Assignment 2

Java Concurrency and Synchronization

December 07, 2016

TAs in charge of the assignment : Hussien and Boaz.

Deadline of the Junit Tests is 13.12.16.
Deadline of the assignment implementation (both Part 1 and 2) is 01.01.17.
Extension's requests must be sent to Majeed **only**.

1 Introduction

In the following assignment you are required to implement a work stealing scheduler, testing it by implementing a simple parallel merge sort algorithm, and afterwards to use it to build a smart phone factory. The work stealing technique has become mainstream and is now often considered when it comes to to dynamically balance the work load among processors. The work stealing principle in this assignment can be synthesized as follows. Pool of Threads $P_1, P_2, ..., P_n$ where each thread runs a Processor, each with own queue are working in order to complete tasks in their queues. When a processor P_i is out of tasks it attempts to steal some from its neighbors..
As an SPL171 team, you are going to implement a work stealing scheduler.

2 Part 1: Work Stealing Scheduler

2.1 Description

In a work stealing scheduler, each processor in the computer system has a queue of work tasks to perform. while running, each task can spawn a new task or more that can feasibly be executed in parallel with its other work, it is advised but not mandatory that these new tasks will initially put on the queue of the processor executing the task. When a processor runs out of work, it looks at the queues of other processors and **steals** their work items.
Each processor is a thread which maintains local work queue. A processor can push and pop tasks from its local queue. Also, a processor can pop tasks from other processor's queue by the steal action, See Figure 1.

2.1.1 Data Structure

The choice of data-structures used for storing task queues affects the efficiency of task enqueuing and dequeuing. A naive approach is to keep a single-ended queue where both owner and thief dequeue from the same end. However, this is likely to lead to contention. Our choice in this assignment is to keep double-ended queue, where the owner dequeues from one end and the thief dequeues from another (see Figure 2).

2.1.2 Stealing Strategy

When it runs out of work, in order for a processor to steal from another one, it should select first a "victim" from which it will steal the task. The victims are selected in circular manner, i.e. thief i will search for a victim in the order $i+1, i+2, ..., N, 1, ..., i-1$ where N is the number of processors in the system (The thief starts searching

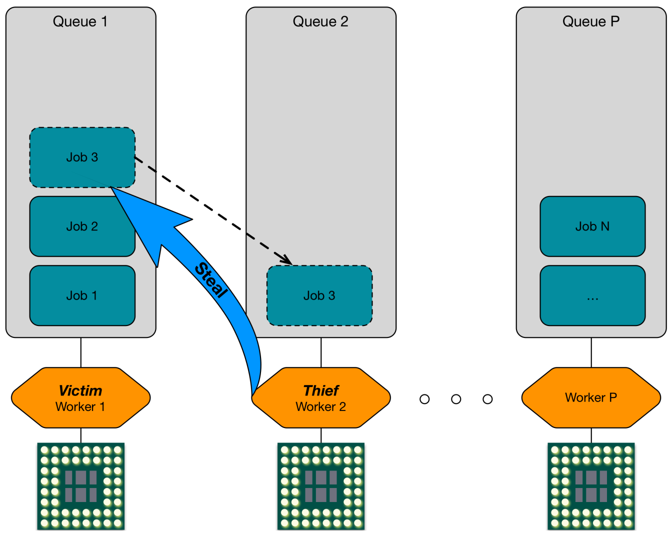


Figure 1: Work Stealing Scheduler

from $i + 1$ in each steal action) . At each stealing action, the thief steals **half** the number of tasks available on the victim processor. That is, if a victim processor has n tasks, the thief can attempt to steal up to $\lfloor \frac{n}{2} \rfloor$ tasks. If the processor failed to steal any task, it should wait and be notified when new tasks are inserted.

2.1.3 Dependency between tasks

In some applications (e.g. the smart phones factory) the tasks have interdependence and ordering constraints. Your scheduler should fulfill these constraints. The task's execution should be suspended until the tasks it depends on are completed.
Suspending a task should not suspend the processor, as processors and tasks are independent. When a task is suspended, the processor should continue with the next task on its queue, and the suspended task should be eventually continued only when all tasks it waits for them are done. There are two approaches to handle this:

- Rescheduling the suspended task on the **same** processor's queue, it means to put the suspended task on the processor's queue when it is ready to be continued. In some point later, the processor will fetch this task from the queue and handle it (this task can be stolen by another processor as well).
- The continuation of the task is executed directly when it is ready to be continued. The continuation is executed by any processor directly without the need to schedule the task again.

