SFDV3006
Concurrent Programming

Lecture 3 – Monitors and Condition Synchronization

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
  - Monitors
  - Condition Synchronization
  - Implementing condition synchronization in Java
  - Some classical synchronization problems

Monitors

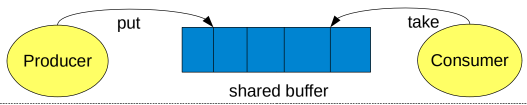
- Monitors are language features for concurrent programming.
- A monitor encapsulates data which can only be observed and modified by monitor access procedures.
- Only a single access procedure may be active at any time.
- An access procedure has mutually exclusive access to data variables in the monitor.
- An object satisfies the data access requirement of a monitor since it encapsulates data, which if declared private can be accessed only by the object's methods.
- The object's methods can be synchronized to provide mutually exclusive access.
- A monitor in Java is a class that has synchronized methods and whose members are encapsulated.

Monitors – condition synchronization

- Monitors support condition synchronization in addition to ensuring that access to data that they encapsulate is mutually exclusive.
- Condition synchronization permits a monitor to block threads until a particular condition holds, such as count becoming non-zero, a buffer becoming empty or new input becoming available.

Condition Synchronization example

- Assume that we have two threads - **a producer** thread and **a consumer** thread which share a common *buffer of fixed size*
- A producer produces some data for the consumer to consume and puts this data in the buffer
- A consumer takes the data from the buffer and consumes it



Condition synchronization

- Can the producer always run?
- Can the consumer always run?
- Producer cannot run when the buffer is full – the producer will block until condition of the buffer changes
- Consumer cannot run when the buffer is empty – he consumer will block until the condition of the buffer changes
- This is called condition synchronization
- Producer – Consumer is one of the classical problems in concurrency and synchronization. This problem is also called the ***bounded buffer problem***

Implementing the Bounded buffer

- How do we implement this ?
- We need to identify the monitor
- Need to identify the threads
- Threads – the active entity which will do actions
- Monitor – the passive entity which responds to the actions
- For the bounded buffer problem we know that the active entities are the producer and consumer – so they need to be implemented as threads
- The monitor will be the shared buffer on which the threads do the *put* and *take* actions – these actions will become the methods of the monitor

Implementing the Bounded buffer problem -2

```
//put() called by producer
public synchronized void put(Object item) {
   while (count == BUFFER_SIZE)
      Thread.yield();
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}

//take() called by consumer
public synchronized Object take() {
   Object  item;
   while (count == 0)
      Thread.yield();
   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   return item;
}
```

Our first try to implement condition synchronization

Why are we doing yield() here?

Implementation of the ***Buffer*** class as a Java Monitor – only the action methods are put() and take() are shown here.

Locks causing deadlocks

- We have made the *put()* and *take()* as *synchronized* since we do not want both the producer and consumer to run at the same time as there will be a race condition on *count*
- However the code in the previous slide can cause a ***deadlock*** because of the way we have implemented condition synchronization
- Assume that the buffer is full and producer is running – it gets the lock for the buffer and enters into the *put()* method however it cannot continue because the buffer is full so it does *yield()* - so that the consumer can run.
- At this point the consumer starts to run but it cannot access the buffer object because the lock for is still with the producer.
- Here the producer is waiting for the consumer to remove some item from the buffer and reduce the *count* so that it can run and the consumer is waiting for producer to release the object lock
- Neither the Producer nor the Consumer will make any ***progress***. Both the producer and consumer are in a deadlock.
- This situation where every thread is waiting for the other thread and neither makes any progress is called a **deadlock**

Preventing deadlocks

- The deadlock was caused by the yield() method which does not release the object lock
- The deadlock situation in the previous slide caused by the yield() in the while loop can be prevented by using two methods: **wait()** and **notify()**.
- In addition to having a lock, every object also has a **wait set**. This wait set consists of a set of threads that are waiting for some condition to change in the shared object.
- When a thread enters a synchronized method, it owns the lock for the object. However this thread maybe unable to continue because a certain condition is not met. For example, the producer calls *put()* and the buffer is full or consumer calls *take()* and the buffer is empty
- Using ***wait()*** will allow a thread to release the lock and wait until the condition that will allow it continue is met. Since the lock is now free other threads can acquire the lock and change the condition in the shared object

Preventing deadlocks – the wait() method

- When a thread calls `wait()`, the following occur
  - the thread releases the object lock.
  - thread state is set to blocked.
  - thread is placed in the wait set.

