IRIX® Admin: Resource Administration（日本語版）

007–3700–006JPN

制作スタッフ

著作 Terry Schultz

編集 Rick Thompson、Susan Wilkening

イラスト Chris Wengelski

製作 Diane Ciardelli

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

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

旧版からの変更点

『IRIX Admin: Resource Administration』の本改訂版では、IRIX リリース 6.5.11 をサポートしています。

新機能の解説

本改訂版では、完全システム・アカウンティング (CSA: Comprehensive System Accounting) ソフトウェアでサポートされているワークロード管理機能について解説しています。

主な変更点

IRIX リリース 6.5.11 に関する本改訂版では、ワークロード管理機能をサポートする 61 ページの 第5章「完全システム・アカウンティング (CSA: Comprehensive System Accounting)」について変更が加えられました。

改訂情報

バージョン	説明
001	1999 年 7 月 初版
002	2000 年 1 月 IRIX リリース 6.5.7 をサポート
003	2000 年 4月 IRIX リリース 6.5.8 をサポート
004	2000 年 8 月 IRIX リリース 6.5.9 をサポート
005	2000 年 11 月 IRIX リリース 6.5.10 をサポート
006	2001 年 2 月 IRIX リリース 6.5.11 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・システム上で動作する IRIX 6.5.11 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・システムが動作している SGI コンピュータ・システムの管理者向けのリファレンス・ドキュメントです。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、次の章で構成されています。

関連マニュアル

このガイドは、IRIX Admin のマニュアル・セットの一部であり、管理者向けの内容になっています。管理者とは、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以下にあるもの以外のファイル構造に対して責任を持つ人のことです。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、以下のマニュアルから構成されます。

- 『IRIX Admin: Software Installation and Licensing』IRIX（SGI による UNIX オペレーティング・システム）における、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、ミニルートの実行方法や `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 で動作する特定のアプリケーションの利用を制御するライセンス管理製品について説明し、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アップし、システム管理タスクについて説明します。オペレーティング・システムの設定方法、ユーザ・アカウン

ト/ユーザ・プロセス/ディスク・リソースの管理方法、PROM モニタ内でのシステムとの対話方法、システム・パフォーマンスのチューニング方法が含まれます。

- 『IRIX Admin: Disks and Filesystems』ディスク、ファイルシステム、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 や EFS (Extent File System) のファイルシステム、XLV 論理ボリューム、帯域保証 I/O に関するシステム管理手順について説明します。
- 『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、保守の方法について説明します。sendmail、UUCP、SLIP、PPP の解説が含まれます。
- 『IRIX Admin: Backup, Security and Accounting』ファイルのバックアップと復元の方法、システムとネットワークのセキュリティを確保する方法、ユーザごとにシステム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・リソース管理の手引きを提供し、IRIX のジョブ制限、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) など、さまざまな IRIX リソース管理機能の使用方法と管理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、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ています。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 のドキュメントを入手するには、SGI Technical Publications Library (`http://techpubs.sgi.com`) にアクセスしてください。

表記上の決まり

このマニュアル全体を通して、以下の表記規則を使用します。

表記規則	意味
`command`	この等幅フォントは、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、リテラル項目を示します。
variable	イタリック体は、変数のエントリや、定義される単語や概念を示します。
`user input`	このボールド体の等幅フォントは、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、ボールド体ではない等幅フォントで示されます。
[ ]	コマンドや指示語 行のオプション部分は、角括弧で囲みます。
...	省略記号は、先行する要素が繰り返し可能であることを示します。

ご意見とお問い合わせ先

技術的正確性、内容、マニュアルの構成についてご意見ございましたら、弊社までお問い合わせください。必ず、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あります。印刷マニュアルの場合は、裏表紙にドキュメント番号が記載されています。)

ご連絡は、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、

  `techpubs@sgi.com`

  までお送りください。

- 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://techpubs.sgi.com`

- お客様相談窓口までご連絡いただき、SGI 問題追跡システムへの入力をお申しつけください。

- 郵送の場合は、次の住所までお送りください。
  Technical Publications
  SGI

1600 Amphitheatre Pkwy., M/S 535
Mountain View, California 94043–1351

- FAX の場合は、+1 650 932 0801 の “Technical Publications” あてにお送りください。

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、プロセスの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、次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、リソースの制限を指定します。リソースの制限は、値の対です。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。プロセスは、ソフト制限をハード制限以下の任意の値に変更できます。プロセスは、ハード制限をソフト制限以上の任意の値に下げることができます（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) と sh(1) のマン・ページを参照してください。プロセスの特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) と 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	プロセスが使用できる 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	プロセスが作成できるファイルの最大サイズ	上書きや追加が失敗し、errno に EFBIG が設定される
rlimit_data_cur rlimit_data_max	RLIMIT_DATA	バイト	プロセスの最大ヒープ・サイズ	brk(2) 呼出しが失敗し、errno に ENOMEM が設定される
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	プロセスが作成できるコア・ファイルの最大サイズ	最大値に到達した時点でコア・ファイルの書込みを終了
rlimit_nofile_cur rlimit_nofile_max	RLIMIT_NOFILE	ファイル記述子	プロセスが同時に保持できる、開いているファイル記述子の最大数	open(2) の試行が失敗し、errno に EMFILE が設定される

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	brk(2) と mmap(2) の呼出しが失敗し、errno に ENOMEM が設定される
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	スレッド数	プロセスが作成できるスレッドの最大数	スレッド作成が失敗し、errno に EAGAIN が設定される

