

リアルタイム・マルチスレッド・プログラミング支援ツールの開発

水沼一郎 大西秀次 島川博光 竹垣盛一

三菱電機産業システム研究所

(661) 兵庫県尼崎市塚口本町 8-1-1

従来、リアルタイムシステムの開発において、その動的挙動や性能の解析はアドホックに行なわれることが多く、S/W生産性の向上を妨げていた。近年、高性能なリアルタイムOSが開発され、また、IEEE Computer Societyの定めるPOSIX 1003.4によって、リアルタイムOSの標準化が行なわれているが、このようなリアルタイムOS上でのリアルタイムシステムの構築技術は必ずしも確立されていない。我々は、POSIX 1003.4準拠のリアルタイムOSと、その上に構築されるアプリケーションとの間に介在するミドルウェアの形で、リアルタイムシステムの開発を支援するツールを開発している。また、このツールを用いて実際にリアルタイムシステム(時系列データサーバ)を構築した例をあげる。

Multi-Thread Programming Tools for Real-Time Systems

Ichiro Mizunuma Hideji Onishi Hiromitsu Shimakawa Morikazu Takegaki
Industrial Electronics & Systems Lab. Mitsubishi Electric Corp.
8-1-1 Tsukaguchi-Honmachi Amagasaki, Hyogo, Japan

Up to now, in developing real-time systems, the behavior and the performance of the system have been analyzed adhoc in many cases, which has prevented the software productivity from increasing. Recently, high-performance real-time operating systems are developed, and IEEE Computer Society proposes POSIX 1003.4 as a standard of real-time operating systems. However, technique is not necessarily established for building real-time systems on such real-time operating systems. We develop the tools which support building real-time systems as middleware between applications and real-time operating systems based on POSIX 1003.4. Using our tools, we build a real-time data server. It illustrates features of the tools.

1 はじめに

従来、制御用リアルタイム・システムなどのためのリアルタイム・プログラムの開発において、システムの動的挙動や性能の解析はアドホックに行なわれることが多く、これがソフトウェアの生産性の向上を妨げる原因となっていた。近年、高速なタスクスイッチ機能、優先度に基づくスケジューリング機能、リアルタイム処理に適した同期機構などの機能を備えた高性能リアルタイムOSが開発されている。また、リアルタイムOSの標準化を目指したものとして、IEEE Computer Societyが定めたPOSIX 1003.4がある。POSIX 1003.4に準拠したリアルタイムOSは、ソフトウェア開発環境面におけるUNIX互換性を持っており、また、リアルタイム処理を行なうためにマルチスレッド機能、優先度に基づくスケジューリング機能、優先度継承機能[Rajkumar91]を持ったセマフォ機能、優先度付きスレッド間メッセージ通信機能、リアルタイムクロック機能、シグナルやイベントによる割り込み機能などの機能を備えている。しかしながら、これらの新しいリアルタイムOS上でのリアルタイム・システム構築技術は、必ずしも確立されていない。

本稿では、従来のリアルタイム・システム構築

における問題点をあげた後、これらの問題点を解決するために我々が提案する、リアルタイム・マルチスレッド・プログラミング支援ツール(MPTR: Multi-Thread Programming Tools for Real-Time Systems)について述べる。我々はこのMPTRをPOSIX 1003.4準拠のリアルタイムOSと、その上に構築されるリアルタイム・アプリケーションとの間に介在するミドルウェアの形で提供する。MPTRの副次的効果として、システム開発の初期段階で、システムの全体ユーティライゼーションを容易に見積もることができることがあげられる。

さらに、本稿ではリアルタイム・システム(時系列データサーバ)を構築した例をあげ、その構築にあたってMPTRを利用することによって得られた効果について述べる。

2 解決すべき問題点

従来のリアルタイム・システム、およびその開発においては、以下にあげるような問題点があった。

2.1 スケジュール可能性の判定

複数の処理が非独立に動作するシステムの開発において、各処理(タスク)の起動、停止、中断、再開などのタイミングについて、設計者が何の規範もなく直接、アプリケーションに依存した形で記述すると、システム全体の実行時の動作を予測することが困難となり、スケジュール可能性のチェックは、実際にシステムを動作させた結果によって行なうことになる。このようなアドホックなシステム開発の方法では、ソフトウェア生産性を向上させることはできない。

2.2 ネットワーク通信における問題点

TCP/IP プロトコル通信のようなストリーム通信は、送受信間でコネクションを張るため片側の処理の遅れが他方に影響を及ぼすので、リアルタイム性を確保するのが困難である。UDP/IP プロトコル通信のようなデータグラム通信は、送受信されるパケットが損失する可能性があるので、アプリケーション・レベルで信頼性を確保する必要がある。

