マルチコア時代のリアルタイムシステムの設計

コンカレント日本株式会社
プロフェッショナルサービス部

目次

リアルタイムシステム

リアルタイムシステムと言って、すぐ思いつくのはシミュレータ用のグラフィック・システムですが、ロケットや航空機、船舶、自動車、鉄道などの種類によって、時間的分解能とデータ量が異なります。

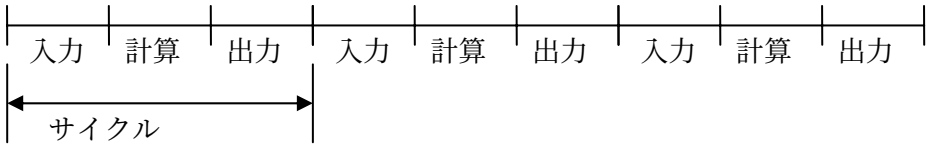
時間的分解能は、計算しなければならない運動方程式の複雑さから決定され、データ量はシミュレータの性格によって決定されます。

例えばグラフィックシミュレータの場合では、データ量（視野＝解像度×ディスプレイ数）は、低速に移動する場合には広く、高速に移動する場合には狭くなります。例えば船舶などの場合には 180 度以上の視野がなくてはなりませんが、航空機などではヘリコプタ等の例外はありますが、180 度までの視野が無くても実用上問題はありません。

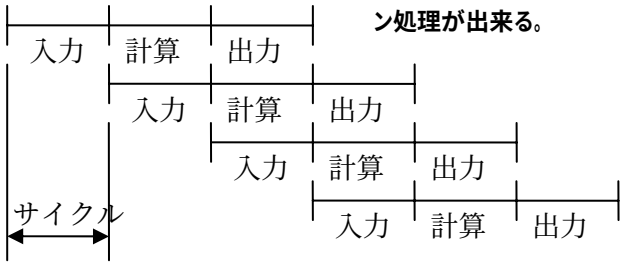
時間的解像度は、運動方程式を解く計算系や実際の目で見る映像系等の分類によって異なり、計算系の時間的解像度は、通常 10Hz から 1000Hz ぐらいの周期の間で行われます。

単純なシミュレータでは、**同期系クローズドループシステム**であるために、運動方程式を解く速度に合わせて、データの入出力を完了させる必要があります。

同期系クローズドループシステム　　**前段の出力に次段の入力が影響するため、サイクル内にすべて完了させなければならない。**



非同期系オープンループシステム　　**前段の出力に次段の入力が影響しないため、パイプライン処理が出来る。**

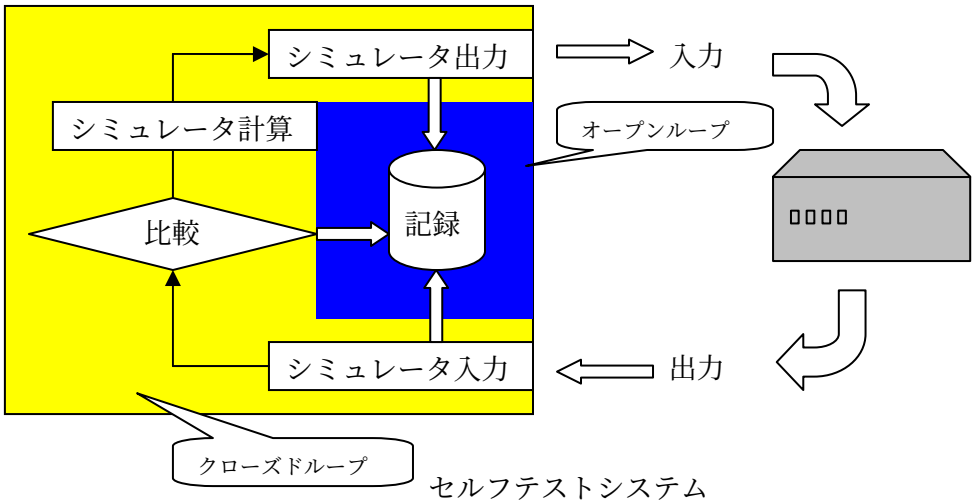


したがって、このサイクル内に運動方程式を解かなくてはならないのですが、最近の CPU の計算パワーはとても大きく、かなり細かくすることができます。

けれども時間的解像度を細かくすると、それに伴って入出力負荷を増大させますので、プロセスの**ディスパッチレイテンシィ(Dispatch Latency)**や実行サイクル時間の変動である**ジッター(jitter)**が重大な問題になってきます。

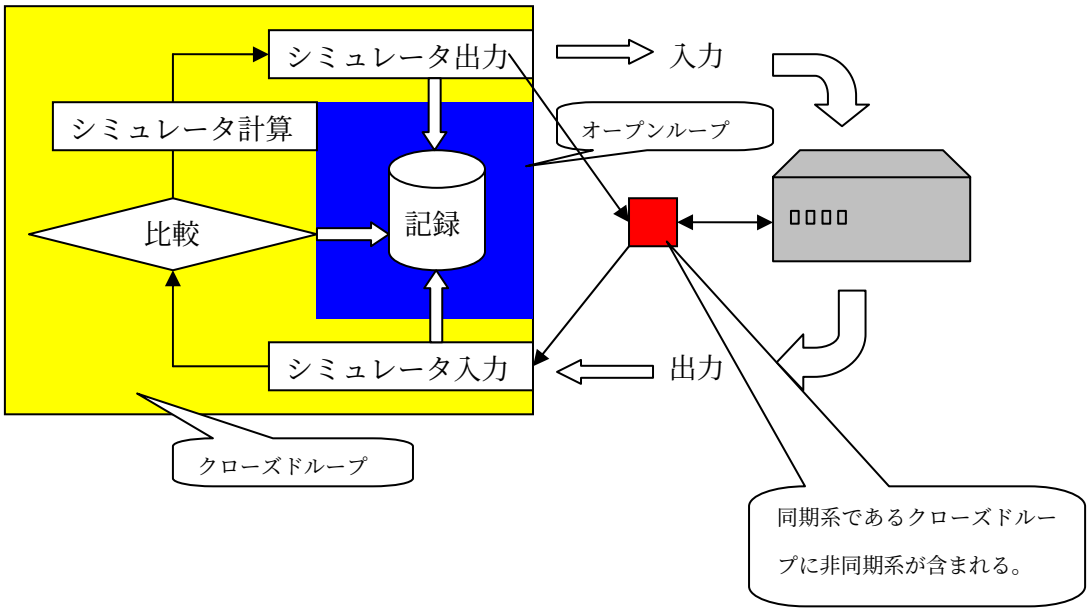
データレコーダのような、**非同期系オープンループシステム**では、クローズドループシステムに比較して、サイクルは短くてもジッターを気にする必要はあまりありません。

また、セルフテストシステムのような、試験体に、出力を与え、実結果と比較し、同時に記録を行うような、複雑なシミュレータでは、このオープンループとクローズドループの組み合わされたケースも存在します。



セルフテストシステム

最近のこのタイプのシステムでは、TCP/IP のような通信系も組み込まれ、同期系に非同期系が加わり、さらに複雑になっています。最悪の場合、サイクル内にデータが入力されないため、**遅れ周期でクローズドループを構成しなければならない場合**があります。

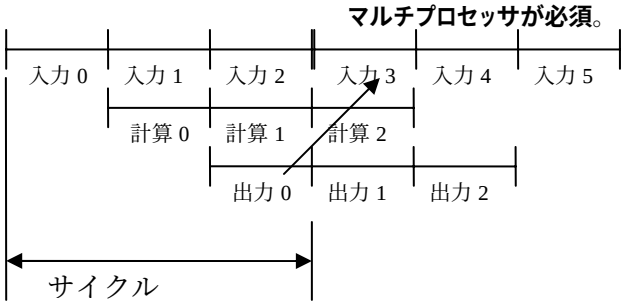


セルフテストシステム

非同期系クローズドループシステム

前段の出力に次段の入力が影響するが、入力0の計算結果の影響は、入力3まで現れない。

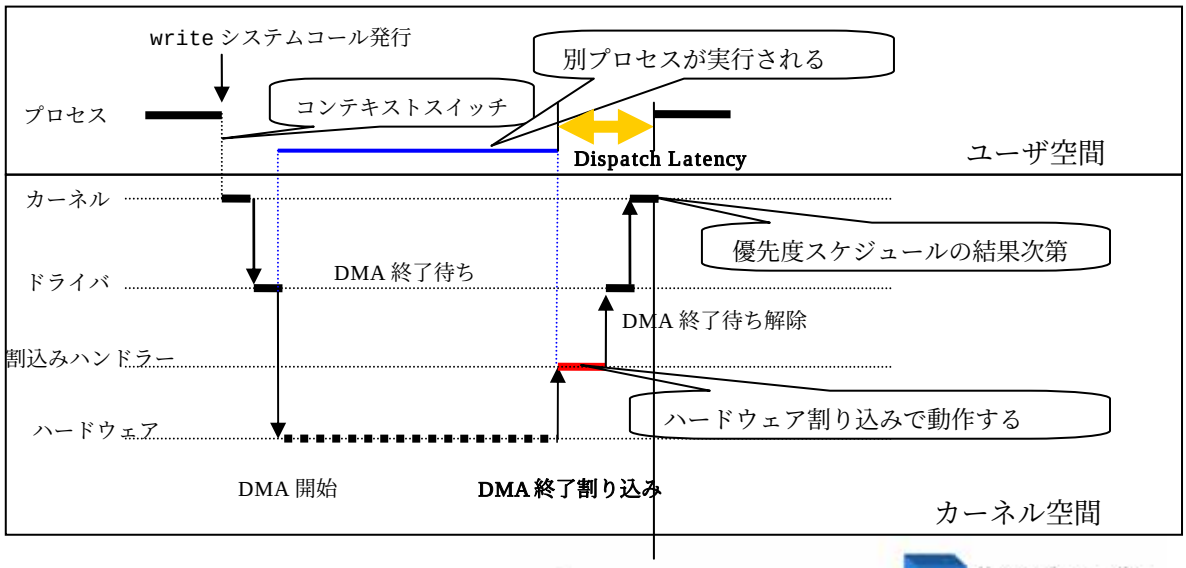
マルチプロセッサが必須。



ディスパッチレイテンシィ(Dispatch Latency)

例えば、単純な read/write システムコールでは、DMA を伴うため、最低でもコンテキストスイッチが２回、割込み待ち１回、優先度スケジュール１回を行っています。

その動作は、下図の様になります。（実際のデバイスとの通信時間を除いたオーバヘッド時間が知りたければ、/dev/null の read/write 時間を計れば判ります。）



上図の”DMA 完了割り込み”から呼び出しプロセスに戻るまでの時間を、図 1 の①から⑤の総和で**ディスパッチレイテンシィ”Dispatch Latency”**と呼びます。

また、このディスパッチレイテンシィには、図 2 に示す様に、割り込み禁止区間が含まれています。

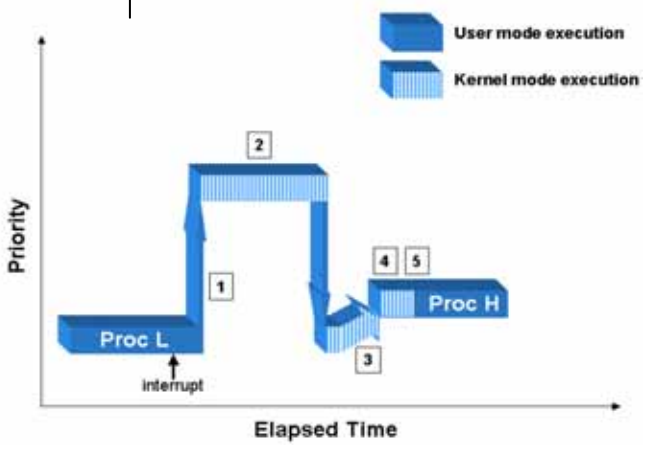


図1 通常のディスパッチレイテンシィ

カーネルは共有データ構造へのアクセスによって発生するデータ構造の破壊を防ぐために割り込みを禁止します。

カーネルのデータ構造がシステムコール･レベルでアクセスされる時、同時に、割り込みレベルでアクセスされる可能性がある場合には、常に、割り込みを禁止する必要があります。

