Assignment 02: Process State Simulation

CSci 430: Introduction to Operating Systems

Overview

In this assignment we will simulate a three-state process model (ready, running and blocked) and a simple process control block structure as introduced in Chapter 3 of our textbook. This simulation will utilize a ready queue and a list of blocked processes. We will simulate processes being created, deleted, timing out because they exceed their time quantum, and becoming blocked and unblocked because of (simulated) I/O events.

Questions

- How does round robin scheduling work?
- How does an operating system manage processes, move them between ready, running and blocked states, and determine which process is scheduled next?
- What is the purpose of the process control block? How does the PCB help an operating system manage and keep track of processes?

Objectives

- Explore the Process state models from an implementation point of view.
- Practice using basic queue data types and implementing in C.
- Use C/C++ data structures to implement a process control block and round robin scheduling queues.
- Learn about Process switching and multiprogramming concepts.
- Practice using STL queues and list data structures.

Introduction

In this assignment you will simulate a three-state process model (ready, running and blocked) and a simple list of processes, like the process control block structure as discussed in Chapter 3. Your program will read input and directives from a file. The input describes events that occur to the processes running in the simulation. These are the full set of events that can happen to and about processes in this simulation:

Event	Description
new	A new process is created and put at tail of the ready queue
done	The currently running process has finished and will exit the system
block eventId	The currently running process has done an I/O operation and is waiting on an event with the particular eventId to occur
unblock eventId	The eventId has occurred, the process waiting on that event should be unblocked and become ready again.
CPU	Simulate the execution of a single CPU cycle in the simulated system. The system time will increment by 1, and if a process is currently running on the CPU, its time and time quantum will be increased. The increase of the process time quantum used is how we determine when a process has exceeded its allotted time and needs to be returned back to the ready queue.

In addition to these events, there are 2 other implicit events that need to occur before and after every simulated event listed above.

Action	Description
dispatch	Before processing each event, if the CPU is currently idle, try and dispatch a process from the ready queue. If the ready queue is not empty, we will remove the process from the head of the ready queue and allocate it the CPU to run for 1 system time slice quantum.
timeout	After processing each event, we need to test if the running process has exceeded its time slice quantum yet. If a process is currently allocated to the CPU and running, check how long it has been run on its current dispatch. If it has exceeded its time slice quantum, the process should be timed out. It will be put back into a ready state, and will be pushed back to the end of the system ready queue.

The input file used for system tests and simulations will be a list of events that occur in the system, in the order they are to occur. For example, the first system test file looks like this:

```
----- process-events-01.sim --------
new
cpu
cpu
cpu
new
cpu
cpu
cpu
cpu
block 83
cpu
cpu
unblock 83
cpu
cpu
done
cpu
cpu
cpu
cpu
---------------------------------
```

The simulation you are developing is a model of process management and scheduling as described in chapter 3 from this unit of our course. You will be implementing a simple round-robin scheduler. The system will have a global time slice quantum setting, which will control the round-robin time slicing that will occur. You will need to create a simple ready queue that holds all of the processes that are currently ready to run on the CPU. When a process is at the head of the ready queue and the CPU has become idle, the system will select the head process and allocate it to run for 1 quantum of time. The process will run on the CPU until it blocks on some I/O event, or until it exceeds its time slice quantum. If it exceeds its time slice quantum, the process should be put back into a ready state and put back onto the end of the ready queue. If instead a block event occurs while the process is running, it should be put into a blocked state and information added to keep track of which event type/id the process is waiting to receive to unblock it. In addition to timing out or becoming blocked, a running process could also finish and exit the system.

Your task is to complete the functions that implement the simulation of process creation, execution and moving processes through the three-state process event life cycle. You will need to define a process list for this assignment, using an STL container like a list or a map. The `Process` class will be given to you, which defines the basic properties of processes used in this simulation. But you will need to write methods for the `ProcessSimulator` and define your process list, ready queue, and other structures to keep track of blocked processes and the events they are waiting on.

Unit Test Tasks

There are 3 classes given to you for this assignment, defined in the `ProcessState.[cpp|hpp]`, `Process.[cpp|hpp]`, and `ProcessSimulator.[cpp|hpp]` files respectively. You will mostly need to add code and functions to the `ProcessSimulator` class. You probably will not need to make any changes to the `ProcessState` type nor the `Process` class, though if you feel it makes your solution or approach easier, you can make changes or additions as needed to those classes.

