オペレーティング・システム 45－6
（1989. 11. 11）

可変構造型並列計算機の並列／分散オペレーティング・システム
－ スレッドの実現 －

福澤祐二　草野和寛　恒冨邦彦　福田晃　村上和彰　富田眞治
（九州大学大学院総合理工学研究科）

「可変構造型並列計算機」はメモリ／ネットワーク・アーキテクチャが可変なマルチプロセッサ・システムであり，解くべき問題に適合したハードウェア構成が可能である．オペレーティング・システム（OS）は，ハードウェア・アーキテクチャと多様な並列処理問題との親和性の研究，ならびに解くべき並列処理問題にオペレーティング・システムを動的に適合させることによって，マルチプロセッサ・システムの応用分野の拡大を図ることを目的としている．開発中のOSは，中粒度の並列処理を効率よく並列実行するために，1つのアドレス空間内に複数の制御フロー（スレッド）を提供している．スケジューリングは，分散制御－動的負荷分散－システム空間分割方式をとる．また，2つのプログラミング・スタイルでスレッドのプログラミングを行えるように，スレッド生成のシステム・コールを2種類用意している．これらのシステム・コールはスレッドの生成とその適用方法が異なっており，本論文では，各々の特徴と問題点について取り上げ，OSの開発状況を述べる．

A Parallel／Distributed Operating System for The Reconfigurable Parallel Processor
－ Implementation of Thread －

Yuji FUKUZAWA, Kazuhiro KUSANO, Kunihiko TSUNEDOMI, Akira FUKUDA,
Kazuaki MURAKAMI, and Shinji TOMITA
Interdisciplinary Graduate School of Engineering Sciences, Kyushu University,
6－1 Kasuga－Koen, Kasuga－shi, Fukuoka, 816 JAPAN
fukuzawa@kyu－is. is. kyushu－u. ac. jp

A reconfigurable parallel processor under development at Kyushu University is a MIMD－type multiprosessor, which employs reconfigurable network／memory architectures. The system can take an architectue in conformity with algorithms. The research object of this operating system is to study on an affinity between hardware architectures and parallel algorithms in applications, and to execute efficiently a broad variety of applications on the systems. The operating system under development provides users with control flows (threads) in an address space to execute efficiently medium grained parallel processings. As scheduling method, it takes a decentralized control, load balancing, and system space partitioning methods. And, two kinds of system calls of threads creation have been provided. The differences between these system calls are in programming styles and threads creation mechanisms. This paper describes the features and problems of these system calls, and the current status of the operating system under development.

1. はじめに

我々が開発中の可変構造型並列計算機は，128台のプロセッシング・エレメント（PE）を128×128のクロスバー網で相互接続したマルチプロセッサ・システムである[1][2]．相互結合網およびメモリ構成にダイナミック・アーキテクチャを採用しており，システムは図1のようなメモリ空間を持つ．これによって，本システムは密結合／疎結合型マルチプロセッサのいずれをもハードウェア・レベルで実現できる．並列処理システムの総合的な評価を目的とする．本稿では，主に現在開発中の密結合型メモリ・モデルを基本としたオペレーティング・システム（OS）[3][4] のスレッドの実現方法について述べる．

2. オペレーティング・システム設計方針

本システム開発の主な目的を以下に示す．

（1）ハードウェア・アーキテクチャと各種並列処理問題との親和性の検討．

（2）解くべき並列処理問題にOSを動的に適合させることによるマルチプロセッサ・システムの応用分野の拡大．

（1），（2）を遂行するには，並列／分散OSの研究を含めた広範囲な並列処理の研究が不可欠である．（2）を実現するためには，まず（1）の検討すなわち，ハードウェア・アーキテクチャと各種並列処理問題との親和性を定量的に検討する必要があると考える．そこで，我々は（2）を実現するOSの開発に先立って，（1）を対象としたOSを開発する．

