数理解析研究所講究録
第 236 巻 1975 年 153-173

Mathematical Theories on Operating System

Nobuo Saito

(Electrotechnical Laboratory)

1. Introduction

- Theory of Computation
  - Computational Complexity (quantitative)
  - Correctness (qualitative)
- Operating System
  - Performance (quantitative)
  - Correctness (qualitative)
- Performance
  - Queuing Model
  - Working Set Model
  - Scheduling Algorithm etc.
- Correctness
  - Synchronization Problem
  - Resource Allocation Problem
  - Structure of System etc.

2. Petri Net Model [1]

Definition 2.1 (Petri Net)

A Petri Net N is a directed graph defined as a quadraplet, $\langle T,P,A,B^0 \rangle$, where

$T = \{t_1,\ldots,t_m\}$ is a finite set of transitions } the node of

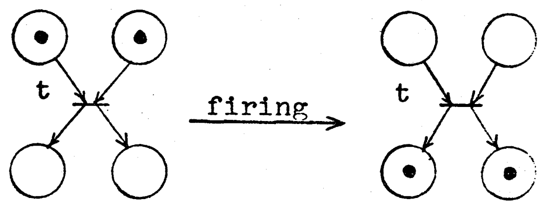
$P = \{p_1,\ldots,p_n\}$ is a finite set of places } the graph

$A = \{a_1,\ldots,a_k\}$ is a finite set of directed arcs of the form $\langle x,y \rangle$ which either connect a transition to a

	place or a place to a transition
$B^0 \subset P$	is the initial stone distribution, the set of places which have stones initially.

transition → event

place → condition



Definition 2.2 (firing)

An occurrence of event e is represented by firing the transition t which represents e :

For any t, if each of its input places has at least one stone, t is enabled to fire. If t fires, a stone is removed from each of its input places and a stone is added to each of its output places.

3. Synchronization Problem

3-1 Synchronizing Primitives

(1) conflict-free type primitives

fork - join

activate - wait

wakeup - block

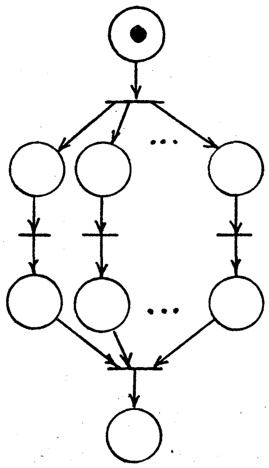
(2) conflict type primitives

lock - unlock

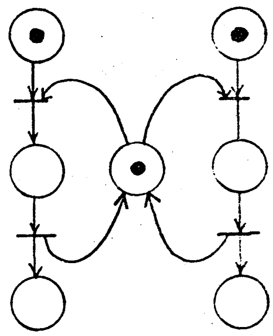
seize - release

(3) interruption type primitives

force



conflict-free



with conflict

Conflict : Two transitions are said to conflict if they share an input place and can be in enabled condition at the same time.

3-2 Semaphore Systems and Its Variations

(1) Semaphore System Proposed by Dijkstra [2]

- Semaphore Variables (integer type)
- P-operation
- V-operation

It can represent both the conflict-free type and the conflict primitives.

(2) Generalized Semaphore Systems

(i) Parallel P-operation [3,4]

$P[s_1, s_2, \ldots s_n]$

which waits for all semaphores to become non-zero, and then simultaneously operates on all of the semaphores.

(ii) Separation of the Door of a Critical Section from P-operation[5]

(a) A phrase P of a program may be preceded by any number n of occurrences of semaphores:

$$s_1 : s_2 : \ldots : s_n : P$$

The set $\{s_1, s_2, \ldots, s_n\}$ is a "semaphore application" whose "values" is $\sum_{i=1}^{n} s_i$. A phrase P thus preceded by a semaphore application cannot be initiated when the value $\sum_i s_i$ is negative. If an unsuccessful attempt has been made to initiate such a phrase, we shall say that the phrase is pre-initiated.

(b) The operation down s decreases the value of the semaphore s by 1.

