システムソフトウェアと オペレーティング・システム 60−10
マルチメディア通信と分散処理 61−10
（1993. 7. 8）

Real-Time Mach 3.0のマルチメディア処理に関する性能評価†

緒方 正暢[1] 和田 英彦[2] 追川 修一[3] 西尾 信彦[4] 徳田 英幸[5,6]

[1] 日本アイ・ビー・エム (株) 東京基礎研究所, [2] 横河電機 (株) オープンシステム研究所,
[3] 慶應義塾大学理工学部, [4] 慶應義塾大学環境情報研究所,
[5] 慶應義塾大学環境情報学部, [6] カーネギーメロン大学計算機科学科

あらまし 分散環境下でビデオや音声などの連続メディアをその時間的制約を保証するように扱うためには, 分散システム全体を通して時間的制約が解析, 予測可能な分散リアルタイムシステムが必要である. 我々は, Real-Time Mach 3.0 マイクロカーネルをもとに分散リアルタイムカーネルを開発している. 本稿では, 分散マルチメディアシステムを構築するために要求されるリアルタイムスケジューリング機能, リアルタイム通信機能, 動的なリアルタイムサービスの質 (Quality of Service, QOS) の制御といった機能について RT-Mach を評価, 検討した.

キーワード リアルタイム, マイクロカーネル, オペレーティングシステム, マルチメディア, 連続メディア, 分散システム

Performance Evaluation of Real-Time Primitives for Continuous Media Processing in Real-Time Mach 3.0

Masanobu Ogata[1] Hidehiko Wada[2]
Shuichi Oikawa[3] Nobuhiko Nishio[4] Hideyuki Tokuda[5,6]

[1] ogata@trl.ibm.co.jp, IBM Research, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamato, Kanagawa, 242 Japan,
[2] wada@crd.yokogawa.co.jp, Yokogawa Electric Co., 2-9-32, Nakacho, Musashino-shi, Tokyo, 180 Japan,
[3] shui@slab.sfc.keio.ac.jp, Keio University, 5322, Endo, Fujisawa-shi, Kanagawa, 252 Japan,
[4] vino@sfc.keio.ac.jp, Keio University, 5322, Endo, Fujisawa-shi, Kanagawa, 252 Japan,
[5] hxt@sfc.keio.ac.jp, Keio University, 5322, Endo, Fujisawa-shi, Kanagawa, 252 Japan,
[6] Carnegie Mellon University, Pittsburgh, PA 15213 USA

Abstract We have been developing a distributed real-time kernel based on Real-Time Mach 3.0 micro-kernel which supports continuous media such as video and voice in distributed environment. In this paper, we evaluate real-time primitives for continuous media processing such as real-time scheduling, real-time communication, and dynamic Quality of Service (QOS) control in RT-Mach.

Keyword Real-Time Processing, Micro-kernel, Operating System, Multimedia, Continuous Media, Distributed System

†この研究は, 情報処理振興事業協会 (IPA) が実施している開放型基盤ソフトウェア研究開発評価事業「マルチメディア統合環境基盤ソフトウェア」プロジェクトのもとに行なわれました.

1 はじめに

分散環境下でビデオや音声などの連続メディアをその時間的制約を保証するように扱うためには, 分散システム全体を通して時間的制約が解析, 予測可能な分散リアルタイムシステムが必要である [8]. 分散リアルタイムシステムでは, リアルタイムスケジューリング問題, リアルタイム通信問題, リアルタイムサービスの質 (Quality of Service, QOS)[9] といった問題を解決しなければならない.

分散リアルタイムアプリケーションの時間的制約を保証するには, クライアントプログラム側でのリアルタイムプロセススケジューリング, ネットワークでのデータ転送に関するリアルタイムスケジューリング, ならびに, サーバプログラム側でのリアルタイムプロセススケジューリング, といったように分散システム全体を通してリアルタイム処理がなされる必要がある.

リアルタイム通信を実現するためには, データ転送の高速化, スループットの向上だけでは十分ではない. スループットの保証, データ転送遅延時間 (latency) の保証, データユニットの遅延分散 (jitter) の保証ができなければならない. そのためには, プロトコル処理に必要な CPU, メモリ資源の管理が行え, ネットワーク資源の帯域の確保ができるリアルタイム通信プロトコルが必要である.

