**OSPP Tutorial – Session 1**

Use jslinux for online Linux Environment

[JSLinux (bellard.org)](https://bellard.org/jslinux/)

Select:

|  |  |  |
| --- | --- | --- |
| riscv64 | Buildroot (Linux) | Console |

ls

ls - alF

**mkdir, rmdir,lpr rm.txt,mv(move or rename of file)**

**cp, cat, cd, mv, grep**

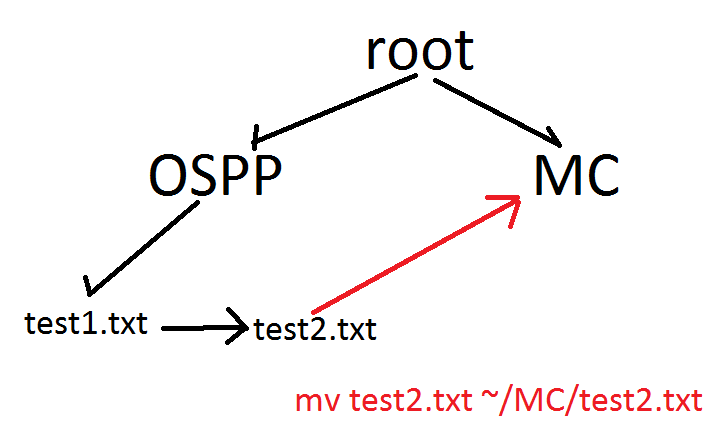
To create a file:

**vi test1.text**

**press i to insert word**

**press ESC and type :wq**

**mv**



**grep**

The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (grep stands for globally search for regular expression and print out).  
**Syntax:** 

