A–03

Java スレッドへの Linux のスケジューリングを活用するための機能の提供
Providing a facility for utilizing Linux Scheduler to Java threads

篠本 一昌[††]　芝 公仁[†]
Kazumasa Shinomoto[††]　Masahito Shiba[†]

1　はじめに

プログラムの実行環境を独自に構築する仮想マシンでは，単純に単一のプログラムを動作させるだけでなく，プログラミング言語のレベルからスレッドをサポートする．スレッドに関しては，言語処理系の研究で様々な検討が行われており，ユーザレベルでも実現可能であるが，マルチプロセッサを活用できるなどの利点から，オペレーティングシステム (以下 OS) レベルでのスレッドを使用して仮想マシン上のスレッドを実現する場合が多くなっている．仮想マシン上にスレッドを実現する場合，仮想マシンには，オペレーティングシステムの持つスレッド管理機能をアプリケーションから有効に活用できるような仕組みを持つことが求められる．しかし，仮想マシンは OS の違いを吸収する役割も持っており，OS 間の差が大きいスレッド管理に関わる機能に関しては，仮想マシン上から利用できないものも多い．

最も広く利用されている仮想マシンのひとつに Java がある．Java 仮想マシンもスレッドをサポートしているが，3D 映像の描画処理などのような CPU 負荷の思い処理を行いながらユーザ操作の応答性を確保する場合などにはスレッドの動作を制御することが必要になる．スレッドの動作を制御するとき，組み込み用途など厳密に時間制約を満たす必要がある場合には，RTSJ などの拡張された Java を用いることが多い．しかし RTSJ はリアルタイムスレッドのクラスを新たに追加するものであり，標準のクラスライブラリを使用しているプログラムはその恩恵を受けることができない．これらのプログラムを利用する場合には，標準クラスライブラリのスレッドを制御できることが望ましい．

このような背景のもと，厳密なリアルタイム性は不要であるが，スレッドの動作を制御したい場合を想定し，OS 依存のスケジューリング機能の利用によって Java スレッドの動作をどの程度制御できるかについて調べた．本稿では，まず，最も利用されている Java 仮想マシンの実装の一つである HotSpot において，Java 仮想マシンが持つスレッドの優先度を管理する機能について評価し，Java 仮想マシンが Linux の持つスケジューリング機能を有効に活用していない事について指摘する．また，この問題を解決するために Java 仮想マシンに行った拡張について述べる．

[††] 龍谷大学大学院理工学研究科
Graduate School of Science and Technology, Ryukoku University
[†]龍谷大学理工学部
Faculty of Science and Technology, Ryukoku University

2　Linux 上で動作する Java 仮想マシンのスレッド管理

本章では，Linux がユーザに提供するスレッド管理機能と HotSpot での Java スレッドの管理について述べる．本研究では，オープンソースとして公開されている OpenJDK の HotSpot を使用して評価を行っている．

2.1　Linux が提供するスレッド管理機能

Linux では，各々のスレッドに静的優先度と nice 値の二つの優先度を設定することができる．静的優先度は，通常 0 に設定されており，sched_setscheduler システムコールを用いて 0 から 99 の範囲の値に変更することができる．スケジューリングにおいては，静的優先度の高いスレッドから優先して実行権を与えられる．また，スケジューリングはプリエンティブに行われ，より高い優先度を持つスレッドが実行可能になると，実行中のスレッドは実行権を取り上げられる．

nice 値は静的優先度が 0 のスレッドにのみ設定され，setpriority システムコールを用いて −20 から 19 の範囲の値に変更することができる．スケジューリングにおいては，タイムクォンタムの計算にのみこの値は使用され，次に実行権を与えるスレッドの選択には直接的には使用されない．

2.2　HotSpot でのスレッド制御方式

Java では java.lang.Thread クラスを用いてスレッドの制御を行う．Java スレッドの優先度は 1 から 10 の範囲の値をとり，通常 5 に設定される．Java の仕様では，優先度はスレッドの優先順位を設定するもの

であり，優先順位の高いスレッドは低いスレッドより優先して実行されると決められている．しかし，実際のスレッドの動作は，実行環境や Java 仮想マシンの実装のしかたによって違いがある．

HotSpot では，他の多くの Java 仮想マシン同様，OS が提供するネイティブスレッドを作成し，これに Java スレッドの処理を割り当てている．このとき，Java スレッドの優先度をネイティブスレッドにどう対応付けるかを UseThreadPriorites と ThreadPriorityPolicy の二つの項目によって起動時に設定することができる．

