The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 2, 2022
Assignment 4 (version 1.1)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 15%
Due: 6:00pm Friday 28 October, 2022

Specification changes since version 1.0 are shown in red and are summarised at the end of the document.

Introduction

The goal of this assignment is to further develop your C programming skills, and to demonstrate your understanding of networking and multithreaded programming. You are to create two programs which together implement a distributed communication architecture known as publish/subscribe. One program – `psserver` – is a network server which accepts connections from clients (including `psclient`, which you will implement). Clients connect, and tell the server that they wish to subscribe to notifications on certain topics. Clients can also publish values (strings) on certain topics. The server will relay a message to all subscribers of a particular topic. Communication between the `psclient` and `psserver` is over TCP using a newline-terminated text command protocol. Advanced functionality such as connection limiting, signal handling and statistics reporting are also required for full marks. CSSE7231 students shall also implement a simple HTTP interface to `psserver`.

The assignment will also test your ability to code to a particular programming style guide, to write a library to a provided API, and to use a revision control system appropriately.

Student Conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on the discussion forum. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students or other people with the actual design, structure and/or coding of your assignment solution. It is **cheating to look at another student's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct actions will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository, and do not allow others to access your computer – you must keep your code secure.

You must follow the following code referencing rules for all code committed to your SVN repository (not just the version that you submit):

Code Origin	**Usage/Referencing**
Code provided to you in writing **this semester** by CSSE2310/7231 teaching staff (e.g. code hosted on Blackboard, posted on the discussion forum, or shown in class).	May be used freely without reference. (You must be able to point to the source if queried about it.)
Code you have personally written this semester for CSSE2310/7231 (e.g. code written for A1 reused in A3)	May be used freely without reference. (This assumes that no reference was required for the original use.)
Code examples found in man pages on `moss`.	May be used provided the source of the code is referenced in a comment adjacent to that code. (Code you have *taken inspiration from* must not be directly copied or just converted from one programming language to another.)
Code you have personally written in a previous enrolment in this course or in another ITEE course and where that code has not been shared or published.	
Code (in any programming language) that you have taken inspiration from but have not copied.	

Code Origin	Usage/Referencing
Other code – includes: code provided by teaching staff only in a previous offering of this course (e.g. previous A1 solution); code from websites; code from textbooks; any code written by someone else, or partially written by someone else; and any code you have written that is available to other students.	May **not** be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken.

Uploading or otherwise providing the assignment specification or part of it to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of these sites and many cooperate with us in misconduct investigations.

The course coordinator reserves the right to conduct interviews with students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process. If you are not able to adequately explain your code or the design of your solution and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate.

In short - **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. You should read and understand the statements on student misconduct in the course profile and on the school web-site: `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`

The publish/subscribe communication model

From wikipedia:

> "In software architecture, publish-subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to receivers, called subscribers. Instead, senders categorise published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are."

In this way, the participants in the communication are very loosely coupled - when publishing a message on a given topic there is no knowledge or requirement that other participant be listening for that topic. When correctly implemented, publish/subscribe can be very efficient for scaling to large numbers of participants - take a look at the wikipedia page for more details if you are interested, but this is not required to complete this assignment.

Specification – `psclient`

The `psclient` program provides a commandline interface that allows you to participate in the publish/subscribe system as a client, connecting to the server, naming the client, subscribing to and publishing topics. `psclient` will also output notifications when topics to which that client is subscribed, get published.

To fully implement this functionality, `psclient` will either require two threads (easiest), or if you wish you may use `select()`. Choose whichever you prefer, as long as you achieve the required functionality.

Command Line Arguments

Your `psclient` program is to accept command line arguments as follows:

```
./psclient portnum name [topic] ...
```

- The mandatory `portnum` argument indicates which localhost port `psserver` is listening on – either numerical or the name of a service.
- The mandatory `name` argument specifies the name to be associated with this client, e.g. `barney`
- Any `topic` arguments, if provided, are to be treated as strings which are topics to which `psclient` should immediately subscribe after connecting to the server, e.g. `news`, `weather`

- If no topic arguments are provided then `psclient` will connect to the server without subscribing to any topics.

`psclient` behaviour

If insufficient command line arguments are provided then `psclient` should emit the following message (terminated by a newline) to `stderr` and exit with status 1:

```
Usage: psclient portnum name [topic] ...
```

If the correct number of arguments is provided, then further errors are checked for in the order below.

Restrictions on the name

The name argument must not contain any spaces, colons, or newlines. The name argument must not be an empty string. If either of these conditions is not met then `psclient` shall emit the following message (terminated by a newline) to `stderr` and exit with status 2:

```
psclient: invalid name
```

Topic argument(s)

The `topic` arguments are optional.

All topic arguments must not contain any spaces, colons, or newlines. All topic arguments must also not be empty strings. If either of these conditions is not met for any of the topics specified on the command line then `psclient` shall emit the following message (terminated by a newline) to `stderr` and exit with status 2:

```
psclient: invalid topic
```

Duplicated topic strings are permitted – your program does not need to check for these but should instead just attempt to subscribe multiple times. (A correctly implemented server will ignore requests to subscribe to a topic that is already subscribed to.)

