The University of Queensland
School of Electrical Engineering and Computer Science

CSSE2310/CSSE7231 — Semester 2, 2023
Assignment 4 (version 1.1 - 13 October 2023)

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

This specification was created for the use of Adnaan BUKSH (s4743556) only.
Do not share this document. Sharing this document may result in a misconduct penalty.
Specification changes since version 1.0 are shown in red and are summarised at the end of the document.

Introduction

The goal of this assignment is to further develop your C programming skills, and to demonstrate your understanding of networking and multithreaded programming. You are to create two programs which together implement a real-time, online auction platform. The main program – `auctioneer` – is a network server which accepts connections from clients (including `auctionclient` which you will implement). Clients connect, and interact with `auctioneer` in various ways to sell and buy items.

Communication between the clients and `auctioneer` 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 `auctioneer`.

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 section is unchanged from assignments one and three – but you should remind yourself of the referencing requirements. Remember that you can't copy code from websites and if you learn about how to use a library function from a website then you must reference it.

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 Origin	Usage/Referencing
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 learned from or that you have taken inspiration from but have not copied[1].	
Code written by, modified by, or obtained from, or code based on code written by, modified 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. The file must be committed to the repository at the same time as any code derived from such a tool. 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 or based on 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.

You must not share this assignment specification with any person, organisation, website, etc. 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 many of these sites and many cooperate with us in misconduct investigations. You are permitted to post small extracts of this document to the course Ed Discussion forum for the purposes of seeking or providing clarification on this specification.

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://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct`.

Online auctions

In a simple auction, an *item* is put up for sale by a *seller*. One or more *buyers* then place *bids* – statements describing how much they are willing to pay for the item. Each bid must be higher than the previous. Buyers may place multiple bids, so long as each is higher than the current maximum bid. Once a buyer has set the highest bid, they may not bid again until a different buyer places a higher bid on the item. At the end of the auction, the highest bidder is the winner.

[1] Code from elsewhere must not be directly copied or just converted from one programming language to another.

The *seller* sets a *reserve* price – the lowest amount they are willing to sell the item for. If the auction does not reach the reserve price, the auction ends with the item unsold. The item is described as being *passed in*, and there is no winner. Any bids below the reserve price are rejected.

The *auctioneer* manages the auction. They receive items and their reserve prices from *sellers*, and accept *bids* from potential *buyers*. The auctioneer is responsible for rejecting invalid bids – such as ones that do not exceed the current maximum bid, and those that are below the reserve.

Online auctions face some challenges – automated buying agents can attempt to wait until the very last moments of an auction, and place a bid that is fractionally above the current highest bid, seeking to win the auction for the lowest possible price. This strategy is known as *sniping*. Because sniping can give an unfair advantage over regular, human-placed bids, it is common to place limits on how frequently bids may be made.

In this assignment, all auctions are time-limited, with the duration being specified by the seller when placing the item up for auction. The auctioneer is the final arbiter of time. Bids received after an auction has finished, even if they were sent before expiration, are rejected.

Specification – `auctionclient`

The `auctionclient` program provides a command line interface that allows you to interact with the server (`auctioneer`) as a client – connecting, placing items for sale, finding items available, bidding on other auctions currently underway, and receiving and displaying the results of the auctions on your items.

To implement this functionality, `auctionclient` will probably require two threads – one for handling `stdin` and sending commands to the server, and another for handling messages from the server. You may optionally use a single thread and multiplexed I/O, such as `poll()` or `select()`, however **this is not required**.

Command Line Arguments

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

```
./auctionclient portno
```

- The mandatory `portno` argument indicates which localhost port the server is listening on – either numerical or the name of a service.