Note: In this document we refer to the first approach, i.e. rescheduling the task, which is the proper way to handle this. However, working with the second case will be accepted as well and will not affect your grade.
Note: When it is resumed, the suspended task should execute a continuation / callback function instead of restarting from scratch (a task might be suspended multiple times) - otherwise it will never end. Note that a task is resumed upon the completion of all the tasks it depends on (a.k.a., its child task or sub tasks).

2.2 Example of Work Stealing schedule

In order to explain how this scheduler works lets assume that we want to write a parallel version of merge sort. The merge sort gets as input a one dimensional array and sorts it.
Sorting an array is one "big" task, which will be assigned to one processor to perform it. However, in merge sort, this task spawns two additional tasks where each task is responsible to sort half of the original array, and the main task will merge the output of these new tasks to finish its work. This is also applied to the new tasks, where each one of them will spawn new two tasks, and so on until we reach an array with one element (see Figure 3).
At the beginning we have only one task, the scheduler will schedule it to one processor, where all the other processors will remain idle. The new spawned two tasks from the parent task are put on this processor queue, as well. Since

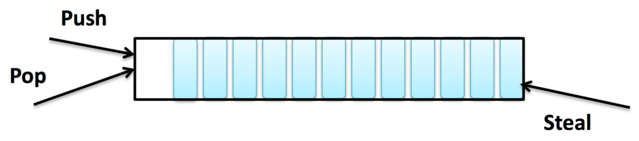


Figure 2: Work Stealing Queue

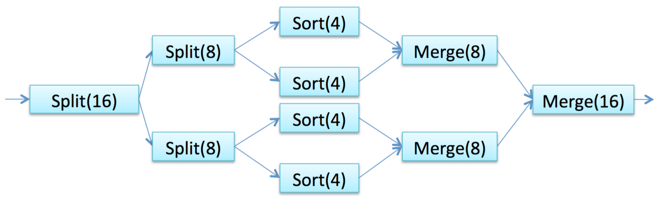


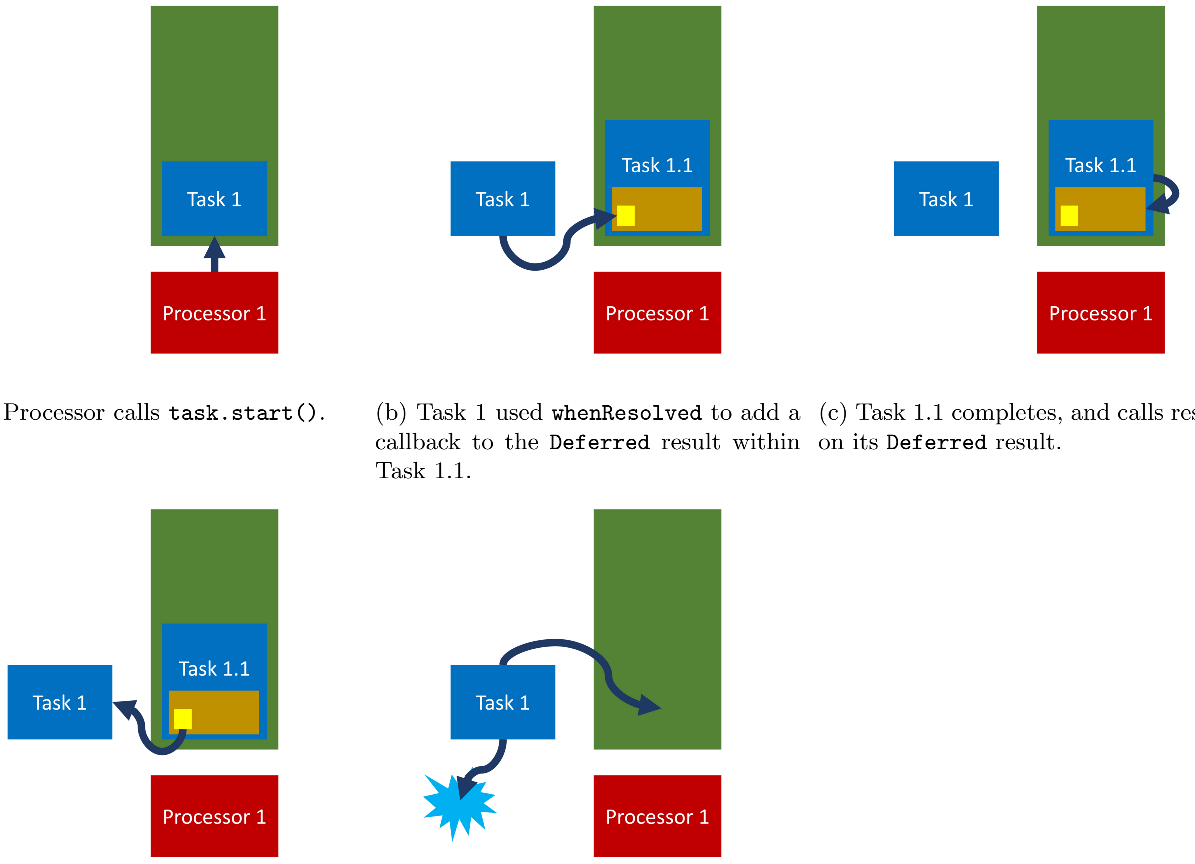
Figure 3: Merge Sort - tasks generated

all the other processors are out of work, each of them will try to steal tasks from other processors, per the stealing strategy defined above. By stealing, and as the more tasks are spawned, more processors will get work, and by this we dynamically balance the load of work on processors. In this example, the tasks are interdependent, in order to complete the parent task, there is a need to wait for the completion of child tasks, however, the processor who runs the parent task will continue running with other tasks, and complete the parent task when it is rescheduled again.

2.3 Implementation of the Work Stealing scheduler

In this part of work you are supplied with 5 classes that you will have to implement in order to construct a working work-stealing thread pool all of them are stored in the bgu.spl.a2 package. You are advised to download and read the javadoc of the supplied interfaces as a complementary material to this section. The rest of this section describes the given classes and the way they should be implemented.
To this end, in our framework each Processor will run on its own thread. Each Processor has own *id*. The WorkStealingThreadPool holds all the local queues of each Processor in our system. The WorkStealingThreadPool initially schedules the tasks to the processors by in the submit method.