Connection error

If `psclient` is unable to connect to the server on the specified port (or service name) of `localhost`, it shall emit the following message (terminated by a newline) to `stderr` and exit with status 3:

```
psclient: unable to connect to port N
```

where `N` should be replaced by the argument given on the command line. (This may be a non-numerical string.)

`psclient` runtime behaviour

Assuming that the commandline arguments are without errors, `psclient` is to perform the following actions, in this order, immediately upon starting:

- Connect to the server on the specified port number (or service name) – see above for how to handle a connection error.
- Provide the client's `name` to the server using the 'name' command (see the Communication protocol section for details of the `name` command).
- Send subscription requests to the server for any topics listed on the command line, in the order that they were specified (see the Communication protocol section for details of the `subscribe` command).

From this point, `psclient` shall simply output to its `stdout`, any lines received over the network connection from the server, without any error checking or processing. A correctly implemented `psserver` will only ever send publication notices, however your `psclient` does not need to check for this.

Specifically – **`psclient` is not responsible for any logic processing or error checking on messages received from the server** – for example if a faulty server keeps sending `psclient` publication messages on a topic to which `psclient` never subscribed, or from which it has unsubscribed, `psclient` does not have to detect this. It shall simply and blindly output those messages to `stdout`.

If the network connection to the server is closed (e.g. `psclient` detects EOF on the socket), then `psclient` shall emit the following message to `stderr` and terminate with exit status 4:

```
psclient: server connection terminated
```

Simultaneously and asynchronously, `psclient` shall be reading newline-terminated lines from `stdin`, and sending those lines unmodified to the server (effectively, this interface requires the user to type in command strings that are sent to the server – this simplifies the implementation of `psclient` dramatically). The three valid message types are:

- `pub <topic> <value>` – publish `<value>` under topic `<topic>`. Note that `<value>`s may contain spaces and colons.
- `sub <topic>` – subscribe this client to topic `<topic>`.
- `unsub <topic>` – unsubscribe this client from future publication of `<topic>`.

Note that `psclient` is not required to perform any error checking on the input lines – simply send them unmodified to the server. In this sense, `psclient` is a lot like `netcat`.

If `psclient` detects `EOF` on `stdin` or some other communication error, it shall ~~close its network connection to the server and~~ terminate with exit status 0 [1].

Your `psclient` program is not to register any signal handers nor attempt to mask or block any signals.

`psclient` example usage

Subscribing to a topic from the commandline, then sending an invalid publish (missing value) and then immediately publishing on that same topic and receiving the publication notice back from the server (assuming the `psserver` is listening on port 49152). Lines in **bold face** are typed interactively on the console, they are not part of the output of the program:

```
$./psclient 49152 fred topic1
pub topic1
:invalid
pub topic1 value 1
fred:topic1:value 1
```

Subscribing to a topic interactively, publishing on it (and receiving the publication notice back from the server), unsubscribing, then publishing again:

```
$./psclient 49152 fred
sub topic1
pub topic1 value1
fred:topic1:value1
unsub topic1
pub topic1 value1
```

Here you can see that the publishing of 'topic1' after subscribing, causes the client to receive the publication notice as expected. Then the client unsubscribes, and no notification is subsequently received despite the republication of the same topic.

Subscribing to several topics on the command line, and having these published by other clients at some point after connecting:

```
$./psclient 49152 fred topic1 topic2
...
barney:topic1:value 1
...
wilma:topic2:value:2
barney:topic2:some Value
...
unsub topic1
wilma:topic2:a String Here
...
```

[1] All file descriptors are closed on exit. Attempting to close a file descriptor in one thread that is blocked for reading in another thread may cause `fclose()` or `close()` to block.

Note that after sending the 'unsub' command, and assuming a correctly functioning server, we do not expect this client to receive any further publication messages regarding 'topic1'. Messages on other subscribed topics will still be received.

Specification – `psserver`

`psserver` is a networked publish/subscribe server, allowing clients to connect, name themselves, subscribe to and unsubscribe from topics, and publish messages setting values for topics. All communication between clients and the server is over TCP using a simple command protocol that will be described in a later section.

Command Line Arguments

Your `psserver` program is to accept command line arguments as follows:

```
./psserver connections [portnum]
```

In other words, your program should accept one mandatory argument (`connections`), and one optional argument which is the port number to listen on for connections from clients.

The `connections` argument indicates the maximum number of simultaneous client connections to be permitted. If this is zero, then there is no limit to how many clients may connect (other than operating system limits which we will not test).

The `portnum` argument, if specified, indicates which localhost port `psserver` is to listen on. If the port number is absent or zero, then `psserver` is to use an ephemeral port.

Important: Even if you do not implement the connection limiting functionality, your program must correctly handle command lines which include that argument (after which it can ignore any provided value – you will simply not receive any marks for that feature).

