オペレーティング・システム 49－2
（１９９０．　１２．　７）

小型計算機用分散オペレーティングシステムについて

白川 洋充　藤原 正之　林 恒俊　大野 豊

立命館大学　理工学部

分散オペレーティングシステム Theta (Threads and Tasks) は単一バスやハイパーキューブのような構造を有する分散メモリ型の並列計算機をサポートする目的で開発されている．核化カーネルをさらに発展させた極小カーネルを実現しており，高速タスク間通信・柔軟な制御構造などの特徴を持っている．さらに，システムの記述にオブジェクト指向の技法を採り入れオペレーティングシステムの抽象化を行った．この抽象化により，オペレーティングシステムの構造と機能の階層化と統合化をより進めることができた．

本論文ではThetaの設計思想を述べると共に，Thetaの構造と機能について説明するものである．

A Distributed Operating System for Small Computers

Hiromitsu Shirakawa　Masayuki Fujiwara　Tsunetoshi Hayashi　Yutaka Ohno

Ritsumeikan University　Faculty of Science and Engineering

A distributed operating system Theta was developed to support a distributed memory type parallel processors. The kernel of Theta is minimal in a sense that only remaining function is a dispatcher. Other facilities within the kernel of the conventional operating system are implemented by system servers in Theta. Other features of the Theta are fast intertask communication and flexible control mechanism in exception handling. Design of Theta is based on object-oriented analysis of conventional operating system and object-oriented synthesis of operating system. This leads to the abstraction of operating system.

This paper describes the design philosophy of Theta, and the structure and functions of the Theta.

1. はじめに

研究上，あるいは実際のアプリケーションのために並列・分散処理を適用する需要がますます高まっている．並列・分散処理を指向した計算機のオペレーティングシステムにＵＮＩＸを採用することは，効率の悪さ，カーネルの肥大など問題が多く，この問題を解決するためにＭａｃｈ[1]が開発された．この考えに沿って種々のオペレーティングシステムが開発された[2)3)4)5)]．これらのオペレーティングシステムは単一のＣＰＵあるいは複数のＣＰＵをサポートするワークステーションを対象としたものと言える．

しかしながら，単一バスやハイパーキューブのような構造を有する分散メモリ型の並列計算機は，それぞれの計算機が局所メモリ，メモリ管理を持たない高機能ＣＰＵと計算機間の通信を行う通信専用プロセッサだけから構成される小型計算機である．このようなシステムにはワークステーションを対象としたＭａｃｈ流のオペレーティングシステムは適当でない．Theta (Threads and Tasks) は分散メモリ型の並列計算機をサポートする目的で開発されたものである．

Thetaは次のような特徴を持つオペレーティングシステムである．

- ・極小カーネル
- ・スレッドを用いた軽量プロセス
- ・高速タスク間通信
- ・柔軟な制御構造
- ・ＵＮＩＸとの親和性
- ・ＲＯＭ化

また，Thetaではオブジェクト指向の技法[6)7)]を採り入れオペレーティングシステムの抽象化を行った．この抽象化により，オペレーティングシステムの構造と機能の階層化と統合化を進めることができ，これを設計に反映させることができた．

本論文ではThetaの設計思想を述べると共に，Thetaの構造と機能について説明するものである．

2. 設計思想

Thetaは並列計算機に使用する目的で並行性をより高めることを最重点課題とし，核化カーネルの概念をさらに進めて極小カーネルを実現した．図１に示すように，カーネルの機能をコンテキストの切り替えのみとして，システムの他の機能をカーネルの外に出し，これらをシステムサーバとして実現した．このことによってオペレーティングシステム自体の並行動作がさらに推進され，システム全体の並行性を高めることができる．