分散マルチメディアシステムでは, CPU, ネットワークのように複数のアプリケーションで共有される資源があるため, 全てのアプリケーションが常に期待するサービスを得ることはできない. 優先度に基づくリアルタイム処理が必要であるが, デッドラインに基づくプロセス制御を行なう通常のリアルタイム OS では, システムが過負荷な状況ではどのプロセスのデッドラインを優先するか制御できない. システムの状態に応じて, アプリケーション間で資源の割当を動的に変化させ, 過負荷な状況にも対応できるシステムが求められている. ユーザによってあらかじめ定義されているビデオ処理や音声処理に対する QOS をもとに, システムが, 実行時に資源の割当やスケジューリングを動的に制御できる必要がある.

我々は, 「マルチメディア統合環境基盤ソフトウェア」プロジェクト [10][11][12] のもとで分散リアルタイムカーネルを研究開発している. カーネルには, カーネギーメロン大学と慶應義塾大学で共同研究開発しているリアルタイムマイクロカーネル **Real-Time Mach 3.0**[6] を採用した. 本稿では, 分散マルチメディアシステムを構築するために要求される諸機能について RT-Mach を評価, 検討し, 改良点について述べる.

2 Real-Time Mach 3.0

Real-Time Mach 3.0 (RT-Mach) は, Mach 3.0 マイクロカーネル [2] が提供している機能に加えて, リアルタイムスレッド, リアルタイムスケジューラ, リアルタイム同期機構, リアルタイムプロセス間通信といったリアルタイム処理機能を備えている.

2.1 リアルタイムスレッド

時間的制約をもつ連続メディアを制御するプログラムでは, 時間的制約が明示的に記述できることが望ましい. RT-Mach は, C-Thread[3] を基本とした**リアルタイムスレッド**を提供している. ユーザは, 各リアルタイムスレッドのプライオリティ, 周期, デッドライン, スタート時間, デッドラインハンドラなどを記述できる. リアルタイムスレッドの時間的制約は, 属性 rt_thread_attr で記述される. リアルタイムスレッドは**クロックオブジェクト**と**タイマオブジェクト** [14] を使って実現されている. タイマとは, ある指示された時刻にアクションを起こすカーネルオブジェクトである. クロックとは, システムが備えるハードウェアクロックに関連づけられたオブジェクトである.

Mach では, thread_create() でスレッドを作成した後, thread_resume() で明示的にスレッドの実行を指示している. 一方, RT-Mach では, ユーザが rt_thread_create() を実行すれば, 設定した時間的制約に基づいてカーネルが自動的にリアルタイムスレッドを起動してくれる. ライブラリの rt_thread_attribute_init() と rt_thread_deadline_handler() を使えば, スタック領域を確保し, デフォルトの設定にすることができる. リアルタイムスレッドに関するプリミティブを以下に示す.

カーネルプリミティブ
rt_thread_create(parent_task, child_task, thread_attr)
rt_thread_exit(thread)
thread_get_attribute(thread, flavor, old_attr, old_attrCnt)
thread_set_attribute(thread, flavor, new_attr, new_attrCnt)

ライブラリ
rt_thread_attribute_init(prioriy, period_secs, period_nsecs, entry, arg, port, thread_attr)
rt_thread_deadline_handler(thread, thread_attr, entry, arg)

2.1.1 デッドラインハンドラ

ハードリアルタイムシステムと異なり, マルチメディアシステムでは全てのリアルタイムスレッドが必ずしもデッドラインを満たす必要はない. 例えば, システムの負荷により, ビデオ再生のリアルタイムスレッドがデッドラインをミスして, フレームの描画ができなくなったとして

も致命的なエラーとはならない. 重要なのは, デッドラインをミスした時の処理をユーザが記述できることである. RT-Machでは, ユーザはデッドラインハンドラを記述し, メッセージ待ち状態のスレッドとしてあらかじめ起動しておくことで, デッドラインをミスした時に備えることができる. リアルタイムスレッドがデッドラインまでに処理を完了できなかった時, カーネルはその処理を一旦中断させ, デッドラインポートを経由してデッドラインハンドラにメッセージを送る. このメッセージにより起動されたデッドラインハンドラは, デッドラインをミスしたリアルタイムスレッドを終了させたり, 回復処理を行なうことができる.

図1に, デッドラインハンドラの処理の様子を示す. ユーザは, 時刻 t1 でデッドラインハンドラを起動し, 時刻 t2 でデッドラインハンドラがメッセージ待ちで休止した後, リアルタイムスレッドを起動する. 時刻 t3 でデッドラインをミスした時は, カーネルはリアルタイムスレッドの処理を中断し, メッセージを送ってデッドラインハンドラを起動させる. この例では, 回復処理を行なった後, 時刻 t4 でリアルタイムスレッドの処理を再開させている. この方法は, リアルタイムスレッド毎に対応するデッドラインハンドラ用のスレッドを必要とするが, デッドラインハンドラ用のスレッドは優先度を低く設定できるので, 他の重要な緊急度の高いリアルタイムスレッドの実行を妨げないという利点を持っている.

