リソース管理および Oracle® Solaris ゾーン 開発者ガイド

Part No: E64446
2016 年 11 月

Part No: E64446

ドキュメントのアクセシビリティについて

オラクルのアクセシビリティについての詳細情報は、Oracle Accessibility ProgramのWeb サイト(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc)を参照してください。

Oracle Supportへのアクセス

サポートをご契約のお客様には、My Oracle Supportを通して電子支援サービスを提供しています。詳細情報は(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info) か、聴覚に障害のあるお客様は (http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs)を参照してください。

目次

このドキュメントの使用

- **概要** – コンピュータリソースを管理するためのユーティリティアプリケーション、または自身の使用状態をチェックして適宜調整できるセルフモニタリングアプリケーションの開発に関連して、Oracle Solaris オペレーティングシステムでのリソース管理について説明します。
- **対象読者** – Oracle Solaris オペレーティングシステム用のリソース管理アプリケーションを記述する開発者。
- **必要な知識** – C プログラミングに関する高度な知識。仮想化環境での経験はプラスになります。

製品ドキュメントライブラリ

この製品および関連製品のドキュメントとリソースは http://www.oracle.com/pls/topic/lookup?ctx=E62101-01 で入手可能です。

フィードバック

このドキュメントに関するフィードバックを http://www.oracle.com/goto/docfeedback からお聞かせください。

♦ ♦ ♦ 第 1 章

Oracle Solaris オペレーティングシステムでのリソース管理

リソース管理は、コンピュータリソースを管理するためのユーティリティアプリケーションや、自身の使用状態をチェックして適宜調整できるセルフモニタリングアプリケーションを記述する開発者に役立ちます。この章では、Oracle Solaris オペレーティングシステムでのリソース管理の概要を示します。内容は次のとおりです。

Oracle Solaris オペレーティングシステムでのリソース管理の理解

リソース管理の主な概念は、システムが効率的に機能するにはサーバー上の作業負荷のバランスがとれていなければならないということです。リソース管理が適切に行われないと、誤って暴走した作業負荷によって進行が中断したり、優先度の高いジョブが不要に遅延したりする可能性があります。効率的なリソース管理を行えば、組織はシステムを統合して経費を節減することもできます。

Oracle Solaris オペレーティングシステムは、作業負荷およびリソースを整理するための構造を提供し、特定の作業負荷が消費できるリソースの量を定義するための制御を可能にします。システム管理者の視点から見たリソース管理の詳しい解説は、『Oracle Solaris 11.3 でのリソースの管理』 の 第 1 章,「リソース管理の紹介」を参照してください。

リソース管理の作業負荷の整理

作業負荷の基本単位はプロセスです。システム全体を通して順にプロセス ID (PID) が振られます。デフォルトでは、システム管理者によって各ユーザーがプロジェクトに

割り当てられ、これがネットワーク全体の管理識別子となります。プロジェクトへのログインが成功するたびに、プロセスのグループ化メカニズムであるタスクが新しく作成されます。タスクには、ログインプロセスとそれに続く子プロセスが含まれます。

プロジェクトおよびタスクの詳細は、『Oracle Solaris 11.3 でのリソースの管理』 の 第 2 章,「プロジェクトとタスクについて」 (システム管理者の視点について) またはこのドキュメントの第2章「ワークロード階層のプロジェクトとタスク」 (開発者の視点について) を参照してください。

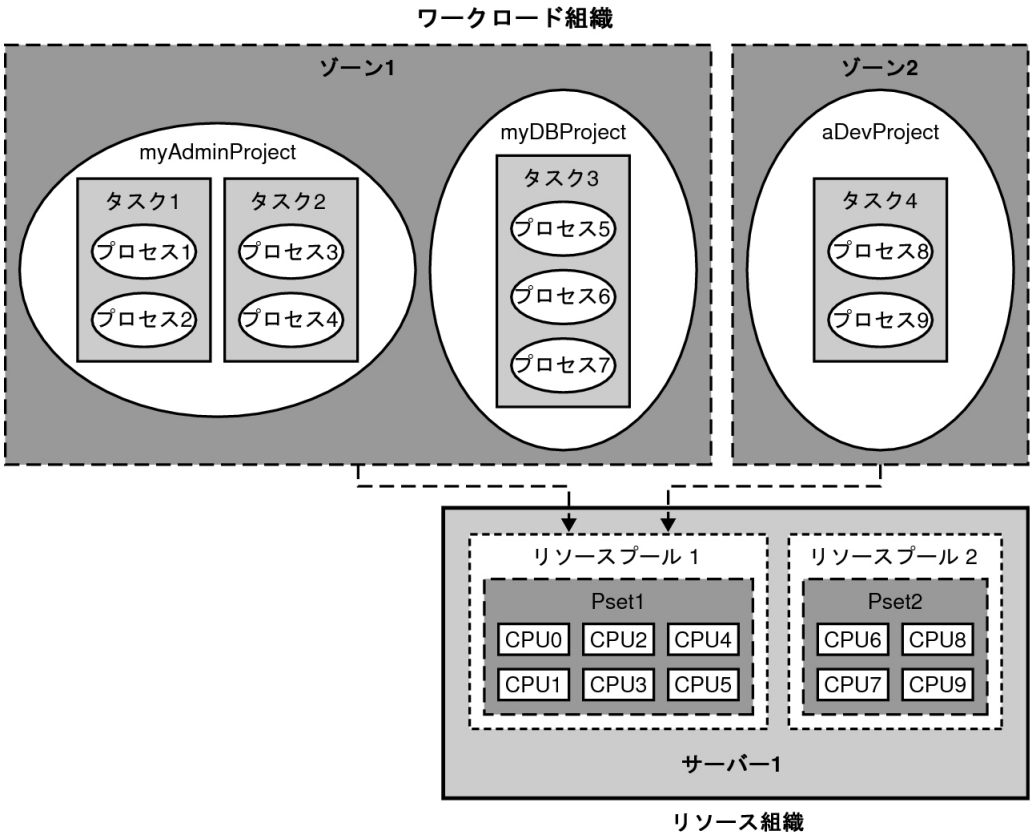
プロセスはオプションで非大域ゾーンにグループ化でき、これはセキュリティー目的でプロセスを分離するためにシステム管理者によって設定されます。ゾーンを使用すると、1 つまたは複数のアプリケーションを、システム上のほかのすべてのアプリケーションから分離した状態で実行できます。非大域ゾーンについては、『Oracle Solaris ゾーンの作成と使用』で詳しく説明しています。ゾーンで実行するリソース管理アプリケーションを記述するための特別な注意事項については、第7章「Oracle Solaris ゾーンでのリソース管理アプリケーションに関する設計上の考慮事項」を参照してください。

リソースの整理

システム管理者は、特定の CPU やシステム内で定義された CPU グループに作業負荷を割り当てることができます。CPU はプロセッサセット (*pset*) にグループ化できます。さらに `pset` は 1 つまたは複数のスレッドスケジューリングクラスに関連付けることができ、これによってリソースプールに CPU の優先度が定義されます。リソースプールは、ユーザーがシステムリソースを利用できるようにするための便利なメカニズムをシステム管理者に提供します。『Oracle Solaris 11.3 でのリソースの管理』 の 第 12 章,「リソースプールについて」では、リソースプールについてシステム管理者向けに説明します。プログラミングの考慮事項は、第6章「リソースプール」で説明します。

次の図は、作業負荷およびコンピュータリソースが Oracle Solaris オペレーティングシステムでどのように整理されるかを示しています。

図 1 Oracle Solaris オペレーティングシステムでの作業負荷とリソースの整理



リソース制御

ユーザーが消費するリソースの量を管理する場合、作業負荷単位をリソース単位に割り当てるだけでは不十分です。リソースを管理するために、Oracle Solaris オペレーティングシステムにはフラグ、アクション、およびシグナルのセットが用意されており、これらをまとめてリソース制御と呼びます。リソース制御は、 zonecfg(1M) で説明する `zonecfg` コマンドを使用して `/etc/project` ファイルまたはゾーンの構成内に格納されます。たとえば、公平配分スケジューラ (FSS) は、作業負荷に指定された重要度係数に基づき、作業負荷間で CPU リソースの配分を割り当てることができます。これらのリソース制御を使用して、システム管理者は特定のゾーン、プロジェクト、タスク、またはプロセスに特権レベルおよび制限の定義を設定できます。システム管

理者がリソース制御を使用する方法については、『Oracle Solaris 11.3 でのリソースの管理』 の 第 6 章,「リソース制御について」を参照してください。プログラミングの考慮事項は、第5章「リソース制御」を参照してください。

拡張アカウンティング機能

作業負荷とリソースの整理に加え、Oracle Solaris オペレーティングシステムには、システムリソースの使用率をモニターして記録するための拡張アカウンティング機能が用意されています。拡張アカウンティング機能は、プロセスとタスクに関するリソース消費の詳細な統計をシステム管理者に提供します。

この機能については、『Oracle Solaris 11.3 でのリソースの管理』 の 第 4 章,「拡張アカウンティングについて」でシステム管理者向けに詳しく説明しています。Oracle Solaris オペレーティングシステムでは、開発者が拡張アカウンティング機能に C インタフェースと Perl インタフェースの両方を使用できます。C インタフェースについては 第3章「拡張アカウンティングに C インタフェースを使用」を、Perl インタフェースについては 第4章「拡張アカウンティングに Perl インタフェースを使用」を参照してください。

リソース管理アプリケーションの記述

このマニュアルでは、開発者の視点から見たリソース管理に焦点を当て、次の種類のアプリケーションを記述するための情報を示します。

- リソース管理アプリケーション — リソースの割り当て、パーティションの作成、ジョブのスケジューリングなどのタスクを実行するユーティリティー。
- リソースモニタリングアプリケーション – `kstats` を使用してシステム統計をチェックし、システム、作業負荷、プロセス、およびユーザーごとにリソースの使用率を確認するアプリケーション。
- リソースアカウンティングユーティリティー – 分析、課金、および容量計画のためのアカウンティング情報を提供するアプリケーション。
- 自己調整アプリケーション – 自身のリソースの使用を判断し、必要に応じて消費を調整できるアプリケーション。

♦ ♦ ♦ 第 2 章

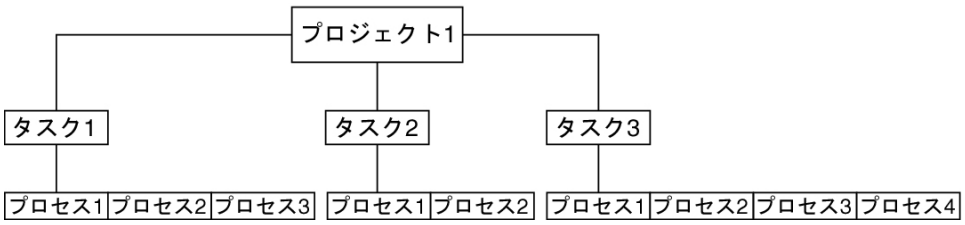
2 ワークロード階層のプロジェクトとタスク

この章では、ワークロード階層について説明し、プロジェクトおよびタスクに関する情報を提供します。内容は次のとおりです。

プロジェクトおよびタスクの概要

Oracle Solaris オペレーティングシステムは、システム上で実行されている作業を編成するためにワークロード階層を使用します。タスクはプロセスの集まりであり、ワークロードのコンポーネントを表します。プロジェクトはタスクの集まりであり、ワークロード全体を表します。任意の時点で、1 つのプロセスは 1 つのタスクと 1 つのプロジェクトのみのコンポーネントになることができます。次の図に、ワークロード階層内の関係を示します。

図 2 ワークロード階層



複数のプロジェクトのメンバーであるユーザーは、同時に複数のプロジェクトのプロセスを実行できます。

プロセスによって起動されるプロセスはすべて、親プロセスによって作成されたプロジェクトとタスクを継承します。起動スクリプトで新しいプロジェクトに切り替えると、すべての子プロセスが新しいプロジェクトで実行されます。

実行中のユーザープロセスには、ユーザー ID (uid)、グループ ID (gid)、およびプロジェクト ID (projid) が関連付けられています。プロセスの属性と機能は、ユーザー、グループ、およびプロジェクト ID から継承されて、タスクの実行コンテキストを形成します。

プロジェクトおよびタスクの詳細は、『Oracle Solaris 11.3 でのリソースの管理』 の 第 2 章, 「プロジェクトとタスクについて」を参照してください。プロジェクトおよびタスクを管理するための管理コマンドについては、『Oracle Solaris 11.3 でのリソースの管理』 の 第 3 章, 「プロジェクトとタスクの管理」を参照してください。

/etc/project ファイル

project ファイルはワークロード階層の中心的な機能です。project データベースは、/etc/project ファイルを介してシステム上で、または NIS や LDAP などのネーミングサービスを介してネットワーク上で保持されます。

/etc/project ファイルには 5 つの標準プロジェクトが含まれます。

system	このプロジェクトは、すべてのシステムプロセスおよびデーモンに使用されます。
user.root	root のログインおよび root の cron、at、および batch ジョブにより発生する、すべての root プロセス。
noproject	この特別なプロジェクトは IPQoS 用です。
default	各ユーザーにデフォルトプロジェクトが割り当てられます。
group.staff	このプロジェクトはグループ staff のすべてのユーザーに使用されます。

プログラムからプロジェクトファイルにアクセスするには、次の構造体を使用します。

```
struct project {
  char      *pj_name;       /* name of the project */
  projid_t   pj_projid;     /* numerical project ID */
  char      *pj_comment;    /* project comment */
  char     **pj_users;      /* vector of pointers to project user names */
  char     **pj_groups;     /* vector of pointers to project group names */
  char      *pj_attr;       /* project attributes */
};
```

project 構造体のメンバーには、次のものがあります。

*pj_name
　プロジェクトの名前。

pj_projid
　プロジェクト ID。

*pj_comment
　ユーザー指定のプロジェクトの説明。

**pj_users
　プロジェクトのユーザーメンバーのポインタ。

**pj_groups
　プロジェクトのグループメンバーのポインタ。

*pj_attr
　プロジェクトの属性。これらの属性は、リソース制御とプロジェクトプールの値を設定するために使用します。

リソース使用率は、プロジェクトの属性を介して制御するか、ゾーンの場合は zonecfg コマンドを使用して構成できます。リソース制御属性のタイプは 4 つの接頭辞を使用してグループ化されています。

- project.* – この接頭辞は、プロジェクトを制御するために使用される属性を示します。たとえば、project.max-locked-memory は、ロックされるメモリーの許容合計量をバイト数で示します。project.pool 属性は、プロジェクトをリソースプールにバインドします。第6章「リソースプール」を参照してください。
- task.* – この接頭辞は、タスクに適用される属性に使用されます。たとえば、task.max-cpu-time 属性は、このタスクのプロセスで使用できる最大 CPU 時間を秒数で表現します。
- process.* – この接頭辞は、プロセス制御に使用されます。たとえば、process.max-file-size 制御は、このプロセスでの書き込みに使用できる最大ファイルオフセットをバイト数で表現します。
- zone.* – zone.* 接頭辞は、ゾーン内のプロジェクト、タスク、およびプロセスに適用されるゾーン規模のリソース制御を示します。たとえば、zone.max-lwps は、あるゾーンの LWP の数が増えすぎてほかのゾーンに影響を与えることを防ぎます。ゾーンの LWP の合計数は、project.max-lwps エントリを使用すると、ゾーン内のプロジェクト間でさらに再分割できます。

リソース制御の完全なリストについては、resource-controls(5) を参照してください。

プロジェクトおよびタスクの API 関数

プロジェクトで作業する開発者を支援するために、次の関数が用意されています。これらの関数では、`project` データベース内のユーザープロジェクトを記述するエントリが使用されます。

これらの関数の詳細は、『man pages section 3: Extended Library Functions, Volume 3』を参照してください。

endprojent (3PROJECT)	処理が完了すると、プロジェクトデータベースをクローズし、リソースの割り当てを解除します。
fgetprojent (3PROJECT)	プロジェクトデータベース内のエントリを含む構造体へのポインタを返します。`fgetprojent()` は、`nsswitch.conf` を使用するのではなく、ストリームから行を読み取ります。
getdefaultproj (3PROJECT)	プロジェクトキーワードの有効性を確認し、プロジェクトを探し、見つかった場合はプロジェクト構造へのポインタを返します。
getprojbyid (3PROJECT)	プロジェクト ID を指定する番号を使用して、`project` データベースでエントリを検索します。
getprojbyname (3PROJECT)	プロジェクト名を指定する文字列を使用して、`project` データベースでエントリを検索します。
getprojent (3PROJECT)	プロジェクトデータベース内のエントリを含む構造体へのポインタを返します。
inproj(3PROJECT)	指定されたユーザーが、指定されたプロジェクトの使用を許可されているかどうかを確認します。
setproject (3PROJECT)	呼び出し元プロセスは、ターゲットプロジェクト内に新しいタスクを作成することによって、ターゲットプロジェクトに参加します。
setprojent (3PROJECT)	繰り返し検索を可能にするために `project` データベースを巻き戻します。

リエントラント関数

`getprojent()`、`getprojbyname()`、`getprojbyid()`、`getdefaultproj()`、および `inproj()` は、呼び出し元から提供されたバッファーを使用して、返される結果を格納します。これらの関数は、シングルスレッドアプリケーションとマルチスレッドアプリケーションの両方で安全に使用できます。

リエントラント関数は 3 つの追加引数を必要とします。

- proj
- buffer
- bufsize

proj 引数は、呼び出し元によって割り当てられた project 構造体へのポインタである必要があります。正常完了時に、これらの関数はこの構造体のプロジェクトのエントリを返します。project 構造体が参照するストレージは、buffer 引数で指定されたメモリーから割り当てられます。bufsize は、サイズをバイト数で指定します。

不正なバッファーサイズが使用されている場合、getprojent() は NULL を返し、errno を ERANGE に設定します。

project データベースのエントリにアクセスするためのコード例

例 1 project データベースの各エントリの最初の 3 つのフィールドを出力する

この例では、次の重要な点に注意してください。

- setprojent() は、最初から開始するために project データベースを巻き戻します。
- getprojent() は、project.h で定義されている保守的な最大バッファーサイズで呼び出されます。
- endprojent() は、project データベースをクローズし、リソースを解放します。

```
#include <project.h>

struct project projent;
char buffer[PROJECT_BUFSZ]; /* Use safe buffer size from project.h */
 ...
struct project *pp;

setprojent();  /* Rewind the project database to start at the beginning */

while (1) {
   pp = getprojent(&projent, buffer, PROJECT_BUFSZ);
   if (pp == NULL)
          break;
    printf("%s:%d:%s\n", pp->pj_name, pp->pj_projid, pp->pj_comment);
  ...
};

endprojent();   /* Close the database and free project resources */
```

例 2 呼び出し元のプロジェクト ID に一致する `project` データベースエントリを取得する

次の例では、`getprojbyid()` を呼び出して、呼び出し元のプロジェクト ID に一致するプロジェクトデータベースエントリを取得します。この例では、次にプロジェクト名とプロジェクト ID を出力します。

```
#include <project.h>

struct project *pj;
char buffer[PROJECT_BUFSZ]; /* Use safe buffer size from project.h */

main()
{
   projid_t pjid;
   pjid = getprojid();
   pj = getprojbyid(pjid, &projent, buffer, PROJECT_BUFSZ);
   if (pj == NULL) {
       /* fail; */
   }
   printf("My project (name, id) is (%s, %d)\n", pp->pj_name, pp->pj_projid);
}
```

