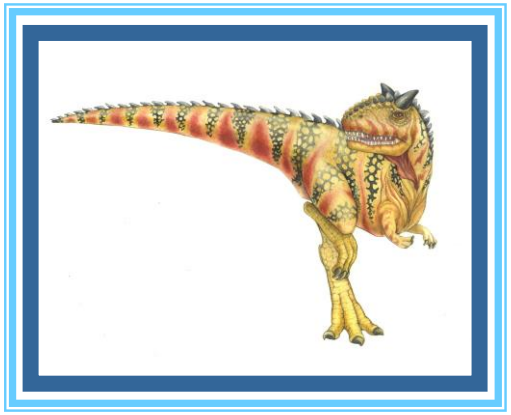
Chapter 7: Synchronization Examples



Chapter 7: Synchronization Examples

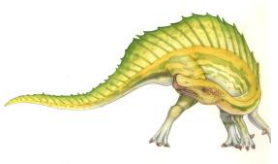
Classic Problems of Synchronization

Synchronization within the Kernel

POSIX Synchronization

Synchronization in Java

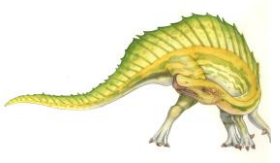
Alternative Approaches



Classical Problems of Synchronization

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem



Bounded-Buffer Problem

n buffers, each can hold one item

Semaphore (訊號) **`mutex`** initialized to the value 1

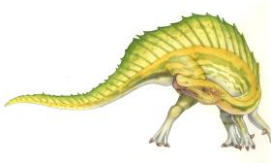
Semaphore **`full`** initialized to the value 0

Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

The structure of the producer process

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced 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

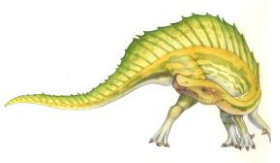
Problem – allow multiple readers to read at the same time

- Only one single writer can access the shared data at the same time

Several variations of how readers and writers are considered – all involve some form of priorities優先順序

Shared Data

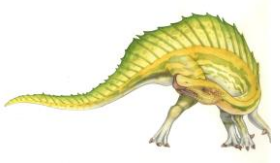
- Data set
- Semaphore **`rw_mutex`** initialized to 1 (讀寫)
- Semaphore **`mutex`** initialized to 1 (讀)
- Integer **`read_count`** initialized to 0



Readers-Writers Problem (Cont.)

The structure of a writer process

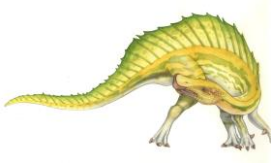
```
do {
        wait(rw_mutex);
           ...
        /* writing is performed */
           ...
     signal(rw_mutex);
} while (true);
```



Readers-Writers Problem (Cont.)

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);
```



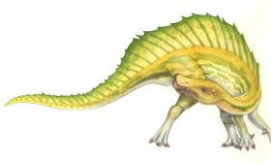
Readers-Writers Problem Variations

First variation變化– no reader kept waiting unless writer has permission to use shared object (除非有witer去使用共享object, 否則不能有reader在一直等待)

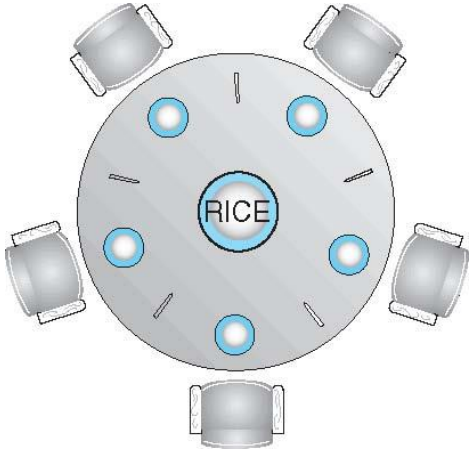
Second variation – once writer is ready, it performs the write ASAP (as soon as possible)

Both may have starvation飢餓 leading to even more variations(上述都可能產生飢餓情況)

Problem is solved on some systems by kernel providing (reader-writer locks: 解決上述的問題)



Dining-Philosophers Problem



Philosophers spend their lives alternating (一生中)thinking and eating

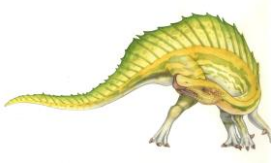
Don't interact with their neighbors, occasionally(偶爾) try to pick up 2 chopsticks (one at a time) to eat from bowl

Need both to eat, then release both when done

In the case of 5 philosophers

Shared data

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

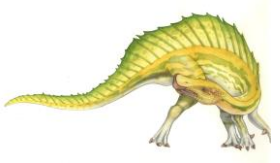
Deadlock的發生:一開始大家都拿右方的筷子並沒有問題，但當大家要拿取左方的筷子時，會因為已經被坐在自己左邊的人給取走了，所以就會形成一個「Deadlock」的problem狀態。

Deadlock的處理:

只允許在一個圓桌上有4個人而已。

要檢查左右是否都已經available，可以的話便能eating。

使用asymmetric方法解決:也就是說，坐在奇位數位置的人先拿取左邊再拿取右邊的筷子;坐在雙數位置的人先拿取右邊再拿取左邊的筷子，方能解決問題。



Dining-Philosophers Problem Algorithm

The structure of Philosopher *i*:

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

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

                // think
} while (TRUE);
```

