計算機アーキテクチャ　104－5
システムソフトウェアと
オペレーティング・システム　62－5
（１９９４．　１．　２７）

軽量プロセスを利用する応用プログラムの開発と性能評価の支援

新城 靖
（琉球大学　情報工学科）
〒903-01　沖縄県西原町千原1番地
電話：098-985-2221　内線3266
Ｆａｘ：098-985-2688
電子メール：yas@ie.u-ryukyu.ac.jp

清木 康
（筑波大学　電子・情報工学系）
〒305　茨城県　つくば市
電話：0298-53-5163
Ｆａｘ：0298-53-5206
電子メール：kiyoki@is.tsukuba.ac.jp

概要　この論文は、軽量プロセス間の同期・通信プリミティブの開発の支援、および、各軽量プロセスの実行時間測定の支援について述べている。軽量プロセスを用いる並列応用プログラムは、しばしばそれぞれ固有の同期・通信プリミティブを利用する。また、性能の改善のため、各軽量プロセスの動きを知る必要がある。この論文は、マイクロプロセス／仮想プロセッサ・モデルに基づく軽量プロセス・ライブラリにおける、それらを支援する機能について述べている。マイクロプロセスとは、利用者レベルの軽量プロセスであり、４層からなるライブラリにより実現される。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．各並列応用プログラムの開発者は、層構造を持つライブラリから必要な部分だけを選択的に利用し、独自の同期・通信プリミティブを開発する。さらに、コルーチンの性質を利用して、効率的な同期・通信プリミティブの開発が可能となる。そのライブラリには、軽量プロセスの実行の軌跡を記録する機能がある。その軌跡を利用することで、並列処理の様子を視覚化することができる。さらに、軽量プロセスごとにＣＰＵ時間、入出力時間の統計を取る道具が用意されている。

Supporting Development of Lightweight Process Applications and their Performance Evaluation

Yasushi Shinjo
Department of Information Engineering
University of the Ryukyus
Nishihara, Okinawa 903-01, Japan
Phone:　+81 98 895 2221 Ext.3266
Fax:　+81 98 895 2688
E-Mail:　yas@ie.u-ryukyu.ac.jp

Yasushi Kiyoki
Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba, Ibaraki 305, Japan
Phone:　+81 298 53 5163
Fax:　+81 298 53 5206
E-Mail:　kiyoki@is.tsukuba.ac.jp

Abstract　This paper presents system support for developing primitives of inter-lightweight process communication and synchronization and for measuring execution times of lightweight processes. A parallel application with lightweight processes often uses its own specialized primitives of communication and synchronization, and needs activities of lightweight processes for improving performace. This paper describes system facilities supporting such applications in a lightweight process library based on the Microprocess/Virtual Processor model. Microprocesses are user-level lightweight processes and implemented by the user-level library which consists of four layers. Virtual processors are entries of real processors which are allocated to a user process by the kernel. Each developer of a parallel application program can choose some parts in the layered library and implements its own specialized primitives of communication and synchronization. Furthermore, the characteristics of coroutines are effectively used to implement the primitives. The microprocess library has a facility to record traces of lightweight process activities. These traces are used to visualize the parallelism in an application program. There is a tool which takes statistics from traces.

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、必要不可欠な機能である。軽量プロセスとは、資源割当てと保護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。資源割当てと保護の単位としてのプロセスと比較して、軽量プロセスは、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。この論文では、軽量プロセス間の同期・通信プリミティブの開発の支援、および、各軽量プロセスの実行時間測定の支援について述べる。

軽量プロセス間の同期・通信プリミティブの開発にあたって、システムと並列応用プログラムが競合する (conflict)ことがある。並列応用プログラムは、システムが提供している同期・通信プリミティブではなく、それぞれ応用固有の同期・通信プリミティブを利用することが多い。その第1の理由は、それらの機能が言語処理系の機能として実現されることが多いからである。特に単一プログラミング・システムにおいて開発されてきたものは、そうである。また、全ての並列応用プログラムが必要とするプリミティブを予めシステムが用意することは、不可能である。各並列応用プログラムがシステム提供のプリミティブを利用して独自のプリミティブを構築する場合、以下のような問題がある。

