# Assignment 02: Process State Simulation

CSci 430: Introduction to Operating Systems

## Objectives

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.

## Description

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.

# Overview and Setup

For this assignment you will be implementing missing member methods of the `ProcessSimulator.[hpp|cpp]` class. As usual before starting the assignment tasks proper, you should make sure that you have completed the following setup steps:

1. Accept the assignment and copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment 02 Process Simulator' for our current class semester and section.
2. Clone the repository using the SSH URL to your host file system in VSCode. Open up this folder in a Development Container to access and use the build system and development tools.
3. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor. Confirm that you C++ Intellisense extension is working, and that your code is being formatted according to class style standards when files are saved.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also make sure you are linking each issue you create with the `Feedback` pull request for the assignment.

# Assignment 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` 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 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 the 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 mostly 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 defines all of the information you will need to keep track of about processes being managed by the simulator. 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. You should only need to use the public functions given for the `Process` class to create and manage the processes in your simulation.

As a starting point, just like in assignment 01, you should begin with the unit tests given to you in the `assg02-tests.cpp` file. The first test case (`task0`) 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 (`task1`), 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.

## Task 1: Implement `ProcessSimulator` Constructor and Accessor Methods

### Task 1.1 Implement Constructor and Getter Methods

As usual for these assignments, start by defining the `task1` test in the `assg02-tests.cpp` unit test file. For the first task of this assignment, the constructor and getter methods have already been declared for you and are all stub functions, you only need to finish them. You will need to declare and add functions starting with `task2`.

You should start `task1` by getting the initial getter function tests to work in the first section for `task1`. We did not give you the implementation of the constructor for the `ProcessSimulator` class, so you will need to start with a constructor that initializes the system time slice quantum. Most all of the other member variables (except the stl containers) need to be initialized to default values in the constructor. This includes the `cpu` member variable, which

should be initialized to `IDLE` for the simulator. The tests for `task1` show what the expected default starting values should be for many of the member variables.

You should be able to pass the first test for `task1`, once you initialize things properly in the constructor, by implementing the `getTimeSliceQuantum()` member function.

Once you have the test for the time slice quantum working, initialize the next process id and the system time to 1 in the constructor. Then implement `getNextProcessId()`, `getSystemTime()` and `getRunningProcessId()` getter methods.

**NOTE**: You should initialize all of simple member variables in this step. The `map`s and `list` will be initialized as empty containers with nothing in them, so you don't need to do anything to them just yet. But all other member variables should be initialized, and if they have a getter method you should check that it is working. In particular, the `cpu` member variable also needs to be initialized to get the `task1` tests to pass before moving on to the next step.

### Task 1.2 Process Control Block accessors

For the second section in `task1`, concentrate on the number of active and finished processes information for the system.

We will be using an stl `map` for our process control block. In the `ProcessSimulator.hpp` private member variables, you will find this member variable declaration:

```
map<Pid, Process> processControlBlock;
```

An stl `map` is a data structure that maps keys to values. Maps are also known as dictionaries or hashes in other programming languages. Here we use the process identifier `Pid` as the key, to be able to look up and access the `Process` object that has that identifier. In practical terms, you can use this map as an array. For example, if you want to access the process with an identifier of 1, and send it a message to make itself ready, you could do this (assuming the process is in the map):

```
Pid pid = 1;
processControlBlock[pid].ready();
```

The `processControlBlock` is an stl data structure. You should look up the documentation of what methods you can use on a `map`. Implement the method called `getNumActiveProcesses()`. The `size` of the `processControlBlock` will correspond to the number of processes currently being managed in the table. Thus you simply need to return the current `processControlBlock`'s size to determine the number of active processes in the system.

Also, when a process is finished in our simulation, we will remove it from the `processControlBlock`. But we won't keep a corresponding map or list of finished processes, we will simply keep track of the number of processes that have finished so far. So you should initialize the member variable named `numFinishedProcesses` to 0 in the constructor, then create a method called `getNumFinishedProcesses()` that returns this member variable when we need to query the simulation to find out the number of processes that have completed so far.

### Task 1.3: Ready Queue member methods

For the next section for `task1`, lets implement some member methods to query the ready queue for the simulation.

In the `ProcessSimulator` header file, you will also find a declaration of the member variable we will use for the ready queue:

```
list<Pid> readyQueue
```

We will be using an stl `list` as a queue, even though there is an actual `queue` type defined in the stl. We can treat a `list` like a queue by pushing new or timed out processes to the back of the list, and dequeuing processes at the front of the list when we need to dispatch a new process. We use a `list` to make it easier later on to iterate over all of the processes currently on the ready queue, which you cannot do with an stl `queue` data structure.

