Homework #5 – Caching and Virtual Memory

Due date: see course website

Directions:

- For short-answer questions, submit your answers in PDF format to the GradeScope assignment "Homework 5 written".
- For programming questions, submit your source file using the filename specified in the question to the GradeScope assignment "Homework 5 code".
- **You must do all work individually, and you must submit your work electronically via GradeScope.**
    - All submitted code will be tested for suspicious similarities to other code, and the test will uncover cheating, even if it is "hidden" (by reordering code, by renaming variables, etc.).

Q1. Cache policies

[5 points] Why are write-back caches usually also write-allocate? *Hint: see "Cache Interaction Policies with Main Memory" linked on the course site.*

Q2. Cache performance

[5] Your L1 data cache has an access latency of 1ns, and your L2 cache has an access latency of 10ns. Assume that 90% of your L1 accesses are hits, and assume that 100% of your L2 accesses are hits. What is the average memory latency as seen by the processor core?

Q3. Cache layout

[20] You have a 64-bit machine and you bought 4GB of physical memory. Pages are 64KB.

(a) [1] How many virtual pages do you have per process?
(b) [1] How many physical pages do you have?
(c) [2] In the translation from a virtual address to a physical address, how many bits of VPN are you mapping to how many bits of PPN (assuming you have just enough bits in the physical address for the amount of physical RAM present)?
(d) [1] How big does a page table entry (PTE) need to be to hold just a single PPN?
(e) [1] How big would a flat page table be for a single process, assuming PTEs are the size computed in part (d)?
(f) [10] Why does the answer above suggest that a “flat page table” isn’t going to work for a 64-bit system like this? Research the concept of a *multi-level page table*, and briefly define it here. Why could such a data structure be much smaller than a flat page table?
(g) [4] Does a TLB miss always lead to a page fault? Why or why not?

Q4. Virtual memory address translator program

[30] In C, write a program called `virt2phys` which translates virtual addresses to physical addresses by means of a page table loaded from a file, per the specifications below.

Calling syntax

The program will be called virt2phys, and will have the following calling syntax:

```
./virt2phys <page-table-file> <virtual-address>
```

Arguments:

- `<page-table-file>`: File containing page table
- `<virtual-address>`: Virtual address to translate (in <u>hex</u>).

Page table file format

The format of the page table file is a whitespace-delimited sequence of numbers (hint: this is an indicator that you can get by with nothing but `fscanf()`). The format is as follows:

```
<address-bits> <page-size>

<ppn>                        (physical page number for virtual page 0; set to -1 if invalid)
<ppn>                        (physical page number for virtual page 1; set to -1 if invalid)
  ...
<ppn>                        (physical page number for virtual page N-1,
                              where N is the number of virtual pages; set to -1 if invalid)
```

All values above are decimal integer numbers. The fields above are:

- `<address-bits>`: The word size of the system, in <u>bits</u>. **Will be ≤ 24.**
- `<page-size>`: Page size of the virtual memory system, in <u>bytes</u>.
- `<ppn>`: A physical page number in the page table. Set to -1 for invalid. **The number of ppn values will be equal to the number of virtual pages, which can be computed from the above two fields.**

Note: while all provided input files will have this structure, your input can simply be a sequence of `fscanf()` calls to consume integers; there is no reason to care about line separation or to use `fgets()`.

Files fed to your program will always meet the following rules:

- Files will always be of valid format and fully specify the page table content
- The number of address bits will be at most 24.
- The configuration will always make sense (e.g., page size less than the size of memory, page size a power of 2, etc.)
- Numbers will always be appropriate, e.g. no ppn will never be larger than the maximum possible ppn

Program output

The program simply outputs the given virtual address's corresponding physical address, in <u>hex</u>, followed by a newline. If the page table indicates that the given address has no valid physical page (i.e., `ppn==-1`), the program will simply print "PAGEFAULT\n".

Below is an example showing a very small 10-bit system (1kB memory space) with 256-byte pages, so 4 physical pages total. The page table shown has no mapping for virtual page 1. Sample runs of the first and last byte of each virtual page are shown.

```
/bin/xhere /bin/bash.exe C:\Users\tkbletsc\Dropbox\Duke\ECE250\Hom...
$ cat example_page_table.txt
10 256

2
-1
3
0

$ ./virt2phys example_page_table.txt 0
200
$ ./virt2phys example_page_table.txt ff
2ff
$ ./virt2phys example_page_table.txt 100
PAGEFAULT
$ ./virt2phys example_page_table.txt 1ff
PAGEFAULT
$ ./virt2phys example_page_table.txt 200
300
$ ./virt2phys example_page_table.txt 2ff
3ff
$ ./virt2phys example_page_table.txt 300
0
$ ./virt2phys example_page_table.txt 3ff
ff
$ 
```

Restriction

In this assignment, **you may NOT use the modulus (%) operator**. You must use bitwise operations to determine the components of the address. Penalty: 50% off overall score.

Additionally, your program should exit with a status of 0 (EXIT_SUCCESS). Penalty: 25% of score.

You do NOT have to worry about memory leaks on this assignment. However, valgrind is still a useful tool for detecting misuse of memory, and may help you find hidden bugs!