- システムが提供するプリミティブは、たとえばC言語から直接利用されることを想定して設計されている。そのため、必ずしも応用独自のプリミティブを構築するために適していない。
- システムが提供するプリミティブは、汎用であり、しばしば複雑なオプションを伴う。並列応用プログラムにとっては不要の機能のために、効率が低下する可能性がある。

逐次応用プログラムと比較して並列応用プログラムの目的は、高速処理を実現することである。そのため、各軽量プロセスの動きを知り、性能の改善に利用することが重要となる。具体的には、次のようなことを行う。

- 軽量プロセスごとの実行時間を調べることで、プログラムのどの部分にボトルネックが存在しているかを調べる。そして、重たい処理をさらに複数の軽量プロセスに分割することを検討する。
- 軽量プロセス間の同期・通信の頻度を調べる。そして、処理単位の粒度を調整するなど、同期・通信、コンテキスト切替えのオーバヘッドを軽減することを検討する。
- 複数の軽量プロセスの実行の重なりを調べる。これにより、スケジューリングの方針を検討する。

ＵＮＩＸでは、軽量プロセスを用いない逐次プログラムに対して profil システム・コール、prof コマンド、gprof コマンドなどが利用可能である [4]。しかしながら、軽量プロセスを用いるプログラムに対しては、それらのコマンドを直接利用することができない。その理由は、第1に、それらのコマンドが手続き呼び出しにのみ対応しており、軽量プロセスを認識できないことによる。たとえば、複数の軽量プロセスが同一の手続きを呼んだとしても、それらを別々に集計することができない。第2に、それらのコマンドは、コンテキスト切替えに対応していない。たとえば、ある手続きにおいて別の軽量プロセスにコンテキストを切り替えた場合、きちんと統計を取ることができない。したがって、軽量プロセスに対する支援が必要である。

我々は、軽量プロセスの実現において、システムと並列応用プログラムの間の競合を解消し、それらの間の調和 (harmonization) を実現するような方式を提案してきた [11][12][13]。この方式では、マイクロプロセスと仮想プロセッサという概念を用いて軽量プロセスを実現する。マイクロプロセスとは、利用者レベルの軽量プロセスでり、層構造を持つライブラリにより実現される。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。すでに、文献 [11] では、基本的なモデル、および、応用固有のスケジューリングの支援について述べた。文献 [12] では、仮想プロセッサを提供するカーネルの構成方式、および、仮想プロセッサごとの固有領域の利用と実現について述べた。この論文では、軽量プロセス間の同期・通信プリミティブの開発の支援、および、各軽量プロセスの実行時間測定の支援について述べる。

マイクロプロセスを実現するライブラリの特徴は、層構造を持っている点にある。各応用プログラムの開発者は、それから必要な部分だけを選択的に利用し、独自の同期・通信プリミティブを開発する。さらに、コルーチンの性質を利用することで、効率的な開発が可能となる。そのライブラリには、軽量プロセスの実行の軌跡を保存する機能がある。その軌跡を利用することで、並列処理の様子を視覚化することができる。さらに、軽量プロセスごとにＣＰＵ時間、入出力時間の統計を取ることができる。

2章では、先に提案した軽量プロセス実現方式の概要について述べる。3章では、軽量プロセス間の同期・通信プリミティブの開発について述べる。4章では、軽量プロセスの実行時間の利用とそれを記録するための仕組みについて述べる。最後にまとめを行う。

2　マイクロプロセスと仮想プロセッサ

我々は、軽量プロセスを多重プログラミング環境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、（重量）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、（重量）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。

我々は、マイクロプロセスと仮想プロセッサという概念を用いて軽量プロセスを実現する。**マイクロプロセス**（microprocess）は、利用者レベルの軽量プロセスであり、資源割当てと保護の単位としての（重量）プロセスの中にある。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・通信など全ての操作は、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、それらの操作が高速に実行される。

マイクロプロセスは、カーネルにより提供される**仮想プロセッサ**（virtual processor）により実行される。仮想プロセッサは、１つの（重量）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。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。共有メモリ型マルチプロセッサでは、カーネルは、１つの利用者プロセスに対応している複数の仮想プロセッサに実プロセッサを割り当てる。その結果、複数のマイクロプロセスがそれらの仮想プロセッサにより並列に実行される。

