≪論　文≫

同期基本命令の実現について*

斎　藤　信　男**

Abstract

The synchronization primitives play an important role in an asychronous control program system, and the problems in the implementation of these primitives are discussed.

In the implementation procedure of these primitives, several operations should be indivisibly executed. The author investigated which part of the procedure must be considered as an indivisible operation. The solutions for the typical synchronization problems are given considering this investigation, and the correctness of these solutions is proved.

The realization of indivisible operations is also an important problem. In order to realize these operations, it is necessary to prepare several hardware mechanisms corresponding to the system organization, and the problem in the usage of these mechanisms are also discussed.

1. まえがき

複数個の逐次型プログラムが同時に並行処理を行う非同期処理プログラム (asynchronous control program) に於ては，プログラム間の相互連絡を行う為に同期基本命令 (synchronization primitives) が用いられ，タイミングの同期がはかられる．非同期処理プログラムの書き易さは，提供されている同期基本命令の集合によって大きく左右される．入出力コマンドや，多重タスク・システムなどに関連して，従来からそれぞれの OS に固有の同期基本命令が提供されていたが，Dijkstra[1] によってセマフォア・システム (semaphore system) が提案され，また，その拡張も試みられ[2)~4)]，統一的な議論が可能となった．

同期基本命令を具体的にある OS のもとで実現する場合には，一般に，変数への代入操作，条件分岐操作を組み合わせてそれらの処理手続きを作る．その際，このような処理手続きは，一度開始されたら外部からの割り込みなどによってその実行が途中で中断されることのないような操作——非可分操作 (indivisible operation) として実行される事が必要となる．

非可分操作に関する１つの問題は，同期基本命令の処理手続きが初めから終り迄必らず非可分でなければならないかという点である．多重プログラミング・システムに於て，非可分操作の継続時間が大きいと，その操作が頻繁に行なわれる場合，その操作に対する待ち行列が増大してシステムの能率が低下し，それによって引き起される非能率性は，多重プロセサ・システムに於て特に問題となる．この論文では，非同期処理プログラムにあらわれる典型的な同期問題 (synchronization problem) に関して，非可分操作がどこ迄必要であるかを検討した．

もう１つの問題は，非可分操作を具体的に実現する方法である．非可分操作は，相互排除問題として記述できるので，ある種の同期基本命令を用いて実現することができる．しかし，それらの同期基本命令を実現するためには，それらの実現を支える低レベルのシステム—たとえば，オペレーティング・システムの核や，ハードウェア・システム—に於て，やはり非可分操作が実現できるような手段がなければならないことになる．この論文では，そのような低レベルでの手段について概括し，特に将来一般化すると思われる多重プロセサ・システムに於ける問題を検討した．

2. 同期基本命令の一般形

従来から，いろいろの計算機システムに於て，幾つかの同期基本命令系が提案されており，たとえば，block-wakeup[5], lock-unlock[6], P-V[1] などがある．これらに共通な性質を抽出すると，次のような一般形と

* On the Implementation of Synchronization Primitives, by Nobuo SAITO (Institute of Informatics, University of Tsukuba)

** 筑波大学電子・情報工学系

してまとめられる.

(1) **同期変数** (Synchronization Variable)

$x=(x_1, x_2, \cdots, x_n)$

同期の状態などの情報を保持しておくためのグローバルな変数で，一般に，n 個のスカラー要素から成るベクトル変数を考える.

(2) **消費操作** (Consume Operation)

同期の状態がある条件を満すのを待ち，その条件が成立したら，必要な状態に遷移する操作である．ある状態を使ってしまうという意味で，消費操作と呼ぶが，block, lock, P などがこの操作に相当する．消費操作の処理の詳細を，**Fig. 1** に示す．ここで，$C(\underset{\sim}{x})$ は，**通過条件** (pass condition) と呼び，消費操作が成功するために成立すべき条件を示す．また，関数 $D: \underset{\sim}{x} \rightarrow \underset{\sim}{x}$ は，**下降関数** (down function) と呼び，一般に，$\underset{\sim}{x}$ に関する単調減少関数である．また，一般に，通過条件に対する判定分岐操作と，下降関数を用いた代入操作 (Fig. 1 のフロー・チャートで破線で囲んだ部分) は，非可分操作として実現しなければならない.

(3) **生産操作** (Produce Operation)

これは，消費操作と逆の命令で，同期の状態をある状態から他の状態へ強制的に遷移させる．一般には，消費操作に於ける通過条件の値が偽となる状態から，真となる状態への遷移をひき起す．ある状態を作り出すという意味で，生産操作という．wakeup, unlock, V などがこの操作に相当する．生産操作の処理の詳細

