オペレーティング・システム 47－1
（1990．6．8）

並列型ＯＳの設計と実現

中山 泰一　田胡 和哉*　森下 巖
東京大学工学部

システム機能自体を並列化したＯＳのことを，並列型ＯＳとよぶ．プロセス・ネットワーク方式を用いて並列型ＯＳを実現することを試みた．プロセス・ネットワーク方式では，相互排除アクセスされる資源の各々に軽量なプロセスを配置し，それらを同期式の通信で結合することによりシステムを実現する．

疎結合型のマルチプロセッサ・システムを対象とし，実際にシステムを設計，試作し，実行性能を評価した．実現したシステムは，広く実用されているＵＮＩＸ**と互換性を持つ．実験の結果，利用者プログラムの処理時間が30％程度短縮され，システム内部での並列処理により処理性能が向上することが確認された．

DESIGN AND IMPLEMENTATION OF A PARALLEL OPERATING SYSTEM

Yasuichi Nakayama　Kazuya Tago*　Iwao Morishita
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113 Japan

A parallel operating system has been designed and implemented on a loosely-coupled multiprocessor system employing the process network architecture. The operating system consists of a number of light-weight processes interconnected by rendezvous communications and is compatible with the UNIX** system. It has been shown that when this process network is distributed on multiple computer units with an optimum assignment, some processes can run in parallel with the others, and the average processing time of typical application programs is decreased by 30%.

１．　はじめに

本論文では，疏結合型マルチプロセッサ・システムを対象とし，プロセス・ネットワーク方式を用いる並列型ＯＳをインプリメントしてその性能を評価した結果を報告する．

プロセス・ネットワーク方式を用いるＯＳの構成法には，

（１）設計，保守が容易である，

（２）異機種への移植が容易である，

（３）疏結合型マルチプロセッサ・システムに自然な形で適用できる．

などの利点のほか，

（４）ＯＳに内在する並列性を自然な形で抽出できる，

という利点がある．すなわち，ＯＳは一群のシステム・プロセスのネットワークとして構成されているので，これらのシステム・プロセスをマルチプロセッサ・システムの各ユニットに適切に配置することにより，

（１）ＯＳのシステム機能自体の並列実行，

（２）ＯＳのシステム機能と利用者プログラムとの並列実行，

が実現できる可能性がある．これにより，システム・コールの応答時間が短縮され，利用者プログラムの実行時間が短縮される可能性がある．

本研究では，まず，ＯＳに内在する並列性の抽出について考察し，試作システム上において種々の配置について実験を行い，どの程度の時間短縮が可能であるかを明らかにする．インプリメントに使用したマルチプロセッサ・システムは，少数のコンピュータ・ユニットが共有メモリで結合された方式のもので，ユニット間のメッセージ通信にはこの共有メモリを使用した．また，インプリメントに使用したＯＳは，田胡，益田[1]，高野ほか[2]が報告したプロセス・ネットワーク方式によるＵＮＩＸである．

２．　プロセス・ネットワーク方式

２．１　プロセス・ネットワークによるＯＳの実現

インプリメントに使用したＯＳは、下記に説明するプロセス・ネットワーク方式[1]によるものである．

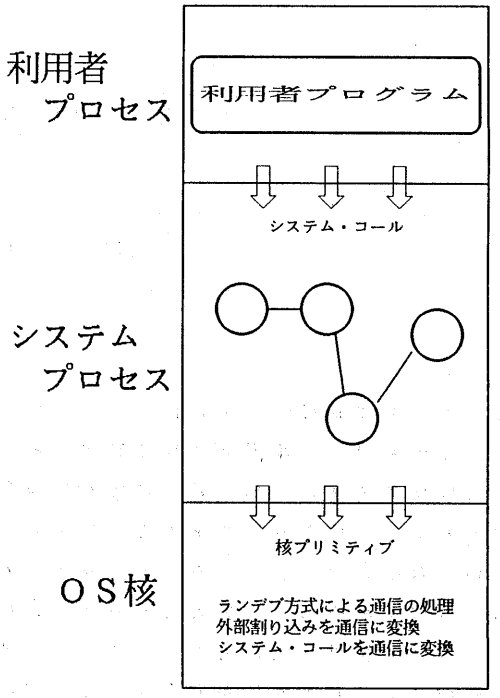


図2．1　プロセス・ネットワーク方式によるＯＳ

並列処理によって処理性能の改善を図るためには，プログラムをできるだけ多数の並列実行単位に分割すること，および，並列処理の実現コストを軽減することが必要である．プロセス・ネットワークは，この目的に適合する性質をもつ．