マイクロプロセスは、コルーチンによる軽量プロセスの実現と類似している。コルーチンと比較して、マイクロプロセスの利点は、共有メモリ型マルチプロセッサにおいてＣＰＵ処理の並列実行が可能である点にある。

プロセスが生成される時には、自動的に１個の仮想プロセッサが生成され、割り当てられる。逐次応用プログラムは、そのまま単一仮想プロセッサにより処理を進める。これにより、従来の逐次応用プログラムを仮想プロセッサに関して全く変更することなく共有メモリ型マルチプロセッサ上で利用可能となる。一方、並列応用プログラムは、カーネル・コール vp_allocate(n) を発行して、複数の仮想プロセッサを要求する。カーネルは、新たに n-1 個の仮想プロセッサを生成する。そして、合計 n 個の仮想プロセッサにより、並列応用プログラムを実行する。

２．１　マイクロプロセスを実現するライブラリ

マイクロプロセスは、利用者レベルで実現されるので、カーネル・コールではなくライブラリにより支援される。このライブラリは、４層構造になっており、各並列応用プログラムの要求に応じた部分的な利用が容易になっている（図１）。

第０層では、カーネルによって実現される仮想プロセッサに関連したカーネル・コールが提供される。この層を利用する並列応用プログラムは、仮想プロセッサだけを利用して、完全に独自の方式により軽量プロセスを実現することになる。

第１層では、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が提供される。この層を利用するプログラムは、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可能となる。

第２層では、基本的なマイクロプロセスの生成・消滅機能（mp_create ()・mp_exit()）と、sleep/wakeup モデルに基づく簡単な同期機能（２．２節）が提供される。また、第２層は、スケジューラとレディ・キューを含んおり、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 [11]。

第３層では、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・通信プリミティブが提供される。これらのプリミティブのソース・コードは、各並列応用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よいプロトタイプを与える。

並列応用プログラムは、通常、第２層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、低いレベルの機能を利用する方が応用固有の同期・通信プリミティブを構築しやすいからである。さらに、第２層を利用することで、４章で述べる軽量プロセスの実行の軌跡を記録するサービスを受けることができる。

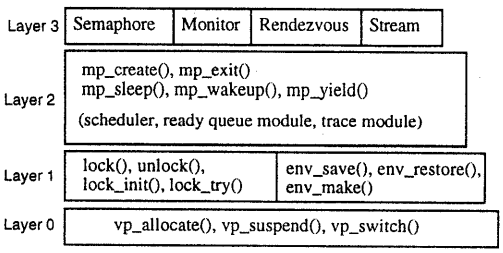

図１　マイクロプロセス・ライブラリの構造

Figure 1: The structure of the Microprocess library.

２．２　sleep/wakeup モデル

本ライブラリが提供する sleep/wakeup モデルでは、主に次の２つの手続きが利用される。

(1) **mp_sleep()**: 実行中状態（running state）の自分

自身（マイクロプロセス）を、待ち状態(blocked state)に移す。

(2) **mp_wakeup()**: 待ち状態にある（他の）マイクロプロセスを、実行可能状態(runnable state、ready state)に移す。既に実行可能、あるいは、実行中の場合には、何もしない。

この他に、自分自身を実行可能状態にしたまま、他の実行可能なマイクロプロセスに制御を移すための手続き、**mp_yield()** がある。

マイクロプロセスが mp_sleep() を実行すると、その仮想プロセッサは、アイドル状態になる。アイドル状態になったプロセッサは、利用者空間内のマイクロプロセス・スケジューラに制御を移す。そこで、実行可能状態にあるマイクロプロセスを選び、それを実行中状態に変え、そのマイクロプロセスへ制御を移す。

3　軽量プロセス間同期・通信プリミティブの開発

3．1　層構造を持つライブラリの利用

2．1節で述べたように、本マイクロプロセス・ライブラリは、層構造を持っている。本ライブラリを利用する場合、各並列プログラムの開発者（あるいは、並列言語処理系の開発者）は、応用固有の同期・通信プリミティブを構築するために、たとえば、以下で示すようなレベルで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、高い性能を得ることができる。

