マルチスレッドアーキテクチャ向けOS「Future」におけるプロセス管理

佐藤 未来子[†] 笹田 耕一[†] 加藤 義人[†]
大和 仁典[†] 河原 章二[†,☆]
中條 拓伯[†] 並木 美太郎[†]

マルチスレッドアーキテクチャプロセッサを搭載したシステムにおいて，スレッドレベル並列性（TLP）の高いアプリケーションを高効率に実行させるために，マルチスレッドアーキテクチャプロセッサ向けOS「**Future**」におけるプロセス管理について検討し，(1) 複数のハードウェアコンテキストを持つプロセスを切り替える方式，(2) OS内で検出したスレッド制御にかかわる事象をユーザレベルへ通知する方式を考案し実現した．その結果，マルチスレッドアーキテクチャプロセッサの複数のハードウェアコンテキストに対し，ユーザレベルスレッドライブラリから直接スレッド制御するプロセスモデルを実現することを可能にした．また，シミュレーションにより，本論文で提案したプロセス切替え方式の基礎評価を行い，その有効性を明らかにした．

Process Management of the Operating System "Future" for On Chip Multithreaded Architecture

Mikiko Sato,[†] Koichi Sasada,[†] Norito Kato,[†]
Masanori Yamato,[†] Shoji Kawahara,[†,☆] Hironori Nakajo[†]
and Mitaro Namiki[†]

We propose a process management model in a dedicated system-software for a system that is installed into an multithreaded architecture processor. In order to gain high performance in executing applications on the system based on thread level parallelism (TLP) like a Multithreaded (MT) architecture processor, we designed an operating system (OS) to assign multiple Architecture (Physical) threads on the MT processor to a process. And all architecture threads in the process are managed on a user level library to parallelize threads directly. In this paper, we introduce a process management of MT-oriented OS "*Future*" and we explain the method of the process switching in *Future* and event informing to user level which is detected in *Future*. Finally, we discuss our implementation and performance about these.

1. はじめに

今日，トランジスタ集積技術の向上により，CPUをチップ上に複数搭載する Chip MultiProcessor（CMP）や，CPU内の演算器などのハードウェアリソースを共通に利用しながら複数の命令流（スレッド）を同時実行させる Simultaneous Multithreading（SMT）アーキテクチャ[1),2)] プロセッサが市場に登場してきた．

CMPおよびSMTアーキテクチャは，ともにスレッドレベルの並列性（TLP: Thread Level Parallelism）の向上を目的とするマルチスレッドアーキテクチャであり，1チップで同時に複数の命令流を扱うことが可能なプロセッサである．これらのマルチスレッドプロセッサを搭載するシステムを汎用機として製品化する場合には，SMP（Symmetric MultiProcessor）システム用の汎用オペレーティングシステム（OS）を一部改変し，マルチスレッドプロセッサの個々の計算実体をSMPのCPU単体に見立てて管理し，既存のアプリケーションプログラムに何ら変更を加えずに実行させているのが現状である[7)]．

従来のSMP向けOSを適用した場合，**図1**に示す

† 東京農工大学大学院工学研究科電子情報工学専攻
Graduate School of Technology, Tokyo University of Agriculture and Technology
☆ 現在，NECシリコンシステム研究所
Presently with Silicon Systems Research Laboratories, NEC Corporation

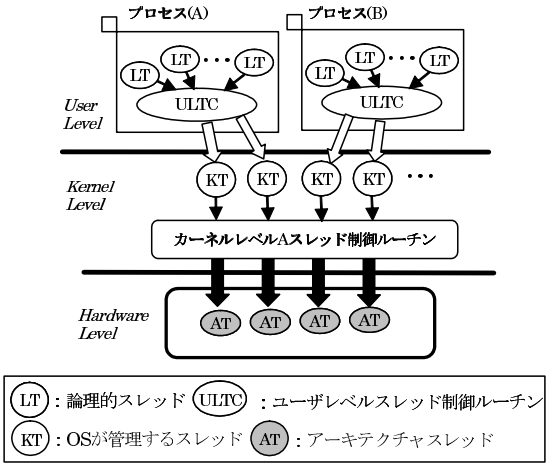

図 1 従来 OS におけるプロセス管理
Fig. 1 Process management model of current operating system.

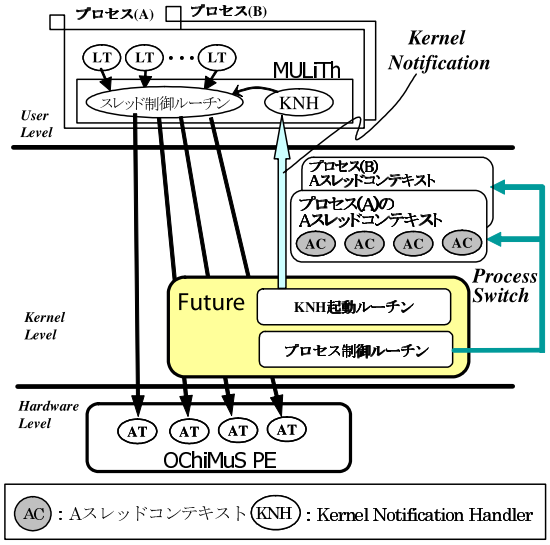

図 2 本研究におけるプロセス管理
Fig. 2 Process management model of *Future*.

ように，1 チップ上で複数の UNIX 流プロセスを割り当てることとなる．このことは，キャッシュメモリや TLB といったメモリ資源に複数のプロセスのデータが混在することとなり，メモリ資源の競合によるキャッシュミス，TLB ミスなどを引き起こし[3]，プログラムの実行効率を下げる原因となる．また逆に，共有データをアクセスするプロセスどうしを割り当てた場合にはメモリアクセス効率が向上し[3]，プログラムの実行効率の向上につながる．

このように，マルチスレッドアーキテクチャプロセッサへの命令流割当て制御がアプリケーションの実行性能に与える影響は大きい．そこで筆者らは，マルチスレッドアーキテクチャプロセッサを搭載したシステム向け OS「**Future**」の研究において，まずプロセスモデルについて検討した[4]．

図 2 に本研究におけるプロセスモデルを示す．**Future** はマルチスレッドアーキテクチャプロセッサを 1 つの CPU 資源として管理し，プロセッサ上の複数の計算実体をプロセスへ割り当てる．プロセスに属する個々の命令流の管理・制御はユーザレベルライブラリ **MULiTh**（Userlevel Thread Library for Multi-threaded architecture）[5] で担い，軽量なスレッド制御を実現する．

本モデルは，マルチスレッドアーキテクチャプロセッサ上で実行するプロセスのメモリアクセス効率の面で有利である．しかし，本プロセスモデルの実現のためには，複数の計算実体を 1 つの CPU 資源として扱う新しいプロセス管理方式が必要となる．

