CS 4760 | **Operating Systems**
Assignment # 6 | Due Date: Ma 9th, 2019

Memory Management

In this project we will be designing and implementing a memory management module for our Operating System Simulator `oss`.

We will be implementing an additional-reference-bits page replacement algorithm for our oss (in particular, 8 reference bits). That is, our frame table will contain one byte of reference bits per frame (which is initially set to zero). When a reference is made to a frame, the most significant bit of that reference byte is set to 1. Every so many total memory references to any frame anywhere (this number being a system constant to be determined by you), the bits in that byte should be shifted such that the most significant bit becomes the 2nd most significant bit and so on. The most significant bit after the shift should then be 0.

When a page-fault occurs, it will be necessary to swap in that page. If there are no empty frames, your algorithm will compare the reference bytes of each frame and then put that page in the frame with the smallest reference byte. Once done, that reference byte should be set to zero.

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 when determining how much time these operations take.

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 who will fork multiple children at random times. The randomness will be simulated by a logical clock that will be updated 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 your programs do not access memory beyond the page table limit. The page table should have all the required fields that may be implemented by bits or character data types.

Assume that your system has a total memory of 256K, so will require a frame table, with data required such as your reference byte and dirty bit.

After these resources have been set up, `fork` a user process at random times (between 1 and 500 milliseconds of your logical clock). You should accept as a command line argument the maximum number of child processes to allow in the system, but you should never allow this to be greater than 18. 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 has been brought in. How you implement this suspend is up to you, you can use either message queues or semaphores. `oss` will monitor these memory references by waiting for messages from user processes who will be sending a request for an address in their space. Effectively, if there is no page fault, `oss` just increments the clock by 10 nanoseconds and sends a message to the process indicating its request was granted. In case of page fault, `oss` queues the request to the device. Note that while we are not actually simulating disk access here, we do want to indicate that it is taking time in our system. Each request for disk read/write takes about 15ms 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 msg 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 the page is referenced, `oss` performs other tasks on the frame and page tables as well such as updating the reference byte, setting up dirty bit, checking if the memory reference is valid and whether any other bookkeeping it has to do.

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 100 memory accesses showing the allocation of frames and any reference bits used. You should, by default, put into the log every memory access request and how it was dealt with. Make sure to terminate this log after a particular number of writes, as you do not want logs that are too large.

For example at least something like...

```
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 12345 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 to.

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 the process memory. 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). This information is also conveyed to `oss`. The user process will `wait` on its semaphore or msg 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 or msgqueue if the page is valid.

At random times, say every 1000 ± 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. This should be tuned to make sure you have enough processes in the system.

The statistics of interest are:

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

Make sure that you have signal handling to terminate all processes, if needed. In case of abnormal termination, make sure to remove shared memory and semaphores.

I suggest you implement these requirements in the following order:

1. Get a makefile that compiles two source files, have master allocate shared memory, use it, then deallocate it. Make sure to check all possible error returns.
2. Get Master to fork off 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 master communicate through message queues. Set up PCB and frame table/page tables
3. Have child request a read/write of a memory address and have master always grant it and log it.
4. Set up more than one process going through your system. Still granting all requests
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 that it is gone) and granting request.
6. Implement a wait queue for I/O delay on needing to swap a process out
7. Do not forget that swapping out a process with a dirty bit should take more time on your device

Deliverables

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 hoare. Once you are done with everything, *remove the executables and object files*, and issue the following commands:

```
chmod 700 username.6

cp -p -r username.6 /home/hauschild/cs4760/assignment6
```