Red Hat Enterprise Linux 6

リソース管理ガイド

Red Hat Enterprise Linux 6 のシステムリソース管理

エディッション 4

Last Updated: 2017-10-12

Red Hat Enterprise Linux 6 リソース管理ガイド

Red Hat Enterprise Linux 6 のシステムリソース管理
エディッション 4

Martin Prpič
Red Hat Engineering Content Services
mprpic@redhat.com

Rüdiger Landmann
Red Hat Engineering Content Services
r.landmann@redhat.com

Douglas Silas
Red Hat Engineering Content Services
dhensley@redhat.com

法律上の通知

概要

Red Hat Enterprise Linux 6 におけるシステムリソースの管理

目次

第1章 コントロールグループについて (CGROUP)

Red Hat Enterprise Linux 6 では、*コントロールグループ*と呼ばれる新たなカーネル機能を搭載しています。本ガイドでは、この機能を *cgroup* という略称で記載しています。cgroup により、ユーザーは、CPU 時間、システムメモリー、ネットワーク帯域幅などのリソースやそれらのリソースの組み合わせを、システム上で実行中のユーザー定義タスクグループ (プロセス) の間で割り当てることができるようになります。また、設定した cgroup のモニタリングを行ったり、特定のリソースに対する cgroup のアクセスを拒否することができるのに加えて、稼働中のシステムで cgroup を動的に再設定することもできます。**cgconfig (control group config)** サービスがブート時に起動し、事前に定義された cgroup を再構築するように設定して、再起動後も永続されるようにすることが可能です。

cgroup を使用することにより、システム管理者は、システムリソースの割り当て、優先度設定、拒否、管理、モニタリングに対する粒度の細かいコントロールが可能となります。ハードウェアリソースは、タスクおよびユーザー間で素早く分配され、全体的な効率が向上します。

1.1. コントロールグループの構成

プロセスと同様に、cgroup は階層的に構成されており、子 cgroup は、親 cgroup の属性の一部を継承するようになっています。ただし、これら 2 つのモデルの間には相異点があります。

Linux プロセスモデル

Linux システム上のプロセスはすべて、**init** プロセスという、共通の親プロセスの子プロセスです。**init** プロセスは、ブート時にカーネルによって実行され、その他のプロセスを開始します (その結果、その他のプロセスがそれら独自の子プロセスを開始する場合もあります)。プロセスはすべて、単一の親プロセスの下位プロセスであるため、Linux プロセスモデルは、単一の階層またはツリーとなっています。

また、**init** を除いた Linux プロセスはすべて、環境 (例: PATH 変数)[1] および親プロセスのその他特定の属性 (例: オープンファイル記述子) を継承します。

cgroup モデル

cgroup は、以下のような点でプロセスと類似しています:

- 階層型である
- 子 cgroup は、親 cgroup から特定の属性を継承する

根本的な相違点は、cgroup の場合には、多数の異なる階層がシステム上に同時に存在可能であることです。Linux プロセスモデルが単一のプロセスツリーとすれば、cgroup モデルは、単一もしくは複数の分離した、連結されていないタスクツリー (すなわちプロセス) ということになります。

cgroups の複数の分離した階層が必要なのは、各階層が **単一または複数の** *サブシステム*に接続されていることが理由です。サブシステム[2] とは、CPU 時間やメモリーなどの単一のリソースを指します。Red Hat Enterprise Linux 6 は、10 の cgroup サブシステムを提供しています。それらの名前と機能は、以下のとおりです。

Red Hat Enterprise Linux で利用可能なサブシステム

- **blkio** – このサブシステムは、物理ドライブ (例: ディスク、ソリッドステート、USB) などのブロックデバイスの入出力アクセスの制限を設定します。
- **cpu** – このサブシステムは、スケジューラーを使用して cgroup タスクに CPU へのアクセスを提供します。

- **cpuacct** – このサブシステムは、cgroup 内のタスクで使用される CPU リソースについての自動レポートを生成します。
- **cpuset** – このサブシステムは、個別の CPU (マルチコアシステム上) およびメモリーノードを cgroup 内のタスクに割り当てます。
- **devices** – このサブシステムは、cgroup 内のタスクによるデバイスへのアクセスを許可または拒否します。
- **freezer** – このサブシステムは、cgroup 内のタスクを一時停止または再開します。
- **memory** – このサブシステムは、cgroup 内のタスクによって使用されるメモリーに対する制限を設定し、それらのタスクによって使用されるメモリーリソースについての自動レポートを生成します。
- **net_cls** – このサブシステムは、Linux トラフィックコントローラー (tc) が特定の cgroup から発信されるパケットを識別できるようにするクラス識別子 (classid) を使用して、ネットワークパケットにタグを付けます。
- **net_prio** – このサブシステムは、ネットワークインターフェース別にネットワークトラフィックの優先度を動的に設定する方法を提供します。
- **ns** – *namespace* サブシステム。

注記

man ページやカーネルのドキュメントなど、cgroup に関連した資料で *リソースコントローラー* または単に *コントローラー* という用語が使用されている場合があります。これらの 2 つの用語は、「サブシステム」と同じ意味です。これは、サブシステムが通常リソースのスケジュールを行ったり、サブシステムが接続されている階層内の cgroups に対する制限を適用したりすることが理由で、このように呼ばれています。

サブシステム (リソースコントローラー) の定義は、極めて広義で、タスクグループ (すなわちプロセス) に作用するものとされています。

1.2. サブシステム、階層、コントロールグループ、タスクの関係

cgroup の用語においては、システムプロセスはタスクと呼ばれることを念頭に置いてください。

ではここで、サブシステム、cgroup の階層、およびタスクの間における関係を管理するにあたってのいくつかの簡単なルールとそれらのルールがもたらす影響について説明しましょう。

ルール 1
単一階層には、単一または複数のサブシステムを接続することができます。

このため、cpu および memory のサブシステム (もしくは任意数のサブシステム) を単一階層に接続できます。ただし、各サブシステムは、別のサブシステムがすでに接続された別の階層に接続されていないことが条件となります (ルール 2 を参照)。

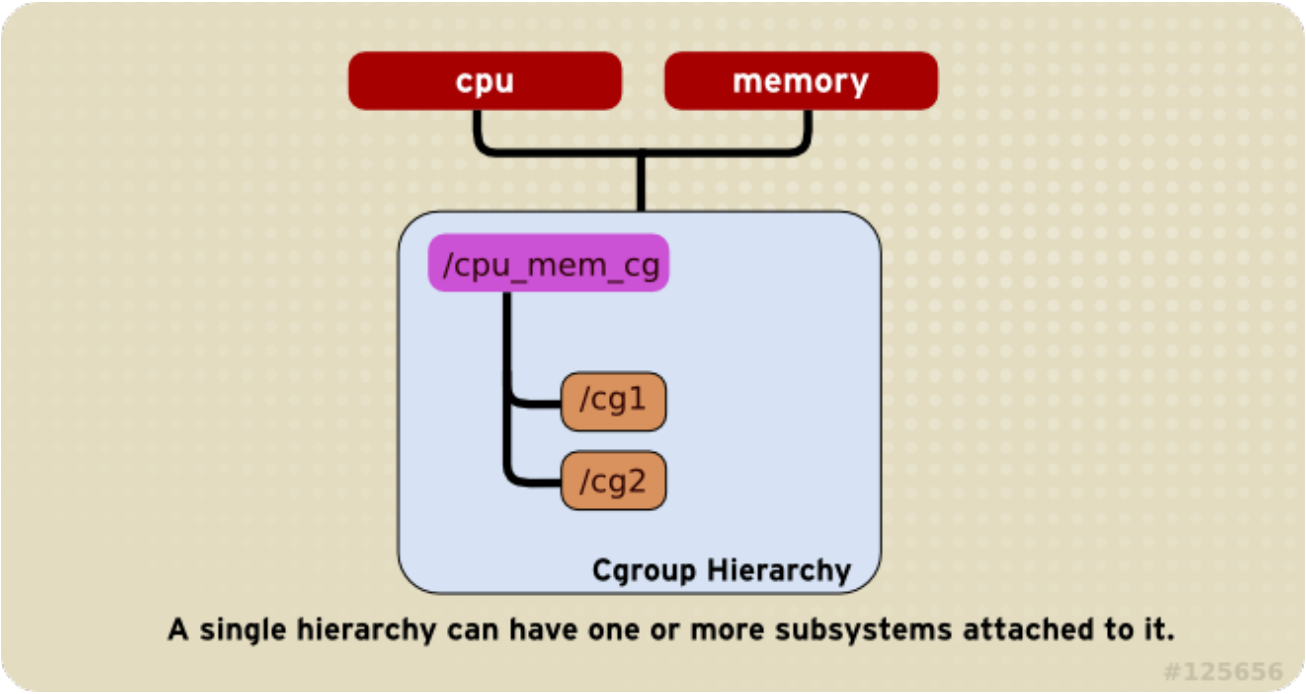


図1.1 ルール 1

ルール 2

別のサブシステムがすでに接続されている階層がある場合には、単一のサブシステム (例: **cpu**) を複数の階層に接続することはできません。

このため、`memory` サブシステムが接続された階層がある場合には、cpu サブシステムは 2 つの異なる階層には決して接続できませんが、両方の階層に接続されているのがそのサブシステムのみの場合には、単一のサブシステムを 2 つの階層に接続することが可能です。

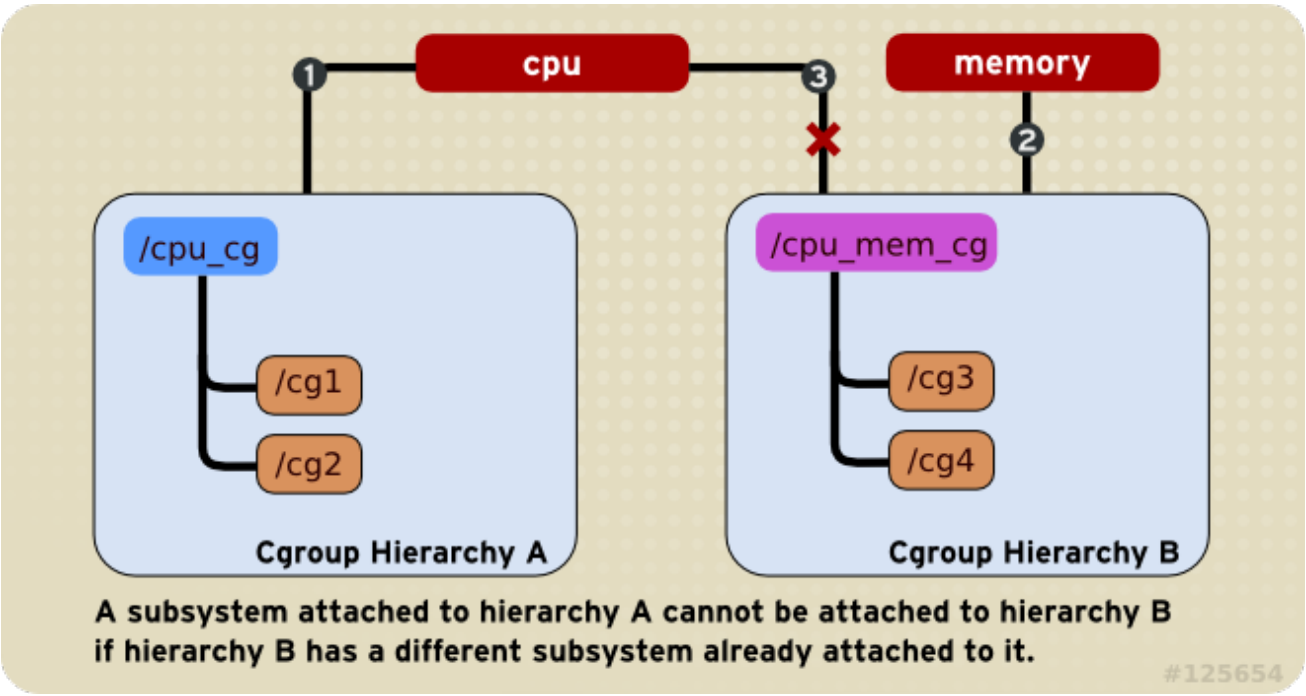


図1.2 ルール 2 (丸数字は、サブシステムが接続される時間的順序)

ルール 3

システムに新たな階層が作成されると、システム上のタスクはすべて、最初にその階層のデフォルトの cgroup のメンバーとなります。これは、*root cgroup* としても知られています。作成された単一階層で

はいずれも、システム上の各タスクをその階層内の **完全に一つの** cgroup のメンバーにすることができます。cgroup がそれぞれ異なる階層にあれば、単一のタスクを複数の cgroups のメンバーとすることが可能です。タスクが同じ階層内の第 2 の cgroup のメンバーとなると、タスクはその階層内の第 1 の cgroup から即時に削除されます。タスクが同じ階層内の異なる 2 つの cgroup のメンバーとなることはありません。

このため、cpu と memory サブシステムが cpu_mem_cg という名前の階層に接続されて、net_cls サブシステムが net という名前の階層に接続された場合、稼働中の httpd プロセスは、cpu_mem_cg 内の任意の単一 cgroup および net 内の任意の単一 cgroup のメンバーとすることができます。

httpd プロセスがメンバーとなっている cpu_mem_cg 内の cgroup によって、CPU 時間が、他のプロセスに割り当てられた時間の半分に制限され、メモリー使用量が最大で 1024 MB に限定される可能性があります。また、メンバーとなっている net 内の cgroup によって、転送速度が 30 メガバイト毎秒に制限される場合があります。

第 1 の階層が作成されると、そのシステム上の全タスクは、最低でも 1 つの cgroup (root cgroup) のメンバーとなります。このため、cgroup を使用すると、すべてのシステムタスクは常に、最低でも一つの cgroup のメンバーとなります。

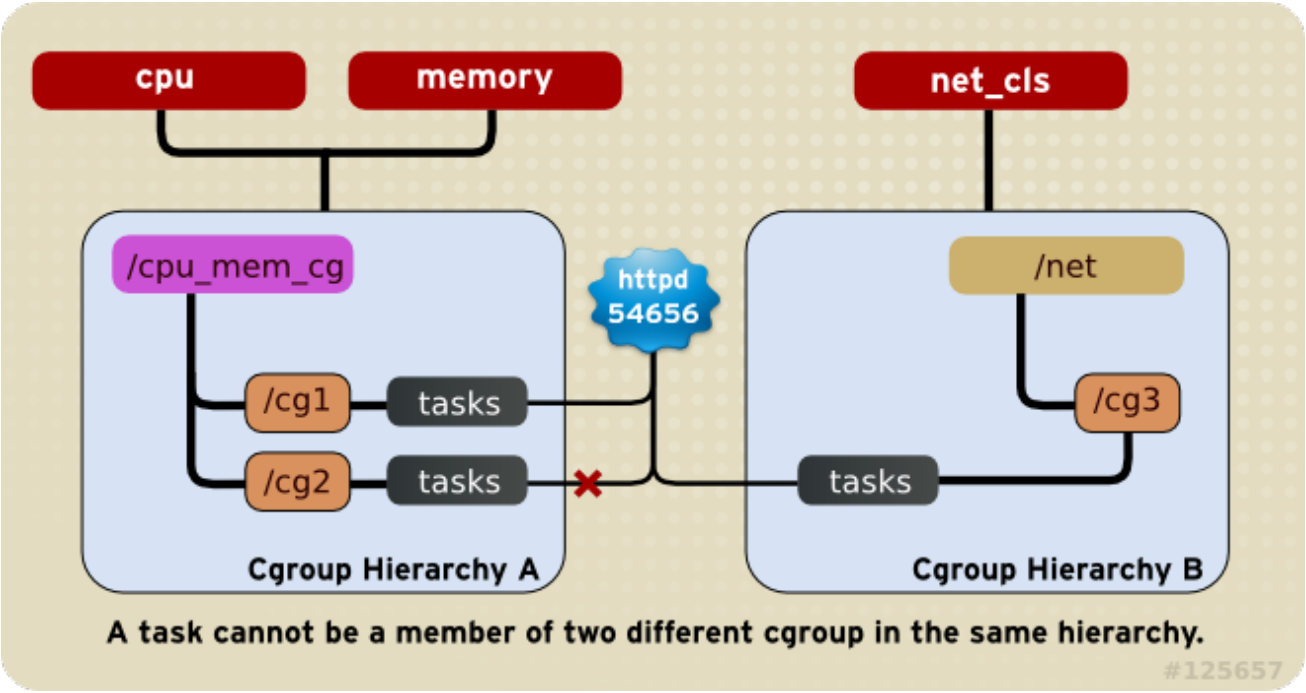


図1.3 ルール 3

ルール 4

システム上のいずれかのプロセス (タスク) が自分自身をフォークすると、子プロセス (タスク) が作成されます。子タスクは、親の cgroup のメンバーシップを自動的に継承しますが、必要に応じて異なる cgroup に移動することが可能です。フォークされた後には、親プロセスと子プロセスは完全に独立します。

その結果として、cpu_and_mem 階層内の half_cpu_1gb_max という名前の cgroup と、net 階層内の trans_rate_30 という cgroup のメンバーである httpd タスクについて検討してください。 この httpd プロセスが自分自身をフォークすると、その子プロセスは自動的にhalf_cpu_1gb_max cgroup と trans_rate_30 cgroup のメンバーとなり、親タスクが属するのとまったく同じ cgroup を継承します。

それ以降には、親タスクと子タスクは完全に相互に独立した状態となり、一方のタスクが属する cgroup を変更しても他方のタスクには影響を及ぼしません。また、親タスクの cgroup を変更しても、孫タスクへも一切影響はありません。つまり、子タスクはいずれも、最初は親タスクとまったく同

