CS 536 (Park) **Assignment VII** **Due: Nov. 4 (Wed.), 1998**

PROBLEM 1

Read Sections 6.1, 6.2 from P & D.

PROBLEM 2

Extend the client/server application of Problem 2 of Assignment V such that client and server processes are allowed to run on different hosts and communication is achieved via socket calls invoking UDP/IP. Thus the FIFO IPC structure needs to be replaced by the appropriate networking calls. Note that UDP is a connectionless (i.e., stateless), unreliable transport protocol. Test your application using 4 machines in the Xinu Lab, one on which the server runs, and three on which client processes run. Assume the server listens on a well-known port, say 10101. A client should accept two command-line arguments

% `remote_exec` *IP-address command*

where *IP-address* is in dotted decimal form. To prevent indefinite waiting from occuring when packets are lost or the server is down, the client, upon making a request to the server, should set an alarm to go off in x msec. When SIGALRM is caught, the client should resend the request with the entire procedure repeating at most 5 times. On your host machine, open four windows, one for each separate host, running the server on one machine and three copies of the client on the other machines. Test the system by giving `host`, `date`, and `ls -ld` as arguments to the clients (run the three commands consecutively on each host). For the first set of runs, set $x = 5$ sec. For the second set of runs, set $x = 50$ msec and compare the two trace behaviors. In addition to the code and script files, hand in a screen dump which shows the contents of the four windows.

PROBLEM 3

Reimplement Problem 2 of Assignment VI with socket calls invoking TCP in place of FIFO IPC. The barrier server process as well as all client application processes are to be run on separate hosts.

The client and server processes are to implement a slight modification in their interaction, namely, in addition to printing their process ID and loop count, they should also print their hostname, time of day (using the `gettimeofday` system call) at *msec* granularity, and a certain time value received from the barrier server. The latter is obtained in the following way. The `barrier` function call, in addition to sending the default information, also sends to the barrier server the time of day value that the client had just printed before making the `barrier` call. The barrier server, when handling synchronization, also determines the maximum time of day value received from all clients during each round (i.e., a new maximum for every round), and returns this maximum time of day value as part of the go-ahead message to the clients. Thus for every synchronization round, each client is informed of the *slowest* process (well, the time value of the slowest process just before calling `barrier`) for that round and this value is printed by all the processes.

Note that you will now have five separate terminal outputs for each application process, and the output trace will clearly indicate whether synchronization has been progressing correctly. Make a screen dump of all your window contents and hand it in in addition to the other parts.

PROBLEM 4

We have discussed the intrinsic problem of reaching consensus or agreement among two logical but untrusting parties on a common course of action using a protocol. In the case of TCP's connection establishment, why is it then that the two end-points come to an "agreement" (or so it seems in practice) about reaching the *Connection Established* state via a three-way handshake? When can this fail, i.e., when can the three-way handshake fail to set up a connection in the sense of putting both end stations into the *Connection Established* state? Does it matter?

Give a similar analysis of the plague of "acking an ACK problem" and its possible manifestation in the connection termination phase of TCP.