College of Engineering and Management, KOLAGHAT

Data Communication Laboratory (6th Semester 2021, CSE)

Experiment: Basic socket related system calls

(Filename - socket1.txt)

Create a directory [socket1] in your home directory and carry out this experiment in that directory.

$ mkdir $HOME/socket1

$ cd $HOME/socket1

A socket is a generalized interprocess communication channel. Like a pipe, a socket is represented by a file descriptor. Unlike pipes, sockets support communication between unrelated processes, and even between processes running on different machines that communicate over a network. Sockets are the primary means of communicating with other machines. [telnet], [rlogin], [ftp], [talk] and the other familiar network programs use sockets.

When a socket is created, the following features of a socket are to be specified: [domain], [type] and [protocol]. You may read manual of [socket(2)].

Synopsis of [socket] system call is

```
int socket(int domain, int type, int protocol);
```

Namespace or communication domain or protocol family of a socket:

Each namespace has a symbolic name that starts with `PF_'. A corresponding symbolic name starting with `AF_' designates the address format for that namespace.

Examples:

PF_UNIX or PF_LOCAL or PF_FILE are the symbolic names of the local namespace.

PF_INET is the symbolic name of IP version 4 (IPv4) Internet namespace.

PF_INET6 is the symbolic name of IP version 6 (IPv6) Internet namespace.

Type:

The type of a socket defines the user-level semantics of sending and receiving data on the socket.

Examples: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW etc etc.

Protocol:

This determines the low level mechanism of sending and receiving data on the socket. Normally only a single protocol exists to support a particular socket type within a given namespace. This is why namespace is sometimes called a protocol family, which is why the namespace names start with `PF_'.

Example: IPPROTO_TCP, IPPROTO_UDP etc etc.

Namespace (or communication domain or protocol family of a socket):

Local namespace:

The symbolic names of local namespace are PF_LOCAL or PF_UNIX or PF_FILE. When IPC is done in the same host, local namespace can be used. The local namespace is also known as "Unix domain sockets". Another name is "file namespace" since socket addresses are normally implemented as file names, in the local namespace. Any valid file name can be used as the address of the socket, but the calling process must have write permission on the directory containing it. It is common to put these files in the '/tmp' directory. After using the socket, the filename should be removed. It is also possible to assign local namespace socket address in abstract namespace.

The Internet namespace:

The symbolic name PF_INET is used to create a socket in IP version 4 (IPv4) Internet namespace. The symbolic name PF_INET6 is used to create a socket in IP version 6 (IPv6) Internet namespace.

Type of a socket:

Some of the socket types are: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET, SOCK_RAW, SOCK_RDM and SOCK_PACKET.

Protocol of a socket:

Normally only a single protocol exists to support a particular socket type within a given protocol family. You may examine the file [/etc/protocols] with [less] command.

```
$ less /etc/protocols
```

The synopsis of [socket] system call is repeated below:

```
int socket( int domain, int type, int protocol );
```

Argument one is the communication domain (a.k.a namespace or protocol family).
Second argument is the type of socket.
Third argument is the protocol.

Though different types of sockets can be created using suitable arguments, Unix domain sockets and Internet sockets (IPv4) would be illustrated in this experiment. Only datagram and stream sockets will be used in the examples.

Save the following program as [f1.c]. In this program four sockets are created:

stream socket in local namespace,

datagram socket in local namespace,

stream socket in IPv4 namespace (TCP socket) and

datagram socket in IPv4 namespace (UDP socket)

```
// file-name  f1.c
// four different sockets are created by defining A or B or C or D

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

//#define A
//#define B
//#define C
//#define D

int main (void)
{
 int                sd;   // to be used as socket-descriptor
 int                n;
 struct   sockaddr_in   d;

 printf("pid = %u \n", getpid() );

 #ifdef A
   printf("Assignment-1a.. stream socket in local namespace\n");
   sd = socket( PF_UNIX, SOCK_STREAM, 0 );
   // PF_UNIX may be replaced with PF_LOCAL or PF_FILE
   if( sd == -1 ) { perror("socket-call"); exit(1); }
   printf("descriptor = %u \n", sd );
 #endif

 #ifdef B
   printf("Assignment-1b... datagram socket in local namespace \n");
```

```
  sd = socket( PF_UNIX, SOCK_DGRAM, 0 );
  if( sd == -1 ) { perror("socket-call"); exit(1); }
  printf("descriptor = %u \n", sd );
#endif

#ifdef C
  printf("Assignment-1c... Internet IPv4 stream socket(TCP)\n");
  sd = socket( PF_INET, SOCK_STREAM, IPPROTO_TCP );
  if( sd == -1 ) { perror("socket-call"); exit(1); }
  printf("descriptor = %u \n", sd );

  /*   A newly created TCP socket has no remote or local address and is not fully specified. To identify the socket created in this experiment, we listen on this socket, though the socket is not yet bound to any address. This is done only to identify the type of the socket.   */
  n = listen( sd, 1 );
  if ( n == -1 ) { perror("listen"); exit(1); }

  /*   When listen( ) is called on a unbound socket, the socket is automatically bound to a random free port with the local address set to all local interfaces   */

#endif

#ifdef D
  printf("Assignment-1d... Internet IPv4 datagram socket(UDP)\n");
  sd = socket( PF_INET, SOCK_DGRAM, IPPROTO_UDP );
  if( sd == -1 ) { perror("socket-call"); exit(4); }
  printf("descriptor = %u \n", sd );

  /*   When a UDP socket is created, its local and remote addresses are unspecified. [ listen ] call was used to identify the socket created in Assignment-1c. This call is allowed on a stream socket. As UDP is not a stream socket, method used in Assignment-1c is not used in this assignment.

  To identify the created socket, the socket is bound to an Internet address 127.0.0.1 ( loopback interface ). [ bind ] system call is used to do that. An Internet address is a structure with three members. At present ignore the calls [inet_aton()], [htons()] and [bind()] calls.   */

  // address family
  d.sin_family = AF_INET;

  // port
```

```
  d.sin_port = htons(0); // any port

  // Internet address
  if( inet_aton( "127.0.0.1" , &(d.sin_addr)  ) == 0 )
  {
    printf("invalid-addr\n");
    exit(1);
  }
  // the created socket is bound to address [d]
  n = bind( sd, (struct sockaddr *) &d, sizeof(d) );
  if( n == -1 ) { perror("bind"); exit(1); }
 #endif
 while ( 1 ) { sleep(1); } // endless loop
 return 0;
}
```

The above program enters an endless loop after creating socket. This enables us to check creation of the sockets.

Assignment-1:

(a) Define A only, compile the program and run it. Ignore warning regarding unused variables.

T-3 $ gcc -Wall ./f1.c -o ./one-a

T-3 $ ./one-a

Examine the output of this command in another terminal

T-4 $ ls -l /proc/pid-value/fd

Record the following...

From output of program	From [proc] file system	
	[$ ls -l /proc/value-of-pid/fd]	
PID	Descriptor	Descriptor	type	Inode-number

Run this command.

T-4 $ netstat -xpn

OR

T-4 $ netstat -xpn | grep one-a

Record the following....

Protocol	Type	Connected-?	I-node	PID	Program-name
	STREAM/DGRAM	yes/no			

Was a stream Unix socket created ? (Y / n)

Terminate program [one-a] with CTRL-C.

(b) Define B only (comment A, C and D), compile and execute the program. Disregard warning about unused variables.

T-3 $ gcc -Wall ./f1.c -o ./one-b

T-3 $ ./one-b

Record the following ...

From output of program	From [proc] file system [$ ls -l /proc/value-of-pid/fd]		
PID	Descriptor	Descriptor	type	Inode-number

Record the following from the output of the command

T-4 $ netstat -axpn | grep one-b

Protocol	Type	Connected-?	I-node	PID	Program-name
	STREAM/DGRAM	yes/no			

Was a Unix datagram socket created ? (Y / n)

Terminate program [one-b] with CTRL-C.

(c) Define C only, (comment A, B and D) compile and execute the program. Disregard warning about unused variables.

