Parallel Programming
0024

Recitation Week 7

Spring Semester 2010

Today‘s program

Assignment 6

Review of semaphores

- Semaphores
- Semaphoren and (Java) monitors

Semaphore implementation of monitors (review)

Assignment 7

Semaphores

Special Integer variable w/2 atomic operations

- **P()** (***Passeren***, **wait/up)**
- **V()** (***Vrijgeven/Verhogen***, **signal/down*)***

Names of operations reflect the Dutch origin of the inventor ...

class Semaphore

```
public class Semaphore {
   private int value;
   public Semaphore() {
            value = 0;
     }
   public Semaphore(int k) {
            value = k;
     }
   public synchronized void P() { /* see next slide */ }
   public synchronized void V() { /* see next slide */ }
}
```

P() operation

```
public synchronized void P() {
    while (value == 0) {
      try {
        wait();
      }
      catch (InterruptedException e) { }
    }
    value --;
  }
```

V() operation

```
public synchronized void V() {
    ++value;
    notifyAll();
  }
```

Comments

You can modify the value of an semphore instance only using the P() and V() operations.

- **Initialize in constructor**

Effect

- **P() may block**
- **V() never blocks**

Application of semaphores:

- **Mutual exclusion**
- **Conditional synchronization**

2 kinds of semaphores

Binary semaphore

- Value is either 0 or 1
- Supports implemention of mutual exclusion

  Semaphore s = new Semaphore(1);

  s.p()

  //critical section

  s.v()

Counting (general) semaphore

- Value can be any positive integer value

Fairness

A semaphore is considered to be “fair“ if all threads that execute a P() operation eventually succeed.

Different models of fairness –

Semaphore is “unfair”: a thread blocked in the P() operation must wait forever while other threads (that executed the operation later) succeed.

Semaphores in Java

java.util.concurrent.Semaphore

acquire() **instead of P()**

release() **instead of V()**

Constructors

Semaphore(int permits);

Semaphore(int permits, boolean fair);

- **permits: initial value**
- **fair: if true then the semphore uses a FIFO to manage blocked threads**

Semaphores and monitors

Monitor: model for synchronized methods in Java

Both constructs are equivalent

One can be used to implement the other

Example from Mar 18

See slides for context

Buffer using condition queues

```
class BoundedBuffer extends Buffer {
  public BoundedBuffer(int size) {
      super(size);
  }
  public synchronized void insert(Object o)
        throws InterruptedException {
   while (isFull())
       wait();
   doInsert(o);
     notifyAll();
  }// insert
```

Buffer using condition queues, continued

```
// in class BoundedBuffer
public synchronized Object extract()
          throws InterruptedException {
  while (isEmpty())
          wait();
  Object o = doExtract();
  notifyAll();
  return o;
 }// extract


} // BoundedBuffer
```

Emulation of monitor w/ semaphores

We need 2 semaphores:

One to make sure that only one synchronized method executes at any tiven time

call this the “access semaphore” (S)

binary semaphore

One semaphore to line up threads that are waiting for some condition

call this the “condition semaphore” (SCond) – also binary

threads that wait must do an “acquire” and this will not complete

For convenience:

- **Counter waitThread to count number of waiting threads**
- **i.e., threads in queue for SCond**

Basic idea

1) **Frame all synchronized methods with S.acquire() and S.release()**

   This ensures that only one thread executes a synchronized method at any point in time

   Recall S is binary.

2) **Translate wait() and notifyAll() to give threads waiting in line a chance to progress (these threads use SCond)**

   To simplify the debate, we require that "notifyAll()" is the last action in a synchronized method

   Java does not enforce this requirement but the mapping of synchronized methods into semaphores is simplified.

Buffer with auxiliary fields

```
class BoundedBuffer extends Buffer {
  public BoundedBuffer(int size) {
      super(size);
    access = new Semaphore(1);
    cond = new Semaphore(0);
  }
  private Semaphore access;
  private Semaphore cond;
  private int waitThread = 0;
  // continued
```

1) Framing all methods

```
public void insert(Object o)throws InterruptedException {
   access.acquire(); //ensure mutual exclusion
   while (isFull())
         wait();
     doInsert(o);
     notifyAll();
   access.release();
 }// insert
```

Notes

There is one semaphore for *all* synchronized methods of one instance of "BoundedBuffer"

Why?

Must make sure that insert and extract don't overlap.

