Red Hat Enterprise Linux for Real Time 8

低レイテンシー操作のための RHEL 8 for Real Time の最適化

Red Hat Enterprise Linux での RHEL for Real Time カーネルの最適化

Last Updated: 2024-05-17

Red Hat Enterprise Linux for Real Time 8 低レイテンシー操作のための RHEL 8 for Real Time の最適化

Red Hat Enterprise Linux での RHEL for Real Time カーネルの最適化

法律上の通知

概要

RHEL for Real Time カーネル上のワークステーションを調整して、一貫した低レイテンシーと、レイテンシーの影響を受けやすいアプリケーションで予測可能な応答時間を実現します。システムリソースを管理し、イベント間のレイテンシーを測定して、厳密な決定論要件でアプリケーションのレイテンシーを分析用に記録することで、リアルタイムカーネルチューニングを実行します。

目次

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

RED HAT ドキュメントへのフィードバック (英語のみ)

Red Hat ドキュメントに関するご意見やご感想をお寄せください。また、改善点があればお知らせください。

Jira からのフィードバック送信 (アカウントが必要)

1. Jira の Web サイトにログインします。
2. 上部のナビゲーションバーで **Create** をクリックします。
3. **Summary** フィールドにわかりやすいタイトルを入力します。
4. **Description** フィールドに、ドキュメントの改善に関するご意見を記入してください。ドキュメントの該当部分へのリンクも追加してください。
5. ダイアログの下部にある **Create** をクリックします。

第1章 RHEL 8 におけるリアルタイムカーネルチューニング

待ち時間、または応答時間は、イベントからの時間とシステム応答を指します。通常、マイクロ秒 (µs) で測定されます。

Linux 環境で実行されているほとんどのアプリケーションでは、基本的なパフォーマンスチューニングにより、レイテンシーを十分に改善できます。レイテンシーが低く、説明責任があり、予測可能性が高い必要がある業界向けに、Red Hat には、これらの要件を満たすレイテンシーを設定できる代替カーネルがあります。**RHEL for Real Time 8** は、**RHEL 8** とのシームレスな統合を提供し、クライアントが組織内のレイテンシーを測定、設定、および記録する機会を提供します。

リアルタイムカーネルは、適切にチューニングされたシステム、および非常に高い決定論要件を持つアプリケーションに使用します。カーネルシステムをチューニングすると、決定論を大幅に向上させることができます。まず、標準の **RHEL 8** システムの一般的なシステムチューニングを実行してから、RHEL for Real Time カーネルをデプロイします。

警告

これらのタスクを実行しないと、RHEL for Real Time デプロイメントから一貫したパフォーマンスが得られなくなる可能性があります。

1.1. チューニングガイドライン

- リアルタイムチューニングは反復的なプロセスで、いくつかの変数を微調整するだけで最適な変更が得られることはありません。お使いのシステムに最適なチューニング設定のセットを絞り込むためには、数日または数週間かかると思ってください。
  また、必ず長時間のテストを実行してください。あるチューニングパラメーターを変更してから 5 分間のテストを実行しても、特定のチューニング変更のセットを適切に検証したとは言えません。テストの長さを調整可能にして、数分間ではなくより長い時間実行してください。数時間のテストを実行して、いくつかの異なるチューニング設定のセットに絞り込み、それらのセットを一度に数時間から数日間実行することで、レイテンシーまたはリソース消費が大きい特殊なケースを検出できます。

- アプリケーションに測定メカニズムを構築し、特定のチューニング変更がアプリケーションのパフォーマンスにどのように影響を及ぼすかを正確に測定できるようにします。たとえば、"マウスの動きがよりスムーズになる" などの事例証拠は、正しくないことがほとんどで、人によって異なります。ハード的に測定を行い、後で分析できるようにそれらを記録します。

- テスト実行の間でチューニング変数に複数の変更を加えがちです。しかし、そうすると、テスト結果に影響を及ぼしたチューニングパラメーターを絞り込むことができなくなります。テスト間のチューニング変更は、可能な限り小さくします。

- また、チューニングを行う場合、大きな変更を加えがちですが、ほとんどの場合、増分変更を行う方が適切です。優先度の低い値から高い値に向かってテストを進めることで、長期的に見て良い結果が得られます。

- 利用可能なツールを使用してください。**tuna** チューニングツールを使用すると、スレッドと割り込みに対するプロセッサーアフィニティー、スレッドの優先順位を簡単に変更したり、アプリケーションで使用するプロセッサーを分離したりできます。**taskset** および **chrt** コマンドラ

インユーティリティーを使用すると、**tuna** が実行するほとんどのことを実行できます。パフォーマンスの問題が発生した場合、**ftrace** および **perf** ユーティリティーがレイテンシーの問題の特定に役立ちます。

- アプリケーションで値をハードコーディングするのではなく、外部ツールを使用してポリシー、優先度、アフィニティーを変更します。外部ツールを使用すると、さまざまな組み合わせを試すことができ、ロジックが簡素化されます。良好な結果が得られる設定をいくつか見つけたら、それらをアプリケーションに追加するか、アプリケーションの起動時に設定が実装されるように起動ロジックを設定します。

1.2. ロギングパラメーターのバランシング

syslog サーバーは、プログラムからのログメッセージをネットワーク経由で送信します。送信の頻度が低いほど、保留中のトランザクションが大きくなる可能性があります。トランザクションが非常に大きい場合、I/O スパイクが発生する可能性があります。これを防ぐには、間隔を合理的な範囲で小さい値に維持します。

システムロギングデーモン **syslogd** は、さまざまなプログラムからメッセージを収集するために使用されます。また、カーネルが報告する情報をカーネルロギングデーモン **klogd** から収集します。通常、**syslogd** によりローカルファイルにログが記録されますが、リモートロギングサーバーにネットワーク経由でログを記録するように設定することもできます。

手順

リモートロギングを有効にするには、以下を実行します。

1. ログの送信先となるマシンを設定します。詳細は、Red Hat Enterprise Linux での rsyslog を使用したリモート Syslog を参照してください。

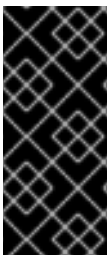
2. ログをリモートログサーバーに送信する各システムを設定して、**syslog** 出力がローカルファイルシステムではなくサーバーに書き込まれるようにします。これを行うには、各クライアントシステムの **/etc/rsyslog.conf** ファイルを編集します。そのファイルで定義されているロギングルールごとに、ローカルログファイルをリモートロギングサーバーのアドレスに置き換えます。

```
# Log all kernel messages to remote logging host.
kern.*     @my.remote.logging.server
```

上記の例では、すべてのカーネルメッセージをリモートマシン (**@my.remote.logging.server**) に記録するようにクライアントシステムを設定します。

または、以下の行を **/etc/rsyslog.conf** ファイルに追加して、ローカルに生成されたすべてのシステムメッセージをログに記録するように **syslogd** を設定できます。

```
# Log all messages to a remote logging server:
.    @my.remote.logging.server
```



重要

syslogd デーモンには、生成されたネットワークトラフィックに対する組み込みのレート制限は含まれていません。したがって、Red Hat は、RHEL for Real Time システムを使用する場合は、組織がリモートでのログ記録を要求するメッセージのみをログに記録することを推奨します。たとえば、カーネルの警告、認証要求などです。その他のメッセージはローカルに記録する必要があります。

関連情報

- **syslog(3)** の man ページ
- **rsyslog.conf(5)** の man ページ
- **rsyslogd(8)** の man ページ

1.3. 不要なアプリケーションの実行を回避することによるパフォーマンスの改善

実行中のアプリケーションはすべて、システムリソースを使用します。システムで不要なアプリケーションが実行されないようにすることで、パフォーマンスを大幅に向上させることができます。

前提条件

- システムの root 権限がある。

手順

1. 絶対に必要とされない場所 (特にサーバー) では、**グラフィカルインターフェイス** を実行しないでください。
   システムが、デフォルトで GUI で起動するように設定されているかどうかを確認します。

```
# systemctl get-default
```

2. コマンドの出力が **graphical.target** の場合は、システムがテキストモードに起動するように設定します。

```
# systemctl set-default multi-user.target
```

3. チューニングしているシステムで **Mail Transfer Agent (MTA)** をアクティブに使用している場合を除き、無効にしてください。MTA が必要な場合は、適切にチューニングされていることを確認するか、専用のマシンに移動することを検討してください。
   詳細は MTA のドキュメントを参照してください。

重要

MTA は、**cron** などのプログラムで実行されるシステム生成メッセージを送信するために使用されます。これには、**logwatch()** などのロギング機能によって生成されるレポートが含まれます。マシン上の MTA が無効になっていると、これらのメッセージを受信することはできません。

4. マウス、キーボード、Web カムなどの **周辺機器** は、レイテンシーに悪影響を与える可能性のある割り込みを送信します。グラフィカルインターフェイスを使用していない場合は、未使用の周辺機器をすべて取り外して無効にします。
   詳細は、デバイスのドキュメントを参照してください。

5. パフォーマンスに影響を及ぼす可能性のある自動 **cron** ジョブの有無を確認します。

```
# crontab -l
```

   crond サービスまたは不要な **cron** ジョブを無効にします。

6. サードパーティーのアプリケーションおよび外部のハードウェアベンダーが追加したコンポーネントの有無についてシステムを確認し、不要なものをすべて削除します。

関連情報

- **cron (8)** の man ページ

1.4. 不均等メモリーアクセス (NUMA)

taskset ユーティリティーは CPU アフィニティーでのみ機能し、メモリーノードなどの他の NUMA リソースについては認識しません。プロセスバインディングを NUMA と併用する場合は、**taskset** の代わりに **numactl** を使用します。

NUMA API の詳細は、Andi Kleen 氏のホワイトペーパー An NUMA API for Linux を参照してください。

関連情報

- **numactl(8)** の man ページ

1.5. DEBUGFS のマウントの確認

debugfs ファイルシステムは、デバッグや、ユーザーへの情報提供を目的として特別に設計されています。RHEL 8 では、これは **/sys/kernel/debug/** ディレクトリーに自動的にマウントされます。

注記

debugfs ファイルシステムは、**ftrace** コマンドおよび **trace-cmd** コマンドを使用してマウントされます。

手順

debugfs がマウントされていることを確認するには、以下を行います。

- 以下のコマンドを実行します。

```
# mount | grep ^debugfs
debugfs on /sys/kernel/debug type debugfs (rw,nosuid,nodev,noexec,relatime,seclabel)
```

debugfs がマウントされている場合は、**debugfs** のマウントポイントとプロパティーが表示されます。

debugfs がマウントされていない場合は、何も返されません。

1.6. RHEL FOR REAL TIME の INFINIBAND

InfiniBand は、帯域幅を増やし、サービス品質 (QOS) を向上させ、フェイルオーバーを提供するためによく使用される通信アーキテクチャーの一種です。また、RDMA (Remote Direct Memory Access) メカニズムでのレイテンシー改善にも使用できます。

RHEL for Real Time での InfiniBand のサポートは、Red Hat Enterprise Linux 8 で利用可能なサポートと同じです。詳細は、InfiniBand および RDMA ネットワークの設定 を参照してください。

1.7. ROCE および高パフォーマンスネットワークの使用

RoCEE (RDMA over Converged Enhanced Ethernet) は、イーサネットネットワークを介した RDMA (Remote Direct Memory Access) を実装するプロトコルです。これにより、重要なトランザクションに対して決定論ベースの低レイテンシーデータ転送を提供する一方で、データセンターで一貫性のある高速環境を維持できます。

High Performance Networking (HPN) は、**RoCEE** インターフェイスをカーネルに提供する共有ライブラリーセットです。**HPN** は、独立したネットワークインフラストラクチャーを経由する代わりに、標準のイーサネットインフラストラクチャーを使用してリモートシステムメモリーにデータを直接配置するため、CPU オーバーヘッドが少なく、インフラストラクチャーコストが削減されます。

RHEL for Real Time の **RoCEE** および **HPN** へのサポートは、RHEL 8 で提供されるサポートと同じです。

関連情報

- RoCE の設定。

1.8. RHEL FOR REAL TIME のコンテナー調整

主要な RHEL カーネルでは、リアルタイムグループスケジューリング機能 **CONFIG_RT_GROUP_SCHED** がデフォルトで有効になっています。ただし、リアルタイムカーネルの場合、この機能は無効になります。

CONFIG_RT_GROUP_SCHED 機能は、**kernel-rt** パッケージで使用される **PREEMPT_RT** パッチセットとは独立して開発され、メインの RHEL カーネル上のリアルタイムプロセスで動作することを目的としています。**CONFIG_RT_GROUP_SCHED** 機能は、レイテンシーのスパイクを引き起こすことが知られているため、**PREEMPT_RT** が有効なカーネルでは無効化されています。したがって、メインの RHEL カーネルで実行しているコンテナーでワークロードをテストする場合は、コンテナー内で **SCHED_FIFO** または **SCHED_RR** タスクを実行できるように、一部のリアルタイム帯域幅をコンテナーに割り当てる必要があります。

手順

1. podman の **--cpu-rt-runtime** コマンドラインオプションを使用する前に、以下のグローバル設定を行います。
   # echo 950000 > /sys/fs/cgroup/cpu,cpuacct/machine.slice/cpu.rt_runtime_us

2. CPU 分離の場合は、既存の推奨事項に従って RT ワークロード用にコアセットを確保してください。

3. 分離した CPU コアのリストを使用して **podman run --cpuset-cpus** を実行します。

4. 使用する NUMA (Non-Uniform Memory Access) メモリーノードを指定します。
   ***podman run --cpuset-mems=number-of-memory-nodes**

   これにより、NUMA ノード間のメモリーアクセスが回避されます。

5. コンテナーで実行されているリアルタイムのワークロードに必要な最小量のメモリーがコンテナーの開始時に使用可能であることを確認するには、***podman run --memory-reservation=limit** コマンドを使用します。

関連情報

- **podman-run(1)** の man ページ

第2章 RHEL FOR REAL TIME のスケジューリングポリシー

リアルタイムでは、スケジューラーは、実行する実行可能なスレッドを決定するカーネルコンポーネントです。各スレッドには、関連付けられたスケジューリングポリシーおよび静的スケジューリング優先度 (**sched_priority**) があります。スケジューリングはプリエンプティブであるため、静的優先度の高いスレッドの実行の準備ができると、現在実行中のスレッドは停止します。その後、実行中のスレッドは静的優先度の **waitlist** に戻ります。

すべての Linux スレッドには、以下のいずれかのスケジューリングポリシーがあります。

- **SCHED_OTHER** または **SCHED_NORMAL**: デフォルトのポリシーです。
- **SCHED_BATCH**: **SCHED_OTHER** に似ていますが、増分指向です。
- **SCHED_IDLE**: **SCHED_OTHER** より優先度の低いポリシーです。
- **SCHED_FIFO**: 先入れ先出しのリアルタイムポリシーです。
- **SCHED_RR**: ラウンドロビンのリアルタイムポリシーです。
- **SCHED_DEADLINE**: ジョブの期限に従ってタスクに優先度を割り当てるスケジューラーポリシーです。絶対期限が最も早いジョブが最初に実行されます。

2.1. スケジューラーポリシー

リアルタイムスレッドは標準スレッドよりも優先度が高くなります。ポリシーには、最小値 1 から最大値 99 までの範囲のスケジューリング優先順位値があります。

次のポリシーは、リアルタイムにとって重要です。

- **SCHED_OTHER** または **SCHED_NORMAL** ポリシー
  これは、Linux スレッドのデフォルトスケジューリングポリシーです。スレッドの特性に基づいてシステムによって変更される動的な優先度があります。**SCHED_OTHER** スレッドの nice 値は、最高の優先度である 20 と最低の優先度である 19 の間です。**SCHED_OTHER** スレッドのデフォルトの nice 値は 0 です。

- **SCHED_FIFO** ポリシー
  SCHED_FIFO を持つスレッドは、**SCHED_OTHER** タスクよりも高い優先度で実行されます。**SCHED_FIFO** は、nice 値を使用する代わりに、最低が 1 で最高が 99 の固定された優先度を使用します。優先度 1 の **SCHED_FIFO** スレッドは、**SCHED_OTHER** スレッドよりも常に先にスケジュールされます。

- **SCHED_RR** ポリシー
  SCHED_RR ポリシーは、**SCHED_FIFO** ポリシーに似ています。同じ優先度のスレッドは、ラウンドロビン方式でスケジュールされます。**SCHED_FIFO** および **SCHED_RR** スレッドは以下のイベントのいずれかが発生するまで実行されます。

  - スレッドはスリープ状態になるか、イベントを待機します。

  - 優先度の高いリアルタイムスレッドを実行する準備が整います。
    上記のイベントのいずれかが発生しない限り、スレッドは指定されたプロセッサーで無期限に実行されますが、優先度の低いスレッドは実行を待機しているキューに残ります。これにより、システムサービススレッドが常駐し、スワップアウトが妨げられ、ファイルシステムデータのフラッシュが失敗する可能性があります。

- **SCHED_DEADLINE** ポリシー

SCHED_DEADLINE ポリシーはタイミング要件を指定します。タスクの期限に従って各タスクをスケジュールします。Earliest Deadline First (EDF) スケジュールを持つタスクが最初に実行されます。

カーネルは、**runtime⇐deadline⇐period** が true である必要があります。必要なオプション間の関係は、**runtime⇐deadline⇐period** です。

2.2. SCHED_DEADLINE ポリシーのパラメーター

各 **SCHED_DEADLINE** タスクは、**period**、**runtime**、および **deadline** パラメーターによって特徴付けられます。これらのパラメーターの値は、ナノ秒の整数です。

表2.1 SCHED_DEADLINE パラメーター

パラメーター	説明
period	**period** はリアルタイムタスクの起動パターンです。 たとえば、ビデオ処理タスクで 1 秒あたり 60 フレームの処理が必要な場合、新しいフレームは 16 ミリ秒ごとにサービスのキューに入れられます。したがって、**period** は 16 ミリ秒になります。
runtime	**runtime** は、出力を生成するためにタスクに割り当てられた CPU 実行時間の量です。リアルタイムでは、最悪実行時間 (WCET) とも呼ばれる最大実行時間は **runtime** です。 たとえば、ビデオ処理ツールが画像を処理するのに最悪の場合で 5 ミリ秒かかる場合、**runtime** は 5 ミリ秒になります。
deadline	**deadline** は、出力が生成される最大時間です。 たとえば、タスクが処理されたフレームを 10 ミリ秒以内に配信する必要がある場合、**deadline** は 10 ミリ秒になります。

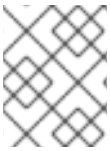
第3章 永続的なカーネルチューニングパラメーターの設定

システムで機能するチューニング設定を決定したら、変更を永続化して再起動後も維持できます。

デフォルトでは、編集したカーネルチューニングパラメーターは、システムが再起動するか、パラメーターが明示的に変更されるまで有効になります。この方法は、初期チューニング設定を確立するために有効です。また、安全性のメカニズムも提供します。編集したパラメーターによって、マシンの動作が不安定になった場合、マシンを再起動すると、パラメーターが以前の設定に戻ります。

3.1. カーネルチューニングパラメーターの変更の永続化

パラメーターを **/etc/sysctl.conf** ファイルに追加することにより、カーネルチューニングパラメーターに永続的な変更を加えることができます。



注記

この手順では、現在のセッションのカーネルチューニングパラメーターは **変更されません**。**/etc/sysctl.conf** に入力した変更は、今後のセッションにのみ影響します。

前提条件

- システムの root 権限がある。

手順

1. テキストエディターで **/etc/sysctl.conf** を開きます。
2. パラメーターの値を使用して、新規エントリーをファイルに挿入します。
   /proc/sys/ パスを削除し、残りのスラッシュ (/) をピリオド (**.**) に変更して、パラメーターの値を追加することで、パラメーター名を変更します。

   たとえば、コマンド **echo 0 > /proc/sys/kernel/hung_task_panic** を永続化するには、以下を **/etc/sysctl.conf** に入力します。

   ```
   # Enable gettimeofday(2)
   kernel.hung_task_panic = 0
   ```

3. ファイルを保存してから閉じます。
4. システムを再起動して、変更を有効にします。

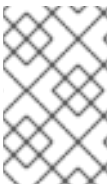
検証

- 設定を確認するには、以下を実行します。

  ```
  # cat /proc/sys/kernel/hung_task_panic
  0
  ```

第4章 アプリケーションのチューニングとデプロイメント

最適な設定を組み合わせてリアルタイムカーネルをチューニングすると、RHEL for Real Time アプリケーションの改良と開発に役立ちます。



注記

一般に、**POSIX** 定義の API (アプリケーションプログラミングインターフェイス) を使用するようにしてください。RHEL for Real Time は **POSIX** 標準に準拠しています。RHEL for Real Time カーネルのレイテンシー削減も、**POSIX** をベースにしています。

4.1. リアルタイムアプリケーションでのシグナル処理

従来の **UNIX** および **POSIX** シグナルは、特にエラー処理に使用されますが、リアルタイムアプリケーションでのイベント配信メカニズムとしての使用には適していません。その理由は、現在の Linux カーネルシグナル処理コードが、非常に複雑なためです。これは主に、従来の動作と多くの API をサポートする必要があるためです。この複雑さは、シグナルの配信時に使用されるコードパスが常に最適とは限らず、アプリケーションでレイテンシーが長くなる可能性があることを意味します。

UNIX シグナルの本来の目的は、実行の異なるスレッド間で 1 つの制御スレッド (プロセス) を多重化することでした。シグナルはオペレーティングシステムの割り込みのように動作します。つまり、シグナルがアプリケーションに配信されると、アプリケーションのコンテキストが保存され、事前に登録したシグナルハンドラーの実行を開始します。シグナルハンドラーが完了すると、アプリケーションはシグナルの配信時の地点から処理を再開します。この動作は、実際には複雑になる可能性があります。

シグナルは、決定論ベースではないので、リアルタイムアプリケーションでは信頼できません。POSIX スレッド (pthreads) を使用してワークロードを分散し、さまざまなコンポーネント間の通信を行うことが望ましいオプションです。ミューテックスの pthreads メカニズム、条件変数、およびバリアを使用して、スレッドのグループを調整できます。このような比較的新しい構造によるコードパスは、シグナルに対する従来の処理コードよりもはるかにクリーンです。

関連情報

- POSIX シグナルモデルの要件

4.2. スレッドの同期

sched_yield コマンドは、優先度の低いスレッドに実行の機会を与える同期メカニズムです。このタイプの要求は、適切に作成されていないアプリケーション内から発行すると失敗する可能性があります。

優先度が高いスレッドは **sched_yield()** を呼び出して、他のスレッドに実行の機会を与えることができます。呼び出しプロセスは、その優先度で実行されているプロセスのキューの末尾に移動します。これは、同じ優先度で他のプロセスが実行していない状況で発生すると、呼び出しプロセスの実行は継続されます。プロセスの優先度が高い場合は、ビジーループが発生し、マシンが使用できなくなる可能性があります。

SCHED_DEADLINE タスクが **sched_yield()** を呼び出すと、設定された CPU が放棄され、残りのランタイムには次の期間まで直ちにスロットリングが適用されます。**sched_yield()** の動作により、タスクは次の期間の開始時に起動できます。

スケジューラーは、実際に実行する他のスレッドがあるかどうかを判別できます。リアルタイムタスクで **sched_yield()** は使用しないでください。

手順

- **sched_yield()** 関数を呼び出すには、以下のコードを実行します。

```
for(;;) {
        do_the_computation();
        /*
 * Notify the scheduler the end of the computation
         * This syscall will block until the next replenishment
         */
    sched_yield();
      }
```

SCHED_DEADLINE タスクには、次の期間 (次のループ実行を開始する) まで、競合ベースの検索 (CBS) アルゴリズムによってスロットリングが適用されます。

関連情報

- **pthread.h(P)** の man ページ
- **sched_yield(2)** の man ページ
- **sched_yield(3p)** の man ページ

4.3. リアルタイムスケジューラーの優先度

systemd は、システムの起動時に実行するサービスのリアルタイムの優先度を設定します。一部のカーネルスレッドには非常に高い優先度が設定される場合があります。これにより、デフォルトの優先度を Real Time Specification for Java (RTSJ) の要件と適切に統合することができます。RTSJ には 10 から 89 までの優先度の範囲が必要です。

RTSJ が使用されていないデプロイメントでは、アプリケーションが使用できるスケジューリングの優先度が幅広くあります (90 未満)。重要なシステムサービスが実行されなくなる可能性があるので、49 を超える優先度のアプリケーションスレッドのスケジューリングには十分な注意を払ってください。そうしないと、ネットワークトラフィックのブロック、仮想メモリーのページングのブロック、ファイルシステムのジャーナリングのブロックによるデータの破損など、予測できない動作が発生する可能性があります。

