Vol. 25 No. 4 情報処理学会論文誌 July 1984

オペレーティング・システムの構造記述に関する一試み†

田胡和哉†† 益田隆司†††

オペレーティング・システム記述法の改善の試みは，とくに，言語技術と関連して数多く行われてきた．このとき問題となるのは，オペレーティング・システムが内部に高い並行性を有している点であり，その制御構造が全体の記述に大きな影響を与える．そのため，制御構造の明確化が重要な課題となるが，従来の試みにおいてこの点が十分明らかにされているとはいいがたい．本論文では，制御構造の明確化を図る目的で，オペレーティング・システムを，通信のみで結合されたプロセスを相互排除アクセスされる資源の管理単位に配置することにより設計する方式を提案する．このような設計法により，プロセスによる変数の抽象化が行われモジュラリティが向上するとともに，従来のモニタを用いる方法等に比較して制御構造が簡素化し，記述の改善が図れると期待される．また，システムの制御をプロセス内部の動作と通信の実行に分割して記述することにより記述が階層化され，制御構造の明確化が図れる．さらに，小型計算機用システムとして広く利用されている UNIX* をその外部仕様を保存したまま提案方式により再設計を行い，その実現可能性を示した．

1. はじめに

1970 年代後半以降，オペレーティング・システム (OS: Operating Systems) をとりまく環境は，半導体技術の急速な進歩とそれに伴うハードウェア価格の低下により，大きく変化している．とくに，小型，超小型機の高機能化が著しい．システムの形態も，従来のバッチ，TSS だけでなく，分散処理，個人専用システムといった形態が広く普及しつつある．そのため，OS に対しても，設計，保守の容易性，異機種への移行の容易性，分散処理への対応等，新たな要求が発生している．70 年代の末に発表されたいくつかの試作 OS[2),10)~12),19)] は，このような要求への対応を目指したものであり，いずれも，何らかの方法によるモジュラリティの改善，および，それを支援する記述言語の利用を試みている．

OS の設計方式を明らかにすることを最初に試みたのは，Dijkstra による THE システムである[3)]．そこでは，同期の機構の考案，OS の機能の階層性に焦点を合わせた設計等を試み，この分野における以後の研究の契機となっている．その後，Concurrent Pascal[6)] 等の抽象データ型言語の OS 記述への利用，順路式 (path expression)[1)]，モニタ (monitor)[8)]，ランデブ (rendezvous)[4)] 等の同期機構の考案等の言語技術と関連した試みが数多くなされている[14)]．

しかしながら，これらの試みにもかかわらず，いまだに，OS の設計に関して，

(i) その構造モデルが明確でなく，客観的なモジュール分割の基準がないこと，また，

(ii) 記述方式と性能あるいは動作との関係が明らかではないこと，

等の問題点が存在している．

そこで，われわれは，OS の構造記述に関する一つの方式を提案し，それを利用した OS の設計・開発を試み，上記の課題に対する知見を得ることを目指している[15)~18)]．提案方式では，OS を資源管理機能の集合体とみなし，各機能をプロセスによって実現する．そして，OS の大域的構造を，プロセスと通信の組合せによって，陽に記述することを試みる．これにより，論理，性能の両面から，根拠あるモジュール分割の方法を求める．

現在，提案方式によってプロセス分けを行った OS の設計，プロセスの動作環境を提供する OS 核の実現が完了した段階である．そこで，本論文では，提案方式による OS の実現可能性を報告することに重点をおいて，提案方式の内容，それに基づいて設計した OS の構造，OS 核の機能と性能，および，これまでに得られたいくつかの知見，考察等について述べる．

2. 設計の基本方針

OS は，内部に高い並行性をもつシステムであり，その制御は，手続き呼出しのほかに同期あるいは割込み機構等によって実現されている．これは，通常の順序的なプログラムが手続き呼出しのみによって結合されているのとは対照的である．このため，OS 記述の

† A Study of Description Method for Operating Systems by KAZUYA TAGO (Doctorial Program in Engineering, University of Tsukuba) and TAKASHI MASUDA (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学工学研究科
††† 筑波大学電子・情報工学系

* UNIX is a trademark of Bell Laboratories.

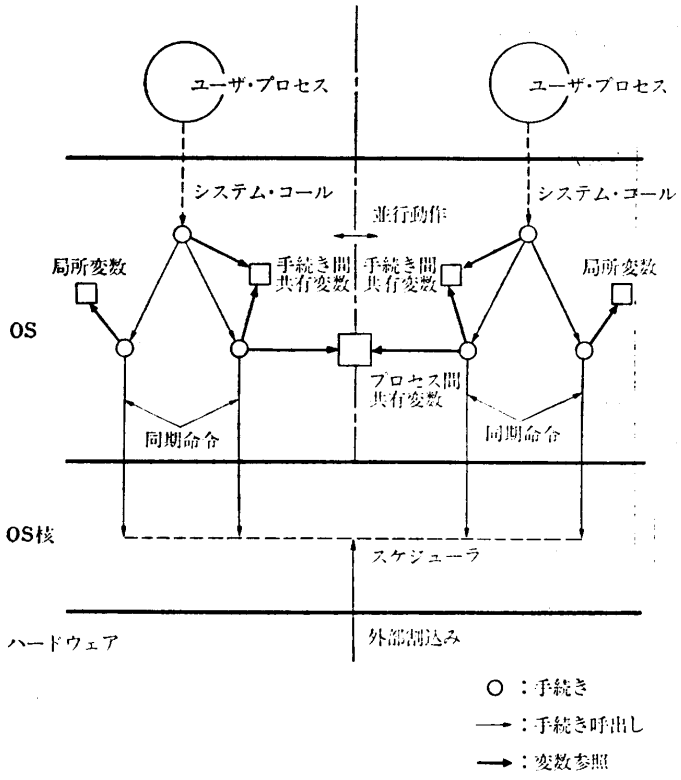


図 1　従来方式による OS の構造
Fig. 1　Structure of operating systems by existent methods.

