オペレーティング・システム 48－6
（1990. 9. 7）

オペレーティングシステムXEROのマルチスレッド機構

猪原茂和 加藤和彦 益田隆司

東京大学 理学部 情報科学科

ネットワークやマルチプロセッサ上の応用を実現する場合，1つのプログラムに複数の実行単位（スレッド）を与えるのが自然である．OSカーネル提供のマルチスレッド機構はシステムの持つ並列性を利用できる反面，高速なスレッド操作が難しいこと，上位層となる並列プログラミング言語との整合性がとりにくいこと，実行するプログラムが静的に決まっているのでスレッドを単位としたプログラミングができないことが問題となる．本稿で紹介するXEROのマルチスレッド機構では，ユーザーとカーネルの協調によるスレッド管理とスレッドプログラムの動的ロード機能によってこれらの問題点を解決し，高速で柔軟なマルチスレッドを提供する．

Muitiple Threads Mechanism in the XERO Operating System

Shigekazu Inohara Kazuhiko Kato Takashi Masuda

Department of Information Science,
Faculty of Science, University of Tokyo

A concurrent program needs the existence of multiple threads of control. A multiple threads facility provided by an operating system kernel enables programs to exploit parallelism of underlying hardware, but has following difficulties: cost of thread manipulation is high, implementing languages with different thread models is difficult, and writing thread-based programs is discouraged since programs are statically linked. This paper describes the multiple threads facility of XERO, in which user kernel co-operation in thread management and dynamic loading of thread programs are used to achieve programming flexibility and run-time efficiency.

1. はじめに

並列性のあるプログラムの記述とその実行のモデルとして，１つのアドレス空間に複数の実行単位（スレッド）を対応させるマルチスレッドモデルが提案されている．マルチスレッドモデルは共有メモリー型マルチプロセッサを仮想化したものと考えられ，並列計算機上ではもちろんシングルプロセッサマシン上でも，適切に実現すればハードウエアのもつ並列性を利用できる．またシングルスレッドモデルにおけるプロセス（１アドレス空間＋１実行単位）と比べると，実行単位を生成（消滅）するたびにアドレス空間の割り当て（解放）をする必要がないので，実行単位の生成・消滅が高速に行なえる．同一アドレス空間内のスレッドの通信と切り替えもプロセスより高速になる可能性がある．

本論文で我々は，オペレーティングシステムXERO[9][11][12]のマルチスレッド機構について述べる．このマルチスレッド機構を設計・実現するうえで目標としたのは，

- スレッドの並列性とスレッド操作（生成・消滅・切り替え・通信）の高速性を両立する
- スレッド単位のプログラミングを可能にする

の２点である．本章の残りの部分はこの２つの点の問題意識について述べる．

1.1 スレッドの並列性とスレッド操作の高速性

現在，マルチスレッド機構の実現形態としては２つの方法が一般的である．その１つはオペレーティングシステム，特にカーネルレベルでマルチスレッドを実現するものである．このようなものの例としては，早くは Thoth[4]，Cm* 上の StarOS[10]や Medusa[15]，最近では V[5]，Mach[1]，Amoeba[14]，Topaz[18]，Clouds[7] などが有名である．もう１つの実現形態は，ユーザーレベルのみでマルチスレッドを実現する方法である．こちらの例としては，Presto[3]，PCR[19] 等の言語独立なシステムと SunOS lwp[16]，Argus[13]，Eden[2] 等の言語システムがある．カーネルが提供するマルチスレッドとユーザーレベルのマルチスレッドの性質は，利用できる並列性とスレッド操作の高速性の点で対照的である．

カーネルが提供するマルチスレッドは，システムのもつ並列度（デバイス・プロセッサ間の並列度や，複数プロセッサの並列度）を最大限に利用できる．それに対して，ユーザーレベルのマルチスレッドでは，カーネルが関与していないのでハードウエアのもつ並列性を利用することは難しい．

