On the Implementation of Synchronization Primitives

Nobuo Saito*

Abstract

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

In the implementation procedure of these primitives, several operations should be indivisibly executed. The author investigated the part of the procedure which should be considered as an indivisible operation by considering typical synchronization problems.

The realization of indivisible operation is also an important problem. Several hardware mechanisms are required to realize these operations, and the problems in using these mechanisms are also discussed from the point of computer system organizations.

1. Introduction

In an asynchronous control program system, a set of synchronization primitives is provided for programs to communicate with each other. A semaphore system, a typical synchronization primitive set, was proposed by Dijkstra[1], and several kinds of its extended systems have been reported[2-4]. It is so universally designed that the properties of other synchronization primitives can be expressed in terms of this system. This paper defines a general form of synchronization primitives, and several sets of primitives are described by using it.

When synchronization primitives are implemented in an operating system, a certain region of an implementing procedure should be executed as an indivisible operation: once started, it should not be intervened by any other operations such as external interruptions until the end of this region. This paper investigates the part of the procedure which should be indivisibly executed in consideration of several typical synchronization problems.

In order to realize an indivisible operation, some basic mechanisms are required in a low level system, or a hardware system. This paper summarizes these mechanisms, and several problems of them are discussed especially for a multiprocessor system.

2. General Form of Synchronization Primitives

Common properties of synchronization primitives reported so far are expressed by

This paper first appeared in Japanese in Joho—Shori (Journal of the Information Processing Society of Japan), Vol. 15, No. 11 (1974), pp. 841~849.

* Institute of Electronics and Information Science, The University of Tsukuba

the following general form.

(1) synchronization variables

$\underset{\sim}{x} = (x_1, x_2, \ldots, x_n)$

(global variables, usually a vector consisting of n scalar variables)

(2) consume operation

The detailed flow of the consume operation is shown in fig. 1. In this flow, $C(\underset{\sim}{x})$ is called a pass condition: it must be satisfied when the consume operation is passed. A mapping $D:\underset{\sim}{x} \rightarrow \underset{\sim}{x}$ is called a down function: it is usually a monotonic decreasing function of $\underset{\sim}{x}$. In general, enclosed part by a dashed line in fig.1 should be indivisibly executed. In this operation, a program waits until its synchronization state satisfies a specified condition, and when satisfied, its state should be transitioned to the prespecified one.

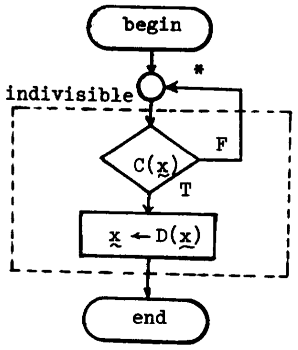

Fig.1 the detailed flow of consume operation

(3) produce operation

The detailed flow of the produce operation is shown in fig.2. A mapping $U:\underset{\sim}{x} \rightarrow \underset{\sim}{x}$ is called an up function: it is usually a monotonic increasing function of $\underset{\sim}{x}$. In this operation, the state which does not satisfy the pass condition is usually transitioned to the state which satisfies it. In this sense, it is an inverse of the consume operation. In the flow of the consume operation, a busy waiting form is used. It is not desirable in view of the processor utilization. It is, however, very convenient