プロジェクトとタスクに関連するプログラミングの問題

アプリケーションを記述する場合は、次の問題を考慮してください。

- 新しいプロジェクトを明示的に作成する関数は存在しません。
- `project` データベースにユーザーのデフォルトプロジェクトが存在しない場合、ユーザーはログインできません。
- ユーザーのログイン時に、ユーザーのデフォルトプロジェクトに新しいタスクが作成されます。
- プロセスがプロジェクトに参加すると、そのプロジェクトのリソース制御およびプールの設定がプロセスに適用されます。
- `setproject()` には特権が必要です。プロセスを所有している場合、`newtask` コマンドには特権は必要ありません。どちらの方法でもタスクを作成できますが、`newtask` のみが実行中のプロセスのプロジェクトを変更できます。
- タスクの間に親/子関係は存在しません。
- ファイナライズ済みタスクを作成するには、`newtask -F` を使用するか、`setproject()` を使用して呼び出し元を新しいプロジェクトに関連付けます。ファイナライズ済みタスクは、リソースアカウンティングの集計を正確に見積もりたい場合に役立ちます。
-

♦ ♦ ♦ 第 3 章

3 拡張アカウンティングに C インタフェースを使用

この章では、拡張アカウンティングの C インタフェースについて説明します。内容は次のとおりです。

拡張アカウンティングの C インタフェースの概要

拡張アカウンティングサブシステムは、システム上で実行されている作業負荷によるリソース消費をモニターします。拡張アカウンティングは、作業負荷タスクおよびプロセスのアカウンティングレコードを生成します。

拡張アカウンティングの概要および拡張アカウンティングを管理する手順の例については、『Oracle Solaris 11.3 でのリソースの管理』 の 第 4 章,「拡張アカウンティングについて」および『Oracle Solaris 11.3 でのリソースの管理』 の 第 5 章,「拡張アカウンティングの管理のタスク」を参照してください。

拡張アカウンティングのフレームワークはゾーンに対して拡張されました。各ゾーンには、タスクおよびプロセスをベースとしたアカウンティング用に独自の拡張アカウンティングファイルが含まれています。大域ゾーン内の拡張アカウンティングファイルには、大域ゾーンおよびすべての非大域ゾーンのアカウンティングレコードが含まれています。アカウンティングレコードには、大域ゾーンの管理者が大域ゾーン内のアカウンティングファイルからゾーンごとのアカウンティングデータを抽出する際に使用できるゾーン名タグが含まれています。

拡張アカウンティング API 関数

拡張アカウンティング API には、次のようなアクションを実行する関数が含まれています。

- exacct システムコール
- exacct ファイルの操作
- exacct オブジェクトの操作

このセクションでは、これらの関数を説明する表を示します。関数名は、そのマニュアルページへのリンクになっています。

exacct システムコール

次の表は、拡張アカウンティングサブシステムと対話するシステムコールを示しています。

表 1 拡張アカウンティングのシステムコール

関数	説明
putacct(2)	アカウンティングレコードにプロセス固有の追加データをタグ付けする機能を特権付きプロセスに提供します
getacct(2)	特権付きプロセスが現在実行中のタスクおよびプロセスのために拡張アカウンティングのバッファーをカーネルに要求できるようにします
wracct(2)	指定されたタスクまたはプロセスのリソース使用率データを書き込むようにカーネルに要求します

exacct ファイルの操作

次の表は、exacct ファイルへのアクセスを提供する関数を示しています。

表 2 exacct ファイルの関数

関数	説明
ea_open(3EXACCT)	exacct ファイルを開きます。
ea_close(3EXACCT)	exacct ファイルを閉じます。
ea_get_object(3EXACCT)	オブジェクトグループをはじめて使用すると、ea_object_t 構造体にデータが読み込まれます。その後グループを使用すると、グループ内のオブジェクト間で繰り返されます。
ea_write_object(3EXACCT)	指定されたオブジェクトを、開いている exacct ファイルに追加します。
ea_next_object(3EXACCT)	基本フィールド (eo_catalog および eo_type) を ea_object_t 構造体に読み込み、レコードの先頭に戻します。
ea_previous_object(3EXACCT)	exacct ファイル内の 1 つのオブジェクトをスキップで戻し、基本フィールド (eo_catalog and eo_type) を ea_object_t に読み込みます。
ea_get_hostname(3EXACCT)	exacct ファイルが作成されたホストの名前を取得します。
ea_get_creator(3EXACCT)	exacct ファイルの作成者を調べます。

exacct オブジェクトの操作

次の表は、exacct オブジェクトへのアクセスに使用する関数を示しています。

表 3 exacct オブジェクトの関数

関数	説明
ea_set_item(3EXACCT)	exacct オブジェクトを割り当ててその値を設定します。
ea_set_group(3EXACCT)	exacct オブジェクトのグループの値を設定します。
ea_match_object_catalog(3EXACCT)	exacct オブジェクトのマスクをチェックして、そのオブジェクトに固有のカタログタグがあるかどうかを確認します。
ea_attach_to_object(3EXACCT)	指定された exacct オブジェクトに exacct オブジェクトを追加します。
ea_attach_to_group(3EXACCT)	exacct オブジェクトのチェーンを、指定されたグループのメンバー項目として追加します。
ea_free_item(3EXACCT)	指定された exacct オブジェクト内の value フィールドを解放します。
ea_free_object(3EXACCT)	指定された exacct オブジェクトと、関連付けられているすべてのオブジェクト階層を解放します。

拡張アカウンティングメモリー管理

次の表は、拡張アカウンティングメモリー管理に関連する関数を示しています。

表 4 拡張アカウンティングメモリー管理の関数

マニュアルページへのリンク	説明
ea_pack_object(3EXACCT)	exacct オブジェクトを、アンパックされた (メモリー内) 表現からパックされた (ファイル内) 表現に変換します。
ea_unpack_object(3EXACCT)	exacct オブジェクトを、パックされた (ファイル内) 表現からアンパックされた (メモリー内) 表現に変換します。
ea_strdup(3EXACCT)	ea_object_t 構造体内に格納する文字列を複製します。
ea_strfree(3EXACCT)	ea_strdup() によって以前コピーされた文字列を解放します。
ea_alloc(3EXACCT)	要求されたサイズのメモリーブロックを割り当てます。このブロックは libexacct 関数に安全に渡すことができ、あらゆる ea_free 関数によって安全に解放できます。
ea_free(3EXACCT)	ea_alloc() によって以前割り当てられたメモリーブロックを解放します。
ea_free_object(3EXACCT)	オブジェクト階層内の可変長データを解放します。
ea_free_item(3EXACCT)	EUP_ALLOC が指定された場合、指定されたオブジェクトの値フィールドを解放します。オブジェクトは解放さ

マニュアルページへのリンク	説明
	れません。ea_free_object() は、指定されたオブジェクトと、関連付けられているすべてのオブジェクト階層を解放します。フラグ引数が EUP_ALLOC に設定されている場合、ea_free_object() はオブジェクト階層内の可変長データもすべて解放します。フラグ引数が EUP_NOALLOC に設定されている場合、ea_free_object() は可変長データを解放しません。特に、これらのフラグは ea_unpack_object(3EXACCT) への呼び出しで指定されたフラグに対応するようにしてください。
ea_copy_object(3EXACCT)	ea_object_t をコピーします。ソースオブジェクトがチェーンの一部である場合は、現在のオブジェクトだけがコピーされます。ソースオブジェクトがグループの場合は、グループオブジェクトだけがコピーされ、メンバーリストは含まれません。グループオブジェクトの eg_nobjs フィールドと eg_objs フィールドは、それぞれ 0 と NULL に設定されます。グループまたは項目リストを再帰的にコピーするには、ea_copy_tree() を使用します。
ea_copy_object_tree(3EXACCT)	ea_copy_object_tree は ea_object_t を再帰的にコピーします。eo_next リスト内のすべての要素がコピーされます。すべてのグループオブジェクトが再帰的にコピーされます。返されたオブジェクトは、ea_free_object(3EXACCT) で EUP_ALLOC フラグを指定することで完全に解放できます。
ea_get_object_tree()	nobj の最上位オブジェクトをファイルから読み取り、本来は ea_write_object() に渡されるのと同じデータ構造体を返します。グループオブジェクトが検出されると、ea_get_object() はグループのグループヘッダー部分だけを読み取ります。ea_get_object_tree() はグループおよびそのすべてのメンバー項目を読み取り、必要に応じてサブレコードへの再帰呼び出しを行います。返されたオブジェクトデータ構造体は、ea_free_object() で EUP_ALLOC フラグを指定することで完全に解放できます。

拡張アカウンティングのその他の操作

これらの関数はその他の操作に関連しています。

- ea_error(3EXACCT) – 拡張アカウンティングライブラリ libexacct のいずれかの関数の呼び出しによって記録された最後の失敗のエラー値を返します。
- ea_match_object_catalog(3EXACCT) – obj で指定された exaact オブジェクトに、catmask で指定されたマスクと一致するカタログタグが含まれている場合は TRUE を返します。

exacct ファイルにアクセスするための C コードの例

このセクションでは、exacct ファイルにアクセスするためのコード例を示します。

例 3 指定された pid の exacct データの表示

この例は、特定の pid の exacct データをカーネルのスナップショットで示しています。

```
...
  ea_object_t *scratch;
  int unpk_flag = EUP_ALLOC;  /* use the same allocation flag */
                              /* for unpack and free */

  /* Omit return value checking, to keep code samples short */

  bsize = getacct(P_PID, pid, NULL, 0);
  buf = malloc(bsize);

  /* Retrieve exacct object and unpack */
  getacct(P_PID, pid, buf, bsize);
  ea_unpack_object(&scratch, unpk_flag, buf, bsize);

  /* Display the exacct record */
  disp_obj(scratch);
  if (scratch->eo_type == EO_GROUP) {
        disp_group(scratch);
  }
  ea_free_object(scratch, unpk_flag);
  ...
```

例 4 カーネルビルド中の個々のタスクの識別

この例では、カーネルビルドを評価し、このタスクによってビルドされるソースツリーの一部を表す文字列を示します。ソースディレクトリごとの分析に役立てるために、ビルドされるソースの一部を示します。

この例について次の重要なポイントに注意してください。

- make の時間の集約には多くのプロセスが含まれることがあるため、各 make はタスクとして開始されます。make の子プロセスは別のタスクとして作成されます。makefile ツリー全体で集約するには、タスクの親子関係を識別する必要があります。
- タスクの exacct ファイルにこの情報を含むタグを追加します。このタスクの make 操作でビルドされるソースツリーの一部を表す現在の作業ディレクトリの文字列を追加します。

```
   ea_set_item(&cwd, EXT_STRING | EXC_LOCAL | MY_CWD,
                                  cwdbuf, strlen(cwdbuf));

  ...
/* Omit return value checking and error processing */
/* to keep code sample short */
ptid = gettaskid(); /* Save "parent" task-id */
tid = settaskid(getprojid(), TASK_NORMAL); /* Create new task */

/* Set data for item objects ptskid and cwd */
ea_set_item(&ptskid, EXT_UINT32 | EXC_LOCAL | MY_PTID, &ptid, 0);
ea_set_item(&cwd, EXT_STRING | EXC_LOCAL | MY_CWD, cwdbuf, strlen(cwdbuf));
```

```
/* Set grp object and attach ptskid and cwd to grp */
ea_set_group(&grp, EXT_GROUP | EXC_LOCAL |  EXD_GROUP_HEADER);
ea_attach_to_group(&grp, &ptskid);
ea_attach_to_group(&grp, &cwd);

/* Pack the object and put it back into the accounting stream */
ea_buflen = ea_pack_object(&grp, ea_buf, sizeof(ea_buf));
putacct(P_TASKID, tid, ea_buf, ea_buflen, EP_EXACCT_OBJECT);

/* Memory management: free memory allocate in ea_set_item */
ea_free_item(&cwd, EUP_ALLOC);
  ...
```

例 5 システムの exacct ファイルの内容の読み取りと表示

この例では、プロセスまたはタスク用にシステムの exacct ファイルの読み取りまたは表示を行う方法を示します。

この例について次の重要なポイントに注意してください。

- ファイル内の次のオブジェクトを取得するには、ea_get_object() を呼び出します。EOF が exacct ファイルの完全なトラバーサルを有効にするまで ea_get_object() をループで呼び出します。
- catalog_name() は、catalog_item 構造体を使用して、Oracle Solaris カタログのタイプ ID をオブジェクトデータの内容を表す意味のある文字列に変換します。タイプ ID は最下位の 24 ビットつまり 3 バイトをマスキングすることで取得されます。

  ```
  switch(o->eo_catalog & EXT_TYPE_MASK) {
    case EXT_UINT8:
        printf(" 8: %u", o->eo_item.ei_uint8);
        break;
    case EXT_UINT16:
    ...
  }
  ```

- TYPE_MASK の上位 4 ビットは、データ型を調べてオブジェクトの実際のデータを出力するために使用されます。
- disp_group() は、グループオブジェクトへのポインタとグループ内のオブジェクト数を取得します。グループ内の各オブジェクトに対して、disp_group() は disp_obj() を呼び出し、オブジェクトがグループオブジェクトの場合は disp_group() を再帰的に呼び出します。

```
/* Omit return value checking and error processing */
/* to keep code sample short */
main(int argc, char *argv)
{
  ea_file_t ef;
  ea_object_t scratch;
  char *fname;

  fname = argv[1];
  ea_open(&ef, fname, NULL,  EO_NO_VALID_HDR, O_RDONLY, 0);
  bzero(&scratch, sizeof (ea_object_t));
```

```
  while (ea_get_object(&ef, &scratch)  != -1) {
      disp_obj(&scratch);
      if (scratch.eo_type == EO_GROUP)
          disp_group(&ef, scratch.eo_group.eg_nobjs);
      bzero(&scratch, sizeof (ea_object_t));
  }
  ea_close(&ef);
}

struct catalog_item {   /* convert Oracle Solaris catalog's type ID */
                        /* to a meaningful string */
  int type;
  char *name;
 } catalog[] = {
  { EXD_VERSION,  "version\t" },
  ...
  { EXD_PROC_PID, "  pid\t" },
  ...
   };

 static char *
 catalog_name(int type)
 {
  int i = 0;
  while (catalog[i].type != EXD_NONE) {
   if (catalog[i].type == type)
    return (catalog[i].name);
   else
    i++;
  }
  return ("unknown\t");
 }

 static void disp_obj(ea_object_t *o)
 {
  printf("%s\t", catalog_name(o->eo_catalog & 0xffffff));
  switch(o->eo_catalog & EXT_TYPE_MASK) {
  case EXT_UINT8:
   printf(" 8: %u", o->eo_item.ei_uint8);
   break;
  case EXT_UINT16:
  ...
 }
 static void disp_group(ea_file_t *ef, uint_t nobjs)
 {
  for (i = 0; i < nobjs; i++) {
   ea_get_object(ef, &scratch));
               disp_obj(&scratch);
   if (scratch.eo_type == EO_GROUP)
                disp_group(ef, scratch.eo_group.eg_nobjs);
        }
 }
```

exacct ファイルに関するプログラミングの問題

- メモリー管理に関する次の問題に注意してください。
  - `ea_free_object()` と `ea_unpack_object()` には同じ割り当てフラグを使用します。

- 文字列オブジェクトの場合、ea_set_item() によって割り当てが行われるため、続けて ea_free_item(*obj*, EUP_ALLOC) を使用して内部ストレージを解放するようにしてください。
- ea_pack_object() および getacct() が使用するサイズはゼロです。サイズを取得するには、getacct() を 2 回呼び出すようにしてください。1 回目は NULL バッファーを指定し、2 回目の呼び出しで渡されるバッファーのサイズを指定します。24 ページの「exacct ファイルにアクセスするための C コードの例」の例 3-1 を参照してください。

- exacct ファイルの内容変更に対して堅牢であるために、アプリケーションはシステムによって生成された exacct ファイル内の未知の exacct レコードをスキップする必要があります。
- アプリケーション固有のレコードを作成するには、カスタマイズアカウンティング用の EXC_LOCAL を使用します。一般的なトレース機能またはデバッグ機能として libexacct を使用します。
  - <sys/exacct_catalog.h> を参照してください。
  - ea_catalog_t のデータ ID フィールドはカスタマイズできます。

♦ ♦ ♦ 第 4 章

拡張アカウンティングに Perl インタフェースを使用

Perl インタフェースは、拡張アカウンティングのタスクとプロジェクトに Perl バインディングを提供します。このインタフェースを使用すると、`exacct` フレームワークによって生成されたアカウンティングファイルを Perl スクリプトで読み取ることができます。このインタフェースでは、Perl スクリプトによる `exacct` ファイルの書き込みも可能です。

この章の内容は、次のとおりです。

拡張アカウンティングの概要

拡張アカウンティング (`exacct`) は Oracle Solaris オペレーティングシステムのためのアカウンティングフレームワークであり、従来の SVR4 アカウンティングメカニズムによって提供される機能に追加機能を提供します。従来の SVR4 アカウンティングにはこれらの欠点があります。

- 従来のアカウンティングによって収集されたデータは変更できません。

  統計 SVR4 アカウンティング収集のタイプまたは数量は各アプリケーション用にカスタマイズできません。従来のアカウンティングが収集するデータへの変更は、アカウンティングファイルを使用する既存のすべてのアプリケーションで機能しません。
- SVR4 のアカウンティングメカニズムはオープンではありません。

  アプリケーションは独自のデータをシステムアカウンティングのデータストリームに埋め込むことができません。
- 従来のアカウンティングメカニズムにはアグリゲーション機能がありません。

Oracle Solaris オペレーティングシステムは、存在する各プロセスについて個々のレコードを書き込みます。一連のアカウンティングレコードをより高いレベルの集合体にグループ化する機能がありません。

exacct フレームワークは従来のアカウンティングの制約に対処し、アカウンティングデータ収集のための構成可能でオープンかつ拡張性のあるフレームワークを提供します。

- 収集されるデータは exacct API を使用して構成できます。
- アプリケーションはシステムアカウンティングファイル内に独自のデータを埋め込むことも、独自のカスタムアカウンティングファイルを作成して操作することもできます。
- 従来のアカウンティングメカニズムのデータアグリゲーション機能の欠如は、タスクおよびプロジェクトが対処します。タスクは作業の単位であるプロセスのセットを識別します。プロジェクトを使用すると、一連のユーザーによって実行されるプロセスをより高いレベルのエンティティーに集約できます。タスクおよびプロジェクトの詳細については、project(4) のマニュアルページを参照してください。

拡張アカウンティングのより広範な概要については、『Oracle Solaris 11.3 でのリソースの管理』 の 第 4 章,「拡張アカウンティングについて」を参照してください。

libexacct に対する Perl インタフェース

Perl インタフェースのオブジェクトモデル

Sun::Solaris::Exacct モジュールは、libexacct(3LIB) ライブラリが提供するすべてのクラスの親です。libexacct(3LIB) で操作できるエンティティーのタイプは、exacct 形式ファイル、catalog タグ、および exacct オブジェクトです。exacct オブジェクトはさらに 2 つのタイプに分類できます。

- アイテム – 単一データ値
- グループ – アイテムのリスト

libexacct に Perl インタフェースを使用する利点

拡張アカウンティングへの Perl 拡張によって、ベースとなる libexacct(3LIB) API に Perl インタフェースが追加され、次の拡張機能を使用できるようになります。

- ベースとなる C API と機能的に同等な Perl インタフェースを提供する C API と完全に同等になる。

このインタフェースは、C コーディングを必要としない exacct ファイルにアクセスするためのメカニズムを提供します。C から使用できるすべての機能は、Perl インタフェースからも使用できます。

