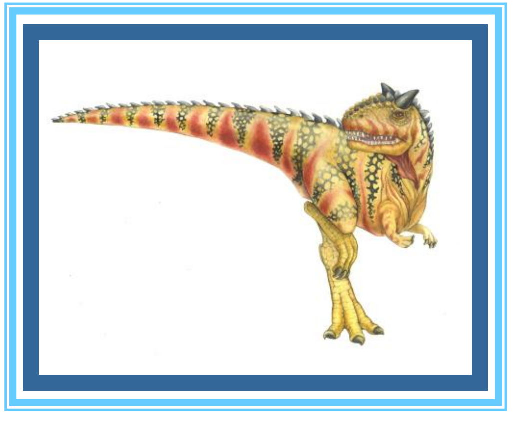
Chapter 6: Synchronization 同步Tools



Chapter 6: Synchronization同步Tools

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Monitors

Objectives

- To present the concept of process synchronization.
- 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 examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

Processes can execute concurrently

- May be interrupted at any time, partially completing execution

Concurrent access to shared data may result in data inconsistency

Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Illustration of the problem:
Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

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

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

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Race Condition

counter++ could be implemented as

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

counter-- could be implemented as

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

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}

中間被分離掉

https://ithelp.ithome.com.tw/articles/10225917

goroutine 是輕量級執行緒(lightweight thread)

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

General structure of process P_i

```
do {
    [entry section]
        critical section
    [exit section]
        remainder section
} while (true);
```

Algorithm for Process P_i

```
do {

    while (turn == j);

            critical section

    turn = j;

            remainder section

 } while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion(排除)**- If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely(推遲無限期)
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted(確認)
    - Assume that each process executes at a nonzero speed
    - No assumption concerning **relative speed** of the ***n*** processes

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily(自願) yields CPU
  - Essentially (基本) free of race conditions in kernel mode

Peterson's Solution

Good algorithmic description of solving the problem

Two process solution

Assume that the **`load`** and **`store`** machine-language 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!

Algorithm for Process P_i

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

Peterson's Solution (Cont.)

Provable可證明 that the three CS requirement are met:

1. Mutual exclusion is preserved

   P_i enters CS only if:

   either **`flag[j] = false`** or **`turn = i`**

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient (低效率) on multiprocessor systems
    - Operating systems using this not broadly寬廣 scalable
- Modern machines provide special atomic hardware instructions(指示)
    - **Atomic** = non-interruptible(原子同一時間只能有一個執行緒對共享資源進行讀寫操作)
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE".

Solution using test_and_set()

Shared Boolean variable lock, initialized to FALSE

Solution:

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

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;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Solution using compare_and_swap

Shared integer "lock" initialized to 0;

Solution:

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

Bounded-waiting Mutual Exclusion with test_and_set

```
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);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- 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()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock自旋鎖**

acquire() and release()

```
acquire() {
   while (!available)

      ; /* busy wait */

   available = false;;

}

release() {

   available = true;

}

do {

acquire lock

   critical section

release lock

  remainder section

} while (true);
```

Semaphore信號

- Synchronization tool that provides more sophisticated（複雜的）ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible不可分割(atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

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

- Definition of the **`signal() operation`**

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted無限制的 domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock (互斥)**
- Can solve various synchronization problems
- Consider ***P_1*** and ***P_2*** that require ***S_1*** to happen before ***S_2***

  Create a semaphore "**`synch`**" initialized to 0

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

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that **no two processes** can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied (critical section 狀況少發生)
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

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

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

Implementation with no Busy waiting (Cont.)

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

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

Deadlock 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

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation飢餓– **indefinite(無限)blocking**

- A process may never be removed from the semaphore queue in which it is suspended

Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

- Solved via **priority-inheritance(繼承) protocol**

Problems with Semaphores

Incorrect use of semaphore operations:

signal (mutex) …. wait (mutex)

wait (mutex) … wait (mutex)

Omitting (省略)of wait (mutex) or signal (mutex) (or both)

Deadlock and starvation are possible.

