並列プログラムを対象とした軽量プロセスの実現方式[†]

新城 靖[††] 清木 康[†††]

この論文は，多重プログラミング環境において，並列プログラムを効率よく実行するための軽量プロセスの実現方式について述べている．軽量プロセスを含む並列プログラムを効率よく実行するためには，軽量プロセスの効率的な実現，および，応用固有のスケジューリングが重要である．この論文では，これらを可能にする軽量プロセスの実現方式として，マイクロプロセスと仮想プロセッサという概念を用いる方式を提案している．マイクロプロセスは，利用者空間内の軽量プロセスであり，カーネル・コールを用いることなく効率的に実現される．その利用者空間にあるスケジューラにより，応用固有のスケジューリングが実現される．仮想プロセッサは，実プロセッサ割当ての単位であり，複数の CPU 処理，および，入出力処理の重ね合せを実現するために用いられる．この論文では，提案方式と他の代表的な軽量プロセス実現方式であるカーネル制御方式，および，コルーチン方式との比較を行っている．さらに，1つの並列応用プログラムについて，提案方式に基づく応用固有のスケジューラの記述例とその効果を示している．これらの実験結果より，提案方式の有効性を明らかにしている．

1. はじめに

最近のオペレーティング・システムの多くは，**軽量プロセス** (lightweight processes) 機能を提供している[2),5),12)]．軽量プロセスとは，従来のプロセスと比較して，プロセスの生成・消滅，プロセス間の同期・通信，および，コンテキスト切替えのオーバヘッドが小さいプロセスである．本論文では，多重プログラミング環境において，軽量プロセスを含む並列プログラムを効率よく実行するための軽量プロセスの実現方式について述べる．

軽量プロセスは，多重プログラミングのオペレーティング・システムにおける並列処理の単位としてのプロセスとして位置付けることができる．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，および，保護の単位として用いられてきた．オペレーティング・システムは，各プロセスごとに資源の管理表を設け，メモリ資源や CPU 資源の公平な分配を行う．さらに，各プロセスに別々のアドレス空間を割り当てることで，他のプロセスの誤動作，あるいは，悪意をもつ利用者による攻撃からプロセスを保護する．一方，並列処理の分野では，プロセスは，並列処理の単位として用いられる．この場合，プロセスは，CPU 割当ての単位であり，CPU 以外の資源割当てや保護の単位ではない．軽量プロセスは，従来のプロセスとは異なる並列処理の単位としてのプロセスである．たとえば，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，軽量プロセスを並列処理の単位として用いることにより，CPU 処理と CPU 処理の重ね合せを，従来のプロセスを用いる方法よりも効率よく実現することが可能となる．また，軽量プロセスは，ウインドウ・サーバやファイル・サーバなど，複数のクライアントを同時に扱う必要があるサーバを実現する際に有効であることが知られている[10)]．この場合，各クライアントごとに軽量プロセスを生成することにより，複数のサーバ・クライアント間の通信処理，入出力処理，および，CPU 処理の重ね合せが実現される．

軽量プロセスの実現における目標は，次のようにまとめられる．

(1) 軽量プロセスの効率よい実行

プロセスの生成・消滅，プロセス間の同期・通信，コンテキスト切替えのオーバヘッドを小さくする．

(2) 並列処理の実現

CPU 処理と CPU 処理の重ね合せ，および，複数の通信処理/入出力処理と CPU 処理の重ね合せを実現する．

以上の目標に加え，われわれは，軽量プロセスのスケジューリングに関して，応用固有のスケジューリングを実現する環境を提供することを目標として設定した．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，応用固有のスケジューリングを行うことにより，著しい性能向上が可能な応用プログラムが存

† An Implementation Method of Lightweight Processes for Parallel Programs by YASUSHI SHINJO (Doctoral Program in Engineering, University of Tsukuba) and YASUSHI KIYOKI (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学工学研究科
††† 筑波大学電子・情報工学系

在する．たとえば，実行に先立ってスケジューリングを行うことにより，同期処理の回数を減らしたり，あるいは，並列性が高くなるように優先順位を決定することができる．

本論文では，上記の目標を達成する軽量プロセスの実現方式について述べる．本方式が対象とするハードウェアは，汎用の共有メモリ型マルチプロセッサ，および，単一プロセッサである．対象とする並列応用プログラムは，CPU 処理，通信処理，および，入出力処理を並列に実行するプログラムである．

本方式の特徴は，従来のプロセスに加えて，マイクロプロセスと仮想プロセッサ[7]という概念を用いて軽量プロセスを実現する点にある．マイクロプロセスは，利用者空間内の軽量プロセスであり，カーネル・コールを用いることなく効率的に実現される．同一の利用者空間内にマイクロプロセスのスケジューラを設けることにより，応用固有のスケジューリングが容易に実現される．応用固有のスケジューリングが必要ない場合には，あらかじめ用意されている汎用的なスケジューリング方式を利用することができる．仮想プロセッサは，カーネルによる実プロセッサ割当ての単位であり，共有メモリ型マルチプロセッサにおいて，複数の CPU 処理を並列に実行するために用いられる．単一プロセッサ上では，1つの CPU 処理，複数の通信処理，および，入出力処理を並列に実行するために用いられる．

