政大資科系

作業系統

Operating System

廖峻鋒

cfliao@nccu.edu.tw

Operating System

Synchronization

Chun-Feng Liao

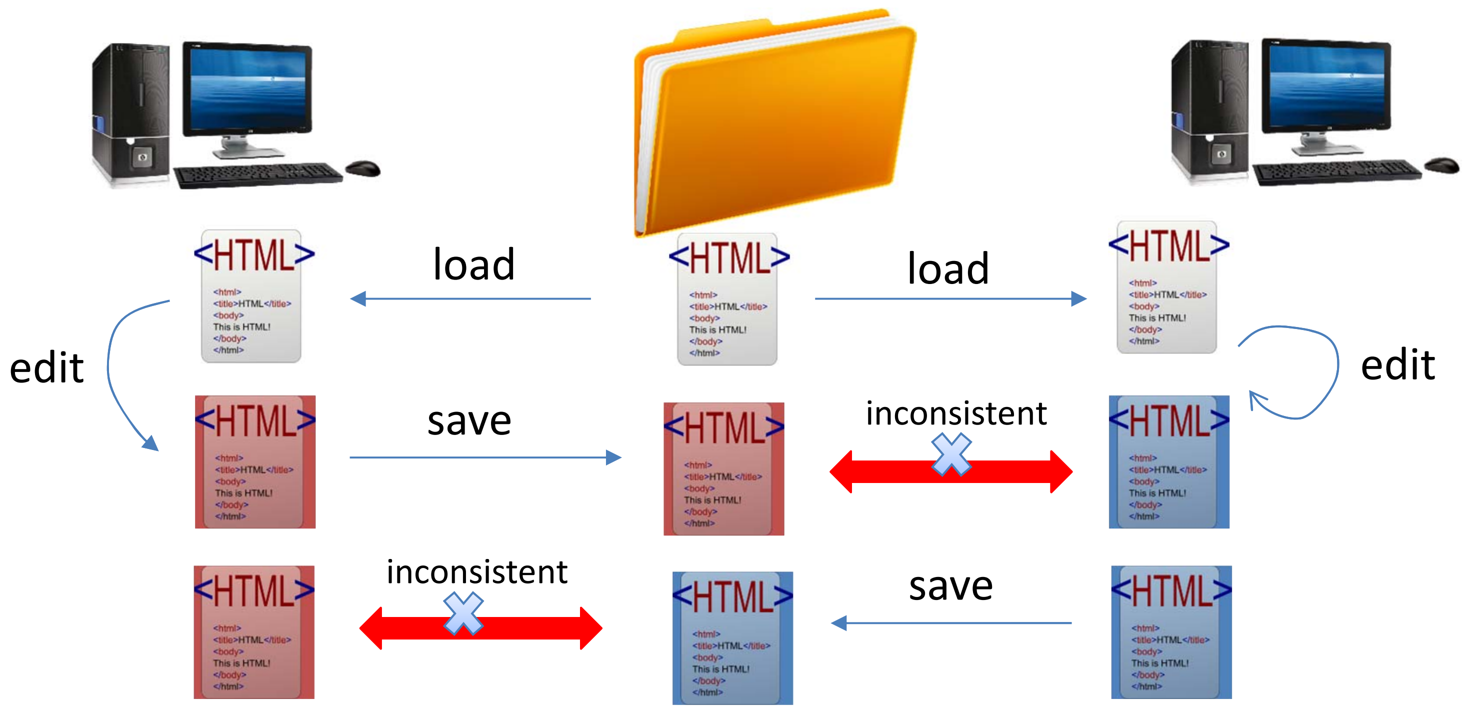
廖峻鋒

Department of Computer Science

National Chengchi University

Background

- Concurrent access to shared data may result in data inconsistency
  - 兩方同時access一個data，指令執行次序導致不同結果
  - 需要一個機制來規範讀寫次序，以利產生出一致的執行結果



Ex: Consumer-Producer Problem (改良版)

- Determine whether buffer is empty or full
  - Previously: use in, out position
  - Now: use a global count value (不浪費掉一個buffer space)

counter: 全域變數

二個process同時存取(concurrent shared data access)!

```
/*producer*/ while (1) {
    nextItem = getItem( );        滿
    while (counter == BUFFER_SIZE) ; buffer[in] = nextItem;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
/*consumer*/ while (1) {
                 空
    while (counter == 0) ; item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

addi $t0, $s0, 0
addi $t0, $t0, 1
addi $s0, $t0, 0

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

addi $t1, $s0, 0
addi $t1, $t1, -1
addi $s0, $t1, 0

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

讀取同一版本，各做不同修改後寫入，後寫入的勝利

Ensure the ordering

S0: producer execute **`register1 = counter`** {register1 = 5}
S1: producer execute **`register1 = register1 + 1`** {register1 = 6}
S2: producer execute **`counter = register1`** {counter = 6 }
S3: consumer execute **`register2 = counter`** {register2 = 6}
S4: consumer execute **`register2 = register2 – 1`** {register2 = 5}
S5: consumer execute **`counter = register2`** {counter = 5}

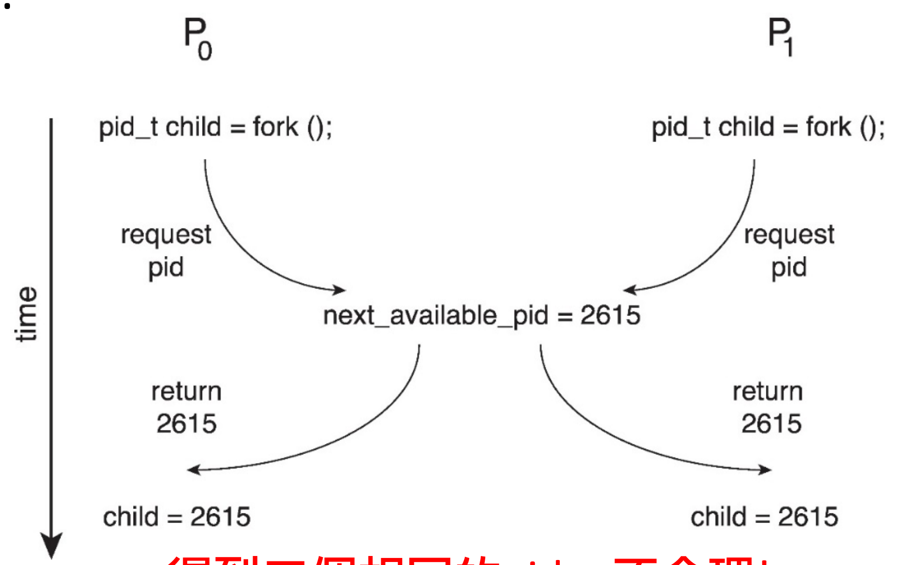
Race Condition

S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

- Definition
  - Several processes update shared data concurrently
  - The final value of the shared data depends upon which process finishes last
- To prevent race condition, concurrent processes must be synchronized
  - Commonly described as critical section problem

Race Condition in Kernel

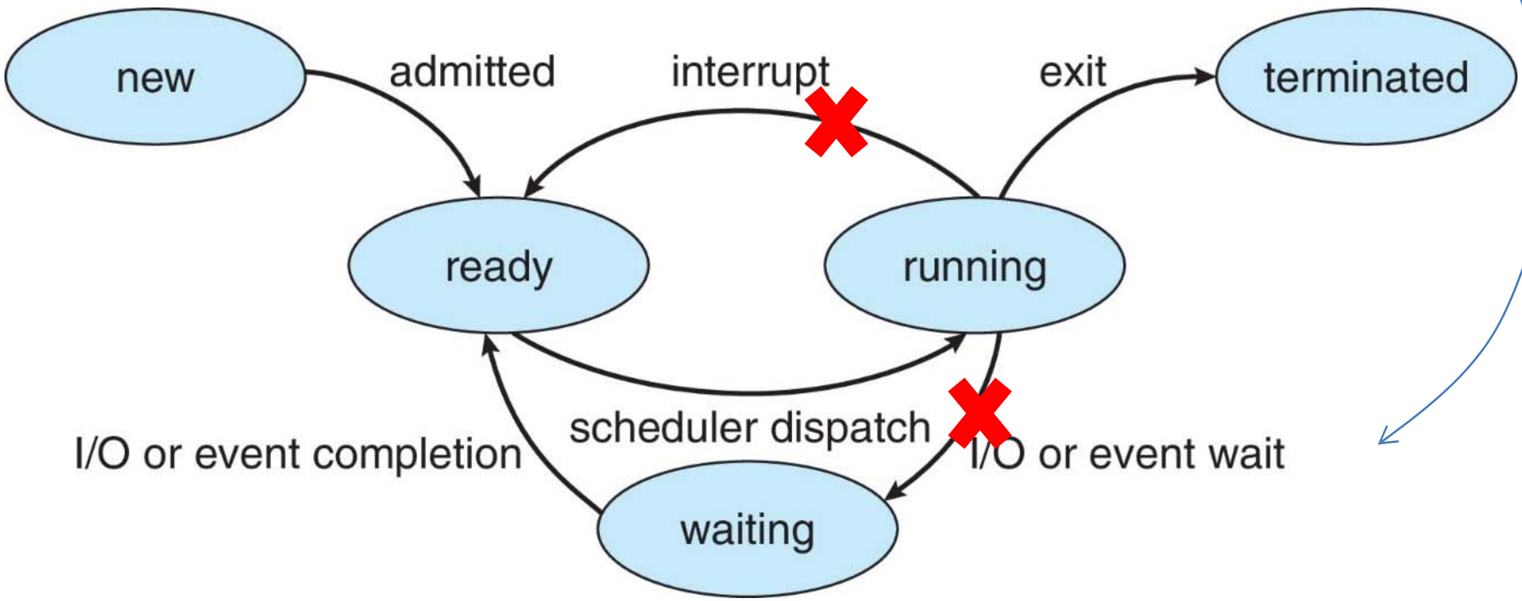
- Multiple kernel-mode processes may be active at the same time and subject to race conditions
- Ex: Processes P0 and P1 are creating child processes using fork()
  - Race condition on kernel variable next_available_pid
    - next available process identifier (pid)
  - Unless there is mutual exclusion, the same pid could be assigned to two different processes!



得到二個相同的pid，不合理!

Race Condition in Kernel

- Possible solution
  - Disable interrupt temporarily
  - 只有Single core才有效 (why?)



Race Condition in Kernel

- Two approaches to handle critical sections in kernel
  - Non-preemptive – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode
    - Multicore: disable all cores' interrupts is not feasible
      - poor efficiency and system clock problem

See P.262上

  - Preemptive kernel – allows preemption of process when they are running in kernel mode
    - More responsive
    - 會有Race Condition，需要同步演算法輔助
    - Multicore下preemptive kernel更難設計: See P.265 Memory Model
      - 一樣會有race condition
      - 不同process的code可能分散在不同core跑

Critical Section Problem

- General form:
  - N processes (threads) are competing to use the same shared data
  - Critical section (CS): the code segment that modifies the shared data
    - Only one process can be executed in CS

