COMSATS University Islamabad,

Lahore Campus

**Defence Road, Off Raiwind Road, Lahore**



**Lab Manual**

**CSC322: operating Systems**

|  |  |  |
| --- | --- | --- |
| **WEEK** | **Topics** |  |
| **Week 1** | Introduction to Linux environment |  |
| **Week 2** | Using shell commands & gdb debugger. |  |
| **Week 3** | Programs dealing with fork, exec and wait |  |
| **Week 4 &5** | Unix I/O, hard Links & Symbolic Links |  |
| **Week 6** | UNIX special files, Pipes. |  |
| **Week 7** | Redirection. |  |
| **Week 8** | Signals. |  |
| **Week 9** | Multithreading |  |
| **Week 10** | Synchronization of threads using mutex and condition variables. |  |
| **Week 11** | System-V inter process communication mechanisms. Semaphore array. |  |
| **Week 12** | Semaphore &Shared memory |  |
| **Week 13-14** | Connection Oriented Communication, Little and Big Endian |  |
| **Week 15-16** | Message Queues |  |

List of practices/Lab Tasks

**CSC 322- Operating System**

***Week1: Introduction to Linux Environment***

**Learning Objectives:**

The objectives of this first experiment are:

1. To make you familiar with the Linux history, environment and highlight its differences from Windows particularly regarding filesystem.
2. For this purpose you will install Ubunto using Virtual Box or make your system dual boot. It is up to you. The systems in the lab are already configured with Ubunto.
3. You will learn how to create files and directories, copy and delete files, navigate the Linux directory hierarchy, and search for specific information contained in a file.
4. Compress and uncompress files using tar

**CSC 322- Operating System**

***Week 2: Using shell commands& gdb debugger.***

**Learning Objectives:**

This lab provides an

1. Introduction to the Linux command line interface and some of the basic commands available in Linux.
2. How to compile multi file C program using make utility.
3. How to Debug C Program using gdb

**Linux Command Structure**

The commands used in Linux have the following general syntax: ***command option(s) argument(s),*** although the options and arguments may be omitted. For example, in the command ***ls –l data.txt*** , **ls** is the command, **l** is an option, and **data.txt** is the argument. Options are generally preceded by a –(dash). Multiple options can be specified for a single command**.**

**Directions**

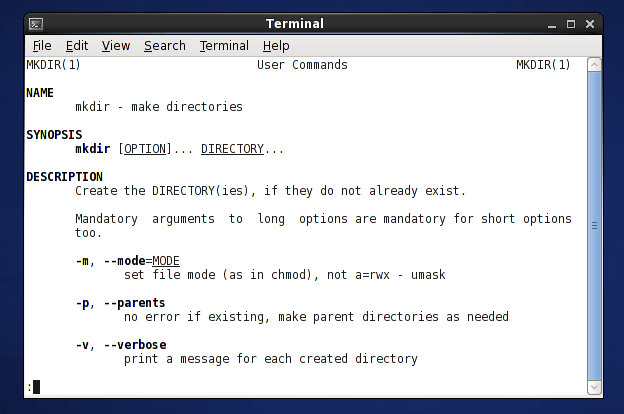
Press the Enter key after typing each command. The general command syntax is the command name, a space, and zero or more parameters. Do not forget to type the space between the command names and the parameters. For example, a **space** separates ***cd*** from **..** in the ***cd ..*** command.

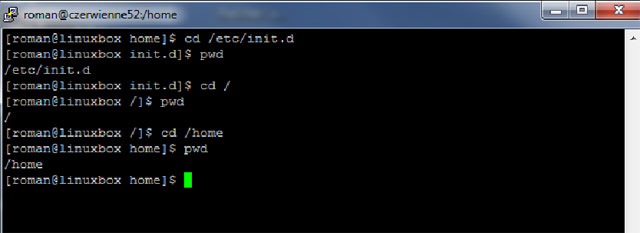
**Exercise 1:** Use different shell commands like, mkdir, ls, rm, mv, cd,cp, gcc, pwd...

1. Use man command for syntax and description of shell commands

For example: man mkdir

The man pages are a user manual that is by default built into most [Linux](http://www.linfo.org/linuxdef.html) [*distributions*](http://www.linfo.org/distributions_list.html) (i.e., versions) and most other [Unix-like](http://www.linfo.org/unix-like.html) [operating systems](http://www.linfo.org/operating_system.html) during installation. They provide extensive [documentation](http://www.linfo.org/documentation.html) about commands and other aspects of the system.

