オペレーティング・システム　45 － 3
（１９８９．１１．１０）

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

猪原茂和　加藤和彦　益田隆司
東京大学 理学部 情報科学科

現在多くのオペレーティング・システムでは，１つの実行環境に１つの実行単位を割り当ててプログラム実行を行なう．この方法では，頻繁に通信し合うプログラム群を効率よく実行できないため，１つの環境に複数の実行単位（スレッド）を与える実行モデルが考えられている．現存するスレッド機構では，タスク生成時に実行プログラムが決定されるため，自由なスレッド間の結合が妨げられるという問題点がある．これを解決するためのプログラムの動的ロード機構，及び大規模なプログラムの動的ロードに伴って必要となるアドレス空間上の部分的な保護機構を持った，高速かつ柔軟なスレッド機構について述べる．

ON PROGRAM EXECUTION UNDER THE MULTI THREAD MECHANISM OF XERO

Shigekazu Inohara Kazuhiko Kato Takashi Masuda

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

7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Most of operating systems execute a program under separately protected environment. Since this style of program execution takes significant time for each context switching, it is impossible to efficiently execute a set of co-operating programs. To overcome this inefficiency, mechanisms for assigning multiple threads of control for each environment have been proposed. This paper discusses a design of an efficient and flexible thread mechanism with the feature of a dynamic loading facility of executable programs and an access protection mechanism inside the address space to facilitate dynamic loading of large applications.

1. はじめに

現在の多くのオペレーティング・システムでは，応用プログラムは各々保護されたアドレス空間上で，それぞれ独立の実行環境とただ1つの実行コンテクストを与えられて実行される．これは，1つのシーケンシャルなプログラムによって記述された応用を実行するためには妥当な方法である．これに対し，一つの大きな処理を複数の実行単位が協調して行なうプログラムが存在する．このようなものの例としては，ウインドウ・システム等のクライアント・サーバー型の応用や，UNIX のシェル上で日常的に使用される複合的なフィルター（いくつかのフィルタープログラムをパイプによって組み合わせたもの）などがあげられる．しかしこれらに限らず，現在単独の実行単位として構成されているプログラムでも，複数の協調する実行単位の集まりとして記述することで，より高い並列性を抽出し，プログラムの部品化を進める事ができる．

このようなスタイルのプログラムを実行しようとすると，1空間＝1環境＝1実行単位のプロセスによる実行モデルでは，各々の独立に取られた環境（アドレス空間，オープンファイル情報，シグナル情報，ユーザーアカウント情報，処理中のシステムコール情報，システムコール用カーネルスタック，共有テキストセグメント情報，他）を切り替える処理が頻繁に起こることになり，実行効率が悪い．そこで考えられるのが，1つの環境の中に複数の実行単位を置いて，環境の切り替えを省くことである．StarOS[5] の task force, Thoth[2] や V[3], Amoeba[9] の light-weight process, Mach[1] の thread 等は，このような観点から導入されたものである．

現在研究中の XERO[6][7] では，これまでデータベース分野と OS 分野で別々に研究が進んできた二次記憶上の情報の管理に関する成果を融合させ，データベース応用のための環境を提供することを目標としている．データベース処理として典型的な関係データベースの問い合わせ演算（projection, join 等の基本演算を組合せたもの）は複数の実行単位が協調する例の1つであり，このような処理を高速に実行するための枠組みをシステムとして提供することは大切である．このような観点から XERO では，1つのタスク空間上に複数のスレッドをおいてプログラム実行を行なう．

現在実現されているスレッド機構の多くは，タスク空間の生成時にあらかじめロードされたプログラムテキストに対して，複数の実行コンテクストを与えるものである．例えば SunOS 上のウインドウシステム SunNeWS のサーバーの場合，独自のスレッド機構を持っており，クライアントとの対話をする特定のプログラムがスレッドとして起動される．この方法は実行すべきスレッドのプログラムがあらかじめ分かっている場合には使えるが，上であげたデータベースの問い合わせや UNIX の複合的なフィルターなどのように，実行時に初めて実行すべきプログラムが決定する場合には使用が難しい．この問題に対応するために XERO では，スレッドのテキストの動的（実行時）ロードを可能にしている[4]．