じ cgroup へのメンバーシップを継承しますが、これらのメンバーシップは後で変更もしくは削除することができるということになります。

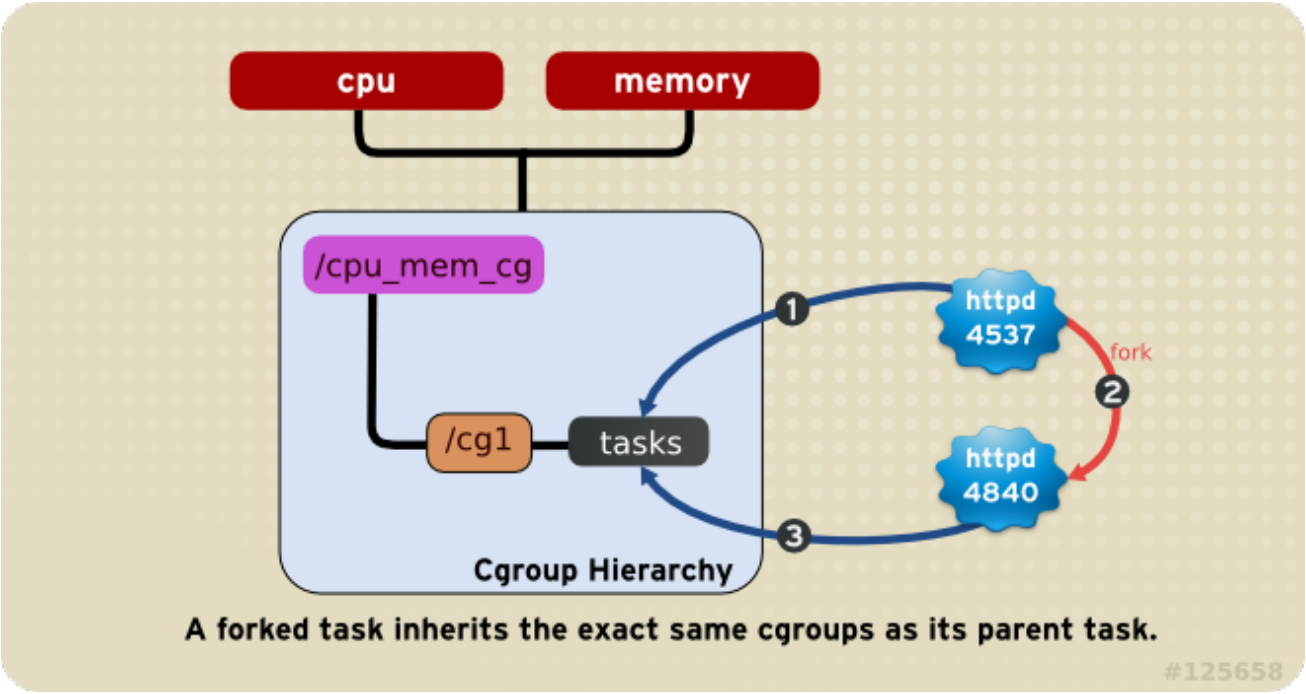


図1.4 ルール 4 (丸数字は、タスクがフォークする時間的順序)

1.3. リソース管理に対する影響

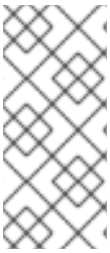
- タスクは、単一の階層内では一つの cgroup のみにしか属すことができないため、単一サブシステムによってタスクが制限されたり、影響を受ける状況は一つのみということになります。これは、制限ではなく特長であり、論理にかなっています。
- 単一階層内の全タスクに影響を及ぼすように、複数のサブシステムをまとめてグループ化することができます。その階層内の cgroup には異なるパラメーターが設定されているため、これらのタスクが受ける影響が異なってきます。
- 場合によっては、階層の *リファクタリング* が必要となることがあります。たとえば、複数のサブシステムが接続された階層からサブシステムを削除したり、新たな別の階層に接続する場合などです。
- 逆に言えば、別個の階層間でサブシステムを分割する必要性が低減した場合には、階層を削除して、そのサブシステムを既存の階層に接続することができるということになります。
- この設計により、**cpu** と **memory** のサブシステムのみが接続されているような単一階層内の特定のタスクに対して、いくつかのパラメーターを設定するなど、cgroup の簡易な使用方法が可能となります。
- また、この設計により、高度に特化した構成も可能となり、システム上の各タスク (プロセス) は、単一のサブシステムが接続された各階層のメンバーとすることができます。このような構成では、システム管理者があらゆるタスクの全パラメーターを全面的に管理することができます。

[1] 親プロセスは、子プロセスに環境を受け渡す前にその環境を変更することが可能です。

[2] libcgroup の man ページおよびその他のドキュメントでは、サブシステムが*リソースコントローラー*もしくは単に *コントローラー*とも呼ばれることを認識しておく必要があります。

第2章 コントロールグループの使用法

cgroup を使用して作業する最も簡単な方法は、libcgroup パッケージをインストールすることです。このパッケージには、数多くの cgroup 関連のコマンドラインユーティリティとそれらに関する man ページが含まれています。任意のシステム上で利用可能なシェルコマンドとユーティリティを使用して、階層を マウントし、cgroup のパラメーターを (非永続的に) 設定することも可能ですが、libcgroup の提供するユーティリティを使用すると、プロセスが簡素化され、機能性が拡張されます。このため、本ガイドでは全体を通して、libcgroup コマンドに重点を置いています。ほとんどの場合、相当するシェルコマンドを記載し、根本的な構造がわかりやすいようにしていますが、差し支えがなければ、libcgroup コマンドを使用することを推奨します。



注記

cgroup を使用するためには、まず最初に、root として以下のコマンドを実行し、libcgroup パッケージがインストールされているかどうかを確認します。

```
~]# yum install libcgroup
```

2.1. CGCONFIG サービス

libcgroup パッケージとともにインストールされる**cgconfig** サービスは、階層を作成して、サブシステムを階層に接続し、それらの階層内の cgroup を管理するのに便利な方法を提供します。システム上の階層と cgroup の管理には、**cgconfig** を使用することを推奨します。

Red Hat Enterprise Linux 6 では、**cgconfig** サービスはデフォルトで起動しません。**chkconfig** を使用してサービスを起動すると、システムは cgroup の設定ファイル **/etc/cgconfig.conf** を読み取ります。そのため、cgroup はセッションからセッションへと再作成されて永続化します。**cgconfig** は、設定ファイルの内容に応じて、階層の作成、必要なファイルシステムのマウント、cgroup の作成、各グループ用のサブシステムパラメーターの設定を行います。

libcgroup パッケージとともにインストールされるデフォルトの**/etc/cgconfig.conf** ファイルは、各サブシステム用の個別の階層を作成およびマウントし、それらの階層にサブシステムを接続します。

cgconfig サービスを停止 (**service cgconfig stop** コマンドを使用) すると、マウントされていたすべての階層がアンマウントされます。

2.1.1. /etc/cgconfig.conf ファイル

/etc/cgconfig.conf ファイルに含まれるエントリには、*mount* と *group* の 2 つの主要なタイプがあります。mount エントリは、仮想ファイルシステムとして階層を作成してマウントし、サブシステムをそれらの階層に接続します。mount エントリは、以下のような構文を用いて定義します。

```
mount {
    <controller> = <path>;
    …
}
```

使用例については、例2.1「mount エントリの作成」 を参照してください。

例2.1 mount エントリの作成

以下の例では、**cpuset** サブシステム用の階層を作成します。

```
mount {
    cpuset = /cgroup/red;
}
```

これに相当するシェルコマンドは以下のとおりです。

```
~]# mkdir /cgroup/red
~]# mount -t cgroup -o cpuset red /cgroup/red
```

group エントリは **cgroup** を作成して、サブシステムパラメーターを設定します。**group** エントリは、以下のような構文を使用して定義します。

```
group <name> {
    [<permissions>]
    <controller> {
        <param name> = <param value>;
        …
    }
    …
}
```

permissions セクションはオプションである点に注意してください。グループエントリのパーミッションを定義するには、以下のような構文を使用します。

```
perm {
    task {
        uid = <task user>;
        gid = <task group>;
    }
    admin {
       uid = <admin name>;
       gid = <admin group>;
    }
}
```

使用例については、例2.2「group エントリの作成」 を参照してください。

例2.2 group エントリの作成

以下の例は、**sqladmin** グループ内のユーザーのパーミッションで SQL デーモン用の **cgroup** を作成し、タスクを **cgroup** と **root** ユーザーに追加してサブシステムのパラメーターを変更します。

```
group daemons/sql {
    perm {
        task {
            uid = root;
            gid = sqladmin;
        } admin {
            uid = root;
            gid = root;
        }
    } cpu {
        cpuset.mems = 0;
```

```
        cpuset.cpus = 0;
    }
}
```

例2.1「mount エントリの作成」の mount エントリの例と組み合わせた場合に、相当するシェルコマンドは以下のとおりです。

```
~]# mkdir -p /cgroup/red/daemons/sql
~]# chown root:root /cgroup/red/daemons/sql/*
~]# chown root:sqladmin /cgroup/red/daemons/sql/tasks
~]# echo 0 > /cgroup/red/daemons/sql/cpuset.mems
~]# echo 0 > /cgroup/red/daemons/sql/cpuset.cpus
```

注記

/etc/cgconfig.conf への変更内容を有効にするには、**cgconfig** を再起動する必要があります。ただし、このサービスを再起動すると、全 cgroup 階層が再構築され、以前に存在していた cgroup (例: **libvirtd** によって使用される既存の cgroup) はいずれも削除される点に注意してください。**cgconfig** サービスを再起動するには、以下のコマンドを実行します。

```
~]# service cgconfig restart
```

libcgroup パッケージをインストールすると、サンプルの設定ファイルが**/etc/cgconfig.conf** に書き込まれます。行頭にハッシュ記号 ('#') が付いている場合、その行はコメントアウトされ、**cgconfig** サービスには不可視となります。

2.2. 階層の作成とサブシステムの接続

警告

以下に示した新規階層作成とサブシステム接続の手順は、ご使用のシステム上で cgroup がまだ設定されていないことを前提としています。その場合、これらの手順は、システム上の操作には影響を及ぼしません。しかし、タスクを持つ cgroup 内の調整可能なパラメーターを変更すると、それらのタスクに直ちに影響を及ぼす可能性があります。本ガイドでは、単一もしくは複数のタスクに影響を及ぼす可能性がある調整可能な cgroup のパラメーターの変更を初回に例示する際に警告します。

cgroup が設定済み (手動もしくは **cgconfig** サービスを使用して) のシステム上では、最初に既存の階層をアンマウントしておかないと、これらのコマンドは失敗し、システムの操作に影響を及ぼします。実稼働システム上ではこれらの手順を試さないようにしてください。

階層を作成して、サブシステムを接続するには、root として、**/etc/cgconfig.conf** ファイルの **mount** セクションを編集します。**mount** セクションのエントリは、以下のような形式になります。

```
subsystem = /cgroup/hierarchy;
```

cgconfig は次回の起動時に、階層を作成し、サブシステムを接続します。

以下の例は、**cpu_and_mem** という名前の階層を作成し、**cpu**、**cpuset**、**cpuacct**、および **memory** のサブシステムを接続します。

```
mount {
    cpuset  = /cgroup/cpu_and_mem;
    cpu     = /cgroup/cpu_and_mem;
    cpuacct = /cgroup/cpu_and_mem;
    memory  = /cgroup/cpu_and_mem;
}
```

その他の方法

シェルコマンドとユーティリティを使用して、階層の作成とサブシステムの接続を行うこともできます。

root として、その階層用の *マウントポイント* を作成します。マウントポイントには、cgroupの名前を入れます。

```
~]# mkdir /cgroup/name
```

例:

```
~]# mkdir /cgroup/cpu_and_mem
```

次に、**mount** コマンドを使用して、階層をマウントし、同時に一つまたは複数のサブシステムを接続します。以下はその例です。

```
~]# mount -t cgroup -o subsystems name /cgroup/name
```

ここで、*subsystems* は、コンマ区切りのサブシステム一覧で、また*name* は階層名です。利用可能な全サブシステムの簡単な説明は Red Hat Enterprise Linux で利用可能なサブシステム に記載しています。また 3章 *サブシステムと調整可能なパラメーター*には詳しい参考情報を記載しています。

例2.3 mount コマンドを使用したサブシステムの接続

この例では、**/cgroup/cpu_and_mem** という名前のディレクトリがすでに存在しており、作成する階層のマウントポイントとして機能します。**cpu**、**cpuset**、および **memory** のサブシステムを **cpu_and_mem** という名前の階層に接続し、**/cgroup/cpu_and_mem** 上の **cpu_and_mem** 階層に **mount** します。

```
~]# mount -t cgroup -o cpu,cpuset,memory cpu_and_mem /cgroup/cpu_and_mem
```

すべての利用可能なサブシステムとそれらの現在のマウントポイント (つまり、接続された階層がマウントされている場所) を一覧表示するには、**lssubsys** [3] コマンドを使用します。

```
~]# lssubsys -am
cpu,cpuset,memory /cgroup/cpu_and_mem
net_cls
ns
cpuacct
```

```
devices
freezer
blkio
```

この出力は、以下のような内容を示しています。

- **cpu**、**cpuset**、および **memory** のサブシステムは、**/cgroup/cpu_and_mem** にマウントされた階層に接続されています。
- **net_cls**、**ns**、**cpuacct**、**devices**、**freezer**、および **blkio** のサブシステムは、対応するマウントポイントがないことから、まだ、どの階層にも接続されていないことを示しています。

2.3. 既存の階層へのサブシステムの接続と接続解除

サブシステムを既存の階層に追加するには、既存の階層から接続を解除するか、異なる階層に移動した上で、root として **/etc/cgconfig.conf** ファイルの **mount** セクションを 「階層の作成とサブシステムの接続」 に記載されているのと同じ構文を用いて編集します。**cgconfig** は、次回の起動時に、指定した階層にしたがってサブシステムを再編成します。

その他の方法

接続されていないサブシステムを既存の階層に追加するには、その階層を再マウントします。**mount** コマンドで、追加するサブシステムと **remount** オプションを指定します。

例2.4 階層の再マウントとサブシステムの追加

lssubsys コマンドは、**cpu_and_mem** 階層に接続されている **cpu**、**cpuset**、および **memory** のサブシステムを表示します。

```
~]# lssubsys -am
cpu,cpuset,memory /cgroup/cpu_and_mem
net_cls
ns
cpuacct
devices
freezer
blkio
```

remount オプションを使用して **cpu_and_mem** 階層を再マウントし、サブシステムの一覧に **cpuacct** を追加します。

```
~]# mount -t cgroup -o remount,cpu,cpuset,cpuacct,memory cpu_and_mem
/cgroup/cpu_and_mem
```

lssubsys コマンドを実行すると、**cpuacct** が **cpu_and_mem** 階層に接続されていると表示されるようになりました。

```
~]# lssubsys -am
cpu,cpuacct,cpuset,memory /cgroup/cpu_and_mem
net_cls
ns
```

```
devices
freezer
blkio
```

また同様に、階層を再マウントしてサブシステム名を **-o** オプションから削除することで、既存の階層からサブシステムの接続を解除することができます。たとえば、**cpuacct** サブシステムの接続を解除するには、単に再マウントして名前を削除します。

```
~]# mount -t cgroup -o remount,cpu,cpuset,memory cpu_and_mem
/cgroup/cpu_and_mem
```

2.4. 階層の接続解除

cgroup の階層を *アンマウント*するには、**umount** コマンドを使用します。

```
~]# umount /cgroup/name
```

例:

```
~]# umount /cgroup/cpu_and_mem
```

階層が現在空である場合 (つまり、root cgroup のみを格納している場合) には、階層はアンマウント時に非アクティブ化されます。階層に他の **cgroup** が含まれる場合には、階層はマウントされていなくても、カーネル内でアクティブな状態を維持します。

階層を削除するには、その階層をアンマウントする前に、すべての子 **cgroup** が削除されていることを確認してください。もしくは、**cgclear** コマンドを使用すると、空でない階層も非アクティブ化することができます – 「コントロールグループのアンロード」を参照してください。

2.5. コントロールグループの作成

cgcreate コマンドを使用して **cgroups** を作成します。**cgcreate** の構文は以下のとおりです。

```
cgcreate -t uid:gid -a uid:gid -g subsystems:path
```

ここで、

- **-t** (オプション) は、ユーザー (ユーザー ID、uid) とグループ (グループ ID、gid) を指定して、この **cgroup** の **tasks** 疑似ファイルを所有するためのオプションです。このユーザーは **cgroup** にタスクを追加することができます。

注記

cgroup からタスクを削除するには、異なる **cgroup** に移動するのが唯一の手段である点に注意してください。タスクを移動するには、ユーザーは **移動先の cgroup** への書き込みアクセスが必要となります。元の **cgroup** への書き込みアクセスは重要ではありません。

- **-a** (オプション) – ユーザー (ユーザー ID、uid) とグループ (グループ ID、gid) を指定して、このcgroup の **tasks** 以外の全疑似ファイルを所有するようにします。このユーザーは cgroup 内のタスクが持つシステムリソースへのアクセスを変更できます。

- **-g** – cgroup が作成されるべき階層を、それらの階層に関連付けられる、コンマ区切りの *subsystems* 一覧として指定します。この一覧内のサブシステムが異なる階層にある場合には、それらの各階層内でグループが作成されます。階層の一覧の後には、コロンならびに階層に対して相対的な子グループへの *path* が続きます。このパスには、階層のマウントポイントを入れないでください。

  たとえば、**/cgroup/cpu_and_mem/lab1/**ディレクトリ内に配置されている cgroup が単に **lab1** という名前である場合でも、1 つのサブシステムに対して階層は 1 つしかないため、そのパスは既に一意に特定されていることになります。また、グループは、cgroup が作成された階層内に存在する全サブシステムによって制御される点にも注意してください。**cgcreate** コマンドでこれらのサブシステムが指定されていない場合でも変わりません。例2.5「cgcreate の使用方法」 を参照してください。

同じ階層内の cgroup はすべて同じコントローラーを持つため、子グループは親グループと同じコントローラーを持つことになります。

> **例2.5 cgcreate の使用方法**
>
> **cpu** および **memory** サブシステムが **cpu_and_mem** 階層に一緒にマウントされており、かつ **net_cls** コントローラーが **net** という名前の別の階層にマウントされているシステムを想定して、以下のコマンドを実行します。
>
> ```
> ~]# cgcreate -g cpu,net_cls:/test-subgroup
> ```
>
> **cgcreate** コマンドにより、**test-subgroup** という名前の 2 つのグループを作成します。一方は、**cpu_and_mem** 階層に、もう一方は **net** 階層に入ります。 **cgcreate** コマンドで指定してなくても、**cpu_and_mem** 階層内の **test-subgroup** グループは、 **memory** サブシステムによって制御されます。

その他の方法

cgroup の子を作成するには、**mkdir** コマンドを使用します:

```
~]# mkdir /cgroup/hierarchy/name/child_name
```

例:

```
~]# mkdir /cgroup/cpuset/lab1/group1
```

2.6. コントロールグループの削除

cgdelete で cgroup を削除します。構文は **cgcreate** と同様です。以下のコマンドを実行してください。

```
cgdelete subsystems:path
```

