システムソフトウェアと
オペレーティング・システム 75−12
（１９９７．６．５）

JavaのReal-Time拡張

三好昭彦[1] 喜多山卓郎[1] 徳田 英幸[1 2]

[1] 慶應義塾大学SFC研究所 [2] 慶應義塾大学環境情報学部

Javaには分散環境下でのセキュリティの機能や，アーキテクチャに依存しないため動作プラットフォームを選ばないなど様々な利点がある．これらの特徴から組み込みシステムや，機械制御など従来CやC++などといった言語が主に使われていた分野でもJavaが取り上げられるようになった．

しかし様々な解決しなければならない問題があることも確かである．例えばリアルタイムの制約などこれらの分野の多くのプログラムが正常に動作するためには必要である．現在のJavaの実行環境および言語スペックにおいてこれらの要求を完全に満たすことは困難である．

我々はJavaにおけるリアルタイムの問題を解決するためにリアルタイムのJava環境を構築した．本稿ではこのJava環境について説明し，その中のリアルタイムJavaスレッドの実装と評価を行なう．

キーワード　分散リアルタイムカーネル，Java，リアルタイムスレッド，マイクロカーネル

Real-Time Exention of Java

Akihiko Miyoshi[1] Takuro Kitayama[1] Hideyuki Tokuda[1 2]

[1] {miyos,takuro}@sfc.keio.ac.jp, Keio Research Institute at SFC, Keio University, 5322, Endo, Fujisawa-shi, Kanagawa, 252 Japan,

[2] hxt@sfc.keio.ac.jp, Faculty of Environmental Information, Keio University, 5322, Endo, Fujisawa-shi, Kanagawa, 252 Japan,

Java has many benefits such as security in distributed environments, reusability of code, and portability because it is architecture neutral. From those characteristics, Java is beginning to be used in many environments where more popular languages such as C and C++ were originally used.

Even though Java provides benefits, still it has problems which must be overcome. One issue is that, there are often real-time constraints that should be met in those applications. Current Java execution environment and language specification cannot satisfy those requirements. To investigate issues in real-time Java, we have implemented a prototype real-time Java environment and Real-Time Java threads. In this paper we will explain our implementation of real-time Java threads and evaluated its performance.

Keywords: Distributed Real-Time Kernel, Java, Real-Time Thread, Microkernel

1 はじめに

Java[4] は Sun Microsystems により開発された C++ に類似しているオブジェクト指向の言語である．分散環境下での利用を意識しアーキテクチャーに依存しないバイトコードを使用し，セキュリティの機能を持つなどが特徴としてあげられる．

最近では Java は今まで使われていなかったような新しい分野でも利用されようとしている．例えば組み込みシステム，情報家電などの分野などである．JavaOS や Network Computer などの商用の製品もこのトレンドの一例としてあげられるだろう．これは Java の持つ様々な特徴からくるもので，例えば C や C++などの言語でマシンを制御するプログラムを書くのであれば，そのターゲットに応じたライブラリなどにリンクし再コンパイルをしなければならなかった．それに対して Java はアーキテクチャに依存しないためプログラマはターゲットのアーキテクチャを事前に知る必要がないなどの利点がある．

言語レベルでのスレッドのサポートもこれらの分野では有用である．組み込みシステムや機械制御などでは，定期的もしくはランダムにおこる外部イベントを効率良く処理しなければならないためスレッドを使用しているが，C や C++ などでは言語レベルでのスレッドサポートがないため各プラットフォーム独自のスレッドインタフェイスに対応しなければならなかった．Java では統一したインタフェイスがあるためプログラムの移植性が高くなる．

Java にはこのような利点があるが，これらの分野で使用するには考慮されなければならない問題もある．機械の制御には指定された時間制限 (timing constraints) や必要な資源が必要な時に保証されなければならないという条件がある．現在の Java の実行環境および言語スペックではこのような条件を完全に満たすことが難しい．