Program Operation

The `psserver` program is to operate as follows:

- If the program receives an invalid command line then it must print the message:

  ```
  Usage: psserver connections [portnum]
  ```

  to `stderr`, and exit with an exit status of 1.

  Invalid command lines include (but may not be limited to) any of the following:

  - no max connections number specified
  - the connections argument is not a non-negative integer
  - the port number argument (if present) is not an integer value, or is an integer value and is not either zero, or in the range of 1024 to 65535 inclusive
  - too many arguments are supplied

- If `portnum` is missing or zero, then `psserver` shall attempt to open an ephemeral localhost port for listening. Otherwise, it shall attempt to open the specific port number. If `psserver` is unable to listen on either the ephemeral or specific port, it shall emit the following message to `stderr` and terminate with exit status 2:

  ```
  psserver: unable to open socket for listening
  ```

- Once the port is opened for listening, `psserver` shall print to `stderr` the port number followed by a single newline character and then flush the output. **In the case of ephemeral ports, the actual port number obtained shall be printed, not zero**.

- Upon receiving an incoming client connection on the port, `psserver` shall spawn a new thread to handle that client (see below for client thread handling).

- If specified (and implemented), `psserver` must keep track of how many active client connections exist, and must not let that number exceed the `connections` parameter. See below on client handling threads for more details on how this limit is to be implemented.

- Note that all error messages must be terminated by a single newline character.
- The `psserver` program should not terminate under normal circumstances, nor should it block or otherwise attempt to handle `SIGINT`.
- Note that your `psserver` must be able to deal with any clients using the correct communication protocol, not just `psclient`. For example, note that a correctly implemented `psclient` will not attempt to publish or subscribe to topics before sending the `name` command, however you still need to ensure that `psserver` behaves correctly if that does happen. Testing with `netcat` is highly recommended.

Client handling threads

A client handler thread is spawned for each incoming connection. This client thread must then wait for commands from the client, one per line, over the socket. The exact format of the requests is described in the Communication protocol section below.

As each client sends subscribe, publish and unsubscribe commands to `psserver`, its client handling thread will need to determine which of the currently connected clients should receive topic publication messages.

Due to the simultaneous nature of the multiple client connections, your `psserver` will need to ensure mutual exclusion around any shared data structure(s) to ensure that these do not get corrupted.

Once the client disconnects or there is a communication error on the socket then the client handler thread is to close the connection, clean up and terminate. Other client threads and the `psserver` program itself must continue uninterrupted.

`SIGHUP` handling (Advanced)

Upon receiving `SIGHUP`, `psserver` is to emit (and flush) to `stderr` statistics reflecting the program's operation to-date, specifically

- Total number of clients connected (at this instant) – even if they have not yet sent a `name` message
- The total number of clients that have connected and disconnected since program start
- The total number of successful `pub` requests processed (since program start)
- The total number of successful `sub` requests received (since program start)
- The total number of successful `unsub` requests received (since program start). (This does not include unsubscriptions due to client disconnections.)

Successful requests are those that are valid and not ignored. (See the Communication protocol section for details.)

The required format is illustrated below.

Listing 1: `psserver` SIGHUP `stderr` output sample

```
Connected clients:4
Completed clients:20
pub operations:4
sub operations:15
unsub operations:0
```

Note that to accurately gather these statistics and avoid race conditions, you will need some sort of mutual exclusion structure protecting the variables holding these statistics.

Global variables are not to be used to implement signal handling. See the Hints section below for how you can implement this.

Client connection limiting (Advanced)

If the `connections` feature is implemented and a non-zero command line argument is provided, then `psserver` must not permit more than that number of simultaneous client connections to the server. `psserver` shall maintain a connected client count, and if a client beyond that limit attempts to connect, it shall block, indefinitely if required, until another client leaves and this new client's connection request can be `accept()`ed. Clients in this waiting state are not to be counted in statistics reporting – they are only counted once they have properly connected.

HTTP connection handling (CSSE7231 students only)

CSSE7231 students shall, in addition, implement a simple HTTP server in their `psserver` implementation. Upon startup, `psserver` shall check the value of the environment variable `A4_HTTP_PORT`. If set, then `psserver` shall also listen for connections on that port number (or service name).

If `psserver` is unable to listen on that port or service name then it shall emit the following message to `stderr` and terminate with exit status 3:

```
psserver: unable to open HTTP socket for listening
```

The ability to listen on this port is checked after the ability to listen on the "main" port (i.e. the one given on the command line). If the `A4_HTTP_PORT` environment variable is not set then `psserver` shall not listen on any additional ports and shall not handle these HTTP connections.

The communication protocol uses HTTP. The connecting program (e.g. `netcat`, or a web browser) shall send HTTP requests and `psserver` shall send HTTP responses as described below. The connection between client and server is kept alive between requests. Multiple HTTP clients may be connected simultaneously.

Additional HTTP header lines beyond those specified may be present in requests or responses and must be ignored by the respective server/client. Note that interaction between a client on the HTTP port and `psserver` is *synchronous* – any one client can only have a single request underway at any one time. This greatly simplifies the implementation of the `psserver` HTTP connection handling threads.