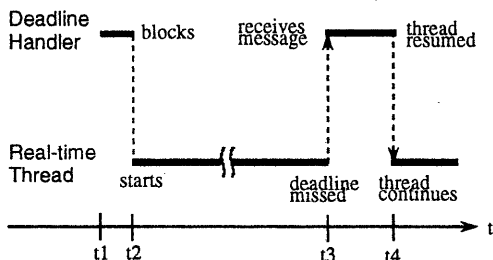

図 1: デッドラインハンドラ

2.1.2 プログラムの例

図2に周期的なリアルタイムスレッドとデッドラインハンドラを使ったプログラムの例を示す. まず, リアルタイムスレッドの時間的制約を rt_thread_attribute_init() を使って指定する (17 行目). この例では, リアルタイムスレッドの周期は 30msec に設定されている. 同様に, デッドラインハンドラの時間的制約を設定する (28 行目). rt_thread_create() を使って, デッドラインハンドラのスレッドと周期スレッドをそれぞれ起動する (37 行目と 44 行目). デッドラインハンドラのスレッドは, メッセージを待って直ちに休止する. リアルタイムスレッドは指定された時間的属性に基づきカーネルによって自動的に起動される. リアルタイムスレッドがデッドラインをミスした時は, メッセージによりデッドラインハンドラが起動される. デッドラインハンドラはメッセージを出力した (67 行目) 後, thread_resume() を使ってリアルタイムスレッドの実行を再開させる (70 行目).

2.2 リアルタイムスケジューラ

RT-Mach は, **Rate Monotonic** スケジューリング [1] に基づく **Integrated Time-Driven Scheduler** を備えている. 現在, RT-Mach カーネルは次に示す 6 種類のスケジューリングポリシを提供している. ユーザは, プロセッサセット単位でスケジューリングポリシを選択できる. processor_set_get_attribute() を使って現在設定されているスケジューリングポリシを得ることができ, processor_set_set_attribute() を使って新しいスケジューリングポリシを設定することができる. プリミティブを以下に示す.

---

processor_set_get_attribute(pset, flavor, old_attr, old_attrCnt)
processor_set_set_attribute(pset, flavor, new_attr, new_attrCnt)

Mach Timesharing: Mach の時分割スケジューリング. 全てのスレッドに平等に CPU 資源を割り当てる. スレッドの優先度は, スケジューラにより動的に変えられる.

Fixed Priority/RR: スレッドの優先度を固定する. 優先度の高いスレッドを先に実行する. 同じ優先度のスレッドはラウンドロビン方式で CPU 資源を割り当てる.

Fixed Priority/FIFO: スレッドの優先度を固定する. 優先度の高いスレッドを先に実行する. 同じ優先度のスレッドは FIFO 方式スケジューリングで CPU 資源を割り当てる.

Rate Monotonic: 周期の短い順にスレッドに優先度を与える先取り可能なスケジューリング.

Deadline Monotonic: デッドラインの短い順にスレッドに優先度を与える先取り可能なスケジューリング.

Earliest Deadline First: デッドラインに近い順にスレッドに優先度を与える先取り可能なスケジューリング.

