Red Hat Enterprise Linux for Real Time 8

RHEL for Real Time の理解

RHEL for Real Time カーネルの概要

Last Updated: 2023-12-30

Red Hat Enterprise Linux for Real Time 8 RHEL for Real Time の理解

RHEL for Real Time カーネルの概要

法律上の通知

概要

RHEL for Real Time カーネルを調整するための基本的な概念と関連するリファレンスを理解して、レイテンシーの影響を受けやすいアプリケーションでレイテンシーが低く、一貫した応答時間を維持します。

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 FOR REAL TIME のハードウェアプラットフォーム

ハードウェアはシステムの動作方法に影響を与えるため、ハードウェアを正しく設定することは、リアルタイム環境をセットアップする上で重要な役割を果たします。すべてのハードウェアプラットフォームがリアルタイム対応であり、微調整が可能であるとは限りません。微調整を実行する前に、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対応であることを確認する必要があります。

ハードウェアプラットフォームは、ベンダーによって異なります。ハードウェア遅延検出器 (**hwlatdetect**) プログラムを使用して、ハードウェアの適合性をリアルタイムでテストおよび検証できます。プログラムは、レイテンシー検出器カーネルモジュールを制御し、基盤となるハードウェアまたはファームウェアの動作によって引き起こされるレイテンシーを検出するのに役立ちます。

低レイテンシー操作に必要なすべての調整手順が完了しました。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。

前提条件

- RHEL-RT 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すべての調整手順が完了している。システムをシステム管理モード (SMM) に遷移させるシステム管理割り込み (SMI) を低減または削除する手順については、ベンダーのドキュメントを参照してください。

警告

システム管理割り込み (SMI) を完全に無効にすると、重大なハードウェア障害が発生する可能性があるため、完全に無効にすることは避けてください。

1.1. プロセッサーコア

リアルタイムプロセッサーコアは、物理的な中央処理装置 (CPU) であり、マシンコードを実行します。ソケットは、プロセッサーとコンピューターのマザーボードとの間の接続です。ソケットは、プロセッサーが配置されるマザーボードの場所です。プロセッサーには次の 2 つのセットがあります。

- 1つのソケットを占有し、1つのコアが利用可能なシングルコアプロセッサー。
- 1つのソケットを占有し、4 つの使用可能なコアを備えたクアッドコアプロセッサー。

リアルタイム環境を設計するときは、使用可能なコアの数、コア間のキャッシュレイアウト、およびコアが物理的に接続されている方法に注意してください。

複数のコアが利用可能な場合は、スレッドまたはプロセスを使用します。これらの構造を使用せずに作成されたプログラムは、一度に1つのプロセッサーで実行されます。マルチコアプラットフォームは、さまざまなタイプの操作にさまざまなコアを使用することで利点を提供します。

キャッシュ

キャッシュは、全体的な処理時間と決定論に顕著な影響を及ぼします。多くの場合、アプリケーションのスレッドは、データ構造などの共有リソースへのアクセスを同期する必要があります。

tuna コマンドラインツール (CLI) を使用すると、キャッシュレイアウトを決定し、相互作用するスレッドをコアにバインドして、キャッシュを共有することができます。キャッシュ共有は、相互除外プリミティブ (mutex、条件変数、または同様の) とデータ構造が同じキャッシュを使用するようにすることで、メモリー障害を軽減します。

相互接続

システムのコア数を増やすと、相互接続に対する要求が競合する可能性があります。これにより、リアルタイムシステムのコア間で発生する競合を検出するのに役立つ相互接続トポロジーを決定する必要があります。

多くのハードウェアベンダーは、非汎用メモリーアクセス (NUMA) アーキテクチャーとして知られるコアとメモリー間の相互接続の透過的なネットワークを提供するようになりました。

NUMA は、マルチプロセッシングで使用されるシステムメモリー設計であり、メモリーアクセス時間はプロセッサーに対するメモリーの場所によって異なります。NUMA を使用すると、プロセッサーは、別のプロセッサー上のメモリーやプロセッサー間で共有されているメモリーなど、非ローカルメモリーよりも高速に自身のローカルメモリーにアクセスできます。NUMA システムでは、相互接続トポロジーを理解すると、隣接するコアで頻繁に通信するスレッドを配置するのに役立ちます。

taskset ユーティリティーおよび **numactl** ユーティリティーは、CPU トポロジーを決定します。**taskset** は、メモリーノードなどの NUMA リソースなしで CPU アフィニティーを定義し、**numactl** はプロセスと共有メモリーの NUMA ポリシーを制御します。

1.2. 関連情報

- RHEL 9 for Real Time のインストール

第2章 RHEL FOR REAL TIME でのメモリー管理

リアルタイムシステムは仮想メモリーシステムを使用します。ここでは、ユーザースペースアプリケーションによって参照されるアドレスが物理アドレスに変換されます。変換は、基盤となるコンピューティングシステムのページテーブルとアドレス変換ハードウェアの組み合わせによって行われます。プログラムと実際のメモリーの間に変換メカニズムがあることの利点は、オペレーティングシステムが必要な時または CPU の要求に応じてページを交換できることです。

リアルタイムでページをストレージからプライマリーメモリーにスワップするために、以前に使用されたページテーブルエントリーは無効としてマークされます。その結果、通常のメモリープレッシャー下でも、オペレーティングシステムは1つのアプリケーションからページを取得し、別のアプリケーションに渡すことができます。これにより、予期しないシステム動作が発生する可能性があります。

メモリー割り当ての実装には、デマンドページングメカニズムとメモリーロック (**mlock()**) システムコールが含まれます。

注記

異なるキャッシュおよび NUMA ドメインの CPU でデータ情報を共有すると、トラフィックの問題やボトルネックが発生する可能性があります。

マルチスレッドアプリケーションを作成する場合は、データの破損を設計する際にマシントポロジーを考慮することが重要です。トポロジーはメモリー階層であり、CPU キャッシュと NUMA (Non-Uniform Memory Access) ノードが含まれます。

2.1. 需要ページング

デマンドページングは、ページスワッピングを備えたページングシステムに似ています。システムは、必要に応じて、または CPU の要求に応じて、セカンダリーメモリーに保存されているページをロードします。プログラムによって生成されたすべてのメモリーアドレスは、プロセッサーのアドレス変換メカニズムを通過します。次に、アドレスはプロセス固有の仮想アドレスから物理メモリーアドレスに変換されます。これは仮想メモリーと呼ばれます。翻訳メカニズムの 2 つの主要コンポーネントは、ページテーブルと翻訳ルックアップバッファー (TLB) です。

