マルチメディア通信と分散処理 29－7
（1986. 5. 15）

UNIXにおけるプロセス制御機能のネットワーク化

谷口 秀夫
NTT 電気通信研究所

本稿では、LANが持つ高速性や同報機能を生かしたサービスを容易に実現のための、ネットワーク化されたプロセス制御機能をUNIX上に実現する方式について報告する。

主な内容を以下に示す。

(1) LANで結ばれた処理装置間を、ファイルアクセス系の環境を保存してプロセスが移動する方式
(2) リモートにあるロードモジュールファイルを用いて、ローカルなプロセス上に起動する方式
(3) 分散したプロセス間の通信方式（パイプ、シグナル）

Networking the Process Control Mechanism on UNIX

Hideo TANIGUCHI
NTT Electrical Communications Laboratories

A network process control mechanism on UNIX kernel is reported, which enables easy construction of services that utilize LAN characteristics such as high speed transfer and broadcasting.

Main contents are as follows,

(1) mechanism to transfer processes among processing units connected with a LAN,
(2) mechanism to execute local processes which use remote program modules,
(3) communication methods inter distributed processes : pipe and signal.

1.はじめに

LANで結ばれた処理装置(以降、ノードと呼ぶ)を有機的に結合し、電子メール[1]や電子会議[1][2]などのサービスシステムを容易に構築するためには、個々のノードのOSが有するリソース管理機能や通信機能をネットワークワイドに拡張することにより、スタンドアロン環境とネットワーク環境との差異を極力OSレベルで吸収することが重要である。これらOS機能のネットワーク化に関し、ファイル管理機能については、文献[3]に報告した。

本稿では、プロセス管理機能に着目し、①他ノードへのプロセス生成、②他ノードにあるロードモジュール(LM)ファイルを利用したプログラム起動、③リモートプロセス間の通信、などのネットワーク環境下でのプロセス制御機能(以降、分散プロセス制御機能と呼ぶ)をUNIX*に実現する際の制御方式について述べる。

UNIXをベースとして分散プロセス制御機能の一部(他ノードへのプロセス生成かつプログラム起動)を実現した例としてCOCANET[4]があるが、本稿で述べる分散プロセス制御機能の特徴は、

(1) ノード間のプロセス移動機能
(2) 他ノードにあるロードモジュール(LM)ファイルを利用したプログラム起動機能
(3) 分散したプロセス間の通信機能

を実現した点である。実現した分散プロセス制御機能の概要を図1に示す。

2.分散プロセス制御機能のねらい

2.1必要性

分散プロセス制御機能によるメリットとして次のものがある。

(1) 処理の効率化

処理の分散により、CPU資源の有効利用が可能になる。例えば、他ノードにあるファイルへのアクセスが多い処理は、リモートファイルアクセス機能を利用してもできるが、本機能を利用して、ファイルが存在するノード上で処理プロセスを走行させて処理結果はリモートプロセス間通信で送受信すると、①ファイルアクセス速度が速くなり、②LAN上を飛び交うパケット数が減少して、効率化が図れる。

(2) ディスク(DK)等の記憶装置の有効利用

ノード間で共通なLMファイルをひとつのノードに置いて他ノードから共用することにより、DKの有効利用を図る。これらのLMファイルをフレキシブルディスク(FD)に格納しておいて利用時にマウントすることも考えられるが、ユーザインタフェース上好ましくない。一方、アクセス速度の面からもひとつのノードに置いて皆で共用した方がよい(多くの場合、リモートDK上のファイルアクセス速度はローカルFD上のファイルアクセス速度の2倍以上である)。これをさらに押し進めれば、DKなしの