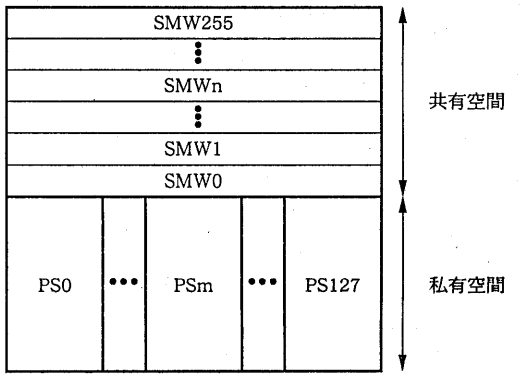
プログラミングをするユーザにとってシステム・コールが計算機との接点であり，システム・コールの機能がユーザからみた計算機のイメージを決定する．並列処理では，その粒度が並列処理モデルを決定する．マルチプロセッサ・システムでは並列処理の粒度が小さい場合，プロセッサ間の通信が頻繁になり，スループットの向上が期待できないと予想される．逆に，並列処理の粒度が大き過ぎると，並列性を引き出せない．また，応用分野によっては，その並列処理の粒度が適合しないといった問題が生じる．以下に我々が対象とする並列処理モデルに関して述べる．

2.1 並列処理モデル

従来シングルプロセッサ・システムでは，ユーザ・プログラムの並列実行可能な部分でも逐次的に実行するのみであった．ハードウェアが並列処理に対応していないだけでなく，OSも並列処理を考慮した設計がなされていなかった．このため，マルチプロセッサ・システムに適したOSの研究が必要である．我々の設計では，マルチプロセッサ・システム上でユーザ・プログラムを中粒度の並列処理に分割し，並列実行可能とする．また，ソフトウェア資産の継承という観点から，一般的なOSであるUNIXに拡張を行う．従って，ユーザのプロセス内部を並列実行可能な処理－スレッド－に分割して実行する．同一プロセス内のスレッドは，その空間を完全に共有する．プロセスは1つの空間とその空間に属する資源および複数のスレッドからなっており，スレッドはプログラム・カウンタ，レジスタ，スタック領域および実行に関する情報からなる．

2.2 スケジューリング

マルチプロセッサ・システムにおける並列処理では，スケジューリングは最も重要な問題の一つである．このため，様々なスケジューリング方式を検討している．負荷の分散を考える場合，スレッド単位，プロセス単位あるいはプロセス・グループ単位で分散を行うことが考えられる．その方式として，静的負荷分散と動的負荷分散があり，また，制御方式の観点から，集中制御方式と



PSm ：私有空間
SMWn ：共有メモリ・ウィンドウ

図1　実メモリ空間

プロセッサ単位の分散制御方式がある．さらに，時間の概念を取り込むと図2のようにシステム空間分割方式（図2（a））と時分割方式（図2（b））に大別できる．システム空間分割方式では，プロセスは実行時に割り当てられたプロセッサ上で，終了するまで実行を続ける．時分割方式では一定時間（タイム・スライス）でプロセスの実行を割り当てる．実際には，プロセッサに負荷を均等に割り当てるようにすべきであり，キャッシュ・メモリや二次記憶装置といったハードウェア資源の問題により，スレッドのプロセッサ間移動を避けるか，積極的に行うかを選択する．多くのシステムは，システム空間分割方式と時分割方式を組み合わせた派生型でスループットを向上させている．現在我々が設計を行っているスケジューリング方式は，分散制御－動的負荷分散－システム空間分割方式である[5]．その他にも多くのスケジューリング方式を実現することによって，スケジューリング方式の評価を行う．

3．スレッドの実現と問題点

3.1 基本的な実行形態

スレッドとは，1つのアドレス空間を共有する仮想的なプロセッサである．図3にスレッドの最も簡単な使用例を示す．プロセスが生成された時点ではスレッドは1つであり，初期化等を行なった後，このスレッドが，並列実行を必要とし，その実行が可能な部分に到達すると，スレッド生成のシステム・コール（thread__lfork（n）またはthread__sfork（n，func，arg1，・・・））を実行することによって複数（n個）のスレッドを生成する．それぞれのスレッドは互いに協調しながら処理を行ない，結果の集計のような並列実行が不要な部分まで実行が到達すると，スレッドを1つに収束させるシステム・コール（thread__join（ ））を実行する．スレッドidが最も小さなスレッドを除く全てのスレッドは，このシステム・コールを実行した後に消滅し，プロセス内のスレッドは1つになる．プロセスの実行が終了するとともに，このスレッドもそのスタック領域と共に消滅する．