システムサーバは，メモリやI/Oデバイス等のシステムのハードウェア資源の管理，タスクやスレッドの生成・消去，スレッドの実行状態の遷移の管理等を行う．スケジューリングサーバはレディキューの管理を行っており，特殊なスレッドとして実現されている．また，図１に示されるように，スケジューリングサーバ以外の全てのスレッドは，カーネルから平等に扱われる．

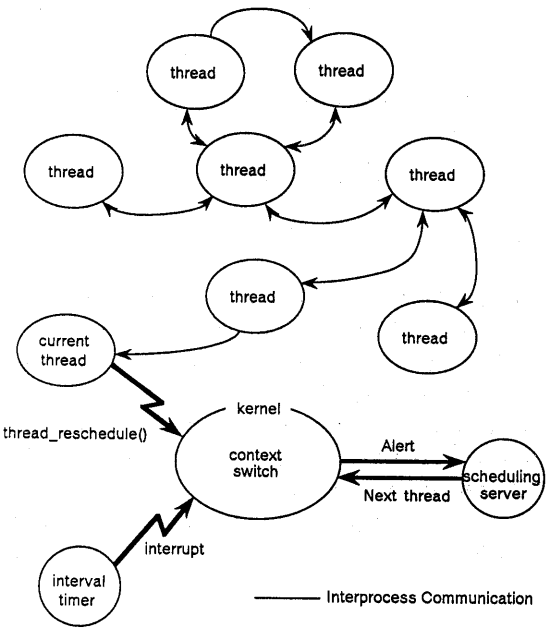

図１　カーネルとシステムサーバ

カーネルの機能は，コンテキスト切り替えのみである．したがって，システムコールは実行権の放棄ただ一つである．タスク間通信やスレッドの実行状態の遷移等は，ライブラリ関数で実現しており，これらの処理を高速に行うことができる．また，割り込み禁止区間をコンテキスト切り替え時のみに限定することにより，割り込み応答性を良くしている．

3. システムの構成

3.1 タスクとスレッド

タスクはThetaにおける資源管理の単位であり，その管理はタスクディスクリプタにより行っている．タスクディスクリプタの内容の主なものは，タスクに属するスレッドへのリンク，タスク間通信に使用されるポートである．一つの処理を複数のスレッドによって行う場合，それらのスレッドはタスクによりポートなどの資源と共にひとまとめにして管理される．

これに対して，スレッドはスケジューリングの単位であり，例外処理等もスレッドをもとに行っている．スレッドはスレッドディスクリプタを持っている．スレッドディスクリプタとはレディキューや条件変数のキューなどにつなぐためのリンク，プロセッサの内部状態の退避領域とスタック，実行状態を表すフラグなどからなるデータ構造である．プロセスに比べ，スレッドはコンテキスト切り替えのときに退避させるべき状態の数を少なくすることが可能であり，いわゆる軽量プロセス(lightweight process)を実現できる．

Thetaではタスクとスレッドの操作に関するライブラリ関数として以下のようなものを用意している．

task_create	タスクの生成
task_exit	タスクの終了
task_self	タスクの取得
task_join	タスクの終了待ち
task_detach	タスクの切り離し
thread_fork	スレッドの生成
thread_exit	スレッドの終了
thread_self	スレッドの取得
thread_name	スレッド名の設定
thread_priority	スレッドの優先順位の設定
thread_join	スレッドの終了待ち
thread_detach	スレッドの切り離し
thread_kill	スレッドの強制終了
thread_reschedule	スレッドの実行権放棄

