CSCE 311 Operating Systems
Fall 2020
Programming Project 3

Assigned: Thursday, October 29nd, 2020
Due: Sunday, November 15th, 2020 at midnight
Deliverables:

- Your ResourceCB.java, RRB.java, and ResourceTable.java
- One-page explanation of how you accomplished the assignment (or what you have currently and why you could not complete the task). Make sure that you comment on the data structures that you used and how you used them.

Objective: To implement a deadlock avoidance algorithm in OSP2. You will need to implement all of the functions in the ResourceCB.java (Resource Control Block), RRB.java (Resource Request Block) and the ResourceTable.java.

Incorporating your implementation and running the simulator.

1. Download the attachment to the Project 3 assignment on Dropbox (Project3.zip).
2. Move Project3.zip into your "CSCE311Projects" folder
3. Unzip the file you should see the following files:
    a. Resources/Demo.jar
    b. Resources /Makefile
    c. Resources /Misc
    d. Resources /OSP.jar
    e. Resources/ResourceCB.java
    f. Resources /RRB.java
    g. Resources/ResourceTable.java
    h. Resources /Misc/params.osp
    i. Resources /Misc/wgui.rdl
4. Demo.jar is a working version of OSP2 using the Banker's algorithm. To run this version of the simulator:
    a. Linux/Unix/Mac: java -classpath .:Demo.jar osp.OSP
    b. Windows: java -classpath .;Demo.jar osp.OSP
    c. PowerShell Windows: java -classpath ./Demo.jar osp.OSP
5. The only three files that you should modify are the ResourceCB.java, RRB.java and ResourceTable.java files. Do not modify any of the other files.

    After modifying the code in these files, you can recompile your code using the following commands:

    Linux/Unix/Mac: javac -g -classpath .:OSP.jar: -d . *.java
    Windows: javac -g -classpath .;OSP.jar; -d . *.java

    This will create an executable called OSP. You can then run the simulator with your new code using the following:

    Linux/Unix/Mac: java -classpath .:OSP.jar osp.OSP

Windows: java -classpath .;OSP.jar osp.OSP

Background

We will be implementing the Banker's algorithm in this assignment to avoid deadlock. Deadlock occurs in systems where resources (such as files, printers, disks, buffers etc.) are shared and many processes are competing to acquire them at once.

The Resource Control Block

The bulk of the work will be in the Resource Control Block (RCB). This is where all of the information about the available resources is maintained and therefore requests for resources must pass through it. Resources are divided into resource types where each resource type can have several instances. Each resource type is represented by a distinct resource control block. At some point, a process may come and request a certain amount of instances of a resource. The request will be granted or denied by the RCB based on the current allocation state and the deadlock handling method.

The Resource Request Block

The RRB class represents the Resource Request Block. A RRB contains information about one outstanding request for one particular resource type issued by a particular thread. When a thread issues a request that cannot be granted then the thread is suspended on the RRB associated with the request. When the resources become available the notifyThreads() operation will let the RRB know to wake up the thread and grant its request.

The Resource Table

An array of RCBs. It lists all available resource types in the system. In OSP2 all creation of resource types are created at the beginning of the simulation (no new resources are added/deleted afterwards). The total number of instances of resources also stay constant. However, the available amount of resource instances will change based on processes acquiring and releasing them.

Resource Types

These are identified by a resource ID (a number between 0 and the resource table size – a call to the static function getSize() will return the size).

Modules

ResourceTable.java

Simple, just a constructor.

public ResourceTable()

call super()

Other built in functions you may use in your other classes:

Public static final ResourceCB getResourceCB(int resourceID)

In a loop, this would let you walk through the RCB of all resource types in the system.

Public static final void getSize(int size)

Returns the size of the resource table (number of resource types available in the system)

RRB.java

RRBs keep track of:

- An ID (accessed via getID())
- The thread that requested the resource (accessed via getThread())
- The resource type involved in the request (getResource())
- The amount of that resource type available (getQuantity())
- The status of an RRB (getStatus() and setStatus(…))
    - Denied
        - This might happen if the requesting thread wants more than what is available
    - Suspended
        - This might happen if the request is going to be granted, just not yet.
    - Granted

Methods for you to complete:

public RRB(….)

