CS 4760 **Operating Systems**
Assignment # 6 Due Date: December 01, 2023

Memory Management

Purpose

The goal of this homework is to learn about memory management using virtual memory (paging) in an operating system. You will work on the specified page management algorithm to allocate and deallocate pages of memory.

Task

In this project, you will design and implement a memory management module for our Operating System Simulator `oss`. In this project, you will implement the second-chance (CLOCK) page replacement algorithm. However, processes will implement their memory accesses in two different ways. Your program should take in a command line option `-m x`, which accepts either a 0 or a 1, which determines how child processes will perform their memory access.

When a page-fault occurs, it will be necessary to swap in that page. If there are no empty frames, your algorithm will select the victim frame based on the CLOCK algorithm.

Each frame should also have an additional dirty bit, which is set on writing to the frame. This bit is necessary to consider dirty bit optimization to determine how much time these operations take. The dirty bit is implemented as a part of the page table.

There is no process scheduling in this project but you will be using IPC primitives (semaphores or message queues) to prevent race conditions.

Operating System Simulator

This will be your main program and serve as the master process. You will start the operating system simulator (call the executable `oss`) as one main process that will fork multiple children at random times. The randomness will be simulated by a logical clock that will be updated by `oss` but used by user processes. Thus, the logical clock resides in shared memory and should be modified only by `oss`. You should have two unsigned integers for the clock; one will show the time in seconds and the other will show the time in nanoseconds, offset from the beginning of a second.

In the beginning, `oss` will allocate shared memory for system data structures, including page table. You can create fixed sized arrays for page tables, assuming that each process will have a requirement of less than 32K memory, with each page being 1K. The page table should also have a delimiter indicating its size so that the users processes do not access memory beyond the page table limit. The page table should have all the required fields that may be implemented by bits (preferable) or character data types. You may want to consider the bit fields in `struct` to implement those bits.

Assume that your system has a total memory of 256K. Use a bit vector to keep track of unallocated frames.

After the resources have been set up, `fork` a user process at random times (between 1 and 500 milliseconds of your logical clock). You should accept a command line option to indicate the maximum number of user processes allowed in the system. However, make sure that you never have more than 18 user processes in the system. If you already have 18 processes, do not create any more until some process terminates. 18 processes is a hard limit and you should implement it using a `#define` macro. Thus, if a user specifies an actual number of processes as 30, your hard limit will still limit it to no more than 18 processes at any time in the system. Your user processes execute concurrently and there is no scheduling performed. They run in a loop constantly till they have to terminate.

`oss` will monitor all memory references from user processes and if the reference results in a page fault, the process will be *suspended* till the page is brought in. It is up to you how you do synchronization for this project; for example you could use message queues or a semaphore for each process. Effectively, if there is no page fault, `oss` just increments the clock by 10 nanoseconds and sends a signal on the corresponding semaphore. In case of page fault, `oss` queues the request to the device. Each request for disk read/write takes about 14ms to be fulfilled. In case of page fault, the request is queued for the device and the process is suspended as no signal is sent on the semaphore or message sent on the message queue. The request at the head of the queue is *fulfilled* once the clock has advanced by disk read/write time since the time the request was found at the head of the queue. The fulfillment of request is indicated by showing the page in memory in the page table. `oss` should periodically check if all the processes are queued for device and if so, advance the clock to fulfill the request at the head. We need to do this to resolve any possible deadlock in case memory is low and all processes end up waiting.

While a page is referenced, `oss` performs other tasks on the page table as well such as updating the page reference, setting up dirty bit, checking if the memory reference is valid and whether the process has appropriate permissions on the frame, and so on.

When the number of free frames falls below some value (let it be 10% of the total), a special daemon program kicks in. In `oss`, the logical place for this daemon to be invoked is the `get_page` routine called by each user process. Once invoked, the daemon sweeps the frame table, turning off the valid bit of the oldest n resident pages (if they don't expect I/O into them), where n is 5% of the total number of pages. This *does not* make the frames free, but only marks them for replacement. If the valid bit of the page in some frame is already off, then the frame is freed up completely (don't forget to save the page on disk if its dirty bit is on). Frames marked for replacement are *reclaimable*. This means that when a page fault occurs over a page sitting in such a frame, there is no need to swap it in – just turn the valid bit back on. Make sure that you perform dirty bit optimization.

When a process terminates, `oss` should log its termination in the log file and also indicate its effective memory access time. `oss` should also print its memory map every logical second showing the allocation of frames. You can display unallocated frames by a period ($\cdot$) and allocated frame by a +.

I would strongly suggest for debugging that you have a log option that displays every memory access request and how it is dealt with, as well as before and after displays of the frame information after the daemon sweeps it. Make sure to terminate this log after a particular number of writes, as you do not want logs that are too large.

User Processes

Each user process generates memory references to one of its locations. This should be done in two different ways. In particular, your project should take a command line argument (`-m x`) where `x` specifies which of the following memory request schemes is used in the child processes:

- The first memory request scheme is simple. When a process needs to generate an address to request, it simply generates a random value from 0 to the limit of the process memory (32k).
- The second memory request scheme tries to favor certain pages over others. You should implement a scheme where each of the 32 pages of a process' memory space has a different weight of being selected. In particular, the weight of a processes page n should be $1/n$. So the first page of a process would have a weight of 1, the second page a weight of 0.5, the third page a weight of 0.3333 and so on. Your code then needs to select one of those pages based on these weights. This can be done by storing those values in an array and then, starting from the beginning, adding to each index of the array the value in the preceding index. For example, if our weights started off as $[1, 0.5, 0.3333, 0.25, \ldots]$ then after doing this process we would get an array of $[1, 1.5, 1.8333, 2.0833, \ldots]$. Then you generate a random number from 0 to the last value in the array and then travel down the array until you find a value greater than that value and that index is the page you should request.

