CS 483 – Operating Systems
Spring 2023

PEX 3: Simulating Page Replacement - 80 Points

Due: 12 May 2023 (T40) @ 23:59

Help Policy

AUTHORIZED RESOURCES: Any, except another cadet's assignment or published solutions to the assigned problem.

NOTE:

- Never copy another person's work and submit it as your own. Here are a few blatant examples of copying:
  - Making an electronic copy of another cadet's solution and then modifying it slightly to make it appear as your own work.
  - Reading a printout or other source of another cadet's work as you implement your solution.
  - Completing your entire solution by following explicit instructions from another cadet, while he/she refers to his/her own solution
- Do not jointly implement a solution.
- Helping your classmates learn and understand the homework concepts is encouraged, but extensive assistance should generally be provided by DFCS instructors. Only provide assistance up to your depth of understanding, beyond which assistance by more qualified individuals is more appropriate and will result in greater learning. If you have to look at your solution while giving help, you are most likely beyond your depth of understanding.
- Help your classmates maintain their integrity by never placing them in a compromising position. Do not give your solution to another cadet in any form (hard copy, soft copy, or verbal).
- You **may not** solve the entire PEX with ChatGPT or any other "generative AI" technology.
- You **may** ask the generative AI questions to fix your code, and you may copy and paste your code into the generative AI. You must document what you asked the generative AI, provide a description of the error in your code, and describe how you were able to fix it.
- You **may** ask the generative AI questions about how to program a general concept related to the PEX
- **DFCS will recommend a grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others. Allowing another cadet to see your assignment to help them will result in a zero on this assignment.**

Documentation Policy

- You must document all help received from sources other than your instructor or instructor-provided course materials (including your textbook).
- The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance.
- If no help was received on this assignment, the documentation statement must state "NONE."

- If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why.
- Vague documentation statements must be corrected before the assignment will be graded and will result in a grade deduction equal to 5% (ceiling) of the total possible points.

OBJECTIVES

- Understand the basic concepts of demand paging.
- Be able to implement page replacement algorithms.
- Understand the relationship between frame allocation and page-fault rate.
- Understand the relationship between page size and page-fault rate.

OVERVIEW

In this project, you will build a simulator to test different page frame allocations using the Least Recently Used (LRU) page replacement algorithm. You will use a reference stream captured on an Intel Pentium system to find the page fault rates as the amount of physical memory allocated to the process varies up to 4MB for four different frame sizes. You will produce graphs in MS Excel (or LibreOffice Calc program, included on your Ubuntu VM) showing the impact of increasing the number of frames allocated for each frame size. You will then present your results in a report created in MS Word (or LibreOffice Writer, included on your Linux VM) and make a recommendation on what size frame to use and how many frames should be allocated to the reference process.

SUBMISSION INSTRUCTIONS

Submit your PEX via Github classroom. Add your

IMPORTANT: You will not be able to submit your project if you include the trace file. Be sure to remove that file prior to zipping and submission (`eon_rush_0.tr`).

NOTE:

- Your documentation statement must be in your LRU.c file. Be sure you are thorough in your documentation of WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance.
- There is a file size limit when pushing to Github. You will not be able to upload your trace file. You must unstage changes to eon_rush_0.tr before pushing to Github.
- On your final commit, ensure you have included your Word document reports in your Github repository. You can visit Github.com and manually add your word document to your repository. Contact your instructor if you need help with this.

GENERAL REQUIREMENTS

- You must implement this program in C on your Ubuntu virtual machine.
- You must use VSCode for this project.
  - Your project should compile cleanly (i.e. free of any warnings or errors)
- As this will be a single-threaded program, you should not have any need for global variables.
- Properly allocate and free memory as necessary.

