仮想プロセッサを支援するオペレーティング・システム・カーネルの構成法

新城 靖[†] 清木 康[††]

軽量プロセスの実現方式として，利用者レベルの軽量プロセスをカーネルが提供する仮想プロセッサにより実行する方式がある．仮想プロセッサとは，共有メモリ型マルチプロセッサにおいて，利用者プロセスに複数の実プロセッサを割り当てるためのエントリである．この論文は，仮想プロセッサを提供するオペレーティング・システム・カーネルの構成法を提案している．この構成法の特徴は，カーネル自身を1つの並列プログラムとしてとらえ，カーネルをカーネル内の軽量プロセスの集合として構築している点にある．カーネル内の軽量プロセスは，固有のメモリ領域を備えた実プロセッサにより制御される．利用者に対する抽象である仮想プロセッサ，および，プロセスは，カーネル内の軽量プロセス，および，軽量プロセスの集合により実現される．仮想プロセッサ，および，プロセスのスケジューラは，カーネル内の軽量プロセスのスケジューラとして実現されている．結果として，カーネルは，並列利用者プログラムと同じ構造になる．これにより，カーネルの開発が容易になり，カーネル用のモジュールと利用者用のモジュールの共通化を図ることが可能となっている．

An Organization Method of Operating System Kernels Supporting Virtual Processors

YASUSHI SHINJO[†] and YASUSHI KIYOKI[††]

This paper proposes an implementation method of virtual processors in operating system kernels for shared-memory multiprocessors. A virtual processor is an entry of a real processor which is assigned to the application program by the kernel. User-level lightweight processes are executed by virtual processors. The main feature of the proposed method is that an operating system kernel is regarded as a parallel program, and constructed as a set of lightweight processes. The lightweight processes in the kernel are executed by real processors which have private memory areas. A virtual processor and a process, which are abstractions for user programs, are realized as a lightweight process and a set of lightweight processes, respectively. Virtual processor schedulers and the process scheduler are implemented by using a lightweight process scheduler in the kernel. As a result, the operating system kernel is similar to user programs which run on it, and modules for user programs can be used for the kernel itself.

1. はじめに

多重プログラミングの共有メモリ型マルチプロセッサにおいて並列処理を行う場合，軽量プロセス (lightweight processes) は，必要不可欠な機能となってきた．軽量プロセスとは，保護と資源割当ての単位としてのプロセスとは異なる，1つの応用プログラム内部にある並列処理の単位としてのプロセスである．軽量プロセスの実現方式は，次の3つの方法に分類される．

（1） コルーチン方式．軽量プロセスは，利用者空間内においてコルーチンとして実現される．

（2） カーネル制御方式[4]．軽量プロセスは，カーネルにより直接実現される．軽量プロセスのコンテキスト切替えにおいて，カーネル・コールが必要となる．

（3） 仮想プロセッサ方式[1),5),6),8),12)~14]．軽量プロセスは，コルーチンと同様に，すべて利用者アドレス空間内において実現される．これらの軽量プロセスは，カーネルが提供する仮想プロセッサにより実行される．仮想プロセッサ (virtual processor) とは，利用者プロセスに複数の実プロセッサ（ハードウェアのプロセッサ）を割り当てるためのエントリである．軽量

† 筑波大学工学研究科
Doctoral Program in Engineering, University of Tsukuba
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

プロセスのコンテキスト切替えにおいて，カーネル・コールを必要としない．

これらの方法の中で，(3)は，効率良い軽量プロセス生成，コンテキスト切替え，並列処理，および，細かいスケジューリングの制御を実現する上で有力な方法として注目されている．我々は，既に文献12)において，(3)に基づく軽量プロセスの実現方式を提案した．その論文では，軽量プロセスを支援するライブラリの外部仕様と内部構造，性能，ならびに，仮想プロセッサを提供するカーネルの外部仕様を提案した．本論文では，提案した外部仕様を満たす仮想プロセッサの実現方式，すなわち，仮想プロセッサを提供するカーネルの構成法について述べる．ここで述べるカーネルの構成法は，一般性があるので，異なる外部仕様を持つ仮想プロセッサを提供するカーネルや(2)に基づくシステムのカーネルの構成法として利用可能である．

先に提案した軽量プロセスの実現方式の特徴の1つとして，仮想プロセッサごとに固有のメモリ領域が存在する点があげられる．文献12)では，その存在について触れただけであり，その利用法，実現方式については，述べていなかった．本論文では，カーネル内における実プロセッサごとに固有のメモリ領域と合わせて，詳しく述べる．仮想プロセッサごとの固有のメモリ領域により，利用者レベルの軽量プロセスの実現が容易になるばかりでなく，並列処理の効率が改善される．また，従来の逐次プログラム用に開発されたライブラリ関数を一切変更することなく並列プログラムにおいて利用するためにも用いられる．

オペレーティング・システムのカーネルは，カーネル上で動作する利用者プログラムと比較して，開発とデバッグが困難であるため，その構成法は，非常に重要である．オペレーティング・システムの機能の大部分をカーネルの外で動作するサーバ・プロセスにより提供することで，カーネルを小さくすることも試みられている[3),4),9)]．しかしながら，仮想プロセッサ機能を提供する部分をカーネルの外に移動することは困難である．したがって，そのようなカーネルを構成する方法が重要な研究課題となっている．本論文では，仮想プロセッサを提供するカーネルの実現を容易にする構成法を提案する．具体的には，カーネルの一部を利用者プロセスとして実行し，動作の確認，および，デバッグを可能にする方法を提案する．

