情報処理学会論文誌　コンピューティングシステム　Vol. 2　No. 1　87–95 (Mar. 2009)

リアルタイムLinuxのための軽量割込み処理

戴　毛兵[†1]　石川　裕[†1]

ランダムな割込み処理は実時間タスクの実行を阻害する1つの要因である．本問題を解決する方法として，割込みハンドラのスレッド化がある．しかし，割込みハンドラをスレッド化することによりスケジューラのコンテキストスイッチ時間が増大し，実時間タスクとの優先度管理が問題になる．本論文では，実時間タスクと周辺デバイスの依存関係を基に周辺デバイスからの割込み処理をスレッド化処理ではなく，実時間タスク実行の前，途中，後にそれぞれ遅延させる方法を提案する．Linux 2.6.20 上で実装した結果，従来のスレッド化方法よりコンテキストスイッチ時間が25%減少し，実時間タスクのOS遅延は約半分に短縮される．

A Light Interrupt Management for Real-time Linux

Maobing Dai[†1] and Yutaka Ishikawa[†1]

One of the factor that disturbs the execution of a real-time task is the execution of an interrupt handler. To solve this problem, in a traditional technique, an interrupt handler is executed as Linux kernel threads so that real-time tasks are executed periodically. Although the real-time OS latency is improved by this method, the time of the scheduler context switch is increased and the priority management between the real-time tasks and the interrupt processing becomes an issue. In this paper, a new technique is proposed to deal with this interrupt problem, in which, accroding to the priority of the whole system, the interrupt processing does not act as a thread and is delayed at three points: previous to real-time task execution, after specific real-time task execution, and behind the whole real-time task execution. A prototype system is implemented on Linux 2.6.20 and the experimental results show that the time of scheduler context switch is reduced to 25% and the maximum OS latency of the real-time tasks is improved to half of the traditional one.

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

1. はじめに

Linuxをベースに実時間システムを構築するときの1つの問題として，ランダムな割込みがある．割込みが発生すると，デッドライン時間に近づいているタスクがあるにもかかわらず，現在実行中の実時間タスクが中断され，割込み処理が実行される．このため，実時間タスクのデッドライン時間が守れなくなる可能性がある．特に高性能ネットワーク環境におけるネットワーク割込みが多い場合に，この問題が顕在化している[1)]．本割込み問題の原因は，Linuxシステムの優先度管理ポリシに起因する．Linuxでは，すべてのハードウェア割込みがソフトウェアタスクより高い優先度を持っている．しかし，実時間システムでは，実時間タスクがハードウェア割込み処理よりも優先的に処理されなければならない場合がある[2)]．したがって，Linuxを基に実時間システムを構築するには，Linux通常タスク，割込み処理，実時間タスクの3つに対して，新たな優先度管理が必要となる．

本割込み問題を解決するための手法として，割込み処理のスレッド化（以下，割込みスレッドと呼ぶ）が提案されている[2)–5)]．実時間タスクと割込みスレッドの優先度を適切に設定することができれば，システム周辺デバイスからの割込み処理のランダム性がなくなる．これにより，実時間タスクはデバイスから割り込まれずに実行することが可能となり，デッドライン時間を満たせられる．

しかし，本手法では以下の2つの問題がある．まず，優先度管理が難しくなる．割込みスレッドの優先度が実時間タスクより高い場合と低い場合がある．ある実時間タスクにおいて，ある割込みスレッドの処理を待つ状態では，割込みスレッドの優先度は，この実時間タスクの優先度を継承しなければならない．すなわち，動的な優先度管理機構が必要となる．動的な優先度管理は一般にシステムのオーバヘッドの増大につながる．このため，Timesys[3)]，Redhawk[4)]などの実時間システムでは，静的に割込みスレッドと実時間タスクの優先度が決められている．

割込み処理をスレッド化するもう1つの問題は，タスクディスパッチャとコンテキストスイッチの回数が増大することである．周辺デバイスからの割込みを処理するために，割込みスレッドが頻繁に起床される．これにともない，タスクディスパッチャとコンテキストスイッチの回数が増加する[1)]．著者らが以前開発したShi-Linux[14)]カーネルを用いて割込み処理をスレッド化した場合と，オリジナルLinux 2.6.20カーネルのコンテキスト回数の違いを測定した．結果を**表1**に示す．実験では，Intel社製3 GHz Pentium 4 CPUを搭載し512 MBの主記憶を有する計算機を使用し，1 GBのファイルをネットワーク経由で取り

表 1　コンテキストスイッチ回数の増加
Table 1　Increments of context switch.