What is the problem with this algorithm?

signal (mutex).....wait(mutex) / wait(mutex).....wait(mutex)

會遺漏wait(mutex)或是signal(mutex), 亦或是兩個一起遺漏掉。

Monitor Solution to Dining Philosophers

```
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;
}

   void putdown (int i) {
          state[i] = THINKING;
                  // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
}
```

Solution to Dining Philosophers (Cont.)

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

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

Solution to Dining Philosophers (Cont.)

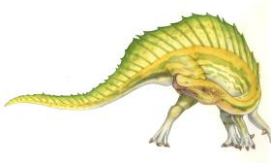
Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

    EAT

DiningPhilosophers.putdown(i);
```

每個philosopher i invoke兩個operations pickup()以及putdown()在sequence中。

因為一次一個進入, 所以並不會產生dealock, 但因為動作慢所以可能會產生starvation的情況。

No deadlock, but starvation is possible



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;
   }
}
```

Monitor:提供方便且有效率的機制給process synchronization使用。有些系統不支持。一次只能有一個process可以在monitor內執行。不能解決所有的synchronization問題，但這已經是較為複雜的方法

Synchronization Examples

Solaris

Windows

Linux

Pthreads

Solaris Synchronization

Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing

Uses **adaptive mutexes** for efficiency when protecting data from short code segments

- Starts as a standard semaphore spin-lock(標準的**semaphore spin-lock**)
- If lock held, and by a thread running on another CPU, spins**(保持lock，以及被thread執行在其他的CPU,spins)**
- If lock held by non-run-state thread, block and sleep waiting for signal of lock being released **(lock被不在執行狀態的thread給block和sleep，等待單一的lock被釋放)**

Uses **condition variables**

Uses **readers-writers** locks when longer sections of code need access to data (**當code需要longer section去存取data**)

Solaris Synchronization(2)

Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock (使用turnstiles去命令在互相等待的threads，將其整理成一個list，等待mutex或是reader-writer解決，並以資料結構的方式處理。)

Turnstiles are per-lock-holding-thread, not per-object

Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile (將priority-inheritance一個turnstile給正在執行thread最高優先全的threads在這個turnstile旋轉門)

Windows Synchronization

Uses interrupt masks to protect access to global resources on uniprocessor systems(保護global resources在uniprocessor systems存取)

Uses **spinlocks** on multiprocessor systems

Spinlocking-thread will never be preempted(永不會被中斷)

Also provides **dispatcher objects** user-land (使用物件分配器給使用者空間) which may act mutexes, semaphores, events, and timers

Events

- An event acts much like a condition variable(類似於condition variable)

Timers notify one or more thread when time expired (timer會通知一個或多個thread)

Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

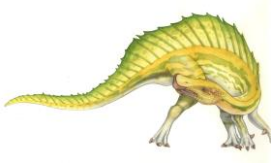
Linux:

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections (但回應速度較慢)
- Version 2.6 and later, fully preemptive (回應速度較佳)

Linux provides:

- Semaphores
- atomic integers
- spinlocks
- reader-writer versions of both

On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption (在單一CPU系統時，spinlock可被可用或不可用的kernel preemption所取代)



Pthreads Synchronization

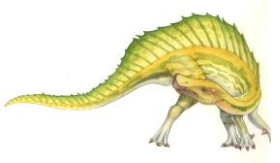
Pthreads API is OS-independent

It provides:

- mutex locks
- condition variable

Non-portable (可移植)extensions(擴增) include:

- read-write locks
- spinlocks



Alternative(其他) Approaches

Transactional Memory

OpenMP

Functional Programming Languages

Transactional Memory

A **memory transaction (交易)**is a sequence of read-write operations to memory that are performed atomically(不被中斷).

```
void update()
 {
        /* read/write memory */
  }
```

OpenMP

OpenMP is a set of compiler directives and API(compiler指令與API所組成) that support parallel progamming支持平行程式執行.

```
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(在程式碼中有#pragma omp critical 的指令，就像對待CS一樣是個atomic，不會中斷。).

Functional Programming Languages

Functional programming languages offer a different paradigm(範例) than procedural languages in that they do not maintain state(在不能維持的狀態).

Variables are treated as immutable(不變) and cannot change state once they have been assigned a value.

There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races(處理 data race condition).

End of Chapter 7