Red Hat Enterprise Linux for Real Time 7

リファレンスガイド

RHEL for Real Time を使用するための中核となる概念および用語

Last Updated: 2020-09-23

Red Hat Enterprise Linux for Real Time 7 リファレンスガイド

RHEL for Real Time を使用するための中核となる概念および用語

Jaroslav Klech
Red Hat Customer Content Services
jklech@redhat.com

Marie Doleželová
Red Hat Customer Content Services

Maxim Svistunov
Red Hat Customer Content Services

Radek Bíba
Red Hat Customer Content Services

David Ryan
Red Hat Customer Content Services

Cheryn Tan
Red Hat Customer Content Services

Lana Brindley
Red Hat Customer Content Services

Alison Young
Red Hat Customer Content Services

法律上の通知

概要

本書は、ユーザーおよび管理者が、Red Hat Enterprise Linux for Real Time を適切に使用するために必要なさまざまな用語や概念を理解するのに役立ちます。インストール手順は、Red Hat Enterprise Linux for Real Time Installation Guide を参照してください。チューニングの詳細は、Red Hat Enterprise Linux for Real Time Tuning Guide を参照してください。

目次

前書き

本ガイドでは、Red Hat Enterprise Linux for Real Time の使用について説明します。

多くの業界や組織には、非常に高いパフォーマンスコンピューティングが必要で、特に業界や通信業界で低かつ予測可能なレイテンシーが必要になる場合があります。レイテンシー (応答時間) は、イベントとシステム応答間の時間として定義され、通常マイクロ秒(μ)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なるだけでなく、予測可能な機能も必要とする業界では、Red Hat は、これを提供する「ドロップイン」カーネル置き換えを開発しました。Red Hat Enterprise Linux for Real Time は、Red Hat Enterprise Linux 7 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。

パート I. ハードウェア

適切なハードウェアを選択して設定することが、リアルタイム環境の設定に不可欠です。ハードウェアは、システムが動作する方法に影響します。システム管理割り込み、CPU キャッシュ設計、および NUMA 使用率はすべてさまざまな方法で処理できます。ハードウェアはベンダーごとに異なる可能性があり、すべてのハードウェアがリアルタイム環境に適しているとは限りません。

Red Hat Enterprise Linux for Real Time 環境をセットアップする場合に、アプリケーションが利用可能なハードウェアと適切に対話するように設計されていることが重要です。本セクションでは、Red Hat Enterprise Linux for Real Time がハードウェアを使用する方法と、検索するエリアを説明します。

第1章 プロセッサーコア

*プロセッサーコア*は、コンピューター内の物理処理ユニット (CPU) です。コアはマシンコードの実行を行います。*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。1つのコアプロセッサーが物理的に1つのソケットを占有しており、1つのコアを利用できます。クアッドコアプロセッサーは物理的に1つのソケットを占有しており、4つのコアを利用できます。

リアルタイムアプリケーションを設計する場合は、利用可能なコアの数を考慮してください。また、コア間でキャッシュがどのように共有されるか、およびコアが物理的に接続されている方法に注意することが重要です。

アプリケーションで複数のコアが利用できる場合は、スレッドまたはプロセスを使用してその利点を活用します。このコンストラクトを使用せずにプログラムが作成された場合には、一度に1つのプロセッサーでのみ実行されます。マルチコアプラットフォームを使用すると、異なるタイプの操作に異なるコアを使用して、利点を得ることができます。

1.1. キャッシュ

多くの場合、アプリケーションのさまざまなスレッドは、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。この場合、システムのキャッシュレイアウトを把握することでパフォーマンスを向上できます。Tuna ツールを使用すると、キャッシュレイアウトの判断に役立ちます。対話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ようにします。*キャッシュ共有*は、相互除外プリミティブ (mutex、condvar、または同様の)とデータ構造自体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。

1.2. 相互接続

コア間の相互接続を確認することが重要です。マシンのコア数が増えると、すべてのメモリーへのアクセスが統合されるため、複雑化して負荷が大きくなります。多くのハードウェアベンダーは、*NUMA* (非汎用メモリーアクセス) 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 システムでは、相互接続トポロジーを把握することで、頻繁に通信するスレッドを隣接コアに配置することができます。

第2章 メモリーの割り当て

Linux ベースのオペレーティングシステムは、仮想メモリーシステムを使用します。ユーザー空間アプリケーションによって参照されるアドレスは、物理アドレスに変換する必要があります。これは、基盤となるコンピューター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り実行されます。

プログラムと実際のメモリー間の翻訳メカニズムを持つことの 1 つの結果として、オペレーティングシステムが必要なときにページを奪うことができることがあります。これは、以前使用されていたページテーブルエントリーを無効なとマークすることで実現されます。そのため、通常のメモリー不足でも、オペレーティングシステムがあるアプリケーションからページをスケートして別のアプリケーションに渡す可能性があります。これは、決定論的な動作を必要とするシステムに悪影響を与える可能性があります。ページフォールトがトリガーされるため、通常、固定時間で実行される命令は通常よりも時間がかかる可能性があります。

2.1. 需要ページング

Linux では、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。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。

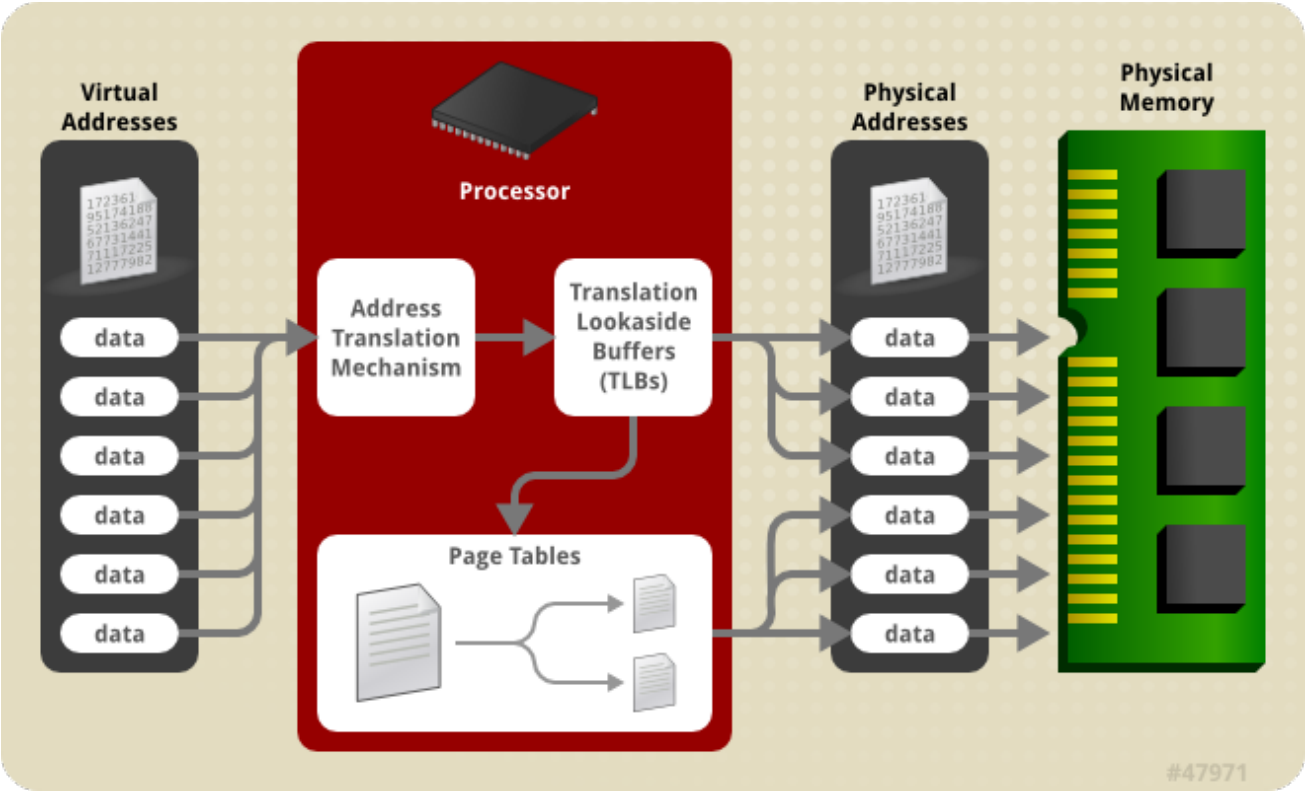


図2.1 Red Hat Enterprise Linux for Real Time 仮想メモリーシステム

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。TLB は、ページテーブル変換のキャッシュです。

ページテーブルエントリーに物理アドレスが割り当てられている場合は、*常駐の作業セット* 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メモリーを解放する必要がある場合、作業セットからページを削除できます。これが発生すると、そのページ内の仮想アドレスへの参照によりページフォー

