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　75–86 (Mar. 2009)

Linux カーネル用 リアルタイムスケジューリングモジュール

加藤真平[†1]　山崎信行[†1]

本論文では，与えられた時間制約のもとで CPU 資源を最大限に使用可能なリアルタイム Linux の実現を目指す．まず，高負荷状態における十分なリアルタイム性と予測可能性の提供を目的として，固定優先度の利点と動的優先度の利点を兼ね備えたスケジューリングアルゴリズムを提案する．提案アルゴリズムは理論的な性能の面で従来の固定優先度アルゴリズムと同等以上であることが保証され，Linux カーネルのスケジューラ実装に対する親和性も高いという利点がある．次に，拡張性の高いリアルタイム Linux の実現のために，カーネルの修正を最小限に抑え，新規のスケジューリングアルゴリズムをカーネルモジュールとして組み込むことを可能にするフレームワークを提案する．シミュレーションによる評価の結果，提案アルゴリズムは従来の固定優先度アルゴリズムよりも約 10∼15%高い CPU 使用率でもリアルタイム性を保証できた．また，実機による評価の結果，開発したカーネルモジュールを組み込むことで，ネイティブな Linux に比べて高負荷状態における低優先度タスクのデッドラインミス率を最大で約 70∼90%削減することができた．

Real-time Scheduling Module for Linux Kernel

Shinpei Kato[†1] and Nobuyuki Yamasaki[†1]

The goal of this paper is to develop a real-time Linux that is capable of utilizing sufficient CPU resource under given timing constraints. We first present such a scheduling algorithm that takes advantages of fixed-priority and dynamic-priority, inder to provide sufficient real-time capability as well as predictability in high-load situations. The algorithm is at least as effective as a traditional fixed-priority algorithm in terms of theoretical schedulability, and its design is also suitable for the scheduler implementation of the Linux kernel. We then propose such a framework that enables new scheduling algorithms to be installed as kernel modules without major modification, for the achievement of a scalable real-time Linux. Simulation results show that the algorithm guaranteed all jobs to be schedulable at 10∼15% higher CPU utilization than a traditional fixed-priority algorithm. In addition, experimental results in real environments show that the deadline miss ratio of low-priority tasks in high-load situations was reduced at most 70∼90% by the developed kernel module, as compared to the native Linux.

1. はじめに

組み込みシステムの高機能化にともない，その基盤となるオペレーティングシステム（OS）には様々な機能および性能が求められている．とりわけ，マルチメディア処理や多様な I/O デバイスを必要とする組み込みシステムでは，ソフトウェアライブラリやデバイスドライバの利用が不可欠であり，従来の専用組み込み OS によってすべての機能を提供することは不可能に近い．このような背景から，組み込みシステムの分野では汎用 OS である Linux に対する期待が高まっている．Linux を利用することでソフトウェアライブラリやデバイスドライバ等のソフトウェア資産が再利用可能になり，さらには技術者の確保も容易になることから開発コストを大きく削減できることが期待できる．しかしながら，Linux はもともと汎用システム向けに設計された OS であり，組み込みシステムに求められるリアルタイム性や応答性に関しては改善の余地が残されている．そのため，多くの企業や研究機関によって，これらの時間制約を満たせるようなリアルタイム Linux の開発が行われている[9)–11)]．

リアルタイム性には大別してハードリアルタイム性とソフトリアルタイム性がある．ハードリアルタイム性とは時間制約の破綻が重大な損害を引き起こす可能性のある性質であり，ソフトリアルタイム性とはある程度の時間制約を維持できればよい性質を指す．Linux カーネルがもともと時間制約を考慮した設計になっていないことから，本論文では主にソフトリアルタイムシステムを対象とする[⋆1]．

マルチメディアシステムに代表されるソフトリアルタイムシステムでは，アプリケーションの処理量が比較的大きいため，与えられた時間制約のもとで CPU 資源を最大限に使用できることが求められる．時間制約下での CPU 使用率の上限は，しばしば OS のスケジューリングアルゴリズムに依存する．理論的に CPU 資源を 100%使用可能なアルゴリズムとして Earliest Deadline First（EDF）[6)] が広く知られているが，あるタスクのデッドラインミスがそのほかのタスクのデッドラインミスを連鎖的に誘発してしまい，それらのデッドラ

†1 慶應義塾大学
Keio University

⋆1 CPU 使用率に十分な余裕があれば，ハードリアルタイムシステムにおいても利用可能であることに注意されたい．

インミスを起こすタスクを事前に解析することも難しいため，リアルタイムシステムにとって重要な予測可能性が低いという欠点がある．また，EDF では動的に優先度を付け替える必要があるので，実行時にタスクキュー操作のためのオーバヘッドが発生する．そのため，多くのリアルタイム Linux や商用リアルタイム OS のスケジューラは固定優先度アルゴリズムを採用している．固定優先度アルゴリズムは実装が簡単であるうえに，優先度の低いタスクが優先度の高いタスクのデッドラインミスを誘発することはなく，必ず優先度の低いタスクが先にデッドラインミスを起こすことが保証される[2]．しかしながら，高負荷状態におけるリアルタイム性の低下が問題となる．最適な固定優先度アルゴリズムである Rate Monotonic（RM）[6] を用いたとしても，最悪の場合には CPU 使用率が 69%を超えると時間制約が破綻してしまう可能性がある．高品質なリアルタイムシステムを構築するためには，固定優先度アルゴリズムの簡潔さと予測可能性を継承しながらも高負荷状態において十分なリアルタイム性を提供可能なスケジューリングアルゴリズムが必要である．

従来のリアルタイム Linux では，スケジューラ等のリアルタイム性を改善する場合に Linux カーネルに独自の修正を加えている．このアプローチは目的に応じて性能を最大限に改善できるという点では妥当であるが，将来的な拡張性に問題がある．Linux カーネルはオープンソースとして日々改良が重ねられているソフトウェアであり，マイナバージョンの更新によってもカーネルに大きな修正が施されることも少なくない．たとえば，カーネル 2.6 のマイナバージョンが 22 から 23 に向上した際に Completely Fair Scheduler（CFS）という機能が加わり，スケジューラの実装が大幅に変更された．このような場合に，カーネルに直接修正を加えるアプローチでは再度修正部分の設計を見直す必要があり，開発コストが高い．最新の Linux カーネルに柔軟に対応するためには，カーネル自体の修正を最小限に抑えることが求められる．