一方，カーネルが提供するマルチスレッドは，スレッドの操作がユーザーレベルのマルチスレッドほど高速にできない．これは，ユーザー空間からカーネルへ制御が移る際のオーバーヘッド，カーネル内のデータ構造の割り当てや解放にかかるコスト，スレッドの切り替えに伴うアドレス空間切り替えの可能性に原因がある．ユーザーレベルのマルチスレッドでは，スレッド操作の大部分をカーネルとのやりとりなしにできるので，スレッドの操作を高速に行なうことができる．

このようにカーネルが提供するマルチスレッドはスレッド操作の高速性の面でユーザーレベルのマルチスレッドに劣り，逆にユーザーレベルのマルチスレッドは並列性の面でカーネルが提供するマルチスレッドに及ばない．これは，スレッド管理にカーネルが行なうべき部分とユーザーレベルで行なうべき部分があるためであると我々は考える．そこで我々のマルチスレッド機構では，ユーザーレベルのスレッド管理モジュールがカーネルと共同でスレッドの管理を行ない，スレッドの並列性とスレッド操作の高速性を両立する．

1.2 スレッド単位のプログラミング

我々のマルチスレッド機構の目標のもう１つは，スレッド単位のプログラミングを可能にすることである．

これまでに提案されているマルチスレッド機構では，スレッドが実行すべきプログラムはあらかじめリンクされたものに限られることが多い．このため，１つのアドレス空間で実行される複数のスレッドのコードを１つのまとまったプログラムとして書く必要があった．しかしスレッドプログラムを実行時にロード及びリンクする機能（動的ロード・動的リンク機能）があれば，１つ１つのスレッドを独立な単位としたプログラミングをすることができる．この

ことは，２つの場合に特に有効である．第１に，UNIX フィルターのような複数の汎用プログラムを組み合わせて高速に実行することが可能になる．第２に，１つのプログラムのモジュールをそれぞれスレッドとして実現しておくことで，外部環境の変化への対応や段階的なプログラム開発が可能になる．

1.1 節にあげたマルチスレッドの典型的な２つの実現形態のうち，カーネルが提供するマルチスレッドの多くは，動的ロード・動的リンク機能を提供していない．この理由の１つは，この種のマルチスレッド機構の主な目標が１つのプログラムの中の並列成分の実行であるためである．またカーネル自身でこれらの機能を提供する場合には，実行ファイル中のシンボルに関する情報によってカーネル内に保持するデータ量が増大してしまうという問題点もある．

ユーザーレベルのマルチスレッドでは，動的ロード・動的リンク機能を実現するうえでの障害は少なく，実際にこの機能を提供しているものもある．しかし，このレイヤーで実現された動的ロード・動的リンク機能は特定の言語や実行時環境に依存したものとなり，他の言語や実行時環境上で書かれたスレッドプログラムとの互換性が失われる可能性がある．複数の言語によるプログラミングは今日まであまり用いられてこなかったが，その重要性は以前から指摘されている[8][19].

我々は動的ロード・動的リンク機能をシステムの標準的な機能として提供して，スレッド単位のプログラミングを可能にする．特にスレッドプログラムの動的ロード・動的リンク機能を 1.1 節で述べたユーザーレベルのスレッド管理モジュールの機能の１つとすることで，特定の言語との独立性を保つと共に，カーネルへの負担をかけない実現ができる．

我々は，XERO のマルチスレッド機構を以上の２つの目標にしたがって設計・実現した．続く第２章ではこのマルチスレッドの実行機構の詳細，第３章では動的ロード・動的リンク機構の実現について述べる．第４章ではこれらをまとめ，今後の課題についてふれる．

2. XERO のマルチスレッドの実行機構

この章では，1.1 節で述べたカーネルとユーザーレベルのスレッド管理モジュールが共同で行なうスレッドの管理について述べる．まずカーネルとユーザーレベルのスレッド管理モジュールの２つがどのようにスレッド管理の実現を分担するかについて述べる．スレッド管理を２つの部分に分けることは，はじめに述べたスレッドの並列性とスレッド操作の高速性を両立させることができるほか，様々な並列プログラミング言語をこのマルチスレッド機構の上で実現するという観点からも好ましい．