2.3 リアルタイム同期機構

RT-Mach でも Mach と同様に, 同一タスク内のすべてのスレッドはタスクの資源を共有するため, スレッド間での同期機構が必要である. Mach では同期待ちにあるタスクが飢餓状態にならないことを保証するために, 一般

```
 1: #include <mach.h>
 2: #include <rt/mach_timer.h>
 3: #include <rt/rt_thread.h>
 4:
 5: mach_port_t               port;
 6: int                       myfunc(), deadlinefunc();
 7: int                       iteration = 0;
 8:
 9: main()
10: {
11:    thread_t               new_thread, new_thread2;
12:    rt_thread_attr_data_t  thread_attr,thread_attr2;
13:    char            *thread_message = "In thread...";
14:    task_t                 mytask = mach_task_self();
15:    kern_return_t          ret;
16:
17:    ret = rt_thread_attribute_init( 1, /* priority */
18:                      0, /* period (in seconds) */
19:               30000000, /* period (in nanoseconds) */
20:                 myfunc, /* thread name */
21:        thread_message, /* new thread's args */
22:                  &port, /* port */
23:          &thread_attr); /* rt_thread template */
24:    if (ret != KERN_SUCCESS) {
25:       mach_error("rt_thread_attribute_init:",ret);
26:       exit(0);
27:    }
28:    ret = rt_thread_attribute_init( 1, 0, 0,
29:                 deadlinefunc, /* deadline handler */
30:                  &new_thread, /* new thread's arg */
31:             0, &thread_attr2); /* rt_thread template */
32:    if (ret != KERN_SUCCESS) {
33:       mach_error("rt_thread_attribute_init:",ret);
34:       exit(0);
35:    }
36:    printf("starting deadline thread\n");
37:    ret = rt_thread_create(mytask, &new_thread2,
38:                                    &thread_attr2);
39:    if (ret != KERN_SUCCESS) {
40:       mach_error("rt_thread_create:",ret);
41:       exit(0);
42:    }
43:    printf("starting periodic thread\n");
44:    ret = rt_thread_create(mytask, &new_thread,
45:                                    &thread_attr);
46:    if (ret != KERN_SUCCESS) {
47:       mach_error("rt_thread_create:",ret);
48:       exit(0);
49:    }
50:    while(1){ sleep(100000); }
51: }
52:
53: myfunc(message)
54: char *message;
55: {
56:    printf("%s iteration %d\n",message, iteration++);
57: }
58:
59: deadlinefunc(thread)
60: thread_t          *thread;
61: {
62:    int          missed;
63:    timespec_t   time;
64:
65:    while(1) {
66:       expire_wait(port, &time);
67:       printf("Missed Deadline, secs = %d, nsecs = %d",
68:               time.seconds,time.nanoseconds);
69:       printf("  total = %d\n", ++missed);
70:       thread_resume(*thread);
71:    }
72: }
```

図 2: デッドラインハンドラを使ったリアルタイムスレッドのプログラム例

的な FIFO 方式の同期機構を提供している. しかし, この方式では優先度の低いスレッドがクリティカルセクションを実行中の時は, 優先度の高いスレッドの実行が無制限に遅れさせられてしまうことがある. これをプライオリティインバージョン (**priority inversion**) と呼ぶ [7]. RT-Mach では, lock 変数ごとにプライオリティ継承プロトコルのようなリアルタイム同期プロトコルをユーザが選択することができるのでプライオリティインバージョンの問題を避けることができる.

同期プロトコルは lock 変数の待ち行列のキューイングの方法に関して, 1) FIFO 方式, 2) 優先度に基づく方式, の 2 方式が考えられる. また, キューの中の優先度を実行中のスレッドに継承させるか, 継承させないかについて 2 方式がある. クリティカルセクションを実行中のスレッドの先取りに関しては, 1) 先取りしない, 2) 先取りする, 3) 実行中のスレッドを強制的に中止させる, の 3 方式が考えられる (図 3). リアルタイム同期のポリシはこれらの組合せにより以下にあげる 5 つが提供されている.

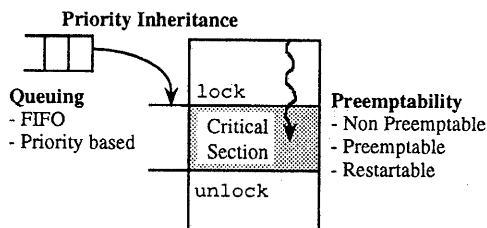

図 3: リアルタイム同期

Kernelized Monitor (KM): クリティカルセクションを実行中のスレッドは先取りされない.

Basic Priority (BP): スレッドがクリティカルセクションを実行中でも, 先取りを許す. 同期待ちキューはスレッドの優先度順である.

Basic Priority Inheritance Protocol (BPI): あるスレッドがクリティカルセクションを実行中に, そのスレッドより高いプライオリティを持つスレッドが同期待ちになった場合, すでに実行中のスレッドは待ちにあるスレッドのの中の最高のプライオリティを継承する.

Priority Ceilling Protocol (PCP): BPI を拡張したプロトコルである. 各ロックに対して, そのロックを獲得する可能性のある全スレッド中の最高プライオリティをそのロックに対するプライオリティの上限値とする.

Restartable Critical Section (RCS): クリティカルセクションを実行中のスレッドよりプライオリティの高いスレッドが同期待ちになった場合, 現在実行中のプライオリティの低いスレッドの処理を中止さ

せ, クリティカル資源の状態をもとに戻し, 待ち行列に戻す.

RT-Mach は, lock 変数を使った相互排除 (mutual exclusion) を元にしたリアルタイム同期プリミティブを提供している. 同期プロトコルはロック変数属性 mutex_attr によって指定する.