The queue of each processor holds objects of type Task, the processor executes a task by invoking the handle method of this task method of this task. In order for a task to complete its computation, it might need results of another tasks (e.g. the merge sort). These results are crucial for its completion. When a task needs a result of another task, it subscribes itself to the Deferred object of that task. After the task spawn new child-tasks, it may wait for their results to be resolved via the Task::whenResolved method.
The Deferred is an object which represents a deferred result of an operation, this object will hold eventually the result of the operation when it completes its computation. Each task has it is own Deferred object, when the task completes its execution its result will be held by its Deferred object.
A deferred object support subscription with a callback using the whenResolved method, this callback will be invoked when the Deferred object is eventually resolved.
A task might need a result of several other tasks at once, then it needs to subscribe to all of the deferreds of these tasks, and should be continued **only** when all the tasks are completed. Resolving all the Deferred objects the task waits on them should result in rescheduling the subscribed task on the same processor which executed it before suspending (the task can also be stolen after rescheduling). At some point, the processor executes the rescheduled task again, it actually will execute the continuation of the task.
Figure 5 describes callback strategy for tasks.

When building a framework, one should change the way they think. Instead of thinking like programmer which writes software for end users, they should now think like a programmer writing a software for other programmers. Those other programmers will use this framework in order to build their own applications. For this part of the assignment you will build a framework (write code for other programmers), the programmer which will use your



(a) Processor calls `task.start()`.

(b) Task 1 used `whenResolved` to add a callback to the `Deferred` result within Task 1.1.

(c) Task 1.1 completes, and calls resolve on its `Deferred` result.

(d) The resolve function calls any callback functions stored in the `Deferred` result.

(e) Task 1 can either be requeued by the callback, or immediately resolved.

Figure 4: Callback strategy for tasks via the `whenResolved` function.

code in order to develop its application will be the future you while he works on the second part of this assignment. Attached to this assignment is a set of interfaces that define the framework you are going to implement. The interfaces are located at the bgu.spl.a2 package. Any Framework related classes that you will add in order to implement the framework should be under the package bgu.spl.a2. Read the javadoc of each interface carefully.