応用固有のスケジューリングに関して，既にいくつかの方法が提案されている[2]．1つは，カーネルが軽量プロセスを制御する方式において，利用者がカーネルに，次に実行すべき軽量プロセスのヒントを渡す方法である．もう1つは，プロセッサ・グループという概念を用いる方法である．この方法では，利用者プロセスにおいて，プロセス（タスク），軽量プロセス（スレッド），および，実プロセッサのスケジューリングを行う．この方法では，軽量プロセスをカーネル・レベルで実現しているのに対して，本論文では，軽量プロセスを利用者レベルで実現し，それが属するプロセスの内部に設けたスケジューラにより制御する方法を提案する．

本論文では，提案方式に基づく軽量プロセスの基本的な性能を示す．他の代表的な軽量プロセス実現方式であるカーネル制御方式，および，コルーチン方式との比較を行う．また，1つの並列応用プログラムについて応用固有のスケジューラを記述し，それを利用した並列処理の実験結果を示す．これらの実験結果により，本方式の有効性を明らかにする．

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

代表的な軽量プロセスの実現方式は，次の2種類に分類される．

（1） **カーネル制御方式**

軽量プロセスの生成・消滅，軽量プロセスのコンテキスト切替えは，カーネルにより制御される．軽量プロセス間の同期・通信を，軽量プロセス間の共有領域を用いて実現しているシステムもある．この方式は，Mach システム[2]で用いられている．

（2） **コルーチン方式**

軽量プロセスの生成・消滅，軽量プロセス間の同期・通信などすべての操作は，利用者レベルにおいて実現される．この方式では，CPU 処理を並列に実行することができない．この方式は，SunOS[9] で用いられている．

本章では，マイクロプロセスと仮想プロセッサという概念を用いた軽量プロセスの実現方式について述べる．本方式は，カーネル制御方式と比較して，効率的な軽量プロセスの実現が可能である．また，コルーチン方式と異なり，共有メモリ型マルチプロセッサにおいて CPU 処理を並列に実行することができる．

2.1 プロセス，マイクロプロセス，仮想プロセッサ

プロセスは，資源割当て，および，保護の単位である．これは，1章で述べた従来の多重プログラミング環境におけるプロセスと一致する．資源としては，ファイル資源やメモリ資源のほかに，プロセッサ資源も含まれる．各プロセスは，アドレス空間の壁により保護されている．

マイクロプロセス (microprocess) は，並列処理の単位としてのプロセス，すなわち，軽量プロセスである．各マイクロプロセスは，プロセスの中にあり，独立のプログラム・カウンタとスタックをもっている．プロセスとマイクロプロセスの関係を図1に示す．マイクロプロセスの生成・消滅，マイクロプロセス間の同期・通信など，すべての制御は，利用者レベルにおいて行われる．オペレーティング・システムのカーネルは，一切介在しない．したがって，それらの制御が高速に実行される．この性質を利用して，利用可能なプロセッサ数の変動への適応性を得るために，並列処

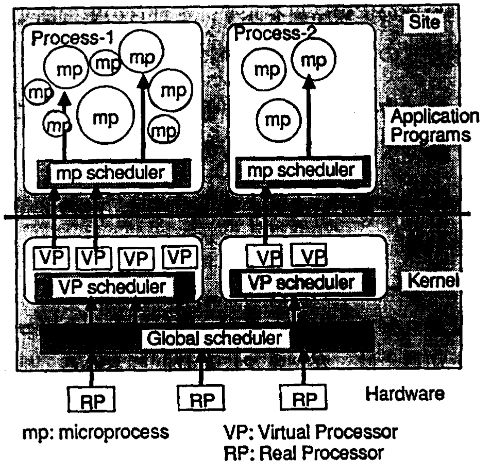

図 1 プロセス，マイクロプロセス，仮想プロセッサの関係
Fig. 1 Processes, microprocesses and virtual processors.

理の単位を小さくすることができる．

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**（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．図1に，プロセスと仮想プロセッサの関係を示す．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している．

仮想プロセッサは，利用者レベルの物理的な並列処理を行うための仕組みである．共有メモリ型マルチプロセッサでは，カーネルは，1つのプロセスに複数の実プロセッサを割り当てる．各実プロセッサは，そのプロセスの仮想プロセッサの中から，実行可能なものを選択し，実行する．その結果，制御が利用者空間へ移動し，複数のマイクロプロセスが並列に実行される．単一プロセッサでは，CPU 処理を並列に実行することはないが，複数の仮想プロセッサを生成することで，1つの CPU 処理，複数の入出力処理，および，複数の通信処理を並列に実行することが可能となる．たとえば，利用者プロセスが，あるマイクロプロセスにおいて入出力処理を行うカーネル・コールを発行した場合，制御は，カーネル空間へ移行する．カーネルは，カーネル・コールを発行した仮想プロセッサを，入出力完了待ち状態に変え，別の実行可能な仮想プロセッサを選択し，コンテキストを切り換える．この結果，制御は，再び利用者空間へ移動し，別の実行可能

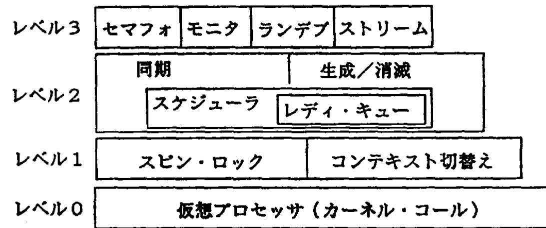

図 2 マイクロプロセス・ライブラリの構造
Fig. 2 The structure of the microprocess library.

