CH6 Process Synchronization

Why need process synchronization?

Consider two cooperating process:

Producer:

```
item nextProduced;
        while (1) {
                while (counter == BUFFER_SIZE)
                                ; /* do nothing */
                buffer[in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                counter++;
        }
```

Consumer:

```
item nextConsumed;
        while (1) {
                while (counter == 0)
                                ; /* do nothing */
                nextConsumed = buffer[out];
                out = (out + 1) % BUFFER_SIZE;
                counter--;
        }
```

Execution sequence: initially, counter=5

________ __________ _________ _________ 5 different
________ __________ _________ _________ 4 counter?!!
________ __________ _________ _________ 6 values

A “cooperating” process is one that can affect or be affected by the other processes executing in the system.
(可能 share data or logical address space => data consistency)

register1=counter
register1= register1 +1
counter= register1
如果這 3 個指令不可分=>則不會出事！

(p.156)

producer: **register1 = counter** (*register1 = 5*)
producer: **register1 = register1 + 1** (*register1 = 6*)
consumer: **register2 = counter** (*register2 = 5*)
consumer: **register2 = register2 – 1** (*register2 = 4*)
producer: **counter = register1** (*counter = 6*)
consumer: **counter = register2** (*counter = 4*)

A situation like this, where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

The Critical –Section problem (臨界區)

- Def: Design a protocol that processes can use to cooperate.
  Where each process has a segment of code, called a critical section, whose execution must be mutually exclusive
- A general structure

```
do {
    entry section;        ~permission request
    critical section;

    exit section;         ~exit notification

    remainder section;
} while(1);
```

- Solution: Three requirements:
  1. Mutual Execution: only one process can be in its critical section (互斥，有你就沒有我)
  2. Progress: (No deadlock)
     a. Only processes not in their remainder section can decide which will enter its critical section
     b. The selection cannot be postponed indefinitely
  3. Bounded Waiting: (no starvation)
     A waiting process only wait for a bounded number of processes to enter their critical sections

  race condition : the outcome of the execution depends on the particular order in which the access take place.

An OS is composed of several concurrent processes.

Processes are concurrent if they exist at the same time.

These processes have to exchange information for their proper function.

Data sharing is one method of information exchange.

Uncontrolled access to shared data by concurrent processes can make the data inconsistent.

Examples of data sharing:

--Memory Management:
Processes: User processes, Garbage Collector.
Shared data: memory allocation table.
--UNIX Shell Pipelining:
cat file1 file2 | lpr
shared data: the printer buffer
one is writing while one is reading.

➔ the producer-consumer problem
(the bounded buffer problem).

Access to shared data must be done in an orderly fashion.

* Mutual Exclusion→each process accessing the shared data excludes all others from doing so simultaneously.

Only one process at a time holds a resource or modifies shared information.

A mechanism for mutual exclusion must guarantee three properties: mutual exclusion , deadlock-free ,starvation-free.

* Critical Section: a code segment in a process in which some shared resource is referenced.

  During the execution of a CS, mutual exclusion with respect to certain information or resources must be ensured.

* If no synchronization, what will happen?

  EX. X=X+1; initially, X=5; processes A and B read X at the same time; after execution, X(A)=X(B)=6.

* Dijkstra introduced semaphores for this purpose.

* A synchronization mechanism must provide a means of expressing exclusion and priority.

  Exclude certain process from the resource under some circumstances.

  Scheduling access to the resource according to given priority.

* Synchronization mechanisms:

  1) busy waiting.(CPU is active→ waste CPU time)
  2) block and sleep; semaphore.(CPU is passive)
  3) event counter
  4) monitor
  5) message passing.(send/receive)
  6) serializer.
  7) Path expression.

* Examples

  1) the bounded buffer problem (producer/consumer).
  2) the reader/writer problem.
  3) The disk-head scheduler

臨界段落

定 義

針對任何共用資源，每一程序中有一段程式碼可以管理此共用資源。如果已有一個程序已進入此管理共用資源的程式段時，則其他的程序就不能使用此共用資源，即其他程序不能進入管理此共用資源的程式段。此程式段的執行對所有程式而言是互斥的。因此此程式段稱為<u>臨界段落</u>。

臨界段落的條件

1. 互斥性：僅能有一個程序進入臨界段落。
2. 進展性（progress）：當在臨界段落的程序離開了，則必有一個等待進入的程序能進入臨界段落。(No deadlock)
3. 有限等待（Bounded waiting）：所有等待進入臨界段落的程序皆能在有限的時間內進入臨界段落。 (No starvation)

臨界段落的結構

如圖 7-3 所示臨界段落分有入口區，臨界段落本身，出口區。期中入口區與出口區的設計是為了滿足臨界段落的條件。

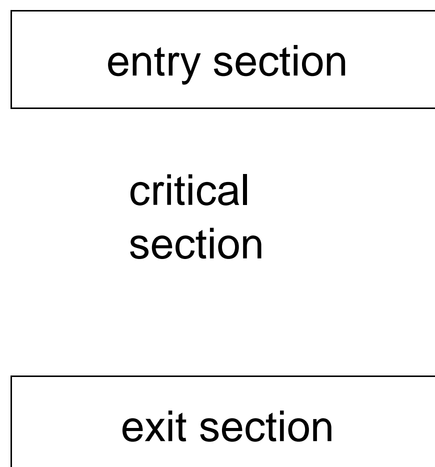


圖 7-3 臨界段落的結構

6.2 Two-Process Solution

– Pi & Pj ; where j=1-i; (P_0 , P_1)

– Assumptions

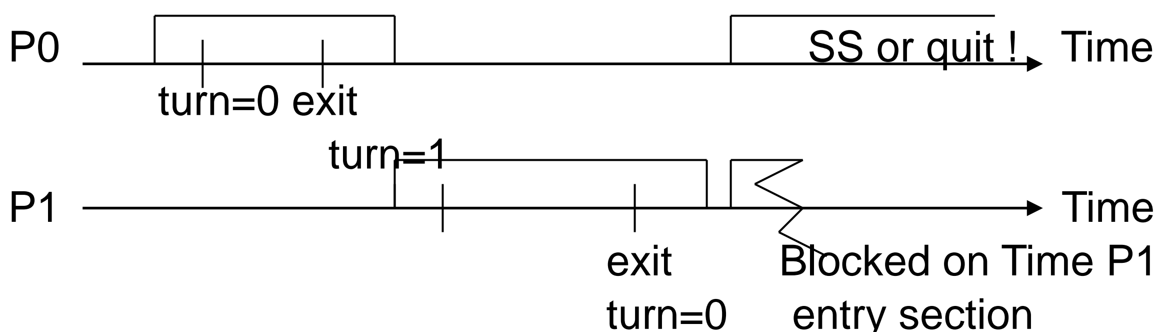
  Every basic machine-language instruction is atomic

Algorithm 1

Idea: Remember which process is allocated to enter its critical section, i.e., process I can enter its critical section if turn = i (輪到我，我進去)

```
do {
        while ( turn != i) ;
            critical section
        turn =j ;  （下一個人）
            remainder section
} while (1) ;
```

* Fail the progress requirement:



得證明 no progress

i→j→i→j (no problem)

i→i (有問題 ➔ 誰都進不去)

turn=0

i = 0 while(t != 0)	i = 1 while (t != 1)

Algorithm 2

Idea: Remember the state of exch process

flag[i] = true => Pi is ready to enter its critical section

```
do {
        flag[i] = true;          - (1)
        while ( flag[j] );       - (2)

        critical section

        flag[i] =false;

} while(1);
```

* Fail the progress requirement (兩人均進不去)

when

flag[0]=flag[1]=true;

=> The correctness of Algorithm 2 depends on the exact timing of the two processes!

Alg.1 問題在它沒有足夠 interaction about the state of each processes.

