The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 2, 2021
Assignment 4 (version 1.0)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 20%
Due: 3:59pm 1 November, 2021

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 will also learn about a fundamental technique in computational mathematics called numerical integration. You are to create two programs. One – `intserver` – is a network server that takes requests from a client, in the form of a mathematical function and additional parameters, and will compute the integral (area under the curve) of that function using multiple threads. The other – `intclient` – is a simple network client that is used to send jobs to `intserver`. Advanced functionality such as thread limiting, signal handling and job statistics reporting are also required for full marks.

The assignment will also test your ability to code to a particular programming style guide, to use a provided library, 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 proceedings 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.

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 they cooperate with us in misconduct investigations.

You may use code provided to you by the CSSE2310/CSSE7231 teaching staff **in this current semester** and you may use code examples that are found in man pages on moss. If you do so, you **must** add a comment in your code (adjacent to that code) that references the source of the code. If you use code from **other** sources then this is either misconduct (if you don't reference the code) or code without academic merit (if you do reference the code). Code without academic merit will be removed from your assignment prior to marking (which may cause compilation to fail) but this will not be considered misconduct.

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`

Numerical integration

For mathematical functions of a single variable, such as $f(x) = sin(x)$, integration is the way of calculating the area underneath the curve from points a to b, as shown in Figure 1.

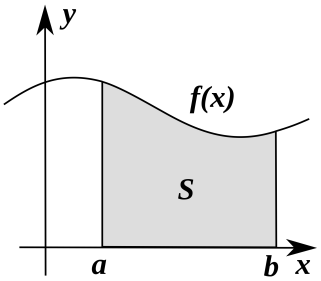

Figure 1: Integration as the area under a curve[1]

Mathematically[2], this is written as

$$S = \int_a^b f(x)dx$$

and a and b are called the lower and upper bounds of the integral respectively.

For some functions there are exact expressions for computing integrals. However, in many cases it is not possible to calculate exactly, and we use methods generally called 'numerical integration'. The idea is to slice the area into fine rectangular strips, calculate the area of those strips (which is easy, it's just width multiplied by height), and add them all together to approximate the area under the curve as shown in Figure 2.

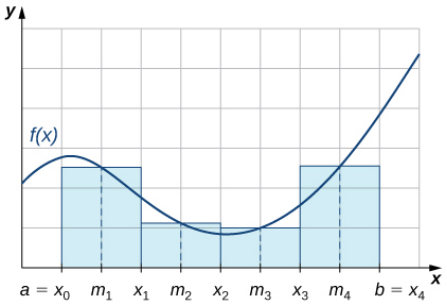

Figure 2: Rectangular approximation of integral[3]

Finally, instead of treating the strips as rectangles we can model them as trapezoids which gives us a more accurate estimate of the area, illustrated in Figure 3.

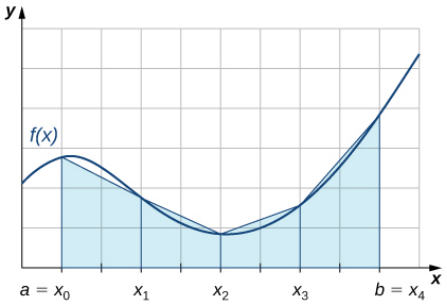

Figure 3: Trapezoidal approximation of integral[3]

[1]By 4C - Own work, based on JPG version, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=1039841`
[2]You do not have to understand this mathematical notation – it is provided for those who do understand this notation.
[3]Source: `https://opentextbc.ca/calculusv2openstax/chapter/numerical-integration/`

Given a trapezoid of width w, and sides y_1, y_2, the area of that trapezoid is given by

$$A = w\frac{y_1 + y_2}{2}$$

In general, the formula for the trapezoidal estimate of the area under a curve $f(x)$ from a to b, with N trapezoids is given as follows. First, slice the range a to b up into evenly spaced x values as follows:

$$a = x_0 < x_1 < \ldots < x_{N-1} < x_N = b$$

then

$$\int_a^b f(x)dx \approx \frac{b-a}{N} * [f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{N-1}) + f(x_N)]$$

Here are some helpful resources to aid your understanding of numerical and trapezoidal integration:

- Khan academy explanation of the trapezoidal method for approximating integrals: `https://www.khanacademy.org/math/ap-calculus-ab/ab-integration-new/ab-6-2/v/trapezoidal-approximation-of-area-under-curve`
- Wikipedia page: `https://en.wikipedia.org/wiki/Trapezoidal_rule`

Parallel numerical integration

The nice thing about numerical integration is we can easily split it into smaller jobs, compute those sub-integrals then add them together at the end.

