システムソフトウェアと
オペレーティング・システム 90－6
（２００２．６．２７）

SMP 環境における Linux スケジューラの評価

谷口 宏樹　石川 裕　平木 敬

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

SMP 環境における Linux 上のスケジューリング向上を目的に標準 Linux スケジューラを改良したマルチキューを持つスケジューラや O(1) スケジューラが提案され実装されている。本論文では、これらスケジューラのスケジューリングオーバーヘッドを 8-way プロセッサ SMP コンピュータ上で評価した。その結果 O(1) スケジューラは他の 2 つに比べてスケジューリングのオーバーヘッドが約 100 倍小さく、スケーラビリティも良いという結果が得られた。しかし、CPU を消費するアプリケーションの実行時間は、いずれのスケジューラを使用しても大差が生じない。

Evaluation of Linux schedulers on SMP environments

Hiroki Taniguchi　Yutaka Ishikawa　Kei Hiraki

Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo

Linux Schedulers, such as a modified Linux schedule with multi-queue and the O(1) scheduler, have been proposed and implemented in order to improve the scheduling performance on SMP environments. In this paper, the standard Linux scheduler and those schedulers are evaluated using an 8-way SMP machine. The result shows that O(1) scheduler is approximately one hundred times faster than other schedulers and good scalability. However, a processor intensive application takes almost the same execution time using those schedulers.

1　はじめに

PC サーバにおける共有メモリ型並列コンピュータが Linux とともに普及し、WEB サーバやメールサーバ、ファイルサーバとして使われ出している。しかし、Linux は、元来単一プロセッサ上のカーネルとして開発されたために、マルチプロセッサにおけるスケーラビリティに問題がある。

スケーラビリティを阻害する一つの要因がスケジューラである。Linux のスケジューラでは、全ての実行可能なプロセスは単一のリストに格納される。スケジューリングの度に、スケジューラはリストのプロセスの優先順位を全て再計算し、次に実行すべきプロセスを選択する。このため、並列コンピュータ上では、各プロセッサがプロセススケジューリングの時にそのリストをアクセスすることによる競合が生じ、オーバヘッドが生じる [1]。

論文 [3] では、待ち行列をプロセッサ毎に持たせ

ることにより、プロセッサ間でのアクセス競合を減らす工夫をしている。しかし、各プロセッサでは、リスト中のプロセスの優先順位を全て再計算するために、スケジューリングコストは実行可能なプロセス数に比例する。本方式を本論文では、マルチキュー・スケジューラと呼ぶことにする。

O(1) スケジューラ [4] では、各プロセッサ毎に優先度付き待ち行列を持つことによりプロセッサ間でのアクセス競合を減らすとともに各プロセッサによる待ち行列内プロセスの優先順位の再計算を省略している。

従来の Linux 標準スケジューラ、マルチキュー・スケジューラ、O(1) スケジューラは共に、並列オペレーティングシステムのスケジューラとしての新規性があるとは言い難い。しかし、コモディティな PC サーバ上で稼働する並列 OS として、これらスケジューラのコストを計測し比較することは、現在のマルチプロセッサハードウエア環境における並列 OS 研究の参考になる。

そこで本論文では、8-way プロセッサ Pentium III Xeon 700MHz 共有メモリ型並列コンピュータにおいて、Linux 2.4 系の標準スケジューラ、マルチキュー・スケジューラ、O(1) スケジューラの 3 つの方式のコストを、Pentium プロセッサが持つタイムスタンプカウンタを使って計測する。

その結果、O(1) スケジューラではスケジューリングにかかる時間が他の 2 つのスケジューラに比べ非常に速くプロセッサ数に対するスケーラビリティもあるという結果が得られたが、CPU を消費するようなアプリケーションではそのスケジューラを使用しても大差が生じない。