	オリジナル Linux 2.6.20	割込み処理スレッド化	増加率（倍）
最大	297,953	427,803	1.435
最小	235,132	322,232	1.370
平均	282,812	403,518	1.426

込む間のコンテキストスイッチ回数を 10 回計測した結果である．割込みスレッド化によりディスパッチャ回数は約 1.4 倍増えていることが分かる．このように，割込み処理をスレッド化する方法は実時間タスクの実行予測性を上げる一方で，システムオーバヘッドが増大することになる．

本論文では，これらの問題を解決するために，Linux カーネルに軽量的かつ動的な優先度管理機能つき割込み処理手法を提案する．本手法では，すべての割込み処理がタスクに関連付けられ，その優先度が以下のように定義される．

(1) タイマ割込みなど即時に処理しなければならない優先度が最も高い割込み処理は，実行中のタスクのコンテキストの下で即時に実行される．

(2) 実時間タスクが，ある割込み処理を待つとき，当該割込みの優先度は，動的にその実時間タスクの優先度を継承する．当該割込みの処理は優先度ベースに現在のタスクの下で実行されるか，低優先度タスクの実行中に実行される．

(3) その他の割込みは実時間タスクよりも優先度が低く設定され，すべての実時間タスクの処理が終了してから実行される．

なお，複数の実時間タスクが同時に同じ割込み処理を待つ場合，当該割込み優先度の定義は複雑になり，本論文ではこの問題は将来研究として残る．

また，実時間タスクの開始は一般にタイマ割込みを用いて，タスクの開始時刻を指定している．しかし，カーネルに割込み禁止区間があるため，タイマ割込みの処理は遅延される．本論文では通常割込み禁止区間に対する割込み禁止命令が使用されず，タイマ割込みが即時に発生する．割込み処理は優先度ベースに処理されるが，従来の割込み禁止期間中に到着した場合，その割込みハンドラの実行は割込み禁止期間の最後まで遅延される．

このように，本提案手法では，デバイスからの割込み処理をスレッド化せずに，割込み処理の優先度を動的に実時間タスクと関連付けることが可能となる．本手法は単一 CPU での実時間 Linux の構築に焦点を当てるが，一般的な実時間システムにおいても，同様な問題がある．本手法を用いて，割込み処理の優先度管理の改善，割込みスレッド化方法を採用する実時間システムの性能の改善にも，有用である．

本論文の構成は次のようになっている．2 章では割込み処理の関連研究について述べる．3 章では提案する割込み処理方式の詳細を述べる．4 章では提案する割込み処理方式の実装を述べる．5 章では実装したシステムの性能を検証する．6 章で本論文をまとめる．

2. 関 連 研 究

Linux をベースに実時間システムを構築するには，割込み問題を解決しなければならない．これまで割込み問題に対処する手法はいくつか提案されている．1 つは割込みスレッド方式である．TimeSys と QNX[6] では，割込み処理の主な部分はスレッドで実行される．割込みが発生すると，対応した割込み処理スレッドが起床され，割込み禁止モードで割込みスレッドが実行される．同手法では，割込みハンドラは 2 つの部分に分かれる．1 つは割込み受付部分であり，もう 1 つは割込みを処理する部分である．割込み受付部分の処理は小さく，実時間タスクの実行への影響を最大限に抑えられる一方，割込み処理スレッドを起床するため，コンテキストスイッチとタスクディスパッチャの回数が増大する．TimeSys や QNX では，スケジューラが割込みスレッドに高い優先度を与えることがなく，割込みスレッドが実質的にすべての実時間タスクの後に実行される．実時間システムでは，実時間タスクの優先度が必ず割込み処理より高いわけではないため，本割込みスレッド方式ではシステム優先度管理の問題が依然残っている．

割込みスレッド方式のもう 1 つの手法は，割込み処理スレッド内でデバイスの状態をポーリングする方式である[7),13)]．同手法ではタイマ割込み以外，外部からの割込みはすべて禁止され，割込み処理スレッドは周期的にデバイスの状態をポーリングする．同方法では，実時間タスクが先行的に実行することができ，静的な優先度を指定した場合は，割込みスレッドと実時間タスクの間に優先順序を指定することができる．しかし，割込みスレッドによって周期的にデバイス状態をポーリングすると，CPU 使用率が落ちる．さらに，スケジューラが頻繁に割込みスレッドを起床するため，タスクディスパッチャとコンテキストスイッチの回数が増大する．

Luis ら[2)] は，予測可能な割込み管理機構を提案している．本機構では，すべての周辺デバイスからの割込みは 1 つの小さな割込み処理プログラムで受理される．割込み処理プログラムは現在処理中のタスクの優先度と受理された割込みの優先度と比べ，受理された割込みの優先度が高い場合は，シグナルによって，対応した割込みスレッドを起床させる．本機