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* には、以下の値が使用できます。

```
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値に優先します。

ULDB についての詳細は、10 ページの「ULDB」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(core やファイルのサイズ)、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 と cpulimit_gracetime について説明します。パラメータはすべて、/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	プロセスやジョブの制限の猶予期間

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。このパラメータには、制限を超えた後にプロセスの実行が許される秒数を設定します。cputlimit_gracetime が設定されていない場合(すなわち、0 の場合)は、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を続けるかぎり、カーネルは SIGXCPU シグナルを周期的に送信します。プロセスは、SIGXCPU シグナルを独自に処理するように登録できるため、CPU 制限を無視することがあります。

systune(1M) コマンドを使用して、cpulimit_gracetime パラメータに 0 以外の値を設定した場合は、異なる処理が行われます。プロセスが CPU 制限を超えると、カーネルは SIGXCPU シグナルを 1 回だけプロセスに送信します。プロセスはこのシグナルを登録して、そこで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 猶予時間が経過してもなおプロセスが実行中の場合、カーネルは SIGKILL シグナルを使用してそのプロセスを終了させます。

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、アカウンティングを適用できます。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、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 が未定義の場合は、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。

この章は、次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. ジョブとジョブ制限に関する一般的な説明については、6 ページの「ジョブ制限の概要」と 8 ページの「サポートされるジョブ制限」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、23 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* の記述と、ULDB の作成方法については、それぞれ 12 ページの「ユーザ制限命令入力ファイルの作成」と 11 ページの「ULDB の作成」を参照してください。

   ジョブ制限に関連するマン・ページのリストは、24 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* コマンドを使用して、システム全体のジョブ制限のデフォルト値を設定する方法については、16 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を表示する方法については、17 ページの「ジョブ制限の表示・設定用ユーザ・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は、24 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) については、111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)」と114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は、8 ページの「サポートされるジョブ制限」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、常に 1 つのジョブ内に含まれます。7 ページの図2-1 は、ジョブの作成を起動するエントリ・ポイントのプロセスを示します。

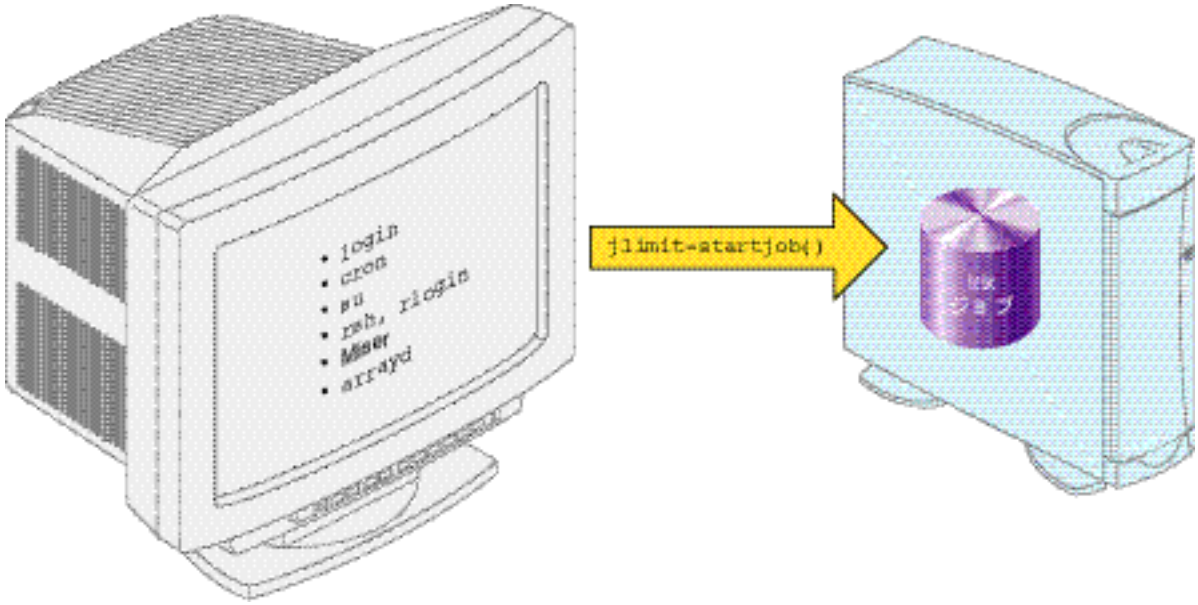

図2-1 エントリ・ポイントのプロセス

IRIX のジョブ制限には以下の特徴があります。

- 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(ルート特権でのシステム・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。
- 各新規プロセスは、親プロセスからジョブ ID と制限を継承します。
- 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)は、新規ジョブを作成し、ジョブ制限を適切に設定します。
- 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・イニシエータが認証とセキュリティ・チェックを行います。

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 0(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 プロセスによって起動され、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」、および arsctl(2) と 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)、bg(1) のマン・ページは、シェルの「ジョブ」に関するもので、IRIX カーネルのジョブとは関係ありません。

メモ: Silicon Graphics 社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、ULDB 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は、特定の名前で制限情報が検索できることを想定しています。この名前は規則に基づいて定義されます。制限ドメインと ULDB についての詳細は、10 ページの「ULDB」を参照してください。

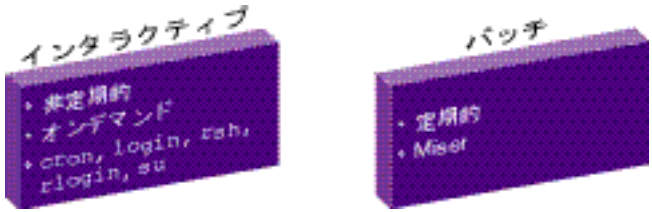

図2-2 制限ドメイン

サポートされるジョブ制限

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	ジョブによるプロセス生成は失敗し、errno に EAGAIN が設定される
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が保持できる、開いているファイル記述子の合計の最大数	ジョブによる open(2) の呼出しが失敗し、errno に EMFILE が設定される
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	ジョブでの brk(2) と mmap(2) の呼出しが失敗し、errno に ENOMEM が設定される
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	ジョブによる brk(2) の呼出しが失敗し、errno に ENOMEM が設定される
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	SIGXCPU シグナルによるジョブ内の全プロセスの終了
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	SIGKILL シグナルによるジョブ内の全プロセスの終了

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。プロセスの優先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) と capabilities(4) のマン・ページを参照してください。

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に設定された終了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、それぞれ、111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)」と114ページの「ULDB の API」、および waitjob(2) と setwaitjobpid(2) のマン・ページを参照してください。

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、16 ページの「systune を使用したジョブ制限の表示と設定」と systune(1M) のマン・ページ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。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別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。

ULDB

ULDB 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）は、特定のユーザのジョブ制限値を ULDB から取得し、その情報を使用して適切に制限を設定します。

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。ユーザごとの値は、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・デフォルト値に優先します。

この節では、ULDB の内容の作成、保守、表示に使用するコマンドや、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 について説明しています。

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` 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 のマスター・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の ULDB の mdbm ファイルです。

ULDB の作成

ULDB を作成するコマンドを以下に示します。

```
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) マスター・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が使用できます。

`-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 が使用可能な場合、ローカル・データベースには、NIS サーバのエントリに優先されるか、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 が使用可能でない場合、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 マスター・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 です。-N *nisfile* オプションを使用すれば、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は、genlimits(1M) コマンドへの入力が含まれ、ULDB の生成に使用するドメイン、制限、ユーザの情報を定義します。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、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は上限がないことを示す「unlimited」と指定することもできます。

