Computer Architecture
Assignment 3

This homework will be split up into three parts. As always it is important you adhere to the following guidelines:

(1) Work individually;

(2) The submission date is March 7, 23:59;

(3) Submission is via the "submit" system;

(4) Ensure that your submission compiles and runs without errors or warnings on BIU's servers before submitting. Failure to do so will result in docked points;

(5) At the beginning of every file you submit, add your Teudat Zehut and name in a comment. For example: `/* 123456789 Israel Israeli */`;

(6) It is forbidden to use AI tools like ChatGPT when writing your homework. Doing so is tantamount to cheating, and will be treated as such;

1 Simulating the Cache

In this part of the assignment, you are asked to write code which simulates how a cache operates. To do so, you will need to define the two following structs, which will be utilized to represent cache sets and the cache itself respectively:

```c
typedef unsigned char uchar;

typedef struct cache_line_s {
    uchar valid;
    uchar frequency;
    long int tag;
    uchar* block;
} cache_line_t;

typedef struct cache_s {
    uchar s;
    uchar t;
    uchar b;
    uchar E;
    cache_line_t** cache;
} cache_t;
```

Then you will need to define the following three functions:

```c
cache_t* initialize_cache(uchar s, uchar t, uchar b, uchar E);
uchar read_byte(cache_t cache, uchar* start, long int off);
void write_byte(cache_t cache, uchar* start, long int off, uchar new);
```

The type `cache_t` contains in it four parameters: s, t, b, E which are defined as in the book: $S = 2^s$ is the number of sets, $B = 2^b$ is the number of blocks per line in the set, E is the number of lines per set, and t is the tag length. Recall that given an address of length $m = s + t + b$, partition the bits into

t bits	s bits	b bits

The `cache` field is an array of array of cache lines. Each array of cache lines can be thought of as a cache set, and so `cache` is simply an array of sets. Each `cache_line_t` has a valid bit (ignore the fact that it is of course actually a byte), its frequency, its tag (which should be t bits long, but we made it a long to make your (my) life easier), and the block of memory.

The cache you implement should replace lines using the LFU (least frequently used) method, meaning if it tries to read in memory to a set with no available lines then it replaces the line whose frequency is minimal. If two lines both have the same minimal frequency, then the first line is chosen (ie. if line 3 and line 10 both have frequency 1 which is the smallest, then line 3 is chosen).

The `read_byte` and `write_byte` functions accept as inputs `start` as well as `off`. You should act as if `start` is the zero address and `off` is the address that you insert into your cache. More specifically, insert the contents of `start[off]` into your cache, while using the bits of `off` as your offset.

Your cache is write-through, meaning that when `write_byte` is called, it writes `new` both to the cache and to memory.

We also provide you with the `print_cache` function, so you don't need to worry about whitespace errors But fair warning: it's going to paste weirdly into your text editor, sorry. You win some, you lose some.

```c
void print_cache(cache_t cache) {
   for (int i = 0; i < 2 << (cache.s-1); i++) {
      printf("Set %d\n", i);
      for (int j = 0; j < cache.E; j++) {
         printf("%1d %d 0x%0*lx ", cache.cache[i][j].valid,
            cache.cache[i][j].frequency, cache.t, cache.cache[i][j].tag);
         for (int k = 0; k < 2 << (cache.b-1); k++)
            printf("%02x ", cache.cache[i][j].block[k]);
         puts("");
      }
   }
}
```

So for example,

```c
int main() {
   uchar arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
   cache_t cache = initialize_cache(1, 1, 1, 2);
   read_byte(cache, arr, 0);
   read_byte(cache, arr, 1);
   read_byte(cache, arr, 2);
   read_byte(cache, arr, 6);
   read_byte(cache, arr, 7);
   print_cache(cache);
}
```

Should print

```
Set 0
1 2 0x0 01 02
0 0 0x0 00 00
Set 1
1 1 0x0 03 04
1 2 0x1 07 08
```

For this section you should create a directory called `cache/` in which you add all your code, plus a makefile which compiles to an executable called `cache/cache`.

2 Fast Matrix Multiplication

A popular problem in computer science is the problem of fast matrix multiplication (FMM). Currently the fastest algorithms can multiply two square matrices of size n in $O(n^\omega)$ time for $\omega > 2$. It is conjectured that ω is strictly larger than 2, meaning there does not exist an algorithm which multiplies two matrices in $O(n^2)$ time.

