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 July 17, 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:

```
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:

```
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) {
   int S = 1 << cache.s;
   int B = 1 << cache.b;

   for (int i = 0; i < S; 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 < B; 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
```

We supply you here your main function. It simply gets as input the size of your data (which simulates RAM), the actual data to store in “RAM”, the cache parameters, and then the bytes it should read (which it reads until it gets a negative value):

```c
int main() {
    int n;
    printf("Size of data: ");
    scanf("%d", &n);
    uchar* mem = malloc(n);
    printf("Input data >> ");
    for (int i = 0; i < n; i++)
        scanf("%hhd", mem + i);

    int s, t, b, E;
    printf("s t b E: ");
    scanf("%d %d %d %d", &s, &t, &b, &E);
    cache_t cache = initialize_cache(s, t, b, E);

    while (1) {
        scanf("%d", &n);
        if (n < 0) break;
        read_byte(cache, mem, n);
    }

    puts("");
    print_cache(cache);

    free(mem);
}
```

So for example, the previous example can be equivalently run like so:

```
Size of data: 8
Input data >> 1 2 3 4 5 6 7 8
s t b E: 1 1 1 2
0 1 2 6 7 -1

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

2 Optimization

A popular problem in computer science is the problem of finding a minimal Vertex Cover. This problem is known to be NP-Hard, meaning we don't know how to solve this problem in polynomial time (and we don't know if it is possible at all). In this problem, we are given a graph $G = (V, E)$, presented as an adjacency matrix $A \in \{0, 1\}^{|V| \times |V|}$, which satisfies $A_{ij} = 1 \Leftrightarrow (i, j) \in E$. Your goal is to find the minimal number of vertices needed to "touch" every edge in the graph.

You are given an unoptimized implementation of this code in a file named "vertex-cover.c" which implements 3 functions:

(**1**) `isVertexCover` check if some optional cover is really a set-cover.

(**2**) `generateCombinations` recusive code that generates optional combinations of covers.

(**3**) `findMinVertexCover` the main called function, needs to print the minimum set cover.

You are also given a "main.c" that asks for an input and runs your code.

Your assignment is to rewrite those 3 functions in order to optimize it, you can not change the main file, only the "vertex-cover.c" file.

You can assume that the input for the problem (which is an adjacency matrix of size n), is valid.

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 adjacency matrices of varying sizes, which you can check on.

It is important that your code will return **the same** result as the original code, and that you compile with the **"-O0"** flag, which means that the compiler can not add optimizations!

Then after you optimize the code, 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 code 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 to an executable named `cache`.

(**2**) For the second section on Optimization, the directory `optimizations/` containing all (and only) the supplied files and a makefile to compile them to an executable named `cover`.

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