* flag：shared variable

* 0→1→0→1 (ok)

0→0→0 (ok)

* if (1),(2) 順序交換 ➔ 連 ME 均不滿足

flag[0] = T; while (flag[1]);	flag[1] = T; while (flag[0]);

Algorithm 3

Idea: Combine ideas of Algorithm 1 & 2

fag[i]
turn=i
=> Pj must wait

```
do {
    flag[i] =true;
    turn =j;
    while (flag[j] && turn == j );
```

(* 只要別人不想盡去，我就可以進行；破其一條件即可進去*)

```
        critical section

    flag[i] =false;

        remainder section
} while(1);
```

Initially, flag[i]=flag[j]=false, and
turn = 0 or 1

- mutual exclusion: turn can be either 0 or 1
- progress: a process can only be stucked in the while loop & processes which can stuck it must be in their critical sections
- bounded waiting: at most one entry by the other processes

＊turn：shared variable

i	flag[j]	Turn=?
ID=0	T	0：我進去 1：我不可
	F	0, 1 → 我去
ID=1	T	0：不可 1：進去
	F	0, 1 → 可

只要某一為 False 即可進去！

P0	P1
(1)flag[0] = T	
	(2) flag[1] = T
	(3) turn = 0
(4) turn = 1	
(5) while (T && turn == 1) waiting	
	(6) while (T && turn==1) False CS
	(7) flag[1] = F
(8) while (F && turn == 1) CS	

- Multiple-Process Solution

  Idea: Processes which are ready to enter their critical section must take a number and wait till the number becomes the lowest
  (*沒人號碼比我小 ➔ 我進去*)
  number[i]: Pi's number if it is nonzero
  choosing[i]:Pi is taking a number(等他一下)

```
do {
    choosing[i] =true;
    number[i]=max(number[j])+1;
    choosing[i] =false;
    for ( j =0 ; j<n ; j++)
    {
       while ( choosing[j] );
       while ((number[j] !=0) && (number[j,j] < number[i, i] ));
    }

        critical section

      number[i] =0;

        remainder section
} while(1);
```

* An observation: If Pi is in its critical section, and Pk (k !=i) has already chosen its number[k], then number[i,i] < number[k,k]

軟體的臨界段落設計

Dekker algothm

Dekker algothm 是 2 個 processes 的臨界段落設計方法，其方式為：

The two processes, p_0 and p_1 , share the following variables:

var flag: array[0..1] of boolean ; (initially false*)*

turn: 0..1;

The program below is for process p_i (I=0 or 1) with process p_j (i=1 or 0) being the other one.

The structure of process p_i is:

```
do {
    choosing[i] =true;
    number[i] =max(number[0],number[1],…,number[n-1])+1;
    choosing[i] =false;
    for (j =0 ;j<n ;j++) {
                while ( choosing[j] );
                while (( number[j]!=0 )
                    && ( number[j,j] < number[i,I]));
            };

        critical section

    number[i] =0;

        remainder section
} while(1);
```

bakery 演算法

bakery 演算法是 n 個 processes 的臨界段落設計方式，其方式為：

The common data structures are:

Var choosing: array[0..n-1] of boolean ;
number: array[0..n-1] of integer;

Initially these data structures are initialized to false and 0 , respectively. For convenience, we define the following notation:

- (a,b)<(c,d) if a<c or if a=c and b<d.
- max(a_0,…,a_{n-1})is a number, k ,such that k≧ a_i for i=0…,n-1

```
do {
    flag[i] =true;
    while ( flag[j] )
        { if ( turn == j )
            {
                flag[i] =false;
                while (turn == j );
                flag[i] =true;
            }
    …
    critical section
    …
    turn =j;
    flag[i] =false;

    …
    remainder section
} while(1);
```

例題 若有二個 processes 內有下列的臨界段落。其中 turn 為 common integer variable ，其值為 0 或是 1，初值為 0。
即如果 turn= i ，則程序 P_i 被允許進入臨界段落。

```
do {
    while ( turn != i );

        critical section

    turn =j;

            remainder section
} while(1);
```

試問此軟體設計方式是否滿足臨界段落條件。

當 turn = 0 ，而此時 p_1 想進入臨界段落，則表示當 p_0 不在臨界段落時，p_1 亦不能進入，因此不滿足 prgress 的條件。

Synchronization Hardware

- Motivation: make programming easier and improve system performance
- Approach:

- Disable interrupt

  Protect code where shared variables are modified!

  *Infeasible in multiprocessor environment where msg passing is used

  *Potential impact on interrupt-driven system clock

- Test and set(Hardware support?)

  An atomic instruction

```
boolean Test-and-set( boolean &target)
        {
            Test-and-set =target; (*第一個人，我一進門
            target =true;             ，就把門鎖上*)
        }
```

```
Boolean lock=false;
do {
  while (TestAndSet(lock));

       critical section

  lock =false;

    remainder section
}
```

➔ Starvation

swap(Hardware support?!)

An atomic instruction

```
void Swap(boolean &a, boolean &b) {
  Boolean temp =a;
  a =b;
  b =temp;
}

do {
  key =true;
  while ( key==true)
     swap(lock, key);

   critical section

  lock =false;

   remainder section
}
```

- lock : global variable i false initially
- key : local variable

➔starvation (no bounded waiting)

— one correct algorithm

```
shared variables (* global i false initially *)
   boolean lock;
   boolean waiting[n];

   var j:0..n-1;     (* local *)
        boolean key;

   do {
          waiting[i] =true;
          key =true;
          while ( waiting[i] && key )
                    key =TestAndSet(lock);

          critical section

          j =(i+1) % n;
          while ((j !=i ) && (not waiting[j] ))
                   j = (j+1) % n;
          if (j==I)   lock =false;
                    else waiting[j] =false;
          waiting[i] = false; (*)

           remainder section
   } while(1);
```

* Atomic Test-and-set is hard to implement in a multiprocessor environment

➔ 解掉 Starvation

硬體的臨界段落設計

若有一個 atomic instruction Test-and-set 定義如下：

```
boolean TestAndSet(boolean &target)
    {
        boolean    rv=target;
        target =true;

        return rv;
    }
```

則下列的臨界段落設計可滿足臨界段落的三條件

The common data structure are:

```
boolean waiting[n];
boolean lock;
```

These data structure are initialized to false.
The structure of process P_i is :

```
var j: 0..n-1;
     boolean key;
do {
  waiting[i] = true;
  key =true;
  while ( waiting[i] &&
     key==TestAndSet(lock));
  waiting[i] =false;

        critical section

  j =(i+1) % n;
  while ((j!=i) && (not waiting[j])) j =(j+1)% n;
  if (j==I)    lock =false;
  else waiting[j] =false;

        remainder section
} while(1);
```

NSYSU CSE DB

例題　若有 n 個 processes 具有下列型態的臨界段落,其中布林變數 lock 被設定為 false,

```
do {
    while (TestAndSet(lock));
     critical section
    lock =false;
        remainder section
 }
```

試問此設計能否滿足臨界段落條件。

由於所有等待進入臨界段落的程序可能搶著執行 atomic instruction Test-and-set,因 atomic instruction 是由硬體結構完成,故在一個指令週期內就能完成。故而有些程序可能一直搶不到執行,而造成 starvation 的現象,因此不能滿足”bounded waiting”的條件。

§6.4 Semaphores

- Motivation: A high-level solution intended for more complex problems (* a new data structure *)
- Two atomic operations

```
wait(s):              /* P */
   while s<=0 do no-op;
   s--;
signal(s);            /* S */
   s++;
```

- Usage:
  - critical section problems (* S=1 initially *)

```
   do {                              for ME
       wait(mutex);

        critical section

       signal(mutex);

        remainder section

    }while(1);
```

  - procedure enforcement (* S=0 initially *)

```
    P1:                           兩人同步
              S1;
              signal(synch);
    P2
              wait(synch);
              S2;
```

- Implementation

– Spinlock～a type of semaphore involving busy waiting such as wait(s):
while s<=0 do no-op; ←CPU cycles wasted!

- advantage: when locks are held for short time ,it is useful since no context switching is involved

– Block-waiting～no busy waiting from the entry to the critical section only! (* block itself → 用一個 waiting queue, control 交給 CPU)

Define a semaphore as a record

```
typedef struct {
    int value;
    struct process *L;
    } semaphore;

wait(s):
   S.value--;
   if (S.value<0) {
      add this process to S.L;
      block; ←system call
   }
signal(s):
       S.value ++;
       if ( S.value<=0 ) { (* 表示有人 waiting *)
               remove a process P from s.L;
               wakeup(p); ←system call
        }
```

- Queueing strategy can be arbitrary, but there is a restriction for the bounded-waiting requirement (no starvation)
- Mutual exclusion in wait() & signal()

  ~ uniprocessor

    ~interrupt

    ~test-and-set

    ~swap

    ~software methods in section 6.2

    ~and more

  ~multiprocessor
- Remarks: busy-waiting is limited to only the critical sections of the wait() & signal()

*6.4.3 Deadlock

A set of processes is in a deadlock state when every process in the set is waiting for a event that can be caused only by another process in the set

```
P0                          P1
wait(S);                    wait(Q);
wait(Q);                    wait(S);

signal(S);                  signal(Q);
signal(Q);                  signal(S);
```

starvation (Indefinite blocking)

processes wait indefinitely(within a semaphore)

~e.g., the queuing mechanism is by LIFO

* Semaphore

A semaphore is an integer variable S and an associated group of waiting processes (i.e., a waiting queue) upon which only two operations may be performed:

1) P(S): if S $\geqq$ 1 then S =S-1
   else the execution process places itself in S's waiting group and relinquishes the CPU by invoking CPU scheduler
   endif;
2) V(S): if S's waiting queue is nonempty then remove one waiting process and make it available for execution (some implementations invoke a CPU scheduler)
   else S =S+1
   end if;

*Let S be a boolean semaphore. Initialization:: S =false.

