論　文

ALGOLあるいはFORTRAN内でコオペレーティング・プロセスを実現するためのオペレーティングシステムSOS*

土　居　範　久**

Abstract

The implementation of the operating system is described which realizes the cooperating process in ALGOL or FORTRAN environments. The nucleus of the system is highly multi-layered.

This is based upon both the abstraction corresponding to the resources and the primitivity of primitives. This approach is an attempt to make the system. structured The required scheduling mechanism is described. The implemented protection system is sketched. And, the handling of the stateword is outlined. Finally, an example of the application of the system is given.

1. はじめに

大学における計算機教育は，わが国においても専門の学科もいくつか設置され，着々と進展しつつある．なかでも，入門的なプログラミング教育は，たとえば，東京大学や慶応義塾大学ではカフェテリア方式を用い，ほぼ全学的に行なわれている．慶応義塾大学では情報科学研究所が5年前に設置され，入門的なプログラミング教育のみならず，数多くの情報科学に関する講座を開設し全学に門戸を開いている．

それらの講座の1つにオペレーティングシステムがある．ところが，オペレーティングシステムは，ここ数年ますます巨大化・複雑化が進み，実際，

（1） システムの規模があまりにも巨大であること
（2） 問題のとらえ方が時間の経過とともに変化すること
（3） 構造化が行なわれていないこと
（4） 効率に重点が置かれ過ぎていること

などから

（1） システムの作成が計画からひどく遅れる
（2） システムの信頼性が非常に欠ける
（3） システムの保守・改良が困難であり，変更に対して極度に敏感である
（4） あらかじめ（稼動しはじめてからでさえも）システムの動作を完全に予測することが困難である
（5） システムの作成には不合理なほど費用がかかる

など数多くの弊害が生じてきている．このように巨大化・複雑化されたオペレーティングシステムの本質をいかに短期間で教えるかについては，極めて問題がある．そこでわれわれはオペレーティングシステムの(部分的ではあるが)原則を一通り把握させることができ，かなりの程度のコオペレーティング・プロセスが実現でき，しかもそれらが実際に高級言語で扱えるシステムが教育上必要であるという見地から，ALGOLおよびFORTRAN内でこれらのことを可能にするオペレーティングシステムのひとつのモデルを作成した．

本論文はそのオペレーティングシステムSOSを記述したものである．このシステムの設計法は，基本的にはニュークリアス・エクステンション・アプローチ[1),2)]に基づいているが，従来までは漫然と一体化されていた機能を分離・明確化しようとするひとつの試みでもあり，システムの構造化のためのひとつのアプローチを試みてみたものである．

---

* An implementation of the operating system (SOS) realizing the cooperating process in ALGOL or FORTRAN, by Norihisa DOI (Keio Institute of Information Science, Keio University)

** 慶応義塾大学情報科学研究所

2. ニュークリアスの構造

ニュークリアス (nucleus) はプロセッサを抽象化 (論理化) するためのひとつの抽象化のレベルである[3]. 主な資源, 原始操作とともにニュークリアスの構造を図示すると **Fig. 1** のようになる. 何層にもレベル化された重層構造になっている. 図の横線がこのレベル化の区分を表わす. 各レベルは一組の関連した機能から構成されている. 各レベルの機能および資源のうちいくつかのものはより高いレベルからは引用できるが, より低いレベルからは引用できない.

レベル化には3種類あり, それは次のようにして行なわれている.

まず最初のレベル化は, 責務 (responsibility) に対応したレベル化である (レベル 0〜3). このレベル化はプロセッサの抽象化を支援するものであり, プロセッサの抽象化のレベル内の論理的な骨子を提供するものである. すなわち, これらのレベル化された責務によりプロセッサの抽象化がなされているわけである.

level	respon-sibility	abstraction	resources	primitives
0	process	process	stateword current-process	storestateword setstartpoint executeprocess
1	domain	basic system of process	process table noofprocess	createprocess
				sibling, relatives ascendant, daughter
				deleteprocess
				readdescendant readsibling
			domain	createobject
				inherent, grant
				checkdomain
2	scheduling	quasi virtual processor	timer	timer, hold
			readylist noofready-process noofactive-process	inreadylist outreadylist
				schedule
				suspend/release
				wakeup/block
				start/stop
3	synchronous operation	cooperating system		succ
				semaphore semaphorearray
				fork
				signal/wait
				await/cause
				chunkp/chunkv

