How to Run Shell Scripts

There are two ways you can execute your shell scripts. Once you have created a script file:

Method 1:

Pass the file as an argument to the shell that you want to interpret your script.

Step 1 : create the script using vi, ex or ed

For example, the script file show has the following lines

```
echo Here is the date and time
date
```

Step 2 : To run the script, pass the filename as an argument to the sh (shell)

```
$ sh show
Here is the date and time
Sat jun 03 13:40:15 PST 2006
```

Method 2:

Make your script executable using the chmod command.

When we create a file, by default it is created with read and write permission turned on and execute permission turned off. A file can be made executable using chmod.

Step 1 : create the script using vi, ex or ed

For example, the script file show has the following lines

```
echo Here is the date and time
date
```

Step 2 : Make the file executable

```
$ chmod u+x script_file
$ chmod u+x show
```

Step 3 : To run the script, just type the filename

```
$ show
Here is the date and time
Sat jun 03 13:40:15 PST 2006
```

How to run C programs

Step 1 : Use an editor, such as vi, ex, or ed to write the program. The name of the file containing the program should end in .c.

For example, the file show.c contains the following lines :

```
main()
{
printf(" welcome ");
}
```

Step 2 : Submit the file to CC (the C Compiler)

$ cc show.c
If the program is okay, the compiled version is placed in a file called a.out
Step 3 : To run the program, type a.out
$ a.out
Welcome

1.Write a shell script that accepts a file name, starting and ending numbers as arguments and displays all the lines between the given line numbers.

```
#!/bin/sh
echo “Enter file name”
read file
echo “First Line  displays :”
head -1 $file
echo “Last line displays :”
tail -1 $file
echo “Displays all lines in a given file :”
cat $file
```

```
“Enter file name”
sample
“First Line displays :”
#!/bin/bash
“Last line displays :”

“Displays all lines in a given file :”
#!/bin/bash
echo "Hello world"
echo "Knowledge is Power"

beast@ubuntu:~$
```

2. Write a shell script that deletes all lines containing the specified word in one or more files supplied as arguments to it.

```
#!/bin/csh
grep -v $argv[1] $argv[2]
```

or

```
sed -e '/word/d' file1 file2 file3 > file.out
```

Arguments can be any word which is to be deleted except commands

To delete first character

sed 's'/^.//p' file name

Deletes last second character in every line.

sed –n 's/\(.\)\(.\)$/\2/p' filename

First word and second word goes to second word and first word in every line.

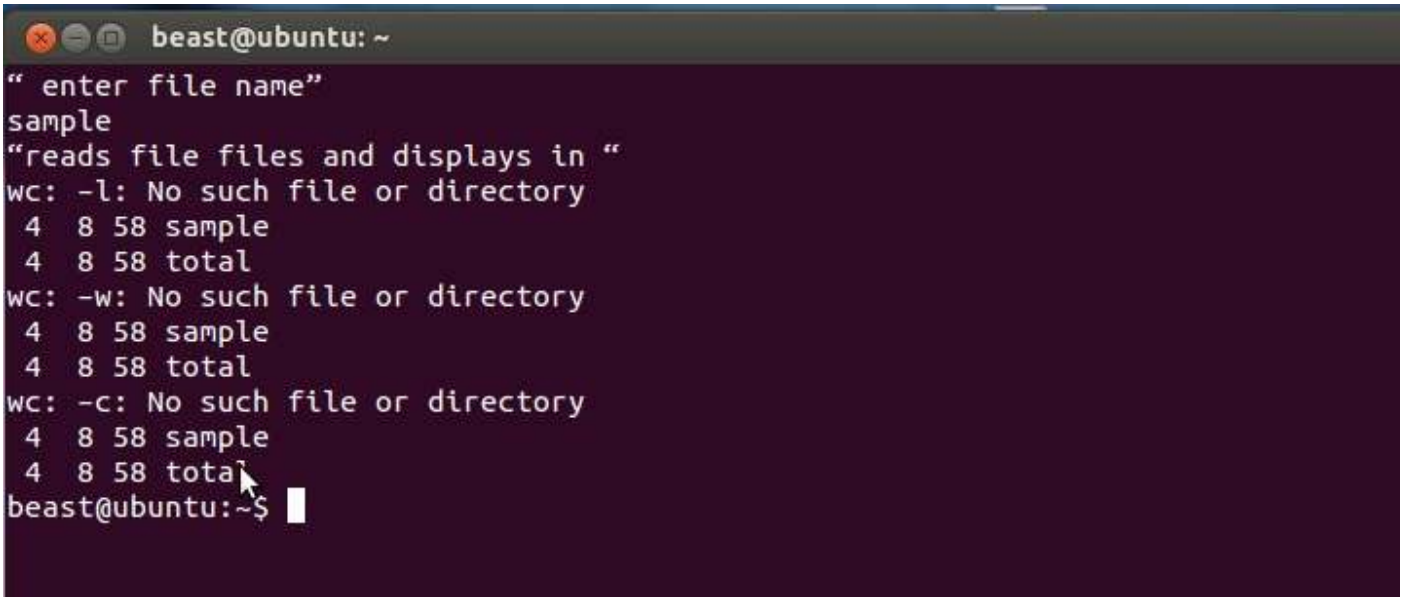
sed –n 's/^\([a-zA-Z0-9]*\) *\([a-zA-Z0-9]*\)/\2\1/p' filename

3. Write a shell script that displays a list of all files in the current directory to which the user has read, write and execute permissions.

```
echo "List of Files which have Read, Write and Execute Permissions in Current Directory"
for file in *
do
if [ -r $file -a -w $file -a -x $file ]
then
echo $file
fi
done
```

4.Write a shell script that receives any number of file names as arguments checks if every argument supplied is a file as its arguments, counts and reports the occurrence of each word that is present in the first argument file on other argument files.

```
echo " enter file name"
read file
echo "reads file files and displays in "
wc –l $file
wc –w  $file
wc –c $file
```