構はシステム全体で優先度管理が可能となり，割込み処理は実時間タスクの後に実行されることはなくなる．しかし，この割込み管理機構では，実時間タスクと割込みスレッドとの優先度指定は静的にしか指定できず，多くのデバイス I/O 処理を待つ実時間タスクにとっては，すぐに処理してほしい割込みスレッドの実行が遅延されることになる．また，本方式でも割込みスレッドを起床するためのタスクディスパッチとシステムコンテキストスイッチ回数が増大する．

このように，割込み問題を解決する割込みスレッド方式では，タスクディスパッチャとコンテキストスイッチの回数が増大し，システム全体の負荷が増大する．このため，軽量的かつ動的な優先度つきの割込み処理手法が必要になる．本論文で提案する軽量的な割込み処理手法では，割込みスレッドを用いずに，動的優先度に基づて割込み処理を行うことが可能である．

3. 提 案 手 法

本章では，軽量割込み処理を提案する．提案する軽量割込み処理では，タスクとハードウェアシステム全体で優先度を定義し，割込み処理の実行時期，動的な優先度の実現などが必要になる．

以下それぞれについて述べる．

3.1 割込み優先度の再定義

Linux では 2 つの独立した実行優先度空間を持つ．1 つはソフトウェアタスク優先度空間，もう 1 つはハードウェア割込み優先度空間である．ソフトウェアタスク優先度空間では，スケジューラがタスクの持つ優先度に従って，タスクを選択し実行権を渡す．ハードウェア割込み優先度空間は，ハードウェアが持つ論理回路（PIC や APIC など）により優先度が決まり，その実行は優先度に従って処理される．Linux カーネルはもともとデマンド駆動型のシステムであり，ソフトウェアタスクの実行は時間制限がなく，その優先度空間はハードウェア優先度空間より低い．ハードウェア割込みからの要求を迅速に処理するため，ソフトウェアタスクのコンテキストの下で，割込み処理が先行的に実行される．本処理方式はディスク I/O やネットワーク性能を要求するデータベース処理などで，高い効率をもたらす．

一方，実時間システムでは，ソフトウェアタスクの実行終了時間が規定され，タスクがデッドライン時間内に終了しなければならない．したがって，Linux を基に実時間システムを構築するには，ソフトウェア優先度空間とハードウェア割込み優先度空間との統合が必要になる．いい換えれば，システム全体で優先度の再定義が必要となる．

提案する軽量割込み処理では，以下に述べるとおり，割込みを，即時割込み，実時間割込み，遅延割込みの 3 種類に分ける．

(1)　**即時割込み**

即時割込みはシステム全体で最も高い優先度を持つ．本割込みは即時に処理されないと，システム実行中の他タスクの実行に影響を及ぼす．即時割込みには，2 タイプの割込みがある．1 つはタイマ割込みである．タイマ割込み処理によりマルチタスクが実現される．本処理が即時に行われないと，マルチタスク実行の公平性が保てないだけでなく，実時間タスクの実行開始時刻も正確にスケジュールできない．

即時割込みのもう 1 つのタイプは，特定デバイスからの割込みである．たとえば，フロッピーディスクの処理では，I/O データを保存するデバイスメモリが小さいため，割込みが発生してから，デバイスドライバは即座にデータをホストメモリに移動させる必要がある．

(2)　**実時間割込み**

実時間タスクが，ある割込み処理を待つとき，その割込みを実時間割込みと呼ぶことにする．実時間タスクが，デバイスを利用していて，デバイスハンドラの処理を待つために待ち行列に入るとき，当該デバイスの割込み処理は実時間タスクの優先度と同じ優先度で処理される必要がある．いい換えれば，本割込みの優先度は，実時間タスクの優先度を継承する．本割込み処理はスケジューラ処理の最初に実行されるか，低優先度タスクの実行中に実行される．

(3)　**遅延割込み**

遅延割込みとは，即時割込みと実時間割込み以外の割込みを指す．遅延割込みの優先度は実時間タスクより低く，通常 Linux タスクより高い．遅延割込みの処理は実時間タスクの実行を終了してから，通常 Linux タスクを実行する前に，スケジューラ処理の最後で実行される．

Linux カーネルのデバイス割込み処理はハード IRQ 処理とソフト IRQ 処理に分類される．ハード IRQ 処理部分は割込み時に即時に実行されなければならない処理部分であり，ソフト IRQ 処理部分は割込み処理の中で実行を遅延してもよい処理部分である．一方，本論文で取り扱っている即時割込み，実時間割込み，遅延割込みは，デバイス割込みと実時間タスクとの関連性を基に区別している．