プロセス・ネットワークを用いたＯＳの構成を図２．１に示す．ＯＳを実行するプロセスをシステム・プロセスとよび，次にあげる方針で設計する．

（１）システム・プロセスを，静的に配置された軽量なプロセスにより実現する．軽量なプロセスなので，全体で単一の論理アドレス空間内で動作し，このためプロセスの切換えを単純な機構で実現できる．システム・プロセスの動的な生成，消滅，およびシステム・プロセスへの動的なメモリ割り当ては行わない．

（２）相互排除アクセスされる計算機資源の管理機能をプロセス分割の単位とする．周辺機器，バッファ，オープンされているファイル等に対し

て，それらを管理するシステム・プロセスを割り当てる．また，利用者プロセスの起動，停止，およびシステム・コールの受け付け等の，利用者プロセスの管理を行うシステム・プロセス（スーパーバイザ・プロセスとよぶ）を設ける．

（3）プロセス間通信のみによって，システム・プロセスを結合する．また，システム・コール等の割出しや，周辺機器からの割込みをシステム・プロセス間の通信と同じ形式に変換して，システム・プロセスに伝達する．システム・プロス間の通信関係は固定で，実行時に新たな通信経路を生成しない．

上記の構成で必要となるプロセス間通信機能はランデブ方式のメッセージ通信を用い，ＯＳ核によって実現する．すなわち，システム・プロセスは，ＯＳ核が提供する核プリミティブ（call,acc,endr）を呼び出すことにより，プロセス間通信を行う．プロセス・ネットワーク方式によるシステムでは，システム・プロセス間で共有変数を持たない．

また，プロセス・ネットワーク方式では，プロセスを実行時に生成，消滅する方式に比べて，実行時のシステム構成の変更が行いにくくなる一方，プログラミング時にシステムの実行時の動作を記述することが可能である．これをもとに，設計，デバッグ，およびシステムの構築がより容易なものとなる．

２．２　プロセス・ネットワークによるＵＮＩＸの再構成

プロセス・ネットワーク方式によりＵＮＩＸシステムを再構成した．システム・プロセスを配置する資源管理機能の分割単位は，ＵＮＩＸにおける資源アクセスの相互排除単位を踏襲し，ＵＮＩＸシステムの資源管理アルゴリズムをそのまま適用した．

利用者プロセスを管理するスーパバイサ・プロセスでは，ＵＮＩＸと同一の仕様のシステム・コールを利用者プロセスに提供するように設計した．

スーパーバイザ・プロセスの利用者プロセスへの割当ては実行中に行われるが，システム・プロセスの配置を固定にしているので，割り当てることができる最大個数のスーパーバイザ・プロセスをあらかじめ定義しておく．このように同一の機能を有する複数のプロセスの集合をプロセス群とよぶことにする．１つのプロセス群に属する各プロセスの通信相手は，すべて同一である．

プロセス群に属するプロセスの割当ては，そのプロセス群の管理プロセスにより実行時に決定される．このようにしてシステム・プロセスの配置を行った結果，これまでに図２．２に示すような構造によるＯＳが設計されている[1]．

