programming exercise

In this assignment you will simulate a basic memory system, which includes user processes, an MMU (memory management unit), and a hard disk. You will use 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.
    - The access is a write with probability 0 < WR_RATE < 1; and a read otherwise.
    - The access is to a page, whose number is picked uniformly at random from [0, 1, 2, …, WS-1]
        - WS is a parameter indicating the *Working Set*, that is, the number of pages allocated for the process.
- 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 offset within a page. Therefore, process 1 should send MMU only the requested page number, and the access mode (wr or rd).

2. Process 2

Process 2 is identical to process 1. We use two processes, so as to simulate a system with multiple processes. For having time difference between them, process 2 should be generated (by "fork") 0.5*INTER_MEM_ACCS_T [ns] after process 1 is generated.

3. Memory Management Unit (MMU)

This unit handles requests from the processes and "interrupts" from the hard disk. In particular, it simulates a *page table* and a *physical memory*.

Physical memory

Contains *N* pages, where N<2WS. Namely, there's no enough space for all the processes' pages within the memory simultaneously.
Recall that we do not really simulate data. Therefore, the “memory” is simply an array of flags, indicating whether a page in the memory is invalid, valid clean, or valid dirty.
You’ll probably need also some pointers / counters, to indicate the next page to load / evict from the memory.
A *write* to the memory takes MEM_WR_T [ns]. A *read* from the memory is immediate.

Virtual to physical address translation and page table

For simplicity, we allocate virtual pages 0, 1 ,2, …, WS-1 to process 1, and virtual pages WS, WS+1, …, 2*WS-1 to process 2. Therefore, when process 1 accesses a page number

i, the virtual page number is merely *i*; and when process 2 accesses a page number *i*, the virtual page number is *WS+i*.
The **page table** is an array of *2*WS* integers. If virtual page *i* is currently found in the memory, entry *i* in the page table should hold the physical address of page *i*. Else, entry *i* should be -1, indicating that this page is found only in the hard disk.
An access to the page table (either read or right) is immediate.

The MMU includes (at least) 3 threads:

3. A. The "main" thread

This thread handles requests from processes 1 and 2 as follows.

- If the requested page is in the memory, the request is a hit. Else, it's a miss (*page fault*).
- In case of a read hit, immediately acknowledge the requesting process that the access was "done".
- In case of a write hit
    - Sleep for MEM_WR_T [ns]
    - Mark the respective page in the memory as dirty.
    - Acknowledge the requesting process that the access was "done".
- In case of a miss (page fault)
    - If the memory is full (namely, all its frames are valid)
        - Wake up the *evicter* (described below)
        - Wait until the evicter wakes me up again, indicating that the memory is not full anymore.
    - Else (namely, 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; updates the page table; 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 FIFO manner, using the clock scheme, as described in the tutorials. For simplicity, we ignore the dirty bit when evicting. That is, the page evicted is merely the oldest page in the memory, no matter whether it is dirty or not.
If the evicted page is dirty, the evictor writes it to the HD by sending a request and receiving an ack from the HD.
Once evicting a page, the evictor updates the memory and the page table accordingly. For simplicity, assume that the page table is very small, so it's OK to linearly search for an entry in it, if needed.
After evicting a single page, the evicter stops, and waits for the main thread to wake it up again.

3.C. print_mem

Each time a new page arrives to the memory (from the HD), and each time a page is evicted from the memory, you should call the function *print_mem ()*
print_mem () prints a snapshot of the page table and the memory as follows.
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, when N=3 and WS=2, the output may be:
Page table
0|0
1|2
2|1
3|-1

Memory
0|0
1|0
2|1

The snapshots have to be consistent. Namely, no read / writes are allowed to / from the memory when *print_mem()* takes the snapshot. However, for minimizing the critical section, *print_mem()* should lock the access to the page table and the memory only for a short time, in which it copies them to local variables. We assume very small memory and page table, so this copy is not a problem. Only after releasing the lock, *print_mem()* prints the output as described above.

Sequential prints are separated by an empty line.

4. HD (Hard Disk)

Forever

1. Receive requests.
2. Sleep for 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 system calls, such as *pthread_mutex_lock(), fork(),msgsnd()* etc. In case of a fail, an appropriate message should be printed, and the simulation should be terminated as described above.
- For Inter-Process Communication use *msgget()*, *msgsnd()* and *msgrcv()*; you may use this example.
- Synchronize the memory’s main thread and the evicter using *mutexes* and *condition variables*; you may use this example. It’s rather similar to the issues of monitors, producer-consumer and sleeping barber, which we learnt at class.
- Initialize the mutexes and the condition variables. This can be done statically as shown here.
- For sleeping any time below 1 second, use *nanosleep()*
- 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 hints

- **Start now. Don't wait to the last moment!**
- For performing the required checks detailed above, it's recommended to code and use simple accessory functions, eg: *my_pthread_create()*, *my_lock()* etc.
- The *%* modulo operator in *C* doesn't always work as expected. You may therefore code explicitly, e.g.
  ```
  if (cnt == N)
    cnt = 0;
  else
    ent = cnt+1;
  ```
- **Use small demo program for every new function you learn.** E.g. write a little program which uses *nanosleep()*; another – for 2 process which use *msgsnd()* and *msgrcv()* etc.
- It's possible to debug using *printf()*. However, recall that *printf()* is a system call which changes the scheduling of the threads; furthermore, the results may vary from what you intuitively expect. For instance, recall what we learnt about *fflush()*)
- It's recommended to code for debugging, e.g.
  ```
  #ifdef DEBUG_MODE
          do_something…
  #endif
  ```
- When coding, use pencil and recycled paper to well understand the expected contents of the memory and the page table.
- Dedicate ½ hour for manually "run" your code using a paper & pencil before starting debugging will save you many hours of gazing at the screen.
- **Start now. Don't wait to the last moment.**