```
while (true) {
    entry section        Get entry permission
        critical section     Modify shared data
    exit section         Release entry permission
        remainder section
}
```

在概念層次，thread和process的解法相同，課本在本章中，以process來指稱process或thread

P.260

Critical Section Requirements

- Mutual Exclusion: if process P is executing in its CS, no other processes can be executing in their CS
  - 一次只讓一個process進CS
- Progress: if no process is executing in its CS and there exist some processes that wish to enter their CS 有意願才能參與競爭
  - Only the processes not in remainder section can enter next (見下頁範例)
- Bounded Waiting: A bound must exist on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS
  - 有限等待時間 (排隊)

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

CS問題→ How to design entry and exit section to satisfy the above requirement?

三個條件都要符合才可解此問題!

A Naïve Algorithm: 強制輪流法

- Only 2 processes, P_0 and P_1
- Shared variables
  - int turn; // initially turn = 0 現在誰有權利進CS
  - turn = i; // Pi can enter its critical section

```
/* Process i */
while(true) {
    while (turn != i) ;
        critical section
    turn = j;
        remainder section
}
```

entry section

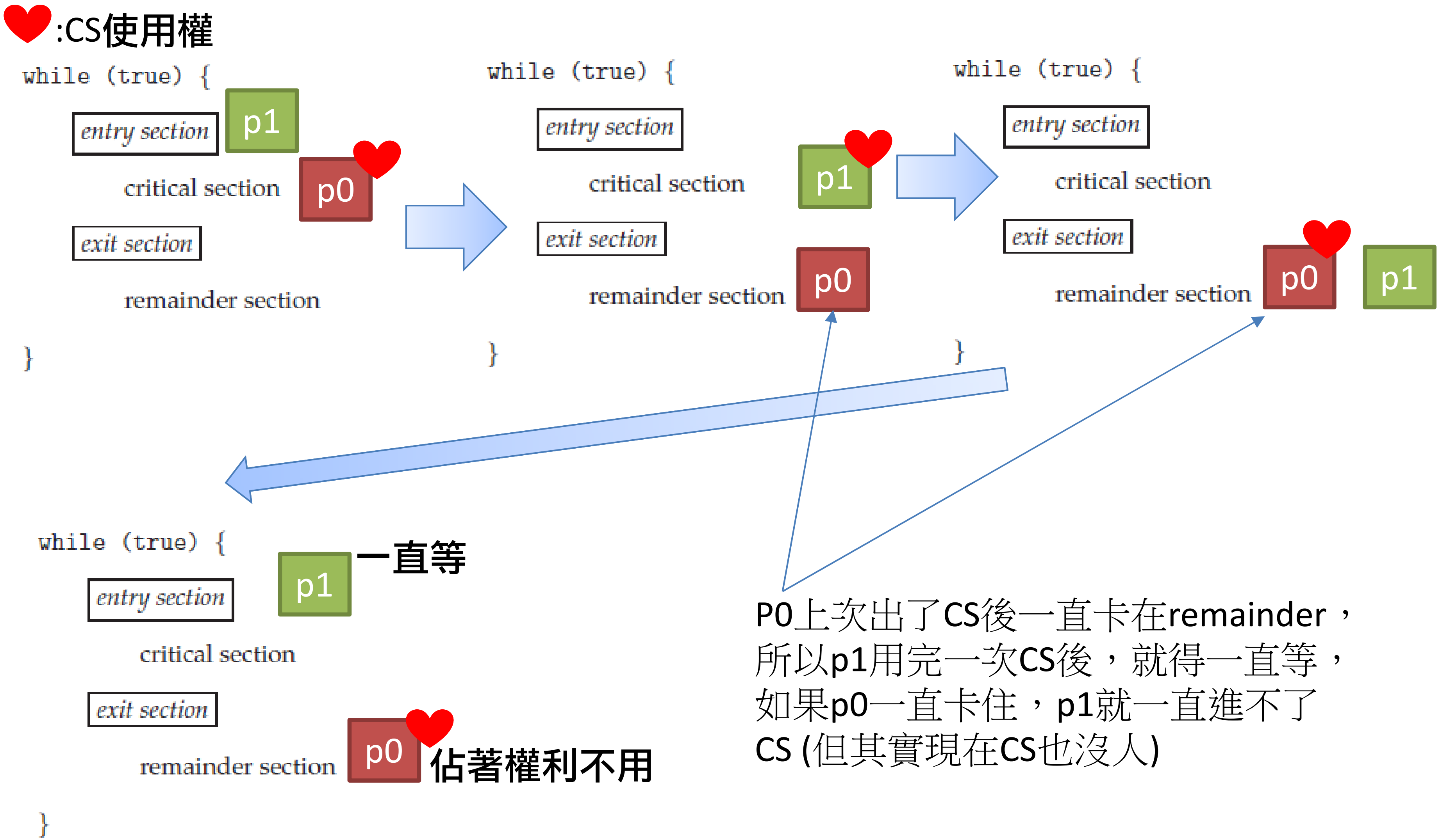
exit section

```
/* Process j */
while(true) {
    while (turn != j) ;
        critical section
    turn = i;
        remainder section
}
```

Mutual exclusion=Yes; Progress=No; Bounded-Wait=Yes;

若輪到的一方一直無意願，另一方就要一直等

p0、p1約好「輪流」進CS



就算有權利使用，如沒有意願用，也不能佔著不放 (progress)
離開remainder，**進**entry section**才代表有意願**

Peterson's Solution

- int turn; // (馬上) 進CS的權利
- boolean flag[2]; // (馬上) 進CS的意願

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

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

表達有意願

權利給另一位

只要另一位「沒權利」或「沒意願」就可以進入

如果另一位(j)有權利也有意願進CS，就要等

可解naïve solution的progress問題，因為若j還卡在remainder，那麼它一定沒意願進CS (flag[j]==false)

表達沒意願了!

這行會導致P在remainder section時，都被標記為沒意願

```
/* process 0 */
while(1) {
  flag[ 0 ] = TRUE;
  turn = 1 ;
  while (flag [ 1 ] && turn == 1 ) ;
     // critical section
  flag [ 0 ] = FALSE ;
     // remainder section
}
```

```
/* process 1 */
while(1) {
  flag[ 1 ] = TRUE;
  turn = 0 ;
  while (flag [ 0 ] && turn == 0 ) ;
     // critical section
  flag [ 1 ] = FALSE ;
     // remainder section
}
```

flag[]意願; turn權利

是否符合Mutual Exclusion?
假設不符合Mutual Exclusion，則P0與P1同時在CS

此時，flag[0] == flag[1] == true (二者都有意願進CS)
又，依據程式:
If P1 in CS → 必須不符合while→ flag[0] == false or turn == 1 → turn ==1
If P0 in CS → 必須不符合while→ flag[1] == false or turn == 0 → turn ==0

→ P0 and P1 in CS → turn ==0 and turn ==1 (矛盾)

flag[]意願; turn權限

```
/* process 0 */
while(1) {
  flag[ 0 ] = TRUE;
  turn = 1 ;
  while (flag [ 1 ] && turn == 1 ) ;
    // critical section
  flag [ 0 ] = FALSE ;
    // remainder section
}
```

flag[]意願; turn權限

```
/* process 1 */
while(1) {
  flag[ 1 ] = TRUE;
  turn = 0 ;
  while (flag [ 0 ] && turn == 0 ) ;
    // critical section
  flag [ 1 ] = FALSE ;
    // remainder section
}
```

Progress: 如果CS中沒人，而且有人有意願進CS，則只有不在Remainder的人才有資格下一個進去

是否符合Progress?

如果CS中沒人，且P0有意願(flag[0]==true; turn ==1)

(1) P1**在**remainder(**沒意願**)，P0**能進去** (**因為**flag[1]==false)

(2) P1**不在**remainder，**代表二人都有意願，則**(flag[0]==flag[1]==true)，**此時誰先跑到**while**就能先進去** (turn **不是**0**就是**1，**後跑到的會將權利**turn**設給另一人**)

在二種情況下，只要有人有意願都一定會有其中一人能進CS

```
/* process 0 */                flag[]意願; turn權限
while(1) {
  flag[ 0 ] = TRUE;
  turn = 1 ;
  while (flag [ 1 ] && turn == 1 ) ;
     // critical section
  flag [ 0 ] = FALSE ;
     // remainder section
}
```

```
/* process 1 */                flag[]意願; turn權限
while(1) {
  flag[ 1 ] = TRUE;
  turn = 0 ;
  while (flag [ 0 ] && turn == 0 ) ;
     // critical section
  flag [ 1 ] = FALSE ;
     // remainder section
}
```

Progress: 如果CS中沒人，而且有人有意願進CS，則只有不在Remainder的人才有資格下一個進去

是否符合Bounded waiting 如果有意願，則不能無限等待(被插隊)?

假設P1在CS中，且P0有意願，卡在while loop
若要不符合Bounded waiting，代表P1想插隊，此時P1出CS後，在P0未進入CS前，又快速走到P1的while (此時turn ==0, flag[0] == flag[1] == 1)→P1一定會被卡住→P0進CS。

反之若P0在CS中，且P1有意願，卡在while loop，同上理由P1也終究能進CS

Limitation of Peterson's Solution

- Limitations
  - Only two process
  - Not guaranteed to work on modern architectures
    - Because of reordering of instructions (in CPU pipeline)

Reordering Instructions

- 假設二個threads同時存取下列全域變數:

