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†

齊藤雅彦†† 上脇正†† 山口伸一朗††

共有メモリ型マルチプロセッサにおける並列処理の単位として，スレッドが注目を浴びている．スレッドは同一仮想空間上で並列に動作するため，切換え，生成時間を短縮でき，柔軟で高速な同期，通信，排他制御が可能である．しかし，仮想空間を共有することは，スレッド間でのデータ保護を弱めることになる．特に，スレッド実行に伴って伸縮するスタック領域の保護が弱い．このため，一般には，固定的に確保したスタック領域間に境界領域を設け，あるスレッドが境界領域を参照した場合にスレッドまたはプロセス自体を異常終了させる方式が用いられる．しかし，この方法では，仮想空間を最大限に使用することができない．そこで，我々は，スレッドのスタック領域を保護するとともに，仮想空間を最大限に使用するメモリ管理方式：「動的スタック拡張方式」を開発した．動的スタック拡張方式では，スタック領域の伸縮に伴って，適宜仮想空間の割当て／解放を行う．今回開発した並列処理用 OS 上に動的スタック拡張方式を実現した結果，5% 程度のオーバヘッドがあるものの，スタック領域の保護と，仮想空間の有効利用を両立させることが可能となった．

1. はじめに

近年，マイクロプロセッサの性能は，半導体技術の進歩や，RISC，スーパスカラ，VLIW といった高速化技術により着実に向上している．しかし，計算機の性能が高まれば高まるほど，計算機で処理する仕事も増加し，計算機性能に対するユーザの要求は単体プロセッサの性能向上を上まわって大きくなってきている．特に，データベース，シミュレーション，グラフィックス処理等では，この傾向が著しい．これらに適した並列処理として，複数のプロセッサを有して並列に動作させるマルチプロセッサ技術が注目されている．

マルチプロセッサシステムにおいては，複数のプロセッサを効率よく動作させることが重要であり，これはそのまま OS の役割となる．このような考えから，我々はマルチプロセッサの基本ソフトウェアとして，並列処理用 OS・SKY-1 (System Kernel for You-1) を開発した[11)~15)]．SKY-1 では，サブルーチンやループといったレベルの並列処理にも対応するため，プロセス内の実行環境を共有して並列に動作するスレッドの概念[1)]を導入している．

同一プロセス内のスレッドは実行環境を共有するため，プロセスに比べて，切換え処理／生成処理を高速化できる．また，スレッドは実行環境の一部として，仮想空間を共有しているため，データ共有が容易である．しかし，スレッド固有であるべき領域も事実上共有することになるため，何らかの手段によりスレッド間でデータを保護しなければならない．特に，スレッドの実行に従って伸縮するスタック領域のデータ保護が問題となる．このため，スレッドを導入した OS においては，一般に，固定的に確保したスタック領域間に境界領域（参照禁止領域）を設け，あるスレッドが境界領域を参照した場合にスレッドまたはプロセス自体を異常終了させる方式[7),8)]が用いられている．しかし，この方法では，仮想空間を最大限に使用することができない．また，仮想空間を有効に使用するため，ユーザにスタック領域の最大値を予測させることになれば，煩雑なメモリ管理をユーザに強いることになり，ユーザインタフェースを著しく低下させることになる．

そこで，我々は，マルチスレッド実行環境に適した新たなメモリ管理方式として，「動的スタック拡張方式」を開発した．動的スタック拡張方式では，スタック領域の伸縮に従って，適宜仮想空間の割当て／解放を行う．これにより，スタック領域の保護を行う際に，ユーザにメモリ管理の負担を負わせることなく，使用している仮想空間の大きさを常に最小に近い値に保つことができる．すなわち，仮想空間を最大限に使用することが可能となる．

本論文では，以降，従来のメモリ管理方式とスタック領域の保護に関する問題点について述べ，これを解決する動的スタック拡張方式を提案する．最後に，動的スタック拡張方式を並列処理用 OS・SKY-1 上に実現した場合の性能評価について述べる．

† Memory Management of Parallel Processing Systems under Multi-Thread Environment by MASAHIKO SAITOH, TADASHI KAMIWAKI and SHIN'ICHIRO YAMAGUCHI (Hitachi, Ltd.).

†† (株)日立製作所

2. 従来のメモリ管理方式と問題点

2.1 プロセス使用時のメモリ管理

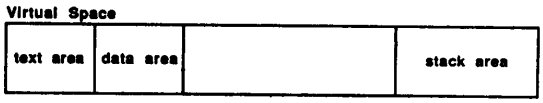
今回我々が開発した SKY-1 は共有メモリ型マルチプロセッサを対象とし，UNIX*[10] をベースとして作成した汎用目的の並列処理用 OS である．ベースとなる UNIX のプログラム実行単位はプロセスであり，実行環境としてそれ自体1つの仮想空間を有する．UNIX のプロセスにおける仮想空間の使用状況例を図1(a)に示す．仮想空間の先頭から，テキスト領域，データ領域が存在する．この例では，関数の呼出し／復帰に伴って伸縮するスタック領域が仮想空間の最後尾に存在し，仮想空間の前方へと伸縮する．したがって，関数呼出しのネストなどにより，スタック領域が拡張されたとしても，仮想空間全体を使い切ってしまわない限り，他の領域に侵入することはない．また，プロセス間では仮想空間が異なるため，仮想記憶管理機構によってデータ保護が完全に行われている．