構造化には，その制御構造を明らかにすることが重要な課題となる．われわれは，OS 内部の相互排除アクセスされる資源おのおのを別個のプロセスで管理するとともにそれらのプロセスを通信のみで結合することにより，その制御構造を記述することを試みる．本章では，まず，従来の OS の設計方式に関する技術について述べ，ついで，提案方式を述べる．

（1）　従 来 方 式

割込み処理あるいはプロセス切換えを行ういわゆる核の部分を除けば，OS 内部はプロセスによって実現されていることが多い．プロセスの機能およびそれらの間の同期あるいは割込みに関する設計を，プロセス分割と呼ぶことにすると，一般的に行われているプロセス分割の方法は，ユーザ・プロセスに 1 対 1 に対応するプロセスを OS 内部に設ける方法である．**図 1** に示すように，このような方式では，システム・コールはシステム・コール割込みを制御の入口とする手続き系列によって処理される．これらの手続きは，核の部分を除く OS の機能の大半を実現している．このとき，手続きを再入可能にし，手続きの局所変数およびスタック等の環境をユーザ・プロセスに 1 対 1 に対応して設けることにより，あるユーザ・プロセスの発行したシステム・コールは，OS 内部の対応するプロセスによって処理されることになる．また，2 次記憶装置等のユーザ・プロセス間で共有される資源の管理は，プロセス間共有変数を用いて実現される．このため，同期は，おもにプロセス間共有変数へのアクセスの相互排除の実現に用いられる．

このような方式は，手続き間あるいはプロセス間で共有変数を用いているので，性能上の問題も少なく，また，見かけの制御構造は単純である．おのおののプロセスは同一の手続きを共有するので，プログラム・テキスト上は，通常の順序動作システムと同一の形式を保つことができる．このため，システムの実現，変更時に並行性を意識する必要が少なくてすむ．また，順序動作システム向きに開発された変数の抽象化等の記述改善技術の導入が容易である．

しかしながら，現実には単一のユーザ・プロセスに関連する事象が同時に複数起こる可能性を考慮しておくことが必要である．システム・コールが発生して，それを処理するための手続き系が動作している最中にも，そのユーザ・プロセスに対するソフトウェア割込み，例外事象，あるいは，主記憶の不足によるスワップ・アウトの要求等の事象がシステム・コールの処理とは非同期に発生し，それらがどの手続きの動作中に起こるのかを限定することができない．そのため制御構造が複雑化し，このようなプロセス分割の方法でモジュラリティの向上を図ろうとする場合の障害となる．

上記のようなプロセス分割による OS の記述方式改善の最初の試みは，Hansen による Solo システムであると考えられる[7]．Solo は，Concurrent Pascal のクラス（class）およびモニタにより共有変数の抽象データ化を図り，モジュラリティの向上を狙っている．最近では，Pilot が，モジュール化支援機能およびモニタによる並行処理機能をもつ Mesa 言語を用いて同様の方法を試みている．そこでの非同期処理，とくにモニタ呼出しによるきわどい領域中での処理の中断は，以後の処理に支障をきたさないための特別な配慮が必要で，Pilot ではこのためにシステムの複雑さが増したことが報告されている[9]．

ユーザ・プロセス単位でのプロセス分割に対して，

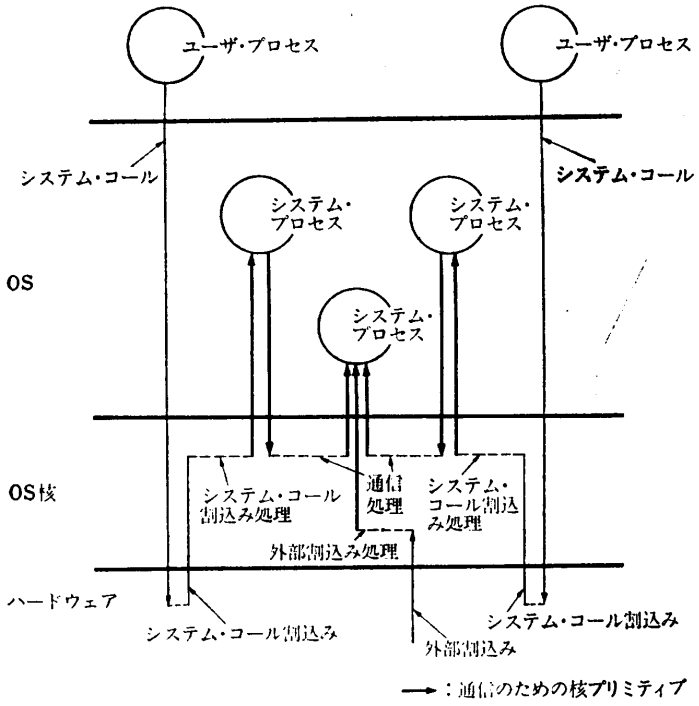


図 2 提案方式の実現法
Fig. 2 Implementation scheme of the proposed method.

他の分割方法として，記憶管理，ユーザ・プロセス管理，ファイル管理等の OS のおもな機能単位にプロセスを割り当てる方法も試みられている．この方法は，複数のプロセッサへの機能分散，あるいは，プロセッサごとの機能の変更が行いやすいために分散処理用 OS にしばしば採用され，最近では，Muss[5]，Medusa[10] 等がこの方法を用いているが，これらは，必ずしも OS の制御構造の明確化をおもな目的としたものではない．

（2） 提 案 方 式

これに対して，われわれは，以下の三つを基本方針として OS のプロセス分割を行う．

（i） プロセス分割を資源アクセスの相互排除性を単位として行う．

（ii） プロセス間は通信のみで結合する．

（iii） プロセス内の記述とプロセス間の記述は独立に行う．プロセス間の通信の記述には専用の言語を用いる．

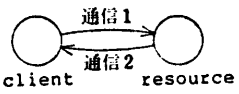
このような方針によって設計された OS は，**図 2** のような構造をもつ．各ユーザ・プロセスに対応したシステム・プロセスを設けてシステム・コールの受付けを行う．そして，相互排除アクセスされるおのおのの資源に対応してそれを管理するプロセスを設ける．