- 使いやすさ。

  ベースとなる C API から取得したデータを Perl データ型として表示します。Perl データ型を使用すると、データへのアクセスが容易になり、またバッファーでのパックとアンパックの操作が不要になります。

- 自動メモリー管理。

  C API では exacct ファイルにアクセスする場合、プログラマがメモリー管理の責任を持つ必要があります。メモリー管理には、ea_unpack_object(3EXACCT) などの関数に適切なフラグを渡し、API に渡すバッファーを明示的に割り当てることなどが含まれます。Perl API では、すべてのメモリー管理が Perl ライブラリによって実行されるため、これらが必要ありません。

- 誤った API の使用を防ぐ。

  ea_object_t 構造体は、exacct レコードのメモリー内表現を提供します。ea_object_t 構造体は、グループレコードとアイテムレコードの両方の操作に使用する共用型です。その結果、誤った型の構造体が一部の API 関数に渡されることがあります。クラス階層を追加することで、このタイプのプログラミングエラーを防止できます。

Perl double 型のスカラー

このドキュメントで説明するモジュールは、Perl double 型スカラー機能を広範囲に使用します。*double* 型のスカラー機能を使用した場合、スカラー値がコンテキストに応じて整数または文字列として動作します。この動作は、$! Perl 変数 (errno) で示されるものと同じです。double 型のスカラー機能を使用すると、値を表示するために整数値を対応する文字列にマッピングする必要がなくなります。次の例は、double 型スカラーの使用を示しています。

```
# Assume $obj is a Sun::Solaris::Item
my $type = $obj->type();

# prints out "2 EO_ITEM"
printf("%d %s\n", $type, $type);

# Behaves as an integer, $i == 2
my $i = 0 + $type;

# Behaves as a string, $s = "abc EO_ITEM xyx"
my $s = "abc $type xyz";
```

Perl モジュール

各 Perl モジュールには、関連するプロジェクト、タスク、および exacct 関連関数のグループが含まれています。各関数には、標準の Sun::Solaris:: Perl パッケージ接頭辞があります。

表 5 Perl モジュール

モジュール	説明
33 ページの「Sun::Solaris::Project モジュール」	次のプロジェクト操作関数にアクセスする機能を提供します。getprojid(2)、setproject(3PROJECT)、project_walk(3PROJ
34 ページの「Sun::Solaris::Task モジュール」	タスク管理関数 settaskid(2) および gettaskid(2) にアクセスする機能を提供します。
35 ページの「Sun::Solaris::Exacct モジュール」	最上位レベルの exacct モジュール。このモジュールの機能は、exacct 関連のシステムコールである getacct(2)、putacct(2)、および wracct(2) に加え、libexacct(3LIB) ライブラリ関数の ea_error(3EXACCT) にもアクセスできます。このモジュールには、exacct EO_*、EW_*、EXR_*、P_*、TASK_* といったさまざまなすべてのマクロの定数が含まれています。
36 ページの「Sun::Solaris::Exacct::Catalog モジュール」	exacct カタログタグ内のビットフィールドや EXC_*、EXD_*、および EXD_* マクロにアクセスするオブジェクト指向型メソッドを提供します。
38 ページの「Sun::Solaris::Exacct::File モジュール」	次の libexacct(3LIB) アカウンティングファイル関数にアクセスするオブジェクト指向型メソッドを提供します。ea_open(3EXACCT)、ea_close(3EXACCT)、ea_get_creator(
40 ページの「Sun::Solaris::Exacct::Object モジュール」	個々の exacct アカウンティングファイルオブジェクトにアクセスする、オブジェクト指向型メソッドを提供します。exacct オブジェクトは、該当する Sun::Solaris::Exacct::Object サブクラスに与えられた、隠された参照として表されます。このモジュールはさらに、使用できる 2 つのオブジェクト型であるアイテムとグループに分けられます。ea_match_object_catalog(3EXACCT) 関数、ea_attach_to_object(3EXACCT) 関数にアクセスするメソッドも提供されます。
41 ページの「Sun::Solaris::Exacct::Object::Item モジュール」	個々の exacct アカウンティングファイルアイテムにアクセスする、オブジェクト指向型メソッドを提供します。このタイプのオブジェクトは、Sun::Solaris::Exacct::Object から継承します。
42 ページの「Sun::Solaris::Exacct::Object::Group モジュール」	個々の exacct アカウンティングファイルグループにアクセスする、オブジェ

モジュール	説明
	クト指向型メソッドを提供します。このタイプのオブジェクトは Sun::Solaris::Exacct::Object を継承しており、ea_attach_to_group(3EXACCT) 関数へのアクセスを提供します。グループ内のアイテムは Perl 配列として表されます。
43 ページの「Sun::Solaris::Exacct::Object::_Array モジュール」	Sun::Solaris::Exacct::Object::Group 内で配列の型として使用されるプライベート配列型。

Sun::Solaris::Project モジュール

Sun::Solaris::Project モジュールは、プロジェクト関連システムコールおよび libproject(3LIB) ライブラリのラッパーを提供します。

Sun::Solaris::Project 定数

Sun::Solaris::Project モジュールは、プロジェクト関連ヘッダーファイルの定数を使用します。

```
MAXPROJID
PROJNAME_MAX
PROJF_PATH
PROJECT_BUFSZ
SETPROJ_ERR_TASK
SETPROJ_ERR_POOL
```

Sun::Solaris::Project 関数、クラスメソッド、およびオブジェクトメソッド

libexacct(3LIB) API への Perl 拡張は、プロジェクト用に次の関数を提供します。

setproject(3PROJECT)
setprojent(3PROJECT)
getdefaultproj(3PROJECT)
inproj(3PROJECT)
getprojent(3PROJECT)
fgetprojent(3PROJECT)
getprojbyname(3PROJECT)
getprojbyid(3PROJECT)
getprojbyname(3PROJECT)

endproject(3PROJECT)

Sun::Solaris::Project モジュールにはクラスメソッドがありません。

Sun::Solaris::Project モジュールにはオブジェクトメソッドがありません。

Sun::Solaris::Project のエクスポート

デフォルトでは、このモジュールからは何もエクスポートされません。次の表にリストされたタグを使用して、このモジュールで定義された定数および関数を個別にインポートできます。

タグ	定数または関数
:SYSCALLS	getprojid()
:LIBCALLS	setproject()、activeprojects()、getprojent()、setprojent()、endprojent()、getproj
:CONSTANTS	MAXPROJID_TASK、PROJNAME_MAX、PROJF_PATH、PROJECT_BUFSZ、SETPROJ_ERR、SETPROJ_ERR
:ALL	:SYSCALLS、:LIBCALLS、:CONSTANTS

Sun::Solaris::Task モジュール

Sun::Solaris::Task モジュールは、settaskid(2) および gettaskid(2) システムコールのラッパーを提供します。

Sun::Solaris::Task 定数

Sun::Solaris::Task モジュールは次の定数を使用します。

TASK_NORMAL
TASK_FINAL

Sun::Solaris::Task 関数、クラスメソッド、およびオブジェクトメソッド

libexacct(3LIB) API への Perl 拡張は、タスク用に次の関数を提供します。

settaskid(2)
gettaskid(2)

Sun::Solaris::Task モジュールにはクラスメソッドがありません。

Sun::Solaris::Task モジュールにはオブジェクトメソッドがありません。

Sun::Solaris::Task のエクスポート

デフォルトでは、このモジュールからは何もエクスポートされません。次の表にリストされたタグを使用して、このモジュールで定義された定数および関数を個別にインポートできます。

タグ	定数または関数
:SYSCALLS	settaskid()、gettaskid()
:CONSTANTS	TASK_NORMAL および TASK_FINAL
:ALL	:SYSCALLS および :CONSTANTS

Sun::Solaris::Exacct モジュール

Sun::Solaris::Exacct モジュールは、ea_error(3EXACCT) 関数およびすべての exacct システムコールのラッパーを提供します。

Sun::Solaris::Exacct 定数

Sun::Solaris::Exacct モジュールは、さまざまな exacct ヘッダーファイルの定数を提供します。P_PID、P_TASKID、P_PROJID、およびすべての EW_*、EP_*、EXR_* マクロはモジュールのビルドプロセス中に抽出されます。マクロは /usr/include にある exacct ヘッダーファイルから抽出され、Perl 定数として提供されます。Sun::Solaris::Exacct 関数に渡される定数は、EW_FINAL のような整数値でも、"EW_FINAL" のような同じ変数の文字列表現でもかまいません。

Sun::Solaris::Exacct 関数、クラスメソッド、およびオブジェクトメソッド

libexacct(3LIB) API への Perl 拡張は、Sun::Solaris::Exacct モジュール用に次の関数を提供します。

getacct(2)
putacct(2)
wracct(2)
ea_error(3EXACCT)

```
ea_error_str
ea_register_catalog
ea_new_file
ea_new_item
ea_new_group
ea_dump_object
```

注記 - ea_error_str() は簡易関数として提供されるため、次のような繰り返されるコードブロックを回避できます。

```
if (ea_error() == EXR_SYSCALL_FAIL) {
        print("error: $!\n");
} else {
        print("error: ", ea_error(), "\n");
}
```

Sun::Solaris::Exacct モジュールにはクラスメソッドがありません。

Sun::Solaris::Exacct モジュールにはオブジェクトメソッドがありません。

Sun::Solaris::Exacct のエクスポート

デフォルトでは、このモジュールからは何もエクスポートされません。次のタグを使用して、このモジュールで定義された定数および関数を個別にインポートできます。

タグ	定数または関数
:SYSCALLS	getacct()、putacct()、wracct()
:LIBCALLS	ea_error()、ea_error_str()
:CONSTANTS	P_PID、P_TASKID、P_PROJID EW_*、EP_*、EXR_*
:SHORTAND	ea_register_catalog()、ea_new_catalog()、ea_new_file()、ea_new_item()、ea_new_gro
:ALL	:SYSCALLS、:LIBCALLS、:CONSTANTS、および :SHORTHAND
:EXACCT_CONSTANTS	:CONSTANTS、および Sun::Solaris::Catalog、Sun::Solaris::File、Sun::Solaris::Object 用の :CONSTANTS タグ
:EXACCT_ALL	:ALL、および Sun::Solaris::Catalog、Sun::Solaris::File、Sun::Solaris::Object 用の :ALL

Sun::Solaris::Exacct::Catalog モジュール

Sun::Solaris::Exacct::Catalog モジュールは、カタログタグとして使用される 32 ビット整数を包むラッパーを提供します。このカタログタグは、Sun::Solaris::

Exacct::Catalog クラスに与えられた Perl オブジェクトとして表されます。メソッドを使用すると、カタログタグ内のフィールドを操作できます。

Sun::Solaris::Exacct::Catalog 定数

EXT_*、EXC_*、および EXD_* のマクロはすべてモジュールのビルドプロセス中に /usr/include/sys/exact_catalog.h ファイルから抽出され、定数として提供されます。Sun::Solaris::Exacct::Catalog メソッドに渡される関数は、EXT_UINT8 のような整数値でも、"EXT_UINT8" のような同じ変数の文字列表現でもかまいません。

Sun::Solaris::Exacct::Catalog 関数、クラスメソッド、およびオブジェクトメソッド

libexacct(3LIB) API への Perl 拡張は、Sun::Solaris::Exacct::Catalog 用に次のクラスメソッドを提供します。Exacct(3PERL) および Exacct::Catalog(3PERL)。

```
register
new
```

libexacct(3LIB) API への Perl 拡張は、Sun::Solaris::Exacct::Catalog 用に次のオブジェクトメソッドを提供します。

```
value
type
catalog
id
type_str
catalog_str
id_str
```

Sun::Solaris::Exacct::Catalog のエクスポート

デフォルトでは、このモジュールからは何もエクスポートされません。次のタグを使用して、このモジュールで定義された定数および関数を個別にインポートできます。

タグ	定数または関数
:CONSTANTS	EXT_*、EXC_*、および EXD_*。
:ALL	:CONSTANTS

さらに、register() 関数で定義されたすべての定数は、オプションで呼び出し元のパッケージにエクスポートできます。

Sun::Solaris::Exacct::File モジュール

Sun::Solaris::Exacct::File モジュールは、アカウンティングファイルを操作する exacct 関数のラッパーを提供します。インタフェースはオブジェクト指向であり、exacct ファイルの作成および読み取りができます。このモジュールでラップされる C ライブラリコールは次のとおりです。

ea_open(3EXACCT)
ea_close(3EXACCT)
ea_next_object(3EXACCT)
ea_previous_object(3EXACCT)
ea_write_object(3EXACCT)
ea_get_object(3EXACCT)
ea_get_creator(3EXACCT)
ea_get_hostname(3EXACCT)

ファイルの読み取りおよび書き込みメソッドは Sun::Solaris::Exacct::Object オブジェクト上で動作します。これらのメソッドは、必要なメモリー管理、パック、アンパック、および必要とされる構造変換をすべて実行します。

Sun::Solaris::Exacct::File 定数

Sun::Solaris::Exacct::File は、EO_HEAD、EO_TAIL、EO_NO_VALID_HDR、EO_POSN_MSK、および EO_VALIDATE_MSK 定数を提供します。new() メソッドで必要なほかの定数は、標準の Perl の Fcntl モジュールにあります。表6 は、$oflags および $aflags のさまざまな値の new() のアクションを示しています。

Sun::Solaris::Exacct::File 関数、クラスメソッド、およびオブジェクトメソッド

Sun::Solaris::Exacct::File モジュールには関数がありません。

libexacct(3LIB) API への Perl 拡張は、Sun::Solaris::Exacct::File 用に次のクラスメソッドを提供します。

new

次の表は、$oflags および $aflags パラメータの組み合わせに対する new() のアクションを示しています。

表 6 $oflags および $aflags パラメータ

$oflags	$aflags	アクション
O_RDONLY	空または EO_HEAD	ファイルの最初で読み取り用に開きます。
O_RDONLY	EO_TAIL	ファイルの最後で読み取り用に開きます。
O_WRONLY	無視	ファイルが存在する必要があり、ファイルの最後で書き込み用に開きます。
O_WRONLY | O_CREAT	無視	ファイルが存在しない場合はファイルを作成します。それ以外の場合は、切り捨てて、書き込み用に開きます。
O_RDWR	無視	ファイルが存在する必要があり、ファイルの最後で読み取りまたは書き込み用に開きます。
O_RDWR | O_CREAT	無視	ファイルが存在しない場合はファイルを作成します。それ以外の場合は、切り捨てて、読み取りまたは書き込み用に開きます。

注記 - $oflags に唯一有効な値は、O_RDONLY、O_WRONLY、O_RDWR または O_CREAT の組み合わせです。$aflags は、O_RDONLY に対するファイル内の必要なポジショニングを示しています。EO_HEAD または EO_TAIL が許可されます。空の場合は EO_HEAD と見なされます。

libexacct(3LIB) API への Perl 拡張は、Sun::Solaris::Exacct::File 用に次のオブジェクトメソッドを提供します。

```
creator
hostname
next
previous
get
write
```

注記 - Sun::Solaris::Exacct::File を閉じます。Sun::Solaris::Exacct::File に対する明示的な close() メソッドはありません。ファイルハンドルオブジェクトが未定義の場合または再度割り当てられた場合にファイルが閉じられます。

Sun::Solaris::Exacct::File のエクスポート

デフォルトでは、このモジュールからは何もエクスポートされません。次のタグを使用して、このモジュールで定義された定数を個別にインポートできます。

タグ	定数または関数
:CONSTANTS	EO_HEAD、EO_TAIL、EO_NO_VALID_HDR、EO_POSN_MSK、EO_VALIDATE_MSK。

タグ	定数または関数
:ALL	:CONSTANTS および Fcntl(:DEFAULT)。

Sun::Solaris::Exacct::Object モジュール

Sun::Solaris::Exacct::Object モジュールは、exacct オブジェクトに使用できる 2 つの型であるアイテムとグループの親として機能します。exacct Item は、単一データ値、埋め込まれた exacct オブジェクト、または raw データのブロックです。単一データ値の例として、プロセスによって消費されるユーザー CPU 時間の秒数があります。exacct Group は、特定のプロセスまたはタスクのすべてのリソース使用率の値など、exacct アイテムの順序付けられたコレクションです。グループ同士を互いにネストする必要がある場合は、内側のグループを埋め込みの exacct オブジェクトとして外側のグループの内部に格納できます。

Sun::Solaris::Exacct::Object モジュールには、exacct のアイテムとグループ両方に共通するメソッドが含まれています。Sun::Solaris::Exacct::Object の属性およびそこから派生したすべてのクラスは、new() で新規作成されたあと、読み取り専用になることに注意してください。属性が読み取り専用になることで、カタログタグおよびデータ値の不整合が生じる原因となる、属性の意図しない変更を防ぎます。読み取り専用属性の唯一の例外は、グループオブジェクト内にアイテムを格納するのに使用される配列です。この配列は、通常の Perl の配列演算子を使用して変更できます。

Sun::Solaris::Exacct::Object 定数

Sun::Solaris::Exacct::Object は、EO_ERROR、EO_NONE、EO_ITEM、および EO_GROUP 定数を提供します。

Sun::Solaris::Exacct::Object 関数、クラスメソッド、およびオブジェクトメソッド

Sun::Solaris::Exacct::Object モジュールには関数がありません。

libexacct(3LIB) API への Perl 拡張は、Sun::Solaris::Exacct::Object 用に次のクラスメソッドを提供します。

dump

libexacct(3LIB) API への Perl 拡張は、Sun::Solaris::Exacct::Object 用に次のオブジェクトメソッドを提供します。

```
type
catalog
match_catalog
value
```

Sun::Solaris::Exacct::Object のエクスポート

デフォルトでは、このモジュールからは何もエクスポートされません。次のタグを使用して、このモジュールで定義された定数および関数を個別にインポートできます。

タグ	定数または関数
:CONSTANTS	EO_ERROR、EO_NONE、EO_ITEM、および EO_GROUP
:ALL	:CONSTANTS

Sun::Solaris::Exacct::Object::Item モジュール

Sun::Solaris::Exacct::Object::Item モジュールは、exacct データアイテムに使用されます。exacct データアイテムは、Sun::Solaris::Exacct::Object クラスのサブクラスである Sun::Solaris::Exacct::Object::Item クラスに与えられた、隠された参照として表されます。ベースとなる exacct データ型は、次のように Perl 型にマッピングされます。

表 7 Perl データ型にマッピングされる exacct データ型

exacct 型	Perl 内部タイプ
EXT_UINT8	IV (整数)
EXT_UINT16	IV (整数)
EXT_UINT32	IV (整数)
EXT_UINT64	IV (整数)
EXT_DOUBLE	NV (double)
EXT_STRING	PV (文字列)
EXT_EXACCT_OBJECT	Sun::Solaris::Exacct::Object サブクラス
EXT_RAW	PV (文字列)

Sun::Solaris::Exacct::Object::Item 定数

Sun::Solaris::Exacct::Object::Item には定数がありません。

Sun::Solaris::Exacct::Object::Item 関数、クラスメソッド、およびオブジェクトメソッド

Sun::Solaris::Exacct::Object::Item には関数がありません。

Sun::Solaris::Exacct::Object::Item は、Sun::Solaris::Exacct::Object 基本クラスのすべてのクラスメソッドに加え、new() クラスメソッドを継承しています。

new

Sun::Solaris::Exacct::Object::Item は、Sun::Solaris::Exacct::Object 基本クラスのすべてのオブジェクトメソッドを継承しています。

