マルチメディア通信と分散処理 58−22
（1992. 11. 20）

異機種間プロセス移送メカニズムの試作

長澤 育範　相田 仁　齊藤 忠夫

東京大学 工学部

プロセス移送は実行中のプロセスを分散システム内の他のコンピュータに移動して再配置する機能である．これは、負荷分散、資源共有、信頼性・可用性の向上の手段として研究されてきた．同機種間のプロセス移送は多くの分散OS上やUNIX上で制限を設けた形などで実現されているが、これを異機種間で行なうには、同機種間の時には生じない問題も解決する必要があり、さらに困難である．

筆者らは、この異機種間のプロセス移送を実現する方法について検討し、プリプロセッサによってソースコードを異機種間移送が可能な形に変換する方式を考案した．そして、その方式に基づいて異機種UNIXワークステーション間で試作した結果、制限事項はあるものの、実際にプロセスを移送することができた．

本稿では、筆者らが考案した異機種間プロセス移送の方法について、その内容とSun-3とSun-4ワークステーション間で試作した結果を制限事項と共に述べる．

A Prototype of Heterogeneous Process Migration Mechanism

Ikunori NAGASAWA,　Hitoshi AIDA　and　Tadao SAITO

The University of Tokyo

7–3–1, Hongo, Bunkyo-ku, Tokyo, 113, JAPAN

Process migration is an operation of dynamic relocation of running processes among the computers in a distributed system. It has been explored for a number of years as a means of load balancing, resource sharing, failure robustness and etc. Homogeneous process migration has been implemented on many distribued operating systems, and on UNIX with some restrictions. Process migration among heterogeneous machines, however, is more difficult because it needs to solve problems which do not occur in the homogeneous case.

We investigated a method for heterogeneous process migration, and designed the method that translates the source code to a migratable form between heterogeneous machines by means of the preprocessor. Implemented prototype succeeded in migrating processes between heterogeneous UNIX workstations, with some restrictions.

In this paper, we discribe our approach to heterogeneous process migration, and its prototype implementaion between Sun-3 and Sun-4 workstations, along with its restrictions.

1 はじめに

近年、コンピュータの低価格化やネットワーク技術の発達に伴い、他数のコンピュータをLANで接続して分散システムを構築しようという傾向が増大している．この状況下で、NFSなどのファイル共有技術が発達してきた．

しかしCPU資源に関しては依然計算機を単位とした利用であり、十分に活用しているとはいえない．現在のUNIXではプロセスの実行開始時に実行するコンピュータを選択できる程度である．これに加えてCPUの負荷などに応じて実行中のプロセスを他のコンピュータに移動して再配置できれば、負荷分散によるスループットの向上、信頼性・可用性の向上を図ることができる．

このように、実行中のプロセスを他のコンピュータに移動して再配置する機能をプロセス移送（process migration）という．現在までに、同機種のコンピュータ間のプロセス移送は多くの分散OS上や、UNIX上で制限を設けた形などで実現されてきた．しかし、ネットワーク上には異機種のコンピュータが混在しているのが一般的であるので、異機種間でもプロセス移送を行なうことができるのが望ましく、それによってプロセス移送の有用性はさらに向上するといえる．ところが、異機種間でプロセス移送を行なうには、同機種間の時には生じない問題も解決する必要があり、さらに困難である．

筆者らは異機種間でプロセス移送を実現する方法について検討・考察し、コンパイル時にソースプログラムにプリプロセッサによる処理を加えることで異機種間のプロセス移送に生じる問題を解決する方法を考案した．そして、第一段階としてSun-3とSun-4ワークステーション間で試作を行ない、制限事項はあるものの実際にプロセスを移送することが可能であるのが確認できた．我々は、このシステムを異機種間プロセス移送サブシステムHPMS（Heterogeneous Process Migration Subsystem）と呼んでいる．このプリプロセッサは、基本的にソースプログラムレベルの書換えしか行なわないので、新たなマシンへの移植は容易である．

本稿では、筆者らが考案した異機種間プロセス移送の方法について、その内容と試作した結果を制限事項と共に述べる．

なお、移送されるプロセスとして本稿では、C言語で記述されたプログラムで1スレッドのものを対象としている．また、移送対象となるプロセスは、比較的長時間実行されるものを想定している．