3.2 システムサーバ

Thetaでは，システムの各種の機能を実現するためにクライアント／サーバモデルを用いている．

Thetaのシステムサーバにはメモリサーバ・タスクサーバ・ポートサーバ・ネームサーバ・I/Oサーバ・スケジューリングサーバの六個がある．

(1) メモリサーバ

メモリサーバはシステムの全てのメモリを管理しており，メモリの割り付け，解放を処理する．

(2) タスクサーバ

タスクとスレッドの生成・終了を処理する．また，スレッドの生成の高速化のためにスレッドのpre-allocationを行っている．システムの起動時に，ある定められた数のスレッドのディスクリプタをメモリサーバから割り付けてもらいフリーキューにつなぐ．スレッドの生成要求が発生するとフリーキューからディスクリプタを割り付け，スレッドを生成する．スレッドの終了要求が発生したときは，そのスレッドのディスクリプタをフリーキューにつなぐ．タスクサーバがメモリサーバにメモリの割り付けを要求するのは，システムの起動時とフリーキューが空になったときだけである．これにより，スレッドの生成時のオーバーヘッドを小さくできる．

(3) ポートサーバ

ポートの生成・消去などを処理する．ポ

ートに関してもスレッドと同様にpre-allocationを行って高速化を図っている．また，ポートが消去されたときにポート内に残っているメッセージや，メッセージの受信で封鎖状態にあるスレッドの処理もポートサーバが行う．

(4) ネームサーバ

ポートのIDのテーブルへの登録を処理する．現在テーブルに登録されているのは，システムサーバのみである．

(5) I/Oサーバ

I/Oサーバはシステムの入出力デバイスの管理を行っており，それぞれのI/Oデバイスに対して入力と出力に対応した二つずつのスレッドで構成されている．

I/Oサーバとクライアントとの間の通信はメッセージパッシングではなく共有メモリを用いている．これは，メッセージパッシングに伴うコンテキスト切り替えのオーバーヘッドを軽減するためである．

(6) スケジューリングサーバ

スケジューリングサーバはレディキューを管理する特殊なスレッドとして実現されている．Thetaのレディキューは，異なる優先順位を持つ複数のスレッドのキューにより構成されている．スケジューリングサーバはカーネルによって起動され，レディキューの中から次に実行権を得るスレッドを選びカーネルに指示する．カーネルはその指示に従いコンテキストの切り替えを行う．

3.3 ＲＯＭ化について

Thetaは分散メモリ型の並列計算機の各ノードで動作させるためＲＯＭ化することを前提に開発されている．

4. タスク間通信

Thetaのタスク間通信は，メッセージパッシングにもとづいている．Mach等のオペレーティングシステムではメッセージパッシングの機構はシステムコールとしてカーネルが行っているが，Thetaではライブラリ関数として実装している．

原則的に，タスク間の通信はメッセージパッシングで行い，同一タスク内の通信は共有メモリを介して行う．しかし，I/Oサーバとクライアント間のデータのやりとりだけは，task間にもかかわらず高速化のために共有メモリを使用している．

4.1 ポート

ポートはメッセージのキューと，メッセージ待ちのスレッドのキューと，その他の管理情報からなるデータ構造である．

メッセージの送信は，図２のように相手のタスクにではなくポートに対して行われる．受信においても同様にポートに対して操作を行う．スレッドの生成時に一つのポートが作られ，その後ユーザの必要に応じて任意の数のポートを作ることができる．各スレッドが生成された時に作られたポートはthread_port()によって得ることができる．

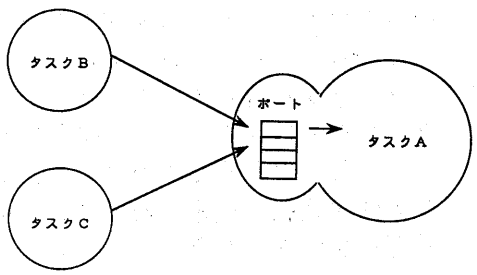

図２． タスク間通信

4.2 非同期型の通信とリモートプロシジャコール

メッセージの送信はmsg_send()によって行われ，受信者がいなくても送信者は封鎖されない．送信されたメッセージはポートの中のメッセージキューにつながれる．メッセージの受信はmsg_receive()によって行われ，ポートにメッセージが無いときには受信者のスレッドは封鎖され，ポートの中