割り込みは、ハードウェアが発生するため、どんな高い優先度を持ったプロセスも、たとえカーネルであっても妨げることは出来ません。これが、割り込みを禁止する理由です。

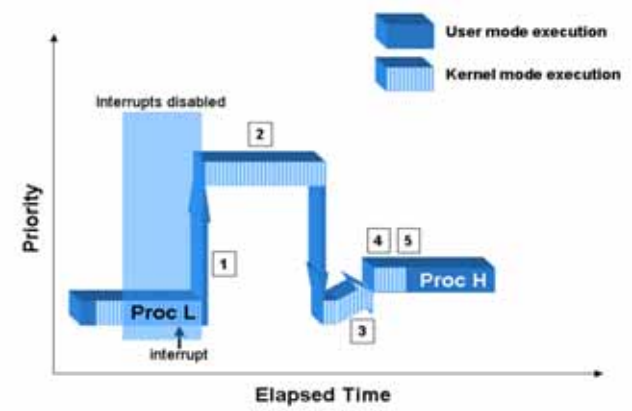


図2 割り込み禁止のディスパッチレイテンシィへの影響

割り込みが禁止される時、プロセスディスパッチレイテンシィも影響を受けます。なぜなら、応答しようとしている割り込みが、割り込みが再び許可されるまでアクティブにならないからです。この場合、割り込みを待っているプロセスにおけるプロセスディスパッチレイテンシィは、割り込み禁止の時間量によって変動し、この状態が**図 2** の中で説明されています。この図の中では、低優先度･プロセスは、割り込みを禁止したシステムコールを呼び出しています。高優先度割り込みが起こる時、それは何からも影響を受けません。なぜなら、現在、割り込みが禁止されているからです。低優先度･プロセスがそのクリティカルセクションを終了した時、割り込みを許可し、割り込みはアクティブになり、そして割り込みサービス･ルーチンが呼び出されます。そして、割り込み応答の通常のステップが、通常の形式で完了します。図 2-中の 1~5 でマークされている番号は、先に説明された通常のプロセスディスパッチレイテンシィのステップを表しています。

明らかに、割り込みが禁止されるオペレーティング･システム内のクリティカル･セクションは、かなり最悪のケースのプロセスディスパッチレイテンシィに備えて、最小にされなければなりません。

高優先度割り込みの影響

割り込みの受信は、割り込みを禁止するのと同じように、プロセスディスパッチレイテンシィに影響を与えます。ハードウェア割り込みを受信する時、システムは、現在の割り込みと同じかまたはより低い優先度の割り込みをブロックします。図 3 の中で、単純なケースが説明されています。そこでは、**ターゲット･プロセスの割り込みよ**

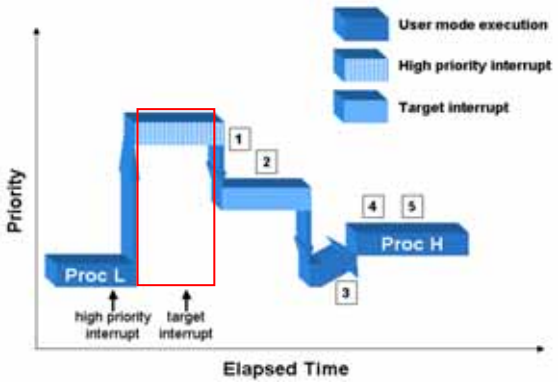


図 3 ディスパッチレイテンシィにおける高優先度割込みの影響

り前に、それより高い優先度の割り込みが起こり（図３の赤いボックスの部分）、より高い優先度の割り込み処理が終了するまで、ターゲット割り込みがブロックされています。図 3 の中の 1～5 でマークされている番号は、先に説明された通常のプロセスディスパッチレイテンシィのステップを表していることに注意してください。

割り込みの相対的な優先度は、プロセスディスパッチレイテンシィには影響を与えません。低優先度割り込みがアクティブになる時さえ、高優先度割り込みにおけるプロセスディスパッチレイテンシィ上に割り込む影響は同じです。これは、割り込みは常にユーザプロセスよりもより高い優先度で走っているからです。よって、高優先度割込みにおいて、割り込みルーチンが終了しても、その他の全ての割り込み処理が完了するまで、カーネルは、ユーザプロセスへのコンテキストスイッチの動作を行いません。

プロセスディスパッチレイテンシィの低優先度割り込みのインパクトは、図 4 内で説明されています。どのように事が扱われるかの順序は、図 3 内の高優先度割込みのケースとは異なりますが、プロセスディスパッチレイテンシィ上の影響は同じであることに注意してください。図 4 の中の 1～5 でマークされている番号は、先に説明したように、通常のプロセスディスパッチレイテンシィのステップを表していることに注意してください。

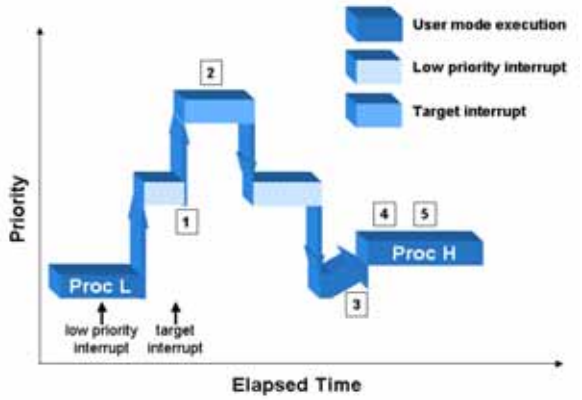


図 4 ディスパッチレイテンシィにおける低優先度割込みの影響

プロセス複数待ち時間における多重割り込みの影響

“割り込みを禁止することの影響”と、**”プロセスディスパッチレイテンシィにおける割り込みの受信の影響”**との最も大きな違いの 1 つは、割り込みはアプリケーションの実行に非同期的に、そして予測不可で起こるという事実です。（割り込みのシールディングの様々なレベルを理解するのに、これは重要なことです。）

ある CPU 上に複数割り込みが受信される時、ワーストケースのプロセスディスパッチレイテンシィへの影響は、重大です。

これは、割り込みがスタックに積まれ、高優先度割り込みにおけるプロセスディスパッチレイテンシィが完了する前に、1 つ以上の割り込みサービス･ルーチンが処理されなければならないようなことになるからです。図 5 は、高優先度割り込みに応答しようとしながら、2 つの割り込みがアクティブになるケースを示しています。図 5 内の 1～5 でマークされている番号は、先に説明されたように通常のプロセスディスパッチレイテンシィのステップを表しています。CPU が割り込みを受け取る時、その CPU は、より低い優先度の割

り込みを禁止します。この時間内により低い優先度の 2 つ目の割り込みがアクティブになれば、オリジナル割り込みがアクティブである間は、割り込めません。第 1 の割り込み処理が完了する時、第 2 の割り込みがアクティブになり、そして処理されます。

もし第 2 の割り込みが最初の割り込みよりも高優先度であるなら、それはすぐにアクティブになります。第 2 の割り込みがその処理を完了する時、第 1 の割り込みが再びアクティブになります。いずれの場合も、中断されている全ての割り込みが処理されるまで、ユーザ･プロセスは処理されません。

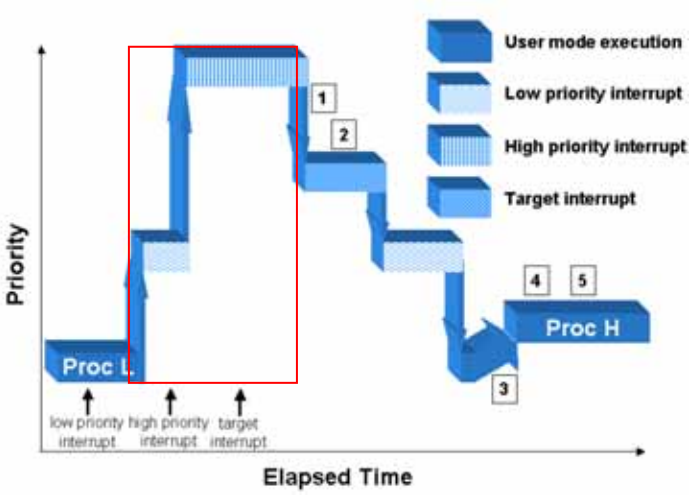


図 5　プロセス複数待ち時間における多重割り込みの効果

最悪の場合、割り込みがアクティブであり続け、システムが高優先度割り込みに応答することを決して許さない致命的なケースがあります。多重割り込みが、特定の CPU に割り当てられる場合、プロセスディスパッチレイテンシィはその CPU 上でより予測できないものになります。なぜなら、割り込みは多重に積み重ねられるからです。(注：図５において、赤の部分が無限に続く)

プリエンプションの禁止の影響

RedHawk Linux には、決定的なセクションがあり、それは割り込みレベルでは決してロックされない共有リソースをプロテクトするものです。この場合、このクリティカル･セクションの間、割り込みをブロックする理由はありません。しかし、このクリティカル･セクションの間に起こるプリエンプションは、もしも新しいプロセスが同じクリティカル･セクションに入ってくれれば、共有リソースを破壊します。よって、この種類のクリティカル･セクションにおけるプロセスの実行の間は、プリエンプションが禁止されます。プリエンプションの禁止は、割り込みの受信を遅らせません。しかし、その割り込みが高優先度･プロセスを呼び起こせば、プリエンプションが再び許可されるまで、そのプロセスに切り換わることはできません。同じ CPU が要求することとして、最悪のケースのプロセスディスパ

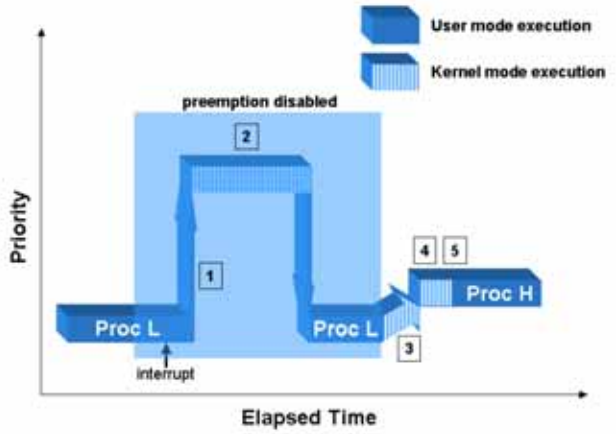


図 6　プリエンプション禁止の効果

ッチレイテンシィ上の実際的な効果は、あたかも割り込みが禁止されたものと同じです。プロセスディスパッチレイテンシィ上のプリエンプションの禁止の効果は、図 6 内で説明されています。図 6 内で 1～5 でマークされている番号は、先に説明された通常のプロセス・ディスパッチ待ち時間のステップを表していることに注意してください。