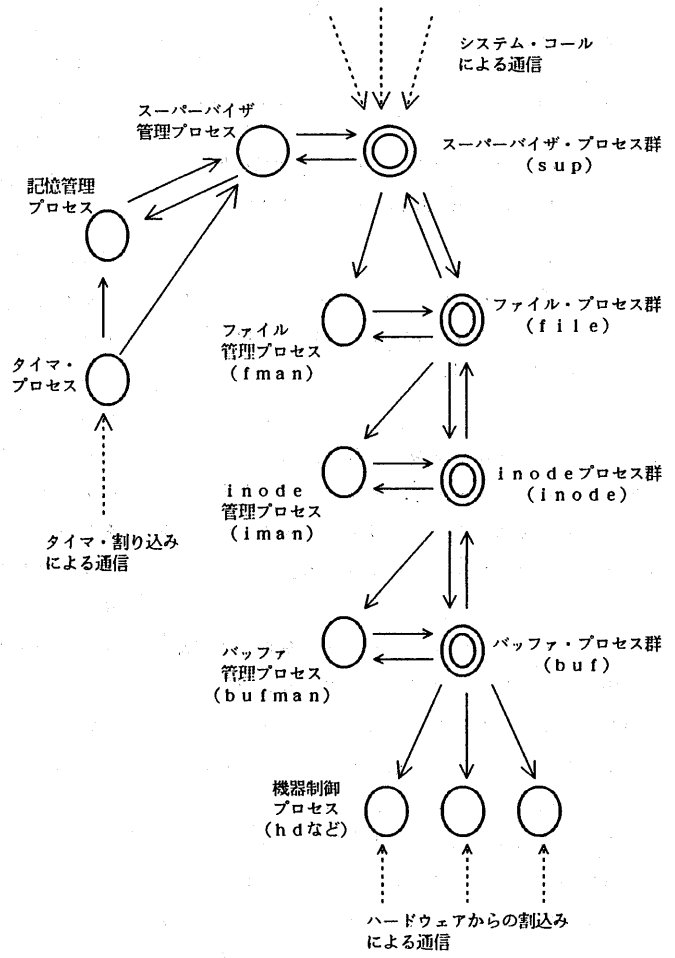
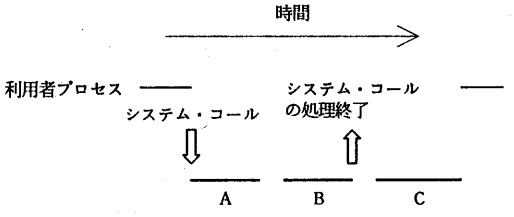
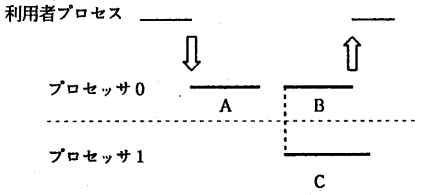


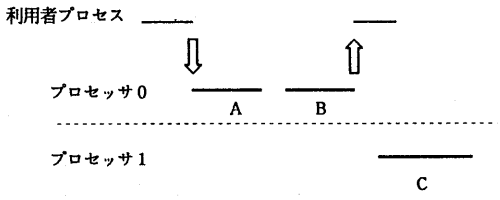
図２．２　プロセス・ネットワーク方式によるＵＮＩＸの再構成



（a） ＵＮＩＸにおけるreadシステム・コールの処理



（b） システム内部での並列処理



（c） システムと利用者プロセスとの並列処理

図3．1 ＯＳに内在する論理的な並列性

３．プロセス・ネットワーク方式のＯＳに内在する並列性

３．１ ＯＳに内在する論理的な並列性

２．で説明したプロセス・ネットワーク方式を用いて，ＵＮＩＸと同一の外部仕様をもつ並列型ＯＳを設計した．ここで，ＯＳに内在する並列性について述べる．

設計したシステムでは，プロセスを配置する資源管理機能の分割単位は，ＵＮＩＸにおける資源アクセスの相互排除単位を踏襲し，ＵＮＩＸシステムの資源管理アルゴリズムをそのまま適用した．まず，ＵＮＩＸシステムに内在している並列性について考える．