アプリケーションスレッドが優先度 89 を超えてスケジュールされている場合は、スレッドが非常に短いコードパスのみを実行するようにしてください。これを行わないと、RHEL for Real Time カーネルの低レイテンシー機能が損なわれます。

必須の権限を持たないユーザーに対するリアルタイム優先度の設定

デフォルトでは、アプリケーションに対して root 権限を持つユーザーのみが優先度とスケジュール情報を変更できます。root 権限を付与するには、設定を変更します。推奨される方法は、ユーザーを **realtime** グループに追加することです。

重要

また、**/etc/security/limits.conf** ファイルを編集して、ユーザー権限を変更することもできます。ただし、これにより重複が発生し、通常のユーザーがシステムを使用できなくなる可能性があります。このファイルを編集する場合は、変更を行う前に必ずコピーを作成してください。

4.4. 動的ライブラリーの読み込み

リアルタイムアプリケーションの開発時には、プログラムの実行中に決定論ベースではないレイテンシーが発生しないように、システムの起動時にシンボルを解決することを検討してください。システムの起動時にシンボルを解決すると、プログラムの初期化に時間がかかる場合があります。動的リンカー/ローダーである **ld.so** を使用して **LD_BIND_NOW** 変数を設定することにより、動的ライブラリーをアプリケーションの起動時に読み込むように指示できます。

たとえば、このスクリプトは **LD_BIND_NOW** 変数を **1** の値でエクスポートしてから、**FIFO** のスケジューラーポリシーと **1** の優先度でプログラムを実行します。

```
#!/bin/sh

LD_BIND_NOW=1
export LD_BIND_NOW

chrt --fifo 1 _/opt/myapp/myapp-server &_
```

関連情報

- **ld.so(8)** man ページ

第5章 システムチューニング用の BIOS パラメーターの設定

BIOS は、システムの機能において重要な役割を果たします。BIOS パラメーターを正しく設定すると、システムのパフォーマンスを大幅に向上できます。

注記

システムおよび BIOS ベンダーはすべて、さまざまな用語とナビゲーション方法を使用します。BIOS 設定の詳細は、BIOS のドキュメントを参照するか、BIOS ベンダーにお問い合わせください。

5.1. 電源管理の無効化による応答時間の改善

BIOS 電源管理オプションは、システムクロックの周波数を変更したり、CPU をさまざまなスリープ状態の 1 つにすることで、電力を節約できるようにします。このようなアクションは、システムが外部イベントに応答する速度に影響を与える可能性があります。

応答時間を改善するには、BIOS の電源管理オプションをすべて無効にします。

5.2. エラー検出と修正ユニットの無効化による応答時間の改善

Error Detection and Correction (EDAC) ユニットは、Error Correcting Code (ECC) メモリーから通知されたエラーを検出および修正するためのデバイスです。通常 EDAC には、ECC をチェックしないオプションから、エラーに関するすべてのメモリーノードを定期的にスキャンするオプションまで、さまざまなオプションがあります。EDAC レベルが高いほど、BIOS が使用する時間が長くなります。これにより、重要なイベントの期限を逃す可能性があります。

応答時間を改善するには、EDAC をオフにします。これができない場合は、EDAC を最小機能レベルに設定します。

5.3. システム管理割り込みの設定による応答時間の改善

System Management Interrupts (SMI) は、システムが正常に動作していることを確認するハードウェアベンダーの機能です。BIOS コードは、通常、SMI 割り込みを処理します。SMI は通常、温度管理、リモートコンソール管理 (IPMI)、EDAC チェック、およびその他のハウスキーピングタスクに使用されます。

BIOS に SMI オプションが含まれる場合は、ベンダーおよび関連ドキュメントを確認して、それらを無効にしても安全な範囲を判断してください。

警告

SMI を完全に無効にすることは可能ですが、Red Hat ではこれを行わないことを強く推奨します。SMI を生成してサービスするシステムの機能を削除すると、壊滅的なハードウェア障害が発生する可能性があります。

第6章 ハードウェアおよびファームウェアのレイテンシーテストの実行および解釈

hwlatdetect プログラムを使用すると、潜在的なハードウェアプラットフォームがリアルタイム操作の使用に適しているかどうかをテストして検証できます。

前提条件

- **RHEL-RT** (RHEL for Real Time) パッケージおよび **rt-tests** パッケージがインストールされている。
- 低レイテンシー操作に必要なチューニング手順については、ベンダーのドキュメントを参照してください。
  ベンダーのドキュメントは、システムを System Management Mode (SMM) に移行する System Management Interrupts (SMI) を減らしたり、削除したりする手順を提供できます。システムが SMM にある間、ファームウェアを実行し、オペレーティングシステムのコードは実行しません。これは、SMM にある間にタイプアップするすべてのタイマーが、システムが通常の操作に戻るまで待機することを意味します。Linux では SMI をブロックできないため、これにより原因不明のレイテンシーが発生する可能性があります。実際に SMI を取得したことを示す唯一の兆候は、ベンダー固有のパフォーマンスカウンターレジスターにしかありません。

警告

致命的なハードウェア障害が発生する可能性があるため、Red Hat は SMI を完全に無効にしないことを強く推奨します。

6.1. ハードウェアおよびファームウェアのレイテンシーテストの実行

テストは、ハードウェアアーキテクチャー、またはBIOS もしくは EFI ファームウェアに起因する遅延を検出するため、**hwlatdetect** プログラムの実行中にシステムに負荷を掛ける必要はありません。**hwlatdetect** のデフォルト値では、毎秒 0.5 秒間ポーリングを行い、時刻を取得する連続した呼び出しの間に 10 マイクロ秒を超えるギャップがあれば、それを報告します。**hwlatdetect** は、システムで **最大限保証可能な** 最大レイテンシーを返します。したがって、10µs 未満の最大レイテンシー値を要求するアプリケーションがある場合、**hwlatdetect** がギャップの 1 つを 20µs と報告する場合、システムは 20µs のレイテンシーしか保証できません。

注記

hwlatdetect が、システムがアプリケーションのレイテンシー要件を満たせないことを示した場合は、BIOS 設定を変更するか、システムベンダーと協力して、アプリケーションのレイテンシー要件を満たす新しいファームウェアを入手してみてください。

前提条件

- **RHEL-RT** パッケージおよび **rt-tests** パッケージがインストールされている。

手順

- **hwlatdetect** を実行し、テスト期間を秒単位で指定します。

hwlatdetect は、クロックソースをポーリングし、原因不明のギャップを探すことにより、ハードウェアおよびファームウェアに起因するレイテンシーを探します。

```
# hwlatdetect --duration=60s
hwlatdetect:  test duration 60 seconds
 detector: tracer
 parameters:
  Latency threshold:    10us
  Sample window:        1000000us
  Sample width:         500000us
  Non-sampling period:  500000us
  Output File:          None

Starting test
test finished
Max Latency: Below threshold
Samples recorded: 0
Samples exceeding threshold: 0
```

関連情報

- **hwlatdetect** の man ページ
- ハードウェアおよびファームウェアのレイテンシーテストの解釈

6.2. ハードウェアおよびファームウェアのレイテンシーテストの解釈

ハードウェア遅延検出器 (**hwlatdetect**) は、トレーサーメカニズムを使用して、ハードウェアアーキテクチャーまたは BIOS/EFI ファームウェアによる遅延を検出します。**hwlatdetect** によって測定された遅延をチェックすることで、潜在的なハードウェアが RHEL for Real Time カーネルのサポートに適しているかどうかを判断できます。

例

- この結果の例は、ファームウェアによるシステム中断を最小限に抑えるように調整されたシステムを表しています。このような場合、**hwlatdetect** の出力は以下のようになります。

```
# hwlatdetect --duration=60s
hwlatdetect:  test duration 60 seconds
 detector: tracer
 parameters:
  Latency threshold: 10us
  Sample window:     1000000us
  Sample width:      500000us
  Non-sampling period:  500000us
  Output File:       None

Starting test
test finished
Max Latency: Below threshold
Samples recorded: 0
Samples exceeding threshold: 0
```

- この結果の例は、ファームウェアによるシステムの中断を最小限に抑えるようにチューニングできなかったシステムを表しています。このような場合、**hwlatdetect** の出力は以下のようになります。

```
# hwlatdetect --duration=10s
hwlatdetect:  test duration 10 seconds
 detector: tracer
 parameters:
  Latency threshold: 10us
  Sample window:     1000000us
  Sample width:      500000us
  Non-sampling period:  500000us
  Output File:       None

Starting test
test finished
Max Latency: 18us
Samples recorded: 10
Samples exceeding threshold: 10
SMIs during run: 0
ts: 1519674281.220664736, inner:17, outer:15
ts: 1519674282.721666674, inner:18, outer:17
ts: 1519674283.722667966, inner:16, outer:17
ts: 1519674284.723669259, inner:17, outer:18
ts: 1519674285.724670551, inner:16, outer:17
ts: 1519674286.725671843, inner:17, outer:17
ts: 1519674287.726673136, inner:17, outer:16
ts: 1519674288.727674428, inner:16, outer:18
ts: 1519674289.728675721, inner:17, outer:17
ts: 1519674290.729677013, inner:18, outer:17----
```

この出力は、システム **clocksource** の連続読み取り中に、15-18 us の範囲で 10 回の遅延が発生したことを示しています。

注記

以前のバージョンでは、**ftrace** トレーサーではなくカーネルモジュールを使用していました。

結果について

テスト方法、パラメーター、および結果に関する情報は、**hwlatdetect** ユーティリティーによって検出された遅延パラメーターと遅延値を理解するのに役立ちます。

テスト方法、パラメーター、および結果の表には、**hwlatdetect** ユーティリティーによって検出されたパラメーターと遅延値が記載されています。

表6.1テスト方法、パラメーター、および結果

パラメーター	値	説明
test duration	**10 秒**	テストの期間 (秒単位)
detector	**tracer**	**detector** スレッドを実行するユーティリティー

パラメーター	値	説明
パラメーター		
Latency threshold	**10us**	許容可能な最大レイテンシー
Sample window	**1000000us**	1 秒
Sample width	**500000us**	0.05 秒
Non-sampling period	**500000us**	0.05 秒
Output File	**なし**	出力が保存されるファイル。
結果		
Max Latency	**18us**	**Latency threshold** を超えたテスト中の最大レイテンシー。**Latency threshold** を超えたサンプルがない場合、レポートは **Below threshold** を表示します。
Samples recorded	**10**	テストによって記録されたサンプルの数。
Samples exceeding threshold	**10**	テストによって記録された、レイテンシーが **Latency threshold** を超過するサンプルの数。
SMIs during run	**0**	テストの実行中に発生した System Management Interrupts (SMI) の数。

注記

内部および外部の **hwlatdetect** ユーティリティーによって出力される値は、最大レイテンシー値です。これらは、現在のシステムクロックソース (通常は TSC または TSC レジスターですが、HPET または ACPI 電力管理クロックの可能性があります) の連続した読み取り間のデルタと、ハードウェアとファームウェアの組み合わせによって導入された連続した読み取り間の遅延です。

適切なハードウェアとファームウェアの組み合わせを見つけたら、次のステップは、負荷がかかった状態でシステムのリアルタイムパフォーマンスをテストすることです。

第7章 システムレイテンシーテストの実行および解釈

RHEL for Real Time は、負荷がかかった状態でシステムのリアルタイムパフォーマンスをテストするための **rteval** ユーティリティーを提供します。

7.1. システムレイテンシーテストの実行

rteval ユーティリティーを使用すると、負荷がかかった状態でシステムのリアルタイムパフォーマンスをテストできます。

前提条件

- **RHEL for Real Time** パッケージグループがインストールされている。
- システムの root 権限がある。

手順

- **rteval** ユーティリティーを実行します。

```
# rteval
```

rteval ユーティリティーは、**SCHED_OTHER** タスクの高いシステム負荷を開始します。次に、それぞれのオンライン CPU でリアルタイムの応答を測定します。負荷は、ループの Linux カーネルツリーと **hackbench** の合成ベンチマークの並列の **make** になります。

その目的は、システムを、それぞれのコアに常にスケジュールするジョブがある状態にすることです。ジョブは、メモリーの割り当て/解放、ディスク I/O、コンピュートタスク、メモリーコピーなどのさまざまなタスクを実行します。

負荷が開始されると、**rteval** は **cyclictest** 測定プログラムを開始します。このプログラムは、各オンラインコアで **SCHED_FIFO** リアルタイムスレッドを起動します。その後、リアルタイムスケジューリングの応答時間を測定します。

各測定スレッドはタイムスタンプを取得し、ある間隔スリープした後、ウェイクアップ後に再度タイムスタンプを取得します。測定されたレイテンシーは **t1 - (t0 + i)** です。これは、実際のウェイクアップ時刻 **t1** と、最初のタイムスタンプの理論的なウェイクアップ時刻 **t0** にスリープ間隔 **i** を加えたものとの差になります。

rteval 実行の詳細は、システムのブートログと共に XML ファイルに書き込まれます。このレポートは画面に表示され、圧縮されたファイルに保存されます。

ファイル名は **rteval-<date>-N-tar.bz2** の形式になっています。ここで、**<date>** はレポートが生成された日付に置き換えます。**N** は **<date>** の N 番目の実行のカウンターになります。

以下は、**rteval** レポートの例です。

```
System:
Statistics:
 Samples:          1440463955
 Mean:            4.40624790712us
 Median:           0.0us
 Mode:             4us
 Range:            54us
 Min:             2us
```

```
Max:              56us
Mean Absolute Dev: 1.0776661507us
Std.dev:          1.81821060672us

CPU core 0       Priority: 95
Statistics:
 Samples:          36011847
 Mean:             5.46434910711us
 Median:           4us
 Mode:             4us
 Range:            38us
 Min:              2us
 Max:              40us
 Mean Absolute Dev: 2.13785341159us
 Std.dev:          3.50155558554us
```

レポートには、システムハードウェア、実行の長さ、使用したオプション、およびタイミング結果 (CPU ごとおよびシステム全体) の詳細が表示されます。

注記

生成されたファイルから **rteval** レポートを再生成するには、以下を実行します。

 # rteval --summarize rteval-<date>-N.tar.bz2

第8章 RHEL FOR REAL TIME での CPU アフィニティーの設定

システム内のすべてのスレッドと割り込みソースには、プロセッサーアフィニティープロパティーがあります。オペレーティングシステムスケジューラーは、この情報を使用して、CPU で実行するスレッドと割り込みを決定します。プロセッサーアフィニティーを効果的なポリシーおよび優先度設定とともに設定することで、パフォーマンスを最大限に高めることができます。アプリケーションは、特に CPU 時間などのリソースに対して、他のプロセスと常に競合します。アプリケーションによっては、関連するスレッドが同じコアで実行されることがよくあります。1 つのアプリケーションスレッドを 1 つのコアに割り当てることができます。

マルチタスクを実行するシステムは、一般的に非決定論の傾向にあります。優先度の高いアプリケーションであっても、優先度の低いアプリケーションがコードの重要なセクションにある場合は、アプリケーションの実行が遅延する可能性があります。優先度の低いアプリケーションが重要なセクションを終了すると、カーネルは優先度の低いアプリケーションのプリエンプションを安全に実行し、プロセッサーで高い優先順位のアプリケーションをスケジュールします。また、キャッシュの無効化により、ある CPU から別の CPU への移行の負荷が大きくなることがあります。RHEL for Real Time には、これらの問題の一部に対処し、レイテンシーをより適切に制御できるようにするツールが含まれています。

アフィニティーはビットマスクで表され、マスクの各ビットが CPU コアを表します。ビットが 1 に設定されている場合は、スレッドまたは割り込みがそのコアで実行されます。0 を指定すると、スレッドまたは割り込みがコア上の実行から除外されます。アフィニティービットマスクのデフォルト値はすべて 1 です。つまり、スレッドまたは割り込みがシステムの任意のコアで実行できます。

デフォルトでは、プロセスは任意の CPU で実行できます。ただし、プロセスのアフィニティーを変更することにより、所定の CPU セットで実行するプロセスを定義できます。子プロセスは、そのロールの CPU アフィニティーを継承します。

次の一般的なアフィニティー設定を設定すると、最大限のパフォーマンスを実現できます。

- すべてのシステムプロセスに単一の CPU コアを使用し、残りのコアで実行するようにアプリケーションを設定する。
- 同じ CPU 上でスレッドアプリケーションと特定のカーネルスレッド (ネットワーク **softirq** やドライバースレッドなど) を設定する。
- 各 CPU でプロデューサーとコンシューマーのスレッドをペアリングする。プロデューサーとコンシューマーは 2 つのクラスのスレッドであり、プロデューサーはデータをバッファーに挿入し、コンシューマーはデータをバッファーから削除します。

リアルタイムシステムでアフィニティーを調整するための通常のグッドプラクティスは、アプリケーションの実行に必要なコアの数を決定してから、それらのコアを分離することです。これは、Tuna ツールまたはシェルスクリプトを使用して、**taskset** コマンドなどのビットマスク値を変更することで実現できます。**taskset** コマンドはプロセスのアフィニティーを変更し、**/proc/** ファイルシステムエントリーを変更すると割り込みのアフィニティーが変更されます。

8.1. TASKSET コマンドを使用したプロセッサーアフィニティーの調整

リアルタイムでは、**taskset** コマンドは、実行中のプロセスの CPU アフィニティーを設定または取得するのに役立ちます。**taskset** コマンドは、**-p** オプションおよび **-c** オプションを取ります。**-p** オプションまたは **--pid** オプションは既存のプロセスを機能させ、新しいタスクを開始しません。**-c** または **--cpu-list** は、**bitmask** の代わりにプロセッサーの数値リストを指定します。リストには、コンマで区切られた複数の項目、およびプロセッサーの範囲を含めることができます。たとえば、0,5,7,9-11 です。

前提条件

- システムの root 権限がある。

手順

- 特定のプロセスのプロセスアフィニティーを確認するには、次の手順を実行します。

```
# taskset -p -c 1000
  pid 1000's current affinity list: 0,1
```

このコマンドは、プロセスのアフィニティーを PID 1000 で出力します。プロセスは、CPU 0 または CPU 1 を使用するように設定されています。

  - (オプション) プロセスをバインドするように特定の CPU を設定するには、以下を実行します。

```
# taskset -p -c 1 1000
pid 1000's current affinity list: 0,1
pid 1000's new affinity list: 1
```

  - (オプション) 複数の CPU アフィニティーを定義するには、以下を実行します。

```
# taskset -p -c 0,1 1000
pid 1000's current affinity list: 1
pid 1000's new affinity list: 0,1
```

  - (オプション) 特定の CPU で優先度レベルとポリシーを設定するには、以下を実行します。

```
# taskset -c 5 chrt -f 78 /bin/my-app
```

さらに細かく設定するために、優先度とポリシーを指定することもできます。この例では、コマンドは **SCHED_FIFO** ポリシーと優先度値 78 を使用して CPU 5 で **/bin/my-app** アプリケーションを実行します。

8.2. SCHED_SETAFFINITY() システムコールを使用したプロセッサーアフィニティーの設定

リアルタイム **sched_setaffinity()** システムコールを使用して、プロセッサーアフィニティーを設定することもできます。

前提条件

- システムの root 権限がある。

手順

- **sched_setaffinity()** でプロセッサーアフィニティーを設定するには、以下を実行します。

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sched.h>
```

```
int main(int argc, char **argv)
{
  int i, online=0;
  ulong ncores = sysconf(_SC_NPROCESSORS_CONF);
  cpu_set_t *setp = CPU_ALLOC(ncores);
  ulong setsz = CPU_ALLOC_SIZE(ncores);

  CPU_ZERO_S(setsz, setp);

  if (sched_getaffinity(0, setsz, setp) == -1) {
    perror("sched_getaffinity(2) failed");
    exit(errno);
  }

  for (i=0; i < CPU_COUNT_S(setsz, setp); i) {
    if (CPU_ISSET_S(i, setsz, setp))
      online;
  }

  printf("%d cores configured, %d cpus allowed in affinity mask\n", ncores, online);
  CPU_FREE(setp);
}
```

8.3. 単一の CPU を分離して、使用率の高いタスクを実行

cpusets メカニズムを使用すると、**SCHED_DEADLINE** タスクに一連の CPU とメモリーノードを割り当てることができます。CPU 使用率の高いタスクと低いタスクが混在するタスクセットにおいて、使用率の高いタスクを実行する CPU を分離し、使用率の低いタスクを異なる CPU セットでスケジューリングすることで、すべてのタスクが与えられた **runtime** を満たすことが可能になります。

前提条件

- システムの root 権限がある。

手順

1. **cpuset** という名前の 2 つのディレクトリーを作成します。

```
# cd /sys/fs/cgroup/cpuset/
# mkdir cluster
# mkdir partition
```

2. ルート **cpuset** の負荷分散を無効にして、**cpuset** ディレクトリーに 2 つの新しいルートドメインを作成します。

```
# echo 0 > cpuset.sched_load_balance
```

3. クラスター **cpuset** で、利用率の低いタスクが CPU 1 から 7 で実行されるようにスケジュールし、メモリーサイズを確認し、CPU に exclusive という名前を付けます。

```
# cd cluster/
# echo 1-7 > cpuset.cpus
# echo 0 > cpuset.mems
```

```
# echo 1 > cpuset.cpu_exclusive
```

4. 使用率の低いすべてのタスクを cpuset ディレクトリーに移動します。

```
# ps -eLo lwp | while read thread; do echo $thread > tasks ; done
```

5. **cpuset** という名前のパーティションを作成し、使用率の高いタスクを割り当てます。

```
# cd ../partition/
# echo 1 > cpuset.cpu_exclusive
# echo 0 > cpuset.mems
# echo 0 > cpuset.cpus
```

6. シェルを cpuset に設定し、期限ワークロードを開始します。

```
# echo $$ > tasks
# /root/d &
```

この設定では、パーティション化された **cpuset** ディレクトリーに分離されたタスクは、クラスター **cpuset** ディレクトリー内のタスクに干渉しません。これにより、すべてのリアルタイムタスクがスケジューラーの期限に間に合うようになります。

8.4. CPU パフォーマンスのスパイクの低減

レイテンシーが急増する一般的な原因は、複数の CPU がカーネルタイマーティックハンドラー内の共通のロックで競合することです。競合の原因となるロックは、通常 **xtime_lock** です。これは時間管理システムと Read-Copy-Update (RCU) 構造のロックによって使用されます。**skew_tick=1** を使用すると、CPU ごとのタイマーティックをオフセットして別の時間に開始させ、潜在的なロックの競合を回避できます。

skew_tick カーネルコマンドラインパラメーターは、コア数が多く、レイテンシーの影響を受けやすいワークロードを備えた中規模から大規模のシステムでのレイテンシーの変動を防ぐ可能性があります。

前提条件

- 管理者権限がある。

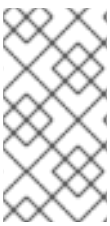
手順

1. **grubby** で **skew_tick=1** パラメーターを有効にします。

```
# grubby --update-kernel=ALL --args="skew_tick=1"
```

2. 変更を有効にするために再起動します。

```
# reboot
```



注記

skew_tick=1 を有効にすると消費電力が大幅に増加します。そのため、レイテンシーの影響を受けやすいリアルタイムワークロードを実行しており、安定したレイテンシーが消費電力よりも重要な考慮事項である場合にのみ、**skew** ブートパラメーターを有効にしてください。

検証

/proc/cmdline ファイルを表示し、**skew_tick=1** が指定されていることを確認します。**/proc/cmdline** ファイルには、カーネルに渡されるパラメーターが表示されます。

- **/proc/cmdline** ファイルの新しい設定を確認します。

```
# cat /proc/cmdline
```

8.5. PC カードデーモンの無効化による CPU 使用量の削減

pcscd デーモンは、並列通信 (PC または PCMCIA) およびスマートカード (SC) リーダーへの接続を管理します。通常 **pcscd** は優先度が低いタスクですが、多くの場合、他のデーモンよりも多くの CPU を使用する場合があります。したがって、背景でさらにノイズが発生することで、リアルタイムのタスクに対してプリエンプションコストが増え、決定論にその他の悪影響を及ぼす可能性があります。

前提条件

- システムの root 権限がある。

手順

1. **pcscd** デーモンのステータスを確認します。

```
# systemctl status pcscd
● pcscd.service - PC/SC Smart Card Daemon
     Loaded: loaded (/usr/lib/systemd/system/pcscd.service; indirect; vendor preset: disabled)
     Active: active (running) since Mon 2021-03-01 17:15:06 IST; 4s ago
TriggeredBy: ● pcscd.socket
       Docs: man:pcscd(8)
   Main PID: 2504609 (pcscd)
      Tasks: 3 (limit: 18732)
     Memory: 1.1M
        CPU: 24ms
     CGroup: /system.slice/pcscd.service
             └─2504609 /usr/sbin/pcscd --foreground --auto-exit
```

   Active パラメーターは、**pcsd** デーモンの状態を表示します。

2. **pcsd** デーモンを実行している場合は停止します。

