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

Due date: Monday, **March 4th at 11:59pm** on Canvas.

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.

You may work in teams of 2 (feel free to change the teams if you want to see how it is like to work with different colleagues). The output of this project includes:

- A report (in PDF) that includes:
  - Your name(s).
  - A short presentation of the problem you address (hopefully not cut and pasted from this project, but written in your own words).
  - A report of the results and detailed interpretation of the results.
  - An estimation of the time you spent working on the project.
  - You can find a more detailed description of the report at the end of this document.

  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. You may do that in 3 pages or 10 pages.

- Code that implements the objectives of this project, submitted as individual files that **include a `makefile`** for easy compilation, the **C program(s),** and **a `README` file** that describes how to run your project (e.g., arguments, etc). Please do not include your name in the code.
- The `makefile` target executable should be called **memsim**.
- Please zip all of your files **in one folder** with your name(s).
  - .zip, .7z ,and .tar are accepted
- Remember if working in teams of 2, **only one submission** per team is sufficient.

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:

- gcc.trace
- swim.trace
- bzip.trace
- 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:

- LRU.
- FIFO.
- VMS' second chance page replacement policy. You can find its description in the textbook (Chapter 23) and a more detailed description at the end of this document.

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. 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.

Use separate functions for each algorithm, i.e.: your program **must declare the following high level functions: lru(), fifo() and vms().**

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 graphs (use either 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?

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 both tables and graphs that show the performance of each algorithm over a range of available memory. For instance, include a graph of hit rate vs. cache size for each algorithm. In the text, summarize what each graph or table shows and point out any interesting or unusual data points. Feel free to focus your report on one trace only. For that trace, give results and discuss in detail the performance of the 4 algorithms. Doing an outstanding job on simulating the addressing of memory on one trace only will give you full credit. If you have more time, identifying and comparing the sizes of the working sets of multiple traces might give you another point to discuss in the report.

4. **Conclusions**: Describe what you have learned from the 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. You might want to polish your academic writing style by following the writing rules presented here: http://www.wisc.edu/writing/Handbook/index.html

VMS

The VMS' second chance algorithm gets a very short treatment in the textbook, which may create confusion. Below is a bit more detailed explanation.

The VMS algorithm is best understood when two or more processes are competing for memory. For this reason and for simplicity, test your VMS implementation using only the `gcc.trace` input file with the following amendment: consider the addresses that start with 3 as being generated by one process, while all the other addresses are generated by a second process. This way you can simulate (and we can grade a deterministic behavior of) memory references from two processes.

This is how the textbook describes the algorithm. The emphases are ours and mark important points for the implementation:

Segmented FIFO (from Three Easy Pieces, Arpaci-Dusseau & Arpaci-Dusseau)

To address these two problems, the developers came up with the segmented FIFO replacement policy [RL81]. The idea is simple: each process has a maximum number of pages it can keep in memory, known as its **resident set size** (**RSS**). Each of these pages is kept on **a FIFO list**; when a process exceeds its RSS, the "first-in" page is evicted. FIFO clearly does not need any support from the hardware, and is thus easy to implement.

Of course, pure FIFO does not perform particularly well, as we saw earlier. To improve FIFO's performance, VMS introduced two **second-chance lists where pages are placed before getting evicted from memory**, specifically a **global** *clean-page free list* and *dirty-page list*. When a process P exceeds its RSS, a page is removed from its per-process FIFO; if clean (not modified), it is placed on the end of the clean-page list; if dirty (modified), it is placed on the end of the dirty-page list.

If another process Q needs a free page, it takes the first free page off of the global clean list. However, if the original process P faults on that page *before* it is reclaimed, P **reclaims it from the free (or dirty) list**, thus avoiding a costly disk access. The bigger these global second-chance lists are, the closer the segmented FIFO algorithm performs to LRU [RL81].

Some hints and the pseudo-code interpretation of the algorithm described above are below:

- Parameters specific to VMS:
    - RSS. You can consider it the same for the two processes and equal to nframes/2
    - Number of processes: 2
- How to recognize which process requests a page: use gcc.trace. Consider addresses starting with 3 as being part of a process, all the others belonging to the other.
- For each process, a list of pages in memory is maintained in a FIFO order. The FIFO queues are of size RSS.
- In addition, the Clean and Dirty lists are maintained and are global. Assume Clean/Dirty are sufficiently large (The maximum space they need, we believe, is `nframes/2+1` elements each). The Clean/Dirty lists work in FIFO order, although there are cases where elements are removed in a non-FIFO order. They contain the candidates for eviction, but if a page is in Dirty or Clean, it should still be in memory (possibly not for much longer).

```
new_page=get_page(address); //get VPN out of address
current_process = get_process(address); //figure out which process issued req
if new_page in FIFO(current_process)
      do nothing; //this means it is also in memory
else:
      // maintain FIFO and Clean/Dirty buffers
      page_out = insert new page into the FIFO of the current process;
      if (page_out) != NULL
             if page_out is clean:
                  insert page_out into Clean;
            else:
                  insert page_out into Dirty;

      if new_page already in memory:
            //it must be in Clean or Dirty, since it's in memory but it was
            //not in FIFO when requested
            remove new_page from Clean or Dirty, wherever it may be; //it may
            //not be the first

      else:
            if there is room in memory, place it into any free frame;
            else:
                  frame_to_empty  =  first_out(Clean,  Dirty);  //remove  first
element from Clean, if any, or else from Dirty and return the frame number
                  place(new_page, frame_to_empty);
```

Example:

nframes = 8
RSS = 4 for each process
2 processes: A generates requests for pages 1-99; B generates requests for pages 100-199

Reference string:

1. R 1
2. W 1
3. R 2
4. R 3
5. R 4
6. R 5
7. W 6
8. R 100
9. R 101
10. W 102
11. R 1
12. W 5
13. R 103
14. R 104
15. W 105

Start:

FIFO(A):	[]
FIFO(B):	[]
Clean:	[]
Dirty:	[]
Memory:	[]

After processing request 8:

FIFO(A):	[3 4 5 6]
FIFO(B):	[100]
Clean:	[2]
Dirty:	[1]
Memory:	[1 2 3 4 5 6 100 empty] (order does not matter, of course)

After processing request 9:

FIFO(A):	[3 4 5 6]
FIFO(B):	[100 101]
Clean:	[2]
Dirty:	[1]
Memory:	[1 2 3 4 5 6 100 101] (order does not matter, of course)

After processing request 10:

FIFO(A):	[3 4 5 6]
FIFO(B):	[100 101 102]
Clean:	[]
Dirty:	[1]
Memory:	[1 102 3 4 5 6 100 101] (order does not matter, of course)

After processing request 11 (and 12, state does not change):

FIFO(A):	[4 5 6 1]
FIFO(B):	[100 101 102]
Clean:	[3]
Dirty:	[]
Memory:	[1 102 3 4 5 6 100 101] (order does not matter, of course)

After processing request 13:

FIFO(A):	[4 5 6 1]
FIFO(B):	[100 101 102 103]
Clean:	[]
Dirty:	[]
Memory:	[1 102 103 4 5 6 100 101] (order does not matter, of course)

After processing request 14:

FIFO(A):	[4 5 6 1]
FIFO(B):	[101 102 103 104]
Clean:	[]

Dirty:	[]
Memory:	[1 102 103 4 5 6 104 101] (order does not matter, of course)

After processing request 15:

FIFO(A):	[4 5 6 1]
FIFO(B):	[102 103 104 105]
Clean:	[]
Dirty:	[]
Memory:	[1 102 103 4 5 6 104 105] (order does not matter, of course)