The following is a summary and additional clarifications about the implementation of different parts of the framework.

- **Task**$< R >$: an abstract class that represents a task that may be executed using the WorkStealingThreadPool. A task is an object which holds the required information to handle a task in the system. The task also holds a Deferred$< R >$ object which will hold its result. Each Task type extends the Task. Tasks objects are held on the local queues of the processors. A task can spawn new tasks. The main task methods are:
  - start: This is the abstract method, that should be implemented for each task type, it implements the task functionality.
  - spawn: This method schedules a new task (a child of the current task) to the same processor as the current task. Notice: it is not necessary that the task is depending on the spawned task result. That means, the spawned task can run in parallel with the parent task if a another process steals it.
  - whenResolved: add a callback to be executed once all the given tasks are completed. Once you find out that all the results has been resolved, re-add the task to the processor queue which will cause it to eventually handle the task again, when this happen this task should be able to recognize that it was

already been started and therefore execute the continuation instead of the start method.

  - complete: resolves the internal result - should be called by the task derivative once it is done.
  - getResult: returns the task deferred result. In order to handle the rescheduling you can add a helper method to Task which be called by a processor in order to start the task or continue its execution in the case where it has been already started.

- **Deferred**$< T >$: this class represents a deferred result i.e., an object that eventually will be resolved to hold a result of some operation, the class allows for getting the result once it is available and registering a callback that will be called once the result is available.
  Deferred includes these methods:
  - get: return the resolved value if such exists.
  - resolve: called upon completing the operation, it sets the result of the operation to a new value, and trigger all the subscribed callbacks.
  - whenResolved: add a callback to be called when this object is resolved if while calling this method the object is already resolved - the callback should be called immediately.
  - isResolved: return true if this object has been resolved.
- **Processor**: this class represents a single work stealing processor, it is a Runnable so it is suitable to be executed by threads. Each processor has *id*. A processor will always try to fetch tasks from its queue and execute them, and if it is out of work it will try to steal tasks from another processors (moving tasks from other processor queue to its one). In each steal action a processor steals **half** of the tasks in the "victim" processor.
  Important note : The tasks and processors are independent, that means, if a task is suspended for waiting to another tasks, the processor will NOT be suspended, it will continue with other tasks.
  The Processor includes these methods:
  - run: the main function, here the processor will fetch tasks and execute them.
- **VersionMonitor**: Describes a monitor that supports the concept of versioning - its idea is simple, the monitor has a version number which you can receive via the method getVersion() once you have a version number, you can call await() with this version number in order to wait until this version number changes. You can also increment the version number by one using the inc() method.
- **WorkingStealingThreadPool**: WorkingStealingThreadPool manages all the the threads in our system, it holds all the queues of the processors and supports functions of submitting, fetching and spawn tasks. Remember, synchronizing this class will affect all the processors in the system (and your grade!) **- no synchronized methods or any other type of locks are allowed in this class aside from the version monitor**
  The constructor of WorkingStealingThreadPool creates a WorkStealingThreadPool which has n processors. The WorkingStealingThreadPool includes these methods:
  - submit: Scheduling a task on a random processor. After the task got submitted, the processor can fetch and execute it.
  - start: start the threads belongs to this thread pool.
  - shutdown: closes the thread pool - this method interrupts all the threads and wait for them to stop - it returns only when there are no live threads in the queue. After calling this method, one should not use the queues anymore.

  The WorkStealingThreadPool holds all the local queues of the processor, so fetching a task by a processor is done via the WorkStealingThreadPool.

It is **mandatory** to read the all javadocs of the interfaces carefully, as these provides necessary hints of the implementation. You can add methods and fields to the classes. But you are NOT allowed to remove or change any method signature that we provided you with. Otherwise you will fail our automatic tests. Notice that you still can add the "synchronized" keyword to some methods if needed. **Remember**: In parallel programming, you should try to find good ways to avoid blocking threads as much as possible. This also applies to this assignment.

Figure 5 describes a typical task "life cycle" within the work stealing architecture.