`auctionclient` behaviour

If an incorrect number of command line arguments is provided then `auctionclient` should emit the following message (terminated by a newline) to `stderr` and exit with status 20:

```
Usage: auctionclient portno
```

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

Connection error

If `auctionclient` is unable to connect to the auction 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 13:

```
auctionclient: cannot connect to port N
```

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

`auctionclient` runtime behaviour

Assuming that the commandline arguments are without errors, `auctionclient` is to perform the following actions:

- connect to the server on the specified port number (or service name) – see above for how to handle a connection error;
- read commands from `stdin`, and handle them as described below; and
- echo all lines received from the `auctioneer` to `stdout`.

If the network connection to the server is closed (e.g. `auctionclient` detects `EOF` on the socket, or receives SIGPIPE), then `auctionclient` shall emit the following message to `stderr` and terminate with exit status 18:

```
auctionclient: server connection closed
```

`auctionclient` shall interpret its input from `stdin` as follows:

- Blank lines (i.e. those lines containing no characters), and those beginning with the `#` character (comment lines), shall be ignored. ("beginning with" means the `#` character is in the leftmost position on the line.)
- The command `quit` on a line by itself shall cause the `auctionclient` to exit (with exit status 0) if the client is not currently involved in any auctions, i.e., any auctions for which that client submitted an item for sale are completed, and that client is not currently the highest bidder on any auctions. If those conditions are not true, then the client must print (and flush) the following message to `stdout`:

  ```
  Auction(s) in progress - can not exit yet
  ```

  and must not exit.
- All other lines shall be sent unaltered to the server (no error checking is required or to be performed)

If `auctionclient` detects EOF on `stdin` then it will exit. If the client still has auctions in progress (i.e. it has offered one or more items for sale and those auctions haven't expired, or if it is currently the highest bidder on one or more auctions) then `auctionclient` must print the following to `stderr` and exit with exit status 9:

```
Exiting with auction still in progress
```

Otherwise (i.e. the client is not involved in any current auctions), it will print nothing and exit with exit status 0.

Upon sending a command to the server, `auctionclient` shall expect a single line reply as per the communication protocol described in Communication protocol.

Other than `SIGPIPE` (which is required to gracefully handle the unexpected disconnection by the server) your `auctionclient` program is not to register any signal handlers nor attempt to mask or block any signals.

`auctionclient` termination

The recommended architecture for the `auctionclient` is to have one thread handling `stdin` and sending commands to the server, and another thread for receiving and interpreting responses from the server. (Note that you only need two threads total, not a "main" thread that creates two other threads to do this.) Since blocking reads will be performed on `stdin` and the server socket, it is difficult to use the methods described in lectures for cleaning up threads (e.g. a "`terminate`" flag variable in a shared data structure). For this reason, **when either thread detects a valid termination condition for the program, it may simply call `exit()` immediately – you should not attempt to use `pthread_cancel()`** or any other, more complicated termination mechanism.

The command protocol (see Communication protocol) is designed such that `auctionclient` can easily keep track of how many auctions it is currently involved in – both as a seller and/or a buyer. All commands sent to `auctioneer`, whether correct or not, generate replies. By interpreting these replies, `auctionclient` can determine when all of its auctions have completed. For the purposes of determining this condition, the client may assume that the server is correct and well-behaved.