Two operations P and V are defined on S:

P(S):: if S then block; S =true; (* wait *)
V(S):: S =false; unblock; (* signal *)

*Generalized(integer) semaphores
Let S be an integer semaphore.
P and V operations are defined as follows:

```
P(S):: S =S-1;
       if (S<0) block;
V(S):: S =S+1;
       if (S<=0) unblock;
```

*To mutual exclusion access to a shared resource:

1) allocate an integer semaphore with the resource.
   Initialization ::S =1.

2) perform a P operation before using the resource.
   (* lock *)
3) perform a V operation to release the resource.
   (* unlock *)

*Example:

Sa = Sb = (mutual exclusion semaphore)=1
(* the first person can access; for mutual exclusion *)
(* enter the same procedure *)

S1 (resource semaphore)=0
(* to suspend; to block; to wait; between procedures *)

```
procedure 1;                 procedure 2;
P(Sa);                        P(Sb);
If.. then P(S1);              if.. then V(S1);
(* to wait *)                (* to signal *)
V(Sa);                         V(Sb);
```

*Consumer<-->Producer (many to many)
S1=n (space); (* for wait/signal *)
S2=0 (source); (* for wait/signal *)
Sa=Sb=1; (* for mutual exclusion *)

```
Producer aa                  consumer bb
P(Sa);..                     P(Sb);..
if …then P(S1);              if … then P(S2);
…                            …
if … then V(S2);             if … then V(S1)
… V(Sa);                     … V(Sb);
```

*6.4.4 Binary Semaphore

Its value ranges from 0 to 1

bounded value range
=>easy to implement!

– Implement a counting semaphore by binary semaphore

```
Var
        binary semaphore S1=1,S2=0;
        int c;
wait(S):
      wait(S1);          /* protect c */
      c--;
      if (c<0) {
              signal(S1);
              wait(S2);
      }
      signal(S1);

signal(S);
      wait(S1);
      c++;
      if (c<=0)   signal(S2);/*wakeup processes queued in S2.L*/
      else   signal(S1);
```

信號機 （semaphore）

P 與 V 運算的定義

若共用變數 S 用來做計數用，S 稱為 Semaphore，P 與 V 運算定義如下：

P(S): while S≦0 do skip;
S:=S-1;

V(S): S:=S+1;

臨界段落的設計

若有 n 個 processes 共用一個 semaphore 為 mutex，其初值為 1，則每個 process 皆有臨界段落設計如下：

```
do {
    P( mutex );

        critical section

    V(mutex);

        remainder section
}
```

§6.5 Classical Synchronization Problems

The Bounded-Buffer Problem

Producer, consumer, and a pool of buffers

empty =n; full =0;

mutex =1; (* for ME *)

Producer

```
do {
        …
        produce an item in nextp;
        …
        wait(empty);<--control buffer availability
        wait(mutex);<--mutual execution
        …
        add nextp to buffer
        …
        signal(mutex);
        signal(full);
} while(1);
```

Consumer

```
 do {
        wait(full);
        wait(mutex);
        …
        remove an item from buffer to nextc;
        …
        signal(mutex);
        signal(empty);
        …
        consume the item in nextc;
        …
  } while (1);
```

*Common Synchronization Problems

1) mutual exclusion: a signal semaphore.
2) Producer/consumer: a set of producer processes supplies messages to a set of consumer processes.

They all share a common pool of spaces into which messages may be placed by producers or removed by consumers.
-->a circular buffer and semaphores.

(nrfull/nrempty-->the no. of full/empty buffers; prevent a producer from overwriting a message or a consumer from obtaining an already used message.)

If only one producer or consumer exists, then the semaphore Sa or Sb unnecessary.

3) reader/writer: any number of readers should be allowed to proceed concurrently in the absence of a writer, but only one writer may execute at a time while readers are excluded.(one writer or many readers)

Several ways to handle priority:
-FCFS
-a strong reader preference.
-a weak reader preference.
-a strong writer preference.

*The mutual exclusion problem using semaphores

```
Shared Variable
var S: semaphore:=1;
Process i                              Process j
loop                                   loop
    …                                      …
    P(S);                                  P(S);
    Access shared data safely          Access shared data safely
    V(S);                                  V(S);
    …                                      …
 endloop
```

*The Producer/Consumer problem using semaphores

```
Shared Variables
var nrfull: semaphore:=0;
    nrempty: semaphore:=N;
    mutualP: semaphore:=1;
    mutualC: semaphore:=1;
    buffer: array[0..N-1] of message;
    in, out:0..N-1 :=0,0;
Producer i                         Consumer j
loop                               loop
  …                                  …
  Create a new message m;             …
  __one producer at a time;        __one consumer at a time;
  P(mutualP);                      P(mutualC);
  __Await an empty cell;            __Await a message;
  P(nrempty);                      P(nrfull);
  Buffer[in]:=m;                    m:=buffer[out];
  in:=(in+1)mod N;                  out:=(out+1)mod N;
  __signal a full buffer;            __signal an empty buffer;
  V(nrfull);                         V(nrempty);
  __Allow other producers;          __Allow other consumers;
  V(mutualP);                      V(mutualC);
  …                                 Consume message m;
  …                                 …
  endloop;                           endloop;
```

信號機的實際應用

生產/消費（producer/consumer）問題

有一個 buffer，共有 n 個位置。兩個 processes 使用此 buffer，其中之一產生資料到此 buffer，另外一個 process 則從 buffer 取出資料。

使用的 semaphore 如下：

1. mutex：作為存取 buffer 的互斥 semaphore，其初值為 1。
2.empty 與 full：當作計算 buffer 的空位置數與滿的位置數，empty 的初值為 n，而 full 的初值為 0。

對於 producer 與 consumer 的程式可編寫如下：

```
type item =… ;
    var buffer=… ;
    full, empty, mutex: semaphore;
    nextp, nextc: item;
    {
        full =0;
        empty =n;
        mute =1;
        parbegin
            producer :repeat
                                        …
                                    produce an item in nextp
                                        …
                                    p(empty);
                                    p(mutex);
                                        …
                                    add nextp to buffer
                                        …
                                    V(mutex);
                                    V(full);
                                 until false;

          consumer :repeat
                                    P(full);
                                    P(mutex);
                                      …
                                    remove an item from buffer to nextc
                                      …
                                    V(mutex);
                                    V(empty);
                                      …
                                    consume the item in nextc
                                      …
                                until false;
            parend;
      }
```

- The Readers and Writers Problem

Readers: Processes only read the shared object
Writers: else
Access Rules:

2. Multiple reads can occur simultaneously
3. Every write is exclusive in accessing the shared object

– The first reader-writers problem: (* strong reader *)

No readers will be kept waiting unless a writers has already obtained permission to use the shared object =>potential hazard to writers! (* 除非有 acting writer, 否則 no waiting reader *)

– The second reader-writers problem: (* strong writer *)

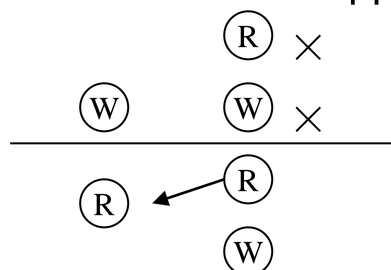
Once a writer is ready, it performs its write as soon as possible !(No new read will be allowed) => potential hazard to readers ! (* 只要有 waiting writer, new reader waits *)

– many others !

– A solution to the R-W Problem: (* Weak reader *)

```
semaphore wrt, mutex; (initialize to 1)
integer readcount; (initialize to 0)

writer:
   wait(wrt);
   …
   writing is performed
   …
   signal(wrt);

Reader
   wait(mutex); (* (n-1)個 reader 可能 block 在此 *)
   readcount++;
   if (readcount==1) wait(wrt); <-- only one reader pending
                                      here!
   (* 第一個 reader 去與 writer 爭 *)
signal(mutex);
…
   reading is performed
…
wait(mutex);
readcount--;
if (readcount==0 ) signal(wrt); (* 最後一個 reader 去叫
signal(mutex);                        writer *)
```

*Queuing mechanisms decide whether a waiting reader or a writer resume execution once a appropriate semaphore is signaled!



