This is a team project. The submission has to be the result of the work of the team members only. No external sources are allowed. No collaboration is allowed between different teams. This project is worth 10% of your final grade. The due date for this assignment is 11:59pm Friday, April 13, 2018.

Late penalty: Late assignment (even by 2 seconds) will be given a -25% decrease penalty per day, for the first 2 days after the deadline. So, if you send an assignment 1 second late, you will receive 75% of your grade for the assignment. If you send it, 24 hours, and 1 second late, you will receive 50% of your grade for the assignment etc. After 48hrs from the deadline there will be a -90% decrease penalty, so you will receive 10% of your grade.

The goal of this assignment is to simulate a virtual memory management system. You will experiment with different page replacement algorithms. Your program must accept the following parameters at the command prompt in the order specified:

- P1: Size of pages, i.e., # of memory locations on each page.
- P2: FIFO, LRU, or Clock for type of page replacement algorithms:
  - FIFO: First-in, First-Out
  - LRU: Least Recently Used
  - Clock: Clock Page Replacement
- P3: flag to turn on/off pre-paging. If pre-paging is not turned on, we use demand paging by default.
  - +: turn it on
  - -: turn it off

Two files, *plist* and *ptrace* are supplied (attached in the assignment description). They should be included in the parameters too. A typical command line should look like this:

VMsimulator plist ptrace 2 FIFO +

It runs the simulator algorithm for page size = 2, FIFO replacement algorithm with pre-paging.

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/pre-paging

5) Record page swaps during a run

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

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 hold **512** memory locations.

Supplied with this assignment are two files

- *plist,* contains the list of programs that we will be loading into main memory. Each line in *plist* is of the format (pID, Total#MemoryLocation), which specifies the total number of memory locations that each program needs.
- *ptrace,* contains a deterministic series of memory access that emulates a real systems memory usage. Each line in *ptrace* is of the format (pID, ReferencedMemoryLocation), which specifies the memory location that the program requests.

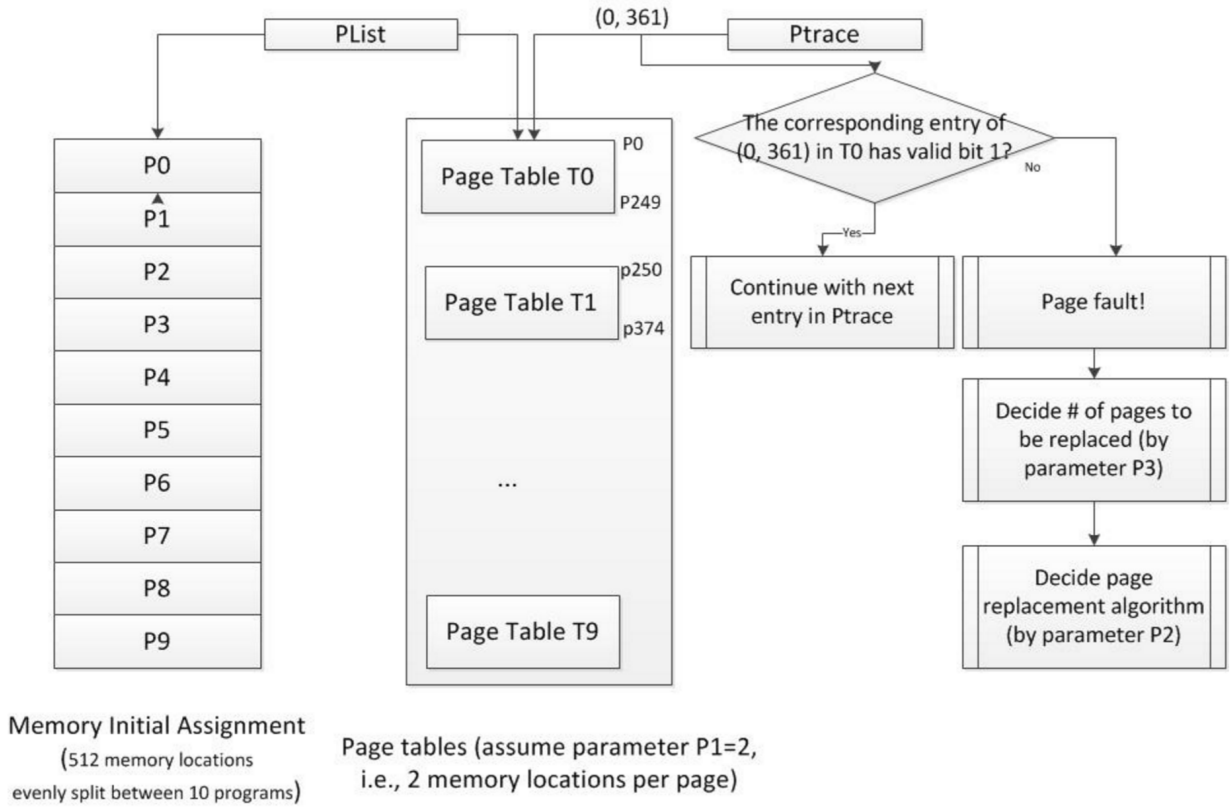
You will need to create page tables for every program listed in *plist*, with a page table for each program. Each page in each page table will need to be given a unique name or number (with respect to *all* pages in *all* page tables) so you can quickly determine if it is present in main memory or not. The size of each page table is decided by the number of pages of the program; it can be calculated by dividing the total number of memory locations of the program (in the *plist* file) by the page size (input parameter P1).