システムの動作特性を測定した経験によれば，システム処理の大半はファイル入出力によって費やされている．そこで，ファイル・システムの動作に注目する．

例として，ファイル入力を行うreadシステム・コールは，図３．１（ａ）に示す手順によって処理される．Ｂはファイル入力そのものの処理であるが，Ｃはもう１つ先のブロックに対する入出力要求をあらかじめ外部入出力装置に対して出し，バッファに先読みしておく部分である．この場合ＢとＣには逐次的に処理する必要がないので，論理的には，図３．１（ｂ）に示すような並列処理が可能である．

図３．１（ａ）のような処理で，ＢとＣとが並列に処理できない場合でも，図３．１（ｃ）に示すように処理すれば，利用者プロセスとシステム・プロセスとが並列に動作し、利用者レベルでレスポンスが上がる可能性がある．

３．２ 並列性の抽出

プロセス・ネットワークによるシステムは，図３．２に示すように，下位のプロセスに要求を出し，その結果を待つということを次々に行う入れ子構造になっている．それぞれのプロセス処理の粒度が小さいので，そのかたまりの形でＯＳの並列性を抽出することができるのではないかと思われる．

例として前節と同様に，readシステム・コールの処理について，プロセス・ネットワーク方式によるＯＳがどのように動作するのかを考える．プロセス・ネットワークの動作はシステム・プロセスの通信参照関係を追うことによりある程度まで推定することができる．これは，プロセス間の通信の実行順序に半順序関係があるためである．

通信の実行に着目してreadシステム・コールの処理を見ると，プロセス・ネットワークの動作は図３．３のようになる．プロセス処理の中で，通信終了のあと両方のプロセスが動作するような分岐の部分があり，その後の処理は互いに異なるプロセッサ上に配置され

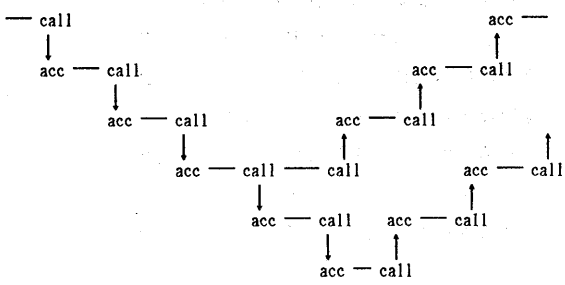


図3．2 プロセス・ネットワークによる並列性の抽出

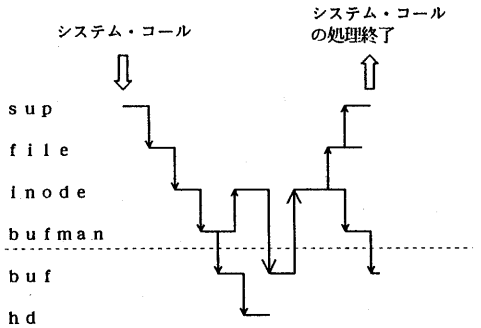


図3．3　readシステムコールにおけるプロセス・ネットワークの動作

ているときには並列に処理される．すなわち図3．3において破線で2台のプロセッサに分けられている場合に，破線の上側の処理と下側の処理は並列実行可能である．

ただし，分岐後の処理であってもその両方が破線の同じ側のプロセスで行われる部分は，どちらか1方だけしか実行できない．このとき，そのどちらが先に実行されるかはシステム・プロセスのスケジューリングのメカニズムに依存している．

システムの並列度は，各プロセスのプロセッサへの配置のしかたに大きく影響される．プロセス配置を適切に行えば，システムの並列度を上げることも可能で，プロセス・ネットワークが効率よく動作するものと思われる．

3．3　通信オーバーヘッドの影響

並列処理による性能の向上の一方で，通信処理のオーバーヘッドの増加が予想される．異なるプロセッサ上のプロセス間の通信は，同一のプロセッサ上のプロセス間の通信に較べて多くの時間を要する．このため，前節までに述べた並列処理の効果がそのまま現れることはない．

システム・プロセスの分散配置の方法によっては，異なるプロセッサ間の通信の頻度，量が変わる．ゆえに，並列度が上がる配置であるとともに，通信オーバーヘッドができるだけ小さくなるような配置を行うことが有効である．

