Slip 1

1. Take multiple files as Command Line Arguments and print their inode numbers and file types

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{
char d[50];
if(argc==2)
{
bzero(d,sizeof(d));
strcat(d,"ls ");
strcat(d,"-i ");
strcat(d,argv[1]);
system(d);
}
else
printf("\nInvalid No. of inputs");
}
```

output:

```
student@ubuntu:~$ mkdir dd
student@ubuntu:~$ cd dd
student@ubuntu:~/dd$ cat >f1
hello
^z
student@ubuntu:~/dd$ cd
student@ubuntu:~$gcc –o flist.out flist.c
student@ubuntu:~$./flist.out dd
hello
46490 f1
```

2. Write a C program to send SIGALRM signal by child process to parent process and parent process make a provision to catch the signal and display alarm is fired.(Use Kill, fork, signal and sleep system call)

```
#include <fcntl.h>
#include <unistd.h>
```

```
#include <stdio.h>
#include<signal.h>
#include<sys/types.h>
#include<sys/wait.h>
#include <stdlib.h>
void Dingdong()
{
printf("Ding!");
exit(1);
}
int main(int argc, char *argv[])
{
if(argc!=3)
{
printf("How much seconds you want to sleep the child process\n");
}
int PauseSecond=(argv[1]);
{
if(fork()==0)
{
printf("waiting for alarm to go off\n");
printf("%d second pause",PauseSecond);
sleep(PauseSecond);
kill(getpid(),SIGALRM);
}
else {
printf("Alarm application starting\n", getpid());
signal(SIGALRM,Dingdong);
printf("done");
}
}
}
```

Slip 2

1. Write a C program to find file properties such as inode number, number of hard link, File

permissions, File size, File access and modification time and so on of a given file using stat() system call.

```
#include<stdio.h>
#include<unistd.h>
#include<dirent.h>
#include<string.h>
#include<time.h>
#include<stdlib.h>
#include<sys/stat.h>
#include<sys/types.h>

int main(int argc,char* argv[])
{
struct stat info;
if(argc!=2)
{
printf("Enter a filename");
scanf("%s",argv[1]);
}
if(stat(argv[1],&info)==-1)
{
printf("stat error/n");
exit(0);
}
printf("inode number=%d\n",info.st_ino);
printf("size = %d",(long)info.st_size);
printf("last file access = %s\n",ctime(&info.st_atime));
printf("notification time = %s\n",ctime(&info.st_mtime));
printf("No of Hardlink = %d\n",info.st_nlink);
printf("File Permissions : \n");
printf((info.st_mode && S_IRUSR)?"r":"-");
printf((info.st_mode && S_IWUSR)?"w":"-");
printf((info.st_mode && S_IXUSR)?"x":"-");

return 0;

}
```

2. Write a C program that catches the ctrl-c (SIGINT) signal for the first time and display the appropriate message and exits on pressing ctrl-c again.

```
#include <stdio.h>
#include <unistd.h>
```

```
#include <stdlib.h>
#include <signal.h>
void sigfun(int sig)
{
printf("You have presses Ctrl-C , please press again to exit");
(void) signal(SIGINT, SIG_DFL);
}
int main()
{
(void) signal(SIGINT, sigfun);
while(1) {
printf("Hello World!
");
sleep(1);
}
return(0);
}
```

Slip 3

Print the type of file and inode number where file name accepted through Command Line

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{
char d[50];
if(argc==2)
{
bzero(d,sizeof(d));
strcat(d,"ls ");
strcat(d,"-i ");
strcat(d,argv[1]);
system(d);
}
else
printf("\nInvalid No. of inputs");
}
```

output:

```
student@ubuntu:~$ mkdir dd
```

```
student@ubuntu:~$ cd dd
student@ubuntu:~/dd$ cat >f1
hello
^z
student@ubuntu:~/dd$ cd
student@ubuntu:~$gcc –o flist.out flist.c
student@ubuntu:~$./flist.out dd
hello
46490 f1
```

