既存のプログラムコードをマルチスレッド環境で実行する方式とその実現

安倍広多[†] 松浦敏雄[†]
安本慶一[††] 東野輝夫[†††]

マルチスレッドは，複数の処理を並行して効率良く実行させるための機構である．スレッド間の通信やコンテクストスイッチは，プロセス間のそれと比較して高速であることが知られている．しかし，マルチスレッドプログラムでは，複数のスレッドが並行して同じデータ領域にアクセスする可能性を考慮する必要があるため，単一スレッドを前提に書かれた既存のプログラムやライブラリの大部分は，マルチスレッドプログラムの一部として利用することは容易ではなかった．本稿では，マルチスレッドの高速性を保ちつつ，この問題を解決する一手法を提案する．通常のマルチスレッドプログラムでは，仮想アドレス空間にはただ1つのプログラムイメージ（実行ファイルをメモリ中に配置したもの）が存在し，すべてのスレッドがコードやデータを共有するが，提案方式では，仮想アドレス空間中に複数のプログラムイメージを配置し，それぞれのイメージごとに1つ以上のスレッドを動作させる．イメージ間でコードやデータは独立しているため，単一のスレッドのみが動作するプログラムイメージ内では，既存のプログラムコードを変更せずに利用できる．実際にこの方式をUNIX上に実装し，評価を行った．その結果，既存のコードの利用が容易であること，提案方式のスレッドは，従来のプログラムのスレッドと同等の効率で動作すること等を確認した．

A Method to Execute Existing Program Codes in Multi-threaded Environments and Its Implementation

Kota Abe,[†] Toshio Matsuura,[†] Keiichi Yasumoto[††] and Teruo Higashino[†††]

Multi-threading is an efficient mechanism to execute multiple tasks concurrently. Multi-threading is known to be faster than multi-processing because interprocess communication and interprocess context switching can be eliminated. However, it is not easy to use most existing single-threaded programs and libraries as parts of a multi-threaded program because multiple threads might access the same data concurrently. This paper proposes a method to resolve this problem without losing the performance of multi-threading. In ordinary multi-threaded programs, only one program image exists and all threads share code and data. Using our method, multiple program images can exist in a single virtual address space and one or more threads can run in each program image. Since codes and data are separated among these program images, existing program codes can be used without any modifications in a program image in which a single thread runs. We have implemented and evaluated this method on UNIX. We have experimentally confirmed that it is easy to use existing codes through this method and that a thread in a multiple-image program runs as efficiently as a thread in an ordinary program.

1. はじめに

単一の計算機内で協調して並行に処理を行う方法として，複数のプロセスを用いる方法（マルチプロセス）と，複数のスレッド（プログラム実行の制御の流れ）を用いて1つのプロセスの中で処理を行わせる方法（マルチスレッド）がある．一般に，プロセス間通信やプロセス間コンテクストスイッチのオーバヘッドのため，マルチプロセスよりもマルチスレッドの方が実行効率が高い．しかし，マルチスレッドプログラムの作成においては，プログラムコードが複数のスレッドから呼ばれても安全（スレッドセーフ）であるように，

† 大阪市立大学学術情報総合センター
Media Center, Osaka City University
†† 滋賀大学経済学部
Faculty of Economics, Shiga University
††† 大阪大学大学院基礎工学研究科
Graduate School of Engineering Science, Osaka University

細心の注意を払う必要がある．一般に，単一スレッドを前提とした既存のコード（**Legacy コード**と呼ぶ）のほとんどはスレッドセーフではないため，既存の膨大なソフトウェア資産（プログラムおよびライブラリ等）をマルチスレッドから利用することは容易ではなく（2 章で詳述），簡単に利用するための方法も提案されていない．このため，従来 Legacy コードを活用して並列アプリケーションを作成するには，マルチスレッドを諦め，マルチプロセスでアプリケーションを構成するしかなかった．

一方，マルチプロセスでの実行効率を改善するための方法として，複数のプロセスを同一アドレス空間に配置する方法（以下，本稿ではプロセスグループ機能と呼ぶ）がいくつかの OS によって提案・実装されている[1),2)]．同一アドレス空間内のプロセス間コンテクストスイッチは，TLB フラッシュを避けることができるため，アドレス空間を切り替える通常のプロセス間コンテクストスイッチよりも効率が良い．また，プロセス間通信も効率的に行える[2)]．この機能を利用すると，同一アドレス空間で複数のプロセスを動作させることができるため，マルチプロセスを用いながらもマルチスレッドと似たような環境を作ることができる．

しかし，プロセスグループ機能は，UNIX 等の一般に広く普及した環境では利用できないため，既存のプログラムコードの再利用を考慮した場合，実用的とはいいがたい．また，プロセスグループ機能はあくまでも複数のプロセスを同一アドレス空間で実行する機能であるため，カーネルがプロセスを管理する必要がある．このため，マルチスレッドよりも実行効率が落ちる（たとえば，コンテクストスイッチはカーネルを経由する必要がある）．

そこで本研究では，上記のプロセスグループ機能とは異なり，カーネルが管理する単一のプロセスのアドレス空間内で既存のプログラムコードを利用可能とするマルチスレッド実行方式を提案する．通常のマルチスレッドプログラムでは，実行するプログラムコードやアクセスするデータはすべてのスレッドによって共有されるが，本稿で提案する方式は，カーネルが管理する 1 プロセスのアドレス空間内に複数のプログラムイメージ（実行ファイルをメモリ中に配置したもの）を配置し，それぞれのイメージを 1 つ以上のスレッドで実行するというものである．イメージ間でコードやデータは共有しないため，単一のスレッドが動作するイメージ内では Legacy コードを問題なく利用できる．それぞれのイメージの管理はユーザレベルで行えるため，ユーザレベルスレッドや 2 レベルスレッド機構と組み合わせて，効率的な実行が可能である．この方式を **ProcThread** と呼ぶ．この方式はユーザ空間で実装可能であり，普及している既存の OS 上で利用できるため実用性が高い．