マルチスレッド機構の設計上の問題となる点はいくつかあるが，そのなかで，システムコールをはじめとするカーネルとのやりとりの方法，プレエンプティブなスケジューリングの実現法，そしてアドレス空間を越える同期的な通信を効率的に行なう方法について議論する．

2.1 スレッド管理機能の分担

我々のマルチスレッド機構では，カーネルとユーザーレベルのスレッド管理モジュールが役割分担をしてスレッド管理にあたる．スレッド管理の分担は，カーネルがユーザーレベルでは効率よくできない（または全くできない）ような最も基本的な機能だけを与え，残りの部分はすべてユーザーレベルで行なう，という基本方針で行なう．

具体的にはカーネルは，仮想化されたプロセッサ（仮想プロセッサと呼ぶ，図中では VP と略す）とそのスケジューリングの低レベル部分，ユーザー空間（タスクと呼ぶ）の間の通信を提供する．仮想プロセッサは，共有メモリー上の複数のプロセッサを仮想化したもので，１つのタスクに対して複数の仮想プロセッサを割り当てることができる．仮想プロセッサはカーネルが物理プロセッサを時分割して作るもので，複数の仮想プロセッサを用いることによりハードウエアのもつ並列性を利用できる．

各タスクにおかれたユーザーレベルのスレッド管理モジュール（タスクスーパーバイザーと呼ぶ）は，カーネルの提供する部品化された機能を用いてマルチスレッド完全な機能を実現する．タスクスーパーバイザーは与えられる仮想プロセッサを再度スケジュールすることでそのタスク上のスレッドを実現する．タスクスーパーバイザーはそのタスク内のスレ

ッドの生成と消滅，切り替え，スレッド間の同期と通信を行なう．これらの操作はアドレス空間の切り替えなしに行なえるので高速に実現できる．このようにカーネルとユーザーレベルの2つのレイヤーが共同でスレッド管理を行なうことによって，スレッド操作の高速性と並列性の両立することができる．

スレッド管理の上位レイヤーをユーザーレベルで行なうことは，ユーザーレベルの機能を容易に変更できることを示す．このことは，複数のマルチスレッドのモデル（スケジューリングポリシー，同期・通信のプリミティブ）を持つ並列プログラミング言語を実現するのに有効である．カーネルが提供するマルチスレッドでは，カーネルがある特定のマルチスレッドのモデルを与えてしまう．このため，カーネルが与えたモデルに適合する言語は自然に実現できるが，そうでない言語は実現することが困難になるという問題があった[17]．このことは，そのシステム上で自然に実現できる応用の範囲を狭めることになる．我々のマルチスレッド機構ではタスクスーパーバイザーを変更することで，複数の言語のスレッドモデルに柔軟に対応することができる．

2.2 システムコール処理・例外処理の設計

ユーザーレベルのみでスレッドを実現する場合のように，カーネルとユーザーが全く協調なしで2段階のスレッドスケジューリングを行なうと，カーネルとのやりとり（システムコールや，ページフォールトを含む例外処理）がカーネル中でブロックした場合に，ユーザーレベルのスレッドの並列度が下がるという問題が生じる．この問題はシングルスレッドモデルのオペレーティングシステムでは特に問題となる．即ち，システムコール等がブロックしている間はタスク内のすべてのスレッドの活動が休止状態になってしまうのである．

これを解決するためにはカーネルとユーザーの協調が必要であり，それには3つの方法が考えられる．第1の方法は，カーネル内でブロックする際にユーザーに対してUPCALL[6]を行なって，ユーザーの対処を促す方法である．ユーザーはUPCALLによって呼び出されたルーチンのなかで，新たな仮想プロセッサを生成することができる（図1）．第2の

