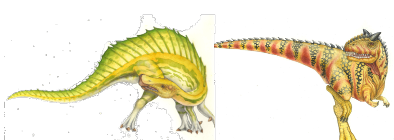
Introduction to Operating System

Process Synchronization

Chap6+7 Synchronization Tool & Example



葉奕成 I-Cheng (Garrett) Yeh

2021/5, V9 for 10th ed.

content

BACKGROUND

Review : Producer-Consumer Problem

- *producer* process produces information
- *consumer* process consume information

Producer-Consumer Problem – A2

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
       . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0; //表示目前在buffer內的item數
```

Producer-Consumer Problem – A2

Producer code

```
item nextProduced;
while (true) {
  // Produce an item in nextProduced
  while (count == BUFFER SIZE)
    ; // do nothing -- no free buffers

  buffer[in] = nextProducer;
  in = (in + 1) % BUFFER SIZE;
  count++;
}
```

Consumer code

```
item nextConsumed;
while (true) {
  while (count == 0)
    ; // do nothing -- nothing to consume

  // remove an item from the buffer
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER SIZE;
  count--;
  return item;
}
```

Race Condition

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

Race Condition

- Initially, let counter = 5.

  S0: P: register1 = count {register1 = 5}

  S1: P: register1 = register1 + 1 {register1 = 6}

  S2: P: count = register1 {count = 6}

  S3: C: register2 = count {register2 = 6}

  S4: C: register2 = register2 - 1 {register2 = 5}

  S5: C: count = register2 {count = 5} – CORRECT!

Race Condition

- Initially, let counter = 5.

  S0: P: register1 = count {register1 = 5}

  S1: P: register1 = register1 + 1 {register1 = 6}

  S2: C: register2 = count {register2 = 5}

  S3: C: register2 = register2 - 1 {register2 = 4}

  S4: P: count = register1 {count = 6}

  S5: C: count = register2 {count = 4} - ??????

Race Condition

- Initially, let counter = 5.

  S0: P: register1 = count {register1 = 5}

  S1: P: register1 = register1 + 1 {register1 = 6}

  S2: C: register2 = count {register2 = 5}

  S3: C: register2 = register2 - 1 {register2 = 4}

  S4: C: count = register2 {count = 4}

  S5: P: count = register1 {count = 6} - ??????

Race Condition

- A Race Condition
  - Several processes concurrently access and manipulate the same data
    - A situation where the outcome of the execution depends on the particular order of process scheduling.

      若沒提供互斥存取控制，可能會因為
      process執行的**順序不同**而結果有所不同

  - We need to ensure only one process at a time can be manipulating the shared data
    - Process synchronization and coordination

THE CRITICAL-SECTION PROBLEM

Framework for analysis of solutions

- Each process executes at nonzero speed but no assumption on the relative speed of N processes
  - = No assumption about order of interleaved execution

- Many CPUs may be present but memory hardware prevents simultaneous access to the same memory location

- For solutions: Processes may share some common variables to synchronize their actions.

The Critical-Section Problem

- Design a protocol that processes can use to cooperate.
  - Each process has a segment of code, called a critical section, whose execution must be mutually exclusive.
    - The section of code implementing this request is called the entry section.
    - The critical section (CS) might be followed by an exit section.
    - The remaining code is the remainder section (RS).
  - mutually exclusive
    - one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

The Critical-Section Problem

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

            critical section

        flag[i] = FALSE;

            remainder section

} while (TRUE);
```

Entry section

Permission request

管制process可否進入C. S. 的程式片段

C.S.

Process中對共享變數存取的程式片段

Exit section

Exit notification

離開C. S. 之後的解除入口管制的動作

R.S.

對非共享變數存取的程式片段

Requirements for a **Valid** Solution to Critical Section Problem

- Three Requirements
  - Mutual Exclusion
    - Only one process can be in its critical section.
  - Progress : *程式要持續有進展
    - If no process is executing in its CS and there exist some processes that wish to enter their CS…
    - (i) Only processes not in their remainder section can decide which will enter its critical section.

      Process在非RS的位置可以決定誰可以進入CS
      => Process在RS的位置不能決定誰可以進入CS
      => 不想進入CS的Process不能阻礙其他Process進入他們各自的CS
    - (ii) The selection cannot be postponed indefinitely.
      - No deadlock

Requirements for a **Valid** Solution to Critical Section Problem

- Three Requirements
  - Bounded Waiting
  - After a process has made a request to enter its CS, there is a bound on the number of times that the other processes are allowed to enter their CS and before that request is granted.
    - A waiting process only waits for a bounded number of processes to enter their critical sections.
    - No starvation

Types of solutions

- Software solutions
  - Algorithms whose correctness does not rely on any other assumptions (see the framework)
- Hardware solutions
  - Rely on some special machine instructions
- Operation System solutions
  - Provide some functions and data structures to the programmer

Software solutions

- We consider first the case of 2 processes
  - Algorithm 1 and 2 are incorrect
  - Algorithm 3 is correct (Peterson's algorithm)
- Then we generalize to *n* processes
  - Bakery algorithm
- Notation
  - We start with 2 processes: P0 and P1
  - When presenting process Pi, Pj always denotes the other process (i != j)

Algorithm 1

- Shared Variable: int **turn**
  - initially **turn** = i or j;
  - // means $P_{i\ or\ j}$ can enter its critical section
- Assumption
  - Every basic machine-language instruction is atomic.

假設基本的組合語言指令都是不可分割的

- Idea
  - Remember which process is allowed to enter its critical section, That is, *process i* can enter its critical section if **turn** = i.

利用turn記錄誰可以進入C.S.
=>誰有輪是(turn)誰就可以進入

Algorithm 1

Pi

```
do
{
while(turn !=i){no-op;   //wait till turn==i
   }
      C.S.
    turn=j;  //release turn to j
      R.S.
}while(1)
```

Entry: **turn**不在手上，只好等待

Exit: 事情做完了
把turn交給另一個process

WRONG

Algorithm 1

P0

```
do
{
while(turn !=0){no-op;
   }
         C.S.
      turn=1;
         R.S.
}while(1)
```

P1

```
do
{
while(turn !=1){no-op;
   }
         C.S.
      turn=0;
         R.S.
}while(1)
```

- When P0 has a large RS and P1 has a small RS....
- Problem: If turn=0, P0 enter its CS and then its long RS (turn=1). P1 enter its CS and then its RS (turn=0) and **tries again** to enter its CS: **request refused!**
- *P1 has to wait that P0 leaves its RS (not CS).

WRONG

Algorithm 1 - step by step

- if turn==0 and P1 is ready to enter its CS, P1 cannot do so, even though P0 may be in its RS. -> NO progress

1. P0 enters, then exits from CS (turn =>1). Now, It is in its RS.
2. Because turn==1, P1 enters, then exits from CS (turn =>0).

   P1 finishes this RS.
3. P1 wants to enter CS **again**. Now turn==0, P1 have to wait

   for P0 (even though P0 is in its RS).

Algorithm 1

WRONG

Pi

```
do
{
while(turn !=i){no-op;
   }
        C.S.
     turn=j;
        R.S.
}while(1)
```

- Mutual Exclusion
  - Only one valid value for **turn**

    ->>> only one proc. has permit.
  - satisfied
- Progress
  - When turn=i, Pj is blocked even if Pi is running in the R.S.

    *若turn=i，但Pi不想進入，則Pj也無法進入
  - No satisfies **(i)** in Progress requirement
- Conclusion: Satisfy mutual exclusion, but not progress

Algorithm 2

- Shared Variable
  - boolean **flag[2]**;
  - initially **flag**[0] =**flag**[1]=false;
  - When **flag**[i] = true >> P_i ready to enter its critical section
- Idea
  - Remember the state of each process.

利用flag記錄誰有意願進入C.S.

Algorithm 2

Pi

```
do {
   flag[i] := true;
   while (flag[j]){no-op;
   }
        C.S.
     flag[i] = false;
        R.S.
} while (1);
```

Entry :
先表達自己有意願
若對方有意願則禮讓對方做完

Exit :
事情做完表達自己沒意願

Pj

```
do {
   flag[j] := true;
   while (flag[i]){no-op;
   }
        C.S.
     flag[j] = false;
        R.S.
} while (1);
```

WRONG

Algorithm 2

- May loop infinitely
  - Both of two proc. want to get into CS

若雙方都有意願，則雙方都在禮讓對方
導致都無法繼續往下執行

- If changes the order to be

while (flag[j]) {no-op;} //wait first
flag[i] := true; //set flag later..
… C.S.