マルチコアを利用したリアルタイムシステム

ここまでの説明で理解出来るように、多重に発生する割り込みを制御することは難しく、デターミニスティックに解析することを困難にします。

シングル CPU でのリアルタイムシステムの設計の難しさは、この点にあり、リアルタイムシステム設計の要点は、割り込みの整理にあると言うことです。

しかし、現在主流のマルチ CPU／マルチコアを利用するとこの問題は、実に単純になります。

手順は、

- 全てのバックグラウンド処理をブートＣＰＵに割り当てる。
- 全ての割り込みをブートＣＰＵに割り当てる。
- リアルタイム性の必要なデバイスの割り込みをブートＣＰＵ以外に割り当てる。

これにより、複雑な多重割り込み問題は解消され、問題は単純化します。
この手法は、Concurrent だけではなく、RedHatMRG や SolarisOS も導入している古典的なマルチ CPU プローチですが、その実装は異なります。
Concurrent/RedHawk では、この手法をシールディングと呼んでいます。

割込みルーチンのマルチ CPU 処理

前述のように、シングル CPU では応答時間を保証することができないため、マルチ CPU にすることで多重割込みの弊害をなくし、重要な外部イベントに対して一定の時間内の応答を保証することを考えました。これを実現するために、次の３点をカーネルで行っています。

１）割込み処理の CPU 割付を自由に変更できる。
２）入出力処理をシンメトリックに処理する。
３）割込み処理をスレッドで、非割込み処理の環境で実行する

１）の機能は、RedHawk だけがユーザコマンドで実行可能です。その他の OS ではコンフィグレーションが必要だったり、ハードウェアのジャンパーの変更を必要とします。

２）の機能は、SMP 対応のすべての Linux で使用可能です。

３）の機能は Linux のドライバによっては、使用出来ないことがあります。

通常、割込み処理ルーチンは、自分自身のコンテキストを持っていないために、割り込

んだプロセスのコンテキストを使用して割込み処理を実行しています。このため割込み処理ルーチンでは、処理を終了するまで待ちに入る事を許されません。

ところが、割り込まれたプロセスが非常に優先度の高い処理であり、割込み処理ルーチンがもっとも低いプロセスのためであっても、割込み処理とプロセスの実行優先度には、なんの関係付けも行われていないため、割り込まれたプロセスの優先実行は、なんら保証されません。

つまり、**”あらゆる割込み処理は、すべてのプロセスに対して優先的に実行される”**という、暗黙の制約条件がついていると考える事ができるわけです。

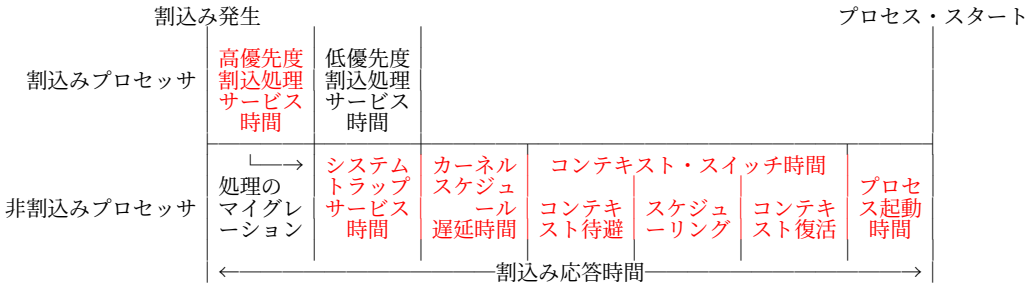
したがって、リアルタイムの応答性を保証するには、割込み処理を長時間続けてはならないことになります。

しかし、現実のデバイス・ドライバでは、ステータス・レジスタ等のチェックなどに時間がかかり、短時間で終了できないケースがままあります。

この問題は、割込み処理においてコンテキストスイッチを起こすか、割込み処理を非割込み処理に移して処理させれば、割込み処理を行うプロセスと割り込まれたプロセスの間で、優先順位による調停が可能になるため、問題は解決できます。

Linux2.6 では、割込み処理を非割込み処理に移すために”タスクレット"(tasklet)と”ワークキュー”(workqueue)を導入しています。

このような機構によって下図のように、割込みを受け付けたブート CPU が他の CPU に対して処理を動的に移動する事ができ、割込みの応答性を保証することができます。



マルチ CPU での割込み応答時間

シールディングはどのようにしてリアルタイム動作を向上させるのか

シールディングを行う時、全てのシールディング属性は、ＣＰＵ単位に行われ、その上で実行されるプロセスに、デフォルトで許可されます。これは、シールドされている CPU 上で最もリアルタイム性の高い実行環境を提供します。

シールディング属性の属性のいくつかは通常のシステムの機能に副作用をもたらすので、シールディング属性のそれぞれの影響を完全に理解しているべきです。

現在サポートされているシールディング属性に 3 つのカテゴリがあります。

- バックグランド･プロセスからのシールディング
- 割り込みからのシールディング
- ローカルな割り込みからのシールディング

これらの属性のそれぞれは、CPU 個々に選択できます。

バックグランド･プロセスのシールディング

このシールディング属性によって、システム内のリアルタイムプロセスのために CPU を確保することができます。このシールディング属性は CPU に、割り込みに対して最速の、最も予想可能な応答を持たせたい時に、許可されるべきです。プロセスディスパッチレイテンシィにおける最良の保証は、割り込みに応答するタスクのみが、その割り込みが指向されている CPU 上で実行を許されている際に得られます。

CPU がバックグランド･プロセスを走らせることを許される時、それは高プライオリティ･タスクのプロセスディスパッチレイテンシィに影響を与え、そのタスクは、その CPU に指向された割り込みに対してとても決定性のある応答を要求するものです。これは、割り込みまたはプリエンプションを禁止するシステムコールを、バックグランド･プロセスが潜在的に生成するからです。これらの動作は、“ 割り込みの禁止の効果 ” 及び “ プリエンプションの禁止の影響 ” のセクションで説明したように、プロセスディスパッチレイテンシィに影響を与えます。

CPU がバックグランド･プロセスを走らせることを許される時、高プライオリティ･プロセスの実行での決定性には影響を与えません。これは、バックグランド･プロセスは、高プライオリティよりも低プライオリティを持っていることを前提とします。バックグランド･プロセスは、シグナルのような他のカーネル･メカニズム経由でプロセスを呼び起こすのに要する時間に影響を与えることに注意します。

システム内のそれぞれのプロセスまたはスレッドは CPU バインドマスクを持ちます。

CPU バインドマスクは、どの CPU 上でプロセスまたはスレッドが実行するのを許されるのかを決定します。CPU バインドマスクは親から継承され、sched_setaffinity (2)システムコール経由で設定されます。CPU がプロセスからシールドされている時、シールドされている CPU を含むだけの CPU のセットへそれらの CPU バインドが明確に設定されているプロセス及びスレッドを、その CPU で実行します。換言すれば、もしプロセスがその CPU バインドマスクの中にシールドされていない CPU を持つなら、プロセスは、シールドされていない CPU 上を走るだけです。バックグランド・プロセスからシールドされている CPU 上でプロセスまたはスレッドを走らせるためには、それは、シールドされた CPU のみを指定する CPU バインドマスクを持たなければなりません。

Linux が生成したある程度のカーネル・デーモンは、システム内のどの CPU 上にも複製されています。プロセスから CPU をシールドすることは、これらの“ CPU ごとの ”デーモンをシールドされている CPU から取り除きません。これらのデーモンの影響は、カーネル・コンフィグレーション、またはアプリケーション動作の考慮された制御によって回避することができます。CPU ごとのカーネル・デーモンからのジッターを回避する方法は、RedHawk Linux User Guide 付録 F で述べられています。

割り込みのシールディング

このシールディング属性によって、システムによって受け取られる割り込みのサブセットのみを処理するために CPU を確保することができます。最速の、最も予測可能なプロセスディスパッチレイテンシィを持つことが望まれる時、またはアプリケーションの実行時間の中で決定性を持つことが望まれる時に、このシールディング属性が許可されるべきです。

CPU 上では、割り込みは常に最高のプライオリティの動作であるため、割り込みの扱いは、プロセスディスパッチレイテンシィ、及び高プライオリティ・タスクの中で通常のコード・パスを実行するのにそれが要する時間双方に影響を与えます。

それぞれのデバイス割り込みは、IRQ と結合しています。これらの IRQ は結合している CPU バインドを持ち、それは、どの CPU が割り込みを受け取ることを許されるかを決定します。割り込みが特定の CPU に経路を持たなければ、割り込みコントローラは、IRQ バインドマスク内の CPU のセットから割り込みが生成される時の割り込みの扱いにおいて、CPU を選択します。IRQ バインドは、shield (1)コマンドによって、または /proc/irq/N/smp_affinity を通して、修正されます。

通常の Linux の i386 アーキテクチャ上では、kirqd デーモンは、CPU における割り込み負荷のバランスをとるために、周期的に IRQ バインドを調整します。このデーモンは割り

込みシールディングと競合し、そして IRQBALANCE カーネル･コンフィグレーション･オプションを通して、全ての RedHawk Linux カーネル･コンフィグレーション中では、デフォルトで禁止されています。それは、カーネル･ブート･パラメータ“ noirqbalance ”で、または IRQBALANCE カーネル･パラメータを許可することによって、許可することができます。

もしも全ての CPU 上で全ての割り込みを禁止することが望まれるのならば、推奨される手順は、ブート CPU の 1 つを除き、残り全てを割り込みからシールドし、そしてシールドされていない CPU 上で local_irq_disable (2)を呼び出してください。

ある程度の動作はシールドされている CPU へ割り込みが送られることを引き起こします。これらの･CPU 間割り込みは、他の CPU にいくつかの CPU ごとの特定のタスクを扱わせる方法として使用されます。CPU 間割り込みは潜在的にシールドされている CPU において、目立つジッターを引き起こします。完全な説明については、RedHawk Linux User Guide 付録 G を参照してください。

ローカル割り込みからのシールディング

ローカル割り込みは、それぞれの CPU と結合しているプライベート･タイマーにおける特別な割り込みです。RedHawk Linux 下では、このタイマーは、カーネル内、及びユーザ･レベルで様々なタイムアウト･メカニズムにおいて使用されています。デフォルトでは、この割り込みはシステム内の全ての CPU 上で許可されています。

この割り込みは 10 ミリ秒ごとに発生し、ローカル割り込みを、システム内で最も頻繁に実行される割り込みルーチンになります。よって、ローカル割り込みは、リアルタイム･アプリケーションへのジッターの大きな要因です。

CPU がローカル･タイマーからシールドされている時、ローカル･タイマーは有効的に禁止されており、そしてその CPU とバインドしているローカル･タイマーが供給する機能はもはや動作しません。しかし、それらは、ローカル･タイマーがシールドされていない他の CPU 上で走り続けます。他の方法経由で他のものが供給されている間に、これらの機能のいくつかは失われます。

ある特定の CPU 上でローカル割り込みが禁止される時に失われる機能の 1 つは、CPU 実行時間アカウンティングにおける、低精度メカニズムです。これは、この CPU 上で実行されているそれぞれのプロセスによって、どれほどの CPU 時間が使用されているかを計るメカニズムです。ローカル割り込みが発生するたびに、時間の最後のクロック間隔は、割り込みされたプロセスにチャージされます。もしも高精度プロセス･アカウントがコンフィ

