|  |  |
| --- | --- |
| **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. |

**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**

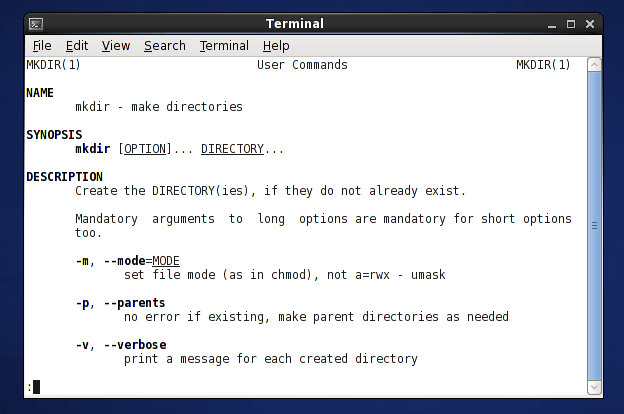
Press the Enter key after typing each command. The general command syntax is the command name, a space, and zero or more parameters. Do not forget to type the space between the command names and the parameters. For example, a **space** separates ***cd*** from **..** in the ***cd ..*** command.

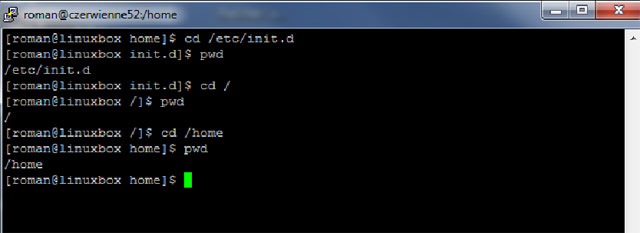
**Exercise 1:** Use different shell commands like, mkdir, ls, rm, mv, cd,cp, gcc, pwd...

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(),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