

ウインドリバー Real-Time Core アプリケーション開発の優良事例

ウインドリバー システムズ エンジニアリングR&D Michael Barabanov

目次

はじめに	1
要求事項の分析と実現性評価	1
デザイン	2
パーティショニング	2
プロセス間通信	2
スケジューリング	3
レート モノトニック スケジューリング	3
周期駆動	3
同期	3
メモリ割り当て	3
実装	4
POSIXスレッド	4
割り込み	4
ソフト割り込み	5
デバイスドライバ	5
同期と一方向キュー	5
デバッグ	6
注記	6

はじめに

Wind RiverのReal-Time Core for Linuxは、Wind River Linuxを最も優先度の低いタスクとして実行させるリアルタイム実行オブジェクトです。この「デュアルカーネル」デザインはリアルタイム動作と非リアルタイム動作を分離可能にし、これによりハード リアルタイム コンポーネントの予測性を向上させ、Linuxの効率を維持します。Wind Riverの Real-Time Core for LinuxはLinuxカーネルのアドレス空間で実行され、リアルタイム アプリケーションはロード可能なカーネル モジュールの形で実装されます*1 (ただし、これはユーザーに対してほとんど透過的に行われます)。

PREEMPT_RTなどLinuxにおけるリアルタイムへの別のアプローチに比べ、Real-Time Coreは優れたハードリアルタイム性能を実現。これは、比較的小さいサイズでありながら厳重にコントロールされたコードベースであることと、特にハードリアルタイム用に設計開発されたという利点によるものです。

Real-Time Coreはよく知られているPOSIXに似たAPIをサポートしているので、既存のアプリケーション コードを迅速にハードリアルタイム環境に移植して実行することができます。極めて効率的な高速ハードリアルタイム・システムの実装を容易にする多数の固有機能と同様に、標準的なスレディング メカニズムとプロセス間通信 (Inter Process Communication: IPC) を使用することができます。

要求事項の分析と実現性評価

Real-Time Coreアプリケーションを開発するには、ユーザ・インタビュー、要求ワークショップ、ユースケースなど、リアルタイム要求を収集するためのあらゆる標準的手法が適用できます。リアルタイムシステムにおいては、ハードリアルタイム性を必要とする機能とそのレベルを確認することが極めて重要です。これに関する検討事項として、以下のようなものがあります。

システムが適切なタイミングでサービスを実行する必要がある外部イベントとデバイスには、どのようなものがあるか？ これらのイベントにはどの程度の速度で対応する必要があるか？ これらのイベントに対しては、どのようなシステム動作が求められるか？ これらのデバイスに対するサンプリングは、どの程度の頻度で行うか？

ソフトウェアのみのソリューションは使用できない場合があります。アプリケーションの中には、特定のハードウェアプラットフォーム上で、ソフトウェアでは実現できない応答時間を必要とするものもあります。このような場合は、より適切なプラットフォームを選ぶか、既存のハードウェアに特殊目的のハードウェアを追加しなければなりません。Real-Time Coreを使用すれば、バッファリング要求を減らして、追加ハードウェアの必要性を最小限に抑えることができます。稀な例として、OSにカスタム キャッシュやTLB (Translation Lookaside Buffer) 割り当てといった変更を加えなければ、デッドライン要求を満たせないという場合もあります。これは、実装に伴うコストとデバッグ作業量の増大を招く恐れがあります (Real-Time Coreはユーザーが実行できる測定環境を備えています。これを使用すれば、特定ターゲットシステム上で実現可能なリアルタイム性能を評価できます)。

リアルタイム・デッドラインに加えて、必要なドライバの入手の可能

性と複雑度も考慮に入れる必要があります。Real-Time Core用に汎用的で複雑なドライバを実装することは可能ですが、そのためのコストと時間が極端に大きくなってしまいます。幸いなことに、ほとんどの場合、複雑な操作（たとえばSCSIディスク上にあるNTFSファイル・システムへの書き込み）は、Linuxのプロセスに代行させることができます。気をつけなければならない点は、Real-Time Coreがリアルタイム化に関するすべての問題を適切に解決できるわけではないこと、そして、そのような場合には考慮すべき代替手段が存在するという事です。たとえば、システム負荷にかかわらず、時間に関するデッドラインの要求を満たすことがそれほど重要でない場合、PREEMPT_RTのような他の「ソフト リアルタイム」ソリューションの方が実装しやすいことがあります。ウインドリバーは、Real-Time CoreとPREEMPT_RTの両方をサポートしているので、あらゆるリアルタイム アプリケーションのニーズを満足する広範なソリューションを提供することができます。