`auctionclient` example usage

The following is an example of an interactive session with `auctionclient` (assuming the `auctioneer` 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. Lines in *italics* are explanatory comments on the overall system state, and do not appear in the terminal output):

```
1 $ ./auctionclient 49152
2 # A comment line - ignored
3 # A blank line - ignored
4
5 # An invalid item listing, missing an argument
6 sell chicken 10
```

```
:invalid
# A simple item listing, reserve 10, 30 second auction
sell chicken 10 30
:listed chicken
# A duplicate item listing
sell chicken 20 60
:rejected
# Attempt to bid on own item
bid chicken 1000
:rejected
# Another item for sale
sell potato 10 30
:listed potato
# Invalid listing - non-numeric reserve price
sell apple zz 60
:invalid
(...time passes, chicken auction over but no bids over reserve ...)
:unsold chicken
# Fast auction on a cow (5 seconds)
sell cow 1000 5
:listed cow
# See all items available (note our items are listed but so too is pumpkin
# listed by a different client before ours. We can see current bids on our items as well)
list
:list pumpkin 10 100 76|potato 10 20 7|cow 1000 2000 2|
# Bid on the pumpkin, which is accepted
bid pumpkin 110
:bid pumpkin
(...time passes, somebody else outbids us on pumpkin ...)
:outbid pumpkin
(...time passes, fast cow auction finishes ...)
:sold cow 2500
# try to quit, but potato still unsold
quit
Auction(s) in progress - can not exit yet
(...time passes ...potato has now sold - we have no more 'live' auctions)
:sold potato 20
# now the quit command can succeed
quit
```

Note that not all possible error conditions are present in this example.

Further note that the `auctionclient` must keep track of items it has listed as they are sold or passed-in – this is the only way it can determine whether it can exit or not. Similarly, it must also keep track of items for which it is the highest bidder.

Specification – `auctioneer`

`auctioneer` is a networked, multithreaded auction server allowing clients to connect, submit items for sale, bid on items, and list the items currently available. All communication between clients and the server is over TCP using a simple command protocol that is described in the Communication protocol section.

Command Line Arguments

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

`./auctioneer [--max` *`connections`*`] [--listenon` *`portno`*`]`

In other words, your program should accept up to ~~three~~ two optional arguments (~~two~~ 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 absent, then there is no limit to how many clients may connect (other than operating system limits which we will not test).

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

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

Program Operation

The `auctioneer` program is to operate as follows:

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

  ```
  Usage: auctioneer [--max connections] [--listenon portno]
  ```

  to `stderr`, and exit with an exit status of 8.

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

  - either of `--max` or `--listenon` does not have an associated value argument
  - the *connections* argument (if present) is not a non-negative integer
  - the port number (*portno*) 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 *portno* is missing or zero, then `auctioneer` shall attempt to open an ephemeral localhost port for listening. Otherwise, it shall attempt to open the specified port number. If `auctioneer` is unable to listen on either the ephemeral or specified port, it shall emit the following message to `stderr` and terminate with exit status 11:

  ```
  auctioneer: socket can't be listened on
  ```

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

- If specified (and implemented), `auctioneer` 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 `auctioneer` program should not terminate under normal circumstances, nor should it block or otherwise attempt to handle `SIGINT`.

- Note that your `auctioneer` must be able to deal with any clients using the correct communication protocol, not just the client programs specified for the assignment. 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.

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

Once a 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 `auctioneer` program itself must continue uninterrupted.

Auction running algorithm

The auction is run according to the following algorithm. Refer to the Communication protocol section for details of message syntax and required responses.

- `auctioneer` receives new items to auction via the `sell` command. The arguments to this command are validated and if correct, the item is now available for sale. If the command is invalid, the client is informed and the `sell` command does not succeed.
- `auctioneer` receives bids on sale items via the `bid` command. If the command syntax is valid, the item exists and is currently available, and the bid on the item is higher than any previous bid on the same item, then the bidding client is remembered as the current winning bidder for that item. If there was a previous winning bidder, that client is sent an `outbid` message informing it that it has been outbid on the item. If an invalid `bid` command is received, then the bidding client is informed and the bid is otherwise ignored by `auctioneer`.
- `auctioneer` accepts the `list` command from any connected client, regardless of whether that client has listed or bid on any items. The `list` command returns a formatted list describing each item available, its reserve price, current maximum bid, and the time remaining in the auction *for that item.*
- When the auction on any item completes (reaches the expiry time), `auctioneer` takes one of the two following actions:
  - If there is a winning bid which is greater than or equal to the reserve price, then the winning bidder is sent a `win` message identifying the item and the winning bid amount. The seller of the item receives a `sold` message indicating the item that was sold and the winning bid amount.
  - If there were no bids on the item, or no bids at or above the reserve price, then `auctioneer` sends a `unsold` message to the seller, informing it that the item did not sell. No message is sent to any previous bidding client in the event that the item goes unsold.
- ~~**Advanced:** `auctioneer` is to limit bids from clients to a maximum of one bid per item per second. That is, if a client has an accepted bid on an item more recently than one second ago, a `rejected` message is sent in reply.~~ Rate limiting is not to be implemented.

