CSCE 311
Spring 2017
Project # 3
Assigned: March 22, 2017
Due: April 11, 2017 (see due date/time in Dropbox)

Objective: To implement RESOURCES project (chapter 9) in the OSP2 simulator to further your understanding of resource management. You are required to implement the deadlock detection algorithm (see section 7.6 of the Silbershatz text) for managing resources to detect deadlocks and recover when deadlocks are discovered.

Required to turn in: Follow all directions regarding hand-in procedures. Points will be deducted if you do not follow directions precisely. You must submit an electronic copy of the *.java files that comprise your solution (or your best try) via dropbox. Late assignments will not be accepted. You must document your code and provide a one-page explanation of how you accomplished the assignment (or what you have currently and why you could not complete). You should describe your use, creation, and manipulation of data structures to accomplish the assignment.

Building and executing the simulation
Download the archive file containing the files for this project.
For unix or linix or Mac: If you are using a unix or linix machine then you will probably want to download the tar file: Resources.tar. Download the archive and extract the files using the command **tar -xvf Resources.tar**

For a windows box: If you are using a windows box, then you will probably be happier with a zipped folder: **Resources.zip**. Download the archive and then extract the files by right-clicking on the file and selecting the "extract all…" option from the popup menu.

You should have extracted the following files:

```
Resources/Demo.jar
Resources/Makefile
Resources/Misc
Resources/OSP.jar
Resources/ResourceCB.java
Resources/ResourceTable.java
Resources/RRB.java
Resources/Misc/params.osp
Resources/Misc/wgui.rdl
```

As per the discussion in the OSP2 text, `Makefile` is for use in unix and linix systems. The demo file `Demo.jar` is a compiled executable. The only files you should have to modify are `ResourceCB.java`, `ResourceTable.java`, and `RRB.java`. Modifying the other files will probably "break" OSP2.

Compile the program using the appropriate command for your environment (unix/linix/windows).

(unix) `javac -g -classpath .:OSP.jar: -d . *.java`

(windows) `javac -g -classpath .;OSP.jar; -d . *.java`

This will create an executable called `OSP`. Run the simulator with the following command:

```
java -classpath .;OSP.jar osp.OSP
```

Resource Management in OSP

There are three classes involved in resource management that have methods that you are required to implement:

ResourceTable – (section 9.4 page 156) this is by far the easiest class. All you need to do is implement the constructor. For the purposes of this project, all you need do is call super().

RRB – (section 9.5 pages157-160) this is a little more involved than the ResourceTable class, but not much more. You must implement:

1. **RRB() -** Simply call super() with the same arguments as RRB(). See page 158 of OSP2 text.
2. **do_grant()** – this method simply takes care of bookkeeping. In particular, you should update the values of current resources and available resources to reflect the granting of a resource request. How do you do this?
    a) use getResource()(page 159) to retrieve the ResourceCB object. This object is the resource for which the request was issued.
    b) calculate the new value of Available for this resource by subtracting the quantity requested (use getQuantity()) from the amount available (use getAvailable())
    c) update Available to the amount you calculated in step b using setAvailable().
    d) calculate the new value of Allocated similarly to what you did in step b using getAllocated() and getQuantity(). See page 159 for the arguments to these methods.
    e) update Allocated to the amount you calculated in step d using setAllocated().
    f) use setStatus() to set the status of the RRB to Granted.
    g) finally use notifyThreads() to resume the thread that was waiting on this RRB.

ResourceCB – (section 9.6 pages 160-166) Ok, this is the tough one. Start by stocking up on your favorite caffeinated beverage. You will need it. Actually there are only 4 methods (not including the constructor and init()) that you are required to implement and one of them will be trivial and the other three won't be so bad. However, since we are doing deadlock detection, you will have to also implement the detection algorithm. Carefully read the description of the ResourceCB class starting on page 160. Pay particular attention to the discussion in the 2nd, 3rd, and 4th paragraphs on page 160 that

describe the need for a data structure like an array of hash tables instead of the 2D tables that we used for the Banker's algorithm in class. Since you may not have used hash tables before I will tell you how to create and access this data structure. Use the declaration:

```
private static Hashtable<ThreadCB,RRB> threadRRBTable=new Hashtable<ThreadCB,RRB>();
```

to create the hashtable for this project. You will also need to have a null rrb object as a place holder. Use the declaration:

```
private static RRB nullRRB = new RRB(null,null,0);
```

The ResourceCB class in OSP contains a constructor, init() as well as four methods that you are required to implement:

`ResourceCB()` – To implement this constructor, simply call super() with the same argument that ResourceCB is invoked with.