デザイン

非リアルタイムシステム用に開発されたデザインパターンであっても、その多くはReal-Time Coreアプリケーションに同じように適用できます。また、リアルタイムシステム専用開発されたデザインパターンも存在します^{*2}。この項では、いくつかの一般的なリアルタイム デザインパターンと、特にReal-Time Coreアプリケーションの開発者が関心を持つデザインパターンについて述べます。

パーティションニング

Real-Time Coreパラダイムでは、システムをリアルタイム コンポーネントと非リアルタイム コンポーネントに分割することに重点が置かれています。リアルタイム コンポーネントはReal-Time Coreによる制御の下に実行され、非リアルタイム機能はLinux上で実行されます。したがって、リアルタイム コンポーネントをできるだけ小さくして、できるだけ多くの機能をLinuxに受け持たせることが極めて重要です。コンポーネントが小さくなれば、デザイン、解析、評価も容易になります。リアルタイム コンポーネントには、論理的に正確であることに加えて、時間の点で正確であることが求められるので、リアルタイム コンポーネントを「贅肉のない」状態に保つことは、実装、デバッグ、サポートのコストを下げることになります。

大まかに言って、グラフィカル ユーザ インタフェース (GUI)、ネットワーク、ファイルシステム、データベース アクセスをLinuxプロセスに受け持たせ、適切なタイミングで処理することが求められる通信やデバイスの初期データ処理をReal-Time Core内で行うことが必要です。もちろん、このルールにも例外はあります。フライトシミュレーションで使用するような特定のGUIでは、保証された頻度でのスクリーン更新が必要とされることがあるので、これらのGUIは、システムのリアルタイム コンポーネント内に存在しなければなりません。リアルタイム パケットルータにも同じルールが当てはまります。一方、デッドラインを満たすことがそれほど重要でない（ソフト）リアルタイム動作もあるので、このようなものについては、代わりにLinuxのユーザ空間で実行することができます。後者の代表的な例がオーディオ/ビデオ処理です。この場合、優先度を逆転させないように注意する必要があります。たとえば、優先度の高いリアルタイムタスクが、優先度の低いタスクから、さらに悪い場合Linuxプロセスからのデータを待つようなことがあってはなりません。

アナログデジタル変換器の例を考えます。この例では、データをファイルに記録するほか、スコープGUIにそのデータを表示しなければなりません。データ収集カード内のハードウェアバッファは、10秒間のバッファリングが可能です。この例では、Linuxによる読み込みプロセスの遅延が10秒を超えず、データ記録速度に追従できるのであれば、おそらくReal-Time Coreは必要ありません。これに対

し、ハードウェアのバッファ可能量が100マイクロ秒分だけだとすると、前述の方法では問題が生じます。バッファアンダーランが発生しないことを保証できなくなるからです。ここでReal-Time Coreが必要となります。ファイルI/O（入出力）とGUIはLinuxに受け持たせたままとしますが（リアルタイム部分をできるだけ小さくする必要があります）、DAQカードポーリングはリアルタイムドメインに移します。最後に、データ転送用にリアルタイムFIFO（First-In, First-Out：先入れ先出し）バッファを設定します。これによって、効率的で分かりやすいシステムを実現することができます。

多くの場合、ハードウェア バッファリングを行ってもリアルタイム プラットフォームの必要性がなくなることはありません。たとえば、閉ループ制御システムには通常、システムが安定した状態を保つために必要な最大フィードバック遅延があります。ほとんどのシステムにおいて、この遅延は標準的なLinuxの能力で対応できる値よりもはるかに小さな値なので、制御システムの入出力を含めた全体的な制御アルゴリズムは、Real-Time Coreドメイン内に置く必要があります。このような場合、オペレータがシステムの現状を確認するためのGUIの提供にはLinuxプロセスが使われます。