Sun::Solaris::Exacct::Object::Item のエクスポート

Sun::Solaris::Exacct::Object::Item にはエクスポートがありません。

Sun::Solaris::Exacct::Object::Group モジュール

Sun::Solaris::Exacct::Object::Group モジュールは、exacct グループオブジェクトに使用されます。exacct グループオブジェクトは、Sun::Solaris::Exacct::Object クラスのサブクラスである Sun::Solaris::Exacct::Object::Group クラスに与えられた、隠された参照として表されます。グループ内のアイテムは Perl 配列内に格納され、その配列への参照は継承された value() メソッドからアクセスできます。つまり、グループ内の個々のアイテムを通常の Perl 配列の構文と演算子で操作できます。配列のすべてのデータ要素は、Sun::Solaris::Exacct::Object クラスから派生している必要があります。グループオブジェクトは、既存のグループをデータアイテムとして追加するだけで互いにネストすることもできます。

Sun::Solaris::Exacct::Object::Group 定数

Sun::Solaris::Exacct::Object::Group には定数がありません。

Sun::Solaris::Exacct::Object::Group 関数、クラスメソッド、およびオブジェクトメソッド

Sun::Solaris::Exacct::Object::Group には関数がありません。

Sun::Solaris::Exacct::Object::Group は、Sun::Solaris::Exacct::Object 基本クラスのすべてのクラスメソッドに加え、new() クラスメソッドを継承しています。

new

Sun::Solaris::Exacct::Object::Group は、Sun::Solaris::Exacct::Object 基本クラスのすべてのオブジェクトメソッドに加え、new() クラスメソッドを継承しています。

as_hash
as_hashlist

Sun::Solaris::Exacct::Object::Group のエクスポート

Sun::Solaris::Exacct::Object::Group にはエクスポートがありません。

Sun::Solaris::Exacct::Object::_Array モジュール

Sun::Solaris::Exacct::Object::_Array クラスは、exacct グループ内に配置されているデータアイテムの型チェックを実施するために内部で使用されます。Sun::Solaris::Exacct::Object::_Array はユーザーが直接作成するべきではありません。

Sun::Solaris::Exacct::Object::_Array 定数

Sun::Solaris::Exacct::Object::_Array には定数がありません。

Sun::Solaris::Exacct::Object::_Array 関数、クラスメソッド、およびオブジェクトメソッド

Sun::Solaris::Exacct::Object::_Array には関数がありません。

Sun::Solaris::Exacct::Object::_Array には内部使用のクラスメソッドがありません。

Sun::Solaris::Exacct::Object::_Array は Perl TIEARRAY メソッドを使用します。

Sun::Solaris::Exacct::Object::_Array のエクスポート

Sun::Solaris::Exacct::Object::_Array にはエクスポートがありません。

Perl コードの例

このセクションでは、exacct ファイルにアクセスするための Perl コードの例を示します。

例 6 疑似コードのプロトタイプの使用

Perl exacct ライブラリの一般的な使用では、既存の exacct ファイルを読み取ります。擬似コードを使用して、さまざまな Perl exacct クラスの関係を示します。exacct ファイルを開くプロセスとスキャンするプロセス、および対象のオブジェクトを処理するプロセスを擬似コードで示します。次の擬似コードでは、わかりやすくするために「簡易」関数を使用しています。

```
-- Open the exacct file ($f is a Sun::Solaris::Exacct::File)
my $f = ea_new_file(...)

-- While not EOF ($o is a Sun::Solaris::Exacct::Object)
while (my $o = $f->get())

        -- Check to see if object is of interest
        if ($o->type() == &EO_ITEM)
             ...

        -- Retrieve the catalog ($c is a Sun::Solaris::Exacct::Catalog)
        $c = $o->catalog()

        -- Retrieve the value
        $v = $o->value();

        -- $v is a reference to a Sun::Solaris::Exacct::Group for a Group
        if (ref($v))
              ....

        -- $v is perl scalar for Items
        else
```

例 7 exacct オブジェクトを再帰的にダンプ

```
sub dump_object
{
    my ($obj, $indent) = @_;
    my $istr = '  ' x $indent;

    #
    # Retrieve the catalog tag.  Because we are doing this in an array
    # context, the catalog tag will be returned as a (type, catalog, id)
    # triplet, where each member of the triplet will behave as an integer
    # or a string, depending on context.  If instead this next line provided
```

```
    # a scalar context, e.g.
    #    my $cat  = $obj->catalog()->value();
    # then $cat would be set to the integer value of the catalog tag.
    #
    my @cat = $obj->catalog()->value();

    #
    # If the object is a plain item
    #
    if ($obj->type() == &EO_ITEM) {
          #
          # Note:  The '%s' formats provide s string context, so the
          # components of the catalog tag will be displayed as the
          # symbolic values.  If we changed the '%s' formats to '%d',
          # the numeric value of the components would be displayed.
          #
          printf("%sITEM\n%s  Catalog = %s|%s|%s\n",
              $istr, $istr, @cat);
          $indent++;

          #
          # Retrieve the value of the item.  If the item contains in
          # turn a nested exacct object (i.e. a item or group), then
          # the value method will return a reference to the appropriate
          # sort of perl object (Exacct::Object::Item or
          # Exacct::Object::Group). We could of course figure out that
          # the item contained a nested item or group by examining
          # the catalog tag in @cat and looking for a type of
          # EXT_EXACCT_OBJECT or EXT_GROUP.
          my $val = $obj->value();
          if (ref($val)) {
             # If it is a nested object, recurse to dump it.
             dump_object($val, $indent);
          } else {
             # Otherwise it is just a 'plain' value, so display it.
             printf("%s  Value = %s\n", $istr, $val);
          }

        #
        # Otherwise we know we are dealing with a group.  Groups represent
        # contents as a perl list or array (depending on context), so we
        # can process the contents of the group with a 'foreach' loop, which
        # provides a list context.  In a list context the value method
        # returns the content of the group as a perl list, which is the
        # quickest mechanism, but doesn't allow the group to be modified.
        # If we wanted to modify the contents of the group we could do so
        # like this:
        #    my $grp = $obj->value();   # Returns an array reference
        #    $grp->[0] = $newitem;
        # but accessing the group elements this way is much slower.
        #
        } else {
               printf("%sGROUP\n%s  Catalog = %s|%s|%s\n",
                   $istr, $istr, @cat);
               $indent++;
               # 'foreach' provides a list context.
               foreach my $val ($obj->value()) {
                       dump_object($val, $indent);
               }
               printf("%sENDGROUP\n", $istr);
        }
}
```

例 8 新規グループレコードの作成とファイルへの書き込み

```
# Prototype list of catalog tags and values.
my @items = (
   [ &EXT_STRING | &EXC_DEFAULT | &EXD_CREATOR      => "me"         ],
   [ &EXT_UINT32 | &EXC_DEFAULT | &EXD_PROC_PID     => $$           ],
   [ &EXT_UINT32 | &EXC_DEFAULT | &EXD_PROC_UID     => $<           ],
   [ &EXT_UINT32 | &EXC_DEFAULT | &EXD_PROC_GID     => $(           ],
   [ &EXT_STRING | &EXC_DEFAULT | &EXD_PROC_COMMAND => "/bin/stuff" ],
);

# Create a new group catalog object.
my $cat = new_catalog(&EXT_GROUP | &EXC_DEFAULT | &EXD_NONE);

# Create a new Group object and retrieve its data array.
my $group = new_group($cat);
my $ary = $group->value();

# Push the new Items onto the Group array.
foreach my $v (@items) {
        push(@$ary, new_item(new_catalog($v->[0]), $v->[1]));
}

# Nest the group within itself (performs a deep copy).
push(@$ary, $group);

# Dump out the group.
dump_object($group);
```

例 9 exacct ファイルのダンプ

```
#!/usr/bin/perl

use strict;
use warnings;
use blib;
use Sun::Solaris::Exacct qw(:EXACCT_ALL);

die("Usage is dumpexacct

# Open the exact file and display the header information.
my $ef = ea_new_file($ARGV[0], &O_RDONLY) || die(error_str());
printf("Creator:  %s\n", $ef->creator());
printf("Hostname: %s\n\n", $ef->hostname());

# Dump the file contents
while (my $obj = $ef->get()) {
        ea_dump_object($obj);
}

# Report any errors
if (ea_error() != EXR_OK && ea_error() != EXR_EOF)  {
       printf("\nERROR: %s\n", ea_error_str());
       exit(1);
}
exit(0);
```

dump メソッドからの出力

この例は、Sun::Solaris::Exacct::Object->dump() メソッドの書式付き出力を示しています。

```
GROUP
  Catalog = EXT_GROUP|EXC_DEFAULT|EXD_GROUP_PROC_PARTIAL
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_PID
    Value = 3
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_UID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_GID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_PROJID
    Value = 0
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TASKID
    Value = 0
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_USER_SEC
    Value = 0
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_USER_NSEC
    Value = 0
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_SYS_SEC
    Value = 890
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CPU_SYS_NSEC
    Value = 760000000
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_START_SEC
    Value = 1011869897
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_START_NSEC
    Value = 380771911
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FINISH_SEC
    Value = 0
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FINISH_NSEC
    Value = 0
  ITEM
    Catalog = EXT_STRING|EXC_DEFAULT|EXD_PROC_COMMAND
    Value = fsflush
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TTY_MAJOR
    Value = 4294967295
  ITEM
    Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_TTY_MINOR
    Value = 4294967295
  ITEM
    Catalog = EXT_STRING|EXC_DEFAULT|EXD_PROC_HOSTNAME
    Value = mower
  ITEM
    Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FAULTS_MAJOR
    Value = 0
  ITEM
```

```
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_FAULTS_MINOR
      Value = 0
    ITEM
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_MESSAGES_SND
      Value = 0
    ITEM
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_MESSAGES_RCV
      Value = 0
    ITEM
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_BLOCKS_IN
      Value = 19
    ITEM
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_BLOCKS_OUT
      Value = 40833
    ITEM
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CHARS_RDWR
      Value = 0
    ITEM
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CONTEXT_VOL
      Value = 129747
    ITEM
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_CONTEXT_INV
      Value = 79
    ITEM
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_SIGNALS
      Value = 0
    ITEM
      Catalog = EXT_UINT64|EXC_DEFAULT|EXD_PROC_SYSCALLS
      Value = 0
    ITEM
      Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_ACCT_FLAGS
      Value = 1
    ITEM
      Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_ANCPID
      Value = 0
    ITEM
      Catalog = EXT_UINT32|EXC_DEFAULT|EXD_PROC_WAIT_STATUS
      Value = 0
  ENDGROUP
```

♦ ♦ ♦ 第 5 章

5 リソース制御

この章では、リソース制御およびそれらのプロパティーについて説明します。

リソース制御の概要

拡張アカウンティング機能を使用して、システム上の作業負荷のリソース消費を調べます。リソース消費が明らかになったら、リソース制御機能を使用してリソース使用率を制限します。リソースを制限することで、作業負荷がリソースを過剰に消費しないようにします。

リソース制御の概要およびリソース制御を管理するコマンドの例については、『Oracle Solaris 11.3 でのリソースの管理』 の 第 6 章,「リソース制御について」および『Oracle Solaris 11.3 でのリソースの管理』 の 第 7 章,「リソース制御の管理のタスク」を参照してください。

リソース制御機能には次の利点があります。

- **動的な設定**

  リソース制御はシステムの実行中に調整できます。
- **包含レベルのきめ細かさ**

  リソース制御は、ゾーン、プロジェクト、タスク、またはプロセスの包含レベルで行われます。包含レベルによって、構成が簡略化され、収集された値が特定のゾーン、プロジェクト、タスク、またはプロセスの近くに配置されます。

リソース制御のフラグとアクション

このセクションでは、リソース制御に関連したフラグ、アクション、およびシグナルについて説明します。

rlimit、リソース制限

rlimit はプロセスベースです。rlimit は、プロセスによるさまざまなシステムリソースの消費に制限の境界を設けます。プロセスが作成する各プロセスは元のプロセスを継承します。リソース制限は値のペアによって定義されます。これらの値は、現在の (ソフト) 制限と最大の (ハード) 制限を指定します。

プロセスによって、ハードリミットがソフトリミット以上の値に不可逆的に下げられる可能性があります。ルート ID を持つプロセスだけがハード制限を引き上げることができます。setrlimit() および getrlimit() を参照してください。

rlimit 構造体には、ソフト制限とハード制限を定義する 2 つのメンバーが含まれます。

```
rlim_t     rlim_cur;       /* current (soft) limit */
rlim_t     rlim_max        /* hard limit */
```

rctl、リソース制御

rctl は、プロジェクトデータベース内で定義されたプロセス、タスク、およびプロジェクトによるリソース消費を制御することで、rlimit のプロセスベース制限を拡張します。

注記 - リソース制限の設定には、rlimit の使用よりも rctl メカニズムが推奨されます。rlimit 機能を使用する唯一の理由は、UNIX プラットフォーム間の移植性が求められる場合です。

アプリケーションは、アプリケーションがリソース制御を処理する方法に応じて、次の幅広いカテゴリに分類されます。行われるアクションに基づいて、リソース制御をさらに分類できます。ほとんどはエラーを報告して操作を終了します。その他のリソース制御では、アプリケーションが操作を再開し、削減されたリソース使用率に適応できます。増加する値での漸進的な一連のアクションを各リソース制御に対して指定できます。

リソース制御の属性リストは、特権レベル、しきい値、しきい値を超えた場合に実行されるアクションで構成されます。

リソース制御値と特権レベル

リソース制御の各しきい値は、次のいずれかの特権レベルに関連付けられている必要があります。

RCPRIV_BASIC

特権レベルは呼び出し元プロセスの所有者が変更できます。RCPRIV_BASIC はリソースのソフトリミットに関連付けられています。

RCPRIV_PRIVILEGED

特権レベルは特権を持っている呼び出し元 (root) だけが変更できます。RCPRIV_PRIVILEGED はリソースのハードリミットに関連付けられています。

`setrctl`(2) は大域ゾーンの特権ユーザーとして呼び出された場合にのみ成功します。非大域ゾーン内では、`root` はゾーン規模の制御を設定できません。

RCPRIV_SYSTEM

特権レベルはオペレーティングシステムインスタンスの期間中は固定されたままです。

図4 は、`/etc/project` ファイルの `process.max-cpu-time` リソース制御によって定義されるシグナルの特権レベルを設定する際のタイムラインを示しています。

ローカルアクションとローカルフラグ

ローカルアクションとローカルフラグは、このリソース制御ブロックで表される現在のリソース制御値に適用されます。ローカルアクションとローカルフラグは値固有です。リソース制御に設定される各しきい値に対して、次のローカルアクションとローカルフラグを使用できます。

RCTL_LOCAL_NOACTION

このリソース制御値を超えた場合に実行されるローカルアクションはありません。

RCTL_LOCAL_SIGNAL

`rctlblk_set_local_action()` によって設定された指定のシグナルが、値シーケンス内にこのリソース制御値を配置したプロセスに送信されます。

RCTL_LOCAL_DENY

このリソース制御値が検出された場合、リソースの要求は拒否されます。この制御に RCTL_GLOBAL_DENY_ALWAYS が設定されている場合、すべての値で設定されます。この制御に RCTL_GLOBAL_DENY_NEVER が設定されている場合、すべての値でクリアされます。

RCTL_LOCAL_MAXIMAL

このリソース制御値は、この制御のリソースの最大量に対するリクエストを表します。このリソース制御に RCTL_GLOBAL_INFINITE が設定されている場合、RCTL_LOCAL_MAXIMAL は決して超えることのない無制限のリソース制御値を示します。

大域アクションと大域フラグ

大域フラグは、このリソース制御ブロックで表される現在のすべてのリソース制御値に適用されます。大域アクションと大域フラグは `rctladm`(1M) によって設定されます。大域アクションと大域フラグは `setrctl()` と一緒に設定することはできません。大域フラグはすべてのリソース制御に適用されます。リソース制御に設定される各しきい値に対して、次の大域アクションと大域フラグを使用できます。

RCTL_GLOBAL_NOACTION

この制御でリソース制御値を超えた場合に実行される大域アクションはありません。

RCTL_GLOBAL_SYSLOG

この制御に関連付けられたシーケンス上でリソース制御値を超えた場合、`syslog()` 機能によって標準メッセージが記録されます。

RCTL_GLOBAL_SECONDS

制限値の単位の文字列を秒として定義します。

RCTL_GLOBAL_COUNT

制限値の単位の文字列を数として定義します。

RCTL_GLOBAL_BYTES

制限値の単位の文字列をバイトとして定義します。

RCTL_GLOBAL_SYSLOG_NEVER

フラグは `rctladm`(1M) を使用してこのリソース制御に RCTL_GLOBAL_SYSLOG を設定できないことを意味します。

RCTL_GLOBAL_NOBASIC

RCPRIV_BASIC 特権を持つ値はこの制御では許可されません。

RCTL_GLOBAL_LOWERABLE

権限を持たない呼び出し元が、この制御の特権付きリソース制御値の値を下げることができます。

RCTL_GLOBAL_DENY_ALWAYS

この制御で制御値を超えた場合に実行されるアクションに、常にリソースの拒否が含まれます。

RCTL_GLOBAL_DENY_NEVER

この制御で制御値を超えた場合に実行されるアクションから、常にリソースの拒否が除外されます。リソースは常に許可されますが、ほかのアクションも実行できます。

RCTL_GLOBAL_FILE_SIZE

ローカルアクションの有効なシグナルに SIGXFSZ シグナルが含まれます。

RCTL_GLOBAL_CPU_TIME

ローカルアクションの有効なシグナルに SIGXCPU シグナルが含まれます。

RCTL_GLOBAL_SIGNAL_NEVER

この制御で許可されるローカルアクションはありません。リソースは常に許可されます。

RCTL_GLOBAL_INFINITE

このリソース制御は無制限値の概念をサポートしています。通常、無制限値は CPU 時間のような蓄積型のリソースにのみ適用されます。

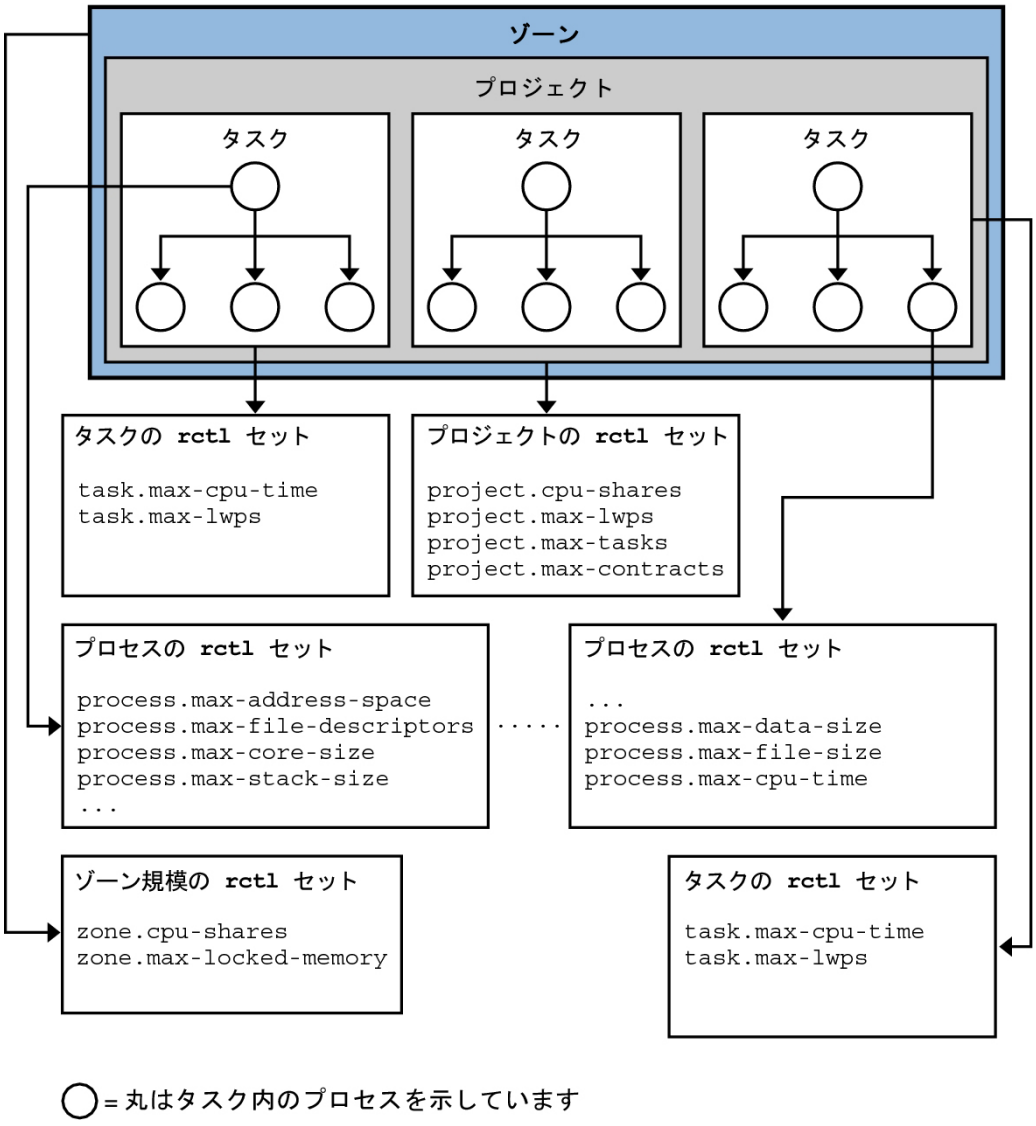
RCTL_GLOBAL_UNOBSERVABLE

通常、タスクまたはプロジェクトに関連するリソース制御は、監視用の制御値をサポートしません。タスクまたはプロセスに設定された RCPRIV_BASIC 特権制御値は、値を設定したプロセスがこの値を超えた場合にのみアクションを生成します。

