュ時にアクティブであった場合
- ジョブが、96ページの「リサイクル・データの削除方法」で説明されているいずれかの方法を使用して、管理者によって手動で終了された場合

リサイクル・セッションの検査が必要な理由

不要なデータをリサイクルすると、大量のディスク領域とCPU時間が消費される可能性があります。sorted pacctファイルとリサイクル・データは、/var/adm/acct/dayを含むファイルシステムで大量のディスク領域を占有します。データをアーカイブするサイトでは、オフライン・メディアの必要量も増加します。また、csarunのデータの再検査とリサイクルにCPUサイクルが浪費されます。このような理由から、ディスク領域とCPUサイクルを節約するために、不要なリサイクル・データをアカウンティング・システムからパージする必要があります。

以下のような状況では、終了したジョブがCSAで誤ってリサイクルされている可能性があります。

- カーネルまたはデーモンのアカウンティングが無効になっている場合

  カーネルまたはcsackpacct(1m)コマンドは、/var/adm/acct/dayを含むファイルシステムに十分な空き領域がない場合、アカウンティングを無効にすることがあります。

- アカウンティング・ファイルが破損している場合。アカウンティング・データは、システムまたはディスクのクラッシュ時に失われたり、破損したりすることがあります。
- リサイクル・データが以前のアカウンティング期間で誤って削除された場合

リサイクル・データの削除方法

リサイクル・データの削除を決める前に、98ページの「リサイクル・データを削除することによる悪影響」で説明されているとおり、削除による影響について理解しておく必要があります。データの削除は課金に影響し、またcsaperiodによって使用される統合データ・ファイルの内容が変更される可能性もあります。

リサイクル・データは、以下の方法を使用してCSAから削除できます。

- csarecy -Aコマンドを対話的に実行します。管理者は、csarecyに-Aオプションを指定して実行することによって、リサイクル対象のアクティブなジョブを選択できます。このように終了したジョブ内で使用されたリソースについては、ユーザは課金されません。また、削除されたデータは、統合データ・ファイルにも含まれません。

以下の例では、csarecy -A の実行方法の 1 つを示します(このコマンドは、2 つのアカウンティング・レポートと 2 つの統合ファイルを生成します)。

1. 定期的にスケジュールされた時刻に csarun を実行します。
2. /usr/lib/acct/csarun のコピーを編集します。csarecy を呼出す行で -r オプションを -A に変更します。また、標準出力は ${SUM_DIR}/recyrpt にリダイレクトしないでください。結果は以下のようになります。

```
csarecy -A -s ${SPACCT} -P ${WTIME_DIR}/Rpacct \ 2> ${NITE_DIR}/Erec.${DTIME}
```

-A オプションと -r オプションはどちらも出力を stdout に書込むため、-r オプションは呼出されず、stdout はファイルにリダイレクトされません。その結果、リサイクル・ジョブ・レポートは生成されません。

3. jstat コマンドを以下のように実行して、現在アクティブなジョブのリストを表示します。

```
jstat -a > jstat.out
```

4. qstat コマンドを実行して、NQS 要求のリストを表示します。qstat コマンドは、現在実行されていない要求があるかどうかを確認します。これには、チェックポイントに入っている要求、保留されている要求、キューに入っている要求、または待機している要求が含まれます。

すべての NQS 要求をリストするには、NQS マネージャまたは NQS オペレータの特権があるログインを使用して、以下のように qstat コマンドを実行します。

```
qstat -a > qstat.out
```

5. csarun の変更バージョンを対話的に実行します。最初の手順が完了した直後に、変更された csarun を実行する場合、存在するデータは多くないので、データはほとんど失われません。

個々のアクティブなジョブについて、csarecy で、ジョブを保存するかどうかの確認が求められます。第 3、第 4 の手順で見つかった、アクティブで実行されていない NQS ジョブを保存します。その他のすべてのジョブは、削除の対象となります。

- csabuild に -o *ndays* オプションを指定して実行します。これによって、指定した日数よりも古いすべてのアクティブなジョブが終了されます。これらの終了したジョブのリソース使用状況は csarun によって報告され、ユーザはジョブに対して課金されます。統合データ・ファイルには、このリソース使用状況に関する情報も含まれます。

  csabuild に -o オプションを指定して実行するには、/usr/lib/acct/csarun のコピーを編集します。csabuild を呼出す行に -o *ndays* オプションを追加します。*ndays* に、サイトにとって適切な値を指定します。

  ndays に不適切な値を指定した場合、現在アクティブなジョブのリサイクル・データが削除されてしまいます。

- csarun に -A オプションを指定して実行します。アクティブなジョブと終了したジョブのリソース使用状況が報告されるので、ユーザはリサイクル・セッションに対して課金されます。このデータは、統合データ・ファイルにも含まれます。

  アクティブなジョブのデータは、現在アクティブなジョブのデータも含め、リサイクルされません。リサイクル・データ・ファイルは、/var/adm/acct/day ディレクトリ内に生成されません。

- /var/adm/acct/day ディレクトリからリサイクル・データ・ファイルを削除します。以下のコマンドを実行することによって、終了したジョブとアクティブなジョブを含むすべてのリサイクル・ジョブのデータを削除できます。

  ```
  rm /var/adm/acct/day/pacct0
  ```

  次に csarun が実行されると、リサイクル・ジョブのデータは見つかりません。そのため、ユーザはリサイクル・ジョブ内で使用されたリソースに対して課金されず、このデータは統合データ・ファイルに含まれません。csarun は、現在アクティブなジョブのデータをリサイクルします。

リサイクル・データを削除することによる悪影響

CSA は、必要なアカウンティング情報がすべて利用できること、つまり、カーネルとデーモンのアカウンティングが有効になっており、リサイクル・データが誤って削除されていないことを想定します。一部のデータが利用できない場合、CSA から誤った課金情報が出力されることがあります。サイトからデータを削除する前に、以下の点に注意する必要があります。

- 終了したリサイクル・ジョブに対して、ユーザは課金される場合とされない場合があります。管理者は、上記の終了方法のうち、どれを使用すると終了したリサイクル・ジョブに対してユーザが課金されるのかを理解する必要があります。終了したジョブに対してユーザが課金されるのが正当であるかどうかは、サイトによって異なります。

  ユーザが課金される方法の場合、csarun と csaperiod の両方でリソース使用状況が報告されます。

- 終了したリサイクル・ジョブを再構築するのは不可能である場合があります。リサイクル・ジョブが管理者によって終了されても、実際には後のアカウンティング期間で終了した場合、そのジョブに関する情報は失われます。リソースの課金についてユーザから問合わせがあった場合、管理者がそのジョブに関するすべてのアカウンティング情報を正しく再構成することは非常に困難であるか、不可能です。

- 手動で終了したリサイクル・ジョブは、以降の課金期間で不適切に課金される場合があります。ジョブの最初の部分のアカウンティング・データが削除された場合、CSA では、そのジョブの残りの部分が正しく識別されない可能性があります。また、NQS 要求やワークロード管理要求が対話型のジョブであるとフラグ付けされたり、間違ったキュー・レートで課金されたりするというエラーが発生する場合があります。この問題については、99 ページの「NQS要求またはワークロード管理要求とリサイクル・データ」で詳しく説明します。

- CSA プログラムでデータの不整合が検出される場合があります。アカウンティング・データが失われた場合、CSA プログラムでは、エラーが検出され、異常終了する場合があります。

以下の表に、96 ページの「リサイクル・データの削除方法」で説明した方法を使用したときの影響をまとめます。

表5-1 リサイクル・データを削除することによる影響

方法	低く課金されるかどうか	不正確に課金される可能性	統合データ・ファイル
`csarecy -A`	される。ユーザは、`csarecy -A` によって終了されたジョブの部分については課金されない。	ある。手動で終了したリサイクル・ジョブは、以降の課金期間で不適切に課金される場合がある。	`csarecy -A` によって終了されたジョブのデータは含まれない。
`csabuild -o`	されない。ユーザは、`csabuild -o` によって終了されたジョブの部分について課金される。	ある。手動で終了したリサイクル・ジョブは、以降の課金期間で不適切に課金される場合がある。	`csabuild -o` によって終了されたジョブのデータを含む。
`csarun -A`	されない。アクティブなジョブおよびリサイクル・ジョブはすべて課金される。	ある。データがリサイクルされないので、最終的に終了するアクティブなジョブおよびリサイクル・ジョブは、すべて以降の課金期間で不適切に課金される場合がある。	すべてのアクティブなジョブおよびリサイクル・ジョブのデータを含む。
`rm`	される。すべてのユーザは、リサイクルされたジョブの部分について課金されない。	ある。最終的に終了するすべてのリサイクル・ジョブは、以降の課金期間で不適切に課金される場合がある。	リサイクル・ジョブのデータは含まれない。

統合データ・ファイルには、デフォルトで、終了したジョブのデータのみが含まれます。リサイクル・ジョブを手動で終了することによって、一部のリサイクル・データが統合ファイルに含まれる可能性もあります。

NQS要求またはワークロード管理要求とリサイクル・データ

CSA ですべての NQS 要求またはワークロード管理要求を認識するには、データが正しくリサイクルされる必要があります。管理者が NQS 要求またはワークロード管理要求のリサイクル・データを手動でパージすると、以下のようなエラーが発生する場合があります。

- CSA がジョブに対する NQS またはワークロード管理のフラグ付けに失敗する。これによって、要求は、NQS またはワークロード管理のキュー・レートではなく、標準のレートで課金されます(104ページの「NQS の SBU」または104ページの「ワークロード管理の SBU」を参照してください)。

- 要求が間違ったキュー・レートで課金される。
- 間違ったキュー待ち時間が要求に関連付けられる。

これらのエラーが発生するのは、NQS またはワークロード管理の情報が管理者によってパージされたためです。NQS デーモンまたはワークロード管理デーモンによって書込まれる NQS またはワークロード管理のアカウンティング・レコードはあまり多くなく、CSA が NQS 要求またはワークロード管理要求に対して正しく課金するには、すべてのレコードが必要です。

NQS またはワークロード管理のアカウンティング・レコードは、以下の状況でのみ書込まれます。

- NQS デーモンまたはワークロード管理デーモンが要求を受取ったとき
- 要求がキューにルーティングされたとき（NQS のみ）
- 要求が実行されたとき。これには、要求を初めて実行する場合、再開する場合、および再実行する場合が含まれます。
- 要求が終了したとき。NQS 要求は、完了、キューへの再指定、プリエンプト、保留、または再実行されたときに終了します。ワークロード管理要求は、完了、キューへの再指定、保留、再実行、または移行されたときに終了します。
- 出力が返されたとき

数日間に渡って実行される要求の場合、NQS またはワークロード管理のデータが書込まれない日もあります。そのため、アカウンティング・データをリサイクルすることは非常に重要です。サイトの管理者がリサイクル・ジョブを手動で終了する場合は、存在しない NQS 要求またはワークロード管理要求のみを終了するように注意する必要があります。

CSA のカスタマイズ

この節では、CSA の以下の操作について説明します。

- SBU の設定
- デーモンのアカウンティングの設定
- ユーザ出口の設定
- ユーザ出口の記述
- NQS またはワークロード管理の終了ステータスに基づく、NQS ジョブまたはワークロード管理ジョブに対する課金の変更
- CSA シェル・スクリプトのカスタマイズ

- cron(1m) ではなく、at(1) を使用した、csarun の定期実行
- スーパー・ユーザのパーミッションのないユーザによる CSA の実行許可
- 代替設定ファイルの使用

システム課金単位 (SBU)

システム課金単位 (SBU: System Billing Unit) は、マシン・リソースの使用量を表す測定単位です。各アカウンティング・レコードのそれぞれのフィールドに関連付けられた重み係数を変更することで、サイトにとって適切な SBU 値を設定できます。SBU は、アカウンティング設定ファイル /etc/csa.conf で定義します。デフォルトでは、すべての SBU が 0.0 に設定されています。

時間帯によって、プライム時間と非プライム時間を指定できます(時間帯は /usr/lib/acct/holidays で指定します)。

以下に、プライム時間と非プライム時間の算出例を示します。

ユーザが 10 秒の CPU 時間 (CPU time) を使用し、100 秒のプライム wall-clock 時間 (prime time) だけ実行して、100 秒の非プライム wall-clock 時間 (nonprime time) だけ停止するとします。この場合、経過時間 (ellapsed time) は 200 秒 (100+100) です。以下のような式が成立します。

```
prime = prime time / elapsed time
nonprime = nonprime time / elapsed time
cputime[PRIME] = prime * CPU time
cputime[NONPRIME] = nonprime * CPU time
```

以下のような結果になります。

```
cputime[PRIME] == 5 seconds
cputime[NONPRIME] == 5 seconds
```

CSA では、sorted pacct ファイルが csabuild によって作成されると、SBU 値がこのファイル内の各レコードに関連付けられます。SBU 値の最終的な集計は、cacct レコード・ファイルの作成時に、csacon によって実行されます。

以下に、サイトにおいて異なる NQS キューまたはワークロード管理キューに対して別々のレートで課金する方法の例を示します。

```
Total SBU = (NQS queue SBU value) * (sum of all process record SBUs
     + sum of all tape record SBUs)
```

または

```
Total SBU = (Workload management queue SBU value) * (sum of
all process record SBUs
     + sum of all tape record SBUs)
```

プロセスの SBU

プロセス・データの SBU は、プライム値と非プライム値に分けられます。プライムと非プライムの使用量は、経過時間の比率に応じて計算されます。プライム時間と非プライム時間を区別しない場合は、非プライム時間の SBU とプライム時間の SBU を同じ値に設定します。プライム時間は /usr/lib/acct/holidays で定義します。デフォルトでは、土曜日と日曜日が非プライム時間になっています。

以下に、プライム時間におけるプロセス SBU の重み係数のリストを示します。非プライム時間における SBU の重み係数の意味と単位も、このリストに示されているものと同様です。SBU の重み係数は、/etc/csa.conf で定義されます。

値	説明
P_BASIC	プライム時間の重み係数。P_BASIC は、プライム時間の SBU 値の合計で乗算され、プロセス・レコードの最終的な SBU 係数が求められます。
P_TIME	一般時間の重み係数。P_TIME は、時間の SBU（P_STIME、P_UTIME、P_QTIME、P_BWTIME、および P_RWTIME で構成される）で乗算され、プロセス・レコードの SBU 値に対する時間の寄与が求められます。
P_STIME	システム CPU 時間の重み係数。この重み係数に使用される単位は、1 秒あたりの課金単位です。P_STIME は、システム CPU 時間で乗算されます。
P_UTIME	ユーザ CPU 時間の重み係数。この重み係数に使用される単位は、1 秒あたりの課金単位です。P_UTIME は、ユーザ CPU 時間で乗算されます。
P_QTIME	キュー実行待ち時間の重み係数。この重み係数に使用される単位は、1 秒あたりの課金単位です。P_QTIME は、キュー実行待ち時間で乗算されます。
P_BWTIME	ブロック I/O 待ち時間の重み係数。この重み係数に使用される単位は、1 秒あたりの課金単位です。P_BWTIME は、ブロック I/O 待ち時間で乗算されます。
P_RWTIME	raw I/O 待ち時間の重み係数。この重み係数に使用される単位は、1 秒あたりの課金単位です。P_RWTIME は、raw I/O 待ち時間で乗算されます。

P_MEM	一般メモリ積算の重み係数。P_MEM は、メモリの SBU（P_XMEM と P_VMEM で構成される）で乗算され、プロセス・レコードの SBU 値に対するメモリの寄与が求められます。
P_XMEM	CPU 時間コア物理メモリ積算の重み係数。この重み係数に使用される単位は、1 Mバイト-分あたりの課金単位です。P_XMEM は、コア・メモリ積算で乗算されます。
P_VMEM	CPU 時間仮想物理メモリ積算の重み係数。この重み係数に使用される単位は、1 Mバイト-分あたりの課金単位です。P_VMEM は、仮想メモリ積算で乗算されます。
P_IO	一般 I/O の重み係数。P_IO は、I/O の SBU（P_BIO、P_CIO、および P_LIO で構成される）で乗算され、プロセス・レコードの SBU 値に対する I/O の寄与が求められます。
P_BIO	転送ブロックの重み係数。この重み係数に使用される単位は、1 転送ブロックあたりの課金単位です。P_BIO は、I/O 転送ブロック数で乗算されます。
P_CIO	文字転送の重み係数。この重み係数に使用される単位は、1 転送文字あたりの課金単位です。P_CIO は、I/O 転送文字数で乗算されます。
P_LIO	論理 I/O 要求の重み係数。この重み係数に使用される単位は、1 論理 I/O 要求あたりの課金単位です。P_LIO は、論理 I/O 要求数で乗算されます。論理 I/O 要求の数は、read システム・コールと write システム・コールの総数です。

全プロセス・レコードの SBU の計算式は、以下のとおりです。

```
PSBU = (P_TIME * (P_STIME * stime + P_UTIME * utime + P_QTIME * qwtime +
P_BWTIME * bwtime + P_RWTIME * rwtime)) + (P_MEM * (P_XMEM * coremem + P_VMEM
* virtmem)) + (P_IO * (P_BIO * bio + P_CIO * cio + P_LIO * lio));

NSBU = (NP_TIME * (NP_STIME * stime + NP_UTIME * utime + NP_QTIME * qwtime +
NP_BWTIME * bwtime + NP_RWTIME * rwtime)) + (NP_MEM * (NP_XMEM * coremem +
NP_VMEM * virtmem)) + (NP_IO * (NP_BIO * bio + NP_CIO * cio + NP_LIO * lio));

SBU = P_BASIC * PSBU + NP_BASIC * NSBU;
```

この式の変数については、以下で説明します。

変数	説明
stime	システム CPU 時間（秒単位）
utime	ユーザ CPU 時間（秒単位）

bwtime	ブロック I/O 待ち時間（秒単位）
rwtime	raw I/O 待ち時間（秒単位）
coremem	コア（物理）メモリ積算（Mバイト-分単位）
virtmem	仮想メモリ積算（Mバイト-分単位）
bio	ブロック転送データの数
cio	文字転送データの数
lio	論理 I/O 要求の数

NQS の SBU

/etc/csa.conf ファイルには、NQS の SBU に関連する設定可能パラメータが含まれています。

NQS_NUM_QUEUES パラメータは、SBU を設定するキューの数を設定します（1 以上の値に設定する必要があります）。設定ファイルの各 NQS_QUEUE *x* 変数には、キュー名と SBU のペアが関連付けられています（キューと SBU のペアの総数は NQS_NUM_QUEUES の値と等しくなければなりません）。キューと SBU のペアによって、そのキューの重み係数が定義されます。SBU 値が 1.0 未満の場合、関連するキュー内のジョブを実行することで課金が割引になります。値が 1.0 の場合、ジョブは NQS 以外のジョブと同レートで課金されます。SBU が 0.0 の場合、関連するキュー内で実行中のジョブには課金されません。設定ファイル内で指定されていないキューの SBU は、自動的に 1.0 に設定されます。

NQS_NUM_MACHINES パラメータは、SBU を設定する発生元マシンの数を設定します（1 以上の値に設定する必要があります）。設定ファイルの各 NQS_MACHINE *x* 変数には、発生元マシンと SBU のペアが関連付けられています（マシンと SBU のペアの総数は NQS_NUM_MACHINES の値と等しくなければなりません）。/etc/csa.conf 内で指定されていない発生元マシンの SBU は、自動的に 1.0 に設定されます。

キューとマシンの SBU を乗算すると、NQS 乗数が求められます。SBU が 1.0 未満の場合、これらのキュー内のジョブまたはこれらのマシンからのジョブを実行することで課金が割引になります。SBU が 1.0 の場合、キュー内のジョブまたは関連ホストからのジョブは通常どおりに課金されます。

ワークロード管理の SBU

/etc/csa.conf ファイルには、ワークロード管理の SBU に関連する設定可能パラメータが含まれています。

WKMG_NUM_QUEUES パラメータは、SBU を設定するキューの数を設定します（1 以上の値に設定する必要があります）。設定ファイルの各 WKMG_QUEUE *x* 変数には、キュー名と SBU のペアが関連付けられています（キューと SBU のペアの総数は WKMG_NUM_QUEUES の値と等しくなければなりません）。キューと SBU のペアによって、そのキューの重み係数が定義されます。SBU 値が 1.0 未満の場合、関連するキュー内のジョブを実行することで課金が割引になります。値が 1.0 の場合、ジョブはワークロード管理

以外のジョブと同レートで課金されます。SBU が 0.0 の場合、関連するキュー内で実行中のジョブには課金されません。設定ファイル内で指定されていないキューの SBU は、自動的に 1.0 に設定されます。

WKMG_NUM_MACHINES パラメータは、SBU を設定する発生元マシンの数を設定します（1 以上の値に設定する必要があります）。設定ファイルの各 WKMG_MACHINE *x* 変数には、発生元マシンと SBU のペアが関連付けられています（マシンと SBU のペアの総数は WKMG_NUM_MACHINES の値と等しくなければなりません）。/etc/csa.conf 内で指定されていない発生元マシンの SBU は、自動的に 1.0 に設定されます。

テープの SBU

テープ・デバイスの各グループには、重み係数のセットがあります。デフォルトのグループは、tape と cart の 2 つだけです。/etc/csa.conf 内の TAPE_SBU *i* パラメータは、各グループの重み係数を定義します。以下の項目に関連付けられた SBU があります。

- マウント回数
- デバイス予約時間（秒）
- 読込みバイト数
- 書込みバイト数

SBU 設定の例

以下に、SBU システムの設定方法を示します。この例では、プロセス・レコードに限定して説明します。

すべての時間はプライム時間と見なします。したがって、非プライム時間の SBU は、対応するプライム時間の SBU と同じ値に設定する必要があります。

ユーザは、ユーザ CPU 時間 1 時間につき 10 ドル課金されます。これは、3600 秒につき 10 ドルの計算になります。つまり、1 秒につき 0.002777777777777 ドル (P_UTIME) です。

したがって、料金は以下のようになります（非プライム時間の SBU は、対応するプライム時間の SBU と同じ値に設定します）。

重み係数	値
P_BASIC	1.0
P_TIME	1.0
P_STIME	0.0
P_UTIME	0.002777777777777
P_QTIME	0.0
P_BWTIME	0.0

P_RWTIME	0.0
P_MEM	0.0
P_XMEM	0.0
P_VMEM	0.0
P_IO	0.0
P_BIO	0.0
P_CIO	0.0
P_LIO	0.0

デーモンのアカウンティング

デーモンのアカウンティング情報は、NQS、ワークロード管理、およびオンライン・テープの各デーモンから取得できます。データは、/var/adm/acct/day ディレクトリの pacct ファイルに書込まれます。

多くの場合、デーモンのアカウンティングは、CSA サブシステムとデーモンの両方で有効にする必要があります。83 ページの「CSA の設定」では、システム起動時にデーモンのアカウンティングを有効にする方法について説明します。システムの起動後にデーモンのアカウンティングを有効にすることもできます。

指定したデーモンのアカウンティングを有効にするには、csaswitch コマンドを使用します。たとえば、テープのアカウンティングを開始するには、以下のコマンドを実行します。

```
/usr/lib/acct/csaswitch -c on -n tape
```

NQS、ワークロード管理、およびオンライン・テープのデーモンでもアカウンティングを有効にする必要があります。NQS のアカウンティングを有効にするには、qmgr set accounting on コマンドを使用します。テープ・デーモンのアカウンティングは、tmdaemon(1m) に -c オプションを指定して実行すると有効になります。ワークロード管理のアカウンティングを有効にする方法については、該当するワークロード管理ガイドを参照してください。

メモ: LSF (Load Sharing Facility) システムを実行している場合、ワークロード管理のアカウンティングを有効にするには、`lsf.conf` ファイル内で 2 つの LSF 設定変数を以下のように設定する必要があります。

```
LSF_ENABLE_CSA=y
LSF_ULDB_DOMAIN = <ULDB_domain_name>
```

`LSF_ENABLE_CSA` を `lsf.conf` ファイル内で定義すると、LSF では、LSF バッチ・ジョブ・イベントを `pacct` ファイルに書込み、CSA を通して処理を行います。LSF のジョブ・アカウンティングに対しては、各 LSF ジョブの開始と終了でレコードが `pacct` に書込まれます。

LSF の ULDB ドメインを `lsf.conf` ファイル内で定義した場合、LSF では、IRIX ジョブを作成した後、設定されたリソース制限をそのジョブに適用します。`lsb.queues` 内で、あるいはジョブの実行時に定義された LSF リソース制限は、ULDB で定義された IRIX ジョブ制限をオーバーライドします。

LSF (Load Sharing Facility) システムおよびワークロード管理のアカウンティングについての詳細は、該当する LSF のドキュメントを参照してください。