SIMULATION REQUIREMENTS

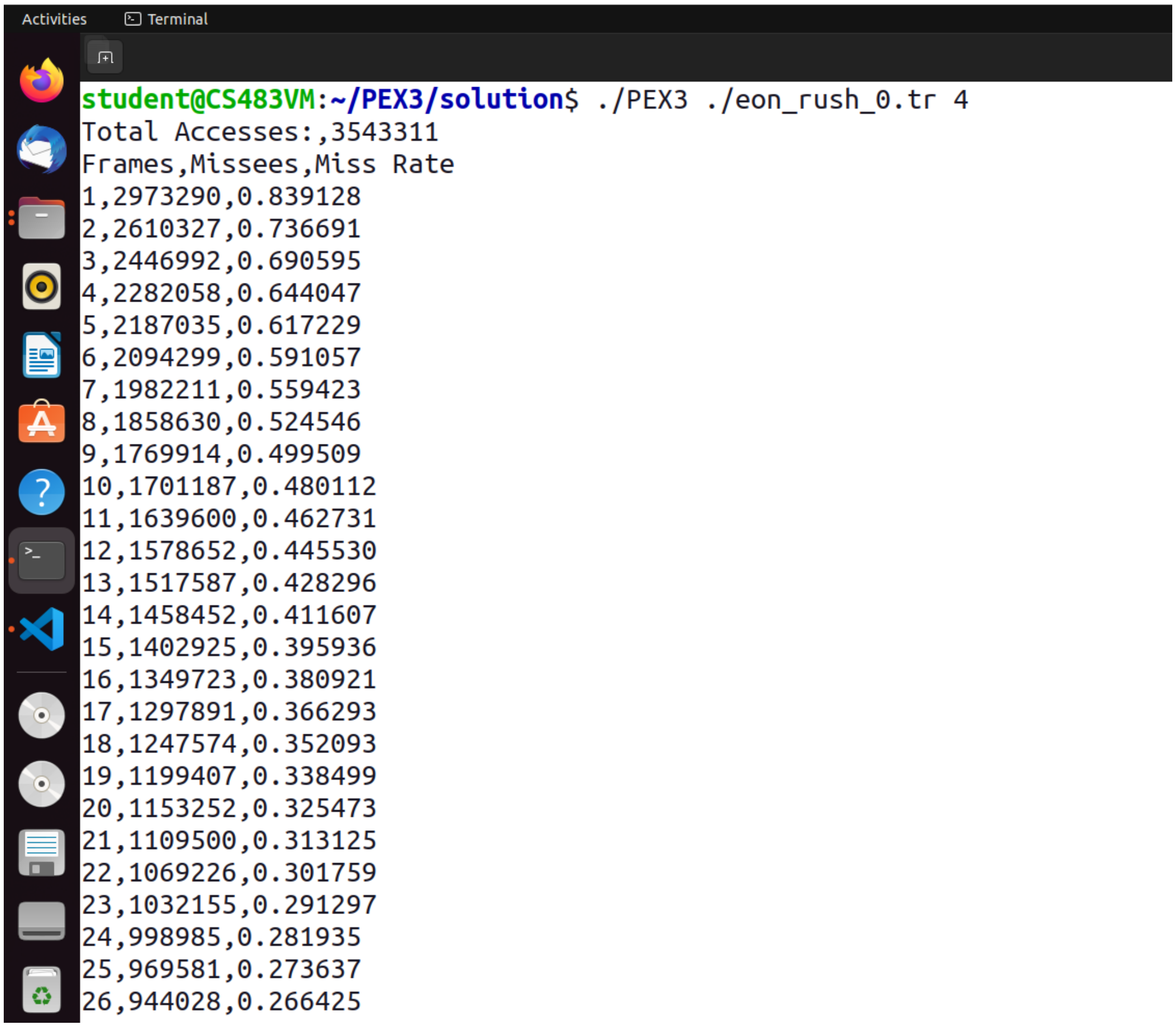
Download the template zip file from the PEX3 page on Github Classroom. It contains three files. `eon_rush_0.zip` contains the trace file `eon_rush_0.tr`. Ensure that you unzip `eon_rush_0.zip` to obtain `eon_rush_0.tr`. The trace file contains, among other items, the physical addresses generated by the reference process. (NOTE: The trace file is 85,039,452 bytes large and contains 3,543,311 memory accesses). The other files, `byutr.h` and `LRU.c` include code that reads the trace file, as it is stored in binary format with several fields that are not needed for our purposes. Note: byutr = Brigham Young University TRace. The process trace was produced at BYU.

For each of four possible page frame sizes: 512 bytes, 1 KB, 2 KB, and 4KB. For each possible allocation of page frames from 1 to n. n is the maximum number of pages frames which can be allocated to the reference process (n * frame size = 4 MB). You will simulate a process, using the provided reference stream, and track the number of page faults. That is, for each reference in the stream, you will check whether the referenced page is already in memory. If it is not in memory, you will insert it into memory. You will use LRU to determine what page to replace if physical memory is full.

In order to calculate the page-fault rate for each memory allocation, you will track how many times the system faults and divide by the total number of references. You will repeat this calculation for all possible allocations of frames up to 4MB. Ideally, you should discover that page-fault rate is a function of the number of frames allocated (more memory implies fewer page-faults) and that the frame size also affects the page-fault rate.

