# /COEN 177: Operating Systems
# Lab 7: Memory Management – basic page replacement algorithms

## Objectives

1. To simulate three kinds of basic page replacement algorithms
2. 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 can 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**

## Generate testInput.txt file

Step 1. [5 points] 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<numRequests; i++){
      sprintf(buffer, "%d\n", rand()%maxPageNumber);
      fputs(buffer, fp);
   }
   fclose(fp);
   return 0;
}
```

## Date types and definitions

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

## Read page requests iteratively.

Step 2. [10 points] Write a C program to read the output of `cat testInput.txt` iteratively 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
```

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 ignore further trailing text on the input lines or 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

Step 3. [45 points] 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. This code can be copied and modified to implement 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 at this link.[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 at this link.[2]

2nd Chance or Clock: gives every page a second chance in the sense that an old page that has been referenced at least twice (one miss, then at least one hit) is likely in use and therefore, will not be swapped out over a new page that has only been referenced once (one miss). A queue may be created and checked, but instead of paging the queued page out immediately, a “referenced” bit for that page is checked, with the page being paged out if it is not set or skipped over (unsetting the “referenced” bit) otherwise. Either a struct for each page or a separate boolean array will be needed to keep track of the reference bits. Check pseudo code at this link.[3]

## Expected Output (VERY IMPORTANT, WILL BE GRADED)

The output of a page replacement program will be every page number that was **not** found to be in the cache, and **only** those numbers (do **not** print **anything** else!). In other words, the output will be a sequence of page numbers representing all the incoming requests resulting 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

1 https://www.geeksforgeeks.org/program-page-replacement-algorithms-set-2-fifo/

2 https://www.geeksforgeeks.org/program-for-least-recently-used-lru-page-replacement-algorithm/

3 https://www.geeksforgeeks.org/second-chance-or-clock-page-replacement-policy/

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 your program manages (the number of page frames or the size of the cache if you treat this as a caching algorithm) will 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

Step 4. [10 points] 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.

```
#!/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 testInput10k.txt | ./fifo 10 | wc -l
cat testInput10k.txt | ./fifo 50 | wc -l
cat testInput10k.txt | ./fifo 100 | wc -l
cat testInput10k.txt | ./fifo 250 | wc -l
cat testInput10k.txt | ./fifo 500 | wc -l
echo
echo "LRU 10K Test with cache size = 10, 50, 100, 250, 500"
cat testInput10k.txt | ./lru 10 | wc -l
cat testInput10k.txt | ./lru 50 | wc -l
cat testInput10k.txt | ./lru 100 | wc -l
cat testInput10k.txt | ./lru 250 | wc -l
cat testInput10k.txt | ./lru 500 | wc -l
echo
echo "Second Chance 10K Test with cache size = 10, 50, 100, 250, 500"
cat testInput10k.txt | ./sec_chance 10 | wc -l
cat testInput10k.txt | ./sec_chance 50 | wc -l
cat testInput10k.txt | ./sec_chance 100 | wc -l
cat testInput10k.txt | ./sec_chance 250 | wc -l
cat testInput10k.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. Your Makefile should be some variant of the following:

```
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
```

### Analysis of your implementation

Analyze your implementation by providing the following:

Step 5. [10 points] Describe your implementations and sample miss-rate (page fault rate) results. This portion of the assignment is as critical, if not more so, than the actual implementation of your solution.

Step 6. [20 points] Provide a complete write-up that will include a test of your solutions and a comparison of the hit rates for the different algorithms you have implemented. Create a table and plot a graph to represent your findings.

### Requirements to complete the lab

1. Show the TA your running page replacement simulators.
2. Provide your analysis and report of simulation.

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.