T-3 $ gcc -Wall ./f1.c -o ./one-c

T-3 $ ./one-c

Record the following...

From output of program | From [proc] file system

| [$ ls -l /proc/value-of-pid/fd]

---|---

PID | Descriptor | Descriptor | type

---|---

| | |

Run the command and save the output in file [cc]

T-4 $ netstat -atpn > ./cc

Examine the file [cc] to fill up the three blanks.

Proto | Local-Address | Foreign-Address | State | PID | Program name

| | | | |

____ | *:_____ | *:* | LISTEN | ____ | /one-c

In the above line two fields [Recv-Q] and [Send-Q] were omitted.

Was a TCP socket created ? (Y / n)

Terminate program [one-c] with CTRL-C.

(d) Define D only, (comment A, B and C) compile and execute the program.

T-3 $ gcc -Wall ./f1.c -o ./one-d

T-3 $ ./one-d

Record the following...

From output of program | From [proc] file system

| [$ ls -l /proc/value-of-pid/fd]

---|---

PID | Descriptor | Descriptor | type

---|---

| | |

In another terminal use

T-4 $ netstat -aupn

Fill up the four blanks.

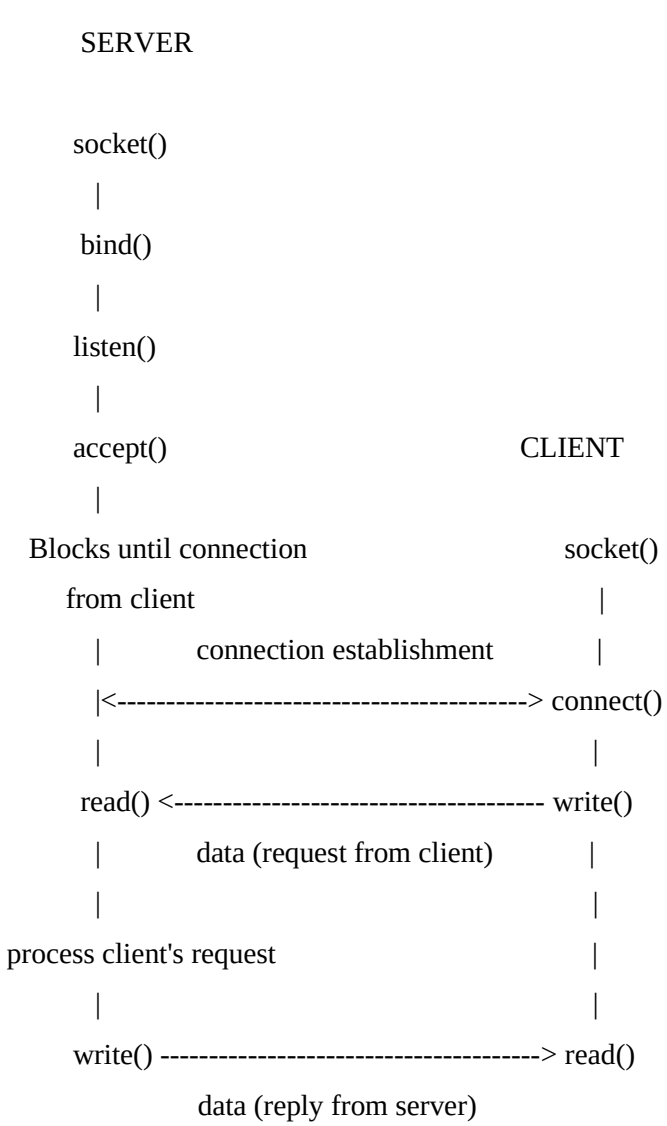
Proto	Local-Address	Foreign-Address	State	PID	Program name
____	*:_____	*:*	LISTEN	____	/one-d

Was a UDP socket created ? (Y / n)

Terminate program [one-d] with CTRL-C.

End of Assignment-1.

A few more operations on the socket are needed for receiving and sending data.



The following program is a simple server using connection oriented protocol (stream socket), in local namespace (Unix domain socket).