以下，2 章では，既存のプログラムコードをマルチスレッド環境で実行するうえでの問題を整理し，3 章で提案する ProcThread 方式の概要を述べる．さらに，4 章で UNIX 上での実現方式を詳述し，5 章で実装の概略を述べる．最後に 6 章で本方式の評価を，7 章で応用について触れる．

2. 既存のプログラムコードをマルチスレッド環境で実行する上での問題

2.1 マルチスレッド環境での Legacy コードの実行

マルチスレッドプログラミングを行う際，既存の Legacy コードを容易に利用できるとプログラムの生産性が向上する．しかし，1 章で述べたように，マルチスレッドプログラムでは，コードがスレッドセーフであることを保証する必要があるため，Legacy コードを単純にマルチスレッドプログラムとリンクし，呼び出すことはできない．この問題への対処としては，以下の方法が考えられる．

(1) Legacy コードの関数を呼び出す際に排他制御を行う．
(2) Legacy コードを修正し，大域的なデータへのアクセス個所に排他制御を追加する．
(3) Legacy コードを修正し，スレッド固有データ（TSD）☆を利用して Legacy コードがアクセスする大域的なデータをスレッドごとに独立させる．

しかし，これらの方法には以下の問題がある．(1) は Legacy コードを複数のスレッドから同時に実行できないため，Legacy コードの実行に時間がかかる場合（I/O 待ち等），並列性が落ち，マルチスレッドの利点を生かすことができない．(2) はソースコードの精査を要し，一般には困難な作業である．(3) の TSD には 2 つの方式がある．値を TSD に格納（または TSD から取得）するための専用の関数を利用するものと[3)]，スレッドごとに異なった値を格納できる特殊な大域変数を利用するものである[4),5)]．前者は TSD 上の変数にアクセスするたびに関数呼び出しを必要とするため，煩雑でオーバヘッドがある．後者はコンパイラが関与する必要があり，標準的な機能ではないため，利

☆ Thread Specific Data（Thread Local Storage とも呼ばれる）．スレッドごとに固有のデータを格納できるようにするための機能[3)]．

用可能な環境が限定される．また，(2)，(3)にはソースコードが必要であり，商用のライブラリ等，ソースコードが公開されていない Legacy コードには適用できない．

2.2 マルチプロセスからマルチスレッドへの移行

UNIX 等の既存の OS の下で開発されたソフトウェアでは，複数のプロセスが協調して動作するものも多い．このようなソフトウェアを効率良く動作させるためにマルチスレッド化したいという要求があるが，これも簡単ではない．多くの場合，各プロセスを構成するそれぞれのプログラムが同一の Legacy コードライブラリをリンクしているため，単純にマルチスレッド化すると前述のスレッドセーフの問題が発生する．さらに，プロセス間通信とスレッド間通信のインタフェースは異なるため，通信処理部分を修正する必要もある．このため，マルチプロセスからマルチスレッドへの移行コストは高い．

3. ProcThread 方式の概要

通常のマルチスレッドでは，1 つのアドレス空間に単一の実行ファイルのイメージが配置され，その中で複数のスレッドが動作するが，ProcThread 方式では 1 つのアドレス空間に，複数の実行ファイルのイメージを配置し，それぞれのイメージを 1 つかそれ以上のスレッドで実行する．つまり，1 つのアドレス空間中で複数のプログラムが動作することになる（**図 1**）．本方式で利用する実行ファイルを **μ 実行ファイル**，実行中のプログラムを **μ プロセス**と呼ぶことにする．μ プロセスは通常のプロセスと異なり独立したアドレス空間を持たず，所属するプロセスのアドレス空間の一部を使用して動作する☆．μ プロセス間でコードやデータは共有しない．プロセスグループ機能を用いた場合と同様，それぞれの μ プロセスは，他の μ プロセスとアドレス空間を共有しているが，プロセスグループ機能と異なり，それぞれの μ プロセスをカーネルが管理する必要はない．

3.1 本方式の特徴

本方式の特徴は以下のとおりである．

- 同一の Legacy コードを含む μ プロセスを複数生成しても，それらの間でコードは共有されないため，単一のスレッドで動作する μ プロセスではコードがスレッドセーフである必要はなく，Legacy コードを問題なく利用可能である．Legacy コード呼び出しのたびに排他制御する必要はなく，並列性を上げることができる．また，Legacy コードのソースコードの修正は不要であり，ソースコードが入手できない場合も利用可能である．
- 複数のスレッドが動作する μ プロセスからは Legacy コードライブラリを直接呼び出すことはできないが，後述する μ プロセス間プロシージャコールを利用すると，このような μ プロセスからも Legacy コードライブラリを利用できる（7.1 節参照）．
- 通常のプロセス間の保護の壁は μ プロセス間には存在しないため，通常のマルチスレッドと同等の実行効率が得られる．
- μ 実行ファイルは，通常の実行ファイルと同じ手順，手間で作成できる．この際，特殊なコンパイラは不要である．
- 既存の単一スレッドプログラムは，リンクし直すだけで μ 実行ファイルとすることができる．複数のプロセスで構成された既存のアプリケーションは，プロセス生成処理を修正するだけで単一アドレス空間内で動作させることが可能である．ProcThread は，一部のプロセス間通信（プロセス間パイプ）をスレッド間通信に置き換える機能を備えており，プロセス間通信処理を修正する手間を削減することができる．実際に，UNIX のフィルタ系のコマンドをまったく修正せずに μ プロセスとして実行可能であることを確認している（6.5 節参照）．

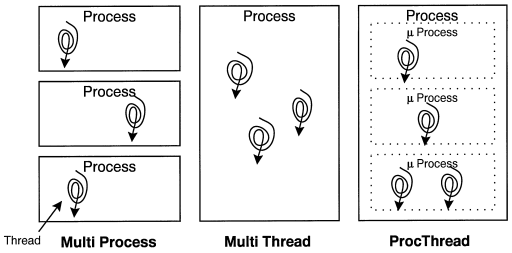

図 1 各方式の比較
Fig. 1 Comparison of each method.