動的ロードの過程では，まずユーザープログラムを一切含まないタスク空間を生成しておき，スレッドのテキストは必要に応じてファイルからロードする．このようにしてスレッドの生成を行なえば，前述の関係データベースの問い合わせ処理やパイプによる複合的なフィルターも実行時に自由に構成することができる．もう一つの例として，XERO では二次記憶上のデータに対して複合オブジェクトの概念に基づいた管理を行なうが，各複合オブジェクトの操作のためには対応するメソッドを呼び出す必要がある．この場合も，各メソッドをスレッドとして提供しておき必要になったオブジェクトのメソッドを実行時にロードして呼び出すことができる．

本稿では，上述のような特徴を持つ XERO のタスク・スレッドの管理，そしてそれに関連する仮想記憶管理について述べる．2章では，タスク空間，スレッド，スレッド間通信についての各概念，スレッドのスケジューリング法について述べる．3章では，スレッドの動的ロード機構について，

そして4章ではスレッド間の共有変数・共有手続きについて述べる．続く5章では多くの応用プログラムを動的ロードした場合にそれらの間で必要となる，タスク空間内のアクセス保護について説明する．そして6章では，現在の状況と今後の課題について述べる．

2. XERO のプログラム実行モデル

2.1 スレッドによるプログラム実行の概要

はじめに述べたように，XERO ではプログラムの実行をその実行環境であるタスク空間と，その中に存在する複数の実行コンテクストであるスレッドに分けて管理する．スレッド同士は遠隔手続き呼び出し型の通信によって情報の交換を行なって処理を進める．システムの提供するサービスは，カーネルのほかユーザー空間のサーバーとして多く存在しており，それらに対するサービス要求もすべてメッセージによって行なわれる．

2.2 タスク空間とタスク・スーパーバイザ

タスク空間は，カーネルへのメッセージ new_task(host_id) によって指定のホスト上に生成される．生成されたタスクにはネットワーク内で一意なタスクID が付けられ，new_task メッセージの応答（reply message）として生成を要求したスレッドに返される．この時点でタスクはユーザースレッド（及びそのテキスト）を持たず，タスク内のスレッドの生成，消滅，スケジューリング，メッセージの管理を行なうタスク・スーパーバイザだけを含んでいる（図1）．次節で述べるように，タスク・スーパーバイザはユーザースレッドの管理をするためにカーネルと協調動作を行う．

2.3 スレッドのスケジューリング

スレッドのスケジューリング方式は，スレッド機構の実現上の1つの重要な選択点である．スケジューリングの方法としては，大別して2つある．1つはすべてのスレッドのコンテクスト情報をカーネルが管理する方法，もう1つは，カーネルはタスクレベルの情報を管理し，各タスクが自分の持つスレッドの情報を管理する方法である．前者の場合，メッセージ等によるスレッド切り替えの度にカーネルへ入って処理を行なうことになり，後者に比べると切り替えの速度が劣る．また，現在のプロセスによる実行環境とちがって，スレッドの数は普通でも数百のオーダーになると思われるので，カーネル内のスケジューリング情報が大きくなって，システム全体の性能に影響を与える恐れがある．前者の主な利点は，マルチプロセッサへの対応が自然にできる点にある．現在商業的に成功している共有バス型マルチプロセッサマシンでは，同時に複数のプロセスにスケジュール可能となるよう UNIX のスケジュール方式を拡張しているものが幾つかある．Mach では，これをタスクとスレッドの整理された概念により実現している．

