CS 4760 **Operating Systems**
Assignment # 6 Due Date: December 01, 2017

Memory Management

Design and implement a memory management module for our Operating System Simulator `oss`.

Implement a Unix-like second-chance algorithm. 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 corresponding resident pages (if they don't expect I/O into them). 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. 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. I want your code to keep track of a dirty bit in the frames as well, which would indicate in a real system that you need to copy that data out to disk.

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). 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. The actual number of processes in the system should be defined using another macro and limit that to 12 for this project. 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. Implement the memory references through 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 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. 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.

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. 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.

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
- 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
```