3.2 割込み処理方式

ハードウェア割込み優先度空間とソフトウェアタスク優先度空間が統合されると，次の問題は割込み処理をどのように処理するかである．

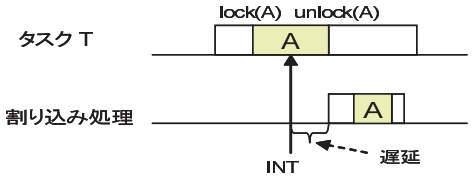

図 1　遅延する割込み処理
Fig. 1　Delayed interrupt processing.

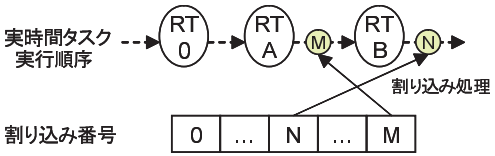

図 2　動的な割込み処理
Fig. 2　Dynamic interrupt treatment.

Linux カーネルでは，タスクと割込みハンドラが同じ資源をアクセスすることによるクリティカルセクションを処理するために，割込み禁止命令が使用される．多くの実時間システム[8),9)] では，タイマ割込みを利用して実時間タスクの実行開始時刻を指定している．しかし，カーネルに多くの割込み禁止区間（クリティカルセクション）があると，実時間タスクの実行を開始させるためのタイマ割込みの処理が遅延される．このように，カーネルクリティカルセクションが実時間タスクの実行開始を妨害する．

本問題が発生する要因は，割込み処理とタスク処理が同じ資源を同時にアクセスする可能性を排除するために割込みを禁止しているからである．本論文では，割込みを禁止しないが，割込み処理を遅延させることによりクリティカルセクション処理を実現する遅延割込み手法を提案する．

図 1 では，割込み処理とタスクが同じ資源をアクセスする可能性がある．提案手法では，すべての割込みが禁止されず，タスク T のクリティカルセクション A の実行中でも割込み INT が発生する．割込み INT が発生しても，タスク T がリソース A を開放されてから実行される．このように，割込み処理を遅延させることで，割込みを禁止にする必要性がなくなり，実時間タスクの開始時刻を指定しているタイマ割込みの処理が可能となる．

しかし，タイマ割込みが指定された時刻にカーネルに割り込んでも，実時間タスクが時刻どおりに実行を開始するとは限らない．なぜならば，タスクプリエンプションカウンタが 0 でない場合は実時間タスクがスケジューリングされず，実行が開始できないからである．これまで，いくつかの研究[10),11)] で，この問題を解決している．また，著者らは，論文 12) において，本問題点を解決する新しい手法を提案している．このため，本論文では本問題点は議論しない．

3.3　動的な割込み優先度の実現

3.2 節で提案した割込み処理方式では，即時割込みを遅延させない．このため，即時割込みがアクセスする資源を他のタスクや他の割込み処理がアクセスする場合，割込みを禁止してアクセスする必要がある．本節では，これ以降，実時間割込みと遅延割込みの処理に関する実行制御方式を述べる．

実時間タスクを，割込み処理に関連する実時間タスクと，割込み処理に関連しない実時間タスクの 2 つに分ける．実時間タスクがあるデバイスを使用するとき，そのタスクは，そのデバイスの割込み処理に関連する実時間タスクとなる．そのような実時間タスクは，デバイスを使用する前に，そのデバイスの割込み番号とデバイス名をカーネルに通知する．Linux カーネルでは割込み番号とその対応したデバイス名が proc システムによって明示されているため，プログラムで割込み番号とデバイス名を指定することは可能である．

今，あるデバイス D_M，D_N の割込み番号が，それぞれ，M 番，N 番と仮定する．実時間タスク A がデバイス D_M を使用し，そのデバイスからの応答を待つ場合，デバイス D_M の割込み処理は実時間タスク A と同じ実行優先度を持つ．また実時間タスク B がデバイス D_N を使用し，そのデバイスからの応答を待つ場合，デバイス D_N の割込み処理は実時間タスク B と同じ実行優先度を持つ．

実時間タスク A の実行優先度が実時間タスク B の実行優先度よりも高い場合，デバイス D_M および D_N の割込み処理を含めた優先度関係は，**図 2** に示すようになる．