`init()` – This method is called by the simulator before the simulation begins. You can use this method to initialize any data structures that you have defined. You can leave the body of this method empty if you do not require data structure initialization.

`do_acquire()` – Carefully read the description of this method on pages 161-162. This method is called by `OSP2` to simulate a request for resources by a thread. The first thing you will need to do is to get the thread that is making the request. This is the currently executing thread. You can get it by retrieving the current task through the PTBR and then retrieving the task's current thread:

The page table base register (PTBR) points to the page table of the current thread. We can use this information to figure out which thread is running. The PTBR is contained in the memory management unit (MMU)

a. The call MMU.getPTBR() returns the page table of the current thread.
   Note: if no thread is running then this call will return null.
b. Applying the method getTask() to the page table returns the task that owns the thread
c. Applying the method getCurrentThread() to the task returns the current thread
d. Altogether: MMU.getPTBR().getTask().getCurrentThread()
e. If you attempt the statement in step d) you had better put it in a try-catch construct to catch the potential NullPointerException

Next, verify that the quantity of resource being acquired + the amount allocated do not exceed the total amount in the system (use methods getAllocated() (page 164)& getTotal() (page 163)). If this test fails then return null.

Recall that we are using a hashtable to keep track of resource requests for threads. If the requesting thread is not currently in the hashtable, then add it to the hashtable. Check if it is in the hashtable using the method `threadRRBTable.containsKey(t)` where t is the requesting thread. Place it in the hashtable if necessary using the method `threadRRBTable.put(t,nullRRB)` where t is the requesting thread and `nullRRB` is the null RRB we declared in the beginning of this class. Next create an RRB object using the RRB constructor and the arguments: requesting thread, the current ResourceCB object, and `quantity`, the argument to the `do_acquire()`.

Next, check if the requested amount is available. If so, then grant the request by invoking `grant()` and exiting. Otherwise, check the status of the thread. If it is NOT ThreadWaiting then change the RRB object status to `Suspended` and suspend the thread with the RRB object as the argument to the suspend method. Finally return the RRB object.

`do_release()` – This method is called by `OSP2` to simulate a thread relinquishing resource. The formal argument to this method is the quantity of instances of that resource to be released. Again, you must first get the current thread (same as for do_acquire()). Then retrieve the quantity of the resource type in question that is currently allocated to this thread. Make sure that you do not try to release more than what is currently allocated to this thread. Update the amount left allocated to the thread after subtracting amount being released using setAllocated(). Update the new amount of this resource type available using setAvailable().
You should also check to see if any suspended RRBs can now be satisfied. You can get these out of the hashtable using the following code:

```
Collection c = threadRRBTable.values();
```

Then create an iterator and iterate though this collection checking each RRB in turn to see if the request can now be granted. Simply check if the requested amount is currently available. If it can be granted, then set the status of the RRB to Granted using setStatus(). Verify that the thread status is not ThreadKill and grant the request by invoking grant(). Update the hashtable entry for this thread and RRB via `threadRRBTable.put(t,nullRRB)`, where t is the thread.

`do_deadlockDetection()` – since we are focusing on deadlock detection in this project you will need to identify those threads that are in a deadlock and then perform deadlock recovery to break the deadlock. There are three main goals that should be accomplished in this method:

1) identify the threads involved in deadlocks
2) deadlock recovery
3) return the list of deadlocked threads

Let's start first by discussing deadlock recovery (step 2) since this will affect how you implement the functionality for identifying those threads involved in the deadlock (step 1). As per the discussion in the OSP manual, deadlock recovery is done by killing some or all of the threads involved in a deadlock. Be aware that OSP will check each time you kill a thread to see if it was involved in a deadlock. If you kill a thread and the deadlock is resolved, OSP will complain if you continue killing threads. Consequently, after you kill a deadlocked thread, you should again run your deadlock identification algorithm (step 1 from above) to see if any threads are still deadlocked. This suggests two things: 1) step 1 should be a separate method so that it is easy to call from different places in this method and 2) that deadlock recovery (step 2) be set up as a loop where if there are deadlocked threads, you kill a thread, then identify threads that are still deadlocked (step 1) and repeat until there are no more deadlocked threads.

