COMP SCI 3004/7064 - Operating Systems

Practical 2: Virtual Memory Management

Due by 11:59pm Wed 24th October

Critical Information

Submission

- Your implementation of the code is due 11:59pm on Wed 24th October 2018 (Week 12)
    - Your code will be submitted using SVN to the Web Submission System
    - The SVN directory for your code is 2018/s2/os/assignment2
    - Your code should be written in C.
    - It will be compiled using `gcc -std=c11 *.c -o memsim`
    - It will be run using `./memsim input_file.trace PAGESIZE NUMPAGES ALGORITHM INTERVAL` (see section *Running your code* below for more details)
    - For late code submissions the maximum mark you can obtain will be reduced by 25% per day (or part thereof) past the due date or any extension you are granted.
- Postgraduate students must complete and submit this assignment individually, making individual submissions.
- Undergraduate students may choose to complete and submit in teams of at most two students.

Marking scheme

This assignment is out of 20:

- 4 marks for Part 1 (second chance algorithm, automarked)
- 4 marks for Part 2 (enhanced second chance algorithm, automarked)
- 4 marks for Part 3 (additional reference bits algorithm, automarked)
- 4 marks for Part 4 (enhanced additional reference bits algorithm, automarked)
- 4 marks for Implementation quality (compilation, running, structure and comments in english; manually marked)

Task Description

Following our discussion of memory management in lectures, this practical allows you to explore the techniques an operating system uses to manage virtual memory. Your task is to implement a program that simulates the behaviour of a memory system that performs demand paging using 4 different page replacement strategies.

The Virtual Memory Simulator

You will need to write a simulator that reads a memory trace and simulates the action of a virtual memory system with a single level page table. The system needs to simulate swapping to disk and use a specified algorithm for page replacement.

- Your simulator should keep track of the status of the pages, including which page frames would be loaded in physical memory and which pages reside on disk.
    - The size of a page (in bytes) will be passed as the 2nd argument on the command line (See *Running your code* below for details).
    - The number of page frames that can be held in main memory will be passed as the 3rd argument on the command line (See *Running your code* below for details).
- As the simulator processes each memory event from the trace:
    - It should check to see if the corresponding page is in physical memory (hit), or whether it needs to be swapped in from disk (pagefault).
    - If a pagefault has occurred, your simulator may need to choose a page to remove from physical memory. When choosing a page to replace, your simulator should use the algorithm specified by the 4th argument passed on the command line (See *Running your code* below for details).
    - If the page to be replaced is dirty, it would need to be saved back to the swap disk. Your simulator should track when this occurs.
    - Finally, the new page is to be loaded into memory from disk, and the page table is updated.
- **Remember: This is just a simulation of the page table, so you do not actually need to read and write data from disk or memory. You just need to keep track of the events that would occur if this was a real system.**
- See textbook section 9.4 and the lecture slides for more details on page replacement.
- See Input/Output Details section below for details on input file structure and output formatting.

The page replacement strategies that you will be implementing are as follows:

Part 1 - Second Chance Algorithm

The least recently used (LRU) page replacement algorithm consistently provides near-optimal replacement, however implementing true LRU can be expensive. A simple way of approxuimating LRU is the Second Chance (SC) algorithm, which gives recently referenced pages a second chance before replacement.

Your task is to set up your simulator to use the Second Chance algorithm.

- Your simulator should use the Second Chance page replacement algorithm if the 4th argument passed on the command line is set to `SC` (See *Running your code* below for details).
- This algorithm is described in the Text Book in section 9.4.5.2
- A copy of this algorithm's description will also be available on MyUni (see assignment description).

Part 2 - Enhanced Second Chance Algorithm

The Second Chance algorithm is simple, but many systems will try to improve on this using an algorithm that takes into account the added delay of writing a modified page to disk during replacement. The Enhanced Second Chance (ESC) algorithm tracks whether a page has been modified or not and uses that information *as well as* whether the page was recently referenced to choose which page to replace.

Update your simulator to use the Enhanced Second Chance algorithm.

- Your simulator should use the Enhanced Second Chance page replacement algorithm if the 4th argument passed on the command line is set to `ESC` (See *Running your code* below for details).
- This algorithm is described in the Text Book in section 9.4.5.3
- A copy of this algorithm's description will also be available on MyUni.