以下、本論文では第 2 章で、Linux 2.4 系での標準のスケジューラ、マルチキュー・スケジューラ、O(1) スケジューラの 3 つの方式の実装方法について紹介した後、第 3 章でスケジューラコスト計測方法について述べる。

2 既存スケジューラの概要

2.1 標準スケジューラ (2.4 系)

2.1.1 タスクリスト

Linux 2.4 系までの標準のスケジューラでは、すべてのタスクがひとつのタスクリスト上に置かれている。スケジューラが呼ばれるたびに、タスクリストの中から次に実行すべき最適なタスクを探し出す。スケジューラが呼ばれるたびに、タスクリストの全てのエントリを検索するので、タスク数に比例して処理時間も比例する。

タスクリストはシングルプロセッサ環境においてもマルチプロセッサ環境においてもひとつである。マルチプロセッサ環境においては、ひとつのタスクリストを複数のプロセッサで共有しているため、ロック獲得のために競合が起きる。

2.1.2 スケジューリング変数

タスクが保持しているスケジューリングのための変数および値は以下の通りである。

- カウンタ
  実行可能ティック数 (タイマ割り込みの回数) の残数をカウンタとして持っている。そのカウンタが 0 になるとスケジュールされない。
- ナイス値
  タスクの優先度を示す値である。 −20 が最も高く、 19 が最も低い。
- スケジューリングポリシ
  通常方式、ラウンドロビン、 FIFO のいずれかである。
- 直前まで動作していたプロセッサ ID
  タスクのプロセッサ間移動を制御するために保持している。

これらの変数から、タスクリストから次に実行すべき最適なタスクを選出するための指標となる Goodness 値が計算される。 Goodness 値は次節で説明する関数`goodness()` の返り値として得ることができる。

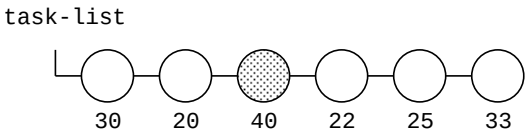

図 1: 標準スケジューラ

2.1.3 Goodness 値

関数 goodness() の返り値は、リアルタイムタスクと通常のタスクで異なる。リアルタイムタスクの場合、 ”1000 + リアルタイムタスクの優先度” で求められる。通常タスクの場合は、タスクのカウンタが 0 であれば無条件に 0 となり、そうでない場合は ”カウンタの値+20−ナイス値” となる。

さらに、現在のアドレス空間と同じアドレス空間である場合は +1 する。マルチプロセッサ環境では、現在のプロセッサと同じプロセッサ上で直前まで動いていたタスクには +15 する。

2.1.4 スケジューリングの概要

スケジューラはまず、スケジューラが呼び出された時に実行されていたタスクのアドレス空間が有効であるか、割り込みの中から呼び出されていないかをチェックする。タスクのスケジューリングポリシがラウンドロビンの場合は、カウンタをナイス値から再計算しタスクリストの最後につける。

次に実行すべき最適なタスクを選出するために、タスクリスト上のすべてのタスクに対して goodness 値を求め、その値が最大となるタスクを候補とする。図 1 においては goodness 値が 40 のタスクが選出される。もしこの最大値が 0 の場合は、タスクリスト上のすべてのタスクのカウンタが 0 であるのでカウンタを再設定して再スケジュールする。もし、次に実行すべきタスクとしてひとつ前と同じタスクが選ばれた場合は、スケジューラは何もしないでそのタスクへ処理を移す。

タスクリスト上のすべてのタスクの goodness 値の計算中はロックしているので、マルチプロセッサ環境では、ひとつのタスクリストに対して複数のプロセッサがロックを獲得しようとするので競合が起きる。