本論文では，**Future** におけるプロセス管理について述べる．2 章で筆者らが研究対象としているマルチスレッドアーキテクチャプロセッサについて述べる．3 章で **Future** のプロセス管理に関する課題を述べ，4 章でプロセス管理に関する目標を述べ，5 章で本研究で定義したプロセスのプロセスコンテキストについて説明する．6 章で，課題の 1 つとしているプロセス切替え方式について述べ，7 章でユーザレベルでのスレッド制御をサポートするための，OS 内事象通知機構の実現方式を述べ，8 章で評価結果を述べる．

なお，**MULiTh** におけるスレッド管理方式，および，OS からの事象をユーザレベルで受信する機構（Kernel Notification）については文献 5) で報告しているので，詳細はそちらを参照されたい．

2. OChiMuS PE

筆者らは，本研究で扱うマルチスレッドアーキテクチャプロセッサとして，オンチップマルチ SMT（OChiMuS）アーキテクチャを並行して検討している[6]．**Future** は，現在，単一の SMT PE（Processing Element）を搭載する OChiMuS プロセッサ（以降，**OChiMuS PE** と呼ぶ）を対象に研究を進めている．**OChiMuS PE** および OChiMuS アーキテクチャについての詳細は文献 6) を参照されたい．

OChiMuS PE は，MIPS プロセッサアーキテクチャをベースとした SMT アーキテクチャプロセッサであり，マルチスレッドをサポートするためのモジュールや命令を新たに拡張している．**OChiMuS PE** では，チップ上の計算資源を利用している実行中の命令流のことを**アーキテクチャスレッド**（Architecture

表 1 アーキテクチャスレッドの状態
Table 1 States of the architecture thread.

A スレッドの状態	状態の意味
NORMAL	命令フェッチ可能
HALT	完全停止
BLOCK	命令フェッチを一時停止

Thread：以下，**A スレッド**）と呼び，プログラムカウンタ（以下，PC），汎用レジスタなどの資源を各 A スレッドごとに備え，演算器などのパイプライン資源やキャッシュメモリ，TLB を A スレッド間で共通に利用する．以下，A スレッドごとに備えたハードウェア資源のことを「A スレッドコンテキスト」と呼ぶ．

OChiMuS PE では A スレッドを仮想化する機能を有しており，システムソフトウェアが管理するスレッド（以下，これを**論理スレッド**と呼ぶ）を **OChiMuS PE** 内のどの A スレッドに割り当てているか，という対応付けを **OChiMuS PE** 内部で管理している．**Future** や **MULiTh** などのシステムソフトウェアは，システムソフトウェアで管理する論理スレッドの識別子（LTN: Logical Thread Number）を指定して，A スレッドの生成・削除・一時停止などを制御する．これらのスレッド制御命令はユーザレベルで実行できる．A スレッドの状態は，スレッド制御命令により**表 1** に示す 3 つのいずれか状態となる．また LTN は，各 A スレッドごとに備えられたユーザレベルからもアクセス可能な専用レジスタに保持されており，レジスタアクセスと同様に設定，参照ができる．

また，**OChiMuS PE** の例外に関しては，外部割込みを固定 A スレッドで発生させ，その他すべての例外☆を例外要因の命令コードを実行させた個々の A スレッドで発生させるという仕様である．

3. Future におけるプロセス管理の課題

本章では，まず **Future** で扱うプロセスに関する特徴を述べ，プロセス管理に関する課題を述べる．

3.1 Future で扱うプロセスの特徴

従来 OS では，プロセスに割り当てるハードウェアコンテキストが 1 つであったため，OS がプロセスごとに管理するハードウェアコンテキストも 1 つであった．一方，**Future** の場合，**OChiMuS PE** 内のすべての A スレッドコンテキストをプロセスへ割り当てるため，プロセスごとに複数のハードウェアコンテキストを含むという特徴がある．

☆ MIPS アーキテクチャで扱う割込み例外以外の例外を指す．たとえば，システムコール例外，アドレスエラー例外，TLB 関連例外，予約命令例外などである．

3.2 プロセス切替えに関する課題

3.1 節で述べた **Future** で扱うプロセスの特徴により，**Future** ではプロセス切替えの際に，**OChiMuS PE** で実行中のすべての A スレッドを対象に退避・復帰する機構の実現が課題となる．特に性能面で，従来 OS のプロセス切替えオーバヘッドに対し，A スレッド数に比例して増大することが予想される．より効率の良いプロセス切替えが **Future** の課題となる．

3.3 ユーザレベルへの OS 事象通知に関する課題

Future では，プロセスに属する各論理スレッドのハードウェアコンテキストの管理および制御に関する権限を，すべてユーザレベルライブラリに任せている．ユーザレベルでより効率の良いスレッド制御を行うためには，Scheduler Activations [12] などの従来手法を用いて，論理スレッドの制御にかかわる，OS 内で発生した事象をユーザレベルへ通知することが有効である．たとえば，I/O リクエスト発生による論理スレッドのブロッキングを検出し通知することで，ブロックした論理スレッドを実行可能な別の論理スレッドへ切り替えることが可能となる．

Future でも，ユーザレベルのスレッドの実行制御をサポートするために，OS 内で検出した事象をユーザレベルへ通知する機構を設けることを課題とする．**Future** の場合，具体的には，**MULiTh** に備える事象の受信処理を担う Kernel Nortification Handler（以下，KNH）[5] へ事象を通知し，KNH で OS から送られた事象やブロックした論理スレッド情報をもとに論理スレッドを再スケジューリングする．**Future** では通知する事象の管理，および効率の良い OS 事象通知処理を設けることが課題となる．

4. Future のプロセス管理の目標

Future では，1 個以上の論理スレッドで構成された実行実体を “**プロセス**” として扱い，プロセスの管理，制御（生成，削除，切替え）を担う．筆者らは，マルチスレッドアーキテクチャプロセッサにて，より高いプロセス実行性能を得るために，以下の目標を掲げた．

(1) プロセスごとに複数の A スレッドコンテキストを管理し，プロセス切替え時に **OChiMuS PE** 内のすべての A スレッドコンテキストを対象とした退避・復帰を可能とする機構を備えること．