讀寫（reader/writer）問題

有一個 data object (可能是 file 或 record)被多個 processes 同時使用，有些 processes 僅是作 read 的工作，另外一些則是作 write 的處理。問題條件是只要有一個 process 再作 read，則所有 writer processes 皆不能使用此 data object，但允許其他的 reader processes 可使用此 data object。然而若是有一個 writer process 在使用此 data object 時，其他的 processes（無論是 reader 或 writer）皆不能使用此 processes。

使用的變數如下：

1. readcount :為 reader processes 的共同變數，紀錄使用 data object 的 reader processes 數。初值為 0。
2. mutex:為一個 semaphore，保證 read count 在被使用時能具有互斥性。
3. wrt:作為 writer processes 的互斥 semaphore。

Reader 與 Writer 的程式結構如下：

```
writer :
    p(wrt);
            …
          writing is performed
    V(wrt);

reader :
    P(mutex);
        readcount++;
        if (readcount==1) P(wrt);
    V(mutex);
            …
        reading is performed
            …
    P(mutex);
        readcount--;
        if (readcount==0) V(wrt);
    V(mutex);
```

∗ The Reader/Writer Problem

1) Weak reader priority: an arriving writer waits until there are no more active readers.
When waiting occurs (writer is inside, and readers and other writers are waiting) →
FCFS (first reader compete with writers).

2) strong reader priority: conditions of weak reader priority solution apply, but also a waiting reader has priority over a waiting writer.
When waiting occurs, waiting reader has high priority. (all readers have a higher priority than a writer, regardless of the order of their arrival.)

3) writer priority: an arriving reader waits until there are no more active or waiting writers.
When waiting occurs, waiting writer has higher priority. Moreover, when waiting occurs, waiting readers wait until waiting writers have finished.

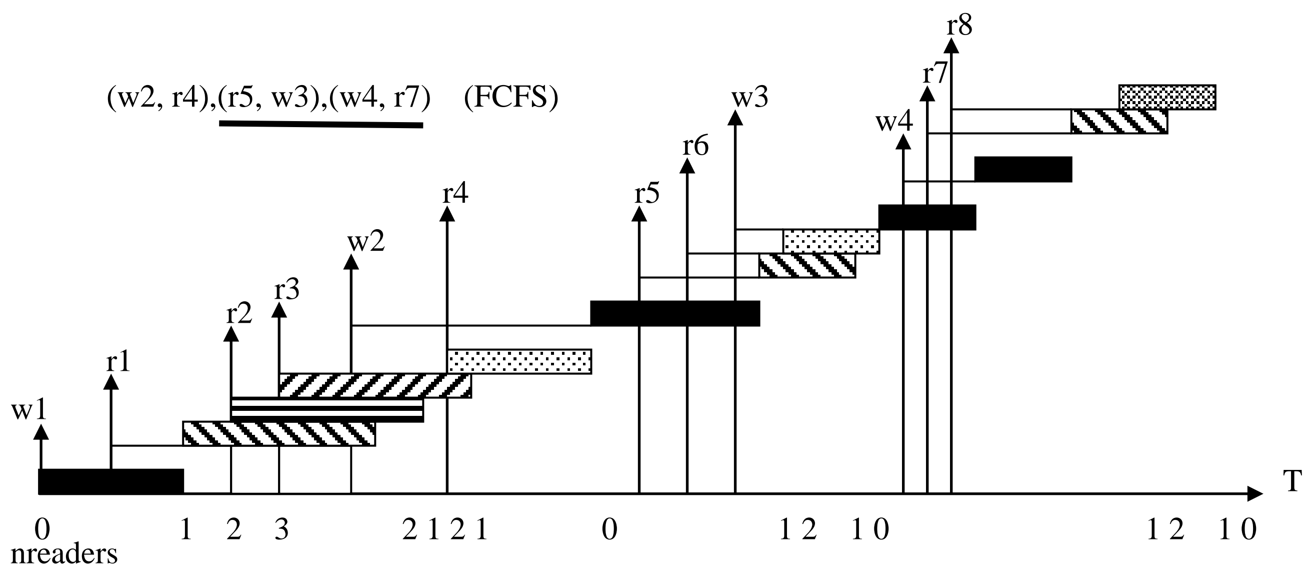


Fig. A weak reader preference solution

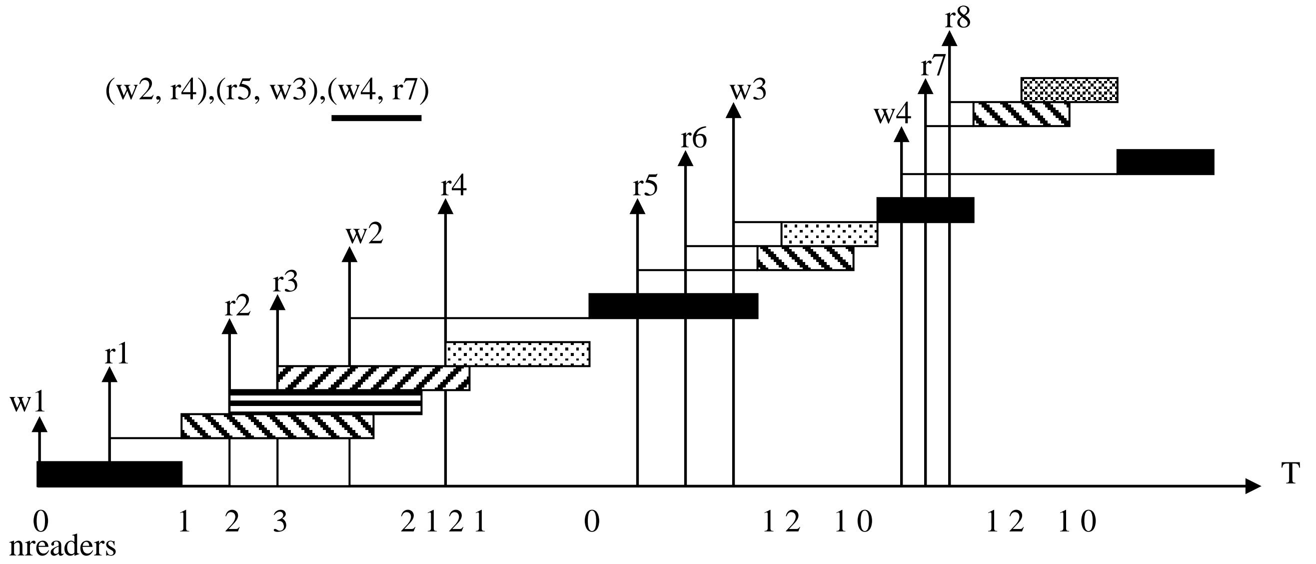


Fig. A strong reader preference solution

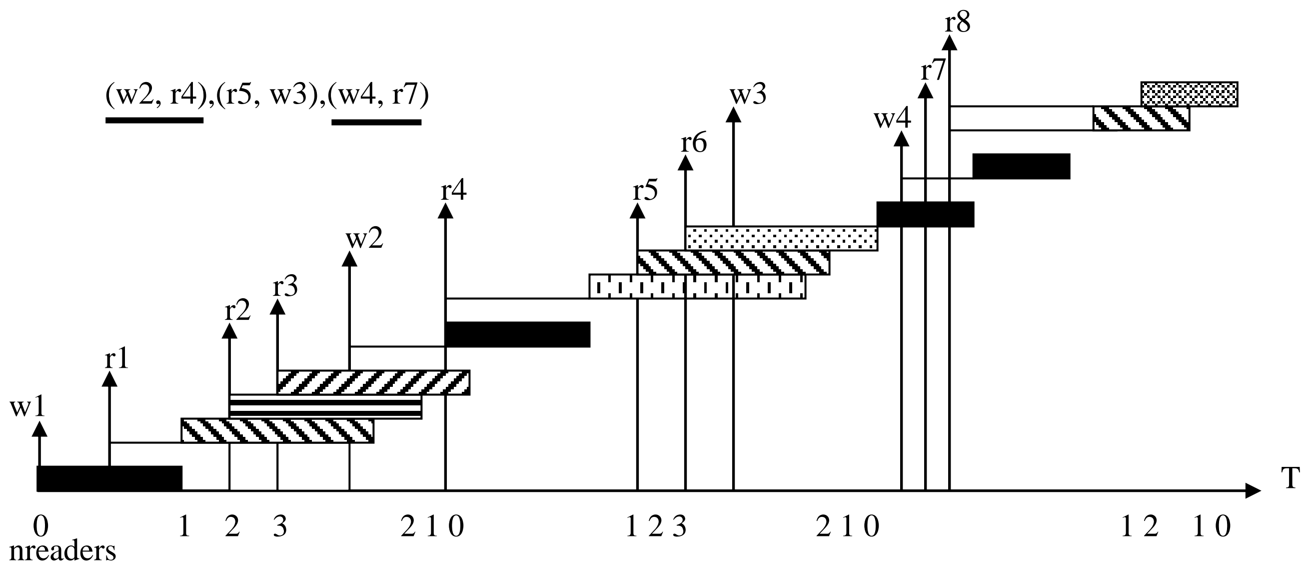


Fig. A strong writer preference solution

*A weak reader preference solution using semaphores

Shared Variables

semaphore wmutex=1, rmutex=1;

integer nreaders =0;