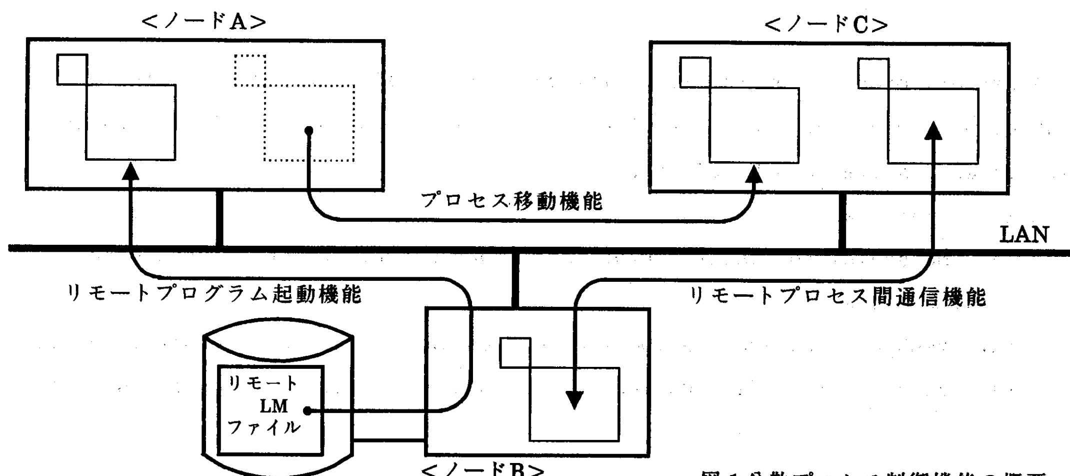


図1分散プロセス制御機能の概要

*: UNIXはAT&Tのベル研究所が開発したOS

ノードも実現できる。

2.2要求条件

分散プロセス制御機能は、上記のメリットを生かすサービスプログラムの作成を容易にし、かつローカル処理用ソフトウェアを流用できるようにするため、次の条件を満足することが望まれる。

(1) ローカル処理用アプリケーション(AP)インタフェースとリモート処理用APインタフェースが同形式であること。

(2) ローカル処理用の提供機能ができるだけリモート処理でも利用できること。

(3) リモート処理特有部分は局所化していること。

3.基本方式

3.1プロセス制御機能のネットワーク化

プロセスの制御機能をネットワーク環境下に拡張する際、ノードとプロセスやプログラムの対応関係として、以下の組み合わせがある。

(1)プロセスの生成
- (A)同一ノード上にプロセスを生成
- (B)他ノード上にプロセスを生成

(2)プログラムの起動
- (A)ローカルのLMファイルをローカルなプロセス上に起動
- (B)ローカルのLMファイルをリモートのプロセス上に起動
- (C)リモートのLMファイルをローカルなプロセス上に起動
- (D)リモートのLMファイルをリモートのプロセス上に起動し、以下の場合
  - (a) LMファイルの存在ノードとプロセスの走行ノードが同じ
  - (b) LMファイルの存在ノードとプロセスの走行ノードが異なる

(1-A)や(2-A)は、従来のforkやexecの機能である。この他にCOCANETのように、プロセス生成とプログラム起動の機能を統合してひとつのシステムコールで提供することも考えられる。

分散プロセス間の通信機能については、既存のローカルな機能(具体的には、パイプ、シグナルやwait/exit)をそのままネットワーク環境に拡張する。

3.2ノードの指定方式

分散プロセス制御機能の実現においては、ローカルな環境では不要な「プロセス生成ノード」や「プログラム存在ノード」および「プロセス走行ノード」の指定が必要になる。

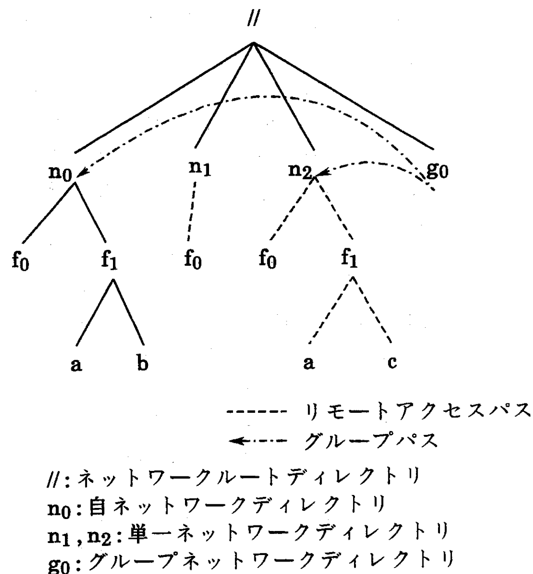


図2グローバルトリーの例(ノードn0からのview)

この方法として、分散したファイルの統一管理方式であるグローバルトリー[3]を利用した方式が考えられる。グローバルトリーは、例えば図2に示す構成になっており、自/単一/グループの各ネットワークディレクトリがノードに対応している。グローバルトリーは分散したファイルを統一管理しているため、既存システムコールのインタフェースのままで「プログラム存在ノード」の指定を可能にしている。これを用いた「プロセス生成ノード」と「プロセス走行ノード」の指定方式案を以下に示す。