プロセス間通信

重要性の低い場合は別として、Real-Time Coreアプリケーションでは、リアルタイムタスクと非リアルタイムLinuxプロセスの間に通信を設定する必要があります。Real-Time Coreは、主に2つの方法、RT-FIFOと共有メモリによってこの通信を行います。RT-FIFOは、一方向または双方向のストリーム指向通信チャンネルです。それぞれのRT-FIFOには2つのエンドポイントがあり、一方がリアルタイム側、もう一方がLinuxユーザ空間アクセス用です。RT-FIFOは、Linuxのファイルシステム内では通常デバイスとして扱われます。RT-FIFOへのアクセスには、どちら側の場合でも標準のPOSIX open/read/write/ioctlインタフェースが使われます。RT-FIFOはメッセージサービスを提供しませんが、簡単に実装できます。

優先度逆転の可能性を減らすために、RT-FIFOの動作はリアルタイム空間をブロックしないようになっています。しかし、読み取り時にFIFO内に十分なデータがない場合、または書き込み時にFIFOに十分な空きスペースがない場合は、呼び出し元のリアルタイム スレッドにはエラーが返されます。Linuxプロセスは、POSIXのopenフラグを使用することにより、ブロッキング動作と非ブロッキング動作のどちらかを選ぶことができます。RT-FIFOはストリーム指向のデータに適しています。つまり、センサからのデータは継続的にサンプルを生成し、このサンプルは、ローカルで保存するか、ネットワークを介して別のマシンに転送する必要があります。RT-FIFOはまた、さまざまなシステムパーツ間や信号イベント間の制御チャンネルとしても使用できます。

より特別なRT-FIFOの使い方としては、Linuxキャラクタ デバイスドライバのエミュレーションがあります。通常、このシナリオにおけるRT-FIFOは双方向モードで使われ、カスタムioctl呼び出しはリアルタイム側に対応付けられます。この方法は、GNUデバッグスタブを介してアプリケーションデバッグ用のシリアルデバイスをエミュレートするために、Real-Time Coreシステム自体の中で使われます。その他の使用方法としては、従来のアプリケーションとのインタフェースがあります。たとえば、携帯電話スタックでは、エミュレートされるシリアル・ポートを介したATコマンドインタフェースを提供します。

共有メモリは、RT-FIFOのシリアル特性では不十分な場合に使用できるもう1つのIPC方法。この方法ではデータコピーの量も少なくなります。デバイスメモリをユーザ空間に直接マップできる場合もあります。この場合は、リアルタイムシステムである程度の事前処理をしてからLinuxにデータを渡し、余分なメモリコピーを回避。開発中

に共有メモリを扱う際は、競合状態に注意する必要があります。通常は、データが入れ替わってしまうのを防ぐために、何らかの形で同期を取らなければなりません。アーキテクチャによっては、ユーザ空間とReal-Time Core (カーネル) 空間の両方からアトミック命令を使用することができます。その他の場合は、より高くつくシステム呼び出しベースの同期命令を使用する必要があります。

多くの場合、共有メモリとRT-FIFOを組み合わせることは有効な方法です。先に述べたデータ収集の例を続けます。今度は、デバイスからのサンプルが大きすぎて、RT-FIFOを効率的に通過することができないとします。この場合は、それぞれが最大のデータサンプルを保持できる共有メモリアreaにデータブロックをプールのすることができます。RT-FIFOは2個使用されます。一方はLinuxの読み込みシステムが使用する有効データを含むブロックのオフセットをキューに入れるために、もう一方はLinuxプロセスがどのバッファを読み込んだかをリアルタイム側に知らせるために使用されます。この種のプールデータ手法(ダブルバッファリングとも呼ばれる)は、リアルタイムドメインとLinuxのユーザ空間の間で大量のデータを転送する際に極めて有効です。

スケジューリング

リアルタイムスケジューリングの目的は、すべてのリアルタイム タスクがそのデッドラインを確実に満足できるようにすることです。タイムシェアリング システムとは異なり、一般にリアルタイム スケジューラは公平性を保証しようとはしません。これまでさまざまなリアルタイム スケジューリング アルゴリズムがデザインされてきましたが、そのトレードオフは通常、CPUの利用効率向上と、予測性および解析の容易さとの間で行われてきました。

