Assignment 4: Page Replacement Schemes, Clock Algorithm

CSci 430: Introduction to Operating Systems

Overview

In this assignment you will be implementing some pieces of a memory paging system simulator framework, and you will be implementing a Clock page replacement algorithm. The simulator will allow you to take a stream of page references, like we have done by hand in our written assignments and in class. The simulator will simulate a physical memory of some number of fixed size frames (e.g. a paging system), where placement and replacement decisions are made and the content of the physical memory frames will change in response to the stream of page references being simulated.

Questions

- What are the basic steps an OS needs to perform in order to manage a paging memory system and make placement and replacement decisions?
- What are the similarities in implementation between different page replacement algorithms? What are their differences?
- What information does a page replacement scheme need to keep track of to make a page replacement decision?
- How does the clock page replacement scheme work? How is it implemented?
- How does a clock scheme perform compared to a FIFO scheme?

Objectives

- Implement a clock paging scheme by hand within a paging system simulator framework.
- Review use of C++ virtual classes and virtual class functions for defining an interface/API
- Better understand how paging systems work, and in particular what information is needed to be tracked to make page replacement decisions.

Introduction

In this assignment you will be implementing some pieces of a memory paging system simulator framework, and you will be implementing a Clock page replacement algorithm. You will be given an implementation of the simple FIFO page replacement scheme, described in chapter 8 of our textbook. You will be implementing a simple version of the Clock page replacement scheme, using a normal frame pointer and a single use bit to approximate usage information for pages being used in the paging system.

The paging system simulator framework consists of the following classes. There is a single class given in the `PagingSystem.[hpp|cpp]` source files that defines the framework of the paging system. This class handles the outline of the algorithm needed for a paging system, and has methods to support loading files of page stream information, or to generate random page streams, to use in simulations. This class has a `runSimulation()` function that is the main hook into the simulator. The basic algorithm of the paging simulator is to process each new page reference. For each new page reference, we first determine if the reference is a "hit" or a "fault". If it is a hit, then nothing further needs to be done except to update any system usage statistics.

For a page fault, the referenced page needs to be loaded into memory. When a page fault occurs, if memory is not yet full, a simple placement decision is made (using the `doPagePlacement()` function). Page placement in this simulation is simple, the first empty physical frame of memory will always be selected to place the new page reference into. When memory is full, a page replacement decision needs to be done first. The page replacement decision is handled by the `doPageReplcement()` and `makeReplacementDecision()` functions.

However, implementation of page replacement decisions are done by a separate helper class (called `scheme` in the `PagingSystem` simulator). A abstract API/interface has been defined that describes how a page re-

placement scheme class is accessed and used. The abstract API/base class is defined in the files named `PageReplcementScheme.[hpp|cpp]`. This is an example of a Strategy and/or Template method design pattern.

Most of the functions in the `PageReplacementScheme` abstract base class are virtual functions, meaning that concrete subclasses must be created of this base class and implement those virtual functions. For the assignment you have been given a working `FifoPageReplacementScheme.[hpp|cpp]` concrete class that implements the simple FIFO page replacement scheme. A class that can act as a `PageReplacementScheme` has the following interface. The main function of such a class is the `makeReplacementDecision()` function. Whenever memory is full, the paging simulator will call this function to ask the page replacement scheme to select which frame of memory should be kicked out and replaced with the new page reference. All subclasses of the `PageReplacementScheme` base class need to implement an algorithm to be able to select the frame for page replacement when needed.

The `PageReplacementScheme` API has a few other functions. The paging system simulator will call the scheme whenever a page hit occurs, because some page replacement schemes will update information about page usage based on when or how often the page has been hit. Another major API function that the `PageReplacementScheme` subclasses implement is the `getSchemeStatus()` function, which is called to get a snapshot of the current status of the replacement scheme, to get insight into its decision making process.

There is a working FIFO page replacement implementation given already to you as part of the assignment. Your main task, after adding some functionality to the `PagingSystem` simulator class, will be to implement a basic Clock page replacement scheme, which is a modification of the basic FIFO scheme.

Overview and Setup

For this assignment you will be implementing missing member methods of the `PagingSystem.[hpp|cpp]` class which controls the main simulation for this assignment, as well as writing the `ClockPageReplacementScheme.[hpp|cpp]` file to implement a Clock page replacement scheme. 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 04 Page Replacement Schemes and Clock Algorithm' 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 (PagingSystem Simulation)

For this assignment, you will first of all be completing some of the functions in the `PagingSystem` class to get the basic simulator running. Then once the simulator is complete, you will be implementing the functions of the `ClockPageReplacmentScheme.[hpp|cpp]` to create a Clock page replacement algorithm.