Finish the method named `readyQueueSize()`. This simply reports the number of processes currently on the ready queue. Query the `readyQueue` to determine and return its `size` from this method.

Then in addition, you need to implement two member methods that will return the `Pid` of the process currently at the front and the back of the queue respectively. A normal stl `list` is a bit unsafe, if you ask for the front item from

an empty list, your program will crash. It does not error check for you that you are trying to remove an item from an empty list.

So first implement the `readyQueueFront()` member method. This method should return the `Pid` of the process at the front of the list. But first check to see if the list is empty, and if it is, return `IDLE` instead of having your program crash.

Likewise, also implement a `readyQueueBack()` member method that will return `IDLE` if the queue is empty, but will return the current back item of the queue if there are 1 or more process identifiers currently in the queue.

## Task 1.4

There are two more sections left in `task1`, but this is the final bit of getter/test methods you need to complete in the first task.

You should be familiar with the requirement here by now. The final data structure in your `ProcessSimulator` is another map named `blockedList`, which will keep track of all of the processes currently blocked and waiting for some event to occur. You need to finish the `blockedListSize()` member function here, which should query the `blockedList` to determine the number of processes that are currently in the blocked state.

There is one final section here which calls another method mostly for testing named `isInState()`. You might want to read over this function before moving on, as it might fail in future tasks if you implement something correctly. At this point in the `task1` tests we have simply initialized the simulator. So this function tests that the simulator has been initialized with the expected time slice quantum of 5, and that the system time, number of active processes, etc. are as expected at the start of a simulation. Later on once you start implementing the simulation events, if something is broken this method will fail if the expected state does not match what should happen in the simulation.

At this point, if you are passing all of the sections for the `task1` tests and are satisfied, remember that you need to make a commit and push it to github of your Task 1 work. Do that now and check the autograder results before moving on to `task2`.

## Task 2: Implement the `newEvent()` member method

Starting with `task2` you will need to again declare the functions you write in the header file, and put their implementations in the `.cpp` source file. Make sure that you find the function documentation for each process and put the implementation immediately below its documentation.

There are 2 functions that are tested in the `task2` test case sections, `newEvent()` and `getProcess()`. After defining the `task2` tests, you should create stub/empty function declarations for both of these and make sure your code still compiles and runs the tests. The `newEvent()` function has a simple signature, id doesn't take any input and it is a void functions, so you don't have to return anything. The `getProcess()` function needs to return a reference to a `Process` instances that you will look up from your process control block. So we will just give you the signature for this function that you need here:

```
Process& getProcess(Pid pid);
```

Notice the function returns a reference to a `Process` instance. You ultimately need to be looking up and returning a reference to a process in the process control block. To get your stub function to compile, you could for example just start by hardcoding it to return the reference to the process with a `Pid` of 0:

```
return processControlBlock[0];
```

### Task 2.1: Implement `newEvent()` member method

Implement the `newEvent()` function. The `newEvent()` function is called whenever a "new" occurs in the simulation.

You need to ultimately do all of the following in this function:

1. create an instance of a new process using the next process id for the simulation.
2. increment the next process id in anticipation of the next process arriving
3. Put the new process into the `READY` state.
4. Push the process id of this new process onto the back of the ready queue.
5. Add the new process to the process control block.

If you look closely at the first section for task 2, you will see that it tests that the process control block now has 1 process in it (the number of active processes is now 1), and that the next process id has been incremented. So to get the first section to pass, you only need to do the steps 1, 2 and 5 at this point.

When you create a new process, use the constructor that takes the process identifier and the system time as its parameters for the newly created process. You should use the next assigned pid, and the current system time for the new process. Also be careful when incrementing the process id, you need to process id assigned to your new process as your key into the `processControlBlock`.

#### Task 2.2: Implement the `getProcess()` accessor

You will also implement the `getProcess()` member function after you get your basic `newEvent()` working. The `getProcess()` function is used mainly for testing, its purpose is to access the `processControlBlock` and return the `Process` that has a particular process identifier (`Pid`). You will need to look up the `Process` in your `processControlBlock` to implement this function.

If you are following along, you should already have a stub function that just tries to return the process with a pid of 0 when called. But now in the second section of the test, we expect that there is a process with an pid of 1 in the table. Implement the `getProcess()` function to correctly look up and return the process reference for a given pid from your `processControlBlock`.

**Note**: it would be prudent if this function were actually used in the simulation to check that the `pid` exists in the process control block, and if not throw an exception. However this function is really only used for testing, so you don't have to be defensive here if you don't want too.

