オペレーティング・システム 51－4
（1991. 7. 18）

分散OS XEROにおけるマルチコンテキストの実現方式について

成田 篤信　加藤 和彦　猪原 茂和　益田 隆司

東京大学理学部情報科学科

分散OS XEROでは、コンテキストと呼ばれるプログラムモジュールの管理単位を同一アドレス空間上に複数ロードし、組合せて実行することによって計算を行なう。計算の進行中も、必要に応じてコンテキストのロードと二次記憶への退避を許している。この機構をマルチコンテキスト機構と呼ぶ。特に、コンテキストの実行過程を二次記憶に保存できる点に特徴があり、コンテキストのマイグレートを自然に実現できる。本稿ではコンテキストの主記憶空間への動的ロード、二次記憶への退避、再配置処理の機構の実現方式について述べ、この機構を実現し、処理速度などについての実験を行なった結果を報告する。

DESIGN AND IMPLEMENTATION OF MULTI-CONTEXT MECHANISM OF XERO DISTRIBUTED OPERATING SYSTEM

Atsunobu Narita　Kazuhiko Kato　Shigekazu Inohara　Takashi Masuda

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

On the XERO distributed operating system, jobs are processed by executing combination of basic program modules called "contexts" in one address space. During its execution, contexts can be dynamically transfered between the address space and the secondary storage. We call this mechanism "multi-context mechanism." The notable feature is that a context can be saved in secondary storage even when they are being executed. This feature directly supports context migration. This paper describes the mechanism of dynamic loading, saving and relocating context, and also reports the performance statistics of the implementation of this mechanism.

1 はじめに

今日の計算機環境は、高速なネットワーク網の普及、機種の多様化などにより、規模の拡大と環境の複雑化が急速に進んでいる。このような現状をふまえ、我々は、単なる地理的な分散性のみならず、時間的な隔たり、ハードウェア／ソフトウェアのアーキテクチャの違いをいかにのりこえるかがこれからの重要な課題であると認識し、今後の分散計算機環境には、以下の4つの分散管理機能が必要であると考えた。

地理的分散管理 地理的に異なる地点にある複数の計算機間で情報の交換と共有を可能する機能。

時間的分散管理 異なる時点に実行されているプログラム間で情報の交換と共有を可能とする機能。機械の動作中／停止中に関わらず、情報を永続的に保存することである。

ハードウェア・アーキテクチャ分散管理 異なるハードウェア・アーキテクチャを持つ計算機間で情報の交換と共有を可能とする機能。

ソフトウェア・アーキテクチャ分散管理 異なるソフトウェア開発環境、異なるプログラミング言語で開発されたプログラム間で情報の交換と共有を実現する機能。

これらの分散管理機能を満たしたシステムを開放されたシステムであると呼び、分散 OS XERO では分散透明な抽象データタイプシステムを実現するプログラミングモデルをとり入れることにより、開放されたシステムをめざしている [3] [4]。

本稿では、これらの分散管理機能のうち、特に、地理的分散管理機能と時間的分散管理機能を果たすマルチコンテキスト機構の設計と実現について述べる。

XERO の提案するプログラミングモデルでは、基本的なプログラムの単位をコンテキストと呼んでいる。コンテキストとは、テキスト領域、データ領域、スタック領域からなるプログラムイメージである。複数のコンテキストをアドレス空間上に動的に配置し、それらを組み合わせて実行することにより、計算が進行する。XERO では、この機構をマルチコンテキスト機構と呼ぶ。

コンテキストは、必要に応じて二次記憶からロードされ、逆に、二次記憶へ退避されるが、その際にコンテキストの実行過程を保存したまま二次記憶に退避することができる。たとえ実行途中であっても、コンテキストは、メモリ上だけでなく二次記憶上にも存在できるという性質により、プログラム実行過程の永続的な保存が可能になっている。