なマイクロプロセスが実行される．

プロセス，マイクロプロセス，および，仮想プロセッサは，それぞれのスケジューラにより制御される．詳細については，2.4 節において述べる．

2.2 マイクロプロセスの実現

本方式では，マイクロプロセスは，利用者空間内に実現されるので，カーネル・コールではなくライブラリを利用して構築される．マイクロプロセスの生成・消滅，同期・通信の機能を提供するライブラリを実現した．このライブラリは，**図2**に示すように，層構造になっており，各並列プログラムの要求に応じて利用される．

レベル0では，カーネルによって提供される仮想プロセッサを実現するカーネル・コールを提供する．このレベルでは，各並列プログラムは，仮想プロセッサだけを利用して，独自の方式により軽量プロセスを実現する．レベル1では，各プロセッサ・アーキテクチャやバス・アーキテクチャに独立なコンテキスト切替えとスピン・ロックを実現する機能を提供する．このレベルでは，それらのアーキテクチャに独立したプログラムを記述することができる．レベル2では，基本的なマイクロプロセスの生成・消滅機能と，sleep/wakeup モデルに基づく簡単な同期機能を提供する．このレベルは，スケジューラとレディ・キューを含んでおり，これらは，応用固有のスケジューリングを行うために利用される．詳細については，3章において述べる．レベル3では，セマフォ，モニタ，ランデブ，バイト・ストリーム（UNIX のパイプと同等の機能）等の同期・通信プリミティブを提供する．これらのプリミティブのソース・コードは，各並列プログラム固有の同期・通信プリミティブを構築するプログラマにプロトタイプを与える．

並列プログラムは，通常，レベル2の機能を用いて構築される．これは，セマフォやモニタを利用するよりも，低レベルの機能を利用する方が各並列プログラ

ム固有の同期・通信プリミティブを構築しやすいからである．たとえば，並列応用プログラムには，非決定的な処理を含むものが多数存在する．この非決定性をセマフォやモニタを使って記述することはできない．

2.3 仮想プロセッサを実現するカーネル・コール

本方式では，仮想プロセッサは，カーネルにおいて実現される．**表1**に，仮想プロセッサに関するカーネル・コールを示す．各応用プログラムは，初期化を行ったあと，カーネル・コール vp_allocate() を発行し，並列処理を開始する．実行可能なマイクロプロセスが存在しなくなった場合，vp_sleep() により，実プロセッサをカーネルに返す．実行可能なマイクロプロセスが増加した場合，vp_wakeup() により，実行を中断している仮想プロセッサの実行再開を要求する．(ただし，このカーネル・コールを用いなかったとしても，vp_sleep() を発行した仮想プロセッサは，指定された時間が経過した後には，実行を再開する．) vp_switch() は，スピン・ロックを保持したまま実プロセッサを奪われた仮想プロセッサへコンテキストを切り替えるために用いられる．

本方式では，仮想プロセッサごとに固有のメモリ領域がある．この領域は，マイクロプロセスを実現するライブラリにおいて，仮想プロセッサごとの情報を格納するために利用される．この領域は，プロセッサのレジスタが拡張されたものと考えることができる．

2.4 プロセッサのスケジューリング

本方式では，マイクロプロセス，プロセス，仮想プロセッサの3つのレベルにおいてプロセッサのスケジューリングを行う(図1)．それぞれ，次のような機能をもつ．

(1) マイクロプロセス・スケジューラ

利用者プロセス中にあり，各応用プログラム固有の情報を利用して，応用固有のスケジューリングを実現する．コンテキスト切替えは，マイクロプロセスが同期プリミティブを発行した時に限り行われる．(横取りを行わない (non-preemptive) スケジューリングを行う．)

(2) 大域スケジューラ (global scheduler)

カーネル中にあり，多重プログラミング環境における複数の利用者プロセス間の公平なプロセッサ資源の分配を行う．これを実現するために，量子時間 (time quantum) ごとに，プロセッサの横取り (preemption) を行い，各利用者プロセスに対して割り当てる実プロセッサの数を調整する．

(3) 仮想プロセッサ・スケジューラ

カーネル中にあり，同一プロセス内の別の仮想プロセッサにコンテキストを切り替える．このコンテキスト切替えは，ある仮想プロセッサが入出力待ち，または，通信待ちのために実行を継続できなくなった時に限り行われる．(横取りを行わないスケジューリングを行う．)

