Middle East Technical University Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2019-2020

Homework 1 - Auction Server

Due date: 15/03/2020, Sunday, 23:59

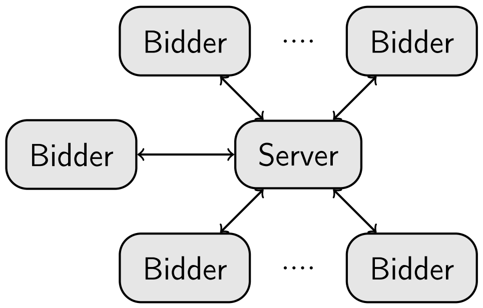
1 Introduction

The objective of this assignment is to learn the basics of IPC (inter-process communication) through the implementation of an auction server.

An auction as the "process of buying and selling goods or services by offering them up for bid, taking bids, and then selling the item to the highest bidder or buying the item from the lowest bidder"[1]. Auctions take place in person or online for selling all kinds of items from cars to house. In an auction, an item is presented to a group of bidders, along with a starting bid. Starting bid is the minimum bid that can be placed on the item. The minimum increment is the lowest increase from the current highest that is needed for a new bid. The bidders makes bids, either by increasing the current bid placed on the item with a minimum increment, or by withdrawing from the bids. At the end of the bidding period, the item is sold to the highest bidder. In this assignment, you will be implementing an online auction server auctioning a single object. The starting bid and minimum increment of the auction will be given as parameters. There will be an unknown number of bidders attending the auction. The auction will continue as long as the bidders are active. Finally, the winning bidder will be determined and announced.

2 Components and Communication

The system will consist of two components: *auction server* and *bidders*. The *server* and each of the *bidder*s will run as separate processes.



[1]https://en.wikipedia.org/wiki/Auction

The *bidder*s are programs capable of communicating with the server to bid on the auction item. They receive auction requirements from the server and place their bids continuously until they stop. After that point they wait for the auction to finish before quitting.

The bidders interact with the server using fixed messages. They send these messages to the standard output and receive the responses from the standard input.

The *server* should store auction information and update it according to the messages it receives. Auction information should at least contain current highest bid and bidder. After receiving the messages, the servers responds to the bidder with updated information. This continues until all of the bidders stop bidding. Finally, the server sends the winner information back to the bidders and quits.

You will be implementing the server program and bidder programs will be provided to you.

The bidder programs will be provided as an executable. You program should fork and spawn the bidders as separate processes before executing the server code. The IPC between the server and the bidders should be coordinated properly.

2.1 Server

The server should read the starting bid, minimum increment, number of bidders and their executable paths together with their arguments from the standard input during initialization (see Input & Output for details). Afterwards, it should fork and execute (see `fork` and `exec` function family) the bidder processes. Note that before forking them, actions necessary to support communication between them via bidirectional sockets (see Communication) and to redirect the standard input and output to these bidders using `dup2` function should be carried out. After the creation of these bidders, the server should wait for the initial messages.

After initialization, the server should run in a loop until all of the bidders stop bidding. Upon begin spawned, the bidders send their bids (the number, type or frequency of these bids are not known in advance) and inform the server when they withdraw (i.e stop bidding).

The server should complete these tasks in the given order in a while loop:

1. Check for any of the file descriptors (from socket) connected to the child processes for an input to be ready. Hint: Use the `poll` or `select` system call.
2. If there is data to be read on a file descriptor
    - Read the message
    - Print the message to the standard output using the provided library as explained in the Input & Output section.
    - Determine the type of the message. The message types and contents are explained in the Communication subsection.
    - Determine the response based on the message type and its contents.
    - Send the response and update the auction information.
    - Print the response to the standard output using the provided library as explained in the Input & Output section.

Once all the bidders withdraw and stop bidding, the server determines the highest bidder and prints this information using the provided library. Then, the highest bidder message should be sent to all bidders, print the message and wait for their exit.

Subsequently, the server should reap the bidders, print the reaping information using the library and terminate. It should also close all the remaining socket connections. No zombie processes should be left after the termination of the server.

The communication messages between server and bidder processes is explained in detail in the Communication section.

2.2 Bidder

The bidder program will be provided by us and you should only focus on the communication of the server with the bidders by responding properly to the messages sent by them. Using the message structure explained in the Communication section, the server should respond to messages coming from the bidders and update the auction information.

There are two aspects of the bidders you should keep in mind: First, there exists an unknown communication delay between the bidder and the server given in milliseconds. The first message of the bidders will contain this delay. Second; each bidder should have a unique positive integer id. This id is assigned and sent to them by the server within the response to the first message.

