Assignment 5: Process Scheduler Simulation

CSci 430: Introduction to Operating Systems

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 framework, 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. So the scheduling system does need to support preemptive schedulers, and maintain information about running processes that might be needed to make preemption decisions by the policies.

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 at the start of 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 framework, 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 are classes 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 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 in the second part of this assignment.

Overview and Setup

For this assignment you will be implementing missing member methods of the `SchedulingSystem.[hpp|cpp]` class which controls the main simulation for this assignment. And you will be implementing a new scheduling policy of your choice from scratch. As usual, you need to perform the following steps before beginning work on this assignment.

1. Accept the assignment and copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment 05: Process Scheduler' 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 tests, though no tests may initially be defined for this assignment. If the project does not build on the first checkout, please inform the instructor. Confirm that your 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 (SchedulingSystem Simulation)

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.

Task 1: Implement basic `SchedulingSystem` accessor methods

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 declaring and implementing the following accessor methods:

- `getSystemTime()`
- `getNumProcesses()`

- `isCpuIdle()`
- `getRunningProcessName()`

Remember that all functions are required to have proper Doxygen function documentation, so you need to document the functions when you implement them in the cpp implementation file.

The first 2 methods `getSystemTime()` and `getNumProcesses()` simply access and return the corresponding member variables. These values are already initialized for you in the class constructor when it calls the `resetSystem()` method.

There is some overlap of this assignment with our second assignment for the class, where we implemented a round robin process scheduler. There is a `cpu` member variable that holds an id/index into the `process` table. In this simulation, valid processess identifiers start with 0, so the `cpu` identifier can be used to directly index into the process table (though we use a simple array instead of a map since we read in all of the processes to simulate at the start of a simulation). We define `IDLE` as -1, which is not a valid index into the table.

So to implement `isCpuIdle()` you simply need to test if the `cpu` member variable is `IDLE` or not, and return `true` if it is and `false` if it is not.

To implement `getRunningProcessName()` however, you need to look up the name in the process table. If the `cpu` is `IDLE` you should return a string "IDLE" as these first unit tests expect. However, if the `cpu` is a valid process id (>= 0) then you need to look up the process in the `process` table and return the process name, something like this:

```
return process[cpu].name
```

Accessing and returning an actual process name, as well as testing if the cpu is not IDLE are done more extensively later on in the unit tests, but you should try and get the implementation of the name and testing if the cpu is idle working at this point.

Once you have implemented these accessor methods are are satisfied with your code, commit your work and push your changes to the Feedback pull request for evaluation of your Task 1 work.

Task 2: Implement `allProcessesDone()` member method

The `allProcessesDone()` function is important for controlling when the simulation ends. For this function, you need to search the process table. If you find a process that is not yet done (use 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.

This member function does not take any parameters as input, and it returns a `bool` result. This function does not modifiy the state of the scheduling simulation, so it should be a `const` member function.

So you will primarily be using the `process` table of processes, and the `done` member variable of the processes to implement this, as well as the `numProcesses` to determine how many processes you need to check in the simulation. The pseudocode for this method might look something like:

```
for each process in the process table
    if the process is not done
        return false

otherwise after the loop all processes checked were done so return true
```

Once your implementation is working and passing the tests for task 2, commit your work and push your changes to the Feedback pull request for your GitHub assignment repository.

Task 3: Implement `dispatchCpuIfIdle()` member method

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 decision. 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 this 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`.

This function is a `void` function that has no parameters as input, nor does it return an explicit result. All of its work is done by examing the current state of the cpu, and working with the scheduling policy to dispatch a process if the cpu is idle. This function cannot be a `const` member function, because it can change the state of the simulation by changing the `cpu` from being idle to currently running a process.

The pseudocode for this function might look like the following:

```
if cpu is idle
    ask the policy to dispatch the next process to run
    the dispatch method returns the Pid of the process, set cpu to be this process id
    update the startTime of this process if it was never started before
```

Once your implementation is working and passing the tests for task 3, create and push your commit to your GitHub classroom repository.

Task 4: Implement `checkProcessFinished()` member method

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 function `checkProcessFinished()` needs to be implemented. This is again a `void` funciton that returns no result and takes no parameters as input. This function will actually do the work to mark a process as finished in the process table and record its end time, so it is not a `const` member function.

