##
Internetworking Assignment 1

There are two programs following, one client and one server that are example
programs that use the Internet addressing scheme for interprocess
communication. This addressing method allows communication between
processes running on different machines. For the present, you will
have to settle for running your client and server on the same machine,
but that will change.

Copy the programs to your accounts and get them running.
The first thing you should do is make sure that the machine
address in clientin.c is correct for your machine. In the example, the
address is 153.90.192.3, but the correct internet address for your host can
be found in the file /etc/hosts. Look and see what it is. Also, you
may want to choose a port number other than 32351. You can choose any
value between 1023 and 65535, but something larger than 8000 is
safest. If you don't choose a different number, you could potentially
have a conflict with someone else using the same port, although that
should be unlikely.

The application is executed by running the server in the background with
something of the form ``server_in &''. Then execute the client.
Once the program is running, modify it to a simple client--server
protocol, where the server waits for a message and then sends a response,
which the client processes. For example, this could be any simple exchange
of information -- ``how are you'', ``I'm fine'', ``what's new'', etc.
When the client is done, it sends a message indicating that this is
the case and quits. In this particular situation, your server should
also quit or be killed. The client and server should print out any
received messages to stdout in a form such as

"Server received: ....."
or
"Client recieved: ....."

Whatever you do, DON'T LEAVE YOUR SERVER PROCESSES RUNNING.

If run a job in the background, it will not be terminated when you log
out. Use the ps command to list your processes. If
you have any unnecessary processes running, use ``kill -KILL pid'' to get
rid of them. If you fail to do this, you will be subjected to scorn and
ridicule by the System Manager. Worse than that, I
will be subjected to scorn and ridicule by the System Manager. This
cannot be emphasized enough. If you have a hard time remembering this,
alias logout to a script that runs ps or reminds you to kill jobs.

If you are using X-windows, you can run each process in a different
window instead of using background execution.

/* ==> client_in.c
 * Simple client model for Internet communication using stream sockets.

```
 * This program simply accesses a server socket and writes a few messages.
 * Then it closes the socket and terminates.
 * ======================================================================
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

void main()
{
   int     sock, addrsize;
   struct  sockaddr_in addr;
   char    buf [80];

   sock = socket(AF_INET, SOCK_STREAM,0);    /* open a socket */
   if (sock == -1)
   {   perror("opening socket");
       exit(-1);
   }

   /*
    * Bind an Internet address to the socket.
    */

   addr.sin_family = AF_INET;
   addr.sin_port = htons (32351);
   addr.sin_addr.s_addr = inet_network ("153.90.192.3");
   if (connect(sock, &addr, sizeof (struct sockaddr_in)) == -1)
   {
      perror("on connect");
      exit(-1);
   }

   /*
    * Write a null terminated message and receive the reply. Note that
    * a single receive may not work perfectly, but it is OK for a simple
    * case.
    */

   send (sock, "client calling server, do you read me", 38, 0);
   recv (sock, buf, 80, 0);

   /*
    * Do a shutdown to gracefully terminate by saying - "no more data"
    * and then close the socket -- the shutdown is optional
    */

   if (shutdown(sock, 1) == -1)
   {
      perror("on shutdown");
      exit(-1);
   }
```

```
  printf ("Client is done\n");
  close(sock);
}

/* ========================================================> server_in.c
 * Generic sort of server process the Internet.
 *
 * This is an extremely simple use of sockets for communication.
 * The server opens a socket and then reads messages and prints them out
 * without further ado until the client closes the socket.
 * ====================================================================
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

void main()
{
  int    sock, clientsock, mlen, addrsize, msgct, chc, chct;
  struct sockaddr_in addr;
  char   ch, buf[80];

  sock = socket(AF_INET, SOCK_STREAM,0);     /* create a socket */
  if (sock == -1)
  {
    perror("opening socket");
    exit(-1);
  }

  /*
   * Bind a name to the socket.  Since the server will bind with
   * any client, the machine address is zero or INADDR_ANY.
   */

  addr.sin_family = AF_INET;
  addr.sin_port = htons (32351);
  addr.sin_addr.s_addr = htonl (INADDR_ANY);

  if (bind(sock, &addr, sizeof (struct sockaddr_in)) == -1)
  {
    perror ("on bind");
    exit (-1);
  }

  if (listen(sock,1) == -1)              /* make socket available */
  {
    perror("on listen");
    exit(-1);
  }

  /*
   * Wait for a client to connect.  When this happens, print out a
```

```
   * message.
   */

  addrsize = sizeof(struct sockaddr_in);
  clientsock = accept(sock, &addr, &addrsize);
  if (clientsock == -1)
  {
    perror("on accept");
     exit(-1);
  }

  printf("connection made with client %s", inet_ntoa (addr . sin_addr));

   /*
   * Receive and print a client message where a null character terminates.
   * Note that a single receive may not work in some cases, but is OK for
   * a simple example.
   */

  mlen = recv (clientsock, buf, 80, 0);
  send (clientsock, "Got your message", 17, 0);

  printf(" Server - all messages read - connection being closed\n");

   /*
   * Close the client socket and also the server socket
    */

  close(clientsock);
  close(sock);
}
```

##

Lab #2

For this lab, you need to build a very simple client-server operation
using the Unix stream capabilities. This is described in the early part
of the handout. That is, you are going to communicate between two
programs, but they are going to reside on the same host. This is
a fairly easy thing to do, but small mistakes can be inordinately difficult
to find in communication software. Be careful!

You are to build a client and server pair, to implement your own little
directory service. Suppose that the server has the following database

```
  0      Bob
  3      Anne
  5      Barb
  7      Ray
  9      Denbigh
 10      Terri
104      John
```

This can be in a file that is read, or it can be a static array in your program - I really don't care about the niceties at this point.

The client is to read a request, which is a numeric address to be sent to the server. The server looks up the matching name and send it back to the client to be printed out. If the address isn't found, the server should send back an error message. For example, "Address not found".

You need to submit the source code for the server, the source code for the client, and a sample run looking up five addresses, one of which must be false, so that the server returns an error message.

Use the script command to capture the run session to turn in.

In order to test a program like this, you will either have to use an X-terminal (or facsimile thereof) and run the client and server in separate windows. Or, run the server in the background and then run the client. CAUTION: DO NOT LEAVE PROCESSES RUNNING THAT ARE UNNECESSARY!! Use the kill command to kill processes. If you don't know the pid, use the ps command to get a list of currently running processes.