コンテキストを手続き付きデータと考えると、手続きにより内部データ構造が隠蔽された抽象データ型を、永続的なものとして実現することが可能になる。また、コンテキストに対して何らかの要求をする時に、そのコンテキストが主記憶上にあるか、二次記憶上にあるのかを透明に見せるインターフェースを自然に構築することができる。これは時間的な分散性への1つの解答となっている。

また、二次記憶へ退避したコンテキストを異なるアドレス空間や異なるホストへ転送することが可能となるので、負荷分散など、地理的分散管理機能も果たしていることになる。頻繁に通信を行なうコンテキストを同じアドレス空間にマイグレートして実行することにより、コンテキスト間の通信を高速に行なえるといった利点も考えられる。

本文では XERO のプログラミングモデルを詳しく述べ、このプログラミングモデルの実現のためのマルチコンテキスト機構について述べる。コンテキストの二次記憶への退避のことを特にアンロードと呼び、コンテキストの動的なロード／アンロード機構の設計と実現について詳しく述べる。

同一アドレス空間への複数コンテキストの動的ロード／アンロード機構の実現の際に最も問題となるのは、コンテキストをアドレス空間にロードした時の再配置処理の問題である。コンテキストはアドレス空間の任意の場所にロードできなければならないので、いかに、コンテキストを再配置可能な形式にし、再配置のためにどの様な処理をすればよいかを考慮しなければならない。この問題を解決するための、コンテキストの実行コード部分を再配置可能にする方法と、データ領域のポインタ変数の補正方法について論じている。

最後に、この機構を実際に実現しその上で実験プログラムを作成、実行させてみた結果、これまでに開発されてきたプログラムを特に変更することなく利用することができることが分り、また、ロード／アンロードを行う際の再配置処理にかかるコストが、現実的なものであることが分ったので報告する。

2 XERO のプログラミングモデル

XERO のプログラミングモデルは次の 3 つの概念で整理される。(図 1)

タスク (Task) 仮想アドレス空間のこと。

コンテキスト (Context) プログラムやデータの基本的な単位であり、テキストセグメント、データセグメント、スタックセグメントの組によって構成される。

スレッド (Thread) カーネルなどによって多重化された、仮想 CPU。

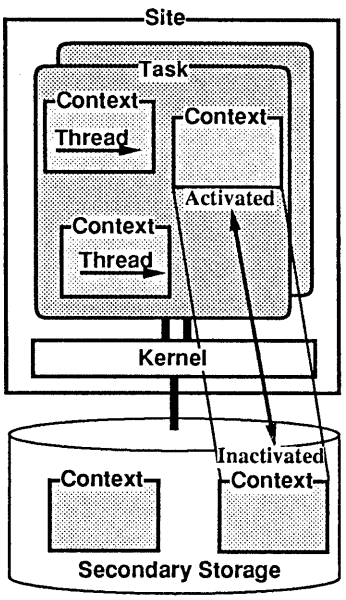

図 1: XERO のプログラミングモデルの概念

XERO における計算は、タスク上にロードされたコンテキスト上をスレッドが走行することによって進行する、と表現することができる。

XERO では、データや実行ファイルなどは全てコンテキストとして扱われる。コンテキストは、その中に含むセグメントの組合せによって以下の 3 つのタイプに分類される。

Type I	Data
Type II	Data+Text
Type III	Data+Text+Stack

タイプ I コンテキストは、スレッドによって実行可能な命令を含まないデータだけのコンテキストである。タイプ II コンテキストは、データセグメントにテキストセグメントが付加されたものである。タイプ II コンテキストは、例えば、内部データを隠蔽した抽象データ型を構成するために用いられる。タイプ II コンテキストにスレッドが割り当てられ、実行が開始されるときには、実行状態を格納するためのスタックセグメントと、CPU 状態を保存する領域が付加され、タイプ III コンテキストに変わる。

また、どのコンテキストも **activated** または **inactivated** のいずれかの状態をとる。**activated** な状態とはコンテキストがタスクにロードされている状態のことをいう。反対に、**inactivated** な状態では、コンテキストは二次記憶に存在し、永続性が保証される。