3.2 プログラミング・スタイル

スレッドを用いたプログラミング・スタイルとして以下の2つを提供する．

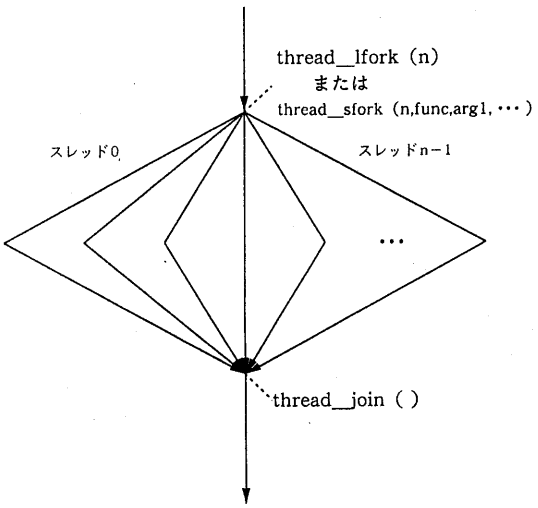
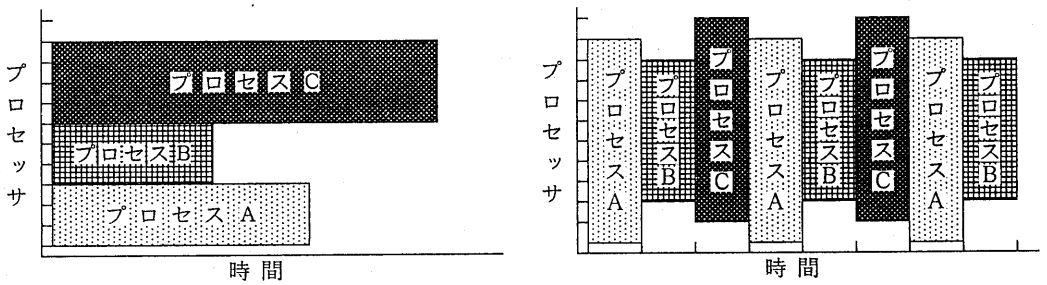


図3 スレッドの使用例



（a）システム空間分割方式　　（b）時分割方式

図2 時間と空間とに着眼したスケジューリングの基本形態

（1）スタイル①（図4（a））

プロセスが生成された時点からブロックAを1つのスレッドで実行する．スレッド生成のシステム・コールを実行し，ブロックBでは複数のスレッドになり，このブロックは並列に実行される．ブロックCでは再び1つのスレッドになり，プロセスの実行を終了する．

（2）スタイル②（図4（b））

スタイル①と同様に，プロセスが生成された時点では，1つのスレッドによって実行される．スレッド生成のシステム・コールによって複数のスレッドが生成されるが，スタイル①との相違点はこのシステム・コールにあり，引数は生成すべきスレッドの数とスレッドによって並列実行される関数のアドレスおよびその関数への引数である．このシステム・コールによってスレッド本体となる関数（ブロックB）を並列に実行し，ブロックCでは1つのスレッドとなりプロセスを終了する．

3.3 スレッドの実現方法

以下にプログラミング・スタイル①，②の実現方法の概要を述べる．

（1）スタイル①

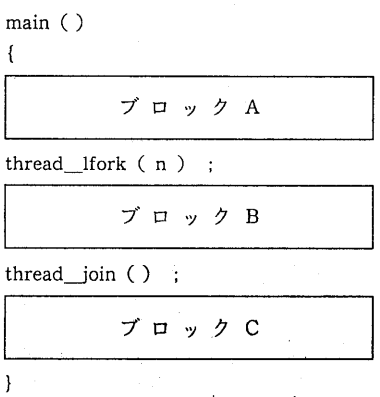
スレッド生成のシステム・コールの発行によって，カーネルはこのシステム・コールを発行したスレッドとスタックおよびスタック・ポインタを除くレジスタ群の内容が同じスレッドを生成する．スタック領域は，同一空間内で重複しないアドレスが割り当てられる．従って，新たにスレッドをn－1個生成する場合，スタックとレジスタのコピー操作がn－1回行なわれる（図5（a））．