  - then the condition of mutual exclusion will not hold.

CORRECT

Algorithm 3: Peterson's Solution

- Combined shared variables of algorithms 1 and 2.
- Shared Variable
  - int **turn**;
    - **turn** = i // P_i can enter its critical section
  - boolean **flag**[2];
    - **flag**[0] = **flag**[1]=false

利用turn記錄誰可以進入C. S.
利用flag記錄誰有意願進入C. S.
=> 有意願且有鑰匙(turn)的process可進入C. S.

- "**turn**" will be set for both *i* and *j* simultaneously, but only one (turn = *i*) or (turn = *j*) will last.
- Meets all three requirements; solves the critical-section problem for two processes.

CORRECT

Algorithm 3

P_i

```
do {
  flag[i] := true;
  turn = j;
  while (flag [j] and turn == j) {no-op;
  }
       C.S.
  flag[i] = false;
       R.S.
} while (1);
```

Entry :
先表達自己有意願
再禮讓對方先做
若對方有意願則禮讓對方做完
若對方無意願則繼續往下做

Exit :
事情做完表達自己沒意願

就算flag[i]和[j]同時都是true，turn也會決定唯一的人選來進入C.S.
turn = j 的這個指令總是有一個人會先執行/一個人會後執行

CORRECT

Algorithm 3

P$_j$

```
do {                                                          j
  flag[j] := true;
  turn = i;
  while (flag [i] and turn == i) {no-op;
  }
        C.S.
  flag[j] = false;
        R.S.
} while (1);
```

CORRECT

Algorithm 3

- Mutual exclusion
  - At the moment that P_i enters its CS
    - flag[*i*]=*true* either flag[*j*]=*false*, or flag[*j*]=*true* and *turn=i*
  - When P_j waits to enter *有兩種可能
    - If flag[*j*]=*false*, (flag[i]=true & turn=i) prevents Pj from entering its CS until Pi exits
    - If flag[*j*]=*true* and *turn=i*, the same (only Pi can can change turn from i to j)

CORRECT

Algorithm 3

- Mutual exclusion
  - …
- Progress
  - suppose P_i wishes to enter CS
    - if flag[*j*]=false, P_i can enter
    - otherwise, *turn* allows either P_i or P_j to enter
- Bounded waiting: wait at most one entry of P_j

What about Process Failures?

- If all 3 criteria (ME, Progress, Bounded waiting) are satisfied, then a valid solution will provide robustness against failure of a process in its remainder section (RS)
  - since failure in RS is just like having an infinitely long RS
- However, no valid solution can provide **robustness** against a process failing in its critical section (CS)
  - A process Pi that fails in its CS does not signal that fact to other processes: for them Pi is still in its CS

Bakery Algorithm, Critical section for n processes

- Before entering its critical section, process receives a **number**. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if ***i* < *j***, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

*不會產生比場上數字還小的票卷
-- 後來的人號碼永遠比較大
-- 完成的人號碼歸零 (不採計大小比較)

Bakery Algorithm (cont.)

- Notation <= lexicographical order (ticket #, process id #)
  - $(a,b) < (c,d)$: if $a < c$ or if $a == c$ and $b < d$
  - max $(a_0,\ldots, a_{n-1})$ is a number, k,
    - such that $k >= a_i$ for $i = 0, \ldots, n - 1$
- Shared data

  boolean choosing[n]; //a state indicator

  int number[n]; //a ticket

Data structures are initialized to **false** and **0** respectively.

Bakery Algorithm (cont.)

```
do {                                                Take a ticket
  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]) ; /* Wait for the choosing of Pj
        while ((number[j] != 0) && (number[j],j) <
                   number[i], i)) ; // smallest first
  }
    critical section
  number[i] = 0;
    remainder section
} while (1);
```

Bakery Algorithm

- Why number[i] = number[j] ?
  - Pi: choosing[i] = true;
  - Pj: choosing[j] = true; The largest ticket is **k** for now
  - Pi: Rigister1 = max(number[0] … number[n-1])+1
  - Pj: Rigister2 = max(number[0] … number[n-1])+1
  - Pi: number[i] = Rigister1 // == k+1
  - Pj: number[j] = Rigister2 // == k+1

Bakery Algorithm

- When we remove while (choosing[j]){no-op} …
  - No satisfies mutual exclusion

default:
number[0..n-1]=0,
Pid i< Pid j,
Pi, Pj prepare to enter the C.S.

Pi : choosing[i] = true;
Pj: choosing[j] = true;   The largest ticket is **k** for now
Pi: number[i] = max(number[0] … number[n-1])+1 //not assign yet!!
Pj: number[j] = max(number[0] … number[n-1])+1 // ==k+1
Pj : choosing[j] = false
Pj : P_j in C.S
Pi : number[i] = k+1 // complete the assignment step now…
Pi : P_i in C.S !!!
=> No satisfies mutual exclusion

Bakery Algorithm (cont.)

- Key for showing the correctness

  if P_i in CS, all other P_k has

  number[k] = 0, or

  (number[i], i) < (number[k], k)
- mutual exclusion: OK
- progress: OK (smallest first)
- bounded waiting: OK
  - Note that processes enter their CSs in a FCFS basis.
  - How many times ? $\underline{n}$

Drawbacks of Software Solutions

- Processes that are requesting to enter in their critical section are busy waiting (consuming processor time needlessly)
- If Critical Sections are long, it would be more efficient to block processes that are waiting...

Synchronization Hardware

HARDWARE SUPPORT FOR SYNCHRONIZATION

Hardware Solutions ?

- Solution to Critical-section Problem Using Locks
- A process …
  - **Acquire** a lock before entering a critical section
  - **Releases** the lock when it exits the critical section

```
do {
        acquire lock
              critical section
        release lock
              remainder section
} while (TRUE);
```

Hardware Solutions: Interrupt Disabling

- On a uniprocessor:
  - Mutual exclusion is preserved but efficiency of execution is degraded: while in CS, we cannot interleave execution with other processes that are in RS.
- On a multiprocessor: mutual exclusion is not preserved
  - CS is now atomic but not mutually exclusive.
  - Potential impacts on interrupt-driven system clocks.

```
Process Pi:
repeat
   disable interrupts
    critical section
   enable interrupts
    remainder section
forever
```

Hardware Solutions: Special Machine Instructions

- Normally, access to a memory location excludes other access to that same location
  - Extension: designers have proposed machines instructions that perform 2 actions atomically (indivisible) on the same memory location (ex: reading and writing)
  - The execution of such an instruction is also mutually exclusive (even with multiple CPUs)
- They can be used to provide mutual exclusion but need to be complemented by other mechanisms to satisfy the other 2 requirements of the CS problem (We still need to avoid starvation and deadlock)

Synchronization Hardware (1)

- Having the support of some simple hardware instructions, the CS problem can be solved very easily and efficiently.
- The CS problem occurs because the modification of a shared variable of a process may be interrupted.
- Two common hardware instructions that execute *atomically*
  - *Test-and-Set*
  - *Swap*

*Atomic = non-interruptable

Atomically executed:
執行過程中不會被中斷
可在一個memory cycle內完成工作

Synchronization Hardware (2)

- Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {
    boolean rv = target; //backup original value
    target = true; //set target = true

    return rv; //send out original value
}
```

No Preemption

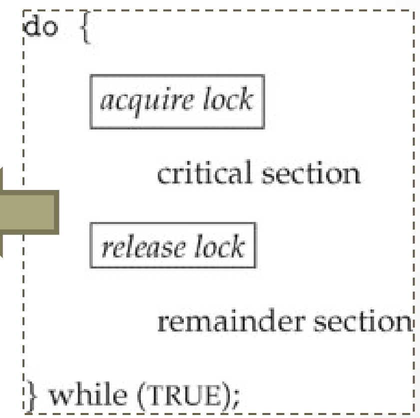
Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
do {
    while (TestAndSet(lock)) ;       no-op
        critical section
    lock = false;
        remainder section
}
```



For example:
Time 0: Pi in: TestAndSet(&lock) //return false and set lock =true > Pi in C.S
Time 1: Pj in: TestAndSet(&lock) //return true and set lock =true > Pj in loop…

Mutual Exclusion with Test-and-Set

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

- Mutual Exclusion
  - 有鑰匙才能進 -> ok
  - 執行過程中不會被中斷：
    *可在一個memory cycle內完成工作
- Progress
  - 不想進入C.S就不拿鑰匙 -> ok
  - 有限時間內可決定鑰匙給誰 -> ok
- Bounded Waiting
  - Pi離開後又想進入C.S, Pi直接執行TestAndSet (比Pj早/Pj不在等待時) 則Pi又進入C.S.
  - Starvation … ??? (有可能一直把持著鑰匙不放)

A improved version of multiple process!

