# Chapter 9

# IPC - III

## Chapter Objectives

To understand concepts of IPC used over the duration of the course.

### Objectives

For this chapter, the following are the objectives:
- Client-Server Interaction
- TCP Sockets,

Slide #9-1

**Notes**

In this chapter, we examine TCP sockets.

The objective of this chapter is to provide an understanding of concepts on TCP sockets.

## Chapter Organization

1. **Objective**: Introduction to
   - TCP sockets,

2. **Description**: This an introduction to IPC mechanisms in Linux.
   This chapter provides an introduction to the concepts used in course.

3. **Concepts Covered in Chapter**:
   - TCP sockets

4. **Prior Knowledge**:
   same as Chapter #1

5. **Teaching & Learning Strategy**:
   Discussion questions are,
   - What are these tools for?

6. **Teaching Format**:
   Theory + Homework Assignments

7. **Study Time**: 120 Minutes (Lecture & Theory)
   + ~45 minutes (Homework Assignments)

8. **Assessment**: Group Homework Assignments

9. **Homework Eval**: Group

10. **Chapter References**:

Programming the Client Server

# The client – server paradigm

also known as Request-Response.
two processes
the client initiates the connection and
    makes the requests.
the server waits for requests and
    services the request.

Slide #9-2

**Notes**

**IPC - review**

1. Allows processes to communicate.
2. Using a file IO based approach - write and reads from a file.

   Alternatively,

   2.1. use file descriptors for syscalls such as read()
   - pipes, sockets.

   2.2. use special functions
   - shm, sem , msg, signals

Sockets

# The Socket

- is similar to pipe in that there are 2 end points to the communications path
- is unlike pipe in that it is bi-directional.
- is similar to pipes in that the communication uses descriptors.
- is unlike pipe in that the fd socket is also known as socket descriptors and, may be in unrelated process.

Slide #9-3

**Notes**

1. Sockets similar to pipes in that there are two end points to the communications.

2. `Pipes()` provides a one-way communication path, while `socket()` provides a bi-directional communication path.

3. Again, sockets are similar to pipes in that the communication is on file descriptors.

4. Also, sockets are unlike pipes in that the file descriptors are also known as socket descriptors and, they may be in unrelated processes.

# Socket ( )

## TCP socket

```
int socket (AF_INET, SOCK_STREAM, PF_UNSPEC)
```

## UDP socket

```
int socket (AF_INET, SOCK_DGRAM, PF_UNSPEC)
```

Slide #9-4

**Notes:**

1. The system call `socket()` returns a file descriptor
2. This file descriptor refers to an entry in the kernel socket table, and hence is also known as **socket descriptor**.

**Code Example** // `pipe()`

```
// Chap 9_1.c
   void main()
   {
      int n =0;
      int fd[2];
      char buf[1024];

      pipe(fd);

      write(fd[1], "Hello World\n", 12);
      n = read(fd[0], buf, sizeof(buf));

      printf("%d == read(%d, %#010x, %d) => '%' \n",
           buf, sizeof(buf), buf);
   }
```

gethostbyname()
getservbyname()
copy to network format

# Programming the client

socket () ... create file descriptor

connect() ... connect to server
- gethostbyname()
- getservbyname()
- copy to network format

shutdown()
close()

Slide #9-5

Notes:

1. socket()
   int socket(address_family, sock_stream, protocol_family)

   1.1. where the address_family is typically AF_INET

   1.2. sock_stream can be SOCK_STREAM, SOCK_DGRAM

   1.3. protocol_ family can be PF_INET or PF_UNSPEC

2. connect
   int connect(int ds,
   struct sockaddr *name,
   int namelen);

   where sd = socket descriptor and
   struct sockaddr {
   u_short sa_family;
   char sa_data[14]; }

   struct sockaddr for IP has the following structure:
   struct sockaddr_in {
   u_short sa_family; /* address_family AF_INET */
   u_short sa_port; /* unique port number */
   struct in_addr sa_addr; /* Internet address of host */

```
                char sa_zero[8]; }   /* padding */
```