ルトが作成され、ページが再割り当てされます。システムの物理メモリーが非常に少ない場合、このプロセスは *thrash* を開始し、そのプロセスから常にページを奪うようになり、プロセスを完了できなくなります。仮想メモリーの統計は、**/proc/vmstat** ファイルの ***pgfault*** 値を検索することで監視できます。

TLB 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 *参照のローカリティーの* 原則で動作します。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。再帰を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取得を回避します。

メモリーレイテンシーの潜在的なソースは、マイナーページフォールトと *呼ばれます*。それらは初期化される前に、プロセスがメモリーの一部にアクセスしようとすると作成されます。この場合、システムは、メモリーマップやその他の管理構造を埋める操作をいくつか実行する必要があります。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。

より深刻なメモリーレイテンシーは、*主要なページ障害* です。これは、システムがディスクとメモリーバッファーを同期したり、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、その他の入出力アクティビティーを取得してメモリーを解放する必要がある場合に発生する可能性があります。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と戻りを指示します。これにより、すべてレイテンシーが大幅に向上します。

マルチスレッドアプリケーションを作成する場合、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 ノードが含まれます。データ情報を異なるキャッシュや NUMA ドメインの CPU で非常に密接に共有すると、トラフィックの問題やボトルネックが生じる可能性があります。

競合により、パフォーマンスに深刻な問題が生じる可能性があります。ハードウェアによっては、さまざまなメモリーバスのトラフィックはいかなる規則にも適用されません。これを回避するために、必ず使用しているハードウェアを確認してください。

メモリー割り当てエラーは、CPU のアフィニティー、スケジューリングポリシー、および優先順位を使用して常に除外することはできません。アプリケーションがパフォーマンス低下を表示する場合は、ページフォールトによる影響を受けているかどうかを確認する利点があります。これには複数の方法がありますが、簡単な方法は **/proc** ディレクトリー内のプロセス情報を確認することです。特定のプロセス PID については、**cat** コマンドを使用して **/proc/*PID*/stat** ファイルを表示します。このファイル内の関連するエントリーは以下のとおりです。

- ***Field 2***: 実行ファイルのファイル名
- ***Field 10***: マイナーページフォールトの数
- ***Field 12***: メジャーページフォールトの数

プロセスがページに遭遇すると、カーネルが障害を処理するまですべてのスレッドがフリーズします。この問題を解決する方法はいくつかありますが、ページ障害を回避するためにソースコードを調整することが最善の方法になります。

例2.1 /proc/PID/stat ファイルを使用したページ障害の確認

この例では、**/proc/*PID*/stat** ファイルを使用して、実行中のプロセスでページ障害の有無を確認します。

cat コマンドとパイプ関数を使用して、**/proc/*PID*/stat** ファイルの 2 行目、10 行目、および 12 行のみを返します。

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

上記の出力では、PID 3366 が **bash** で、マイナーページフォールトが 5389 件報告され、主要なページフォールトはありません。

注記

詳細情報や参照文書は、以下の文書が、このセクションの情報に関連しています。

- 『Linux System Programming』 by Robert Love

2.2. ページ I/O を避けるために MLOCK を使用

mlock および **mlockall** システムコールは、指定したメモリー範囲にロックし、そのメモリーをページングできないように指示します。つまり、物理ページがページテーブルエントリーに割り当てられると、そのページへの参照は常に高速になります。

mlock システムコールには、2 つのグループがあります。**mlock** および **munlock** は、特定のアドレス範囲のロックおよびロック解除を行います。**mlockall** および **munlockall** は、プログラム領域全体をロックまたはアンロックします。

mlock 慎重に検討し、注意して使用してください。アプリケーションが大きい場合や、大規模なデータドメインがある場合は、システムが他のタスクにメモリーを割り当てできない場合に **mlock** 呼び出しがスラッシュする可能性があります。

注記

mlock は常に注意して使用してください。これを過剰に使用すると、メモリー不足 (OOM) エラーが生じる可能性があります。アプリケーションの先頭には **mlockall** 呼び出しを付けないでください。アプリケーションのリアルタイム部分のデータとテキストのみがロックされることが推奨されます。

mlock は、このプログラムにページ I/O がないことを保証しません。これは、データがメモリー内に留まるが、同じページに留まることを確認するのに使用されます。**move_pages** やメモリー圧縮関数は、**mlock** を使用してもデータを移動できます。

重要

非特権ユーザーは、大きなバッファーで **mlockall** または **mlock** を使用できるようにするために、**CAP_IPC_LOCK** 機能が必要です。詳細は capabilities(7) man ページを参照してください。

さらに、メモリーロックはページベースで作成され、スタックされないことを通知することが推奨されます。つまり、2 つの動的に割り当てられたメモリーセグメントが、**mlock** または **mlockall** への呼び

出し 2 回ロックされた同じページを共有する場合、一致するページの **munlock** への単一の呼び出し、または **munlockall** によってアンロックされます。そのため、この二重ロック/シングルロックの問題を防ぐために、アプリケーションがロック解除しているページを認識する必要があります。

double-lock/single-unlock の問題を軽減する最も一般的な 2 つの方法は次のとおりです。

- 割り当て済みおよびロックされたメモリー領域を追跡し、ページをロックする前にラッパー機能を作成すると、そのページにあるユーザー数 (割り当て) を確認します。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。
- 同じページで二重ロックを防ぐために、ページサイズとアライメントを考慮して割り当てを実行します。

以下のコード例は、2 番目の代替を示しています。

mlock の最適な利用方法は、アプリケーションのニーズとシステムリソースによって異なります。すべてのアプリケーションには単一のソリューションはありませんが、以下のコード例は、メモリーバッファーを割り当て、ロックする関数の実装のスタートポイントとして使用できます。

例2.2 アプリケーションでの mlock の使用

```
#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);
}
```

この関数 **alloc_workbuf** 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。メモリーの割り当ては、メモリー領域をページに合わせるために **posix_memalig** によって行われます。**size** 変数が小さい場合は、ページサイズよりも小さいと、通常の **malloc** 割り当てで残りのページを使用できます。ただし、この手法を安全に使用するために、通常の **malloc** 割り当てでは **mlock** 呼び出しを行うことができません。これにより、二重ロック/シングルアンロックの問題が回避されます。この関数 **free_workbuf** は、メモリー領域のロックを解除し、解放します。

mlock および **mlockall** の使用方法に加えて、**MAP_LOCKED** フラグで **mmap** を使用してメモリー領域の割り当て、ロックすることもできます。以下の例は、**mmap** を使用した前述のコードの実装です。

例2.3 アプリケーションでの mmap の使用

```
#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);
}
```

メモリー **mmap** をページベースで割り当てると、同じページにロックが 2 つ存在せず、二重ロック/シングルアンロックの問題を防ぐのに役立ちます。一方、**size** 変数がページサイズの倍数ではない場合は、残りのページが無駄になります。さらに、**mmap** によってロックされたメモリーの **munlockall** ロックを解除するための呼び出しが必要になります。

フットプリントが小さいアプリケーションのもう 1 つは、コードの時間機密領域を入力する前に、**mlockall** を呼び出し、その後に時間機密領域の最後に **munlockall** を呼び出します。これにより、重要なセクションにおいてページングを減らすことができます。同様に、**mlock** は比較的静的または、ページ I/O なしでアクセスが必要な徐々に増大するデータリージョンで使用できます。

注記

詳細は、以下の man ページは本セクションに記載の情報に関連しています。

- capabilities(7)
- mlock(2)
- mlock(3)
- mlockall(2)
- mmap(2)
- move_pages(2)
- posix_memalign(3)
- posix_memalign(3p)

第3章 ハードウェア割り込み

ハードウェア割り込み は、オペレーティングシステムから注意する必要があることを示すデバイスにより使用されます。一般的な例は、一連のデータブロックを読み取り、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ているハードディスクシグナリングです。また、割り込みは、外部ネットワークからの新規データの取得など、非同期イベントにも使用されます。ハードウェア割り込みは、割り込み管理およびルーティングデバイスの小規模ネットワークを使用して CPU に直接送信されます。本章では、さまざまなタイプの割り込みと、ハードウェアおよびオペレーティングシステムによる処理方法について説明します。また、割り込みのタイプを処理するために、Red Hat Enterprise Linux for Real Time カーネルが標準のカーネルとどのように異なるかについても説明しています。

標準システムは、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」割り込みを含む、運用期間中に多くの割り込みを受け取ります。NMI (Non- maskable Interrupts) や SMI (System Management Interrupts) などの特別な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 *割り込み番号* によって参照されます。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。