レベル4：第3層の同期・通信プリミティブから選択して利用する。この場合、複数のプリミティブを混在させて利用することも可能である。

レベル3：第3層の同期・通信プリミティブを利用して、独自の同期・通信プリミティブを開発する。

レベル2：第2層の sleep/wakeup モデルに基づく同期・通信プリミティブを利用して、独自のプリミティブを開発する。

レベル1：第1層のスピンロック・ライブラリとコンテキストの保存・回復の機能を用いて、独自のプリミティブを開発する。このレベルでは、アーキテクチャから独立したプログラムを開発することができる。

レベル0：第0層のカーネル・コールだけを利用して同期・通信プリミティブを開発する。

3．2　コルーチンの性質の利用

マイクロプロセス間の同期・通信プリミティブは、応用プログラムごとに開発される。本方式では、コルーチンの性質を活用して、応用固有の同期・通信プリミティブを効率的に開発することができる。

2章で述べたように、マイクロプロセスは、コルーチンと非常に類似している。相違点は、マイクロプロセスの場合、並列処理の対象となる点にある。そのためマイクロプロセスの場合、軽量プロセス間共有変数の相互排除を行う必要がある。逆に、マイクロプロセスにおいて、相互排除操作を削除することにより、完全にコルーチンに還元することも可能である。

並列プログラムの開発・デバッグにおいて、実行の再現性がないことが大きな障害となっている。対照的に、コルーチンの場合、プログラムの実行に再現性がある。これは、プログラムが逐次的に実行されるからである。この再現性があるという性質は、プログラムのデバッグにおいて、非常に都合がよい。本システムでは、1個の仮想プロセッサを用いることにより、容易にコルーチンと同じ動作を行わせることができる。すなわち、カーネル・コール vp_allocate() の引数として1を指定するか、または、そのカーネル・コールを発行しないことにより、完全に逐次的な動作を行わせることができる。

以上のことより、本マイクロプロセス方式において、新たにマイクロプロセス間の同期・通信プリミティブを開発する場合、次の様な手順で効率的に行うことができる（図2）。

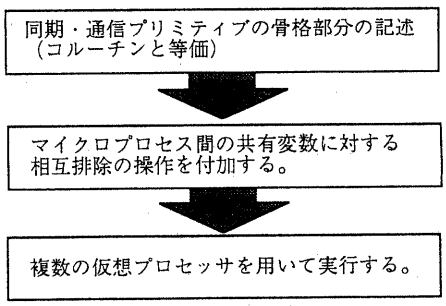

図2　コルーチンの性質を利用した軽量プロセス間同期・通信プリミティブの開発

ステップ1：同期・通信プリミティブの骨格部分（論理的な動作に関係した部分）についてのみ記述し、デバッグを行う。この段階では、マイクロプロセス間の共有変数に対する相互排除操作が含まれていない。すなわち、コルーチンと完全に等価である。この段階では、完全に逐次的に動作するため、開発が容易である。

ステップ2：ステップ1で開発したものに、マイクロプロセス間の共有変数に対する相互排除の操作を付加する。そしてこれを、1つの仮想プロセッサを用いて実行する。この時に、ロックの解除を忘れる、あるいは、1つのマイクロプロセスで同じ変数に2回ロックを試みるなどのバグが取除かれる。この段階でも、完全に逐次的に動作するため、開発が容易である。この時点で、実行のタイミングに関するバグを除いて、ほとんどのバグが取り除かれる。

ステップ3：最後に、複数の仮想プロセッサを用い

て実行し、タイミングに関するバグを取る。

ステップ2において、ロックを実現するライブラリ手続きとして、デバッグ用の特殊なものを用意している。それは、既にロックされている変数に対して再びロックを試みた場合に、メッセージを表示して直ちに処理を終了するものである。1個の仮想プロセッサにおいては、ロックが重なることは、デッドロックを意味する。