3. `shutdown()`
   shutdown cuts off communication in a certain direction

4. `close()`
   closes the kernel socket descriptor and de-allocates kernel resources

```
// #include "tcp_client.h"

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <netdb.h>
#include <errno.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>

#ifdef _POSIX_SOURCE
#define SA_RESTART 0x000004
#endif

#define MYSERVER_PORT_ADDRESS 29876
#define MYSERVER_DATA_FOUND "LineNumber Found"

void ProcessCommand(char *combuf, int sd);
int ParseCommand(char *str, char **argv, int max_args);
int DoCommand(char **argv, int argc, int sd, char *errbuf);
int open_socket_connection(char *host, struct sockaddr_in *saddr_in);
int GetBinaryData(int linenum, int sd, char *errbuf);

struct Example {
      int linenum;
      char data[1024];
};

typedef struct Example Example;

char *ProgramName;

int open_socket_connection(char *host, struct sockaddr_in *saddr_in) {

      int sd, i;
```

```
      struct hostent *hostentry;

      // open a socket
      if ((sd=socket(AF_INET,SOCK_STREAM, PF_UNSPEC)) < 0) {
            fprintf(stderr,"Socket Open Failure: %s\n", strerror(errno));
            return -1;
      }

      // get IP address on Server NIC
      if ((hostentry=gethostbyname(host))==NULL) {
            fprintf(stderr,"host name lookup failure: %s\n", strerror(errno));
            return -1;
      }

      // get into Network Independent Format
      saddr_in->sin_family=AF_INET;
      saddr_in->sin_port=MYSERVER_PORT_ADDRESS;

      // assuming BIG Endian on server
      if (memcpy (&(saddr_in->sin_addr), hostentry->h_addr, sizeof(struct
in_addr))==NULL) {
            fprintf(stderr,"memcpy error: %s\n", strerror(errno));
            return -1;
      }

      // connect()
      if (connect(sd,(struct sockaddr *)saddr_in,sizeof(struct sockaddr_in))
< 0) {
            fprintf(stderr,"connect() failure: %s\n", strerror(errno));
            return -1;
      }

      return sd;
}

void ProcessCommand(char *combuf, int sd) {

      char *vbuf[256];
      int maxargs = sizeof(vbuf) / sizeof(char *);
      int numargs;

      numargs = ParseCommand(combuf, vbuf, maxargs);

      if (numargs==maxargs) {
            fprintf(stderr,"too many args\n");
      }
      else if (numargs) {
            int status;
            char errbuf[128];

            if ((status=DoCommand(vbuf,numargs,sd, errbuf))==-1) {
                  fprintf(stderr,"input args error: %s\n", errbuf);
            }
      }

      {
            int i=0;
            while (i++<numargs) free(vbuf[i]);  // free argument vector
      }
```

```
}

int DoCommand(char **argv, int argc, int sd, char *errbuf) {

      int status;

      if (strcasecmp(*argv,"print") == 0) {

            int linenum;

            for (++argv; *argv; argv++) {
                  errbuf[0]=0;

                  linenum=atoi(*argv);

                  if (linenum) {
                        fprintf(stderr, "Printing Line# %d:\n",linenum);
                        status=GetBinaryData(linenum, sd, errbuf);

                        if (*errbuf)
                              fprintf(stderr, errbuf);

                        if (status==-1)
                              return -1;
                  }
            }

            return 0;
      }
      else if ((strcasecmp(*argv,"quit") == 0) || (strcasecmp(*argv,"exit")
== 0)) {
            exit(0);
      }
      else if (strcasecmp(*argv,"help") == 0) {
            fprintf(stderr, "\tprint Line# Line# ... \tprints line\n");
            fprintf(stderr, "\thelp              ... \tthis message\n");
            fprintf(stderr, "\t[quit|exit|^D]    ... \tquits program\n");

            return 0;
      }

      sprintf(errbuf, "invalid command: %s", *argv);

      return -1;
}

int GetBinaryData(int linenum, int sd, char *errbuf) {

      Example     ExampleData[1];

      char str[256];

      int i, flags, status;

      str[0]=0;

      sprintf(str,"%d", linenum);
```

```
      i=strlen(str);

      if (write(sd,str,i)<i) {
            fprintf(stderr,"write() Error: %s\n", strerror(errno));
            return -1;
      }

      if ((read(sd,str,sizeof(str))) < 0) {
            fprintf(stderr,"read() Error: %s\n", strerror(errno));
            return -1;
      }
      else if (i==0) {
            fprintf(stderr,"Host Reply Error: %s\n", strerror(errno));
            return -1;
      }

      /*
            Protocol Note:
                  Server Sends MYSERVER_DATA_FOUND initially,
                  followed by data.
      */

      // no MYSERVER_DATA_FOUND message recd
      if (strncmp(str,MYSERVER_DATA_FOUND,sizeof(MYSERVER_DATA_FOUND)-1)) {
            sprintf(errbuf,"Line# %d:No Data Found.\n",linenum);

            /*
                  flush data in socket buffers.
            */

            //    set to non-blocking read, and discard until read fails

            if ((flags=fcntl(sd,F_GETFL,0))==-1) {
                  fprintf(stderr, "fcntl() getflags error: %s\n",
strerror(errno));
            }
            else {
                  flags |= O_NONBLOCK;

                  if (fcntl(sd,F_SETFL,flags)> -1) {

                        // consume until EOF
                        while (read(sd,str, sizeof(str))>0);

                        flags ^= O_NONBLOCK;
                        fcntl(sd,F_SETFL,flags);
                  }
            }

            return 0;
      }

      // we know data exists
      if (read(sd, &ExampleData, sizeof(Example))<=0) {
            sprintf(errbuf,"read() error.Connection Lost!, %s\n",
strerror(errno));
            return -1;
      }
```

```
	if (linenum==ExampleData->linenum) {
		fprintf(stdout,"\n\tLine Number\t: %s\n", ExampleData->linenum);
		fprintf(stdout,"  \t\tData     \t: %s\n", ExampleData->data);
	}

	return 0;
}

int ParseCommand(char *str, char **argv, int max_args) {

	int i=0, len=0;

	char *s=str;

	for (;*str;str++,len++) {
		if (isspace(*str)) {
			char *narg;

			narg=malloc(len+1);
			strncpy(narg,s,len);
			narg[len]=0;

			argv[i++]=narg;

			// consume multiple spaces
			for (;*str && isspace(*str);str++);

			s=str;
			len=0;

			if (i==max_args-1)
				break;
		}
	}

	if (len) {
			char *narg;

			narg=malloc(len+1);
			strncpy(narg,s,len);
			narg[len]=0;

			argv[i++]=narg;
	}

	argv[i]=0;

	return i;
}

int main(int argc, char **argv) {

	char  combuf[1024];

	char  *prompt = "myshell> ";
	int		sd;
	int   flags=O_RDONLY;

	char  *host="localhost";
```

```
      char  *serv="dataserver";
      char  *prot="tcp";

      char  errbuf[256];

      struct  sockaddr_in saddr_in[1];

      ProgramName= *argv;
      errbuf[0]=0;

      if ((sd=open_socket_connection(host,saddr_in)) < 0) {
            fprintf(stderr,"Error: %s\n", strerror(errno));
            return -1;
      }

      fprintf(stderr, "\n%s", prompt);

      while (fgets(combuf, sizeof(combuf), stdin) != NULL) {
            ProcessCommand(combuf, sd);

            fprintf(stderr, "\n%s", prompt);
      }

      shutdown(sd,2);
      close(sd);
}
```