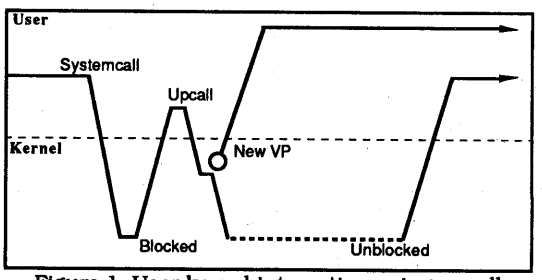


Figure 1. User kernel interaction using upcalls

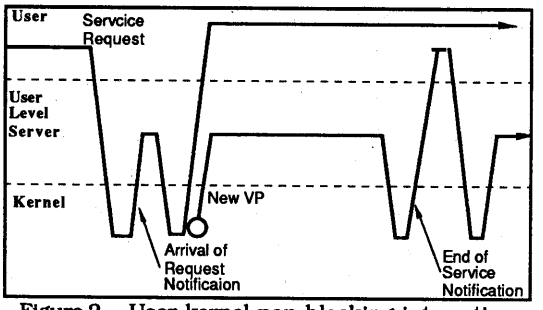


Figure 2. User kernel non-blocking interaction with notifications

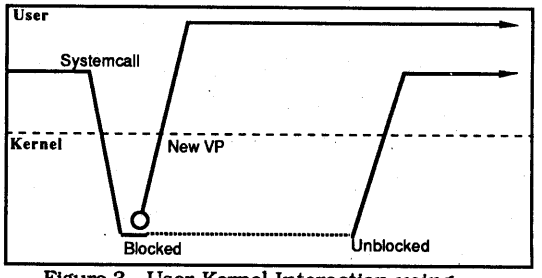


Figure 3. User Kernel Interaction using Automatic VP Creation

方法は，カーネル内の処理をブロックしないよう作ることである．通信を完全に非同期的にすることでこの効果が得られる．何らかの同期が必要な場合には，ソフトウエア割り込みを用いた通知(notification)を用いる（図2）．第3の方法，これは我々が提案する方法であるが，システムコール等がカーネル内でブロックする際にユーザーの指定に応じて新たな仮想プロセッサを生成する．ユーザーは新たに作られた仮想プロセッサを用いて他のスレッドの実行を続けることができる（図3）．我々がこの方法を用いる理由は以下の通りである．

第1の方法は，UPCALLの実現コストが問題になる．UPCALLを1回行なうのに伴って，カーネルとユーザーの間の制御の切り替えを2回行なうこ

とになる．また，UPCALL 自体は一般的かつ強力な概念であるが，この問題では UPCALL の主な目的は新たな仮想プロセッサの生成である．そこで，UPCALL の代わりにカーネル内で直接新しい仮想プロセッサを生成する第3の方法は，第1の方法をこの問題に関して最適化したものと考えることができる．

第2の方法をとると，カーネルの提供する概念は非常に単純になるが，現実にはカーネル内の処理を完全にブロックしないようにするのはそれほど簡単でない．例えばデバイスの操作は特権モードでしか行なえないのが普通なので，カーネルがそれを行なうのが最も自然で効率も高い．特権モードで動作するサーバーを用意する方法も考えられるが，こうすると要求を出すクライエントタスクとサーバータスクのアドレス空間の切り替えが通信の度に起こって効率がよくない．

2.3 システムコール処理・例外処理の実現

我々のマルチスレッドでは，システムコールや例外処理がカーネル内でブロックした場合に，ユーザーレベルの並列度を下げないために新たな仮想プロセッサを要求することができる．仮想プロセッサを生成するかどうかの指定とそれに伴うカーネルとユーザーの間の情報交換を行なうために，2つのものを用いる．1つは，個々のシステムコールの際にそのシステムコールの動作を決定する変数 Create_VP_On_Block で，（スタックを介して渡すシステムコール自体の引数とは別に）レジスタを介してカーネルに与える．もう1つは，カーネルとユーザーの間に設けた共有領域で，例外処理の動作を決定する変数 Create_VP_On_Intr とカーネルからユーザーへプロセッサコンテクストを渡すための領域に分かれる．