---

rt_mutex_allocate(&mutex, mutex_attr)
rt_mutex_deallocate(mutex)
rt_mutex_lock(mutex, timeout, context)
rt_mutex_unlock(mutex)
rt_mutex_control(mutex, cmd, arg, size)

2.4 リアルタイムプロセス間通信

Mach 3.0 ならびに RT-Mach のプロセス間通信は, カーネル内のポートを通して行なわれる. 送信元からのメッセージはポートにキューされ, 送信先のタスクへ渡される. Mach 3.0 では, 送受信の順序は FIFO である. この方式の欠点は, すでにキューイングされている全てのメッセージの処理が終了するまで, あとから待ちに入った優先度の高いメッセージの処理が待たされてしまうことである. そこで, RT-Mach ではポートのメッセージキュー待ちにおいて発生するこのようなプライオリティインバージョンの問題を回避するためにリアルタイムプロセス間通信を提供している. ユーザは, 以下に示す 3 つのポリシを指定できる.

2.4.1 メッセージデスパッチポリシ

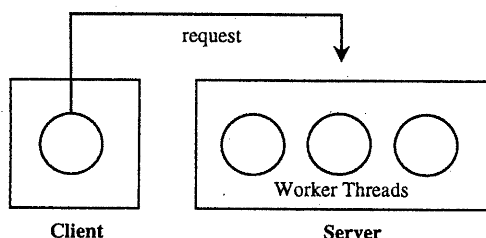

図 4: メッセージデスパッチポリシ

サーバが複数のワーカースレッドから構成される場合, クライアントスレッドからどのワーカースレッドへメッセージを配送するか決定するポリシを指定する (図 4).

FIFO 空いているサーバスレッドへ FIFO 方式でメッセージを配送する.
WORKER サーバはあらかじめ優先度付けされているワーカースレッドから構成され, クライアントの優先度に応じて対応するワーカースレッドへメッセージを配送する

2.4.2 メッセージキューイングポリシ

キュー内のメッセージの順序を決定するポリシを指定する.

FIFO FIFO 方式でキューイングする.
PRIO 優先度順にキューイングする.
BPI 優先度継承をする.

2.4.3 メッセージハンドオフポリシ

クライアントスレッドが持っている優先度を, サーバスレッドへ伝達させる方式を決定するポリシを指定する.

HANDOFF ON 優先度をサーバへ継承する.
HANDOFF OFF 優先度をサーバへ継承しない.

ポリシはポート属性で指定する. プリミティブを以下に示す.

---

rt_mach_port_allocate(task, right, port_attr, &port)
rt_mach_port_associate(thread, port)
rt_mach_msg(msg, option, send_size, rcv_size,
rcv_port, timeout, notify)

---

3 リアルタイムサーバ

Mach 3.0 と同様に, RT-Mach 上でも UNIX サーバ [5] や X ウィンドウシステムを稼働させることができる (図 5). ただし, UNIX サーバや X ウィンドウシステムはクライアントからの要求を FIFO 方式に基づいて処理している. つまり, リアルタイム処理機能を利用して書かれたクライアントアプリケーションであっても, これらのサーバの機能を呼出した場合, サーバ内の処理におけるプライオリティインバージョンが原因でリアルタイム性が保証されなくなることがある. そこで, 連続メディアを扱うためのメディアサーバなどは, リアルタイム同期機構やリアルタイムプロセス間通信などの RT-Mach が提供するリアルタイム処理機能を使って実現しなければならない. 現在, RT-Mach 上で稼働するリアルタイムサーバは, **Real-Time Server (RTS)** と **Network Protocol Server (NPS)** である [13](図 6).

3.1 Real-Time Server (RTS)

RTS は, プロセス制御, ファイル管理, TTY 管理などを備え, ユーザにプログラム実行環境を提供する. クライアントスレッドからのメッセージはリアルタイムプロセス

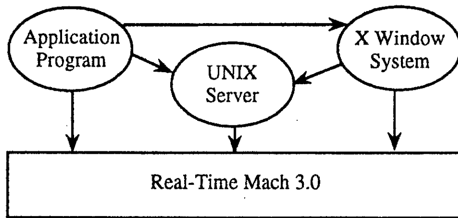

図 5: RT-Mach 上の UNIX サーバ

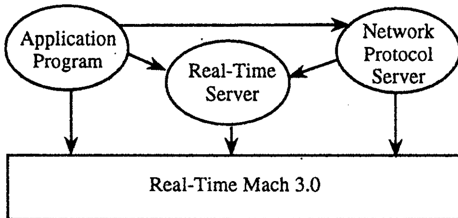

図 6: RT-Mach 上のリアルタイムサーバ

間通信を使ってサーバ内のスレッドに伝えられる (図 7). サーバ内の処理は, クライアントの優先度を反映したリアルタイム処理がなされる.

