EEE335 – Lab 5

Page Replacement Algorithms

Objective

The goal of this lab is to put into practice what we saw in class on paging and page replacement algorithms.

Lab Instructions

Read through the entire lab handout before "hands on keyboard".

Introduction

This lab will have you implement several page replacement algorithms seen in class:

a) First In – First Out (FIFO);
b) Second Chance;
c) Clock;
d) Not Recently Used; and
e) Working Set.

The algorithms will be used inside a C ++ program that simulates paging in memory. For this lab, you will use the same Linux virtual machine (VM) as in the previous lab. Make sure your `C:\Users\Public\Documents\share` directory is still present on your computer before starting VM, otherwise it will freeze during start-up.

Preparation of the Lab Environment

The starter code contains the simulation program and an implementation of a random page replacement algorithm. Download the `.zip` file available on the course site and extract the lab5 folder to your VM desktop. To import the project, start Eclipse and select `File->Import->General->Existing Projects into Workspace`, then navigate to the lab5 folder and make sure to select the `Copy projects into workspace` option before clicking `Finish`. To complete the lab, you will modify the classes in the `pageReplacementAlgortihms` folder, but you should not have to modify the other classes or files. If you think it is necessary to modify these other classes or files, ask the instructor before doing so. When marked, it will be assumed that no changes have been made to the simulation program.

Description of the GUI

Once the project is installed, you can compile and run it. The GUI of the program as shown in Figure 1 below should be displayed. The `Start` button starts the simulation. Once the simulation starts, this button changes and now displays `Pause` which enables the simulation to be paused. When the simulation is paused, you can use the `Step` button to perform a single iteration at a time. The `Reset` button allows you to reset everything without having to restart the application. On the right side, the

Page Replacement Algorithm (`PRA`) menu allows you to choose the page replacement algorithm. In the base code, only the `Random` algorithm is implemented. If you use the other algorithms before you have programmed them, the program will fail after a few iterations. The `Speed` menu item speeds up and slows down the simulation speed. It simply changes the waiting time between each iteration. The `Stop at` menu item allows you to automatically pause the simulation when a certain number of iterations have been reached. This is useful for comparing the performance of different algorithms. The `Sim turn` field displays the number of iterations executed so far. The `Page fault` field displays the number of page faults encountered so far.

The two tables below show the state of the virtual memory and the state of the physical memory. An `X` in a virtual memory box means that this page has not been used yet or that its contents are paginated on the disk. A number in a box of the virtual memory means that the content of this page is in physical memory in the frame indicated by the number. On the physical memory side, an X means that the frame is empty while a number identifies the number of the virtual page loaded in that frame. At each iteration, the changes in these two tables are identified in yellow for better viewing. In Figure 1, we notice the virtual page 0 is not used or is paginated on the hard disk. The virtual page 1 is loaded into the physical memory in the frame 2. If we look at the frame 2 of the physical memory, we see that it contains the virtual page 1. The boxes in yellow indicate that the last operation was to load the virtual page 10 in the physical frame 4

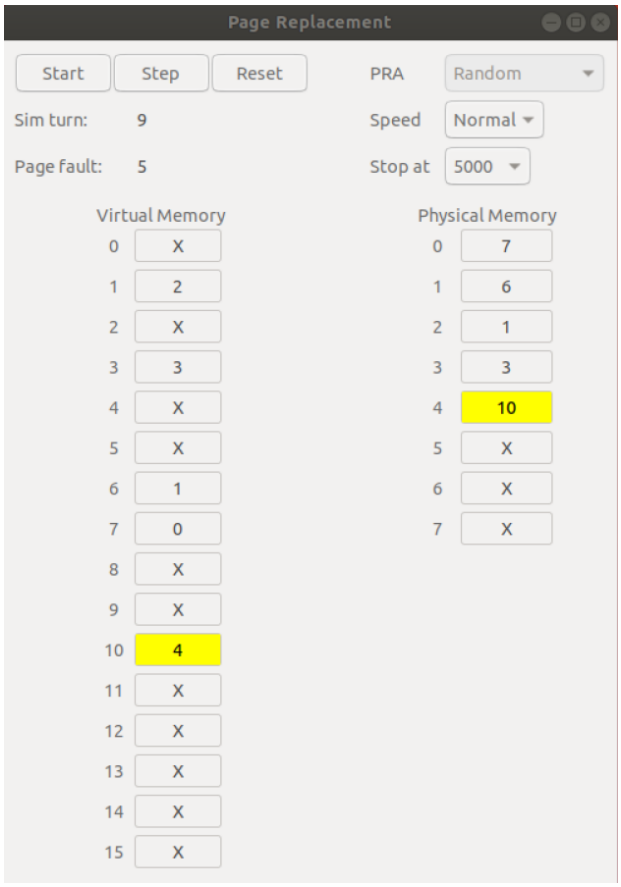

Figure 1 – Screen Capture of the Paging Simulation Program

Description of Operation

The program simulates the execution of a process on a system with virtual memory and paging. When the simulation runs, the `doTurn()` method of the `Simulation class` is executed. This method chooses a page of virtual memory to access. If the page is already loaded in physical memory, everything is fine. Otherwise, a page fault occurs. The `doTurn()` function then traverses the physical memory to find an empty frame where the virtual page could be loaded. If the physical memory is full and there is no empty frame, then the `findVirtualPageToEvict()` method of the abstract `PageReplacementAlgorithm` is called. Depending on the option chosen on the GUI, this abstract method will be implemented by a concrete method of a subclass of `PageReplacementAlgorithm`. Recall the concept of polymorphism studied in your object-oriented programming course. The virtual page returned will then be evicted from the physical memory leaving an empty frame for the virtual page. The virtual page is then loaded into the physical memory and accessed by the process. Access is a read or write.