変数 Create_VP_On_Block と Create_VP_On_Intr はフラグとして働く．システムコール（または例外処理）が起動されたときに対応する変数の値が真なら，カーネル内の処理がブロックした時点で新たな仮想プロセッサが生成される．この新たな仮想プロセッサは，起動されるとタスクスーパーバイザー内のスレッドをスケジュールするルーチンから実行を開始する．カーネルはその時に共有領域をつかって，システムコール（例外処理）の時点でのプロセッサコンテクストをユーザー側に渡す．この情報はどのスレッドが停止したのかをタスクスーパーバイザーが知るのに使われる．

カーネルとユーザーの間の共有領域中のプロセッサコンテクストを渡す領域をいつ，またいくつ確保するかは，実現上の問題として考慮しなくてはならない．この領域を1つだけにすると，複数の仮想プロセッサが同時にカーネルからユーザー側に入ろうとした場合にボトルネックになる．仮想プロセッサの数に応じてこの領域を確保すればこの問題は生じない．しかし仮想プロセッサが生成・消滅する度にこの領域の割り当て・解放の処理を行なうことになり，仮想プロセッサの生成・消滅にかかるコストが上がってしまう．

実際には，仮想プロセッサはカーネルが物理プロセッサを時分割して作りだすものであるから，実際に起こりうる共有領域への同時アクセスは物理プロセッサの数を越えない．そこで，タスク内で動作可能な物理プロセッサの数だけの共有領域をタスク生成時に割り当てておくことができる．この方法では，タスクスーパーバイザーに物理プロセッサを意識させなければならないが，共有領域の排他制御も動的割り当ても必要ないので効率上は最も優れている．我々の実現では，システムコール等のブロックに伴って新たな仮想プロセッサがユーザー側に渡された時，仮想プロセッサは仮想プロセッサ ID と物理プロセッサ ID をレジスタに持っている．物理プロセッサ ID は共有領域のスロットを特定するのに用いる．仮想プロセッサ ID は，共有領域の中に保存されているプロセッサコンテクストがタスク上のどのスレッドのものかを知るのに用いる．

2.4 プレエンプティブなスケジューリングの実現

このマルチスレッド機構では，ユーザーレベルのマルチスレッドでは実現が難しいプレエンプティブなスレッドスケジューリングを自然に行なうことができる．我々の行なう2レベルのスレッド管理に伴って，仮想プロセッサ単位とスレッド単位の2つのプレエンプションが考えられる．

仮想プロセッサ単位のプレエンプションは，カーネルが物理プロセッサを時分割して仮想プロセッサを作り出すときに行なうものであり，ユーザー側の意志とは関係なく行なわれている．このレベルのプレエンプションは通常ユーザーには見えない．

スレッド単位のプレエンプションは，仮想プロセッサ単位のプレエンプションのたびにユーザーにスレッドを切り替える機会を与えることによって実現できる．これは，ユーザー側の指定により行なうか行なわないかを選択できる．スレッド単位のプレエンプションを行なう場合には，仮想プロセッサがタイムアウト後に再び活動を開始する時点で，タスクスーパーバイザーに制御を移す．タイムアウトの時点で保存されたプロセッサコンテクストは，システムコールの場合と同様にカーネルとユーザーの間の共有領域を使ってタスクスーパーバイザーに渡す．

このようにして，仮想プロセッサ単位のプレエンプションとスレッド単位のプレエンプションをどちらも実現できる．

2.5 タスク間の通信に伴うオーバーヘッドの回避

タスク内のスレッド間通信は，そのタスクのタスクスーパーバイザーが実現しており，その形式は完全にユーザー任せである．それに対して，タスクを越える通信についてはカーネルが提供する必要がある．我々のマルチスレッド機構では，カーネルはRPC型のメッセージ通信をタスク間の通信プリミティブとして提供する．

この同期的なメッセージの管理は，単純に行なうとシステムの資源を浪費する恐れがある．というのは，メッセージは，特にネットワーク上では頻繁に使用され，それに伴って多くのスレッドがメッセージの相手を待ってブロック状態にはいることが予想されるからである．仮想プロセッサはカーネル内のスタックをはじめとするいくつかのメモリー資源をもっているが，それらはブロック状態にある間使用されることなく保持される．他のカーネルサービスの多くと違って，メッセージは同期が成立しないかぎり終了しないので，このようにして保持されるメモリー資源はシステム全体のメモリーの使用効率を下げる原因になる．