タイプ III コンテキストさえも **inactivated** な状態をとることができる点に注意されたい。XERO では、プログラムの実行状態を first class object として扱えるようになっている。二次記憶上に退避されているコンテキストとも通信が行なえる。これは、他の分散 OS にはない XERO の大きな特長である。

このプログラミングモデルを用いることにより、異なるアドレス空間で実行されていた頻繁に通信を行なうコンテキストを同じアドレス空間にマイグレートしてくることにより、カーネルを介さない通信が可能となるなどの利点がある。

3 コンテキストの動的ロード / アンロード実行方式

1 つのタスクに、複数のコンテキストを動的にロード/アンロードする機能について前章で述べた。この章では、この方式の実現機構について述べる。

3.1 実現の概要

タスクが作られると、まずはじめに、タスクスーパーバイザ (以下 TSV) と呼ばれるプログラムが、ロードされ、制御がそこに移される。TSV は、ユーザレベルで実行されるプログラムであるが、カーネルと協調することにより、コンテキストの配置管理、ロード/アンロード処理、仮想プロセッサの割り当て、アドレス空間内のシンボルテーブルの管理やシンボルのリンク、などの役目を持つ [1] [2]。カーネルとコンテキス

ト（ユーザプログラム）のなかだちを行なうものである。

TSV が扱う実行可能なコンテキストにはタイプ II とタイプ III があるが、TSV はタイプ II コンテキストをロードするとスタックセグメントなどを付加し、タイプ III コンテキストに変換している。

コンテキストは、いつ、アドレス空間のどこにロードされるか分らないので、常に再配置可能でなければならない。CPU によってはセグメンテーションの機能により、再配置が容易に実現できる場合もあるが、その機能を持たない環境では、コンテキストのロード／アンロード機構を実現する場合、コンテキストのロードされる位置により、テキストセグメント及びデータセグメントを、何らかの方法によって再配置処理を行わなければならない。

再配置可能コンテキストを実現するためには次の 2 つの方法が考えられる。

(1) ロード位置に依存しているコード及びデータをコンテキストロード時に直接書き換えておく方法

(2) あるレジスタにコンテキストのロード位置を保存するようにしておき、メモリ上の情報はレジスタ相対の表現としておく方法

まずコード部分について考えてみると、(2) の方法は、データなどの参照がレジスタ相対となってしまうため実行速度の点で不利となるが、(1) の方法では複数のコンテキストがコード部分を共有できないという欠点があるため、(2) の方法がよいと判断した。

一方、データセグメントの場合は、複数のコンテキストからデータセグメントが共有されることが一般的に多くないと考えられ、(1) の方法を採用することにした。

テキストセグメントにおける再配置可能なオブジェクトコードの生成法と、データセグメント、スタックセグメントにおける再配置処理方法について、以下に述べる。

3.2 再配置可能な実行コードの生成

今回の開発環境は CPU m68030 であり、セグメンテーション機能を持たないため、コンテキストを再配置可能にするために、再配置可能な実行コードを得るようにする必要があった。

3.1 章で述べたように、コンテキストのテキストセグメントのコードは、シンボル参照を全てレジスタ相対のコードにし、再配置の際にテキストセグメントを書き換える必要がないようにした。コンテキストの実行中は、CPU m68030 のレジスタの一つである a5 レジスタにコンテキストのロードアドレスを常時格納するようにし、実行コードの中でシンボル参照を行なう部分を全て a5 相対アドレッシングでおこなうようにした。

再配置可能なコードの実現は、既存のコンパイラ GNU C Compiler (以下 GCC) [5] に次のような変更を施すことによって行なっている。具体的な出力例は次のようになる。

通常のコード	a5 相対コード
`move.l d1,_var`	`move.l d1,(_var,a5)`
`jsr _printf`	`jsr (_printf,a5)`

3.3 データセグメント、スタックセグメントの再配置処理

コンテキストをアドレス空間の異なる位置に再配置する場合、データセグメント、スタックセグメント内のポインタ変数の扱いが問題となってくる。コンテキストの位置が変ると、ポインタ変数の値は本来指しているべきデータのアドレスとは異なったものとなってしまい、データ構造を正しく再現できなくなってしまう。この問題を解決するためには、コンテキストの再配置が行なわれるたびにポインタ変数を正しく補正してやる必要がある。