一例をあげると Java のスレッドには `sleep(t)` というメソッドがある．このメソッドは走っているスレッドを `t` ミリ秒だけ静止させることができる．しかしこのメソッドだけでは多くのリアルタイムのプログラムには不十分である．`within(t) do s except q`[5]，すなわち `t` 時間内に `s` という仕事を終了させもしそれが満たされない場合は例外処理として `q` を行なうということが表現できるコンストラクトが必要な場合もある．他の問題点として，Java ではメモリや CPU の時間などの資源を表現したり予約ができないということがあげられる．リアルタイム処理を行なう場合より高い quality of service(QOS) を提供するために資源を予約したり保証したりすることが必要であるが，現在の環境ではそれができない．

現在我々はこのようなリアルタイム処理における問題点を解決すべくリアルタイム Java 環境を構築している．本稿では我々の Java 環境の設計について説明し，Java のスレッドの拡張であるリアルタイム Java スレッドの実装について述べる．その後，その性能やリアルタイムの機能について評価を行なう．

2 システム構造

我々のリアルタイム Java の環境は RT-Mach マイクロカーネル [12] 上のユーザレベルサーバとして実現されている．RT-Mach は米国カーネギーメロン大学にて開発され，Mach3.0 マイクロカーネルを拡張したものである．分散環境下でのリアルタイム処理をサポートし，Pentium, SPARC, MIPS, Power PC などの幅広いアーキテクチャに移植されている．RT-Mach にて提供されているリアルタイムの機能としてリアルタイムスレッド，リアルタイムの排他制御機能 [13]，リアルタイム IPC[2]，プロセッサ帯域予約機能 [17] などがある．

我々の Java サーバは RTS(Real-Time Server)[7] という同じく RT-Mach 上のユーザレベルサーバを基に実装された．RTS はオブジェクトベースのサーバでタスク管理，ファイル管理，ネームサービスや例外処理を行なう機能を持っている．Java サーバは RTS を拡張しネイティブコードだけではなく Java のバイトコードをも実行できるようにしたものである．またインメモリファイルシステムを持ち，フロッピー，ハードディスクや RAM ディスクなど様々なファイルシステムをマウントすることが可能である．ファイルはサーバのインメモリファイルシステムに連続したメモリ領域としてコピーすることも可能である．現在の実装では， Java サーバは初期化時にローカルにあるハードディスク上の Unix ファイルシステムをマウントし必要なクラスファイルなどをインメモリのファイルシステムにコピーを行なう．このことにより Java のプログラムを実行中にディスク I/O 処理でブロックしてしまうことを防いでいる．しかし，十分なメモリを持たないデバイスの場合はメモリ内に必要なファイルを保持するよりもマウントされたファイルシステムから直接読むことも可能である．Java のバイトコードを解釈し実行するインタプリタとして我々は kaffe[19] という仮想マシンを使用した．Kaffe はインタプリタと JIT(Just In Time compilation) 両方のモードをサポートしている．我々のサーバでは JIT モードで Java のバイトコードを実行する．

3 リアルタイム Java スレッド

RT-Mach にて提供されているリアルタイムスレッドを基にリアルタイム Java スレッドを実装した．RT-Mach には二つのリアルタイムスレッドの実装がある．一つは

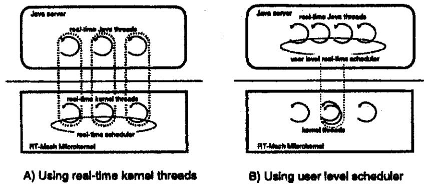

図 1: リアルタイムスレッドモデル

カーネル内で実装されている RT-Thread である．もう一つは C-Thread[1] を基に開発されたユーザレベルスレッドパッケージの RTC-Thread[8, 9] である．RT-Thread はカーネルのスケジューラでスケジュールされ短い間隔でインタラプトをかけるカーネル内のクロックデバイスを使用し時間制御を行なっている．RTC-Thread は時間制御とスレッドの管理機能をユーザレベルで行なうことにより柔軟性と効率性を実現している．両モデルにおいて，時間の属性を新たに指定することによりスレッドはリアルタイムスレッドになる．リアルタイムスレッドのモデルには周期スレッドと非周期スレッドがある．周期スレッドには時間属性として周期が設定され，その周期に基づき自動的にスレッドが起こされ指定された作業を繰り返し行なう．非周期スレッドは周期を持たず外部のイベントなどにより起こされる．