# Programming the server

socket () ... create file descriptor

- bind() => get hostbyname(),
  get servbyname()
  copy to network format
- listen ()

accept()
fork()
shutdown()
close()

Slide #9-6

Notes:

1. socket()
2. bind()
3. listen()
4. accept()
5. shutdown()

```
// #include "tcp_server.h"

#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>

#include <signal.h>

#define MYSERVER_PORT_ADDRESS 29876
#define MYSERVER_CLIENTS 3
#define MYSERVER_SOCKET_READ_TIMEOUT 240
```

```
#define MYSERVER_DATA_FILE "myserver.dat"
#define MYSERVER_DATA_FOUND "LineNumber Found"

int init_myserver(char *host, struct sockaddr_in *);
int do_request(int csd, char *errbuf);
int process_request(int sd, struct sockaddr_in *, char *errbuf);

int do_select(int csd, char *errbuf);
int read_and_send(int csd, int rfd, char *errbuf);
int signal_handler(void);

void wait_child(void);

struct Example {
      int linenum;
      char data[1024];
};

typedef struct Example Example;

/* --

*/

void wait_child(void) {
      while (waitpid(-1,0,WNOHANG) > 0);
}

/* --
      int signal_handler(void);
*/

int signal_handler(void) {

      struct sigaction new[1];
      struct sigaction old[1];

      sigemptyset(&new->sa_mask);
      new->sa_handler=wait_child;
      new->sa_flags=SA_RESTART;

      sigaction(SIGCHLD,new,old);
}

int init_myserver(char *host, struct sockaddr_in *saddr_in) {

      struct hostent *hostentry;

      int sd;

      // open a socket
      if ((sd=socket(AF_INET,SOCK_STREAM, PF_UNSPEC)) < 0) {
            fprintf(stderr,"Socket Open Failure: %s\n", strerror(errno));
            return -1;
      }

      // get IP address on Server NIC
      if ((hostentry=gethostbyname(host))==NULL) {
            fprintf(stderr,"host name lookup failure: %s\n", strerror(errno));
```

```
            return -1;
      }

      // get into Network Independent Format
      saddr_in->sin_family=AF_INET;
      saddr_in->sin_port=MYSERVER_PORT_ADDRESS;

      // assuming BIG Endian on server
      if (memcpy (&(saddr_in->sin_addr), hostentry->h_addr, sizeof(struct
in_addr))==NULL) {
            fprintf(stderr,"memcpy error: %s\n", strerror(errno));
            return -1;
      }

      // Socket binding
      if ((bind(sd,(struct sockaddr *) saddr_in, sizeof(struct sockaddr_in)))
< 0) {
            fprintf(stderr,"bind() error: %s\n", strerror(errno));
            return -1;
      }

      // Setup for listening on port
      if (listen(sd,MYSERVER_CLIENTS) < 0) {
            fprintf(stderr,"listen() error: %s\n", strerror(errno));
            return -1;
      }

      return sd;
}

int process_request(int sd, struct sockaddr_in *saddr_in, char *errbuf) {

      int csd;

      char str[256];

      unsigned long in_addr = sizeof(struct sockaddr_in);

      // wait for connection request from clients
      if ((csd=accept(sd, (struct sockaddr *) saddr_in, &in_addr)) < 0) {
            fprintf(stderr,"accept() error: %s\n", strerror(errno));
            return -1;
      }

      shutdown(sd,2);
      do_request(csd, errbuf);
      return 0;
}

int do_request(int csd, char *errbuf) {

      int cpid=0, rfd=0;

      if ((cpid=fork()) < 0) {
            fprintf(stderr,"fork() error: %s\n", strerror(errno));
            return -1;
      }

      if (cpid==0) {
```

```
            // child

            if ((rfd=open(MYSERVER_DATA_FILE, O_RDONLY)) < 0) {
                  sprintf(errbuf,"File open error:%s:%s\n",
                        MYSERVER_DATA_FILE,strerror(errno));
