1. Write a C/C++ program to implement UNIX commands ln, mv, rm commands using APIs.

Program 1:

```
#include <iostream>
#include <stdio.h>
#include <unistd.h>
using namespace std;
int main ( int argc, char* argv[] ) {
     if (argc != 3) {
     cout<<"usage: ./a.out Source_FileName Link_FileName\n";
     exit(0);
     }
     if (( link ( argv[1], argv[2 ] )) == -1 ) perror( "link error"
);
     else cout<<"Link has been created with the name"<<argv[2];
     return 0;
}
```

Description:

Creates a hard link to an existing file.
Takes two command-line arguments:
    The source file name
The link file name (the new link to be created)

Sample Output:

```
$ ./a.out source_file link_file
Link has been created with the name link_file
```

Program 2:

```
#include <iostream>
#include <stdio.h>
#include <unistd.h>
using namespace std;
int main(int argc, char *argv[])
{
      int i;
      if (argc <=1) {
      cout<<"No files to delete"
            <<"usage: ./a.out FileName1 FileName2 …..\n";
      exit(1);
      }
      for(i=1;i<argc;i++) {
      if(unlink(argv[i])==-1) perror("Error in deleting...\n");
      else cout<<"the file ”<<argv[i]<<” is deleted...\n";
      }
      return 0;
}
```

Description:

Deletes multiple files specified as command-line arguments.
Takes one or more file names as command-line arguments.

Sample Output:

```
$ ./a.out file1 file2 file3
the file file1 is deleted...
the file file2 is deleted...
Error in deleting...: No such file or directory
```

Program 3:

```
#include <stdio.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
     if (argc != 3) {
     cout<<"usage: ./a.out Source_FileName Link_FileName\n";
     exit(1);
     }
     if((link(argv[1],argv[2]))==-1) perror("Link eror\n");
     else unlink(argv[1]); //deletes first file
     return 0;
}
```

Description:

Creates a hard link to an existing file, then deletes the original file.
Takes two command-line arguments:
    The source file name
    The link file name

Sample Output:

```
$ ./a.out source_file link_file
(No output displayed, as the original file is deleted after
successful link creation)
```

2. Write a program in C/C++ to display the contents of a named file on standard output device., Also Write a program to copy the contents of one file to another.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
      int n,fd1, fd2;
      char buf[10];
      if (argc != 3) {
      cout<<"usage: ./a.out Source_FileName Destination_FileName\n";
      exit(1);
      }
      if((fd1= open(argv[1], O_RDONLY))==-1) {
      cout<<"Can't open file" <<argv[1]<<" for Reading\n";
      exit(0);
      }
      if((fd2= open(argv[2],O_WRONLY | O_CREAT | O_TRUNC ,744))==-1)
{
      cout<<"Can't open file "<<argv[2]<<" for Writing\n";
      exit(0);
      }
      while((n = read(fd1, buf, 1))>0) {
      write(1, buf,n); // write to terminal
      write(fd2, buf,n); // write to file
      }
      close (fd1);
      close (fd2);
      return 0;
}
```

Description:

- Copies the contents of a file to another file and prints the contents to the terminal.
- Takes two command-line arguments:
    - The source file name
    - The destination file name

Sample Output:

Hello, world! (given that this was the contents of the file we read)

3. Write a C program that reads every 100th byte from the file, where the file name is given as command -line argument

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <iostream>
using namespace std;

int main(int argc,char *argv[]) {
  int fd,n,skval;
  char c;
  if(argc!=2) {
      cout<<"usage: ./a.out FileName\n";
      exit(1);
  }
  if((fd= open(argv[1], O_RDONLY))==-1) {
      cout<<"Can't open file "<<argv[1]<<" for Reading\n";
      exit(0);
  }
  while((n=read(fd,&c,1))==1) {
      cout<<"\nChar: "<<c;
      skval=lseek(fd,99,1);
      cout<<"\nNew seek value is:"<<skval;
  }
  cout<<endl;
  exit(0);
}
```

Explanation:

1. Checks for correct arguments (filename).
2. Open the file in read-only mode.
3. Loops:
    - Reads 1 byte and prints it.
    - Moves the file pointer 99 bytes forward using `lseek`.
    - Prints the new pointer position.
4. Ends loop when reaching file end.
5. Closes the file and exits.