****



**Exercise 2:** Compile a “C” Programusing the GNU Compiler (GCC)

1. Write a simple ‘c’ Program using gedit editor.

#include <stdio.h>

int

main (void)

{

printf ("Hello, world!\n");

return 0;

}

1. Open Terminal. We will assume that the source code is stored in a file called ‘hello.c’. To compile the file ‘hello.c’ with gcc, use the following command:

**$ gcc -Wall hello.c -o hello**

This compiles the source code in ‘hello.c’ to machine code and stores it in an executable file ‘hello’. The output file for the machine code is specified using the -o option. This option is usually given as the last argument on the command line. If it is omitted, the output is written to a default file called ‘a.out’. Note that if a file with the same name as the executable file already exists in the current directory it will be overwritten. The option -Wall turns on all the most commonly-used compiler warnings.

1. To run the program, type the path name of the executable like this:

**$ ./hello**

**Hello, world!**

This loads the executable file into memory and causes the CPU to begin executing the instructions contained within it. The path ./ refers to the current directory, so ./hello loads and runs the executable file ‘hello’ located in the current directory.

**Exercise 3:** Write a C Program to Find the Largest Number Among Three Numbers?

**Exercise 4:** Write a C Program to Check Whether a number is Even or Odd?

**Exercise 5:** Use debugger to debug following programs

1. Compile the C program with debugging option –g. This allows the compiler to collect the debugging information. The above command creates a.out file which will be used for debugging

**gcc –g factorial.c**

1. Launch the C debugger (gdb)

**gdb a.out**

**l for listing**

1. Set up a break point inside C program

**b line\_number**

1. Execute the C program in gdb debugger

**run**

1. Printing the variable values inside gdb debugger

**p**

1. Continue, stepping over and in – gdb commands

* c or continue: Debugger will continue executing until the next break point.
* n or next: Debugger will execute the next line as single instruction.
* s or step: Same as next, but does not treats function as a single instruction, instead goes into the function and executes it line by line.

**Program 1:**

#include <stdio.h>

#include <stdlib.h>

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

{

char \*buf;

buf = malloc(1<<31);

fgets(buf, 1024, stdin);

printf("%s\n", buf);

return 1;

}

The program is meant to read in a line of text from the user and print it.

* Compile the program with debugging flags:

**gcc -g segfault.c**

* Now we run the program:

**a.out**

**output**

Hello World!

Segmentation fault

1. **gdb a.out**
2. **Run**

SIGSEGV signal from the operating system. Access an invalid memory address.

1. Backtrace

A back trace is a summary of how your program got where it is.

**4.** **frame 3**

**5. p buf**

The value of buf is 0x0, which is the NULL pointer.

**6. Kill**

**7. b 8**

**8. Run**

check the value of buf before the malloc call.

**P buf**

the value should be garbage

step over the malloc call and examine buf again

**n**

**P buf**

After the call to malloc, buf is NULL

malloc returns NULL when it cannot allocate the amount of memory requested.

The value of the expression 1 << 31 (the integer 1 right-shifted 31 times) is 429497295, or 4GB (gigabytes). Very few machines have this kind of memory.

So of cousre malloc would fail.Change the 1<<31 to 1024 (or 1<<9), and the program will work as expected.

**Exercise 6: Debug the following C program using gdb**

# include <stdio.h>

int main()

{

int i, num, j;

printf ("Enter the number: ");

scanf ("%d", &num );

for (i=1; i<num; i++)

j=j\*i;

printf("The factorial of %d is %d\n",num,j);

}

**CSC 322- Operating System**

***Week 3:*** **Programs dealing with fork, exec and wait**

**Learning Objectives:** The objective of this exercise is to get you to

1. Write, compile and run a number of programs in C which make use of fork (), exec and wait system calls.

**Exercise 1:** Write a C program that illustrates the creation of child process using fork() system call?

1. Start
2. Declare pid
3. create new process using fork( ) system call
4. If pid!=0 then
5. Display parent process getpid(),getppid().
6. Else
7. Display child process getpid().getppid().
8. End

**Source Code**

#include<stdio.h>

int main( )

{

printf(“original process with pid %d ppid %d\n”,

getpid() ,getppid());

pid=fork();

if(pid!=0)

printf(“parent process with pid %d ppid %d \n”,

getpid(),getppid());

else

{

sleep(5);

printf(“child process with pid %d ppid %d\n”,

getpid(),getppid());

}

printf(“ pid %d terminates “,getpid());

}

**Out Put**

original process with pid 3456 and ppid 3525

child process with pid 3457 and ppid 3456

pid 3457 terminates

parent process with pid 3456 and ppid 3525

pid 3456 terminates

**Exercise2:** Implementing wait system call using C program

**wait()**

* The wait system call suspends the calling process until one of its immediate children terminates.
* If the call is successful, the process ID of the terminating child is returned.
* Zombie process—a process that has terminated but whose exit status has not yet been received by its parent process or by init.