##

Lab #3

In this lab, you will convert your previous program to work with the Internet protocol family. Use the inet stream services and build a client and server to perform the same functions that you did before. Determine the Internet address of the machine your server uses and put it in the connect structure. You can use any port that you want, between 13000 and 65535 safely. Note that if you happen to pick the same port as someone else, you could have a conflict and a server may fail when it tries to open the port, but the likelihood is small.

The only logical change you need to make is to implement your exchange of messages as a protocol. The protocol will be structured as follows (note, a string of digits followed by a b means a binary number):

Requests:

byte	content
0	request code, 0000001b = name, 00001001b = number
1-n	request data
n+1	end-of-request, 00000011b (ETX)

Replies:

byte	content
0-n	reply data
n+1	end-of-reply, 00000011b (ETX)

Note that the requests allow for both name and number matching, so you need to also be able to match a name. Nothing fancy, an exact match is required.

Also note that the character string can contain any type of byte oriented data, including binary representations of numbers. For example,

```
short     val;
char      *message;

message = &val;
send (sock, message, 2, 0);
```

##

Lab #4

The next thing that you should add to your repetoire of network programming skills is the use of the utility functions that make it easier to write robust general purpose programs. Two of these are:

```
gethostbyname
getservbyname
```

Convert your previous program to a form so that the server accepts a service name as an argument, and the client accepts both the service name and the host name as an argument. For example,

```
servername cs440-01
clientname cs440-01 host.subdomain.edu
```

The possible services are listed in /etc/services, and are essentially a mapping of names to port numbers. If you look in /etc/services, you will see that there are 8 services reserved for this class. You can use any of them.

The idea is that you could run the server anywhere, and by giving the correct name in the arguments, the client will connect to the server on that host. Since we only have on host, that isn't very useful at this point, but it will be later. gethostbyname will convert a name into a structure that contains the IP address of the host. Similarly, getservbyname will convert a service name to a structure that contains the port number. Take a look at the man pages for these system calls to learn more about them. It's good practice.

Functionally, the only thing you should change is that the server should print out a message for each connection with the following general form:

```
Request received from host_name
```

where host_name is the name of the host making the request. To get the request from the address, use the gethostbyaddr system call.

###

Lab #5

The next thing you should do to your server, is modify it to handle more than one client at a time. This could be quite difficult if you try to multiplex several clients by keeping track of the state of each client in arrays and global variables. You have no control over which client is going to send a request next, so you need to be ready to handle any one of them.

It is easier to create a copy of the server to handle each client. The server itself stays simple, because it can be written to handle just one client, and all that is needed is a mechanism to create the copies. The Unix fork command does just this. An example of using the fork command shown below. This example has been stripped down to not do much, but it is a working example of a server that can support multiple clients.

CAUTION!!
The fork command can be quite dangerous. If you make a mistake and the child process for the client also does a fork, you could cascade a lot of processes and make both yourself and others unhappy. Your fork code should always follow the general form of:

```
pid = fork ();
if (pid < 0)
{
  fprintf (stderr, "Error in fork call\n");
  close (client_sock);
}
if (pid == 0)
{
  /* Do the child stuff - and NO FORK CALLS */

  close (client_sock);
  exit (0);   /* THIS MUST BE HERE */
}

/*
 * Do the parent stuff, including a non-blocking wait.  If you
 * don't do this, you create zombies and that is BAD, BAD, BAD.
 */

wait3 (NULL, WNOHANG, NULL);
```

When the child process finishes doing its stuff, it must exit. The parent process never exits, and in fact, you probably want the server to be an infinite loop, but don't forget to kill it.

Hint: Build a small version of the server that does the fork and then the

child process outputs a message and exits, and a client that does a connect and then exits. Make sure that you get all of the right sockets handled.

DO NOT experiment to see what will happen if you do this or that with fork. If you don't know, ask first. This is one place where you can create a lot of problems for the system if you make a mistake, and we don't need any more problems.

!!!

The protocol you have been using is a classic stop and wait protocol, because the clients send each request as a separate message and get the results. This is what is called a stateless protocol, because the server doesn't have to keep any state information around about what the clients have done before. This is a common paradigm in client-server computing, but it is interesting to consider protocols where that is not the case.

An example would be server that handles remote cash registers. A cash register opens a session with the server and then supplies a sequence of codes for products. The server returns the price of each one, and also keeps a running total of purchases. When the client (cash register) closes the session, the server returns the total cost. This is how point-of-sale terminals work, especially those with scanners. This way, the database only has to be kept one place

This needs to be kept relatively simple, so ignore things like sale prices, returns and the like. The client can send only the following commands:

Open
Close
Item

The format of the protocol is:

byte	content
0	type (0 = open, 1 = close, 2 = item)
1-n	other

The Open and Close commands have no parameters, but the Item command has the following format

byte	content
1-10	UPC code
11-12	number of items

The server returns a 0 if the open command works. For the Close command, the server returns a 4 byte integer in network byte order, which is the total, multiplied by 100.

For the Item command, the server returns:

byte	content
0	code - 0 = OK, 1 = error
1-n	null terminated string containing the error or

1-4 the cost of the item(s) multplied by 100
5-n the text description of the item

For example, given the following database,

5010040268Healthy Choice Split Pea and Ham Soup 1.30
7615020219Act II Light Microwave Popcorn 0.50
4460000628Formula 409 All Purpose Cleaner 2.29

the result of sending UPC code 5010040268 with a number of items of 3 would result in a code of 0, an amount of 390 (3 times 1.30 multiplied by 100) and a description of Healthy Choice Split Pea and Ham Soup.

The only possible errors are a protocol error, or a UPC that's not in the database.