2.3 動的資源割り当てによる問題点

マルチスレッド・プログラムの実行では、スレッドの生成、消滅が繰り返される。生成する側のスレッドから生成される側のスレッドへ受け渡す情報を格納する領域(引数領域と呼ぶ。)や、生成されたスレッドのためのスタック領域は、必要になった時点で動的に確保され、不要になった時点で解放される。特に、生成する側のスレッドから受け渡される情報を格納する領域は一般に不定長である。このような領域の動的確保、解放を繰り返すことに伴って生成されるゴミ領域を再利用するためにガーベジ・コレクションが必要となるが、これは大きなCPU時間を必要とする処理であり、さらにその発生タイミングは予測困難である。よって、システムのリアルタイム性が失われる可能性がある。

3 MPTR

第2章で述べた問題点を解決するために、我々は、リアルタイム・マルチスレッド・プログラミング支援ツール(MPTR)の研究、開発を行っており、本支援ツールをPOSIX 1003.4 準拠のリアルタイムOSと開発すべきリアルタイム・アプリケーションとの間に介在するミドルウェアの形で提供する。

```
typedef struct {
    pthread_t *tid;      /* スレッド ID */
    time_t interval;     /* 周期 (× 10 m sec) */
    time_t deadline;     /* デッドライン (同上) */
    time_t offset;       /* オフセット (同上) */
    struct timeval startTime; /* 開始時刻 */
    /* = PERIODIC_NOW の場合即開始 */
    void (*func)();      /* 周期毎に実行する関数 */
    void *arg;           /* func() への引数 */
    void (*timeout)();   /* タイムアウトハンドラ */
    void *timeout_arg;   /* timeout() への引数 */
    int prio;            /* スレッドの優先度 */
} periodic_thread_t;
```

図 1: 周期スレッドの属性を格納する構造体

3.1 周期スレッド生成ライブラリ

リアルタイム・プログラムにおけるスレッドは、大きく、周期スレッドと非周期スレッドに分類できる。周期タスクのデッドラインを満たすためのスケジューリング方法として、Rate Monotonic (RM) スケジューリング [Liu,Layland73] [Cheng87] がある。また、RM スケジューリングは、各タスクが互いに独立であるという理想的な場合を前提としているが、非独立の場合でも、優先度継承プロトコルを用いて相互排除による影響を極力少なくすることにより、RM によるスケジューリングが可能となる [Rajkumar91]。Real-Time Mach では、リアルタイム・プログラミングのモデルとして、RT-thread が提供されている [Tokuda91]。我々はこれをベースにして、POSIX 1003.4 上で、周期スレッド生成ライブラリを設計した。

3.1.1 インタフェース

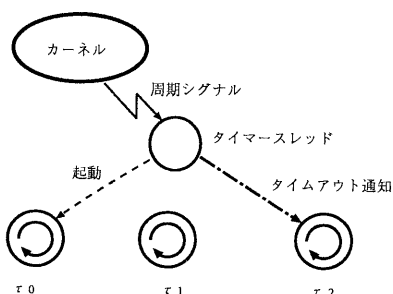
周期スレッドの属性を格納するための構造体 `periodic_thread_t` を図 1 のように定義する。この構造体のメンバに必要な情報を書き込んだ後に、その構造体へのポインタを引数とするライブラリ関数 `PeriodicThreadCreate()` をコールすることにより、周期スレッドを生成することができる。

```
int PeriodicThreadCreate(periodic_thread_t attr)
```

本ライブラリでは、周期スレッドの生成に際してタイムアウト時の例外処理の登録を義務づけている。これによりプログラム保守が容易になる。

3.1.2 実現方法

POSIX 1003.4 準拠の OS では、各スレッド毎に独立なインターバル・タイマー (OS カーネルからの周期シグナル) を用いることができる。OS カーネルからの周期シグナルを受信するタイマースレッドを設け、これが、全ての周期スレッドの動作タイミングを管理する (図 2 参照)。



	I	I'	D	D'	S
τ_0	10	10	10	10	wait
τ_1	20	10	10	10	wait
τ_2	50	25	25	25	run
τ_3	100	50	50	50	wait
...	...	...	...	...	...