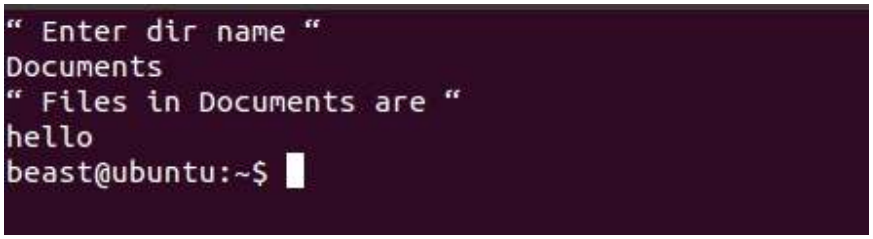
5. Write a shell script that accepts a list of file names as its arguments, counts and reports the occurrence of each word that is present in the first argument file on other argument files.

```
echo "Enter file or dir name"
read file
if [ -d $file ]
then
echo " given name is directory"
elif [ -f $file ]
then
echo " given name is file: $file"
echo "No of lines in file are : wc –l $file "
else
echo " given name is not a file or a directory
```

6. Write a shell script to list all of the directory files in a directory.

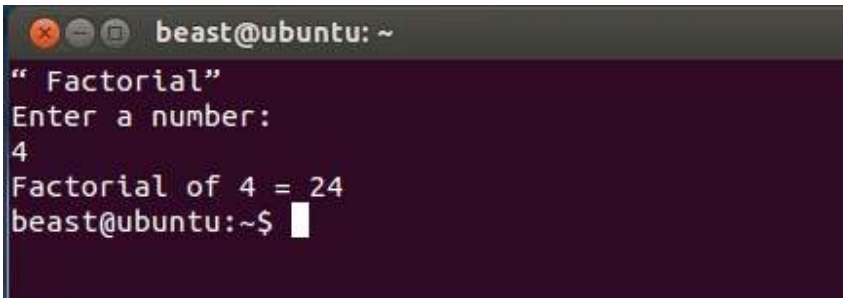
```
echo " Enter dir name "
read dir
if [ -d $dir ]
then
echo " Files in $dir are "
ls $dir
else
echo " Dir does not exist"
fi
```

Output:



7. Write a shell script to find factorial of a given number.

```
echo " Factorial"
echo "Enter a number: "
read f
fact=1
factorial=1
while [ $fact -le $f ]
do
factorial=`expr $factorial \* $fact`
fact=`expr $fact + 1`
done
echo "Factorial of $f = $factorial"
```



8. Implement in C the following Unix commands and System calls.

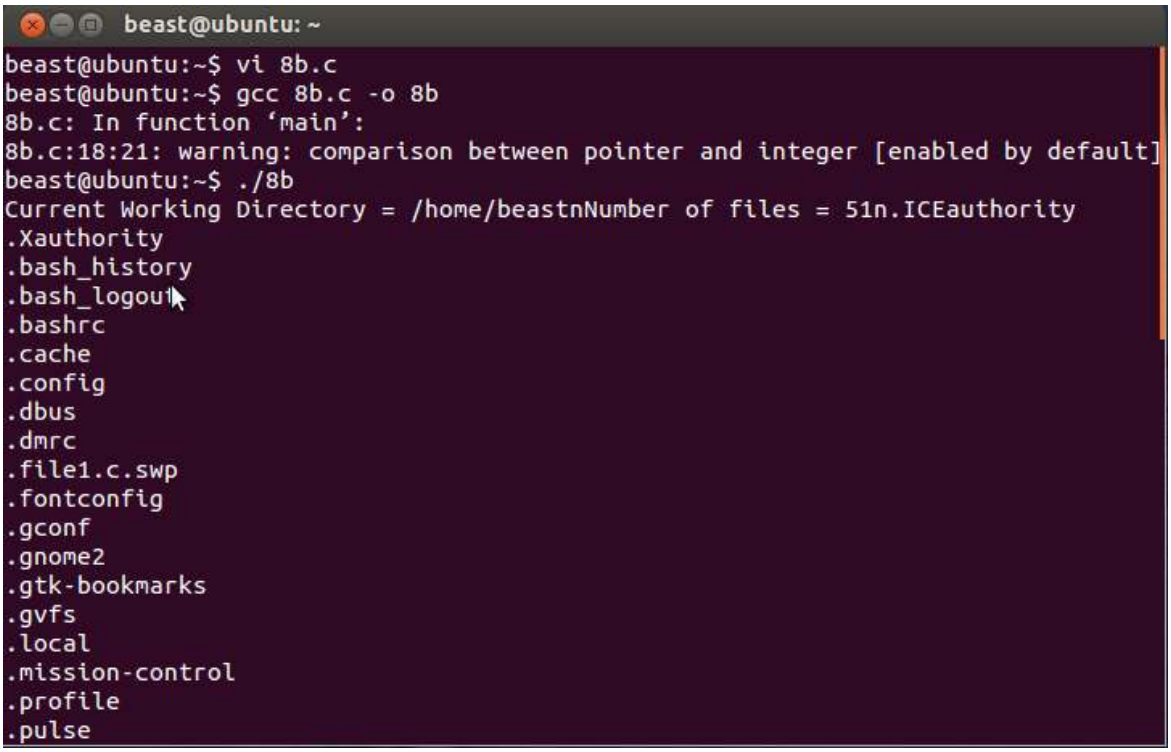
a. cat b. ls c. mv.