Part 3 - Additional Reference Bits Algorithm

A closer approximation of LRU than the second chance algortihm is the Additional Reference Bits (ARB) algorithm, which uses multiple bits to keep track of page access history. These bits are stored in a shift register that regularly shifts right, removing the oldest bit.

Your task is to update your simulator to use the Additional Reference Bits algorithm.

- Your simulator should use the Additional Reference Bits page replacement algorithm if the 4th argument passed on the command line is set to `ARB` (See *Running your code* below for details).
- This algorithm is described in the Text Book in section 9.4.5.1
- Your implementation should use an 8-bit shift register, that shifts to the right.
- Bit shifting (right) should occur at a regular interval specified by the 5th argument passed on the command line (See *Running your code* below for details).
    - For example, if the interval provided is 5, then bit shifting should occur for all pages every 5th memory access.
- A copy of this description will also be available on MyUni.

Part 4 - Enhanced Additional Reference Bits Algorithm

Let’s combine the best aspects of ESC and ARB to create a new algorithm, Enhanced ARB (EARB). This algorithm will combine the improved access history of ARB with the awareness of modified pages from ESC.

Your task is to update your simulator to use the Enhanced ARB algorithm described below.

- Your simulator should use the Enhanced Additional Reference Bits page replacement algorithm if the 4th argument passed on the command line is set to `EARB` (See *Running your code* below for details).
- This algorithm works as follows:
    - If no pages are modified, or if only modified pages are resident, the algorithm should perform the same as ARB.
    - If both modified and unmodified pages are resident:
        - We want to avoid replacing the modified page unless it is several intervals older than the non modified page, so
        - A modified page should only be replaced if there does not exist a non-modified page that has been referenced within 3 intervals of the modified page.
        - e.g. given a modified page **`a`** with shift register values `00000100` and
            - a non modified page **`b`** with values `00001000`, replace **`b`**
            - a non modified page **`b`** with values `00010000`, replace **`b`**
            - a non modified page **`b`** with values `00100000`, replace **`b`**
            - In the above cases, `a` was last referenced 1, 2, and 3 intervals before `b` respectively, so `b` will be replaced
            - a non modified page **`b`** with values `01000000`, replace **`a`**
            - Here, `a` was last referenced more 3 than intervals (4 in this case) before `b`, so `a` will be replaced

Input/Output Details

Simulator Input - Memory Traces

We will provide you with a selection of memory traces to assist you developing your simulator. These will be a mix of specific test cases and real traces from running systems.
Each trace is a series of lines, containing two(2) values that represent memory accesses:

1. A character `R` or `W` that represents whether the memory access is a Read or Write respectively.
2. A hexadecimal memory address.

A trace may also contain blank lines, and lines that start with `#` which are comments. Your system should ignore both of these.

An example of a trace:

```
# This is a comment

R 0041f7a0
R 13f5e2c0
R 05e78900
R 004758a0
W 31348900
```

Running your code

Your code will be compiled using `gcc -std=c11 *.c -o memsim`, and will use all `.c` and `.h` files in your SVN folder.

The simulator should accept arguments as follows:

```
./memsim input_file.trace PAGESIZE NUMPAGES ALGORITHM INTERVAL
                1              2        3        4        5
```

1. The filename of the trace file
2. The page/frame size in bytes (we recommend you use 4096 bytes when testing).
3. The number of page frames in the simulated memory.
4. The algorithm used (one of `SC`, `ESC`, `ARB`, `EARB`).
5. The ARB shift interval (only used for ARB and EARB, not present otherwise)

The trace provided should be opened and read as a file, **not** parsed as text input from stdin.

For example, your code might be run like this:

```
    ./memsim test.trace 4096 32 EARB 4
```

Where:

- The program being run is `./memsim`,
- the name of the input file is `test.trace`,
- a page is `4096` bytes,
- there are `32` frames in physical memory,
- the Enhanced ARB algortihm (`EARB`) is used for page replacement, and
- the (E)ARB shift register shifts every 4 memory accesses.

Expected Output

The simulator should run silently with no output until the very end, at which point it prints out a summary like this:

```
events in trace:     1002050
total disk reads:    1751
total disk writes:   932
```

Where:

- **events in trace** is the number of memory access' in the trace. Should be equal to number of lines in the trace file that start with R or W. Blank line, or lines starting with # do not count.
- **total disk reads** is the number of times pages have to be read from disk (page fault).
- **total disk writes** is the number of times pages have to be written back to disk.

We will provide a set of expected outputs to match the given memory traces.