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 `PagingSystem` class to work you will first need to complete the following tasks.

Task 1: Implement basic `PagingSystem` accessor methods

The first setup test case of the unit tests this week simply tests some of the accessor methods of the `PagingSystem` class, and the functions to load and generate simulated page streams.

As a warm up exercise the get accessor methods of the `PagingSystem` class have been left unimplemented. You need to implement these functions to get the first test case working: `getMemorySize()`, `getSystemTime()`

`getNumPageReferences()`. All of these accessor methods are simply accessing and returning a corresponding member variable of the `PagingSystem` class. These functions have not yet been declared, so you have to create a function declaration in the `PaginsSystem.hpp` header file, then the implementation in the `PagingSystem.cpp` implementation file. Also remember that all functions are required to have proper Doxygen function documentation, so you need to create the function documentation as well for each of these functions just before each implementaiton.

Each of the 3 accessor methods have a test sub task, so define and implement them one by one. All 3 of these methods should be `const` member methods. This is common for an accessor method, they simply return information about the current state of the object, they do not cause the state of the object to change. Declaring a member method to be a `const` member method guarantees that by calling this method the state of the object will not be changed.

All 3 of these accessor methods take no input parameters, which is also the usual case for an accessor method. And all 3 should return an appropriate result.

As in the past, you must also create appropriate Doxygen function documentation for these and all member methods that you add to the `PagingSystem` in the first parts of this assignment.

Once you have implemented these accessor methods and 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 `isMemoryFull()` member method

The next function you need to implement is the `isMemoryFull()` member method. This method returns a `bool` result. It should return `true` if memory is currently full, that is to say if all frames have a page currently loaded into them (or equivalently, no frame is currently an `EMPTY_FRAME`). This function does not change the state of the simulation, it only returns information about the current state of memory, so it should be a `const` member function. This function does not have any parameters as input.

This method returns `true`, as mentioned, if all frames in the simulated memory have a valid page in them. It returns `false` if one or more frames are `EMPTY_FRAME`. There are two member variables you need in order to implement the work of this function. The member variable `memorySize` that you returned in a getter method in the previous task tells you how many physical frames of memory there currently are. There is also a member variable called `memory` which is an array of `Page` instances (a typedefed int).

Valid page numbers in our simulations are unsigned integers with values of 1 or larger. We use 0 to indicate an empty or missing page. The global defined constant `EMPTY_FRAME` is set to 0, but you are required to use this globally defined constant when testing if frames are empty or not in this function.

The suggested pseudocode to implement this function could look something like this:

```
for each physical frame of memory
   if this frame in memory is EMPTY_FRAME
       return false, the memory is not yet full

otherwise if all frames were not empty, return true, the memory is full
```

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 `isPageHit()` member method

The next function you need to implement is the `isPageHit()` information function. This function also returns a boolean result, and it also does not actually modify the state of the running simulation, so it is required to be a `const` member function.

The current page being referenced in the simulation will be given in `pageReference[systemTime]`, that is to say, given the current `systemTime` the page referenced at that time by the simulated page reference stream is found in the array `pageReference[systemTime]`. Given the current `systemTime`, the `isPageHit()` function should return `true` if the page being referenced is currently in `memory` (which is a page hit). Otherwise it should return `false`.

So a possible pseudocode implemention of this function is as follows:

```
for each physical frame of memory
   if this memory frame holds the current page reference
      return true, this reference was a hit, we found the page in memory

otherwise if no frame currently has the referenced page loaded
then return false, this reference was a miss
```

Make sure that you are creating Doxygen function documentation for all functions when you commit them to your repository as you are writing each one. The funciton documentation should be present as part of the commit for each task.

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 `doPagePlacement()` member method

Before defining the task 4 tests, Start by uncommenting if/else at bottom of `processNextPageReference()` in `PagingSystem.cpp` for handling a page fault/miss. Now that the `isPageHit()` and `isMemoryFull()` are implemented, we can uncomment this code, to do placements or replacements, to test your implementation of `doPagePlacement()` next.

The code will not compile after uncommenting at this point until you also create at least a stub function for the `doPagePlacement()` method. So as usual define the tests for task 4, and create a header and stub function for `doPagePlacement()`, and make sure code is compiling and running again before proceeding.