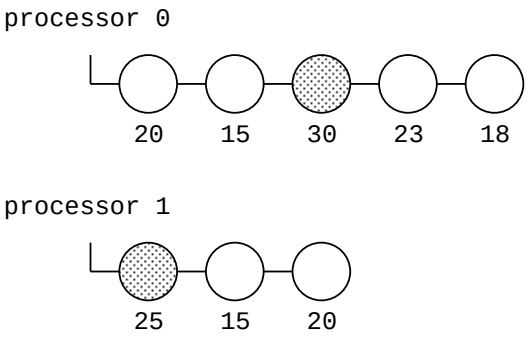

図 2: マルチキュー・スケジューラ

2.2 マルチキュー・スケジューラ [3]

2.2.1 プロセッサ毎のタスクリスト

マルチキュー・スケジューラでは、標準スケジューラではマルチプロセッサ環境でもひとつしかなかったタスクリストを各プロセッサごとに持つ。スケジューラは各プロセッサごとに自身のタスクリスト上に存在するタスクに対してのみ goodness 値を求める (図 2)。2.4 系の標準スケジューラであったような、他のプロセッサがスケジュール中で待たされるということが回避される。

2.2.2 Goodness 値

タスクの goodness 値の計算はスケジュールの際、何回も必要となるため、各プロセッサにおけるタスクリスト上のタスクの goodness 値の計算をできるだけ簡単にしたい。マルチキュー・スケジューラでは、毎回の goodness 値の計算において、自身が管理するタスクリスト中のタスクのみ goodness 値を計算する。

各プロセッサはタスクリスト中のタスクの goodness 値の大きいものから 2 つ目のタスクの goodness 値を別に計算する。最大のものは次に実行されるので 2 つ目を計算する。各プロセッサはこの

値を自身のタスクリスト上のタスクの goodness 値の最大値として保持している。

2.2.3 スケジューリングの概要

標準スケジューラと同じく、アドレス空間やスケジューリングポリシのチェックを行う。そのあと、現在のプロセッサのタスクリスト上のタスクの goodness 値を計算し、その値が最大なものを次にスケジューリングされる候補として選出する。図 2 においてはプロセッサ 0 では goodness 値が 30 のタスクが、プロセッサ 1 では goodness 値が 25 のタスクが選出される。他のプロセッサのタスクリスト上のタスクに関しては`examine_rmt_rq()` においてチェックする。

関数 `examine_rmt_rq()` では、現在のプロセッサのタスクリスト上のタスクより、他のプロセッサのタスクリスト上のタスクの goodness 値の最大値が大きければ、プロセッサのタスクリスト間でそのタスクを移動し、次に実行すべきタスクとして採用する。各プロセッサのタスクリスト上のタスクの goodness 値の最大値は、各プロセッサが自身のタスクリスト上のタスクの goodness 値を計算したときに保持していたものを使うため、ここでは計算はしていない。各プロセッサは自身のタスクリストに関しては、 goodness 値を計算しタスクを選出する際にロックをとる必要があるが、他のプロセッサのタスクリストに関しては、関数 `examine_rmt_rq()` で goodness 値を比較しプロセッサのタスクリスト間でタスクを移動する際にのみロックをとる必要がある。これによりプロセッサ間でロックの競合が起こりにくい。

2.3 O(1) スケジューラ [4] (2.5 系)

2.3.1 優先度付き待ち行列

Linux 2.5 系や Red Hat Linux 7.3 にも採用された O(1) スケジューラでは、各プロセッサごとにリアルタイムタスク用と通常のタスク用から成る優先度付き待ち行列を 2 組づつ持っている (図 3)。2 組の待ち行列はそれぞれ`active expired` と呼ばれる。 `active` には、これからスケジュールされるタスクが入っており、 `expired` には、既にスケジュールされたタスクが入っている。