Write a C program which creates a child process to run linux/ unix command or any user defined program. The parent process set the signal handler for death of child signal and Alarm signal. If a child process does not complete its execution in 5 second then parent process kills child process.

```
// C program to implement sighup(), sigint()
// and sigquit() signal functions
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

// function declaration
void sighup();
void sigint();
void sigquit();

// driver code
void main()
{
    int pid;

    /* get child process */
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) { /* child */
        signal(SIGHUP, sighup);
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        for (;;)
            ; /* loop for ever */
```

```
    }

    else /* parent */
    { /* pid hold id of child */
        printf("\nPARENT: sending SIGHUP\n\n");
        kill(pid, SIGHUP);

        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGINT\n\n");
        kill(pid, SIGINT);

        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGQUIT\n\n");
        kill(pid, SIGQUIT);
        sleep(3);
    }
}

// sighup() function definition
void sighup()

{
    signal(SIGHUP, sighup); /* reset signal */
    printf("CHILD: I have received a SIGHUP\n");
}

// sigint() function definition
void sigint()

{
    signal(SIGINT, sigint); /* reset signal */
    printf("CHILD: I have received a SIGINT\n");
}

// sigquit() function definition
void sigquit()
{
    printf("My DADDY has Killed me!!!\n");
    exit(0);
}
```

Write a C program to find whether a given files passed through command line arguments are present in current directory or not.

```
#include<stdio.h>
#include<unistd.h>

int main(int argc,char *argv[])
{
if(access(argv[1],F_OK)==0)
printf("File %s exists.",argv[1]);
else
printf("File not exists.");
return 0;
}
```

Write a C program which creates a child process and child process catches a signal SIGHUP, SIGINT and SIGQUIT. The Parent process send a SIGHUP or SIGINT signal after every 3 seconds, at the end of 15 second parent send SIGQUIT signal to child and child terminates by displaying message "My Papa has Killed me!!!".

```
// C program to implement sighup(), sigint()
// and sigquit() signal functions
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

// function declaration
void sighup();
void sigint();
void sigquit();

// driver code
void main()
{
        int pid;

        /* get child process */
        if ((pid = fork()) < 0) {
```

```
                perror("fork");
                exit(1);
        }

        if (pid == 0) { /* child */
                signal(SIGHUP, sighup);
                signal(SIGINT, sigint);
                signal(SIGQUIT, sigquit);
                for (;;)
                        ; /* loop for ever */
        }

        else /* parent */
        { /* pid hold id of child */
                printf("\nPARENT: sending SIGHUP\n\n");
                kill(pid, SIGHUP);

                sleep(3); /* pause for 3 secs */
                printf("\nPARENT: sending SIGINT\n\n");
                kill(pid, SIGINT);

                sleep(3); /* pause for 3 secs */
                printf("\nPARENT: sending SIGQUIT\n\n");
                kill(pid, SIGQUIT);
                sleep(3);
        }
}

// sighup() function definition
void sighup()

{
        signal(SIGHUP, sighup); /* reset signal */
        printf("CHILD: I have received a SIGHUP\n");
}

// sigint() function definition
void sigint()

{
```

```
        signal(SIGINT, sigint); /* reset signal */
        printf("CHILD: I have received a SIGINT\n");
}

// sigquit() function definition
void sigquit()
{
        printf("My DADDY has Killed me!!!\n");
        exit(0);
}
```

5.

Count specific file types

If you want to count only files of a specific type, you can add a pattern to the `ls` command. For example, to count only `.txt` files, run the following command:

```
ls *.txt | wc -l
```

This command will list all `.txt` files in the current directory, and `wc -l` will count the number of lines in the output.

In our case, there are five files with the `.txt` extension in the `demo` directory. So, the output we get is 5, as shown below:

As we've seen, the combination of `ls` and `wc` commands lacks flexibility. It works best when used to count the number of files inside a directory that doesn't contain any subdirectories. If there are subdirectories present, the command will count each subdirectory as a single item, adding it to the total file count, which will result in an incorrect answer. This is because we're interested in counting the number of files, not directories.

Count the Number of Files Using find and wc Commands in Linux

In the previous section, we discovered how the combination of `ls` and `wc` commands can be effective for counting files under specific conditions. Now, we're going to explore the use of `find` and `wc` commands together, which provide a greater degree of flexibility in comparison.