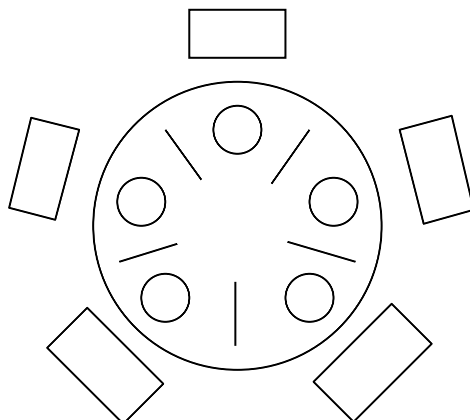
A Reader

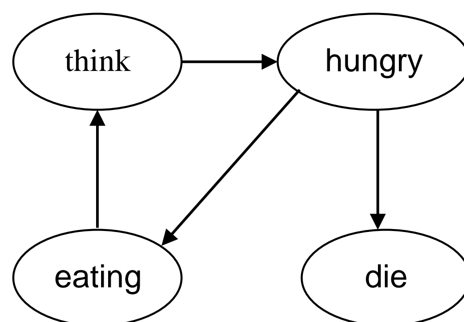
```
loop
  __Readers enter one at a time
  P(rmutex);
  __First reader waits for
  reader's turn,
  __then inhibits other writers;
  if nreaders=0 then
     P(wmutex);
  endif
  nreaders:=nreaders+1;
  __Allow other reader enter entries/exits;
  V(rmutex);
  Perform read operations;
  __Readers exit one at a time;
  P(rmutex);
  nreaders:=nreaders-1;
  Last reader allows writers;
  If nreaders=0 then
     V(wmutex)
  endif;
  __Allow reader entry/exit
  V(rmtuex);
endloop;
```

B Writer

```
loop
__Each writer operates alone;
P(wmutex);

Perform write operations;
V(wmutex);
endloop;
```

* The Dining-Philosopher Problem






```
semaphore chopstick[5]; (initially all values are 1)
do{
        wait(chopstick[i]);
        wait(chopstick[(i+1)% 5]);
        …
        eat
        …
        signal(chopstick[i]);
        signal(chopstick[(i+1)% 5]);
        …
        think
        …
} while(1);
```

* Deadlock can occur!

– Solutions (deadlock free ←→ starvation free)
  1. At most 4 philosophers appear !
  2. Pick up only two chopsticks at a time. (全拿到才行)
  3. Order their behavior:
     odd one → pick up his left chopstick
     even one → pick up his right chopstick

  →binary semaphore 來 implement counting semaphore

* Dining philosopher problem

有五個哲學家，其生活形態只有吃和想，他們為做成如圖 7-4。每個哲學家想吃時就會拿取相鄰的一雙筷子。吃完後就會放下筷子。為了防止"deadlock"的現象，即每個人都拿一隻筷子致使所有人都不能拿一雙筷子而吃。因此利用 P 與 V 運算解決之。

使用的 semaphores 如下：

chopstick: array[0..4] of semaphore，指每個筷子相對一個 semaphore，故對第 i 個哲學家的程式結構如下：

```
do{
      P(chopstick[i]);
      P(chopstick[(i+1)% 5]);
         …
         eat
         …
      V(chopstick[i]);
      V(chopstick[(i+1)% 5]);
         …
         think
         …
} while(1);
```

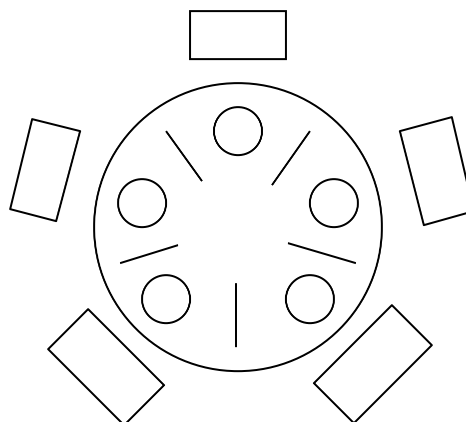


圖 7-4 哲學家問題

＊ AND Synchronization
-->Deadlock problem.

Process A	Process B
P(Dmutex)	P(Emutex)
P(Emutex)	P(Dmutex)

Solution 1: restrict the order in which request can be made.

Solution 2: request at once all the resources necessary and acquires either all of them or none.

*Dining Philosopher Problem (AND) (Dijkstra)

Five philosophers sit around a table.
Each philosopher alternates between thinking and eating.
In front of each philosopher there is a rice bowel.

When a philosopher wishes to eat, he picks up two chopsticks (forks) next to his plate.(only five chopsticks)
So, a philosopher can eat only when neither of his neighbors is eating.

The problem is to write the algorithm for philosopher($0 \leqq i \leqq 4$).
Solution based on semaphores cannot prevent a philosopher (process) from being permanently locked out by two conspiring neighbor processes; otherwise, it will result in starvation or infinite delay.

∗ 6.6 Critical Regions

- Motivation: need high-level language construct to reduce the possibility of errors (* Semaphore is too low level *)
  e.g., programming errors in using semaphores

interchange the order of wait & signal
miss some waits or signals
replace waits with signals,…

- Def. Var V: shared T;

```
region v when B do S;
e.g., a piece of code for consumer & producer
struct buffer {
        int pool[n];
        int count, in, out;
}
```

```
producer:   …
        region buffer when (count<n)
        {
                pool[in] =nextp;
                in =(in+1)% n;
                count++;
        }

consumer:…
        region buffer when (count>0)
        {
                nextc =pool[out];
                out =(out+1) % n;
                count --;
        }
```

- Implement by semaphores

```
region x when B do S;
var
   semaphore mutex; /* mutual exclusion of critical section
                              -->initial 1*/
   semaphore first-delay, second-delay; /* B evalution */
   integer first-count, second-delay; /*# of processes */
   wait(mutex);
   while not B
      do {
Overhead                 first-count++;
of reevaluation          if (second-count>0)
or Pi  ---->                   signal(second-delay);
fails B                  else signal(mutex);
       fail B            wait(first-delay);
         first-delay     first-count --;
                         second-count++;
                         if (first-count>0)
                               signal(first-delay);
     second-delay        else signal(second-delay);
                         wait(second-delay);
                         second-count--;
                    }
evaluate  S:
 B        if (first-count>0)
                 signal(first-delay);
           else if (second-count>0)
                 signal(second-delay);
            else signal(mutex);
```

