Ancient Book Curses

While you pay a library fine for not returning a library book on time these days, medieval curses were often added to books by medieval librarians. The book curse was intended to discourage the theft of manuscripts. Here is a colorful curse[1]:

> *If anyone taketh away this book, let him die the death; let him be fried in a pan; let the falling sickness and fever seize him; let him be broken on the wheel, and hanged. Amen.*

Indeed, book curses were not limited to medieval europe. Mesoapotamian tablets were also known to have curses.

> *Whosoever shall carry off this tablet, or shall inscribe his name on it, side by side with mine own, may Ashur and Belit overthrow him in wrath and anger, and may they destroy his name and posterity in the land.*

This assignment exercises writing the server side of a networking program. The server program serves ancient book curses from a curse database, back to the client that requests a curse.

---

[1] Ancient swearing and curses were often religious oaths. Fashions in swearing have since changed, and went from religious ones to excrement and secretions to sacrament oaths (swearing by baptism) to the four letter ones which we are now more familiar with.

Imprecate[2]

You're given the client. The client program is written in Python in **imprecate.py** and is called "imprecate". You invoke it on the command line with

```
python3 test/imprecate.py hostname port num_curses
```

Imprecate connects to the given hostname at the given port, and requests num_curses from the server. Imprecate prints out each curse on a separate line and exits with a code of 0. For example, calling **python3 test/imprecate.py localhost 4444 2** will connect to the server running on the same computer that is listening on port 4444, and it could print two curses like this:

```
May the sword of anathema slay if anyone steals this book away.
Let him be struck with palsy & all his members blasted.
```

Imprecate doesn't have a lot of error checking and embeds constants in the code. You probably don't want to do that in your own code.

Maledict[3]

Your job is to write the server. The server program is written in C in **maledict.c** and is called "maledict". You invoke it on the command line with

```
./maledict [-s random_seed]
```

Maledict reads a database of curses stored in the file **curses.txt**. The optional `-s` argument is used to pass a seed to maledict to initialize the random number generator with. Without the seed argument, the random seed used is 1. Maledict opens four TCP ports at `CURSES_PORT`*x*, where *x* is 0, 1, 2 or 3, and listens for a client connection on each of the ports.

When a client connection is made, maledict responds to the client request and returns the appropriate number of randomly chosen curses to the user.

Maledict is the world's least reusable server: it runs once, then quits and returns with an exit code of 0 if successful. If wrong arguments were given, it exits with a code of 1. If the database cannot be read, the program exits with code 2. If any networking error occurs, it exits with a code of 3. If the request was in error, it exits with an error code of 4.

---

[2] imprecate. [im-pri-keyt] Verb (used with object), im·pre·cat·ed, im·pre·cat·ing. to invoke or call down (evil or curses), as upon a person.
[3] maledict. [mal-i-dikt]Archaic. Adjective: accursed. Verb (used with object): to put a curse on.

Random Curse Generation

A random number generator is used to choose which curse to return to the user.
To use `random(3)`, we initialize it with a seed value. If not specified, the seed value is 1.

```
unsigned int random_seed = ..;       // some value
srandom(random_seed);                // initialize with a seed
```

Then if there are a total of `num_curses`, we can generate a random curse number between 0 and `num_curses - 1` by calling

```
int curse_number = random() % num_curses;
```

Normally we would use the random(3) function from the standard library, but the implementation can differ across platforms (it's different on MacOS versus Linux versus Windows, for example). So we will use our own simple random number generator provided for you in **`simplerand.c`**. This file provides **our own implementation of `srandom(3)` and `random(3)`**. Since we provided it, the compiler and linker will use ours instead of pulling the two functions in from the standard libraries. Note that the test programs rely on this, so don't use any other method, or forget to initialize the seed properly.

Curse numbering starts from `0`. The last curse is `num_curses - 1`.

Cursed Text Transfer Protocol

The Cursed Text Transfer Protocol is used to define communication between the client and the server. Due to its cursed nature, it's a combination of binary and textual data.