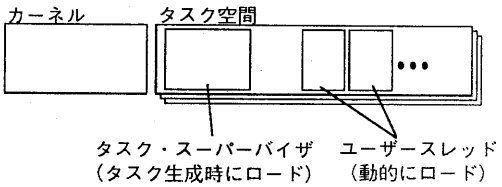

図1. タスク空間の構成

XERO ではスレッドのスケジューリング方式をより高速で，ユーザーに対する柔軟性に富む（ユーザーがスレッド管理に参加する事ができる）と考えられる後者の方法を基礎にしてスレッド機構を設計した．ここで，後者の方法を純粋に実現すると2つの欠点が生じる．1つはシステムコール（XEROでいえば，カーネルへのメッセージ）を発行することによってタスク内のすべてのスレッドがブロックされること，もう1つは，全体をコルーチン的にスケジュールするので，1つのスレッドが制御を独占してしまえることである．第1点は，並列度をあげるために是非とも解決したい問題である．第2点に関しても，XERO では比較的大規模な応用プログラムを自タスク上にロードして動作させることを考えているので，preemptive なスケジューリングを実現して，1つのスレッド

がプロセッサを独占しないようにしたい．

メッセージを複数許すことは，カーネルとタスク・スーパーバイザによる割り出し，及び割り出しからの復帰の処理を工夫することで解決する．また，プロセッサの独占を防ぐ問題は，外部から送られる割り込み処理（タイマー割り込みによるタイムアウト，デバイスからの割り込みなど）を利用して，スレッドをタスク同様に preemptive に動作させることで解決する．

まず，割り出し処理について考える．タスク空間から発せられる割り出しとしてまず考えられるのは，メッセージ転送に伴うトラップ命令である．メッセージの伝送は，タスク・スーパーバイザとカーネルが協同で行なう．あるタスク空間内でのスレッド間通信は，そのタスク空間のタスク・スーパーバイザが解決する．

２つのタスクにまたがるスレッド間通信では，メッセージはユーザースレッドから，ローカルなタスク・スーパーバイザ，カーネル，リモートのタスク・スーパーバイザを通じて，最終的に目的のスレッドに送られる．ユーザースレッドからメッセージを受取ったタスク・スーパーバイザは，メッセージ伝送のための特別なトラップ命令を発行する．このトラップ命令は例外処理を終了してカーネルから戻る時に，トラップを発生した位置に戻るのではなく，スレッドを再スケジュールするタスク・スーパーバイザ内のルーチン sched に制御を渡す．そして sched ルーチンが次にスケジュールするスレッドを決定し，そのスレッドに制御を渡す．

１つのタスクから同時に複数のメッセージを発行するためは，１回目のメッセージをカーネルが受取った後でも，そのタスク空間の動作可能なスレッドに制御を渡すことができる必要がある．この目的で，タスク・スーパーバイザはカーネルに送るメッセージにタスク状態フラグ（「まだ自分のタスク内に動作可能なスレッドが残っているか」をカーネルに教える）を加え，カーネルのスケジューリングを助ける．このフラグがオンになっていれば，カーネルは１つ以上メッセージを発行中のタスクも，動作可能なタスクとしてスケジューリングの対象として扱う．カーネルは，動作可能なタスクの中から，必要に応じてタスク空間内を調べてのスレッドに関する情報を得，制御を渡すタスクを決定する．メッセージを１回以上発行しているタスクに制御を渡す場合，そのタスク空間の sched ルーチンに制御を与える際に，どのメッセージもまだ応答が無いことを伝える．そこで，sched ルーチンはこれを意識して動作させるスレッドを決定し，制御を渡す．この方法で，１つのタスク空間から同時に複数のメッセージを発行する事ができる（図2）．