デーモンのアカウンティングは、システムのシャットダウン時に無効になります（83 ページの「CSA の設定」を参照してください）。また、`csaswitch` コマンドで `off` オペランドを指定することによって、いつでも無効にできます。たとえば、NQS のアカウンティングを無効にするには、以下のコマンドを実行します。

```
/usr/lib/acct/csaswitch -c off -n nqs
```

`csaswitch` を使用したこれらの動的な変更は、システムの再起動時に保持されません。

ユーザ出口の設定

CSA では、以下のユーザ出口を使用できます。これらの出口は、下記に示す `csarun` 状態から呼出すことができます。

`csarun` 状態	ユーザ出口
ARCHIVE1	/usr/lib/acct/csa.archive1
ARCHIVE2	/usr/lib/acct/csa.archive2
FEF	/var/lib/acct/csa.fef
USEREXIT	/usr/lib/acct/csa.user

CSA では、以下のユーザ出口を使用できます。これらの出口は、下記に示す `csaperiod` 状態から呼出すことができます。

csaperiod 状態	ユーザ出口
USEREXIT	/usr/lib/acct/csa.puser

管理者は、これらのユーザ出口を利用して、日別アカウンティング時にサイト固有の追加処理を実行するスクリプトを作成することで、csarun プロシージャ(または csaperiod プロシージャ)を個々のサイトのニーズに合わせてカスタマイズできます(以下の説明は csaperiod にも当てはまります)。

csarun の実行時、ARCHIVE1、ARCHIVE2、FEF、および USEREXIT の各状態で、それぞれ上記の名前のシェル・スクリプトがチェックされます。

スクリプトが存在する場合は、そのスクリプトがシェルの .(ドット)コマンドで実行されます。スクリプトが存在しない場合、ユーザ出口は無視されます。(ドット)コマンドではコンパイルされたプログラムは実行できませんが、ユーザ出口スクリプトの中から実行できます。csarun の変数は、エクスポートされていなくても、ユーザ出口スクリプトで利用できます。csarun では、ユーザ出口から返されるステータスがチェックされ、0 以外の場合、csarun の実行は中断されます。

スーパー・ユーザのパーミッションのないユーザが CSA を実行する場合、ユーザ出口は、このユーザが読込み可能および実行可能である必要があります(113ページの「スーパー・ユーザ以外による CSA の実行許可」を参照してください)。

次に、ユーザ出口の例を示します。

```
rain1# cd /usr/lib/acct

rain1# cat csa.archive1

#!/bin/sh
mkdir -p /tmp/acct/pacct${DTIME}
cp ${WTIME_DIR}/${PACCT}* /tmp/acct/pacct${DTIME}


rain1# cat csa.archive2

#!/bin/sh
cp ${SPACCT} /tmp/acct

rain1# cat csa.fef

#!/bin/sh
mkdir -p /tmp/acct/jobs
/usr/lib/acct/csadrep -o /tmp/acct/jobs/dbin.${DTIME} -s ${SPACCT}
/usr/lib/acct/csadrep -n -V3 /tmp/acct/jobs/dbin.${DTIME}
```

ユーザ出口の記述

この節では、ユーザ出口の記述について説明します。最初に、日別アカウンティングの実行後に sorted pacct ファイルを保存するユーザ出口の例を示します。次に、ユーザ別ではなくプロジェクト別に日別レポートの情報を統合するユーザ出口の例を示します。

サンプル5-1 日別アカウンティングの実行時に sorted pacct ファイルを保存する

csarun(1M) スクリプトと csaperiod(1M) スクリプトでは、ユーザ出口スクリプト内で使用できるシェル変数を使用します。たとえば、sorted pacct ファイルは、日別アカウンティングの実行に成功すると削除されます。ただし、このファイルを保存したい場合は、sorted pacct ファイルの作成後に実行される任意のユーザ出口を使用することができます（csarun(1M) のマン・ページを参照してください）。次に、このファイルの保存だけを行う簡単なユーザ出口スクリプトを示します。

```
#! /bin/sh
echo "Copying spacct file to /tmp/spacct"
cp ${SPACCT} /tmp/spacct
```

サンプル5-2 ユーザ別ではなくプロジェクト別に統合された情報レポート

日別レポートから統合された情報のデフォルトの出力は、次のとおりです。

```
CONSOLIDATED INFORMATION REPORT BETWEEN 08/09 04:00   AND   08/09 14:48

PROJECT    USER    LOGIN    CPU-TIM  KCORE *  KVIRT *   IOWAIT [SECS]
 NAME       ID     NAME     [SECS]   CPU-MIN  CPU-MIN   BLOCK     RAW
======== ======== ======== ======== ======== ======== ======== ========
sysadm   0        root           30      536     1177       48        0
root     4        sys             0        5       11        0        0
csa      5        adm             5       24      194        1        0
root     1461     security        1        2       16        0        0
nqe      10320    user12          2        5       68        1        0
```

ユーザ別ではなくプロジェクト別に統合された日別レポートの情報を表示するには、以下のように csacon(1M) コマンドと csacrep(1M) コマンドにプロジェクト・オプションを指定して実行します。

```
/usr/lib/acct/csacon -Ap -s /tmp/spacct > /tmp/cacct_p
/usr/lib/acct/csacrep -hpcw < /tmp/cacct_p > /tmp/csacrep.out.p
```

出力は、以下のとおりです。

```
PROJECT    USER    LOGIN    CPU-TIM  KCORE *  KVIRT *   IOWAIT [SECS]
 NAME       ID     NAME     [SECS]   CPU-MIN  CPU-MIN   BLOCK     RAW
======== ======== ======== ======== ======== ======== ======== ========
root     Unknown  Unknown         1        8       28        0        0
```

```
sysadm    Unknown  Unknown         31       537      1187        49         0
csa       Unknown  Unknown          5        24       194         1         0
nqe       Unknown  Unknown          2         7        83         1         0
```

次の /usr/lib/acct/csa.user スクリプトの例では、上記の csacon(1M) コマンドと csacrep(1M) コマンドの例と同じ操作が実行され、日別レポート内にプロジェクト別に統合された情報レポートが含まれます。

```
#!/sbin/sh
#
csacon ${ALLJOBS} -p -s ${SPACCT} > ${SUM_DIR}/cacct_p.${DTIME} \
        2> ${NITE_DIR}/Econ.${DTIME}
if [ ${?} -ne 0 ]
then
        CSAERRMSG="REPORT - csacon errors \
                \n\tSee ${NITE_DIR}/Econ.${DTIME} and/or ${NITE_DIR}/fd2log"
        ERROR_EXIT
fi
chgrp ${CHGRP} ${SUM_DIR}/cacct_p.${DTIME}
#
csacrep -hpcw < ${SUM_DIR}/cacct_p.${DTIME} \
> ${SUM_DIR}/conrpt_p.${DTIME} 2> ${NITE_DIR}/Ecrpt_p.${DTIME}
if [ ${?} -ne 0 ]
then
        CSAERRMSG="REPORT - csacrep errors \
                \n\tSee ${NITE_DIR}/Ecrep_p.${DTIME} and/or ${NITE_DIR}/fd2log"
        ERROR_EXIT
fi
#
cd ${SUM_DIR}
echo "${RPTHDR}\n" > tmprprt
echo "Put some header message here\n"  >> tmprprt
cat conrpt_p.${DTIME} >> tmprprt
pr -h "${DAYHDR} ${SYSNAME} ${RELMSG}" tmprprt >> rprt.${DTIME}
#
```

新しいバイナリ・データ・ファイル（上記のユーザ出口の例の cacct_p）を定期レポートに使用する場合は、 /usr/lib/acct/csaperiod に対してユーザ出口を作成する必要があります。

NQS ジョブに対する課金

デフォルトで、SBU は、ジョブの NQS 終了コードに関係なく、すべての NQS ジョブについて計算されます。NQS 要求の一部に対して課金したくない場合は、/etc/csa.conf ファイル内の該当する

NQS_TERM_*xxxx* 変数（各終了コードに対応）を 0 に設定します。これによって、この部分の SBU は 0.0 に設定されます。デフォルトでは、要求のすべての部分が課金されます。

以下の表では、終了コードについて説明します。

コード	説明
NQS_TERM_EXIT	要求の実行が完了し、キューに入っていない状態になったときに生成されます。NQS シャットダウン時は、qsub で -nc（チェックポイントなし）と -nr（再実行なし）のオプションが指定された要求にも、NQS_TERM_EXIT レコードが書込まれます。また、qsub で -nr オプションが指定され、システム・クラッシュ時に実行中であった要求についても、このレコードが書込まれます。
NQS_TERM_REQUEUE	チェックポイントに入り、NQS シャットダウン時に再度キューに入れられる、実行中の要求について書込まれます。
NQS_TERM_PREEMPT	要求が qmgr preempt request コマンドによってプリエンプトされるときに書込まれます。
NQS_TERM_HOLD	qmgr hold request コマンドによってチェックポイントに入れられる要求について書込まれます。hold request コマンドは、デーモンのシャットダウン時に実行されるチェックポイントとは異なります。hold request の場合、ジョブは、qmgr release コマンドが実行されるまでスケジュールされません。
NQS_TERM_OPRERUN	要求が qmgr rerun request コマンドによって再実行されるときに書込まれます。 NQS シャットダウン時に、チェックポイントに入れることができず、qsub で -nr（再実行なし）オプションが指定されていないジョブに、このタイプの終了レコードが書込まれます。要求は、このステータスで再度キューに入れられます。
NQS_TERM_RERUN	要求がオペレータ以外による再実行要求である場合に書込まれます。

ワークロード管理ジョブに対する課金

デフォルトで、SBU は、ジョブのワークロード管理終了コードに関係なく、すべてのワークロード管理ジョブについて計算されます。ワークロード管理要求の一部に対して課金したくない場合は、/etc/csa.conf ファイル内の該当する *WKMG_TERM_xxxx* 変数（各終了コードに対応）を 0 に設定します。これによって、この部分の SBU は 0.0 に設定されます。デフォルトでは、要求のすべての部分が課金されます。

以下の表では、終了コードについて説明します。

コード	説明
WKMG_TERM_EXIT	要求の実行が完了し、キューに入っていない状態になったときに生成されます。
WKMG_TERM_REQUEUE	再度キューに入れられる要求について書込まれます。
WKMG_TERM_HOLD	チェックポイントに入り保留される要求について書込まれます。
WKMG_TERM_RERUN	要求が再実行される場合に書込まれます。
WKMG_TERM_MIGRATE	要求が移行される場合に書込まれます。

メモ: 終了コードに関する上記の説明は、非常に一般的です。このため、ワークロード管理者によっては、製品に合わせてこれらのコードの意味がカスタマイズされることもあります。LSF で現在使用される終了コードは、WKMG_TERM_EXIT のみです。

CSA のシェル・スクリプトとコマンドのカスタマイズ

必要に応じて、/etc/csa.conf 内の以下の変数を変更します。

変数	説明
MAIL_LIST	アカウンティング・シェル・スクリプト内で致命的なエラーが検出された場合のメールの送付先ユーザのリスト。デフォルトは、root と adm です。
WMAIL_LIST	クリーンアップ時にアカウンティング・スクリプトによって警告エラーが検出された場合のメールの送付先ユーザのリスト。デフォルトは、root と adm です。
MIN_BLKS	csarun または csaperiod を実行するために var/adm/acct ディレクトリが存在するファイルシステムで必要な未使用のブロックの最低数。デフォルトは、2000 個の空きブロックです。ブロック・サイズは 1024 バイトです。

at を使用した csarun の実行

cron の代わりに at コマンドを使用して、csarun を定期的に実行できます。csarun が cron を介して実行されるようスケジュールされている時間にシステムがダウンしていた場合、csarun は、次のスケジュール時間まで実行されません。一方、at ジョブは、スケジュールされた実行時間にシステムがダウンしていた場合、マシンの再起動時に実行されます。

at を使用して csarun を実行する方法は複数あります。たとえば、csarun を実行するための独立したスクリプトを記述して、指定の時刻にジョブを再実行できます。また、at による csarun の呼出しをユーザ出口スクリプト /usr/lib/acct/csa.user 内に記述できます。このスクリプトは、csarun の

USEREXIT セクションから実行されます。詳細については、107ページの「ユーザ出口の設定」を参照してください。

スーパー・ユーザ以外による CSA の実行許可

サイトによっては、スーパー・ユーザのパーミッションのないユーザに対して CSA アカウンティングの実行を許可したい場合があります。CSA は、adm グループに所属し、CAP_ACCT_MGT capability を持っているユーザが実行できます。プロセスの操作権に対する細かな調整を可能にする capability についての詳細は、capability(4) および capabilities(4) のマン・ページを参照してください。

以下に、スーパー・ユーザのパーミッションのないユーザによって CSA が毎日および定期的に自動実行されるよう設定する手順を説明します。この例では、スーパー・ユーザのパーミッションを持たないユーザとして adm を仮定します。

1. ユーザ adm がグループ adm のメンバーであり、CAP_ACCT_MGT capability を持っていることを確認します。
2. 以下のユーザ出口が存在する場合は、ユーザ adm によって読取り可能および実行可能であることを確認します。
   - /usr/lib/acct/csa.archive1
   - /usr/lib/acct/csa.archive2
   - /usr/lib/acct/csa.fef
   - /usr/lib/acct/csa.user
   - /usr/lib/acct/csa.puser
3. 83 ページの「CSA の設定」の手順 1～5 に従って、システム課金単位を設定し、システム起動時刻を記録して、システム・シャットダウンの前にアカウンティングを無効にします。
4. /var/spool/cron/crontabs/root に以下のようなエントリを追加して、cron によって dodisk(1m) が自動的に実行されるよう設定します。

```
0 2 * * 4 if /etc/chkconfig csaacct; then /usr/lib/acct/dodisk -c 2> /var/adm/acct/nite/csa/dk2log; fi
```

dodisk コマンドは root で実行する必要があります。その他のユーザには、/dev/dsk/* の読込みパーミッションがありません。dodisk(1M) コマンドについての詳細は、dodisk(1M) のマン・ページを参照してください。

5. /var/spool/cron/crontabs/adm に以下のようなエントリを追加して、ユーザ adm が cron を使用して日別アカウンティングを自動的に実行するよう設定します。

```
0 4 * * 1-6 su adm -C CAP_ACCT_MGT+pi -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/csarun 2> /var/adm/acct/nite/csa/fd2log; fi"
5 * * * 1-6 su adm -C CAP_ACCT_MGT+pi -c "if /etc/chkconfig csaacct;
then /usr/lib/acct/csackpacct; fi"
```

csarun コマンドは、dodisk が完了できる時間を考慮して実行する必要があります。csarun が実行される前に dodisk が完了しなかった場合、ディスク・アカウンティング情報が見つからないか、不完全になる可能性があります。

6. 月別アカウンティングを実行するには、/var/spool/cron/crontabs/adm に以下のようなエントリを追加します(このコマンドでは、/var/adm/acct/sum/csa 内で見つかったすべての統合データ・ファイルに関する月別レポートが生成され、それらのデータ・ファイルが削除されます)。

```
Change the crontab entry for #6 to the following:
  0 5 1 * * if /etc/chkconfig csaacct;
then /usr/lib/acct/csaperiod -r 2> /var/adm/acct/nite/csa/pd2log; fi
```

7. 83 ページの「CSA の設定」の説明に従って、休日ファイルを更新します。

メモ: 上記の cron エントリは、ユーザ adm のログイン・シェルが sh または ksh の場合にのみ機能します。

代替設定ファイルの使用

デフォルトでは、いずれかの CSA コマンドが実行されたときに /etc/csa.conf 設定ファイルが使用されます。シェル変数 CSACONFIG を別の設定ファイルに指定して、CSA コマンドを実行することもできます。

たとえば、csarun の実行中に設定ファイル /tmp/myconfig を使用するには、以下のコマンドを実行します。

```
CSACONFIG=/tmp/myconfig
/usr/lib/acct/csarun 2> /var/adm/acct/nite/fd2log
```

CSA レポート

CSA を使用して、アカウンティング・レポートを作成できます。レポートを使用すると、システム使用状況の追跡、パフォーマンスのモニタ、およびシステム上でのユーザ時間に対する課金に役立てることができます。

CSA の日別レポートは /var/adm/acct/sum/csa ディレクトリにあり、定期レポートは /var/adm/acct/fiscal/csa ディレクトリにあります。レポートを表示するには、レポート・ディレクトリ内の ASCII ファイル rprt.*MMDDhhmm* を開きます。

CSA レポートには、ほかのアカウンティング・レポートよりも詳細なデータが含まれます。CSA アカウンティングの場合、日別レポートは csarun コマンドによって生成されます。日別レポートには、以下の項目が含まれます。

- ディスク使用量の統計情報
- 未完了ジョブ情報
- コマンド集計データ
- 統合アカウンティング・レポート
- 最終ログイン情報
- デーモン使用状況レポート

定期レポートは、csaperiod コマンドによって生成されます。また、diskusg コマンドを使用してディスク使用状況レポートを作成することもできます。

CSA 日別レポート

この節では、以下のレポートについて説明します。

- 116ページの「統合情報レポート」
- 116ページの「未完了ジョブ情報レポート」
- 116ページの「ディスク使用状況レポート」
- 116ページの「コマンド集計レポート」
- 117ページの「最終ログイン・レポート」
- 117ページの「デーモン使用状況レポート」

統合情報レポート

統合情報レポートは、ユーザ ID およびプロジェクト ID 別に順にソートされます。以下の使用状況の値は、報告期間中に、指定されたユーザおよびプロジェクトの全プロセスで使用されるリソースの合計量です。

ヘッダ	説明
PROJECT NAME	リソースの使用状況情報に関連付けられたプロジェクト
USER ID	ユーザ識別子
LOGIN NAME	ユーザのログイン名
CPU_TIME	CPU 時間の累積(秒単位)
KCORE * CPU-MIN	CPU 時間 1 分あたりに使用されたコア(物理)メモリの累積(Kバイト単位)
KVIRT * CPU-MIN	CPU 時間 1 分あたりに使用された仮想メモリの累積(Kバイト単位)
IOWAIT BLOCK	ブロック I/O 待ち時間の累積(秒単位)
IOWAIT RAW	raw I/O 待ち時間の累積(秒単位)

未完了ジョブ情報レポート

未完了ジョブ情報レポートには、まだ終了しておらず、次のアカウンティング期間にリサイクルされるジョブに関する説明が記述されます。

ヘッダ	説明
JOB ID	ジョブ識別子
USERS	ジョブの所有者のログイン名
PROJECT ID	ジョブに関連付けられたプロジェクト識別子
STARTED	ジョブの開始時間

ディスク使用状況レポート

ディスク使用状況レポートには、消費されたディスク・リソースの量がログイン名別に記述されます。

このレポートには列ヘッダがありません。最初の列は、ユーザ識別子を示します。2 番目の列は、ユーザ識別子に関連付けられたログイン名を示します。3 番目の列は、このユーザによって使用されたディスク・ブロックの数を示します。

コマンド集計レポート

コマンド集計レポートには、報告期間中のコマンドの使用状況がまとめられます。使用状況の値は、指定されたコマンドのすべての呼出しで使用されたリソースの合計量です。一度しか実行されなかったコ

マンドは、「***other」エントリ内にまとめられます。日別レポートでは、最初の 44 個のコマンド・エントリのみが表示されます。定期レポートでは、すべてのコマンド・エントリが表示されます。

ヘッダ	説明
COMMAND NAME	コマンド（プログラム）の名前
NUMBER OF COMMANDS	コマンドの実行回数
TOTAL KCORE-MINUTES	CPU 時間 1 分あたりに使用されたコア（物理）メモリの合計（Kバイト単位）
TOTAL KVIRT-MINUTES	CPU 時間 1 分あたりに使用された仮想メモリの合計（Kバイト単位）
TOTAL CPU	使用された CPU 時間の合計（分単位）
TOTAL REAL	使用された実 (wall-clock) 時間の合計（分単位）
MEAN SIZE KCORE	使用されたコア（物理）メモリの平均量（Kバイト単位）
MEAN SIZE KVIRT	使用された仮想メモリの平均量（Kバイト単位）
MEAN CPU	使用された CPU 時間の平均（分単位）
HOG FACTOR	合計 CPU 時間を合計実時間（経過時間）で除算した値
K-CHARS READ	読込まれた文字の合計数（Kバイト単位）
K-CHARS WRITTEN	書込まれた文字の合計数（Kバイト単位）
BLOCKS READ	読込まれたブロックの合計数
BLOCKS WRITTEN	書込まれたブロックの合計数

最終ログイン・レポート

最終ログイン・レポートには、リストされた各ログイン・アカウントの最後のログイン日付が表示されます。

このレポートには列ヘッダがありません。最初の列は、最終ログイン日付を示します。2 番目の列は、ログイン・アカウント名を示します。

デーモン使用状況レポート

デーモン使用状況レポートには、NQS またはワークロード管理とテープのデーモンの使用状況が表示されます。このレポートは、報告期間中に NQS、ワークロード管理、またはテープのデーモン・アクティビティがあったかどうかによって、複数のレポートが生成されます。

ジョブ・タイプ・レポートには、NQS ジョブと対話型ジョブの使用数が表示されます。

ヘッダ	説明
Job Type	ジョブのタイプ（対話型、NQS、またはワークロード管理）

Total Job Count	ジョブ・タイプごとのジョブの数と割合
Tape Jobs	対話型と NQS またはワークロード管理のジョブに関連付けられたテープ・ジョブの数と割合

CPU 使用状況レポートには、NQS またはワークロード管理と対話型のジョブの、CPU 関連の使用状況が表示されます。

ヘッダ	説明
Job Type	ジョブのタイプ（対話型、NQS、またはワークロード管理）
Total CPU Time	使用された CPU 時間の合計（秒単位）と CPU 時間の割合
System CPU Time	CPU 時間のうち、システム CPU 時間に使用された時間と、その割合
User CPU Time	CPU 時間のうち、ユーザ CPU 時間に使用された時間と、その割合

テープ使用状況レポートには、NQS またはワークロード管理と対話型のジョブの、テープ・アクティビティ関連の使用状況が表示されます。

ヘッダ	説明
Job Type	ジョブのタイプ（対話型、NQS、またはワークロード管理）
Device Group	テープ・デバイスのグループ名
Rsv Time	テープ予約時間（秒単位）
Mounts	テープのマウント回数
KBytes Read	読込まれたテープの量（Kバイト単位）
KBytes Written	書込まれたテープの量（Kバイト単位）
User CPU	使用されたユーザ CPU 時間の合計（秒単位）
Sys CPU	使用されたシステム CPU 時間の合計（秒単位）

バッチ・キュー・レポートには、各 NQS キューまたはワークロード管理キューについて以下の情報が表示されます。

Queue Name	NQS キューまたはワークロード管理キューの名前
Number of Jobs	キューから開始されたジョブの数
CPU Time	キューのジョブによって使用されたシステム CPU 時間とユーザ CPU 時間の合計、および使用された CPU 時間の割合
Used Tapes	テープを使用した、キューのジョブの数
Ave Queue Wait	開始前のキューの平均待ち時間（秒単位）

定期レポート

この節では、以下の 2 つの定期レポートについて説明します。

- 119ページの「統合アカウンティング・レポート」
- 119ページの「コマンド集計レポート」

統合アカウンティング・レポート

以下の統合アカウンティング・レポートの使用状況の値は、報告期間中に、指定されたユーザおよびプロジェクトの全プロセスによって使用されたリソースの合計量です。

ヘッダ	説明
PROJECT NAME	リソースの使用状況情報に関連付けられたプロジェクト
USER ID	ユーザ識別子
LOGIN NAME	ユーザのログイン名
CPU_TIME	CPU 時間の累積（秒単位）
KCORE * CPU-MIN	プロセスの CPU 時間 1 分あたりに使用されたコア（物理）メモリの累計（Kバイト単位）
KVIRT * CPU-MIN	CPU 時間 1 分あたりに使用された仮想メモリの累積（Kバイト単位）
IOWAIT BLOCK	ブロック I/O 待ち時間の累積（秒単位）
IOWAIT RAW	raw I/O 待ち時間の累積（秒単位）
DISK BLOCKS	使用されたディスク・ブロックの合計数
DISK SAMPLES	ディスク・ブロックの使用値を取得するためにディスク・アカウンティングが実行された回数
FEE	`csachargefee`(1M) からユーザに対して請求された合計料金
SBUs	ユーザとプロジェクトに対して請求されたシステム課金単位

コマンド集計レポート

以下の情報は、報告期間中のコマンドの使用状況の概要です。使用状況の値は、指定されたコマンドのすべての呼出しで使用されたリソースの合計量です。日別コマンド集計レポートとは異なり、定期コマンド集計レポートには、すべてのコマンド・エントリが表示されます。定期コマンド集計レポートでは、一度しか実行されなかったコマンドは「***other」エントリ内にまとめられず、個別にリストされます。