**grep [options] pattern [files]**

**Options Description**

**-c** : This prints only a count of the lines that match a pattern

**-h :** Display the matched lines, but do not display the filenames.

**-i :** Ignores, case for matching

**-l :** Displays list of a filenames only.

**-n :** Display the matched lines and their line numbers.

**-v :** This prints out all the lines that do not matches the pattern

**-e exp :** Specifies expression with this option. Can use multiple times.

**-f file :** Takes patterns from file, one per line.

**-E :** Treats pattern as an extended regular expression (ERE)

**-w :** Match whole word

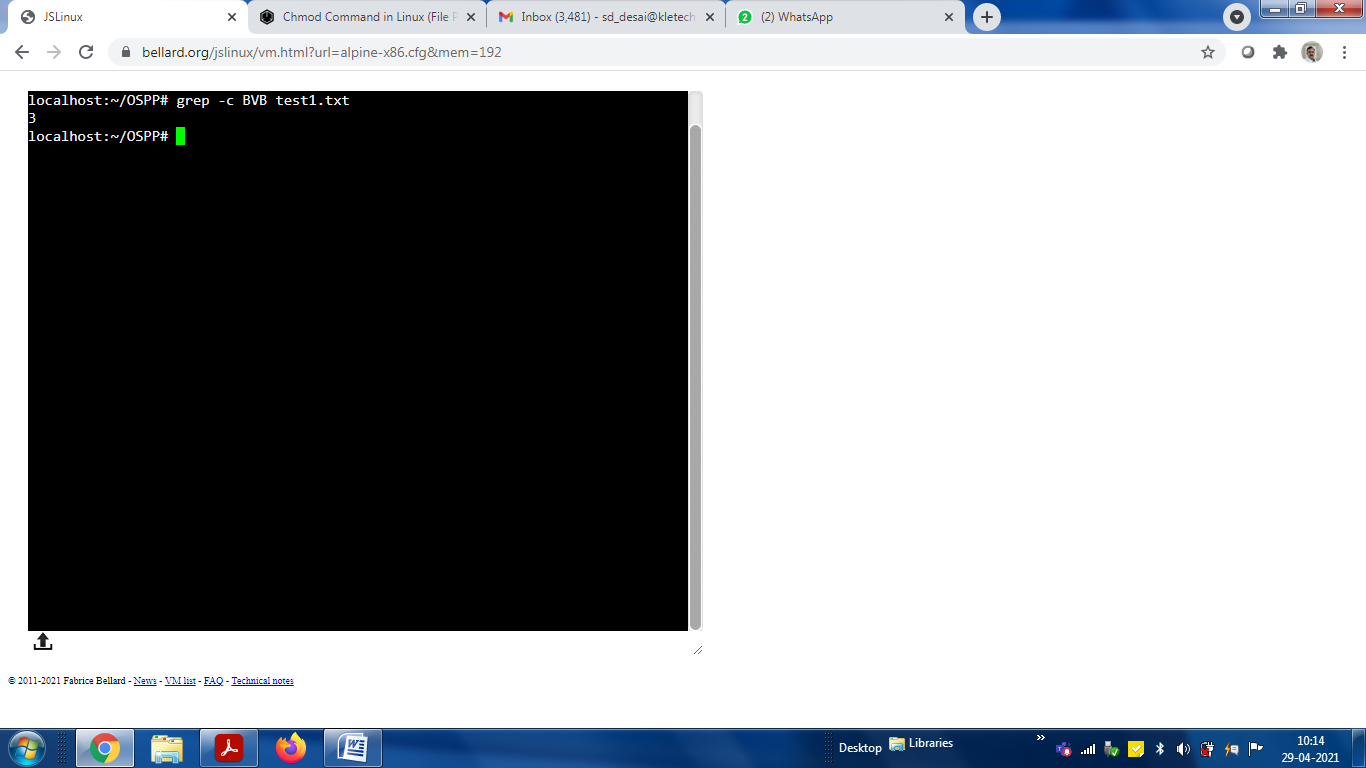
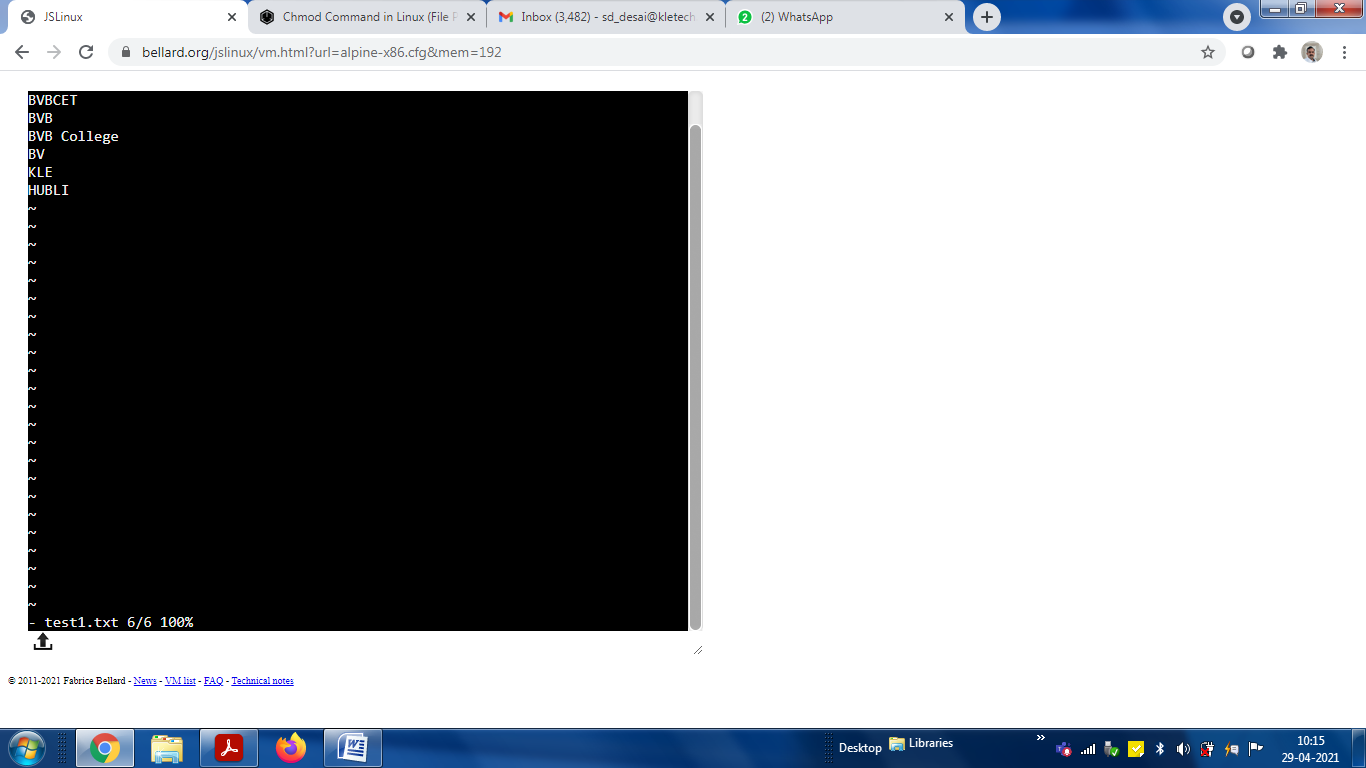
**-o :** Print only the matched parts of a matching line,

with each such part on a separate output line.

**-A n** **:** Prints searched line and nlines after the result.

**-B n :** Prints searched line and n line before the result.

**-C n :** Prints searched line and n lines after before the result.

**chmod**

File permissions can be viewed using the [ls](https://linuxize.com/post/how-to-list-files-in-linux-using-the-ls-command/) command:

ls -l filename.txtCopy

-rw-r--r-- 12 linuxize users 12.0K Apr 8 20:51 filename.txt

|[-][-][-]- [------] [---]