ULDB 作成方法についての詳細は、`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、使用が可能です。予約されているドメイン名を以下に示します。

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第 2 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。マシン名を指定しない制限は、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、マシン名を指定しない制限に加えて、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が、生成されるデータベースのタイプ（11 ページの「ULDB 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 マスター・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、個別の制限のある 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、24 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、以下の変数があります。

```
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 - プロセス制限の数

- jlimit_nofile - ファイル制限の数
- 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) と 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・設定用ユーザ・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。

- 17 ページの「showlimits」
- 19 ページの「jlimit」
- 20 ページの「jstat」

showlimits

ULDB の制限情報を表示するコマンドを以下に示します。

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

showlimits コマンドは ULDB 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が使用できます。

-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は、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を以下に示します。

`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 ULDB 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常に制限の違反時の動作が発生します。プロセスの特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) と `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が使用できます。

-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を以下に示します。

```
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が使用できます。

-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。
`-Pk`	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` オプションと一緒に使用します。

`-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、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』、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、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。JID チェックポイント・タイプが cpr -p オプションに追加されています。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 パラメータを 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) と 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 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 ジョブを実行する場合、デフォルトで 400 に設定されている `files` の現在の制限はすぐに到達し、以下のようなエラー・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方法についての詳細は、10 ページの「ULDB」と 16 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) と `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、以下のサブシステムをインストールします。eoe.sw.jlimits

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。

ジョブ制限ソフトウェアは IRIX フィーチャ・ストリームでのみ利用可能です。

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どれが無効なの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 コマンドを使用すると、すべてのリソース管理コマンドに関するオンライン・ヘルプを参照できます。マン・ページをオンラインで表示するには、man *コマンド名*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)	ULDB 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)	ULDB 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強制終了します。
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。
setwaitjobpid(2)	指定されたプロセス ID (PID: process ID) を待ってから、waitjob(2) 関数を呼出す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)	ULDB 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です。

エラー・メッセージ

ジョブ制限に関する以下のエラー・メッセージが返されます。

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/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、次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、28 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、39 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は、30 ページの「Miser の設定」を参照してください。
4. Miser ジョブの指定方法については、40 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・ページについては、43 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、時間と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、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 を管理するように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/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に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 の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に要求します。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 が、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 予約の方法です。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 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 スレッドは実行を一時停止され、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 に管理されるジョブだけが、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 すべてを利用できますが、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・キューが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/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)、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。

以下に示すシステム・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、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、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ファイル・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、全般的な 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・ポリシーです。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に再スケジュールを試みます。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/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）、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。

以下に示すシステム・キュー定義ファイルでは、「default」という名前のポリシーを使用してキューを定義しています。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`) には、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。このファイルには、すべてのキュー名とそのキュー定義ファイルが列挙されます。

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 のシステム・キューを指定します。

Miser 設定ファイルのサンプルを以下に示します。

```
# 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に、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。

以下に示す例では、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* と *5* メガバイトに設定しています。

```
-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。

最初のステップでは、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を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
```

次のステップでは、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
```

最後のステップでは、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 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 の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
```

最後のステップでは、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は、36 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に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）、**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、そのセグメントが実行するように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`(4) と `miser_submit`(1) 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/説明に関する Miser への問い合わせ

指定したジョブのスケジュール/説明について、Miser に問い合わせを行うコマンドは、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 への問い合わせ

Miser のキューに関する情報、キューのリソース・ステータス、キューにスケジュールされたジョブのリストを Miser に問い合わせるコマンドは、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、start 時刻から end 時刻まで、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、ベクトル定義は変更しません。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と、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。開始時刻と終了時刻は、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。メモリは、**k**（キロバイト）、**m**（メガバイト）、**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 が強制終了されるか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) と `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 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。こうしたサイトでは、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 コマンドを使用すると、すべてのリソース管理コマンドについてオンライン・ヘルプを参照できます。マン・ページをオンラインで表示するには、man *コマンド名*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 に問い合わせます。
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、キューにスケジュールされたジョブのリストを Miser に問い合わせます。
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)	Miser キューにジョブを指定します。

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 設定ファイル
miser_move(4)	Miser リソース移動リスト
miser_submit(4)	Miser 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、以下のその他のマン・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、大規模なシステムが持つ CPU 群を分割できます。システムを分割すると、指定したプロセスの集合が特定の CPU の集合に含まれるように制御でき、これらのプロセスと、システム上のほかの作業との間の干渉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 で実行するようにスケジュールできます。開放された 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に制限し、これらのプロセスがシステムのほかの部分に及ぼす影響を最小限に抑えることができます。

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に使用できるようにできます。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を厳密に分離します。管理者は、たとえばシステムを 2 つに分割し、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構成と比較したときの利点は、簡単なロボットで構成を変更でき、構成をハードウェア・モジュールの境界に合わせる必要がないことです。

cpuset は、LSF (Load Sharing Facility) や PBS (Portable Batch System) などのバッチ処理システムとともにデータ・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。

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 ライブラリのインタフェースを使用すると、cpuset を作成、破棄したり、既存の cpuset に関する情報を取出したり、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、次の節で構成されています。

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、48 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、57 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、以下の手順に従ってください。

1. cpuset の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、54 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、47 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 に関する情報を取出したり、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` コマンドの構文は以下のとおりです。

`cpuset [-q` *cpuset_name* `[-A` *command*`]|[-c -f` *filename*`]|[-d]|[-l][-m]|[-Q]] | -C | -Q | -h`

`cpuset` コマンドでは以下のオプションが使用できます。

`-q` *cpuset_name* [`-A` *command*]

`-q` パラメータで指定された cpuset で、指定され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いか、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` パラメータで指定された設定ファイルと、`-q` パラメータで指定された名前で、cpuset を作成します。cpuset 名がすでに存在するか、cpuset の設定ファイルで指定されている CPU がすでに cpuset のメンバーであるか、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`	指定の 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のみ破棄できます。
`-q` *cpuset_name* `-l`	cpuset 内の全プロセスのリストを示します。
`-q` *cpuset_name* `-m`	アタッチされたプロセスをすべて cpuset の外に移動します。
`-q` *cpuset_name* `-Q`	cpuset に含まれる CPU のリストを出力します。
`-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 の名前のリストを示します。
`-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. 以下の `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 の使用方法についての詳細は、`cpuset`(1) のマン・ページ、47 ページの「Cpuset 内の CPU に対する制限」、および 48 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 のみに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。
- 制限付きの CPU または隔離された CPU は(`mpadmin`(1) と `sysmp`(2) を参照)、cpuset のメンバーにできません。

- 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 コマンドの引数やその詳細については、cpuset(1)、cpuset(4)、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。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。

49 ページの図4-1 は、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ています。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。

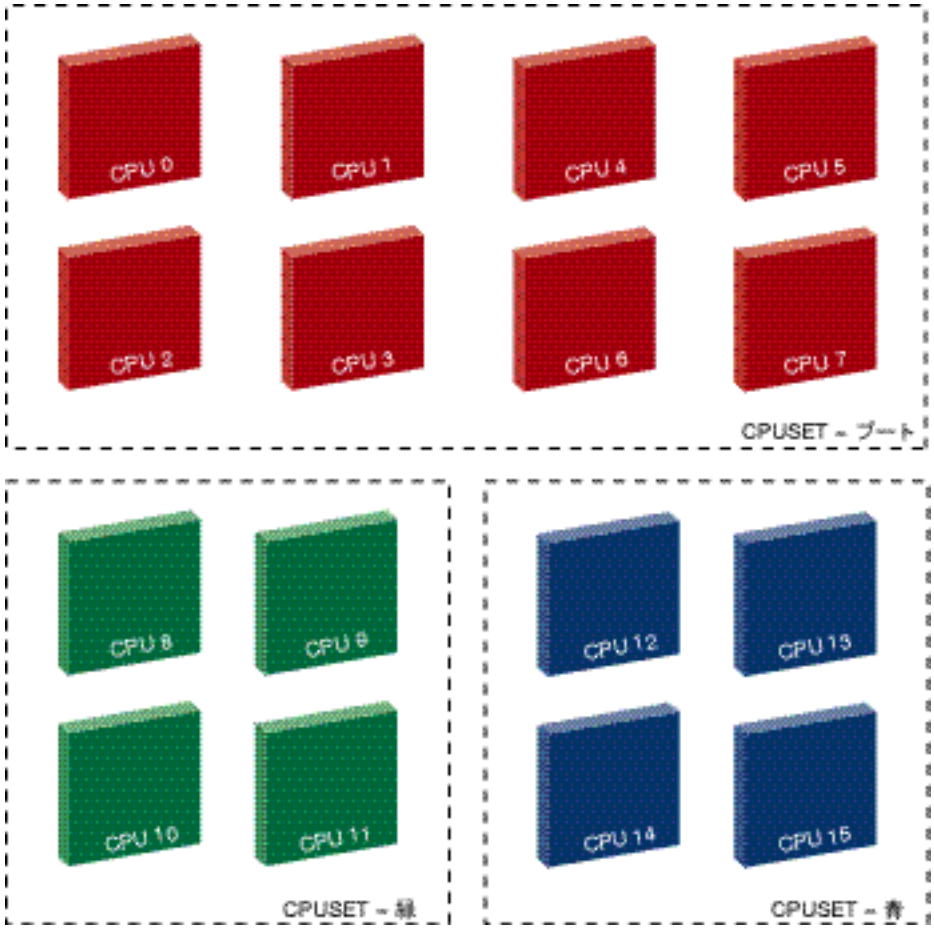

図4-1 cpuset を使用したシステム分割

以下の手順に従い、プロセッサ 16 個のシステムを、49 ページの図4-1 に示すように 3 個の cpuset に分割します。

1. `boot_cpuset.config` というファイルを作成して起動 cpuset を作成し、CPU 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を以下に示します。