### Task 2.3: Put new processes in Ready state

As noted for the first part of this task, the new process also needs to be put into the `READY` state, and its pid added to the back of the ready queue. The last 2 sections of the task 2 tests check that you do this for a new process.

To make a process ready, you need to call the appropriate member function on your process instance, read the `Process.hpp` header file for the public member functions you can invoke on an instance of a process.

Also you should notice that the `readyQueue` is not a list of processes, it is actually maintained as a list of `Pid` process identifiers. So you need to push the process identifier of the newly created process onto the back of the ready queue here.

Both of these need to be done before you insert the new process into the process control block, or else your changes may be done on your local copy of the process instead of being done to the process that actually ends up being placed into your PCB.

If you get all 5 parts working, you should not find that all of the sections of the `task2` tests are now passing. When you are satisfied with your work, create a commit of your `task2` implementation and push it to your GitHub classroom repository to be evaluated.

## Task 3: Implement the 'dispatch()' member method

As with the previous task there are more than 1 methods that need to be declared and created for task 3. So start by defining the `task3` tests, and add in stub functions so that your code can compile and run the tests as you work on implementing the functions.

You will be reusing several functions you already have here, including the `getRunningProcessId()` function. However, you need to declare a `isCpuIdle()` and the `dispatch()` member functions here. The `dispatch()` function like the `newEvent()` takes no input parameters and is a void function, so you can just use an empty function implementation for your stub function. `isCpuIdle()` returns a bool result, so just return true or false as your stub implementation of this function.

#### Task 3.1: Keep track of the running process

For the first section of `task3` we are expecting that the simulated cpu is initially IDLE and no process is currently running. Your previous implementation of `getRunningProcessId()` should be working here. Implement your `isCpuIdle()` to return a true result if the cpu is currently `IDLE`, and false if not.

**Task 3.2: Implement `dispatch()`**

Between the first and second section of the `task3` tests, we actually call your `dispatch()` function, so you need to implement it as follows.

The `dispatch()` function has no parameters for input and it is a `void` function that returns no result.

The basic pseudocode of what the `dispatch()` should do is as follows. You should reuse your functions, like `isCpuIdle()` instead of reimplementing the same logic in your code (Don't Repeat Yourself DRY principle), so here you should be using that function for the first check, and one of your ready queue functions for the second check.

```
// first check if something is currently running, we only dispatch
// something new when cpu is not currently running anything
if cpu is not idle:
   do nothing

// otherwise cpu is idle, so try and dispatch a process, but if
// ready queue is empty, we still can't dispatch the next process
if ready queue is empty:
   do nothing

// otherwise cpu is idle and there is one or more process ready to run,
// so get the process at front of queue and make it the running process
pid = ready queue front process
cpu = pid
look up process in the processControlBlock and put it in the running state (e.g. dispatch it)
```

Don't forget that you need to remove the front pid item from your ready queue when you dispatch a process here. If this is implemented correctly, you should find that all of the remaining sections for `task3` pass, as they check what happens if other processes are created and subsequently dispatched using your function here. Also don't forget that when you look up the process in your process control block, you should call the appropriate process member method to put the process into the running state.

When you are satisfied with your implementation, make sure to create a commit of your task 3 work and push it to your GitHub classroom repository.

## Task 4: Implement the `cpuEvent()` member method

Implement basic `cpuEvent()` to simulate the cpu running cycles. The `cpuEvent()` is relatively simple. As with most of the event simulation methods, it is a `void` function that takes no parameters as input.

We simulate work being done by incrementing the system time, and also keeping track of how many time slices each process has run in its current time slice quantum. 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.

So the pseudocode for the `cpuEvent()` function is:

```
// increment the system time
systemTime++

// if the cpu is running a process, call the cpuCycle member function of the
// process to update its time used and quantum used
if cpu is not idle
  access the process from the processControlBlock
  call the cpuCycle() method on the process that is currently running
```

Once you are satisfied with your work on `task4`, push your commits to GitHub and continue on to the next task.

## Task 5: Implement the `timeout()` member method

Implement the `timeout()` function to check if the current running process has exceeded running its time slice quantum. As with your previous simulation event functions, the `timeout()` is a `void` function, that does not take any parameters as input.

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.

The pseudocode for the `timeout()` function is:

```
// if cpu is idle, no process to check currently
if cpu is idle
    return

// otherwise check the current running process
look up running process in the processControlBlock
if process time slice quantum is exceeded:
   timeout the process
   put the process on the back of the ready queue
   set the cpu to IDLE
```

Once you are satisfied with your work for `task5`, push your commits to GitHub and continue on to the next task.