| | | | | | |

| | | | | | +-----------> 7. Group

| | | | | +-------------------> 6. Owner

| | | | +--------------------------> 5. Alternate Access Method

| | | +----------------------------> 4. Others Permissions

| | +-------------------------------> 3. Group Permissions

| +----------------------------------> 2. Owner Permissions

+------------------------------------> 1. File Type

Copy

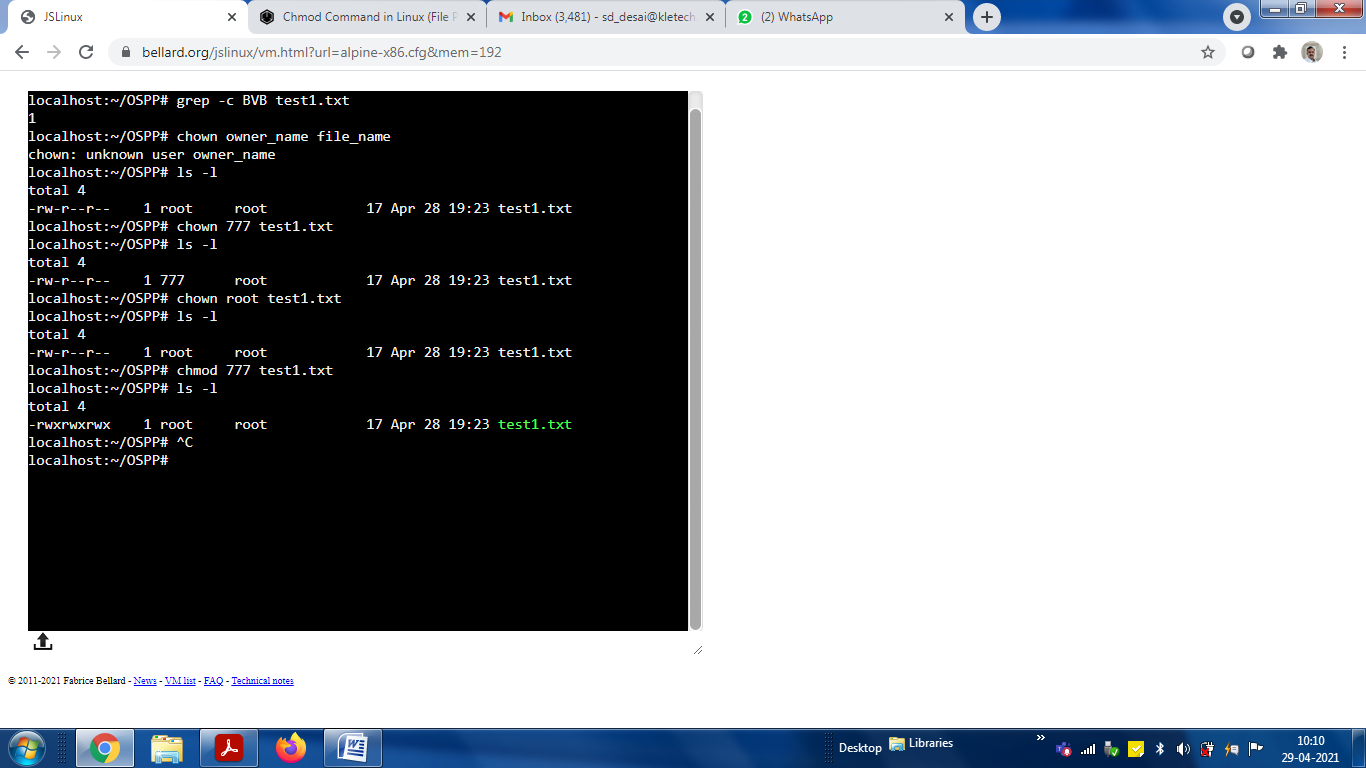
The first character shows the file type. It can be a regular file (-), directory (d), a [symbolic link](https://linuxize.com/post/how-to-create-symbolic-links-in-linux-using-the-ln-command/) (l), or any other special type of file.

The next nine characters represent the file permissions, three triplets of three characters each. The first triplet shows the owner permissions, the second one group permissions, and the last triplet shows everybody else permissions. The permissions can have a different meaning depending on the file type.

In the example above (rw-r--r--) means that the file owner has read and write permissions (rw-), the group and others have only read permissions (r--).

Each of the three permission triplets can be constructed of the following characters and have a different effects, depending on whether they are set to a file or to a directory:

|  |  |  |
| --- | --- | --- |
| **Read** | - | The file is not readable. You cannot view the file contents. |
|  | r | The file is readable. |
| **Write** | - | The file cannot be changed or modified. |
|  | w | The file can be changed or modified. |
| **Execute** | - | The file cannot be executed. |
|  | x | The file can be executed. |



**Shell Scripting**

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

* The Bourne Shell
* The C Shell
* The Korn Shell
* The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

#!/bin/sh

# Author : Zara Ali

# Copyright (c) Tutorialspoint.com

# Script follows here:

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"

Here is a sample run of the script −

$./test.sh

What is your name?