このように機能を分割することにより，それぞれのスケジューラが単純化される．一方，スケジューラが多段になり，コンテキスト切換えのオーバヘッドが大きくなることが予想される．しかしながら，スケジューラを実行する頻度がレベル間で異なり，圧倒的にマイクロプロセス・スケジューラの頻度が大きいので，仮想プロセッサ・スケジューラと大域スケジューラは，性能に大きな影響を与えない．

本方式では，仮想プロセッサ・スケジューラにおいてプロセッサの横取りを行わない．理由は，次の2点にある．

(1) マイクロプロセス・スケジューラを活用する．

仮想プロセッサ・スケジューラにおいて，プロセッサの横取りを行うならば，1つの並列プログラムの実行において，2つのスケジューラが同時に動作することになる．その場合には，利用者空間内のマイクロプロセス・スケジューラが機能しなくなる．

(2) 無駄なコンテキスト切替えを避ける．

本来，プロセッサの横取りは，複数のプロセス間において，プロセッサ資源の公平な利用を実現するための機能である．本方式では，資源割当ての単位は，プロセスであるので，同一プロセス内において資源の公平な利用を実現する必要がない．

2.5 他の実現方式との比較

2.5.1 コルーチン方式との比較

コルーチン方式と比較して，本方式の利点は，共有メモリ型マルチプロセッサにおいて CPU 処理の並列

表1 仮想プロセッサに関するカーネル・コール
Table 1 The kernel calls for virtual processors.

vp_allocate (n)	そのプロセスに対して n 個の仮想プロセッサを割り当てる．
vp_sleep (t)	現在の仮想プロセッサの実行を t μ秒間中断する．
vp_wakeup(n)	実行を中断している仮想プロセッサの実行再開を要求する．
vp_switch ()	同一プロセス中の別の仮想プロセッサに制御を切り替える．

実行が可能である点にある．コルーチン方式の利点は，実行時のオーバヘッドが本方式よりもさらに小さい点にある．この理由は，コルーチン方式では，動作が完全に逐次的であるため，軽量プロセス間の共有変数の排他制御を行う必要がないからである．

2.5.2 カーネル制御方式との比較

カーネル制御方式と比較して，本方式の利点は，効率的な実現が可能である点にある．この理由は，本方式では，軽量プロセスをサブルーチンと同等に扱い，保護の機能を設けていないことによる．保護とは，軽量プロセス操作を実現している手続き，または，カーネル・コールの入口において，パラメタのチェックを行うことで，不正な操作から利用者プロセスが破壊されないようにすることである．カーネル制御方式では，パラメタのチェックを省略することができないため，結果として同一プロセス内において保護が行われていることに相当する．また，本方式の方が，軽量プロセスの数に関する制約が緩いという利点がある．本方式では，仮想記憶の許す限り多くの軽量プロセスを生成することができる．

カーネル制御方式の利点は，横取りを行うスケジューリングの実現が容易である点にある．また，複数のプロセスから構成される応用プログラムを記述する場合も，カーネル制御方式が有利である．カーネルが別のプロセス中に直接，軽量プロセスを生成する機能を提供することができるからである．本方式では，プロセス間通信を行い，間接的に軽量プロセスを生成する方法しか存在しない．

本方式における仮想プロセッサは，カーネル制御方式の軽量プロセスと類似している．相違点は，本方式では，軽量プロセスが利用者レベルにおいて実現されていることを前提にして，機能が設計されていることである．たとえば，カーネル制御方式においては，各軽量プロセスに固有領域を設ける必要はない．本方式の仮想プロセッサでは，利用者レベルにおいて軽量プロセスを実現するために，仮想プロセッサに固有領域を設ける必要がある．また，カーネル制御方式の軽量プロセスでは，スケジューリングにおいて優先順位が重要な役割を果たすのに対して，本方式の仮想プロセッサでは，優先順位の概念が存在しない．

プロセッサのアーキテクチャによっては，カーネルにおいて制御を行った方が効率がよい場合がある．たとえば SPARC プロセッサ[9]では，レジスタ・ファイルのフラッシュを特権モードで行う必要があるため，コンテキスト切替えにおいて必ずカーネル・コールが必要となる．この場合，カーネルにおいてコンテキスト切替えを制御する方が効率が良いことが予想される．

3. 応用固有のスケジューリング

本方式では，応用固有のスケジューラを，マイクロプロセス・スケジューラとして，利用者空間内に構築する．この方法の利点は，以下のとおりである．

（1） 応用固有の情報の入手が容易である．プロセス間の保護の壁を越えることなく，実行の状況を調べることができる．

（2） カーネル中にスケジューラを設ける方法と比較して，スケジューラを取り替えることが容易である．

（3） 同時に動作する複数の並列応用プログラムのスケジューラが互いに干渉しあうことがない．

3.1 スケジューラ記述の支援

2.2 節において述べたマイクロプロセス・ライブラリを利用する場合，各並列プログラムのプログラマは，応用固有のマイクロプロセス・スケジューラを構築するために，次の5レベルの機能を利用することができる．高いレベルの機能を利用するほど，細かいスケジューリングの記述から開放される．一方，低いレベルの機能を利用するほど，きめ細かい制御を行うことができる．