```
boolean flag = false;
int x = 0;
```

- Thread 1

```
while (!flag)
    ; // wait for flag==true
print x;
```

- Thread 2

```
x = 100;
flag = true; // flag == true after assignment
```

- What is the expected output? x=100 or x=0?
  - The operations for Thread 2 may be reordered

若T2二個指令對調，結果就要看是print x先或是x=100先被執行 ➔ race condition

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code :

```
a = b + e;
c = b + f;
```

若緊接著load就要用，則會stall一個stage (即使用forwarding)

調整後，二個lw指令的結果都至少一個指令後才被用到，所以沒有stall

stall

stall

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

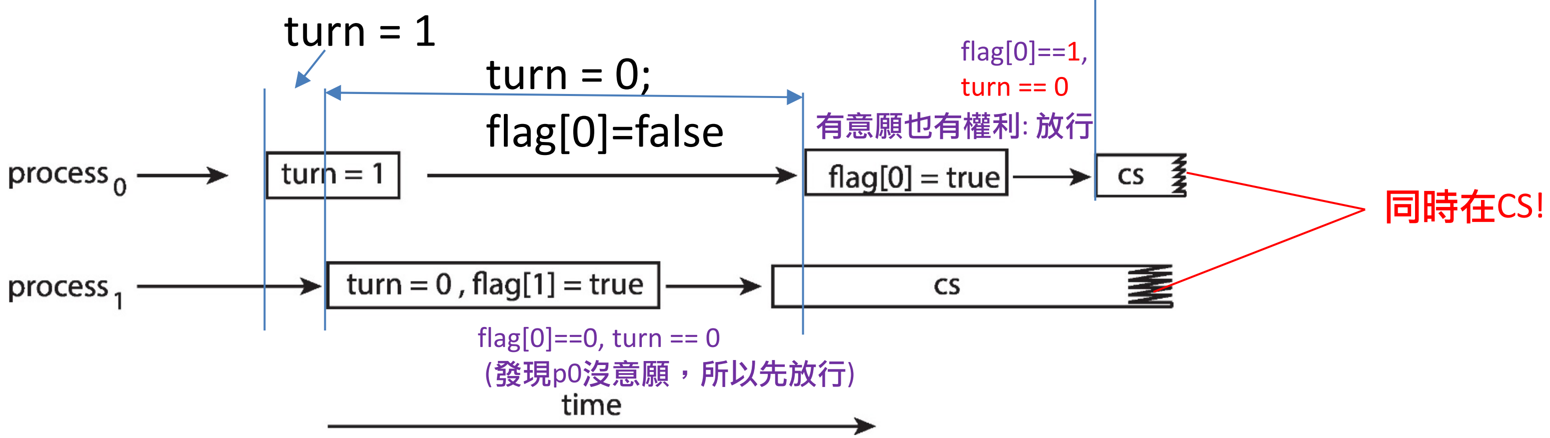
13 cycles

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

11 cycles

Reordering Instructions

- This allows both processes to be in their critical section at the same time!




```
/* process 0 */
while(1) {
  flag[ 0 ] = TRUE;  (4)
  turn = 1 ;  (1)
  while (flag [ 1 ] && turn == 1 ) ;
     // critical section
  flag [ 0 ] = FALSE ;
     // remainder section
}
```

```
/* process 1 */
while(1) {
  flag[ 1 ] = TRUE;(3)
  turn = 0 ; (2)
  while (flag [ 0 ] && turn == 0 ) ;
     // critical section
  flag [ 1 ] = FALSE ;
     // remainder section
}
```

Synchronization by Hardware

- Two types
  - 確保二個外部指令間的次序
    - Memory barriers
  - 確保一群內部指令的次序
    - test_and_set
    - compare_and_swap
    - Atomic variable (以compare_and_swap實現)

Memory barrier

- 功能
  - An instruction that forces any change in memory to be propagated (made visible) to all other processors
  - Ensure all loads and stores are completed in order

要確保二道指令的次序，就在中間加一個
memory_barrier()

- Thread 1

```
while (!flag);
memory_barrier(); // flag load before x load
print x
```

- Thread 2

```
x = 100;
memory_barrier(); // x store before flag store
flag = true
```

Peterson’s Solution with Memory Barrier

- /* process 0 */

```
while(true) {
  flag[ 0 ] = TRUE;
  memory_barrier(); // flag store first, then turn is stored
  turn = 1 ;
  while (flag [ 1 ] && turn == 1 ) ;
  //critical section
  flag [ 0 ] = FALSE ;
  //remainder section
}
```

要確保二道指令的次序，就在中間加一個 memory_barrier()

Hardware Instructions

- Test-and-Set
  - Test-and modify the content of a word atomically

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = true;
        return rv:
}
```

test_and_set是一道atomic指令
方框內的功能由硬體確保一次做完不中斷

· 回傳target的上一個狀態
· 執行指派true的動作 (set true)

boolean lock;

test_and_set(&lock); // **如果**lock == false; **回傳**false; **改成**true

// **如果**lock == true; **回傳**true

// 如果已上鎖，執行了也沒用; 如果未上鎖，執行了就會鎖
// 第一個讀到lock==false的人，可以得到false，然後就上鎖了

Hardware Instructions

```
while (true)
{
    while (test_and_set(&lock))
      ; /* do nothing */
        /* critical section */
    lock = false;
        /* remainder section */
}
```

MUTEX: Yes (只會有一個人拿到false回傳值，在此之後除非它出CS，不然lock永遠true)
Progress: Yes (進remainder前會release lock給別人)
Bounded-wait:No (無法掌握誰先得到lock，有可能某人一直搶不到)

Hardware Instructions

- Compare-and-swap (CAS)
  – Swap the contents of two words atomically

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

test_and_set的泛化版
- 回傳狀態值
- 合條件才設值; 可自由指定所設之值:
  如果value == expected，就將value設為new_value
- 在這裡的用途和test_and_set相同

```
int lock = 0;                      // lock==1 代表上鎖
compare_and_swap(&lock, 0, 1);     // 如果lock == 0; 回傳0; 改成1
                                   // 如果lock == 1; 回傳1;
```

Hardware Instructions

一旦拿到key (lock==1)，lock馬上又會鎖住，
因為是一次完成，所以只會有一個人拿到

Lock==0:**解鎖**

```
while (true)
{
    while (compare_and_swap(&lock, 0, 1) !=0 )
      ; /* do nothing */
        /* critical section */
    lock = 0; //用完解鎖
        /* remainder section */
}
```

MUTEX: Yes (**只會有一個人拿到0回傳值，在此之後除非它出CS，不然lock永遠不為0**)
Progress: Yes (**只要進**remainder**就會**release lock**給別人**)
Bounded-wait:No (**無法掌握誰先得到0回傳值**)

Implementation

- Intel x86 instruction for compare and swap
  - Lock cmpxchg <dest register> <source register>

https://c9x.me/x86/html/file_module_x86_id_41.html

Opcode	Mnemonic	Description
0F B0 /r	CMPXCHG r/m8,r8	Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL.
0F B1 /r	CMPXCHG r/m16,r16	Compare AX with r/m16. If equal, ZF is set and r16 is loaded into r/m16. Else, clear ZF and load r/m16 into AX
0F B1 /r	CMPXCHG r/m32,r32	Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into EAX

解 Bounded-waiting

boolean waiting[n];
int lock=0;//解鎖

```
while (true) {
   waiting[i] = true; 是否有意願
   key = 1; 是否上鎖
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;

   /* critical section */

   j = (i + 1) % n; 找下個
   while ((j != i) && !waiting[j])
      j = (j + 1) % n; 再找下一個
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;

   /* remainder section */
}
```

Key = lock 的狀態

是否全找完

下個是否有意願

全部卡住，直到
waiting[i]被改為false或lock被改成0

依序找出下一個有意願的人，找到就跳出

沒找到，就解鎖lock (隨便放行一人進CS)
有找到，就解鎖那個人進CS

具有針對特定人放行的功能

Solution with CAS

boolean waiting[n];
int lock=0;

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)

      key = compare_and_swap(&lock,0,1);

   waiting[i] = false;

   /* critical section */

   j = (i + 1) % n;

   while ((j != i) && !waiting[j])

      j = (j + 1) % n;

   if (j == i)

      lock = 0;

   else

      waiting[j] = false;

   /* remainder section */

}
```

全部卡住，直到
waiting[i]被改為false或lock被改成0

Mutual Exclusion: 一定要有Process離開CS才可能有人被放行

依序找出下一個卡住的人，找到就跳出

沒找到，就解鎖lock (隨便放行一人進CS)
有找到，就解鎖那個人進CS

Solution with CAS

```
boolean waiting[n];
int lock=0;
```

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)

      key = compare_and_swap(&lock,0,1);

   waiting[i] = false;

   /* critical section */

   j = (i + 1) % n;

   while ((j != i) && !waiting[j])

      j = (j + 1) % n;

   if (j == i)

      lock = 0;

   else

      waiting[j] = false;

   /* remainder section */

}
```

要到這裡才可能有意願

全部卡住，直到
waiting[i]被改為false或lock被改成0

Progress: 只要有意願進CS的人，一定會有人進

進CS前就解除意願

依序找出下一個卡住的人，找到就跳出

經過這個步驟，有意願的其中一個
一定被選到

沒找到，就解鎖lock (隨便放行一人進CS)
有找到，就解鎖那個人進CS

Solution with CAS

boolean waiting[n];
int lock=0;

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)

     key = compare_and_swap(&lock,0,1);

  waiting[i] = false;

  /* critical section */

  j = (i + 1) % n;

  while ((j != i) && !waiting[j])

     j = (j + 1) % n;

  if (j == i)

     lock = 0;

  else

     waiting[j] = false;