4．　システムの実現

4．1　インプリメント対象

2．で述べたプロセス・ネットワーク方式を用いて並列型ＯＳを実現した[3]．

今回，設計を行った並列型ＯＳは，沖電気製のＩＴＣシステム上に実現した．ＩＴＣシステムは図4．1に示す通り，68020マイクロプロセッサ，68851ＭＭＵ，4ＭＢの局所メモリなどから構成されるコンピュータ・ユニットを，共有バスを用いて最大7台まで実装することができる．これに2ＭＢの共有メモリが装備されている．

ユニット間の通信用として，各ユニットにパイプ・レジスタが用意されている．送信先のユニット用のパイプ・レジスタにデータを書き込むことにより，送信先のプロセッサに割込みをかけることが可能である．この機能と共有メモリとを使って高速度のユニット間メッセージ通信機構を実現した．共有メモリは通信路としてのみの利用である．

4．2　ＯＳ核の通信機構

並列型ＯＳのＯＳ核として，マルチプロセッサ用の通信機構の設計を行なった[4]．この通信機構は，システム・プロセス間での分散透明な通信機能を実現する．

システム・プロセスは各プロセッサの局所メモリに配置され，局所スタックを用いて動作している．システム・プロセスが通信を行うとき，通信相手のプロセスが配置されているプロセッサを識別しないでよい環境が必要である．

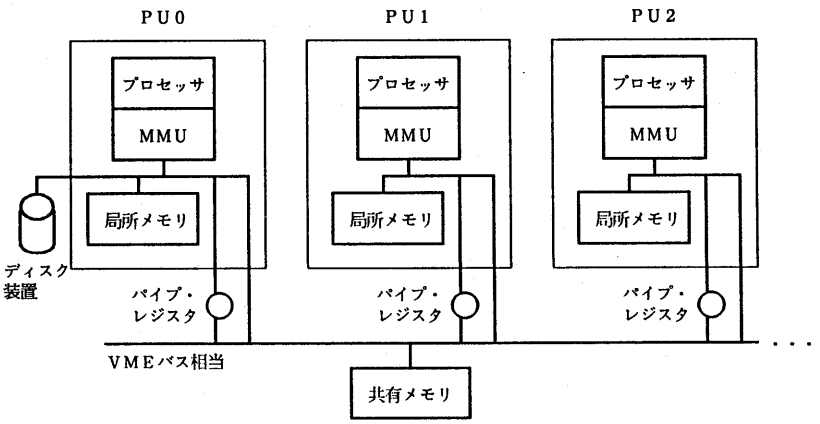


図4．1　ＩＴＣシステムのハードウェア構成

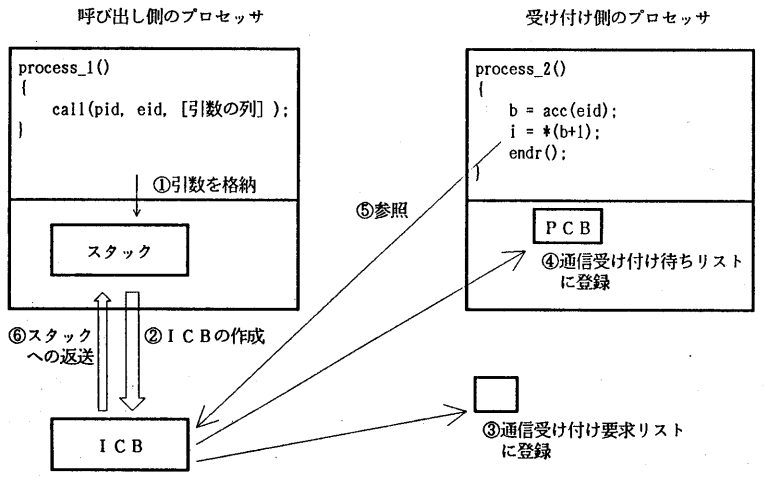


図4．2　ＯＳ核のプロセッサ間通信機構

図４．２にＯＳ核におけるプロセッサ間通信の実現方式の概要を示す．異なるプロセッサ間通信用の通信パケットであるＩＣＢ(Interprosessor Communication Buffer)のみを共有メモリに置く．