```
# boot
MEMORY_LOCAL
MEMORY_MANDATORY

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

boot_cpuset.config ファイルについては、52 ページの「Boot cpuset 」を参照してください。

2. chkconfig(1M) コマンドで -f オプションを指定して、以下の行が含まれる /etc/config/boot_cpuset ファイルを作成します。

   ```
   chkconfig boot_cpuset on
   ```

   /etc/config/boot_cpuset ファイルについては、52 ページの「Boot cpuset 」を参照してください。

   MEMORY_LOCAL フラグと MEMORY_MANDATORY フラグについては、54 ページの「Cpuset 設定ファイル」を参照してください。

   システムが再起動したときに、起動 cpuset が作成されます。

3. Green という名前の専用 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。以下の手順に従ってください。

   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
```

EXCLUSIVE、MEMORY_LOCAL、MEMORY_MANDATORY の各フラグについては、54 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. 以下の 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、54 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 のみで実行されます。MovieMaker のジョブでは、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 のみで実行するアプリケーションを指定します。以下の手順に従ってください。

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 を作成します。-f パラメータで設定ファイルを指定し、-q パラメータで名前を指定します。

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

d. 以下の cpuset(1) コマンドを実行して、Blue 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 のみで実行されます。BookMaker のジョブでは、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。

Boot 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。

メモ: boot_cpuset.so ライブラリは、SGI 2000 および SGI 3000 シリーズの、ccNUMA アーキテクチャに基づくシステムのみで使用できます。

---

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 ファイルは chkconfig(1M) コマンドの規則に従い、以下のような行が含まれます。

```
chkconfig 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 で通常の処理が継続されます。