Step 3 simply entails you returning original vector of deadlocked threads that you found in Step 1.

Identifying threads involved in deadlock

So how do you identify the threads involved in a deadlock (step 1)? See section 7.6 of the Silberschatz book for the outlines of the deadlock detection algorithm. Notice that it is very similar to the Banker's Algorithm. Start by copying the available amount of each resource type into a `Work[]` array BEFORE you start making changes to these values.

Algorithm for identifying deadlocked threads:

As in the bankers algorithm you loop through the current threads looking for an order in which you could execute the threads. When you are done, if there are any threads that could not be executed, then those are the deadlocked threads. Unlike the Banker's Algorithm where we only cared whether we could execute all of the threads or not, here we need to keep track of which threads we are unable to execute. We need a data structure analogous to `Finish[]` array in section 7.6. You have been keeping track of current threads with the hashtable threadRRBTable. Unfortunately, a hashtable is not a practical data structure for iteration. The solution is to create an enumeration of threads with the threads extracted from the hashtable. This can be accomplished by declaring an Enumeration of keys from the hashtable

Enumeration keys = threadRRBTable.keys()

Then loop through the enumeration looking for a thread to see if it can finish. Note: If the rrb for a thread has quantity of zero, then the thread CAN NOT be in a deadlock. These threads should be marked as "finished". Threads whose request quantity is less than or equal to available (`Work[]`)should also be marked as "finished".

Finally, the main body of the algorithm consists of looping through the enumeration of threads looking for a thread such that REQUEST is <= `Work[]`.
You need to keep looking through the enumeration of threads until either the enumeration is empty, in which case there is no deadlock or you can not find any other threads from which REQUEST <= `Work[]` in which case there is a deadlock and those threads with REQUEST > `Work[]` are deadlocked. You should then create a vector containing those threads that are participating in the deadlock. Return this vector when you exit this method. In any case, don't forget to restore the original values of the available amount of each resource type before returning from this method.

`do_giveupResources()` – This method is called by `OSP2` to cause all of the resources held by the thread specified by the formal argument `thread` to be released. After releasing all of the resources, you should attempt to satisfy pending requests as was done for `do_release()`. Try to satisfy pending requests for all resource types that were released for the process. Grant those that can be granted. Simply check if the requested amount is currently available. Obviously, you should not try to satisfy a request by the thread for which you have just given up resources since it is either being `killed` or `terminated`. One way to avoid this is to remove the thread from the hashtable

AFTER you have released all of its resources and *BEFORE* your start looking for pending RRBs. **Suggestion:** since both `do_release()` and `do_giveupResources()` try to satisfy pending requests after freeing up resources, you should probably implement that functionality as a separate method and have them call that method rather than duplicating that code.

How do I get an A on this assignment?

This is very simple to do. Implement the methods as outlined above using the banker's algorithm to avoid deadlocks and hand in everything specified in the "Required to turn in" section on time. Your solution must deal with resource allocation and deallocation while avoiding deadlocks. It should also be well documented so the grader can understand your implementation decisions. For an example of how a correct solution might look, run `Demo.jar` selecting deadlock avoidance (as opposed to deadlock detection). Your solution should complete successfully with no warnings or aborts of the OSP2 simulator.

Turning in Your Assignment via dropbox

When you are satisfied that your implementation of the RESOURCES project works, use dropbox to submit your project. Use dropbox to submit the following files:

1. `ResourceCB.java` (deadlock avoidance using the banker's algorithm)
2. `ResourceTable.java`
3. `RRB.java`
4. One page write-up of how you accomplished the assignment (or what you have currently and why you could not complete).