Multithreaded webserver

Invoking the program

```
./make ~/ 8000 4 6
```

Output showing the 4 worker threads and one main thread:

`ps -eLf | less`:

```
UID PID PPID LWP C NLWP STIME TTY TIME CMD

sigbbe 35444 35430 35444 0 5 07:59 pts/1 00:00:00 ./mtwwwd /home/sigbbe/ 8000 4 6
sigbbe 35444 35430 35445 0 5 07:59 pts/1 00:00:00 ./mtwwwd /home/sigbbe/ 8000 4 6
sigbbe 35444 35430 35446 0 5 07:59 pts/1 00:00:00 ./mtwwwd /home/sigbbe/ 8000 4 6
sigbbe 35444 35430 35447 0 5 07:59 pts/1 00:00:00 ./mtwwwd /home/sigbbe/ 8000 4 6
sigbbe 35444 35430 35448 0 5 07:59 pts/1 00:00:00 ./mtwwwd /home/sigbbe/ 8000 4 6
```

a) Implementation of single-threaded server

We have a method for setting up the server, one for binding the sockets and one for When we refactored the code to make it multithreaded, we created n-threads by calling the `pthread_create` syscall and passing in the `handle_req` method. before this the main thread handled the reading and responding of requests and responses.

b) Counting semaphores

The counting semaphore struct is defined by:

```
struct SEM {
    int count;
    pthread_cond_t cond;
    pthread_mutex_t mutex;
}
```

The Semaphore, SEM, structure has three attribute members. (1) the `int count` fields to keep track of the state, if the count is less than or equal to 0, the P (wait) method will block until the V (release) method is called. (2) the `pthread_cond_t cond` field is used for calling the `pthread_cond_wait` method. (3) the `pthread_mutex_t mutex` is the lock struct for locking and unlocking access to the resource protected by the semaphore.

c) Ring buffer

The ring buffer struct is defined by:

```
struct BNDBUF {
    int size;
    int in;
    int out;
    int count;
    int *buffer;
    struct SEM *sem_empty;
    struct SEM *sem_full;
}
```

The ring buffer contains attributes for specifying the size of the buffer, keeping track of wich element that should be returned at next access (call to the `int bb_get(BNDBUF *bb)` method) and keeping track of which index to insert the next element (at next call to the `void bb_add(BNDBUF *bb, int fd)` method). The ring buffer does not delete elements in the buffer, only overwrites old elements by new values. The ring buffer also includes two semaphore, one for blocking on calls to the *insert* method when the buffer is full, and one for blocking the *get* method when the buffer is empty.

d) Threading implementation

As specified in the task, the server runs by one one producer (the main thread) and *n* number of threads (consumers). The main thread waits for incomming connections, creates new file descriptors and inserts the file descriptors into the ring buffer. The worker threads are wating for connections by blocking on the ring buffer's getter method which returns valid connection file descriptors when the ring buffer isn't empty.

e) Security update

Directory traversal exploit

As the server is hosted locally it was not allowed to use "../" commands in the url. We had to use telnet to send request to the server to exploit the Directory traversal. By sending requests this way you could climb up the directory tree, allowing arbitrary file access in the server file system.

Implentations to prevent directory traversal

We implemented validation for the web root server path. Web root server path is required to be the start of the requested url, or else you will not have access.

Another validation we implemented was to check the public read permission on requested files. If this was allowed you will recieve access, or else you are denied. We only check for public read permission were we could implement owner and group permissions as well, this was not necessary in this exercise.