2.2 スレッド導入時のメモリ管理

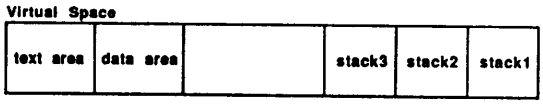
プロセスはプログラムレベルの並列処理に適している（図2(a)）．これに対して，共有メモリ型マルチプロセッサにおいては，サブルーチンレベルの並列処理が効果的である．サブルーチンレベルの並列処理に適した概念として「スレッド」[1)~7)] がある．SKY-1 でも，並列処理の単位として，従来のプロセスに加え，スレッドを導入した．スレッドは論理的なプロセッサとみなすことができる（図2(b)）．同一プロセス内のスレッド切換え時には，実行環境を切り換える必要がなく，プロセスに比べて，切換え処理を高速化することができる．スレッドを生成する場合においても，仮想記憶管理のアドレス変換テーブル等を生成する必要がない．また，データの共有が容易という利点もある．

スレッドを導入した場合，仮想空間の使用状況として，図1(b)(c)に示す2種類がある．スレッドはテキスト領域，データ領域を共有するが，スレッド固有のデータを有するスタック領域は個別に所有する．図1(b)は同一仮想空間上の異なったアドレスにスタック領域を確保する方式であり，図1(c)はスタック領域のみを多重仮想空間化する方式[9)] である．

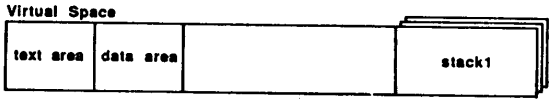
後者の方式を用いると，仮想記憶管理機構により，スレッドのスタック領域間でのデータ保護を完全に行うことができる．しかし，スレッドの切換え／生成において，スタック領域に対応しているアドレス変換



(a) プロセスの仮想空間構成
(a) Virtual space of process.

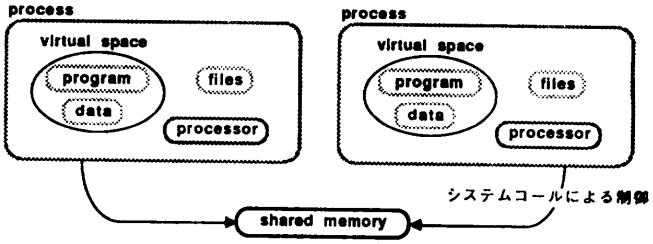


(b) スレッド導入時の仮想空間構成 (1)
(b) Virtual space at the introduction of thread (1).

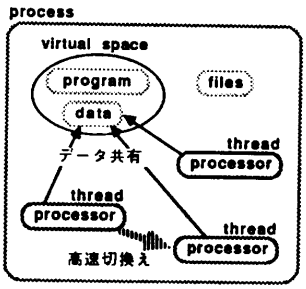


(c) スレッド導入時の仮想空間構成 (2)
(c) Virtual space at the introduction of thread (2).

図1 仮想空間の構成
Fig. 1 Virtual space.



(a) プロセスによる並列処理
(a) Parallel processing with processes.



(b) スレッドによる並列処理
(b) Parallel processing with threads.

図2 プロセスとスレッド
Fig. 2 Process & thread.

* UNIX は AT & T 社の Bell 研究所で開発され，AT & T 社がライセンスしている OS である．

テーブルの切換え／生成が必要となり，高速な処理単位としてスレッドを用いることができない．以降，この理由から，本論文では，前者の方式を前提として考察を行う．

2.3 スレッド導入時のメモリ管理における問題点

図1(b)のように同一仮想空間上の異なった位置にスタック領域を割り当てる場合，個々のスタック領域に仮想空間を大きく割り当てすぎると，プロセス内で実行できるスレッドの数が少なくなる．すなわち，並列処理を実行できる単位が少なくなるため，複数のプロセッサを有効に活かすことができない．逆に，並列実行できるスレッド数を多くするために，スタック領域に割り当てる空間を小さくすると，あるスレッドのスタック領域が割り当てられた空間をオーバフローする可能性が高くなる．関数呼出しのネストなどによって，スタック領域がオーバフローし，他スレッドのスタック領域のデータを破壊してしまうことになれば，ユーザは，自分のプログラムにバグが存在するのか，あるいはスタック領域への空間の割当てが小さすぎたためか判断できない．

同一プロセス内で実行できるスレッド数を多く保ったまま，スタック領域のオーバフローによるデータ破壊を避けるためには，スタック領域間の保護を行わなければならない．スタック領域間の保護策としては，スタック領域に割り当てる仮想空間の一端に境界領域（参照禁止領域）を設け，ここを参照するか否かによって他のスタック領域へ侵入しようとしているかを判定する手段が一般的である[7),8)]．境界領域を参照しようとした場合，スレッドまたはプロセス自体を異常終了させる（これをレッドゾーン保護と呼ぶ）．