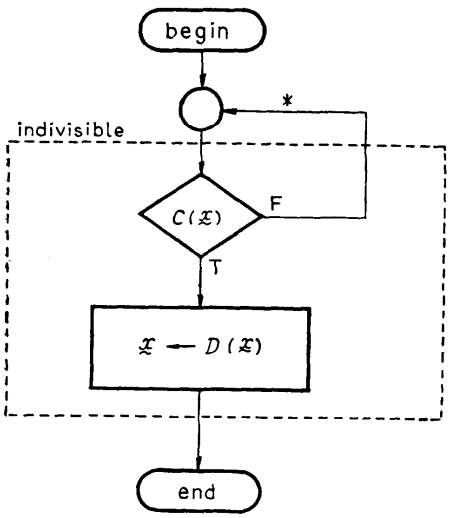

Fig. 1 the detailed flow of consume operation

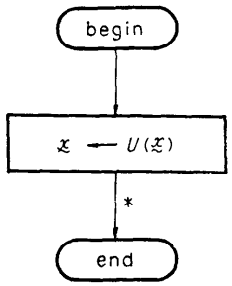

Fig. 2 the detailed flow of produce operation

を，**Fig. 2** に示す．関数 $U: \underset{\sim}{x} \rightarrow \underset{\sim}{x}$ は，**上昇関数** (up function) と呼び，一般に $\underset{\sim}{x}$ 関する単調増加関数である.

なお，消費操作に於て通過条件が偽の場合，先頭まで戻って通過条件の値が真になる迄同じ処理を繰り返している．このような実現法を，ビジー待ち (busy waiting) 方式と呼ぶが，プロセサの使用能率などから見れば，ビジー待ち方式は好ましくない．一般には，Fig. 1 の＊印に於て，下のレベルで用意されている同期基本命令を用いて，プロセサの放棄を宣言し，他のタスクに制御を移す．また，Fig. 2 の＊印に於ては，やはり，下のレベルで用意されている同期基本命令を用いてプロセサ使用可能という状態に戻しておかなければならない．一方，同期基本命令の形式的意味を議論する観点から見ると，ビジー待ち方式の方が扱い易いので，以下ではビジー待ち方式を用いて議論している．従来から提案されている種々の同期基本命令を，上述した一般形としてまとめ，**Tadle. 1** (次頁参照) に示す.

3. 非可分操作の必要性

同期基本命令は，前節に述べた一般形として実現することができるが，消費操作に於ては，Fig. 1 に示したように，通過条件の判定分岐と，下降関数を用いた代入操作とを，非可分操作として実現することが一般的に要請される．そこで，このように非可分操作を用いることは全ての同期問題に於て必要かどうか考えてみるために，典型的な 2 つの問題に関して検討してみる.

3.1 相互排除問題

非同期処理プログラムに於ては，共有カウンタなどは，同時に複数個のプロセスがアクセスすると，その値の正当性が保証できなくなる．このようなプログラムの部分を，危険部分 (critical section) というが危

Table. 1 synchronization primitives in general form

同期基本命令システム	同期変数 $\underline{x}$	消費操作			生産操作	
			通過条件 $C(\underline{x})$	下降関数による代入操作 $\underline{x}\leftarrow D(\underline{x})$		上昇関数による代入操作 $\underline{x}\leftarrow U(\underline{x})$
標準セマフォア・システム[1]	S	$P(S)$	$S>0$	$S\leftarrow(S-1)$	$V(S)$	$S\leftarrow(S+1)$
PV Multiple[3]	$S_1, S_2, \cdots, S_n$	$P(S_1, S_2, \cdots, S_n)$	$\forall i\ S_i>0$ $(1\leqq i\leqq n)$	$\forall j\ S_j\leftarrow(S_j-1)$ $(1\leqq j\leqq n)$	$V(S_1, S_2, \cdots, S_n)$	$\forall j\ S_j\leftarrow(S_j+1)$ $(1\leqq j\leqq n)$
PV Chunk[3]	S	$P(s, n)$	$S\geqq n$	$S\leftarrow(S-n)$	$V(S, n)$	$S\leftarrow(S+n)$
up/down システム[*4]	$S_1, S_2, \cdots, S_n$	down (S_i)	$\sum_{i=1}^{n} S_i\geqq 0$**	$S_i\leftarrow(S_i-1)$	up(S_i)	$S_i\leftarrow(S_i+1)$
wakeup/block[5]	WWS (wakeup-waiting switch)	block	WWS=ON	WWS←OFF	wake up	WWS←ON
lock/unlock[6]	interlock key	lock	interlock key =OFF	interlock key ←ON	unlock	interlock key ←OFF

* 通過条件による判定分岐と，下降関数による代入操作とは独立に行ってもよい．
** これは，semaphore application $\{S_1: S_2: \cdots: S_n\}$ に対する通過条件である．

険部分に対しては，同時には，唯一つのプロセスだけにしかアクセスを許さないという管理をしなくてはならない．このような管理の制御問題を，**相互排除問題** (mutual exclusion problem)[1]という．

ここでは，プロセス2個から成る非同期処理プログラム $P=(p1, p2)$ に対する相互排除問題を考える．**Fig. 3** に示すように，各プロセスは，危険部分 CS に入ることを許可して貰う Enter 命令，その使用の放棄を示す Exit 命令によって危険部分の前後を囲み，残りの仕事Rを終えたのちに，再び危険部分の実行を繰り返すと仮定する．このとき，相互排除問題は，次のように定義される．なお，この定義にあらわれる状態遷移関数，実行関数，プロセス指定列等の定義は，