Save the following file as [f2.c]

```
// file-name f2.c
// ECHO server using stream UNIX domain socket

#include  <sys/types.h>
#include  <sys/stat.h>
#include  <stdio.h>
#include  <sys/socket.h>
#include  <sys/un.h>
#include  <unistd.h>
#include  <stdlib.h>

#define   PATH       "/tmp/aa"

#define   BUFFERSIZE  2048

int main ( void )
{
  int                 xxx; // to be used as socket descriptor
  int                 new_sock_fd;  // socket descriptor
  int                 n;
  ssize_t               i,j;
  char                 buf[BUFFERSIZE];
  socklen_t             addr_len;
  struct     sockaddr_un   server_address;


  // remove stale socket, if any from a previous run
  n = unlink( PATH );
  if( n == -1 )  perror("unlink");

  printf("pid = %u\n", getpid() );

  // UNIX domain socket is created.
  // PF_UNIX or PF_LOCAL or PF_FILE may be used as protocol family
 // xxx = socket( PF_UNIX, SOCK_STREAM, 0 );
```

```
  //xxx = socket( PF_LOCAL, SOCK_STREAM, 0 );
  //xxx = socket( PF_FILE, SOCK_STREAM, 0 );

  xxx = socket( PF_LOCAL, SOCK_STREAM, 0 );
  if( xxx == -1 ) { perror("socket-call" ); exit(1); }

  printf("socket descriptor = %u\n", xxx );

  // uncomment next three lines for Assignment-4
  //umask(0000);
  //n = fchmod( xxx,0707);
  //if( n == -1 ) { perror("chmod"); }



  /*
   The data type for Unix domain socket is [ struct sockaddr_un ]
   This has the following two members:
   struct sockaddr_un {
       short int       sun_family // AF_UNIX
       char            sun_path   // pathname of address
   }
  */

  // the above two members are initialised
  server_address.sun_family = AF_UNIX;
  strncpy( server_address.sun_path, PATH, sizeof(server_address.sun_path) );

  /* size of the address is determined with macro [ SUN_LEN ]. Address length is needed as one of
the arguments of [ bind ] system call. */
  addr_len = SUN_LEN ( &server_address );

  /*  An address is to be assigned to the [socket] as the local endpoint of communication. This is done
using the [bind] system call.  Read [bind(2)].  Note that the Unix domain socket address is cast to a
  generic data type [ struct sockaddr ].  */

    n = bind( xxx, ( struct sockaddr *) &server_address, addr_len );
  if( n == -1 ) { perror("bind-call"); exit(1); }

  /*  Then  [ listen ] system call is used, to be in readiness to accept incoming connections to the
socket   */
```

```
 n = listen( xxx, 5 );
 if( n == -1 ) { perror("listen-call"); exit(1); }
```

/* The second parameter[5] indicates that five incoming requests can be kept in queue. Read manual [listen(2)] */

/* Then [accept] system call is used to accept incoming request. This system call blocks, till a client's request comes on the socket. [accept] system call extracts the first connection request on the queue of pending connections, creates a new connected socket and allocates a new descriptor for the socket. In the following lines this descriptor is named [new_sock_fd], which the [accept] system call returns. This new descriptor, is full duplex (it can be read from and can be written into) and communication is done with the client with it. Read manual [accept(2)]. */

```
 while( 1 ) // In this endless loop a single client request is served
 {
   printf( "Waiting for a client's connection\n" );

   new_sock_fd=accept(  xxx,
                        (struct sockaddr *) &server_address,
                        &addr_len
                      );
   if( new_sock_fd == -1 ) { perror("accept"); exit(1); }

   memset( buf, '\0', BUFFERSIZE );
   i = read( new_sock_fd, buf, BUFFERSIZE);
   if( i == -1 ) { perror("read-call"); close(new_sock_fd ); }
   printf("uds-server: read %u byte request from client\n", i );
   j = write( new_sock_fd, buf, i );
   if( j == -1 ) { perror("write"); exit(1); }
   printf("uds-server: written %u byte reply to client\n", j );
   close( new_sock_fd );

 }  // while ends

 return 0;
} // end of main
```

Assignment-2:

Compile and execute the program...
T-3 $ gcc -Wall ./f2.c -o ./uds-server1
T-3 $ ./uds-server1

PID =

Use the following command to find out I-Node number of the newly created socket
T-4 $ ls -l /proc/pid-of-process/fd

I-Node number =

Check creation of file(socket)
T-4 $ ls -l /tmp | grep aa

Check type of the file
T-4 $ file /tmp/aa

Use the following command.
T-4 $ netstat -xapn

Examine the output and fill up the following table

Protocol Type State I-Node PID Path

Keep this server running in this terminal. A client program would connect to this server.