のスレッドキューにつながれる．この封鎖されたスレッドは，そのポートに他のスレッドがメッセージを送信する際に起こされ，メッセージを受け取って実行を再開する．

メッセージパッシングにおいて同期をとる必要がある場合には，msg_rpc()によってリモートプロシジャコールを行う．msg_rpc()は，クライアントがサーバに対してサービス要求を送信し，その後サーバからの返信を待つ．サーバはクライアントからの要求をmsg_receive()で受け取って処理を行いmsg_reply()で結果をクライアントへ返す．

Thetaではメッセージパッシングに関するライブラリ関数として以下のようなものを用意している．

msg_send	メッセージの送信
msg_receive	メッセージの受信
msg_rpc	リモートプロシジャコール
msg_reply	リモートプロシジャコールに対する返信
task_port	タスクのポートの取得
thread_port	スレッドのポートの取得

5. 制御機構

Thetaでは高速な同期と例外処理の機構を用意することによって，並列プログラミングにおける柔軟な実行の制御を可能にしている．

5.1 同期

Thetaでは，同期をmutex[8]と条件変数(condition variable)の組合せによって行う．各スレッドは，まずmutexを用いて相互排除を行った後，条件変数を用いて他のスレッドと同期をとる．

mutexはThetaにおける唯一の排他制御の機構である．mutexのロックはmutex_lock()を用いて行う．mutex_lock()は，mutexのロックに成功した場合は直ちにリターンする．もしmutexのロックに失敗した場合は，ロックに成功するまでmutexのロックの試行と実行権の放棄を繰り返す．

Thetaではmutexの操作に関するライブラリ関数として以下のようなものを用意している．

mutex_lock	mutex のロック
mutex_unlock	mutex のアンロック

並列プログラミングにおいて，同期をとることが必要な局面が非常に多く現れる．しかし，同期をとることはコンテキスト切り替えを伴うので，オーバヘッドが生じる．このオーバヘッドをできるかぎり小さくするために，Thetaでは条件変数を採用している．

条件変数の操作に関するライブラリ関数として以下のようなものを用意している．

・condition_wait
現在実行中のスレッドを条件変数のキューにつなぎmutexをアンロックする．

・condition_signal
条件変数のキューの先頭のスレッドを起こす．

・condition_broadcast
条件変数のキューにつながれている全てのスレッドを起こす．

5.2 例外処理

Thetaは，例外処理のための機構としてalert[9]とsignalの二つを用意している．しかし，Thetaでは主にalertを用いて柔軟な制御構造を実現しており，signalの用途はデバッグを支援するためのみに限定されている．

alertは，alert()やalert_broadcast()によって送られ，受け手はalert_test()またはalert_wait()によって明示的にalertを受け取る．alert に関するライブラリ関数として以下のものを用意している．

・alert
スレッドに対しalertを送る．

・alert_broadcast
条件変数のキューにつながれている全てのスレッドにalertを送る.
・alert_test
alertが到着しているかどうかチェックし，到着していればalertのタイプを，そうでなければ0を返すテストアンドセット命令である.
・alert_wait
条件変数に対して用いられ，現行スレッドを条件変数のキューにつなぐ.condition_signal(), condition_broadcast()によって起こされた場合は0を，alert()またはalert_broadcast()によって起こされた場合はalertのタイプを返す.

signalは特殊なalertとして実現されており，スレッドに対して強制的に例外処理を起こさせる.signalには，スレッドを終了させるkillと，スレッドを停止させるsuspendの二種類のみがある.

6. オペレーティングシステムの抽象化

オペレーティングシステムの内部構造は極めて複雑であり，扱うデータの種類は非常に多い.そして，このことがオペレーティングシステム自体を複雑で巨大なものにしている一因でもある.

そこで，オペレーティングシステム内部のデータと処理の抽象化によって，システム全体の統一的な記述を行い，記述性・生産性の向上を図った.