Monitors

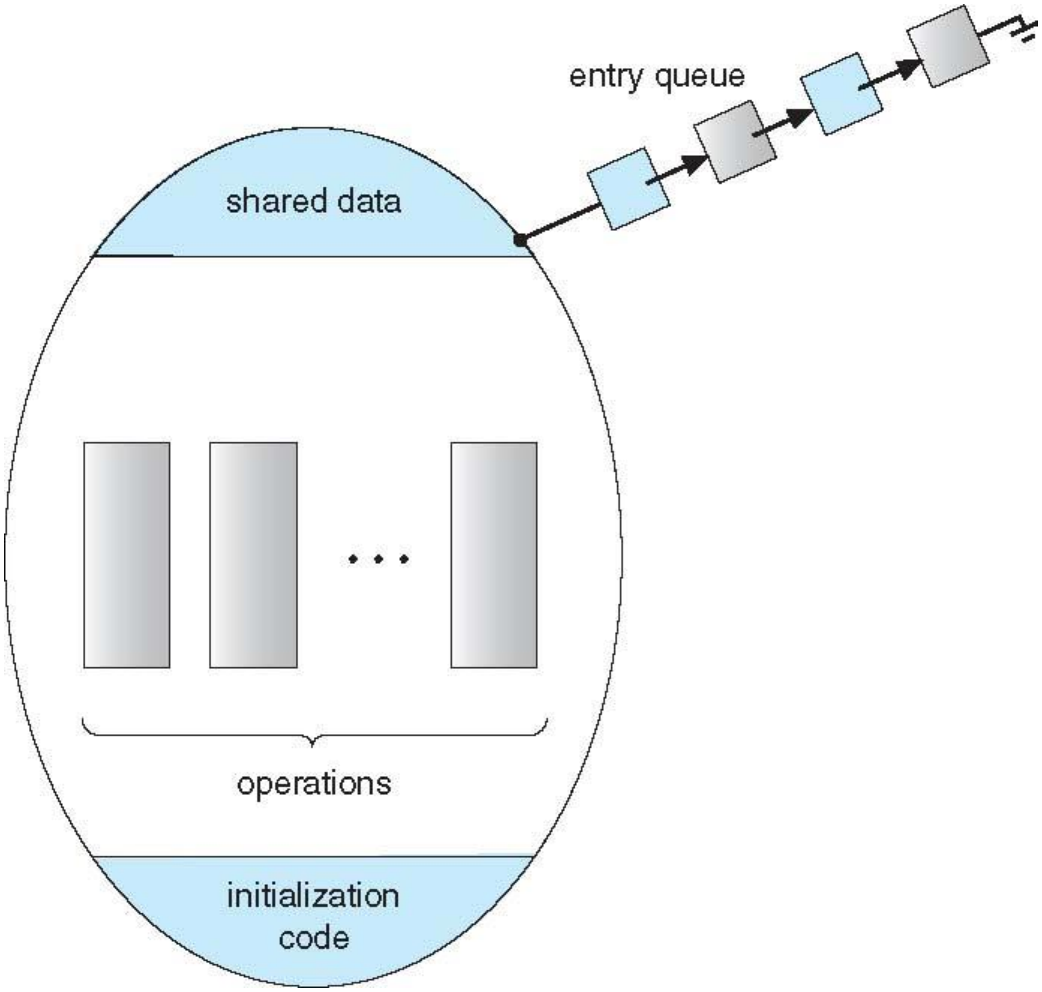
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables (内部)only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