Fig. 1 The structure of the nucleus

各責務をはたすためには, 一般に, 対応する資源とくにデータ・ベースが必要である. この意味で, このレベル化は資源に対応した抽象化であり, 抽象化の結果が責務である. スケジューリングのレベルを例にとると, timer, readylist, noofreadyprocess, noofactive-process, といった資源にもとづいてスケジューリングが行なわれ, これによりここまでのレベルで (レベル 0〜2) プロセッサの基本的な抽象化が行なわれたことになる. noofactiveprocess は ready ∧ unsuspended 状態[4]のプロセスの数であり, noofreadyprocess は ready ∧ unsuspended 状態あるいは ready ∧ suspended 状態にあるプロセスの数である (3. 参照).

2番目のレベル化は, 責務の抽象化のレベルを構成するこれらの資源をさらに細分化した機能的なレベル化である. 各レベルに属する資源は, そのレベルより高いレベルで引用することはできるが, 他のレベルから直接変更することは許されない. このレベル化は, 前段階までの抽象化の過程を明確にさせる働きをもつと同時に, これにより段階的な設計が可能になる.

3番目のレベル化は, 純粋に機能的なレベル化である. スケジューリングのレベルを再び例にとると, readylist, noofreadyprocess, noofactiveprocess といった資源に対応したレベルが, さらに機能的な関係によってレベル化されている. すなわち, inreadylist および outreadylist はこれらの資源を陽に加工する原始操作であるにの対し, schedule はこれらの資源を単に引用するだけの原始操作であるという機能的な相違があるのである. 同期操作のレベル (レベル3) 内のレベル化も主としてこの段階のレベル化である.

さらに, 原始操作 (primitives) および一般の操作は原始度の高いものをもとに, その上により原始度の低いものを組み立てるといった徹底した積み上げ方式を用いて作られている. ここで原始操作とは, あるレベルを構成する機能のうち, それよりも高いレベルで利用される機能のことである. 〔一般には, 汎用性の高いものに限って原始操作といい, これらの原始操作のための手続きの集合をニュークリアスと呼んでいる.〕 また, 原始度とはこの (原始) 操作の機能的 (および構造的) 規模のことで, 規模が小さいとき原始度が高いといい, 規模が大きいとき原始度が低いという. たとえば, fork は非同期操作を始動するための原始操作であるが, この操作は storestateword, createprocess, setstartpoint および start の上に組み立てられている (**Fig. 2**[8]). また, signal/wait は storestateword,

```
type  P=1..maximum number of process;
      D=..1maximum number of capabilities;
procedure fork
  (var i: P; pname: alpha; startpoint: addres; p: integer;
  d: array [D] of integer);
begin
  storestateword;
  i:=createprocess (pname, p);
  setstartpoint (startpoint, process[i]. stateword);
  start (i, d);
end
```

Fig. 2 primitive *fork*

```
procedure await (var s1,s2: array);
  begin
    storestateword;
    s1[s1[3]] :=currentprocess;
    s1[3] :=succ (s1[3], s1[2]);
    s1[1] :=s1[1]-1;
    suspend (currentprocess);
    signal (s2);
    schedule;
  end;
procedure cause (var s1,s2: array);
  begin
    while s1[1]<0 do
      begin
        s1[4] :=succ (s1[4],s1[2]);
        s2[s2[3]] :=s1[s1[4]];
        s2[3] :=succ (s2[3], s2[2]);
        s2[1] :=s2[1]-1;
        s1[1] :=s1[1]+1;
      end
    end
```

Fig. 3 primitives *await/cause*

primitives	primitives used to construct the primitive
schedule	executeprocess, setcurrentprocess, hold
suspend	storestateword, modifysuspended, ascendant, daughter, sibling, modifyactiveprocess, schedule
release	modifysuspended, ascendant, daughter, sibling, modifyactiveprocess
wakeup	setstate, setwws, ascendant, daughter, sibling, inreadylist
block	storestateword, setstate, setwws, daughter, outreadylist, schedule
start	storestateword, setstate, daughter, inreadylist, schedule
stop	storestateword, setstate, ascendant, outreadylist, schedule
fork	storestateword, setstartpoint, createprocess, start
signal	succ, release
wait	storestateword, suspend, succ

Fig. 4 The state of the pile of primitives

release/suspend, succ (および semaphore あるいは semaphorearray) の上に (付録参照), await/cause は signal/wait と対にして条件付きクリティカル・リージョンを設定するための原始操作であるが (6. 参照), storestateword, schedule, suspend, signal, succ (および semaphore) の上に組み立てられている (**Fig. 3**). 主な原始操作とその原始操作を実現するために用いている原始操作とを **Fig. 4** に示す.