本カーネル構成法の特徴を，以下にまとめる．

(1)　カーネル自身を1つの並列プログラムとしてとらえることで，カーネルを（カーネル内の）軽量プロセスの集合として構成している．

(2)　カーネル内の軽量プロセスは，固有メモリ領域を供えた実プロセッサにより実行される．固有メモリ領域は，文献12)で提案した仮想プロセッサの固有メモリ領域と同一の技術により実現される．

(3)　（利用者に対する抽象である）仮想プロセッサとプロセスを，（カーネル内の）軽量プロセスとその集合として実現する．これにより，プロセス・スケジューラ（大域スケジューラ）と仮想プロセッサ・スケジューラをカーネル内の軽量プロセス・スケジューラとして実現することが可能となる．

(4)　上記の軽量プロセス・スケジューラは，2段階のレディ・キューを持っている．これにより，同一プロセス内における仮想プロセッサのコンテキスト切替えが高速化され，また，プロセスの操作も容易になる．

結果として，カーネルは，並列利用者プログラムと同じ構造を持つことになる．これによりカーネルの開発において，その一部を利用者プログラムとして走らせることが可能となり，カーネルの開発とデバッグが容易になる．さらに，利用者プログラムにサービスを提供するモジュールとカーネル自身の動作のために必要なモジュールの共通化を図ることができるという利点も生じる．

第2章では，文献12)において提案した軽量プロセス実現方式の概要についてまとめる．第3章では，仮想プロセッサごとの固有メモリ領域の外部仕様とその利用を説明し，それを実現するカーネルの内部構造について述べる．第4章では，実プロセッサごとの固有領域について述べる．第5章では，カーネル内部のスケジューラの外部仕様と目的を説明し，それを実現する内部構造について述べる．第6章では，提案する構成法に基づき，裸の共有メモリ型マルチプロセッサにおけるカーネルの実現について述べる．第7章では，関連した研究との比較を行う．最後にまとめを行う．

2. マイクロプロセス／仮想プロセッサに基づく軽量プロセスの実現方式の概要

我々は，軽量プロセスを多重プログラミングのオペレーティング・システムにおける個々の応用プログラム内の並列処理の単位としてのプロセスとして位置付けている．多重プログラミングのオペレーティング・

システムにおいて，プロセスは，資源割当てと保護の単位を指す言葉として用いられてきた．一方，単一プログラミング環境における並列処理においては，プロセスは，並列処理の単位を指す言葉として用いられてきた．並列処理におけるプロセスは，資源割当てや保護の機能を含んでいない．我々は，これら2つのプロセスを明確に区別することで軽量プロセスを規定している．すなわち，プロセスは，資源割当て，および，保護の単位である．軽量プロセスとは，プロセスの内部にあり，個々の応用プログラム内部の並列処理の単位としてのプロセスである．これをマイクロプロセス (microprocess) とよんでいる．仮想プロセッサ (virtual processor) は，カーネルがプロセスに対して実プロセッサを割り当てるためのエントリである．各仮想プロセッサは，利用者プロセスと対応しており，その利用者プロセスに割り当てられた資源，利用者識別子，アクセス権，プロセスの優先順位などを共有する．

仮想プロセッサは，主に次の3つのカーネル・コールを通じて利用される．

（1） vpid＝vp_allocate(n)

これは，新たに $n-1$ 個の仮想プロセッサを生成することを要求するカーネル・コールである．返り値は，仮想プロセッサの識別子である．(普通，返り値は，仮想プロセッサに対応している実プロセッサのレジスタに格納されて利用者に渡される．)

（2） vp_switch(vpid)

これは，同一プロセス内の他の仮想プロセッサへ制御を切り替えるためのカーネル・コールである．これは，スピンロックを実現するルーチンにおいて用いられる．スピンロックにより相互排除を実現している場合，際どい領域 (critical section) を実行中に実プロセッサが横取り (preempt) されることがある．この時，他の仮想プロセッサが同一のスピンロックを確保することを試みると，大量の CPU 時間が浪費される．このカーネル・コールは，これを避けるために用いられる．vpid は，次に実行すべき仮想プロセッサのヒントである．

（3） vp_sleep(t)

これは，一時的に不用になった実プロセッサをカーネルに返すためのカーネル・コールである．このカーネル・コールは，マイクロプロセス・スケジューラにおいて，実行可能なマイクロプロセスが存在しなくなった時に呼び出される．t は，実プロセッサを放棄する時間のヒントである．システムの負荷が軽い時には，指定された時間よりも早く復帰する．システムの負荷が重い時には，指定された時間が経過した後に，カーネル内のスケジューラにおいて優先順位等に従い，実プロセッサが割り当てられるかどうかが決定される．

仮想プロセッサを削除するカーネル・コールは，存在しない．それは，仮想プロセッサの数は，実プロセッサの数と同様に，プロセスの実行の途中で変化しないからである．

3. 仮想プロセッサの固有領域

3.1 仮想プロセッサの固有領域の外部仕様

仮想プロセッサの固有領域 (private area) とは，アドレス空間の一部に存在する，仮想プロセッサごとに異なる物理メモリが参照される領域のことである．これに対して，固有領域以外の領域を共有領域 (shared area) とよぶことにする．

図1に，1つのプロセスに属する2つの仮想プロセッサの論理アドレス空間と物理メモリの対応を示す．これは，**図2**に示すプログラムの実行の結果，構築されるものである．図2第7行の vp_allocate() カーネル・コールにより，新たに仮想プロセッサが1個作られ，合計2つの仮想プロセッサが活動することになる．