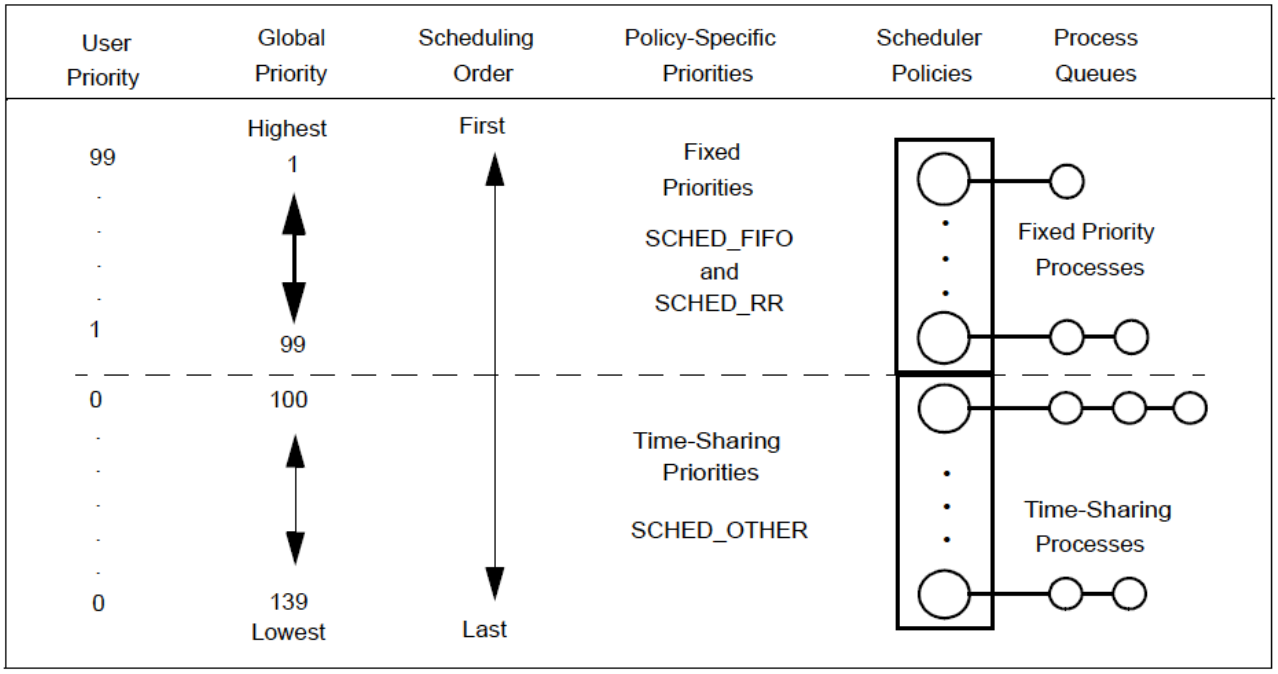
グされれば、CPU 時間はローカル割り込みが許可されているかいないかに関わり無く正確にアカウントされます。

CPU がローカル･タイマーからシールドされている時、ローカル割り込みは POSIX タイマーにおいて使用され続け、そしてプロセスによるナノスリープの機能は、シールドされている CPU にバイアスされます。この理由のため、特定のシールドされている CPU 上の最良の動作のためにローカル・タイマー割り込みを完全に消すことがクリティカルなら、**POSIX タイマーまたはナノスリープ機能を利用しているアプリケーションは、その CPU にバイアスされるべきではありません。**もしもそのシールドされている CPU 上でプロセスが走ることが許されないのなら、そのタイマーはプロセスが走ることが許される CPU へ移動されます。

プロセス･スケジューラ

図 7 は、どのようにスケジューラが動作するのかを説明しています。

図 7　RedHawk Linux スケジューラ



プロセスが生成される時、それは、そのポリシー内のスケジューリング･ポリシー及びプライオリティを含めて、そのスケジューリング･パラメータを継承します。デフォルトのコンフィグレーションでは、**SCHED_OTHER** ポリシーでスケジュールされたタイムシェアリングプロセスとして、プロセスは開始します。

ユーザ･プライオリティ値（sched_priority）は、それぞれのプロセスに割り当てられています。**SCHED_OTHER** プロセスは、0 のユーザ･プライオリティに割り当てられるのみです。**SCHED_FIFO** 及び **SCHED_RR** プロセスは 1 から 99 までの範囲のユーザ･プライオリティを持ちます。

スケジューラは、スケジューリング･ポリシー特定のプライオリティをグローバル･プライオリティへと変換します。グローバル･プライオリティは、RedHawk Linux カーネルが内部的に使用しているスケジューリング･ポリシー値です。スケジューラは、それぞれのグローバル･プライオリティ値において、動作可能なプロセスのリストを維持しています。**SCHED_OTHER** スケジューリング･ポリシーと関連する 40 のグローバ

ル･スケジューリング･プライオリティがあり、固定プライオリティ･スケジューリング･ポリシー（**SCHED_RR** 及び **SCHED_FIFO**）と結合する 99 のグローバル･スケジューリング･プライオリティがあります。スケジューラは最高グローバル･プライオリティの空でないリストを探し、そして、現在の CPU 上の実行においてこのリストの最前にあるプロセスを選択します。

スケジューリング･ポリシーは、それぞれのプロセスにおいて、リスト内のプロセスがブロックされたかまたは動作可能になった時に、同等のユーザ･プライオリティのプロセスのリストのどこに入れるのか、またこのリスト内でのプロセスの相対的位置を決定します。

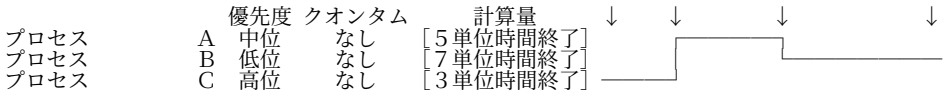
固定プライオリティ･プロセスが、ある特定の CPU において即実行のものである限り、その CPU 上でタイムシェアリングプロセスが走ることはありません。

スケジューラが CPU へプロセスを割り当てれば、プロセスは、それがそのタイム・クワンタム、スリープ、ブロックを使い切るまで、またはより高いプライオリティ･プロセスにプリエンプトされるまで、実行させます。

ps (1)及び **top (1)**によって表示されているプライオリティは、内部的に計算された値で、ユーザが設定したプライオリティを間接的にのみ反映していることに注意してください。

固定優先度ＦＩＦＯスケジューリング(SCHED_FIFO)

このスケジューリングポリシーでは、優先度の高いプロセスが実行権を持っている場合、そのプロセスが（システムコール発行等で）待ちの状態になるか、より高い優先度を持ったプロセスが実行可能状態にならない限り実行権を放棄しません。同一の優先度の場合ＦＩＦＯキューで、先に実行されたプロセスの終了を待ちます。

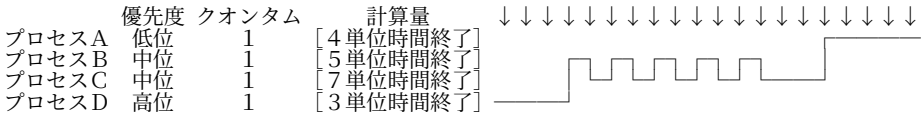


固定優先度スケジューリング

固定優先度ラウンドロビン・スケジューリング(SCHED_RR)

このスケジューリングポリシーでは同じ優先度のタスクに指定時間だけCPUの使用権を与えます。もし優先度の高いプロセスが実行可能になるとこちらが優先され、そのプロセスの実行が終了後、元のタスクの途中から実行を続けます。

リアルタイム優先度をもったプロセスは、クオンタム時間だけは、スケジューラに邪魔されず実行を続けることができますが、この時間を使いきるとシステムがトラップを発生し、スケジューラは他に優先度の高いプロセスが実行できるかどうかをチェックします。このクオンタム時間は、優先順位に関係づけてカーネル生成時にパラメータ設定するため動的に変化させることはできません。

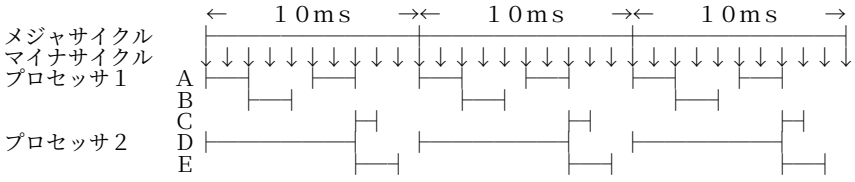


固定優先度ラウンドロビン・スケジューリング

タイマ割込みによる定周期スケジューリング(FBS)

これはタイマに同期させて周期的にプロセスを起動するもので、シミュレータやプロセス制御などのアプリケーションには、このスケジューリング・アルゴリズムをよく使用します。

シミュレータ等の場合１m秒から５０m秒ぐらいのフレームタイムで処理しますが、一定周期で起動できる最小時間がリアルタイムシステムの性能を決定するため、ユーザの要求はよりきびしいものになります。



定周期スケジューリング

プロセス制御やシミュレータ等のアプリケーションプログラムは、このメジャーサイクルの中で［入力］－［計算］－［出力］を終えなければなりません。

したがって、マイナーサイクルの周期が短くなれば、サイクルに占めるオーバーヘッドの割合が多くなり、オーバーヘッド時間の変動が問題になります。

これは、前述のすべてのパフォーマンスが影響してくるのに加えて、カーネルのチューニングが巧くいっていないと、（たとえば１００回に１回、起動が遅れるといったように）周期的に誤動作するといったことも起こるためどうしても長いマイナーサイクルになりがちです。

多くの場合この原因はシステム設計にあり、タイマのイベント処理部がブートCPUでし

か実行できない、他のスケジューリング機構と競合を起こしている等が考えられます。（このため、周期スケジュール機能を提供しているベンダは少数です。）

メモリ管理とキャッシュ

リアルタイムプロセスは主記憶を大量に消費します。ＵＮＩＸが出回り始めた当初は少ない主記憶と遅い二次記憶装置であったために、少し大きなプログラムを実行するとシステムがスワップイン／スワップアウトを繰り返し、その実行速度の遅さに耐えられないユーザが多くいました。

メモリとディスクが速く安く大容量になった現在、こんなことが問題になるケース少ないでしょうが、リアルタイムシステムでは、ちょうど同じような現象がおこります。

この場合耐えられないのは人間ではなくリアルタイム処理です。リアルタイムプロセスがページング／スワッピングされるとディスク装置からスワップインされる分だけ応答時間が遅くなります。したがってリアルタイムプロセスは、主記憶に常駐させる機構が必要です。

Linux では、このメモリロック機構がシステムコールとして実装されています。

このシステムコールでは、プロセスのテキストセグメント、データセグメントのどちらかあるいはその両方をロックする事に加えて、データをページ単位でロックする機構が存在します。

メモリ管理を考えるときに、もう一つ問題になるのは、キャッシュの一致性とバスアクセス競合の問題です。

現在使用される、ほとんどの代表的なマイクロプロセッサにはキャッシュが内蔵されていますし、マルチ CPU 構成のワークステーションも一般的になってきました。

キャッシュの不一致問題は、このようなマルチ CPU 構成はもとよりシングル CPU でもＤＭＡコントローラ等のマルチマスタ構成だとこのキャッシュの不一致が起こります。

キャシュの不一致問題とは図の初期状態のように正しい状態で、同一データを共有しているときに、いずれかの CPU が、データを更新すると図の中間状態のように、キャッシュの内容が不一致状態になることをいいます。

図Ａで示したライトスループロトコルの場合ただちに同様のデータをメモリに書き出し、他のキャッシュのタグをダーティ状態にします。その後ダーティ状態のキャッシュを読みだそうとするとメモリから正しいデータを読み出しキャッシュを更新します。