４．３　システムの実現

単一の68010システム上で、プロセス・ネットワーク方式によって実現されたＵＮＩＸシステムと同一の外部仕様をもつＯＳが動作している．このプロセス・ネットワークを複数のプロセッサ上に分散して配置することにより，システムを実現した．これは，標準のＵＮＩＸシステムのアプリケーション・プログラムをそのまま実行することができる．

実現効率の点から，固定ディスクのかわりにＲＡＭディスクを用意した．システムの構成は，例として，図４．３のようになる．システム・プロセスのプロセッサへの配置は自由である．このとき，同一のプロセス群に属するプロセスは同一のプロセッサ上に配置する．

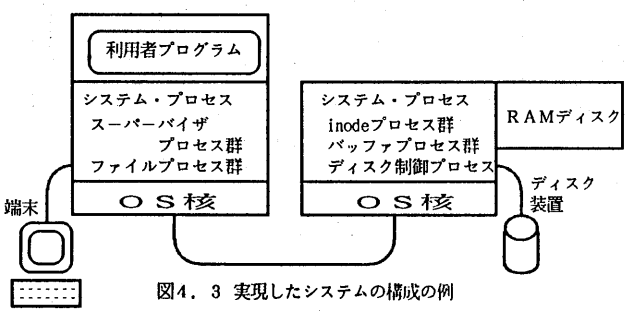


図4．3 実現したシステムの構成の例

５．　実験と考察

５．１　実験

プロセス・ネットワークの複数のプロセッサへの分散配置の方法を変えながら，アプリケーション・プログラムを実行して，実行時間を測定した．これは，１つのプログラムの開始から終了までに実際にかかる応答時間を測定した．

ここでは，プロセッサの台数を２台とし，利用者プロセスは１つのプロセッサ上にしか配置しないこととした．また，システム処理の大半はファイル入出力によって費やされているので，ファイル・システムに関するシステム・プロセスの配置を変え，それ以外のシステム・プロセスは固定して実験を行った．

７種類のシステムを実現し，利用者プログラムの実行時間について測定を行った．その概要を図５．１に示す．それぞれの構成をＡ～Ｇと名づける．Ａ構成が単一プロセッサ構成であり，Ｂ～Ｇはシステム機能の一部が他のプロセッサ上に移行されている．

測定に用いたアプリケーション・プログラムは，ファイルのコピーについてはcp，Ｃ言語で書かれたプログラムのコンパイルにはccを用いた．どちらも標準のＵＮＩＸシステム上で動作しているものである．

５．２　実験結果

実行結果を表５．１に示す．これによると，システム機能の一部を他のプロセッサに移行したＢ，Ｄ，Ｅ，Ｆ，Ｇ構成のものでは性能が向上した．利用者プロセスの処理時間は一定であるので，システム・コールの処理時間が改善されていると言える．