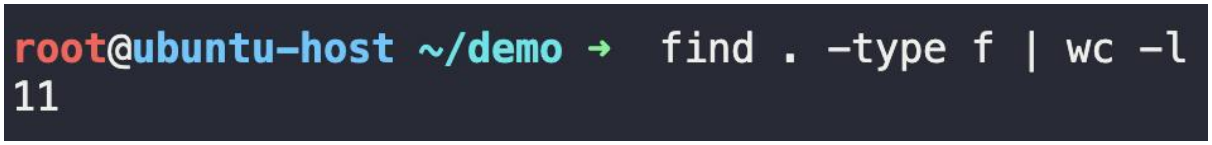
Count all files in a directory, including subdirectories

To count all files, both visible and hidden, in a directory and its subdirectories, you can use the `find` command with the `-type f` option. This tells the `find` command to count all types of files but not directories. Then you pipe ("|") this output to the `wc -l` command, which counts the number of lines in the output. Each line corresponds to a file, giving you the total number of files.

Run the following command:

```
find . -type f | wc -l
```

You'll see an output that represents the total number of files (both visible and hidden) in the `demo` directory and its subdirectory:



But what if you only want to count visible files, excluding the hidden ones? The `find` command doesn't have a direct option to exclude hidden files. However, there's a workaround: you can exclude hidden files by using a pattern. Run the following command:

```
find . -type f ! -name ".*" | wc -l
```

In this command, we're adding the `! -name ".*"` segment. What this does is it tells the `find` command to exclude any files that start with a dot (the hidden files).

When you run this command, the output will represent the total count of visible (non-hidden) files in the `demo` directory and the `subdemo` subdirectory, as shown below:

```
root@ubuntu-host ~/demo ➜  find . -type f ! -name ".*" | wc -l
8
```

Count all files in a directory excluding subdirectories

If you want to count the files in a directory but ignore any files in its subdirectories, you can do so by using the `-maxdepth` option with the `find` command. This option limits the depth of directories `find` searches. When `-maxdepth` is set to 1, `find` only looks at the current directory level.

To find the total count of both visible and hidden files in the `demo` directory, run the following command:

```
find  -maxdepth 1 -type f | wc -l
```

After you run the above command, you'll get the following output:

```
root@ubuntu-host ~/demo ➜  find  -maxdepth 1 -type f | wc -l
6
```

Now, if you want to exclude hidden files from this count, you can add the `! -name ".*"` segment to your command, like so:

```
find . -maxdepth 1 -type f ! -name ".*" | wc -l
```

After running the command, you'll get the count of the visible files in the `demo` directory, excluding both hidden files and files in subdirectories, as shown below:

```
root@ubuntu-host ~/demo ➜  find . -maxdepth 1 -type f ! -name ".*" | wc -l
5
```

Count specific file types

Sometimes, you might want to count only files of a specific type. In such cases, you can use the `-name` flag followed by the file extension. For instance, if you want to count only `.txt` files, run this command:

```
find . -type f -name "*.txt" | wc -l
```

The output, as shown below, is the total count of `.txt` files in the current directory and all its subdirectories.

```
root@ubuntu-host ~/demo →  find . -type f -name "*.txt" | wc -l
5
```

Again, if you want to limit your search to the current directory, you can use the `-maxdepth` flag. For instance, to find only `.txt` files in the current directory, modify and run the command as follows:

```
find . -maxdepth 1 -type f -name "*.txt" | wc -l
```

Once you run the above command, you'll get the following output, which is the total number of `.txt` files in the current directory *only*, excluding subdirectories.

```
root@ubuntu-host ~/demo →  find . -maxdepth 1 -type f -name "*.txt" | wc -
l
5
```

Count the Number of Files Using the tree Command in Linux

The `tree` command in Linux provides an easy way to visualize the hierarchy of your directories and files, resembling a tree structure – hence the name. It's particularly useful when dealing with many nested subdirectories, as it provides a clear layout of your files. Moreover, `tree` also presents a summary at the end that counts the number of directories and files.

Please note that the `tree` command might not be pre-installed on all Linux distributions. To check whether `tree` is available on your system, run the following command:

```
tree --version
```

```
root@ubuntu-host ~/demo →  tree --version
-bash: tree: command not found
```