The only supported request method is a `GET` request in the following format:

- Request: `GET /stats HTTP/1.1`
  - Description: the client is requesting statistics from `psserver`
  - Request
    - Request Headers: none expected, any headers present will be ignored by `psserver`.
    - Request Body: none, any request body will be ignored
  - Response
    - Response Status: 200 (OK).
    - Response Headers: the `Content-Length` header with correct value is required (number of bytes in the response body), other headers are optional.
    - Response Body: the ASCII text containing the same `psserver` statistics information described in the `SIGHUP` handling section above.

If `psserver` receives an invalid HTTP request then it should close the connection to that requestor (and terminate the thread associated with that connection). Similarly, if a HTTP client disconnects, `psserver` should handle this gracefully and terminate the thread associated with the connection.

Program output

Other than error messages, the listening port number, and `SIGHUP`-initiated statistics output, `psserver` is not to emit any output to `stdout` or `stderr`.

Server data structures

While the publish/subscribe model is conceptually simple, its implementation in a multithreaded server is not trivial. This section provides some hints and guidance on how to architect `psserver` to keep it manageable.

Start by noting that `psserver` maintains one thread per connected client (call it the client-handler thread), however when a client sends a `publish` message to its handler thread, somehow that thread must identify **all** of the currently-connected clients that are subscribed to this topic.

For this you will clearly need a data structure that is accessible to all threads (and protected by mutual exclusion appropriately). But how should that structure be organised?

What we recommend, and are somewhat encouraging with the `stringmap` library and API that is part of this assignment (see the `stringmap` API section below), is as follows:

- Create a data structure to manage information about a specific client – its name, the file descriptor or `FILE*` handles used to talk to it and so on. Let's call this `struct Client`. A new one of these gets created and populated each time a new client connects.

- Create another data structure that can manage a collection of these `struct Client` (or rather, pointers to them) – it could be a linked list, or a dynamically allocated array, or however it makes most sense to you to implement. You need at least to be able to add, remove and iterate through the pointers to `struct Client` that are stored in this data structure.

- Once you can successfully manage collections of clients, you can use the `stringmap` API described below – this is a data structure that allows you to define mappings between strings and `void *` pointers. If you treat topics as stringmap keys, and the client collection data structure as the associated item then you get a nice way of identifying which clients are subscribed to which topics. When you need to publish a topic, you should retrieve the collection of subscribed clients from the stringmap, iterate through it and send the message to each one in turn. Note that in this case it is the client thread that receives a message that does the sending to all clients that are subscribed to that topic. Don't forget the mutual exclusion – you don't want another thread making changes to this while you are iterating!

You don't have to implement `psserver` this way, however if you want maximum marks you do have to implement the `stringmap` API anyway, so you might as well use it in your solution.

Communication protocol

The communication protocol between clients and `psserver` uses simple newline-terminated text message strings as described below. Messages from client to server are space delimited. Messages from server to client are colon delimited. Note that the angle brackets `<foo>` are used to represent placeholders, and are not part of the command syntax.

Supported messages from `psclient` to `psserver` are:

- `name <name>` – the client is naming itself `<name>`

- `sub <topic>` – the client is subscribing to `<topic>`, i.e. wishes to receive any future publication notifications on this topic

- `unsub <topic>` – the client is unsubscribing from `<topic>` and should no longer receive publication messages

- `pub <topic> <value>` – the client is publishing to the `<topic>` the given `<value>`. Any clients (including this one) that are subscribed to this topic, should receive a publication message

Single spaces are used as separators between fields but note that the `<value>` field in a `pub` message may contain spaces (and colons).

Supported messages from `psserver` to `psclient` are

- `:invalid` – sent by the server to a client if the server receives an invalid command from that client (see below)

- `<name>:<topic>:<value>` – sent by the server to all connected clients that have subscribed to `topic`. The `name` is the name of the client that published the message. Note that the `<value>` field may contain colons (and spaces).

Invalid commands that might be received by `psserver` from a client include:

- Invalid command word – an empty string or a command word which is not `pub`, `sub`, `name` or `unsub`

- Invalid arguments such as empty strings (for any field) or names or topics containing spaces or colons

- Too few or too many arguments

- Any other kind of malformed message

`psserver` shall send a `:invalid` response to a client on receipt of an invalid command.

Other requirements are as follows:

- `psserver` shall ignore any `name` command coming from a client that has already provided a name.

- `psserver` shall ignore duplicate subscription requests from a client (i.e. if a client subscribes more than once to a given topic it will only ever receive one copy of matching messages).

- `psserver` shall ignore `unsub` requests from a client on a topic that the client is not subscribed to.
- `psserver` shall ignore any commands coming from a client before that client has named itself. Anonymous clients may neither subscribe to nor publish topics not unsubscribe from topics.

Note that these are not "invalid" commands – they are well-formed commands that will just be ignored by `psserver`.

