Program 1

Write a C/C++ POSIX compliant program to check the following limits:
(i) No. of clock ticks (ii) Max. no. of child processes (iii) Max. path length
(iv) Max. no. of characters in a file name (v) Max. no. of open files/ process

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE
#include<stdio.h>
#include<unistd.h>
#include<iostream>
using namespace std;

        int main()
        {
          int res;

         if( (res = sysconf(_SC_CLK_TCK))= = -1)
         perror("sysconf error");
          else
         cout<<"max no of clock ticks:"<<res<<endl;

         if((res = sysconf(_SC_CHILD_MAX))= = - 1)
         perror("sysconf error");
          else
         cout<<"max no of child processes:"<<res<<endl;

         if((res = pathconf("/", _PC_PATH_MAX)) = =-1)
         perror("pathconf error");
          else
         cout<<"max no of characters in path name:"<<res<<endl;

         if((res = pathconf("/", _PC_NAME_MAX))= = -1)
         perror("pathconf error" );
         else
         cout<<"max no of characters in filename:"<<res<<endl;

        if((res=sysconf(_SC_OPEN_MAX)) = = -1)
        perror("sysconf error");
         else
        cout<<"max no of open files:"<<res<<endl;

         return 0;
        }
```

****
Program1 output:
****
[root@localhost ~]# vi p1.cpp
[root@localhost ~]# g++ p1.cpp
[root@localhost ~]# ./a.out
Max no. of clock ticks:100
Max no. of child processes: 1024
Max. no. of characters in a pathname:4096
Max. no. of characters in a filename:255
Max. no. of open files:1024

Viva Questions

1. **What is UNIX?**

It is a portable operating system that is designed for both efficient multi-tasking and multi-user functions. Its portability allows it to run on different hardware platforms. It was written is C and lets user do processing and control under a shell.

2. What is the use of sysconf,pathconf,fpathconf?

sysconf is used to query system-wide configuration limits that are implemented on a given system. **Pathconf and fpathconf** are used to query file related configuration limits.

3.**Difference between pathconf( ) and fpathconf( ).**

pathconf( ) takes file's pathname as argument, whereas fpathconf takes a file descriptor as argument.

Prototypes:
#include<unistd.h>

longsysconf(constintlimit_name);
longpathconf(constchar*pathname,intflimit_name);
longfpathconf(constintfdesc,intflimit_name);

The *limit_name*argumentvalueisamanifestedconstantasdefinedinthe<unistd.h>header.The possible values and the corresponding data returned by the *sysconf* function are:

Limitvalue	*Sysconf*returndata
_SC_ARG_MAX	Maximumsizeof argumentvalues(inbytes)thatmay
_SC_CHILD_MAX	Maximumnumberofchildprocessesthatmaybeowned bya process
_SC_OPEN_MAX	Maximumnumberofopened filesper process

_SC_NGROUPS_MAX	Maximumnumberofsupplemental groups perprocess
_SC_CLK_TCK	Thenumber ofclockticksper second
SC_JOB_CONTROL	The_POSIX_JOB_CONTROLvalue
_SC_SAVED_IDS	The_POSIX_SAVED_IDSvalue
_SC_VERSION	The_POSIX_VERSIONvalue
_SC_TIMERS	The_POSIX_TIMERSvalue
_SC_DELAYTIMERS_MAX	Maximumnumberof overruns allowed pertimer
_SC_RTSIG_MAX	Maximumnumberofreal timesignals.
SC_MQ_OPEN_MAX	Maximumnumberofmessagesqueuesperprocess.
_SC_MQ_PRIO_MAX	Maximumpriorityvalueassignable toamessage
SC_SEM_MSEMS_MAX	Maximumnumberofsemaphores perprocess
_SC_SEM_VALUE_MAX	Maximumvalueassignabletoa semaphore.
_SC_SIGQUEUE_MAX	Maximumnumberofreal timesignals thata processmayqueueatanyonetime
_SC_AIO_LISTIO_MAX	Maximumnumberofoperations inonelistio.
_SC_AIO_MAX	Numberof simultaneous asynchronous I/O.

Allconstantsusedasasysconfargumentvaluehavethe_SCprefix.Similarlytheflimit_nameargumentvalueis a manifestedconstantdefinedbythe<unistd.h>header. Theseconstants allhavethe_PC_prefix.

pathconf or fpathconf functions for anamed fileobject.

Limitvalue	*Pathconf*returndata
_PC_CHOWN_RESTRICTED	The_POSIX_CHOWN_RESTRICTEDvalue
_PC_NO_TRUNC	Returns the_POSIX_NO_TRUNC value
_PC_VDISABLE	Returns the_POSIX_VDISABLEvalue
_PC_PATH_MAX	Maximumlengthof apathname(inbytes)
_PC_NAME_MAX	Maximumlengthof afilename(inbytes)
_PC_LINK_MAX	Maximumnumberoflinksa filemayhave
_PC_PIPE_BUF	Maximumsizeof ablock ofdata
_PC_MAX_CANON	maximumsizeof aterminal's canonicalinputqueue
_PC_MAX_INPUT	Maximumcapacityof aterminal's inputqueue.

Program 2

Write a C/C++ POSIX compliant program that prints the POSIX defined configuration options supported on any given system using feature test macros.

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE
#include<iostream>
using namespace std;
#include<unistd.h>

int main()
{
 #ifdef _POSIX_JOB_CONTROL
        cout<<"System supports job control\n";
 #else
        cout<<"System does not support job control\n";
 #endif


 #ifdef _POSIX_SAVED_IDS
       cout<<"System supports saved set-UID and saved set-GID\n";
 #else
      cout<<"System does not support saved set-UID and saved set-GID\n";
 #endif


 #ifdef _POSIX_CHOWN_RESTRICTED
        cout<<" System supports chown_restricted \n" ;
 #else
        cout<<"System does not support chown_restricted \n";
 #endif


  #ifdef _POSIX_NO_TRUNC
        cout<<" System supports posix no trunc  \n";
 #else
        cout<<"System does not support posix no  trunc\n";
 #endif

  #ifdef _POSIX_VDISABLE
        cout<<" System supports Vdisable  \n";
```

```
#else
        cout<<"System does not support Vdisable \n"; ;
#endif

 return 0;
}
```

