Assignment 4: Page Replacement Schemes, Clock Algorithm

CSci 430: Introduction to Operating Systems

Fall 2020

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 had in our written assignments and in class, and simulate a physical memory of some number of frames, 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.
- Look at 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 replacement scheme class is accessed and used. The abstract API/base class is defined in the files named `PageReplcementScheme.[hpp|cpp]`. Most of the functions in this 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. 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.

Unit Test Tasks

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 simulator class completed you will first need to complete the following tasks.

1. The first test case of the unit tests this week simple tests the accessor methods of the `PagingSystem` class, and the functions to load and generate simulated page streams. As a warmup exercise the get accessor methods of the `PagingSystem` class have been left unimplemented. You need to complete these functions to get the first test case working: `getMemorySize()`, `getSystemTime()` `getNumPageReferences()`.
2. The next function you need to implement, still used in the first test case, is the `isMemoryFull()` function. This function should return `false` if any of the frames of `memory` are an `EMPTY_FRAME`, and it will return `true` if all of the frames are non empty.
3. The next function you need to implement is the `isPageHit()` information function. This function also returns a boolean result. The current page being referenced in the simulation will be `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`.
4. The final task you need to do to get the simulator working is to finish the `doPagePlacement()` function. 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]`).

Once these 4 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 test Case 6 will not be working as it tests the Clock page replacement scheme class.

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.

Perform the following tasks:

1. 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. I recommend you use an array of `int` or an array of `bool` types for your use bits.
2. 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()` class. You should initialize the `framePointer` like the fifo class does. But in addition, you need to dynamically create your array of use bits here. 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. You can query the `sys` object for needed information. For example you can do `sys->getMemorySize()` to find out what the size of the simulated memory is. This may be useful because in addition to initialize 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.
3. Next implement the `pageHit()` member function. The fifo class doesn't need to do anything for a page hit, but for the Clock scheme, you should set the use bit to 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 1 to handle a page hit.
4. Implement the `makeReplacementDecision()` function next. The replacement decision for clock is more complex than for fifo. You have a `framePointer`, but you first need to scan memory until you find the next frame that has a use bit set to 0. So if the frame that the frame pointer has a use bit of 1, you need to flip it to 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 0 until it finds a use bit of 0. Once a frame is found with a use bit of 0 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.
5. If you get these 4 steps working correctly, you should now be able to pass all of the unit tests. However, the system tests will not pass yet until you implement the `getSchemeStatus()` function. 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 get scheme status 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. Once you get this output correct, your system tests should then pass successfully as well.

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 `assg04-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: As with the previous assignment, the assg04-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 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
```

The simulator requires 3 command line arguments to run. You first specify the page replacement scheme to use. Only `fifo` and `clock` will be supported at this point in implementing the assignment. The next parameter specifies the size of the memory that will be used in the simulation (in terms of the number of frames of physical memory available). The last parameter is the name of a file that contains a page reference stream that should be used for the simulation.

As mentioned, if you correctly modify the `getSchemeStatus()` function to display the use bits for each frame in addition to the other information, then you should be able to pass the given system tests for this assignment as well. Just add the use bit information you are keeping track for each frame of the system in the output.

Extra Credit Opportunity

In this assignment you have been given a FIFO implementation, and if you were successful, you implemented a Clock page replacement scheme. The other two page replacement schemes we study in this course are the least recently used (LRU) and optimal (OPT) schemes. If you are interested in this paging system simulation, and/or would like an opportunity to make up some points for the class for previous assignments or tests, I will give up to 10 points each for a LRU and/or an OPT page replacement scheme implementation. You should start by simply copying the `FifoPageReplacementScheme.[hpp|cpp]` files and renaming the class from that to for example `LruPageReplacementScheme` and `OptPageReplacementScheme`. Both of these schemes need to look at the history (or future references) of the page reference stream. So if you need to you can add an accessor method to return the `pageReference` array from the `PagingSystem` simulator so that these classes can access the page reference stream. For LRU, another approach is to use time stamps. You can just use the system time, and any time a page is loaded or is referenced (a hit) you update the time stamp. Then for LRU you would want to search your time stamps to find the oldest time reference, which will be the least recently used page/frame, and select it for replacement.

To get the full 10 points, make sure you also update PagingSystem so that the new scheme can be specified as a potential page replacement scheme. Also you need to add unit tests for your scheme, and most importantly, you should add at least 1 or 2 system tests giving correct output/working of your replacement scheme. For example the `pageref-01.sim` page reference stream is the same as the one used for the examples in our book. So if you implement OPT or LRU you should get the same final result shown in the text if you use a simulation with 3 physical frames of memory.

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 assg04.tar.gz PagingSystem.hpp PagingSystem.cpp
    ClockPageReplacementScheme.hpp ClockPageReplacementScheme.cpp
    PagingSystem.hpp PagingSystem.cpp ClockPageReplacementScheme.hpp
    ClockPageReplacementScheme.cpp
```

The result of this target is a tared and gziped (compressed) archive, named `assg04.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. 20 pts all accessor methods implemented and first test case passes.
3. 10 pts `isMemoryFull()` function works and passes test cases.
4. 10 pts `isPageHit()` function is working and passing test cases.
5. 10 pts `doPagePlacement()` function implemented as asked for. Function correctly selects first empty page and replaces it when asked.
6. 10 pts Have a good representation of the use bit defined for the clock page replacement class.
7. 10 pts `resetScheme()` is implemented and correctly creates and initialize needed data structures for the clock algorithm.
8. 10 pts `pageHit()` function implemented and correctly updates use bit for frame when a page hit occurs.
9. 15 pts `makeReplacementDecision()` implemented and working as asked for. Correctly uses use bit and clock algorithm to select page and return it for replacement.
10. 5 pts `getSchemeStatus()` added and working correctly, all system tests passing for the assignment.
11. 10 bonus pts for a working `LruPageReplacementScheme` class.
12. 10 bonus pts for a working `OptPageReplacementScheme` class.

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.