You should probably begin by familiarizing yourself with the `ProcessState` enumerated type that is give to you. This is a user defined data structure that simply defines an enumerated type of the valid process states that processes can be in in your simulation. These correspond to the 3/5 process states from our textbook, e.g. `NEW, READY, RUNNING, BLOCKED` and `DONE`. For your simulation, processes will pretty much be in one of the `READY/RUNNING/BLOCKED` states. You will need to handle the creation of `NEW` processes, but in your simulation when a `NEW` process enters the system it should immediately be transitioned into a `READY` state and added to the end of the ready queue, so it will not stay in the `NEW` state long enough to see this state normally.

The other class that is given to you for this assignment is the `Process` class defined in the `Process.hpp` header file and the `Process.cpp` implementation file. The `Process` class should define most all of the information you will need to keep track of the current state and information about processes being managed by your simulation. For example, if you look in the `Process` header file you will see that a `Process` has member variables to keep track of the processes unique identifier (its pid), the state the process is currently in, the time when the process entered the system and was started, etc. For the most part, you should only need to use the public functions given for the `Process` class to create and manage the processes you will need to implement your simulation.

As a starting point, just like in assignment 1, you should begin with the unit tests given to you in the `assg02-tests.cpp` file. The first test case in the unit tests actually test the `Process` class. These tests should all be passing for you. You can look at that code to get an idea of how you should be using the `Process` class in your simulation.

Your work will begin with the second test case, that starts by testing the initial construction and setup of the `ProcessSimulator`, then tests the individual methods you will need to complete to get the simulation working.

So for this assignment, you should start by getting all of the unit tests to pass, and I strongly suggest you work on implementing the functions and passing the tests in this order. You will need to perform the following tasks.

1. You should start by getting the initial getter function tests to work in the second test case. We did not give you the implementation of the constructor for the `ProcessSimulator` class, so you will need to start with a constructor that specifies the system time slice quantum and saves that value. The other functions that are tested in this first unit test are things like `getNextProcessId()`, `getNumActiveProcesses()`, `readyQueueSize()`, `blockedListSize()`, etc. You will need to initialize member variables in the constructor, like the `timeSliceQuantum`, `systemTime`, `nextProcessId`, etc., and modify some or all of these getter methods to return the member variable value. I would suggest that you start by simply hard coding the expected initial values you need to return from these functions and just get these tests to pass. Then later on as you are forced to implement more, you will add in the actual code you will need in these methods. Most of these methods are used for debugging the unit tests, so that we can query different properties of the current state of your simulation and see if they return the expected value or not.

2. Implement the `newEvent()` function. The `newEvent()` function is called whenever a "new" occurs in the simulation. Basically you need to create a new process, assign it the correct next process id, make the process ready, and add it to the end of your ready queue. I would suggest again you work on implementing code to get the unit tests to pass in the order given in the third unit test. For example, just get the check of the `sim.getNextProcessId()` == 2 to work first by defining a member variable in your `ProcessSimulator` that keeps track of the next process id that will be assigned and returns it in this function. You will want to use the constructor for the `Process` and the `ready()` member function of the `Process` in your implementation of `newEvent()`.

3. Implement the `dispatch()` function. There are two actions that don't directly correspond to explicit events in our simulation. Later on when we get to implementing the whole simulation, the `dispatch()` should basically occur before you process the next explicit event of the simulation (and the `timeout()` will always occur after you process each explicit event). The first unit test of `dispatch()` are where you may need to implement a real ready queue (you could probably fake it or ignore it through the previous unit tests). Before you work on defining a queue structure for your ready queue, you will need to define some mechanism by which you keep

track of whether or not the CPU is currently idle or is currently running a process, and if it is running a process you need to know which process is currently running on the CPU.

4. Implement basic `cpuEvent()` CPU cycles. The `cpuEvent()` is relatively simple. The system time should be incremented by 1 every time a CPU event occurs. Also, if a process is currently running on the CPU, its `timeUsed` should be incremented by 1 and its `quantumUsed` as well. You should use the `cpuCycle()` member function of the `Process` class to do the work needed to increment the time used and quantum used of the current running process.

