A4 Network Programming

Due: Electronically, by 10:00 PM on April 2

Introduction

In this assignment, you'll be working with Unix sockets to build a server for a text-based battle game, modelled after a Pokemon battle. You will **not** write a client. To make testing easier, your executable must be called `battle`.

You may work in pairs for this assignment. MarkUs will only create the appropriate directory in your repository when you log into MarkUs and either create your group, or declare that you will work alone. The groups will get a new shared repository, and the students working solo may also get a new repository. Please log into MarkUs well before the deadline to take these steps. (If you create the directory in svn, then MarkUs won't know about it and we won't be able to see your work.)

Playing the Game

Before detailing your tasks, let's first play the game to see how it works. On `wolf` port 8888, I have a battle server running. To connect to it, type the following at the shell: `stty -icanon; /bin/nc wolf.cdf.utoronto.ca 8888`. This carries out two commands: `stty` to prevent the terminal driver from waiting for complete lines of text (you'll see why soon), and `nc` to connect to the battle server.

Once you've connected, you'll be asked for your name. Type something (Dan, for example) and hit enter. You'll then be told that you are waiting for an opponent, and you will be stuck there until someone else connects to the server. So, instead of waiting for someone else to log-in, just open another terminal and connect to `wolf` port 8888 from there too. (If you're using ssh, you can just start a new ssh instance. You could also job control back and forth between two `nc` instances running on the same terminal.) You will be asked for the name of this second client; type something (Karen, for example) and hit enter.

At this point, two players are available, so they will engage in combat, with one of the two players randomly earning first strike. Prior to each match (i.e. battle), each player is given a random number of hitpoints and a random number of powermoves. (It doesn't matter how many hitpoints or powermoves you had remaining after a match; when the next match starts, these stats are freshly randomized.) Players repeatedly exchange attacks, until one of the players loses by virtue of having 0 hitpoints remaining.

There are two kinds of attacks: you can press a for a regular attack or p for a powermove. Regular attacks are weak but guaranteed to hit; powermoves are strong, but not guaranteed to hit, and limited in number. Also, you'll notice that a player is unaware of the quantity of powermoves held by their opponent. So you never know whether your opponent is saving a powermove for later or they are all out of powermoves.

As per the menus printed by the battle server, the other available option is to `(s)peak` something. Only the currently-attacking player can talk, and saying something does not take up a turn. It's like in-game messaging. I trust we will all play nice.

Similar to when you entered your name, no text is sent to your opponent until you hit enter. That is, the server is buffering a full line. Remember that `stty -icanon` command we ran earlier? That command causes each character to be fired out as soon as you press it at the keyboard. (You noticed this when pressing a or p; the action happened immediately, and you didn't have to hit enter first.) So it's the server here that's buffering the text until it gets a newline.

Play around with attacking, powermoving, and sending text, until the match is over. At that point, Dan and Karen are still connected, waiting for new opponents. Importantly, you'll notice that they won't battle again; they'll just sit there waiting for someone else to come online. (Consider what would happen if they were allowed to battle again: after their first match, they'd likely be the only two people free and would battle again and again and again.) So, add a third client, and that third client will then engage one of the two existing clients. Add a fourth client, and they will engage the lone player not currently in a battle. This also shows that multiple matches can occur simultaneously and independently. The server never blocks waiting for anything; only matches with new action are serviced.

The last thing I'd like to highlight here is what happens when a client drops. Initiate a match between two players, and then hit ctrl+c on one of them to kill their `nc` process. Switch to the remaining client's window; you'll notice that this client is deemed the winner, and that they go back to seeking another opponent.

When you're finished, type `stty icanon` at the shell to return the terminal back to canonical mode. If you don't do this, programs will work in unexpected ways. (For example, do a `stty -icanon` to get back to noncanonical mode, then type `cat`. You'll notice that instead of waiting for full lines, each character is echoed by `cat` as soon as you type it.)

Your Tasks

Your task is to implement the battle server, as outlined above and further detailed below. You must support the same functionality offered by the sample implementation on `wolf` port 8888.

Login

When a new client arrives, add them to the end of the active clients list, and ask for and store their name. Tell the new client that they are awaiting an opponent; tell everyone else that someone new has entered the arena. Scan the connected clients; if a suitable client is available, start a match with the newly-connected client.