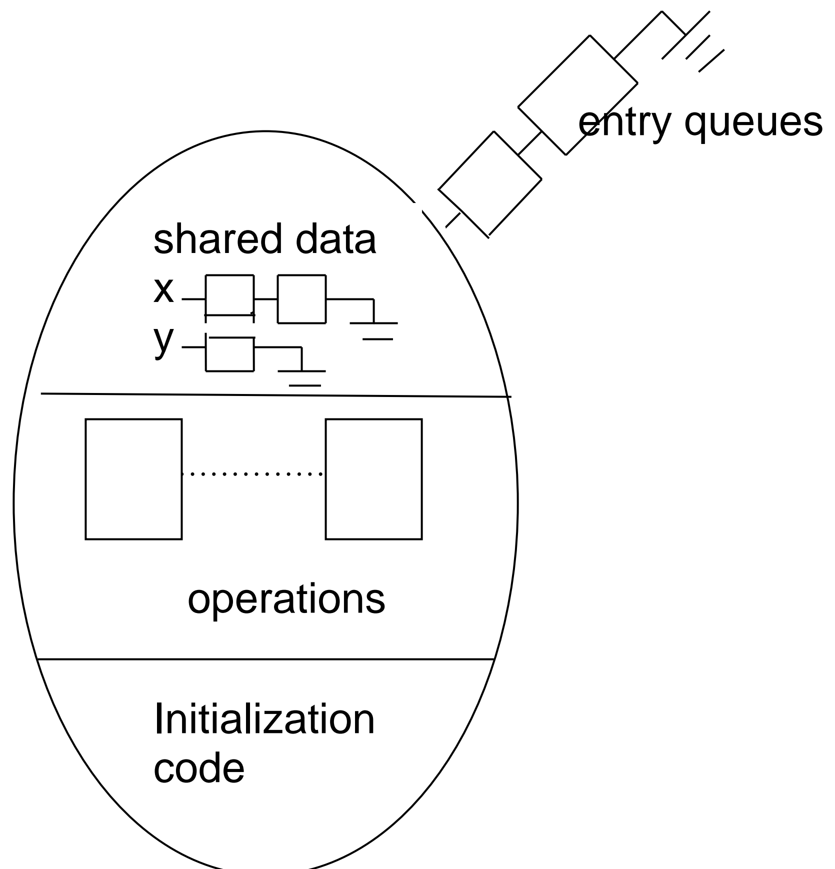
- first-delay (剛進 waiting queue, priority 最高)
- mutex (想進此門之人)
- second-delay (已在 waiting queue, 試著出來, 不一定成功)
- B : 每次有人離開 CS, 都得重新 test 一次
- 總是去叫醒別人，自己才睡。

6.7 Monitor

--Def.

```
monitor monitor-name
{
    shared variable declaration
    procedure entry P1(…)
    {…}
    procedure entry P2(…)
    {…}
    procedure entry Pn(…)
    {…}
    {
          initialization code
    }
}
```

```
var
condition x,y;

x.wait();
x.signal();
```



*only one process can be active within a monitor at a time

- Semantics of signal & wait
  - x.signal resume one suspend process or if there is done ,no effect is imposed
  - P x.signal a suspended process Q (* P,Q 均在 monitor，不行 *)
    1. P either waits until Q leaves the monitor or waits for another condition
    2. Q either waits until P leaves the monitor, or waits for another condition (會出問題，因為 Q 可能下次就永遠出不來了！可能 waiting condition 又被改了！)
    3. else…

- Example: Implement of the Dining Philosopher Problem

```
monitor dp
{
        enum {thinking, hungry, eating} state[5];
        condition self[5];
        void pickup(int i)          // following slides
        void putdown(int i)   // following slides
        void test(int i)          // following slides
        void init() {
            for (int i = 0; i < 5; i++)
                state[i] = thinking; (* No deadlock; but starvation *)
        }
}
void pickup(int i) {
        state[i]=hungry;
        test[i];
        if (state[i]!=eating) self[i].wait();
}

void putdown(int i) {
        state[i] =thinking;
        test((i+4)% 5);
        test((i+1)% 5);
}

void test(int i)
      {
        if (state[(i+4) % 5] !=eating && (state[i]==hungry)
        && (state[(i+1) % 5] !=eating)) { (我想吃，且左右沒人吃*)
                state[i] =eating; self[i].signal();
        }
      }
```

—Implementation a monitor by using semaphores

```
mutex        /* protect the monitor */ (* for ME *)
next         /* initialized to zero to handle signaling &
                resumption of processes */
-Every external function F
   =>wait(mutex);
       …
       body of F;
       if (next-count>0)
             signal(next); (*有人在 monitor 內 waiting*)
       else
             signal(mutex);(*叫外面人進來*)
-For every condition x, have a semaphore x-sem & an
integer variable x-count
 • x.wait:
          x-count++;
          if (next-count>0)
                  signal(next); (*priority 高*)(*原就在 monitor
          else signal(mutex);                          內之人*)
          wait(x-sem); (* wait for condition *)
          x-count--;
 • x.signal:
          if (x-count>0)
               {
                   next-count++;
                   signal(x-sem); (* 我去叫醒別人 *)
                   wait(next);    (* 我再去睡覺 *)
                   next-count--;
               }
```

- Process-resumption order (*在 waiting queue 中,想重新開始*)
  <~> Queuing mechanism in a monitor
  A solution:
  x.wait(c);
  where the expression c is evaluated to determine its process's resumption order.

- Another difficulty:
  Motivation: resource allocation
  R.acquire(t);
  ....
  access the resource;
  R.release;

  Concerns:
  -processes may access resource without consulting the monitor
  -processes may never release resources
  -processes may release resources never requested
  -processes may even request resources twice

Remark: whether the monitor is correctly used?

=>Requirements for correct computations

1. processes always make their calls on the monitor in correct order.
2. No uncooperative process can access resource directly without using the access protocols

Note: Scheduling behavior should consult the built-in monitor scheduling algorithm if resource access PRC are built inside the monitor.

*Language mechanisms for concurrency

*So far, the synchronization mechanisms-->too low level; assembly language programming.
It is useful for implementing the primitives typically offered by operating system; too primitive to build large, reliable systems.

Therefore, we need higher level concepts integrated into modern programming so that correctness is supported and underlying hardware implementations are of no importance to the programmer.(the object model)

==>Monitor, Serializer, Path Expression.

*Basic concepts of <u>concurrent programming</u>

1) monitor. (serializer; path expression; concurrent pascal)
2) messages.(smalltalk)
3) input/output statements.
   (CSP: communicating sequential processes)
4) procedures.(DP: distributed processes)
5) guarded commands.(Argus)
6) rendezvous: ADA.

The Producer/Consumer problem Using Direct Interprocess Communication.

```
Producer Pi
var m1: message;
loop
   …
   Create a new message m1;
   send(Pj,m1);
   …
endloop
```

```
Consumer Pj
var m2: message;
loop
   …
   receive(Pi,m2)
   Use message m2;
   …
endloop
```

*Monitor

An abstract data type implementation of the object model with additional capabilities for process synchronization with respect to resource objects of the type.
(for centralized computer systems)

A monitor is a data type used manage an operating system resource ,either hardware or software.

The monitor allows only one process to be active (executing a procedure) within the monitor at a time automatically.

Six parts:

1) resource.
2) local data.
3) scheduler: control the order of resource allocation.
   (implicit)(for those who want to enter the monitor)
   (wait outside the monitor)

4) Queues (condition variables): if a process has to wait, it will be placed in a queue and the next process is allowed to enter into the monitor.(wait inside the monitor)

Condition variables are explicitly referenced by processes executing procedures(local variables).
The associated queues hold processes that are blocked by each condition variable.

The guard (implicit) performs this regulation.

Each condition variable has an implicit (FIFO or Priority) queue.

Wait causes the process to be added at the end of the queue.

Signal wakes up the process at the front of the queue.

5) procedures: operations.
(only this part is visible outside the monitor)

6) Initialization code: is executed when an instance of the monitor is defined.

Since the monitor definition includes provision for mutual exclusive execution of any procedure automatically, the procedures themselves need not be written to solve any associated problem.(good)

*The reasons of condition variables:

1) for an executing process to be delayed until a specified condition is satisfied.

2) avoid the case: only one process may execute in a monitor at a time precludes a busy waiting form the resource to become free.(release CPU)

One(condition variable) is declared for each different condition that may cause an executing process to wait.

*Three operations:

1) wait: join the queue.
(clear the way for another process to enter/resume).

2) signal: cause one process (waiting in the queue) to begin immediate execution.

3) Queue: boolean variable. true when queue is not empty.

Bad: since mutual exclusive execution is the rule (in a monitor), we cannot encapsulate the data shared by readers and writers within the monitor itself (i.e., shared data must be put outside the monitor); simultaneous access by readers would be impossible.

When resource is inside the monitor -->no concurrency.

But by excluding the shared data from the monitor, we must once again rely on correct behavior by reader and writer process.
In this case, the monitor has not added to the reliability of the solution.(allow concurrency)

Bad: Explicit signal.

Expressive power, easy of use and modifiability are good.
Modularity and correctness are poor.

