The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 1, 2023
Assignment 4 (version 1.2)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 15%
Due: 4:00pm Friday 26 May, 2023

Specification changes since version 1.0 are shown in red (and since 1.1 are shown in blue) 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 brute-force password cracking system. One program – `crackserver` – is a network server which accepts connections from clients (including `crackclient` which you will implement). Clients connect, and provide encrypted passphrases that the server will attempt to crack (recover the original unencrypted passphrase). Clients may also request the server to encrypt passwords for later analysis. Communication between the `crackclient` and `crackserver` 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 `crackserver`.

The assignment will also test your ability to code to a particular programming style guide 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. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository until at least three months after the result release date for the course (or check with the course coordinator if you wish to post it sooner). Do not allow others to access your computer – you must keep your code secure. Never leave your work unattended.

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, found in `/local/courses/csse2310/resources` on `moss`, posted on the discussion forum by teaching staff, provided in Ed Lessons, or shown in class).	May be used freely without reference. (You must be able to point to the source if queried about it – so you may find it easier to reference the code.)
Code you have personally written this semester for CSSE2310/7231 (e.g. code written for A1 reused in A3) – provided you have not shared or published it.	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 you understand the code AND the source of the code is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.
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[1].	

[1] Code you have *taken inspiration from* must not be directly copied or just converted from one programming language to another.

Code Origin	Usage/Referencing
Code written by or obtained from, or based on code written by or obtained from, a code generation tool (including any artificial intelligence tool) that you personally have interacted with, without the assistance of another person.	May be used provided you understand that code AND the source of the code is referenced in a comment adjacent to that code (in the required format) AND an ASCII text file (named `toolHistory.txt`) is included in your repository and with your submission that describes in detail how the tool was used. If such code is used without appropriate referencing and without inclusion of the `toolHistory.txt` file then this will be considered misconduct.
Other code – includes (but may not be limited to): code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution); code from public or private repositories; code from websites; code from textbooks; any code written or partially written or provided by or written with the assistance of 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 teaching staff will conduct interviews with a subset of 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 legitimately use code from other sources (following the usage/referencing requirements in the table above) then you are expected to understand that code. If you are not able to adequately explain the design of your solution and/or adequately explain your submitted code (and/or earlier versions in your repository) 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 and/or your submission may be subject to a misconduct investigation where your interview responses form part of the evidence. Failure to attend a scheduled interview will result in zero marks for the assignment. The use of referenced code in your submission (particularly from artificial intelligence tools) is likely to increase the probability of your selection for an interview.

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. Don't help another CSSE2310/7231 student with their code no matter how desperate they may be and no matter how close your relationship. You should read and understand the statements on student misconduct in the course profile and on the school website: `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`.

Simple encryption and salting

Hashing is a common method of protecting sensitive data such as passwords. Instead of storing or transmitting the password itself (*plaintext*) where it is at risk of interception, the password is first transformed by passing it through a one-way function called a *hash*, yielding the *ciphertext*. A one-way function is one that is easy to apply in one direction (plaintext to ciphertext), but impossible to apply in the reverse.

Because hash functions are deterministic, identical passwords encrypted with a hash function yield identical ciphertext, which can assist an adversary in compromising a system. For this reason, the hashing scheme is usually extended with a method called *salting*. The plaintext is extended with a random value, or `salt`, prior to applying the hash function. The salt is stored along with the encrypted password, as both are required to verify a given password and its hash.

With a sufficiently large possible range of salt values, this helps prevent attackers from building tables that store the hash results of every possible (salted) password, increasing security by making the cracking more challenging.

`libcrypt` is a POSIX library that supports a wide range of hashing functions and salting schemes – in this assignment you will use it in its simplest mode. Note that this mode is now considered obsolete, and should not be used for protecting data in modern systems.

The `crypt()` function is all that is required. In the simplest mode it is used as follows:

```
char *cipher = crypt(plaintext, salt)
```

where `plaintext` is a pointer to a string containing the plaintext password, and `salt` is a pointer to a string containing the salt. Only the first 8 characters of `plaintext` are used, and only the first two characters of `salt` are used. Any additional characters are ignored.