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.

1. Record the `endTime` for the running process, as it is now finished.
2. 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.
3. 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.

A possible pseudocode implementation of `checkProcessFinished()` is:

```
if cpu is idle
   do nothing and return immediately

if the process that is running's time used is less than its service time
   then process is not done, so do nothing and return immediately

othewise the process is done so
1. record its endTime as the current systemTime
2. mark the process as done
3. set the cpu to be idle again
```

Once your implementation is working and passing the tests for task 4, create a commit and push it to the Feedback pull request in your GitHub classroom repository.

Task 5: Enable full FCFS Simulations

As with the previous assignment, some of the functions you implemented in tasks 1-4 before are needed in order for full SchedulingSystem simulations to work. Though in this case, there is only 1 place that you need to uncomment in the `runSimulation()` main method. Uncomment the main loop there that calls the methods needed to simulate the scheduling system, and define the task 5 tests. Full simulations using FCFS should now pass in the task 5 unit tests, as well as in the system tests now when you run them.

As usual, make a commit once you have the full simulations enabled and are passing the task 5 unit tests.

Assignment Tasks (Implement Page Replacement Scheme of your choice)

Once these 5 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.

You should make multiple commits while implementing your choosen scheduling policy. You need to do at least the following steps

Copy FCFSSchedulingPolicy header and implementation files

Copy over the `FCFSSchedulingPolicy` header file and implementation file into new files, and rename the file based on the scheduling policy you have choosen to implement. For example, if you are going to implement Highest Response Ratio Next (HRRN), you should end up with a file named `HRRNSchedulingPolicy.hpp` in the `include` directory, and `HRRNSchedulingPolicy.cpp` in the `src` directory. Use abreviations SPN, SRT, RR, HRRN, FB for shortest process next, shortest remaining time, round robin, highest response ratio and feedback schedulers respectively.

After copying the FCFS files to the new policy you are creating, do a global search and replace to change the name of the class. For example, if you are using HRRN, you should search for `FCFSSchedulingPolicy` in your copied files and replace it with `HRRNSchedulingPolicy`. Also in the header file there is an `#ifdef` to guard from multiple inclusions of the class that looks like this:

```
#ifndef FCFSSCHEDULING_POLICY_HPP
#define FCFSSCHEDULING_POLICY_HPP
```

You should modify this to be `HRRNSCHEDULING_POLICY_HPP` in all caps.

Next you need to add this new file to the build system. There are lines commented out in the `Makefile` for each of the scheduling policies for the `assg_src` makefile variable. Add your source file to this list of files that is compiled. Be careful, all lines but the last line must have the `\` to continue the line, so if you are uncommenting to add a new line, you need to also put the `\` on the previous line.

Once you have done this, your file should compile and be linked into the test and sim executables when the compiler and linker run. Your file is still implementing FCFS policy at this point, but it is an important milestone that you can build with your new policy class files and run the unit and system tests. You should make a commit for task 6 here once you get to this point and can build code again with your new policy as part of the build system.

Implement the SchedulingPolicy API Methods

You should next attempt to implement the API methods for your new scheduling policy one by one. You will also probably need to define some member variable data structures, like queues or lists, to hold information that your policy needs to make the dispatching scheduling decision.

You should probably implement the methods in this order, and it would be best to make a commit of each one when you think it is (mostly) working:

- `resetPolicy()`
- `preempt()`
- `newProcess()`
- `dispatch()`

If you copied over the FCFS implementation to start work on your new policy as suggested, you will have the implementations of the FCFS scheduling policy to begin with. You will need to remove these and implement each method for your new policy. Make sure that you also read the function documentation and modify it when doing this step, to make sure that it correctly reflects the implementation of your choosen policy.

Briefly here are what each of these methods is supposed to do:

`resetPolicy()` is called when a new simulation is created/reset. You should initialize any member variables used by your policy implementation here. For example FCFS uses a simple queue of processes to keep track of which current ready process arrived first. So `resetPolicy()` simply ensures that this queue is empty in its implementation.