About time

`auctioneer` relies on time to determine when auctions expire. Expiry time is at the resolution of **milliseconds**. The following approach to handling time and auction expiry is recommended. You may choose another approach if you wish as long as the behaviour is the same.

- the provided library function "`double get_time_ms(void)`" returns a floating point time value in seconds, to millisecond resolution, e.g. the value 16.745 represents 16.745 seconds.
- for each auction item, store an expiry time (not a duration). So, as each new item is listed, store its expiry time calculated by `(get_time_ms() + duration)`.
- to report auction time remaining (e.g. for the `list` command), calculate `(item.expiry - get_time_ms())` which will yield a floating point value, in seconds, until the auction expires. Note that the `list` command shall report time remaining rounded down to the nearest second – see the Communication protocol section for details.

The simplest way to check on expiring auctions is to run a dedicated server thread which periodically checks the current time (`get_time_ms()`) against each item's expiry time, and marks as 'complete' any such items which have expired, and handle any necessary communication and cleanup associated with expired auctions. Be mindful of locking and mutual exclusion, because client threads may also be accessing the auction data structures while this is taking place.

Important: To avoid excessive busy waiting and CPU use, please have this auction expiry thread `sleep()` for 100 milliseconds between each check through the item list. The `usleep()` library function can be used to achieve this, although note that its argument is expressed as microseconds, not milliseconds! **Excessive CPU use in your solution will be checked for and may be penalised.**

Disconnecting clients

If a client disconnects before the end of an auction, (e.g. a `read()` or equivalent) from the client returns `EOF`, or a `write()` or equivalent fails), the `auctioneer` is to simply ignore that fact for the purposes of the auction, and close the network connection to that client, cleaning up data structures as required.

If a client disconnects, any auctions it was involved with proceed until they naturally expire. So, if a disconnected client was selling an item, other clients may still bid on the item in the normal manner. When the auction expires, `auctioneer` will inform the winning bidder, but will not attempt to send the `sold` message to the now-disconnected seller, nor send the `unsold` message if no valid bids were received.

Similarly if a bidder disconnects mid-auction, the auction proceeds and their bid remains valid. If this disconnected client ends up winning the auction, the selling client still receives the `sold` message, there is just no attempt to communicate to the now-disconnected bidder. If a disconnected client gets outbid, the `outbid` message is not sent.

`SIGHUP` handling (Advanced)

Upon receiving `SIGHUP`, `auctioneer` 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 number of active auctions in progress (at this instant)
- The total number of `sell` requests received (since program start), including invalid[2]. and rejected requests
- The total number of accepted `sell` requests received (since program start)
- The total number of `bid` requests received (since program start), including invalid[3] and rejected bids
- The total number of accepted `bid` requests received (since program start)

The required format is illustrated below. Each of the seven 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: `auctioneer` SIGHUP `stderr` output sample

```
Connected clients: 4
Completed clients: 20
Active auctions: 2
Total sell requests: 20
Successful sell requests: 14
Total bid requests: 34
Successful bid requests: 31
```

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 `--max` *`connections`* feature is implemented and a non-zero command line argument is provided, then `auctioneer` must not permit more than that number of simultaneous client connections to the server. `auctioneer` 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.

[2] An *invalid* `sell` request is one where the command word (characters before the first space on the request line) is "`sell`" but the arguments are invalid – see the Communication protocol section for details.