                  return -1;
            }

            while (do_select(csd, errbuf)) {
                  errbuf[0]=0;

                  if (read_and_send(csd,rfd,errbuf)< 0) {
                        break;
                  }
            }

            // no more data from client
            shutdown(csd,2);
            close(rfd);
            close(csd);

            _exit(0);
      }
      else {

            // parent

            close(csd);

            return 0;

      }

}

int do_select(int csd, char *errbuf) {

      fd_set      fdset;
      struct timeval tmout[1];

      int nrd=0, sd=csd;

      tmout->tv_sec=MYSERVER_SOCKET_READ_TIMEOUT;
      tmout->tv_usec=0;

      FD_ZERO(&fdset);

      FD_SET(sd,&fdset);

      if ((nrd=select(sd+1, &fdset,0,0,tmout)) < 0) {
            sprintf(errbuf,"select() failed: %s\n", strerror(errno));
            return 0;
      }
      else if (nrd==0) {
            sprintf(errbuf,"select() timed out\n");
            return 0;
      }
```

```
      if (FD_ISSET(sd, &fdset)) {
            return 1;
      }

      return 0;
}

int read_and_send(int csd,int rfd,char *errbuf) {

      Example ExampleData[1];

      char str[256];

      int i, j, readpos, ExampleLineNum;

      str[0]=0;

      if (read(csd, &str, sizeof(str)) <= 0) {
            sprintf(errbuf, "Socket Read Error: %s\n", strerror(errno));
            return -1;
      }

      ExampleLineNum=atoi(str);
      readpos=ExampleLineNum * sizeof(Example);

      if (lseek(rfd, readpos, SEEK_SET) < 0) {
            sprintf(errbuf,"lseek() Error: %s\n", strerror(errno));
            return -1;
      }

      if ((i=read(rfd, ExampleData, sizeof(struct Example))) <= 0) {
            fprintf(stderr,"ERR: LineNum: %d: NOT FOUND\n", ExampleLineNum);
            sprintf(errbuf,"ERR: LineNum: %d: NOT FOUND\n", ExampleLineNum);
            strcpy (str,errbuf);
      }
      else {
            strcpy(str, MYSERVER_DATA_FOUND);
      }

      j=strlen(str);
      if (write(csd,str,j)<j) {
            fprintf(stderr, "write() str Error: %s\n", strerror(errno));
            return -1;
      }

      j=sizeof(ExampleData);
      if (i) {
      // fprintf(stderr,"\n\tLine Number\t: %s\n",   ExampleData->linenum);
      // fprintf(stderr,"  \t\tData     \t: %s\n",   ExampleData->data);
            if (write(csd,ExampleData,j)<j) {
                  fprintf(stderr, "write() ExampleData Error: %s\n",
strerror(errno));
                  return -1;
            }
      }

      return 0;

}
```

```
int main(int argc, char **argv) {

      int i=0, cnt=0;

      int sd;

      // char *host="127.0.0.1";

      char *host="localhost";

      char *srv="myserver";
      char *proto="tcp";

      char str[256];
      char errbuf[256];

      struct sockaddr_in saddr_in[1];

      errbuf[0]=0;

      signal_handler();             // setup signal handler for exiting
children

      if ((sd=init_myserver(host,saddr_in))<1) {
            fprintf(stderr,"init_myserver() error: %s\n", strerror(errno));
            return -1;
      }

      while (cnt++ < MYSERVER_CLIENTS) {
            if (process_request(sd, saddr_in, errbuf)) {
                  fprintf(stderr, "Request Processing Error: %s\n", errbuf);
                  return -1;
            }
      }

      fprintf(stderr, "Reached Max. Sessions for Server\n", MYSERVER_CLIENTS);

      while (wait(0)>0) {
                  fprintf(stderr, "Waiting for Sessions to Terminate.\n");
      }

      return 0;
}
```

