SFDV3006 Concurrent Programming

Lecture 6 – Deadlocks, livelocks, Starvation

Introduction

- Last week we covered semaphore and how to use them for both synchronization and condition synchronization
- This week we cover some of the problems of synchronization
  - Deadlocks
  - Livelocks
  - Starvation

Deadlock: four necessary and sufficient conditions

- Serially reusable resources: *the processes involved share resources which they use under mutual exclusion.*
- Incremental acquisition (hold and wait):

processes hold on to resources already allocated to them while waiting to acquire additional resources.

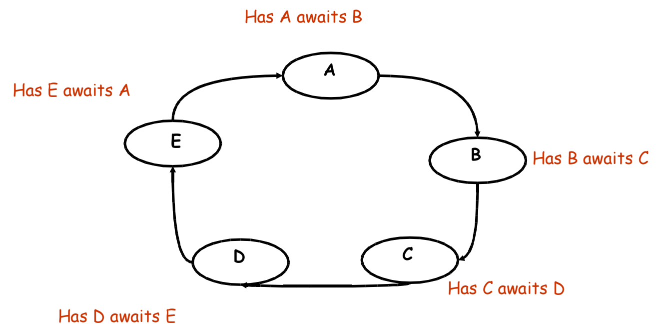
- No pre-emption:

once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.

- Wait-for cycle:

a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

Wait-for cycle



From Magee & Kramer

Deadlock avoidance

- Acquire resources in the same order
  - All the processes acquire resources in the same order
  - Assume that there are two threads *t1* and *t2* which use two semaphores *s1* and *s2* both of which are initialized to 1 as follows:

t1	*t2*
s1.down()	s2.down()
s2.down()	s1.down()
work....	work....
s1.up()	s2.up()
s2.up()	s1.up()

Deadlock can happen?

Deadlock avoidance

- Deadlock happens because resources (semaphores s1 and s2 in this example) are not acquired in the same order which leads to hold and wait and circular wait

t1	*t2*
s1.down()	**s1.down()**
s2.down()	**s2.down()**
work....	work....
s1.up()	s1.up()
s2.up()	s2.up()

No deadlock!

Deadlock avoidance – another example

- Assume that P and Q are two processes or threads that need to use a printer and scanner to scan from scanner and print to printer, both the printer and scanner are represented as monitors with *acquire()* and *release()* methods
- *acquire()* permits only one thread to use the resource at any one time and blocks other threads

	P	***Q***
	printer.acquire()	scanner.acquire()
	scanner.acquire()	printer.acquire
Deadlock??	copy....	copy....
	printer.release()	scanner.release()
	scanner.release()	printer.release()

Deadlocks while using locks

- Deadlocks are possible if locks are not acquired in correct order

```
class Monitor {
  private Object lock1 = new Object();
  private Object lock2 = new Object();

  public void doX(){
    synchronized(lock1){
      synchronized(lock2){
                  doWorkX();
  } } }

  public void doY(){
    synchronized(lock2){
      synchronized(lock1){
                  doWorkY();
  } } }
}
```

Suppose there are two Threads t1 and t2 both of which call doX() and doY() repeatedly. Will t1 and t2 deadlock?

Download and run this code from the website

Deadlocks while using locks - 2

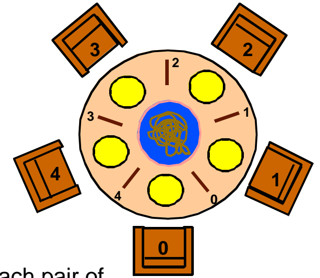
- Deadlocks are not possible all threads *acquire locks in the same order*

```
class Monitor{
  private Object lock1 = new Object();
  private Object lock2 = new Object();

  public void doX(){
    synchronized(lock1){
      synchronized(lock2){
                  doWorkX();
  } } }

  public void doY(){
    synchronized(lock1){
      synchronized(lock2){
                  doWorkY();
  } } }
}
```

Suppose there are two Threads t1 and t2 both of which call doX() and doY() repeatedly. Will t1 and t2 deadlock?

Deadlocks – dining philosophers problem

Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti.



One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.

How will deadlock happen? Solutions?

Livelock

- Livelock is a similar situation like a deadlock however the processes are not blocked as in a deadlock since they run continuously trying to do some operation
- As in a deadlock processes/threads in a livelock make no progress
- A livelock like a deadlock violate the progress property because threads do not run to their completion

Livelock - 2

- Example: consider the same example of deadlock between P and Q on slide 7

P	***Q***
printer.acquire()	scanner.acquire()
scanner.acquire()	printer.acquire()

or timeout/retry if printer/scanner unavailable and release the held resource

copy....	copy....
printer.release()	scanner.release()
scanner.release()	printer.release()

We do timeout to prevent a deadlock. There will be no deadlock but there will be a live Lock since both the P and Q release and retry again and same situation happens again..

Starvation

- Starvation is a scenario where the actions of one thread or process do not get chance to execute
- This is a violation of progress.
- For example in unbounded buffer producer can always run because the buffer is never full, so now the consumer may never get a chance to run
-

Starvation - Database monitor

```
class Database{
   private protected int readers = 0;
   private protected boolean writing = false;

   synchronized public void acquireRead(){
       while(writing){.. wait();..} ++readers;
   }

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

   synchronized public void acquireWrite(){
       while(readers>0 || writing) {..wait(); ..}
       writing = true;
   }

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


Writer starvation

- `notifyAll()` awakes both readers and writers.
- <u>Program relies on Java having a fair scheduling strategy.</u>
- When readers continually read a resource:

  Writers never get a chance to write. This is called starvation.
- Solution: Avoid writer starvation by making readers defer if there is a writer waiting.

Database monitor v2

```
class Database{
  //same as before
  private int waitingWriters = 0; //no of waiting writers
  synchronized public void acquireRead(){
    while(writing || waitingWriters > 0){..wait();..}
    ++readers;
  }
  synchronized public void releaseRead(){..}
  synchronized public void acquireWrite(){
  while(readers > 0 || writing){
    ++waitingWriters;
    try { ..wait(); }
  }
  --waitingWriters; writing = true;
  }
  synchronized public void releaseWrite(){ ..}
}
```


Reader starvation

- If there is always a waiting writer: readers starve.
- Solution: Alternating preference between readers and writers
  - Add another `boolean` attribute `readersTurn` in the database monitor that indicates whose turn is it
  - `readersTurn` is set by `releaseWrite()` and cleared by `releaseRead()`

Database monitor v3

```
public class Database{
   //same as before
   private boolean readersTurn = false;
   synchronized public void acquireRead(){
   while(writing || waitingWriters > 0 && !readersTurn){
       ..wait();..
   }
        ++readers;
   }
   synchronized public void releaseRead(){
       --readers; readersTurn = false;
       if(readers == 0) notifyAll();
   }
   synchronized public void acquireWrite(){…}
   synchronized public void releaseWrite(){
       writing = false; readersTurn = true;
       notifyAll();
   }
}
```

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*