A sample bidder will be provided to test your server code. You can also write your own bidders to test different scenarios. When bidders stop bidding, they wait for the highest bidder to be announced and quit afterwards.

2.2.1 Sample Bidder

The sample bidder repeatedly increases the bid until the bid reaches a limit. This limit is randomly selected at every run. The communication delay of the bidder is given as an argument as shown below:

```
> ./Bidder 50
```

where the communication delay is 50. The auction requirements will be received from the server, and the bidder will bid according to these parameters.

2.3 Communication

The communication between server and bidder processes will be carried out via bidirectional sockets which can be created as follows:

```
#include<sys/socket.h>
#define PIPE(fd) socketpair(AF_UNIX, SOCK_STREAM, PF_UNIX, fd)
```

If you use `PIPE(fd)`, data written on `fd[0]` will be read on `fd[1]` and data written on `fd[1]` will be read on `fd[0]`. Moreover, both file descriptors can be used to write and read data.

The server should be serving an arbitrary number of bidders. That means, it should create **n** sockets and read requests from **n** different file descriptors. If it blocks in one of them the requests from others has to wait. Therefore, it should not block on any socket, instead should check whether there is data to be read on that socket. In order to check whether there is data on a particular socket, the `poll()` or `select()` system calls can be used.

The server pseudo-code is given as:

```
while there active bidders:
    select/poll on socket file descriptors of active bidders
    for all file descriptors ready (have data)
        read request
        serve request
```

2.3.1 Messages

The message structure between server and the bidders is defined in the `message.h` file.

Bidder Messages

The messages of the bidder to the server are defined with the following structure:

```
typedef struct client_message {
    int message_id; //non-zero and unique message id
    cmp params;
} cm;

typedef union client_message_parameters {
    int status;
    int bid;
    int delay;
} cmp;
```

The bidders communicate with the `client_message` struct. The `message_id` determines the type of message. Depending on the message type, different message parameters is used. There are three message types as defined in the `message.h` file. The different message parameters for each message type are represented as an union `client_message_parameters`.

1. `CLIENT_CONNECT`: This is the first message that the bidders send to the server. It is sent to inform the server about the delay and receive the auction information to start bidding. The `delay` field of the parameters is used to indicate the delay. This delay will be simulated by the bidder and minimum of the bidder delay can be used when there is no data on any socket. The delay is given in milliseconds.

2. `CLIENT_BID`: This message is used when the bidders want to place a bid. The `bid` field is used to indicate the bid amount.

3. `CLIENT_FINISHED`: This message is used to indicate that the bidder will stop bidding from this point forward. After this message is sent, the bidder will wait for the auction to finish and upon receiving the corresponding message, quit with a status. The status number is sent to the server early with this message to check whether the bidder process has quit normally. The `status` is used to indicate that information.

Server Messages
The messages of the server to the bidder are defined with the following structure:

```
typedef struct server_message {
    int message_id; //non-zero and unique message id
    smp params;
} sm;

typedef struct connection_established_info {
    int client_id;
    int starting_bid;
    int current_bid;
    int minimum_increment;
} cei;

typedef struct bid_info {
    int result;
    int current_bid;
} bi;

typedef struct winner_info {
    int winner_id;
    int winning_bid;
} wi;

typedef union server_message_parameters {
    cei start_info;
    bi result_info;
    wi winner_info;
} smp;
```

Specifically, the server communicates with the `server_message` struct. The `message_id` determines the type of message. Depending on the message type, different message parameters are used. There are three message types as defined in the `message.h` file. The different message parameters for each message type, represented as a union `server_message_parameters`, are defined as follows:

1. `SERVER_CONNECTION_ESTABLISHED`: This message is sent in response to `CLIENT_CONNECT` message. It uses `start_info` field from the parameters. This field is a `connection_established_info` struct containing all of the information needed by the bidder before it starts bidding. Its fields are:

    - `client_id`: The unique id assigned to the bidder by the server.
    - `starting_bid`: The minimum amount the bidders can bid.
    - `current_bid`: The current highest bid.
    - `minimum_increment`: The required minimum increase from the current highest bid.

2. `SERVER_BID_RESULT`: This message is sent in response to the `CLIENT_BID` message. It uses the `result_info` field from the parameters. This `bid_info` struct should contain the result of the bid and the updated current highest bid. Its fields are explained below:

    - The `result` represents the result of the bid. It can be one of four different values. They are also given to you in the `message.h` file. They are explained below:
        - `BID_LOWER_THAN_STARTING_BID`: The bid amount is lower than the starting bid.
        - `BID_LOWER_THAN_CURRENT`: The bid amount is lower than the current highest bid.
        - `BID_INCREMENT_LOWER_THAN_MINIMUM`: The difference between the sent bid and current highest bid is smaller the minimum increment.
        - `BID_ACCEPTED`: The bid does not contain the problems described above and accepted.

