Part 1
Sleeping time feature is implemented with the help of the function `poisson_random_interarrival_delay` provided in the specification.

Bad client feature is implemented in the simplest way: random variable is calculated on every client with `rand() % 101` (so that the values are from 0 to 100). Then it is checked to be less than or equal to the provided number of bad clients - `bad`. If it is true, then this client is bad, and `sleep` function is called.
It was tested by calculating manually the print statements indicating the bad clients given a small number of producers and consumers (10-20).

The values of new command-line arguments are checked at the beginning of `main` function with if-else statements according to the specified boundaries.

Part 3
All three files contain the same chunks of code related to the streaming of bytes with the checking of some edge cases:

- if the size of the item is less than the `BUFSIZE`, sent all letters at once
- else go in a loop and send chunks of `BUFSIZE` bytes until you reach the end, which leads to another check - if the size divides by `BUFSIZE` evenly, just stop, if not - send the last remaining bytes.

In the beginning, I gathered all the repeating code into functions but it caused strange errors related to allocating memory and sharing of pointers which I was not able to fix. So all the code is inside the original functions.
The code works with `MAX_LETTERS` equal to 1 billion and `BUFSIZE` equal to 2048/4096, 500 producers, and 490 consumers.

Part 4
The array with arrival times of clients is maintained throughout the server lifetime. Initially, all the values are set to -1; when the new socket is accepted the arrival time is recorded; when the client is served the value is changed back to -1 again.
I check the time client is active in the same loop where the socket descriptors are checked with `FD_ISSET` function, so slow clients are removed instead of (uselessly) checking them. Also, if `select` returned 0, it does not make sense to check all the socket descriptors to be ready to read, so I added `continue` to the `for` loop as well. Attaching picture for more understanding:

```
for ( fd = 0; fd < nfds; fd++ ) {
        if (select_val == 0) continue;

        time( &raw_time );
        if ( fd != msock && arrival_times[fd] != -1 && raw_time - arrival_times[fd] > REJECT_TIME ) {

                printf( "Client too long has not identified itself. Rejecting.\n" );
                fflush( stdout );
                pthread_mutex_lock( &mutex );
                clients_not_identified++;
                clients--;
                pthread_mutex_unlock( &mutex );
                arrival_times[fd] = -1;
                close( fd );
                FD_CLR( fd, &afds );
                if ( fd+1 == nfds ) nfds--;

        } else if (fd != msock && FD_ISSET(fd, &rfds)) {
                if(( cc = read( fd, buffer, 9 ) )) {
                        buffer[cc] = '\0';
                        arrival_times[fd] = -1;

                        if ( strcmp( buffer, "PRODUCE\r\n" ) == 0 ) {
                                pthread_mutex_lock( &mutex );
                                can_accept = producers < MAX_PROD;
                                if ( can_accept ) producers++;
                                pthread_mutex_unlock( &mutex );

                                if( can_accept ) {
                                        descriptors[fd] = fd;
                                        pthread_create( &thread, NULL, handle_producer, (void *) descriptors[fd]);
                                } else {
                                        printf( "Server: rejected the producer.\n" );
                                        fflush( stdout );

                                        pthread_mutex_lock( &mutex );
                                        clients--;
                                        prod_rejected++;
                                        pthread_mutex_unlock( &mutex );
                                        close( fd );
```

I have not come up with better balancing between not checking every client every time to be slow and not allow slow clients to hang around for a long time. I think this implementation is not the worst though in terms of performance since the code enters the `for` loop above anyways and removing the slow clients before checking the `fd_set` may be an optimization.

Part 5

The new function `handle_status_client` was added where the buffer is checked for matching the known status commands, the corresponding value is sent to the client if the command is matched, and the socket is closed.

Steady state

I tried different rates (1, 10, 50, 100) and bufsizes (2K, 4K, 10K) and was not able to identify the steady state. The server runs quite the same every time with serving 600 producers and consumers but sometimes there are some clients (around 1-8) that remain hanging somewhere and not terminating at the end. For example, running `check.sh` with the server running gave:
600
586
8
0

And running `check.sh` with terminated server gave this:
600
586
14
0
Which means that 6 clients were hanging. I think this is because of bug(s) that I have not identified and fixed.
Also, I don't know anything about the capacity of Ubuntu installed on VirtualBox on the mac, it acted strangely with different test cases before, so possibly it also affected the performance of the server.

I ran the new status client from Part 5 on my server while trying to identify the steady state:

```
aiya@aiya-VirtualBox:~/Desktop/operating_systems/final_project$
./status 4444
Enter the status value you want to get or type q to quit.
CURRCLI
Server's response to requested command: 1

Enter the status value you want to get or type q to quit.
TOTPROD
Server's response to requested command: 592

Enter the status value you want to get or type q to quit.
REJSLOW
Server's response to requested command: 18

Enter the status value you want to get or type q to quit.
TOTCONS
Server's response to requested command: 590

Enter the status value you want to get or type q to quit.
```