/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) の設定ファイルと同じ規則に従います。

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

```
# 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、47 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、53 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 の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 に関する情報を取出したり、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 および SGI 3000 シリーズ、すなわちccNUMA アーキテクチャに基づくシステムでは、管理者は cpuset で定義された CPU を含むノードへのメモリ割当てを制限できます。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 のリストを記述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。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。

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、それ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」の範囲で与えられます。mpadmin -n コマンドを実行すると、システム上の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構成を表示できます。CPU の命名規則とシステムのハードウェア構成についての詳細は、『IRIX Admin: System Configuration and Operation』の第 4 章「Configuring the IRIX Operating System」、および mpadmin(1) と 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 やグローバルな 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。

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 内部のノードのみからメモリを割当てようとします。cpuset の外部のメモリからの割当ては、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	この cpuset の外部でメモリが利用できない場合を除き、この cpuset に割当てられていないスレッドは、この 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。(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。)
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制限します。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はこのポリシーがデフォルトです。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することを認めます。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、39 ページの「Miser の有効化と無効化」を参照してください。

Cpuset ライブラリの使用

cpuset ライブラリのインタフェースを使用すると、プログラマは cpuset を作成、破棄したり、既存の cpuset に関する情報を取出したり、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 ライブラリの使用方法については、117ページの「Cpuset システムの API」を参照してください。

Cpuset システムのマン・ページ

man コマンドを使用すると、すべてのリソース管理コマンドについてオンライン・ヘルプを参照できます。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 A117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。
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分離します。
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。
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 にアタッチされているすべての PID のリストを取得します。

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 (CSA: Comprehensive System Accounting)

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、CSA 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、レポートされるデータのスーパーセットです。

CSA は、より詳細で、より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と拡張アカウンティングについての詳細は、『IRIX Admin: Backup, Security and Accounting』マニュアル、第 7 章「System Accounting」の「About the Process Accounting System」と「IRIX Extended Accounting」の節をそれぞれ参照してください。

この章は、次の節で構成されています。

はじめに

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、63 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、66 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は、67 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、74 ページの「CSA に関する詳細」を参照してください。
5. CSA に関するマン・ページのリストについては、108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は、101ページの「CSA レポート」を参照してください。

CSA の概要

CSA は、C プログラムとシェル・スクリプトの集合です。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リング、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。

- ジョブ単位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、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、システム稼働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、任意のジョブ識別子の、1 回のシステム起動期間中の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・データのみをレポートします。対話型ジョブ、`cron` ジョブ、`at` ジョブは、ジョブ内の最後のプロセスが終了するとき、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。

SBU 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 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、92 ページの「SBU」を参照してください。

CSA 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) と 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 csacom(1) と ja(1) が用意されています。csacom コマンドは、CSA の pacct ファイルを読込み、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) と 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、66 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

以下に示す概念と用語は、アカウンティング機能を使用するうえ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、生のアカウンティング・データの解析、整理、レポートを、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、任意のジョブ 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、ワークロード管理リクエスト 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、複数のシステム起動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フェーズが、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・レポートを詳細に分析、レポート、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、生のアカウンティング・データの解析、整理、レポートを、デーモン固有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、生のアカウンティング・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レポートします。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、以下の手順を実行してください。

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 に 0 以外の値を設定することによって、CSA をカーネル内で有効にします。このステップを終了した後でシステムを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したときは、このステップ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に、crontabs ファイルにエントリを追加する方法については、75 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、以下の手順を実行してください。

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 に 0 を設定することによって、CSA をカーネル内で無効にします。このステップを終了した後でシステムを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・ファイルが入っています。/var/adm/acct には、CSA で使用されるデータ収集ファイルが入っています。CSA と IRIX の基本アカウンティングでは、別個の pacct ファイルにアクセスされます。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。

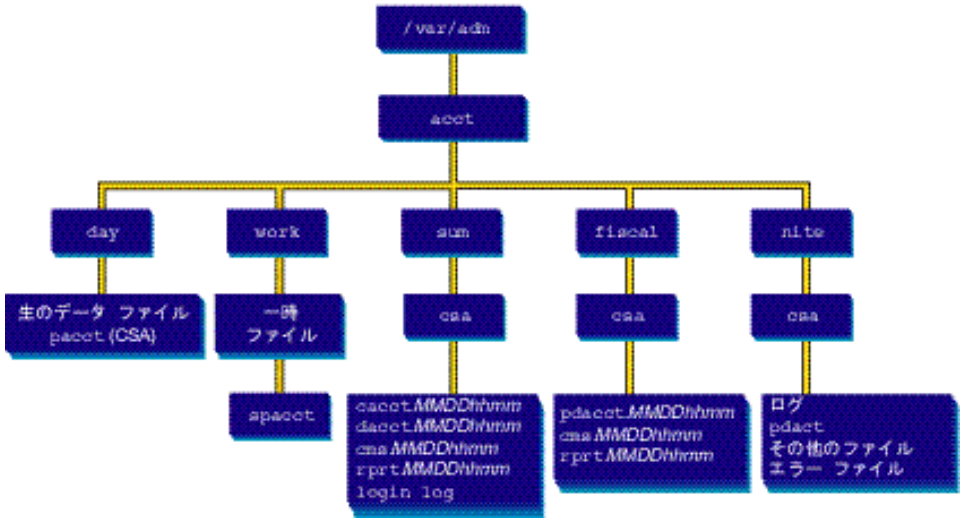

図5-1 /var/adm/acct ディレクトリ

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-分形式のサフィックスが付きます。

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 グループのみに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	現在の生のアカウンティング・データ・ファイルが pacct 形式で入っています。
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 から移動された生データ・ファイルと spacct ファイルが入っています。

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入っています。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*、 /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入っています。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*、 /usr/adm/acct/fiscal/csa/pdacct.*MMDDhhmm*
nite/csa	ログ・ファイル、csarun の状態ファイル、実行日時記録ファイルが含まれます。

/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）
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生のアカウンティング・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やエラー・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（75 ページの「CSA の設定」の、cron の dodisk のエントリを参照）
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たエラー/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たエラー/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たエラー/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) によって生成されたエラー/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たエラー/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たエラー/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) によって生成されたエラー/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たエラー/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たエラー/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たエラー/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たエラー/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たエラー/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たエラー/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た無効なレコード・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たエラー/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)によって生成されたエラー/警告メッセージ
`fd2log`	`csarun` の実行中に作成される診断出力(75 ページの「CSA の設定」の、`cron` の `csarun` のエントリを参照)
`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(75 ページの「CSA の設定」で、`cron` の `csaperiod` のエントリを参照)
`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・スクリプトが入っています。

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・データについてレポートします。
csadrep	デーモンの使用状況についてレポートします。
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、指定され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	異なる種類の CSA のステータスをチェックしたり、有効/無効を切替えたり、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検証します。

/usr/bin ディレクトリには、CSA 関連のユーザ・コマンドがあります。

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 プロセス・アカウンティング・ファイルを検索、出力します。

サイトでアカウンティング・ユーザ出口を使用する場合、/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。

CSA に関する詳細

この節では、CSA の詳細について解説します。以下のトピックについて説明します。

- 74 ページの「日別操作の概要」
- 75 ページの「CSA の設定」
- 78 ページの「csarun コマンド」
- 82 ページの「データ・ファイルの検証と編集」
- 82 ページの「CSA のデータ処理」
- 85 ページの「データのリサイクル」
- 91 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（csaswitch(1M) のマン・ページを参照）コマンドが /etc/rc2 によって呼出されます。
2. デフォルトでは、csa、メモリ、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、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、75 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）。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 を起動することによって課金できます（csachargefee(1M) を参照）。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・レコードにマージされます。

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、91 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。プロセスの優先権に対する細かな調整をする capability 機構についての詳細は、capability(4) と 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、100ページの「スーパー・ユーザ以外による CSA の実行」を参照してください。

1. 必要に応じて、SBU 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 をレポートしたい場合は、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、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、『IRIX 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) にシグナルが送られます。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)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。

ACCT_FS 変数と MIN_BLKS 変数が /etc/csa.conf 設定ファイルで正しく設定されていることを確認します。ACCT_FS は /var/adm/acct を含むファイル・システムです。デフォルトのディレクトリは /var です。MIN_BLKS は、ACCT_FS ファイル・システムで必要な 1K ブロックの最低数を指定します。デフォルトは 2000 個です。

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)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 を次に起動したときに、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 を起動することによって、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、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出力されます。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 が異常終了するか、誤った情報をレポートする可能性があります。

