Assignment 5: Process Scheduler Simulation

CSci 430: Introduction to Operating Systems

Fall 2020

Overview

In this assignment you will be implementing some of the process schedulers that are discussed in our textbook chapter 9. As with the previous assignment, your first task will be to implement some of the missing pieces of the general process scheduling simulator framework, that supports plugging in different job scheduling policies to make dispatching decisions. The simulator will take a table of job arrival information along with the service time (run time) of the jobs, just as is used in our textbook for discussing the various processes scheduling policies. We will be simulating a single CPU system with this process scheduling simulator, so only 1 process will be running at any given time, and the scheduler needs to make a decision when the job finishes or when the job needs to be preempted. The output from the simulator will be a simple sequence of the scheduled processes at each time step of the simulation, along with a final table of statistics with finish time, turnaround time (T_r) and ratio of turnaround time to service time (T_r/T_s).

A working fist come first server (FCFS) scheduler has been given to you already in this assignment. You will be asked to implement one of the other scheduling policies. You can choose to implement a round robin scheduler (RR), shortest process next (SPN) shortest remaining time (SRT), highest response ratio next (HRRN), or a feedback scheduler (FB).

Questions

- How is process scheduling accomplished in the OS.
- What are the similarities in implementation between different process scheduling policies? What are their differences?
- What information does a scheduling algorithm need in order to select the next job to run on the system.
- What are the differences between preemptive and non-preemptive scheduling policies?
- How does the FCFS policy work? How do the other process scheduling policies we discuss work?
- How do the various process scheduling policies compare in terms of performance? How do we measure good or bad performance for a process scheduler?

Objectives

- Implement a process scheduling policy using the process scheduler framework for this assignment.
- Look at use of C++ virtual classes and virtual class functions for defining an interface/API
- Better understand how process scheduling works in real operating systems, and in particular what information is needed to make decisions on when to preempt and which process to schedule next.

Introduction

In this assignment you will be implementing some of the process schedulers that are discussed in our textbook chapter 9. As with the previous assignment, your first task will be to implement some of the missing pieces of the general process scheduling simulator framework, that supports plugging in different job scheduling policies to make dispatching decisions. The simulator will take a table of job arrival information along with the service time (run time) of the jobs, just as is used in our textbook for discussing the various processes scheduling policies. We will be simulating a single CPU system with this process scheduling simulator, so only 1 process will be running at any given time, and the scheduler needs to make a decision when the job finishes or when the job needs to be preempted. The output from the simulator will be a simple sequence of the scheduled processes at each time step of the simulation, along with a final tabla of statistics with finish time, turnaround time (T_r) and ratio of turnaround time to service time (T_r/T_s).

The process scheduler simulator framework consists of the following classes. There is a single class given in the `SchedulingSystem[.hpp|cpp]` source files that defines the framework of the process scheduler policy simulation. This class handles the outline of the simulation framework needed to simulate process scheduling in an operating system. The class has methods to support loading files of job arrival and service time information, or to generate random job arrival tables. This class has a `runSimulation()` function that is the main hook into the simulator. The basic algorithm of the process scheduling simulator is to simulate time steps occurring in discrete time steps. At each time step the scheduling simulator framework determines if a new process arrives, if the current running process has finished, and/or if the current running process should be preempted. All of these events are communicated to the scheduler class, so that it may update its data structures (like ready queues or other information about waiting processes). If the cpu is idle at the beginning of the time step, it asks the scheduler class to make a process scheduling decision, to select a process to begin running on the system.

A separate abstract API/base class hierarchy is defined, using the Strategy design pattern, to implement the actual mechanisms for a process scheduling policy. The base class is named `SchedulingPolicy.[cpp|hpp]`. Most of the functions of this class are pure virtual functions, they describe an interface that is used by the `SchedulingSystem` class to interact with a particular process scheduling policy. The first come first serve (FCFS) policy has been implemented already in this assignment, `FCFSSchedulingPolicy[.cpp|hpp]`. You may use this class as a reference when developing a new scheduling policy to add to the framework.

Unit Test Tasks

For this assignment, you will first of all be completing some of the functions in the `SchedulingSystem` class to get the basic simulator running. Then once the simulator is complete, you will be implementing one of the remaining `SchedulingPolicy` to add capabilities to the system.

So for this assignment, as usual, 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 the given order. To get the simulator class completed you will first need to complete the following tasks.

1. The first test case of the unit tests this week tests a few accessor methods that you need to implement. These methods are found in the `SchedulingSystem` class. As a warm up exercise, and to get familiar with this class, start by completing the following accessor methods: `getSystemTime()`, `getNumProcesses()`, `isCpuIdle()`, `getRunningProcessName()`.
2. The `allProcessesDone()` function is important for controlling when the simulation ends. You need to implement this function. as well to get the first unit tests to completely pass. For this function, you need to search the process table. If you find a process that is not yet done (example the `done` member field for each `Process`), then the answer is `false`, not all processes are done. But if all of them are done, you should return true.
3. The `runSimulation()` function, as usual, is the hook into the simulation class to run a whole system simulation. The first function of the simulation, `checkProcessArrivals()` has been given already. It shows an example of sending a message to the policy instance to notify it when new processes arrive. However, the next function `dispatchCpuIfIdle()` has not yet been implemented. If the cpu is idle (use `isCpuIdle()` to test this), then we need to work with our policy to make a dispatching system. The `policy` instance has a method that, when called, will return a `Pid` which will be the process identifier of the process that should next be "dispatched" to run on the cpu. So if the cpu is idle, you should queary the `policy` object in this function to return the identifier of the process to run next. Once you have the `pid`, you should set the `cpu` member variable to be this `pid`, which is equivalent to starting the process running on the `cpu` in thi system. You also need to record the start time for the process. If the process has never run before, its startTime will be `NOT_STARTED`. If the process is running for the first time, make sure you record the current time as the processes `startTime`.
4. The `simulateCpuCycle()` function has been given to you, but you should take a look at it and understand it at this point. To simulate a cpu cycle, we simply increment the time used of the currently running process. We also record the schedule history of which process runs given this the process that is currently running. However, the next process `checkProcessFinished()` needs to be implemented. If the cpu is currently `IDLE` then there is nothing to do, and you should return immediately (look at the next `checkProcessPreemption()` for an example). But if a process is currently running, check the current running process to see if its timeUsed