ここで、

- *subsystems* は、コンマ区切りのサブシステム一覧です。

- *path* は、階層の root に対して相対的な cgroup へのパスです。

例:

```
~]# cgdelete cpu,net_cls:/test-subgroup
```

cgdelete で **-r** のオプションを使用すると、すべてのサブグループを再帰的に削除することもできます。

cgroup を削除すると、その cgroup のタスクは、親グループに移動します。

2.7. パラメーターのセッティング

該当する cgroup を修正できるパーミッションを持つユーザーアカウントから、**cgset** コマンドで、サブシステムのパラメーターを設定します。たとえば、**/cgroup/cpuset/group1** が存在する場合には、以下のようなコマンドで、このグループがアクセスできる CPU を指定します。

```
cpuset]# cgset -r cpuset.cpus=0-1 group1
```

cgset の構文は以下のとおりです。

```
cgset -r parameter=value path_to_cgroup
```

ここで、

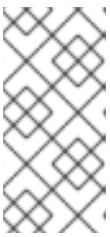
- *parameter* は設定するパラメーターで、特定の cgroup のディレクトリ内のファイルに対応しています。
- *value* はパラメーター用の値です
- *path_to_cgroup* は **階層の root に相対的な** cgroup へのパスです。たとえば、root グループのパラメーターを設定するには、**(/cgroup/cpuacct/** が存在する場合)、以下のコマンドを実行します。

  ```
  cpuacct]# cgset -r cpuacct.usage=0 /
  ```

  また、**.** は root グループ (つまり、root グループ自体) に対して相対的であるため、以下のコマンドを実行することもできます。

  ```
  cpuacct]# cgset -r cpuacct.usage=0 .
  ```

  ただし、**/** が推奨の構文である点に注意してください。



注記

root グループに設定できるパラメーターはごくわずかです (例: 上記の例に示した、**cpuacct.usage** パラメーター)。これは、root グループが既存のリソースをすべて所有しており、特定のパラメーター (例: **cpuset.cpu** パラメーター) を定義することによって既存のプロセスを制限することは意味がないためです。

root グループのサブグループである **group1** のパラメーターを設定するには、以下のコマンドを実行します。

```
cpuacct]# cgset -r cpuacct.usage=0 group1
```

グループ名の末尾のスラッシュ (例: **cpuacct.usage=0 group1/**) はオプションです。

cgset で設定できる値は、特定の階層のよりも高いレベルで設定されている値によって左右される可能性があります。たとえば、**group1** がシステム上の CPU 0 のみを使用するように制限されている場合、**group1/subgroup1** が CPU 0 および 1 を使用するように、もしくは CPU 1 のみを使用するようには設定できません。

また、**cgset** を使用して、異なる **cgroup** からパラメーターをコピーすることもできます。以下はその例です。

```
~]# cgset --copy-from group1/ group2/
```

cgset を使用してパラメーターをコピーする構文は以下のとおりです。

```
cgset --copy-from path_to_source_cgroup path_to_target_cgroup
```

ここで、

- *path_to_source_cgroup* は、コピーするパラメーターを持つ、その階層の **root** グループに対して相対的な **cgroup** へのパスです。
- *path_to_target_cgroup* は、その階層の **root** グループに対して相対的な、コピー先 **cgroup** へのパスです。

一つのグループから別のグループにパラメーターをコピーする前には、様々なサブシステムの必須パラメーターが設定済みであることを確認してください。必須パラメーターが設定されていない場合にはコマンドが失敗してしまいます。必須パラメーターに関する詳しい情報は、必須パラメーター を参照してください。

その他の方法

cgroup ディレクトリ内のパラメーターを設定するには、**echo** コマンドを使用して その値を該当するサブシステムの疑似ファイルに書き込みます。たとえば、以下のコマンドは、値 **0-1** を **cgroup group1** の **cpuset.cpus** 疑似ファイルに書き込みます。

```
~]# echo 0-1 > /cgroup/cpuset/group1/cpuset.cpus
```

この値を入れると、この **cgroup** のタスクはシステム上の CPU 0 と 1 に限定されます。

2.8. コントロールグループへのプロセス移動

cgclassify コマンドを実行して、プロセスを **cgroup** に移動します。

```
~]# cgclassify -g cpu,memory:group1 1701
```

cgclassify の構文は以下のとおりです。

```
cgclassify -g subsystems:path_to_cgroup pidlist
```

ここで、

- *subsystems* は、コンマ区切りのサブシステム一覧、または、利用可能なすべてのサブシステム

に関連付けられた階層内のプロセスを起動するための ***** とします。同じ名前の **cgroup** が複数の階層に存在する場合には、**-g** オプションを指定すると、それらの各グループ内にプロセスが移動される点に注意してください。ここで指定するサブシステムの各階層内に **cgroup** が存在していることを確認してください。

- *path_to_cgroup* は、その階層内の **cgroup** へのパスです
- *pidlist* は、*プロセス識別子* (PID) のスペースで区切られた一覧です。

pid の前に **--sticky** オプションを追加すると、同じ **cgroup** 内の任意の子プロセスを維持することもできます。このオプションを設定せず、かつ **cgred** サービスが稼働中の場合、子プロセスは **/etc/cgrules.conf** の設定に基づいて **cgroup** に割り当てられます。そのプロセス自体は、それを起動した **cgroup** に残ります。

cgclassify を使用すると、いくつかのプロセスを同時に移動することができます。たとえば、以下のコマンドは **1701** と **1138** の PID を持つプロセスを **cgroup group1/** に移動します:

```
~]# cgclassify -g cpu,memory:group1 1701 1138
```

移動する PID は、スペースで区切り、また指定したグループは異なる階層内にある必要がある点に注意してください。

その他の方法

プロセスを **cgroup** のディレクトリに移動するには、その PID を **cgroup** の **tasks** ファイルに 書き込みます。たとえば、PID **1701** の付いたプロセスを **/cgroup/lab1/group1/** にある **cgroup** に 移動する場合は、以下のコマンドを実行します。

```
~]# echo 1701 > /cgroup/lab1/group1/tasks
```

2.8.1. cgred サービス

cgred は、**/etc/cgrules.conf** ファイルに設定されているパラメータセットにしたがってタスクを **cgroup** に移動するサービスです (このサービスが **cgrulesengd** デーモンを起動します)。**/etc/cgrules.conf** ファイル内のエントリは、次の 2 つの形式のいずれかとなります。

- *user subsystems control_group*
- *user*:*command subsystems control_group*

例:

```
maria   devices  /usergroup/staff
```

このエントリは、**maria** というユーザーに属するプロセスはいずれも**/usergroup/staff** cgroup 内に指定されたパラメーターにしたがって **devices** サブシステムにアクセスすることを指定します。特定のコマンドを特定の **cgroup** に関連付けるには、以下のようにして *command* パラメーターを追加します。

```
maria:ftp  devices  /usergroup/staff/ftp
```

このエントリにより、**maria** という名前のユーザーが **ftp** コマンドを使用する時には、**devices** サブシステムが入っている階層の **/usergroup/staff/ftp** cgroup へプロセスが自動的に移動するように指定されるようになります。 ただし、このデーモンは、該当する条件が満たされている場合にのみ、プ

ロセスを **cgroup** に移動する点に注意してください。このため、**ftp** プロセスが、誤ったグループで短時間実行される可能性があります。また、そのプロセスが誤ったグループ内にある間に子プロセスが急速に発生した場合には、それらは移動されない可能性があります。

/etc/cgrules.conf ファイル内のエントリには、以下のような注記を追加することが可能です。

- **@** – *user* にプレフィックスを付けた場合には、個別のユーザーではなくグループを示します。たとえば、**@admins** は **admins** グループ内のすべてのユーザーです。
- ***** –「すべて」を示します。たとえば、**subsystem** フィールド内の ***** は全サブシステムを示します。
- **%** – その上の行の項目と同じ項目を示します。以下はその例です。

```
@adminstaff  devices  /admingroup
@labstaff  %  %
```

2.9. コントロールグループ内のプロセスの開始

重要

サブシステムによっては、それらのサブシステムのいずれかを使用する **cgroup** にタスクを移動できる前に設定しておく必要のある必須パラメーターがあります。たとえば、**cpuset** サブシステムを使用する **cgroup** にタスクを移動する前には、その **cgroup** に対して、**cpuset.cpus** と **cpuset.mems** のパラメーターを定義する必要があります。

このセクション内の例は、コマンド用の正しい構文を示していますが、これは例の中で使用されているコントローラー用に関連した 必須パラメーターを設定しているシステム上でのみ機能します。関連したコントローラーをまだ設定していない場合は、このセクションから サンプルコマンドを直接コピーしても自分のシステム上で機能させることは期待できません。

特定のサブシステムでそのパラメーターが必須であるかについては、3章 *サブシステムと調整可能なパラメーター* を参照してください。

cgexec コマンドを実行して、**cgroup** 内のプロセスを開始することもできます。たとえば、 以下のコマンドは、そのグループに課せられた制限にしたがって、**group1 cgroup** 内で **lynx Web** ブラウザーを起動します。

```
~]# cgexec -g cpu:group1 lynx http://www.redhat.com
```

cgexec の構文は以下のとおりです。

```
cgexec -g subsystems:path_to_cgroup command arguments
```

ここで、

- *subsystems* は、コンマ区切りのサブシステム一覧、または、利用可能なすべてのサブシステムに関連付けられた階層内のプロセスを起動するための ***** とします。「パラメーターのセッティング」 でも説明しているように、同じ名前の **cgroup** が複数の階層に存在する場合には、**-g** オ

プションを指定すると、それらの各グループ内にプロセスが作成される点に注意してください。ここで指定するサブシステムの各階層内に cgroup が存在していることを確認してください。

- *path_to_cgroup* は、階層に対して相対的な cgroup へのパスです。
- *command* は実行するコマンドです。
- *arguments* はそのコマンドの引数です。

command の前に **-- sticky** オプションを追加すると、同じ cgroup の子プロセスを維持することもできます。このオプションを設定しないで **cgred** が稼働すると、子プロセスは **/etc/cgrules.conf** にあるセッティングに基づいて cgroup に割り当てられます。しかし、プロセス自体はそれを開始した cgroup 内に残ります。

その他の方法

新たなプロセスを開始すると、そのプロセスは、親プロセスのグループを継承します。このため、特定の cgroup でプロセスを開始するもう一つの方法として、シェルプロセスをそのグループに移動し (「コントロールグループへのプロセス移動」 を参照)、そのシェルからプロセスを起動することができます。以下はその例です。

```
~]# echo $$ > /cgroup/lab1/group1/tasks
~]# lynx
```

lynx を終了したあとには、既存のシェルは依然として **group1** cgroup にあることに注意してください。したがって、より適切な方法は以下のようになります。

```
~]# sh -c "echo \$$ > /cgroup/lab1/group1/tasks && lynx"
```

2.9.1. コントロールグループ内のサービスの開始

cgroup 内の特定のサービスを起動することができます。cgroup 内で起動できるサービスは、以下の条件を満たしている必要があります。

- **/etc/sysconfig/*servicename*** ファイルを使用する
- サービスを起動するのに **/etc/init.d/functions** の **daemon()** 関数を使用する

cgroup 内で条件に適合したサービスを開始するには、**/etc/sysconfig** 内でそのサービスのファイルを編集して、 **CGROUP_DAEMON="*subsystem*:*control_group*"** の 形式でエントリを追加します。ここで、*subsystem* は特定の階層に関連付けられたサブシステムであり、*control_group* はその階層内の cgroup です。以下はその例です。

```
CGROUP_DAEMON="cpuset:daemons/sql"
```

2.9.2. root コントロールグループ内のプロセスの振る舞い

blkio および **cpu** の特定の設定オプションは、root cgroup 内で実行中のプロセス (タスク) に影響を及ぼします。これは、サブグループ内で実行中のプロセスの場合とは方法が異なります。以下の例を検討してください。

1. root グループ下に 2 つのサブグループを作成します: **/rootgroup/red/** および **/rootgroup/blue/**

2. 各サブグループおよび root グループで **cpu.shares** 設定オプションを定義し、**1** に指定します。

上記で設定したシナリオでは、各グループにプロセスを 1 つずつ配置すると (**/rootgroup/tasks**、**/rootgroup/red/tasks**、および **/rootgroup/blue/tasks** に 1 タスク)、CPU を 33.33% 消費します。

```
/rootgroup/ process:      33.33%
/rootgroup/blue/ process: 33.33%
/rootgroup/red/ process:  33.33%
```

サブグループ **blue** および **red** にその他のプロセスを配置すると、その特定のサブグループに割り当てられている 33.33% の CPU は、そのサブグループ内の複数のプロセス間で分割されます。

しかし、複数のプロセスが root グループに配置されると、 CPU プロセスはグループ別ではなくプロセス別に分割されます。たとえば、**/rootgroup/** に 3 つのプロセス、**/rootgroup/red/** に 1 つのプロセス、**/rootgroup/blue/** に 1 つのプロセスがあり、全グループで**cpu.shares** オプションが **1** に設定されている場合、CPU リソースは以下のように分配されます。

```
/rootgroup/ processes:    20% + 20% + 20%
/rootgroup/blue/ process: 20%
/rootgroup/red/ process:  20%
```

したがって、使用可能なリソースをウェイトまたは配分に基づいて分配する **blkio** および **cpu** 設定オプション (例: **cpu.shares**、**blkio.weight** など) を使用する場合には、全プロセスを root グループから特定のサブグループに移動することを推奨します。root グループから特定のサブグループに全タスクを移動するには、以下のコマンドを使用することができます。

```
rootgroup]# cat tasks >> red/tasks
rootgroup]# echo > tasks
```

2.10. /ETC/CGCONFIG.CONF ファイルの生成

/etc/cgconfig.conf ファイルの設定は、**cgsnapshot** ユーティリティを使用して現在の cgroup 設定から生成することができます。このユーティリティは全サブシステムおよびその cgroup の現在の状態のスナップショットを作成し、その設定を **/etc/cgconfig.conf** ファイルで表示されるのと同じように返します。例2.6「cgsnapshot ユーティリティの使用法」 には、**cgsnapshot** ユーティリティの使用例を示しています。

例2.6 cgsnapshot ユーティリティの使用法

以下のコマンドを使用して、システムで cgroup を設定します。

```
~]# mkdir /cgroup/cpu
~]# mount -t cgroup -o cpu cpu /cgroup/cpu
~]# mkdir /cgroup/cpu/lab1
~]# mkdir /cgroup/cpu/lab2
~]# echo 2 > /cgroup/cpu/lab1/cpu.shares
~]# echo 3 > /cgroup/cpu/lab2/cpu.shares
~]# echo 5000000 > /cgroup/cpu/lab1/cpu.rt_period_us
~]# echo 4000000 > /cgroup/cpu/lab1/cpu.rt_runtime_us
~]# mkdir /cgroup/cpuacct
~]# mount -t cgroup -o cpuacct cpuacct /cgroup/cpuacct
```

上記のコマンドにより、**cpu** サブシステムに、一部のパラメーターには特定の値を指定して、2 つのサブシステムがマウントされ、2 つの cgroup が作成されました。**cgsnapshot** コマンドを (**-s** オプションおよび空の **/etc/cgsnapshot_blacklist.conf** ファイル[4] を使用して) 実行すると、以下のような出力が生成されます。

```
~]$ cgsnapshot -s
# Configuration file generated by cgsnapshot
mount {
        cpu = /cgroup/cpu;
        cpuacct = /cgroup/cpuacct;
}

group lab2 {
        cpu {
                cpu.rt_period_us="1000000";
                cpu.rt_runtime_us="0";
                cpu.shares="3";
        }
}

group lab1 {
        cpu {
                cpu.rt_period_us="5000000";
                cpu.rt_runtime_us="4000000";
                cpu.shares="2";
        }
}
```

上記の例で使用した **-s** オプションは、**cgsnapshot** に対して、**cgsnapshot** ユーティリティのブラックリストまたはホワイトリストで定義されていないパラメーターによって生じるすべての警告を無視するように指示します。パラメーターのブラックリスト化についての詳しい情報は 「パラメーターのブラックリスト化」 を参照してください。パラメーターのホワイトリスト化についての詳しい情報は 「パラメーターのホワイトリスト化」を参照してください。

オプションを指定しない場合、**cgsnapshot** によって生成される出力は、標準出力に返されます。出力をリダイレクトするファイルを指定するには、**-f** オプションを使用します。以下はその例です。

```
~]$ cgsnapshot -f ~/test/cgconfig_test.conf
```

警告

-f オプションを使用する際には、指定したファイルの内容がすべて上書きされる点に注意してください。このため、直接 **/etc/cgconfig.conf** ファイルには出力指定しないことを推奨します。

cgsnapshot ユーティリティはサブシステム別に設定ファイルを作成することもできます。サブシステムの名前を指定すると、 出力はそのサブシステムの対応する設定で構成されます。

```
~]$ cgsnapshot cpuacct
# Configuration file generated by cgsnapshot
mount {
        cpuacct = /cgroup/cpuacct;
}
```

2.10.1. パラメーターのブラックリスト化

cgsnapshot ユーティリティにより、パラメーターのブラックリスト化が可能になります。パラメーターがブラックリスト化されると、**cgsnapshot** によって生成された出力には表示されません。デフォルトでは、**/etc/cgsnapshot_blacklist.conf** ファイルでブラックリストパラメーターをチェックします。パラメーターがブラックリストに入っていない場合には、ホワイトリストをチェックします。別のブラックリストを指定するには、**-b** オプションを使用します。以下はその例です。

```
~]$ cgsnapshot -b ~/test/my_blacklist.conf
```

2.10.2. パラメーターのホワイトリスト化

cgsnapshot ユーティリティにより、パラメーターのホワイトリスト化も可能になります。パラメーターがホワイトリスト化されると、**cgsnapshot** によって生成される出力に表示されます。パラメーターがブラックリスト化もホワイトリスト化もされていない場合には、警告が表示され、以下のような内容を通知します。

```
~]$ cgsnapshot -f ~/test/cgconfig_test.conf
WARNING: variable cpu.rt_period_us is neither blacklisted nor whitelisted
WARNING: variable cpu.rt_runtime_us is neither blacklisted nor whitelisted
```

デフォルトではホワイトリスト設定ファイルはありません。ホワイトリストとして使用するファイルを指定するには、**-w** オプションを使用します。以下はその例です。

```
~]$ cgsnapshot -w ~/test/my_whitelist.conf
```

-t オプションを指定することにより、**cgsnapshot** に対してホワイトリストのみからのパラメーターを使用して設定を生成するように指示します。

2.11. コントロールグループに関する情報の取得

2.11.1. プロセスの確認

プロセスが属する **cgroup** を確認するには以下のコマンドを実行します:

```
~]$ ps -O cgroup
```

また、プロセスの **PID** がわかっている場合は、以下のコマンドを実行します:

```
~]$ cat /proc/PID/cgroup
```

2.11.2. サブシステムの確認

カーネルで使用可能なサブシステムおよびそれらがどのようにして階層にまとめてマウントされているかを確認するには、以下のコマンドを実行します。

```
~]$ cat /proc/cgroups
```

また、特定のサブシステムのマウントポイントを確認するには、以下のコマンドを実行します。

```
~]$ lssubsys -m subsystems
```

subsystems は、対象となるサブシステムの一覧です。**lssubsys -m** コマンドでは、各階層ごとの最上位のマウントポイントのみが返される点に注意してください。