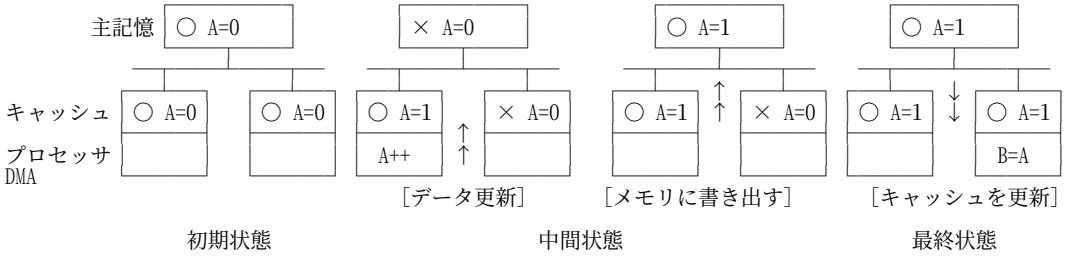
このプロトコルの欠点は、キャッシュ更新時に必ずデータをメモリに書き出すためマルチ CPU（マスタ）構成時にバストラフィックを増加させる原因になることです。

またＤＭＡなどでメモリの内容を直接書き換えた場合、キャッシュは更新されませんので、特別な操作でキャッシュをフラッシュし正常な動作を保証しなくてはなりません。（デバイスドライバでキャッシュをフラッシュします）

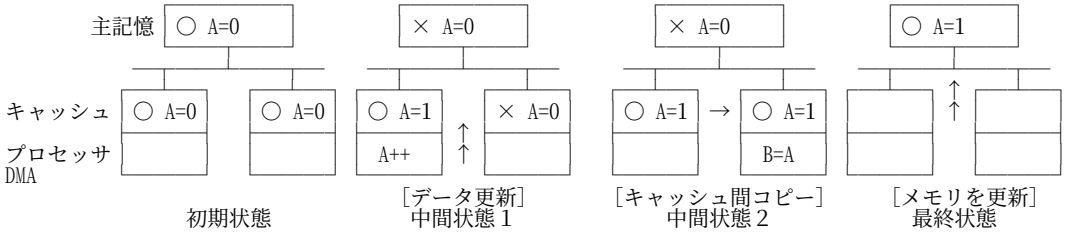
これらの欠点を解決したのがコピーバック・プロトコルです。（スヌーピング・プロトコルとも呼ぶます）
このプロトコルを実現するためには、バスを監視する専用モジュールが存在することを前提にしています。このようなコントローラをスヌーピング・コントローラと呼びます。

基本的な考え方は、メモリアクセスを最小限におさえるため、キャッシュの更新時にキャッシュのみ書換えをおこない、メモリの内容はダーティ状態のままにしておきます。メモリの内容が更新されるのは、キャッシュの内容がキャッシュから排出される際にメモリに書き戻されます。

他の CPU のキャッシュの更新は、スヌーピングコントローラによりキャシュ間で高速に行われ、他のバスマスタ（DMA等）のアクセスに対しても同様の動作になります。



図A　ライトスルー方式のキャッシュ制御



図B　コピーバック方式のキャッシュ制御

共有メモリ

共有メモリは、以下の３つの方法で実現できます。いずれも同様の機能を提供しますが、それぞれ特徴があります。

１）POSIX1003.1 規格のシェアードメモリ

shm??()コールの形での標準共有メモリ。この方法によりプロセスは、データを共有データとして宣言できその利点は移植性にありますが、カーネルにその管理がゆだねられます。

共有メモリ（shared memory:shm）は、複数のプロセス間で共通に参照されるメモリ空間で、各プロセスに同一メモリ空間がマップされます)。

２）POSIX1003.1b 規格のシェアードメモリ

shm_xx()コールの形での標準共有メモリ。この方法では、共有メモリをファイルと同じように共有データとして宣言できます。ユーザプロセスにその管理がゆだねられます。

３）スレッド

多重スレッドは、あまりシステム資源を必要とせず、プロセスより制御が容易なもっとも便利で効果的なデータの共有方法です。

４）POSIX1003.1b 規格のメモリマッピング

シェアードメモリと同様に、プロセスはデータを共有できますが、その他にバス上のデバイスを直接制御する場合に大きな効果を生みます。

メモリマップはＢＳＤ版ＵＮＩＸに基礎を置き、標準の共有メモリと同様の機能を提供します。

このインターフェースは、Linux から独立しているので、メモリマッピングを使用するプログラムは（Linux の設定変更があっても）システム・コンフィグレーション・ファイルやプログラムを変更せずにシステムで実行できる特長を持ち、主な用途として以下の２つが

あります。

- メインメモリの一部をＯＳの管理下から外し、予約されたメモリ（リザーブドメモリ）としてプロセス間通信に使用する。
- デバイスを外部メモリとして使用し、プロセス間通信あるいはプログラムＩ／Ｏデバイスとして使用する。

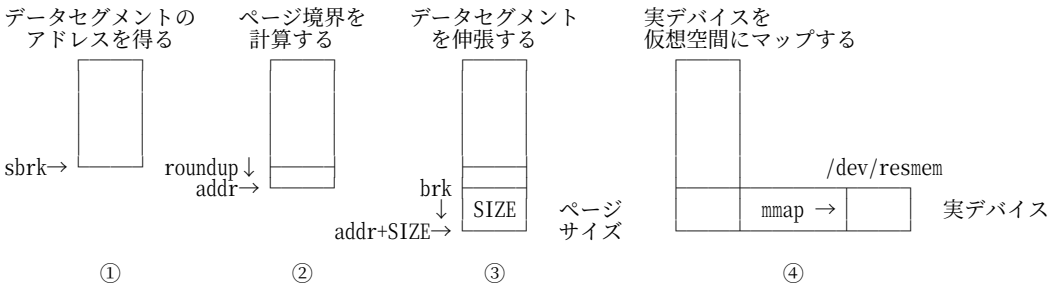
リザーブドメモリは、メインメモリの一部を予約メモリとして、Linux のメモリ管理から除外されます。
したがって、主記憶を物理的に減少させるため、システムのパフォーマンスを劣化させる場合もありますが、Linux の管理を離れますので、プロセス終了後にもメモリイメージが存在し、使用方法が完全にユーザプロセスに任されます。

この機能は、PCIbus 上のメモリ、あるいはプログラムＩ／Ｏデバイスとしても使用できるため、仮想記憶を使用せず大容量の共有メモリを使用したい場合や、リフレクティブメモリを使用してマルチシステムを構成する場合にも使用されます。

この機能を動作させるためには、カーネル起動時に mem=指定して立ち上げをおこなう必要があります。

実際のプログラムでは、brk() によってデータセグメントの最終アドレスを得た後、roundup マクロでページ境界にそろえて、sbrk() によって SIZE バイト分（これはページサイズの倍数）データセグメントを拡張します。

その後、mmap() によって、マップしたいスペシャルファイルをマップします。



リアルタイムアプリケーションとメモリマップデバイス

ディスパッチレイテンシィの項で説明したように、read/write 等の入出力システムコールには、ジッターが伴います。

上記の問題を単独に解決しても、シミュレータに必要なプロセスを同時に起動して動作させるとなると、優先順位間のスイッチングや、割込み処理、キャッシュのミスヒット率の増加など動的特性は違ったものになりがちです。

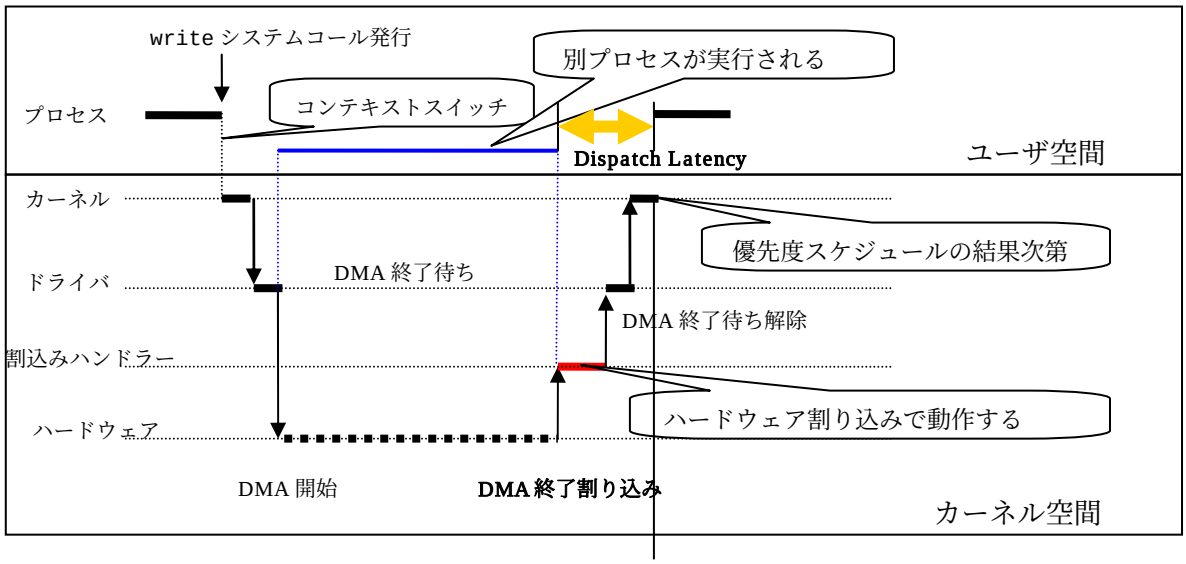
このジッターの原因は、システムコールを行う際に発生する、

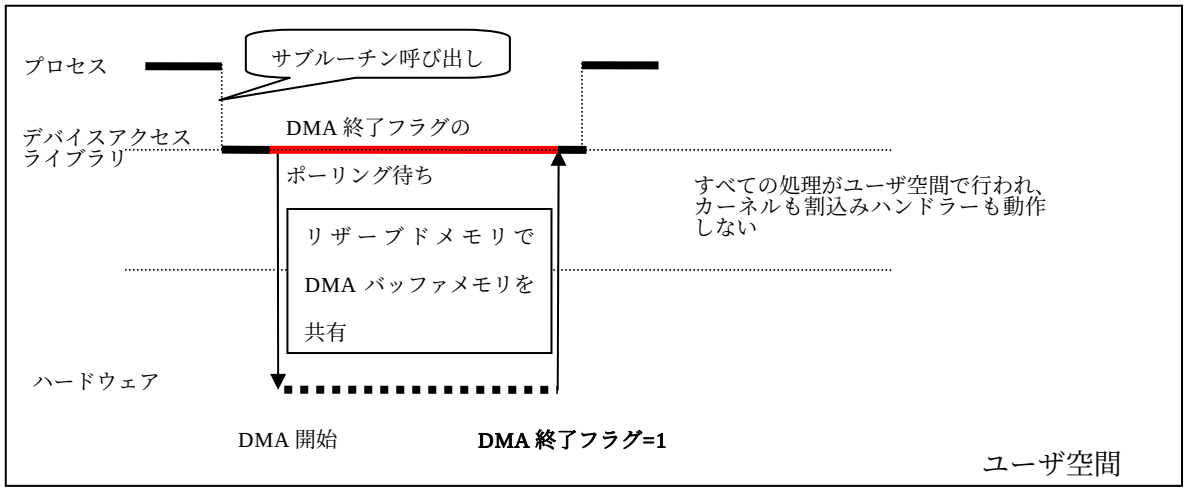
- **プロセススイッチ**
- **キャッシュのフラッシュ**
- **ディスパッチレイテンシィ**
- **カーネルによる優先度再スケジューリング**