Extend Test-and-Set to N Processes

false:無意願進入C.S 或即將進入C.S
true:有意願 & 正在等待進入C.S

- Shared data:

boolean lock = false;
boolean waiting[0..n-1] //default : false waiting

- Process P_i

key : boolean
PID j : 0 … n-1

找下一個
有意願的

```
do {
    waiting[i]=true; //表達意願
    key = true;
    while(waiting[i] and key) {key = TestAndSet(&lock);}
    waiting[i]=false;
        critical section
    j=(i+1) mod n;
    while (j != i and (not waiting[j])) {j=(j+1) mod n;}
    if (j ==i) {lock = false;} //j=i表所有人無意願
    else {waiting[j]=false;} //讓下一個有意願的離開while
        remainder section
}
```

Mutual Exclusion with Test-and-Set

- Mutual Exclusion:
  靠別人幫忙
  - Pi can enter CS only if either waiting[i]==false or key==false Pi必須自己先搶到
  - key becomes false only if TestAndSet is executed
    - First process to execute TestAndSet find key==false; others wait …
  - waiting[i] becomes false only if other process leaves CS
    - Only one waiting[i] is set to false (the next available one)

- Progress
  - 不想進入C.S就不拿鑰匙 -> waiting[i]=false -> ok
  - 有限時間內可決定鑰匙給誰 -> ok

Mutual Exclusion with Test-and-Set

- Bounded Waiting
  - 若所有process皆想進入C.S

    T0: P0 : waiting[1]=false;

    T1: P1 : in C.S....

    Po想再進入C.S,需等待Pn-1完成後才能進入C.S
  - No Starvation : 一定會換下一個人

Synchronization Hardware

TYPE 2

- Atomically swap two variables.

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

No Preemption
Swap(lock,key);

Mutual Exclusion with Swap

TYPE 2

- Shared data (initialized to **false**):

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

- Process P_i

```
do {
    key = true;
    while (key == true)
            Swap(lock,key);
        critical section
    lock = false;
        remainder section
}
```

Mutual Exclusion with Swap

```
do {
  key = true;
  while (key == true)
    Swap(lock,key);
    critical section
  lock = false;
    remainder section
}
```

- Mutual Exclusion
  - 有鑰匙(& lock==false)才能進 -> ok
  - 交換鑰匙過程中不會被中斷 ：
    *可在一個memory cycle內完成工作
- Progress
  - 不想進入C.S就不拿鑰匙 -> ok
  - 有限時間內可決定鑰匙給誰 -> ok
- Bounded Waiting
  - Pi離開後又想進入C.S > Pi比Pj早下Swap,則Pi又進入C.S.
  - Starvation … ??? （把持著鑰匙不放）

A similar version…

Mutual Exclusion with Swap

- Shared data:

```
boolean lock = false;
boolean waiting[0..n-1] //default : false
```

false:無意願進入C.S 或即將進入C.S
true:有意願 & 正在等待進入C.S
waiting

- Process P_i

key : boolean
PID j : 0 … n-1

```
do {
    waiting[i]=true; //表達意願
    key = true;
    while(waiting[i] and key) {Swap(lock,key);}
    waiting[i]=false;
        critical section
    j=(i+1) mod n;
    while (j != i and (not waiting[j])) {j=(j+1) mod n;}
    if (j ==i) {lock = false;} //j=i表所有人無意願
    else {waiting[j]=false;} //讓下一個有意願的離開while
        remainder section
}
```

找下一個有意願的

SEMAPHORE

Overview -- Semaphore

- A high-level solution for more complex problems.
  - Synchronization tool (provided by the OS) that does not require busy waiting
- A semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 atomic and mutually exclusive operations:
  - wait(S) – P operation *proberen*（荷蘭文:測試）
  - signal(S) – V operation *verhogen*（荷蘭文:增加）
- To avoid busy waiting: when a process has to wait, it will be put in a blocked queue of processes waiting for the same event

Negative -> Blocked

Semaphore cont.

- **wait(S)**: decrements the semaphore value. If the value becomes **negative**, then the process (who is executing the wait(s)) is **blocked**
- **signal(S)**: increments the semaphore value. If the value is not positive, then a process blocked by a wait(s) operation is **unblocked**.
- wait and signal are assumed to be atomic
  - They cannot be interrupted and each routine can be treated as an indivisible step

不能分割的

Semaphore

```
wait(S) {
  while (S <= 0) ; // no-op
  S--;
}
```

```
signal(S) {
  S++;
}
```

Semaphores – Usages

- Handling "Critical Sections Problem"
  - default : Semaphore S = 1;
- Enforce the requirement of precedence
  - default : Semaphore S = 0;

Semaphores – Usages

- Critical Sections
  - Shared Variable
    - semaphore mutex= 1; // allow first process go into C.S.

P$_i$

```
do
{
    wait(mutex);
        C.S
    signal(mutex);
        R.S
}while(TRUE)
```

Semaphores – Usages

- Precedence Enforcement
  - Shared Variable
    - semaphore synch= 0;
  - Procedure
    - S1 (in P1) -> S2 (in P2)

P1

```
S1;
   signal(synch);
```

P2

```
   wait(synch);

S2;
```

Semaphores – Usages

- Counting semaphore = General semaphore
  - This initialized number represents a *number of available resources*.
  - (Non-negative) integer value can range over an unrestricted domain.
- Binary semaphore
  - Integer value can range only between 0 and 1
  - can be simpler to implement
  - Also known as mutex locks
- We can implement a counting semaphore S as a binary semaphore.

Semaphores – Usages

- Using Binary semaphore implement a counting semaphore

```
semaphore s1= 1 ; // protect c
semaphore s2= 0 ; // wait for ( c <=0 )
int c = 1 ; //counter
```

Semaphores – Usages

```
WAIT(S) // check&use resource
{
  wait(s1);         // protect C
  c--;
  if (c < 0) {
    signal(s1);
    wait(s2); // block …
  }
  else
    signal(s1);
}
```

Semaphores – Usages

```
SIGNAL(S) //release resource
{
  wait(s1);
  c++;
  if (c <= 0)
    signal(s2); //wakeup others
  signal(s1);
}
```

Semaphores – Usages

- P1 in
  - P1 get into C.S

```
do
{
   WAIT(S);
      C.S
   SIGNAL(S);
      R.S
}while(TRUE)
```

```
semaphore s1= 1 ; //1->0->1
semaphore s2= 0 ;
int c = 0 ;  // 1->0
```

Protect C

```
WAIT (S)
{                    1->0
    wait(s1);
    c--;
    if (c < 0) {
      signal(s1);
      wait(s2);
    } else signal(s1);
}                              0->1
```

```
SIGNAL (S)
{
    wait(s1);
    c++;
    if (c <= 0)
       signal (s2); /* wakeup */
    signal (s1);
}
```

Semaphores – Usages

- P2 in
  - P2 wait

```
semaphore s1= 1 ;
semaphore s2= 0 ;
int c = -1 ;
```

```
do
{
   WAIT(S);
      C.S
   SIGNAL(S);
      R.S
}while(TRUE)
```

```
WAIT (S)
{
    wait(s1); /* protect C */
    c--;
    if (c < 0) {
      signal(s1);
      wait(s2);        s2==0
    } else signal(s1);
}
```

```
SIGNAL (S)
{
    wait(s1);
    c++;
    if (c <= 0)
      signal (s2); /* wakeup */
    signal (s1);
}
```

Semaphores – Usages

- P3 in
  - P3 wait (same as P2)

|c| : mean # of process wait

```
semaphore s1= 1 ;
semaphore s2= 0 ;
int c = -2 ;
```

```
do
{
   WAIT(S);
       C.S
   SIGNAL(S);
       R.S
}while(TRUE)
```

```
WAIT (S)
{
    wait(s1); /* protect C */
    c--;
    if (c < 0) {
      signal(s1);
      wait(s2);        s2==0
    } else signal(s1);
}
```

```
SIGNAL (S)
{
    wait(s1);
    c++;
    if (c <= 0)
       signal (s2); /* wakeup */
    signal (s1);
}
```

