オペレーティング・システム 53－1
（1991. 12. 6）

ユーザプログラムによるカーネル外仮想記憶管理

新井 克也　　小野 諭
NTT ソフトウェア研究所　　NTT 研究開発技術本部

多重度の高い並列処理の方式として多重スレッド処理が注目を浴びているが、仮想空間を共有するスレッドは、スレッド間のデータ保護が不十分であるという問題点を持っている。本稿では、スレッド間のデータ保護技術として、OS 記憶管理と協調してスレッド専用の仮想空間を構築するユーザレベルの記憶管理方式を提案する。本方式を利用し、スレッドをそれ専用の仮想空間で実行することにより、スレッド間のデータ保護が実現される。ユーザレベルの記憶管理は、1) スレッドにみあった粒度の保護実現、2) 必要十分な保護のレベルの選択、3) garbage collection のような、本来、記憶管理を必要とするアルゴリズムの実現、などの手段として有効である。本稿では、ユーザレベルの記憶管理の手順と、OS に付加する新たなシステムサービスを定義する。また、そのオーバヘッドの最適化手法として、類似性にもとづくスレッド仮想空間の再利用、および、スケジューリング情報にもとづく記憶資源の使用率の効率化を行う。

Out–of–Kernel Memory Management

Katsuya ARAI　　Satoshi ONO
NTT Software Laboratories　NTT R&D Headquarters

Recently, the *multithread* technique is widely used with high degree of multiprocessing. However, the *thread*'s own data is not protected from others, because it is placed in shared address space. In this paper, a new user–level memory management technique for protecting *threads*'s data on *multithread* environment is proposed. This technique works cooperatively with memory manager in kernel, and provides *thread* a virtual address space that is restricted to each. User–level memory management facility offers following functions, 1) fine grained protection, 2) a selectively for adequate protection level, 3) a mechanism for algorithms that need memory management, such as garbage collection, check pointing, and others. We describe algorithms for user–level memory management, and define some additional system–calls. To reduce overheads arising from proposed technique, we use the following methods, i) reuse of virtual address spaces for similar *threads*, ii) manage of resource–allocation on the state of the *thread*.

1 はじめに

近年、コンピュータの応用範囲の拡大による処理の複雑化とマルチプロセッサシステムなどのハードウェア技術の進歩を背景として、複雑な処理を細かい粒度で並列に実行する多重処理への要求が高まっている。このような粒度の細かい多重処理では、一般に cpu 数に比べて多重度が高く、処理の生成 / 切り替えが頻繁に起こる。 そのため、コンテキストの生成 / 切り替えのオーバヘッドが少ない、スレッドによる多重処理が注目を浴びている [1]。

しかしながら、他と共有するアドレス空間上で実行されるスレッドは、データの保護が不十分であるという問題点がある。この問題に対して、スレッドのスタック領域の保護方式の研究がある [7],[8]。

本稿では、多重スレッド処理におけるスレッド間の保護を目的として、ユーザレベルで記憶管理を行なう、カーネル外仮想記憶管理の手法を提案する。 本手法は、OS の記憶管理と協調してユーザレベルで仮想アドレス空間を構築し、この上でスレッドを実行することで、スレッド間の保護を実現する。 また、本稿では、OS の一部を変更して新たなシステムサービスを追加することで、本手法を実現するアプローチをとる。

ユーザレベルで記憶管理を行なう利点は、スレッドの粒度にみあった、細粒度の保護を実現できることの他に、アプリケーションプログラム (AP) の知識にもとづく制御の最適化や、記憶管理方策の設定の局所性などがあるが、詳しくは本稿 2.2 で述べる。 ユーザレベルでの OS 制御機構に関する他の研究としては、スケジューリング機構を Out-of-kernel 化し、これを AP で制御する並列記号処理言語 [4] や並列記号処理言語の実行系 [5] がある。

また、本方式のオーバヘッドに対して、

- アドレス空間の類似性にもとづく部分的更新によるアドレス空間の再利用。
- カーネル外スケジューラなど、他のユーザモード制御機構との協調による制御コストの削減。
- 必要最小限の保護レベルを用いた保護の実現コストの削減。