複数のデバイスが 1 つの割込み番号を共有している場合，実時間タスクと関連するデバイスドライバだけを実行するには，その割込み番号とデバイス名を用いて特定することになる．また，複数の実時間タスクがネットワーク通信している場合，到着したパケットのポート番号などから受信するタスクを同定し，その実行優先度に従ってパケットを処理する必要がある．しかし，この処理はシステムに一定のオーバヘッドをもたらす．本論文ではこの問題を将来研究として残す．

実時間タスクが待ち行列に入り，スケジューラが呼びされると，スケジューラは，実時間割込みの存在を確認して実行する．実時間割込み処理の実行が終わると，待ち行列に入った実時間タスクが再び実行可能状態に変わっている可能性が高い．この場合，スケジューラは

再び現在の実時間タスクを選択して実行させる．このため，コンテキストスイッチは発生しない．

遅延割込みに対しては，優先度が実時間タスクより小さいため，すべての実時間タスクが終了してから実行される．カーネルスケジューラが通常 Linux タスクを選択して，実行させる前に遅延割込み処理を行う．

実時間割込みと遅延割込みの発生は通常 Linux タスクが実行中の場合，タスクのプリエンプトカウンタが 0 であれば，ただちに実行される．プリエンプトカウンタが 0 でない場合は *preempt_enable()* マクロによりカウンタが 0 になる時点で実行される．

4. 実　　装

本章では，Linux 2.6.20 カーネルに提案手法を実装する．

提案手法の実装では，Linux の実行モデルを継承する．すなわち，デバイスからの割込みは禁止しない．割込みが発生すると，カーネルは即時に割込みを受け付ける．割込み受付処理の概略を**図 3** に示す．図 3 において，割込みが発生すると，*do_IRQ()* 関数でその割込み番号を登録する．現在のタスクの優先度より割込み優先度が低い場合（*is_lower_priority_IRQ()*）で，かつ，タスクプリエンプトカウンタが 0 以上の場合，割込み処理は遅延される．

割込みが発生したとき，プリエンプトカウンタが 0 でない場合，つまり，タスクが割込み可能期間に入っていない場合，実時間割込みと遅延割込みの実行はタスクのプリエンプトカウンタが 0 となる時点まで遅延される．一般に，Linux では，タスクのプリエンプトカウンタを減らすには，マクロ *sub_preempt_count()* を用いる．したがって，現在のタスクのプリエンプトカウンタが 0 であるかどうかはこの時点で判断すればよい．概略を**図 4** に示す．マクロ *exist_higher_priority_IRQ()* では，割込みの優先度と実行中のタスク優先度が比較される．割込みの優先度が高い場合はその処理が実行される．

現在実行中のタスクが割込み処理と関連する実時間タスクである場合，実時間割込みと遅延割込みの実行はスケジューラの最初と最後に遅延される．この仮想コードは**図 5** のようになっている．図 5 では，Linux スケジューラの最初と最後に割込み実行コードが入る．ブロック A では，現在の実時間タスクのタスク構造体から，実時間割込み番号とデバイス名をもらって，その割込みがある場合は実行される．ブロック B では，遅延割込みの実行を表している．コードはさらにシンプルになり，遅延割込みがある場合はただちに実行される．

このように，提案手法の実装プログラムは，軽量かつ単純であり，Linux カーネルソースコードの変動も少ない．また，提案手法の実装は Linux カーネル排他区間の構造に変更がなく，リアルタイムシステムの実現においてタスク優先度逆転問題の発生がない．

```
do_IRQ {
  if (is_lower_priority_IRQ()  && (preempt_count() > 0))  {
      desc->chip->mask_ack();
      register_irq  |= (1 << irq);
       return 0;
  }
  ...
}
```

図 3　割込み受付ルーチン
Fig. 3　Interrupt accepting routine.

```
#define sub_preempt_count (val)
do {
      preempt_count()  −= (val);
      if  (preempt_count() == 0 && exist_higher_priority_IRQ()) {
        do_higher_priority_IRQ();
      }
}
```

図 4　遅延された割込みの処理
Fig. 4　Processing of delayed interrupt.

```
scheduler()  {
   if  (is_rt_task()) {                          A
      if  (get_announced_interrupt()) {
          do_rt_IRQ();
      }
   }
   ......
   if  (is_non_rt_task) {                        B
      do_pending_IRQ();
   }
}
```

図 5　スケジューラでの割込み実行
Fig. 5　Interrupt execution codes in scheduler.

5. 実　　験

提案した割込み処理方式を検証するため，Linux 2.6.20 カーネルに実装したシステムを用いて，以下の 4 つの実験を行う．まず，実時間タスクが正しくスケジュールされているかどうかを検証する．次に，提案手法によるシステムにおいて，実時間タスクの効率がどこまで改善されたかを検証する．そして，従来の割込みスレッド方式と比べて，提案システムの

表 2　実験環境
Table 2　Experiment environment.