      The checks on the bid amount should be done in order given above.
    - The `current_bid` represents the current highest bid.

3. `SERVER_AUCTION_FINISHED`: This message is sent to all the bidders after they stop bidding and the winner is determined. It uses `winner_info` field from the parameters. The `winner_id` should contain the id of the winning bidder and the `winning_bid` should contain the winning bid amount.

The overall architecture can be depicted using the following diagram:

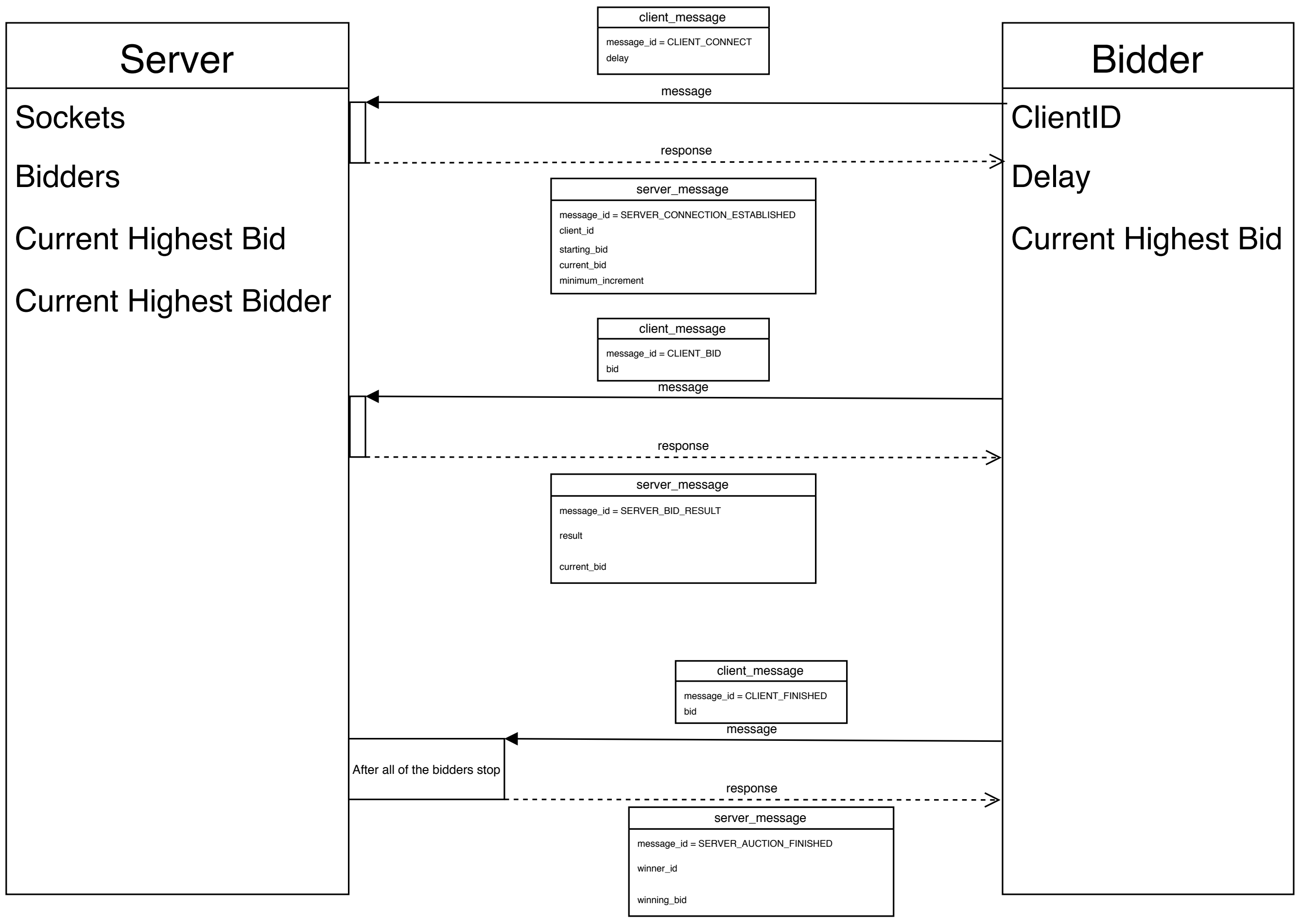


Figure 1: Server & Bidder Diagram

3 Input & Output

The server reads the auction requirements and the bidder information from the standard input and print to the standard output.