Output the number of page frames, number of page faults, and the page-fault rate for each memory size. Each number of frames allocated should be on its own line, and all values within a line should be separated by a comma. Your output should resemble the output below (note the comma in "Total Accesses:,3543311" makes the string and the number separate values).

Processing the trace file will take several minutes. Print a progress indication of how many memory references were read from the trace file. We have provided commented code in pex3.c to print this status message. Uncomment the code to use it.

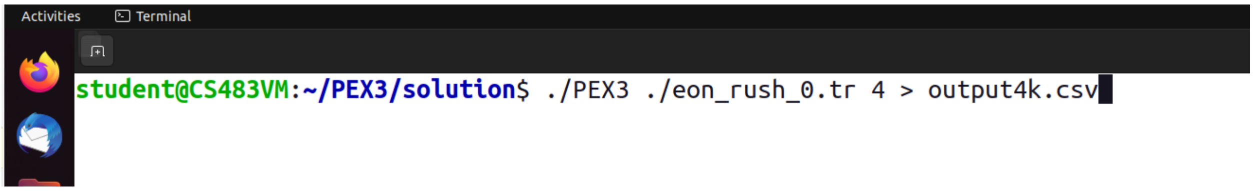


```
student@CS483VM:~/PEX3/solution$ ./PEX3 ./eon_rush_0.tr 4
Total Accesses:,3543311
Frames,Missees,Miss Rate
1,2973290,0.839128
2,2610327,0.736691
3,2446992,0.690595
4,2282058,0.644047
5,2187035,0.617229
6,2094299,0.591057
7,1982211,0.559423
8,1858630,0.524546
9,1769914,0.499509
10,1701187,0.480112
11,1639600,0.462731
12,1578652,0.445530
13,1517587,0.428296
14,1458452,0.411607
15,1402925,0.395936
16,1349723,0.380921
17,1297891,0.366293
18,1247574,0.352093
19,1199407,0.338499
20,1153252,0.325473
21,1109500,0.313125
22,1069226,0.301759
23,1032155,0.291297
24,998985,0.281935
25,969581,0.273637
26,944028,0.266425
```

Figure 1: Example PEX3 run with 4kB page and page frame size

REPORT REQUIREMENTS

Run your program for each of the 4 frame sizes, redirecting your output to a file each time. This is done with the redirect operator ">" as shown below. Copy this file to your host (Windows) machine and open it using Microsoft Excel, being sure to include commas as a delimiter. Create two graphs for each frame size: 1) the number of frames vs the number of page faults and 2) the number of frames vs page-fault rate.



```
student@CS483VM:~/PEX3/solution$ ./PEX3 ./eon_rush_0.tr 4 > output4k.csv
```

Figure 2: Example redirecting PEX3 output to a .file for the final report. Name each .csv file in accordance with the page size.

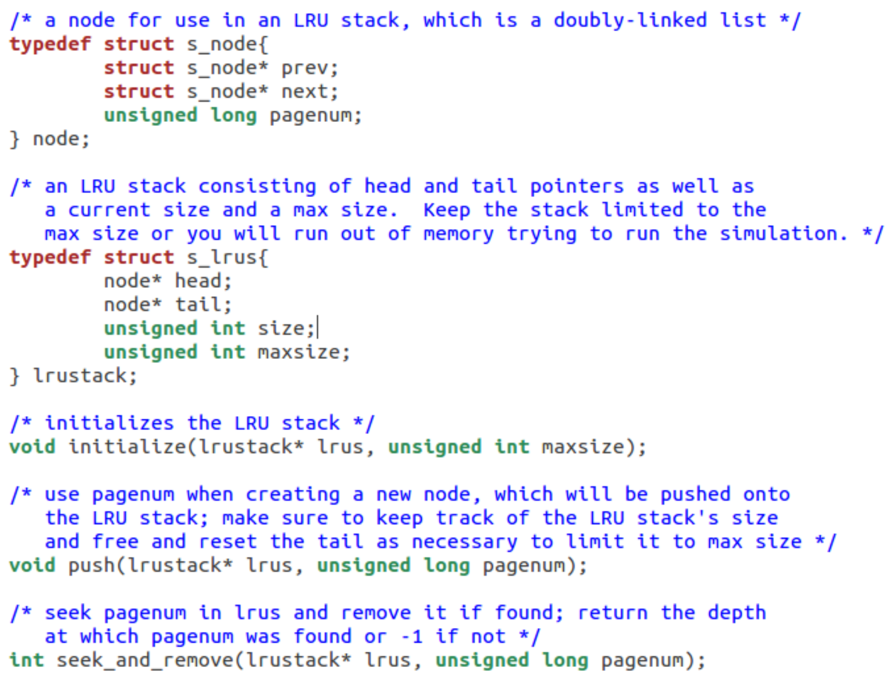
Based on your results, write a short (1-2 pages, a minimum of 250 words) recommendation for the "optimum" size (number of frames to allocate) and number of frames for this process. Be sure to back up your recommendation. Assume there is a requirement to keep the page-fault rate below 10%. Be sure to address any interesting features on your graphs or relationships between them.

