CSCE 311
Spring 2020
Project #4

Assigned: April 7, 2020
Due: April 21, 2020

Objective

To implement DEVICES package in OSP2 in particular disk scheduling. The Devices package is described in chapter 6 of the OSP 2 manual. The java classes to be implemented in this project will deal with the creation and management of IORBs, the Device class and the DiskInterruptHandler class to support disk scheduling. see sections 10.1-10.4 of the Silberschatz textbook for related material.

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 incur a 10% per day penalty. 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. Submit you write-up via dropbox.

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

For a windows box: If you are using a windows box, then you will probably be happier with a zipped folder: **Devices.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:

```
Devices/Demo.jar
Devices/Makefile
Devices/Misc
Devices/OSP.jar
Devices/IORB.java
Devices/Device.java
Devices/DiskInterruptHandler.java
Devices/Misc/params.osp
Devices/Misc/wgui.rdl
```

As per the discussion in the OSP2 text, `Makefile` is for use in unix and linux systems. The demo file `Demo.jar` is a compiled executable. The only files you should have to modify are

IORB.java, Device.java, and DiskInterruptHandler.java. Modifying the other files will probably "break" OSP2.
Compile the program using the appropriate command for your environment (unix/linux/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`

Disk Scheduling in OSP
There are three classes involved in disk scheduling that have methods that you are required to implement:

IORB – (section 6.3 page 106-108) 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()` with the same 6 formal arguments that the constructor is invoked with.

Device– (section 6.4 pages109-114) this is a little more involved than the `IORB` class. You must implement:

1. **`Device()`–** This is the class constructor. It should call super with the same two arguments that it was invoked with. In addition, it should also create the iorbQueue. The iorbQueue is the queue that is used to hold the disk I/O requests (IORBs). This variable is already declared in the IflDevice class so you must use this variable with exactly this spelling. **Do not re-declare this variable**. My advice is that you create an instance of the GenericList() class and assign that to iorbQueue, i.e.,
   iorbQueue = new GenericList();

2. **`init()` –**As in all OSP2 projects, the `init()` method is only called once, at the beginning of the simulation to allow you to initialize any data structures or static variables that you are using that require initialization. If you don't have any extra data structures then leave this blank.

3. **`do_enqueue()`–** This method is used to place an IORB on the queue that holds the I/O requests for the disk. As the OSP manual indicates there are several things that you must do before placing the IORB on the queue:
   a) First, you must lock the page associated with the IORB. (Recall that the disk does I/O directly to memory without going through the MMU so the page cannot be moved until the I/O is finished). You can get the page associated with the IORB using the IORB method `getPage()`. Then use the `lock()` method of the PageTableEntry class to lock the page.
   b) Second, you need to increment the IORB count of the file associated with the iorb. You can access this file by invoking the `getOpenFile()` method of the IORB. You increment the count using `incrementIORBCount()` method.
   c) Third, you must set the IORB's cylinder to the cylinder that contained the block specified by the IORB using the method `setCylinder()`. This is not straightforward, since you have to calculate which cylinder holds the bock in

question. So first we have to calculate the cylinder. First, read the description of how to compute a cylinder from a block on page 111. Then follow my directions:

i. The cylinder = blocknumber/(blocksPerTrack * numberOfPlatters).
ii. You can get the blocknumber using the IORB method `getBlockNumber()`
iii. You can get the numberOfPlatters using the Disk method `getPlatters()`. Since Disk is a subclass of Device, you can access this method in this do_enqueue by using `(Disk)this.getPlatters()`.
iv. Getting blocksPerTrack requires you to compute it: First you need to know the size of a block: it is the same as the page size. The page size can be determined by first computing the number of bits in the page offset `(getVirtualAddressBits() - getPageAddressBits())` and raising 2 to the power indicated by this quantity. In other words if getVirtualAddressBits() - getPageAddressBits() = x, then the page size is 2^x. This is also the size of a block in bytes. Now you can compute the blockPerTrack by computing the bytes per track and dividing by the bytes per block. The bytes per track can be gotten by multiplying the sectors per track by the bytes per sector. Soooooo:

    bytesPerBlock = 2^(getVirtualAddressBits() - getPageAddressBits())
    bytesPerTrack = sectorsPerTrack * bytesPerSector
    blocksPerTrack = bytesPerTrack/bytesPerBlock

    Wasn't that easy? ;-)

d) Now you can enqueue the IORB. However, before you get too excited, you must check that the thread is still alive (compare thread status with ThreadKill). If the thread is dead, then return FAILURE. If the thread is not dead then enqueue the IORB on the iorbQueue using the `append()` method and return SUCCESS.