3．2．1　2進セマフォの例

ここでは、典型的な同期プリミティブとして、2進セマフォを取り上げ、同期・通信プリミティブの開発手順の例を示す。図3に、完成した2進セマフォを示す。

```
 1: extern private struct mpcb *current ;
 2:
 3: struct sem
 4: {
 5:       int            inuse ;
 6:       lock_t         lock ;
 7:       struct mpcb *queue ;
 8: };
 9: typedef struct sem sem_t ;
10:
11: P( sem )
12:    sem_t *sem ;
13: {
14: loop: lock( &sem->lock );
15:       lock( &current->lock );
16:       if( sem->inuse )
17:       {
18:          enqueue(
                current,&sem->queue );
19:          unlock( &sem->lock );
20:          mp_sleep( 0, 0 );
21:          goto loop;
22:       }
23:       sem->inuse = TRUE ;
24:       unlock( &sem->lock );
25:       unlock( &current->lock );
26: }
27:
28: V( sem )
29:    sem_t *sem ;
30: {
31:    struct mpcb *next ;
32:    struct mpcb *dequeue();
33:
34:       lock( &sem->lock );
35:       sem->inuse = FASLE ;
36:       next = dequeue(
                &sem->queue );
37:       if( next != MP_NULL )
38:          mp_wakeup( next );
39:       unlock( &sem->lock );
40: }
```

図3　マイクロプロセス・ライブラリを利用した2進セマフォの実現

Figure 3: Binary semaphore by using Microprocess Library

図3において、陰影を施した部分が、上記ステップ2において付加された相互排除の操作である。それ以外の部分が、ステップ1において最初に実現する骨格部分である。lock()、unlock()は、第1層の手続きであり、それぞれ、スピンロックによるロック／アンロックを行うものである。

第1行のcurrentは、現在実行中のマイクロプロセスを示す変数である。privateというキーワードは、この変数を仮想プロセッサの固有領域に割り付けるように指示するものである。すなわち、各仮想プロセッサは、それぞれcurrentという変数を持ち、それぞれ現在実行中のマイクロプロセスを保持する。（このような機能は、Ｄｙｎｉｘシステム上のいくつかの言語処理系で実現されている[2]。）

Ｐ命令の実現を図3第11行に示す。第16行において、他のマイクロプロセスがそのセマフォを保持している時（sem->inuseがTRUE）には、自分自身currentを、待ち行列に入れ、ロックを開放する。そして、第19行において、mp_sleep()を呼び出し、現在実行中のマイクロプロセスを待ち状態にする。セマフォが利用されていない時（sem->inuseがFALSE）には、自分自身が利用していることをマーク（第22行）している。

Ｖ命令の実現を、図3第28行に示す。Ｐ命令の中でmp_sleep()により実行を中断したマイクロプロセスは、第37行のmp_wakeup()により、実行可能状態に戻される。

第15行において、現在実行中のマイクロプロセス（current）に掛けられたロックは、mp_sleep()の内部で解除される。この解除を、この部分で先に行うことはできない。それは、sem->lockが開放された瞬間に、第36行において、mp_wakeup()が呼び出される可能性があるからである。実行可能中のマイクロプロセスに対してmp_wakeup()が適用された場合、何も行われないため、そのシグナルが失われる。

3．3　ＳＭＡＳＨシステム

我々が開発しているデータベースの並列処理システムＳＭＡＳＨにおいても、マイクロプロセス・ライブラリが利用それている。ＳＭＡＳＨは、データベース、および、知識ベースを対象とした並列処理システムである[7][8][10]。ＳＭＡＳＨが対象とするハードウェアは、共有メモリ型マルチプロセッサと単一プロセッサが高速ネットワークにより結合された環境である。

ＳＭＡＳＨの特徴は、関数型プログラミングの概念をデータベース処理に適応している点にある。関数の評価方式としては、要求駆動型評価（demand driven evaluation）を用いている。ＳＭＡＳＨでは、関数間のデータの受け渡しにストリームを用いている。任意のデータベース演算を関数として記述し、システム内に組み込むことが可能である。このため、データベース

演算内ではなく演算間に存在する並列性を抽出することを可能になり、データベースの多様な応用分野に柔軟に対応することが可能となっている。

ＳＭＡＳＨシステムでは、並列処理の単位は関数インスタンスと呼ばれる。関数インスタンス間のストリームの通信路は、チャネルと呼ばれる。データベースのデータやデータベース演算の中間結果は、ストリームとして表現される。

ＳＭＡＳＨシステムは、単一プログラミング・システムを想定して設計されている。それを多重プログラミング環境で動作させるために、マイクロプロセス・ライブラリを用いている。ＳＭＡＳＨの関数インスタンスをマイクロプロセスとして実現し、関数インスタンス間の通信プリミティブを、本マイクロプロセス・ライブラリの第２層の機能を利用して実現している。また、ネットワーク通信を実現するために、関数インスタンス以外にマイクロプロセスを生成し利用している [10]。