(2) OS 内で検出した論理スレッド制御にかかわる事象を管理し，KNH に対して通知する機構を備えること．

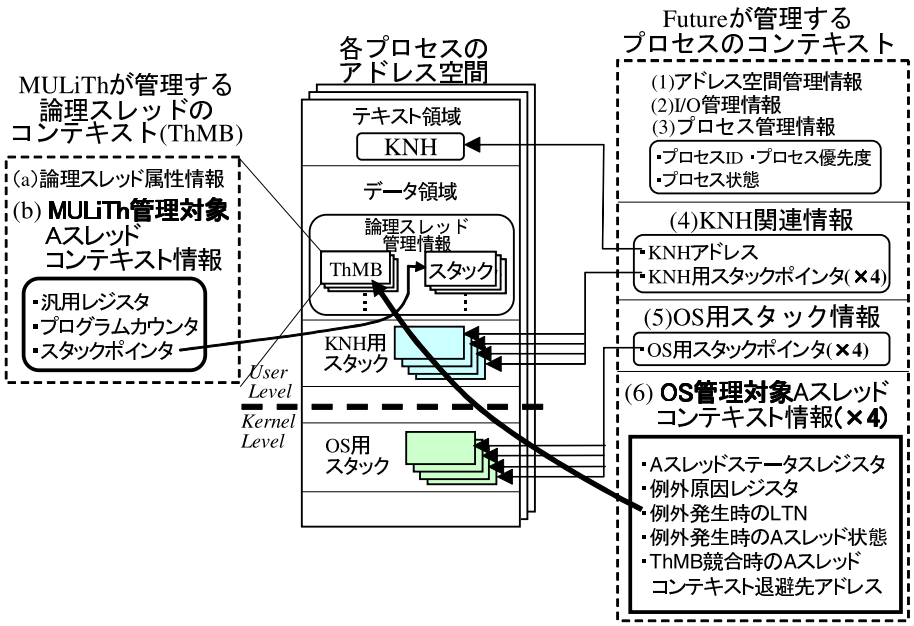

図 3 各プロセスのアドレス空間上の情報とプロセスコンテキストの関係
（OChiMuS PE 内 A スレッドが 4 個の場合）
Fig. 3 The relations between an address space and a process context.

文献 5) では，KNH による OS 事象受信方式を述べているが，OS 内の事象の管理，KNH の起動方法についての記載はないため，本論文で詳細を述べる．

5. Future におけるプロセスについて

5.1 プロセス管理の概要

Future では 1 つのプロセスに対して，**OChiMuS PE** 内のすべての A スレッドコンテキスト，仮想アドレス空間，および，仮想 I/O 資源を割り当てる．同じプロセスに属するすべての論理スレッドは，プロセスに割り当てられたこれらの資源を共有する．その結果，データ共有する論理スレッドどうしは，ある論理スレッドによってすでにキャッシュメモリへロードされたデータを他の論理スレッドでもアクセスすることで結果としてキャッシュミスを減らすという「建設的干渉（constructive interference）[2),3)]」が期待できるようになる．

図 3 に，**Future** と **MULiTh** とで分担して管理するプロセスコンテキストとアドレス空間上での配置関係を示す．**Future** は，プロセスごとに図 3 内の（1）〜（6）に示すプロセスコンテキストを管理する．各プロセスのアドレス空間上に割り当てたテキスト領域・データ領域・KNH 用のスタック領域などのアドレス空間管理情報や，プロセスが使用中の I/O 資源情報や，プロセススケジューリングに必要なプロセス管理情報をプロセスコンテキスト（1）〜（3）で管理する．**Future** のプロセスコンテキストの特徴は，次の 2 点である．

- プロセスコンテキスト（5）で OS 用スタックを A スレッド個数分管理し，各スタック上で A スレッドコンテキスト（6）を管理すること．
- プロセスコンテキスト（4）で KNH アドレス情報，および A スレッド個数分の KNH 用スタックポインタを管理すること．

OS 用スタックを A スレッド個数分管理することにより，2 章で触れた **OChiMuS PE** の例外が発生した場合に，個々の A スレッドが並行して例外処理可能となる．

Future では，KNH 登録用システムコールを提供しており，プロセス起動時に **MULiTh** がこのシステムコールを使い，プロセスコンテキスト（4）の KNH のアドレスを登録する．また，**Future** では，プロセス生成時に KNH 用スタックを A スレッド個数分プロセスへ割り当て，プロセスコンテキスト（4）で各スタックポインタを管理する．これにより，7.2 節および 7.3 節内に示す KNH への事象通知処理を個々の A スレッドで並列に実行可能としている．

次に，**Future** が扱う A スレッドコンテキストとその退避先を**表 2** に示す．**Future** では A スレッドコンテキストのうち，汎用レジスタと，例外発生時の

表 2 A スレッドコンテキストの退避情報と退避場所
Table 2 Relations of the architecture thread context informations and save areas.

退避先	退避情報
ThMB	汎用レジスタ 例外発生時の PC
OS 用スタック	例外発生時のステータスレジスタや 例外原因レジスタの内容 例外発生時の A スレッドの LTN 例外発生時の A スレッドの状態 A スレッドコンテキスト退避領域情報

PC をユーザレベルで管理する論理スレッド用データ管理ブロック（ThMB: Thread Management Block）へ退避し，その他のコプロセッサレジスタの情報，および A スレッド関連レジスタの情報を OS 用スタック内に退避し管理する．**MULiTh** では，論理スレッド生成時に，論理スレッド用スタックと ThMB を各論理スレッドへ割り当て，ThMB で各論理スレッドの属性情報（図 3 (a)）と，実行を中断した論理スレッドの A スレッドコンテキスト（図 3 (b)）を管理する．また，ThMB の先頭アドレスを論理スレッドの識別子（LTN）として使用する．LTN が ThMB の先頭アドレスを指すことにより，**MULiTh** の ThMB 検索を容易にするだけではなく，ThMB を管理していない **Future** からも実行途中の論理スレッドの A スレッドコンテキスト退避先アドレスを得られるようにしている．

このように，ThMB へつねに実行途中の論理スレッドの汎用レジスタと PC を退避しておくことにより，7 章で示す KNH 起動処理の際に，A スレッドコンテキストコピーの回数を削減できるという効果が得られる☆．ただし，ユーザレベルにおいて論理スレッド切替え途中に割込みが発生したような場合には，**Future** と **MULiTh** 間で ThMB に対する競合が発生する[5]．これに対し **MULiTh** がユーザレベルで ThMB を利用している場合には LTN を特別な値に設定するというルールを設け，ThMB の競合を回避している[5]．以下本論文ではこの特別な LTN 値を仮に「0」とする．**Future** では LTN が「0」であった場合には，**Future** では，すべての A スレッドコンテキストを OS 用スタックへ退避し，その退避場所をプロセスコンテキスト（6）の「ThMB 競合時の A スレッドコンテキスト退避先アドレス」へ保持することで，ThMB 内の情報破壊を回避している．

なお，**MULiTh** では，`pthread_yield` 関数による明示的な論理スレッド切替え時や，論理スレッドの実行終了時や，KNH における OS 事象受信時に，論理スレッドスケジューラを起動し，実行待ち状態の論理スレッドがあれば，論理スレッドの LTN から ThMB のアドレスを得て，A スレッドコンテキストを切り替える[5]．スタックポインタはハードウェアアーキテクチャの仕様に従い汎用レジスタ内で保持しているため，A スレッドコンテキスト切替えにともない切り替わる．