Operations

The set of operations allowed is defined in this enumeration.

```
// An enumeration of the different client operations
typedef enum {
    // PING the server
    PING = 0,

    // Get a single curse.
    GET_SINGLE,

    // Get a random number of n curses.
    GET_MULTI,
} Operation;
```

If the operation is PING, there is no accompanying argument, and the server returns OK with no curses attached. If the operation is GET_SINGLE, there is no accompanying argument. The server is expected to return a single, randomly chosen curse. If the operation is GET_MULTI, then the client must provide an integer *n* specifying the number of curses to get. The server is expected to return *n* randomly chosen curses.

Request

The client sends a request to the server by making a connection to the server at the appropriate port number. It then sends a `RequestHeader` to the server. The definition is:

```
typedef struct req {
    int32_t protocol_id;
    Operation op;
    int16_t numargs;
} RequestHeader;
```

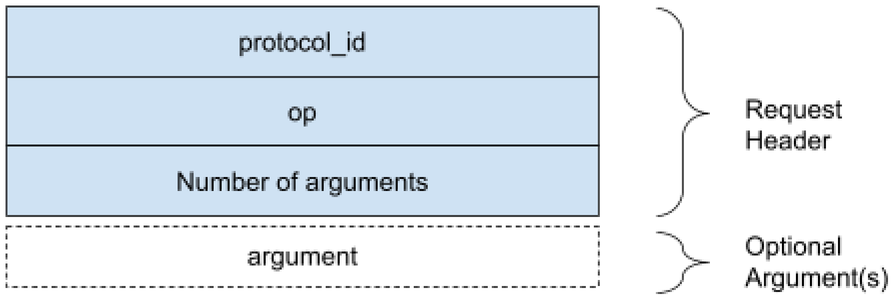
The RequestHeader holds a protocol_id. This number is defined as

```
// The protocol id for this version of the Curses Transfer Protocol
#define CURSES_PROTOCOL_ID     13
```

We use an unlucky 13 as this version of the protocol.

The op field holds the operation, which is one of either PING, GET_SINGLE, or GET_MULTI.

The `numargs` field holds the number of arguments we are passing. This field is always either 0 for GET_SINGLE, or 1 for GET_MULTI. If there is an argument, then the arguments are passed as a series of consecutive `int32_t` values directly after the header. You might ask why the protocol is defined this way when the only two operations have either 0 or 1 argument. If we were to extend the number of operations and some require more arguments, the protocol won't have to change very much.



Response

The maledict server can reply with one of several status values, as defined in this enum.

```
// An enumeration of the different server response statuses
typedef enum {
    // Success
    OK = 0,

    // Bad protocol
    INVALID_PROTOCOL,

    // The request was malformed somehow
    BAD_REQUEST,

    // There were not enough arguments
    INSUFFICIENT_ARGUMENTS,
} Status;
```

The server sends back a `ResponseHeader`, defined in this struct:

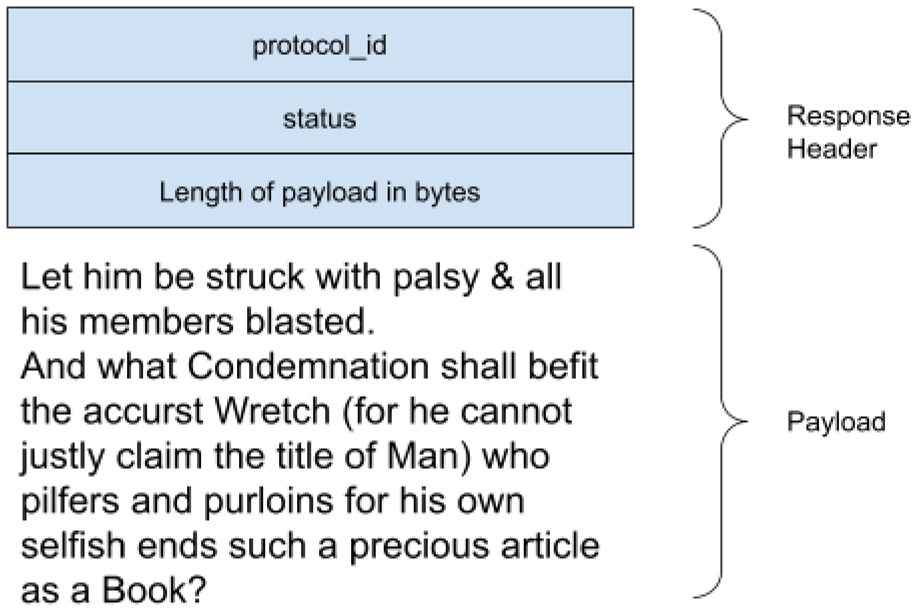
```
typedef struct res {
    int32_t protocol_id;
    Status status;
    int32_t length;
} ResponseHeader;
```

