The project is due to May 19 (11:59 pm).

The program should be submitted ONLY via GradeScope (it will be available later).
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. If I find any that doesn't, I penalize 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**
- **debug printing remaining (actually this one can just fail the whole project when used with test driver)**

In this assignment you will create a C++ library that has a class ***SimOS***. Your library should contain a header file ***SimOS.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 SimOS.hpp instead of SimOS.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.

Create a test driver file (with function main()) and place it in folder with your library. Then build it. It should compile, run, and show correct output. On Linux lab, the compilation command will be exactly:
g++ -std=c++17 *.cpp -o runme

Your submitted program should do no debug printing. Make sure your debug printing is disabled or deleted before you submit your work.

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 round-robin. Ready-queue is a real first-come-first-serve queue. Each process has a time limit for how long it can use the CPU at once. If a process uses the CPU for a longer time, it goes back to the end of the ready-queue.

For **memory management**, our simulation uses paging. If the memory is full, the least recently used frame is removed from memory.
I can ask your library to simulate large amounts of memory (say 64 GB or even more).
It is <u>not allowed</u> to represent each byte of memory individually. For example, it is <u>not allowed</u> to use vector with 1000 elements to represent 1000 bytes of memory.

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

Use the following code:

struct FileReadRequest

```
{
   int  PID{0};
   std::string fileName{""};
};

struct MemoryItem
{
   unsigned long long pageNumber;
   unsigned long long frameNumber;
   int PID; // PID of the process using this frame of memory
};

using MemoryUsage = std::vector<MemoryItem>;

constexpr int NO_PROCESS{ 0 };
```

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

- ***SimOS(int numberOfDisks, unsigned long long amountOfRAM, unsigned int pageSize)***
  The parameters specify number of hard disks in the simulated computer, amount of memory, and page size.
  Disks, frame, and page enumerations all start from 0.

- ***void NewProcess( )***
  Creates a new process 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.

- ***void SimFork()***
  The currently running process forks a child. The child is placed in the end of the ready-queue.

- ***void SimExit()***
  The process that is currently using the CPU terminates. Make sure you release the memory used by this process immediately. If its parent is already waiting, the process terminates immediately and the parent becomes runnable (goes to the ready-queue). If its parent hasn't called wait yet, the process turns into zombie.
  To avoid the appearance of the orphans, the system implements the cascading termination. Cascading termination means that if a process terminates, all its descendants terminate with it.

- ***void SimWait()***
  The process wants to pause and wait for any of its child processes to terminate. Once the wait is over, the process goes to the end of the ready-queue or the CPU. If the zombie-child already exists, the process proceeds right away (keeps using the CPU) and the zombie-child disappears. If more than one zombie-child exists, the system uses one of them (any!) to immediately resumes the parent, while other zombies keep waiting for the next wait from the parent.

- ***void TimerInterrupt()***
  Interrupt arrives from the timer signaling that the time slice of the currently running process is over.

- ***void DiskReadRequest(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.

- ***void 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.

- ***void AccessMemoryAddress(unsigned long long address)***
  Currently running process wants to access the specified logical memory address. System makes sure the corresponding page is loaded in the RAM. If the corresponding page is already in the RAM, its “recently used” information is updated.

- ***int GetCPU( )***
  GetCPU returns the PID of the process currently using the CPU. If CPU is idle it returns NO_PROCESS (see the supplied definitions above).

- ***std::deque<int> GetReadyQueue( )***
  GetReadyQueue returns the std::deque with PIDs of processes in the ready-queue where element in front corresponds start of the ready-queue.

- ***MemoryUsage GetMemory( )***
  GetMemory returns MemoryUsage vector describing all currently used frames of RAM. Remember, Terminated "zombie" processes don't use memory, so they don't contribute to memory usage.
  MemoryItems appear in the MemoryUsage vector in the order they appear in memory (from low addresses to high).

- ***FileReadRequest GetDisk(int diskNumber)***
  GetDisk returns an object with PID of the process served by specified disk and the name of the file read for that process. If the disk is idle, *GetDisk* returns the default *FileReadRequest* object (with PID 0 and empty string in *fileName*)

- ***std::deque<FileReadRequest> GetDiskQueue(int diskNumber)***
  GetDiskQueue returns the I/O-queue of the specified disk starting from the "next to be served" process.

If a disk with the requested number doesn't exist throw std::out_of_range exception.

If instruction is called that requires a running process, but the CPU is idle, throw std::logic_error exception.

Good luck!