#### COEN 177: Operating Systems

**Lab assignment 8: Memory Management**

**Objectives**

##### To simulate two kinds of basic page replacement algorithms

##### To evaluate the performance, in terms of miss/hit rate, of these algorithms

##### 

##### **Guidelines**

The goal of this assignment is to gain experience with page replacement (and to a lesser extent, caching) algorithms. In this assignment, your goal is to write programs that simulate page replacement algorithms. Your initial program is to accept at least one numeric command-line parameter, which it will use as the number of available page frames. In this case:

int main(int argc, char \*argv[]){

int cacheSize = atoi(argv[1]); // Size of Cache passed by user

A simulation of a page replacement algorithm will then use cacheSize as a number of pages/blocks of memory/cache size. This value can be read for example using “lru” as follows:

**$lru 27**

or

**$simulate -lru 7**

Your program should expect page requests to arrive on standard input (**stdin)**, so a basic **fgets()**, or **scanf()** call cab be used to read in the unsigned integer page numbers being requested. Assuming you have a sequence of page numbers in a text file called "testInput.txt" you should be able to run your simulator by typing:

**$cat testInput.txt| lru 42**

**You’ll test with the accesses.txt file** in the end to submit a report of your findings. **Skeleton code is provided** for you in skeleton.c also to help you get started.

##### **Generate testInput.txt file**

1. Write a C program to generate testInput.txt file. You may use the following code snippet:

int main(int argc, char \*argv[]) {

FILE \*fp;

char buffer [sizeof(int)];

int i;

fp = fopen("testInput.txt", "w");

for (i=0; i<someNumber; i++){

sprintf(buffer, "%d\n", rand()%capNumber);

fputs(buffer, fp);

}

fclose(fp);

return 0;

}

##### **Date types and definitions**

1. Define the following data types in your page replacement code:

typedef struct {

int pageno;

. . .

} ref\_page;

ref\_page cache[cacheSize]; // Cache that stores pages

char pageCache[100]; // Cache that holds the input from test file

int totalFaults = 0; // keeps track of the total page faults

##### **Read page requests iteratively**

1. Write a C program to read iteratively the output of **$cat testInput.txt** pipelined as a standard input to the executable page replacement program. This can be implemented using

char \*fgets(char \*str, int n, FILE \*stream)

Where fgets( ) library function reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. So, you may capture the page number pipelined to the standard input as follows:

while (fgets(pageCache, 100, stdin)) {

int page\_num = atoi(pageCache); // Stores number read from file as an int

In this case, your program will accept page requests on stdin as individual numbers, one per line, where each number indicates the requested page number. Each program is to further ignore any trailing text on the input lines, or any lines that do not start with a number. Your program terminates its simulation when it encounters an end-of-file.

Test for memory sizes of between 10 and 500 pages.

##### **Page replacement algorithm**

1. Write a program for each of the following replacement algorithms: FIFO, LRU, and Second Chance Page Replacement.

FIFO (First In First Out): is the simplest page replacement algorithm that keeps track of all the pages in the memory in a queue with the oldest page at the front of the queue. On a page fault, it replaces the oldest page that has been present in memory for the longest time. The following code snippet is provided as a guidance. A similar code can be implemented for the LRU and 2nd chance:

bool foundInCache = false;

for (i=0; i<cacheSize; i++){

if (cache[i].pageno == page\_num){

foundInCache = true;

break; //break out loop because found page\_num in cache

}

}

if (foundInCache == false){

//You may print the page that caused the page fault

cache[placeInArray].pageno = page\_num;

totalFaults++

placeInArray++; //Need to keep the value within the cacheSize

}

Check pseudo code[[1]](#footnote-1)

LRU (Least Recently Used): replaces the page that was previously least recently used. You may need to use an index to keep track of the most recently used page. Check pseudo code[[2]](#footnote-2)

2nd Chance or Clock: gives every page a second chance in the sense that an old page that has been referenced is likely in use and therefore, will not swapped out over a new page that has not been referenced. A queue may be created and will be checked, but instead of paging the page out, a referenced bit would be checked to see if it is set. The page would be swapped out if the reference bit is not set, otherwise the page is inserted at the back of the queue. A counter needs to be implemented to keep track of the reference bit. Check pseudo code.[[3]](#footnote-3)

##### **Expected Output**

1. The output of a page replacement program will be every page number that was **not** found to be in the cache. In other words, the output will be a sequence of page numbers that represents all the incoming requests that resulted in a page fault. Using your program, you should be able to get two numbers from the Unix command line (by counting the number of lines read from the input file, and the number of lines produced by your simulator). The first of these numbers is the total number of page/block requests your simulator program has received (you get this by counting the number of valid lines in your input file), and the second number is how many of these page requests did result in a page fault (you get this by counting the number of lines produced as output by your program - which is faithfully reproducing the page replacement algorithm's behavior).