多くのコンピューターシステムでは、割り込みは、できるだけ迅速に処理されます。割り込みを受信すると、現在のアクティビティーが停止し、*割り込みハンドラー* が実行されます。ハンドラーは、その他の実行中のプログラムやシステムアクティビティーを阻止します。これにより、システム全体が遅くなり、レイテンシーが作成されます。Red Hat Enterprise Linux for Real Time は、パフォーマンスを改善するために割り込みの処理方法を変更し、レイテンシーを短縮します。

例3.1 システムにおける割り込みの表示

Linux システムで受信したハードウェア割り込みのタイプと数を確認するには、**cat** コマンドで **/proc/interrupts** を表示します。

```
~]$ cat /proc/interrupts
  CPU0       CPU1
0:   13072311          0   IO-APIC-edge      timer
1:      18351          0   IO-APIC-edge      i8042
8:        190          0   IO-APIC-edge      rtc0
9:     118508       5415   IO-APIC-fasteoi   acpi
12:    747529      86120   IO-APIC-edge      i8042
14:   1163648          0   IO-APIC-edge      ata_piix
15:         0          0   IO-APIC-edge      ata_piix
16:  12681226     126932   IO-APIC-fasteoi   ahci, uhci_hcd:usb2, radeon, yenta, eth0
17:   3717841          0   IO-APIC-fasteoi   uhci_hcd:usb3, HDA, iwl3945
18:         0          0   IO-APIC-fasteoi   uhci_hcd:usb4
19:       577         68   IO-APIC-fasteoi   ehci_hcd:usb1, uhci_hcd:usb5
NMI:        0          0   Non-maskable interrupts
LOC:  3755270    9388684   Local timer interrupts
RES:  1184857    2497600   Rescheduling interrupts
CAL:    12471       2914   function call interrupts
TLB:    14555      15567   TLB shootdowns
TRM:        0          0   Thermal event interrupts
SPU:        0          0   Spurious interrupts
ERR:        0
MIS:        0
```

この出力には、さまざまなタイプのハードウェア割り込み、受信した CPU の数、割り込みのターゲットであった CPU、および割り込みを生成したデバイスが表示されます。

3.1. レベルシグナル割り込み

*レベルシグナル割り込み*は、専用割り込みラインを使用して電圧移行を提供します。

専用ラインは、バイナリー 1 または バイナリー 0 を示す 2 つの 電圧のいずれかを送信できます。このラインからシグナルが送信されると、CPU が特別にリセットされるまで、その状態のままになります。これは、CPU がラインの *アサート* を停止するよう生成デバイスを要求することで実行されます。これにより、複数のデバイスが 1 つの割り込み行を共有できます。CPU がラインのアサートを停止するようデバイスに指示し、かつアサートされたままの場合は、保留中の別の割り込みがあります。

レベルシグナル割り込みでは、デバイスと CPU の両方で高度なハードウェアロジックが必要になりますが、数多くの利点も提供されます。複数のデバイスで使用できるだけでなく、ほぼ完全に割り込みをミスすることができません。

3.2. メッセージシグナル割り込み

最新のシステムの多くは、*メッセージシグナル割り込み* 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ジャンプバスにシグナルを専用メッセージとして送信します。

このタイプのバスの一般的な例として、PCI Express (Peripheral Component Interconnect 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タイプとしてメッセージを送信します。ホストコントローラーはメッセージを CPU に送信します。

ハードウェアによっては、PCIe システムは PCIe ホストコントローラーと CPU 間の専用割り込みラインを使用してシグナルを送信するか、たとえば CPU HyperTransport バス 越しにメッセージを送信して、シグナルを送信することがあります。多くの 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。従来の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

割り込みは、無効になった場合や、CPU が無視するよう指示されたときに *マスクされる*と考えられます。*マスク不可割り込み* (NMI) は無視できず、通常は重要なハードウェアエラーにのみ使用されます。

NMI は通常、別の割り込み行で配信されます。CPU が NMI を受信すると、重大なエラーが発生し、システムがクラッシュする可能性が分かります。NMI は通常、問題の原因となった可能性があることを最もよく示しています。

NMI は無視できないため、一部のシステムではハードウェアモニターとして使用されます。デバイスは NMI のストリームを送信します。これはプロセッサーの NMI ハンドラーによってチェックされます。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。

3.4. システム管理割り込み

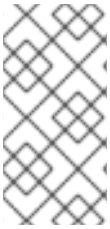
システム管理割り込み (SMI) は、レガシーハードウェアデバイスエミュレーションなどの拡張機能を提供するために使用されます。システム管理タスクに使用することもできます。SMI は NMI と似てお

り、一般的には特殊文字のシグナリングラインを CPU に直接使用し、マスクできない点が挙げられます。

SMI を受信すると、CPU は *System Management Mode* (SMM)に入ります。このモードでは、SMI の処理に非常に低レベルのハンドラールーチンが実行されます。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。

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のに最もよく使用されます。一般的な例としては、ディスクドライブをエミュレートする方法が挙げられます。システムにディスクが接続されていない場合は、代わりに仮想化ネットワーク管理エミュレーションを使用できます。オペレーティングシステムがディスクへのアクセスを試みると、SMI がトリガーされ、ハンドラーは代わりにエミュレートされたデバイスでオペレーティングシステムを提供します。その後、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自体のように処理します。

Red Hat Enterprise Linux for Real Time は、オペレーティングシステムを直接起動せずに行われるため、SMI の悪影響を受ける可能性があります。適切に記述されていない SMI 処理ルーチンは、CPU 時間をミリ秒単位で消費し、必要な場合は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くなります。このような状況では、調整され、高度に応答するシステムで、定期的なレイテンシーが発生します。ただし、ベンダーは SMI ハンドラーを使用して CPU 温度およびステンシーを制御することができるため、これらを無効にすることはできません。代わりに、問題のベンダーに通知することが推奨されます。



注記

Red Hat Enterprise Linux for Real Time システムで SMI を分離するには、**rt-tests** パッケージで利用可能な **hwlatdetect** ユーティリティーを使用します。このユーティリティーは、CPU が SMI 処理ルーチンによって引き起こされた期間を測定するように設計されています。

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

プログラム可能な高度な割り込みコントローラー(APIC) は、大量の割り込みを処理する機能を提供して、これらを特定の利用可能な CPU セットにプログラムでルーティング (およびこれに応じて変更) し、CPU 間の通信をサポートするために Intel® によって開発されました。また、1 つの割り込みラインを共有する多数のデバイスが不要になりました。

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。ハードウェアデバイスが割り込みを生成すると、それが接続されている IO-APIC によって検出され、システム APIC バス全体で特定の CPU にルーティングされます。オペレーティングシステムは情報ソースの組み合わせにより、どの IO-APIC がどのデバイスに接続しているか、およびそのデバイス内の特定のどの割り込みラインに接続しているかを認識します。まず、ACPI DSDT (Advanced Configuration and Power Interface Differentiated System Description Table) があります。これには、ホストシステムのマザーボードおよび境界コンポーネントの特定の接続に関する情報が含まれます。次に、デバイスは利用可能な割り込みソースに関する特定の情報を提供します。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。

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 ベースの割り込みが可能です。

パート II. アプリケーションのアーキテクチャー

Red Hat Enterprise Linux for Real Time カーネルは、ソフトウェア開発者が、可能な限り高い基準で動作するアプリケーションをビルドできるように設計されたコンストラクトを多数提供します。本項では、これらの機能とそれらの使用方法について説明します。

このセクションと次のセクションでは、Red Hat Enterprise Linux for Real Time カーネルを直接調整する手順を説明します。ほとんどの変更は、*Tuna* と呼ばれるツールを使用して実行することもできます。グラフィカルインターフェースがあるか、コマンドシェルから実行できます。

Tuna を使用すると、スケジューリングポリシー、スケジューラーの優先度、プロセッサーアフィニティーなどのスレッドおよび割り込みの属性を変更できます。このツールは、実行中のシステムでの使用を目的として設計されており、直ちに変更が行われます。これにより、アプリケーション固有の測定ツールは、変更の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。

第4章 スレッドおよびプロセス

すべてのプログラムはスレッドとプロセスを使用しますが、Red Hat Enterprise Linux for Real Time は標準の Red Hat Enterprise Linux とは別の方法で処理します。本章では、スレッドおよびプロセスに対する Red Hat Enterprise Linux for Real Time のアプローチを説明します。

各 CPU コアは、処理可能な作業量に制限されます。効率性を向上させるために、アプリケーションは複数のコアで異なるタスクを同時に実行できます。これは*並列化*と呼ばれます。

プログラムは *スレッド* を使用して並列化できます。ただし、スレッドとプロセスは混同されることが多く、用語の違いを理解することが重要です。

プロセス

UNIX® スタイルのプロセスとは、以下を含むオペレーティングシステムのコンストラクトです。

1. 仮想メモリーのアドレスマッピング
2. 実行コンテキスト (PC、スタック、レジスター)
3. 状態/アカウント情報

Linux プロセスは、このスタイルのプロセスとして開始しました。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ていました。これは、プロセスデータ構造が小さい限り機能します。本書の残りの部分では、*プロセス* という用語は、複数のスレッドが含まれる可能性がある独立したアドレス空間を指します。