ヘッダ	説明
COMMAND NAME	コマンド（プログラム）の名前

NUMBER OF COMMANDS	コマンドの実行回数
TOTAL KCORE-MINUTES	CPU 時間 1 分あたりに使用されたコア（物理）メモリの合計（Kバイト単位）
TOTAL KVIRT-MINUTES	CPU 時間 1 分あたりに使用された仮想メモリの合計（Kバイト単位）
TOTAL CPU	使用された CPU 時間の合計（分単位）
TOTAL REAL	使用された実 (wall-clock) 時間の合計（分単位）
MEAN SIZE KCORE	使用されたコア（物理）メモリの平均量（Kバイト単位）
MEAN SIZE KVIRT	使用された仮想メモリの平均量（Kバイト単位）
MEAN CPU	使用された CPU 時間の平均（分単位）
HOG FACTOR	合計 CPU 時間を合計実時間（経過時間）で除算した値
K-CHARS READ	読込まれた文字の合計数（Kバイト単位）
K-CHARS WRITTEN	書込まれた文字の合計数（Kバイト単位）
BLOCKS READ	読込まれたブロックの合計数
BLOCKS WRITTEN	書込まれたブロックの合計数

CSA と既存の IRIX ソフトウェア

この節では、IRIX オペレーティング・システムの既存のドキュメントに対する変更および追加事項について説明します。

acct(1M) マン・ページ

acctdisk コマンドには、-c オプションがあります。このオプションを指定すると、標準入力が読込まれ、レコードが cacct 形式に変換された後、標準出力に書込まれます。

acctsh(1M) マン・ページ

lastlogin(1M) コマンドには、*infile* 引数を指定する -c オプションがあります。このオプションは、lastlogin に *infile* の処理を指定します。infile は、cacct 形式のアカウンティング・ファイルです。

dodisk コマンドの情報は、新しい dodisk(1M) のマン・ページに移動されています。

dodisk(1M) マン・ページ

IRIX 6.5.8 リリースでは、dodisk(1M) のマン・ページが新しく追加されました。dodisk コマンドの情報は、以前は acctsh(1M) のマン・ページに記載されていました。

explain(1) マン・ページ

CSA では、メッセージ・カタログ・システムが使用されます。CSA でメッセージ・カタログに使用されるファイルには、次の 2 つがあります。

- /usr/lib/locale/C/LC_MESSAGES/acct.cat
- /usr/lib/locale/C/LC_MESSAGES/acct.exp

6.5.8f リリースの IRIX オペレーティング・システムでは、CSA ソフトウェア製品のグループ・コード acct が、explain(1) のマン・ページに追加されました。

capabilities(4) マン・ページ

基本アカウンティングと CSA では、同じ capability が必要です。アカウンティング設定システム・コール acct(2) を使用する際に必要な特権は、CAP_ACCT_MGT です。新しい acctctl(3c) を呼出す場合も、この特権が必要です。6.5.8f リリースの IRIX オペレーティング・システムでは、capabilities(4) のマン・ページに acctctl(3c) が追加されました。

アカウンティング・データの移行

基本アカウンティングおよび拡張アカウンティングのレコードに対する変更はありません。これら 2 つの IRIX アカウンティング方式と CSA の間では、アカウンティング・データの移行は行われません。つまり、基本アカウンティングには基本アカウンティングのコマンドを使用し、サード・パーティ製のパッケージには拡張アカウンティング・データを使用する必要があります。

CSA アカウンティングのコマンドは、CSA アカウンティング・データに対してのみ使用できます。CSA のコマンドは、基本アカウンティングまたは拡張アカウンティングのレコードを処理できません。基本アカウンティングのコマンドは、CSA で生成されたアカウンティング・データを処理できません。

CSA のマン・ページ

man コマンドは、すべてのリソース管理コマンドに関するオンライン・ヘルプを提供します。マン・ページをオンラインで表示するには、「man *commandname*」と入力します。

一般ユーザ用マン・ページ

CSA ソフトウェアでは、以下の一般ユーザ用マン・ページが用意されています。

一般ユーザ用マン・ページ	説明
csacom(1)	CSA のプロセス・アカウンティング・ファイルを検索、出力します。
ja(1)	ユーザ・ジョブのアカウンティング情報の取得を開始、停止します。

管理者用マン・ページ

CSA ソフトウェアでは、以下の管理者用マン・ページが用意されています。

管理者用マン・ページ	説明
csaaddc(1m)	cacct レコードを結合します。
csabuild(1m)	アカウンティング・レコードをジョブ・レコードにまとめます。
csachargefee(1m)	ユーザに対して料金を請求します。
csackpacct(1m)	CSA プロセス・アカウンティング・ファイルのサイズをチェックします。
csacms(1m)	プロセスごとのアカウンティング・レコードから、コマンドの使用状況を集計します。
csacon(1m)	sorted pacct ファイルのレコードを圧縮します。
csacrep(1m)	統合アカウンティング・データについて報告します。
csadrep(1m)	デーモンの使用状況について報告します。
csaedit(1m)	アカウンティング情報を表示、編集します。
csagetconfig(1m)	アカウンティング設定ファイル内で、指定した引数を検索します。
csajrep(1m)	sorted pacct ファイルのジョブ・レポートを出力します。
csarecy(1m)	未完了のジョブを次回のアカウンティング実行時にリサイクルします。

`csaswitch`(1m)	異なる種類の CSA のステータスをチェックしたり、有効または無効に切替えたり、アカウンティング・ファイルを保守の目的で交換したりします。
`csaverify`(1m)	アカウンティング・レコードが有効であることを確認します。

IRIX のメモリ使用量

この節では、IRIX オペレーティング・システムの物理および仮想メモリの使用量の情報を提供するコマンドについて説明します。

この章は、以下の節で構成されています。

メモリ使用量コマンド

ほとんどのメモリ使用量コマンドは、プロセスごとまたはジョブごとの現在のメモリ使用量のスナップショットを表示します。

以下に、プロセスごとのコマンドの例を示します。

- `gmemusage`(1)
- `pmem`(1)
- `top`(1)
- `ps`(1)

これらのコマンドについての詳細は、該当するマン・ページを参照してください。

ジョブごとのコマンドには以下が含まれます。

- `jstat`(1)

`csacom`(1) や `ja`(1) などの完全システム・アカウンティング (CSA: Comprehensive System Accounting) コマンドは、プロセスまたはジョブが終了した後にメモリ使用量の履歴情報を提供します。

`jstat`(1) コマンドは、現在の使用量、およびジョブ内で同時に実行されているすべてのプロセスの最大メモリ値を報告します。

-l オプションが指定されている場合、jstat コマンドは、現在のジョブに関する現在の使用量、最大使用量、現在の制限、および最大制限の情報を表示します（vmemory は仮想メモリ、ressetsize は常駐セット・サイズです）。

以下に、jstat -l オプションの出力例を示します。

```
% jstat -l

JID                 OWNER          COMMAND
------------------- -------------- --------------
0x106f              user1          -tcsh



LIMIT NAME          USAGE          HIGH USAGE     CURRENT LIMIT  MAX LIMIT
------------------- -------------- -------------- -------------- --------------
cputime             0              0              unlimited      unlimited
datasize            272k           544k           unlimited      unlimited
files               8              32             400            5000
vmemory             4224k          14112k         unlimited      unlimited
ressetsize          3520k          6384k          unlimited      unlimited
threads             1              1              unlimited      unlimited
processes           2              7              1024           1024
physmem             3520k          6384k          unlimited      unlimited
```

ja(1) コマンドの -s オプションは、ジョブ内の単一の最大プロセス・メモリの最大メモリ値を報告します。

この値は終了したプロセスのアカウンティング・レコードから収集されるため、ジョブ内のすべてのプロセスの累計最大値にはなりません。

以下に、ja -s オプションの出力例を示します。

```
% ja -s

Job CSA Accounting - Summary Report
          ===================================

          Job Accounting File Name         : /tmp/ja.username
          Operating System                 : IRIX64 snow 6.5 10120733 IP27
          User Name (ID)                   : username (10320)
          Group Name (ID)                  : resmgmt (16061)
          Project Name (ID)                : CSA(40)
          Array Session Handle             : 0x000000000000034b
```

```
Job ID                          : 0x310
Report Starts                   : 01/23/00 18:13:38
Report Ends                     : 01/23/00 18:17:05
Elapsed Time                    :          207      Seconds
User CPU Time                   :            0.9340 Seconds
System CPU Time                 :            0.0643 Seconds
Run Queue Wait Time             :            0.6463 Seconds
Block I/O Wait Time             :            0.1888 Seconds
Raw I/O Wait Time               :            0.1323 Seconds
CPU Time Core Memory Integral   :            0.4305 Mbyte-seconds
CPU Time Virtual Memory Integral :           4.3298 Mbyte-seconds
Maximum Core Memory Used        :            0.1094 Mbytes
Maximum Virtual Memory Used     :           38.0000 Mbytes
Characters Read                 :            0.0603 Mbytes
Characters Written              :            0.0023 Mbytes
Blocks Read                     :            7
Blocks Written                  :            0
Logical I/O Read Requests       :           35
Logical I/O Write Requests      :           42
Number of Commands              :            7
System Billing Units            :            0.0000
```

CSA のメモリ積算 (Memory Integral) は、クロック周期で測定した CPU 時間に対して、使用されたメモリ量の時間積算をレポートします。

CSA、拡張アカウンティング、および `jstat` (1) コマンドはすべて同じカーネル・カウンタにアクセスして、プロセスごとのメモリ・サイズを取得します。これらのプロセスごとのメモリ・サイズの値は、追加のカーネル・カウンタにより、`jstat` コマンドで報告されたジョブ・メモリ・サイズの値に累積されます。CSA では、カーネル外部でジョブ値への累積を行います。

共有メモリ

ジョブ制限と CSA のどちらも、ジョブ内のすべてのプロセスのメモリ使用量の値を報告します。ジョブ内のプロセスは、共有メモリ・セグメントにアクセスできます。これらのセグメントは、使用する共有メモリ・セグメントのタイプに応じて、ジョブ内のプロセス間で、またはジョブ外のプロセスと共有できます。ジョブ全体のメモリ使用量を判断する場合は、セグメントにアクセスする各プロセスに対して 1 回ずつ、共有メモリ・セグメントがカウントされます。このため、使用量の値が予想を大きく上回る場合があります。これは特に、多数のプロセスが 1 つまたは複数のメモリ・セグメントを共有する並列実行アプリケーションに当てはまります。

プロセス間の共有メモリは、CSA または `jstat` コマンドでは比例分割計算されません。物理および仮想の両方の共有メモリ・ページは、ページにアクセスする各プロセスのメモリ・サイズのカウントに重複して含まれます。

物理メモリ

カーネルは、物理最大メモリ値、現在の使用量の値、およびメモリ積算値を定期的に計算しています。これらの値はプロセスまたはジョブの常駐セット・サイズですが、マップされたデバイス（例: グラフィック・デバイス）に関連付けられているページは含まれません。

仮想メモリ

物理メモリ使用量の値と異なり、仮想メモリの値はカーネルによってカーネル・カウンタで継続的に最新に保たれています。プロセスの仮想メモリ・サイズが増加すると、カーネルは CSA 最大値を増やします。`jstat` の現在の使用量と最大値は、必要に応じてカーネル内で定期的に設定されます。また、カーネルは CSA 仮想メモリ積算も定期的に計算します。

これらの値にはプロセスまたはジョブの仮想メモリ・サイズ（テキスト、データ、スタック、共有メモリ、マップされたファイル、および共有ライブラリ）が含まれますが、マップされたデバイス（例: グラフィック・デバイス）に関連付けられているページは含まれません。

Array Service

Array Service には、管理者用コマンド、ライブラリ、デーモン、およびアレイ全体にわたるプログラムの実行をサポートするカーネル拡張が含まれます。

Array Service の中心的な概念はアレイ・セッション・ハンドル (ASH: Array Session Handle) で、これは、複数のシステムに分散されている可能性がある互いにする関連プロセスを論理的にグループ化するために使用されます。ASH によって、アレイ全体にわたるグローバル・プロセス名前空間が作成され、アカウンティングと管理が容易になります。

Array Service は、アレイを構成するノードをリストした設定データベースも提供します。アレイ・インベントリ問い合わせ機能により、各ノードの設定を標準的な方法で一元的に表示できます。その他のアレイ・ユーティリティを使用すると、管理者は分散アレイ・アプリケーションへの照会や操作を行うことができます。

Array Service パッケージは、以下の主要なコンポーネントで構成されます。

array デーモン	ASH 値を割当て、ノード設定に関する情報、およびプロセス ID と ASH の関連に関する情報を保持します。array デーモンは各ノードに常駐し、協調動作します。
アレイ設定データベース	array デーモンおよびユーザ・プログラムが使用するアレイ設定を記述します。各ノードに 1 つのアレイ設定データベースがあります。
`ainfo` コマンド	ユーザまたは管理者が、アレイ設定データベース、および ASH 値とプロセスに関する情報を照会できるようにします。
`array` コマンド	指定したコマンドを 1 つまたは複数のノードで実行します。コマンドは管理者があらかじめ設定データベースに定義しておきます。
`arshell` コマンド	現在の ASH 値を使用して、別のノードでリモートでコマンドを起動します。
`aview` コマンド	各ノードのステータスをマルチウィンドウでグラフィカルに表示します。
`libarray` ライブラリ	array デーモンのサービスおよびアレイ設定データベースに対してユーザ・プログラムが呼出すことができる関数のライブラリ。

`ainfo`、`array`、`arshell`、および `aview` コマンドの使い方については、130ページの「アレイの使用」で説明しています。`libarray` ライブラリの使い方については、158ページの「Array Service ライブラリ」で説明しています。

アレイの使用

アレイ・システムは、高速ネットワークと Array Service 3.5 ソフトウェアで相互に結び付けられたサーバであるノードの集合です。アレイのユーザは、より高いパフォーマンスと追加のサービスという利点を得られます。アレイのユーザは、ジョブ制御、ログイン、およびパスワード管理用の使い慣れたコマンドとリモート実行を使用してシステムにアクセスします。

Array Service 3.5 では、従来の機能にアレイのユーザやアレイ管理者向けの追加のサービスが加わっています。これらの拡張には、グローバル・セッション管理のサポート、アレイ設定の管理、バッチ処理、メッセージ・パッシング、システム管理、およびパフォーマンスの視覚化のサポートが含まれます。

この節では、アレイを使用するための拡張を説明するとともに、より詳細な情報への参照を示します。主なトピックは、以下のとおりです。

- 130ページの「アレイ・システムの使用」では、ユーザが理解しておく必要がある情報と、ユーザが利用できる主な機能の概要を説明します。
- 133ページの「ローカル・プロセスの管理」では、1 つのノード内のプロセスをリストおよび制御するための従来のツールについて理解します。
- 134ページの「Array Service コマンドの使用」では、Array Service コマンドで使われる共通の概念、オプション、および環境変数について説明します。
- 138ページの「アレイへの問い合わせ」では、Array Service コマンドを使用してアレイとその負荷を理解する方法の概要を、例を使って説明します。
- 136ページの「共通のコマンド・オプションの概要」では、コマンド・オプションの概要を説明します。
- 142ページの「分散プロセスの管理」では、Array Service コマンドを使用して、複数のノードのプロセスをリストおよび制御する方法の概要を説明します。

アレイ・システムの使用

アレイ・システムでは、アレイの複数のノードで分散セッションを実行できます。アレイには以下のいずれかからアクセスできます。

- ワークステーション
- X 端末
- ASCII 端末

どの場合でも、リモート UNIX ホストにログインするときと同じ方法で、アレイの 1 つのノードにログインします。ワークステーションまたは X 端末から複数の端末ウィンドウを開いて、複数のノードにログインできます。

基本的な使用法情報の検索

アレイを使用するには、以下の情報が必要です。

- アレイの名前

  Array Service コマンドでは、この名前 (*arrayname*) を使用します。

- アレイで使用するログイン名とパスワード

  アレイにログインして使用する場合に使用します。

- アレイ・ノードのホスト名

  通常、これらの名前は単純なパターンに沿っており、多くの場合は、*arrayname1*、*arrayname2* というようになります。

- ジョブ・スケジューリング・システムでユーザまたはユーザが属するグループに適用される可能性がある特殊なリソース分散またはアカウンティング規則

アレイ名がわかっている場合は、以下の `ainfo` コマンドを使用して、アレイ・ノードのホスト名を参照できます。

ainfo -a *arrayname* **machines**

アレイへのログイン

アレイ内の各ノードは、関連付けられたホスト名と IP ネットワーク・アドレスを持ちます。通常は、1 つのノードに直接ログインするか、または別のホスト(Array コンソールまたはネットワークに接続されたワークステーションなど)からリモートでログインして、アレイを使用します。たとえば、同じネットワーク上のワークステーションから、次のコマンドを使用して `hydra6` という名前のノードにログインします。

rlogin hydra6

`rlogin` コマンドについての詳細は、`rlogin`(1) のマン・ページを参照してください。

アレイ・リソースをスケジュールするために、アレイのシステム管理者がノードへの直接ログインを禁止している場合があります。ノードへの直接ログインを禁止するようサイトが設定されている場合は、アレイに作業を送信する方法(多くの場合は、リモート実行ソフトウェアまたはバッチ・キュー機能を使用)を管理者に問い合わせてみてください。

プログラムの呼出し

アレイにアクセスした後、以下の複数のクラスのプログラムを呼出すことができます。

- 通常の(逐次実行)アプリケーション

- ノード内の並列実行・共有メモリ・アプリケーション
- ノード内の並列実行・メッセージパッシング・アプリケーション
- 複数のノード（または、Array Service 3.5 が実行されている同じネットワーク上の他のサーバ）に分散された並列実行・メッセージパッシング・アプリケーション

許可されている場合は、ログインしているシェルのコマンド・ラインから明示的にプログラムを呼出すか、またはリモート実行またはバッチ・キュー・システムを使用できます。

X Windows クライアントであるプログラムは、X サーバ（X 端末または X Windows が実行されているワークステーションのいずれか）から起動する必要があります。

アプリケーション・クラスの中には、実行時に、コマンド・ライン・オプション、環境変数、またはサポート・ファイルの形式での入力が必要なものがあります。例は、次のとおりです。

- X クライアント・アプリケーションでは、DISPLAY 環境変数を、ウィンドウが表示される X サーバ（ワークステーションまたは X 端末）が指定されるように設定する必要があります。
- マルチスレッド化プログラムでは、スレッドの数を記述する環境変数を設定しなければならない場合があります。

  たとえば、並列処理ディレクティブを使用する C および Fortran プログラムでは、MP_SET_NUMTHREADS 変数がテストされます。
- メッセージ・パッシング・インタフェース (MPI: Message Passing Interface) および並列仮想マシン (PVM: Parallel Virtual Machine) メッセージパッシング・プログラムでは、指定したノードで呼出すタスクの数をサポート・ファイルに記述しなければならない場合があります。

132ページの表7-1 に、プログラム呼出しに関する参考情報を示します。

表7-1 プログラム呼出しの参考情報

トピック	マン・ページ
リモート・ログイン	rlogin(1)
環境変数の設定	environ(5)、env(1)

ローカル・プロセスの管理

各 UNIX プロセスは、プロセスが実行されるノード内でプロセスを識別する番号であるプロセス ID (PID) を持ちます。PID はノードにローカルである点を理解しておいてください。このため、異なるノードのプロセスに同じ PID 番号が使用される可能性があります。

ノード内で、プロセスをプロセス・グループに論理的にグループ化できます。プロセス・グループは、親プロセスと、親プロセスによって作成されるすべてのプロセスから構成されます。各プロセス・グループは、プロセス・グループ ID (PGID) を持ちます。PID と同様に、PGID もノードにローカルに定義され、アレイ全体での一意性は保証されません。

ローカル・プロセスとシステム使用状況のモニタ

プロセスのステータスは、システム・コマンド `ps` を使用して照会します。ローカル・システム上のすべてのプロセスの完全なリストを生成するには、以下のようなコマンドを使用します。

```
ps -elfj
```

コマンド `top` を使用して、プロセスのアクティビティをモニタできます（端末のウィンドウでの ASCII 表示）。

ローカル・プロセスのスケジューリングと強制終了

`nice` コマンドを使用して、他のプロセスに干渉しないよう、特定のプロセスを低い優先度で起動できます。csh シェルを使用する場合は、`/usr/bin/nice` を指定して、内蔵のシェル・コマンド `nice` を使用しないようにします。シェル全体を低い優先度で起動するには、以下のようなコマンドを使用します。

```
/bin/nice /bin/sh
```

`at` コマンドを使用して、特定の時刻に実行されるようコマンドをスケジュールできます。`kill` コマンドを使用して、プロセスを強制終了または停止できます。PID 13032 のプロセスを破棄するには、以下のようなコマンドを使用します。

```
kill -KILL 13032
```

ローカル・プロセス管理コマンドの概要

134ページの表7-2 に、ローカル・プロセス管理に関する情報の概要を説明します。

表7-2 参考情報: ローカル・プロセス管理

トピック	マン・ページ
プロセス ID とプロセス・グループ	intro(2)
プロセスをリストおよびモニタする	ps(1)、top(1)
低い優先度でプログラムを実行する	nice(1)、batch(1)
スケジュールした時刻にプログラムを実行する	at(1)
プロセスを終了する	kill(1)

Array Service コマンドの使用

アプリケーションによって複数のノードでプロセスが起動される場合、PID と PGID はアプリケーションの管理に適しません。Array Service 3.5 のコマンドを使用することで、アレイ全体を表示したり、マルチノード・プログラムのプロセスを制御できます。

メモ: Array Service コマンドは、アレイ・システムに接続された任意のワークステーションから使用できます。アレイ・ノードにログインしている必要はありません。

134ページの表7-3 に示されるコマンドは Array Service の操作に一般的に使用されます。

表7-3 Array Service 一般コマンド

トピック	マン・ページ
Array Service の概要	array_services(5)
ainfo コマンド	ainfo(1)
array コマンド	使用: array(1)、設定: arrayd.conf(4)
arshell コマンド	arshell(1)

トピック	マン・ページ
aview コマンド	aview(1)
newsess コマンド	newsess (1)

アレイ・セッションについて

Array Service は、起動時に開始されるバックグラウンド・プロセスであるデーモンと、ainfo(1) などのコマンドの集合から構成されます。これらのコマンドは、各ノードのデーモン・プロセスを呼出して、必要な情報を取得します。

Array Service の基礎となる概念の 1 つは、アレイ・セッションです。これは、プロセスが実行されている場所を問わず、1 つのアプリケーションのすべてのプロセスを表す用語です。通常は、ログイン・シェルと、そこから起動するプログラムがアレイ・セッションを構成します。バッチ・ジョブは 1 つのアレイ・セッションです。新しいアレイ・セッション ID で新しいシェルを作成できます。

各セッションは、アレイ・セッション・ハンドル (ASH: Array Session Handle) で識別されます。ASH は、そのセッションに含まれる任意のプロセスで共有される番号です。プログラムのプロセスが異なるノードで実行されている場合でも、ASH を使用して、すべてのプロセスを照会および制御できます。

アレイとノードの名前について

各ノードはサーバであり、ホスト名を持ちます。ノードのホスト名は、そのノードで実行する hostname(1) コマンドによって返されます。

```
% hostname
tokyo
```

このコマンドは単純で、hostname(1) のマン・ページで説明されています。ホスト名の構文や、ホスト名をハードウェア・アドレスに解決する方法に関するより複雑な問題については、hostname(5) で説明されています。

アレイ・システム全体も名前を持ちます。ほとんどのインストールではアレイは 1 つだけで、目的のアレイを指定する必要はありません。ただし、1 つのネットワークで複数のアレイを利用することも可能で、この場合、Array Service コマンドを特定のアレイに送信できます。

認証キーについて

アレイ管理者は、アレイ内の一部またはすべてのノードに認証コード（64 ビットの数値）を設定できます。58 ページの「認証コードの設定」を参照してください。認証コードが設定されている場合は、Array Service コマンドを使用するたびに、使用するノードに適した認証コードをコマンド・オプションとして指定する必要があります。認証コードが必要であるかどうかは、システム管理者に問い合わせてください。

共通のコマンド・オプションの概要

Array Service コマンド `ainfo`(1)、`array`(1)、`arshell`(1)、`aview`(1)、および `newsess`(1) のコマンド・オプションは共通です。表7-4 に、これらのオプションの概要を示します。すべてのオプションがすべてのコマンドで有効なわけではありません。また、各コマンドには、ここに示す以外にも固有のオプションがあります。一部のオプションのデフォルト値は、次のトピックに示す環境変数によって設定されます。