☆ 本効果については文献 5) に記載されている．

5.2 Future の A スレッドコンテキストの退避・復帰の流れ

例外発生時 **Future** では，ThMB を競合して利用しないことを確認し，A スレッドコンテキストのうち，汎用レジスタと例外発生時の PC を ThMB（図 3 (b)）へ退避し，その他のコプロセッサレジスタの情報，A スレッドの LTN，表 1 で示した A スレッドの状態を OS 用スタック内（図 3（6））へ退避する．例外処理から元の論理スレッドに復帰する場合には，OS 用スタック内に退避した LTN の値から ThMB を参照し，汎用レジスタや PC の情報を復帰する．また，発生した例外がタイマ割込みや I/O 割込み，sleep システムコールなどプロセス切替えを引き起こす例外だった場合には，上記と同様に ThMB および OS 用スタックへ A スレッドコンテキストを退避した後，次章で示すプロセス切替え方式によりスタックなどの情報とともにプロセスコンテキストを切り替える．プロセス切替え後にユーザレベルへ戻る際には，7.2 節に示す KNH への事象通知を行う．

KNH では，**Future** の処理中に一時停止させた論理スレッドの LTN と，論理スレッドの一時停止の理由（すなわち 7 章で示す通知事象）を受け取り，これらの情報をもとにスレッドスケジューリングを行う．

なお，ThMB 競合を回避した場合でも KNH への事象通知が必要となる場合がある．この場合，**Future** では KNH への事象通知後，OS 用スタック内に退避した A スレッドコンテキスト（図 3 (b)）の復帰を行ったり，事象通知を行わずに A スレッドコンテキストの復帰のみを行ったりするなどの方法で対処する．それらの処理については，7.1 節で示す．

6. プロセス切替え方式

Future では，従来の OS とは異なり，プロセス切替え時に複数個の A スレッドコンテキストをすべて切り替える必要がある．まず筆者らは，**Future** におけるプロセス切替えのときの A スレッドコンテキストの退避・復帰方法に関して以下の方式を検討した．

(1) ある 1 つの A スレッドが退避・復帰を担う方式

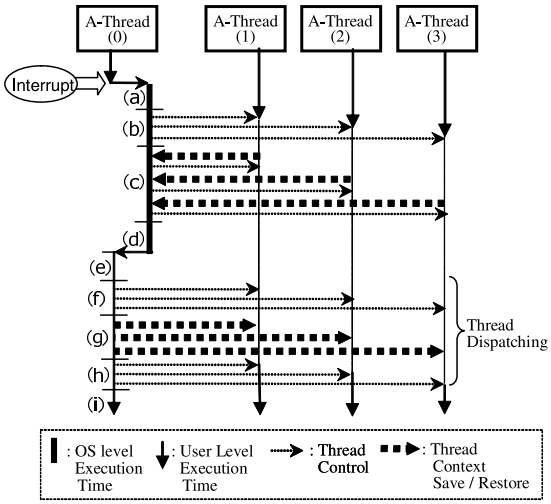

図 4 方式 (1) のプロセス切替えの概念図
Fig. 4 The process switching flow (1).

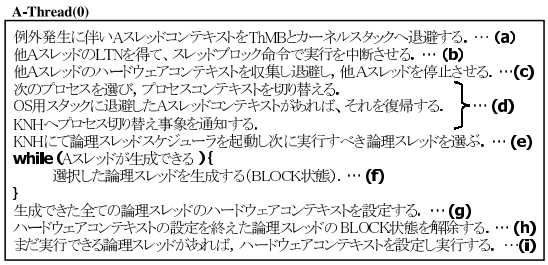

図 5 方式 (1) のプロセス切替えの流れ
Fig. 5 The chart of the process switching flow (1).

(2) 各 A スレッドで退避・復帰処理を分担する方式

図 4 および**図 5** に 4 つの A スレッドを備えた **OChiMuS PE** でプロセス切替えの契機となる例外を A スレッド「0」が受け付けた場合の方式 (1) の処理の流れを，概念図および C 言語風の擬似コードによって示し，**図 6** および**図 7** に方式 (2) の処理の流れを同様に示す．

方式 (1) は，プロセス切替えの契機となる例外を受け付けた A スレッドが，OChiMuS アーキテクチャの「スレッド一時停止命令（PBLK 命令）」を用いて他の A スレッドの実行を一時停止させ，PE 内の全 A スレッドコンテキストを退避し，次のプロセスの A スレッドコンテキストを復帰する方式である．

方式 (1) には 3 つの問題がある．第 1 の問題は，A スレッドコンテキストの退避・復帰処理をシーケンシャルに行うため，**OChiMuS PE** の A スレッドの個数の増加にともない，プロセス切替えオーバヘッドの増加の割合が大きくなるという問題である．この問題については，8 章内の評価で結果を示す．第 2 の問題は，以下に示す複数の専用命令が必要となることである．

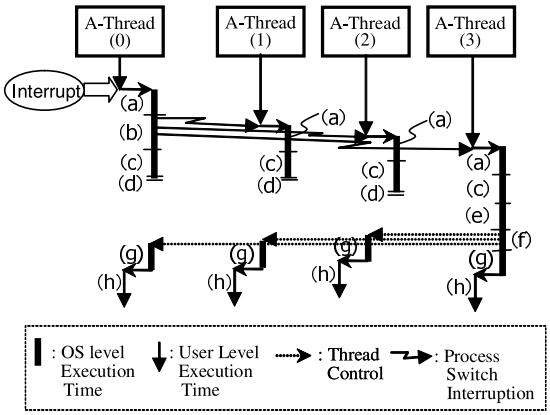

図 6 方式 (2) のプロセス切替えの概念図
Fig. 6 The process switching flow (2).

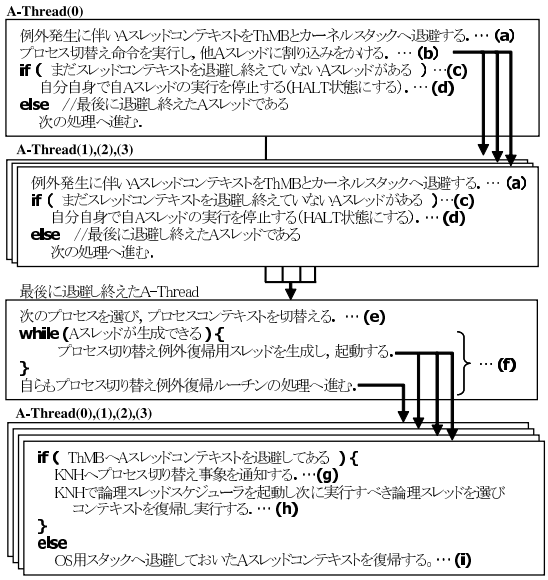

図 7 方式 (2) のプロセス切替えの流れ
Fig. 7 The chart of the process switching flow (2).