しかし，レッドゾーン保護では，あるスタック領域が割り当てられた仮想空間をオーバフローしたとき，たとえ仮想空間の一部が割り当てられずに残っていたとしても，スレッドまたはプロセス自体を異常終了させなければならず，仮想空間を最大限に使用することができない．この対策として，スタック領域の最大値を予測して，あらかじめその大きさ分の仮想空間を確保しておく方式も考えられるが，これはメモリ管理をユーザに負わせることになり，スレッドのユーザインタフェースを著しく低下させることになる．また，このときの予測値が正しいという保証もない．

3. マルチスレッド実行環境のためのメモリ管理方式

並列処理用 OS・SKY-1 のメモリ管理方式は UNIX[10)]のメモリ管理方式を基本として，スレッド導入によって必要となるスタック領域の管理を拡張した*．SKY-1 では，第2章に示した問題点を解決するため，以下の考え方を基本として，スタック領域の保護を行う．

［基本思想］ あるスレッドのスタック領域が他のスタック領域へ侵入しようとした場合，必要な領域を同一仮想空間上で別の位置に拡張する．

この考え方を基本として保護を実現すれば，スレッドのスタック領域間でデータを保護するとともに，仮想空間全体を最大限に使用することができる．また，ユーザがスレッドのスタック領域の大きさを考慮する必要もない．この保護を実現するため，SKY-1 において，スタック領域の伸縮に従って，仮想空間の割当て／解放を行う「動的スタック拡張方式」を開発，導入した．

ただし，この方式はスタック領域に対するより高級な保護を実現するため，ある程度の性能低下を招く．この性能評価については第4章で記述する．

3.1 動的スタック拡張方式

動的スタック拡張方式は仮想空間を複数個の領域に分割し，これを単位としてスタック領域に適宜仮想空間を割り当てる方式である．動的スタック拡張方式では，この割当て単位となる領域を「ユーザページ」と呼ぶ．ユーザページはページング方式の仮想記憶管理におけるページと似ているが，ページが全空間を均等に分割して仮想空間と物理空間を対応させるのに対して，ユーザページはあくまでも，スタック領域に仮想空間を割り当てる場合に単位となる領域である．割り当てられている空間をスタック領域がオーバフローしそうになった場合，使用されていないユーザページを検索し，新たに割り当てる．一般的に，ページの大きさは数から十数 Kbyte であるため，これをスタック領域への割当て単位とすると，オーバフローが頻繁に発生し，性能が大幅に低下する．このため，複数個の

* SKY-1 と UNIX のメモリ管理では，スタック領域の保護のほかに，領域 (Region) 管理が異なっている．従来の UNIX ではデータ領域，テキスト領域，スタック領域が存在し，プロセスの管理情報等が設定される u エリアはいずれかの領域内に置かれていた．スレッドを導入することによって，u エリアをスレッド単位に設ける必要がある．このため，u エリアをまとめて独立させ，u エリア領域を新設した．

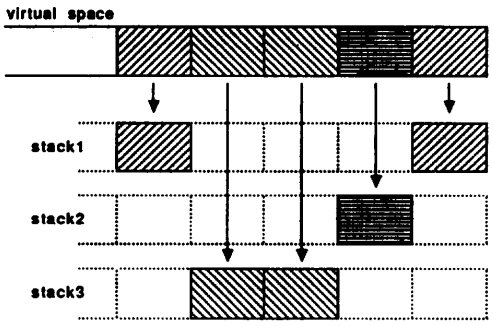


図 3 動的スタック拡張方式
Fig. 3 Dynamic stack allocation.

ページをまとめて1つの割当て単位（ユーザページ）とした．

図3に動的スタック拡張方式を使用した場合の仮想空間の使用形態を示す．図3では3つのスレッドが並列動作を行う例を示している．各スレッドのスタック領域には，初期領域として，1ユーザページが割り当てられている．関数呼出しなどにより，ユーザページをオーバフローした場合には，新たなユーザページを割り当てる．また，関数からの復帰などによりユーザページを使用しなくなれば，そのユーザページを解放する．

動的スタック拡張方式を使用することにより，スタック領域に割り当てられている仮想空間の大きさを常に最小に近い大きさに節約でき（ただし，内部フラグメンテーションは発生しうる），プロセス内に存在しうるスレッド数を増やすことができる．また，仮想空間全体を使い切らない限り異常終了させることはなく，仮想空間を最大限に使用することができる．

しかし，動的スタック拡張方式において，ユーザページ単位でスタック領域に空間を割り当てる場合には，次の2つの課題がある．

- ユーザページのオーバフローチェック手段の実現
- ユーザページ間での論理アドレス不連続性の克服

これらの課題を解決しておかなければ，動的スタック拡張方式を実際に使用することはできない．以下，これらの詳細と，SKY-1 における解決法について示す．

（1） オーバフローチェック手段の実現

スレッドのスタック領域がユーザページをオーバフローした場合，仮想記憶管理を行う OS に対して，オーバフローの発生を伝える必要がある．OS は，オーバフローの発生を認識すると，スレッドに新たなユーザページを割り当てる．