ゾーン、プロジェクト、プロセス、およびタスクに関連付けられたリソース制御セット

次の図は、ゾーン、タスク、プロセス、およびプロジェクトに関連付けられたリソース制御セットを示しています。

図 3 ゾーン、タスク、プロジェクト、およびプロセスのリソース制御セット



プロセスモデルの包含レベルごとにリソース制御を設定して、1 つのリソースに複数のリソース制御を設定できます。プロセスと集合タスクまたは集合プロジェクトの両方に対して、同じリソースでリソース制御をアクティブにできます。この場合、プロセスに対するアクションが優先されます。たとえば、両方の制御が同時に検出された

場合は、task.max-cpu-time の前に process.max-cpu-time に対するアクションが実行されます。

プロジェクトに関連付けられたリソース制御

プロジェクトに関連付けられたリソース制御には次が含まれます。

project.cpu-cap
: 1 つのプロジェクトで消費可能な CPU リソース量に対する絶対的な制限。project.cpu-cap 設定と同様、100 の値は 1 つの CPU の 100% を意味します。125 の値は 125% になります。CPU キャップの使用時は、100% がシステム上の 1 つの CPU の上限となります。

project.cpu-shares
: 公平配分スケジューラ FSS(7) と一緒に使用する際にこのプロジェクトに付与される CPU 配分の数。

project.max-crypto-memory
: ハードウェアによる暗号化処理の高速化のために libpkcs11 が使用できるカーネルメモリーの合計量。カーネルバッファーおよびセッション関連の構造体の割り当ては、このリソース制御に対してチャージされます。

project.max-locked-memory
: ロックされる物理メモリーの許容合計量。

  project.max-device-locked-memory (これは削除されています) が、このリソース制御に置き換えられたことに注意してください。

project.max-msg-ids
: 1 つのプロジェクトに許容される System V メッセージキューの最大数。

project.max-port-ids
: イベントポートの許容最大数。

project.max-processes
: このプロセスで同時に使用できるプロセステーブルスロットの最大数。

注記 - 通常のプロセスとゾンビプロセスはどちらもプロセステーブルスロットを占有します。そのため、max-processes リソース制御はゾンビプロセスによってプロセステーブルが使い果たされるのを防ぎます。ゾンビプロセスには定義上 LWP が含まれないため、ゾンビプロセスがプロセステーブルを使い尽くすのを max-lwps で防ぐことはできません。

`project.max-sem-ids`
1 つのプロジェクトに許容されるセマフォー ID の最大数。

`project.max-shm-ids`
このプロジェクトに許容される共有メモリー ID の最大数。

`project.max-msg-ids`
このプロジェクトに許容されるメッセージキュー ID の最大数。

`project.max-shm-memory`
このプロジェクトに許容される System V 共有メモリーの合計量。

`project.max-lwps`
このプロジェクトで同時に使用できる LWP の最大数。

`project.max-tasks`
このプロジェクトに許容されるタスクの最大数

`project.max-contracts`
このプロジェクトに許容される契約の最大数

タスクに関連付けられたリソース制御

タスクに関連付けられたリソース制御には次が含まれます。

`task.max-cpu-time`
このタスクのプロセスで使用できる最長 CPU 時間 (秒)。

`task.max-lwps`
このタスクのプロセスで同時に使用できる LWP の最大数。

`task.max-processes`
このタスクのプロセスで同時に使用できるプロセステーブルスロットの最大数。

注記 - 通常のプロセスとゾンビプロセスはどちらもプロセステーブルスロットを占有します。そのため、`max-processes` リソース制御はゾンビプロセスによってプロセステーブルが使い果たされるのを防ぎます。ゾンビプロセスには定義上 LWP が含まれないため、ゾンビプロセスがプロセステーブルを使い尽くすのを `max-lwps` で防ぐことはできません。

プロセスに関連付けられたリソース制御

プロセスに関連付けられたリソース制御には次が含まれます。

`process.max-address-space`
: このプロセスで使用できる、セグメントサイズの総計としての最大アドレス空間 (バイト)。

`process.max-core-size`
: プロセスによって作成されるコアファイルの最大サイズ (バイト)。

`process.max-cpu-time`
: このプロセスで使用できる最長 CPU 時間 (秒)。

`process.max-file-descriptor`
: このプロセスで使用できる最大のファイル記述子インデックス。

`process.max-file-size`
: このプロセスでの書き込みに使用できる最大ファイルオフセット (バイト)。

`process.max-msg-messages`
: メッセージキュー上のメッセージの最大数。この値は `msgget()` 時にリソース制御からコピーされます。

`process.max-msg-qbytes`
: メッセージキュー上のメッセージの最大数 (バイト)。この値は `msgget()` 時にリソース制御からコピーされます。新しい `project.max-msg-qbytes` の値を設定した場合、そのあとに作成される値でのみ初期化が行われます。新しい `project.max-msg-qbytes` の値は既存の値に影響しません。

`process.max-sem-nsems`
: 1 つのセマフォーセットに許容されるセマフォーの最大数。

`process.max-sem-ops`
: 1 つの `semop()` 呼び出しに許容されるセマフォー操作の最大数。この値は `msgget()` 時にリソース制御からコピーされます。新しい `project.max-sem-ops` の値はそのあとに作成される値の初期化にのみ影響し、既存の値には影響しません。

`process.max-port-events`
: イベントポートごとに許容されるイベントの最大数。

ゾーン規模のリソース制御

ゾーン規模のリソース制御は、ゾーンがインストールされたシステムで使用できます。ゾーン規模のリソース制御は、ゾーン内のすべてのプロセスエンティティーによる総リソース消費を制限します。

`zone.cpu-cap`
: 1 つの非大域ゾーンで消費可能な CPU リソース量に対する絶対的な制限。`project.cpu-cap` 設定と同様、`100` の値は 1 つの CPU の 100% を意味します。`125` の値は 125% になります。CPU キャップの使用時は、100% がシステム上の 1 つの CPU の上限となります。

`zone.cpu-shares`
: ゾーンに対する公平配分スケジューラ (FSS) のCPU 配分の数の制限。スケジューリングクラスは FSS である必要があります。CPU 配分は、まずゾーンに対して割り当てられたあとで、`project.cpu-shares` エントリの指定に従って、ゾーン内のプロジェクトに分配されます。`zone.cpu-shares` の数が大きいゾーンは、配分数が少ないゾーンよりも多くの CPU を使用できます。

`zone.max-locked-memory`
: ゾーンで使用できるロックされた物理メモリーの合計量。

`zone.max-lofi`
: ゾーンによって作成できる `lofi` デバイスの最大数。

`zone.max-lwps`
: このゾーンで同時に使用できる LWP の最大数

`zone.max-msg-ids`
: このゾーンに許容されるメッセージキュー ID の最大数

`zone.max-processes`
: このゾーンで同時に使用できるプロセステーブルスロットの最大数

注記 - zone.max-processes リソース制御は、1 つのゾーンがあまりに多くのプロセステーブルスロットを消費してほかのゾーンに影響を与えるのを防ぐことによって、リソースの隔離性を高めます。ゾーン内のプロジェクト間でのプロセステーブルスロットリソースの割り当ては、project.max-processes リソース制御を使用して制御できます。この制御のグローバルプロパティー名は max-processes です。zone.max-processes リソース制御は、zone.max-lwps リソース制御を含むこともできます。zone.max-processes が設定されていて zone.max-lwps が設定されていない場合、zone.max-lwps はゾーンのブート時に暗黙的に zone.max-processes の 10 倍に設定されます。

通常のプロセスとゾンビプロセスはどちらもプロセステーブルスロットを占有します。そのため、max-processes リソース制御はゾンビプロセスによってプロセステーブルが使い果たされるのを防ぎます。ゾンビプロセスには定義上 LWP が含まれないため、ゾンビプロセスがプロセステーブルを使い尽くすのを max-lwps で防ぐことはできません。

zone.max-sem-ids	このゾーンに許容されるセマフォー ID の最大数
zone.max-shm-ids	このゾーンに許容される共有メモリー ID の最大数
zone.max-shm-memory	このゾーンに許容される共有メモリーの合計量
zone.max-swap	このゾーンのユーザープロセスのアドレス空間マッピングと tmpfs マウントで消費できるスワップの合計量。

詳細は、『Oracle Solaris 11.3 でのリソースの管理』 の 「ゾーン規模のリソース制御」を参照してください。

ゾーン規模のリソース制御の構成については、『Oracle Solaris ゾーン構成リソース』 の 第 1 章, 「非大域ゾーンの構成」および『Oracle Solaris ゾーンの作成と使用』 の 第 1 章, 「非大域ゾーンの計画および構成方法」を参照してください。

zonecfg コマンドを使用して、非大域ゾーンがインストールされたシステムの大域ゾーンにゾーン規模のリソース制御を適用できます。また、setctrl コマンドは大域ゾーンの特権ユーザーとして呼び出された場合にのみ成功します。非大域ゾーン内では、root はゾーン規模のリソース制御を設定できません。

リソース制御で使用するシグナル

リソース制御に設定された各しきい値に対して、次の制限付きシグナルセットを使用できます。

SIGBART
: プロセスを終了します。

SIGXRES
: リソース制御の制限を超えた場合にリソース制御機能によって生成されるシグナル。

SIGHUP
: 開いた回線上でキャリアが検出されなくなった場合に、端末を制御しているプロセスグループにハングアップシグナルである SIGHUP が送信されます。

SIGSTOP
: ジョブ制御シグナル。プロセスを停止します。端末以外からの停止シグナル。

SIGTERM
: プロセスを終了します。ソフトウェアによって送信される終了シグナルです。

SIGKILL
: プロセスを終了します。プログラムを強制終了します。

SIGXFSX
: プロセスを終了します。ファイルサイズの制限超過です。RCTL_GLOBAL_FILE_SIZE プロパティーを持つリソース制御でのみ使用できます。

SIGXCPU
: プロセスを終了します。CPU 時間の制限超過です。RCTL_GLOBAL_CPUTIME プロパティーを持つリソース制御でのみ使用できます。

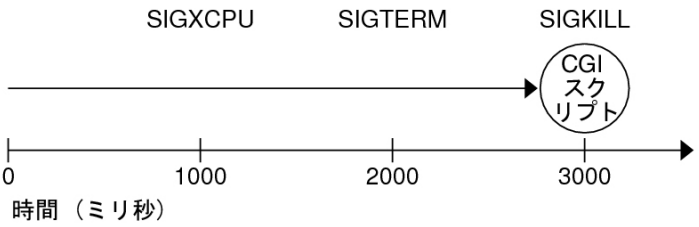
特定の制御のグローバルプロパティーに起因して、その他のシグナルが許可される場合があります。

注記 - 不正なシグナルが指定された `setrctl()` の呼び出しは失敗します。

図 4 シグナルに対する権限レベルの設定

/etc/project

```
cgi-bin:103:cgi-bin scripts:root,apache::\
  process.max-cpu-time=(privileged,1000,signal=SIGXCPU),\
                       (privileged,2000,signal=SIGTERM),\
                       (privileged,3000,signal=SIGKILL),\
```



リソース制御 API 関数

リソース制御 API には、次を行う関数が含まれます。

- 61 ページの「リソース制御のアクション-値ペアの操作」
- 61 ページの「ローカルな変更可能値の操作」
- 62 ページの「ローカルな読み取り専用値の取得」
- 62 ページの「グローバルな読み取り専用アクションの取得」

リソース制御のアクション-値ペアの操作

次のリストは、リソース制御ブロックを設定または取得する関数を示しています。

`setrctl(2)`
`getrctl(2)`

ローカルな変更可能値の操作

次のリストは、ローカルな変更できるリソース制御ブロックに関連する関数を示しています。

rctlblk_set_privilege(3C)
rctlblk_get_privilege(3C)
rctlblk_set_value(3C)
rctlblk_get_value(3C)
rctlblk_set_local_action(3C)
rctlblk_get_local_action(3C)
rctlblk_set_local_flags(3C)
rctlblk_get_local_flags(3C)

ローカルな読み取り専用値の取得

次のリストは、ローカルな読み取り専用のリソース制御ブロックに関連する関数を示しています。

rctlblk_get_recipient_pid(3C)
rctlblk_get_firing_time(3C)
rctlblk_get_enforced_value(3C)

グローバルな読み取り専用アクションの取得

次のリストは、グローバルな読み取り専用のリソース制御ブロックに関連する関数を示しています。

rctlblk_get_global_action(3C)
rctlblk_get_global_flags(3C)

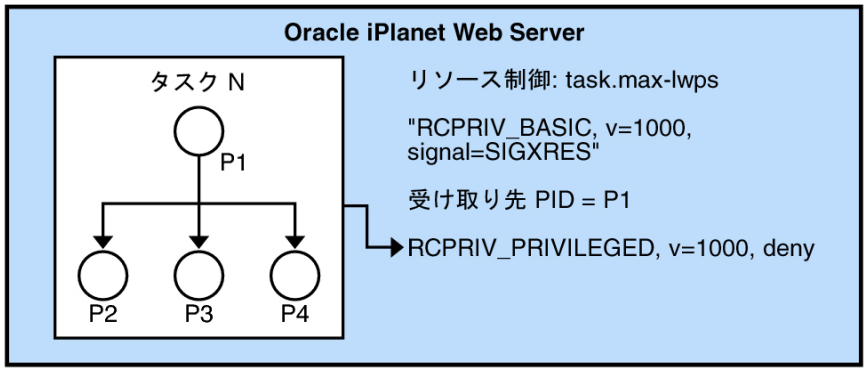
リソース制御のコード例

リソース制御のマスター監視プロセス

次の例は、マスターオブザーバプロセスです。図5 は、マスター監視プロセスのリソース制御を示しています。

注記 - `/etc/project` ファイル内の行ブレークは無効です。行ブレークは、印刷されるページまたは表示されるページ上に例を表示するためだけにここに示されています。`/etc/project` ファイル内の各エントリは個別の行に記述する必要があります。

図 5 マスター監視プロセス



/etc/project

```
...
OracleiPlanetWebServer:200:Oracle iPlanet Web Application Server:root::\
task.max-lwps=(PRIVILEGED,1000,deny)
```

この例の重要なポイントは次のとおりです。

- タスクの制限には特権が付与されているため、アプリケーションは制限を変更したり、シグナルなどのアクションを指定したりできません。マスタープロセスは、同じリソース制御を基本リソース制御としてタスク上に確立することでこの問題を解決します。マスタープロセスは、リソース上の同じ値またはわずかに少ない値を、異なるアクション (シグナル = XRES) で使用します。マスタープロセスはこのシグナルを待機するスレッドを作成します。
- `rctlblk` は不透明です。構造体は動的に割り当てる必要があります。
- スレッドを作成する前に、sigwait(2) の要求に応じてすべてのシグナルがブロックされることに注意してください。
- スレッドはシグナルをブロックするために `sigwait`(2) を呼び出します。`sigwait()` が SIGXRES シグナルを返すと、スレッドがマスタープロセスの子に通知し、これによって使用する LWP 数の削減に適応します。それぞれの子についても、1 つの子に 1 つのスレッドが入った状態で同様にモデル化し、このシグナルを待機して、そのプロセスの LWP 使用率に適切に適応するようにしてください。

```
rctlblk_t *mlwprcb;
sigset_t smask;

/* Omit return value checking/error processing to keep code sample short */
/* First, install a RCPRIV_BASIC, v=1000, signal=SIGXRES rctl */
```

```
mlwprcb = calloc(1, rctlblk_size());  /* rctl blocks are opaque: */
       rctlblk_set_value(mlwprcb, 1000);
       rctlblk_set_privilege(mlwprcb, RCPRIV_BASIC);
       rctlblk_set_local_action(mlwprcb, RCTL_LOCAL_SIGNAL, SIGXRES);
       if (setrctl("task.max-lwps", NULL, mlwprcb, RCTL_INSERT) == -1) {
           perror("setrctl");
           exit (1);
       }

/* Now, create the thread which waits for the signal */
        sigemptyset(&smask);
        sigaddset(&smask, SIGXRES);
        thr_sigsetmask(SIG_BLOCK, &smask, NULL);
thr_create(NULL, 0, sigthread, (void *)SIGXRES, THR_DETACHED, NULL));

/* Omit return value checking/error processing to keep code sample short */

void *sigthread(void *a)
{
        int sig = (int)a;
        int rsig;
        sigset_t sset;

        sigemptyset(&sset);
        sigaddset(&sset, sig);

        while (1) {
                rsig = sigwait(&sset);
          if (rsig == SIGXRES) {
              notify_all_children();
              /* e.g. sigsend(P_PID, child_pid, SIGXRES); */
       }
        }
}
```

