オペレーティング・システム 51－6
（1991. 7. 18）

利用者レベルで実現したプロセス移送ライブラリ

森山 茂男，多田 好克
電気通信大学 電子情報学科

近年，ネットワーク上に数十台の *UNIX* ワークステーションが接続されているという状況は珍しくない．しかし，現在の利用形態は，依然，計算機単位を中心にしたものであり，特に，ネットワーク上の *CPU* 資源を十分に活用しているとはいえない．

筆者らは，*UNIX* カーネルを変更することなくプロセスを移送する仕組みを構築し，プロセス移送の各種実験等に利用している．また，本システムの利用により，数値計算等の実行に時間のかかるプログラムをワークステーションシステム上で実行している人々にとっては，処理時間の短縮が見込まれる．

本稿では，その実現法について議論し，負荷分散の可能性を示す．

Process Migration by a User Level Library

Shigeo Moriyama, Yoshikatsu Tada
The University of Electro-Communications,
1–5–1 Choufugaoka Choufu-shi, Tokyo, 182, JAPAN

In recent years, it becomes quite common that a number of *UNIX* workstations are connected via a network. However, the present way of usage still concents on a single computer, and the *CPU* resources are not utilized efficently.

We made a process migration library on *UNIX* without kernel modification. Using this library, we can implement a testing environment of process migration easily. Those who execute *CPU* bound programs (e.g. mathematical computation) on the workstation system, will have an opportunity to gain throughput by using our library.

In this paper, we discuss the implementation of this method and point out the possibility of load balancing with this method.

1 はじめに

近年，ネットワーク上に数十台の *UNIX* ワークステーションが接続されているという状況は珍しくない．このような状況の下，ファイルの共有等の機構が生まれてきたことは，ネットワーク上の資源の有効な利用という点からも自然なことである．しかし，現在の利用形態は，依然，計算機単位を中心にしたものであり，ネットワーク上の *CPU* 資源を十分に活用しているとはいえない．例えば，図1は，筆者らが使用しているネットワーク上の計算機の負荷を500時間にわたって測定したものである．

計算機 A,B の負荷は計算機 C,D と比較して，長時間にわたって高いものとなっている様子がうかがえる．

また、時刻によって，計算機 A,B のプロセスの一部を計算機 C,D に移送できれば，全体のスループットは向上すると考えられる．

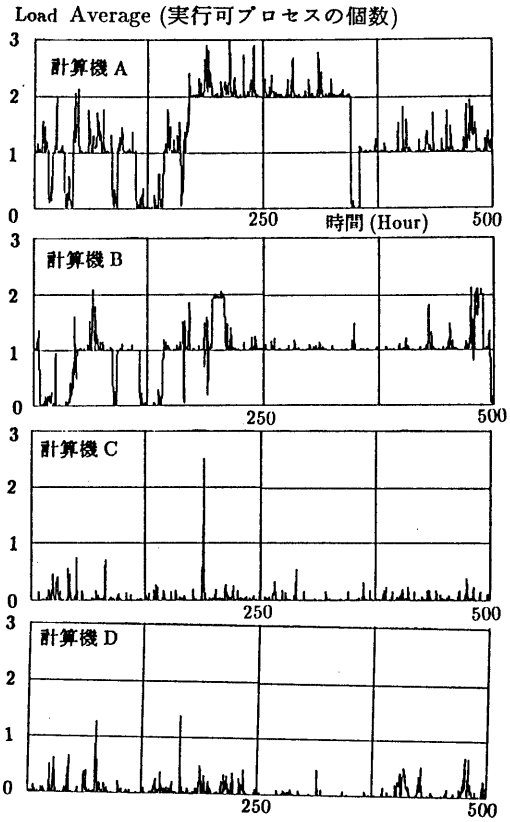

図 1: 計算機の負荷の様子

分散オペレーティングシステム LOCUS[1], V-System[2] 等では，この機能をオペレーティングシステムで実現している．しかし，現在のネットワークは，*UNIX* を中心にしたワークステーションで構成されており，オペレーティングシステムを変更することは，一般の利用者に受け入れられにくい．また，プロセス移送に関する各種の実験を行なおうとすると，オペレーティングシステムの再構成を行なう必要があり，容易ではない．