データセグメント、スタックセグメント内に存在する全てのポインタを表しているデータに対して、コンテキストの前回のロードアドレスと現在のロードアドレスの差を足し込むことにより、補正を行なっている。a5 というレジスタにコンテキストのロード位置が格納されているので、これを使って表すと

[New Pointer Value]
= [Old Pointer Value] + (new_a5 - old_a5)

という変換を全てのポインタ変数に施すことになる。ここで注意すべき点は、NULL ポインタの存在である。多くのプログラミング言語の流儀では NULL (=0) ポインタを特殊なポインタとして扱う。そこで、ポインタ変数の補正の際に、NULL ポインタだけは、例外として補正を施さないようにしている。

また、全てのポインタ変数に対して補正を行なわなければならないので、コンテキストはデータ領域内の全ポインタ変数の位置を記録しておかなければならな

い。コンテキストがコンパイラによって出力されたシンボル情報および型情報を常に保持するようにしておくことにより、ポインタ変数の検索を実現している。コンテキストがロードされた時、シンボル情報、型情報をもとに、ポインタ変数を検索、補正を行なってゆく。

このシンボル情報は GCC の -g オプションにより出力される。タイプII、タイプIII コンテキストはいつでも再配置を行なえるように、このシンボル情報を常に保持していなければならない。

タイプIII コンテキストの、データセグメントは、大きく分けると静的データセグメントとヒープセグメントに分けられる。この2つのセグメントとスタックセグメントの3つのセグメントについてポインタデータの検索、補正の方法を考えてみる。

まず、あるポインタ変数がどこのセグメント内のデータを指しているのかは一般には分らない。従って、指しているデータの存在するセグメントによってポインタ変数の補正値を変えることは困難である。そこで今回の実現では、1つのコンテキスト内ではテキストセグメントを含む各セグメント間の相対位置は変らないものとする制約を設けた。この制約を設けることにより、ポインタ変数に対する補正値が一定になり、データセグメントの再配置を容易にしている。

静的データセグメント、ヒープセグメント、スタックセグメントの3つのセグメントは、どれも少しずつ異なる性質を持ったものであり、実際の再配置処理時のポインタ変数の検索方法は、各セグメントごとに異なる方法で行なわれる。3つの各セグメントについて、再配置の際のポインタ変数の検索方法を以下に述べる。

3.3.1 静的データセグメント

静的データセグメントとは、コンパイル時にアドレス位置、データ型が決まってしまう変数領域のことである。GCC の -g オプションの出す型情報をもとに、宣言されている全ての変数に対し、ポインタ変数の存在している部分を検索し、全てのポインタ変数に対し、補正を行なう。

3.3.2 ヒープセグメント

各コンテキストはそれぞれ個別にヒープ領域を持ち、必要に応じヒープ領域からメモリを確保している。静的データセグメントがセグメントサイズ固定であるのに対し、ヒープセグメントは確保される領域が時とともに変わる。さらに静的データセグメント内の型情報はコンパイル時にほぼ決まるが、ヒープセグメントの型情報はコンパイル時には一般的に決定できない。このような性質の相違により、データセグメント部とヒープセグメント部とのポインタ変数の再配置処理は多少違ったものになっている。

```
      :
ptr = malloc( sizeof(struct STAG)*100 );
assign_type( ptr, "struct STAG", 100 );
      :
free ( ptr );
unassign_type( ptr, "struct STAG", 100 );
      :
```

図 2: 型割当て関数の使用例

ヒープセグメントに動的に確保される領域は、コンパイラの出力するシンボル情報だけからでは、変数の型を判別できない。そこで、ヒープセグメントの再配置を可能とするために、ユーザが、明示的に型を宣言できる機能を設けた。2つの関数

```
assign_type( ptr, type, elements )
unassign_type( ptr, type, elements )
    char *ptr; /*pointer to the variable*/
    char *type; /*type name (of an element)*/
    int elements; /*number of elements*/
```

