COMP SCI 3004/7064 Operating Systems
Practical 2 – Virtual Memory Simulation

Aim

By doing this practical work, you will learn how to implement page replacement algorithms, gain experience in creating and evaluating a simple simulator, and develop your skills in scientific writing.

You should work in **groups of size 2 or 3.** Each group will submit one simulator and one report.

Deadlines: Code is due **Tuesday 5th September 2023.**
Report due **end of week 8 - Friday 15th September.**

Introduction

In chapter 22, we explore a variety of page replacement algorithms for managing virtual memory. The choice of a page replacement algorithm is actually quite a complex matter. To make the proper choice, we must know something about real applications. How do they access memory? Do they generate many page accesses in order? Do they skip around memory randomly? The only way to answer these questions is to see what real applications do.

In this practical, you will evaluate how real applications respond to a variety of page replacement algorithms. Of course, modifying a real operating system to use different page replacement algorithms is quite difficult, so we will simulate it instead. You will write a program that emulates the behaviour of a memory system using a variety of page replacement algorithms.

Then, you will use memory traces from real applications to evaluate your algorithms properly. A main outcome of your work will be a report. The report itself counts for 60% of this assignment.

Memory Traces

We provide you with **four** memory traces to use with your simulator. Each trace is a real recording of a running program, taken from the SPEC benchmarks. Real traces are enormously big: billions and billions of memory accesses. However, a relatively small trace will be more than enough to capture their memory access patterns. Each trace consists of *only* one million memory accesses taken from the beginning of each program.

Each trace is a series of lines, each listing a hexadecimal memory address followed by R or W to indicate a read or a write. For example, gcc.trace trace starts like this:

```
0041f7a0 R
13f5e2c0 R
05e78900 R
004758a0 R
31348900 W
```

Each trace is compressed with gzip, so you will have to download each trace and then uncompress it with a command like this:

```
> gunzip –d gcc.trace.gz
```

Simulator Requirements

Your job is to build a simulator that reads a memory trace and simulates the action of a virtual memory system with a single level page table. The current simulator fixes the pages and page frames size to 4 KB (4096 bytes). Your program should keep track of what pages are loaded into memory. The simulator accepts 4 arguments as follows:

- the name of the memory trace file to use.
- the number of page frames in the simulated memory.
- the page replacement algorithm to use: rand/lru/esc
- the mode to run: quiet/debug

If the mode is "debug", the simulator prints out messages displaying the details of each event in the trace. The output from "debug" it is simply there to help you develop and test your code. If the mode is "quiet", then the simulator should run silently with no output until the very end, at which point it prints out a summary of disk accesses and the page fault rate.

As it processes each memory event from the trace, the simulator checks to see if the corresponding page is loaded. If not, it should choose a page to remove from memory. Of course, if the page to be replaced is dirty, it must be saved to disk. Finally, the new page is to be loaded into memory from disk, and the page table is updated. As this is *just a simulation* of the page table, we do not actually need to read and write data from disk. When a simulated disk read or disk write must occur, we simply increment a counter to keep track of disk reads and writes, respectively.

Most of the input (reading a trace), simulation counters and output messages has already being implemented in the skeleton files provided for you.

The skeleton reads the parameters, processes the trace files and for each access it generates a page read or write request. Your job is to complete the simulation of the memory management unit for each replacement policy:

- **rand** replaces a page chosen completely at random,
- **lru** always replaces the least recently used page
- **clock** performs the replacement algorithm described in the textbook section 22.8.

You should start thinking how you can keep track of what pages are loaded, how to find if the page is resident or not, and how to allocate frames to pages. Some short traces (trace1, trace2 and trace3) will be used in the testing script and are provided to facilitate local testing of your code.

Report

An important component of this practical is a report describing and evaluating the replacement algorithms. Your goal is run the simulator to learn as much as you can about the four memory traces (swim, bzip, gcc and sixpack). For example,

How much memory does each traced program actually need?
Which page replacement algorithm works best when having a low number of frames?
Does one algorithm work best in all situations?

Think carefully about how to run your simulator. Do not choose random input values. Instead, explore the space of memory sizes intelligently to learn as much as you can about the nature of each memory trace.

Your group report should have the following sections:

- Introduction: A brief section that describes using your own words the essential problem of page replacement you are trying to investigate. Do not copy and paste text from this project description.
- Methods: A description of the set of experiments that you performed. As it is impossible to run your simulator with all possible inputs, so you must think carefully about what measurements you need. Make sure to run your simulator with an excess of memory, a shortage of memory, and memory sizes close to what each process actually needs.
- Results: A description of the results obtained by running your experiments. Present the results using graphs that show the performance of each algorithm on each memory trace over a range of available memory sizes (alike figures 22.6 to 22.9 in the textbook). For each graph, explain the results and point out any interesting or unusual data points.
- Conclusions: Summarize what you have learned from the results.

The group report must be concise, well structured and free of typos and errors. For reference, a typical report length should be around 4 to 6 pages, roughly one page for the introduction and methods, half to one page per trace (graph and analysis of its results) and half to one page for conclusions.