スレッド

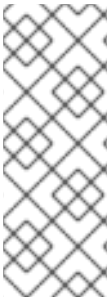
厳密には、スレッドは以下が含まれるスケジュール可能なエンティティーです。

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

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。

Red Hat Enterprise Linux for Real Time システムでプログラミングする場合は、プログラムを並列化する可能性がある方法が 2 つあります。

1. **fork** および **exec** 関数を使用した新規プロセスの作成
2. Posix Threads (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。



注記

コンポーネントの並列処理方法を決定する前に、コンポーネントがどのように対話するかを評価します。コンポーネントが相互に独立していて、十分に対話しない場合や、新しいアドレス空間を作成し、新規プロセスとして実行される場合は、通常は適切なオプションになります。ただし、コンポーネントがデータを共有したり、頻繁に通信する必要がある場合は、1 つのアドレス空間内のスレッドとして実行すると、通常はより効率的になります。

注記

詳細は、以下の man ページと書籍は本セクションに記載の情報に関連しています。

- fork(2)
- exec(2)
- 『Programming with POSIX Threads』, David R. Butenhof, Addison-Wesley, ISBN 0-201-63392-2
- 『Advanced Programming in the UNIX Environment』, 2nd Ed., W. Richard Stevens and Stephen A. Rago, Addison-Wesley, ISBN 0-201-43307-9
- 「POSIX Threads Programming」, Blaise Barney, Lawrence Livermore National Laboratory, http://www.llnl.gov/computing/tutorials/pthreads/

第5章 優先順位およびポリシー

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**: デフォルトポリシー
- **SCHED_BATCH**: **SCHED_OTHER** と同様ですが、スループットが維持されます。
- **SCHED_IDLE**: **SCHED_OTHER** より低い優先度
- **SCHED_FIFO**: 最初にリアルタイムポリシーを出力
- **SCHED_RR**: ラウンドロビンのリアルタイムポリシー

Red Hat Enterprise Linux for Real Time に重要なポリシーは **SCHED_OTHER SCHED_FIFO**および **SCHED_RR** です。

SCHED_OTHER または **SCHED_NORMAL** 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** スレッドの優先度に悪影響を与えるもう 1 つのものは、*nice 値*です。nice 値は、-20 (最も高い優先度) と 19 (最も低い優先度) の数値です。デフォルトでは、**SCHED_OTHER** スレッドの適切な値は 0 になります。nice 値を調整すると、スレッドの処理方法が変わります。

SCHED_FIFO ポリシーのあるスレッドは、**SCHED_OTHER** タスクよりも先に実行されます。適切な値を **SCHED_FIFO** 使用する代わりに、1 (最低) と 99 (最大) の優先度を使用します。優先度が 1 の **SCHED_FIFO** スレッドは常に **SCHED_OTHER** スレッドよりも先にスケジュールされます。

この **SCHED_RR** ポリシーは、**SCHED_FIFO** ポリシーと非常に似ています。**SCHED_RR** ポリシーでは、優先度が等しいスレッドは *ラウンドロビン方式* でスケジュールされます。通常、**SCHED_FIFO** は、**SCHED_RR** よりも優先されます。

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、またはイベントの待機を開始します。
- 優先度の高いリアルタイムスレッドが実行できるようになります

これらのイベントのいずれかが発生すると、スレッドはそのプロセッサー上で無期限に実行され、優先順位の低いスレッドを実行できなくなります。これにより、システムサービススレッドの実行に失敗し、メモリースワップやファイルシステムデータのフラッシュが期待どおりに行われない可能性があります。

第6章 アフィニティー

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。

プロセッサーアフィニティーを、有効なポリシーおよび優先度設定とともに設定すると、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、他のプロセスと、特に CPU 時間などのリソースに対して常に競合する必要があります。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する可能性があります。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。Red Hat Enterprise Linux for Real Time には、これらの問題の一部に対応し、レイテンシーをより適切に制御でき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、その役割の 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とコンシューマースレッド。

予想される動作と一致させるには、アフィニティーの設定をプログラムと併用することを推奨します。

リアルタイムシステムでアフィニティーを調整する通常のプラクティスは、アプリケーションを実行するために必要なコア数を判断し、それらのコアを分離することです。これは、**Tuna** ツールを使用して実行することも、シェルスクリプトを使用してビットマスクの値を変更することもできます。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ますが、**/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。

注記

詳細情報や参照文書は、taskset(1) man ページが、このセクションの情報に関連しています。

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

taskset コマンドは、特定のプロセスのアフィニティー情報を設定し、確認します。これらのタスクは、Tuna ツールを使用して実行することもできます。

-p または **--pid** オプションと、チェックするプロセスの PID を指定して taskset コマンドを使用します。**-c** または **--cpu-list** オプションは、情報をビットマスクではなく、コアの数値リストとして表示します。

以下のコマンドは、PID 1000 のプロセスのアフィニティーをチェックします。この場合、PID 1000 は CPU 0 または CPU 1 のいずれかの使用が許可されます。

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

アフィニティーを設定するには、プロセスのバインド先の CPU の数を指定します。この例では、PID 1000 を CPU 0 または CPU 1 のいずれかで実行でき、アフィニティーは変更され、CPU 1 でのみ実行できるように変更されています。

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

複数の CPU アフィニティーを定義するには、コンマで区切って両方の CPU 番号を一覧表示します。

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

この **taskset** コマンドは、特定のアフィニティーで新規プロセスを開始するためにも使用できます。このコマンドは、CPU 4 で **/bin/my-app** アプリケーションを実行します。

```
~]# taskset -c 4 /bin/my-app
```

さらに細かく設定するために、優先順位とポリシーも設定できます。このコマンドは、**SCHED_FIFO** ポリシーと 78 の優先度で、CPU 4 で **/bin/my-app** アプリケーションを実行します。

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

6.2. 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

taskset コマンドの他に、**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。

以下のコードの抜粋は、指定した PID の CPU アフィニティー情報を取得します。これに渡される PID が 0 の場合は、現在のプロセスのアフィニティー情報を返します。

```
#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);
}
```

注記

詳細は、以下の man ページは本セクションに記載の情報に関連しています。

- sched_setaffinity(2)

第7章 スレッドの同期

共有リソースへのアクセスが必要なスレッドは、*スレッド同期* を使用して調整されます。Linux で使用される 3 つのスレッド同期メカニズムは *mutexes*、*barriers*、*condvars* です。

7.1. ミューテックス

この **mutex** は *相互除外 (mutual-exclusion)* という用語から派生しています。mutex は POSIX スレッド構成で、**pthread_create_mutex** ライブラリー呼び出しを使用して作成されます。mutex は、コードの各セクションへのアクセスをシリアライズし、アプリケーションのスレッドが同時にコードを実行するようにします。

mutex と同様に **futex** があります。これはミューテックスの実装に使用する内部メカニズムである Fast User muTEX です。futexes は、カーネルと C ライブラリーの間で共有規則を使用します。これにより、カーネル領域へのコンテキスト切り替えがないと、競合のない mutex がロックまたは解放されます。

7.2. バリア

Barriers は、他のスレッド同期方法と非常に異なる方法で動作します。コードリージョンへのアクセスをシリアライズする代わりに、事前に決定された数までのスレッドが累積されるまですべてのスレッドをブロックします。次に、バリアにより、すべてのスレッドが続行されます。バリアは、実行を続行する前にすべてのスレッドがタスクを完了していることを確認する必要がある環境で使用されます。

7.3. CONDVARS

condvar、または条件変数は、POSIX スレッドの構成で、特定の条件が達成されるのを待機してから続行します。一般的に、シグナル化している条件は、スレッドが別のスレッドと共有するデータの状態を保持します。たとえば、condvar を使用して、データエントリーが処理キューに格納されたことを通知でき、キューからデータを処理するのを待機するスレッドが続行できるようになりました。

7.4. 他のタイプの同期

POSIX スレッドが出現する前に、プロセス間でスレッドの同期が発生していました。最も一般的なメカニズムは、共有メモリー、メッセージキュー、およびセマフォの System V IPC 呼び出しです。POSIX スレッド呼び出しが推奨されるため、System V IPC コールの使用が非推奨になりました。

第8章 ソケット

ソケットは双方向のデータ転送メカニズムです。これらは、2 つのプロセス間でデータを転送するために使用されます。2 つのプロセスは、Unix-domain またはループバックソケットと同じシステム、またはネットワークソケットとして異なるシステムで実行できます。