UseThreadPriorites は，Java スレッドに対応するネイティブスレッドの優先度の設定を行うかどうかを指定するものであり，標準で有効になっている．ThreadPriorityPolicy は，Java スレッドの優先度とネイティブスレッドの優先度の割り当て方法について指定するものであり，0 か 1 の値をとることができる．0 の場合，HotSpot は Java スレッドに対応するネイティブスレッドの優先度を変更しない．すなわち UseThreadPriorites を無効にした場合と同様の動作となる．この場合，Java スレッドに対応する全てのネイティブスレッドが同一の優先度で動作し，結果として，Java スレッドは優先度が無視された形で実行される．ThreadPriorityPolicy が 1 の場合，HotSpot は，setpriority システムコールを用いて，ネイティブスレッドの nice 値を対応する Java スレッドの優先度に合わせて設定する．

3　スレッド優先度の制御

前章で述べたように，HotSpot が優先度を設定するために用いるのは setpriority システムコールであり，対応するネイティブスレッドに設定される優先度は nice 値である．nice 値は絶対的なものではないため，HotSpot 上で，より高い優先度を持つスレッドが実行可能状態にあっても，低い優先度を持つスレッドに実行権が与えられる場合がある．

可視化ツールなどで，compute-bound な処理をしている場合でもユーザ操作に対する応答性を確保したいときなど，高い優先度の Java スレッドを必ず優先して動作させたい場合がある．これを行うために，Java スレッドの優先度に応じてネイティブスレッドの静的優先度を設定するよう HotSpot のスレッド管理部を拡張した．このとき，静的優先度の値には Java スレッドの優先度と同じ値を設定するようにし，スケジューリングポリシにはラウンドロビンを用いている．スケジューリングに関する変更は既存アプリケーションへの影響が大きいため，この拡張は既存の機能の置き換えではなく，ThreadPriorityPolicy の一つとして追加する形で行った．静的優先度を設定する機能は，ThreadPriorityPolicy を 2 にした場合に有効になる．これにより HotSpot は nice 値によるものと静的優先度によるものの二つからスケジューリングポリシを選択し，使用することが可能となる．

表 1　実験環境

プロセッサ	Intel Pentium 4（3.20GHz）
物理メモリ	1GB
カーネル	Linux 2.6.32
Java 仮想マシン	OpenJDK 1.7

4　性能評価

本章では，Java スレッドが実際にどのように動作するかを確認し，HotSpot に対して行った拡張が有効に機能するかについて評価を行う．以下の実験では，Hyper-Threading を有効にしたプロセッサを使用しているため一度に 2 つのスレッドが動作可能である．今回使用した環境を表 1 に示す．

4.1　Java スレッドのスケジューリング

実際に，Java スレッドがどのように動作しているのかを調べるために，以下の処理を行うスレッドを同時に複数動作させ評価を行った．

(1) 初期値が 0 であるカウンタを 1 ずつ増加させる．
(2) カウンタが 7000 万を越えると処理を終了する．

カウンタは，整数型の変数であり，それぞれの Java スレッドに固有のものである．本実験では，4 個の Java スレッド T1，T2，T3，T4 を動作させた．このとき，各スレッドの優先度はそれぞれ 1，3，7，9 である．これらは，最低優先度，通常より低い優先度，通常より高い優先度，最高に近い優先度に相当する．

ThreadPriorityPolicy を 1 とした場合の各 Java スレッドのカウンタの値の変化を図 1 に示す．グラフの横軸は経過時間，縦軸はカウンタの値を示している．

各スレッドのカウンタの値の増加に差が現れており，全ての場合において優先度の高いスレッドは優先度の低いスレッドよりカウンタの増加が速くなっている．このことから，優先度が高いと割り当てられる CPU 時間が長くなることが分かる．また，スレッド T4 の処理が終了する以前から，より優先度の低いスレッド T1，T2，T3 が動作していることから，優先度の高いスレッドを完全に優先して実行することはできていないことが分かる．