今回，筆者らは，*UNIX* カーネルを変更することなく，プロセスを移送する仕組みを実現した．本実現法は，

- *UNIX* ネットワーク上で負荷分散が計れる;
- *UNIX* 上で，プロセス移送の各種の実験を容易に効率よく行なえる；
- 既存の *UNIX* 上の開発環境を利用できる;
- すべての仕組みを言語 C[4] で記述した;

という特長を持つ．また，この仕組みは SONY NEWS(4.3BSD:NWS-800 シリーズ) 上で実際に稼働しており，各種実験にも利用されている．

本稿では，このプロセス移送機構の具体的な実現法を示すとともに，この仕組みを利用した負荷分散の可能性を議論する．

今回の実現法では，移送先となり得るすべての計算機上でサーバプロセスを動かす．利用者プロセスは，本システムのライブラリ関数を使用することで，サーバプロセスと通信し，移送されるようになる．プロセス移送の詳細は以下の手順で示される (図2参照)．

1. プロセスは実行開始時に，その計算機上のサーバプロセスと (ソケットで) 通信して，登録を行なう．
2. サーバプロセスが，このプロセスの転送を決定し，シグナルで通知する．
3. シグナルを受け取ったプロセスは，そのシグナル処理関数中でプロセスのコンテクスト (内部状態) をファイルに出力し，その計算機上のサーバプロセスに終了の返事を

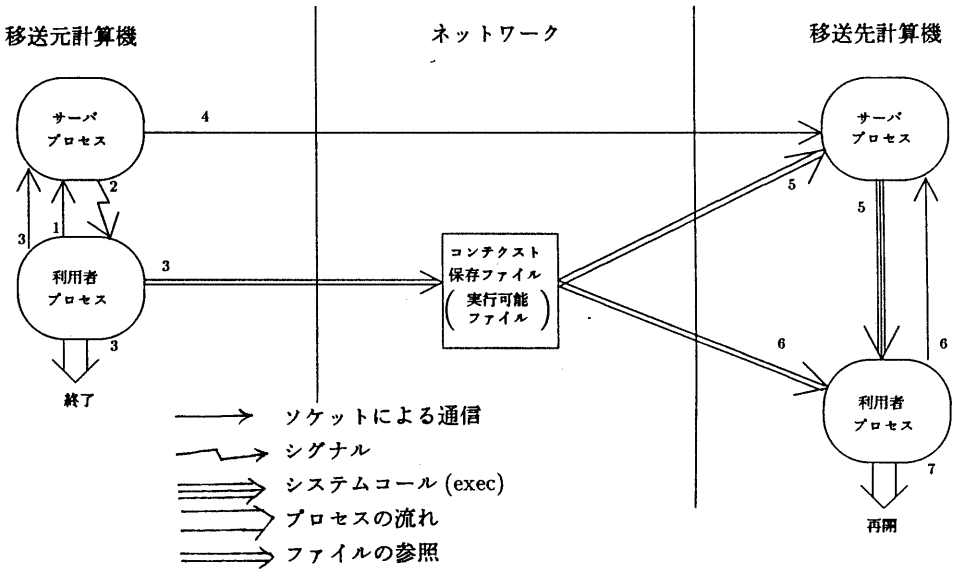

図 2: プロセス移送の手順

(ソケットで) 返す．その後，このプロセスは実行を終了する．ここで出力したコンテクスト保存ファイルは，ネットワーク上でのファイル共有機構を通して参照される．また，このファイルは *UNIX* 上の実行可能ファイル (a.out 形式) となっている (詳細は，3.2節で議論する)．

4. その返事を受け取ったサーバプロセスは，移送先の計算機上のサーバプロセスと (ソケットで) 通信し，移送対象プロセスの実行再開を依頼する．

5. 移送先のサーバプロセスは，先のコンテクスト保存ファイルを実行する．

6. このプロセスは，実行が始まるときに，コンテクストを回復し，1と同じように，サーバプロセスに登録を行なう．その後，3で中断されたシグナル処理関数内へ復帰する．これで実行は再開する．

7. 2の状態となり，以後，これを繰り返す．

以下，本稿では，このプロセス移送の実現法について議論する．まず，第2章では，本プロセス移送ライブラリのインターフェースについて議論する．その後，第3章ではプロセス移送の実現法の詳細を論じる．そして，第4章では実行時のオーバーヘッドについて考察する．最後の第5章では，今後の展望を述べる．