For output, you are provided with a library that outputs information in a specific format.

3.1 Input

The input to your program should be:

```
<starting_bid> <minimum_increment> <number_of_bidders>
<first_bidder_executable> <number_of_arguments> [<arguments>]
<second_bidder executable> <number_of_arguments> [<arguments>]
...
...
<last_bidder_executable> <number_of_arguments> [<arguments>]
```

where `starting_bid` and `minimum_increment` are the auction parameters and `number_of_bidders` is the number of bidders to be executed. The remaining lines provide the parameters for the bidders. A sample input is:

```
0 5 3
./bidder 1 5
./bidder 1 10
./different_bidder 2 1 5
```

3.2 select() and poll()

The select() and poll() system calls are used to block on multiple file descriptors simultaneously. In this homework, the server should read data from **N** bidders. If it tries reading from multiple descriptors one by one, `read()`ing from the first file descriptor may block due to lack of data, while data being available on other descriptors.

These system calls lets you block on *any* of the file descriptors. If at least one of them have data to read (or write), call will return the set of file descriptors that are ready so far. Then you can read those data without blocking.

3.3 Output

The output of the server should be printed **only** using functions provided in `logging.h` and `logging.c`.

3.3.1 Output

The header file is given in below:

```
#ifndef CENG334HOMEWORK1_LOGGING_H
#define CENG334HOMEWORK1_LOGGING_H


#include "message.h"
#ifndef __cplusplus
#include <stdio.h>
#else
#include <cstdio>
#endif

typedef struct output_info {
    int type;
    int pid;
    smp info;
} oi;

typedef struct input_info {
    int type;
    int pid;
    cmp info;
} ii;

void print_output(oi* data, int client_id);
void print_input(ii* data, int client_id);
void print_server_finished(int winner, int winning_bid);
void print_client_finished(int client_id, int status, int status_match);
#endif //CENG334HOMEWORK1_LOGGING_H
```

The function `print_output` is used to print the messages sent from the server. The `output_info` struct contains the information about the message. Similarly, `print_input` is used to print the messages received by the server. The `input_info` struct contains the information about the message. Both message structs share the same fields and they are explained below:

- `type`: The message type.
- `pid`: The process id of the bidder that the server sent message to or receive from.
- `info`: The message parameters of the message. It is `server_message_parameters` struct for the messages sent from the server. It is `client_message_parameters` for the messages received by the server.

The function `print_server_finished` is used to indicate the winner of the bid. It is called before sending the winner information to the bidders. The parameters `winner` and `winning_bid` indicate unique client id and winning amount of the winning bidder. The function `print_client_finished` is called after reaping each client. The `client_id` is the reaped id of the bidder, `status` is the exit status of the client received from the `wait` family function and the `status_match` is used to indicate whether the actual status of the bidder matches the previously sent status.

4 Specifications

- Your codes must be written in C or C++.
- Your programs will be compiled with gcc or g++ and run on the department inek machines. No other platforms and/or gcc/g++ versions etc. will be accepted. Therefore make sure that your code compiles and executes on inek machines before submission.
- Do not forget to close the socket connections and reap (see `wait` & `waitpid`) the bidders processes.
- Your program must not leave any zombie processes. If you leave zombie processes in a testcase, you will lose points.
- Do not make any changes to the codes provided to you and only use the given message structures for communication between the server and the clients.
- Your code will be tested with black box inputs. Due to the unpredictability of the execution sequences, it will not be a direct comparison to a correct output. It will be checked as per execution basis.
- **Using code that is not your own is strictly forbidden and constitutes as cheating. This includes code from your friends, previous homework, or the internet. We have a zero tolerance policy on cheating.**
- Follow the course page on Cow for any updates and clarifications. Please ask your questions on Cow instead of e-mailing if the question does not contain code or solution.

5 Submission

Submission will be done via ODTUClass. You will submit a tar file called “hw1.tar.gz” that contain all your source code together with your makefile. You do not need to submit the libraries we have provided. Even if you do, they will be replaced with the originals, so do not make any changes on them. Your tar file should not contain any folders. Your makefile should be able to create a single executable named server and it should be able to run using the following command sequence.

```
> tar -xf hw1.tar.gz
> make all
> ./server
```

You can assume that all the executables and library files are present for the required compilation and execution. **If there is a mistake in any of the 3 steps mentioned above, you will lose 10 points.**