ＳＭＡＳＨのストリーム通信プリミティブの実現を通じて、本ライブラリの第２層の機能の有効性を確認することができた。４章では、ＳＭＡＳＨシステムを並列応用プログラムの例として、軽量プロセスの実行の軌跡の利用を示す。

３．４　他の軽量プロセス実現方式との比較

軽量プロセスの実現方式としては、軽量プロセスをコルーチンとして実現するコルーチン方式 [14]、軽量プロセスをカーネルにより直接実現するカーネル制御方式 [3]、そして本システムと同様に仮想プロセッサと利用者レベルの軽量プロセスを用いる方式 [1][5][9] があげられる。

本マイクロプロセス／仮想プロセッサ方式の第１の特徴は、システムと並列応用プログラムの間の調和が実現されている点にある。カーネル制御方式では、カーネルが提供するプリミティブを利用して独自の同期・通信プリミティブを構築することになる。この時、両プリミティブの間のギャップにより、効率が低下する、あるいは、応用プログラムが必要とするプリミティブを実現するためにカーネルの機能が役に立たないことがある。本方式では、層構造のライブラリにより、必要に応じてシステムの機能を利用することが可能となっている。

本方式の第２の特徴は、コルーチンの性質の利用にある。これは、本方式では、仮想プロセッサの数を柔軟に制御することができる点にある。すなわち、開発の初期においては、仮想プロセッサの数を１個に制限することで、コルーチンと同様に効率的に軽量プロセス間の同期・通信プリミティブを開発することができる。そして、開発が完了した時点で仮想プロセッサの数を増やせばよい。カーネル制御方式 [3] や、他の仮想プロセッサを用いる方式 [1][5][9] では、カーネル・レベルの軽量プロセスや仮想プロセッサの数を開発のために柔軟に制御することが難しい。

４　軽量プロセスの実行の軌跡の記録と解析

本マイクロプロセス・ライブラリには、マイクロプロセスの生成・消滅や、コンテキスト切替えが起きたことをイベントとしてその時刻を記録する機能がある。この機能は、並列応用プログラムを開発する上で、並列性の抽出の様子を確認するために利用される。

４．１　統計情報

記録されたイベントを用いて、各マイクロプロセス、および、仮想プロセッサごとのＣＰＵ利用時間、入出力時間、コンテキスト切り替えの回数といったの統計情報を得ることができる。その例を、図４に示す。これは、３．３節で述べたＳＭＡＳＨシステム上で関係データベースへの問合わせ処理を実行したものである。利用したのは、４プロセッサ構成の共有メモリ型マルチプロセッサ Luna88k である [6]。この例では、マイクロプロセスが９個（０番～８番）、仮想プロセッサが３個生成されている。

マイクロプロセス関連の情報において、マイクロプロセス０番は、最初のマイクロプロセスを生成するスケジューラである。１番から８番は、普通のマイクロプロセスである。この部分の列は、それぞれ、次のような意味を持つ。

(1) マイクロプロセスの番号
(2) そのマイクロプロセスが利用したＣＰＵ時間の合計とその割合
(3) ＣＰＵを利用した回数（ＣＰＵバーストの回数）
(4) そのマイクロプロセスが行った入出力の時間の合計とその割合
(5) 入出力回数
(6) ＣＰＵ時間と入出力時間の合計とその割合
(7) コンテキスト切替えの回数