このようにして, より原始度の低い操作を有機的に実現したこと, 資源が独占的に各レベルに属するようにしたこと, および極めて多層化された重層構造にしたことから, 開発・改良が極めて容易であった.

3. スケジューリング

2章で述べたニュークリアスの下にプロセスのシステムが形成される. システム内のプロセスのとりうる状態としては, dead, dormant, ready, blocked, running の5状態があり, 状態遷移と原始操作の関係を図示すると **Fig. 5** のようになる. ただし, ready および blocked 状態にはそれぞれ suspended 状態と unsuspended 状態とがある. この間の状態遷移を操作およびシステムの状態変数と関連づけて厳密に記述すると **Fig. 6** のようになる. 実際には running 状態は ready 状態の一部とみなしているので, wakeup/block 以外の同期操作では, ready∧unsuspended と ready∧suspended との間を行き来することになる. readylist では suspended, unsuspended にかかわらず ready 状態のプロセスを管理している. readylist 内では, プロセスは優先度順に並べられている. そこでニュークリアスで行なう短期のスケジューリングとしては ready∧unsuspended 状態にあるプロセスの内で最も優先度が高いプロセスを選び, そのプロセスにプロセッサを割り当てるようにしている. こうすることによって, すべての同期操作の共存をはかっているのである.

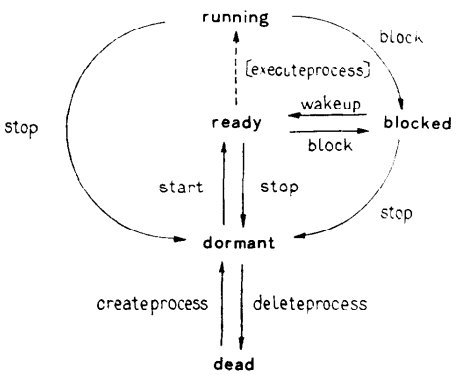


Fig. 5 The states of processes and the primitives that cause the transitions between them

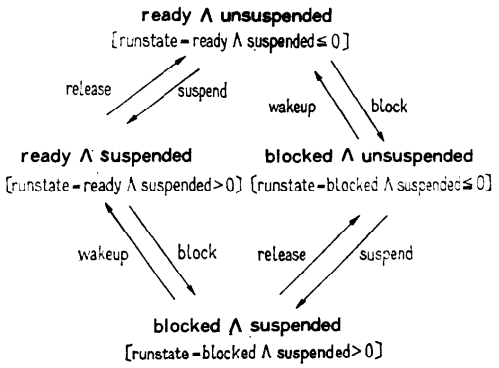


Fig. 6 The relation between ready/blocked, suspended/unsuspended, and the primitives.

ところで，SOS は言語プロセッサ内で擬似並行処理を実現するものであるから，実際にはシステム全体が1つのプロセス〔UNIVAC 1106 EXEC-8 ではアクティビティ〕として処理される．そこで，readylist にひとつもプロセスが登録されていないとき，および登録されていてもそれらがすべて ready∧suspended 状態にあるときには，無限ループに入ってしまいオペレーティングシステムとしての機能をはたさなくなる．したがって，そのような場合には，時間およびプロセスを指定しておくと，指定された時間だけ経過したら強制的に指定されたプロセスにプロセッサを割り当てるようにしてある．〔厳密にいうと，このときには指定された時間だけ SOS 自体が自発的にプロセッサを放棄するのである．そのための原始操作が hold である．この結果が SOS で扱ういわば唯一の外部割り込みである．〕

4. 保 護

SOS では，現在のところ，プロセス，手続き（ニュークリアで定義されている原始操作を含む）の引用，および番地空間を保護の対象とし，定義域の概念[5]を応用している．