  /* remainder section */

}
```

全部卡住，直到
waiting[i]被改為false或lock被改成0

依序找出下一個卡住的人(waiting[j]==true)，
找到就跳出

Bounded wait: 依次序，所以waiting的一定輪得到

沒找到，就解鎖lock (隨便放行一人進CS)
有找到，就解鎖那個人進CS

P.270

Mutex (Spinlock)

- **一般應用程式開發人員不會直接接觸到上述的**HW instruction**，通常會透過一些較高階的機制**
- Mutex: a high-level tool to solve critical section problem
  - **適合資源**(cs)**一次只會被佔用一小段時間的情況**
    - <2 context switches P.272
  - Protect a critical section by first acquire() a lock then release() the lock
  - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() is atomic
  - Implemented via hardware instructions
  - Typically needs <u>busy waiting</u>: spinlock
- Spinlock
  - In spinlock: Waste CPU cycles;
    But, no context switch→save switching time
  - Widely used in modern multicore computer
    - 因為core多不怕少數被暫時佔用

```
while (true) {
    acquire lock
    critical section
    release lock
    remainder section
}
```

Mutex Lock Definitions

- ```
  acquire() {
      while (!available)
  ```
  Blocked until the lock is released
  ```
          ; /* busy wait */
      available = false;
  }
  ```
- ```
  release() {
      available = true;
  }
  ```

```
acquire() {
    while (test_and_set(&lock))
      ; /* busy wait */
}

release() {
    lock = false;
}
```

These two functions must be implemented atomically.
Both test-and-set and compare-and-swap can be used to implement these functions

POSIX Mutex Lock

- 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

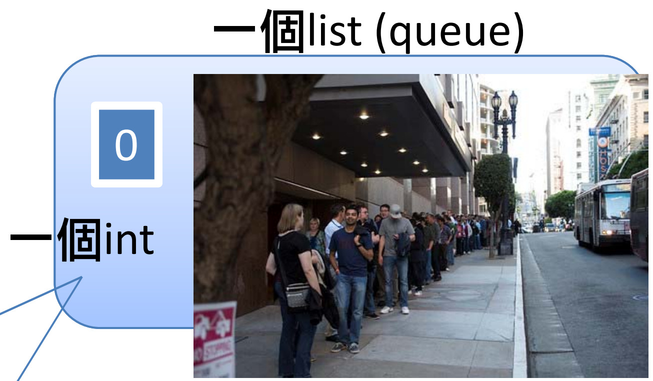
```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

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

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex);
 while (true)
 {
        pthread_mutex_lock (&mutex);
               /* critical section */
        pthread_mutex_unlock (&mutex);
               /* remainder section */
}
```

Why? 幫mutex加個queue排隊如何?

Mutual Exclusion: Yes; Progress: Yes; Bounded waiting: No

Semaphore



- A tool for the synchronization problem
  - easier to solve, but no guarantee for correctness
- 語意: 管控資源的計數器 (#record)
  - 計數器=多少資源可用
  - <=0時，需要排隊等待
- Two types
  - Binary semaphore: #record equals to either 0 or 1

    管控一個資源
  - Counting semaphore
    - #record = how many instances of a particular resource are available

      管控多個資源
    - If #record > 1 → more than one available
    - If #record = 1 → exactly one
    - If #record = 0 → not available (缺貨，無人排隊等待)
    - If #record < 0 → someone waiting for (缺貨，有人排隊等待，負值代表多少人在queue等)

      (或想成缺少資源的數量)
- Accessed through 2 atomic operations: wait & signal

Semaphore 概念

- 概念示意 (Spinlock) :
  - Semaphore S is typically an integer
  - 下列程式碼僅示意計數器的控制，未包含Queue

Binary Semaphore: S == 1 or S==0

spinlock

```
wait (S) {
    while (S <= 0) ; // 大於0才繼續
    S--;
}
```

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

Semaphore Usage

- 假設有 P_1 與 P_2 二個processes;希望 S_1 執行完再執行 S_2

  此時: Create a semaphore “**`record`**” initialized to 0

  P1:

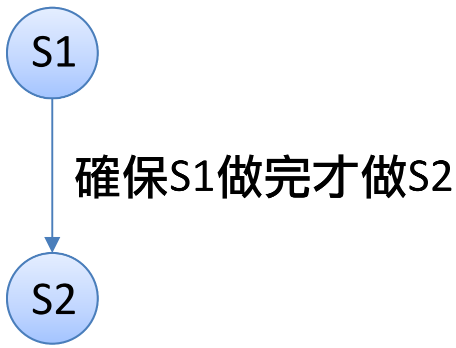
      S_1;

      `signal(record);` //S1做完才開放（1）

  P2:

      `wait(record);` // 一開始鎖住(0)

      S_2;




```
wait (S) {
    while (S <= 0) ;
    S--;
}
```

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

Semaphore Implementation

- Non-busy/waiting Implementation
  - Can use any queuing strategy that meets <u>bounded waiting RQ</u>

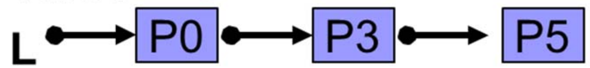
```
typedef struct {
        int value;
        struct process *list; // PCB list
} semaphore;
```

負值代表: 多少人在等此資源

在等的人的waiting queue

E.g.,: Counting semaphore: 少幾個資源就有幾個人等

Value = -3



  - 如何使用Semaphore?
    - wait() and signal(): 見下頁

Semaphore Implementation

Must be atomic!

```
wait(semaphore *S) {
    S->value--; // 修改計數器，負值代表等待的人數
    if (S->value < 0) {
        add this process to S->list;
                                   // 如果要等，就排隊(added to list)並等(sleep)
        sleep();
    }
}

signal(semaphore *S) {
    S->value++; // 調整計數器(自己不需要)
    if (S->value <= 0) { // <=0代表還有人在等
        remove a process P from S->list;
                                   // 喚醒下一個，並將它移出排隊隊伍
        wakeup(P);
    }
}
```

Semaphore Implementation

- How to ensure atomic wait & signal ops?
  - Single-core: disable interrupts
  - Multi-core: spinlock
    - SW solution (busy waiting - spinlock)
      - 若佔有資源時間短，使用spinlock還是可行的
    - HW support (e.g. Test-And-Set, CAS) 實作上其實也是busy waiting

用法:
n-Process Critical Section Problem

- Shared data:

```
semaphore sem ;      // initially sem = 1
```

- Process P_i:

```
while(1) {
   wait (sem) ;
        // critical section
   signal (sem);
        // remainder section
}
```

Mutex? Yes; Progress? Yes; (出CS後，只有到wait才能再排隊)
Bounded waiting? Yes (queue不可插隊)

POSIX Semaphore

- POSIX Semaphore routines:

設定semaphore是否可讓不同process使用(否則只能同屬一個process的thread使用)

  - sem_init(sem_t *sem, int pshared, unsigned int value)

Initial value of the semaphore
一開始有多少資源?

  - sem_wait(sem_t *sem)
  - sem_post(sem_t *sem) // **等同於** signal
  - sem_getvalue(sem_t *sem, int *valptr)

Current value of the semaphore

  - sem_destory(sem_t *sem)

Example:

```
#include <semaphore.h>
sem_t  sem;
sem_init(&sem);
sem_wait(&sem);
// critical section
sem_post(&sem);
sem_destory(&sem);
```

Java Semaphores

- Constructor:

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

問: Java Semaphore 實作上是Spinlock或notify?
答: 二種都支援
acquire() => notify
acquireUninterruptibly() => spinlock

- Usage:

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

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

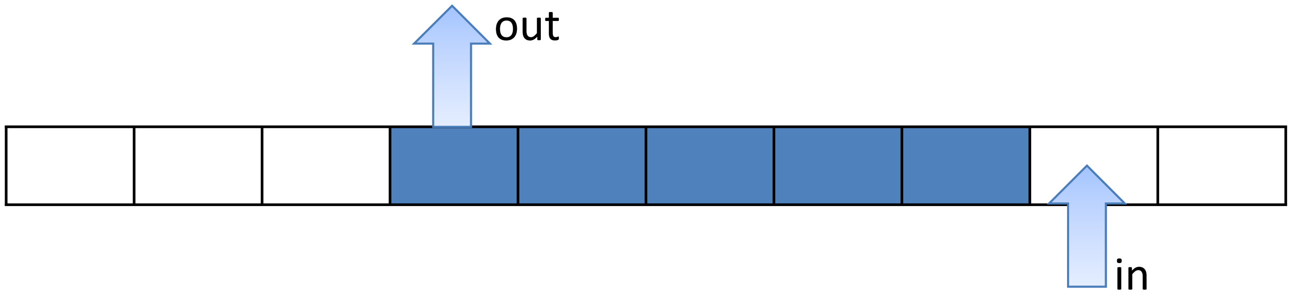
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Semaphore.html

範例1: Bounded-Buffer Problem

- A pool of n buffers, each capable of holding one item
- Producer:
  - grab an empty buffer
  - place an item into the buffer
  - waits if no empty buffer is available
- Consumer:
  - grab a buffer and retracts the item
  - place the buffer back to the free pool
  - waits if all buffers are empty

Bounded-Buffer Problem

```
while (true) {
      while (count == 0)
         ; /* do nothing */

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

      /* consume the item in next_consumed */
}
```

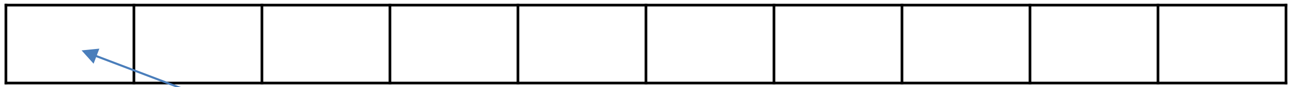



```
while (true) {
      /* produce an item in next_produced */

      while (count == BUFFER_SIZE)
         ; /* do nothing */

      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      count++;
}
```

Bounded-Buffer Problem

為什麼同時要有empty和full?
答: 對生產者與消費者來說，「資源」的意義不同!

int n = 10; // buffer size
semaphore mutex = 1; // **控制**CS (boolean)
semaphore empty = n; // **管控空位(個數)資源**
semaphore full = 0; // **管控產品(個數)資源**



producer

```
while (true) {
    . . .
  /* produce an item in next_produced */
    . . .
  wait(empty);  等到空位(個數)>0
  wait(mutex);
    . . .
  /* add next_produced to the buffer */
    . . .
  signal(mutex);
  signal(full);  產品++
}
```

consumer

```
while (true) {
  wait(full);   等到產品(個數)>0
  wait(mutex);
    . . .
  /* remove an item from buffer to next_consumed */
    . . .
  signal(mutex);
  signal(empty);  空位++
    . . .
  /* consume the item in next_consumed */
    . . .
}
```

範例2: Readers-Writers Problem

- A shared database for shared processes
- The group of processes
  - Reader (read): Many reader can access at the same time
  - Writer (read/ write): Only one writer can access at a time
- Different variations involving priority
  - First RW problem: the writer may starve
    - 一旦reader取得mutex，所有reader讀完才會release lock
  - Second RW problem: the readers may starve
    - Once a writer is ready, no new reader may start reading

First Readers-Writers (Writer may starved!)