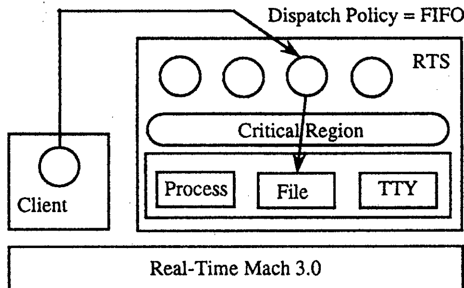

図 7: Real-Time Server (RTS) の構成

3.2 Network Protocol Server (NPS)

NPS は RTS 上で動作するリアルタイムネットワークサーバである. サーバ内には, ワーカスレッドと呼ばれる優先度に基づくリアルタイムスレッドがある. ユーザスレッドからの処理は, リアルタイムプロセス間通信を使ってサーバ内のワーカスレッドに伝えられる. 一方, カーネルがネットワークからパケットを受信すると, そのパケットはその優先度に対応するワーカスレッドに渡される. このように NPS では優先度に基づくプロトコル処理が実現されている (図 8).

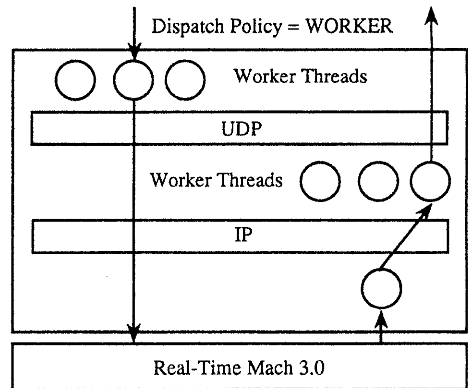

図 8: Network Protocol Server (NPS) の構成

表 1: スレッドの切替え時間

Mach 3.0 Kernel Thread	32 μsec
RT-Thread (Mach Timesharing)	34 μsec
RT-Thread (Fixed Priority)	39 μsec
RT-Thread (Rate Monotonic)	41 μsec

4 性能評価

Intel i486DX2/66MHz[1] を搭載した IBM PC AT[2] 互換機 Gateway 2000 4DX2-66V で 250 ナノ秒精度のタイマボード[3] を使ってカーネルの基本性能とプリミティブについて性能を測定した. 実験には, RT-Mach MK78, UNIX サーバ UX39 を使用した.

4.1 カーネルの基本性能

4.1.1 スレッドの切替え時間

リアルタイムスレッドを明示的に切替えるのに要する時間を測定した. 現在の RT-Mach のリアルタイムスレッドは, C-Thread[3] と同様なインタフェースを提供しているが, C-Thread のようにユーザレベルでは動作させることはできず, 必ずカーネルレベルで動作する. つまり, スレッドを切替えるためには, ユーザ空間とカーネル空間の切替の時間が必要となる. 表 1が示すように, スレッドの切替え時間は決して小さくはない. ユーザレベルのスレッド [15][16] を実現することによりスレッド処理の高速化が期待できる.

4.1.2 割込み処理時間

RT-Mach カーネルの割込み処理に要する時間をキーボード割込み, ネットワーク処理, ディスク処理に関して

[1] Intel, i486DX はインテル社の商標です.
[2] IBM, Personal Computer AT は IBM Corp. の商標です.
[3] タイマーボードのカウンタを操作したり, データを読み出すときのオーバヘッドは, 約 2 マイクロ秒であった.

表 2: 割込み処理時間

	最小実行時間	最大実行時間
キーボード割込み	6.00 μsec	270.25 μsec
Ethernet パケット受信	106.25 μsec	487.25 μsec
Ethernet パケット送信	20.50 μsec	132.00 μsec
IDE ディスク Read と Write	43.50 μsec	251.00 μsec

測定した. RT-Mach の割込み処理はカーネル内の割込みハンドラで実行され, その処理は先取りされることがない. 表 2に示すように, これらの割込み処理時間はリアルタイム処理を実現する上で無視できないほど大きい.

4.1.3 プリエンプション時間

外部イベントによって起動された優先度の高いスレッドが, 実行中の優先度の低いスレッドを先取りし, 実際に実行されるまでの時間を測定した (図 9). 割り込まれたスレッドの処理が停止するまでに 38.50μsec (区間 A), 次に実行するスレッドを選択するのに 6.75μsec (区間 B), 選択したスレッドをディスパッチするのに 8.00μsec (区間 C), コンテキストスイッチに 19.25μsec (区間 D) であった. 外部イベントが発生してから, 実際に割込みが認識されるまでの時間は測定していない.