3.2 本方式の制約

本方式には以下の制約，欠点がある．

- コードを共有しないため，通常のマルチスレッドよりメモリを消費する．ただし，マルチプロセスと比べるとほぼ同程度の消費量である．
- μ プロセスの生成は，スレッドの生成よりも時間を要する．
- 本方式は，Legacy コードを使って複数のスレッ

☆ 本稿では「プロセス」をカーネルが管理する実体としてのプロセス，「アドレス空間」をカーネルがプロセスごとに管理する仮想アドレス空間の意で用いる．

ドに同一のデータにアクセスさせたいような場合は適用できない（たとえば，Legacy コードのリスト処理ライブラリを使って，複数のスレッドが同一のリストにアクセスできるようにすることはできない）．このような場合は，従来どおり排他制御を行う必要がある．

- 基本的に，あるコードを実行することで影響が及ぶ範囲は，当該コードを実行している μ プロセス内である．このため，ある μ プロセスがプロセス全体に影響を与えることはできない．たとえば，ファイル記述子は μ プロセスの資源としたため（4.4.1 項で後述），オープンしたファイル記述子を複数の μ プロセスで共用することはできない☆．
- プロセス ID を利用している Legacy コードでは問題が発生する可能性がある．たとえば，プロセス ID を一時ファイルや名前付きパイプ等のファイル名の一部として利用している既存の単一スレッドプログラムを同一プロセス内で複数実行した場合，複数の μ プロセスが同一のプロセス ID を共有するため，ファイル名が衝突する可能性がある．
- プロセスの外部から特定の μ プロセスにシグナルを送信できないため，シグナルを利用する Legacy コードの利用は制約を受ける．

4. ProcThread の実現方式

ここでは，UNIX 上での ProcThread の実現方式について述べる．

ProcThread 方式では，プロセスのアドレス空間内に，μ プロセス管理等のサービスを提供する ProcThread コアと，1 つ以上の μ プロセスが配置される．それぞれの μ プロセスには，それを実行するスレッドが存在する．このように，ProcThread では，プロセスのアドレス空間内に，小さな OS（ProcThread コア）と，1 つ以上の小さなプロセスを配置するような構造になっている（**図 2**）．

初期状態では，プロセスのアドレス空間には，ProcThread コアと最初の μ プロセス（初期 μ プロセスと呼ぶ）が配置される．その後，μ プロセスはスレッドや，他の μ プロセスを必要に応じて生成しながら動作する．以下，詳細を述べる．

4.1 ProcThread コア

ProcThread コアは，プロセスのアドレス空間中に

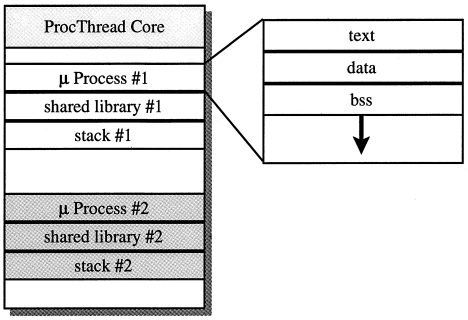

図 2 アドレス空間のレイアウト
Fig. 2 Address space layout.

1 つ存在し，以下の機能を提供する．

- μ プロセス管理（生成・消滅等）
- スレッド管理（生成・消滅・スケジューリング等）
- μ プロセス間通信（μ プロセス間プロシージャコール，パイプ，ファイル記述子管理等）

スレッドは，これらの機能にアクセスするために **ProcThread システムコール**を発行する．ProcThread システムコールは，プロシージャコールとして実装できるため，オーバヘッドは少ない．これら以外のサービスは，UNIX カーネルが提供する．

4.2 μ 実行ファイル

μ 実行ファイルの作成は，通常の実行ファイルの作成と同じように，ソースコードをコンパイル，リンクすることで行う．UNIX の実行ファイルは，固定された仮想アドレスに配置されるが，μ 実行ファイルはどのアドレスに配置されても動作する必要がある．このためには，(1) ファイルに再配置情報を記録する，(2) 位置独立コードを用いる，の 2 つの方法が考えられるが，一般に既存の静的ライブラリは位置独立ではない．ソースコードが入手不能な Legacy コードの静的ライブラリをリンクできるようにするため☆☆，ProcThread では (1) の方式を用いる．再配置情報はオブジェクトファイルや静的ライブラリから抽出することが可能である．

μ 実行ファイルは，UNIX の実行ファイルと互換性があるため，ProcThread 独自の機能（μ プロセス間プロシージャコール等）を使用していなければ，シェルから実行したり，通常のデバッガを用いてデバッグすることも可能である．

4.3 μ プロセス

μ プロセスは，μ 実行ファイルから生成され，μ プ

☆ この問題は，確保した資源を μ プロセス間で受け渡しする機構を導入することである程度解決できる．

☆☆ 一般に，位置独立コードを得るためにはソースコードからコンパイルし直す必要がある．

ロセスIDで識別される．既存の単一スレッドプログラムを μ プロセスとして容易に利用するため，μ プロセスの実行環境は，UNIXプロセスの実行環境とほぼ同じになるようにしている．

μ プロセスごとに独立したメモリ領域で動作させるため，μ プロセスにリンクされるライブラリは，動的リンクされるものを含めて μ プロセスごとに存在するようにした．

4.3.1 μ プロセスの生成と消滅

μ プロセスの生成は，μ 実行ファイルのパス名と，引数（argv形式），環境変数，属性等を指定して行う．μ 実行ファイルがプロセスのアドレス空間に配置され，それを実行するスレッド（初期スレッド）が1つ生成される．初期スレッドは，μ プロセスのスタートアップルーチンから実行を開始する．