Zara Ali

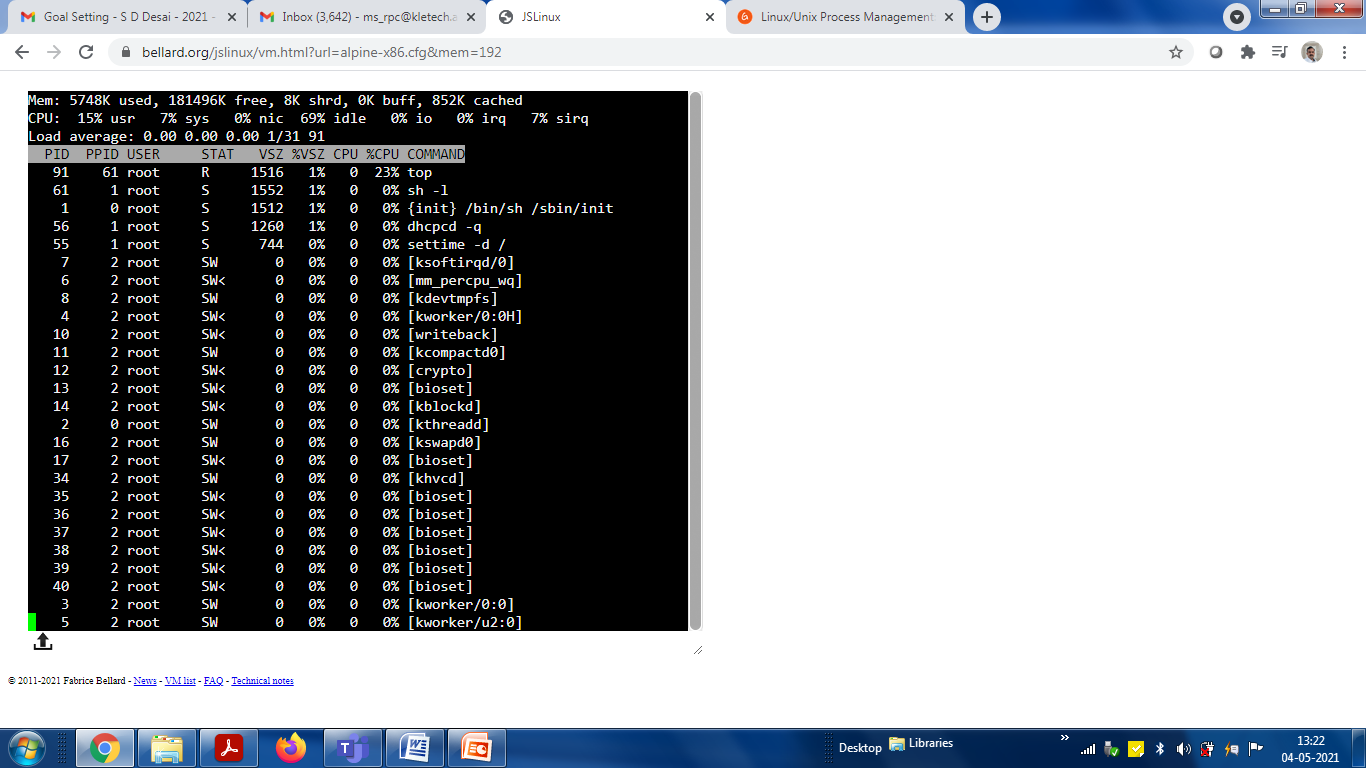
Hello, Zara Ali

$

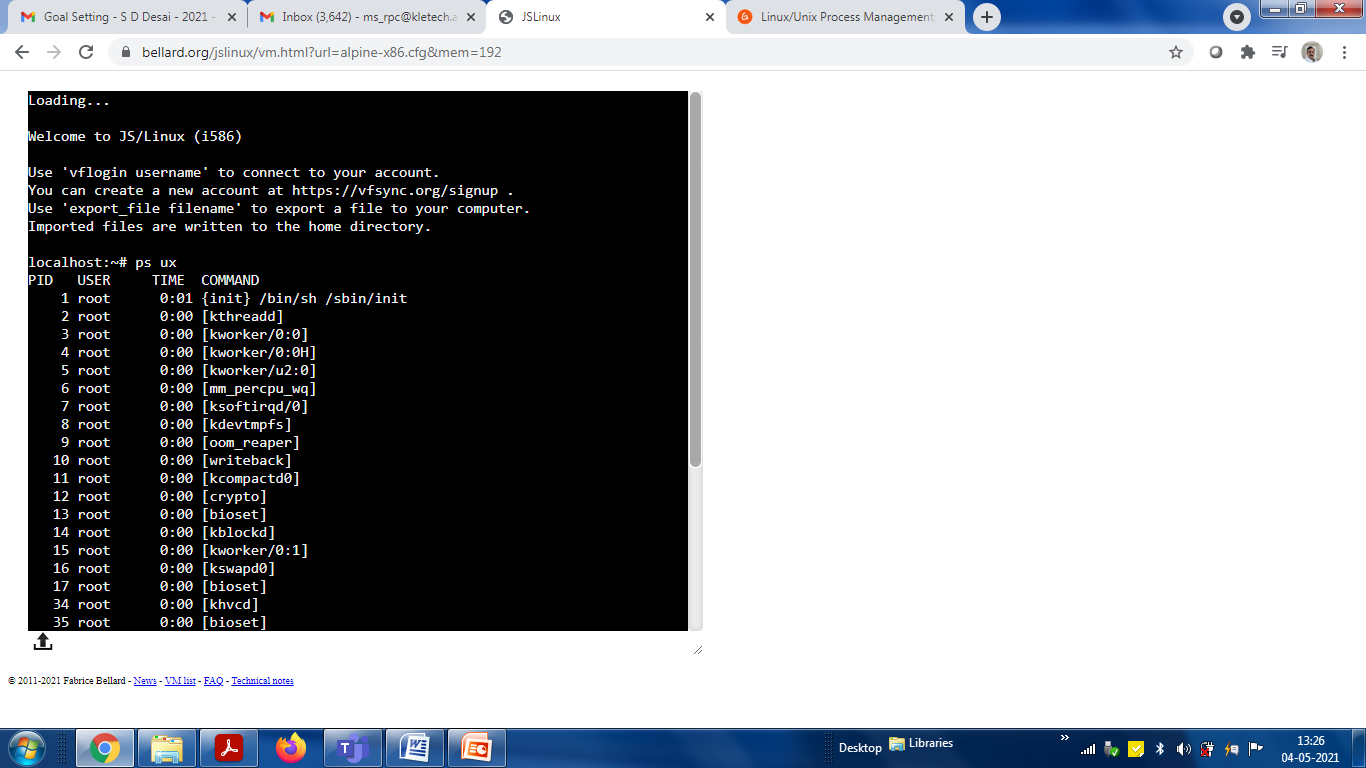
**Program on**

pid

localhost:~# top



localhost:~# ps ux



To know PID and Parent PID

1. Run following program and guess, which is Parent ID and which is Child ID?

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>  #include<unistd.h>  #include<sys/wait.h>    int main(int argc, char\* argv[])  {  int id = fork();  printf("%d\n”, getpid());  return 0;  } |

localhost:~# gcc file1.c

localhost:~# ./a.out

129

130

1. Now run this and build family tree with PID numbers

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>  #include<unistd.h>  #include<sys/wait.h>    int main(int argc, char\* argv[])  {  int id = fork();  printf("Current ID: %d\n, parent ID: %d\n",getpid(),getppid());  return 0;  } |

localhost:~# gcc file1.c

localhost:~# ./a.out

Current ID: 121

, parent ID: 61

Current ID: 122

, parent ID: 121

**fork**

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>  #include<unistd.h>  #include<sys/wait.h>    int main()  { fork( ); printf(“\n hello OSSP”); } |

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>  #include<unistd.h>  #include<sys/wait.h>    int main()  {  fork();  fork();  fork();  printf("hello\n");  return 0;  } |

**vfork**

|  |
| --- |
| #include <stdio.h>  #include<stdlib.h>  #include<unistd.h>  #include<sys/wait.h>  int glob = 6;  int main(void)  {  int var;  pid\_t pid;    var =88;  /\* printf("Before vfork\n");\*/  **pid = fork();**    if (pid ==0) {  glob++;  var++;  \_exit(0);  }    /\* Parent continues here\*/  printf("pid = %d, glob = %d, var = %d\n",getpid(), glob, var);  exit(0);  } |