ページテーブル

ページテーブルは、物理メモリーの仮想メモリーから物理メモリーへのマッピングを含むマルチレベルテーブルです。これらのマッピングは、プロセッサーの仮想メモリー変換ハードウェアにより読み取り可能です。

物理アドレスが割り当てられたページテーブルエントリーは、常駐ワーキングセットと呼ばれます。オペレーティングシステムが他のプロセスのためにメモリーを解放する必要がある場合、オペレーティングシステムは常駐ワーキングセットからページを交換できます。ページを交換する場合、そのページ内の仮想アドレスへの参照はページフォールトを作成し、ページの再割り当てを引き起こします。

システムの物理メモリーが極端に少なくなると、スワッププロセスがスラッシュを開始します。これにより、プロセスからページが絶えず盗まれ、プロセスの完了が許可されなくなります。**/proc/vmstat** ファイルで **pgfault** 値を探すことにより、仮想メモリーの統計を監視できます。

Translation Lookaside Buffer

TLB (Translation Lookaside Buffer) は、仮想メモリー変換のハードウェアキャッシュです。TLB を持つプロセッサーコアはいずれも並行して TLB をチェックし、ページテーブルエントリーのメモリー読み取りを開始します。仮想アドレスの TLB エントリーが有効であれば、メモリー読み取りが中止され、TLB の値がアドレス変換に使用されます。

TLB は、参照の局所性の原則に基づいて動作します。つまり、コードが (ループやコール関連の関数など) 長い期間メモリー領域内に留まる場合、TLB 参照はアドレス変換のメインメモリーを回避します。これにより、処理時間が大幅に短縮されます。

決定論的および高速コードを記述する場合は、参照のローカリティーを維持する関数を使用します。これは、再帰ではなくループを使用することを意味します。再帰が避けられない場合は、関数の最後に再帰呼び出しを配置します。これは tail-recursion と呼ばれ、これは比較的小さいメモリー領域でコードが機能し、メインメモリーからのテーブル変換の呼び出しを回避します。

2.2. メジャーページフォールトとマイナーページフォールト

RHEL for Real Time は、物理メモリーをページと呼ばれるチャンクに分割してメモリーを割り当て、それらを仮想メモリーにマップします。リアルタイムで障害が発生するのは、マップされていないか、メモリーで使用できなくなった特定のページをプロセスが必要とする場合です。したがって、障害は基本的に、CPU が必要とするときにページが使用できないことを意味します。プロセスでページフォールトが発生すると、カーネルがこのフォールトを処理するまで、すべてのスレッドがフリーズします。この問題に対処する方法はいくつかありますが、最善の解決策は、ページフォールトを回避するためにソースコードを調整することです。

マイナーページフォールト

リアルタイムでのマイナーページフォールトは、プロセスが初期化される前にメモリーの一部にアクセスしようとすると発生します。このようなシナリオでは、システムはメモリーマップまたはその他の管理構造を埋める操作を実行します。マイナーページフォールトの重大度は、システムの負荷およびその他の要因に依存できますが、通常は短く、影響を及ぼす影響があります。

メジャーページフォールト

リアルタイムでの重大な障害は、システムがメモリーバッファーをディスクと同期させたり、他のプロセスに属するメモリーページをスワップしたり、メモリーを解放するために他の入出力 (I/O) 活動を行わなければならないときに発生します。これは、プロセッサーが、プロセッサーに物理ページが割り当てられていない仮想メモリーアドレスを参照すると発生します。空のページを参照すると、プロセッサーがフォールトを実行し、カーネルコードにページの割り当てを指示します。これにより、すべてレイテンシーが大幅に向上します。

リアルタイムでアプリケーションでパフォーマンスの低下が見られる場合は、**/proc/** ディレクトリーでページ障害に関連するプロセス情報を確認すると便利です。特定のプロセス ID (PID) については、**cat** コマンドを使用して、次の関連エントリーの **/proc/PID/stat** ファイルを表示できます。

- フィールド 2: 実行可能ファイル名。
- フィールド 10: マイナーページ障害の数。
- フィールド 12: 主要なページ障害の数。

次の例は、**cat** コマンドおよび **pipe** 関数を使用してページ障害を表示し、**/proc/PID/stat** ファイルの 2 行目、10 行目、および 12 行目のみを返す方法を示しています。

```
# cat /proc/3366/stat | cut -d\ -f2,10,12
  (bash) 5389 0
```

出力例では、PID 3366 のプロセスは bash であり、5389 のマイナーページフォールトがあり、主なページ障害はありません。

関連情報

- Linux System Programming (Robert Love 著)

2.3. MLOCK() システムコール

メモリーロック (**mlock()**) システムコールを使用すると、呼び出しプロセスが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るようになり、Linux がロックされたメモリーをスワップ空間にページングするのを防ぐことができます。物理ページをページテーブルエントリーに割り当てると、そのページへの参照は比較的高速になります。メモリーロックシステムコールは、**mlock()** および **munlock()** カテゴリーに分類されます。

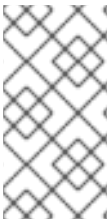
mlock および **munlock** システムコールは、特定の範囲のプロセスアドレスページをロックおよびロック解除します。成功すると、指定された範囲内のページは、**munlock()** システムコールがページのロックを解除するまで、メモリーに常駐したままになります。

mlock() および **munlock()** システムコールは、次のパラメーターを取ります。

- **addr**: アドレス範囲の開始を指定します。
- **len**: アドレス空間の長さをバイト単位で指定します。

成功すると、**mlock()** および **munlock()** システムコールは 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。

mlockall() および **munlockall()** システムコールは、すべてのプログラム空間をロックまたはロック解除します。



注記

mlock() システムコールは、プログラムがページ I/O を持たないことを保証しません。データがメモリー内にとどまることを保証しますが、同じページにとどまることを保証することはできません。**move_pages** やメモリーコンパクタなどの他の関数は、**mlock()** の使用に関係なくデータを移動できます。

メモリーロックはページベースで行われ、スタックしません。動的に割り当てられた 2 つのメモリーセグメントが、**mlock()** または **mlockall()** によって 2 回ロックされた同じページを共有している場合、1 つの **munlock()** または **munlockall()** システムコールを使用してロックを解除します。そのため、二重ロックまたは単一ロック解除の問題を回避するために、アプリケーションがロック解除するページに注意することが重要です。

以下は、二重ロックまたは単一ロック解除の問題を軽減するための最も一般的な 2 つの回避策です。

- 割り当てられたメモリー領域とロックされたメモリー領域を追跡し、ページのロックを解除する前にページ割り当ての数を検証するラッパー関数を作成します。これは、デバイスドライバーで使用されるリソースカウントの原則です。
- ページの二重ロックを回避するために、ページサイズと配置に基づいてメモリーを割り当てます。