Note that `psserver` shall not attempt to detect or reject simultaneous duplicated names from multiple clients (e.g. two clients trying to call themselves `fred`). Such behaviour is undefined and will not be tested[2]. The only exception to the duplicated name restriction is that a name may be reused by a client from connection to connection – i.e. a client connects, calls itself `fred`, interacts with the server for some time and then disconnects. Later, if another client connects, then it may reuse the name `fred`.

The `stringmap` library and API

An API (Application Programming Interface) for the data structure implementation known as 'stringmap' is provided to you in the form of a header file, found on moss in `/local/courses/csse2310/include/stringmap.h`.

An implementation of `stringmap` is also available to you on moss, in the form of a shared library file (`/local/courses/csse2310/lib/libstringmap.so`). This will allow you to start writing `psserver` without first implementing the stringmap API.

However, **to receive full marks you must implement your own version of stringmap** according to the API specified by `stringmap.h`. You must submit your `stringmap.c` and your `Makefile` must build this to your own `libstringmap.so`.

Your `psserver` must use this API and link against `libstringmap.so`. Note that we will use our `libstringmap.so` when testing your `psserver` so that you are not penalised in the marking of `psserver` for any bugs in your stringmap implementation. You are free to use your own `libstringmap.so` when testing yourself – see below.

`stringmap.h` defines the following functions and types:

Listing 3: `stringmap.h` contents

```
#ifndef STRINGMAP_H
#define STRINGMAP_H

typedef struct StringMap StringMap;

// data structure stored in the StringMap
typedef struct {
    char *key;
    void *item;
} StringMapItem;

// Allocate, initialise and return a new, empty StringMap
StringMap *stringmap_init(void);

// Free all memory associated with a StringMap.
// frees stored key strings but does not free() the (void *)item pointers
// in each StringMapItem. Does nothing if sm is NULL.
void stringmap_free(StringMap *sm);

// Search a stringmap for a given key, returning a pointer to the entry
// if found~~, else NULL~~. If not found or sm is NULL or key is NULL then returns NULL.
void *stringmap_search(StringMap *sm, char *key);

// Add an item into the stringmap, return 1 if success else 0 (e.g. an item
// with that key is already present or any one of the arguments is NULL)
// The 'key' string is copied before being stored in the stringmap.
// The item pointer is stored as-is, no attempt is made to copy its contents.
```

[2] you can implement this if you want but it won't be easy or pretty, nor will it get you any marks

```
int stringmap_add(StringMap *sm, char *key, void *item);

// Removes an entry from a stringmap
// free()stringMapItem and the copied key string, but not
// the item pointer.
// Returns 1 if success else 0 (e.g. item not present or any argument is NULL)
int stringmap_remove(StringMap *sm, char *key);

// Iterate through the stringmap - if prev is NULL then the first entry is returned
// otherwise prev should be a value returned from a previous call to stringmap_iterate()
// and the "next" entry will be returned.
// This operation is not thread-safe - any changes to the stringmap between successive
// calls to stringmap_iterate may result in undefined behaviour.
// Returns NULL if no more items to examine or sm is NULL.
// There is no expectation that items are returned in a particular order (i.e.
// the order does not have to be the same order in which items were added).
StringMapItem *stringmap_iterate(StringMap *sm, StringMapItem *prev);

#endif
```

Note that it is not expected that your stringmap library be thread safe – the provided `libstringmap.so` is not.

For example code that shows how to use the `stringmap` API see `/local/courses/csse2310/resources/a4/stringmaptest.c` on moss. If you copy this file to one of your own directories then you can build it with the command:

```
gcc -I/local/courses/csse2310/include -L/local/courses/csse2310/lib -lstringmap
stringmaptest.c -o stringmaptest
```

Creating shared libraries – `libstringmap.so`

This section is only relevant if you are creating your own implementation of `libstringmap.o`.

Special compiler and linker flags are required to build shared libraries. Here is a Makefile fragment you can use to turn `stringmap.c` into a shared library, `libstringmap.so`:

Listing 4: Makefile fragment to compile and build a shared library

```
CC=gcc
LIBCFLAGS=-fPIC -Wall -pedantic -std=gnu99

# Turn stringmap.c into stringmap.o
stringmap.o: stringmap.c
        $(CC) $(LIBCFLAGS) -c $<

# Turn stringmap.o into shared library libstringmap.so
libstringmap.so: stringmap.o
        $(CC) -shared -o $@ stringmap.o
```

`stringmap.h` should only be listed as a dependency in your Makefile if you have a local copy in your repository. This is not required, but if you do include a copy of `stringmap.h` in your submission then it must be styled correctly.

Running `psserver` with your `libstringmap.so` library

The shell environment variable `LD_LIBRARY_PATH` specifies the path (set of directories) searched for shared libraries. On moss, by default this includes `/local/courses/csse2310/lib`, which is where the provided libraries `libcsse2310a4.so` and our implementation of `libstringmap.so` live.