(c) The operation up s increases the value of the semaphore s by 1. If the operation up s makes one or more semaphore applications take the value 0, then all pre-initiated phrases containing these applications are initiated collaterally.

(3) Conditional Critical Region [6,7]

$\{$ resource r; $Q_1//Q_2//\ldots//Q_n$ $\}$

$Q_1, Q_2, \ldots, Q_n$: disjoint processes executed in parallel

with r do C   C: critical region

with r when B do C   B: Boolean expression

with r do C await B

3-3 Classes in Synchronization Problems

- <u>simple</u> Petri Net: Petri Net in which no more than one input place of a transition is shared as input place with other transitions
- <u>non-simple</u> Petri Net: Petri Net in which more than two input place of a transition is shared as input place with other transitions
- <u>conflict-free</u> Petri Net: Petri Net in which two transitions which share an input place are not enabled (ready to fire) at the same time.
- Petri Net <u>with conflict</u>: Petri Net in which two transitions which share an input place may be enabled at the same time.

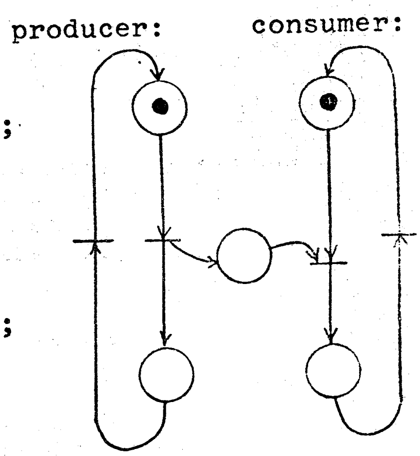
	conflict-free	with conflict
simple	producer-consumer problem	mutual exclusion problem readers-writers problem
non-simple	cigarette smokers' problem	dinig philosopher problem

(1) producer-consumer problem [2]

The producer produces a certain portion of information that has to be processed by the consumer.

The consumer can process the next portion of information which is produced by the producer.

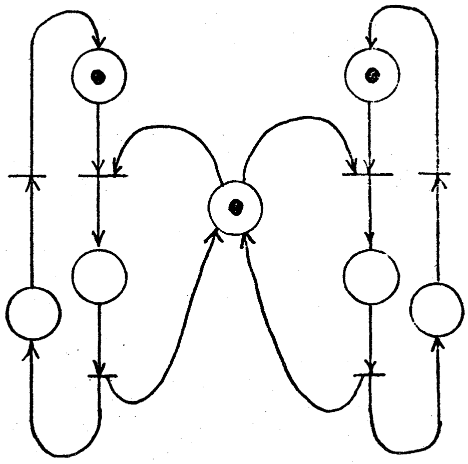
```
begin
  semaphore  numq ;   numq := 0;
  parbegin
    producer: begin
                again1:  produce the next portion ;
                         add portion to buffer ;
                         V(numq) ;
                         go to again1
              end ;
    consumer: begin
                again2:  P(numq) ;
                         take portion from buffer ;
                         process portion taken ;
                         go to again2
              end
  parend
end ;
```



(2) mutual exclusion problem [2]

Construct the N processes, each with a critical section, the execution of which must exclude one another in time.

```
begin
  semaphore  free ;  free := 1 ;
  parbegin
    .
    ..
    process i:  begin
                  Li:  P(free) ;
                       critical section i ;
```



```
                V(free) ;
                remainder of cycle i ;
                go to Li
            end ;
  .
  .
  .
 parend
end ;
```

(3) readers-writers problem [8]
Two classes of processes wish to the resources.

Writers must have exclusive access.