Real-Time Coreは、優先度に基づくプリエンティブ スケジューリングを実装します。各スレッドには優先度が割り当てられます(優先度は実行時に変更可能)。優先度の高いスレッドの準備が完了するごとに、その時点で実行中のタスクが中断されて新しいタスクが実行されます。マルチプロセッサ システムでは、各CPUは、スケジューリングのため個別に扱われます。優先度ベースのプリエンティブ スケジューリングを使用し、アプリケーション自体によって他のスケジューリング アルゴリズムを実装することもできます。スケジューラは、Linuxを最も優先度の低いリアルタイム タスクとして扱います。したがってLinuxは、リアルタイム システムが処理するものが何もない場合のみ実行されます。このことは、Linuxカーネルの枯渇など予想しない事態を招くことがあります。While (1)と同じ動作を行うリアルタイム タスクは一見したところ問題はなく、タイムシェアリング システムでは問題なく実行することができます。しかし、Real-Time Core内ではLinuxが実行されなくなります。これは、ユーザにはシステムがハングアップしたように見えます。Real-Time Coreはソフトウェア ウォッチドッグを備えており、Linuxに変化がない場合はすべてのリアルタイム スレッドを停止します。概算ですが、リアルタイム タスクが約1秒を超えて連続的にCPUを占有することはできません^{*3}。

レート モノトニック スケジューリング

周期的タスクの場合、優先度を割り当てる自然な方法は、レートモノトニックスケジューリング アルゴリズムによるものです^{*4}。このアルゴリズムによれば、周期の短いタスクほど高い優先度が与えられます。レート モノトニック アルゴリズムによってスケジューリングされるn個の独立した周期的タスクのセットに関しては、以下の条件が満たされればすべてのデッドラインが保証されます。ここでCiはタスクiのワーストケースの実行時間、Tiはタスクiの周期です。

$$\frac{C1}{T1} + \frac{C2}{T2} + \dots + \frac{Cn}{Tn} \leq n(2^{\frac{1}{n}} - 1)$$

散発的なタスクは、多くの場合、優先度割り当て時に周期的タスクとして扱うことができます^{*5}。

周期駆動

ユーザ実装スケジューラの例として、リアルタイム スケジューリングでよく使われる周期駆動型(Cyclic Executive) スケジューラを考えます。これは本質的にテーブル駆動型のスケジューラで、各周期に対して実行すべきリアルタイム タスクがあらかじめ定められています。Real-Time Coreでは、ユーザ タスク ループを作成し、対応する周期で行うべき動作を実行する前に個々のセマフォ上で待つことによってこれを実現します。この場合、より高い優先度を持つマスタースレッドがセマフォに通知し、要求されたスケジュールに従ってタスクを起動します。マスタータスクはスーパーバイザとしての役割も果たします。これには、タスクがその周期を超えないようにする、各周期の一定部分をLinuxに予約する、時間を計上するといった動作が含まれます。

同期

Real-Time Coreは、豊富な同期プリミティブのセットを提供します。これらには、POSIXスピンロック、セマフォ、ミューテックス、条件変数、ならびにカスタム同期メカニズムの実装に使用できる低水準アトミック命令が含まれています。リアルタイムシステムでは、同期の使用によって、分かりにくい依存性やデッドロックを生じさせることがあるので注意が必要です。

ブロッキング命令は容易に優先度の逆転につながるがあるので、特に危険です。よく見られる優先度逆転は、優先度の低いタスクがセマフォを取得することによって、優先度の高いタスクがブロックされるというケースです。セマフォを取得したより優先度の低いタスクに中間優先度のタスクが割り込むと、この状況はさらに悪化します。極端な場合は、結局、最も優先度の高いタスクが、システム内にある優先度の低いすべてのタスクを待つことになってしまいます。優先度逆転の問題を解決する1つの方法は、POSIXのミューテックス優先度制限プロトコルを使用することです。このプロトコルは、対応するミューテックスが取得されると、固定された値まで優先度を上げます。より一般的なもう1つの方法は、十分に高い優先度で実行されるサーバスレッドを使用することです。

可能であれば、ブロックの可能性のある同期プリミティブを避けることが望まれます。たとえば、クリティカルセクションが非常に短い場合は、スピンロックを使ってクリティカル・セクションへの割り込みの可能性を完全に排除する方がより効率的です。ビットのテストやセット、あるいは整数変数のインクリメントといった一定の単純処理に対して、Real-Time Coreは、ロックされ、特定のターゲットハードウェア用に最適化されたアトミック命令を提供しています。この種のロックフリー通信はシステムの効率を向上させるとともに、解析をより容易にします。