スレッドが_exitを呼び出すと，当該スレッドが所属する μ プロセスは消滅する．Legacyコードでは，プロセス終了時に利用していた資源が消滅することを前提に，確保したプロセスの資源（オープンしたファイル記述子，mmapによってマップした領域等）を明示的に解放しない場合が多いため，ProcThreadの_exitでは，μ プロセスが確保したこれらの資源の解放も行う．これにより，Legacyコードを μ プロセスで利用しても，資源の解放処理を追加する必要はない．

4.3.2 μ プロセスでのマルチスレッド

ProcThreadでは，μ プロセス内で複数のスレッドを動作させることも可能である．ProcThreadコアは，POSIXスレッド[3]のインタフェースを提供する．スレッドは，特定の μ プロセスに所属する（ただし，後述する μ プロセス間プロシージャコールを使って一時的に他の μ プロセスのコードを実行することは可能）．_exitの呼び出し等で μ プロセスが消滅する場合，所属しているすべてのスレッドも消滅する．μ プロセスで複数のスレッドを動作させる場合，通常のマルチスレッドプログラムと同様，コードがスレッドセーフとなるように注意する必要がある．

このような μ プロセスからLegacyコードライブラリを呼ぶ方法については7.1節で述べる．

4.4 μ プロセス間通信機構

ProcThreadでは μ プロセス間通信機構として，パイプ機構と，μ プロセス間プロシージャコール機構を用意している．μ プロセス間の共有変数は存在しないため，μ プロセス間の通信にはこれらの機構を使用する必要がある．

4.4.1 パイプ機構

ProcThreadのパイプ機構は，UNIXのプロセス間パイプと同様，μ プロセス間にパイプを張って通信を行うものである．UNIXのパイプと同じように，read，write等でアクセスできる．UNIXのパイプは，writeされたデータをカーネル内にいったんコピーし，read時に再度コピーすることによって実装されている（2コピー）が，ProcThreadのパイプはプロセス内で動作するため，以下のように1コピーで実現できる．

あるスレッド（W）がパイプにwriteした場合，他のスレッド（R）がreadするまで，W を待たせる（逆に，先にスレッドがreadを発行した場合，writeするスレッドが現れるまで待たせる）．readが発行されると，W は，書き込むデータをreadで指定する領域にコピーする．

さらに，ProcThreadのパイプの読み書きにはUNIXカーネルを介する必要がない．これらの理由で，ProcThreadのパイプは高速に動作する．

また，UNIXのフィルタコマンド等，標準入出力（ファイル記述子0，1）をパイプで接続し，通信に使用するプログラムは数多い．これを修正することなく，μ プロセスとして利用するため，ProcThreadではファイル記述子は μ プロセスの資源とした．これにより，たとえば μ プロセス1のファイル記述子0と，μ プロセス2のファイル記述子0が別のものを指すことができるようになる．これは，ProcThreadコアが，UNIXカーネルが管理するプロセスのファイル記述子との間で変換を行うことによって実現する．これによって，μ プロセスの標準入出力を，ProcThreadのパイプに割り当てることができるため，シェルのパイプラインのように，μ プロセス間をパイプで接続し，通信することができる．この機能により，UNIXのフィルタコマンドを μ プロセスとして取り込み，利用することが簡単に行える．

4.4.2 μ プロセス間プロシージャコール

μ プロセス間での通信を簡単に行うため，μ プロセス間プロシージャコール機構を用意した．μ プロセスは，他の μ プロセスに対していくつかの手続きを輸出できる．輸出された手続きは，他の μ プロセスから呼び出すことができる．手続きの輸出および呼び出しには，専用のProcThreadシステムコールを使用する．輸出は，システムコールの引数として関数のテーブルを指定し，呼び出しは，μ プロセスID，手続き番号，手続きへの引数列☆を格納した配列へのポインタ，返り値を格納する変数へのポインタを指定する．

☆ 現在の実装では，それぞれの引数は4バイト（intやlong，ポインタのサイズ）で格納できるものに限定している．

μプロセス間プロシージャコールは，RPC（Remote Procedure Call）[6]と類似しているが，次の点が異なる．(1) 同一アドレス空間内での通信であるため，通常のプロシージャコールを使って実装できる．引数としてポインタも利用可能である，(2) RPCでは，サーバ側のコンテクストで手続きを実行するが，μプロセス間プロシージャコールでは，呼び出し側のコンテクストをそのまま使って手続きを実行する，(3) サーバクライアントモデルではなく，どのμプロセスも他のμプロセスの手続きを呼び出すことが可能である．

μプロセス間プロシージャコールで輸出する手続きはスレッドセーフである必要がある．たとえば，内部に確保した静的領域へのポインタを返すような手続きを作成してはならない．ただし，あるμプロセスをμプロセス間プロシージャコールで呼び出すスレッドが1つしか存在しない場合はこの限りではない（7.1節参照）．

μプロセス間プロシージャコールで呼び出されるμプロセスでは，最初から存在するスレッドに加え，μプロセス間プロシージャコールによって他のμプロセスから訪れるスレッドが存在することになるため，μプロセス内でマルチスレッド動作することになる．このため，このようなμプロセスではLegacyコードの利用は困難である．このようなμプロセスでLegacyコードを利用するためには，最初から存在するスレッドを停止させ，μプロセスでμプロセス間プロシージャコールをたかだか1つしか受付けないようにする（手続きの入口で排他制御する）という方法がある．

5. 実　　装

ProcThreadを，フリーUNIXとして公開されているNetBSD-1.4.1（i386）[7]に実装した．

5.1 ProcThreadコア

ProcThreadコアは，我々が研究・開発している，移植性の高いマルチスレッド機構PTL[8]をベースに開発した．PTLは，ユーザレベルでのスレッド実装であり，POSIXスレッドのほぼすべての機能を備えている．ProcThreadでもPTLと同様，スレッドはユーザレベルスレッドとして実現している．

