SFDV3006
Concurrent Programming

Lecture 4 – Semaphore
Properties of Concurrent Systems

Introduction

- Last week we have seen why synchronization is needed to prevent interference which causes race conditions and corrupts the data
- This week we cover
  - Semaphores
  - Semaphores for synchronization and condition Synchronization
  - Safety and liveness properties
  - Some classical synchronization problems using semaphores

Semaphores

- Semaphores are most commonly used synchronization tool in concurrent systems such as operating systems
- Semaphores can be used for both synchronization and condition synchronization.
- A semaphore has **a single integer *value*** with two methods ***up()*** and ***down()***
  - ***up()*** increments the value of the semaphore value by 1. up() is also called *signal()*
  - ***down()*** decrements the value of the semaphore by 1 and blocks the calling thread or process if the count is 0. *down()* is also called *wait()*
  - The blocked processes are held in a FIFO queue
  - The value of the semaphore can never become negative
  - Calling up also notifies others threads that the value of the semaphore has changed
  - Semaphores should be initialized to some non-negative value

Semaphore - pseudocode

***down(s)*:** **if** *s* >0 **then**
decrement *s*
else
block execution of the calling process

***up(s)*:** **if** processes blocked on *s* **then**
awaken one of them
else
increment *s*

From Magee & Kramer

Semaphores in Java

- Semaphores are in Java since JDK 1.5 (Java 5) however we will implement the semaphore using a Java monitor use wait() and notify () to illustrate how easy it is to implement semaphores

```
public class Semaphore {
  private int value;

  public Semaphore (int initial)
    {value = initial;}

  synchronized public void up() {
     ++value;
     notify();
  }

  synchronized public void down()
       throws InterruptedException {
    while (value == 0) wait();
    --value;
  }
}
```

Semaphores are passive entities and have to be implemented as monitors

In reality semaphores are a low level construct used to implement monitors

From Magee & Kramer

Using semaphore for mutual exclusion

- One semaphore for each critical section
- Initialize semaphore to 1
- Embed critical sections in *down()* (wait) / *up()* (signal) pair
- Example of usage in Java

```
Semaphore mutex = new Semaphore(1);
mutex.down();
//critical section//
mutex.up();
```

How does this prevent race conditions due to interference?

Using semaphores as locks -2

```
public class Sequence {
   private int next = 0;
   public int getNextInt() {
      next = next + 1;
      return next;
   }
}
```

Code from lecture 1 – class is not thread safe

```
public class Sequence {
   private int next = 0;
   Semaphore mutex = new Semaphore(1);
   public int getNextInt() {
      mutex.down();
      next = next + 1;
      mutex.up();
      return next;
   }
}
```

Thread safe version of Sequence using Semaphore as a lock

Advantages / disadvantages of semaphores

- Advantages
  - Nice and simple mechanism
  - Can be efficiently implemented
  - Available in every programming language
- Disadvantages
  - Low level of abstraction
  - Errors can occur in usage such as using up before down or never signalling (up is never called after executing critical section)
  - Omitting signal (up) leads to deadlocks
  - Omitting wait (down) leads to safety violations

Using semaphores for condition synchronization

- Semaphores can be used for resource control and condition synchronization since a semaphore can act as a counter using its up() and down() methods
- Consider a database similar to lecture 3 for the readers-writers problem however assume that now our database is a read only database and for performance reasons we not want more than 20 concurrent readers at any one time
- To solve this problem we initialize the semaphore to 20
- The reader threads will decrement the semaphore value before reading using down(). When the number of readers is 20 the count will be zero and any more readers will be blocked
- As from the previous lecture the Database monitor class has the **actions** of *startRead* and *endRead* which become the methods of the monitor (see next page)

ReadOnly Database Using Semaphores

```
public class ReadOnlyDatabase {

    //initialize to maximum number of concurrent readers
   private Semaphore readers = new Semaphore(20);

   public void startRead() {
      readers.down();
   }

   public void endRead() {
      readers.up();
   }
}
```

Here the condition synchronization is does by the semaphore – will allow a max of 20 thread in the ReadOnlyDatabase

up() will signal and notify other waiting readers

Safety and liveness properties

- A **property** is an attribute of a program that is true for every possible execution of that program.
- Properties of interest for concurrent programs fall into two categories:
  - ***safety***
  - ***liveness***
- A **safety property** asserts that nothing bad happens during execution.
  - Safety is concerned with a program not reaching bad state
- A **liveness property** asserts that something good eventually happens.
  - Liveness is concerned with a program eventually reaching a good state.
- In sequential programs the most important safety property is that the final state is correct.
- For concurrent programs, important safety properties are mutual exclusion and absence of deadlocks.