なお，一般に，ネットワーク上には異なったアーキテクチャの計算機が混在しているが，今回の実現法では異アーキテクチャ計算機間でのプロセス移送は行なわない．今回は，実現の容易さから，同一アーキテクチャで，かつ，*UNIX* カーネルのコードが同一な計算機を想定している．

また，本稿では言語 C と *UNIX* とに関する知識を仮定している．

2 ライブラリ関数のインターフェース

プロセス移送を実現するために，*pms_main*, *pms_open*, *pms_close*, *pms_fopen*, *pms_fclose* の各種関数を用意した．

これらの関数のインターフェースは，それぞれの関数名から *pms_* を除いた同名の言語 C の関数・システムコールと同じである．

利用者が，言語 C での *main* の代わりに，*pms_main* を用いることによって，そのプロセスは移送されるようになる．また，用意されたファイルのオープン・クローズ用関数を用いることで，ファイルへの一貫した入出力が行える．ファイルのオープン・クローズ関数を用意した理由は 3.2節で述べる．

今回は，移送対象となるプロセスとして，比較的長時間実行されるものを想定している．このようなプロセスが，利用者と対話しながら処理を進めるとは考えにくい．このような理由から，入出力を通常ファイル (regular file) に限定した．入出力に使用するファイルは，移送先からも同一の絶対パスによって参照できるものとする．また，プロセス間の通信に関しては対処していない．

3 プロセス移送の実現法

本章ではプロセス移送の実現法について論じる．まず，3.1節で計算機間で移送されるプロセスのコンテクストについて考える．その後，3.2節で，それらコンテクストの取得方法について論じる．3.3節では実行の再開方法について議論する．最後の 3.4節ではプロセス移送の決定を下すサーバプロセスについて説明する．

3.1 プロセスのコンテクスト

まず，計算機間で移送されるプロセスにとってのコンテクストとして何が必要であるか考える．

UNIX におけるプロセスコンテクストは，

(A) プロセス仮想アドレス空間を占めるテキスト領域，データ領域，スタック領域

(B) ハードウェアレジスタの値

(C) カーネル中のコンテクスト

の 3 つに分類される [5]．ここで，*UNIX* カーネル中のコンテクストとは，プロセスの実行によって，プロセステーブル，*u* エリア，カーネルスタック上に存在するコンテクストのことである．

次に，これら *UNIX* におけるプロセスコンテクストの内から本実現法で必要とされるコンテクストを考える．

(A) について
データ領域，スタック領域は，変数の値の変更，関数呼出し等によって変化するので必要である．また，テキスト領域についても，実行中にもとの実行可能ファイルが削除されることがあり得るので必要となる．

(B) について
スタック・ポインタやプログラム・カウンタ，あるいはレジスタ変数等に使用されており，プロセス実行中に変化するので，すべてのレジスタの値が必要である．

(C) について
プロセスの実行が，異なる計算機に移ることから，本来はすべての情報が必要であるが，本システムが実験用であることを考慮し，システムコールで取得することのできない一部の情報は無視する．以下，各情報について，個々に説明する;

- プロセステーブル上
  ここには，プロセスの状態，利用者識別子，プロセス識別子，シグナルマスク等の情報，スケジューリングの為の情報等がある．これらのうち利用者識別子，シグナルマスクの状態だけを扱い，残りの情報は無視する．
- *u* エリア上
  ここには，現ディレクトリ，シグナル処理関数，オープンしたファイルに対するカーネル内ファイルテーブルを指す利用者ファイル記述子表，システムコールの結果，制御端末などの情報がある．このうち，現ディレクトリ，シグナル処理関数，オープンしたファイルの情報のみを扱う．
- カーネルスタック上
  カーネルスタックは，カーネルモード実行時に使用されるスタックである．

本実現法では，利用者モードで処理を行なうから，カーネルスタック上にコンテクストは存在しない．

3.2 コンテクストの取得

本節では，前節で述べた各コンテクストをどのように取得するのか議論する．

(A) について
これらは，利用者がアクセス可能なアドレス空間に存在するので，直接読み出すことができる．

(B) について
このコンテクストは，更に 2 つに分類される．言語 C の標準ライブラリ関数_setjmp, _longjmp [1]によって保存・回復されるもの (NWS-800 シリーズでは，*CPU* に 68020 を使用しており，スタックポインタ，フレームポインタ，プログラムカウンタ，レジスタ変数用レジスタがこれにあたる (図 3)) と言語 C の一時レジスタである．前者は_setjmp を使用して，データ領域に保存した．後者の取得には，シグナルを利用することができる．シグナルを受け取った *UNIX* プロセスは，その一時レジスタの値等を利用者スタック上に積み上げた後，シグナル処理関数を実行する．したがって，このコンテクストは，シグナル処理関数実行中にはスタック領域上に存在する．