The `doPagePlacement()` member method does change the state of the simulation, so it is not a `const` member method this time. This method does not have any input parameters, and it is a `void` function that does not return any result. All of the work it performs is done as a side effect, it places a newly referenced page into the simulated memory.

Basic paging systems usually split loading a new page into two separate cases. When there are still free frames of memory available, simple page placements are performed, where a free frame is picked and the page that was referenced is loaded into that free frame. We can commonly use a simple frame pointer, or just pick the first free frame in physical memory to do this placement, because for a paging system initial placement has no performance implications. However, when memory is full, we instead have to make a replacement decision first, where we select a frame of memory with a page in it to kick out of memory, so we can load the newly referenced page.

Page placement happens whenever there are free frames of memory, so that we simply want to pick the next free frame to load the current referenced page into. The `doPageReplacment()` function has already been completed for you, because it actually relies on calling the helper `scheme` instance to do the actual page replacement algorithm.

For the `doPagePlacement()` function, you should first check if memory is full. Page placement should never be called if memory is full, so if memory is actually full you need to throw a `SimulatorException()`. But if memory is not full, you need to do the actual work of a page placement. For a page placement, you should search through memory and find the first frame that is an `EMPTY_FRAME`. Once found, this frame should be replaced with the current page reference (e.g. `pageReference[systemTime]`).

So an example implementation of the `doPagePlacement()` funciton might look like this:

```
if memory is full
   throw exception indicating this is a simulation error and this method should not be
   called when memory is full


for each frame of memory
   if this frame of memory is empty
      place the current page reference in this empty frame
      and return immediatly (e.g. don't make mistake of putting page in multiple empty frames)
```

Once your implementation compiles and passes the unit tests for task 4, commit your work and push it to the Feedback pull request of your GitHub classroom repository.

Task 5: Enable full `PagingSystem` simulation and fix Fifo class

For task 5 you need to add in some code to the `FifoPageReplacementScheme` class, and uncomment some more code in the `PagingSystem` class. All of this is because parts of the simulation need to use functions you implemented in tasks 1-4 previously, but in order to get things to compile, we had to comment out the calls to these functions and/or remove them. But now that the `PagingSystem` class is basically complete, we want to test that basic full simulations will work with the Fifo page replacement scheme that has been (mostly) implemented for you as an example of how the page replacement scheme Strategy pattern works.

Start by uncommenting the code that uses `getMemorySize()`, `isPageHit()` and `doPagePlacement()` in the `PagingSystem.cpp` implementation file. You already enabled some of this in previous task. You should find there is code also commented out in the following functions of the `PagingSystem` that needs to be enabled:

- `getPageStatus()` needs to call both `isPageHit()` and `isMemoryFull()` to determine status for output for full simulations.
- You should already have enabled the use of `isPageHit()`, `isMemoryFull()` and `doPagePlacement()` in the `processNextPageReference()` method in previous task 4.
- `doPageReplacement()` also has commented out a call to `isMemoryFull()`, simply checking for an error condition. Page replacement should NOT be done if there are still empty frames of memory. The check here is similar to what you should have done in task 4 to test if memory is full when being asked to do a page placement.

You might want to enable that code in the `PagingSystem` and then check that your code still compiles and successfully runs and passes the task 1-4 tests.

However, there are also two things that need to be done in the `FifoPageReplacementScheme.cpp` implementation of the Fifo scheme as well, both of which are that the Fifo algorithm needs to know the size of memory (number of physical frames) to do its work. In the `getSchemeStatus()` method of the `FifoPageReplacementScheme` there is a loop that needs to iterate over all of the frames of the simulation. However, currently this loop iterates 0 times as written. Change the for statement of the loop to look like this:

```
for (FrameNumber frame = 0; frame < sys->getMemorySize(); frame++)
```

The size of memory (or number of physical frames) is information that is kept by the `PagingSystem`. However, all `PageReplacementScheme` classes have a pointer back to the `PagingSystem` system that they are working with that is named `sys`. So if we want to figure out the size of memory of our simulation (or any other simulation property), we can query the `sys` object to ask it for that information, as shown here.

Likewise, in the `makeReplacementDecision()` member method for the Fifo scheme, we need to know the size of memory so that we can correctly wrap the frame pointer around if it goes past the end of memory. So replace the line of code that increments the `framePointer` and wraps it around the memory frame buffer with the following:

```
framePointer = (framePointer + 1) % sys->getMemorySize();
```

Again after making these changes in `FifoPageReplacementScheme.cpp` you should check that your project still compiles and runs all of the tests for tasks 1-4 before proceeding.

