Synchronization

M1 MoSIG : Operating Systems

Poupin Pierre Rouby Thomas

04/11/2014

1 Semaphores

Proposed by Dijkstra, Semaphores are special counters on which 2 operations are defined :

- P or wait on a semaphore s :

```
if (s.counter == 0)
  wait
  s.counter--
```

- V or post on a semaphore s :

```
s.counter++
```

1.1 Typical use :

- restrict the access to a region to a fixed number of threads. Example :

```
s = semaphore initialized to N
threads execute :
    wait(s);
        restricted section;
    post(s);
```

- as a lock (binary semaphore). Example :

```
s = semaphore initialized to 1
Some code as above for threads.
```

1.2 Example: Procuder/Consumer

In this example, we will use a circular buffer of size N as our queue. If an operation is not possible, threads wait until the operation is possible.

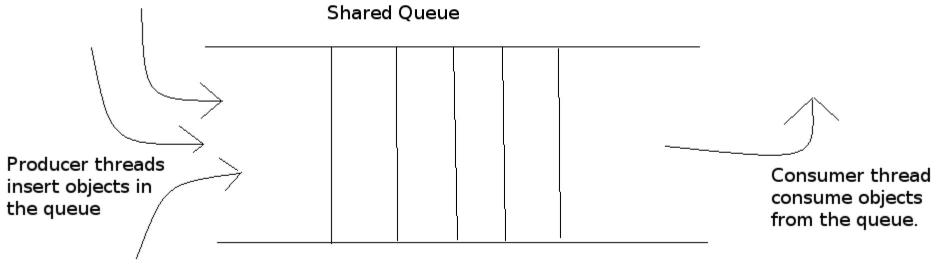

Figure 1: Situation of the producer/consumer example

```
semaphore s = 1, available = N, occupied = 0;

produce(int object) {

     //possibility to wait
     wait(available);
     // at most N producers
     wait(s);
     queue.buffer[(queue.head + queue.size) % N ] = object;
     queue.size++;
     post(s);
     post(occupied);
}

int consume() {
     int result;
     wait(occupied);
     // number of consumers waiting bounded...
     wait(s);
     result = queue.buffer[queue.head];
     queue.head = (queue.head +1) % N;
     queue.size--;
     post(available);
     post(s);
}
//Idea : use semaphores to count :
// the number of available slots (semaphore initialized to N)
// the number of occupied slots (semaphore initialized to 0)

Comment on the code: complicated because the order of wait operation matters.
```

1.3 Implementation of Semaphores :

A semaphore is :

- an integer value : counter
- a list of blocked processes : l
- a boolean variable : lock

```
typedef struct{
int lock;
list l;
int counter;
} semaphore_t;

// initially, lock=0, l is empty, counter is defined at semaphore creation

void wait(semaphore_t s){
     while(test_and_set(&s.lock,1) {}
     if (s.counter>0){
          s.counter--;
          s.lock =0;
     }
     else{
          tid = get_current_thread_id();
          unschedule(tid);
          mark_as_blocked(tid);
          insert(s.l,tid);
          s.lock = 0;
          schedule();
     }
}

void post(semaphore_t s) {
     while(test_and_set(&s.lock,1) {}
     if (is_empty(s.l)){
          s.counter++;
          s.lock =0;
     }
     else {
          tid = remove(s.l);
          mark_as_unblocked(tid);
          s.lock = 0;
     }
}
```

`Comment : the wait operation takes advantage of the scheduler to perform its task.`

2 Monitors

Introduced by Haare, monitors are objects in which methods are executed in mutual exclusion.

threads ⋮ ⋮ ⋮ — — > | attributes : : : |

Only one thread at a time can execute something inside the monitor .

This model makes things simpler because it is not necessary to manage locks anymore.

2.1 Example: Producer/Consumer

The queue will be implemented as a monitor, with buffer, size, head as attributes and two methods :

Thr monitors support two operations :

- wait(condition C) : releases the access to the monitor then waits until some signal is sent on C, then compete for the lock to retrieve access to the monitor.
- signal(condition C) : sends a signal on C. The signal is lost if no thread waits on C.

```
condition not_full , not_empty;

 produce(int object){
      while(size==N)
      wait(not_full);
      buffer[(head+size)%N] =object;
      size++;
      signal(not_empty);
 }

 int consume(){
      int result;
      while(size==0)
      wait(not_empty);
      result = buffer[head];
      head =(head+1)%N;
      size--;
      signal(not_full);
 }
```

2.2 Implementation

Similar to semaphores. Notice that both models are equivalent. It is possible to implement one of them using the other.