社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－OS－103 (6)
2006／7／31

厳密な周期タスク実行を支援する実時間Linuxカーネルの実装

戴 毛兵[†] 石川 裕[††]

本研究では、マイクロ秒以下のタスク開始時間の遅延、効率良い実時間スケジューラ、及び周辺デバイスからの割り込み禁止を実現する実時間カーネル SHI-Linux を実装する。CPU が 2.6GHz、カーネルに 2000 個通常プロセスが存在する環境でのテストの結果、開発したカーネルでの周期タスクの開始遅延時間は最大 452us、最小 10ns、平均 700ns 以下になった。

Implementation of a Real-Time Linux Supporting Strict Periodic Tasks

MAOBING DAI [†] and YUTAKA ISHIKAWA[††]

In this study, a real time kernel named SHI-Linux is implemented which supports an efficient real time scheduler. The start time of the real time task has one microsecond or less delay, and interrupts from devices are prohibited in this kernel. As a result of the test in an environment where 2000 oirginal Linux processes run, the delay of the periodic task start time has maximum value of 452us, minimum value of 10ns and average value of 700ns with a 2.6GHz CPU.

1. はじめに

Linuxを実時間カーネル化するには、実時間タスクをいかにデッドライン時間内に終了させるかが重要である。しかし、従来のLinuxカーネルでは、デバイスからのランダムな割り込み処理、スケジューラのオーバーヘッド、割り込み禁止期間の存在などにより、実時間タスクの開始時間が大幅に遅延する。本研究では、スケジューラの起動から、実時間タスクの実行開始までの遅延を予め測定しておき、実時間タスク実行開始時刻よりも遅延時間分早い時刻にスケジューラを起動させる方法を取る。また、時間通りにスケジューラを呼び出す方法としては、APIC(Advanced Programmable Interrupt Controller) の one-shot モードを利用する。このようにして、タスクの実行開始時間の遅延を減少させることができる。

実時間カーネルの実現には、リアルタイムアルゴリズムにしたがって構成された実時間スケジューラが不可欠である。これまで多くのリアルタイムスケジューリングアルゴリズムが提案された。たとえば、タスクの周期時間によって、静的に優先度を割り当てる RM(Rate Monotonic)[1] アルゴリズム、タスクのデッドライン時間によって、動的に優先度をつける EDF(Early Deadline First)[2] アルゴリズムなどがある。EDF アルゴリズムは実時間タスクの優先度を動的に変更するので、実装上のオーバーヘッドが高いと言われている。しかし、実装上 RM アルゴリズムと同じコストであると報告されている[10]。本研究では、EDF をベースに実時間スケジューラを実現している。タスクランキューは Linux のタイマー機構に習い、タスク周期時間によって、4 つに分かれている。それぞれのランキューでは、実時間タスクがスタート時間順に分布されている。スケジューラがスタートラインに立つすべての実時間タスクのなかで、もっともデッドライン時間が小さいものを選び出す。スタート時間が違う場合、実時間タスクの優先度は、O(1) の操作で決められる。同じスタート時間に多数のタスクがある場合は、O(n) の操作で決められる。

実時間タスクの実行を妨げるもう一つの要因は割り込みである。割り込みが発生すると、デッドライン時間に近付いているタスクがあるにもかかわらず、現在実行中のタスク実行が中断され、割り込み処理が実行される。このため、デッドラインをミスする可能性がある。本研究では、これまでの一般的な実時間 Linux カーネルと違い、デバイスからの割り込みを禁止し、デバイスドライバーを一つの実時間周期タスクにする方法を採用する。このようにすると、カーネル内の割り込みによるランダムな実行がなくなり、タスクの実行時間の予測性が保証される。

本研究で実装した SHI-Linux カーネルを用いて以下の実験結果が得られた。システムのスケーラビリティを評価するのに使う hackbench ベンチマーク[11] を利用して、システムに 2000 個通常プロセスが存在する

† 東京大学情報理工学研究科コンピュータ専攻
Deparment of Computer Science, University of Tokyo
†† 東京大学情報理工学研究科
Graduate School of Information Science and Technology, The University of Tokyo