場合によっては、複数のリアルタイムタスクを1つにまとめることによって、リアルタイムタスクのグループ内での同期を避けることができます。もう1つの方法は、同じリソースに対する競合を減らすことができるように、スレッド構造とデータ構造をCPUごとのグループに分割することです。

メモリ割り当て

動的メモリ割り当ては、プログラムのフロー制御が直接反映されていないようなライフスパンを持つオブジェクトを扱うための非常に強力な技術です。しかし、リアルタイムシステムでは、これが予測し得ない事態を招くこともあります。前項に示した優先度逆転と同じように、mallocやfreeなどによって共有プールから割り当てを行うと暗黙的な依存関係が生じ、展開フェーズに至るまで露見しないよ

うな不具合が発生する恐れがあります。低優先度のタスクが使用可能メモリをすべて使い果たしてしまった状態で、バッファ割り当てを必要としている高優先度のタスクに割り込まれた場合を考えてみましょう。もう1つの複雑化要因は、プールの断片化のためにメモリ割り当て要求を満足できなくなる恐れがあるということです。ソフトウェア エンジニアリングの観点から、共有プールを使用することはモジュール性に反することになります。1つのモジュールの動作が別のモジュールに依存するようになるからです。

割り当て要求の拒否や遅延を認めることができないシステムでは、アプリケーション自体の中で、あらかじめ割り当てられたバッファから動的割り当てを行う必要があります。断片化を避けるには、配列実装と空きスペースストレスのための連結リストを使うことができます。この実装では、割り当てと割り当て解除の両方が、一定時間内に完了します。割り当てられるオブジェクトのサイズが大きく異なる場合は、この種のプールが複数使われることもあります。モジュール性を確保するには、モジュールごとにプールセットを使用することが望まれます。目的は、任意の時点におけるモジュールの最大メモリ要求が、そのプール内の使用可能メモリ量を超えないようにして、すべての要求が遅滞なく満足されるようにすることです。それでも、一定の条件下では一般的な動的メモリ割り当てが必要となることがあり、実際にReal-Time Coreは、よく使われるmalloc機能やfree機能を備えています。アロケータは、断片化に影響されないスマートプール管理を使用しています。

実装

この項では、プログラマがシステムデザインを実装するためにReal-Time Coreから提供される機能について考えます。詳細な検討内容については、「Wind River Real-Time Core Programmer's Guide」(Wind River Real-Time Coreプログラマーズガイド)^{*6}を参照してください。

POSIXスレッド

リアルタイム マルチタスキングは、POSIXスレッド インタフェースを使って実現されます。以下に、簡単なデータ収集プログラムの例を示します^{*7}。

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
long *sensor _ address; // initialized elsewhere
fifo _ int fd;
int sampling _ period _ ns = 50000; // 20kHz sampling
frequency
void *collect _ data(void *unused)
{
    int data;
    clock _ gettime(CLOCK _ REALTIME, &next);
    while (1) {
        timespec _ add _ ns(&next, sampling _ period _ ns);
        clock _ nanosleep(CLOCK _ REALTIME, TIMER _
            ABSTIME, &next, NULL);
        data = *sensor _ address;
        write(fifo _ fd, &data, sizeof(data));
    }
}
int main(int argc, char **argv)
{
    pthread _ t thread;
    pthread _ attr _ t attr;
    pthread _ attr _ init(&attr, NULL);
    pthread _ attr _ setcpu _ np(&attr, 0);
    mkfifo("/myfifo", 0755);
    fifo _ fd = open("/myfifo", O _ WRONLY|O _ NONBLOCK);
    pthread _ create(&thread, &attr, collect _ data,
        NULL);
    rtl _ main _ wait();
    pthread _ cancel(thread);
    pthread _ join(thread, NULL);
    close(fifo _ fd);
    unlink("/myfifo");
    return 0;
}
```

これはプログラマには通常ユーザ空間Linuxプログラムに見えますが、実際には、Linuxモジュールの作成とmain関数の実行はLinuxのカーネルコンテキストで行われます。プログラムのmain関数は、シグナルによって終了させられるまでrtl_main_waitでブロックされます。Linuxカーネル モジュール プログラミング インタフェースを明示的に使用して、Real-Time Coreアプリケーションを実装することも可能です。

この例はFIFOとリアルタイム スレッドを作成します。このリアルタイム スレッドは、pthread CPU属性を使用して、CPU番号0で実行するよう割り当てられます。スレッドが作成されると、このスレッドはPOSIXのclock_nanosleep関数を使用して、自分自身のための周期的スケジュールを確立します。絶対時間モードのclock_nanosleep (TIMER_ABSTIME) の使用は、周期的リアルタイムスレッド実装の鍵となります。相対時間モードのclock_nanosleepはクロック スキューを発生させ、蓄積されてシステムの誤差となります。

各周期においてメモリマップド センサレジスタが読み込まれ、その結果がRT-FIFOに書き込まれます。最も単純なユーザ空間データ記録プログラムは次のようなものとなります。

