Chapter 5　同步(Synchronization)

林政宏
國立臺灣師範大學電機工程學系

CHAPTER 5　同步

- 6.1 背景
- 6.2 臨界區間問題
- 6.3 Peterson’s解決方案
- 6.4 同步之硬體
- 6.5 號誌
- 6.6 典型的同步問題
- 6.7 監督程式
- 6.8 同步範例
- 6.9 不可分割的交易

Module 5: Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Background

- 多個程序同時對共享資料(shared data)進行存取可能會造成資料不一致(data inconsistency)
- 為保持資料的一致性, 必須確保合作程序的執行順序
- 假設生產者/消費者問題中, 共用一個緩衝(buffer)。透過一個變數count來記錄這個緩衝的使用數量。其中count初始化為0。當生產者生產一個資料時, count加一, 相反的當消費者消費一個資料時, count減一。

Producer

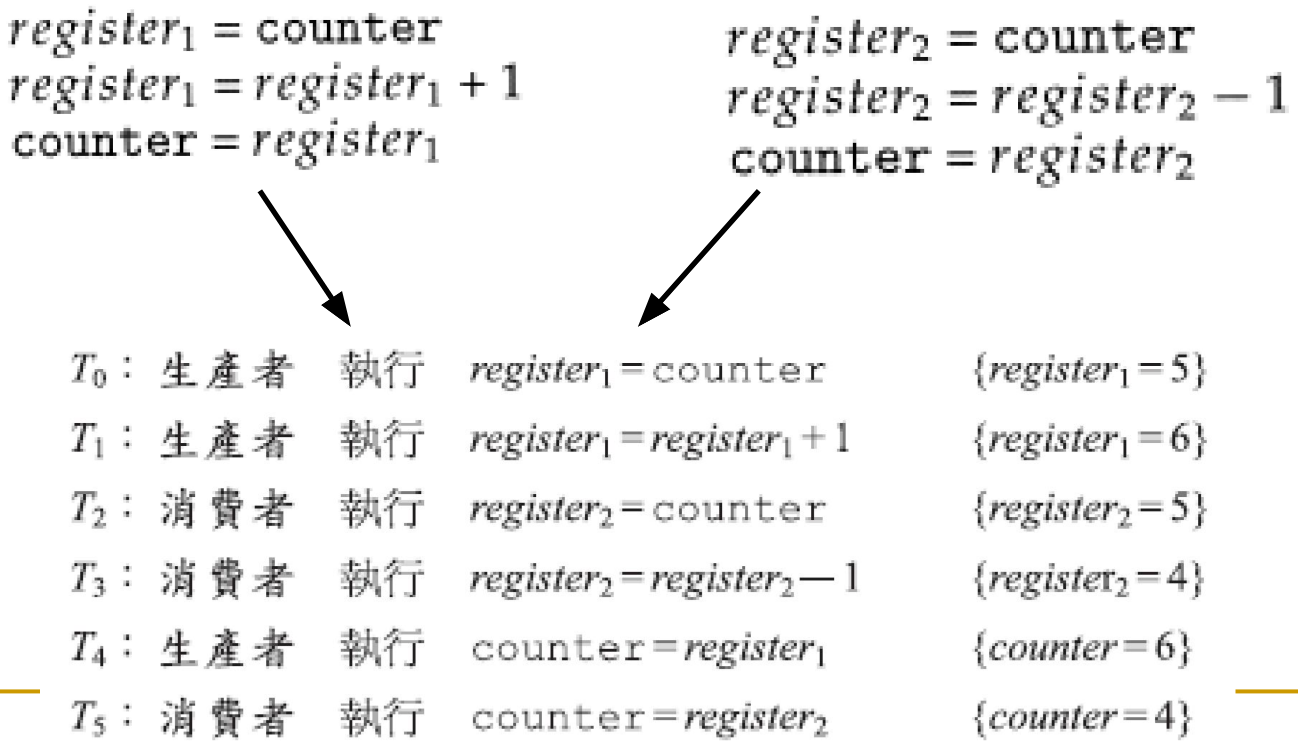
```
while (true) {
        /*  produce an item and put in nextProduced  */
        while (count == BUFFER_SIZE); // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Consumer

```
while (true)  {
        while (count == 0); // do nothing
  nextConsumed =  buffer[out];
  out = (out + 1) % BUFFER_SIZE;
        count--;