図1では，アドレス空間の上部（アドレスが大きい方）に固有領域(①)，アドレス空間の下部に共有領域(②)が割り付けられている．共有領域は，テキスト・セグメント (text segment) (③)，データ・セグメント (data segment) (④) に分けられる．前者は，機械

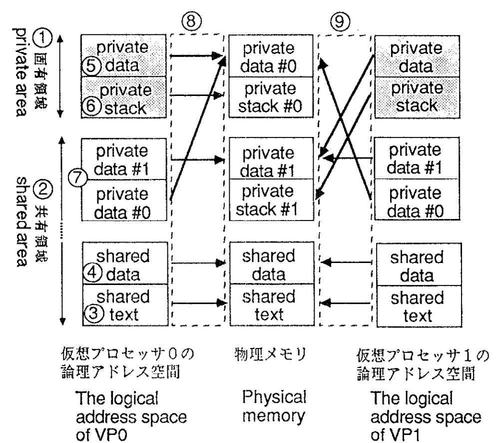


図 1 仮想プロセッサの論理アドレス空間と物理メモリの対応

Fig. 1 Mapping of the logical address spaces of two virtual processors and physical memory.

```
 1: private int vpid ;           /* Virtual Processor ID */
 2: private struct mpcb *mp_current ; /* Microprocess Control
 3:                                       Block */
 4: main()
 5: {
 6:     init_single();            /* initialize in a sigle VP. */
 7:     vpid = vp_allocate( 2 ); /* create a new VP. */
 8:     init_multi();             /* initialize in multiple VPs. */
 9:     mp_scheduler();           /* jump into mp_scheduler(). */
10:     /* Not reached. */
11: }
```

図 2 仮想プロセッサを生成するプログラムの断片
Fig. 2 A program fragment which produces a new virtual processor.

語命令を格納する部分であり，読込み専用になっている．後者は，データを格納する部分であり，読み書き可能になっている．固有領域は，固有データ・セグメント（private data segment）(⑤) と固有スタック・セグメント（private stack segment）(⑥) に分けられ，両方とも読み書き可能である．前者は，仮想プロセッサごとに固有の変数を割り付けるために利用される．後者は，マイクロプロセス・スケジューラ（図2第9行の mp_scheduler()) のスタックとして用いられる．各仮想プロセッサの固有データ・セグメントは，他の仮想プロセッサによる参照を可能にするために，アドレス空間の一部に割り付けられている(⑦).

図2において，キーワード private（第1行，第2行）は，この変数を固有領域に割り付けるように指示するものである．このような機能は，Dynix システム[2)]におけるいくつかの言語処理系において既に実現されている．Dynix では，UNIX のオブジェクト・コード形式（a. out 形式）を拡張して，共有と固有の２つのデータ・セグメントを提供している．

3.2 仮想プロセッサの固有領域の利用

仮想プロセッサの固有領域は，次の４種類のデータを格納するために用いられる．(１)，(２)，(３)は，固有データ・セグメントに，(４)は，固有スタックセグメントに割り付けられる．

(１) 仮想プロセッサの識別子．

(２) 仮想プロセッサごとに必要となるデータ．利用者空間において軽量プロセスを実現する場合，このようなデータが必ず必要になる．たとえば，現在実行しているマイクロプロセスの制御ブロック(mpcb, microprocess control block) へのポインタが仮想プロセッサごとに必要となる（図２の第２行).

(３) 共有する必要がない静的変数．たとえばＣ言語のライブラリ関数の中には，静的変数（static variable) に結果を代入してそのポイントを返すものや，静的変数に中間状態を格納するものが多数存在する．前者の例として，時刻を文字列に変換する ctime()，後者の例としては，引数を解析するルーチン getopt() があげられる．このような静的変数を固有領域に割り付けることにより，逐次プログラム用に開発されたライブラリ関数を並列プログラムにおいて利用することが可能となる．

(４) マイクロプロセス・スケジューラ用のスタック．これは，(２) の特殊なものである．(どの仮想プロセッサも任意のマイクロプロセスを実行することを可能にするために，マイクロプロセスのスタックは，共有領域に置かれる.)

固有スタック・セグメントの存在により，図２に示したようなプログラムをスタック・ポインタの切替えなしに動作させせることが可能になっている．たとえば，図２の第７行において，カーネル・コール vp_allocate() を発行する仮想プロセッサ（仮想プロセッサ０）も新たに生成される仮想プロセッサ（仮想プロセッサ１）も，同一のアドレスに復帰する．この時，どの仮想プロセッサも同一のアドレスにあるスタックを利用することが可能となる．

3.2.1 固有領域を利用しない方法との比較

仮想プロセッサごとの固有領域を利用しない方法として，仮想プロセッサの識別子を実プロセッサのレジスタに格納する方法が考えられる．この方法では，上記の(２)，(３)，(４)を（共有領域にある）配列として実現しなければならない．たとえば，現在実行中のマイクロプロセス制御ブロック（struct mpcb）へのポインタ mp_current は，**図３**に示すように配列を用いて実現することができる．ここで，MAX_NVP（図３第１行）は，仮想プロセッサ数の最大値である．get_vpid()（図３第３行）は，実プロセッサに格納された仮想プロセッサの識別子を参照するための関数で

```
1: #define MAX_NVP 32
2: struct mpcb *mp_current_array[MAX_NVP];
3: #define mp_current mp_current_array[get_vpid()]
4: ....
5: {
6:     mp_lock( mp_current );
7:     ....
8:     mp_unlock( mp_current );
9: }
```

図 3 仮想プロセッサの識別子と配列による仮想プロセッサごとの変数
Fig. 3 A per-virtual-processor variable by using virtual processor identifiers and an array.