スタック領域の先頭を指すスタックポインタがユーザページ境界を越えて移動したことを調べ，割込みを発生させるハードウェアが存在すれば，オーバフローチェックを容易に行うことができる．しかし，このような特殊なハードウェアは通常の計算機には装備されていないため，SKY-1 では，ソフトウェアによって代行する．レッドゾーン保護と同様，境界領域（参照禁止領域）を設定し，この領域を参照するか否かによって，オーバフローチェックを行う．

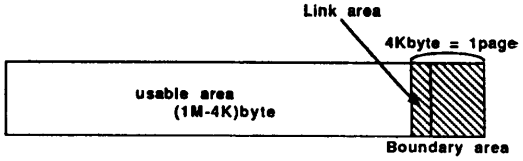
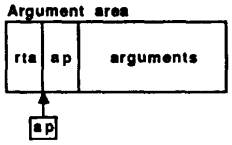


図 4 ユーザページの構成
Fig. 4 Userpage.

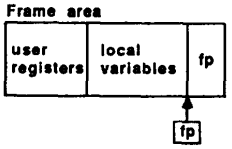
本メモリ管理方式では，境界領域をユーザページ内に設ける（図 4）．ユーザページの大きさは 1 Mbyte，境界領域の大きさは 4 Kbyte（1ページ）であり，ユーザページ中，境界領域の割合は 0.4% にすぎない．境界領域は仮想記憶管理におけるページテーブルのエントリを参照禁止状態とすることによって設定する．さらに，境界領域中には，拡張されたユーザページを結び付けるリンク領域を設け，前に使用していたユーザページのアドレスを格納する．関数からの復帰時において，前のユーザページに復帰する場合，リンク領域に設定された情報を利用する．

（2） 論理アドレス不連続性の克服

図3に示すように，連続していないユーザページを1つのスタック領域に割り当てる場合，スタック領域が論理アドレス上分離されることになる．これは，アドレス上連続していなければならないデータ，例えば，配列等が，論理アドレスの不連続点にまたがって配置される可能性があることを意味する．動的スタック拡張方式では，この問題を解決するため，スタック領域の伸縮単位として，論理アドレス上で連続していなければならないまとまった大きさの「ユニット」を設ける．スタック領域を拡張する際に，次に使用するユニットの大きさが使用中のユーザページの残りより大きければ，新たなユーザページを割り当て，その位置にユニットを確保する．ユニットとしては，関数の引数領域と内部変数領域の2種類がある．ユニットの種類は少ないほうがよい（ユニットを指すためのポインタが少なくなる）が，関数の引数は関数呼出しを行う側，内部変数は関数呼出しを受ける側が設定するも



(a) 引数領域
(a) Argument area.



(b) 内部変数領域
(b) Frame area.
rta: Return Address, ap: Argument Pointer, fp: Frame Pointer.

図 5 ユニットの構成
Fig. 5 Unit.

のであり，これらを1つの領域にまとめることは不可能である．したがって，引数領域と内部変数領域とをそれぞれユニットとし，これを単位としてスタック領域の伸縮を行う．

各領域の構成を**図5**に示す．それぞれのユニットを参照するためのレジスタとして，引数領域に対しては引数ポインタ (ap)，内部変数領域にはフレームポインタ (fp) を用意している．関数呼出しに伴ってこれらのポインタの値を退避する必要があり，ユニット内に退避領域を設けた．また，引数領域には，サブルーチンコール命令の戻り番地を設け，関数からの復帰アドレスとして使用する．内部変数領域には，レジスタ退避領域を設け，関数内で使用するユーザレジスタ (C言語の register 宣言によって使用されるレジスタ) の退避を行う．

これらのユニット内部には論理アドレス上で連続しなければならないという制限を加えるが，ユニット間で論理アドレスが連続している必要はない．

3.2 動的スタック拡張方式における関数呼出し方式

動的スタック拡張方式においては，前節の解決法を行うため，従来の関数呼出し方式を拡張している．本節ではその関数呼出し方式について述べるが，まず最初に，従来の関数呼出し方式について簡単に説明する．**図6**にその方式を示した．なお，ここではプログラミング言語として，C言語を仮定している．また，スタック領域の使用法として，仮想空間上でスタック領域がアドレスの減少する方向へ伸びること，ならびに，スタック領域への情報格納がスタックポインタを減少させた後に行われることを仮定する．

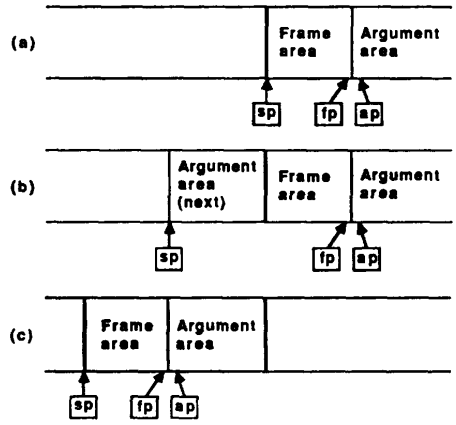


図 6 従来の関数呼出し方式
Fig. 6 Function call.