```
# systemctl stop pcscd
Warning: Stopping pcscd.service, but it can still be activated by:
  pcscd.socket
```

3. システム起動時に **pcsd** デーモンが再起動しないようにシステムを設定します。

```
# systemctl disable pcscd
Removed /etc/systemd/system/sockets.target.wants/pcscd.socket.
```

検証手順

1. **pcscd** デーモンのステータスを確認します。

```
# systemctl status pcscd
● pcscd.service - PC/SC Smart Card Daemon
     Loaded: loaded (/usr/lib/systemd/system/pcscd.service; indirect; vendor preset: disabled)
     Active: inactive (dead) since Mon 2021-03-01 17:10:56 IST; 1min 22s ago
TriggeredBy: ● pcscd.socket
       Docs: man:pcscd(8)
   Main PID: 4494 (code=exited, status=0/SUCCESS)
        CPU: 37ms
```

2. **Active** パラメーターの値が **inactive (dead)** であることを確認します。

第9章 RHEL FOR REAL TIME での MLOCK() システムコールの使用

RHEL for Real-Time メモリーロック (**mlock()**) 関数を使用すると、リアルタイムの呼び出しプロセスで、アドレス空間の指定された範囲をロックまたはロック解除できます。この範囲は、Linux がメモリースペースを交換するときに、ロックされたメモリーをページングすることを防ぎます。ページテーブルエントリーに物理ページを割り当てると、そのページへの参照が高速になります。**mlock()** システムコールには、**mlock()** と **mlockall()** の 2 つの関数が含まれています。同様に、**munlock()** システムコールには、**munlock()** 関数および **munlockall()** 関数が含まれています。

9.1. MLOCK() および MUNLOCK() システムコール

mlock() および **mlockall()** システムコールは、指定されたメモリー範囲をロックし、このメモリーをページングしません。以下は、**mlock()** システムコールグループです。

- **mlock()** システムコール: 指定された範囲のアドレスをロックします。
- **munlock()** システムコール: 指定された範囲のアドレスのロックを解除します。

mlock() システムは、**addr** から始まり、**len** バイトまで続くアドレス範囲のロックページを呼び出します。呼び出しが正常に戻ると、指定されたアドレス範囲の一部を含むすべてのページは、後でロックが解除されるまでメモリーに残ります。

mlockall() システムコールを使用すると、マップされたすべてのページを指定されたアドレス範囲にロックできます。メモリーロックはスタックしません。いくつかの呼び出しによってロックされたページは、単一の **munlock()** システム呼び出しで、指定されたアドレス範囲または領域全体のロックを解除します。**munlockall()** システムコールを使用すると、プログラム空間全体をロック解除できます。

特定の範囲に含まれるページのステータスは、**flags** 引数の値によって異なります。**flags** 引数は 0 または **MLOCK_ONFAULT** です。

メモリーロックは fork によって子プロセスに継承されず、プロセスが終了すると自動的に削除されます。

警告

mlock() システムコールは注意して使用してください。過度に使用すると、メモリー不足 (OOM) エラーが発生する可能性があります。アプリケーションが大きい、またはアプリケーションに大きなデータドメインがあるとき、システムが他のタスクにメモリーを割り当てることができない場合は、**mlock()** 呼び出しによってスラッシングが発生する可能性があります。

リアルタイムプロセスに **mlockall()** 呼び出しを使用する場合は、十分なスタックページを予約してください。

9.2. MLOCK() システムコールを使用したページのロック

リアルタイム **mlock()** システムコールは、**addr** パラメーターを使用してアドレス範囲の開始を指定し、**len** を使用してアドレス空間の長さをバイト単位で定義します。この **alloc_workbuf()** 関数はメモリーバッファーを動的に割り当ててロックします。メモリー割り当ては、**posix_memalig()** 関数によっ

て行われ、メモリー領域がページに整列されます。**free_workbuf()** 関数は、メモリー領域のロックを解除します。

前提条件:

- root 権限、または大きなバッファーで **mlockall()** または **mlock()** を使用するための **CAP_IPC_LOCK** 機能を持っている。

手順

- **mlock()** システムコールでページをロックするには、次のコマンドを実行します。

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

void *alloc_workbuf(size_t size)
{
  void ptr;
  int retval;

  // alloc memory aligned to a page, to prevent two mlock() in the same page.
  retval = posix_memalign(&ptr, (size_t) sysconf(_SC_PAGESIZE), size);

  // return NULL on failure
  if (retval)
    return NULL;

  // lock this buffer into RAM
  if (mlock(ptr, size)) {
    free(ptr);
    return NULL;
  }
    return ptr;
 }

void free_workbuf(void *ptr, size_t size) {
  // unlock the address range
  munlock(ptr, size);

  // free the memory
  free(ptr);
}
```

検証

リアルタイムの **mlock()** および **munlock()** 呼び出しは、成功すると 0 を返します。エラーの場合は、-1 を返し、エラーを示す **errno** を設定します。

9.3. MLOCKALL() システムコールを使用して、マップされたすべてのページをロックする

mlockall() および **munlockall()** システムコールを使用してリアルタイムメモリーをロックおよびロック解除するには、**flags** 引数を 0、もしくは定数 **MCL_CURRENT** または **MCL_FUTURE** のいずれかに設定します。**MCL_FUTURE** を使用すると、ロックされたバイト数が許可された最大数を超えるた

め、**mmap2()**、**sbrk2()**、または **malloc3()** などの将来のシステムコールが失敗する可能性があります。

前提条件

- システムの root 権限がある。

手順

- **mlockall()** および **munlockall()** リアルタイムシステムコールを使用するには、以下を実行します。
  - **mlockall()** システムコールを使用して、マップされたすべてのページをロックします。

```
#include <sys/mman.h>
int mlockall (int flags)
```

  - **munlockall()** システムコールを使用して、マップされたすべてのページのロックを解除します。

```
#include <sys/mman.h>
int munlockall (void)
```

関連情報

- **capabilities(7)** man ページ
- **mlock(2)** man ページ
- **mlock(3)** man ページ
- **move_pages(2)** man ページ
- **posix_memalign(3)** man ページ
- **posix_memalign(3p)** man ページ

9.4. MMAP() システムコールを使用してファイルまたはデバイスをメモリーにマップする

リアルタイムシステムで大量のメモリーを割り当てる場合、メモリー割り当て (**malloc**) メソッドは **mmap()** システムコールを使用してメモリー空間を見つけます。**flags** パラメーターに **MAP_LOCKED** を設定することで、メモリー領域を割り当てて、ロックできます。**mmap()** はページ単位でメモリーを割り当てるため、同じページで 2 つのロックが行われるのを回避し、二重ロックまたは単一ロック解除の問題を防ぎます。

前提条件

- システムの root 権限がある。

手順

- 特定のプロセスアドレス空間をマッピングするには、次のようにします。

```
#include <sys/mman.h>
#include <stdlib.h>

void *alloc_workbuf(size_t size)
{
 void *ptr;

 ptr = mmap(NULL, size, PROT_READ | PROT_WRITE,
           MAP_PRIVATE | MAP_ANONYMOUS | MAP_LOCKED, -1, 0);

 if (ptr == MAP_FAILED)
  return NULL;

 return ptr;
}

void
free_workbuf(void *ptr, size_t size)
{
 munmap(ptr, size);
}
```

検証

- **mmap()** 関数が正常に完了すると、マップされた領域へのポインターが返されます。エラーの場合は、**MAP_FAILED** 値を返し、エラーを示す **errno** を設定します。
- **munmap()** 関数が正常に完了すると、**0** が返されます。エラーの場合は **-1** を返し、エラーを示す **errno** を設定します。

関連情報

- **mmap(2)** man ページ
- **mlockall(2)** man ページ

9.5. MLOCK() システムコールのパラメーター

mlock パラメーターの表で、メモリーロックシステムコールのパラメーターとそれらが実行する機能を一覧にして説明します。

表9.1mlock パラメーター

パラメーター	説明
addr	ロックまたはロック解除するプロセスアドレス空間を指定します。NULL の場合、カーネルはメモリー内のデータのページ整列配置を選択します。**addr** が NULL でない場合、カーネルは近くのページ境界を選択します。これは常に **/proc/sys/vm/mmap_min_addr** ファイルで指定された値以上です。

パラメーター	説明
len	マッピングの長さを指定します。これは 0 より大きくなければなりません。
fd	ファイル記述子を指定します。
prot	**mmap** および **munmap** 呼び出しは、このパラメーターを使用して目的のメモリー保護を定義します。**prot** は、**PROT_EXEC**、**PROT_READ**、**PROT_WRITE**、または **PROT_NONE** 値の 1 つまたは組み合わせを取ります。
flags	同じファイルをマップする他のプロセスへのマッピングの可視性を制御します。**MAP_ANONYMOUS**、**MAP_LOCKED**、**MAP_PRIVATE**、または **MAP_SHARED** のいずれかの値を取ります。
MCL_CURRENT	現在プロセスにマップされているすべてのページをロックします。
MCL_FUTURE	後続のメモリー割り当てをロックするモードを設定します。これらは、増大するヒープとスタックに必要な新しいページ、新しいメモリーマップファイル、または共有メモリー領域である可能性があります。

第10章 ジャーナリングに起因するシステムの速度低下を最小限に抑えるか回避する

ジャーナルの変更がディスクに書き込まれる順序は、到着する順序と異なる場合があります。カーネル I/O システムは、ジャーナルの変更の並べ替えを行い、利用可能なストレージ領域の使用を最適化できます。ジャーナルアクティビティーは、ジャーナルの変更を並べ替え、データおよびメタデータをコミットすることで、システムレイテンシーが発生する可能性があります。その結果、ファイルシステムのジャーナリングによってシステムの速度が低下する可能性があります。

XFS は、RHEL 8 で使用されるデフォルトのファイルシステムです。これはジャーナリングファイルシステムです。**ext2** と呼ばれる古いファイルシステムは、ジャーナリングを使用しません。特にジャーナリングが必要な場合を除き、**ext2** ファイルシステムを検討してください。Red Hat の最良なベンチマーク結果の多くでは、**ext2** ファイルシステムが使用されています。これは、重要な初期チューニングの推奨項目の 1 つです。

XFS のようなジャーナリングファイルシステムは、ファイルが最後にアクセスされた時刻を記録します (**atime** 属性)。ジャーナリングファイルシステムを使用する必要がある場合は、**atime** を無効にすることを検討してください。

10.1. ATIME の無効化

atime 属性を無効にすると、ファイルシステムジャーナルへの書き込み回数が制限されるため、パフォーマンスが向上し、電力使用量が減少します。

手順

1. 任意のテキストエディターを使用して **/etc/fstab** ファイルを開き、ルートマウントポイントのエントリーを見つけます。

```
/dev/mapper/rhel-root      /      xfs    defaults…
```

2. オプションのセクションを編集して、**noatime** および **nodiratime** という用語を追加します。**noatime** オプションは、ファイルの読み取り時のアクセスタイムスタンプの更新を阻止し、**nodiratime** オプションは、ディレクトリーの inode へのアクセス時刻の更新を停止します。

```
/dev/mapper/rhel-root      /      xfs    noatime,nodiratime…
```

重要

一部のアプリケーションは、更新される **atime** に依存します。したがって、このオプションは、このようなアプリケーションが使用されていないシステムでのみ妥当です。

または、前のアクセス時刻が現在の変更時刻よりも古い場合にのみ、アクセス時刻が更新されるようにする **relatime** マウントオプションを使用することができます。

10.2. 関連情報

- **mkfs.ext2(8)** の man ページ
- **mkfs.xfs(8)** の man ページ
- **mount(8)** の man ページ

第11章 レイテンシーの影響を受けるワークロードのグラフィックコンソール出力の無効化

カーネルは、起動直後に **printk()** にメッセージを渡し始めます。カーネルはログファイルにメッセージを送信して、ヘッドレスサーバーに接続されているモニターがない場合でも、グラフィックコンソールにも表示します。

一部のシステムでは、グラフィックコンソールに送信された出力が原因でパイプラインが停滞する可能性があります。これにより、データ転送の待機中にタスクの実行が遅延する可能性があります。たとえば、**teletype0 (/dev/tty0)** に送信された出力が原因で、システムの停滞が発生する可能性があります。

予期しない停滞を防ぐため、グラフィックコンソールに送信される情報を以下で制限または無効にできます。

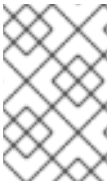
- **tty0** 定義を削除する。
- コンソール定義の順序を変更する。
- ほとんどの **printk()** 関数をオフにし、必ず **ignore_loglevel** カーネルパラメーターを **not configured** に設定する。

グラフィックコンソール出力のログオンを無効にし、グラフィックコンソールに出力されるメッセージを制御することにより、レイテンシーの影響を受けやすいワークロードでレイテンシーを改善できます。

11.1. グラフィックコンソールのグラフィックアダプターへのロギングの無効化

デフォルトのカーネルコンソールである **teletype** (**tty**) は、入力データをシステムに渡してグラフィックコンソールに出力情報を表示することで、システムとの対話を可能にします。

グラフィックコンソールを設定しないと、グラフィックアダプターにログの記録ができなくなります。これにより、**tty0** はシステムで利用できず、グラフィックコンソールでのメッセージの出力を無効にするのに役立ちます。



注記

グラフィックコンソールの出力を無効にしても、情報は削除されません。この情報はシステムログに出力され、**journalctl** ユーティリティーまたは **dmesg** ユーティリティーを使用して情報にアクセスできます。

手順

- カーネル設定から **console=tty0** オプションを削除します。

```
# grubby --update-kernel=ALL --remove-args="console=tty0"
```

11.2. メッセージのグラフィックコンソールへの出力の無効化

必要なログレベルを **/proc/sys/kernel/printk** ファイルに設定すると、グラフィックコンソールに送信される出力メッセージの量を制御できます。

手順

1. 現在のコンソールログレベルを表示します。

```
$ cat /proc/sys/kernel/printk
  7    4    1    7
```

このコマンドは、システムログレベルの現在の設定を出力します。数字は、システムロガーの current、default、minimum、および boot-default の値に対応します。

2. **/proc/sys/kernel/printk** ファイルで希望のログレベルを設定します。

```
$ echo “1” > /proc/sys/kernel/printk
```

コマンドは、現在のコンソールログレベルを変更します。たとえば、ログレベル 1 を設定すると、警告メッセージのみが出力され、グラフィックコンソールに他のメッセージが表示されないようになります。

第12章 アプリケーションのニーズを満たすためのシステムクロックの管理

NUMA や SMP などのマルチプロセッサーシステムには、複数のハードウェアクロックインスタンスがあります。起動時に、カーネルは利用可能なクロックソースを検出し、使用するクロックソースを選択します。パフォーマンスを改善するために、リアルタイムシステムの最小要件を満たすために使用されるクロックソースを変更できます。

12.1. ハードウェアクロック

Non-Uniform Memory Access (NUMA) や Symmetric multiprocessing (SMP) などのマルチプロセッサーシステムに見られるクロックソースの複数のインスタンスは、それらの間で相互作用し、CPU 周波数スケーリングまたはエネルギーエコノミーモードへの移行などのシステムイベントへの反応により、それらがリアルタイムカーネルに適したクロックソースであるかどうかを判断します。

推奨されるクロックソースは Time Stamp Counter (TSC) です。TSC が利用できない場合は、High Precision Event Timer (HPET) が 2 番目に最適なオプションとなります。ただし、すべてのシステムに HPET クロックがあるわけではなく、一部の HPET クロックは信頼できない可能性があります。

TSC および HPET がない場合のオプションとして、ACPI Power Management Timer (ACPI_PM)、Programmable Interval Timer (PIT)、Real Time Clock (RTC) などがあります。最後の 2 つのオプションは、読み取るのにコストがかかるか、分解能 (時間粒度) が低いかのどちらかであるため、リアルタイムカーネルでの使用は準最適となります。

12.2. システムで利用可能なクロックソースの表示

システムで利用可能なクロックソースのリストは、**/sys/devices/system/clocksource/clocksource0/available_clocksource** ファイルにあります。

手順

- **available_clocksource** ファイルを表示します。

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

この例では、システムで利用可能なクロックソースは TSC、HPET、および ACPI_PM です。

12.3. 現在使用中のクロックソースの表示

システムで現在使用されているクロックソースは、**/sys/devices/system/clocksource/clocksource0/current_clocksource** ファイルに保存されています。

手順

- **current_clocksource** ファイルを表示します。

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

この例では、システムの現在のクロックソースは TSC です。

12.4. 使用するクロックソースの一時的な変更

クロックの既知の問題により、システムのメインアプリケーションに最適なクロックが使用されないことがあります。問題のあるすべてのクロックを除外した後、システムに残ったハードウェアクロックが、リアルタイムシステムの最低要件を満たせないことがあります。

重要なアプリケーションの要件は、システムごとに異なります。そのため、各アプリケーション、したがって各システムに適したクロックも異なります。一部のアプリケーションはクロックの分解能に依存し、信頼できるナノ秒の読み取りを提供するクロックの方が適しています。また、クロックを読み取るアプリケーションは、読み取りコスト (読み取り要求と結果の間隔) の小さいクロックからメリットを得ることができます。

これらのケースでは、カーネルが選択したクロックをオーバーライドできます。ただし、このオーバーライドの副次的な影響を理解し、そのハードウェアクロックの既知の欠点をトリガーしない環境を作成できる場合に限ります。

重要

カーネルは、利用可能な最適なクロックソースを自動的に選択します。選択したクロックソースのオーバーライドは、影響が明確に理解されない限り推奨されません。

前提条件

- システムの root 権限がある。

手順

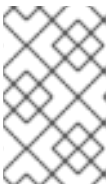
1. 利用可能なクロックソースを表示します。

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

   例として、システムで利用可能なクロックソースが TSC、HPET、および ACPI_PM であると考えてください。

2. 使用するクロックソースの名前を **/sys/devices/system/clocksource/clocksource0/current_clocksource** ファイルに書き込みます。

```
# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
```



注記

変更は現在使用中のクロックソースに適用されます。システムの再起動後、デフォルトのクロックが使用されます。変更を永続化させるには、カーネルチューニングパラメーターの変更の永続化 を参照してください。

検証手順

- **current_clocksource** ファイルを表示して、現在のクロックソースが指定されたクロックソースであることを確認します。

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
hpet
```

この例では、システムの現在のクロックソースとして HPET を使用します。

12.5. ハードウェアクロックソースの読み込みコストの比較

システムのクロックの速度を比較できます。TSC からの読み取りは、プロセッサーからレジスターを読み取ることを意味します。HPET クロックからの読み取りには、メモリーエリアを読み取る必要があります。TSC からの読み取りがより高速です。毎秒大量のメッセージのタイムスタンプ処理を行う場合、パフォーマンスが大幅に向上します。

前提条件

- システムの root 権限がある。
- **clock_timing** プログラムがシステム上にある。詳細は、clock_timing プログラム を参照してください。

手順

1. **clock_timing** プログラムが保存されるディレクトリーに移動します。

```
# cd clock_test
```

2. システムで利用可能なクロックソースを表示します。

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
```

この例では、システムで利用可能なクロックソースは **TSC**、**HPET**、および **ACPI_PM** になります。

3. 現在使用中のクロックソースを表示します。

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

この例では、システムの現在のクロックソースは **TSC** です。

4. ./**clock_timing** プログラムと共に **time** ユーティリティーを実行します。この出力は、クロックソースを 1,000 万回読み込むために必要な期間を表示します。

```
# time ./clock_timing

real 0m0.601s
user 0m0.592s
sys 0m0.002s
```

この例は、以下のパラメーターを示しています。

- **real**: プログラムの呼び出しから始まり、プロセスが終了するまでに費やした合計時間。**real** には、ユーザーとカーネル時間が含まれており、通常は後者 2 つの合計値よりも大きくなります。このプロセスが、優先度の高いアプリケーションや、ハードウェア割り込み (IRQ) などのシステムイベントによって中断される場合、待機に費やされたこの時間も **real** として計算されます。

- **user**: カーネルの介入を必要としないタスクを実行するプロセスがユーザー空間で費やした時間。
- **sys**: ユーザープロセスで必要なタスクの実行中にカーネルが費やした時間。これらのタスクには、ファイルのオープン、ファイルまたは I/O ポートの読み取りおよび書き込み、メモリーの割り当て、スレッドの作成、およびネットワーク関連のアクティビティーが含まれます。

5. **/sys/devices/system/clocksource/clocksource0/current_clocksource** ファイルに、テストする次のクロックソースの名前を書き込みます。

```
# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
```

この例では、現在のクロックソースが **HPET** に変更されています。

6. 利用可能なすべてのクロックソースに対して、ステップ 4 と 5 を繰り返します。
7. 利用可能なすべてのクロックソースについて、ステップ 4 の結果を比較します。

関連情報

- **time(1)** の man ページ

12.6. OPTERON CPU での TSC タイマーの同期

AMD64 Opteron プロセッサーの現行世代は、大きな **gettimeofday** スキューの影響を受けやすい可能性があります。このスキューは、**cpufreq** および **Time Stamp Counter** (TSC) の両方が使用されている場合に発生します。RHEL for Real Time は、すべてのプロセッサーが同じ頻度に同時に変更することで、このスキューを防ぐ方法を提供します。そのため、1 つのプロセッサーの TSC は、別のプロセッサーの TSC とは異なる速度で増加することはありません。

前提条件

- システムの root 権限がある。

手順

1. **clocksource=tsc** および **powernow-k8.tscsync=1** カーネルオプションを有効にします。

```
# grubby --update-kernel=ALL --args="clocksource=tsc powernow-k8.tscsync=1"
```

これにより、TSC の使用が強制され、同時にコアプロセッサーの周波数遷移が有効になります。

2. マシンを再起動します。

関連情報

- **gettimeofday (2)** man ページ

12.7. CLOCK_TIMING プログラム

clock_timing プログラムは、現在のクロックソースを 1,000 万回読み取ります。 **time** ユーティリティーと共に、これを行うために必要な時間を測定します。

手順

clock_timing プログラムを作成するには、以下を実行します。

1. プログラムファイルのディレクトリーを作成します。

```
$ mkdir clock_test
```

2. 作成したディレクトリーに移動します。

```
$ cd clock_test
```

3. ソースファイルを作成してテキストエディターで開きます。

```
$ {EDITOR} clock_timing.c
```

4. ファイルに以下のコマンドを入力します。

```
#include <time.h>
void main()
{
 int rc;
 long i;
 struct timespec ts;

        for(i=0; i<10000000; i++) {
  rc = clock_gettime(CLOCK_MONOTONIC, &ts);
 }
}
```

5. ファイルを保存して、エディターを終了します。

6. ファイルをコンパイルします。

```
$ gcc clock_timing.c -o clock_timing -lrt
```

clock_timing プログラムの準備ができ、保存されているディレクトリーから実行できます。

第13章 電源管理移行の制御

電源管理の移行を制御すると、レイテンシーが改善されます。

前提条件

- システムの root 権限がある。

13.1. 省電力の状態

最新のプロセッサーは、低い省電力状態から高い省電力状態 (C-state) にアクティブに移行します。ただし、高い省電力状態から稼働状態に戻ると、リアルタイムアプリケーションの理想よりも多くの時間を消費してしまいます。アプリケーションは Power Management Quality of Service (PM QoS) インターフェイスを使用して、これらの移行を防ぐことができます。

PM QoS インターフェイスを使用すると、システムは **idle=poll** および **processor.max_cstate=1** パラメーターの動作をエミュレートできますが、省電力の状態をより詳細に制御できます。**idle=poll** は、プロセッサーが **idle** 状態になることを防ぎます。**processor.max_cstate=1** は、プロセッサーがより深い C-state (省電力モード) に入ることを阻止します。

アプリケーションが **/dev/cpu_dma_latency** ファイルを開いたままにすると、PM QoS インターフェイスはプロセッサーが深いスリープ状態に入ることを阻止します。これにより、終了する際に予期せぬレイテンシーが発生します。ファイルが閉じられると、システムは省電力状態に戻ります。

13.2. 電源管理状態の設定

次のいずれかの方法で電源管理の状態を設定することで、電源管理の移行を制御できます。

- **/dev/cpu_dma_latency** ファイルに値を書き込んで、プロセスの最大応答時間をマイクロ秒単位で変更し、低レイテンシーが必要になるまでファイル記述子を開いたままにします。
- アプリケーションまたはスクリプトで **/dev/cpu_dma_latency** ファイルを参照します。