For example, to integrate $f(x)$ from a to b, we can pick another point c between a and b, compute the two integrals, then add them together

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$$

In an analogous way, we can divide the numerical integration problem into any number of subproblems, run them in parallel, and combine the results to estimate the overall integral. In this assignment, your `intserver` program will do exactly this, spawning threads to do the computation in parallel.

A concrete example

Consider an integral from $a = 0$ to $b = 10$, in $N = 100$ segments, to be distributed over $M = 10$ threads (with each thread doing 10 segments).

The range $[0.0 - 10.0]$ divided into 100 segments will yield 100 trapezoids each of width $10/100 = 0.1$ unit. Thus, for this integral to be distributed over 10 threads, each thread will do the sub-integrals as follows:

thread	lower	upper	segments
1	0.0	1.0	10 (each 0.1 wide)
2	1.0	2.0	10 (each 0.1 wide)
...			
10	9.0	10.0	10 (each 0.1 wide)

The general case

In general, given the range $[a - b]$, N segments and M threads, for each thread $i = 1 \ldots M$ we will have

$w_i = \frac{b-a}{M}$ (the width of the region assigned to thread i)

$a_i = a + (i-1) * w_i$ (the lower bound of thread i's segment)

$b_i = a + i * w_i$ (the upper bound of thread i's segment)

and each trapezoidal strip will be of equal width

$w = \frac{b-a}{N}$

Specification – `intclient`

The `intclient` program reads job specifications either from an input file or `stdin`, and communicates those job parameters to `intserver` via a network socket connection. Jobs are described by the function to be integrated, the bounds of the integration, the number of integration segments and the number of parallel processing threads to be used. Assuming the job is accepted, `intclient` then waits for `intserver` to respond with the integration result, which is then emitted to `intclient`'s `stdout` in a particular format. The communication protocol between `intclient` and `intserver` is described in detail in Section Communication protocol.

`intclient` does not need to be multithreaded, but you may do so if you wish.

Command Line Arguments

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

```
./intclient [-v] portnum [jobfile]
```

- The `-v` optional argument indicates that `intclient` is to operate in verbose mode – details below.
- The `portnum` argument indicates which port `intserver` is listening on.
- The optional `jobfile` argument specifies the path to a file which describe the integration calculations to be performed (syntax below). If `jobfile` is not provided, then `intclient` is to read job specifications from `stdin`.

`intclient` Job File Syntax

Lines beginning with the '`#`' character[4] are treated as comments and are to be ignored. Empty lines, including lines with only space and/or tab characters, are to be ignored. All other lines in a job file must specify a job, in the following comma-separated syntax:

```
function,lower,upper,segments,threads
```

The meaning of these fields is described below. All fields are mandatory.

- `function` – the function expression to be integrated, expressed in terms of variable `x`. Any valid `tinyexpr` expression without whitespace present is supported, including but not limited to the following examples:
  - `0` (a constant is a valid expression)
  - `sin(x)`
  - `sqrt(x-1)`
- `lower` – the lower bound of the integral (a in the integration summary section), expressed in any valid floating point format that can be parsed by `sscanf()` and the `'%lf'` format specifier, such as 1.3, 4.5E2, -3.141592365
- `upper` – the upper bound of the integral (b) - same formatting as 'lower'
- `segments` – the number of trapezoidal segments (N) to be used in the integral computation – must be a positive integer
- `threads` – the number of computation threads to be spawned to perform the integration – must be a positive integer

`intclient` Job Rules and Error Checking

If `intclient` identifies a syntax error when parsing the job file (or `stdin`), it should emit the following message to `stderr` and continue processing lines from the file (or `stdin` as appropriate)

```
intclient: syntax error on line N
```

where `N` is replaced by the line number containing the offending job specifier.

Syntax errors include:

[4] i.e. the `#` character is the very first character on the line. `#` characters in any other position do not have any special meaning.

- there are not five non-empty comma separated fields on the line;
- the values in the `lower` or `upper` fields can't be parsed with the '`%lf`' specifier to `sscanf())` or there are surplus characters remaining in the field after such parsing;
- the values in the `segments` or `threads` fields can't be parsed with the '`%d`' specifier to `sscanf())` or there are surplus characters remaining in the field after such parsing.