If we run this code with “fork” then OS may schedule either parent or child process. And Parent may run first hence global and local variable values are printed without update

If we run this code with “vfork” then OS will first schedule child. Where global and local variable gets updated. But child exits abnormally

Now parent will run and display updated value.

Thats the difference...

**“vfork guarantees that the child runs first, until the child calls exec or exit.”**

**zombie and orphan**

|  |
| --- |
| #include<stdio.h>  #include<unistd.h>  #include<stdlib.h>  int main()  {  pid\_t pid;  pid = fork();    if (pid ==0){  printf("I am child and my PID is %d\n",getpid());  printf("My parent PID is %d\n", getppid());  }  else {  sleep(10);  printf("I am parent and my PID is %d\n", getpid());  printf("My child PID is %d\n", pid);  }  return 0;  } |

Wehen we run this making parent to sleep for 10 sec, and then run ps command we get

**localhost:~# gcc file1.c**

**localhost:~# ./a.out &**

**localhost:~#ps**

**PID TTY TIME CMD**

**173 tty1 00.00.00 a.out <defunct>**

**Orphan**

<https://www.youtube.com/watch?v=DYDHNL_AImo>

|  |
| --- |
| #include<stdio.h>  #include<unistd.h>  #include<stdlib.h>  int main()  {  pid\_t pid;  pid = fork();    if (pid ==0){  sleep(2);  printf("I am child and my PID is %d\n",getpid());  printf("My parent PID is %d\n", getppid());  }  else {  printf("I am parent and my PID is %d\n", getpid());  printf("My child PID is %d\n", pid);  }  return 0;  } |