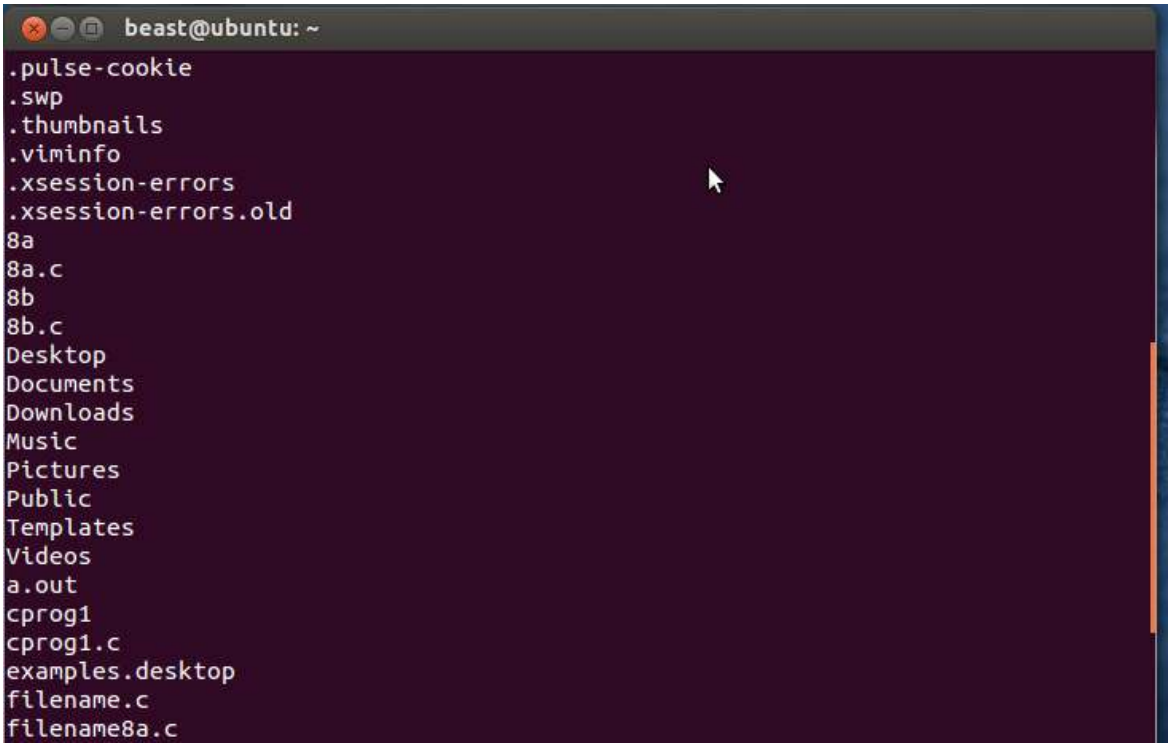
a) Implement in C the cat Unix command using system calls

```
#include<fcntl.h>
#include<sys/stat.h>
#define BUFSIZE 1
int main(int argc, char **argv)
{
  int fd1;
  int n;
  char buf;
  fd1=open(argv[1],O_RDONLY);
 printf("SuhritSolutions Printing Files\n");
  while((n=read(fd1,&buf,1))>0)
  {
   printf("%c",buf);
/*         or
    write(1,&buf,1);  */
   }
  return (0);
}
```

b) Implement in C the following ls Unix command using system calls

```
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
#include <stdio.h>

#define FALSE 0
#define TRUE 1

extern  int alphasort();

char pathname[MAXPATHLEN];

main()   {
int count,i;
struct dirent **files;
int file_select();

if (getwd(pathname) == NULL )
{ printf("Error getting pathn");
exit(0);
}
                printf("Current Working Directory = %sn",pathname);
        count = scandir(pathname, &files, file_select, alphasort);

                        if (count <= 0)
        {                printf("No files in this directoryn");
                                                exit(0);
                                        }
                        printf("Number of files = %dn",count);
                for (i=1;i<count+1;++i)
                        printf("%s  \n",files[i-1]->d_name);

        }
int file_select(struct direct   *entry)
 {
if ((strcmp(entry->d_name, ".") == 0) ||(strcmp(entry->d_name, "..") == 0))
                                        return (FALSE);
else
                        return (TRUE);                  }
```





c) Implement in C the Unix command mv using system calls

```
#include<fcntl.h>
#include<stdio.h>
#include<unistd.h>
#include<sys/stat.h>
int main(int argc, char **argv)
{
  int fd1,fd2;
  int n,count=0;
  fd1=open(argv[1],O_RDONLY);
fd2=creat(argv[2],S_IWUSR);
rename(fd1,fd2);
unlink(argv[1]);
return (0);
}
```

9. Write a C program that takes one or more file or directory names as command line input and reports the following information on the file.

1. file type

2. number of links

3. read, write and execute permissions

4. time of last access

(Note: use /fstat system calls)