ある．このプログラムでは，マクロを用いて配列を単純な変数として扱えるようにしている．

配列と実プロセッサのレジスタに格納された仮想プロセッサの識別子を用いる方法には，次のような問題点がある．

（1） 配列のアドレス計算のオーバヘッドにより実行速度が低下する．

（2） C言語のように静的に配列の大きさを決定する言語では，最大の仮想プロセッサ数分のメモリが常に必要になり，メモリの無駄が生じる．メモリの無駄を避ける方法として，ヒープ上に必要なメモリを動的に確保する方法が考えられる（C言語では，malloc() ライブラリ関数を用いる）．しかしながら，この方法では，メモリへのアクセス回数が増加し，実行速度の低下を招く．

（3） 実プロセッサのレジスタを1個専有することになるため，応用プログラムの利用可能レジスタ数が減る．

（4） 大域的なレジスタ変数機能を提供している言語処理系が一般的に利用可能にはなっていない．

仮想プロセッサの固有領域を用いることにより，これらの問題を解決することができる．

仮想プロセッサの識別子を実プロセッサのレジスタに格納する代りに，必要になった時にカーネル・コールを発行する方法も考えられる．しかしながら，利用者空間内において軽量プロセスを実現しているという利点が失われてしまうので，この方法を採用することはできない．

3.3 仮想プロセッサの固有領域の内部実現

この節では，3.1節で述べた外部仕様を実現するカーネルの構造について述べる．論理的には，図1に示したようなアドレス変換（図1の⑧と⑨）を実現するように MMU (Memory Management Unit) の変換表 (translation table) を作成すればよい．しかしながら，単純に1つの仮想プロセッサについて変換表を作成したならば，1つのプロセスについて仮想プロセッサの数だけの変換表が存在することになり，次のような問題点が生じる．

（1） 変換表の操作の実行速度が低下する．たとえば，プロセスにメモリを割り当てた場合，そのプロセスに属するすべての仮想プロセッサについて変換表を書き換えなければならない．そのためメモリ資源を割り当てる操作は，仮想プロセッサの数に比例することになる．

（2） 変換表のためのメモリ資源が無駄になる．

この節では，変換表の，利用者空間における共有領域に対応する部分を共有することにより上記の問題を解決する方法を示す．

3.3.1 2段階の変換表を利用する MMU

この項では，図4に示すような2段階の変換表を利用する MMU における変換表の設定について述べる．実際に利用したのは，モトローラ社の MC88200 である[11]．変換表の第1段をセグメント表，第2段をページ表とよぶ．CPU が生成する論理アドレスは，上位ビットからセグメント番号，ページ番号，ページ内オフセットに分けられる．アドレス変換は，次のようにして行われる．

（1） 論理アドレス中のセグメント番号を添え字としてセグメント表のエントリを選択する．

（2） そのエントリには，ページ表の物理アドレスが含まれている．それを用いて，ページ表を選択する．

（3） そのページ表において，論理アドレスのページ番号を添え字としてエントリを選択する．

（4） そのエントリには，メモリの物理アドレスが含まれている．このアドレスとページ内オフセットを加えて，物理アドレスを生成する．

3.3.2 プロセス変換表と仮想プロセッサ変換表

この項では，プロセス用と仮想プロセッサ用の2種

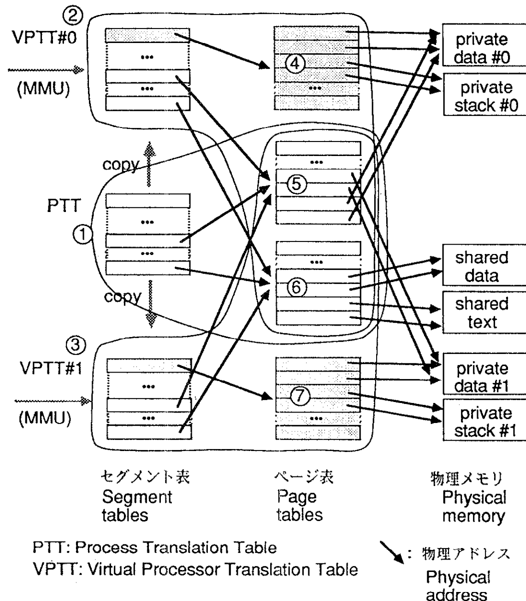


図 4 仮想プロセッサの固有領域を実現するアドレス変換表
Fig. 4 Address translation tables for the private areas of two virtual processors.

類の変換表を用いて仮想プロセッサのアドレス空間を設定する方法を示す．前者をプロセス変換表 (PTT, Process Translation Table, 図4の①)，後者を仮想プロセッサ変換表 (VPTT, Virtual Processor Translation Table, 図4の②と③) とよぶことにする．プロセス変換表は，共有領域の管理を行うためのものであり，MMU によって参照されることはない．仮想プロセッサ変換表は，その仮想プロセッサを実プロセッサが実行する時に MMU により参照するための変換表であり，共有領域と固有領域の両方のアドレス変換に関する記述を含む．

図1に示した論理アドレスから物理メモリへの変換を実現する変換表を図4に示す．図4は，図1と同様に，アドレスが小さい（0に近い）部分が下になるように描かれている．図1の⑧，⑨に示した変換を実現しているのは，図4の②，③に示す2つの仮想プロセッサ変換表である．それらの変換表（②，③）は，2つのページ表（⑤，⑥）を共有している．そして，物理ページも共有している．この部分は，3.1節において述べた共有領域に対応している．

図4において，影を付けた部分が固有領域に関係している部分である．たとえば，図1の⑤に示した固有データ・セグメントの論理アドレスは，図4の②と④を用いて物理アドレスに変換される．

