FIT2100 Practical #6
Virtual Memory Page Faults
Week 12 Semester 2 2017

Dr Jojo Wong
Lecturer, Faculty of IT
Email: Jojo.Wong@monash.edu

October 16, 2017

Revision Status:

`$Id: FIT2100-Practical-06.tex, Version 1.0 2017/10/12 17:30 Jojo $`

Contents

1 Introduction

In this last practical, you are going to set up simulated experiments to determine the effects on the page fault rates by changes in these parameters: **page size**, **memory size**, and **page replacement algorithm**.

(Note: You are not required to understand all aspects of the C programs provided in this lab. It should be sufficient to be aware of the nature of the programs and their functional descriptions provided here.)

The pre-lab preparation (under the Try It Yourself subsections in Section 2.1) should be completed before attending the lab session. The three practical tasks presented (under Sections 3.1 – 3.3) are to be completed and assessed in the lab.

2 Pre-lab Preparation

2.1 Try it yourself

Create a new directory called `mem` under the working directory on your Unix system. Copy all the C programs (5 in total) from the Moodle site to your newly-created directory `mem`.

Compile these programs into executables with names matching the names of the C programs. Thus, `LRU.c` is compiled to generate the executable LRU. Here are the commands:

```
$ gcc -o nameSorter nameSorter.c
$ gcc -o pageRefGen pageRefGen.c
$ gcc -o FIFO FIFO.c
$ gcc -o LRU LRU.c
$ gcc -o OPT OPT.c
```

3 Practical Tasks

3.1 Task 1: nameSorter.c (1 mark)

The program `nameSorter.c` is our generator program. It randomly generates a lot of names. The version given can generate up to 5000 names. It then sorts these names using a quick

sorting algorithm. You can print unsorted names or sorted list of names on your screen. What you want this program to do is specified through command line arguments (one or more character strings placed in the command line after the command).

Read this program (you should at least read the comment lines to get to know what each function does). Here is how you run the program to create 500 names and list them in an *unsorted* order.

```
$ ./nameSorter 500 listnames
```

(Note: If you check the source code you would see that `listnames` is one of the arguments specified by this program.)

You cannot see 500 names on your screen, can you? Use the following command line:

```
$./nameSorter 500 listnames | more
```

To create 1000 names and list them in a *sorted* order you use the following command:

```
$ ./nameSorter 1000 listsorted
```

(Note: If you check the source code you would see that `listsorted` is also one of the arguments specified by this program.)

Two important data structures in the program `nameSorter.c` are `char` arrays: `list` and `names`. During a sorting exercise, data stored in these arrays are accessed for reading their content and for writing new values into them. The program uses C pointer arithmetic to list these accesses on your screen.

The output of the following command line will be different:

```
$ ./nameSorter 500 listaccess
```

(Note: Similar to `listnames` and `listsorted`, `listaccess` is also one of the arguments specified by the program.)

What you see on your screen now are the (virtual) memory addresses, or logical addresses, of the process being accessed during the sorting. The output uses letter **R** to indicate a read access; **W** to indicate a write access; and **F 000** to indicate finish.

Your task: Find out how many read accesses are there in the list generated by the previous command line. How many are writes?

3.2 Task 2: pageRefGen.c (0 marks)

The file `pageRefGen.c` is a C program from which you generate the executable `pageRefGen`. This filter can read the list of memory accesses generated by the `nameSorter` and convert them into references to the page numbers.

For example, if we know that address `1000` was read and each page is 256-byte long, then the page number `1000/256 = 3` was accessed to read data from the address `1000`. (This means that the address `1000` is within page 3). The command line argument for the program specifies the size (number of bytes) of a page.

Your task: To test the program run the following command:

```
$ ./pageRefGen 256
R 1000 (user input)
R 3 (response of the program)
```

Your task: To continue the testing enter the following lines (and press enter at the end of each line). To quit the program, enter `F 0`.

```
R 9987
R 6740
R 7609
```

Now, we run the program by feeding it input from another program through a pipe.

```
$ ./nameSorter 500 listaccess | ./pageRefGen 256
```

The output of this command line can be considered as a long memory reference string. From Week 9 lectures, you know that we can evaluate a *page replacement* algorithm by running it on a particular memory reference string and computing the number of *page faults* — this is what you are going to do next.

3.3 Task 3: Virtual memory page faults (4 marks)

We provide three programs `FIFO.c`, `LRU.c`, and `OPT.c` to implement page replacement algorithms for simulated virtual memory systems. The following is a brief summary of these algorithms.

- **FIFO (First In First Out)** policy is very simple to implement. It assumes that a page that came in the main memory first will be the first to go out when we need space for some fresh pages.
- **LRU (Least Recently Used)** policy is based on the following premises — if a page has not been used for sometime then it will not find a use in the future too. So the page that is most likely to be never used in the future is the one which has not been used for longest continuous period ending at the present time. Usually finding such a page is a moderately difficult process.
- **OPT (Optimal)** policy is computationally very expensive but provides best results.

What we will do here is to test the page faults that result from the use of these algorithms. A *page fault* happens when the accessed page is not found in the memory and causes the page to be read from the disk. In addition, some page faults may cause a disk write of the modified data to precede the disk read (i.e., write a dirty page to disk first).

`FIFO`, `LRU`, and `OPT` need one command line argument to know the size of the simulated memory (the number of initially available frames). We also need this to determine the number of page faults.

Your task: Run the following commands and explain how memory size (the number of available frames) and page size affect the page fault count.

```
$ ./nameSorter 500 listaccess | ./pageRefGen 64 | ./LRU 64
$ ./nameSorter 500 listaccess | ./pageRefGen 32 | ./LRU 64
$ ./nameSorter 500 listaccess | ./pageRefGen 64 | ./LRU 32
$ ./nameSorter 500 listaccess | ./pageRefGen 32 | ./LRU 32
```

Your task: Run the following commands to compare the number of page faults using the three page replacement algorithms.

```
$ ./namesorter 500 listaccess | ./pageRefGen 64 | ./FIFO 64
$ ./namesorter 500 listaccess | ./pageRefGen 64 | ./LRU 64
$ ./namesorter 500 listaccess | ./pageRefGen 64 | ./OPT 64

$ ./namesorter 500 listaccess | ./pageRefGen 32 | ./FIFO 64
$ ./namesorter 500 listaccess | ./pageRefGen 32 | ./LRU 64
$ ./namesorter 500 listaccess | ./pageRefGen 32 | ./OPT 64

$ ./namesorter 500 listaccess | ./pageRefGen 64 | ./FIFO 32
$ ./namesorter 500 listaccess | ./pageRefGen 64 | ./LRU 32
$ ./namesorter 500 listaccess | ./pageRefGen 64 | ./OPT 32


```

```
$ ./namesorter 500 listaccess | ./pageRefGen 32 | ./FIFO 32
$ ./namesorter 500 listaccess | ./pageRefGen 32 | ./LRU 32
$ ./namesorter 500 listaccess | ./pageRefGen 32 | ./OPT 32
```

Collect the results from the above experiments (if needed, run more with other values) and try to interpret the result. Think of a way of displaying the result. Show your conclusion to the tutor before you leave the lab.