Once again, the size of the memory being managed by your program (the number of page frames, or the size of the cache if you treat this as a caching algorithm) is to be accepted as a command-line argument to your program. Any status output (e.g., messages you wish to print for debugging/user) should be sent to stderr (standard error, in other words, it should be possible to use your program and see nothing in standard output other than the page-faults/cache-misses, by redirecting only stdout).

##### **Test your Page Replacement Algorithms using a series of commands in a shell script**

1. In lab 1, you learned about shell scripting as a tool that allows you to execute a series of commands by running a shell program that contains them instead of typing all commands. Create a .sh file to generate different test cases, e.g. Test this first with your own testInput.txt file you made in step 1, to help you with debugging you can scale down your testInput.txt file to 10-20 numbers and use the part of the script below that runs on a cache size of 10, replicate it by hand and see if there are any problems you find. Then, run the entire script with all the cache sizes on **accesses.txt** file – you will use these results to create graphs and tables to submit in your final report.

#!/bin/bash

make;

echo "----------FIFO----------"

cat testInput.txt | ./fifo 10

echo "----------End FIFO----------"

echo

echo "----------LRU----------"

cat testInput.txt | ./lru 10

echo "----------End LRU----------"

echo

echo "----------Second Chance----------"

cat testInput.txt | ./sec\_chance 10

echo "----------End Second Chance----------"

echo "FIFO 10K Test with cache size = 10, 50, 100, 250, 500"

cat accesses.txt | ./fifo 10 | wc -l

cat accesses.txt | ./fifo 50 | wc -l

cat accesses.txt | ./fifo 100 | wc -l

cat accesses.txt | ./fifo 250 | wc -l

cat accesses.txt | ./fifo 500 | wc -l

echo

echo "LRU 10K Test with cache size = 10, 50, 100, 250, 500"

cat accesses.txt | ./lru 10 | wc -l

cat accesses.txt | ./lru 50 | wc -l

cat accesses.txt | ./lru 100 | wc -l

cat accesses.txt | ./lru 250 | wc -l

cat accesses.txt | ./lru 500 | wc -l

echo

echo "Second Chance 10K Test with cache size = 10, 50, 100, 250, 500"

cat accesses.txt | ./sec\_chance 10 | wc -l

cat accesses.txt | ./sec\_chance 50 | wc -l

cat accesses.txt | ./sec\_chance 100 | wc -l

cat accesses.txt | ./sec\_chance 250 | wc -l

cat accesses.txt | ./sec\_chance 500 | wc -l

echo

make clean;

echo

make is a utility that requires a Makefile, which defines set of tasks to be executed, e.g. your Makefile may include:

all: fifo.c lru.c sec\_chance.c

gcc -o lru lru.c

gcc -o fifo fifo.c

gcc -o sec\_chance sec\_chance.c

clean:; rm -f \*.out lru fifo sec\_chance

**Requirements to complete the lab**

1. Show the TA your running page replacement simulators.
2. Write up a description of your implementations and **sample miss-rate (page fault rate)** results, and submit it alongside your code. This portion of the assignment is as critical, if not more so, than the actual implementation of your solution.
3. Provide a complete write-up that will include a test of your solutions and a comparison of **the hit rates (1-fault rate)** for the different algorithms you have implemented. Create a table and plot a graph of the results you got from running your code with the accesses.txt file to represent your findings.
4. Include the output results of running step 6 on your own testInput.txt file, you don’t need to convert these results into graphs and tables though.
5. Submit all source code

Note: Your tests use random numbers but be aware that assignments are tested against a test file, so it's a good idea to study your results carefully and to pay particular attention to verifying the correct behavior of your replacement algorithms.

1. https://www.geeksforgeeks.org/program-page-replacement-algorithms-set-2-fifo/ [↑](#footnote-ref-1)
2. https://www.geeksforgeeks.org/program-for-least-recently-used-lru-page-replacement-algorithm/ [↑](#footnote-ref-2)
3. <https://www.geeksforgeeks.org/second-chance-or-clock-page-replacement-policy/> [↑](#footnote-ref-3)