スレッド管理テーブル

I : 周期
 I' : 周期までの時間
 D : デッドライン
 D' : デッドラインまでの時間
 S : 各周期スレッドの状態
 (run: 実行中, wait: 起動待ち)
 $\tau_1, \tau_2, \dots$: 周期スレッド

図 2: 周期スレッドの実現方法

スレッド管理テーブルは、各周期スレッドの周期、デッドライン、オフセットなどの動作時間に関するパラメータ、および、次の周期までの時間、デッドラインまでの時間などの各周期スレッドの状態などの情報を格納する。タイマースレッドはすべての周期スレッドの動作時間に関するパラメータの最大公約数に等しい周期で起動され、その周期毎に、待機している各周期スレッドが起動すべき時刻になっているかを調べ、該当するものに対して起動シグナルを送信する。また、実行中の各周期スレッドについてデッドラインを越えていないかどうかを調べ、該当するものに対して、割り込みシグナルによってタイムアウトの通知を行なう。通知を受けた周期スレッドは例外ハンドラへ制御を移す。

スレッド管理テーブルは、タイマースレッドだけでなく、周期スレッドを生成、消滅する他のスレッドからもアクセスされる。このテーブルへのアクセスに対する相互排除において、タイマースレッドの最悪ブロック時間 B_{timer} が大きくならないことが要求される。我々は、 B_{timer} を小さく抑えるための相互排除のアルゴリズムを開発した。

以上述べた実現方法の特徴として、ユーザはタイマースレッドの存在を意識する必要がないこと、タイマースレッドが全ての周期スレッドの動作時刻を管理しているので、システム・クロックを他のシステムのそれに合わせる機構の実現が容易であること

などがあげられる。

3.2 リアルタイム通信ライブラリ

リアルタイム・システムにおけるネットワーク通信には、a) 送受信に関する処理が、より優先度の高い他のリアルタイム処理を妨げないこと、b) 時間上の制約の厳しいメッセージの送受信を、時間上の制約のあまり厳しくないメッセージの送受信に優先させること、c) 送受信されるメッセージの欠損を極力抑える、あるいは、あまり重要でないメッセージの欠損はあっても、重要なメッセージの欠損は起こらないことなどが要求される。これらの要求を満たすために、リアルタイム性を持ち、かつ、データの損失を極力抑える、データグラム通信プロトコルを用いるネットワーク送受信機構を提案し、それらをシステム上に構築するためのライブラリを提供する。

3.2.1 レシーバ生成ライブラリ

ネットワークを介してシステムに送られるメッセージの受信処理を行なう機構をレシーバと呼ぶ。システムに対するネットワークからのメッセージの受信はすべてレシーバを介して行なわれる。レシーバは、レシーバ・スレッド、および、受信したメッセージを一時格納するためのメッセージキューからなる。

データグラム通信ではメッセージの欠損のおそれがあるので、受信側にメッセージが届いた際に、必ずレシーバ・スレッドが起動される必要がある。よって、レシーバ・スレッドには、システム内の他のスレッドよりも高い優先度を持たせる。欠損の可能性をより少なくするために、レシーバ・スレッドを複数用意してもよい。

レシーバ・スレッドの処理量が多いと、その優先度が高いため他の処理を大きく妨げてしまうので、レシーバ・スレッドの処理は軽い必要がある。レシーバ・スレッドはメッセージ受信後、それを即座にキューに入れ、次のメッセージ到着を待つ。

システムに送られるメッセージには、あまり重要でなく、それが受信されないことよりも、その受信によって他のリアルタイム処理が妨げられることを避けたいようなものがある。このような要求に応えるためには、メッセージの優先度に応じた、それぞれ異なるポートを用いるレシーバ・スレッドを複数個用意する。

以上述べた方法により、受信に際するメッセージ

の欠損を極力減らすことができるが、欠損を完全になくすためには、送受信両者で受信の確認を行なうなど、アプリケーションレベルでのサポートが必要であろう。

以上述べたレシーバの機構は、我々の開発したライブラリを用いて容易に実現できる。

3.2.2 トランスミッタ生成ライブラリ

システムからネットワークを介して外部システムに送られるメッセージの送信処理を行なう機構をトランスミッタと呼ぶ。システムからネットワークへのメッセージの送信はすべてトランスミッタを介して行なわれる。トランスミッタは、トランスミッタ・スレッド、および、送信するメッセージを一時格納するためのメッセージキューからなる。リアルタイム性を要求されるメッセージの送信を、そうでないメッセージの送信に優先させるために、メッセージに優先度を持たせ、各優先度に応じたメッセージキューを複数個用意し、優先度の高いメッセージキューのメッセージを優先して送信するようにする。