[案1] システムコールのパラメータにより、ノードに対応するネットワークディレクトリで指定する方式

[案2] プログラム存在ノードと同じにする方式

[案3] カレントディレクトリ(またはカレントルートディレクトリ)があるネットワークディレクトリに対応するノードと同じにする方式

しかし、上記の方式案には以下のデメリットがある。

(1) 案1は、APインタフェースがローカルとリモートで異なるため、APプログラム作成上好ましくない。また、3.1節で述べた各リ

モート機能に対応するシステムコールを新たに実現する必要がある。

(2) 案2は、3.1節で述べた(1-B)と(2-D-a)の機能を統合したもの(例：COCANET)であるため、これらの機能を独立して利用できない。また、(2-B)や(2-C)および(2-D-b)の機能が実現できない。

(3) 案3のデメリットを以下に示す。

(A)システムコール chdir (または chroot) と fork や exec を関連づけるため、使い方が制限される。例えば、ノードn_0上でカレントディレクトリ"//n_1"のプロセスが、同じカレントディレクトリを持つローカルな子プロセスを生成するには、カレントディレクトリを一度"//n_0"に変更して fork した後、再び両プロセスがカレントディレクトリを"//n_1"に変更しなくてはならない。

(B)リモートファイルアクセス機能との整合性が悪く、制御が複雑になる。つまり、リモートファイルアクセスを要求するシステムコールは、アクセスするファイルが存在するノードがシステムコール転送[3]先ノードになるのに対し、例えば exec ではアクセスするファイルが存在するノードに関係なくシステムコール転送先ノードが決定される。グローバルトリーが図2の時の例を図3に示す。

(C)3.1節で述べた(1-B)と(2-B)および(1-B)と(2-D)の機能が統合されるため、それぞれの機能を独立して利用できない。

このように、システムコール fork や exec をアレンジして「プロセス生成ノード」や「プロセス走行ノード」を指定することは難しい。

そこで、プロセスが現在走行しているノードを意味する「カレントノード」の概念を新たに提案する。「カレントノード」は、グローバルトリーと関連して次の特徴を持つ。

(1)「カレントノード」としては、ネットワークルートディレクトリまたはネットワークディレクトリのみが成り得る。

(2)「カレントノード」がネットワークルートディレクトリまたは自ネットワークディレクトリの時、走行ノードが自ノードである。

(3)「カレントノード」をあるネットワークディレクトリに変更するということは、そのプロセスが自ノードからそのネットワークディレクトリに対応するノードへ移動して走行することを意味する。

「カレントノード」を変更してノード間をプロセスが移動する機能(以降、プロセス移動機能と呼ぶ)をひとつのシステムコール(例えば warp と命名する)で提供すると、3.1節で述べた機能は、本システムコールと既存のシステムコール fork や exec と組み合わせることですべて実現できる。

chdir("//n0"); chmod("//n2/f1/a",…); exec("//n2/f1/a",…);	chmodはファイルが存在するノードn2で実行されるが、execは自ノードn0で実行される。

図3 整合性が悪い例(グローバルトリーは図2の場合)

3.3機能の選択

3.2節でも述べたように、グローバルトリーは分散したファイルを統一管理しているため、既存システムコールのインタフェースのままで「プログラム存在ノード」の指定を可能にしている。そこで、3.1節で述べた(2-C)の機能は、グローバルトリーを導入して、APインタフェースを保存したシステムコール exec 機能の自然な拡張で実現する。

プロセス移動機能を新システムコールで実現する。これにより、3.1節で述べた(1-B)の機能は fork&warp の組み合わせ、(2-B)や(2-D)の機能は warp&exec の組み合わせで実現できる。

プロセス移動機能により、既存のシステムコール仕様に「そのシステムコールを実行するノード」をパラメータとして追加したライブラリを提供するだけで、リモートに拡張可能なシステムコール機能の約3/4はネットワーク化できる。リモートへのプロセス生成とリモートファイルのモード変更の場合について、ライブラリの例を図4に示す。図4の例2にある分散した親子プロセス間の wait/exit による通信については、5章で述べる。

4.リモートプログラム起動方式

UNIXでプログラムを起動する時(exec システムコール)の処理フローを図5に示す。図5の処理の中で、起動プログラムがローカルかリモートかにより処理内容が異なるのは処理AとBおよびDの部分である。この処理部分でリモートファイルア

例 1. リモートへのプロセス生成

```
rtfork ( place )  char    *place;
{
        int    pid;
        if( ( pid = fork( ) ) = = 0 )
                warp ( place );
        return ( pid );
}
```