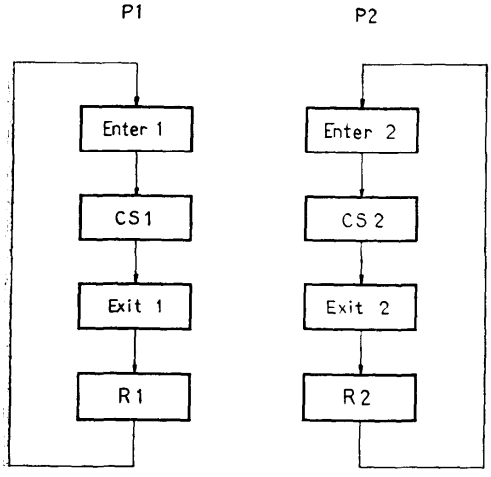

Fig. 3 mutual exclusion problem

付録1 (p. 847 参照) に示してある．

定義 1

状態遷移関数 $TR(\varepsilon(p1, p2, \gamma), S_0)$ が定義されているような任意のプロセス指定列 γ と，初期状態 S_0 とに対して，実行関数 ε $(p1, p2, \gamma)$ によって与えられる実行順序列が，次のような部分列

$$\tau_1=\text{Enter } 1\cdot\tau_1'\cdot\text{Enter } 2, \qquad (1)$$

$$\tau_2=\text{Enter } 2\cdot\tau_2'\cdot\text{Enter } 1, \qquad (2)$$

ここに，τ_1' は Exit 1 を含まない任意の部分列*，

τ_2' は Exit 2 を含まない任意の部分列，

を決して含まないとき，プロセス P1 と P2 とは，危険部分 CS に関して，相互排除の関係にあるという．

Enter が消費操作に，Exit が生産操作に対応するが，非可分操作の適用範囲を違えることにより，**Fig. 4** の解1及び **Fig. 5** の解2が得られる(次頁参照)．解1，解2に於て，同期変数に対する判定分岐操作と代入操作それ自身は，非可分操作として行われることを仮定する．また，解2に於ては，判定分岐操作とそれに続く代入操作とは，非可分操作として実現している．以上の2つの解に関して，次のことが成り立つ．

命題 1

解1は，相互排除問題の正しい解ではない．

(証明は，**付録 2.1** p. 848 参照)

命題 2

解2は，相互排除問題の正しい解である．

(証明は，**付録 2.2** p. 848 参照)

3.2 生産者—消費者問題

* 部分列は空列 (null sequence) をも含む

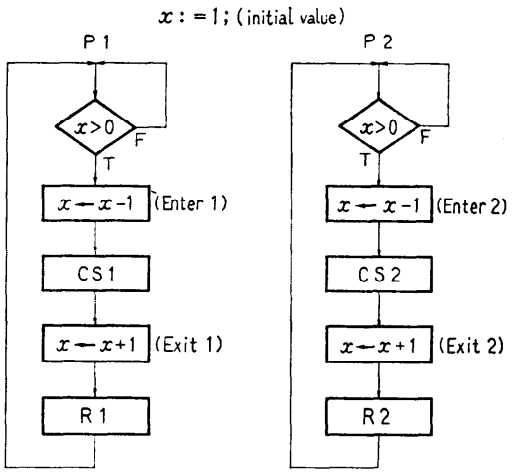

Fig. 4 Solution 1

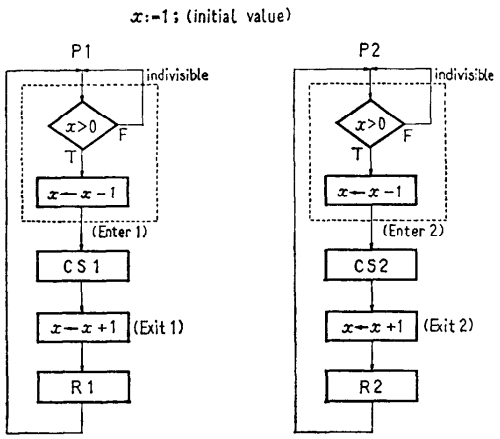

Fig. 5 Solution 2

非同期処理プログラムに於て，1つのプロセスは情報を作り出す事を繰り返し，他のプロセスはその情報を使用する事を繰り返すという場面が存在する．この場合，情報を作り出すプロセスを生産者 (producer)，情報を使用するプロセスを消費者 (consumer) といい，この両者の関係を管理する制御の問題を，**生産者—消費者問題** (producer-consumer problem) という．

Fig. 6 に示すように，生産者を PR，消費者を CON とすると，PR は Produce 処理を行ってから，情報を送り出す Send 命令を出し，また，CON は，情報の伝送を待つ Receive 命令を出してから Consume 処理を行う．このとき，生産者—消費者問題は，次のように定義される．