環境で、実時間タスクの開始時間の遅延が最大 452us、最小 10ns、平均 700ns となった。また、実時間スケジューラのオーバーヘッドは実時間タスクの数が増えても平均 270ns になっている。このほか、FSMLabs 社の RTLinux[4)5)] や ART-Linux[6)] など従来の実時間カーネルにおいて、割り込みが頻繁に起こる、実時間タスクがデッドラインを守れないことに対して、開発したカーネルでは守られることを確認した。

本論文の構成は次のようになっている。第 2 章では既存の実時間カーネルを述べる。第 3 章は実時間カーネルの実現方針を述べる。4 章では、3 章に基づいて、実装内容について述べる。第 5 章では、実装したシステムの評価と考察に触れる。最後に 6 章では本論文をまとめ、結論を述べる。

2. 既存実時間カーネル

Linux をベースとした実時間カーネルはこれまで多く開発された。本章では FSMLabs 社の RTLinux と ART-Linux について述べる。この他にも実時間化した Linux には、KURTLinux[8)]、Linux/RK[7)]、SRTLinux[9)] などがある。

2.1 RTLinux

FSMLabs 社の RTLinux は Victor Yodaiken によって開発された実時間カーネルである。RTLinux は、Linux カーネルを最も優先度の低い一つのプロセスとして実現している。実時間タスクは Linux カーネルと同じように、ハードウェアにアクセスできるため、ユーザからハードウェアが保護されていない。また、独自のメモリ空間で実行されているため、Linux 現有の多くのリソースをそのまま実時間タスクが利用することはできない。

RTLinux の割り込み処理はエミュレートされている。実時間タスクが存在している場合は、デバイスからの割り込み処理が遅延されている。そのため、ネットワークなどの割り込み頻度が高い環境では、処理が間に合わない恐れがある。

このように、RTLinux は実時間タスクの要求を満たす一方、現有の多くの Linux リソースをそのまま実時間タスクに利用できないことになっている。また、普通のユーザからハードウェアが保護されていないこともひとつのデメリットになる。

2.2 ART-Linux

ART-Linux は石綿氏によって、1995 年に開発されたリアルタイムカーネルである。

ART-Linux では、実時間周期タスクおよび固定優先度スケジューラが実現されている。これらの拡張を用いて RM アルゴリズムを実現できる。しかし、周期タスクセットが与えられた時に、ART-Linux 側でタスクの優先度を決めず、優先度はユーザの設定に任されている。

ART-Linux では、周期的に割り込み信号を監視して、デバイスドライバを実行している。ドライバの周期はカーネル内に静的で決まっている。

ART-Linux のタイマ精度はカーネルをコンパイルするとき、決めなければならない。つまり、ART-Linux のタイマ精度は静的であり、実時間タスクの要求に応じて変更できない。

RTLinux も ART-Linux も実時間周期タスクの構成には、タスクのスタート時間が決めることができなく、スタート時間の遅延処理もない。

3. 実現方針

厳密な実時間タスクには、スタート時間と周期時間とデッドライン時間などの時間制約要素がある。カーネルは普通のタスクの実行から、実時間タスクに切替えるとき、実時間タスクのスタート時間通りにタスクを実行させる必要がある。スタート時間の遅延が大きくなると、ランキューの中の実行待ちのすべての実時間タスクに大きな影響を及ぼし、システムのリアルタイム性能を大幅に低下させる恐れがある。したがって、実時間スケジューラの早い段階での呼び出しとスケジューラが実時間タスクを時間通りに実行させることがポイントになる。

Linux のスケジューラの呼び出しは三つのタイミングで起こる。一つ目はプロセスが自主的にスケジューラを呼び出すとき、二つ目はカーネル空間からユーザ空間に戻るとき、三つ目は割り込み処理から復帰するときである。この内、一つ目と二つ目のタイミングはランダムでコントロールできない。しかし、タイマ割り込みのタイミングは制御できる。したがって、実時間スケジューラの呼び出しはタイマ割り込みを利用して実現できる。APIC では、ある時間を経てからタイマ割り込みを一回だけ発生させる one-shot モードがある。本研究では、実行可能な実時間タスクがない時、次の実行すべき実時間タスクのスタート時間を APIC のカウンタレジスタに設定して、実時間タスクを呼び出せる方法を取っている。ただし、APIC のタイマ割り込みから、スケジューラの呼び出し、ないしは実時間タスクの実行開始までは遅延がある。この遅延時間も含めて、動的に APIC に時間間隔を設定する。