今回の実装では，ProcThreadコアは1つのUNIXプログラムとした．このプログラムを実行することで，メモリ中にProcThreadコアが配置され，引数で指定された初期μプロセスの実行を開始する．以下，実装の概略を述べる．

5.2 μ実行ファイルの配置

μプロセスのtext，data，bss領域は連続したメモリ領域を占めるが，bss領域はμプロセスの実行にともなって拡大する可能性がある．しかし，複数のμプロセスがアドレス空間内に存在するため，bss領域を無制限に拡大できるわけではない．このため，μプロセスのbss領域の最大長は固定長とした．この大きさは，μプロセスの生成時に変更することは可能である．

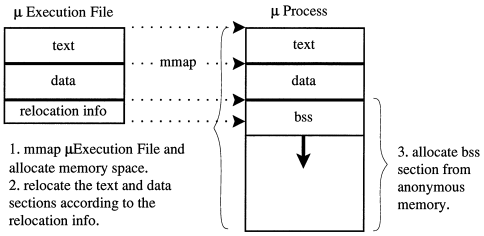

図3 μプロセスの生成
Fig. 3 μ process creation.

μ実行ファイルの配置は，以下のように行う（**図3**）．

(1) μ実行ファイルをmmapシステムコールでメモリにマップする（このとき，メモリに書き込んでもμ実行ファイルは書き変わらないように，MAP_PRIVATEを指定する）．これにより，text，data領域を用意する．

(2) ファイルの再配置情報を参照して，text，data領域の再配置を行う．

(3) bss領域は，mmapで匿名メモリセグメント☆をマップする．bss領域の拡大のため，bss領域の最大長をマップする．

(4) スレッドのスタックは，ProcThreadコアが管理するメモリ領域（mmapによって作られたセグメント）から割り当てる．

この方法では，μプロセスのbss領域を固定長で扱っているため，割り当てた領域が大きすぎれば効率が悪く，小さすぎればμプロセスのヒープ領域が溢れてしまうという問題がある．この問題は以下のようにすれば解決できる．

通常のUNIXプロセスで，bss領域がsbrkシステムコール等によって拡張されるのは，malloc関連の関数による場合がほとんどであるが，mallocで確保するメモリ領域は，bss領域から確保しなければならないわけではない．メモリ領域をbss領域から確保せず，大きな匿名メモリセグメントから確保していくようなmalloc関数群を用意し，μ実行ファイルとリンクさせることにより，bss領域は最小限とすることができる．匿名メモリセグメントを使いきった場合，新

☆ mmapでMAP_ANONを指定することによって得られるメモリセグメント．

たなセグメントをマップする．

5.3 μプロセスの実行

UNIXカーネルがプログラムを実行（exec）する場合，引数や環境変数をユーザのスタックに積み，プログラムのスタートアップルーチンを呼ぶ．ProcThreadでのμプロセス生成処理でもまったく同じようにスタックを設定し，新たに生成した初期スレッドのコンテクストでスタートアップルーチンを呼ぶようにした．これにより，μ実行ファイルがμプロセスとして，あるいはUNIXプロセスとして動作するいずれの場合も，スタートアップルーチンは共通の方法で引数や環境変数を受け取ることができる．

スタートアップルーチンは，まずμプロセスとして起動されたのか，通常のUNIXプロセスとして起動されたのかどうかを環境変数で判定する．前者の場合，ProcThreadコアの実行時リンク機能（5.4節）を呼び出す．後者の場合，通常のスタートアップルーチンの処理（実行時リンカのマップと呼び出し等）を行う．いずれの場合でも，その後，main関数に制御を移す．

5.4 実行時リンク機能

通常の動的リンクされたUNIXプログラムでは，実行時にスタートアップルーチンが実行時リンカをマップし，実行時リンク処理（共有ライブラリをメモリ上へ配置し，動的リンクの解決を行う処理）を行うが，ProcThreadでは，ProcThreadコアに実行時リンク機能を内蔵した．これにより，mmapや実行時リンカの初期化処理等を省略でき，μプロセスの実行時リンク処理を効率化できる．

内蔵する実行時リンク機能は，NetBSDの実行時リンカのソースコードをほぼそのまま流用して実装した．一部，実行ファイルが固定アドレスにロードされることを前提としている部分を修正する必要があった．

5.5 μプロセスの再利用

μプロセスの生成は，通常のスレッドの生成と比較すると高価な処理である（6.1節参照）．このため，スレッドの実行が終了した後でもメモリ上のμプロセスのイメージを保持しておき，後で同一のμ実行ファイルからμプロセスを生成する際に再利用できるようにした．UNIXのプログラムでは，system関数等を用いて，外部プログラムを呼び出すということがよく行われるが，ProcThreadでは，再利用機能を利用し，外部プログラムの起動を2回目以降高速化することが可能である（外部プログラムをμ実行ファイル化し，μプロセスとして実行するようにしておく）．μプロセスを再利用可能とするかどうかは，生成時にユーザが指定する．

μプロセス中のtext領域はスレッドの実行によって変更されることはないが，data，bss領域は変更されるため，再利用時にはこれらの領域を初期値に戻す必要がある．このため，実行時リンク処理が終了した時点でのdata領域の内容を一時ファイルにコピーする（この処理のため，再利用可能なμプロセスの生成には多少時間がかかる）．再利用時には，data領域にこのファイルをmmapする（このときも5.2節と同様MAP_PRIVATEを使用する）．bss領域は，mmapによって新しいメモリセグメントを割り当てる．これにより，再利用時には高速にμプロセスを再生成できる．