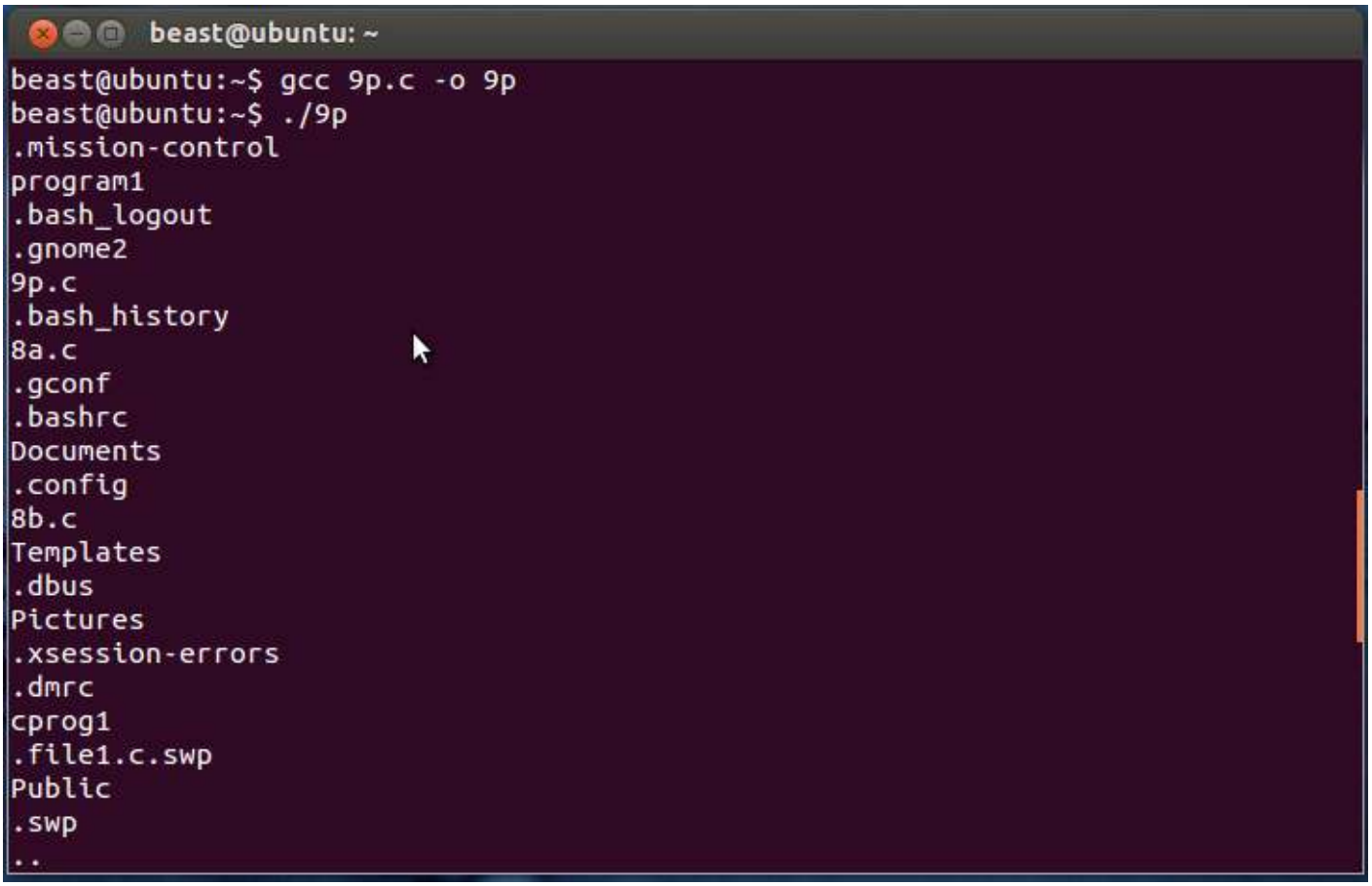
PROGRAM

```
#include<stdio.h>
main()
{
FILE *stream;
int buffer_character;
stream=fopen("test","r");
if(stream==(FILE*)0)
```

```
{
fprintf(stderr,"Error opening file(printed to standard error)\n");
fclose(stream);
exit(1);
}}
if(fclose(stream))==EOF)
{
fprintf(stderr,"Error closing stream.(printed to standard error)\n);
exit(1);
}
return();
}
```

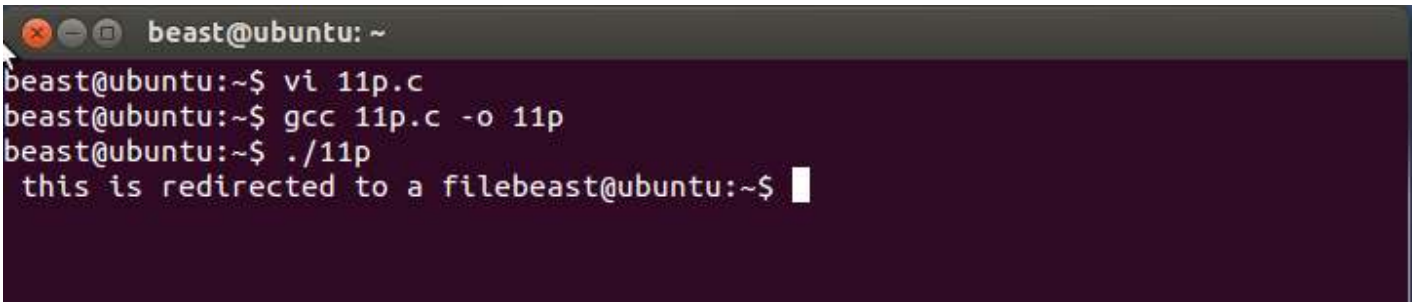
10. Write a C program to emulate the Unix ls – l command.

```
        #include <stddef.h>
        #include <stdio.h>
    #include <sys/types.h>
    #include <dirent.h>

    int main (void) {

    DIR *dp;
    struct dirent *ep;
    dp = opendir (".");
    if (dp != NULL) {
 while (ep = readdir (dp))
printf("%s\n", ep->d_name);
closedir (dp);
    }
    else
perror ("Couldn't open the directory");
    return 0;
    }
```



11. Write a C program that redirects a standard output to a file. Ex: ls >f1.

```
/* freopen example: redirecting stdout */
#include <stdio.h>

int main ()
{
  freopen ("myfile.txt","w",stdout);
  printf ("This sentence is redirected to a file.");
  fclose (stdout);
  return 0;
}
```



12. Write a C program to create a child process and allow the parent to display "parent" and the child to display "child" on the screen.

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <strsafe.h>

#define BUFSIZE 4096

HANDLE g_hChildStd_IN_Rd = NULL;
HANDLE g_hChildStd_IN_Wr = NULL;
HANDLE g_hChildStd_OUT_Rd = NULL;
HANDLE g_hChildStd_OUT_Wr = NULL;

HANDLE g_hInputFile = NULL;

void CreateChildProcess(void);
void WriteToPipe(void);
void ReadFromPipe(void);
void ErrorExit(PTSTR);

int _tmain(int argc, TCHAR *argv[])
{
  SECURITY_ATTRIBUTES saAttr;

  printf("\n->Start of parent execution.\n");

// Set the bInheritHandle flag so pipe handles are inherited.

  saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
  saAttr.bInheritHandle = TRUE;
  saAttr.lpSecurityDescriptor = NULL;
```

```
// Create a pipe for the child process's STDOUT.

  if ( ! CreatePipe(&g_hChildStd_OUT_Rd, &g_hChildStd_OUT_Wr, &saAttr, 0) )
    ErrorExit(TEXT("StdoutRd CreatePipe"));

// Ensure the read handle to the pipe for STDOUT is not inherited.

  if ( ! SetHandleInformation(g_hChildStd_OUT_Rd, HANDLE_FLAG_INHERIT, 0) )
    ErrorExit(TEXT("Stdout SetHandleInformation"));

// Create a pipe for the child process's STDIN.

  if (! CreatePipe(&g_hChildStd_IN_Rd, &g_hChildStd_IN_Wr, &saAttr, 0))
    ErrorExit(TEXT("Stdin CreatePipe"));

// Ensure the write handle to the pipe for STDIN is not inherited.

  if ( ! SetHandleInformation(g_hChildStd_IN_Wr, HANDLE_FLAG_INHERIT, 0) )
    ErrorExit(TEXT("Stdin SetHandleInformation"));

// Create the child process.