3.1 スタート遅延時間

割り込みが多い普通の Linux 環境では、実時間タスクのスタート遅延時間は、カーネルにおける割り込み禁止期間、プリエンプション禁止期間などに大幅に左右される。いわゆる実時間スタート時間のばらつき (ジッター) が顕在化している。しかし、デバイスからの割り込みを禁止した本研究のカーネルでは、実時間周期タスクと普通のカーネルスレッドがある期間中に安定している場合は、APIC のタイマ割り込みから実時間タスクの実行までのパターンが安定している。したがって、実時間タスクの遅延時間も安定している。実時間タスクは毎回前回の遅延時間を取り、次の周期で、この遅延も含めて、APIC タイマに設定する。図 1 では、スタート遅延時間の処理を表している。

このように、実時間タスクのジッターが大きく抑えることができる。

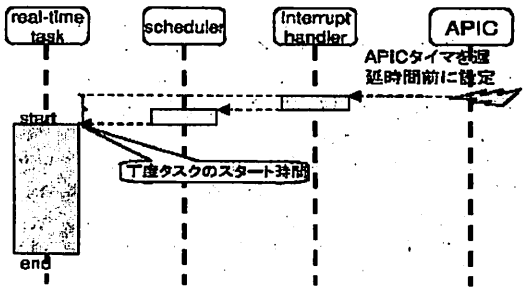

図 1　スタート遅延時間の処理

3.2 デバイス割り込みの禁止

割り込みは実時間タスクの実行時間の予測に影響を及ぼすため、ハードウェアデバイスからの割り込みを禁止する。デバイスハンドラは実時間カーネルスレッドに変える。このなか、タイマ割り込みだけが許され、スケジューラの呼び出し契機に使われる。タイマ割り込みルーチンは、set_need_resched() と jiffies を増加する。大域システムタイマの維持はタイマ割り込みハンドラを周期タスクに変えることによって実現する。

3.3 スケジューラ

Linux をベースに実時間カーネルを実現するには、周期タスク、非周期タスク、一般的な Linux プロセスの三種類に対応しなければならない。そのため、スケジューラを二つに分ける。一つは一般 Linux プロセスを扱う元の Linux スケジューラである。もう一つは実時間タスクを扱う実時間スケジューラである。実時間タスクがある場合、リアルタイムスケジューラが呼び出され、実時間タスクを優先的に実行させる。リアルタイムスケジューラのアルゴリズムは EDF を採用する。カーネルは周期タスクの一周期実行終了後、次の周期が始まるまで、実行権を他の実時間タスクに渡すか、元の Linux スケジューラを呼び出して、一般タスクの実行を行う。

リアルタイムスケジューラは実時間タスクの増加によって、ランキュー操作のオーバーヘッドが増加する。したがって、このオーバヘッドの最悪値が求められている。本研究では、ロボットセンサーなどのハードリアルタイム環境を想定していて、1ms 以下の周期タスク時間のランキュー操作を O(1) で終ることを目指す。そのための一つの対策はランキュー操作の分散である。スケジューラの中で、実行するランキュー操作は次の実時間タスクの選択のみにする。タスクの優先度の再設定やデキュー操作は実時間タスクの実行が終った段階に遅延する。ランキューには、実行待ち状態となるタスクへのポインタを持っているので、スケジューラのランキュー操作は単なる選択したタスクの再確認だけとなる。したがって、スケジューラのオーバーヘッドは一定に近い。ランキュー操作のもっとも重い部分はタスクの優先度を設定する部分である。本研究でのランキューは図 2 のように、タスク周期時間によって us、ms、100ms、s と四つに分かれている。それぞれの実時間タスクの周期時間内に対応したランキューにそのタスクが入る。たとえば、周期時間がマイクロ秒単位のタスクは us ランキューに入る、1ms から 100ms までの周期時間を持つタスクが ms ランキューに入る。