（レベル4） あらかじめ用意されているスケジューラから選択して利用する．（現在，代表的なスケジューリング方式として，FIFO，および，固定優先順位方式を用意している．）

（レベル3） マイクロプロセス・ライブラリ中のレディ・キュー・モジュール（図2）と同じ外部仕様をもつモジュールを記述し，残りのマイクロプロセス・ライブラリと結合して利用する．（4.3.1 項において，このレベルを用いて記述した優先順位式スケジューラについて述べる．）

（レベル2） 2.2 節で述べた sleep/wakeup モデルに基づく同期プリミティブと同じレベルに新たなプリミティブを追加する．たとえば，sleep_and_wakeup のように，複合的なプリミティブを追加する．

（レベル1） マイクロプロセス・ライブラリ中のスケジューラ・モジュール（図2）と同じ外部仕様をもつモジュールを記述し，残りのマイクロプロセス・ライブラリと結合して利用する．

(レベル0) 2.3 節において述べたカーネルの機能だけを利用して，スケジューラを記述する．

4. 実　験

本方式と他の軽量プロセス実現方式の基本的なプロセス操作の性能を比較する実験を行った．基本的なプロセス操作としては，プロセスの生成・消滅，コンテキスト切替えを伴わないプロセス間の同期，コンテキスト切替えを伴うプロセス間の同期をとり上げた．さらに，1つの並列応用プログラムについて応用固有のスケジューラを記述し，その効果を調べる実験を行った．これにより，本方式において応用固有のスケジューラの記述が可能であることを示す．

4.1 実験環境

プロセッサのアーキテクチャ，および，個々のオペレーティング・システムの影響を排除して性能を論じるために，次の5とおりの環境において実験を行った．実験に用いたシステムのプロセッサのアーキテクチャ，オペレーティング・システム，主記憶容量，キャッシュ容量（制御方式）を以下に示す．

共有メモリ型マルチプロセッサ

(1) Sequent Balance 8000, Dynix 2.1, 4× NS 32032 10 M Hz, 8 M Bytes, 8 k Bytes (write-through)[1]

(2) Omron Luna88 k, Mach 2.5 (UniOS-Mach 1.10), 4×M88100 25 M Hz, 32 M Bytes, 32 k Bytes (copy-back)[11]

単一プロセッサ

(3) SPARCstation 2, SunOS 4.1.1, SPARC 40 MHz, 32 M Bytes, 64 k Bytes[9]

(4) Sun3/60, SunOS 4.0.3, M68020 16 MHz, 12 M Bytes[9]

(5) NeXT (cube), Mach 2.5, M68030 25MHz, 8 M Bytes, on-chip[6]

(1)と(2)は，共に4プロセッサ構成の共有メモリ型マルチプロセッサであるが，プロセッサのアーキテクチャが大きく異なる．(1)は，CISC (Complex Instruction Set Computer)系のプロセッサを，(2)は，RISC (Reduced Instruction Set Computer) 系のプロセッサを備えている．RISC の場合，CISC と比較してキャッシュが性能に大きく影響すると予想される．

(3)と(4)では，同一のオペレーティング・システムが動作しているが，プロセッサのアーキテクチャが大きく異なる．(3)は，RISC系のプロセッサを，(4)は，CISC 系のプロセッサを備えている．特に(3)の SPARC プロセッサは，巨大なレジスタ・ファイルを持っていることから，軽量プロセスのコンテキスト切替えの処理時間が長くなることが予想される．SPARC プロセッサでは，手続き1個あたり 16 個のレジスタを局所変数として利用することができる．コンテキスト切換えにおいては，本来，利用しているレジスタのみを保存／回復すれば十分である．しかしながら，レジスタの利用状況を得る手段がないため，すべてのレジスタを保存／回復しなければならない．さらに，レジスタ・ファイルのフラッシュを特権モードで行う必要があるため，コンテキスト切換えにおいて必ずカーネル・コールが必要となる．

(5)と(2)では，同一のオペレーティング・システムが動作しているが，プロセッサのアーキテクチャと数が異なる．

今回の実験では，仮想プロセッサに関するカーネル・コールについては，vp_allocate() を UNIX の fork() システム・コールを用いて，vp_sleep() を usleep() ライブラリ関数を用いて実現した．ここであげる実行時間は，UNIX のシステム・コール gettimeofdate() を用いて測定したものである．実行時間の精度が異なるのは，各システムごとに測定可能な最小時間が異なることによる．マルチユーザ・モードにおいて，システムの負荷が低い状態で実験を行った．