Note that `sscanf()` will skip over leading spaces, so leading spaces are permitted in numerical fields, but trailing spaces or other surplus characters are a syntax error.

If a job line parses correctly (i.e. no syntax error is identified), `intclient` must then check the validity of the job parameters as follows. Note:

- All of the following must be true for each job.
- If any of these conditions is not met for a given job, then `intclient` should output the indicated error message and proceed immediately to the next input line (i.e. no further errors are checked for this line).
- Job error conditions should be checked in this order, and emit the error message for the first problem found.
- In all cases below, the line number placeholder *`N`* should be replaced by the offending line number.

Job error conditions and messages are as follows:

- `function` must not contain any whitespace characters (i.e. characters for which `isspace()` returns true). If whitespace characters are detected then `intclient` shall emit the following to `stderr`

  `intclient: spaces not permitted in expression (line N)`

- `upper` must be strictly greater than `lower`. If not, then `intclient` shall emit the following to `stderr`

  `intclient: upper bound must be greater than lower bound (line N)`

- `segments` must be a positive integer. If this condition is not met `intclient` shall emit the following to `stderr`

  `intclient: segments must be a positive integer (line N)`

- `threads` must be a positive integer. If this condition is not met `intclient` shall emit the following to `stderr`

  `intclient: threads must be a positive integer (line N)`

- The number of segments must be an integer multiple of the number of threads – that is each thread is to be given an integer number of segments to compute. Thus 1000 segments and 100 threads is valid (10 segments per thread), but 1000 segments and 30 threads is invalid. If this condition is not met `intclient` shall emit the following to `stderr`

  `intclient: segments must be an integer multiple of threads (line N)`

- `function` must be passed to `intserver` to check whether the expression is a valid function of the single variable `x` (see Communication protocol below). An invalid expression should result in the following error message on `stderr`:

  `intclient: bad expression "`*`function`*`" (line N)`

  where the quotation characters are printed, and *`function`* is replaced with the offending expression.

All error messages must be terminated by a single newline.

`intclient` Job File Examples

Following are several examples of valid integration job specification files.

Listing 1: Single threaded job specification file

```
# integrate sin(x) from zero to 3.14159265, in 1000 steps with 1 thread
sin(x),0,3.14159265,1000,1
```

Listing 2: Multi-threaded job specification file

```
# integrate sin(x) from zero to 3.14159265, in 1000 steps across 100 threads
sin(x),0,3.14159265,1000,100
```

Listing 3: Multi-job, multi-threaded specification file

```
# integrate sin(x) from zero to 3.14159265, in 1000000 steps across 1000 threads!
sin(x),0,3.14159265,1000000,1000
# Calculate pi as the 4 times the area of a quarter unit circle - slow
4*sqrt(1-x^2),0,1,1000000,1
# Calculate pi as the 4 times the area of a quarter unit circle - fast!
4*sqrt(1-x^2),0,1,1000000,100
```

Program Operation

When `intclient` runs it shall first check the command line arguments. If the command line is not valid, `intclient` shall emit the following to `stderr` and exit with error code 1:

```
Usage: intclient [-v] portnum [jobfile]
```

`intclient` shall next attempt to connect to `intserver` on the given port number (on `localhost`). If `intclient` is unable to connect to the port, it shall emit the following message to `stderr` and exit with error code 2:

```
intclient: unable to connect to port N
```

where `N` is replaced by the port number.

Once connected, `intclient` shall read and parse the jobfile (or `stdin`), one line at a time. As each integration job is successfully parsed, it shall be send to `intserver` to perform the integration using the request protocol described in the Communication protocol section below.