このとき，プロセス間を通信のみで結合することにより，プロセスを用いて変数の抽象化を行うとともに，OS の並行動作に関連した，相互排除，待合せ，前述の非同期的な動作等をすべて通信の実行によって実現する．すなわち，通信受付けの相互排除によって資源アクセスの相互排除を実現し，外部割込みも通信に変換してプロセスに伝達することにより入出力終了の待合せを実現する．また，通信待合せをつねに複数の要因に対して行うことにより，非同期的な動作を実現する．たとえば，入出力終了待合せでは，同時に入出力中断要求の通信をも待ち合わせることにより，入出力の中断を実現する．

提案方式の狙いは，第 1 に OS の制御構造の明確化である．相互排除アクセスされる資源を単位として OS 内部の資源を分類し，分類された資源ごとにその管理をプロセスで行うことにより，各プロセスの機能決定に根拠を与えることが可能になる．

提案方式を，従来のモニタを利用した方式と比較してみる．相互排除アクセスされる資源の管理をプロセスによって行えば，資源はプロセスの局所変数，資源へのアクセスは通信によって実現される．資源を管理するプロセスと資源を利用するプロセスが制御的に独立しているので，利用するプロセスは，資源へのアクセスを行っている最中でもそれを中断してソフトウェア割込みあるいは例外事象等を処理することができる．これに対して，相互排除アクセスされる資源の管理をモニタによって実現すると，モニタが制御的には資源を利用するプロセスの手続きとして動作するので，モニタとは論理的に無関係な利用者に関連する事象をもモニタが受け付けて利用者に伝達する必要が生じることになる．そのため，モニタによる資源管理では，プロセスを用いるのに比較して，処理すべき事象の数が増加し制御が複雑になりやすい．

提案方式の第 2 の狙いは，OS の記述法の改善である．モジュールとして記述される単位をプロセスに限定するとともに，モジュール間の参照関係を，一般に行われているようなモジュール内への名前の輸入 (import)，および，外への輸出 (export) によってではなく，参照関係のみをプロセス間通信として陽に記述する別個の言語を用いて表現する．これにより，システムの大域的な構造の把握が容易になると期待さ



(a) プロセス間関係の例

```
process {------------------------------ ① プロセス仕様宣言の始まり
        text    "client.o";----------- ② プロセス内部記述の存在するファイル名の指定
        call    request;-------------- ③ 通信呼出しエントリ名の宣言
        ent     get;------------------ ④ 通信受付けエントリ名の宣言
} client;------------------------------ ⑤ プロセス実体の定義
process {------------------------------ ⑥
        text    "resource.o";--------- ⑦
        call    ans;------------------ ⑧
        ent     alloc;---------------- ⑨
} resource;---------------------------- ⑩
example {------------------------------ ⑪ 通信の記述の始まり
        client.request > resource.alloc;--- ⑫ 通信1の記述
        resource.ans > client.get;-------- ⑬ 通信2の記述
}-------------------------------------- ⑭ 通信の記述の終り
```

(b) PNLによる(a)の記述

```
#call   REQUEST request ---------------- ① call マクロによる REQUEST エントリの定義
#ent    GET     get ------------------- ② ent マクロによる GET エントリの定義

process()------------------------------ ③ process 手続きの始め
{

-----
        call(REQUEST, ----);------------ ④ call プリミティブによる REQUEST エントリ
-----                                       に対する通信呼出し
        acc(GET);----------------------- ⑤ acc プリミティブによる GET エントリに対す
-----                                       る通信の受付け
}-------------------------------------- ⑥ process 手続きの終り
```

(c) client プロセスの内部記述

図 3 提案方式によるプロセス記述の例
Fig. 3 Example of process description by the proposed method.

れる.

われわれは，上記のプロセス間通信の記述を行う言語を設計・開発し，PNL(Process Network Language)と名づけた.

3. プロセス集合体による OS の設計

3.1 概　要

提案方式による OS は，図2に示すように，システム・プロセスの集合体によって実現される．システム・プロセス間は通信によって結合される．また，ユーザ・プロセスから発行されるシステム・コールおよび外部割込み等は，OS 核によって通信に変換され，システム・プロセスに伝達される．OS 核は，これらの機能を含めて，プロセスの集合体に対して動作環境を提供するための基本的な機能を果たすプログラム群である.

提案方式では，システム・プロセス間の通信方式は以下の二つの条件を満たしていることが必要である.

(i) 性能がよいこと．提案方式では，単一のシステム・コールの処理に対して複数のプロセスが関連するため，それらの間を結合する通信によるオーバヘッドが大きくなることが予想され，通信方式としては，性能がよいことがとくに重要である.

(ii) PNL では，通信の相手の指定および実行の順序をプロセス内部の記述とは別個に行っており，通信方式もこのような分割記述が可能な方式であること．すなわち，通信を受け付けるプロセスの数など，プロセス間の関係に関する指定を行わなくてもよいこと，発行した通信がいつ受け付けられるのかをプロセス外部から把握できること等が必要である.

従来提案された種々の通信方式のなかから，上記の条件を満たす通信方式として，ランデブ (rendezvous) を用いることにした.

また，記述方式としては，2章で述べたように，プロセス内部の記述とプロセス間参照の記述を別個に行う．プロセス内部は，C言語にランデブ実現のためのプリミティブを手続き呼出しの形でうめ込むことによ

り記述する．

提案方式による記述の例を**図3**に示す．図3(a)に示すような，資源要求プロセス client と，資源管理プロセス resource の間の関係を記述することを考えてみる．client は，通信1により resource に資源を要求し，得られた資源は通信2により client に渡される．図3(b)は，(a)の関係を PNL により記述したプログラムである．②～④は，client の外部仕様である．client の内部記述はファイル client.o に存在し，外部に対して通信を行う request エントリと，外部からの通信を受け付ける get エントリをもつことを宣言している．⑤において，以上のような仕様をもつプロセスの実体を一つ定義し，それを "client" と名づけている．同様に，⑥～⑩において，resource の宣言，定義を行っている．(a)の通信1，2は，⑪～⑭のように記述される．client の request エントリは resource の alloc エントリに，resource の ans エントリは client の get エントリに結合されることを明示し，これらの通信からなるシステムを "example" と名づけている．プロセス内部は，(c)に示すように，process 手続きを主手続きとする手続きの集合として記述される．ここでは，client 内部の記述を示している．PNL のなかで定義されたエントリ名をもとに，①では call マクロにより通信受付けエントリ REQUEST，②では ent マクロにより通信受付けエントリ GET が定義され，プリミティブの引数として用いられる．