表7-4 Array Service コマンド・オプションの概要

オプション	使用できるコマンド	説明
`-a` *array*	`ainfo`、`array`、`aview`	複数のアレイにアクセスできる場合に特定のアレイを指定する。
`-D`	`ainfo`、`array`、`arshell`、`aview`	`array` デーモンを通じてではなく、直接他のノードにコマンドを送信する。
`-F`	`ainfo`、`array`、`arshell`、`aview`	他のノードに `array` デーモンを通じてコマンドを転送する。
`-Kl` *number*	`ainfo`、`array`、`aview`	ローカル・ノードの認証キー（64 ビットの数値）。
`-Kr` *number*	`ainfo`、`array`、`aview`	リモート・ノードの認証キー（64 ビットの数値）。

オプション	使用できるコマンド	説明
-l （文字「l(エル)」）	ainfo、array	送信先のノードのコンテキスト内で実行する(必ずしも現在のノードではない)。
-l *port*	ainfo、array、arshell、aview	array デーモンの非標準のポート番号。
-s *hostname*	ainfo、array、aview	送信先ノードを指定する。

単一のノードの指定

-l および -s オプションは連携して動作します。-l(「ローカル」を意味する文字「l(エル)」)オプションは、コマンドのスコープを、コマンドが実行されるノードに制限します。デフォルトでは、コマンドを入力するノードです。-l を使用しない場合は、アレイのすべてのノードが照会コマンドのスコープになります。-s(サーバ、またはノード名)オプションは、コマンドを転送して、アレイ内の指定したノードで実行します。これらのオプションは、照会コマンドで以下のように連携して動作します。

- ローカル・ノードによって認識されているすべてのノードに問い合わせるには、どちらのオプションも使用しません。
- ローカル・ノードのみに問い合わせるには、-l のみを使用します。
- 指定したノードによって認識されているすべてのノードに問い合わせるには、-s オプションのみを使用します。
- 特定のノードのみに問い合わせるには、-s および -l の両方を使用します。

共通の環境変数

Array Service コマンドは、より一般的ではないコマンド・オプションのデフォルト値を定義する場合、環境変数に依存します。表7-5 に、これらの変数の概要を示します。

表7-5 Array Service の環境変数

変数名	使用されるケース	未定義の場合のデフォルト
ARRAYD_FORWARD	文字 *y* で始まる文字列で定義されている場合、コマンドはすべてデフォルトで array デーモンに転送される(オプション -F)。	デフォルトではコマンドは直接通信する(オプション -D)。
ARRAYD_PORT	送信先のノードの array デーモンによってモニタされるポート(ソケット)番号。	標準のポート番号 5434、またはオプション -p で指定した番号。
ARRAYD_LOCALKEY	ローカル・ノードの認証キー(オプション -Kl)。	-Kl オプションを使用した場合を除き、認証なし。
ARRAYD_REMOTEKEY	送信先のノードの認証キー(オプション -Kr)。	-Kr オプションを使用した場合を除き、認証なし。
ARRAYD	送信先のノード(-s オプションで指定していない場合)。	ローカル・ノード、または -s で指定したノード。

アレイへの問い合わせ

アレイ・システムのどのユーザでも、Array Service コマンドを使用して、アレイのハードウェア・コンポーネントやソフトウェアのワークロードをチェックできます。必要なコマンドは、ainfo、array、および aview です。

アレイ名の参照

ネットワークに複数のアレイ・システムが含まれる場合は、次の例のように、特定のアレイ・ノードで ainfo arrays を使用して、設定されているすべてのアレイ名をリストできます。

```
homegrown% ainfo arrays
Arrays known to array services daemon
ARRAY DevArray
    IDENT 0x3381
ARRAY BigDevArray
```

```
    IDENT 0x7456
ARRAY test
    IDENT 0x655e
```

アレイ名は、管理者によってアレイ・データベースに設定されています。異なるアレイでは、他のアレイ名の集合が認識される場合があります。

ノード名の参照

次の例のように、ainfo machines を使用して、現在のアレイ内のすべてのノードの名前と一部の機能を参照できます。

```
homegrown 175% ainfo -b machines
machine homegrown homegrown 5434 192.48.165.36 0
machine disarray disarray 5434 192.48.165.62 0
machine datarray datarray 5434 192.48.165.64 0
machine tokyo tokyo 5434 150.166.39.39 0
```

この例では、ainfo の -b オプションを使用して、簡潔に表示しています。

ノードの機能の参照

ainfo nodeinfo を使用して、アレイ内の 1 つまたはすべてのノードに関する詳細な情報を要求できます。ローカル・ノードに関する情報を取得するには、ainfo -l nodeinfo を使用します。一方、他の特定のノード(例: ノード tokyo)のみに関する情報を取得するには、次の例のように、-l および -s を使用します(簡潔にするために編集してあります)。

```
homegrown 181% ainfo -s tokyo -l nodeinfo
Node information for server on machine "tokyo"
MACHINE tokyo
    VERSION  1.2
    8 PROCESSOR BOARDS
        BOARD: TYPE 15   SPEED 190
            CPU:   TYPE 9   REVISION 2.4
            FPU:   TYPE 9   REVISION 0.0
...
    16 IP INTERFACES  HOSTNAME tokyo   HOSTID 0xc01a5035
        DEVICE  et0    NETWORK    150.166.39.0    ADDRESS   150.166.39.39  UP
        DEVICE atm0    NETWORK 255.255.255.255    ADDRESS         0.0.0.0  UP
        DEVICE atm1    NETWORK 255.255.255.255    ADDRESS         0.0.0.0  UP
...
```

```
    0 GRAPHICS INTERFACES
    MEMORY
        512 MB MAIN MEMORY
        INTERLEAVE 4
```

-l オプションを省略した場合、送信先のノードは、そのノードで認識されているすべてのノードに関する情報を返します。

ユーザ名とワークロードの参照

システム・コマンド who(1)、top(1)、および uptime(1) は、一般に特定のサーバのユーザとワークロードに関する情報を取得する場合に使用します。array(1) コマンドは、これらのコマンドと同様の内容でアレイ全体に関する情報を提供します。

ユーザ名の参照

アレイ全体にログインしているすべてのユーザの名前を取得するには、array who を使用します。特定のノード(例: tokyo)にログインしているユーザの名前を参照するには、次の例のように、-l および -s を使用します(簡潔にするため、およびセキュリティの点から編集してあります)。

```
 homegrown 180% array -s tokyo -l who
joecd    tokyo        frummage.eng.sgi -tcsh
joecd    tokyo        frummage.eng.sgi -tcsh
benf     tokyo        einstein.ued.sgi. /bin/tcsh
yohn     tokyo        rayleigh.eng.sg vi +153 fs/procfs/prd
...
```

ワークロードの参照

array コマンドの 2 つのバリエーションは、ワークロード情報を返します。uptime と同様の内容でアレイ全体の情報を返すコマンドは、次に示すように array uptime です。

```
homegrown 181% array uptime
   homegrown:  up 1 day,  7:40,  26 users,  load average: 7.21, 6.35, 4.72
    disarray:  up  2:53,  0 user,  load average: 0.00, 0.00, 0.00
    datarray:  up  5:34,  1 user,  load average: 0.00, 0.00, 0.00
       tokyo:  up 7 days,  9:11,  17 users,  load average: 0.15, 0.31, 0.29
homegrown 182% array -l -s tokyo uptime
       tokyo:  up 7 days,  9:11,  17 users,  load average: 0.12, 0.30, 0.28
```

コマンド array top は、現在最も CPU 時間を使用しているプロセスを ASH 値とともにリストします。次に、例を示します。

```
homegrown 183% array top
        ASH         Host            PID User       %CPU Command
-------------------------------------------------------------------
0x1111ffff00000000 homegrown        5 root       1.20 vfs_sync
0x1111ffff000001e9 homegrown     1327 guest      1.19 atop
0x1111ffff000001e9 tokyo        19816 guest      0.73 atop
0x1111ffff000001e9 disarray      1106 guest      0.47 atop
0x1111ffff000001e9 datarray      1423 guest      0.42 atop
0x1111ffff000000c0 homegrown    29683 kchang     0.37 ld
0x1111ffff0000001e homegrown     1324 root       0.17 arrayd
0x1111ffff00000000 homegrown      229 root       0.14 routed
0x1111ffff00000000 homegrown       19 root       0.09 pdflush
0x1111ffff000001e9 disarray      1105 guest      0.02 atopm
```

このコマンドでも、-l および -s オプションを使用して単一のノードに関するデータを選択できます。

「ArrayView」を使用した参照

「ArrayView」ウィンドウには、アレイのステータスが表示されます。このウィンドウはコマンド aview を使用して起動でき、図7-1 に示すようなウィンドウが表示されます。最上部のウィンドウには、各ノードに対して 1 行が表示されます。各ノードに対して 1 つのウィンドウがあり、見出しとしてノード名とハードウェア設定が表示されます。各ウィンドウには、そのノード内で最も使用率の高いプロセスのスナップショットが含まれます。

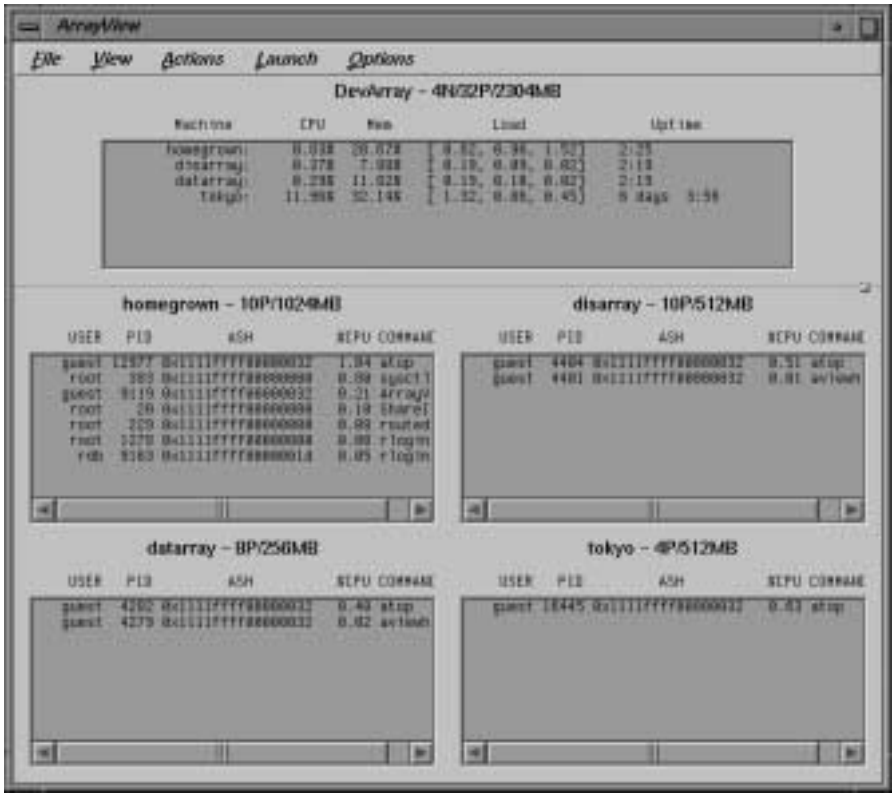

図7-1「ArrayView」の一般的な表示

分散プロセスの管理

Array Service 3.5 のコマンドを使用して、アレイ・システムの複数のノードに分散されているプロセスを作成および管理できます。

アレイ・セッション・ハンドル (ASH: Array Session Handles) について

アレイ・システムでは、複数のノードに分散されたプロセスを持つプログラムを起動できます。このようなプロセスの集合に名前を付けるため、Array Service 3.5 ソフトウェアは、各プロセスを アレイ・セッション・ハンドル (ASH: Array Session Handle) に割当てます。

ASH は、PID または PGID と異なりアレイ全体で一意の数値です。ASH は、プロセスがどのノードで実行されているかにかかわらず、単一のアレイ・セッションの一部であるすべてのプロセスで同じです。ASH 値は、Array Service コマンドを使用して表示および使用します。アレイ・ノードにログインするたびにシェルに ASH が割当てられ、そのシェルから起動するすべてのプロセスでこの ASH が使用されます。

コマンド `ainfo ash` は、ローカル・ノードの現在のプロセスの ASH を返します。これは、常にその `ainfo` コマンド自体の ASH になります。

```
homegrown 178% ainfo ash
Array session handle of process 10068: 0x1111ffff000002c1
homegrown 179% ainfo ash
Array session handle of process 10069: 0x1111ffff000002c1
```

上の例では、`ainfo` コマンドの各インスタンスは新しいプロセスで、最初は PID 10068、次は PID 10069 でしたが、ASH はどちらの場合も同じです。これは、「すべてのプロセスがその親の ASH を継承する」という非常に重要な規則を表しています。この場合、`array` の各インスタンスはコマンド・シェルによってフォークされており、表示される ASH 値は、子プロセスによって継承されているシェルの ASH 値になります。

次のように、コマンド `ainfo newash` を使用して、新しいグローバル ASH を作成できます。

```
homegrown 175% ainfo newash
Allocating new global ASH
0x11110000308b2f7c
```

現時点では、この機能にはほとんど用途はありません。ASH を変更できる既存のコマンドがないため、新しい ASH を別のコマンドに割当てることができません。コマンド・ライン・オプションから ASH を取得し、Array Service 関数 `setash()` を使用して ASH を変更するプログラムなどが考えられます（ただし、このようなプログラムには特権を与える必要があります）。このようなプログラムは Array Service 3.5 では配布されていません（ただし、163ページの「アレイ・サービス・ハンドルの管理」を参照してください）。

プロセスと ASH 値のリスト

コマンド `array ps` は、アレイ内のすべてのノードで実行されているすべてのプロセスの概要を返します。画面には、ASH、ノード、PID、関連付けられているユーザ名、CPU 時間の累積、およびコマンド文字列が表示されます。

特定のノードのプロセスをすべてリストするには、-l および -s オプションを使用します。特定の ASH または特定のユーザ名に関連付けられているプロセスをリストするには、次の例のように、grep を使用して戻り値をパイプします(スペースを節約するために編集してあります)。

```
homegrown 182% array -l -s tokyo ps | fgrep wombat
0x261cffff0000054c        tokyo 19007   wombat    0:00 -csh
0x261cffff0000054a        tokyo 17940   wombat    0:00 csh -c (setenv...
0x261cffff0000054c        tokyo 18941   wombat    0:00 csh -c (setenv...
0x261cffff0000054a        tokyo 17957   wombat    0:44 xem -geometry 84x42
0x261cffff0000054a        tokyo 17938   wombat    0:00 rshd
0x261cffff0000054a        tokyo 18022   wombat    0:00 /bin/csh -i
0x261cffff0000054a        tokyo 17980   wombat    0:03 /usr/gnu/lib/ema...
0x261cffff0000054c        tokyo 18928   wombat    0:00 rshd
```

プロセスの制御

arshell コマンドを使用して、他の単一のノードで任意のプログラムを起動できます。array コマンドを使用して、指定した ASH に関連付けられているすべてのプロセスを中断、再開、または強制終了できます。

arshell の使用

arshell コマンドは、一般的な rsh コマンドの Array Service 拡張版で、指定したアレイ・ノードで単一のシステム・コマンドを実行します。rsh と違うのは、リモート・シェルが呼出し元のシェルと同じ ASH で実行される点です(これは単純な rsh には当てはまりません)。次の例に、この違いを示します。

```
homegrown 179% ainfo ash
Array session handle of process 8506: 0x1111ffff00000425
homegrown 180% rsh guest@tokyo ainfo ash
Array session handle of process 13113: 0x261cffff0000145e
homegrown 181% arshell guest@tokyo ainfo ash
Array session handle of process 13119: 0x1111ffff00000425
```

arshell を使用して、関連のないプログラムの集合を、単一の ASH を使用して複数のノードで起動できます。その後、146ページの「セッション・プロセスの管理」で説明するコマンドを使用して、これらのプログラムを停止、再開、または強制終了できます。

MPI および PVM は、どちらも arshell を使用して分散プロセスを起動します。

Tip: シェルはそれぞれ固有の ASH を持つプロセスです。array コマンドを使用して、シェルから起動したすべてのプロセスを停止または強制終了した場合は、そのシェルも停止または強制終了することになります。安全に強制終了できる単一の ASH を持つプログラムのグループを作成するには、以下の手順に従います。

1. newsess を使用して、新しい ASH を持つネストされたシェルを作成します。ASH 値を記録します。
2. 新しいシェル内で、arshell を使用して、1 つまたは複数のプログラムを起動します。
3. ネストされたシェルを終了します。

元のシェルに戻ります。ネストされたシェルから起動したすべてのプログラムの ASH は、記録した ASH 値を持っています。現在のシェルは影響を受けないため、この ASH を持つすべてのジョブを安全に強制終了できます。

分散例について

arshell を使用して起動したプログラムは協調動作しません（ソケットなどを使用して、相互に通信するように作成することは可能です）。また、各プログラムを個々に起動する必要があります。

array コマンドは、単一のコマンドを使用してすべてのノードで同時にプログラムを起動できるよう設計されていますが、array で起動できるプログラムは、Array Service 設定ファイルでこのコマンドに設定されているプログラムのみです（このファイルの作成と管理については、147ページの「アレイ設定について」で説明します）。

コマンド・ラインから簡単にプロセス管理の例を示すことができるよう、設定ファイル /usr/lib/array/arrayd.conf には次のコマンドが挿入されています。

```
#
# Local commands
#
command spin                    # Do nothing on multiple machines
        invoke /usr/lib/array/spin
        user    %USER
        group   %GROUP
        options nowait
```

次に示すように、呼出されるコマンド /usr/lib/array/spin は、何もしないループを実行するシェル・スクリプトです。

```
#!/bin/sh
# Go into a tight loop
#
```

```
interrupted() {
        echo "spin has been interrupted - goodbye"
        exit 0
}
trap interrupted 1 2
while [ ! -f /tmp/spin.stop ]; do
        sleep 5
done
echo "spin has been stopped - goodbye"
exit 1
```

このように準備しておくことにより、コマンド array spin を実行すると、アレイ内のすべてのプロセッサでこのスクリプトを実行するプロセスが起動されます。または、array -l -s nodename spin を使用すると、1 つの特定のノードでプロセスが起動されます。

セッション・プロセスの管理

以下のコマンド・シーケンスは、すべてのノードで spin プロセスを作成した後、このプロセスを強制終了します。最初の手順では、それぞれ固有の ASH を持つ新しいセッションを作成します。これは、後で、対話型シェルを強制終了することなく array kill を使用できるようにするためです。

```
homegrown 175% ainfo ash
Array session handle of process 8912: 0x1111ffff0000032d
homegrown 176% newsess
homegrown 175% ainfo ash
Array session handle of process 8941: 0x11110000308b2fa6
```

ASH が 0x11110000308b2fa6 の新しいセッションにおいて、コマンド array spin を使用して、すべてのノードで /usr/lib/array/spin スクリプトを起動します。このテスト・アレイでは、コマンド実行当日には homegrown と tokyo の 2 つのノードのみが存在しています。

```
homegrown 176% array spin
```

終了して元のシェルに戻った後、コマンド array ps を使用して、ASH が 0x11110000308b2fa6 のプロセスをすべて検索します。

```
homegrown 177% exit
homegrown 178% homegrown 177%
homegrown 177% ainfo ash
Array session handle of process 9257: 0x1111ffff0000032d
homegrown 179% array ps | fgrep 0x11110000308b2fa6
0x11110000308b2fa6  homegrown  9033   guest   0:00 /bin/sh /usr/lib/array/spin
0x11110000308b2fa6  homegrown  9618  guest  0:00 sleep 5
```

```
0x11110000308b2fa6        tokyo 26021  guest   0:00 /bin/sh /usr/lib/array/spin
0x11110000308b2fa6      tokyo 26072  guest  0:00 sleep 5
0x1111ffff0000032d  homegrown  9642  guest  0:00 fgrep 0x11110000308b2fa6
```

各ノードに、spin スクリプトに関連する 2 つのプロセスがあります。次のコマンドで、これらをすべて終了します。

```
homegrown 180% array kill 0x11110000308b2fa6
homegrown 181% array ps | fgrep 0x11110000308b2fa6
0x1111ffff0000032d  homegrown 10030  guest  0:00 fgrep 0x11110000308b2fa6
```

コマンド array suspend 0x11110000308b2fa6 を使用すると、プロセスを強制終了する代わりに中断できます（ただし、sleep コマンドが中断されていることを示すことは困難です）。

ジョブ・コンテナ ID について

IRIX 6.5.7f 以降および Array Service バージョン 3.4 以降が実行されているアレイ・システムでは、開始元ホストからジョブ ID (JID) を転送できます。アレイ内の 1 つまたは複数のノードで同じ ASH で実行されているすべてのプロセスは、同じジョブにも属します。ジョブ・コンテナとその使用法についての詳細は、job_limits(5) のマン・ページまたは『IRIX Admin: Resource Administration』を参照してください。

プロセスが開始元ホストで実行されている場合、それらのプロセスは開始元プロセスと同じジョブに属し、そのジョブに対して設定されている制限に従って動作します。リモート・ノードでは、開始元プロセスと同じ JID を持つ新しいジョブが作成されます。リモート・ノードのジョブのジョブ制限には systune のデフォルトが使用され、開始元ホストの systune(1M) コマンドを使用して設定されます。

アレイ設定について

システム管理者は、アレイ設定データベースを初期化する必要があります。アレイ設定データベースは、ほぼすべての ainfo および array コマンドを実行する際に、Array Service デーモンによって使用されるファイルです。アレイ設定についての詳細は、表7-6 に示すマン・ページを参照してください。

表7-6 参考情報: アレイ設定

トピック	マン・ページ
Array Service の概要	array_services(5)
Array Service ユーザ・コマンド	ainfo(1)、array(1)

トピック	マン・ページ
Array Service デーモンの概要	arrayd(1m)
設定ファイルの形式	arrayd.conf(4) 、/usr/lib/array/arrayd.conf.template
設定ファイルの検証プログラム	ascheck(1)
Array Service の簡易設定プログラム	arrayconfig(1m)

設定ファイルの使用について

設定ファイルは、Array Service デーモンの起動時にこのデーモンによって読取られます。通常、このデーモンはシステムの起動中に各ノードで起動されます（設定ファイルの構文をチェックするために、コマンド・ラインから実行することもできます）。

設定ファイルには、ainfo および array に必要な以下のデータが含まれます。

- アレイ・システムの名前。現在のアレイだけでなく、ユーザが Array Service コマンドを実行できるその他の任意のアレイ（ainfo で報告されます）を含みます。
- 各アレイ内のノードの名前とタイプ。Array Service コマンドで使用するホスト名（ainfo で報告されます）は、ここで指定します。
- 該当する場合は、Array Service コマンドで使用しなければならない認証キー（-Kl および -Kr コマンド・オプションとして必要です。136ページの「共通のコマンド・オプションの概要」を参照してください）。
- array コマンドで有効なコマンド。

設定ファイルの形式と内容について

設定ファイルはユーザが判読できるテキスト・ファイルで、以下の 4 つのタイプのエントリが含まれます。これらのエントリについては、後述のトピックで詳しく説明します。

アレイ定義　このアレイと他の既知のアレイを記述します。アレイ名、およびノード名とタイプを含みます。

コマンド定義	array コマンドを使用して呼出すことができるコマンドの使用法と動作を指定します。
認証	アレイにアクセスするために使用しなければならない認証番号を指定します。
ローカル・オプション	その他のエントリまたは arrayd の動作を変更するオプション。

読みやすいように、空白行、空白、および「#」で始まるコメント行を自由に使用できます。エントリは、arrayd によって読取られるどのファイルにでも好きな順序で記述できます。

エントリは、区切り文字とキーワードベースの構文を使用して形成します。キーワードが認識される際に大文字と小文字は区別されませんが、このテキストやマン・ページでは、キーワードを大文字で示しています。arrayd.conf(4) のマン・ページで詳しく説明されているように、エントリは、主にキーワード、数値、引用符付きの文字列から形成されます。

設定データのロード

Array Service デーモン arrayd は、1 つまたは複数のファイル名を引数として取ることができます。このデーモンは、これらの引数をすべて読取り、論理的に継続していると見なして処理します（つまり、引数ファイルを連結します）。ファイル名が指定されていない場合は、/usr/lib/array/arrayd.conf および /usr/lib/array/arrayd.auth を読取ります。起動時に arrayd を起動するスタートアップ・スクリプトによって読取られるファイル /etc/config/arrayd.options に、ファイルの異なる集合、およびその他の arrayd コマンド・ライン・オプションを書込むことができます。