```
cat /myfifo >file
```

もちろん、インターネットを介してGUIにグラフを表示しながらSQLデータベースにデータを格納するような、複雑なものとなる可能性もあります。デフォルトでは、Real-Time Coreスケジューラは、スレッド用の浮動小数コンテキストの保存と復元のサポートを可能にしていません。スレッド内でハードウェア浮動小数演算を使用する必要がある場合は、FP pthread属性をセットすることによって、これを明示的に有効にしなければなりません。スケジューリング優先度などのその他のスレッド パラメータも、属性を使用して指定できます。POSIXのpthread_setschedparam関数を使用して優先度を実行時に変更することも可能です。

割り込み

Real-Time Core APIには、ユーザ割り込みサービスルーチン (ISR) のインストール手段とアンインストール手段があります。ISRは常に、プロセッサ内のハードウェア割り込みが無効にされた状態で実行されるので、他の高優先度イベントの遅延を避けるには、ISRをできるだけ高速に保つことが推奨されます。割り込みラインはISRが呼び出される時点までに自動的に無効にされます。システムが同じライン上でその後の割り込みを受信できるように、明示的な形で有効に再設定される必要があります。

非常に高速の割り込み応答が求められるアプリケーションでは、通常、直接ISRで割り込み処理を行うことが望まれます。さもなければ、リアルタイムスレッドに割り込み処理を代行させることが推奨されます。この方法では、インクルードされたGDBスタブを使用して、割り込み処理ルーチンのソースレベル デバッグを行うことができます。ハードウェア浮動小数がサポートされているのはリアルタイムスレッドに対してのみで、ISRではサポートされていません。以下の例は、割り込み処理をスレッドに代行させた場合の例です。

```

#include <rtl_core.h>
#include <pthread.h>
#include <semaphore.h>
sem_t intsem;
pthread_t th;
unsigned int isr(unsigned int irq, struct rtl_frame
*int_frame)
{
// initial interrupt processing goes here
sem_post(&intsem); // wake up the thread
return 0;
}
void *irq_thread(void *unused)
{
while (1) {
sem_wait (&intsem); // wait for interrupt
// additional interrupt processing here
rtl_hard_enable_irq(irq); // re-enable the
interrupt
}
return NULL;
}
int main(int argc, char **argv)
{
sem_init (&intsem, 1, 0); // "public" semaphore,
// initial value is 0
rtl_thread_create(&th, NULL, irq_thread, NULL);
rtl_request_irq(irq, intr_handler);
rtl_main_wait(); // wait until the application is
killed
rtl_free_irq(irq);
rtl_thread_cancel(th);
rtl_thread_join(th, NULL);
return 0;
}

```

ソフト割り込み

Real-Time Coreアプリケーションはロード可能なカーネルモジュールなので、多くの場合は、Linuxカーネル機能呼び出しで処理を行うのに好都合です。しかし、Linuxがクリティカルセクションにある場合を含め、Real-Time CoreはいつでもLinuxに割り込むことができるので、Real-TimeCoreのスレッドや割り込みサービス・ルーチンからLinuxの関数を呼び出すことは、一般に安全ではありません。Real-TimeCoreは、この問題に対応するために、ソフトウェア割り込みメカニズムを備えています。rtl_get_soft_irq関数を使用することによって、プログラム内の関数は割り当てられたLinux IRQ番号に対応付けられます。rtl_global_pend_irq関数を使用すれば、割り込みを安全にリアルタイム・コンテキストからスケジュールすることができます。登録された関数を安全に呼び出すことができる場合、Linuxはその関数を呼び出して割り込みを処理します。

デバイスドライバ

多くの場合、Real-Time Coreアプリケーションはハードウェア用の独立したドライバを必要としません。たとえばデータ収集アプリケーションは、スレッドからDAQカード・レジスタに直接アクセスできます。メモリマップドPCIデバイスにアクセスする場合は、Linuxのカーネル関数か、Real-TimeCore付属のPCIドライバを使用できます。ターゲット ハードウェア デバイスを複数のアプリケーションに再利用する場合は、アクセスライブラリまたはデバイスドライバを実装することが望まれます。

Real-Time Coreは、アプリケーションへのPOSIXのopen/read/writeインタフェースを備えたドライバを実装するシンプルなインタフェースを提供します。仮定のセンサ用にread呼び出しをサポートするドライバの簡単な例を以下に示します。