最も性能が向上したのはＥ構成のもので，単一プロセッサ構成のＡ構成に比べて３０％前後処理時間が短縮した．

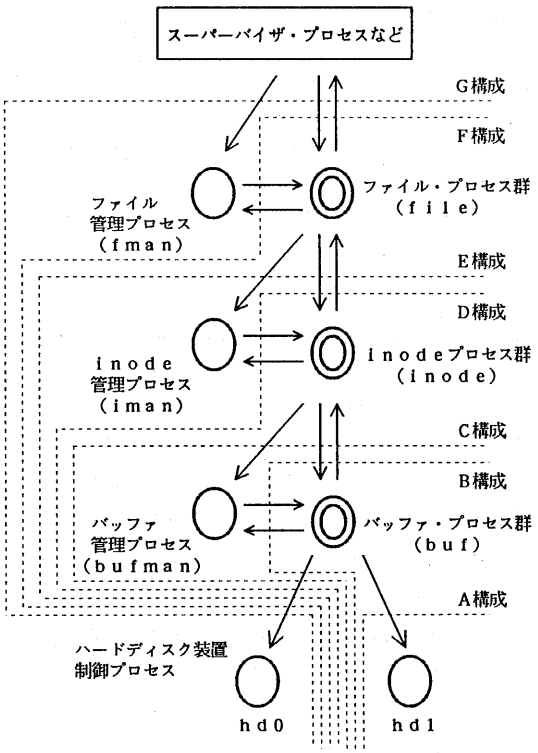


図5．1　実験に用いた7種類のシステム構成
（破線はそれぞれの構成でのプロセッサ分割を示す）

逆にC構成では，単一プロセッサ構成のA構成に比べて処理性能が悪化した．このように，システムの配置によって，単一プロセッサ構成のシステムより性能が改善されているものと，逆に悪化しているものがあり，プロセスの配置法の検討が重要であると確かめられた．

5．3　実験結果の考察

（1）プロセス配置と並列性の考察

実現したシステム上での，readシステム・コールの処理における実際の実行系を考える．

単一プロセッサ構成のA構成（図5．2）においては，結果的にすべてのシステム処理が逐次的に行われる．これに対し，E構成（図5．3）ではバッファの先読み機構がシステム・コールの処理と並列に動作している．さらに，システム・コールの処理が完了した後に行われる，次のシステム・コール受付けの準備やスケジューリング処理といった，後処理の部分が並列に動作する．このため，システム・コールの処理が完了した段階ですぐに利用者プロセスが動き始めるので，A構成に較べてシステム処理全体の処理時間が短縮されている．これは，3．で述べたOSに内在する並列性が効率よく抽出されている場合といえる．

逆にC構成の場合（図5．4）は，システム機能が複数に分散しているにもかかわらず，バッファ先読みの部分も，システム・コールの後処理の部分も逐次的に動いてしまい，2台のプロセッサで処理を行う効果が現れない．結果として，むしろ単一プロセッサの場合よりも性能が悪化している．

（2）　通信オーバーヘッドの考察

実現したシステムにおいて実測を行った結果，異なるプロセッサ上のプロセス間の通信が，同一のプロセッサ上のプロセス間の通信に比べて2.5～7.4倍のコストを要することがわかった．通信引数の数，通信で送られるデータの量により，異なるプロセッサ上のプロセス間の通信のコストにちがいがある．

表5．1　　実行結果（単位はsec）

	A	B	C	D	E	F	G
1つのファイルを読みこむ							
(250Kbytes)							
i)hd0上のとき	3.2	3.0	3.6	2.4	2.3	2.4	2.4
ii)hd1上のとき	3.4	2.6	3.4	2.1	2.0	2.5	2.5
複数のファイルを読みこむ							
(7files, 260Kbytes)							
i)hd0上のとき	3.6	3.4	3.9	2.8	2.6	2.8	2.8
ii)hd1上のとき	3.6	3.0	3.6	2.3	2.2	2.7	2.7
(29files, 450Kbytes)							
i)hd0上のとき	6.5	6.0	7.2	5.1	4.9	5.2	5.2
ii)hd1上のとき	6.7	6.2	6.6	4.4	4.1	5.1	5.1
hd0からhd1へのコピー							
(260Kbytes)	7.8	6.6	8.4	5.6	5.2	6.1	6.1
Cプログラムのコンパイル							
プログラムa	13.9	12.8	15.1	11.6	11.1	12.4	12.5
プログラムb	14.8	13.8	16.1	12.3	11.8	13.3	13.4