Write a C program to create an unnamed pipe. The child process will write following three messages to pipe and parent process display it. Message1 = "Hello World" Message2 = "Hello SPPU" Message3 = "Linux is Funny"

```
#include<stdio.h>
#include<unistd.h>
int main() {
int pipefds[2];
int returnstatus;
char writemessages[3][20]={"Hello World", "Hello SPPU","Linux is Funny"};
char readmessage[20];
returnstatus = pipe(pipefds);
if (returnstatus == -1) {
printf("Unable to create pipe\n");
return 1;
}
int child = fork();
if(child==0){
printf("Child is Writing to pipe - Message 1 is %s\n", writemessages[0]);
write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
printf("Child is Writing to pipe - Message 2 is %s\n", writemessages[1]);
write(pipefds[1], writemessages[1], sizeof(writemessages[1]));
printf("Child is Writing to pipe - Message 3 is %s\n", writemessages[2]);
write(pipefds[1], writemessages[2], sizeof(writemessages[2]));
}
else
{
read(pipefds[0], readmessage, sizeof(readmessage));
printf("Parent Process is Reading from pipe – Message 1 is %s\n",
readmessage);
read(pipefds[0], readmessage, sizeof(readmessage));
printf("Parent Process is Reading from pipe – Message 2 is %s\n",
readmessage);
read(pipefds[0], readmessage, sizeof(readmessage));
printf("Parent Process is Reading from pipe – Message 3 is %s\n",
readmessage);
}
return 0;
}
```

6

Display all the files from current directory which are created in particular month

```
#include<stdio.h>
#include<dirent.h>
#include<string.h>
#include<sys/stat.h>
#include<time.h>
#include<stdlib.h>
int main(intargc, char *argv[])
{
char in[100],st[100],*ch,*ch1,c,buff[512];
DIR *dp;
int i;
structdirent *ep;
struct stat sb;
charmon[100];
dp=opendir("./");
if (dp != NULL)
{
while(ep =readdir(dp))
{
if(stat(ep->d_name,&sb) == -1)
{
perror("stat");
exit(EXIT_SUCCESS);
}
strcpy(mon,ctime(&sb.st_ctime));
ch=strtok(mon," ");
ch=strtok(NULL,",");
ch1=strtok(ch," ");
if((strcmp(ch1,argv[1]))==0)
{
printf("%s\t\t%s",ep->d_name,ctime(&sb.st_ctime));
}
}
(void)closedir(dp);
}
return 0;
}
```

Write a C program to create n child processes. When all n child processes terminates, Display total cumulative time children spent in user and kernel mode

```
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<time.h>
#include<sys/times.h>
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
int i, status;
 pid_t pid;
 time_t currentTime;
struct tms cpuTime;
if((pid = fork())==-1) //start child process
 {
 perror("\nfork error");
 exit(EXIT_FAILURE);
 }
else if(pid==0) //child process
 {
 time(&currentTime);
 printf("\nChild process started at %s",ctime(&currentTime));
 for(i=0;i<5;i++)
 {
 printf("\nCounting= %dn",i); //count for 5 seconds
 sleep(1);
 }
 time(&currentTime);
 printf("\nChild process ended at %s",ctime(&currentTime));
 exit(EXIT_SUCCESS);
 }
else
 { //Parent process
 time(&currentTime); // gives normal time
 printf("\nParent process started at %s ",ctime(&currentTime));
 if(wait(&status)== -1) //wait for child process
```

```
 perror("\n wait error");
 if(WIFEXITED(status))
 printf("\nChild process ended normally");
 else
 printf("\nChild process did not end normally");
 if(times(&cpuTime)<0) //Get process time
 perror("\nTimes error");
 else
 { // _SC_CLK_TCK: system configuration time: seconds clock tick
 printf("\nParent process user time= %fn",((double)
cpuTime.tms_utime));
 printf("\nParent process system time = %fn",((double)
cpuTime.tms_stime));
printf("\nChild process user time = %fn",((double)
cpuTime.tms_cutime));
 printf("\nChild process system time = %fn",((double)
cpuTime.tms_cstime));
 }
 time(&currentTime);
 printf("\nParent process ended at %s",ctime(&currentTime));
 exit(EXIT_SUCCESS);
 }
}
```

7.