このようにプロセス内部の記述とプロセス間の記述を分ける方法は，モジュールに対する外部名の輸入，輸出をモジュール内部で宣言する方法に対して，以下のような利点を有している．

(i) モジュール間参照関係の把握が容易になる．一般に，並行動作システムでは，機能の階層性は必ずしも参照関係の階層性には結び付かない．たとえば，図3(a)において，client の機能は resource に依存するが，resource の機能は client の機能に依存しない場合でも，参照関係の点からは対等である．このため，両プロセスは相互に名前を輸入する必要が生じ，参照関係の理解をむずかしくするとともに，プロセスの内部記述をライブラリによって管理することを困難にしている．モジュール間参照関係の記述をモジュール内部の記述と分けることにより，この点を改善することができる．

(ii) 通信の把握が容易になる．PNL による記述は，モジュール間参照関係の記述であると同時に，プロセス間通信の記述であり，動的構造の解析に有用である．たとえば，プログラミング時にデッドロックの可能性を判定するには，通信待ちにより停止しているプロセスに対して行われる通信をすべて把握する必要がある．これは，PNL によれば容易に行えるが，外部名の輸入，輸出による方法では容易でない．

提案方式による OSの実現は，新しい仕様の OS を設計することはせずに，米国ベル研究所で開発され，現在，とくに，研究教育機関に広く普及している UNIX[18] システムにその外部仕様を合わせた形で行うことにした．これは，第1に，既存システムとの比較により，提案方式の客観的な評価が可能になること，第2に，OS の外部環境をそのまま利用できる利点が大きいことによる．

本章では，OS 核の実現法，UNIX と同一の外部仕様を有する OS の提案方式による設計の結果について述べる．

3.2 OS 核の実現

OS 核は，通信によって結合されたプロセスの集合体として OS を実現するための環境を提供する．そのおもな機能は，

(i) プロセスの実現
(ii) プロセス間通信の実現
(iii) ユーザ・プロセスの実行の制御，および，システム・コールを通信に変換するユーザ・インタフェース機能
(iv) 外部割込みを通信に変換する機能

である．(ii)，(iii)は**表1**のような核プリミティブによって，システム・プロセスから関数呼出しの形で利用される．個々のプリミティブの機能については，以下必要に応じて述べる．

OS 核の実現に際しては，規模の縮小化，動作の高速化を基本方針とした．その実現は，DEC 社製の PDP-11/60 システム上で，UNIX を開発環境として行った．PDP-11/60 は，語長16ビットで，64 k バイトの論理アドレス空間を有する．ハードウェアのマッピング機構を用いて，物理アドレス空間上に複数個の

表1 核プリミティブ
Table 1 Kernel primitives.

通信に関するもの	call, acc, endr, calln
ユーザ・プロセス・インタフェースに関するもの	runu, haltu, timeu

論理アドレス空間が実現される．OS 核は，アセンブリ言語で約1kステップである．OS核，および，すべてのシステム・プロセスは，PNL 処理系により，同一論理アドレス空間上で動作するように結合される．

以下，OS 核の機能をより詳細に述べる．

（1） ユーザ・プロセス・インタフェース

2章で述べたように，各ユーザ・プロセスに1対1に対応してシステム・プロセスを設ける．これを，スーパバイザ・プロセスと名づける．スーパバイザ・プロセスは，システム・コールの受付け，ユーザ・プロセスへのプロセッサ割当てに関する時分割スケジューリング等を OS 核と協調して実現する．

スーパバイザ・プロセスと OS 核との間の情報のやりとりは，upcb (user process control block) を介して行う．upcb は，ユーザ・プロセス管理のためのデータ構造で，スケジューリングのためのリンク，状態フラグ，使用 CPU 時間の保持領域，状態退避領域等からなり，スーパバイザ・プロセスの局所変数として定義される．runu 核プリミティブにより，upcb をスケジューリングの優先度とともに OS 核に渡すことにより，ユーザ・プロセス実行の制御がなされる．一方，haltu 核プリミティブでは，upcb をスケジューリング・キューからはずし，スーパバイザ・プロセスに返却する．スーパバイザ・プロセスは，一定時間ごとに haltu を発行し，優先度を再計算する．runu, haltu を組み合わせることにより，時分割スケジューリングが実現される．また，優先度の再計算の際に必要な，ユーザ・プロセスの CPU 時間の計測は，timeu 核プリミティブにより行う．

システム・コール割込みは，OS 核によって通信に変換される．システム・コール割込みが発生すると，OS 核は，ユーザ・プロセスの状態を upcb に退避し，そのユーザ・プロセスを起動したスーパバイザ・プロセスに対する通信を生成し，upcb を通信データの形で渡す．

（2） 通信の実現

ランデブは，acc, call, endr, calln の四つの核プリミティブにより実現される．これらの核プリミティブは，通信に関する同期のみを実現し，データ転送は手続き呼出しと同一の機構により実現される．これにより，手続き呼出しと比較した通信のオーバヘッドは，プロセス切換えによるもののみとなる．

call を，相手プロセス名，エントリ名，および，通信データを引数として呼び出すことにより，ランデブ呼出しが実現される．通信のエントリ名は，1語中のビット位置，プロセス名は，そのプロセスを管理する pcb (process control block) の番地によって識別される．また，エントリ名を引数として acc を呼び出すことにより，ランデブの受付けが実現される．acc は，ランデブが成立すると，相手プロセスの call の引数列へのポインタを値として返すことにより，通信データの番地を知ることができる．データの返送，大量のデータ転送は番地呼出しによって実現する．ランデブは endr により終了する．calln は，ランデブがただちに成立しない場合そのまま復帰する点を除いて，call と同様の動作を行う．