Matching

A client should never wait for a match if a suitable opponent exists. Consider clients A and B:

- If A or B is currently in a match, then A and B cannot be matched
- If A last battled B, and B last battled A, then A and B cannot be matched
- Otherwise, they can be matched

In particular, new matches are possible when a new client logs-in and when an existing match terminates by completing normally or due to a client dropping. Suitable partners should be searched starting from the beginning of the client list. Once a match finishes, both partners should be moved to the end of the client list.

Combat

When two players are matched, they combat until one loses all their hitpoints or one of the players drops. Players take turns attacking. Call the currently-attacking player the **active** player. Only the active player can say something; any text sent by the inactive player should be discarded. Any invalid commands sent by the active player should also be discarded. (This includes hitting p when no powermoves are available.)

You are encouraged to experiment with these parameters, but here is what I used:

- Each player starts a match with between 20 and 30 hitpoints. (Note that hitpoints and powermoves are reset to random values on the start of a new match, independent of what the values may have been following their previous match.)
- Each player starts a match with between one and three powermoves.
- Damage from a regular attack is 2-6 hitpoints.
- Powermoves have a 50% chance of missing. If they hit, then they cause three times the damage of a regular attack.

The active player should be sent a menu of valid commands before each move. The p command should not be printed (or accepted) if the player has no powermoves remaining.

To generate random numbers, use `srand` once on program entry to seed the random number generator, and then keep calling `rand` to get random numbers. King 10.2 contains an example of doing this.

Dropping

When a client drops, advertise to everyone else that the client is gone. If the dropping client was engaged in a match, their opponent should be notified as the winner and told that they are awaiting a new opponent. The match involving the dropping client should be removed.

Using Select

The server must never block waiting for input from a particular client or the listening socket. (After all, it can't know whether a client will talk next or whether a new client will connect. And, in the former case, it can't know **which** client will talk next.) This means that you must use `select` rather than blocking on one file descriptor.

Makefile

Create a Makefile that compiles a program called `battleserver`. In addition to building your code, your makefile must permit choosing a port at compile-time. To do this, first add a `#define` to your program to define the port number on which the server will expect connections (to be safe, choose a number x between 10000 and 32767):

```
#ifndef PORT
  #define PORT x
#endif
```

Then, in your makefile, include the following code, where y should be set to a number possibly different from x:

```
PORT=<y>
CFLAGS= -DPORT=\$(PORT) -g -Wall
```

Now, if you type `make PORT=1500` the program will be compiled with `PORT` defined as `1500`. If you type just `make`, `PORT` will be set to `y` as defined in the makefile. Finally, if you use `gcc` directly and do not use `-D` to supply a port number, it will still have the `x` value from your source code file. This method of setting a port value will make it possible for us to test your code by compiling with our desired port number. (It's also useful for you to know how to use `-D` to define macros at commandline.)

Sample Server

We are providing this sample server (./simpleselect.c) (fixed bugs March 17) that you will hopefully find useful for the assignment. It demonstrates keeping a list of clients, using `select`, broadcasting messages, etc. Make sure you understand how that code works or you will run into all kinds of problems.

Testing

Since you're not writing a client program, the `nc` tool mentioned above can be used to connect clients to your server. Your server is required to work in noncanonical mode (`stty -icanon`); we will test in this mode. For completeness, you might also wish to test in canonical mode (`stty icanon`). We won't explicitly be testing in this mode, but problems here might indicate unfounded assumptions in your server.

Note that on CDF, you should run `nc` as `/bin/nc` (not just `nc`), because you might have another unrelated `nc` earlier in your path.

To use `nc`, type `/bin/nc wolf.cdf.utoronto.ca xxx`, where `xxx` is the port on which your server is listening.

What to submit

You will commit to your repository in the `a4` directory all `.h` and `.c` files, and your makefile.

Remember: when we run `make` on `cdf`, it must create an executable called `battle` without errors or warnings. If you are not able to fully complete the assignment, please include a `README.txt` file that explains what is working and what is not working.

Make sure that you've done `svn add` on every file. Check from within MarkUs, or checkout your repository into a new directory, to verify that everything is there. There's nothing we can do if you forget to submit a file.