**pid\_t wait(int \****status***);**

Where status is an integer value where the UNIX system stores the value returned by child process

#include <stdio.h>

void main()

{

int pid, status;

pid = fork();

if(pid == -1) {

printf(“fork failed\n”);

exit(1);

}

if(pid == 0) { /\* Child \*/

printf(“Child here!\n”);

}

else { /\* Parent \*/

wait(&status);

printf(“Well done kid!\n”);

}

}

**Exercise 3:** Write a program that creates child processes and waits for the child to finish before termination.

**Source Code:**

#include <stdio.h>

#include <sys/wait.h> /\* contains prototype for wait \*/

int main(void)

{

int pid;

int status;

printf("Hello World!\n");

pid = fork( );

if (pid == -1) /\* check for error in fork \*/

{

perror("bad fork");

exit(1);

} if (pid == 0)

printf("I am the child process.\n");

else

{

wait(&status); /\* parent waits for child to finish \*/

printf("I am the parent process.\n");

}

}

**Exercise 4:** Write aprogram that creates a chain of n processes, where n is a command-line argument.

**Exercise 5:** Write a program that creates a fan of n processes where n is passed as a command-line argument.

**Exercise 6:** Write a program that creates a child process to run ls -l.

**Source Code**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main(void) {

pid\_t childpid;

childpid = fork();

if (childpid == -1) {

perror("Failed to fork");

return 1;

}

if (childpid == 0) { /\* child code \*/

execl("/bin/ls", "ls", "-l", NULL);

perror("Child failed to exec ls");

return 1;

}

if (childpid != wait(NULL)) { /\* parent code \*/

perror("Parent failed to wait due to signal or error");

return 1;

}

return 0;

}

**CSC 322- Operating System**

***Week 4& 5: UNIX I/O, hard Links & Symbolic Links***

**Learning Objectives:** The objective of this exercise is to get you to know

1. Difference between Hard links and symbolic links
2. How to create links using shell commands
3. Reading and writing data in ASCII files & in binary files.

**Exercise 1:** Create hard links and soft links using ln & ln –s Commands

Step1: Create a new file (myfile) using cat or touch command.

Step2: To create the 2nd, 3rd and etc. hard links, use the command:

* ln myfile ***link-name***

Step 3: Run the command “ls -il” to display the ***i-node number*** and ***link counter***

Step4: To create a symbolic link to the file “myfile”, use

* **ln -s myfile symlink**

Step5: Run the command “ls -il” to display the ***i-node number*** and ***link counter***

**Exercise 2:** Write a C program to count no. of blanks, characters, lines using standard i/o function.

1. Start.

2. open the file using fopen( ) function in “r” mode

3. ch=fgetc(fp) (to read character by character)

4. if ch = ‘ ‘ or ‘\n’

b=b+1.

5. if ch = ‘\n’

l=l+1.

6. c=c+1.

7. Repeat 3,4,5&6 steps until ch = eof

8. End

**Source Code**

#include<stdio.h>

int main( )

{

file \*fp:

int b=0,nl=0,c=0;

char ch;

fp=fopen(“text.txt”,”r”);

while(ch!=eof)

{

ch=fgetc(fp);

if(ch==’ ‘)

b++;

if(ch==’\n’)

nl++;

c++;

}

printf(“no.of blanks %d”,b);

printf(“no.of lines %d”,nl);

printf(“no.of characters %d”,c);

}

**Input:**

./a.out sss.txt

**Output:**

no.of blanks 5

n.of lines 2

no.of characters 36

**Exercise 3:** Write a C program to illustrate the mv command using system Calls

1. Open one existed file and one new open file using open( ) system call

2. Read the contents from keyboard using read( )

3. Write these contents into file using write()

4. Repeat 2,3 steps until eof

5. Close 2 file using fclose( ) system call

6. Delete existed file using using unlink( ) system.

**Exercise 4:** Write a program to illustrate “ls” command using system calls

1. Start.

2. Open directory using opendir( ) system call.

3. Read the directory using readdir( ) system call.

4. Print dp.name and dp.inode .

5. Repeat above step until end of directory.

6. End

**CSC 322- Operating System**

***Week 6 & 7:*** ***UNIX special files, Pipes and Redirection***

**Learning Objectives:** The objective of this exercise is to get you to

1. Write, compile and run a number of programs in C which make use of pipes and redirection.
2. How to redirect standard output and input using ‘cat’ command and ‘|’ in Linux

**Pipes**

* Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process
* It is possible to have a series of processes arranged in a pipeline, with a pipe between each pair of processes in the series.
* Implementation: A pipe can be implemented as a 10k buffer in main memory with 2 pointers, one for the FROM process and one for TO process
* One process cannot read from the buffer until another has written to it
* The UNIX command-line interpreter (e.g., csh) provides a pipe facility.