3.3.3 共有領域へのメモリの割り付け

仮想プロセッサのアドレス空間の大部分は，共有領域である．したがって共有領域への物理メモリの割り付けの速度が重要となる．ここでは，ページ表を共有している利点を活用し，高速化を図る手順を示す．

（1） プロセス変換表（図4の①）を根として，ページ表を共有しない方式と全く同じ手順を用いて，変換表を構築する．

（2） （1）において，プロセス変換表のセグメント表のエントリの中で変更したものを，そのプロセスに属するすべての仮想プロセッサの変換表のセグメント表（図4では②，③）にコピーする．（1）においてセグメント表を変更しなかった場合は，なにもしない．

（3） 実プロセッサが割り当てられている仮想プロセッサについて，MMU 中の変換表のキャッシュ (TLB, Translation Look-aside Buffer, MC88200では，PATC (Page Table Translation Cache) と呼ばれている[2]) をフラッシュする．

（4） 実プロセッサが割り当てられていない仮想プロセッサについて，変換表が変更されたことを示すマークを付ける．

これらの操作の中で，（2）の操作においてページ表を共有している利点が活かされている．もしページ表を共有していなかったならば，（1）の操作を仮想プロセッサの数だけ繰り返す必要が生じる．

ページ表を共有するためには，ある制限が必要である．それは，セグメント表の1エントリによって記述されるアドレス空間内に固有領域と共有領域の混在を許さないことである．この制限により，上の（2）において単純にセグメント表のエントリのコピー（浅いコピー，shallow copy）を行うだけでよいことになる．

4. 実プロセッサの固有領域

3章では，利用者プロセスにおける仮想プロセッサごとの固有領域の外部仕様，利用，および，内部実現について述べた．一方，カーネルの中においては，実プロセッサごとに必要となるデータが存在する．ゆえに，カーネルにおいて実プロセッサごとの固有領域を設定することは，有効である．この章では，その外部仕様，利用，および，内部実現について述べる．

4.1 実プロセッサの固有領域の外部仕様

実プロセッサの固有領域とは，アドレス空間の一部に存在する，実プロセッサごとに異なる物理メモリが参照される領域のことである．これに対して，実プロセッサの固有領域以外の領域を実プロセッサの共有領域とよぶことにする．

実プロセッサの固有領域は，利用者レベルのアドレス空間の設定から独立している．すなわち，別の外部仕様を持つ仮想プロセッサを提供するカーネルや，カーネル・レベルの軽量プロセスを提供するカーネルの実現においても，実プロセッサの固有領域を利用することができる．たとえば，Mach システムのカーネルを実現する場合においても，実プロセッサの固有領域を設定し利用することが可能である．

4.2 実プロセッサの固有領域の利用

3.2節で行った議論は，仮想プロセッサを実プロセッサに置き換えてもそのまま成り立つ．

4.3 実プロセッサの固有領域の内部実現

実プロセッサの固有領域は，利用者空間とカーネル空間の変換表の形式が同じ場合，3.3節で述べた仮想プロセッサの固有領域とほとんど同じ技術を用いて実現することが可能である．これにより，利用者プログラムに提供するモジュールとカーネル自身の動作のために必要なモジュールの共通化を図ることができると

いう利点が生じる．

利用者空間とカーネル空間の変換表の形式が異なる場合には，共通化を図ることができない．しかしながら，カーネル空間において MMU が利用可能であるならば，実プロセッサの固有領域を実現することは，可能である．

固有領域の実現において，仮想プロセッサと実プロセッサで異なる操作が必要な点が存在する．それらを以下にまとめる．

（1） カーネルには，固有スタック領域（スケジューラ用のスタック）が不用である．これは，自力で立上がる (bootstrap) 時に利用するスタックをそのままスケジューラ用のスタックとして利用できるからである．

（2） カーネル用の変換表が完成するまで，セグメント表やページ表に用いるページの属性の設定を遅らせる必要がある．セグメント表やページ表に用いる物理メモリは，キャッシングの機能を停止しなければならない．そのためには，カーネル用のセグメント表やページ表があるページの属性を変更する必要がある．ところが，カーネル用の変換表を作成している途中の段階では，この属性変更を行うことは不可能である．したがって，一度変換表を作成した後に改めてセグメント表とページ表の物理アドレスを調べ，該当するページの属性を変更する必要がある．

5. 仮想プロセッサとプロセスのスケジューラ

3章では，仮想プロセッサのアドレス空間の設定についた述べた．この章では，仮想プロセッサのスケジューラとプロセスのスケジューラの実現について述べる．

4章では，利用者レベルの抽象である仮想プロセッサの固有領域と同一の技術を用いて実プロセッサの固有領域を実現する方法について述べた．この章では，利用者レベルの軽量プロセスのスケジューラ（マイクロプロセス・スケジューラ）の技術を用いて，仮想プロセッサとプロセスのスケジューラを実現する方法について述べる．

5.1 プロセス・スケジューラと仮想プロセッサ・スケジューラの外部仕様

仮想プロセッサ・スケジューラとは，同一プロセス内において，別の仮想プロセッサへの制御の移動を行うためのスケジューラである．制御の移動の原因としては，同期式入出力，同期式プロセス間通信，仮想記憶システムではページ・フォールト等に関する仮想プロセッサの停止／再開，および，制御の移動を要求するカーネル・コール（2章で述べた vp_switch()）の発行があげられる．

プロセス・スケジューラ (大域スケジューラ (global scheduler)) とは，資源割当ての単位であるプロセスの間で公平な CPU 時間の分配を実現するためのスケジューラである．公平な分配を実現するために，実プロセッサの横取り (preemption) が行われる．