6.1 データと処理の抽象化

オペレーティングシステム内部のデータを，処理の対象となる一般的なデータであるオブジェクトと，このオブジェクトを管理するためのデータであるアレイの二種類に大別する.そして，このオブジェクトとアレイを抽象化し，それぞれ基本クラスbase_object, object_arrayとして定義する.

そして，この二つの基本クラスを用いてシステム全体を記述する.オペレーティングシステムの記述はこれら二種類の基本クラスで十分である.

基本クラスbase_objectの定義を図3に示す.

```
class base_object {
friend class object_array;
        base_object* next;
        long id;
public:
        base_object();
        ~base_object();
        obj_type get_type();
}
```

図3　base_objectの定義

base_objectはアレイにつなぐためのポインタnextと，オブジェクト識別子idを持っている.また，メンバ関数としてコンストラクタbase_object()，デストラクタ~base_object()，型の取得のための関数get_type()の三つを持っている.

次に，オペレーティングシステムの処理について抽象化を行う.オペレーティングシステムの内部の動作は，オブジェクトの管理と生成・消去がその大半を占めている.そこで，処理を以下の六つに分類し抽象化を行った.

(1) オブジェクトの生成
(2) オブジェクトの消去
(3) アレイの末尾へのオブジェクトの追加
(4) アレイの先頭からのオブジェクトの取り出し
(5) アレイからの指定したオブジェクトの取り出し
(6) オブジェクトの型の取得

(1)のオブジェクトの生成は，新しいオブジェクトのためのメモリなどの資源の割り付け，オブジェクトの初期値の設定などを行う.(2)のオブジェクトの解放は，オ

ブジェクトが使っていた資源を解放し，各種の後処理を行う．

(3)は各種の管理データ構造へのオブジェクトの追加である．ポートへのメッセージの送信，スレッドのレディキューへのキューイングなどがこの処理に含まれる．(4)は管理データ構造からのオブジェクトの取り出しである．ポートからのメッセージの受信，レディキューからのスレッドの取り出しなどがこの処理に含まれる．

(5)は管理データ構造からの指定したオブジェクトの取り出しである．そして，(6)はそれぞれのオブジェクトの型の取得である．

そして，この処理の抽象化に基づき基本クラスobject_arrayを図4のように定義する．

```
class object_array : public base_object {
        base_object* head;
        base_object* tail;
public:
        object_array();
        ~object_array();
        void put_tail(base_object*);
        base_object* take_head();
        base_object* take(base_object*);
        bool is_empty();
}
```

図4　object_array の定義

object_arrayはアレイの先頭と末尾のオブジェクトへのポインタheadとtailを持っている．そして，メンバ関数としてコンストラクタobject_array()，デストラクタ~object_array()，アレイの処理put_tail()，take_head()，take()，アレイの状態のチェックis_empty() の六個を持っている．

6.2　クラスの階層構造

基本クラスから各種のデータ構造に対応したクラスを導出することによって処理を統一することができプログラムの記述性・信頼性は格段に向上する．基本クラスbase_objectとobject_arrayから他のクラスを図5のように導出する．

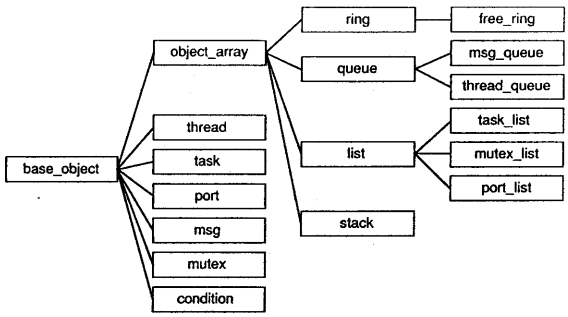

図5　クラスの階層構造

base_objectからobject_array，thread，task，port，msg，mutex，conditionなどのクラスを導出する．導出するときに基本クラスであるbase_objectのメンバはそのまま受け継ぎ，それに各クラスで固有のメンバを追加する．例えばmsgならメッセージパッシングのための管理情報と処理関数がメンバとして追加されている．ここで，object_arrayをbase_objectから導出しているのは，アレイが他のオブジェクトを管理するためのデータであるのと同時に，アレイ自身も管理の対象となっていることを示している．