5.6 システムコールの扱い

μプロセスは，ProcThreadとUNIXの2種類のシステムコールを発行する．ProcThread専用の共有ライブラリ（libProcThread.so）が，ProcThreadシステムコールのスタブ関数を提供する．この共有ライブラリは，すべてのμプロセスと動的リンクされる．スタブ関数はProcThreadコアを関数呼び出しすることによってサービスを提供する．

ProcThreadでは，いくつかのUNIXシステムコールを，UNIXカーネルではなくProcThreadコアで横取り，実行する必要がある（**表1**に示す）．このため，libProcThread.soは，これらのUNIXのシステムコールと同名のスタブ関数も提供し，通常のUNIXシステムコールが呼ばれないようにした（interposition[9]）☆．

5.7 μプロセス間プロシージャコール

ProcThreadでは，μプロセス間でアドレス空間を共有しているため，μプロセス間プロシージャコールの実装は容易である．手続きの呼び出し処理は，呼び出すスレッドのスタックに指定された引数列を積み，呼び出すμプロセスの手続きのエントリ関数をコールすることで実装した．

ProcThreadコアは，μプロセスごとにいくつのスレッドが動作しているかを管理し，μプロセス間プロシージャコール実行中に_exitが呼ばれても安全なようにしている．

5.8 リンカ（ld）

μ実行ファイルには再配置情報が必要であるが，一般にUNIXの実行ファイルには再配置情報が存在しないため，再配置情報を残すようにリンカを修正した．

☆ 実際にはいくつかのライブラリ関数についても横取りしている．3.2節で述べたようにプロセスIDを利用して一時ファイル名を生成する関数（tmpnam()等）は，ProcThread環境では問題となる可能性があるため，生成するファイル名がスレッドごとにも異なるような関数で置き換えている．

表 1 UNIX システムコールの横取り
Table 1 UNIX system calls to be intercepted by ProcThread Core.

システムコール	横取りする理由
brk, sbrk	μ プロセスごとの bss 領域を扱う
_exit	μ プロセスを終了させるために用いる
chdir, umask	カレントディレクトリ，ファイル作成モードマスクは，μ プロセスごとに管理する
ファイル記述子を扱うもの	μ プロセスごとのファイル記述子を扱う
プロセスの資源を確保・解放するもの	μ プロセス終了時に，解放し忘れた資源を解放するため，μ プロセスが確保した資源を記憶しておく
fork	Legacy コードを利用できるようにするため，子プロセスでは fork を呼び出した μ プロセスだけを実行する
exec	exec 時に，プロセスのファイル記述子と μ プロセスのファイル記述子が同一となるように，プロセスのファイル記述子の対応を変更する．これにより fork&exec を用いた Legacy コードを実行できるようになる
setuid, seteuid 等	ProcThread では複数の μ プロセスがアドレス空間を共有するため，安全性を考慮してこれらのシステムコールはサポートしない

UNIX の実行ファイル（a.out 形式）とオブジェクトファイル（.o ファイル）のフォーマットは同一であるため，構造上は実行ファイルにも再配置情報を格納できる．これを利用し，実行ファイルに再配置情報を残すようにした☆．

6. 評　　価

作成した ProcThread 処理系の評価を行った．OS として NetBSD-1.4.1（i386），ハードウェアとして IBM-PC 互換機（Pentium 133 MHz，Memory 64 MB）を使用した．一時ファイル用の領域（/tmp）は高速化のため MFS（Memory File System）を使用した．

6.1 μ プロセス生成・消滅

何もしないプログラム（`main(){}`）を μ 実行ファイルとして用意し，以下の処理に要する時間を測定した．

- ProcThread で μ プロセスを生成，実行し，終了（join）するまでの時間（初期スレッドの生成時間を含む）．以下の 3 通りの場合を測定した．
  （1） μ プロセス生成時に再利用可能にしない場合（5.5 節参照）
  （2） 再利用可能にする場合
  （3） 以前に生成した μ プロセスを再利用した場合
- UNIX で fork&exec し，実行，終了（wait）が完了するまでの時間
- ProcThread の POSIX スレッド機能を用いて，何もしない関数をスレッドとして生成，実行，終了に要する時間．

☆ ELF 形式の場合も同様にリンカの修正が必要である．

表 2 μ プロセス生成・終了の時間
Table 2 Time to create and terminate a null (μ) process.

	Time (msec)
μ プロセス（再利用不能）	17.2
μ プロセス（再利用可能）	18.8
μ プロセス（再利用時）	1.41
UNIX fork&exec	19.3
POSIX thread	1.18

表 3 コンテクストスイッチ速度
Table 3 Context switching performance.

	Time (μsec)
ProcThread	1.95
POSIX thread (PTL)	1.95
UNIX process	82

なお，実験に使用した μ 実行ファイルの大きさは約 8.6 KB（text 領域 4 KB，data 領域 4 KB，再配置情報 0.6 KB）で，共有ライブラリ（libc）を動的リンクしている．再配置が必要な個所は 81 カ所であった．

結果を**表 2** に示す．ProcThread の μ プロセスの生成は UNIX の fork&exec よりも若干高速である．μ プロセスを再利用した場合には，μ プロセス生成のためのシステムコール（open，mmap×2），再配置処理，実行時リンク処理を省略できるため，POSIX スレッド生成時間とほぼ同じ時間で実行できる．このため，複数回，同一の μ プロセスを生成・実行するコストは十分に低い．

6.2 コンテクストスイッチ

ProcThread のスレッドの効率性を評価するため，コンテクストスイッチの速度を測定した．比較対象として，POSIX スレッド（PTL）と UNIX プロセスのコンテクストスイッチ速度も測定した（**表 3**）．なお，UNIX プロセスのコンテクストスイッチ速度の測定には，lmbench [10] を用いた．ProcThread のスレッドは，通常のユーザレベルスレッド実装と同性能であり，UNIX プロセス間のコンテクストスイッチより高速に動作することが分かる．

6.3 パイプ機構

ProcThread のパイプ機構の速度を測定するため，以下の実験を，ProcThread のパイプを用いた場合と，UNIX のパイプを用いた場合の両方で行い，要する時

表 4 パイプによる転送速度（msec）
Table 4 Time to transfer with pipe (msec).