Semaphores – Usages

- P1 out

```
do
{
   WAIT(S);
      C.S
   SIGNAL(S);
      R.S
}while(TRUE)
```

```
semaphore s1= 1 ;
semaphore s2= 1 ; // s2++ : 0->1
int c = -1 ;  // c++
```

```
WAIT (S)
{
   wait(s1); /* protect C */
   c--;
   if (c < 0) {
     signal(s1);
     wait(s2);
    } else signal(s1);
}
```

```
SIGNAL (S)
{
   wait(s1);
   c++;
   if (c <= 0)
     signal (s2); /* wakeup */
   signal (s1);
}
```

s2++

Semaphores – Usages

- P2 in C.S
  - Only P3 is wating

```
do
{
   WAIT(S);
        C.S
   SIGNAL(S);
        R.S
}while(TRUE)
```

```
semaphore s1= 1 ;
semaphore s2= 0 ; // s2-- : 0->1
int c = -1 ;
```

```
WAIT (S)
{
    wait(s1); /* protect C */
    c--;
    if (c < 0) {
      signal(s1);
      wait(s2);          <---- s2--
    } else signal(s1);
}
```

```
SIGNAL (S)
{
     wait(s1);
     c++;
     if (c <= 0)
        signal (s2); /* wakeup */
     signal (s1);
}
```

Semaphores – Usages

- P4 in
  - P4(&P3) wait

    |c| : mean # of process wait

```
semaphore s1= 1 ; //1->0->1
semaphore s2= 0 ;
int c = -2 ;
```

```
do
{
    WAIT(S);
        C.S
    SIGNAL(S);
        R.S
}while(TRUE)
```

New P comes here.

```
WAIT (S)
{
    wait(s1); /* protect C */
    c--;
    if (c < 0) {
      signal(s1);
      wait(s2);        s2==0
     } else signal(s1);
}
```

```
SIGNAL (S)
{
    wait(s1);
    c++;
    if (c <= 0)
       signal (s2); /* wakeup */
    signal (s1);
}
```

Semaphores - Implementation

- The main **disadvantage** of **previous semaphore** and **other schemes** is that they all require ***busy waiting***.
  - It wastes CPU cycles that some other process might be able to use productively in multiprogramming system.

Semaphores - Implementation

- This type of semaphore is also called a **spinlock**
  盤旋鎖
  - A Busy-Waiting Semaphore
  - while (S<=0) causes the wasting of CPU cycles
- Advantage : (in multiprocessor system)
  - When locks are held for a short time, spinlocks are useful since no context switching is involved.
  - *One thread can “spin” on one processor while another thread performs its critical section on another processor.)

因不用context switching，所以若只有很短的時間就可以離開while，spinlock是有用的

Semaphores - Implementation

- Hence, in fact, a semaphore is a **record** (structure)
- When a process must **wait** for a semaphore S, it can block itself and put on the semaphore's queue
  - The queue links PCB
- The signal operation removes (according to a fair policy like FIFO) one process from the queue and puts it in the list of ready processes
- Block and wakeup change process state – they are basic system calls
  - Block: from running to waiting
  - Wakeup: from waiting to ready

Semaphores - Implementation

- Semaphores with Block-Waiting
  - No busy waiting from the entry to the critical section
  - With each semaphore there is an associated waiting queue.
  - Each semaphore has two data items:
    - value (an integer)
    - pointer to a list of process

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

Semaphores - Implementation

- Semaphores with Block-Waiting, two operations:
  - **block**
    - place the process who invoking the operation on the appropriate waiting queue.

    將process自己放到適當的waiting queue
  - **wakeup**
    - remove one of processes in the waiting queue and place it in the ready queue.

    將某process從waiting queue取出並放到ready queue

Semaphores - Implementation

- Semaphores with Block-Waiting

```
wait(semaphore *S) {
   S->value--;  |S->value| : mean # of process wait
   if (S->value < 0) {
      add this process to S->list;
      block();
   } // S->value >= 0
}
```

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Semaphores - Observations

- When S.value >=0: the number of **processes that can execute** wait(S) without being blocked = S.value = # or resource
- When S.value < 0: the number of **processes waiting** on S is = |S.value|
- Atomicity and mutual exclusion: no 2 process can be in wait(S) and signal(S) (on the same S) at the same time (even with multiple CPUs)
- Hence the blocks of code defining wait(S) and signal(S) are, in fact, critical sections, too.

Semaphores - Observations cont.

- Wait and Signal must be executed atomically (mutual execution)
- The critical sections defined by wait(S) and signal(S) are very short: **typically 10 instructions**
- Solutions:
  - uniprocessor: disable interrupts during these operations (ie: for a very short period). This does not work on a multiprocessor machine.
  - multiprocessor: use previous software or hardware schemes. The amount of busy waiting should be small.

Deadlocks and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

Let S and Q be two semaphores initialized to 1

```
P0:   wait(S);          P1: wait(Q);
      wait(Q);              wait(S);
         ...                   ...
      signal(S);            signal(Q);
      signal(Q);            signal(S);
```

One more thing…

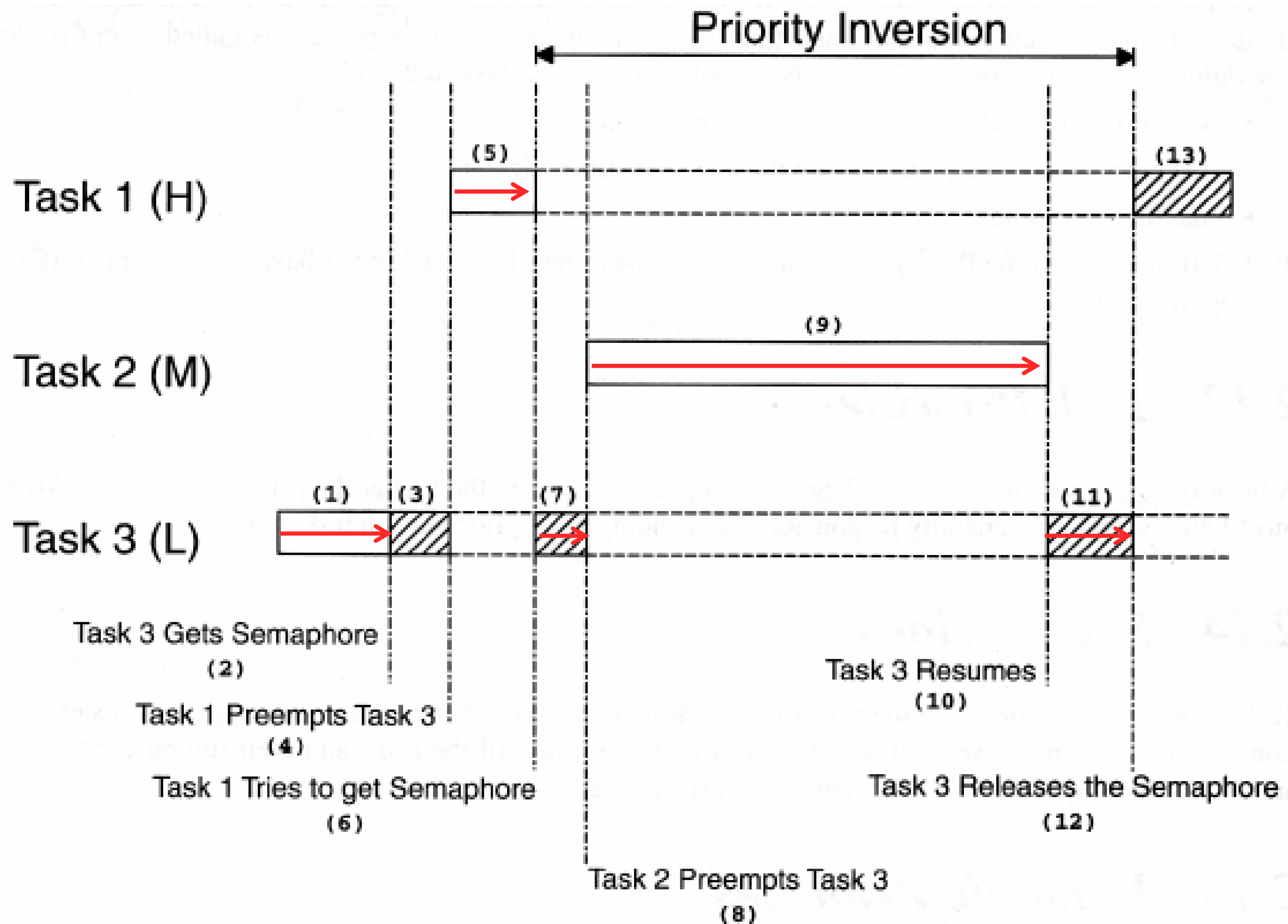
- The queuing strategy can be arbitrary, but there is a restriction for the bounded waiting requirement.
  - For example, **FIFO** is a common choice.
  - Starvation (or indefinite blocking) may occur if we add and remove processes from the semaphore queue in **LIFO** order

ISSUE: Priority Inversion

- Priority Inversion　優先權倒置
  - A higher-priority task is blocked by a lower-priority task due to some resource access conflict.
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - The situation will be worse if the lower – priority process is preempted by other higher-priority processes.