による最適化を行なった。

本稿の構成は以下の通りである。 まず、多重処理の実行環境と従来の記憶管理の関係、および、ユーザレベルでの記憶管理の利点と欠点について述べる。 次に、本方式の概要を述べ、その実現手順と新たに必要なシステムサービスを定義する。 そして、本手法の応用について述べ、最後は、まとめとする。

2 仮想記憶

本節では、仮想記憶と保護の関係、および、ユーザレベルで記憶管理を行なう事の意義、そして、仮想記憶の従来技術について述べる。

2.1 実行環境の保護

一般に、計算機システムで実行される処理は、仮想アドレス空間上のコンテキストとそのコンテキストにもとづいて実行される制御点からなる。 制御点とは、cpu 上で動作する実体であり、program counter をはじめとする cpu のレジスタをそのコンテキストとして持つ。 また、仮想アドレス空間は、オペレーティングシステム (OS) によって、アクセスすることを許可されたデータの集合である。 以下では、このような仮想アドレス空間と制御点の組を処理実行単位と呼ぶ。

処理実行単位には、一つの制御点に一つの仮想アドレス空間を割り当てるプロセスと一つの仮想アドレス空間をいくつかの制御点で共有するスレッドがある。 プロセスは、制御点ごとに仮想アドレス空間を持つため、そのコンテキストは OS により保護されているが、プロセス数が大きい多重処理では、コンテキストの生成 / 切り替えのオーバヘッドが大きい。 それに対し、スレッドは、複数の制御点が一つの仮想アドレス空間を共有するため、コンテキストの生成 / 切り替え時のオーバヘッドは軽減されるが、スレッド間のコンテキストの保護が不十分である。

たとえば、数千の呼処理を同時に実行する交換処理では、全ての処理が単一のアドレス空間上で実行されている [9]。 しかし、この方法では、アドレス空間を共有する処理の独立性が低いために、新たに付加した高機能サービス処理の誤りが基本的な交換処理に波及することを防ぐ手段がない。

このようなスレッド間保護の問題に対しては、スレッドの粒度にみあった細かい粒度の保護を実現する、ユーザレベルの記憶管理が有効である。

2.2 ユーザレベルの記憶管理

ユーザレベルで記憶管理を行なうメリットは、スレッドの粒度にみあった、小さい粒度の保護が実現できることの他に、OS が持つ制御機構を Out-of-kernel 化することにより、保護の粒度とそのオーバヘッドの