方式 (1) では他の A スレッドを一時停止させるために，他の A スレッドで実行中の論理スレッドの LTN を知る必要があるが，OS は論理スレッドを管理していないため，他の A スレッドの LTN の値を得る命令が必要となる．また，一時停止している他の A スレッドコンテキスト（汎用レジスタ，PC など）の情報を得るための命令も必要となる．また第 3 の問題は，処理（b）で OS 実行中の A スレッドが存在する場合の実装が複雑になる点である．OS レベルで中断不可能な処理（たとえば，TLB の設定やページ管理表・I/O 管理表などの更新中など）をしている A スレッドがある場合，OS レベルのロックを獲得した状態の A スレッドを中断させてしまうこととなり，システム全体

に不具合が発生してしまう．これを避けるために，OS レベルのクリティカルな処理を中断させないための実装が必要となる．たとえば，ユーザレベルの A スレッドだけを一時停止させる命令を用意し，OS 実行中の A スレッドがユーザレベルに復帰した後にプロセス切替えのための A スレッドコンテキスト退避を開始する，といった実装が必要となる．

方式 (2) は，プロセス切替えの契機となる例外を受け付けた A スレッドが，他の A スレッドで「プロセス切替え用例外」を発生させる命令（以下，この命令を「プロセス切替え命令」と呼ぶ）を実行して例外を発生させ，各 A スレッドが A スレッドコンテキストの退避・復帰処理を行う方式である．図 7 内の処理 (a), (c), (d) で例外処理の枠組みで各 A スレッドが並列に A スレッドコンテキストを退避したり，処理 (g)〜(i) でユーザレベルへ復帰したりするための「プロセス切替え例外復帰用スレッド」を各 A スレッドで実行するなど，SMT アーキテクチャの並列性をコンテキスト退避・復帰に利用した効率的なプロセス切替えを可能とする．また，例外処理の枠組みでカーネルレベルで各 A スレッドが A スレッドコンテキストを退避するので，OS 実行中の A スレッドにプロセス切替え例外を発生させた場合も，従来の多重例外処理の枠組みで，実行途中の OS の処理を済ませてからプロセス切替え例外の処理を開始するといった実装も容易である．

方式 (2) の問題点は，プロセス切替え用例外のサポート，およびプロセス切替え命令の追加を要することである．しかし，これらのハードウェアに対する要求は，方式 (1) で追加する必要がある命令数に比べ少ない．

筆者らは，性能を第 1 に考え，同時にシステムソフトウェア実装の容易性，ハードウェアアーキテクチャに追加する機能が少ないという点で，方式 (2) が最良であると判断し，採用した．OChiMuS アーキテクチャにはプロセス切替え用例外，およびプロセス切替え命令を追加した．

なお，**Future** におけるプロセス切替え例外復帰用スレッドの処理では，OS 用スタックにすべての A スレッドコンテキスト退避していた（すなわち，ThMB の競合を起こしていた）場合を除き，プロセス切替えが発生したことを KNH へ通知する．KNH の起動方法および，プロセス切替え時に **Future** が KNH を起動する理由，起動するための A スレッドコンテキストの内容などについては，次章に示す．

7. Future の OS 事象通知方式

Future では現状，次に示す事象を検出してユーザレベルへ通知する．

(1) I/O アクセスなどのシステムコールにより，論理スレッドがブロックしたこと
(2) ブロックしていた論理スレッドの I/O リクエストが完了したこと
(3) プロセス切替えが発生し論理スレッドを中断させたこと

これらの事象以外にも，ページフォールト発生によりスレッド実行がブロックしたこと，シグナル通知なども，本枠組みによりユーザレベルへ通知可能と考えている[5]．本章では，**Future** における KNH の起動方法を述べ，上記の事象管理と通知方法について述べる．

7.1 KNH の起動方法

図 8 に，**Future** が KNH を起動する方法を C 言語風擬似コードで示す．**Future** では，処理（a）で A スレッドコンテキスト退避場所を確認し，A スレッドコンテキストが ThMB へ退避されていた場合には，処理（b）で，通知する事象情報，KNH 関連の各種情報（プロセスコンテキストで保持する KNH 用スタックと KNH アドレス），そして ThMB の競合を避けるための LTN 値（0）を設定してユーザレベルへ戻るという手続きで KNH を起動する．

例外的に，事象を検出しても KNH へ戻らない場合がある．それはプロセス切替えの発生時にユーザレベルで論理スレッド切替えが行われていた場合である（処理（d））．この場合，すでにユーザレベルで論理スレッドの再スケジュールを行っていた状況なので，KNH をあえて起動せず，処理（e), (f）により，OS

```
Aスレッドコンテキストの Aスレッドコンテキスト退避場所情報を確認する. …(a)
if ( 汎用レジスタとPCはThMBへ退避済みである ){ …(b)
    KNHへ通知するOS事象の情報を汎用レジスタへ設定する. …①
    KNH用スタックポインタを設定する. …②
    KNHのアドレスをユーザレベル復帰PCとして設定する. …③
    KNHを起動するためにLTNを「0」に設定する. …④
    ユーザレベルへ戻る. …⑤
}
else { …(c)
    if ( 通知事象が「プロセス切り替え」である ){ …(d)
        OSスタック内に保持していたAスレッドコンテキストを復帰する. …(e)
        ユーザレベルへ戻る. …(f)
    }
    else { // その他の通知事象である …(g)
        事象受信処理後にまたOSへ遷移してもらうための情報を
            汎用レジスタへ設定する. …(h)
        ①〜⑤の処理を施し、ユーザレベルへ戻る。…(i)
    }
}
```

図 8 KNH 起動処理の流れ
Fig. 8 KNH execution flow.

用スタックに退避されている A スレッドコンテキストを復帰する．

また，**Future** が複数の通知事象をかかえる場合がある．たとえば，以下の状況がある．

- OS 用スタック内に汎用レジスタ，PC 情報を退避した場合で KNH へ情報通知する必要が生じた場合（処理（g））．
- 複数の I/O 完了や複数のシグナルを通知する場合．
- プロセス切替えと I/O 完了やシグナルなどを組み合わせて通知する場合．

これらの場合，**Future** では，1 つの事象を KNH へ知らせると同時に，再び **Future** に戻るよう指示を出す．KNH では，**Future** から OS 再入の指示を受け取った場合，**Future** へ再び戻るために専用のシステムコールを用いて OS へ戻る．**Future** では，OS 再入の理由を調べ，OS スタックに保持しておいた A スレッドコンテキストの復帰や，第 2 の事象通知を行う．

以上の方法により，**Future** が様々な状況で検出した事象を，すべて KNH へ通知できるようになり，**Future** が検出した事象をユーザレベルでの適切な論理スレッドスケジューリングに活かせるようになる．