本論文では，与えられた時間制約のもとで CPU 資源を最大限に使用可能なリアルタイム Linux の実現を目指す．まず，高負荷状態における十分なリアルタイム性と予測可能性の提供を目的として，固定優先度の利点と動的優先度の利点を兼ね備えたスケジューリングアルゴリズムを提案する．提案アルゴリズムは各タスクに固定優先度を与えるが，従来の固定優先度アルゴリズムとは異なり，スケジューラの起動時に各ジョブの残り実行時間とデッドラインを比較して，デッドラインミスが予想されるジョブには動的に最高優先度を与えてデッドラインミスを回避する．それ以外の場合は固定優先度アルゴリズムとして振る舞うので，理論的な性能は固定優先度アルゴリズムと同等以上であることが保証され，Linux のスケジューラ実装に対する親和性も高いという利点がある．次に，拡張性の高いリアルタイム Linux の実現のために，カーネルの修正を最小限に抑え，新規のスケジューリングアルゴリズムをカーネルモジュールとして組み込むことを可能にするフレームワークを提案する．最終的に，最新の Linux カーネル 2.6.25 に 1 行の修正を加えるだけで開発したモジュールを組み込むことができ，高負荷状態におけるリアルタイム性を大きく改善できたことを示す．

本論文の構成は以下のとおりである．次章ではリアルタイム Linux の関連研究について述べる．3 章では本論文で提案するスケジューリングアルゴリズムの設計と解析を行う．4 章では提案アルゴリズムを実装したカーネルモジュールの設計と実装を詳述する．5 章では開発したカーネルモジュールを組み込んだリアルタイム Linux の有効性をネイティブの Linux と比較して評価する．最後に 6 章で本論文の結論と今後の予定について述べる．

2. 関 連 研 究

RT-Linux [10] では，Linux のスケジューラやプロセス間通信機能を修正して，Linux カーネル自体を優先度の低い 1 つのタスクとしてスケジューリングする．リアルタイムタスクと Linux を切り分けているため，すべてのリアルタイムタスクはカーネルレベルで実行されることになり，カーネル自体の応答性が低下する可能性があるという問題がある．また，リアルタイムタスクはカーネルモジュールとして作成する必要があるので，従来よりもアプリケーションの作成が難しくなるという欠点もある．

ART-Linux [11] はカーネル自体にリアルタイム性を持たせるように設計されたリアルタイム Linux である．RT-Linux とは異なり，従来の Linux プログラミングとの互換性が高く，ユーザ空間でもリアルタイムタスクを実行できるので，安全性が高いことやプログラミングが簡単になるといった利点がある．しかしながら，リアルタイムタスクには単純に固定優先度が与えられるだけなので，リアルタイムタスクが複数存在するシステムにおいて CPU 使用率が高くなってしまうと，最高優先度タスクの応答性は確保できても全体的なリアルタイム性の低下は防ぐことができない．

TimeSys Linux [9] は固定優先度スケジューリングに加えて，Resource Kernel [8] や Portable RK [7] といった資源予約技術を導入しており，高負荷状態においても任意のタスクのリアルタイム性を保証することができる．ただし，システム全体のリアルタイム性は固定優先度スケジューリングの性質に依存する．また，拡張機能をカーネルモジュールとして提供しているため，システム設計者は必要に応じて機能を追加，削除することができる．

本論文では，スケジューリングアルゴリズム等のプリミティブな機能もカーネルモジュールとして提供できるフレームワークを考える．そして，最新の Linux カーネルに実装され

ている固定優先度スケジューラや応答性の高い周期実行を最大限に活用しつつ，カーネルモジュールを使って効果的に性能を改善することで，リアルタイム性と拡張性を兼ね備えたリアルタイム Linux の実現を目指す．

3. スケジューリングアルゴリズム

本章では，固定優先度アルゴリズムの簡潔さと予測可能性を継承しながらも高負荷状態において高いリアルタイム性を提供可能なスケジューリングアルゴリズムを設計し，リアルタイム性を保証するためのスケジュール可能性判定の解析を行う．

3.1 設　　計

まず，固定優先度アルゴリズムの問題点を理解するために，固定優先度アルゴリズムによるデッドラインミスを考える．**図 1** は周期の短いタスクに高い優先度を与える Rate Monotonic（RM）を用いて，実行時間が同じで周期が異なる 3 つのタスク T_1，T_2，T_3 のスケジューリングを行う様子を示している．この例では，優先度の低い T_3 は，優先度の高い T_1 と T_2 にブロックされ，結果としてデッドラインミスが発生してしまう．そこで，マルチプロセッサシステムにおけるリアルタイムスケジューリングアルゴリズムとして近年注目されている Earliest Deadline Zero Laxity（EDZL）[3] の概念を導入する．

Zero Laxity（ゼロ余裕時間）とは，Laxity（余裕時間）が 0 になった状態を指す．タスクの余裕時間が負の値になった場合，デッドラインまでに実行完了できないことから，EDZL は Zero Laxity 状態になったタスクに最高優先度を与えることで，高負荷状態におけるリアルタイム性の改善を実現している．ここで，タスク T_i の時刻 t における余裕時間 $x_i(t)$ は，T_i のデッドライン d_i と残り実行時間 e_i を用いて式 (1) によって定義される．

$$x_i(t) = d_i - (t + e_i) \tag{1}$$

この概念を RM に取り入れたアルゴリズムを RM Zero Laxity（RMZL）と定義する．先の 3 つのタスクを RMZL によってスケジュールする様子を**図 2** に示す．T_3 の余裕時間が 0 になったときに，最高優先度を与えることでデッドラインミスを回避できることが分かる．しかしながら，T_3 が最高優先度になったことで T_1 の実行がプリエンプションされることに注目されたい．Linux や多くの商用リアルタイム OS が採用する固定優先度アルゴリズムでは，スケジューラが起動される条件は以下の 2 通りに限定される．

- 現在のタスクより優先度の高いタスクの周期が開始された場合
- 現在のタスクが実行可能ではなくなった場合（実行完了やスリープ）

よって，RMZL をそれらの OS に実装する場合には，余分にスケジューラを起動しなくてはならない．また，Zero Laxity 状態になるタイミングによっては細粒度のタイマ起動が必要になってしまうことや，余裕時間が 0 になってから最高優先度を与えてもタスクの実行完了はデッドラインと同時刻であり実システムでは危険がともなうことも問題となる．

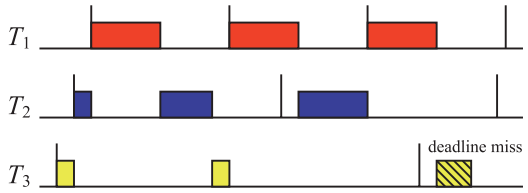

図 1　RM スケジューリング
Fig. 1　RM scheduling.

