Introduction to the OS Simulator

A simulation of a single-cpu operating system has been written in C++. The basic operating system is rather simple minded. You will improve it in a number of ways over the next several weeks.

What our program simulates is the *process management*, the *memory management*, and the *disk management* portions of the OS. In the following, a few of the more important classes are described.

The *memory manager* is comprised of (primarily) two classes. The `PhysicalMemory` class (in file `memory.H`) represents the actual memory of the machine. It is essentially an array of physical frames indexed by a variable of type `addr`. In a real computer, such an index would be the high-order bits (frame number) of an actual address. (The simulation does not deal with the displacement portion of an address.)

The `VirtualMemory` class (in file `memory.H`) represents the virtual memory of a particular process. The primary data structure is an array of logical pages that make up the page table for that process.

The `Disk` class (in file `disk.H`) represents the disk drive hardware in the system. On this system a disk drive has only one platter with only one side. A disk is divided into several tracks; each track contains some number of blocks. A block on the disk is the same size as a page in memory. The current position of the drive head is marked by `headPos`.

The `DiskFile` class (in file `disk.H`) represents an individual file on a disk. The starting address of the file is held in `firstFile`. For every page table in the system there is an instantiation of a `DiskFile` (called `swapFile`) for use by the page replacement algorithm.

The `Scheduler` class (in file `scheduler.H`) is used to control which process is using the CPU. There is an object of class `Semaphore` that is used to represent the *ready queue* for the system. When a process successfully performs a `ready.P()`, that process "owns" the CPU. When a process is blocked attempting to do a `ready.P()`, then that process is on the CPU's ready queue. Since objects of type `Semaphore` queue up waiting `P()` requests in FIFO order, this simulates a FIFO scheduler. At the end of its time slice, a process must call `checkTimeSlice()`. This routine first delays a few time units (to simulate the time that the process used on the CPU), and then releases the CPU and goes to the end of the ready queue (by issuing a `ready.P()` again). You won't have to worry about calling the function `checkTimeSlice()`; this is taken care by the class `Process` (in file `process.H`), which simulates the execution of a single process on the system. Note also that we are not measuring *actual* time, but *simulated* time instead (nominally in milliseconds).

The workload on the system is simulated by a number of *user processes*, which are objects of class `Process` (in file `process.H`). These processes can be either compute bound (belong to the subclass `CPUBoundProcess`) or IO bound (belong to the class `IOBoundProcess`). These represent numerical optimization applications and transaction processing applications, respectively. These processes are initialized and "launched" by the main program of the simulation (file `main.C`) after the OS has been initialized. There should be no need for you to modify any code in `process.H` or `process.C`.

The basic synchronization primitive provided in this simulation are objects of the class `Semaphore`. Semaphores can be initialized with arbitrary values, and the operations `P()` and `V()` behave in

"textbook manner".

Please take notice of the following important detail in our simulation: Whenever a process makes a `P()` operation on a semaphore, and therefore can potentially be blocked, it should release the CPU beforehand (by calling the `dispatch()` operation of the scheduler). After the `P()` operation, the execution can be resumed by calling the `resume()` operation of the scheduler. In our code, every `P()` operation should therefore be enclosed between two calls to the scheduler in the following way:

```
scheduler->dispatch();
mutex->P();
scheduler->resume();
```

The class `Task` (file `task.H`) can be used to simulate tasks (not necessarily processes) that go on in the system; these can be monitoring tasks, disk controllers, and such. Processes (which are derived from tasks) for instance are tasks that need a CPU in order to execute. You define a task for your purposes (e.g. `MonitorTask`) by creating a derived class `MonitorTask` from `Task` and by redefining the virtual method `execute_program` to do what you want the task to do. For an example, look at class `Process`.

In order to support the simulation, there are a few function that can be of interest. They are, in particular:

- `float simtime()`: This function is provided by the underlying simulation environment and returns the current time in the simulation (which, of course, has nothing to do with real time!).
- `void hold(float holdtime)`: This function simulates the process waiting for the given amount of time. You probably don't need to use this, unless you program special tasks used for e.g. monitoring (used e.g. in MP 1).
- `char * Process::current_process()`: This function returns the name of the process (typically a user process) currently being simulated. Note that this function if a static method of the class `Process`.

In addition, there is a number of utility classes. You should use them whenever possible. For example, rather than reinvent the wheel, you should use the classes `List` that are provided, and try to derive queues and similar data structures from it whenever possible

An item of advice: The fact that there is a simulation environment behind the program you are writing may initially seem confusing to some of you. Don't worry. The machine problems have been structured so that the simulation environment is all hidden, except for some include files. If at some point you have the impression that you need to know more about the simulator, you are very probably doing something wrong.