Building and testing

You can simply build your program as per usual with g++:

```
g++ -g -o virt2phys virt2phys.c
```

A number of input files are in the `pagetables` subdirectory; these are described by `pagetables/INFO.txt`. As with prior assignments, a suite of tests is provided in the `tests` subdirectory and an automated testing tool, `tester.py`, has been provided to automate testing.

Q5. Cache simulator program

[70] In C, write a simulator of a single-level cache and the memory underneath it.

The simulator, called `cachesim`, takes the following input parameters on the command line: name of the file holding the loads and stores, cache size (not including tags or valid bits) in kB, associativity, and the block size in bytes. The replacement policy is always LRU. For example, "`cachesim tracefile 1024 4 32`" should simulate a cache that is 1024kB (=1MB), 4-way set-associative, has 32-byte blocks, and uses LRU replacement. This cache will be processing the loads and stores in the file called tracefile.

Important Assumptions: Addresses are 24-bits (3 bytes), and thus addresses range from 0 to 2^{24}-1 (i.e., there is 16MB of address space). The machine is byte-addressed and big-endian. The cache size, associativity, block size, and access size will all be powers of 2. Cache size will be no larger than 2MB, block size will be no larger than 64B, and no access will be larger than the block size. No cache access will span multiple blocks (i.e., each cache access fits within a single block).

All cache blocks are initially invalid. All cache misses are satisfied by the main memory (and you must track the values written through to memory in case they are subsequently loaded). If a block has never been written before, then its value in main memory is zero. The cache is **write-through** and **write-no-allocate**. This means your program will need to store both the state of cache and the entire content of simulated memory; the memory part can be represented as a simple array of 16M bytes.

If you have any known bugs, please include those in a README file to help the grader give partial credit.

Calling syntax

The program will be called `cachesim`, and will have the following calling syntax:

```
./cachesim <trace-file> <cache-size-kB> <associativity> <block-size>
```

Arguments:

- `<trace-file>`: Filename of the memory access trace file.
- `<cache-size-kB>`: Total capacity of the cache, kilobytes (kB). A power of two between 1 and 2048.
- `<associativity>`: The set associativity of the cache, AKA the number of ways. A power of two.
- `<block-size>`: The size of the cache blocks, in bytes. A power of two between 2 and 512.

All numeric arguments are in decimal format.

Trace file format

The trace file will be in the following format. There will be some number of lines. Each line will specify a single load or store, the 24-bit address that is being accessed (in base-16), the size of the access in bytes, and the value to be written if the access is a store (in base-16). For example:

```
store 0xd53170 4 7d2f13ac
load  0xd53172 1
store 0xd53170 2 f0b1
store 0x1a25bb 2 c77a
load  0xd53170 4
load  0x12 2
store 0x23 8 d687eb9f1bc687ec
```

As can be seen, leading 0 bits will not be in addresses in the trace file. Also, as viewed in the store commands, values following the access size in bytes will be the correct size. Accesses will be no larger than 8 bytes at a time. Because all parts of the file are whitespace-delimited tokens, fscanf will be your friend.

Program output

Your simulator must produce the following output. For every access, it must print out what kind of access it is (load or store), what address it's accessing (in base-16), and whether it is a hit or a miss. For each load, it must print out the value that is loaded (possibly after satisfying the miss from memory). The output format must be as follows and may not be graded if format is ignored. Here is output for the example input file shown above with a 1MB 4-way cache with 32-byte blocks:

```
$ ./cachesim traces/example.txt 1024 4 32
store 0xd53170 miss
load 0xd53172 miss 13
store 0xd53170 hit
store 0x1a25bb miss
load 0xd53170 hit f0b113ac
load 0x12 miss 0000
store 0x23 miss
```

The "0x" before each address is required; leading 0s are not printed.

Always print out the exact number of hex digits corresponding with the number of bytes missed. Memory defaults to all zeroes on boot.

Restriction

In this assignment, **you may NOT use the modulus (%) operator**. You must use bitwise operations to decompose the address. Penalty: 50% off overall score.

Additionally, your program should exit with a status of 0 (EXIT_SUCCESS). Penalty: 25% of score.

You do NOT have to worry about memory leaks on this assignment. However, valgrind is still a useful tool for detecting misuse of memory, and may help you find hidden bugs!

Building and testing

You can simply build your program as per usual with g++:

```
g++ -g -o cachesim cachesim.c
```

A number of input files are in the `traces` subdirectory; these are described by `traces/INFO.txt`. As with prior assignments, a suite of tests is provided in the `tests` subdirectory and an automated testing tool, `tester.py`, has been provided to automate testing.

Tips relevant to Q4/Q5

- To manipulate bit fields, use bitwise operators (`&`, `|`, `~`), shifts (`<<`, `>>`) and masks.
- You can get a bit string of N ones with the expression: `((1<<N)-1)`
- Here's a simple implementation of base-2 log using only integer math, that way you don't have to mess with the math library:

```
int log2(int n) {
   int r=0;
   while (n>>=1) r++;
   return r;
}
```

- Parsing tips for `cachesim`:
  - You can `fscanf` two hex digits at a time using the "%2hhx" specifier.
  - You can provide known parts of the input format in the `fscanf` format, such as the "0x" part of the hex address for `cachesim`.
  - See the `scanf` manpage for more information.