****

Program 2 output:

****

```
[root@localhost ~]# vi p2.cpp
[root@localhost ~]# g++  p2.cpp
[root@localhost ~]# ./a.out
System supports job control
System supports saved ids
System supports chown restricted
System supports posix no_trunc
System supports vdisable
```

Viva Questions

POSIX STANDARDS:

POSIX: Portable Operating System Interface

1. Why we need POSIX standard?

Because many versions of UNIX exist today and each of them provides its own set of API functions, it is difficult for system developrs to create applications that can be easily ported to different versions of UNIX. Hence POSIX standards are introduced.

SomeofthesubgroupsofPOSIXare**POSIX.1,POSIX.1b&POSIX.1c**

- **POSIX.1**
  - This committee proposes a standard for a base operating system API; this standard specifies APIs for the manipulating of files and processes.
  - It is formally known as IEEE standard 1003.1-1990 and it was also adopted by the ISO as the international standard ISO/IEC 9945:1:1990.
- **POSIX.1b**
  - This committee proposes a set of standard APIs for a real time OS interface; these include IPC (inter- process communication).
  - This standard is formally known as IEEE standard 1003.4-1993.

- **POSIX.1c**

  - This standard specifies multi-threaded programming interface. This is the newest POSIX standard.

  - These standards are proposed for a generic OS that is not necessarily be UNIX system.

  - **E.g.:** VMS from Digital Equipment Corporation, OS/2 from IBM, & Windows NT from Microsoft Corporation are POSIX-compliant, yet they are not UNIX systems.