## Task 6: Implement the `blockEvent()` member method

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.

The `blockEvent()` function is a `void` function as before, but `blockEvent()` does take a parameter. It takes an `EventId` parameter, which is simply a `typedef` for an `int`. This parameter is the identifier of the event that the current running process is being blocked upon.

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. In our simulation it doesn't make sense for a block event to occur when the cpu is `IDLE`. So the first thing you need to do is check if the cpu is `IDLE` and if it is you should throw the expected `SimulatorException`.

You should use the `block()` `Process` member function in your implementation of `blockEvent()`.

Also, you need to add the mapping into the `blockedList` of the `EventId` to the `Pid`, so that you can look up the process that needs to be unblocked in the next step. However, we also consider it an error in this simulation to have more than 1 process blocked on a given event. So if there is already a process waiting on the event that just occurred, you should throw an exception. **HINT**: The `count()` member method of the stl `map` is probably the easiest way to check for this.

And finally, since the current running process just blocked, don't forget to set the cpu to the `IDLE` state.

The pseudo code for the `blockEvent()` is then:

```
# test for errors first, should be an error if cpu is IDLE when a
# block occurs
if cpu is IDLE
   throw SimulatorException
```

```
# it is also an error if there is already a process waiting on
# the event that just happened
if blockedList already has a process waiting on this event
   throw SimulatorException

# otherwise there is a process running and we can block it on
# the indicated event
Look up the current running process in the processControlBlock and block it
Enter the mapping between the event and this process in the blockedList
Set the cpu to IDLE
```

Once you are satisfied with your work for `task6`, push your commits to GitHub and continue on to the next task.

### Task 7: Implement the `unblockEvent()` member method

Implement the `unblockEvent()` simulation function. This function will try and unblock the process that is waiting on the event that just occurred. So again, like the previous block function, the `unblockEvent()` is a void function, but it takes an `eventId` as an input parameter.

Unblocks can happen when the cpu is idle, so that is not an error condition here. But again we consider it an error if an unblock occurs for an event aht doesn't have any process currently waiting on it. So again check the `blockedList` first but this time if there is no process waiting on the event, throw an exception.

Otherwise you need to look up the process in the process control block and call `unblock()` on it to unblock the process and make it ready again. But then you also need to put the process back onto the tail of the ready queue. Finally, since there is nothing waiting on this event anymore, you need to remove the mapping from the `blockedList` between this event and the process. **HINT**: there isn't a real simple way to remove a key/value pair from a C++ stl map. You need to use the `find()` member method to get an iterator pointing to the item in the map, then call `erase()` to erase it. Read the `erase()` documentation on cplusplus.com for an example of doing this.

Once you are satisfied with your work for `task7`, push your commits to GitHub and continue on to the next task.

### Task 8: Implement the `doneEvent()` member method

Finally, 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.

It is considered an error in the simulation for a done event to occur if the cpu is IDLE. So first check for this error case.

Then for a done event, you should remove the process from the `processControlBlock` map (same way you removed the event/process pair from the `blockedList`). Also don't forget to set the cpu back to the IDLE state, because the current running process just finished, and also increment the member variable keeping track of the number of finished processes.

Once you are satisfied with your work on the `task8`, push your commits to GitHub and continue on to the next task.

## 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 finish the `runSimulation()` method, and also finish the `toString()` method and add some output when running the simulation..

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 `assg 02-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 = processControlBlock[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/display 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 tests succeeded (10 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 process 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

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 50 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 5 to 10 points are awarded for completing each of the remaining 6tasks. However you should note that the autograder awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

# 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. 40 points for keeping code that compiles and runs. A minimum of 50 points will be given if at least the first task is completed and passing tests.
3. 5 to 10 points are awarded for completing each subsequent task 2-8.
4. +5 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 first assignment. If you uses the VS Code editor as described for this class, part of the configuration is to automatically run the `clang-format` code style checker/formatter on your code files every time you save the file. You can run this tool manually from the command line as follows:

```
$ make format
clang-format -i include/*.hpp src/*.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 large 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 refdocs
Generating doxygen documentation...
doxygen config/Doxyfile 2>&1 | grep -A 1 warning | egrep -v "assg.*\.md" | grep -v "Found unknown command"
Doxygen version used: 1.9.1
```

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 refdocs` 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 refdocs
doxygen config/Doxyfile 2>&1 | grep -A 1 warning | egrep -v "assg.*\.md" | grep -v "Found unknown command"

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.