まず，プロセス間の支配権および通信権はプロセスの階層関係を保持しているリストで管理される．プロセスを削除できるものは，その親プロセスに限られ，通信権はそのプロセスの兄弟および直系の先祖に与えられる．これらの権利の有無をチェックするための原始操作が sibling, relatives, ascendant および daughter である．

プロセスの定義域には，さらに，手続きの引用の可否が含まれる．プロセスに潜在的な手続きの引用の可否に関する資格は，原始操作 inherent によって定義できる．〔本質的には，手続きの引用に限らず資源の使用に関する潜在的な資格の決定は，記述言語の言語プロセッサにゆだねられるべきものであり，それらが手続きの定義域のうちの潜在的な資格を構成するものと考えることができる．〕ニュークリアスの原始操作は資格の対象となる．また，createobject によって，使用者プログラムで定義した手続きを資格の対象として定義できる．その場合の所有権は定義したプロセスにある．ある対象の所有権をもつプロセスは，その対象の使用権を他のプロセスに与えることができる (grant).

親プロセスは，子プロセスの実行開始時に自分のもっている権利以内の資格を，子プロセスに資格として渡すことができる〔start のパラメタとして渡す〕．createobject によって定義した手続きでは，check-domain を用いて，その使用権をチェックできる．

番地空間の引用の可否に関して，手続きが潜在的にもつ資格の定義は，言語の性質をそのまま利用している．したがって，これに対する手続きの定義域は，その手続きの実行中にのみ有効であり，その手続きの実行が終了すると，その手続きを呼び出した手続きの定義域に戻る．

5. ステートワードの処理

start, stop あるいは wait といった他のプロセスに制御が移る可能性のある原始操作を引用する際には，引用時点のステートワード(stateword)[4]を退避しておく必要がある．そのために，各種中央レジスタ〔SOS の場合には，アキュムレータ 15 個，インデクスレジスタ 12 個，および R レジスタ 15 個．サブルーチン呼び出しの慣習により，インデクス 11 がプログラムカウンタに相当する．〕を退避する手続きをアセンブラで用意し，コレクション時にそれら原始操作に対する手続きの入口点を，この退避用の手続きを呼び出す命令で置き換えると同時に，退避用の手続きの入口点に原始操作の入口点の命令を挿入している．この退避用の手続きでは，一時的な作業用領域にそのときのステートワードを格納する．この作業用領域から所定の場所への転記は，原始操作 storestateword で行なう．

プロセスの再開に際しては，やはりアセンブラで組んだ原始操作 executeprocess でステートワードを回復する．

こうすることによって，プロセスの切り換えが可能になる．

6. 応用例

SOSでは，同じコードを異なるプロセスとして同時に走らせることは本質的にできないが，そのような必要性が生じたときには，そのコードを必要とするプロセス分だけ用意すれば一般に済みそうである．そこでプログラム自体は多少長めになるが，かなりのことが行なえると考えられる．

SOSの応用例として，The Five Dining Philosophers[6] のシミュレーション用プログラム (FORTRAN版) の一部を **Fig. 7** および **Fig. 8** に示す．

Fig. 7 が賢者1に対するアルゴリズムである[7]．MODIFY は両隣りの賢者の使用可能なフォークの数を修正するための手続きである．TT および TTT がスパゲッティを食べ始めた時刻，および食べ終る時刻である．