ここでは，それぞれのスケジューラにおける方針 (policy) とは独立した構成法を示す．ここで述べる構成法は，プロセスと仮想プロセッサという概念が存在するならば，他の方針においても利用可能である．(文献12において，仮想プロセッサ・スケジューラでは，横取りを行わない，プロセス・スケジューラでは，横取りを行うという方針を提案した．)

5.2 プロセス・スケジューラと仮想プロセッサ・スケジューラの内部構造

5.2.1 カーネル内の軽量プロセスとしての仮想プロセッサ

オペレーティング・システムのカーネルは，利用者からのトラップを扱う視点から，次の2つの方式に分類される[4]．

（1） プロセス・モデル：利用者プロセスと1対1に対応し，固有のスタックを持つプロセスをカーネル内に設ける．後述する割込みモデルと比較して，実現が容易である．UNIX システムにおいて採用されている．

（2） 割込みモデル：カーネルは，利用者からのトラップを割込みとして扱う．カーネル内には，プロセッサごとにスタックを設ける．前述のプロセス・モデルと比較して，効率的な実現が可能である．Mach 3.0 システム[4]，V システム[3]等で採用されている．

ここでは，(1)を採用する．その理由は，仮想プロセッサをカーネル内の軽量プロセスとして実現することで，仮想プロセッサとプロセスのスケジューラをカーネル内の軽量プロセスのスケジューラとして実現可能であるからである．さらに，オペレーティング・システムのカーネルを1つの並列プログラムと考えた場合，(利用者レベルの抽象である) 仮想プロセッサは，並列処理の単位となり得るため，軽量プロセスにより実現することが自然である．(カーネル中には，仮想プロセッサ以外にも軽量プロセスが存在する．)

5.2.2 カーネル内軽量プロセスの集合としての利用者プロセス

実プロセッサの制御という視点では，プロセスは，仮想プロセッサを実現している軽量プロセスのグループとしてとらえることができる．たとえば，プロセスを削除することは，そのグループに属するすべての仮想プロセッサを削除することに相当する．プロセスを停止させることは，そのグループに属する実行中の軽量プロセスから実プロセッサを奪いとることに相当する．

5.2.3 2段階のレディ・キューを持つスケジューラ

5.1.1項，および，5.1.2項において述べたように，仮想プロセッサは，カーネル内の軽量プロセスであり，プロセスは，カーネル内の軽量プロセスの集合である．この構造に従い，**図5**に示すような2段階のレディ・キューを用いてスケジューラを構築することができる．第1段は，(実行可能な仮想プロセッサを保持している) プロセスを保持するレディ・キューである．これを操作する部分が，5.1節で述べたプロセス・スケジューラに相当する．第2段は，各プロセス内部の実行可能な仮想プロセッサを保持するレディ・キューである．これを操作する部分が，5.1節で述べた仮想プロセッサ・スケジューラに相当する．

図5は，プロセスのレベルにおいて8つ，仮想プロセッサのレベルにおいて4つの優先順位に分割されたレディ・キューを示している．実プロセッサは，プロセス・レディ・キューからプロセスを選ぶ．そしてそのプロセス中の仮想プロセッサ・レディ・キューから実行可能な仮想プロセッサを得て，それを実行する．

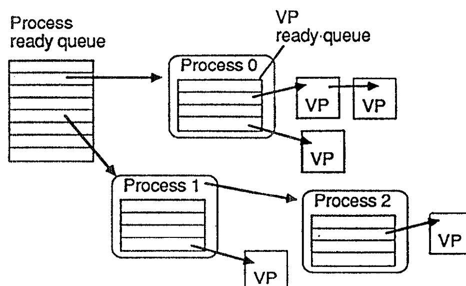


図 5 2段階のレディ・キューによるプロセスと仮想プロセッサのスケジューラ

Fig. 5 The process scheduler and virtual processor schedulers by using two-level ready queues.

プロセスの操作

このような構造を持つスケジューラでは，プロセスの操作は以下のように実現される．

(1) 停止：停止するプロセスをプロセス・レディ・キューから削除する．そのプロセスの実行中状態の仮想プロセッサについて，それを実行している実プロセッサに割込みを行う．割込まれた実プロセッサは，実行していた仮想プロセッサを，そのプロセスの仮想プロセッサ・レディ・キューに入れる．実プロセッサは，他のプロセスの仮想プロセッサに制御を移す．

(2) 実行再開：実行を再開するプロセスを単にプロセス・レディ・キューに接続するだけでよい．

(3) 生成：アドレス空間や最初の仮想プロセッサを生成した後，(2)を行う．

(4) 終了，および，削除：(1)の操作を行った後，プロセスが保持している資源を開放し，終了コードを保存する．

このように，プロセスの操作は，プロセス・レディ・キューへの操作により容易に実現される．

6. 実　　現

提案した構成法に基づき，共有メモリ型マルチプロセッサ Luna 88 k[7]において仮想プロセッサを提供するカーネルの実現を行った．この実現においては，次の2点を確認することを目標とした．

(1) カーネルの大部分を利用者プログラムと同様に開発することが可能であること．

(2) 仮想プロセッサの性能が，Mach 2.5 の Threads のようなカーネル制御方式における軽量プロセスの性能と同程度であること．Mach 2.5 の Threads は，本構成法と同様，プロセス・モデルに基づき実現されている．