例 2. リモートファイルのモード変更

```
rtchmod ( place, path, mode )
char    *place, *path;
{
        int    pid;
        if( ( pid = fork( ) ) = = 0 ){
                warp ( place );
                chmod ( path, mode );
                exit( );
        }
        wait ( &pid );
        return;
}
```

図 4 ネットワーク化された
システムコールライブラリの例

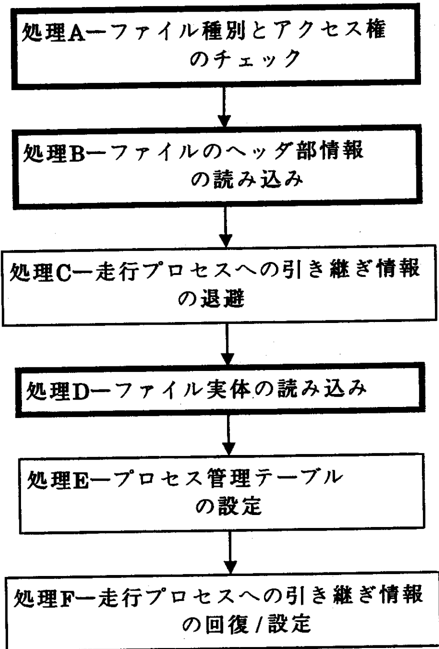


図 5 execシステムコールの処理概要

クセス機能[3]を利用して、他ノードにあるLMファイルをメモリ上にロードする。

4.1常駐/非常駐の制御方式

リモートプログラム起動機能により起動したプロセスは、走行しているノード上にLMファイルがないため、そのプロセスに対するメモリ常駐/非常駐の制御方式が問題になる。方式案を以下に示す。

[案A] プロセス走行ノードのローカルDKを用いたスワップ方式

[案B] LMが存在するリモートDKを用いたスワップ方式

[案C] スワップせずメモリ常駐化

案Aは、ローカルDK内にLMファイルがないため制御が複雑となる。また、案Bは、スワップ処理時間が大きく、かつCPUやLANの負荷も大きい。そのため、案Cを採用する。

プロセスをメモリ常駐化する場合、メモリ資源を圧迫しないためには、起動できるプログラムサイズに制限を設けることが必要となる。一方、OSが提供しているコマンドやツールのプログラムは、種類が多いものの、各LMファイルサイズは小さい。例えば、4.1bsd版UNIXの場合、プログラムは2百数十個あり、

①32キロバイト以下のLMファイルが約80%

②64キロバイト以下のLMファイルが約90%

③128キロバイト以下のLMファイルが約95%

である。従って、起動できるプロセスの最大サイズは、64キロバイト程度で充分と思われる。

4.2グループの扱い

システムコール execのパラメータでグループファイルを指定した場合の問題点は、「プログラム存在ノード」をどこにするかの扱いである。

「プログラム存在ノード」をグループファイルからの入力系システムコールと同様に setnod システムコール[3]の指定に従うことも考えられる。しかし、本OS上で提供する電子会議サービスを実現するため、自ノードを「プログラム存在ノード」とする。これは、電子会議サービスのプログラムが、ファイルへの入出力は setnod システムコールの指定に従って行い、プログラム起動はローカルのLMファイルで行うことを必要としているためである。

5.プロセス移動と分散プロセス間通信方式

プロセス移動機能を以下のねらいで実現する。

(1) プロセス移動機能により、①リモート処理用の種々のライブラリ構築を容易にし、

②パイプやシグナルやwait/exitによる分散プロセス間通信を実現する。

(2) 同一CPUで同一版のUNIXであっても、OS生成パラメータ値が異なっているノード間でのプロセス移動を実現する。例えば、利用者が接するワークステーションと大量の記憶装置を持つファイルサーバ間(ブロック入出力用のバッファ量などが異なる)のプロセス移動など。

5.1移動情報の管理方式

シグナルやwait/exitによる通信は、パイプと違い、前もって通信のアクセスパス(パイプのファイル記述子に相当するもの)を持たない。そのため、移動したプロセスが移動元のプロセスと通信するために、プロセスは過去に走行したノードに関する情報を何らかの形で保持しなくてはならない。

移動に関する情報を、LAN内の特定ノードで集中管理する方式が考えられるが、プロセスの生成/移動/消滅の度に、そのノード上の管理情報の更新が必要となり、LANの負荷が増大し好ましくない。移動速度の向上とLAN負荷の軽減を図るために、移動に関する情報を、関連するノードで管理して通信する方式として、以下の3案が考えられる。