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、レポートされます。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. 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/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。行の先頭文字を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。

     年の通算日 月 日 休日の説明

     年の通算日フィールドには 1～366 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機構が組込まれており、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。81 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、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、ファイルに書き込まれます。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 csarun によって呼出された csabuild 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 を再開できるように、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。

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

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・ファイルは /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きません。

ARCHIVE1　csarun スクリプトの最初のユーザ出口です。/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済みのプログラムは実行されませんが、ユーザ出口スクリプトからは実行できます。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 番目のユーザ出口です。/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済みのプログラムは実行されませんが、ユーザ出口スクリプトからは実行できます。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に、/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 番目のユーザ出口です。/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済みのプログラムは実行されませんが、ユーザ出口スクリプトからは実行できます。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。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 番目のユーザ出口です。/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済みのプログラムは実行されませんが、ユーザ出口スクリプトからは実行できます。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。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 のエントリ状態を示します。

csarun を再開するには、以下の手順に従ってください。

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、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

状態 ARCHIVE2、CMS、REPORT、DREP、または FEF で再開する場合、sorted pacct ファイルが /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・ファイルの検証と編集

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検証し、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レポートします。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。生成される出力ファイルは csaedit(1M) への入力として使用して、pacct 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・ラインを使用して検証します。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検証し、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。データの流れの様子を図5-2 に示します。

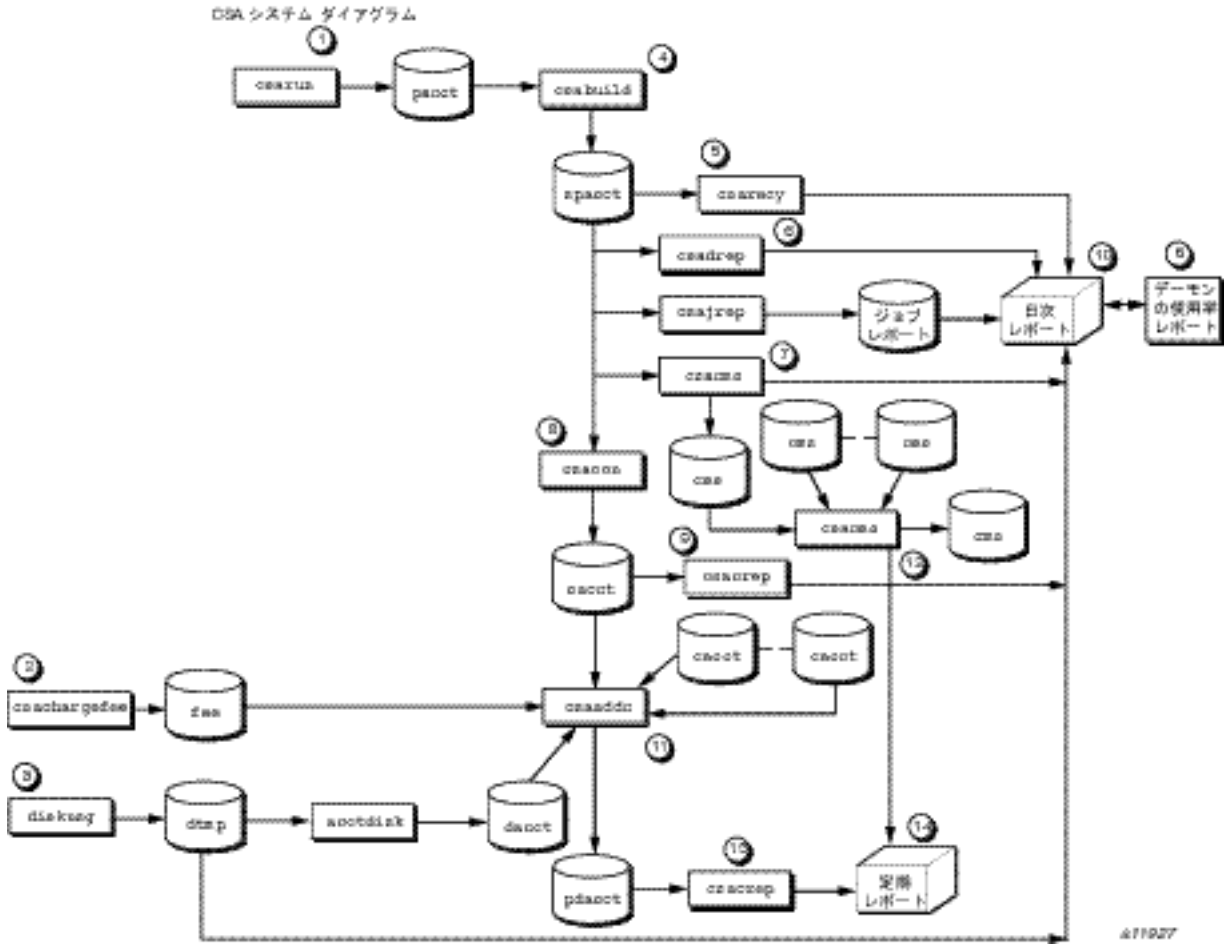

図5-2 CSA のデータ処理

1. 生のアカウンティング・ファイルを生成します。さまざまなデーモンやシステム・プロセスが生の 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とることができます。dodisk は使用量を動的にレポー

トしません。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レポートします。ディスク使用量は、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 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 およびブート時間別にジョブ・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 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き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 85 ページの「データのリサイクル」を参照)。

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、テープのデーモンの使用状況をレポートします。入力は、csabuild(1M) によって作成される 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るバイナリ・ファイルから取得されます。files オペランドはバイナリ・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、ソートしたうえで、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集約します。csacon(1M) コマンドが、sorted pacct ファイルのレコードを集約し、統合されたレコードを 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、raw I/O 待ち時間がレポートされます。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。

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

ステップ 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（ステップ 12～14）は、csaperiod(1m) コマンドによって実行されます。日別アカウンティング、定期アカウンティング、課金とディスク使用量に関するレポートの生成は（ステップ 2～3）、cron(1m) を使用して、定期的に実行されるようにスケジュールできます。詳細については、75 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に、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レポートします。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、csabuild が、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、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されたアカウンティング・データを手動で除去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除去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、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 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 リクエストまたはワークロード管理リクエストの出力を送信すると、そのリクエストは終了します。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 またはワークロード管理のリクエストは、複数の EOJ レコードが関連付けられる場合があります。リクエストの 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、リクエスト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、リクエストに代わって、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。

- 対話型、cron、at のジョブで、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 リクエストで、リクエストの 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リクエストで、リクエストの 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

- 対話型、cron、at のジョブで、システムのクラッシュ時にジョブがアクティブであった場合
- ジョブが、87 ページの「リサイクル・データの削除方法」で説明した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には、不要なリサイクル・データをアカウンティング・システムから除去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、89 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」で説明しているように、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があります。

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。このように終了され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。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。

  csarecy -A の実行例を以下に示します(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、標準出力を ${SUM_DIR}/recyrpt にリダイレクトしないでください。以下のようになります。

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

-A 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 リクエスト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リクエストがあるかどうかを確認します。これには、チェックポイントまたは保留されているリクエスト、キューに入っているリクエスト、待機しているリクエストが含まれます。

すべての NQS リクエストをリストするには、以下のように、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 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。最初のステップが完了したすぐ後に変更された 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 のステップで見つかった、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。このオプションによって、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 によってレポートされ、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レポートされるので、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 が実行されたとき、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 では、現在アクティブなジョブのデータがリサイクルされます。

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前提とします。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レポートされます。

- 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たが、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い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。NQS リクエストやワークロード管理リクエスト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エラーが発生する場合があります。この問題については、90 ページの「NQS リクエストまたはワークロード管理リクエストとリサイクル・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、87 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、将来の課金期間で不当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、将来の課金期間で不当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、将来の課金期間に不当に課金される場合がある。	アクティブなジョブおよびリサイクル・ジョブすべてのデータを含む。
`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、将来の課金期間で不当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 リクエストまたはワークロード管理リクエストとリサイクル・データ

CSA ですべての NQS リクエストまたはワークロード管理リクエストを認識するには、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 リクエストまたはワークロード管理リクエストのリサイクル・データを手動で除去すると、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します。これによって、リクエストは、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（95 ページの「NQS の SBU」または 95 ページの「ワークロード管理の SBU」を参照）。

- リクエストが間違ったキュー・レートで課金されます。
- 間違ったキュー待ち時間がリクエストに関連付けられます。

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除去されたためです。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 リクエストまたはワークロード管理リクエスト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リクエストを受取ったとき
- リクエストがキューに転送されたとき（NQS のみ）
- リクエストが実行されたとき。これには、リクエストを初めて実行する場合、再開する場合、再実行する場合が含まれます。
- リクエストが終了したとき。NQS リクエストは、完了、キューに再指定、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。ワークロード管理リクエストは、完了、キューに再指定、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リクエストの場合、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 リクエストまたはワークロード管理リクエスト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
- 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・パーミッションのないユーザによる CSA の実行の許可
- 代替設定ファイルの使用

SBU

SBU 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)。

プライム/非プライム時間の算出例を以下に示します。

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、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き、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。

以下の例では、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 に同じ値を設定します。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。

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以下に示します。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 値の合計に乗算され、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、P_RWTIME）に乗算され、プロセス・レコードの SBU 値に対する時間的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 時間に乗算されます。
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 時間に乗算されます。
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に乗算されます。
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 待ち時間に乗算されます。
P_RWTIME	ロウ 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 待ち時間に乗算されます。
P_MEM	一般メモリ積算重み係数。P_MEM は、メモリの SBU（P_XMEM と P_VMEM）に乗算され、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重み係数。この重み係数に使用される単位は、1 メガバイト-分あたりの*課金単位*です。P_XMEM は、コア・メモリ積算に乗算されます。

P_VMEM	CPU 時間仮想物理メモリ積算重み係数。この重み係数に使用される単位は、メガバイト-分あたりの*課金単位*です。P_VMEM は仮想メモリ積算に乗算されます。
P_IO	一般-I/O 重み係数。P_IO は、I/O の SBU（P_BIO、P_CIO、P_LIO で構成）に乗算され、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	ブロック転送重み係数。この重み係数に使用される単位は、ブロック転送 1 つあたりの*課金単位*です。P_BIO は I/O 転送ブロック数に乗算されます。
P_CIO	文字転送重み係数。この重み係数に使用される単位は、転送文字 1 つあたりの*課金単位*です。P_CIO はI/O 転送文字数に乗算されます。
P_LIO	論理 I/O 要求重み係数。この重み係数に使用される単位は、論理 I/O 要求 1 つあたりの*課金単位*です。P_LIO は、論理 I/O 要求の数に乗算されます。論理 I/O 要求の数は、read および write システム・コールの総数です。

全プロセス・レコードの SBU の計算式を以下に示します。