この例では、マイクロプロセス１番が全体の１９％の実行時間を占めていること、特に入出力が重たくなっていること等が読み取れる。

仮想プロセッサ関連の部分では、次のような情報が得られる。

(1) 仮想プロセッサ番号
(2) ＣＰＵ稼働時間（この仮想プロセッサがマイクロプロセスを実行していた時間）とその割合。
(3) 入出力時間（この仮想プロセッサにより入出力行っていた時間）とその割合
(4) その仮想プロセッサがアイドル状態にあった時間とその割合
(5) この軌跡をファイルに保存するために要した時間と割合

```
## microprocess information
No  cputime   % / #ex   iotime   % / #io    total   % / #++
 0    0.000    0 /   1    0.000    0 /   0   0.000   0 /   1
 1    0.140    8 /  67    0.370   36 /  62   0.510  19 / 129
 2    0.370   22 /  20    0.000    0 /   0   0.370  14 /  20
 3    0.100    6 /  66    0.180   18 /  65   0.280  10 / 131
 4    0.360   21 /  18    0.000    0 /   0   0.360  13 /  18
 5    0.110    6 /  66    0.180   18 /  65   0.290  11 / 131
 6    0.350   21 /  13    0.000    0 /   0   0.350  13 /  13
 7    0.130    8 /  66    0.160   16 /  65   0.290  11 / 131
 8    0.140    8 /  72    0.130   13 /  65   0.270  10 / 137
##    1.700  100 / 389    1.020  100 / 322   2.720 100 / 711

## virtual processor information
No      cpu    %|      io    %|    idle    %|  trace   %| kernel   %|  total   %|
 0    0.600   54|   0.240   21|   0.230   21|  0.000   0|  0.050   4|  1.120 100|
 1    0.560   50|   0.370   33|   0.160   14|  0.000   0|  0.030   3|  1.120 100|
 2    0.540   48|   0.410   37|   0.130   12|  0.000   0|  0.040   4|  1.120 100|
##    1.700   51|   1.020   30|   0.520   15|  0.000   0|  0.120   4|  3.360 100|
```

図4 実行の軌跡の記録より得られる統計情報

Figure 4: Statistics from microprocess traces.

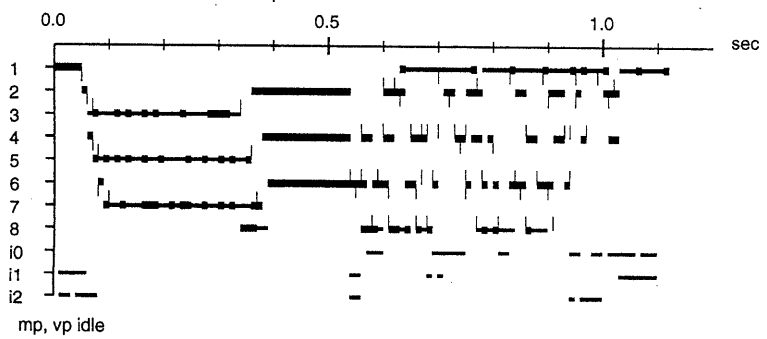

図5 実行の軌跡の記録を視覚化したもの

Figure 5: Visualized traces of lightweight process excution.

(6) その他の時間（時刻の入手、コンテキスト切替えなど）と割合

(7) 合計とその割合

最終行は、全仮想プロセッサの平均である。図4の例では、プロセッサのアイドル時間が15%であったことがわかる。

4．2 実行の軌跡の視覚化

記録されたイベントを、実行の軌跡として図5に示すように視覚化することも可能である。1番から8番において、黒い部分がマイクロプロセスの実行を表す。i0,i1,i2は、それぞれ仮想プロセッサの0番、1番、2番が、他に実行するマイクロプロセスが存在しないため、アイドル状態になったことを表す。並列応用プログラムの開発者は、このような情報を利用して、並列性の抽出の様子を観察することが可能である。

4．3 イベントの記録の仕組み

マイクロプロセス・ライブラリでは、具体的には、次のようなイベントを記録している。

(1) 生成（created）

(2) 終了（exit）

(3) 中断（sleep）

(4) 実行可能化（waked）

(5) プロセッサの譲渡（yield）

(6) 実行開始（scheduled）

(7) 入出力開始（io start）

(8) 入出力終了（io stop）

(9) プロセス全体の終了（process exit）

(10) 仮想プロセッサの一時停止（vp suspend）

(11) 仮想プロセッサの実行再開（vp resume）

(12) 軌跡の記録のための入出力開始（trace file io start）

(13) 軌跡の記録のための入出力終了（trace file io stop）

最初の5つは、それぞれ第2層の mp_create()、mp_exit()、mp_sleep()、mp_wakeup()、mp_yield() に関係している。ただし、mp_wakeup() において、既に実