  - To ensure a user program conforms to POSIX.1 standard, the user should either define the manifested constant _POSIX_SOURCE at the beginning of each source module of the program (before inclusion of any header) as;

```
#define _POSIX_SOURCE
```

Or specify the-D_POSIX_SOURCEoptiontoaC++compiler(CC) in acompilation;

```
%  CC-D_POSIX_SOURCE*.C
```

.

  - POSIX.1b defines different manifested constant to check conformance of user program to that standard. The new macro is _POSIX_C_SOURCE and its value indicates POSIX version to which a user program conforms. Its value can be:

_POSIX_C_SOURCE VALUES	MEANING
198808L	First version of POSIX.1 compliance
199009L	Second version of POSIX.1
199309L	POSIX.1 and POSIX.1b compliance

  - POSIX_C_SOURCE may be used in place of _POSIX_SOURCE. However, some systems that support POSIX.1 only may not accept the _POSIX_C_SOURCE definition.

  - There is also a _POSIX_VERSION constant defined in <unistd.h> header. It contains the POSIX version to which the system conforms.

Program to check and display _POSIX_VERSION constant of the system

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE   199309L
#include<iostream.h>
#include<unistd.h>
int main() {
#ifdef _POSIX_VERSION
cout<<"System conforms to POSIX"<<"_POSIX_VERSION<<endl;
 #else
#endif
}
cout<<"_POSIX_VERSION undefined\n";
return 0;
}
```

The POSIX Feature Test Macros

POSIX.1 defines a set of feature test macro's which if defined on a system, means that the system has implemented the corresponding features. All these test macros are defined in **<unistd.h>** header.

Feature test macro	Effects if defined
_POSIX_JOB_CONTROL	The system supports the BSD style job control.
_POSIX_SAVED_IDS	Each process running on the system keeps the saved set UID and the set-GID, so that they can change its effective user-ID and group-ID to those values via seteuid and setegid API's.
_POSIX_CHOWN_RESTRICTED	If the defined value is -1, users may change ownership of files owned by them, otherwise only users with special privilege may change ownership of any file on the system.
_POSIX_NO_TRUNC	If the defined value is -1, any long pathname passed to an API is silently truncated to NAME_MAX bytes, otherwise error is generated.
_POSIX_VDISABLE	If defined value is -1, there is no disabling character for special characters for all terminal device files. Otherwise the value is the disabling character value.

3. Differentiate relative path from absolute path.

Relative path refers to the path relative to the current path. Absolute path, on the other hand, refers to the exact path as referenced from the root directory.

4.What are the differences among a system call, a library function, and a UNIX command?
A system call is part of the programming for the kernel. A library function is a program that is not part of the kernel but which is available to users of the system. UNIX commands, however, is stand-alone programs; they may incorporate both system calls and library functions in their programming.

5. Is it possible to see information about a process while it is being executed?
Every process is uniquely identified by a process identifier. It is possible to view details and status regarding a process by using the ps command.

Program 3

Consider the last 100 bytes as a region. Write a C/C++ program to check whether the region is locked or not. If the region is locked, print pid of the process which has locked. If the region is not locked, lock the region with an exclusive lock, read the last 50 bytes and unlock the region.

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

int main(int argc,char * argv[])
{
 int fd;
struct flock fLock;
char buf[80]={" "};

if((fd=open(argv[1],O_RDWR))==-1)
{
 perror("open");
return -1;
}

fLock.l_type=F_WRLCK;
fLock.l_whence=SEEK_END;
fLock.l_start= - 100;
fLock.l_len = 100;

while(fcntl(fd,F_SETLK,&fLock)==-1)
{
 while(fcntl(fd,F_GETLK,&fLock)!=1)
 {
    cout<<argv[1]<<"is locked by the process"<<fLock.l_pid<<"from"<<fLock.l_start<<endl;
   cout<<"for"<<fLock.l_len<<"bytes" ;
   cout<<"for"<< ( fLock.l_type = = F_WRLCK ? "write" : "read")<<endl;
 return 1;
 }
}
cout<<"\nLock acquired on file:"<<argv[1]<<"by process with id"<<getpid()<<endl;
 sleep(30);
 lseek(fd,-50,SEEK_END);
```

```
 if(read(fd,buf,50)==-1)
  perror("read error");
 else
 {
        cout<<"\nreading 50 bytes of the file\n";
        cout<<buf;
        cout<<endl;
 }

fLock.l_type=F_UNLCK;
fLock.l_whence=SEEK_SET;
fLock.l_start=0;
fLock.l_len=0;
if(fcntl(fd,F_SETLK,&fLock)==-1)
 perror("fcntl");
else
 cout<<endl<<argv[1]<<"unlocked by process"<<getpid()<<endl;
 return 0;
}
```