Build your program to handle multiple cash registers simultaneously, and use the database stored in ~harkin/public/inst/cs440/asgn/scanner.db. You will have to be able to run the server and at least three clients at one time. The server should print out enough information to be able to show that all of this is happening at once. More on this later.

```
/* ================================================================

   This simple example demonstrates a server that processes accepts
   but creates a child process to handle all of the communication
   with the client.  You have to remember that the child
   inherits an environment that is virtually identical to the
   parent, including all file descriptors.  So the child process
   has access to the sockets created by the parent.

   The function of this server is to allow the client to have
   access to certain operating system functions that might not
   otherwise be available.  This is the sort of thing a distributed
   operating system or database might do, although, hopefully, in a
   more robust and complete way.

   This server also function differently in that the main program starts
   and spawns a child process that is the server.  That way, the
   server does not have to be run in the background.

    Syntax:
      fork_server port

  ================================================================ */

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/wait.h>
```

```
#define       NAMELEN       64
#define       CMDLEN        32

typedef int   BOOLEAN;


void client_wait ();
void client_server ();
BOOLEAN legal_name ();


main
   (
      int argc,
     char *argv []
   )
{
   int               child_id;
   int               status;        /* the returned child status */
   int               l_sock, chct, addrlen;
   char               ch, buf[80];
   char               hostname [NAMELEN];
   struct             hostent *hostp;
  struct sockaddr_in addr;


   /*
    * Check the args - client_server port
    */

  if (argc < 2)
  {
     fprintf (stderr, "Syntax: fork_server portnum\n");
     exit (0);
  }

  /*
   *  Open a stream socket for public access and listen on it
   *   for new clients.
    */

  if ((l_sock = socket (AF_INET, SOCK_STREAM, 0)) < 0)
  {
     fprintf (stderr, "Could not open the socket\n");
     exit (0);
  }

 bzero (addr, sizeof (addr));
 addr . sin_family = AF_INET;
 addr . sin_port = htons (atoi (argv[1]));
 addr . sin_addr . s_addr = htonl (INADDR_ANY);
 if (bind (l_sock, &addr, sizeof (addr)) < 0)
 {
    fprintf (stderr, "Could not bind the address\n");
```

```
      exit (0);
   }

   if ((listen (l_sock, 1)) < 0)
   {
     fprintf (stderr, "Unable to listen\n");
     close (l_sock);
     exit (0);
   }

   /*
    *  client_wait does the work.
    */

   client_wait (l_sock);

   fprintf (stderr, "cmd_server terminating\n");
   close (l_sock);
}



/* ============================== client_wait ============================

   Function:   Accept client_wait requests and spawn client servers.

   Usage:      client_wait (l_sock)

   Arguments:  int l_sock       - the advertised socket

   Return:     None.

   Notes:

  ======================================================================== */

void client_wait (l_sock)
   int l_sock;
{
   int              child_id, client_sock;
   struct sockaddr addr;
   int              addrlen;

   /*
    *  Wait for connections.  If there is a failure, simply go on.
    *  When a connection is made, fork a child process to do the work.
    */

   do
   {
      /*
       * When a child process dies, it must be recognized by the parent, or
       * the child hangs around as a zombie.  This is handled with a wait.
       * In this case a plain wait () won't work, because we don't want to
```

```
       * wait forever.  Instead, use wait3, which clears the child
        * termination signal, but if there isn't anything, will return if
        * the WNOHANG flag is set.
        */

      wait3 (NULL, WNOHANG, NULL);
      if ((client_sock = accept (l_sock, &addr, &addrlen)) < 0)
       {
         fprintf (stderr, "Failed connection with client\n");
          continue;
       }

       /*
        *  Create the child process to do the actual client communications.
        *  Again, the child sees a return of zero and the parent sees the
        *  child id being returned.
         */

      if ((child_id = fork ()) < 0)
       {
         fprintf (stderr, "Failure in creating child process\n");
         close (client_sock);
       }
       else
         if (child_id == 0)              /* Do the client stuff */
            client_server (client_sock);

       /*
        *  The server (parent) has no more use for the client
        *  socket and can close it.  Remember that the child
        *  client server still has it open.
        */

      close (client_sock);

   } while (1);

   /*
    *  The server never terminates, so don't worry about anything here.
    */

   return;
}



/* ============================== client_server ============================

   Function:   Handle client communications.  The task of the server
               is to execute simple commands over the protected
               directory, which would normally be prohibited.

   Usage:      client_server (client_sock)
```

```
   Arguments:  int client_sock           - the advertised socket

   Return:     None.

   Notes:

  ========================================================================*/

void client_server (client_sock)
   int client_sock;
{
   char buf [CMDLEN];
   int buflen;
   static char *ack = "\06";
   static char *logout_msg = "Pleased to be of service - Goodbye\n";



   /*
    *  First, make sure that the client address is acceptable. When the
    *  connection is successful, the client should send it's name.
    */

   buflen = recv (client_sock, buf, CMDLEN, 0);
   buf [buflen] = '\0';
   if ( ! legalname (buf))
   {
      fprintf (stderr, "Attempt to access by an illegal client named ");
      buf [buflen] = '\0';
      fprintf (stderr, "%s\n", buf);
      exit (0);
   }

   if (send (client_sock, ack, 1, 0) < 0)
   {
     perror ("On connection acknowledgement");
      close (client_sock);
      exit (0);
   }

   /*
    *  Set it up so that standard output for this client server
    *  will be the client socket.  Then whatever output would
    *  normally go to standard output will go back to the client.
    *  dup2 does this by closing stdout (descriptor 1) and making
    *  it a duplicate of the client socket.
    */

   dup2 (client_sock, 1);


   /*
    *  As long as the client is interested, accept communications,
    *  and execute the desired commands.  At the end of each
```

```
   *  command, return an end-of-command character to the client.
   *  A command of "logout" indicates a desire to terminate the connection.
    */


  do
  {
     buflen = recv (client_sock, buf, CMDLEN, 0);
     if (strncmp (buf, "logout", 6) == 0)
     {
        /*
        *  Send a nice goodbye and send an ack just in case.
         */

        send (client_sock, logout_msg, strlen (logout_msg), 0);
        send (client_sock, ack, 1, 0);
        break;
     }

     buf [buflen] = '\0';      /* Supply a terminator */
     system (buf);
     send (client_sock, ack, 1, 0);
  } while (1);

  /*
   *  Terminate the client socket and the process
    */

  close (client_sock);
  exit (0);
}



/* ============================== legal_name ======================

   Function:   Check the legality of a client address/name.

   Usage:      if (legal_name (client))

   Arguments:  char *client       - the client name

   Return:     TRUE if legal, FALSE otherwise.

   Notes:


  ========================================================================== */

BOOLEAN legalname (client_name)
  char *client_name;
{
  char client [NAMELEN], *cp;
  FILE *clientfile;
```

```
  BOOLEAN found;

  if ((clientfile = fopen ("client_file", "r")) == NULL)
  {
     perror ("Opening the client file");
      exit (0);
  }

  found = FALSE;
  while (! feof (clientfile))
  {
      /*
       *  Get a line from the file, replace the newline, if any, with a
       *  null and then match it to the client name.
       */

      fgets (client, NAMELEN, clientfile);
      cp = index (client, '\n');
      if (cp != NULL)
          *cp = '\0';
      if (strcmp (client, client_name) != 0)
          continue;

      /*
       *  Name is located.  Close the file and return.
       */

      fclose (clientfile);
      return (TRUE);
  }

  return (FALSE);         /* No where to be found. */
}
```

##

Lab #6

One thing that's interesting to do, is talk to some existing server.
For example, the NFS server, the ftp server or the telnet server. This
would be a cruel thing to do. One that is easier to talk to is the
Post Office Protocol (POP) server. Some of you are probably familiar
with the POP server. If you start a mail client from a PC and tell it
that your mail is stored on somehost, it will contact to the POP server on
somehost and it will download your mail to the PC.