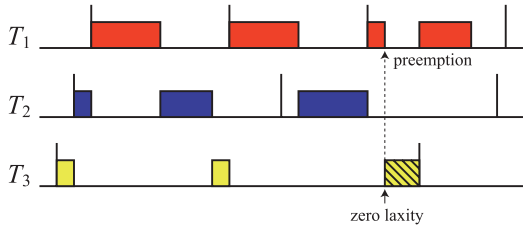

図 2　RMZL スケジューリング
Fig. 2　RMZL scheduling.

本論文では，Zero Laxity の概念を拡張した **Critical Laxity**（危険余裕時間）を提案する．時刻 t_s を RM スケジューリングにおける任意のスケジューリングポイント[*1]とする．ここで，RM は固定優先度アルゴリズムであり，スケジューリングポイント t_s の出現は上述の 2 通りに限定されることに注意されたい．時刻 t_s における最高優先度タスク T_{hp} の残り実行時間を e_{hp} とすると，条件式 (2) が成り立つ場合にタスク T_i の余裕時間 $x_i(t_s)$ を Critical Laxity と定義する．

$$x_i(t_s) < e_{hp} \tag{2}$$

以降，タスクの余裕時間が Critical Laxity であることを危険状態という．RM スケジューリングにおいて危険状態になったタスクは，その時点で最高優先度を与えない限り，デッド

*1 スケジューラ起動時刻．

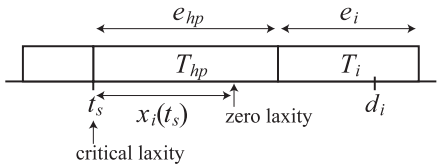

図 3　Critical Laxity
Fig. 3　Critical Laxity.

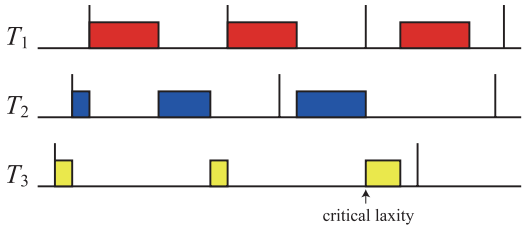

図 4　RMCL スケジューリング
Fig. 4　RMCL scheduling.

ラインミスを回避することはできない．**図 3** の例を用いて説明する．タスク T_i が時刻 t_s に危険状態になったとする．$x_i(t_s) < e_{hp}$ であるため，T_{hp} の実行が完了する前に T_i の余裕時間は必ず 0 になる．よって，T_i は必ずデッドラインミスを起こすことになる．しかしながら，危険状態になった T_i に最高優先度を与えれば，T_i はデッドラインミスを起こすことはない．本論文では，RM スケジューリングにおいて危険状態になったタスクに最高優先度を与えるアルゴリズムを RM Critical Laxity（RMCL）と定義する．

図 4 は RMCL を用いて，先の 3 つのタスクをスケジュールする様子を示している．T_3 が危険状態になった時点で最高優先度を与えることでデッドラインミスを回避できており，かつ余分なスケジューラ起動を必要としていないことが分かる．実際，RMCL によるスケジューラ起動回数は RM によるそれと同じである．また，各スケジューリングポイントで危険状態になったタスクが存在するかどうかを確認する以外は，RM と同じ振舞をする．

RMCL の最大の特徴は，スケジューラ起動やランキュー操作等のコストは RM とほぼ同等であり，かつ RM と同等かそれ以上の CPU 使用率でも時間制約を満たすことができる点である．すなわち，理論的に RM によって時間制約を満たせるタスクセットは RMCL によっても必ず時間制約を満たせることが保証される．RM スケジューリングでは危険状態に

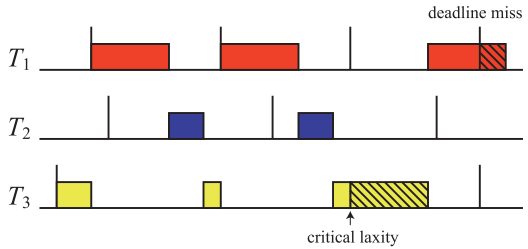

図 5　Critical Laxity によるデッドラインミス
Fig. 5　Deadline miss due to Critical Laxity.

なったタスクが必ずデッドラインミスを起こしてしまう事実から，このことは自明である．

次に，危険状態になったタスクへの最高優先度割当ての制限を考える．固定優先度アルゴリズムでは，優先度の低いタスクが先にデッドラインミスを起こすことが知られている[2]．よって，危険状態になるタスクは優先度の低いタスクということになる．RMCL スケジューリングでは，危険状態になったタスクに最高優先度が与えられるが，この優先度付けによって元々優先度の高いタスクがデッドラインミスを起こすことも考えられる．**図 5** の例では，T_3 の実行中に優先度の高い T_1 の周期が始まったが，T_3 が危険状態であるために T_3 に最高優先度を与えてスケジューリングを行っている．しかしながら，これが原因となり T_1 も危険状態となり，結果としてデッドラインミスを起こしてしまっている．先に述べたように，固定優先度アルゴリズムの利点の 1 つは，優先度の低いタスクからデッドラインミスを発生するという予測可能性がある点である．よって，本論文ではこの性質を保つために，危険状態になって最高優先度が与えられるのは，そのほかのタスクが危険状態にならない場合に限定する．この制限により，理論的な性能低下をいっさい起こすことなく，効果的にリアルタイム性を改善することが可能となる．

3.2　スケジューリング解析

システム全体で一定のリアルタイム性を保つためには，スケジュール可能性判定によって実行可能なタスクセットをアドミッションコントロールする必要がある．本論文では，伝統的な Response Time Analysis（RTA）[1] を応用して RMCL アルゴリズムのスケジューリング解析を行う．以降，タスク T_i の周期を p_i，（最悪）実行時間を c_i と表記する．すなわち，T_i は p_i 時間ごとに c_i の実行時間を持つジョブを生成するものとする．また，相対デッドラインは p_i とする．

固定優先度スケジューリングでは，各タスク T_i は自分よりも高い優先度を割り当てられ

たタスクからのみ干渉を受ける．この性質を利用して RTA では公式 (3) を用いて T_i のジョブの最悪応答時間 R_i を求める．ここで，表記の簡略化のため，すべての i に対して優先度の大小関係は $T_i \geq T_{i+1}$ であるものとする．

$$R_i = \sum_{k=0}^{i-1} \left\lceil \frac{R_i}{p_k} \right\rceil c_k + c_i \tag{3}$$