特定のリソース制御の値-アクションペアをすべて表示

次の例では、特定のリソース制御 `task.max-lwps` の値-アクションペアがすべて一覧表示されています。この例の重要なポイントは、`getrctl`(2) が 2 つのリソース制御ブロックを取り、RCTL_NEXT フラグのリソース制御ブロックを返すことです。すべてのリソース制御ブロックを繰り返し処理するには、ここに示すように `rcb_tmp` `rctl` ブロックを使用してリソース制御ブロックの値を繰り返しスワップします。

```
rctlblk_t *rcb1, *rcb2, *rcb_tmp;
 ...
/* Omit return value checking/error processing to keep code sample short */
rcb1 = calloc(1, rctlblk_size()); /* rctl blocks are opaque: */
                                  /* "rctlblk_t rcb" does not work */
rcb2 = calloc(1, rctlblk_size());
getrctl("task.max-lwps", NULL, rcb1, RCTL_FIRST);
while (1) {
     print_rctl(rcb1);
     rcb_tmp = rcb2;
     rcb2 = rcb1;
     rcb1 = rcb_tmp;        /* swap rcb1 with rcb2 */
     if (getrctl("task.max-lwps", rcb2,  rcb1, RCTL_NEXT) == -1) {
          if (errno == ENOENT) {
```

```
                break;
        } else {
                perror("getrctl");
                exit (1);
        }
        }
}
```

project.cpu-shares の設定と新しい値の追加

この例の重要なポイントは次のとおりです。

- この例は、83 ページの「pool.comment プロパティーの設定と新規プロパティーの追加」に示されている例と似ています。
- 64 ページの「特定のリソース制御の値-アクションペアをすべて表示」のバッファースワッピングではなく bcopy() を使用します。
- リソース制御値を変更するには、RCTL_REPLACE フラグを指定して setrctl() を呼び出します。新しいリソース制御ブロックは、新しい制御値を除いて古いリソース制御ブロックと同じです。

```
rctlblk_set_value(blk1, nshares);
if (setrctl("project.cpu-shares", blk2, blk1, RCTL_REPLACE) != 0)
```

この例ではプロジェクトの CPU 配分割り当て (project.cpu-shares) を取得し、その値を *nshares* に変更します。

```
/* Omit return value checking/error processing to keep code sample short */
blk1 = malloc(rctlblk_size());
getrctl("project.cpu-shares", NULL, blk1, RCTL_FIRST);
my_shares = rctlblk_get_value(blk1);
printout_my_shares(my_shares);
/* if privileged, do the following to */
/* change project.cpu-shares to "nshares" */
blk1 = malloc(rctlblk_size());
blk2 = malloc(rctlblk_size());
if (getrctl("project.cpu-shares", NULL, blk1, RCTL_FIRST) != 0) {
     perror("getrctl failed");
     exit(1);
}
bcopy(blk1, blk2, rctlblk_size());
rctlblk_set_value(blk1, nshares);
if (setrctl("project.cpu-shares", blk2, blk1, RCTL_REPLACE) != 0) {
     perror("setrctl failed");
     exit(1);
}
```

リソース制御ブロックを使用して LWP 制限を設定

次の例では、超えることのできない 3000 LWP の特権付き制限をアプリケーションが設定しています。さらに、アプリケーションは 2000 LWP の基本制限も設定していま

す。この制限を超えると、SIGXRES がアプリケーションに送信されます。SIGXRES を受け取るとアプリケーションがすぐにその子プロセスに通知を送信し、次にその子プロセスが、使用するまたは必要となる LWP 数を減らす場合があります。

```
/* Omit return value and error checking */

#include <rctl.h>

rctlblk_t *rcb1, *rcb2;

/*
         * Resource control blocks are opaque
         * and must be explicitly allocated.
         */
rcb1 = calloc(rctlblk_size());

rcb2 = calloc(rctlblk_size());


/* Install an RCPRIV_PRIVILEGED, v=3000: do not allow more than 3000 LWPs */
rctlblk_set_value(rcb1, 3000);
rctlblk_set_privilege(rcb1, RCPRIV_PRIVILEGED);
rctlblk_set_local_action(rcb1, RCTL_LOCAL_DENY);
setrctl("task.max-lwps", NULL, rcb1, RCTL_INSERT);


/* Install an RCPRIV_BASIC, v=2000 to send SIGXRES when LWPs exceeds 2000 */
rctlblk_set_value(rcb2, 2000);
rctlblk_set_privilege(rcb2, RCPRIV_BASIC);
rctlblk_set_local_action(rcb2, RCTL_LOCAL_SIGNAL, SIGXRES);
setrctl("task.max-lwps", NULL, rcb2, RCTL_INSERT);
```

リソース制御に関するプログラミングの問題

アプリケーションを記述する場合は、次の問題を考慮してください。

- リソース制御ブロックは不透明です。制御ブロックは動的に割り当てる必要があります。
- 基本リソース制御がタスクまたはプロジェクト上で確立されている場合、このリソース制御を確立するプロセスがオブザーバになります。このリソース制御ブロックのアクションがオブザーバに適用されます。ただし、一部のリソースはこの方法では監視できません。
- 特権付きリソース制御がタスクまたはプロジェクトに設定されている場合、オブザーバプロセスは存在しません。ただし、限度を超えるプロセスはリソース制御アクションの対象になります。
- グローバルとローカルという各タイプについて 1 つのアクションだけが許可されます。
- プロセスごと、リソース制御ごとに 1 つの基本 `rctl` だけが許可されます。

ゾーンのリソース使用率をモニターするための zonestat ユーティリティー

zonestat ユーティリティーは、現在実行中のゾーンの CPU、メモリー、およびリソース制御の使用効率について報告します。各ゾーンの使用効率は、システムのリソースとゾーンの構成済み制限の両方の割合 (パーセント) として報告されます。詳細は、第7章「Oracle Solaris ゾーンでのリソース管理アプリケーションに関する設計上の考慮事項」、zonestat(1) のマニュアルページ、および『Oracle Solaris ゾーンの作成と使用』 の 「zonestat ユーティリティーを使用したアクティブなゾーンの統計情報の報告」を参照してください。

♦ ♦ ♦ 第 6 章

6 リソースプール

この章では、リソースプールおよびそれらのプロパティーについて説明します。

リソースプールの概要

リソースプールは、プロセッサセットおよびスレッドスケジューリングクラスを管理するためのフレームワークを提供します。リソースプールはシステムリソースの区分に使用されます。リソースプールを使用すると、作業負荷によって特定のリソースが重複して消費されないように、作業負荷を分離できます。リソースを確保すると、さまざまな作業負荷が混在するシステム上で予測どおりの性能を得ることができます。

リソースプールの概要およびリソースプールを管理するコマンドの例については、『Oracle Solaris 11.3 でのリソースの管理』 の 第 12 章,「リソースプールについて」および『Oracle Solaris 11.3 でのリソースの管理』 の 第 13 章,「リソースプールの作成と管理のタスク」を参照してください。

プロセッサセットがシステム上の CPU を区切られたエンティティーにグループ化し、1 つまたは複数のプロセスをそこで排他的に実行できます。プロセッサセットの範囲を超えてプロセスを拡張したり、ほかのプロセスをプロセッサセットに拡張したりすることはできません。プロセッサセットを使用すると、特性の似たタスクを一緒にグループ化して、CPU 使用のハード上限を設定できます。

リソースプールフレームワークを使用すると、CPU カウントの最大要件と最小要件を指定してソフトプロセッサセットを定義できます。また、フレームワークはそのプロセッサセット用にハード定義されたスケジューリングクラスを提供します。

ゾーンは、ゾーン構成の `pool` プロパティーを使用してリソースプールにバインドできます。ゾーンはゾーンの作成時に指定したプールにバインドされます。プール構

成は大域ゾーンからのみ変更できます。ゾーンは複数のプールにまたがることはできません。ゾーン内のすべてのプロセスは同じプールで実行されます。ただし、複数のゾーンを同じリソースプールにバインドすることはできます。

リソースプールは次を定義します。

- プロセッサセットグループ
- スケジューリングクラス

スケジューリングクラス

スケジューリングクラスは、アルゴリズムロジックに基づいたスレッドにさまざまな CPU アクセス特性を提供します。スケジューリングクラスには次が含まれます。

- リアルタイムスケジューリングクラス
- 対話型スケジューリングクラス
- 固定優先度スケジューリングクラス
- 時分割スケジューリングクラス
- 公平配分スケジューリングクラス

公平配分スケジューラの概要および公平配分スケジューラを管理するコマンドの例については、『Oracle Solaris 11.3 でのリソースの管理』 の 第 8 章, 「公平配分スケジューラについて」および『Oracle Solaris 11.3 でのリソースの管理』 の 第 9 章, 「公平配分スケジューラの管理のタスク」を参照してください。

CPU セット内でスケジューリングクラスを混在させないでください。スケジューリングクラスが CPU セット内で混在していると、システムパフォーマンスが不安定で予測できなくなる可能性があります。プロセッサセットを使用して、アプリケーションを特性ごとに分離してください。アプリケーションが最適なパフォーマンスを発揮するようにスケジューリングクラスを割り当てます。個々のスケジューリングクラスの特性については、`priocntl`(1) を参照してください。

リソースプールの概要およびプールをいつ使用するかについては、第6章「リソースプール」を参照してください。

動的リソースプールの制約および目標

`libpool` ライブラリは、プール機能で管理されるさまざまなエンティティーに使用できるプロパティーを定義します。各プロパティーは次のカテゴリに分類されます。

構成の制約

制約はプロパティーの境界を定義します。一般的な制約は、`libpool` 構成内で指定される最大割り当ておよび最小割り当てです。

目標

目標は、現在の構成のリソース割り当てを変更して、確立される制約を監視する新しい構成の候補を生成します。目標には次のカテゴリがあります。

作業負荷に依存する目標
作業負荷に依存する目標は、作業負荷によって課される条件に応じて変化します。作業負荷に依存する目標の例は、使用率の目標です。

作業負荷に依存しない目標
作業負荷に依存しない目標は、作業負荷によって課される条件に応じて変化しません。作業負荷に依存しない目標の例は、CPU ローカリティーの目標です。

目標には、目標の重要度を示す接頭辞をオプションで指定できます。目標にこの接頭辞 (0 から INT64_MAX までの整数) が乗算され、目標の重要性が決定されます。

使用例については、『Oracle Solaris 11.3 でのリソースの管理』 の 「構成の制約を設定する方法」および『Oracle Solaris 11.3 でのリソースの管理』 の 「構成の目標を定義する方法」を参照してください。

システムプロパティー

`system.bind-default` (書き込み可能なブール値)
指定されたプールが `/etc/project` に見つからない場合、`pool.default` プロパティーを TRUE に設定してプールにバインドしてください。

`system.comment` (書き込み可能な文字列)
システムのユーザー説明。構成が `poolcfg` ユーティリティーによって開始される場合を除き、デフォルトのプールコマンドによって `system.comment` が使用されることはありません。この場合、システムはその構成用に `system.comment` プロパティー内の通知メッセージを配置します。

`system.name` (書き込み可能な文字列)
構成のユーザー名。

`system.version` (読み取り専用の整数)
この構成を操作するのに必要な `libpool` のバージョン。

プールのプロパティー

`pool.default` と `pool.sys_id` を除くプールのプロパティーはすべて書き込み可能です。

`pool.active` (書き込み可能なブール値)

TRUE の場合、このプールをアクティブとしてマークします。

`pool.comment` (書き込み可能な文字列)

プールのユーザー説明。

`pool.default` (読み取り専用のブール値)

TRUE の場合、このプールをデフォルトのプールとしてマークします。`system.bind-default` プロパティーを参照してください。

`pool.importance` (書き込み可能な整数)

このプールの相対的な重要性。リソースの競合が発生した場合の解決に使用されます。

`pool.name` (書き込み可能な文字列)

プールのユーザー名。`setproject`(3PROJECT) は、`project`(4) データベース内で `project.pool` プロジェクト属性の値として `pool.name` を使用します。

`pool.scheduler` (書き込み可能な文字列)

このプールのコンシューマがバインドされるスケジューラクラス。このプロパティーはオプションであり、指定されていない場合、このプールのコンシューマのスケジューラバインディングは影響を受けません。個々のスケジューリングクラスの特性については、`priocntl`(1) を参照してください。スケジューラクラスには次が含まれます。

- リアルタイムスケジューラの RT
- 時分割スケジューラの TS
- 対話型スケジューラの IA
- 公平配分スケジューラの FSS
- 固定優先度スケジューラの FX

`pool.sys_id` (読み取り専用の整数)

これはシステムによって割り当てられるプール ID です。

プロセッサセットのプロパティー

`pset.comment` (書き込み可能な文字列)

リソースのユーザー説明。

`pset.default` (読み取り専用のブール値)

デフォルトのプロセッサセットを識別します。

`pset.load` (読み取り専用の符号なし整数)
: このプロセッサセットの負荷。もっとも低い値は 0 です。値は、システム実行キューにあるジョブの数によって測定されるセット上の負荷に応じて、直線的に増加します。

`pset.max` (書き込み可能な符号なし整数)
: このプロセッサセットで許可される CPU の最大数。

`pset.min` (書き込み可能な符号なし整数)
: このプロセッサセットで許可される CPU の最小数。

`pset.name` (書き込み可能な文字列)
: リソースのユーザー名。

`pset.size` (読み取り専用の符号なし整数)
: このプロセッサセット内の現在の CPU の数。

`pset.sys_id` (読み取り専用の整数)
: システムによって割り当てられるプロセッサセット ID。

`pset.type` (読み取り専用文字列)
: リソースタイプの名前を指定します。すべてのプロセッサセットの値が `pset` です。

`pset.units` (読み取り専用文字列)
: サイズに関するプロパティーの意味を識別します。すべてのプロセッサセットの値が `population` です。

`cpu.comment` (書き込み可能な文字列)
: CPU のユーザー説明。

`pset.resmin`
: `pset.policy == minmax` (書き込み可能な符号なし整数) の場合
: プロセッサセットの CPU の最小数。
: `pset.policy == assigned` (読み取り専用符号なし整数) の場合
: 割り当てられている CPU、コア、またはソケットの数。

`pset.resmax`
: `pset.policy == minmax` (書き込み可能な符号なし整数) の場合
: プロセッサセットの CPU の最大数。
: `pset.policy == assigned` (読み取り専用符号なし整数) の場合
: 割り当てられている CPU、コア、またはソケットの数。

pset.ressize (読み取り専用文字列)
: プロセッサセット内の CPU、コア、またはソケットの現在の数。pset.policy == minmax の場合、これは常に CPU の数です。

pset.restype
: 割り当てられている CPU リソースのタイプ: CPU、コア、またはソケット。pset.policy == minmax の場合、これは常に CPU です。

pset.reslist
: 割り当てられている CPU、コア、またはソケットの ID。pset.policy == minmax の場合、これは常に空白です。

pset.policy
: プロセッサセットが量ベースの割り当てと固有の割り当てのどちらを使用しているかに応じて、minmax または assigned になります。

libpool を使用したプール構成の操作

libpool(3LIB) プール構成ライブラリは、プール構成ファイルの読み取りおよび書き込み用のインタフェースを定義します。このライブラリは、実行中のオペレーティングシステム構成になるように既存の構成をコミットするためのインタフェースも定義します。<pool.h> ヘッダーは、すべてのライブラリサービスのタイプおよび関数の宣言を提供します。

リソースプール機能は、プロセスにバインドできるリソースを、プールと呼ばれる共通の抽象概念にまとめます。プロセッサセットおよびその他のエンティティーは、永続的な方法で構成、グループ化、およびラベル付けできます。作業負荷コンポーネントは、システム全体のリソースのサブセットに関連付けることができます。libpool(3LIB) ライブラリは、リソースプール機能にアクセスするための C 言語 API を提供します。pooladm(1M)、poolbind(1M)、および poolcfg(1M) は、シェルからコマンド呼び出しを介してリソースプール機能を使用できるようにします。

psets の操作

次のリストは、psets の作成または破棄、および psets の操作に関する機能を示しています。

processor_bind(2)
: LWP (軽量プロセス) または LWP のセットを、指定したプロセッサにバインドします。

pset_assign(2)
: プロセッサをプロセッサセットに割り当てます。

pset_bind(2)	1 つ以上の LWP (軽量プロセス) をプロセッサセットにバインドします。
pset_create(2)	プロセッサを含まない空のプロセッサセットを作成します。
pset_destroy(2)	プロセッサセットを破棄して、構成要素である関連付けられたプロセッサおよびプロセスを解放します。
pset_setattr(2), pset_getattr(2)	プロセッサセット属性を設定または取得します。

リソースプール API 関数

このセクションでは、すべてのリソースプール関数を一覧表示します。各関数に、マニュアルページへのリンクとその関数の簡単な目的を示しています。関数は、アクションとクエリーのどちらを実行するかに応じて 2 つのグループに分類されます。

- 75 ページの「リソースプールおよび関連付けられた要素を操作するための関数」
- 78 ページの「リソースプールおよび関連付けられた要素にクエリーを実行するための関数」

スワップセット用にインポートされた libpool のインタフェースは、このドキュメントで定義されたインタフェースと同じです。

リソースプールおよび関連付けられた要素を操作するための関数

このセクションで示されているインタフェースは、プールおよび関連付けられた要素に関するアクションを実行するためのものです。

pool_associate(3POOL)
: 指定されたプールにリソースを関連付けます。

pool_component_to_elem(3POOL)
: 指定されたコンポーネントをプール要素タイプに変換します。

pool_conf_alloc(3POOL)
: プール構成を作成します。

pool_conf_close(3POOL)
: 指定されたプール構成を閉じて、関連付けられたリソースを解放します。

`pool_conf_commit(3POOL)`
: 指定されたプール構成への変更を永続ストレージにコミットします。

`pool_conf_export(3POOL)`
: 指定された場所に指定された構成を保存します。

`pool_conf_free(3POOL)`
: プール構成を解放します。

`pool_conf_open(3POOL)`
: 指定された場所にプール構成を作成します。

`pool_conf_remove(3POOL)`
: 構成の永続ストレージを削除します。

`pool_conf_rollback(3POOL)`
: プール構成の永続ストレージに保持されている状態に構成の状態を復元します。

`pool_conf_to_elem(3POOL)`
: 指定されたプール構成をプール要素タイプに変換します。

`pool_conf_update(3POOL)`
: カーネル状態のライブラリスナップショットを更新します。

`pool_create(3POOL)`
: デフォルトプロパティーと各タイプのデフォルトリソースを使用して新しいプールを作成します。

`pool_destroy(3POOL)`
: 指定されたプールを破棄します。関連付けられたリソースは変更されません。

`pool_dissociate(3POOL)`
: 特定のリソースとプール間の関連付けを削除します。

`pool_put_property(3POOL)`
: 要素の名前付きプロパティーを指定された値に設定します。

`pool_resource_create(3POOL)`
: 指定された構成に対して、指定された名前とタイプで新しいリソースを作成します。

`pool_resource_destroy(3POOL)`
: 指定されたリソースを構成ファイルから削除します。

pool_resource_to_elem(3POOL)
: 指定されたプールリソースをプール要素タイプに変換します。

pool_resource_transfer(3POOL)
: 基本単位をソースリソースからターゲットリソースに転送します。

pool_resource_xtransfer(3POOL)
: 指定されたコンポーネントをソースリソースからターゲットリソースに転送します。

pool_rm_property(3POOL)
: 名前付きプロパティーを要素から削除します。

pool_set_binding(3POOL)
: 指定されたプロセスを、実行中のシステム上のプールに関連付けられたリソースにバインドします。

pool_set_status(3POOL)
: プール機能の現在の状態を変更します。

pool_to_elem(3POOL)
: 指定されたプールをプール要素タイプに変換します。

pool_value_alloc(3POOL)
: プールのプロパティー値用に不透明なコンテナを割り当てて返します。

pool_value_free(3POOL)
: 割り当てられているプロパティー値を解放します。

pool_value_set_bool(3POOL)
: `boolean` 型のプロパティー値を設定します。

pool_value_set_double(3POOL)
: `double` 型のプロパティー値を設定します。

pool_value_set_int64(3POOL)
: `int64` 型のプロパティー値を設定します。

pool_value_set_name(3POOL)
: プールプロパティーの `name=value` ペアを設定します。

pool_value_set_string(3POOL)
: 渡された文字列をコピーします。

pool_value_set_uint64(3POOL)
: uint64 型のプロパティー値を設定します。

リソースプールおよび関連付けられた要素にクエリーを実行するための関数

このセクションで示されているインタフェースは、プールおよび関連付けられた要素に関するクエリーを実行するためのものです。

pool_component_info(3POOL)
: 指定されたコンポーネントを説明する文字列を返します。

pool_conf_info(3POOL)
: 構成全体を説明する文字列を返します。

pool_conf_location(3POOL)
: 指定された特定の構成に対して pool_conf_open() に提供された場所の文字列を返します。

pool_conf_status(3POOL)
: プール構成の有効性ステータスを返します。

pool_conf_validate(3POOL)
: 指定された構成の内容の有効性をチェックします。

pool_dynamic_location(3POOL)
: プールフレームワークによって動的構成を格納するのに使用された場所を返します。

pool_error(3POOL)
: リソースプール構成のライブラリ関数を呼び出すことによって記録された最後の失敗のエラー値を返します。

pool_get_binding(3POOL)
: 指定されたプロセスがバインドされるリソースのセットが含まれる、実行中のシステム上のプールの名前を返します。

pool_get_owning_resource(3POOL)
: 指定されたコンポーネントを現在含んでいるリソースを返します。

pool_get_pool(3POOL)
: 指定された構成から指定された名前のプールを返します。

pool_get_property(3POOL)
: 名前付きプロパティーの値を要素から取得します。

pool_get_resource(3POOL)
: 指定された構成から指定された名前とタイプのリソースを返します。

pool_get_resource_binding(3POOL)
: 指定されたプロセスがバインドされるリソースのセットが含まれる、実行中のシステム上のプールの名前を返します。