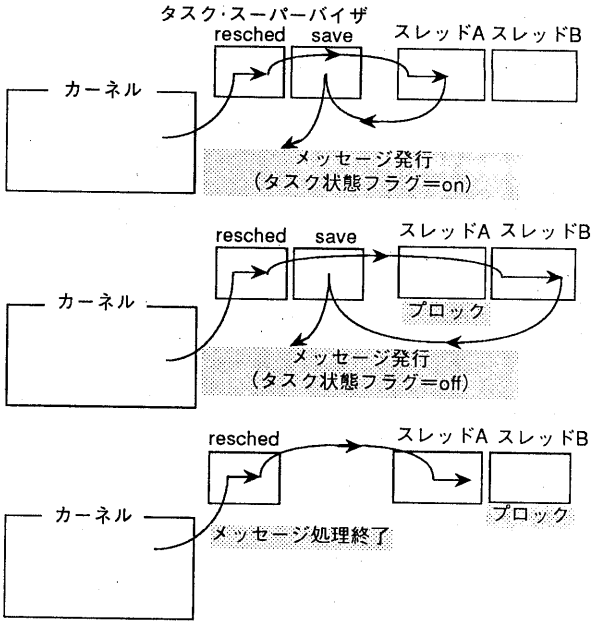

図2.メッセージの多重化

また，割り出しの別の要因として，０による割算の様なエラー処理が考えられる．このような場合もカーネルによる例外処理の終了後に，sched に制御を渡し，その結果を伝えるので，sched はスレッドのスケジューリングを行うことができる．

次に，割り込み処理について考える．割り込みがおきると，その時実行中だったスレッドのコンテクスト情報（プログラムカウンタ，スタックポインタ，プロセッサの状態レジスタ，汎用レジスタ群，及びその他の，割り込みからの復帰用に必

要なプロセッサ状態）がカーネルの中に保存されて，割り込み処理が開始される．その後，割り込み処理が終了して，ふたたびそのタスク空間に制御を渡す場合には，中断したスレッドのコンテクスト情報を sched に伝えて，sched が中断したスレッドを再開するか，別のスレッドを動かすかを決定できる．

これをクロック割り込みの例を考えてみると，タスクレベルでの preemptive なスケジューリングを利用してスレッドのスケジューリングも preemptive にすることができる．一旦タイムアウトしたタスク空間に制御を戻す場合，上記の要領で sched ルーチンを呼んでタスク空間内のスレッドの再スケジュールを行なう．この方法で，XERO では同一のスレッドが連続して動作するのを回避することができる（図3）．

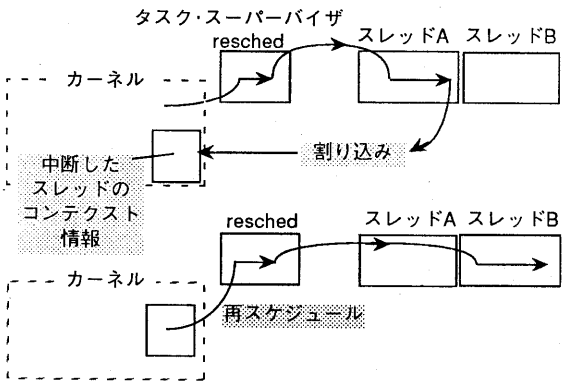

図3. 割り込み処理によるスレッドスケジューリング

3. スレッドの生成：動的ロード機構

3.1 スレッドの生成

スレッドの生成はタスク・スーパーバイザへのメッセージ new_thread(task_id, region, file) によって行なわれる．メッセージの中で指定されたタスク上のタスク・スーパーバイザが new_thread メッセージを受け取ると，指定されたスレッドの実行ファイルをタスク空間上に読み込み，実行時リンクによってスレッド間の共有変数及び共有ライブラリ（これらについてはこの後述べる）を結合した後，スケジューリングの対象に加える．new_thread メッセージへの応答はスレッドID で，これはそのスレッドをネットワーク上で一意に決定する．

