Assignment 3: Virtual Memory

- Due Mar 15 by 10pm
- Points 9
-

Introduction

In this assignment, you will simulate the operation of page tables and page replacement. This will give you some practice working with the algorithms we have been talking about in class.

You have three tasks in this assignment, which will be based on a virtual memory simulator. The first task is to implement virtual-to-physical address translation and demand paging using a two-level page table. The second task is to implement three different page replacement algorithms: FIFO, Clock, exact LRU. The third task is to create memory reference traces to run through your page replacement algorithms and carry out a small analysis.

Before you start work, you should complete the set of readings about memory, if you haven't done so already:

- Paging: Introduction (Links to an external site.)Links to an external site.

Requirements

Setup

Log into MarkUs to create or update your repo and get the starter code. Remember that you cannot manually create a new a3 directory in your repo or MarkUs won't see it. As usual, please make sure in advance that you can access your a3 directory, to avoid last-minute surprises. It is your responsibility to add the code in your repository and make sure that you submit all the necessary files!

Note that you may be generating some large trace files and must **not** commit any of the trace files that you generate to your repository or you will run into problems with disk quota. Most of the trace programs should be familiar to you from the online exercise, which you should complete first, to get used to the traces. We have added a blocked version of matrix multiply, blocked.c which should exhibit fewer page faults under at least some of the page replacement algorithms. The Makefile shows you exactly how to compile and run the traces. Note that it takes quite a while to run the trace collection.

Compile the trace programs and generate the traces.

You may have noticed while doing the Exercise that the traces generated by Valgrind are

enormous since they contain every memory reference from the entire execution. We have provided a program, fastslim.py to reduce the traces by removing repeated references to the same page that occur within a small window of each other while preserving the important characteristics for virtual memory simulation. (For example, a sequence of references to pages A and B such as "ABABABABAB...AB" are reduced to just "AB".) The runit script pipes the output of valgrind through this program to create the reduced trace. If you wish, you can experiment with fastslim.py to try omitting the instruction references from the trace or using a smaller or larger window (fastslim.py --help). You may also want to create traces from other programs, and you will definitely want to create small manual traces for testing.

The format of the traces will look like the following:

```
I 4000000
M 4226000
L fff000000
I 400b000
S 4227000
I 4002000
L 4225000
L 400000
I 4003000
L 4227000
L 400000
S 4226000
```

Task 1 - Address Translation and Paging

Implement virtual-to-physical address translation and demand paging using a two-level pagetable.

The main driver for the memory simulator, `sim.c`, reads memory reference traces in the format produced by the `fastslim.py` tool from `valgrind` memory traces. For each line in the trace, the program asks for the simulated physical address that corresponds to the given virtual address by calling find_physpage, and then reads from that location. If the access type is a write ("M" for modify or "S" for store), it will also write to the location. *You should read `sim.c` so that you understand how it works but you should not have to modify it..*

The simulator is executed as `./sim -f <tracefile> -m <memory size> -s <swapfile size> -a <replacement algorithm>` where memory size and swapfile size are the number of frames of simulated physical memory and the number of pages that can be stored in the swapfile, respectively. *The swapfile size should be as large as the number of unique virtual pages in the trace, which you should be able to determine easily.*

There are four main data structures that are used:

1. `char *physmem`: This is the space for our simulated physical memory. We define a simulated page size (and hence frame size) of SIMPAGESIZE and allocate SIMPAGESIZE * "memory size" bytes for physmem.

2. `struct frame *coremap`: The `coremap` array represents the state of (simulated) physical memory. Each element of the array represents a physical page frame. It records if the physical frame is in use and, if so, a pointer to the page table entry for the virtual page that is using it.
3. `pgdir_entry_t pgdir[PTRS_PER_PGDIR]`: We are using a two-level page table design; the top-level is referred to as the page directory, which is represented by this array. Each page directory entry (`pde_t`) holds a pointer to a second-level page table (which we refer to simply as page tables, for short). We use the low-order bit in this pointer to record whether the entry is valid or not. The page tables are arrays of page table entries (`pte_t`), which consist of a frame number if the page is in (simulated) physical memory and an offset into the swap file if the page has been written out to swap. The format of a page table entry is shown here:

   Note that the frame number and status bits share a word, with the low-order PAGE_SHIFT bits (12 in our implementation) used for status (we only have 4 status bits, but you can add more if you find it useful). *Thus, for a given physical frame number (e.g. 7), remember to shift it over to leave room for the status bits (e.g., `7 << PAGE_SHIFT`) when storing into the pte and to shift it back when retrieving a frame number from a pte (e.g., `p->frame >> PAGE_SHIFT`).*
4. `swap.c`: The swapfile functions are all implemented in this file, along with bitmap functions to track free and used space in the swap file, and to move virtual pages between the swapfile and (simulated) physical memory. *The `swap_pagein` and `swap_pageout` functions take a frame number and a swap offset as arguments. Be careful not to pass the frame field from a page table entry (pte_t) directly, since that would include the extra status bits*. The simulator code creates a temporary file in the current directory where it is executed to use as the swapfile, and removes this file as part of the cleanup when it completes. It does not, however, remove the temporary file if the simulator crashes or exits early due to a detected error. *You must manually remove the swapfile.XXXXXX files in this case*.