定義 2

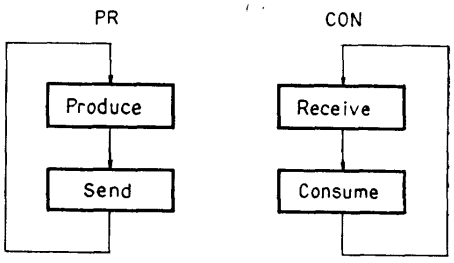

Fig. 6 producer-consumer problem

状態遷移関数 TR (ε(PR, CON, γ), S_0) が定義されているような任意のプロセス指定列 γ と，初期状態 S_0 とに対して，実行関数 ε (PR, CON, γ) によって与えられる実行順序列の任意の前部分列 (prefix) τ に対して，

$$N_\tau(\text{Receive}) \leqq N_\tau(\text{Send}) \qquad (3)$$

が成り立つとき，PR と CON とは，生産者—消費者の関係にあるという．

但し，N_τ (Receive): τ 中に現われる Receive 命令の数，

N_τ (Send): τ 中に現われる Send 命令の数．

Receive が消費操作に，Send が生産操作に対応する．この場合，**Fig. 7** の解3に示すように，Receive の実現に於ては，一般形の消費操作のように，通過条件に対する判定分岐操作と下降関数による代入操作とを非可分操作として行わなくてもよいことが証明できる．

命題 3

解3は，生産者—消費者問題の正しい解である．

(証明は**付録 2.3** p. 848 参照)

命題3により，同期基本命令の実現に於て，通過条件の判定分岐操作と，下降関数による代入操作とを，可分操作として行ってもよい場合があることが示され

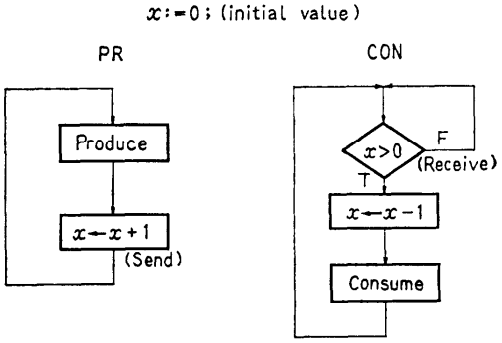

Fig. 7 Solution 3

た．

4. 非可分操作の実現

前節に述べた同期基本命令の実現に於ては，ある範囲の操作を非可分操作として実現する事が要請される．非可分操作は，相互排除問題として記述できるので，それを解くことのできる同期基本命令を与えられているシステムのレベルでは，容易に実現できる．しかし，今度は，そのような同期基本命令の実現のために，一つ下のレベルのシステムで非可分操作が必要となってくる．このように，次々とシステムのレベルを下げてゆくと，ある種の基本的なハードウェアの機能の完備が必要となってくる．ここでは，計算機システムの構成に対応して，非可分操作に必要なハードウェアの機能とその問題点を検討してみよう．

4.1 単一プロセサ・システム

単一プロセサから成るシステムに於ては，任意の時間間隔に対し割込み禁止を指定できるようなモードの設定により，必要な非可分操作が可能となる．スーパーバイザ・モード，マスタ・モードなどの名称で，このようなモードが設けられているのが一般的である．

4.2 共有主記憶装置を有した多重プロセサ・システム

多重プロセサ・システムに於て，共有主記憶装置を介して接続されている場合，消費操作に対応する動作を機械命令のレベルで実現したTS (Test and Set) 命令を用いれば，非可分操作が可能となる．**Fig. 8** に，TS 命令の詳細を示す．フラッグの内容の判定操作とフラッグへの書き込み操作とは，主記憶装置への2回のアクセスを必要とし，一般には非可分な操作ではないが，主記憶装置の制御ユニットは1つに絞られているので，それを利用して非可分操作を実現することができる．TS 命令の対として，生産操作に対応するフラッグの CLEAR 命令が必要であるが，これは，一般的な機械命令を使用すれば良い．TS 命令を用いた同期基本命令の実現に於ける諸問題は，文献(9)に詳しく検討されている．

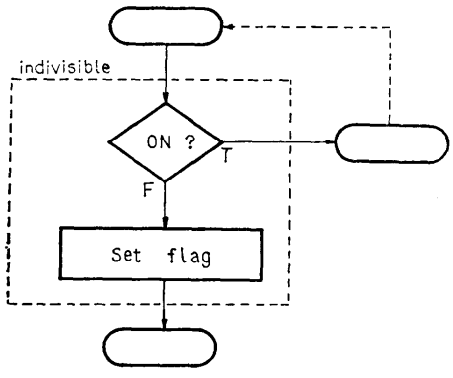

Fig. 8 TS instruction

4.3 共有主記憶装置を有しない多重プロセサ・システム

計算機システムを．通信回線やデータ交換装置を介して接続するシステム構成は，計算機ネットワークや計算機複合体等の必要性に伴い，今後益々増加するものと思われるが，このようなシステムにおいては，主記憶装置は共有せず，プロセス間の連絡線が共有のハードウェアとなっている．プロセス間の連絡線は，一般に，(1)データの転送，(2)割り込み信号の転送，の2つの機能を備えている．このような連絡線を用いた非可分操作の実現は，TS 命令のシミュレーションを行う事を目ざせばよいが，プロセサ間の物理的な接続形態により，種々の場合が考えられる．

