**HW 5 (wet exercise)**

In this assignment you will simulate a basic memory system, using multi-thread and multi-process programming. The system parameters are detailed (in capitalized letters) below.

**The system contains the following modules**

**1. Process 1**

Simulates a process, which runs on the CPU. The process is merely an endless loop, which does the following:

* 1. Wait for INTER\_MEM\_ACCS\_T [ns].
* 2. Invoke a memory access, which is a write with probability 0 < WR\_RATE < 1; and a read otherwise.
* 3. Send a request to the MMU (Memory Management Unit).
* 4. Wait for an ack from the MMU.
* 5. GoTo 1.

For simplicity, we will **totally discard** the data and the virtual addresses. Therefore, process 1 should inform MMU only that it requests a memory access, and the access mode (wr or rd).

**2. Process 2**

Identical to process 1. We use two processes, so as to simulate parallel execution.

**3. Memory Management Unit (MMU)**

The memory contains *N* pages. Denote as “empty” a page in the memory which is invalid; and by “used”, a valid page.

We say that the memory is *empty* if all the pages in it are invalid, and *full* if all the pages are valid.

A *write* to the memory takes MEM\_WR\_T [ns]. A *read* from the memory is immediate.

Recall, that for the sake of simplicity, we do not really simulate data. Therefore, the “memory” is merely an array of *N* Booleans, indicating whether a page in the memory is dirty or not. You’ll probably need also some pointers / counters, to indicate the next page to load / evict from the memory.

The MMU includes (at least) 3 threads:

**3. A. The “main” thread**

Receives requests from processes 1 and 2.

* If the memory is empty, the request is a miss (*page fault*).
* If the memory isn’t empty, the request is a hit with probability 0 < HIT\_RATE < 1, and a miss otherwise.
* In case of a read hit, immediately acknowledge the requesting process that the access was “done”.
* In case of a hit write
  + Sleep for MEM\_WR\_T [ns]
  + Choose uniformly at random one of the used pages in the memory, and mark it as dirty.
  + Acknowledge the requesting process that the access was “done”.
* In case of a miss (page fault)
  + If the memory is full
    - Wake up the *evicter* (to be described shortly)
    - Wait until the evicter wakes me up again, indicating that the memory is not full anymore.
  + Once the memory is not full, the thread sends the HD(*hard disk*) a request to read a page. After receiving an acknowledge from the HD, the thread “writes” the page to the memory and acknowledges the requesting process, same as described above in the case of a hit.

**3. B. Evicter**

The evicter is woken up by the main thread every time the memory is full.

The evicter chooses which page to evict in a FIFO manner, using the clock scheme, as described in the tutorials. If the page is dirty, the eviction requires sending a request and receive an ack from the HD, same as describe above for the main thread.

If the number of used slots in the memory is *N-1* (namely, the memory was full before evicting), the evicter wakens up the main thread, to let him load a page, if needed.

In any case, the evicter continues evicting pages, until the number of the used slots in the memory is below USED\_SLOTS\_TH. Then, the evicter stops evicting, and waits for the main thread to wake it up again.

**3.C. printer**

At the beginning of the simulation, and later every TIME\_BETWEEN\_SNAPSHOTS [ns], the printer prints the “memory”. Every slot in the memory is marked by *0* if it’s valid and clean; *1* if it’s valid and dirty; and *–* if it’s invalid (*empty*). For instance:

0|-

1|0

2|0

3|1

4|-

The snapshots have to be consistent. Namely, no read / writes are allowed to / from the memory when the printer takes the snapshot. However, for minimizing the critical section, the printer should lock the access to the memory only for a short time, in which it copies the memory and relevant counters to local variables (we simulate a small memory, so this is not a problem). Only after releasing the lock, the printer prints the memory in the format described above.

Every 2 prints are separated by 2 empty lines.

**4. HD (Hard Disk)**

Forever

1. Receive requests.
2. Wait HD\_ACCS\_T [ns].
3. Send the requester an indication, that the request was “done”.

**Simulation termination**

The simulation takes SIM\_TIME **seconds**. Later, the message "Successfully finished sim" should be printed, and the simulation should be finished.

**Additional Requirements**

* Upon terminating the simulation from any reason (either a successful finish, or an error), you should destroy all the mutexes, release the dynamically allocated memory, if you used such, and kill all the processes and threads.
* You should check the return values of calls to functions such as locking mutexs, *fork()*, *pthread\_create()*, *msgsnd()*and so on. In case of a fail, an appropriate message should be printed, and the simulation should be terminated as described above.
* You should initialize all the mutexes and condition variables. This can be done statically as shown [here](https://computing.llnl.gov/tutorials/pthreads/#MutexCreation).
* Minimize the sections of code which require mutual exclusion.
* The program should avoid deadlocks and race conditions.
* The program should print nothing beside what was detailed above.

**Help and clues**

* For performing the required checks detailed above, it’s recommended to code and use simple accessory functions, e.g.: *my\_pthread\_create()*, *my\_lock()* etc.
* For Inter-Process Communication you may use [*msgget()*](http://www.tldp.org/LDP/lpg/node34.html#SECTION00742300000000000000)*,* [*msgsnd()*](http://www.tldp.org/LDP/lpg/node35.html) and *msgrcv()*, as in [this example](https://macboypro.wordpress.com/2009/05/15/posix-message-passing-in-linux/).
* For making threads wait / wake up each other, you may use mutexes and [condition variables](https://computing.llnl.gov/tutorials/pthreads/#ConVarCreation) as in the answer [here](http://stackoverflow.com/questions/13675132/pthread-cond-wait-for-2-threads). It’s rather similar to the issues of monitors and producer-consumer, which we learnt at class.
* The *%* modulo operator in *C* doesn’t work as expected with negative numbers.