CS 422 (Park) **Assignment V** **Due: Apr. 5 (Wed.), 2006**

Submission instructions: Please type your answers and submit electronic copies using `turnin` *by 5pm on the due date. You may use any number of word processing software (e.g., Framemaker, Word, LaTeX), but the final output must be in pdf or ps format that uses standard fonts (a practical test is to check if the pdf/ps file prints on a CS Department printer without problem). For experiments and programming assignments that involve output to terminal, please use* `script` *to record the output and submit the output file. Use* `gnuplot` *to plot graphs. Use* `ps2gif` *to convert a eps/ps plot to gif format (e.g., for inclusion in Word) if there is a need.*

PROBLEM 1

Read Chapters 28, 29, and 30 from Comer.

PROBLEM 2 (60 pts)

As a continuation of Problem 4, Assignment III, reimplement the client/server application so that it uses UDP sockets in place of FIFO. Most of your code will remain the same, the changes relating to replacing the FIFO based client/server interaction with a socket based interface. In the server code, remove the 3 second sleep in the child before `exec`. In the client code, use a SIGALRM value of 10 msec. Test your client/server system on the same benchmark suite, albeit with clients executed on different hosts in the lab. Use `ssh -X` *login-name*`@`*machine-name* to open remote shell windows from a single machine. Record your terminal interaction using `script`. How do your results in terms of client/server behavior differ from those observed in Problem 4, Assignment III? Include a discussion of your results.

PROBLEM 3 (60 pts)

Under ∼park/pub/cs422, you will find `talk_client` and `talk_server`, simplified implementations of `talk`, the precursor of today's messenger applications. From a programming technique perspective, the main feature of interest in `talk` is that you be able to see (i.e., read) what the other party is writing to you in the midst of your own write. This means that when you write—i.e., the code reads the characters you type on the keyboard—packets arriving from the other party carrying messages must be handled asynchronously and displayed on the terminal. Hence, a solution where write followed by read is repeated in an infinite loop is inadequate. To achieve asynchronous I/O, a SIGIO (or SIGPOLL) handler must be written and registered with the kernel, which is a callback function that the kernel invokes whenever a new packet arrives. The client/server talk example is incomplete in several respects. First, the syntax for turning a file descriptor into asynchronous mode uses System V UNIX (e.g., Solaris) convention, which must be modified to match the syntax suited for Linux. This is one of the few OS dependent aspects of network programming that must be carefully handled when porting code across different UNIX platforms (e.g., also different for BSD UNIX). Second, the SIGIO handler is missing. Writing a SIGIO handler is straightforward but more important is an application design decision: how to display a newly arriving message on the same terminal so that what you are writing is not scrambled with what you are receiving. Using the `curses` library, one can split the terminal in two so that what you write appears on the top half and what you read appears on the bottom. By using X-windows, the same can be accomplished by demuxing write/read across two windows. Third, `talk` is by its very nature a peer-to-peer application where there is no separation of client from server. They are one and the same. Complete the provided code and get it to work by solving these three problems. When registering your SIGIO handler, use `sigaction` in place of `signal`. Do not use `curses` nor separate X-windows to separate input/output. Here, we are not concerned with pretty output. Invent a rudimentary but working solution using *stdio*. Remove any client/server dependencies in the code so that the same code, call it `my_talk`, is used at both parties. The convention for using `my_ talk` should be

```
% my_talk IP-address port-number
```

where `IP-address` and `port-number` are the coordinates of the party you wish to talk to. Test your talk application on two hosts—open two shells, one of them remote—and record the interaction using `script`. In this instance, you are talking to yourself. At the start, make sure that `my_talk` is running on both hosts before initiating actual text messaging. This is to prevent potential system crash resulting from the other party not being online when you type a message. We will deal with these issues separately.