この問題を解決するために，メッセージ通信の同期の時に起こるブロックについては他のブロックとは異なる処理を行なう．同期が即座に成立したものについてはそのままカーネル内の処理を続けるが，そうでないものについては，仮想プロセッサのその時点での継続（continuation）をスタックからテーブル上の情報に変換して保存し，一旦その仮想プロセッサの資源を解放する．そして，同期が成立した時点で改めて資源を割り当てて仮想プロセッサの活動を再開する．こうすることで，メッセージの同期に伴うブロック中，メモリー資源を無駄にすることを防ぐことができる．

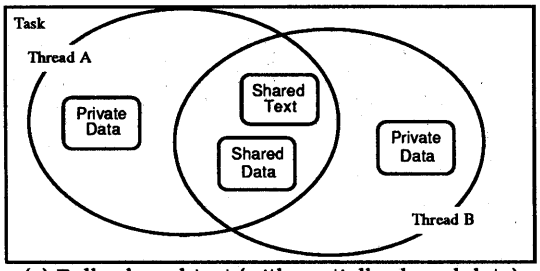
3. スレッドの動的ロード・動的リンク

1.2節で述べたように，スレッドプログラムの動的ロード・動的リンク機能は，ユーザーレベルのスレッド管理モジュールであるタスクスーパーバイザーが実現する．動的ロード・動的リンクはスレッドプログラミングの自由度を増すということ以外に，もう1つの利点がある．それは，これらの機能の実現方法によってはタスク内外でテキストやデータの共有が可能になることである．これによってシステム全体のメモリーの使用効率を上げることができる．このため，動的ロード・動的リンクの実現法を考える場合，どのようなメモリー共有が可能かは重要な選択基準となる．

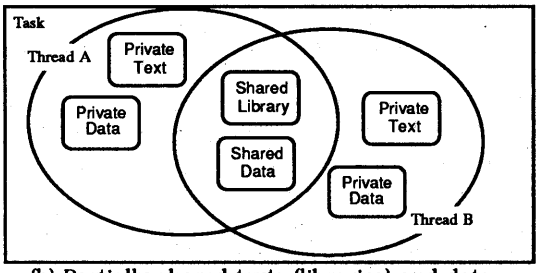
3.1 動的ロード・動的リンクによるメモリー共有

動的ロード・動的リンクによって可能になるメモリー共有には2種類ある．

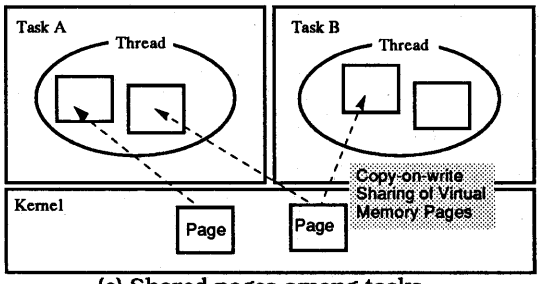
1つはタスク内でのメモリー共有である．新たに生成されるスレッドの実行プログラムが自タスク内にない場合，タスクスーパーバイザーはカーネルにそのファイルを要求し，適当な再配置処理を行なった後スレッドを割り当てる．既に実行ファイルがタスク内にある場合（即ち同じファイルを実行中のスレッドがすでにある場合）には，そのファイルを古いスレッドと新しいスレッドが共有して使用する．この場合，複数のスレッドが1つのテキストを完全に共有することになる（図4 (a)）．さらに，いくつかのスレッドが部分的なテキスト（ライブラリ）や



(a) Fully shared text (with partially shared data)



(b) Partially shared texts (libraries) and data



(c) Shared pages among tasks

Figure 4. Possible memory sharing as a result of dynamic loading and dynamic linking of thread programs

データを選択的に共有する場合もある（図4 (b)）．

もう1つは複数のタスクにまたがる仮想空間ページの共有である．動的ロードのためにカーネルからユーザー空間に読み込まれるプログラムは，通常書き込み禁止のテキスト（コード）部分と書き込み可能なデータ部分にわかれる．実行ファイルのうちテキスト部分は copy-on-write の機構によって，ページ単位の共有が可能になる（図4 (c)）．この場合にはそのページに書き込みが起こらないことが条件となる．