It's basic function is just that. If you connect to the POP server, it
expects to download the contents of a mail file to you. Rather than
repeat a lot of stuff about the POP protocol, I am going to run off
a copy of an article that contains lots of useful stuff about POP and
IMAP, which is another protocol.

Your assignment is to create a client that will use the POP server
to manage your mail file. This doesn't have to be anything fancy, unless
you want to put in the extra effort. A simple command line interface that
lets a person list their messages and look at or delete selected messages
would be great. You could of course, just open the mail file and do this,
but what fun would that be.

For this assignment, use port 110, since that's where the POP server is
running. You might notice in /etc/services that port 109 is reserved for
something called "pop2". This is actually an IMAP server.

You might think about how you would do this from a remote system. That is,
if the client you wrote would connect over the network to somehost and get the
data and display it at the remote host. This really wouldn't be too hard
to do from a PC or another host, since the network programming calls will
work just as well from any other host that supports them.

You may have to do some experimenting here to figure out exactly what the
POP server returns in its messages. Welcome to the real world.

You should turn in a couple of test runs showing lists of messages, and
the interaction that displays and/or deletes messages.

One problem here is that you need to send a password to the pop server.
If you type the password on the terminal, it will be plainly visible to
everyone, and it is equally dangerous to hard code it in a program. The
following is a short program that turns off the echo in order to read
a password.

```
/* ============================================================> noecho.c
 *
 * Example of a program that turns off the echo on stdin to read
 * a password and then turns it back on.
 *
 * ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <termios.h>

void main ()
{
    char            name [32], pass [32];
    struct termios  tios;

    /*
     * Get the name with echoing
     */

    printf ("Enter your name: ");
    scanf ("%s", name);
    printf ("The name is %s\n", name);
```

```
    /*
     * Output the prompt and then toggle echoing.
     */

    printf ("Enter your password: ");

    /*
     * Get the terminal characteristics
     */

    if (tcgetattr (0, &tios) < 0)
    {
       printf ("Could not get terminal attributes\n");
       exit (-1);
    }

    /*
     * Change echo to off and reset.
     */
    tios . c_lflag ^= ECHO;           /* echo off */
    tcsetattr (0, TCSAFLUSH, &tios);

    scanf ("%s", pass);

    /*
     * Reset the property and reset the terminal line.
     */

    tios . c_lflag ^ ECHO;           /* echo on */
    tcsetattr (0, TCSAFLUSH, &tios);

    printf ("\nThe password is %s\n", pass);

}
```

##

Lab 7

This is the first part of a two part assignment that uses other machines
to do work for you. In this part, you are going to use the UDP protocol
instead of TCP, and you are going to try out a new system call.

When using UDP, you specify that the socket is of type SOCK_DGRAM, and you
must use sendto and recvfrom instead of send and receive. Look at the man
pages on these calls, and you can look at the programs following.

The important thing to remember is that UDP communications are completely
connectionless, so the only clue that the receiver has as to the identity
of the sender is in the address portion of the receive call. If a message
is to be returned, the receiver must get and use that address.

In this program, instead of choosing your own port, use one of the
ports established for this class. Each port has both a tcp and a udp

version, so they can be used for either protocol. This is what is meant by the protocol-port addressing pair.

So what is this program going to do. Again, we will stick with a client server model. The client side will be executed with:

execute host command

where host is the name of the host where the server must reside, and command is a command to be executed at the host. The remote client will send the command to the server on host and then print anything that is returned at stdout.

The server will receive the string containing the command, which could be any legal Unix shell command and use the "system" call to perform the execution, redirecting the command output to go back to the client.

For example, if

execute somehost.thisdomain.edu ls

is entered, the server would execute the ls command, and send the result back to the client, where it would be output to stdout. There are a number of issues here.

How does the "system" call work? Check the man pages, its easy.

If the server were running on the machine as a root process, how would it know what directory to "ls"?

The solution is to to make the server handle a login for the user on the other machine. This also provides security so that all sorts of riffraff don't use your machine for executing things. We will ignore this problem,

How does the child process redirect the command output to the client? Well, this does present problems. Basically, you need to change the default assignment for stdout for the commmand, but this could go badly. If you attempt to store all of the output in memory, or even a file, you could run into resource availability problems. What is preferable is to direct the output to the network so that it is sent directly to the client process. While this isn't directly possible, you can play some games to make it work.

First, you need to create a pipe with a pipe call.

```
int     pd [2];
if (pipe (pd) < 0)
{
  fprintf (stderr, "Error opening pipe\n");
   exit (-1);
}
```

A pipe is a buffer with a read descriptor (pd[0]) and a write descriptor

(pd[1]). After the pipe is opened, you can read and write the pipe, and the processes doing so are synchronized so that if the pipe is full, the reader is blocked on a read and if the pipe is empty, the writer is blocked on a write. The two descriptors cannot be switched.

```
read (pd[0], buf, n);          write (pd[1], buf, n);
```

Pipes can be thought of a being like a socket, except that if two processes are using a pipe, they must be related in the parent-child sense.

The dup call can be used to create a duplicate of any file descriptor, and it will get the lowest numbered file number available.

If the process being created has any output to stdout, you can redirect it to the pipe by doing the following.

1. create a pipe (pipe (pd))
2. close file descriptor 1, which is stdout (close (1))
3. dup pd [1], the output side (newfd = dup (pd[1]))

Because stdout was closed and stdin is still open, the newfd will be 1, and it will be assigned to the same i/o structure as pd[1], which is the write side of the pipe. Any attempt to output to stdout, will actually write to the pipe. If the server process reads the pipe, it reads what would normally go to stdout and it can send it back to the receiver.

When the process is started to run with the system call, anything it writes will go to the pipe. The server child can read the pipe and forward the data to the client for output. There is one problem, the pipe has a limited size, so if the system command generates a lot of output, it will block on the pipe, and the server child is blocked waiting for the system call to end. This can all be avoided by using asynchronous I/O which is the Unix version of interrupt driven I/O.