Safety and liveness properties – cont'd

- The most important liveness property for a sequential program is that it eventually terminates.
- In concurrent programming, we will usually deal with systems which do not terminate
- Other liveness issues are related to resource access
  - Are process requests for shared resources eventually granted?
- Liveness properties are affected by the scheduling policy that determines which of a set of actions are chosen for execution.

Progress Property

- A **progress property** asserts that it is always the case that an action is *eventually* executed.
- Progress is the opposite of starvation
- **Starvation** is the name given to a concurrent programming situation in which an action is never executed.
  - For example in the Readers-Writers problem from Lecture 3 if the readers arrive continuously the writer(s) will never get a chance to execute – they will suffer from starvation
  - For the Producer-Consumer problem with unbounded buffer the producer can run continuously and there will be no chance for the consumer to run remove some items from the buffer

Preventing writer starvation in Readers-Writers

```
class DatabaseWithWritePriority {

  private int readers = 0;
  private boolean writing = false;
  private int waitingW = 0; // no of waiting Writers.

  public synchronized void startRead()
              throws InterruptedException {
    while (writing || waitingW > 0)
       wait();
    ++readers;
  }

  public synchronized void endRead() {
    --readers;
    if (readers == 0)
       notify();
  }
```

Need some way to let readers know that there is a waiting writer

Code from Magee &Kramer

For Writer methods see next slide

Preventing writer starvation in Readers-Writers

```
synchronized public void acquireWrite() {
    ++waitingW;
    while (readers > 0 || writing) try{ wait();}
          catch(InterruptedException e){}
    --waitingW;
    writing = true;
  }

  synchronized public void releaseWrite() {
    writing = false;
    notifyAll();
  }
}
```

Whenever a writer wants to write it first increments `waitingW`, however when it starts writing it decrements it

Is this solution fair to both the readers and writers?

BoundedBuffer using Semaphore

We use two semaphores to count the number of items instead of **count**

```
class SemaBuffer implements BufferInterface{
  …
  Semaphore full;  //counts number of items
  Semaphore empty; //counts number of spaces

  SemaBuffer(int size) {
    this.size = size; buf = new Object[size];
    full = new Semaphore(0);
    empty= new Semaphore(size);
  }
…
}
```

For put() and take() see next slide

BoundedBuffer using Semaphore - 2

```
synchronized public void put(Object o)
              throws InterruptedException {
   empty.down();
   buf[in] = o;
   ++count; in=(in+1)%size;
   full.up();
 }
 synchronized public Object take()
              throws InterruptedException{
   full.down();
   Object o =buf[out]; buf[out]=null;
   --count; out=(out+1)%size;
   empty.up();
   return (o);
 }
```

This leads to a deadlock because of the ***nested monitor problem***.
A thread which enters put() or take() takes the lock for the buffer and also the lock for the empty / full semaphore before it can execute the down() method.
When the producer find the empty semaphore is 0 it blocks and release the lock for empty semaphore but not for the buffer.
Similarly when the consumer find the full semaphore is 0 it blocks and Release the lock for the full semaphore but for the buffer.

Corrected BoundedBuffer using Semaphore

```
 public void put(Object item)
              throws InterruptedException {
    empty.down();
    synchronized (this) {
       buf[in] = o;
       ++count; in = (in + 1) % size;
     }
    full.up();
  }
  public Object take()
              throws InterruptedException{
    full.down();
   Object item = null;
    synchronized (this) {
        item = buf[out]; buf[out] = null;
       --count; out=(out+1)%size;
    }
    empty.up();
    return (item);
  }
```

This code prevents deadlock because the object lock is acquired after decrementing the semaphores

Resources and references

- Concurrent Programming in Java, 2e, Doug Lea, Addison Wesley
- Concurrency: State Models & Java Programs, 2e, Jeff Magee & Jeff Kramer, Wiley
- Java Concurrency tutorial – is the official Java tutorial for using the Thread API and concurrency in Java - *http://bit.ly/jconctut*
- *Course Website – http://sfdv3006.wikidot.com*