（2）スタイル②

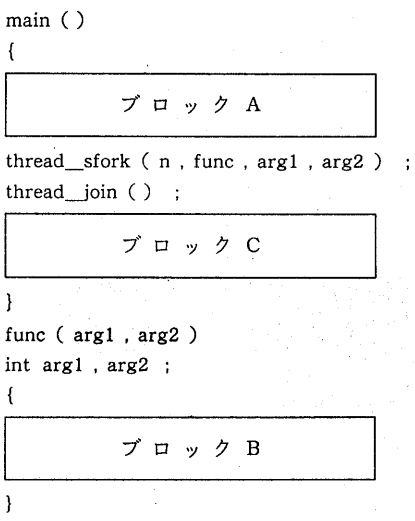
スレッド生成のシステム・コールを発行する際に，引数にスレッドによって並列実行すべき関数のアドレスとその関数に渡す引数を列挙する．関数の引数はスタック内にあり，カーネルはこのシステム・コールの引数の内で，目的の関数に渡す引数だけを同一空間内で重複しないアドレスに新たに割り当てられたスタック領域にコピーし，スタック・ポインタがこの領域を指し示すように書き換える（図5（b））．

3.4 スレッド生成方法の比較

これら2つのスレッドの生成に関する最も大きな違いは，そのプログラミング・スタイルにある．スタイル①は図4（a）に示すように，ブロックA，B，C間で共通の自動変数が使用できる．従って，プログラマの好みによるところが大きいが，スタイル①のスレッドは，スタイル②のスレッドの場合と比べて，プログラム全体にわたって見通しが良いと考えられる．既存のプログラムの移植性についても，スタイル①はループの部分を少し手直しするだけでよいが，スタイル②ではループの部分を関数として抽出しなければならないためスタイル①に比べて手間がかかると思われる．



（a）スタイル①



（b）スタイル②

図4　プログラミング・スタイル

更に，スタイル①のスレッドでは，ブロックA，B，C間でコンパイラによる最適化が期待できる．これに比べて，スタイル②ではブロックA，C間での最適化しか望めない．

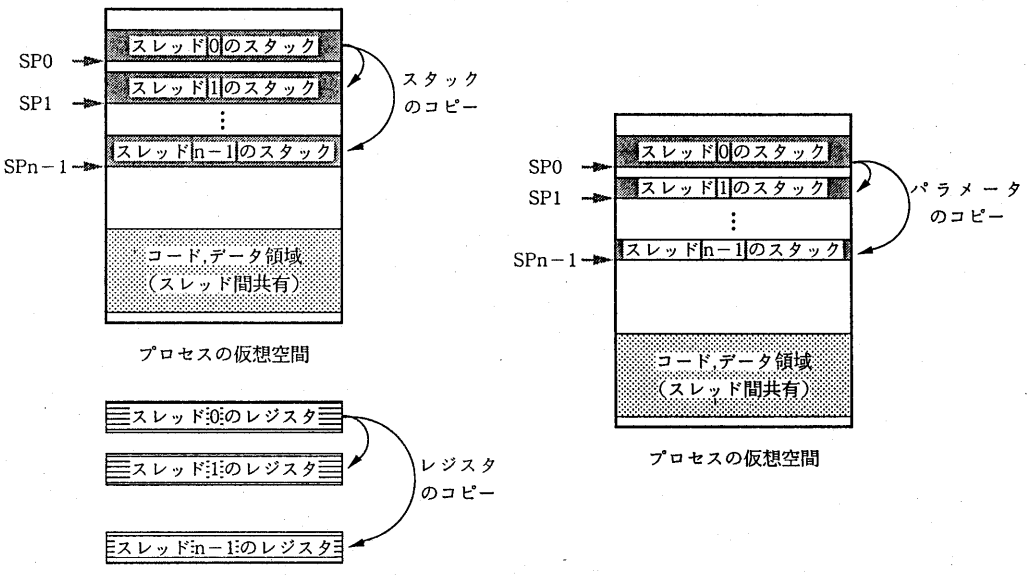
また，先に述べたようにスタイル①はスタック領域をすべてコピーしなければならないが，スタイル②はパラメータのコピーだけでよい．従って，スタイル②はスタイル①に比べて，スレッドの生成が軽いという特徴がある．更に，スタイル①では次に述べるようなスコープ規則の乱れが生じる．