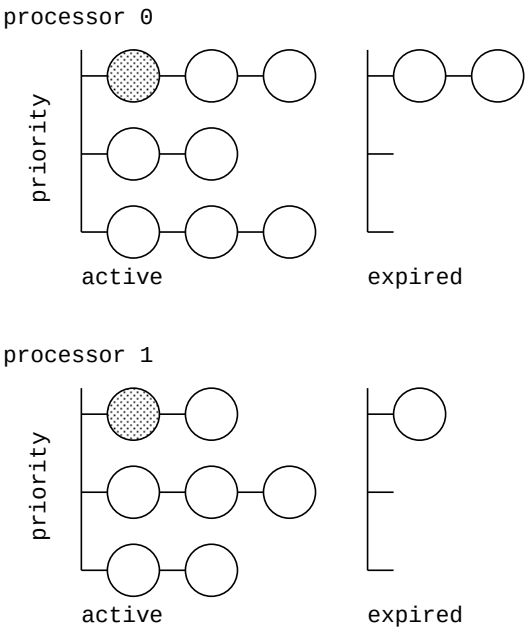

図 3: O(1) スケジューラ

スケジューラは `active` な待ち行列の最も優先度の高い先頭のタスクを次に実行すべきタスクとして選出する。タスクの存在する最も優先度の高い `active` な待ち行列のタスクがすべてスケジュールされてなくなった際には `active` と `expired` を入れ換え、スケジューリングを再開する。

各プロセッサの待ち行列に存在するタスクの総数は、変数`nr_running` に保存されている。リアルタイムタスクに関しては、通常のタスクより優先度の高い待ち行列にエンキューされる。

2.3.2 負荷分散

関数 `load_balance()` は、スケジューラ自体からとタイマから呼ばれる関数 `schedule_tick` から呼ばれる。スケジューラからの呼び出しは上で述べた通りである。タイマに関わる呼び出しは、タイマ割り込み 25 回に 1 度 `load_balance` が呼ばれる。

関数 `load_balance()` では、待ち行列に存在

するタスクの数が最も多いプロセッサを見つけ出し、その待ち行列を busiest とする。現在のプロセッサの待ち行列の nr_running が、その busiest な待ち行列の nr_running の約半分より小さいときに、現在のプロセッサに busiest な待ち行列の先頭のタスクが移動可能かどうかをチェックし、その先頭のタスクを busiest な待ち行列から現在のプロセッサの待ち行列に移動する。この際、移動されるタスクの優先度が移動元より下がるような場合は、あとで再スケジュールされるようにフラグを立てておく。

2.3.3 スケジューリングの概要

アドレス空間やスケジューリングポリシのチェックを行ったのち、現在のプロセッサの待ち行列の nr_running をチェックし、0 であれば関数load_balance() が呼ばれ、2 つプロセッサ間で負荷分散が行われる。その際、待ち行列の active と expired を入れ換える必要があるかどうかもチェックし、必要であれば入れ換える。標準スケジューラやマルチキュー・スケジューラで行っていた各タスクの goodness 値の計算は行っていない。その後、 active な待ち行列の最も優先度の高いタスクを次に実行すべきタスクとして選出し タスク切り換えを行う。

各プロセッサは自身の待ち行列からタスクを選出する際にロックをとる必要があるが、他のプロセッサの待ち行列に関してはロックをとる必要がない。負荷分散の際には、現在のプロセッサの待ち行列と busiest な待ち行列の 2 つ待ち行列に対してロックをとる必要があるが、すべてのプロセッサに対してロックするわけではないので効率的である。

3 測定方法

3.1 実験環境

測定は、富士通 IA サーバ PRIMERGY N800 (8-way プロセッサ Pentinum III Xeon 700MHz メモリ 4GB) 上で行った。

デーモン類などのプロセスはすべて起動しないようにし、ネットワークからも切り放した状態で測定した。 mingetty も必要最低限のひとつだけ起動する。起動しているプロセスは、カーネル関連のプロセスと init とログインシェルのみである。使用したカーネルのバージョンは 2.4.17 である。ただし、マルチキュー・スケジューラはパッチ [3] の都合で 2.4.14 を使用した。