関連情報

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

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

2.4. 共有ライブラリー

RHEL for Real Time の共有ライブラリーは、動的共有オブジェクト (DSO) と呼ばれ、関数と呼ばれるコンパイル済みのコードブロックのコレクションです。これらの関数は複数のプログラムで再利用可能であり、実行時またはコンパイル時にロードされます。

Linux は、次の 2 つのライブラリークラスをサポートしています。

- 動的ライブラリーまたは共有ライブラリー: 実行可能ファイルの外部に個別のファイルとして存在します。これらのファイルはメモリーにロードされ、実行時にマップされます。
- 静的ライブラリー: コンパイル時にプログラムに静的にリンクされたファイルです。

ld.so ダイナミックリンカーは、プログラムに必要な共有ライブラリーをロードしてから、コードを実行します。DSO 関数は、ライブラリーをメモリーに 1 回ロードすると、複数のプロセスがプロセスのアドレス空間にマッピングすることでオブジェクトを参照できます。**LD_BIND_NOW** 変数を使用して、コンパイル時にロードするようにダイナミックライブラリーを設定できます。

プログラムの初期化の前にシンボルを評価すると、パフォーマンスが向上する可能性があります。これは、アプリケーションの実行時に評価すると、メモリーページが外部ディスクにある場合に遅延が発生する可能性があるためです。

関連情報

- **ld.so(8)** man ページ

2.5. 共有メモリー

RHEL for Real Time では、共有メモリーは複数のプロセス間で共有されるメモリー空間です。プログラムスレッドを使用すると、1 つのプロセスコンテキストで作成されたすべてのスレッドが同じアドレス空間を共有できます。これにより、すべてのデータ構造にスレッドがアクセスできるようになります。POSIX 共有メモリー呼び出しを使用すると、アドレス空間の一部を共有するようにプロセスを設定できます。

以下のサポートされている POSIX 共有メモリー呼び出しを使用できます。

- **shm_open()**: 新しい POSIX 共有メモリーオブジェクトを作成して開くか、既存の POSIX 共有メモリーオブジェクトを開きます。
- **shm_unlink()**: POSIX 共有メモリーオブジェクトのリンクを解除します。
- **mmap()**: 呼び出しプロセスの仮想アドレス空間に新しいマッピングを作成します。

注記

System V IPC **shmem()** の一連の呼び出しを使用して 2 つのプロセス間でメモリー領域を共有するメカニズムは廃止され、RHEL for Real Time ではサポートされなくなりました。

関連情報

- **shm_open(3)** man ページ
- **shm_overview(7)** man ページ
- **mmap(2)** man ページ

第3章 RHEL FOR REAL TIME のハードウェア割り込み

リアルタイムシステムは、その動作の過程で多くの割り込みを受けます。その中には、定期的にメンテナンスとシステムスケジューリング決定を行う半規則的な「タイマー」割り込みが含まれます。また、マスク不可割り込み (NMI) やシステム管理割り込み (SMI) などの特殊な種類の割り込みを受け取る場合もあります。ハードウェア割り込みは、注意が必要なシステムの物理的状態の変化を示すためにデバイスによって使用されます。たとえば、ハードディスクが一連のデータブロックを読み取ったことを通知したり、ネットワークデバイスがネットワークパケットを含むバッファーを処理した場合などです。

リアルタイムで割り込みが発生すると、システムはアクティブなプログラムを停止し、割り込みハンドラーを実行します。

リアルタイムでは、ハードウェア割り込みは割り込み番号で参照されます。これらの番号は、割り込みを作成したハードウェアの部分にマッピングされます。これにより、システムが割り込みを作成したデバイスと、その発生時を監視できるようになります。リアルタイムで割り込みが発生すると、システムはアクティブなプログラムを停止し、割り込みハンドラーを実行します。ハンドラーは、実行中の他のプログラムおよびシステムアクティビティーをプリエンプトします。これにより、システム全体の速度が低下し、遅延が発生する可能性があります。

RHEL for Real Time は、パフォーマンスを向上させ、レイテンシーを短縮するために、割り込みの処理方法を変更します。**cat/proc/interrupts** コマンドを使用すると、結果を出力して、発生したハードウェア割り込みのタイプ、受信した割り込みの数、割り込みのターゲット CPU、および割り込みを生成しているデバイスを表示できます。

3.1. レベル信号割り込み

リアルタイムでは、レベル信号割り込みは、電圧遷移を提供する専用の割り込みラインを使用します。デバイスコントローラーは、割り込み要求ラインで信号をアサートすることによって割り込みを発生させます。割り込みラインは、バイナリー 1 またはバイナリー 0 を表す 2 つの電圧のいずれかを送信します。

割り込み信号が回線から送信されると、CPU がリセットするまでその状態のままになります。CPU は状態保存を実行し、割り込みをキャプチャーして、割り込みハンドラーをディスパッチします。割り込みハンドラーは、割り込みの原因を特定し、必要なサービスを実行して割り込みをクリアし、デバイスの状態を復元します。レベル信号による割り込みは、実装は複雑ですが、信頼性が高く、複数のデバイスをサポートします。

3.2. メッセージシグナル割り込み

リアルタイムでは、多くのシステムがメッセージシグナル割込み (MSI) を使用します。これは、パケットまたはメッセージベースの電気バスに専用のメッセージとして信号を送信します。このタイプのバスの一般的な例として、Peripheral Component Interconnect Express (PCI Express または PCIe) があります。これらのデバイスは、PCIe ホストコントローラーが割り込みメッセージとして解釈するメッセージタイプを送信します。ホストコントローラーはメッセージを CPU に送信します。

リアルタイムでは、ハードウェアに応じて、PCIe システムは次のいずれかを実行します。

- PCIe ホストコントローラーと CPU の間で専用の割り込みラインを使用して信号を送信します。
- CPU HyperTransport バスを介してメッセージを送信します。

リアルタイムでは、PCIe システムはレガシーモードで動作することもできます。レガシーモードでは、レガシー割り込み行は、古いオペレーティングシステムをサポートするために実装されます。または、カーネルコマンドライン上のオプション **pci=nomsi** を使用して Linux カーネルを起動することも

できます。

3.3. マスク不可割り込み

リアルタイムでは、マスク不可割り込みは、システムの標準的な割り込みマスキング技術が無視できないハードウェア割り込みです。NMI は、マスク可能な割り込みより優先度が高くなります。NMI は、回復不可能なハードウェアエラーの注意を促すために発生します。

