

## Home Project

Attached Files:

-  [main.cpp](#) (969 B)

**The project is due to December 14 (11:59 pm).**

**The program should be submitted ONLY via GradeScope.**

**No late submissions.**

**The project is meant for individual work only. My “questions about the project” policy is in place.**

**The program must compile and run on Linux Lab computers (0 points if it doesn't).**

**Don't submit archives.**

**All your source and header files should start with a comment with your name in it.**

**I will subtract large amount of points for poor programming practices, including but not limited to:**

- ***using namespace std;* in any header files**
- **goto operator**
- **non-constant global variables**
- **using C-style arrays instead of C++ vectors and other containers**
- **doing explicit dynamic memory allocation**
- **lack of classes and poor program structure**
- **bad variable naming**
- **extreme inefficiencies**

In this assignment you will create a C++ library that has a class ***SimulatedOS***. Your library should contain a header file ***SimulatedOS.h*** that I will include in the test driver. Number and names of other files are up to you, but they should follow reasonable programming practices. Your submission should not have function `main()`. But you will surely need it while working on the assignment.

All the specifications in this assignment should be followed exactly as given. Even small discrepancies will make your project fail with the test driver and will result in 0 points! For example, if you name your header `SimulatedOS.hpp` instead of `SimulatedOS.h`, my test driver will fail to include your library, fail all the tests, and your project will be graded 0 points. For the same reason, don't add your own namespaces.

Please find a tiny test driver attached to the assignment. Place it in folder with your library and build it. It should compile, run, and show the things specified in comments. On Linux lab, the compilation command will be exactly:  
***g++ -std=c++17 \*.cpp -o runme***

The test driver I provide is intentionally tiny and very incomplete. I want you to think about use cases, find edge cases, and create your own test driver. It is an important skill to master.

Pay attention, the whole assignment deals with simulation. There is no real CPU and memory management. You can do the whole thing with the tools you learned in CSCI 23500.

## “OS simulation”

**CPU scheduling** is priority-based. Every process has a priority number. The higher is the number, the higher is priority. The process with higher priority uses the CPU. The scheduling is **preemptive**. It means that if a process with the higher priority arrives to the ready-queue while a lower-priority process uses the CPU, the lower-priority process is preempted (that is moved back to ready-queue) while the higher priority process immediately starts using the CPU. Pay attention, higher-priority process never waits in the ready-queue while lower-priority process uses the CPU.

If there are two or more processes with the same highest priority in the ready-queue, your system can schedule any of them to the CPU.

For **memory management**, our simulation uses paging. If the memory is full, the least recently used frame is removed from memory.

**Disk management** is “first-come-first-served”. In other words, all disk I/O-queues are real queues (FIFO).

Create a class ***SimulatedOS***. The following methods should be in it:

- ***SimulatedOS( int numberOfDisks, int amountOfRAM, int pageSize )***

The parameters specify number of hard disks in the simulated computer, amount of memory and page size.

Disks and pages enumeration starts from 0.

The amountOfRAM should always be divisible by pageSize. Feel free to throw any exception if it is not so. When testing, I promise to use only

correct and greater than zero constructor parameters.

- ***NewProcess( int priority )***

Creates a new process with the specified priority in the simulated system. The new process takes place in the ready-queue or immediately starts using the CPU.

Every process in the simulated system has a PID. Your simulation assigns PIDs to new processes starting from 1 and increments it by one for each new process. Do not reuse PIDs of the terminated processes.

Every process has a program counter. PC starts from 0 for a new process. This PC of ours doesn't behave like a real PC and updates only when we issue a corresponding instruction.

Page 0 that has the starting address of 0 should be loaded in RAM when a new process appears. It is considered "freshly used". No process can run until page with the current PC address is loaded in the memory.

- ***Exit()***

Currently running process wants to terminate. Immediately remove the process from the system and free all the memory it was using.

- ***DiskReadRequested( int diskNumber, std::string fileName )***

Currently running process requests to read the specified file from the disk with a given number. The process issuing disk reading requests immediately stops using the CPU, even if the ready-queue is empty.

- ***FetchFrom( unsigned int memoryAddress )***

A currently running process wants to fetch instruction from RAM at the specified logical address. PC must be updated with this new address. This is the only function in our simulation that updates PC (again, this is very unrealistic).

When *FetchFrom* is called, your simulation should make sure the page with the needed address is in memory.

- ***DiskJobCompleted( int diskNumber )***

A disk with a specified number reports that a single job is completed. The served process should return to the ready-queue or immediately start using

the CPU (depending on the priority).

- ***PrintCPU()***

*PrintCPU* prints on the screen the PID of the process currently using the CPU.

- ***PrintReadyQueue()***

*PrintReadyQueue* prints the PIDs of processes in the ready-queue in any order.

- ***PrintRAM()***

*PrintRAM* prints a sorted list of all used frames, page number stored in it, and PID of the process that owns that page. See the test file for the possible output format.

- ***PrintDisk( int diskNumber )***

*PrintDisk* prints the PID of the process served by specified disk and the name of the file read for that process.

- ***PrintDiskQueue( int diskNumber )***

*PrintDiskQueue* prints PIDs of the processes in the I/O-queue of the specified disk starting from the “next to be served” process PID.

If a disk with the requested number doesn't exist, just ignore the instruction, and output a message that the instruction was ignored.

If instruction is called that requires a running process, but the CPU is idle, just ignore the instruction, and output a message that the instruction was ignored.

When a process uses a CPU, the memory page that contains its current PC is considered currently in use.

Good luck!