(I) 民主型

プロセサが，**Fig. 9** (1) に示すように完全に対等に接続されている場合を，民主型と呼ぶ．民主型の例として，2台のプロセサから成るシステムにおける非可分操作の実現を考えてみよう．各プロセサ毎に，フラッグ fl1，fl2を用意し，両者の Test and Set が完了したときに，TS 命令がシミュレートされたとする．**Fig. 10** に，プロセサ1側の処理手続きと，プロセサ2側の割込み処理手続きとを示す．これらの処理

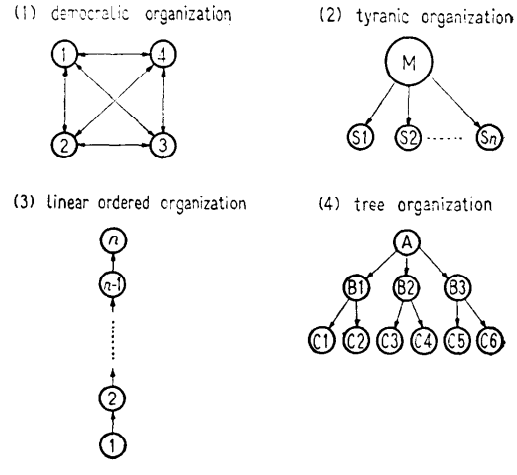

Fig. 9 processor organization

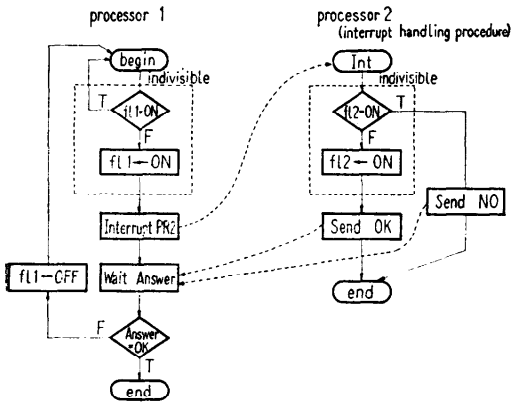

Fig. 10 implemetation for democratic organization

手続きにあらわれる非可分操作は，割込み禁止機能か，TS 命令かを用いて実現する．この場合，2つのプロセサの処理は対称であるから，全く同じタイミングで処理が進行すると，お互に相手のフラッグのセットができずに，無駄な繰り返しを続けてしまう可能性がある．これを防ぐためには，処理を非対称にすれば良く，たとえば，(1)フラッグのセットの順序を全てのプロセサに共通にする，(2)タイマなどを用いて，処理の繰り返しの間隔をプロセサごとに変える，などの方法が考えられる．

民主型の構成においては，全て均質な制御システム(ハードウェアとソフトウェア)を持たせることができるので，システムの設計は比較的容易である．また，どれかのプロセサがダウンしても，全体に及ぼす影響は少ないので，信頼性は高くなる．一方，全てのプロセサのフラッグを調べなければならず，プロセサ間の信号のやり取りが頻繁になり，能率が大幅に低下する．

(Ⅱ) 専制型

Fig. 9 (**2**) に示すように，1台のマスタプロセサ(master processor)に，n 台のスレーブ・プロセサ(slave processor)が接続されている構成を，専制型という．この場合，マスタ・プロセサの主記憶装置内に共通のフラッグを設定し，それに対する Test and Set を行うことにより，TS 命令をシミュレートする．**Fig. 11** に，マスタ・プロセサおよびスレーブ・プロセサ S_i の処理手続きを示す．S_i に複数個のプロセスが存在する場合，それぞれのプロセスが行うマスタ・プロセサとの交信過程が交錯することがあるので，Fig. 11 における Interrupt M と，Wait Answer の対応関係をきちんと管理する必要がある．

専制型の構成においては，民主型構成に比べて，プロセサ間の交信の回数が1回で済むので，処理能率が向上する．一方，マスタ・プロセサは，他のプロセサとは異質の制御システムを持つことになり，設計が複雑化する．また，マスタ・プロセサがダウンしたら，その影響がシステム全体におよぶので，信頼性の低下を来たす．

(Ⅲ) 階層型

プロセサの接続形態として，幾つかのレベルを持った階層型の構成を考えることができる．

たとえば，**Fig. 9** (**3**) に示す線形順序構成においては，プロセサ i は，プロセサ $j(i \geqq j \geqq 1)$ の持つフラッグを全て Test and Set した時に非可分操作が実現