ThreadPriorityPolicy を 2 とした場合の各 Java スレッドのカウンタの値の変化を図 2 に示す．Thread-

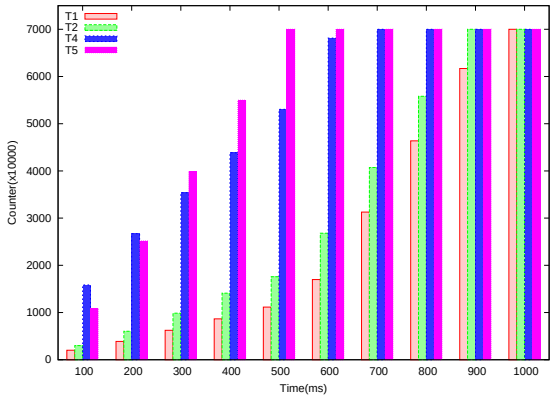

図 1　ThreadPriorityPolicy が 1 のときの Java スレッドのカウンタの値の変化

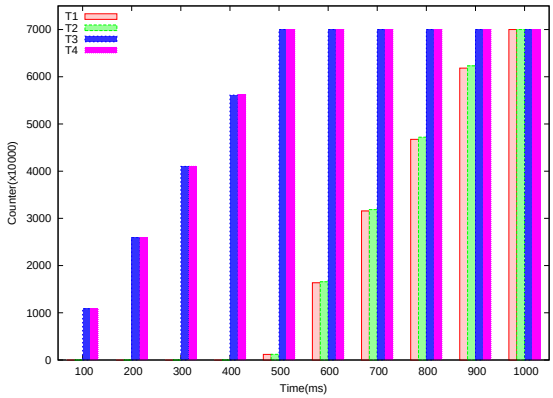

図 2　ThreadPriorityPolicy が 2 のときの Java スレッドのカウンタの値の変化

PriorityPolicy を 1 とした場合と異なり，優先度の高いスレッド T3，T4 が動作している間，優先度が低いスレッド T1，T2 のカウンタの値は変化せず 0 のままである．これは，各 Java スレッドに対応するネイティブスレッドの静的優先度を適切に設定することによって，OS よる静的優先度に基づくスケジューリングが行われるようになったためである．

4.2　Java スレッドのデッドラインミス率

次に，スレッドの優先度がスレッドの挙動に与える影響に関して，特に複数のスレッドが同時に動作する場合について評価を行う．複数のスレッドが動作する場合，資源の共有について考慮する必要がある．特に，Java では，すべての Java スレッドから共有されるヒープとそれを管理する Garbage Collection(以下 GC) が最も大きな共有資源であると言える．GC を共有資源ととらえ，HotSpot でのスレッドの動作が GC にどのような影響を受けるかについて評価を行う．

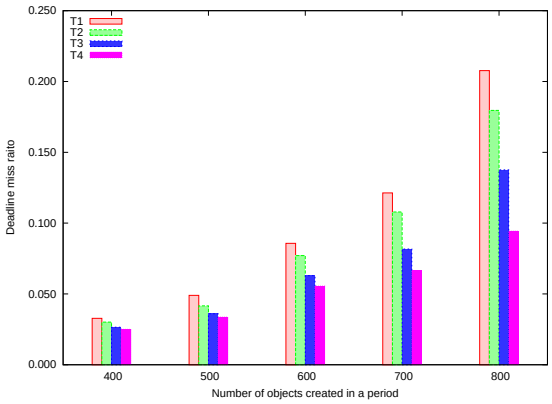

図 3　ThreadPriorityPolicy が 1 のときの Java スレッドのデッドラインミス率

Java スレッドがどの程度実行されるのかを確認するために，周期的に Java スレッドを動作させ，どの程度周期を守って処理を行えるかを調べた．本実験において，Java スレッドは，1ms 毎に起動し，一定の処理を行った後，次の周期の開始時刻までスリープする．このとき，周期内で処理を終えることができず，次の周期の開始時刻になった場合，当該周期にデッドラインミスが発生したと判断する．

4 個の Java スレッド，T1，T2，T3，T4 を 1ms の周期で動作させ，各周期で 400，500，600，700，800 のオブジェクトを生成させたときのデッドラインミス率を測定した．スレッド T1，T2，T3，T4 の優先度は，それぞれ，2，4，6，8 である．GC の効率はオブジェクトの使用のされ方やオブジェクト間のリンクのされ方によって変化するが，本実験では，単純な文字列オブジェクトの生成を行った．

ThreadPriorityPolicy を 1 とした場合の，各 Java スレッドのデッドラインミス率を図 3 に示す．グラフの横軸はオブジェクトの生成数，縦軸はデッドラインミス率を示している．すべての Java スレッドに関して，生成するオブジェクトの数が増えると，デッドラインミス率が高くなっている．このことから，優先度の高い Java スレッドの時間制約が優先的に満たされているわけではないことがわかる．

ThreadPriorityPohcy を 2 とした場合の，各 Java スレッドのデッドラインミス率を図 4 に示す．ThreadPriorityPolicy を 1 とした場合と異なり，生成するオブジェクトの数が増えても，優先度の低いスレッド T1，T2 のデッドラインミス率のみが高くなっており，優先度の高いスレッド T3，T4 のデッドラインミス率は大きく変化していない．また，全体的なデッドラインミス率も ThreadPriorityPohcy を 1 とした場合よ

