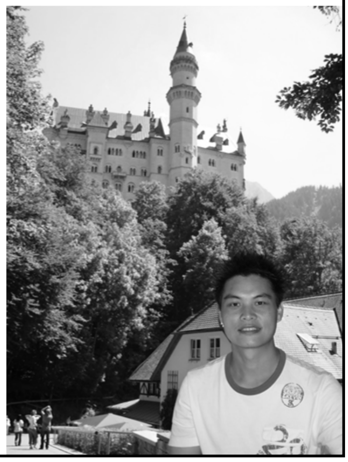
作業系統
Operating Systems

靜宜大學資訊傳播工程學系

劉國有 助理教授

kyliu@pu.edu.tw



Disadvantages of busy waiting or spin lock

- wasting CPU time: when a process wants to enter its critical region, it checks to see if the entry is allowed
- priority inversion problem: two processes in a computer with high priority, H, and with low priority, L. At a certain moment, with L in its critical region, H becomes ready to run. H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever.

Sleep & Wakeup

- Sleep is a system call that causes the caller to block, that is , be suspended until another process wakes it up
- Wakeup call has one parameter, the process to be awakened

Producer-consumer problem

- Two processes share a common, fixed-size buffer. The producer puts information into the buffer, and the consumer takes it out (also known as the bounded-buffer problem)
- Trouble arises when the producer wants to put a new item in the buffer, but it is already full.
  - Solution: the producer goes to sleep, to be awakened when the consumer has removed one or more items
- Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty.
  - Solution: the consumer goes to sleep until the producer put something in the buffer and wakes it up

```
#define N 100                                  /* number of slots in the buffer */
int count = 0;                                 /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          item = produce_item( );              /* generate next item */
          if (count == N) sleep( );            /* if buffer is full, go to sleep */
          insert_item(item);                   /* put item in buffer */
          count = count + 1;                   /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);    /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          if (count == 0) sleep( );            /* if buffer is empty, got to sleep */
          item = remove_item( );               /* take item out of buffer */
          count = count – 1;                   /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                  /* print item */
     }
}
```

Lost wakeup problem

(due to the access to *count* is unconstrained)

1. The buffer is empty and the consumer has just read count to see if it is 0.
2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer.
3. The producer inserts an item in the buffer, increments *count*, and notices that it is now 1.
4. Reasoning that *count* was just 0, and thus the consumer must be sleeping, the producer calls *wakeup* to wake the consumer up.

Lost wakeup problem

(due to the access to *count* is unconstrained)

5. Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost.
6. When the consumer next runs, it will test the value of *count* it previously read, find it to be 0, and go to sleep.
7. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

- Solution: to add a **wakeup waiting bit** (but it is a piggy bank for storing wakeup signals)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

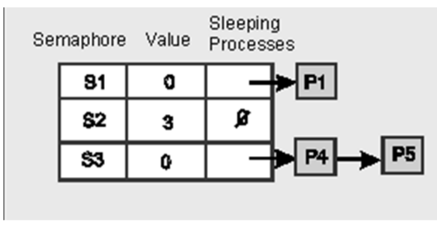
Semaphores

- A **semaphore** could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending
- Two operations: *down* and *up*
- Data type of semaphore:
```
struct semaphore{
  int val;
  process_list waiting;
}
```
- Down operation: checks to see if the value is greater than 0.
```
down(semaphore s){
   if (s.val > 0){
      s.val = s.val -1;
   } else{
      add this process to s.waiting;
      sleep();
   }

   return;
}
```

Semaphores

- Up operation: increments the value of the semaphore
```
up(semaphore s){
   if (s.waiting is not empty){
      remove a process p from s.waiting;
      wakeup(p);
   }else{
      s.val = s.val + 1;
   }
}
```



- Both operations are **atomic actions (indivisible)**: once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.
- ***Atomic actions****: a group of related operations are either all performed without interruption or not performed at all*

Semaphores

- Semaphores for enforcing mutual exclusion
  - Example: any critical section problem for n processes
  - Solution:
    - The *n* processes share a semaphore **mutex**, initialized to 1
    - Use down(**mutex**) just before critical section, up(**mutex**) just after
- Semaphores for synchronization
  - Semaphores can be used to make sure that one thing happens before another across processes
  - Example: a synchronization problem for 2 concurrent processes, such that
    - Process 1 has a statement **S1**
    - Process 2 has a statement **S2**
    - Require that **S1 must be completed before S2 is executed**