送信されるメッセージは、その優先度に応じた優先度を持つトランスミッタ・スレッドによって送信されるべきである。さもなくば、送信処理がより高い優先度を持つ送信以外の処理を妨げたり、より低い優先度を持つ送信以外の処理によって妨げられたりすることが起こりうる。POSIX 1003.4 準拠の OS では、あるスレッドの優先度を、キュー待ちしているメッセージの優先度のうちの最高の優先度と等しい値に継承させる機能はない。よって、優先度別に用意したメッセージキュー各々に対して、等しい優先度を持つトランスミッタ・スレッドを持たせる複数のトランスミッタ・スレッドが同時にネットワーク資源へのアクセスを要求する可能性があるので、相互排除のために優先度継承機能を持ったセマフォを設ける。これにより、優先度の高い通信処理の待ち時間を極力抑えることができる。

以上述べたトランスミッタの機構は、我々の開発したライブラリを用いて容易に実現できる。

3.3 スレッドのプール化

第 2.3 節で述べた、動的資源割り当てによる問題を解決するために、スレッドのプール化手法を提案する。スレッドのプール化手法とは、生成されるスレッドのためのメモリ領域をシステムの初期化時に、その実行中に必要となる最大量（システムの設

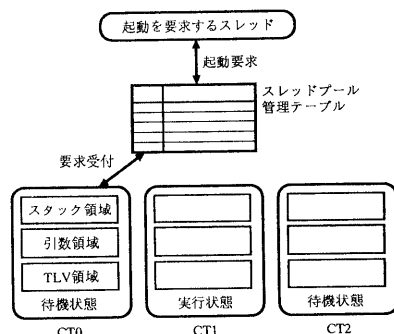


図 3: スレッドプールの構成

計時に開発者によって静的に見積られる。) だけあらかじめ確保しておき、スレッドの生成（起動）要求に対して必要な領域を割り当てることにより、リアルタイム性を保ってスレッドの生成、消滅を行なう手法である。スレッドのプール化手法を実現する機構をスレッドプールと呼ぶ。

3.3.1 スレッドプールの構成

スレッドプールはスレッド管理テーブル、および、複数のチャイルド・スレッド (CT) から構成される (図 3 参照)。

CT は、スレッドの起動要求に対して、処理を受け付け、実行するスレッドである。各 CT は、まだ処理を受け付けておらず起動要求を待っている待機状態か、処理を受け付け実行中の実行状態のいずれかの状態にある。各 CT は、スタック領域、引数領域、後述するスレッドローカル変数 (TLV) のための領域などのメモリ資源、および、実行する関数、実行の優先度などに関する情報を持つ。スレッドプール管理テーブルは、スレッドプール全体に関する情報を格納し、CT と、その CT に対して起動を要求するスレッドとの間での同期や通信のために用いられる。起動を要求するスレッドは、スレッドプール管理テーブルに対して起動要求を出し、待機状態の CT があれば、そのうちの一つがこの要求を受け付けたことをスレッドプール管理テーブルを通して知らせる。

3.3.2 スレッドプール・ライブラリ

スレッドプールの生成、あるいは、CT の起動、停止を容易に記述できるようにするために、スレッドプール・ライブラリを用意した。スレッドプール

```

#define STACK xxxx /* スタックサイズ */
#define PRI0 yyyy /* 優先度 */
struct Arg_t { /* 引数の型 */
};
...
struct TLV_t { /* TLV 構造体 */
};
...
CreateThread()
{
    struct Arg_t SendArg;
    int poolID, tid;
    poolID
        = ThreadPoolInit(5, STACK, sizeof(Arg_t),
            sizeof(TLV_t));
    /* チャイルド・スレッドを 5 個生成する */
    SendArg.??? = ???; /* 引数の設定 */
    tid
        = ThreadPoolRequest(poolID, &SendArg,
            BodyOfThread, PRI0);
}
...
BodyOfThread(RecvArg)
struct Arg_t *RecvArg;
/* 引数領域へのポインタ */
{
    ??? = RecvArg -> ???;
    /* 受け取った引数にアクセス */
}

```

図 4: スレッドプール・ライブラリの使用例

の生成は、ライブラリ関数 `ThreadPoolInit()` によって記述する。返り値は生成されたスレッドプールの ID である。

```

int ThreadPoolInit(
    int ThreadMax, /* CT の最大数 */
    int StackSize, /* 各 CT のスタックサイズ (byte) */
    int ArgSize, /* 各 CT の引数領域のサイズ (byte) */
    int TLVSize) /* 各 CT の TLV 領域のサイズ (byte) */