Sample Output:

```
Char: #
New seek value is: 99
Char: i
New seek value is: 199
Char: n
New seek value is: 299
Char: c
New seek value is: 399
Char: l
New seek value is: 499
Char: u
New seek value is: 599
Char: d
New seek value is: 699
Char: e
New seek value is: 799
```

4. Write a C program to display information of a given file which determines type of file and Inode information, where the file name is given as a command line argument.

```
#include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<time.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
      struct stat sb;
      if (argc != 2) {
      cout<<"Usage: ./a.out FileName \n";
      exit(0);
      }
      if (stat(argv[1], &sb) == -1) {
      perror("stat");
      exit(0);
      }
      cout<<"File: "<<argv[1]<<" is ";
      switch (sb.st_mode& S_IFMT) {
      case S_IFBLK: cout<<"Block device file \n";
                    break;
      case S_IFCHR: cout<<"Character device file\n";
                    break;
      case S_IFDIR: cout<<"Directory file\n";
                    break;
      case S_IFIFO: cout<<"FIFO/pipe file\n";
                    break;
      case S_IFLNK: cout<<"Symbolic link file \n";
                    break;
      case S_IFREG: cout<<"Regular file \n";
                    break;
      default: cout<<"Unknown file? \n";
      }
      cout<<"and its Inode number is : "<< sb.st_ino<<endl;
      return 0;
}
```

Description:

- This program takes a single file name as a command-line argument.
- It uses the `stat` function to obtain information about the file.
- Based on the information obtained, it determines the file type (regular file, directory, symbolic link, etc.) and prints it to the console.
- It also prints the file's inode number.

Steps:

1. Check if a single command-line argument is provided. Exit with an error message if not.
2. Use `stat` function to get information about the file specified by the argument.
3. Analyse the `st_mode` field within the `stat` structure to determine the file type:
    - `S_IFBLK`: Block device file
    - `S_IFCHR`: Character device file
    - `S_IFDIR`: Directory file
    - `S_IFIFO`: FIFO/pipe file
    - `S_IFLNK`: Symbolic link file
    - `S_IFREG`: Regular file
    - `default`: Unknown file type
4. Print the file type and its inode number.

Example Output:

```
File: myfile.txt is Regular file
and its Inode number is: 1283456
```

5. Write a C program to create a process by using fork () and vfork() system call

Using fork():

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
int main( )
{
      pid_t pid;
      cout<<"\n Original Process:”<<getpid()
      <<”\tParent Process:”<<getppid()<<endl;

      if ((pid = fork()) ==-1)
      {
            perror("Fork Error");
            exit(0);
      }
      if (pid == 0) { /* child */
            cout<<"\n Child Process:”<<getpid()
            <<”\tParent Process:”<<getppid()<<endl;
      }
      if (pid > 0) { /* parent */
            cout<<"\n Original Process:”<<getpid()
            <<”\tParent Process:”<<getppid()<<endl;
      }
      cout<<”\n end of Main\n”;
      return 0;
}
```

Description:

This program demonstrates process creation using the fork system call, printing the PID and PPID for both the original and child processes.

Using vfork():

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
int main( )
{
      pid_t pid;
      cout<<"\n Original Process:”<<getpid()
      <<”\tParent Process:”<<getppid()<<endl;
      if ((pid = vfork()) ==-1)
      {
            perror("Fork Error");
            exit(0);
      }
      if (pid == 0)
      {     /* child */
            cout<<"\n Child Process:”<<getpid()
            <<”\tParent Process:”<<getppid()<<endl;
      }
      if (pid > 0)
      {     /* parent */
            cout<<"\n Original Process:”<<getpid()
            <<”\tParent Process:”<<getppid()<<endl;
      }
      cout<<”\n end of Main\n”;
      return 0;
}
```

Description:

This program uses vfork() to create a new process. Just like the previous version, it prints the PID and PPID for both the original and child processes

6. Write a program to demonstrate the process is Zombie, and to avoid Zombie process.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <iostream>
using namespace std;
int main(void)
{
      pid_t pid;
      if ((pid = fork()) ==-1)
      {
            perror("Fork Error");
            exit(0);
      }
      if (pid == 0) /* child */
      {
            cout<<"\nChild :"<<getpid()
            <<" Waiting for parent to retrieve its exit status\n";
            exit(0);
      } /* parent */
      if (pid > 0)
      {
            sleep(5);
            cout<<"\nParent:"<<getpid();
            system("ps u");
      }
      return(0);
}
```

Description:

This C++ program demonstrates the creation of a child process using the fork() system call. The child process prints its PID and waits for the parent to retrieve its exit status. The parent process sleeps for 5 seconds, prints its PID, and displays process information using the "ps u" command.

7. Write a C program to create an Orphan Process.