Once the job details are sent, `intclient` shall wait for a response from the server. This will either be an indication of success or failure. If it is an indication of failure, then `intclient` shall print the following to `stderr` and move on to the next job line:

```
intclient: integration failed
```

If the integration job succeeded and if the `-v` option was provided to `intclient`, then it shall emit to `stdout` the partial integral results returned by `intserver` as follows:

Listing 4: `intclient` verbose output sample

```
thread 1:0.000->0.001:1.2345
thread 2:0.001->0.002:3.45678
...
```

Each line in the verbose listing should have the following structure:
`thread` *N*`:`*from*`->`*to*`:`*partial_result*
where

- *N* is the computation thread number (from 1 to the number of threads)
- *from* and *to* are the lower and upper bounds of that integral segment (printed in default '`%lf`' `printf()` format)
- *partial_result* is the partial integral for that segment ('`%lf`' format)

Finally, `intclient` shall emit the overall result of a successful integration computation to `stdout` as follows, with the appropriate substitutions for the *<italicised fields>*

`The integral of` *expr* `from` *a* `to` *b* `is` *result*
where

- *expr* is the function given in the job specification;

- *a* and *b* are the lower and upper bounds of that integral ('%lf' format);
- *result* is the overall result of the integration (printed using '%lf' format).

All lines printed must be terminated by a single newline.
Additional notes on verbose output:

- If present, the verbose output will appear **before** the overall integral result.
- The message protocol (see Communication protocol below) provides a way for `intclient` to instruct `intserver` to return verbose result information in the required format.

If at any time `intclient` receives an unexpected end of file or other error on the socket connection, or receives an badly formed response from `intserver`, it shall emit the following message to `stderr` and terminate with exit code 3:

```
intclient: communications error
```

Once `intclient` reaches `EOF` on the input file (or `stdin`) – i.e. no jobs remain – then it should terminate with exit code 0.

Specification – `intserver`

Command Line Arguments

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

```
./intserver portnum [maxthreads]
```

In other words, your program should accept one mandatory argument (`portnum`), and one optional argument which is the maximum number of computation threads to be launched at any one time.

The `portnum` argument indicates which port `intserver` is to listen on. If the port number of zero is specified, then `intserver` is to use an ephemeral port.

The `maxthreads` argument, if present, specifies that maximum number of computing threads that the server is to spawn at any given time. If no `maxthreads` argument is given, then there is no limit to the number of computation threads.

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

Program Operation

The `intserver` program is to operate as follows:

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

  ```
  Usage: intserver portnum [maxthreads]
  ```

  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 port number specified
  - the port number parameter is not an integer value in the range of 0 to 65535 inclusive
  - the maxthreads argument, if specified, is not a positive integer

- If `portnum` is zero, then `intserver` shall attempt to open an ephemeral port for listening. Otherwise, it shall attempt to open the specific port number.
- If `intserver` is unable to open either the ephemeral or specific port, it shall emit the following message to `stderr` and terminate with exit code 3:

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

- Once the port is opened for listening, `intserver` shall print and flush the port number to `stderr`, followed by a single newline character. **In the case of an ephemeral port, the actual port number obtained shall be printed, not zero**.
- Upon receiving an incoming client connection on the port, `intserver` shall spawn a new thread to handle that client (see below for client thread handling).
- If specified (and implemented), `intserver` must keep track of how many worker threads (computing threads) are in operation at any instant, and must not let that number exceed the `maxthreads` 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 `intserver` program should not terminate under normal circumstances, nor should it block or otherwise attempt to handle `SIGINT`.

Client handling threads, and multithreaded integration

A client handler thread is spawned for each incoming connection[5]. This client thread must then wait for expression parsing or integration job requests, one at a time, over the socket. The exact format of the job request is described in the Communication protocol section below.

For an expression validation request, the client thread must use the `tinyexpr` library to check whether the expression provided is a valid function of the single variable `x`.

For an integration request, the client thread must first validate the job request parameters (including the function expression), in a similar manner to that described for `intclient`. The only difference is that if an incoming integration job request is invalid in some way, the `intserver` client thread is to send a bad request message back to the client (see Communication protocol below for details).

If the job is valid the client thread shall then spawn worker (computation) threads to perform the integration. The integration parts shall be divided evenly among all threads as described earlier. The client thread shall add the results from each computation thread to calculate the overall result.

If you do not implement multithreaded integration then the client handler thread may do the integral computation itself, however you will lose marks.

Once the integral computation is completed, the client handler thread shall send the result back to the client, and then return to waiting for a new request from that client.

Once the client disconnects or there is a communication error on the socket or a badly formed request is received then the client handler thread is to close the connection, clean up and terminate. (A properly formed request for an unavailable service shall be rejected with a Bad Request response - it will not result in termination of the client thread.)