**% prog | more**

* This command runs the prog1 program and sends its output to the more program.

**Pipe System Call**

* pipe() is a system call that facilitates inter-process communication. It opens a **pipe**, which is an area of main memory that is treated as a "virtual file". The pipe can be used by the creating process, as well as all its child processes, for reading and writing.
* One process can write to this "virtual file" or pipe and another related process can read from it.
* If a process tries to read before something is written to the pipe, the process is suspended until something is written.
* The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe. Recall that the open system call allocates only one position in the open file table.

Syntax in a C program:

#include <unistd.h>

int pip[2];

(void) pipe(pip);

With error checking:

#include <unistd.h>

int pip[2];

int result;

result = pipe(pip);

if (result == -1)

{

perror("pipe");

exit(1);

}

**Exercise 1:** Write a program in which a parent writes a string to a pipe and the child reads the string.

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#define BUFSIZE 10

int main(void) {

char bufin[BUFSIZE] = "empty";

char bufout[] = "hello";

int bytesin;

pid\_t childpid;

int fd[2];

if (pipe(fd) == -1) {

perror("Failed to create the pipe");

return 1;

}

bytesin = strlen(bufin);

childpid = fork();

if (childpid == -1) {

perror("Failed to fork");

return 1;

}

if (childpid) /\* parent code \*/

write(fd[1], bufout, strlen(bufout)+1);

else /\* child code \*/

bytesin = read(fd[0], bufin, BUFSIZE);

fprintf(stderr, "[%ld]:my bufin is {%.\*s}, my bufout is {%s}\n",

(long)getpid(), bytesin, bufin, bufout);

return 0;

}

**Exercise 2:** Write a program to execute the equivalent of ls -l | sort -n +4.

#include <errno.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main(void) {

pid\_t childpid;

int fd[2];

if ((pipe(fd) == -1) || ((childpid = fork()) == -1)) {

perror("Failed to setup pipeline");

return 1;

}

if (childpid == 0) { /\* ls is the child \*/

if (dup2(fd[1], STDOUT\_FILENO) == -1)

perror("Failed to redirect stdout of ls");

else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))

perror("Failed to close extra pipe descriptors on ls");

else {

execl("/bin/ls", "ls", "-l", NULL);

perror("Failed to exec ls");

}

return 1;

}

if (dup2(fd[0], STDIN\_FILENO) == -1) /\* sort is the parent \*/

perror("Failed to redirect stdin of sort");

else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))

perror("Failed to close extra pipe file descriptors on sort");

else {

execl("/bin/sort", "sort", "-n", "+4", NULL);

perror("Failed to exec sort");

}

return 1;

}

**Exercise 3:** Write a program that redirects standard output to the file my.file. and then appends a short message to that file.

**Exercise 4:** Write a program that will perform the following tail -5 alpha.txt | grep ee | sort.

**Exercise 5:**

Sometimes you will want to put output of a command in a file, or you may want to issue another command on the output of one command. This is known as redirecting output. Redirection is done using either the ">" (greater-than symbol), or using the "|" (pipe) operator which sends the standard output of one command to another command as standard input.

The **cat** command concatenates files and puts them all together to the standard output. By redirecting this output to a file, this file name will be created - or overwritten if it already exists, so take care.

1. Create a file named ‘tmp.txt’ having contents ‘a b c’.
2. cat tmp.txt
3. cat > tmp.txt

1

2

3

ctrl d

1. cat tmp.txt
2. cat >> tmp.txt

a

b

c

ctrl d

1. cat < tmp.txt
2. cat < tmp.txt > tmp2.txt
3. cat tmp2.txt
4. Ps > file1.txt
5. ps | more

**CSC 322- Operating System**

***Week 8-10: Signals & Threads***

**Learning Objectives:** The objective of this exercise is to get you familiar with the working of signals and Threads.

**Threads**

Just as a process is identified through a process ID, a thread is identified by a thread ID. But interestingly, the similarity between the two ends here.