5. Implement the `timeout()` function. This is the other implicit action needed for your simulation. The basic thing that `timeout()` should do is to test if the quantumUsed of the current running process is equal to or has exceeded the system time slice quantum. If it has, then the process needs to be timed out, which means it goes back to a ready state and is returned back to the tail of the ready queue. You should use the `isQuantumExceeded()` and `timeout()` member functions from the `Process` class in your implementation of the simulation `timeout()` member function.

There is a test case after the `timeout()` test case that does some more extensive testing of a dispatch/cpu/timeout cycle. Hopefully if you implemented these 3 functions well, these tests will be passing as well from your implementations of `dispatch()`, `cpuEvent()` and `timeout()`.

6. Implement the `blockEvent()` simulation function. Besides the round robin scheduling of processes, your simulation will also simulate blocking and unblocking on simulated I/O or other types of events. An event in our simulation is simple, we just abstractly say that some event of a given unique `eventId` will occur, and that processes block until this `eventId` occurs, when they become unblocked. In your simulation, we simplify things and say that only 1 process can ever be waiting on any particular `eventId`. In some real systems it is possible for 1 event to cause multiple processes to become unblocked, but we will not implement that idea here.

The `blockEvent()` function should put the current running process into a BLOCKED state, and should record the `eventId` that the process is now waiting on. You should use the `block() Process` member function in your implementation of `blockEvent()`.

7. Implement the `unblockEvent()` simulation function. You would not need this for the previous unit test, but now you need to have some way to find out which process is blocked waiting on a particular `eventId` to occur. You could just do a simple search of your process list to find the blocked process waiting on the particular `eventId`. In the example solution I will post after this assignment, I used an STL map, to map from an `eventId` to a process id, and thus be able to directly query the map to find which process should be unblocked when an `eventId` occurs. However you implement keeping track of the mapping, once you identify the process that should be unblocked, you should use the `unblock()` member function of the `Process` class in your `unblockEvent()` function. You will also need to put the blocked process back onto the tail of the ready queue when it unblocks.

8. Implement the `doneEvent()` simulation function. This function simulates a process finishing and exiting the system. There is no `done()` function in the `Process` class, though you could add one if you think you need it. But for a done event, you can simply remove the process from the list of active processes (for example take it out of your process list).

System Tests: Putting it all Together

Once all of the unit tests are passing, you can begin working on the system tests. Once the unit tests are all passing, your simulation is actually working correctly. But to test a full system simulation we have to add some output to the running simulator.

I will give up to 5 bonus points for correctly adding the output and getting all of the system tests to pass as well for this assignment. For the `ProcessSimulator`, you have already been given the implementation of the `runSimulation()` function that is capable of opening one of the process event simulation files, reading in each event, and calling the appropriate function you implemented above while working on the unitTests.

As with the previous assignment, the assg02-sim.cpp creates program that expected command line arguments, and it uses the `ProcessSimulator` class you created to load and run a simulation from a simulation file. The command line process simulator program expects 2 arguments. The first argument is the setting for the system time slice quantum to use. The second is the name of a process events simulation file to load and run. If the sim target builds successfully, you can run a system test of a process simulation manually by invoking the sim program with the correct arguments:

```
$ ./sim
Usage: sim timeSliceQuantum events-file.sim
Run process simulation on the given set of simulated process events file

timeSliceQuantum   Parameter controlling the round robin time slicing
                     simulated by the system.  This is the maximum
                     number of cpu cycles a process runs when scheduled
                     on the cpu before being interrupted and returned
                     back to the end of the ready queue
events-file.sim    A simulation definition file containing process
                     events to be simulated.
```

So for example, you can run the simulation from the command line with a time slice quantum of 5 on the first event file like this:

```
$ ./sim 5 simfiles/process-events-01.sim
--------------------------------------------------------------------------------
Event: new

<Simulation> system time: 1
  timeSliceQuantum      : 5
  numActiveProcesses    : 1
  numFinishedProcesses  : 0

  CPU
  CPU

  Ready Queue Head
  Ready Queue Tail

  Blocked List
  Blocked List

--------------------------------------------------------------------------------
Event: cpu

<Simulation> system time: 2
  timeSliceQuantum      : 5
  numActiveProcesses    : 1
  numFinishedProcesses  : 0

  CPU
  CPU

  Ready Queue Head
  Ready Queue Tail

  Blocked List
  Blocked List

... output snipped ...
```

We did not show all of the output, the simulation will run to time 16 actually for this simulation. To complete the simulator, you simply need to output the information about which process is currently running on the CPU, which processes are on the Ready Queue (ordered from the head to the tail of the queue), and which processes are currently blocked. If you look at the file named `simfiles\process-events-01-q05.res` you will see what the correct expected output should be from the simulator.