（3） 外部割込み処理

個々の外部割込み要因に対応して，dcb (device control block) を用意する．OS 核は，外部割込み

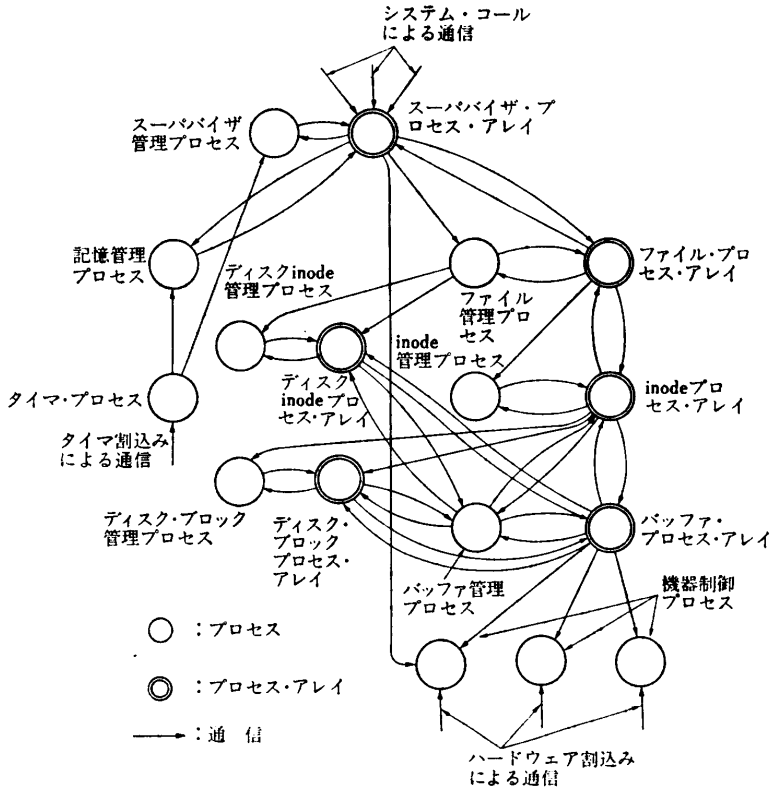


図 4 UNIX の再設計におけるプロセス配置

Fig. 4 Schematic diagram of process arrangement for reconstruction of the UNIX.

```
process {
        text    "supman.o";
        call    timer, -----;
        ent     time, ----;
} supman;                                   ① スーパバイザ管理プロセスの定義
process {
        text    "sup.o";
        call    maloc, mfree, saloc, sfree,
                mhalt, swapio, iodone, ----;
        ent     swapreq, time, mans, ----;
} sup[10];                                  ② スーパバイザ・プロセス・アレイの定義
process {
        text    "mem.o";
        call    ans, sreq;
        ent     mfree, sfree, maloc, saloc,
                uhalt, time, ----;
} mem;                                      ③ 記憶管理プロセスの定義
process {
        text    "time.o";
        call    suptime, memtime, ----;
        ent     ----;
} timer;                                    ④ タイマ・プロセスの定義
process {
        text    "rk.o";
        call    ans;
        ent     swap, ----;
} rk;                                       ⑤ ディスク機器制御プロセスの定義

unix {
/* exec */
        sup.maloc > mem.maloc;--------  ⑥ 記憶資源の要求
        mem.ans > sup.mans;-----------  ⑦ ⑥に対する返答
/* user halt */
        sup.mhalt > mem.uhalt;--------  ⑧ ユーザ・プロセスが長期間停止することの通知
/* swapping out */
        mem.sreq > sup.swapreq;-------  ⑨ スワップ・アウト要求
        sup.saloc > mem.saloc;--------  ⑩ スワップ領域の割当て
        sup.swapio > rk.swap;---------  ⑪ スワッピング入出力の要求
        rk.ans > sup.iodone;----------  ⑫ スワッピング入出力終了通知
        sup.mfree > mem.mfree;--------  ⑬ 主記憶領域の解放
/* swapping in */
        mem.ans > sup.mans;-----------  ⑭ 主記憶領域の割当て
        sup.swapio > rk.swap;---------  ⑮ スワッピング入出力の要求
        rk.ans > sup.iodone;----------  ⑯ スワッピング入出力終了通知
        sup.sfree > mem.sfree;--------  ⑰ スワップ領域の解放
/* time siren */
        timer.suptime > supman.time;--  ⑱ スーパバイザ管理プロセスへの通知
        supman.timer > sup.time;------  ⑲ 各スーパバイザ・プロセスへの通知
        timer.memtime > mem.time;-----  ⑳ 記憶管理プロセスへの通知
-----
}
```

図 5 ユーザ・プロセス・スケジューリングの実現法
Fig. 5 Implementation method of user process scheduling.

を，dcb 中に記入されたプロセスへの通信に変換する．

3.3 OS の設計

提案方式による OS 記述の有効性を確かめるために，UNIX システム（バージョン6）の再設計を試みた．

2章で従来方式として述べた OS の構造と同様に，UNIX は，ユーザ・プロセスに1対1に対応するプロセスによって実現されている．そのため，システム・コール以外の要因によって起動される，入出力処理，ユーザ・プロセスのスワッピングの実行，ソフトウェア割込み等の処理の制御構造が必ずしも明確ではない．また，UNIX では，データの抽象化の概念が導入されておらず，大域変数を介しての手続き間の結合がきわめて密である．提案方式を採用することにより，これらの点が改善されると期待される．

提案方式を最適に実現するためには，プロセス割当ての基準となる資源割当ての単位に注目した OS の内部設計を行うことが望ましいが，今回は，UNIX の資源アクセスの相互排除単位を踏襲し，それに対応し