If you are building your own version and wish to use it when you run `psserver`, then you'll need to make sure that **your** version of `libstringmap.so` is used. To do this you can use the following:

```
LD_LIBRARY_PATH=.:${LD_LIBRARY_PATH} ./psserver
```

This commandline sets the `LD_LIBRARY_PATH` **just for this specific run of `psserver`**, causing it to dynamically link against your version of `libstringmap.so` in the current directory instead of the one we have provided.

Note that we will use our `libstringmap.so` when testing your `psserver` unless you statically link your stringmap into your `psserver`. Your `libstringmap.so` will be tested independently.

Provided Libraries

libstringmap

See above. This must be linked with `-L/local/courses/csse2310/lib -lstringmap`. It is recommended not to statically link your own `stringmap.o` into your `psserver` or hardwire the path to `libstringmap.so`. If you wish to test your own `libstringmap.so` then you can use the approach described above.

libcsse2310a4

A `split_by_char()` function is available to break a line up into multiple parts, e.g. based on spaces. This is similar to the `split_line()` function from `libcsse2310a3` though allows a maximum number of fields to be specified.

Several additional library functions have been provided to aid CSSE7231 students in parsing/construction of HTTP requests/responses. The functions in this library are:

```
char** split_by_char(char* str, char split, unsigned int maxFields);

int get_HTTP_request(FILE *f, char **method, char **address,
        HttpHeader ***headers, char **body);

char* construct_HTTP_response(int status, char* statusExplanation,
        HttpHeader** headers, char* body);

int get_HTTP_response(FILE *f, int* httpStatus, char** statusExplain,
        HttpHeader*** headers, char** body);

void free_header(HttpHeader* header);

void free_array_of_headers(HttpHeader** headers);
```

These functions and the `HttpHeader` type are declared in `/local/courses/csse2310/include/csse2310a4.h` on moss and their behaviour is described in man pages on moss – see `split_by_char(3)`, `get_HTTP_request(3)`, and `free_header(3)`.

To use these library functions, you will need to add `#include <csse2310a4.h>` to your code and use the compiler flag `-I/local/courses/csse2310/include` when compiling your code so that the compiler can find the include file. You will also need to link with the library containing these functions. To do this, use the compiler arguments `-L/local/courses/csse2310/lib -lcsse2310a4`

(You only need to specify the `-I` and `-L` flags once if you are using multiple libraries from those locations, e.g. both `libstringmap` and `libcsse2310a4`.)

libcsse2310a3

You are also welcome to use the `"libcsse2310a3"` library from Assignment 3 if you wish. This can be linked with `-L/local/courses/csse2310/lib -lcsse2310a3`. Note that `split_space_not_quote()` is not appropriate for use in this assignment – quotes have no particular meaning in the communication protocol for `psserver`.

Style

Your program must follow version 2.2 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site. Note that, in accordance with the style guide, function comments are not expected in your `stringmap.c` for functions that are declared and commented in `stringmap.h`.

Hints

1. Review the lectures related to network clients, HTTP, threads and synchronisation and multi-threaded network servers. This assignment builds on all of these concepts.
2. You can test `psclient` and `psserver` independently using `netcat` as demonstrated in the lectures. You can also use the provided demo programs `demo-psclient` and `demo-psserver`.
3. The `read_line()` function from `libcsse2310a3` may be useful in both `psclient` and `psserver`
4. The multithreaded network server example from the lectures can form the basis of `psserver`.
5. Use the provided library functions (see above).
6. Consider the `strchr()` function to search a string for a particular character (e.g. colon).
7. Consider a dedicated signal handling thread for `SIGHUP`. `pthread_sigmask()` can be used to mask signal delivery to threads, and `sigwait()` can be used in a thread to block until a signal is received. You will need to do some research and experimentation to get this working.

Forbidden Functions

You must not use any of the following C functions/statements. If you do so, you will get zero (0) marks for the assignment.

- `goto`
- `longjmp()` and equivalent functions
- `system()`
- `mkfifo()` or `mkfifoat()`
- POSIX regex functions
- `fork()`, `pipe()`, `popen()`, `execl(), execvp()` and other `exec` family members.

Submission

Your submission must include all source and any other required files (in particular you must submit a `Makefile`). Do not submit compiled files (eg `.o`, compiled programs) or test files. **You are not expected to submit `stringmap.h`. If you do so, it will be marked for style.**

Your programs (named `psclient` and `psserver`) and your stringmap library (named `libstringmap.so` if you implement it) must build on `moss.labs.eait.uq.edu.au` with:

```
make
```

If you only implement one or two of the programs/library then it is acceptable for `make` to just build those programs/library – and we will only test those programs/library.

Your programs must be compiled with gcc with at least the following switches (plus applicable `-I` options etc. – see *Provided Libraries* above):

```
-pedantic -Wall -std=gnu99
```

You are not permitted to disable warnings or use pragmas to hide them. You may not use source files other than `.c` and `.h` files as part of the build process – such files will be removed before building your program.

If any errors result from the `make` command (e.g. an executable can not be created) then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality).

Your program must not invoke other programs or use non-standard headers/libraries (besides those we have provided for you to use).