2 異機種間でのプロセス移送の問題点

プロセス移送を行なう際には、プロセスのアドレス空間の内容やプロセスの実行状態に関する各種の情報を移動することが必要である．

同機種間の移送は、プロセスのアドレス空間やレジスタの内容をそのまま移送先にコピーして、実行を再開することで実現できる．

しかし、異機種間では命令コードやレジスタの数及び扱い等が異なるため、移送して意味のあるものは基本的にデータの内容だけである．そのデータに関しても

- データの存在する領域の開始番地の違い
- データの割り付け規則（Alignment）の違い
- データの内部表現の違い（Endianなど）の違い
- アドレスを値として持つ変数（ポインタ）の存在

などがあるため、そのまま移送して、移送先で移送元と同じ所に読み込んでも意味をなさない可能性がある．（今回試作の対象としたSun-3とSun-4のデータ割り付け規則は表1のようである．）

表1: 各データ型に対する記憶割り付け規則

データ型	内部表現		
	サイズ	アラインメント	
		Sun-3	Sun-4
char	1 byte	—	—
short	2 byte	2 byte	2 byte
int	4 byte	2 byte	4 byte
long	4 byte	2 byte	4 byte
float	4 byte	2 byte	4 byte
double	8 byte	2 byte	8 byte
pointer	4 byte	2 byte	4 byte
structure	—	2 byte	メンバ中で最大のもの

1 byte = 8 bits

Endian : Big Endian

移送時の実行再開位置についても、プログラムカウンタの値が意味をなさないため、別の方法で知らせなければならない．

以上のことから、異機種間で移送を行なうには、同機種間の場合には必要でない処理や表現の変換を行なう必要がある．

これらの問題に対して、全マシン上に仮想のハードウェア（抽象マシン）のシミュレータ/インタプリタ/エミュレータを設けて共通の論理表現を与え、プロセスをその抽象マシン上で走らせる方法がある．この方法は、移送自体は基本的に抽象マシンという同機種間の移送になるので比較的容易であるが、実行時に逐次、インタプリタによって共通表現から物理表現への変換を行なうので、移送が行なわれない場合でも走行速度が遅くなってしまう．

そこで我々は異機種間のプロセス移送の際に生じる問題に対して、コンパイル時にプリプロセッサによる処理を加えることによって、実行時にはできるだけ速度を落さずに、移送時に最小限の必要な変換を行なうというアプローチをとった.

次節からは、アドレス空間の各領域の移送に必要となる処理について述べる.

3 テキスト領域について

テキスト領域にはそのプロセスが実行している命令コードが存在する. これは、機種によって全く異なるので、移送しても意味がない. そこでここでは、機種ごとに実行可能ファイルを持ち、移送先のマシンで機種に対応した実行可能ファイルを実行することで、テキスト領域を再構築する.

4 データ領域移送用の処理

データ領域にはプログラムのグローバル変数、関数内の静的 (static) な記憶クラスを持つ変数などが割り付けられる. これらは、コンパイル時に静的に割り付けられ、リンカによって絶対アドレスが決められる. そこでここでは、

- 構造体は各メンバのオフセット及び全体のサイズが全機種で同一になるようにする.
- 各変数が全機種で同一アドレスに割り付けられるようにする.

という方法を用いる.

この場合、移送によってデータのアドレスは変わらないのでデータ領域の変数へのポインタはそのまま利用できる.

データ領域が全機種で同一アドレスに割り付けられるような実行可能プログラムは、

- 構造体は定義部でプリプロセッサによりダミー挿入を行ない、各メンバのオフセット及び全体のサイズが全機種で揃うようにする.
- 記憶指定子が static の変数は、識別子名を一意なものに付け変える. テキスト内の識別子も可視範囲 (scope) に応じて置き換える.
- テキスト部と変数定義部を分離して、データ定義部から全ての機種のメモリ割り付け規則に合うようなデータ割り付けを行なうアセンブリルーチンを生成してコンパイル・リンクする. テキスト部では、これらの変数は外部 (extern) 参照になる.

という方法で生成する. このコンパイルの手順は、図1のようになる.