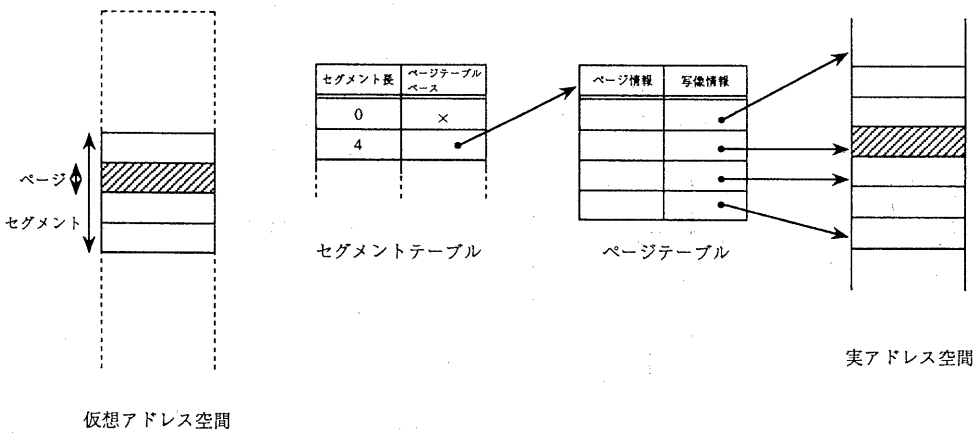

図1．ページ化セグメントによる仮想記憶

妥協点をアプリケーションプログラム(AP)で局所的に設定できる点、および、APに関する詳細な知識とOSの制御機構をあわせ持ち、APの性質に合わせて最適化された、いわゆる拡張OS機能の一つとして利用できる点である。さらに、Out-of-kernel化した記憶管理機構は、本来、記憶管理を必要とするアルゴリズムを効率的に実現する手段としても有効である。このような特徴を持つアルゴリズムとしては、ヒープメモリ管理技術における並行garbage collectionのアルゴリズム、および、フォールト・トレラント技術における永続性オブジェクト、チェックポインティングのアルゴリズムのように、記憶管理と密接な関係を持つアルゴリズム[2]や、記憶管理方策を特殊化することによって、大幅な効率化が可能な分枝限定法(Branch-and-Bound)アルゴリズム[6]などがある。

2.3 従来の記憶管理方式

2.3.1 カーネル記憶管理による仮想記憶

仮想記憶を実現する基本方式は、二次記憶上に常駐するプログラムをブロックに分け、それを仮想アドレスに対応づけ、そのブロックがアクセスされた時、それが主記憶上になければOSが自動的に主記憶に転送することである。

仮想記憶を実現する代表的な手法には、プログラムを同じ大きさのブロック(ページ)で分割するページング方式と、手続きやデータの種別をブロック(セグメント)としてプログラムを分割するセグメンテーションがある。また、主記憶上に転送されたブロックが占有する主記憶域の単位をそれぞれページフレーム、セグメントフレームと呼ぶ。

ブロック(ページ/セグメント)と仮想アドレスの対応は、ページテーブル/セグメントテーブルと呼ばれるデータで定義される。このテーブルは、テーブル項目(PTE/STE)を仮想アドレス順に並べた配列である。テーブル項目は、それぞれのブロックに関する情報、たとえば、許されるアクセスの種類、アクセスの履歴、ブロックが主記憶上にあるかどうかなど、を保持するデータである。このように、仮想アドレス空間は、ページテーブル/セグメントテーブルで定義される論理的なアドレス空間である。

OSの記憶管理は、プログラムに対して、それが必要とするブロック(ワーキングセット)を主記憶上に保持する。この時OSは、記憶資源の使用率を高めるために、プログラムの実行に必要十分であり、かつ最小限のワーキングセットを割り当てるという制御を行なう。この制御をワーキングセット管理と呼ぶ。

2.3.2 ページ化セグメント

セグメンテーションは、手続きなどが一つのセグメントにまとまっているため、その共用が容易である反面、セグメントの大きさが一定ではなく、主記憶の外部断片化が生じるという問題がある。その一方、ページングは、ページのサイズが、みな同じであるため主記憶の外部断片化が起こらず、主記憶を有効に利用することができるが、プログラムが単純なページの配列として写像されているため、手続きやデータの種別の境界がページの境界と一致するとは限らず、その共用

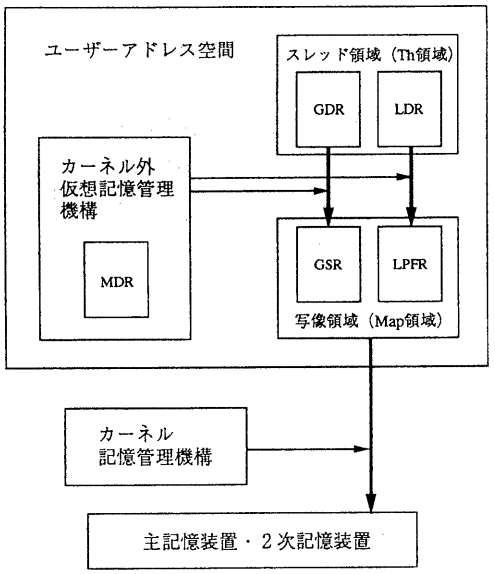

図2．カーネル外仮想記憶管理の概念

が困難な場合がある。

このように相補的な関係にあるページングとセグメンテーションを組み合わせて、お互いの欠点を改善するために広く用いられる方式として、図1に示すようなページ化セグメント方式がある。この方式では、セグメントをページ単位で分割することでセグメンテーションの外部断片化の問題を解決している。また、セグメントテーブル項目は、そのセグメントを定義するページテーブルへのポインタを保持しているため、セグメントを共有する場合には、ページテーブルを共有することができる。