リアルタイムでは、NMI は、一部のシステムでハードウェアモニターとしても使用されます。プロセッサーが NMI を受信すると、割り込みベクターが指す NMI ハンドラーを呼び出すことにより、NMI を即座に処理します。指定された時間後に割り込みがトリガーされないなど、特定の条件を満たす場合、NMI ハンドラーは問題に関する警告やデバッグの情報を生成する可能性があります。これは、システムのロックアップを特定し、回避するのに役立ちます。

リアルタイムでは、マスク可能な割り込みは、割り込みマスクレジスターのビットマスクにビットを設定することで無視できるハードウェア割り込みです。CPU は、重要な処理中にマスク可能な割り込みを一時的に無視できます。

3.4. システム管理割り込み

リアルタイムでは、システム管理割り込み (SMI) は、レガシーハードウェアデバイスエミュレーションなどの拡張機能を提供し、システム管理タスクにも使用できます。SMI は、特殊な電気信号線を使用し、通常はマスクできないという点で、マスク不可割り込み (NMI) に似ています。SMI が発生すると、CPU はシステム管理モード (SMM) に入ります。このモードでは、SMI を処理するために特別な低レベルハンドラーが実行されます。通常、SMM はシステム管理ファームウェアから直接提供されます。通常は BIOS または EFI です。

リアルタイムの SMI は、レガシーのハードウェアエミュレーションを提供するために最もよく使用されます。一般的な例は、ディスケットドライブを模倣することです。ディスケットドライブが接続されていない場合、オペレーティングシステムはディスケットへのアクセスを試み、SMI をトリガーします。このシナリオでは、ハンドラーが代わりにエミュレートされたデバイスをオペレーティングシステムに提供します。次に、オペレーティングシステムは、エミュレーションをレガシーデバイスとして扱います。

リアルタイムでは、SMI は、オペレーティングシステムが直接関与することなく実行されるため、システムに悪影響を与える可能性があります。不適切に記述された SMI 処理ルーチンは、数ミリ秒の CPU 時間を消費する可能性があり、オペレーティングシステムがハンドラーをプリエンプションできない可能性があります。これにより、他の点では適切に調整された応答性の高いシステムで、定期的に大きな遅延が発生する可能性があります。ベンダーは SMI ハンドラーを使用して CPU 温度およびファン制御を管理する場合があるため、それらを無効にできない場合があります。このような状況では、これらの割り込みを使用するときに発生する問題をベンダーに通知する必要があります。

リアルタイムでは、**hwlatdetect** ユーティリティーを使用して SMI を分離できます。**rt-tests** パッケージで入手できます。このユーティリティーは、CPU が SMI 処理ルーチンによって使用されている期間を測定します。

3.5. 高度なプログラミング可能割り込みコントローラー

Intel Corporation によって開発された高度なプログラム可能な割り込みコントローラー (APIC) は、次の機能を提供します。

- 大量の割り込みを処理して、それぞれを特定の CPU セットにルーティングします。
- CPU 間通信をサポートするため、複数のデバイスが単一の割り込み線を共有する必要がなくなります。

リアルタイムの APIC は、一連のデバイスとテクノロジーを表し、スケーラブルかつ管理可能な方法で多数のハードウェア割り込みを生成し、ルーティングして、処理します。これは、各システム CPU に組み込まれたローカルの APIC と、ハードウェアデバイスに直接接続されている入出力 APIC の組み合わせを使用します。

リアルタイムで、ハードウェアデバイスが割り込みを生成すると、接続された I/O APIC が割り込みを検出し、システム APIC バスを介して特定の CPU にルーティングします。オペレーティングシステムは、IO-APIC がデバイスに接続されていることを認識し、そのデバイス内の回線に割り込みます。Advanced Configuration and Power Interface Differentiated System description Table (ACPI DSDT) には、ホストシステムのマザーボードと周辺コンポーネントの特定の接続に関する情報が含まれて、デバイスは利用可能な割り込みソースに関する情報を提供します。これら 2 つのデータを組み合わせて割り込み階層全体に関する情報を提供します。

RHEL for Real Time は、階層で接続されたシステム APIC を使用し、特定の CPU や CPU をターゲットにするのではなく、負荷分散された方法で CPU に割り込みを提供することで、複雑な APIC ベースの割り込み管理ストラテジーをサポートします。

第4章 RHEL FOR REAL TIME プロセスおよびスレッド

オペレーティングシステムの RHEL for Real Time の主な要素は、最小の割り込みレイテンシーおよび最小のスレッドスイッチングレイテンシーです。すべてのプログラムはスレッドおよびプロセスを使用しますが、RHEL for Real Time は、標準の Red Hat Enterprise Linux とは異なる方法でそれらを処理します。

リアルタイムで並列処理を使用すると、タスクの実行およびレイテンシーの効率を高めることができます。並列処理とは、CPU のマルチコアインフラストラクチャーを使用して、複数のタスクまたは複数のサブタスクを同時に実行することです。

4.1. PROCESSES

リアルタイムのプロセスは、簡単に言えば、実行中のプログラムです。プロセスという用語は、複数のスレッドを含む可能性のある独立したアドレス空間を指します。あるアドレス空間内で実行中の 1 つ以上のプロセスの概念が開発されると、Linux は別のプロセスでアドレス空間を共有するプロセス構造に移行していました。これは、プロセスデータ構造が小さいと機能します。

UNIX® スタイルのプロセス設定には、次のものが含まれます。

- 仮想メモリーのアドレスマッピング
- 実行コンテキスト (PC、スタック、レジスター)
- 状態およびアカウント情報

リアルタイムでは、各プロセスは、親スレッドと呼ばれることが多い単一のスレッドで始まります。**fork()** システムコールを使用して、親スレッドから追加のスレッドを作成できます。**fork()** は、新しいプロセス ID を除いて、親プロセスと同じ新しい子プロセスを作成します。子プロセスは、作成プロセスとは独立して実行します。親プロセスおよび子プロセスは同時に実行できます。**fork()** と **exec()** のシステムコールの違いは、**fork()** が親プロセスのコピーである新しいプロセスを開始し、**exec()** が現在のプロセスイメージを新しいプロセスイメージに置き換えることです。

リアルタイムでは、**fork()** システムコールが成功すると、子プロセスのプロセス ID が返され、親プロセスはゼロ以外の値を返します。エラーの場合は、エラー番号を返します。

4.2. THREADS

リアルタイムでは、プロセス内に複数のスレッドが存在する可能性があります。プロセスのすべてのスレッドは、その仮想アドレス空間およびシステムリソースを共有します。スレッドは、以下を含むスケジュール可能なエンティティーです。