Unix will let you capture signals, which are interrupts that have been handled by Unix, but then passed to a process. The best description is probably an example, which follows.

```
/* ================================================================
 * Example of using a pipe to handle stdout.  Under OSF/1, this must
 * be compiled in the following way:
 *
 *      cc  prog.c -lsys5 -o prog
 *
 * with the System V library loaded.  You need the  System V library
 * as it defines the semantics for the signal handling.  For some reason,
 * the BSD library doesn't work right.
 *
 * ================================================================
 */

#include <stdio.h>
#include <signal.h>
```

```
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

void piperead ();

int   dd, pd[2];
void    (*oldsigio) ();

void main (int argc, char *argv[])
{
   int      flags, ct;
   char     buf [80], ch;

   /*
    * Open a pipe, then close stdout and using dup, set up pd[1] to
    * be the same file descriptor.
    */

   if (pipe (pd) < 0)
   {
      fprintf (stderr, "Couldn't open pipe");
      exit (0);
   }

   close (1);
  dd = dup (pd[1]);

   /*
    * Make I/O on pd[1] asynchronous, which means this program can get
    * the signals from the OS.
    */

   flags = fcntl (pd[0], F_GETFL, 0);
   fcntl (pd[0], F_SETFL, flags | FASYNC | O_NDELAY);

   /*
    * Tell the system to catch the SIGIO signal.  Technically, we could get
    * in trouble, since it will catch all asynchronous I/O, but that shouldn't
    * be a problem here.
    */

   oldsigio = signal (SIGIO, piperead, -1);

   /*
    * Do the system command, which if it writes to stdout (file 1), it
    * will come back to the process via the signal handler.  Note that the
    * "system" call will not return until the created child process completes
    * and terminates.
    */

   system ("ls");

   /*
```

```
    * Reset signal to be safe and clean up.
    */

   signal (SIGIO, oldsigio, -1);


   close (pd[0]);
   close (pd [1]);
   close (dd);
}

/*
 * Asynchronous signal handler.  Reads anything waiting at pd[1] and
 * prints it out.  It only executes when the system knows that something
 * is waiting.
 */

void piperead ()
{
   char     buf [80];

    /*
    * Read the pipe and output to the terminal.  Stderr has to be used
    * because stdout has been closed.  Note, this may output a strange
    * line at the end because it doesn't check for a line containing
    * a new line alone.
    */

   if (read (pd[0], buf, 80) > 3)
      fprintf (stderr, "%s\n", buf);

   return;
}
```

Asynchronous I/O has two parts. The first part is specifying that a specific I/O descriptor is to be handled asynchronously by using the fcntl call. You are telling the operating system that you want to take more control of the I/O for this descriptor by not waiting for the OS to decide when it is time for you to do something with the data. Instead, you want to be informed when input is ready or output is done, so that you can act accordingly. In this case, that means that you want to know when there is input ready in the pipe, so that it doesn't fill up and block before the "system" call returns.

Next, the system has to be told what to do when I/O is ready, and that is done by indicating that you have a handler for the signal called SIGIO. Unix does not have a special signal for each I/O device. Instead, all I/O interrupts for your program that are set up to be asynchronous cause the same signal, and so they all have the same signal handler. There are calls to allow your handler to decide what has happened. In this case, there is only one asynchronous file, so that isn't a problem. Other signals are for pressing the control-C key (SIGINT), bus errors (SIGBUS), floating point errors (SIGFPE) and so on (up to 64 signals on modern Unix systems).

This is, in large part, the code for the child process, except that you need to get the data coming in from the pipe sent off to the client.

This assignment has quite a few new things in it for most people - pipes, signal handling, datagrams and asynchronous I/O. But the applications are simple. Implement things as shown and read the man pages. In future labs, you will do some things that are more complicated.

```
/* ================================================> dg_client.c
 * Simple Unix domain datagram type peer process.  It comes up and
 * sends messages to another process.
 * ============================================================
 */
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/un.h>

void main(argc, argv)
   int argc;
   char argv[];
{
   int      flag, sock, addrsize, saddrsize, response;
   struct   sockaddr_un addr, saddr;
    int      getmsg(), done, len, rlen;
   char     socketname[20], buf[80];

   sock = socket(AF_UNIX, SOCK_DGRAM,0);     /* create a socket for input */
   if (sock == -1)
   {   perror("opening socket");
       exit(-1);
   }

   addr.sun_family = AF_UNIX;
   strcpy(addr.sun_path, "dgclient");        /* create and bind a name */

   addrsize = strlen ("dgclient") + 2;
   if (bind(sock, &addr, addrsize) == -1)
   {  perror("on bind");
      exit(-1);
   }

   /* Send messages to the server to be processed and returned.
    * Note that the null message is sent to kill the server.
    */

   strcpy(addr.sun_path, "dgserver");
   addrsize = 10;
   done = 0;
   do
   {
```

```
      printf ("Enter a short message to be sent, return to halt\n");
      gets (buf);
      len = strlen (buf);
      if (len < 2)
         done = 1;
      if(sendto(sock, buf, len, 0, &addr, addrsize) < 0)
         perror("on client write");
      saddrsize = 32;
      rlen = recvfrom (sock, buf, len, 0, &saddr, &saddrsize);
      printf ("client: %s <s returned from %s>\n", buf, saddr.sun_path);
   } while (! done);

   /* Close the socket and unlink the socket name
    * which means delete the file representing the socket
     */

   shutdown(sock, 2);    /* not receiving or sending anymore */
   close(sock);
   unlink("dgclient");
}


/* ================================================> dg_server.c
 * Simple datagram server for the Unix domain sockets.  It opens
 * a sockets and then performs a simple transposition on
 * incoming messages before sending them back.
 * ============================================================
 */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/un.h>

void main()
{
   int     s, addrsize, nbyte, ct;
   struct sockaddr_un addr;
   char    buf[80], tc;

    /*
     * Make sure that the socket file is gone and then create a socket.
      */

   unlink ("dgserver");
   s = socket(AF_UNIX, SOCK_DGRAM, 0);
   if (s < 0)
   {   perror("opening socket");
       exit(-1);
   }
   strcpy(addr . sun_path, "dgserver");         /* create and bind a name */

   addr . sun_family = AF_UNIX;
```

```
  addrsize = strlen ("dg_server") + 2;
  if (bind(s, &addr, addrsize) == -1)
  {  perror("on bind");
     exit(-1);
  }

  /*  Wait for a message to come in through the socket.  The
   * address of the sender is contained in the addr structure
   * so that a return can be sent.  Note that this is an infinite loop
   */

  do
  {  addrsize = 32;        /* maximum length of address name */
     nbyte = recvfrom(s, buf, 80, 0, &addr, &addrsize);
     if (nbyte > 0)
        printf("server: %s <is received from %s>\n", buf, addr . sun_path);
     else
        break;

     /*
     * Encode the string by exchanging odd and even pairs.
      */

     for (ct = 0; ct < nbyte-1; ct += 2)
     {  tc = buf[ct];
        buf[ct] = buf[ct+1];
        buf[ct+1] = tc;
     }
     nbyte = sendto (s, buf, nbyte, 0, &addr, addrsize);
  }while (1);

  /* close the socket and unlink the socket name
   * which means delete the file representing the socket
    */

  printf ("Datagram server terminating\n");
  shutdown(s, 2);    /* not receiving or sending anymore */
  close(s);
}
```

##

Lab 8

Remote Execution, Part 2.