# Putting it all together

| Client | Server |
|---|---|
| – socket() | – socket() |
| connect() | bind() |
| shutdown() | listen() |
| close() | accept() & fork() |
| | shutdown() |
| | close() |

Slide #9-7

**Notes:**

1. Steps to follow for TCP sockets.

**Makefile**

```
all:  tcp_server tcp_client gendata myserver.dat

tcp_server: tcp_server.c
      gcc -o tcp_server -lsocket -lnsl tcp_server.c

tcp_client: tcp_client.c
      gcc -o tcp_client -lsocket -lnsl tcp_client.c

myserver.dat gendata: gendata.c
      -rm -f myserver.dat myserver.txt
      gcc -o gendata gendata.c
      ./gendata
      chmod 644 myserver.dat
      ls -l myserver.dat

clean:
      rm -f myserver.dat tcp_server tcp_client tcp_sample.tar
tcp_sample.tar.gz gendata

move: clean
      tar -cvf tcp_sample.tar tcp_server.c tcp_client.c gendata.c Makefile
      gzip tcp_sample.tar
```

**gendata.c**

```
#define MAXLINES 3

#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

struct Example {
      int linenum;
      char data[1024];
};

typedef struct Example Example;

int main() {

      Example ExampleData[1];

      int wfd, i=1, j=0;

      umask(777);

      if ((wfd=open("myserver.dat",O_CREAT|O_TRUNC|O_WRONLY,0x644)) < 0) {
            fprintf(stderr,"open() Failure: %s\n", strerror(errno));
            return -1;
      }

      while (i<=MAXLINES) {
            ExampleData->linenum=i;
            ExampleData->data[0]=0;
            sprintf(ExampleData->data,"This is Line '%d'", i);

            j=sizeof(struct Example);
            if ((write(wfd,ExampleData, j)) < j) {
                  fprintf(stderr,"write() Failure: %s\n", strerror(errno));
                  return -1;
            }
            i++;
      }

      fprintf(stderr,"Wrote %d lines.\n",i);

      return 0;
}
```

# Sending data via Sockets

ASCII or binary protocol?
Who the talker is, and, who the listener is?

Slide #9-8

**Notes:**

1. Protocol Considerations for TCP sockets.

# Chapter 9

# IPC - III

## Assignment Questions

**Questions:**

7.1 Implement the "-socket tcp" option to communicate between the requester and the server.

7.2 Modify "mysh" (from chapter 8) to include to create a TCP client that will process 'lmywc', 'lmycat' and 'lmyls' as client-side builtin commands, and 'mywc', 'mycat' and 'myls' as server-side built-in commands,

7.3 Implement the 'get' and 'put' commands to get/put a single file, using socket communication.

7.4 Other commands can be ignored,
(or, handled appropriately, for extra credit … ☺