4.2 他の軽量プロセスとの比較

比較対象としては，カーネル制御方式である Mach の C-Threads[2]，および，コルーチン方式である SunOS の SunLWP[9] を取り上げた．C-Threads では，軽量プロセスの生成・消滅，および，コンテキスト切替えには，カーネル・コールが必要である．コンテキスト切替えを伴わない同期は，カーネル・コールを用いることなく実現される．C-Threads では，軽量プロセスのキャッシングを行っている．すなわち，終了した軽量プロセスを破壊せずにそのまま保存しておき，次に生成要求を受け付けた時に再利用している．SunLWP は，SunOS 4.x に付属している軽量プロセス・ライブラリであり，コルーチン方式により実現されている．軽量プロセスのスタックをレッド・ゾーン方式により保護する機能がある．この機能は，システム・

コール mprotect() により実現されている．SunLWP では，そのスタックをキャッシングする機能がある．SPARCstation 2 では，コンテキスト切換えにおいて，レジスタ・ウインドウをフラッシュするために，カーネル・コールを発行する．本方式においても，そのカーネル・コールを利用した．

なお，Balance 8000 では，他の軽量プロセスを利用することができなかったので，比較を行わなかった．

4.2.1 プロセスの生成・消滅

2重ループ構造をもつプログラムを用いて，プロセスの生成・消滅操作の性能を測定した．内側のループでは，プロセスの入れ子を作り，共有メモリ型マルチプロセッサにおける並列処理の効果，および，軽量プロセスやスタックのキャッシングの効果を抑えている．逆に，外側のループの実行回数を変化させることで，キャッシングの効果を調べることができる．このプログラムを用いて，プロセスの生成処理，コンテキスト切替え2回(親から子へ，子から親へ)，同期処理1回(子の終了待ち)，消滅処理の全体の処理時間を測定した．

表 2(a)は，最初にプロセスを生成する時のプロセス1個に関する実行時間，表 2(b)は，2回目以降のプロセス1個に関する実行時間を示す．表2において，繰返し回数が $m \times n$ とは，外側のループを m 回，内側のループを n 回実行したことを意味する．表2(b)における実行時間は，次の式により計算した．

$$\frac{(m+1)\times n \text{ 回の実行時間} - 1\times n \text{ 回の実行時間}}{m\times n}$$

この実験では，あらかじめメモリを参照することで，測定時間に仮想記憶の処理時間が含まれないようにした．さらに，仮想記憶の影響により実行時間が長くならない範囲内において，生成する軽量プロセスの数を大きくした．(本方式では，軽量プロセスの数の上限は，利用者プロセスの仮想アドレス空間の大きさによってのみ制約される．)

表 2(a)からわかるように，スタックの保護を行った SunLWP (mprot) と比較すると，本実現 (Microprocess) が 10 倍から 20 倍程度高速になっている．主な原因は，SunLWP では，レッド・ゾーン方式によ

表 2 プロセスの生成・消滅操作の実行時間の比較
Table 2 Comparisons of process creation and termination execution times.

(a) 最初にプロセスを生成する場合のプロセスの生成・消滅操作の実行時間
上段：プロセス1個当りの実行時間(単位：ミリ秒)，下段：繰返し回数(単位：回)
(a) Execution times of process creation and termination in the first creation of processes. (In milliseconds) (The numbers of iterations.)

Machine\Method	Microprocess	SunLWP(mprot)	SunLWP(none)	C-Threads
SPARCstation 2	0.149 (1×1k)	0.97 (1×100)	0.43 (1×100)	— —
Sun3	0.400 (1×500)	3.2 (1×100)	1.0 (1×100)	— —
NeXT	0.444 (1×500)	— —	1.03 (1×100)	10.0 (1×10)
Luna88k	0.19 (1×1k)	— —	— —	10.5 (1×20)

(b) 2回目以降のプロセスの生成・消滅操作の実行時間
上段：プロセス1個当りの実行時間(単位：ミリ秒)，下段：繰返し回数(単位：回)
(b) Execution times of process creation and termination in the second creation of processes. (In milliseconds) (The numbers of iterations.)

Machine\Method	Microprocess	SunLWP(mprot)	SunLWP(none)	C-Threads
SPARCstation 2	0.149 6 ([11−1]×1k)	0.443 ([11−1]×100)	0.339 ([11−1]×100)	— —
Sun3	0.384 ([11−1]×500)	1.46 ([11−1]×200)	0.98 ([11−1]×100)	— —
NeXT	0.362 ([11−1]×500)	— —	1.01 ([11−1]×100)	0.147 ([11−1]×10)
Luna88k	0.130 ([11−1]×1k)	— —	— —	1.9 ([11−1]×20)

りスタックの保護を行っているためである．SunLWPにおいて，スタックの保護機能を外したとしても（表2のSunLWP (none)），本実現は，約3倍高速になっている．この差は，軽量プロセスの実現方式の違いによるものではなく，両実現の機能の違いによる．たとえば，SunLWPでは，UNIX固有のソフトウェア割込みや例外を扱う機能があり，本マイクロプロセスよりも複雑になっている．軽量プロセスの実現方式の違いだけならば，2.5.1項で述べたように，コルーチン方式のSunLWPの方の効率がよくなる．