前提条件

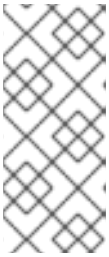
- 管理者権限がある。

手順

- **/dev/cpu_dma_latency** に最大応答時間 (マイクロ秒単位) を表す 32 ビットの数値を書き込むことでレイテンシー許容度を指定し、低レイテンシー操作によってファイル記述子を開いたままにします。値 **0** は C-状態を完全に無効にします。
  以下に例を示します。

```
import os
import os.path
import signal
import sys
if not os.path.exists('/dev/cpu_dma_latency'):
  print("no PM QOS interface on this system!")
  sys.exit(1)
fd = os.open('/dev/cpu_dma_latency', os.O_WRONLY)
   os.write(fd, b'\0\0\0\0')
   print("Press ^C to close /dev/cpu_dma_latency and exit")
```

```
    signal.pause()
except KeyboardInterrupt:
    print("closing /dev/cpu_dma_latency")
    os.close(fd)
    sys.exit(0)
```



注記

Power Management Quality of Service (**pm_qos**) インターフェイスは、ファイル記述子が開いている間のみアクティブになります。したがって、**/dev/cpu_dma_latency** へのアクセスに使用するスクリプトまたはプログラムは、電源状態の移行が許可されるまでファイルを開いたままにしておく必要があります。

第14章 割り込みとユーザープロセスを分離してシステムレイテンシーを最小限に抑える

リアルタイム環境では、さまざまなイベントに応答する際にレイテンシーを最小限に抑える必要があります。これを行うには、割り込み (IRQ) とさまざまな専用 CPU 上のユーザープロセスを相互に分離します。

14.1. 割り込みおよびプロセスバインディング

割り込み (IRQ) をさまざまな専用 CPU 上のユーザープロセスから分離することで、リアルタイム環境でのレイテンシーを最小限に抑えるか、なくすことができます。

通常、割り込みは CPU 間で均等に共有されます。これにより、CPU が新しいデータおよび命令キャッシュを書き込む必要があるときに、割り込み処理が遅延する場合があります。これらの割り込みの遅延は、同じ CPU で実行されている他の処理との競合を引き起こす可能性があります。

タイムクリティカルな割り込みおよびプロセスを特定の CPU (または CPU の範囲) に割り当てることができます。これにより、この割り込みを処理するコードおよびデータ構造がプロセッサーおよび命令キャッシュにある可能性が非常に高くなります。その結果、専用のプロセスはできるだけ迅速に実行でき、他のすべてのタイムクリティカルでないプロセスは他の CPU で実行されます。これは、関連する速度がメモリーおよび使用可能なペリフェラルバス帯域幅の限界に近いか限界に達している場合に、特に重要になる可能性があります。メモリーがプロセッサーキャッシュにフェッチされるのを待つと、全体的な処理時間と決定論は著しく影響を受けます。

実際には、最適なパフォーマンスはアプリケーションによってまったく異なります。たとえば、同様の機能を持つアプリケーションを異なる企業向けにチューニングする場合、全く異なる最適なパフォーマンスチューニングが必要になります。

- ある企業では、4 つの CPU のうち 2 つをオペレーティングシステムの機能と割り込み処理用に分離した時に、最適な結果が得られました。残りの 2 つの CPU は、純粋にアプリケーション処理専用でした。
- 別の会社では、ネットワーク関連のアプリケーションプロセスを、ネットワークデバイスドライバーの割り込みを処理する単一の CPU にバインドしたときに、最適な決定論が得られました。

重要

プロセスを CPU にバインドするには、通常、特定の CPU または CPU の範囲の CPU マスクを把握する必要があります。CPU マスクは通常、使用するコマンドに応じて、32 ビットのビットマスク、10 進数、または 16 進数で表されます。

表14.1 特定の CPU の CPU マスクの例

CPU	ビットマスク	10 進数	16 進数
0	00000000000000000000000000000001	1	0x00000001
0, 1	00000000000000000000000000000011	3	0x00000011

14.2. IRQBALANCE デーモンの無効化

irqbalance デーモンはデフォルトで有効になっており、複数の CPU に対して定期的に割り込みの均等な処理を強制します。ただし、リアルタイムのデプロイメントでは、アプリケーションは通常特定の CPU にバインドされているため、**irqbalance** は必要ありません。

手順

1. **irqbalance** のステータスを確認します。

```
# systemctl status irqbalance
irqbalance.service - irqbalance daemon
   Loaded: loaded (/usr/lib/systemd/system/irqbalance.service; enabled)
   Active: active (running) …
```

2. **irqbalance** が実行されている場合、これを無効にして停止します。

```
# systemctl disable irqbalance
# systemctl stop irqbalance
```

検証

- **irqbalance** ステータスが非アクティブであることを確認します。

```
# systemctl status irqbalance
```

14.3. IRQ バランスからの CPU の除外

IRQ バランシングサービスを使用して、割り込み (IRQ) バランシングを考慮する際に除外する CPU を指定できます。**/etc/sysconfig/irqbalance** 設定ファイルの **IRQBALANCE_BANNED_CPUS** パラメーターは、この設定を制御します。パラメーターの値は 64 ビットの 16 進数ビットマスクで、マスクの各ビットは CPU コアを表します。

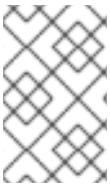
手順

1. 任意のテキストエディターで **/etc/sysconfig/irqbalance** を開き、**IRQBALANCE_BANNED_CPUS** というファイルのセクションを見つけます。

```
# IRQBALANCE_BANNED_CPUS
# 64 bit bitmask which allows you to indicate which cpu's should
# be skipped when reblancing irqs. Cpu numbers which have their
# corresponding bits set to one in this mask will not have any
# irq's assigned to them on rebalance
#
#IRQBALANCE_BANNED_CPUS=
```

2. **IRQBALANCE_BANNED_CPUS** 変数のコメントを解除します。
3. 適切なビットマスクを入力し、IRQ バランスメカニズムで無視される CPU を指定します。
4. ファイルを保存してから閉じます。
5. 変更を有効にするには、**irqbalance** サービスを再起動します。

```
# systemctl restart irqbalance
```

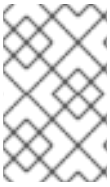


注記

最大 64 個の CPU コアを持つシステムを実行している場合は、それぞれ 8 桁の 16 進数のグループをコンマで区切ります。例:
IRQBALANCE_BANNED_CPUS=00000001,0000ff00

表14.2 例

CPU	ビットマスク
0	00000001
8 - 15	0000ff00
8 - 15, 33	00000002,0000ff00



注記

IRQBALANCE_BANNED_CPUS が **/etc/sysconfig/irqbalance** に設定されていない場合、RHEL 7.2 以降では、**irqbalance** ユーティリティーは **isolcpus** カーネルパラメーターを介して分離した CPU コアの IRQ を自動的に回避します。

14.4. 個々の IRQ への CPU アフィニティーの手動割り当て

CPU アフィニティーを割り当てると、指定した CPU または CPU の範囲にプロセスとスレッドをバインドおよびバインド解除できます。これにより、キャッシュの問題を減らすことができます。

手順

1. **/proc/interrupts** ファイルを表示して、各デバイスで使用されている IRQ を確認します。

```
# cat /proc/interrupts
```

各行には、IRQ 番号、各 CPU で発生した割り込みの数、IRQ タイプ、および説明が表示されます。

```
           CPU0       CPU1
  0:   26575949         11         IO-APIC-edge  timer
  1:         14          7         IO-APIC-edge  i8042
```

2. 特定の IRQ の **smp_affinity** エントリーに CPU マスクを書き込みます。CPU マスクは、16 進数で表記する必要があります。
たとえば、以下のコマンドは、IRQ 番号 142 が CPU 0 でのみ実行されるように指示します。

```
# echo 1 > /proc/irq/142/smp_affinity
```

この変更は、割り込みが発生した場合にのみ有効になります。

検証手順

1. 指定の割り込みをトリガーするアクティビティーを実行します。

2. **/proc/interrupts** で変更を確認します。
   設定された IRQ の指定された CPU の割り込みの数は増加し、指定されたアフィニティー外の CPU で設定された IRQ の割り込みの数は増加しませんでした。

14.5. TASKSET ユーティリティーを使用したプロセスの CPU へのバインド

taskset ユーティリティーは、タスクのプロセス ID (PID) を使用して、その CPU アフィニティーを表示または設定します。このユーティリティーを使用して、選択した CPU アフィニティーでコマンドを実行できます。

アフィニティーを設定するには、CPU マスクを 10 進数または 16 進数にする必要があります。mask 引数は、変更されるコマンドまたは PID に対して有効な CPU コアを指定する **bitmask** です。

重要

taskset` ユーティリティーは NUMA (Non-Uniform Memory Access) システムで動作しますが、ユーザーが CPU および最も近い NUMA メモリーノードにスレッドをバインドすることはできません。このようなシステムでは、taskset は推奨されるツールではなく、その高度な機能を使用するには、代わりに **numactl** ユーティリティーを使用する必要があります。

詳細は、**numactl(8)** の man ページを参照してください。

手順

- 必要なオプションおよび引数を指定して **taskset** を実行します。
  - CPU マスクの代わりに -c パラメーターを使用して CPU リストを指定できます。この例では、**my_embedded_process** は、CPU 0、4、7 - 11 でのみ実行するように指示されています。

    ```
    # taskset -c 0,4,7-11 /usr/local/bin/my_embedded_process
    ```

    ほとんどの場合、この呼び出しは便利です。

  - 現在実行されていないプロセスのアフィニティーを設定するには、**taskset** を使用して、CPU マスクとプロセスを指定します。
    この例では、**my_embedded_process** は、CPU 3 のみを使用するように指示されています (CPU マスクの 10 進数バージョンを使用)。

    ```
    # taskset 8 /usr/local/bin/my_embedded_process
    ```

  - ビットマスクで複数の CPU を指定できます。この例では、**my_embedded_process** は、プロセッサー 4、5、6、および 7 で実行するように指示されています (CPU マスクの 16 進数バージョンを使用)。

    ```
    # taskset 0xF0 /usr/local/bin/my_embedded_process
    ```

  - 変更するプロセスの CPU マスクと PID を指定して、**-p** (**--pid**) オプションを使用することにより、すでに実行されているプロセスの CPU アフィニティーを設定できます。この例では、PID が 7013 のプロセスは CPU 0 でのみ実行するように指示されています。

    ```
    # taskset -p 1 7013
    ```

注記

リスト表示されているオプションを組み合わせることができます。

関連情報

- **taskset(1)** の man ページ
- **numactl(8)** の man ページ

第15章 OUT OF MEMORY (OOM) 状態の管理

Out-of-memory (OOM) は、スワップ領域を含む利用可能なメモリーがすべて割り当てられているコンピューティング状態です。通常、これによりシステムがパニックになり、想定どおりに機能しなくなります。以下で説明する手順は、システムの OOM 状態を回避するのに役立ちます。

前提条件

- システムの root 権限がある。

15.1. OUT OF MEMORY 値の変更

/proc/sys/vm/panic_on_oom ファイルには、Out of Memory (OOM) の動作を制御するスイッチである値が含まれます。ファイルに **1** が含まれる場合、カーネルは OOM でパニックになり、期待どおりに機能しなくなります。

デフォルト値は **0** で、システムが OOM 状態の場合に **oom_killer()** 関数を呼び出すようカーネルに指示します。通常、**oom_killer()** は不要なプロセスを終了します。これにより、システムの存続が可能になります。

/proc/sys/vm/panic_on_oom の値を変更できます。

手順

1. **/proc/sys/vm/panic_on_oom** の現在の値を表示します。

```
# cat /proc/sys/vm/panic_on_oom
0
```

   /proc/sys/vm/panic_on_oom の値を変更するには、次のコマンドを実行します。

2. echo コマンドを使用して、新しい値を **/proc/sys/vm/panic_on_oom** に代入します。

```
# echo 1 > /proc/sys/vm/panic_on_oom
```

注記

OOM 時に Real-Time カーネルをパニック状態にすることが推奨されます (**1**)。そうしないと、システムが OOM 状態になった場合、その状態は決定論的ではなくなります。

検証手順

1. **/proc/sys/vm/panic_on_oom** の値を表示します。

```
# cat /proc/sys/vm/panic_on_oom
1
```

2. 表示される値が指定された値と一致していることを確認します。

15.2. OUT OF MEMORY 状態のときに強制終了するプロセスの優先順位付け

oom_killer() 関数で終了するプロセスに優先順位を付けることができます。これにより、優先順位の高いプロセスが OOM 状態の間も実行され続けることが保証されます。各プロセスには **/proc/PID** ディレクトリーがあります。各ディレクトリーには、以下のファイルが含まれます。

- **oom_adj**: **oom_adj** の有効なスコアは、-16 から +15 の範囲にあります。この値は、他の要因の中でも特に、プロセスの実行時間も考慮に入れるアルゴリズムを使用して、プロセスのパフォーマンスフットプリントを計算するために使用されます。
- **oom_score**: **oom_adj** の値を使用して計算されたアルゴリズムの結果が含まれます。

メモリー不足の状態では、**oom_killer()** 関数は **oom_score** が最も高いプロセスを終了します。

プロセスの **oom_adj** ファイルを編集して、終了するプロセスに優先順位を付けることができます。

前提条件

- 優先するプロセスのプロセス ID (PID) を把握している。

手順

1. プロセスの現在の **oom_score** を表示します。

```
# cat /proc/12465/oom_score
79872
```

2. プロセスの **oom_adj** の内容を表示します。

```
# cat /proc/12465/oom_adj
13
```

3. **oom_adj** の値を編集します。

```
# echo -5 > /proc/12465/oom_adj
```

検証手順

1. プロセスの現在の **oom_score** を表示します。

```
# cat /proc/12465/oom_score
78
```

2. 表示される値が以前の値よりも小さいことを確認します。

15.3. プロセスの OUT OF MEMORY KILLER の無効化

プロセスの **oom_killer()** 関数を無効にするには、**oom_adj** を **-17** の予約値に設定します。これにより、OOM 状態でもプロセスが存続します。

手順

- **oom_adj** の値を **-17** に設定します。

```
# echo -17 > /proc/12465/oom_adj
```

検証手順

1. プロセスの現在の **oom_score** を表示します。

```
# cat /proc/12465/oom_score
0
```

2. 表示される値が **0** であることを確認します。

第16章 TUNA CLI を使用したレイテンシーの向上

tuna CLI を使用して、システムのレイテンシーを改善できます。**tuna** コマンドで使用されるオプションにより、レイテンシーを改善するために呼び出されるメソッドが決定されます。**tuna** CLI を使用すると、次の機能を実行できます。

- スケジューラーの調整パラメーターを変更する
- IRQ ハンドラーとスレッドの優先順位を調整する
- CPU コアとソケットを分離する
- 複雑さを軽減してタスクを調整する

16.1. 前提条件

- **tuna** および **python-linux-procfs** パッケージがインストールされている。
- システムの root 権限がある。

16.2. TUNA CLI

tuna コマンドラインインターフェイス (CLI) は、システムのチューニング変更を行うためのツールです。

tuna ツールは実行中のシステムで使用するように設計されており、変更がすぐに反映されます。これにより、アプリケーション固有の測定ツールは、変更が加えられた直後にシステムパフォーマンスを確認および分析できます。

tuna CLI には、アクションオプションと修飾子オプションの両方があります。修飾子オプションは、変更するアクションの前にコマンドラインで指定する必要があります。すべての修飾子オプションは、修飾子オプションがオーバーライドされるまで続くアクションに適用されます。

16.3. TUNA CLI を使用した CPU の分離

tuna CLI を使用して、割り込み (IRQ) をさまざまな専用 CPU 上のユーザープロセスから分離し、リアルタイム環境でのレイテンシーを最小限に抑えることができます。CPU の分離に関する詳細は、割り込みおよびプロセスバインディング を参照してください。

前提条件

- **tuna** および **python-linux-procfs** パッケージがインストールされている。
- システムの root 権限がある。

手順

- 1 つ以上の CPU を分離します。

```
# tuna --cpus=<cpu_list> --isolate
```

cpu_list は、分離する CPU のコンマ区切りリストまたは範囲です。

以下に例を示します。

```
# tuna --cpus=0,1 --isolate
```

16.4. TUNA CLI を使用した指定の CPU への割り込みの移動

tuna CLI を使用して、割り込み (IRQ) を専用の CPU に移動し、リアルタイム環境でのレイテンシーを最小限に抑えるか、なくすことができます。IRQ の移動に関する詳細は、割り込みおよびプロセスバインディング を参照してください。

前提条件

- **tuna** および **python-linux-procfs** パッケージがインストールされている。
- システムの root 権限がある。

手順

1. IRQ のリストが割り当てられている CPU をリスト表示します。

```
# tuna --irqs=<irq_list> --show_irqs
```

irq_list は、割り当てられている CPU を一覧表示する IRQ のコンマ区切りリストです。

以下に例を示します。

```
# tuna --irqs=128 --show_irqs
   # users         affinity
 128 iwlwifi         0,1,2,3
```

2. IRQ のリストを CPU のリストに割り当てます。

```
# tuna --irqs=irq_list --cpus=<cpu_list> --move
```

irq_list は割り当てる IRQ のコンマ区切りリストで、 **cpu_list** はその割り当て先の CPU のコンマ区切りリストまたは範囲です。

以下に例を示します。

```
# tuna --irqs=128 --cpus=3 --move
```

検証

- IRQ を指定された CPU に移動する前後で、選択された IRQ の状態を比較します。

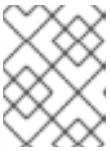
```
# tuna --irqs=128 --show_irqs
   # users         affinity
 128 iwlwifi         3
```

16.5. TUNA CLI を使用したプロセススケジューリングポリシーおよび優先順位の変更

tuna CLI を使用して、プロセススケジューリングポリシーおよび優先順位を変更できます。

前提条件

- **tuna** および **python-linux-procfs** パッケージがインストールされている。
- システムの root 権限がある。



注記

OTHER および **BATCH** スケジューリングポリシーの割り当てには、root 権限は必要ありません。

手順

1. スレッドの情報を表示します。

```
# tuna --threads=<thread_list> --show_threads
```

thread_list は、表示するプロセスのコンマ区切りリストです。

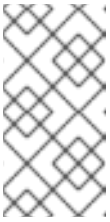
以下に例を示します。

```
# tuna --threads=rngd --show_threads
                    thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
 3571   OTHER     0  0,1,2,3    167697          134           rngd
```

2. プロセススケジューリングポリシーとスレッドの優先度を変更します。

```
# tuna --threads=<thread_list> --priority scheduling_policy:priority_number
```

- **thread_list** は、スケジューリングポリシーおよび優先度を表示するプロセスのコンマ区切りリストです。
- **scheduling_policy** は以下のいずれかになります。
  - OTHER
  - BATCH
  - **FIFO**: First In First Out (先入れ先出し)
  - **Rr**: Round Robin (ラウンドロビン)
- **priority_number** は 0 から 99 までの優先順位で、 **0** は優先度がなく、**99** は優先度が最も高くなります。



注記

OTHER および **BATCH** スケジューリングポリシーでは、優先度を指定する必要はありません。さらに、唯一の有効な優先度 (指定する場合) は **0** です。**FIFO** および **RR** スケジューリングポリシーには、**1** 以上の優先度が必要です。

以下に例を示します。

```
# tuna --threads=rngd --priority FIFO:1
```

検証

- スレッドの情報を表示して、情報が変更されることを確認します。

```
# *tuna --threads=rngd --show_threads*
                    thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
 3571   FIFO     1  0,1,2,3    167697          134             rngd
```

第17章 スケジューラーの優先順位の設定

Red Hat Enterprise Linux for Real Time カーネルを使用すると、スケジューラーの優先度を詳細に制御できます。また、アプリケーションレベルのプログラムをカーネルスレッドよりも優先度の高い状態でスケジュールすることもできます。

警告

スケジューラーの優先順位を設定すると、悪影響が出る可能性があり、重要なカーネルプロセスが想定どおりに実行されない場合、システムが応答しなくなることや、予期せぬ動作を示すことがあります。最終的には、正しい設定はワークロードによって異なります。

17.1. スレッドのスケジューリングの優先度の表示

スレッドの優先度は、**0** (最低優先度) から **99** (最高優先度) まで、一連のレベルを使用して設定されます。**systemd** サービスマネージャーは、カーネルの起動後に、スレッドのデフォルトの優先度を変更するのに使用できます。

手順

- 実行中のスレッドのスケジューリング優先度を表示するには、tuna ユーティリティーを使用します。

```
# tuna --show_threads
                    thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
2      OTHER     0    0xfff       451            3        kthreadd
3       FIFO     1        0     46395            2     ksoftirqd/0
5      OTHER     0        0        11            1    kworker/0:0H
7       FIFO    99        0         9            1   posixcputmr/0
...[output truncated]...
```

17.2. 起動時のサービスの優先度の変更

systemd を使用すると、システムの起動時に起動するサービスのリアルタイムの優先度を設定できます。

ユニット設定ディレクティブを使用して、起動プロセス時のサービスの優先度を変更します。ブートプロセスの優先順位の変更は、**/etc/systemd/system/service.system.d/priority.conf** の service セクションにある以下のディレクティブを使用して行います。

CPUSchedulingPolicy=

実行されるプロセスの CPU スケジューリングポリシーを設定します。Linux で利用可能なスケジューリングクラスのいずれかを設定できます。

- **other**
- **batch**

- **idle**
- **fifo**
- **rr**

CPUSchedulingPriority=

実行されるプロセスの CPU スケジューリングの優先度を設定します。設定可能な優先度の範囲は、選択した CPU スケジューリングポリシーにより異なります。リアルタイムスケジューリングポリシーでは、**1** (最低優先度) から **99** (最高優先度) までの整数を使用できます。

前提条件

- 管理者権限がある。
- システムの起動時に実行するサービス。

手順

既存のサービスに対して以下を実行します。

1. サービスの補助サービス設定ディレクトリーファイルを作成します。

```
# cat <<-EOF > /etc/systemd/system/mcelog.service.d/priority.conf
```

2. ファイルの **[SERVICE]** セクションに、スケジューリングポリシーと優先度を追加します。以下に例を示します。

```
[Service]
CPUSchedulingPolicy=fifo
CPUSchedulingPriority=20
EOF
```

3. **systemd** スクリプトの設定を再読み込みします。

```
# systemctl daemon-reload
```

4. サービスを再起動します。

```
# systemctl restart mcelog
```

検証

- サービスの優先度を表示します。

```
$ tuna -t mcelog -P
```

この出力は、設定されているサービスの優先度を示します。

以下に例を示します。

```
                  thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
826     FIFO    20  0,1,2,3        13            0          mcelog
```

関連情報

- systemd ユニットファイルの使用.