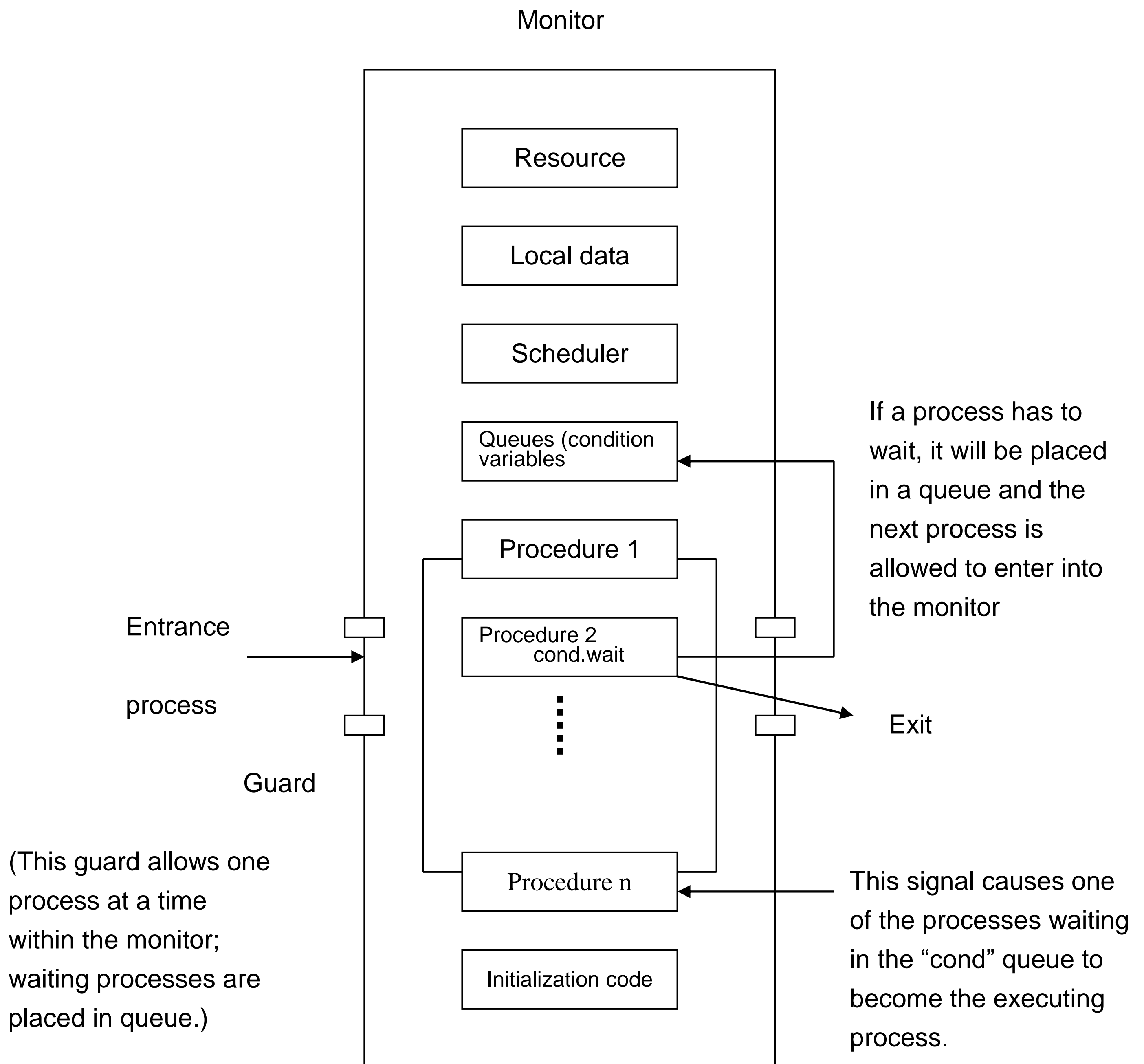


Fig. Components of a Monitor

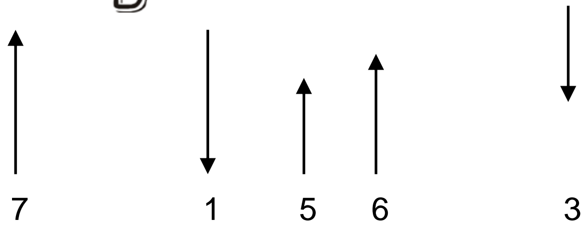
```
Monitor_name: monitor
    Declarations of data local to the monitor
    …
    CV1, CV2, …: condition;
    Declaration of procedures to implement operations
    …
    Procedure Name (… formal parameters …);
    begin
    …
    procedure body may include
        “Cvi.wait” and
        “Cvi.signal” statements
        …
        end;
        … (more procedures)
    begin
        Initialization statements for local DATABASE
    end
```

∗ A strong reader preference solution based on monitors

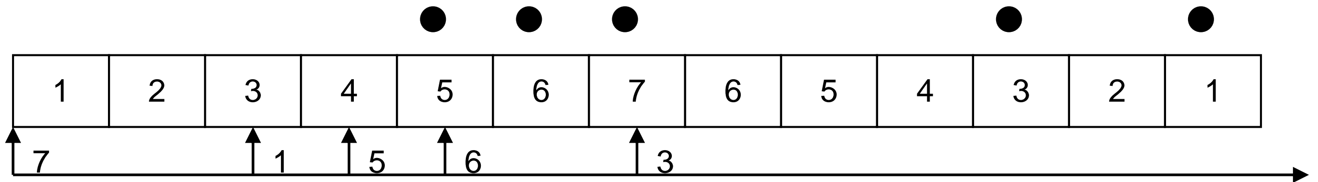
```
readers_and_writers: monitor
var readercount: integer;
    busy: Boolean;
    Oktoread, Oktowrite: condition;

void starread()
{
    if (busy) Oktoread.wait ;
    readercount++;
    if (Oktoread.queue) Oktoread.signal;
}

void endread()
{
    readercount --;
    if (readercount == 0) Oktowrite.signal;
}

void starwrite()
{
    if (readercount != 0 || busy) Oktowrite.wait;
    busy = true;
}

void endwrite()
{
    busy = false;
    if (Oktoread.queue ) Oktoread.signal (***);
    else Oktowrite.signal ;
}
void main()
{
    readercount = 0;
    busy = false;
}
```

∗ A bounded buffer based on monitor

```
bound_buffer: monitor;
var buffer: array 0 .. N-1 of portion;
    lastpointer: 0 .. N-1;
    count: 0 .. N;
    nonempty, nonfull: condition;

void append (x: portion) (* producer *)
{
    if (count == N) nonfull.wait;
    buffer[lastpointer] = x;
    lastpointer++;
    count++;
    nonempty.signal;
}

void remove (result x: portion) (* consumer *)
{
    if (count == 0) nonempty.wait;
    X = buffer[lastpointer – count];
    (* X:= buffer[(lastpointer – count + N) mod N] *)
    count--;
    nonfull.signal;
}
void main()
{
    count = 0;
    lastpointer = 0;
}
```

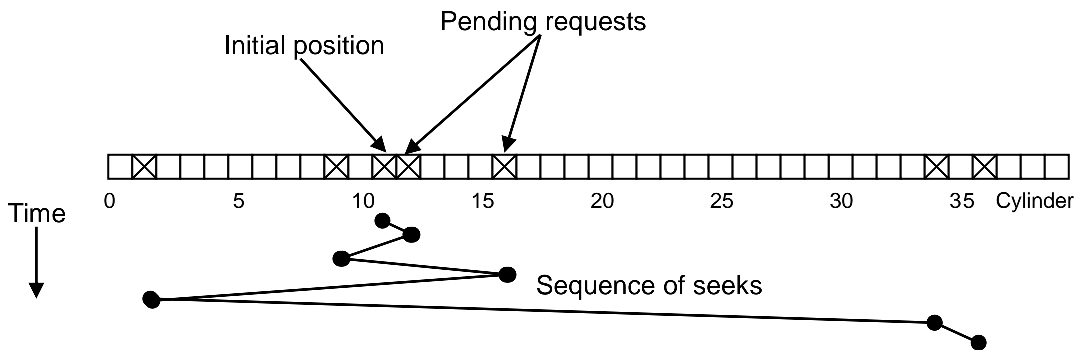


(a) FCFS

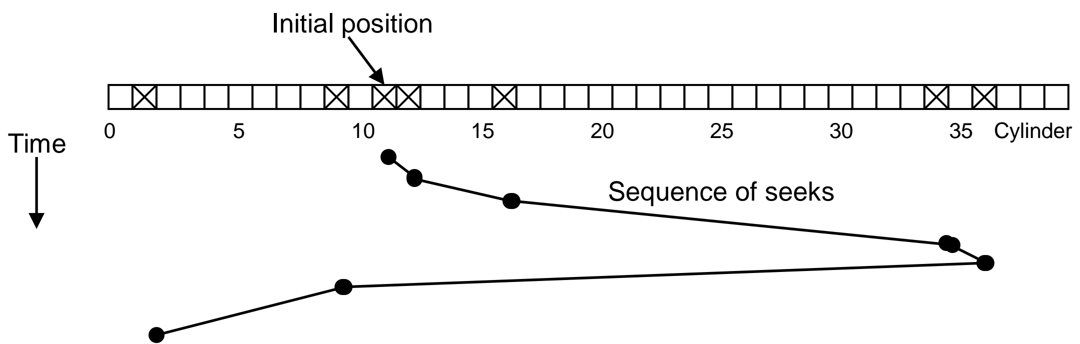


(b) Elevator

Fig. A disk-head scheduler



Shortest first (SSF) disk scheduling algorithm.