名称	構成
CPU	Pentium4 3.0 GHz with 1 MB L2 cache
Memory	512 MB
HDD	80 GB（ext3 file-system）

タスクディスパッチとコンテキストスイッチの回数がどれだけ改善されるかを調べる．最後に，提案システムの実時間タスクのスタート最大遅延時間（以下，OS 遅延時間）を検証する．実験環境を**表 2** に示す．

5.1　実時間タスクの実行状態

提案システムにおいて，実時間タスクと割込み処理が設計どおりにスケジューリングされているかを調べるために，以下の実時間タスクと通常タスクの 2 つのタスクを実行したときの実行状態を検証する．実時間タスクの実行時間は 600 μs で，周期を 1 ms に指定する．通常タスクは，ネットワークから 2 GB のファイルをダウンロードする．このタスクにより，ネットワーク割込みが頻繁に生じる．

オリジナル Linux カーネルと提案カーネルでの実行の様子を**図 6** と**図 7** に示す．図 6 では IRQ 割込み処理の優先度が実時間タスク RT-Task より高いため，実時間タスクの実行はランダムに阻害される．一方，提案方式カーネルでは割込みの優先度が実時間タスクより低く，その実行は遅延され，実時間タスクの実行の予測性が増大する．

5.2　実時間割込みの効率

提案システムの実時間タスク応答性能を検証するため，ネットワーク関連の実時間タスクによる実験を行う．本タスクでは，周期的にネットワーク上のサーバに 1 つの文字を送り，送信が成功すると一周期が終了する．本実験では，実時間タスクが文字を送ってからネットワーク割込み処理されるまでの時間差を計測する．また，割込みスレッド方式と比較するために，文献 14) で提案した割込みスレッド化したシステムを用いる．割込みスレッド方式では，1 msec 周期で割込み処理を行う．

実時間タスクの周期回数を 10,000 回とし，10 回の実験を行った結果を**表 3** に示す．提案手法に基づいたシステムでは，平均遅延が 0.73 μs，割込みスレッド方式のシステムでは，遅延が約 1 ms となっている．割込みスレッド方式では，スレッドの優先度が静的であり，実行の遅延は周期時間に依存する．周期を短くすると，タスクディスパッチャとコンテキストスイッチの回数が増大し，かつ，無駄な実行が多くなり，空き CPU 時間が減る．一方，提案手法に基づいたシステムでは，実時間タスクが文字を送ると，ネットワーク割込みがタスクの優先度を継承して，すぐに実行されていることにより短い遅延時間で処理が行われていることが分かる．

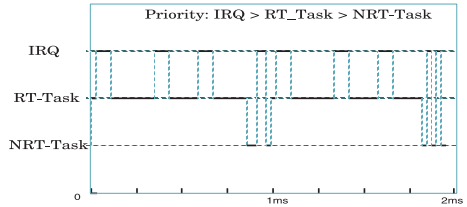

図 6　Linux カーネルでの実時間タスクの実行
Fig. 6　Executed real-time tasks in Linux.

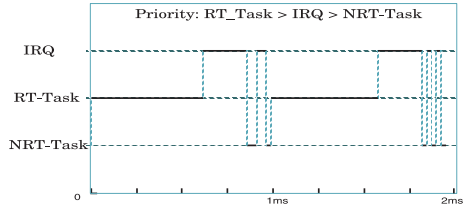

図 7　提案カーネルでの実時間タスクの実行
Fig. 7　Executed real-time tasks in proposed kernel.

表 3　実時間タスクから実時間割込みまでの遅延
Table 3　Latency from real-time tasks to real-time interrupts.

	スレッド方式	提案方式
Min	1.006 ms	0.71 μs
Max	1.010 ms	0.76 μs
Avg	1.007 ms	0.73 μs

5.3　割込みためのスケジューリング時間

システムのタスクディスパッチャとコンテキストスイッチ回数の増減を検証するために，提案システムと割込みベースにしたシステムで，タスクディスパッチャとコンテキストスイッチの回数を測る．実験では，テストタスクが周期的にサーバ側に 1 つの文字を送り，その文字の送信が成功すると，1 周期の実行が終わる．テストタスクを 10,000 周期実行し，そ

表 4　タスクディスパッチャ時間の比較
Table 4　Comparison of task dispatcher time.

	スレッド方式		提案モデル	
	回数	実行時間 (msec)	回数	実行時間 (msec)
Min	132,081	119.13	98,834	117.25
Max	142,078	124.75	100,077	119.94
Avg	132,121	122.93	100,067	118.13

表 5　タスクコンテキストスイッチ時間の比較
Table 5　Comparison of task context switch time.