Red Hat Enterprise Linux for Real Time システムでソケットを使用する特別なオプションや制限はありません。

8.1. ソケットオプション

Red Hat Enterprise Linux for Real Time アプリケーションに関連する 2 つのソケットオプション **TCP_NODELAY** および **TCP_CORK**があります。

TCP_NODELAY

TCP は最も一般的なトランスポートプロトコルです。つまり、多くの異なるニーズに対応するために使用されます。新しいアプリケーションおよびハードウェア機能が開発され、カーネルアーキテクチャーの最適化が行われると、TCP は変更を効果的に処理するために新しいヒューリスティックを導入する必要がありました。

このヒューリスティックにより、プログラムが不安定になる可能性があります。動作は基盤となるオペレーティングシステムのコンポーネントが変更されるため、注意して処理する必要があります。

TCP のヒューリスティックな動作の 1 つの例は、小さいバッファーが遅延することです。これにより、1 つのネットワークパケットとして送信できます。通常、これは正常に機能しますが、レイテンシーを作成することもできます。Red Hat Enterprise Linux for Real Time アプリケーションでは、**TCP_NODELAY** は、この動作をオフにするのに指定できるソケットオプションです。以下の機能を使用すると、**setsockopt** ソケット API で有効にできます。

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

このオプションを効果的に使用するには、TCP はこのバッファーを個別のパケットとして送信するため、バッファーの小さい書き込みは回避する必要があります。また、**TCP_NODELAY** はその他の最適化ヒューリスティックと対話でき、全体的なパフォーマンスが低下します。

アプリケーションに論理的に関連し、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複数ある場合は、送信前に連続したパケットを構築することで、レイテンシーとパフォーマンスが向上します。その後、パケットは **TCP_NODELAY** が有効なソケットを使用して送信できます。

メモリーバッファーが論理的に関連付けられていても、まだ連続していない場合は、それらを使用して I/O ベクトルを構築します。その後、**TCP_NODELAY** を有効にしたソケットで **writev** を使用してカーネルに渡すことができます。

TCP_CORK

同様の方法で機能する別の TCP ソケットオプションは、**TCP_CORK** です。有効にすると、アプリケーションが cork を削除されるまで TCP パケットがすべて遅延し、保存されたパケットが送信されます。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、異なるライブラリーを使用して層の抽象化を提供する場合に便利です。

以下の関数を使用すると、**TCP_CORK** オプションを有効にできます。

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

多くの場合、**TCP_CORK** の有効化は、**corking the socket** と呼ばれます。

cork の削除タイミングをカーネルが特定できない場合は、関数を使用して手動で削除できます。

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

ソケットのコードが解除されると、TCP は、アプリケーションからの追加のパケットを待たずに累積された論理パッケージを即座に送信します。

例8.1TCP_NODELAY および TCP_CORK の使用

この例では、**TCP_NODELAY** と **TCP_CORK** による、アプリケーションのパフォーマンスへの影響を示しています。

サーバーは 30 バイトのパケットを待機した後、2 バイトパケットを応答で送信します。はじめに、TCP ポートと処理するパケット数を定義します。この例では、これは 10,000 パケットです。

```
~]$ ./tcp_nodelay_server 5001 10000
```

サーバーにはソケットオプションを設定する必要はありません。

クライアントが引数を指定せずに実行すると、デフォルトのソケットオプションが使用されます。**TCP_NODELAY** ソケットオプションを有効にするには、**no_delay** オプションを指定します。**TCP_CORK** を有効にする場合は、**cork** オプションを指定します。すべてのケースで 15 パケットが送信され、それぞれ 2 バイトの送信とサーバーからの応答を待機します。

この例では、ループバックインターフェースを使用して、3 つの変動を示しています。

最初のバリアントでは、**TCP_NODELAY** も **TCP_CORK** も使用されません。これはベースラインの測定です。TCP のコアレッシングは書き込みを行い、アプリケーションがネットワークパケットに最適に適合できる以上のデータがあるかどうかを確認する必要があります。

```
~]$ ./tcp_nodelay_client localhost 5001 10000
10000 packets of 30 bytes sent in 400129.781250 ms: 0.749757 bytes/ms
```

2 つ目のバリエーションでは **TCP_NODELAY** のみを使用します。TCP は小さなパケットを結合しないように指示されますが、バッファーを即座に送信するよう指示されます。これによりパフォーマンスが大幅に改善されますが、各論理パケットに多数のネットワークパケットが作成されます。

```
~]$ ./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
```

3 つ目のバリアント **TCP_CORK** は使用します。これは、同じ数の論理パケットを送信するのに必要な時間を 2 倍にします。これは、TCP がバッファーに論理パケット全体を結合し、ネットワークパケット全体を送信するためです。

```
~]$ ./tcp_nodelay_client localhost 5001 10000 cork
10000 packets of 30 bytes sent in 850.796448 ms: 352.610779 bytes/ms using TCP_CORK
```

このシナリオでは、**TCP_CORK** が最適な手法です。これにより、アプリケーションはパケットが終了していることを正確に伝えることができ、遅延なく送信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** とともに使用することを考慮してください。

■

注記

詳細やその他文書は、以下の man ページとサンプルアプリケーションが、本セクションの情報に関連しています。

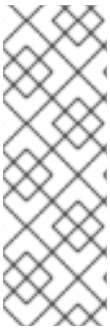
- sendfile(2)
- 「TCP nagle サンプルアプリケーション」 (C で書かれた両方のソケットオプションのアプリケーションの例)。ダウンロードするには、以下のリンクから右クリックして保存します。
  - https://github.com/acmel/libautocork/blob/master/tcp_nodelay_client.c
  - https://github.com/acmel/libautocork/blob/master/tcp_nodelay_server.c

第9章 共有メモリー

プログラムスレッドの主な利点の1つは、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することです。つまり、すべてのデータ構造にアクセスできるようになります。ただし、アプリケーションがスレッドを使用することが常に適切とは限りません。この場合、プロセスはアドレス空間の一部を共有する必要がある場合があります。これは、*共有メモリー* を使用することで、通常のカーネルとリアルタイムカーネルの両方で実現できます。

2 つのプロセス間でメモリー領域を共有する元のメカニズムは System V IPC **shmem** 呼び出しセットでした。これらの呼び出しは極めて優れていますが、ほとんどのユースケースでは非常に複雑で複雑で複雑です。このため、Red Hat Enterprise Linux for Real Time カーネルで非推奨となったため、使用すべきではありません。

Red Hat Enterprise Linux for Real Time は、**shm_open**や **mmap** などの POSIX 共有メモリー呼び出しを使用します。



注記

詳細は、以下の man ページは本セクションに記載の情報に関連しています。

- shm_open(3)
- shm_overview(7)
- mmap(2)

第10章 共有ライブラリ

動的共有オブジェクト (DSO) は、一般的に *共有ライブラリー* と呼ばれ、個別のプロセスアドレス空間間でコードを共有するために使用されます。DSO は、**ld.so** システムローダーによって 1 度読み込まれます。そこから、ライブラリーからのシンボルを必要とするプロセスのアドレス空間にマッピングされます。記号の最初の参照が検出されるまでは、評価できません。シンボルが参照されている場合にのみ、シンボルの評価はレイテンシーのソースになります。これは、メモリーページがディスク上で可能で、キャッシュを無効にすることができるためです。シンボルを事前に評価することは、レイテンシーを改善するのに役立つ安全なサイド手順です。

プログラム起動時にシンボルを解決すると、プログラムの初期化が遅くなる可能性があります。ただし、シンボルの検索によって引き起こされる可能性のあるプログラムの実行中に、決定論的でない遅延が発生しないようにします。アプリケーションの起動時のシンボル解決は、**LD_BIND_NOW** 環境変数を使用して実行できます。null 以外の値を設定 **LD_BIND_NOW** すると、プログラムの読み込み時にシステムローダーが解決していないシンボルをすべて検索します。

注記

詳細は、以下の man ページは本セクションに記載の情報に関連しています。

- ld.so(8)

パート III. ライブラリーサービス

システムコマンドは、優先順位、プロセッサーアフィニティー、およびスケジューリングポリシーを操作するために使用されます。ライブラリー関数を使用して、ユーザーアプリケーション内からこれらの要素を操作することもできます。

本セクションでは、ライブラリー関数を使用して優先順位、プロセッサーアフィニティー、およびスケジューラーポリシーを選択する方法を説明します。また、これらの変更の結果を確認する方法を説明します。

第11章 スケジューラーの設定

プロセスの設定と監視には、コマンドラインユーティリティーと Tuna グラフィカルツールの 2 つの方法があります。このセクションではコマンドラインツールを使用しますが、Tuna を使用してすべてのアクションも実行できます。

11.1. CHRT を使用したスケジューラーの設定

chrt は、スケジューラーポリシーおよび優先順位の確認および調整に使用されます。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。