すべての T_i に対して $R_i \leq p_i$ が成り立っていればタスクセットはスケジュール可能であることが保証され，逆に 1 つ以上の T_i が $R_i > p_i$ となるようであればデッドラインミスが発生する可能性がある．

RMCL スケジューリングでは，あるタスク T_i が $R_i > p_i$ となった場合でも Critical Laxity による最高優先度割当てによって T_i のデッドラインミスを回避することが可能である．デッドラインミスが発生する状況は，T_i に最高優先度を与えたことでほかのタスク T_j が危険状態になってしまう場合に限られる．また，T_i が危険状態になって最高優先度が与えられた場合，T_i の実行は早まることがあっても遅れることはない．よって，影響を受けるのは T_i よりも優先度の高いタスクのみであることに注意されたい．この性質を利用して RMCL のスケジュール可能性判定式を導き出す．

まず，優先度の低いタスクから順に RTA の公式 (3) を用いて最悪応答時間を算出する．すべての T_i に対して $R_i \leq p_i$ であればタスクセットがスケジュール可能であることは自明である．ここで，T_i が $R_i > p_i$ になったとする．Critical Laxity による最高優先度を考えない場合に T_i が周期 p_i 以内に消費しきれない最悪実行時間 W_i は式 (4) で表せる．

$$W_i = \max\{R_i - p_i, c_i\} \tag{4}$$

Critical Laxity によって T_i に最高優先度が割り当てられると，最悪の場合にはすべての T_j（$j < i$）の実行が W_i 時間遅れることになる．その結果，$R_j > p_j$ となる T_j が存在すればタスクセットはデッドラインミスを発生する可能性がある．いいかえると，すべての T_j が式 (5) を満たしていればタスクセットはスケジュール可能であることが保証される．

$$R_j + W_i \leq p_i \tag{5}$$

4. スケジューリングモジュール

本章では，RMCL を実装したカーネルモジュールである **Re**al-time **sch**eduling（Resch）モジュールの設計と実装を行う．まず，カーネルの修正を最小限に抑え，新規のスケジューリングアルゴリズムをカーネルモジュールとして組み込むためのフレームワークを述べる．次に，Resch モジュールに対する RMCL の実装を述べる．本モジュールは http://www.ny.ics.keio.ac.jp/~shinpei/t-rex/ よりダウンロード可能である．

4.1 Linux スケジューラの概要

Linux カーネル 2.6.23 以降，スケジューラの実装は 3 つのスケジューリングクラス（rt_sched_class，fair_sched_class，idle_sched_class）に分けられている．クラスは構造体として定義され，各クラス専用のタスク選択関数やタスクキュー操作関数等へのポインタがメンバとなっている．このクラス化により，リアルタイムタスクのスケジューリング（rt_sched_class）をそのほかのタスクのスケジューリング（fair_sched_class，idle_sched_class）から切り離して実装することができるようになった．本論文では，クラス化を有効利用するために最新のバージョン 2.6.25 を対象とした．

Linux カーネルでは，スケジューリングは一括して schedule 関数で行われている．schedule 関数の大まかな内容は以下のとおりである．

(1)　プリエンプションの無効化やランキューのロック獲得
(2)　現在実行中のタスク（prev）の状態チェック
(3)　次に実行するタスク（next）の選択
(4)　prev から next へコンテキストスイッチ
(5)　プリエンプションの有効化やランキューのロック解放

スケジューリングアルゴリズムの仕事は次に実行するタスクを決定することなので，それに関連するのはステップ (3) だけである．Linux のスケジューラ実装では，ステップ (3) は pick_next_task 関数として用意されている．その疑似コードを**図 6** に示す．

pick_next_task 関数は，まずリアルタイムスケジューリングクラスである rt_sched_class の pick_next_task 関数を実行する．実行可能なリアルタイムタスクが存在しない場合には返り値 p は NULL になり，次のスケジューリングクラスの pick_next_task 関数を実行する．3 つのスケジューリングクラスは next メンバによって，rt_sched_class → fair_sched_class → idle_sched_class の順にリストされており，つねに rt_sched_class が優先して実行される．ここで，各クラスの pick_next_task メンバは関数ポインタであり，rt_sched_class の pick_next_task はカーネル内に別途実装された pick_next_task_rt 関数を参照していることに注意されたい．実際には，pick_next_task_rt 関数が実行可能なタスクの中で固定優先度が最も高いタスクを選択することになる．

rt_sched_class では，スケジューリングアルゴリズムとして SCHED_FIFO と SCHED_RR が用意されている．SCHED_FIFO と SCHED_RR の違いは，同一優先度のタスクをタイム

```
pick_next_task() {
    class = rt_sched_class;
    for ( ; ; ) {
        p = class->pick_next_task();
        if (p)
            return p;
        class = class->next;
    }
}
```

図 6　pick_next_task 関数
Fig. 6　pick_next_task function.

スライスごとに切り替えるか否かであり，固定優先度アルゴリズムいう点では等価である．これらのアルゴリズムでは，現在のタスクより優先度の高いタスクの周期が開始された場合もしくは現在のタスクが実行可能ではなくなった場合にだけスケジューラが起動され，pick_next_task_rt 関数が実行される．RMCL のようにスケジューリングポイントがこれら 2 通りに限定されるアルゴリズムであれば，pick_next_task_rt 関数を置き換えるだけで簡単に組み込むことができる．

4.2　Linux カーネルの修正

rt_sched_class の pick_next_task メンバは関数ポインタなので，この関数ポインタの参照先を変更できれば，独自の pick_next_task 関数をカーネルモジュール内に用意して外部から Linux カーネルに組み込むことができる．Linux カーネルでは，rt_sched_class は静的な構造体として以下のように宣言されている．

```
const struct sched_class rt_sched_class
```

よって，const を取り除いてシンボルをエクスポートすれば外部のカーネルモジュールから pick_next_task 関数を上書き可能になる．本論文では，以下の 1 行の修正を加えた Linux カーネルを **Linux+** と呼ぶ．

```
struct sched_class rt_sched_class
```

Linux+ではこれ以上の修正をカーネルに施すことはない．Linux カーネルの今後のバージョンで上記の変更が行われることがあれば，カーネルにいっさい修正を施すことなく Resch モジュールを利用できるようになる．現在，この変更が可能かどうか Linux 開発コミュニティと議論中である．

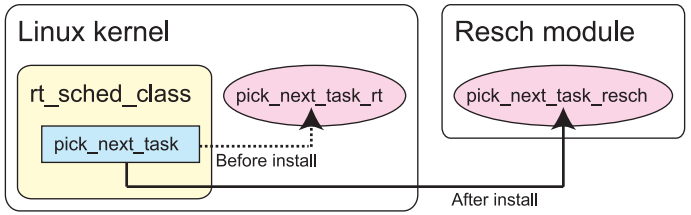

図 7　Resch モジュールのインストール
Fig. 7　Install of Resch module.