	スレッド方式		提案モデル	
	回数	実行時間 (msec)	回数	実行時間 (msec)
Min	122,081	24.81	98,901	18.53
Max	142,116	25.43	100,077	19.35
Avg	122,133	25.03	100,069	18.82

れぞれのシステムのタスクディスパッチャとコンテキストスイッチの回数およびこれらにかかった時間を**表 4** と**表 5** に示す．

表 4 に示すとおり，提案手法に基づくシステムでは，タスクディスパッチャ時間は割込みスレッド方式より，平均実行時間で約 4%の減少，平均回数で約 25%減少している．これは提案した方式では，スケジューリング時間は実時間割込みの処理時間も含まれていて，測定した時間が増大したからである．

表 5 に示すとおり，提案手法に基づくシステムの平均コンテキストスイッチ時間および回数は 18.82 ms および 100,069 回であることに対して，スレッド方式は平均 25.03 ms および 122,133 回となっている．提案した方式は，実行時間で約 25%，回数で約 18%減少していることが分かる．

提案手法では，割込み処理のためのタスクディスパッチャとコンテキストスイッチはなくなる．一方，割込みスレッド方式では，実時間タスクが周期ごとに文字を送るたびに，スケジューラを呼び出し，ネットワークスレッドを起動させるため，そのタスクティスパッチャとコンテキストスイッチ回数が増大する．

5.4　OS 遅延時間

負荷時においても，OS 遅延時間がオリジナル Linux 2.6.20 カーネルよりも提案手法に基づいたカーネルが優れていることを示す．実時間タスクの周期を 1 ms とし，負荷として

表 6　OS 遅延時間
Table 6　OS latency.

	Linux		提案モデル	
	負荷なし	負荷あり	負荷なし	負荷あり
Min	0.85 μs	0.90 μs	0.41 μs	0.45 μs
Max	268.85 μs	1221.96 μs	110.98 μs	651.23 μs
Avg	12.23 μs	15.33 μs	3.23 μs	4.04 μs

表 7　実装コードのオーバヘッド（割込み処理を除く）
Table 7　Overhead of implementation (except interrupt processing).

	do_irq	sub_preempt_count	scheduler
Min	0.01 μs	0.01 μs	0.03 μs
Max	3.21 μs	1.73 μs	2.53 μs
Avg	1.10 μs	0.53 μs	0.04 μs

hackbench [15] ベンチマークを使う．本負荷ではカーネルに約 2,000 個のプロセスが生成され，頻繁に I/O 処理を行う．負荷がある場合と負荷がない場合の 2 つの結果を**表 6** に示す．提案手法はオリジナル Linux に比べて，最大遅延時間で 41%から 53%減少していることが分かる．

しかし，依然，OS 遅延時間は数百マイクロ秒のオーダで残っている．これは，3.2 節でも述べたように，タイマ割込みが指定された時刻にカーネルに割り込んでも，実時間タスクが時間どおりに実行を開始するとは限らず，タスクプリエンプションカウンタが 0 でない場合は実時間タスクがスケジューリングされず，実行が開始できないことによる．したがって，提案したモデルにおいて，負荷ある場合の最大遅延時間は依然として大きい．これまで，いくつかの研究[10),11)] で，この問題が解決されている．また，著者らは，論文 12) において，本問題点を解決する新しい手法を提案している．このため，本論文では本問題点は議論しない．

5.5　実装したコードによるオーバヘッド

4 章で述べたように，実装したカーネルでは，通常 Linux の 3 カ所を修正しており，それぞれ，*do_IRQ()*，*sub_preempt_count()*，*scheduler()* である．追加したコードの実行時間（割込み処理を除く）を**表 7** に示す．表に示すとおり，それぞれ追加したコードの最大実行時間は数マイクロ秒であり，システムへの影響は軽量であることが分かる．

6. ま と め

本論文では，単一 CPU における効率的な予測性のある実時間 Linux を構築するため，動的優先度を基にした割込み処理手法を提案した．本手法では，Linux のハードウェア割込み優先度空間とソフトウェアタスク優先度空間が統合される．割込みは優先度ベースに，即時割込み，実時間割込み，遅延割込み，の 3 種類に分類される．(1) 即時割込みは，緊急割込みであり，その優先度がシステムで最も高く，その処理がただちに実行される．(2) 実時間割込みは，実時間タスクが，ある割込み処理を待つときに，関連づけられる．当該割込みの優先度は，動的にその実時間タスクの優先度を継承する．実時間割込みの処理は現在のタスクの下で実行されるか，低優先度タスクの実行中に実行される．(3) 遅延割込みは残りの割込みを意味し，その優先度が実時間タスクより低く，通常タスクにより高い．その実行はすべての実時間タスクの実行が終了してから実行される．本手法で提案された割込み処理はスレッドとして実現せず，割込みの実行はカーネルクリティカルセクションが終了した時点で遅延される．

