Assignment 2
Memory Management

In this assignment, you are tasked with simulating a virtual memory management system.

Broadly speaking, your simulation will have the following responsibilities:
1) Simulate a paging system
2) Implement three different page replacement algorithms
3) Implement a variable page size
4) Implement demand/prepaging
5) Record page swaps during a run

Let's discuss each phase of this assignment in turn.

1: Simulate a paging system

This simulation will use the idea of a 'memory location' atomic unit. The pages in our simulation will be expressed in terms of this idea. Thus, if our page size is 2, we have two memory locations on each page.

Main memory, in our program, will be **512** memory locations big.

Supplied with this assignment are two files, *programlist*, which contains the list of programs that we will be loading into main memory, and *programtrace*, which contains a deterministic series of memory access that emulates a real systems memory usage (additionally, program scheduling is implicit through use of this file). Both of these are given in terms of memory locations as well. The former in terms of size, the latter in terms of which location is requested.

You will need to create page tables for every program listed in *programlist*. Each page in each page table will need to be given a unique name or number (with respect to *all* pages in *all* page tables) so you can quickly determine if it is present in main memory or not.

Once you have these, you will perform a default loading of memory before we start reading *programtrace* commands. That is, we will load an initial resident set of each program's page table into main memory.

Finally, you will need to begin reading in *programtrace*. Each line of this file represents a memory location request within a program. You will need to translate this location into the unique page number that you stored in the page tables you made later, and figure out if the requested page is in memory or not. If it is, simply continue with the next command in *programtrace*. If it is not, record that a page swap was made, and initiate a replacement algorithm. (Page swaps are the metric we will use to compare various page replacement methods).

2: Implement three different page replacement algorithms

Clock Based policy

Use the simple version of this algorithm with only one use bit. This can be found in the text, or the slides.

First In, First Out(FIFO)

The oldest page in memory will be the first to leave when a page swap is performed.

Least Recently Used

The page with the oldest access time will be replaced in memory when a page swap is performed.

3: Implement a variable page size

This affects not only how many pages each program will take up, but also the 'size' of main memory. If the page size is 2, our main memory will have 256 available page spots. This simulation should be able to use page sizes that are powers of 2, up to a max of 32.

4: Implement demand paging and prepaging

Demand paging replaces 1 page with the requested page during a page fault. Prepaging will bring 2 pages into memory for every swap: the requested page and the next contiguous page.

5:Record page swaps

Anytime we read a memory location, translate it to page number, and do not find it in main memory, that means we need to initiate a page swap. We will record this in a counter form during the run of the program. It will be used as a metric of each algorithm's performance. This makes sense: if a particular algorithm is using the disk less to swap pages into memory, the whole system will be running faster

Write a Report:

Once you have implemented all of this, please try running each algorithm (clock, LRU, FIFO) with page sizes of 1,2,4,8, and 16. Obtain simulation results for both demand paging and prepaging. In other words, you will obtain 10 measurements (runs) for each page replacement policy (LRU, clock, FIFO). Plot all these results on a graph (x:page size, y: page swaps) which will have 6 curves altogether (each with a different legend). Write a 1-2 page report detailing your findings, including a discussion of complexity of each algorithm vs. its performance benefit. Also, explain how prepaging affects the performance.

Submit your report electronically as a .doc file.

Implementation Details:

- We will be implementing "Variable Allocation with Global Scope" as explained in slide 27 at http://web.mst.edu/~ercal/284/slides-1/CHAP8.ppt. In other words, all 3 replacement algorithms will have a global scope and the number of pages held by a process will change dynamically (not fixed!) over the lifetime of the process.
- Use a struct to contain any data structures you're going to be passing around...bookkeeping arrays, main memory, etc.
- Use a separate struct to keep all the page tables you generate. You can keep page tables in a bunch of different ways, but some good ideas are:
  - A vector which contains arrays
  - an array of arrays
  - For both of these, keep the array sizes in the page table structs, so you have them. Vector has its own size function, so you don't need to keep that.
- You can name your pages in any unique fashion you like, but numbers are pretty easy. After you finish parsing program 1 which has pages 1...n, don't restart your number. Program 2 should have pages n+1.....m , program 3 should have m+1....o.. etc
- To look up a memory location, simply divide its number by page size. Thus, Program 0's 120th memory location in a pagesize=4 system would be on page 30. 121 would be 30.25(second location on page 30. simply take the floor or integer division)
- Once you have that, you can look up absolute page number. Go to program 0's page table, and look in spot 30 for that page's unique identifier. Then see if its in RAM.
- For LRU and FIFO, time() and clock() functions are not sufficiently sensitive to timestamp memory accesses. Use an external library if you like, or simply keep a global counter that keeps track of memory accesses. This will be a relative measure of age. In real operating system, this counter would have some kind of hardware implementation. Keep in mind this counter may grow very large. Make it unsigned and long to give it room to grow, it should not overflow with the files we are supplying.
- Please make all options settable on the command line. A typical command line should look like this:
  - `./memorysimulator programlist commandlist 2 lru d`
- Please check for usage errors, and print a nice error. Do not let seg faults just print out if the program is run with no arguments. Also, don't just check for a minimum argc. If the arguments don't make sense, print an error and quit.
- The files are available at:
  - http://web.mst.edu/~ercal/284/programlist.txt
  - http://web.mst.edu/~ercal/284/programtrace.txt
- There will be a **recitation session**. In addition, please don't hesitate to address questions to cljvf9@mail.mst.edu. Help for those who ask questions well before the deadline will be much more likely than those who are emailing us at 11:00PM on the due date with subject lines like "IT DOESNT WORK!!eleven!! HELP?!"
- A grading rubric for this assignment will be made available soon.

General Algorithm for the assignment:

Step One: Declare your main memory struct. Note that the size of your main memory array will vary depending on your page size. It may be useful to create a constructor that takes a page size and figures everything out from there. In addition, this **struct** should contain any data structures you need to perform lru, clock, and fifo.

Step Two: Read in the programlist file using ifstream. Again, these values are in terms of memory locations, not pages, so you'll have to divide these values by page size. You need to keep these in a vector or array of arrays, as I said. In addition, you will probably want to keep a reference to the size of each array in the list- that way you will never run off the end for various calculations. You may want to make this part of your main memory **struct**, or make a new one. In addition, this is where you make unique page references. Fill these arrays with unique non duplicated names of some time, so you can look up which pages are referred to as what at all times.

Step Three: Default load the memory. Figure out how many pages each program should get in the main memory (divided equally), and load those pages into memory. For example, if there are 10 programs and page size=1, each program will get 512/10 □ 51 pages. These should be the first pages in the program. If a program doesn't have enough pages for its default load, leave a blank spot. At this time, you should also initialize the values in your bookkeeping data structure. (for instance, if lru, just write 0.....n, in your timestamp array) Of course, by "load the memory", we mean put the unique page names for each programs default load into your main memory array.

Step Four: Perform the actions recorded in programtrace. Again, using an ifstream get data from the programtrace file. For each line, you'll be doing the following things: increment the programcounter for timestamping purposes, then figure out if the page is currently in your memory array. If it is, do nothing (other than update the clock or timestamp array if you're running lru or clock). If it is not, run the appropriate replacement algorithm. You can probably define your lru, clock, and fifo functions such that they can take only the memorystruct and the requested page that was not found as arguments

Note: As I stated in the notes above, the programtrace is in memory locations, so to figure out what page you're looking for, you have to divide it by page size, and take the floor or integer division, then go to that programs page table(array) and find its unique name. Then you search for that unique page name in the main memory array.

After you've finished reading the file, output the number of page swaps.