	4 KB（5000 回）	`cat|grep`
ProcThread	980	316
UNIX	2700	590

表 5 μ プロセス間プロシージャコールの速度
Table 5 Time to perform inter μ process procedure call.

	Time (μsec)
関数呼び出し	0.068
μ プロセス間プロシージャコール	4.1
UNIX メッセージキュー	125
UNIX ドメインソケット	265

間を測定した（**表 4**）．

- ProcThread の 2 つの μ プロセス（あるいは UNIX の 2 つのプロセス）間で，パイプを経由して 4 K バイトのデータを 5000 回転送する．
- 実際のアプリケーションとして，以下のシェルパイプライン相当を実行する（ProcThread の場合，cat，grep コマンドは μ 実行ファイルとしておく）．実験は数回繰り返して行い，カーネルのディスクキャッシュを満たした状態で測定を行った．
  `cat /usr/share/dict/words|grep zonation`
  （words は約 2.4 MB のテキストファイル）

ProcThread のパイプは，UNIX のパイプよりも 3 倍程度高速であり，実際のアプリケーションでも高速化の効果が得られることが分かる．

6.4 μ プロセス間プロシージャコール

ダミー関数（`int foo(void)`）に対して，通常の関数呼び出し，μ プロセス間プロシージャコール，プロセス間通信のそれぞれの場合で，呼び出しから復帰に要する時間を測定した．プロセス間通信は，2 つのプロセス間で，メッセージキュー（msgsnd/msgrcv）と UNIX ドメインソケットを用いて通信を行い，同等の処理をエミュレートすることによって測定した（**表 5**）．

μ プロセス間プロシージャコールは，CPU の関数呼び出しには及ばないものの，他のプロセス間通信を用いた方法と比較すると非常に高速である．なお，μ プロセス間プロシージャコールの実行時間のほとんどは，エラーチェックや，μ プロセスを実行しているスレッド数の管理のために費やされている．

6.5 Legacy コードの利用

NetBSD のフィルタコマンドが，ソースに変更を加えることなく μ プロセスとして動作するかどうかを調査した．利用頻度の高いフィルタ（`grep`，`cat`，`sort`，`uniq`，`wc`）を調査した結果，μ 実行ファイルとしてリンクし直すだけで，μ プロセスとして正常に動作することを確認した．

表 6 μ 実行ファイルの実行に要するメモリサイズ
Table 6 Memory size to execute a μ process.

領域	サイズ（バイト）
text + data	8 K
bss	1 M（デフォルト）
スレッドスタック	16 K（デフォルト）
共有ライブラリ（libc）	550 K
共有ライブラリ（libProcThread）	12 K

6.6 再配置情報の大きさ

上記のフィルタコマンド群で，μ 実行ファイルに追加される再配置情報の大きさを調査したところ，再配置情報の大きさは，text と data 領域の大きさの合計の，8〜19%程度であった．このことから，再配置情報の追加による影響はそれほど大きくないといえる．

6.7 アドレス空間の使用量

`printf("Hello, World\n")` のみを行う μ 実行ファイルの実行に要する仮想アドレス空間のサイズは，1.6 M バイト程度である（**表 6**）．32 ビット CPU を用いた UNIX では，ユーザプロセスが使用できる仮想アドレス空間は一般に 4 G バイト程度であるが，mmap で使用できる領域は 2 G バイト程度であることが多い．このため，この程度の大きさの μ 実行ファイルは理論的には 1300 個程度生成できることになる．なお，text，data 領域は，それぞれページサイズ（4 K バイト）に切り上げられている．bss 領域は，使用するしないにかかわらずデフォルトの 1 M バイトを確保しているため無駄になっているが，5.2 節で述べた方法を用いれば効率化できる．

7. 応　　用

本手法は以下のような応用が考えられる．

7.1 マルチスレッドプログラムからの Legacy コードライブラリの利用

Legacy コードライブラリをリンクした μ プロセスを μ プロセス間プロシージャコールを使って呼び出すことで，複数のスレッドが動作する μ プロセスから，Legacy コードライブラリを利用することができる．このためには，まずライブラリ（L）をリンクした，ライブラリ実行専用の μ 実行ファイル（L_F）を作成する．L_F は L が提供する関数を輸出するだけのプログラムである．L を利用したい別の μ プロセス（M：マルチスレッド動作）は，L_F を μ プロセス（L_P）として生成する．M が L の関数を利用したい場合，μ プロセス間プロシージャコールで L_P が輸出した関数を呼び出す．L_P を複数生成することで，複

数のスレッドが同時に L の関数を実行することが可能である．

Legacy コードライブラリでは，内部に確保した静的領域を利用するような関数が存在する（getpwnam() 等）．このようなライブラリを利用する場合，静的領域をスレッドごとに独立させる必要があるため，呼び出すスレッドと呼び出される μ プロセスを 1 対 1 に対応させる必要がある．

7.2 マルチプロセスからマルチスレッドへの移行

ProcThread を用いることで，既存の複数のプロセスを用いたアプリケーションを，わずかな手間で単一プロセスで動作させることができる．6 章で示したように，μ プロセス間のコンテクストスイッチや通信のコストはプロセス間のそれよりも小さい．ProcThread を利用することによる特段の速度低下の要因は存在しないため，アプリケーションをより高速に動作させることが可能である☆．単一プロセスで動作させるためには，プログラムをリンクし直して μ 実行ファイルとし，プログラム中の fork&exec している個所を μ プロセス生成関数に置き換えるだけでよい．さらに，プロセス間通信に UNIX パイプを使用している場合，プロセス間パイプの確立処理（通常 fork&exec と同じ場所にある）を，μ プロセス間パイプを利用するように変更することで効率化できる．パイプの読み書き処理（read や write 等）を変更する必要はない．