Write a C Program that demonstrates redirection of standard output to a file

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{
char d[50];
if(argc==2)
{
bzero(d,sizeof(d));
strcat(d,"ls ");
strcat(d,"> ");
strcat(d,argv[1]);
system(d);
}
```

```
else
printf("\nInvalid No. of inputs");
}
```
output:

```
student@ubuntu:~$ gcc –o std.out std.c
student@ubuntu:~$ls
downloads   documents   listing.c   listing.out   std.c   std.out
student@ubuntu:~$ cat > f1
^z
student@ubuntu:~$./std.out f1
student@ubuntu:~$cat f1
downloads
documents
listing.c
listing.out
std.c
std.out
```

Implement the following unix/linux command (use fork, pipe and exec system call) ls –l | wc –l

```
// C code to implement ls | wc command

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include<errno.h>

#include<sys/wait.h>

#include <unistd.h>

int main(){
```

```
// array of 2 size a[0] is for
// reading and a[1] is for
// writing over a pipe
int a[2];

// using pipe for inter
// process communication
pipe(a);

if(!fork())
{
    // closing normal stdout
    close(1);

    // making stdout same as a[1]
    dup(a[1]);

    // closing reading part of pipe
    // we don't need of it at this time
```

```
    close(a[0]);


    // executing ls
    execlp("ls","ls",NULL);
}
else
{
    // closing normal stdin
    close(0);


    // making stdin same as a[0]
    dup(a[0]);


    // closing writing part in parent,
    // we don't need of it at this time
    close(a[1]);


    // executing wc
    execlp("wc","wc",NULL);
```

```
    }

}
```

8

Write a C program that redirects standard output to a file output.txt. (use of dup and open system call).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
int main(void) {
int number1, number2, sum;
int input_fds = open("./input.txt", O_RDONLY);
if(dup2(input_fds, STDIN_FILENO) < 0) {
printf("Unable to duplicate file descriptor.");
exit(EXIT_FAILURE);
}
scanf("%d %d", &number1, &number2);
sum = number1 + number2;
printf("%d + %d = %d\n", number1, number2, sum);
return EXIT_SUCCESS;
}
```

Same as 7

9.

Generate parent process to write unnamed pipe and will read from it

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int pipe_fd[2]; // File descriptors for the pipe
```

```
// Create the pipe
if (pipe(pipe_fd) == -1) {
    perror("Pipe creation failed");
    exit(EXIT_FAILURE);
}

pid_t child_pid = fork(); // Fork a child process

if (child_pid == -1) {
    perror("Fork failed");
    exit(EXIT_FAILURE);
}

if (child_pid == 0) {
    // Child process

    close(pipe_fd[1]); // Close the write end of the pipe in the child process

    char buffer[100];
    ssize_t bytes_read = read(pipe_fd[0], buffer, sizeof(buffer));
    close(pipe_fd[0]); // Close the read end of the pipe in the child process

    if (bytes_read == -1) {
        perror("Read from pipe failed");
        exit(EXIT_FAILURE);
    }

    printf("Child received: %.*s\n", (int)bytes_read, buffer);

} else {
    // Parent process

    close(pipe_fd[0]); // Close the read end of the pipe in the parent process

    const char *message = "Hello from parent!";
    ssize_t bytes_written = write(pipe_fd[1], message, strlen(message));
    close(pipe_fd[1]); // Close the write end of the pipe in the parent process

    if (bytes_written == -1) {
```

```
            perror("Write to pipe failed");
            exit(EXIT_FAILURE);
        }

        printf("Parent sent: %s\n", message);
    }

    return 0;
}
```

Write a C program to Identify the type (Directory, character device, Block device, Regular file, FIFO or pipe, symbolic link or socket) of given file using stat() system call.

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

void identifyFileType(const char *path) {
    struct stat fileStat;

    // Use stat() to get information about the file
    if (stat(path, &fileStat) == -1) {
        perror("Error in stat");
        return;
    }

    // Check the file type using st_mode field in the stat structure
    if (S_ISREG(fileStat.st_mode)) {
        printf("%s is a regular file.\n", path);
    } else if (S_ISDIR(fileStat.st_mode)) {
        printf("%s is a directory.\n", path);
    } else if (S_ISCHR(fileStat.st_mode)) {
        printf("%s is a character device.\n", path);
    } else if (S_ISBLK(fileStat.st_mode)) {
        printf("%s is a block device.\n", path);
    } else if (S_ISFIFO(fileStat.st_mode)) {
```

```
        printf("%s is a FIFO or pipe.\n", path);
    } else if (S_ISLNK(fileStat.st_mode)) {
        printf("%s is a symbolic link.\n", path);
    } else if (S_ISSOCK(fileStat.st_mode)) {
        printf("%s is a socket.\n", path);
    } else {
        printf("%s is of unknown type.\n", path);
    }
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    const char *filename = argv[1];

    identifyFileType(filename);

    return 0;
}
```