現在我々は性能比較のため二つのリアルタイム Java スレッドの実装を行なっている．一つ目はカーネル内で提供されている RT-Thread を使用するものである．(図 1-A 参照) ユーザがリアルタイム Java スレッドを新たに作成すると，バーチャルマシン内では RT-Thread を作成し Java スレッドをこれにマッピングする．つまり Java のスレッドと RT-Thread の間には一対一の関係がある．プライオリティインバージョン問題を避けるためにスレッド同士の同期を行なう場合，仮想マシン内で `rt_mutex_lock` と `rt_mutex_unlock` というプリミティブを使用している．

二つ目のモデルは図 1-B にあるように，ユーザレベルのスレッドとスケジューラを使用する．RTC-Thread パッケージを拡張しリアルタイムの排他制御を行なえるようにする．現在は一つ目のモデルを完成し二つ目のモデルの実装を行なっている最中である．

非リアルタイムスレッドとリアルタイムスレッドのプライオリティを分け図 2のようにした．非リアルタイムスレッド (Java において標準で提供されるスレッド) は 10 段階のプライオリティを持つ．(Java では高いプライオリティを表現するにはより高い数字を使用する．) 非リアルタイムスレッドより優先される位置にあるのがガーベージコレクションやファイナライザスレッドなどのシステムの状態を管理する活動である．リアルタイムスレッドは仮想マシン内での一番高いプライオリティを持ち同じく 10 段階に分けられる．すべての Java 内のスレッドは RT-Mach のスレッドに図 2のようにマッピングされている．

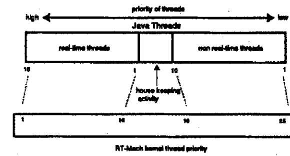

図 2: Java スレッドと RT-Mach スレッドのマッピング

4 プログラミング インタフェイス

リアルタイムスレッドを使用するためにいくつかの新しいクラスを提供した．この章ではこれらのクラスで提供されているメソッドの説明をする．`RtThread.class` は Java 言語の `Thread.class` を拡張したもので以下のようなメソッドがある．

- **`RtThread.setAttr(Time start, Time period, Time deadline)`**
  時間属性をリアルタイムスレッドに対し設定する．
- **`Time RtThread.getAttrStart()`**
  リアルタイムスレッドに設定されている開始時間の値を得る．
- **`Time RtThread.getAttrPeriod()`**
  リアルタイムスレッドに設定されている周期の値を得る．
- **`Time RtThread.getAttrDeadline()`**
  リアルタイムスレッドに設定されているデッドラインの値を得る．
- **`void RtThread.setHandler(RtHandler handler, String meth)`**
  このメソッドはリアルタイムスレッドに対してデッドラインハンドラを設定しデッドラインが守られなかった場合実行するメソッドを指定する．
- **`void RtThread.setStartAbsolute(int flag)`**
  フラグを 1 に設定すると，リアルタイムスレッドを絶対時間で開始させる．

`RtThread.class` をサポートするためにあるのが `RtHandler.class` と `Time.class` である．`Time.class`

は時間属性を秒とナノ秒で表現するクラスでリアルタイムスレッドの開始時刻，周期，デッドラインを指定するのに使用される．

- `Time(int seconds, int nanoseconds)`
  Time オブジェクトを作成するときに使用するコンストラクタで秒とナノ秒を指定する．
- `int Time.getSec()`
  設定されている秒数の値をかえす．
- `int Time.getNsec()`
  設定されているナノ秒数の値をかえす．
- `void Time.setSec(int sec)`
  秒数を設定する．
- `void Time.setNsec(int nsec)`
  ナノ秒数を設定する．
- `void Time.setCurrent()`
  現在の時刻を Time のクラスに設定する．

RtHandler.class はデッドラインハンドラを作成するために使用する．リアルタイムスレッドがデッドラインを守れなかった場合，これを検知し指定された対応を行なうのがデッドラインハンドラの役割である．一つのデッドラインハンドラで複数のリアルタイムスレッドのデッドラインミスを検知することが可能である．RtHandler.class は RtThread.class を拡張しているため自分自身もリアルタイムスレッドである．