3.2 動的ロードのための実行ファイル

動的ロード可能なスレッド機構を実現するためには，各スレッドの実行ファイルの中では絶対アドレスによるデータや手続きの参照が許されない．実行ファイルがロードされる位置は実行時に決定され毎回異なるので，テキスト及びデータ部分については動的ロードの度に再配置を行なわなくてはならない．このため，変数や手続きの参照関係を再配置情報として実行ファイルに保存しておき，参照するアドレスやオフセットを調整することになる．再配置を行うための最も直接的な方法としては，テキストやデータのうち変更が必要な部分を直接書き換える方法が考えられる．しかしこの方法では，一般に仮想記憶の共有率がよい copy-on-write による memory mapped file によるローディングを行っても，大半のページが書き込みを受けて共有不能になってしまう．これは，システム全体の1次時記憶の使用効率を下げるので，システム側から見ると望ましくない．これに対して，再配置が必要になるようなアクセスを変数へのオフセットをおさめたテーブルを介した間接アドレッシングにするようなコードを生成しすれば，実行時の再配置での書き換えがそのテーブルのあるページに集中するので，他の大部分のページは書き込みを免れて共有を続けられる．XERO では，コンパイラ及びリンカはスレッドの実行ファイルを生成する際に，このようなテーブルを持ったオブジェクトファイルを作る．

3.3 共有ライブラリの実現

このテーブル参照型の再配置可能な実行ファイル形式は，SunOS 4.0 の共有ライブラリ[11] のためのファイル形式をもとにして設計し，後に述べるスレッド間の共有変数のための拡張を行なったものである（図4）．このため，この実行ファイル形式で共有ライブラリもサポートできる．共有ライブラリとは，ライブラリに対するコンパイ

ル時の処理は名前の解決をおこなってテーブルのエントリを作るだけにとどめ，実行時に実行時リンカを起動して最終的なリンクを行なってプログラムのロードイメージを作るものである．共有ライブラリを用いると２つの利点がでる．１つはよく利用されるライブラリルーチンが，数多くの実行ファイルにコピーされて二次記憶を圧迫するのをさけられること，２つめは実行時のライブラリのコードはいくつかのスレッドから共有されうるので，システム全体の一次記憶の使用効率が上がることである．

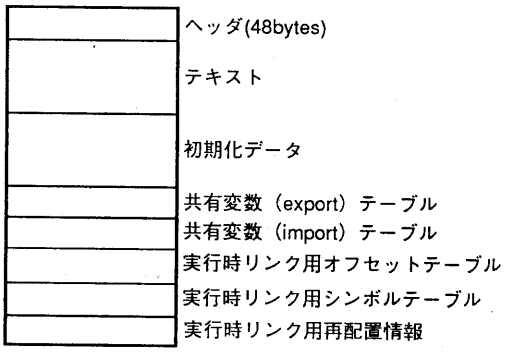

図4. スレッドの実行ファイル形式

4 スレッド間共有変数

4.1 スレッド間共有変数の必要性

同一タスク内に存在するスレッド群は，アドレス空間を共有しているので，ポインタによる直接参照や，関数ポインタのやりとりによって常に自由な情報の参照や交換ができる．しかし，このような方法を無制限に用いることは，プログラムの安全性を低下させると共に，コードの可読性を損なう原因となる．また，コンパイル時のチェックのためには，プログラムで明示的に共有変数の宣言をする必要がある．このため，スレッド間においても共有変数や共有手続きを導入することが望まれる．特に，密接に関連して仕事をするスレッドの間では，すべてをメッセージ通信によって処理するよりもデータ構造を共有したほうが，効率及び記述の自然さの両面から有利である．

4.2 スレッド間共有変数の記述

スレッド間共有変数のプログラム中での宣言は，C言語に追加した予約語 shared で行なう．例えば，

```
shared int        i;
shared extern int j;
```

とすると，変数 i と j を共有変数として宣言し，i については自スレッド内に領域を確保する指定となる．i および j の参照関係の解決は，実行時に，タスク・スーパーバイザへのメッセージ link_symbols(thread_id) を用いて行なう．このメッセージで，タスク・スーパーバイザはメッセージを発行したスレッドの共有変数テーブルに引数で指定されたスレッドの共有変数テーブルを結合する．この操作によってスレッド間の変数の共有が開始され，これ以降は自由に共有変数へのアクセスができるようになる(図5)．