4. **`do_dequeue()`** – Fortunately, this method is very simple:
    a) Check if the iorbQueue is empty
        i. If empty return NULL.
        ii. If not empty, return the IORB at the head of the queue. Since you are a smart person and you took my advice to initialize iorbQueue to be a GenericList, you will use the method removeHead() to return the head of the queue while at the same time removing that IORB from the queue.

5. **`do_cancelPendingIO()`** – This method removes pending IORBs from the iorbQueue. It also undoes the page locking and IORB count increment that took place in `do_enqueue()`.
    a) Check if the iorbQueue is empty. If empty return.
    b) Otherwise you will need to iterate through the queue removing only those IORBs that belong to the ThreadCB specified as the argument to this method.
        i. Set up your iterator for the iorbQueue.

ii. For each IORB: compare the IORB thread (accessed via the iorb method `getThread()`) with the ThreadCB argument to this method.
iii. If they are the same then
1. unlock the page associated with the IORB. (Recall in do_enqueue() you locked the page associated with the IORB.)
2. Decrement the IORB count of the file associated with the iorb. (Recall in do_enqueue() you incremented the IORB count of the file associated with the IORB.)You can access this file by invoking the `getOpenFile()` method of the IORB. You decrement the count using `decrementIORBCount()` method.
3. Check the IORB count of the file. If it is zero, then close the file using the OpenFile method `close()`.

DiskInterruptHandler()– (section 6.5 pages114-117) The only method in this class is do_handleInterrupt(). However, it is a doozy.

1. Start by getting the information about the interrupt from the interrupt vector (section 1.4 page 10). you will want to get: 1) the iorb, and 2) the thread:
   a) As stated on page 114, the IORB is the "event" that caused the interrupt. Therefore you will use the `InterruptVector` method `getEvent()` to return the IORB. Be sure to cast it as an IORB when assigning to your iorb variable.
   b) Next use `InterruptVector` method `getThread()` to return the associated ThreadCB.
   c) Get the page using the IORB method `getPage()`.
   d) From the page get the associated frame using the PageTableEntry method `getFrame()`.
   e) Get the open-file handle from the IORB using the `getOpenFile()` method of the IORB.
   f) Continuing from step 2 on page 115 of the OPS2 manual: decrement the IORB count of the open-file handle.
   g) If the IORB count is now zero and the `closePending` flag of the open-file handle is set then you should close the file.
   h) Unlock the page.
   i) Continuing from step 5 on page 115 of the OSP2 manual: check if the thread associated with the IORB is still alive.
      i. If the thread is not dead AND the frame is null then return.
      ii. **(Set frame reference bit?)** If the thread is alive **and** the I/O is not a swap-in/swap-out you will need to set the reference bit of the frame associated with the IORB page.
         1. You can check if it is a swap I/O by checking if the IORB device ID is the `SwapDeviceID`.
         2. You can set the frame reference bit using the FrameTableEntry method `setReferenced(true)`.
      iii. **(Set frame dirty bit?)** If the I/O is a `FileRead` **and** not a swap I/O **and** the task status is not `TaskTerm` (dying/dead task) then you will also need to set the frame's dirty bit using the FrameTableEntry method

`setDirty(true)`. Note: you can tell if it is a `FileRead` by using the IORB method `getIOType()`.

iv. **(Clear frame dirty bit?)** If it is a swap I/O **and** the task is not dead/dying then you will need to clear the frame's dirty bit using the FrameTableEntry method `setDirty(false)`.

j) Continuing with step 7 on page 115: If the task is dead/dying then you will need to unreserve the frame. As the text suggests, start by verifying that the task that reserved the frame is the same as the dying/dead task. Use `getReserved()` to find out which task reserved the frame.

k) Step 8, notify the threads waiting the IORB using `notifyThreads()`.

l) Step 9, use `setBusy(false)` to idle the device. You can get the device by getting the deviceID from the IORB and using that as the argument to the `Device.get()` method.

m) Step 10, get the next IORB using the `dequeueIORB()` method. Assuming it isn't null, initiate the next I/O using this as the argument to the method `StartIO()`.

n) Step 11, invoke `dispatch()`.

Standard Tips

- This is assignment is more difficult than thread scheduling. It is advisable that you start as soon as possible.
- Seek help from me if you need it. You may discuss the project with others but you may not share code.
- Minimize the amount of code you write. It will not affect your grade (unless poorly commented), but will reduce the complexity of your solution and make it easier to debug.