Remote execution can be a very valuable service indeed. For example, if you consider all the computers on this campus alone and the amount of time each day that the CPU is idle, it adds up to probably more computing power than was cumulatively available in the entire world before most of you were born. The problem is how to take advantage of all of that computing power, and its not easy.

First, you have to be able to separate problems into pieces that can be run independently of each other - this called parallelization. It is difficult to do this, because most problems have lots of sequential character - things that have to be done in some order and this can't be parallelized. Then you have the problem of getting software servers running on all of the machines so that you can send them work to do. Some narrow-minded folks aren't to happy with the idea that someone else may have access to their computer, even when they're not using it. Also, you need to make it reasonably easy for people to distribute their problem parts out to remote machines or they won't do it. None of these problems have been completely solved yet, but people are working on it

What does all of this mean to you, you ask cautiously. As an example of this sort of problem, we know that sorts of various kinds are parallelizable, For example, a merge sort could easily be distributed across N machines, where each server sorts 1/N'th part of the list, and then one machine performs the merge. Other examples of parallelizable problems are matrix multiplication, numerical integration and differentiation and lots of database problems.

For this assignment, you are going to use sort servers that sort a list of values on demand, and then return the sorted list to the caller. The number of items to be sorted will always be unknown until the list arrives. The sort method the server uses doesn't matter.

The organization of the clients and servers in this problem is that there will be one client and several servers, which is different than what you have done in the past. That means that there has to be a well-defined way of interacting with the servers if your program is going to work. The preferred strategy is to have the clients make a call like this:

```
sort_rpc (list, num)
```

where list is the list of values and num is the number of items in the list. The name sort_rpc includes rpc to represent remote procedure call, which is what this type of activity is called. In essence, you are going to create a distributed sorting mechanism that is transparent to the function that makes the call. As far as it knows, the sort is done locally.

sort_rpc has to perform the following tasks:

- find sort servers to use
- divide the data and send to the sort servers
- wait for the servers to finish
- merge the sorted sublists into the complete list
- return the sorted list

Broadcasting

How do you find all of the sort servers that are available. In the past, there has always been just one, waiting patiently at a particular port on a particular host. If you don't happen to know the host location of the servers, the easiest way is to broadcast a message that all local hosts

can see, but directed to a particular port on each host. If the servers use the same port on every host, they will get the broadcast message and they can send a message indicating that they are available and including their IP address.

In order to send broadcast messages, you need to tell the operating system that this is what you want to do. This is done by setting the socket options:

```
/*
 * Set the socket to handle broadcast messages.
 */

int       sockopt;
sockopt = 1;
if (setsockopt(socket, SOL_SOCKET, SO_BROADCAST,
      (char *) &sockopt, sizeof(int)) < 0)
{
    perror ("on set socket options");
    exit(-1);
}
```

and then use a broadcast address. For the cs domain, the address 153.90.192.0 is the appropriate address to broadcast to all hosts on the local network. Zero in the host portion means that all hosts are to receive and process. For example,

```
addr.sin_port = htons (PORT);
addr.sin_addr.s_addr =  inet_addr ("153.90.192.0");
sendto (mysock, buf, strlen (buf) + 1, 0, &addr, sizeof (addr));
```

will send buf to the server at <udp, PORT> on every host on the local network. Note that broadcasts are always datagrams, because you need a specific IP address to make the connection for a stream.

Timers

When you do something like a broadcast for servers, the responses are asynchronous - not specifically organized in time. That means that some servers might respond very quickly, while other take longer, either because the host is busy or because they are simply slower. In any case, when the broadcast is sent, your program needs to give a reasonable amount of time for the servers to respond, and then not wait any longer. If some server takes 5 minutes to respond, you probably don't want it doing any sorts for you anyway. Therefore, you need to set up a timer that will let you terminate the query-for-server process at a reasonable time.

A timer is a simple signal handling process such as this.

```
/* ============================================================>
 *
 * Example of setting up a unix timer.  The setitimer call starts a
 * timer running and when it expires, it generates a signal.  The
 * process is to set up a signal handler for the particular timer
```

```
 * being used, which is the alarm timer in this case, and then
 * use setitimer to get the alarm clock started.  When it expires,
 * the signal handler gets called.
 *
 * ================================================================
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <signal.h>


void AlarmHandler ();

void main ()
{
   struct itimerval      itimer;
   int                   interval;

   /*
    * Set up the timer mechanism for 5 seconds.
    */

  signal (SIGALRM, AlarmHandler);
   interval = 5;


   /*
    * Set the timer up to be non repeating, so that once it expires, it
    * doesn't start another cycle.  What you do depends on what you need
    * in a particular application.
    */

   itimer . it_interval . tv_sec = 0;
   itimer . it_interval . tv_usec = 0;

   /*
    * Set the time to expiration to interval seconds.
    * The timer resolution is milliseconds.
    */

   itimer . it_value . tv_sec = interval;
   itimer . it_value . tv_usec = 0;

   if (setitimer (ITIMER_REAL, &itimer, NULL) < 0)
   {
      perror ("StartTimer could not setitimer");
      exit (0);
   }

   /*
    * Hang around and wait until something happens.
    */
```

```
  printf ("The timer has started\n");
  pause ();

  /*
   * Note that if conditions indicate that you don't want to wait for
   * a timer to expire.  For example, if in a sliding window protocol
   * you get the ACK for a message, you can kill the alarm by calling
   * setitimer and setting the interval value to 0.
   */
}

void AlarmHandler ()
{
  printf ("The 5 seconds has expired\n");

  return;
}
```

So after sending out a broadcast, you can set a timer and you can receive and process server responses for a period of time, such as 3 seconds, and then you can simply ignore any other responses. Note that this means that you will want to use asynchronous I/O on the datagram socket, since you can't control the sequence of events very well.

The Assignment

I will write a server for this assignment and run it on at least two machines in the cs.montana.edu domain. You need to write the client side that implements the sort_rpc, as well as the test program that reads a file and uses sort_rpc to do the sorting. Believe me, you are getting the easy part of this deal.

Because you only want to have one client/server interface, we have to be very specific about how the data is passed to the server and then back to the client.