`SIGHUP` handling (Advanced)

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

- Total number of clients connected (at this instant)
- Total number of expression validation requests received (since program start)
- Total number of integration jobs calculated (since program start)
- Total number of bad (rejected) integration requests (since program start)
- Total number of computation threads (since program start, including any currently active)

The required format is illustrated below.

Listing 5: `intserver` SIGHUP `stderr` output sample

```
Connected clients:4
Expressions checked:20
Completed jobs:10
Bad jobs:4
Total threads:125
```

[5]There is no limit on the number of client handling threads – just computing threads.

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

Computational Thread Limiting (Advanced)

If the `maxthreads` feature is implemented and the command line argument is provided, then `intserver` must not launch more than the specified number of computation threads (not including the 'main' (listening) thread, and individual client handling threads). `intserver` must maintain a thread count, and individual client handling threads must block and not launch a new worker thread until that count is less than the maximum. The `maxthreads` limit is an overall limit, not a per-client limit. See the Hints section below for advice.

Program output

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

Note on floating point precision

Consider the following regarding floating point precision and related matters:

- All floating point variables in your programs are to be of type `double`.
- All floating point output to `stdout`, `stderr`, and in network requests/responses are to be output using the '`%lf`' `printf()` format specifier.
- *Where appropriate*, the marking scripts will allow for a *small* margin of error in floating point precision, however if you implement your integration as described in this document there should be no floating point accuracy issues.
- Floating point variables can take on special values under certain error conditions, such as `inf` ('infinity' e.g. 1/0) and `NaN` (not a number, e.g. `sqrt(-1)`). You do not need to check for, or do any special handling for these error conditions. The test cases will not yield such results, so if you start seeing these values in your output then it is a sign you are doing something wrong!

Communication protocol

The communication protocol uses HTTP. The client (`intclient`) shall send HTTP requests to the server, as described below, and the server (`intserver`) shall send HTTP responses as described below. The connection between client and server is kept alive between requests. All requests are `GET` requests. 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 `intclient` and `intserver` is *synchronous* - a single client can only have a single request underway at any one time. This greatly simplifies the implementation of `intclient` and the `intserver` client-handling threads.

- Request: `GET /validate/`*`expression`* `HTTP/1.1`
  - Description: the client is requesting the server to check whether the given *`expression`* is a valid function of the single variable `x`. (Note that the expression may contain '`/`' characters but may not contain spaces.)
  - Request Headers: none expected, all headers present will be ignored by `intserver`.
  - Response Status: either 200 (OK) or 400 (Bad Request). 200 will be returned if the expression can be validly compiled – see below, otherwise 400 will be returned.
  - Response Headers: `Content-Length: 0` is expected, other headers are optional.
  - Response Data: none.
- Request: `GET /integrate/`*`lower/upper/segments/threads/expression`* `HTTP/1.1`
  - Description: the client is requesting the server to perform the specified integration. The fields have the same meaning as described earlier in this document. (Note that the expression may contain `/` characters but may not contain spaces.)
  - Request Headers: the header field "`X-Verbose: yes`" may be present to indicate that `intserver` should return verbose job details. Any other headers will be ignored by `intserver`.

  - Response Status: either 200 (OK) or 400 (Bad Request). 200 will be returned if the integration proceeds and a result is returned, otherwise 400 will be returned.
  - Response Headers: the `Content-Length` header is expected (the value will be the number of bytes in the response data), other headers are optional.
  - Response Data: If the "`X-Verbose`" header is present in the request with value "`yes`" then `intserver` shall return details of the partial integration results – in exactly the format described earlier in this document (i.e. one line for each computation thread where thread numbers increase linearly from 1, and the range of x values of each sub-integral are similarly ordered from lowest to highest.). This will be followed by one line containing the integration result – printed using the '`%lf`' specifier for `printf()`. If verbose output is not requested, the response data shall just consist of the one line with the integration result. All lines in the response data are to be terminated by newlines.

- Any other (well-formed) requests should result in `intserver` sending a 400 (Bad Request) HTTP response. Badly formed requests (i.e. the data received can not be parsed as a HTTP request) will result in the server disconnecting the client (as described earlier).

Note that library functions are provided to you to do most of the work of parsing/constructing HTTP requests/responses. See below.

Provided Libraries

libtinyexpr

The `tinyexpr` library has been pre-compiled into a shared library for you, providing all of the functions and data types required.