てプロセスを割り当てることを試みた．このような方法によっても提案方式の実現可能性あるいは良否は十分に評価でき，かつ，実現の手間は大幅に減少する．

UNIX における資源アクセスの相互排除は，大域変数へのアクセスの相互排除によって実現されている．そこで，この点に注目し，かつ3．2節で述べたように，ユーザ・プロセス管理のために，各ユーザ・プロセス対応にスーパバイザ・プロセスを設けることにしてプロセスの配置を行った結果，**図4**に示すような構造による OS の記述が得られた．

スーパバイザ・プロセスは，ユーザ・プロセスの生成，ユーザ・プロセス単位のスケジューリング，システム・コールの解釈・実行を行うプロセスである．その割当ては実行時に行われるが，システム・プロセスの配置を固定としているので，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・アレイと名づけることにする．プロセス・アレイに属する各プロセスが参照する通信相手は，すべて同一である．また，実行時にプロセス・アレイに属するプロセスを識別する場合は，アレイ名にインデクスを付けて行う．スーパバイザ・プロセスのユーザ・プロセスへの割当ては，ユーザ・プロセス生成の要求に際して，スーパバイザ管理プロセスが決定する．

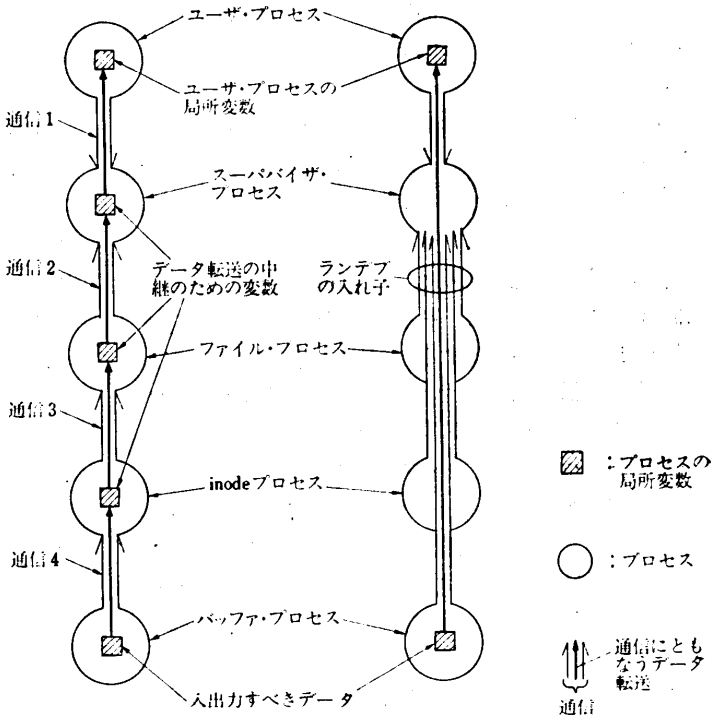
記憶管理プロセスは，主記憶および2次記憶上のスワップ領域を管理する．また，タイマ・プロセスは，タイマを管理し，スーパバイザ・プロセス，記憶管理プロセス等に時刻を通知する．それ以外のプロセスは，おもに，ファイル・システムに関係したものである．

以下，OS の主要な機能であるユーザ・プロセス・スケジューリング，および，ファイル・システムについて述べる．

（1） ユーザ・プロセス・スケジューリング

プロセッサおよび記憶資源の割当てにより，ユーザ・プロセスの実行を制御するユーザ・プロセス・スケジューリングの機能は，OS の最も重要な機能であり，OS 核，スーパバイザ管理プロセス，スーパバイザ・プロセス・アレイ，記憶管理プロセス，タイマ・プロセス，ディスク機器制御プロセスが関係する．これらの間の関係を**図5**に示す．①～⑤は，これらのプロセス，および，プロセス・アレイの定義である．

スーパバイザ・プロセスは，図5，⑥のように，maloc エントリを通じて記憶管理プロセスの maloc エントリに対して主記憶領域を要求し，⑦で割当てを受け，プログラムのロードを行い，upcb を初期化した後，runu プリミティブを用いてユーザ・プロセスを実行する．要求された主記憶領域の割当てができない場合には，記憶管理プロセスは，⑦において2次記憶上のスワッピング領域の番地をスーパバイザ・プロセスに通知する．スーパバイザ・プロセスは，ユーザ・プロセスのスワップ・アウトを実行し，領域の割当てを待つ．記憶管理プロセスは，このように，記憶資源が不足した場合，他のユーザ・プロセスのスワップ・アウトをスケジュールする．スーパバイザ・プロセスは，ユーザ・プロセスが端末へのアクセス等により長時間停止する場合には，⑧に示すように記憶管理プロセスにユーザ・プロセスのスワップ・アウト可能を通知する．記憶管理プロセスは，このようなユーザ・プロセス，あるいは，主記憶上に2秒以上存在し



（a）省略前のデータ転送法　　（b）省略後のデータ転送法

図6 ファイル入出力におけるデータ転送の省略法
Fig. 6 Omitting data transfer in the file system

ているユーザ・プロセスを管理しているスーパバイザ・プロセスに対して，⑨に示すように，スワップ・アウトの要求を行い，⑩〜⑬の過程を経て領域が解放されるのを待つ．その後，割当てが可能になった時点で，⑭〜⑰の過程により，主記憶領域の再割当て，ディスク機器制御プロセスによるスワッピング入出力の実行，等が行われる．スーパバイザ・プロセスによるユーザ・プロセスの優先度計算，および，記憶管理プロセスのスケジューリング時期は，⑱〜⑳の経路により，タイマ・プロセスから通知される．

（2） ファイル・システム

UNIX における大域変数へのアクセスの相互排除を基準にして，再設計後のファイル・システムを，ファイル，inode，バッファの各プロセス・アレイ，および，機器制御プロセスを主体として構成する．

ファイル入出力の相互排除を実現するために，オープンされたファイルのおのおのに，ファイル・プロセスを割り当てる．スーパバイザ・プロセスは，ファイル入出力の実行をファイル・プロセスに依頼する．

UNIX では，2次記憶上のファイルは，一定長のブロック単位に分割され，inode と呼ばれる管理テーブルによって管理されている．そこで，inode へのアクセスの相互排除を行うために，オープンされたファイルの inode のおのおのに inode プロセスを割り当てる．