17.3. サービスの CPU 使用率の設定

systemd を使用して、サービスを実行できる CPU を指定できます。

前提条件

- 管理者権限がある。

手順

1. サービスの補助サービス設定ディレクトリーファイルを作成します。

```
# md sscd
```

2. **[SERVICE]** セクションの **CPUAffinity** 属性を使用して、サービスに使用する CPU をファイルに追加します。
以下に例を示します。

```
[Service]
CPUAffinity=0,1
EOF
```

3. systemd スクリプトの設定を再読み込みします。

```
# systemctl daemon-reload
```

4. サービスを再起動します。

```
# systemctl restart service
```

検証

- 指定したサービスを実行可能な CPU を表示します。

```
$ tuna -t mcelog -P
```

service は、指定したサービスに置き換えます。

以下の出力は、**mcelog** が CPU 0 および 1 に制限されていることを示しています。

```
                  thread       ctxt_switches
  pid SCHED_ rtpri affinity voluntary nonvoluntary            cmd
12954   FIFO    20      0,1         2            1          mcelog
```

17.4. 優先順位マップ

スケジューラーの優先順位はグループで定義され、一部のグループは特定のカーネル機能の専用となります。

表17.1 スレッド優先度テーブル

優先度	Threads	説明
1	優先度の低いカーネルスレッド	通常、この優先度は **SCHED_OTHER** よりは優先度の高いタスク用に予約されます。
2 - 49	利用可能	標準的なアプリケーションの優先度に使用される範囲。
50	デフォルトの IRQ 値	この優先度は、ハードウェアベースの割り込みのデフォルト値です。
51 - 98	優先度の高いスレッド	この範囲は、定期的に実行され、応答時間が短くなければならないスレッドに使用されます。低いレベルの割り込みに対答できなっくなるので、CPU にバインドされたスレッドには、この範囲を使用 **しないでください**。
99	ウォッチドッグおよび移行	最も高い優先順位で実行される必要があるシステムスレッド。

17.5. 関連情報

- systemd ユニットファイルでの作業

第18章 ネットワーク決定のヒント

TCP はレイテンシーに大きな影響を及ぼす可能性があります。TCP は、効率を高め、輻輳を制御し、信頼できる配信を保証するためにレイテンシーを追加します。チューニング時には、以下の点を考慮してください。

- 順番どおりの配信が必要か。
- パケットロスに対して保護する必要があるか。
  複数回パケットを送信すると遅延が発生する可能性があります。
- TCP を使用する必要があるか。
  ソケットで **TCP_NODELAY** を使用して Nagle バッファリングアルゴリズムを無効にすることを検討してください。Nagle アルゴリズムはすべてを一度に送信するために小さな送信パケットを収集するので、レイテンシーに悪影響を及ぼす可能性があります。

18.1. 遅延またはスループットの扱いに注意が必要なサービス向けに RHEL を最適化する

結合チューニングの目標は、特定のワークロードに必要な割り込みの数を最小限に抑えることです。高スループットの状況では、高いデータレートを維持しながら、割り込みをできるだけ少なくすることが目標となります。待ち時間が短い状況では、より多くの割り込みを使用してトラフィックを迅速に処理できます。

ネットワークカードの設定を調整して、1 つの割り込みに結合されるパケットの数を増減できます。その結果、トラフィックのスループットまたは遅延を向上させることができます。

手順

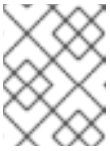
1. ボトルネックが発生しているネットワークインターフェイスを特定します。

```
# ethtool -S enp1s0
NIC statistics:
     rx_packets: 1234
     tx_packets: 5678
     rx_bytes: 12345678
     tx_bytes: 87654321
     rx_errors: 0
     tx_errors: 0
     rx_missed: 0
     tx_dropped: 0
     coalesced_pkts: 0
     coalesced_events: 0
     coalesced_aborts: 0
```

   名前に "drop"、"discard"、または "error" を含むパケットカウンターを特定します。これらの特定の統計は、ネットワークインターフェイスカード (NIC) の結合によって発生する可能性がある、NIC のパケットバッファーでの実際のパケットロスを測定します。

2. 前の手順で特定したパケットカウンターの値を監視します。
   これらをネットワークの予想値と比較して、特定のインターフェイスにボトルネックが発生しているかどうかを判断します。ネットワークのボトルネックの一般的な兆候には次のようなものがありますが、これらに限定されません。
   - ネットワークインターフェイス上での多数のエラー

- 高いパケットロス
- ネットワークインターフェイスの多用



注記

ネットワークのボトルネックを特定する際のその他の重要な要素としては、CPU 使用率、メモリー使用率、ディスク I/O などがあります。

3. 現在の結合設定を表示します。

```
# ethtool enp1s0
Settings for enp1s0:
        Supported ports: [ TP ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Full
        Supported pause frame use: No
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Full
        Advertised pause frame use: No
        Advertised auto-negotiation: Yes
        Speed: 1000Mb/s
        Duplex: Full
        Port: Twisted Pair
        PHYAD: 0
        Transceiver: internal
        Auto-negotiation: on
        MDI-X: Unknown
        Supports Wake-on: g
        Wake-on: g
        Current message level: 0x00000033 (51)
                               drv probe link
        Link detected: yes
```

この出力では、**Speed** フィールドおよび **Duplex** フィールドを監視します。これらのフィールドには、ネットワークインターフェイスの操作に関する情報と、それが期待値で実行されているかどうかが表示されます。

4. 現在の割り込み結合設定を確認します。

```
# ethtool -c enp1s0
Coalesce parameters for enp1s0:
        Adaptive RX: off
        Adaptive TX: off
        RX usecs: 100
        RX frames: 8
        RX usecs irq: 100
        RX frames irq: 8
        TX usecs: 100
        TX frames: 8
        TX usecs irq: 100
        TX frames irq: 8
```

- **usecs** 値は、受信機または送信機が割り込みを生成する前に待機するマイクロ秒数を指します。
- **frames** 値は、受信機または送信機が割り込みを生成する前に待機するフレーム数を指します。
- **irq** 値は、ネットワークインターフェイスがすでに割り込みを処理している場合に、割り込み調整を設定するために使用されます。

注記

すべてのネットワークインターフェイスカードが、出力例のすべての値のレポートと変更をサポートしているわけではありません。

- **Adaptive RX/TX** 値は、割り込み結合設定を動的に調整する適応割り込み結合メカニズムを表します。**Adaptive RX/TX** が有効な場合、NIC ドライバーはパケット条件に基づいて、結合値を自動計算します (アルゴリズムは NIC ドライバーごとに異なります)。

5. 必要に応じて結合設定を変更します。以下に例を示します。
   - **ethtool.coalesce-adaptive-rx** が無効になっている間に、RX パケットの割り込みを生成するまでの遅延を 100 マイクロ秒に設定するように **ethtool.coalesce-rx-usecs** を設定します。

     ```
     # nmcli connection modify enp1s0 ethtool.coalesce-rx-usecs 100
     ```

   - **ethtool.coalesce-rx-usecs** がデフォルト値に設定されている間、**ethtool.coalesce-adaptive-rx** を有効にします。

     ```
     # nmcli connection modify enp1s0 ethtool.coalesce-adaptive-rx on
     ```

     Red Hat では、Adaptive-RX 設定を次のように変更することを推奨します。
     - 低レイテンシー (50us 未満) が気になるユーザーは、**Adaptive-RX** を有効にしないでください。
     - スループットを懸念するユーザーは、おそらく問題なく **Adaptive-RX** を有効にすることができます。適応割り込み結合メカニズムを使用したくない場合は、**ethtool.coalesce-rx-usecs** に 100us や 250us などの大きな値を設定してみることができます。
     - 自分のニーズがわからないユーザーは、問題が発生するまでこの設定を変更しないでください。

6. 接続を再度有効にします。

   ```
   # nmcli connection up enp1s0
   ```

検証手順

- ネットワークパフォーマンスを監視し、ドロップされたパケットを確認します。

  ```
  # ethtool -S enp1s0
  NIC statistics:
  ```

```
     rx_packets: 1234
     tx_packets: 5678
     rx_bytes: 12345678
     tx_bytes: 87654321
     rx_errors: 0
     tx_errors: 0
     rx_missed: 0
     tx_dropped: 0
     coalesced_pkts: 12
     coalesced_events: 34
     coalesced_aborts: 56
...
```

rx_errors、**rx_dropped**、**tx_errors**、および **tx_dropped** フィールドの値は 0 またはそれに近い値 (ネットワークトラフィックとシステムリソースに応じて最大数百まで) である必要があります。これらのフィールドの値が高い場合は、ネットワークに問題があることを示します。カウンターには異なる名前を付けることができます。名前に "drop"、"discard"、または "error" を含むパケットカウンターを注意深く監視します。

rx_packets、**tx_packets**、**rx_bytes**、および **tx_bytes** の値は時間の経過とともに増加します。値が増加しない場合は、ネットワークに問題がある可能性があります。パケットカウンターは、NIC ドライバーに応じて異なる名前を持つことができます。

重要

ethtool コマンドの出力は、使用している NIC とドライバーによって異なる場合があります。

極めて低いレイテンシーを重視するユーザーは、監視目的でアプリケーションレベルのメトリクスまたはカーネルパケットタイムスタンプ API を使用できます。

関連情報

- Initial investigation for any performance issue
- What are the kernel parameters available for network tuning?
- How to make NIC ethtool settings persistent (apply automatically at boot)
- Timestamping

18.2. イーサネットネットワークのフロー制御

イーサネットリンクで、ネットワークインターフェイスとスイッチポートの間で継続的にデータ送信が行われると、バッファー容量がいっぱいになる可能性があります。バッファー容量がいっぱいになると、ネットワークの輻輳が発生します。この場合、送信側が受信側の処理能力よりも高いレートでデータを送信すると、パケットロスが発生する可能性があります。リンクの反対側のネットワークインターフェイス (スイッチポート) のデータ処理能力が低いためです。

フロー制御メカニズムは、送信側と受信側の送受信能力がそれぞれ異なるイーサネットリンクを介したデータ送信を管理します。パケットロスを回避するために、イーサネットフロー制御メカニズムはパケット送信を一時的に停止し、スイッチポート側の高い伝送レートを制御します。なお、ルーターがスイッチポートを越えてポーズフレームを転送することはありません。

受信 (RX) バッファーがいっぱいになると、受信側は送信側にポーズフレームを送信します。その後、

送信側は、1秒未満の短い期間、データ送信を停止しますが、この一時停止期間中は受信データのバッファリングを続けます。この期間は、受信側がインターフェイスバッファーを空にして、バッファーオーバーフローを防ぐのに十分な時間を提供します。

注記

イーサネットリンクのどちら側も、ポーズフレームを反対側に送信できます。ネットワークインターフェイスの受信バッファーがいっぱいになると、ネットワークインターフェイスはポーズフレームをスイッチポートに送信します。同様に、スイッチポートの受信バッファーがいっぱいになると、スイッチポートはネットワークインターフェイスにポーズフレームを送信します。

デフォルトでは、Red Hat Enterprise Linux のほとんどのネットワークドライバーではポーズフレームのサポートが有効になっています。ネットワークインターフェイスの現在の設定を表示するには、次のように入力します。

```
# ethtool --show-pause enp1s0
Pause parameters for enp1s0:
...
RX:     on
TX:     on
...
```

スイッチのベンダーに問い合わせて、スイッチがポーズフレームをサポートしているかどうかを確認してください。

関連情報

- **ethtool(8)** man ページ
- ネットワークリンクフロー制御とは何ですか? Red Hat Enterprise Linux ではどのように機能しますか?

18.3. 関連情報

- **ethtool(8)** man ページ
- **netstat (8)** の man ページ

第19章 TRACE-CMD を使用したレイテンシーのトレース

trace-cmd ユーティリティーは、**ftrace** ユーティリティーのフロントエンドです。**trace-cmd** を使用すると、**/sys/kernel/debug/tracing/** ディレクトリーに書き込む必要なく、**ftrace** アクションを有効にすることができます。**trace-cmd** は、そのインストールにオーバーヘッドを追加しません。

前提条件

- 管理者権限がある。

19.1. TRACE-CMD のインストール

trace-cmd ユーティリティーは、**ftrace** ユーティリティーのフロントエンドを提供します。

前提条件

- 管理者権限がある。

手順

- **trace-cmd** ユーティリティーをインストールします。

```
# yum install trace-cmd
```

19.2. TRACE-CMD の実行

trace-cmd ユーティリティーを使用して、すべての **ftrace** 機能にアクセスできます。

前提条件

- 管理者権限がある。

手順

- **trace-cmd command** を入力します。
  ここで、**command** は **ftrace** オプションに置き換えます。

注記

コマンドとオプションのリストは、**trace-cmd(1)** の man ページを参照してください。各コマンドのほとんどにも、独自の man ページ (**trace-cmd-command**) があります。

19.3. TRACE-CMD の例

以下のコマンド例で、**trace-cmd** ユーティリティーを使用してカーネル機能をトレースする方法を示します。

例

- myapp の実行中に、カーネル内で実行中の記録機能を有効にして開始します。

```
# trace-cmd record -p function myapp
```

これにより、myapp に無関係なタスクであっても、すべての CPU およびすべてのタスクの関数が記録されます。

- 結果を表示します。

```
# trace-cmd report
```

- myapp の実行中に、sched で始まる関数のみを記録します。

```
# trace-cmd record -p function -l 'sched*' myapp
```

- すべての IRQ イベントを有効にします。

```
# trace-cmd start -e irq
```

- **wakeup_rt** トレーサーを起動します。

```
# trace-cmd start -p wakeup_rt
```

- 関数トレースを無効にしながら、**preemptirqsoff** トレーサーを起動します。

```
# trace-cmd start -p preemptirqsoff -d
```

注記

RHEL 8 の **trace-cmd** は、**function-trace** ではなく **ftrace_enabled** を無効にします。**trace-cmd start -p** 機能を使用すると、**ftrace** を再度有効にできます。

- **trace-cmd** が変更を開始する前の状態にシステムを戻します。

```
# trace-cmd start -p nop
```

trace-cmd を使用した後に **debugfs** ファイルシステムを使用する場合は、システムを再起動したかどうかに関係なく、これが重要になります。

- 1 つのトレースポイントをトレースします。

```
# trace-cmd record -e sched_wakeup ls /bin
```

- トレースを停止します。

```
# trace-cmd record stop
```

19.4. 関連情報

- **trace-cmd(1)** の man ページ

第20章 TUNED-PROFILES-REAL-TIME を使用した CPU の分離

アプリケーションスレッドの実行時間を最大化するために、CPU を分離できます。そのため、無関係なタスクを可能な限り多く CPU から削除します。通常、CPU の分離には、以下の操作が必要です。

- ユーザー空間のスレッドをすべて削除する。
- バインドされていないカーネルスレッドを削除します。カーネル関連のバインドされたスレッドは特定の CPU にリンクされており、移動できません)。
- システム内の各 Interrupt Request (IRQ) 番号が **N** の **/proc/irq/N/smp_affinity** プロパティーを変更して割り込みを削除する。

package`tuned-profiles-realtime` パッケージの **isolated_cores=cpulist** 設定オプションを使用することで、CPU を分離する操作を自動化できます。

前提条件

- 管理者権限がある。

20.1. 分離する CPU の選択

分離する CPU を選択する際は、システムの CPU トポロジーを慎重に考慮する必要があります。ユースケースごとに異なる設定が必要です。

- スレッドがキャッシュを共有することによって相互に通信する必要があるマルチスレッドアプリケーションがある場合は、それらを同じ NUMA ノードまたは物理ソケットに保持する必要があります。
- 複数の無関係なリアルタイムアプリケーションを実行する場合は、CPU を NUMA ノードまたはソケットで分離することが適切な場合があります。

hwloc パッケージは、**lstopo-no-graphics** および **numactl** などの CPU に関する情報を取得する際に便利なユーティリティーを提供します。

前提条件

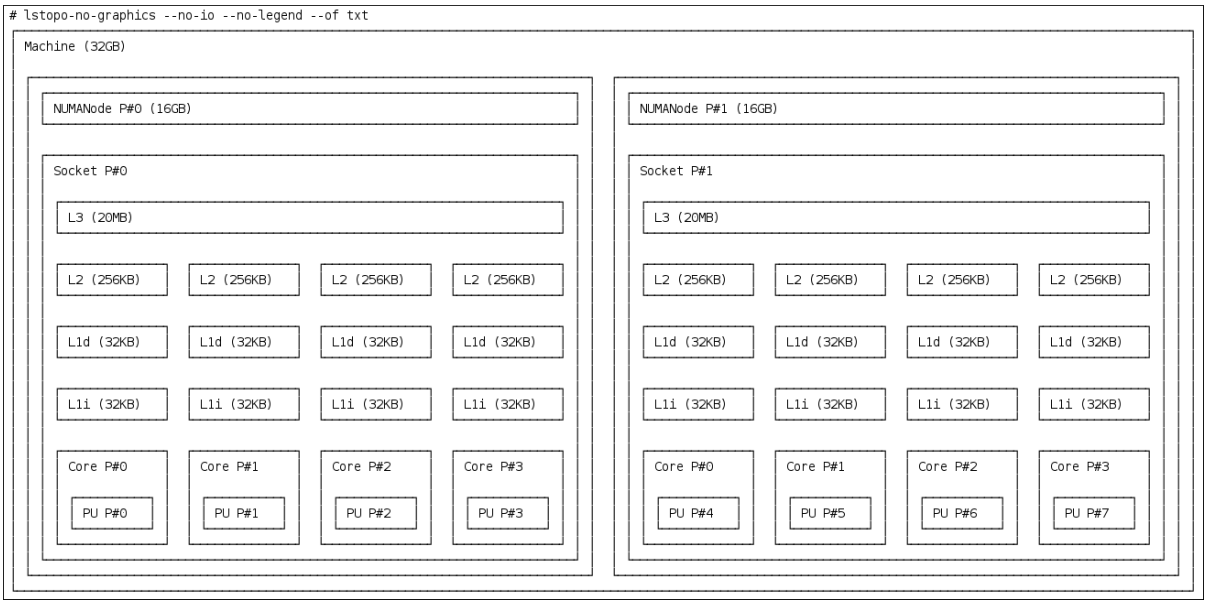
- **hwloc** パッケージがインストールされます。

手順

1. 物理パッケージで利用可能な CPU のレイアウトを表示します。

```
# lstopo-no-graphics --no-io --no-legend --of txt
```

図20.1 lstopo-no-graphics を使用した CPU のレイアウトの表示



このコマンドは、使用可能なコア数とソケット数、および NUMA ノードの論理距離を示すため、マルチスレッドアプリケーションに役立ちます。

また、**hwloc-gui** パッケージには、グラフィカル出力を生成する **lstopo** ユーティリティーが含まれます。

2. ノード間の距離など、CPU の詳細を表示します。

```
# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3
node 0 size: 16159 MB
node 0 free: 6323 MB
node 1 cpus: 4 5 6 7
node 1 size: 16384 MB
node 1 free: 10289 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

関連情報

- **hwloc(7)** の man ページ

20.2. TUNED の ISOLATED_CORES オプションを使用した CPU の分離

CPU を分離する最初のメカニズムは、カーネルブートコマンドラインでブートパラメーターの **isolcpus=cpulist** を指定することです。RHEL for Real Time での推奨の方法は、 **TuneD** デーモンとその **tuned-profiles-realtime** を使用することです。

注記

tuned-profiles-realtime バージョン 2.19 以降では、組み込み関数 **calc_isolated_cores** が初期 CPU セットアップを自動的に適用します。**/etc/tuned/realtime-variables.conf** 設定ファイルには、デフォルトの変数コンテンツが **isolated_cores=${f:calc_isolated_cores:2}** として含まれています。

デフォルトでは、**calc_isolated_cores** はソケットごとに 1 つのコアをハウスキーピング用に予約し、残りを分離します。デフォルト設定を変更する必要がある場合は、**/etc/tuned/realtime-variables.conf** 設定ファイルの **isolated_cores=${f:calc_isolated_cores:2}** 行をコメントアウトし、TuneD の **isolated_cores** オプションを使用して CPU を分離する手順に従います。

前提条件

- **TuneD** パッケージおよび **tuned-profiles-realtime** パッケージがインストールされている。
- システムの root 権限がある。

手順

1. root ユーザーとして、テキストエディターで **/etc/tuned/realtime-variables.conf** を開きます。
2. **isolated_cores=cpulist** を設定して、分離する CPU を指定します。CPU 番号および範囲を使用できます。

   例 :

   ```
   isolated_cores=0-3,5,7
   ```

   コア 0、1、2、3、5、および 7 を分離します。

   8 コアの 2 ソケットシステム (NUMA ノード 0 にはコア 0-3 があり、NUMA ノード 1 にはコア 4-8 がある) で、マルチスレッドアプリケーションに 2 つのコアを割り当てるには、以下を指定します。

   ```
   isolated_cores=4,5
   ```

   これにより、ユーザー空間のスレッドが CPU 4 および 5 に割り当てられなくなります。

   無関係なアプリケーション用に、異なる NUMA ノードから CPU を選択するには、以下のコマンドを実行します。

   ```
   isolated_cores=0,4
   ```

   これにより、ユーザー空間のスレッドが CPU 0 および 4 に割り当てられなくなります。

3. **tuned-adm** ユーティリティーを使用して、リアルタイムの **TuneD** プロファイルを有効にします。

   ```
   # tuned-adm profile realtime
   ```

4. マシンを再起動して変更を有効にします。

検証

- カーネルコマンドラインで **isolcpus** パラメーターを検索します。

```
$ cat /proc/cmdline | grep isolcpus
BOOT_IMAGE=/vmlinuz-4.18.0-305.rt7.72.el8.x86_64 root=/dev/mapper/rhel_foo-root ro
crashkernel=auto rd.lvm.lv=rhel_foo/root rd.lvm.lv=rhel_foo/swap console=ttyS0,115200n81
isolcpus=0,4
```

20.3. NOHZ パラメーターおよび NOHZ_FULL パラメーターを使用した CPU の分離

nohz パラメーターおよび **nohz_full** パラメーターは、指定された CPU の動作を変更します。このようなカーネルブートパラメーターを有効にするには、**realtime-virtual-host**、**realtime-virtual-guest**、または **cpu-partitioning** のいずれかのプロファイルをチューニングする必要があります。

nohz=on

特定の CPU セットでのタイマーアクティビティーを減らします。
nohz パラメーターは、主にアイドル状態の CPU でタイマー割り込みを減らすために使用されます。これにより、アイドル状態の CPU を低消費電力モードで実行でき、バッテリーの寿命を長持ちさせます。リアルタイムの応答時間に対して直接的なメリットはありませんが、**nohz** パラメーターはリアルタイムの応答時間に直接的な悪影響を与えることはありません。ただし、リアルタイムパフォーマンスにプラスの影響を与える **nohz_full** パラメーターを有効にするには、**nohz** パラメーターが必要です。

nohz_full=cpulist

nohz_full パラメーターで指定した CPU のリストでは、タイマーティックを処理する方法が異なります。CPU が **nohz_full** CPU として指定され、CPU に実行可能なタスクが 1 つしかないと、カーネルはその CPU へのタイマーティックの送信を停止します。その結果、アプリケーション実行の時間が長くなり、割り込みの処理やコンテキストの切り替えに対する時間が短くなります。

関連情報

- カーネルティックタイムの設定

第21章 SCHED_OTHER タスクの移行の制限

sched_nr_migrate を使用すると、**SCHED_OTHER** が他の CPU に移行するタスクを制限できます。

前提条件

- 管理者権限がある。

21.1. タスクの移行

SCHED_OTHER タスクが他の多数のタスクを生成する場合、それらはすべて同じ CPU 上で実行されます。**migration** タスクまたは **softirq** は、アイドル状態の CPU で実行できるように、これらのタスクのバランスをとろうとします。

sched_nr_migrate は、一度に移動するタスクの数を指定するように調整できます。リアルタイムタスクの移行方法は異なるため、この影響を直接受けることはありません。ただし、**softirq** がタスクを移動すると、実行キューのスピンロックが有効になり、割り込みが無効になります。

移行が必要なタスクが多数存在すると、そのタスクは割り込みが無効になっている間に発生するため、タイマーイベントやウェイクアップは同時には行われません。これにより、**sched_nr_migrate** を大きい値に設定した場合に、リアルタイムタスクで深刻なレイテンシーが発生する可能性があります。

21.2. SCHED_NR_MIGRATE 変数を使用した SCHED_OTHER タスクの移行の制限

sched_nr_migrate 変数の値を大きくすると、リアルタイムレイテンシーを犠牲にして、多くのタスクを起動する **SCHED_OTHER** スレッドから高パフォーマンスが得られます。

SCHED_OTHER タスクのパフォーマンスを犠牲にしてリアルタイムのタスクレイテンシーを低くするには、値を小さくする必要があります。デフォルト値は **8** です。

手順

- **sched_nr_migrate** 変数の値を調整するには、echo コマンドで値を直接 **/proc/sys/kernel/sched_nr_migrate** に出力します。

```
# echo 2 > /proc/sys/kernel/sched_nr_migrate
```

検証

- **/proc/sys/kernel/sched_nr_migrate** のコンテンツを表示します。

```
# cat > /proc/sys/kernel/sched_nr_migrate
2
```

第22章 TCP パフォーマンスのスパイクの低減

TCP タイムスタンプを生成すると、TCP パフォーマンスのスパイクが発生する可能性があります。**sysctl** は、TCP 関連のエントリー値を制御し、**/proc/sys/net/ipv4/tcp_timestamps** で検出された timestamps カーネルパラメーターを設定します。

前提条件

- 管理者権限がある。

22.1. TCP タイムスタンプの無効化

TCP タイムスタンプを無効にすると、TCP パフォーマンスの急激な変化を低減できます。

手順

- TCP タイムスタンプをオフにします。

```
# sysctl -w net.ipv4.tcp_timestamps=0
net.ipv4.tcp_timestamps = 0
```

この出力は、**net.ip4.tcp_timestamps** オプションの値が **0** であることを示しています。つまり、TCP タイムスタンプが無効です。

22.2. TCP タイムスタンプの有効化

タイムスタンプを生成すると、TCP パフォーマンスのスパイクが発生する可能性があります。TCP タイムスタンプを無効にすることで、TCP パフォーマンスのスパイクを低減できます。TCP タイムスタンプを生成しても TCP パフォーマンスのスパイクが発生しない場合は、タイムスタンプを有効にできます。

手順

- TCP タイムスタンプを有効にします。

```
# sysctl -w net.ipv4.tcp_timestamps=1
net.ipv4.tcp_timestamps = 1
```

この出力は、**net.ip4.tcp_timestamps** の値が **1** であることを示しています。つまり、TCP タイムスタンプが有効です。

22.3. TCP のタイムスタンプステータスの表示

TCP タイムスタンプの生成ステータスを表示できます。

手順

- TCP タイムスタンプの生成ステータスを表示します。