3.2 測定プログラム

測定プログラムは、複数のスレッド (pthread) を生成しそれぞれのスレッドが独立なメモリ領域を read するプログラム。キャッシュの効果を測定したいので write はしない。

測定項目としては、複数のスレッドが生成されコンテキストスイッチが頻繁に起こるような状態で、関数 schedule() におけるスケジューリングにかかる時間とスケーラビリティと、測定プログラムの処理に要する総所要時間を測定する。

生成するスレッド数は、 2 から 1024 までの 2 のべき乗とし、それぞれのスレッドにおいて 4MB のメモリを read する操作を 50 回繰り返す。

3.3 測定項目

3.3.1 スケジューリングのコスト

それぞれのスケジューリングのコストを測るため、関数 schedule() での所用時間を Pentinum プロセッサのタイムスタンプカウンタで測定した。関数schedule() の先頭で、タイムスタンプカウンタの値を保存し、 return の前で集計していった。何回 関数schedule() が呼ばれたかも測定し、スケジューリング 1 回当りにかかるコストを算出した。

測定したデータは、カーネル内のメモリ空間あるため通常プロセスでは取得できない。取得するためには、システムコールを追加する方法とデータ取得のためのインターフェイスを設ける方法がある。今回は、/proc にエントリを追加しそれを経由して得る方法をとった。

3.3.2 総所要時間

測定プログラムの生成したすべてのスレッドでメモリの read が終了するまでの総所要時間を測定した。スレッド生成 `pthread_create()` から、スレッド終了 `pthread_join()` までの時間を総所要時間とした。

3.4 キャッシュの影響

タスクのプロセッサ間移動の際のキャッシュの効果を測定するため、`schedule()` 等において以下の 3 つの場合を考えた。

change 次に実行するタスクを他のプロセッサ上で動いていたものから選ぶ

default デフォルトのまま

stay タスクが常に同一プロセッサ上にスケジュールされるようにする

3.4.1 標準スケジューラ (2.4 系)

次に実行すべき最適なタスクの選出の際の goodness 値の計算において、タスクのプロセッサ間移動のペナルティのため、同じプロセッサのタスクには goodness 値に **default** では 15 足すが、**stay** では $15 + 100$ に, **change** では逆に異なるプロセッサに $15 - 100$ 足すことにする。

3.4.2 マルチキュー・スケジューラ

標準スケジューラのときと同様、 次に実行すべき最適なタスクの選出の際の goodness 値の計算において、タスクのプロセッサ間移動のペナルティのため、同じプロセッサのタスクには goodness 値に **default** では 15 足すが、**stay** では $15 + 100$ に, **change** では逆に異なるプロセッサに $15 - 100$ 足すことにする。

3.4.3 O(1) スケジューラ (2.5 系)

負荷分散する際において、待ち行列に存在するタスクの総数が最もな待ち行列からタスク を移動させることが必要だが、**stay** においてはそのタスクの移動を禁止する。この際、**stay** では移動させるタスクを選出する必要がなくなる. **default** の場合は `busiest` な待ち行列の`nr_running` の約半分より小さいときにしか負荷分散が行われるが、**change** では現在の待ち行列と `busiest` な待ち行列の`nr_running` の差が 3 以上のときに負荷分散するようにする。

4 結果

4.1 標準スケジューラとマルチキュー・スケジューラの比較

2、4、8 プロセッサ環境で **default** のままの標準スケジューラとマルチキュー・スケジューラについてスケジューリングにかかる時間の平均を測定した結果を図 4 に示す。

どちらのスケジューラでもスレッド数が多くなるとスケジューリングにかかる時間が少なくなるという結果となった。プロセッサ数が大きくなるとスケジューリングにかかる時間は多くなっている。プロセッサ数よりも少ないスレッド数の場合にばらつきがあるのは、アイドル状態のプロセッサがあるためである。