 /*  consume the item in nextConsumed
}
```

Race condition

- 這個不正確的狀態是因為允許兩個行程並行處理這個 counter變數。
- 這種數個行程同時存取和處理相同資料的情況，而且執行的結果取決於存取時的特殊順序，就叫競爭情況 (race condition)。



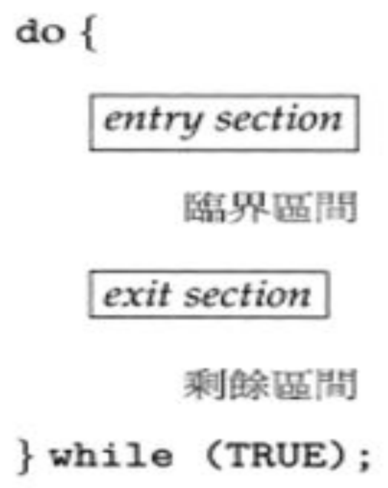
$register_1$ = counter
$register_1 = register_1 + 1$
counter = $register_1$

$register_2$ = counter
$register_2 = register_2 - 1$
counter = $register_2$

T_0：	生產者	執行	$register_1$ = counter	$\{register_1 = 5\}$
T_1：	生產者	執行	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2：	消費者	執行	$register_2$ = counter	$\{register_2 = 5\}$
T_3：	消費者	執行	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4：	生產者	執行	counter = $register_1$	$\{counter = 6\}$
T_5：	消費者	執行	counter = $register_2$	$\{counter = 4\}$

Solution to critical section problem

- 解決critical section problem的方案必須滿足下面三個要求
  - 互斥(mutual exclusion):如果行程Pi正在臨界區(critical section)間內執行，則其它的行程不能在其臨界區間內執行。
  - 進行(Progress):如果沒有行程在臨界區間內執行，同時某一行程想要進入臨界區間，那麼只有那些不在剩餘區間執行的行程才能加入決定誰將在下一次進入臨界區間，並且這個選擇不得無限期地延遲下去。
  - 限制住的等待(bounded waiting):在一個行程已經要求進入臨界區間，而此要求尚未被答應之前，允許其它的行程進入臨界區間的次數有一個限制。



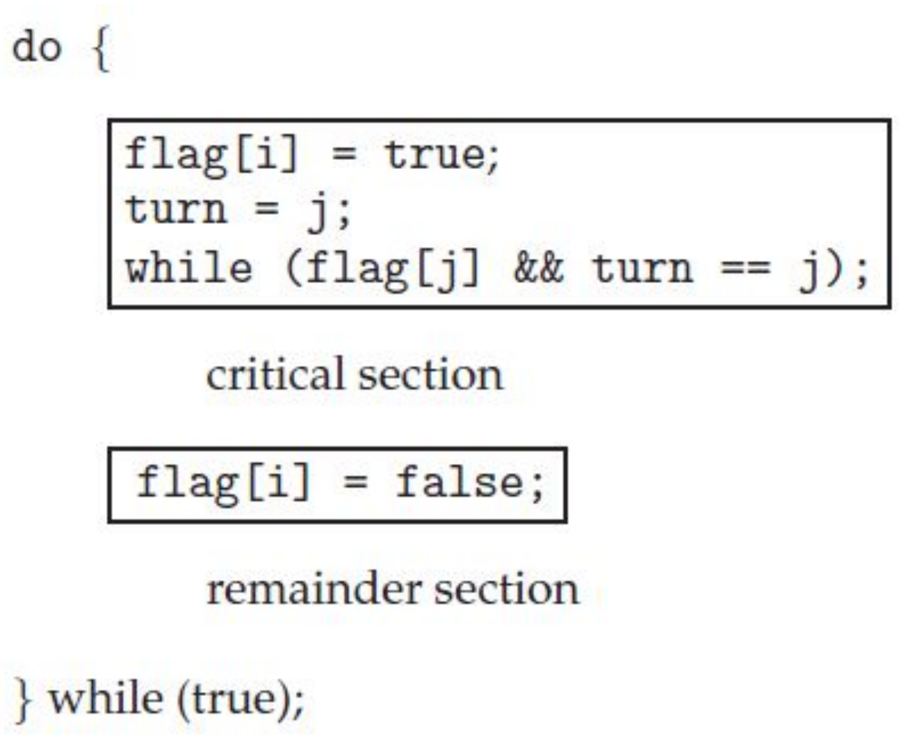
```
do {
        entry section
            臨界區間
        exit section
            剩餘區間
} while (TRUE);
```

圖 6.1 典型行程 P_i 的一般結構圖

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

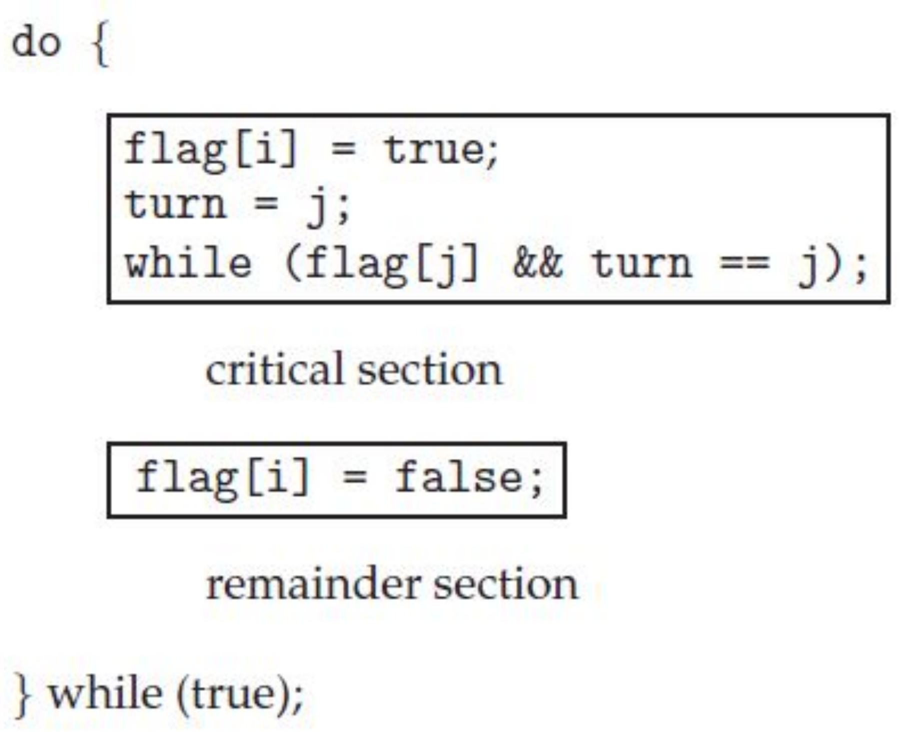
- 要證明
1. 互斥性(Mutual exclusion)存在，
2. 進行(progress)的要求能被滿足，
3. 限制性的等待(bounded-waiting)要求亦能符合。



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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

–



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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

Synchronization Hardware

- Many systems provide hardware support for critical section code

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable

TestAndndSet Instruction

- Definition:

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

Solution using TestAndSet

- Shared boolean variable lock., initialized to **false**.
- Solution:

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

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

compare_and_swap Instruction

- Definition:

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

Solution using compare_and_swap

- Shared Boolean variable lock initialized to **false**; Each process has a local Boolean variable key
- Solution:

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


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

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
                key = test_and_set(&lock);
        waiting[i] = false;
        /* critical section */
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = false;
        else
                waiting[j] = false;
        /* remainder section */
} while (true);
```