```
# sysctl net.ipv4.tcp_timestamps
net.ipv4.tcp_timestamps = 0
```

1 は、タイムスタンプが生成されていることを示します。**0** は、タイムスタンプが生成されていないことを示しています。

第23章 RCU コールバックを使用した CPU パフォーマンスの改善

Read-Copy-Update (**RCU**) は、カーネル内でスレッドを相互に排他的にするロックレスのメカニズムです。RCU 操作を実施することで、コールバックが CPU のキューに置かれ、今後メモリーを安全に削除できる状況になると実行されます。

RCU コールバックを使用して CPU のパフォーマンスを改善するには、以下の操作を行います。

- CPU を、CPU コールバックの実行の候補から除外する。
- すべての RCU コールバックを処理する CPU を割り当てる。この CPU は、ハウスキーピング CPU と呼ばれます。
- CPU を、RCU オフロードスレッド起動の処理から除外する。

この組み合わせにより、ユーザーのワークロードに特化した CPU への干渉が軽減されます。

前提条件

- 管理者権限がある。
- **tuna** がインストールされている。

23.1. RCU コールバックのオフロード

rcu_nocbs および **rcu_nocb_poll** カーネルパラメーターを使用して、**RCU** コールバックをオフロードできます。

手順

- RCU コールバックを実行する候補から 1 つ以上の CPU を除外するには、**rcu_nocbs** カーネルパラメーターで CPU のリストを指定します。以下に例を示します。

```
rcu_nocbs=1,4-6
```

  または、以下を実行します。

```
rcu_nocbs=3
```

  2 つ目の例では、CPU 3 が no-callback CPU であることをカーネルに指示します。つまり、RCU コールバックは、CPU 3 に固定された **rcuc/$CPU** スレッドではなく、**rcuo/$CPU** スレッドで実行されます。このスレッドをハウスキーピング CPU に移動すると、CPU 3 に RCU コールバックジョブが割り当てられなくなります。

23.2. RCU コールバックの移動

ハウスキーピング CPU を割り当てて、すべての RCU コールバックスレッドを処理できます。これには、**tuna** コマンドを使用して、すべての RCU コールバックをハウスキーピング CPU に移動します。

手順

- RCU コールバックスレッドをハウスキーピング CPU に移動します。

```
# tuna --threads=rcu --cpus=x --move
```

- ここで、**x** は、ハウスキーピング CPU の CPU 番号に置き換えます。

このアクションにより、CPU **X** 以外のすべての CPU が RCU コールバックスレッドを処理しなくなります。

23.3. CPU の RCU オフロードスレッド起動からの除外

RCU オフロードスレッドは別の CPU の RCU コールバックを実行できますが、各 CPU は対応する RCU オフロードスレッド起動を処理する必要があります。この処理を CPU で行わないようにすることができます。

手順

- **rcu_nocb_poll** カーネルパラメーターを設定します。
  このコマンドにより、タイマーが RCU オフロードスレッドを定期的に起動して、実行するコールバックがあるかどうかを確認します。

23.4. 関連情報

- Avoiding RCU Stalls in the real-time kernel

第24章 FTRACE を使用したレイテンシーのトレース

ftrace ユーティリティーは、RHEL for Real Time Kernel で提供される診断機能の 1 つです。 **ftrace** は、開発者がユーザー空間外で発生するレイテンシーおよびパフォーマンスの問題を分析およびデバッグするのに使用できます。**ftrace** ユーティリティーには、さまざまな方法でユーティリティーを使用できるさまざまなオプションがあります。これは、コンテキストスイッチの追跡、優先順位の高いタスクでのウェイクアップにかかる時間の測定、割り込みが無効になっている期間の測定、特定の期間中に実行されたカーネル関数のリストの表示に使用できます。

ftrace トレーサーなどの一部のトレーサーは、大量のデータ量を生成し、トレースログ分析を時間の消費タスクに切り替えます。ただし、トレーサーに対し、アプリケーションが重要なコードパスに到達した場合にのみ開始および終了するように指示することが可能です。

前提条件

- 管理者権限がある。

24.1. FTRACE ユーティリティーを使用したレイテンシーの追跡

ftrace ユーティリティーを使用して、レイテンシーを追跡できます。

手順

1. システムで利用可能なトレーサーを表示します。

```
# cat /sys/kernel/debug/tracing/available_tracers
function_graph wakeup_rt wakeup preemptirqsoff preemptoff irqsoff function nop
```

   ftrace のユーザーインターフェイスは、**debugfs** 内の一連のファイルです。

   ftrace ファイルは、**/sys/kernel/debug/tracing/** ディレクトリーにあります。

2. **/sys/kernel/debug/tracing/** ディレクトリーに移動します。

```
# cd /sys/kernel/debug/tracing
```

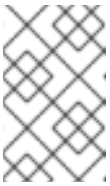
   トレースを有効にするとシステムのパフォーマンスに影響を及ぼす可能性があるため、このディレクトリーのファイルを変更することができるのは root ユーザーのみです。

3. トレースセッションを開始するには、以下を行います。

   a. **/sys/kernel/debug/tracing/available_tracers** で利用可能なトレーサーのリストから、使用するトレーサーを選択します。

   b. セレクターの名前を **/sys/kernel/debug/tracing/current_tracer** に挿入します。

```
# echo preemptoff > /sys/kernel/debug/tracing/current_tracer
```



注記

echo コマンドと >1つを合わせて使用する場合は、ファイル内の既存の値が上書きされます。ファイルに値を追記する場合は、代わりに '>>' を使用します。

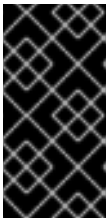
4. **function-trace** オプションは、**wakeup_rt**、**preemptirqsoff** などを使用してレイテンシーをトレースすると、関数のトレースが自動的に有効になり、オーバーヘッドが誇張される可能性があるため便利です。
function および **function_graph** のトレースが有効になっているかどうかを確認します。

```
# cat /sys/kernel/debug/tracing/options/function-trace
1
```

- 値を **1** に設定すると、**function** と **function_graph** のトレースが有効になります。
- 値が **0** の場合は、**function** および **function_graph** のトレースが無効になっていることを示します。

5. デフォルトでは、**function** および **function_graph** トレースは有効になっています。**function** および **function_graph** のトレースのオン/オフを切り替えるには、**/sys/kernel/debug/tracing/options/function-trace** に適切な値を echo で追加します。

```
# echo 0 > /sys/kernel/debug/tracing/options/function-trace
# echo 1 > /sys/kernel/debug/tracing/options/function-trace
```



重要

echo コマンドを使用する場合は、値と **>** 文字の間に空白文字を配置するようにしてください。**0>**、**1>**、および **2>** (空白文字なし) を使用するシェルプロンプトでは、標準入力、標準出力、および標準エラーを参照します。誤ってそれらを使用すると、トレースが予期せぬ出力になる可能性があります。

6. **/debugfs/tracing/** ディレクトリー内のさまざまなファイルの値を変更して、トレーサーの詳細とパラメーターを調整します。
以下に例を示します。

irqsoff、**preemptoff**、**preempirqsoff**、および **wakeup** トレーサーは、レイテンシーを継続的に監視します。**tracing_max_latency** に記録されたレイテンシーよりも大きいレイテンシーを記録すると、そのレイテンシーのトレースが記録され、**tracing_max_latency** が新しい最大時間に更新されます。これにより、**tracing_max_latency** は、最後にリセットされてから記録された最大のレイテンシーを常に表示します。

- 最大レイテンシーをリセットするには、**0** を **tracing_max_latency** ファイルに echo で追加します。

```
# echo 0 > /sys/kernel/debug/tracing/tracing_max_latency
```

- 設定された量よりも大きいレイテンシーのみを表示するには、マイクロ秒単位で量を echo で出力します。

```
# echo 200 > /sys/kernel/debug/tracing/tracing_max_latency
```

トレースのしきい値を設定すると、最大レイテンシー設定が上書きされます。しきい値より大きいレイテンシーが記録されると、最大レイテンシーに関係なく記録されます。トレースファイルを確認すると、最後に記録されたレイテンシーのみが表示されます。

- しきい値を設定するには、それを超えるとレイテンシーを記録する必要があるマイクロ秒数を echo で出力します。

```
# echo 200 > /sys/kernel/debug/tracing/tracing_thresh
```

7. トレースログを表示します。

```
# cat /sys/kernel/debug/tracing/trace
```

8. トレースログを保存するには、別のファイルにコピーします。

```
# cat /sys/kernel/debug/tracing/trace > /tmp/lat_trace_log
```

9. トレースされている関数を表示します。

```
# cat /sys/kernel/debug/tracing/set_ftrace_filter
```

10. **/sys/kernel/debug/tracing/set_ftrace_filter** で設定を編集して、トレースしている関数をフィルターにかけます。ファイルにフィルターが指定されていない場合、すべての関数がトレースされます。

11. フィルター設定を変更するには、トレースする関数名を echo で追記します。このフィルターでは、検索用語の先頭または末尾に * ワイルドカードを使用できます。
例は ftrace の例 を参照してください。

24.2. FTRACE ファイル

/sys/kernel/debug/tracing/ ディレクトリーの主なファイルを以下に示します。

ftrace ファイル

trace

ftrace トレースの出力を表示するファイル。これは、このファイルが読み込まれるとトレースが停止し、読み込まれたイベントを使用しないため、実際にはトレースのスナップショットです。つまり、ユーザーがトレースを無効にしてこのファイルを読み取ると、読み取り時に毎回同じ内容を報告します。

trace_pipe

トレースをライブで読み込む際に、**ftrace** トレースの出力を表示するファイル。これは、プロデューサー/コンシューマーのトレースです。つまり、読み取りごとに、読み取られたイベントが消費されます。これは、読み取り時にトレースを停止せずに、アクティブなトレースの読み取りで使用できます。

available_tracers

カーネルにコンパイルされた ftrace トレーサーのリスト。

current_tracer

ftrace トレーサーを有効または無効にします。

events

トレースするイベントが含まれ、イベントを有効または無効にするのに使用できるディレクトリーと、イベントのフィルターの設定を行うことができます。

tracing_on

ftrace バッファーへの録画を無効および有効にします。**tracing_on** ファイル経由でトレースを無効にしても、カーネル内で行われている実際のトレースは無効になりません。バッファーへの書き込みのみを無効にします。トレースを実行する作業は継続されますが、データはどこにも移動しません。

24.3. FTRACE トレーサー

カーネルの設定方法によっては、指定のカーネルですべてのトレーサーが利用できるとは限りません。RHEL for Real Time カーネルの場合、トレースカーネルおよびデバッグカーネルには、実稼働カーネルとは異なるトレーサーがあります。これは、トレーサーの一部にトレーサーがカーネルに設定され、アクティブではない場合に大きなオーバーヘッドが発生するためです。このトレーサーは、**trace** および **debug** カーネルに対してのみ有効になります。

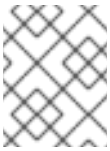
トレーサー

function

最も広く適用されるトレーサーの1つ。カーネル内の関数呼び出しを追跡します。トレースされる関数の数によっては、目立ったオーバーヘッドが発生する場合があります。アクティブでない場合は、ほとんどオーバーヘッドが発生しません。

function_graph

function_graph トレーサーは、より視覚に訴える形式で結果を表示するように設計されています。このトレーサーは、関数の終了を追跡し、カーネル内の関数呼び出しのフローを表示します。



注記

このトレーサーは、有効化されると **function** トレーサーよりもオーバーヘッドが高くなりますが、無効化されると同じオーバーヘッドが低くなります。

wakeup

すべての CPU でアクティビティーが発生することを報告する完全な CPU トレーサー。リアルタイムタスクであるかに関わらず、システム内で最も優先度の高いタスクを起動するのにかかる時間を記録します。非リアルタイムタスクを起動するのにかかる最大時間の記録では、リアルタイムタスクを起動するのにかかる時間が非表示になります。

wakeup_rt

すべての CPU でアクティビティーが発生することを報告する完全な CPU トレーサー。現在の最も高い優先度タスクから、ウェイクアップ時間まで経過時間を記録します。このトレーサーは、リアルタイムタスクの時間のみを記録します。

preemptirqsoff

プリエンプションまたは割り込みを無効にするエリアを追跡し、プリエンプションまたは割り込みが無効となった最大時間を記録します。

preemptoff

preemptirqsoff トレーサーと似ていますが、プリエンプションが無効化された最大間隔のみをトレースします。

irqsoff

preemptirqsoff トレーサーと似ていますが、割り込みが無効化された最大間隔のみをトレースします。

nop

デフォルトのトレーサー。トレース機能自体は提供しませんが、イベントがトレーサーにインターリーブする可能性があるため、**nop** トレーサーは、イベントのトレースに特に関心がある場合に使用されます。

24.4. FTRACE の例

以下では、トレースする関数のフィルター処理を変更する例を多数説明します。単語の先頭と末尾の両方に * ワイルドカードを使用できます。たとえば、***irq*** は、名前に **irq** を含むすべての関数を選択します。ただし、ワイルドカードは単語内で使用できません。

検索用語とワイルドカード文字を二重引用符で囲むと、シェルが検索を現在の作業ディレクトリーに拡張しないようにします。

フィルターの例

- **schedule** 関数のみをトレースします。

```
# echo schedule > /sys/kernel/debug/tracing/set_ftrace_filter
```

- **lock** で終わるすべての関数をトレースします。

```
# echo "*lock" > /sys/kernel/debug/tracing/set_ftrace_filter
```

- **spin_** で始まるすべての関数をトレースします。

```
# echo "spin_*" > /sys/kernel/debug/tracing/set_ftrace_filter
```

- 名前に **cpu** のあるすべての関数をトレースします。

```
# echo "cpu" > /sys/kernel/debug/tracing/set_ftrace_filter
```

第25章 アプリケーションのタイムスタンプ

アプリケーションがタイムスタンプを頻繁に実行する場合には、CPU によるクロック読み取りが原因でパフォーマンスに影響があります。クロックの読み取りに使用するコストや時間がかさむと、アプリケーションのパフォーマンスに悪影響を及ぼす可能性があります。

読み出しメカニズムが備わっているハードウェアクロックを選択すると、デフォルトのクロックよりも速くなり、クロック読み取りのコストが軽減されます。

RHEL for Real Time では、POSIX クロックを **clock_gettime()** 関数とともに使用して、CPU のコストを可能な限り低く抑えて、クロックの読み取り値を生成し、パフォーマンスをさらに向上させることができます。

読み取りコストの高いハードウェアクロックを使用するシステムで、このような利点がより明確になります。

25.1. POSIX クロック

POSIX は、タイムソースを実装して表すための標準です。システム内のその他のアプリケーションに影響を及ぼさずに、POSIX クロックをアプリケーションに割り当てることができます。これは、カーネルによって選択され、システム全体に実装されるハードウェアクロックとは対照的です。

指定の POSIX クロックを読み取るために使用される関数は **<time.h>** で定義される **clock_gettime()** です。**clock_gettime()** に相当するカーネルはシステムコールです。ユーザープロセスが **clock_gettime()** を呼び出すと、以下が行われます。

1. 対応する C ライブラリー (**glibc**) は、**sys_clock_gettime()** システムコールを呼び出します。
2. **sys_clock_gettime()** は、要求されたオペレーションを実行します。
3. **sys_clock_gettime()** は、結果をユーザープログラムプログラムに戻します。

ただし、このコンテキストはユーザーアプリケーションからカーネルへの切り替えには CPU コストがかかります。このコストは非常に低くなりますが、操作が数千回繰り返し行われると、累積されたコストはアプリケーション全体のパフォーマンスに影響を及ぼす可能性があります。カーネルへのコンテキストの切り替えを回避し、クロックの読み出しを速くするために、VDSO (Virtual Dynamic Shared Object) ライブラリー機能の形式で **CLOCK_MONOTONIC_COARSE** クロックおよび **CLOCK_REALTIME_COARSE** POSIX クロックのサポートが追加されました。

_COARSE クロックバリアントのいずれかを使用して **clock_gettime()** が実行する時間測定は、カーネルの介入を必要とせず、ユーザー空間全体で実行されます。これにより、パフォーマンスが大幅に向上します。**_COARSE** クロックの時間読み取りの分解能はミリ秒 (ms) です。つまり、1ms 未満の時間間隔は記録されません。POSIX クロックの **_COARSE** バリアントは、ミリ秒のクロック分解能に対応できるアプリケーションに適しています。

注記

_COARSE 接頭辞の有無にかかわらず、POSIX クロックの読み出しコストと分解能を比較するには、RHEL for Real Time Reference ガイド を参照してください。

25.2. CLOCK_GETTIME での _COARSE クロックバリアントの使用

コード出力例は、**CLOCK_MONOTONIC_COARSE** POSIX クロックを使用した **clock_gettime** 関数の使用を示しています。

```
#include <time.h>

main()
{
 int rc;
 long i;
 struct timespec ts;

 for(i=0; i<10000000; i++) {
  rc = clock_gettime(CLOCK_MONOTONIC_COARSE, &ts);
 }
}
```

上記の例を改善するには、より多くの文字列を使用して **clock_gettime()** の戻りコードを確認したり、**rc** 変数の値を確認したり、**ts** 構造のコンテンツが信頼できるようにしたりします。

注記

clock_gettime() の man ページでは、信頼できるアプリケーションを作成する方法が説明されています。

重要

clock_gettime() 関数を使用するプログラムは、**'-lrt'** を **gcc** コマンドラインに追加して、**-lrt** ライブラリーにリンクする必要があります。

$ gcc clock_timing.c -o clock_timing -lrt

25.3. 関連情報

- **clock_gettime()** man ページ

第26章 TCP_NODELAY を使用したネットワーク遅延の改善

デフォルトでは、**TCP** は Nagle のアルゴリズムを使用して、小さな送信パケットを集めて一度に送信します。これにより、レイテンシーの発生率が高くなる可能性があります。

前提条件

- 管理者権限がある。

26.1. TCP_NODELAY の使用による影響

送信されるすべてのパケットでレイテンシーを低く保つ必要のあるアプリケーションは、**TCP_NODELAY** オプションを有効化しているソケットで実行する必要があります。イベントが発生するとすぐに、カーネルにバッファー書き込みを送信します。

注記

TCP_NODELAY を効果的に使用するには、アプリケーションが小規模な論理的に関連するバッファー書き込みを行わないようにする必要があります。これを行わないと、このような小さい書き込みにより、**TCP** はこれらの複数のバッファーを個別のパケットとして送信するため、全体的なパフォーマンスが低下します。

論理に関連し、1つのパケットとして送信する必要があるバッファーがアプリケーションに複数ある場合は、パフォーマンスの低下を回避するために、以下の回避策のいずれかを適用します。

- メモリー内に連続したパケットを構築し、**TCP_NODELAY** で設定したソケット上で論理パケットを **TCP** に送信する。
- I/O ベクターを作成し、**TCP_NODELAY** で設定したソケット上で **writev** コマンドを使用してカーネルに渡す。
- **TCP_CORK** オプションを使用する。**TCP_CORK** は、アプリケーションがコルクを削除するのを待ってからパケットを送信するように **TCP** に指示します。このコマンドにより、受信するバッファーが既存のバッファーに追加されます。これにより、アプリケーションはカーネル領域にパケットを構築できます。これは、レイヤーの抽象化を提供する異なるライブラリーを使用する場合は必要です。

アプリケーションのさまざまなコンポーネントにより、論理パケットがカーネルに構築されている場合は、ソケットのコルクを解除する必要があります。これにより、**TCP** が、累積した論理パケットをすぐに送信できるようになります。

26.2. TCP_NODELAY の有効化

イベントが発生すると、**TCP_NODELAY** オプションは遅滞なくバッファー書き込みをカーネルに送信します。**setsockopt()** 関数を使用して **TCP_NODELAY** を有効にします。

手順

1. 次の行を **TCP** アプリケーションの **.c** ファイルに追加します。

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_NODELAY, &one, sizeof(one));
```

2. ファイルを保存して、エディターを終了します。

3. パフォーマンスの低下を防ぐために、以下の回避策のいずれかを適用します。

   - メモリー内に連続したパケットを構築し、**TCP_NODELAY** で設定したソケット上で論理パケットを **TCP** に送信する。
   - I/O ベクターを作成し、**TCP_NODELAY** が設定されたソケット上で **writev** を使用してカーネルに渡す。

26.3. TCP_CORK の有効化

TCP_CORK オプションは、ソケットが "uncorked" になるまで **TCP** がパケットを送信しないようにします。

手順

1. 次の行を **TCP** アプリケーションの **.c** ファイルに追加します。

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_CORK, &one, sizeof(one));
```

2. ファイルを保存して、エディターを終了します。
3. アプリケーション内のさまざまなコンポーネントにより論理パケットがカーネルで構築されたら、**TCP_CORK** を無効にします。

```
int zero = 0;
setsockopt(descriptor, SOL_TCP, TCP_CORK, &zero, sizeof(zero));
```

   TCP は、アプリケーションからのパケットを待たずに、直ちに累積した論理パケットを送信します。

26.4. 関連情報

- **tcp(7)** man ページ
- **setsockopt(3p)** の man ページ
- **setsockopt(2)** の man ページ

第27章 ミューテックスの使用によるリソースの過剰使用の回避

相互排他 ミューテックス) アルゴリズムは、共通リソースの過剰使用を防ぐために使用されます。

27.1. ミューテックスオプション

相互除外 (ミューテックス) アルゴリズムは、プロセスが共通のリソースを同時に使用するのを防ぐために使用されます。高速ユーザー空間ミューテックス (futex) は、ミューテックスが別のスレッドによって保持されていない場合に、カーネル領域にコンテキストスイッチを要求せずにユーザー空間スレッドがミューテックスを要求することを可能にするツールです。

標準属性で **pthread_mutex_t** オブジェクトを初期化すると、プライベートで再帰的ではなく、堅牢ではない優先度継承対応ではないミューテックスが作成されます。このオブジェクトには、**pthreads** API および RHEL for Real Time カーネルで提供される利点はありません。

pthreads API および RHEL for Real Time カーネルの利点を活用するには、 **pthread_mutexattr_t** オブジェクトを作成します。このオブジェクトは、futex に定義した属性を保存します。

注記

futex および **mutex** という用語は、POSIX スレッド (**pthread**) のミューテックス構造を説明するために使用されます。

27.2. ミューテックス属性オブジェクトの作成

mutex に追加の機能を定義するには、**pthread_mutexattr_t** オブジェクトを作成します。このオブジェクトは、futex に定義した属性を保存します。これは基本的な安全手順で、常に実行する必要があります。

手順

- 以下のいずれかを使用して、ミューテックス属性オブジェクトを作成します。
  - **pthread_mutex_t(my_mutex);**
  - **pthread_mutexattr_t(&my_mutex_attr);**
  - **pthread_mutexattr_init(&my_mutex_attr);**

拡張ミューテックス属性の詳細は、拡張ミューテックス属性 を参照してください。

27.3. 標準属性のミューテックスの作成

標準属性で **pthread_mutex_t** オブジェクトを初期化すると、プライベートで再帰的ではなく、堅牢ではない優先度継承対応ではないミューテックスが作成されます。

手順

- 以下のいずれかを使用して、**pthreads** にミューテックスオブジェクトを作成します。
  - **pthread_mutex_t(my_mutex);**
  - **pthread_mutex_init(&my_mutex, &my_mutex_attr);**
    &my_mutex_attr; は、ミューテックス属性オブジェクトです。

27.4. 拡張ミューテックス属性

以下の拡張ミューテックス属性は、ミューテックス属性オブジェクトに格納できます。

ミューテックス属性

共有およびプライベートのミューテックス

共有ミューテックスはプロセス間で使用できますが、大きなオーバーヘッドが発生します。
pthread_mutexattr_setpshared(&my_mutex_attr, PTHREAD_PROCESS_SHARED);

リアルタイム優先度の継承

優先度の継承を使用して、優先度が反転する問題を回避できます。
pthread_mutexattr_setprotocol(&my_mutex_attr, PTHREAD_PRIO_INHERIT);

強固なミューテックス

pthread が停止すると、pthread の下の強固なミューテックスが解放されます。ただし、これによりオーバーヘッドコストが高くなります。この文字列の **_NP** は、このオプションが非 POSIX であるか、移植性がないことを示します。
pthread_mutexattr_setrobust_np(&my_mutex_attr, PTHREAD_MUTEX_ROBUST_NP);

ミューテックスの初期化

共有ミューテックスはプロセス間で使用できますが、大きなオーバーヘッドが発生します。
pthread_mutex_init(&my_mutex_attr, &my_mutex);

27.5. ミューテックス属性オブジェクトの削除

ミューテックス属性オブジェクトを使用してミューテックスを作成した後に、属性オブジェクトを保持して同じタイプのミューテックスをさらに初期化することや、削除することができます。ミューテックスはいずれの場合も影響を受けません。

手順

- **_destroy** コマンドを使用して、属性オブジェクトを削除します。
  pthread_mutexattr_destroy(&my_mutex_attr);

  ミューテックスは通常の **pthread_mutex** として動作するようになり、通常どおりにロック、ロック解除、破棄できます。

27.6. 関連情報

- **futex(7)** man ページ
- **pthread_mutex_destroy(P)** man ページ
- **pthread_mutexattr_setprotocol(3p)** の man ページ
- **pthread_mutexattr_setprioceiling(3p)** の man ページ

第28章 アプリケーションのパフォーマンスの分析

perf はパフォーマンス分析ツールです。これは、簡単なコマンドラインインターフェイスを提供し、Linux のパフォーマンス測定における CPU ハードウェアの相違点を抽出します。**Perf** は、カーネルがエクスポートした **perf_events** インターフェイスに基づいています。

perf の利点の 1 つは、カーネルとアーキテクチャーの両方に依存しないことです。分析データは、特定のシステム設定なしに確認できます。

前提条件

- **perf** パッケージがシステムにインストールされている。
- 管理者権限がある。

28.1. システム全体の統計の収集

perf record コマンドは、システム全体の統計を収集するために使用されます。すべてのプロセッサーで使用できます。

手順

- システム全体のパフォーマンス統計を収集します。