設定データは 2 つ以上のファイルに保存できるため、次の例に示すように、さまざまな方法を組合わせることができます。

- 特定のファイルに、他と異なるアクセス・パーミッションを指定できます。通常、/usr/lib/array/arrayd.conf はどのユーザも読取り可能で、使用可能な array コマンドが含まれます。一方、/usr/lib/array/arrayd.auth は root のみが読取り可能で、認証コードが含まれます。
- 特定のノードに、他と異なる設定データを指定できます。たとえば、特定のコマンドを特定のノードのみに定義したり、端末からのログインに使用されるノードにのみ、他のすべてのノードの名前を認識させることができます。
- NFS マウントされた設定ファイルを使用できます。各マシンには小さな設定ファイルを配置してアレイと認証キーを定義しておき、特定のノードから NFS マウントした、array コマンドを定義するより大きなファイルを使用できます。

設定ファイルを変更した後に、各マシンで arrayd を強制終了して再起動することにより、設定ファイルをデーモンに再ロードできます。この操作は、スクリプト /etc/init.d/array でサポートされています。

デーモンを強制終了するには、次のコマンドを実行します。

```
/etc/init.d/array stop
```

1 回の操作でデーモンの強制終了と再起動を行うには、次のコマンドを実行します。

```
/etc/init.d/array restart
```

メモ: Linux システムでは、スクリプトのパス名は `/etc/rc.d/init.d/array` です。

どのノードの Array Service デーモンも、該当するノードで使用可能な設定ファイル内の情報のみを認識します。これは、特定のノードの使用を制限できるので利点となり得ますが、共通する情報の同期を保つ作業が必要です(この作業を自動的に行う方法の概要は、157ページの「新しい Array コマンドの設計」で説明します)。

置換構文について

`arrayd.conf`(4) のマン・ページでは、設定ファイル内のエントリを形成する場合の構文規則が詳しく説明されています。この構文の重要な特徴は、複数の種類のテキスト置換を使用し、実行時に変数テキストをエントリに置換する点です。

サポートされているほとんどの置換は、コマンド・エントリで使用されます。これらの置換は、`array` コマンドがサブコマンドを呼出すたびに動的に実行されます。その際に、置換によって、該当するサブコマンドの呼出しに固有な値が挿入されます。たとえば、値 `%USER` では、`array` コマンドを呼出すユーザのユーザ ID が挿入されます。コマンドの実行時以外は、このような置換に意味はありません。

その他の設定エントリ内の置換は、`arrayd` が設定ファイルを読取るときに一度だけ実行されます。これらのエントリでは、環境変数の置換のみ有効です。置換される環境変数値は、このデーモンによって、`arrayd` を呼出すスクリプト (`/etc/init.d/array`) から継承された値です。

設定の変更のテスト

設定ファイルには、多くのセクションやオプションが含まれる場合があります(この節の後の節で詳しく説明します)。Array Service コマンド `ascheck` は、アレイ内のすべての設定ファイルの基本的な健全性チェックを実行します。

変更を加えた後に、`-c` および `-f` オプションを使用して `arrayd` をコマンドとして実行することで、個々の設定ファイルの構文が正しいかどうかをチェックできます。たとえば、`/usr/lib/array/arrayd.local` に新しいコマンド定義を追加した場合は、次のコマンドを使用して構文をチェックできます。

```
arrayd -c -f /usr/lib/array/arrayd.local
```

新しいコマンドが正しく動作するかどうかをテストする場合は、arrayd とその子プロセスによって生成される警告とエラー・メッセージを確認する必要があります。通常、デーモンからの stderr メッセージは表示されません。これらのメッセージは、次の手順で表示できます。

1. ノードでデーモンを強制終了します。
2. 該当するノードの特定のシェル・ウィンドウで、-n -v オプションを指定して arrayd を起動します。デーモンはバックグラウンドに移動せず、シェル端末に接続されたままになります。

   メモ: このモードでは、arrayd は動作可能になりますが、/etc/config/arrayd.options を参照しません。したがって、標準以外の設定ファイルの名前など、すべてのコマンド・ライン・オプションを明示的に指定する必要があります。

3. 同じノードまたは他のノードの別のシェル・ウィンドウから、ainfo および array コマンドを発行して、新しい設定データをテストします。診断出力が arrayd のシェル・ウィンドウに表示されます。
4. arrayd を強制終了して、デーモンとして(-n を指定せずに)再起動します。

手順 1、2、および 4 では、テスト・ノードが ainfo および array コマンドに応答しなくなる場合があります。そのため、アレイがテスト・モードであることをユーザに警告しておく必要があります。

アレイとマシンの設定

各 ARRAY エントリは、ユーザがアクセスできるアレイ・システムの名前と構成を指定します。使用中のアレイのすべてのノードで、少なくとも 1 つの ARRAY が定義されている必要があります。

メモ: ARRAY はキーワードです。

アレイ名とマシン名の指定

次に、ARRAY 定義の簡単な例を示します。

```
array simple
        machine congo
        machine niger
        machine nile
```

アレイ名 simple は、-a オプションでユーザが指定しなければならない値です。136ページの「共通のコマンド・オプションの概要」を参照してください。デフォルトのアレイ(ainfo dflt で報告されます)と

して、DESTINATION ARRAY ローカル・オプションに 1 つのアレイ名を指定する必要があります。ローカル・オプションは、156ページの「ローカル・オプションの設定」に示されています。

ローカルホストのみが含まれる me という名前のアレイを少なくとも 1 つ設定することをお薦めします。デフォルトの arrayd.conf ファイルでは、デフォルトの送信先アレイとして me アレイが定義されています。

ARRAY の MACHINE サブエントリは、-s オプションでユーザが指定できるノード名を定義します。これらの名前は、コマンド ainfo machines でも報告されます。

IP アドレスとポートの指定

次の例に示す単純な MACHINE エントリは、ホスト名がドメイン・ネーム・サービス (DNS: Domain Name Services) に対するマシンの名前と同じであるという想定に基づきます。指定したホスト名からマシンの IP アドレスを取得することができない場合は、HOSTNAME サブエントリを提供して、次のように完全修飾ドメイン名または IP アドレスのいずれかを指定する必要があります。

```
array simple
        machine congo
            hostname congo.engr.hitech.com
            port 8820
        machine niger
            hostname niger.engr.hitech.com
        machine nile
            hostname "198.206.32.85"
```

この例は、PORT サブエントリを使用して、特定のマシンの arrayd でデフォルトの 5434 以外のソケット番号を使用するよう指定する方法も示しています。

追加の属性の指定

ARRAY と MACHINE の両方で、名前付きの文字列値である属性を指定できます。これらの属性は Array Service では使用されませんが、ainfo で表示され、Array Service ライブラリ(158ページの「Array Service ライブラリ」を参照してください)を使用してプログラムに返すことができます。次に、属性の例を示します。

```
array simple
       array_attribute config_date="04/03/96"
       machine a_node
       machine_attribute aka="congo"
       hostname congo.engr.hitech.com
```

Tip: アレイ内の任意のノードから任意のアレイ名、マシン名、または属性文字列を取得するコードを作成できます。163ページの「データベースへの問い合わせ」を参照してください。

認証コードの設定

Array Service 3.5 では、任意の Array Service コマンドで要求できる単純な数値キーの認証 1 種類のみが提供されています。各ノードに対して、単一の認証コード番号を指定できます。ユーザは、該当するノードで入力するコマンド、またはそのノードに送信する任意のコマンドで、-s オプションを使用してコードを指定する必要があります(136ページの「共通のコマンド・オプションの概要」を参照してください)。

arshell コマンドは、呼出し元のユーザのユーザ ID を使用して別のマシン上でコマンドを実行するという点では、rsh と同様です。Array Service では認証コードを使用するため、rsh よりも多少安全です。

アレイ・コマンドの設定

ユーザは、arshell コマンドを使用して、単一のノードで任意のシステム・コマンドを呼出すことができます(144ページの「arshell の使用」を参照してください)。また、自動的に複数のノードに分散される MPI および PVM プログラムを起動することもできます。ただし、協調動作する複数のシステム・プログラムをすべてのノードで一度に起動するには、array コマンドを使用する方法しかありません。このコマンドはシステム・コマンドを受付けず、管理者が Array Service データベースに設定したコマンドのみ実行できます。

ユーザが必要とするコマンドの任意の集合を定義できます。単一のアレイ・ノードでのコマンドの実行方法は、完全に制御できます(異なるノードでは、異なる定義を使用できます)。コマンドは、単に標準のシステム・コマンドを呼出したり、スクリプトを起動するように定義できるので、任意に複雑にできます。

アレイ・コマンドの操作

ユーザが array コマンドを呼出すと、-s で指定した送信先のノードによってサブコマンドとその引数が処理されます。-l オプションを指定しないかぎり、サブコマンドとその引数は、デーモンにより、認識されているその他のすべてのアレイ・ノードにも配布されます(送信先のノードでは、ノードのサブセットだけが設定されている場合があります)。各ノードで、arrayd は、array サブコマンドと同じ名前を持つ COMMAND エントリを設定データベースで検索します。

次の例では、サブコマンド uptime が arrayd によってノード tokyo で処理されます。

```
array -s tokyo uptime
```

arrayd によってサブコマンドが無効であることが検出された場合は、ノード tokyo でデフォルトのアレイに設定されているすべてのノードにサブコマンドが配布されます。

uptime に対する COMMAND エントリは、次の形式で配布されます(ファイル /usr/lib/array/arrayd.conf で参照できます)。

```
command uptime          # Display uptime/load of all nodes in array
        invoke /usr/lib/array/auptime %LOCAL
```

INVOKE サブエントリは、このコマンドの実行方法を arrayd に指示します。この場合、シェル・スクリプト /usr/lib/array/auptime を実行し、1 つの引数(ローカル・ノードの名前)をデーモンに渡します。%LOCAL がノードの名前に置換され、すべてのノードでこのコマンドが実行されます。

コマンド定義構文の概要

Array Service 3.5 で配布されているコマンドの基本セット (/usr/lib/array/arrayd.conf) に見られるように、各 COMMAND エントリは、表7-7 に示すサブエントリを使用して定義されています(これらについては、arrayd.conf(4) のマン・ページで詳しく説明されています)。

表7-7 COMMAND 定義のサブエントリ

キーワード	後続の値の意味
COMMAND	ユーザが array に対して指定するコマンドの名前。
INVOKE	すべてのノードで実行されるシステム・コマンド。引数の値には、リテラル、ユーザが指定した引数、またはその他の置換値を使用できます。
MERGE	配布元ノードだけで実行されるシステム・コマンド。すべてのノードから出力のストリームを収集し、単一のストリームに組合わせます。
USER	INVOKE および MERGE コマンドを実行するユーザ ID。通常は、array を呼出したユーザとして実行されるよう、USER %USER として指定します。
GROUP	INVOKE および MERGE コマンドを実行するグループ名。通常は、array を呼出したユーザのグループで実行されるよう、GROUP %GROUP として指定します(groups(1) のマン・ページを参照してください)。

キーワード	後続の値の意味
PROJECT	INVOKE および MERGE コマンドを実行するプロジェクト。通常は、array を呼出したユーザのプロジェクトで実行されるよう、PROJECT %PROJECT として指定します（projects(5) のマン・ページを参照してください）。
OPTIONS	このコマンドを変更するためのさまざまなオプション。表7-9 を参照してください。

arrayd には実行パスが定義されていないため、INVOKE および MERGE で呼出すシステム・コマンドは、完全パス名として指定する必要があります。シェル・スクリプトと同様に、多くの場合、これらのシステム・コマンドは、数個のリテラル値と多くの置換文字列から構成されます。表7-8 に、サポートされている置換の概要を示します（詳細については、arrayd.conf(4) のマン・ページを参照してください）。

表7-8 COMMAND 定義で使用される置換

置換	置換値
%1..%9、%ARG(*n*)、%ALLARGS、%OPTARG(*n*)	ユーザのサブコマンドからの引数トークン。%OPTARG で指定した引数が省略されている場合、エラー・メッセージは生成されません。
%USER、%GROUP、%PROJECT	array を呼出したユーザの有効ユーザID、有効グループ ID、およびプロジェクト。
%REALUSER、%REALGROUP	array を呼出したユーザの実際のユーザ ID および実際のグループ ID。
%ASH	INVOKE または MERGE コマンドを実行する ASH。
%PID(*ash*)	指定された ASH の PID 値のリスト。%PID(%ASH) が一般的な使用法です。
%ARRAY	デフォルトのアレイ名、または -a オプションで指定したアレイ名。
%LOCAL	実行元ノードのホスト名。

置換	置換値
%ORIGIN	array コマンドが実行され、出力が表示されるノードの完全ドメイン名。
%OUTFILE	一時ファイルの名前のリスト。各ファイルには、特定のノードの INVOKE コマンドからの出力が含まれます（MERGE サブエントリ内でのみ有効）。

OPTIONS サブエントリを使用すると、コマンドの実行方法について多くの重要な変更を行うことができます。表7-9 に、これらの概要について説明します。

表7-9 COMMAND 定義のオプション

キーワード	コマンドに対する効果
LOCAL	他のノードに配布しない（実質的に、強制的に -l オプションを使用します）。
NEWSESSION	新しく作成される ASH で INVOKE コマンドを実行する。INVOKE 行の %ASH が新しい ASH です。MERGE コマンドは元の ASH で実行され、該当する行では、%ASH は古い ASH に置換されます。
SETRUID	USER サブエントリを使って実際のユーザ ID と有効ユーザ IDを設定する（通常、USER では有効 UID のみが設定されます）。
SETRGID	GROUP サブエントリを使って実際のグループ ID と有効グループ ID を設定する（通常、GROUP では有効 GID のみが設定されます）。
QUIET	MERGE サブエントリが指定されている場合を除き、INVOKE の出力を無視する。MERGE サブエントリが指定されている場合は、通常と同様に INVOKE の出力を MERGE に渡し、MERGE の出力を無視します。
NOWAIT	出力を無視し、プロセスが呼出されると同時に戻る。完了まで待機しません（MERGE サブエントリは無効です）。

ローカル・オプションの設定

LOCAL エントリは、arrayd 自体に対してオプションを指定します。表7-10 に、最も重要なオプションの概要を示します。

表7-10 LOCAL エントリのサブエントリ

サブエントリ	用途
DIR	arrayd の作業ディレクトリのパス名。これが、INVOKE および MERGE コマンドの初期の現在の作業ディレクトリになります。デフォルトは /usr/lib/array です。
DESTINATION ARRAY	ユーザが -a オプションを省略した場合に使用されるデフォルトのアレイの名前。1 つの ARRAY エントリのみが指定されている場合は、そのアレイがデフォルトの送信先になります。
USER、GROUP、PROJECT	COMMAND 定義で USER、GROUP、または PROJECT が省略されている場合の COMMAND 実行用のデフォルト値。
HOSTNAME	このノードで %LOCAL によって返される値。デフォルトはホスト名です。
PORT	arrayd が使用するソケット。

LOCAL USER、GROUP、および PROJECT の値を指定しなかった場合は、USER および GROUP のデフォルト値は「guest」になります。

HOSTNAME エントリは、hostname コマンドが ARRAY MACHINE エントリで指定されているノード名を返さない場合に必要になります。各ノードに固有の LOCAL HOSTNAME エントリを提供するには、各ノードに、少なくとも 1 種類の、個別の設定ファイルを用意する必要があります。

新しい Array コマンドの設計

コマンドの基本的な集合は、ファイル /usr/lib/array/arrayd.conf.template で配布されています。独自のコマンドを定義する前に、このファイルを入念に調べる必要があります。アレイ・システムのユーザが利用できる新しいコマンドを定義できます。

通常、新しいコマンドは、sh、csh、または Perl の構文で作成されたスクリプトを指定する INVOKE サブエントリで定義します。スクリプトの引数は、置換値を使用して設定します。スクリプトの実行条件は、USER、GROUP、PROJECT、および OPTIONS サブエントリを使用して設定します。単純なスクリプトを使用したコマンド定義の例については、145ページの「分散例について」を参照してください。

呼出されるスクリプト内に、引数を確認・検証したり、任意のコマンド・シーケンスを実行する任意の量の論理を作成できます。Perl によるスクリプトの例については、array ps コマンドによって呼出される /usr/lib/array/aps を参照してください。

メモ:Perl はソケット I/O をネイティブでサポートしているため、array コマンド用として特に適しています。少なくとも原理的には、array によって複数のインスタンスを起動し、ソケットを使用してデータを調整・交換する分散アプリケーションを Perl で作成できます。パフォーマンスの点では、入念に調整されている MPI および PVM ライブラリに及びませんが、開発は Perl の方が容易です。

設定ファイルはアレイ全体に分散されているため、管理者は分散アプリケーションも必要になります。次に、すべてのノードで一度に Array Service データベースを再初期化する分散コマンドの例を示します。各ノードで実行される /usr/lib/array/arrayd-reinit という名前のスクリプトは、次のようになります。

```
#!/bin/sh
# Script to reinitialize arrayd with a new configuration file
# Usage:  arrayd-reinit <hostname:new-config-file>
sleep 10      # Let old arrayd finish distributing
rcp $1 /usr/lib/array/
/etc/init.d/array restart
exit 0
```

このスクリプトでは、rcp を使用して、指定したファイル(多くの場合は arrayd.conf などの設定ファイル)を /usr/lib/array にコピーします(%USER に特権が付与されていない場合は失敗します)。続いて、arrayd(/etc/init.d/array を参照してください)を再起動して、設定ファイルを再読取りします。

コマンド定義は、以下のようになります。

```
command reinit
   invoke /usr/lib/array/arrayd-reinit %ORIGIN:%1
   user   %USER
   group  %GROUP
   options nowait   # Exit before restart occurs!
```

INVOKE サブエントリは、上に示した restart スクリプトを呼出します。NOWAIT オプションを使用して、デーモンがスクリプトの完了まで待機するのを防いでいます。これは、スクリプトによってデーモンが強制終了されてしまうためです。

Array Service ライブラリ

Array Service は、設定データベース、各ノードで実行されてサービスを提供するデーモン (arrayd)、および複数のユーザレベル・コマンドで構成されます。開発者は、Array Service ライブラリを通じて Array

Service の機能を利用することもできます。Array Service ライブラリは関数の集合で、これらを使用して、設定データベースに問い合わせたり、arrayd のサービスを呼出すことができます。

Array Service のコマンドについては、134ページの「Array Service コマンドの使用」で説明されています。Array Service の管理については、147ページの「アレイ設定について」と、それ以降の節で説明されています。これらの節では、Array Service ライブラリを理解する上で役立つ基礎知識が提供されています。

Array Service のプログラミング・インタフェースは、ヘッダ・ファイル /usr/include/arraysvcs.h で宣言されています。オブジェクト・コードは /usr/lib/libarray.so にあり、コンパイル時に -larray を指定することによってプログラムにインクルードします。このライブラリは、IRIX では o32、n32、および 64 ビット・バージョン、Linux では IA-64 バージョンでそれぞれ配布されています（すべてインストールする必要はありません）。

ライブラリ関数は、以下のカテゴリにグループ化できます。

- ローカルまたはその他のノードの Array Service デーモンに接続したり、arrayd オプションを取得および設定するための関数。
- アレイ、ノード、およびアレイとノードの属性をリストした Array Service 設定データベースに問い合わせるための関数。
- アレイ・セッション・ハンドル (ASH: Array Session Handle) の割当て、アクティブな ASH の照会、および PID と ASH の関係の変更を行うための関数。
- array コマンドに対してコマンドを実行するための関数（153ページの「アレイ・コマンドの操作」を参照してください）。
- アレイ・ノードで任意のユーザ・コマンドを実行するための関数。

以下の節では、これらの関数について詳しく説明していきます。

データ構造体

Array Service 関数は、arraysvcs.h に記述されている多くのデータ構造体と連携します。一般的に、各データ構造体は、関数の結果として構造体へのポインタを返すある特定の関数によって割当てられます。返された構造体をコードで使用し、他の関数の引数として渡すことができます。

コード中で構造体の使用を完了する場合は、そのタイプの構造体を解放する特定の関数を呼出すことが期待されます。コードで各構造体を解放しないと、メモリ・リークが発生します。

表7-11 に、データ構造体とそれらの内容の概要を示します。

表7-11 Array Service のデータ構造体

構造体	内容	解放する関数
asarray_t	アレイの名前と属性。	`asfreearray()`
asarraylist_t	*asarray_t* 構造体のリスト。	`asfreearraylist()`
asashlist_t	ASH 値のリスト。	`asfreeashlist()`
ascmdrslt_t	一時ファイルとソケット番号を含む、特定のノードでの `array` コマンドの実行出力を記述する。	リストの一部として解放される。
ascmdrsltlist_t	コマンド結果のリスト。`array` コマンドが実行されたノードごとに 1 つの *ascmdrslt_t* があります。	`asfreecmdrsltlist()`
asmachine_t	特定のノードに関する設定データ(マシン名および属性)。	リストの一部として解放される。
asmachinelist_t	*asmachine_t* 構造体のリスト。照会されたアレイ内のマシンごとに 1 つあります。	`asfreemachinelist()`
aspidlist_t	PID 値のリスト。	`asfreepidlist()`

エラー・メッセージの表記規則

Array Service ライブラリの関数では、エラー・リターン・コードに複雑な表記規則があります。表7-12 に、この表記規則に関連するマン・ページを示します。

表7-12 エラー・メッセージ関数

関数	動作
`aserrorcode`(3X)	エラー・コードの表記規則、およびエラー・コードからサブフィールドを抽出する場合に使用するいくつかのマクロ関数を説明する。

関数	動作
asmakeerror(3X)	コンポーネント部品からエラー・コード値を構成する。
asstrerror(3X)	指定したエラー・コード値の説明文字列を返す。
asperror(3X)	指定したヘッダ文字列と共に説明文字列を stderr に出力する。

一般的に、各関数は、タイプが *aserror_t*（*int* である必要はありません）の *aserrorcode* グローバル構造体の値を設定します。エラー・コードは、以下の部品で構造化された値です。

- *aserrno* - sys/errno.h で宣言されているものと同様の一般的なエラー番号です。
- *aserrwhy* - エラーの原因を記述します。
- *aserrwhy* - エラーを検出したコンポーネントを記述します。
- *aserrextra* - 追加の情報が記述されている場合があります。

これらのサブフィールドを *aserrorcode* グローバル構造体から抽出するためのマクロ関数が提供されています。

Array Service デーモンへの接続

表7-13 に示す関数を使用して、プログラムが実行されるノードと、同じまたは別のノードの arrayd のインスタンスの間の接続を開きます。

表7-13 Array Service デーモンに接続するための関数

関数	動作
asopenserver(3X)	指定したノードの arrayd との論理的な接続を確立し、他の関数で使用できるよう、接続を表すトークンを返す。
ascloseserver(3X)	asopenserver() で作成された arrayd 接続を閉じる。

関数	動作
asgetserveropt(3X)	arrayd のインスタンスによって現在使用中のローカル・オプションを返す。
asdfltserveropt(3X)	arrayd のインスタンスで有効なデフォルトのオプションを返す。
assetserveropt(3X)	arrayd のインスタンスに対して新しいオプションを設定する。

重要な関数は asopenserver() です。この関数は、ノード名を文字列として取ります(この文字列はユーザが -s オプションで指定します。136ページの「共通のコマンド・オプションの概要」を参照してください)。また、オプションでソケット番号を取り、arrayd のデフォルトのソケット番号をオーバーライドします。この関数は、arrayd の指定したインスタンスへのソケット接続を開きます。返されるトークン(*asserver_t* タイプ)はその接続を表し、他の関数に渡されます。

サーバ・オプションを取得および設定するための関数を使用して、表7-14 に示す設定済みオプションを変更できます。これらのオプションを設定することは、Array Service コマンドでコマンド・ライン・オプションを渡すのと同等の処理をプログラムによって行うことです(147ページの「アレイ設定について」、および134ページの「Array Service コマンドの使用」を参照してください)。

表7-14 関数による照会または変更が可能なサーバ・オプション

定数	変更可能かどうか	意味
AS_SO_TIMEOUT	可能	このサーバに対する要求のタイムアウト周期
AS_SO_CTIMEOUT	可能	このサーバへの接続のタイムアウト周期
AS_SO_FORWARD	可能	Array Service 要求をローカル arrayd を通じて転送するか、それとも -F オプションを使用して直接送信するか
AS_SO_LOCALKEY	可能	ローカル認証キー(-Kl コマンド・オプション)
AS_SO_REMOTEKEY	可能	リモート認証キー(-Kr コマンド・オプション)

定数	変更可能かどうか	意味
AS_SO_PORTNUM	不可	デフォルトのソケット番号（デフォルト・オプションのみ）
AS_SO_HOSTNAME	不可	この接続のホスト名

データベースへの問い合わせ

表7-15 に概要を示す関数を使用して、指定したノードの arrayd が使用する設定データベースに問い合わせます（147ページの「アレイ設定について」を参照してください）。