特定のプロセスの属性を確認するには、**--pid** または **-p** オプションのみを使用してプロセス ID (PID) を指定します。

```
~]# 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
```

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を使用します。

表11.1 chrt コマンドのポリシーオプション

短いオプション	長いオプション	詳細
-f	**--fifo**	スケジュールを **SCHED_FIFO** に設定します。
-o	**--other**	スケジュールを **SCHED_OTHER** に設定します。
-r	**--rr**	スケジュールを **SCHED_RR** に設定します。

プロセスの優先度を設定するには、変更しているプロセスの PID の前に値を指定します。以下のコマンドは、PID 1000 のプロセスを **SCHED_FIFO** に、優先度 50 で設定します。

```
~]# chrt -f -p 50 1000
```

以下のコマンドは、同じプロセス (PID 1000) を **SCHED_OTHER** に、優先度 0 で設定します。

```
~]# chrt -o -p 0 1000
```

指定されたポリシーおよび優先度で新しいアプリケーションを起動するには、属性とともにアプリケーションの名前 (および必要な場合はパス) を指定します。以下のコマンドは、ポリシー **SCHED_FIFO** および優先度が 36 で **/bin/my-app** を起動します。

```
~]# chrt -f 36 /bin/my-app
```

注記

詳細は、以下の man ページは本セクションに記載の情報に関連しています。

- chrt(1)

11.2. PREEMPTION

プロセスは、完了したか、またはイベント (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 を使用する場合に起こります。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。

1 つのプロセスで自発的および自発的なプリエンプションが実行されるかどうかを確認するには、*PID* がプロセスの PID である、**/proc/*PID*/status** の内容を確認します。以下のコマンドは、PID 1000 のプロセスのプリエンプションを確認します。

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

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。

11.3. ライブラリー呼び出しを使用した優先度の設定

以下のライブラリー呼び出しは、リアルタイム以外のプロセスの優先度を設定するために使用されます。

- **nice**
- **getpriority**
- **setpriority**

これらの関数は、リアルタイム以外のプロセスの *nice 値* を取得および調整します。*nice 値* は、プロセッサー上で実行可能なリアルタイム以外のプロセスの一覧を順序付ける方法についてスケジューラーに提案されます。一覧の先頭にあるプロセスは、その後に戻るよりも早く実行します。

リアルタイムプロセスは、異なるライブラリーコールのセットを使用してポリシーおよび優先度を制御します。これは、本セクションで説明します。

重要

以下の関数はすべて、**sched.h** ヘッダーファイルを含める必要があります。関数から常に戻りコードを確認するようにしてください。適切な man ページには、使用されるさまざまなコードの概要が記載されています。

11.3.1. sched_getscheduler

sched_getscheduler() 関数は、指定 PID のスケジューラーポリシーを取得します。

```
#include <sched.h>

int policy;

policy = sched_getscheduler(pid_t pid);
```

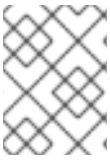
シンボル **SCHED_OTHER**、**SCHED_RR** および **SCHED_FIFO** は **sched.h** 定義されています。これらは定義されたポリシーを確認するか、またはポリシーの設定に使用できます。

```
#include <stdio.h>
#include <unistd.h>
#include <sched.h>

main(int argc, char *argv[])
{
  pid_t pid;
  int policy;

  if (argc < 2)
    pid = 0;
  else
    pid = atoi(argv[1]);

  printf("Scheduler Policy for PID: %d  -> ", pid);

  policy = sched_getscheduler(pid);

  switch(policy) {
    case SCHED_OTHER: printf("SCHED_OTHER\n"); break;
    case SCHED_RR:   printf("SCHED_RR\n"); break;
    case SCHED_FIFO:  printf("SCHED_FIFO\n"); break;
    default:   printf("Unknown...\n");
  }
 }
```

11.3.2. 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 1 つのパラメーター**sched_priority** 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。

この **sched_setscheduler** 関数には、**sched_setscheduler(pid_t pid, int policy, const struct sched_param *sp);** の 3 つのパラメーターが必要です。



注記

sched_setscheduler(2) man ページは、エラーコードを含む **sched_setscheduler** の考えられる値を一覧表示します。

pidが ゼロの場合、**sched_setscheduler()** 関数は呼び出しプロセスで動作します。

以下のコードの抜粋は、現在のプロセスのスケジューラーポリシーを **SCHED_FIFO** に、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;
}
```

11.3.3. sched_getparam および sched_setparam

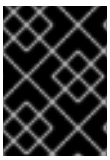
sched_setparam() 関数は、特定プロセスのスケジューリングパラメーターを設定するために使用されます。その後、**sched_getparam()** 関数を使用して確認できます。

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。

以下のコードは、指定のリアルタイムプロセスの優先度を読み取り、それを 2 つ増やします。

```
struct sched_param sp;
int ret;

/* reads priority and increments it by 2 */
ret = sched_getparam(0, &sp);
sp.sched_priority += 2;

/* sets the new priority */
ret = sched_setparam(0, &sp);
```

注記: 実際のアプリケーションで上記のコードを使用している場合は、関数から戻り値を確認し、エラーを適切に処理する必要があります。



重要

優先順位の増加に注意してください。この例では、継続的に 2 つを追加すると、最終的に無効な優先度になる可能性があります。

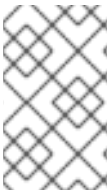
11.3.4. sched_get_priority_min および sched_get_priority_max

sched_get_priority_min および **sched_get_priority_max** 関数は、指定のスケジューラーポリシーの有効な優先度範囲を確認するために使用されます。

指定したスケジューラーポリシーがシステムによって認識されていない場合は、この呼び出しで唯一のエラーが発生します。この場合、関数は ***-1*** を返し、***EINVAL*** は **errno** に設定されます。

```
#include <stdio.h>
#include <unistd.h>
#include <sched.h>

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));
}
```



注記

SCHED_FIFO と **SCHED_RR** は両方とも、1 から 99 の範囲で数値指定できます。POSIX は、この範囲を適用する保証はありませんが、移植可能なプログラムはこれらの呼び出しを使用する必要があります。

11.3.5. sched_rr_get_interval

この **SCHED_RR** ポリシーは、**SCHED_FIFO** ポリシーとは若干異なります。**SCHED_RR** は、ラウンドロビンローテーションで同じ優先順位を持つ同時プロセスを割り当てます。この方法では、各プロセスに複数回割り当てられます。**sched_rr_get_interval()** 関数は、各プロセスに割り当てられた回数を報告します。

POSIX では、この関数が **SCHED_RR** プロセスでのみ機能しなければならない必要がありますが、この **sched_rr_get_interval()** 関数は Linux 上の プロセスの時系列の長さを取得することができます。

時系列情報は ***timespec*** または、ベース時間 1970 年 1 月 1 日 00:00:00:00 GMT以降の秒とナノ秒の数値で返されます。秒数とナノ秒を返します。

```
struct timespec {
  time_t tv_sec;  /* seconds */
  long tv_nsec; /* nanoseconds */
}
```

この **sched_rr_get_interval** 関数には、プロセスの PID と構造の timespec が必要です。

```
#include <stdio.h>
#include <sched.h>

main()
{
   struct timespec ts;
   int ret;

   /* real apps must check return values */
   ret = sched_rr_get_interval(0, &amp;ts);
```

```
    printf("Timeslice: %lu.%lu\n", ts.tv_sec, ts.tv_nsec);
}
```

以下のコマンドは、さまざまなポリシーおよび優先順位を使用して、テストプログラム **sched_03** を実行します。**SCHED_FIFO** ポリシーのあるプロセスは、0 秒と 0ナノ秒の回数を返します。これは無限であることを示します。