具体的な応用としては，WWW サーバ（httpd）を修正し，httpd と CGI プログラムを同一プロセス内で実行して CGI の実行を高速化することがあげられる．この場合，μ プロセスの再利用機能により 2 回目以降の CGI の起動が高速化されるという利点もある☆☆．

7.3 シェルパイプラインの高速実行

6.3 節で述べたように，ProcThread のパイプ機構は高速であるため，シェルパイプラインを ProcThread を使用して実行できるようにシェルを修正し，パイプラインで使用する実行ファイルを μ 実行ファイル化することでシェルパイプラインの実行を高速化できる．

☆ ただし，現在の実装には以下の制約があるため，却って遅くなる可能性はある．(1) 通常，マルチプロセッサシステム上では複数のプロセスを並列に実行できるが，現在の ProcThread 実装はマルチプロセッサに対応していないため，同時に 1 スレッドしか実行できない．(2) ユーザレベルスレッドとして実装しているため，複数の UNIX システムコールを同時に実行できない．これらは現在の実装の制約であり，方式自体の制約ではない．

☆☆ ただし，CGI プログラムにバグがある場合，httpd の動作に影響を与える可能性がある．

8. ま と め

本稿では，Legacy コードを容易に利用できる ProcThread 方式を提案し，UNIX 上に実装，評価を行った．その結果，μ プロセスの生成は UNIX の fork&exec よりも若干高速であり，ProcThread のスレッドは，通常のマルチスレッドと同等の効率で動作すること，既存のコードの利用が容易であること等を確認した．

従来のマルチスレッドを用いたプログラムでは，スレッドは互いに相互排除や同期によって結び付くため，スレッド単体の独立性は高くないが，μ プロセス（μ 実行ファイル）は独立性が高い単位であり，それ自体をモジュールとして再利用することも可能である．

今後の課題としては，マルチプロセッサへの対応があげられる．現在の実装はマルチプロセッサを有効に活用できないが，提案方式自体にはマルチプロセッサでの実装を制限する要素は含まれていない．その他の課題としては，μ プロセス間通信機構として，メッセージキュー（msgsnd/msgrcv）互換のインタフェース等を提供し，既存のコードをより簡単に利用できるようにすることや ProcThread 環境上で動作するデバッガの開発等があげられる．

参 考 文 献

1) 谷口秀夫，長嶋直希，田端利宏：単一仮想記憶と多重仮想記憶を共存させたヘテロ仮想記憶の実現，情報処理学会研究報告，Vol.98, No.33, pp.87–94 (1998).
2) 毛利公一，大久保英嗣：マイクロカーネル Lavender の設計と開発，電子情報通信学会論文誌，Vol.J82-D-I, No.6, pp.730–739 (1999).
3) ISO/IEC: *Information Technology – Portable Operating System Interface (POSIX) – Part1: System Application Program Interface (API) [C Language]*, IEEE (1996).
4) Microsoft Corporation: Microsoft Visual C++ 6.0 Language Reference (1998).
5) Hewlett Packard: HP C/HP-UX Reference Manual (1998).
6) Sun Microsystems, Inc.: RPC: Remote Procedure Call Protocol specification: Version 2, RFC1057 (1988).
7) The NetBSD Foundation: NetBSD Project, `http://www.netbsd.org/`.
8) 安倍広多，松浦敏雄，谷口健一：BSD UNIX 上での移植性に優れた軽量プロセス機構の実現，情報処理学会論文誌，Vol.36, No.2, pp.296–303 (1995).
9) Gingell, R.A., Lee, M., Dang, X.T. and Weeks, M.S.: Shared Libraries in SunOS, *Proc. USENIX Association Summer 1987 Confer-*

ence, Atlanta, GA, pp.131–145, The USENIX Association (1989).
10) McVoy, L. and Staelin, C.: lmbench: Portable tools for performance analysis, *Proc. USENIX 1996 Annual Technical Conference*, San Diego, CA, pp.279–294, The USENIX Association (1996).
11) 安倍広多，松浦敏雄，安本慶一，東野輝夫：UNIXプログラムをスレッドとして動作させるProcThreadライブラリの設計と実現，情報処理学会研究報告，99-OS-82, Vol.99, No.65, pp.65–72 (1999).

(平成11年9月30日受付)
(平成12年7月 5 日採録)

安倍 広多（正会員）

平成4年大阪大学基礎工学部情報工学科卒業．平成6年同大学大学院博士前期課程修了．同年NTT入社．平成8年大阪市立大学助手．博士（工学）．マルチスレッド機構の実装，オペレーティングシステムの設計等に興味を持つ．電子情報通信学会会員．

松浦 敏雄（正会員）

昭和50年大阪大学基礎工学部情報工学科卒業．昭和54年同大学大学院博士後期課程退学後，同大学助手．平成4年同大学情報処理教育センター助教授，平成7年大阪市立大学教授．工学博士．ユーザインタフェース，マルチメディア，情報教育等に興味を持つ．ACM，IEEE，電子情報通信学会等会員．

安本 慶一（正会員）

平成3年大阪大学基礎工学部情報工学科卒業．平成7年同大学大学院博士後期課程退学後，滋賀大学経済学部助手．現在同大学助教授．博士（工学）．平成9年モントリオール大学客員研究員．通信プロトコルや分散システムの形式仕様記述・実装法に関する研究に従事．

東野 輝夫（正会員）

昭和54年大阪大学基礎工学部情報工学科卒業．昭和59年同大学大学院博士課程修了．同年同大学助手．平成2，6年モントリオール大学客員研究員．現在，大阪大学大学院基礎工学研究科教授．工学博士．分散システム，通信プロトコル等の研究に従事．電子情報通信学会，ACM各会員．IEEE Senior Member．