普通の Java スレッドを作成する場合，Thread.class を継承したオブジェクトをつくり start() メソッドを呼びスレッドを動かす．同様にリアルタイムスレッドも RtThread.class を継承したオブジェクトをつくり start() メソッドを呼ぶことにより動かすことができる．以下に周期スレッドとデッドラインハンドラを使用した簡単なプログラム例を示す．

```
1. public class myProgram {
2.
3.   public static void main(String args[]) {
4.
5.       MyHandler handler = new MyHandler();
6.       handler.setPriority(10);
7.       handler.start();
8.
9.       Time period = new Time(1, 0);
10.      Time deadline = new Time(1, 0);
11.      Time start = new Time(0, 0);
12.      MyRtThread rtThread = new myRtThread();
13.      rtThread.setAttr(1, start, period, deadline);
14.      rtThread.setHandler(handler, "Handler");
15.      rtThread.start();
16.      try {
17.            Thread.sleep(60000);
18.      } catch (catch (Exception e) {
19.      }
20.      rtThread.stop();
21.      handler.stop();
22.   }
23. }
24.
25. class MyHandler extends RtHandler {
26.      public void       Handler(RtThread Th) {
27.            // describe recovery job here
28.      }
29. }
30.
31. class MyRtThread extends RtThread {
32.       public void       run() {
33.          // Thread body
34.
35.          // Do some Job
36.      }
37. }
```

5-7 行でデッドラインハンドラを作成し 10 という一番高いプライオリティを指定している．その後 start メソッドを呼び動作が開始される．9-11 行ではリアルタイムスレッドの時間属性を指定するために Time オブジェクトが作成されている．周期とデッドラインには 1 秒 0 ナノ秒が指定され，開始時刻には 0 が指定されている．開始時刻を 0 に設定しているということは start メソッドが呼ばれたらすぐに開始するという意味である．開始時刻を現在時刻+1 秒というような絶対時間で指定することも可能である．この場合 setStartAbsolute メソッドを使い以下のように明示的に指定を行なう．

```
rtThread.setStartAbsolute(1);
start.setCurrent();
start.setSec( start.getSec() + 1 );
```

13-15 行では新しいリアルタイムスレッドオブジェクトを作成し開始させる．25-29 行で記述されている MyHandler クラスの中でデッドラインを守れなかった場合の処理を記述する．例えば周期とデッドラインの値を伸ばすことによりシステムの負荷を軽減させることも可能であるし，アプリケーション特有の対処も可能である．動画再生プログラムであれば解像度やフレームレートを下げるなどがあげられる．31 行以降に記述されている MyRtThread が実際に作成したリアルタイムスレッドを記述するもので run メソッドの中で周期スレッドが実際に行なうべき作業を記す．

5 評価

すべての評価は 32MB のメモリを積んだ Intel Pentium 166MHz のマシンで計測した．正確に時間を計測するため

	Java server	kaffe (microseconds)
lock object	33.4	5.9
new Thread object	851.7	101.9
set priority	1.8	1.8
get priority	1.0	1.0
start Thread	299.7	140.3
context switch	20.5	5.5

表 1: Java スレッドの基本性能比較

に Pentium プロセッサにある RDTSC (read time stamp counter)[15] インストラクションを使用した.

最初に Java サーバと，このサーバで使用した基の仮想マシン kaffe の性能を表 1にて比較した. Java サーバは RT-Mach の上で他のサーバの無い環境で測定を行なった. また kaffe は Unix 上のアプリケーションであるため RT-Mach 上の Lite 4.4 BSD サーバにて測定を行なった. 表に記述されている値は各測定項目を 1000 回計り平均をとったものである.

オブジェクトを synchronized メソッドを使用しロックするのに Java サーバでは 30 マイクロ秒かかり kaffe では 6 マイクロ秒かかった. これは仮想マシン内の排他制御のメカニズムの違いによるものである. Java サーバではカーネルで提供されているロックを使用しリアルタイム排他制御をサポートしている. それに対して kaffe はユーザレベルにて排他制御を行なっている. この差はスレッドを新たに作る場合にも現れている. スレッドオブジェクトを新たに作る場合仮想マシン内でガーベージコレクション用のデータなど様々な共有情報をロックしなければならないからである. Java サーバでは約 850 マイクロ秒かかるのに対して kaffe では 10 マイクロ秒であった. プライオリティの値を得るコストおよび設定をするコストはオブジェクト内の値を操作するだけの作業なのでほぼ同じ性能であった. スレッドを走らせるために Java サーバでは 299.7 マイクロ秒かかり kaffe では 140.3 マイクロ秒かかっている. このコストの差は Java サーバではスレッドを走らせる時に RT-Mach のスレッドを作り Java のスレッドオブジェクトとマッピングを行なっているためである.