2.4 JUnit Tests for The Work Stealing Scheduler

Before your class implementations are used in a production environment, you will have to verify that they work as expected. In order to do so, you must write tests for `Deferred` and `VersionMonitor`. Write at least one test for each public function in each class and submit them alongside your project. Be sure to follow the package and naming conventions detailed in the JUnit tutorial on the course website.
JUnit test are due for submission **before** the final assignment due date, check the course website for the deadline and be sure to submit them on time.

2.5 Testing Work Stealing scheduler: Simple Parallel Merge Sort

Before you hurry into implementing the smartphones factory application, you are required to test your implementation of work stealing scheduler. To achieve this you will implement a parallel merge sort algorithm based on your Work Stealing scheduler. You have to implement the class MergeSort which extends Task. This implementation is simple. You are encouraged to test your program with random arrays with different sizes.

2.6 An Example with Code

In this simple example we implement a program which gets a matrix, and its result is a vector where the $i-th$ coordinate holds the sum of the $i-th$ row. First we define a new task called SumRow which holds a reference to the matrix (array) and the number of the row it works on (r).

```
public class SumRow extends Task<Integer> {
    private int[][] array;
    private int r;
      public SumRow(int[][] array,int r) {
          this.array = array;
          this.r=r;
      }
      protected void start(){
        int sum=0;
        for(int j=0 ;j<array[0].length;j++)
           sum+=array[r][j];
        complete(sum);
      }

}
```

Each SumRow task returns one value which is the sum of the row, so the class is Task$<Integer>$. By complete(sum) we resolve the deferred of the task, and it will hold the sum.

The SumMatrix holds a reference to the matrix. In SumMatrix task, we spawn SumRow tasks where each of them is responsible for a result of one row. By calling whenRsolved we subscribe to all the deferreds of the tasks. When all the results are resolved (all SumRow tasks are done) the callback will be executed. In the callback we build the vector and assign it to SumMatrix result by complete(res). By calling getResult on the SumRow task we get a deferred of the result, and calling get() on this deferred we get the result (which is in this case the sum of a row which is of type Integer.)

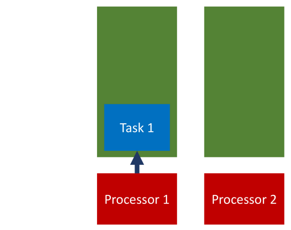
```
public class SumMatrix extends Task<int[]>{
private int[][] array;
public SumMatrix(int[][] array) {
    this.array = array;

}
```

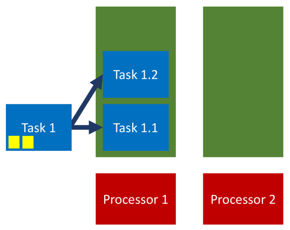
```
protected void start(){
   int sum=0;
   List<Task<Integer>> tasks = new ArrayList<>();
   int rows = array.length;
    for(int i=0;i<rows;i++){
       SumRow newTask=new SumRow(array,i);
       spawn(newTask);
       tasks.add(newTask);
    }
    whenResolved(tasks,()->{
       int[] res = new int[rows];
       for(int j=0; j< rows; j++){
          res[j] = tasks.get(j).getResult().get();
       }
        complete(res);
    }
   );
}
```

Note: do NOT call whenResolved from spawn method. Since as we mentioned, it is not in necessary that the task needs the spawned task result to continue.
Creating the thread pool and submitting the SumMatrix task are done in the main function by these lines:

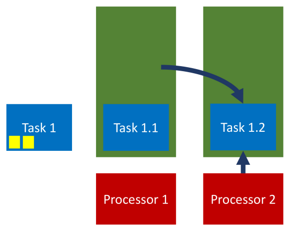
```
    WorkStealingThreadPool pool = new WorkStealingThreadPool(4);
    int[][] array = new int[5][10];
   // some stuff
    SumMatrix myTask = new SumMatrix(array);
    pool.start();
    pool.submit(myTask);
   //some stuff
    pool.shutdown(); //stopping all the threads
```



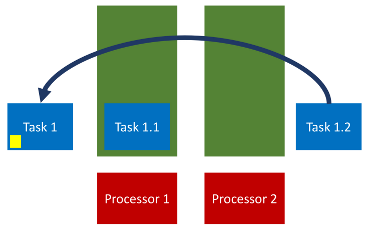
(a) Processor 1 calls the start function of Task 1.