(a) 関数呼出し前:

スタックポインタ (sp) によりスタック領域の先頭を，引数ポインタ (ap)，フレームポインタ (fp) により関数の引数，内部変数の領域を指す．

(b) 引数の格納:

引数はスタックポインタを使用して格納する．これにより，スタックポインタが引数の先頭に移動する．

(c) 関数呼出し，内部変数領域の確保:

引数ポインタとフレームポインタを新たな関数の位置に移動させるとともに，関数の内部変数領域を確保してその先頭までスタックポインタを移動する．このとき，引数ポインタとフレームポインタの値を退避する．関数からの復帰の際には，退避していた引数ポインタとフレームポインタの値を回復し，(a)の状態に戻る．

このような関数呼出し方式では，引数ポインタとフレームポインタは同一の値となるか，もしくは，その差が一定となる．したがって，通常，2つのレジスタを共通化して，レジスタを1つ節約する方式をとる．しかし，動的スタック拡張方式においては，引数領域，内部変数領域が不連続なユーザページ上に位置することがあるため，引数ポインタ，フレームポインタを個別に用意する．

次に，動的スタック拡張方式における関数呼出し方式について述べる．動的スタック拡張方式での主な処理は関数呼出し処理とそれに伴うユーザページ割当て処理，関数からの復帰処理とそれに伴うユーザページ解放処理である．なお，これらの処理のうち，ユーザ

ページ割当て／解放処理は，OS 内部で行う割込み処理であり，ユーザページがオーバフロー／アンダフローした場合に起動される．

3.2.1 関数呼出し処理

図7に関数呼出し処理の流れを示す．従来の関数呼出し処理に対して，ユーザページのオーバフローチェックが拡張されている．

（a） 関数呼出し前：

スタックポインタがスタック領域の先頭を，引数ポインタ，フレームポインタが引数領域，内部変数領域を指す．

（b） オーバフローチェック（引数領域）：

確保する引数領域の大きさをレジスタ等に格納するとともに，引数領域が境界領域に重なるか否かを調べる．境界領域の大きさが 4 Kbyte（1ページ）となっているため，引数領域が境界領域を越えて確保される場合に対処する必要がある．このため，境界領域のチェックは，以下のように行う．

① 引数領域の大きさが 4 Kbyte を越える場合，オーバフローチェックの命令を複数回実行する．すなわち，スタックポインタの指す位置から 4 Kbyte ごとに読み出しチェックを行い，最後に，確保すべき引数領域の先頭に読み出しチェックを行う．

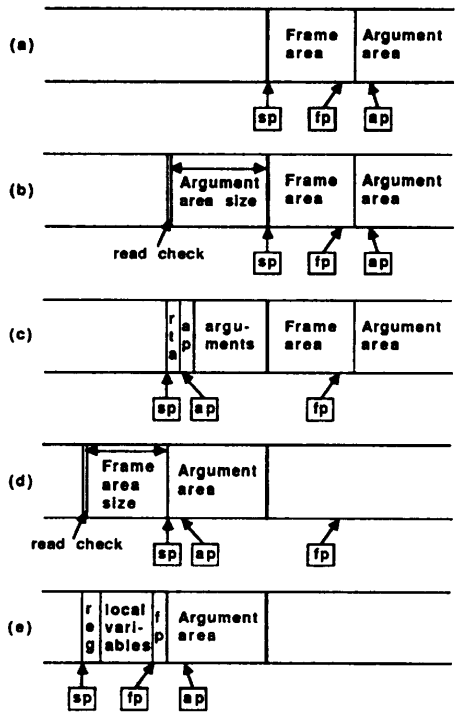


図 7 関数呼出し処理
Fig. 7 Function call in dynamic stack allocation.

② 引数領域の大きさが 4 Kbyte 以下の場合，読み出しチェックを引数領域の先頭だけに施す．

これらの判断はコンパイラによって行える．境界領域から読み出そうとした場合，参照禁止違反の割込みが発生し，ユーザページ割当て処理を行う．

（c） 関数呼出し：

旧引数ポインタの値を退避するとともに，新引数領域の先頭を指すように引数ポインタとスタックポインタを更新する．さらに，サブルーチンコール命令を実行して，新たな関数へ制御を移す．呼出し側のアドレスが退避され，復帰アドレスとして使用される．ここで使用する領域が境界領域に重ならないことは(b)で確認済みである．

（d） オーバフローチェック（内部変数領域）：

関数の内部変数領域のチェックを引数領域と同様の方式で実行する．オーバフロー時にはユーザページ割当て処理を行う．

（e） 内部変数領域の確保：

旧フレームポインタを退避し，その位置を指すようにフレームポインタを更新する．また，スタックポインタを内部変数領域分だけ前に移動する．さらに，関数内で使用するユーザレジスタの内容を退避する．ユーザレジスタを使用しない場合，この領域および処理は存在しない．ここで使用する領域が境界領域に重ならないことは(d)で既に確認済みである．この処理の終了をもって，関数の内部処理に移行する．