****

Program 3 output:

****

[root@localhost ~]# vi p3.cpp
[root@localhost ~]# g++ p3.cpp
[root@localhost ~]# ./a.out
Bad address

[root@localhost ~]# ./a.out filename

Lock acquired on file: filename.cpp by process with id : 8345
Reading 50 bytes of the file :

```
unlocked by process"<<getpid()<<endl;
 return 0;
}
```

filename.cpp unlocked by process : 8345

Viva Questions

1. What is the use of including fcntl.h?
Defines file control options. Header file for fcntl API.

File and Record Locking

- Multiple processes performs read and write operation on the same file concurrently.
- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file. So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- File locking is applicable only for regular files.
- Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.

Write Lock:

- When write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file. The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region, so write lock is termed as "**Exclusive lock**".

Read Lock:

- Similarly when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region. The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region. Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as "**shared lock** ".

Mandatory Locks:

- File lock may be **mandatory** if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region.
- These mechanisms can be used to synchronize reading and writing of shared files by multiple processes.

- If a process locks up a file, other processes that attempt to write to the locked regions are blocked until the former process releases its lock.

- **Problem with mandatory lock is** – if a runaway process sets a mandatory exclusive lock on a file and never

  unlocks it, then, no other process can access the locked region of the file until the runway process is killed or the system has to be rebooted.

- If locks are not mandatory, then it has to be **advisory** lock.

Advisory Locks:

- An advisory lock is not enforced by a kernel at the system call level.

- This means that even though a lock(read or write) may be set on a file, other processes can still use the read and write APIs to access the file.

- To make use of advisory locks, process that manipulate the same file must co-operate such that they follow the given below procedure for every read or write operation to the file.

1. Try to set a lock at the region to be accesses. If this fails, a process can either wait for the lock request to become successful.
2. After a lock is acquired successfully, read or write the locked region.
3. Release the lock.

Demerit of advisory locks are that programs that create processes to share files must follow the above file locking procedure to be co-operative.this may be difficult to contol when programs are obtained from different sources(eg: from different software vendors). All unix and POSIX systems support advisory locks

Lock Promotion:

- If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512, the previous read lock from 0 to 256 is now covered by the write lock and the process does not own two locks on the region from 0 to 256. This process is called "**Lock Promotion**".

Lock Splitting:

- Furthermore, if a process now unblocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called "**Lock Splitting**".

- UNIX systems provide fcntl function to support file locking. By using fcntl it is possible to impose read or write locks on either a region or an entire file.

NOTE: Read all APIs (read,write,open,close,lseek etc) with its arguments and its values

fcntl

- The fcntl function helps a user to query or set flags and the close-on-exec flag of any file descriptor.

- The prototype of fcntl is

  #include<fcntl.h>

  int *fcntl*(int fdesc, int cmd, …);

- The first argument is the file descriptor.

- The second argument cmd specifies what operation has to be performed.

- The third argument is dependent on the actual cmd value.

