CSCE 311
Fall 2013
Project #4

Assigned: Oct. 21, 2014
Due: Nov 6 2014

Objective

To implement virtual memory management in the OSP2 simulator. Virtual memory management is described in chapter 5 of the OSP 2 manual. The algorithms to be implemented in this project will deal with paging and page replacement in memory management, see chapters 8 and 9 of the Silberschatz textbook for related material. In this first part, you are required to implement a FIFO global page replacement algorithm.

Submission requirements

- An electronic submission of with the FIFO page replacement algorithm.
- A one-page report of what you have submitted including explanations of how you implemented the algorithm (not the algorithm itself) with a focus on how you used data structures. The report should also include a comparison between this algorithms and the provided Demo.jar solution in terms of how well these algorithm compare to the algorithm used in Demo.jar.

Follow all directions for hand-in procedures.

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: Resources.tar. Download the archive and extract the files using the command **tar –xvf Memory.tar**

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

```
Memory/Demo.jar
Memory/Makefile
Memory/Misc
Memory/OSP.jar
Memory/FrameTableEntry.java
Memory/MMU.java
Memory/PageFaultHandler.java
Memory/PageTable.java
Memory/PageTableEntry.java
Memory/Misc/params.osp
```

`Memory/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 `FrameTableEntry.java`, `MMU.java`, `PageFaultHandler.java`, `PageTable.java` and `PageTableEntry.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
```

Overview of memory management in OSP

The OSP simulation uses virtual memory with paging. This is discussed in detail in section 5.2 (pages 75-83) of the OSP2 manual. Briefly, there are two types of address space in this scheme: the *virtual address space* of the process and the *real address space* of the entire system. The virtual address space of a process is stored using the task's page table, while the real address space is the physical memory. Both address spaces are divided into equally sized units called pages and page frames respectively.

All memory addresses used in memory references have to be converted to real addresses (address translation). This address translation is done in OSP by using the virtual address to index the task's page table. If the page is in memory and valid, ten you need only set the appropriate referenced and dirty bits of the page. If the page is not in memory, this memory reference generates a *page fault*. The page fault brings the desired page into physical memory from a secondary storage device. If there are no free page frames, one will have to be chosen for replacement by the *page replacement algorithm*, which is what will be implemented in the assignment.

The classes that you will implement as part of this project are:

FrameTableEntry (section 5.3)
PageTableEntry (section 5.4)
PageTable (section 5.5)
MMU (section 5.6)
PageFaultHandler (section 5.7)

FrameTableEntry

Of these, FrameTableEntry is the simplest to implement as all that is required of you is to implement the constructor. Simply call `super(frameID)` from within the constructor.

PageTableEntry

For the class PagetableEntry, you must implement two methods:

`do_lock(IORB iorb)` and `do_unlock()`.

`do_lock(IORB iorb)` The goal of the do_lock method is to increment the lock count of the frame holding a page that is to be locked into memory. The point of this is to ensure that the page is not swapped out until it has been unlocked. One complication that may arise is the page to be locked is not currently in a frame. You can determine this by using the method `isValid()`. This tests the validity bit of a page. If the page is invalid, then you must load the page into memory by initiating a pagefault. This can be done by calling the static method `handlePageFault()`.
A further complication can be caused when two threads from the same task try to lock the same page before it has been loaded into memory. Only one of these should cause a pagefault (both should increment the lock count). See the discussion at the bottom of page 87 in the OSP2 manual.

`do_unlock()` The goal of the do_unlock method is to decrement the lock count of the frame holding a page. Use the method decrementLockCount() of the class FrameTableEntry to accomplish this.

PageTable

`PageTable(TaskCB ownerTask)`
To implement the constructor, start by calling `super(ownerTask)`. Next initialize the variable `pages` to an array of `PageTableEntry` of size 2^x. 2^x can be inferred by calling `MMU.getPageAddressBits()`. It will return `x`, the number of address bits. If the number of page address bits is `x` then the size of the array is 2^x. Note that each page must be initialized with a PageTableEntry object using the constructor of that class.

<u>Implementing memory_init()</u>
This function is called once (similar to cpu_init() in module CPU). It initializes any global variables. A structure used to record the access times of frames could be initialized here for example.

`do_deallocateMemory()`