4.3　フレームワークの設計の実装

本フレームワークでは，新規のスケジューリングアルゴリズムを **pick_next_task_resch** 関数として Resch モジュール内に実装する．そして，Resch モジュールのインストール時にカーネルの pick_next_task_rt 関数と置き換えることでスケジューリングアルゴリズムを Linux に組み込むことができる．Resch モジュールのインストールの様子を**図 7** に示す．前節のカーネル修正により，外部のカーネルモジュールから rt_sched_class の pick_next_task 関数ポインタの値を上書き可能になっているので，ポインタの参照アドレスを Linux カーネル内の pick_next_task_rt 関数からモジュール内の pick_next_task_resch 関数に書き換える．また，モジュールをアンインストールする際にポインタの参照アドレスを pick_next_task_rt 関数のアドレスに戻すことで，Linux 従来のスケジューリングアルゴリズムに戻すことができる．

Resch モジュールでは，スケジューリングのために以下の 4 つのカーネル関数が用意されている．

- int resch_init(long priority)
- int resch_exit(void)
- int resch_run(long period, long timeout)
- int resch_yield(void)

resch_init 関数は，呼び出し元のアプリケーションプログラム（タスク）を Resch モジュールが管理するタスクキューに挿入する．引数にはタスクの優先度（priority）を指定する．Resch モジュールによる管理を終了したい場合は resch_exit 関数を実行する．resch_run 関数は，実際に Resch モジュールによるスケジューリングを開始する．引数にはタスクの実行周期（period）と何マイクロ秒後に最初の周期を実行開始するか（timeout）を指定する．

```
struct rt_data {
    long period;
    long deadline;
    long wcet;
    long remaining_time;
}
```

図 8　rt_data 構造体
Fig. 8　rt_data structure.

resch_yield 関数は，ほかのタスクに CPU を譲る関数であり，周期タスクは各周期の最後に必ず実行する必要がある．

Resch モジュールはカーネル空間で動作し，タスクは一般的にユーザ空間で動作するので，ユーザプログラムからカーネルモジュールの関数は直接実行はできない．そこで，Resch モジュールを仮想的なキャラクタデバイスとして実装（/dev/resch）し，上述した API 関数が/dev/resch に write システムコールを発行することでカーネルモジュールを操作できるように設計する．Resch モジュールのインストール時に write システムコールによって resch_write 関数が実行されるように登録しておく．たとえば，以下のように resch_init 関数を実行したとする．

```
resch_init(99);
```

実際には resch_init 関数は以下のコードを実行する．

```
((long*)data)[0] = ID_INIT;
((long*)data)[1] = 99;
fd = open("/dev/resch", O_RDWR);
write(fd, data, sizeof(data));
```

ID_INIT は関数の種類を示す ID 値である．Resch モジュールの resch_write 関数では ID の値をチェックして，それに対応した処理を行う．

リアルタイムスケジューリングを行うためには，各タスクにデッドラインや実行時間等のデータを関連付ける必要がある．Resch モジュールでは，これらのデータを rt_data 構造体として定義した．**図 8** に代表的なメンバを示す．wcet は最悪実行時間である．Resch モジュールはタスクの実行時間を追跡していき，resch_yield 関数が実行されたときの実行時間がそれ以前の周期での最悪実行時間（wcet）より大きかったら wcet の値を更新する．

```
/* t is current time. */
pick_next_task_resch() {
    T_hp = pick_next_task_rt();
    for (each task T_i) {
        if (d_i − t + e_i < e_hp &&
            d_hp − t + e_hp ≥ e_i)
                return T_i;
    }
    return T_hp;
}
```

図 9　RMCL アルゴリズムの疑似コード
Fig. 9　Pseudo code of RMCL algorithm.

remaining_time はタスクの wcet までの残り実行時間を指す．wcet と remaining_time はスケジューリングアルゴリズムによっては必要ない情報である場合もあるが，リアルタイム処理という点では有益な情報であるため，本論文ではこれらの情報をフレームワークに組み込むことにする．

rt_data 構造体は Resch モジュール内で宣言されているため，各タスクに割り当てるためには Linux カーネルのタスク構造体（task_struct）と関連付ける必要がある．Resch モジュールでは，実装対象を Linux のタイムスライス情報を必要としないアルゴリズムに制限し，task_struct 構造体から参照可能な long 型の rt.time_slice メンバを rt_data 構造体へのポインタを格納する変数として使用する．具体的には，rt_data 構造体と task_struct 構造体の関連付けは，resch_init 関数が実行されたときに以下のように行う（適宜省略）．

```
rt_data *rt = kmalloc(sizeof(*rt));
current->rt.time_slice = (int)rt;
```

最後に，ユーザが指定した優先度と SCHED_FIFO を引数として，Linux カーネルの sched_setscheduler 関数を実行すれば，タスクは RMCL によってスケジューリングされるようになる．

本フレームワークは新規のスケジューリングアルゴリズムを Linux カーネルに簡単に組み込むことを目的としている．適用範囲としては，Linux のタイムスライス情報を必要とせず，かつスケジューリングポイントが現在のタスクより優先度の高いタスクの周期開始時も

しくは現在のタスクの実行完了時およびスリープ時に限られるアルゴリズムであれば本フレームワークを利用可能である．我々の知る範囲では，多くのアルゴリズムがこれらの制限内で実装可能であるため，本フレームワークの拡張性は高いと考える．

4.4 RMCL の実装

本節では，RMCL の pick_next_task_resch 関数における実装を述べる．**図 9** にその実装の疑似コードを示す．まず，Linux カーネルに実装されている pick_next_task_rt 関数を呼んで最高優先度タスク T_{hp} を取得する．次に，そのほかのタスク T_i に対して危険状態であるかどうかを調べる．そして，T_i が危険状態であり，かつ T_i に最高優先度を与えた場合に T_{hp} が危険状態にならなければ T_i を選択する．もし，危険状態であるタスクが存在しなければ単純に最高優先度タスクである T_{hp} を選択する．