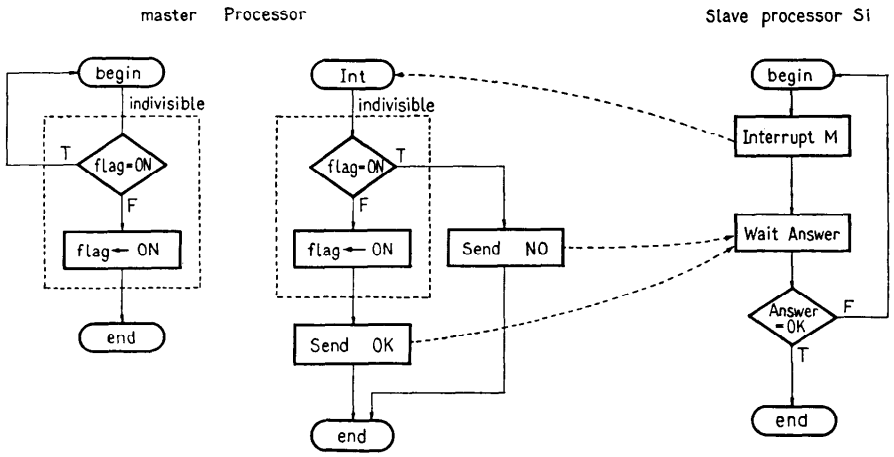

Fig. 11 implementation for tyranic organization

したと定義する．このために，プロセサ i は自分のフラッグをセットしたあと，プロセサ $(i-1)$ のフラッグのセットを要求し，以下同様にして，順々にプロセサ1までのフラッグのセットの要求を伝える．途中のプロセサ $k\,(i>k\geqq 1)$ で，フラッグが既にセットされている場合は，プロセサ k が管理している待ち行列に入ってフラッグのリセットを持つ．このように非可分操作を定義すると，その非可分操作を実行している最中は，自分よりも上位のプロセサに中断される可能性はあるが，自分よりも下位のプロセサによって中断されることはない．

また，**Fig. 9**（4）に示すような木構造構成においては，各部分木の根が専制型のマスタ・プロセサに対応し，各部分木ごとにグループを作り，そのグループに関して非可分操作を実現することができる．

このような階層構造を用いると，システムの環境に応じて，非可分操作を能率良く実現することが可能となる．

5. むすび

非同期処理プログラムで重要な働きをする同期基本命令の一般形を定義し，その実現における非可分操作の必要性を論じた．同期基本命令の消費操作においては，一般に，通過条件に対する判定分岐操作と，下降関数を用いた代入操作を非可分に実現しなければならないが，生産者—消費者問題においては必ずしもその必要性がないことが示された．

また，非可分操作を具体的に実現するために必要なハードウェアの機能について，計算機システムの構成に応じて論じ，問題点を検討した．共有主記憶装置を持つ多重プロセサ・システムにおいては，TS 命令を用いて，比較的容易に非可分操作を実現できるが，共有主記憶装置を持たない多重プロセサ・システムにおいては，非可分操作の能率の良い実現法を今後検討する必要がある．

謝辞

日頃御指導頂いている電子技術総合研究所石井治ソフトウェア部長，同西野博二パターン情報部長，筑波大学西村敏男教授に感謝の意を捧げる次第です．

参考文献

1) E. W. Dijkstra: Co-operating Sequential Processes, *Programming Languages*, (ed. F. Genuys), Academic Press, New York, pp. 43–112 (1968)
2) S. S. Patil: Limitations and Capabilities of Dijkstra's Semaphore Primitives among Processes, Computation Structures Group Memo, Project MAC, MIT (1971)
3) H. Vantilborgh and A. van Lamsweerde: On an Extension of Dijkstra's Semaphore Primitives, Information Processing Letters, Vol. 1, No. 5, pp. 181–186 (1972)
4) P. L. Woden: Still Another Tool for Synchronizing Co-operating Processes, Carnegie-Mellon University (1972)
5) J. H. Saltzer: Traffic Control in a Multiplexed Computer System, MAC-TR-30, Project MAC, MIT (1966)
6) J. B. Dennis and E. C. Van Horn: Programming Semantics for Multiprogrammed Computations, CACM, Vol. 9, No. 3, pp. 143–155 (1966)
7) D. Scott and C. Strachey: Towards a Mathematical Semantics for Computer Languages, Proc. of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, pp. 19–46 (1971)
8) J. M. Cadiou and J. J. Lévy: Mechanizable Proofs about Parallel Processes, IE^3 14 th Annual Symposium on Switcling and Automata Theory, pp. 34–48 (1973)
9) B. W. Lampson: A Scheduling philosophy for Multi-processing Systems, 1 st ACM Symposium on Operating Systems Principles, pp. 99–122 (1967)

付録 1　＜非同期処理プログラムの数学的意味論＞

同期問題の解を証明するために，Scott 等[7]の提唱した数学的意味論 (mathematical semantics) を並列プログラムに適用した Cadiou 等[8]の方法を用いて，2つのプロセス P1 と P2 とから成る非同期処理プログラム P に対する数学的意味の定義を与える．

プロセスに属する各演算要素には，それによってひき起される状態の遷移を与える意味関数 $\sigma: \pi\times S\to S$ が与えられている．π は，演算要素，S は関数 $S: Id\to Val$ で与えられる状態を表わす．(Id は，プログラム内で用いられる変数名の集合，Val はそれらがグり得る値の集合)