```
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（メガバイト-分単位）
virtmem	仮想メモリ積算（メガバイト-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なパラメータが記述されています。

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/SBU ペアの総数は NQS_NUM_QUEUES の値と等しい必要があります)。キュー/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、関連するキュー内の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 NQS 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/SBU ペアの総数は NQS_NUM_MACHINES の値と等しい必要があります)。/etc/csa.conf 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なパラメータが記述されています。

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/SBU ペアの総数は WKMG_NUM_QUEUES の値と等しい必要があります)。キュー/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、関連するキュー内の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/SBU ペアの総数は WKMG_NUM_MACHINES の値と等しい必要があります)。/etc/csa.conf 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、一連の重み係数があります。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

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

すべての時間を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、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。75 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を説明しています。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、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。

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(75 ページの「CSA の設定」を参照)。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、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 の実行は終了します。

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(100ページの「スーパー・ユーザ以外による CSA の実行」を参照)。

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 リクエストの一部に対して課金したくない場合は、/etc/csa.conf 内の該当する NQS_TERM_*xxxx* 変数(終了コード)に 0 を設定します。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、リクエストのすべての部分が課金されます。

以下の表で、終了コードについて説明します。

コード	説明
NQS_TERM_EXIT	リクエストの実行が完了し、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リクエストも、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リクエストについても、このレコードが書込まれます。
NQS_TERM_REQUEUE	チェックポイントされ、NQS シャットダウン時に再度キューに入れられる、実行中のリクエストについて書込まれます。
NQS_TERM_PREEMPT	リクエストが 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されるリクエスト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	リクエストが 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、チェックポイントできず qsub の -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。リクエストはこのステータスで再度キューに入れられます。
NQS_TERM_RERUN	リクエストがオペレータ以外による再実行リクエストである場合に書込まれます。