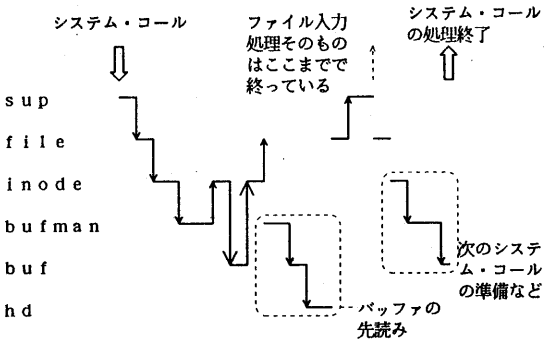


図5．2　A構成のシステムにおけるreadシステムコールの実行系

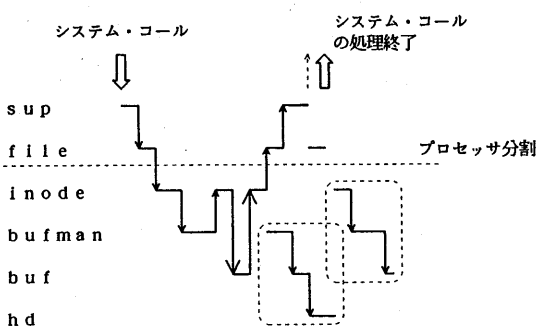


図5．3　E構成のシステムにおけるreadシステムコールの実行系

open，read，closeの各システム・コールの処理中に行なわれる通信のオーバーヘッドを合計してみると表5．2のようになる．数値は，同一のプロセッサ間での1回の通信に必要な通信オーバーヘッドを1としたときの比率である．

E構成のものは比較的に通信オーバーヘッドが小さい．その反面，C構成では通信オーバーヘッドが非常に大きなものとなっている．これは，複数のプロセッサに分散したとき，通信回数が多いところ，または，通信データ量が多い通信が，異なるプロセッサ間で行われることになったためと思われる．

表5．2　通信オーバーヘッドの合計

	A	B	C	D	E	F	G
openシステム・コール	69.0	74.3	84.8	71.4	70.3	71.8	74.3
read	19.0	19.1	22.4	20.7	20.7	21.0	21.0
close	5.0	5.0	5.0	8.3	6.8	8.0	6.5

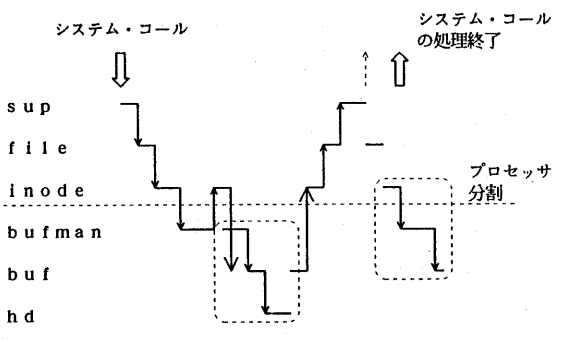


図5．4　C構成のシステムにおけるreadシステムコールの実行系

6．まとめ

通信により結合されたプロセス集合体を用いて，並列型ＯＳを設計した．そして，システムの実現を行った結果，処理性能が向上することが確かめられた．

さらに，実現した並列型ＯＳの性能評価の方法を考え，実験を行い結果を評価した．その結果，最も性能が向上した構成では，システム内部での並列処理が効率よく行われていることが判明した．ＯＳに内在している並列性が実際に，3．で考えたように抽出されている．

また，配置の方法によってはシステム内部で並列処理が行われず，かえって性能が悪化する場合があることがわかった．システム機能の配置法の検討が重要であり，これが並列型ＯＳの性能改善を図るときの大きな課題である．今後も，配置法についての詳細な検討を行っていく予定である．

参考文献

[1] 田胡和哉，益田隆司：ＯＳの構造記述に関する試み，情報処理学会論文誌，vol.25 No.4 (1984)

[2] 高野陽介，田胡和哉，益田隆司：プロセス・ネットワークによる分散型ＯＳの設計，情報処理学会論文誌，vol.29 No.4 (1988)

[3] 中山泰一：並列型ＯＳの構成方式に関する研究，東京大学大学院工学系研究科情報工学専攻修士論文 (1990)

[4] 田胡和哉，中山泰一：軽量なプロセスの集合によるマルチプロセッサ用ＯＳの設計，第38回情報処理学会全国大会 (1989)