2.11.3. 階層の確認

階層は **/cgroup** 下にマウントすることを推奨します。ご使用のシステムがそのようになっていることを前提として、そのディレクトリの内容を一覧表示または閲覧し、階層の一覧を取得します。**tree** がインストールされている場合には実行して、全階層およびその中の **cgroup** の概観を確認します。

```
~]$ tree /cgroup
```

2.11.4. コントロールグループの確認

システム上の **cgroup** を一覧表示するには、以下のコマンドを実行します。

```
~]$ lscgroup
```

controller*:*path の形式でコントローラーとパスを指定すると、特定の階層への出力を限定することができます。以下はその例です。

```
~]$ lscgroup cpuset:adminusers
```

cpuset サブシステムが接続されている階層内の**adminusers** cgroup のサブグループのみを一覧表示します。

2.11.5. コントロールグループのパラメーターの表示

特定の **cgroup** のパラメーターを表示するには、以下のコマンドを実行します。

```
~]$ cgget -r parameter list_of_cgroups
```

ここで *parameter* は、サブシステムの値を含んだ擬似ファイルで、*list_of_cgroups* は **cgroup** のスペース区切りの一覧です。以下はその例です。

```
~]$ cgget -r cpuset.cpus -r memory.limit_in_bytes lab1 lab2
```

cgroup **lab1** および **lab2** の **cpuset.cpus** 値と **memory.limit_in_bytes** 値を表示します。

パラメーター自体の名前がわからない場合には、以下のようなコマンドを使用してください。

```
~]$ cgget -g cpuset /
```

2.12. コントロールグループのアンロード

警告

cgclear コマンドにより、全階層内のすべての cgroup が破棄されます。これらの階層が設定ファイル内に記載されていない場合は、簡単には再構築できません。

cgroup ファイルシステム全体を消去するには、**cgclear** コマンドを使用します。

cgroup 内のタスクはすべて、階層内の root ノードに再割り当てされ、全 cgroup が削除されて、ファイルシステム自体が システムからアンマウントされます。したがって、以前にマウントされていた階層をすべて破棄することになります。最後に、cgroup ファイルシステムがマウントされていたディレクトリが実際に削除されます。

注記

mount コマンドを使用して cgroup を作成すると (**cgconfig** サービスを使用して作成するのではなく)、**/etc/mtab** ファイル (マウント済みファイルシステムテーブル) にエントリが作成されます。この変更は、**/proc/mounts** ファイルにも反映されます。しかし、**cgclear** コマンドを使用した cgroup のアンロードには、他の**cgconfig** コマンドと同様に直接のカーネルインターフェースが使用され、その変更は **/etc/mtab** ファイルに反映されず、新規情報のみが **/proc/mounts** ファイルに書き込まれます。このため、**cgclear** コマンドを使用して cgroup をアンロードした後に、マウント解除された cgroup が **/etc/mtab** で可視状態のままとなって、 **mount** コマンドを実行すると表示されてしまう場合があります。マウントされている cgroup の正確な一覧は **/proc/mounts** ファイルを参照してください

2.13. 通知 API の使用

cgroup の通知 API により、ユーザースペースアプリケーションは cgroup のステータス変更についての通知を受信することができます。現在、通知 API は Out of Memory (OOM) 制御ファイルのモニタリングのみをサポートしています: **memory.oom_control**。通知ハンドラーを作成するには、以下の手順にしたがって、C プログラムを作成します。

1. **eventfd()** 関数を使用して、イベント通知のファイル記述子を作成します。詳しくは、**eventfd(2)** の man ページを参照してください。
2. **memory.oom_control** ファイルをモニタリングするには、**open()** 関数を使用して開きます。詳しくは **open(2)** の man ページを参照してください。
3. モニタリングする **memory.oom_control** ファイルの cgroup の **cgroup.event_control** ファイルに以下の引数を書き込むには、**write()** 関数を使用します。

   ```
   <event_file_descriptor> <OOM_control_file_descriptor>
   ```

   ここで、

   - **cgroup.event_control** ファイルを開くには、*event_file_descriptor* を使用します。
   - 適切な **memory.oom_control** ファイルを開くには、*OOM_control_file_descriptor* を使用します。

ファイルへの書き込みについての詳しい情報は、**write(1)** の man ページを参照してください。

上記のプログラムが起動すると、モニタリング対象の cgroup 内の OOM 状態が通知されます。OOM 通知は、root 以外の cgroup でしか機能しない点に注意してください。

memory.oom_control の調節可能なパラメーターについての詳しい情報は、「memory」 を参照してください。OOM 制御の通知設定についての詳しい情報は、例3.3「OOM の制御と通知」 を参照してください。

2.14. その他のリソース

cgroup コマンドに関する最も確実な資料は、libcgroup パッケージで提供されている man ページです。以下にあげる man ページの一覧には、セクション番号を記載しています。

libcgroup の man ページ

- **man 1 cgclassify** – **cgclassify** コマンドは、実行中のタスクを単一もしくは複数の cgroup に移動するのに使用します。

  man 1 cgclear – **cgclear** コマンドは、一つの階層内のすべての cgroup を削除するのに使用します。

  man 5 cgconfig.conf – cgroup は **cgconfig.conf** ファイル内で定義されます。

  man 8 cgconfigparser – **cgconfigparser** コマンドは、**cgconfig.conf** ファイルを解析して、階層をマウントします。

  man 1 cgcreate – **cgcreate** コマンドは、階層内に新たな cgroup を作成します。

  man 1 cgdelete – **cgdelete** コマンドは、特定の cgroup を削除します。

  man 1 cgexec – **cgexec** コマンドは、特定の cgroup 内のタスクを実行します。

  man 1 cgget – **cgget** コマンドは、cgroup のパラメーターを表示します。

  man 1 cgsnapshot – **tcgsnapshot** コマンドは、既存のサブシステムから設定ファイルを生成します。

  man 5 cgred.conf – **cgred.conf** は、**cgred** サービスの設定ファイルです。

  man 5 cgrules.conf – **cgrules.conf** には、特定の cgroup にタスクが属する場合に判断するためのルールが含まれます。

  man 8 cgrulesengd – **cgrulesengd** サービスは、タスクを cgroup に配分します。

  man 1 cgset – **cgset** コマンドは、cgroup のパラメーターを設定します。

  man 1 lscgroup – **lscgroup** コマンドは、階層内の cgroup を一覧表示します。

  man 1 lssubsys – **lssubsys** コマンドは、特定のサブシステムを含む階層を一覧表示します。

[3] **lssubsys** コマンドは、libcgroup パッケージによって提供されるユーティリティの一つです。これを使用するには、libcgroup をインストールする必要があります。**lssubsys** を実行できない場合には、2章*コントロールグループの使用法* を参照してください。

[4] **cpu.shares** パラメーターは、デフォルトで**/etc/cgsnapshot_blacklist.conf** ファイルに指定されるため、例2.6「cgsnapshot ユーティリティの使用法」で生成される出力では省略されます。したがって、この例では空の **/etc/cgsnapshot_blacklist.conf** ファイルを使用しています。

第3章 サブシステムと調整可能なパラメーター

*サブシステム*とは、cgroup を認識するカーネルモジュールで、通常は、異なるレベルのシステムリソースを異なる cgroup に割り当てるリソースコントローラーです。ただし、サブシステムは、プロセスグループによって異なった扱いをする必要がある場合に、カーネルとのその他の対話用にプログラムすることも可能です。新規サブシステムを開発するための *アプリケーションプログラミングインターフェース* (API) については、ご使用のシステムの **/usr/share/doc/kernel-doc-*kernel-version*/Documentation/cgroups/** にインストールされているカーネルのドキュメント **cgroups.txt** (kernel-doc パッケージにより提供) に記載されています。cgroup に関するドキュメントの最新バージョンは、オンラインでもご覧いただけます: http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt。ただし、最新版のドキュメントに記載されている機能は、ご使用のシステムにインストールされているカーネルで利用可能な機能と一致しない場合がある点に注意してください。

cgroup 用のサブシステムパラメーターを含んでいる *状態オブジェクト (State object)* は cgroup の仮想ファイルシステム内で *疑似ファイル (pseudofiles)* として表示されます。これらの疑似ファイルは、シェルコマンドまたはそれに相当するシステムコールで操作することができます。たとえば、**cpuset.cpus** は、cgroup によるアクセスが許可されている CPU を指定する疑似ファイルです。システム上で稼働する Web サーバー用の cgroup が **/dev/cgroup/webserver/** である場合には、以下のコマンドを実行してください。

```
~]# echo 0,2 > /cgroup/cpuset/webserver/cpuset.cpus
```

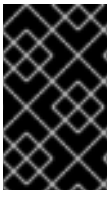
上記のコマンドは、**0,2** の値を **cpuset.cpus** 疑似ファイルに書き込むことにより、**/cgroup/cpuset/webserver/tasks** に記載されている PID のタスクがシステム上の CPU 0 と CPU 2 のみを使用するように限定します。

3.1. BLKIO

ブロック I/O (**blkio**) サブシステムは、cgroup 内のタスクによる、ブロックデバイス上の I/O へのアクセスを制御・監視します。これらの疑似ファイルに値を書き込むと、アクセスまたは帯域幅が限定され、またこれらの疑似ファイルから値を読み取ると、I/O 操作についての情報が提供されます。

blkio サブシステムは I/O へのアクセスを制御する 2 つのポリシーを提供します。

- *重み付け比例配分* – Completely Fair Queuing I/O スケジューラーに実装されているこのポリシーにより、特定の cgroup にウェイトを設定することができます。これは、各 cgroup に、全予約済み I/O 操作に対する一定の割合が (その cgroup のウェイト応じて) 設定されることを意味します。詳しい情報は 「重み付け比例配分の調整可能なパラメーター」を参照してください。
- *I/O スロットリング (上限)* – このポリシーは、特定のデバイスが実行する I/O 操作数の上限を設定するのに使用します。これは、デバイスの **読み取り** または **書き込み** 操作の速度を制限できることを意味します。詳しくは 「I/O スロットリングの調整可能なパラメーター」を参照してください。



重要

現在、block I/O サブシステムは、バッファリングされた **書き込み** 操作には機能しません。これは、バッファリングされた **読み取り** 操作には機能しますが、主として直接 I/O を対象としています。

3.1.1. 重み付け比例配分の調整可能なパラメーター

blkio.weight

デフォルトで cgroup に提供される ブロック I/O アクセスの相対的比率 (*ウェイト*) を **100** から **1000** の範囲内で指定します。この値は、**blkio.weight_device** パラメーターを使用すると、特定のデバイスでオーバーライドされます。たとえば、ブロックデバイスにアクセスするためのデフォルトのウェイト **500** を cgroup に割り当てるには、以下のコマンドを実行します。

```
echo 500 > blkio.weight
```

blkio.weight_device

cgroup に提供される特定のデバイス上の I/O アクセスの相対的比率 (*ウェイト*) を **100** から **1000** の範囲内で指定します。このパラメーターの値は、指定したデバイスの **blkio.weight** パラメーターの値を上書きします。これらの値は、*major:minor weight* の形式を取り、*major* と *minor* は、http://www.kernel.org/doc/Documentation/devices.txt から入手可能な 『Linux Allocated Devices』 (別名 『Linux Devices List』) で指定されているデバイスタイプとノード番号となります。たとえば、cgroup による **/dev/sda** へのアクセスに **500** のウェイトを割り当てるには、以下のコマンドを実行します。

```
echo 8:0 500 > blkio.weight_device
```

『Linux Allocated Devices』 の表記では、**8:0** は **/dev/sda** を示します。

3.1.2. I/O スロットリングの調整可能なパラメーター

blkio.throttle.read_bps_device

デバイスが実行できる **読み取り** 操作数の上限を指定します。**読み取り** 操作の速度はバイト毎秒単位で指定します。エントリは *major*、*minor*、および *bytes_per_second* の 3 つのフィールドで構成されます。*major* と *minor* は、『Linux Allocated Devices』 で指定されているデバイスタイプとノード番号です。また *bytes_per_second* は、**読み込み** 操作を実行できる速度の上限です。たとえば、**/dev/sda** デバイスが最大 10 MBps で **読み取り** 操作を実行できるようにするには、以下のコマンドを実行します。

```
~]# echo "8:0 10485760" >
/cgroup/blkio/test/blkio.throttle.read_bps_device
```

blkio.throttle.read_iops_device

デバイスが実行できる **読み取り** 操作数の上限を指定します。**読み取り** 操作は毎秒の操作数で指定します。エントリは、*major*、*minor*、および *operations_per_second* の 3 つのフィールドで構成されます。*major* と *minor* は、『Linux Allocated Devices』 で指定されているデバイスタイプとノード番号です。また *operations_per_second* は **読み取り** 操作を実行できる速度の上限です。たとえば、**/dev/sda** デバイスが最大 10 回の **読み取り** 操作を実行できるようにするには、以下のコマンドを実行します。

```
~]# echo "8:0 10" > /cgroup/blkio/test/blkio.throttle.read_iops_device
```

blkio.throttle.write_bps_device

デバイスが実行できる **書き込み** 操作数の上限を指定します。**書き込み** 操作の速度はバイト毎秒単位で指定します。エントリは *major*、*minor*、および *bytes_per_second* の 3 つのフィールドで構成されます。*major* と *minor* は、『Linux Allocated Devices』 で指定されているデバイスタイプとノード番

号です。また *bytes_per_second* は、**書き込み** 操作を実行できる速度の上限です。たとえば、**/dev/sda** デバイスが最大 10 MBps で **書き込み** 操作を実行できるようにするには、以下のコマンドを実行します。

```
~]# echo "8:0 10485760" >
/cgroup/blkio/test/blkio.throttle.write_bps_device
```

blkio.throttle.write_iops_device

デバイスが実行できる **書き込み** 操作数の上限を指定します。**書き込み** 操作の速度は、毎秒の操作数で指定します。エントリは、*major*、*minor*、および *operations_per_second* の 3 つのフィールドで構成されます。*major* と *minor* は、『Linux Allocated Devices』 で指定されているデバイスタイプとノード番号です。また *operations_per_second* は、**書き込み** 操作を実行できる速度の上限です。たとえば、**/dev/sda** デバイスが **書き込み** 操作を最大で毎秒 10 回実行できるようにするには、以下のコマンドを実行します。

```
~]# echo "8:0 10" > /cgroup/blkio/test/blkio.throttle.write_iops_device
```

blkio.throttle.io_serviced

スロットリングのポリシーに認識されるように、特定のデバイス上で cgroup により実行された I/O 操作の回数をレポートします。エントリは *major*、*minor*、*operation*、および *number* の 4 つのフィールドで構成されます。*major* と *minor* は 『Linux Allocated Devices』 で指定されているデバイスタイプとノード数で、*operation* は操作のタイプ (**read**、**write**、**sync**、または **async**)、*number* は操作回数を示します。

blkio.throttle.io_service_bytes

cgroup により、特定のデバイスとの間で転送されたバイト数をレポートします。***blkio.io_service_bytes*** と ***blkio.throttle.io_service_bytes*** の唯一の相違点は、前者の場合 CFQ スケジューラーが要求キューで稼働している時には更新されない点です。エントリは *major*、*minor*、*operation*、および *bytes* の 4 つのフィールドで構成されます。*major* と *minor* は、『Linux Allocated Devices』 で指定されているデバイスタイプとノード番号です。また *operation* は操作のタイプ (**read**、**write**、**sync**、または **async**)、*bytes* は転送されるバイト数を示します。

3.1.3. blkio 共通の調整可能なパラメーター

以下のパラメーターは、「blkio」 に記載のいずれのポリシーにも使用することができます。

blkio.reset_stats

その他の疑似ファイルに記録されている統計をリセットします。この cgroup の統計をリセットするには、このファイルに整数を書き込みます。

blkio.time

cgroup が特定のデバイスに I/O アクセスを行った時間をレポートします。エントリは、*major*、*minor*、および *time* の 3 つのフィールドで構成されます。*major* と *minor* は 『Linux Allocated Devices』 で指定されているデバイスタイプとノード番号、また*time* はミリ秒 (ms) 単位の時間です。

blkio.sectors

cgroup により、特定のデバイスとの間で転送されたセクターの数をレポートします。エントリは、*major*、*minor*、および *sectors* の 3 つのフィールドで構成されます。*major* と *minor* は 『Linux

Allocated Devices』 で指定されているデバイスタイプとノード番号、また*sectors* はセクター数です。

blkio.avg_queue_size

グループ存在の全時間にわたる、cgroup による I/O 操作の平均キューサイズをレポートします。キューサイズは、この cgroup がタイムスライスを取得する度にサンプリングされます。このレポートは、システム上で **CONFIG_DEBUG_BLK_CGROUP=y** が設定されている場合のみ利用可能である点に注意してください。

blkio.group_wait_time

cgroup が一つのキューで費した待ち時間の合計を (ナノ秒単位 – ns で) レポートします。レポートは、この cgroup がタイムスライスを取得する度に更新されるので、cgroup がタイムスライスを待っている間にこの疑似ファイルを読んだ場合には、現在キューに入っている操作を待つのに費した時間は含まれません。このレポートは、システム上で **CONFIG_DEBUG_BLK_CGROUP=y** が設定されている場合のみ利用可能である点に注意してください。

blkio.empty_time

cgroup が保留中の要求なしに費した時間の合計を (ナノ秒単位 – ns で) レポートします。レポートは、保留中の要求がこの cgroup のキューに入る度に更新されるので、cgroup に保留中の要求がない時に疑似ファイルを読んだ場合には、現在の空の状態で費した時間はレポートには含まれません。このレポートは、システム上で **CONFIG_DEBUG_BLK_CGROUP=y** が設定されている場合のみ利用可能である点に注意してください。

blkio.idle_time

すでにキューに入っている別の要求や別のグループからの要求よりも高い要求に備えて、cgroup に対してスケジューラーがアイドリング状態で費した時間の合計を (ナノ秒単位 – ns で) レポートします。レポートは、グループがアイドリング状態でなくなった時点で毎回更新されるため、cgroup がアイドリング状態の間にこの疑似ファイルを読み込んだ場合には、最新のアイドリング状態で費した時間はレポートには含まれません。このレポートは、システム上で **CONFIG_DEBUG_BLK_CGROUP=y** が設定されている場合のみ利用可能である点に注意してください。

blkio.dequeue

cgroup による I/O 操作の要求がキューから削除された回数をデバイス別にレポートします。エントリは、*major*、*minor*、および *number* の 3 つのフィールドで構成されます。*major* と *minor* は、『Linux Allocated Devices』 で指定されているデバイスタイプとノード番号です。*number* は、グループがキューから削除された要求の回数です。このレポートは、システム上で **CONFIG_DEBUG_BLK_CGROUP=y** が設定されている場合のみ利用可能である点に注意してください。

blkio.io_serviced

CFS スケジューラーに認識されるように、cgroup により特定のデバイス上で実行された I/O 操作の回数をレポートします。エントリは *major*、*minor*、*operation*、および *number* の 4 つのフィールドで構成されます。*major* と *minor* は 『Linux Allocated Devices』 で指定されているデバイスタイプとノード数で、*operation* は操作のタイプ (**read**、**write**、**sync**、または **async**)、*number* は操作回数を示します。

blkio.io_service_bytes

CFQ スケジューラーに認識されるように、cgroup により特定のデバイスとの間で転送されたバイト数をレポートします。エントリは、*major*、*minor*、*operation*、および *bytes* の 4 つのフィールドで構成されます。*major* と *minor* は、『Linux Allocated Devices』 で指定されているデバイスタイプと