There must be separate semaphore to deal with waiting threads.

Why?

Imagine the buffer is full. A thread that attempts to insert an item must wait. But it must release the access semaphore. Otherwise a thread that wants to remove an item will not be able to executed the (synchronized!) extract method.

2) Translate wait and notifyAll

The operation

```
wait()
```

is translated into

```
waitThread ++
S.release(); // other threads can execute synchronized
                methods
SCond.acquire(); // wait until condition changes
S.acquire();
waitThread --
```

Each occurrence of `NotifyAll()` is translated

```
if (waitThread > 0) {
  for (int i=0; i< waitThread; i++) {
      SCond.release();
  }
}
```

All threads waiting are released and will compete to (re)acquire S.

They decrement waitThread after they leave SCond.acquire

Note that to enter the line (i.e., increment waitThread) the thread must hold the access semaphore S

Recall that S.release is done at the end of the synchronized method

So all the threads that had lined up waiting for SCond compete to get access to S

No thread can line up while the SCond.release operations are done since this thread holds S.

Note

We wake up only all threads – they might not be able to enter their critical section if the condition they waited for does not hold. But all threads get a chance.

- **This approach is different from what we discussed in the lecture**

Translate “wait()”

```
public void insert(Object o)  throws InterruptedException {
   access.acquire();
   while (isFull()) {
       waitThread ++;
       access.release(); // let other thread access object
       cond.acquire(); // wait for change of state
       access.acquire()
      waitThread --;
    }
   doInsert(o);
   notifyAll();
    access.release();
 }// insert
```

Translate “notifyAll()”

```
public void insert(Object o) throws InterruptedException {
   access.acquire();
   while (isFull()) {
       waitThread ++;
        access.release(); // let other thread access object
       cond.acquire(); // wait for change of state
        access.acquire()
         waitThread --;
    }
   doInsert(o);
   if (waitThread > 0) {
       for (int i=0; i<waitThread; i++) {
             cond.release(); } }
    access.release();
  }// insert
```

Example

Consider the buffer, one slot is empty, four operations

insert I1

insert I2

insert I2

extract E

I1 – access.acquire

I2 – access.acquire: blocks on access

I3 – access.acquire: blocks on access

I1 – waitThread == 0

I1.release

(cont)

I2 – access.acquire completes

I2 – buffer full

waitThread =1

I2 – access.release

I2 – cond.acquire – blocks on cond

I3 – access.acquire completes

I3 – buffer full

waitThread = 2

I3 – access.release

I3 – cond.acquire – blocks on cond

(cont)

E – access.acquire

remove item

E – SCond.release

E – SCond.release

E – access.release

One of I2 or I3 will succeed with access.acquire and be able to insert the next item

Exercise

How would the method “extract” look like (if we used semaphores to emulate monitors)?

Assignment 7

Hint: `Thread.currentThread()` returns a handle to the current thread and might be useful for the assignment.

Overview

Your task is to implement a Read/Write Lock

Max. 4 Threads

Max. 2 Reader Threads (shared access is allowed) and 1 Writer Thread

A thread that executes read() is a reader

At a later time it can be a writer ....

The challenge

No starvation

Efficient implementation

- **If there are fewer than 2 readers and no waiting writers then the next reader must be allowed to proceed**
- **If there is no contention then a thread must be allowed to proceed immediately**

Your implementation must be fair (you may want to use FIFOQueue.java)

Other comments

Keep your solution as simple as possible

Decide what you want to use [your decision]

- **Monitors (synchronized methods)**
- **Semaphores**
  - **http://java.sun.com/j2se/1.5.0/docs/api/**
  - **→ Semaphore**

Only change class Monitor.java

- **If you feel it‘s necessary to change other files, please let us know. ☺**

Please comment your code!

You may want to recall

Thread.currentThread().getId()

- **wait_list.enq(Thread.*currentThread().getId()*)**
- **wait_list.getFirstItem() == Thread.*currentThread().getId()***

Your log could look like this ...

READ LOCK ACQUIRED 1

READ LOCK RELEASED 0

WRITE LOCK ACQUIRED 1

WRITE LOCK RELEASED 0

READ LOCK ACQUIRED 1

READ LOCK ACQUIRED 2

READ LOCK RELEASED 1

READ LOCK ACQUIRED 2

READ LOCK RELEASED 1

READ LOCK ACQUIRED 2