表7-15 設定を問い合わせるための関数

関数	動作
asgetdfltarray(3X)	指定したサーバで認識されているデフォルトのアレイのアレイ名とすべての属性文字列を *asarray_t* 構造体で返す。
aslistarrays(3X)	指定したサーバから、すべてのアレイの名前を属性文字列と共に *asarraylist_t* 構造体として返す。
aslistmachines(3X)	指定したサーバから、すべてのマシンの名前を属性文字列と共に *asmachinelist_t* 構造体として返す。
asgetattr(3X)	属性文字列のリスト内で特定の属性名を検索し、その値を返す。

これらの関数を使用して、開いた arrayd インスタンスで認識されている任意のアレイ名、ノード名、または属性を抽出できます。

アレイ・サービス・ハンドルの管理

表7-16 に概要を示す関数を使用して、ASH 値の作成や問い合わせを行います。

表7-16 アレイ・サービス・ハンドルを管理するための関数

関数	動作
asallocash(3X)	新しい ASH 値を割当てる。値の作成のみで、プロセスへの割当ては行われません。
aspidsinash(3X)	指定したサーバーで ASH に関連付けられている PID 値のリストを *aspidlist_t* 構造体として返す。
asashofpid(3X)	指定した PID に関連付けられている ASH を返す。
setash(2)	呼出し元プロセスの ASH を返す。

asallocash() 関数は ainfo newash コマンドと似ています(143ページの「アレイ・セッション・ハンドル (ASH: Array Session Handles) について」を参照してください)。setash() システム関数を使用して現在のプロセスの ASH を変更できるのは、root 特権を持つプログラムのみです。非特権プロセスは、新しい ASH 値を作成することはできますが、それらの ASH を変更することはできません。

表7-17 に概要を示す関数を使用して、指定したノードのアクティブな ASH 値を列挙します。どの場合でも、ASH 値のリストは *asashlist_t* 構造体で返されます。

表7-17 ASH を問い合わせるための関数

関数	動作
aslistashs(3X)	指定したサーバを経由して、指定したアレイの 1 つまたはすべてのノードからアクティブな ASH 値を返す。
aslistashs_array(3X)	アレイからアクティブな ASH 値を名前別に返す。
aslistashs_server(3X)	指定したサーバ・ノードで認識されているアクティブな ASH 値を返す。

関数	動作
aslistashs_local(3X)	ローカル・ノードのアクティブな ASH 値を返す。
asashisglobal(3X)	ASH がグローバルかどうかを確認するためのテストを実行する。

array コマンドの実行

ascommand() 関数は、array コマンドと同等の処理をプログラムによって行います(153ページの「アレイ・コマンドの操作」、および array(1) のマン・ページを参照してください)。このコマンドには多くのオプションがあり、独立した 3 つのモードでコマンドを実行する場合に使用できます。

実行するコマンドは、以下のフィールドが含まれる ascmdreq_t 構造体で準備されている必要があります。

```
typedef struct ascmdreq {
   char *array;            /* Name of target array */
   int flags;              /* Option flags */
   int numargs;            /* Number of arguments */
   char **args;            /* Cmd arguments (ala argv) */
   int ioflags;            /* I/O flags for interactive commands */
   char rsrvd[100];        /* reserved for expansion: init to 0's */
} ascmdreq_t;
```

コマンドを実行するには、プログラムでこの構造体を準備する必要があります。オプション・フラグを使用することで、array のコマンド・ライン・オプションと同じ制御が可能です。

コマンドの結果は、*ascmdrsltlist_t* 構造体として返されます。これは、*ascmdrslt_t* 構造体のベクトルで、コマンドが実行される各ノードに 1 つずつあります。各 *ascmdrslt_t* には、以下のフィールドが含まれます。

```
typedef struct ascmdrslt {
    char     *machine;  /* Name of responding machine */
    ash_t    ash;       /* ASH of running command */
    int      flags;     /* Result flags */
    aserror_t error;    /* Error code for this command */
    int      status;    /* Exit status */
    char     *outfile;  /* Name of output file */
    int      ioflags;   /* I/O connections (see ascmdreq_t) */
    int      stdinfd;   /* File descriptor for command's stdin */
    int      stdoutfd;  /* File descriptor for command's stdout */
```

```
    int       stderrfd;  /* File descriptor for command's stderr */
    int       signalfd;  /* File descriptor for sending signals */
} ascmdrslt_t;
```

フィールド machine、ash、flags、error、および status は、該当するマシンでのコマンドの実行結果を反映します。その他のフィールドは、実行のモードに応じて変わります。

通常のバッチ実行

コマンドを通常の方法(コマンドが完了するまで待機し、出力を収集する方法)で実行する場合は、ASCMDREQ_NOWAIT または ASCMDREQ_INTERACTIVE のどちらもコマンド・オプション・フラグに設定しません。

すべてのノードでコマンドが完了すると、ascommand() から制御が戻ります。ASCMDREQ_OUTPUT フラグが指定されていて、コマンド定義に MERGE サブエントリ(154ページの「コマンド定義構文の概要」を参照してください)が指定されていない場合、outfile 結果フィールドには、特定のノードの出力ストリームを含む一時ファイルの名前が含まれます。

MERGE サブエントリを使用してコマンドが実装されている場合は、呼出されるノードの数に関係なく、出力ファイルは 1 つだけです。この場合、返されるリストには 1 つの ascmdrslt_t 構造体のみが含まれます。ここには ASCMDRSLT_MERGED および ASCMDREQ_OUTPUT フラグが含まれ、outfile 結果フィールドには、結合された出力を含む一時ファイルの名前が含まれます。

即時実行

コマンドに有益な出力がなく、呼出し元プログラムと同時に実行する必要がある場合は、ASCMDREQ_NOWAIT オプションを指定します。この場合、出力を使用するために待機しているプログラムがないため、出力を収集することはできません。制御は、コマンドが配布されると同時に戻ります。結果構造体にはコマンドの結果は反映されず、コマンドの起動を試みた結果のみが反映されます。

対話型実行

プログラムがすべてのノードのコマンド・プロセスの入力および出力ストリームと直接対話を持つような方法で、コマンドを起動できます。この場合は、プログラムで入力を提供して、ほぼリアルタイムに出力を検査できます。

対話型実行を確立するには、コマンド・オプション・フラグで ASCMDREQ_INTERACTIVE を指定します。また、ioflags フィールドで以下の 1 つまたは複数のフラグも設定します。

ASCMDIO_STDIN	コマンドの stdin にアタッチされているソケットを要求します。
ASCMDIO_STDOUT	コマンドの stdout にアタッチされているソケットを要求します。
ASCMDIO_STDERR	コマンドの stderr にアタッチされているソケットを要求します。

ASCMDIO_SIGNAL　　シグナルの配信に使用できるソケットを要求します。

ASCMDREQ_NOWAIT と同様に、制御は、コマンドが配布されると同時に戻ります。各結果構造体には、該当するノードのコマンド・プロセスに対する要求されたソケットのファイル記述子が含まれます。

プログラムは、あるノードの *stdinfd* ファイル記述子にデータを書込んで、そのノードの stdin ストリームにデータを送信します。また、*stdoutfd* ファイル記述子からデータを読取って、ノードの出力ストリームを読取ります。

一般的に、いつソケットの 1 つの使用準備ができたかを確認するには、select() または poll() システム関数を使用します。fork() を使用して 1 つまたは複数のサブプロセスを起動するよう選択して、各ノードのソケットに対する I/O を処理できます。select(2)、poll(2)、および sproc(2) のマン・ページを参照してください（sproc() を使用してサブプロセスを作成することもできますが、libarray はスレッドセーフではないため、共有グループ内の 1 つのプロセスからのみ使用しなければならない点に注意してください）。

ユーザ・コマンドの実行

asrcmd() 関数は、指定したノードでプログラムから任意のユーザ・コマンド文字列を開始できます。これは、標準の rcmd() 関数と異なり root 特権を必要としないリモート実行に対して強力な機能を提供します（asrcmd(3) と rcmd(3) のマン・ページを比較してください）。

asrcmd() 関数は、以下を指定する引数を取ります。

- asopenserver() で返される、使用するアレイ・ノード（161ページの「Array Service デーモンへの接続」を参照してください）。
- リモート・ノードで使用するユーザ名。
- 実行するコマンド・ライン。

rcmd() と同様に、戻り値は、実行元コマンドの標準入力および出力ストリームを表すソケットです。オプションで、標準エラー・ストリームの独立したソケットを取得できます。

リソース管理用のプログラミング・ガイド

この付録には、ジョブ制限、ユーザ制限データベース (ULDB: User Limits DataBase)、および cpuset システム・プログラミングに関する情報を記載しています。

この付録は、次の節で構成されています。

ジョブ制限のアプリケーション・プログラミング・インタフェース (API: Application Programming Interface)

この節では、API 関数へのライブラリ・インタフェースで使用されるデータ・タイプと関数コールについて説明します。

データ・タイプ

ここでは、API 関数へのライブラリ・インタフェースで使用される固有のデータ・タイプについて説明します。

すべての制限値は、/usr/include/sys/resource.h システム・インクルード・ファイル内でプロセス制限に対して定義された rlimit 構造体によって指定されます。

```
typedef unsigned long rlim_t;
   struct rlimit_t {
           rlim_t         rlim_cur;
           rlim_t         rlim_max;
   };
```

ジョブ ID は、符号付き 64 ビット値として定義されます。これは、アプリケーションによって非透過的に処理されます。jid_t は、システム・インクルード・ファイル sys/types.h で定義されています。

```
typedef int64_t jid_t;
```

メモ: ジョブ制限の値 (rlim_t) は、n32 バイナリと n64 バイナリの両方で 64 ビットです。そのため、n32 バイナリで 64 ビット制限を設定できます。rlim_t は o32 バイナリでは 32 ビットなので、o32 バイナリで 64 ビット制限を設定することはできません。IRIX では、o32、n64、および n32 の 3 つのアプリケーション・バイナリ・インタフェース (ABI: Application Binary Interface) をサポートします（ABI についての詳細は、abi(5) のマン・ページを参照してください）。

rlimit_* の値についての詳細は、2 ページの「systune を使用したプロセス制限の表示と設定」および 18 ページの「showlimits」を参照してください。

関数コール

ジョブ制限の API は、libc.a ライブラリで定義されている一連の関数によって定義されています。各関数は、syssgi(2) システム・インタフェースを呼出して必要な操作を行います。関数のプロトタイプは、システム・インクルード・ファイル /usr/include/sys/resource.h にあります。

getjlimit および setjlimit

getjlimit 関数は各種のシステム・リソースの利用に関する制限をジョブごとに取得し、setjlimit 関数はこれらの制限を設定します。

```
#include <sys/resource.h>
int getjlimit(jid_t jid, int resource, struct rlimit *rlp)
int setjlimit(jid_t jid, int resource, struct rlimit *rlp)
```

詳細については、getjlimit(2) のマン・ページを参照してください。

getjusage

getjusage 関数は、指定されたジョブ ID のリソース使用量の値を取得します。

```
#include <sys/resource.h>
int getjusage(jid_t jid, int resource, struct jobrusage *up)
```

jid パラメータがゼロの場合は、現在のジョブの使用状況の値が返されます。*jid* がゼロ以外の場合は、使用状況の値が取得されるジョブのジョブ ID を表します。*resource* パラメータは、使用状況の値が返されるリソースを指定します。使用可能な値は、sys/resource.h ファイル内の JLIMIT_xxx マクロから取得されます。たとえば、JLIMIT_CPU マクロは CPU 時間用です。up パラメータは、使用状況の値が返されるユーザ・プログラム内の rusage 構造体を示します。

詳細については、getjusage(2) のマン・ページを参照してください。

getjid

getjid 関数は、現在のプロセスに関連付けられているジョブ ID を返します。

```
#include <sys/resource.h>
jid_t getjid(void);
```

詳細については、getjid(2) のマン・ページを参照してください。

killjob

killjob 関数は、指定したジョブ ID に属するすべてのプロセスにシグナルを送信します。

```
#include <sys/resource.h>
int killjob(jid_t jid, int signal)
```

詳細については、killjob(2) のマン・ページを参照してください。

jlimit_startjob

jlimit_startjob 関数は、ジョブを新規作成し、ジョブ制限を ULDB 内の制限値に設定します。

jlimit_startjob 関数の形式を以下に示します。

```
#include <sys/resource.h>
jid_t    jlimit_startjob(char *username, uid_t uid, char *domainname);
```

詳細については、jlimit_startjob(2) のマン・ページを参照してください。

makenewjob

makenewjob 関数は、新しいジョブ・コンテナを作成します。

```
#include <sys/resource.h>
jid_t makenewjob(uid_t user, jid_t rjid)
```

詳細については、makenewjob(2) のマン・ページを参照してください。

setjusage

setjusage 関数は、指定されたジョブ ID のリソース使用量の値を更新します。

setjusage 関数の形式を以下に示します。

```
#include <sys/resource.h>

int setjusage(jid_t jid, int resource, struct jobrusage *up)
```

setjusage 関数は、指定されたジョブ ID のリソース使用量の値を更新します。jid パラメータがゼロの場合は、現在のジョブの使用量の値が更新されます。jid がゼロ以外の場合は、使用量の値が更新されるジョブのジョブ ID を表します。resource パラメータは、使用量の値が更新されるリソースを指定します。使用可能な値は、sys/resource.h ファイル内の JLIMIT_xxx マクロから取得されます。たとえば、JLIMIT_CPU マクロは CPU 時間用です。up パラメータは、使用量の値が格納されるユーザ・プログラム内の jobrusage 構造体を示します。

setjusage を使用してリソース使用量の値を更新できるようにするには、ジョブが、指定されたリソースに対する制限の累積および違反時の動作を無視する必要があります。特定のリソース制限を無視するかどうかは、ジョブの作成時に、以下のシステム調整可能パラメータの値に基づいて決定されます。

[jlimit_numproc_ign]	ジョブ内のプロセスの数に対する制限の累積および違反時の動作を無視します。
[jlimit_pmem_ign]	ジョブの物理メモリ使用量総計の累積および違反時の動作を無視します。
[jlimit_pthread_ign]	ジョブ内の pthread の数の累積および違反時の動作を無視します。
[jlimit_nofile_ign]	ジョブ内の開いているファイルの数の累積および違反時の動作を無視します。
[jlimit_rss_ign]	ジョブの常駐セット・サイズ総計の累積および違反時の動作を無視します。
[jlimit_vmem_ign]	ジョブの仮想メモリ・サイズ総計の累積および違反時の動作を無視します。
[jlimit_data_ign]	ジョブのデータ・セグメント・サイズ総計の累積および違反時の動作を無視します。
[jlimit_cpu_ign]	ジョブの CPU 時間使用量の累積および違反時の動作を無視します。

これらの調整可能パラメータの値は、ランタイム時に変更できます。デフォルトでは、リソース使用量の制限の累積および違反時の動作が無視されないように設定されています。これらの値をランタイム時に変更した場合、パラメータの変更後に作成したジョブの動作のみが変更されます。パラメータの変更以前に存在していたジョブは、リソース使用量に対するジョブ制限の累積および違反時の動作に関しては、変更されないまま動作し続けます。

システム調整可能パラメータの詳細については、systune(1M) のマン・ページを参照してください。

setjusage の使用を試みるプロセスには、CAP_PROC_MGT capability が必要です。プロセスの操作権に対する細かな調整を可能にする capability についての詳細は、capability(4) および capabilities(4) のマン・ページを参照してください。

詳細については、setjusage(2) のマン・ページを参照してください。

setwaitjobpid

setwaitjobpid 関数は、指定の pid が waitjob 関数を呼出すのを待つようにジョブを設定します。

setwaitjobpid 関数の形式を以下に示します。

```
#include <sys/resource.h>
int setwaitjobpid(jid_t rjid, pid_t wpid)
```

詳細については、setwaitjobpid(2) のマン・ページを参照してください。

waitjob

waitjob 関数は、setwaitjobpid 引数を使用して待機するよう設定された停止ジョブに関する情報を取得します。

waitjob 関数の形式を以下に示します。

```
#include <sys/resource.h>
jid_t waitjob(job_info_t *jobinfo)
```

詳細については、waitjob(2) のマン・ページを参照してください。

エラー・メッセージ

エラー・メッセージについては、該当するマン・ページおよび 27 ページの「エラー・メッセージ」を参照してください。

ULDB の API

この節では、ULDB へのライブラリ・インタフェースで使用されるデータ・タイプと関数コールについて説明します。

データ・タイプ

ここでは、ユーザ制限情報へのライブラリ・インタフェースで使用される固有のデータ・タイプを定義します。ULDB 定義はすべてインクルード・ファイル /usr/include/uldb.h にあります。

2 進の制限値は、以下のように符号なし 64 ビット値として保持されます。

```
typedef rlim_t uldb_limit_t;
```

uldb_namelist_t

uldb_namelist_t データ・タイプは、制限名、ドメイン名などの名前リストを保持するために使用します。namelist 構造体には、アイテムの個数と名前ポインタのリストへのポインタが含まれます。uldb_namelist_t データ・タイプを以下に示します。

```
typedef struct uldb_namelist_s {
      int uldb_nitems,                # number of names in the list
      char **uldb_names               # list of name pointers
   } uldb_namelist_t;
```

uldb_limitlist_t

uldb_limitlist_t データ・タイプは、2 進の制限値のリストを保持するために使用します。制限リスト構造体には、アイテムの個数と制限値の配列へのポインタが含まれます。uldb_limitlist_t データ・タイプを以下に示します。

```
typedef struct uldb_limitlist_s {
      int uldb_nitems,                # number of limit values in the list
      uldb_limit_t *uldb_limits       # list of limit pointers
   } uldb_limitlist_t;
```

関数コール

ここでは、ユーザ制限情報へのライブラリ・インタフェースで使用される関数コールを定義します。

制限値を取得する関数を以下に示します。

- uldb_get_limit_values
- uldb_get_value_units
- uldb_get_limit_names
- uldb_get_domain_names

uldb_get_limit_values

uldb_get_limit_values 関数は、ドメインやユーザの制限値のセットを取得します。指定したユーザに対する明示的なエントリがない場合は、ドメインのデフォルト値を返します。要求した制限のセットは、uldb_namelist_t 構造体として取得されます。返された制限リストのポインタは、malloc ルーチン・コールで作成された新規の uldb_limitlist_t 構造体を参照します。不要になった構造体はアプリケーション側で解放する必要があります。返された uldb_limitlist_t 構造体に格納される制限値の順序は、入力した uldb_namelist_t 構造体の制限名の順序に対応します。ユーザ名が NULL の場合、ユーザ制限の代わりにドメインの制限のリストが返されます。

uldb_get_limit_values の例を以下に示します。

```
#include include/uldb.h
uldb_limitlist_t *        # returns pointer to limit list or NULL if error
   uldb_get_limit_values (         #
      char *domain_name,           # pointer to domain name
      char *user_name,             # name of user
      uldb_namelist_t *limits);    # namelist containing limit names
```

uldb_get_value_units

uldb_get_value_units 関数は、指定した制限のリストの修飾値や単位を含む制限リスト構造体を返します。使用可能な修飾値は、ヘッダ・ファイル uldb.h に定義されています。返される名前リストは、malloc ルーチン・コールで作成された uldb_namelist_t 構造体に格納されます。不要になった構造体はアプリケーション側で解放する必要があります。

uldb_get_value_units の例を以下に示します。

```
#include <include/uldb.h>
uldb_limitlist_t *        # returns pointer to limit list or NULL if error
   uldb_get_value_units (          #
      char *domain_name,           # pointer to domain name
      char *user_name,             # name of user
      uldb_namelist_t *limits);    # namelist containing limit names
```

uldb_get_limit_names

uldb_get_limit_names 関数は、ドメインに対して定義されたすべての制限名のリストを取得します。返される名前リストは、malloc ルーチン・コールで作成された uldb_namelist_t 構造体に格納されます。不要になった構造体はアプリケーション側で解放する必要があります。

uldb_get_limit_names の例を以下に示します。

```
#include <include/uldb.h>
   uldb_namelist_t *               # returns pointer to name list or NULL if error
   uldb_get_limit_names (
       char *domain_name);         # pointer to domain name
```

uldb_get_domain_names

uldb_get_domain_names 関数は、ULDB 内に定義されたドメイン名の完全なリストを取得します。返される名前リストは、malloc ルーチン・コールで作成された uldb_namelist_t 構造体に格納されます。不要になった構造体はアプリケーション側で解放する必要があります。

```
#include <include/uldb.h>
uldb_namelist_t *               # returns pointer to name list or NULL if error
   uldb_get_domain_names (
       void);
```

メモリを管理する関数を以下に示します。

- uldb_free_namelist
- uldb_free_limit_list

uldb_free_namelist

uldb_free_namelist 関数は、namelist 構造体とそのコンポーネントをすべて削除します。

uldb_free_namelist の例を以下に示します。

```
#include <include/uldb.h>
void                                # returns 0 if okay, -1 on error
   uldb_free_namelist (               #
      uldb_namelist_t *names);      # pointer to namelist to be freed
```

uldb_free_limit_list

uldb_free_limit_list 関数は、制限リスト構造体とそのコンポーネントをすべて削除します。

uldb_free_limit_list の例を以下に示します。

```
#include <include/uldb.h>
void                                # returns 0 if okay, -1 on error
   uldb_free_limit_list (           #
```

```
uldb_limit_list_t *limits);  # pointer to limit list to be freed
```

エラー・メッセージ

エラー・メッセージについては、uldb_get_limit_values(3c) と jlimit_startjob(3c) のマン・ページ、または 27 ページの「エラー・メッセージ」を参照してください。

cpuset システムの API

cpuset ライブラリが提供するインタフェースにより、プログラムは、cpuset の作成・破棄、既存の cpuset に関する情報の取得、既存の cpuset に関連付けられているプロパティの情報の取得、およびプロセスとその子プロセスの cpuset へのアタッチを行うことができます。

cpuset ライブラリを使用するには、作成する cpuset に対するパーミッション・ファイルを定義する必要があります。パーミッション・ファイルは、空のファイルでもかまいません。このファイルのファイル・パーミッションにより、cpuset へのアクセスが定義されます。パーミッションをチェックする必要がある場合、ファイルの現在のパーミッションが使用されます。したがって、特定の cpuset へのアクセスは、cpuset を破棄して再作成しなくても、単にアクセス・パーミッションを変更するだけで変更できます。ユーザは、読込みパーミッションがあれば cpuset に関する情報を取得し、実行パーミッションがあれば cpuset にプロセスをアタッチできます。

cpuset ライブラリは、N32 ダイナミック・シェア・オブジェクト (DSO: Dynamic Shared Object) ライブラリとして提供されます。ライブラリ・ファイルは libcpuset.so で、通常はディレクトリ /lib32 にあります。ライブラリを使用するには、/usr/include にある cpuset.h ヘッダ・ファイルをインクルードする必要があります。cpuset ライブラリ内の関数インタフェースは、ライブラリに新しいインタフェースが追加されたときに下位互換性を保つため、オプションのインタフェースとして宣言されています。

メモ: cpuset ライブラリは、IRIX 6.5.8 以降のリリースのみで使用できます。

この DSO とそのインタフェースがある場合はこれらを使用し、ない場合は実行を続けるプログラムをコンパイルし、実行できます。この場合、libcpuset.so の代用ライブラリが利用可能になっている必要があります。代用ライブラリの作成方法については、cpuset(5) のマン・ページを参照してください。DSO についての詳細は、DSO(5) のマン・ページを参照してください。

cpuset ライブラリ内の関数インタフェースを以下に示します。

関数インタフェース	説明
cpusetCreate(3x)	cpuset を作成します。
cpusetAttach(3x)	現在のプロセスを cpuset にアタッチします。

`cpusetAttachPID`(3x)	特定のプロセスを cpuset にアタッチします。
`cpusetDetachAll`(3x)	cpuset からすべてのスレッドをデタッチします。
`cpusetDetachPID`(3x)	特定のプロセスを cpuset からデタッチします。
`cpusetDestroy`(3x)	cpuset を破棄します。
`cpusetGetCPUCount`(3x)	システム上で構成されている CPU の数を取得します。
`cpusetGetCPUList`(3x)	cpuset に割当てられたすべての CPU のリストを取得します。
`cpusetGetName`(3x)	プロセスがアタッチされている cpuset の名前を取得します。
`cpusetGetNameList`(3x)	定義されたすべての cpuset の名前のリストを取得します。
`cpusetGetPIDList`(3x)	cpuset に割当てられたすべての PID のリストを取得します。
`cpusetGetProperties`(3x)	cpuset に関連付けられているさまざまなプロパティを取得します。
`cpusetMove`(3x)	プロセス(PID または ASH によって識別される)を、指定した cpuset から一時的に移動します。
`cpusetMoveMigrate`(3x)	特定のプロセス(PID または ASH によって識別される)とそれに関連付けられているメモリを、特定の cpuset から別の cpuset に移動します。
`cpusetAllocQueueDef`(3x)	`cpuset_QueueDef_t` 構造体を割当てます。
`cpusetFreeQueueDef`(3x)	`cpuset_QueueDef_t` 構造体で使用されているメモリを解放します。
`cpusetFreeCPUList`(3x)	`cpuset_CPUList_t` 構造体で使用されているメモリを解放します。
`cpusetFreeNameList`(3x)	`cpuset_NameList_t` 構造体で使用されているメモリを解放します。
`cpusetFreePIDList`(3x)	`cpuset_PIDList_t` 構造体で使用されているメモリを解放します。
`cpusetFreeProperties`(3x)	`cpuset_Properties_t` 構造体で使用されているメモリを解放します。