HINTS/PLAN OF ATTACK

At first glance it appears you need a separate simulation "run" for each size of memory (number of frames allocated), but actually you can do this in a single run using a LRU Stack (more on this on the next page). Review the Zybook's description of a stack implementation of LRU in section 27.1.

Most of the programming portion of this PEX will consist of the building the stack Abstract Data Type (ADT). You will need to be able to find a value in the stack, report its depth, and move it to the top of the stack. If the value is not in the stack, you will push a new node onto the top of the stack and report back an invalid depth (e.g. -1). Print out the referenced page and the depth returned by your stack function. Desk-check your program by tracing the use of the stack by hand and ensuring correct functionality for the first several references.

Here are some structs and function signatures you may want to use (`LRUstack.c` `LRUstack.h` are blank files from which you can build your ADT)



```c
/* a node for use in an LRU stack, which is a doubly-linked list */
typedef struct s_node{
        struct s_node* prev;
        struct s_node* next;
        unsigned long pagenum;
} node;

/* an LRU stack consisting of head and tail pointers as well as
   a current size and a max size.  Keep the stack limited to the
   max size or you will run out of memory trying to run the simulation. */
typedef struct s_lrus{
        node* head;
        node* tail;
        unsigned int size;
        unsigned int maxsize;
} lrustack;

/* initializes the LRU stack */
void initialize(lrustack* lrus, unsigned int maxsize);

/* use pagenum when creating a new node, which will be pushed onto
   the LRU stack; make sure to keep track of the LRU stack's size
   and free and reset the tail as necessary to limit it to max size */
void push(lrustack* lrus, unsigned long pagenum);

/* seek pagenum in lrus and remove it if found; return the depth
   at which pagenum was found or -1 if not */
int seek_and_remove(lrustack* lrus, unsigned long pagenum);
```

One way to deal with the pass by value nature of C and the need to ensure modifications to the LRU stack are visible in the scope of main is to declare, in main, a variable of type lrustack:

```c
lrustack lrus;
```

and pass `&lrus` to the appropriate functions.

The trick here will be to not limit our stack depth to the number of frames available, and thus never remove a page from the stack as we "swap out" pages. Instead, the LRU stack will become large than the number of page frames available in your simulation. This will allow you to consider the page to have been in memory if its depth in the stack was less than or equal to the number of physical frames available. Otherwise, if the page request is not in the stack or it is deeper than the size of memory allocated to your current simulation you count this page request as a page fault. In this way, you will be able to simulate all memory sizes simultaneously and only require a single pass through the trace file.

Use an array to track the number of page faults for each memory size. If your stack function reports a "-1", this is a page fault for all memory sizes and every element in the array needs to be incremented. If your stack function reports back a valid depth *d*, then this is a hit for all memory sizes greater than or equal to *d*, but a miss for all frame sizes less than *d*. Therefore, increment the array elements for sizes 1 to *d*-1.

EXPECTATIONS

Requirements 1. Coding Standards a. Proper decomposition, commenting, naming conventions, indentation, no global variables, etc. b. Memory management i. Proper allocation of memory ii. Free of memory leaks c. Coding and submission i. properly compiles program without warnings or errors ii. submission standards followed	**+8**
2. Program Functionality a. Correctly parses the trace file b. Correct LRU functionality c. Correct accounting of page faults & total accesses d. Correctly calculates page-fault rate e. Correctly outputs required data formatted for Microsoft Excel	**+48**
3. Analysis a. Correctly builds graphs b. Graphs are professionally formatted c. Makes a clear recommendation d. Recommendation backed up by discussion of simulation results e. Demonstrates knowledge of domain f. Professionally written i. Free of spelling and grammar errors ii. No first person iii. Active voice **iv. Clear and concise language**	**+24/24**
Penalties 4. Vague/Missing Documentation (5%) 5. Late Submission (25% cap/day)	**-4**
Total	**80**