行可能になっているマイクロプロセスを再び実行可能にすることを試みた場合は、記録されない。(6) は、スケジューラにおいて、実行可能のマイクロプロセスに対してプロセッサが割り当てられたことを示すイベントである。(7), (8) は、マイクロプロセスのＣＰＵ処理と入出力処理を区別して記録する必要がある場合に用いられる。(9), (10) は、仮想プロセッサの動作、(11) は、プロセス全体の動作の示すイベントである。(12), (13) は、軌跡を記録するためのファイル出力を表すイベントである。仮想プロセッサごとにイベントを保存するバッファが設けられている。それらのバッファが一杯になると、その内容がファイルに出力される。

このようなイベントは、各仮想プロセッサごとに記録される。これにより、仮想プロセッサ間の同期処理のオーバヘッドを軽減されている。

本マイクロプロセス・ライブラリでは、静的、あるいは、動的に実行の軌跡を抑止する機能がある。静的な抑止とは、Ｃコンパイラのプリプロセッサのマクロ機能を利用して、イベントを記録するための関数呼び出しを消去する方法である。これにより、完全に実行の軌跡の保存によるオーバヘッドを削除することができる。動的な抑止とは、パラメタの指定により、イベントを記録する関数においてなにもしないで戻ることである。現在の実現では、ＵＮＩＸのシステム・コール、gettimeofday() を用いて時刻を得ている。動的な抑止では、システム・コールの発行を抑える効果がある。

4．4　探り針効果

実行の軌跡を記録する場合、注意すべきことは、探り針効果（probe effect）である。探り針効果とは、イベントを記録するために、その記録という行為が応用プログラムの実行に影響を与えてしまうことである。このマイクロプロセス・ライブラリでは、特に、イベントが発生した時刻を得るためのカーネル・コールのオーバヘッドが大きい。これを改善するために、カーネル・コールによらない時刻の取得方法が求められる。

5　おわりに

この論文では、マイクロプロセス／仮想プロセッサ方式に基づく軽量プロセス実現方式における軽量プロセス間の同期・通信プリミティブの開発の支援、および、各軽量プロセスの実行時間測定の支援について述べた。同期・通信プリミティブの開発においては、層構造のライブラリ、および、コルーチンの性質を活用することができる。その結果、応用固有の同期・通信プリミティブの開発が容易になる。マイクロプロセス・ライブラリには、軽量プロセスの実行の軌跡を保存する機能がある。その軌跡を利用することで、全体の並列処理の様子を視覚化することができる。さらに、軽量プロセスごとにＣＰＵ時間、入出力時間の統計を取ることができる。各並列応用プログラムの開発者は、それらの情報を利用して性能を改善することができる。

今後は、複数の仮想プロセッサを必要とするような応用プログラムに対応していきたい。さらに、実行時間の測定における探り針効果の軽減を計る方式を開発したい。

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", SOSP13, ACM Operat. Sys. Rev., Vol.25, No.5, pp.95-109 (1991).

[2] "Balance 8000 Parallel Programming", Sequent Computer Systems, Inc. (1985).

[3] Black,D.: "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", IEEE Computer,Vol.23, No.5, pp.35-43 (1990)

[4] Graham, S.L., Kessler, P.B., McKusick, M.K.: "gprof: A Call Graph Execution Profiler", Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol.17, No. 6, pp. 120-126 (1982).

[5] Inohara,S., Kato,K., Narita,A. and Masuda,T.: "A Thread Facility Based on User/Kernel Cooperation in the XERO Operating System", Proc. 15th Intl. Comput. Software &Applications Conf, pp.398-405 (1991).

[6] 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).

[7] 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).

[8] 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).

[9] Marsh,B. and Scott,M.: "First-Class User-Level Threads", SOSP13, ACM Operat. Sys. Rev., Vol.25, No.5,pp.110-121 (1991).

[10] 佐藤, 清木 : "関数型計算に基づくデータベースシステムの並列処理実行系の実現方式", 情報処理学会データベースシステム研究会研究報告, 92-DBS-89, pp.151-160 (1992).

[11] 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).

[12] 新城, 清木: "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", 情報処理学会論文誌, Vol.34,No.3, pp.478-488 (1993).

[13] Shinjo,Y. and Kiyoki,Y.: "Harmonizing a Distributed Operating System with Parallel and Distributed Applications", Proc. 1st International Symposium on High Performance Distributed Computing (HPDC-1), pp.114-123 (1992).

[14] "SunOS Refernce Manual", Sun Microsystems,Inc. (1988).