[3] An *invalid* `bid` request is one where the command word is "`bid`" but the arguments are invalid – see the Communication protocol section for details.

[redacted]

Program output

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

Communication protocol

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

Supported messages from a client to `auctioneer` are:

- `sell <item> <reserve> <duration>` – the client is requesting to sell the item called `<item>`, with a reserve price of `<reserve>`, with the auction lasting for `<duration>` seconds.
  See below for the conditions that must be met for a `sell` command to be accepted.
- `bid <item> <amount>` – the client is requesting to bid `<amount>` on the specified `<item>`.
  See below for the conditions that must be met for a `bid` command to be accepted.
- `list` – the client is requesting a list of items currently available for auction. See below for the reply format.

Single spaces are used as separators between fields, which implies that `<item>` must not contain spaces. All numerical values must be valid decimal integers (and subject to additional value requirements as described below).

Supported messages from `auctioneer` to a client are as follows. Error conditions in messages should be tested in this order as well – first look for "invalid" commands, then check to see if the command should be "rejected":

- `:invalid` – sent by the server to a client if any of the following conditions are true:
  - Invalid command word – an empty string or a command word which is not `sell`, `bid` or `list`
  - Too few or too many arguments
  - An invalid value for a numerical field, including
    - not a properly formed integer (e.g. "`10.2`", "`12p`" etc)
    - a negative reserve price in a `sell` command
    - a duration of less than 1 second in a `sell` command
    - a bid value of less than 1 in a `bid` command
  - Any other kind of malformed message
- `:rejected` – sent by the server to a client if one any of the following conditions is true:
  - a client attempts to `bid` on an item which is not for sale
  - a client attempts to `bid` on an item for which they are already the highest bidder
  - the client sends a `bid` command but the bid amount is less than the reserve price or less than or equal to the current maximum bid
  - a client attempts to `bid` on an item that they themselves have listed for sale
  - the client sent a `sell` command but there is already an item for sale by the same name
- `:listed <item>` – sent to the client when `<item>` is successfully listed for auction
- `:bid <item>` - sent to the client when a bid is successfully placed on `<item>`
- response to a `list` command, in the following format:

  `:list <item1> <reserve1> <maxBid1> <remain1>|...|<itemN> <reserveN> <maxBidN> <remainN>|`

  That is, for each item currently available for auction, the name, reserve price, maximum bid so far, and remaining time (rounded down to the nearest second) is emitted in a space-separated format, followed by a vertical bar "`|`" character. **Items should be reported in the same order that they were submitted to `auctioneer` – there is no need to perform additional sorting on the list, simply storing them in order of arrival is sufficient.** Note that the rounding down of time means that the remaining time for an item should be reported as 0 (zero) if the time remaining is between 0 and 1 seconds.

  If there are no items currently available for auction, then the string "`:list`" is returned.
- `:outbid <item> <maxbid>` – the receiving (bidding) client has been outbid on `<item>`, the new highest bid is `<maxbid>`.

  Note, the outbid client will receive no further notifications about this item unless they bid again. They are effectively out of the auction at this point.
- `:won <item> <price>` – the receiving bidding client won the auction on `<item>` with their bid of `<price>`
- `:sold <item> <price>` – the receiving selling client just sold `<item>` for `<amount>`
- `:unsold <item>` – the receiving selling client had the given item passed-in, i.e. there were no bids at the reserve price or higher when the auction expired

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.

The `get_time_ms()` function will return a millisecond-precision, monotonically increasing time value to support all time-related functions in this assignment. The specific value returned is not particularly meaningful, however the difference between values returned from calls to this function can be used to accurately measure the passage of time.

[redacted]

```
double get_time_ms(void);

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);
```

[redacted]

These functions and the HttpHeader type are documented on moss and their behaviour is described in the man pages on moss (`man 3 split_by_char`, `man 3 get_HTTP_request`, and `man 3 free_header`).

