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

Due date: Friday, February 19th on Canvas.

Project Description:

In this project you are to evaluate how real applications respond to the ***vms*** and ***lru*** page replacement algorithms. For this, you are to write a memory simulator and evaluate memory performance using provided traces from real applications. Assume that all pages and page frames are 4 KB (4096 bytes).

This is a *group* project. You may work in teams of 2.

You must submit a tar file named `p2-<netid>.tar` containing the following:

- Code in C (`memsim.c`) that implements the following functions:
  - `vms()` – implements the VMS' second change page replacement policy. You can find its description in the textbook (page 249)
  - `lru()` – implements the least recently used algorithm.
  - `main()` – reads command line arguments and calls the corresponding function

  You may use additional helper functions if needed. Do not include your name(s) in the source files.
- A `makefile` for easy compilation.
- A 2-page PDF report, but not more than 3 pages, named `report.pdf`, that includes:
  - Your name(s).
  - A report of the results and detailed interpretation of the results.

Input:

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);
```

Command line options:

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> <vms|lru> <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 *total memory frames, events in trace, total disk reads, and total disk writes*.

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 the debug output, it is simply there to help you debug and test your code.

Example Execution:

```
> tar xf p1-hpalombo.tar
> ls
makefile memsim.c report.pdf
> make
...
> ls
makefile memsim memsim.c report.pdf
> memsim test.trace 12 lru quiet
total memory frames: 12
events in trace: 1002050
total disk reads: 1751
total disk writes: 932
```

Project Report:

Briefly summarize the structure and implementation of your simulator.

Present and analyze your results using both tables and graphs (use Excel or Matlab or another tool of your choice).

Answer the following questions:

- *How much memory does each traced program actually need?*
- *Which page replacement algorithm works best?*
- *Does one algorithm work best in all situations? If not, what situations favor what algorithm?*

Hints:

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.

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.

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

It might be impossible to run your simulator with all possible inputs, so you can think carefully about what measurements you need to answer the questions above. 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. 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, $2^{\wedge 2}$, $2^{\wedge 3,}$ $2^{\wedge 4}$, ..., $2^{\wedge 10}$ 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 2TB of memory for a process is not particularly common these days.

Describe 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 the performance of the 2 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.

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

Grading Rubric:

All projects will be tested in the C4 lab.

We will be using automated scripts to grade your project. Therefore, it is very important that your submission meets the requirements exactly as described above.

Percentage(%)	Rubric Comment
5%	General requirements (file names, tar file, directory structure)
10%	Compilation and execution (makefile is present, code compiles and runs without errors)
30%	VMS
30%	LRU
5%	Code is organized and well-commented
20%	Report