CSCE 311 Operating Systems
Fall 2020
Programming Project 2

Assigned: Tuesday, September 22nd, 2020
Due: Sunday, October 4th, 2020 at midnight
Deliverables:

- Your FCFS solution (i.e. ThreadCB.java and TimerInterruptHandler.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 CPU scheduling on the OSP2 simulator. You will implement the methods in module ThreadCB.java and the single method do_handleInterrupt() in TimerInterrupHandler.java to further your understanding of CPU scheduling. You are required to implement the *FCFS* scheduling algorithm.

Incorporating your implementation and running the simulator.

1. Download the attachment to the Project 2 assignment on Dropbox (Project2.zip).
2. Move Project2.zip into your "CSCE311Projects" folder
3. Unzip the file you should see the following files:
    a. Threads/Demo.jar
    b. Threads/Makefile
    c. Threads/Misc
    d. Threads/OSP.jar
    e. Threads/ThreadCB.java
    f. Threads/TimerInterruptHandler.java
    g. Threads/Misc/params.osp
    h. Threads/Misc/wgui.rdl
4. Demo.jar is a working version of OSP2 using FCFS scheduling. 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. Ubuntu on Windows: java -classpath ./Demo.jar osp.OSP
5. The only two files that you should modify are the ThreadCB.java and TimerInterruptHandler.java files. Do not modify any of the other files.

    After modifying the code in these file, 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

In order to run a FCFS scheduler, you will need two things: a ready list of some kind and the ability to context switch.

The Ready Queue

In the ThreadCB.java, you will need to keep track of the ready list. You may use any reasonable structure to do so (ArrayList, Map… etc.). OSP2 also comes with a GenericList (linked list, more in section 1.5 in OSP2 book).

Context Switching

You will need to be able to pass control of the CPU from one thread to another. To do this you will first need to preempt the current thread and then dispatch another thread.

Steps to preempt the current thread:

1. Determine which thread is the 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). To find the current thread: MMU.getPTBR().getTask().getCurrentThread();
    a. The call MMU.getPTBR() returns the page table of the current thread. *If no thread is running then this call will return null. You will probably want a try-catch to check for this.*
    b. The getTask() method returns the task that owns the thread.
    c. The getCurrentThread() method returns the current thread in that task.
2. Change the state of the current thread to either ThreadReady/ThreadWaiting/ThreadKill
3. Set the page table base register to null -> MMU.setPTBR(null)
4. Tell the task that its thread is no longer running (use getTask() to find the task associated with the thread and then use the setCurrentThread() method to set the current thread to null.

Steps to dispatch another thread:

1. Select a thread (t) to run (this will be dependent on what scheduling algorithm you are using).
2. If there are no threads to run (ready list is empty) then set the PTBR to null -> MMU.setPTBR(null)
3. If there is a thread, set the PTBR to point to the page table of the task that owns the thread to be dispatched -> MMU.setPTBR(t.getTask().getPageTable())
4. Let the task know that its thread is the current thread (t.getTask().setCurrentThread(t))

Modules

ThreadCB

public ThreadCB()

Just the constructor. You can just call the super's constructor (super())

public static void init()

This is called once at the very beginning of the simulation. You should use it to initialize your global variables and data structures (such as your ready list).

static public ThreadCB do_create(TaskCB task)

This is where you will create threads, add them to the ready list and dispatch. You will always want to call the dispatch() method before leaving this method. You will first want to check to see if the Task that was sent into the do_create is null or if the task sent in already has the max # of

threads (task.getThreadCount() == MaxThreadsPerTask) that it is allowed to spawn. If this is the case, then you will want to simply call the dispatcher (dispatch()) and return null. If this is not the case then you will follow these steps:

1. Create a new thread.
2. Set the status of the thread -> setStatus(ThreadReady)
3. Associate the task with the thread -> setTask(task)
4. Associate the thread with the task -> task.addThread(thread)
    a. If this step fails, then you will need to just call the dispatcher and return null
5. Add the thread to the ready list.
6. Call dispatch()
7. Return the newly created thread

public void do_kill()

This method destroys the current thread. You can follow these steps:

1. Get the status of the thread.
    a. If status == ThreadReady then remove it from the ready list
    b. If status == ThreadRunning then preempt it (see context switching above).
2. Set the status of the thread to ThreadKill.
3. Loop through the device table (Device.getTableSize()) to purge any IORB associated with the thread (Device.get(i).cancelPendingIO(this)).
4. Release all resources -> ResourceCB.giveupResources()
5. Remove the thread from the task (task.removeThread(this)). Then check to see if the task still has threads (getThreadCount()). If it does not then kill the task (.kill()).
6. Call dispatch()

public void do_suspend(Event event)

This method suspends/increments the ThreadWaiting status of a thread.

1. Find the current status of the thread (getStatus())
2. If status == ThreadRunning then suspend it (context switch again).
    a. Set the thread's status to ThreadWaiting
    b. Set the PTBR to null.
    c. Set the current thread of the task to null.
3. If it is already waiting then increment its waiting status. To test waiting status compare getStatus() >= ThreadWaiting.
4. Make sure that the thread is not in the ready queue.
5. Add the thread to the event queue using event.addThread(thread).
6. Call dispatch()

public void do_resume()

This method resumes/decrements the ThreadWaiting status of a thread.

1. Check the status of the thread
    a. If ThreadRunning -> call dispatch and return
    b. If ThreadWaiting -> set status to ThreadReady, add to the ready list and call dispatch()
    c. If any other status -> set status to getStatus – 1, call dispatch()

public static int do_dispatch()

This method selects a thread to dispatch. For this part you will use FCFS scheduling.

1. First check to see if there is a thread currently running. Hint: MMU.getPTBR().getTask().getCurrentThread() will only return null if there is no thread running.
2. If there is a thread running then check to see if there is another thread waiting in the ready queue.
    a. If not, then set the PTBR to null and return FAILURE -> this means that the dispatcher was not able to perform a context switch because there was nothing to switch to.
    b. If the ready queue is not empty then do a context switch (see Background section). The thread that was running should have its status set to ThreadReady and it should be put back on the ready queue. Based on FCFS the thread that we should take out of the ready list is the thread at the head of the queue. This thread should be the new thread that is running. Then return SUCCESS.

TimerInterruptHandler

```
public void do_handleInterrupt(){
        ThreadCB.dispatch();
}
```

That's it.

Additional Guidelines:

- 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.