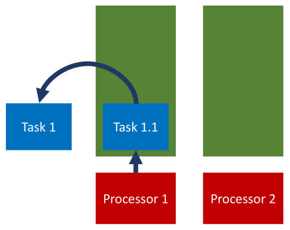
(b) Task 1 spawns two sub-tasks, while keeping a "promise" deferred object for each new task.



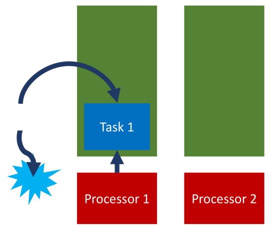
(c) Processor 2 steals Task 1.2 from the Processor 1 queue. Processor 2 then calls the start function for Task 1.2.



(d) Task 1.2 completes, and its callback decrements Task 1.



(e) Processor 1 calls the start function for task 1.1, which completes and its callback decrements Task 1.



(f) Task 1 no longer awaits additional callbacks and spawns itself to the Processor 1 queue.

Figure 5: Task "life cycle" - a sample of class interaction within the work-stealing architecture.

3 Part 2: Work Stealing Mobile Phone Factory

In this section you will simulate a mobile phone factory where diligent workers practice work stealing in order to quickly assemble products ordered by their clients.

Products and their parts are identified by *ID's* (these ID's will be used to check your work!). Parts are assembled according to *plans* using special *tools*. The following sections will detail the implementation of *plans*, *tools* as well as their interactions. Finally, the simulation framework and input format will be described.

3.1 Plans

A plan provides instructions for assembling a product using tools and parts. The plan consists of a *tool list* and a *part list*, each of these lists can either be empty, contain one item, or contain several items.

In order to complete a plan, a worker must first acquire all products in the part list. This is achieved by spawning a new manufacture task for each part. Once the last product in the part list is manufactured, the manufacturing task callback should trigger the part assembly process. Each product should keep a list of its parts.

During the part assembly process, each tool in the tool list must be acquired from the warehouse (see sec. 3.3). Once the tool is acquired its `useOn` function is called on each part in the part list, this is repeated for each tool in the tool list. Finally the return values of all `useOn` function calls are summed and become the ID of the resulting product.

3.2 Tools

The interface `Tool` requires two public functions:
getType() - Which returns a string describing the tool type.
useOn(Product p) - Which returns a `long` value representing the result of tool use.

Our mobile phone factory will stock the following tools in its warehouse:

3.2.1 RandomSumPliers

The random sum pliers initialize a `java.util.Random` with a seed equal to the product ID. The pliers then collect `[product id % 10000]` random integers from the `Random` object and sum them.

This sum then becomes the tools return value.

3.2.2 GCD Screwdriver

The GCD screwdriver returns the greatest common divider of `[product id]` and `reverse([product id])`.
Example: product id = 12345 return GCD(12345, 54321).
Hint: You may find some useful classes in `java.math` which will assist you in your implementation.

3.2.3 Next Prime Hammer

The next prime hammer returns the first prime number following the product id (i.e. if the product id is 7, the next prime hammer will return 11).

3.3 Warehouse

The warehouse class holds a finite amount of each tool, as defined in the simulation settings (see sec. 3.5). The warehouse will define the following functions:

- `acquireTool(String type)` - used by workers to acquire tool, returns a `Deferred<Tool>`. Do not use wait/notify here - instead use the "resolving" functionality of the `Deferred` class in order to use the tool once it is available via a callback.
- `releaseTool(Tool tool)` - Releases a tool, returning it to the warehouse inventory.
- `getPlan(String product)` - Returns the plan for constructing a `product` product.
- `addPlan(ManufactoringPlan plan)` - Adds a manufacturing plan to the warehouse.

The warehouse should be initialized using the the tool instances and manufacturing plans defined in sec. 3.5.

3.4 Simulator

The simulator class is tasked with running the simulation. This classes constructor receives only one parameter: a `WorkStealingThreadPool` instance. We may replace your `WorkStealingThreadPool` implementation with our own during testing, so be sure to implement all simulation functionality in the `Simulator` class, not `WorkStealingThreadPool`!

Once constructed, calling the simulators `start()` function will perform the following:

- Read a wave from the input file (see sec. 3.5).
- Add a manufacturing task for all products in the current wave to the task queue.
- Proceed to the next wave once **all** products in the current wave have been manufactured.