ノード番号です。*operation* は操作のタイプ (**read**、**write**、**sync**、または **async**)、*bytes* は転送されたバイト数を示します。

blkio.io_service_time

CFQ スケジューラーに認識されるように、cgroup により特定のデバイス上で行われる I/O 操作の要求がディスパッチされてから完了するまでの合計時間をレポートします。エントリは、*major*、*minor*、*operation*、および *time* の 4 つのフィールドで構成されます。*major* と *minor* は、『Linux Allocated Devices』 で指定されているデバイスタイプとノード番号です。*operation* は操作のタイプ (**read**、**write**、**sync**、または **async**)、*time* は時間をナノ秒 (ns) 単位で示します。時間は、大きな単位ではなく、ナノ秒単位でレポートされるため、ソリッドステートのデバイスでもレポートが有意となります。

blkio.io_wait_time

スケジューラーキュー内のサービスを待つのに費した、cgroup による 特定のデバイス上の I/O 操作の合計時間をレポートします。このレポートを解析する際には、以下の点に注意してください。

- レポートされる時間は、cgroup 自体が I/O 操作を待つのに費した時間ではなく、cgroup の全 I/O 操作の累計であるため、経過時間の合計よりも長い場合があります。グループ全体として費した待ち時間を確認するには、**blkio.group_wait_time** パラメーターを使用します。
- デバイスに **queue_depth > 1** がある場合は、レポートされる時間には、デバイスが要求を並べ替える間に費した待ち時間ではなく、要求がデバイスにディスパッチされるまでの時間のみが含まれます。

エントリは、*major*、*minor*、*operation*、および *time* の 4 つのフィールドで構成されます。*major* と *minor* は、『Linux Allocated Devices』 で指定されているデバイスタイプとノード番号です。*operation* は操作のタイプ (**read**、**write**、**sync**、または **async**)、*time* はナノ秒 (ns) 単位の時間を示します。時間は、大きな単位ではなく、ナノ秒単位でレポートされるため、ソリッドステートのデバイスでもレポートが有意となります。

blkio.io_merged

cgroup により、I/O 操作要求にマージされた、BIOS 要求数をレポートします。エントリは *number* と *operation* の 2 つのフィールドで構成されます。*number* は、要求数、*operation* は操作のタイプ (**read**、**write**、**sync**、または **async**) を示します。

blkio.io_queued

cgroupt により、I/O 操作のキューに入れられた要求の数をレポートします。エントリは、*number* と *operation* の 2 つのフィールドで構成されます。*number* は、要求数、*operation* は操作のタイプ (**read**、**write**、**sync**、または **async**) を示します。

3.1.4. 使用例

さまざまな **blkio.weight** 値を使用して 2 つの異なる cgroup で 2 つの **dd** スレッドを実行する簡易テストについては、例3.1「blkio の重み付け比例配分」 を参照してください。

例3.1 blkio の重み付け比例配分

1. **blkio** サブシステムをマウントします。

```
~]# mount -t cgroup -o blkio blkio /cgroup/blkio/
```

2. **blkio** サブシステム用に 2 つの cgroup を作成します。

```
~]# mkdir /cgroup/blkio/test1/
~]# mkdir /cgroup/blkio/test2/
```

3. あらかじめ作成した cgroup に別々の **blkio** ウェイトを設定します。

```
~]# echo 1000 > /cgroup/blkio/test1/blkio.weight
~]# echo 500 > /cgroup/blkio/test2/blkio.weight
```

4. 大容量ファイルを 2 つ作成します。

```
~]# dd if=/dev/zero of=file_1 bs=1M count=4000
~]# dd if=/dev/zero of=file_2 bs=1M count=4000
```

上記のコマンドにより、サイズが 4 GB のファイルが 2 つ (**file_1** および **file_2**) 作成されます。

5. 各テスト cgroup で、1 つの大容量ファイルに対して **dd** コマンド (ファイルの内容を読み取り、null デバイスに出力するコマンド) を実行します。

```
~]# cgexec -g blkio:test1 time dd if=file_1 of=/dev/null
~]# cgexec -g blkio:test2 time dd if=file_2 of=/dev/null
```

これらのコマンドはいずれも、完了すると完了時間を出力します。

6. **iotop** ユーティリティを使用すると、実行中の 2 つの **dd** スレッドと同時に、リアルタイムでパフォーマンスを管理することができます。**iotop** ユーティリティをインストールするには、root として **yum install iotop** のコマンドを実行します。 以下は、前に起動した **dd** スレッドの実行中に **iotop** ユーティリティで表示される出力の例です。

```
Total DISK READ: 83.16 M/s | Total DISK WRITE: 0.00 B/s
    TIME  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN
IO    COMMAND
15:18:04 15071 be/4 root       27.64 M/s    0.00 B/s  0.00 %
92.30 % dd if=file_2 of=/dev/null
15:18:04 15069 be/4 root       55.52 M/s    0.00 B/s  0.00 %
88.48 % dd if=file_1 of=/dev/null
```

例3.1「blkio の重み付け比例配分」 で最も正確な結果を得るには、**dd** コマンドを実行する前に以下のコマンドを実行して、すべてのファイルシステムのバッファーをフラッシュし、ページキャッシュ、デントリ、inode を解放しておきます。

```
~]# sync
~]# echo 3 > /proc/sys/vm/drop_caches
```

また、スループットを代償にして、グループ間の分離を強化する *group isolation* を有効化することができます。グループ分離が無効になっている場合、公平性が期待できるのは順次ワークロードに対してのみです。デフォルトでは、グループ分離は有効化されており、ランダム I/O ワークロードでも公平性が期待できます。グループ分離を有効化するには、以下のコマンドを実行します。

```
~]# echo 1 > /sys/block/<disk_device>/queue/iosched/group_isolation
```

ここで *<disk_device>* は対象のデバイス名を表しています (例:**sda**)。

3.2. CPU

cpu サブシステムは cgroup への CPU アクセスをスケジュールします。CPU リソースへのアクセスは、次の 2 つのスケジューラーを使用してスケジュールすることができます。

- *Completely Fair Scheduler (CFS)*– タスクの優先度/ウェイトや cgroup に割り当てられている割合に応じて、タスクグループ (cgroup) 間で CPU 時間 (CPU 帯域幅) を比例配分するプロポーショナルシェアスケジューラー。CFS を使用したリソース制限についての詳しい情報は 「CFS の調整可能なパラメーター」 を参照してください。

- *リアルタイムスケジューラー (RT)*– リアルタイムのタスクが使用できる CPU 時間を指定する方法を提供するタスクスケジューラー。リアルタイムのタスクのリソース制限についての詳しい情報は、「RT の調整可能なパラメーター」 を参照してください。

3.2.1. CFS の調整可能なパラメーター

スケジューラーには作業を節約する性質があるため、CFS では、十分なアイドル CPU サイクルが利用可能な場合に、cgroup は割り当てられている配分以上に CPU を使用することができます。これは通常、相対的配分に基づいて CPU 時間を消費する cgroup の場合に該当します。cgroup が利用できる CPU の量に対するハードリミットが必要な場合には、上限の適用を使用することができます (タスクが一定の CPU 時間以上を使用できないようにします)。

以下のオプションは、CPU の上限の適用または相対的配分を設定するのに使用することができます。

上限の適用の調整可能なパラメーター

cpu.cfs_period_us

cgroup による CPU リソースへのアクセスを再割り当てする一定間隔をマイクロ秒単位 (µs、ただしここでは "***us***" と表示) で指定します。cgroup 内のタスクが 1 秒あたり 0.2 秒間、単一の CPU にアクセスできる必要がある場合には、**cpu.cfs_quota_us** を **200000**に、**cpu.cfs_period_us** を **1000000** に設定してください。**cpu.cfs_quota_us** パラメーターの上限は 1 秒、下限は 1000 マイクロ秒です。

cpu.cfs_quota_us

cgroup 内の全タスクが (**cpu.cfs_period_us** で定義された) 一定の期間に実行される合計時間をマイクロ秒単位 (µs、ただしここでは "***us***" と表示) で指定します。クォータによって指定された時間を cgroup 内のタスクがすべて使い切ってしまうと、その期間により指定されている残りの時間はタスクがスロットリングされ、次の期間まで実行を許可されなくなります。cgroup 内のタスクが 1 秒あたり 0.2 秒間、単一の CPU にアクセスできる必要がある場合には **cpu.cfs_quota_us** を **200000** に、**cpu.cfs_period_us** を **1000000** に設定します。クォータおよび期間のパラメーターは CPU ベースで動作する点に注意してください。プロセスが 2 つの CPU を完全に使用できるようにするには、たとえば、**cpu.cfs_quota_us** を **200000** に、 **cpu.cfs_period_us** を **100000** に設定します。

cpu.cfs_quota_us の値を **-1** に設定すると、cgroup が CPU 時間制限を順守しないことを示します。これは、全 cgroup のデフォルト値でもあります (root cgroup は除く)。

cpu.stat

以下の値を使用して、CPU 時間の統計をレポートします。

- **nr_periods** – 経過済みの期間間隔の数 (**cpu.cfs_period_us** で指定されている)
- **nr_throttled** – cgroup 内のタスクがスロットリングされた回数 (クォータによって指定された利用可能な時間をすべて使い果たしたため、実行することができない)
- **throttled_time** – cgroup 内のタスクがスロットリングされた合計時間 (ナノ秒単位)

相対的配分の調整可能なパラメーター

cpu.shares

cgroup 内のタスクで使用できる CPU 時間の相対的配分を指定する整数値を含みます。たとえば、**cpu.shares** が **100** に設定された 2 つの cgroup のタスクには同等の CPU 時間が提供されますが、**cpu.shares** が **200** に設定された cgroup のタスクには、**cpu.shares** が **100** に設定された cgroup のタスクの 2 倍の CPU 時間が提供されます。**cpu.shares** ファイルで指定する値は、**2** 以上とする必要があります。

CPU 時間の配分は、マルチコアシステム上の全 CPU コアを対象に分配されることに注意してください。マルチコアシステムで cgroup の上限が CPU の 100% に設定されている場合、各 CPU コアの 100% を使用できるということになります。次の例を検討してください: cgroup **A** が CPU の 25%、cgroup **B** が CPU の 75% を使用するように設定されている場合、4 コアのシステムで CPU を集中的に使用するプロセスを起動すると (**A** で 1 プロセス、**B** で 3 プロセス)、CPU 配分は以下のように分配されます。

表3.1 CPU 配分の分配

PID	cgroup	CPU	CPU 配分
100	A	0	CPU0 の 100%
101	B	1	CPU1 の 100%
102	B	2	CPU2 の 100%
103	B	3	CPU3 の 100%

相対的配分を使用して CPU アクセスを指定する場合に考慮する必要のあるリソース管理への影響は以下の 2 点です。

- CFS は同等の CPU 使用率を要求しないので、cgroup が消費できる CPU 時間を予測するのは困難です。1 つの cgroup 内のタスクがアイドル状態で CPU 時間を全く消費していない場合、その残り時間は未使用 CPU サイクルのグローバルプールに収集されます。他の cgroup は、このプールから CPU サイクルを借りることができます。
- cgroup が使用できる実際の CPU 時間は、システムに存在する cgroup の数によって異なります。cgroup の相対的配分が **1000** に設定され、かつ他の 2 つの cgroup の相対的配分が **500** に設定されている場合、全 cgroup 内のプロセスが CPU の 100% の使用を試みると、最初の cgroup に全 CPU 時間の 50% が割り当てられます。しかし、**1000** の相対的配分が設定された別のグループが追加されると、 最初の cgroup は CPU の 33% しか使用できなくなります (残りの cgroup は、CPU の 16.5%、16.5%、33% となります)。

3.2.2. RT の調整可能なパラメーター

RT スケジューラーは CFS の上限適用の制御 (詳細は 「CFS の調整可能なパラメーター」 を参照) と同様に機能しますが、CPU アクセスはリアルタイムのタスクのみに限定されます。リアルタイムのタスクが CPU にアクセスできる時間は、各 cgroup に対してランタイムと時間を割り当てることによって決定されます。これにより、cgroup 内の全タスクには、1 回のランタイムに定義された時間の CPU アクセスが許可されます (例: cgroup 内のタスクを 1 秒あたり 0.1 秒間実行するのを許可することができる)。

cpu.rt_period_us

リアルタイムスケジューリングのタスクにのみで使用できます。このパラメーターは、cgroup による CPU リソースへのアクセスを再割り当てする一定間隔をマイクロ秒単位 (µs、ただしここでは "*us*" と表示) で指定します。cgroup 内のタスクが 1 秒あたり 0.2 秒間、単一の CPU にアクセスできる必要がある場合には、**cpu.rt_runtime_us** を **200000**に、**cpu.rt_period_us** を **1000000** に設定してください。

cpu.rt_runtime_us

リアルタイムスケジューリングのタスクのみに適用されます。cgroup 内のタスクによる CPU リソースへのアクセスの最長連続時間をマイクロ秒（µs、ただしここでは "*us*" と表示）で指定します。この上限を設定することにより、cgroup 内のタスクが CPU 時間を独占できないようにします。cgroup 内のタスクが 1 秒あたり 0.2 秒間、単一の CPU にアクセスできるようにする必要がある場合は、**cpu.rt_runtime_us** を **200000**、**cpu.rt_period_us** を **1000000** に設定します。ランタイムおよび時間のパラメーターは CPU ベースで動作する点に注意してください。リアルタイムのタスクが 2 つの CPU を完全に使用できるようにするには、たとえば **cpu.cfs_quota_us** を **200000** に、**cpu.cfs_period_us** を **100000** に設定します。

3.2.3. 使用例

例3.2 CPU アクセスの制限

以下の例は、既存の cgroup 階層が設定済みで、**cpu** サブシステムがシステム上にマウントされていることを前提としています。

- 1 つの cgroup が単一の CPU の 25% を使用し、別の cgroup が 同じ CPU の 75% を使用できるようにするには、以下のコマンドを実行します。

```
~]# echo 250 > /cgroup/cpu/blue/cpu.shares
~]# echo 750 > /cgroup/cpu/red/cpu.shares
```

- cgroup が単一の CPU を完全に使用するように制限するには、以下のコマンドを実行します。

```
~]# echo 10000 > /cgroup/cpu/red/cpu.cfs_quota_us
~]# echo 10000 > /cgroup/cpu/red/cpu.cfs_period_us
```

- cgroup が単一の CPU の 10 % を使用するように制限するには、以下のコマンドを実行します。

```
~]# echo 10000 > /cgroup/cpu/red/cpu.cfs_quota_us
~]# echo 100000 > /cgroup/cpu/red/cpu.cfs_period_us
```

- マルチコアシステムで cgroup が 2 つの CPU コアを完全に使用できるようにするには、以下のコマンドを実行します。

```
~]# echo 200000 > /cgroup/cpu/red/cpu.cfs_quota_us
~]# echo 100000 > /cgroup/cpu/red/cpu.cfs_period_us
```

3.3. CPUACCT

CPU Accounting (**cpuacct**) サブシステムは、cgroup 内のタスクで使用される CPU リソースに関する自動レポートを生成します。3 つのレポートが利用できます:

cpuacct.usage

この cgroup 内の全タスク (下位階層のタスクを含む) により消費される総 CPU 時間 (ナノ秒単位) をレポートします。

注記

cpuacct.usage の値をリセットするには、以下のコマンドを実行します。

```
~]# echo 0 > /cgroup/cpuacct/cpuacct.usage
```

上記のコマンドは、**cpuacct.usage_percpu** の値もリセットします。

cpuacct.stat

この cgroup 内の全タスク (下位階層のタスクを含む) により消費されている CPU 時間を、以下のような形でユーザーとシステムにレポートします。

- **user** – ユーザーモード内のタスクによって消費されている CPU 時間
- **system** – システム (カーネル) モードのタスクによって消費されている CPU 時間

CPU 時間は、**USER_HZ** 変数によって定義されている単位でレポートされます。

cpuacct.usage_percpu

この cgroup 内の全タスク (下位階層のタスクを含む) により、各 CPU 上で消費される CPU 時間 (ナノ秒単位) をレポートします

3.4. CPUSET

cpuset サブシステムは、個別の CPU とメモリーノードを cgroup に割り当てます。各 cpuset は、以下のパラメーターにしたがって、それぞれを cgroup 仮想ファイルシステム内の別々の *疑似ファイル* 内で指定することができます。

重要

一部のサブシステムには、それらのいずれかを使用する cgroup にタスクを移動する前に設定しておく必要がある必須パラメーターがあります。たとえば、**cpuset** を使用する cgroup にタスクを cgroup に移動する前に、その cgroup に対して **cpuset.cpus** と **cpuset.mems** のパラメーターを定義しておく必要があります。

cpuset.cpus (必須)

cgroup 内のタスクがアクセスを許可される CPU を指定します。これは ASCII 形式のコンマ区切りの一覧で、ダッシュ (**"-"**) はその範囲を示します。以下はその例です。

```
0-2,16
```

これは、CPU 0、1、2、および 16 を示します。

cpuset.mems (必須)

この cgroup 内のタスクがアクセスを許可されるメモリーを指定します。これは ASCII 形式のコンマ区切りの一覧で、ダッシュ (**"-"**) は範囲を示します。以下はその例です。

```
0-2,16
```