- プログラムカウンター (PC)
- レジスターコンテキスト
- スタックポインター

リアルタイムにおける並列処理を作成するための潜在的なメカニズムは次のとおりです。

- **fork()** および **exec()** 関数呼び出しを使用して、新しいプロセスを作成します。**fork()** 呼び出しは、呼び出されたプロセスの完全な複製を作成し、一意のプロセス識別子を持ちます。
- Posix スレッド (**pthreads**) API を使用して、実行中のプロセス内に新しいスレッドを作成します。

リアルタイムスレッドをフォークする前に、コンポーネントの相互作用レベルを評価する必要があります。新しいアドレス空間を作成し、それを新しいプロセスとして実行することは、コンポーネントが互いに独立している場合、または相互作用が少ない場合に役立ちます。コンポーネントがデータを共有したり頻繁に通信したりする必要がある場合は、1つのアドレス空間内でスレッドを実行する方が効率的です。

リアルタイムでは、**fork()** システムコールは、成功すると値 0 を返します。エラーの場合は、エラー番号を返します。

4.3. 関連情報

- **fork(2)** man ページ
- **exec(2)** man ページ

第5章 RHEL FOR REAL TIME のアプリケーションタイムスタンプ

アプリケーションが **timestamps** 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。

5.1. 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。

5.2. 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。

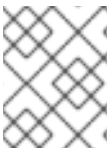
5.3. CLOCK_GETTIME() 関数

以下のコードは、**CLOCK_MONOTONIC_COARSE** POSIX クロックで **clock_gettime()** 機能を使用したコード例を示しています。

```
#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

5.4. 関連情報

- **clock_gettime()** man ページ

第6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。

6.1. 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。

6.2. 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6.1 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7章 RHEL FOR REAL TIME のアフィニティー

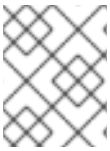
リアルタイムでは、システムの各スレッドおよび割り込みソースには、プロセッサーアフィニティープロパティーがあります。オペレーティングシステムスケジューラーは、この情報を使用して、どの CPU で、どのスレッドおと割り込みを実行するのかを決めます。

リアルタイムのアフィニティーは、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で、プロセスが事前定義された CPU の選択で実行されるように指示できます。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。

より一般的なアフィニティー設定には、以下が含まれます。

- すべてのシステムプロセス用に CPU コアを 1 つ予約し、残りのコアでアプリケーションを実行できるようにします。
- 同じ CPU でスレッドアプリケーションと指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) を許可します。
- 各 CPU のペアプロデューサーおよびコンシューマースレッド。



注記

アフィニティーの設定は、期待される良い動作のために、プログラムと連動して設計する必要があります。

7.1. プロセッサーのアフィニティー

リアルタイムでは、プロセスはデフォルトで任意の 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、事前に選択した CPU で実行するようにプロセスを設定できます。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。

システム上でアフィニティーをチューニングするためのリアルタイムプラクティスは、アプリケーションの実行に必要なコア数を決定してから、それらのコアを分離することです。これは、Tuna ツール、またはビットマスク値を変更するシェルスクリプトを使用して実現できます。

この **taskset** コマンドは、プロセスのアフィニティーを変更するのに使用でき、**/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。**-p** オプションまたは **--pid** オプション、およびそのプロセスのプロセス識別子 (PID) を指定して **taskset** コマンドを使用すると、プロセスのアフィニティーをチェックします。

-c オプションまたは **--cpu-list** オプションは、ビットマスクとしてではなく、コアの数値リストを表示します。アフィニティーは、特定のプロセスをバインドする CPU の数を指定することで設定できます。たとえば、以前に CPU 0 または CPU 1 のいずれかを使用していたプロセスの場合は、CPU 1 でのみ実行できるようにアフィニティーを変更できます。**taskset** コマンドに加えて、**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。

関連情報

- **taskset(1)** の man ページ

- **sched_setaffinity(2)** man ページ

7.2. SCHED_DEADLINE および CPUSETS

カーネルのデッドラインスケジューリングクラス (**SCHED_DEADLINE**) は、期限が制限された散発的なタスクに対して、Early Deadline First Scheduler (EDF) を実装します。ジョブ期限に従ってタスクに優先順位を付けます。つまり、最も早い絶対期限が最初になります。EDF スケジューラーに加えて、期限スケジューラーは定帯域幅サーバー (CBS) も実装します。CBS アルゴリズムは、リソース予約プロトコルです。

CBS は、各タスクがすべての期間 **(T)** で実行時間 **(Q)** を受け取ることを保証します。タスクのすべてのアクティブ化の開始時に、CBS はタスクの実行時間を補充します。ジョブが実行すると、**runtime** が消費され、タスクが **runtime** を使い果たした場合、タスクは抑制され、スケジュールが解除されます。スロットリングメカニズムは、単一のタスクがそのランタイムを超えて実行されるのを防ぎ、他のジョブのパフォーマンスの問題を回避するのに役立ちます。

リアルタイムでは、**deadline** タスクによるシステムの過負荷を回避するために、**deadline** スケジューラーは、タスクが **deadline scheduler** で実行するように設定されるたびに実行される受け入れテストを実装します。受け入れテストは、**SCHED_DEADLINE** タスクが **kernel.sched_rt_runtime_us/kernel.sched_rt_period_us** ファイルで指定されているよりも多くの CPU 時間を使用しないことを保証します。これは、デフォルトでは 1 秒で 950 ミリ秒です。

第8章 RHEL FOR REAL TIME のスレッド同期メカニズム

リアルタイムでは、2 つ以上のスレッドが同時に共有リソースにアクセスする必要がある場合、スレッドはスレッド同期メカニズムを使用して調整します。スレッドの同期により、一度に 1 つのスレッドのみが共有リソースを使用するようになります。Linux で使用される 3 つのスレッド同期メカニズムは、ミューテックス、バリア、および条件変数 (**condvars**) です。

8.1. ミューテックス

ミューテックスは、相互排除という用語に由来します。相互排除オブジェクトは、リソースへのアクセスを同期します。これは、一度に 1 つのスレッドのみがミューテックスを取得できるようにするメカニズムです。

mutex アルゴリズムは、コードの各セクションへのシリアルアクセスを作成するため、一度に 1 つのスレッドだけがコードを実行します。ミューテックスは、**mutex** 属性オブジェクトと呼ばれる属性オブジェクトを使用して作成されます。これは抽象オブジェクトであり、実装するために選択した POSIX オプションに依存するいくつかの属性が含まれています。属性オブジェクトは、**pthread_mutex_t** 変数で定義されます。オブジェクトは、ミューテックスに定義された属性を格納します。成功すると、**pthread_mutex_init(&my_mutex, &my_mutex_attr)** 関数、**pthread_mutexattr_setrobust()** 関数および **pthread_mutexattr_getrobust()** 関数は 0 を返します。エラーが発生すると、エラー番号が返されます。

リアルタイムでは、属性オブジェクトを保持して同じ型のミューテックスをさらに初期化するか、属性オブジェクトをクリーンアップ (破壊) 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。ミューテックスには、標準タイプと高度なタイプのミューテックスが含まれます。

標準ミューテックス

リアルタイム標準のミューテックスは、プライベート、非再帰的、非堅牢、非優先度継承が可能なミューテックスです。**pthread_mutex_init(&my_mutex, &my_mutex_attr)** を使用して **pthread_mutex_t** を初期化すると、標準のミューテックスが作成されます。標準のミューテックスタイプを使用する場合、アプリケーションは、**pthreads** API および RHEL for Real Time カーネルによって提供される利点の恩恵を受けない場合があります。

高度なミューテックス

追加機能で定義されたミューテックスは、高度なミューテックスと呼ばれます。高度な機能には、優先度継承、ミューテックスの堅牢な動作、共有およびプライベートミューテックスが含まれます。たとえば、ロバストミューテックスの場合は、**pthread_mutexattr_setrobust()** 関数を初期化すると、ロバスト属性が設定されます。同様に、属性 **PTHREAD_PROCESS_SHARED** を使用すると、スレッドが割り当てられたメモリーにアクセスできる場合に限り、任意のスレッドがミューテックスで動作できるようになります。属性 **PTHREAD_PROCESS_PRIVATE** は、プライベートミューテックスを設定します。

非堅牢なミューテックスは自動的に解放されず、手動で解放するまでロックされたままになります。

関連情報

- **futex(7)** man ページ
- **pthread_mutex_destroy(P)** man ページ

8.2. バリア