管理用関数

この節では、以下の cpuset システムのライブラリ関数のマン・ページを記載します。

`cpusetCreate`(3x)	cpuset を作成します。
`cpusetAttach`(3x)	現在のプロセスを cpuset にアタッチします。
`cpusetAttachPID`(3x)	特定のプロセスを cpuset にアタッチします。
`cpusetDetachPID`(3x)	特定のプロセスを cpuset からデタッチします。
`cpusetDetachAll`(3x)	cpuset からすべてのスレッドをデタッチします。
`cpusetDestroy`(3x)	cpuset を破棄します。
`cpusetMove`(3x)	プロセス(PID、JID または ASH によって識別される)を、指定した cpuset から一時的に移動します。
`cpusetMoveMigrate`(3x)	特定のプロセス(PID、JID または ASH によって識別される)とそれに関連付けられているメモリを、特定の cpuset から別の cpuset に移動します。

cpusetCreate(3x)

名前

cpusetCreate - cpuset を作成します。

形式

```
#include <cpuset.h>
int cpusetCreate(char *qname, cpuset_QueueDef_t *qdef);
```

説明

cpusetCreate 関数を使用して、cpuset キューを作成します。root ユーザ ID で実行されているプロセスだけが、cpuset キューを作成できます。

qname 引数は、新しい cpuset に割当てられる名前です。cpuset の名前は、3〜8 文字の文字列にする必要があります。1〜2 文字のキュー名は、IRIX オペレーティング・システム用に予約されています。

qdef 引数は、作成されるキューの属性を定義する cpuset_QueueDef_t 構造体（cpuset.h インクルード・ファイルで定義）へのポインタです。cpuset_QueueDef_t のメモリは cpusetAllocQueueDef(3x) を使用して割当てられ、cpusetFreeQueueDef(3x) を使用して解放されます。cpuset_QueueDef_t 構造体は、以下のように定義されます。

```
typedef struct {
                    int                 flags;
                    char                *permfile;
                    cpuset_CPUList_t    *cpu;
                } cpuset_QueueDef_t;
```

flags メンバーを使用して、cpuset キューのさまざまな制御オプションを指定します。ビット単位の排他的論理和の演算子を以下の 0 個以上の値に適用して作成します。

CPUSET_CPU_EXCLUSIVE
: cpuset を制限付きと定義します。cpuset キューにアタッチされたスレッドだけが、cpuset に含まれる CPU で実行可能です（アタッチされたスレッドの子孫はアタッチを継承）。

CPUSET_MEMORY_LOCAL
: cpuset に割当てられたスレッドは、その cpuset 内のノードからのみメモリを割当てようとします。cpuset の外部からのメモリ割当ては、cpuset 内で空きメモリが利用できない場合にのみ発生します。cpuset の外部で実行しているスレッドへのメモリ割当てには制限は加えられません。

CPUSET_MEMORY_EXCLUSIVE	cpuset に割当てられたスレッドは、その cpuset 内のノードからのみメモリを割当てようとします。cpuset の外部からのメモリ割当ては、cpuset 内で空きメモリが利用できない場合にのみ発生します。cpuset の外部でメモリが利用できない場合を除き、cpuset に割当てられないスレッドは、その cpuset 内のメモリを使用しません。cpuset の作成時に、実行中のスレッドにメモリがすでに割当てられている場合は、このメモリを明示的に移動する試みは行われません。ページ移動が可能な場合、ページへの参照のほとんどが非ローカルであることがシステムで検出されると、そのページは移動されます。
CPUSET_MEMORY_KERNEL_AVOID	カーネルは、この cpuset に含まれるノードからのメモリ割当てを避けようとします。カーネルのメモリ要求が、この cpuset の外部では満たされない場合、このオプションは無視され、cpuset 内部でのメモリ割当てが発生します（この回避動作は現在、保護されたノードからバッファ・キャッシュを隔離するだけです）。
CPUSET_MEMORY_MANDATORY	カーネルは、すべてのメモリ割当てをこの cpuset に含まれるノードに限定します。メモリ要求が満たされない場合、メモリが使用可能になるまで割当てプロセスはスリープします。これ以上のメモリを割当てられない場合、プロセスは強制終了されます。以下のポリシーを参照してください。
CPUSET_POLICY_PAGE	MEMORY_MANDATORY が必要です。ポリシーが指定されていない場合のデフォルトのポリシーです。このポリシーでは、カーネルが、ユーザ・ページをスワップ・ファイル（swap(1M) を参照）に移動し、この cpuset に含まれるノード上の物理メモリを解放します。スワップ・スペースの空きがなくなった場合、プロセスは強制終了されます。

CPUSET_POLICY_KIL	MEMORY_MANDATORY が必要です。カーネルは、カーネルのヒープからなるべく多くの領域を解放しようとしますが、ユーザ・ページをスワップ・ファイルに移動しません。cpuset に含まれるノード上のすべての物理メモリに空きがなくなった場合、プロセスは強制終了されます。

permfile メンバーは、cpuset キューのアクセス・パーミッションを定義するファイルの名前です。permfile が参照する filename のファイル・パーミッションにより、cpuset へのアクセスが定義されます。パーミッションをチェックする必要があるときは、このファイルの現在のパーミッションが使用されます。したがって、特定の cpuset へのアクセスは、cpuset を破棄して再作成しなくても、単にアクセス・パーミッションを変更するだけで変更できます。ユーザは、permfile の読込みパーミッションがあれば cpuset に関する情報を取得し、実行パーミッションがあれば cpuset にプロセスをアタッチできます。

cpu メンバーは、cpuset_CPUList_t 構造体へのポインタです。cpuset_CPUList_t 構造体のメモリは、cpuset_QueueDef_t 構造体の割当てと解放のときに割当ておよび解放されます（cpusetAllocQueueDef(3x) を参照）。cpuset_CPUList_t 構造体には、cpuset に割当てられた CPU のリストが含まれます。cpuset_CPUList_t 構造体（cpuset.h インクルード・ファイルで定義）は、以下のように定義されます。

```
typedef struct {
                    int    count;
                    int    *list;
          } cpuset_CPUList_t;
```

count メンバーは、リストに含まれる CPU 数を定義します。

list メンバーは、CPU ID のリスト（割当てられた配列）へのポインタです。list 配列のメモリは、cpuset_CPUList_t 構造体の割当てと解放のときに割当ておよび解放されます。

例

この例では、ファイル /usr/tmp/mypermfile でアクセスが制御され、CPU ID 4、8、12 が含まれ、CPU 排他的およびメモリ排他的である cpuset キューを作成します。

```
cpuset_QueueDef_t *qdef;
               char              *qname = "myqueue";

               /* Alloc queue def for 3 CPU IDs */
              qdef = cpusetAllocQueueDef(3);
              if (!qdef) {
                   perror("cpusetAllocQueueDef");
                   exit(1);
```

```
}

/* Define attributes of the cpuset */
qdef->flags = CPUSET_CPU_EXCLUSIVE
            | CPUSET_MEMORY_EXCLUSIVE;
qdef->permfile = "/usr/tmp/mypermfile";
qdef->cpu->count = 3;
qdef->cpu->list[0] = 4;
qdef->cpu->list[1] = 8;
qdef->cpu->list[2] = 12;

/* Request that the cpuset be created */
if (!cpusetCreate(qname, qdef)) {
    perror("cpusetCreate");
    exit(1);
 }
 cpusetFreeQueueDef(qdef);
```

注記

cpusetCreate 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetAllocQueueDef(3x)、cpusetFreeQueueDef(3x)、および cpuset(5)

診断

成功した場合、cpusetCreate 関数は値 1 を返します。cpusetCreate 関数が失敗した場合、値 0 が返され、errno が設定されます。errno の値は、fopen(3S) および sysmp(2) で設定される値、または以下のいずれかです。

ENODEV	システムに存在しない CPU ID が要求されました。
EPERM	排他的 cpuset の一部としての CPU 0 の要求は許可されません。

cpusetAttach(3x)

名前

cpusetAttach - 現在のプロセスを cpuset にアタッチします。

形式

```
#include <cpuset.h>
     int cpusetAttach( char *qname);
```

説明

cpusetAttach 関数を使用して、現在のプロセスを、qname で識別される cpuset にアタッチします。すべての cpuset キューには、キューのアクセス・パーミッションを定義するファイルがあります。このファイルの実行パーミッションにより、特定のユーザが所有するプロセスが cpuset キューにプロセスをアタッチできるかどうかが決定します。

qname 引数は、現在のプロセスをアタッチする cpuset の名前です。

例

この例では、現在のプロセスを、mpi_set という名前の cpuset キューにアタッチします。

```
char *qname = "mpi_set";

               /* Attach to cpuset, if error - print error & exit */
               if (!cpusetAttach(qname)) {
                   perror("cpusetAttach");
                   exit(1);
               }
```

注記

cpusetAttach 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、 cpusetCreate(3x)、および cpuset(5)

診断

成功した場合、cpusetAttach 関数は値 1 を返します。cpusetAttach 関数が失敗した場合、値 0 が返され、errno が設定されます。errno の値は、sysmp(2) で使用される値と同じです。

cpusetAttachPID(3x)

名前

cpusetAttachPID - 特定のプロセスを cpuset にアタッチします。

形式

```
#include <cpuset.h>

     int cpusetAttachPID(qname, pid);

     char *qname;

     pid_t pid;
```

説明

cpusetAttachPID 関数を使用して、PID で識別される特定のプロセスを、qname で識別される cpuset にアタッチします。すべての cpuset キューには、キューのアクセス・パーミッションを定義するファイルがあります。このファイルの実行パーミッションにより、特定のユーザが所有するプロセスが cpuset キューにプロセスをアタッチできるかどうかが決定します。

qname 引数は、指定されたプロセスをアタッチする cpuset の名前です。

例

この例では、現在のプロセスを、 mpi_set という名前の cpuset キューにアタッチします。

```
char *qname = "mpi_set";

/* Attach to cpuset, if error - print error & exit */
if (!cpusetAttachPID(qname, pid)) {
perror("cpusetAttachPID");
exit(1);                }
```

注記

cpusetAttachPID はライブラリ libcpuset.so にあり、cc(1) または ld(1) でオプション -l cpuset が使用されたときに読込まれます。

関連項目

cpuset(1) cpusetCreate(3x)cpusetDetachPID(3x) および cpuset(5)

診断

成功した場合、cpusetAttachPID は 1 を返します。cpusetAttachPID が失敗した場合、値 0 が返され、errno が設定されます。errno の値は、sysmp(2) で使用される値と同じです。

cpusetDetachPID(3x)

名前

cpusetDetachPID - 特定のプロセスを cpuset からデタッチします。

形式

```
#include <cpuset.h>

     int cpusetDetachPID(qname, pid);

     char *qname;

     pid_t pid;
```

説明

cpusetDetachPID 関数を使用して、PID で識別される特定のプロセスを、qname で識別される cpuset からデタッチします。すべての cpuset キューには、キューのアクセス・パーミッションを定義するファイルがあります。このファイルの実行パーミッションにより、特定のユーザが所有するプロセスが cpuset キューからプロセスをデタッチできるかどうかが決定します。

qname 引数は、指定されたプロセスをデタッチする cpuset の名前です。

例

この例では、現在のプロセスを、mpi_set という cpuset キューからデタッチします。

```
char *qname = "mpi_set";

/* Detach from cpuset, if error - print error & exit */
if (!cpusetDetachPID(qname, pid)) {
perror("cpusetDetachPID");
exit(1);                }
```

注記

cpusetDetachPID はライブラリ libcpuset.so にあり、cc(1) または ld(1) でオプション -l cpuset が使用されたときに読込まれます。

関連項目

cpuset(1) cpusetCreate(3x)cpusetAttachPID(3x) および cpuset(5)

診断

成功した場合、cpusetDetachPID は 1 を返します。cpusetAttachPID が失敗した場合、値 0 が返され、errno が設定されます。errno の値は、sysmp(2) で使用される値と同じです。

cpusetDetachAll(3x)

名前

cpusetDetachAll - cpuset からすべてのスレッドをデタッチします。

形式

```
#include <cpuset.h>
     int cpusetDetachAll(char *qname);
```

説明

cpusetDetachAll 関数を使用して、指定の cpuset に現在アタッチされているスレッドをすべてデタッチします。root ユーザ ID で実行されているプロセスだけが、cpusetDetachAll を正常に実行できます。

qname 引数は、操作を実行する cpuset の名前です。

例

この例では、現在のプロセスを、 mpi_set という cpuset キューからデタッチします。

```
char *qname = "mpi_set";

               /* Detach all members of cpuset, if error - print error & exit */
               if (!cpusetDetachAll(qname)) {
                   perror("cpusetDetachAll");
                   exit(1);
               }
```

注記

cpusetDetachAll 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、 cpusetAttach(3x)、および cpuset(5)

診断

成功した場合、cpusetDetachAll 関数は値 1 を返します。cpusetDetachAll 関数が失敗した場合、値 0 が返され、errno が設定されます。errno の値は、sysmp(2) で使用される値と同じです。

cpusetDestroy(3x)

名前

cpusetDestroy - cpuset を破棄します。

形式

```
#include <cpuset.h>
int cpusetDestroy(char *qname);
```

説明

cpusetDestroy 関数を使用して、指定の cpuset を破棄します。qname 引数は、破棄する cpuset の名前です。root ユーザ ID で実行されているプロセスだけが、cpuset キューを破棄できます。cpuset は、現在アタッチされているスレッドがない場合のみ破棄できます。

例

この例では、mpi_set という名前の cpuset キューを破棄します。

```
char *qname = "mpi_set";

              /* Destroy, if error - print error & exit */
              if (!cpusetDestroy(qname)) {
                  perror("cpusetDestroy");
                  exit(1);
              }
```

注記

cpusetDestroy 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetCreate(3x)、および cpuset(5)

診断

成功した場合、cpusetDestroy 関数は値 1 を返します。cpusetDestroy 関数が失敗した場合、値 0 が返され、errno が設定されます。errno の値は、sysmp(2) で使用される値と同じです。

cpusetMove(3x)

名前

cpusetMove - ID に関連付けられているプロセスを別の cpuset に移動します。

形式

```
#include <cpuset.h>

int cpusetMove(char *from_qname, char *to_qname, int idtype, int64_t id);
```

説明

cpusetMove 関数を使用して、ID 値 id に関連付けられるプロセスを、ある cpuset から他の cpuset に一時的に移動できます。この関数は、プロセスに関連付けられているメモリは移動しません。この関数は、cpusetMoveMigrate と共に使用する必要があります。

from_qname 引数は、プロセスの移動元の cpuset の名前です。この引数に対して NULL を使用すると、id で識別されるすべてのプロセスがグローバル cpuset に移動される結果となります。グローバル cpuset とは、cpuset 内には含まれないすべての CPU を説明するために使用される用語です。

to_qname 引数は、指定された ID の移動先の cpuset の名前です。この引数に対して NULL を使用すると、id で識別されるすべてのプロセスがグローバル cpuset に移動される結果となります。

idType 引数は、id で渡される値のタイプを定義します。idType に対して使用できるオプションは、CPUSET_PID（プロセス ID）、CPUSET_PGID（プロセス・グループ ID）、CPUSET_JID（ジョブ ID）、CPUSET_SID（セッション ID）、または CPUSET_ASH（アレイ・セッション・ハンドル）です。

この関数で移動を行うには、id に関連付けられているプロセスを停止する必要があります。すべてのプロセスが停止されているかどうかを確認するため、テストが実行されます。id にプロセス A、B、および C があり、B が停止されている場合は、A および C が停止されます。続いて、移動後に A および C が再起動されます（B は再起動されません）。

この関数を使用するには、標準の IRIX においては root 特権、Trusted IRIX (TRIX) においては CAP_SCHED_MGMT が必要です。

例

この例では、mpi_set という名前の cpuset キューから、my_set という名前の cpuset キューにプロセス ID を移動します。

```
char *from_qname = "mpi_set";
char *to_qname = "my_set";
int64_t id = 1534;
```

```
/* move from mpi_set to my_set,
* if error - print error & exit
*/
if (!cpusetMove(from_qname, to_qname, CPUSET_PID, id)) {
perror("cpusetMove");
exit(1);
}
```

注記

cpusetMove 関数はライブラリ libcpuset.so にあり、cc(1) または ld(1) でオプション -l cpuset を使用した場合に読込まれます。

関連項目

cpuset(1)、cpusetCreate(3x)、cpusetMoveMigrate(3x)、および cpuset(5)

診断

成功した場合、cpusetMove 関数は値 1 を返します。cpusetMove 関数が失敗した場合、値 0 が返され、errno が設定されます。errno の値は、sysmp(2) で使用される値と同じです。

cpusetMoveMigrate(3x)

名前

cpusetMoveMigrate - ID によって識別される特定のプロセスとそれらに関連付けられているメモリを、特定の cpuset から別の cpuset に移動します。

形式

```
#include <cpuset.h>
int cpusetMoveMigrate(char *from_qname, char *to_qname, int idtype,
int64_t id);
```

説明

cpusetMoveMigrate 関数を使用して、id で識別されるプロセスとそれらに関連付けられているメモリを、特定の cpuset から別の cpuset に移動します。

from_qname 引数は、プロセスの移動元の cpuset の名前です。この引数に対して NULL を使用すると、id で識別されるすべてのプロセスがグローバル cpuset に移動される結果となります。グローバル cpuset とは、cpuset 内には含まれないすべての CPU を説明するために使用される用語です。

to_qname 引数は、指定された ID の移動先の cpuset の名前です。この引数に対して NULL を使用すると、id で識別されるすべてのプロセスがグローバル cpuset に移動される結果となります。

idtype 引数は、id で渡される値のタイプを定義します。idtype に対して使用できるオプションは、CPUSET_PID（プロセス ID）、CPUSET_PGID（プロセス・グループ ID）、CPUSET_JID（ジョブ ID）、CPUSET_SID（セッション ID）、または CPUSET_ASH（アレイ・セッション・ハンドル）です。

この関数で移動を行うには、id に関連付けられているプロセスを停止する必要があります。すべてのプロセスが停止されているかどうかを確認するため、テストが実行されます。id にプロセス A、B、および C があり、B が停止されている場合は、A および C が停止されます。続いて、移動後に A および C が再起動されます（B は再起動されません）。

この関数を使用するには、標準の IRIX においては root 特権、Trusted IRIX (TRIX) においては CAP_SCHED_MGMT が必要です。

例

この例では、mpi_set という名前の cpuset キューから、my_set という名前の cpuset キューにプロセス ID を移動します。

```
char *from_qname = "mpi_set";
char *to_qname = "my_set";
int64_t id = 1534;
```

```
/* move from mpi_set to my_set,
* if error - print error & exit
*/
if (!cpusetMoveMigrate(from_qname, to_qname, CPUSET_PID, id)) {
perror("cpusetMoveMigrate");
exit(1);
}
```

注記

cpusetMoveMigrate 関数はライブラリ libcpuset.so にあり、cc(1) または ld(1) でオプション -l cpuset を使用した場合に読込まれます。

関連項目

cpuset(1)、cpusetCreate(3x)、cpusetMove(3x)、および cpuset(5)

診断

成功した場合、cpusetMoveMigrate 関数は値 1 を返します。cpusetMoveMigrate 関数が失敗した場合、値 0 が返され、errno が設定されます。errno の値は、sysmp(2) で使用される値と同じです。

取得関数

この節では、以下の cpuset システム・ライブラリの取得関数のマン・ページを記載します。

`cpusetGetCPUCount`(3x)	システム上で構成されている CPU の数を取得します。
`cpusetGetCPUList`(3x)	cpuset に割当てられたすべての CPU のリストを取得します。
`cpusetGetName`(3x)	プロセスがアタッチされている cpuset の名前を取得します。
`cpusetGetNameList`(3x)	定義されたすべての cpuset の名前のリストを取得します。
`cpusetGetPIDList`(3x)	cpuset に割当てられたすべての PID のリストを取得します。
`cpusetGetProperties`(3x)	cpuset に関連付けられているさまざまなプロパティを取得します。
`cpusetAllocQueueDef`(3x)	`cpuset_QueueDef_t` 構造体を割当てます。

cpusetGetCPUCount(3x)

名前

cpusetGetCPUCount - システム上で構成されている CPU の数を取得します。

形式

```
#include <cpuset.h>
int cpusetGetCPUCount(void);
```

説明

cpusetGetCPUCount 関数は、システム上で構成されている CPU の数を返します。

例

この例では、システム上で構成されている CPU の数を取得し、結果を出力します。

```
int ncpus;

if (!(ncpus = cpusetGetCPUCount())) {
perror("cpusetGetCPUCount");
exit(1);
}
printf("The systems is configured for %d CPUs\n",
        ncpus);
```

注記

cpusetGetCPUCount 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1) および cpuset(5)

診断

成功した場合、cpusetGetCPUCount 関数は 1 以上の値を返します。cpusetGetCPUCount 関数が失敗した場合、値 0 が返され、errno が設定されます。errno の値は、sysmp(2) で使用される値または以下のいずれかです。

ERANGE	システム上で構成されている CPU の数が 1 以上の値ではありません。

cpusetGetCPUList(3x)

名前

cpusetGetCPUList - cpuset に割当てられたすべての CPU のリストを取得します。

形式

```
#include <cpuset.h>
cpuset_CPUList_t *cpusetGetCPUList(char *qname);
```

説明

cpusetGetCPUList 関数を使用して、指定の cpuset に割当てられた CPU のリストを取得します。ユーザ ID またはグループ ID で実行され、パーミッション・ファイルの読込みパーミッションのあるプロセスだけが、この関数を正常に実行できます。qname 引数は、指定する cpuset の名前です。

この関数は、cpuset_CPUList_t 型の構造体（cpuset.h インクルード・ファイルで定義）へのポインタを返します。cpusetGetCPUList 関数が、構造体のメモリを割当てます。メモリの解放は、cpusetFreeCPUList(3x) 関数を使用してユーザが行います。cpuset_CPUList_t 構造体は以下のようになっています。

```
typedef struct {
                    int    count;
                    pid_t  *list;
                } cpuset_CPUList_t;
```

count メンバーは、リスト内の CPU ID 数です。list メンバーは、CPU ID のリストが含まれるメモリ配列を参照します。list のメモリは、cpuset_CPUList_t 構造体が割当てられるときに割当てられ、cpuset_CPUList_t 構造体が解放されるときに解放されます。

例

この例では、cpuset mpi_set に割当てられた CPU のリストを取得し、CPU ID の値を出力します。

```
char *qname = "mpi_set";
    cpuset_CPUList_t *cpus;

    /* Get the list of CPUs else print error & exit */
    if (!( cpus = cpusetGetCPUList(qname))) {
        perror("cpusetGetCPUList");

        exit(1);
    }
    if (cpus->count == 0) {
```

```
        printf("CPUSET[%s] has 0 assigned CPUs\n",
                qname);
    } else {
        int i;

        printf("CPUSET[%s] assigned CPUs:\n",
                qname);
        for (i = 0; i < cpuset->count; i++)
            printf("CPU_ID[%d]\n", cpuset->list[i]);
    }
    cpusetFreeCPUList(cpus);
```

注記

cpusetGetCPUList 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetFreeCPUList(3x)、および cpuset(5)

診断

成功した場合、cpusetGetCPUList 関数は cpuset_CPUList_t 構造体へのポインタを返します。cpusetGetCPUList 関数が失敗した場合、NULL が返され、errno が設定されます。errno の値は、sysmp(2) および sbrk(2) で設定される値です。

cpusetGetName(3x)

名前

cpusetGetName - プロセスがアタッチされている cpuset の名前を取得します。

形式

```
#include <cpuset.h>
cpuset_NameList_t *cpusetGetName(pid_t pid);
```

説明

cpusetGetName 関数を使用して、指定のプロセスがアタッチされた cpuset の名前を取得します。*pid* 引数はプロセス ID を指定します。現在は、*pid* の唯一の有効な値は 0 で、このとき、現在のプロセスがアタッチされている cpuset の名前が返されます。