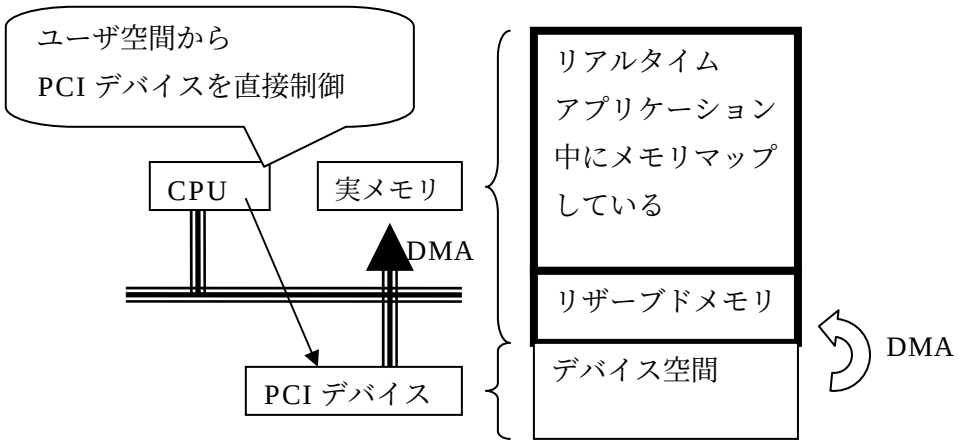
にあります。

プロセススイッチングは、実行プログラムが異なるため、コードやデータのキャッシュのミスヒットを伴うため、実行時間の変動原因になり、ディスクなどの read/write の場合には、i-node を更新する都度、ディスクをシークしなければならないため、最悪の場合では、ディスクのヘッドが移動する数ミリ秒ものむだ時間がかかってしまいます。

通常、100Hz 以上の閉ループ処理になると、カーネルによる再スケジュールを伴う read/write システムコールのジッターが大きいため、メモリマップをつかってアプリケーションプログラムが直接デバイスをアクセスしたほうが有利です。







上図のように、リアルタイムプロセスが、主記憶の一部をリザーブドメモリとしてメモリマップし、PCI デバイスもメモリマップします。

リアルタイムプロセスは、直接デバイス空間のレジスタをアクセスして DMA をセットアップします。この時に扱うアドレスは実アドレスのみなので、デバイスドライバのように、仮想空間－実アドレス変換や、DMA 領域のメモリロックなどの手順はすべて不要になり、メモリ空間のスキャッターギャザー問題も発生しないため、１回の DMA 転送ですべての DMA 領域にデータを転送できます。

デバイスドライバでは、ページ単位（４K バイト）の領域に分割し、ページ単位でしか DMA 転送できないため、かなりのオーバーヘッドがかかっています。

リアルタイムディスクアクセスの手法

しかし、ディスクへのロギングなどの場合、上記の手法は利用できません。
あまりにデバイスの制御が複雑であるからです。

また、ディスク転送の場合には、セクタ単位（512 バイト）でしかデータを扱えないため、カーネル内にディスクバッファを用意し、ユーザデータはこのバッファにコピーしているだけです。

この結果としてデータ転送が 2 回起こってしまいます。（ユーザデータ領域→カーネルバッファ→ディスクブロック）

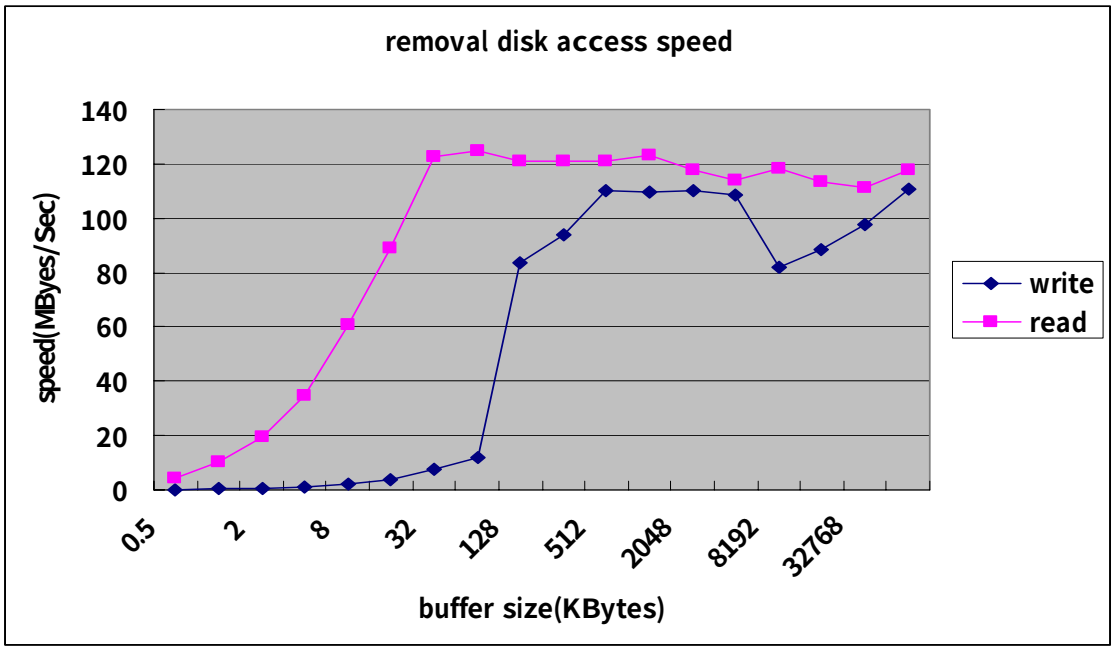
このような先読み後書き機構（read-ahead/write-behind）は、アクセスするデータが小さい場合はキャッシングの効果がありますが、データが大きい場合あきらかな無駄時間になり、入出力時間の予測を難しくします。

この問題を解決するためには、システムのバッファキャッシュの利用を禁止し、ユーザデータ領域とディスクブロックとの直接的なＤＭＡ転送を行う、直接ディスク転送を可能にしなくてはなりません。

POSIX1003.1b では、この連続領域ファイルに対する直接ディスク転送の機能は、ファイル作成時のモードの追加(O_DIRECT)のみで利用できます。これにより、安全でかつ最小の変更で連続領域ファイルを使用することが可能になります。

しかし、このダイレクトディスク転送は、先のリザーブドメモリ空間を転送元とすることは出来ないため、注意が必要です。

また、実際のディスク転送速度は、read/write のサイズが大きいほど高速で、低データサイズが小さいとその速度はあまりにも低速になります。



このことから、シミュレーション周期で発生するデータ量で書き出すと、ディスクの性能が出せないことがよくあります。

これを解決するためには、データをバッファリングし、別プロセスで書き出すテクニックが要求されます。

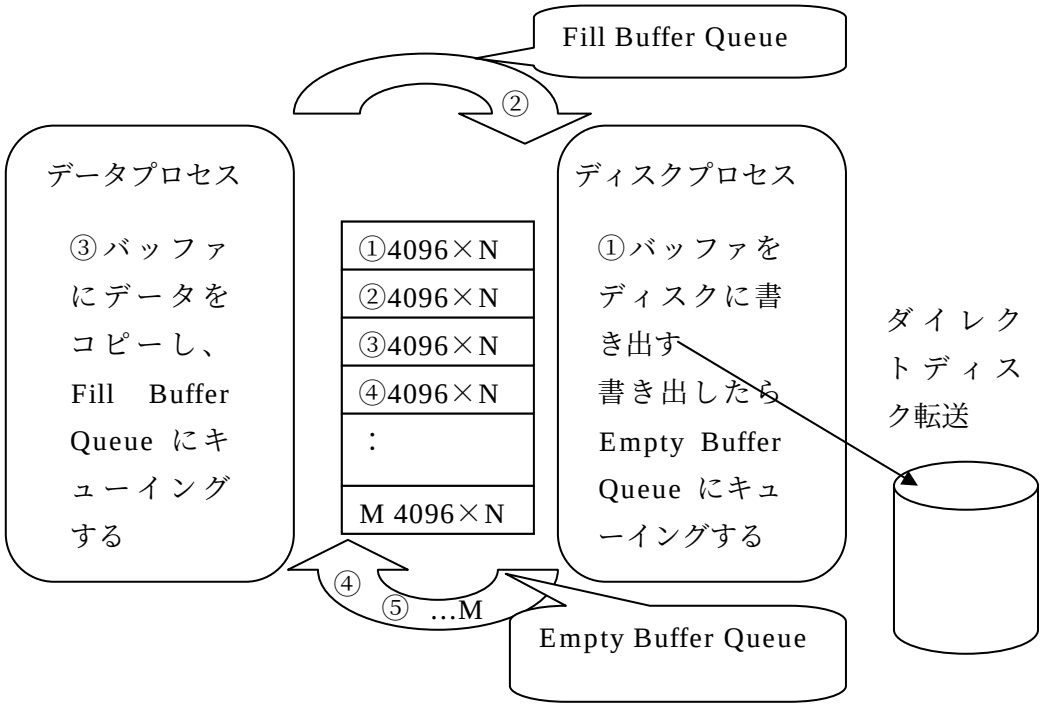
BufferSize(KB)	WriteSpeed(MB/sec)	ReadSpeed(MB/sec)
0.5	0.26	4.54
1	0.5	10.19
2	0.5	19.32
4	0.99	34.98
8	1.92	60.65
16	3.8	88.75
32	7.37	122.71
64	12.14	124.69
128	83.56	120.89
256	93.95	121.12
512	110.23	121.05
1024	109.48	122.94
2048	110.22	117.51
4096	108.7	114.12
8192	81.81	118.39
16384	88.55	113.22
32768	97.58	111.33
65536	110.75	117.74

別プロセスにする理由は、ディスクには DMA 転送が必須であるため、別の CPU に割り当てなければならないためとシミュレーション周期から独立させ、スケジューリングをまったく別に行う必要からです。

したがって、これらを満足するためには、共有メモリとメッセージキューを利用することになります。

メッセージのバッファをディスクバッファとして利用する方法もありますが、この方法にはコピーが伴うため、共有メモリを併用します。

このとき共有メモリは、ページ境界に割り付け、マルチバッファにし、そのインデックスを２つのメッセージキューを使って、リングバッファを構成します。



初期状態では、Empty Buffer Queue に未使用のバッファスロットをキューイングしておきます。

データプロセスは、この Empty Buffer Queue からバッファスロットを取り出し、データをコピーして Fill Buffer Queue にキューイングします。

ディスクプロセスは、この Fill Buffer Queue からバッファスロットを取り出し、データをディスクに書き出し Empty Buffer Queue にキューイングします。

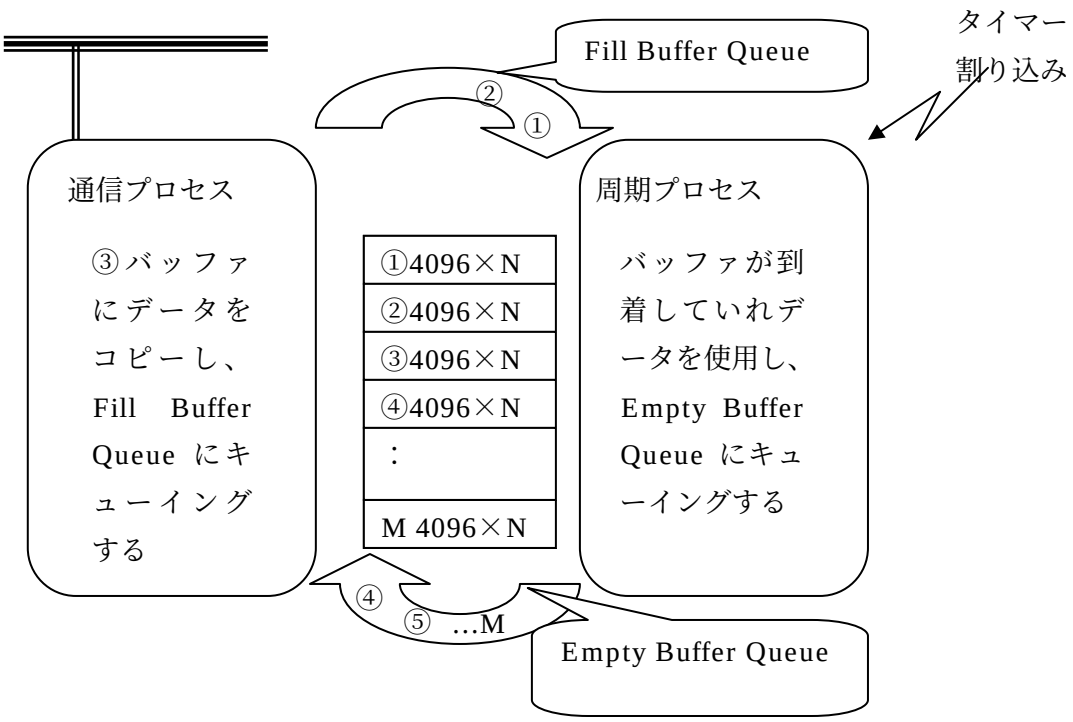
このバッファスロットは、１つの共有メモリで構成し、メモリロックをかければ、常にメモリ常駐で、ページバウンダリからページサイズの整数倍でディスクに書き出せるため、ダイレクトディスク転送を使用できます。