This method is invoked to unset the flags for frames allocated to the task on the PageTable object. In particular it frees up the frames that are occupied by pages belonging to the task using the method setPage() to nullify the page field. Notice from section 5.5 in the oSP2 manual, it also invokes setDirty() and setReferenced() to clear the dirty bit and unset the reference bit. This is usually done when a task is terminating.

MMU

This class represents the memory management unit. Normally this is a hardware component. However, we must simulate this in OSP2. There are two methods that you must implement: init() and do_refer().

`init()`
As in all OSP projects, the init() method is only called once, at the beginning of the simulation to allow you to initialize any data structures that you are using that require initialization. IN the

case of this project, you should use this method to initialize the FrameTable. The size of the FrameTable can be gotten by calling MMU.getFrameTableSize(). You should loop through the FrameTable and initialize each Frame entry. When the simulator starts, the frame entries in the table are null-objects. You should set each entry to a real frame object using the constructor FrameTableEntry() to create a new object for each entry. Use the setFrame() method to the entry. As mentioned on page 92 of the OSP2 manual, if you have declared private static variables in other classes in the MEMORY package, you could define an init() method in those classes and invoke those init() methods from the init() method in the MMU class.

`do_refer(int memoryAddress, int referenceType, ThreadCB thread)`
This function is used by OSP to simulate memory accesses. For each memory access, do_refer() is called. This function performs a sequence of operations which typically take place in hardware:

- Convert logical (virtual) address into real page address: use the virtual address to obtain a page id which is used to index the task's page table. You can do this by dividing the virtual address (`memoryAddress`) by 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.
- Check the page table's entry using the page id: if the valid bit in the process's page table is true (use the method isValid()). If so then set the appropriate reference bits according and dirty bits according to the type of reference and quit. If the page is invalid there are two possibilities:
    1. Some other thread from the same task has already caused a pagefault and the page is being loaded.
    2. No other thread has caused a pagefault on the invalid page.

In the first case, suspend the thread on the page and wait until the page is loaded. When it is loaded and become valid, the method should set the reference and dirty bits appropriately according to the type of reference. You suspend the thread using the Thread method suspend(). This method requires an event argument. Use the page object as the even argument. As mentioned in the OSP2 manual, the simulator might decide to kill the waiting thread before the page is loaded. If this happens do not modify the dirty and reference bits. So be careful use the ThreadCB method getStatus() to verify that the thread does not have ThreadKill status before attempting to modify the diry and reference bits.

In the second, this thread must initiate a pagefault. Read the section starting on page 93 to see how to initiate a pagefault. Finally, do_refer() should return the referenced page.

PageFaultHandler

You are only required to implement one method in this class:
`do_handlePageFault(ThreadCB thread, int referenceType, PageTableEntry page)`
This routine puts the requested page into memory, if the page is not already in memory. If free page frames are available, one is selected. If a frame is locked (has a non-zero lock_count), whether the frame is free or being used, it cannot be selected for loading the new page.

When a free page frame is unavailable, a page frame that is already in use by a task must be selected using the page replacement algorithm. If it is not possible to find a page frame that isn't locked or reserved then you should return `NotEnoughMemory`. If a free frame is found, then update the page's frame attribute and perform a swap-in operation to load the page. If the page frame holds a clean page, then free the frame and swap-in the new page. If page frames holds a dirty page, then you must swap-out the dirty page, free the frame, and then swap-in the new page. Read pages 96-98 for more details.

Overview of FIFO Page Replacement Algorithm

When all frames are occupied, FIFO selects the frame with the oldest page for replacement. Locked page frames cannot be selected regardless of the algorithm used. A separate data structure could be used to store the reference times stamp for each page frame (FRAME) in the frame table, or a data structure could be attached to each FRAME structure: implementation is up to the programmer. Free frames should have a zero reference count.

No page replacement algorithm should be run unless there are no free frames available. These algorithms are only run when a page must be selected for replacement.

Tips

- This is assignment is more difficult than thread scheduling. It is advisable that you start as soon as possible.
- Seek help if you need it (within certain guidelines as stated below).
- 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.
- Once the first algorithm is working, it is advisable copy and modify it to implement the other algorithms. The algorithms are very similar.