Shared Boolean variable **lock** initialized to **false**;
Each process has a local Boolean variable **key**

滿足下面三個要求

- 互斥(mutual exclusion):如果行程Pi正在臨界區(critical section)間內執行, 則其它的行程不能在其臨界區間內執行。
- 進行(Progress):如果沒有行程在臨界區間內執行, 同時某一行程想要進入臨界區間, 那麼只有那些不在剩餘區間執行的行程才能加入決定誰將在下一次進入臨界區間, 並且這個選擇不得無限期地延遲下去。
- 限制住的等待(bounded waiting):在一個行程已經要求進入臨界區間, 而此要求尚未被答應之前, 允許其它的行程進入臨界區間的次數有一個限制。

Mutex Locks

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

```
acquire(){
      while(!available);//busy wait
      available = false;
}
```

```
release(){
      available = true;
}
```

Mutex Locks

- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- 要進入critical section先透過acquire() 取得 lock 之後再透過release() 釋放lock
  - 利用布林變數指示lock 可用還是不可用

- **acquire()** and **release()** 必須是原子化(atomic)
  - 通常用hardware atomic instructions實現

- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations

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

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

Semaphore

- 計數號誌 (counting semaphore)的值可以不受限制。
- 二元號誌 (binary semaphore)的數值可以是0或 1。
- 在某些系統，二元號誌稱為互斥鎖 (mutex locks)，當它們是鎖而且互斥。
- synch is initialized to 0.
- Consider P_1 and P_2 that require S_1 to happen before S_2

P1:

S_1;
signal(synch);

P2:

wait(synch);
S_2;

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

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

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

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

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
        wait(semaphore *S) {
         S->value--;
         if (S->value < 0) {
               add this process to S->list;
               block();
         }
 }
```

- Implementation of signal:

```
  signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) { //means some or one process waiting in queue
              remove a process P from S->list;
              wakeup(P);
        }
  }
```

Wait Operation (P)

- When a process wants to enter a critical section or use a resource, it performs a wait operation on the corresponding semaphore.
- If the value of the semaphore is greater than zero, it means that the resource is available or the entry to the critical section is permitted. The semaphore is then decremented by one and the process proceeds.
- If the semaphore value is zero, this indicates that no resources are available or entry is not allowed. The process is then blocked and added to the semaphore's queue, rather than engaging in busy waiting.

Signal Operation (V)

- When a process leaves a critical section or releases a resource, it performs a signal operation.
- This increments the semaphore's value. If there are processes waiting in the semaphore's queue (those that were blocked during their wait operation), one of them gets unblocked, and it can proceed with its operation.
- This process is then removed from the queue

Advantage of Semaphore

- **Blocking instead of spinning**: When no resources are available, processes are put into a blocked state instead of continuously checking the semaphore's value. This blocking is handled by the operating system, which schedules other tasks that can proceed. This way, CPU resources are used more efficiently and are not wasted on unproductive checks.
- **System-Managed Queues**: The operating system manages the queue of processes waiting for the semaphore. This ensures that processes are resumed in an orderly fashion and that the system remains efficient.
- Semaphores avoid busy waiting through blocking processes when resources are not available and resuming them when they become available, instead of letting processes consume CPU cycles by continuously polling the semaphore's status.

- 6.5.3 死結和餓死(Deadlock and Starvation)
  - 使用waiting queue製做signal可能導致有二個或以上的行程等待一項僅能由一個等待的行程所引發事件的情形。此處所謂的事件是指一個 signal()運算的執行;而當上述情形發生時，我們稱這些行程產生了死結 (deadlocked)。
- Let ***S*** and ***Q*** be two semaphores initialized to 1

```
         P0                  P1
     wait(S); ①        wait(Q);②
     wait(Q); ③        wait(S);④
 .        .
     signal(S);             signal(Q);
     signal(Q);        signal(S);
```

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

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

- **餓死(Starvation) – infinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended

6.5.4 優先權倒置 (Priority Inversion)

- 假設有三個行程L、M、H，其優先權順序是L < M < H。假設行程H要求正被行程L存取的資源R。通常行程H會等待L完成使用資源R。然而，現在因為行程M變成可執行的，因此可搶先行程L。間接地一個有較低優先權的行程(行程M)已經影響到行程H。**這個問題被稱為優先權倒置。**

- 優先權繼承(priority-inheritance)協定必須允許行程L暫時地繼承行程H的優先權，因此才能阻止行程M搶先它的執行。當行程L已經完成使用資源R時，它將放棄它從H所繼承的優先權並假設其最初的優先權。因為資源R是可用的，接下來讓行程H使用，而不是M。

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value *n*

Bounded Buffer Problem (Cont.)

- The structure of the **producer** process

```
int n;
semaphore mutex = 1 ;
semaphore empty = n;
semaphore full = 0;
```

```
do{
      //   produce an item in next_produced
      wait (empty);
      wait (mutex);