4.2 RT-Mach プリミティブの性能

4.2.1 メモリ管理

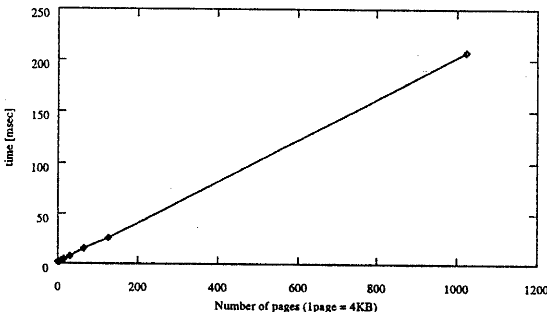

図 10: vm_wire() のコスト

遅延評価 (Lazy Evaluation) 手法に基づくメモリ管理では, ページフォルトが起こった時の処理のコストがあらかじめ計算できない. そこで, リアルタイムスレッドが使うメモリオブジェクトのアクセス時間を保証するためには, vm_wire() を使ってあらかじめ実記憶域に固定しておかなければならない. しかし, 表 10が示すように, 連続メディアを現在のメモリオブジェクトに格納し, 実記憶域へ固定しておくには大量のメモリが必要で, かつ, 処理時間がかかるのでこの方法が現実的ではないことがわかる.

4.2.2 リアルタイムスレッド

表 3: リアルタイムスレッド

rt_thread_create()	460 μsec
thread_resume()	116 μsec
rt_thread_attribute_init()	1,560 μsec
rt_thread_deadline_handler()	2,646 μsec

リアルタイムスレッドの生成, 初期化に関する実行時間を測定した (表 3). ライブラリ呼び出しは, スタックの確保, その実記憶への固定などの処理を行なっている. 動的にリアルタイムスレッドを生成するコストは大きいことがわかる.

4.2.3 リアルタイムファイルアクセス

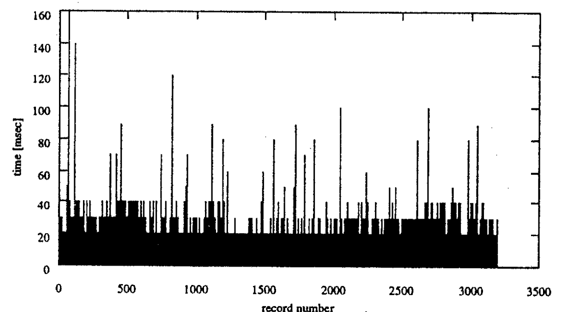

図 11: 固定長データのシーケンシャル読込み

現在, RT-Mach はアクセス時間を保証するリアルタイムファイルシステムを備えていない. そこで, UNIX サーバの fread()[4] を使ってファイルから 10KB ずつ読込む実験を行なった. これは, 連続メディアをハードディスク装置から読込み, 再生する場合[5] を想定してしている. 実験機には, IDE 340MB のハードディスク装置が内蔵されている. 結果を図 11に示す. ディスクのアクセス時間はばらつきが大きく値がバウンドされていないことがわかる. ファイルアクセスに関するサービスの質が保証できる機構を実現する必要がある.

5 おわりに

本稿では, 分散マルチメディアシステムを構築するために分散リアルタイムカーネルに対して要求される諸機能について RT-Mach を評価した. RT-Mach が提供するリアルタイムスレッド, リアルタイムスケジューラ, リアルタイム同期機構, リアルタイムプロセス間通信は, 分散マルチメディア環境において連続メディアを扱うのに有効な機能であることが確認できた.

[4] buffered binary input

[5] 320 × 240 24bit カラーの静止画を JPEG を使って圧縮した時, データのサイズが平均 10KB であった.