ここでデータ割り付け用のアセンブリルーチンは、各識別子に対して

- 変数の識別子のグローバル宣言
- 識別子に絶対アドレスを割り当てる命令

の2つの命令を行なうのみであり、データ割り付け情報から機械的に簡単に生成できる. 例えば、Sun-3 及び Sun-4 で変数 a にアドレス 0x00004000 を割り当てる場合、

```
.globl _a
_a = 0x00004000
```

というアセンブリルーチンを生成すればよい.

データ領域の移送にはファイルを利用する. 移送時に、移送元でデータ領域全体をビットイメージでファイルに書き出し、移送先で同じ位置に読み込むことによって移送を行なう. 以降で述べる他の領域の移送にもファイルを利用する.

5 ヒープ領域移送用の処理

ヒープ領域は、一般的には実行時に malloc のような関数を使って獲得し、free 関数で解放する (他のヒープ領域割り付け関数はこの2つの関数を利用して実現されている).

これに対してはまず、リストの先頭を示す変数はデータ領域の他の変数と同じ方法で移送する.

そして、実行開始時にはデータ領域は前節の方法によりどの機種でも同じアドレスまで伸びることになるので、

1. malloc 内で sbrk を呼び出した時、アドレスチェックを行ない、全機種で最も厳しいアラインメントを満たすアドレスからのメモリを保持する.
2. malloc が呼び出された時、最も厳しいアラインメントの規則 (Sun-3 と Sun-4 の場合 Sun-4 の double の 8byte) の倍数かつ全機種で同じ大きさを単位として割り当ていくようにする.

ということを行なえば、そこに格納されるオブジェクトは全機種で同一番地に割り当てられる. free の方は割り当てた領域をリストに戻すだけなので、実現は容易である.

このような動作を行なう malloc 及び free を作成し、ライブラリとして利用すれば、ヒープ領域の移送はデータ領域と合わせて行なうことができる.

6 スタック領域移送用の処理

スタック領域に割り付けられるデータとしては、関数の引数とローカル変数がある. これらは、

1. 実行時に割り付けられる.

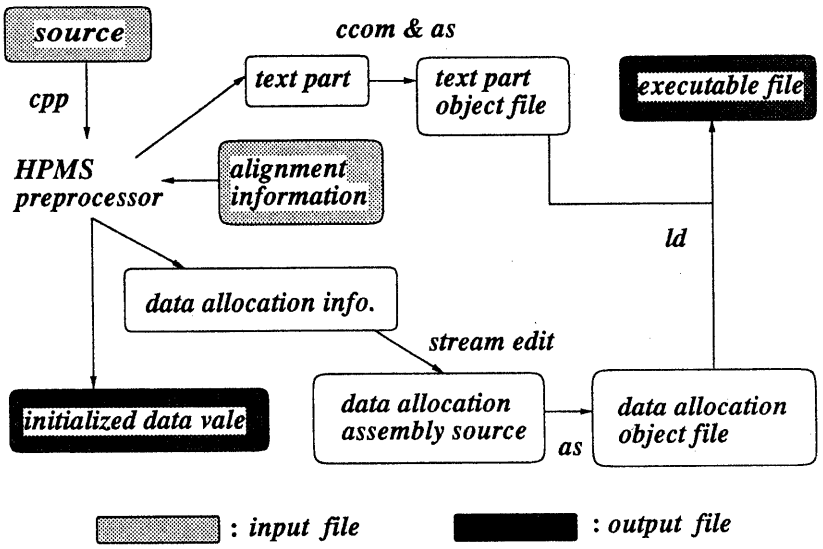

図 1: 本方式におけるコンパイルの手順

2. スタックの底は機種間でちがう可能性がある.

3. スタックフレームの形が機種によって違う.

ということがあり、リンカによる操作は不可能であるし、変数間へのバディングだけで変数を全機種で同一アドレスに割り付けるようにするのは困難である.

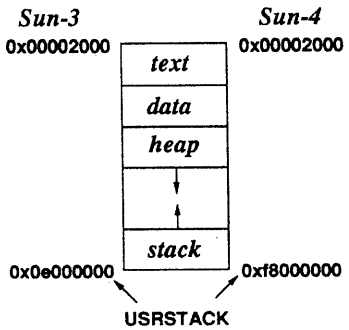

図 2: スタックの底のアドレスの違い

6.1 データ構造

前節のような問題に対処する方法としてここでは関数内のローカル変数全体を構造体として宣言してしまい、各変数はそのメンバとしてアクセスするようにする. この際、データ領域の時と同じようなダミー挿入によって、全機種で構造体内の各メンバのオフセットおよび構造体のサイズを揃える.