```
include<stdio.h>
#include<stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <iostream>
using namespace std;
int main()
{
      pid_t pid;
      if ((pid = fork()) ==-1)
      {
            perror("Fork Error");
            exit(0);
      }
      if (pid == 0) /* child */
      {
            sleep(5);
            cout<<"\n Child :"<<getpid()
            <<"\tOrphan's Parent :"<<getppid()<<endl;
      }
      if (pid > 0) /* parent */
      {
            cout<<"\n Original Parent:"<<getpid()<<endl;
            exit(0);
      }
      return 0;
}
```

Description:

This program creates a child process using fork(). The child sleeps for 5 seconds, prints its PID, and the orphan's parent (PPID). The parent process prints its PID and exits, leaving the child as an orphan with the init process as its new parent.

8. Write a C program to demonstrate a parent process that uses wait() system call to catch the child 's exit code.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;
void pr_exit(int status)
{
      if (WIFEXITED(status))
            cout<<"\n Normal termination, exit status ="
            << WEXITSTATUS(status);
      else if (WIFSIGNALED(status))
            cout <<"\nAbnormal termination, signal number = "
            <<WTERMSIG(status);
      else if (WIFSTOPPED(status))
            cout<<"\nChild stopped, signal number = "
            <<WSTOPSIG(status);
}
int main()
{
      pid_t pid, childpid;
      int status;
      if ((pid = fork()) ==-1){
            perror("\nFork Error");
            return 0;
      }
      if(pid==0)
            exit(23);
      childpid=wait(&status);
      pr_exit(status);
      if ((pid = fork()) ==-1){
            perror("\nFork Error");
            return 0;
      }
      if(pid==0)
            abort();
      childpid=wait(&status);
```

```
      pr_exit(status);
      if ((pid = fork()) ==-1){
            perror("\nFork Error");
            return 0;
      }
      if(pid==0)
            int res=5/0;
      childpid=wait(&status);
      pr_exit(status);
      return 0;
}
```

Description:

This C++ program uses fork() to create child processes and demonstrates various termination statuses using the wait system call and the pr_exit function.

Outputs for 5-8:

#Q5

Case 1: Successful Fork

```
$ ./a.out
Original Process: 58745    Parent Process: 0
Child Process: 56817       Parent Process: 58745
Original Process: 58745    Parent Process: 0
end of Main
```

Case 2: Child Process Terminates First

```
$ ./a.out
Original Process: 58745    Parent Process: 0
Child Process: 56817  Parent Process: 58745
end of Main
```

Case 3: Parent Process Terminates First

```
$ ./a.out
Original Process: 58745    Parent Process: 0
Child Process: 56817  Parent Process: 58745
```

#Q6

```
Child : 12345 Waiting for parent to retrieve its exit status
54321

USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START TIME COMMAND
user1      54321  0.0  0.1 11974 2992 pts/1    S+   18:19:34 my_program
user1      12345  0.0  0.0 11974 2828 pts/1    S+   18:19:34 my_program
```


```
Child : 12345 Waiting for parent to retrieve its exit status
54321

USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START TIME COMMAND
user1      54321  0.0  0.1 11974 2992 pts/1    S+   18:19:34 my_program
user1      12345  0.0  0.0 11974 2828 pts/1    S+   18:19:34 my_program
```


#Q7

```
$ ./a.out
Child : 57345   Orphan's Parent : -1
Original Parent: 69351
```

#Q8

```
$ ./a.out
Normal termination, exit status = 23
Abnormal termination, signal number = 6
Abnormal termination, signal number = 8
```

9. Write a Program to demonstrate a race condition.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
void charatatime(char *str)
{
     char *ptr;
     int c;
     setbuf(stdout, NULL); /* set unbuffered */
     for (ptr = str; (c = *ptr++) != 0; )
          putc(c, stdout);
}
int main(int argc , char *argv[])
{
     pid_t pid;
     if ((pid = fork()) ==-1)
     {
          perror("Fork Error");
          return 0;
     }
     if (pid == 0)
          charatatime("CHILD   PROCESS   WRITING   DATA   TO   THE
          TERMINAL\n");
     if (pid > 0)
          charatatime("parent   process   writing   the   data   to
          terminal\n");
     return 0;
}
```

Description:

A simple C program demonstrating process creation using fork() and writing data to the terminal.

- <unistd.h>: Provides access to the POSIX operating system API, including functions like fork() and exec().
- <sys/types.h>: Defines various data types used in system calls, such as pid_t.
- <stdio.h>: Standard input/output header for functions like printf() and perror().
- <stdlib.h>: Standard library header for functions like exit().

- fork(): System call for creating a new process by duplicating the calling process.
- putc(): Writes a character to the standard output stream.
- pid_t: Data type representing process IDs.

10. Write a program to implement UNIX system (), using APIs.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
int status;
int system(const char *cmd)
{
     pid_t pid;
     if (cmd == NULL)
          return(1); /* always a command processor with Unix */
     if ( (pid = fork()) < 0) {
          status = -1; /* probably out of processes */
     }
     else if (pid == 0) {
          /* child */
          execl("/bin/sh", "sh", "-c", cmd ,(char *) 0);
          _exit(127);
      /* execl error */
     }
     else {
          /* parent */
          while (waitpid(pid, &status, 0) < 0)
               if (errno != EINTR)
               {
                    status = -1; /* error other than EINTR from
                    waitpid() */
                    Break;
               }
          }
     return(status);
}
```

```
int main(void)
{
     if ( (status = system("date; exit 0")) < 0)
          perror("system error");
     if ( (status = system("daaate")) < 0)
          perror("system error");
     if ( (status = system("who; exit 44")) < 0)
          perror("system error");
     exit(0);
}
```