Semaphores

- Solution:
  - The 2 processes share a semaphore **synch**, initialized to 0
  - In Process 1, have:
    S1;
    up(**synch**);
  - In Process 2, have:
    down(**synch**);
    S2;

Types of semaphores

- ***Counting semaphores***: semaphore can take on any value
- ***Binary semaphores***: semaphore may only have a value of 0 or 1
  - Most often used to provide mutual exclusion on a critical section

Solving the Producer-Consumer Problem Using Semaphores

- Three semaphores:
  - *full* is for counting the number of slots that are full.
  - *empty* is for counting the number of slots that are empty.
  - *mutex* is to make sure the producer and consumer do not access the buffer at the same time.
- The *mutex* is used for mutual exclusion (*binary semaphore*): designed to guarantee that only one process at a time will be reading or writing the buffer and associated variables.
- The *full* and *empty* semaphores (*counting semaphores*) are for synchronization: to guarantee that certain event sequences do or do not occur.

```
#define N 100                         /* number of slots in the buffer */
typedef int semaphore;                /* semaphores are a special kind of int */
semaphore mutex = 1;                  /* controls access to critical region */
semaphore empty = N;                  /* counts empty buffer slots */
semaphore full = 0;                   /* counts full buffer slots */

void producer(void)
{
      int item;

      while (TRUE) {                  /* TRUE is the constant 1 */
            item = produce_item( );   /* generate something to put in buffer */
            down(&empty);             /* decrement empty count */
            down(&mutex);             /* enter critical region */
            insert_item(item);        /* put new item in buffer */
            up(&mutex);               /* leave critical region */
            up(&full);                /* increment count of full slots */
      }
}


void consumer(void)
{
      int item;

      while (TRUE) {                  /* infinite loop */
            down(&full);              /* decrement full count */
            down(&mutex);             /* enter critical region */
            item = remove_item( );    /* take item from buffer */
            up(&mutex);               /* leave critical region */
            up(&empty);               /* increment count of empty slots */
            consume_item(item);       /* do something with the item */
      }
}
```

Problems

- Deadlock problem: two down operations in the producer's code were reversed in order.
- Starvation problem: if we access the waiting list associated with a semaphore in LIFO order.

```
void producer(void)
{
      int item;

      while (TRUE) {
            item = produce_item( );
            down(&empty);
            down(&mutex);
            insert_item(item);
            up(&mutex);
            up(&full);
      }
}
```

```
void consumer(void)
{
      int item;

      while (TRUE) {
            down(&full);
            down(&mutex);
            item = remove_item( );
            up(&mutex);
            up(&empty);
            consume_item(item);
      }
}
```

mutexes

- A mutex is a variable that can be in unlocked or locked state. (to solve the *spin lock* in busy waiting)

```
mutex_lock:
      TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
      CMP REGISTER,#0             | was mutex zero?
      JZE ok                      | if it was zero, mutex was unlocked, so return
      CALL thread_yield           | mutex is busy; schedule another thread
      JMP mutex_lock              | try again later
ok:   RET | return to caller; critical region entered
```

```
mutex_unlock:
      MOVE MUTEX,#0               | store a 0 in mutex
      RET | return to caller
```

Monitors

- To solve the deadlock problem in using semaphores (two downs in the producer's code were reversed in order)
- **Monitor**: a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Mutual exclusion achievement: only one process can be active in a monitor at any instant.

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

    procedure consumer( );
    .
    .
    .
    end;
end monitor;
```

The Producer-Consumer Problem with monitors

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Message Passing

- Two primitive methods: send and receive.
  - Send(destination, &message)
  - Receive(source, &message)
- The producer-consumer problem with M messages

```
#define N 100                               /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                              /* message buffer */

    while (TRUE) {
        item = produce_item( );             /* generate something to put in buffer */
        receive(consumer, &m);              /* wait for an empty to arrive */
        build_message(&m, item);            /* construct a message to send */
        send(consumer, &m);                 /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);              /* get message containing item */
        item = extract_item(&m);            /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```