4.2 標準スケジューラと O(1) スケジューラの比較

2、4、8 プロセッサ環境で **default** のままの標準スケジューラと O(1) スケジューラについてスケジューリングにかかる時間の平均を測定した結果を図 5 に示す。

O(1) スケジューラが約 100 倍速い結果となった。どちらのスケジューラともプロセッサ数が大きくなるとスケジューリングにかかる時間は多くなっている。

4.3 総所要時間について

2、4、8 プロセッサ環境で測定プログラムの総所要時間を測定した。どのスケジューラに関しても総所要時間はそれほど違いはみられなかった。

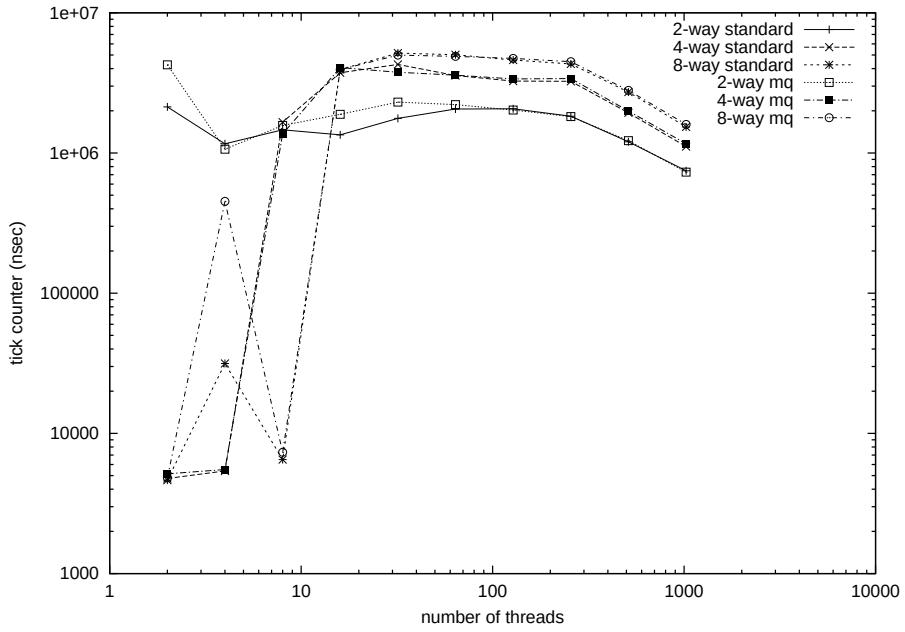

図 4: 標準スケジューラとマルチキュー・スケジューラのスケジューリングにかかる時間

表 1: スケジューリングにかかる時間の割合の比較

プロセッサ数	標準	マルチキュー	O(1)
2	0.75%	0.73%	0.013%
4	1.1%	1.1%	0.014%
8	1.5%	1.5%	0.016%

5 考察とまとめ

標準スケジューラとマルチキュー・スケジューラでは、スケジューリングにかかる時間ともにほとんど違いがみられないことがわかった。タスクリストをプロセッサごとに持ってもマルチプロッサにおけるタスクリストアクセス競合が性能に大きな差を生じていないことが分かる。

O(1) スケジューラでは他の 2 つのスケジューラに比べ、スケジューリングにかかる時間は約 100 倍速い。他のスケジューラでは、スケジューラが呼ばれる度に全てのタスクの goodness 値を計算しているが、 O(1) はそれを行なっていない。これによるコスト軽減が大きいと推察される。

O(1) スケジューラは他の 2 つのスケジューラに比べ、スケジューリングにかかる時間は約 100 倍速く、プロセッサ数に対するスケーラビリティも良い。しかし、今回測定に使用した CPU のみを消費するアプリケーションだけから構成されるプロセスのスケジューリングにおいては、実行時間に顕著な差が現われなかった。

