

CSCE 311
Spring 2003
Project # 4
Assigned: April 2, 2003
Due: April 16, 2003 (2:30pm)

Objective

To implement the memory module of the OSP simulator. The algorithms to be implemented in this module will deal with paging and page replacement in memory management, see chapters 8 and 9 for related material. You are required to implement two global page replacement algorithms in this order:

1. First In First Out (FIFO) page replacement
2. Least Frequently Used (LFU) page replacement with a second chance for dirty pages

Submission requirements

A hard copy of `memory.c` for each algorithm implemented.

A hard copy of one `simulation.run` for each algorithm implemented (each must be produced by "`OSP -d simulation.parms`").

An electronic submission of LFU with second chance for dirty pages. If this algorithm was not finished, an electronic submission of FIFO is required.

A one-page report of what you have submitted including explanations of how you implemented the algorithms (not the algorithms themselves) with a focus on how you used data structures. The report should also include comparisons among the performance of each implemented algorithm, and a comparison between these algorithms and the provided Least Recently Used (LRU) solution in terms of how well these algorithms compare to LRU.

Follow all directions for hand-in procedures. Late assignments are unacceptable.

Building and executing the simulation

Copy these files into your chosen directory:

```
~osp/project_4_002.sun4/Makefile  
~osp/project_4_002.sun4/memory.c  
~osp/project_4_002.sun4/dialog.c  
~osp/project_4_002.sun4/simulation.parms
```

A working (LRU implementation) version of OSP is in this directory:

```
~osp/project_4_002.sun4/OSP.demo
```

This version can be run by using the full path from your directory:

```
~osp/project_4_002.sun4/OSP.demo -d simulation.parms
```

Your version of the simulation is built and run in exactly the same manner as previous assignments.

Overview of memory management in OSP

The OSP simulation uses virtual memory with paging. 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 process'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 process's page table. If the page is in memory, the page table's entry contains the id of the page frame which contains the page being referenced. 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. Details of memory management in OSP can be found in Chapter 1.6 on Memory management on page 22 of the OSP manual. Data structures of interest include PAGE_ENTRY, PAGE_TBL, FRAME, and PCB.

The following functions (used in prepaging for CPU scheduling) are not to be implemented:

```
prepage(pcb)
start_cost(pcb)
```

The following functions must be implemented:

```
memory_init()
deallocate(pcb)
refer(logic_addr, action)
get_page(pcb, page_id)
lock_page(iorb)
unlock_page(iorb)
```

Implementing memory_init()

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.

Implementing deallocate(pcb)

This function frees the memory held by the process represented by the `pcb` argument.

Implementing refer(logic addr, action)

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

Convert logical (virtual) address into real address: use the virtual address to obtain a page id which is used to index the process's page table.

Check the page table's entry using the page id: if the valid bit in the process's page table is false OR the valid bit is true and the page frame is assigned to another process, the page is not in memory

If the page is not in memory cause a page fault:

Set the fields of `Int_Vector` to indicate the cause as a page fault and the page id and pcb causing the page fault.

Invoke `gen_int_handler()`. Note: this may change the currently scheduled process (`PTBR->pcb`), so saving this before the interrupt may be a good idea.

The page should be in memory by this stage: if the action is store (write), set the dirty flag of the page frame.

Implementing `get_page(pcb, page_id)`

When a page fault occurs, the page fault handler calls `get_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 process must be selected using the page replacement algorithm. When a page frame has been selected for replacement, the page frame's contents are written to the storage device if it is dirty. The page table of the process which was using this frame before is then updated by marking the entry invalid.

The replacement page is read from disk into the selected page frame. The frame table entry is updated (`free, dirty, pcb, page_id`) and the page table of the process causing the page fault is updated (`frame_id` and `valid`).

Implementing `lock_page(iorb)` and `unlock_page(iorb)`

These functions are called before I/O begins on a page of memory and must be locked (`lock_page(iorb)`) and when the I/O operation has ended (`unlock_page(iorb)`). This prevents the page being swapped out during the I/O operation (see interlocking I/O in the text). The operation of `lock_page()` is similar to `refer()`: if the page is not in memory a page fault is caused to bring it into memory. Once the page is in memory, its `lock_count` is incremented to keep track of how many I/O operations are occurring using the page frame. When the operation has ended, `unlock_page()` is called to decrement the `lock_count` of the page frame. Multiple I/O operations can occur using one page frame, so after `unlock_page()` is called, the `lock_count` may not necessarily be zero. Note: each

`lock_page(iorb)` is a memory reference.

Overview of Page Replacement Algorithms

FIFO selects the page frame that has not been referenced for the longest time for replacement. LFU selects the least used frame for replacement. LFU with second chance for dirty pages is the LFU algorithm with the additional condition that the first iteration excludes pages from consideration if they are dirty. If no clean pages can be found for replacement, then a second iteration is done for the least frequently used page regardless of whether they are dirty or not.

Locked page frames cannot be selected regardless of the algorithm used. Separate data structures could be used to store the clock time and the number of references 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.

Neither algorithm is run unless there are no free frames available. These algorithms are only run when a page must be selected for replacement.

Tips

This assignment is a little more complicated than CPU scheduling or interrupt handling. It is advisable that you start as soon as possible.

Seek help if you need it (within certain guidelines as stated below).

Any cooperative work must be within the guidelines of the non-collaboration agreement signed by YOU. Straying from the guidelines will have dire consequences.

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 to copy and modify it to implement the other algorithms.