The server will listen on <UDP, cs440-06> for broadcasts from clients asking for a response. The requests for server identification will have the form:

```
  CS440sorter?
```

Yes, there is a question mark at the end. You are asking, "Are there any cs440 sorters out there?". The strings must be null terminated.

The server will respond with:

```
  CS440sorter-nnn-mmm
```

where nnn is the ASCII representation of the integer port number that the server is using and mmm is the ascii representation of the ip address

of the host it is running on. Again, they are not going to be three characters long in most cases. For example

"CS440sorter-12500-1722105853

where 12500 is the port and 153.90.192.3 is the IP address. The integer value returned is the IP address but not in dotted decimal form.

Note that the port number returned is the TCP port that the server is listening on, and may not be the same for all servers, or even the same for different renditions of the same server.

After the servers are known, the client will create a TCP connection with each server. The first thing that should be sent is a message of the form:

CS440auth-nnn

Where nnn is an authentication key, in the form of a string which is null-terminated. This more or less lets the server know that everything is synchronized. In this instance, the authentication key will be your first initial and last name. For example,

CS440auth-BKool

The server will return a message containing "OK".

After authentication, the server will be ready to receive data. The data will be floating point and will be passed in the following form:

bytes	data
0-3	number of items,n, following in network byte order
4-7	mantissa of value 0 in network byte order
8-9	exponent of value 0 in network byte order
10-13	mantissa of value 1 in network byte order
.	.
- n*6+3	exponent of nth value in network byte order

I chose floating point intentionally, because floating point values have no network byte order. The problem being that there is no standard internal format for floating point values that is universally accepted. So floats are always a big pain in the neck.

The server will return the data as follows:

bytes	data
0	status as a one byte integer 0 = OK, 1 = communication error, 2 = memory error, 3 = data error
if status is 0	
1-4	number of items

5-n (everything else is identical to
the format of the data sent)

if status is not 0,
1-n error message in ASCII text form and null terminated

If an error occurs during the startup phase, you will get back a text message with an error message in it.

Testing

If you can't connect to at least two servers, some have probably died and need to be restarted - let me know.

Use all of the servers that are available, which should generally be more than 2.

Use a list of at least 100 floating point values and divide them evenly among the servers.

As servers respond, print out a message showing the IP address in dotted-decimal notation, the DNS name of the server and the port that the server is accepting connections on. For example

153.90.192.1 esus.cs.montana.edu 6305

When the data comes back, print the IP address of the server and the status of the result.

The main program should print the list in sorted order - try to minimize the space, such as:

-3.1 4.2 5.6 7.5 13.2 21.6 33.4 34.2 50.3 60.7
100.5 101.6 200.5

...

Data

The following program generates 100 floating point numbers between -100.0 and +100.0. You can modify it to create your list of numbers.

```
/* ============================================================>
 *
 * Generate 100 random floating point numbers.
 *
 * ============================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
```

```
void RandomSetSeed ();
int  RandomGetInt (int, int);
float RandomGetFlt (float, float);

void main ()
{
   int     ct;
   float   val;

   RandomSetSeed ();

   for (ct = 0; ct < 100; ct++)
   {
      val = RandomGetFlt (-100.0, 100.0);
      printf ("%f\n", val);
   }
}

/*
 * Initialize the random number seed to the time of day.
 */

void RandomSetSeed ()
{
   struct timeval     timeofday;
   struct timezone    tz;
   long               seed;


   if (gettimeofday (&timeofday, &tz) < 0)
      seed = 38498757;
   else
      seed = timeofday . tv_sec + timeofday . tv_usec;

   srand48 (seed);
}

/*
 * Generate a random integer.
 */

int RandomGetInt
(
   int      low,
   int      high
)
{
   double    rnd;

   rnd = drand48 ();

   return ((int) (rnd * (high - low) + low + 0.5));
}
```

```
/*
 * Generate a random float.
 */

float RandomGetFlt
(
   float      low,
   float      high
)
{
  double    rnd;

  rnd = drand48 ();

  return ((float) (rnd * (high - low) + low));
}
```

##

Last Lab Assignment

==

Let's play BlackJack

I think everyone may know how to play BlackJack. What? You have never played BlackJack? That's much better. I don't want to always lose money. Anyway I'll explain the rules of our BlackJack later.

Since this is the last cs440 lab assignment, it's relatively easier, especially compared to the previous one. In this lab you are required to just write a BlackJack client program which includes:

-- find the BlackJack server using UDP broadcast;
-- connect to the BlackJack server using TCP;
-- use SELECT to read data from server and to control writes; and
-- construct a Finite State Machine(FSM) to process data.

You can directly borrow the UDP broadcast and the TCP connection stuff from last lab assignment. What you need to change is the port number. The BlackJack server will be listening on port 12003.

The SELECT system call is very flexible. It can do your job in either sync or async mode. Use "man 2 select" to read its man page and refer to the pop_client.c example posted in cs440 newsgroup.

If you read documents on protocol specification you will find that many protocols are specified not only in plain English but also in some kind of formal or semi-formal language. A State Transition Diagram is one of those things used to specify a protocol. In our daily life we can also find a lot of cases where the State Transition Diagram can be applied.

One way to implement a State Transition Diagram is to use a Finite State Machine(FSM) or finite automata. In FSM, what we usually have are

(1) a current state;
(2) an input event;
(3) an action taken based on (1) and (2); and
(4) a next state.

Ok, now, let's recall the rules of the BlackJack game and then go to see how it can be specified with a State Transition Diagram. Because there may be different variations (or versions in our terminology) of BlackJack, we use only the essential part of its rules.

When you want to play BlackJack in a casino, you first can choose to play one or two decks of cards. Then a dealer gives you two cards and takes two for him/herself. Based on the total points of your cards you may want more cards. If and only if you don't want more cards, does the dealer start to get more cards according to his/her total card points on hand.

After the dealer stops, your cards and the cards of the dealer are compared. The one who has more points but less than 21 points will win this game. If the player exceeds 21 he/she loses. If the user is under 21 and the dealer is over, the dealer loses. The dealer must exceed 17 points. The cards and their points are listed below.

Cards are: S=Spade, H=Heart, D=Diamond, C=Club.
The points for 2 3 4 5 6 7 8 9 T J Q K A
are 2 3 4 5 6 7 8 9 10 10 10 10 11/1.