```

CT の起動は、ライブラリ関数 `ThreadPoolRequest()` によって記述する。返り値は起動要求を受けた CT の ID である。

```

int ThreadPoolRequest(
    int ThreadPoolID, /* 起動を要求するスレッドプールの ID */
    void (* func)(), /* CT に実行させる関数 */
    char *SendArg, /* 関数 func への引数 */
    int prio) /* 実行の優先度 */

```

図 4 にこれらのライブラリの使用例を示す。

3.4 スレッドローカル変数 (TLV)

スレッドローカル変数 (TLV) という概念を提案する。TLV とは、あるスレッドの実行する全ての関数から見る事ができ、かつ、他のスレッドからは見る事のできない変数である。

関数間でデータのやりとりを行なう場合、一般に、大域変数を用いる方法か、あるいは、引数によって共通領域へのポインタを渡す方法がとられる。マルチスレッド・プログラムにおいて、ある関数 `f()` を複数のスレッドが同時に実行する可能性がある場合、`f()` は再入可能である必要がある。しか

```

f()
{
    struct st A;
    struct st *a = &A;
    ...
    a->member_1 = x;
    g(a);
    ...
}
g(a)
struct st *a;
{
    h(a);
    ...
}
h(a)
struct st *a;
{
    ...
    y = a->member_1;
    ...
}

```

a) TLV を使用しない場合 b) TLV を使用する場合

図 5: TLV を用いるプログラム例

し、`f()` と他の関数とのデータの受渡しに大域変数を用いると、関数 `f()` は再入可能ではなくなる。

一方、引数によってポインタ渡しをする方法の場合、プログラムの記述が複雑になるという問題点がある。図 5(a) のプログラム例では、関数 `f()` とその孫関数 `h()` の間でのデータの受渡しが行なわれる。st 構造体 A を `f()` のオートの変数として確保し、構造体 A へのポインタ `a` を、`f()` からの `g()` の呼び出し、および、`g()` からの `h()` の呼び出しで引数として伝える。ここで、構造体 A にアクセスしない `g()` の記述においても、ポインタ `a` の存在を意識しなければならない。この問題は、関数のネスト構造がより深く、より複雑になるほど深刻になる。

以上のような問題は、各スレッド毎に独立した大域変数のような機能を持つ変数、すなわち、TLV を用いることにより解決できる。TLV を格納する領域は、スレッドプールの生成の際に、各 CT 毎に確保される。各 CT はそれぞれに割り当てられた TLV のための領域へのポインタを知る必要がある。ライブラリ関数 `getTLV()` は、この関数をコールした CT の TLV 領域へのポインタを返す。`getTLV()` を用いて、図 5(a) のプログラムを書き直したものを図 5(b) に示す。TLV にアクセスする関数 `f()`、`h()` では、その先頭でマクロ定義 `USETLV;` により TLV へのポインタを構造体へのポインタ `a` に代入する。そして、あたかも大域変数 “`a->member_1`” にアクセスするように TLV へアクセスし、他関数とのデータのやりとりを行なう。また、TLV にアクセスしない関数 `g()` では TLV に関する記述は不要である。ただし、マクロ定義 `USETLV` は次のように定義されている。

```

#define USETLV; struct st *a = (struct st *)getTLV();

```

また、TLV を用いて、シグナル等による割り込

みによって起動されるハンドラと、割り込まれた関数の間でのデータの受渡しを容易に実現することができる。ハンドラ側で、割り込まれた関数の、割り込みが発生時点での実行箇所、各変数の値などを知らことができ、元の関数へ制御を移す際に、適切な処理を行なうようにデータを受け渡すことができる。

4 応用例 — 時系列データサーバ

MPTRを用いたリアルタイム・システムの開発例として、時系列データサーバRASCALについて説明する。RASCALは、分散型実時間カーネルARTS [Tokuda, Mercer89] 上の分散型実時間プログラミング言語 RTC++ [Ishikawa, Tokuda92] におけるリアルタイム・オブジェクトをベースにして、我々が設計したものである。

時系列データサーバとは、プラントなどから定期的にサンプルされたデータに時刻印を付けて、これを時刻順に列べて保持し、かつ、外部からネットワークを通じて送られる要求によって、現在時刻のデータを周期的に要求元に返したり（周期要求）、指定された期間内のサンプルデータを時系列データとして要求元に返したりする（非周期要求）データサーバである。定期的サンプルはハード・デッドラインを、クライアントへの周期転送はソフト・デッドラインを持つ。クライアントへの非周期転送は特にデッドラインは持たない。