を用いることにより、プログラマーが、ヒープセグメントのデータに対して、明にデータの型を宣言することができる。

assign_type() を用いることにより、以降は ptr で始まるメモリ領域が、type で示される型を持つ変数の elements 個の配列であるとみなされる。また、unassign_type() は、assign_type() で行なわれた型の宣言を無効にすることができる。これらは通常、図2のように用いられる。

このように、これら2つの関数を用いることにより、malloc() で確保されるヒープ領域に関してもコンテキストロード時に適切に再配置処理を行なえる。

この型割当て関数は、静的データセグメントに対しても有効であるので、ポインタを含む共用体変数の再配置処理などにも用いられる。ポインタを含む共用体変数の場合、その共用体に格納されているデータがポインタであるのかそうでないのかは、プログラムの実行とともに変化しシンボル情報だけからでは判断でき

ない。そこで、プログラマーが、その共用体には現在どのような変数が入っているのか実行時に宣言することができるようになっている。この2つの関数によってコンテキストのロード時に細かな再配置制御を行なうことができる。

3.3.3 スタックセグメント

スタックセグメントには、呼び出されている各関数のスタックフレームがプログラムの状態に応じて積み重ねられており、格納されているデータの型は静的データのように位置固定ではない。

まず、シンボル情報をもとに、各関数のスタックフレームがどのようなデータ構造となるかを解析する。次に、最下位のスタックフレームから順に、そのスタックフレームがどの関数のものであるかを調べ、引数、ローカル変数に含まれているポインタ変数の値を適切に変換する。さらに、各スタックフレームに保存されているリターンアドレス、フレームポインタも同様に変換する。

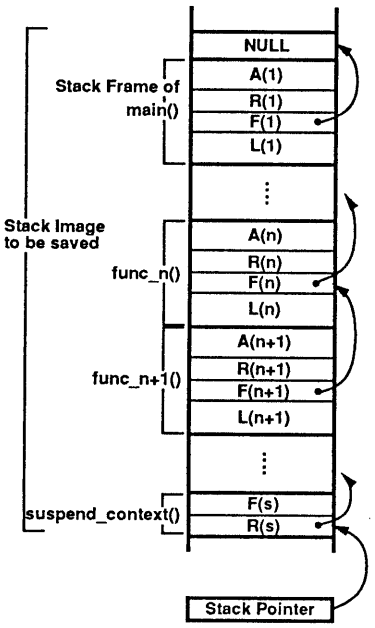

図 3: スタックセグメントの構造

図3 は、コンテキストをロード／アンロードする時のスタックセグメントのイメージである。スタックセグメントは、Auguments A(n) , Return address R(n), Frame pointer F(n), Local variables L(n) によって構成される。n はスタックフレームの深さを示している。

スタックフレームはフレームポインタにより順次たどることができる。コンテキストがロード／アンロードされる時は、タイプ III コンテキストは必ずコンテキストの一時中断を行なう suspend_context() という命令を実行した状態にある。その時点でのスタックポインタは、必ず suspend_context() のスタックフレームを指している。また、コンテキスト実行開始時に NULL ポインタだけを含む疑似スタックフレームを作っておくことにより、どのスタックフレームが最上位のものかが容易にわかるようになっている。

ここで、n 番目のスタックフレームがどの関数のものであるかを調べる場合、n+1 番目のスタックフレームの Return address R(n+1) を調べる。R(n+1) の示すアドレスが、どの関数の領域のアドレスかを見ることにより、n 番目のスタックフレームの関数が特定できる。実際のプログラムが、

```
func_n()
{
        :
    func_n+1();
lab:
        :
}
```

のようになっている場合、R(n+1) は、ラベル lab: のアドレスとなる。R(n+1) が func_n() 内のアドレスであることより、n 番目のスタックフレームは func_n() のものであることがわかる。