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
ACCT_FS	/var/adm/acct が常駐するファイル・システム。デフォルトは /var です。
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 の実行時に ${ACCT_FS} に必要な空きブロックの最少数。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に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、97 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

サイトによっては、スーパー・ユーザ・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。プロセスの優先権に対する細かな調整を可能にする capability 機構についての詳細は、capability(4) と capabilities(4) のマン・ページを参照してください。

スーパー・ユーザ・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に設定する手順を以下に示します。この例では、スーパー・ユーザ・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、グループ ID が adm で、グループ adm によって読込み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. 75 ページの「CSA の設定」のステップ 1〜5 に従って、システム課金単位を設定し、システムの起動時刻を記録し、システム・シャットダウンの前の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に設定します。

```
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に設定します。

```
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）。

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

7. 75 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 の場合のみに機能します。

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 に別の設定ファイルを指定して、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。レポートを使用して、システム使用状況の追跡、パフォーマンスのモニタリング、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ます。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。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。

- 102ページの「統合情報レポート」
- 103ページの「未完了ジョブ情報レポート」
- 103ページの「ディスク使用状況レポート」
- 103ページの「コマンド集計レポート」
- 104ページの「最終ログイン・レポート」
- 104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 別にソートされます。以下の使用状況の値は、レポート期間中に、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	CPU 時間 1 分あたりに使用されたコア（物理）メモリの累積（キロバイト単位）
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（キロバイト単位）
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、レポート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（キロバイト単位）
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（キロバイト単位）
TOTAL CPU	使用された CPU 時間の合計（分単位）
TOTAL REAL	使用された実時間 (wall clock) 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（キロバイト単位）
MEAN SIZE KVIRT	使用された仮想メモリの平均量（キロバイト単位）

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）で割った値
K-CHARS READ	読込まれた文字の合計数（キロバイト単位）
K-CHARS WRITTEN	書込まれた文字の合計数（キロバイト単位）
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、最終ログインの日付を示します。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、ワークロード管理、テープのデーモンの使用状況が示されます。このレポートは、レポート期間中に NQS、ワークロード管理、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示されます。

ヘッダ	説明
Job Type	ジョブのタイプ（対話型、NQS、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比率
Tape Jobs	対話型、NQS、ワークロード管理の各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、ワークロード管理、対話型の各ジョブの、CPU 関連の使用状況が示されます。

ヘッダ	説明
Job Type	ジョブのタイプ（対話型、NQS、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、その比率
User CPU Time	CPU 時間のうち、ユーザ CPU 時間に使用された時間と、その比率

テープ使用状況レポートには、NQS、ワークロード管理、対話型の各ジョブの、テープ・アクティビティ関連の使用状況が示されます。

ヘッダ	説明
Job Type	ジョブのタイプ（対話型、NQS、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（キロバイト単位）
KBytes Written	書込まれたテープの量（キロバイト単位）
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。

- 105ページの「統合アカウンティング・レポート」
- 106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

以下に示す統合アカウンティング・レポートの使用状況の値は、レポート期間中に、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（キロバイト単位）
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（キロバイト単位）
IOWAIT BLOCK	ブロック I/O 待ち時間の累積（秒単位）
IOWAIT RAW	ロウ 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、レポート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（キロバイト単位）
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（キロバイト単位）
TOTAL CPU	使用された CPU 時間の合計（分単位）
TOTAL REAL	使用された実時間 (wall clock) 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（キロバイト単位）
MEAN SIZE KVIRT	使用された仮想メモリの平均量（キロバイト単位）
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）で割った値
K-CHARS READ	読込まれた文字の合計数（キロバイト単位）
K-CHARS WRITTEN	書込まれた文字の合計数（キロバイト単位）
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) のマン・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) のマン・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) のマン・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) のマン・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 では、以下の 2 つのファイルがメッセージ・カタログに使用されます。

- /usr/lib/locale/C/LC_MESSAGES/acct.cat
- /usr/lib/locale/C/LC_MESSAGES/acct.exp

このリリースの 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) のマン・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。このリリースの 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・パーティー製のパッケージ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 コマンドで、すべてのリソース管理コマンドについてオンライン・ヘルプを参照できます。マン・ページをオンラインで表示するには、man *コマンド名* 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 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・データについてレポートします。
csadrep(1m)	デーモンの使用状況についてレポートします。
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、指定され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/無効を切替えたり、アカウンティング・ファイルを保守のために切替えたりします。
csaverify(1m)	アカウンティング・レコードが有効であるかどうかを検証します。

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;
```

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)
```

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。

setwaitjobpid

setwaitjobpid 関数は、指定の pid を待ってから waitjob 関数を呼出す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に設定された終了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 setwaitjobpid(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 25 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 25 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、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、それ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。
cpusetDetachAll(3x)	cpuset からすべてのスレッドを分離します。
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 にアタッチされているすべての PID のリストを取得します。
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。

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。
cpusetDetachAll(3x)	cpuset からすべてのスレッドを分離します。
cpusetDestroy(3x)	cpuset を破棄します。

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。ルート・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 内部のノードのみから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 内部のノードのみから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、この cpuset に含まれるノードへのメモリ割当てをすべて制限します。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。これは、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、それ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 cpusetDetachAll(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。

cpusetDetachAll(3x)

名前

cpusetDetachAll - cpuset からすべてのスレッドを分離します。

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分離します。ルート・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 キューから分離します。

```
char *qname = "mpi_set";
               /* Attach to 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。ルート・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。

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。
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);
```

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

cpusetGetCPUList 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。

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 内部のノードのみから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 内部のノードのみから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、この cpuset に含まれるノードへのメモリ割当てをすべて制限します。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、それ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。

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。

この関数を使用して、cpusetAllocQueueDef(3x) 関数の前回の呼出し時に割当てられたメモリを解放する必要があります。

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。

この関数を使用して、cpusetGetCPUList(3x) 関数の前回の呼出し時に割当てられたメモリを解放する必要があります。

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。

この関数を使用して、cpusetGetNameList(3x) 関数または cpusetGetName(3x) 関数の前回の呼出し時に割当てられたメモリを解放する必要があります。

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。

この関数を使用して、cpusetGetPIDList(3x) 関数の前回の呼出し時に割当てられたメモリを解放する必要があります。

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)

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5-1 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5-2 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、前のステップで作成した 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

ふ