  CreateChildProcess();

// Get a handle to an input file for the parent.
// This example assumes a plain text file and uses string output to verify data flow.

  if (argc == 1)
    ErrorExit(TEXT("Please specify an input file.\n"));

  g_hInputFile = CreateFile(
     argv[1],
     GENERIC_READ,
     0,
     NULL,
     OPEN_EXISTING,
     FILE_ATTRIBUTE_READONLY,
     NULL);

  if ( g_hInputFile == INVALID_HANDLE_VALUE )
```

```
    ErrorExit(TEXT("CreateFile"));

// Write to the pipe that is the standard input for a child process.
// Data is written to the pipe's buffers, so it is not necessary to wait
// until the child process is running before writing data.

  WriteToPipe();
  printf( "\n->Contents of %s written to child STDIN pipe.\n", argv[1]);

// Read from pipe that is the standard output for child process.

  printf( "\n->Contents of child process STDOUT:\n\n", argv[1]);
  ReadFromPipe();

  printf("\n->End of parent execution.\n");

// The remaining open handles are cleaned up when this process terminates.
// To avoid resource leaks in a larger application, close handles explicitly.

  return 0;
}

void CreateChildProcess()
// Create a child process that uses the previously created pipes for STDIN and STDOUT.
{
  TCHAR szCmdline[]=TEXT("child");
  PROCESS_INFORMATION piProcInfo;
  STARTUPINFO siStartInfo;
  BOOL bSuccess = FALSE;

// Set up members of the PROCESS_INFORMATION structure.

  ZeroMemory( &piProcInfo, sizeof(PROCESS_INFORMATION) );

// Set up members of the STARTUPINFO structure.
// This structure specifies the STDIN and STDOUT handles for redirection.

  ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );
  siStartInfo.cb = sizeof(STARTUPINFO);
  siStartInfo.hStdError = g_hChildStd_OUT_Wr;
  siStartInfo.hStdOutput = g_hChildStd_OUT_Wr;
```

```
  siStartInfo.hStdInput = g_hChildStd_IN_Rd;
  siStartInfo.dwFlags |= STARTF_USESTDHANDLES;

// Create the child process.

  bSuccess = CreateProcess(NULL,
    szCmdline,     // command line
    NULL,          // process security attributes
    NULL,          // primary thread security attributes
    TRUE,          // handles are inherited
    0,             // creation flags
    NULL,          // use parent's environment
    NULL,          // use parent's current directory
    &siStartInfo,  // STARTUPINFO pointer
    &piProcInfo);  // receives PROCESS_INFORMATION

  // If an error occurs, exit the application.
  if ( ! bSuccess )
    ErrorExit(TEXT("CreateProcess"));
  else
  {
    // Close handles to the child process and its primary thread.
    // Some applications might keep these handles to monitor the status
    // of the child process, for example.

    CloseHandle(piProcInfo.hProcess);
    CloseHandle(piProcInfo.hThread);
  }
}

void WriteToPipe(void)

// Read from a file and write its contents to the pipe for the child's STDIN.
// Stop when there is no more data.
{
  DWORD dwRead, dwWritten;
  CHAR chBuf[BUFSIZE];
  BOOL bSuccess = FALSE;

  for (;;)
  {
```

```
    bSuccess = ReadFile(g_hInputFile, chBuf, BUFSIZE, &dwRead, NULL);
    if ( ! bSuccess || dwRead == 0 ) break;

    bSuccess = WriteFile(g_hChildStd_IN_Wr, chBuf, dwRead, &dwWritten, NULL);
    if ( ! bSuccess ) break;
  }

// Close the pipe handle so the child process stops reading.

  if ( ! CloseHandle(g_hChildStd_IN_Wr) )
    ErrorExit(TEXT("StdInWr CloseHandle"));
}

void ReadFromPipe(void)

// Read output from the child process's pipe for STDOUT
// and write to the parent process's pipe for STDOUT.
// Stop when there is no more data.
{
  DWORD dwRead, dwWritten;
  CHAR chBuf[BUFSIZE];
  BOOL bSuccess = FALSE;
  HANDLE hParentStdOut = GetStdHandle(STD_OUTPUT_HANDLE);

// Close the write end of the pipe before reading from the
// read end of the pipe, to control child process execution.
// The pipe is assumed to have enough buffer space to hold the
// data the child process has already written to it.

  if (!CloseHandle(g_hChildStd_OUT_Wr))
    ErrorExit(TEXT("StdOutWr CloseHandle"));

  for (;;)
  {
    bSuccess = ReadFile( g_hChildStd_OUT_Rd, chBuf, BUFSIZE, &dwRead, NULL);
    if( ! bSuccess || dwRead == 0 ) break;

    bSuccess = WriteFile(hParentStdOut, chBuf,
                         dwRead, &dwWritten, NULL);
    if (! bSuccess ) break;
  }
```

```
}

void ErrorExit(PTSTR lpszFunction)

// Format a readable error message, display a message box,
// and exit from the application.
{
   LPVOID lpMsgBuf;
   LPVOID lpDisplayBuf;
   DWORD dw = GetLastError();

   FormatMessage(
      FORMAT_MESSAGE_ALLOCATE_BUFFER |
      FORMAT_MESSAGE_FROM_SYSTEM |
      FORMAT_MESSAGE_IGNORE_INSERTS,
      NULL,
      dw,
      MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
      (LPTSTR) &lpMsgBuf,
      0, NULL );

   lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
      (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(TCHAR));
   StringCchPrintf((LPTSTR)lpDisplayBuf,
      LocalSize(lpDisplayBuf) / sizeof(TCHAR),
      TEXT("%s failed with error %d: %s"),
      lpszFunction, dw, lpMsgBuf);
   MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB_OK);