Finally, the `start()` function will return a `ConcurrentLinkedQueue` (see `java.util.concurrent`) containing all products manufactured during the simulation.

The result of the simulators `start()` function should be saved as a serialized object to the file `"result.ser"`. You may do so using this code, or similar:

```
ConcurrentLinkedQueue<Product> SimulationResult;
SimulationResult = SimulatorImpl.start();
FileOutputStream fout = new FileOutputStream("result.ser");
ObjectOutputStream oos = new ObjectOutputStream(fout);
oos.writeObject(SimulationResult);
```

Be sure to import the `FileOutputStream` and `ObjectOutputStream` classes.

3.5 Input Format

3.5.1 The JSON Format

All your input files for this assignment will be given as JSON files. You can read about JSONs syntax: http://www.json.org. In Java, there are a number of different options for parsing JSON files. Our recommendation is to use the library Gson. See the Gson User Guide (https://github.com/google/gson/blob/master/UserGuide.md) and APIs to see how to work with Gson. There are a lot of informative examples.

3.5.2 Input File

Defined below is a json input file that describes a single execution of the our shoe store. The simulation JSON file will be provided to your program as the first command line argument.
The following is an example of that file.

```
{
    "threads": 4,

    "tools": [
        {
            "tool": "gs-driver",
            "qty": 35
        },
        {
            "tool": "np-hammer",
            "qty": 17
        },
        {
            "tool": "rs-pliers",
            "qty": 23
        }
    ],
    "plans": [
            {
            "product": "yphone30",
            "tools": ["gs-driver", "rs-pliers"],
            "parts": ["5'-screen", "round-button"]
        },
        {
            "product": "5'-screen",
            "tools": ["np-hammer"],
            "parts": ["glass", "touch-controller"]
        },
        {
            "product": "glass",
            "tools": ["np-hammer"],
            "parts": []
        },
        {
            "product": "touch-controller",
            "tools": ["rs-pliers", "np-hammer"],
            "parts": []
        },
        {
            "product": "round-button",
            "tools": ["gs-driver"],
            "parts": []
        }
    ],
    "waves": [
        [
            {
                "product": "yphone30",
                "qty": 100,
                "startId": 50123450
            }
        ]
    ]
}
```

The file holds a json object which contains the following fields:

- `threads` - an integer defining the amount of threads in the `WorkStealingThreadPool`.
- `tools` - an array of tool definitions, each tool contains a `tool` field defining the tool type (see sec. 3.2) and a `qty` field defining the amount of tools of that type which the warehouse will hold (see sec. 3.3).

- `plans` - an array of plan definitions, each plan definition contains:
  - `product` - a string identifying the product which can be constructed using the plan.
  - `tools` - an array of strings defining the tools needed to construct the product.
  - `parts` - an array of strings defining the products needed to complete the plan.
- `waves` - an array of wave definitions, each containing an array of orders. Each order contains:
  - `product` - the product to be manufactured.
  - `qty` - the amount of items needed.
  - `startId` - the ID of the first product to be manufactured. The second product to be manufactured should receive an ID of `startId` + 1 and so on. Each of the parts which assemble the product receive `startId` + 1.

  Note that each wave can include *multiple orders* for different products. The above example contains one wave with one order.

The above input file will spawn 100 manufacturing tasks for `yphone30` products. When a processor begins processing a `yphone30` manufacturing task, it will spawn a `5'-screen` manufacturing task and a `round-button` manufacturing task. The `round-button` manufacturing task requires no additional parts, and can be immediatly resolved by a processor, while the `5'-screen` manufacturing task will again spawn two manufacturing tasks fro `glass` and `touch-controller`. Once all the tasks and sub-tasks spawned by the `yphone30` manufacturing task are resolved, the `yphone30` manufacturing task may resolve and a new `yphone30` can be added to the output `ConcurrentLinkedQueue`.
The startId of the first `yphone30` is 50123450, the startIds of the its `5'-screen` and `round-button` are both 50123451. The startId of the `5'-screen`'s `glass` and `touch-controller` is 50123452. The startId of the i-th `yphone30` is 50123450+i.