```

#include <rtl_posixio.h>
#include <rtl_unistd.h>
static rtl_ssize_t mydev_read(struct rtl_file
*filp, char *buf, rtl_size_t count, rtl_off_t* ppos)
{
long data = *sensor_address;
if (data_is_invalid(data)) {
return -1;
}
if (count != sizeof(data)) {
return -1;
}
*((long *)buf) = data;
return sizeof(data);
}
static int rtl_mydev_open(struct rtl_file *filp)
{
return 0;
}
static int rtl_mydev_release(struct rtl_file *filp)
{
return 0;
}
static struct rtl_file_operations rtl_mydev_fops = {
read: rtl_mydev_read,
open: rtl_mydev_open,
release: rtl_mydev_release
};
int main(int argc, char **argv)
{
rtl_register_dev("/dev/mydev, &rtl_mydev_fops, 0);
rtl_main_wait();
rtl_unregister_dev("/dev/mydev");
return 0;
}

```

このドライバノードには、先に述べたpthreadsの例におけるリアルタイムFIFOと同様にアクセスできます。

同期と一方向キュー

上述した「デザイン」の項に示すように、IPCは控えめに使う必要があります。そうすることによって、優先度の逆転やデッドロックを防ぐことが可能です。ブロッキング方法の代替策としてロックフリーを使用できる場合は、その代替策を使用するのが最良の方法です。リアルタイムFIFO、共有メモリ、POSIX同期プリミティブの他、Real-Time Coreは、一方向キューと呼ばれるロックフリー通信のメカニズムを備えています。このメカニズムでは、同期のための割り込み無効化やスピンロックは使用しません。したがって、ユーザ空間でも同じ効率で使用できます。シングルリーダー/ライタの場合はロックフリーアクセス法が安全ですが、同時読み込みまたは書き込みは、スピンロックまたはミューテックスを使用してシリアル化しなければなりません。

以下のコードフラグメントでは、eventqという名前の一方向キュータイプが宣言されています。キュー長は20です。キューに保存されるデータのタイプはintで、キューが空の場合はeventq_deqが0を返し、キュー内にスペースがない場合はeventq_enqが-1を返します。

```

#include <rtl_onewayq.h>
DEFINE_OWQTYPE(eventq,20,int,0,-1); // declare the types
DEFINE_OWQFUNC(eventq,20,int,0,-1); // define one-way
queue access functions

```

宣言の実行後は、キューのインスタンスを作成し、自動的に定義されるアクセス関数を使用してデータの読み込みと書き込みが可能です。