(以降、この構造体を“ローカル変数構造体”、この構造体と引数列を合わせたものを“フレーム”と呼ぶ.)

内部ブロックにおける変数の宣言は、他のローカル変数と衝突しない名前に変更して関数の先頭に移動し、ローカル変数構造体に含める. 初期化文は通常の文に直して残される.

さらに、このローカル変数構造体の先頭に

- スタック上の前の関数の構造体へのポインタ (通常のスタックフレーム中の退避フレームポインタに相当)
- スタック上の次の関数の構造体へのポインタ
- このローカル変数構造体のサイズ
- 引数の先頭へのポインタ
- 引数のサイズ
- 関数内位置 (ラベル) 情報 (通常のスタックフレーム中の退避プログラムカウンタに相当)

等を宣言しておく. 以降、初めの 2 つのフレーム間のポインタをまとめて“フレームリンク”と呼ぶ.

さらに、スタックの底の関数 (main) のフレームへのポインタとスタックのトップの関数のフレームへのポインタ (通常のフレームポインタに相当) をグローバル変数 stack_bottom、stack_top に設定しておく.

実行再開位置は、あらかじめソースプログラム内ラベルを付けることによって、全機種で認識可能にする．また、関数呼び出し中の移送を可能にするために、関数呼び出しの前にもラベルを付ける．

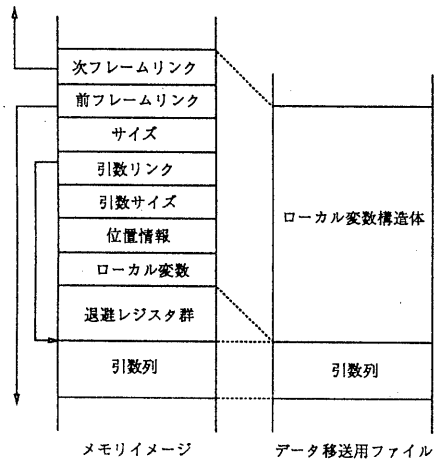

図 3: 各関数のフレーム

このようにして生成した各関数のローカル変数構造体を移送元でファイルに書き出し、移送先で読み込めばよいということになる．以下、関数呼び出し / 復帰時、移送時にそれぞれ必要な処理について述べる．

6.2 関数呼び出し / 復帰時の処理

関数呼び出し

関数呼び出しには、

- 通常呼び出し
- 移送時実行再開呼び出し

の 2 つのモードが存在する．この 2 つのモードはあるグローバル変数によって識別する．

関数呼び出し時には、次の処理を行なう．

呼び出し側

1. 関数内位置情報（ラベル番号）を退避する．
2. 引数サイズをグローバル変数にセットする．
3. 関数の呼び出し

呼び出された側

1. フレームリンク、引数リンク、引数サイズ、グローバル変数 stack_top 等の設定．
2. 移送時実行再開呼び出しの時は、1 の前後で
   - その関数に相当するフレームをファイルから読み込む．
   - 関数内位置情報の値にしたがって、それに相当する位置に goto でジャンプする．
3. 実行（実行中には、ラベル位置で関数内位置情報を退避する．）

関数からの復帰時（関数内の return を含む）

- フレームリンク、stack_top の再設定

を行なう．

6.3 引数の受渡しについて

ここでは、引数のサイズ及びその先頭のアドレスによって、移送時のファイルへの書き出し及びファイルからの読み込みを行なう．

最近の RISC を用いたシステムでは、最初のいくつかの引数はレジスタにのせて関数に引き渡され、関数側では仮引数として宣言された分だけをスタックメモリにストアしてアクセスをそのメモリに対して行なう．その場合、仮引数として宣言されていない引数に対してはメモリの内容は意味を持たない．

これが問題になるのは、可変個の引数を渡される関数の場合である．例えば Sun-4 の C コンパイラでは、最初の 6×8 byte 分の引数はレジスタにのせて関数に渡され、残りはスタックメモリを用いて渡される．仮引数として宣言された引数が、レジスタにのせられて渡されるものより少ない場合、いくつかの引数の値はメモリにストアされていないことになる．