Readers may share the resource with an unlimited number of other readers.

```
begin
  integer  readcount ;
  semaphore  mutex,w ;
  readcount := 0 ;  mutex := w := 1 ;
  parbegin
    begin
      Reader i:  P(mutex) ;
                 readcount := readcount + 1;
                 if readcount = 1  then  P(w) ;
                 V(mutex) ;
                 --------------------

                 reading is performed
                 --------------------

                 P(mutex) ;
                 readcount := readcount - 1;
                 if readcount = 0  then  V(w) ;
                 V(mutex)
    end ;
    begin
      Writer j:  P(w) ;

                 --------------------

                 writing is performed

                 --------------------

                 V(w)
    end
  parend
end ;
```

(4) cigarette smokers' problem[3]

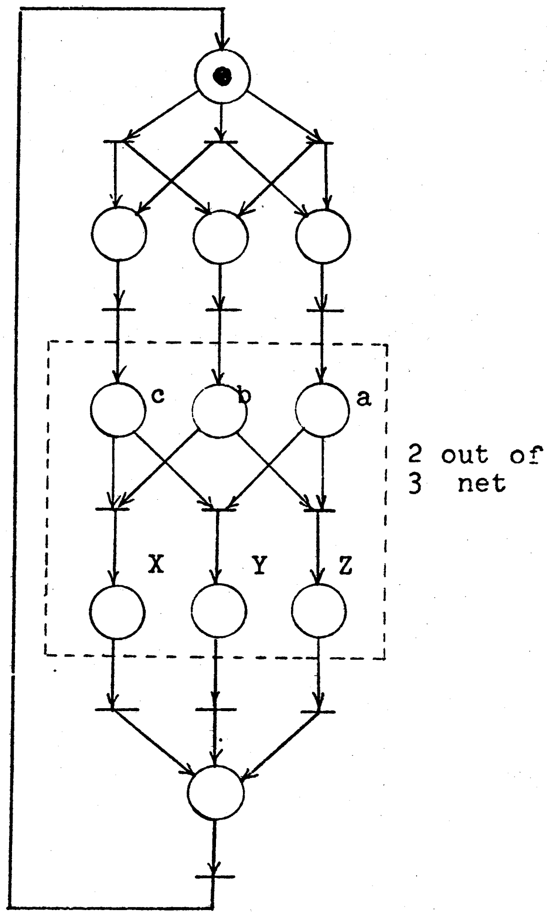
Three smokers X,Y,Z are sitting at the table.

X - with tobacco

Y - with paper

Z - with matches

Each one of them are not allowed to give any ingredient to another. On the table in front of them, two of the three ingredient will be placed, and the smoker who has the necessary third ingredient should pick up the ingredient from the table, make a cigarette and smoke it. Since a new set of ingredient will not be placed until this action is completed, coordination is needed among the smokers.



<u>programs</u>

r_3: P(c)
V(S^c_x)
V(S^y_c)
go to r_3

r_2: P(b)
V(S^x_b)
V(S^z_b)
go to r_2

r_1: P(a)
V(S^y_a)
V(S^z_a)
go to r_1

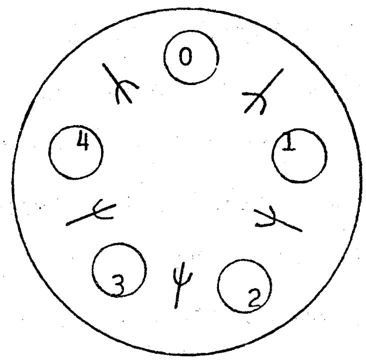
βx: P(X)
V(c)
go to βx

βy: P(Y)
V(b)
go to βy

βz: P(Z)
V(a)
go to βz

```
αx: P(S_b^x)            αy:  .          αz:  .
    P(S_c^x)                 .               .
    P(tx)                    .               .
    if  x > 0  then ( x←x - 1 ;
                      V(xt)
                      go to αx)
    else
    V(tx)
    P(ty)
    P(tz)
    y←y + 1 ;
    z←z + 1 ;
    V(ty)
    V(tz)
    V(S_a^y)
    V(S_a^z)
    V(X)
    go to αx
```

(5) dining philosopher problem [4]



The life of a philosopher consists of an alternation of thinking and eating. Five philosophers are sitting at the table, each philosopher having his own place at the table.

They need to use two forks when eating. Five forks are provided, one between each philosopher's place. The only forks that a philosopher can pick up are those on his immediate right and his immediate left.

The problem is to write a program for each philosopher which will ensure that he contributes at all times to the greatest good of the greatest number.

