Assignment 2: Measuring the Performance of Page Replacement Algorithms on Real Traces

Due date: Sunday, Oct 9, 2022 at 11:59pm on Canvas.

Learning Objective:

- Understand the costs of page replacement algorithms
- Observe the effect of memory traces and the effect of page replacement algorithms on system performance.

In this project you are to evaluate how real applications respond to a variety of page replacement algorithms. For this, you are to write a memory simulator and evaluate memory performance using provided traces from real applications.

IMPORTANT: For ABET Assessment purposes, you MUST work in teams of 3, or you can NOT work alone, only one submission per team is needed. Your teams have been already decided by your name orders, please check this link. For each submission, you MUST specify whether your major is Computer Science (CS) or Computer Engineering (CE). For each group, you need to vote for one leader, and the other two are members. Add submit the ABET Assessment form according to your major using one of these two form templates CS Template/CE Template

The deliverables of this project are:

1. A report (in PDF) that includes:
   - Your name(s).
   - A short presentation of the problem you address (hopefully not cut and pasted from this document or the paper, but written in your own words).
   - A report of the results and detailed interpretation of the results.
   - An estimation of the time (each of) you spent working on the project.

   There is no requirement on the length of your report: you'll be graded based on how well your report shows your understanding of the problem.

2. A zip file with all of your files in one folder with your name(s). The zip file needs to include the following:
   - A `makefile` for easy compilation, the C program(s). The `makefile` target executable should be called `memsim`.
   - A `README` file that describes how to run your project (e.g., arguments, etc).
   - Code that implements the objectives of this project. Please do not include your name in the code.

Memory Traces

You are provided with 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 keep you busy. Each trace only consists of one million memory accesses taken from the beginning of each program. The traces are the following (linked from the Canvas project): 1. bzip.trace 2. sixpack.trace
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:

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

Note, to scan in one memory access in this format, you can use fscanf() as in the following:

```
 unsigned addr; char
      rw;
      ...
      fscanf(file,"%x %c",&addr,&rw);
```

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. Your simulator should keep track of what pages are loaded into memory. As it processes each memory event from the trace, it should check to see if the corresponding page is loaded. If not, it should choose a page to remove from memory. If the page to be replaced is “dirty” (that is, previous accesses to it included a Write access), it must be saved to disk. Finally, the new page is to be loaded into memory from disk, and the page table is updated. Assume that all pages and page frames are 4 KB (4096 bytes).

Of course, this is just a simulation of the page table, so you do not actually need to read and write data from disk. Just keep track of what pages are loaded. When a simulated disk read or write must occur, simply increment a counter to keep track of disk reads and writes, respectively.

Implement the following page replacement algorithms to replicate the experimental evaluation in this 1981 paper1[1]:

1. FIFO
2. LRU
3. Segmented FIFO (see 1981 paper on Canvas)

[1] *Rollins Turner and Henry Levy. 1981. Segmented FIFO page replacement. In Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems (SIGMETRICS '81). Association for Computing Machinery, New York, NY, USA, 48–51. DOI:https://doi.org/10.1145/800189.805473*

Structure and write your simulator in any reasonable manner. You may need additional data structures to keep track of which pages need to be replaced depending on the algorithm implementation. Think carefully about which data structures you are going to use and make a reasonable decision.

You need to follow strict requirements on the interface to your simulator so that we will be able to test and grade your work in an efficient manner. The simulator (called `memsim`) should take the following arguments:

```
memsim <tracefile> <nframes> <lru|fifo|vms> <debug|quiet>
```

The first argument gives the name of the memory trace file to use (thus, <tracefile> can be any file in the format described above and in the trace files provided. The second argument gives the number of page frames in the simulated memory. The third argument gives the page replacement algorithm to use. The fourth argument may be "debug" or "quiet" (explained below.)

If the fourth argument is "quiet", then the simulator should run silently with no output until the very end, at which point it should print out a few simple statistics like this (follow the format as closely as possible):

```
total memory frames: 12 events
in trace: 1002050 total disk
reads: 1751 total disk writes:
932
```

If the fourth argument is "debug", then the simulator should print out messages displaying the details of each event in the trace. You may use any format for this output, it is simply there to help you debug and test your code.

If the third argument is "vms", you have to take an extra parameter that indicates the percentage of the total program memory to be used in the secondary buffer. In that case, the simulator should take the following arguments:

```
memsim <tracefile> <nframes> <lru|fifo|vms> <p> <debug|quiet>
```

“p” will be a number between 1 to 100.

Use separate functions for each page replacement algorithm, i.e., your program must declare the following high level functions: `fifo()`, `lru()`, `segmented-fifo()`.

Project Report

An important component of this project will be to write a report describing and evaluating your work and presenting your results using both tables and plots (use Excel or Matlab or another tool of your choice). You should think of this project as if it were a lab experiment in a physics or chemistry class. Your goal is to use the scientific method to learn as much as you can about the provided memory traces. For example, how much memory does each traced program actually need? What is the working set of each program?

Which page replacement algorithm works best? Does one algorithm work best in all situations? If not, what situations favor what algorithms? Use the paper that described Segmented FIFO as guidance for how to present your experimental data. Do your observations confirm the results of the paper?

Your report should have the following sections:

1. Introduction: A section that explains the essential problem of page replacement and briefly summarizes the structure and implementation of your simulator. Do not copy and paste text from this project description. In your own words, describe the overall structure and purpose of the experiment.

2. Methods: A description of the experiments that you performed in order to learn something about each memory trace. Of course, it is impossible to run your simulator with all possible inputs, so you must think carefully about what measurements you need to answer the questions above. 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. For instance, you may want to think strategically of what values you use for the `<nframes>` parameter. On one hand, you want to cover enough of the memory space to see when (a) your "process" does not have enough physical memory (thus, a lot of misses!); (b) when your process' working set fits in the memory; and (c) where increasing the allocated memory does not improve performance significantly. On the other hand, the more values for `<nframes>` you test with, the more hours you'll spend starring at the computer. To help you decide, think of number of frames as power of 2. So perhaps a sequence of 1, 2, 2^2, 2^3, 2^4, ..., 2^{10} might be better than 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, for example. Also, think of what `<nframes>`x`<frame size>` means for today's processes – perhaps an allocation of 2 TB of memory for a process is not particularly common these days (and neither is a physical memory of 32KB).

3. Results: A description of the results obtained by running your experiments. Present the results using tables and plots that show the performance of each algorithm over a range of available memory. For instance, include a plot of hit rate vs. memory size for each algorithm. In the text, summarize what each plot or table shows and point out any interesting or unusual data points. Compare the results obtained from the two trace files – are there any differences in the performance you see? What might explain these differences? Which is the algorithm that shows the biggest difference? What do you think happens?

4. Conclusions: Describe what you have learned from the experimental results. What have you learned about the memory traces? What have you learned about the paging algorithms? How does the size of available memory affect memory performance? Be sure to describe clearly how specific results above lead you to these conclusions.

The majority of your report grade will be drawn from the methods, results, and conclusions. Think carefully about how to run your simulator. Do not choose random input values. Instead, explore the space of memory sizes intelligently in order to learn as much as you can about the nature of each memory trace.

As with any written matter, the report should be well structured, clearly written, and free of typos and grammatical errors. A portion of your grade will cover these matters.