高優先權需要的資源被低優先權把持住，高優先權的process等待。若之後有一個次高優先權的process進來，並搶走低優先權process 的CPU，則發生高優先權等待次高優先權process的情況

ISSUE: Priority Inversion



ISSUE: Priority Inversion

- Priority-inheritance protocol　優先權繼承協定
  - all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources

把持高優先權所需資源的低優先權process，將繼承高優先權process的優先權，直到釋放資源為止

Classical Problems of Synchronization

- There are three classical problem
  - Bounded-Buffer Problem (Producer-Consumer Problem)
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- The three problems are important, because …
  - They are examples for a large class of concurrency-control problems
  - They has been used for testing nearly every newly proposed synchronization scheme.

Classical Problems of Synchronization

How to solve these problems by using semaphore?

Producer-consumer (bounded-buffer) problem:

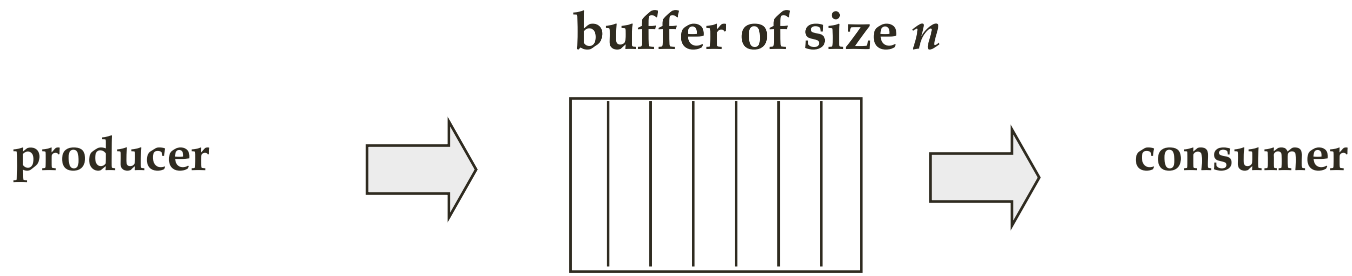
* A pool consists of *n* buffers (each holds a item)

semaphore full, empty, mutex;

mutex(1): control of mutual exclusion

empty(*n*), full(0): the number of empty and full buffers

Bounded-Buffer Problem



- Shared variable
  - *N* buffers, each can hold one item
  - Semaphore mutex = 1;
  - Semaphore full = 0;
    - 0 : no item
    - >0 : many items
  - Semaphore empty = n;
    - 0 : buffer full
    - >0 : buffer is not full

Bounded-Buffer Problem

- Producer

等待buffer 有空間，若有空間，則產生item，空間數-1

```
do {
   //   produce an item in nextp
   wait (empty); // if empty = 0 ; buffer full; wait
        wait (mutex);
        //  add the item to the  buffer
        signal (mutex);
   signal (full); // full +1 提醒consumer ， buffer已經有item
} while (TRUE);
```

Bounded-Buffer Problem

- Consumer

等待buffer 有item，若有item，則消耗item，item數-1

```
do {
   wait (full);
        wait (mutex);
        //  remove an item from  buffer
        signal (mutex);
   signal (empty); // empty +1 提醒producer, buffer已經有空間
  //  consume the item in nextc
} while (TRUE);
```

Bounded-Buffer Problem

Producer

```
do {
   //   produce an item in nextp
  wait (empty);
  // if empty = 0 ; buffer full; wait
     wait (mutex);
     //  add the item to the  buffer
     signal (mutex);
  signal (full); // full +1
} while (TRUE);
```

Consumer

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

Readers-Writers Problem (1)

- each writer is required to have exclusive access to the shared object.
- At the same time, either several reads, or only a write
- Shared data
  semaphore mutex, wrt;
  readcount: (0, initially; # of current readers)

  mutex: (1, initially; mutual exclusion for *readcount*)

  wrt: (1, initially)
    - mutual exclusion for writers.
    - It also is used by the *first-in* and *last-out* readers

Readers-Writers Problem (2)

- If a writer is in the CS and *n* readers are waiting, then
  - one is queued on *wrt*
  - the other *n*-1 are queued on *mutex*
- When a writer executes signal(*wrt*), either the waiting readers or a single writer are resumed. The selection is made by the scheduler.

Readers and Writers Problem (4) for the first solution

writer

```
wait(wrt);
  ...
     writing is performed
  ...
signal(wrt);
```

reader

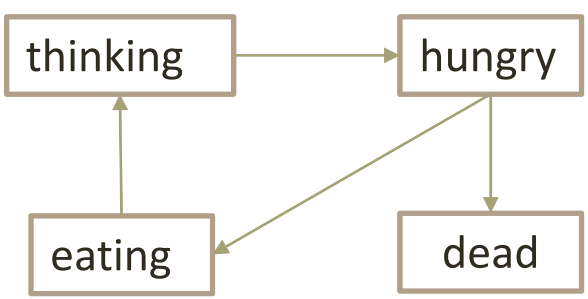
first-in

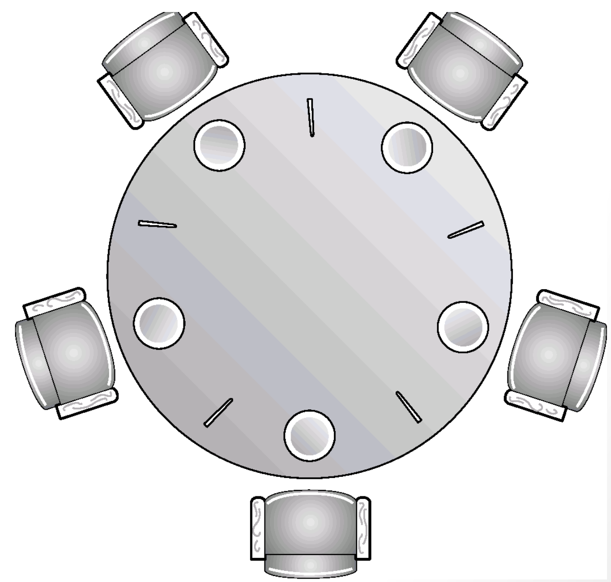
```
wait(mutex);
  readcount++;
  if (readcount == 1) wait(wrt);
signal(mutex);
   ...
     reading is performed
   ...
wait(mutex);
  readcount--;
  if (readcount == 0)  signal(wrt);
signal(mutex);
```

update readcount

Dining-Philosophers Problem

- Five philosophers, either thinking or eating
  - Each philosopher must pick up one chopstick beside him/her at a time
  - When two chopsticks are picked up, the philosopher can eat.
- Shared variable
  - Bowl of rice (data set)
  - Semaphore chopstick [5] = 1;





Dining-Philosophers Problem

philosopher *i*

```
do {                                  看左邊的筷子是否可拿
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        ...                           看右邊的筷子是否可拿
            eat
        ...
    signal(chopstick[i]);
    signal (chopstick[(i+1) % 5]);
        ...
     think
        ...
} while(1);
```

get chopsticks
- left
- right

free chopsticks
- left
- right

deadlock !

Dining-Philosophers Problem

- Deadlock occur
  - If all philosophers pick up their right one simultaneously.

- Common Solutions for Deadlocks:
  1. Allow at most four philosophers to be sitting simultaneously at the table.
  2. Pick up two chopsticks simultaneously.
  3. Order their behaviors,
     - e.g., odds pick up their right one first, and evens pick up their left one first.
  - Besides deadlock, any satisfactory solution to the DPP problem must avoid the problem of *starvation*.

Dining-Philosophers Problem

- Solutions to Deadlocks
  - Semaphore num = 4;

```
do  {
 wait (num);
   wait ( chopstick[i] );
      wait ( chopStick[ (i + 1) % 5] );
         //  eat
      signal ( chopstick[i] );
   signal (chopstick[ (i + 1) % 5] );
 signal (num);
        //  think
} while (TRUE);
```

MONITOR

Monitor

- Semaphores provide a convenient and effective mechanism for process synchronization.

- However, using them **incorrectly** can result in timing errors that are difficult to detect.
  - Since these errors happen **only if particular execution sequences** take place and these sequences do not always occur.
  - For example, exchange wait() and signal() in the client code, or simply omit them…

Monitor

An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.