Additionally, the salt must only contain characters from the following character set:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 . /
```

The return value from `crypt()` is a pointer to a static buffer, so the caller must copy this before making any further calls to `crypt()`. A reentrant version of `crypt()` called `crypt_r()` is also available – refer to the `man` page for details and usage.

Here are some examples:

- plaintext `"foobar"`, salt `"AA"` → ciphertext `"AAZk9Aj5/Ue0E"`
- plaintext `"dinosaur"`, salt `"0z"` → ciphertext `"0zD1fV.Yez8RI"`

Notice how the supplied salt characters always form the first two characters of the encrypted string.

The `crypt()` family of functions is declared in `<crypt.h>`, and you will need to link your `crackserver` with the `-lcrypt` argument to use them in your programs.

Specification – `crackclient`

The `crackclient` program provides a command line interface that allows you to interact with `crackserver` as a client – connecting to the server, sending crack and encryption requests, and displaying the results.

To implement this functionality, `crackclient` will only require a single thread.

Command Line Arguments

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

```
./crackclient portnum [jobfile]
```

- The mandatory `portnum` argument indicates which localhost port `crackserver` is listening on – either numerical or the name of a service.
- The optional `jobfile` argument specifies the name of a file to be used as input commands. If not specified, then `crackclient` is to read input from `stdin`.

`crackclient` behaviour

If an incorrect number of command line arguments are provided then `crackclient` should emit the following message (terminated by a newline) to `stderr` and exit with status 1:

```
Usage: crackclient portnum [jobfile]
```

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

Job file error

If a job file is specified, and `crackclient` is unable to open it for reading, `crackclient` shall emit the following message (terminated by newline) to `stderr` and exit with status 2:

```
crackclient: unable to open job file "jobfile"
```

where `jobfile` is replaced by the name of the specified file. Note that the file name is enclosed in double quote characters.

Connection error

If `crackclient` 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:

```
crackclient: 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.)

`crackclient` runtime behaviour

Assuming that the commandline arguments are without errors, `crackclient` 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;
- read commands either from the jobfile, or `stdin`, and process them as described below; and
- when `EOF` is received on the input stream (job file or `stdin`), `crackclient` shall close any open network connections, and terminate with exit status 0.

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

```
crackclient: server connection terminated
```

Note that `crackclient` need only detect EOF on the socket when it is reading from the socket. It will only do this after it has read a line from the jobfile (or `stdin`), sent that to the server, and is then awaiting a response. It is not expected that `crackclient` will detect EOF on the network socket while it is waiting to read from `stdin`.

`crackclient` shall interpret its input commands as follows:

- Blank lines (i.e. those lines containing no characters), and those beginning with the `#` character (comment lines), shall be ignored
- All other lines shall be sent unaltered to the server (no error checking is required or to be performed)

Upon sending a command to the server, `crackclient` shall wait for a single line reply, and interpret it as follows:

- Response `":invalid"` → emit the following text to `stdout`:

  ```
  Error in command
  ```

- Response `":failed"` → emit the following text to `stdout`:

  ```
  Unable to decrypt
  ```

- Otherwise, the raw output received from the server shall be output to `stdout`.

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

`crackclient` example usage

The following is an example of an interactive session with `crackclient` (assuming the `crackserver` 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:

```
$ ./crackclient 49152
# A comment line - ignored
# A blank line - ignored

# A malformed crypt request (no salt)
crypt chicken
Error in command
```

```
# A malformed crypt request (bad salt)
crypt chicken %%
Error in command
# Valid crypt requests
crypt dog K9
K930xTkdzhuMg
crypt foobar zZ
zZT8RjNYjMLz2
# Invalid crack request - no thread count specified
crack zZT8RjNYjMLz2
Error in command
# Failed crack request - the plaintext is not in the dictionary
crack zZT8RjNYjMLz2 10
Unable to decrypt
# Successful crack request, 1 thread
crack K930xTkdzhuMg 1
dog
# Successful crack request, 10 threads
crack K930xTkdzhuMg 10
dog
# Misc invalid crack requests
crack K930xTkdzhuMg 0
Error in command
crack K930xTkdzhuMg -1
Error in command
crack K930xTkdzhuMg 100
Error in command
crack K930xTkdzhuMg 1X
Error in command
```

Note that not all possible error conditions are present in this example. Note also that the `crackserver` for this example (and the next) was not using the default dictionary. The plaintext `foobar` can be found in the default dictionary but was not in the dictionary used in these examples.

