COEN 177: Operating Systems
Lab assignment 8: Memory Management

Objectives

1. To simulate three basic page replacement algorithms
2. To evaluate the comparative performance of these algorithms with different cache sizes

Guidelines

The goal of this assignment is to gain experience with page replacement algorithms. To do this, you will write programs that simulate three prototypical 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, e.g. this code:

```
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 or blocks of memory to use as a cache. This value will be input for example using “lru” as follows:

```
./lru 50
```

Your program should expect page requests to arrive on standard input (`stdin`), so `fgets()`, or `scanf()` can be used to read in the 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 with a cache size of 20 by typing:

```
cat testInput.txt | ./lru 20
```

The report must be written based on the accessesForReport.txt file, not a random test file! Additionally, skeleton code is provided for you in skeleton.c to help you get started.

Generate testInput.txt file

Step 1. Write a C program to generate a 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++){     //Use a number you can test by hand, >possiblePages to force cache hits
      sprintf(buffer, "%d\n", rand()%possiblePages);     //Small for hand testing, >cacheSize to force cache miss
      fputs(buffer, fp);
   }
   fclose(fp);
   return 0;
}
```

Date types and definitions

Step 2. Use these definitions in your page replacement code:

```
typedef struct {
   int pageno;
   //Any other useful values associated with the page, e.g. referenced
} ref_page;

ref_page cache[cacheSize]; // Virtual cache for simulator
char inputBuffer[100]; // Buffer to hold input from test file
int totalFaults = 0; // keeps track of the total page faults
```

Read page requests iteratively

Step 3. Write a C program to read in and reprint the output from a `cat` command piped via standard input to the page replacement program. This can be implemented using the function

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

This 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 piped via standard input as follows:

```
while (fgets(inputBuffer, 100, stdin)) {
int page_num = atoi(inputBuffer); // 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. Your program terminates its simulation when it encounters an end-of-file. For now, simply print out each page request as if it were a cache miss, e.g. with `printf("%d\n", page_num)`.

Page replacement algorithm

Step 4. Write a program for each of the following algorithms: FIFO, LRU, and Second Chance.

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 page that has been present in memory for the longest time. The following code snippet is provided as guidance. Similar code can be written for 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 link[1]

LRU (Least Recently Used): replaces the page that was last used (hit or miss) at the earliest point in time. You may need to implement a counter that increments each time a new page request is made to store in the ref_page struct. Check link[2]

2nd Chance or Clock: gives every page a second chance in the sense that any page that has been the cause of a cache hit is likely going to cause further cache hits, and therefore, should not be swapped out in place of a newer page that has never caused a cache hit. Similar logic to FIFO may be used, but instead of paging the oldest page out immediately, a "reference" bit would be checked. If the bit is unset (0), then the page can be removed, but if it is set (1), the page should be skipped over while unsetting the reference bit (to avoid infinite loops). Check link[3]

Expected Output

Step 5. 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 each caused a page fault. **Do not output any other lines to stdout.** With this behavior, to get the total number of page faults, you must pipe the output to a shell command `wc -l` to count the number of lines of output.

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 for debugging must be sent to stderr** (e.g. fprintf(stderr, "val=%d\n", val)) **or each line will be counted as an additional, erroneous page fault.**

---

[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/

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

Step 6. 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 run your test cases. To debug your program replicate it by hand and see if there are any problems you find. Then, run the entire script with all the cache sizes on **accessesForReport.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 accessesForReport.txt | ./fifo 10 | wc -l
cat accessesForReport.txt | ./fifo 50 | wc -l
cat accessesForReport.txt | ./fifo 100 | wc -l
cat accessesForReport.txt | ./fifo 250 | wc -l
cat accessesForReport.txt | ./fifo 500 | wc -l
echo
echo "LRU 10K Test with cache size = 10, 50, 100, 250, 500"
cat accessesForReport.txt | ./lru 10 | wc -l
cat accessesForReport.txt | ./lru 50 | wc -l
cat accessesForReport.txt | ./lru 100 | wc -l
cat accessesForReport.txt | ./lru 250 | wc -l
cat accessesForReport.txt | ./lru 500 | wc -l
echo
echo "Second Chance 10K Test with cache size = 10, 50, 100, 250, 500"
cat accessesForReport.txt | ./sec_chance 10 | wc -l
cat accessesForReport.txt | ./sec_chance 50 | wc -l
cat accessesForReport.txt | ./sec_chance 100 | wc -l
cat accessesForReport.txt | ./sec_chance 250 | wc -l
cat accessesForReport.txt | ./sec_chance 500 | wc -l
echo
make clean;
echo
```

make is a utility that requires a Makefile, (case sensitive, no file extension) which defines set of tasks to be executed. Your Makefile can be as follows (indentation is required, use real tab characters not spaces!)

```
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 (numFaults/numReqs)** 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-numFaults/numReqs)** for the different algorithms you have implemented. Create a table and plot a

simple graph of the hit rate results you got from running your code with the accesses.txt file to represent your findings.

4. Submit all source code

   Note: Your first three tests use random numbers but the reports are graded against expected values from the accessesForReport.txt file to verify the correct behavior of your replacement algorithms.

Please start each file with a descriptive block that includes at minimum the following information:

```
//Name:
//Date:
//Title: Lab8 –
//Description:
```