4 Format and Submission Instructions

4.1 Packages

In order for your implementation to be properly testable, you must conform to the following package structure.

```
package structure
└bgu
  └spl
    └a2
      ├─Deferred
      ├─Processor
      ├─Task
      ├─VersionMonitor
      ├─WorkStealingThreadPool
      ├sim
      │ ├─Warehouse
      │ ├─Product
      │ ├─Simulator
      │ ├tools
      │ │ ├─Tool
      │ │ ├─RandomSumPliers
      │ │ ├─NextPrimeHammer
      │ │ └─GcdScrewDriver
      │ ├tasks
      │ └conf
      │   └─ManufactoringPlan
      └test
        └─MergeSort
```

You may add classes as you see fit, but they should not be inter-dependent. Your application should run correctly if we replace any of the above with our own implementation.
All the classes extending task in Part2 should be placed in bgu.spl.a2.sim.tasks.

Important: If you do not conform to the above structure, automated tests may fail, resulting in a reduction of your assignment grade.

4.2 Building the Application: Maven

In this assignment you are going to use maven as your build tool. Maven is the de-facto java build tool. In fact, maven is much more than a simple build tool, it described by its authors as a software project management and comprehension tool. You should read a little about how to use it, you can find a short tutorial here: https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html.
IDEs such as eclipse, netbeans or intellij all have native support for maven and may simplify interaction with it - but you should learn yourself how.
Use the following command to install maven in the lab:
/home/studies/course/spl161/java/install-maven
Please do not attempt to install it in your lab any other way.

In this assignment you required to work with java 8 (1.8), You will need to have JDK8 installed and to configure your maven build to use java 8 - you should findout how to acheive that yourself.

4.3 Deadlocks, Waiting, Liveness and Completion

Deadlocks You should identify possible deadlock scenarios and prevent them from happening.
Waiting You should understand where wait cases may happen in your program, and how you have solved these issues.
Liveness Locking and Synchronization are required for your program to work properly. Make sure not to lock too much so that you don't damage the liveness of the program. Remember, the faster your program completes its tasks, the better.
Completion As in every multi-threaded design, special attention must be given to the shut down of the system. When the CLI service receives a "shutdown" command, the program needs to terminate. This needs to be done gracefully, not abruptly.

4.4 Documentation and Coding Style

There are many programming practices that **must** be followed when writing any piece of code.

- Follow Java Programming Style Guidelines. A part of your grade will be checking if you have followed these mandatory guidelines or not. It is important to understand that programming is a team oriented job. If your code is hard to read, then you are failing to be a productive part of any team, in academics research groups, or at work. For this, you must follow these guidelines which make your code easier to read and understand by your fellow programmers as well as your graders.
- Do not be afraid to use long variable and method names as long as they increase clarity.
- Avoid code repetition - In case where you see yourself writing same code block in two different places, it means this code must be put in a private function, so both these places may use it.
- Full documentation of the different classes and their public and protected methods, by follow- ing Javadoc style of documentation. You may, if you wish, also document your local methods but they are not mandatory.
- Add comments for code blocks where understanding them without, may be difficult.
- Your files must not include commented out code. This is garbage. So once you finish coding, make sure to clean your code.
- Long functions are frowned upon (30+ lines). Long functions mean that your function is doing too many tasks, which means you can move these tasks to their own place in private functions, and use them as appropriate. This is an important step toward increased readability of your code. This is to be done even in cases where the code is used once.
- Magic numbers are bad! Why? Your application must not have numbers throughout the code that need deciphering.
- Do not send collection of items to constructors. They can reveal internal implementation. Instead, create a method for the object which adds one item to the object. Example: instead of adding a collection of items to the database, we define the function addProduct to add a single product.

Your implementation must follow these steps in order to keep the code ordered, clean, and easy to read.

4.5 Submission Instructions

- Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than two.

- You must submit one .tar.gz file with all your code. The file should be named ”assignment2.tar.gz”.
  Note: We require you to use a .tar.gz file. Files such as .rar, .zip, .bz, or anything else which is not a .tar.gz file will not be accepted and your grade will suffer.
- The submitted zip should contain the src folder and the pom.xml file only! no other files are needed.
- Extension requests are to be sent to majeek at cs. Your request email must include the following information:
  - Your name and your partner’s name.
  - Your id and your partner’s id.
  - Explanation regarding the reason of the extension request.
  - Official certification. Request without a compelling reason will not be accepted.