5. 評　　価

本章では，提案アルゴリズムである RMCL の理論的な性能評価と Resch モジュールを組み込んだリアルタイム Linux の実環境における性能評価を行う．

5.1 シミュレーションによる評価

シミュレーションにより多様なタスクセットに対する RMCL アルゴリズムと従来の RM アルゴリズムのスケジュール成功率（Success Ratio）[⋆1]を計測し，リアルタイム性を保証可能な CPU 使用率を評価した．

5.1.1 シミュレーション環境

RM では CPU 使用率が 69%を超えるとデッドラインミスを起こす場合があるので，シミュレーションでは CPU 使用率 70% から 100% までのスケジュール成功率を評価した．CPU 使用率 U に対して 1,00000 個のタスクセットを作成した．タスクの周期 p_i は $1 \sim 30$ ms を想定して $[100, 3000]$ の範囲で無作為に決定した．固定優先度アルゴリズムの性能が個々の CPU 使用率やタスク数に依存するため，個々のタスクの CPU 使用率 u_i は $[0.1, 1.0]$ の範囲と $[0.1, 0.5]$ の範囲で一様分布によって決定する 2 通りを評価した．タスク T_i の実行時間は $c_i = u_i p_i$ で算出できる．

5.1.2 シミュレーション結果

図 10 は個々のタスクの CPU 使用率が $0.1 \sim 1.0$ の場合のスケジュール成功率を示している．RM と RMCL は与えられたタスクセットを実際にスケジューリングした場合のスケジュール成功率であり，RM-test と RMCL-test は RTA を用いたスケジュール可能性判定でアドミッションコントロールした場合のスケジュール成功率である．

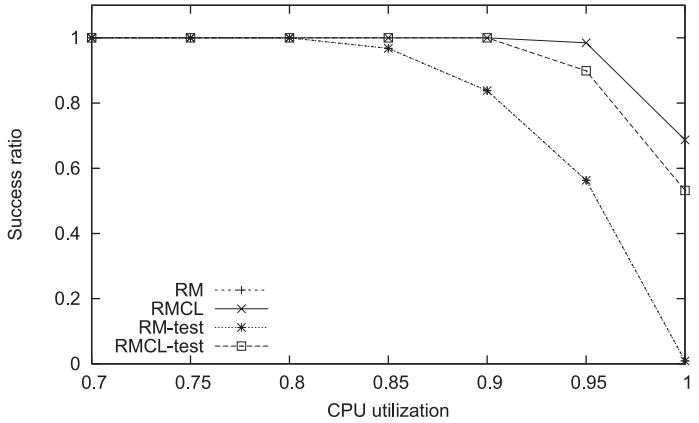

図 10　スケジュール成功率（$u_i = [0.1, 1.0]$）
Fig. 10　Success ratio ($u_i = [0.1, 1.0]$).

まず，文献 1) でも報告があるように RM の RTA は非常に厳密であるため RM と RM-test に差は見られなかった．一方，3.2 節で述べた RMCL の RTA では，2 つ以上のタスクが危険状態になる条件を満たした場合にスケジュール不可能と判定し，危険状態になるタイミングによってはスケジュール可能であるタスクセットもスケジュール不可能と判定してしまう場合があるため，実際のスケジューリングによる結果とスケジュール可能性判定による結果に差が見られた．この差を縮めることは今後の課題とする．

上述したように RMCL のスケジュール可能性判定は厳密ではないが，それでも RM に比べて 10%程度高い CPU 使用率でもスケジュール成功率を 100%に維持できた．実際にタスクセットを RMCL スケジューリングした場合は，さらに 2~5%程度スケジュール成功率が高く，CPU 使用率が 95%の段階でもスケジュール成功率は 98.5%であり，最適な EDF に匹敵する性能を発揮できたことが分かる．

図 11 は個々のタスクの CPU 使用率が $0.1 \sim 0.5$ の場合のスケジュール成功率を示している．図 10 の結果と同様に，RM では実際のスケジューリングと理論的なスケジュール可能性判定による差は見られなかったが，RMCL では図 10 の結果よりも大きな差が見られた．これは個々の CPU 使用率が $0.1 \sim 0.5$ となったことで，全体的にタスクセットのタ

⋆1 Success Ratio $= \frac{\#\text{ of successfully scheduled task sets}}{\#\text{ of scheduled task sets}}$

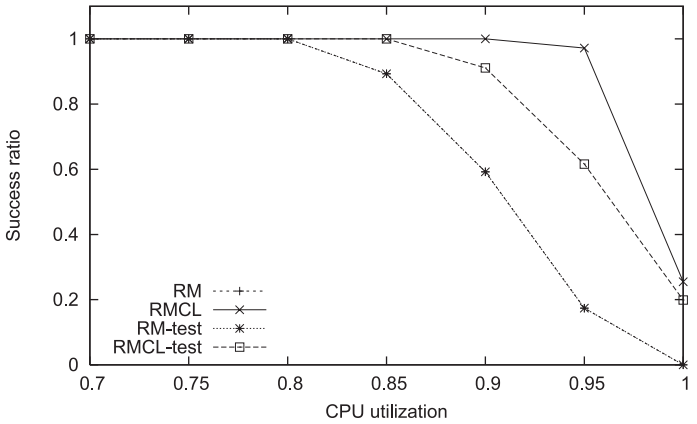

図 11　スケジュール成功率（$u_i = [0.1, 0.5]$）
Fig. 11　Success ratio ($u_i = [0.1, 0.5]$).

スク数が増え，実際にはスケジュール可能であるタスクをスケジュール不可能であると判定してしまう確率が高くなったためであると考えられる．それでも RM より 5% 程度高い CPU 使用率でスケジュール成功率を 100%に保つことができた．実際に RMCL によってスケジューリングした場合は，さらに 5% 高い CPU 使用率 90%でもスケジュール成功率を 100%に保つことができた．

これらの結果から，理論的に RM より RMCL のほうが高い CPU 使用率でリアルタイム性を保証可能であり，実際にスケジューリングした場合も RM より RMCL のほうが高い CPU 使用率でリアルタイム性を維持可能であることが分かった．

5.2　実機による評価

実機上での画像処理による実験で Resch モジュールを組み込んだ Linux とネイティブの Linux のデッドラインミス率を計測し，高負荷状態におけるリアルタイム性を評価した．Resch モジュールを組み込んだ Linux を T-ReX（The Real-time eXtension の略）と表記し，ネイティブの Linux を Native と表記する．スケジューリングアルゴリズムとして，T-ReX は RMCL を採用し，Native は RM を採用する．

5.2.1　実 験 環 境

評価には Intel Core2 Duo CPU E6750 2.66 GHz を搭載し，主記憶 2 GB の PC を使用した．本論文ではシングルコアを対象としたスケジューリングアルゴリズムを提案しているので，Linux は CPU の 1 コアのみを使う設定でコンパイルを行った．

表 1　デッドラインミス率（OpenCV，タスク数 4）
Table 1　Deadline miss ratio (OpenCV, four tasks).