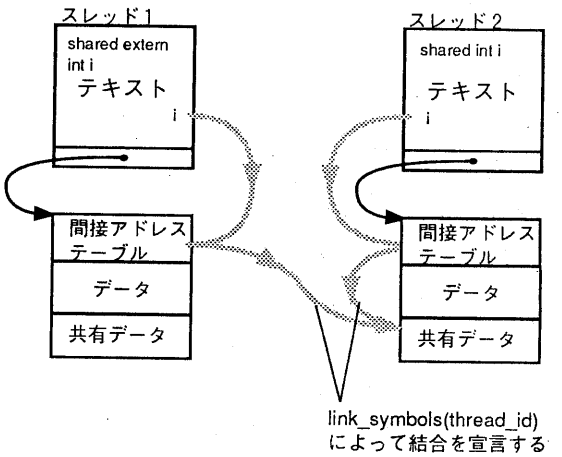

図5. スレッド間共有変数

5. スレッドのための仮想記憶管理

5.1 タスク空間内の保護：リージョン(region)

XERO では，スレッドの動的ロード機構を用いて，自タスク内に大規模な応用プログラムやシステムの提供するサービスを読み込んで実行することを想定している．これは，スレッドの高速性を利用して，ユーザーが起動した多くの実行単位間の交信や生成，消滅を高速に行い，システムの性能の向上を狙うためである．

このような状況下では，タスク空間の全体を保

護なしで用いると，以下の様な問題が生じる．

(1) タスク空間の安定性を保つことは難しい．

(2) タスク空間内で起こるすべてのアクセスを許すと秘密保護が不可能になって，自タスク内にロードできる応用の範囲を狭める．

(3) 2章で述べたようにタスク・スーパーバイザは性能及び並列性を引き出すために，カーネルとの協調動作が多く存在する．つまり，タスク・スーパーバイザには他のスレッドから勝手にアクセスされてはならない部分が存在する．タスク・スーパーバイザを保護してその動作を保証するうえでも，タスク空間内の空間保護が必要である．

そこで，スレッドの持つ様々な高速性をできるかぎり損なわずに，最低限の空間保護を行なう機構が必要になる．XERO ではこのような要請に応えるため，タスク空間内にリージョンと呼ぶ空間保護の単位を設ける．リージョンは，システム既定の大きさの連続したアドレス空間（現在のバージョンでは 4Mbytes）で，リージョン毎の属性として4つのレベル（システム設定時に固定）を設定することができ，数字の小さい方が権限が強い．これらのレベルは，new_thread によってそのリージョンに作られるスレッドのレベル（これは各々のスレッドの実行ファイルに与えられる属性である）のうち，最も権限の低いものの値をとる．一般ユーザーはプログラムに 2と3を，スーパーユーザーはさらに 0と1も付けることができる．

リージョンは所有者（そのリージョン内から）のアクセスとその他（そのリージョン外から）のアクセスに対して，それぞれ保護属性をつけることができ，これらはページ単位の保護属性との AND を取って使用される（図6）．リージョンに与えられる保護属性は read と write である．

強い属性を持つリージョンに属するスレッドは，自分より弱い属性のリージョンの保護属性を越えてその中の情報にアクセスする事ができる．この様な，多レベルのメモリー保護は，Multics においてハードウェアの機能を用いて導入されたリングレベルプロテクション[10]をページベースのメモリー管理ユニットの上で実現したものといえる．現在の多くの仮想記憶ハードウェアは Multics が用いたようなレベル付きの保護を提供していないので，リージョンの保護はページアクセス違反の検出にともなう例外処理によってエミュレートする．