7.2 プロセス切替え事象の管理と通知

Future では，プロセス復帰時に KNH へプロセス切替えが発生したことを通知する．**Future** が KNH へ通知する情報は以下のとおりである．

(i) プロセス切替えの発生したことを示すフラグ（“KNH_STATUS_SWITCH”）

(ii) プロセス切替えにより，**Future** が実行を中断した論理スレッドの LTN

ユーザレベルで論理スレッドを定期的に切り替えるためには，何らかの時間情報を得る必要がある．しかし，時間情報を得るためにシステムコールを用いる方法では，システムコールによる OS 遷移のオーバヘッドが大きい．そこで，本プロセス切替えの事象 (i) を通知することで，「プロセス切替え」という時間間隔を KNH へ通知する．これにより，システムコールを用いた時間情報の取得を行わなくても，少なくとも，プロセス切替えのタイミングで，ユーザレベルでの論理スレッドの再スケジューリングが可能となる．

7.3 I/O ブロッキング事象の管理と通知

Future では，I/O 資源ごとに **I/O リクエストキュー**を備え，I/O 待ち状態となった論理スレッドに関し，次の情報を管理する．

- プロセス ID
- I/O リクエストを実行した論理スレッドの LTN

そして **Future** では，I/O リクエストが完了時に，本 I/O リクエストキューの情報を参照し，KNH へ I/O ブロック解除となった論理スレッドの LTN を通知する．

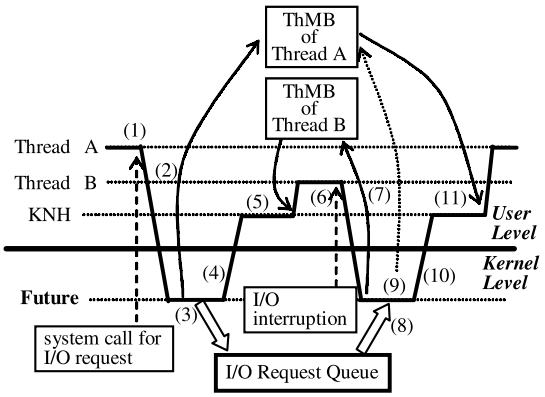

図 9 Kernel Notification 機構の流れ（I/O ブロック発生・解除の例）
Fig. 9 The time chart of Kernel Notification.

図 9 に，システムコールによる I/O リクエストを受け取ってから，I/O リクエストが完了したことを **Future** が通知するまでの処理の流れを示す．

まず，(1) I/O リクエストのシステムコールにより，カーネルレベルへ遷移する．(2) **Future** で A スレッドコンテキストを ThMB へ退避する．(3) **Future** で I/O リクエストを処理する．ここで I/O ブロックを検出する．(4) KNH へ I/O ブロックが発生したことを通知する．KNH へは，I/O ブロックが発生されたことを示すフラグ（KNH_STATUS_BLOCKED）だけを通知する．(5) KNH では，これらの情報を得て，別の論理スレッドを再開するために論理スレッドを再スケジュールする．

次に I/O リクエストが完了したときの流れを説明する．まず，(6) I/O 割込み例外が発生し，OS へ遷移する．(7) **Future** では，割込み発生時に実行中だった論理スレッドのコンテキストを ThMB へ退避する．(8) 割込み例外処理において，完了した I/O のリクエストキューを調べ，待機中の論理スレッドの LTN を得る．(9) LTN が指す ThMB に対し，完了した I/O リクエストの結果を反映させる．(10) KNH へ I/O ブロックが解除されたことを通知する．KNH へ通知する情報として，以下の 3 つの情報を渡す．

(i) I/O ブロックが解除されたことを示すフラグ（KNH_STATUS_UNBLOCKED）

(ii) OS 遷移時に実行中だったスレッドの LTN

(iii) I/O ブロックが解除された論理スレッドの LTN

(11) KNH では，I/O ブロックが解除されたことと，

上記（ii），（iii）の LTN 情報を得て，これらの論理スレッドを含め再スケジュールする．

本事象通知方法により，ユーザレベルにおいて I/O ブロック中の論理スレッドの管理を省くことができる．

8. 実装と評価

筆者らは，本論文で述べた **Future** のプロセス切替え方式に関して，実行駆動型シミュレータ「MUTHASI（MultiTHreaded Architecture Simulator）[6]」を用いて評価した．本シミュレータは **OChiMuS PE** をシミュレートし，A スレッドの個数やキャッシュなどのパラメータを設定可能である．筆者らは，GNU の binutils-2.12.91☆，gcc-3.2☆☆，newlib-1.9.0 を用いて，シミュレータ上で動作させる評価版 **Future** を構築した．

6 章で示したプロセス切替え処理における実行性能を計測するために，プロセス切替えを要求するシステムコール「yield()」を **Future** に用意した．そして，**OChiMuS PE** に搭載する A スレッド数を 1，2，4，8 に増加させた場合，プロセス切替えに要する時間がどの程度増加するかを計測した．ここで，プロセス切替えに要する時間とは，yield() による OS 遷移後から次の論理スレッドのディスパッチまでの時間とする．したがって，A スレッド数が 1 の処理が，ほぼ従来 OS におけるプロセス（またはカーネルレベルスレッド）切替え処理に相当すると考えてよい．

図 10 に筆者らが提案するプロセス切替え方式に要するサイクル数をメモリオーバヘッドのない状態で計測した結果と，6 章で比較検討した方式 (1) のプロセス切替えに要するサイクル数を机上計算で求めた結果を示す．どちらも，ThMB の競合が起きない場合のプロセス切替え処理のサイクル数を示している．

方式 (1) のサイクル数の算出では，図 5 内に示した処理 (a), (b), (c) に要する時間をそれぞれ **Ta**，**Tb**，**Tc** とし，図 5 内処理（d）でプロセスをスケジューリングおよび切替えに要する時間を **Td**，同じく処理（d）内で KNH を起動し事象通知をする時間を **Te**，図 5 内処理（e）〜（h）で論理スレッドを復帰する時間を **Tf** とする．A スレッド数を n とした場合の **Ta**〜**Tf** の算出式を以下に示す．方式 (2) と同等の処理は，方式 (2) の計測値を流用した．流用できない部分は，スレッド制御命令（一時停止命令<PBLK>，一時停止解除命令<PUBLK>，完全停止命令<PDALL>，スレッド生成命令<PALLC>，A スレッド間汎用レジスタ転送命令<FWD>），メモリへのストア命令<SW>などを組み合わせてほぼ同等の処理を行う命令列を作成し，方式 (2) と同じシミュレーション環境で実行サイクル数を計測した．なお，アーキテクチャに実装されていない他の A スレッドコンテキストを得る命令（一時停止中の A スレッドの PC，LTN，汎用レジスタを得る命令）については，汎用レジスタ転送命令<FWD>に置き換えた．また，MIPS アーキテクチャでは汎用レジスタ 0 番は 0 固定であり，26 番と 27 番はカーネル処理のために予約されているのでこれらを復帰，退避の対象からはずし，一方で PC，LTN を汎用レジスタと同様に復帰，退避すると仮定しているため，<FWD>や<SW>の実行回数を「30」とした．

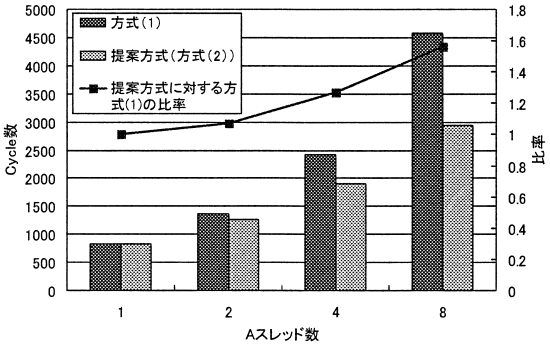

図 10 プロセス切替えサイクル数の比較
Fig. 10 Process switching cycles.