The following is an example of a file driven session with `crackclient` (assuming the `crackserver` 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, given that `cmd.in` has the following contents:

```
# this is cmd.in - a list of commands to input to crackclient
# A comment line - ignored
# A blank line - ignored

# A malformed crypt request (no salt)
crypt chicken
# A malformed crypt request (bad salt)
crypt chicken \%\%
# Valid crypt requests
crypt dog K9
crypt foobar zZ
# Invalid crack request - no thread count specified
crack zZT8RjNYjMLz2
# Failed crack request - the plaintext is not in the dictionary
crack zZT8RjNYjMLz2 10
# Successful crack request, 1 thread
crack K930xTkdzhuMg 1
# Successful crack request, 10 threads
crack K930xTkdzhuMg 10
# Misc invalid crack requests
crack K930xTkdzhuMg 0
crack K930xTkdzhuMg -1
```

```
crack K930xTkdzhuMg 100
crack K930xTkdzhuMg 1X
```

```
$ ./crackclient 49152 cmd.in
Error in command
Error in command
K930xTkdzhuMg
zZT8RjNYjMLz2
Error in command
Unable to decrypt
dog
dog
Error in command
Error in command
Error in command
Error in command
```

Specification – `crackserver`

`crackserver` is a networked, multithreaded password cracking server, allowing clients to connect and provide encrypted ciphertext for cracking, and also allows clients to provide plaintext passwords for encrypting. 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 `crackserver` program is to accept command line arguments as follows:

`./crackserver [--maxconn connections] [--port portnum] [--dictionary filename]`

In other words, your program should accept up to three optional arguments (with associated values) – which can be in any order.

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

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).

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

The dictionary `filename` argument, if specified, indicates the path to a plain text file containing one word or string per line, which represents the dictionary that `crackserver` will search when attempting to crack passwords. If not specified, `crackserver` shall use the system dictionary file `/usr/share/dict/words`.

Program Operation

The `crackserver` program is to operate as follows:

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

  ```
  Usage: crackserver [--maxconn connections] [--port portnum] [--dictionary filename]
  ```

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

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

  - any of `--maxconn`, `--port` or `--dict` does not have an associated value argument
  - the maximum connections argument (if present) 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

  - any of the arguments is specified more than once
  - any additional arguments are supplied

- If the dictionary `filename` argument refers to a file that does not exist or cannot be opened for reading, `crackserver` shall emit the following error message to `stderr` and terminate with exit status 2:

```
crackserver: unable to open dictionary file "filename"
```

  where `filename` is replaced by the name of the dictionary file provided on the command line. Note that the double quote characters must be present.

- The dictionary words must be read into memory. Lines in the dictionary are terminated by newline characters (except possibly the last line) and each line (excluding that newline character) is considered to be a word. Any words longer than 8 characters should be discarded, i.e. not saved into memory, as the `crypt()` family of functions only considers at most 8 characters of any supplied plain text. The order of words must be preserved. It is possible the dictionary may contain duplicate words and these should be preserved also. You may assume that words in the dictionary are no longer than 50 characters. Your `crackserver` must read the dictionary only once.

- If the dictionary contains no words that are 8 characters long or shorter, then `crackserver` shall emit the following error message to `stderr` and terminate with exit status 3:

```
crackserver: no plain text words to test
```

- If `portnum` is missing or zero, then `crackserver` shall attempt to open an ephemeral localhost port for listening. Otherwise, it shall attempt to open the specified port number. If `crackserver` is unable to listen on either the ephemeral or specified port, it shall emit the following message to `stderr` and terminate with exit status 4:

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

- Once the port is opened for listening, `crackserver` 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, `crackserver` shall spawn a new thread to handle that client (see below for client thread handling).

- If specified (and implemented), `crackserver` 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 above must be terminated by a single newline character.

- The `crackserver` program should not terminate under normal circumstances, nor should it block or otherwise attempt to handle `SIGINT`.

- Note that your `crackserver` must be able to deal with any clients using the correct communication protocol, not just `crackclient`. 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 `crack` requests to `crackserver`, ~~it~~ the client thread shall spawn threads to perform the brute-force password cracking action against the dictionary. The number of cracking threads spawned per `crack` request will be specified as part of the request. Even if only one cracking thread is requested, the client thread must spawn an additional thread to do the cracking.

`crypt` requests from the client ~~may~~ must be handled directly in the client handling thread ~~if you wish~~ – it is not computationally expensive and there is no need to spawn an additional thread for this operation.

Due to the simultaneous nature of the multiple client connections, your `crackserver` 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 `crackserver` program itself must continue uninterrupted.