提案手法を Linux 2.6.20 カーネル上に実装し，既存システムと比較した．まず，提案システムにおいて，実時間タスクが正確にスケジュールされることを示した．次に，提案システムにおいて，実時間割込み処理が平均 0.73 μsec で開始されることを確認した．割込みスレッドによる処理方式では，割込みに関連する実時間タスクが割込み処理を待つ時間は，割込みスレッドの周期にほぼ等しくなり，本論文の実験では平均 1.007 ms かかる．本提案手法で実現した応答性能を割込みスレッド方式で実現しようとすると，マイクロ秒以下の周期を設定する必要があるが，これは非現実的である．また，提案手法と割込みスレッド方式のコンテキストスイッチ回数を比較した結果，提案手法のコンテキストスイッチの時間および回数はそれぞれ約 25%および約 18%減り，OS 遅延時間は割込み遅延させない手法より約半分まで短縮された．

本論文では，提案手法を Linux に適用したが，割込みを基にした一般的な実時間イベント型オペレーティングシステムに対しても，適応可能である．

参 考 文 献

1) Regehr, J. and Duongsaa, U.: Preventing Interrupt Overload, *Proc. 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, pp.50–58 (2005).
2) Luis, E.L., Pedro M. and de Niz, D.: Predictable Interrupt Management for Real-Time Kernels over conventional PC Hardware, *Proc. 12th IEEE RTAS* (2006).
3) TimeSys Corporation: TimeSys Linux. http://timesys.com/
4) Redhawk Linux, Real-Time Sfotware Environment. http://www.ccur.com/
5) Kleiman, S. and Eykholt, J.: Interrupts as threads, *ACM SIGOPS Operating Systems Review*, Vol.29, Issue 2, pp.21–26 (Apr. 1995).
6) Hildebrand, D.: An architectural overview of QNX, *Proc. USENIX Workshop on Micro-kernels and Other Kernel Architectures*, Seattle, WA, pp.113–126 (Apr. 1992).
7) Kopetz, H., et al.: Distributed Fault-Tolerant Real-Time Systems: The MARS Approach, *IEEE Micro*, Vol.9, No.1 (1989).
8) Srinivasan, B., Pather, S., Hill, R., Ansari, F. and Niehaus, D.: A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software, *Proc. IEEE RTAS* (June 1998).
9) Barabanov, M. and Yodaiken, V.: Introducing Real-Time Linux, *Linux Journal* (Feb. 1997).
10) Lee, J. and Park, K.-H.: Delayed Locking Technique for Improving Real-Time Performance of Embedded Linux by Prediction of Timer Interrupt, *Proc. 11th IEEE RTAS* (2005).
11) Shi, H., Cai, M. and Dong, J.: Interrupt Synchronization Lock for Real-time Operationg System, *Proc. 6th IEEE CIT* (2006).
12) Dai, M., Matsui, T. and Ishikawa, Y.: A Light Lock Management Mechanism for Optimizing Real-Time and Non-Real-Time Performance in Embedded Linux, To appear at *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing* (2008).
13) 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).
14) 戴，石川：予測可能リアルタイムカーネルの設計と実装，情報処理学会研究報告 2005-OS-99, pp.121–126 (2005).
15) OSDL. http://devresources.linux-foundation.org/craiger/hackbench/

(平成 20 年 7 月 30 日受付)
(平成 20 年 11 月 30 日採録)

戴　　毛兵（正会員）

2003 年電気通信大学情報理工学科卒業，2005 年東京大学大学院情報理工学研究科博士前期課程修了．同年より同大学院情報理工学研究科コンピュータ専攻博士後期課程在学中．実時間オペレーティングシステムに関する研究に従事．

石川　　裕（正会員）

1987 年慶応義塾大学大学院工学研究科電気工学専攻博士課程修了．工学博士．同年電子技術総合研究所入所．1993 年技術研究組合新情報処理開発機構出向．2002 年より東京大学大学院情報理工学系研究科コンピュータ科学専攻教授．クラスタ・グリッドシステムソフトウェア，高信頼システムソフトウェア開発技術，実時間分散システム，次世代高性能コンピュータシステム等に興味を持つ．