```
semaphore rw_mutex = 1;   // 用來保護database Reader/Writer共享
semaphore rc_mutex = 1;   // 用來保護readcount Reader間共享
int readcount = 0;
```

```
// reader
while (true) {
    wait(rc_mutex);
    readcount++;
    if (readcount == 1)   // first reader should
        wait(rw_mutex);   // acquire the lock
    signal(rc_mutex);
. . .
/* reading is performed */
. . .
    wait(rc_mutex);       // 用來保護readcount
    readcount--;
    if (readcount == 0)   // last reader should
        signal(rw_mutex); // release the lock
    signal(rc_mutex);
}
```

```
// writer
while (true) {
    wait(rw mutex);
    . . . /* writing is performed */
    signal(rw mutex);
}
```

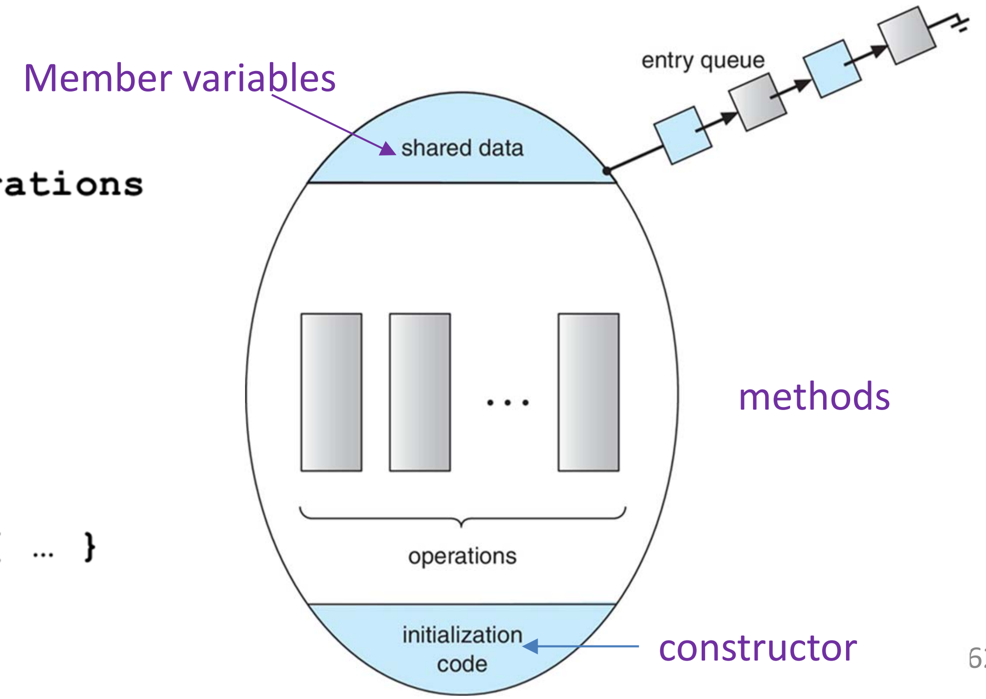
The Need for Modular Sync. Mechanisms

- Correct usage of semaphore is depending on the developer
  - must execute wait() and signal() in the right order and right place
- Incorrect use of semaphore
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) and/or signal (mutex)

Monitor

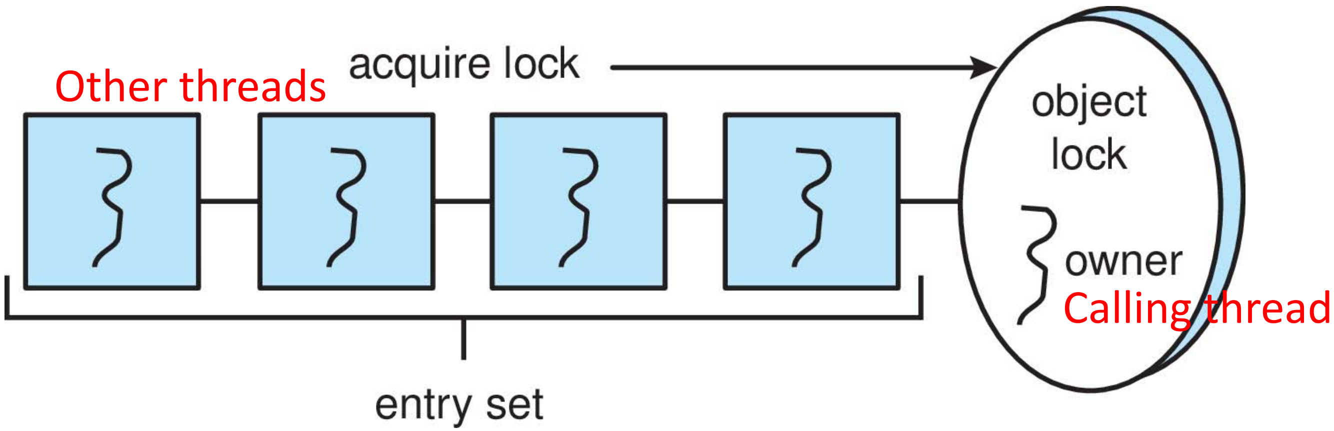
- A high-level language construct
  - Similar to a class in OO language
  - The member function can access only
    - Member variables in the monitor
    - Variables passed via function parameters
  - Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

  initialization code (…) { … }
}
```



Java Monitors

- Every Java object has associated with it a single lock
- To enable a monitor
  - Just declare one of the method as synchronized
  - Usage
    - Calling thread own the lock (for this object)
    - Other threads wait in an entry set

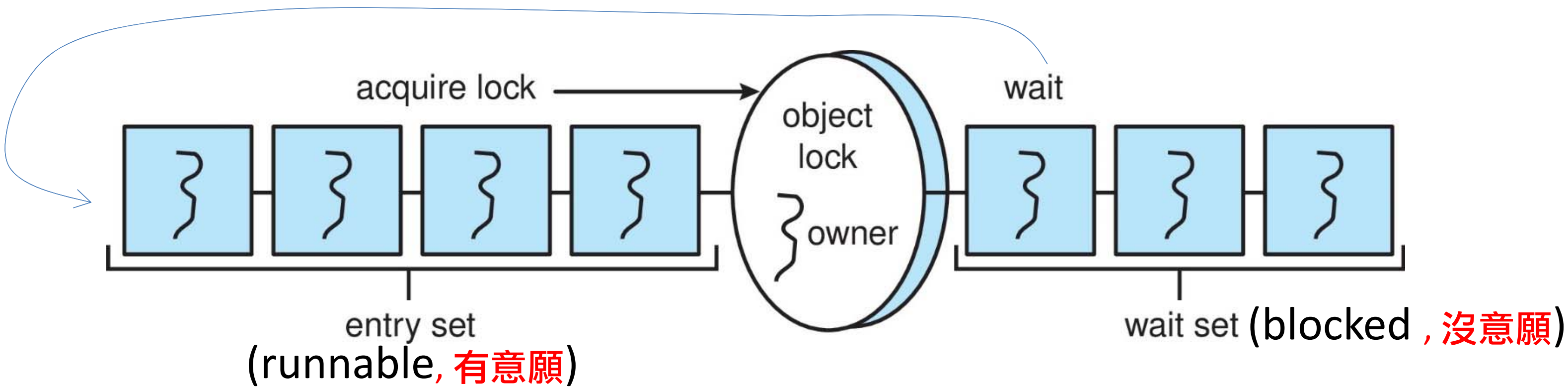


Wait Set

- A thread typically calls wait() when it is waiting for a condition to become true

曾取得過object，但執行到
一半因故暫時unlock

- 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



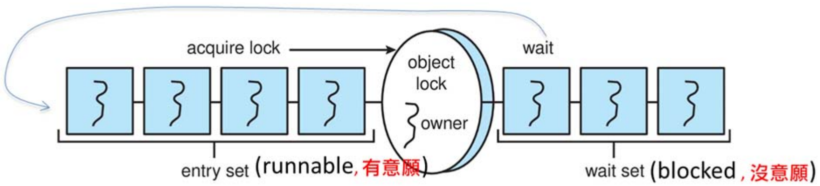
Bounded Buffer – Java Synchronization

```
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 */ P.306
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */ P.306
   }
}
```

Bounded Buffer – Java Synchronization

整個object是atomic，一次只有一個thread會進method
不能預知，下一個執行者是Consumer或Producer

```
/* Producers call this method */
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE) { 已滿
      try {
         wait();   先在wait set等; 讓別人先用
      }
      catch (InterruptedException ie) { }
   }

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

   notify();  用完後隨機從wait set取一個T到entry set
}
```




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

   while (count == 0) { 已空
      try {
         wait();  先在wait set等; 讓別人先用
      }
      catch (InterruptedException ie) { }
   }