Each page table is of the data structure as following:

page number	valid bit	Last time accessed
N1	0 (i.e., it is not in the memory)	T1
N2	1 (i.e., it is in the memory)	T2

Once you have the page tables, you will perform a default loading of memory before start reading the pages as indicated in *ptrace*. That is, you will load an initial resident set of each program's page table into main memory. The main memory is divided equally among all programs in *plist*. Figure out how many pages each program should get into its assigned main memory part, and load those pages into memory. These should be the first pages in the program. If a program doesn't have enough pages for its default load, leave its unused load blank. After initializing memory allocation, you update the page tables (i.e., set valid bit of corresponding entries in page tables to 1) according to the page assignment.

Finally, you will need to begin reading in *ptrace*. 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 *ptrace*. If it is not, record that a page swap was made, and initiate a replacement algorithm. For each program, the pages to be replaced should be picked from those pages allocated to itself (which is called as the *local* page allocation policy).



Implementation details:

- You can use *ifstream* to read from both *plist* and *ptrace* files. Google ifstream for its reference. You can also use other methods to parse both files.

- Use a structure to record the pages in main memory. For every time a page is loaded into the memory, its unique page names will be put into this data structure.
- Each program in *plist* has an array as its page table. There are various ways to implement page tables. Some good ideas are:
    - A vector which contains arrays,
    - An array of arrays.

- 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, continue from n+1 for the next program. Program 2 should have pages n+1.....m, program 3 should have m+1....o.. etc.

- Translation of a memory location of a program to its unique page number consists of two steps:

    - Step 1: look up the memory location by dividing the location by page size. Thus, Program 0's 6th memory location in a pagesize=4 system would be on page 2, which can be calculated by taking the ceiling of 6/4.
    - Step 2: look up absolute page number. Go to program 0's page table, and look for that page's unique identifier.

Implementation of 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.

Implementation details:

- 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. 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.

Page of Variable Sizes

This affects not only how many pages each program will take up, but also the 'size' of main memory. If the page size is 4, our main memory will have 128 available page spots. This simulation should be able to use page sizes that are powers of 2, up to a max of 32.

Demand paging and pre-paging

- Demand paging: replaces 1 page with the requested page during a page fault.
- Pre-paging: brings 2 pages into memory for every swap: the requested page and the next contiguous page. If the next contiguous page is in the memory already, you need to repeat looking for the next page, until you either reach a page that is not in the memory or you have checked all pages.

Record page swaps

Anytime a memory location is read, it will be translated to page number. If it is not found in main memory, a page swap must be initiated. 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.

Experimentation

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, demand paging. Then plot all three of these on a graph (x:page size, y: page swaps). Write a 1-2 page report detailing your findings, including a discussion of complexity of each algorithm vs. its performance benefit. Then repeat the above procedure with pre-paging. Compare the results with demand paging. Discuss these results as well.

Implementation Requirements:

• The program must be written in C or C++, and run on a linux machine. (coordinate with your CA)

• ALL source code you submit must be well documented (documentation is an indicator of understanding!)
• Programs that cannot be compiled by CA will receive an automatic grade of zero (0).

Grading

- Simulation of page system (25 points)
  - Main memory needs to be correctly created in terms of page size
  - Page tables need to be correctly created, all pages have unique names
  - Must handle all page sizes that are powers of two correctly
  - Default load correctly (consider case of program smaller than default load size)
- Page replacement algorithms: FIFO, LRU, and clock (30 points, 10 points for each)
- Implementation of pre-paging & demand paging (20 points)
- Experimentation and analysis of results (20 points)
  - Graph page faults as a function of page size across all algorithms(5 pts)
  - Data points at 1 - 2 - 4 - 8 - 16 page size
  - All algorithms represented
  - Explain your results(15 points)
    - Discuss what you expected to see vs. what you actually saw
    - If the results do not mirror your expectations / textbooks claims, discuss why
    - Discuss the relative complexity of programming each algorithm
    - Discuss how the data might have changed if a completely random memory access trace had been supplied
- Documentation (comments in code) (5 points)

Submission:

- Create a zip file of your assignment.
- Submit the zip file using Canvas. Login using your Stevens Pipeline account which should have been created for you upon enrollment.
- Ensure your name and login name appear in each file you submit.
- You are advised to commit your contributions to the bitbucket/Github.

Hints and help:

- Start early. Tackle the problem in units.
- Use the Unix *man* command to find out about how to use the various function calls mentioned above.
- Please don't hesitate to address questions to the CAs or the instructor.