(C) について
このコンテクストは，オープンしたファイルの情報を除けば，システムコールで取得することができる．オープンしたファイルの情報は，オープンしているファイル名を知る手段がないので，open 用の関数を別途用意した．その関数内では，ファイル名を保存した後，open を呼び出してファイルをオープンする．本実現法では，これらのコンテクストをデータ領域に保存することにした．

[1] *UNIX* マニュアルの setjmp(3) を参照

jmp_buf 構造体

添字	内容 (レジスタ名)
0	D4
1	D5
2	D6
3	D7
4	A2
5	A3
6	A4
7	A5
8	フレームポインタ
9	スタックポインタ
10	プログラムカウンタ
11	シグナルマスク等 (_setjmp,_longjmp では，未使用)
12	
13	
14	
15	

図 3: setjmp,longjmp がレジスタの保存・回復に使用する構造体 jmp_buf の構造

結局，移送されるプロセスのコンテクストには，シグナル処理関数実行中にそのテキスト領域，データ領域，スタック領域を保存すれば十分である．本実現法では，これらを実行可能ファイル形式+スタック領域という形でファイルにして保存した (図 4)．

3.3 実行の再開

実行の再開とは，前述のコンテクストを別の計算機上で回復することを意味する．

コンテクスト保存ファイルが，実行可能ファイル形式+スタック領域という形であるから，このファイルを実行することによって，テキスト領域とデータ領域を回復することができる．一方，スタック領域については，利用者プログラムの実行再開に先立って，ファイルからプロセスのスタック領域に回復することで処理する．しかし，単純にスタック領域を回復すると，実行中のスタックに上書きしてしまう．これは，スタックポインタを上書きされない領域にセッ