これへの対策として、

1. 関数宣言の仮引数を付加して（Sun-4 なら 6×8 byte になるようにする）、メモリにすべての引数がストアされ、アクセスがそのメモリに対して行なわれるようにする．
2. 可変個引数を使用する関数では、必ず <varargs.h> で定義されたマクロを使用するようにさせる．（効果は 1と同じことである．）

の 2 つの方法が考えられるが、今回の試作では実装の行ない易さから 2の方法を用いた．

6.4 移送時の処理

移送時には、stack_bottom 側からフレームリンクにそって順番に各関数のローカル変数構造体及び引数をファイルに書き出す．

移送先の再開時には、移送時に呼び出し中であった各関数は上で述べたように（gotoの繰り返しを用いて）stack_bottom側から順番に実際に呼び出される（移送時実行再開モード）ので、各関数の先頭で各リンクの設定及びファイルから構造体と引数の読み込みを行なう．

移送時にスタックのトップにあった関数が呼び出され、実行再開位置に達した時点で、移送時実行再開モードは通常実行モードに移行する．

HPMSプリプロセッサでは、オリジナルのソースプログラムを以上に述べたような変数宣言及び各処理を行なうようなソースプログラムに変形し、テキスト部として出力する．

この時、式の中に2つの関数呼び出しがある場合や、関数の引数に関数が用いられている場合は、一時変数などを用いて評価順序にしたがって分解する．その際、各呼び出しの前に呼び出し時処理コードを挿入する．

7 通常ファイルに対する処理

実行時にオープンされているファイルについては、後からファイル名を知ることができない．

そこで通常ファイルに対する処理として、

1. オープン時に“完全パス名”を、移送時に“変位”をデータ領域に保存して移送する．
2. 移送先では、そのファイルを再オープンし、ファイル記述子及び変位を再設定する．

ことを行なうことによって移送時にオープンしている通常ファイルに対処する．（これを行なうには、NFSなどのようにどのマシンからも同じパス名でファイルを参照できる環境が必要である．）

8 その他の処理

その他の処理として、setjmp/longjmpへの対応がある．これらは、setjmpでレジスタの内容をメモリに退避し、longjmpで退避したレジスタを回復するので、移送ポイントまたいで利用することはできない．移送ポイントをまたがない場合でも、longjmpによって関数外部に飛び出す可能性があるので、フレームリンクを設定し直さなければならない．

そこで移送ポイントをまたがない場合は利用可能にするために、setjmp時にstack_topの値も退避しておき、longjmp時に退避したstack_topの値を見て、フレームリンクの回復などの必要な処理を行なうようにする．

9 異機種間プロセス移送システムの試作

上記の方式に基づき、Sun-3(Motorola MC68020)マシンとSun-4(Sun SPARC)マシンの間で実際に異機種間プロセス移送機構を試作した．

（ただし、次節に述べるローカル変数へのポインタの処理は、この試作では行なっていない．）

その際、

- ソースファイルを加工するプリプロセッサ
- 移送を行なうのに必要となるライブラリ関数
- 移送時にプロセスを再開するサーバ

等を作成した．各プロセス間の通信にはソケットインタフェースを用い、ファイルの受渡しについては、NFSが稼働していることを仮定した．

これを用いて、実際に数値計算等を行なうプログラムをいくつか実行してみたところ、いずれも正しい結果を得ることができた．

また、移送時のオーバヘッドに関しては、各コンピュータの負荷やネットワークのトラフィックにも左右されるが、データ領域、スタック領域がそれぞれ約50KBであるプロセスついて、Sun3/60とSPARC station 2との間で移送を行なった時の平均オーバヘッドは、表2のようであった．

表 2: 移送時の平均オーバヘッド

		Sun-4 ->Sun-3	Sun-3 ->Sun-4
移送用ファイルへの書き出し処理	データ領域	320 ms	320 ms
	スタック領域	450 ms	460 ms
プロセスの生成		120 ms	65 ms
移送用ファイルからの読み込み処理	データ領域	140 ms	90 ms
	スタック領域	100 ms	55 ms
その他（サーバ間の通信など）		270 ms	260 ms
全体		1400 ms	1250 ms