pool_get_status(3POOL)
: プール機能の現在の状態を取得します。

pool_info(3POOL)
: 指定されたプールの説明を返します。

pool_query_components(3POOL)
: 指定されたプロパティーのリストに一致するすべてのリソースコンポーネントを取得します。

pool_query_pool_resources(3POOL)
: 現在、プールに関連付けられているリソースの NULL 終端配列を返します。

pool_query_pools(3POOL)
: 指定されたプロパティーのリストに一致するプールのリストを返します。

pool_query_resource_components(3POOL)
: 指定されたリソースを構成するコンポーネントの NULL 終端配列を返します。

pool_query_resources(3POOL)
: 指定されたプロパティーのリストに一致するリソースのリストを返します。

pool_resource_info(3POOL)
: 指定されたリソースの説明を返します。

pool_resource_type_list(3POOL)
: このプラットフォームのプールフレームワークでサポートされているリソースタイプを列挙します。

pool_static_location(3POOL)
: プールフレームワークのインスタンス化のデフォルト構成を格納するためにプールフレームワークによって使用された場所を返します。

pool_strerror(3POOL)
: 有効な各プールエラーコードの説明を返します。

`pool_value_get_bool(3POOL)`
: `boolean` 型のプロパティー値を取得します。

`pool_value_get_double(3POOL)`
: `double` 型のプロパティー値を取得します。

`pool_value_get_int64(3POOL)`
: `int64` 型のプロパティー値を取得します。

`pool_value_get_name(3POOL)`
: 指定されたプールプロパティーに割り当てられた名前を返します。

`pool_value_get_string(3POOL)`
: `string` 型のプロパティー値を取得します。

`pool_value_get_type(3POOL)`
: 指定されたプール値に含まれるデータの型を返します。

`pool_value_get_uint64(3POOL)`
: `uint64` 型のプロパティー値を取得します。

`pool_version(3POOL)`
: プールライブラリのバージョン番号を取得します。

`pool_walk_components(3POOL)`
: リソースに含まれているすべてのコンポーネントのコールバックを呼び出します。

`pool_walk_pools(3POOL)`
: 構成内で定義されているすべてのプールのコールバックを呼び出します。

`pool_walk_properties(3POOL)`
: 指定された要素に定義されたすべてのプロパティーのコールバックを呼び出します。

`pool_walk_resources(3POOL)`
: プールに関連付けられているすべてのリソースのコールバックを呼び出します。

リソースプールのコード例

このセクションでは、リソースプールインタフェースのコード例を示します。

リソースプール内の CPU 数の確認

sysconf(3C) は、システム全体の CPU 数に関する情報を提供します。次の例は、特定のアプリケーションのプール pset に定義された CPU 数の確認の粒度を示しています。

この例の重要なポイントは次のとおりです。

- pvals[] は NULL 終端配列になります。
- pool_query_pool_resources() は、アプリケーションのプール my_pool から pvals 配列型 pset に一致するすべてのリソースのリストを返します。プールは pset リソースのインスタンスを 1 つしか持てないため、各インスタンスは常に nelem で返されます。reslist[] には、唯一の要素である pset リソースが含まれます。

```
pool_value_t *pvals[2] = {NULL};  /* pvals[] should be NULL terminated */

/* NOTE: Return value checking/error processing omitted */
/* in all examples for brevity */

conf_loc = pool_dynamic_location();
conf = pool_conf_alloc();
pool_conf_open(conf, conf_loc, PO_RDONLY);
my_pool_name = pool_get_binding(getpid());
my_pool = pool_get_pool(conf, my_pool_name);
pvals[0] = pool_value_alloc();
pvals2[2] = { NULL, NULL };
pool_value_set_name(pvals[0], "type");
pool_value_set_string(pvals[0], "pset");

reslist = pool_query_pool_resources(conf, my_pool, &nelem, pvals);
pool_value_free(pvals[0]);
pool_query_resource_components(conf, reslist[0], &nelem, NULL);
printf("pool %s: %u cpu", my_pool_ name, nelem);
pool_conf_close(conf);
```

すべてのリソースプールの表示

次の例では、アプリケーションのプール pset 内で定義されているすべてのリソースプールが一覧表示されています。

この例の重要なポイントは次のとおりです。

- 動的 conf ファイルを読み取り専用の PO_RDONLY で開きます。pool_query_pools() は pl のプールのリストおよび nelem のプール数を返します。要素から pval 値に pool.name プロパティーを取得するには、各プールに対して pool_get_property() を呼び出します。
- pool_get_property() は pool_to_elem() を呼び出して libpool エンティティーを不透明な値に変換します。pool_value_get_string() は不透明なプール値から文字列を取得します。

```
conf = pool_conf_alloc();
pool_conf_open(conf, pool_dynamic_location(), PO_RDONLY);
pl = pool_query_pools(conf, &nelem, NULL);
pval = pool_value_alloc();
for (i = 0; i < nelem; i++) {
    pool_get_property(conf, pool_to_elem(conf, pl[i]), "pool.name", pval);
    pool_value_get_string(pval, &fname);
    printf("%s\n", name);
}
pool_value_free(pval);
free(pl);
pool_conf_close(conf);
```

指定されたプールのプール統計の報告

次の例は、指定されたプールの統計を報告しています。

この例の重要なポイントは次のとおりです。

- `pool_query_pool_resources()` は、`rl` のすべてのリソースのリストを取得します。`pool_query_pool_resources()` の最後の引数が NULL のため、すべてのリソースが返されます。各リソースについて、`name`、`load`、および `size` プロパティーが読み取られ、出力されます。
- `strdup()` への呼び出しは、ローカルメモリーを割り当て、`get_string()` によって返される文字列をコピーします。`get_string()` への呼び出しは、`get_property()` への次の呼び出しによって解放されるポインタを返します。`strdup()` への呼び出しが含まれていない場合、文字列を次に参照した際に、セグメンテーション違反によってアプリケーションにエラーが発生する可能性があります。

```
printf("pool %s\n:" pool_name);
pool = pool_get_pool(conf, pool_name);
rl = pool_query_pool_resources(conf, pool, &nelem, NULL);
for (i = 0; i < nelem; i++) {
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]), "type", pval);
  pool_value_get_string(pval, &type);
  type = strdup(type);
  snprintf(prop_name, 32, "%s.%s", type, "name");
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]),
        prop_name, pval);
  pool_value_get_string(val, &res_name);
  res_name = strdup(res_name);
  snprintf(prop_name, 32, "%s.%s", type, "load");
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]),
        prop_name, pval);
  pool_value_get_uint64(val, &load);
  snprintf(prop_name, 32, "%s.%s", type, "size");
  pool_get_property(conf, pool_resource_to_elem(conf, rl[i]),
        prop_name, pval);
  pool_value_get_uint64(val, &size);
  printf("resource %s: size %llu load %llu\n", res_name, size, load);
  free(type);
  free(res_name);
}
free(rl);
```

pool.comment プロパティーの設定と新規プロパティーの追加

次の例では、pset の pool.comment プロパティーを設定します。また、この例では pool.newprop に新規プロパティーも作成します。

この例の重要なポイントは次のとおりです。

- 静的構成ファイルで PO_RDWR を使用して pool_conf_open() への呼び出しを行う場合は、呼び出し元が root である必要があります。
- このユーティリティーの実行後にこれらの変更を pset にコミットするには、pooladm -c コマンドを発行します。ユーティリティーに変更をコミットさせるには、ゼロ以外の第 2 引数を指定して pool_conf_commit() を呼び出します。

```
pool_set_comment(const char *pool_name, const char *comment)
{
  pool_t *pool;
  pool_elem_t *pool_elem;
  pool_value_t *pval = pool_value_alloc();
  pool_conf_t  *conf = pool_conf_alloc();
  /* NOTE: need to be root to use PO_RDWR on static configuration file */
  pool_conf_open(conf, pool_static_location(), PO_RDWR);
  pool = pool_get_pool(conf,  pool_name);
  pool_value_set_string(pval, comment);
  pool_elem = pool_to_elem(conf, pool);
  pool_put_property(conf, pool_elem, "pool.comment", pval);
  printf("pool %s: pool.comment set to %s\n:" pool_name, comment);
  /* Now, create a new property, customized to installation site */
  pool_value_set_string(pval, "New String Property");
  pool_put_property(conf, pool_elem, "pool.newprop", pval);
  pool_conf_commit(conf, 0); /* NOTE: use 0 to ensure only */
                             /* static file gets updated */
  pool_value_free(pval);
  pool_conf_close(conf);
  pool_conf_free(conf);
  /* NOTE: Use "pooladm -c" later, or pool_conf_commit(conf, 1) */
  /* above for changes to the running system */
}
```

プールのコメントを変更して新規プールプロパティーを追加するもう 1 つの方法は、poolcfg(1M) を使用する方法です。

```
poolcfg -c 'modify pool pool-name (string pool.comment = "cmt-string")'
poolcfg -c 'modify pool pool-name (string pool.newprop =
                                   "New String Property")'
```

リソースプールに関するプログラミングの問題

アプリケーションを記述する場合は、次の問題を考慮してください。

- サイトごとに独自のプロパティーリストをプール構成に追加できます。

複数の構成ファイルに複数の構成を保持できます。システム管理者は、異なるファイルをコミットして、異なるタイムスロットのリソース消費に対して変更を反映できます。これらのタイムスロットには、負荷の状態に応じて、日、週、月、または季節単位で異なる時間を含めることができます。

- リソースセットはプール間で共有できますが、プールは特定のタイプのリソースセットを 1 つしか持ちません。そのため、`pset_default` は、デフォルトと特定のアプリケーションのデータベースプールとの間で共有できます。
- `pool_value_*()` インタフェースは慎重に使用してください。文字列のプール値のメモリー割り当ての問題に留意してください。82 ページの「指定されたプールのプール統計の報告」を参照してください。

Oracle Solaris ゾーンでリソースプールをモニターするための zonestat ユーティリティー

`zonestat` ユーティリティーは、現在実行中のゾーンの CPU、メモリー、およびリソース制御の使用効率の報告に使用できます。各ゾーンの使用効率は、システムのリソースとゾーンの構成済み制限の両方の割合 (パーセント) として報告されます。詳細は、`zonestat(1)` のマニュアルページおよび『Oracle Solaris ゾーンの作成と使用』を参照してください。

♦ ♦ ♦ 第 7 章

7 Oracle Solaris ゾーンでのリソース管理アプリケーションに関する設計上の考慮事項

この章では、Oracle Solaris ゾーンテクノロジの簡単な概要を示し、開発者がリソース管理アプリケーションを作成する際に発生する可能性のある潜在的な問題について説明します。

Oracle Solaris ゾーンの概要

ゾーンは、Oracle Solaris オペレーティングシステムの単一インスタンス内に作成された、仮想化されたオペレーティングシステム環境です。Oracle Solaris ゾーンは、隔離されたセキュアな環境をアプリケーションに提供する区分技術です。ゾーンを作成すると、そのアプリケーション実行環境で実行されるプロセスは、システムのほかの部分から隔離されます。このように隔離されているので、あるゾーンで実行中のプロセスが、ほかのゾーンで実行中のプロセスからモニタリングまたは操作されることがありません。root 資格で実行されているプロセスであっても、ほかのゾーンの活動を表示したり影響を与えたりすることはできません。また、ゾーンにより、ゾーンが配備されたシステムの物理的属性からアプリケーションを分離する抽象層も提供されます。このような属性の例として、物理デバイスパスやネットワークインタフェース名があります。Oracle Solaris 11.3 リリースでは、デフォルトの非大域ゾーンブランドは `solaris` ゾーンです。

デフォルトでは、すべてのシステムに大域ゾーンが存在します。スーパーユーザー (root) モデルに似て、大域ゾーンでは Oracle Solaris 環境をグローバルに見渡せます。ほかのすべてのゾーンは非大域ゾーンと呼ばれます。非大域ゾーンは、スーパーユーザーモデルの非特権ユーザーに似ています。非大域ゾーン内のプロセスは、そのゾーン内のプロセスおよびファイルのみを制御できます。通常、システム管理作業は主に大域ゾーンで実行されます。まれに、システム管理者が隔離されている必要がある場合は、特権付きアプリケーションを非大域ゾーンで使用できます。ただし、一般に、リソース管理アクティビティーは大域ゾーンで行われます。

追加の隔離として、読み取り専用ルートを持つ `solaris` ゾーンを構成できます。『Oracle Solaris ゾーンの作成と使用』 の 第 11 章,「不変ゾーンの構成と管理」を参照してください。

`solaris` ゾーンの詳細については、『Oracle Solaris ゾーンの作成と使用』を参照してください。

Oracle Solaris ゾーンの IP ネットワーク

非大域ゾーンがそれ独自の排他的 IP インスタンスを与えられているか、それとも IP 層の構成および状態を大域ゾーンと共有しているかによって、ゾーンの IP ネットワークは 2 つの異なる方法で構成できます。ゾーンは、デフォルトでは排他的 IP タイプで作成されます。ネットワーク接続構成が指定されない場合、仮想ネットワーク (VNIC) は `zonecfg anet` リソースによってゾーン構成内に自動的に含まれます。

排他的 IP ゾーンは 0 (ゼロ) 以上の VNIC インタフェース名を割り当てられ、これらのネットワークインタフェースに対して任意のパケットの送受信、スヌープ、および IP 構成の変更 (IP アドレスとルーティングテーブルも含む) を行うことができます。これらの変更はシステム上のほかの IP インスタンスには影響しません。

`zonecfg` コマンドおよびゾーンのネットワークの詳細については、『Oracle Solaris ゾーン構成リソース』を参照してください。

Oracle Solaris ゾーン内のアプリケーションについて

すべてのアプリケーションは、従来の Oracle Solaris オペレーティングシステムの場合と同様に、大域ゾーン内で完全に機能します。大部分のアプリケーションは、特権を必要としないかぎり、非大域環境で問題なく動作します。アプリケーションが特権を必要とする場合、開発者は、どの特権が必要か、および特定の特権がどのように使用されるかを詳しく調べる必要があります。特権が必要な場合、システム管理者は必要な特権をゾーンに割り当てることができます。『Oracle Solaris ゾーンの紹介』を参照してください。

非大域ゾーン用アプリケーションの作成時の一般的な考慮事項

開発者による調査が必要であることがわかっているのは、次の場合です。

- システム時間を変更するシステムコールは、PRIV_SYS_TIME 特権を必要とします。このようなシステムコールには、`adjtime(2)`、`ntp_adjtime(2)`、`stime(2)` があります。
- スティッキビットが設定されているファイルを操作する必要のあるシステムコールは、PRIV_SYS_CONFIG 特権を必要とします。このようなシステムコールには、`chmod(2)`、`creat(2)`、`open(2)` があります。

- ioctl(2) システムコールは、STREAMS モジュールのアンカーをロック解除するために、PRIV_SYS_NET_CONFIG 特権を必要とします。
- link(2) および unlink(2) システムコールは、非大域ゾーン内のディレクトリのリンクを作成またはリンクを解除するために、PRIV_SYS_LINKDIR 特権を必要とします。ソフトウェアをインストールまたは構成するアプリケーションや、一時ディレクトリを作成するアプリケーションは、この制限の影響を受ける可能性があります。
- PRIV_PROC_LOCK_MEMORY 特権は、mlock(3C)、munlock(3C)、mlockall(3C)、munlockall(3C)、および plock(3C) 関数と、memcntl(2) システムの MC_LOCK、MC_LOCKAS、MC_UNLOCK、および MC_UNLOCKAS フラグに必要です。この特権は非大域ゾーンのデフォルト特権です。詳細は、『Oracle Solaris ゾーンの作成と使用』の「非大域ゾーン内の特権」を参照してください。
- mknod(2) システムコールは、ブロック型 (S_IFBLK) または文字型 (S_IFCHAR) 特殊ファイルを作成するために、PRIV_SYS_DEVICES 特権を必要とします。この制限は、デバイスノードをオンザフライで作成する必要のあるアプリケーションに影響します。
- msgctl(2) システムコールの IPC_SET フラグは、メッセージキューのバイト数を増やすために、PRIV_SYS_IPC_CONFIG 特権を必要とします。この制限は、メッセージキューのサイズを動的に変更する必要のあるアプリケーションに影響します。
- nice(2) システムコールは、プロセスの優先順位を変更するために、PRIV_PROC_PRIOCNTL 特権を必要とします。この特権は非大域ゾーンでデフォルトで利用可能です。優先順位を変更するもう 1 つの方法は、アプリケーションが実行されている非大域ゾーンをリソースプールにバインドすることです。ただし、そのゾーンのスケジューリングプロセスは、最終的に公平配分スケジューラによって決定されます。
- p_online(2) システムコールの P_ONLINE、P_OFFLINE、P_NOINTR、P_FAULTED、P_SPARE、および PZ-FORCED フラグは、プロセスの動作ステータスを返すまたは変更するために、PRIV_SYS_RES_CONFIG 特権を必要とします。この制限は、CPU を有効または無効にする必要のあるアプリケーションに影響します。
- priocntl(2) システムコールの PC_SETPARMS および PC_SETXPARMS フラグは、軽量プロセス (LWP) のスケジューリングパラメータを変更するために、PRIV_PROC_PRIOCNTL 特権を必要とします。
- プロセッサセット (psets) への LWP のバインドや、pset 属性の設定など、psets の管理を行う必要のあるシステムコールは、PRIV_SYS_RES_CONFIG 特権を必要とします。この制限は、次のシステムコールに影響します: pset_assign(2)、pset_bind(2)、pset_create(2)、pset_destroy(2)、および pset_setattr(2)。
- shmctl(2) システムコールの SHM_LOCK および SHM_UNLOCK フラグは、メモリー制御操作を共有するために、PRIV_PROC_LOCK_MEMORY 特権を必要とします