Just call the super and pass the same arguments that were passed to this constructor

public do_grant()

Note: This does not actually make the decision as to whether or not the resource is granted. This method just does the bookkeeping only. It will:

- Decrement the number of available instances of the requested resource by the requested amount
    - getAvailable() returns the available amount of the resource
    - setAvailable(int) will set the available
- Increments the allotted instances of the resource by the requested amount
    - getAllocated(ThreadCB thread) returns the number of allocated instances of the resource type for the thread passed in
    - setAllocated(ThreadCB, int) sets the number of allocated instance of this resource for the thread
- You should also set the status of the RRB to Granted.
- Call notifyThreads()

Other built in functions you may use in this class:

Int getStatus() – Returns the status of the RRB: Denied, Suspended, Granted
Void setStatus(int) – sets the status of the RRB to the status passed in
Int getID() – returns the ID of the RRB
Int getQuantity() – returns the quantity of the resource requested by the thread that issued the request
ThreadCB getThread() – returns the thread that issued the request

ResourceCB.java

This is where the deadlock avoidance will happen. Note that when we do this in class, by hand, we assume that we know how many threads are in the system and we run it as if those are the only threads ever coming in. In a real-life system threads will be coming and going so we will have to account for that. One nice aspect about OSP2 is that the number of resource types is fixed. This simplifies things, however, we should not think to use a 2d array like we do in class to model this algorithm. Instead, a better alternative is to use a HashTable of Threads and RRBs. In my solution, I use a Hashtable<ThreadCB, RRB>

Methods for you to complete:

Public ResourceCB(int)

Just call super(qty).

Public static void init() Same as when we did the ThreadCB -> you will likely just use this method to initialize you data structures.

Public RRB do_acquire(int quant)
This is invoked by an OSP 2 thread on a given resource type (represented by a ResourceCB object) in order to obtain "quant" number of instances of that resource type. You will need to:

1. Determine what thread issued the request (use the PTBR like before).
2. Determine if what the thread is asking for goes over the amount that the resource has remaining (use getTotal()). You also want to add the total that the process currently has allocated (getAllocated(currentThread))
3. Create an RRB object with the current thread, the current resource and the quantity that was sent into the do_acquire function.
4. Next you should run the bankers algorithm to see if the system will be in a safe state after the request has gone through.
    - I would suggest another function here that either returns the RRB that can be granted OR true or false on whether it can be granted or not. You will want to do the following:
        - Check to see if the thread is asking more than can be allocated
            - Hint: To get the total amount of the resource allocated to the thread use getAllocated(currThread), the quantity requested is passed in and the max that the thread can request of that resource is getMaxClaim(currThread)
        - If it is asking for more than the max then set the RRB's status to Denied
        - Next, check to see if the quantity requested is greater than the total available units for the resource (getAvailable())
        - If true:
            - Check to see if the thread is not already suspended (in ThreadWaiting status). If yes, then call the suspend method for the thread and pass in the RRB
            - Set the status of the RRB to Suspended.
        - If false:
            - Loop through the resource table and grab the current available for each of the resources (we did this in the do_deadlock detection)
            - Pretend like the request of the resource has been allocated
                - One way you can do this: Loop through the resources again and allocate back the available what the thread that we are hypothetically granting
            - Loop through the hashtable of RRBs and Threads and mark the ones that you will consider in the rest of bankers algorithm. (We did this using a local hash table using a Boolean as the value for the thread keys)
            - Note: you should consider all threads that have any resources allocated to it
            - Run a while loop to check for deadlock. You can use the code that we did in the deadlock detection. However, you will need to modify it to check for the max of the threads for each of the resources (just like we did on paper with the bankers alg).

        - If the Max – the allocated of the process can be satisfied then you can hypothetically grant it (just add the allocated back to the available)
    - If at the end of your loop you have threads that are in deadlock (that cannot hypothetically run) then you should either return false (meaning that the thread should not run) or grant the RRB or return the RRB (however you set up the function).
5. If you are in a safe state then you should call the grant function in the RRB (use RRB from step 3 and call .grant().
6. If you are not in a safe state then you should suspend the thread and set the RRB status to Suspended as well.
7. Make sure that you have added this thread, RRB combo to your hashtable.
8. Whether the RRB was granted immediately or the thread was suspended, do_acquire() returns the RRB that was created in step 3.