Password cracking

`crackserver` is to use a brute-force method for cracking passwords as follows:

For each received ciphertext password

- Extract the password salt (the first two characters of the ciphertext)
- For each word saved from the dictionary
  - encrypt the word with the salt using `crypt()` or `crypt_r()`
  - compare the sample ciphertext with the encrypted dictionary word
  - if the two ciphertexts are the same, terminate the search and return the plaintext dictionary word

Note – with just over 200,000 words of the right length in the default dictionary on `moss` and 64^2 possible salt strings, it is not feasible to pre-calculate and store all possible salt+word combinations. Your program is expected to use the brute-force method described above, trading CPU work for memory storage.

The above brute-force algorithm is trivially parallelised across multiple threads. If you implement it, and the client requests more than one thread be used for cracking, you must distribute the dictionary roughly equally across all threads as described below:

If the dictionary contains D words and N threads are requested then:

- if $D < N$ or $N = 1$ then only one thread must be used
- if $D \geq N$ and $N \geq 2$ then $N - 1$ threads must each cover $\lfloor D/N \rfloor$ words (i.e. D divided by N rounded down if necessary to the nearest integer) and the remaining thread must cover the remaining words

Words must be allocated in groups based on the order they are present in the dictionary, i.e., the first $\lfloor D/N \rfloor$ words saved from the dictionary must be allocated to one thread, the next $\lfloor D/N \rfloor$ words to another thread, etc.

For example, if there are 100,000 words saved from the dictionary spread across 7 threads – one thread will get the first $\lfloor 100000/7 \rfloor = 14285$ words, the next thread will be allocated the next 14285 words, etc., and the last (seventh) thread gets the last 14290 words saved from the dictionary.

Each thread must check its allocated words in the same order as they are present in the dictionary.

If a match is found then the thread that found the match must (a) indicate to the other threads in the job that they should stop work immediately, and (b) not check any further of its own words. You must not use `pthread_cancel()` to terminate other threads[2]. It is recommended that you set a (volatile) flag variable that is checked in other threads before each word is processed.

Note that even though `/usr/share/dict/words` is sorted, the supplied dictionary may not be sorted.

Note also that it is possible that multiple threads may find matches because the same word may be present multiple times in the dictionary.

`SIGHUP` handling (Advanced)

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

- Total number of clients connected (at this instant)
- The total number of clients that have connected and disconnected since program start
- The total number of `crack` requests received (since program start), including invalid requests
- The total number of valid but unsuccessful `crack` requests completed (since program start)
- The total number of valid but successful `crack` requests completed (since program start)
- The total number of `crypt` requests received (since program start), including invalid requests

[2] As discussed in lectures, `pthread_cancel()` is not reliable.

- The total number of calls to `crypt()` and/or `crypt_r()` (since program start) – including for both `crack` and `crypt` requests[3]. This count does not have to be updated on a call-by-call basis as this may impact performance. It must be updated before any response to a `crypt` or `crack` request is sent.

Unsuccessful `crack` requests include those that fail to find a matching dictionary word. Successful crack requests are those that are valid and for which a matching dictionary word is found. Invalid `crack` requests are those that have the correct command word (`crack`) but are otherwise invalid (e.g. have invalid arguments). Note also that the number of `crack` requests received may include requests that are still in progress – i.e. it is not yet known whether the request is successful or unsuccessful. (See the Communication protocol section for details.)

All `crypt` requests should be counted, including those that are invalid e.g. due to an invalid salt string.

The required format is illustrated below. Each of the six lines is terminated by a single newline. You can assume that all numbers will fit in a 32-bit unsigned integer, i.e. you do not have to consider numbers larger than 4,294,967,295.

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

```
Connected clients: 4
Completed clients: 20
Crack requests: 37
Failed crack requests: 15
Successful crack requests: 19
Crypt requests: 34
crypt()/crypt_r() calls: 34873425
```

Note that to accurately gather these statistics and avoid race conditions, you will need some sort of mutual exclusion 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 `crackserver` must not permit more than that number of simultaneous client connections to the server. `crackserver` 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 `crackserver` implementation. Upon startup, `crackserver` shall check the value of the environment variable `A4_HTTP_PORT`. If set, then `crackserver` shall also listen for connections on that port number (or service name).