Note that the naive algorithm

```c
int* smm(int* m1, int* m2, int* result, int n) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         result[i * n + j] = 0;
         for (int k = 0; k < n; k++)
            result[i * n + j] += m1[i * n + k] * m2[k * n + j];
      }
   }

   return result;
}
```

runs in $O(n^3)$ time. There exist algorithms which speed up multiplication to $\omega = 2.3728596$ as per Wikipedia, Feburary 2024.

Your goal for this exercise is to write a program to multiply two matrices together as fast as possible. In the `FMM/` directory you will find the following files: `main.c`, `utilities.c`, `fmm.h`, and `fmm.c`. Let us take a look at the `fmm.h` file:

```c
int* create_matrix(char* file_name, int n);
int* read_matrix(char* file_name, int n);
void free_matrix(int* mat, int n);
double measure_time(char* file1, char* file2, char* file_result, int n);

void fmm(int n, int* m1, int* m2, int* result);
```

The way we will represent matrices in this exercise are as `int*` — integer pointers. So an $n \times n$ matrix A will be represented by an integer pointer p to an area of memory of size n^2, where $a_{ij} = p[i \cdot n + j]$ as seen in the above example. These matrices will also be saved in files, which is why the parameters to the above functions are file paths.

(1) `create_matrix(char* file_name, int n)` returns a matrix of size $n \times n$. Changes to this matrix (ie. changes to the area of memory pointed to by the pointer) will be saved to the file specified by `file_name`.

(2) `read_matrix(char* file_name, int n)` loads a matrix of size $n \times n$ from the file specified by `file_name` and returns a pointer to it.

(3) `free_matrix(int* mat, int n)` frees the memory associated with the matrix pointed to by `mat` of size `n`.

(4) `measure_time` is the primary function, it is called by `main`. It loads the matrices from `file1` and `file2`, multiplies them according to `fmm` and saves the result to `file_result`.

Your assignment is to rewrite the `fmm` function to be as efficient as possible. You are given the size of the matrices (`n`), pointers to the matrices (`m1` and `m2`), and a pointer to the result matrix (`result`). You must write all your computations to `result`, meaning at the end you should have for all $0 \le i, j < n$:

$$\texttt{result}[i \cdot n + j] = \sum_{k=0}^{n-1} \texttt{m1}[i \cdot n + k] \cdot \texttt{m2}[k \cdot n + j]$$

In `fmm.c` you are given a naive implementation of `fmm` which you can optimize any way you'd like. Currently on my machine it takes 17 seconds to multiply two matrices of $n = 1000$. Do better than this.

To test your code, you can create tests as follows: you have been given a python script `create-matrices.py`. Use it to create randomly generated matrices of varying sizes (eg. create two matrices with $n = 1000$). Then use the current implementation of `fmm` (which can serve as a baseline; you know it works) to produce the results.

Then after you optimize `fmm`, rerun it with the generated matrices and compare your new results to your old results using bash commands like `diff` and similar (as always `man diff` for more information).

Your grade will be determined based off of how quick your `fmm` runs (on BIU servers) relative to your peers. Do not unionize and all submit bad code: code which runs at subpar speed will receive a low grade even if it is the quickest code submitted.

The best of luck and may the best person win (get 100)!

3 Patching

For this section, you are given a binary `patchwork`, which you must somehow patch in order to get it to print the secret flag. Use any reverse engineering tools you'd like, and write down the steps you took to solve this problem in a file called `cheating_is_bad` (which can be a plain txt, docx, or pdf file).

If you are unsuccessful but have a good direction for your solution, write your ideas down in `cheating_is_bad`, and you may receive partial credit.

I recommend using the tool `Cutter` for reverse engineering and patching. In case you cannot run the file `patchwork` locally, upload it to BIU's servers and run it there.

4 What to Submit

You must submit using the `submit` system a zip file containing the following:

(**1**) For the first section on simulating the cache, a directory by the name `cache/` containing all your code plus a makefile to compile it.

(**2**) For the second section on FMM, the directory `FMM/` containing all (and only) the supplied files and a makefile to compile them.

(**3**) For the third section on patching, the patched binary file `patchwork`.