図6はRASCALの全体の構成を示したものである。周期スレッド τ_{p0} （優先度 P_{p0} ）はその周期毎にプラントからのサンプルデータを長さ1の受信キュー（2ポートメモリ）を介して受けとり、これにサンプル時の時刻を時刻印として付加し、メインメモリ上のリングバッファへ格納する。周期スレッド τ_{p1} （優先度 P_{p1} ）は周期スレッド τ_{p0} の周期の n 倍（ n は自然数）の周期毎にリングバッファ上の n 個のデータをディスクへ格納する。

クライアントからネットワークを通じてRASCALに要求メッセージが送られると、レシーバ・スレッド τ_{recv} （優先度 P_{recv} ）がこのメッセージを受信し、受信ポート内のFIFOキューに蓄積する。その後マザー・スレッド τ_{mother} （優先度 P_{mother} ）がそのメッセージを取り出し、その内容に従った処理を行なう。周期要求が送られた場合には、マザースレッド τ_{mother} は、トランスミッタを通じて要求を受けたことを示す確認メッセージを返した後、周期スレッド τ_{p2} （優先度 P_{p2} ）を生成

する。周期スレッド τ_{p2} は一定周期毎にリングバッファ上にある現在時刻の時系列データを読み出し、トランスミッタを介してクライアントに送る。周期要求の停止要求が送られた場合には、マザースレッド τ_{mother} は、周期スレッド τ_{p2} （優先度 P_{p2} ）を消滅させた後、クライアントに確認メッセージを返す。非周期要求が送られた場合には、マザースレッド τ_{mother} は非周期スレッド τ_a （優先度 P_a ）を生成する。非周期スレッド τ_a は要求された時刻のデータがリングバッファ上にあるか、ディスク上にあるか、あるいは両者にまたがって蓄積されているかを判定して読み出し、トランスミッタを介してクライアントに送る。

トランスミッタは各スレッドから送られたメッセージを一旦蓄積し、それらをネットワークを通じてクライアントに送信する。トランスミッタ内のトランスミッタ・スレッドおよび送信用キューは優先度別にそれぞれ3個ずつ用意される。これらのトランスミッタ・スレッドを $\tau_{t0}, \tau_{t1}, \tau_{t2}$ 、優先度を P_{t0}, P_{t1}, P_{t2} （ $P_{t0} > P_{t1} > P_{t2}$ ）とする。優先度 P_{t0} のトランスミッタ・スレッドおよび送信用キューは確認メッセージの転送に、優先度 P_{t1} のそれらは周期要求の転送に、優先度 P_{t2} のそれらは非周期要求の転送にそれぞれ用いられる。

リングバッファに対してアクセスするスレッドは、周期スレッド $\tau_{p0}, \tau_{p1}, \tau_{p2}$ 、および、非周期スレッド τ_a である。これらのスレッド間での相互排除は、リングバッファの各要素に直接ロックをかけず、唯一書き込みを行なう τ_{p0} が、現在書き込みを行なっている要素のインデックスを格納する領域を設け、この領域へのアクセスのみに対してロックをかける、楽観的な方法をとっている。また、周期スレッド τ_{p1} と非周期スレッド τ_a との間での、ディスク上のファイルに対する相互排除も、同様な方法によって行なわれる。

5 評価

以下、RASCALの開発にあたってMPTRを用いることにより得られた利点について述べる。

5.1 スケジューラビリティの判定

RASCALにおける周期スレッドの生成の記述に周期スレッド生成ライブラリを用いた。これにより、システムの設計時に、各スレッド毎にその実行時間を見積もることにより、RMによるスケジューリング可能性判定の手法を用いて、システム全体の

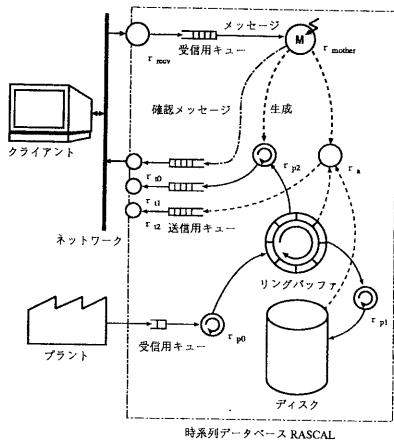


図 6: RASCAL の全体構成

おおまかなユーティライゼーションを見積もることができる。

周期スレッド τ_{p0} , τ_{p1} , τ_{p2} それぞれの実行時間を C_{p0} , C_{p1} , C_{p2} 、周期を T_{p0} , T_{p1} , T_{p2} とする。また、周期スレッド以外に時間上の制約を持つスレッドとして、レシーバ・スレッド τ_{recv} 、マザー・スレッド τ_{mother} 、確認メッセージ送信用のトランスミッタ・スレッド τ_{t0} 、周期転送用のトランスミッタ・スレッド τ_{t1} がある。それぞれの最悪実行時間を C_{recv} , C_{mother} , C_{t0} , C_{t1} とする。