3　カーネル外仮想記憶管理

本節では，これまでの議論をふまえて、本稿で提案するカーネル外仮想記憶管理の実現法、およびそのオーバヘッドと最適化手法について述べる。以下では、多重スレッド実行環境、および、ページ化セグメントによって仮想記憶を提供するOSを前提として議論する。

(1) セグメントの確保 / 構築 / 伸長	
allocate_segment(address, size)	return: result_code
free_segment(address)	return: result_code
grow_segment(size)	return: result_code
(2) ページの map/unmap	
map_page(address, data_object)	return: result_code
unmap_page(address)	return: result_code

表1　前提とするシステムサービス

(1) スレッド領域 (Th 領域) の構築 / 解放	
create_ThR()	return: ThR_handle
free_ThR(ThR_handle)	return: result_code
(2) 広域データ領域 (GD 領域) の load/unload	
load_GDR(GSR_segment, ThR_handle)	return: result_code
unload_GDR(GSR_segment, ThR_handle)	return: result_code
(3) 局所データマップ (LD マップ) の構築 / 解放	
create_LDmap(segment_node_path)	return: LDmap_handle
free_LDmap(LDmap_handle)	return: LDmap_handle
(4) 局所データマップ (LD マップ) の更新 / 複写	
connect_segment_node(seg_node, LDmap_handle)	return: LDmap_handle
disconnect_segment_node(seg_node, LDmap_handle)	return: LDmap_handle
copy_LDmap(LDmap_handle)	return: LDmap_handle
(5) 局所データ領域 (LDR) の load/unload	
load_LDR(LDmap_handle, ThR_handle, local_WS)	return: result_code
unload_LDR(LDmap_handle, ThR_handle)	return: local_WS

表2　本方式で新たに定義したシステムサービス

3.1　カーネル外仮想記憶管理の概要

カーネル外仮想記憶管理は、後で述べるOSの記憶管理機構とのインターフェースを利用して、ユーザレベルで仮想アドレス空間を構築し、この上でスレッドを実行することで、スレッド間の保護を実現する手法である。具体的には、ユーザアドレス空間の一部をページまたはセグメント単位で並べ替え、これをユーザアドレス空間上に再配置することで仮想アドレス空間を構築する。

本方式では、図2に示すように、ユーザアドレス空間上にスレッド領域 (Th 領域)、写像領域 (Map 領域)、および、管理データ領域 (MD 領域) の3つの領域を定義する。

Th 領域は、Map 領域に写像されたページやセグメントを部品とし、それ再配置することで構築される。スレッドが参照できるのは、ユーザアドレス空間のうちこの領域だけである。また、この領域には、ス

レッドからユーザアドレス空間を直接操作するシステムサービスのエントリを配置しない。

Map 領域は、後で述べるカーネル記憶管理があらかじめ用意しているシステムサービスを用いて、Th 領域に必要なデータを写像する。 すなわち、この領域に写像されたページやセグメントは、Th 領域を構成する部品である。

MD 領域は、スレッド実行環境の構成情報を配置する領域である。 スレッド実行環境の構成は、Th 領域に再配置すべきページやセグメントが、Map 領域上のどの位置に写像されているかといった情報で定義される。 また、後で述べるセグメントノードなど、Th 領域の類似性に関する情報もこの領域に配置する。

これらの領域は、いずれも OS によって提供されるユーザアドレス空間の一部である。 したがって、表 1 に示すような、セグメントの生成や解放、および、ページの配置などの基本的な手続きは、OS によってあらかじめ提供されているものとする。 しかし、Map 領域に配置されたページやセグメントを Th 領域に再配置する手続きは、ページテーブルやセグメントテーブルの操作を陽に指定するため、表 2 に示すような新たなシステムサービスが必要となる。 表 2 の手続きについては、本稿 3.3 で詳しく述べる。