10.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    int pipe_fd[2]; // File descriptors for the pipe

    // Create the pipe
    if (pipe(pipe_fd) == -1) {
        perror("Pipe creation failed");
        exit(EXIT_FAILURE);
    }
```

```
pid_t child_pid1 = fork(); // Fork the first child process

if (child_pid1 == -1) {
    perror("Fork failed");
    exit(EXIT_FAILURE);
}

if (child_pid1 == 0) {
    // First child process (write to the pipe)

    close(pipe_fd[0]); // Close the read end of the pipe in the first child process
    dup2(pipe_fd[1], STDOUT_FILENO); // Redirect stdout to the pipe

    // Execute the first command (e.g., "ls")
    execlp("ls", "ls", NULL);

    perror("Exec failed"); // This line will be reached only if execlp fails
    exit(EXIT_FAILURE);
}

pid_t child_pid2 = fork(); // Fork the second child process

if (child_pid2 == -1) {
    perror("Fork failed");
    exit(EXIT_FAILURE);
}

if (child_pid2 == 0) {
    // Second child process (read from the pipe)

    close(pipe_fd[1]); // Close the write end of the pipe in the second child process
    dup2(pipe_fd[0], STDIN_FILENO); // Redirect stdin to the pipe

    // Execute the second command (e.g., "wc -l")
    execlp("wc", "wc", "-l", NULL);

    perror("Exec failed"); // This line will be reached only if execlp fails
    exit(EXIT_FAILURE);
}
```

```
    // Parent process

    close(pipe_fd[0]); // Close the read end of the pipe in the parent process
    close(pipe_fd[1]); // Close the write end of the pipe in the parent process

    // Wait for both child processes to finish
    waitpid(child_pid1, NULL, 0);
    waitpid(child_pid2, NULL, 0);

    return 0;
}
```

Generate parent process to write unnamed pipe and will write into it. Also generate child process which will read from pipe

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int pipe_fd[2]; // File descriptors for the pipe

    // Create the pipe
    if (pipe(pipe_fd) == -1) {
        perror("Pipe creation failed");
        exit(EXIT_FAILURE);
    }

    pid_t child_pid = fork(); // Fork a child process

    if (child_pid == -1) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }

    if (child_pid == 0) {
        // Child process
```

```
        close(pipe_fd[1]); // Close the write end of the pipe in the child process

        char buffer[100];
        ssize_t bytes_read = read(pipe_fd[0], buffer, sizeof(buffer));
        close(pipe_fd[0]); // Close the read end of the pipe in the child process

        if (bytes_read == -1) {
            perror("Read from pipe failed");
            exit(EXIT_FAILURE);
        }

        printf("Child received: %.*s\n", (int)bytes_read, buffer);

    } else {
        // Parent process

        close(pipe_fd[0]); // Close the read end of the pipe in the parent process

        const char *message = "Hello from parent!";
        ssize_t bytes_written = write(pipe_fd[1], message, strlen(message));
        close(pipe_fd[1]); // Close the write end of the pipe in the parent process

        if (bytes_written == -1) {
            perror("Write to pipe failed");
            exit(EXIT_FAILURE);
        }

        printf("Parent sent: %s\n", message);
    }

    return 0;
}
```

11

Write a C program to get and set the resource limits such as files, memory associated with a process

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

void printLimits() {
    struct rlimit limits;

    // Get the current resource limits for file descriptors
    if (getrlimit(RLIMIT_NOFILE, &limits) == -1) {
        perror("getrlimit for RLIMIT_NOFILE failed");
        exit(EXIT_FAILURE);
    }

    printf("Current maximum file descriptors limit: %ld\n", (long)limits.rlim_cur);

    // Get the current resource limits for stack size
    if (getrlimit(RLIMIT_STACK, &limits) == -1) {
        perror("getrlimit for RLIMIT_STACK failed");
        exit(EXIT_FAILURE);
    }

    printf("Current maximum stack size limit: %ld bytes\n", (long)limits.rlim_cur);
}

void setLimits() {
    struct rlimit new_limits;