In order to pass the system tests, you will need to do some additional work to output the contents of the CPU, ready

queue and blocked list. You will need to add output to display your ready and blocked list items, since it was left up to you to decide how to implement these data structures. The `Process` class has a defined `operator<<()` that you can reuse to display the state information for your processes. But you will need to add some code in the `toString()` method of the `ProcessSimulator` to display the contents of your CPU, ready queue a blocked list.

For example, lets say you used a simple integer called `cpu` that holds the pid of the process currently running on the CPU. Lets further say you have a vector or a regular C array of Process items to represent your process control block, and you index into this array using the pid. Then you could output the current running process on the CPU with code similar to this in your `toString()` method.

```
// Assumes processControlBlock is a member variable, and is an array or a
// vector of Process objects that you create when a new process is simulated
// Further  assumes the member variable cpu holds the pid of the running process

// first check and display when cpu is idle
if (isCpuIdle() )
{
  stream << "   IDLE" << endl;
}
// otherwise display process information using overloaded operator<<
else
{
  Process p = processControllBlock[cpu];
  stream << "    " << p << endl;
}
```

You would need to add something like this so that the process that is on the CPU is correctly displayed in the simulation output. Likewise you need to do similar things to display the processes on the ready queue and the blocked list, though of course you will need loops to go through and output/dispaly all such processes in either of these states in the appropriate output location.

If you get your output correct, you can see if your system tests pass correctly. The system tests work simply by doing a `diff` of the simulation output with the correct expected output for a simulation. You can run all of the system tests like this.

```
$ make system-tests
./run-system-tests
System test process-events-01 quantum 03: PASSED
System test process-events-01 quantum 05: PASSED
System test process-events-01 quantum 10: PASSED
System test process-events-02 quantum 03: PASSED
System test process-events-02 quantum 05: PASSED
System test process-events-02 quantum 10: PASSED
System test process-events-03 quantum 05: PASSED
System test process-events-03 quantum 15: PASSED
System test process-events-04 quantum 05: PASSED
System test process-events-04 quantum 11: PASSED
================================================================================
System test failures detected (5 tests passed of 10 system tests)
```

The most common reason that some of the system tests will pass but some fail is because the output of the processes on the blocked list is not in the order expected for the system tests. The processes on the ready queue need to be listed in the correct order, with the process at the front or head of the queue output first, down to the tail or back of the queue as the last process.
Likewise the system tests expect blocked processes to be listed by pid, so that the smallest blocked proces by pid is listed first, then the next pid, etc. I consider it mostly correct (4/5 bonus points) if the only failing system tests are failing because you do not correctly order the output of the blocked processes. But it is definitely incorrect to not order the ready processes by the ready queue ordering, so issues with the ready queue ordering mean few or not bonus points for this part.

Assignment Submission

In order to document your work and have a definitive version you would like to grade, a MyLeoOnline submission folder has been created named Assignment-02 for this assignment. There is a target in your `Makefile` for these assignments named `submit`. When your code is at a point that you think it is ready to submit, run the submit target:

```
$ make submit
tar cvfz assg02.tar.gz ProcessSimulator.hpp ProcessSimulator.cpp
  Process.hpp Process.cpp ProcessState.hpp ProcessState.cpp
ProcessSimulator.hpp
ProcessSimulator.cpp
Process.hpp
Process.cpp
ProcessState.hpp
ProcessState.cpp
```

The result of this target is a tared and gziped (compressed) archive, named `assg02.tar.gz` for this assignment. You should upload this file archive to the submission folder to complete this assignment. I will probably be also directly logging into your development server, to check out your work. But the submission of the files serves as documentation of your work, and as a checkpoint in case you keep making changes that might break something from when you had it working initially.