これは、メモリーノード 0、1、2、および 16 を示します。

cpuset.memory_migrate

cpuset.mems 内の値が変更された場合に、メモリー内のページが新規ノードに移行すべきかどうかを指定するフラグ (**0** または **1**) が含まれます。デフォルトでは、メモリー移行は無効 (**0**) になっており、元のノードが **cpuset.mems** に指定されているノードの１つでなくなっても、ページは最初に割り当てられたノードに残ります。有効 (**1**) にすると場合、システムは **cpuset.mems** により指定された新規のパラメーター内のメモリーノードにページを移行します。また、可能な場合には、それらの相対的配置を維持します。たとえば、最初に **cpuset.mems** で指定されていた一覧の第 2 のノードにあるページは、この場所が使用可能な場合には、**cpuset.mems** で今回指定された一覧の第 2 のノードに割り当てられます。

cpuset.cpu_exclusive

他の cpuset がこの cpuset 用に指定された CPU を共有できるかどうかを指定するフラグ (**0** または **1**) が含まれます。デフォルト (**0**) では、CPU は １つの cpuset 専用には割り当てられません。

cpuset.mem_exclusive

この cpuset 用に指定されたメモリーノードを他の cpuset が共有できるかどうかを指定するフラグ (**0** または **1**) が含まれます。デフォルト (**0**) では、メモリーノードは１つの cpuset 専用には割り当てられません。cpuset 専用にメモリーノードを確保 (**1**) することは、**cpuset.mem_hardwall** パラメーターを使用してメモリーのハードウォールを有効にすることと機能的に同じです。

cpuset.mem_hardwall

メモリーページとバッファーデータのカーネル割り当てが、この cpuset 用に指定されたメモリーノードに制限されるべきかどうかを指定するフラグ (**0** または **1**) が含まれます。デフォルト (**0**) では、ページと バッファーデータは複数ユーザーに属するプロセス全体にわたって共有されます。ハードウォールが有効 (**1**) になっていると 各タスクのユーザー割り当ては別々に維持できます。

cpuset.memory_pressure

この cpuset 内のプロセスによって発生した *メモリー負荷*の累積平均を含む読み取り専用のファイルです。**cpuset.memory_pressure_enabled** が有効化されている場合、この疑似ファイル内の値は自動的に更新されます。そうでない場合、疑似ファイルには、値 **0** が含まれます。

cpuset.memory_pressure_enabled

この cgroup 内のプロセスによって発生した *メモリー負荷*をシステムが計算すべきかどうかを指定するフラグ (**0** または **1**) が含まれます。計算された値は **cpuset.memory_pressure** に出力されて、プロセスが使用中のメモリーの解放を試みるレートを示します。これは、1 秒あたりのメモリー

再生試行回数に 1000 を乗じた整数値としてレポートされます。

cpuset.memory_spread_page

この cpuset に割り当てられたメモリーノード全体にわたってファイルシステムバッファーを均等に分散すべきかどうかを指定するフラグ (**0** または **1**) が含まれます。デフォルト (**0**) では、これらのバッファー用にメモリーページを均等に分散しようとする試みはなく、バッファーはそれを作成したプロセスを実行しているのと同じノードに配置されます。

cpuset.memory_spread_slab

ファイルの入力/出力演算用のカーネルスラブキャッシュが cpuset 全体に均等に分散されるべきかどうかを指定するフラグ (**0** または **1**) が含まれます。デフォルト (**0**) では、カーネルスラブキャッシュを均等に分散しようとする試みはなく、スラブキャッシュはそれを作成したプロセスを実行しているのと同じノード上に配置されます。

cpuset.sched_load_balance

カーネルがこの cpuset 内の CPU 全体にわたって負荷を分散するかどうかを指定するフラグ (**0** または **1**) が含まれます。デフォルト (**1**) では、カーネルは過負荷状態の CPU から、使用頻度のより低い CPU へ プロセスを移動して負荷を分散します。

ただし、いずれかの親 cgroup でロードバランシングが有効化されている場合には、ロードバランシングがより高いレベルで既に実行されていることになるため、cgroup 内におけるこのフラグの設定は、無効となる点に注意してください。したがって、cgroup 内でロードバランシングを無効にするには、その階層内の各親 cgroup でもロードバランシングを無効にしてください。この場合には、対象となる cgroup の兄弟のロードバランシングも無効にすべきかどうかも検討すべきです。

cpuset.sched_relax_domain_level

-1 から小さい正の値までの間の整数が含まれます。これはカーネルが負荷を分散するために試行すべき CPU の範囲の幅を示します。**cpuset.sched_load_balance** が無効になっている場合には、この値は意味がありません。

この値の正確な効果はシステムアーキテクチャーに応じて変化しますが、以下の値が標準的です:

cpuset.sched_relax_domain_level の値

値	効果
-1	ロードバランシングにシステムデフォルト値を使用
0	ロードバランシングを即実行せず、定期的に負荷を分散
1	同じコア上のスレッド全体にわたって、ロードバランシングを即実行
2	同じパッケージ内のコア全体にわたって、ロードバランシングを即実行
3	同じノードまたはブレード上の CPU 全体にわたって、ロードバランシングを即実行

値	効果
4	NUMA (非均等メモリーアクセス) を使用するアーキテクチャー上のいくつかの CPU にわたって、ロードバランシングを即実行
5	NUMA を使用するアーキテクチャー上の全 CPU にわたって、ロードバランシングを即実行

3.5. DEVICES

devices サブシステムは、cgroup 内のタスクによるデバイスへのアクセスを許可または拒否します。

重要

Red Hat Enterprise Linux 6 では、Device Whitelist (**devices**) サブシステムはテクノロジープレビュー扱いとなります。

*テクノロジープレビュー*機能は、現在、Red Hat Enterprise Linux 6 サブスクリプションサービスではサポートされていません。機能的に完全でない可能性があり、通常は実稼働環境でのご使用には適切ではありませんが、Red Hat はお客様の便宜を図るために、これらの機能をオペレーティングシステムに組み込み、幅広く公開しています。これらの機能は、非実稼働環境で役立てていただくことができます。テクノロジープレビューの機能が完全にサポートされる前に、フィードバックや機能についてのご意見・ご希望をお気軽にお寄せください。

devices.allow

cgroup 内のタスクがアクセスをするデバイスを指定します。エントリは *type*、*major*、*minor*、および *access* の 4 つのフィールドで構成されます。*type*、*major*、および *minor* のフィールドに使用される値は、http://www.kernel.org/doc/Documentation/devices.txt に掲載の 『Linux Allocated Devices』 (別名 『Linux Devices List』) で指定されているデバイスタイプ、ノード番号に対応します。

type

type は以下の 3 つの値のいずれか 1 つを取ります。

- **a** – *文字デバイス* と *ブロックデバイス* の両方を併せた全デバイスに適用します
- **b** – ブロックデバイスを指定します
- **c** – 文字デバイスを指定します

major, minor

major と *minor* は、『Linux Allocated Devices』 で指定されているデバイスノード番号です。メジャー (major) とマイナー (minor) 番号はコロンで区切られます。たとえば、**8** は、SCSI ディスクドライブを指定するメジャー番号であり、マイナー番号 **1** は第 1 の SCSI ディスクドライブ上の第 1 のパーティションを指定します。したがって、**8:1** は、このパーティションを完全に指定し、**/dev/sda1** の ファイルシステムの場所に相当します。

***** は、すべてのメジャーまたはマイナーデバイスノードを表します。たとえば、**9:*** (全 RAID デバイス) または ***:*** (全デバイス) というように表示します。

access

access は、以下の文字 (単一または複数) からなるシーケンスです。

- **r** – タスクによる指定デバイスの読み取りを許可します
- **w** – タスクによる指定デバイスへの書き込みを許可します
- **m** – タスクによる、まだ存在していないデバイスファイルの作成を許可します

たとえば、*access* が **r** と指定されている時は、タスクは指定デバイスから読み取るだけですが、*access* が **rw** と指定されていると、タスクはデバイスからの読み取りとデバイスへの書き込みができます。

devices.deny

cgroup 内のタスクがアクセスできないデバイスを指定します。エントリの構文は **devices.allow** と全く同じです。

devices.list

この cgroup 内のタスクによるアクセス制御が設定されている対象デバイスをレポートします。

3.6. FREEZER

freezer サブシステムは cgroup 内のタスクを一時停止あるいは再開します。

freezer.state

freezer.state は root 以外の cgroup でのみ使用することができます。設定可能な値は以下の 3 つです。

- **FROZEN** – cgroup 内のタスクは一時停止しています。
- **FREEZING** – システムが cgroup 内のタスクを一時停止している最中です。
- **THAWED** – cgroup 内のタスクが再開しています。

特定のプロセスを一時停止するには、

1. **freezer** サブシステムが接続された階層内の cgroup にそのプロセスを移動します。
2. 特定の cgroup をフリーズさせて、その中に含まれるプロセスを一時停止します。

一時停止 (フリーズ) した cgroup にプロセスを移動することはできません。

FROZEN と **THAWED** の値は **freezer.state** に書き込むことができますが、**FREEZING** の書き込みは出来ず、読み取りのみが可能である点に注意してください。

3.7. MEMORY

memory サブシステムは、**cgroup** 内のタスクによって使用されるメモリーリソースの自動レポートを生成し、他のタスクによるメモリー使用の上限を設定します。

memory.stat

以下の表に記載した、広範囲なメモリーの統計をレポートします。

表3.2 memory.stat によりレポートされる値

統計	説明
cache	**tmpfs** (**shmem**) を含むページキャッシュ (バイト単位)
rss	**tmpfs** (**shmem**) を**含まない**匿名のスワップキャッシュ (バイト単位)
mapped_file	**tmpfs** (**shmem**) を含むメモリーマップドファイルのサイズ (バイト単位)
pgpgin	メモリー内へページされたページ数
pgpgout	メモリーからページアウトされたページ数
swap	スワップの使用量 (バイト単位)
active_anon	**tmpfs** (**shmem**) を含む、アクティブな最長時間未使用 (LRU) 一覧上の匿名のスワップキャッシュ (バイト単位)
inactive_anon	**tmpfs** (**shmem**) を含む、非アクティブ LRU 一覧上の匿名のスワップキャッシュ (バイト単位)
active_file	アクティブ LRU 一覧にある、ファイルと関連づけされたメモリー (バイト単位)
inactive_file	非アクティブ LRU 一覧にある、ファイルに関連付けされたメモリー (バイト)
unevictable	再生不可のメモリー (バイト単位)
hierarchical_memory_limit	**memory** cgroup が含まれる階層のメモリー制限 (バイト単位)
hierarchical_memsw_limit	**memory** cgroup が含まれる階層のメモリーとスワップの制限 (バイト単位)

また、これらのファイルの中で、**hierarchical_memory_limit** および **hierarchical_memsw_limit** 以外のファイルには、それぞれ、**total_** というプレフィックスの付いた対応ファイルがあり、**cgroup** についてだけでなく、その子グループについてもレポートします。たとえば、**swap** は **cgroup** によるスワップの使用量をレポートし、**total_swap** は **cgroup** とその子グループによるスワップの使用量をレポートします。

memory.stat によってレポートされた値を解析する際には、さまざま統計が相互に関連している点に注意してください。

- **active_anon** + **inactive_anon** = 匿名メモリー + **tmpfs** のファイルキャッシュ + スワップキャッシュ

  したがって、**active_anon** + **inactive_anon** ≠ **rss** となります。これは、**rss** に **tmpfs** が含まれないのが理由です。

- **active_file** + **inactive_file** = cache - size of **tmpfs**

memory.usage_in_bytes

cgroup 内のプロセスによる現在のメモリー総使用量をレポートします (バイト単位)。

memory.memsw.usage_in_bytes

cgroup 内のプロセスによる現在のメモリー使用量と使用済みスワップ領域の和をレポートします (バイト単位)。

memory.max_usage_in_bytes

cgroup 内のプロセスによるメモリー最大使用量をレポートします (バイト単位)。

memory.memsw.max_usage_in_bytes

cgroup 内のプロセスによるスワップメモリー最大使用量と使用済みスワップ領域をレポートします (バイト単位)。

memory.limit_in_bytes

ユーザーメモリーの最大値 (ファイルキャッシュを含む) を設定します。単位が指定されていない場合、その値はバイト単位と解釈されますが、より大きな単位を示すサフィックスを使用することが可能です (キロバイトには **k** または **K**、メガバイトには **m** または **M**、ギガバイトには **g** または **G**)。

root cgroup を制限するのには、**memory.limit_in_bytes** は使用できません。値を適用できるのは、下位階層のグループに対してのみです。

memory.limit_in_bytes に **-1** と書き込み、現行の制限値を削除します。

memory.memsw.limit_in_bytes

メモリーとスワップ使用量の合計の最大値を設定します。単位が指定されていない場合、その値はバイト単位と解釈されますが、より大きな単位を示すサフィックスを使用することが可能です (キロバイトには **k** または **K**、メガバイトには **m** または **M**、ギガバイトには **g** または **G**)。

root cgroup を制限するのに、**memory.memsw.limit_in_bytes** は使用できません。値を適用できるのは、下位階層のグループに対してのみです。

memory.memsw.limit_in_bytes に **-1** と書き込み、現行の制限値を削除します。

重要

memory.limit_in_bytes パラメーターは、**memory.memsw.limit_in_bytes** を**設定する前に**設定しておくことが重要となります。逆の順序で設定を試みると、エラーが発生します。これは、**memory.memsw.limit_in_bytes** を使用できるようになるのが、(**memory.limit_in_bytes** で事前に設定されている) メモリー制限をすべて使い切った後のみであるためです。

次の例を検討してください: 特定の cgroup に対して **memory.limit_in_bytes = 2G** と **memory.memsw.limit_in_bytes = 4G** と設定すると、cgroup 内のプロセスが 2 GB のメモリーを割り当てることが可能となり、それを使い果たすと、さらに 2 GB のスワップのみを割り当てます。**memory.memsw.limit_in_bytes** パラメーターはメモリーとスワップの合計を示しています。**memory.memsw.limit_in_bytes** パラメーターが設定されていない cgroup 内のプロセスは、(設定されているメモリーの上限を消費した後に) 使用可能なスワップをすべて使い果たしてしまい、空きスワップがなくなるために Out Of Memory (OOM) の状態を引き起こす可能性があります。

また、**/etc/cgconfig.conf** ファイルで **memory.limit_in_bytes** と **memory.memsw.limit_in_bytes** のパラメーターを設定する順序も重要です。この設定の正しい例は以下のとおりです。

```
memory {
    memory.limit_in_bytes = 1G;
    memory.memsw.limit_in_bytes = 1G;
}
```

memory.failcnt

memory.limit_in_bytes に設定されているメモリーの上限値に達した回数をレポートします。

memory.memsw.failcnt

memory.memsw.limit_in_bytes に設定されているメモリーとスワップ領域の合計が上限に達した回数をレポートします。

memory.force_empty

0 に設定されている場合には、cgroup 内のタスクによって使用される全ページのメモリーを空にします。このインターフェイスは、cgroup がタスクを持たない時にのみ使用できます。メモリーを解放できない場合は、可能ならば 親 cgroup に移動されます。cgroup を削除する前には、**memory.force_empty** を使用して、未使用のページキャッシュが親 cgroup に移動されないようにしてください。

memory.swappiness

ページキャッシュからページを再生する代わりに、カーネルがこの cgroup 内のタスクで使用されるプロセスメモリーをスワップアウトする傾向を設定します。これはシステム全体用に **/proc/sys/vm/swappiness** 内に設定されているのと同じ傾向で、同じ方法で算出されます。デフォルト値は **60** です。これより低い値を設定すると、カーネルがプロセスメモリーをスワップアウトする傾向が低減します。また **100** 以上に設定すると、カーネルはこの cgroup 内のプロセスのアドレス領域となる部分のページをスワップアウトできるようになります。

0 の値に設定しても、プロセスメモリーがスワップアウトされるのを防ぐことはできない点に注意してください。グローバル仮想メモリー管理ロジックは、cgroup の値を読み取らないため、システ

ムメモリーが不足した場合に、依然としてスワップアウトが発生する可能性があります。ページを完全にロックするには、cgroup の代わりに **mlock()** を使用してください。

以下にあげるグループの swappiness は変更できません。

- **/proc/sys/vm/swappiness** に設定された swappiness を使用している root cgroup
- 配下に子グループがある cgroup

memory.use_hierarchy

cgroup の階層全体にわたって、メモリー使用量を算出すべきかどうかを指定するフラグ (**0** または **1**) が含まれます。有効 (**1**) となっている場合、メモリーサブシステムはメモリーの上限を超過しているプロセスとその子プロセスからメモリーを再生します。デフォルト (**0**) では、サブシステムはタスクの子からメモリーを再生しません。

memory.oom_control

cgroup に対して Out of Memory Killer を有効化/無効化するフラグ (**0** または **1**) が含まれています。これを有効にすると (**0**)、許容量を超えるメモリーを使用しようとするタスクは OOM Killer によって即時に強制終了されます。OOM Killer は、**memory** サブシステムを使用するすべての cgroup でデフォルトで有効になっています。これを無効にするには、**memory.oom_control** ファイルに **1** と記載します。

```
~]# echo 1 > /cgroup/memory/lab1/memory.oom_control
```

OOM Killer が無効になると、許容量を超えるメモリーを使用しようとするタスクは、追加のメモリーが解放されるまで一時停止されます。

memory.oom_control ファイルは、現在の cgroup の OOM ステータスも **under_oom** エントリにレポートします。cgroup がメモリー不足の状態で、その cgroup 内のタスクが一時停止されている場合には、**under_oom** エントリで値が **1** とレポートされます。

memory.oom_control ファイルは、通知 API を使用して OOM 状態の発生をレポートすることができます。詳しくは、「通知 API の使用」 および 例3.3「OOM の制御と通知」 を参照してください。

3.7.1. 使用例

例3.3 OOM の制御と通知

以下の例は、cgroup 内のタスクが許容量を超えるメモリーの使用を試みた場合に OOM Killer がどのように対応し、通知ハンドラーが OOM 状態のどのようにレポートするかを示した実例です。

1. **memory** サブシステムを階層に接続し、cgroup を作成します。

```
~]# mount -t memory -o memory memory /cgroup/memory
~]# mkdir /cgroup/memory/blue
```

2. **blue** cgroup 内のタスクが使用できるメモリーを 100 MB に設定します。

```
~]# echo 104857600 > memory.limit_in_bytes
```

3. **blue** ディレクトリに移動して、OOM Killer が有効になっていることを確認します。

```
~]# cd /cgroup/memory/blue
blue]# cat memory.oom_control
oom_kill_disable 0
under_oom 0
```

4. 現在のシェルプロセスを **blue** cgroup の **tasks** ファイルに移動し、このシェルで起動したその他すべてのプロセスが自動的に **blue** cgroup に移動するようにします。

```
blue]# echo $$ > tasks
```

5. ステップ 2 で設定した上限を超える大容量のメモリーを割り当てようとするテストプログラムを起動します。**blue** cgroup の空きメモリーがなくなるとすぐに OOM Killer がテストプログラムを強制終了し、標準出力に **Killed** をレポートします。

```
blue]# ~/mem-hog
Killed
```