[Cheng87] によると、非周期スレッドを、その起動される最小間隔を周期とする周期スレッドと考えることにより、RM によるスケジューリングが可能となる。 τ_{recv} , τ_{mother} , τ_{t0} の周期は、クライアントからのメッセージの到着の最小間隔と考えることができ、これを T_{msg} とする。この値はクライアント側で決定することができる。また、 τ_{t1} の周期は周期転送の周期に等しく T_{p2} である。

各スレッドの周期には $T_{p0} < T_{p1} < T_{msg} < T_{p2}$ なる関係があると考えて良い。RM スケジューリングでは、より短い周期を持つタスクにより高い優先度を持たせる。ただし、スレッド τ_a , τ_{t2} はデッドライン持たないので、これらの優先度は最低と考える。よって、各スレッド間の優先度の大小関係を、 $P_{p0} > P_{p1} > P_{recv} > P_{mother} > P_{t0} > P_{p2} > P_{t1} > P_a > P_{t2}$ と定める。

[Rajkumar91] によると、互いに独立でない周期スレッドの、実行時間、より低い優先度のタスクによる最大ブロック時間、周期がわかれば RM によるスケジューリング可能性の判定が可能である。 n 個

の周期タスク $\tau_1, \tau_2, \dots, \tau_n$ があり、これらの優先度 $P_1, P_2, \dots, P_n$ の間に、 $P_1 \geq P_2 \geq \dots \geq P_n$ の関係があり、タスク τ_i ($1 \leq i \leq n$) の周期を T_i 、実行時間を C_i 、最大ブロック時間を B_i とする時、

$$\sum_{i=1}^n \frac{C_i}{T_i} + \max_{i=1}^{n-1} \frac{B_i}{T_i} \leq n(2^{1/n} - 1)$$

が成立すれば、RM によるスケジューリングが可能である。

上式にしたがってスケジューリング可能性の判定を行なうためには、各スレッドの最大ブロック時間を求める必要がある。第 4 章で述べたように、リングバッファ、および、ディスク上のファイルに対する相互排除は、これらの共有資源そのものではなく、これらに対して書き込みが行なわれているインデックスを格納する領域にロックをかける樂觀的方法で行なわれている。これらの領域へのアクセスは、数バイトのデータを書き込む、あるいは読み出す処理なので、その実行時間はほぼ一定で、かつ、それらの処理を行なうスレッドの実行時間に比べて、無視して良いほど小さい。これら以外に生じる、各スレッド間の相互排除は、各トランスミッタ・スレッドのネットワークへのアクセス権に対するもののみである。トランスミッタ・スレッド τ_{t0} , τ_{t1} , τ_{t2} からのネットワークへのアクセスは、優先度継承機能 [Rajkumar91] を持ったセマフォによって相互排除される。各トランスミッタ・スレッドの送信するデータの量は常に一定であり、 $C_{t0} = C_{t1} = C_{t2} = C_t$ である。ただし、 C_{t2} はトランスミッタ・スレッド τ_{t2} の実行時間である。各トランスミッタ・スレッドの優先度 P_{t0} , P_{t1} , P_{t2} の間には $P_{t0} > P_{t1} > P_{t2}$ の関係があり、また、相互排除のためのセマフォが優先度継承機能を持つので、 τ_{t0} の最大ブロック時間 B_{t0} 、および、 τ_{t1} の最大ブロック時間 B_{t1} はともに C_t となる。

簡単のため、全てのスレッドが同時に動作するという悲観的な仮定の元での判定を行なう。

$$\begin{aligned} & \frac{C_{p0}}{T_{p0}} + \frac{C_{p1}}{T_{p1}} + \frac{C_{p2}}{T_{p2}} + \frac{C_{recv}}{T_{msg}} + \frac{C_{mother}}{T_{msg}} \\ & + \frac{C_t}{T_{msg}} + \frac{C_t}{T_{p2}} + \max\left(\frac{C_t}{T_{msg}}, \frac{C_t}{T_{p2}}\right) \\ & \leq 7(2^{1/7} - 1) \approx 0.73 \end{aligned}$$