3.2 動的ロード・動的リンクの実現

読み込まれたモジュールの再配置の方法としては，(1) ロード位置固定，(2) テキスト書き換え，(3) ベースポインタ相対，(4) 固定テーブル相対，そして (3) と (4) を組み合わせた新たな方法として (5) 浮動テーブル相対，が考えられる．これらの5つの方法は必要な処理（ロード時点での処理とシンボル参照ごとのコスト）とメモリーの共有能力がそれぞれ異なっている．

(1) の方法は，オブジェクト生成の時点でそのオブジェクトがロードされるべき位置を決定してしまう．これは，オブジェクトの生成をシステム全体の取り決めにしたがって行なうか，衝突が起きないようなオブジェクトのみに動的ロードを制限することによって可能になる．この方法ではロード時点での再配置にかかる処理もなく，シンボル参照も絶対アドレスによって行なえる．共有 (a) と (b) はこの方法ではできない．特に同一のタスク内で同一のプログラムを2つのスレッドとして実行することもできない．共有 (c) はできる．

(2) の方法は，読み込んだテキストの内容を再配置情報にしたがって書き換える．この方法を行なうためには，再配置情報をオブジェクトファイル内に保存しておく必要がある．ロード時点での処理はテキスト中でのシンボルの使用数に比例する．ロード処理の完了後は，シンボルの参照を絶対アドレスによって行なえる．共有 (a) と (b) は制限付きで可能だが，共有 (c) はテキストページに対して書き込みが起こるために不可能になる．

(3) の方法は，テキストセグメント・データセグメントの先頭をそれぞれのベースレジスタで指しておき，すべてのシンボル参照をこれらのレジスタ相対で行なうものである．この方法では，ロード時の処理は2つのレジスタの初期化のみである．シンボルの参照はレジスタ＋オフセットによって行なわれる．共有 (a) と (c) が可能である．

(4) の方法は，各シンボルの絶対アドレスを保持したテーブルを介した間接参照を行なう．これは，SunOS の Shared Library の実現に用いられているもので，テーブルの再配置ができないためライブラリの共有にしか使えない．ロード時のコストは，シンボルの数に比例してかかる．シンボル参照は，テーブルを介したメモリー間接となる．共有 (b) と (c) が

可能である．

我々の用いた (5) の方法は，(4) の方法で問題となったテーブルの再配置をレジスタによって行なうものである．ロード時の処理は (4) と同じくシンボルの数に比例してかかる．シンボル参照のコストも (4) と同じで，テーブルを介したメモリー間接となる．この方法でのみ，共有 (a), (b), (c) がすべて可能となる．実現にかかるコストは (1), (2), (3) に比べて大きい．特に，シンボル参照は常に2回のメモリーアクセスを必要とする．しかし仮想空間ページが複数のアドレス空間で共有されることによってスワップのアクセスが少なくなることで，実行時の総合的な効率低下はそれほどないと予想している．

4. おわりに

本稿では，スレッドの並列性とスレッド操作の高速性をあわせもち，また，柔軟なスレッドプログラミングが可能な XERO のマルチスレッド機構について述べた．このマルチスレッド機構は，ユーザーレベルとカーネルレベル共同のスレッド管理と，スレッドプログラムの動的ロード・動的リンク機能という2つの特徴をもつ．この2つの特徴は，従来の典型的なマルチスレッド機構の問題点である，スレッドの高速な操作と並列性を両立することが難しいこと，複数のプログラミング言語に対するサポートが困難であること，そして，スレッドとして実行するプログラムがコンパイル時に決まっているためにスレッドをプログラミングと実行の基本単位とできないことを解決するためのものである．

現在，カーネルとタスクスーパーバイザーの実装を進めている．今後の課題としては，4.3BSD プロセスや Mach スレッドとの性能比較を行なうこと，スレッドの動的ロード・動的リンク機能を用いた本格的なプログラムの開発，タスクスーパーバイザーによる複数の言語サポートを行なうことなどがある．

参考文献

[1] Accetta, M., Baron, R., et al., "Mach: A New Kernel Foundation for UNIX Development", Proceedings of the 1986 Summer USENIX Conference, pp. 93-112, Summer 1986.