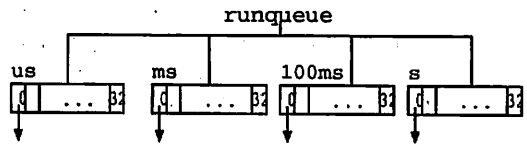

図 2　ランキューの構造

例を示す。us ランキュー (長さは 32 とする) に、

task1:

start_time = t1

us_runqueue_location = s1 で示されるタスクが入っている状態で新しいタスク

new_task:

start_time = t1 + d1 が us ランキューに入るとする。その位置 (優先度) は

us_runqueue_location = s1 + d1/(単位間隔のタイマ) になる。us ランキューの最大のタイマ期間が 1000us になっているため、

単位間隔のタイマ = 1000 / 32 になる。同じランキューにすでにタスクが入っていた状況では、線形探索としても、長さ (1000/32) が決められているので、スタート時間が違う場合は O(1) になり、スタート時間が同じ場合は O(n) になる。

3.4 時間精度

実時間タスクの周期時間が理論上、タイマの精度 (ns 単位) まで設定できる。しかし、タスク実行時間とシステム割り込み禁止期間、プリエンプション禁止期間を考慮して、最小 10us と設定されている。

3.5 実時間タスク

実時間タスクの制御は、大きく以下の部分に分かれる。

- 実時間タスクへの変更時
  通常 Linux プロセスは、rt_enter() 関数を呼び出すことにより実時間タスク化する。rt_enter() 関数引数で渡されるタスクスタート時間、デッドライン時間、周期時間は、CPU 時間に変換され、該当タスクが実時間ランキューに入る。タスクスタート時間が現在のタイムであれば、直ちにリアルタイムスケジューラが呼び出される。
- 一周期実行終了時
  実時間タスクが一周期実行を終わり、周期タスクのカウンターと状態を設定した後、現在実時間タスクの実行要求があるかどうかを検査する。要求がある場合、リアルタイムスケジューラを呼び出す。なければ、Linux のスケジューラを呼び出す。
- 通常タスクへの変更時

実時間タスクの実行が全て終って、終了段階に入るとき、実時間ランキューから取り出される。後は非実時間ランキューにエンキューして、Linuxのスケジューラを呼び出す。

カーネルは以上の制御をサポートするために、三つのシステムコールを追加する。それぞれは、sys_rt_enter()、sys_rt_next_period()、sys_rt_exit()である。また、ユーザが実時間タスクがデッドライン内で実行されているかどうかをチェックするための機構として、Linuxのprocファイルシステム上に情報が提供される。このほか、ユーザ側から動的に実時間タスクの周期を変更するできるデバイスファイルも提供する。

4. 実　　装

第2章では、Linuxをベースにリアルタイムカーネルの設計を述べた。この章では、Intel社x86系の基で、実際どのように実装したのか、具体的に述べる。

4.1 タスクスタート時間遅延の縮小

実際の実時間タスクスタート時間はスケジューラがどのような状況で呼び出されるかによって、遅延時間も違っている。割り込み処理から復帰するときとすると、遅延は割り込み処理のオーバーヘッドプラススケジューラのオーバーヘッドとなる。普通のLinuxプロセスからの場合は、単なるスケジューラのオーバーヘッドとなる。いずれにしても、スケジューラのオーバーヘッドは避けられないことになる。そこで、このスケジューラのオーバーヘッドをカバーするために、実時間タスクのスタート時間を早い段階でのタイマーを設定する。実装では以下のステップでこの遅延を隠蔽する。

- 遅延の測定
  実時間スケジューラの実行パスの最初で現在の時間を取る。後にスケジューラが終わり、実時間タスクが選択され、実行されるパスの最初にもう一度CPU時間を取る。この二つの時間差が一つの遅延の値になる。
- スタート時間の計算
  一回測定した遅延時間は場合によって、大きくずれることもあり得るので、スタート時間の計算は平均遅延時間を利用する。実際設定に使うタスクスタート時間はこの平均時間を減算したものとなる。
- スタート時間の設定
  実行待ち状態の実時間タスクがなくなると、次の実時間タスクのスタート時間を設定する必要がある。設定はAPICチップを通じて行う。APICはタイマー割り込みが一回しかしないone-shotモードと周期的に割り込みをかける普通のモードがある。実装ではone-shotモードを使う。実時間タスクのスタート時間が決まると、その時間がAPICのカウンター回数に直され、カウンターレジスタに書き込まれる。