These functions are declared in `/local/courses/csse2310/include/tinyexpr.h` on moss. To use them, you will need to add `#include <tinyexpr.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 -ltinyexpr`

Note that the `tinyexpr` library uses the C floating point library, so you will also need to link your programs with the `-lm` option.

Important note: **You must use the version of `tinyexpr.h` and the pre-compiled library provided for you on `moss`** – do not download and use the `tinyexpr.c` and `tinyexpr.h` from Github. The original version will generate style warnings which will cost you marks, and we will be instrumenting the `libtinyexpr` to support the marking process. **Failure to use the provided version will cause your marking tests to fail**.

See the Using the `tinyexpr` library section below for more details.

libcsse2310a4

Several library functions have been provided to you to aid parsing/construction of HTTP requests/responses. These are

- `char** split_by_char(char* str, char split, unsigned int maxFields);`
- `int parse_HTTP_request(void* buffer, int bufferLen, char** method, char** address, char*** headers, char** body);`
- `char* construct_HTTP_response(int status, char* statusExplanation, char** headers, char* body);`
- `int parse_HTTP_response(void* buffer, int bufferLen, int* status, char** statusExplanation, char*** headers, char** body);`

These functions are declared in `/local/courses/csse2310/include/csse2310a4.h` on moss and their behaviour is described in man pages on moss, e.g. run `man split_by_char`. 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 `libtinyexpr` and `libcsse2310a4`.)

libcsse2310a3

You are also welcome to use the `"libcsse2310a3"` library from Assignment 3 if you wish.

Using the `tinyexpr` library

`tinyexpr` is a small C library for parsing and evaluating mathematical expressions. For example, given the string `"sqrt(sin(x)+1)"`, using `tinyexpr` you can evaluate that function using C calls. This will give your programs the ability to work with arbitrary mathematical functions when computing integrals.

There a few steps to using `tinyexpr`, as illustrated below

Listing 6: `tinyexpr` usage example

```c
#include <tinyexpr.h>

...

double x;           // The C variable that will be passed to the expression evaluation

// Create an expression variable called 'x', pointing to x
te_variable vars[] = {{"x", &x}};

int err_pos;

// Parse and compile the expression, passing the vars and var count (1), and ptr to an error return
te_expr* expr = te_compile("sqrt(sin(x)+1)", vars, 1, &err_pos);

// expr is non NULL if compilation succeeded
if(expr) {
    // Loop from zero to 10 in increments of 0.5, evaluating the function
    for(x=0;x<=10; x+=0.5) {
        printf("f(%lf) = %lf\n", x, te_eval(expr));
    }

    // Free the memory associated with the expression
    te_free(expr);
} else {
    printf("Error parsing expression at character %i\n", err_pos);
}
```

A few things to note from the above example:

- You must declare and bind the `te_variable` first
- After variable binding, `te_compile` is used to compile the expression string for later evaluation
- If compilation fails, the `err_pos` variable will be updated to the character number in the string where the parse error occurred
- After parsing and compilation, updates to the orginal variable (`x` in this example) will be reflected on each call to `te_eval()`
- After you are finished with an expression, release the memory by calling `te_free()`

This little introduction should be enough for the use of `tinyexpr` required in this assignment, however you may also refer to the source code and more complex examples at the project Github repository if you wish: `https://github.com/codeplea/tinyexpr`

Style

Your program must follow version 2.1.1 of the CSSE2310/CSSE7231 C programming style guide available on the course BlackBoard site.

Hints

1. Review the lectures related to network clients (lecture 14), HTTP (lecture 15), threads and synchronisation (lectures 16 and 17) and multi-threaded network servers (lecture 18). This assignment builds on all of these concepts.
2. Play with the `tinyexpr` test program to get familiar with how to use this library.
3. Write a small program to explore the integral computation before you worry about threads and network sockets – just get the basic computation working. The heart of this program will later become the body of the computation threads.
4. You can test `intclient` and `intserver` independently using `netcat` as demonstrated in the lectures.
5. The multithreaded network server example from the lectures can form the basis of `intserver`.
6. Once you know how to compute integrals (single-threaded), and have a basic threaded server working, you can combine these two concepts to implement a multi-client, single-threaded integration server. Then you can work on making the computation multi-threaded.
7. Because both `intclient` and `intserver` error check the expressions and integral parameters, you should never see a 400 (Bad Request) response to an integrate request in normal program operation (`intclient` will detect the bad job parameters and not send the request). However, you still need to implement and test for bad requests (e.g. test using `netcat`).
8. Use the provided library functions (see above).
9. For computational thread limiting you will need to use some sort of locking or mutual exclusion mechanism to ensure that all client handler threads correctly access and update the thread count. Consider using a semaphore with an initial value of `maxthreads` to manage thread resources.
10. Consider the '`%n`' specifier to `sscanf` for checking whether surplus characters remain in a field.
11. 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()`, `dup()/dup2()` etc

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 (jobspec) files.