`preempt()` is called by the `SchedulingSystem` for each simulated time cycle. It is the job of the scheduling policy to tell the scheduling system if it is time to preempt the current running process. FCFS is a nonpreemptive policy, so you will see it always returns false. But for a preemptive policy, for example a RR round robin scheduler, processes can be preempted when running. So if your choosen policy is preemptive, you need to decide if it is time to preempt the current running process, and if so you should return `true` instead of `false` when asked if preemption should occur or not.

`newProcess()` is called anytime a new processes enteres the scheduling simulation. Most scheduling policies need to know when a new process arrives, in order to keep track of it, note that it is currently ready and waiting, and do other things. For example, for FCFS, we keep track of new arriving processes by simply adding them onto the back of the ready queue of processes that this policy maintains.

`dispatch()` finall is called to make the actual dispatching decision, e.g. to decide which process to run next. `dispatch()` will be called by the scheduling simulator anytime the cpu is `IDLE` at the beginning of a simulation cycle. Your scheduling policy needs to determine which of the currently available and waiting processes should be scheduled next to run. This method returns the `Pid` process identifier of the process to be scheduled to run. For FCFS the dispatching decision is simple, whichever process that is ready and is at the front of the `readyQueue` (because it has been waiting the longest) should be selected to be scheduled next.

Again you should probably make individual commits for the implementation of each of these methods.

You might find it useful to add additional unit tests while working on these implementations. You can add in unit tests to the `assg05-tests.cpp` file for this assignment to test the implementation of your functions if you wish (normally you are not allowed to add or modify tests in this file). If so, create a `task6` test declaration, and create `SchedulingSystem` instances that use your new scheduling policy as the scheme for the simulation.

Enable and Test Full Scheduling System Simulations

Finally make sure that you enable full scheduling system simulations using your new scheduling policy. You will need to add code into the `assg05-sim.cpp` file to allow for your new scheduling policy to be selected from the command line. Read the code and add/uncomment the necessary lines for the scheduling policy you are implementing.

You should test your code with the given system tests at a minimum. There are defined system tests for the schedulers already in the `simfiles` directory.

You can and should remove the system tests from the `scripts/run-system-tests` for scheduling policies that you do not implement. Just leave in the system tests of the given FCFS scheduler, and the scheduling policy you implement. If you get your system tests to pass, and remove tests of other policies that you don't implement, you should be able

to get the final 20 points awarded to you in GitHub when you push a commit that is passing all of the system tests for FCFS and your scheduling policy.

System Tests: Putting it all Together

As with the previous assignment, the `assg04-sim.cpp` creates a program that expects command line arguments, and it uses the `PagingSystem` and `ClockPageReplacementScheme` classes you created to load and run a simulation from a simulation file. The command line simulation expects 3 parameters as input, the page replacement scheme to use, the size of memory to simulate, and the input page reference stream file:

```
$ ./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
```

So for example, you can run the simulation using a fifo page replacement scheme and a memory size of 3 for the first page reference stream like this:

```
$ ./sim

TBD
```

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
./scripts/run-system-tests

TBD
```

Assignment Submission

For this class, the submission process is to correctly create a pull request with changes committed and pushed to your assignment 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 10 to 15 points are awarded for completing each of the 4 tasks. 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. An initial 33 points of the assignment is reserved for having a final commit that compiles and runs the unit tests. If you have made an attempt at task 1, and your code is compiling and running, you will receive the baseline number of points.
3. You will receive a total of at least 50 points if your program compiles and runs, and the preliminary getter functions for task 1 are all working and pass the unit tests.
4. Up to 75 points can be received then for completing tasks 2-4, which complete the `SchedulingSystem` simulation member methods.
5. 20 more points, for a total of 95 points, will be awarded for completing a process scheduling scheme of your choice, including modifying the build system to build your scheme and integrate it into the full simulation.
6. A final 5 points, for a total score of 100, will be awarded for completing all of the tasks and getting system tests to pass, including adding in at least 1 system test of your new process scheduling scheme to the 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 `.clang-format` file defines a particular code style, like indentation, where to place opening and closing braces, whitespace around operators, etc. By running the code formatter on your files it reformats your code to conform to the defined class style guidelines. The code style formatter / checker 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.