3.5 プログラミング上の問題点

スタイル①の場合のプログラミング上の問題点として，右のようなc言語によるプログラムを考える．

このプログラムで問題となるのは，スレッド生成時にスタック領域をコピーするため，ユーザは全ての自動変数がスレッド間で独立しているような印象を受けるが，③で複数のスレッドが生成されて，スタックをコピーしたときに，②でポインタ型の変数であるpに自動変数の配列dのアドレスを渡す．p自体はスレッド間で共有してはいないが，その内容はどのスレッドでも一様に，最初のスレッドの中のスタックの配列dの先頭を指している．このため⑤でpの示すアドレスに書き込みを行った場合，全てのスレッドが同じアドレスのデータに書き込むことになる．従って，⑤以降で＊pを参照した場合，その内容は不定となる．従って，ユーザはこれを意識したプログラムを書く必要がある．

```
main ( )
{
        int     i , *p , d [ 100 ] , thread__id ;
        ...
        i = 1 ;①
        p = d ;②
        ...
        thread__id = thread__lfork ( 10 ) ;③
                ...
                ++i ;④
                *p = thread__id ;⑤
                ...
        thread__join ( ) ;⑥
        ...
        printf ( "%d\n" , *p ) ;⑦
        ...
}
```



（a）スタイル①のスレッド生成　　（b）スタイル②のスレッド生成

図5　スレッドの生成

次にスタイル②のプログラムを示す.

```
main ( )
{
        int     *p , d [ 100 ] ;
        ...
        p = d ;①
        ...
        thread__sfork ( 10 , func , p ) ;②
        thread__join ( ) ;③
        ...
        printf ( "%d\n" , *p ) ;④
        ...
}

func ( p )
int   *p ;
{
        ...
        *p = getthreadid ( ) ;⑤
        ...
}
```

この場合,②で関数thread__sforkのパラメータに自動変数へのポインタを渡すことによって,スタイル①と同様に同じスタックの中の配列を共有することになる.しかし,c言語の文法通りであり,関数funcがスレッドの本体になるという事が理解されていれば,⑤のような記述は行われることはない.また,スレッドの本体となる関数を実行しても,スレッド生成のシステム・コールによって間接的に呼び出された関数であるため,その関数は直接値を返すことができない.仮に,スレッド生成のシステム・コールによって間接的に値を返せるようにすると,スレッドの生成が成功したのか失敗したのかがわからなくなる.従って,スレッドの本体となる関数からは値を返さないことにする.値を返す場合は,静的な領域を介して行う必要がある.

3.6 スレッドに関するシステム・コール

thread__lforkとthread__sforkは,スレッドの生成とその適用方法が異なり,前述したように2つのプログラミング・スタイル(それぞれ①,②)を記述できる.我々が考慮中のスレッド関連の主なシステム・コールを示す.

- thread__lfork (n)
  この関数以降の実行をn個のスレッドによって行う.スレッドidを返す.(スタイル①)
- thread__sfork (n , func , arg1 , •••)
  関数funcをn個のスレッドで実行する.
  arg1, ••• は関数funcへの引数である.
  成功時0,失敗時−1を返す.(スタイル②)
- thread__join ( )
  プロセスに属しているスレッドを1つのスレッドに収束させる.成功時0,失敗時−1を返す.
- getthreadid ( )
  このシステム・コールを発行したスレッドのidを得る.
- thread__suspend ( )
  このシステム・コールを実行したスレッドは待ち状態に移行する.

4. 開発の現状

現在のところSun-4上に,マルチスレッドが実現できる環境を開発中である.

Sun-4自身はシングルプロセッサのマシンであるが,UNIXのプロセスを1つのプロセッシング・エレメント(PE)として,プロセス間の共有メモリ上にPE間の通信のために使用する領域とスレッドのテキスト領域とデータ領域を割り当て,更に,個々のスレッドのスタック領域も共有メモリ上にそれぞれ割り当てる(図6).また,この環境はスーパーバイザ・モードで実行を行っているのではないため,システム・コールはトラップではなくブランチ命令で特定のアドレス(フック)に一旦飛び,そこからこの環境のシステム・コールの本体に相当する関数を呼び出す.

マルチスレッドの実現は次のような順序で行われる.

