CS-E4110 Concurrent Programming
Autumn 2016 - Tutorials
Java Monitors

2016-09-27

Task 1

Implement the interface BlockingStack using only Java monitors (i.e. `notify()`, `notifyAll()` and `wait()`). Do not use classes from the package `java.util.concurrent`. Your implementation must have a single constructor, that takes a positive integer as an argument for the stack capacity.

```
1  public interface BlockingStack<T> {
2
3      /**
4       * Add a new object to the top of the stack. If there is no free space, the
5       * call will block until space becomes available.
6       * @param object
7       * @throws InterruptedException
8       */
9      void push(T object) throws InterruptedException;
10
11     /**
12      * Remove the topmost object from the stack. If the stack is empty, the
13      * call will block until objects are pushed onto the stack.
14      * @return
15      * @throws InterruptedException
16      */
17     T pop() throws InterruptedException;
18
19     /**
20      * Return the number of items in the stack: 0 <= size <= capacity
21      * @return
22      */
23     int size();
24
25 }
```

Hint: A stack is often implemented by using an array as the backend data structure. Java does not support creating generic arrays, but you can instead use an array of type `Object[]` together with explicit type casting in `push(T object)` and `pop()`. Alternatively consider `Stack<T>` or `ArrayList<T>` from the package `java.util`.

Download links
BlockingStack.java Required interface

Task 2

Examine the class `SingleElementBox`. Is the implementation correct? Argue your answer or provide a trace to showcase unwanted behaviour.

```
1  public class SingleElementtBox<T> {
2      private T value = null;
3
```

```
4       public synchronized T get() {
5           while (value == null) {
6               wait();
7           }
8           T returnValue = value;
9           value = null;
10          notify();
11          return returnValue;
12      }
13
14      public synchronized void put(T newValue) {
15          while (value != null) {
16              wait();
17          }
18          value = newValue;
19          notify();
20      }
21  }
```

Download links
SingleElementBox.java Data structure implementation

Task 3

Read through the contents of the package `java.util.concurrent`. The package contains data structures and utilities useful for concurrent programming. Solve the following problem with the help of a class of your choosing from the package.

A *producer-consumer problem* is a classical problem of producer and consumer threads (or processes), which exchange data via a shared data buffer. Producers generate data asynchronously and consumers process this data asynchronously. Since non-existent data cannot be consumed and it is desirable to limit the size of the structure to avoid excessive memory consumption, the data structure is often a capacity bound FIFO structure with blocking operations for adding and removing elements.

Download the linked program skeleton and implemented the missing parts denoted by comments.

Download links
ProducerConsumerSystem.java Program skeleton

Task 4

A `ThreadPool` is a collection of threads which can execute a number of asynchronous tasks efficiently. Using the pooled threads eliminates the need to start a new thread for each individual task, which conserves resources since starting new threads is expensive. Implement the methods `addTask()` and `threadLoop()` for the following Java class:

```
import java.util.ArrayDeque;
import java.util.Queue;

public class ThreadPool {

    private static final int NUMBER_OF_THREADS = 4;
    private final Queue<Task> taskQueue = new ArrayDeque<Task>();

    public ThreadPool() {
        for(int i = 0; i < NUMBER_OF_THREADS; i++) {
            Thread t = new Thread() {
                @Override
                public void run() {
                    threadLoop();
                }
            };
            t.start();
        }
    }

    /**
    * Add a new non−null task to the task queue.
    */
    public void addTask(Task t) {
        //add code
    }

    /**
    * The pooled threads run in an infinite loop, acquiring tasks from
    * the queue and executing them concurrently.
    */
    private void threadLoop() {
        //add code
    }
}
```

Tasks are defined by the interface:

```
public interface Task {
    /**
    * Execute the computations for this task.
    * @throws Exception
    */
    public void process() throws Exception;

    /**
    * Each task has some error handling code, which is called if process()
    * encounters an error and throws an Exception.
    * @param e
    */
    void handleError(Exception e);
}
```

Download links

ThreadPool.java	Program skeleton for ThreadPool
Task.java	Interface definition for tasks

Task 5

Interested in astrophysics, you have written a program simulating the collision of our Milky Way galaxy and the nearby Andromedan galaxy. Because the gravity of each star affect the trajectory of every other star, the computational complecity of the problem is N^2, which means the simulation has to perform a huge number of operations per simulation step. This type of simulation is known as an n-body problem.

The program works correcly, but because it is sequential, it computes simuluation steps very slowly: roughly 1 - 2 minutes per frame on a desktop PC. Your task is to write a multi-threaded version of the program. Since everything else is ready, the remaining task is to write a multi-threaded versions of the following two for-loops:

```
Star[] stars = super.galaxies.getStars();

//Phase 1: Calculate changes to the velocities
//from gravity interactions
for(Star currentStar : stars) {
    for(Star otherStar : stars) {
        currentStar.accelerationFrom(otherStar);
    }
}

//Phase 2: Move the stars according to their
//updated velocity
for(Star currentStar : stars) {
    currentStar.updatePosition(super.simulationStep);
}
```

For both phases, the problem is data parallel: each star can be processed individually. Thus the solution can scale very efficiently to a high number of processors. The two phases must be executed sequentially.

The source is an Eclipse project. Import the zip-file with *File* → *Import* → *Existing projects into Workspace* → *Select archive file*. Write your implementation in `ThreadedSimulation.java`. The main method is in `NBody.java`.

How much speedup did you achieve compared to the sequential simulation? A relatively recent desktop PC should be able to compute a single simulation step in 10 - 20 seconds.

Download links

nbody-project.zip	Eclipse project archive
nbody-sample-hq.mov	High quality pre-rendered sample (170mb)
nbody-sample-lq.mov	Low quality pre-rendered sample (41mb)