バリアは、他のスレッドの同期方法と比較すると、非常に異なる方法で動作します。バリアは、すべてのスレッドとプロセスがこのバリアに到達するまで、すべてのアクティブなスレッドが停止するコード

内のポイントを定義します。バリアは、実行中のアプリケーションが実行を続行する前にすべてのスレッドが特定のタスクを完了していることを確認する必要がある状況で使用されます。

リアルタイムのバリアミューテックスは、次の 2 つの変数を取ります。

- 最初の変数は、バリアの **stop** と **pass** の状態を記録します。
- 2 番目の変数は、バリアに入るスレッドの総数を記録します。

バリアは、指定された数のスレッドが定義されたバリアに達したときにのみ **pass** するように状態を設定します。バリア状態が **pass** するように設定されると、スレッドとプロセスはさらに進みます。**pthread_barrier_init()** 関数は、定義されたバリアを使用するために必要なリソースを割り当て、**attr** 属性オブジェクトによって参照される属性でバリアを初期化します。

成功すると、**pthread_barrier_init()** 関数および **pthread_barrier_destroy()** 関数はゼロ値を返します。エラーの発生時に、エラー番号が返されます。

8.3. 条件変数

リアルタイムでは、条件変数 (**condvar**) は、POSIX スレッドの設定で、特定条件の達成を待機してから続行します。一般に、通知された状態は、スレッドが別のスレッドと共有するデータの状態に関連しています。たとえば、**condvar** を使用して、処理キューへのデータエントリーと、キューからのそのデータの処理を待機しているスレッドを通知できます。**pthread_cond_init()** 関数を使用して、条件変数を初期化できます。

成功すると、**pthread_cond_init()** 関数、**pthread_cond_wait()** 関数、および **pthread_cond_signal()** 関数はゼロ値を返します。エラーの場合は、エラー番号を返します。

8.4. ミューテックスクラス

前述のミューテックスオプションは、アプリケーションの作成または移植時に考慮すべきミューテックスクラスに関するガイダンスを提供します。

表8.1 ミューテックスオプション

高度なミューテックス	説明
共有ミューテックス	特定の時間にミューテックスを取得するための複数のスレッドの共有アクセスを定義します。共有ミューテックスにより遅延が発生する可能性があります。属性は **PTHREAD_PROCESS_SHARED** です。
プライベートミューテックス	同じプロセス内で作成されたスレッドのみがミューテックスにアクセスできるようにします。属性は **PTHREAD_PROCESS_PRIVATE** です。
リアルタイム優先度の継承	優先度の低いタスクの優先度を、現在の優先度の高いタスクよりも高く設定します。タスクが完了すると、リソースが解放され、タスクは元の優先度に戻り、優先度の高いタスクを実行できるようになります。属性は **PTHREAD_PRIO_INHERIT** です。

高度なミューテックス	説明
強固なミューテックス	所有しているスレッドが停止したときに自動的に解放されるように堅牢なミューテックスを設定します。文字列 **PTHREAD_MUTEX_ROBUST_NP** の値サブ文字列 **NP** は、堅牢なミューテックスが非 POSIX であるか、移植性がないことを示します

関連情報

- **futex(7)** man ページ

8.5. スレッドの同期機能

前述の関数タイプのリストと説明は、リアルタイムカーネルのスレッド同期メカニズムに使用する関数に関する情報を提供します。

表8.2 関数

機能	説明
pthread_mutexattr_init(&my_mutex_attr)	**attr** で指定された属性でミューテックスを開始します。**attr** が NULL の場合は、デフォルトのミューテックス属性が適用されます。
pthread_mutexattr_destroy(&my_mutex_attr)	指定されたミューテックスオブジェクトを破棄します。**pthread_mutex_init()** を使用して再初期化できます。
pthread_mutexattr_setrobust()	ミューテックスの **PTHREAD_MUTEX_ROBUST** 属性を指定します。**PTHREAD_MUTEX_ROBUST** 属性は、ミューテックスのロックを解除せずに停止できるスレッドを定義します。このミューテックスを所有するための将来の呼び出しは自動的に成功し、値 **EOWNERDEAD** を返し、前のミューテックス所有者がもう存在しないことを示します。
pthread_mutexattr_getrobust()	ミューテックスの **PTHREAD_MUTEX_ROBUST** 属性をクエリーします。
pthread_barrier_init()	属性オブジェクト **attr** を使用してバリアを使用および初期化するのに必要なリソースを割り当てます。**attr** が NULL の場合は、デフォルト値が適用されます。
pthread_cond_init()	条件変数を初期化します。引数 **cond** は、条件変数属性オブジェクト **attr** の属性で開始するオブジェクトを定義します。**attr** が NULL の場合は、デフォルト値が適用されます。
pthread_cond_wait()	別のスレッドからシグナルを受信するまで、スレッドの実行をブロックします。さらに、この関数を呼び出すと、ブロックする前にミューテックスの関連するロックも解放されます。引数 **cond** は、ブロックするスレッドの **pthread_cond_t** オブジェクトを定義します。**mutex** 引数は、ブロックを解除するミューテックスを指定します。

機能	説明
pthread_cond_signal()	指定された条件変数でブロックされているスレッドの少なくとも 1 つをブロック解除します。引数 **cond** は、**pthread_cond_t** オブジェクトを使用してスレッドのブロックを解除することを指定します。

第9章 RHEL FOR REAL TIME のソケットオプション

リアルタイムソケットは、UNIX ドメインやループバックデバイスなどの同じシステム、またはネットワークソケットなどの異なるシステム上の 2 つのプロセス間の双方向のデータ転送メカニズムです。

伝送制御プロトコル (TCP) は最も一般的なトランスポートプロトコルであり、一定の通信を必要とするサービスの一貫した低遅延を実現するため、または優先度の低い制限された環境でソケットをコルクするためによく使用されます。

新しいアプリケーション、ハードウェア機能、およびカーネルアーキテクチャーの最適化により、TCP は変更を効果的に処理するための新しいアプローチを導入する必要があります。新しいアプローチは、不安定なプログラム動作を引き起こす可能性があります。基盤となるオペレーティングシステムコンポーネントを変更するとプログラムの動作も変更となるため、慎重に扱う必要があります。

TCP でのこのような動作の一例は、小さなバッファーの送信の遅延です。これにより、それらを 1 つのネットワークパケットとして送信できます。TCP への小さな書き込みのバッファーを行い一度に送信することは一般的に適切に機能しますが、レイテンシーを生み出す可能性もあります。リアルタイムアプリケーションの場合、**TCP_NODELAY** ソケットオプションは遅延を無効にし、準備ができ次第小さな書き込みを送信します。

データ転送に関連するソケットオプションは、**TCP_NODELAY** および **TCP_CORK** です。

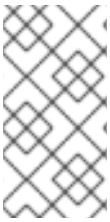
9.1. TCP_NODELAY ソケットオプション

TCP_NODELAY ソケットオプションは、Nagle のアルゴリズムを無効にします。**setsockopt** ソケット API 関数を使用して **TCP_NODELAY** を設定すると、準備が整うとすぐに、複数の小さなバッファー書き込みが個別のパケットとして送信されます。

送信前に連続したパケットを作成することにより、論理的に関連する複数のバッファーを単一のパケットとして送信すると、遅延およびパフォーマンスが向上します。または、メモリーバッファーが論理的に関連しているが連続していない場合は、I/O ベクトルを作成し、**TCP_NODELAY** が有効になっているソケットで **writev** を使用してカーネルに渡すことができます。

次の例は、**setsockopt** ソケット API を介して **TCP_NODELAY** を有効にする方法を示しています。

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



注記

TCP_NODELAY を効果的に使用するには、論理的に関連する小さなバッファー書き込みを回避します。**TCP_NODELAY** を使用すると、小さな書き込みにより、TCP が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する可能性があります。

関連情報

- **sendfile(2)** man ページ

9.2. TCP_CORK ソケットオプション

TCP_CORK オプションは、ソケット内のすべてのデータパケットを収集し、バッファーが指定された制限を満たすまでそれらを送信しないようにします。これにより、**TCP_CORK** が無効になっている場合に、アプリケーションがカーネル領域にパケットを作成し、データを送信できるようになりま

す。**TCP_CORK** は、**setsocketopt()** 関数を使用してソケットファイル記述子に設定されます。プログラムを開発する際にファイルから一括データを送信する必要がある場合は、**TCP_CORK** を **sendfile()** とともに使用することを考慮してください。

論理パケットがさまざまなコンポーネントによってカーネルに組み込まれている場合は、**setsockopt** ソケット API を使用して値 1 に設定することにより、**TCP_CORK** を有効にします。これはソケットのコーキングとして知られています。**TCP_CORK** は、コルクが適切なタイミングで取り外されていないと、バグを引き起こす可能性があります。

次の例は、**setsockopt** ソケット API を介して **TCP_CORK** を有効にする方法を示しています。

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

一部の環境では、カーネルがコルクをいつ取り除くかを識別できない場合は、次のように手動でコルクを取り除くことができます。

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

関連情報

- **sendfile(2)** man ページ

9.3. ソケットオプションを使用したプログラム例

TCP_NODELAY および **TCP_CORK** ソケットオプションは、ネットワーク接続の動作に大きく影響します。**TCP_NODELAY** は、準備ができたらすぐにデータパケットを送信することで利益を得るアプリケーションで、Nagle のアルゴリズムを無効にします。**TCP_CORK** を使用すると、複数のデータパケットを遅延なく同時に転送できます。

注記

TCP_NODELAY などのソケットオプションを有効にするには、次のコードを使用してビルドし、適切なオプションを設定します。