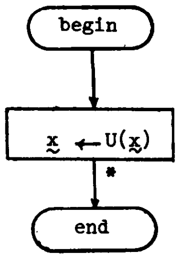

Fig.2 the detailed flow of produce operation

Table 1 synchronization primitives in general form

		consume operation			produce operation	
synchronization primitive system	synchronization variable $\underset{\sim}{x}$		pass condition $C(\underset{\sim}{x})$	assignment using down func. $\underset{\sim}{x} \leftarrow D(\underset{\sim}{x})$		assignment using up func. $\underset{\sim}{x} \leftarrow U(\underset{\sim}{x})$
standard semaphore 1)	S	P(S)	$S > 0$	$S \leftarrow (S-1)$	V(S)	$S \leftarrow (S+1)$
PV Multiple 2)	$S_1, S_2, .., S_n$	$P(S_1, S_2, .. S_n)$	$\forall i\ S_i > 0$ $(1 \leq i \leq n)$	$\forall j\ S_j \leftarrow (S_j - 1)$ $(1 \leq j \leq n)$	$V(S_1, S_2, .., S_n)$	$\forall j\ S_j \leftarrow (S_j + 1)$ $(1 \leq j \leq n)$
PV Chunk 3)	S	P(S,n)	$S \geq n$	$S \leftarrow (S-n)$	V(S,n)	$S \leftarrow (S+n)$
up/down system4)*	$S_1, S_2, .., S_n$	$down(S_i)$	$\sum_{i=1}^{n} S_i \geq 0$ **	$S_i \leftarrow (S_i - 1)$	$up(S_i)$	$S_i \leftarrow (S_i + 1)$
wakeup/block 5)	WWS(wakeup-waiting switch)	block	WWS=ON	WWS ← OFF	wakeup	WWS ← ON
lock/unlock 6)	interlock key	lock	interlock key=OFF	interlock key ← ON	unlock	interlock key ← OFF

* conditional branch operation for the pass condition and assignment operation using the down function may not be executed in one indivisible operation.

** this is the pass condition for a semaphore application $\{S_1, S_2, \ldots, S_n\}$.

to use this form when the formal semantics of a synchronization primitive is discussed, and it is used through this paper.

Several kinds of synchronization primitives proposed so far are summarized in table 1 by using a general form.

3. The necessity of Indivisible Operation

A synchronization primitive can be implemented as a general form mentioned in the previous section. In the consume operation, a certain part of operations should be executed as one indivisible operation. This section investigated such a part for two synchronization problems.

3.1 Mutual Exclusion Problem

Consider the mutual exclusion problem[1)] in a system P=(P1,P2), consisting of two processes. As is shown in fig.3, a critical section is preceded by an Enter command and is succeeded by an Exist command.

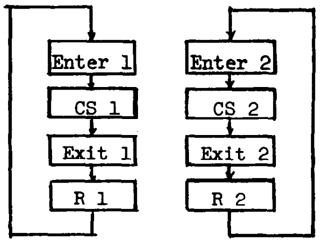

Fig.3 mutual exclusion problem

Definition 1 (Definitions of several terms are given in Appendix)

The process P1 and P2 are mutually exclusive with respect to the critical section CS if the following two partial sequences cannot be included in the execution sequence generated by the execution function $\varepsilon(P1,P2,\gamma)$, provided that an initial condition S_0 and an oracle γ define the state transition function $\mathcal{TR}(\varepsilon(P1,P2,\gamma),S_0)$.

$$\tau_1 = \text{Enter1} \cdot \tau_1' \cdot \text{Enter2} \qquad (1)$$

$$\tau_2 = \text{Enter2} \cdot \tau_2' \cdot \text{Enter1} \qquad (2)$$

where τ_1' is a finite sequence which does not include Exit1;

τ_2' is a finite sequence which does not include Exit2.

Two solutions are shown in fig.4(solution 1) and in fig.5(solution 2). In the solution 2, the region enclosed by a dashed line is assumed to be executed as one indivisible operation.