以下は、このようなテストプログラムの一例です。[5]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define KB (1024)
#define MB (1024 * KB)
#define GB (1024 * MB)

int main(int argc, char *argv[])
{
 char *p;

again:
 while ((p = (char *)malloc(GB)))
  memset(p, 0, GB);

 while ((p = (char *)malloc(MB)))
  memset(p, 0, MB);

 while ((p = (char *)malloc(KB)))
  memset(p, 0,
    KB);

 sleep(1);

 goto again;

 return 0;
}
```

6. OOM Killer を無効にし、テストプログラムを再度実行します。今回は、テストプログラムが一時停止の状態のままとなり、追加のメモリーが解放されるのを待機します。

```
blue]# echo 1 > memory.oom_control
blue]# ~/mem-hog
```

7. テストプログラムが一時停止されている間は、**cgroup** の **under_oom** 状態が変わり、空きメモリーが不足していることを示している点に注意してください。

```
~]# cat /cgroup/memory/blue/memory.oom_control
oom_kill_disable 1
under_oom 1
```

OOM Killer を再度有効にすると、テストプログラムは即時に強制終了されます。

8. すべての OOM 状態についての通知を受信するためには、「通知 API の使用」 に記載したようにプログラムを作成してください。以下はその一例です[6]。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/eventfd.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

static inline void die(const char *msg)
{
 fprintf(stderr, "error: %s: %s(%d)\n", msg, strerror(errno),
errno);
 exit(EXIT_FAILURE);
}

static inline void usage(void)
{
 fprintf(stderr, "usage: oom_eventfd_test <cgroup.event_control>
<memory.oom_control>\n");
 exit(EXIT_FAILURE);
}

#define BUFSIZE 256

int main(int argc, char *argv[])
{
 char buf[BUFSIZE];
 int efd, cfd, ofd, rb, wb;
 uint64_t u;

 if (argc != 3)
  usage();

 if ((efd = eventfd(0, 0)) == -1)
  die("eventfd");

 if ((cfd = open(argv[1], O_WRONLY)) == -1)
  die("cgroup.event_control");
```

```
 if ((ofd = open(argv[2], O_RDONLY)) == -1)
  die("memory.oom_control");

 if ((wb = snprintf(buf, BUFSIZE, "%d %d", efd, ofd)) >= BUFSIZE)
  die("buffer too small");

 if (write(cfd, buf, wb) == -1)
  die("write cgroup.event_control");

 if (close(cfd) == -1)
  die("close cgroup.event_control");

 for (;;) {
  if (read(efd, &u, sizeof(uint64_t)) != sizeof(uint64_t))
   die("read eventfd");

  printf("mem_cgroup oom event received\n");
 }

 return 0;
}
```

上記のプログラムはコマンドラインの引数として指定された cgroup 内の OOM 状態を検出し、**mem_cgroup oom event received** の文字列を使用して標準出力にレポートします。

9. **blue** cgroup の制御ファイルを引数として指定して、上記の通知ハンドラープログラムを別のコンソールで実行します。

```
~]$ ./oom_notification /cgroup/memory/blue/cgroup.event_control
/cgroup/memory/blue/memory.oom_control
```

10. 別のコンソールで **mem_hog** テストプログラムを実行し、OOM 状態を発生させて **oom_notification** プログラムがそれを標準出力にレポートするのを確認します。

```
blue]# ~/mem-hog
```

3.8. NET_CLS

net_cls サブシステムは、Linux トラフィックコントローラー (**tc**) が特定の cgroup から発信されるパケットを識別できるようにするクラス識別子 (classid) を使用して、ネットワークパケットをタグ付けします。トラフィックコントローラーは、異なる cgroup からのパケットに異なる優先順位を割り当てるように設定できます。

net_cls.classid

net_cls.classid には、トラフィック制御*ハンドル*を示す単一値が含まれます。**net_cls.classid** ファイルから読み取られる ***classid*** の値は、10 進数形式で表示されますが、ファイルに書き込まれる値は 16 進数形式となります。たとえば、**0x100001** は、iproute2 で使用されている形式では従来 **10:1** として記述されていたハンドルを示します。**net_cls.classid** ファイルでは、**1048577** の数字で示されます。

これらのハンドルの形式は: **0x*AAAABBBB*** です。ここで *AAAA* は 16 進法のメジャー番号、*BBBB* は

16 進法のマイナー番号です。また、先頭のゼロを省略することができ、**0x10001** は **0x00010001** と同じで、**1:1** を示します。以下は、**net_cls.classid** ファイルでの **10:1** ハンドルの設定例です。

```
~]# echo 0x100001 > /cgroup/net_cls/red/net_cls.classid
~]# cat /cgroup/net_cls/red/net_cls.classid
1048577
```

net_cls がネットワークパケットに追加するハンドルを使用するためのトラフィックコントローラーの設定法を確認するには **tc** の man ページを参照してください。

3.9. NET_PRIO

ネットワーク優先度 (**net_prio**) サブシステムは、さまざまな **cgroup** 内でアプリケーション用の各ネットワークインターフェイス毎にネットワークトラフィックの優先度を動的に設定する方法を提供します。ネットワーク優先度はネットワークトラフィックに割り当てられる番号で、システムおよびネットワークデバイスにより内部で使用されます。ネットワーク優先度は、送信、キューに配置、またはドロップされるパケットを区別するために使用されます。**tc** コマンドは、ネットワーク優先度の設定に使用することができます (**tc** コマンドを使用したネットワーク優先度の設定は本ガイドのスコープ外です。詳しくは **tc** man ページを参照してください)。

通常アプリケーションは、**SO_PRIORITY** ソケットオプションによりトラフィックの優先度を設定しますが、アプリケーションが優先度の値を設定するようにコードが書かれていなかったり、アプリケーションのトラフィックがサイト固有で定義された優先度を提供しない場合が多くあります。

cgroup 内で **net_prio** サブシステムを使用すると、管理者はプロセスを特定の **cgroup** に割り当てて、任意のネットワーク上の送信トラフィックの優先度を定義することができます。

net_prio.prioidx

この **cgroup** の内部表現としてカーネルが使用する、一意の整数値を含む読み取り専用ファイル。

net_prio.ifpriomap

このグループ内のプロセスが送信元となっているトラフィック、およびさまざまなインターフェースでシステムから外に送信されるトラフィックに割り当てられた優先度のマップを含みます。このマップは、***<network_interface> <priority>*** の形式でペアで示されます

```
~]# cat /cgroup/net_prio/iscsi/net_prio.ifpriomap
eth0 5
eth1 4
eth2 6
```

net_prio.ifpriomap ファイルの内容は、上記の形式を使用して、文字列を **echo** コマンドでファイルに書き込むことによって変更することができます。以下はその例です。

```
~]# echo "eth0 5" > /cgroup/net_prio/iscsi/net_prio.ifpriomap
```

上記のコマンドは、**iscsi net_prio cgroup** に属するプロセスから送信されるトラフィック、および **eth0** ネットワークインターフェース上で送信されるトラフィックで優先度の値が**5** に設定されるよう強制します。親 **cgroup** には、システムのデフォルト優先度を設定するのに使用できる書き込み可能な **net_prio.ifpriomap** ファイルもあります。

3.10. NS

ns サブシステムは、プロセスを異なる*名前空間*にグループ化する手段を提供します。特定の名前空間内では、プロセス間における相互の対話が可能ですが、他の名前空間で実行されているプロセスからは分離しています。このように分離した名前空間は、オペレーティングシステムレベルの仮想化に使用される場合には、*コンテナー*とも呼ばれています。

3.11. PERF_EVENT

perf_event サブシステムが階層に接続されると、その階層内の全 cgroups は、グループプロセスおよびスレッドに使用できることが可能となり、プロセス/スレッド別または CPU 単位ではなく、**perf** ツールでモニタリングできるようになります。**perf_event** サブシステムを使用する cgroup には、「共通の調整可能なパラメーター」に記載されている共通のパラメーター以外の特殊な調整可能パラメーターは含まれません。

perf ツールを使用した cgroup 内のタスクのモニタリング方法に関する詳しい情報は、http://access.redhat.com/knowledge/docs/Red_Hat_Enterprise_Linux/ の Red Hat Enterprise Linux 『Developer Guide』 を参照してください。

3.12. 共通の調整可能なパラメーター

以下のパラメーターは、使用するサブシステムを問わず、作成されたすべての cgroup に存在します。

tasks

cgroup で実行中のプロセスの一覧が含まれ、PID で表示されます。PID の一覧が順位付けされていることや、一意であることは保証されません (つまり、重複したエントリが含まれている可能性があります)。PID を cgroup の **tasks** ファイルに書き込むと、そのプロセスは cgroup に移動します。

cgroup.procs

cgroup で実行中のスレッドグループの一覧が含まれ、TGID で表示されます。TGID の一覧が順位付けされていることや、一意であることは保証されません (つまり、重複したエントリが含まれている可能性があります)。TGID を cgroup の **tasks** ファイルに書き込むと、そのスレッドグループは cgroup に移動します。

cgroup.event_control

cgroup 通知 API とともに、cgroup のステータス変更についての通知を送付できるようにします。

notify_on_release

ブール値が含まれ、**1** または **0** でリリースエージェントの実行を有効化または無効化します。***notify_on_release*** が有効化されると、cgroup にタスクがなくなった時にカーネルが **release_agent** ファイルの内容を実行します (つまり、cgroup の **tasks** ファイルにいくつかの PID が含まれ、それらの PID が削除されてファイルが空の状態となっています)。空の cgroup へのパスは、空の cgroup へのパスは、引数としてリリースエージェントに提供されます。

root cgroup 内の ***notify_on_release*** パラメーターのデフォルト値は **0** です。root 以外の cgroups はすべて、親 cgroup から***notify_on_release*** 内の値を継承します。

release_agent (root cgroup のみに存在)

「notify on release」 がトリガーされた時に実行されるコマンドが含まれます。cgroup の全プロセスが空となると、***notify_on_release*** フラグが有効化され、カーネルが**release_agent** ファイル内のコマンドを実行して、相対パス(root cgroup に相対) で空の cgroup に引数として提供しま

す。リリースエージェントは、たとえば、空の **cgroup** を自動的に削除するのに使用することができます。詳しくは、例3.4「空の cgroup の自動削除」 を参照してください。

例3.4 空の cgroup の自動削除

以下の手順にしたがって、空になった **cgroup** を自動的に **cpu cgroup** から削除するように設定します。

1. 空の **cpu cgroups** を削除するシェルスクリプトを作成して **/usr/local/bin** などに配置し、実行できるようにします。

```
~]# cat /usr/local/bin/remove-empty-cpu-cgroup.sh
#!/bin/sh
rmdir /cgroup/cpu/$1
~]# chmod +x /usr/local/bin/remove-empty-cpu-cgroup.sh
```

   $1 の変数には空になった **cgroup** への相対パスを記載します。

2. **cpu cgroup** で ***notify_on_release*** フラグを有効にします。

```
~]# echo 1 > /cgroup/cpu/notify_on_release
```

3. **cpu cgroup** には、使用するリリースエージェントを指定します。

```
~]# echo "/usr/local/bin/remove-empty-cpu-cgroup.sh" >
/cgroup/cpu/release_agent
```

4. 設定をテストして、空になった **cgroup** が適切に削除されることを確認します。

```
cpu]# pwd; ls
/cgroup/cpu
cgroup.event_control  cgroup.procs  cpu.cfs_period_us
cpu.cfs_quota_us  cpu.rt_period_us  cpu.rt_runtime_us
cpu.shares  cpu.stat  libvirt  notify_on_release  release_agent
tasks
cpu]# cat notify_on_release
1
cpu]# cat release_agent
/usr/local/bin/remove-empty-cpu-cgroup.sh
cpu]# mkdir blue; ls
blue  cgroup.event_control  cgroup.procs  cpu.cfs_period_us
cpu.cfs_quota_us  cpu.rt_period_us  cpu.rt_runtime_us
cpu.shares  cpu.stat  libvirt  notify_on_release  release_agent
tasks
cpu]# cat blue/notify_on_release
1
cpu]# cgexec -g cpu:blue dd if=/dev/zero of=/dev/null bs=1024k
&
[1] 8623
cpu]# cat blue/tasks
8623
cpu]# kill -9 8623
cpu]# ls
cgroup.event_control  cgroup.procs  cpu.cfs_period_us
```

```
cpu.cfs_quota_us  cpu.rt_period_us  cpu.rt_runtime_us
cpu.shares  cpu.stat  libvirt  notify_on_release  release_agent
tasks
```

3.13. その他のリソース

サブシステム固有のカーネルのドキュメント

以下のファイルはすべて **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/cgroups/** ディレクトリ下に配置されています (kernel-doc パッケージにより提供)。

- **blkio** サブシステム – **blkio-controller.txt**
- **cpuacct** サブシステム – **cpuacct.txt**
- **cpuset** サブシステム – **cpusets.txt**
- **devices** サブシステム – **devices.txt**
- **freezer** サブシステム – **freezer-subsystem.txt**
- **memory** サブシステム – **memory.txt**
- **net_prio** サブシステム – **net_prio.txt**

また、**cpu** サブシステムについての詳しい情報は、以下のファイルを参照してください。

- リアルタイムスケジューリング – **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-rt-group.txt**
- CFS スケジューリング – **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-bwc.txt**

---

[5] ソースコード提供: Red Hat のエンジニア František Hrbata 氏

[6] ソースコード提供: Red Hat のエンジニア František Hrbata 氏

第4章 ユースケースシナリオ

本章には、cgroup の機能性を活用したユースケースシナリオを記載します。

4.1. データベース I/O の優先

独自の専用仮想ゲスト内でデータベースサーバーの各インスタンスを実行することにより、優先度に基づいてデータベースごとにリソースを割り当てることができます。次の例を検討してください: システムが 2 台の KVM ゲスト内で 2 つのデータベースを実行しています。一方のデータベースは優先度が高く、もう一方は優先度の低いデータベースです。両方のデータベースサーバーが同時に稼働すると、I/O スループットが低減し、両データベースからの要求に同等に対応します。図4.1「リソース割り当てを使用しない I/O スループット」 は、このシナリオを示しています。– 優先度の低いデータベースが起動されると (時間軸 45 前後)、I/O スループットが両データベースサーバーで同じとなっています。

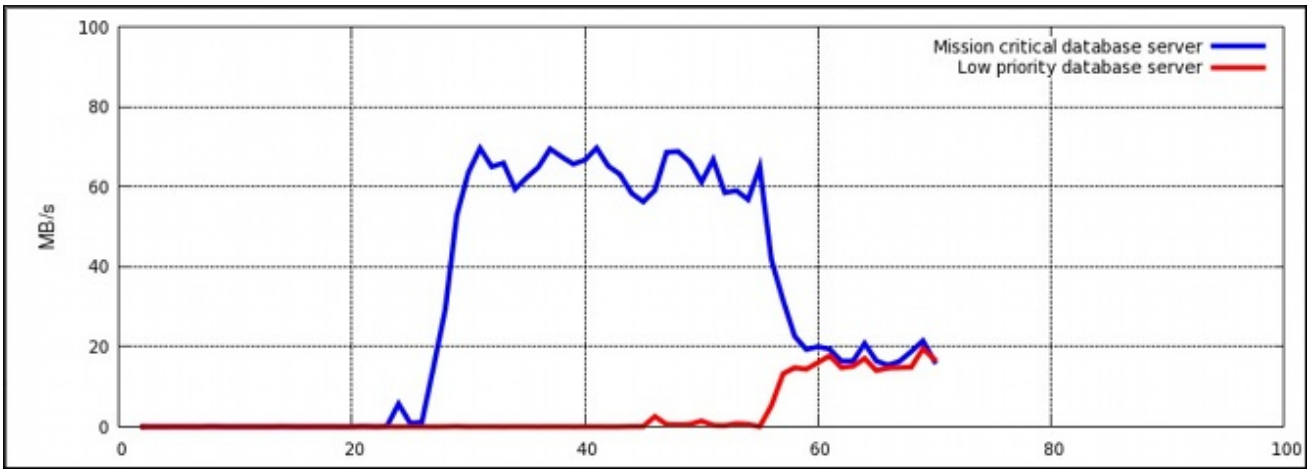


図4.1 リソース割り当てを使用しない I/O スループット

優先度の高いデータベースサーバーを優先するには、予約済みの I/O 操作の高い数値を cgroup に割り当てる一方、優先度低いデータベースサーバーには予約済み I/O 操作の低い数値を cgroup に割り当てます。この設定は手順4.1「I/O スループットの優先度設定」 の手順にしたがって行います。作業はすべてホストシステム上で実行します。

手順4.1 I/O スループットの優先度設定

1. **blkio** サブシステムを **/cgroup/blkio** cgroup に接続します。

```
~]# mkdir /cgroup/blkio
~]# mount -t cgroup -o blkio blkio /cgroup/blkio
```

2. 優先度の高い cgroup と低い cgroup を作成します。

```
~]# mkdir /cgroup/blkio/high_prio
~]# mkdir /cgroup/blkio/low_prio
```

3. 両仮想ゲスト (データベースサーバーを実行している) を示す PID を取得し、それら固有の cgroup に移動します。この例では、 **VM_high** は優先度の高いデータベースサーバーを実行している仮想ゲストを示し、**VM_low** は優先度の低いデータベースサーバーを実行している仮想ゲストを示しています。以下はその例です。

```
~]# ps -eLf | grep qemu | grep VM_high | awk '{print $4}' | while
read pid; do echo $pid >> /cgroup/blkio/high_prio/tasks; done
```

```
~]# ps -eLf | grep qemu | grep VM_low | awk '{print $4}' | while
read pid; do echo $pid >> /cgroup/blkio/low_prio/tasks; done
```

4. **high_prio** cgroup と **low_prio** cgroup の比を 10:1 に設定します。それらの cgroup 内のプロセス (前のステップでそれらの cgroup に追加した仮想ゲストを実行しているプロセス) は、それらのプロセスが利用可能なリソースのみを即時に使用します。

```
~]# echo 1000 > /cgroup/blkio/high_prio/blkio.weight
~]# echo 100 > /cgroup/blkio/low_prio/blkio.weight
```

この例で、優先度の低い cgroup は、優先度の低いデータベースサーバーが約 10 % の I/O 操作を使用するのを許可する一方、優先度の高い cgroup は、優先度の高いデータベースサーバーが約 90 % の I/O 操作を使用するのを許可します。

図4.2「I/O スループットとリソース割り当て」は、優先度の低いデータベースを制限し、優先度の高いデータベースを優先した結果を図示しています。データベースサーバーが適切な cgroup に移動されると (時間軸 75 前後) 即時に I/O スループットが 10:1 の比率で両サーバー間で分配されます。