- The possible cmd values are defined in <fcntl.h> header.

cmd value	Use
F_GETFL	Returns the access control flags of a file descriptor fdesc
F_SETFL	Sets or clears access control flags that are specified in the third argument to fcntl. The allowed access control flags are O_APPEND & O_NONBLOCK
F_GETFD	Returns the close-on-exec flag of a file referenced by fdesc. If a return value is zero, the flag is off; otherwise the return value is nonzero and the flag is on. The close-on-exec flag of newly opened flag is off by default.
F_SETFD	Sets or clears the close-on-exec flag of a fdesc. The third argument to fcntl is an integer value, which is 0 to clear the flag, or 1 to set the flag

F_DUPFD Duplicates file descriptor fdesc with another file descriptor. The third argument to fcntl is an integer value which specifies that the duplicated file descriptor must be greater than or equal to that value. The return value of fcntl is the duplicated file descriptor

If fcntl is used for file locking then it can values :

F_SETLK	sets a file lock, do not block if this cannot succeed immediately.
F_SETLKW	sets a file lock and blocks the process until the lock is acquired.
F_GETLK	queries as to which process locked a specified region of file.

- For file locking purpose, the third argument to fctnl is an address of a *struct flock* type variable. This variable specifies a region of a file where lock is to be set, unset or queried.

```
struct flock
{
short  l_type;   /* what lock to be set or to unlock file */
short  l_whence;  /* Reference address for the next field */
off_t  l_start ;  /*offset from the l_whence reference addr*/
off_t  l_len ;   /*how many bytes in the locked region */
pid_t  l_pid ;    /*pid of a process which has locked the file */
};
```

- The l_type field specifies the lock type to be set or unset.
- The possible values, which are defined in the <fcntl.h> header, and their uses are

l_type value	Use
F_RDLCK	Set a read lock on a specified region
F_WRLCK	Set a write lock on a specified region
F_UNLCK	Unlock a specified region

- The l_whence, l_start & l_len define a region of a file to be locked or unlocked.
- The possible values of l_whence and their uses are

l_whence value	Use
SEEK_CUR	The l_start value is added to current file pointer address
SEEK_SET	The l_start value is added to byte 0 of the file
SEEK_END	The l_start value is added to the end of the file

Program 4

Write a C/C++ program which demonstrates inter-process communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.

NOTE: Type the program in 2 separate files with .cpp extension (one for reader and other for writer) and use g++ compiler to compile the programs.

//READER PROCESS

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

#define FIFO_NAME "My_FIFO"

int main()
{
    char s[300];
    int num, fd;
    cout<<"Waiting for reader process to initilize..."<<endl;
    fd=open(FIFO_NAME, O_RDONLY);
```

```
    cout<<"Got a writer process..."<<endl;
      while(1)
      {
            if((num=read(fd, s, sizeof(s)))==-1)
                  perror("read");
            else
            {
                  if(num==0)
                  {
                        cout<<"Writer process has exited."<<endl;
                        close(fd);
                        return 0;
                  }


                  s[num]='\0';
                  cout<<"Receiver read "<<num<<" bytes: \""<<s<<"\""<<endl;

            }
      }
}
```

//WRITER PROCESS

```
#include <iostream>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

using namespace std;

#define FIFO_NAME "My_FIFO"

int main()
{
	char s[300];
	int num, fd;
	mkfifo(FIFO_NAME, S_IFIFO|S_IRWXU|S_IRWXG|S_IRWXO);
	cout<<"Waiting for reader process to initilize..."<<endl;
	fd=open(FIFO_NAME, O_WRONLY);
	cout<<"Got a reader.\n";
	while(cout<<"Type Something: ", gets(s), !feof(stdin))
	{
		if((num=write(fd, s, strlen(s)))==-1)
			perror("write");
```