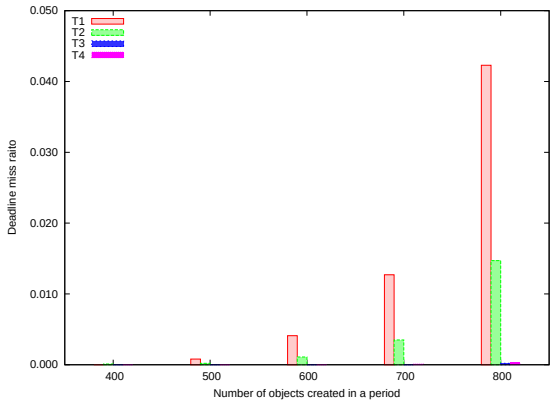

図 4　ThreadPriorityPolicy が 2 のときの Java スレッドのデッドラインミス率

り低くなっている．これは，優先度の高いスレッドが優先的に実行権限を与えられ，他のスレッドに動作を妨げられることなく処理を行えたためと考えられる．

5　関連研究

プログラムの実行において複数のコンテキスト持つことが出きるよう，単一のプロセスを複数のスレッドで実行できるようにしている OS が多い．また，多くの仮想マシンが，それぞれの仮想マシン上でスレッドを実現するために，OS が提供するスレッドの機能を利用している．スレッドに関する機能は，POSIX Thread 仕様のインタフェースで提供されることが多く，一部に違いがあるが，Linux でも Native POSIX Thread Library でこれを利用可能である [2]．このようにインタフェースが共通化されると，仮想マシンを様々なシステムに対応させることが容易になる．

しかし，このように共有化されたインタフェースでは，個々の OS に固有の機能を使用することが難しくなる．特にスケジューリングは，システムの中心となる部分であり，システム毎に大きく異なる．Linux においても，CFS（Completely Fair Schedular）など活発に開発が行われている [1]．また，Solaris も FSS（Fair Share Scheduler）や TS（Time Share Scheduler）等の優れたスレッド管理機能を持っており，Solaris 上で動作する HotSpot はこれを利用し，Java の仕様のような 10 段階の優先度ではないが，優先度の高いものが優先して実行されるようにしている．

このように，スレッド管理機能に関しては，共有化されたインタフェースでは利用が難しいものも多い．仮想マシンではオペレーティングシステムの違いを吸収することと，OS 固有の機能を活用することはトレードオフになるが，本研究では，OS 固有の機能を活用する手法をとり，Linux のスケジューリング固有の機能である静的な優先度を使用できるようにしている．

時間制約のある処理をサポートすることは，仮想マシンに対しても要求されるようになってきており，Java でもリアルタイム処理に関する拡張がいくつか存在する．J Consortium では，新たな拡張ライブラリを用いて，ハードリアルタイム処理のための機能を実現している [3]．また JTRON では，組み込みシステムへ Java を適応させるために，リアルタイム OS と Java 実行環境を融合させたハイブリットアーキテクチャを実現している [4]．RTSJ では，リアルタイム処理のために Java の機能の拡張を行っている．独自のスケジューラの実装や物理メモリへのアクセス、非同期イベントの制御などによって高度なリアルタイム処理を実現している [5]．これらは，クラスや API 群などを新たに追加するものであり，リアルタイム OS での使用を前提にしている．そのため，標準クラスライブラリのスレッドを使用している既存のアプリケーションにそのまま適応することは難しい．これらのアプリケーションを活用する場合には，標準クラスライブラリのスレッドを制御できることが望ましい．

6　おわりに

本論文では，Java スレッドへの Linux のスケジューリングを活用するための機能の提供について述べた．Linux 上の HotSpot では，Linux の持つスケジューリング機能を有効に活用していない事について指摘し，Linux 固有の機能である静的な優先度を使用できるよう HotSpot の拡張を行った．この拡張によって，Linux の機能である nice 値と静的優先度から一つを選択し，Java スレッドの優先度にある程度の信頼性を持たせることが可能となる．

本研究では，単一の Java 仮想マシン内のスレッドを対象とし，Java における優先度と Linux における優先度を単純に 1 対 1 に対応付けた．今後は，複数の Java 仮想マシンが動作する環境や，ネイティブアプリケーションも同時に動作する環境を対象とし，仮想マシンでの優先度をシステム全体の優先度に適応させる手法について検討する．

参考文献

[1] Wong, C., Tan, I., Kumari, R., Lam, J. and Fun, W.: Fairness and interactive performance of O

(1) and CFS Linux kernel schedulers, *Information Technology, 2008. ITSim 2008. International Symposium on*, Vol. 4, IEEE, pp. 1–8 (2008).

[2] Books, H.: *Posix Standards, Including: Posix, Single Unix Specification, the Open Group, Bourne Shell, Fork (Operating System), Native Posix Thread Library,*, Hephaestus Books (2011).

[3] J-Consortium. Real-time Core Extensions for the Java Platform. International J Consortium Specification, 2000.

[4] Nakamoto, Y. and Hachiya, S.: JTRON: a hybrid architecture integrating an object-oriented and real-time system, *Object-Oriented Real-Time Distributed Computing, 2001. ISORC-2001. Proceedings. Fourth IEEE International Symposium on*, IEEE, pp. 243–250 (2001).

[5] Greg Bollella et al. The Real-Time Specification for Java. Addison-Wesley, 2001.