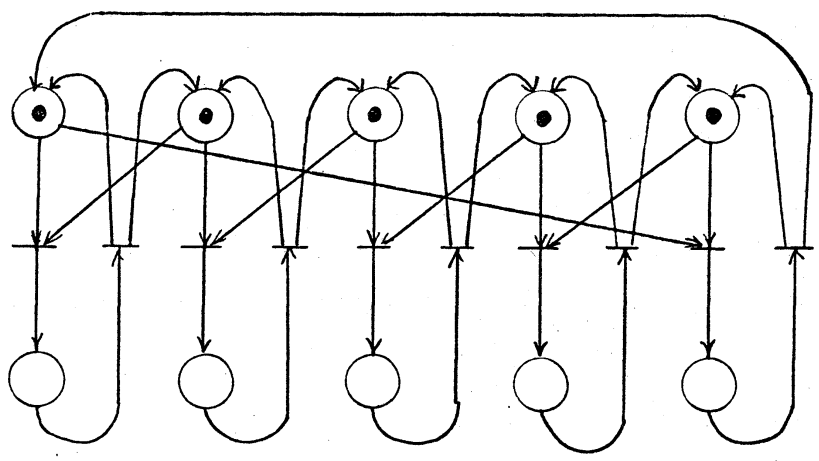
Introduce a state variable "C".

C[i] = 0 : philosopher i is thinking.

C[i] = 1 : philosopher i is hungry.

C[i] = 2 : philosopher i is eating.

```
begin
  semaphore  mutex ; (initially = 1)
  semaphore array  psem[0:4] ; (initially all elements = 0)
  integer array  C[0:4] ;  (initially all elements = 0)
  procedure  test(integer value  k) ;
    begin
      if  C[(k - 1)mod 5]  =  2 and  C[k] = 1 and C[(k + 1) mod 5] = 2 do
        begin
          C[k] := 2 ;
          V(psem[k])
    end; end
  parbegin
     .
     .
    begin
      philosopher i: think ;
                     P(mutex) ;
                     C[i] := 1 ;
                     test(i);
                     V(mutex) ;
                     P(psem[i]) ;
                     eat ;
                     P(mutex) ;
                     C[i] := 0;
                     test((i + 1)mod 5) ;
                     test ((i - 1)mod 5) ;
                     V(mutex) ;
                     go to philosopher i
    end ;
     .
     .
  parend
end ;
```



Proposition

A problem which is represented by a non-simple Petri Net cannot have a solution in a program with P- and V-operations but without any conditional statement.

3-4 Comparison of Various Semaphore Systems

Readers-Writers Problem

(1) Semaphore Application

```
begin
  semaphore  r,w ;
             r := w := 0;
  parbegin
    begin
     "READER  i"
         w:  down  r;
             reading ;
             up  r
    end ;
    begin
     "WRITERS j"
       r: w: down  w ;
             writing ;
             up  w
    end
  parend
end ;
```

(2) Conditional Critical Region [7]

```
begin
  resource  v ;
  integer aw,rr ;
  parbegin
    begin
     READER:  with  v  when  aw = 0
                do  rr := rr + 1;
              reading ;
              with  v  do  rr := rr - 1
    end ;
    begin
     WRITER:  with  v  do  aw := aw + 1 await  rr = 0 ;
              writing ;
              with  v  do  aw := aw - 1
    end
  parend
end ;
```

3-5 Proof of Correctness

(1) Case Analysis [2],[4],[5],[7]

(2) Basic Properties of P- and V-operations [9]

(3) Application of Floyd's Method [10]

4. Resource Allocation Problem

4-1 Resource Allocation [12]

- Reusable Resources

  fixed total number of units in a pool

- Consumable Resources

  no fixed total number of units. If a unit is acquired by a process, the unit ceases to exist. Only a process which is a producer of the resource can release units of the resources. Any released units immediately become available.

- Exclusive Control

  no resource sharing

- Shared Control

  resource sharing

4-2 Deadlocks [11,12,13]

For reusable resource systems without resource sharing

Deadlocks

The situation that co-operating processes prevent their mutual progress even though no single one requests more resources than are available.