C-Threadsと比較すると，キャッシングが有効でない場合（表2(a)），本実現が約20から50倍高速になっている．この差は，軽量プロセスの実現方式の違いによる．本方式では，カーネル・コールのオーバヘッドがなく，必要となるデータ構造をすべて利用者空間に置くことができるので，高速になっている．

また，表2(a)と表2(b)を比較すると，C-Threads，および，SunLWP (mprot)において，2回目以降の実行時間が1回目と比較して著しく短縮していることがわかる．これは，C-Threadsでは，軽量プロセスのキャッシング，SunLWP (mprot)では，スタックのキャッシングが有効に働いているからである．本実現では，軽量プロセスやスタックのキャッシングを行っていないが，メモリのキャッシュの効果により，処理時間が短くなっている．

4.2.2 コンテキスト切替えを伴わないプロセス間の同期

表3は，モニタ，あるいは，ロックを用いて測定したコンテキスト切替えを伴わないプロセス間の同期操作の実行時間である．測定に用いたプログラムでは，1つのプロセスにおいて，1つのモニタ，あるいは，ロックに対して入口操作／出口操作，あるいは，ロック操作／アンロック操作を繰り返し行っている．

SunLWPと比較すると，SPARCstation 2では，本実現が5倍程度遅くなっているのに対して，逆に，Sun3とNeXTでは，本実現が25%程度高速になっている．この差は，4.2.1項のSunLWP (none)との比較において生じた差と同じく，軽量プロセスの実現方式の違いによるものではなく，両者の機能の違いによる．

C-Threadsと比較すると，本実現の効率が悪くなっている．この差は，軽量プロセスの実現方式の違いによるものではなく，両実現の機能の違いによる．C-Threadsでは，この操作をカーネル・コールを用いることなく実現しているので，たとえば，C-Threadsのロックを，Machのカーネル・コールの代りに本方式のマイクロプロセス・ライブラリを利用して実現したとしても，全く同じ実行時間になる．逆に本実現のモニタを，Machのカーネル・コールを用いて実現したとしても，全く同じ実行時間になる．

表3 プロセス間の同期処理の実行時間の比較（コンテキスト切替えなし）
上段：モニタ入口／出口，ロック／アンロックの実行時間（単位：μ秒），下段：繰返し回数（単位：回）

Table 3 Comparisons of execution times of inter-process synchronization without context switches.
Execution times of monitor entry/exit or lock/unlock operations. (In microseconds) (The numbers of iterations.)

\Method Machine	Microprocess	SunLWP	C-Threads
SPARC-station 2	27.1 (100k)	5.6 (100k)	— —
Sun3	31.2 (100k)	40.8 (100k)	— —
NeXT	29.9 (10k)	39.6 (10k)	4.30 (100k)
Luna88k	8.3 (100k)	— —	2.1 (100k)

4.2.3 コンテキスト切替えを伴うプロセス間の同期

表4に，コンテキスト切替えを伴うプロセス間の同期処理の比較を示す．実験には，2つのプロセス，1つのモニタ（またはロック），2つの条件変数を用いたプログラムを使用した．

表4より，本実現は，C-Threadsよりもコンテキスト切替えのオーバヘッドが小さいことがわかる．この差は，軽量プロセスの実現方式の違いによる．カーネル制御方式のC-Threadsでは，コンテキスト切替えが必要となる場合，必ずカーネル・コールを発行しなければならないのに対して，本方式のマイクロプロセスでは，その必要がない．

SunLWPと比較すると，本実現の効率が悪くなっている．この差は，軽量プロセス実現方式の違いによる．本方式では，共有メモリ型マルチプロセッサ上の並列処理を行うので，細かい単位で共有変数の排他制御を行っている．また，Sun3と比較してSPARCstation 2における効率の低下が著しい．理由は，この操作において手続き呼出しの深度が深く，その結果，レ

表 4 プロセス間の同期処理の実行時間の比較（コンテキスト切替えあり）
上段：条件変数操作 2 回の実行時間(単位：μ 秒)，
下段：繰返し回数(単位：回)
Table 4 Comparisons of execution times of interprocess synchronization with context switches.
Execution times of condition variable operations. (In microseconds) (The numbers of iterations.)

\Method Machine	Microprocess	SunLWP	C-Threads
SPARC-station 2	265.8 (10k)	80.2 (10k)	— —
Sun3	322 (1k)	236 (1k)	— —
NeXT	305 (1k)	231 (1k)	1146 (1k)
Luna88k	66 (10k)	— —	170 (1k)

ジスタ・ファイルのフラッシュの処理時間が長くなっているからである．

4.3 応用固有のスケジューラの記述

データベースの並列処理[3),4)]を例に，3.1 節で述べた機能を利用して，応用固有のスケジューラを記述してみた．さらに，そのスケジューラの効果を調べる実験を行った．

4.3.1 スケジューラの記述

次のような 3 種類のスケジューラを記述してみた．

(a) 優先順位を用いない．実行可能になった段階でレディ・キューに入れる．

(b) 確実に必要な計算を行うマイクロプロセスに高い優先順位を与える．それ以外のマイクロプロセスも，実行可能になった段階で低い優先順位を与え，レディ・キューに入れる．