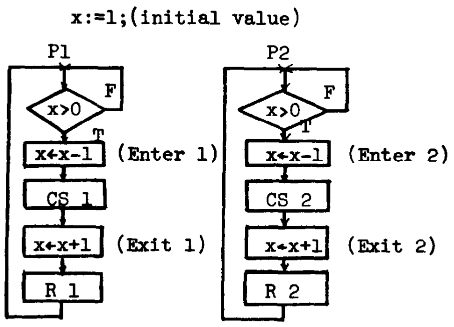

Fig.4 solution 1

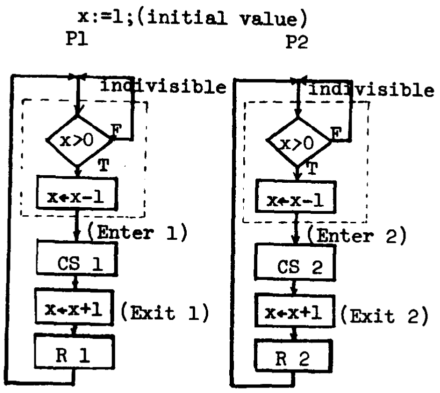

Fig.5 solution 2

Proposition 1

The solution 1 does not solve the mutual exclusion problem, but the solution 2 solves the mutual exclusion problem correctly. (Proof is omitted.)

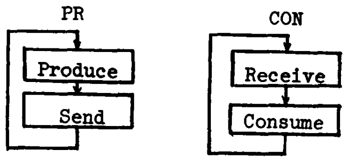

Fig.6 producer-consumer problem

3.2 Producer-Consumer Problem

Consider the producer-consumer problem[2)] consisting of a producer(PR) and a consumer (CON) (fig.6).

x:=0; (initial value)

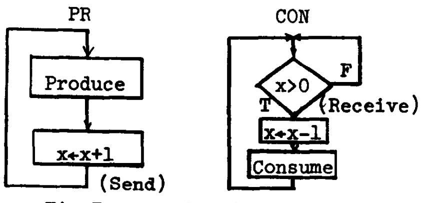

Fig.7 solution 3

Definition 2

The processes PR and CON are correctly related as the producer-consumer if any prefix τ of the execution sequence generated by the execution function $\varepsilon(PR,CON,\gamma)$ satisfies the following condition, provided that an initial state S_0 and an oracle γ define the state transition function $\mathcal{TR}(\varepsilon(PR,CON,\gamma),S_0)$.

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

where N_τ(Receive) is the number of Receive commands in the prefix τ, and N_τ(Send) is the number of Send commands in the prefix τ.

The solution for this problem is shown in fig.7(solution 3).

Proposition 2

The solution 3 solves the producer-consumer problem correctly (Proof is omitted).

Proposition 2 shows that for some cases of synchronization, the conditional branch operation for the pass condition and the assignment operation using the down function are not necessarily executed in one indivisible operation.

4. Realization of Indivisible Operation

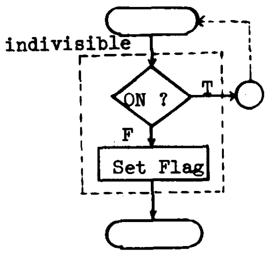

Fig.8 TS instruction

In the implementation of synchronization primitives mentioned in the previous section, it is required that a certain sequence of operations should be indivisibly executed. Some basic hardware mechanism is required to realize an indivisible operation, and this section discusses such a mechanism from the point of a computer system organization.

In a single processor system, it is possible to realize an indivisible operation if the processor has an execution mode (for example, supervisor mode or master mode) in which an interruption is prohibited for an arbitrary time interval.

When processors in a multiprocessor system are connected via shared memory, it is possible to realize an indivisible operation if you use TS(Test and Set) instruction, a kind of the consume operation implemented as a hardware instruction. Fig. 8 shows the detail of this instruction.

In a multiprocessor system without shared memory, processors are usually connected via communication lines or data exchange modules. In order to realize an indivisible operation through use of such a communication device, it is necessary to simulate TS instruction.

In a democratic organization (fig.9(1)), each of the processors prepares a flag. TS instruction is considered to be simulated when 'Test and Set' for all these flags are completed.

In a tyrannic organization (fig.9(2)), TS instruction can be simulated by testing and setting a common flag prepared in the main memory of the master processor.