Deadlock Strategies

- Detection and Recovery [14]
- Prevention
  - Static Prevention [15,16]
  - Dynamic Prevention [12,13,17,18,19]

1) Detection and Recovery

Deadlocks are detected when they happen. When a deadlock is detected, the system can recover by terminating the deadlocked processes or by pre-empting resources from processes.

2) Prevention

By using some information about users' demand, the system allocates resources so that a deadlock is not possible.

Static — The allocation rule does not depend on the current state of the system.

Dynamic — The allocation rule depends on the current state of the system.

(1) Habermann's Formalization [13]

It utilizes information about maximum claims of all the users.

n : number of processes in the system $(P_1, P_2, \ldots, P_n)$

m : number of the types of resources in the system

$$\underset{\sim}{a} = \begin{pmatrix} a_1 \\ a_2 \\ \cdot \\ \cdot \\ \cdot \\ a_m \end{pmatrix} :$$ available resource vector first prepared in the system; a_i = number of resources of type i

$$\underset{\sim}{B} = (\underset{\sim}{b}_1, \underset{\sim}{b}_2, \ldots, \underset{\sim}{b}_n) = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ \vdots & & & \vdots \\ b_{m1} & b_{m2} & & b_{mn} \end{pmatrix} :$$ user claim matrix

b_{ik} = maximum number of resources of type i that will be needed at one time by process P_k

$$\underset{\sim}{C} = (\underset{\sim}{c}_1, \underset{\sim}{c}_2, \ldots, \underset{\sim}{c}_n) = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ \vdots & & & \vdots \\ c_{m1} & c_{m2} & & c_{mn} \end{pmatrix}$$: allocation matrix

c_{ik} = number of resources of type i allocated to P_k

$\underset{\sim}{a}$, $\underset{\sim}{B}$: constant in time

$\underset{\sim}{C}$: variable in time $\underset{\sim}{C}(t)$

allocation state = ($\underset{\sim}{a}$, $\underset{\sim}{B}$, $\underset{\sim}{C}(t)$)

Allocation state, for which the following three conditions are satisfied, are called realizable states.

R1 : $\forall\, k \quad \underset{\sim}{b}_k \le \underset{\sim}{a}$ (no process claims more resources than are available)

R2 : $\underset{\sim}{C} \le \underset{\sim}{B}$ (no process will try to seize more resources that it has claimed)

R3 : $\sum_{k=1}^{n} \underset{\sim}{c}_k \le \underset{\sim}{a}$ (at most all resources are allocated)

If we use $\underset{\sim}{r}(t) := \underset{\sim}{a} - \sum_{k=1}^{n} \underset{\sim}{c}_k$

R3': $\underset{\sim}{r}(t) \geq \underset{\sim}{0}$

Definition 4.1 (safe state)

A realizable state ($\underset{\sim}{a}$, $\underset{\sim}{B}$, $\underset{\sim}{C}(t)$) is called a safe state if ther is a full sequence S such that

R4 : $\forall\, P_{k \in S} \quad \underset{\sim}{b}_k \le \underset{\sim}{r}(t) + \sum_{S(l) \le S(k)} \underset{\sim}{c}_l(t)$

safe sequence : a full sequence satisfying condition R4

Theorem 4.1

When no process will release its resources until it has been allocated all its claimed resources, the process will not get into a deadlock if and only if the allocation state is safe.

Theorem 4.2

If the allocation state is safe and a subsequence S fulfills condition R4, S can be extended into a safe sequence.

Theorem 4.1 --- n!

Theorem 4.2 --- n^2

Russell[14],Holt[12] --- n (linear algorithm)

(2) Hebalkar's Formalization [17]

It utilizes information about user demand for each job step.