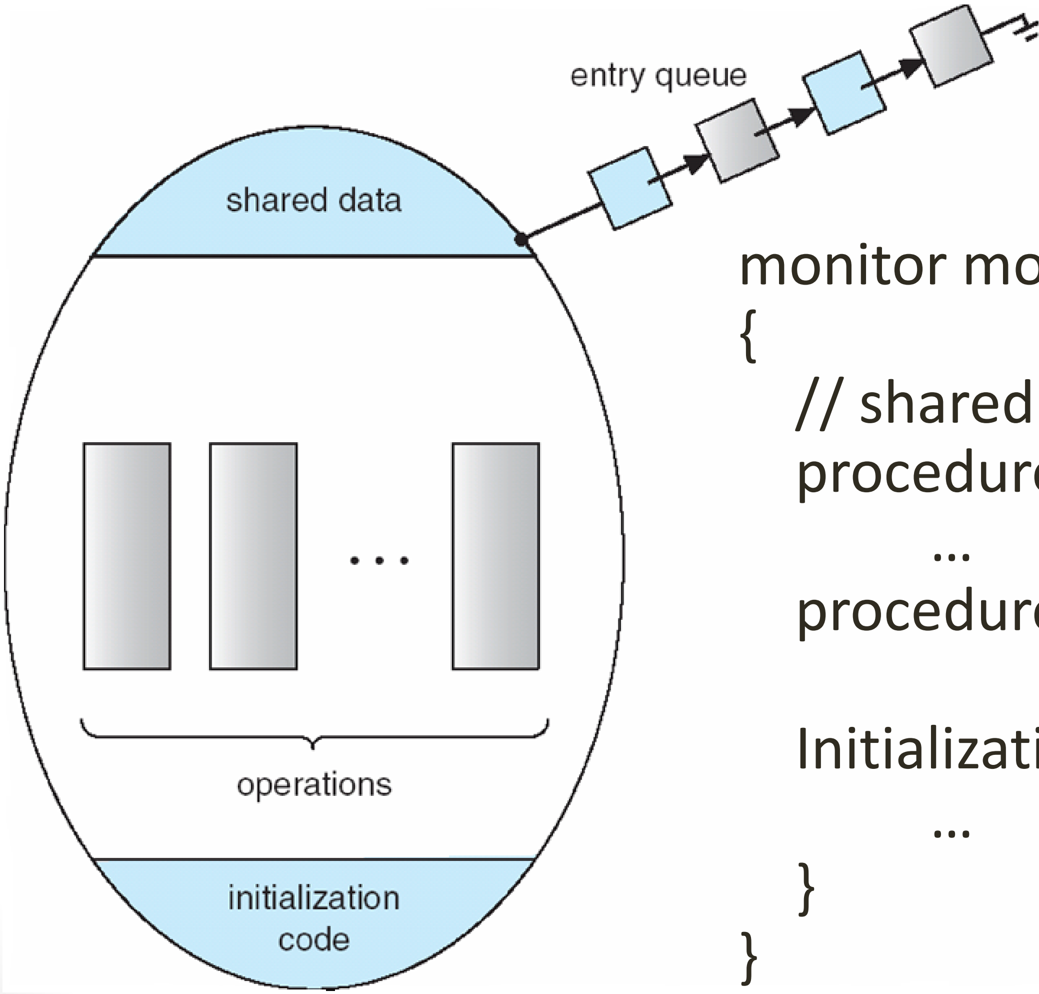
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization—the ***monitor type***.
  - Only one process may be active within the monitor at a time

- A ***monitor type*** is an ADT that includes a set of operations that are provided with mutual exclusion within the monitor.
  - The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

Monitor: A High-level Language Constructs

- The representation of a monitor type consists of
  - **declarations of variables** whose values define the state of an instance of the type
  - **procedures or functions** that implement operations on the type.
- A procedure within a monitor can access only variables defined in the monitor.
  - The local variables of a monitor can be used only by the local procedures.
- The monitor construct ensures that only one process at a time can be active within the monitor.
  - Consequently, the programmer does not need to code this synchronization constraint explicitly

Monitor



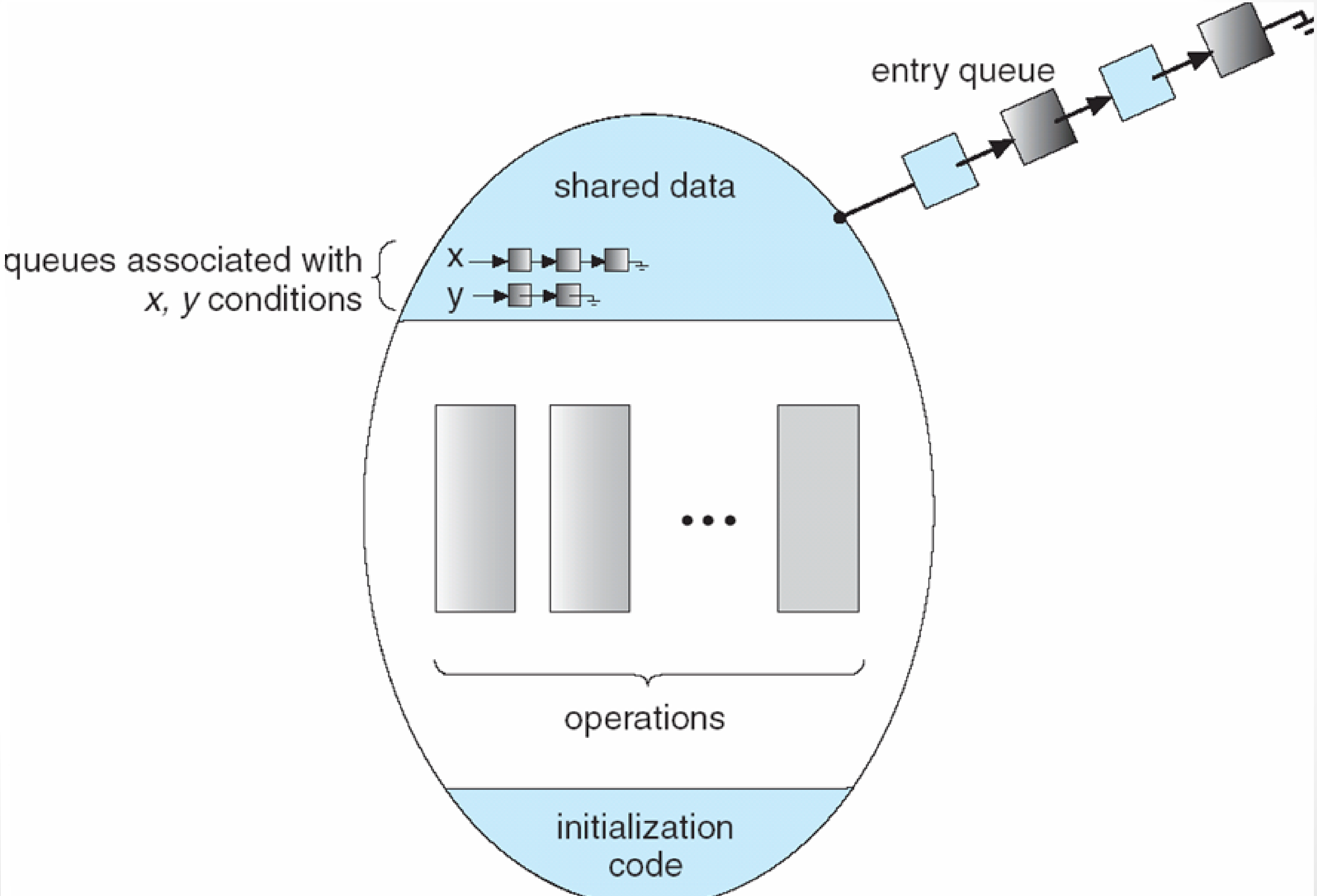
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }
    …
  procedure Pn (…) {……}

  Initialization code (….) { … }
    …
  }
}

Monitor

- We define additional synchronization mechanisms which provided by the **condition construct**.
  - A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:
  - condition x, y;
- Condition variables
  - x.wait () : a process that invokes the operation is suspended.
  - x.signal () : Resumes exactly one of processes (if any) that invoked x.wait ()
  - Contrast this operation with the signal() operation associated with semaphores, which always affects the state of the semaphore.