このように、APICからの割り込みがくるとき、ちょうど次の実時間タスクが実行しようとするときである。カーネルは割り込み処理、そしてスケジューラを経て、実時間タスクにたどり着いたとき、実時間タスクのスタート時間の遅延は短縮することができる。

4.2 スケジューラ

スケジューラは普通のLinuxのスケジューラと実時間スケジューラと二つに分ける。実行待ち状態の実時間タスクがあると、実時間スケジューラが呼び出される。実時間スケジューラの操作は主にタスク実行状態のチェック、次のタスクの選択、タスクスイッチなどに分かれる。実際のタスク優先度の再設定や、エンキューは実時間タスクの実行が終った段階で行う。

実時間タスクランキューでは、常に最も早いスタート時間を持つタスクへのポインタを持つ。しかし、ランキューのなかでスタートラインに立ったタスク(実行待ちマスタ)が一つではない可能性や、もっとも早いスタート時間をもつタスクのデッドライン時間が一番小さいとはならない可能性があるため、スケジューラがタスクを再確認する必要がある。再確認作業では実行待ちタスク(スタート時間が現在時間より小さいタスク)に対して、最もデッドライン時間が近いタスクが選び出される。

図2から分かるように、実時間タスクは時間順に四つのランキューに分布している。したがって、再確認作業はすべてのランキューを調べる必要はない。この上、再確認作業が必要とした原因はスケジューラが呼び出される時間と最も早いタスクスタート時間の間の差である。一般にこの差は極めて小さいため、再確認作業ではランキューを走査する長さは短い。また、各ランキューの状態に対応する32bit unsigned long変数を導入する。ランキューのn番目要素に実時間タスクが入ると、変数のn番目のビットを1に設定する、したがって、ランキューの走査は変数のビット列の走査と同じである。変数のビット列の走査は1命令で済む。

以上のように、実時間スケジューラはタスクの数が増えても、効率よくスケジューリングすることができる。

4.3 デバイス割り込み処理

APICタイマー割り込み以外の割り込みはすべて禁止される。カーネルから割り込みを禁止する手法は二つある。CPUのEFLAGSレジスタのIFビットを0にする方法と割り込みコントローラのマスクレジスタをセットする方法である。本研究では、低レベルで割り込みを禁止するため、後者を取る。

デバイス割り込みハンドラを実時間タスクに変換するには、Linux割り込みハンドラ登録データ構造irq_desc[]->actionを利用する。デバイスがアクティブになるために、デバイス割り込みハンドラを構造体struct irqaction *actionに登録する必要がある。したがって、デバイス割り込みハンドラを実時間タスクに変換するには、ハンドラをactionに登録した時点

で周期タスクリストに登録する。

タイマー割り込みは独自の処理ルーチンを定義する、タイマ割り込みハンドラはほかの実時間タスクの実行に影響を与えないために、単なる need_resched フラグの設定とシステム基礎時間 jiffies の増加だけを行う。

4.4 インタフェースの実装

ユーザ側が、実時間タスクを構成するために三つのシステムコールを追加する。このほか、ユーザは周期タスクの実行状態を確認するためにプロセスファイル/proc/shi-rtinfor を追加する。ユーザは実時間タスクが時間通りに実行しているか、またデッドラインミスする時間と回数をこのファイルを通じて入手できる。

さらに、ユーザが実時間タスクの周期時間を動的に変えるために、デバイスファイル/dev/shi-rt を追加する。ユーザが実時間タスクの id と変動する周期時間を指定して、このデバイスファイルを通じて、設定することができる。

5. 実験と考察

本章では、第 3 章で実装したカーネルの実験と性能を述べる。実験は FSMLabs 社の RTLinux と実装した SHI-Linux カーネルの 2 種類を用いる。実験環境を表 1 に示す。