   LocalFree(lpMsgBuf);
   LocalFree(lpDisplayBuf);
   ExitProcess(1);
}
```

The following is the code for the child process. It uses the inherited handles for STDIN and STDOUT to access the pipe created by the parent. The parent process reads from its input file and writes the information to a pipe. The child receives text through the pipe using STDIN and writes to the pipe using STDOUT. The parent reads from the read end of the pipe and displays the information to its STDOUT.

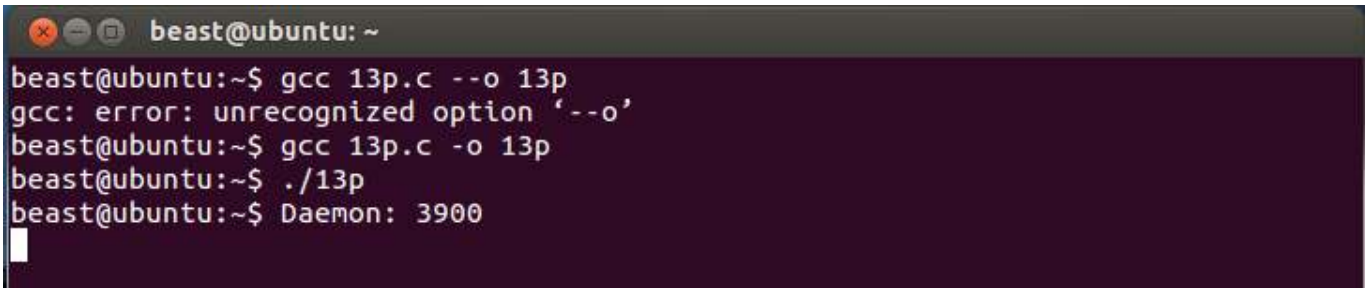
```
#include <windows.h>
#include <stdio.h>
```

```
#define BUFSIZE 4096

int main(void)
{
  CHAR chBuf[BUFSIZE];
  DWORD dwRead, dwWritten;
  HANDLE hStdin, hStdout;
  BOOL bSuccess;

  hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
  hStdin = GetStdHandle(STD_INPUT_HANDLE);
  if (
     (hStdout == INVALID_HANDLE_VALUE) ||
     (hStdin == INVALID_HANDLE_VALUE)
    )
    ExitProcess(1);

  // Send something to this process's stdout using printf.
  printf("\n ** This is a message from the child process. ** \n");

  // This simple algorithm uses the existence of the pipes to control execution.
  // It relies on the pipe buffers to ensure that no data is lost.
  // Larger applications would use more advanced process control.

  for (;;)
  {
  // Read from standard input and stop on error or no data.
    bSuccess = ReadFile(hStdin, chBuf, BUFSIZE, &dwRead, NULL);

    if (! bSuccess || dwRead == 0)
      break;

  // Write to standard output and stop on error.
    bSuccess = WriteFile(hStdout, chBuf, dwRead, &dwWritten, NULL);

    if (! bSuccess)
      break;
  }
  return 0;
}
```

13. Write a C program to create a zombie process.

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

#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
        /*
         * Funzione che mi crea un demone
         */

        int pid;

        // create - fork 1
        if(fork()) return 0;

        // it separates the son from the father
        chdir("/");
        setsid();
        umask(0);

        // create - fork 2
        pid = fork();

        if(pid)
        {
                printf("Daemon: %d\n", pid);
                return 0;
        }

        /****** Codice da eseguire ********/
        FILE *f;

        f=fopen("/tmp/coa.log", "w");

        while(1)
        {
                fprintf(f, "ciao\n");
```

```
                fflush(f);
                sleep(2);
        }
        /*********************************/
}
```



14. Write a C program that illustrates how an orphan is created.

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

int main(){
      pid_t i;

      i=fork();

      if (i==0){
      while (1){
            printf("child process pid = %d, ppid=%d\n",(int)getpid(), (int)getppid());
            sleep(3);
      }
       }else{
             printf("parent has finished");
       }
      return 0;
}
```

```
beast@ubuntu: ~
beast@ubuntu:~$ vi 14p.c
beast@ubuntu:~$ gcc 14p.c -o 14p
beast@ubuntu:~$ ./14p
parent has finishedbeast@ubuntu:~$ child process pid = 3987, ppid=1
child process pid = 3987, ppid=1
child process pid = 3987, ppid=1
child process pid = 3987, ppid=1
child process pid = 3987, ppid=1
```

15. Write a C program that illustrates the following.

a) Creating a message queue.

b) Writing to a message queue.

c) Reading from a message queue.

```
#include <stdio.h>
 #include <sys/ipc.h>
 #include <fcntl.h>
 #define MAX 255
      struct mesg
      {
          long type;
          char mtext[MAX];
      }  *mesg;
      char buff[MAX];
 main()
 {
      int mid,fd,n,count=0;;
      if((mid=msgget(1006,IPC_CREAT | 0666))<0)
      {
            printf("\n Can't create Message Q");
            exit(1);
      }
      printf("\n Queue id:%d", mid);
      mesg=(struct mesg *)malloc(sizeof(struct mesg));
      mesg ->type=6;
      fd=open("fact",O_RDONLY);
      while(read(fd,buff,25)>0)
      {
```

```
        strcpy(mesg ->mtext,buff);
        if(msgsnd(mid,mesg,strlen(mesg ->mtext),0)== -1)
            printf("\n Message Write Error");
    }

    if((mid=msgget(1006,0))<0)
    {
        printf("\n Can't create Message Q");
        exit(1);
    }
    while((n=msgrcv(mid,&mesg,MAX,6,IPC_NOWAIT))>0)
        write(1,mesg.mtext,n);
        count++;
    if((n= = -1)&(count= =0))
        printf("\n No Message Queue on Queue:%d",mid);