End of Assignment-2.

The uds-client program must know the address of uds-socket of server for communication. The connection oriented client program has to call [socket] and [connect] calls to establish connection to the server. Then the client sends request to the server and then reads the reply from the server.

Save the following program as [f3.c].

```
// file-name   f3.c
// ECHO client program using stream UNIX domain socket
```

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

#define  BUFFERSIZE  1024

#define  SERVER      "/tmp/aa"

int main( void )
{
  int                 sock_fd;
  int                 n;
  socklen_t             length;
  ssize_t               i,j;
  char                  buffer[BUFFERSIZE];
  struct   sockaddr_un   unix_addr;


  /* uncomment any one of the following three lines */
  //sock_fd = socket( PF_UNIX, SOCK_STREAM, 0 );
  //sock_fd = socket( PF_LOCAL, SOCK_STREAM, 0 );
  sock_fd = socket( PF_FILE, SOCK_STREAM, 0 );
  if( sock_fd == -1 ) { perror("socket"); exit(1); }

  unix_addr.sun_family = AF_UNIX;
  strncpy( unix_addr.sun_path, SERVER, sizeof(unix_addr.sun_path) );
  length = SUN_LEN( &unix_addr );

  n = connect( sock_fd, (struct sockaddr*) &unix_addr, length );
  if( n == -1 ) { perror("connect"); exit(1); }

  // clear the buffer with null characters
  memset( buffer, '\0', BUFFERSIZE );
  write( STDOUT_FILENO, "Enter a string ", 15 );
  i = read( STDIN_FILENO, buffer, BUFFERSIZE );

  // writing to server socket
  j = write( sock_fd, buffer, i );
```

```
  if( j == -1 ) { perror("write"); exit(1); }

  // clear the buffer with null characters
  memset( buffer, '\0', BUFFERSIZE );
  // reading from server socket
  i = read( sock_fd, buffer, BUFFERSIZE );
  if( i == -1 ) { perror("read"); exit(1); }

  write( STDOUT_FILENO, "Received from server->", 22 );
  j = write( STDOUT_FILENO, buffer, i );
  if( j == -1 ) { perror("write-2"); exit(1); }

  return 0;
}
```

Assignment-3:

(a) Compile and execute the program...
T-4 $ gcc -Wall ./f3.c -o ./uds-client
T-4 $ ./uds-client

Did the echo server-client work ? (Y / n)

(b) Execute the client program but do not enter a string now
T-4 $ ./uds-client

In this condition client and server are connected. To verify this, run
the following command in yet another terminal.
T-5 $ netstat -axpn | grep uds

(c) Stop the server with SIGINT
T-3 CTRL-C

Run the client when server is not running
T-4 $ ./uds-client

Did [connect] call fail with "Connection refused" message ? (Y / n)

Assignment-4:

Copy the client program under [/tmp/] directory so that other users can use the client program.

T-4 you $ cp ./uds-client /tmp/

Start the server.

T-3 you $ ./uds-server1

Login in another terminal as [sumit] or [guest] and try to run the client program.

T-6 sumit $ /tmp/uds-client

Connection would be refused as user [sumit] does not have write permission on the socket. Check with...

T-6 sumit $ ls -l /tmp/

Stop the server

T-3 CTRL-C

Edit [f2.c] to uncomment the three lines that change permissions.

Compile the changed program and start the server

T-3 you $ gcc -Wall ./f2.c -o ./uds-server2

T-3 you $ ./uds-server2

Run the client program again as [sumit].

T-6 sumit $ /tmp/uds-client

Did the server-client work ? (Y / n)

End of Assignment-4.