Definition 4.2 (demand graph)

A demand graph is a finite directed graph with arcs and nodes; the nodes are called transitions. Associated with each arc is a quantity called a demand. A quantity called the capacity C is given to the demand graph, and the demands associated with the arcs of a demand graph are always less than or equal to the capacity C.

rectilinear demand graphs

an acyclic demand graphs with the property that the components are unilateral

chain C_i: an unilateral component corresponding to one process i

arc a^i_j : j-th arc on the chain C_i $(1 \leq j \leq p_i)$, where C_i has p_i arcs

initial and terminal arcs : defined as usual

slice S : a set of arcs, one from each C_i

$S \sqcap C_i$: the arc from a chain C_i that goes into a slice S

initial slice S_I : a slice composed only of initial arcs

terminal slice S_T : a slice composed only of terminal arcs

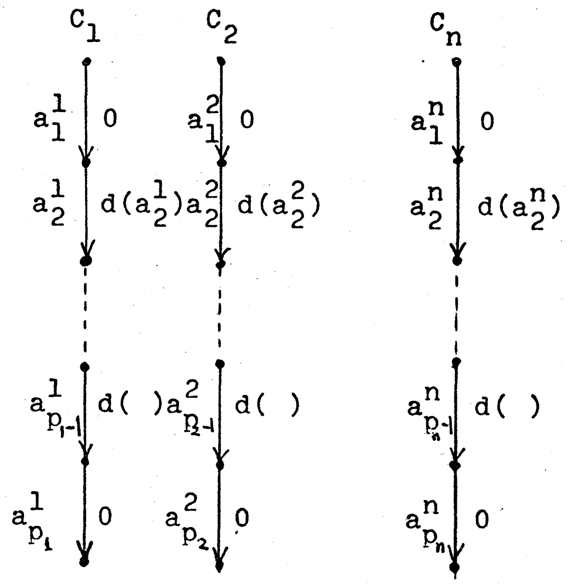
predecessor set of the slice S : transitions that lie above the slice

successor set Suc(S) of the slice S : transitions that lie below the slice

demand $d(a^i_j)$: demand associated with an arc a^i_j

The demand associated with initial and terminal nodes are assumed to be 0.

Example 4.1 (demand graph)



The allocation state is represented by a slice.

move on a chain C_i : $S_i \dashrightarrow S_j$, where S_j is an immediate successor of S_i

Two moves are said to be connected if they can be represented in the form $S_1 \dashrightarrow S_2$ and $S_2 \dashrightarrow S_3$, respectively.

macro-move : a sequence of moves, every pair of which is connected.

uni-chain macro-move : a macro-move all of whose components are moves on the same chain

Definition 4.3

A slice is said to be feasible if the sum of the demands associated with the arcs in it is no greater than C.

Definition 4.4

A feasible slice of a demand graph is safe if there exists a connected sequence of feasible slices from the slice in question to the terminal slice of the graph.

A slice all of whose immediate successors are infeasible represents a state of <u>deadlock</u>.

Let's consider rectilinear <u>scalar</u> demand graphs, (i.e. all the demands are scalar values.)

<u>Safeness Algorithm</u>

Algorithm to test the safeness of a given slice σ .

S: variable to represent slices

$\{C\}$: set of chains of the demand graph

<u>Step 0</u>

Set S equal to σ and $\{C\}$ equal to $\{C_1, C_2, \ldots, C_n\}$. If S is feasible, go to Step 1. If S is infeasible, go to Step 5.

<u>Step 1</u>

Pick a chain from $\{C\}$, and go to Step 2.

<u>Step 2</u>

Attempt to construct a uni-chain macro-move down C_i from S so that the slice resulting from each component move is feasible. Terminate the macro-move at the first point where a slice S' results that satisfies both

$d(S' \bowtie C_i) \leq d(S \bowtie C_i)$

and $d(Suc(S') \bowtie C_i) \not\leq d(S' \bowtie C_i)$.

If such a sequence can be constructed, go to Step 4; if not (i.e. if some move results in an feasible slice), go to Step 3.

<u>Step 3</u>

Delete C_i from $\{C\}$. If $\{C\}$ is now empty, go to Step 5; if not go to Step 1.

<u>Step 4</u>

If S' is not S_T , then replace S by S', set $\{C\}$ equal to

$\{C_1, C_2, \ldots C_n\}$ and go to Step 1. If S' is S_T, then stop. (σ is safe.)