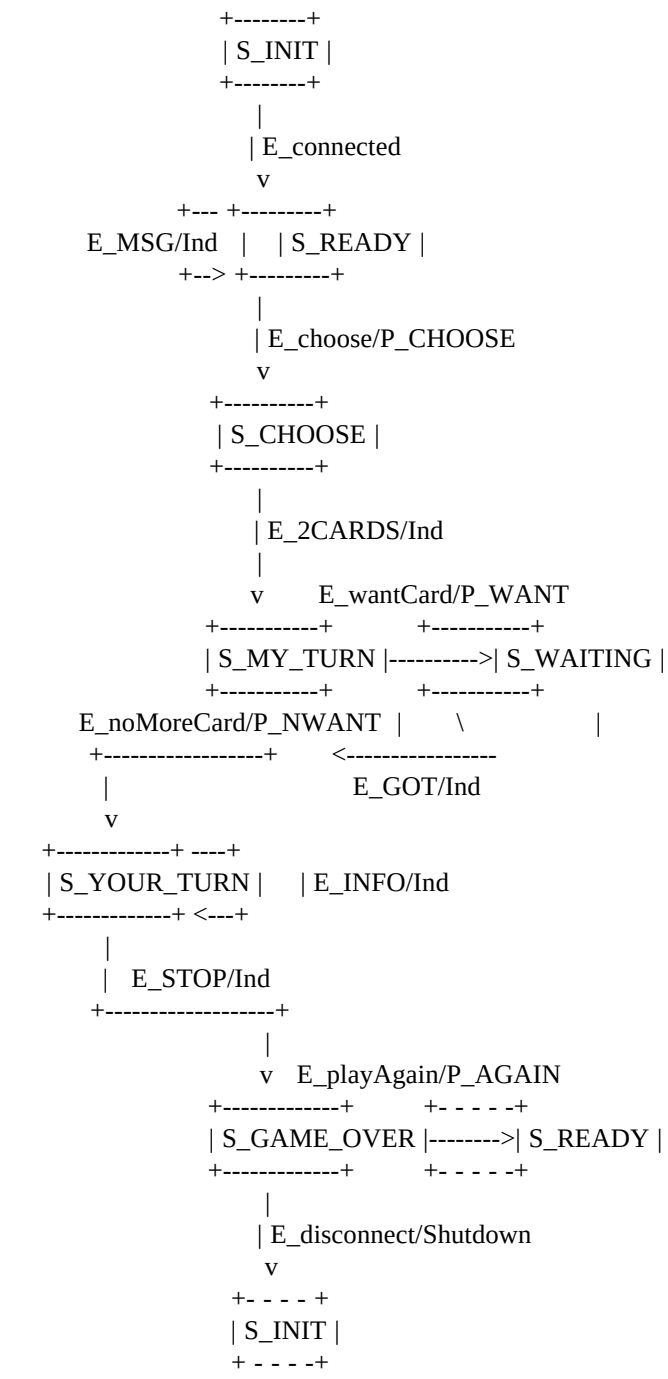
==========

State Transition Diagram

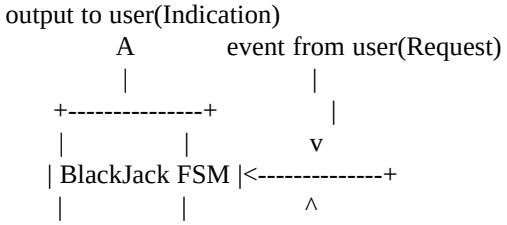
To simulate BlackJack our state definition for a BlackJack client is:

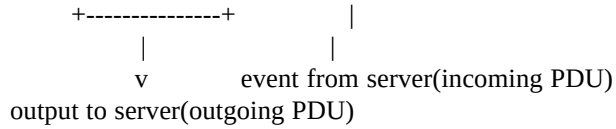
-- S_INIT : Initial state before a TCP connection is established.
-- S_READY : Ready state after a TCP connection is established.
FSM can receive server's welcome message(E_MSG event)
and user's choosing request(E_choose event).
-- S_CHOOSE : Choosing state after getting user's choice and before
receiving the first 2 cards(E_2CARDS event).
-- S_MY_TURN : My_turn state after client gets the first 2 cards and
before make any decision.
-- S_WAITING : Waiting for card state. After getting E_wantCard event
and before get E_GOT event.
-- S_YOUR_TURN : After get user's E_noMoreCard event and before dealer
stops(E_STOP event). In this state FSM can only receive
E_INFO event.
-- S_GAME_OVER : A state after FSM get an E_STOP event.

In the diagram below, an event name with all capital letters is caused by a corresding PDU (Protocol Data Unit) from server, otherwise is caused be user input. The state on the right side of '/' is either a PDU sent out due to a user command or an indication(Ind) to the user due to a received event from server.



If you want to see the layered model of this machine, it looks like:





In your client programming you are supposed to use the following definitions to write a subprogram called

int Automata(int state, int event, char *par)

which will perform the functions described before. Below you will find the definitions of PDU, states, events and example code.

```
/* definitions of PDU type -- each PDU type takes up one byte followed
                           by zero or more bytes depending on that
                            type */

#define P_MSG    1  // Message from server, like "Welcome to BlackJack\0"
                    // e.g. "\1Welcome to BlackJack\0"
                    // This is the only one having '\0' as terminator.
#define P_CHOOSE 2  // Choose to play # of decks (one deck=52 cards)
                    // 1st byte=2, 2nd byte=1 or 2
#define P_2CARDS 3  // Dealer gave me my initial 2 cards
                    // 1st byte=3, the rest are, e.g., "H2SJ"
#define P_WANT   4  // Want one more card
                    // one byte PDU
#define P_GOT    5  // Got one card from server
                    // 1st byte=5, the rest are, e.g., "CT"
#define P_NWANT  6  // Don't want more card
                    // one byte PDU
#define P_INFO   7  // Dealer's card
                    // 1st byte=7, the rest are, e.g., "DA"
#define P_STOP   9  // Dealer stops to take cards
                   // 1st byte=9, the rest are dealer's cards, e.g., "DADK"
#define P_AGAIN 10  // User wants to play again
                    // one byte PDU

/* definitions of states */

#define S_INIT      0
#define S_READY     1
#define S_CHOOSE    2
#define S_MY_TURN   3
#define S_WAITING   4
#define S_YOUR_TURN 5
#define S_GAME_OVER 6
#define max_state   7

/* definitions of events */

#define E_connected   0
#define E_MSG         1
#define E_choose      2
#define E_2CARDS      3
```

```
#define E_wantCard    4
#define E_GOT         5
#define E_noMoreCard  6
#define E_INFO        7
#define E_STOP        8
#define E_disconnect  9
#define E_playAgain  10
#define max_event    11

/* ================================================================= */
/*
   In the following examples, code piece 1 is a main loop which takes all
   in put data using select and then call Automata. How to pass the third
   parameter to the Automata is your local business.
*/
```