Fig. 8 が賢者プロセスを生成，始動してから，タイムテーブルを用いて賢者プロセスを管理するためのスケジューラ（プロセスの1つ）の部分である．スケジューラと各賢者プロセスとはセマフォ (semaphore) MASTER および配列セマフォ SEM を用いて同期をとらせている．すべての賢者が食事中か思索中のときには，このスケジューラで食事か思索かが一番はやく終る時刻までループするか，あるいは，その時刻までプロセッサを放棄するかのいずれかになる．後者の場合，その時刻になったとき，強制的にプロセッサが割り当たえられるプロセスは，このスケジューラである．

```
C
C
C     KENJA=1
C
C
   10 CONTINUE
      T  = RANDU(IV)*10.
      IF( T  .EQ. 0 ) GO TO 10
      CALL SIGNAL(MASTER)
      P = P1
      CALL WAIT(SEM(1,P1))
      CALL WAIT(FSEM)
   12 IF( FORKS(1) .EQ. 2 ) GO TO 11
       CALL AWAIT(EVENT,FSEM)
       GO TO 12
   11  CONTINUE
       CALL MODIFY(1,-1)
       CALL SIGNAL(FSEM)
C
   13 CONTINUE
      T = RANDU(IV)*10.
      IF( T .EQ. 0 ) GO TO 13
      CALL TIMER(TT)
      TTT = TT+T*100
      WRITE(6,100) TT,TTT
  100 FORMAT(/' * 1 *',I15,I15/)
      CALL SIGNAL(MASTER)
      P = P1
      CALL WAIT(SEM(1,P1))
C
      CALL WAIT(FSEM)
      CALL MODIFY(1,1)
      CALL CAUSE(EVENT,FSEM)
      CALL SIGNAL(FSEM)
C
      GO TO 10
C
C
```

Fig. 7 The algorithm for the phylosopher 1.

```
C
C
C
      CALL FORK(P1,'KENJA1',$10,3,D)
      CALL FORK(P2,'KENJA2',$20,3,D)
      CALL FORK(P3,'KENJA3',$30,3,D)
      CALL FORK(P4,'KENJA4',$40,3,D)
      CALL FORK(P5,'KENJA5',$50,3,D)
C
C
C     SCHEDULER
C
C
      DO 7 K = 1,10
       TIMETL(K) = 2**34
       MARK(K) = .FALSE.
    7  CONTINUE
    5 CONTINUE
      CALL WAIT(MASTER)
C
      IF( .NOT. ENFLAG ) GO TO 1002
       CALL SIGNAL(MASTER)
       ENFLAG = .FALSE.
       GO TO 1000
 1002 CONTINUE
C
      CALL TIMER(TT)
      TIMETL(P) = TT+T*100
      MARK(P) = .TRUE.
C
      SW = .FALSE.
    6 CONTINUE
      J = 3
    4 CONTINUE
      IF( J .GT. 7 ) GO TO 1
       IF( .NOT. MARK(J) ) GO TO 2
        IF( TT .LT. TIMETL(J) ) GO TO 3
         MARK(J) = .FALSE.
         CALL SIGNAL(SEM(1,J))
         SW = .TRUE.
    3    CONTINUE
    2   CONTINUE
        J = J+1
        GO TO 4
    1  CONTINUE
       IF( (READYP .EQ. 1) .AND. (.NOT. SW) ) GO TO 1000
        ENTIME = 2**34
        DO 1001 I = 3,7
        IF( .NOT. MARK(I) ) GO TO 1001
         IF( ENTIME .LE. TIMETL(I) ) GO TO 1001
          ENTIME = TIMETL(I)
 1001     CONTINUE
          CALL TIMER(TT)
          ENTIME = ENTIME-TT
          GO TO 5
 1000   CONTINUE
        CALL TIMER(TT)
        SW = .FALSE.
        GO TO 6
C
```

Fig. 8 The scheduler for the phylosophers' system.

Fig. 9（次頁参照）に実行結果の一部を示す．＊印に囲まれた番号が食事を始めた賢者を表わし，その後の2つの値がそれぞれ TT および TTT である．

7. OS機能の組込みに関する問題

OS機能は，本来ならばプログラミング言語の機能として，一般には予約語的な要素として言語そのものに含まれるべきものである．しかし，われわれは言語プロセッサには一切手を加えず，その上にしかもその言語で組立てようとしているので，OS機能の組込み方には自ずと制限がついてくる．できる限り本来の姿に近いものを実現する必要があるが，待ち行列の処理等で完全に同じものを作ることは不可能である．その

```
* 2 *      6297966A      62979968
                                  * 5 *      62979779      62980479
                        * 3 *      62979995      62980195
* 2 *      62980216      62980516
                                  * 4 *      62980507      62981307
* 1 *      62980541      62980641
* 2 *      62980745      62981645
```

Fig. 9 The result of the simulation of the dining phylosophers' system.