If `crackserver` is unable to listen on that port or service name then it shall emit the following message to `stderr` and terminate with exit status 5:

```
crackserver: 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 `crackserver` 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 `crackserver` 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 `crackserver` is *synchronous* – any one client can only have a single request underway at any one time. This greatly simplifies the implementation of the `crackserver` HTTP connection handling threads.

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

[3]Note that this value will not be predictable if your tests include multi-threaded `crack` requests that are successful. This is because you can't predict where other threads are up to when they stop work after a match is found.

- Request: `GET /stats HTTP/1.1`
  - Description: the client is requesting statistics from `crackserver`
  - Request
    - Request Headers: none expected, any headers present will be ignored by `crackserver`.
    - 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 `crackserver` statistics information described in the `SIGHUP` handling section above.

If `crackserver` 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, `crackserver` should handle this gracefully and terminate the thread associated with the connection. Note that HTTP clients are NOT included in the client count statistics.

Program output

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

Communication protocol

The communication protocol between clients and `crackserver` 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 `crackclient` to `crackserver` are:

- `crack <ciphertext> <crackthreads>` – the client is requesting the string `<ciphertext>` be cracked, using the specified number of threads.

  IMPORTANT: even if you do not implement multi-threaded parallel password cracking functionality, your `crackserver` must still accept and parse/validate the `<crackthreads>` field - it will simply be ignored during the cracking process.

- `crypt <string> <salt>` – the client is requesting the server to encrypt the supplied `<string>` using the specified `<salt>`.

Single spaces are used as separators between fields, which implies that `<ciphertext>` and `<string>` fields must not contain spaces.

Supported messages from `crackserver` to `crackclient` are

- `:invalid` – sent by the server to a client if the server receives an invalid command from that client (see below)
- `:failed` – sent by the server to a client if the server is unable to crack the supplied ciphertext
- `<plaintext>` – if the server is able to decrypt the ciphertext, then the decrypted plaintext is sent back on a single line.
- `<ciphertext>` – the ciphertext result of a `crypt` operation is sent back on a single line

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

- Invalid command word – an empty string or a command word which is not `crack` or `crypt`
- Invalid arguments such as

  - empty strings (for any field)
  - ciphertext not exactly 13 characters in length (including the two salt characters at the beginning)
  - missing or invalid integer for the `<threads>` argument. **The number of threads requested must be greater than or equal to 1, and less than or equal to 50.**

- Invalid salt string provided for the `crypt` command (e.g. not two characters or uses invalid characters)
- Invalid salt substring in the ciphertext for the `crack` command.
- Too few or too many arguments
- Any other kind of malformed message

Provided Libraries

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`

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 `crackserver`.

Style

Your program must follow version 2.3 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site.

Hints

1. Review the lectures related to network clients, threads and synchronisation and multi-threaded network servers (and HTTP for CSSE7231 students). This assignment builds on all of these concepts.
2. You should test `crackclient` and `crackserver` independently using `netcat` as demonstrated in the lectures. You can also use the provided demo programs `demo-crackclient` and `demo-crackserver`.
3. The `read_line()` function from `libcsse2310a3` may be useful in both `crackclient` and `crackserver`.
4. The multithreaded network server example from the lectures can form the basis of `crackserver`.
5. Use the provided library functions (see above).
6. 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. Be sure to properly reference any code samples or inspiration you find online or via generative AI such as ChatGPT.

Possible Approach

1. Try implementing `crackclient` first. (The programs are independent so this is not a requirement, but when you test it with `demo-crackserver` it may give you a better understanding of how `crackserver` works.)
2. For `crackserver`, try writing a simple non-networked cracking program first to ensure you understand the algorithm and use of `libcrypt`.
3. Once you can do single threaded cracking, work out how to split it across multiple threads.
4. Finally, integrate your cracking algorithm into the multi-threaded network server.

Forbidden Functions

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

- `goto`
- `#pragma`
- `gcc` attributes

You must not use any of the following C functions. If you do so, you will get zero (0) marks for any test case that calls the function.

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

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.

Your programs (named `crackclient` and `crackserver`) 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 program – and we will only test that 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 or other methods 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-sem1-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)[4]. 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[5] will incur a late penalty – see the CSSE2310/7231 course profile for details.

Note that Gradescope will run the test suite immediately after you submit. When complete[6] you will be able to see the results of the "public" tests.

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.