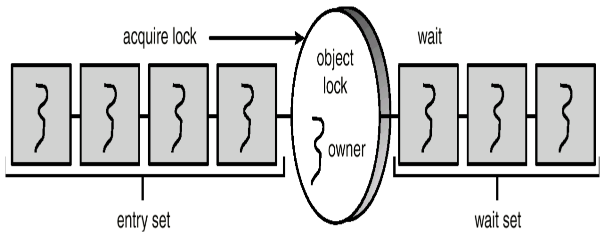


Image from Operating System Concepts with Java

Signalling using notify() / notifyAll()

- If the producer calls the put() method and sees that the buffer is full, it calls the wait() method. This call releases the lock, blocks the producer and puts the producer in the wait set for the object.
- Releasing the lock allows the consumer to enter the take() method, where it frees space in the buffer for the producer.
- How does the consumer thread signal that the producer may now enter the put() method?
- At the end of the synchronized put() and take() methods, notify/notifyAll() must be called.
- When a thread calls notify(), the following occurs:
  - selects an arbitrary thread *T* from the wait set.
  - moves *T* to the entry set.
  - sets *T* to Runnable.

  T can now compete for the object’s lock again.

Implementing the Bounded buffer problem -3

```
   //put() called by producer
   public synchronized void put(Object item) throws
InterruptedException {
      while (count == BUFFER_SIZE)
         wait();
      ++count;
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      notify();
   }

   //take() called by consumer
   public synchronized Object take() throws
InterruptedException {
      Object  item;
      while (count == 0)
         wait();
      --count;
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      notify(); return item;
   }
```

Our second try to implement condition synchronization

This works fine when there is only one Producer and only one Consumer thread

Implementation of the ***Buffer*** class as a Java Monitor – only the action methods are put() and take() are shown here.

notify() or notifyAll() - ?

- **`notify()`** selects an *arbitrary* thread from the wait set. This may not be the thread that you want to be selected.
- Java does not allow you to specify the thread to be selected.
- **`notifyAll()`** removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.
- **`notifyAll()`** is a conservative strategy that works best when multiple threads may be in the wait set.

Implementing the bounded buffer - 4

```
   //put() called by producer
   public synchronized void put(Object item) throws
InterruptedException {
      while (count == BUFFER_SIZE)
         wait();
      ++count;
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      notifyAll();
   }

   //take() called by consumer
   public synchronized Object take() throws
InterruptedException {
      Object  item;
      while (count == 0)
         wait();
      --count;
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      notifyAll(); return item;
   }
```

Our final try to implement condition synchronization

We use notifyAll() here – this is better when there are many producer and consumer threads

Implementation of the ***Buffer*** class as a Java Monitor – only the action methods are put() and take() are shown here.

Condition Synchronization in Java

- As we have seen in our solution to the producer consumer problem with bounded buffer, condition synchronization in Java is implemented with the following methods from **java.lang.Object** class
  - **`public final void notify() -`** Wakes up a single thread that is waiting in the object's wait set.
  - **`public final void notifyAll()-`** Wakes up all the threads that are waiting in the object's wait set.
  - **`public final void wait()throws InterruptedException -`** Waits to be notified by another thread. The waiting thread releases the synchronization lock associated with the shared object/monitor

Readers-Writers Problem

- The readers-writers problem involves a shared database or file which is concurrently accessed by many readers or writers
- This is also one of classical synchronization problems
- The solution to this problem is as follows:
  - Many readers can read concurrently
  - A writer may only write when there is no reader and no writer

Readers-Writers Problem - 2

- We need to analyze what will be the monitor and what will be threads
- Both the readers and writers would be threads whereas the monitor would be the shared database
- Reader actions
  - To read the database a reader follows the following steps
  - Start read → read → end read
- Writer actions
  - To write to the database a writer follows the following steps
  - Start Write → write → end write

The Database class

```
public class Database {
   public Database() {
      readerCount = 0;
      dbWriting = false;
   }

  //called by readers
  public synchronized int startRead() { //see next slides  }
  public synchronized int endRead()  { //see next slides }

  //called by writers
  public synchronized void startWrite() { //see next slides }
  public synchronized void endWrite()  {//see next slides }

  private int readerCount;
  private boolean dbReading;
}
```

This example from Chap 6 of Operating System Concepts With Java book

startRead() and endRead() methods

```
//startRead()
public synchronized int startRead() {
    while (dbWriting == true) {
        try {
          wait();
        }
        catch (InterruptedException e) { }
    ++readerCount;
    return readerCount;
}


//endRead()
public synchronized int endRead() {
      --readerCount
      if (readerCount == 0)
         notifyAll();
      return readerCount;
   }
```

startWrite() and endWrite() methods

```
//startWrite()
public synchronized void startWrite() {
  while (readerCount > 0 || dbWriting == true) {
       try {
           wait();
       }
       catch (InterruptedException e) { }
       dbWriting = true;
  }
}

//endWrite()
public synchronized void endWrite() {
   dbWriting = false;
   notifyAll(); //must use notifyAll for waiting readers/writers

}
```

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*