* A process ID is unique across the system where as a thread ID is unique only in context of a single process.
* A process ID is an integer value but the thread ID is not necessarily an integer value. It could well be a structure
* A process ID can be printed very easily while a thread ID is not easy to print.

The above points give an idea about the difference between a process ID and thread ID.

Thread ID is represented by the type **‘pthread\_t’**.  In most of the cases this type is a structure, so there has to be a function that can compare two thread IDs.

**#include <pthread.h>**

**int pthread\_equal(pthread\_t tid1, pthread\_t tid2);**

So as you can see that the above function takes two thread IDs and returns nonzero value if both the thread IDs are equal or else it returns zero.

Another case may arise when a thread would want to know its own thread ID. For this case the following function provides the desired service.

**#include <pthread.h>**

**pthread\_t pthread\_self(void);**

So we see that the function ‘pthread\_self()’ is used by a thread for printing its own thread ID.

### Thread Creation

Normally when a program starts up and becomes a process, it starts with a default thread. So we can say that every process has at least one thread of control.  A process can create extra threads using the following function :

**#include <pthread.h>**

**int pthread\_create(pthread\_t \*restrict tidp, const pthread\_attr\_t \*restrict attr, void \*(\*start\_rtn)(void), void \*restrict arg)**

The above function requires four arguments:

* The first argument is a pthread\_t type address. Once the function is called successfully, the variable whose address is passed as first argument will hold the thread ID of the newly created thread.
* The second argument may contain certain attributes which we want the new thread to contain.  It could be priority etc.
* The third argument is a function pointer. This is something to keep in mind that each thread starts with a function and that functions address is passed here as the third argument so that the kernel knows which function to start the thread from.
* As the function (whose address is passed in the third argument above) may accept some arguments also so we can pass these arguments in form of a pointer to a void type. Now, why a void type was chosen? This was because if a function accepts more than one argument then this pointer could be a pointer to a structure that may contain these arguments.

### A Practical Thread Example

Following is the example code where we tried to use all the three functions discussed above.

#include<stdio.h>

#include<string.h>

#include<pthread.h>

#include<stdlib.h>

#include<unistd.h>

pthread\_t tid[2];

void\* doSomeThing(void \*arg)

{

unsigned long i = 0;

pthread\_t id = pthread\_self();

if(pthread\_equal(id,tid[0]))

{

printf("\n First thread processing\n");

}

else

{

printf("\n Second thread processing\n");

}

for(i=0; i<(0xFFFFFFFF);i++);

return NULL;

}

int main(void)

{

int i = 0;

int err;

while(i < 2)

{

err = pthread\_create(&(tid[i]), NULL, &doSomeThing, NULL);

if (err != 0)

printf("\ncan't create thread :[%s]", strerror(err));

else

printf("\n Thread created successfully\n");

i++;

}

sleep(5);

return 0;

}

So what this code does is :

* It uses the pthread\_create() function to create two threads
* The starting function for both the threads is kept same.
* Inside the function ‘doSomeThing()’, the thread uses pthread\_self() and pthread\_equal() functions to identify whether the executing thread is the first one or the second one as created.
* Also, Inside the same function ‘doSomeThing()’ a for loop is run so as to simulate some time consuming work.

Now, when the above code is run, following was the output :

**$ ./threads**

**Thread created successfully**

**First thread processing**

**Thread created successfully**

**Second thread processing**

As seen in the output, first thread is created and it starts processing, then the second thread is created and then it starts processing. Well one point to be noted here is that the order of execution of threads is not always fixed. It depends on the OS scheduling algorithm.

**Exercise 2:** write a multithreaded program with two threads in which one thread prints A and the other prints B.

**Exercise 3.** Write a multithreaded program that will multiply a 3 x 3 matrix with itself and write the result in another matrix. The program should assign one “row” to each thread. The resultant matrix will be printed when all three threads are done. Use Condition variable to accomplish the synchronization not pthread\_join.

**Exercise 4:** Write a program that uses asynchronous I/O with signals to copy a file while doing other work.

#include <errno.h>

#include <fcntl.h>

#include <signal.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <sys/stat.h>

#include "asyncmonitorsignal.h"

#define BLKSIZE 1024

#define MODE (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)

void dowork(void);

int main(int argc, char \*argv[]) {

char buf[BLKSIZE];

int done = 0;

int error;

int fd1;

int fd2;

/\* open the file descriptors for I/O \*/

if (argc != 3) {

fprintf(stderr, "Usage: %s filename1 filename2\n", argv[0]);

return 1;

}

if ((fd1 = open(argv[1], O\_RDONLY)) == -1) {

fprintf(stderr, "Failed to open %s:%s\n", argv[1], strerror(errno));

return 1;