```
~]$ chrt -o 0 ./sched_03
Timeslice: 0.38994072
```

```
~]$ chrt -r 10 ./sched_03
Timeslice: 0.99984800
```

```
~]$ chrt -f 10 ./sched_03
Timeslice: 0.0
```

注記

詳細は、以下の man ページは本セクションに記載の情報に関連しています。

- nice(2)
- getpriority(2)
- setpriority(2)

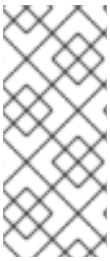
第12章 スレッドおよびプロセスの作成

プロセスとスレッドの作成はシステムの負荷に依存し、リソースの割り当てと CPU 時間の共有に不可欠です。シナリオによっては、イベント発生と処理の遅延が許容可能である場合があります。ただし、ほとんどの場合、不要なレイテンシーが生まれます。これを防ぐには、要求を処理するためにプロセスまたはスレッドのプールを常に事前に作成する必要があります。詳細は4章*スレッドおよびプロセス*を参照してください。

第13章 MMAP

mmap システムコールでは、ファイル (またはファイルのパーツ) をメモリーにマップできます。これにより、メモリー操作でファイルコンテンツを変更でき、システムコールや入出力操作を回避できます。

ディスクへの変更を常に同期し、データが失われる可能性のあるプロセスのハングを計画します。



注記

詳細は、以下の man ページと書籍は本セクションに記載の情報に関連しています。

- mmap(2)
- 『Linux System Programming』 by Robert Love

第14章 システムコール

14.1. SCHED_YIELD

この **sched_yield** 関数は、最初にプロセッサーが実行中のプロセス以外のプロセスを選択するよう設計されました。このタイプの要求は、書き込みの低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。

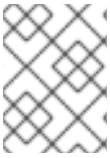
この **sched_yield()** 関数がリアルタイム優先度のプロセス内で使用されると、予期しない動作が表示される可能性があります。**sched_yield** を呼び出したプロセスは、その優先度で実行中のプロセスのキューの末尾に移動します。これは、同じ優先度で他のプロセスが実行していない状況で発生すると、**sched_yield** を呼び出したプロセスの実行は継続されます。プロセスの優先度が高い場合、ビジーループが発生し、マシンが使用できなくなる可能性があります。

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。

14.2. GETRUSAGE()

getrusage 関数は、指定のプロセスおよびそのスレッドから重要な情報を取得するために使用されます。これにより、利用可能なすべての情報が提供されるわけではありませんが、コンテキストスイッチやページフォールトなどの情報について報告します。

パフォーマンスチューニングとデバッグのアクティビティーの両方に関連する情報を提供するために、アプリケーションをインストルメント化することは簡単です。この **getrusage()** 関数は、指定されたプロセスとそのスレッドから重要な情報を取得するために使用されます。これは、**/proc/** ディレクトリー内の複数の異なるファイルからカタログ化する必要がありますが、アプリケーション上の特定のアクションまたはイベントと同期することが困難になります。自発的なコンテキストスイッチおよび自発的なコンテキストスイッチの量、メジャーおよびマイナーページのフォールト、使用中のメモリー量、その他の情報は、**getrusage()** 関数で取得できます。



注記

getrusage() 結果の報告に使用される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訳ではありません。一部は、互換性の理由でのみ保持されます。

この関数の詳細は、man ページの getrusage(2) を参照してください。

第15章 タイムスタンプ

15.1. ハードウェアクロック

NUMA や SMP などのマルチプロセッサーシステムには、複数のクロックソースインスタンスがあります。クロック自体とシステムイベントに対話する方法 (CPU 周波数のスケーリングや電源モードの入力など) は、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。

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。希望するクロックソースは Time Stamp Counter (TSC) ですが、利用できない場合には、HPET (High Precision Event Timer) が 2 番目に適しています。ただし、すべてのシステムに HPET クロックがあり、一部の HPET クロックは信頼できない場合があります。

TSC および HPET がない場合には、ACPI Power Management Timer (ACPI_PM)、プログラム可能な間隔タイマー (PIT)、リアルタイムクロック (RTC) などがあります。最後の 2 つのオプションは、読み込むコストが大きいか、解像度が低い (時間粒度) であるため、リアルタイムカーネルに対して最適なものです。

システムで使用可能なクロックソースの一覧を表示するには、**/sys/devices/system/clocksource/clocksource0/available_clocksource** ファイルを表示します。

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

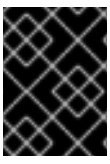
上記の出力例では、TSC、HPET、および ACPI_PM クロックソースが利用可能です。

現在使用中のクロックソースは、**/sys/devices/system/clocksource/clocksource0/current_clocksource** ファイルを読み取りて検査できます。

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

/sys/devices/system/clocksource/clocksource0/available_clocksource ファイルに表示されている一覧から、別のクロックソースを選択できます。これを行うには、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。たとえば、以下のコマンドは、使用中のクロックソースとして HPET を設定します。

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



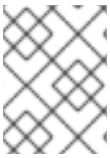
重要

カーネルは、最適なクロックソースを選択します。選択したクロックソースの上書きは、影響が明確に理解されない限り推奨されません。

TSC は通常優先されるクロックソースですが、ハードウェア実装の一部には足りない場合があります。たとえば、一部の TSC クロックは、システムがアイドル状態に切り替わったり、CPU がより深い C 状態 (電力節約状態) に入る場合や、速度または周波数のスケーリング操作を実行したときに停止することがあります。

ただし、追加のカーネルブートパラメーターを設定することで、これらの TSC の欠点の一部を回避できます。たとえば、***idle=poll*** パラメーターはアイドル状態にならないようにクロックを強制

し、***processor.max_cstate=1*** パラメーターによりクロックがより深い C 状態に入るのを防ぎます。ただし、どちらの場合も、システムが常に最大速度で実行されるため、電力消費量が増加することに注意してください。



注記

クロックソースの包括的なリストは、『Understanding The Linux Kernel』 by Daniel P. Bovet and Marco Cesati の 『Timing Measurements』 の章を参照してください。

15.1.1. ハードウェアクロックソースの読み取り

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 クロックから読み取ると、メモリー領域を読み取ることになります。TSC からの読み取りが速く、毎秒数百万のメッセージをタイムスタンプにすると、パフォーマンスが大幅に向上します。

現在のクロックソースを 10,000,000 回読み取りする簡単なプログラムを行で使用し、利用可能なクロックソースの読み取りに必要な期間を確認できます。

例15.1 ハードウェアクロックソースの読み込みコストの比較

この例では、**cat** コマンドの出力で示されているように、現在使用中のクロックソースは TSC です。この **time** コマンドは、クロックソースを 10 万回読み込むのに必要な期間を表示するために使用されます。

```
~]# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
~]# time ./clock_timing

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

クロックソースは HPET に変更され、1,000 万のタイムスタンプの生成に必要な期間を比較します。

```
~]# echo hpet > /sys/devices/system/clocksource/clocksource0/current_clocksource
~]# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
hpet
~]# time ./clock_timing

 real 0m12.263s
 user 0m12.197s
 sys 0m0.001s
```

手順は ACPI_PM クロックソースで繰り返します。

```
~]# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
~]# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
acpi_pm
~]# time ./clock_timing

 real 0m24.461s
 user 0m0.504s
 sys 0m23.776s
```

time(1) man ページには、コマンドの使用方法と、その出力の解釈方法の詳細情報が記載されています。上記の例では、以下のカテゴリーを使用しています。

- **real**: プログラム呼び出しからプロセスが終了するまでに費やされた合計時間。**real** には、**user** と **sys** 時間が含まれます。通常は、後の 2 つの合計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、この時間も待機に費やされた時間も **real** 下で計算されます。
- **user**: プロセスがユーザー空間で費やした時間。カーネルの介入を必要としないタスクを実行します。
- **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。

例15.1「ハードウェアクロックソースの読み込みコストの比較」 の結果から分かるように、タイムスタンプの生成の効率は、TPC、HPET、ACPI_PM です。これは、HPEET および ACPI_PM タイマーから時間値にアクセスするためのオーバーヘッドが増加するためです。

15.2. 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。カーネルによって選択され、システム全体に実装されているハードウェアクロックとは異なり、POSIX クロックは、システムの他のアプリケーションに影響を与えずに各アプリケーションで選択できます。

- **CLOCK_REALTIME**: これは、実際の時間を表します。つまり「wall time」とも呼ばれます。これは、壁の時計からわかる時間を意味します。このクロックは、タイムスタンプイベントとユーザーと対話する場合に使用されます。これは、適切な権限を持つユーザーが変更できます。ただし、クロックの値が 2 つの読み取り間で変更された場合に、誤ったデータが作成される可能性があるため、ユーザーの変更は注意して使用してください。
- **CLOCK_MONOTONIC**: システム起動以降に増加する時間を表します。このクロックはプロセスでは設定できず、イベント間の時間差を計算するのに推奨されるクロックです。このセクションの以下の例は、POSIX クロックとして **CLOCK_MONOTONIC** を使用します。

注記

POSIX クロックの詳細は、以下の man ページおよびガイドを参照してください。

- clock_gettime()
- 『Linux System Programming』 by Robert Love

所定の POSIX クロックの読み取りに使用される関数は **clock_gettime()** で、**<time.h>** 出て異議されます。この **clock_gettime()** コマンドは、POSIX クロック ID と timespec 構造の 2 つのパラメーターを取ります。これは、クロックの読み取りに使用する期間が埋められます。以下の例は、クロックの読み取りコストを測定する関数を示しています。

例15.2 clock_gettime() を使用した読み取り用 POSIX クロックのコストの測定

```
#include <time.h>

main()
```

```
{
 int rc;
 long i;
 struct timespec ts;

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

上記の例は、その他のコードを追加して、**clock_gettime()** の戻りコードの確認、***rc*** 変数の値の確認、または ***ts*** 構造の内容が信頼されるようにすることで改善できます。**clock_gettime()** man ページは、より信頼できるアプリケーションを作成するのに役立つより多くの情報を提供します。

重要

この **clock_gettime()** 関数を使用するプログラムは、**gcc** コマンドライン ***'-lrt'*** に追加して **rt** 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
```

