CSCI 2800 — Computer Architecture and Operating Systems (CAOS)
Homework 4 (document version 1.2)
Multi-Threaded Programming and Network Programming

- This homework is due in Submitty by 10:00PM Thursday, December 11, 2025
- You can use at most **two late days** on this assignment, which means your final submission **must** be in by 9:59PM Saturday, December 13
- This homework is to be done individually, so **please do not share your code with others**
- Place all of your code in `hw4.c` for submission; no other files may be included
- You **must** use C for this assignment, and all submitted code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors
- All submitted code **must** successfully compile and run on Submitty, which currently uses Ubuntu v22.04.5 LTS and `gcc` version 11.4.0 (`Ubuntu 11.4.0-1ubuntu1~22.04`)
- This assignment will be entirely auto-graded by Submitty; you will have **eight** penalty-free submissions, after which points will slowly be deducted, e.g., `-1` on submission #9, etc.
- You will have at least **three** days before the due date to start submitting your code to Submitty; if auto-grading is not available three days before the due date, the due date will be 10:00PM three days after auto-grading becomes available

Hints and reminders

Do **not** rely on program output to confirm whether your code is correct. Consistently allocate **exactly** the number of bytes you need regardless of whether you use static or dynamic memory allocation. Further, deallocate dynamically allocated memory via `free()` at the earliest possible point in your code. Be careful to deallocate multi-layer structures from the deepest level outward.

Make use of `valgrind` (or `drmemory` or other dynamic memory checkers) to check for errors with dynamic memory allocation and dynamic memory usage. Also, close open file descriptors as soon as you are done using them.

Always read (and re-read!) the `man` pages for library functions (section 3), system calls (section 2), etc. To understand how `man` pages are organized, check out the `man` page for `man` itself (via `man man`).

Strategy

This homework incorporates both multi-threaded programming and network programming. Therefore, as a strategy, first focus entirely on the multi-threaded programming. Then, move on to the network programming portion in which you make your multi-threaded program available on the network.

Homework specifications

In this fourth and final homework, you will use C to implement a multi-threaded UDP server that "encodes" or "decodes" messages that it receives.

There are two parts to this assignment. First, you will focus on implementing a multi-threaded program to encode/decode messages, somewhat similar to Checkpoint 2 of Lab 10. These threads must all run in parallel.

Second, once this first part is completed, you will shift your focus to making your encode/decode program available on a network as a UDP server. Specifically, clients can send encode and decode requests to your server. Submitty will automatically send a variety of UDP datagrams to your server and verify you send back the correct response messages.

Command-line arguments

There is one required command-line argument, which is only applicable to the network programming portion of this assignment. This command-line argument specifies the UDP port number your server receives UDP datagrams on.

Validate the input to confirm it is a valid port number; if not, display the following to `stderr` and return `EXIT_FAILURE`:

```
ERROR: Invalid argument
USAGE: ./hw4.out <UDP-server-port>
```

No square brackets allowed!

To emphasize the use of pointers and pointer arithmetic, one last time, **you are not allowed to use square brackets** anywhere in your code! If a `'['` or `']'` character is detected in your code, **including within comments,** Submitty will completely remove that line of code before running `gcc`. Remember to use `grep` (or the `find` feature in your IDE) to detect any extraneous square brackets.

Dynamic memory allocation

As with previous homeworks, you must use `calloc()` to dynamically allocate memory. You may also use `realloc()` if you would like.

Of course, you must use `free()` for all such allocations and have no memory leaks.

Do **not** use `malloc()` or `memset()`.

Encoding and decoding messages

Similar to Checkpoint 2 of Lab 10, your code must process text by going character by character, potentially replacing individual characters with one or more characters. Your "encoding" must follow the algorithm below.

1. Replace each newline `'\n'` character with string `"%%%%"`.
2. Replace each space character with string `'.'`.
3. Replace each tab `'\t'` character with string `"..|.."`.
4. Replace each digit character with an expanded triplicate string, i.e., `"0#0"`, `"1#1"`, `"2#2"`, etc.
5. Replace each alpha character, regardless of case, with the alpha character exactly three steps forward, looping around to `'A'` or `'a'` as necessary, i.e., `'A'` maps to `'D'`, `'B'` maps to `'E'`, `'C'` maps to `'F'`, etc., `'W'` maps to `'Z'`, `'X'` maps to `'A'`, `'Y'` maps to `'B'`, etc. (Apply the same approach shown here to lowercase letters, too.)

You must also implement the "decoding" algorithm, which is the opposite of the above. Note that there is no guarantee that decoding will result in exact same original message. **(v1.2)** Always process the data in the UDP datagram from left to right, i.e., from the first byte to the last byte.

Multi-threaded requirements

Your `main` thread is responsible for assigning messages to child threads for text processing. Once you have the network programming portion complete (see the next page), messages will be sent to your UDP server (and handled by your `main` thread).

Ignoring the network programming requirements for now, your `main` thread should either read in messages from a series of files or have multiple messages hard-coded for testing. This testing should include both encoding and decoding of the given messages.

