“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.

Memory management uses contiguous memory allocation with the “best-fit” approach. The description of “best fit” contiguous memory allocation will be given in class and is also available in our textbook.
I can ask your library to simulate large amounts of memory (say 64 GB or even more). C++ datatype *unsigned long long* should be able to store such numbers.
It is not allowed to represent each byte of memory individually. For example, it is not allowed to use vector with 1000 elements to represent 1000 bytes of memory.
When multiple memory holes simultaneously satisfy “best fit” condition, use the first one (the one with the smallest memory address).

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 itemAddress;
   unsigned long long itemSize;
   int PID; // PID of the process using this chunk of memory
};

using MemoryUsage = std::vector<MemoryItem>;
```

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

- ***SimOS(int numberOfDisks, unsigned long long amountOfRAM)***
  The parameters specify number of hard disks in the simulated computer and amount of memory.
  Disks enumeration starts from 0.

- ***bool NewProcess(int priority, unsigned long long size)***
  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.
  For example, the command ***NewProcess(5, 1000)*** means that a new process with priority level 5 should be created and it requires 1000 bytes of memory.
  NewProcess returns true if a new process was successfully created and false if otherwise. One of the reasons a process wasn't created is insufficient free memory in the system.

- ***bool SimFork()***
  The currently running process forks a child. The child's priority and size are inherited from the parent. The child is placed in the end of the ready-queue.
  SimFork() returns true if a new process was successfully created and false if otherwise. One of the reasons a process wasn't created is insufficient free memory in the system.

- ***void SimExit()***
  The process that is currently using the CPU. 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 or CPU). 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 then one zombie-child exists, the system uses one of them (any!) to immediately restart the parent and other zombies keep waiting for the next wait from the parent.

- ***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 or immediately start using the CPU (depending on the priority).

- ***int GetCPU( )***
  GetCPU returns the PID of the process currently using the CPU. If CPU is idle it returns 0.

- ***std::vector<int> GetReadyQueue( )***
  GetReadyQueue returns the vector with PIDs of processes in the ready-queue in any order.

- ***MemoryUsage GetMemory( )***
  GetMemory returns MemoryUsage vector describing locations of all processes in memory.
  Terminated "zombie" processes don't use memory, so they don't contribute to memory usage.
  Processes appear in the MemoryUsage vector in the same 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::queue<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, 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.

Good luck!