3.2 カーネル外仮想記憶管理の最適化手法

以下では、先に列挙した 3 つの最適化手法を本方式でどのように具体化したかについて述べる。

・アドレス空間の再利用 一般に、fork した関係にある二つの処理実行単位の固有データのうち、fork 前に配置されたデータは共通であるが、fork 後、新たに配置されたデータの共通性は保証されない。

本手法では、fork した関係にある処理実行単位の、このような性質に着目し、スレッドが fork したという情報からアドレス空間の類似性情報を生成する。 そして、これにもとづき、アドレス空間の非共通部分を差し替えて再利用する。 アドレス空間の類似性情報を表現するデータについては本稿 3.3.2 で、具体的なアドレス空間再利用の手順は本稿 4.1 で、それぞれ述べる。

・ユーザモード制御機構との協調 ここでは、ユーザモードのスケジューリング機構 (sc 機構) を仮定して、ユーザモードの記憶管理 (mm 機構) との協調による最適化手法を考える。 まず、mm 機構の知識にもとづき、類似アドレス空間を持つ処理実行単位を優先的に実行するという sc 機構の方策によって、アドレス空間の再利用性を高める手法がある。 また、それとは逆に、sc 機構の知識にもとづき、さしあたって実行されないアドレス空間や記憶域を再利用するという mm 機構の方策によって、記憶資源の有効利用をはかる手法がある。

本手法では、これらのうち、後者の手法を実現する。 sc 機構を仮定して、スレッドがさしあたって実行されないことがわかった場合にアドレス空間の再利用をうながす手続きとして、アドレス空間構築データの load/unload の手段を本稿 3.3.1, 3.3.2 に、また、ページフレームの再利用をうながす手順を本稿 3.3.3 で述べる。

・必要最小限の保護 従来、汎用的な OS において信頼できない手続きからデータを保護するためには、遠隔手続き呼び出し (RPC: Remote Procedure Call) を用いて、独立なアドレス空間上でそれを実行するしかなかった [3]。

本稿では、新たなアドレス空間を生成することなしに、手続きの実行環境を構築し、信頼できない手続き呼び出しを効率的に実現する手順を本稿 4.2 で述べる。 これは、本手法が提供する必要最小限の保護によって、保護の実現コストが削減される例である。

3.3 スレッド実行環境の構築

スレッドの実行環境は、プログラムコードや共有データなどのスレッド間で共有する広域データと、制御スタックや局所変数などのスレッド固有の局所データに大別できる。 一般に、広域データは、データの種類ごとに比較的大きなサイズを単位として共有されることが多い。 それに対して、局所データは、fork した親子関係によって部分的に共通のデータを持つような、いわゆる木構造をなす。

本方式では、上記の実行環境の性質に着目し、Th 領域をさらに広域データ領域 (GD 領域)、および局所データ領域 (LD 領域) の二つに分割し、それぞれを別の手順で管理する。 また、Map 領域もこれにならって、それぞれ広域セグメント領域 (GS 領域) および論理ページフレーム領域 (LPF 領域) に分割して管理する (図 2)。

Th 領域を構築する手順は、まず、Th 領域を定義するセグメントテーブルを表 2(1) の手続きで確保し、次に、GD 領域と LD 領域をそのセグメントテーブルに登録 (load) することである。