Sun-4 : SPARC Station 2
Sun-3 : Sun-3/60

・データ、スタック領域がそれぞれ約50KB
・スタック領域の書き出し、読み出し処理は、全フレームの合計

前述したように、移送されるプロセスとして今は長時間走行するものを想定しているので、この程度のオーバヘッドであれば十分使用に耐えうると思われる．

10 ローカル変数へのポインタの移送用処理に関する考察

ここまで述べた方法でローカル変数へのポインタ変数以外の変数は移送可能だが、スタック領域は移送先で積み直され、移送前後でローカル変数は異なるアドレスに割り付けられるため、そのアドレスを値として持つポインタをそのまま移送しても意味をなさない．この場合、

ローカル変数へのポインタは、ある地点からのオフセットといった相対的な値を使って、全機種で変換可能な共通表現に変換して移送し、移送先で絶対アドレスに戻してやる必要がある．

この場合に必要な情報は、

- ポインタ変数の位置情報．
  これも全機種で共通の表現にする必要がある．
- テキスト、データ＋ヒープ、スタックの各領域の範囲

である．各ポインタ変数の値によって、どの領域の変数を指すものであるかがわかる．

各領域の範囲は、

テキスト　グローバル変数 etext のアドレスまで．
データ＋ヒープ　etext から sbrk(0) の戻り値まで．
スタック　グローバル変数 stack_top/ stack_bottom 及びその構造体サイズ等から識別できる．

という方法で知ることができる．

例えば、前述の各関数のローカル変数構造体内でのオフセットが考えられる．この場合、各関数に対して構造体が１つ存在するため、どのローカル変数構造体内でのオフセットかを示す情報も必要である．これは機種間で共通な表現をしなければならないので、アドレスは使えない．またその情報は、何らかの手段で構造体の先頭アドレスに変換できるようにしなければならない．

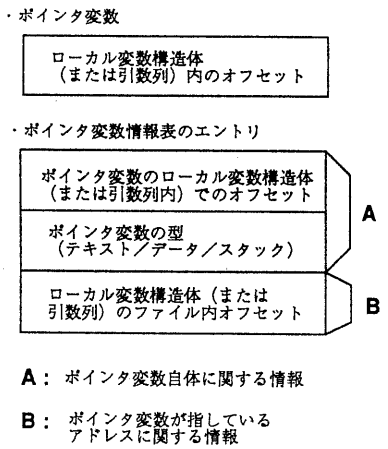

図 4: ポインタ変数情報表のエントリ

これには、図4のようなポインタ変数の情報テーブルを付加する方法が考えられる．

移送時に各領域をファイルに書き出す時にポインタ変数の値を各領域の値と比較して、スタック領域を指している場合は、

1. ポインタ変数の値を、そのポインタが指している変数が存在するローカル変数構造体（または引数列）内でのオフセットに直す．
2. ポインタ情報表にスタック領域を指していることをマークする．
3. ポインタが指している変数が含まれるローカル変数構造体のファイル内のオフセットをポインタ情報表に記録する．

という変換処理を行なってから、そのポインタ変数が存在するローカル変数構造体をファイルに書き出す．これによって、移送用ファイル内に全機種共通のスタックを生成することになる．

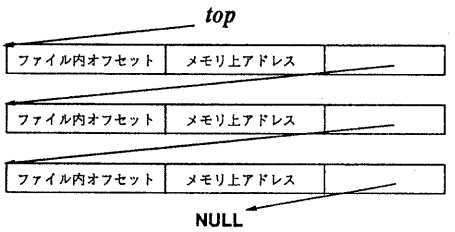

図 5: アドレス変換表

移送先では、ファイルからのローカル変数の読み込み時に、各ローカル変数構造体の

（ファイル内オフセット、メモリ上アドレス）

のアドレス変換表の組を保持する（図5参照）．

すべての読み込みが終了した時点で、ポインタ情報表の型フィールドがスタックを示しているものに対して、

1. ポインタ変数内のファイル内オフセットフィールドと比較してそのポインタ変数が指すローカル変数構造体（または引数列）を識別する．
2. そのローカル変数構造体のメモリ上アドレスをポインタ変数の内容に加える．

という処理を行なう．これによって、共通表現から実際の各マシン表現へ逆変換される．この後、通常実行モードになる前に、アドレス変換表は解放される．

ポインタ情報表は、

- データ領域のポインタ変数の情報はデータ領域に定義して配置する．
- スタック領域のポインタ変数の情報は各関数のローカル変数構造体に含めて宣言する．引数内のポインタ変数に関しては、呼び出し時に呼び出し側から与える．

これらの変数宣言や処理を行なうコードをプリプロセッサによって加えてやる必要がある．また、

- ヒープ領域のポインタ変数情報のエントリはメモリ割り当て関数呼び出し時に作成してやる必要がある.