このとき、**”4096×N”**は、ディスク書き出し時間に、**データプロセスで発生するデータ量**をカバーできる大きさに設定し、バッファスロットの数 M は、ディスク書き出し時間のジッターを解消できる時間分のバッファ数以上（通常は５）に設定します。

さらに高速にデータ収録を行うためには、ディスクのフォーマット時にオプションを指定し、1 個の i-node サイズを**”4096×N”**に設定し、収録に使うすべてのファイルを予め０書き出しし作成しておくと良いでしょう。（mke2fs　-b 4096 -i サイズ -j /dev/sdb1 ）

これは、**1 つの i-node が 1 回のデータサイズで表現できる事と、**linux が遅延 i-node 割り当てを行っているため、実行時に i-node を割り当て、ファイルサイズを動的に変化させるとリアルタイム性能に影響を与えるためです

この手法は、ディスクだけではなく、TCP/IP 等でも利用できます。
この場合には、Fill Buffer Queue は非ブロックに設定しておくことが重要で、周期内に届いたデータだけを利用できます。

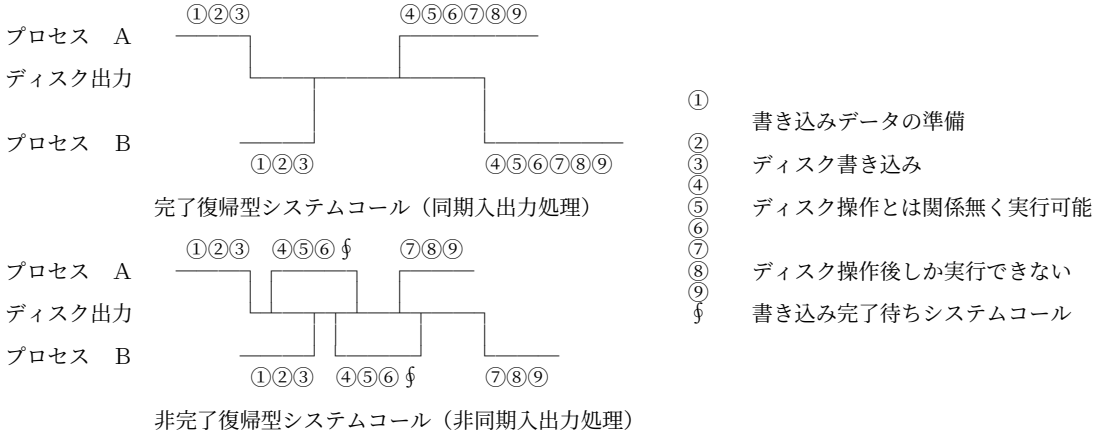


非同期入出力

ディスクへの read/write ジッターのもう一つの解決方法は非同期入出力命令を使用することです。Linix ではこの非同期入出力呼び出しは、スレッドを作成しそこでディスクへの入出力を行わせることとと同義ですが、プログラム作成者はそれを意識しなくてすみます。

read/write の問題は入出力操作のシステムコールが、ＵＮＩＸでは完了復帰型の設計になっているため呼びだしたプロセスが長時間ブロックされることです。ディスパッチレイテンシィの項で説明したように、複数のプロセスからのアクセスが競合した場合、最後にシステムコールを発行したプロセスは、すべての入出力動作が終了するまで封鎖され、貴重な時間を無為に費やす結果になります。

入出力処理が非同期に実行できる場合、図のように実際の入出力操作時間は同じでも処理全体の時間は短くなります。



同期と排他制御

マルチCPU内のプログラムによる共有データへのアクセスの同期と排他制御における最も効率的なメカニズムは、スピン･ロックを使用することです。しかし、スピン･ロックを保持している間に再スケジューリング変数を使用してプリエンプションに対してプロテクトすることなしに、ユーザ･レベルからスピン･ロックを使用することは安全ではありません。

もし移植性が効率性よりも重要であるなら､POSIX カウンティング･セマフォ及び mutex が、共有データへの同期化アクセスにおける次の最良の選択です。さらに、System V セマフォが提供されており、それによってプロセスがセマフォ値のやり取りを通して通信することができます。

一般的に３つのタイプのメカニズムが相互排他を提供するのに使用されます。
――ビジーウェイトを含むもの、スリープ待ちを含むもの、そしてプロセスがロックされているクリティカル･セクションへ入ろうと試みる時に 2 つの組合せを含むもの、です。

“スピン・ロック”としても知られている**”ビジーウェイト”**メカニズムは、ハードウェアでサポートされている ***Test & Set 命令***を使用します。

現在ロック状態にあるビジー待ちロックを、プロセスが取得しようとすれば、現在ロックを保持しているプロセスがそれを解除し、そして評価及びセット・オペレーションが成功するまで、ロッキング・プロセスは ***Test & Set 命令***を再トライし続けます。対照的に、セマフォのようなスリープ待ちメカニズムは、もしそれが現在ロック状態にあるロックを取得しようとすれば、プロセスを休眠状態（スリープ）へと移行させまず。

ロックを取得しようとするほとんどの試みが成功すれば、ビジーウェイトメカニズムはとても効率的です。これは、単純な ***Test & Set 命令***がビジーウェイトロックを取得するために必要な全てだからです。

リソースをプロテクトするのに必要な時間量が短い時に、ビジーウェイトメカニズムは適切です。これには 2 つの理由があります： 1）ロック保持時間が短い時、ロックされていない状態内でロッキング・プロセスではロックを見つけやすく、よってロック・メカニズムのオーバーヘッドも最小で、2）ロック保持時間が短い時、ロックの取得における遅延もまた短いからです。

ロックがアンロックになる間、ビジーウェイトメカニズムは CPU のリソースを無駄にするため、ビジーウェイト排他制御を使用する時、ロックの取得における遅延を短くすることが重要です。一般的なルールとして、ロックが保持されている時間が、2 つのコンテキスト・スイッチを実行する時間よりも短ければ、ビジーウェイトメカニズムが適切です。

“セマフォ”は相互排他を提供する他のメカニズムです。既にロックされているセマフォをロックしようとするプロセスは、ブロックされるかまたはスリープへと置かれるため、セマフォはスリープ待ちの排他制御の形式です。POSIX カウンティング・セマフォは、共有リソースへのアクセスを制御する移植性の良い方法を提供します。ロック及びアンロック・操作において、最速の動作を達成するために実行される単純なインターフェイスを、カウンティング・セマフォは提供します。

“Mutexes” は同じリソースを共有するプログラムで、しかし同時にアクセス出来ない多数のスレッドを許します。 mutex が作られ、そして、リソースを使う時に、リソースを必要とするスレッドが他のスレッドから mutex をロックして、それがもう必要とされないとき、それをアンロックしなくてはなりません。 POSIX mutexes は、特にリアルタイムアプリケーションのために有用な per-mutex 基礎の上に個々に構成可能な：robust(強靭な) mutexes とプライオリティインヘリタンス mutexes ２種類の関数を持っています。 もしアプリケーションのスレッドの１つが、 mutex を持つ間に、死ぬなら、ロバストネスがアプリケーションに回復するチャンスを与えます。 **プライオリティインヘリタンス** mutex

を使っているアプリケーションは、時折引き上げられた mutex の所有者のプライオリティをになります。

プライオリティインヘリタンス（優先度継承）

スリープ待ちの相互排他メカニズムとして使用されているセマフォは、プライオリティ反転を生み出すことがあります。プライオリティ反転は、1 つかそれ以上の低プライオリティ・プロセスの実行が、クリティカルセクションで 1 つまたはそれ以上の高プライオリティ・プロセスの進行を止まらせる時に起こります。

プライオリティ継承は、クリティカル･セクションでの低プライオリティ・プロセスの実行を一時的に、最高プライオリティ待ちプロセスへと上げることを伴います。

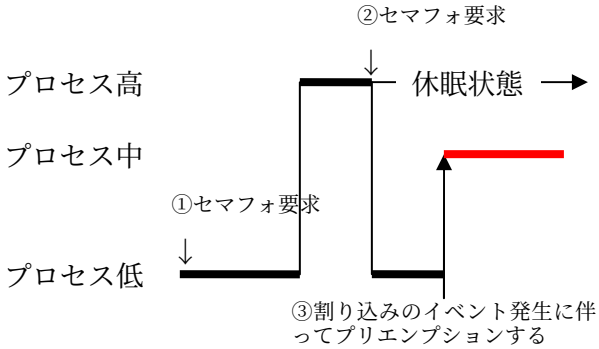
これは、クリティカル･セクションで実行しているプロセスが、クリティカル･セクションから離れるまで実行を続けるのに充分なプライオリティを持つことを確実にします。
詳細は RedHawk User Guide の５章を参照してください

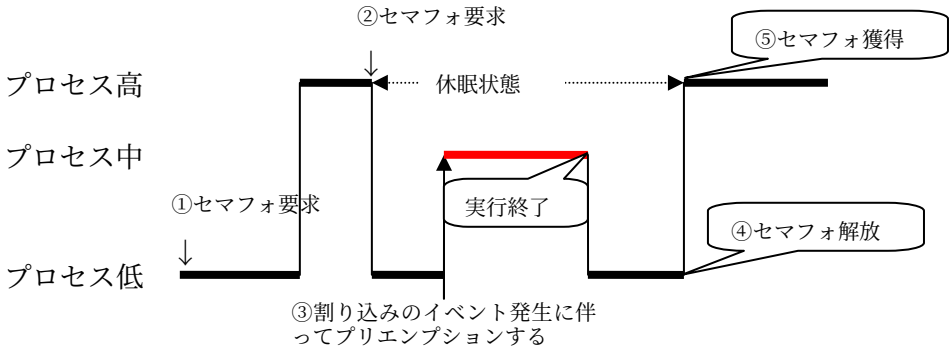
下図に示すように今、プロセス高とプロセス低の間で、単純なセマフォでクリティカルセクションの排他制御を行っているとします。

セマフォには、優先度は無関係であるため、
①で、**”プロセス低”**がセマフォを獲得し、実行している状態で、優先度の高い**”プロセス高”**が実行状態になったとします。
②で、”プロセス高”は、同じセマフォを要求しますが、”プロセス低”が既にセマフォを獲得しているため、”プロセス高”は休眠状態になり、この時、カーネルは、”プロセス低”を実行させます。