4 マルチコンテキスト機構をもつ TSV の実現について

現在、Sony NEWS ワークステーション (CPU m68030) 上のユーザーレベルプログラムとして、コンテキストの動的ロード／アンロードを行なう機能を持った TSV を実現し、この TSV の管理するアドレス空間内にコンテキストを動的にロード／アンロードし、実行できるようになっている。

現在 TSV には以下のような TSV システムコールがインプリメントされている。

load_context	コンテキストのロード
unload_context	コンテキストのアンロード
kill_context	コンテキストの消去
exit_context	コンテキストの終了
suspend_context	コンテキストの中断
assign_type	再配置のための型宣言
unassign_type	再配置のための型宣言キャンセル
common_lib	共有ライブラリの呼び出し

タイプIIコンテキストは、UNIX実行ファイル形式となっている。ただし、実行コードが再配置可能となっている点、シンボル情報を保存している点が、異なっている。タイプIIコンテキストがひとたびロードされると、TSVにより、スタックセグメント、ヒープセグメントが付加され、タイプIIIコンテキストとして扱われるようになる。

今回行なった実現では、Cで書かれたソースコードはほぼ制限なく利用でき、これまでに書かれたプログラムを利用できるようになっている。従来のライブラリ関数も使用できるが、入出力関数、メモリ割り当て関数など、一部のライブラリ関数には同一アドレス空間内では複数存在すると、不都合の生じるライブラリ関数がある。そのようなライブラリ関数をコンテキスト内にリンクし、組み込んでしまうと、同じライブラリをリンクしている別のコンテキストを同じアドレス空間にロードすることができなくなってしまう。

本来ならば、ロードされたコンテキストの未解決シンボルを読むことにより、TSVが動的に必要な共有ライブラリ（コンテキスト）をロードするようにすべきであると考えられるが、今回は、シンボルの動的リンクの機構までは動作していないので、共有ライブラリをTSV自身が持ち、コンテキストからの要求により、共有ライブラリを実行するようになっている。

同一アドレス内の複数コンテキストの実行は、suspend_context() という命令を用いることにより、ノンプリエンプティブに複数コンテキスト間で実行の切替を行なっている。

コンテキストのアンロードが行なわれる時、オープンされているファイルの状況など、カーネルに保存されている状態までは保存されないことに注意する必要がある。従って、コンテキストはアンロードされる時には、アンロードされるための一切の処理をしておかなければならない。この問題に対しては、コンテキストのアンロードは原則的に、他のコンテキストから強制的にアンロードされるのではなく、それぞれのコンテキストが自発的にアンロードを発行するようにすることで解決している。外部からの要求でコンテキストがアンロードされる場合にも、直前に、コンテキスト自身にアンロードのための処理をする機会を与えることで解決する。

マルチコンテキスト実行環境の実現にはTSVの作成の他に、コンテキスト用スタートアップルーチンを用意し、必要なライブラリ関数も全てa5レジスタ相対のコードにコンパイルし直す必要もある。

表1: コンテキストのロード/アンロード

context size	user time	system time	response time
18Kb	11.3	25.3	100.3
36Kb	23.1	40.4	170.8
56Kb	34.3	54.7	254.5
107Kb	78.3	92.6	376.6

(単位:1/1000 sec.)

表2: UNIX read/wirte

context size	user time	system time	response time
18Kb	0.1	12.5	50.3
36Kb	0.1	21.5	100.0
56Kb	0.1	31.6	133.6
107Kb	0.1	61.5	200.3

(単位:1/1000 sec.)