このような処理を行なえば、ローカル変数へのポインタを移送することは可能であると思われるが、ポインタの変換処理を行なう時に指している領域を知るために比較を行なうので、ポインタと他の変数との共用体（union）等は禁止されることになる.

また、構造体内のポインタ変数情報の表現、ヒープ領域内のポインタ変数情報の与え方なども問題となる.

これらは検討中の事項であり、現在の試作ではこの処理は行なっていない. そのため、移送ポイントをまたいでローカル変数へのポインタを使用することを制限している. 今後、さらに検討を加え、実装・評価していく予定である.

11 他の課題

本稿で述べた方式で異機種間の移送が可能にすることができたが、完全なものではない. その他の課題としては、

- 関数へのポインタの対処

  これは、シグナルハンドラの設定等にも関連する. これには、ローカル変数へのポインタの移送用処理と同様に、シンボルテーブル等を用いたアドレス変換が必要である.
- Endian の違いへの対処

  これに対処するには、サイズが2byte以上の基本データ型を持つ変数（構造体内のものを含む）に関する位置及びサイズの情報テーブルを付加する必要がある.

といったものがある. また、

- シグナル、パイプ、ソケット等のプロセス間通信

には、UNIX のセマンティクスのままでは完全に対処するのは難しいが、分散 OS を用いて“位置透過性”を達成することで対応可能な問題であると思われる.

12 まとめ

本稿では、プリプロセッサ処理による異機種間プロセス移送メカニズムの実現可能性を示した. 制限事項はあるものの、長時間数値計算を実行するようなプロセスに対しては、十分使用可能であり、負荷の分散によるスループットの向上が見込まれる.

我々が作成したプリプロセッサ自体は、基本的にソースレベルの書き換えを行なうだけなので、機種の相違がある程度の範囲内なら新たなマシンの追加は容易に行なえる.

現在は、更に一般的なプログラムに対応するために、課題の項で述べた関数及びローカル変数へのポインタに関するアドレス変換処理について考慮中である.

参考文献

[1] 森山茂男、多田好克：“カーネルの変更を伴わないプロセス移送の実現法”, The 18th jus UNIX Symposium Proceedings, pp.151-159 (November, 1991).

[2] G.Attardi, A.Baldi, U.Boni,F.Carignani, G.Cozzi, A.Pelligrini, E.Durocher, I.Filotti, and W.Qing：“Techniques for dynamic software migration”, Proceedings of the 5th Annual Esprit Conference, pp.475-491 (November, 1988).

[3] C.Sub：“Native code process-originated migration in a heterogeneous environment”, Proceedings of the ACM 18th Annual Computer Science Conference, pp.266-270 (February, 1990).

[4] Marvin M.Theimer, Barry Hayes：“Heterogeneous Process Migration by Recompilation”, Proceedings of The 11th International Conference on Distributed Computing Systems, pp.18-25 (May, 1991).

[5] 長澤、大胡、相田、齊藤：“異機種間プロセスマイグレーションの一手法,” 情報処理学会第 45 回全国大会予稿集 (October, 1992).

[6] B.W.Kernighan, D.M.Ritchie：“The C programming language, Second Edition,” Prentice-Hall, 1988（邦訳 プログラミング言語 C 第2版, 石田晴久訳, 共立出版, 1989）.

[7] A.Aho, R.Sethi, J.Ullman：“Compilers – Principles Techniques and Tools,” Addison-Wesley, 1986（邦訳 コンパイラ – 原理・技法・ツール – I・II, 原田賢一訳, サイエンス社, 1990）.

[8] 多田好克：“機種に依存しない利用者 threads ライブラリ,” 情報処理学会 PRG 研究会報告, 92-PRG-8-22, pp.171-178 (August, 1992).