      //  add the item to the  buffer

      signal (mutex);
      signal (full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the **consumer** process

```
do{
          wait (full);
          wait (mutex);

               //  remove an item from  buffer to next_consumed

          signal (mutex);
          signal (empty);

               //  consume the item in next_consumed

} while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write

- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time

- Shared Data
  - Data set
  - semaphore **rw_mutex** = 1;
  - semaphore **mutex** = 1;
  - int **read_count** = 0;

6.6.2 讀取者-寫入者問題

- The structure of a writer process

```
semaphore rw_mutux = 1;
semaphore mutux = 1;
int read_count = 0;
do {
        wait(rw_mutex);
      ...
      /* writing is performed */
      ...
      signal(rw_mutex);
} while (true);
```

- The structure of a reader process

```
do {
      wait(mutex);
      read_count++;
      if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);

        /* reading is performed */

   wait(mutex);
      read_count--;
      if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

除非所有的reader完成工作，writer無法取得lock(rw_mutex)取寫入資料

- 6.6.3 哲學家進餐的問題

有五個哲學家圍繞坐在一個圓桌, 每個哲學家面前有一個碗, 但只有五支筷子, 肚子餓的哲學家必須拿起左右的筷子才能吃飯

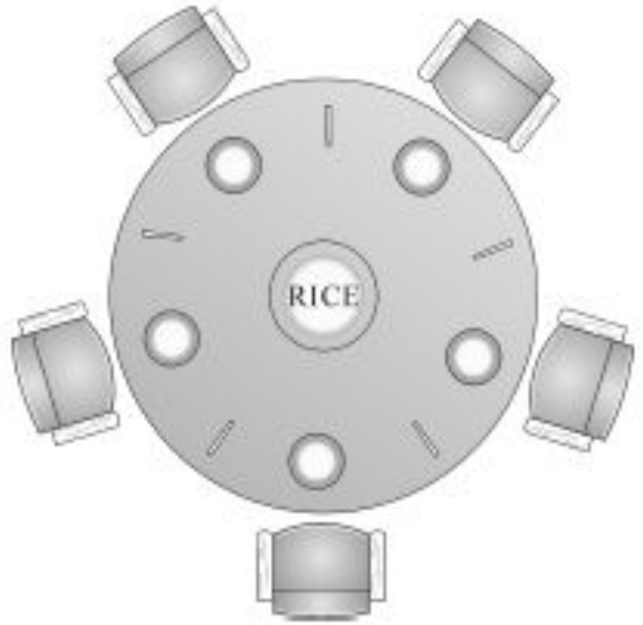


圖 6.14 哲學家吃飯

- The structure of Philosopher *i*:

```
do  {
        wait ( chopstick[i] );
        wait ( chopstick[ (i + 1) % 5] );

      //  eat for awhile

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

         //  think for awhile

} while (true);
```

Dining-Philosophers Problem (Cont.)

- 五個哲學家同時巴度么
- 每個哲學家同時拿起左邊的筷子，然後大家都在等右邊的筷子，結果大家都餓死了！
  - Deadlock
- Solution
  - 允許最多四個哲學家可以同時上桌。
  - 允許一位哲學家吃飯，當他左右兩邊的筷子都可用時。
  - 非對稱解法，當奇數位的哲學家先拿起他左邊的筷子，再拿起他右邊的筷子，而偶數位的哲學家先拿起他右邊的筷子，再拿起他左邊的筷子。

任何解法都必須思考有沒有哪位哲學家會因為拿不到筷子而餓死!
一個免於deadlock的解法不見得會沒有餓死的機率!

另一種哲學家吃飯的問題之一

- 考慮哲學家就餐問題的版本，其中筷子放在桌子的中央，並且**任何兩個筷子都可以被哲學家使用**。假設對筷子的要求是一次一根的。
- 描述一個簡單的規則，用於決定是否可以滿足特定請求而不會導致僵局(deadlock)。

另一種哲學家吃飯的問題之一

- 考慮哲學家就餐問題的版本，其中筷子放在桌子的中央，並且**任何兩個筷子都可以被哲學家使用**。假設對筷子的要求是一次一根的。描述一個簡單的規則，用於決定是否可以滿足特定請求而不會導致僵局(deadlock)。

- ANS:當有一個哲學家提出要第一根筷子的請求時，以下情況不可以同意請求:如果沒有任何其他哲學家拿著兩根筷子而且只剩下一根筷子。(表示每個哲學家都只拿到一根筷子)

