Assignment 3: Virtual Memory

Due Dec 9 by 10pm **Points** 12

Introduction

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

Part 1: Memory reference traces

There are three different programs that can be found in `/u/csc369h/fall/pub/a3` on the teach.cs machines:

- `simpleloop.c` - loops over an array allocated in the heap (You can also modify the code to run the same loop using stack memory)
- `matmul.c` - naive matrix multiply with the ability to change the element size to change the memory access behaviour
- `blocked.c` - a more memory aware matrix multiply that should exhibit a better hit rate under some page replacement algorithms

Valgrind has an option that will allow you to print out the memory reference trace of a running program, and you can see how we use it by looking at the shell script `runit`.

There are two aspects of the output of valgrind that are unsatisfactory for our purposes. First, it includes the entire program memory trace including loading the program and libraries into memory which makes is harder to focus on the access patterns of the algorithm. To address this problem, we have added special variables to the programs whose only purpose is to serve as a marker, so that we will only keep the reference trace between the two markers. The python program `trimtrace.py` carries out this task. These address level traces for the three programs are stored in `/u/csc369h/fall/pub/a3/traces/addr-*.ref`. **These are the trace files that you will analyze in Part 1.**

This still produces a massive trace because it displays every memory reference, even if they are to the same page. The `fastslim.py` program reduces the size of the traces and focus on page accesses. **You will use these traces to run your page replacement algorithms in Part 2.** These traces can be found in `/u/csc369h/fall/pub/a3/traces/page-*.ref`.

You can find all traces in compressed archive format at `/u/csc369h/fall/pub/a3-traces.tgz`. Use scp to download them to your machine. The file can probably be unpacked by a program on your system by clicking on it, or you can use `tar xf a3-traces.tgz`.

The directory includes all of the code to build and generate the traces, so that you can try it out on different programs if you like. If you want to do this, you will need to copy the files to your account. Remember not to commit trace files to your repo because two of them are quite large.

Your task is to read the programs to get a picture of how they are using memory and write a short program in the language of your choice to analyze the trace output. Then compare your counts to the program to match the access pattern to variables and operations in the program.

The resulting address trace files have the following format, where the first character indicates whether the access is an Instruction, Store, Load, or Modify, the second value is the address in hex, and the third value is the size of the access:

```
S 04228b70,8
I  04001f4f,4
I  04001f53,3
L 04227f88,8
I  04001f99,7
L 04228a50,8
I  04001fa0,3
I  04001fa3,2
I  04001fa5,4
M 04227e80,8
I  04001fa9,7
L 04228a48,8
```

Your analysis program will translate each memory reference into a page number assuming pages are 4096 bytes. It will output a table of each unique instruction pages with a count of the number of accesses for each page, and a table of unique data pages with a count of the number of accesses for each page.

For example, the above snippet will produce the following:

```
Counts:
  Instructions 7
  Loads        3
  Stores       1
  Modifies     1

Instructions:
0x4001000,7
Data:
0x4228000,3
0x4227000,2
```

What to submit for Part 1:

- The program you wrote to produce the table with instructions for how to run it on the teach.cs machines.
- A text file called `analysis.txt` containing the following information about each of the provided traces:

  - The number of instruction pages (I).
  - The number of data pages (S, L, M).
- In `analysis.txt`, identify which pages are accessed the most and explain briefly which variable(s) or code from the program might be stored in these pages.

Note: The intent is that you should be able to do this exercise in about an hour or two. You do not need to write more than a few sentences.

Part 2: Virtual to physical translation

The remainder of the assignment is based on a virtual memory simulator that uses the memory reference traces from Part 1. The next task is to implement virtual-to-physical address translation and demand paging using a two-level page table. Then you will implement three different page replacement algorithms: FIFO, Clock, exact LRU. Finally, you will create some small memory reference traces and use the reference traces from Part 1 to analyze the replacement algorithms.

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

- **Paging: Introduction (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf)**

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.

The traces from Part 1 will be interesting to run once you have some confidence that your program is working, but you will definitely want to create small traces by hand for testing.

