Uses of Threads

CS151

Chris Pollett

Nov. 30, 2005.

Outline

- Bounded Queues
- Producer Consumer
- Bounded Queues with Guards
- Liveness Failures
- Socket Basics

Bounded Queue

- The book gives several implementations for a bounded queue.
- A bounded queue supports the methods put(o) -- to put an object at the end of the queue, get() -- to get the object at the start of the queue, isEmpty(), isFull(), getCount().
- A bounded queue has some fixed size after which put(o) will fail.
- The first implementation called BoundedQueue in the book makes no use of threads or synchronization and is quite straightforward.
- This would be a reasonable implementation in a non-threaded environment.
- It is not thread safe. That is, if two threads had access to the same BoundedQueue object q, one could do a put while a get was executing and an inconsistent state might result.

Synchronized Bounded Queue

- To make the original BoundedQueue thread-safe one can sub-class it to make a class SyncBoundedQueue.
- For each method in the sub-class override it and make it synchronized. For example,

```
synchronized public void put(Object e)
{
        super.put(e);
}
```

Producer Consumer

- A typical use of a synchronized bounded queue is in a producer consumer setting.
- Here we have two threads one a Producer of items which are put into a given queue another consumer of items from this queue.

Producer

```
public class Producer extends Thread
{
        protected BoundedQueue queue;
        protected int n;

        public Producer(BoundedQueue queue, int n)
        {
                this.queue = queue;
                this.n = n;
        }
        public void run()
        {
             for(int i=0; i <n; i++)
             {
                queue.put(new Integer(i));
                System.out.println("produce"+i);
                try{sleep((int)(Math.random()*100);}catch(InterruptedException e){}}
             }
        }
}
```

Consumer

```
public class Consumer extends Thread
{
        protected BoundedQueue queue;
        protected int n;

        public Consumer(BoundedQueue queue, int n)
        {
                this.queue = queue;
                this.n = n;
        }
        public void run()
        {
             for(int i=0; i <n; i++)
             {
                Object obj = queue.get();
                if(obj != null) System.out.println("\tconsume"+obj);
                try{sleep((int)(Math.random()*400);}catch(InterruptedException e){}}
             }
        }
}
```

Driver Code

```
SyncBoundedQueue queue = new
    SyncBoundedQueue(5);
new Producer(queue, 15).start();
new Consumer(queue, 10).start();

/*the output roughly consumes one thing for each for
    4 produced up until the end when everything is
    consumed. */
```

Guards

- We would like the Producer and Consumer to cooperate.
- We would like that if the Producer attempts to put a new item into the queue while it is already full, then it waits till the consumer consumes one.
- Similarly, if there is nothing to consume, the Consumer should wait on the Producer.
- A *guard* is a precondition for a certain action to complete successful.
- *Guarded suspension* is a requirement for threads to cooperate by:
  - Testing the guard before a method is executed
  - Executing only if the guard is true
  - Otherwise, temporarily suspending execution until the guard becomes true.

Queue with Guards

- We use wait() and notify() to implement guards on our queue.
- We subclass SyncBoundedQueue to make BoundedQueueWithGuard.
- We override put and get. For example:

```
synchronized void put(Object obj)
{
        try{while(isFull()){wait();}}
        catch(InterruptedException e){}
        super.put(obj); notify();
}
```

```
synchronized Object get()
{
        try{while(isEmpty()){wait();}}
        catch(InterruptedException e){}
        Object result=super.get(); notify();
        return result;
}
```

Liveness Failures

- Liveness refers to the desirable condition that will come about during the lifetime of a program.
- For instance, a certain task will complete, a thread should always respond to user input, the status of the systems should be constantly displayed and updated.
- Some things which might stop liveness are:
  - contention -- thread never gets to run
  - dormancy -- thread waits but is never notified
  - deadlock -- thread1 waits on thread 2 and thread 2 waits on thread 1.
  - premature termination -- a thread dies before it is supposed to preventing other thread from executing.

Socket Basics

- We would like to be able to communicate over the internet between two programs to do this we can use sockets.
- There are two types of sockets: Server sockets and client sockets.
- Example server socket code in Java looks like:

```
try{ ServerSocket s= new ServerSocket(port);
  while(true){Socket in= s.accept(); //spawn thread for this
   connection}
}catch(IOException ie){}
//s.close() could be used to close a connection
```

Client Socket

- A client can connect to a server using:

  Socket s= new Socket(host, port);

- One a connection is established a socket can read or write from its stream by getting an input or output stream:

  InputStream in = s.getInputStream();

  OutputStream out = s.getOutputStream();