が成立することが、RM によるスケジューリングが可能であるための十分条件であることがわかる。システムの動作をいくつかの状態に分類し、各状態毎に判定することにより、より厳密な判定結果を得られるであろう。

非周期 転送	最大値	最小値	平均値	標準偏差
なし	1010.00	990.00	1000.00	9.66
あり	1000.00	999.00	999.97	0.180

(最大値、最小値、平均値の単位は $msec$)

注1) 周期転送は1秒周期で30回行なった。
注2) 非周期転送は周期転送を30回行なう間に、
1秒間隔で30回行なった。

表1: 周期要求によるメッセージの到着間隔

5.2 ネットワーク通信への効果

RASCALにおけるレシーバをリアルタイム通信ライブラリを用いて実現したことにより、クライアントから送られるメッセージの欠損をなくすることができ、信頼性を確保できた。

次に、トランスミッタをリアルタイム通信ライブラリを用いて実現したことにより、リアルタイム性を確保できたことを示す。表1は、周期転送によってクライアントに送られるメッセージの到着間隔を、非周期転送を受け付けた場合と、受け付けない場合に分けて測定したものである。この結果より、非周期転送によるリアルタイム性を要求される周期転送への影響はないことがわかる。

5.3 スレッドのプール化による効果

リアルタイム性を保証するために、スレッドの生成、消滅に伴うガーベジのまとめ回収は避けられなくてはならない。RASCALでは、クライアントからの周期要求、非周期要求に応じてスレッドを生成する部分でスレッドのプール化手法を用いることにより、ガーベジのまとめ回収を避けている。

6 今後の課題

RASCALでは、 T_{recv} , T_{mother} , T_{t0} のような、デッドラインを持つ非周期スレッドに対して、クライアントからのメッセージの到着の最小間隔をもって、これらのスレッドの周期、およびデッドラインとみなしている。しかし、複数のクライアントからの要求を受け付ける場合には、メッセージの到着の最小間隔は見積もることができない。

さらに、マザースレッドは、さまざまな処理を同一の優先度で行なっているが、理想的にはそれに応じた優先度、デッドラインで実行すべきである。例えば、マザースレッドによる非周期要求によるスレッド生成が、既に生成されている周期スレッド τ_{p2} の実行を妨げるべきではない。

非周期タスクのデッドラインがその最小起動間隔と等しいか、あるいは、より大きい場合には Sporadic サーバ [Sprunt89] を用いることにより、RM によるスケジューリングが可能となる。RASCAL に、Sporadic サーバのような機能を持たせることにより、デッドラインを持つ非周期スレッドの適切なスケジューリングを行なうことができるであろう。

7 おわりに

本論文では、従来のリアルタイム・システム構築における問題点を解決した、新しい構築技術を提供するために我々が開発した、リアルタイム・マルチスレッド・プログラミング支援ツール (MPTR) について述べた。MPTR の副次的効果として、システム開発の初期段階でのシステムの全体ユーティライゼーションの見積りを容易にすることがあげられる。さらに、本論文では MPTR を用いて開発したリアルタイム・システムの例として時系列データサーバ RASCAL をあげ、その作成にあたって MPTR を利用することによって得られた利点について述べた。

参考文献

- [Liu,Layland73] C.L.Liu and James W.Layland, "Scheduling Algorithm for Multiprogramming in a Hard-Real-Time Environment", JACM, Vol.20 No.1, Jan. 1973
- [Cheng87] S.Cheng, J.A. Stankovic, and K. Ramamrithan, "Scheduling Algorithms for Hard Real-Time System — A Brief Survey", J.A. Stankovic and K. Ramamrithan(ed.) Tutorial Hard Real-Time Systems, IEEE Computer Society Press, 1988
- [Tokuda, Mercer89] H.Tokuda and C.W.Mercer, "ARTS : A Distributed Read-Time Kernel", Operating Systems Review Vol.23 No.3, ACM Press, Jul. 1989
- [Sprunt89] B.Sprunt, L.Sha and J.Lehoczky, "Aperiodic Task Scheduling for Hard-Real-Time Systems", The Journal of Real-Time Systems Vol.1, 1989
- [Tokuda91] 徳田 英幸, 「チュートリアル 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, 1991
- [Rajkumar91] R. Rajkumar, "Synchronization In Real-Time Systems, A Priority Inheritance Approach", Kluwer Academic Publishers, 1991
- [Ishikawa, Tokuda92] 石川 裕, 徳田 英幸, 「分散型実時間プログラミング言語 RTC++」, コンピュータソフトウェア Vol.9 No.2, 1992