}

if ((fd2 = open(argv[2], O\_WRONLY | O\_CREAT | O\_TRUNC, MODE)) == -1) {

fprintf(stderr, "Failed to open %s: %s\n", argv[2], strerror(errno));

return 1;

}

if (initsignal(SIGRTMAX) == -1) {

perror("Failed to initialize signal");

return 1;

}

if (initread(fd1, fd2, SIGRTMAX, buf, BLKSIZE) == -1) {

perror("Failed to initate the first read");

return 1;

}

for ( ; ; ) {

dowork();

if (!done)

if (done = getdone())

if (error = geterror())

fprintf(stderr, "Failed to copy file:%s\n", strerror(error));

else

fprintf(stderr, "Copy successful, %d bytes\n", getbytes());

}

}

**Exercise 1:** Write a program that blocks and unblocks SIGINT?

**CSC 322- Operating System**

***Week 11*: *System-V inter process communication mechanisms. Semaphore array***

**Learning Objectives:** The objective of this exercise is to get you familiar with

1. *System-V inter process communication mechanisms*
2. *Semaphore*
3. *Producer-Consumer Problem*

**Exercise 1:** To implement the Producer- Consumer problem using semaphores.

**Steps:**

1. Create two functions called producer and consumer.

2. Set semaphore variable as 1.

3. When producer active set the semaphore variable as 1 and allow producer to put data into the buffer and don’t allow consumer to consume anything.

4. After producer complete the process release the semaphore and signal the consumer.

5. When consumer active again set the semaphore variable as 1 and allow the consumer to get data from buffer and don’t allow the producer to add data.

6. After the consumer taken release the semaphore variable and signal the producer.

**Source Code:**

#include<stdio.h>

#include<conio.h>

int n\_semaphore=0; // keep track of no of item in the buffer

int s\_semaphore=1; // to enforce mutual exclusion

char s;

void producer()

{

s\_semaphore=0; // set semaphore to avoid access to consumer

if(!s\_semaphore)

printf("Now producer can add data to buffer\n");

else

printf("Critical Region \n");

s\_semaphore=1; // release semaphore

signal\_c(); // call to consumer

}

void consumer()

{

buffer\_check(); // check buffer is empty or not

s\_semaphore=0; // set semaphore to avoid access to producer

if(!s\_semaphore)

printf("Consumer takes from the buffer\n");

else

printf("Critical Region \n");

s\_semaphore=1; // release semaphore

signal\_p(); // call to producer

}

signal\_c()

{ n\_semaphore=n\_semaphore+1;

consumer();

return 0;

} signal\_p()

{ n\_semaphore=n\_semaphore-1;

printf("Enter n to stop\n");

scanf("%c",&s);

if(s=='n')

exit();

return 0;

}

buffer\_check()

{

if(n\_semaphore<=0)

{

printf("Buffer is empty\n");

exit();

}

return 0;

}

void main()

{

clrscr();

n\_semaphore=0;

while(1)

{

producer();

}

}

**Sample Output;**

Now producer can add data to buffer

Consumer takes from the buffer

Enter n to stop

Now producer can add data to buffer

Consumer takes from the buffer

Enter n to stop

n

**Exercise 2:** Write a function that creates and initializes a semaphore set containing a single semaphore.

**CSC 322- Operating System**

***Week 13& 14*: *Connection Oriented Communication, Little and Big Endian***

**Learning Objectives:**

The objectives of this first experiment are:

1. Students should be able to understand the basics of Socket Programming.
2. Understand the working of Client-Server architecture.
3. Write and compile Client server programs

**Your lab report is expected to contain the following for each exercise:**

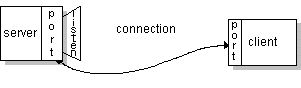
* **C Source Code**

**What is a Socket?**

The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

### Exercise 1: Server

Let’s build a very simple web server. The steps to make a web server are as follows:

1. Create socket  
2. Bind to address and port  
3. Put in listening mode  
4. Accept connections and process there after