図 2にてリアルタイム Java スレッドの基本的な動作の評価を行なった. 非周期スレッドを作成するのに 440 マイクロ秒かかり周期スレッドを作成するのに 519 マイクロ秒かかる. この差は時間の属性の設定と周期タイマを作成するという作業が周期スレッドの場合必要なためである.

また，周期スレッドにデッドラインハンドラを設定し動作させるのに 586 マイクロ秒かかっている. これはデッドラインを知らせるメカニズムを設定するためのオーバ

	microseconds
new RtThread object	877.1
set priority	2.2
get priority	1.4
set attribute	2.2
get attribute	1.4
start aperiodic thread	440.3
start periodic thread	519.1
start periodic thread with deadline	586.2
delay from absolute start time	81.8

表 2: リアルタイム Java スレッドの性能

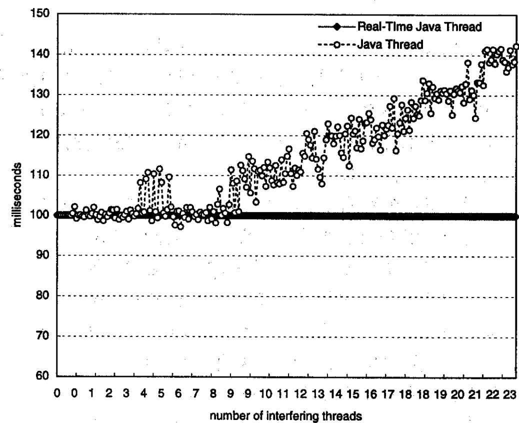

図 3: 外乱を加えての比較

ヘッドが含まれているからである. 開始時刻に絶対時間を設定した場合，実際に最初の Java バイトコードが実行されるまでに指定された絶対時間から 81.8 マイクロ秒のオフセットがあることがわかった.

図 3にて時間制約を加えた Java スレッドとリアルタイム Java スレッドの性能を外乱を加えた状態で比較した. 外乱はすべて測定対象のスレッドと同等のプライオリティを持つスレッドで I/O オペレーションを必要としないランダムな量の仕事を常に行なっている. Java スレッドでは周期的な活動を行なうために `sleep` メソッドを使用した. `while` ループの中で 100 ミリ秒 `sleep` させ `while` ループの先頭に戻ってくる時間の間隔を 1 周期とし，その間隔を測定した. リアルタイム Java スレッドは周期スレッドを周期 100 ミリ秒に設定し作成，その実際の周期を測定した. いずれの場合も周期活動を行なうスレッドは負荷のかかる作業をせず，順次外乱スレッドを増やしていった. 図 3は外乱スレッドの数を横軸に，周期の間隔を縦軸にとっている.

図から外乱スレッドが増えるにしたがい Java スレッド

は指定された周期を守れなくなっていることが明らかである．スケジューリングの段階で競合するスレッドが増えるにしたがい遅延おこり外乱スレッドが20個ある場合，遅延は20ミリ秒以上になっている．それに対して我々のリアルタイムJavaスレッドは外乱スレッドが20以上に増えても影響を受けずに周期を正確に守っている．この実験により負荷のかかった状態においてリアルタイムJavaスレッドの予測性が普通のJavaスレッドよりも高いことがわかった．

6 まとめ

本稿では組み込みシステム，機械制御などの従来CやC++などの言語を主に使っていた分野でJavaを使用する利点を述べた．しかしこのような分野で利用するためには時間制約などリアルタイムの機能を言語仕様および実行環境で持たなければならない．

我々はこの問題を解決するため，リアルタイムのJava環境をRT-Machマイクロカーネル上のJavaサーバとして実装している．Javaサーバの実装の説明をした後，Javaスレッドの拡張であるリアルタイムJavaスレッドについて述べた．リアルタイムJavaスレッドはJavaスレッドに新たに開始時刻，周期，デッドラインなどの時間属性を与えることにより作られる．