Details of the PageReplacementAlgorithm class

The code for the abstract class `PageReplacementAlgoirthm` is provided below:

```
#pragma once

#include "PageTableEntry.h"

class PageReplacementAlgorithm
{

public:
    PageReplacementAlgorithm() { }
    virtual ~PageReplacementAlgorithm() { }

    // Returns the number of the virtual page to evict
    virtual int findVirtualPageToEvict(
                        TableEntry *pageTable) = 0;

    // Notifies the PRA that a virtual page was evicted from memory
    virtual void virtualPageEvicted(
                        TableEntry *pageTable,
                        int virtualPage) = 0;

    // Notifies the PRA that a virtual page was loaded in memory
    virtual void virtualPageLoaded(
                        TableEntry *pageTable,
                        int virtualPage) = 0;

    // Notifies the PRA that a virtual page was accessed
    virtual void virtualPageAccessed(
                        TableEntry *pageTable,
                        int virtualPage,
                        bool modified) = 0;
};
```

The `PageReplacementAlgorithm()` method is the constructor of the class. In this example, this method is implemented directly in the `.h` file, but contains no code as seen by the empty `{}` brackets.

The `~PageReplacementAlgorithm()` method is the destructor of the class. It is in this method that any memory allocated by the constructor or during execution, is released. In this example, the destructor is also empty. In C ++, the virtual keyword that is in front of the declaration of a method means that the redefined method of the child class must be executed if it exists. In this case, when an object of type `PageReplacementAlgorithm` is destroyed, the destructor of the subclass will be executed.

The abstract method `findVirtualPageToEvict()` has as a parameter a pointer to the virtual page table and must return the number of the virtual page to be evicted. Each page replacement algorithm implements this method differently, depending on the strategy used. We know that this method is abstract because of `= 0;` at the end of this statement.

The abstract method `virtualPageEvicted()` has as parameters a pointer to the table of virtual pages as well as the number of the page that has just been evicted from the physical memory. This method is called whenever a page is pushed out of physical memory. This method, as well as the following, will be useful if you need to implement your own representation of physical memory.

The abstract method `virtualPageLoaded()` has as parameters a pointer to the table of virtual pages as well as the number of the page which has just been loaded into physical memory. This method is called whenever a page is loaded into physical memory.

Finally, the abstract `virtualPageAccessed()` method has as parameters a pointer to the virtual page table, the number of the virtual page accessed as well as a boolean identifying whether the page has been modified or not.

When the simulation runs and a page fault occurs, the method calls are in the following order: `findVirtualPageToEvict() -> virtualPageEvicted() -> virtualPageLoaded() -> virtualPageAccessed()`. When there is no page fault, only the `virtualPageAccessed()` method is called. Note that a point to the virtual page table is passed to each of these methods, this allows you to access the contents of the page table and modify it if necessary. Inspect the code of the `Simulation::doTurn()` method to see what is already updated by the simulation and what needs to be updated by your page replacement algorithms.

Tasks

Task 1: Familiarization with simulation

Start the simulation and observe the operation of the random algorithm provided. Observe paging statistics and the content of virtual and physical memory. Stop the program when you understand the information shown to you.

In the program code, specifically in the config.h file, change the number of page frames in physical memory to equal the number of virtual pages. Observe the effect of the change. Be sure to return the constant to its original value for the rest of the lab.

Task 2: Implementing paging algorithms

It is now your turn to implement some page replacement algorithms. You must implement:

1) First come, first served (Tanenbaum - 3.4.3);
2) Second chance (3.4.4);
3) Clock (3.4.5);
4) No Recently used (3.4.2); and
5) Work package (3.4.8).

The C++ classes for each of the algorithms have already been created for you, you just have to modify them to implement the correct behaviors. A secondary goal of this lab is to introduce yourself to the C++ programming language, but not to make you experts. If you have questions about the syntax or organization of `.h` and `.cpp` files, feel free to ask your instructor.

Questions

1) For each of the implemented algorithms, run the simulation for 10000 iterations and note the number of page faults. Repeat the test three times for each algorithm and calculate the average.
2) If you could apply any page replacement algorithm to this system, which one would you choose? Why?
3) What is the effect of increasing the number of physical page frames? What happens if it equals the number of virtual pages? Why?
4) How did you define the working set?
5) What is the effect when you vary the size of the working set?

Laboratory report

Your lab report should include a cover page, a brief introduction, discussion and conclusion. Your discussion should explain how you implemented each algorithm and discuss their respective performance. Your discussion should also explain what you have learned, the discoveries you have made, which has been difficult in the laboratory. It should not be a list of followed steps or empty phrases about the importance of this lab. Be concise, a good discussion paragraph is better than 10 pages of fill.

The report should also include answers to the questions above. Be sure to include both the questions and answers.

Submission

You must submit your report in pdf format and source code files for each of your classes. Submit these files as a single compressed file (e.g. .zip or .7zip) with a meaningful name (e.g. `lab5_Your_two_last_names.zip`) and submitted via the course Moodle site.