CSCI 340 - Operating Systems I

Assignment 1 Total Points 100

Objectives

In this assignment you will develop a simple *cache simulator* using the C programming language. This assignment will allow you to gain experience in the following areas:

- **Basic Programming**: This includes variable declaration, data types, arrays, pointers, operators, expressions, selection statements, looping statements, functions, structs, and header files.
- **Cache mapping functions**: Become familiar with two different cache mapping functions: 1) Direct Mapping and 2) Fully associative (see section 13.4.1 in the PDF named CacheAndRAM in the content section on OAKS). Specific examples where reviewed in class (refer to class notes) and this is a opportunity to code them.
- **Physical Memory**: Become familiar with physical memory, dividing memory into blocks, and then associating blocks of memory with caching operations.

System and Standard Lib Functions

This assignment will use the system and standard library functions listed below. Please ensure your familiar with the syntax, and usage of them. Detailed information about each function listed below can be found using the `man` command from the console: i.e.

- **Memory Allocation**: `void* malloc(size_t size)`
- **Basic IO**: `printf()` and `fscanf()` and `scanf()`

Provided Files

The three files listed below are provided to you.

- **cache.h**: Header file that defines the cache *struct* used in this assignment, constants, and the function prototypes to be completed by you. **Please note**: You may not add new function definitions to this header file.
- **cache.c**: The file containing the implementation of the functions listed in *cache.h*. Having a different file for the implementation separates interface (the include file) from the implementation (the .c file).
- **memory.h**: Header file that defines the physical memory *array* used in this assignment, constants, and the function prototypes to be completed by you. **Please note**: You may not add new function definitions to this header file.
- **memory.c**: The file containing the implementation of the functions listed in *memory.h*. Having a different file for the implementation separates interface (the include file) from the implementation (the .c file).
- **hw1.c**: Source code file that includes the main function, and defines the libraries and constants used in this assignment. **Please note**: You may not remove, modify, or add (i.e. #include) additional libraries. The ones that are provided, are the only libraries needed for this assignment.

- **memory.txt**: Text file that simulates physical memory with random byte values at each address location. The simulated memory has 8-bits addressable space, i.e. the total number of addresses is 2^8

The *hw1.c*, *memory.h* and *cache.c* files contain many comments that provide basic definitions and step-by-step instructions. Please read the comments carefully.

Todo

Using the *memory.h* file, you must provide working implementations within the corresponding *memory.c* file for the following function prototypes:

- *int numberOfBlocks(unsigned int addr_bits, unsigned int num_block_offset_bits)*

Using the *cache.h* file, you must provide working implementations within the corresponding *cache.c* file for the following function prototypes:

- *int cread(unsigned int cmf, unsigned int* hex_addr, unsigned int* found, unsigned int* replace)*

For each function prototype listed above, numerous comments are provided in the header file to guide you in this assignment. Please read them carefully, they either provide basic step-by-step instructions, or basic calculations.

In the *hw1.c* file, the main function has been fully implemented for you. **Please note**: You may not change the code in the main method.

Collaboration and Plagiarism

This is an **individual assignment**, i.e. **no collaboration is permitted**. Plagiarism will not be tolerated. Submitted solutions that are very similar (determined by the instructor) will be given a grade of zero. Please do your own work, and everything will be OK.

Submission

Create a compressed tarball, i.e. *tar.gz*, that only contains the completed *cache.c* and *memory.c* files. The name of the compressed tarball must be your last name. For example, *ritchie.tar.gz* would be correct if the original co-developer of UNIX (Dennis Ritchie) submitted the assignment. Only assignments submitted in the correct format will be accepted (no exceptions). Submit the compressed tarball (via OAKS) to the Dropbox setup for this assignment. You may resubmit the compressed tarball as many times as you like, Dropbox will only keep the newest submission.

To be fair to everyone, late assignments will not be accepted. Exceptions will only be made for extenuating circumstances, i.e. death in the family, health related problems, etc. You will be given a week to complete this assignment. Poor time management is not excuse. Please do not email assignment after the due date, it will not be accepted. Please feel free to setup an appointment to discuss the assigned coding problem. I will be more than happy to listen to your approach and make suggestions. However, I cannot tell you how to code the solution. Additionally, code debugging is your job. You may use the debugger (gdb) or print statements to help understand why your solution is not working correctly, your choice.

Grading Rubric

For this assignment the grading rubric is provided in the table shown below.

Program compiles	5 points
Program runs with no errors	5 points
numberOfBlocks() function implementation	5 points
cread() function implementation	5 points
Eight test cases (10 points each) chosen by instructor	80 points

In particular, the assignment will be graded as follows, if the submitted solution

- does not compile: 0 of 100 points
- compiles but does not run: 5 of 100 points
- compiles and runs without errors: 10 of 100 points
- all functions correctly implemented: 20 of 100 points
- all test cases work correctly: 100 of 100 points

Simulator Guidance

Bit structure

In this assignment, you may assume the following for direct mapping:

- **tag bits**: are bits 7,6,and 5
- **line bits**: are bits 4,3, and 2
- **block offset bits**: are bits 1 and 0

To refresh your knowledge, please refer to the bit structure Fig. 13-20.

In this assignment, you may assume the following for fully associative:

- **tag bits**: are bits 7,6,5,4,3,2
- **block offset bits**: are bits 1 and 0

To refresh your knowledge, please refer to the bit structure Fig. 13-21.

Cache Structure

The cache is defined in the cache.h header file. In general, cache is a one-dimensional array and each element in the array is a cache_line struct. You may assume:

- **lines**: Each element in the array represents a line, e.g. index 0 corresponds to line 0 in the cache simulator.
- **tag**: The tag field in the cache_line struct stores the value of the tag bits.
- **hit**: The hit_count field in the cache_line struct stores the number of cache hits for current tag value.

Please note: For the simulator program, the block is not copied to the cache line. To simulate the cache read you can use the following formula:

$$byte = physical\_memory[\ block\_location + block\_offset\ ]$$

Cache Replacement Algorithm

Please use the lest frequently used (LFU) algorithm to determine which line to replace in cache. To determine the LFU line, you must use the hit_count field defined in the cache_line struct. If two or more cache lines have the same LFU value, then select the cache line at or near line zero to replace, e.g. if lines 4,2,5 all have the same LFU value, then line 2 would be selected for replacement.

Memory and Blocks

The memory is defined in the memory.h header file. In general, two a one-dimensional arrays are defined:

- **phy_memory**: Each element in the array stores one byte, and each element represents an addressable location, e.g. index 0 corresponds to address 0 in physical memory.
- **block_location**: Each element in the array represents a block, and each element stores a starting address location in physical memory, e.g. index 0 corresponds to block 0, and the value stored at the index location is the starting address location.