我々のJavaサーバと基になった仮想マシンkaffeの性能比較を行ない，Javaサーバでは排他制御を行なうために現在はカーネル内の機能を使用しているためオブジェクトのロックや作成時のコストに差があることがわかった．しかし，システムに負荷をかけた状態で時間制約を与えたJavaスレッドとリアルタイムJavaスレッドを比較した場合はリアルタイムJavaスレッドは正確に時間制約を守れている．Javaスレッドは負荷が高くなればなるほど時間の制約が守られなくなっていった．現在スケジューリングや排他制御をユーザレベルで行なうリアルタイムJavaスレッドを実装中である．

謝辞

本研究を進めるにあたり，MKngプロジェクトの皆様から多大な助言をいただきました．ここに感謝の意を表します．

参考文献

[1] E. C. Cooper, and R. P. Draves, C threads, *Technical report, Computer Science Department, Carnegie Mellon University*, CMU-CS-88-154, March, 1987.

[2] T. Kitayama, T. Nakajima, and H. Tokuda, RT-IPC: An IPC extension for Real-Time Mach, In *Proceedings of the USENIX Symposium on Microkernel and Other Kernel Architectures*, September 1993

[3] N. Gehani and K. Ramamritham, Real-Time Concurrent C: A language for Programming Dynamic Real-Time Systems, *J. Real-Time Systems*, Vol. 3, No. 4, 1991

[4] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison Wesley, 1996

[5] Y. Ishikawa,H. Tokuda, and C. Mercer, An Object-Oriented Real-Time Programming Language, *IEEE Computer*, Vol.25, No.10, 1992

[6] Kevin B. Kenny and Kwei-Jay Lin Building Flexible Real-Time Systems Using the Flex Language *IEEE Computer*, Vol. 24, No. 5, 1991

[7] T. Nakajima, T. Kitayama and H. Tokuda, Experiments with Real-Time Servers in Real-Time Mach, In *Proceedings of USENIX 3rd Mach Symposium*, 1993

[8] S. Oikawa and H. Tokuda, User-Level Real-Time Threads, In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, May 1994

[9] S. Oikawa and H. Tokuda, Efficient Timing Management for User-Level Real-Time Threads, In *Proceedings of the 1995 IEEE Real-Time Technology and Applications Symposium*, May 1995

[10] John Sasinowski and Jay Strosnider, ARTIFACT: A Platform for Evaluating Real-Time Window System Designs, In *Proceedings of the 1995 Real-Time Systems Symposium*, December 1995

[11] L.Sha, R.Rajkumar, and J. P.Lehoczky, Priority inheritance protocols: An approach to real-time synchronization, *IEEE Transactions on Computers*, Vol39, No.9, September 1990

[12] H. Tokuda, T. Nakajima, and P. Rao, Real-Time Mach: Towards a Predictable Real-Time System, In *Proceedings of USENIX Mach Workshop*, October 1990

[13] H. Tokuda and T. Nakajima, Evaluation of Real-Time Synchronization in Real-Time Mach, In *Proceedings of USENIX 2nd Mach Symposium*, October 1991

[14] H. Tokuda and T. Kitayama, Dynamic QOS Control based on Real-Time Threads, In *Proceedings of the 4th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 1993

[15] Terje Mathisen, Pentium Secrets, Byte magazine *http://www.byte.com/art/9407/sec12/art3.htm*

[16] C. W. Mercer, Y. Ishikawa, and H. Tokuda, Distributed Hartstone: A Distributed Real-time Benchmark Suite, In *Proceedings of the 10th Internatinal Conference on Distributed Computing Systems*, May 1990

[17] C. W. Mercer and S. Savage and H. Tokuda, Processor Capacity Reserves for Multimedia Operating Systems In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994

[18] Kelvin Nilsen, Java for Real-Time, *Real-Time Systems Journal*, Vol. 11, No. 2, 1996

[19] T. J. Wilkinson & Associates, Kaffe A free virtual machine to run Java code, *http://www.kaffe.org*, 1997