Your assignment submission must be committed to your subversion repository under

```
https://source.eait.uq.edu.au/svn/csse2310-sem2-sXXXXXXX/trunk/a4
```

where sXXXXXXX is your moss/UQ login ID. Only files at this top level will be marked so **do not put source files in subdirectories**. You may create subdirectories for other purposes (e.g. your own test files) but these will not be considered in marking – they will not be checked out of your repository.

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and within the `trunk/a4` directory in your repository (and not within a subdirectory) and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly encouraged to check out a clean copy for testing purposes – the `reptesta4.sh` script will do this for you.

To submit your assignment, you must run the command

```
2310createzip a4
```

on `moss` and then submit the resulting zip file on Blackboard (a GradeScope submission link will be made available in the Assessment area on the CSSE2310/7231 Blackboard site)[3]. The zip file will be named

`sXXXXXXX_csse2310_a4_`*timestamp*`.zip`

where sXXXXXXX is replaced by your moss/UQ login ID and *timestamp* is replaced by a timestamp indicating the time that the zip file was created.

The `2310createzip` tool will check out the latest version of your assignment from the Subversion repository, ensure it builds with the command '`make`', and if so, will create a zip file that contains those files and your Subversion commit history and a checksum of the zip file contents. You may be asked for your password as part of this process in order to check out your submission from your repository.

You must not create the zip file using some other mechanism and you must not modify the zip file prior to submission. If you do so, you will receive zero marks. Your submission time will be the time that the file is submitted via GradeScope on Blackboard, and **not** the time of your last repository commit nor the time of creation of your submission zip file.

We will mark your last submission, even if that is after the deadline and you made submissions before the deadline. Any submissions after the deadline[4] will incur a late penalty – see the CSSE2310/7231 course profile for details.

Marks

Marks will be awarded for functionality and style and documentation. Marks may be reduced if you are asked to attend an interview about your assignment and you are unable to adequately respond to questions – see the **Student conduct** section above.

Functionality (60 marks)

Provided your code compiles (see above) and does not use any prohibited statements/functions (see above), and your zip file has not been modified prior to submission, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. Reasonable time limits will be applied to all tests. If your program takes longer than this limit, then it will be terminated and you will earn no marks for the functionality associated with that test.

Exact text matching of files and output (`stdout` and `stderr`) is used for functionality marking. Strict adherence to the output format in this specification is critical to earn functionality marks.

The markers will make no alterations to your code (other than to remove code without academic merit). Note that your client and server will be tested independently.

Marks will be assigned in the following categories. There are 10 marks for `libstringmap.o`, 10 marks for `psclient` and 40 marks (CSSE2310) or 50 marks (CSSE7231) for `psserver`.

1. `libstringmap` – correctly create and free memory associated with a stringmap object (3 marks)
2. `libstringmap` – correctly add and retrieve items (3 marks)
3. `libstringmap` – correctly delete items (2 marks)

[3] You may need to use scp or a graphical equivalent such as WinSCP, Filezilla or Cyberduck in order to download the zip file to your local computer and then upload it to the submission site.

[4] or your extended deadline if you are granted an extension.

4. `libstringmap` – correctly iterate through a stringmap (2 marks)

5. `psclient` correctly handles invalid command lines (including invalid names and topics) (2 marks)
6. `psclient` connects to server and also handles inability to connect to server (2 marks)
7. `psclient` correctly sends the required `name` and `sub` commands upon startup (2 marks)
8. `psclient` correctly displays lines received from the server and sends input lines to the server (2 marks)
9. `psclient` correctly handles communication failure and EOF on stdin (2 marks)

10. `psserver` correctly handles invalid command lines (3 marks)
11. `psserver` correctly listens for connections and reports the port (including inability to listen for connections) (3 marks)
12. `psserver` correctly handles invalid command messages (5 marks)
13. `psserver` correctly handles subscription and publication requests (8 marks)
14. `psserver` correctly handles unsubscription requests (5 marks)
15. `psserver` correctly handles multiple simultaneous client connections using threads (including protecting data structures with mutexes) (5 marks)
16. `psserver` correctly handles disconnecting clients and communication failure (3 marks)
17. `psserver` correctly implements client connection limiting (4 marks)
18. `psserver` correctly implements SIGHUP statistics reporting (4 marks)

19. (CSSE7231 only) `psserver` correctly listens on the port specified by `A4_HTTP_PORT` (and handles inability to listen or environment variable not set) (3 marks)
20. (CSSE7231 only) `psserver` correctly responds to HTTP requests (including invalid requests) from a single client issuing one request per connection (4 marks)
21. (CSSE7231 only) `psserver` supports multiple simultaneous HTTP clients and multiple sequential requests over each connection (3 marks)

Some functionality may be assessed in multiple categories, e.g. it is not possible test the handling of unsubscription messages without being able to publish messages. Note that the ability to support multiple simultaneous clients will be covered in multiple categories. Category 15 is about using threads and ensuring data structures are protected.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.2 of the CSSE2310/CSSE7231 C Programming Style Guide (found on Blackboard). Style marks will be made up of two components – automated style marks and human style marks. These are detailed below.