5.1 タスクスタート時間の遅延

タスクスタート時間の遅延を測定した。それぞれのカーネルにおいて、実時間タスクがスケジューラによって選択され、実行しようとしている最初の時点で現在の CPU 時間を取り、その時間と実際に実時間タスクのスタート時間の差を取る。また、実験に使う実時間タスクの周期は 1ms で、何の処理もしない。この遅延の最大値と最小値の 10000 回の平均値を表 2 にまとめる。

また、システム負荷による変化にも着目するため、hackbench[11] を用いた。hackbench は OSDL が提供している性能測定用のツールである。実行すると多くのプロセスが生成され、パイプを用いた I/O 処理を頻繁に行う。今回の実験では、hackbench 50 を使って、約 2000 個のプロセスが存在している環境で、実時間タスクのスタート時間の遅延も測定する。結果を表 3 にまとめる。

表 2 と表 3 から、実装したカーネルは実時間タスクの平均スタート時間の遅延がほぼ 1us 以下であることが分かった。また、実装したカーネルでは、瞬間最大 452us のスタート遅延時間があり、これは Linux カーネルパスの割り込み禁止期間や、プリエンプション禁止期間が現時点でやはり長いことが分かった。

表 1 実験環境

構成要素	構成
CPU	Pentium4 2.6Hz
メモリ	768MB
HDD	40GB(ext3 ファイルシステム)

表 2 スタート時間の遅延 (単位 us)

kernel	max	min	average
RTLinux	16.5	8.2	9.3
SHI-Linux	12.0	-0.02	0.6

表 3 スタート時間の遅延 (負荷あり) (単位 us)

kernel	max	min	average
RTLinux	16.6	8.2	10.0
SHI-Linux	452.0	0.01	0.7

5.2 スケジューラ性能

スケジューラの性能としてスケジューラのオーバーヘッドに着目する。実験で使う実時間タスクは全て周期が 1ms で、なんの処理もしない。方法は次のようになっている。まずスケジューラの最初の部分に CPU 時間をとり、次にタスクスイッチの直前にもう一度 CPU 時間を取り、この二つの時間差を 10000 回測定して、その平均値を求める。また、実時間タスクの数を 4 から 32 個まで増やして、その平均値を同様に求める。結果は図 3 のようになっている。

図 3 から分かるように、実時間スケジューラのオーバーヘッドがほぼ 270ns となって、実時間タスクの数が増えても、増えないことが分かった。

また、実時間タスクの優先度設定に伴うランキュー操作のオーバーヘッドも測定する。測定では同じく、タスク数を 4 から 32 個まで増やし、10000 回の平均値を取る。結果は図 4 のようになっている。

測定したオーバーヘッドはデキュー操作とエンキュー操作のものである。図 4 から分かるように、実時間タスクの数が増えても、実際のオーバーヘッドはほぼ 300ns となって、増えないことが分かった。

5.3 割り込み環境での性能

割り込みが頻繁に生じる環境で、実時間カーネルに与える影響を測定する。それぞれのカーネルで、バックグランドでネットワークから約 1GB のファイルをドウンロードする。測定には 100us 周期の実時間タスクと 10ms 周期の実時間タスクを用いる。この状況下