**race condition**

|  |
| --- |
| #include<pthread.h>  #include<stdio.h>  #include<unistd.h>  void \*fun1();  void \*fun2();  int shared=1; //shared variable  int main()  {  pthread\_t thread1, thread2;  pthread\_create(&thread1, NULL, fun1, NULL);  pthread\_create(&thread2, NULL, fun2, NULL);  pthread\_join(thread1, NULL);  pthread\_join(thread2,NULL);  printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable  }  void \*fun1()  {  int x;  x=shared;//thread one reads value of shared variable  printf("Thread1 reads the value of shared variable as %d\n",x);  x++; //thread one increments its value  printf("Local updation by Thread1: %d\n",x);  sleep(1); //thread one is preempted by thread 2  shared=x; //thread one updates the value of shared variable  printf("Value of shared variable updated by Thread1 is: %d\n",shared);  }  void \*fun2()  {  int y;  y=shared;//thread two reads value of shared variable  printf("Thread2 reads the value as %d\n",y);  y--; //thread two increments its value  printf("Local updation by Thread2: %d\n",y);  sleep(1); //thread two is preempted by thread 1  shared=y; //thread one updates the value of shared variable  printf("Value of shared variable updated by Thread2 is: %d\n",shared);  } |

When we execute this program, two threads thread1 and thread2 try to access and modify the value of “shared” variable concurrently. This leads to race condition. Hence final value of shared will be either 2 or 0 depending on who wins the race...

This shall be solved by using mutesx or semaphore.

**Wait()**

|  |
| --- |
| #include <stdio.h>  #include<unistd.h>  #include<sys/types.h>  #include<sys/wait.h>  int main()  {  pid\_t p;  printf("before fork\n");  p=fork();  if(p==0)//child  {  printf("I am child having id %d\n",getpid());  printf("My parent's id is %d\n",getppid());  }  else//parent  {  wait(NULL);  printf("My child's id is %d\n",p);  printf("I am parent having id %d\n",getpid());  }  printf("Common\n");  } |

**If we run above code without “wait(NULL)” then output may be either from child or from parent or mix.**

**wait() system call is added to the parent section of the code. Hence, the moment processor starts processing the parent, the parent process is suspended because the very first statement is wait(NULL). Thus, first, the child process runs, and the output lines are all corresponding to the child process. Once the child process finishes, parent resumes and prints all its printf() statements. The NULL inside the wait() means that we are not interested to know the status of change of state of child process.**

**We should not call wait() in child process, because in that case child will be waiting for its child to complete first.. and that may not be the case.**

[Program to use wait system call || wait() || Program - YouTube](https://www.youtube.com/watch?v=eSoOghJZm-c)

**Viva questions on wait() system call**

Q1. Can we use wait() to make the child process wait for the parent process to finish?  
Q2. What does the wait() system call return on success?

**Practice Program for wait() system call**

Q1. Write a program to create two child process. The parent process should wait for both the child to finish.  
Q2. Create a parent-child relationship between two process. The parent should print two statements:  
A) its own PID  
B) PID of its child  
The child should print two statements:  
C) its own PID  
D) PID of its parent  
Make use of wait() in such a manner that the order of the four statements A, B, C and D is:  
A  
C  
D  
B  
You are free to use any other relevant statement/printf as you desire and their order of execution does not matter.

**PIPES and FIFOs**

**PIPES in single window**

|  |
| --- |
| #include<stdio.h>  #include<unistd.h>  #include<sys/types.h>  #include<sys/wait.h>  int main()  {  int fd[2],n;  char buffer[100];  pid\_t p;  pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]  p=fork();  if(p>0) //parent  {  printf("Parent Passing value to child\n");  write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe  }  else // child  {  printf("Child printing received value\n");  n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe  write(1,buffer,n);  }  } |

**IPC by FIFO (Named PIPEs)**

|  |  |  |
| --- | --- | --- |
| //Program1: Creating fifo/named pipe ( mkfifo.c )  #include<stdio.h>  #include<sys/types.h>  #include<sys/stat.h>  int main()  {  int res;  res = mkfifo("fifo1",0777); //creates a named pipe with the name fifo1  printf("named pipe created\n");  } | //**Program2:**Writing to a fifo/named pipe ( sender.c )  #include<unistd.h>  #include<stdio.h>  #include<fcntl.h>  int main()  {  int res,n;  res=open("fifo1",O\_WRONLY);  write(res,"Message",7);  printf("Sender Process %d sent the data\n",getpid());  } | //**Program 3**: Reading from the named pipe ( 3.c )  #include<unistd.h>  #include<stdio.h>  #include<fcntl.h>  int main()  {  int res,n;  char buffer[100];  res=open("fifo1",O\_RDONLY);  n=read(res,buffer,100);  printf("Reader process %d started\n",getpid());  printf("Data received by receiver %d is: %s\n",getpid(), buffer);  } |