You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final – it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style marks can never be more than your functionality mark – this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You are encouraged to use the `style.sh` tool installed on `moss` to style check your code before submission. This does not check all style requirements, but it will determine your automated style mark (see below). Other elements of the style guide are checked by humans.

All `.c` and `.h` files in your submission will be subject to style marking. This applies whether they are compiled/linked into your executable or not[5].

[5]Make sure you remove any unneeded files from your repository, or they will be subject to style marking.

Automated Style Marking (5 marks)

Automated style marks will be calculated over all of your `.c` and `.h` files as follows. If any of your submitted `.c` and/or `.h` files are unable to be compiled by themselves then your automated style mark will be zero (0). (Automated style marking can only be undertaken on code that compiles. The provided `style.sh` script checks this for you.)

If your code does compile then your automated style mark will be determined as follows: Let

- W be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)
- A be the total number of style violations detected by `style.sh` when it is run over each of your `.c` and `.h` files individually[6].

Your automated style mark S will be

$$S = 5 - (W + A)$$

If $W + A \geq 5$ then S will be zero (0) – no negative marks will be awarded. Note that in some cases `style.sh` may erroneously report style violations when correct style has been followed. If you believe that you have been penalised incorrectly then please bring this to the attention of the course coordinator and your mark can be updated if this is the case. Note also that when `style.sh` is run for marking purposes it may detect style errors not picked up when you run `style.sh` on moss. This will not be considered a marking error – it is your responsibility to ensure that all of your code follows the style guide, even if styling errors are not detected in some runs of `style.sh`.

Human Style Marking (5 marks)

The human style mark (out of 5 marks) will be based on the criteria/standards below for "comments", "naming" and "other". The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide. Note that functions longer than 50 lines will be penalised in the automated style marking. Functions that are also longer than 100 lines will be further penalised here.

Comments (2.5 marks)

Mark	Description
0	The majority (50%+) of comments present are inappropriate OR there are many required comments missing
0.5	The majority of comments present are appropriate AND the majority of required comments are present
1.0	The vast majority (80%+) of comments present are appropriate AND there are at most a few missing comments
1.5	All or almost all comments present are appropriate AND there are at most a few missing comments
2.0	Almost all comments present are appropriate AND there are no missing comments
2.5	All comments present are appropriate AND there are no missing comments

Naming (1 mark)

Mark	Description
0	At least a few names used are inappropriate
0.5	Almost all names used are appropriate
1.0	All names used are appropriate

[6] Every `.h` file in your submission must make sense without reference to any other files, e.g., it must `#include` any `.h` files that contain declarations or definitions used in that `.h` file.

Other (1.5 marks)

Mark	Description
0	One or more functions is longer than 100 lines of code OR there is more than one global/static variable present inappropriately OR there is a global struct variable present inappropriately OR there are more than a few instances of poor modularity (e.g. repeated code)
0.5	All functions are 100 lines or shorter AND there is at most one inappropriate non-struct global/static variable AND there are at most a few instances of poor modularity
1.0	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no or very limited use of magic numbers AND there is at most one instance or poor modularity
1.5	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no use of magic numbers AND there are no instances of poor modularity

SVN commit history assessment (5 marks)

Markers will review your SVN commit history for your assignment up to your submission time. This element will be graded according to the following principles:

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section)
- Appropriate use of log messages to capture the changes represented by each commit. (Meaningful messages explain briefly what has changed in the commit (e.g. in terms of functionality) and/or why the change has been made and will be usually be more detailed for significant changes.)

The standards expected are outlined in the following rubric:

Mark (out of 5)	Description
0	Minimal commit history – single commit OR all commit messages are meaningless.
1	Some progressive development evident (more than one commit) OR at least one commit message is meaningful.
2	Some progressive development evident (more than one commit) AND at least one commit message is meaningful.
3	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful.
4	Multiple commits that show progressive development of all functionality AND meaningful messages for most commits.
5	Multiple commits that show progressive development of all functionality AND meaningful messages for ALL commits.

Total Mark

Let

- F be the functionality mark for your assignment (out of 60 for CSSE2310, or 70 for CSSE7231).
- S be the automated style mark for your assignment (out of 5).
- H be the human style mark for your assignment (out of 5)
- C be the SVN commit history mark (out of 5)

Your total mark for the assignment will be:

$$M = F + \min\{F, S + H\} + \min\{F, C\}$$

out of 75 (for CSSE2310 students) or 85 (for CSSE7231 students).

In other words, you can't get more marks for style or SVN commit history or documentation than you do for functionality. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on the discussion forum or emailed to `csse2310@uq.edu.au`.

Version 1.1

- Updated `stringmap.h` contents to clarify expected behaviour of the library functions.
- Clarified `psclient` behaviour on EOF on `stdin` (and added checking this to marking category 9).
- Clarified that connected clients (for statistics) include those that have not yet sent a `name` message.
- Clarified that anonymous clients (those yet to send a valid `name` message) can't unsubscribe from topics.