```
# perf record -a
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.725 MB perf.data (~31655 samples) ]
```

この例では、オプション **-a** によりすべての CPU が表示され、数秒後にプロセスが終了しています。結果には、収集したデータは 0.725 MB で、新しく作成した **perf.data** ファイルに保存されたことが示されています。

検証

- 結果のファイルが作成されたことを確認します。

```
# ls
perf.data
```

28.2. パフォーマンス分析結果のアーカイブ

perf archive を使用すると、他のシステムでの **perf** の結果を分析できます。以下の場合は必要ありません。

- バイナリーやライブラリーなどの動的共有オブジェクト (DSO) は、~/**.debug**/ キャッシュなどの分析システムにすでに存在している。
- 両方のシステムでバイナリーのセットが同じである。

手順

1. **perf** コマンドによる結果のアーカイブを作成します。

```
# perf archive
```

2. アーカイブから tarball を作成します。

```
# tar cvf perf.data.tar.bz2 -C ~/.debug
```

28.3. パフォーマンス解析結果の分析

perf record 機能によるデータを、**perf report** コマンドを使用して直接調査できるようになりました。

手順

- **perf.data** ファイルまたはアーカイブした tarball から直接結果を分析します。

```
# perf report
```

レポートの出力は、アプリケーションの最大 CPU 使用率順に並べ替えられます。これは、サンプルがカーネルまたはプロセスのユーザー空間で発生したかどうかを示します。

レポートには、サンプルを取得したモジュールの情報が表示されます。

- カーネルモジュールで実行されていないカーネルサンプルには、**[kernel.kallsyms]** という表記が使用されます。
- カーネルモジュールで発生したカーネルサンプルには、**[module]**、**[ext4]** のマークが付けられます。
- ユーザー空間のプロセスでは、プロセスにリンクされた共有ライブラリーが結果に表示される可能性があります。
  レポートは、プロセスがカーネルまたはユーザースペースでも発生するかどうかを示します。
- 結果の **[.]** は、ユーザー空間を示しています。
- 結果の **[k]** は、カーネル領域を示しています。

経験豊富な **perf** 開発者に適したデータなど、詳細を確認することができます。

28.4. 定義済みイベントのリスト表示

ハードウェアトレースポイントアクティビティーを取得するために利用可能なオプションは複数あります。

手順

- 定義済みのハードウェアイベントおよびソフトウェアイベントのリストを表示します。

```
# perf list
List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                               [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  instructions                                       [Hardware event]
```

```
cache-references                          [Hardware event]
cache-misses                             [Hardware event]
branch-instructions OR branches                  [Hardware event]
branch-misses                            [Hardware event]
bus-cycles                               [Hardware event]

cpu-clock                                [Software event]
task-clock                               [Software event]
page-faults OR faults                          [Software event]
minor-faults                             [Software event]
major-faults                             [Software event]
context-switches OR cs                           [Software event]
cpu-migrations OR migrations                     [Software event]
alignment-faults                         [Software event]
emulation-faults                         [Software event]
...[output truncated]...
```

28.5. 指定したイベント統計の取得

perf stat を使用すると、特定のイベントを表示できます。

手順

1. **perf stat** 機能を使用して、コンテキストスイッチの数を表示します。

```
# perf stat -e context-switches -a sleep 5
^Performance counter stats for 'sleep 5':

            15,619 context-switches

       5.002060064 seconds time elapsed
```

結果には、5 秒間に 15619 のコンテキストスイッチが発生したことが示されています。

2. スクリプトを実行してファイルシステムのアクティビティーを表示します。以下はスクリプトの例になります。

```
# for i in {1..100}; do touch /tmp/$i; sleep 1; done
```

3. 別の端末で、**perf stat** コマンドを実行します。

```
# perf stat -e ext4:ext4_request_inode -a sleep 5
 Performance counter stats for 'sleep 5':

                5 ext4:ext4_request_inode

       5.002253620 seconds time elapsed
```

結果には、スクリプトが 5 秒間に 5 つのファイルの作成を要求したことが示され、**inode** 要求が 5 つあることが分かります。

28.6. 関連情報

- **perf help COMMAND**

- **perf**(1) の man ページ

第29章 STRESS-NG を使用したリアルタイムのシステムのストレステスト

stress-ng ツールは、望ましくない条件下で良好なレベルの効率を維持するシステムの機能を測定します。**stress-ng** ツールは、すべてのカーネルインターフェイスに負荷およびストレスをかけるためのストレスワークロードジェネレーターです。これには、ストレッサーと呼ばれるさまざまなストレスメカニズムが含まれています。ストレステストにより、マシンに負荷がかかり、システムが過負荷になっているときに発生するサーマルオーバーランやオペレーティングシステムのバグなどのハードウェアの問題が発生します。

270 以上の異なるテストがあります。これらには、浮動小数点、整数、ビット操作、制御フロー、および仮想メモリーテストを実行する CPU 固有のテストが含まれます。

注記

一部のテストは、設計が不十分なハードウェア上のシステムのサーマルゾーントリップポイントに影響を与える可能性があるため、**stress-ng** ツールの使用には注意が必要です。これは、システムパフォーマンスに影響を与え、過度のシステムスラッシングを引き起こし、停止が困難になる可能性があります。

29.1. CPU 浮動小数点ユニットとプロセッサーデータキャッシュのテスト

浮動小数点ユニットは、浮動小数点算術演算を実行するプロセッサーの機能部分です。浮動小数点ユニットは算術演算を処理し、浮動小数点数または小数の計算を簡単にします。

--matrix-method オプションを使用すると、CPU 浮動小数点演算とプロセッサーデータキャッシュのストレステストを行うことができます。

前提条件

- システムの root 権限がある。

手順

- 1 つの CPU で浮動小数点を 60 秒間テストするには、**--matrix** オプションを使用します。

```
# stress-ng --matrix 1 -t 1m
```

- 複数のストレッサーを複数の CPU で 60 秒間実行するには、**--times** または **-t** オプションを使用します。

```
# stress-ng --matrix 0 -t 1m

stress-ng --matrix 0 -t 1m --times
stress-ng: info:  [16783] dispatching hogs: 4 matrix
stress-ng: info:  [16783] successful run completed in 60.00s (1 min, 0.00 secs)
stress-ng: info:  [16783] for a 60.00s run time:
stress-ng: info:  [16783] 240.00s available CPU time
stress-ng: info:  [16783] 205.21s user time   ( 85.50%)
stress-ng: info:  [16783] 0.32s system time (  0.13%)
stress-ng: info:  [16783] 205.53s total time  ( 85.64%)
stress-ng: info:  [16783] load average: 3.20 1.25 1.40
```

ストレッサーが 0 の特別なモードでは、実行可能な CPU をクエリするため、CPU 番号を指定する必要はなくなります。

必要な合計 CPU 時間は 4 x 60 秒 (240 秒) で、そのうち 0.13% がカーネル、85.50% がユーザー時間、**stress-ng** がすべての CPU の 85.64% を実行します。

- POSIX メッセージキューを使用してプロセス間でメッセージが渡されることをテストするには、**-mq** オプションを使用します。

```
# stress-ng --mq 0 -t 30s --times --perf
```

mq オプションは、POSIX メッセージキューを使用してコンテキストスイッチを強制する特定の数のプロセスを設定します。このストレステストは、データキャッシュミスを少なくすることを目的としています。

29.2. 複数のストレスメカニズムを使用した CPU のテスト

stress-ng ツールは、複数のストレステストを実行します。デフォルトモードでは、指定されたストレッサーメカニズムを並行して実行します。

前提条件

- システムの root 権限がある。

手順

- 次のように、CPU ストレッサーの複数のインスタンスを実行します。

```
# stress-ng --cpu 2 --matrix 1 --mq 3 -t 5m
```

この例では、**stress-ng** は CPU ストレッサーの 2 つのインスタンス、マトリックスストレッサーの 1 つのインスタンス、およびメッセージキューストレッサーの 3 つのインスタンスを実行し、5 分間テストを行います。

- すべてのストレステストを並行して実行するには、**–all** オプションを使用します。

```
# stress-ng --all 2
```

この例では、**stress-ng** はすべてのストレステストの 2 つのインスタンスを並行して実行します。

- 異なるストレッサーをそれぞれ特定の順序で実行するには、**--seq** オプションを使用します。

```
# stress-ng --seq 4 -t 20
```

この例では、**stress-ng** はすべてのストレッサーを 1 つずつ 20 分間実行し、各ストレッサーのインスタンスの数はオンライン CPU の数と一致します。

- テスト実行から特定のストレッサーを除外するには、**-x** オプションを使用します。

```
# stress-ng --seq 1 -x numa,matrix,hdd
```

この例では、**stress-ng** は、**numa**、**hdd**、および **key** ストレッサーメカニズムを除いて、すべてのストレッサーを実行します。

29.3. CPU 発熱量の測定

CPU の発熱量を測定するために、指定されたストレッサーが短時間高温を発生させ、最大発熱量でのシステムの冷却の信頼性と安定性をテストします。**--matrix-size** オプションを使用すると、短時間で CPU 温度を摂氏単位で測定できます。

前提条件

- システムの root 権限がある。

手順

1. 指定された期間、高温で CPU の動作をテストするには、次のコマンドを実行します。

```
# stress-ng --matrix 0 --matrix-size 64 --tz -t 60

 stress-ng: info:  [18351] dispatching hogs: 4 matrix
 stress-ng: info:  [18351] successful run completed in 60.00s (1 min, 0.00 secs)
 stress-ng: info:  [18351] matrix:
 stress-ng: info:  [18351] x86_pkg_temp   88.00 °C
 stress-ng: info:  [18351] acpitz   87.00 °C
```

この例では、**stress-ng** は、60 秒間摂氏 88 度になるようにプロセッサーパッケージのサーマルゾーンを設定します。

2. (オプション) 実行の最後にレポートを印刷するには、**--tz** オプションを使用します。

```
# stress-ng --cpu 0 --tz -t 60

 stress-ng: info:  [18065] dispatching hogs: 4 cpu
 stress-ng: info:  [18065] successful run completed in 60.07s (1 min, 0.07 secs)
 stress-ng: info:  [18065] cpu:
 stress-ng: info:  [18065] x86_pkg_temp   88.75 °C
 stress-ng: info:  [18065] acpitz   88.38 °C
```

29.4. BOGO 操作によるテスト結果の測定

stress-ng ツールは、1秒あたりの bogo 操作を測定することにより、ストレステストのスループットを測定できます。bogo 操作のサイズは、実行されているストレッサーによって異なります。テスト結果は正確ではありませんが、概略のパフォーマンスを提供します。

この測定値を正確なベンチマークメトリックとして使用しないでください。これらの見積もりは、**stress-ng** のビルドに使用されるさまざまなカーネルバージョンまたはさまざまなコンパイラーバージョンでのシステムパフォーマンスの変化を理解するのに役立ちます。**--metrics-brief** オプションを使用して、マシンで利用可能な bogo 操作の合計とマトリックスストレッサーのパフォーマンスを表示します。

前提条件

- システムの root 権限がある。

手順

- bogo 操作でテスト結果を測定するには、**--metrics-brief** オプションを使用します。

```
# stress-ng --matrix 0 -t 60s --metrics-brief

stress-ng: info: [17579] dispatching hogs: 4 matrix
stress-ng: info: [17579] successful run completed in 60.01s (1 min, 0.01 secs)
stress-ng: info: [17579] stressor bogo ops real time usr time sys time   bogo ops/s bogo ops/s
stress-ng: info:  [17579]                 (secs)   (secs)  (secs)  (real time) (usr+sys time)
stress-ng: info:  [17579] matrix  349322   60.00    203.23   0.19      5822.03      1717.25
```

--metrics-brief オプションは、テスト結果と **matrix** ストレッサーによって 60 秒間実行されたリアルタイムの bogo 操作の合計を表示します。

29.5. 仮想メモリーの逼迫の生成

メモリーが不足すると、カーネルはページをスワップに書き込み始めます。**--page-in** オプションを使用して、非常駐ページを強制的に仮想メモリーにスワップバックすることにより、仮想メモリーに負荷をかけることができます。これにより、仮想マシンが高負荷で実行されます。**--page-in** オプションを使用すると、**bigheap**、**mmap**、および仮想マシン (**vm**) ストレッサーに対してこのモードを有効にできます。**--page-in** オプションは、コアにない割り当てられたページをタッチして、強制的にページインさせます。

前提条件

- システムの root 権限がある。

手順

- 仮想メモリーのストレステストを行うには、**--page-in** オプションを使用します。

```
# stress-ng --vm 2 --vm-bytes 2G --mmap 2 --mmap-bytes 2G --page-in
```

この例では、**stress-ng** は、4 GB のメモリー (割り当てられたバッファーサイズ (**--page-in** が有効な 2 x 2 GB の **vm** ストレッサーおよび 2 x 2 GB の **mmap** ストレッサー) よりも小さい) を備えたシステムでメモリー逼迫のテストを行います。

29.6. デバイスでの大きな割り込み負荷のテスト

タイマーを高頻度で実行すると、大きな割り込み負荷が発生する可能性があります。適切に選択されたタイマー周波数を持つ **–timer** ストレッサーは、1秒あたり多くの割り込みを強制する可能性があります。

前提条件

- システムの root 権限がある。

手順

- 割り込み負荷を生成するには、**--timer** オプションを使用します。

```
# stress-ng --timer 32 --timer-freq 1000000
```

この例では、**stress-ng** は 1 MHz で 32 個のインスタンスをテストします。

29.7. プログラムでの深刻なページ障害の生成

stress-ng を使用すると、メモリーに読み込まれていないページで深刻なページ障害を生成することにより、ページ障害率をテストおよび分析できます。新しいカーネルバージョンでは、**userfaultfd** メカニズムは、プロセスの仮想メモリーレイアウトのページ障害について、障害検出スレッドに通知します。

前提条件

- システムの root 権限がある。

手順

- 初期のカーネルバージョンで深刻なページ障害を生成するには、以下のコマンドを使用します。

```
# stress-ng --fault 0 --perf -t 1m
```

- 新しいカーネルバージョンで深刻なページ障害を生成するには、以下のコマンドを使用します。

```
# stress-ng --userfaultfd 0 --perf -t 1m
```

29.8. CPU ストレステストメカニズムの表示

CPU ストレステストには、CPU を使用するメソッドが含まれています。**which** オプションを使用して出力を印刷し、すべてのメソッドを表示できます。

テスト方法を指定しない場合、デフォルトでは、ストレッサーはすべてのストレッサーをラウンドロビン方式でチェックして、各ストレッサーで CPU をテストします。

前提条件

- システムの root 権限がある。

手順

1. 利用可能なすべてのストレッサーメカニズムを印刷するには、**which** オプションを使用します。

```
# stress-ng --cpu-method which

cpu-method must be one of: all ackermann bitops callfunc cdouble cfloat clongdouble
correlate crc16 decimal32 decimal64 decimal128 dither djb2a double euler explog fft
fibonacci float fnv1a gamma gcd gray hamming hanoi hyperbolic idct int128 int64 int32
```

2. **--cpu-method** オプションを使用して、特定の CPU ストレス方式を指定します。

```
# stress-ng --cpu 1 --cpu-method fft -t 1m
```

29.9. 検証モードの使用

verify モードは、テストがアクティブなときに結果を検証します。テスト実行からメモリーの内容の健全性チェックを行い、予期しない障害があれば報告します。

すべてのストレッサーには **verify** モードがなく、このモードを有効にすると、このモードで実行される追加の検証ステップのために、bogo 操作の統計が減少します。

前提条件

- システムの root 権限がある。

手順

- ストレステストの結果を検証するには、**--verify** オプションを使用します。

```
# stress-ng --vm 1 --vm-bytes 2G --verify -v
```

この例では、**stress-ng** は、**--verify** モードが設定された **vm** ストレッサーを使用して、仮想的にマッピングされたメモリーでの完全なメモリーチェックの出力を表示します。メモリーの読み取りと書き込み結果の健全性をチェックします。

第30章 コンテナーの作成と実行

このセクションでは、リアルタイムカーネルを使用してコンテナーを作成および実行する方法を説明します。

前提条件

- **podman** およびその他のコンテナー関連のユーティリティーをインストールしている。
- RHEL での Linux コンテナーの管理に精通している。
- **kernel-rt** パッケージおよびリアルタイム関連のパッケージをインストールしている。

30.1. コンテナーの作成

以下のオプションはすべて、リアルタイムカーネルとメインの RHEL カーネルの両方で使用できます。**kernel-rt** パッケージにより潜在的な決定論が改善され、通常のトラブルシューティングが可能となります。

前提条件

- 管理者権限がある。

手順

以下の手順では、リアルタイムカーネルに関して、Linux コンテナーを設定する方法を説明します。

1. コンテナーに使用するディレクトリーを作成します。以下に例を示します。

```
# mkdir cyclictest
```

2. 対象のディレクトリーに移動します。

```
# cd cyclictest
```

3. コンテナーレジストリーサービスを提供するホストにログインします。

```
# podman login registry.redhat.io
Username: my_customer_portal_login
Password: ***
Login Succeeded!
```

4. 以下の **Dockerfile** を作成します。

```
# vim Dockerfile
FROM rhel8
RUN subscription-manager repos --enable=rhel-8-for-x86_64-rt-rpm
RUN dnf -y install rt-tests
ENTRYPOINT cyclictest --smp -p95
```

5. Dockerfile が含まれるディレクトリーからコンテナーイメージをビルドします。

```
# podman build -t cyclictest .
```

30.2. コンテナーの実行

Dockerfile でビルドしたコンテナーを実行できます。

手順

1. **podman run** コマンドを使用して、コンテナーを実行します。

```
# podman run --device=/dev/cpu_dma_latency --cap-add ipc_lock --cap-add sys_nice --cap-add sys_rawio --rm -ti cyclictest

/dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.08 0.10 0.09 2/947 15

T: 0 ( 8) P:95 I:1000 C: 3209 Min: 1 Act: 1 Avg: 1 Max:  14

T: 1 ( 9) P:95 I:1500 C: 2137 Min: 1 Act: 2 Avg: 1 Max:  23

T: 2 (10) P:95 I:2000 C: 1601 Min: 1 Act: 2 Avg: 2 Max:   7

T: 3 (11) P:95 I:2500 C: 1280 Min: 1 Act: 2 Avg: 2 Max:  72

T: 4 (12) P:95 I:3000 C: 1066 Min: 1 Act: 1 Avg: 1 Max:   7

T: 5 (13) P:95 I:3500 C:  913 Min: 1 Act: 2 Avg: 2 Max:  87

T: 6 (14) P:95 I:4000 C:  798 Min: 1 Act: 1 Avg: 2 Max:   7

T: 7 (15) P:95 I:4500 C:  709 Min: 1 Act: 2 Avg: 2 Max:  29
```

この例は、必須のリアルタイム固有のオプションを指定した **podman run** コマンドを示しています。以下に例を示します。

- 先入れ先出し (FIFO) スケジューラーポリシーは、**--cap-add=sys_nice** オプションを介してコンテナー内で実行されているワークロードで使用できます。このオプションでは、スレッドの CPU アフィニティーを設定することもできます。これは、リアルタイムのワークロードをチューニングする際のもう 1 つの重要な設定ディメンジョンです。

- **--device=/dev/cpu_dma_latency** オプションは、コンテナー内でホストデバイスを利用できるようにします (その後、CPU アイドル時間管理を設定するために cyclictest の負荷によって使用されます)。指定したデバイスを利用できない場合は、以下のようなエラーメッセージが表示されます。
  WARN: stat /dev/cpu_dma_latency failed: No such file or directory

  このようなエラーメッセージが表示された場合は、 podman-run(1) の man ページを参照してください。コンテナーで特定のワークロードを実行するには、別の **podman-run** オプションが役に立ちます。

  場合によっては、**--device=/dev/cpu** オプションを追加してそのディレクトリー階層を追加して、**/dev/cpu/*/msr** などの CPU ごとのデバイスファイルをマッピングする必要があります。

30.3. 関連情報

- RHEL 9 での Linux コンテナーの構築、実行、および管理

- RHEL 9 for Real Time のインストール

第31章 プロセスの優先度の表示

sched_getattr 属性を使用して、プロセスの優先度に関する情報およびプロセスのスケジューリングポリシーに関する情報を表示できます。

前提条件

- 管理者権限がある。

31.1. CHRT ユーティリティー

chrt ユーティリティーは、スケジューラーポリシーおよび優先度を確認して調整します。希望するプロパティーで新しいプロセスを開始するか、実行中のプロセスのプロパティーを変更できます。

関連情報

- **chrt(1)** の man ページ

31.2. CHRT ユーティリティーを使用したプロセス優先度の表示

指定したプロセスの現在のスケジューリングポリシーおよびスケジューリング優先度を表示できます。

手順

- 実行中のプロセスを指定して、**-p** オプションを指定して **chrt** ユーティリティーを実行します。

```
# chrt -p 468
pid 468's current scheduling policy: SCHED_FIFO
pid 468's current scheduling priority: 85

# chrt -p 476
pid 476's current scheduling policy: SCHED_OTHER
pid 476's current scheduling priority: 0
```

31.3. SCHED_GETSCHEDULER() を使用してプロセスの優先度を表示する

リアルタイムプロセスでは、一連の関数を使用してポリシーと優先順位を制御します。**sched_getscheduler()** 関数を使用して、指定したプロセスのスケジューラーポリシーを表示できます。

手順

1. **get_sched.c** ソースファイルを作成し、テキストエディターで開きます。

```
$ {EDITOR} get_sched.c
```

2. ファイルに以下の行を追加します。

```
#include <sched.h>
#include <unistd.h>
#include <stdio.h>
```

```
int main()
{
  int policy;
  pid_t pid = getpid();

  policy = sched_getscheduler(pid);
  printf("Policy for pid %ld is %i.\n", (long) pid, policy);
  return 0;
}
```

policy 変数は、指定されたプロセスのスケジューラーポリシーを保持します。

3. プログラムをコンパイルします。

```
$ gcc get_sched.c -o get_sched
```

4. さまざまなポリシーでプログラムを実行します。

```
$ chrt -o 0 ./get_sched
Policy for pid 27240 is 0.
$ chrt -r 10 ./get_sched
Policy for pid 27243 is 2.
$ chrt -f 10 ./get_sched
Policy for pid 27245 is 1.
```

関連情報

- **sched_getscheduler(2)** の man ページ

31.4. スケジューラーポリシーの有効範囲の表示

sched_get_priority_min() 関数および **sched_get_priority_max()** 関数を使用して、特定のスケジューラーポリシーの有効な優先順位の範囲を確認できます。

手順

1. **sched_get.c** ソースファイルを作成し、テキストエディターで開きます。

```
$ {EDITOR} sched_get.c
```

2. ファイルに以下のコマンドを入力します。

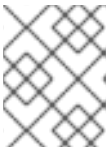
```
#include <stdio.h>
#include <unistd.h>
#include <sched.h>

int main()
{

  printf("Valid priority range for SCHED_OTHER: %d - %d\n",
        sched_get_priority_min(SCHED_OTHER),
        sched_get_priority_max(SCHED_OTHER));
```