CPU のみを消費するアプリケーションだけから構成されるプロセスのスケジューリングにおいては、100msec に一回スケジューリングされる。スレッド数 1024 の時、100msec に一回のスケジューリングオーバヘッドの割合を表 1 に示す。表 1 から分かる通り、100msec に一回スケジューリングされるような場合、標準スケジューラにおけるオーバヘッドは 1 % に過ぎないことが分かる。

謝辞

本論文の測定においては、 OSDL ジャパン (www.osdl.jp) の機材を借り測定しました。OSDL ジャパンのみなさまに深く感謝いたします。

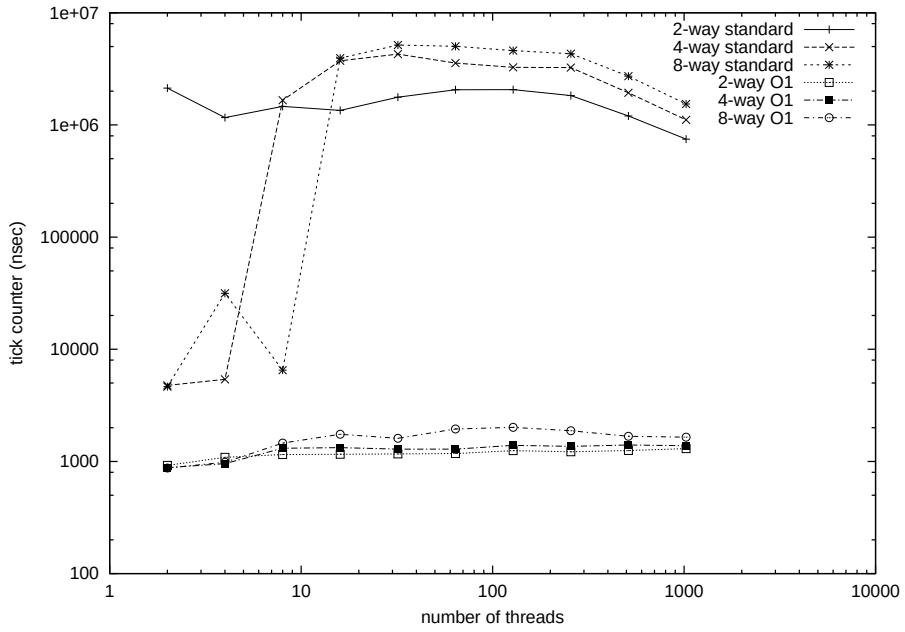

図 5: 標準スケジューラと O(1) スケジューラのスケジューリングにかかる時間

参考文献

[1] Ray Bryant, Bill Hartner, Qi He, Ganesh Venkitachalam: SMP Scalability Comparisons of Linux(R) Kernel 2.2.14 and 2.3.99, *the 4th Annual Linux(R) Showcase & Conference, 2000*

[2] Ray Bryant, John Hawkes, Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux(R) Kernel, *the 4th Annual Linux(R) Showcase & Conference, 2000*

[3] Mike Kravertz, Hubertus Franke; Implementation of a Multi-Queue Scheduler for Linux, *Linux Scalability Effort Homepage, 2001*; http://lse.sourceforge.net/scheduling/mq1.html

[4] Ingo Molnar: ultra-scalable O(1) SMP and UP scheduler, *Linux-Kernel Archive, 2002*; http://www.uwsg.indiana.edu/hypermail/linux/kernel/0201.0/0810.html; http://people.redhat.com/mingo/O(1)-scheduler/

[5] 富士通研究所: キャッシュミス削減によるプロセススケジューラの高速化; http://www.labs.fujitsu.com/techinfo/linux/

[6] Davide Libenzi: Linux Scheduler Stuff Page; http://www.xmailserver.org/linux-patches/lnxsched.html

[7] David A Rusling: The Linux Kernel, *Linux Documentation Project*; http://www.linuxdoc.org/LDP/tlk/

[8] IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide, *Intel Corporation*