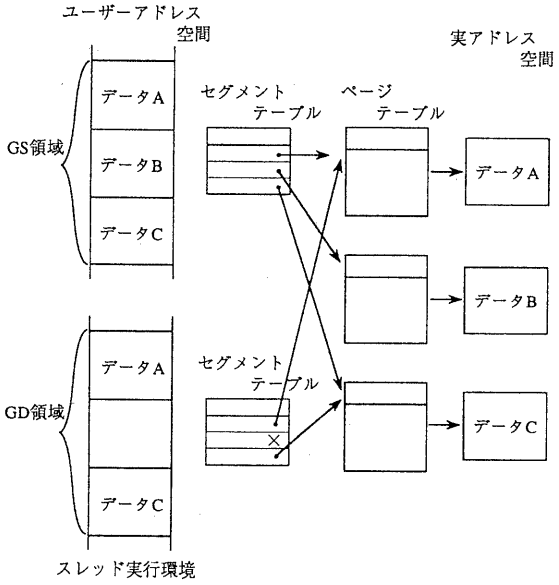

図3．広域データ領域の構築

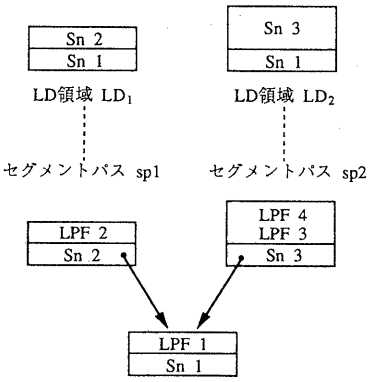

図4．セグメントノードとセグメント木

3.3.1 広域データ領域 (GD 領域) の構築手順

GD 領域は、Map 領域のうち GS 領域に写像されたデータをセグメント単位で再配置して構築するので、ページテーブルなどのページング機構は、GS 領域と共有することができる。この方法は、ページ化セグメント方式でセグメント共有する一般的な方法である。図3は、GS 領域のデータ A とデータ C を写像するセグメントを GD 領域に再配置した状態を示している。

GS 領域を構築する手順は、必要なデータを GS 領域に写像したあと、GS 領域上のセグメントを直接指定して、表2(2) の手続きで Th 領域に load することである。

3.3.2 局所データ領域 (LD 領域) の構築手順

LD 領域は、Map 領域のうち LPF 領域に写像されたデータをページ単位で再配置する。

先に述べたように、fork した関係にある二つのスレッドの LD 領域は、fork 前に構築される共通ページ列と、fork 後に構築される独立ページ列の2つの部分で構成されている。本方式では、このような LD 領域の共有関係を表現するために、セグメントノードと呼ぶデータ構造を導入する。

セグメントノードは、LD 領域を構成するページ列の共有単位であり、ページ列の共通関係は、セグメント木と呼ぶ木状に構造化したセグメントノードで表現する。そして、各スレッドの LD 領域をセグメント木のリーフ位置からルートに至るセグメントパスで定義する。セグメントノードの具体的構造は、セグメントノード間の関係や構造を定義するデータ、および、セグメントノードを構成するページ列を保持したデータである。図4は、二つの LD 領域 LD_1, LD_2 の構造を定義するセグメントパス $sp1$ と $sp2$、および、その共通関係を定義するセグメント木を示している。

次に、セグメントパスで定義される LD 領域の構築手順について述べる。LD 領域の構築は、LPF 領域に写像されたページを単位として並べ替えるので、ページテーブルを共有することはできない。そこで、セグメントパスで定義されるページ列に対応する LPF 領域の PTE を別のページテーブル (LD マップ) に複写することが必要である。その具体的な手順は、表2(3) の手続きにより、セグメントパスで定義されるページ列を写像する LD マップを作り、次に、表2(5) の手続きで、これを Th 領域に load することである。図5は、図4のセグメントパス sp_1 で定義される LD マップ map_1 を生成し、これを LD 領域に load したところを示している。

3.3.3 広域的なワーキングセット管理

スレッドが LD 領域を通して無効ページを参照すると、ページフォールト処理により、LD マップの PTE が変更され、有効ページへの参照経路が確立される。しかし、さしあたって実行されないスレッドが保持す