}
```

```
beast@ubuntu:~$ vi 15p.c
beast@ubuntu:~$ gcc 15p.c -o 15p
beast@ubuntu:~$ ./15p
message queue created 0
message placed on the queue successfully
msgrcv:received message:mtype: 0mtext: hello worldbeast@ubuntu:~$
```

16. Write a C program that implements a producer-consumer system with two processes.(using semaphores)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define NUM_LOOPS 20
int main(int argc, char* argv[])
{
   int sem_set_id;
   union semun sem_val;
```

```
   int child_pid;
   int i;
   struct sembuf sem_op;
   int rc;
   struct timespec delay;

   sem_set_id = semget(IPC_PRIVATE, 1, 0600);
   if (sem_set_id == -1) {
          perror("main: semget");
          exit(1);
   }
   printf("semaphore set created,
semaphore set id '%d'.\n", sem_set_id);

   sem_val.val = 0;
   rc = semctl(sem_set_id, 0, SETVAL, sem_val);
   child_pid = fork();
   switch (child_pid) {
          case -1:
             perror("fork");
             exit(1);
          case 0:
             for (i=0; i<NUM_LOOPS; i++) {
                    sem_op.sem_num = 0;
                    sem_op.sem_op = -1;
                    sem_op.sem_flg = 0;
                    semop(sem_set_id, &sem_op, 1);
                    printf("consumer: '%d'\n", i);
                    fflush(stdout);
                  sleep(3);
             }
             break;
          default:
             for (i=0; i<NUM_LOOPS; i++)
          {
                    printf("producer: '%d'\n", i);
                    fflush(stdout);
                    sem_op.sem_num = 0;
                    sem_op.sem_op = 1;
                    sem_op.sem_flg = 0;
                    semop(sem_set_id, &sem_op, 1);
```

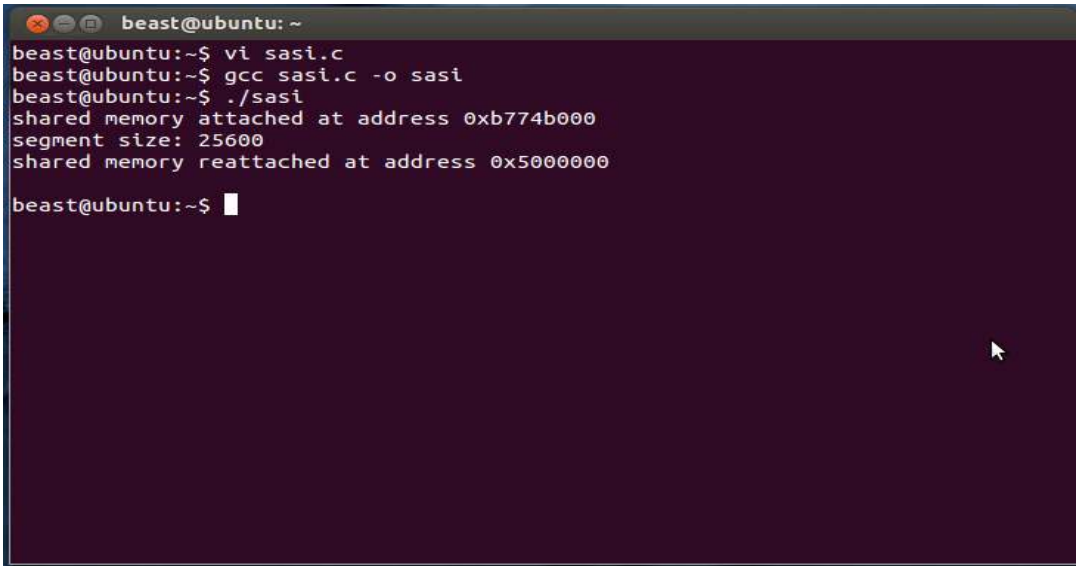
```
                sleep(2);
                if (rand() > 3*(RAND_MAX/4))
                {
                   delay.tv_sec = 0;
                   delay.tv_nsec = 10;
                   nanosleep(&delay, NULL);
                }
            }
            break;
    }   return 0;
}
```

```
beast@ubuntu:~$ vi 17p.c
beast@ubuntu:~$ gcc 17p.c -o 17p
beast@ubuntu:~$ ./17p
semaphoer set createdsemaphore set id '327690'producer: '0'consumer:'0'producer:
'1'beast@ubuntu:~$
```

17. Write a C program that illustrates inter process communication using shared memory.

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main ()
{
int segment_id;
char* shared_memory;
struct shmid_ds shmbuffer;
int segment_size;
const int shared_segment_size = 0x6400;
/* Allocate a shared memory segment. */
segment_id = shmget (IPC_PRIVATE, shared_segment_size,
IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
/* Attach the shared memory segment. */
shared_memory = (char*) shmat (segment_id, 0, 0);
printf ("shared memory attached at address %p\n", shared_memory);
/* Determine the segment's size. */
shmctl (segment_id, IPC_STAT, &shmbuffer);
segment_size = shmbuffer.shm_segsz;
printf ("segment size: %d\n", segment_size);
/* Write a string to the shared memory segment. */
printf (shared_memory, "Hello, world.");
```