The elevator algorithm for scheduling disk requests.

∗ A disk-head scheduler

Resource schedulers often use information about requests to dynamically determine the order in which requests will be satisfied.
To reduce seek time, the algorithm gives preference to requests nearest the current cylinder.
Look algorithm: the algorithm sweeps in one direction until no outstanding requests "lie ahead" and then reverses to sweep cylinders in the other direction until requests in that direction are exhausted.

∗ The Elevator Algorithm
We have a fixed-head disk where data is stored on several cylinders.
To serve a user request for data on a particular cylinder, the disk-head has to be first moved to that cylinder.
Latency times in disk-head movement can be quite high and can affect the total performance of a file system.
We need a scheduling algorithm that minimizes the head movement.
The well known Elevator Algorithm serves our purpose.

1) if the head is already in one direction, it looks only for requests still pending in that direction.
2) When no more requests are pending in the direction of head movement, the direction is reversed and serving of requests that are pending in the new direction starts.

At any time, the heads are kept moving in the given direction, and they serve the program requesting the nearest cylinder in that direction.

If there is no such request, the direction changes, and the heads make another sweep across the surface of the disk.

∗A disk-head scheduler based on monitors

```
diskhead: monitor
Var headpos, maxcylinderindex: cylinderindex;
    direction: (up, down);
    busy: Boolean;
    upsweep, downsweep: condition;

void request ( dest: cylinderindex )
{
    if (busy){
        if ((headpos < dest) ||
                    ( headpos == dest && direction == up ))
                        upsweep.wait(dest);
        else downsweep.wait(maxcylinderindex – dest);
    }
    busy = true;
    headpos = dest;
    move disk head to cylinder dest
}

void release()
{
    busy = false;
    if (direction == up) {
        if (upsweep.queue) upsweep.signal;
        else {
            direction = down; downsweep.signal; }
    }
    else {
        if (downsweep.queue) downsweep.signal;
        else {
            direction = up ; upsweep.signal;
```

```
        }
    }
}
void main()
{
    headpos = 0;
    direction = up;
    busy = false;
}
```

```
type dining-philosophers = monitor
    var state : array[0..4] of (thinking, hungry, eating);
    var self : array[0..4] of condition;

     void pickup (int i)
       {
          state [i] = hungry;
          test (i);
          if (state[i] != eating ) self[i].wait();
       }

    void putdown (int i)
       {
          state[i] = thinking;
          test ((i+4)% 5);
          test ((i+1)% 5);
       }

    void test (int i)
       {
          if ((state[(i+4)% 5] != eating) && (state[i] == hungry) &&
          (state[(i+1)% 5] !=eating) )
              {
                 state[i] = eating;
                 self[i].signa()l;
              }
       }
    void main()
    {
       for (i = 0 ;i<5; i++)
          state[i] = thinking;
    }
```

Figure 5.21 A monitor solution to the dinning-philosopher problem.

∗Nested monitor calls

Deadlock can occur.

Example. (process A call M first; wait)

```
M monitor;              N monitor
{                       {
   call N;                 if .. the P.wait; (* A satisifies *)
}                          if .. then P.signal (* B satisifies *)
                        }
```

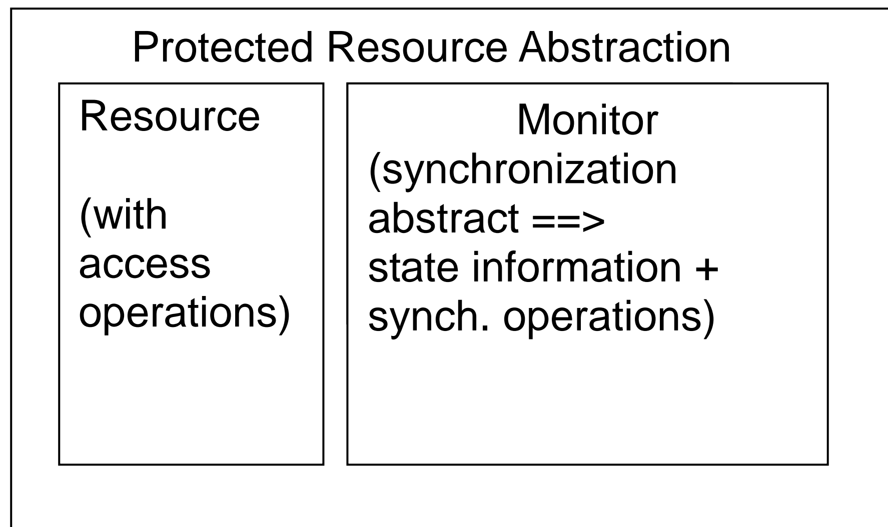
∗ Bad about monitor:

(1) nested monitor calls (solution ?)
(2) restricted concurrency.
All procedures defined by a monitor are mutually exclusive.
We would like several users to read a database simultaneously.

(3) poor resource abstraction.
Monitor → resource manager.
But, where do we put resource?
Outside the monitor: danger of unlocked ccess.
Inside the monitor: monitor is locked while resource is in use.

Serializer → hind the resource but still allow independent access.

(4) unstructed synchronization constructs.
The conditions on which a process Waits are not easily visible.
Signals have to be explicitly specified (not automatic).
If put signals at the end of a procedure → good?



∗Implementations P and V (acquire and release) by using monitors

```
SingleResource : monitor;
var busy : boolean;
    nonbusy : condition;

void acquire (* P *)
{
    if (busy) nonbusy.wait();
    busy = true;
}

void release (* V *)
{
    busy = false;
    nonbusy.signal();
}
void main()
{
    busy = false;
}
```

∗Using P & V to implement Monitors

How to implement monitors by using P and V ?
Monitors and semaphores are equivalent.

* Implementation of monitors

For mutual exclusion use an integer semaphore mutex.
Initialization :: mutex = 1;
For each procedure of the monitor:
P(mutex) is executed on entry;
V(mutex) is executed on exit;

For each condition variable **cond** that is used in the monitor, the implementation uses an integer semaphore **condem**.
Initialization :: condem = 0;
We need to implement cond.wait and cond.signal.

When a process signals a condition on which another process waiting, the signaling process must wait (somewhere) until the resumed process permits it to proceed.

We use another integer semaphore **urgent** for this purpose.
Initialization :: urgent = 0;
We maintain a count of urgent processes in urgentcount.

*Each exit from a monitor procedure should be coded as :

```
if (urgentcount > 0) V(urgent) ;
else V(mutex);
```

```
*Cond.wait ::
   condcount ++;
   if (urgentcount > 0) V(urgent) ;
   else V(mutex);
   P(condsem);
   Condcount --;

*Cond.signal ::
   urgentcount ++;
   if (condcount > 0) {
      V(condsem);
      P(urgent) ;
   }
   urgentcount --;
```

*SingleResource after compilation

```
SingleResource : class;
var busy : Boolean;
      condcount, urgentcount : integer;
      mutex, urgent, condsem : semaphore;

void acquire()
{
    P(mutex);
    if (busy)
    {
        condcount ++;
        if (urgentcount > 0) V(urgent);
        else V(mutex);
        P(condsem);
        condcount --;
    }
    busy = true;
    if (urgentcount > 0) V(urgent);
    else V(mutex);
}

void release()
{
    P(mutex);
    busy = false;
    urgentcount ++;
    if (condcount > 0 )
    {
        V(condsem);
        P(urgent);
    }
```

```
    urgentcount --;
    if (urgentcount > 0) V(urgent) ;
    else V(mutex);
}
void main()
{
    busy = false;
    condcount=0; urgentcount = 0;
    mutex=1 ; urgent=0; condsem = 0;
}
```