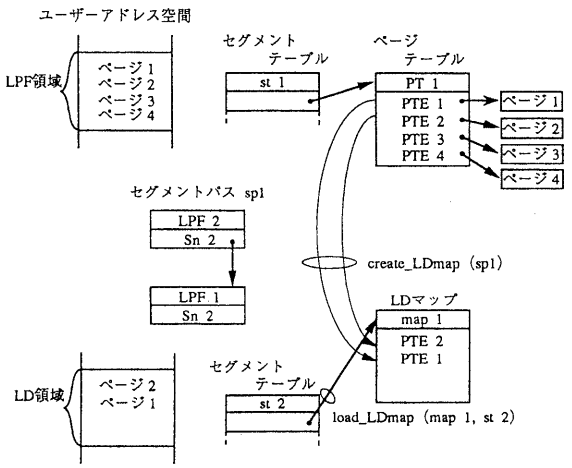

図５．局所データ領域の構築

る有効ページへの参照経路は、それが占有するページフレームの再割り当てを阻害するため、ページフレームを無意味に消費することとなる。

そこで本方式では、LD マップが unload される時には、それが持つ参照経路を取り外し、有効ページが占有していたページフレームを他の LD 領域へ再割り当てすることをうながすといった、複数領域にわたる広域的なワーキングセット管理を行なう。

また、unload 時に取り外された有効ページの情報(局所 WS) は、unload した LD マップとともに保存しておき、これを再び load する際に、取り外されていた有効ページへの参照経路を復元するために用いられる。つまり、LD マップを再 load する時に、局所 WS に含まれるページが、依然として有効ページであった場合、そのページへの参照経路を無条件で確立する。

3.4 カーネル外仮想記憶管理のオーバヘッド

本方式では、既存の記憶管理にくらべて、以下のようなオーバヘッドが見込まれる。

1. LD マップの構築にともなう PTE の複写
2. セグメントノードおよびセグメント木に対する操作や検索
3. 本方式で導入した新たなシステムサービスの呼び出しのオーバヘッド

これらのうち、1. に関しては、後で述べる LD マップの再利用による効率化が期待できる。

また、上記のオーバヘッドに対するハードウェア支援としては、2. に対しては、仮想計算機 (たとえば IBM, VM/370 など) で実現されている、主記憶の二段階の仮想化を支援する VM 補助機構の利用などが考えられ、また、3. に対しては、セグメントノードの構造を持つ構造メモリなどによる支援が考えられる。

4 カーネル外仮想記憶管理の応用

本節では、ユーザレベルの記憶管理の最適化手法として、アドレス空間再利用の手順、および、必要最小限の保護によって効率化された、信頼できない手続きの呼び出し手順について述べる。

4.1 スレッド実行環境の再利用

交換処理のように、多重化される処理数が cpu 数に比べてはるかに多く、かつ、それらの処理が類似した小数のグループからなる多重処理に対しては、処理の類似性を利用したスレッド実行環境の再利用によって、本機構のオーバヘッドの効率化が期待できる。

ここでは、fork 関係から類似性の情報が得られる LD 領域について、その再利用の方法を示す。また、GD 領域に関しては、その類似性の情報をプログラム上の保護に関する明示的な記述、または、プログラムの解析によって得ることが考えられるが、その具体的方法については、未考察である。

以下では、二つのスレッドの LD 領域が、図 4 に示すセグメント木の二つのセグメントパス sp_1, sp_2 で定義されている場合を例とし、sp_1 で定義されるアドレス空間を写像する LD マップ (map_1) が存在し、sp_2 を写像する LD マップがない場合を考える。この場合、セグメント木から、sp_1 と sp_2 の共通部分が sn_1 であり、map_1 の sn_2 を sn_3 に更新することにより、sp_2 の LD マップが得られることがわかる。

そこで、map_1 がどの LD 領域にも load されていない場合には、以下の手順で map_1 を更新して再利用する。

$disconnect\_segment\_node(sn_2, map_1) \rightarrow map_1$
$connect\_segment\_node(sn_3, map_1) \rightarrow map_1$