5 実験

これまでに、TSVの機能のうち、コンテキストのロード／アンロードの機能と、簡単なスレッド割当機能の実装行なった。この環境のもとで、いくつかの簡単なプログラムを作成し、得られたコンテキストを、動的にタスク内に配置し、実行させてみたが、問題なく動作している。コンテキストをアンロード+ロードし、異なるアドレスにロード(再配置)した場合も、再配置処理が正しく機能している。

そこで、この実装をもとに、各サイズのコンテキストの、ロード／アンロードにかかる時間を計測した。シングルユーザモードにて、いくつかの大きさのコンテキストをタスクへのロード／アンロードを繰り返し行ない、1回（ロード+アンロード）あたりの時間を測定した。表1に結果を示す。user timeは、TSVによる処理にかかる時間、system timeはカーネル

による実行時間、response time は、ディスクアクセス時間も含めた実行開始から終了までにかかる全時間である。

比較のために、同じ大きさの UNIX (Sony NEWS-OS 3.3) のファイルの read/write にかかる時間の測定結果を表2に示した。(read+write) の1回あたりの値である。

system time に注目すると、同じサイズの UNIX ファイルを1回の read/write 命令で書き込み読み込みを行なう場合に比べ、コンテキストのロード／アンロードの入出力処理のためにかかる時間は、各セグメントごとに分割して read/write している分だけ余計に時間がかかり、system time は2倍程度になることがわかる。

また、UNIX ファイルの read/write プログラムでは、ユーザモードで実行される部分がほとんど0に等しいことより、コンテキストのロード／アンロードの計測での user time の値は、TSV による再配置処理にかかる時間にほぼ相当していると考えられる。再配置処理にかかる時間は、型割り当て宣言の数、シンボルテーブルの大きさ、スタックの深さなどによって、多少のばらつきが生じると考えられるが、一般的なコンテキストの場合、system time の約50% から 80%、応答時間 (response time) の 10% から 20% の時間がかかる。

response time は、バッファリングの効果により、純粋な disk read/write 時間が計測に反映されてはいないが、コンテキストのロード／アンロードは同じ大きさの、UNIX ファイルの転送にかかる時間の倍程度の時間で完了することが分った。

TSV による再配置処理にかかるコストは、シンボルテーブルの大きさなどにより、変ってくるが、現段階では、コンテキストのもつ、シンボル情報は多少冗長になっている。ポインタ変数を含まない変数の宣言を登録しないようにするなどのシンボルテーブルの最適化により、速度向上の余地が残されていると思われる。

6 まとめ

XERO のプログラミングモデルでは、特に、実行途中のコンテキストのイメージにも永続性を与えている点に大きな特徴があり、このことによってコンテキストのマイグレーションが自然に実現され、通信の高速化などが期待されることを述べた。また、コンテキストのロード／アンロード機構の実現の際には、コンテキストのアドレス空間内の異なる位置への再配置をどのように可能にするかが問題となるが、コード部は全てのシンボル参照をコンテキストのロードアドレス相対にし、データ領域ではシンボル情報をもとにポインタ変数を補正することにより、再配置が可能であることを述べた。実際にこの機構を実現し、いくつかのプログラムを作成してみた結果、わずかな制約があるものの、問題なく再配置が行なえることが分った。

コンテキストのロード／アンロード機構を実際に実現し、この処理にかかるコストを測定した。再配置処理およびコンテキストの転送にかかるコストは、コンテキストのコピーにかかるコストに対し、2倍程度であり、現実的な性能が出せるということが分かった。

今後の課題としては、異なるホスト間のコンテキストの転送機構の実現と性能評価、コンテキスト間の RPC プリミティブの実現、タスク内のシンボル管理と動的リンク機構の実現、などが挙げられる。

参考文献

[1] 猪原, 加藤, 成田, 益田. A Thread Facility-Based on User/Kernel Cooperation in the XERO Operating System. To appear in COMPSAC '91, Tokyo, September 1991.

[2] 猪原, 加藤, 益田. オペレーティングシステム XERO のマルチスレッド機構. 情報処理学会オペレーティングシステム研究会, 48-6, September 1990.

[3] 加藤, 猪原, 成田, 千葉, 益田. 無指向的オペレーティングシステム XERO の設計. 情報処理学会コンピュータシステムシンポジウム, 91-1, March 1991.

[4] 加藤, 猪原, 成田, 千葉, 益田. Design of the XERO Open Distributed Operating System. Submitted for publication.

[5] R. M. Stallman. Using and Porting GNU CC. Free Software Foundation, Inc.