タスク	$U = 0.85$		$U = 0.9$		$U = 0.95$		$U = 1.0$	
	Native	T-ReX	Native	T-ReX	Native	T-ReX	Native	T-ReX
T_3	0%	0%	0%	0%	0%	0%	7.31%	0%
T_4	0%	0%	9.5%	0%	18.9%	0%	99.9%	19.3%

まず，画像処理用ライブラリである OpenCV [5] を使用して特徴点を抽出して画像認識を行うプログラムを作成し，各タスクに入力する画像サイズや画像処理の周期を調節することで CPU 使用率が {0.85, 0.9, 0.95, 1.0} となる状態を作り出してデッドラインミス率を計測する実験を行った．5.1.2 項のシミュレーションによる評価結果より RM と RMCL でデッドラインミスが発生するのは CPU 使用率が 85%以上の場合なので，CPU 使用率が 80%以下の実験は省略した．

OpenCV による画像処理実験では同一タスクに対する入力画像のサイズが変わらないので，各周期での実行時間は比較的一定であった．しかしながら，マルチメディア処理ではしばしば各周期で実行時間が大きく変動することが知られている．そこで，FFmpeg [4] を使用して MPEG4 デコードプログラムを作成し，各タスクに入力する動画のフレームレートを調節することで CPU 使用率が平均的に {0.85, 0.9, 0.95, 1.0} となる状態を作り出してデッドラインミス率を計測する実験も行った．OpenCV を使用して MPEG4 ファイルを作成することでフレームレートを調節した．

固定優先度アルゴリズムの性能が個々のタスクの CPU 使用率やタスク数に依存することを考慮して，それぞれの実験においてタスク数が 4 つの場合と 8 つの場合で評価を行った．CPU 使用率はおおよその値で算出した．

5.2.2　OpenCV による実験結果

表 1 は OpenCV を使って 4 つの画像処理タスク $T_1 \sim T_4$ を実行した場合の Native と T-ReX におけるデッドラインミス率を示している．周期の長さは $T_1 < T_2 < T_3 < T_4$ とした．紙面の都合上，Native か T-ReX かのどちらかでデッドラインミスを発生したタスクの結果のみ示す．CPU 使用率が 85%の場合，Native と T-ReX の両方ともデッドラインミスなしに全タスクを実行することができた．しかしながら，CPU 使用率が 90%以上になると，RM によってスケジューリングを行う Native ではデッドラインミスが増加し，最終的に CPU 使用率が 100%になると最低優先度タスクである T_4 のデッドラインミス率は

99.9%に達し，ほとんど適切に処理が行い状況になってしまった．一方，RMCL によってスケジューリングを行う T-ReX では，CPU 使用率が 95%に達してもデッドラインミスは発生せず，CPU 使用率が 100%になっても最低優先度タスク T_4 のデッドラインミス率は 19.3%に抑えることができた．

表 2 は OpenCV を使って 8 つの画像処理タスク $T_1 \sim T_8$ を実行した場合の Native と T-ReX におけるデッドラインミス率を示している．この実験では CPU 使用率が 90%になっても Native と T-ReX の両方ともデッドラインミスを起こさなかった．タスク数が 8 つに増えたことで個々のタスクの CPU 使用率は比較的小さくなり，その分すべてのタスクがリアルタイム性を維持できる CPU 使用率が上がったのだと考えられる．しかしながら，CPU 使用率が 95%になると Native では 3 つの低優先度タスクがデッドラインミスを起こしてしまった．さらに CPU 使用率が 100%になると，デッドラインミス率は T_7 では 40%，T_8 では 99.9%も検出され，リアルタイム性が劇的に低下してしまった．先の実験に比べて個々の CPU 使用率は小さくてもタスク数が多いので，1 度リアルタイム性が保てなくなるとデッドラインミス率の増加量が大きくなったのだと考えられる．一方，T-ReX では CPU 使用率が 90%に達してもデッドラインミスは検出されず，CPU 使用率が 100%の状態でも T_6 と T_7 にはデッドラインミスは検出されず T_8 のデッドラインミス率も 28.5%に抑えることができた．

表 2　デッドラインミス率（OpenCV，タスク数 8）
Table 2　Deadline miss ratio (OpenCV, eight tasks).

タスク	$U = 0.85$		$U = 0.9$		$U = 0.95$		$U = 1.0$	
	Native	T-ReX	Native	T-ReX	Native	T-ReX	Native	T-ReX
T_6	0%	0%	0%	0%	2.79%	0%	9.21%	0%
T_7	0%	0%	0%	0%	7.80%	0%	40.1%	0%
T_8	0%	0%	0%	0%	10.4%	0%	99.9%	28.5%

これらの結果から，個々のタスクの CPU 使用率やタスク数に依存せず，Native よりも T-ReX のほうが高い CPU 使用率でリアルタイム性を維持できることが分かった．

5.2.3　FFmpeg による実験結果

OpenCV による画像処理とは異なり，MPEG4 デコードは各タスクの実行時間に変動が見られた．T-ReX では Critical Laxity の検出のために各タスクの残り実行時間を算出する必要があるが，最悪実行時間を基にして残り実行時間を算出すると実際の残り実行時間と誤差が生じてしまうので，残り時間の算出に最悪実行時間を使う T-Rex(W) と平均実行時間を使う T-Rex(A) の 2 通りの実装を用意した．

表 3 と**表 4** は FFmpeg を使って，それぞれ 4 つの MPEG4 デコードタスク $T_1 \sim T_4$ と 8 つの MPEG4 デコードタスク $T_1 \sim T_7$ を実行した場合の Native と T-ReX におけるデッドラインミス率を示している．デッドラインミスが発生する傾向は先の OpenCV による実験とほとんど同じであった．T-ReX(W) と T-ReX(A) を比べると最悪実行時間を基に残り

表 3　デッドラインミス率（FFmpeg，タスク数 4）
Table 3　Deadline miss ratio (FFmpeg, four tasks).