    // Set the new maximum file descriptors limit
    new_limits.rlim_cur = 1024;  // Set to the desired value
    new_limits.rlim_max = 1024;  // Set to the same value as rlim_cur for simplicity

    if (setrlimit(RLIMIT_NOFILE, &new_limits) == -1) {
        perror("setrlimit for RLIMIT_NOFILE failed");
        exit(EXIT_FAILURE);
    }

    printf("New maximum file descriptors limit set: %ld\n", (long)new_limits.rlim_cur);

    // Set the new maximum stack size limit
    new_limits.rlim_cur = 1024 * 1024 * 2;  // Set to the desired value (2 MB)
```

```
    new_limits.rlim_max = 1024 * 1024 * 2;  // Set to the same value as rlim_cur for simplicity

    if (setrlimit(RLIMIT_STACK, &new_limits) == -1) {
        perror("setrlimit for RLIMIT_STACK failed");
        exit(EXIT_FAILURE);
    }

    printf("New maximum stack size limit set: %ld bytes\n", (long)new_limits.rlim_cur);
}

int main() {
    printf("Before setting limits:\n");
    printLimits();

    printf("\nSetting new limits:\n");
    setLimits();

    printf("\nAfter setting limits:\n");
    printLimits();

    return 0;
}
```

Write a C program that redirects standard output to a file output.txt. (use of dup and open system call).

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

int main() {
    // Open the file for writing (create if it doesn't exist, truncate to zero if it does)
    int fileDescriptor = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);

    if (fileDescriptor == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }
```

```
    // Duplicate file descriptor to standard output (file descriptor 1)
    if (dup2(fileDescriptor, STDOUT_FILENO) == -1) {
        perror("Error duplicating file descriptor");
        close(fileDescriptor);
        exit(EXIT_FAILURE);
    }

    // Close the original file descriptor (since it's duplicated to standard output)
    close(fileDescriptor);

    // Now, anything written to standard output will go to "output.txt"
    printf("This will be written to output.txt.\n");

    // Restore standard output to the console (file descriptor 1)
    if (dup2(STDOUT_FILENO, 1) == -1) {
        perror("Error restoring standard output");
        exit(EXIT_FAILURE);
    }

    // The following will be printed to the console, not "output.txt"
    printf("This will be printed to the console.\n");

    return 0;
}
```

12.
Write a C program that print the exit status of a terminated child process

```
/ C code to find the exit status of child process
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

// Driver code
int main(void)
{
    pid_t pid = fork();
```

```
    if ( pid == 0 )
    {
      /* The pathname of the file passed to execl()
         is not defined   */
      execl("/bin/sh", "bin/sh", "-c", "./nopath", "NULL");
    }

    int status;

    waitpid(pid, &status, 0);

    if ( WIFEXITED(status) )
    {
       int exit_status = WEXITSTATUS(status);
       printf("Exit status of the child was %d\n",
                                exit_status);
    }
    return 0;
}
```

Write a C program which receives file names as command line arguments and display those filenames in ascending order according to their sizes. I) (e.g $ a.out a.txt b.txt c.txt, …)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

// Structure to hold file information
struct FileInfo {
    char *name;
    off_t size;
};

// Function to compare two FileInfo structures by size
int compareFileInfo(const void *a, const void *b) {
    return ((struct FileInfo *)a)->size - ((struct FileInfo *)b)->size;
}

int main(int argc, char *argv[]) {
    // Check if at least one file name is provided
```

```
    if (argc < 2) {
        fprintf(stderr, "Usage: %s file1 file2 file3 ...\n", argv[0]);
        return EXIT_FAILURE;
    }

    // Allocate an array of FileInfo structures
    struct FileInfo *files = (struct FileInfo *)malloc((argc - 1) * sizeof(struct FileInfo));

    // Populate the array with file names and sizes
    for (int i = 1; i < argc; ++i) {
        files[i - 1].name = argv[i];

        struct stat fileStat;
        if (stat(argv[i], &fileStat) == -1) {
            perror("Error getting file size");
            return EXIT_FAILURE;
        }

        files[i - 1].size = fileStat.st_size;
    }

    // Sort the array of FileInfo structures based on file size
    qsort(files, argc - 1, sizeof(struct FileInfo), compareFileInfo);

    // Display the sorted file names
    printf("File names in ascending order based on size:\n");
    for (int i = 0; i < argc - 1; ++i) {
        printf("%s - %ld bytes\n", files[i].name, (long)files[i].size);
    }