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

   notify();  用完後隨機從wait set取一個T到entry set

   return item;
}
```

Condition Variables

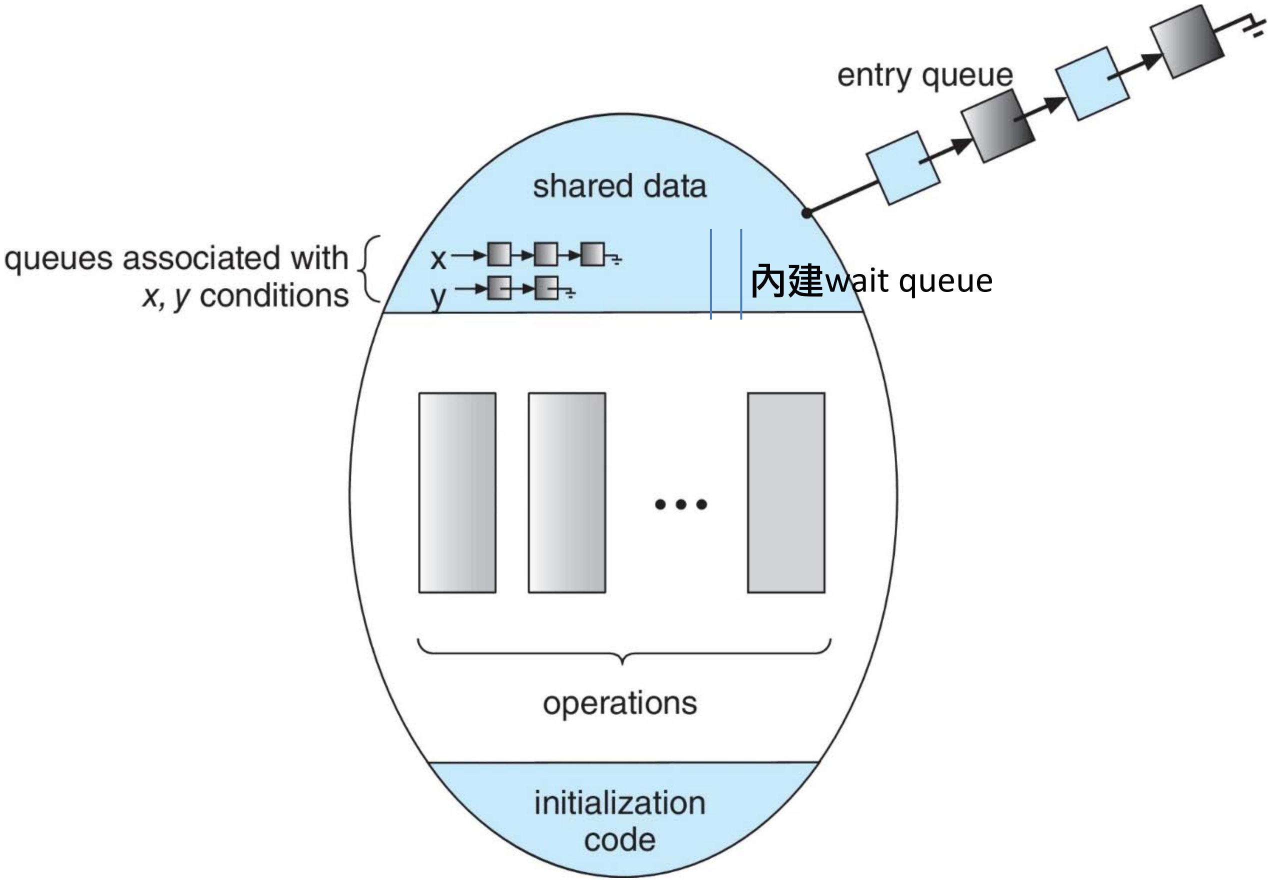
- condition x;
  - 內部維護一個Queue
  - 二個動作: wait→等待被叫; signal→依規則從Queue中選一個叫
- Details
  - x.wait() – the process invoking this operation is suspended (in the CV queue) until another process invokes x.signal()
  - x.signal() – resumes exactly one suspended process
    - If no process is suspended (CV queue沒有人), then signal() does nothing

Condition Variables (CV)

- Monitor lock the entire object sometimes not efficient
  - Use condition variables for more sophisticated synchronization
    - Many different conditions to wait for
    - 可以做到自由控制那個thread can enter
    - 等同於在Monitor中做多個wait queues
- CV represent some condition that a thread can:
  - Wait until the condition x occurs; or
  - Notify other waiting threads that the condition has occurred

```
condition   x;
…
x.wait(); // the calling process suspend itself
…
x.signal(); // resume exactly one process
```

Monitor with Condition Variables



Condition Variables Choices

- If thread P invokes **`x.signal()`**, and thread Q is suspended in **`x.wait()`**, what should happen next?
  - 針對CV x，P完成針對x的工作且呼叫了x.signal
  - P與Q在同一個Monitor:
    - CV x在Monitor中→Monitor中同時只能1個thread active→P還想繼續在Monitor中其它地方活動→P和Q只能有一個繼續
    - 馬上要叫Q嗎? 還是等P在Monitor中辦完其它事離開後再叫Q?
- Options

課本採用Hoare

  - **(P) Signal and wait – Hoare P等Q**
    - P waits until Q either leaves the monitor or it waits for another condition
  - **(P) Signal and continue – Mesa Q等P**
    - Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide

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

- 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
  - 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:

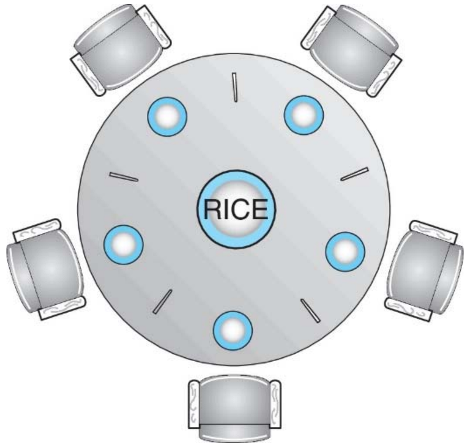
```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

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

```java
/* 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();
  }
}
```

指定下一個thread

Dining-Philosophers Problem

- 5 persons sitting on 5 chairs with 5 chopsticks
- A person is either thinking or eating
  - thinking: no interaction with the rest 4 persons
  - eating: need 2 chopsticks at hand
  - a person picks up 1 chopstick at a time
  - done eating: put down both chopsticks
- Problems implied
  - Resource contention (mutual exclusion)
  - deadlock problem (progress)
  - starvation problem (bounded waiting)

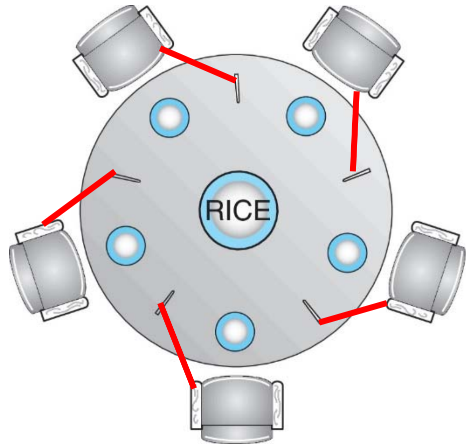


錯誤解法 (Deadlock)

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

等待並拿左筷

等待並拿右筷



- Possible solutions

  四個人用五支筷子

  1. Allow at most four philosophers to be sitting simultaneously at the table.
  2. Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  3. Asymmetric—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick

Monitor Solution to Dining Philosophers

```
monitor dp {
  enum {thinking, hungry, eating} state[5]; //哲學家的狀態
  condition self[5]; //delay eating if can’t obtain chopsticks
                     // 代表五個哲學家
  void pickup(int i)         // pickup chopsticks
  void putdown(int i)        // putdown chopsticks
  void test(int i)           // try to eat
  void init() {
       for (int i = 0; i < 5; i++)
                state[i] = thinking;
  }
}
```

使用condition variable的原因:
必須指定「特定process」被wait or 被signal

Monitor Solution to Dining Philosophers

Allow a philosopher to pick up her chopsticks only if both chopsticks are available

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait(); Test失敗 (剛好隔壁有人在吃)，就要等
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);      叫右方吃; 叫左方吃 (各一次機會)
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&                如果左右方人都沒在吃，而且自己有意願吃
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal(); →開始吃
      }
   }

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

DiningPhilosophers.pickup(i);

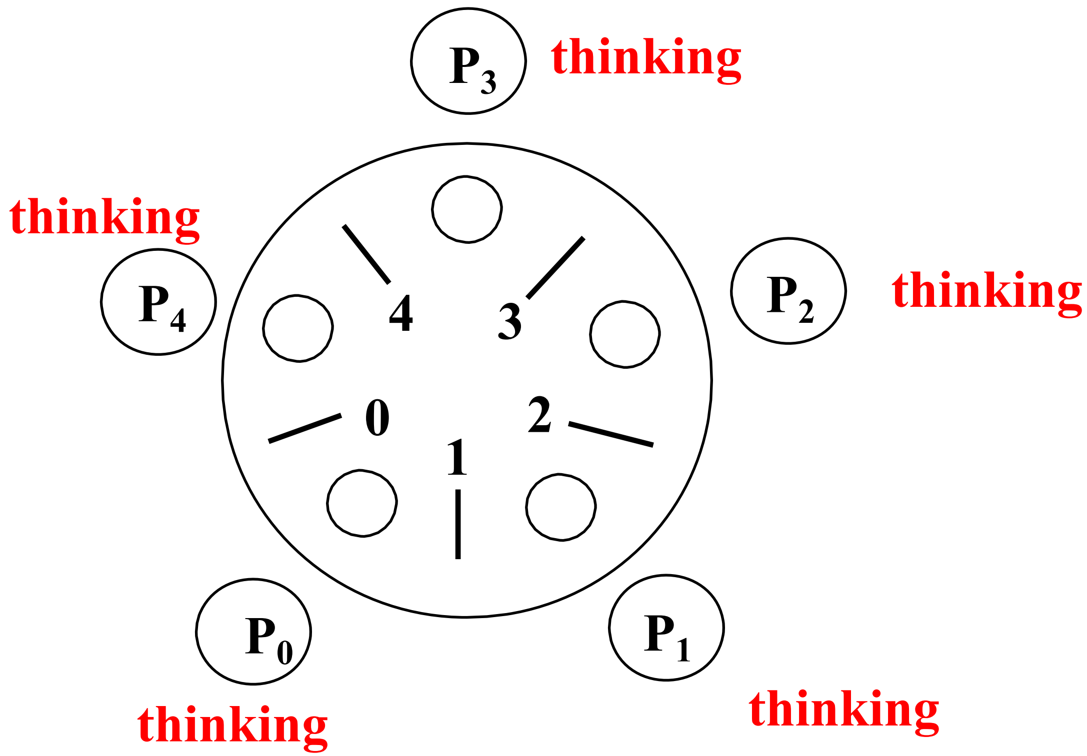
...

eat

...