ために考慮しなければならない主な問題としては2つある．第1は引用形である．ALGOL/FORTRAN を対象とした場合，実現方法としては，機能的には手続き（副プログラム）化する以外にない．ALGOL の場合には，手続き化することにより，利用する場合には，あたかも言語自体に備わっている予約語的にすることができる．FORTRAN では，これに対応させ，CALL 文を用いて引用するようにした．第2は機能の実現方法そのものに関する問題で，使い易さに影響を与える．しかしOS機能をプログラミング言語の上に実現しようとすると，本来オペレーティングシステムが暗黙のうちに行なうべきことを，具体的に指示するようにせざるを得ない．われわれは実現方法を単純化する一方，組込もうとしている機能を本来の姿にできる限り近づけることに努力した．しかし，幾つかの原始操作では潜在的な機能が表面にでている．たとえば semaphore は宣言詞であるが，ALGOL/FORTRAN の上に組立てるためには単なる宣言詞として扱うわけにはいかない．関連した待ち行列と密接な関係がある．われわれはセマフォを単なる整数型の変数ではなく，待ち行列までも含む配列として実現した．したがって使用者は待ち行列の大きさまで考慮し，整数型の配列宣言を行なう必要がある．semaphore はその待ち行列を初期化するための命令として実現している（付録参照）．同様のことが await/cause などについてもいえる．しかし，本質的な機能には変わりがない．

8. ALGOL/FORTRAN で OS を実現する上での問題点

ALGOL/FORTRAN 内でオペレーティングシステムを実現する上での一番の問題点はスケジューラである．前述した通り SOS 自身が1つのプロセスとして処理されている関係上，SOS 内のプロセスの状態如何によっては機能しなくなる恐れがある．そこで何らかの逃げ手を工夫する必要がある（3. 参照）．次にALGOL の場合にはプロセス切換えに関して問題がある．1つはすべてのOS機能を手続き化したことに起因する問題である．プロセス切換えの可能性のある原始操作を引用すると，場合によって，その中からスケジューラを引用する．その場合には切換えの有無によらず，スケジューラから executeprocess を経て強引にプロセスに制御を移す．ALGOL の手続きは再帰呼出しに耐えるよう設計されているので，そのために次第にスタックが成長する．このための領域が，場合によってはかなりの部分を占める可能性がある．第2にプロセスの切換えに際し，各種レジスタを退避回復するが，UNIVAC ALGOL (SIMULA) は最も外側で宣言された最も非局所的なデータ領域に関しても実行時にそのポインタを動的に割付けるので，レジスタの退避回復だけでは済まない可能性が残っている．

しかし番地空間の引用の可否に関して，手続きが潜在的にもつ資格の定義は，ブロック構造(および **own**)が都合がよい．しかし，そのブロックにとって非局所的なデータの引用を禁止する手段がない．この点に関しては，FORTRAN 版では名前付き共通ブロックを用いて逃げているが，定義・未定義の問題で JIS に違反している．

9. ま と め

ALGOL あるいは FORTRAN といった既存の並行処理が不可能な言語でコオペレーティング・プロセスを実現させるための方法，および，そのために必要

	No. of FORTRAN statements*	Instruction code (words)	Data (words)
level 0		227	
level 1	149	477	187
level 2	158	494	170
level 3	119	531	142
dat base			1,904
miscellaneous**		3,702	2,156
total	426	5,431	6,715

(a) The summary of the SOS

	No. of FORTRAN statements*	Instruction code (words)	Date (words)
semaphore	10	30	12
signal	18	68	27
wait	19	71	25
await	9	56	14
cause	13	73	18

(b) The size of the typical primitives.

* FORTRAN program part (including debugging aids)
** FORTRAN libraries

Fig. 10

とするオペレーティングシステムについて具体的に述べた．このようなオペレーティングシステムを作成することにより，最新の原理を多少の無理はあるが簡便に実施してみることが可能であると同時に，教材としても十分に役に立つと思われる．

また，オペレーティングシステムの設計に関しては，システムを構造化するためのひとつのアプローチを試みた．この方法は，オーバーヘッド等に多少の難点はあるが，システムの理解・設計・開発・保守等が容易である〔記述言語が高級言語であることにもよる〕．