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including C#, Java

Resuming Processes within a Monitor

- If several processes queued on condition variable `x`, and `x.signal()` is executed, which process should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form `x.wait(c)`
  - Where `c` is **priority number**
  - Process with lowest number (highest priority) is scheduled next

Example with priority numbers

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
    ...
  access the resurce;
    ...
R.release;
```

- Where R is an instance of type **`ResourceAllocator`**

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
  boolean busy;
  condition x;
  void acquire(int time) {
          if (busy)
             x.wait(time);
          busy = true;
  }
  void release() {
          busy = FALSE;
          x.signal();
  }
   initialization code() {
   busy = false;
  }
}
```

the maximum time a process plans to use the resource

The monitor allocates the resource to the process that has the shortest time-allocation request

Solution to Dining Philosophers

- We illustrate **monitor** concepts by presenting a deadlock-free solution to the dining-philosophers problem.

- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

Solution to Dining Philosophers

- The solution
  - declare a monitor dp of type dining-philosophers

dp: dining-philosophers

  - To eat, philosopher P_i performs

```
dp.pickup(i);
    ...
   eat
    ...
dp.putdown(i);
```

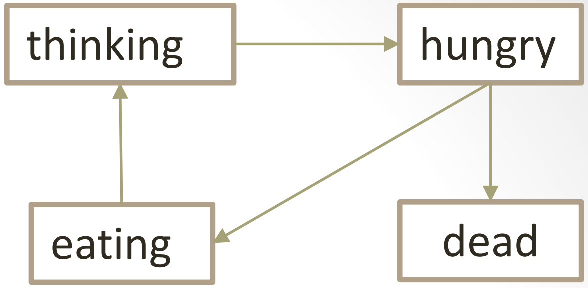
Solution to Dining Philosophers

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence

```
DP DiningPhilosophers;
DiningPhilosophers.pickup(i);
            ... eat ...
DiningPhilosophers.putdown(i);
            ... think ...
```

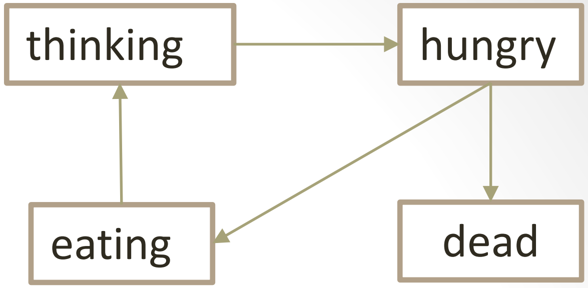
- Philosopher *i* can set the variable state[i] = EATING only if her two neighbors are not eating.

Solution to Dining Philosophers



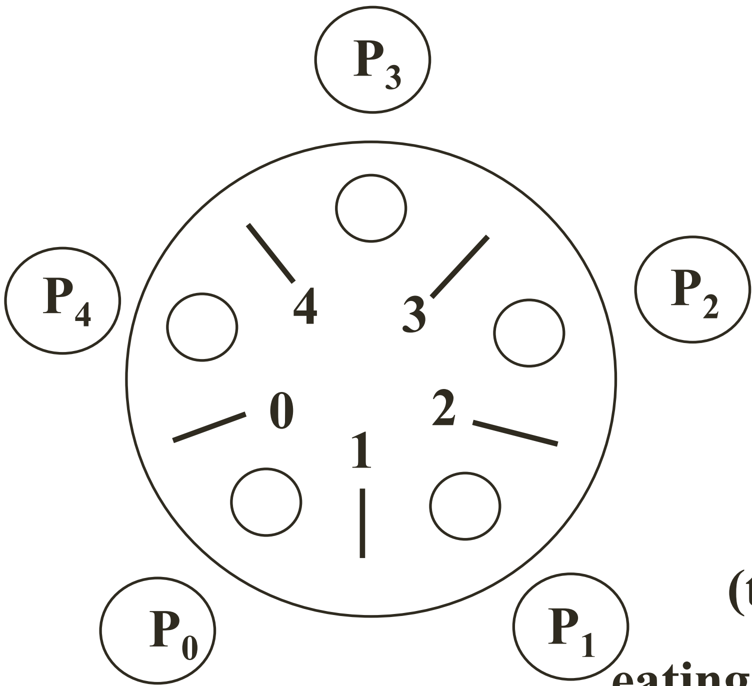

```
monitor DP
  {
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

    void pickup (int i) {  /*肚子餓撿起筷子*/
       state[i] = HUNGRY;
       test(i); //測試是否可以拿起筷子
       if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {  /*吃飽放下筷子*/
         state[i] = THINKING;
          // test left and right neighbors
          test((i + 4) % 5); //詢問旁邊是否要吃飯（把他們叫起來測試）
          test((i + 1) % 5); //詢問旁邊是否要吃飯
     }
     ....
```

Solution to Dining Philosophers




```
…
void test (int i) { /*測試是否可以拿筷子*/
    if ( (state[(i + 4) % 5] != EATING) && //看旁邊是否正在吃
     (state[i] == HUNGRY) && //自己是否餓了？
     (state[(I + 1) % 5] != EATING) //看旁邊是否正在吃
     ) {
           state[i] = EATING ;
           self[i].signal () ;
     }
 }

 initialization_code() {
     for (int i = 0; i < 5; i++)
          state[i] = THINKING;
 }
}
```

An illustration



(time 2)

hungry

self[2].wait;

(time 1)

eating;

self[1].signal; /* no effect */

thinking

(time 3)

pushdown(1)

test (0) ;

test (2) ;

if P_3 is not eating > self[2].signal

Monitor Implementation using Semaphores

- A possible implementation of the monitor mechanism using semaphores.
  - A binary semaphore **mutex** (initialized to 1) is provided to ensure mutual exclusion for each monitor.
  - A process must execute wait(**mutex**) before entering the monitor and must execute signal(**mutex**) after leaving the monitor

Monitor Implementation using Semaphores

- We will use the signal-and-wait scheme
  - Since a signaling process must wait until the resumed process either leaves or waits, an additional binary semaphore, **next**, is introduced, initialized to 0.
- The signaling processes can use **next** to suspend themselves. An integer variable **next_count** is also provided to count the number of processes suspended on next.

Monitor Implementation Using Semaphores

- Variables
  - semaphore mutex = 1; //是否可進入monitor
  - semaphore next = 0; //用來強迫P等待 **P: signaling processes**
  - int next-count = 0; //紀錄有多少個P在等待next

- For every condition x
  - semaphore x-sem = 0; //用來強迫Q block
  - int x-count = 0; //統計有多少個Q被卡住

P either waits until Q leaves the monitor or waits for another condition

Monitor Implementation Using Semaphores

Procedure F

```
wait(mutex);    沒有人在monitor 內活動才可進

…body of F…;

if (next_count > 0)
   signal(next);   若有其他Proc.存在，記得先解救他們
else
   signal(mutex); 若沒有，則開放monitor讓其他人進來
```

在範例中，**P**正結束工作，接著要**Signal**其他人。
若是有人可以**Singal**，就先把對方叫起來，並且自己等待
若沒有，則開放**monitor**讓其他人進來

Monitor Implementation Using Semaphores – condition x

x.wait

```
x-count++;    Q個數加1
if (next_count > 0) {signal(next)};    若有P存在，先解救P
else {signal(mutex)}    若沒有，讓其他進來
wait(x_sem);    Q等待condition x 而卡住
x-count--;    Q被救之後，Q數量減1
```

x.signal

```
if (x-count > 0) {    如果有Q被卡住才做
    next_count++; P 個數加1
    signal(x_sem); 解救Q
    wait(next);   P 被迫等待
    next_count--; P被救之後，P數量減1
}
```

在範例中，**P**正結束工作，接著要**Signal**其他人。
若是有人可以**Singal**，就先把對方叫起來，並且自己等待
若沒有，則開放**monitor**讓其他人進來

Monitor

- Process-Resumption Order
  - Queuing mechanisms for a monitor and its condition variables. One simple solution is to use a first-come, first-served (FCFS) ordering …

- The **conditional-wait** construct can be used to adequate more complicate case
  - x.wait(c)
    - C : **priority number,** stored with the name of the process that is suspended.
  - x.signal()
    - The process with the smallest priority number is resumed next

A Monitor to Allocate Single Resource

- To illustrate these, consider the **ResourceAllocator** monitor, which controls the allocation of a single resource among competing processes.
  - Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource.
  - The monitor allocates the resource to the process that has the shortest time-allocation request.