```
gcc tcp_nodelay_client.c -o tcp_nodelay_client -lrt
```

tcp_nodelay_server および **tcp_nodelay_client** プログラムを引数なしで実行すると、クライアントはデフォルトのソケットオプションを使用します。**tcp_nodelay_server** および **tcp_nodelay_client** プログラムの詳細は、TCP changes result in latency performance when small buffers are used の記事を参照してください。

サンプルプログラムは、これらのソケットオプションが与える可能性のあるアプリケーションへのパフォーマンスの影響に関する情報を提供します。

クライアントへのパフォーマンスの影響

TCP_NODELAY および **TCP_CORK** ソケットオプションを使用せずに、クライアントで小さなバッファー書き込みを送信できます。引数なしで実行すると、クライアントはデフォルトのソケットオプションを使用します。

- データ転送を開始するには、サーバーの TCP ポートとサーバーが処理する必要のあるパケット数を定義します。たとえば、このテストでは 10,000 パケットです。

```
$ ./tcp_nodelay_server 5001 10000
```

このコードは、それぞれ 2 バイトのパケットを 15 個送信し、サーバーからの応答を待ちます。ここではデフォルトの TCP 動作を採用します

ループバックインターフェイスのパフォーマンスへの影響

ソケットオプションを有効にするには、**gcc tcp_nodelay_client.c -o tcp_nodelay_client -lrt** を使用してビルドし、適切なオプションを設定します。

次の例では、ループバックインターフェイスを使用して、次の 3 つのバリエーションを示します。

- バッファー書き込みをすぐに送信するには、**TCP_NODELAY** で設定されたソケットに **no_delay** オプションを設定します。