(c) 確実に必要な計算を行うマイクロプロセスだけをレディ・キューに入れる．

スケジューラ(a)，および，(b)では，結果的に無駄となる計算が行われることがある．

ここで(a)を，あらかじめ用意してあるFIFOスケジューラを用いて実現した．(3.1節で述べたレベル 4 に対応する．) また，(b)，(c)を，図 2 におけるレディ・キュー・モジュールを入れ替えることで実現した．(3.1 節で述べたレベル 3 に対応する．) (b)では，マイクロプロセスの優先順位は，動的に決定される．これは，実時間システムで用いられる優先順位継承方式[8)]の単純な場合になっている．(c)の実現を，(b)において，優先順位が上がるまでマイクロプロセスの実行を遅延することにより行った．

表 5 並列応用プログラム（データベース処理）の実行時間(単位：秒)
Table 5 Execution times of a parallel application program. (database processing) (In seconds).

	Balance 8000			Luna88k		
# of VPs	(a)	(b)	(c)	(a)	(b)	(c)
1	40	38	38	4.5	3.7	3.7
2	22	21	26	3.2	2.9	2.7
3	18	16	19	3.0	3.0	2.9

4.3.2 実験結果

表 5 は，8 個の軽量プロセスより構成された並列応用プログラムの実行結果である．表の結果より，優先順位を用いないスケジューラ(a)と比較して，優先順位を用いるスケジューラである(b)，または(c)の性能がよいことがわかる．特に，利用可能なプロセッサの数が少ない場合において，優先順位を用いるスケジューリングの効果が大きいことがわかる．(この実験では，他にプロセスが存在しないので，仮想プロセッサの数と利用可能な実プロセッサの数は，一致している．) これは，動的に優先順位を決定することにより，無駄な計算を避けることができたからである．

以上のことから，本方式において，今回利用した並列応用プログラムについて，応用固有のスケジューラが容易に記述できること，および，そのスケジューラにより効率が改善されることが示された．

5. おわりに

本論文では，マイクロプロセスと仮想プロセッサの概念に基づく軽量プロセスの実現方式について述べた．本方式では，利用者レベルにおいて軽量プロセスを実現することにより，高い性能を得ることができる．さらに，利用者プロセス内の軽量プロセスのスケジューラにより，応用固有のスケジューリングが実現される．本方式の軽量プロセスと，カーネル制御方式，および，コルーチン方式の軽量プロセスの基本的な性能を比較する実験，および，1 つの並列応用プログラムについて応用固有のスケジューラの記述とその性能を調べる実験を行い，本方式の有効性を示した．

今後の課題は，言語処理系の利用について，本実現

方式の有効性を確かめることである．たとえば，同期処理やコンテキスト切替え処理をインライン展開することや，コンテキスト切替え時に待避するレジスタの数を削減することが考えられる．さらに，ネットワーク通信機能と軽量プロセス機能の連係について検討を進めていく予定である．

参考文献

1) Balance 8000 Parallel Programming, Sequent Computer Systems, Inc. (1985).
2) Black, D.: Scheduling Support for Concurrency and Parallelism in the Mach Operating System, *IEEE Comput.*, Vol. 23, No. 5, pp. 35-43 (1990).
3) Kiyoki, Y., Kato, K. and Masuda, T.: A Relational Database Machine Based on Functional Programming Concepts, *Proc. ACM-IEEE Computer Society Fall Joint Computer Conf.*, pp. 969-978 (1986).
4) Kiyoki, Y., Kurosawa, T., Kato, K. and Masuda, T.: The Software Architecture of a Parallel Processing System for Advanced Database Applications, *Proc. 7th IEEE Conf. on Data Engineering*, pp. 220-229 (1991).
5) Mullender, S., Rossum, G., Tanenbaum, A., Renesse, R. and Staveren, H.: Amoeba: A Distributed Operating System for the 1990s, *IEEE Comput.*, Vol. 23, No. 5, pp. 44-53 (1990).
6) The NeXT System Reference Manual, NeXT, Inc. (1988).
7) 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).
8) Sha, L., Rajkumar, R. and Lehoczky, J.: Priority Inheritance Protocols: An Approach to Real-Time Synchronizaion, *IEEE Trans. Comput.*, Vol. 39, No. 9, pp. 1175-1185 (1990).
9) SunOS Reference Manual, Sun Microsystems, Inc. (1988).
10) Tanenbaum, A. and Renesse, R.: Distributed Operating Systems, *ACM Comput. Surv.*, Vol. 7, No. 4, pp. 419-470 (1985).
11) UniOS-Mach Reference Manual, オムロン (1990).
12) Weiser, M., Demers, A. and Hauser, C.: The Portable Common Runtime Approach to Interoperability, SOSP 12, *ACM Operat. Sys. Rev.*, Vol. 23, No. 5, pp. 114-122 (1989).

(平成3年6月25日受付)
(平成3年11月5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年慶應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．