[4] 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.

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

[6] Gradescope marking may take only a few minutes or more than 30 minutes depending on the functionality/efficiency of your code.

Functionality (60 marks CSSE2310/ 70 marks CSSE7231)

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`) and communication messages is used for functionality marking. Strict adherence to the formats 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 `crackclient` and 50 marks (CSSE2310) or 60 marks (CSSE7231) for `crackserver` .

1. `crackclient` correctly handles invalid command lines and inability to read from job files (2 marks)
2. `crackclient` connects to server and also handles inability to connect to server (2 marks)
3. `crackclient` correctly handles input from `stdin` or a job file (i.e. sends appropriate lines to server) (2 marks)
4. `crackclient` correctly displays lines received from the server ~~and sends input lines to the server~~ (2 marks)
5. `crackclient` correctly handles communication failure and `EOF` on input (stdin or job file) (2 marks)

6. `crackserver` correctly handles invalid command lines (3 marks)
7. `crackserver` correctly handles unopenable dictionaries and dictionaries with no valid words (2 marks)
8. `crackserver` correctly listens for connections and reports the port (including inability to listen for connections) (~~3~~ 2 marks)
9. `crackserver` correctly handles invalid command messages (5 marks)
10. `crackserver` correctly handles single-threaded `crack` requests (with at most one client connected at a time) (5 marks)
11. `crackserver` correctly handles multi-threaded `crack` requests (with at most one client connected at a time) (5 marks)
12. `crackserver` correctly handles `crypt` requests (with at most one client connected at a time) (4 marks)
13. `crackserver` correctly handles multiple simultaneous client connections (using threads) (~~8~~ 7 marks)
14. `crackserver` correctly handles disconnecting clients and communication failure (3 marks)
15. `crackserver` correctly implements client connection limiting (4 marks)
16. `crackserver` correctly implements early termination of threads when a match is found (3 marks)
17. `crackserver` correctly implements SIGHUP statistics reporting (including protecting data structures with mutexes or semaphores) (7 marks)

18. (CSSE7231 only) `crackserver` correctly listens on the port specified by `A4_HTTP_PORT` (and handles inability to listen or environment variable not set) (3 marks)

19. (CSSE7231 only) `crackserver` correctly responds to HTTP requests (including invalid requests) from a single client issuing one request per connection (4 marks)

20. (CSSE7231 only) `crackserver` supports multiple simultaneous HTTP clients and multiple sequential requests over each connection (3 marks)

Some functionality may be assessed in multiple categories. The ability to support multiple simultaneous clients will be covered in multiple categories (12 to 16).

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.3 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[7].

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[8].

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`. You can check the result of Gradescope style marking soon after your Gradescope submission – when its test suite completes running.

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.

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

[8] 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.

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

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 Marking (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 – only one or two commits OR all commit messages are meaningless.
1	Some progressive development evident (three or more commits) AND at least one commit message is meaningful.
2	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful.
3	Multiple commits that show progressive development of ALL functionality (e.g. no large commits with multiple features in them) AND at least half the commit messages are meaningful.
4	Multiple commits that show progressive development of ALL functionality AND meaningful messages for all but one or two of the 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 students or out of 70 for CSSE7231 students).
- 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).
- V is the scaling factor (0 to 1) determined after interview(s) (if applicable – see the Student Conduct section above) – or 0 if you fail to attend a scheduled interview without having evidence of exceptional circumstances impacting your ability to attend.

Your total mark for the assignment will be:

$$M = (F + \min\{F, S + H\} + \min\{F, C\}) \times V$$

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

In other words, you can't get more marks for style or SVN commit history 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

- Clarified expected update rate of `crypt()` call count.
- Added further examples of invalid arguments to `crackserver` commands.
- Made it clear that even if only one cracking thread is requested then an additional thread must still be created by the client thread.
- Clarified that a missing `--maxconn` argument means no limit on client connections. «««¡ HEAD
- Updated `crackclient` marking criteria to include inability to read from job files.
- Clarified that `crackclient` is only expected to detect EOF on the network socket when it is reading from the network socket.
- Clarified meaning of "blank lines" in `crackclient` input.
- Clarified that marking category 5 includes handling EOF on the source of input (rather than `stdin` specifically).

Version 1.2

- Clarified `crackclient` marking criteria about handling input.
- Clarified argument count for `crackserver`.
- Added marking category for `crackserver` dictionary errors (and renumbered subsequent marking categories and adjusted allocated marks).
- Added requirement that `crypt` requests must be handled in the client thread – no additional thread is to be created for this.
- (CSSE7231) Clarified that HTTP clients are not included in statistics.
- Clarified that the `crackclient` examples were not using a server with the default dictionary.