Description:

A C program implementing a simplified version of the system function to execute shell commands and wait for their completion.

- <sys/wait.h>: Provides functions for waiting on child processes to change state.
- fork(): System call for creating a new process by duplicating the calling process.
- execl(): System call for executing a file with specified arguments.
- waitpid(): System call for waiting on a specific child process to change state.
- system(): User-defined function to execute shell commands.

11. Write a C/C++ program to catch, ignore and accept the signal, SIGINT.

1. Catch the signal

```
#include <signal.h>
#include <iostream.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
void handler(int signo)
{
     cout<<"\nsignal handlar : caught signal num is =>"
     <<signo<<endl;
     exit(0);
}
```

```
int main()
{
      signal(SIGINT,handler) ;

      while(1)
            cout<<"hello \t";
}
```

2. Taking default action

```
#include <signal.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
      signal(SIGINT, SIG_DFL) ;
      while(1)
            cout<<"hello\t";
}
```

3. Ignore the Signal

```
#include <signal.h>
#include <iostream.h>
#include <unistd.h>
int main() {
      signal(SIGINT,SIG_IGN)
      while(1)
            cout<<"hello\t ";
}
```

Description:

Program 1: C++ program demonstrating how to catch and handle the SIGINT signal.

- <signal.h>: Header file for signal handling in C programs.
- signal(): Function to set a signal handler for a specific signal.
- SIGINT: Signal generated by the Ctrl+C keyboard interrupt.
- handler(): User-defined signal handler function.
- exit(): Function to terminate the program.

Program 2: C++ program illustrating taking the default action for the SIGINT signal.

- SIG_DFL: Constant representing the default action for a signal.

Program 3: C++ program demonstrating how to ignore the SIGINT signal.

- SIG_IGN: Constant representing ignoring a signal.

12. Write a program to create, write to, and read from a pipe. Also write a program to create a pipe from the parent to child and send data down the pipe.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
      int pfds[2];
      char buf[30];
      if (pipe(pfds) == -1)
      {
            perror("pipe");
            exit(1);
      }
      cout<<"writing to file descriptor #"<<pfds[1]<<endl;
      write(pfds[1], "test", 5);
      cout<<"reading from file descriptor #"<<pfds[0]<<endl;
      read(pfds[0], buf, 5);
      cout<<"read : "<<buf<<endl;
      return 0;
}
```

Description:

A C++ program demonstrating inter-process communication using pipes.

- <stdio.h>: Standard input/output header for functions like perror().
- <stdlib.h>: Standard library header for functions like exit().
- <errno.h>: Header file defining error numbers for error handling.
- <unistd.h>: Provides access to POSIX operating system API, including functions like pipe(), read(), and write().
- pipe(): System call for creating a pipe, which establishes a unidirectional communication channel.
- write(): System call for writing data to a file descriptor.
- read(): System call for reading data from a file descriptor.

Outputs for 9-12:

#Q9:

```
$ ./a.out
parent process writing the data to terminal
CHILD PROCESS WRITING DATA TO THE TERMINAL
```

#Q10:

```
$ ./a.out
Thu Feb 17 12:34:56 UTC 2024
/bin/sh: 1: daaate: not found
USERNAME tty1 2024-02-17 12:34 (:0)
system error: No child processes
```

#Q11:

Program 1:

```
$ ./a.out
hello    hello    hello    hello    hello    hello    hello    hello
hello    hello
^C
signal handler: caught signal num is =>2
```

Program 2:

```
$ ./a.out
hello    hello    hello    hello    hello    hello    hello    hello
hello    hello
^C
```

Program 3:

```
$ ./a.out
hello    hello    hello    hello    hello    hello    hello    hello
hello    hello
^C
```

#Q12:

```
$ ./a.out
writing to file descriptor #3
reading from file descriptor #4
read: test
```