(1) 初期状態では1つのテキスト領域とデータ領域および1つのスタック領域が有り,1つのスレッドが走行している.このスレッドに関する情報はスレッド・コントロール・ブロックとしてPE間の通信のための共有メモリ上に存在する.

(2) スレッド生成のシステム・コールによって,PE間の通信のための共有メモリ上に必要とされるだけのスレッド・コントロール・ブロックを準備する.

(3) スレッド生成のシステム・コールを受信したPEはPE間の通信（実際は共有メモリによるプロセス間通信）によって他のPEに連絡をとる.

(4) 通信を受けたPEは，PE間の共有領域上にスタック領域を確保する.

(5) 各々のPEはスレッド本体を実行する.

(6) スレッド収束のシステム・コールを発行したスレッドは，実行を停止する.

(7) すべてのスレッドがスレッド収束のシステム・コールを発行し終ると，スレッドidが最も小さなスレッド（現段階ではスレッドidが0のスレッド）がシステム・コールの直後から処理を再開し，他のスレッドは消滅する.

(8) プログラムの終了まで1つのスレッドが実行する.

5. おわりに

以上，開発中の並列／分散OSのスレッド実現に関して述べた．マルチスレッドの実行環境を早期に実現し，共有メモリ構成時の性能予測を行う予定である.

謝辞

我々と共に開発を進めている森，蒲池，廣谷，岩田，甲斐の各氏，および日頃ご討論頂く富田研究室の皆様に感謝致します.

参考文献

[1] K. Murakami, S. Mori, A. Fukuda, T. Sueyoshi, and S. Tomita : "The Kyushu University Reconfigurable Parallel Processor － Design Philosophy and Architecture －", Proc. of IFIP 11th World Computer Congress, pp.995－1000 (1989).

[2] K. Murakami, S. Mori, A. Fukuda, T. Sueyoshi, and S. Tomita : "The Kyushu University Reconfigurable Parallel Processor － Design of Memory and Intercommunication Architectures －", Proc. of ACM SIGARCH Int'l Conf. on Supercomputing, pp.351－360 (1989).

[3] 福田，福澤，廣谷，村上，末吉，富田：" 可変構造型並列計算機の並列／分散オペレーティング・システム", 情報処理学会オペレーテイング・システム研究会, 89－OS－43－8 (1989).

[4] 福澤，廣谷，福田，村上，末吉，富田：" 可変構造型並列計算機のソフトウェア", 第3回情報処理学会九州支部研究会，pp. 29－38 (1989).

[5] 恒冨，福澤，福田，村上，富田：" 可変構造型並列計算機の並列／分散オペレーティング・システム －プロセス管理－", 情報処理学会第39回全国大会講演論文集，6M－7，pp.1030－1031 (1989).

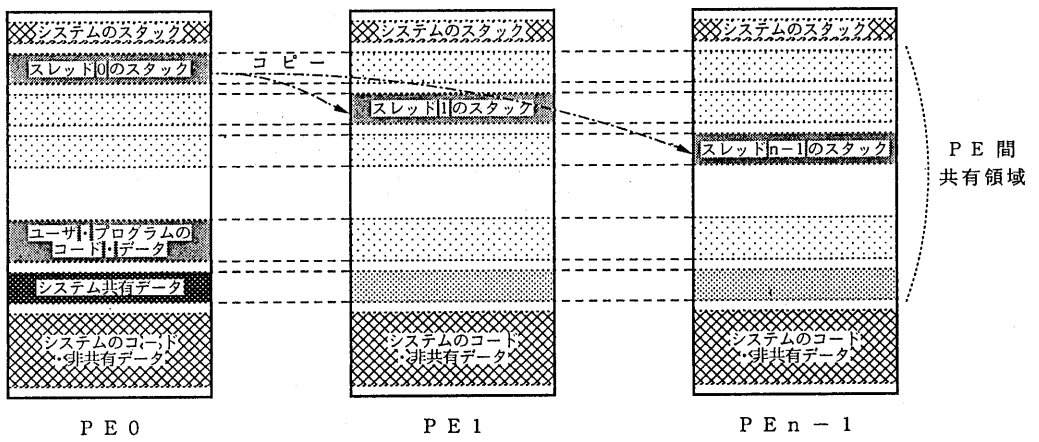


図6 Sun-4上でのマルチスレッドの実現