The format of the traces will look like the following. Note that the addresses are all the 0th byte of a page (the lower 12 bits (3 hex digits) are always 0).

```
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

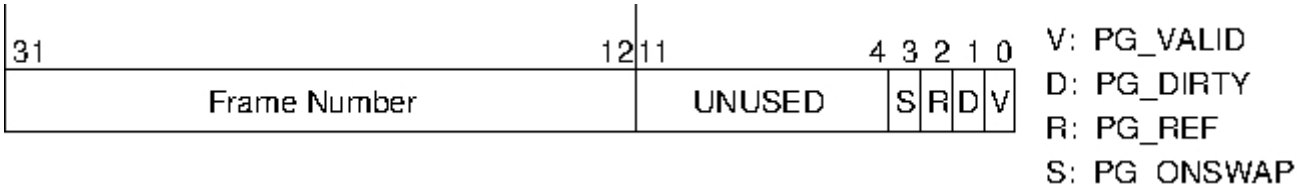
Implement virtual-to-physical address translation and demand paging using a two-level pagetable.

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, but you *should* take advantage of features already provided in the page table entries.

Note: to test your page replacement algorithms, we will replace your `pagetable.c` with a solution version, so you page replacement algorithm must be contained to the provided functions.

Task 3

Once you are done implementing the algorithms, use the provided traces from Part 1 to check the results. 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. The tables must be organized such that the trends in data can be easily seen. 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 file 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) or have a hit rate of 0
- `trace2` - at least one of the page replacement algorithms that you implemented will have the optimal hit rate, and none of the algorithms have a hit rate of 0
- `trace3` - LRU, FIFO and CLOCK of the page replacement algorithms have a hit rate of 0 even though each page is referenced at least 3 times and the OPT algorithm has a hit rate higher than 0

The point of this exercise is to generate traces that differentiate the algorithms and show some interesting behavior. You will not get full marks for the traces if they trivially satisfy the properties. Unless specified otherwise, each trace should have evictions and hits for each algorithm. (This is not a trick question.)

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. The number of clean or dirty evictions doesn't matter.

Important notes

When we run the autotests on your code, your page replacement algorithms will be compiled with a different `pagetable.c` file (the one from the solution). All the code of the page replacement algorithms must be in their separate `.c` files, not in `pagetable.c` (except for additions to `struct frame` in `pagetable.h`).

The 2 PTE bits "valid" and "on swap" describe 3 possible states: (i) the page is in memory; (ii) the page is not in memory and is stored in the swap file; and (iii) the page is not in memory and not in the swap file. The value of the swap bit is not important for a valid page. However, to keep things well-defined, you should keep the "on swap" bit set after the page has been written to swap once (i.e. do not clear it when the page is brought back into memory from the swap).

When a page is being evicted, there should be only 2 possibilities: (i) the page is dirty and needs to be written to the swap; and (ii) the page is clean and already has a copy in the swap. To avoid the situation where an evicted clean page doesn't have a copy in the swap, a newly initialized page should be marked dirty on the very first access.

CLOCK must use the "ref" bit stored in the PTE. All the algorithms must utilize their `ref()` functions (if necessary) instead of adding any algorithm-specific code to `pagetable.c`.

The simulator and the page replacement algorithms must not produce any additional or different (from starter code) output (except for errors that should be printed to `stderr`), otherwise the tests will fail.

The trace files submitted in Task 3 must have the exact file names specified in the description, and must be located in the main directory of the assignment (alongside the source code), otherwise the autotester won't be able to find them.

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 offense. 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

- Part 1: 10%
- Task 1: Page tables 35%
- Task 2: Page replacement algorithms
  - FIFO 5%
  - LRU 10%
  - CLOCK 10%
  - (must be able to run all traces in a reasonable amount of time)
- Task 3: Analysis
  - Tables 5%
  - 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%
  - Compiler warnings: -10%
  - Extra output (other than what `sim.c` produces): -10%

Submission

The assignment must be pushed to the `a3` directory in your git repository. Submission checklist:

- analysis.txt (Part 1)
- your analysis program (Part 1)
- Updated code and Makefile (Part 2)
- hand-created trace files (Part 2)
- README.pdf (or README.txt if you would prefer to submit a text file) (Part 2)

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.

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 on completing the final CSC369 assignment!