また，UNIX における入出力バッファの大きさは 512 バイトであり，これが，相互排除アクセスの単位になっているので，バッファのおのおのをバッファ・プロセスによって実現する．

物理入出力実現のために，周辺機器対応に，機器制御プロセスを配置する．機器制御プロセスは，周辺機器の入出力を直接起動するとともに，入出力終了割込みによる通信を待つ．さらに，端末制御プロセスでは，端末からのクイット信号によるソフトウェア割込みをスーパバイザ・プロセスに通知する．

ユーザ→スーパバイザ→ファイル→inode→バッファの各プロセスの順に通信によって伝達された入出力要求の結果は，入出力の実行中，スーパバイザ・プロセスが停止しないように別経路により，上記の逆順に伝達される．

ファイル入出力を通信によって実現する場合の問題点は性能である．**図6**（a）に示すように，いまの場合には，通信1〜4に伴い，データ量の4倍の転送が必要になる．このオーバヘッドを防ぐために，（b）に示すように，ランデブを入れ子にして用いることにする．入れ子で用いることにより，入出力に関連する五つのプロセス間で同期が成立し，ユーザ，ファイル，inode，バッファの各プロセスが停止するので，スーパバイザ・プロセスはバッファ・プロセスからユーザ・プロセスへ直接データ転送を行うことが可能になり，データ転送の回数を1回に減少させることができる．このような方法を用いることにより，プロセス分割に伴うデータ転送量の増加の問題を解決することが可能である．

4. 考　察

提案方式によって UNIX システムを再設計し，現在，その開発を進めている．実現可能性についての見通しは得たが，その定性的，定量的な評価は，今後のシステム開発後の課題である．本章では，これまでの過程で気づいたいくつかの点について考察する．

（1） 提案方式の第1の目的は，OS の制御構造をわかりやすくすることである．再設計後のシステムでこの点がどのように達成されているかを考えてみる．UNIX は，通常の OS と同様，システム・コール割込みを制御の入り口とする手続きの集合として設計されているために，システム・コール以外の要因により起動される処理の実現が複雑である．記憶資源管理，物理入出力処理等がこの例である．すなわち，ユーザ・プロセスのスワップ・アウトは，システム・コールによる記憶資源要求以外にスワッピングを管理する専用プロセスが一定時間周期で起動されユーザ・プロセスの状態とは独立に行うために，スワップ・インの際の状態回復が複雑になっている．また，物理入出力待合せは，入出力を行いたいバッファをキューにつなぎ，入出力を起動し，終了を同期命令で待つことにより実現されている．その後，入出力終了の外部割込みにより起動された割込み処理手続きは，終了を待ち合わせているプロセスを起動し，次の入出力を連続して起動する．このように，入出力の起動，終了待合せ，割込み処理が制御的に分割されているために，入出力処理全体の動作の把握がむずかしい．

これに対して，再設計後の記憶管理の実現は，3.3節で述べたように記憶管理プロセスが集中的に行い，資源の要求，スワップ・アウトの通知のいずれも通信によって行われるので，スワッピングの実現に特別な制御あるいは状態退避を必要としない．また，物理入出力の処理に関しても，その起動，終了待合せ，およ

び，再起動を単一のプロセスで行うので，前記のような制御的な分離は存在しない．このように，資源単位にプロセスを割り付け，それらを通信のみで結合することにより，OS の制御構造が非常にわかりやすくなっている．

（2） 提案方式のいま一つの狙いとして，記述法の改善があるが，この点を，プロセスによる変数の抽象化および PNL による通信の陽な表現によって試みている．従来の UNIX の記述法と比較してみる．

UNIX は，C 言語で記述されており，その手続きは数個ずつ約 30 個のファイルに配置されている．コンパイル単位はファイルであり，これが，UNIX における記述単位（以下，モジュールと呼ぶことにする）であると考えることができる．Cの手続き名は，すべて外部名として定義され，他のモジュールから自由に参照できる．また，手続きは，引数以外に大域変数を介して結合されており，モジュール間の結合は強固である．

これに対して，提案方式では，プロセスがモジュールとなる．プロセスは複数のファイルにまたがって記述されることもあるが，各プロセスに属する手続きおよび変数のスコープはプロセス内に制限されている．

この点を，ファイル・システムを例にして，モジュールの外部参照の数を尺度として比較してみる．UNIX において，ファイル・システムは表 2（a）に示す 13 個のモジュールによって構成されている．これらのモジュールは，外部の手続き，変数を参照し，外部から参照される手続き名を定義している．モジュール 1 個当り平均で 35 個の名前を参照している．再設計後は，表 2（b）に示す 13 種類のプロセスによってファイル・システムが構成される．ここでは，通信相手のプロセス名，および，エントリ名が外部参照名となり，エントリ名が外部から参照される．これらの個数を各プロセスごとに求めると，プロセス 1 個当り平均で 5 個の名前の参照を行い，5 個の名前が参照されている．再設計の結果，各モジュールの外部参照の数が大幅に減少していることがわかる．

（3） 提案方式の問題は性能である．その評価は今後のシステム開発が必要であるが，ここで大雑把な推測を行ってみる．

プロセスを資源管理に用いた場合のオーバヘッドは，プロセスの切換え，通信データの転送，および，余分な相互排除によって生ずる．

ファイル・システムを例にして，プロセスの切換えによるオーバヘッドを推測する．1 回のファイル入出力の実現に 10〜18 回の通信が行われ，実測によれば，通信によるプロセス切換えのオーバヘッドは 450 μsec であった．そこで，ファイル入出力全体で 4.5〜8.1 msec のプロセス切換え時間を必要とすることになる．一方，UNIX のファイル入出力の平均処理時間が約 23 msec であり，このうち約 9 msec が CPU 時間であることが測定された．したがって，プロセス切換えのオーバヘッドはかなり大きな比重を占めている

表 2 再設計前後のモジュール間参照数の比較
Table 2 Number of external symbols of file system in the UNIX system and in the system by the proposed method.