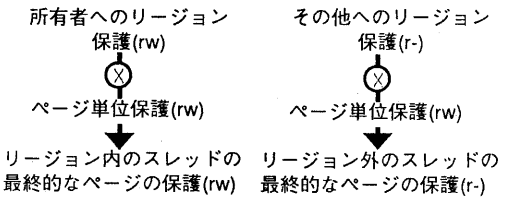

図6. リージョン保護の例

5.2 リージョン操作とスケジューリング

タスク空間は，生成の時点ではただ1つのリージョン（タスク・スーパーバイザのためのリージョン）のみを含んでいる．その他のリージョンはタスク生成後にユーザーによって作られる．リージョン操作のためのカーネルへのメッセージとしては，new_region, protect_region, destroy_region がある．

リージョン境界を越えるスケジューリングが起こる場合，リージョン保護の変更を行なうためにカーネルへのメッセージを発行することが必要になる．1回のリージョン操作に対するカーネル内の処理量は，数個のページテーブルエントリの書き換えが必要になる程度である．これは，XERO のリージョンがかなり大きな連続ブロックであり，開発マシンである MC68030 のメモリー管理ユニットの機能によってページテーブルを木構造にすることができることによる．

これはそれほどコストの高い処理ではないが，スレッド間のコンテクスト切り替えを管理するタスク・スーパーバイザ全体をリージョン保護しようとすると，タスク空間内のコンテクスト切り替えがすべてカーネルへのメッセージ通信を必要とすることになって，効率の低下が予想される．そこでタスク・スーパーバイザを2分し，ユーザースレッドすべてからアクセス可能な部分とリージ

ョン間で保護されるべき情報を含んだ部分を作る．リージョン内のスレッド情報については，そのリージョン内に置くことにして，そのリージョン内のスレッドにはスレッド管理情報を公開しつつ，他のリージョンのスレッドからは保護することにする．このようにすると，リージョン内でのコンテクスト切り替えはカーネルへの切り替えを一切含まずに行なえる．さらにリージョンの境界を越えるスレッド間のコンテクスト切り替えを高速に行う例外処理機構についても検討中である．

6. 現在の状況と今後の課題

現在，Sony NEWS-1750（現在のターゲットマシン）上でカーネル部分とスレッド管理部分の開発を行っている．今後の課題としては 4.3BSD のプロセスとの性能比較を行うことがあげられる．また，カーネル内部の記述の独立性を高め，再コンフィギュレーションを容易にするために各種のカーネル提供のサービスをスレッドとして実行する「カーネルのスレッド化」を行いたい．

参考文献

[1] Accetta, M., Baron, B., et al., "Mach: A New Kernel Foundation for UNIX Development," proc. USENIX 1986 Summer Conf., pp.93-112, Summer 1986.

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

[3] Cheriton, R. D., "The V Distributed System," Comm. ACM, vol. 31, no. 3, pp. 314-333, Mar. 1988.

[4] 猪原，加藤，益田，”データベース処理を指向した分散OS XEROにおけるスレッドの動的ロード機構，”情報処理学会第39回全国大会論文集，1989.

[5] Jones, A. K., Chansler, R.J., et al., "StarOS, a Multiprocessor Operating System for the Support of Task Forces," Proc. 7th Symp. Oper. Syst. Prin., pp-117-127, 1979.

[6] 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システム研究会 並列処理に関する「指宿」ミニシンポジウム，pp. 87-92, 1989.

[7] 加藤，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 の設計，”情報処理学会第39回全国大会論文集，1989.

[8] Motorola, Inc., MC68030 Enhanced 32-bit Microprocessor User's Manual, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[9] Mullender, S. L., and Tannenbaum, A. S., "The Design of a Capability-Based Distributed Operating System," Comput. J., vol.29, no. 4, pp.289-299, 1986.

[10] Graham, R. G., "Pretection in an Information Processing Utility," Comm. ACM, vol.5, no.3, pp.365-369, 1968.

[11] Sun Microsystems, Inc., SunOS 4.0 Reference Manual, Sun Microsystems, Inc, 1988.