DiningPhilosophers.putdown(i);

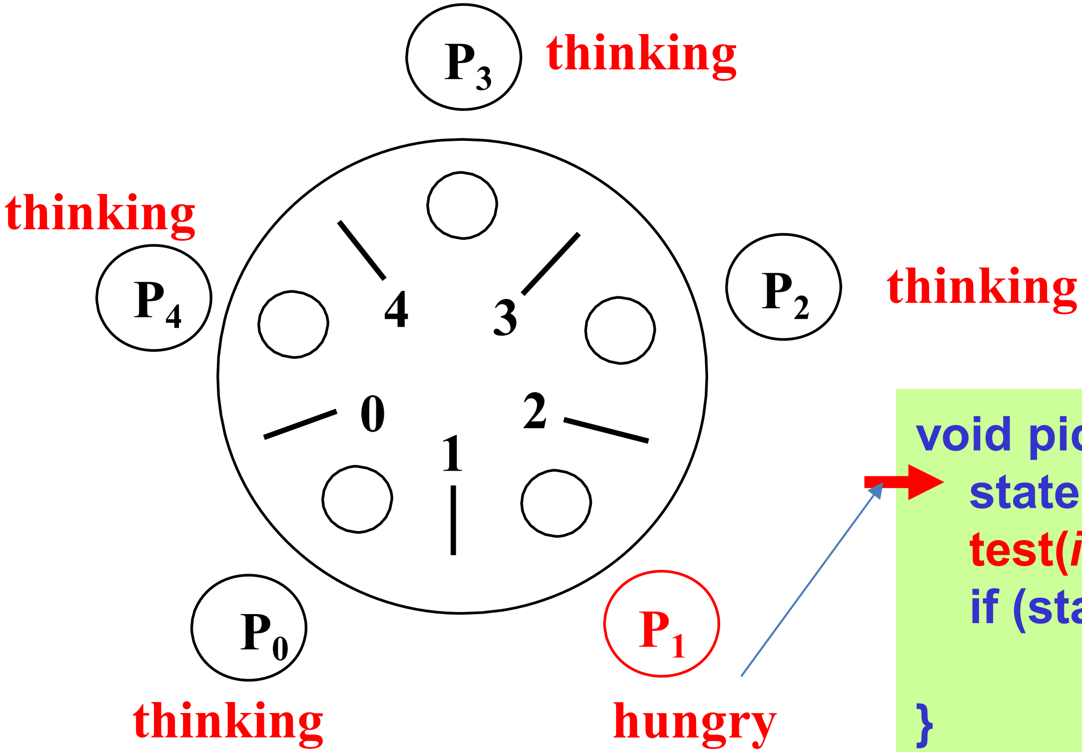
An illustration




```
P1:
DiningPhilosophers.pickup(1)
          eat
DiningPhilosophers.putdown(1)
```

```
P2:
DiningPhilosophers.pickup(2)
          eat
DiningPhilosophers.putdown(2)
```

An illustration

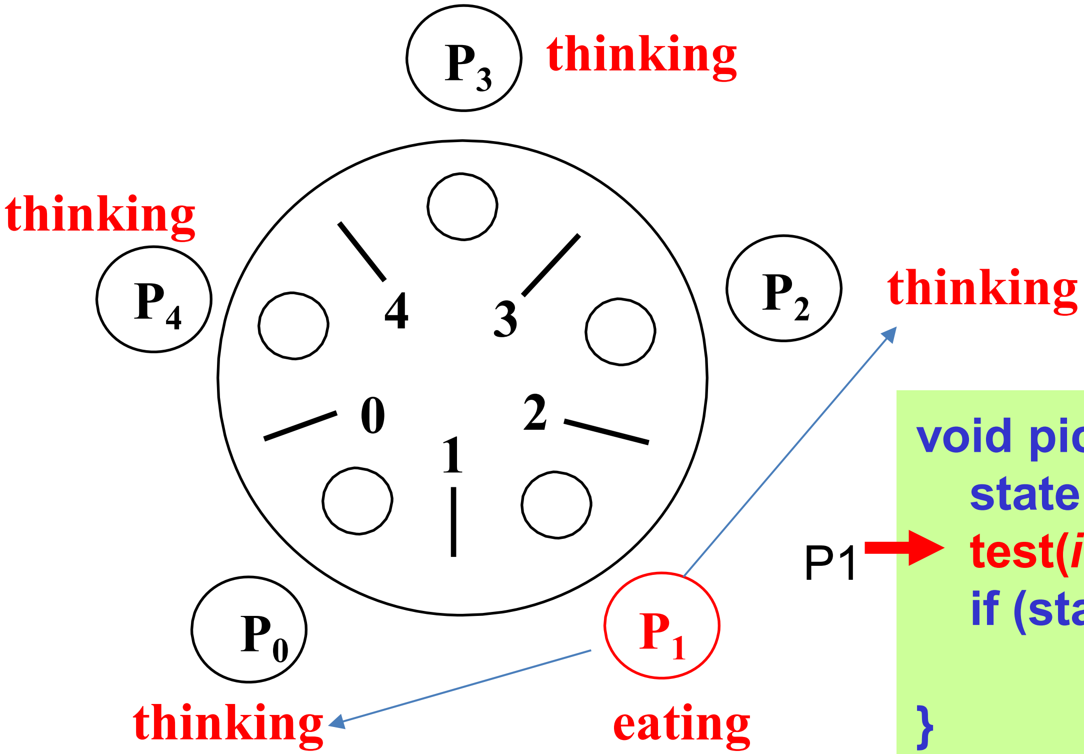



```
void pickup(int i) {
    state[i] = hungry;
    test(i); //try to eat
    if (state[i] != eating)
            self[i].wait();//wait to eat
}
```

```
P1:
DiningPhilosophers.pickup(1)
        eat
DiningPhilosophers.putdown(1)
```

```
P2:
DiningPhilosophers.pickup(2)
        eat
DiningPhilosophers.putdown(2)
```

An illustration




```
void test(int i) {
  if ((state[(i + 4) % 5] != EATING) &&
   (state[i] == HUNGRY) &&
   (state[(i + 1) % 5] != EATING)) {
     state[i] = EATING;
     self[i].signal();
  }
}
```

```
void pickup(int i) {
   state[i] = hungry;
P1 → test(i); //try to eat
   if (state[i] != eating)
         self[i].wait();//wait to eat
}
```

P1:
→ DiningPhilosophers.pickup(1)
eat
DiningPhilosophers.putdown(1)

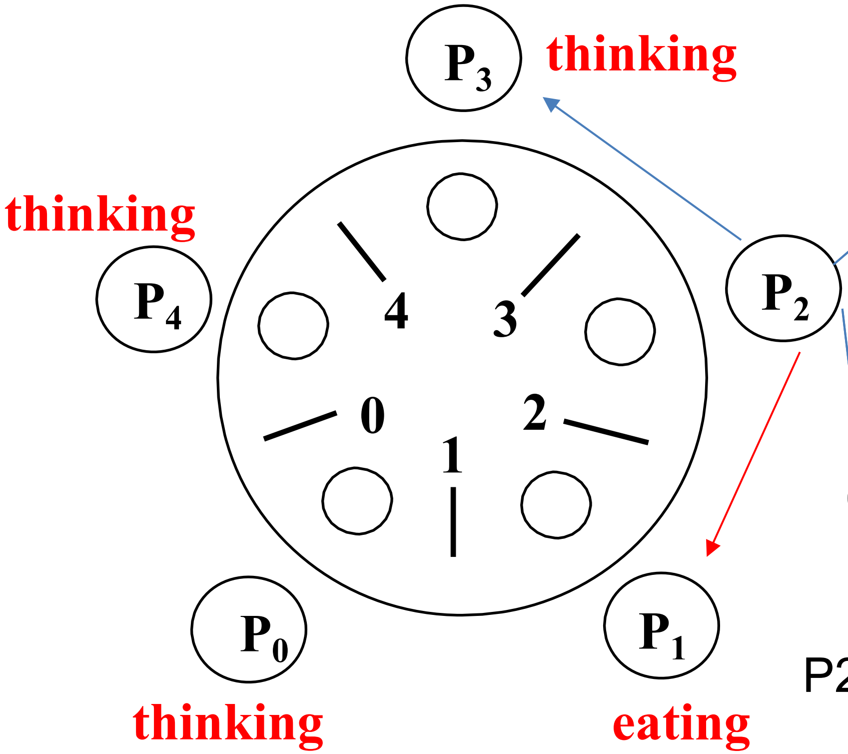
P2:
DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration

```
void test(int i) {
  if ((state[(i + 4) % 5] != EATING) &&
   (state[i] == HUNGRY) &&
   (state[(i + 1) % 5] != EATING)) {
     state[i] = EATING;
     self[i].signal();
  }
}
```

不符合if

(2)



hungry → self[2].wait

(1)

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

P2 → (3)