If you successfully enable that code in the simulation, you should now be able to define the task 5 tests, and they should all run and successfully pass now, if you haven't missed anything. The task 5 tests check that full simulations using the default Fifo page replacement policy are now working and running as expected. You should also find that all of the system tests that use the Fifo page replacment scheme will now pass as well.

Once you have enabled the described functionality and are able to pass full simulation tests, commit your code for task 5 and push it to your GitHub classroom repository.

Assignment Tasks (ClockPageReplacementScheme)

Once these 5 tasks are complete, the first 5 test cases of this assignment should be passing. These test the simulator, load page streams, and test using the basic FIFO page replacement scheme to make page replacement decisions. However the final unit tests and the system tests of the Clock page replacement algorithm will not pass until we fully implement the Clock paging algorithm.

So the next step is to implement a Clock page replacement policy. Most of the functions in the `ClockPageReplacementScheme.[hpp|` have been left for you to implement. However, many of the implementations of these functions will be similar to the

same functions given in the `FifoPageReplacementScheme.[hpp|cpp]` files.

All of the functions already have stubs declared for them in the `ClockPageReplacementScheme.cpp` implementation file, as well as function documentation. So you need only implement the missing functionality. The Clock page replacement scheme is a combination of a Fifo scheme and the Least Recently Used (LRU) scheme, as you should have learned from your materials for this unit. You should look at the implementation of these member methods in the `FifoPageReplacementScheme` and modify them for your implementation of the Clock schem. In fact, a good starting point for all of the following tasks is to copy the method from the Fifo scheme, then modify it for the described Clock scheme.

Perform the following tasks:

Task 6: Add use bits DataStructures to `ClockPageReplacementScheme` and implement `resetScheme()` member method to initialize them

Start by defining the task 6 tests in the unit test file. You can also copy over the implementation of the `reset()` method from the Fifo scheme to your clock implementation to get started.

You will need a `framePointer` for your clock scheme, just like the FIFO page replacement scheme. But you will also need to keep an array of use bits, 1 bit for each physical frame of memory. You are required to use an array of `bool` bits for your use bits array. This array must be called `useBit`, as this will be tested explicitly in the task 6 tests. You will need to dynamically allocate the memory for the array of use bit values, depending on the simulation memory size. So you should declare the member variable to be a `bool*`, a pointer to a bool, so you can dynamically allocate a block of bool values to use as the `useBit`

You have been given the implementation of the constructor for your class. It works the same as the FIFO class, it simply calls the base class constructor, then calls the `resetScheme()` member function.

As your second task you should implement the `resetScheme()` member function. You should initialize the `framePointer` like the FIFO class does. But in addition, you need to dynamically create your array of use bits here and initialize them.

Subclasses of the `PageReplacementScheme` have a member variable named `sys` which is a pointer to an instance of the `PagingSystem` class that the scheme is working with. In task 5 you had to add some code to the Fifo class to find out the memory size of the current simulation. Likewise, ou can query the `sys` object for this needed information here as well. For example you can do `sys->getMemorySize()` to find out what the size of the simulated memory is.

This is necessary because in addition to initializing the framePointer, you should dynamically allocate your array of use bits here to hold `memorySize` bits. And you should initialize all of the use bits to be 1 or true at this point, because after initial page placement, all of the use bits should initially be 1.

The task 6 tests simply test that your implementation of the `resetScheme()` can be called, and it peeks into your private member variables to check that you have correctly initialized them.

Once you are satisfied with your implementation of the `resetScheme()` method and your tests are passing, commit your work and push it to the Feedback pull request of your GitHub classroom repository.

Task 7: Implement `pageHit()` member method

Next implement the `pageHit()` member function. The FIFO class doesn't need to do anything for a page hit (so there is nothing to copy to begin with here). But for the Clock scheme, you have to set the use bit to true (1) for a page hit. When the `pageHit()` function is called, the frame number of the page that was hit is provided as the parameter, so you simply need to set the use bit of that corresponding frame to true to handle a page hit.

You should define the task 7 tests, and implement the `pageHit()` function to correctly set the use bit of the indicated frame when it is called.

Once your implementation is passing the tests for task 7 and you are satisfied with your code, commit your work and push it to the Feedback pull request of your GitHub classroom repository.

Task 8: Implement `makeReplacementDecision()` member method

Finally implement the `makeReplacementDecision()` function. The replacement decision for Clock is more complex than for Fifo, though you can start by copying the Fifo replacement decision function.