```
/* Detach the shared memory segment. */
shmdt (shared_memory);
/* Reattach the shared memory segment, at a different address. */
shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
printf ("shared memory reattached at address %p\n", shared_memory);
/* Print out the string from shared memory. */
printf ("%s\n", shared_memory);
/* Detach the shared memory segment. */
shmdt (shared_memory);
/* Deallocate the shared memory segment. */
shmctl (segment_id, IPC_RMID, 0);
return 0;
}
```

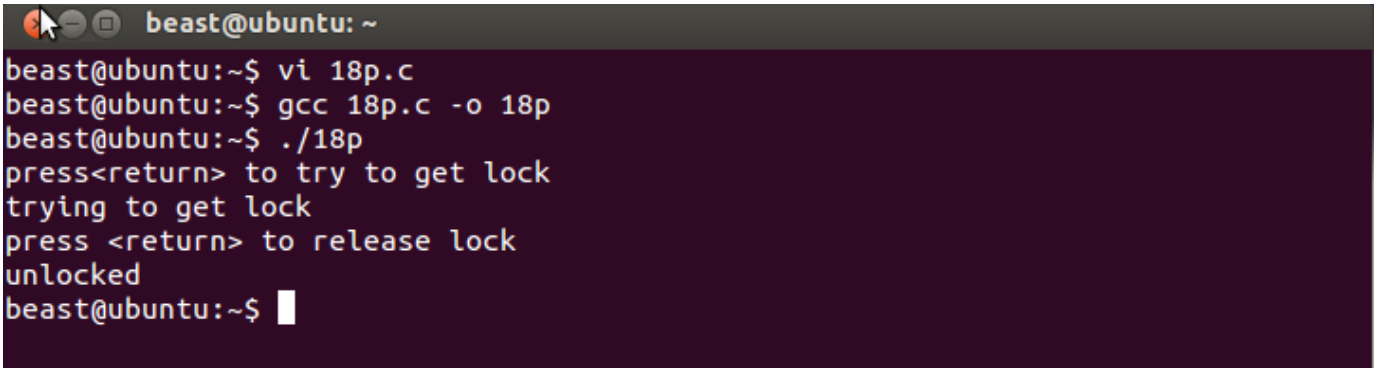


18. Write a C program that illustrates file locking using semaphores.

```
#include <stdio.h>
 #include <sys/file.h>
 #include <error.h>
 #include <sys/sem.h>
 #define MAXBUF 100
 #define KEY 1216
 #define SEQFILE "suhritfile"
 int semid,fd;
 void my_lock(int);
 void my_unlock(int);
```

```
union semnum
{
    int val;
    struct semid_ds *buf;
    short *array;
}arg;
int main(])
{
     int child, i,n, pid, seqno;
     char buff[MAXBUF+1];
     pid=getpid();
     if((semid=semget(KEY, 1, IPC_CREAT | 0666))= = -1)
     {
          perror(“semget”);
          exit(1);
     }
     arg.val=1;
     if(semctl(semid,0,SETVAL,arg)<0)
          perror(“semctl”);
     if((fd=open(SEQFILE,2))<0)
     {
          perror(“open”);
          exit(1);
     }
     pid=getpid();
     for(i=0;i<2;i++)
     {
          my_lock(fd);
          lseek(fd,01,0);
          if((n=read(fd,buff,MAXBUF))<0)
          {
               perror(“read”);
               exit(1);
          }
          printf(“pid:%d, Seq no:%d\n”, pid, seqno);
          seqno++;
          sprintf(buff,”%d\n”, seqno);
          n=strlen(buff);
          lseek(fd,01,0);
          if(write(fd,buff,n)!=n)
          {
```

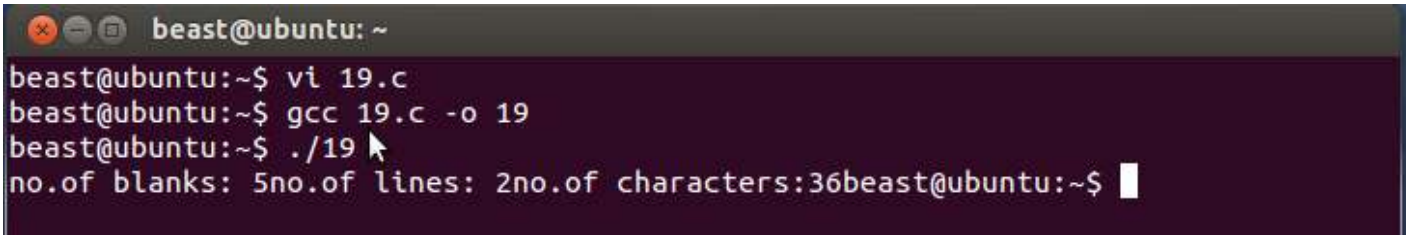
```
                perror("write");
                exit(1);
            }
            sleep(1);
            my_unlock(fd);
        }
    }
    void my_lock(int fd)
    {
        struct sembuff sbuf=(0, -1, 0);
        if(semop(semid, &sbuf, 1)= =0)
            printf("Locking: Resource…\n");
        else
            printf("Error in Lock\n");
    }
    void my_unlock(int fd)
    {
        struct sembuff sbuf=(0, 1, 0);
        if(semop(semid, &sbuf, 1)= =0)
            printf("UnLocking: Resource…\n");
        else
            printf("Error in Unlock\n");
    }
```



19. Write a C program that counts the number of blanks in a text file using standard I/O.

```
#include <fcntl.h>
#include < sys/stat.h>
#include <stdio.h>
int main(int argc, char **argv)
{
```

```
  FILE *fd1;
  int n,count=0;
  char buf;
  fd1=fopen(argv[1],"r");
  while(!feof(fd1))
  {
  buf=fgetc(fd1);
   if(buf==' ')
      count=count+1;
  }
  printf("\n Total Blanks= %d",count);
return (0);
}
```



20. Write a C program that illustrates communication between two unrelated processes using named pipe.

A regular pipe can only connect two related processes. It is created by a process and will vanish when the last process closes it.
A named pipe, also called a FIFO for its behavior, can be used to connect two unrelated processes and exists independently of the processes; meaning it can exist even if no one is using it. A FIFO is created using the mkfifo() library function.
WRITER.C

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

int main()
{
   int fd;
   char * myfifo = "/tmp/myfifo";

   /* create the FIFO (named pipe) */
   mkfifo(myfifo, 0666);
```

```
   /* write "Hi" to the FIFO */
   fd = open(myfifo, O_WRONLY);
   write(fd, "Hi", sizeof("Hi"));
   close(fd);

   /* remove the FIFO */
   unlink(myfifo);

   return 0;
}
```

READER.C

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

#define MAX_BUF 1024

int main()
{
   int fd;
   char * myfifo = "/tmp/myfifo";
   char buf[MAX_BUF];

   /* open, read, and display the message from the FIFO */
   fd = open(myfifo, O_RDONLY);
   read(fd, buf, MAX_BUF);
   printf("Received: %s\n", buf);
   close(fd);

   return 0;
}
```