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`

Finally, will need to link your programs with the `-lm` flag to add the maths library, required by the `get_time_ms()` function.

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`.

Style

Your program must follow version 2.4.1 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 `auctionclient` and `auctioneer` independently using `netcat` as demonstrated in the lectures. You can also use the provided demo programs `demo-auctionclient` and `demo-auctioneer`.

3. The `read_line()` function from `libcsse2310a3` may be useful in both `auctionclient` and `auctioneer`.
4. The multithreaded network server example from the lectures can form the basis of `auctioneer`.
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 use.
7. Remember to `fflush()` output that you `printf()` or `fprintf()`!

Possible Approach

1. Try implementing `auctionclient` first. (The programs are independent so this is not a requirement, but when you test it with `demo-auctioneer` it may give you a better understanding of how `auctioneer` works.)
2. For `auctioneer`, start with the multithreaded network server example from the lectures, gradually adding functionality for supported message types.
3. Design a simple data structure for `auctioneer` to store items for sale. This data structure will be accessed by client threads to add new items, determine validity of bids, and respond to `list` commands. It will also need to be accessed by whichever thread checks for auction expiry. Locking and mutual exclusion will be essential, but don't over-complicate it!

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()`
- Functions described in the man page as non standard

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 `auctionclient` and `auctioneer`) 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-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)[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 will be asked to confirm references in your code.

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 16 marks for `auctionclient` and 44 marks (CSSE2310) or 54 marks (CSSE7231) for `auctioneer`.

1. `auctionclient` correctly handles invalid command lines (2 marks)
2. `auctionclient` connects to server and also handles inability to connect to server (2 marks)
3. `auctionclient` correctly handles input from `stdin` (i.e. sends appropriate lines to server) (3 marks)
4. `auctionclient` correctly displays lines received from the server (2 marks)
5. `auctionclient` correctly handles communication failure on the server socket (includes handling SIGPIPE) (2 ~~3~~ marks)
6. `auctionclient` correctly handles the `quit` command and `EOF` on `stdin` (5 ~~4~~ marks)

7. `auctioneer` correctly handles invalid command lines (3 marks)
8. `auctioneer` correctly listens for connections and reports the port (including inability to listen for connections) (2 marks)
9. `auctioneer` correctly handles invalid command messages (3 marks)
10. `auctioneer` correctly handles a single auction with correct expiry and bidding behaviour (5 marks)
11. `auctioneer` correctly handles multiple bidders on a single auction with correct expiry and bidding behaviour (7 marks)
12. `auctioneer` correctly handles `list` requests (4 marks)
13. `auctioneer` correctly handles multiple simultaneous auctions from multiple clients and multiple bidders (using threads) (7 marks)
14. `auctioneer` correctly handles disconnecting clients and communication failure (3 marks)
15. `auctioneer` correctly implements client connection limiting (3 marks)
16. `auctioneer` correctly implements SIGHUP statistics reporting (including protecting data structures with mutexes or semaphores) (7 marks)

17. (CSSE7231 only) `auctioneer` correctly listens on the port specified by `A4_HTTP_PORT` (and handles inability to listen or environment variable not set) (3 marks)
18. (CSSE7231 only) `auctioneer` correctly responds to HTTP requests (including invalid requests) from a single client issuing one request per connection (4 marks)
19. (CSSE7231 only) `auctioneer` 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.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.4.1 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[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 number of arguments for `auctioneer`.
- Added `:unsold` message to protocol description and noted that it should not be sent to a disconnected client.
- Removed the description of advanced functionality – no rate limiting is to be implemented.
- Added hint about `fflush()`ing.
- Added note about two threads in the client.
- Added note about the `:list` response including remaining times of 0 (zero) if the time remaining is between 0 and 1 seconds.
- Made it clear that the client handling communication failure on the server socket includes handling SIGPIPE.
- Modification of marks for `auctionclient` category 5 and 6.