is equal to or exceeds the `serviceTime` for the process. If you look back at the `simulateCpuCycle()` method, you will see that `timeUsed()` is incremented for a process each cycle it is running on the `cpu`. So in the `checkProcessFinished()` function, you can test if the `timeUsed` has reached the `serviceTime`. When the process is finished, you need to perform 3 tasks.

a. Record the `endTime` for the running process, as it is now finished.
b. Most important, set the process to be `done`. Each `Process` has a member variable named `done` that will be initially `false`. When all processes are marked as done, then the simulation is finished.
c. Also very important, the `cpu` should now be `IDLE`. You should set the simulator `cpu` member variable to the `IDLE` value, so that a new process will be dispatched the next time it is checked to see if the `cpu` is currently idle or not.

Once these 4 tasks are complete, all of the unit tests you were given should now be passing. The `runSimulation()` function has been implemented for you, and it will call the `dispatchCpuIfIdle()` and `checkProcessFinished()` functions you implemented (as well as need to use the getter methods from step 1).

The next step is a bit more open ended. The simulation and unit tests you were given will use the `FCFSSchedulingPolicy` object by default to perform the tests using a first come first server (FCFS) scheduling policy. In the second part of the assignment, you are to implement 1 additional scheduling policy and add it to the simulator. You can choose any of the simulation policies from chapter 9 (besides FCSF), such as a round robin (RR) scheduler, shortest process next (SPN), shortest remaining time (SRT), highest response ratio (HRRN) or a feedback scheduler (FB). You should start by copying the files names `FCFSSchedulingPolicy.[hpp|cpp]` and renaming the file name replacing FCFS with the mnemonic for the scheduling policy you will implement. You should also rename the class names inside of the files as first step towards implementing them.

You will then need to modify the class to implement your given strategy. You can and may need to use STL containers for your policy class. For example, the FCFS class uses a standard STL `queue` instance to represent its ready queue. If you were to choose round robin, for example, you would also need a ready queue, but you would have to add in a bit of extra mechanism to keep track of the running processes being assigned and using its time slice quantum, and to preempt the process when it time slice quantum has been used up, returning it to your ready queue.

System Tests: Putting it all Together

Once all of the unit tests are passing, you can begin working on the system tests.

As with the previous assignment, the `assg05-sim.cpp` creates a sim program that uses command line argument to set up and run a simulation. The paging system simulator is invoked like this:

```
$ ./sim
Usage: sim policy process-table.sim [quantum]
  Run scheduling system simulation.  The scheduling
  simulator reads in a table of process information, specifying
  arrival times and service times of processes to simulate.
  This program simulates the indicated process scheduling
  policy.  Simulation is run until all processes are finished.
  Final data are displayed about the scheduling history
  of which process ran at each time step, and the statistics
  of the performance of the scheduling policy.

Options:
  policy       The page job scheduling policy to use, current
               'FCFS', 'RR', 'SPN' are valid and supported
  process-table.sim  Filename of process table information to
               to be used for the simulation.
  [quantum]    Time slice quantum, only used by some policies
               that perform round-robin time slicing
```

The simulator requires 2 command line arguments to run, and a 3rd optional parameter. You first specify the policy, such as FCFS or RR, and then the location of the process table simulation file. For policies that use time slicing, the system time quantum can/should be specified as well.

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-04 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 assg05.tar.gz FCFSSchedulingPolicy.cpp RRSchedulingPolicy.cpp
    SchedulingPolicy.cpp SchedulingSystem.cpp SPNSchedulingPolicy.cpp
    FCFSSchedulingPolicy.hpp RRSchedulingPolicy.hpp SchedulingPolicy.hpp
    SchedulingSystem.hpp SPNSchedulingPolicy.hpp
FCFSSchedulingPolicy.cpp
RRSchedulingPolicy.cpp
SchedulingPolicy.cpp
SchedulingSystem.cpp
SPNSchedulingPolicy.cpp
FCFSSchedulingPolicy.hpp
RRSchedulingPolicy.hpp
SchedulingPolicy.hpp
SchedulingSystem.hpp
SPNSchedulingPolicy.hpp
```

The result of this target is a tared and gziped (compressed) archive, named `assg05.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. 10 pts all accessor methods from step 1 implemented and first test case passes.
3. 10 pts the `allProcessesDone()` member function is implemented and working correctly.
4. 20 pts `checkProcessArrivals()` implemented and working, test cases using this function are passing.
5. 20 pts `checkProcessFinished()` implemented and working. Test cases using this function are passing.
6. 40 pts Implemented a new process scheduling policy class. Implementation is correct and working. Added unit tests to demonstrate the policy. Added the policy to the system simulation.

Program Style and Documentation

This section is supplemental for the first 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 every time 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.