CS 4760 **Operating Systems**
Assignment # 6 Due Date: April 25th, 2018

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 a second-chance FIFO page replacement algorithm. That is, assume you have a hardware reference bit that is turned on each time a page is referenced. (Figure out how the reference bit can be simulated in `oss`.) In addition, you should keep a circular FIFO queue of pages. When a page-fault occurs, it will be ncessary to swap in that page. Normally, the oldest page must be swapped out, unless it has been referenced (the reference bit is 1). In that case, reset the bit to 0 and move the head pointer to the next page in the queue. Do not forget 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` as well as user processes. Thus, the logical clock resides in shared memory and is accessed as a critical resource using a semaphore. 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. Use a bit vector to keep track of the 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 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. 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. 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 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 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 second showing the allocation of frames and any reference bits used.

For example at least something like...

```
....U.DDU.UUU...UDDU...UU
....1.110.111...1101...10
```

where . is a free frame and D is a dirty frame, with U being simply an occupied frame. The second line is showing the reference bits used by the second-chance algorithm.

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

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.

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
- Throughput

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.

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
```