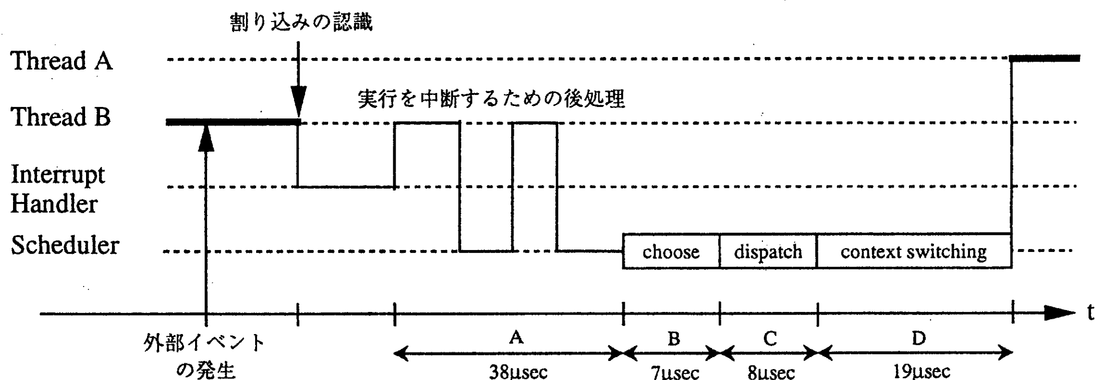

図 9: プリエンプション時間

今後の課題は, ユーザレベルリアルタイムスレッドとユーザレベルスケジューラの実現, アクセス時間が保証できる実時間メモリオブジェクトの実現, ユーザレベルのI/Oドライバの開発, ST-II[4] のような実時間プロトコルの開発とリアルタイム処理機能を使ったプロトコル処理の実装などがある.

6 謝辞

本研究を行なうにあたり御指導頂いた慶應義塾大学環境情報学部の斎藤信男教授に感謝致します. また, 性能評価実験に協力して頂いた開放型基盤ソフトウェア研究開発評価事業「マルチメディア統合環境基盤ソフトウェア」プロジェクトの皆様に感謝致します.

参考文献

[1] C. L. Liu and J. W. Layland: "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," Journal of ACM, Vol. 20, No. 1, pp. 46–61 (1973).

[2] M. J. Accetta, W. Baron, R. V. Bolosky, D. B. Golub, R. F. Rashid, A. Tavanian and M. W. Young: "Mach: A New Kernel Foundation for Unix Development," Proc of USENIX Summer Conference, pp. 93–112 (1986).

[3] E. C. Cooper and R. P. Draves: "C Threads," Technical Report CMUCS-88-154, Department of Computer Science, Carnegie Mellon University (1987).

[4] C. Topolcic: "Experimental Internet Stream Protocol, Version 2 (ST-II)," CIP Working Group, RFC 1190 (1990).

[5] D. Golub, R. Dean, A. Forin and R. Rashid: "Unix as an Application Program," Proc of USENIX Summer Conference, pp. 87–95 (1990).

[6] H. Tokuda, T. Nakajima and P. Rao: "Real-Time Mach: Towards a Predictable Real-Time System," Proc of USENIX Mach Workshop, pp. 73–82 (1990).

[7] L. Sha, R. Rajkumar and J. P. Lehoczky: "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Trans Computers, Vol. 39, No. 9, pp. 1175–1185 (1990).

[8] 徳田: "分散リアルタイム OS の技術動向," コンピュータソフトウェア, Vol. 9, No. 3, pp. 4–14 (1992).

[9] H. Tokuda, Y. Tobe, S. T.-C. Chou and J. M. F. Moura: "Continuous Media Communication with Dynamic QOS Control Using ARTS with and FDDI Network," Proc of ACM SIGCOMM'92, pp. 88–98 (1992).

[10] 徳田, 斎藤: "マルチメディア統合環境プロジェクトにおけるリアルタイム処理技術," 情処研報, Vol. 92, No. 20, 93-ARC-99, pp. 9–15 (1993).

[11] 斎藤, 徳田, 萩野, 追川, 天明, 緒方, 大町, 和田, 堀切, 平林, 多田, 藤井, 矢崎, 薄, 田中, 人見, 小野, 南部: "マルチメディア統合環境のテストベッドとその評価," 情処研報, Vol. 92, No. 20, 93-ARC-99, pp. 17–24 (1993).

[12] 緒方, 人見, 和田, 追川, 徳田: "Real-Time Mach 3.0 の評価と改良," 情処研報, Vol. 92, No. 20, 93-ARC-99, pp. 61–68 (1993).

[13] T. Nakajima, T. Kitayama and H. Tokuda: "Experiments with Real-Time Servers in Real-Time Mach," Proc of USENIX 3rd Mach Symposium, pp. 1–19 (1993).

[14] S. Savage and H. Tokuda: "RT-Mach Timers: Exporting Time to the User," Proc of USENIX 3rd Mach Symposium, pp. 111–118 (1993).

[15] B. Davis, et al.: "Adding Scheduler Activations to Mach 3.0," Proc of USENIX 3rd Mach Symposium, pp. 119–136 (1993).

[16] R. Dean: "Using Continuations to Build a User-Level Threads Library," Proc of USENIX 3rd Mach Symposium, pp. 137–152 (1993).