[2] Almes, G. T., Black, A. P., et al., "The Eden System: A Technical Review", IEEE Transactions on Software Engineering, vol. 11, no. 1, pp. 43-59, Jan. 1985.

[3] Bershad, B. N., Lazowska, E. D., et al., "An Open Environment for Building Parallel Programming Systems", Proceedings of ACM/SIGPLAN PPEALS 1988 Parallel Programming: Experience with Applications, Languages and Systems, SIGPLAN Notices, vol. 23, no. 9, pp. 1-9, Sep. 1988.

[4] Cheriton, D. R., Malcolm, M. A., et al., "Thoth, a Portable Real-Time Operating System", Communications of the ACM, vol. 22, pp. 105-115, Feb. 1979.

[5] Cheriton, D. R., "The V Distributed System", Communications of the ACM, vol. 31, pp. 314-333, Mar. 1988.

[6] Clark, D., "The Structure of Systems Using Upcalls", Proceedings of the 10th ACM Symposium on Operating Systems Principles, pp. 128-140, Dec. 1985.

[7] Dasgapta, P., LeBlanc Jr., R. J., et al., "The Clouds Distributed Operating System: Functional Description, Implementation, Details and Related Work", Proceedings of the 8th International Conference on Distributed Computing Systems, pp. 2-9, Jun. 1988.

[8] Einarsson, B., "Mixed Language Programming", Software-Practice and Experience, vol. 14, no. 4, pp. 383-395, Apr. 1984.

[9] 猪原，加藤，益田，"分散OS XERO におけるスレッドによるプログラム実行について，" 情報処理学会 OS研究会報告, vol. 89, No. 94, Nov. 1989.

[10] Jones, A. K., Chansler, R. J., et al., "StarOS, a Multiprocessor Operating System for the Support of Task Forces", Proceedings of the 7th Symposium on Operating Systems Principles, pp. 117-127, 1979.

[11] 加藤，猪原，脇田，益田，"データベース処理を指向した分散オペレーティング・システム XERO の構想，" 電子情報通信学会コンピューターシステム研究会, CPSY-98-29, pp. 87-92, 1989.

[12] 加藤，千葉，猪原，益田，"無指向オペレーティングシステム XERO における複数言語間のメッセージ通信と永続オブジェクトアクセス方式"，電子情報通信学会コンピューターシステム研究会, CPSY-90-49, Jul. 1990.

[13] Liskov, B., Curtis, D., et al., "Implementation of Argus", Proceedings of the 11th Symposium on Operating Systems Principles, pp. 111-122, Nov. 1987.

[14] Mullender, S. L., Tanenbaum, A. S., "The Design of a Capability-Based Distributed Operating System", Computer, vol. 29, no. 4, pp. 289-299, 1986.

[15] Ousterhout, J. K., Scelza, D. A., and Sindhu, P. S., "Medusa: An Experiment in Distributed Operating System Structure", Communications of the ACM, vol. 23, no. 2, Feb. 1980.

[16] Sun Microsystems, "Lightweight Process Library", SunOS Reference Manual, SunOS Release 4.0, 1988.

[17] Scott, M. L., "The Interface between Distributed Operating System and High-Level Programming Language", Proceedings of the 1986 International Conference on Parallel Processing, pp. 242-249, Aug. 1986.

[18] Thacker, C. P., and Stewart, L. C., "Firefly: A Multiprocessor Workstation", IEEE Transaction on Computers, vol. 37, no. 8, pp. 909-920, Aug. 1988.

[19] Weiser, M., Demers, A., Hauser, C., "The Portable Common Runtime Approach to Interoperability", Proceedings of the 12th Symposium on Operating Systems Principles, pp. 114-122, Dec. 1989.