CMPUT 379 - Assignment #4 (8%)

A Demand Paging Virtual Memory Simulator

Due: Friday, December 8, 2017, 09:00 PM
(electronic submission)

Objectives

This programming assignment is intended to give you experience in developing a discrete event simulation program that enables the study of key performance measures of a conceptual demand paging virtual memory system.

The Simulator

In this assignment you are asked to design, implement, and test a C/C++ program, called `'a4vmsim'`, that simulates a number of *local* page replacement algorithms for managing a conceptual demand paging virtual memory system. Each run of the program reads from the standard input a synthetic reference string that models the behaviour of some hypothetical process and, upon termination, the program writes to the standard output a number of statistics about the performance of the system in processing the input reference string. More specifically, the simulator makes the following assumptions.

1. The length of each page in the system is a power of two between 256 bytes and 8192 bytes.

2. The CPU generates 32-bit addresses. Correspondingly, the reference string generator program generates a sequence of 32-bit **binary** words where each word encodes the page number used in a memory reference.

3. The CPU has a special accumulator register that is cleared to zero before the processing of any reference string.

4. Since a valid page length is at least 256 bytes, the least significant byte of each generated 32-bit word is not used in computing the referenced page number. The generator program utilizes such least significant byte in each generated word to encode information about the operation performed by the CPU in that particular memory reference. In particular, the generator utilizes the most significant two bits of each such byte to classify the operation performed by the CPU as follows:

   (a) $\boxed{00}\boxed{b_5b_4\cdots b_0}$: The operation increments the accumulator with the (unsigned) integer stored in bits $b_5b_4\cdots b_0$.

   (b) $\boxed{01}\boxed{b_5b_4\cdots b_0}$: The operation decrements the accumulator with the (unsigned) integer stored in bits $b_5b_4\cdots b_0$.

   (c) $\boxed{10}\boxed{*\,*\cdots *}$: The operation modifies the same page containing the reference word. The least significant 6 bits are not used.

(d) $\boxed{11}\boxed{*\cdots*}$: The operation reads data from the same page containing the reference word. The least significant 6 bits are not used.

Example. Assuming a 1024-byte page size, and a 32-bit reference word of $0x01220179$, the word is decoded into a page number, operation code, and operation value as follows:

$$\underbrace{0000\ 0001\ 0010\ 0010\ 0000\ 00}_{\text{page number}} \quad \underbrace{01}_{\text{unused bits}} \quad \underbrace{01}_{\text{operation}} \quad \underbrace{11\ 1001}_{\text{value}}.$$

Since the operation code is 01, the reference specifies an operation that decrements the accumulator by a decimal value of 57. ■

Simulator Inputs

In each run, the simulator reads an input reference string from the standard input. As mentioned above, the string is composed of a sequence of 32-bit **binary** words where each word encodes a referenced page number and information about the operation performed by the CPU in that particular reference.

> **Note.** Reading from the standard input allows the output of the generator program to be piped directly to the simulator to avoid storing a possibly huge reference string in a file.

The program should be callable using the following parameters:

> **a4vmsim** pagesize memsize strategy

where

- **pagesize**: is a power of two between 256 bytes and 8192 bytes, inclusively.
- **memsize**: is the size of the physical memory in bytes. Internally, the simulator should round up this value to the nearest multiple of page size. For example, if $pagesize = 512$ and $memsize = 1000$ then the system has two pages.
- **strategy:** is one of the following strings: `none`, `mrand`, `lru`, or `sec`. The corresponding strategies are as follows:
  - `none`: is a strategy that ignores the specified physical memory size, and assumes that the system has enough memory to simulate the reference string. (This strategy is used mainly to test basic functions of the program.)
  - `mrand`: is a modified version of RAND where the victim page is selected at random from all pages present in the physical memory, except the page(s) that have been referenced in the $k = 3$ references that occurred immediately before the fault. This strategy applies when the physical memory has more than $k = 3$ pages.
  - `lru`: is the LRU strategy.
  - `sec`: is the Second Chance strategy.

Simulator Outputs

After processing a reference string, the simulator prints the following statistics on the standard output:

1. The total number of memory references in the input string, and the number of write operations in the input string.
2. The total number of page faults.
3. The total number of flushes, i.e., cases when a victim page has been modified since the last time it was brought into memory.
4. The final value stored in the accumulator.