The `protocol_id` field should be set to `CURSES_PROTOCOL_ID.`

The `status` field should be set to one of the `Status` enum values.

The `length` field is used to tell the client how many bytes of data will follow the header. For all status codes except OK, the length field should be 0.

The actual data payload consists of a string of *n* curses, separated by newlines. The total number of bytes should match the value of `length` in the header above.


protocol_id
status
Length of payload in bytes
Response Header
Let him be struck with palsy & all his members blasted. And what Condemnation shall befit the accurst Wretch (for he cannot justly claim the title of Man) who pilfers and purloins for his own selfish ends such a precious article as a Book?
Payload

Network Byte Ordering

Note that all values in `RequestHeader` and `ResponseHeader` as well as arguments are passed over the network in **network byte order**. You must convert to network byte order when sending, and then convert from network byte order to host byte order when you receive data.

Listening on Multiple Sockets

The maledict server opens four separate sockets and listens on each one. You accomplish this by using the select(2) system call. Learn how to use this call – don't fork multiple children.

A traditional server like a web server that stays up and serves clients over and over doesn't exit after one connection like maledict does. It usually wakes up on select(2), accepts the connection from the client, and then forks a child to deal with that connection. The new child deals with the request. The child exits when it is done serving the request. Meanwhile, the parent server program goes right back to waiting on select(2) for new connections.

Our maledict server serves just one request and then exits.

Reusing Ports

One thing you will find when developing a networking program is that you often have your program crash, or you need to hit Control-C to kill the server or when a program hangs. This can often result in a port that is not released. The system can hang onto a port to make sure that any data in transit or was buffered is properly released, or when some timeout occurs.

To get around that when developing, it's useful to have this in your server:

```
// Avoid bind error for port already in use
int tr = 1;
if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &tr, sizeof(int)) == -1) {
    perror("setsockopt");
    exit(2);
}
```

The `setsockopt` call manipulates socket options. `SO_REUSEADDR` says to reuse a local address so that frequent binds will not cause an issue.

Memory Leaks

Your client and server should have no memory leaks at the end of using it. You can use `valgrind` (or `leaks` on MacOS) to confirm that it is not leaking memory.

Building and Testing

There's a Makefile provided for you. **Build** the server with

```
make maledict
```

Functional tests are available with

```
make test-functional
```

Some of the functional tests look for output from `imprecate.py` to check that what it returns is the same as what's expected. You can see the expected output in the `test/output-*` files that match the Makefile's `test-*` targets.

For example, this is a target that checks for GET_SINGLE

```
test-single: $(SERVER)
      @echo "Testing GET_SINGLE..."
      @python3 test/runserverclient.py "./maledict > /dev/null"
            "python3 test/imprecate.py localhost 4444 1" > test/out
      @echo "Comparing outputs..."
      @cmp test/out test/output-single
```

```
        @echo "Success!"
```

The test forks a child server for maledict while sending the output of the server to /dev/null (/dev/null is emptiness; all output to that is ignored). This is done so that the maledict server's output does not confuse the output of imprecate.py. Then imprecate.py is run to connect to the server at port 4444. The output of imprecate is redirected to test/out. Finally test/out is compared with test/output-single. If the outputs match, you pass the tests. Please don't change the test/output-* files!

You can test for **memory leaks** with

```
    make test-memory
```

You can test for **style** with

```
    make test-style
```