また、map_1 が使用中のため、それを更新して再利用することができない場合には、

$copy\_LDmap(map_1) \rightarrow map_2$

の手順により、map_1 を複写して得られるテンプレート map_2 を生成し、これを上記のように部分的に更新することで、sp_2 の LD マップが得られる。

以上の手順により、仮想アドレス空間の構築データである LD マップは、多重処理に割り当てられる cpu 数と、スレッドの類似グループ数にみあった数だけ生成するように制御できる。 たとえば、一つの機能を一つのスレッドで実行するといった、類似グループの数が静的に決まっているような場合には、グループ一つに対してシステムの cpu の数を最大数とするなどの制御が考えられる。

また、LD マップが再利用できない場合には、あらかじめ類似グループの LD マップのテンプレートを生成しておくことにより、そのオーバヘッドを分散することが可能である。

4.2 信頼できない手続きの呼びだし

ここでは、新しいコンテキストの生成、および、引数データの通信が不要な、信頼できない手続き呼び出しの効率的な実現手順について述べる。

その手順は、まず、スレッドの実行環境から保護したい領域を取り外し、次に、手続きに制御を移すことである。 保護したいデータを実行環境から取り外すことで、信頼できない手続きの不当なアクセスを防ぐことができる。 この時、信頼できない手続きが取り外されたデータを実行中に再び load することを防ぐために、スレッド実行環境を操作するシステムサービスのエントリもこの実行環境から取り外しておく。 信頼できない手続きが処理を終了した時は、カーネル外仮想記憶管理機構に制御が移され、取り外されていたデータが元の状態に復元されたあと、スレッドに制御が戻される。

5 まとめ

本稿では、多重スレッド実行において、ユーザレベルでスレッド仮想空間を構築し、スレッド間保護を実現する、カーネル外仮想記憶管理の手法を提案し、その実現手順を示した。 また、オーバヘッドの最適化手法として、スレッドの fork 関係に着目したアドレス空間の再利用、必要最小限の保護による保護コストの最適化、スケジューリング知識にもとづく広域的なワーキングセット管理について述べた。

また本稿では、ユーザレベルで仮想記憶を行なう意義として、粒度の細かい保護の実現の他に、記憶管理方策の局所性、記憶管理を必要とするアルゴリズムの実現手段としての有効性などについて述べた。

今後は、実システム上での試作にもとづく本方式の定量的評価を進めていきたい。

謝辞 日頃ご指導くださる NTT 基礎研究所の小川瑞史研究主任、ならびに、本論文に関してご助言を頂いた斉藤康己リーダをはじめ NTT ソフトウェア研究所の諸氏に感謝します。

参考文献

[1] ACCETTA, M. AND OTHERS, : Mach: A New Kernel Foundation For UNIX Development, in Proceedings of Summer Usenix(1986).

[2] APPEL, A. W. AND LI, K.: Virtual Memory Primitives for User Programs, in Proceedings of ASPLOS-IV, ACM(Apr. 1991), 96–107.

[3] BERSHAD, B. N., ANDERSON, T. E., LAZOWSKA, E. D., AND LEVY, H. M.: Lightweight Remote Procedure Call, *ACM Transactions on Computer Systems* 8, 1(Feb. 1990), 37–55.

[4] GABRIEL, R. P. AND MCCARTHY, J.: Queue-based Multi-processing Lisp, in ACM Conf. Record of the Symposium on LISP and Functional Programming(1984), 25–43.

[5] HOZUMI, M. AND OTHERS, : Multiprocessor Common Lisp on TOP-1, in Proceedings of US/JAPAN workshop on parallel LISP(1989).

[6] YU, C. F. AND WAH, B. W.: Virtual Memory Support for Branch-and-Bound Algorithms, in Proceedings of COMPSAC, IEEE(1983), 618–626.

[7] 斎藤雅彦, ほか: マルチスレッド実行環境に適した並列処理システムのメモリ管理方式, 情処論 32, 4(Apr. 1991), 481–490.

[8] 新井潤, ほか: 分散 OS ToM, 情処研報 89-OS-45 89, 94(Nov. 1989).

[9] 久保田稔, ほか: 通信網ワイドの交換サービスを実現するためのプログラム制御方式, 情処研報 90-OS-49 90, 98(Dec. 1990).