（a） UNIX のモジュール間参照数

ファイル名	外部参照		外部被参照
	変数	手続き	手続き
alloc. c	26	15	7
bio. c	39	7	21
fio. c	26	5	9
iget. c	29	24	6
kl. c	13	7	7
nami. c	20	8	2
rdwri. c	19	13	6
rk. c	11	7	9
subr. c	13	5	6
sys 2. c	24	22	10
sys 3. c	30	12	6
sys 4. c	34	11	18
tty. c	30	13	11
平均	35		9

（b） 再設計後のモジュール間参照数

プロセス名	外部参照	外部被参照
	エントリ	エントリ
スーパバイザ	25	9
ファイル	16	12
ファイル管理	1	3
inode	12	8
inode 管理	3	2
バッファ	5	4
バッファ管理	4	5
機器制御（端末）	1	3
機器制御（ディスク）	1	3
ディスク inode	3	6
ディスク inode 管理	2	4
ディスク・ブロック	5	4
ディスク・ブロック管理	2	6
平均	5	5

ことがわかる．また，プロセスを用いて変数の抽象化を進めた結果，従来，変数の共有によって直接アクセスできた変数も通信によってアクセスしなければならなくなり，データ転送および余分な相互排除によるオーバヘッドが生ずる．この点に関して，大量のデータ転送では3.3節で述べた方法で，少量のものではプロセス間で同一の変数を多重にもつことで改善を試みている．

（4） 再設計によって生じたいま一つの問題点として，デッドロックの問題がある．提案方式による設計が，従来方式に比較してデッドロックの危険が増加する理由は，余分な相互排除，および，通信方式の特性によるものである．前者は，相互排除されていなかった変数へのアクセスが相互排除されたことにより生ずるデッドロックである．また，後者に関しては，ランデブは，いわゆる握手方式であり，通信が終了するまで送り手のプロセスも受け手のプロセスも停止するために，デッドロックの危険が生ずる．

現在，これらのデッドロックの危険性は，設計者の注意によって取り除いているが，PNL の記述から自動的にそれを検出する方法についても検討を進めている．

5. むすび

OS を資源管理機能の集合体とみなし，その並行性に注目した一つの設計方式を提案した．すなわち，相互排除アクセスされる資源対応にそれぞれ独立したプロセスを割り当て，プロセス間をすべて通信によって結合する．このような設計方式によって，OS の構造の明確化，記述法の改善を図ることができる．また，提案方式によって，実用 OS である UNIX システムを再設計した結果，および，それに伴う考察結果について述べた．

提案方式による OS の実現可能性の見通しを得たので，今後，その実現を行うことにより，性能に関する評価を行うとともに，OS の論理的性質を明らかにすることを目指す．

また，このような設計方式は，分散 OS，あるいは，OS 移植のための技術としても有効であると考えられ，それらに対する考察を行うことも今後の課題である．

参考文献

1) Campbell, R. H. and Habermann, A. N.: The Specification of Process Synchronization by Path Expressions, in Goos, G. and Hartmanis, J. (eds.), *Lecture Notes in Computer Science*, Vol. 16, pp. 89-102, Springer-Verlag, New York (1974).
2) Cheriton, D. R., Malcolm, M. A., Melen, L. S. and Sager, G. R.: Thoth, a Portable Real-Time Operating System, *CACM*, Vol. 22, No. 2, pp. 105-115 (1979).
3) Dijkstra, E. W.: The Structure of the "THE" Multiprogramming System, *CACM*, Vol. 11, No. 5, pp. 341-346 (1968).
4) DoD: プログラム言語 Ada 基準文法書，共立出版，東京 (1981).
5) Frank, G. K. and Theaker, C. J.: The Design of the Muss Operating Systems, *Softw. Pract. Exper.*, Vol. 9, pp. 599-620 (1979).
6) Hansen, P. B.: The Programming Language Concurrent Pascal, *IEEE Trans. Softw. Eng.*, Vol. 1, No. 2, pp. 190-207 (1975).
7) Hansen, P. B.: The Solo Operating System, *Softw. Pract. Exper.*, Vol. 6, pp. 141-205 (1976).
8) Hoare, C. A. R.: Monitors: An Operating System Structuring Concept, *CACM*, Vol. 17, No. 10, pp. 549-557 (1974).
9) Lampson, B. W. and Redell, D. D.: Experience with Processes and Monitors in Mesa, *CACM*, Vol. 23, No. 2, pp. 105-117 (1980).
10) Ovsterhout, J. K., Scelza, D. A. and Sindhu, P. S.: Medusa: An Experience in Distributed Operating System Structure, *CACM*, Vol. 23, No. 2, pp. 92-105 (1980).
11) Redell, D. D. et al.: Pilot An Operating System for a Personal Computer, *CACM*, Vol. 23, Nol. 2, pp. 81-92 (1980).
12) Richards, M. et al.: Tripos—A Portable Operating System for Mini-Computers, *Softw. Pract. Exper.*, Vol. 9, pp. 513-526 (1979).
13) Ritchie, D. M. and Thompson, K.: The UNIX Time-Sharing System, *CACM*, Vol. 17, No. 7, pp. 365-375 (1974).
14) 斎藤信男：並行処理言語，情報処理，Vol. 22, No. 6, pp. 531-534 (1981).
15) 田胡和哉，益田隆司：オペレーティング・システムの論理記述に関する考察，情報処理学会第23回全国大会講演論文集，pp. 281-282 (1981).
16) 田胡和哉，益田隆司：オペレーティング・システムの論理記述に関する考察（第二報），情報処理学会第24回全国大会講演論文集，pp. 179-180 (1981).
17) 田胡和哉，益田隆司：オペレーティング・システムの論理構造記述法，情報処理学会第25回全国大会講演論文集，pp. 219-220 (1982).
18) 田胡和哉，益田隆司：オペレーティング・システムの論理構造記述言語の設計，情報処理学会第26回全国大会講演論文集，pp. 341-342 (1983).
19) 和田英一：操作システムまわりの話題から，情報処理，Vol. 21, No. 5, pp. 566-573 (1980).

（昭和58年4月11日受付）
（昭和58年12月13日採録）