タスク	$U = 0.85$		$U = 0.9\%$		$U = 0.95$		$U = 1.0$	
	Native	T-ReX(W, A)	Native	T-ReX(W, A)	Native	T-ReX(W, A)	Native	T-ReX(W, A)
T_2	0%	0%, 0%	0%	0%, 0%	0%	0%, 0%	0%	0.01%, 0.02%
T_3	0%	0%, 0%	0.1%	0%, 0%	0%	0%, 0%	7.81%	0.08%, 1.42%
T_4	0.3%	0%, 0%	8.20%	0%, 0.01%	19.3%	0%, 0.09%	94.8%	1.97%, 3.69%

表 4　デッドラインミス率（FFmpeg，タスク数 8）
Table 4　Deadline miss ratio (FFmpeg, eight tasks).

タスク	$U = 0.85$		$U = 0.9$		$U = 0.95$		$U = 1.0$	
	Native	T-ReX(W, A)	Native	T-ReX(W, A)	Native	T-ReX(W, A)	Native	T-ReX(W, A)
T_5	0%	0%, 0%	0%	0%, 0%	0%	0%, 0%	0%	0%, 0.01%
T_6	0%	0%, 0%	0%	0%, 0%	0%	0%, 0%	2.23%	0.02%, 0.09%
T_7	0%	0%, 0%	0.1%	0%, 0%	2.22%	0%, 0%	39.1%	0.07%, 2.13%
T_8	0%	0%, 0%	0%	0%, 0%	2.93%	0%, 0.01%	91.3%	1.36%, 3.07%

実行時間を算出して Critical Laxity を扱うほうがデッドラインミス率が小さかった．このことから，最悪実行時間を基にして残り実行時間を算出することで，実際には危険状態になっていないタスクに最高優先度を与えてしまっても大きなリアルタイム性の低下を招くことはないことが分かった．一方，平均実行時間を基にして残り実行時間を算出すると，実際には Critical Laxity 状態になっているタスクを検出できず，結果としてリアルタイム性が低下してしまったのだと考えられる．

これらの結果から，Resch モジュールを組み込んだ T-ReX は実行時間の変動が大きいリアルタイム処理に対しても有効であり，Critical Laxity の検出には最悪実行時間を用いたほうが高いリアルタイム性を実現できることが分かった．

6. 結　　論

本論文では，Linux カーネルのリアルタイム性改善を目的とした Resch モジュールを開発した．具体的には，固定優先度アルゴリズムの利点を継承しながらも高負荷状態において十分なリアルタイム性を提供可能な RMCL アルゴリズムを提案し，スケジュール可能性解析を行った．また，新規のスケジューリングアルゴリズムを簡単に Linux カーネルに組み込むためのフレームワークの設計と実装を行った．

シミュレーションによる評価では，RMCL が従来の固定優先度アルゴリズムである RM よりも約 10∼15%高い CPU 使用率でリアルタイム性を保証可能であることを示した．また，OpenCV と FFmpeg を使った実機による評価では，ネイティブな Linux に比べて Resch モジュールを組み込んだ Linux が高負荷状態におけるデッドラインミス率を約 70∼90%削減できることを実証した．

今後は本論文で開発した Resch モジュールを土台として，マルチコアへの対応を考える．本論文で提案した RMCL はもともとマルチプロセッサ向けの EDZL を応用したアルゴリズムなので，マルチコアでも高い性能を発揮すると予想できる．また，RMCL のスケジュール可能性判定には改善の余地があるため，今後はさらに厳密なスケジューリング解析を考える．さらには，電圧周波数制御による省電力化や特定のタスクに対してリアルタイム性を確保できる資源予約等，次世代の組み込みシステムに必要とされる機能を提供できるモジュールを開発していく予定である．

謝辞　本研究の一部は文部科学省グローバル COE プログラム「環境共生・安全システムデザインの先導拠点」および日本学術振興会の支援によるものである．

参 考 文 献

1) Audsley, N., Burns, A., Richardson, M., Tindell, K. and Wellings, A.: Applying New Scheduling Theory to Static Priority Preemptive Scheduling, *Software Engineering Journal*, Vol.8, No.5, pp.285–292 (1993).
2) Buttazzo, G.: Rate Monotonic vs. EDF: Judgment Day, *Real-Time Systems*, Vol.29, pp.5–26 (2005).
3) Cho, S., Lee, S., Han, A. and Lin, K.: Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems, *IEICE Trans. Communications*, Vol.E85-B, No.12, pp.2859–2867 (2002).
4) FFmpeg Project. FFmpeg, http://ffmpeg.mplayerhq.hu/
5) Intel Corporation: OpenCV. http://www.intel.com/technology/computing/opencv/index.htm
6) Liu, C. and Layland, J.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, *J. ACM*, Vol.20, No.1, pp.46–61 (1973).
7) Oikawa, S. and Rajkumar, R.: Portable RT: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior, *Proc. 5th IEEE Real-Time Technology and Aplications Symposium* (1999).
8) Rajkumar, R., Lee, C., Lehoczky, J. and Siewiorek, D.: A QoS-based Resource Allocation Model, *Proc. 18th IEEE Real-Time Systems Symposium* (1997).
9) TimeSys Corporation: TimeSys Linux. http:// www.timesys.com/
10) Yodaiken, V.: The RTLinux Manifesto, *Proc. 5th Linux Expo* (1999).
11) 石綿陽一，松井俊浩，國吉　康：高度な実時間処理機能を持つ Linux の開発，第 16 回日本ロボット学会学術講演会予稿集，pp.355–356 (1998).

(平成 20 年 7 月 23 日受付)
(平成 20 年 11 月 12 日採録)

加藤　真平（学生会員）

1982 年生．2004 年慶應義塾大学理工学部情報工学科卒業．2006 年同大学大学院理工学研究科開放環境科学専攻修士課程修了．博士（工学）．同年より同大学訪問研究員としてリアルタイムシステム，オペレーティングシステム等の研究に従事．

山崎　信行（正会員）

1966 年生．1991 年慶應義塾大学理工学部物理学科卒業．1996 年同大学大学院理工学研究科計算機科学専攻博士課程修了．博士（工学）．同年電子技術総合研究所入所．1998 年 10 月慶應義塾大学理工学部情報工学科助手．同専任講師を経て 2004 年 4 月より同准教授．現在産業技術総合研究所特別研究員を兼務．並列分散処理，リアルタイムシステム，システム LSI，ロボティクス等の研究に従事．