実現に用いた Luna 88 k は，4プロセッサ構成で，主記憶容量は，32 M バイトである．各プロセッサは，命令とデータそれぞれ 16 k バイトのキャッシュを供えている．Luna 88 k に付属のオペレーティング・システムである UniOS-Mach 1.10[7] (Mach 2.5) を開発システムとして利用した．

3章，および，4章において述べた固有領域の実現において，ハードウェア (MMU) を操作する部分を除き，その大部分を開発システム上でデバッグ，および，動作の確認を行うことができた．5章において述べたスケジューラについても，割込み制御の部分を除

表1　実現システムの性能（単位：m秒）
Table 1　Performance of our system. (In milliseconds)

Type＼Operation	Creation	Switch
Process (Kernel)	15.	0.121
Virtual Processor (Kernel)	7.7	0.068
Microprocess (User-level LWP)	0.19	0.031

き，その大部分を開発システム上でデバッグと動作の確認を行うことができた．これにより，上記の目標(1)が達成された．

実現したシステムの性能を**表1**に示す．実現システムのプロセスと仮想プロセッサは，開発システムとして用いたMach 2.5 システムのプロセス（UNIX プロセス）と Threads と同程度の性能が得られることが確認された．ただし，本実現の方が内部構造が単純であるため，生成において25%，コンテキスト切替えにおいて3倍程度高速であった．

文献12)において提案した利用者レベルの軽量プロセスであるマイクロプロセスと比較すると，仮想プロセッサは，生成において40倍，コンテキスト切替えにおいて2倍ほど，実行時間が長くなっている．これは，カーネル・コールに伴うトラップ処理，および，パラメタのチェックに伴うオーバヘッドによる．

7. 他のシステムとの比較

7.1 共有 fork との比較

3.1節で述べたアドレス空間の構造は，Dynix システム[2]の fork システム・コール，および，Sprite システム[10]における共有 fork (shared fork) の結果と類似している．相違点は，Dynix や Sprite の場合，新たにプロセスが作られる点にある．すなわち，元のプロセスと新たに生成されたプロセスは，異なるプロセス識別子を持ち，それらが協調して動作していることは，外部から観測している限り分からない．本方式では，プロセスと仮想プロセッサは異なるレベルにあり，任意のプロセスは，内部の仮想プロセッサの数にかかわりなく一様に操作することが可能である．たとえば，プロセスが終了すると，内部のすべての仮想プロセッサも自動的に消去される．これに対して，Dynix や Sprite では，個々のプロセスが別々に終了する．

7.2 他の仮想プロセッサとの比較

カーネルにおいて仮想プロセッサを提供し，利用者レベルの軽量プロセスを実行する方式は，文献12)～14)において提案した方式のほかにも提案されている[1),5),6),8)]．しかしながら，これらの論文では，主にカーネルの外部仕様についての提案を行っており，そのようなカーネルをいかに構成するかについては，詳細に記述されていない．この論文では，カーネルの構成法について提案を行った．その特徴は，利用者レベルの並列応用プログラムとほとんど同じ技術により，仮想プロセッサを実現するカーネルを構成している点にある．すなわち，実プロセッサに固有領域が存在すること，および，カーネル内の軽量プロセスのスケジューラにより仮想プロセッサとプロセスのスケジューラが実現されている点にある．

この論文では，文献12)において提案した仮想プロセッサの外部仕様の具体的な内部実現を示した．この論文で提案した実プロセッサの固有領域やカーネル内軽量プロセス・スケジューラによる仮想プロセッサとプロセスのスケジューラは，異なる外部仕様を持つ仮想プロセッサ，あるいは，カーネル・レベルの軽量プロセスを提供するカーネルの実現においても利用することが可能である．たとえば，スケジューラ活動体 (scheduler activations)[1]や Mach の Threads[4]と同等の機能をもつものを本論文で提案した固有領域や2段階のレディ・キューを持つスケジューラを用いて実現することも可能である．ただしこの場合，カーネルと利用者プロセスは，全く異なる構造を持つことになる．

7.3 軽量プロセスの集合によるカーネルの構築

分散型オペレーティング・システムを軽量プロセスの集合として実現する方法は，文献15)において提案されている．その方法では，相互排除アクセスされる資源の管理単位に軽量プロセスを配置する．軽量プロセス間のデータのやりとりは，軽量プロセス間の共有メモリではなく，プロセス間通信で行う．このため，オペレーティング・システムのカーネル自身をネットワーク上の別のプロセッサに配置することが可能となっている．

この構成法と本構成法の最大の違いは，本構成法では，並列処理の単位に軽量プロセスを配置する点にある．また，本構成法では，軽量プロセスが同一アドレス空間に存在することを利用して，軽量プロセス間のデータのやり取りを軽量プロセス間の共有メモリを用いて行う．

7.4 アクティビティ方式との比較

文献16)では，共有メモリ型マルチプロセッサにおいて並列実行環境を提供する方式として，アクティビ

ティ方式が提案されている．アクティビティとは，手続きと引数の組のようなもので，軽量プロセスにより実行される．アクティビティは，生成されるとキューに接続される．実プロセッサの個数だけ生成された軽量プロセスは，アクティビティをキューから取りだし，実行する．この時，アクティビティが終了するまで，軽量プロセスとアクティビティの対応が保持される．アクティビティ間の同期により，それを実行していた軽量プロセスが中断されることがある．この場合，新たな軽量プロセスが生成される．

アクティビティ方式と先に提案した軽量プロセス実現方式を比較する場合，次の２つの視点が考えられる．

（１）両者は，直交した概念であり，互補的に利用される．なぜならば，アクティビティをマイクロプロセス（利用者レベルの軽量プロセス）により実行することが可能であるからである．