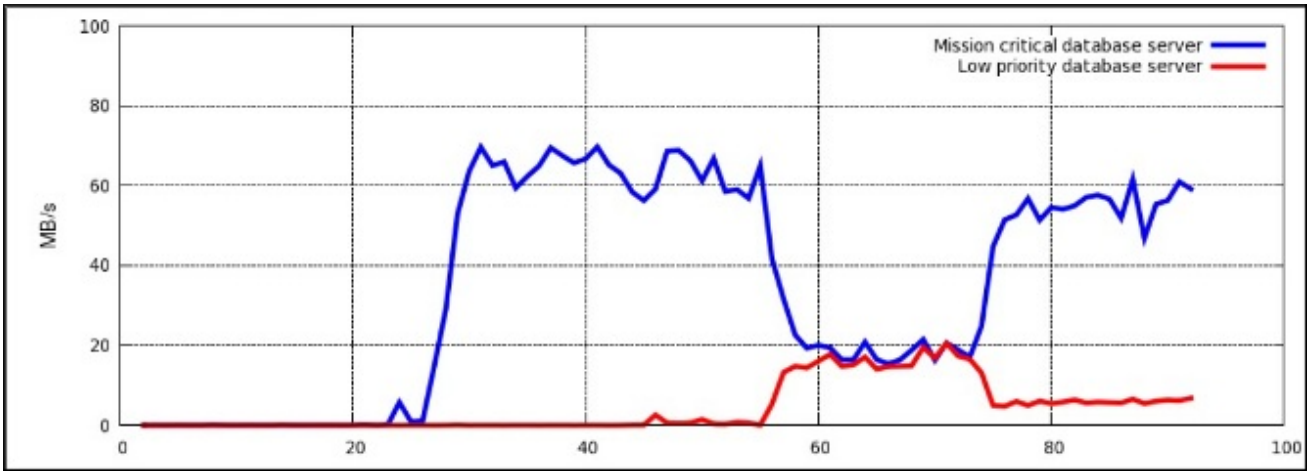


図4.2 I/O スループットとリソース割り当て

あるいは、ブロックデバイス I/O スロットリングを使用して、優先度の低いデータベースの読み取り/書き込み操作を制限することができます。**blkio** サブシステムに関するさらに詳しい情報は、「blkio」 を参照してください。

4.2. ネットワークトラフィックの優先度設定

単一のサーバーシステムでネットワーク関連サービスを複数実行している場合には、それらのサービス間におけるネットワークの優先度を定義することが重要です。これらの優先度を定義することにより、特定のサーバーから発信されるパッケージの優先度を、その他のサービスから発信されるパッケージよりも優先度を高くすることができます。たとえば、そのような優先度は、サーバーシステムが同時に NFS および Samba サーバーとして機能する場合に役立ちます。NFS のトラフィックは、ユーザーが高スループットを期待するので、優先度を高くする必要があります。Samba のトラフィックは、NFS サーバーのパフォーマンスを向上させるために、優先度を低くすることができます。

net_prio サブシステムを使用して、cgroup 内のプロセスの優先順位を設定することができます。次に、これらの優先度が Type Of Service (TOS) ビットに変換され、各パケットに埋め込まれます。二つのファイル共有サービス (NFS と Samba) の優先度を設定するには、手順4.2「ファイル共有サービスのネットワーク優先度の設定」 の手順にしたがってください。

手順4.2 ファイル共有サービスのネットワーク優先度の設定

1. **net_prio** サブシステムを **/cgroup/net_prio** cgroup に接続します。

```
~]# mkdir /cgroup/net_prio
~]# mount -t cgroup -o net_prio net_prio /cgroup/net_prio
```

2. サービスごとに 2 つの **cgroup** を作成します。

```
~]# mkdir /cgroup/net_prio/nfs_high
~]# mkdir /cgroup/net_prio/samba_low
```

3. **nfs_high** cgroup に **nfs** を自動的に移動するには、**/etc/sysconfig/nfs** ファイルに以下の行を追加します。

```
CGROUP_DAEMON="net_prio:nfs_high"
```

この設定は、**nfs** サービスが起動または再起動された時に、**nfs** サービスプロセスが **nfs_high** cgroup に移動するようにします。**cgroup** への サービスプロセス移動についての詳細は、「コントロールグループ内のサービスの開始」を参照してください。

4. **smbd** デーモンの設定ファイルは **/etc/sysconfig** ディレクトリにはありません。**smbd** デーモンを **samba_low** cgroup に自動的に移動するには、**/etc/cgrules.conf** ファイルに以下の行を追記してください。

```
*:smbd                  net_prio                samba_low
```

このルールにより、**/usr/sbin/smbd** のみではなく、すべての **smbd** デーモンが **samba_low** cgroup に移動する点に注意してください。

同様に、**nmbd** および **winbindd** デーモンを **samba_low** cgroup に移動させるルールを定義することができます。

5. **cgred** サービスを起動して、前の手順からの設定を読み込みます。

```
~]# service cgred start
Starting CGroup Rules Engine Daemon:                       [  OK  ]
```

6. この例では、両サービスが **eth1** ネットワークインターフェースを使用していることを前提とします。各 **cgroup** にネットワークの優先度を定義します。ここで **1** 優先度低く、**10** は優先度が高い数値を示します。

```
~]# echo "eth1 1" > /cgroup/net_prio/samba_low
~]# echo "eth1 10" > /cgroup/net_prio/nfs_high
```

7. **nfs** および **smb** のサービスを起動し、それらのプロセスが正しい **cgroup** に移動したことを確認します。

```
~]# service smb start
Starting SMB services:                                     [  OK
]
~]# cat /cgroup/net_prio/samba_low
16122
16124
```

```
~]# service nfs start
Starting NFS services:                                     [  OK
]
Starting NFS quotas:                                       [  OK
]
Starting NFS mountd:                                       [  OK
]
Stopping RPC idmapd:                                       [  OK
]
Starting RPC idmapd:                                       [  OK
]
Starting NFS daemon:                                       [  OK
]
~]# cat /cgroup/net_prio/nfs_high
16321
16325
16376
```

NFS から発信されるネットワークトラフィックの優先度が、Samba から発信されるトラフィックよりも高くなりました。

手順4.2「ファイル共有サービスのネットワーク優先度の設定」と同様に、**net_prio** サブシステムはクライアントアプリケーション (例: Firefox) のネットワーク優先度設定に使用することができます。

4.3. CPU およびメモリーリソースのグループ別配分

多数のユーザーが単一のシステムを使用する場合、特定のユーザーにより多くのリソースを提供すると役立ちます。次の例を検討してください: ある会社で、finance (財務)、sales (営業)、engineering (エンジニアリング) の 3 つの部署があるとします 。エンジニアは、他の部署よりもシステムとそのリソースを多く使用するので、全部署で CPU とメモリーを集中的に使用するタスクを実行する場合に、エンジニアにより多くのリソースを提供するのが当然です。

cgroups は、システムユーザーグループ別にリソースを制限する手段を提供します。この例では、システム上で以下のユーザーを作成済みであることを前提とします。

```
~]$ grep home /etc/passwd
martin:x:500:500::/home/martin:/bin/bash
john:x:501:501::/home/john:/bin/bash
mark:x:502:502::/home/mark:/bin/bash
peter:x:503:503::/home/peter:/bin/bash
jenn:x:504:504::/home/jenn:/bin/bash
mike:x:505:505::/home/mike:/bin/bash
```

これらのユーザーは、次のシステムグループに割り当てられています。

```
~]$ grep -e "50[678]" /etc/group
finance:x:506:jenn,john
sales:x:507:mark,martin
engineering:x:508:peter,mike
```

この例が適切に機能するには、**libcgroup** パッケージがインストール済みである必要があります。**/etc/cgconfig.conf** および **/etc/cgrules.conf** のファイルを使用して階層を作成し、各ユーザー用のリソースの量を決定するルールを設定することができます。この設定は、手順4.3「グループ別の CPU およびメモリーリソースの管理」 に記載した手順にしたがって行ってください。

手順4.3 グループ別の CPU およびメモリーリソースの管理

1. **/etc/cgconfig.conf** ファイルで、以下のようなサブシステムをマウントして、**cgroup** を作成するように設定します。

```
mount {
    cpu     = /cgroup/cpu_and_mem;
    cpuacct = /cgroup/cpu_and_mem;
    memory  = /cgroup/cpu_and_mem;
}

group finance {
        cpu {
                cpu.shares="250";
        }
        cpuacct {
                cpuacct.usage="0";
        }
        memory {
                memory.limit_in_bytes="2G";
                memory.memsw.limit_in_bytes="3G";
        }
}

group sales {
        cpu {
                cpu.shares="250";
        }
        cpuacct {
                cpuacct.usage="0";
        }
        memory {
                memory.limit_in_bytes="4G";
                memory.memsw.limit_in_bytes="6G";
        }
}

group engineering {
        cpu {
                cpu.shares="500";
        }
        cpuacct {
                cpuacct.usage="0";
        }
        memory {
                memory.limit_in_bytes="8G";
                memory.memsw.limit_in_bytes="16G";
        }
}
```

上記の設定ファイルが読み込まれると、**cpu**、**cpuacct**、および **memory** のサブシステムが単一の **cpu_and_mem** cgroup にマウントされます。これらのサブシステムについての詳しい情報は、3章 *サブシステムと調整可能なパラメーター*を参照してください。次に **cpu_and_mem** に階層が作成されます。これには、**sales**、**finance**、**engineering** の 3 つの **cgroup** が含まれます。これらの **cgroup** にはそれぞれ、各サブシステムに対するカスタムパラメーターが設定されます。

- **cpu** – ***cpu.shares*** パラメーターは、全 cgroup 内の各プロセスに提供する CPU リソースの配分を決定します。このパラメーターを finance cgroup に **250**、sales cgroup に **250**、engineering cgroup に **500** と設定すると、これらのグループで起動されたプロセスはリソースを 1:1:2 の割合で分割することになります。実行されているプロセスが 1 つの場合、そのプロセスはどの cgroup に配置されているかに関わらず、 必要なだけ CPU を消費する点に注意してください。CPU の制限は、複数のプロセスが CPU リソースを競い合う場合のみに有効となります。

- **cpuacct** – ***cpuacct.usage="0"*** パラメーターは、**cpuacct.usage** および **cpuacct.usage_percpu** のファイルに保存されている値をリセットするのに使用します。これらのファイルは、1 つの cgroup 内の全プロセスが消費する CPU 時間の合計 (ナノ秒単位) をレポートします。

- **memory** – ***memory.limit_in_bytes*** パラメーターは、特定の cgroup 内の全プロセスに提供されるメモリーの容量を示します。以下の例は、finance cgroup で起動したプロセスに 2 GB のメモリー、sales cgroup には 4 GB のメモリー、engineering cgroup には 8 GB のメモリーが割り当てられます。***memory.memsw.limit_in_bytes*** パラメーターは、スワップ領域のプロセスが使用できるメモリー容量の合計を指定します。finance cgroup 内のプロセスが 2 GB のメモリー上限に達すると、追加で 1 GB のスワップ領域を使用することができるので、合計で 3GB が設定されることになります。

2. 特定の cgroup にプロセスを移動するために **cgrulesengd** デーモンが使用するルールを定義するには、**/etc/cgrules.conf** を以下のように設定します。

```
#<user/group>          <controller(s)>          <cgroup>
@finance               cpu,cpuacct,memory       finance
@sales                 cpu,cpuacct,memory       sales
@engineering           cpu,cpuacct,memory       engineering
```

上記の設定により、特定のシステムグループ (例: ***@finance***) に使用可能なリソースコントローラー (例: **cpu**、**cpuacct**、**memory**) とそのシステムグループから起動される全プロセスを格納する cgroup (例: **finance**) を割り当てるルールを作成します。

この例では、**service cgred start** コマンドによって起動された **cgrulesengd** デーモンが、finance システムグループに属するユーザー (例: **jenn**) によって起動されたプロセスを検出すると、そのプロセスは自動的に **/cgroup/cpu_and_mem/finance/tasks** ファイルに移動し、finance cgroup で設定されているリソース制限の対象となります。

3. **cgconfig** サービスを起動し、cgroup の階層を作成して、作成した全 cgroup 内で必要なパラメーターを設定します。

```
~]# service cgconfig start
Starting cgconfig service:                                 [  OK
]
```

cgred サービスを起動して、**/etc/cgrules.conf** ファイルで設定されたシステムグループ内で起動されたプロセスを **cgrulesengd** デーモンに検出させます。

```
~]# service cgred start
Starting CGroup Rules Engine Daemon:                       [  OK  ]
```

cgred とは、**cgrulesengd** デーモンを起動するサービスの名前である点に注意してください。

4. リブート後にも変更をすべて保持するには、**cgconfig** および **cgred** のサービスがデフォルトで起動するように設定します。

```
~]# chkconfig cgconfig on
~]# chkconfig cgred on
```

この設定が機能するかどうかをテストするには、CPU またはメモリーを集中的に使用するプロセスを実行して、結果を観察します。たとえば、**top** ユーティリティを使用します。CPU リソース管理をテストするには、各ユーザー下で以下の **dd** コマンドを実行します。

```
~]$ dd if=/dev/zero of=/dev/null bs=1024k
```

上記のコマンドは **/dev/zero** を読み取り、その内容を 1024 KB 単位で **/dev/null** に出力します。**top** ユーティリティが起動すると、以下のような結果を確認することができます。

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 8201 peter     20   0  103m 1676  556 R 24.9  0.2   0:04.18 dd
 8202 mike      20   0  103m 1672  556 R 24.9  0.2   0:03.47 dd
 8199 jenn      20   0  103m 1676  556 R 12.6  0.2   0:02.87 dd
 8200 john      20   0  103m 1676  556 R 12.6  0.2   0:02.20 dd
 8197 martin    20   0  103m 1672  556 R 12.6  0.2   0:05.56 dd
 8198 mark      20   0  103m 1672  556 R 12.3  0.2   0:04.28 dd
 ⋮
```

全プロセスが cgroup に正しく割り当てられ、提供された CPU リソースのみを表示することができるようになります。finance と engineering の cgroup に属する 2 つのプロセス以外がすべて停止された場合、残りのリソースは両プロセス間で均等に分割されます。

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 8202 mike      20   0  103m 1676  556 R 66.4  0.2   0:06.35 dd
 8200 john      20   0  103m 1672  556 R 33.2  0.2   0:05.08 dd
 ⋮
```

その他の方法

cgrulesengd デーモンは、**/etc/cgrules.conf** に設定された該当する条件が満たされた後でしかプロセスを cgroup に移動しないので、そのプロセスが誤った cgroup で数ミリ秒間実行される場合があります。指定の cgroup にプロセスを移動する別の方法として、**pam_cgroup.so** PAM モジュールを使用する方法があります。このモジュールは、**/etc/cgrules.conf** ファイルで定義されているルールにしたがって使用可能な cgroup にプロセスを移動します。手順4.4「PAM モジュールを使用した、cgroup へのプロセス移行」 に記載した手順にしたがって **pam_cgroup.so** PAM モジュールを設定してください。

手順4.4 PAM モジュールを使用した、cgroup へのプロセス移行

1. オプションの Red Hat Enterprise Linux Yum リポジトリから libcgroup-pam パッケージをインストールします。

```
~]# yum install libcgroup-pam --enablerepo=rhel-6-server-optional-
rpms
```

2. PAM モジュールがインストール済みで、存在していることを確認します。

```
~]# ls /lib64/security/pam_cgroup.so
/lib64/security/pam_cgroup.so
```

32 ビットのシステムでは、モジュールは **/lib/security** ディレクトリに配置される点に注意してください。

3. **/etc/pam.d/su** ファイルに以下の行を追記して、**su** コマンドが事項されるたびに **pam_cgroup.so** モジュールを使用するようにします。

```
session         optional        pam_cgroup.so
```

4. 手順4.4「PAM モジュールを使用した、cgroup へのプロセス移行」に示したように、**/etc/cgconfig.conf** および **/etc/cgrules.conf** のファイルを設定します。

5. **/etc/cgrules.conf** ファイルの cgroup 設定の影響を受けるユーザーをすべてログアウトし、上記の設定を適用します。

pam_cgroup.so PAM モジュールを使用する際には、**cgred** サービスを無効にすることができます。

付録A 改訂履歴

改訂 1.0-18.5.400 **2013-10-31** **Rüdiger Landmann**
Rebuild with publican 4.0.0

改訂 1.0-18.5 **Fri Mar 29 2013** **Credit Translator's**
エラーを修正

改訂 1.0-18.4 **Tue Mar 19 2013** **Credit Translator's**
レンダリング修正のために Publican 3.1.5 で再ビルド

改訂 1.0-18.3 **Mon Mar 11 2013** **Credit Translator's**
翻訳を更新

改訂 1.0-18.2 **Mon Mar 4 2013** **Credit Translator's**
翻訳完了

改訂 1.0-18.1 **Mon Mar 4 2013** **Credit Translator's**
翻訳ファイルを XML ソース 1.0-18 と同期

改訂 1.0-18 **Thu Feb 21 2013** **Martin Prpič**
Red Hat Enterprise Linux 6.4 『リソース管理ガイド』 の GA リリース。各種バグ修正と新規コンテンツを含む。
- 最終のユースケースシナリオ –584631
- **perf_event** コントローラーについての説明を記載 –807326
- 共通の cgroup ファイルについての説明を記載 –807329
- OOM 制御および通知 API についての説明を記載 –822400、822401
- CPU 上限の適用についての説明を記載 –828991

改訂 1.0-7 **Wed Jun 20 2012** **Martin Prpič**
Red Hat Enterprise Linux 6.3 『リソース管理ガイド』 の GA リリース
- ユースケースを 2 つ追加
- **net_prio** サブシステムについての説明を追加

改訂 1.0-6 **Tue Dec 6 2011** **Martin Prpič**
Red Hat Enterprise Linux 6.2 『リソース管理ガイド』 の GA リリース

改訂 1.0-5 **Thu May 19 2011** **Martin Prpič**
Red Hat Enterprise Linux 6.1 『リソース管理ガイド』 の GA リリース

改訂 1.0-4 **Tue Mar 1 2011** **Martin Prpič**
- 複数の例を修正 – BZ#667623、BZ#667676、BZ#667699
- cgclear コマンドについての説明を明確化 –BZ#577101
- lssubsystem コマンドについての説明を明確化 –BZ#678517
- プロセスのフリーズ – BZ#677548

改訂 1.0-3 **Wed Nov 17 2010** **Rüdiger Landmann**
再マウントの例を修正 –BZ#612805

改訂 1.0-2 **Thu Nov 11 2010** **Rüdiger Landmann**
プレリリースのフィードバック手順を削除

改訂 1.0-1 **Wed Nov 10 2010** **Rüdiger Landmann**
QE からの修正 –BZ#581702 および BZ#612805

改訂 1.0-0 **Tue Nov 9 2010** **Rüdiger Landmann**
GA 用の機能実装完了バージョン