15.2.1. CLOCK_MONOTONIC_COARSE および CLOCK_REALTIME_COARSE

clock_gettime() やなどの関数 **gettimeofday()** は、システムコールの形式でカーネル内の関数です。ユーザープロセスが **clock_gettime()** を呼び出す際に、対応する C ライブラリー (**glibc**) が要求された操作を実行する **sys_clock_gettime()** システムコールを呼び出し、その結果をユーザープロセスに返します。

ただし、このコンテキスト切り替えは、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。

カーネルにコンテキストが切り替わらないように、クロックの読み取りが速くなり、**CLOCK_MONOTONIC_COARSE** および **CLOCK_REALTIME_COARSE** POSIX クロックのサポートが VDSO ライブラリー機能の形式で作成されました。***_COARSE*** バリアントは読み取りが速く、1 ミリ秒 (ミリ秒) の精度 (解像度とも呼ばれます) を持ちます 。

15.2.2. clock_getres() を使用したクロック解決の比較

この **clock_getres()** 関数を使用すると、特定の POSIX クロックの解決が可能となります。**clock_getres()** は、使用される POSIX クロックの ID と、結果が返される timespec 構造のポインターと同じ 2 つのパラメーターを **clock_gettime()** として使用します。以下の関数を使用すると、**CLOCK_MONOTONIC** と **CLOCK_MONOTONIC_COARSE** 間の正確性を比較できます。

```
main()
{
int rc;
struct timespec res;

 rc = clock_getres(CLOCK_MONOTONIC, &res);
 if (!rc)
  printf("CLOCK_MONOTONIC: %ldns\n", res.tv_nsec);
```

```
 rc = clock_getres(CLOCK_MONOTONIC_COARSE, &res);
 if (!rc)
  printf("CLOCK_MONOTONIC_COARSE: %ldns\n", res.tv_nsec);
}
```

例15.3 clock_getres のサンプル出力

```
TSC:
 ~]# ./clock_resolution
 CLOCK_MONOTONIC: 1ns
 CLOCK_MONOTONIC_COARSE: 999848ns  (about 1ms)

HPET:
 ~]# ./clock_resolution
 CLOCK_MONOTONIC: 1ns
 CLOCK_MONOTONIC_COARSE: 999848ns  (about 1ms)

ACPI_PM:
 ~]# ./clock_resolution
 CLOCK_MONOTONIC: 1ns
 CLOCK_MONOTONIC_COARSE: 999848ns  (about 1ms)
```

15.2.3. C コードを使用したクロック解決の比較

以下のコードスニペットを使用すると、**CLOCK_MONOTONIC** POSIX クロックから読み取られたデータの形式を確認できます。timespec 構造の ***tv_nsec*** フィールドにある 9 桁すべては、クロックにナノ秒の解像度があるため意味があります。**clock_test.c** という名前のサンプル関数は、以下のようになります。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main()
{
 int i;
 struct timespec ts;

 for(i=0; i<5; i++) {
  clock_gettime(CLOCK_MONOTONIC, &ts);
  printf("%ld.%ld\n", ts.tv_sec, ts.tv_nsec);
  usleep(200);
 }
}
```

例15.4 clock_test.c および clock_test_coarse.c のサンプル出力

上記のコードで指定されるとおり、関数はクロックを 5 回読み取り、各読み取りごとに 200 マイクロ秒で読み取ります。

```
~]# gcc clock_test.c -o clock_test -lrt
~]# ./clock_test
218449.986980853
```

```
218449.987330908
218449.987590716
218449.987849549
218449.988108248
```

同じソースコードを使用して、これを **clock_test_coarse.c** に変更し、 **CLOCK_MONOTONIC** を **CLOCK_MONOTONIC_COARSE** で置き換えることで、以下のような結果になります。

```
~]# ./clock_test_coarse
218550.844862154
218550.844862154
218550.844862154
218550.845862154
218550.845862154
```

_COARSE クロックの精度は 1 ミリ秒であるため、timespec 構造の ***tv_nsec*** フィールドの最初の 3 桁のみが重要になります。上記の結果は、以下のように読み込むことができます。

```
~]# ./clock_test_coarse
218550.844
218550.844
218550.844
218550.845
218550.845
```

POSIX クロックの ***_COARSE*** バリアントは、タイムスタンプをミリ秒単位で実行できる場合に特に便利です。この利点は、ACPI_PM などの読み取り操作にかかる高コストのハードウェアクロックを使用するシステムにおいて、より明確になります。

15.2.4. time コマンドを使用した読み取りクロックのコストの比較

time コマンドを使用してクロックソースを 1,000 万回読み取り、利用可能なハードウェアクロックの読み取り **CLOCK_MONOTONIC** および **CLOCK_MONOTONIC_COARSE** 表現のコストを比較することができます。以下の例では、TSC、HPET、および ACPI_PM のハードウェアクロックを使用します。**time** コマンドの出力の暗号を解除する方法は、「ハードウェアクロックソースの読み取り」 を参照してください。

例15.5 読み取り用 POSIX クロックのコストの比較

```
TSC:
 ~]# time ./clock_timing_monotonic

 real 0m0.567s
 user 0m0.559s
 sys 0m0.002s

 ~]# time ./clock_timing_monotonic_coarse

 real 0m0.120s
 user 0m0.118s
 sys 0m0.001s

HPET:
```

```
~]# time ./clock_timing_monotonic

real 0m12.257s
user 0m12.179s
sys 0m0.002s

~]# time ./clock_timing_monotonic_coarse

real 0m0.119s
user 0m0.118s
sys 0m0.000s

ACPI_PM:
~]# time ./clock_timing_monotonic

real 0m25.524s
user 0m0.451s
sys 0m24.932s

~]# time ./clock_timing_monotonic_coarse

real 0m0.119s
user 0m0.117s
sys 0m0.001s
```

例15.5「読み取り用 POSIX クロックのコストの比較」 からわかるように、***_COARSE*** クロックが使用されると、**sys** 時間 (ユーザープロセスで必要なタスクを実行するためにカーネルが費やした時間) が大幅に短縮されます。これは、特に ACPI_PM クロックのタイミングで明確です。これは、POSIX クロックの ***_COARSE*** バリアントにより、読み取りコストが高いクロックのパフォーマンスが向上します。

第16章 詳細情報

16.1. バグの報告

バグの診断

バグレポートを作成する前に、以下の手順に従って、問題発生場所を診断します。これにより、問題解決に大きくサポートします。

1. 最新バージョンの Red Hat Enterprise Linux 7 カーネルがあることを確認してから、**GRUB** メニューから起動します。問題を標準カーネルで再現してみてください。問題が解決しない場合は、Red Hat Enterprise Linux 7 にバグを報告してください。

2. 標準カーネルの使用時に問題が発生しなかった場合は、Red Hat Enterprise Linux for Real Time 固有の機能拡張 Red Hat がベースライン (3.10.0) カーネルに適用したバグにより、バグにより変更が加えられる可能性があります。

バグの報告

バグが Red Hat Enterprise Linux for Real Time に固有であると判断した場合は、以下の手順に従ってバグレポートを入力します。

1. Bugzilla アカウントがまだない場合には作成します。

2. Enter A New Bug Report をクリックします。必要な場合はログインします。

3. **Red Hat** 分類を選択します。

4. **Red Hat Enterprise Linux 7** 製品を選択します。

5. カーネルの問題である場合は、コンポーネントとして **kernel-rt** を入力します。それ以外の場合は、影響を受けるユーザー空間コンポーネントの名前を入力します。

6. 問題を詳細に説明して、バグ情報の入力を継続します。問題の説明を入力する際には、標準の Red Hat Enterprise Linux 7 カーネルで問題を再現できるかどうかの詳細情報が含まれるようにします。

付録A 改訂履歴

改訂 1-6 7.7 GA 公開用ドキュメントの準備	Tue Aug 6 2019	Jaroslav Klech
改訂 1-5 7.6 GA 公開用ドキュメントの準備	Thu Oct 18 2018	Jaroslav Klech
改訂 1-4 7.5 GA 公開用ドキュメントの準備	Tue Mar 20 2018	Marie Doleželová
改訂 1-3 7.4 GA 公開用バージョン	Tue Jul 25 2017	Jana Heves
改訂 1-2 7.3 GA リリースのバージョン	Mon Nov 3 2016	Maxim Svistunov
改訂 1-1 7.2 GA 公開用バージョン	Fri Nov 06 2015	Tomáš Čapek
改訂 1-0 7.1 GA 公開用バージョン	Thu Feb 12 2015	Radek Bíba