3.2.2 関数からの復帰処理

図8に関数からの復帰処理を示す．関数からの復帰処理では，ユーザページのアンダフローチェックが新たに行われる．

（a） 内部変数領域の解放：

ユーザレジスタの値を復帰する．さらに，フレームポインタの値を元に戻すとともに，スタックポインタをスタックの先頭位置に移動する．

（b） アンダフローチェック（内部変数領域）：

内部変数領域の解放により，使用しているユーザページを解放すべきか否かをチェックする．具体的には，スタックポインタの指す位置を読み出すことによりチェックを行う．読み出した領域が境界領域の場合，参照禁止違反の割込みが発生し，ユーザページ解放処理を行う．

（c） 関数からの復帰：

リターン命令により関数呼出し元に制御を戻した後，引数ポインタの値を復帰する．また，スタックポ

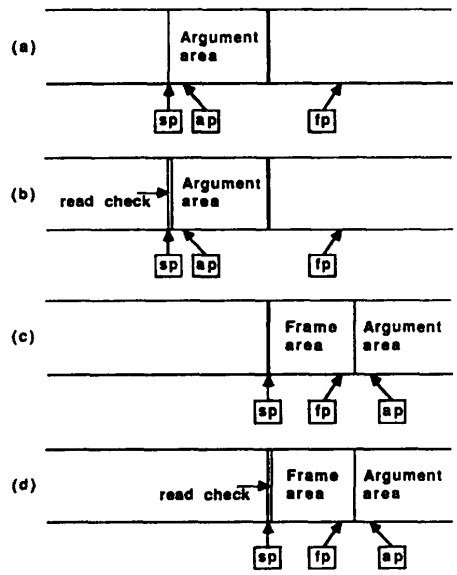


図 8 関数からの復帰処理
Fig. 8 Return from function in dynamic stack allocation.

インタを元に戻す.

(d) アンダフローチェック (引数領域):

ユーザページを解放するか否かのチェックを内部変数領域と同様に行う. アンダフロー発生時にはユーザページ解放処理を行う. この処理の終了後, 呼出し元関数の実行を継続する.

3.2.3 ユーザページ割当て処理

ユーザページ割当て処理は, 関数呼出し時の境界領域からの読み出し (ユーザページのオーバフロー) によって割込みが発生した場合, 使用されていないユーザページを確保するものである. **図 9** にこの割込み処理の概要を示す.

(a) ユーザページの確保:

関数呼出し処理でのオーバフローチェックにおいて, 確保すべきユニットの大きさがレジスタ等に格納されている. これにより, 確保すべきユーザページの数を決定する. 使用されていないユーザページを検索し, 必要な数の連続したユーザページを確保する. ユーザページを複数個確保する場合には, 最後尾のユーザページにのみ境界領域を設定する.

(b) リンク領域の設定:

境界領域中のリンク領域には前ユーザページのアドレスを設定する. 具体的には現スタックポインタの値を格納して, スタック領域の復帰を可能とする.

(c) スタックポインタの移動:

割り当てたユーザページの利用可能領域の最後尾がスタック領域の先頭となる. このため, スタックポインタがリンク領域を指すように移動する.

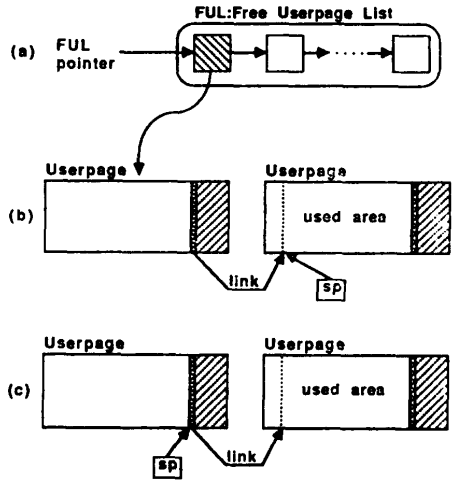


図 9 ユーザページ割当て処理
Fig. 9 Allocation of userpage.

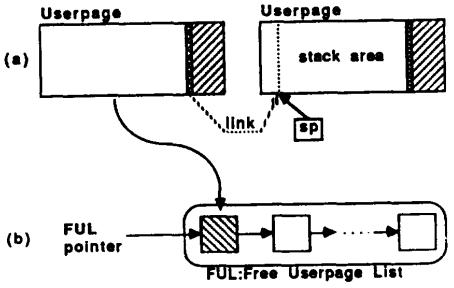


図 10 ユーザページ解放処理
Fig. 10 Detachment of userpage.

3.2.4 ユーザページ解放処理

関数からの復帰時における境界領域からの読み出し (ユーザページのアンダフロー) によって割込みが発生した場合, 使用中のユーザページを解放する. **図 10** に処理の内容を示す.

(a) スタックポインタの復帰:

リンク領域に設定されている値をスタックポインタに復帰する.

(b) ユーザページの解放:

使用しなくなったユーザページを解放し, 他スレッドが使用できるようにする.