To complete this task, you will have to write code in `pagetable.c`. Read the code and comments in this file -- it should be clear where implementation work is needed and what it needs to do. The rand replacement algorithm is already implemented for you, so you can test your translation and paging functionality independently of implementing the replacement algorithms.

Task 2

Using the starter code, implement each of the three different page replacement algorithms: FIFO, exact LRU, CLOCK (with one ref-bit).

You will find that you want to add fields to the `struct frame` for the different page replacement algorithms. You can add them in `pagetable.h`, but please label them clearly. You may NOT modify the pgtbl_entry_t or pgdir_entry_t structures.

Task 3

Once you are done implementing the algorithms, run the `simpleloop` program and the `matmul` program from the provided `traceprogs`, using each of your algorithms (include rand as well). For each algorithm, run the programs on memory sizes 50 and 100. Use the data from these runs to create a set of tables that include the following columns. (Please label your columns in the following order,)

- Hit rate
- Hit count
- Miss count
- Overall eviction count
- Clean eviction count
- Dirty eviction count

Efficiency: Page replacement algorithms must be fast, since page replacement operations can be critical to performance. Consequently, you must implement these policies with efficiency in mind.
For example, we will give you the expected complexities for some of the policies:

- FIFO: init, evict, ref: O(1) in time and space
- LRU: evict, ref: O(1) in time and space; init: O(M) in time and space, where M = size of memory
- CLOCK: init, ref: O(1) in time and space; evict: O(M) in time, O(1) in space, where M = size of memory

Next, create by hand three different small memory traces of 30-50 page references to be run with a memory size of 8. The traces should have the following names and properties:

- trace1 - LRU, FIFO, and CLOCK algorithms each have a different hit rate and none of them are the optimal hit rate (trace the opt algorithm by hand)
- trace2 - at least one of the page replacement algorithms that you implemented will have the optimal hit rate
- trace3 - all of the page replacement algorithms have a hit rate of 0 even though each page is referenced at least 3 times

You will find this task much easier if you think about different patterns of referencing the pages and draw some pictures. You will also want to keep the number of unique pages fairly small. All of the evictions could be clean evictions.

Write up

Include a file called README.pdf that includes the following information.

- The tables prepared in Task 3
- One paragraph comparing the various algorithms in terms of the results you see in the tables.
- A table showing the hits and misses of your three traces on LRU, FIFO, and CLOCK.

Remember that you are expected to do your own work and that submitting someone else's work in part or in whole is plagiarism and an academic offence. It would be better to leave out parts of

the assignment rather than copy code you find on the internet or from another student.

Marking Scheme

- Task 1: 40%
- Task 2:
  - FIFO 6%
  - LRU 10%
  - CLOCK 10%
  - (must be able to run all traces in a reasonable amount of time)
- Task 3:
  - Tables 10%
  - Small traces that you create 10%
  - paragraph 5%
- Program readability and organization 10%
- **Negative deductions (please be careful about these!):**
  - Code does not compile -100% for *any* mistake, for example: missing source file necessary for building your code (including Makefile, provided source files, etc.), typos, any compilation error, etc. If you receive 0 for this reason, please submit a remarking request explaining how to fix your program so that it will compile and then file a remark request. There will be a penalty of -20%
  - Warnings: -10%
  - Extra output (other than what sim.c produces): -10%
  - Code placed in subdirectories: -20% (only place your code directly under your a3 directory)

Submission

The assignment must be pushed to the `a3` directory in your git repository (again, please do not create this directory manually, MarkUs should create that for you). Don't forget to push your updated simulator code, Makefile and your `README.pdf` (in text or pdf format). We will retrieve the last revision before the deadline for marking.

If you are not able to fully complete the assignment or you have made some design decisions that you think need more explanation, please include an INFO.txt file that contains this type of information.

Make sure that you do not leave any other printf messages, other than what sim.c is printing. This will affect marking, so if you don't follow this requirement, you will be deducted 10% for leaving extra output in your final submission.
Make sure your code compiles without any errors or warnings.
Code that does not compile will receive zero marks!

As previously, to check that your assignment submission is complete, please do the following:

1. create an empty temporary directory in your cdf account (not in a subdirectory of your

repo)
2. check out a copy of your repository for this assignment
3. verify that **all the required** files are included (double-check the submission instructions above)
4. run make and ensure that you are able to build sim without any errors or warnings (This is an excellent way to verify that the right source files have been committed to the repo.)
5. run a few tests using the same traces you used to create the tables in your README.pdf, to ensure that your code behaves as you expect
6. make sure to clean up any unnecessary files (executables, traces, etc.), and make sure your files are directly under the a3 directory (no subdirectories!). Remember that you do need to submit your 3 hand-created traces.
7. congratulate yourself and enjoy a well-earned break, knowing that your strategy and hard work will pay off!