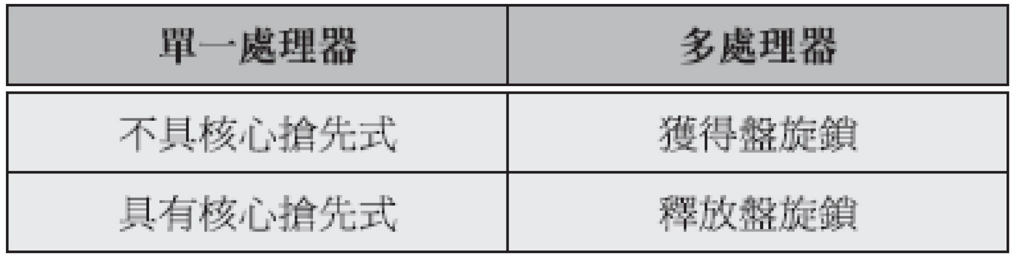
另一種哲學家吃飯的問題之二

- 再次考慮上一個問題中的設置, 現在假設每個哲學家需要三支筷子才能吃完。筷子的請求仍然一次一根。描述一些簡單的規則, 用於決定是否可以滿足特定請求而不會導致僵局(deadlock)。

- ANS:當哲學家提出要筷子的請求時, 在以下情況下可以同意請求:1) 哲學家有兩根筷子, 並且桌面上至少剩餘一根筷子, 2) 哲學家有一根筷子, 並且桌面上至少剩餘兩根筷子, 3) 桌面上至少剩下一根筷子, 並且至少有一位哲學家已經有三根筷子, 4) 哲學家沒有筷子, 桌面上還剩下兩根筷子, 並且至少有另外一位哲學家分配了兩根筷子。

6.8.3 Linux的同步

- Linux在2. 6版之前是一個不可搶先的核心，也就是行程在核心模式執行是不可搶先，甚至如果一個較高優先的權行程可以執行時。然而，現在 Linux完全可以搶先，所以當任務正在核心執行時是可以搶先的。



單一處理器	多處理器
不具核心搶先式	獲得盤旋鎖
具有核心搶先式	釋放盤旋鎖

6.8.4 Pthreads的同步

- Threads API在執行緒同步提供mutex locks、condition variables和read-write locks。這一個API對電腦程式設計師是可用的以及不是任何特別核心的一部份。mutex locks呈現使用Pthreads的基本同步技術。mutex locks用來保護臨界區間的程式碼,也就是執行緒在進入臨界區間前獲得鎖，然後在離開臨界區間將鎖釋放。

Pthreads Synchronization

```
#include <pthread.h>
pthread_mutex_t mutex;
/*create the mutex lock*/
pthread_mutex_init(&mutex, NULL);

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

/* critical section*/

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

Pthreads Synchronization

```
#include <semaphore.h>

sem_t sem;
/*create a semaphore and initialized to 1 */
sem_init(&sem, 0, 1);

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

/*critical section*/

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

Race condition

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *thread(void *param);
int count = 0;
#define N 10000
int main(int argc, char *argv[])
{
        int i;
        pthread_t tid[N];
        pthread_attr_t attr[N];

        //Create N threads
        for(i = 0; i < N; i++){
                pthread_attr_init(&attr[i]);
                pthread_create(&tid[i], &attr[i], thread, NULL);
        }
        for(i = 0; i < N; i++){
                pthread_join(tid[i], NULL);
        }
        printf("count is %d\n", count);
        return 0;
}
```

```
void *thread(void *param)
{
        count++;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread(void *param);
int count = 0;
pthread_mutex_t mutex;
#define N 10000
int main(int argc, char *argv[]){
	int i;
	pthread_t tid[N];
	pthread_attr_t attr[N];
	pthread_mutex_init(&mutex, NULL);

	//Create N threads
	for(i = 0; i < N; i++){
		pthread_attr_init(&attr[i]);
		pthread_create(&tid[i], &attr[i], thread, NULL);
	}
```

```
    for(i = 0; i < N; i++){
        pthread_join(tid[i], NULL);
    }
    printf("count is %d\n", count);

    return 0;
}

void *thread(void *param)
{
    //acquire the mutex lock
    pthread_mutex_lock(&mutex);
    count++;
    //release the mutex lock
    pthread_mutex_unlock(&mutex);

}
```