How to run

|  |  |
| --- | --- |
| Command | Remark / Output |
| vi mkfifo.c | Edit the main program and save |
| gcc mkfifo.c  ./a.out | * **Named pipe created** |
| ls fifo1 | * **Fifo1** |
| ls –l fifo1 | * **prwxr** – xp...... |
|  |  |
| vi sender.c | Edit sender program and save |
| gcc –o sender sender.c |  |
| ./sender | No display (Bcoz Blocked mode) |
| Ctrl+c |  |
|  |  |
| vi receiver.c |  |
| gcc –o receiver receiver.c |  |
| ./receiver | No display (Bcoz Blocked mode) |
| Ctrl+C |  |
| ./sender & ./receiver | Running sender in background and Receiver in foreground |
|  | Program for IPC using named pipes (mkfifo()) |

|  |  |  |
| --- | --- | --- |
| #include<stdio.h>  void main()  {  int f1,f2;  f1 = mkfifo("pipeA",0666);  if(f1<0)  printf("\npipeA was not created");  else  printf("\npipeA created");    f2 = mkfifo("pipeB",0666);    if(f2<0)  printf("\npipeB was not created");  else  printf("\npipeB is created\n");  } | //pipe\_creation.c  //leftTerminal.c  #include<stdio.h>  #include<fcntl.h>  #include<string.h>  #include<stdlib.h>  void main()  {  char str[256]="start";  int fifo\_write,fifo\_read,i;  while(strcmp(str,"end")!=0)  {    fifo\_write= open("pipeA",O\_WRONLY);  if(fifo\_write<0)  printf("\nError opening pipe");  else  {  printf("\nEnter text:\n");    i=0;  while (1)  {  scanf("%c",&str[i]);  if (str[i] == '\n')  break;  i++;  }  // scanf("%s",str);  write(fifo\_write,str,255\*sizeof(char));  close(fifo\_write);  }    fifo\_read=open("pipeB",O\_RDONLY);  if(fifo\_read<0)  printf("\nError opening write pipe");  else  {  read(fifo\_read,str,255\*sizeof(char));  close(fifo\_read);  printf("\n%s",str);    }  }  } | //rightTerminal.c  #include<stdio.h>  #include<fcntl.h>  #include<string.h>  #include<stdlib.h>  void main()  {  char str[256]="start";  int fifo\_read,fifo\_write,i;  while(strcmp(str,"end")!=0)  {  fifo\_read=open("pipeA",O\_RDONLY);  if(fifo\_read<0)  printf("\nError opening read pipe");  else    {  read(fifo\_read,str,255\*sizeof(char));  close(fifo\_read);  printf("\n%s",str);  }  fifo\_write=open("pipeB",O\_WRONLY);  if(fifo\_write<0)  printf("\nError opening write pipe");  else  {  printf("\nEnter text:\n");  i=0;  while (1)  {    scanf("%c",&str[i]);  if (str[i] == '\n')  break;  i++;  }  write(fifo\_write,str,255\*sizeof(char));  close(fifo\_write);  }  }  } |

Execl()

#include<stdio.h>

#include<unistd.h>

int main()

{

        printf("Before\n");

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

        printf("After\n");

}

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

int main()

{

        pid\_t q;

        q = fork();

        if (q ==0)

        { execl("/bin/ps","ps",NULL);

        printf("I am child and my pid is %d\n", getpid());

        sleep(10);}

else

        { wait(NULL);

        printf("I am parent having pid %d\n",getpid());

        }

}

##### **Program to create threads in linux**. Thread prints 0-4 while the main process prints 20-24

#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<pthread.h>  
void \*thread\_function(void \*arg);  
 int i,j;  
 int main() {  
 pthread\_t a\_thread; //thread declaration

pthread\_create(&a\_thread, NULL, thread\_function, NULL);   
//thread is created  
 pthread\_join(a\_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference  
 printf("Inside Main Program\n");  
 for(j=20;j<25;j++)  
 {  
 printf("%d\n",j);  
 sleep(1);  
 }  
 }

void \*thread\_function(void \*arg) {   
// the work to be done by the thread is defined in this function  
 printf("Inside Thread\n");  
 for(i=0;i<5;i++)  
 {  
 printf("%d\n",i);  
 sleep(1);  
 }  
 }

**Note**: To compile any program which involves creation of thread(s) use pthread library (lpthread)  
Suppose the above program is named “Thread.c”, then to compile write  
$gcc thread.c -lpthread  
To run the command remains same  
$./a.out

### Output

0

1

2

3

4

Inside Main Program

20

21

22

23

24

### Rerun the program by commenting pthread\_join(a\_thread, NULL);

And see the result

### How it works?

pthread\_create() creates a new thread which starts to execute thread\_function. This function creates a loop which prints 0-4. The sleep function makes the thread go to sleep after each digit is printed. pthread\_join() makes the main function wait until the newly created thread finishes its execution. So the control returns to the main function only when the thread finishes. Then the main function prints “Inside Main program” and executes the loop from 20-24.

## How a thread returns a value to the main process?

pthread\_exit() is used to return a value from the thread back to the main process. The program below shows this. The program also shows how to pass value to a thread from the main process.