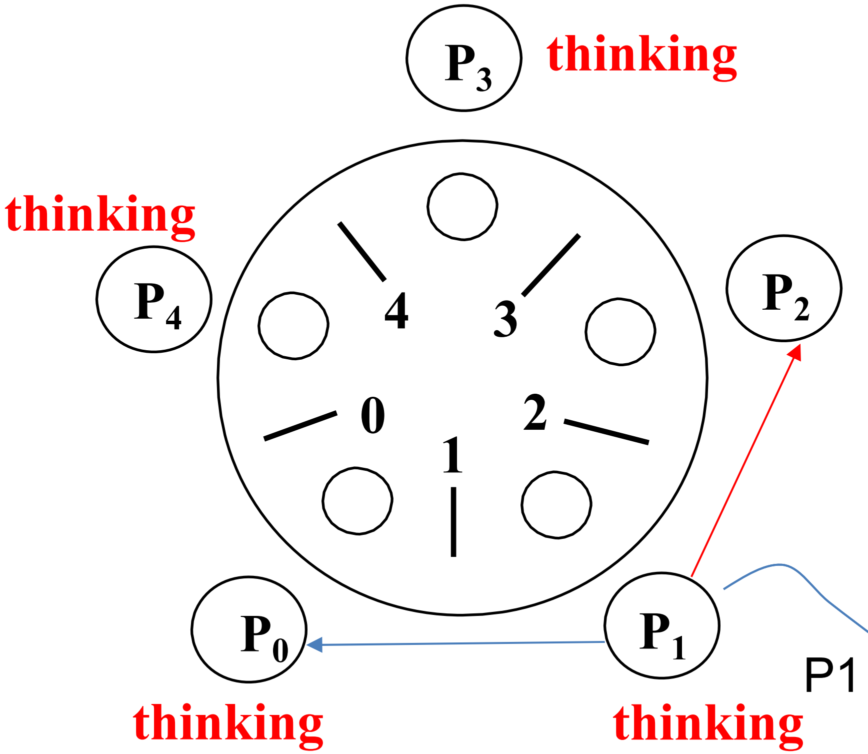
P1:
DiningPhilosophers.pickup(1)
→ eat
DiningPhilosophers.putdown(1)

P2:
→ DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration

```
void test(int i) {
  if ((state[(i + 4) % 5] != EATING) &&
   (state[i] == HUNGRY) &&
   (state[(i + 1) % 5] != EATING)) {
     state[i] = EATING;
     self[i].signal();
  }
}
```

符合if



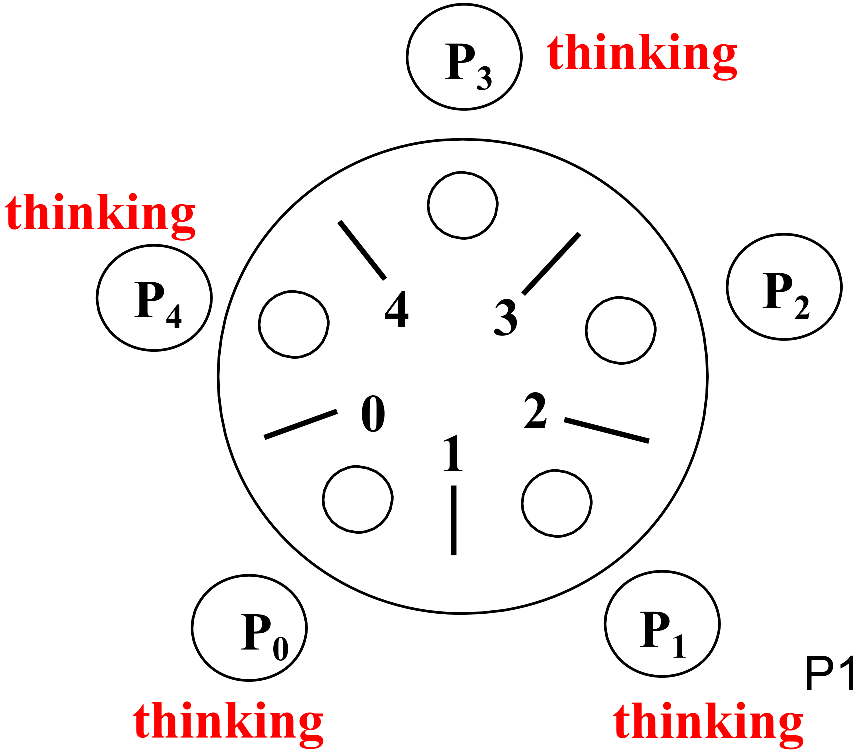
hungry ➔ self[2].wait

```
void putdown(int i) {
   state[i] = thinking;
   // check if neighbors
   // are waiting to eat
   test((i+4) % 5); test((i+1) % 5);
}
```

P1:
DiningPhilosophers.pickup(1)
eat
DiningPhilosophers.putdown(1)

P2:
DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration



eating ➔ self[2].signal

P1 ➔

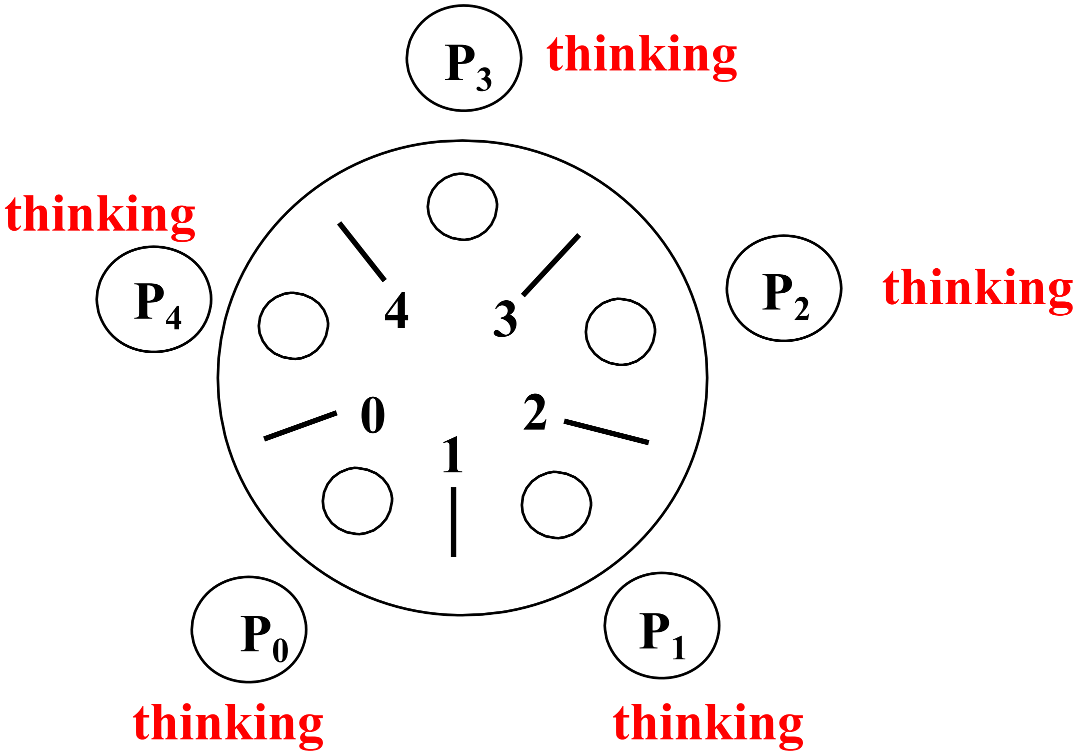
```
void putdown(int i) {
    state[i] = thinking;
    // check if neighbors
    // are waiting to eat
    test((i+4) % 5); test((i+1) % 5);
}
```

P1:
DiningPhilosophers.pickup(1)
eat
➔ DiningPhilosophers.putdown(1)

➔

P2:
DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration

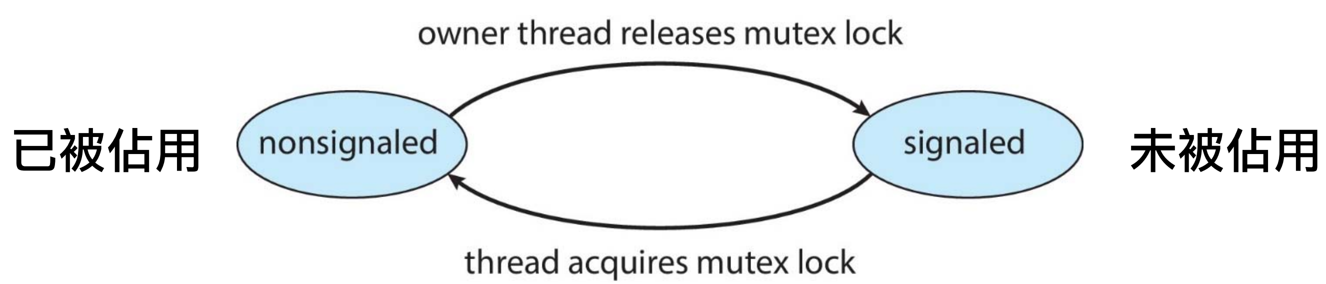



```
P1:
DiningPhilosophers.pickup(1)
          eat
DiningPhilosophers.putdown(1)
```

```
P2:
DiningPhilosophers.pickup(2)
          eat
DiningPhilosophers.putdown(2)
```

Synchronization - Windows

- Global resource protection In kernel
  - Uniprocessor: enable/disable interrupt
  - Multi-processor: spinlock
- Dispatcher objects Outside kernel
  - Implemented using semaphore, event, and timer
  - Event: notify a waiting thread when a condition occurs



P.298

Linux Synchronization

- Kernel
  - Version 2.6 and later, fully preemptive
  - Uniprocessor:
    - enable/disable interrupt 使系統暫時變成non-preemptive
  - Multi-Processor:
    - Short period task: spinlock (with preempt_count)
    - Long period task: hw supported semaphore or mutex lock

P.299 什麼是preempt_count, Linux中如何使用它來決定現在kernel是否適合preemption?

Liveness

- Processes may have to wait indefinitely while trying to acquire a lock
- Waiting indefinitely violates the progress and bounded-waiting
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress

Liveness

- 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                 P1
wait(S);          wait(Q);   // S==0; Q==0
wait(Q);          wait(S);
     Circular wait  .
    .                  .
    .                  .
signal(S);        signal(Q);
signal(Q);        signal(S);
```

  - Consider if P0 executes wait(S) and P1 wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q)
  - However, P1 is waiting until P0 execute signal(S).
  - Since these signal() operations will never be executed, P0 and P1 are deadlocked.

課後閱讀

- P.271 Lock contention
- P.299 什麼是preempt_count, Linux中如何使用它來決定現在kernel是否適合preemption?
- P.296 Critical-section object 為何效能會好: kernel mutex is allocated on demand
- P.313 Sect. 7.5.3 第6、7章探討的問題，為何在Functional Programming中不存在?

Q & A