Your programs (named `intclient` and `intserver`) must build on `moss.labs.eait.uq.edu.au` with:
`make`

If you only implement one of the programs then it is acceptable for `make` to just build that one program – and we will only test that one program.

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 the one 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)[6]. 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.

If you make a submission prior to the deadline[7] then your final submission prior to the deadline will be the submission that we mark – i.e. your last "on-time" submission will be marked. If your first submission is after the deadline (i.e. your submission is late) then that first submission will be the submission that we mark. A late penalty will apply in this case – 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

[6] 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.

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

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 (standard out 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).

Marks will be assigned in the following categories. There are 20 marks for `intclient` and 40 marks for `intserver`.

1. `intclient` correctly handles invalid command lines (2 marks)
2. `intclient` connects to server and also handles inability to connect to server (2 marks)
3. `intclient` correctly detects errors in job files (other than bad expression errors) (4 marks)
4. `intclient` correctly sends validation requests and handles responses (4 marks)
5. `intclient` correctly sends integration requests (both verbose and not) (3 marks)
6. `intclient` correctly report integral job results (including verbose results) (3 marks)
7. `intclient` correctly handles communication failure (2 marks)

8. `intserver` correctly handles invalid command lines (2 marks)
9. `intserver` correctly listens for connections and reports the port (3 marks)
10. `intserver` correctly handles invalid and badly formed HTTP requests (3 marks)
11. `intserver` correctly handles and responds to expression validation requests (5 marks)
12. `intserver` correctly handles multiple simultaneous client connections (3 marks)
13. `intserver` correctly computes integrals and returns results (5 marks)
14. `intserver` correctly implements verbose integration result reporting (5 marks)
15. `intserver` correctly handles disconnecting clients and communication failure (3 marks)
16. `intserver` correctly implements multithreaded integration (4 marks)
17. `intserver` correctly implements computational thread limiting (4 marks)
18. `intserver` correctly implements SIGHUP statistics reporting (3 marks)

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.1.1 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[8].

[8] 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[9].

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)

- V will be the number of **additional** style violations detected by human markers. Violations will not be counted twice (e.g. if a poorly named variable is used multiple times it will count as a single violation).

Your human style mark H will be

$$H = 5 - V$$

If $V \geq 5$ then H will be zero (0) – no negative marks will be awarded. If your code has many style violations, human style marking may be incomplete, e.g. the marker may stop after reaching 5 violations.

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

Documentation (10 marks) – for CSSE7231 students only

CSSE7231 students must submit a PDF document containing a written overview of the architecture and design of your program. This must be submitted via the Turnitin submission link on Blackboard.

Please refer to the grading criteria available on BlackBoard under "Assessment" for a detailed breakdown of how these submissions will be marked. Note that your submission time for the whole assignment will be considered to be the later of your submission times for your zip file and your PDF design document. Any late penalty will be based on this submission time and apply to your whole assignment mark.
This document should describe, at a general level, the functional decomposition of the program, the key design decisions you made and why you made them. It must meet the following formatting requirements:

- Maximum two A4 pages in 12 point font
- Diagrams are permitted up to 25% of the page area. The diagram(s) must be discussed in the text, it is not ok to just include a figure without explanatory discussion.

[9] 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.

Don't overthink this! The purpose is to demonstrate that you can communicate important design decisions, and write in a meaningful way about your code. To be clear, this document is not a restatement of the program specification – it is a discussion of your design and your code.

If your documentation obviously does not match your code, you will get zero for this component, and will be asked to explain why.

Total mark

Let

- F be the functionality mark for your assignment (out of 60).
- 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).
- D be the documentation mark for your assignment (out of 10 for CSSE7231 students) – or 0 for CSSE2310 students.

Your total mark for the assignment will be:

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

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@helpdesk.eait.uq.edu.au`.