Vector do_deadlockDetection()
I will give you the code for this. We will do it in class. However, you will not need it in your final solution (the param file that you use to simulate will be in Avoidance (not detection) mode).

void do_release(int quantity)
This method will be invoked by a thread on a given resource type in order to release quantity instances of that resource type.

1. Get the thread that is currently running (MMU.getPTBR().getTask().getCurrentThread())
2. Use the setAllocated(ThreadCB, int) function to set the number of units allocated to the thread to its current allocation (getAllocated(thread)) – quantity.
3. Use the setAvailable(int) function to set the new availability (new number of units) of the resource after the thread releases its units.
4. Now new units of the resources are available! Yay! Now it is possible that some of the other Threads that are blocked on an RRB involving that Resource can run. You will need to loop through the Threads/RRBs in your HashTable to figure out if an RRB can be granted. One possible solution:
    a. Set up an Enumeration<ThreadCB> of keys hTable.keys()
    b. Run a while loop through the Enumeration of keys
        i. Hint: you can iterate using keys.hasMoreElements() and then grab a key using keys.nextElement()
    c. For each key, you should extract the RRB associated with that Thread.
    d. Check to see if the requested amount for that RRB is now currently available and is SAFE. (Compare rrb.getQuantity() with rrb.getResource().getAvailable() and run your bankers Algorithm function)
        i. If yes then set the status of the RRB to Granted, verify that thread is not in the ThreadKill state and then grant the request by invoking grant(). Then update the HashTable entry for this thread and RRB to the thread and a null RRB.

void do_giveupResources(ThreadCB thread)
When a thread is killed, this method is called to release the resources owned by the thread.

1. Loop through the resource table (you can get the size using ResourceTable.getSize())
    a. For each resource (ResourceTable.getResourceCB(i) – assuming you use i as the iterator in your loop)

    b. If there is some units of the resource allocated to the thread then set the available of that resource to its currently available units + the resource units that were given up by the thread.
    c. Set the allocated units of that resource for that thread to 0
2. Remove the thread, RRB from the HashTable
3. Follow the same process as you did in step 4 of the do_release to figure out if other processes can run now.

Other built in functions of interest:
Int getTotal() – returns the total number of instances (free plus allocated) for the given resource type.
Int getAllocated(ThreadCB) – returns the number of allocated instance of this resource type
Void setAllocated(ThreadCB, int) – sets the number of allocated instances for this resource type
Int getAvailable() – returns the number of free instances for this resource type
Void setAvailable(int) – sets the number of free instances for this resource type
Int getMaxClaim(ThreadCB) – returns the maximal number of instance of this resource type that can ever be acquired by the given thread. You can use this in your banker's algorithm.
Int getSize() – returns the size of the resource table
ResourceCB getResourceCB(int) – returns the ResourceCB at a given index in the ResourceTable
Void notifyThreads() – resumes all threads that might be waiting on this event.
Void setStatus(int) – sets the status of the RRB to Denied, Suspended or Granted
ResourceCB getResource() – the resource for which the request was issued
Int getQuantity() – the quantity of the resource requested by the thread that issued the request.

Remember our old friends from the last assignment:
PageTable getPTBR() – returns the value of the page table base register which is either null or the page table of the currently running task (part of MMU class)
TaskCB getTask() – indicates which task owns the given page table (part of PageTable class)
ThreadCB getCurrentThread() – returns the running thread of a given Task (part of TaskCB)
Void suspend(Event) – suspends the thread on which this method is called and puts the thread on the waiting queue of the event (ThreadCB)
Void kill() – Kills the thread on which it is called and releases resources (ThreadCB)
Int getStatus() – returns the status of the thread (ThreadCB)

<u>Additional Guidelines:</u>

- Feel free to create any helper functions to get tasks done.
- Your code will need to compile and run to get credit, so you should start small and work to the final solution in stages.
- You will need to document your code well so that the grader and I will be able to clearly see what you are doing (or trying to do) in each step.
- Do not rip the solution off the internet. This is considered cheating.
- For an example of how a correct solution might look when it runs, run the demo version in the directory that was provided.
- Your solution should not have core dumps and should complete the simulation successfully with no warnings or aborts of the OSP2 simulator.