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
        boolean busy;
        condition x;
        void acquire(int time) {
            if (busy)
                x.wait(time);
             busy = TRUE;
        }
        void release() {
             busy = FALSE;
             x.signal();
        }
          initialization code() {
             busy = FALSE;
        }
}
```

```
R.acquire(t);
…
access the resource;
…
R.release;
```

time =
the maximum time it plans to use the resource

The monitor allocates the resource to the process that has the shortest time-allocation request.

Note! Defect still possible!

- The monitor concept cannot guarantee that the preceding access sequence will be observed.
  - A process might **access a resource without first gaining access permission** to the resource.
  - A process might **never release a resource** once it has been granted access to the resource.
  - A process might **attempt to release a resource that it never requested**.
  - A process might **request the same resource twice** (without first releasing the resource).

We need to make sure …

- First, user processes must always make their **calls on the monitor in a correct sequence**.
- Second, we must be sure that **an uncooperative process does not simply ignore the mutual-exclusion** gateway provided by the monitor and try to access the shared resource directly, **without using the access protocols**.
  - These treatment may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. – jmp to Chapter 17. for more detail.

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization

- Atomic variables
  `atomic_t` is the type for atomic integer

- Consider the variables
  `atomic_t counter;`
  `int value;`

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores can be used by unrelated processes, unnamed cannot.

POSIX **Named** Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX **Unnamed** Semaphores

- Creating an initializing the semaphore:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Condition Variables

- POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

POSIX Condition Variables

- Thread waiting for the condition **`a == b`** to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

Once this lock is acquired, the thread can check the condition.

releases the mutex lock, thereby allowing another thread to access the shared data and possibly update its value so that the condition clause evaluates to true.

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Note: the call to pthread_cond_signal() does not release the mutex lock.

Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns control from the call to pthread_cond_wait().

Java Synchronization

- Java provides rich set of synchronization features:
  - Java monitors
  - Reentrant locks
  - Semaphores
  - Condition variables

Java Monitors

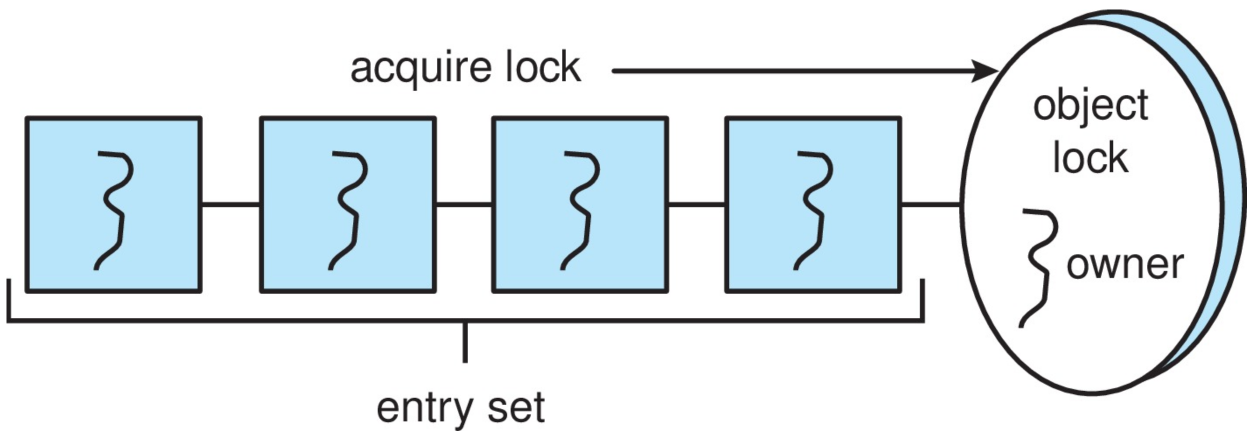
- Every Java object has associated with it a single lock.
- If a method is declared as **`synchronized`**, a calling thread must own the lock for the object.
- If the lock is owned by another thread, the calling thread must wait for the lock until it is released.
- Locks are released when the owning thread exits the **`synchronized`** method.

Bounded Buffer – Java Synchronization

```
public class BoundedBuffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */
   }
}
```

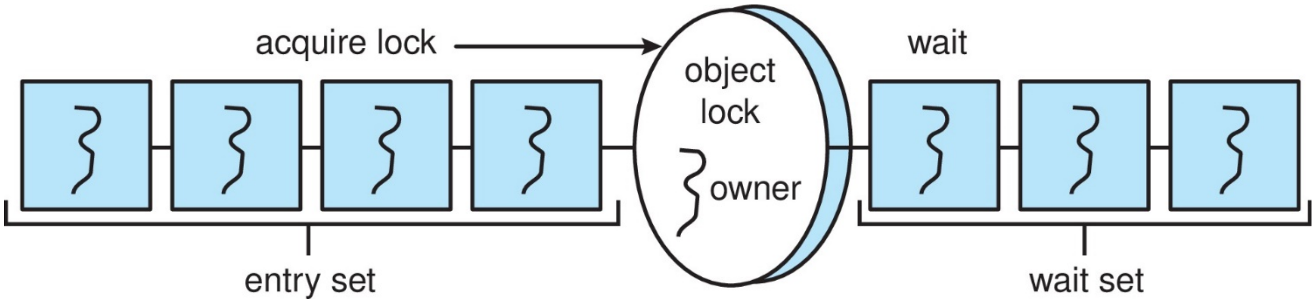
Java Synchronization

- A thread that tries to acquire an unavailable lock is placed in the object's **entry set**:



Java Synchronization

- Similarly, each object also has a **wait set**.
- When a thread calls **`wait()`**:
  1. It releases the lock for the object
  2. The state of the thread is set to blocked
  3. The thread is placed in the wait set for the object



Java Synchronization

- A thread typically calls wait() when it is waiting for a condition to become true.
- How does a thread get notified?
- When a thread calls **`notify()`**:
  - An arbitrary thread T is selected from the wait set
  - T is moved from the wait set to the entry set
  - Set the state of T from blocked to runnable.
- T can now compete for the lock to check if the condition it was waiting for is now true.

Bounded Buffer – Java Synchronization

```
/* Producers call this method */
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   count++;

   notify();
}
```

Bounded Buffer – Java Synchronization

```
/* Consumers call this method */
public synchronized E remove() {
   E item;

   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;

   notify();

   return item;
}
```

Java Reentrant Locks

- Similar to mutex locks.
- The **`finally`** clause ensures the lock will be released in case an exception occurs in the **`try`** block.

```
Lock key = new ReentrantLock();

key.lock();
try {
   /* critical section */
}
finally {
   key.unlock();
}
```

Java Semaphores

- Constructor:

```
Semaphore(int value);
```

- Usage:

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   /* critical section */
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Java Condition Variables

- Condition variables are associated with an **`ReentrantLock`**.
- Creating a condition variable using **`newCondition()`** method of **`ReentrantLock`**:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- A thread waits by calling the **`await()`** method, and signals by calling the **`signal()`** method.

Java Condition Variables

- Example:
- Five threads numbered 0 .. 4
- Shared variable **`turn`** indicating which thread's turn it is.
- Thread calls **`doWork()`** when it wishes to do some work. (But it may only do work if it is their turn.
- If not their turn, wait
- If their turn, do some work for awhile ……
- When completed, notify the thread whose turn is next.
- Necessary data structures:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
   condVars[i] = lock.newCondition();
```

Java Condition Variables

- Example:
- Five threads numbered 0 .. 4
- Shared variable **turn** indicating which thread's turn it is.

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
   condVars[i] = lock.newCondition();
```

- Necessary data structures:

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
  lock.lock();

  try {
    /**
     * If it's not my turn, then wait
     * until I'm signaled.
     */
    if (threadNumber != turn)
      condVars[threadNumber].await();

    /**
     * Do some work for awhile ...
     */

    /**
     * Now signal to the next thread.
     */
    turn = (turn + 1) % 5;
    condVars[turn].signal();
  }
  catch (InterruptedException ie) { }
  finally {
    lock.unlock();
  }
}
```

Transactional Memory

- The concept of transactional memory originated in **database** theory.
- A **memory transaction** is a sequence of memory read–write operations that are atomic.
  - As an alternative to traditional locking methods, new features that take advantage of transactional memory **can be added to a programming language.**
- It is only a concept

Transactional Memory

- Consider a function update() that must be called atomically. One option is to use mutex locks:

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically. A transaction can be completed by adding **`atomic{S}`** which ensure statements in **`S`** are executed atomically:

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```

Using OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
void update(int value)
{
   #pragma omp critical
   {
       count += value
   }
}
```

The code contained within the **`#pragma omp critical`** directive is treated as a critical section and performed atomically.

SUGGESTION!

OR

OBJECTION?

Let’s stop here,

TAKE A BREAK