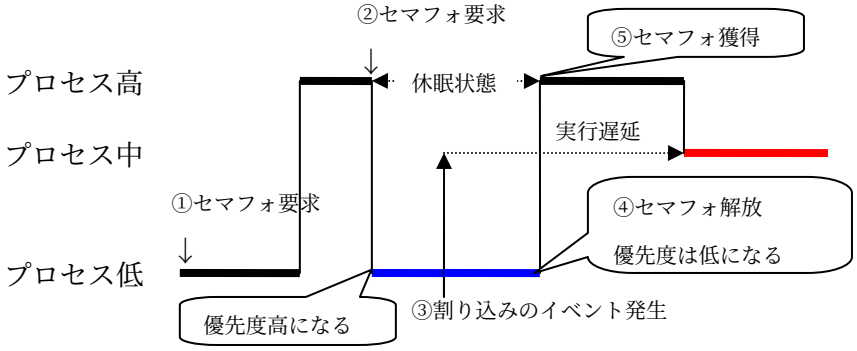
このタイミングで、**”プロセス高”**、**”プロセス低”**の獲得しているセマフォに無関係の”プロセス中”が起動した場合、プライオリティインバージョンが発生します。

“プロセス中”は、**”プロセス低”**に割り込む形でプリエンプションし、**”プロセス中”**が終了するまで、**”プロセス高”**も、**”プロセス低”**も実行されません。
その後、**”プロセス中”**が終了すると、“プロセス低”が実行を終え、セマフォを解放し、**”プロセス高”**が実行できます。





この現象を解決するためには、**”プロセス低”**が**”プロセス高”**の優先度を持つことが必要です。そうすると、③の状態でプリエンプションすることはありません。



すると、**”プロセス低”**が連続して実行され、**”プロセス高”**の終了を待って、**”プロセス中”**が実行されます。

リアルタイムシステム環境設定

- バックグランド･プロセスをブート CPU に集中させる
  (run –g0 –b0 コマンドを実行し、shield -p 1,2,3… を実行する)
- IRQ 割り込みからのシールディング(shield –i 1,2,3… を実行する)
- 必要であれば、ローカルな割り込みからのシールディング(shield –l 1,2,3… を実行する)

  ローカル･タイマーの禁止は、POSIX タイマーや nanosleep,ユーザシステム時間アカウント及びラウンド･ロビン･クワンタム満了の禁止のような、いくつかの副作用を引き起こします。

リアルタイムプログラム作成手順

(1) リアルタイム優先度とスケジュール(通常は SCHED_FIFO)を決定する。

(2) 実行 CPU を指定する。(ブート CPU 以外)

実 CPU とシブリング CPU の違いに注意！(キャッシュを共有している)

(3) メモリをロックする。

(4) 必要なデバイス、ファイルメッセージキューをオープンする。

DMA デバイスと PIO デバイスを意識する。

(5) 共有メモリやローカルメモリの初期化を行う。
メモリをロックした後、0クリアを行うことで、キャッシュに乗せる効果がある。
メモリ割付は、malloc()ではなく、memalign()を使用することで、メモリのアライメントを意識する。
通常は、ページバウンダリから割り付けるのが有効、例外はキャッシュの TLB 変動を嫌う場合。

(6) シグナルマスクを設定する。

(7) プロセス／スレッドを生成する。
生成後、シグナルマスクを再設定する。実行 CPU も再指定する。

一般的な注意

1. **リアルタイムプロセスでのビジーウェイトはシステムをハングアップさせます。必ず、nanosleep()を挿入すること。**
2. **メッセージキューや共有メモリには、リソースリミットがあります。使用前にチェックすること。**
3. **root 以外のユーザでは、命令がリアルタイム命令が自由に実行できません。特権を必ず設定すること。**
4. **優先度の異なるプロセスの間で、排他制御を行う場合には、優先度反転現象を考慮してください。**
   優先度反転を防ぐためには、プライオリティインヘリタンスを供給する POSIX pthread 拡張関数を使用してください。

用語集

ビジーウェイト

ハードウェア・サポートの評価及びセット・オペレーションを使用してロックを取得する排他制御の方法です。もしも現在ロック状態にあるビジー待ちロックを取得しようとプロセスが試みれば、現在ロックを保持しているプロセスがそれを解除して、そして評価とセット・オペレーションが成功するまで、ロッキング・プロセスは評価及びセット・オペレーションを再トライし続けます。スピン・ロックとしても知られています。

コンテキスト・スイッチ

マルチ・タスク・オペレーティングシステムが 1 つのプロセスの稼動を停止し、そして他の稼動を開始する時。

クリティカルセクション

ソフトウェアの正確なオペレーションを保証するために、連続でそして割り込みが無く実行されなければならないインストラクションのシーケンス。

デッドロック

2 つまたはそれ以上のプロセスが、それら両方が一方が何かのソースをリリースするのを待っているために進行しない状況の数。

遅延されている割り込み扱い

割り込みレベルで行われていた処理を遅らせる割り込みルーチンでの方法。RedHawk Linux は *softirq*、*tasklet*、及び *work queue* をサポートし、それらはカーネル・デーモンのコンテキスト内で実行します。これらのデーモンのプライオリティ及びスケジューリング・ポリシーは、高プライオリティ・*リアルタイム*・タスクが遅延されている割り込み機能の動作を*プリエンプト*することができるようにコンフィグすることができます。

決定性（デターミニズム）

一定の時間量内で特定のコード・パス（連続で実行されるインストラクションのセット）を実行できるコンピュータ・システムの能力。コード・パスにおける実行時間が 1 つのインスタンスから他のものにおいて変化する量が、システム内の決定性の度合いを示します。ユーザアプリケーションのタイム・クリティカル部分を実行するのに要される時間量とカーネル内のシステム・コードを実行するのに要される時間量の両方に、

決定性は適用されます。

ダイレクト入出力(Direct IO)

カーネルのデータバッファリングをバイパスする IO のアンバッファ形式。直接 IO では、ファイル・システムはディスクとユーザバッファの間を直接データ転送します。

ハイパー・スレッディング

Intel Pentium XeonCPU の機能で、それは単一の物理 CPU がソフトウェア・アプリケーションの複数スレッドを同時に走らせることを可能にします。1 つのセットの CPU 実行リソースを共有しつつ、それぞれの CPU は 2 つのアーキテクチャ状態のセットを持ちます。それぞれのアーキテクチャ状態は、システム内で 2 倍の論理 CPU になった論理 CPU として考えることができます。ハイパー・スレッディングが許可されている 1 つの CPU・システムは 2 つの論理 CPU を持ち、割り込み及びバックグランド・プロセスからそれらの 1 つを*シールド*することを可能にします。RedHawk Linux ではハイパー・スレッディングはデフォルトで許可されています。

ジッター

周期的処理の終了時または開始時における時間の変動。コード・セグメントの実行または割り込みへの応答のいずれかにおいて計測される最悪ケース時間が一般的ケースよりも大きく異なる時、アプリケーションのパフォーマンスはジッターを経験していると言われます。アクション全てが正確な周期内にとどまっている限り、ジッターは通常問題を起こしません。しかしジッターが可能な限り最小であることをリアルタイム・タスクは一般的に要求します。

メッセージキュー

1 つまたはそれ以上のプロセスが 1 つまたはそれ以上のリーディング･プロセスによってリードされるメッセージをライトすることを可能にするプロセス間通信（IPC）メカニズムです。RedHawk Linux は、POSIX 及び System V メッセージキュー機能を持っています。

排他制御

共有リソースへのアクセスを順番に並べることによって、いっときにクリティカルセクションでは、共同プロセスのセットの 1 つのみが実行することができることを確実にするメカニズム。3 つのタイプのメカニズムが一般的に相互排他を提供するのに使用されます。ビジー待ちを含むもの、スリープ待ちを含むもの、そしてその 2 つの組合せです。

プリエンプション

CPU 上を走っていたプロセスがより高いプライオリティでのプロセスによって置き換えられる時のことです。RedHawk Linux 内に含まれているカーネル・プリエンプションは、カーネル空間内で動作していても、より低いプライオリティ・プロセスのプリエンプトを可能にし、向上されたシステム応答の結果を生みます。プロセス・プリエンプションは再スケジューリング変数の使用を通して制御することができます。

プライオリティ継承

プライオリティ反転を回避するために必要とされる、1 つのプロセスのプライオリティと共に他へ一時的に渡すメカニズムです。

プライオリティ反転

より高いプライオリティ・プロセスがより低いプライオリティ・プロセスの実行待ちを強制される状態です。

権限

それを通してユーザまたはプロセスがオペレーションを動作する、またはシステム制約を無効にすることを許されるメカニズムです。スーパーユーザは全ての（ルート）権限を持ちます。機能を通して、個々のユーザ及びプロセスにおいて権限は許可または禁止することができます.

プロセス

実行されているプログラムのインスタンスです。それぞれのプロセスはユニークな PID を持ち、それはカーネルのプロセス・テーブル内のそのプロセスのエントリです。

プロセスディスパッチレイテンシィ

割り込みによって示される外的イベントの発生から、外的イベントを待っているプロセスがその最初のインストラクションをユーザ・モードで実行するまでに経過する時間です。

セマフォ

その値が 1 プロセスのみで評価されそして設定されるメモリ内の変数。既にロックされているセマフォをロックしようと試みるプロセスはブロックされるかまたはスリープへと置かれるので、セマフォはスリープ待ち相互排他メソッドです。RedHawk Linux は最速のパフォーマンスを達成するための単純なインターフェイスを与える POSIX カウンティング・セマフォ、及び多くの追加機能を与える *System V* セマフォ

を提供します

共有メモリ

1 つ以上のプロセスのバーチャル・アドレス・マップを通してアクセス可能なメモリ。一般的なオペレーティング・システム・サービスを使用しての read/write よりも速く、共有メモリを使用してプロセスがデータをやり取りすることができます。*POSIX* と同様に System V から派生した標準の共有メモリ・インターフェイスを RedHawk Linux も持っています。

スリープ待ち

プロセスが、もしもそれが現在ロックされている状態のロックを取得しようと試みている場合にスリープ状態へと移行する、*セマフォ*のような相互排他の方法です。

SMP

対称並列処理。1 つのオペレーティング・システムに管理される 2 つまたはそれ以上の CPU を使用する計算の方法で、しばしば同じメモリを共有し、そして入力/出力デバイスへの同等のアクセスを持ちます。システム内のいくつかのまたは全ての CPU 上をアプリケーション・プログラムが走ります。

softirq

それによって割り込みルーチンの実行を次の利用可能な“安全ポイント”まで遅らせることができる方法。割り込みルーチンを起動する代わりに、次の安全ポイントでの起動を引き起こす“トリガー”が代わりに使用されます。安全ポイントは、カーネルがハードウェアまたはソフトウェア割り込みをサービスしておらず、そしてブロックされている割り込みで稼動していない時間です。

スピン・ロック

リソースにおいて相互排他を確実にするビジー待ち方法。スピン・ロック上で待っているタスクは、スピン・ロックが利用可能になるまでビジー・ループ内にとどまります。

System V

Linux 及び System V システムを含む多くの UNIX 系システムによってサポートされているプロセス間通信（IPC）オブジェクトにおける標準。System V IPC オブジェクトには 3 つの種類があります。System V メッセージキュー、セマフォ・セット、そして共有メモリ・セグメントです。

tasklet

ユーザ空間への戻り、またはハードウェア割り込みの後で、ソフトウェア割り込みが受信される時に走っているソフトウェア割り込みルーチン。tasklet はマルチ CPU 上では、同時に走りませんが、しかし動的に割り当て可能です。

workqueue

softirq 及び *tasklet* に加えて、実行を遅らせる方法。しかしこれらの方法と異なり、Linux はカーネル・デーモンのプロセス・コンテキスト内で work キューを処理し、スリープすることが可能です。