次に，object_arrayからring，queue，list，stackの四つのクラスを導出する．object_arrayはオブジェクトが何らかの形で並んだものであり処理には特に制限はない．しかし，導出された四つのクラスはそれぞれ特徴を持っており，オブジェクトの並び方と処理が異なる．ringはオブジェクトがリング状に並んだものであり，基本操作に加えてリングを回転させる処理(rotate)を持っている．queueはFIFOのデータ構造であり末尾への追加(enqueue)と先頭からの取り出し(dequeue)のみが許されている．listはオブジェクトの並び方が特に意味を持たないデータ構造で，追加 (put) とオブジェクトを指定した取り出しのみである．stackはLIFOのデータ構造で，末尾への追加

(push)と末尾からの取り出し(pop)のみが許されている.

表1に各処理の導出関係を示す.

object_array		ring	queue	list	stack
put_tail	→	put_tail	enqueue	put	push
take_head	→	take_head	dequeue	—	—
take	→	take	—	take	—
—		rotate	—	—	—
—		—	—	—	pop

表1　処理の導出関係

基本クラスobject_arrayにおいて定義されている処理を，導出された各クラスで使用する場合は，基本クラスで記述されたプログラムをそのまま使っている．このとき，各処理の意味をより明確にするために関数名を変えて受け継ぐ場合がある．また，基本クラスで定義されていない処理については新しく記述している．

そして，これらのクラスからシステムで実際に使用するクラスを導出している．ringからは空きメモリ管理用のリングを導出している．queueからはポートのためのメッセージのキューと，レディキューを構成するためのスレッドのキューを導出している．また，listからは資源管理用の種々のリストを導出している．

7. おわりに

Thetaは分散メモリ型の並列計算機への応用を指向した分散オペレーティングシステムである．現在はシングルプロセッサのみで開発されている．並列計算機のプログラミング言語としては，分散型の関数型言語を使用する予定であり，現在本分散オペレーティングシステム上へ移植し，言語処理系とオペレーティングシステムの評価を行っている．

8. 参考文献

1) Accetta, M. et al.: Mach: A New Kernel Foundation for UNIX Development, Proc. of the Summer 1986 USENIX Conf. pp. 93-112 (1986).
2) McJones, P. and Swart, G.: Evolving the UNIX System Interface to Support Multi-Threaded Programs, Proc. of the Winter 1989 USENIX Conf. (1989).
3) Mullender, S.J., van Rossum, G., Tanenbaum, A.S., van Renesse, R. and van Staveren, H.: Amoeba: A Distributed Operating System for the 1990s, IEEE Computer, vol. 23, no. 5, pp. 44-53, May (1990).
4) 新井，桜井，立木，萩野，服部，森島: 分散オペレーティング・システムＴｏＭの概要. JUS 14th UNIX Symposium Proceedings (1989).
5) 京谷，竹岡，西垣内 : 実時間指向のオペレーティング・システムCinnamonの概要. 情報処理学会第39回大会(1989).
6) Russo, V., Johnston, G. and Campbell, R.: Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques, OOPSLA'88 Proceedings, pp.248-258, September (1988).
7) Stroustrup, B.: The C++ Programming Language, Addison-Wesley, (1986).
8) Birrell, A., Guttag, J., Horning, J. and Levin, R.: Synchronization Primitives for Multiprocessor: Formal Specification, Proc. of the 11th Symposium on Operating Systems Principles, pp. 94-102 (1987).
9) Birrell, A.: An Introduction to Programming with Threads, Research Report 35. Digital Equipment Corporation Systems Research Center (1989).