**Homework:**

## How a thread returns a value to the main process?

pthread\_exit() is used to return a value from the thread back to the main process. The program below shows this. The program also shows how to pass value to a thread from the main process.

##### Program 2: Write a Program to create a thread. The thread prints numbers from zero to n, where value of n is passed from the main process to the thread. The main process also waits for the thread to finish first and then prints from 20-24.

#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<pthread.h>   
#include<string.h>  
void \*thread\_function(void \*arg);  
 int i,n,j;  
 int main() {  
 char \*m="5";  
 pthread\_t a\_thread; //thread declaration  
 void \*result;  
 pthread\_create(&a\_thread, NULL, thread\_function, m); //thread is created  
 pthread\_join(a\_thread, &result);   
 printf("Thread joined\n");  
 for(j=20;j<25;j++)  
 {  
 printf("%d\n",j);  
 sleep(1);  
 }  
 printf("thread returned %s\n",(char \*)result);  
 }  
 void \*thread\_function(void \*arg) {   
 int sum=0;  
 n=atoi(arg);

for(i=0;i<n;i++)  
 {  
 printf("%d\n",i);  
 sleep(1);  
 }  
 pthread\_exit("Done"); // Thread returns "Done"  
 }

## How to pass multiple values to a thread using structure?

##### Program 3: Program to create a thread. The thread is passed more than one input from the main process. For passing multiple inputs we need to create structure and include all the variables that are to be passed in this structure.

#include<stdio.h

#include<pthread.h>

struct arg\_struct { //structure which contains multiple variables that are to passed as input to the thread

int arg1;

int arg2;

};

void \*arguments(void \*arguments)

{

struct arg\_struct \*args=arguments;

printf("%d\n", args -> arg1);

printf("%d\n", args -> arg2);

pthread\_exit(NULL);

}

int main()  
 {  
 pthread\_t t;  
 struct arg\_struct args;  
 args.arg1 = 5;  
 args.arg2 = 7;  
 pthread\_create(&t, NULL, arguments, &args);   
//structure passed as 4th argument  
 pthread\_join(t, NULL); /\* Wait until thread is finished \*/  
}

Note: Just as above we can also pass an array to a thread.

**File Locking**

**This program will write lock a sample.txt file. As it is write lock, when try to execute the same program in another window, it wont lock. Only when the first program (first window) releases lock, the second window will apply lock**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

#include <unistd.h>

int main()

{

char buf[100];

struct flock fl;

fl.l\_type = F\_WRLCK;

fl.l\_whence = SEEK\_SET;

fl.l\_start = 0;

fl.l\_len= 0;

fl.l\_pid = getpid();

printf("P1 Applying\nwrite lock");

int fd = open("sample.txt", O\_WRONLY|O\_APPEND);

if (fd == -1)

{

perror("Unable to open file: ");

exit(EXIT\_FAILURE);

}

printf("Press Enter to try to get lock -");

getchar();

printf("Locking...\n");

if (fcntl(fd, F\_SETLKW, &fl) == -1)

{

perror("fcntl caused some error: ");

exit(EXIT\_FAILURE);

}

printf("Locked\n");

strcpy(buf,"we are appending file\n");

if (write(fd,buf,strlen(buf))<0)

{

perror("Problem in writing to file");

exit(1);

}

printf("Press Enter to release lock -");

getchar();

fl.l\_type = F\_UNLCK;

if (fcntl(fd, F\_SETLK, &fl) == -1)

{

perror("fcntl caused some error: ");

exit(EXIT\_FAILURE);

}

printf("Unlocked.\n");

close(fd);

return 0;

}

==========================================================

**This program will read lock a sample.txt file. As it is read lock, when try to execute the same program in another window, it will lock. This is because two reader shall operate on file simultaneously**

include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

#include <unistd.h>

int main()

{

char buf[100];

struct flock fl;

fl.l\_type = F\_RDLCK;

fl.l\_whence = SEEK\_SET;

fl.l\_start = 0;

fl.l\_len= 0;

fl.l\_pid = getpid();

printf("Applying\nread lock");

int fd = open("sample.txt", O\_RDONLY);

if (fd == -1)

{

perror("Unable to open file: ");

exit(EXIT\_FAILURE);

}

printf("Press Enter to try to get lock -");

getchar();

printf("Locking...\n");

if (fcntl(fd, F\_SETLKW, &fl) == -1)

{

perror("fcntl caused some error: ");

exit(EXIT\_FAILURE);

}

printf("Locked\n");

int flag=read(fd,buf,100);

if(flag<0){

printf("Error occured while reading\n");

exit(1);

}

printf("Read file content:\n");

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

printf("Press Enter to release lock -");

getchar();

fl.l\_type = F\_UNLCK;

if (fcntl(fd, F\_SETLK, &fl) == -1)

{

perror("fcntl caused some error: ");

exit(EXIT\_FAILURE);

}

printf("Unlocked.\n");

close(fd);

return 0;

}