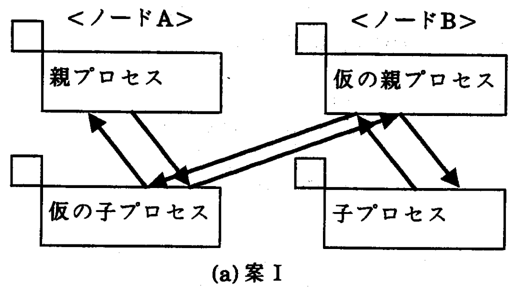
[案Ⅰ]移動元に仮の子プロセス、移動先に仮の親プロセスを生成して通信する方式

[案Ⅱ]プロセスの管理テーブルに情報を追加し、移動元に仮の子プロセスの管理テーブル、移動先に仮の親プロセスの管理テーブルを生成して通信する方式

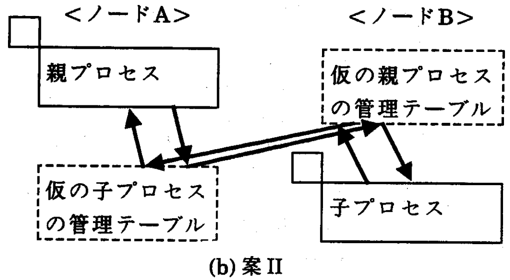
[案Ⅲ]プロセスの管理テーブルに情報を追加し、移動元に仮の子プロセスの管理テーブルを生成して通信する方式

各方式案の概念を図6に示す。

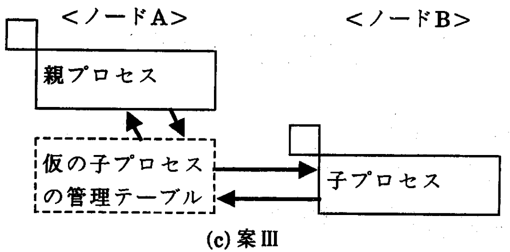
案Ⅰは、親/子プロセスの処理はローカル/リモートを意識しないという長所があるものの、①通信速度が遅く、②仮のプロセスが多数存在してシステム性能の低下をまねく。案Ⅱは、プロセスの管理テーブルへの情報追加は案Ⅲよりやや少ないものの、①通信速度は案Ⅲより遅く、②プロセスの管理テーブル資源を圧迫する。案Ⅲは、処理の効率が良く通信速度が速い。また、プロセスの管理テーブルに追加する情報は、以下に示す程度であり、管理テーブル規模の増加は少ないため、案Ⅲが望ましい。



(a) 案Ⅰ



(b) 案Ⅱ



(c) 案Ⅲ

⇄ 通信パス

図6移動プロセスの管理方式と通信パス

(1) 移動元のノード情報
(2) 移動先のノード情報
(3) 移動先のプロセス識別子
(4) プロセスの実体はなく管理テーブルのみという状態の情報

移動プロセスの管理の例を表1に示す。

このようにプロセスを管理することにより、シグナルやwait/exitによる通信は、システムコールkillやexitを管理テーブルの情報に従い、リモートへシステムコール転送することで行える。

5.2移動時の転送データ内容

移動時には、プロセスのテキストやデータだけではなく、ユーザスタックの内容や管理テーブルの情報も転送する必要がある。

これは、プロセスが持っているファイルの

オープン数やアクセス位置といったファイルアクセス系の環境を、プロセス移動の前後おいて同じにすることにより、①リモート処理用のライブラリ構築を容易にし、②分散プロセス間でのパイプによる通信(システムコール pipe と warp およびリモートファイルアクセス機能を利用)を可能にするためである。

5.3移動後のプログラム開始

移動の前後でOS生成パラメータ値が異なっている可能性があるため、移動前に利用していたカーネルスタックの内容は、移動先ではほとんど利用できない。そのため、移動後のプログラム開始処理では、強制的にプロセス移動要求直後のユーザプロセスの位置から走行させる。

プロセス移動のデータ転送と処理シーケンスの例を図7に示す。

5.4グループの扱い

グローバルトリーでは、グループのメンバを管理していない(メンバの管理はAP責任となっている)ため、グループへの移動時に、ファイルアクセス系の環境を保存したり、移動情報の管理を完全に行うことはできない。しかし、複数ノードへの一括処理を行う場合(例えば、エラー情報の一括通知)にはグループへのプロセス移動機能が有効なので、以下の形で実現する。