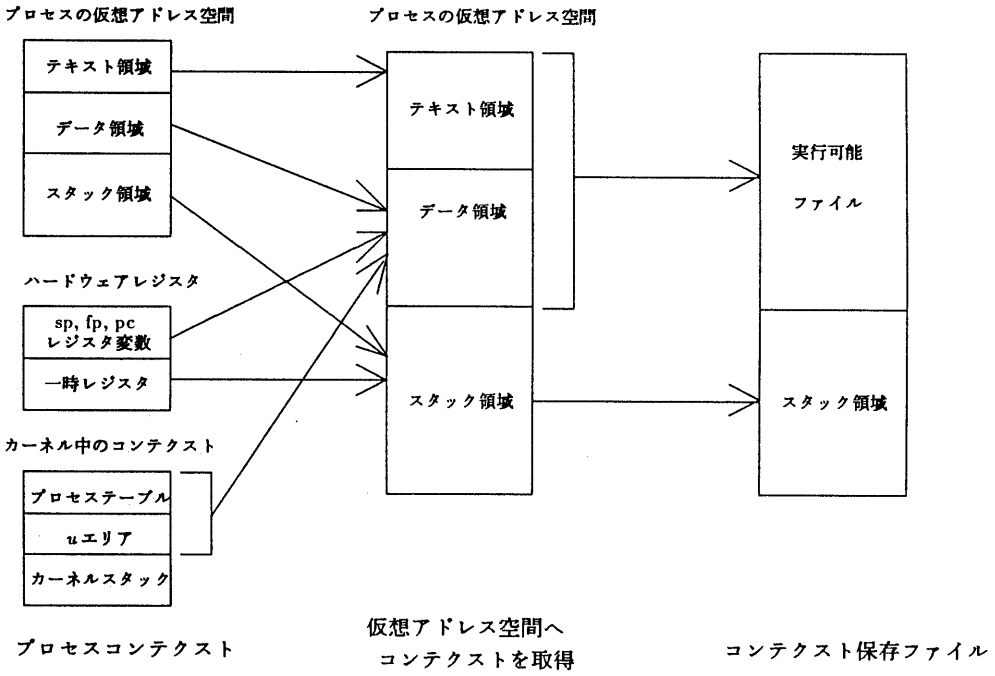

図 4: コンテクスト保存の様子

トした後，スタック領域回復を行なう関数を呼び出すことで回避できる．この関数中で，

- オープンしていたファイルの再オープン (ファイル内変位の設定等も行なう)
- シグナル処理関数の再設定
- 現ディレクトリの設定

を行なった後，シグナル処理関数の続きを実行するために大域ジャンプ (_longjmp) する．残った仕事はシグナル処理関数から復帰することだけである．

シグナル処理関数から復帰することで，一時レジスタの値，シグナルマスクの値が回復される．その後，プロセスは移送されたことに気付くことなく，実行を再開することとなる．

3.4 サーバプロセス

ここまでで，移送するプロセスのコンテクストを保存する方法，保存されたコンテクストを回復する方法を構築した．残るは，プロセス移送を決定する方法，コンテクスト保存ファイルを実行する方法を考えればよい．本実現法では，これらの処理を，各計算機上のサーバプロセスに行なわせる．

サーバプロセスは，

- その計算機上で移送対象となっているプロセスの管理
- 移送のスケジューリング (移送プロセスの決定)
- 他の計算機から移送されてきたプロセスの実行再開

という処理を行なう．

ところで，プロセス移送のスケジューリング・アルゴリズムは，負荷分散を考える上で，重要なものである．本実現法においては，各種のスケジューリング・アルゴリズムの実験をサーバプログラムの変更だけで行なうことができる．

4 実行時のオーバーヘッドについて

次に，プロセス移送にともなうオーバーヘッドについて考える．本システムにおけるオーバーヘッドは，

1. プロセス・コンテクストの取得
2. 移送に関わるサーバプロセス間，サーバプロセス・利用者プロセス間の通信
3. プロセス・コンテクストの回復

にかかる時間がほとんどである．

NWS-830(*CPU*:68020, 16.67MHz) で測定した結果，1の処理に113ミリ秒，3の処理にプロセス生成 (fork+exec) 時間+66ミリ秒かかった．2はネットワークの状態にもよるので時間としては評価しなかったが，やり取りされるデータ量は48バイトであった．

5 おわりに

本稿では，*UNIX* カーネルを変更することなしに構築されたプロセス移送機構の実現法を示し，負荷分散の可能性を示した．この仕組みを利用すれば，プロセス移送の各種実験が *UNIX* 上で効率よく行なえる．また，大規模数値計算をワークステーションシステム上で実行している人々にとっては，処理速度の向上が見込まれる．さらに，サーバープロセスにシグナル等で指示を与え，プロセスを別の計算機へ移送してから，計算機の電源を落とすことも可能である．

現在，移送のスケジューリングのアルゴリズムとして単純なものを採用しているが，我々の研究室ではさらに効率のよいアルゴリズムを研究中である．

また，今回，プロセスへの入出力の手段としては通常ファイルのみを利用しているが，より一般的なファイルへの拡張も現在考慮中である．

参考文献

[1] Popek,G.J. and B.J.Walker: "The LOCUS Distributed System Architecture," The MIT Press (1985).

[2] Cheriton,D.R.: "The V kernel: A software base for distributed systems," *IEEE Software*, 1, 2, pp.19–42(Apr. 1984).

[3] 清水謙多郎:"分散オペレーティング・システム", 情報処理学会 OS 研究会報告, 89–OS–45 (1989).

[4] 石田晴久訳:"プログラミング言語 C", 共立出版,1981.

[5] 坂本文, 多田好克, 村井純訳:"UNIX カーネルの設計", 共立出版,*bit* 1990 年 10 月号別冊.

[6] 多田好克, 寺田実:"並行プロセス実験キット", 情報処理学会 OS 研究会報告, 86–OS–33 (1986).

[7] 多田好克, 寺田実:"移植性・拡張性に優れたCのコルーチンライブラリー実現法", 電子情報通信学会論文誌 D–I, Vol.J73–D–I, No.12, pp.961–970 (Dec. 1990).

[8] Leffer,S.J., M.K.McKusick, M.J.Karels, and J.S.Quarterman:"*The Design and Implementation of the 4.3BSD UNIX Operating System*," Addison–Wesley, Reading, MA, 1989.