Consider a system organization with several hierarchical levels. In a linear ordered organization (fig.9(3)), a processor i is considered to realize an indivisible operation if it finishes testing and setting flags of processors j(1 ≦ j ≦ i). In a tree structure organization (fig.9(4)), the processor of the root of each subtree plays the same role of a master processor in a tyrannic organization. Thus, an indivisible operation can be easily realized among the processor belonging to a subtree.

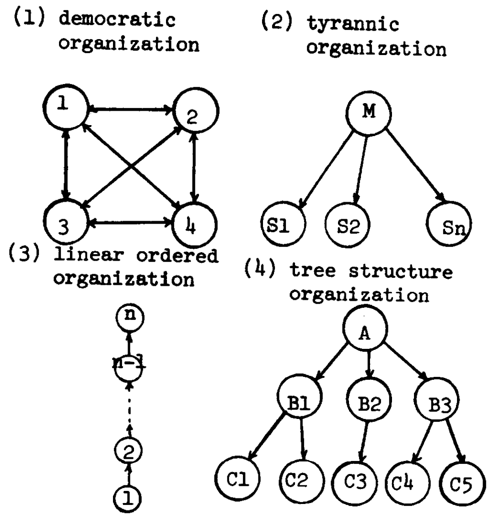
(1) democratic organization
(2) tyrannic organization
1
2
3
4
M
S1
S2
Sn
(3) linear ordered organization
(4) tree structure organization
n
n-1
2
1
A
B1
B2
B3
C1
C2
C3
C4
C5

Fig. 9 processor organization

5. Concluding Remarks

This paper defines a general form of synchronization primitives. The regions of indivisible operations in implementing procedures of these primitives are then discussed. It is shown that the conditional branch operation for the pass condition and the assignment operation using a down function in the consume operation are not necessarily executed indivisibly when used in a producer-consumer problem.

It is also discussed what kinds of hardware mechanisms are required in realizing these indivisible operations for several system organizations. A further discussion will be necessary to investigate an efficient method of realization of the indivisibility in a multiprocessor system without shared memory.

Acknowledgement

The author would like to thank Dr. O. Ishii and Dr. H. Nishino of Electrotechnical Laboratory, Prof. T. Nishimura of the University of Tsukuba for their encouragement to this work.

References

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 ans 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.Wodon: 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.Levy: Mechanizable Proofs about Parallel Processes, IEEE 14th Annual Symposium on Switching and Automata Theory, pp.34-48 (1973)

Appendix

< Mathematical Semantics of Asynchronous Control Programs >[7,8]

Consider P consisting of two processes P1 and P2. A system state S is a function $S:Id \rightarrow Val$, where Id is a set of identifiers and Val is a set of values. An execution function $\varepsilon: P1 \times P2 \times \Gamma \rightarrow \Sigma$ gives an execution sequence of operators, and a state transition function $\mathcal{TR}: \Sigma \times S \rightarrow S$ gives a global state transition caused by P. They are defined by the following recursive definitions.

$$\varepsilon(P1,P2,\gamma) \Leftarrow \underline{\text{if}}\ hd(\gamma)=1\ \underline{\text{then}}\ hd(P1)\cdot\varepsilon(tl(P1),P2,tl(\gamma))\ \underline{\text{else}}\ hd(P2)\cdot\varepsilon(P1,tl(P2),tl(\gamma)) \quad (4)$$

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

where

$P1 \triangleq \Pi_1^{\omega}$* $\quad \Pi_1 =$ {operators in process P1},

$P2 \triangleq \Pi_2^{\omega}$ $\quad \Pi_2 =$ {operators in process P2},

$\gamma \in \Gamma \quad \Gamma \triangleq \{1,2\}^{\omega}$ (oracle, a random sequence of the number of the processes which specifies the process to be executed next),

$\Sigma = (\Pi_1 \cup \Pi_2)^{\omega}$ (execution sequence),

$\sigma : \Pi \times S \rightarrow S$ (semantic function of operator Π),

hd: head function, tl: tail function, $\cdot$: concatenation for symbol strings.

* Given a set X of symbols, X^{ω} is a string consisting of elements of X.