Reference String Generation

To generate synthetic reference strings, download and compile program `'mrefgen'` from the WEB page. The program can be invoked as follows:

$\%$ **mrefgen -m** $memsize$ **-l** $locality$ **-f** $focus$ **-w** wr **-n** $nref$

where

- **memsize**: specifies the maximum address to be generated. All addresses are in the range $[0, memsize - 1]$.
- **locality**: is a single digit $[0 - 9]$ specifying the degree of locality of the generated reference string.
- **focus**: is a single digit $[0 - 9]$ specifying the likelihood of referencing a small "permanently needed" memory area.
- **wr**: (the write ratio) is a floating point number in the range 0.0 to 1.0 specifying the fraction of write references in the generated string.
- **nref**: is the total number of references generated.

The program writes the generated reference string to its standard output file descriptor, and some statistics about the generated string to its standard error file descriptor. So, the reference string alone (without the statistics) can be piped directly to the simulator program or saved in a file.

> **Note.** You don't have to worry about the byte order (e.g., small or big endian) used to represent the generated 32-bit binary values since the generator and the simulator will be tested on the same machine.

```
Example: Running a script "run.sh" that has the following settings:
         #!/bin/sh
         page=256
         memsize=25600           # 100*page
         policy="lru"
         virtual=2560000         # 100*memsize
         nref=25600000           # 10*virtual address space
         mrefgen -m $virtual -l 3 -f 3 -w 0.4 -n $nref | a4vmsim $page $memsize $policy

         gives:

         a4vmsim [page=256, mem= 25600, lru]
         [mrefgen] 25600000 references generated, write count= 10239762
         [mrefgen] Accumulator final value= -72878
         [a4vmsim] 25600000 references processed using 'lru' in   36.25 sec.
         [a4vmsim] page faults= 15707259, write count= 10239762, flushes= 6943146
         [a4vmsim] Accumulator= -72878
```

Note: `mrefgen` generates a new pseudo random reference string each time it runs.

Running Time

The running time of your solution is not an issue as long as it can process reference strings of 20 million references in less than 2 minutes when executed on a lab machine with no other major programs running.

More Details

1. This is an individual assignment. Do not share code.
2. Although many details about this assignment are given in this description, there are many other design decisions that are left for you to make. In such cases, you should make reasonable design decisions that do not contradict the specifications and do not significantly change the purpose of the assignment. Document such design decisions in your source code, and discuss them in your report. Of course, you may ask questions about this assignment (e.g., in the Discussion Forum) and we may choose to provide more information or provide some clarification. However, the basic requirements of this assignment will not change.
3. When developing and testing your program, **make sure you clean up all processes before you logout of a workstation.** Marks will be deducted for processes left on workstations.

Deliverables

1. All programs should compile and run on the lab machines.
2. Make sure your programs compile and run in a fresh directory.
3. Your work (including a Makefile) should be combined into a single tar archive 'submit.tar'.

(a) Executing 'make' should produce the required executable file(s).

(b) Executing 'make clean' should remove all files produced by compilation, and all unused files.

(c) Executing 'make tar' should produce the 'submit.tar' archive.

(d) Your code should include suitable internal documentation of the key functions.

4. Typeset a project report (e.g., one to three pages either in HTML or PDF) with the following (minimal set of) sections:

   – **Objectives:** state the project objectives and value from your point of view (which may be different from the one mentioned above)

   – **Design Overview:** highlight in point-form the important features of your design

   – **Project Status:** describe the status of your project; mention difficulties encountered in the implementation

   – **Testing and Results:** comment on how you tested your implementation

   – **Acknowledgments:** acknowledge sources of assistance

5. Upload your tar archive using the **Assignment #4 submission/feedback** link on the course's web page. Late submission (through the above link) is available for 24 hours for a penalty of 10%.

6. It is strongly suggested that you **submit early and submit often**. Only your **last successful submission** will be used for grading.

Marking

Roughly speaking, the breakdown of marks is as follows:

17% : successful compilation of a complete program that is: modular, logically organized, easy to read and understand, and includes error checking after important function calls

03% : ease of managing the project using the makefile

70% : correctness of executing the program (tentatively, around 20% for 'none', 15% for 'mrand', 20% for 'lru', and 15% for 'sec')

10% : quality of the information provided in the project report