この関数は、cpuset_NameList_t 型の構造体(cpuset.h インクルード・ファイルで定義)へのポインタを返します。cpusetGetName 関数は、構造体とそのすべての関連データのメモリを割当てます。メモリの解放は、cpusetFreeNameList(3x) 関数を使用してユーザが行います。cpuset_NameList_t 構造体は以下のように定義されます。

```
typedef struct {
                    int    count;
                    char   **list;
                    int   *status;
                } cpuset_NameList_t;
```

count メンバーは、リスト内の cpuset 名の数です。cpusetGetName 関数の場合、このメンバーの値は 0 または 1 である必要があります。

list メンバーは、名前のリストを参照します。

status メンバーは、list 内の対応する cpuset 名のステータスを示すステータス・フラグのリストです。以下のフラグの値が使用されます。

CPUSET_QUEUE_NAME	list 内の対応する名前は cpuset キューの名前であることを示します。
CPUSET_CPU_NAME	list 内の対応する名前は、制限付き CPU の CPU ID であることを示します。

list と status のメモリは、cpuset_NameList_t 構造体が割当てられるときに割当てられ、cpuset_NameList_t 構造体が解放されるときに解放されます。

例

この例では、現在のプロセスがアタッチされている cpuset 名または CPU ID を取得します。

```
cpuset_NameList_t *name;

                /* Get the list of names else print error & exit */
                if (!(name = cpusetGetName(0))) {
                    perror("cpusetGetName");
                    exit(1);
                }
                if (name->count == 0) {
                    printf("Current process not attached\n");
                } else {
                    if (name->status[0] == CPUSET_CPU_NAME) {
                        printf("Current process attached to"
                                " CPU_ID[%s]\n",
                                name->list[0]);
                    } else {
                        printf("Current process attached to"
                                " CPUSET[%s]\n",
                                name->list[0]);
                    }
                }
                cpusetFreeNameList(name);
```

注記

cpusetGetName 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetFreeNameList(3x)、cpusetGetNameList(3x)、および cpuset(5)

診断

成功した場合、cpusetGetName 関数は cpuset_NameList_t 構造体へのポインタを返します。cpusetGetName 関数が失敗した場合、NULL が返され、errno が設定されます。errno の値は、sysmp(2) および sbrk(2) で設定される値、または以下のいずれかです。

EINVAL	*pid* に無効な値が指定されました。現在は、0 だけを使用でき、このとき、現在のプロセスがアタッチされている cpuset 名が取得されます。
ERANGE	システム上で構成されている CPU の数が 1 以上の値ではありません。

cpusetGetNameList(3x)

名前

cpusetGetNameList - 定義されたすべての cpuset の名前のリストを取得します。

形式

```
#include <cpuset.h>
cpuset_NameList_t *cpusetGetNameList(void);
```

説明

cpusetGetNameList 関数を使用して、システム上のすべての cpuset の名前のリストを取得します。

この関数は、cpuset_NameList_t 型の構造体(cpuset.h インクルード・ファイルで定義)へのポインタを返します。cpusetGetNameList 関数は、構造体とそのすべての関連データのメモリを割当てます。メモリの解放は、cpusetFreeNameList(3x) 関数を使用してユーザが行います。cpuset_NameList_t 構造体は、以下のように定義されます。

```
typedef struct {
                    int    count;
                    char   **list;
                    int   *status;
                } cpuset_NameList_t;
```

count メンバーは、リスト内の cpuset 名の数です。

list メンバーは、名前のリストを参照します。

status メンバーは、list 内の対応する cpuset 名のステータスを示すステータス・フラグのリストです。以下のフラグの値が使用されます。

CPUSET_QUEUE_NAME	list 内の対応する名前は cpuset キューの名前であることを示します。
CPUSET_CPU_NAME	list 内の対応する名前は、制限付き CPU の CPU ID であることを示します。

list と status のメモリは、cpuset_NameList_t 構造体が割当てられるときに割当てられ、cpuset_NameList_t 構造体が解放されるときに解放されます。

例

この例では、システム上で構成されているすべての cpuset キューの名前のリストを取得します。cpuset または制限付き CPU ID のリストが出力されます。

```
cpuset_NameList_t *names;

/* Get the list of names else print error & exit */
if (!(names = cpusetGetNameList())) {
    perror("cpusetGetNameList");
    exit(1);
}
if (names->count == 0) {
    printf("No defined CPUSETs or restricted CPUs\n");
} else {
    int i;

    printf("CPUSET and restricted CPU names:\n");
    for (i = 0; i < names->count; i++) {
        if (names->status[i] == CPUSET_CPU_NAME) {
            printf("CPU_ID[%s]\n", names->list[i]);
        } else {
            printf("CPUSET[%s]\n", names->list[i]);
        }
    }
}
cpusetFreeNameList(names);
```

注記

cpusetGetNameList 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetFreeNameList(3x)、および cpuset(5)

診断

成功した場合、cpusetGetNameList 関数は cpuset_NameList_t 構造体へのポインタを返します。cpusetGetNameList 関数が失敗した場合、NULL が返され、errno が設定されます。errno の値は、sysmp(2) および sbrk(2) で設定される値です。

cpusetGetPIDList(3x)

名前

cpusetGetPIDList - cpuset にアタッチされているすべての PID のリストを取得します。

形式

```
#include <cpuset.h>
cpuset_PIDList_t *cpusetGetPIDList(char *qname);
```

説明

cpusetGetPIDList 関数を使用して、指定の cpuset に現在アタッチされているすべてのプロセスの PID のリストを取得します。ユーザ ID またはグループ ID で実行され、パーミッション・ファイルの読込みパーミッションのあるプロセスだけが、この関数を正常に実行できます。

qname 引数は、現在のプロセスをアタッチする cpuset の名前です。

この関数は、cpuset_PIDList_t 型の構造体（cpuset.h インクルード・ファイルで定義）へのポインタを返します。cpusetGetPIDList 関数が、構造体のメモリを割当てます。メモリの解放は、cpusetFreePIDList(3x) 関数を使用してユーザが行います。cpuset_PIDList_t 構造体は以下のようになっています。

```
typedef struct {
                    int    count;
                    pid_t  *list;
                } cpuset_PIDList_t;
```

count メンバーは、list 内の PID 値の数です。list メンバーは、PID 値のリストが含まれるメモリ配列を参照します。list のメモリは、cpuset_PIDList_t 構造体が割当てられるときに割当てられ、cpuset_PIDList_t 構造体が解放されるときに解放されます。

例

この例では、cpuset mpi_set に割当てられた PID のリストを取得し、PID の値を出力します。

```
(char              *qname = "mpi_set";)
cpuset_PIDList_t *pids;

                /* Get the list of PIDs else print error & exit */
                if (!(pids = cpusetGetPIDList(qname))) {
                    perror("cpusetGetPIDList");
                    exit(1);
                }
```

```
if (pids->count == 0) {
    printf("CPUSET[%s] has 0 processes attached\n",
            qname);
} else {
    int i;
    printf("CPUSET[%s] attached PIDs:\n",
             qname);
    for (i=o; i<pids->count; i++)
             printf("PID[%d]\n", pids->list[i] );
}
cpusetFreePIDList(pids);
```

注記

cpusetGetPIDList 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetFreePIDList(3x)、および cpuset(5)

診断

成功した場合、cpusetGetPIDList 関数は cpuset_PIDList_t 構造体へのポインタを返します。cpusetGetPIDList 関数が失敗した場合、NULL が返され、errno が設定されます。errno の値は、sysmp(2) および sbrk(2) で設定される値と同じです。

cpusetGetProperties(3x)

名前

cpusetGetProperties - cpuset に関連付けられているさまざまなプロパティを取得します。

形式

```
#include <cpuset.h>
cpuset_Properties_t * cpusetGetProperties(char *qname);
```

説明

cpusetGetProperties 関数を使用して、qname で識別されるさまざまなプロパティを取得します。この関数は、cpuset_Properties_t 構造体へのポインタを返します。すべての cpuset キューには、キューのアクセス・パーミッションを定義するファイルがあります。このファイルの読込みパーミッションにより、特定のユーザが所有するプロセスが cpuset からプロパティを取得できるかどうかが決定します。

qname 引数は、プロパティを取得する cpuset の名前です。

例

この例では、mpi_set という cpuset キューからプロパティを取得します。

```
char *qname = "mpi_set";
                    cpuset_Properties_t             *csp;


                    /* Get properties, if error - print error & exit */
                    csp=cpusetGetProperties(qname);
                    if (!csp) {
                            perror("cpusetGetProperties");
                            exit(1);
                    }
                    .
                    .
                    .
                    cpusetFreeProperties(csp);
```

有効なポインタが返された後に、フラグ CPUSET_ACCESS_ACL、CPUSET_DEFAULT_ACL、および CPUSET_MAC_LABEL を使用して、cpuset_Properties_t 構造体の extFlags メンバーをチェックし、有効な ACL または有効な MAC ラベルが返されたかどうかを確認する必要があります。チェック・フラグは、<sys\cpuset.h> ファイルにあります。

注記

cpusetGetProperties 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetFreeProperties(3x)、および cpuset(5)

診断

成功した場合、cpusetGetProperties 関数は cpuset_Properties_t 構造体へのポインタを返します。cpusetGetProperties 関数が失敗した場合、NULL が返され、errno が設定されます。errno の値は、sysmp(2) で設定される値です。

cpusetAllocQueueDef(3x)

名前

cpusetAllocQueueDef - cpuset_QueueDef_t 構造体を割当てます。

形式

```
#include <cpuset.h>
cpuset_QueueDef_t *cpusetAllocQueueDef(int count)
```

説明

cpusetAllocQueueDef 関数を使用して、cpuset_QueueDef_t 構造体のメモリを割当てます。このメモリは、cpusetFreeQueueDef(3x) 関数を使用して解放できます。

count 引数は、cpuset 定義構造体に割当てられる CPU 数を示します。cpuset_QueueDef_t 構造体は、以下のように定義されます。

```
typedef struct {
                    int                 flags;
                    char                *permfile;
                    cpuset_CPUList_t    *cpu;
                } cpuset_QueueDef_t;
```

flags メンバーを使用して、cpuset キューのさまざまな制御オプションを指定します。ビット単位の排他的論理和の演算子を以下の 0 個以上の値に適用して作成します。

CPUSET_CPU_EXCLUSIVE
: cpuset を制限付きと定義します。cpuset キューにアタッチされたスレッドだけが、cpuset に含まれる CPU で実行可能です（アタッチされたスレッドの子孫はアタッチを継承）。

CPUSET_MEMORY_LOCAL
: cpuset に割当てられたスレッドは、その cpuset 内のノードからのみメモリを割当てようとします。cpuset の外部からのメモリ割当ては、cpuset 内で空きメモリが利用できない場合にのみ発生します。cpuset の外部で実行しているスレッドへのメモリ割当てには制限は加えられません。

CPUSET_MEMORY_EXCLUSIVE	cpuset に割当てられたスレッドは、その cpuset 内のノードからのみメモリを割当てようとします。cpuset の外部からのメモリ割当ては、cpuset 内で空きメモリが利用できない場合にのみ発生します。cpuset の外部でメモリが利用できない場合を除き、cpuset に割当てられないスレッドは、その cpuset 内のメモリを使用しません。cpuset の作成時に、実行中のスレッドにメモリがすでに割当てられている場合は、このメモリを明示的に移動する試みは行われません。ページ移動が可能な場合、ページへの参照のほとんどが非ローカルであることがシステムで検出されると、そのページは移動されます。
CPUSET_MEMORY_KERNEL_AVOID	カーネルは、この cpuset に含まれるノードからのメモリ割当てを避けようとします。カーネルのメモリ要求が、この cpuset の外部では満たされない場合、このオプションは無視され、cpuset 内部でのメモリ割当てが発生します（この回避動作は現在、保護されたノードからバッファ・キャッシュを隔離するだけです）。
CPUSET_MEMORY_MANDATORY	カーネルは、すべてのメモリ割当てをこの cpuset に含まれるノードに限定します。メモリ要求が満たされない場合、メモリが使用可能になるまで割当てプロセスはスリープします。これ以上のメモリを割当てられない場合、プロセスは強制終了されます。以下のポリシーを参照してください。
CPUSET_POLICY_PAGE	MEMORY_MANDATORY が必要です。ポリシーが指定されていない場合のデフォルトのポリシーです。このポリシーでは、カーネルが、ユーザ・ページをスワップ・ファイル（swap(1M) を参照）に移動し、この cpuset に含まれるノード上の物理メモリを解放します。スワップ・スペースの空きがなくなった場合、プロセスは強制終了されます。

CPUSET_POLICY_KILL	MEMORY_MANDATORY が必要です。カーネルは、カーネルのヒープからなるべく多くの領域を解放しようとしますが、ユーザ・ページをスワップ・ファイルに移動しません。cpuset に含まれるノード上のすべての物理メモリに空きがなくなった場合、プロセスは強制終了されます。

permfile メンバーは、cpuset キューのアクセス・パーミッションを定義するファイルの名前です。permfile が参照する filename のファイル・パーミッションにより、cpuset へのアクセスが定義されます。パーミッションをチェックする必要があるときは、このファイルの現在のパーミッションが使用されます。したがって、特定の cpuset へのアクセスは、cpuset を破棄して再作成しなくても、単にアクセス・パーミッションを変更するだけで変更できます。ユーザは、permfile の読込みパーミッションがあれば cpuset に関する情報を取得し、実行パーミッションがあれば cpuset にプロセスをアタッチできます。

cpu メンバーは、cpuset_CPUList_t 構造体へのポインタです。cpuset_CPUList_t 構造体のメモリは、cpuset_QueueDef_t 構造体の割当てと解放のときに割当ておよび解放されます（cpusetFreeQueueDef(3x) を参照）。cpuset_CPUList_t 構造体には、cpuset に割当てられた CPU のリストが含まれます。cpuset_CPUList_t 構造体（cpuset.h インクルード・ファイルで定義）は、以下のように定義されます。

```
typedef struct {
                    int    count;
                    int    *list;
                } cpuset_CPUList_t;
```

count メンバーは、リストに含まれる CPU 数を定義します。

list メンバーは、CPU ID のリスト（割当てられた配列）へのポインタです。list 配列のメモリは、cpuset_CPUList_t 構造体の割当てと解放のときに割当ておよび解放されます。リストのサイズは、cpusetAllocQueueDef 関数に渡される count 引数によって決定します。

例

この例では、cpusetCreate(3x) 関数を使用して cpuset キューを作成し、cpusetAllocQueueDef 関数の使用方法の例を示します。作成される cpuset は、ファイル /usr/tmp/mypermfile でアクセスが制御され、CPU ID 4、8、12 が含まれ、CPU 排他的およびメモリ排他的です。

```
cpuset_QueueDef_t *qdef;
                char               *qname = "myqueue";

                /* Alloc queue def for 3 CPU IDs */
                qdef = cpusetAllocQueueDef(3);
                if (!qdef) {
```

```
    perror("cpusetAllocQueueDef");
    exit(1);
}

/* Define attributes of the cpuset */
qdef->flags = CPUSET_CPU_EXCLUSIVE
            | CPUSET_MEMORY_EXCLUSIVE;
qdef->permfile = "/usr/tmp/mypermfile";
qdef->cpu->count = 3;
qdef->cpu->list[0] = 4;
qdef->cpu->list[1] = 8;
qdef->cpu->list[2] = 12;

/* Request that the cpuset be created */
if (!cpusetCreate(qname, qdef)) {
    perror("cpusetCreate");
    exit(1);
}
cpusetFreeQueueDef(qdef);
```

注記

cpusetAllocQueueDef 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetFreeQueueDef(3x)、および cpuset(5)

診断

成功した場合、cpusetAllocQueueDef 関数は cpuset_QueueDef_t 構造体へのポインタを返します。cpusetAllocQueueDef 関数が失敗した場合、NULL が返され、errno が設定されます。errno の値は、sbrk(2) で返される値、または以下のいずれかです。

EINVAL	無効な引数が指定されました。ユーザは、0 以上の値を指定する必要があります。

クリーンアップ関数

この節では、以下の cpuset システム・ライブラリのクリーンアップ関数のマン・ページを記載します。

`cpusetFreeQueueDef`(3x)	`cpuset_QueueDef_t` 構造体で使用されているメモリを解放します。
`cpusetFreeCPUList`(3x)	`cpuset_CPUList_t` 構造体で使用されているメモリを解放します。
`cpusetFreeNameList`(3x)	`cpuset_NameList_t` 構造体で使用されているメモリを解放します。
`cpusetFreePIDList`(3x)	`cpuset_PIDList_t` 構造体で使用されているメモリを解放します。
`cpusetFreeProperties`(3x)	`cpuset_Properties_t` 構造体で使用されているメモリを解放します。

cpusetFreeQueueDef(3x)

名前

cpusetFreeQueueDef - cpuset_QueueDef_t 構造体で使用されているメモリを解放します。

形式

```
#include <cpuset.h>
void cpusetFreeQueueDef(cpuset_QueueDef_t *qdef);
```

説明

cpusetFreeQueueDef 関数を使用して、cpuset_QueueDef_t 構造体に使用されているメモリを解放します。この関数は、cpuset_QueueDef_t 構造体に関連するメモリをすべて解放します。

qdef 引数は、メモリを解放する cpuset_QueueDef_t 構造体へのポインタです。

cpusetAllocQueueDef(3x) 関数を呼出して割当てられたメモリを解放するには、この関数を使用します。

注記

cpusetFreeQueueDef 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetAllocQueueDef(3x)、および cpuset(5)

cpusetFreeCPUList(3x)

名前

cpusetFreeCPUList - cpuset_CPUList_t 構造体で使用されているメモリを解放します。

形式

```
#include <cpuset.h>
void cpusetFreeCPUList(cpuset_CPUList_t *cpu);
```

説明

cpusetFreeCPUList 関数を使用して、cpuset_CPUList_t 構造体に使用されているメモリを解放します。この関数は、cpuset_CPUList_t 構造体に関連するメモリをすべて解放します。

cpu 引数は、メモリを解放する cpuset_CPUList_t 構造体へのポインタです。

cpusetGetCPUList(3x) 関数を呼出して割当てられたメモリを解放するには、この関数を使用します。

注記

cpusetFreeCPUList 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetGetCPUList(3x)、および cpuset(5)

cpusetFreeNameList(3x)

名前

cpusetFreeNameList - cpuset_NameList_t 構造体で使用されているメモリを解放します。

形式

```
#include <cpuset.h>
void cpusetFreeNameList(cpuset_NameList_t *name);
```

説明

cpusetFreeNameList 関数を使用して、cpuset_NameList_t 構造体に使用されているメモリを解放します。この関数は、cpuset_NameList_t 構造体に関連するメモリをすべて解放します。

name 引数は、メモリを解放する cpuset_NameList_t 構造体へのポインタです。

cpusetGetNameList(3x) 関数または cpusetGetName(3x) 関数を呼出して割当てられたメモリを解放するには、この関数を使用します。

注記

cpusetFreeNameList 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetGetName(3x)、cpusetGetNameList(3x)、および cpuset(5)

cpusetFreePIDList(3x)

名前

cpusetFreePIDList - cpuset_PIDList_t 構造体で使用されているメモリを解放します。

形式

```
#include <cpuset.h>
void cpusetFreePIDList(cpuset_PIDList_t *pid);
```

説明

cpusetFreePIDList 関数を使用して、cpuset_PIDList_t 構造体に使用されているメモリを解放します。この関数は、cpuset_PIDList_t 構造体に関連するメモリをすべて解放します。

pid 引数は、メモリを解放する cpuset_PIDList_t 構造体へのポインタです。

cpusetGetPIDList(3x) 関数を呼出して割当てられたメモリを解放するには、この関数を使用します。

注記

cpusetFreePIDList 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetGetPIDList(3x)、および cpuset(5)

cpusetFreeProperties(3x)

名前

cpusetFreeProperties - cpuset_Properties_t 構造体で使用されているメモリを解放します。

形式

```
#include <cpuset.h>
void cpusetFreeProperties(cpuset_Properties_t *csp);
```

説明

cpusetFreeProperties 関数を使用して、cpuset_Properties_t 構造体に使用されているメモリを解放します。この関数は、cpuset_Properties_t 構造体に関連するメモリをすべて解放します。

csp 引数は、メモリを解放する cpuset_Properties_t 構造体へのポインタです。

cpusetGetProperties(3x) 関数を呼出して割当てられたメモリを解放するには、この関数を使用します。

注記

cpusetFreeProperties 関数は libcpuset.so ライブラリにあり、cc(1) コマンドまたは ld(1) コマンドで -lcpuset オプションが使用されたときに読込まれます。

関連項目

cpuset(1)、cpusetGetProperties(3x)、および cpuset(5)

cpuset ライブラリの使用

この節では、cpuset ライブラリの関数を使用した cpuset の作成例と /lib32/libcpuset.so の代用ライブラリの作成例を示します。

サンプルA-1 cpuset の作成例

この例では、CPU 4、8、12 を含む、myqueue という cpuset を作成します。この例では、cpuset ライブラリ /lib32/libcpuset.so 内にインタフェースがある場合はそのインタフェースを使用します。インタフェースがない場合は、cpuset(1) コマンドを使用して cpuset を作成しようとします。

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

```
#define PERMFILE "/usr/tmp/permfile"

int
main(int argc, char **argv)
{
    cpuset_QueueDef_t *qdef;
    char              *qname = "myqueue";
    FILE              *fp;

    /* Alloc queue def for 3 CPU IDs */
    if (_MIPS_SYMBOL_PRESENT(cpusetAllocQueueDef)) {
        printf("Creating cpuset definition\n");
        qdef = cpusetAllocQueueDef(3);
        if (!qdef) {
            perror("cpusetAllocQueueDef");
            exit(1);
    }
        /* Define attributes of the cpuset */
        qdef->flags = CPUSET_CPU_EXCLUSIVE
                    | CPUSET_MEMORY_LOCAL
                    | CPUSET_MEMORY_EXCLUSIVE;
        qdef->permfile = PERMFILE;
        qdef->cpu->count = 3;
        qdef->cpu->list[0] = 4;
        qdef->cpu->list[1] = 8;
        qdef->cpu->list[2] = 12;
    } else {
        printf("Writing cpuset command config"
                " info into %s\n", PERMFILE);
        fp = fopen(PERMFILE, "a");
        if (!fp) {
            perror("fopen");
            exit(1);
        }
        fprintf(fp, "EXCLUSIVE\n");
        fprintf(fp, "MEMORY_LOCAL\n");
        fprintf(fp, "MEMORY_EXCLUSIVE\n\n");
        fprintf(fp, "CPU 4\n");
        fprintf(fp, "CPU 8\n");
        fprintf(fp, "CPU 12\n");
        fclose(fp);
```

```
}

/* Request that the cpuset be created */
if (_MIPS_SYMBOL_PRESENT(cpusetCreate)) {
    printf("Creating cpuset = %s\n", qname);
    if (!cpusetCreate(qname, qdef)) {
        perror("cpusetCreate");
        exit(1);
    }
} else {
    char command[256];

    fprintf(command, "/usr/sbin/cpuset -q %s -c"
           "-f %s", qname,
           [PERMFILE];
    if (system(command) < 0) {
        perror("system");
        exit(1);
    }
}

/* Free memory for queue def */
if (_MIPS_SYMBOL_PRESENT(cpusetFreeQueueDef)) {
    printf("Finished with cpuset definition,"
            " releasing memory\n");
    cpusetFreeQueueDef(qdef);
    }
    return 0;
}
```

サンプルA-2 代用ライブラリの作成例

この例では、/lib32/libcpuset.so の代用ライブラリを作成する方法を示します。代用ライブラリを作成すると、cpuset ライブラリがなくても、そのインタフェースを使用するように作成されたプログラムが実行されます。

1. 以下のコード行を含む replace.c を作成します。

   ```
   static void cpusetNULL(void) { }
   ```

2. 以下のように入力して、replace.c を編集します。

   ```
   cc -mips3 -n32 -c replace.c
   ```

3. 以下のように入力して、前の手順で作成した replace.o オブジェクトをライブラリに格納します。

```
ar ccrl libcpuset.a replace.o
```

4. 以下のように入力して、ライブラリを DSO に変換します。

```
ld -mips3 -n32 -quickstart_info -nostdlib        \
   -elf -shared -all -soname libcpuset.so       \
   -no_unresolved -quickstart_info -set_version \
   sgi1.0 libcpuset.a -o libcpuset.so
```

5. 以下のように入力して、DSO をシステムにインストールします。

```
install -F /opt/lib32 -m 444 -src libcpuset.so \
                      libcpuset.so
```

代用ライブラリは、LD_LIBRARYN32_PATH 環境変数によって定義されているディレクトリにインストールできます（rld(1) を参照）。代用ライブラリを、共有ライブラリのデフォルトの検索パスに含まれるディレクトリにインストールする必要がある場合は、/opt/lib32 にインストールします。

索引

き

し