For Clock, you have a `framePointer`, just like for the Fifo scheme. But you don't just immediately replace the frame that the `framePointer` is pointing to. You first need to scan memory until you find the next frame that has a use bit set to false (0). So if the frame that the frame pointer points too has a use bit of true (1), you need to flip it to false (0) and move to the next frame. Thus you need to implement a loop here that keeps checking the use bit, and flipping it to false until it finds a use bit that is unset, wrapping around to the beginning of the memory buffer as needed.

Once a frame is found with a use bit of false, that should be the frame that is selected to be replaced. That frame number should be returned from this function. But before you return, you should make sure that the `framePointer` points to the frame after the one that will be replaced. You should also set the use bit of the frame that will be replaced to be 1, because whenever a new page is loaded its use bit should initially be set to 1.

The pseudocode to implement the `makeReplacementDecision()` looks something like the following:

```
while use bit at current frame pointer is set (true)
    flip the use bit to false
    increment the frame pointer by 1, wrapping around the end of the buffer back to 0 if needed

// frame pointer now points to first frame found with use bit of false
remember this frame pointer as it should be returned as the candidate frame to replace

but before returning set the use bit of the frame to replace to true
and increment the frame pointer by 1, wrapping around the end of the buffer if needed

return the rememered frame pointer for replacement
```

Once you are satisfied with your implementation and are passing the task 8 tests, make a commit and push it to your Feedback pull request in your GitHub classroom repository.

Task 9: Implement `getSchemeStatus()` member method

If you get the 8 steps working correctly, you should now be able to pass all of the unit tests. There are no task 9 unit tests defined. However, the full system tests will not yet pass until you implement the final missing member method of the Clock page replacement scheme.

This function is pretty similar to the implementation of the FIFO class, so you can start by copying the code from the FIFO class for this function to your clock class. The only difference is that the clock `getSchemeStatus()` function should display the use bits for each frame of its output. So you will need to add code to show the use bit associated with each frame to the output string returned. Take a look at the result files of running the simulations for the Clock scheme. You need to replicate the output of the use bits, to display if the `useBit` is set to 0 (false) or (true) for each frame in the simulation.

Once you get this output correct, your system tests should then pass successfully as well when you run them.

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 scheme memorySize pageref.sim
  Run page replacement simulation on the given page reference
  stream file.  Output shows the state of memory (loaded pages
  in each memory frame) after each page reference as well as
  summary information about hit/miss performance.  You can
```

```
  select from different page replacement schemes by specifying
  the scheme parameter.
Options:
  scheme       The page replacement scheme to use, current 'fifo'
               and 'clock' are supported
  memorySize   The number of physical frames of memory to simulate
  pageref.sim  Filename with page references, one per line,
               that represent references to pages of a running
               (simulated) process or set of processes
```

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 fifo 3 simfiles/pageref-01.sim
Paging Simulation
====================================
memory size          : 3
reference stream size: 12
paging scheme        : fifo

------------------------------------
system time   : 0
page reference: 2
page status   : fault (placement)

frame[000]     2 <-- framePointer
frame[001] EMPTY
frame[002] EMPTY

Hit  ratio : 0 / 1 (ratio = 0)
Fault ratio: 1 / 1 (ratio = 1)

Hit count        : 0
Placement count  : 1
Replacement count: 0


... <snip output> ...
```

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
System test pageref-01 fifo memorySize 03: PASSED
System test pageref-01 fifo memorySize 04: PASSED
System test pageref-02 fifo memorySize 03: PASSED
System test pageref-02 fifo memorySize 05: PASSED
System test pageref-01 clock memorySize 03: PASSED
System test pageref-01 clock memorySize 04: PASSED
System test pageref-02 clock memorySize 03: PASSED
System test pageref-02 clock memorySize 05: PASSED
System test pageref-03 fifo memorySize 04: PASSED
System test pageref-03 clock memorySize 04: PASSED
================================================================================
All system tests passed     (10 tests passed of 10 system tests)
```

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 80 points can be received then for completing tasks 2-5, which complete the `PagingSystem` simulation member methods.
5. 15 more points, for a total of 95 points, will be awarded for completing the `ClockPageReplacementScheme` member methods that implement the full clock page replacement policy, and for having all of the unit tests passing for this assignment.
6. A final 5 points, for a total score of 100, will be awarded for completing the `getSchemeStatus()` member function of the Clock scheme, and successfully passing all of the system tests for this assignment.

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.