Fig. 10 に SOS (FORTRAN 版) の規模の概要を示すので参考にされたい．

この計画の実施にあたっては，情報科学研究所の大野義夫，山本喜一の両氏および日本ユニバック株式会社の長谷川光邦氏にいろいろお世話になった．また日頃ご指導いただいている浦昭二教授ならびに林喜男教授に深く感謝する次第である．最後に本稿をまとめるにあたって適切なご指示を頂いた査読者に御礼を申し上げる．

参 考 文 献

1) Brinch Hansen, P.: "The Nucleus of a Multiprogramming system," Comm. ACM, Vol. 13, No. 4, pp. 238～241, 250 (1970).
2) 高橋秀俊・亀田寿夫: "オペレーティングシステムの一構成法", 情報処理, Vol. 11, No. 1, pp. 20～31 (1970).
3) Dijkstra, E. W.: "The Structure of the "THE"-Multiprogramming System," Comm. ACM, Vol. 11, No. 5, pp. 341～346 (1968).
4) Lampson, B. W.: "A Scheduling Philosophy for Multiprocessing System," Comm. ACM, Vol. 11, No. 5, pp. 347～360 (1968).
5) Lampson, B. W.: "Dynamic Protection Structures," Proc. of AFIPS FJCC, pp. 27～38 (1969).
6) Dijkstra, E. W.: "Hierarchical Ordering of Sequential Processes," Acta Informatica, Vol. 1, No. 2, pp. 115～138 (1971).
7) 土居範久: "ナノピコ教室解答", **bit**, Vol. 6, No. 4, pp. 58～61, (1974).
8) Wirth, N.: "The Programming Language Pascal," Acta Informatica, Vol. 1, pp. 35～63 (1971).

付 録

セマフォおよび signal/wait の実現方法

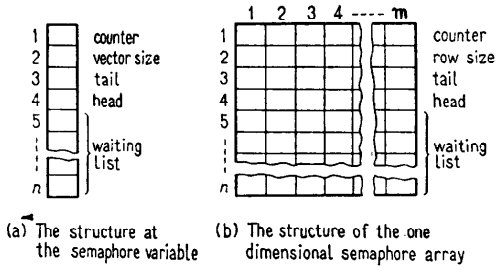
セマフォとしては，変数と1次元配列を許している．本来，セマフォは整数を値としてもつカウンタであるが，それに関連した待ち行列をもつ．そこで，SOS のセマフォの構造は **Fig. 11** (次頁参照) のようになっている．

プログラムでセマフォを使用する場合には，対応する待ち行列までも含めて配列の寸法を定め，整数型の配列として宣言すると同時に，原始操作 semaphore あるいは semaphorearray でセマフォであることを宣言する．

semaphore および semaphorearray は待ち行列の初期化をするための原始操作である．たとえば，原始操作 semaphore の機能は次のとおりである．

```
procedure semaphore (var s: vector; n: integer);
  begin
    if n≦4
    then semaphore violation
    else begin
           s[2]:=n;
           s[3]:=5
           s[4]:=n;
         end
  end
```

待ち行列はサーキュラ・リストとして使用する．2つのポインタ tail (s[3]) と head ([s4]) を用い，tail で最初の空きフレームを指し，head で待ち行列の先頭の1つ前のフレームを指す．このセマフォに対する



(a) The structure at the semaphore variable (b) The structure of the one dimensional semaphore array

Fig. 11 The structure of the semaphore

原始操作 signal/wait の機能およびそれらで用いる原始操作 succ の機能は次のとおりである.

```
procedure signal (var s: vactor);
  begin
    s[1]:=s[1]+1;
    if s[1]≦0
      then begin
             s[4]:=succ (s[4], s[2]);
             release (s[s[4]]);
           end
  end;
procedure wait (var s: vector);
  begin
    storestateword;
    s[1]:=s[1]−1;
    if s[1]<0
      then begin
             s[s[3]]:=currentprocess;
             s[3]:=succ (s[3], s[2]);
             suspend (currentprocess);
             end
  end;
function succ
(pointer: integer; vecsize: integer): integer;
  begin
    if pointer=vecsize
      then succ:=5
      else succ:= pointer+1;
  end
```

await/cause で用いるイベント待ち行列もこのセマフォを転用しており，構造は全く同じである．したがってイベント待ち行列も semaphore を用いて宣言する必要がある.

(昭和49年3月14日受付)
(昭和49年8月10日再受付)