```
$ ./tcp_nodelay_client localhost 5001 10000 no_delay

 10000 packets of 30 bytes sent in 1649.771240 ms: 181.843399 bytes/ms using TCP_NODELAY
```

TCP はすぐにバッファーを送信し、小さなパケットを組み合わせるアルゴリズムを無効にします。これによりパフォーマンスは向上しますが、論理パケットごとに大量の小さなパケットが送信される可能性があります。

- 複数のデータパケットを収集し、1 回のシステムコールで送信するには、**TCP_CORK** ソケットオプションを設定します。

```
$ ./tcp_nodelay_client localhost 5001 10000 cork

 10000 packets of 30 bytes sent in 850.796448 ms: 352.610779 bytes/ms using TCP_CORK
```

コルク技術を使用すると、バッファー内の完全な論理パケットを組み合わせて送信するネットワークパケット全体が少なくなるため、データパケットの送信に必要な時間が大幅に短縮されます。適切なタイミングで **cork** を削除する必要があります。

プログラムを開発する際にファイルから一括データを送信する必要がある場合は、**TCP_CORK** を **sendfile()** オプションとともに使用することを考慮してください。

- ソケットオプションを使用せずにパフォーマンスを測定します。

```
$ ./tcp_nodelay_client localhost 5001 10000

 10000 packets of 30 bytes sent in 400129.781250 ms: 0.749757 bytes/ms
```

これは、TCP がバッファー書き込みを組み合わせて、ネットワークパケットに最適に収まるよりも多くのデータをチェックするのを待つ場合のベースライン測定値です。

関連情報

- **sendfile(2)** man ページ

第10章 RHEL FOR REAL TIME スケジューラー

RHEL for Real Time は、コマンドラインユーティリティーを使用して、プロセス設定の設定や監視を行うことができます。

10.1. スケジューラーを設定するための 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現在のプロパティーを変更できます。

chrt ユーティリティーは、**--pid** または **-p** オプションのいずれかを使用して、プロセス ID (PID) を指定します。

chrt ユーティリティーは、次のポリシーオプションを取ります。

- **-f** または **--fifo**: スケジュールを **SCHED_FIFO** に設定します。
- **-o** または **--other**: スケジュールを **SCHED_OTHER** に設定します。
- **-r** または **--rr**: スケジュールを **SCHED_RR** に設定します。
- **-d** または **--deadline**: スケジュールを **SCHED_DEADLINE** に設定します。

次の例は、指定されたプロセスの属性を示しています。

```
# chrt -p 468
pid 468’s current scheduling policy: SCHED_FIFO
pid 468’s current scheduling priority: 85
```

10.2. プリエンプティブスケジューリング

リアルタイムのプリエンプションは、実行中のタスクを一時的に中断して、後で再開することを目的としたメカニズムです。これは、優先度の高いプロセスが CPU の使用を中断したときに発生します。プリエンプションは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、プロセスは常にプリエンプティブされ、プロセスを完全に実行でき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。

/proc/PID/status ファイルの内容を表示することにより、単一のプロセスで発生する自発的および非自発的なプリエンプションを確認できます。ここで、PID はプロセス ID です。

次の例は、PID 1000 のプロセスのプリエンプションステータスを示しています。

```
# grep voluntary /proc/1000/status
voluntary_ctxt_switches: 194529
nonvoluntary_ctxt_switches: 195338
```

10.3. スケジューラー優先度のライブラリー機能

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。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。シンボル **SCHED_OTHER**、**SCHED_RR**、および **SCHED_FIFO** も、**sched.h** ヘッダーファイルで定義する必要があります。

この表では、リアルタイムスケジューラーのポリシーおよび優先度を設定する関数を示します。

表10.1 リアルタイムスケジューラー用のライブラリー関数

関数	説明
sched_getscheduler()	特定のプロセス識別子 (PID) のスケジューラーポリシーを取得します。
sched_setscheduler()	スケジューラーポリシーおよびその他のパラメーターを設定します。この関数には、3 つのパラメーター (**sched_setscheduler(pid_t pid、int policy、const struct sched_param *sp)**) が必要です。
sched_getparam()	スケジューリングポリシーのスケジューリングパラメーターを取得します。
sched_setparam()	すでに設定されており、**sched_getparam()** 関数を使用して検証できるスケジューリングポリシーに関連付けられたパラメーターを設定します。
sched_get_priority_max()	スケジューリングポリシーに関連付けられている有効な最大優先度を返します。
sched_get_priority_min()	スケジューリングポリシーに関連付けられている有効な最小優先度を返します。
sched_rr_get_interval()	プロセスごとに割り当てられた **timeslice** を表示します。

第11章 RHEL FOR REAL TIME のシステムコール

リアルタイムシステムコールは、アプリケーションプログラムがカーネルと通信するために使用する関数です。これは、プログラムがカーネルからリソースを注文するためのメカニズムです。

11.1. SCHED_YIELD() 関数

sched_yield() 関数は、実行中のプロセス以外のプロセスをプロセッサーが選択できるように設計されています。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。

この **sched_yield()** 関数がリアルタイム優先度のプロセス内で使用されると、予期しない動作が表示される可能性があります。**sched_yield()** を呼び出すプロセスは、同じ優先度で実行しているプロセスのキューの末尾に移動します。同じ優先度で実行されている他のプロセスがない場合は、**sched_yield()** を呼び出したプロセスは引き続き実行されます。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。

一般的には、リアルタイムプロセスでは **sched_yield()** を使用しないでください。

11.2. GETRUSAGE() 関数

getrusage() 関数は、指定されたプロセスまたはそのスレッドから重要な情報を取得します。次のような情報について報告します。

- 自発的および非自発的なコンテキストスイッチの数。
- 主なページ障害およびマイナーなページ障害。
- 使用中のメモリーサイズ。

getrusage() を使用すると、アプリケーションにクエリーを実行して、パフォーマンスの調整とデバッグの両方のアクティビティーに関連する情報を提供できます。**getrusage()** は、**/proc/** ディレクトリー内のいくつかの異なるファイルからカタログ化する必要があり、アプリケーションの特定のアクションまたはイベントと同期するのが難しい情報を取得します。

注記

getrusage() の結果で満たされた構造に含まれるすべてのフィールドがカーネルによって設定されるわけではありません。一部は、互換性の理由でのみ保持されます。

関連情報

- **getrusage(2)** man ページ

第12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。

12.1. 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。

12.2. 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。

12.3. 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。