Requirements and Grading Rubrics

Program Execution, Output and Functional Requirements

1. Your program must compile, run and produce some sort of output to be graded. 0 if not satisfied.
2. 12.5 pts each (100 pts) for completing each of the 8 listed steps in this assignment to write the functions needed to create the `ProcessSimulator`.
3. +10 bonus pts if all system tests pass and your process simulator produces correct output for the given system tests.

Program Style and Documentation

This section is supplemental for the second assignment. If you use the VS Code editor as described for this class, part of the configuration is to automatically run the `uncrustify` code beautifier on your code files everytime you save the file. You can run this tool manually from the command line as follows:

```
$ make beautify
uncrustify -c ../../config/.uncrustify.cfg --replace --no-backup *.hpp *.cpp
Parsing: HypotheticalMachineSimulator.hpp as language CPP
Parsing: HypotheticalMachineSimulator.cpp as language CPP
Parsing: assg01-sim.cpp as language CPP
Parsing: assg01-tests.cpp as language CPP
```

Class style guidelines have been defined for this class. The `uncrustify.cfg` file defines a particular code style, like indentation, where to place opening and closing braces, whitespace around operators, etc. By running the beautifier on your files it reformats your code to conform to the defined class style guidelines. The beautifier may not be able to fix all style issues, so I might give comments to you about style issues to fix after looking at your code. But you should pay attention to the formatting of the code style defined by this configuration file.

Another required element for class style is that code must be properly documented. Most importantly, all functions and class member functions must have function documentation proceeding the function. These have been given to you for the first assignment, but you may need to provide these for future assignment. For example, the code documentation block for the first function you write for this assignment looks like this:

```
/**
 * @brief initialize memory
 *
 * Initialize the contents of memory.  Allocate array larget enough to
```

```
 * hold memory contents for the program.  Record base and bounds
 * address for memory address translation.  This memory function
 * dynamically allocates enough memory to hold the addresses for the
 * indicated begin and end memory ranges.
 *
 * @param memoryBaseAddress The int value for the base or beginning
 *   address of the simulated memory address space for this
 *   simulation.
 * @param memoryBoundsAddress The int value for the bounding address,
 *   e.g. the maximum or upper valid address of the simulated memory
 *   address space for this simulation.
 *
 * @exception Throws SimulatorException if
 *   address space is invalid.  Currently we support only 4 digit
 *   opcodes XYYY, where the 3 digit YYY specifies a reference
 *   address.  Thus we can only address memory from 000 - 999
 *   given the limits of the expected opcode format.
 */
```

This is an example of a `doxygen` formatted code documentation comment. The two `**` starting the block comment are required for `doxygen` to recognize this as a documentation comment. The `@brief`, `@param`, `@exception` etc. tags are used by `doxygen` to build reference documentation from your code. You can build the documentation using the `make docs` build target, though it does require you to have `doxygen` tools installed on your system to work.

```
$ make docs
doxygen ../../config/Doxyfile 2>&1
  | grep warning
  | grep -v "\file statement"
  | grep -v "\pagebreak"
  | sort -t: -k2 -n
  | sed -e "s|/home/dash/repos/csci430-os-sims/assg/assg01/||g"
```

The result of this is two new subdirectories in your current directory named `html` and `latex`. You can use a regular browser to browse the html based documentation in the `html` directory. You will need `latex` tools installed to build the `pdf` reference manual in the `latex` directory.

You can use the `make docs` to see if you are missing any required function documentation or tags in your documentation. For example, if you remove one of the `@param` tags from the above function documentation, and run the docs, you would see

```
$ make docs
doxygen ../../config/Doxyfile 2>&1
  | grep warning
  | grep -v "\file statement"
  | grep -v "\pagebreak"
  | sort -t: -k2 -n
  | sed -e "s|/home/dash/repos/csci430-os-sims/assg/assg01/||g"

HypotheticalMachineSimulator.hpp:88: warning: The following parameter of
HypotheticalMachineSimulator::initializeMemory(int memoryBaseAddress,
    int memoryBoundsAddress) is not documented:
  parameter 'memoryBoundsAddress'
```

The documentation generator expects that there is a description, and that all input parameters and return values are documented for all functions, among other things. You can run the documentation generation to see if you are missing any required documentation in you project files.