**Source Code**

**Server**

|  |  |
| --- | --- |
|  | include<stdio.h>  #include<string.h>    //strlen  #include<sys/socket.h>  #include<arpa/inet.h> //inet\_addr  #include<unistd.h>    //write    int main(int argc , char \*argv[])  {      int socket\_desc , client\_sock , c , read\_size;      struct sockaddr\_in server , client;      char client\_message[2000];        //Create socket      socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);      if (socket\_desc == -1)      {          printf("Could not create socket");      }      puts("Socket created");        //Prepare the sockaddr\_in structure      server.sin\_family = AF\_INET;      server.sin\_addr.s\_addr = INADDR\_ANY;      server.sin\_port = htons( 8888 );        //Bind      if( bind(socket\_desc,(struct sockaddr \*)&server , sizeof(server)) < 0)      {          //print the error message          perror("bind failed. Error");          return 1;      }      puts("bind done");        //Listen      listen(socket\_desc , 3);        //Accept and incoming connection      puts("Waiting for incoming connections...");      c = sizeof(struct sockaddr\_in);        //accept connection from an incoming client      client\_sock = accept(socket\_desc, (struct sockaddr \*)&client, (socklen\_t\*)&c);      if (client\_sock < 0)      {          perror("accept failed");          return 1;      }      puts("Connection accepted");        //Receive a message from client      while( (read\_size = recv(client\_sock , client\_message , 2000 , 0)) > 0 )      {          //Send the message back to client          write(client\_sock , client\_message , strlen(client\_message));      }        if(read\_size == 0)      {          puts("Client disconnected");          fflush(stdout);      }      else if(read\_size == -1)      {          perror("recv failed");      }        return 0;  } |

The above code example will start a server on localhost (127.0.0.1) port 8888.Once it receives a connection, it will read some input from the client and reply back with the same message.  
To test the server run the server and then connect from another terminal using the telnet command like this

$ telnet localhost 8888

### Client

Now instead of using the telnet program as a client, why not write our own client program. Quite simple again

|  |  |
| --- | --- |
|  | #include<stdio.h> //printf  #include<string.h>    //strlen  #include<sys/socket.h>    //socket  #include<arpa/inet.h> //inet\_addr    int main(int argc , char \*argv[])  {      int sock;      struct sockaddr\_in server;      char message[1000] , server\_reply[2000];        //Create socket      sock = socket(AF\_INET , SOCK\_STREAM , 0);      if (sock == -1)      {          printf("Could not create socket");      }      puts("Socket created");        server.sin\_addr.s\_addr = inet\_addr("127.0.0.1");      server.sin\_family = AF\_INET;      server.sin\_port = htons( 8888 );        //Connect to <span id="k2085f6x0y\_9" class="k2085f6x0y">remote server</span>      if (connect(sock , (struct sockaddr \*)&server , sizeof(server)) < 0)      {          perror("connect failed. Error");          return 1;      }        puts("Connected\n");        //keep communicating with server      while(1)      {          printf("Enter message : ");          scanf("%s" , message);            //Send some data          if( send(sock , message , strlen(message) , 0) < 0)          {              puts("Send failed");              return 1;          }            //Receive a reply from the server          if( recv(sock , server\_reply , 2000 , 0) < 0)          {              puts("recv failed");              break;          }            puts("Server reply :");          puts(server\_reply);      }        close(sock);      return 0;  } |

The above program will connect to localhost port 8888 and then ask for commands to send. Here is an example, how the output would look

$ gcc client.c && ./a.out  
Socket created  
Connected

Enter message : hi  
Server reply :  
hi  
Enter message : how are you

**Exercise 2:** Write a pair of client server programs in which client sends character A to the server. The server reads the character, increments to B and transmits it back to client.

**Exercise 3:** Write a pair of client server programs in which client sends integer to the server . The server reads the integer, increments it and transmits it back to client.

**Exercise 4:** Write a simple Client-Server application in C. The client sends a message to the server and server sends back a response? The server should be a multi-threaded server; meaning that multiple clients can connect to the server simultaneously. The program should be compiled using the make utility.

**CSC 322- Operating System**

***Week 15& 16*: *Message Queues***

**Learning Objectives:**

The objectives of this first experiment are:

1. Students should be able to understand the basics of Message Queues
2. Understand the working of Message Queues.

**Introduction to Message Queues**

Message queues allow one or more processes to write messages that will be read by one or more reading processes. Linux maintains a list of message queues, the msgque vector: each element of which points to a msqid\_ds data structure that fully describes the message queue. When message queues are created, a new msqid\_ds data structure is allocated from system memory and inserted into the vector.