```

static eventq myeventq;
eventq_enq(&myeventq, i); // enqueue i into the
one-way queue
j = eventq_deq(&myeventq); // dequeue an integer from
the queue

```

付属のプリミティブを使用すれば、より複雑なIPC機能を組み込むことができます。たとえば、メッセージパッシングはRT-FIFOを基にして実装可能です。また、「デザイン」の項に示すように、共有メモリとFIFOを使用すれば効率的なスプーリングメカニズムを実装できます。その他にも、条件変数を使用したイベントフラグ通知の実行など、さまざまな機能を実現することが可能です。

デバッグ

リアルタイムプログラムのデバッグには困難が伴うことがあります。ブレークポイントの設定やステップ実行などの伝統的なデバッグ手法が適切でない場合があるからです。これは、主に以下のような理由によります。

- 一部のエラーはシステムのタイミング内で発生します。プログラムを停止させるとこのタイミングが変わってしまうので、動作を変更することなくシステムを解析することができなくなります。
- リアルタイムプログラムが特定のハードウェアを制御している場合、解析のためにプログラムを中断すると、ハードウェアに不具合が発生したり、最悪の場合は破損したりする恐れがあります。

従来からある対話的なソースレベルのデバッグに加えて、Real-Time Coreでは、POSIXトレース機能のサブセットが実装されています。これらの機能を使用すれば、リアルタイムプログラムを実行しながら、リアルタイム性能を解析して評価を加えることができます。POSIXトレーシングに関する概要は、SUS (Single UNIX Specification) ^{※8}に示されています。

トレーシング機能を使用すれば、開発者はイベントプローブをプログラムに挿入して、プログラム実行中にイベントを記録することができます。トレーシングに関わるものは2つあります。解析対象となるプログラムと解析プロセスです。解析対象のプログラムにトレーシング用の措置を施せば、実行中に発生したイベントに関する情報が記録されます。それぞれのイベントに対しては、現在のCPUに関する情報、現在のスレッドID、タイム・スタンプ、オプションのユーザデータが、インメモリバッファに記録されます。

Real-Time Coreのトレーサでは、コンテキストスイッチなどの数多くのシステムイベントに対応する組み込みトレースポイントを使用できます。トレースイベントストリームは、ユーザ空間の解析プロセスによって継続的に読み込むか、プログラムに従ってコンソール上に表示することができます。後者の代表的な用途が、システムの事後解析です。重大な障害状態が検出されると、プログラムはコンソールへのトレースダンプを行い、障害発生直前に何が起こったのかについての情報をプログラマに提供します。場合によっては、展開後もシステムのトレーシングを可能にしておくのも有効な方法で、稀にしか発生しない障害のデバッグを容易にすることができます。

注記

- ※1 選択されたアーキテクチャ上で、Wind River User Space Real-Timeコンポーネントは、Linuxプロセスのコンテキストにハードリアルタイムリアルタイム スレッドを実装する方法を提供します。
- ※2 Bruce Powell Douglass, Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems (Boston: Addison-Wesley, 2002). (「リアルタイム デザイン パターン:リアルタイムシステム用の頑丈でスケーラブルなアーキテクチャ」)
- ※3 マルチプロセッサシステムでは、リアルタイムタスク用にCPUを予約できます。予約されたCPUには、この制限は適用されません。
- ※4 C.L. Liu and J.W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” Journal of the ACM 20 (1973): 44-61. (「ハードリアルタイム環境におけるマルチプログラミング用スケジューリング アルゴリズム」)
- ※5 Sang H. Son, Advances in Real-Time Systems (Englewood Cliffs, NJ: Prentice Hall, 1984), 225-248. (「リアルタイム システムの利点」)
- ※6 Wind River, “Wind River Real-Time Core Programmer’s Guide,” 2007, <http://www.windriver.com/support/>. (「ウインドリバーReal-Time Coreプログラマーズ・ガイド」)
- ※7 本資料における例ではAPI関数からの戻り値をチェックしていませんが、実際のアプリケーションでは、デバッグを容易にするためにチェックを行うことを推奨します。
- ※8 The Open Group, Single UNIX Specification, [http:// www.unix.org/version3/](http://www.unix.org/version3/).

ウインドリバーはスマートデバイス搭載ソフトウェアの最適化 (DSO) をワールドワイドに提供するリーディングカンパニーです。企業がスマートデバイスに搭載するソフトウェアを、品質および信頼性のさらなる向上を実現しつつ、リーズナブルなコストで開発することを可能にし、早期にマーケットへ投入することを支援します。

WIND RIVER ウインドリバー株式会社

東京本社 〒150-0012 東京都渋谷区広尾1-1-39 恵比寿プライムスクエアタワー TEL.03-5778-6001 (代表) FAX.03-5778-6002
大阪営業所 〒532-0011 大阪市淀川区西中島7-5-25 新大阪ドイビル TEL.06-6100-5760 (代表) FAX.06-6100-5761
E-mail:info-jp@windriver.com <http://www.windriver.co.jp>

登録商標: Wind River, Wind Riverロゴ, Tornado, VxWorksは、Wind River Systems, Inc.の登録商標または商標です。記載されているすべての名称は、各社の登録商標、商標またはサービスマークです。