```
        else
            cout<<"\tWriter wrote "<<num<<" bytes of data\n";
    }
    close(fd);
    return 0;
}
```

****

Program 4 output:

Note : open two terminals i.e one for reader and other for writer simultaneously.

****

Reader.cpp	**Writer.cpp**
	1. [root@localhost ~]# g++ writer.cpp **2. [root@localhost ~]# ./a.out MY_FIFO** waiting for reader process to initialise….
3. [root@localhost ~]# g++ reader.cpp **4. [root@localhost ~]# ./a.out MY_FIFO** waiting for reader process to initialise…. got a writer process……	

	Got a reader……
	5. Type Something: UNIX
	Writer wrote 4 bytes of data
reader read 4 bytes : " UNIX "	
	Type Something: lab
	Writer wrote 3 bytes of data
reader read 3 bytes : " lab "	
	Type Something: [ctrl+c]
writer process is exited	

Viva Questions

1. **What are IPC Mechanisms?**
   Interprocess communication(IPC) is a mechanism whereby two or more processes communicate with each other to perform/complete tasks. The processes interact in client-server manner or peer-peer fashion.

   The various forms of IPC that are supported on a UNIX system are as follows :

   1) Half duplex Pipes.

   2) FIFO's

   3) Full duplex Pipes.

   4) Named full duplex Pipes.

   5) Message queues.

   6) Shared memory.

   7) Semaphores.

   8) Sockets.

   9) STREAMS.

   The first seven forms of IPC are usually restricted to IPC between processes on the same host. The final two i.e. Sockets and STREAMS are the only two that are generally supported for IPC between processes on different hosts.

   PIPES

   Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.

   - Historically, they have been half duplex (i.e., data flows in only one direction).
   - Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

A pipe is created by calling the pipe function.

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].

FIFO file API's

- FIFO files is a special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer. Fifo files are created with both mkfifo and mknod command.
- Pipes can be used only between related processes when a common ancestor has created the pipe.
- Creating a FIFO is similar to creating a file.
- Indeed the pathname for a FIFO exists in the file system.
- The prototype of mkfifo is

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

#include<sys/stat.h>

#include<unistd.h>
```

- The first argument pathname is the pathname(filename) of a FIFO file to be created.
- The second argument mode specifies the access permission for user, group and others and as well as the S_IFIFO flag to indicate that it is a FIFO file.
- On success it returns 0 and on failure it returns –1.
- **Example**

mkfifo("FIFO5",S_IFIFO | S_IRWXU | S_IRGRP | S_ROTH);

- The above statement creates a FIFO file "FIFO5" with read-write-execute permission for user and only read permission for group and others.

- Once we have created a FIFO using mkfifo, we open it using open.
- Indeed, the normal file I/O functions (read, write, unlink etc) all work with FIFOs.
- When a process opens a FIFO file for reading, the kernel will block the process until there is another process that opens the same file for writing.
- Similarly whenever a process opens a FIFO file write, the kernel will block the process until another process opens the same FIFO for reading.
- This provides a means for synchronization in order to undergo inter-process communication.

NOTE: 1. Read all APIs (read,write,open,close,lseek etc) with its arguments and its values

2. Read all different types of files

Program 5

5 a) Write a C/C++ program that outputs the contents of its Environment list

```
#include<iostream>
using namespace std;
int main(int argc,char *argv[],char *envp[])
{
 int index;
for(index=0;envp[index]!='\0';index++)
{
 cout<<envp[index]<<endl;
}
}
```

****

Program 5a output: ./a.out

****

```
[root@localhost ~]# vi 5a.cpp
[root@localhost ~]# g++  5a.cpp
[root@localhost ~]# ./a.out
NCTUNSHOME=/usr/local/nctuns
SSH_AGENT_PID=2505
HOSTNAME=localhost.localdomain
SHELL=/bin/bash
TERM=xterm
DESKTOP_STARTUP_ID=
```