4. 性能評価

動的スタック拡張方式では, スタック間の保護と仮想空間の有効利用を同時に実現することができる. しかし, 関数の呼出し/復帰の時点で, オーバヘッドが生じるため, 性能低下を招くことが予測される. 本章

では，動的スタック拡張方式による性能低下の度合いについて測定した結果とその考察を行う．比較するプログラムの関数呼出し方式は以下の3種類である．

（1） 通常の関数呼出し方式-1（フレームポインタと引数ポインタとを共通化）

（2） 通常の関数呼出し方式-2（フレームポインタと引数ポインタは個別）

（3） 動的スタック拡張方式による関数呼出し方式

ここで，フレームポインタと引数ポインタを共通化する方式は，関数呼出しの際に，一方のポインタの値を退避する必要がなく，オーバヘッドが最も小さい．**表1**に各プログラムにおいて，関数の呼出しと復帰に必要な命令数を示した（モトローラ社のマイクロプロセッサ MC 68030 の命令による）．動的スタック拡張方式では，ユニット確保時のオーバフローチェックにそれぞれ2命令，ユニット解放時のアンダフローチェックにそれぞれ1命令必要となっており，その結果，表1に示すように，関数の呼出し／復帰1回あたりのオーバヘッドは一定となる（各ユニットの大きさが 4 Kbyte 以下の場合）．したがって，プログラム中の関数呼出しの頻度によって性能低下率が変化する．

表2に関数呼出しの頻度の異なる各種のプログラムに対して上記3種類の関数呼出し方式を適用し，性能測定を行った結果を示す．プログラム中，Ackerman 関数は関数呼出しの頻度が非常に高く，性能低下が著しい．関数呼出し方式（2）と比較すると 17%，関数呼出し方式（1）に比べると 25% 近く性能が低下している．しかし，その他のプログラムでは，性能低下は5%以下で済んでいる．特に，関数呼出しの頻度が小さい素数計算プログラムや連立方程式の計算プログラムでは，性能低下は 1% にも満たず，動的スタック拡張方式による影響がほとんど存在しないことを示している．

表3にユーザページ割当て／解放処理に要する時間を示す．物理メモリの確保と解放に要する時間が大きいため，それぞれ，2から 3 ms 程度の時間が必要となる．しかし，表2に示すように，ユーザページ割当て／解放処理を行う頻度はプログラム実行時間に比べて非常に小さく，これが性能低下に与える影響は皆無に等しい．さらに，いったん確保した物理メモリをユーザページ解放処理において解放せず，再利用可能とすれば，2回目以降のユーザページ割当て／解放に必要な時間を 1/10 程度に短縮できる．

Ackerman 関数のように特殊な関数を使用したプログラムを作成することは稀である．また，SKY-1 では，動的スタック拡張方式のほかに，従来と同様，レッドゾーン保護を用いることもできる．レッドゾーン保護を用いると，関数呼出しごとのチェックを行わないため，上記の性能低下は解消できる．スタック領域の大きさが不定，不明であるような場合には，性能低下を容認して，動的スタック拡張方式を使用し，大きさが明確な場合はレッドゾーン保護を用いればよい．また，動的スタック拡張方式において各スレッドがどの程度のスタック領域を必要とするかを調べ，これを基にしてレッドゾーン保護によりスタック長を固

表1 関数呼出しに必要な命令数
Table 1 Number of instructions for function call/return.

関数呼出しの形態		通常 (fp, ap 共通)	通常 (fp, ap 個別)	動的スタック拡張方式
呼出し元	呼出し先			
引数無	register 変数無	4	6	12
	register 変数有	6	8	14
引数有	register 変数無	5+X*	7+X	13+X
	register 変数有	7+X	9+X	15+X

* Xは引数の数を示す．

表2 性能評価
Table 2 Performance evaluation.

評価プログラム	A 通常 (fp, ap 共通)	B 通常 (fp, ap 個別)	C 動的スタック拡張方式	性能低下率 $(1-A/C)$	性能低下率 $(1-B/C)$	ユーザページ割当回数	ユーザページ解放回数
Ackerman 関数	64.23	70.22	84.89	24.34%	17.28%	0	0
トランスレータ*	53.36	53.80	56.02	4.75%	3.96%	0	0
巡回セールスマン**	130.92	131.52	133.33	1.81%	1.36%	1	1
マンデルブロー集合計算	49.96	50.17	50.73	1.52%	1.10%	1	1
素数計算	54.21	54.11	54.47	0.48%	0.66%	0	0
LINPACK 計算（連立方程式）	84.37	84.55	84.79	0.50%	0.28%	1	1

実行時間の単位は秒．
* トランスレータ：アセンブリ言語トランスレータ．
** 巡回セールスマン：ホップフィールド型ニューラルネットによる解法．

表 3 ユーザページ割当て時間
Table 3 Execution time for allocating/detaching userpage.

		ユーザページ割当	ユーザページ解放
物理メモリ再利用無		3.2	2.4
物理メモリ再利用有	1回目	3.2	0.2
	2回目以降	0.3	0.2

単位はミリ秒.

定化してプログラムを実行することも可能である.