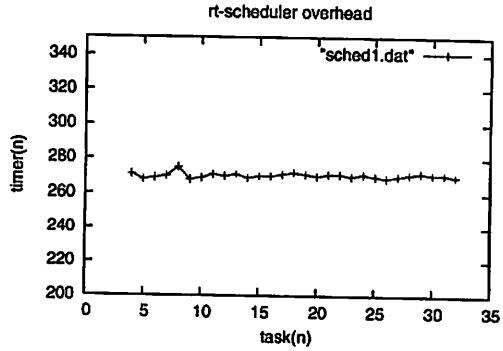

図 3 スケジューラのオーバーヘッド

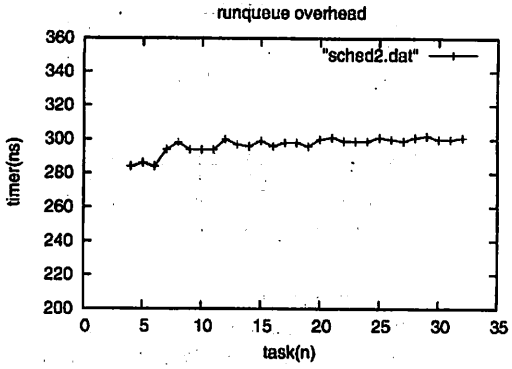

図 4 ランキュー操作のオーバーヘッド

でデッドラインミスを起こすかどうかを計測する結果は表 4 の通りである。

表 4 割り込み環境でのデッドラインミス状況

kernel	task(100us)	task(10ms)
RTLinux	X	○
SHI-Linux	○	○

表 4 からわかるように、実装したカーネルはデバイスドライバを周期タスクにしたため、たとえ割り込み混雑する環境においても、実時間タスクに影響なく、デッドラインミスすることもない。

以上の測定の結果から、本研究で実装した実時間カーネルはスタート遅延時間が 1us 以下と短く、スケジューラのオーバーヘッドも特定の環境において一定となって、ランダムな割り込み影響を受けずに、効率よく実時間タスクを支援することが確認された。

6. おわりに

本論文では、Linux をベースに、厳密な周期タスクを支援する実時間カーネルを実現するために、三つの有用な手法を用いた。一つは、実時間タスクがスタート時間通りに実行開始するため、早い段階でスケジューラを呼び出す方法である。二つは、特定環境で、スケジューラランキュー操作のオーバーヘッドを抑えるために、Linux タイマ機構のようなランキュー機構を作り上げ、効率良いスケジューラをつくる方法である。最後はランダムなデバイスからの割り込みを禁止して、デバイスドライバを周期タスクにする方法である。性能測定の結果、実時間タスクのスタート時間の遅延は最大 452us、最小 10ns、平均 700ns 以下と短く、スケジューラのオーバーヘッドがタスクの数が増えても一定となって、割り込み頻繁な環境でも実時間タスクに影響がないという結果が確認された。よって、実装したカーネルの有効性が確かめられた。

一方、Linux カーネルのソースコードの割り込み禁止期間とプリエンプション禁止期間は SHI-Linux の実時間性に大きな影響を与える、これらの期間を短くする必要がある。また、実装したカーネルのマルチスレッドアーキテクチャや、SMP 環境への対応も課題として残されている。

謝　辞

本研究の一部は、科学技術振興機構 (JST) の戦略的創造研究推進事業 (CREST) の支援を受けました。

参考文献

1) C. W. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment", Journal of the ACM, 20(1):46-61, 1973
2) W. Horn, "Some simple scheduling algorithms", Naval Research Logistics Quarterly, 21, 1974
3) J. P. Lehoczky, L. Sha, and Y. Ding ,"The rate monotone scheduling algorithm: Exact characterization and average case behavior", In Proceedings of the 10th Real-Time Systems Symposium, page 166-171, IEEE Computer Society Press, December 1989
4) M. Barabanov and V. Yodaiken, "Introducing Real-Time Linux", Linux Journal, February 1997.
5) V. Yodaiken, "The RTLinux Manifesto", In Proc. of the 5th Linux Expo, Raleigh, NC, March 1999
6) 石綿陽一, "リアルタイム処理を実現する ART-Linux の設計と実装", Interface, Vol.25, No. 11, 1999
7) S. Oikawa and R. Rajkumar "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior", In Proceedings of the IEEE Real-Time Technology and Applications Symposium Vancouver, June 1999
8) B. Srinivasan, S. Pather, and D. Niehaus, "A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Software", In Proceedings of IEEE Real-Time Technology and Applications Symposium, June 1998
9) Ingram, D, "Integrated Quality of Service Management", University of Cambridge Computer Laboratory Technical Report, No. 501, 2000
10) Giorgio C. Butiazzo "Rate Monotonic vs. EDF: Judgment Day", University of Pavia, Italy, Real-Time Systems, 29, 5-26, 2005
11) http://developer.osdl.org/craiger/hackbench/