- 方式 (1) の実行サイクル数
  $= \mathbf{Ta} + \mathbf{Tb} + \mathbf{Tc} + \mathbf{Td} + \mathbf{Te} + \mathbf{Tf}$
- **Ta** = A スレッド数 1 の方式 (2) の処理（a）
- **Tb** =（<FWD> × 1 + <PBLK>）×（n − 1）
- **Tc** = {(<FWD>+<SW>) × 30 + <PDALL>}×(n − 1)
- **Td** = A スレッド数 n の方式 (2) の処理（e）
- **Te** = A スレッド数 1 の方式 (2) の処理（g）
- **Tf** = A スレッド数 1 の方式 (2) の処理（h）× n
  + {<PALLC> + <FWD> × 30 + <PUBLK>} × (n − 1)

方式 (1) の場合，A スレッド数 1 個のプロセス切替え実行サイクル数に対し，4 個の場合に約 3 倍，8 個の場合に約 5.5 倍の割合でサイクル数が増加している．一方，提案する方式 (2) の場合，A スレッドが 4 個の場合には約 2.3 倍，8 個の場合には約 3.5 倍の割合でサイクル数が増加している．結果として，方式 (1) の方が方式 (2) に比べて，A スレッドが 4 個の場合に 1.3 倍，8 個の場合に 1.6 倍の実行時間を要している．特に方式 (1) は **Tb**，**Tc**，**Tf** の処理サイクル数，A スレッド数 n にほぼ比例して増加するため，A スレッド数が増えるに従って方式 (2) に対する処

☆ OChiMuS PE のスレッド制御命令，プロセス切替え命令を利用可能にしたもの．
☆☆ 最適化オプション-O2 を設定した．

理性能差が大きくなる傾向を示している．定性的にも方式 (1) では，OS レベルの処理を実行中の A スレッドがある場合の対処などを含めたシステムソフトウェアの実装が複雑になることや，ハードウェアアーキテクチャで用意する必要がある専用命令や機能の量が多いことなどから，筆者らが提案した，プロセス内の A スレッドコンテキストを並行して退避，復帰するプロセス切替え方式 (2) の有効性を示している．

9. 関連研究

本研究で実現した，複数のハードウェアコンテキストを含むプロセスを OS で管理し，複数のハードウェアコンテキストを扱うプロセス切替え方式に関しては，ほかに提案例がなく，筆者らが初めての試みである．

従来 OS に関して，Solaris[8] では「軽量プロセス」というプロセス内スレッドを実行させるための仮想実行環境をカーネル・スレッドへ割り当てており，カーネルへ遷移する際には，軽量プロセスのハードウェアコンテキストを軽量プロセスのスタック上に退避・復帰して管理している．Mach[9] では，タスクの実行単位であるスレッドごとにハードウェアコンテキストを管理しており，スレッドはタスクへ割り当てられるアドレス空間や I/O 資源などを共通に利用する．また Linux[10] では，clone システムコールを用いて共通のアドレス空間や I/O 資源などを利用するプロセスを定義しており，カーネルレベルで個々のプロセスの実行コンテキストを管理している．いずれの従来 OS も，共通の資源を使う実行実体（軽量プロセス，スレッド，Linux 流プロセス）という定義はあるものの，プロセッサ資源への割当ては個々の実行実体であり，個々のハードウェアコンテキストを管理している．そのため，本研究で提案したように共通の資源を使う実行実体だけをマルチスレッドアーテクチャプロセッサ上で実行させるためには，カーネルレベルのスケジューラでアドレス空間を共有する実行実体を集約するといった方式が必要となる．文献 11) では，UNIX 流プロセスの集約を行うという研究が行われており，今後マルチスレッドアーキテクチャプロセッサへの適用も考えられている．しかし，従来 OS でマルチスレッドアーキテクチャプロセッサへ共通の資源を使う実行実体の集約を行えたとしても，従来の OS ではカーネルレベルで実行実体のプロセッサ割当てを制御しており，カーネル遷移のオーバヘッドがともなうこととなる．この点で，ユーザレベルで軽量なスレッド制御行う本研究のプロセス管理の方が，スレッドの生成，削除，同期の性能面でより有利であると考えている．

OS からの事象通知に関しては，いくつかの研究が報告されている[12)~15)]．これらの研究では，カーネルレベルで検出された事象をユーザレベルのスレッドスケジューラへ伝えるために，別途カーネル・スレッドを確保し，そのカーネル・スレッドのコンテキスト内に中断したスレッドのハードウェアコンテキストや事象内容を設定し，伝達するという方法をとっている．**Future** ではカーネルレベルで管理する仮想実行環境（すなわちカーネル・スレッドに相当する実行実体）がないため，カーネル・スレッドの確保を不要とし，さらに ThMB へ A スレッドコンテキストを直接退避するため，カーネル内部に一度退避する従来方式に比べて，A スレッドコンテキストコピー回数が少なく，効率の良い事象通知を実現している[5)]．

10. まとめ

筆者らは，マルチスレッドアーキテクチャプロセッサを従来 OS で管理する際の問題を提示し，その問題を解決するために，OS“**Future**” のプロセス管理に関する課題と解決方式について述べた．

第 1 に，マルチスレッドアーキテクチャプロセッサの複数のハードウェアコンテキストを含むプロセスを定義し，プロセス切替えの際に，複数のハードウェアコンテキストを退避・復帰しなければならない課題に対し，筆者らは，プロセッサ上の各計算実体（A スレッド）においてコンテキストの退避・復帰を並列に実行させる方法を提案し，実現した．

第 2 に，ユーザレベルでのより効率的なスレッド制御のために，OS 内で検出した事象をユーザレベルに通知するための処理実現という課題に対し，筆者らが提案している事象受信機構である Kernel Notification Handler の起動方法を明示し，プロセス切替えという事象の通知方法，および I/O リクエストにより発生する論理スレッドのブロッキング事象の管理と通知方法について明らかにした．