    // Free the allocated memory
    free(files);

    return EXIT_SUCCESS;
}
```

13 . Write a C program that illustrates suspending and resuming processes using signals

```
#include <stdio.h>
#include <ospace/unix.h>
```

```
int child_function()
{
  while (true) // Loop forever.
    {
    Printf("Child loop\n");
    os_this_process::sleep( 1 );
    }
  return 0; // Will never execute.
}
int main()
{
  os_unix_toolkit initialize;
  os_process child ( child function ); // Spawn child.
  os_this_process::sleep( 4 );
  printf("child.suspend()\n");
  child.suspend();
  printf("Parent sleeps for 4 seconds\n");
  os_this_process::sleep (4);
  printf("child.resume()");
  child.resume ();
  os_this_process::sleep (4);
  printf("child.terminate()");
  child.terminate ();
  printf("Parent finished");
  return 0;
}
```

Output:

```
Child loop
Child loop
Child loop
Child loop
Child loop
child.suspend()
Parent sleeps for 4 seconds
child.resume()
Child loop
Child loop
Child loop
Child loop
child.terminate()
Child loop
Parent finished
```

Write a C program that a string as an argument and return all the files that begins with that name in the current directory. For example > ./a.out foo will return all file names that begins with foo

```
#include<stdio.h>
#include<dirent.h>

int main(void)
{
   DIR *d;
   struct dirent *dir;
   d = opendir(".");
   if (d)
   {
      while ((dir = readdir(d)) != NULL)
      {
         printf("%s\n", dir->d_name);
      }
      closedir(d);
   }
   return(0);
}
```

14

Display all the files from current directory whose size is greater that n Bytes Where n is accept from user.

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>

void listFilesWithSize(char *path, off_t sizeThreshold) {
   DIR *dir;
   struct dirent *entry;
   struct stat fileStat;

   // Open the directory
```

```
    dir = opendir(path);

    if (dir == NULL) {
        perror("Error opening directory");
        exit(EXIT_FAILURE);
    }

    printf("Files larger than %ld Bytes in %s:\n", (long)sizeThreshold, path);

    // Read directory entries
    while ((entry = readdir(dir)) != NULL) {
        // Ignore "." and ".."
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }

        // Construct the full path to the file
        char filePath[256];
        snprintf(filePath, sizeof(filePath), "%s/%s", path, entry->d_name);

        // Get file information
        if (stat(filePath, &fileStat) == -1) {
            perror("Error getting file information");
            exit(EXIT_FAILURE);
        }

        // Check if the file size is greater than the threshold
        if (S_ISREG(fileStat.st_mode) && fileStat.st_size > sizeThreshold) {
            printf("%s - %ld Bytes\n", entry->d_name, (long)fileStat.st_size);
        }
    }

    // Close the directory
    closedir(dir);
}


int main() {
    char path[256];
    off_t sizeThreshold;
```

```
    // Get the directory path from the user
    printf("Enter the directory path: ");
    scanf("%s", path);

    // Get the size threshold from the user
    printf("Enter the size threshold in Bytes: ");
    scanf("%ld", &sizeThreshold);

    // List files in the specified directory with size greater than the threshold
    listFilesWithSize(path, sizeThreshold);

    return 0;
}
```

Write a C program to find file properties such as inode number, number of hard link, File permissions, File size, File access and modification time and so on of a given file using stat() system call.

```
#include<stdio.h>
#include<unistd.h>
#include<dirent.h>
#include<string.h>
#include<time.h>
#include<stdlib.h>
#include<sys/stat.h>
#include<sys/types.h>

int main(int argc,char* argv[])
{
struct stat info;
if(argc!=2)
{
printf("Enter a filename");
scanf("%s",argv[1]);
}
if(stat(argv[1],&info)==-1)
{
printf("stat error/n");
exit(0);
}
```

```
printf("inode number=%d\n",info.st_ino);
printf("size = %d",(long)info.st_size);
printf("last file access = %s\n",ctime(&info.st_atime));
printf("notification time = %s\n",ctime(&info.st_mtime));
printf("No of Hardlink = %d\n",info.st_nlink);
printf("File Permissions : \n");
printf((info.st_mode && S_IRUSR)?"r":"-");
printf((info.st_mode && S_IWUSR)?"w":"-");
printf((info.st_mode && S_IXUSR)?"x":"-");

return 0;

}
```

15

Same as 14

Same as 3