The `main` thread creates a child thread to process a message. Here, the child thread is responsible for either encoding or decoding the given message. While there are multiple ways to implement this, you should create two thread functions called `encode_message()` and `decode_message()`. The `main` thread will determine which function to use and specify the function by name in the `pthread_create()` call.

Remember that `pthread_create()` allows you to send a generic pointer (`void *`) to the thread function. This pointer should point to dynamically allocated memory that contains the message to be encoded or decoded.

Application-layer protocol

The specifications below focus on the application-layer protocol that your server must implement to successfully communicate with multiple UDP clients simultaneously.

Your `main` thread is responsible for receiving each UDP datagram via `recvfrom()`. For each message received, the `main` thread calls `pthread_create()` to handle the UDP datagram and send a response. **(v1.2)** Use a maximum UDP datagram size of 128 bytes; in other words, datagrams of length greater than 128 bytes should be truncated to 128 bytes.

Since UDP is connection-less, when a UDP datagram is received, the `main` thread extracts the client's host address and port number (via `sin_addr` and `sin_port`, as shown in the `udp-server.c` example). This information must be handed off to the child thread; therefore, consider creating a `struct` to hold this information, as well as the message to be encoded or decoded.

After processing the request, the child thread sends the UDP datagram response to the same address and port, which may change for each UDP datagram your server receives.

The first byte of the UDP datagram received specifies whether the client would like to encode or decode the rest of the message. Specifically, an `'E'` indicates a request to encode, whereas a `'D'` indicates a request to decode.

The child thread must construct a response message as follows:

- The first byte echoes back the first byte of the request, indicating either a request to encode (`'E'`) or to decode (`'D'`).
- The remaining bytes are the encoded or decoded message.

The protocol for an encode request is shown below. In general, the encoded message will be longer than the original message.

```
                   +---+-----------------------+
  CLIENT REQUEST:  | E | <bytes-to-encode> ... |
                   +---+-----------------------+

                   +---+-----------------------------------------+
 SERVER RESPONSE:  | E | <encoded-message> ...                   |
                   +---+-----------------------------------------+
```

Next, the protocol for a decode request is shown below. In general, the decoded message will be shorter than the original message.

```
                   +---+-----------------------------------------+
  CLIENT REQUEST:  | D | <bytes-to-decode> ...                   |
                   +---+-----------------------------------------+

                   +---+-----------------------+
 SERVER RESPONSE:  | D | <decoded-message> ... |
                   +---+-----------------------+
```

Signals and server termination

Since servers are typically designed to run "forever" without interruption, ignore signals `SIGINT`, `SIGTERM`, and `SIGUSR2`.

Still, we need a mechanism to shut down the server. **(v1.2)** Set up a signal handler for `SIGUSR1` that ~~gracefully~~ shuts down your server by ~~shutting down any running child threads (if necessary), freeing up dynamically allocated memory, closing any open descriptors, and~~ exiting the process with `EXIT_SUCCESS`.

(v1.2) Use the signal-related code in `udp-server-v2.c` for your homework implementation; specifically, your signal handler can simply call `exit()` to terminate the process (and therefore immediately and abruptly terminate all of the threads). This also means you can just use `pthread_detach()`.

Required output

For output, your program must display a line of output for each UDP datagram received and sent, also a summary of the encoding or decoding process.

As an example, consider the original message shown below, which includes a tab `'\t'` character and ends in a newline `'\n'` character.

```
ABCDefgh.IJKL?mnop&QRST(uvwx)YZ 123\t890.-\n
                                   ^^     ^^
                       ...these are '\t' and '\n' characters
```

The example program execution below shows the output format you must follow.

```
bash$ ./hw4.out 8888
MAIN: UDP server is ready to receive datagrams...
THREAD: Received encode request (42 bytes)
THREAD: Sent encoded response (62 bytes)
MAIN: SIGUSR1 received; shutting down the server...
bash$
```

The encoded response for the above request should be as follows. (**(v1.1)** Corrected a typo below.)

```
DEFGhijk.LMNO?pqrs&TUVW(xyza)BC.1#12#23#3..|..8#89#90#0.-%%%%%
```

(v1.1) If the above encoded response is then sent to your server to be decoded, the decoded message should be as follows.

```
ABCDefgh IJKL?mnop&QRST(uvwx)YZ 123\t890 -\n
                                   ^^     ^^
                 ...these are again '\t' and '\n' characters
```

Error handling

In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then either exit the thread or exit the program (i.e., **all** threads) with `EXIT_FAILURE`. Only use `perror()` if the given library or system call sets the global `errno` variable.

Error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

Submission instructions

To submit your assignment and perform **final** testing of your code, we will use the auto-grading features of Submitty.

Submitty will use the following to compile your code:

```
bash$ gcc -Wall -Werror -o hw4-server.out hw4.c
```

To help ensure that your program executes properly, consider making use of the `DEBUG_MODE` preprocessor technique shown below. This helps to avoid accidentally displaying extraneous output in Submitty. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of q is %d\n", q );
    printf( "here123\n" );
    printf( "why is my program crashing here?!\n" );
#endif
```

To compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -g -o hw4-server.out -D DEBUG_MODE hw4.c
```