上記プロセス管理方式に関する実装を行い基本性能を評価し，プロセス切替え方式に関しては，**OChiMuS PE** 上の A スレッドが 4 個の場合には約 2.3 倍，8 個の場合には約 3.5 倍のサイクル数でプロセス切替えを実行できることを示した．

今後は様々な条件下での予備評価を行ったうえで，I/O ブロックを含むテストプログラムで本プロセス管理の有効性を確認する予定である．また，**Future** のメモリ管理機能について検討し，メモリ管理のオーバヘッドを含むシステムソフトウェアの評価を行いたいと考えている．

参 考 文 献

1) Tullsen, D.M., Eggers, S.J. and Levy, H.M.: Simultaneous multithreaing: Maximizing on-chip parallelism, *Proc. 22nd Annual International Symposium on Computer Architecture*, pp.392–403 (1995).
2) Lo, J.L., Barroso, L.A., Eggers, S.J., Kourosh, G., Levy, H.M. and Parekh, S.S.: An analysis of database workload performance on simultaneous multithreaded processors, *Proc. 25th Annual International Symposium on Computer Architecture*, pp.39–50 (1998).
3) 山崎真矢，本多弘樹，弓場敏嗣：マルチスレッドアーキテクチャにおけるデータキャッシュ構成方式の提案，情報処理学会研究報告 HPC-73-14, pp.79–84 (1998).
4) 佐藤未来子，河原章二，中條拓伯，並木美太郎：SOC 時代に向けた SMT 用 OS の構想，情報処理学会研究報告，Vol.2002, No.79, pp.31–38 (2002).
5) 笹田耕一，佐藤未来子，河原章二，加藤義人，大和仁典，中條拓伯，並木美太郎：マルチスレッドアーキテクチャにおけるスレッドライブラリの実装と評価，情報処理学会論文誌：コンピューティングシステム，Vol.44, No.SIG11 (ACS3), pp.215–225 (2003).
6) 河原章二，佐藤未来子，並木美太郎，中條拓伯：システムソフトウェアとの協調を目指すオン h シップマルチスレッドアーキテクチャの構想，コンピュータシステムシンポジウム，Vol.2002, No.18, pp.1–8 (2002).
7) Intel Technology Journal (Hyper-Threading Technology). http://developer.intel.com/technology/itj/2002/volume06issue01/index.htm
8) Jim, M. and Richard, M.: *Solaris Internals: Core Kernel Architecutre, 1st Edition*, Pearson Education Company (2001). 福本　秀，兵頭武文，細川一茂，大嶺朋之，佐藤　敬（訳）：Solaris インターナル，（株）ピアソン・エデュケーション (2001).
9) Tevanian Jr., A., Rashid, R.F., Golub, D.B., Black, D.L., Cooper, E. and Young, M.W.: Mach Treads and the UNIX Kernel: The Battle for Control, *Proc. Summer 1987 USENIX Technical Conference*, pp.185–197 (1987).
10) Daniel, P.B. and Marco C.: *Understanding the Linux Kernel*, O'Reilly & Associates, Inc. (2001). 高橋浩和，早川　仁（監訳）：詳解 Linux カーネル，オライリー・ジャパン (2001).
11) 近藤拓也，日下部茂：アドレス空間を共有するプロセスの集約を行う定数オーダースケジューリング，SACSIS2003 予稿集，pp.21–24 (2003).
12) Anderson, T., Bershad, B., Lazowska, E. and Levy, H.: Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism, *Proc. 13th ACM Symp. On Operating System Principles*, pp.95–109 (1991).
13) Marsh, B., Scott, M., LeBlanc, T. and Markatos, E.: First-class User-level Threads, *Proc. 13th Symp. On Operating Systems Principles*, pp.110–121 (1991).
14) 岡坂，清水，芦原，亀田：ユーザプログラムとカーネルの協調に基づくスレッドの設計と実現，情報処理学会論文誌，Vol.36, No.4, pp.913–924 (1995).
15) 猪原，益田：ユーザとカーネルの非同期的な協調機構によるスレッド切替え動作の最適化，情報処理学会論文誌，Vol.36, No.10, pp.2498–2510 (1995).

(平成 15 年 7 月 31 日受付)
(平成 15 年 11 月 5 日採録)

佐藤未来子（学生会員）

1966 年生まれ．1990 年東京農工大学大学院工学研究科修了．同年（株）日立製作所入社，サーバシステムの設計・性能評価等に従事．2002 年より東京農工大学大学院工学研究科博士後期課程に在学中．オンチップマルチスレッドアーキテクチャプロセッサ，オペレーティングに関する研究に興味を持つ．

笹田　耕一（学生会員）

1979 年生まれ．2003 年東京農工大学工学部情報コミュニケーション工学科卒業．現在，同大学工学研究科博士前期課程情報コミュニケーション工学専攻在学中．オペレーティングシステムやシステムソフトウェア，並列処理システム，言語処理系，プログラミング言語に関する研究に興味を持つ．

加藤　義人

2003 年東京農工大学工学部情報コミュニケーション工学科卒業．現在，同大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻在学中．プロセッサアーキテクチャに興味を持つ．

大和 仁典

2003 年東京農工大学工学部情報コミュニケーション工学科卒業．現在，同大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻在学中．オンチップマルチスレッドアーキテクチャ，メモリアーキテクチャに関する研究に興味を持つ．

河原 章二（正会員）

1978 年生まれ．平成 13 年東京農工大学工学部電子情報工学科卒業．平成 15 年同大学大学院工学研究科修了．同年 NEC シリコンシステム研究所に入社．プロセッサアーキテクチャ，並列処理システムに興味を持つ．

中條 拓伯（正会員）

1961 年生まれ．1985 年神戸大学工学部電気工学科卒業．1987 年同大学大学院工学研究科修了．1989 年神戸大学工学部助手を経て，現在，東京農工大学工学部情報コミュニケーション工学科助教授．1998 年より 1 年間 Illinois 大学 Urbana-Champaign 校 Center for Supercomputing Research and Development（CSRD）にて，Visiting Research Assistant Professor．プロセッサアーキテクチャ，並列処理，クラスタコンピューティング，高速ネットワークインタフェースに関する研究に従事．電子情報通信学会，IEEE CS 各会員．博士（工学）．

並木美太郎（正会員）

1984 年東京農工大学工学部数理情報工学科卒業．1986 年同大学大学院修士課程修了．同年 4 月（株）日立製作所基礎研究所入社．1988 年東京農工大学工学部数理情報工学科助手．1989 年電子情報工学科助手．1993 年 11 月電子情報工学科助教授．1998 年 4 月情報コミュニケーション工学科助教授．博士（工学）．オペレーティングシステム，言語処理系，ウィンドウシステム等のシステムソフトウェア，並列処理，コンピュータネットワークおよびテキスト処理の研究・開発・教育に従事．ACM，IEEE 各会員．