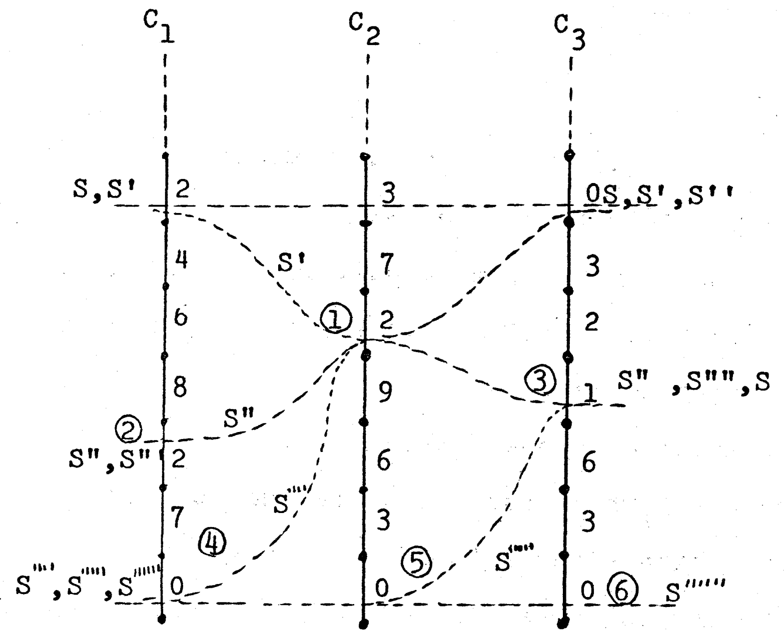
Step 5

Stop with failure. (σ is unsafe.)

Theorem 4.3

A feasible slice σ of a demand graph D is safe if and only if the Safeness Algorithm terminates successfully when applied to σ and D.

Example 4.2



rectilinear vector demand graph

demand graph with loops and decisions

4-3 Effective Deadlocks

Assume that each of the processes iterates its loop indefinitely. Assume also that the resource scheduler continues granting requests in the queue as long as the requests are safe. Then, there is a possibility that the progress of some processes are delayed indefinitely even though deadlock danger does not exist.

This situation is called <u>effective deadlock</u> (<u>permanent blocking</u>[20], <u>indefinite postponement</u>[21], <u>individual starvation</u> [22]).

<u>Prevention of Effective Deadlocks</u>

(1) Holt [20]

<u>array</u> t[1:n],u[1:n] ;

t[i] : When P_i requests, t[i] is set to the time of the request. When P_i releases, t[i] is set to -1.

u[i] : the maximum time process P_i must wait for a request before the resource scheduler will activate a special strategy.

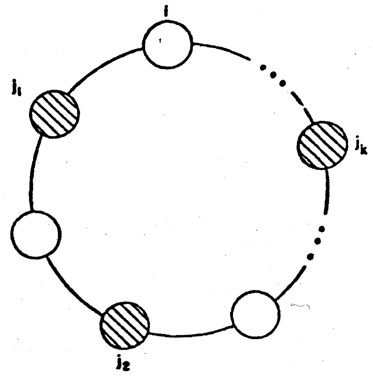
<u>if</u> t[i] = -1 <u>then</u> waittime := 0
<u>else</u> waittime := now - t[i]

If there is any process P_i which has waited beyond its maximum time, the scheduler must activate the following strategy.

(a) All safe requests by processes having nonzero allocations should be granted. This continues until enough resources are available so that the request by P_i is safe. Process P_i is then granted its requests.

(b) Each process other than P_i is examined to see if its maximum waiting time has been exceeded. If so, the process which has waited longest beynd its maximum waiting time is designated process P_i and this strategy is repeated by returning to (a).

(2) Saito [23]



When all the processes are totally ordered based upon the values of the maximum demands, make a directed cycle C whose n-th node is labelled by the n-th process identifier in the ordered set.

The Allocation Strategy

When the process P_i releases several resources:

(a) Search the next waiting process P_j along the cycle C starting from i. Stop when there is no waiting process along this cycle C before P_i appears again.

(b) Pre-allocate resources to P_j as much as possible so long as the allocation state is safe.

(c) Stop when there is no resource left. Otherwise, let P_j be P_i, and go to (a) to repeat it.