P 全体の意味を表わすために，実行関数 (execution function) ε: p1×p2×$\Gamma\to\Sigma$ と，状態遷移関数 (state transition function) $TR: \Sigma\times S\to S$ とを，次の帰納定義式で与える．

$$\begin{aligned}&\varepsilon(\mathrm{P}1,\mathrm{P}2,\gamma)\Leftarrow \text{if } hd(\gamma)=1\\&\quad\text{then } hd(\mathrm{P}1)\cdot\varepsilon(\mathrm{tl}(\mathrm{P}1),\ \mathrm{P}2,\ \mathrm{tl}(\gamma))\\&\quad\text{else } hd(\mathrm{P}2)\cdot\varepsilon(\mathrm{P}1,\ \mathrm{tl}(\mathrm{P}2),\ \mathrm{tl}(\gamma))\end{aligned}\qquad(4)$$

$$TR(\Sigma, S) \Leftarrow \text{if } hd(\Sigma)=\text{null then } S \text{ else } TR(\text{tl}(\Sigma),\ \sigma(hd(\Sigma),\ S)) \quad (5)$$

ここに,

$P1 \triangleq \Pi_1^{*}$** $\Pi_1=\{$P1を構成する演算要素$\}$

$P2 \triangleq \Pi_2^{*}$ $\Pi_2=\{$P2を構成する演算要素$\}$

$r \in \Gamma, \Gamma \triangleq \{1,2\}^{*}$ (プロセス指定列)

$\Sigma \triangleq (\Pi_1 \cup \Pi_2)^{*}$ (実行順序列)

hd, tl は, 記号列に対するヘッド関数 (head function), テイル関数 (tail function), また, ・は, 記号列の連結 (concatenation) をあらわす.

ε は, P が実行されるときの演算要素の実行順序を与え, TR は, P によってひき起される全体的な状態の遷移を与える. 非同期に実行されていることを示すために, プロセスの番号のランダムな列であるプロセス指定列 Γ を用い, その番号に従って, 各プロセスの実行を1つづつ進めてゆく.

付録 2.1 <命題1の証明>

プロセス指定列 γ として, $\gamma=1212$ を考えると, 明らかに, $\tau=$Enter 1・Enter 2 という部分列を生じ, 定義1に示した相互排除の関係に反する. (q. e. d.)

付録 2.2 <命題2の証明>

Enter 命令における非可分操作は, 次に与える演算をしている.

$$f(x) \Leftarrow \text{if } x>0 \text{ then } x-1 \text{ else } f(x) \quad (6)$$

その意味関数は, (6)式の最小固定点関数を用いて,

$$\sigma(\text{Enter}(x), S) \equiv [x>0 \rightarrow \lambda z(z=x \rightarrow x-1, S(z)),\ \perp^{**}]^{***} \quad (7)$$

とあらわす. 他の意味関数は, 次のように与えられる

$$\sigma(\text{Exit}(x), S) \equiv \lambda z[z=x \rightarrow x+1, S(z)] \quad (8)$$

$$\sigma((x \leftarrow e),\ S) \equiv \lambda z[z=x \rightarrow e,\ S(z)] \quad (9)$$

$$\sigma(\text{CS}(x),\ S) \equiv \lambda z[z=x \rightarrow S(z),\ \text{CS}(x)] \quad (10)$$

$$\sigma(R(x),\ S) \equiv \lambda z[z=x \rightarrow S(z),\ R(z)] \quad (11)$$

CS(z), $R(z)$ は, CS および R で行われる操作をあらわす関数であるが, "$\perp$" 要素に対しては, 必らず, "$\perp$" 要素へ写像するものとする.

各プロセスの動作は, 次のようにあらわす.

$$P1=(\text{Enter 1} \cdot \text{CS 1} \cdot \text{Exit 1} \cdot \text{R 1})^{****} \quad (12)$$

$$P2=(\text{Enter 2} \cdot \text{CS 2} \cdot \text{Exit 2} \cdot \text{R 2})^{*} \quad (13)$$

γ の長さ n の前部分列 (prefix) を $\gamma(n)$ とし, ε_n(P1, P2, γ)$\triangleq\varepsilon$(P1, P2, $\gamma(n)$) としたとき, 証明したい性質 P に対して,

$$\frac{P(\varepsilon_0)\ P(\varepsilon_n) \vdash P(\varepsilon_{n+1})}{P(\varepsilon)} \quad (14)$$

という帰納原理 (Induction Principle) が成り立つ.

$P(\varepsilon_0)$: 明らか.

$P(\varepsilon_n) \perp P(\varepsilon_{n+1})$: ε_n(P1, P2, γ)$=\pi_0 \cdot \pi_1 \cdot \cdots \cdot \pi_n$, $\gamma(n)=\gamma_0 \cdot \gamma_1 \cdot \cdots \cdot \gamma_n$ とおき, ε_n(P1, P2, γ) が相互排除の関係を満すと仮定. プロセスの対称性により, (1)式の τ_1 だけ考えれば十分. $\gamma_{n+1}=1$ の場合, π_{n+1} により τ_1 を生ずることはない. $\gamma_{n+1}=2$ の場合, ε_n(P1 P2, γ) にあらわれる P1 側の最後の演算要素を π_m とする.