(1) 移動元ではプロセス消滅扱いとする

(2) 移動先ではプロセス生成扱いとする(親プロセスは init プロセス)

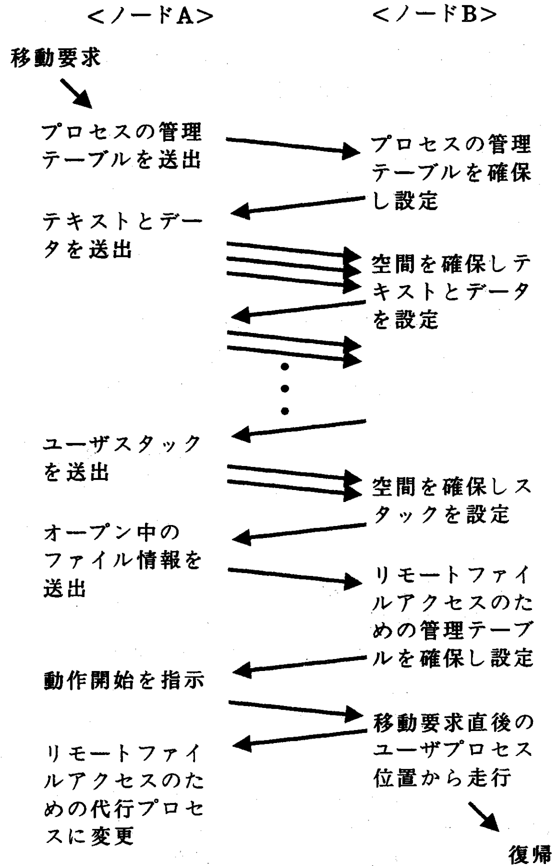


図7プロセス移動の処理シーケンス例

表1 移動プロセスの管理の例

場　　合	移動元のノード情報	移動先のノード情報	移動先のプロセス識別子
自ノードで生成したプロセスが他ノードに移動した場合#	意味なし	移動先の他ノードに対応するネットワークディレクトリを管理しているテーブルへのポインタを設定	移動先の他ノードにおけるプロセス識別子を設定
他ノードで生成されたプロセスが自ノードに移動してきた場合	移動元の他ノードに対応するネットワークディレクトリを管理しているテーブルへのポインタを設定	意味なし	意味なし
他ノードで生成されたプロセスが自ノードから他ノードに移動した場合#	移動元の他ノードに対応するネットワークディレクトリを管理しているテーブルへのポインタを設定	移動先の他ノードに対応するネットワークディレクトリを管理しているテーブルへのポインタを設定	移動先の他ノードにおけるプロセス識別子を設定

#:プロセス状態は管理テーブルのみという特別状態を示す

6.まとめ

UNIXをベースとして分散処理OSを構築する際に重要なプロセス制御機能をネットワーク化する方式について述べた。

本方式の特徴は、「カレントノード」の概念を導入して走行環境を保ったままノード間をプロセスが移動する機能と、グローバルトリーを利用したリモートのプログラムによるローカルプロセス起動機能、にある。

具体的な実現機能は以下の通りである。

(1) 走行環境を保存し、かつOS生成パラメータ値が異なるノード間でのプロセス移動を可能にする。

(2) リモートファイルアクセス機能を利用して、リモートにあるプログラムを用いてローカルなプロセス上での起動を可能にする。

(3) ノード間を移動したプロセスを移動元や移動先の情報とともにテーブルで管理し、かつリモートファイルアクセス機能を利用することにより、パイプ・シグナル・親子プロセス間の同期の機能をネットワーク化する。

以上の結果、処理の分散による効率化や記憶装置の有効利用が可能になる。さらに、既存の多くの機能をそのままネットワーク環境で利用することも可能になる。

最後に、検討を進める際に、貴重な御意見をいただいた関係各位に深く感謝します。

<参考文献>

[1] 坂本 他 :EINSにおける分散処理システムの構成, 情処学会「LAN/マルチメディアの応用と分散処理」シンポジウム, pp151-158(1984).

[2] 鈴木, 谷口 :分散処理OSにおける電子会議サービスの構成, 情処学会 第29回全国大会 1H-2.

[3] 谷口, 鈴木, 瀬々 :ファイル管理機能のネットワーク化による分散処理OSの構成法, 情処学会論文誌, 第27巻第1号, pp.56-63(1986).

[4] Rowe, L.A.and Birman,K.P. : A Local Network Based on the UNIX Operating System, IEEE Trans. Software Eng., SE-8(2), pp.137-146(1982).