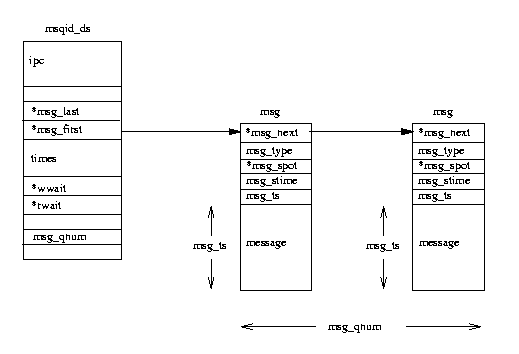


Figure: System V IPC Message Queues

Each msqid\_ds data structure contains an ipc\_perm data structure and pointers to the messages entered onto this queue. In addition, Linux keeps queue modification times such as the last time that this queue was written to and so on. The msqid\_ds also contains two wait queues: one for the writers to the queue and one for the readers of the queue.

Each time a process attempts to write a message to the write queue, its effective user and group identifiers are compared with the mode in this queue's ipc\_perm data structure. If the process can write to the queue then the message may be copied from the process' address space into a msg data structure and put at the end of this message queue.

Each message is tagged with an application specific type, agreed between the cooperating processes. However, there may be no room for the message as Linux restricts the number and length of messages that can be written. In this case the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. It will be awakened when one or more messages have been read from this message queue.

Reading from the queue is similar. Again, the process' access rights to the write queue are checked. A reading process may choose to either get the first message in the queue regardless of its type or select messages with particular types. If no messages match this criteria the reading process will be added to the message queue's read wait queue and the scheduler run. When a new message is written to the queue this process will be awakened and run again.

Each message has a message type associated with it. A Message Queue reader can specify which type of message that it will read. Or it can say that it will read all messages in order.

It is quite possible to have any number of Msg Queue readers, or writers. In fact the same process can be both a writer and a reader.

Each message structure must start with a long message type:

**struct mymsg**

**{  
           long msg\_type;  
           char mytext[512]; /\* rest of message \*/  
           int somethingelse;  
      };**

Each message queue is limited in terms of both the maximum number of messages it can contain and the maximum number of bytes it may contain.

New messages cannot be added if either limit is hit (new writes will normally block).

On linux, these limits are defined as (in /usr/include/linux/msg.h):   
        –MSGMAX 8192 /\*total number of messages \*/   
        –MSBMNB 16384 /\* max bytes in a queue \*/

**Creating a Message Queue:**

#include <sys/types.h>  
#include <sys/ipc.h>  
#include <sys/msg.h>  
  
            **int msgget (key\_t key, int msgflg);**

The key parameter is either a non-zero identifier for the queue to be created or the value IPC\_PRIVATE, which guarantees that a new queue is created.

The msgflg parameter is the read-write permissions for the queue OR’d with one of two flags:   
  
IPC\_CREAT will create a new queue or return an existing one.  
  
IPC\_EXCL added will force the creation of a new queue, or return an error.

**Writing to a Message Queue:**  
  
**int msgsnd (int msqid, const void \* msg\_ptr, size\_t msg\_size, int msgflags);**   msgqid is the id returned from the msgget call    msg\_ptr is a pointer to the message structure   
   msg\_size is the size of that structure   
   msgflags defines what happens when no message of the appropriate type is waiting, and can be set to the following:

          IPC\_NOWAIT (non-blocking, return –1 immediately if queue is empty)

**Reading from a Message Queue:**  
  
      **int msgrcv(int msqid, const void \* msg\_ptr, size\_t msg\_size, long msgtype, int msgflags);**  
  
   msgqid is the id returned from the msgget call   
   msg\_ptr is a pointer to the message structure   
   msg\_size is the size of that structure   
   msgtype is set to: = 0 first message available in FIFO stack

> 0 first message on queue whose type equals type

 msgflags defines what happens when no message of the appropriate type is waiting, and can be set to the following:

  IPC\_NOWAIT (non-blocking, return –1 immediately if queue is empty)

**Message Queue Control:**

    int msgctl(int msqid, int cmd, struct msqid\_ds \* buf);

**Exercise 1:** A pair of programs in which one process writes messages into message queue and the other process reads from it.

**Exercise 2:** Write a code for producer & consumer programs. The producer will produce a message of type passed in as command line argument and store it in the message queue **.**The consumer will read one message of type long passed in as command line argument and display it **.**Assume that both producer and consumer use 555 as the key for the message queue. A sample run of the pair is given below.

$mproducer.exe 5

$mconsumer.exe 5

The current state of the message queue is as follows :

PID of the last proc that wrote a message : 2345

PID of the last proc that read a message : 0

Current number of bytes on queue : 44

Current number of messages on queue : 1

Message read from msg queue is

I am producer process and my proc id is 2345

The current state of the message queue is as follows :

PID of the last proc that wrote a message : 2345

PID of the last proc that read a message : 2346

Current number of bytes on queue : 0

Current number of messages on queue : 0