# Lab 10

**Due** Mar 27 by 6:30pm **Points** 1

# Lab 10: Sockets

## Important warning about file format

As part of this lab, you are required to to use `script` to capture your `gdb` sessions. For Lab 6, some students submitted files that were not viewable via MarkUs . Be sure to complete the `gdb` part of the lab on teach.cs to avoid file format issues.

## Introduction

The purpose of this exercise is to practice using the socket-related system calls. The buffering part of this exercise will be particularly useful in assignment 4.

The Unix programming concepts you will be using in this exercise are:

- `socket`
- server functions: `bind`, `listen`, `accept`
- client functions: `connect`

## Setup: starter code and choosing a port

Do a `git pull` in your repository to download the starter code for Lab 10.

The port on which the server listens is defined on the second line of the `Makefile` found in the starter code. We have it set to 30001 by default; before continuing, we'd like you to change this.

To avoid port conflicts, use the following number as the port on which your server will listen: take the last four digits of your student number, and add a 5 in front. For example, if your student number is 1008123456, your port would be 53456. Using this base port, you may add 1 to it as necessary in order to use new ports (for example, the fictitious student here could also use 53457, 53458, 53459, 53460). Sometimes, when you shutdown your server (e.g. to compile and run it again), the OS will not release the old port immediately, so you may have to cycle through ports a bit.

Note that you only need to change the `PORT` variable value in the `Makefile`. It uses this variable and the `-D` flag to tell `gcc` to define a constant during compilation that will be accessible to the program during execution.

## Introduction: sending and receiving messages

Open `randclient.c`. It is a client program that sends multiple copies of the same message to a server. Each message ends with a network newline `"\r\n"`.

You might expect that a server using `read` will receive one complete line of text with each `read` call. However, this isn't guaranteed to happen. For example, in a busy network or with a slow client, one line of text might be split across multiple TCP segments. A server cannot do a single `read` and expect to get any sensible unit of communication.

To simulate this, `randclient.c` artificially splits up its copies of the messages over multiple `write` calls. This will cause trouble for any server that expects to read entire messages at a time -- `readserver.c` is one such server.

Run `readserver &` and then run `randclient 127.0.0.1`. Notice that the server thinks that every small piece of a message is a complete message, which of course is incorrect. Your task is to add buffering to your server so that it waits for a complete line (ending with the network newline) before printing the message.

## Task 1: A Buffering Server

In `bufserver.c`, we have given you the skeleton for a buffering server. The `Step` comments in the code indicate what you should do to complete the server. The main idea is that you have a buffer (a character array) where you store pieces of a message until you have a network newline ("`\r\n`"). Each message piece goes after the data that you've already placed in the buffer. When you find a network newline, you know that you have a complete message, so you can print that message and then shift any remaining bytes to the front of the buffer as the start of the next copy of the message.

An example of how these buffered reads will work is given in a **diagram**  .

Once you finish the steps, try running `bufserver &` and then run `randclient 127.0.0.1` to connect to the server. The server should have one line of output per message. If there is extra garbage printed or other errors, carefully go over the steps again -- it's very easy to get off-by-one mistakes here. Another tip is to be careful to note when you are guaranteed to have a null-terminated string in `buf`, and when you aren't.

# Task 2: Using the Debugger

For this part, you are asked to run the client and the server in the debugger to help you understand how the socket calls work. To carry out this exercise, you will need to have two terminal windows open on your machine. One will be used to run the server and the other will be used to run the client. If you're working remotely, a simple option is to open two ssh connections (but make sure they are both on the same machine, like wolf).

## Debugging Exercise 1

In the client window, run `script client.1`, and in the server window run `script server.1`. This will cause transcripts of your sessions to be written to files that you will submit.

In the server window, run `gdb bufserver` and set a breakpoint at `set_up_server_socket` (`break set_up_server_socket`). Step through the code using `next` (or `n`) until you see the `bind` call. (Remember that gdb shows you the line just before it is to be executed. You want to stop before the `bind` call happens.) Try running the client (`randclient 127.0.0.1`) in the client window and notice that the connection is refused.

In the server window, step once (just before `listen`), and try running the client again. Still the connection is refused because the socket is not fully set up on the server.

In the server window, step until you see the `accept` call, and then step one more time. gdb does not give you the next prompt because the `accept` call has not returned yet. The `accept` call is blocked waiting for a connection from a client.

In the client window, run the client again. Note that the `accept` call returns in the server. Keep stepping in the server until the `read` call. This is where the server starts reading the message from the client. Keep stepping in the server until all of the bytes are received.

In both the client and server windows, type `exit` to complete the scripts. Commit your script files before you move on.

## Debugging Exercise 2

In the client window, run `script client.2` and in the server window, run `script server.2`.

Now run `gdb bufserver` in the server window, and set a breakpoint at the start of `main`; do the same in the client window, but with `randclient`. In the client, type `run 127.0.0.1` to run the client in gdb.

Try stepping through the two programs in different orders so that you can see how the messages are transferred. Look at the values of variables (e.g. `buf`, `inbuf`) at different times to see their values.

This is your chance to really think about how the client and server talk to each other, so take advantage of the opportunity to learn a bit more gdb. Pay attention to how the data is being sent. Reading and writing on a socket has to be done carefully. Time you spend understanding what is happening with this example code will help you get started on Assignment 4.

After you have tried several variations, exit from gdb and type `exit` in both windows. Commit your script files.

# Submission

Submit your final `bufserver.c`, `client.1`, `client.2`, `server.1`, and `server.2` files to MarkUs under the `Lab10` folder in your repository. You should not make changes to any other files.