  Now you have the page of the request, but you still need the offset. Multiply that page number by 1024 (or left shift by 10 bits) and then add a random offset of from 0 to 1023 to get the actual memory address requested.

Each user process generates memory references to one of its locations. This will be done by generating an actual byte address, from 0 to the limit of process memory (as above). In addition, the user process will generate a random number to indicate whether the memory reference is a read from memory or write into memory (the percentage of reads vs writes should be configurable with a fair bias towards read). This information is also conveyed to `oss`. The user process will wait on its semaphore (or message queue) that will be signaled by `oss`. `oss` checks the page reference by extracting the page number from the address, increments the clock as specified above, and sends a signal on the semaphore if the page is valid.

Processes should have a small probability to request an invalid memory request. In this case the process should simply make a request for some memory that is outside of its legal page table. `oss` should detect this and deal with it by terminating the process. This should be indicated in the log.

At random times, say every 1000 $\pm$ 100 memory references, the user process will check whether it should terminate. If so, all its memory should be returned to `oss` and `oss` should be informed of its termination.

The statistics of interest are:

- Number of memory accesses per second
- Number of page faults per memory access
- Average memory access speed
- Number of seg faults per memory access

Termination Criterion

You should terminate after more than 40 processes have gotten through your system, or if more than 10 real life seconds have passed. Make sure that you have signal handling to terminate all processes, if needed. In case of abnormal termination, make sure to remove shared memory, queues, and semaphores.

Invoking the solution

Execute `oss` with at least one parameter `-p` to indicate the number of user processes in the system. Use 20 as the default value. Make sure to specify the memory type parameter as well. You can indicate any other parameters used in your `README`.

Log Output

An example of possible output may be:

```
Master: P2 requesting read of address 25237 at time xxx:xxx
Master: Address 25237 in frame 13, giving data to P2 at time xxx:xxx
Master: P5 requesting write of address 12345 at time xxx:xxx
Master: Address 12345 in frame 203, writing data to frame at time xxx:xxx
Master: P2 requesting write of address 03456 at time xxx:xxx
Master: Address 03456 is not in a frame, pagefault
Master: Clearing frame 107 and swapping in p2 page 3
Master: Dirty bit of frame 107 set, adding additional time to the clock
Master: Indicating to P2 that write has happened to address 03456

Current memory layout at time xxx:xxx is:
         Occupied   RefByte  DirtyBit
Frame 0: No            0        0
Frame 1: Yes          13        1
Frame 2: Yes           1        0
Frame 3: Yes         120        1
```

where `Occupied` indicates if we have a page in that frame, the `refByte` is the value of our reference bits in the frame and the dirty bit indicates if the frame has been written into.

In your `README` file, discuss the performance of the page replacement algorithm on both the page request schemes. Which one is a more realistic model of the way pages would actually be requested?

Suggested Implementation Steps

I suggest you implement these requirements in the following order:

1. Get a `Makefile` that compiles two source files, have `oss` allocate shared memory, use it, then deallocate it. Make sure to check all possible error returns. [Day 1]
2. Get `oss` to fork and exec one child and have that child attach to shared memory and check the clock and verify it has correct resource limit. Then test having child and `oss` communicate through message queues or semaphore. Set up PCB and frame table/page tables. [Day 2]
3. Have child request a read/write of a memory address (using the first scheme) and have `oss` always grant it and log it. [Day 3]
4. Set up more than one process going through your system, still granting all requests. [Day 4]
5. Now start filling out your page table and frame table; if a frame is full, just empty it (indicating in the process that you took it from is gone) and granting request. [Day 5-6]
6. Implement a wait queue for I/O delay on needing to swap a page. [Day 7]
7. Do not forget that swapping a page with a dirty bit should take more time on your device. [Day 8-9]
8. Implement the page replacement policy. [Day 10-12]
9. Implement the $1/n$ weight request scheme. [Day 13]
10. Add your observations into `README`. [Day 14]

Criteria for Success

Make sure that the code adheres to specifications. Document the code appropriately to show where the specs are implemented. I expect to see the use of bit vector to keep track of unallocated frames. *You must clean up after yourself.* That is, after the program terminates, whether normally or by force, there should be no shared memory, semaphore, or message queue that is left allocated to you.

Grading

1. *Overall submission: 25pts.* Program compiles and upon reading, seems to solve the assigned problem in the specified manner.
2. *README/Makefile: 5pts.* Ensure that they are present and work appropriately.
3. *Code readability: 10pts.* Code should be readable with appropriate comments. Author and date should be identified.
4. *Page replacement algorithm: 20pts.* Page replacement algorithm properly implemented.
5. *Dirty bit optimization: 10pts.* dirty bit optimization properly performed; used extra time to handle writing page onto disk.
6. *Use of bit vector to track unacllocated frames: 5pts.* Properly use the bit vector.
7. *Conformance to specifications: 20pts.* Algorithm is properly implemented and documented.
8. *Report: 5pts.* Observations are properly added into `README`.

Submission

Handin an electronic copy of all the sources, `README`, Makefile(s), and results. Create your programs in a directory called *username*.6 where *username* is your login name on opsys. Once you are done with everything, *remove the executables and object files*, and issue the following commands:

```
% cd
% chmod 755 ~
% ~sanjiv/bin/handin cs4760 6
% chmod 700 ~
```