す。アプリケーションがパフォーマンスのためにメモリーをロックしている場合は、回避策として Intimate Shared Memory (ISM) 機能を使用できます。

- swapctl(2) システムコールは、スワップリソースを追加または削除するために、PRIV_SYS_CONFIG 特権を必要とします。この制限は、インストールおよび構成ソフトウェアに影響します。
- uadmin(2) システムコールは、A_REMOUNT、A_FREEZE、A_DUMP、および AD_IBOOT コマンドを使用するために、PRIV_SYS_CONFIG 特権を必要とします。この制限は、特定の状況で強制的にクラッシュダンプを実行する必要のあるアプリケーションに影響します。
- clock-settime(3C) 関数は、CLOCK_REALTIME および CLOCK_HIRES クロックを設定するために、PRIV_SYS_TIME 特権を必要とします。
- cpc_bind_cpu(3CPC) 関数は、要求セットをハードウェアカウンタにバインドするために、PRIV_CPC_CPU 特権を必要とします。回避策として、cpc_bind_curlwp(3CPC) 関数を使用して、問題の LWP の CPU カウンタをモニターできます。
- pthread_attr_setschedparam(3C) 関数は、スレッドの基盤のスケジューリングポリシーおよびパラメータを変更するために、PRIV_PROC_PRIOCNTL 特権を必要とします。
- timer_create(3C) 関数は、高精度のシステムクロックを使用してタイマーを作成するために、PRIV_PROC_CLOCK_HIGHRES 特権を必要とします。
- 次のライブラリ一覧で提供されている API は、非大域ゾーンではサポートされません。共有オブジェクトはゾーンの /usr/lib ディレクトリに存在するため、コードにこれらのライブラリへの参照が含まれている場合、リンク時エラーは発生しません。make ファイルを調べると、アプリケーションがこれらのライブラリのいずれかに明示的なバインドを持っているかどうかを判定でき、アプリケーションの実行中に pmap(1) を使用すると、これらのライブラリのいずれも動的にロードされないことを確認できます。
  - libdevinfo(3LIB)
  - libcfgadm(3LIB)
  - libpool(3LIB)
  - libsysevent(3LIB)
- ゾーンは、Oracle Solaris のプログラミング API の一部である疑似デバイスで主に構成される、デバイスの限定的なセットを持ちます。このような擬似デバイスには、/dev/null、/dev/zero、/dev/poll、/dev/random、/dev/tcp などがあります。物理デバイスには、そのデバイスがシステム管理者によって構成されている場合を除き、ゾーン内から直接アクセスすることはできません。一般に、デバイスはシステム内の共有リソースなので、システムのセキュリティーを損なわずにデバイスをゾーン内で使用可能にするには、次のようないくつかの制限が必要です。
  - /dev 名前空間は、/devices 内の物理パスへのシンボリックリンク (論理パス) から成ります。/devices 名前空間 (大域ゾーンでのみ使用可能) は、ドライバによって作成された接続されたデバイスインスタンスの現在の状態を反映します。非大域ゾーンから見えるのは論理パス /dev だけです。

- 非大域ゾーン内のプロセスは新しいデバイスノードを作成できません。たとえば、mknod(2) は非大域ゾーンに特殊ファイルを作成できません。creat(2)、link(2)、mkdir(2)、rename(2)、symlink(2)、および unlink(2) システムコールは、/dev 内のファイルが指定されると EACCES で失敗します。/dev 内のエントリへのシンボリックリンクを作成することはできますが、/dev 内には作成できません。
- システムのデータを公開するデバイスは、大域ゾーンでのみ利用できます。そのようなデバイスの例として、dtrace(7D)、kmem(7D)、kmdb(7d)、ksyms(7D)、lockstat(7D)、trapstat(1M) があります。
- /dev 名前空間は、デフォルトの「安全な」ドライバセットから成るデバイスノードと、zonecfg(1M) コマンドによってゾーンに指定されているデバイスノードから成ります。

Oracle Solaris 10 の非大域ゾーンに固有の考慮事項

LIFC_UNDER_IPMP は Oracle Solaris 10 では利用できないため、このリリースでサポートされているアプリケーションは LIFC_UNDER_IPMP を使用しません。したがって、Oracle Solaris 10 ゾーン内で SIOCGLIFCONF 要求を発行するアプリケーションには、基盤のインタフェースは見えず、代わりに IPMP メタインタフェースだけが見えます。solaris10 ゾーンでは Oracle Solaris 11 バージョンの ifconfig が使用され、これは特殊な LIFC_UNDER_IPMP を渡すため、ifconfig コマンドを -a オプション付きで使用すると基盤のインタフェースが表示されます。詳細は、if_tcp(7P) を参照してください。

共有 IP 非大域ゾーンに固有の考慮事項

共有 IP インスタンスを使用するように構成されている非大域ゾーンの場合、次の制限が適用されます。

- socket(3SOCKET) 関数は、プロトコルを IPPROTO_RAW または IPPROTO_IGMP に設定して raw ソケットを作成するために、PRIV_NET_RAWACCESS 特権を必要とします。この制限は、raw ソケットを使用するアプリケーション、あるいは TCP/IP ヘッダーを作成または検査する必要のあるアプリケーションに影響します。
- t_open(3NSL) 関数は、トランスポートのエンドポイントを確立するために、PRIV_NET_RAWACCESS 特権を必要とします。この制限は、/dev/rawip デバイスを使用してネットワークプロトコルを実装するアプリケーション、および TCP/IP ヘッダーを操作するアプリケーションに影響します。
- DLPI プログラミングインタフェースをサポートする NIC デバイスには、共有 IP 非大域ゾーンではアクセスできません。

- 共有 IP 非大域ゾーンは、それぞれ独自の論理ネットワークおよびループバックインタフェースを持っています。上位層ストリームと論理インタフェースの間のバインドは制限され、ストリームは同じゾーン内の論理インタフェースに対してのみバインドを確立できます。同様に、論理インタフェースからのパケットを渡すことができるのは、論理インタフェースと同じゾーン内の上位層ストリームに対してだけです。ループバックアドレスへのバインドはゾーン内に保持されますが、1 つ例外があります。それは、1 つのゾーン内のストリームが別のゾーン内のインタフェースの IP アドレスにアクセスしようとする場合です。ゾーン内のアプリケーションは、特権ネットワークポートにバインドすることはできますが、IP アドレスやルーティングテーブルなどのネットワーク構成を制御することはできません。

ただし、これらの制限は排他的 IP ゾーンには適用されません。

solaris ゾーンでのパッケージングに関する考慮事項

ゾーンパッケージバリアントを使用すると、パッケージ内の各種のコンポーネントが、大域ゾーン (global) または非大域ゾーン (nonglobal) のいずれかのみにインストールされるように明確にタグ付けされます。指定されるパッケージには、非大域ゾーンにインストールされないようにタグ付けされたファイルを含めることができます。

ゾーンモニタリング統計用 API

libzonestat.so.1 は、ゾーン関連のリソース使用効率の情報を取得および計算するために zonestat コマンドで使用される公開 API であり、ソートおよびフィルタリングのオプションを備えています。zonestat ライブラリは、システム全体およびゾーン別の物理メモリー、仮想メモリー、および CPU リソースの使用効率を報告します。zonestat コマンドは zonestat(1) のマニュアルページに記載されています。

libzonestat は、2 つのサンプル間の差や、使用されている量の割合など、一般に必要とされる値を計算します。これらの統計により、コンシューマは複雑な計算を実行する必要がなくなります。物理リソースの使用率に加えて、libzonestat は、各ゾーンの構成済みリソース制限に対するリソース使用率も報告します。

libzonestat の基本的な使用方法は次のとおりです。

```
zs_ctl_t zsctl;
     zs_usage_t usage;

     /* open the statistics facility */
     zsctl = zs_open();

     for (;;) {
       /* read the current usage */
```

```
usage = zs_usage_read(ctl);

... Interrogate the usage object for desired information ...
...

if (quit)
 break;

sleep(some_interval);
   }
   zs_close(zsctl);
```

ライブラリインタフェースは次のマニュアルページで説明されています (使いやすい順序で並べてあります)。

- zs_open(3ZONESTAT)
- zs_usage(3ZONESTAT)
- libzonestat(3LIB)
- zs_resource(3ZONESTAT)
- zs_zone(3ZONESTAT)
- zs_pset(3ZONESTAT)
- zs_pset_zone(3ZONESTAT)
- zs_property(3ZONESTAT)

ゾーンのファイルシステムアクティビティーのモニタリング

fsstat ユーティリティーを使用して、非大域ゾーンのファイル操作の統計情報を報告できます。

大域ゾーンでは、システム上のすべてのゾーンの kstat を表示できます。非大域ゾーンでは、ユーティリティーが実行されているゾーンに関連付けられた kstats のみが表示されます。非大域ゾーンでは、ほかのゾーンのファイルシステムアクティビティーをモニターできません。

-z オプションは、ゾーン別のファイルシステムアクティビティーを報告するときに使用します。複数の -z オプションを使用して、選択したゾーンのアクティビティーをモニターできます。このオプションは、fstypes ではなく mountpoints のみをモニターする場合に使用しても効果はありません。

-Z オプションは、システム上のすべてのゾーンのファイルシステムアクティビティーを報告するときに使用します。このオプションは、-z オプションとともに使用しても効果はありません。このオプションは、fstypes ではなく mountpoints のみをモニターする場合に使用しても効果はありません。

-A オプションは、指定した fstypes のファイルシステムアクティビティーのすべてのゾーンにわたる集計を報告するときに使用します。これは、-z または -Z オプションが

どちらも使用されなかった場合のデフォルトの動作です。-A オプションは、fstypes ではなく mountpoints のみをモニターする場合に使用しても効果はありません。

-A オプションを -z または -Z オプションとともに使用すると、指定した fstypes のすべてのゾーンにわたる集計が別個の行に表示されます。このオプションは、fstypes ではなく mountpoints のみをモニターする場合に使用しても効果はありません。

詳細は、fsstat(1M) のマニュアルページを参照してください。

Oracle Solaris 10 ゾーン

Oracle™ Solaris 10 ゾーンは、Oracle Solaris 11 カーネルで動作する x86 および SPARC Solaris 10 9/10 (またはそれ以降にリリースされた Oracle Solaris 10 アップデート) ユーザー環境をホストする solaris10 ブランドゾーンです。カーネルパッチの 142909-17 (SPARC) または 142910-17 (x86/x64)、あるいはそれらよりも後のバージョンを最初に元のシステムにインストールしている場合は、10/09 よりも前の Oracle Solaris 10 リリースを使用できます。

solaris10 ブランドゾーンの詳細は、次のガイドを参照してください。

- 『Oracle Solaris 10 ゾーンの作成と使用』。
- 『System Administration Guide: Oracle Solaris Containers-Resource Management and Oracle Solaris Zones』 (これは、このガイドの Oracle Solaris 10 バージョンです)。
- 『Solaris Containers: Resource Management and Solaris Zones Developer's Guide』 (これは、このガイドの Oracle Solaris 10 バージョンです)。

solaris10 ゾーンおよびネイティブゾーンで使用される SVR4 パッケージングおよびパッチについては、『System Administration Guide: Oracle Solaris Containers-Resource Management and Oracle Solaris Zones』の第 25 章「ゾーンがインストールされている Solaris システムでのパッケージについて (概要)」および第 26 章「ゾーンがインストールされている Solaris システムでのパッケージとパッチの追加および削除 (タスク)」を参照してください。これは、このガイドの Oracle Solaris 10 バージョンです。

Oracle Solaris カーネルゾーン

Oracle Solaris カーネルゾーン機能は、ゾーン内の完全なカーネルおよびユーザー環境を提供します。さらに、カーネルゾーンはホストシステムとゾーンの間でカーネルの分離を強化します。ブランド名は solaris-kz です。カーネルゾーンは既存のツールを使用して管理されます。大域ゾーンにインストールされているパッケージに制限されることなく、カーネルバージョンを含め、ゾーンのインストール済みパッケージを完全に更新および変更できます。カーネルゾーン内で solaris ゾーンを実行し

て、階層 (ネストされた) ゾーンを生成できます。カーネルゾーンでは一時停止および再開がサポートされます。Oracle Solaris カーネルゾーンは、SRU (Support Repository Update)、またはホストシステムのものとは異なるカーネルバージョンを実行できます。

Oracle Solaris カーネルゾーンの詳細は、『Oracle Solaris カーネルゾーンの作成と使用』および『Oracle Solaris ゾーン構成リソース』 の 第 1 章,「非大域ゾーンの構成」を参照してください。

♦ ♦ ♦ 第 8 章

8 構成例

この章では、/etc/project ファイルの構成例を示します。

zonecfg コマンドを使用してゾーンにリソース制御を設定するには、『Oracle Solaris ゾーン構成リソース』 の 第 1 章, 「非大域ゾーンの構成」および『Oracle Solaris ゾーンの作成と使用』 の 「ゾーンの構成方法」を参照してください。

/etc/project プロジェクトファイル

project ファイルはプロジェクト情報のローカルソースです。このプロジェクトファイルは、NIS マップの project.byname と project.bynumber、および LDAP データベースプロジェクトを含むその他のプロジェクトソースと組み合わせて使用できます。プログラムは getprojent(3PROJECT) ルーチンを使用してこの情報にアクセスします。

2 つのプロジェクトを定義する

/etc/project は、database と appserver という 2 つのプロジェクトを定義します。*user* のデフォルトは user.database および user.appserver です。*admin* のデフォルトは user.database と user.appserver の間で切り替えることができます。

```
hostname# cat /etc/project

.
.
.
user.database:2001:Database backend:admin::
user.appserver:2002:Application Server frontend:admin::
.
```

.

リソース制御を構成する

/etc/project ファイルには、アプリケーションのリソース制御が表示されます。

```
hostname# cat /etc/project

.
.
.
development:2003:Developers:::task.ax-lwps=(privileged,10,deny);
process.max-addressspace=(privileged,209715200,deny)
.
.
```

リソースプールを構成する

/etc/project ファイルには、アプリケーションのリソースプールが表示されます。

```
hostname# cat /etc/project

.
.
.
batch:2001:Batch project:::project.pool=batch_pool
process:2002:Process control:::project.pool=process_pool
.
.
.
```

プロジェクトの FSS project.cpu-shares を構成する

2 つのプロジェクト *database* と *appserver* の FSS を設定します。*database* プロジェクトには CPU 配分を 20 割り当てます。*appserver* プロジェクトには CPU 配分を 10 割り当てます。

```
hostname# cat /etc/project

.
.
.
user.database:2001:database backend:admin::project.cpu-shares=(privileged,
     20,deny)
user.appserver:2002:Application Server frontend:admin::project.cpu-shares=
     (privileged,10,deny)
.
.
.
```

注記 - /etc/project ファイル内の「20,deny」および「(privileged,」の前にある行内の行ブレークは無効です。行ブレークは、印刷されるページまたは表示されるページ上に例を表示するためだけにここに示されています。/etc/project ファイル内の各エントリは 1 行で記述する必要があります。

FSS は有効になっていて、各ユーザーおよびアプリケーションが一意のプロジェクトに割り当てられていない場合、そのユーザーおよびアプリケーションはすべて同じプロジェクトで実行されます。同じプロジェクトで実行されると、すべてが同じ配分をめぐって時分割方式で競合します。これは、配分がユーザーやプロセスではなくプロジェクトに割り当てられているために起こります。FSS のスケジューリング機能を活用するには、各ユーザーおよびアプリケーションを一意のプロジェクトに割り当てます。

プロジェクトを構成するには、『Oracle Solaris 11.3 でのリソースの管理』 の 「ローカルの /etc/project ファイルの形式」を参照してください。

特性の異なる 5 つのアプリケーションを構成する

次の例では、特性の異なる 5 つのアプリケーションを構成します。

表 8 ターゲットアプリケーションと特性

アプリケーションのタイプと名前	特性
アプリケーションサーバー、app_server。	2 CPU を超えるとスケーラビリティーが低くなります。2 CPU プロセッサセットを app_server に割り当てます。TS スケジューリングクラスを使用します。
データベースインスタンス、app_db。	極度にマルチスレッド化されます。FSS スケジューリングクラスを使用します。
テストおよび開発、development。	モチーフベースです。テストされていないコードの実行をホストします。対話型スケジューリングクラスによってユーザーインタフェースの応答性が保証されます。メモリー制限をかけてアンチソーシャル処理の効果を最小限に抑えるために process.max-address-space を使用します。
トランザクション処理エンジン、tp_engine。	応答時間が最優先されます。応答待機時間を最小限に維持するために、少なくとも 2 つの CPU から成る専用セットを割り当てます。時分割スケジューリングクラスを使用します。
スタンドアロンのデータベースインスタンス、geo_db。	極度にマルチスレッド化されます。複数のタイムゾーンに対応します。FSS スケジューリングクラスを使用します。

注記 - データベースアプリケーション (app.db および geo_db) を少なくとも 4 つの CPU から成る単一のプロセッサセットに統合します。FSS スケジューリングクラスを使用します。アプリケーション app_db は project.cpu-shares の 25% を取得します。アプリケーション geo_db は project.cpu-shares の 75% を取得します。

/etc/project ファイルを編集します。app_server、app_db、development、tp_engine、および geo_db プロジェクトエントリのリソースプールにユーザーをマッピングします。

```
hostname# cat /etc/project

.
.
.
user.app_server:2001:Production Application Server::
     project.pool=appserver_pool
user.app_db:2002:App Server DB:::project.pool=db_pool,
     project.cpu-shares=(privileged,1,deny)
development:2003:Test and delopment::staff:project.pool=dev.pool,
     process.max-addressspace=(privileged,536870912,deny)
user.tp_engine:Transaction Engine:::project.pool=tp_pool
user.geo_db:EDI DB:::project.pool=db_pool;
     project.cpu-shares=(privileged,3,deny)
```

注記 - 「project.pool」、「project.cpu-shares=」、「process.max-addressspace」、および「project.cpu-shares=」で始まる行の行ブレークは、プロジェクトファイル内では無効です。行ブレークは、印刷されるページまたは表示されるページ上に例を表示するためだけにここに示されています。各エントリは 1 行で記述する必要があります。

pool.host スクリプトを作成してリソースプールのエントリを追加します。

```
hostname# cat pool.host

create system host
create pset dev_pset (uint pset.max = 2)
create pset tp_pset (uint pset.min = 2; uint pset.max = 2)
create pset db_pset (uint pset.min = 4; uint pset.max = 6)
create pset app_pset (uint pset.min = 1; uint pset.max = 2)
create pool dev_pool (string pool.scheduler="IA")
create pool appserver_pool (string pool.scheduler="TS")
create pool db_pool (string pool.scheduler="FSS")
create pool tp_pool (string pool.scheduler="TS")
associate pool pool_default (pset pset_default)
associate pool dev_pool (pset dev_pset)
associate pool appserver_pool (pset app_pset)
associate pool db_pool (pset db_pset)
associate pool tp_pool (pset tp_pset)
```

pool.host スクリプトを実行して、pool.host ファイルで指定されているとおりに構成を修正します。

```
hostname# poolcfg -f pool.host
```

pool.host リソースプールの構成ファイルを読み取り、システム上のリソースプールを初期化します。

```
hostname# pooladm -c
```

索引

F

L

O

S

Z