Schematic view of a Monitor



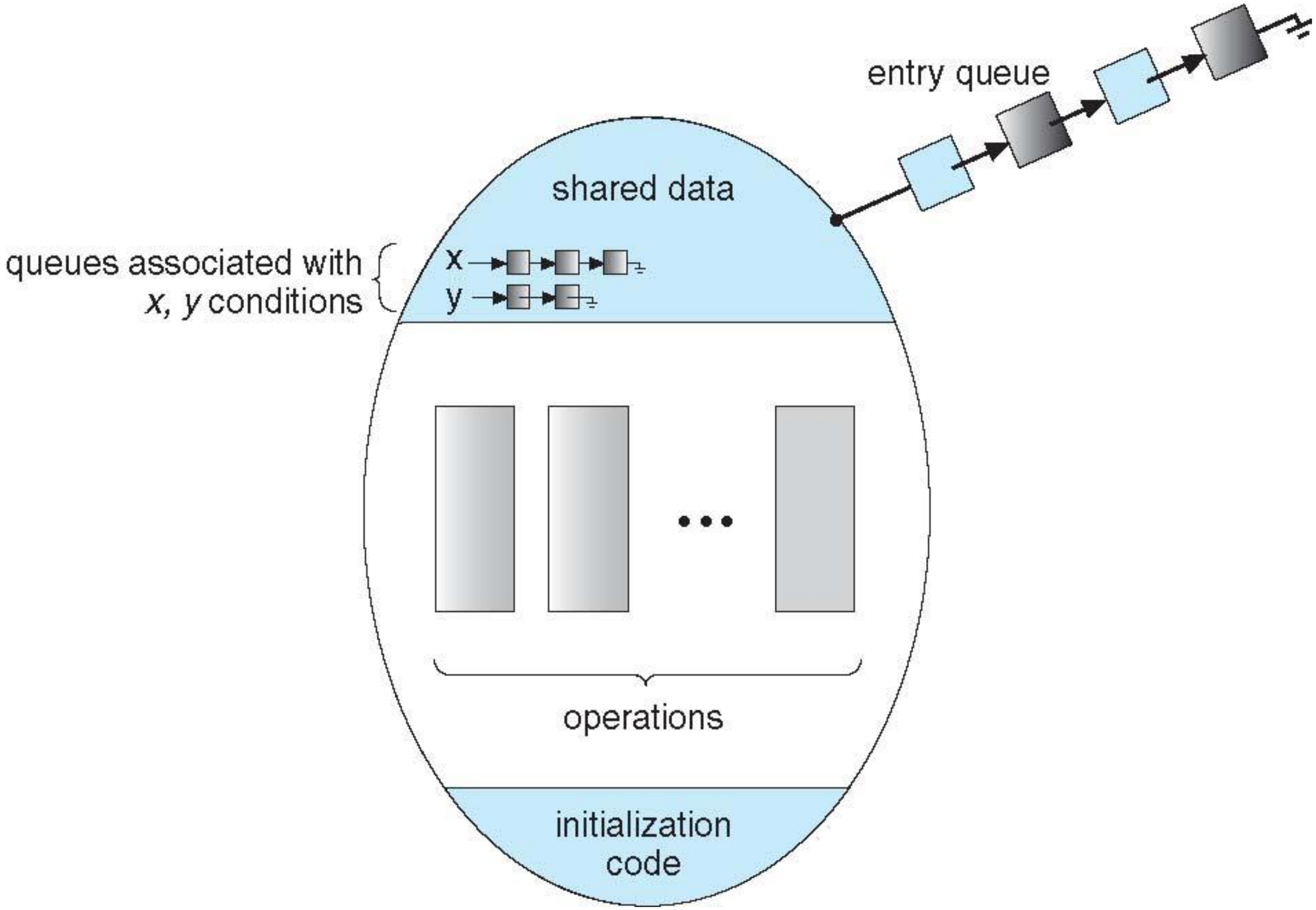
Condition Variables

`condition x, y;`

Two operations are allowed on a condition variable:

- **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
- **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
  - If no **`x.wait()`** on the variable, then it has no effect on the variable

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 paralel. 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 Mesa, C#, Java

Monitor Implementation Using Semaphores

Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

Each procedure ***F*** will be replaced by

```
wait(mutex);
        …
   body of F;
        …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

The operation x.wait can be implemented as:

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

Monitor Implementation (Cont.)

The operation **`x.signal`** can be implemented as:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming恢復Processes within a Monitor

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

Single Resource allocation

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;
   }
}
```

End of Chapter 6