（２）アクティビティは，（利用者レベルの）軽量プロセスの生成において，キャッシングと遅延の技術を導入したものである．キャッシングとは，終了した軽量プロセスを破壊せずに保存し，次の生成要求を受付けた時に再利用することである．遅延とは，軽量プロセスの生成を，要求受付け時ではなく，実際にプロセッサにより実行する直前まで遅らせることである．

8. おわりに

この論文では，仮想プロセッサを提供するカーネルの構成法について述べた．本構成法の特徴は，利用者プロセスとカーネルが同一の構造を持っている点にある．両者とも軽量プロセスから構成される．そして，利用者レベルの軽量プロセスは，仮想プロセッサにより，カーネル内の軽量プロセスは，実プロセッサにより実行される．各仮想プロセッサ，各実プロセッサは，固有メモリ領域を持ち，それぞれの識別子や固有のデータを保持する．仮想プロセッサは，カーネルの内部において並列処理の単位であり，カーネル内の軽量プロセスとして実現される．プロセスは，仮想プロセッサを実現している軽量プロセスの集合として実現される．利用者レベルの抽象であるプロセスと仮想プロセッサのスケジューリングは，カーネル内の軽量プロセスのスケジューラにより行われる．そのスケジューラは，プロセス・レベルと仮想プロセッサ・レベルの２段階のレディ・キューから構成される．

カーネル内で利用した軽量プロセスのグループ化は，カーネルの外においても有益な機能であると思われる．今後，カーネルと利用者プログラムが同一の構造であることを活かし，利用者レベルの軽量プロセスのグループ化の実現をすすめると同時に，それを支援するためのカーネル機能の拡張を検討していきたい．

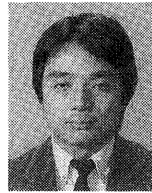
参考文献

1) Anderson, T., Bershad, B., Lazowska, E. and Levy, H.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, SOSP 13, *ACM Operat. Sys. Rev.*, Vol. 25, No. 5, pp. 95-109 (1991).
2) Balance 8000 Parallel Programming, Sequent Computer Systems, Inc. (1985).
3) Cheriton, R.: The V Distributed System, *CACM*, Vol. 31, No. 3, pp. 314-333 (1988).
4) Draves, R., Bershad, B., Rashid, R. and Dean, R.: Using Continuations to Implement Thread Management and Communication in Operating Systems, SOSP 13, *ACM Operat. Sys. Rev.*, Vol. 25, No. 5, pp. 122-137 (1991).
5) Govindan, R. and Anderson, D.: Scheduling and IPC Mechanisms for Continuous Media, SOSP 13, *ACM Operat. Sys. Rev.*, Vol. 25, No. 5, pp. 68-80 (1991).
6) Inohara, S., Kato, K., Narita, A. and Masuda, T.: A Thread Facility Based on User/Kernel Coöperation in the XERO Operating System, *Proc. 15th Intl. Comput. Software & Applications Conf.*, pp. 398-405 (1991).
7) 乾：分散 OS Mach とそのインプリメンテーション，情報処理学会研究会報告，90-OS-49-1 (1990).
8) Marsh, B. and Scott, M.: First-Class User-Level Threads, SOSP 13, *ACM Operat. Sys. Rev.*, Vol. 25, No. 5, pp. 110-121 (1991).
9) Mullender, S., Rossum, G., Tanenbaum, A., Renesse, R. and Staveren, H.: Amoeba: A Distributed Operating System for the 1990s, *IEEE Computer*, Vol. 23, No. 5, pp. 44-53 (1990).
10) Ousterhount, J., Cherenson, A., Douglis, F., Nelson, M. and Welch, B.: The Sprite Network Operating System, *IEEE Computer*, Vol. 21, No. 2, pp. 23-36 (1988).
11) MC88200 Cache/Memory Management Unit User's Manual, Motorola (1990).
12) 新城，清木：並列プログラムを対象とした軽量プロセスの実現方式，情報処理学会論文誌，Vol. 33, No. 1, pp. 64-73 (1992).
13) Shinjo, Y. and Kiyoki, Y.: ReSC: A Distributed Operating System for Parallel and Distributed Applications, *Proc. First Intl. Conf.*

on Parallel and Distributed Information Systems, p. 171 (1991).
14) 新城，清木：データベースの並列処理を支援するオペレーティング・システムの基本機能，情報処理学会研究会報告，89-OS-44, 89-DBS-73 (1989).
15) 田胡，益田：オペレーティング・システムの構造記述に関する一試み，情報処理学会論文誌，Vol. 25, No. 4, pp. 524-534 (1984).
16) 田胡，檜垣，森下：共有メモリ型並列機のためのアクティビティ方式を用いる並列実行環境，情報処理学会論文誌，Vol. 35, No. 2, pp. 229-236 (1991).

(平成4年5月7日受付)
(平成5年1月18日採録)

新城　靖（正会員）
1965年生．1988年筑波大学第三学群情報学類卒業．現在同大学大学院博士課程工学研究科電子・情報工学専攻に在学中．オペレーティング・システム，データベース・システム，並列処理，分散処理に興味を持つ．ACM会員．



清木　康（正会員）
昭和31年生．昭和53年慶応義塾大学工学部電気工学科卒業．昭和58年慶応義塾大学大学院工学研究科博士課程修了．工学博士．同年，日本電信電話公社武蔵野電気通信研究所入所．昭和59年より筑波大学電子・情報工学系に勤務，現在同学系助教授．データベース・システム，計算機アーキテクチャ，関数型プログラミングの研究に従事．日本ソフトウェア科学会，電子情報通信学会，ACM, IEEE 各会員．