(3) Dijkstra [22]

Prevention of individual starvation in the dinig philosopher problem

5. Concluding Remarks

(1) Formal Proof Method for Synchronizing Processes

(2) Implementation of Deadlock Prevention Algorithm by using Synchronizing Processes and the Proof of its Correctness

(3) Structure of Synchronizing Processes

References

[1] Holt,A. and Commoner,F. Events and Conditions, Record of the Project MAC Conference on Concurrent Systems and Parallel Computation,ACM, 1970

[2] Dijkstra,E.W. Co-operating Sequential Processes, Programming Languages, edited by Genuys,F., Academic Press, New York,1968

[3] Patil, S.S. Limitations and Capabilities of Dijkstra's Semaphore Primitives among Processes, Computation Structures Group Memo 57, Project MAC, MIT, Feb. 1971

[4] Dijkstra,E.W. Hierarchical Ordering of SequentialProcesses, Acta Informatica, Vol.1, No.2, pp115-138, 1971

[5] Woden,P.L. Still Another Tool for Synchronizing Co-operating Processes, Carnegie-Mellon University(AD-750 538),Aug.1972

[6] Hoare,C.A.R. Towards a Theory of Parallel Programming, Operating Systems techniques, edited by Hoare,C.A.R. and Perrott, R.H., Academic Press, New York, 1972

[7] Hansen,P.B. A Comparison of Two Synchronizing Concepts, Acta Informatica, Vol.1,No.3, pp 190-199, 1972

[8] Courtois,P.J., Heymans,F. and Parnas,D.L. Concurrent Control with "Readers" and "Writers", CACM, Vol.14, No.10, pp 667-668, Oct. 1971

[9] Habermann,A.N. Synchronization of Communicating Processes, Porc. 3rd ACM Symposium on Operating Systems Principles, Oct. 1971

[10] Levitt, K.N. The Application of Program-Proving Techniques to the Verification of Synchronization Processes, Proc. FJCC, pp 33-47, 1972

[11] Coffman,E.G., Elphick,M.J. and SHoshani,A. System Deadlocks,

Computing Surveys, Vol.3, No.2, pp 67-78, June 1971

[12] Holt, R.C. Some Deadlock Properties of Computer Systems, Computing Surveys, Vol.4, No.3, pp 179-196, Sep. 1972

[13] Habermann, A.N. Prevention of System Deadlocks, CACM, Vol.12, No.7, pp 373-377, July 1969

[14] Russell, R.D. A Model for Deadlock-Free Resource Allocation, CGTM #93, SLAC, Stanford University, June 1970

[15] Havender, J.W. Avoiding Deadlock in Multitasking Systems, IBM Systems J., Vol.7, No.2, pp 74-84, 1968

[16] Murphy, J.E. Resource Allocation with Interlock Detection in a Multitask System, Proc. FJCC, pp 1169-1176, Dec. 1968

[17] Hebalkar, P.G. Deadlock-Free Sharing of Resources in Asynchronous Systems, MAC-TR-75, MIT, Sep. 1970

[18] Shoshani, A. and Coffman, E.G. Sequencing Tasks in Multiprocess Systems to Avoid Deadlocks, IEEE Conf. Record on 11th Annual Symposium on Sw. & Aut. Th., pp 225-235, Oct. 1970

[19] Fontao, R.O. A Concurrent Algorithm for Avoiding Deadlocks in Multiprocess Multiple Resource Systems, Proc. 3rd ACM Symp. on Operating Systems Principles, Oct. 1971

[20] Holt, R.C. Comments on Prevention of System Deadlocks, CACM, Vol.14, No.1, pp 36-38, Jan. 1971

[21] Parnas, D.L. and Habermann, A.N. Comments on Deadlock Prevention Method, CACM, Vol.15, No.9, pp 840-841, Sep. 1972

[22] Dijkstra, E.W. A Class of Allocation Strategies Inducing Bounded Delays Only, Proc. SJCC, pp 933-936, 1972

[23] Saito, N. Prevention of Effective Deadlocks, 13th National Convention Records of Inf. Proc. Soc. of JAPAN, 1972