HISTSIZE=1000

XDG_SESSION_COOKIE=99f05b634e4e55b0fc5035004e607800-1355288651.798130-994245798

GTK_RC_FILES=/etc/gtk/gtkrc:/root/.gtkrc-1.2-gnome2

WINDOWID=54567593

QTDIR=/usr/lib/qt-3.3

NCTUNS_TOOLS=/usr/local/nctuns/tools

QTINC=/usr/lib/qt-3.3/include

GTK_MODULES=gnomebreakpad

NCTUNS_BIN=/usr/local/nctuns/bin

USER=root

http_proxy=http://:8080/

LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:*.exe=00;32:*.com=00;32:*.btm=00;32:*.bat=00;32:*.sh=00;32:*.csh=00;32:*.tar=00;31:*.tgz=00;31:*.arj=00;31:*.taz=00;31:*.lzh=00;31:*.zip=00;31:*.z=00;31:*.Z=00;31:*.gz=00;31:*.bz2=00;31:*.bz=00;31:*.tz=00;31:*.rpm=00;31:*.cpio=00;31:*.jpg=00;35:*.gif=00;35:*.bmp=00;35:*.xbm=00;35:*.xpm=00;35:*.png=00;35:*.tif=00;35:

CCACHE_DIR=/var/cache/ccache

GNOME_KEYRING_SOCKET=/tmp/keyring-nZC4er/socket

SSH_AUTH_SOCK=/tmp/ssh-enCMdV2407/agent.2407

SESSION_MANAGER=local/unix:@/tmp/.ICE-unix/2407,unix/unix:/tmp/.ICE-unix/2407

MAIL=/var/spool/mail/root

PATH=/usr/local/nctuns/bin:/usr/lib/qt-3.3/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/usr/lib/ccache:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/bin

DESKTOP_SESSION=default

GDM_XSERVER_LOCATION=local

INPUTRC=/etc/inputrc

PWD=/root

CCACHE_UMASK=002

LANG=en_US.UTF-8

GNOME_KEYRING_PID=2406

KDE_IS_PRELINKED=1

GDM_LANG=en_US.UTF-8

KDEDIRS=/usr

GDMSESSION=default

SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass

SHLVL=2

HOME=/root

GNOME_DESKTOP_SESSION_ID=Default

no_proxy=localhost,127.0.0.0/8

LOGNAME=root

QTLIB=/usr/lib/qt-3.3/lib

CVS_RSH=ssh

DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-PYaJzn5WHC,guid=f07b431580cb3f250942470050c8104c

XDG_DATA_DIRS=/usr/local/share/:/usr/share/:/usr/share/gdm/

LESSOPEN=|/usr/bin/lesspipe.sh %s

WINDOWPATH=7

DISPLAY=:0.0

G_BROKEN_FILENAMES=1

COLORTERM=gnome-terminal

XAUTHORITY=/tmp/.gdm6D30OW

_=./a.out

Viva Questions

1. What is the use of command 'env'?

It will get a list of name=value pairs. This represents your shell environment. Similarly, a process also has its environment.

2. What are the ways to access a process environment?

Through the global variable 'extern char **extern'. Through the third argument to the main() function char *envp[].

3. Write a program to emulate 'mv' command.

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
int main(int argc, char* argv[])
{
    if(argc!=3||!strcmp(argv[1],argv[2])
    {
      printf("Error usage: %s <oldlink><newlink>\n", argv[0]);
      return 0;
    }