```
  printf("Valid priority range for SCHED_FIFO: %d - %d\n",
        sched_get_priority_min(SCHED_FIFO),
        sched_get_priority_max(SCHED_FIFO));

  printf("Valid priority range for SCHED_RR: %d - %d\n",
        sched_get_priority_min(SCHED_RR),
        sched_get_priority_max(SCHED_RR));
  return 0;
}
```



注記

指定されたスケジューラーポリシーがシステムに認識されていない場合、関数は **-1** を返し、**errno** は **EINVAL** に設定されます。

注記

SCHED_FIFO および **SCHED_RR** はどちらも **1** ~ **99** の範囲内の任意の数値にすることができます。POSIX がこの範囲に従う保証はありませんが、移植可能なプログラムはこれらの関数を使用する必要があります。

3. ファイルを保存して、エディターを終了します。
4. プログラムをコンパイルします。

```
$ gcc sched_get.c -o msched_get
```

これで **sched_get** プログラムの準備が整い、保存先のディレクトリーから実行できるようになりました。

関連情報

- **sched_get_priority_min(2)** の man ページ
- **sched_get_priority_max(2)** の man ページ

31.5. プロセスのタイムスライスの表示

SCHED_RR (ラウンドロビン) ポリシーは、**SCHED_FIFO** (先入れ先出し) ポリシーとは少し異なります。**SCHED_RR** は、ラウンドロビンローテーションで同じ優先度を持つ並行プロセスを割り当てます。このようにして、各プロセスにタイムスライスが割り当てられます。**sched_rr_get_interval()** 関数は、各プロセスに割り当てられたタイムスライスを報告します。

注記

POSIX では、この関数が **SCHED_RR** スケジューラーポリシーで実行するように設定されたプロセスでのみ機能する **必要があります** が、**sched_rr_get_interval()** 関数は Linux 上の任意のプロセスのタイムスライス長を取得できます。

タイムスライス情報は **timespec** として返されます。これは、1970 年 1 月 1 日のグリニッジ標準時 00:00:00 の基準時間からの秒数およびナノ秒数です。

```
struct timespec {
  time_t tv_sec;  /* seconds / long tv_nsec; / nanoseconds */
};
```

手順

1. **sched_timeslice.c** ソースファイルを作成し、テキストエディターで開きます。

```
$ {EDITOR} sched_timeslice.c
```

2. 次の行を **sched_timeslice.c** ファイルに追加します。

```
#include <stdio.h>
#include <sched.h>

int main()
{
   struct timespec ts;
   int ret;

   /* real apps must check return values */
   ret = sched_rr_get_interval(0, &ts);

   printf("Timeslice: %lu.%lu\n", ts.tv_sec, ts.tv_nsec);

   return 0;
}
```

3. ファイルを保存して、エディターを終了します。

4. プログラムをコンパイルします。

```
$ gcc sched_timeslice.c -o sched_timeslice
```

5. さまざまなポリシーと優先順位でプログラムを実行します。

```
$ chrt -o 0 ./sched_timeslice
Timeslice: 0.38994072
$ chrt -r 10 ./sched_timeslice
Timeslice: 0.99984800
$ chrt -f 10 ./sched_timeslice
Timeslice: 0.0
```

関連情報

- **nice(2)** の man ページ
- **getpriority(2)** の man ページ
- **setpriority(2)** の man ページ

31.6. プロセスのスケジューリングポリシーおよび関連する属性の表示

sched_getattr() 関数は、PID で識別される、指定されたプロセスに現在適用されているスケジューリングポリシーを照会します。PID がゼロに等しい場合は、呼び出しプロセスのポリシーが取得されます。

size 引数は、ユーザースペースに認識されている **sched_attr** 構造体のサイズを反映する必要があります。カーネルは、**sched_attr::size** をその **sched_attr** 構造体のサイズに入力します。

入力構造体が小さい場合、カーネルは指定されたスペースの外側の値を返します。その結果、システムコールが **E2BIG** エラーで失敗します。他の **sched_attr** フィールドは、The sched_attr structure で説明されているように入力されます。

手順

1. **sched_timeslice.c** ソースファイルを作成し、テキストエディターで開きます。

```
$ {EDITOR} sched_timeslice.c
```

2. 次の行を **sched_timeslice.c** ファイルに追加します。

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <linux/unistd.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <sys/syscall.h>
#include <pthread.h>

#define gettid() syscall(__NR_gettid)

#define SCHED_DEADLINE    6

/* XXX use the proper syscall numbers */
#ifdef __x86_64__
#define __NR_sched_setattr        314
#define __NR_sched_getattr        315
#endif

struct sched_attr {
    __u32 size;
    __u32 sched_policy;
    __u64 sched_flags;

    /* SCHED_NORMAL, SCHED_BATCH */
    __s32 sched_nice;

    /* SCHED_FIFO, SCHED_RR */
    __u32 sched_priority;

    /* SCHED_DEADLINE (nsec) */
    __u64 sched_runtime;
    __u64 sched_deadline;
```

```
    __u64 sched_period;
};

int sched_getattr(pid_t pid,
          struct sched_attr *attr,
          unsigned int size,
          unsigned int flags)
{
    return syscall(__NR_sched_getattr, pid, attr, size, flags);
}

int main (int argc, char **argv)
{
    struct sched_attr attr;
    unsigned int flags = 0;
    int ret;

    ret = sched_getattr(0, &attr, sizeof(attr), flags);
    if (ret < 0) {
       perror("sched_getattr");
       exit(-1);
    }

    printf("main thread pid=%ld\n", gettid());
    printf("main thread policy=%ld\n", attr.sched_policy);
    printf("main thread nice=%ld\n", attr.sched_nice);
    printf("main thread priority=%ld\n", attr.sched_priority);
    printf("main thread runtime=%ld\n", attr.sched_runtime);
    printf("main thread deadline=%ld\n", attr.sched_deadline);
    printf("main thread period=%ld\n", attr.sched_period);

    return 0;
}
```

3. **sched_timeslice.c** ファイルをコンパイルします。

```
$ gcc sched_timeslice.c -o sched_timeslice
```

4. **sched_timeslice** プログラムの出力を確認します。

```
$ ./sched_timeslice
main thread pid=321716
main thread policy=6
main thread nice=0
main thread priority=0
main thread runtime=1000000
main thread deadline=9000000
main thread period=10000000
```

31.7. SCHED_ATTR 構造体

sched_attr 構造体には、指定されたスレッドのスケジューリングポリシーとそれに関連する属性が含まれるか、それらを定義します。**sched_attr** 構造体の形式は次のとおりです。

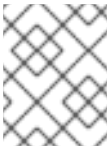
```
struct sched_attr {
 u32 size;
 u32 sched_policy
 u64 sched_flags
 s32 sched_nice
 u32 sched_priority

 /* SCHED_DEADLINE fields */
        u64 sched_runtime
        u64 sched_deadline
        u64 sched_period
    };
```

sched_attr データ構造体

size

バイト単位のスレッドサイズ。構造体のサイズがカーネル構造体よりも小さい場合、追加のフィールドは **0** と見なされます。サイズがカーネル構造体よりも大きい場合、カーネルはすべての追加フィールドを **0** として検証します。



注記

sched_attr 構造体がカーネル構造体よりも大きく、カーネル構造のサイズを含むようにサイズを更新すると、**sched_setattr()** 関数は **E2BIG** エラーで失敗します。

sched_policy

スケジューリングポリシー

sched_flags

プロセスが **fork()** 関数を使用してフォークする場合のスケジューリング動作を制御するのに役立ちます。呼び出し元のプロセスは親プロセスと呼ばれ、新しいプロセスは子プロセスと呼ばれます。有効な値は以下のとおりです。

- **0**: 子プロセスは親プロセスからスケジューリングポリシーを継承します。
- **SCHED_FLAG_RESET_ON_FORK: fork()**: 子プロセスは親プロセスからスケジューリングポリシーを継承しません。代わりに、デフォルトのスケジューリングポリシー **(struct sched_attr){ .sched_policy = SCHED_OTHER, }** に設定されます。

sched_nice

SCHED_OTHER または **SCHED_BATCH** スケジューリングポリシーを使用する際に設定する **nice** 値を指定します。**nice** 値は、**-20** (優先度が高い) から **+19** (優先度が低い) の範囲の数値です。

sched_priority

SCHED_FIFO または **SCHED_RR** をスケジュールするときに設定する静的優先度を指定します。その他のポリシーの場合は、優先度を **0** として指定します。

SCHED_DEADLINE フィールドは、期限スケジューリングの場合にのみ指定する必要があります。

- sched_runtime: 期限スケジューリングの **runtime** パラメーターを指定します。値はナノ秒で表されます。
- sched_deadline: 期限スケジューリングの **deadline** パラメーターを指定します。値はナノ秒で表されます。

- sched_period: 期限スケジューリングの **period** パラメーターを指定します。値はナノ秒で表されます。

第32章 プリエンプション状態の表示

CPU を使用するプロセスは、自発的または非自発的に、使用している CPU を放棄する可能性があります。

32.1. プリエンプション

プロセスは、完了したか、イベント (ディスクからのデータ、キー操作、またはネットワークパケットなど) を待機しているため、CPU に 自主的に優先度を譲ることがあります。

また、プロセスは不本意に CPU に優先度を譲ることもあります。これはプリエンプションと呼ばれ、優先度の高いプロセスが CPU を使用する場合に発生します。

プリエンプションはシステムパフォーマンスに重大な影響を及ぼす可能性があります。また、継続的なプリエンプションにより、スロットリングと呼ばれる状態が発生する可能性があります。この問題は、プロセスが常にプリエンプションされ、プロセスの実行が完了しない場合に発生します。

タスクの優先度を変更すると、自発的なプリエンプションを減らすことができます。

32.2. プロセスのプリエンプション状態の確認

指定されたプロセスの自発的および非自発的なプリエンプションステータスを確認できます。ステータスは **/proc/PID/status** に保存されます。

前提条件

- 管理者権限がある。

手順

- **/proc/PID/status** の内容を表示します。ここで、**PID** はプロセスの ID です。以下に、PID 1000 のプロセスのプリエンプションステータスを示します。

```
# grep voluntary /proc/1000/status
voluntary_ctxt_switches: 194529
nonvoluntary_ctxt_switches: 195338
```

第33章 CHRT ユーティリティーを使用したプロセス優先度の設定

chrt ユーティリティーを使用してプロセスの優先度を設定できます。

前提条件

- 管理者権限がある。

33.1. CHRT ユーティリティーを使用したプロセス優先度の設定

chrt ユーティリティーは、スケジューラーポリシーおよび優先度を確認して調整します。希望するプロパティーで新しいプロセスを開始するか、実行中のプロセスのプロパティーを変更できます。

手順

- プロセスのスケジューリングポリシーを設定するには、適切なコマンドオプションおよびパラメーターを指定して **chrt** コマンドを実行します。次の例では、コマンドの影響を受けるプロセス ID は **1000** であり、優先度 (**-p**) は **50** です。

  ```
  # chrt -f -p 50 1000
  ```

  指定されたスケジューリングポリシーと優先度でアプリケーションを起動するには、アプリケーションの名前と、必要に応じてアプリケーションへのパスを属性と共に追加します。

  ```
  # chrt -r -p 50 /bin/my-app
  ```

  chrt ユーティリティーオプションの詳細は、chrt ユーティリティーのオプション を参照してください。

33.2. CHRT ユーティリティーのオプション

chrt ユーティリティーのオプションには、コマンドオプションと、コマンドのプロセスと優先度を指定するパラメーターが含まれます。

ポリシーオプション

-f
スケジューラーポリシーを **SCHED_FIFO** に設定します。

-o
スケジューラーポリシーを **SCHED_OTHER** に設定します。

-r
スケジューラーポリシーを **SCHED_RR** (ラウンドロビン) に設定します。

-d
スケジューラーポリシーを **SCHED_DEADLINE** に設定します。

-p n
プロセスの優先度を **n** に設定します。
プロセスを SCHED_DEADLINE に設定するときは、**runtime**、**deadline**、および **period** パラメーターを指定する必要があります。

以下に例を示します。

```
# chrt -d --sched-runtime 5000000 --sched-deadline 10000000 --sched-period 16666666 0
video_processing_tool
```

ここでは、以下のようになります。

- **--sched-runtime 5000000** は、ナノ秒単位の実行時間です。
- **--sched-deadline 10000000** は、ナノ秒単位の相対的な期限です。
- **--sched-period 16666666** は、ナノ秒単位の期間です。
- **0** は、**chrt** コマンドに必要な未使用の優先度のプレースホルダーです。

33.3. 関連情報

- **chrt(1)** の man ページ

第34章 ライブラリー呼び出しを使用したプロセスの優先度の設定

chrt ユーティリティーを使用してプロセスの優先度を設定できます。

前提条件

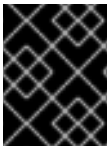
- 管理者権限がある。

34.1. 優先度を設定するためのライブラリー呼び出し

リアルタイムプロセスは、異なる一連のライブラリー呼び出しを使用して、ポリシーおよび優先度を制御します。次のライブラリー呼び出しは、非リアルタイムプロセスの優先度を設定するのに使用されます。

- **nice**
- **setpriority**

これらの関数は、非リアルタイムプロセスの nice 値を調整します。**nice** 値は、プロセッサー上で実行可能な非リアルタイムプロセスのリストにどのように優先度を設定するかをスケジューラーに提案します。リストの上位にあるプロセスは、リスト上で下位のプロセスより先に実行されます。



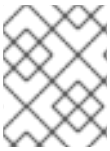
重要

関数には、**sched.h** ヘッダーファイルをインクルードする必要があります。関数からの戻りコードを必ず確認するようにしてください。

34.2. ライブラリー呼び出しを使用したプロセス優先度の設定

スケジューラーポリシーおよびその他のパラメーターは、**sched_setscheduler()** 関数を使用して設定できます。現在、リアルタイムポリシーには、**sched_priority** というパラメーターがあります。このパラメーターは、プロセスの優先度を調整するために使用されます。

sched_setscheduler() 関数には、**sched_setscheduler(pid_t pid, int policy, const struct sched_param *sp);** 形式の 3 つのパラメーターが必要です。



注記

sched_setscheduler(2) の man ページには、エラーコードを含む **sched_setscheduler()** の可能な戻り値がすべて記載されています。

プロセス ID がゼロの場合、**sched_setscheduler()** 関数は呼び出し元のプロセスに作用します。

次のコードの抜粋は、現在のプロセスのスケジューラーポリシーを **SCHED_FIFO** スケジューラーポリシーに設定し、優先度を **50** に設定します。

```
struct sched_param sp = { .sched_priority = 50 };
int ret;

ret = sched_setscheduler(0, SCHED_FIFO, &sp);
if (ret == -1) {
```

```
  perror("sched_setscheduler");
  return 1;
}
```

34.3. ライブラリー呼び出しを使用したプロセス優先度パラメーターの設定

sched_setparam() 関数は、特定プロセスのスケジューリングパラメーターを設定するために使用されます。その後、**sched_getparam()** 関数を使用して確認できます。

スケジューリングポリシーのみを返す **sched_getscheduler()** 関数とは異なり、**sched_getparam()** 関数は指定のプロセスのすべてのスケジューリングパラメーターを返します。

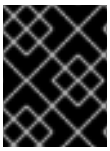
手順

指定したリアルタイムプロセスの優先度を読み取り、それを 2 増加させる以下のコードの抜粋を使用します。

```
struct sched_param sp;
int ret;

ret = sched_getparam(0, &sp);
sp.sched_priority += 2;
ret = sched_setparam(0, &sp);
```

このコードを実際のアプリケーションで使用する場合は、関数からの戻り値を確認し、エラーを適切に処理する必要があります。



重要

優先度の増加に注意してください。この例のように、スケジューラーの優先度を 2 増加させ続けると、最終的に無効な優先度になる可能性があります。

34.4. プロセスのスケジューリングポリシーおよび関連する属性の設定

sched_setattr() 関数は、PID で指定されたインスタンス ID のスケジューリングポリシーとそれに関連する属性を設定します。pid=0 の場合、**sched_setattr()** は呼び出し元のスレッドのプロセスおよび属性に作用します。

手順

- **sched_setattr()** を呼び出して、呼び出しが作用するプロセス ID と、以下のリアルタイムスケジューリングポリシーのいずれかを指定します。

リアルタイムスケジューリングポリシー

SCHED_FIFO
先入れ先出し方式をスケジュールします。

SCHED_RR
ラウンドロビンポリシーをスケジュールします。

SCHED_DEADLINE
期限スケジューリングポリシーをスケジュールします。

Linux は、以下の非リアルタイムスケジューリングポリシーもサポートしています。

非リアルタイムスケジューリングポリシー

SCHED_OTHER
標準のラウンドロビンタイムシェアリングポリシーをスケジュールします。

SCHED_BATCH
プロセスのバッチスタイルの実行をスケジュールします。

SCHED_IDLE
非常に優先度の低いバックグラウンドジョブをスケジュールします。**SCHED_IDLE** は静的優先度 **0** でのみ使用でき、nice 値はこのポリシーに影響を与えません。
このポリシーは、非常に低い優先度 (**SCHED_OTHER** または **SCHED_BATCH** ポリシーを使用した +19 nice 値よりも低い) でジョブを実行することを目的としています。

34.5. 関連情報

- sched_attr 構造体

第35章 リアルタイムカーネルの問題と解決策のスケジューリング

場合によっては、リアルタイムカーネルでのスケジューリングによって影響が発生することがあります。提供される情報を使用すると、リアルタイムカーネル上のスケジューリングポリシー、スケジューラーのスロットリング、およびスレッドの枯渇状態に関する問題と、考えられる解決策を理解できます。

35.1. リアルタイムカーネルのスケジューリングポリシー

リアルタイムスケジューリングポリシーには、共通した大きな特徴が 1 つあります。それは、より優先度の高いスレッドがスレッドに割り込むか、スレッドがスリープまたは I/O の実行によって待機状態になるまで、リアルタイムスケジューリングポリシーは実行を続けるという点です。

SCHED_RR の場合、オペレーティングシステムは、実行中のスレッドに割り込み、同じ **SCHED_RR** 優先度を持つ別のスレッドを実行可能にします。このようないずれの場合も、優先度の低いスレッドが CPU 時間を取得できるようにするポリシーを定義する **POSIX** 仕様によりプロビジョニングが行われることはありません。リアルタイムスレッドのこの特性は、特定の CPU の 100% を独占するアプリケーションの作成が非常に簡単であることを意味します。ただし、これにより、オペレーティングシステムに問題が発生します。たとえば、オペレーティングシステムは、システム全体のリソースと CPU ごとのリソースの両方を管理し、これらのリソースを記述するデータ構造を定期的に調べて、それらのハウスキーピングアクティビティーを実行する必要があります。しかし、コアが **SCHED_FIFO** スレッドによって独占されていると、そのコアはハウスキーピングタスクを実行できません。最終的にシステム全体が不安定になり、クラッシュする可能性があります。

RHEL for Real Time カーネルでは、割り込みハンドラーは優先度が **SCHED_FIFO** のスレッドとして実行されます。デフォルトの優先度は 50 です。割り込みハンドラースレッドよりも高い **SCHED_FIFO** ポリシーまたは **SCHED_RR** ポリシーが割り当てられた cpu-hog スレッドでは、割り込みハンドラーの実行を防ぐことができます。これにより、これらの割り込みによるシグナルのデータを待機しているプログラムが枯渇し、エラーが発生します。

35.2. リアルタイムカーネルでのスケジューラーのスロットリング

リアルタイムカーネルには、リアルタイムタスクで使用する帯域幅の割り当てを可能にする保護メカニズムが搭載されています。この保護メカニズムは、リアルタイムスケジューラーのスロットルと呼ばれています。

リアルタイムスロットリングメカニズムのデフォルト値は、リアルタイムタスクで CPU 時間の 95% を使用できるように定義します。残りの 5% はリアルタイム以外のタスク (**SCHED_OTHER** および同様のスケジューリングポリシーで実行されるタスク) に割り当てられます。1 つのリアルタイムタスクが CPU タイムスロットの 95% を占有している場合、その CPU 上の残りのリアルタイムタスクは実行されないことに注意してください。残りの 5% の CPU 時間は、リアルタイム以外のタスクでのみ使用されます。デフォルト値は、次のようなパフォーマンスの影響を与える可能性があります。

- リアルタイムタスクで使用できる CPU 時間は最大 95% です。これはリアルタイムタスクのパフォーマンスに影響を与える可能性があります。
- リアルタイムタスクは、リアルタイム以外のタスクの実行を許可せず、システムをロックアップしません。

リアルタイムスケジューラーのスロットリングは、**/proc** ファイルシステム内の次のパラメーターによって制御されます。

/proc/sys/kernel/sched_rt_period_us パラメーター

100% の CPU 帯域幅である期間を **µs** (マイクロ秒) 単位で定義します。デフォルト値は 1,000,000 µs、つまり 1 秒です。期間の値が非常に高いか低いと問題が発生する可能性があるため、期間の値の変更は慎重に検討する必要があります。

/proc/sys/kernel/sched_rt_runtime_us パラメーター

すべてのリアルタイムタスクに使用できる合計帯域幅を定義します。デフォルト値は 950,000 µs (0.95 秒) です。これは CPU 帯域幅の 95% です。値を **-1** に設定すると、リアルタイムタスクが CPU 時間を最大 100% 使用するように設定されます。これは、リアルタイムタスクが適切に設定され、無制限のポーリングループなどの明らかな注意点がない場合にのみ適切です。

35.3. リアルタイムカーネルでのスレッド枯渇

スレッドスタベーションは、スレッドがスタベーションしきい値よりも長く CPU 実行キューにあり、進行しない場合に発生します。スレッドスターベーションの一般的な原因は、CPU にバインドされた **SCHED_FIFO** や **SCHED_RR** など、固定優先度のポーリングアプリケーションを実行することです。ポーリングアプリケーションは I/O をブロックしないため、**kworkers** などの他のスレッドがその CPU で実行されなくなる可能性があります。

スレッドスタベーションを減らすための初期の試みは、リアルタイムスロットリングと呼ばれます。リアルタイムスロットリングでは、各 CPU に非リアルタイムタスク専用の実行時間の一部があります。スロットリングのデフォルト設定はオンであり、CPU の 95% がリアルタイムタスク用に割り当てられ、5% がリアルタイム以外のタスク用に予約されます。これは、1 つのリアルタイムタスクが原因でスタベーションが発生している場合には機能しますが、CPU に複数のリアルタイムタスクが割り当てられている場合には機能しません。以下を使用して問題を回避できます。

stalld 機能

stalld 機能は、リアルタイムスロットリングの代替手段であり、スロットリングの欠点の一部を回避します。**stalld** は、システム内の各スレッドの状態を定期的に監視するデーモンであり、指定された時間、実行されずに、実行キューにあるスレッドを探します。**stalld** は、**SCHED_DEADLINE** ポリシーを使用するようにそのスレッドを一時的に変更し、指定された CPU でスレッドにわずかな時間を割り当てます。次にスレッドが実行され、タイムスライスが使用されると、スレッドは元のスケジューリングポリシーに戻り、**stalld** はスレッドの状態を監視し続けます。
ハウスキーピング CPU は、すべてのデーモン、シェルプロセス、カーネルスレッド、割り込みハンドラー、および分離された CPU からディスパッチできるすべての作業を実行する CPU です。リアルタイムスロットリングが無効になっているハウスキーピング CPU の場合、**stalld** はメインワークロードを実行する CPU を監視し、その CPU に **SCHED_FIFO** ビジーループを割り当てます。これは、停止したスレッドを検出し、事前に定義された許容可能な追加ノイズを使用して、必要に応じてスレッドの優先度を向上させるのに役立ちます。リアルタイムのスロットリング機能によってメインワークロードに不当なノイズが発生する場合は、**stalld** が優先される可能性があります。

stalld を使用すると、不足しているスレッドをブーストすることによって導入されるノイズをより正確に制御できます。シェルスクリプト **/usr/bin/throttlectl** は、**stalld** の実行時にリアルタイムスロットリングを自動的に無効にします。**/usr/bin/throttlectl show** スクリプトを使用して、現在のスロットル値を一覧表示できます。

リアルタイムスロットリングの無効化

/proc ファイルシステムの次のパラメーターは、リアルタイムスロットリングを制御します。

- **/proc/sys/kernel/sched_rt_period_us** パラメーターは、期間のマイクロ秒数を指定します。デフォルトは 100 万、つまり 1 秒です。

- **/proc/sys/kernel/sched_rt_runtime_us** パラメーターは、スロットリングが発生する前にリアルタイムタスクで使用できるマイクロ秒数を指定します。デフォルトは 950,000、つまり使用可能な CPU サイクルの 95% です。**echo -1 >**

/proc/sys/kernel/sched_rt_runtime_us コマンドを使用して、値 **-1** を **sched_rt_runtime_us** ファイルに渡すことで、スロットルを無効にできます。