$\pi_m=$CS 1 のとき, Enter 1 の場合と同じ議論を適用.

$\pi_m=$Exit 1 または, $\pi_m=$R 1 のとき, π_{m+1} により τ_1 が生ずることはない.

$\pi_m=$Enter 1 の場合を考える. $\pi_{m+1}=$CS 2 または $\pi_{m+1}=$Exit 2 の場合, いずれも ε_n(P1, P2, γ) の仮定に矛盾. $\pi_{m+1}=$Enter 2 のとき, $m<n$ ならば, ε_n (P1, P2, γ) の仮定に矛盾. また, $m=n$, すなわち, $\pi_{n+1}=$Enter のとき, $x=0$ が成り立っているから, $TR(\varepsilon_{n+1}$(P1, P2, γ), $S_0) \equiv \perp$ となり, 相互排除の関係を保つ. $\pi_{n+1}=$R 2 のとき, $m=n$, すなわち, $\pi_{n+1}=$R 2 のとき, τ_1 を生じさせないことは明らか. また, $m<n$ のとき, R 2 の次の演算要素は, Enter 2 であるから, $m=(n-1)$ の場合だけを考えればよい. このとき, $\pi_{n+1}=$Enter 2 となり, 上と同じ理由で, 相互排除の関係を破壊することはない. (q. e. d.)

付録 2.3 <命題3の証明>

各演算要素の意味関数は, 次のように与えられる.

$$\sigma(\text{Receive}(x), S) \equiv [x>0 \rightarrow \lambda z.\ S(z), \perp] \quad (15)$$

$$\sigma(\text{Send}(x), S) \equiv \lambda z[z=x \rightarrow x+1, S(z)] \quad (16)$$

$$\sigma((x \leftarrow x-1), S) \equiv \lambda z[z=x \rightarrow 1, S(z)] \quad (17)$$

$$\sigma(\text{Produce}(x), S) \equiv \lambda z[z=x \rightarrow S(z),\ \text{Pro}(z)] \quad (18)$$

$$\sigma(\text{Consume}(x), S) \equiv \lambda z[z=x \rightarrow S(z), \text{Con}(z)] \quad (19)$$

Pro(z), Con(z) は, Produce および Consume で行われる操作をあらわす関数であるが, "$\perp$" 要素に対しては, "$\perp$" 要素へ写像するものとする.

各プロセスの動作は, 次のようにあらわす.

$$PR=(\text{Produce} \cdot \text{Send})^{*} \quad (20)$$

---

* X を記号の集合とすると, X^{*} は, X の要素から成る有限または無限の記号列をあらわす.

** "$\perp$" は, 全域未定義の関数をあらわす.

*** これは, McCarthy の条件式の表記法である.

**** α を記号列とすると, α^{*} は, 部分列 α を有限または無限個連結した記号列をあらわす.

$$\mathrm{CON}=(\mathrm{Receive}\cdot(x\leftarrow x-1)\cdot\mathrm{Consume})^{*} \quad (21)$$

また，プロセス指定列γにおいて，$hd(\gamma)=1$ のとき PR を，$hd(\gamma)=2$ のとき CON を指定するものとする．

証明する性質をPとして，帰納原理(14)を用いる．

$P(\varepsilon_0)$：明らか

$P(\varepsilon_n)\vdash P(\varepsilon_{n+1})$：$\varepsilon_n(\mathrm{PR},\ \mathrm{CON},\ \gamma)=\pi_0\cdot\pi_1\cdot\cdots\cdot\pi_n$，$\gamma(n)=\gamma_0\cdot\gamma_1\cdot\cdots\cdot\gamma_n$ とおき，$\varepsilon_n(\mathrm{PR},\mathrm{CON},\gamma)$ が生産者一消費者の関係を満すと仮定．

$\gamma_{n+1}=1$ の場合，π_{n+1} が何であっても，(3)式が保たれることは明らか．

$\gamma_{n+1}=2$ の場合，$\pi_{n+1}=(x\leftarrow x-1)$，または $\pi_{n+1}=$ consume ならば，(3)式が保たれることは明らか．$\pi_{n+1}=$Receive の場合，$x=N_{\varepsilon_n}(\mathrm{Send})-N_{\varepsilon_n}(\mathrm{Receive})$ が成り立っている．$x>0$ のとき，$\pi_{n+1}=$Receive としても，(3)式を保つ．$x=0$ のとき，$\pi_{n+1}=$Receive とすると，$\mathrm{TR}(\varepsilon_{n+1}(\mathrm{PR},\mathrm{CON},\gamma),S_0)\equiv\perp$ となる．いずれの場合も，生産者一消費者の関係を満している．

(q. e. d.)

(昭和 49 年 5 月 25 日受付)
(昭和 49 年 7 月 17 日再受付)