5. おわりに

並列処理用 OS・SKY-1 では，プロセス内で仮想空間を共有するスレッドを導入した．しかし，仮想空間を共有することは，スレッド間のデータ保護を弱めることになり，ユーザに対して新たな負担を負わせることになる．このため，我々はスレッドのスタック領域を保護し，かつ，仮想空間を最大限に使用する動的スタック拡張方式を開発した．

動的スタック拡張方式は仮想空間を複数個のユーザページに分割し，これを単位として，スレッドのスタック領域に仮想空間を割り当てる方式である．割り当てられているユーザページをオーバフローした場合，新たなユーザページを割り当ててスタック領域を拡張する．動的スタック拡張方式の特徴は以下のとおりである．

（1） ユーザがスタック領域の大きさを指定したり，気にかける必要はない．

（2） スタック領域が使用している仮想空間の大きさを最小に近い値に保つ．これにより，プロセス内で動作できるスレッド数を多く保ち，仮想空間を最大限に使用することができる．

（3） 動的スタック拡張方式の関数呼出しでのオーバヘッドにより，ある程度の性能低下が起こることが予測されたが，通常のプログラムでは 5% 以下の性能低下に留まることを確認した．

謝辞 本研究の機会を与えて頂いた，当社中央研究所の坂東忠秋部長，大甕工場の中西宏明部長，日立研究所の小林芳樹主任研究員に深謝いたします．また，御討論頂いた川端薫氏，御支援，御協力頂いた鈴木昭二氏，宮崎直人氏，小野和子氏，梶田和子氏ほか多くの方々に感謝いたします．

参考文献

1) Accetta, M. et al.: Mach: A New Kernel Foundation for UNIX Development, *Proceedings of USENIX Summer Conference*, pp. 93-112 (1986).
2) Aral, Z. et al.: Variable Weight Processes with Flexible Shared Resources, *Proceedings of USENIX Winter Conference*, pp. 405-412 (1989).
3) Thacker, C. P. et al.: Firefly: A Multiprocessor Workstation, *IEEE Trans. Comput.*, Vol. 37, No. 8, pp. 909-920 (1988).
4) Cheriton, D.: The V Kernel—A Software Base for Distributed Systems, *IEEE Softw.*, Vol. 1, No. 2, pp. 19-42 (1984).
5) Mullender, S. J. et al.: The Design of a Capability-Based Distributed Operating System, *Comput. J.*, Vol. 29, No. 4, pp. 289-299 (1986).
6) Armand, F. et al.: Revolution 89, or 'Distributing UNIX Brings It Back to Its Original Virtues,' *Proceedings of the 1st Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 153-174 (1989).
7) 下山智明：Sun Lightweight Process プログラミング，*UNIX MAGAZINE*, Vol. 3, No. 7, pp. 90-115 (1988).
8) 恒冨邦彦ほか：可変構造型並列計算機の並列オペレーティングシステム—プロセス管理とメモリ管理，情報処理学会計算機アーキテクチャ研究会報告，No. 82, pp. 1-8 (1990).
9) 新井 潤ほか：分散 OS ToM，情報処理学会オペレーティングシステム研究会報告，No. 45, pp. 1-7 (1989).
10) Bach, M. J. et al.: *The Design of the UNIX Operating System*, Prentice-Hall International ed. (1986).
11) 山口伸一朗ほか：並列処理用 OS SKY-1—開発構想，第 39 回情報処理学会全国大会論文集，pp. 1193-1194 (1989).
12) 齊藤雅彦ほか：並列処理用 OS・SKY-1 のメモリ管理方式，第 39 回情報処理学会全国大会論文集，pp. 1195-1196 (1989).
13) 上脇 正ほか：並列処理用 OS・SKY-1 のスケジューリング方式，第 39 回情報処理学会全国大会論文集，pp. 1197-1198 (1989).
14) 齊藤雅彦ほか：並列処理用 OS・SKY-1 のスレッド制御用ライブラリ，第 40 回情報処理学会全国大会論文集，pp. 742-743 (1990).
15) 齊藤雅彦ほか：並列処理用 OS・SKY-1 のスレッドインタフェース，情報処理学会計算機アーキテクチャ研究会報告，No. 83, pp. 145-150, (1990).

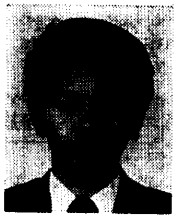
(平成 2 年 9 月 12 日受付)
(平成 3 年 1 月 11 日採録)

齊藤　雅彦（正会員）

昭和39年生．昭和61年京都大学工学部情報工学科卒業．昭和63年同大学院修士課程情報工学専攻修了．同年(株)日立製作所に入社．現在同社日立研究所第八部に勤務．電子情報通信学会会員．

上脇　　正（正会員）

昭和37年生．昭和60年東京工業大学工学部情報工学科卒業．昭和62年同大学修士課程修了．同年(株)日立製作所入社．現在日立研究所に勤務．マルチプロセッサの研究に従事．



山口伸一朗（正会員）

昭和55年九州大学工学部電子工学科卒業．昭和57年九州大学総合理工学研究科情報システム学修士課程修了．同年(株)日立製作所日立研究所入所．制御用計算機のアーキテクチャ，OS の研究に従事．計算機アーキテクチャ，並列処理，分散システムに興味を持つ．