  else if(link(argv[1],argv[2])==0)
   {
        return unlink(argv[1]);
   }
      return -1;
}
```

ENVIRONMENT LIST

The environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string. The address of the array of pointers is contained in the global variable environ:

extern char **environ;

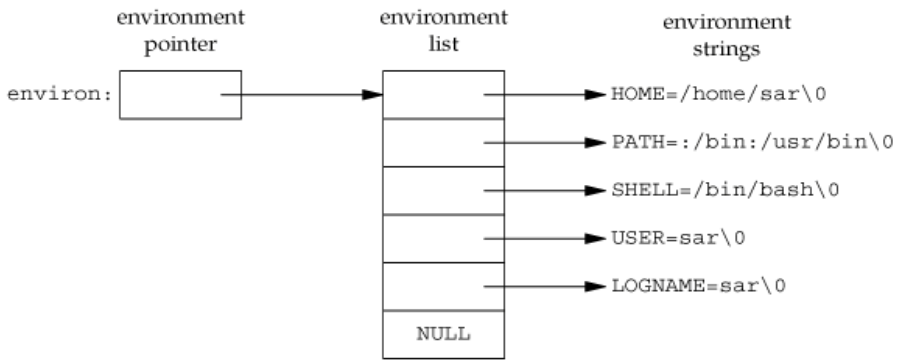

Figure: Environment consisting of five C charactestrings

The environment consists of name=value pairs shown in above fig.

ENVIRONMENT VARIABLES:

The environment strings are usually of the form: ***name=value.***

The functions that we can use to set and fetch values from the variables are setenv, putenv, and getenv functions. The prototype of these functions are

#include <stdlib.h>

char *getenv(const char *name);

Returns: pointer to value associated with name, NULL if not found.

We should always use getenv to fetch a specific value from the environment, instead of accessing environ directly. In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to change the value of an existing variable or add a new variable to the environment. **The prototypes of these functions are:**

#include <stdlib.h>int unsetenv(const char *_name_);

All return: 0 if OK, nonzero on error.

The **putenv** function takes a string of the form **name=value** and places it in the environment list. If name already exists, its old definition is first removed.

The **setenv** function sets name to value. If name already exists in the environment, then

(a) if rewrite is nonzero, the existing definition for name is first removed;

(b) if rewrite is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.

The **unsetenv** function removes any definition of name. It is not an error if such a definition does not exist. Note the difference between **putenv** and **setenv**. Whereas **setenv** must allocate memory to create the **name=value** string from its arguments, **putenv** is free to place the string passed to it directly into the environment.

5 b) Write a C / C++ program to emulate the unix ln command

```
#include<iostream>
#include<string.h>
using namespace std;

int main(int argc,char * argv[])
{
 if(argc!=3 && argc!=4)
 {
     cout<<"error:please specify source file name and destination file name\n";
     return 0;
 }

if(strcmp(argv[1],"-s")!=0)
 {
  if(link(argv[1],argv[2])==-1)
          cout<<"hard link failed\n";
  else
          cout<<"hardlink between"<<argv[1]<<"and"<<argv[2]<<"has been created\n\n";
 }
else
 {
  if(symlink(argv[2],argv[3])==-1)
          cout<<"soft link failed";
  else
          cout<<"symbolic link created:"<<argv[3]<<"-->"<<argv[2]<<"\n";
```

```
 }
}
```

****

Program 5b Output:

****

[root@localhost ~]# vi p5b.cpp

[root@localhost ~]# g++ p5b.cpp

[root@localhost ~]# ./a.out

Error: please specify source filename and destination filename

[root@localhost ~]# ./a.out 1 2

Hardlink between 1 and 2 has been created.

[root@localhost ~]# ./a.out –s 3 4

Symbolic link created:4 - - > 3

[root@localhost ~]# ./a.out 1 2

Hardlink failed

[root@localhost ~]# ./a.out –s 3 4

Softlink failed

Note : io see inode number of file specify: ls - i -l filename

Hard and Symbolic(soft) Links:

- A hard link is a UNIX pathname for a file. Generally most of the UNIX files will be having only one hard link.

- A symbolic link file contains a path name which references another file in either local or a remote file system.

- In order to create a hard link, we use the command **ln**.

  Example : Consider a file /usr/ divya/old, to this we can create a hard link by

  ln /usr/ divya/old /usr/ divya/new

  after this we can refer the file by either /usr/ divya/oldor /usr/ divya/new

- Symbolic link can be creates by the same command ln but with option –s

  Example: ln –s /usr/divya/old /usr/divya/new

- **ln** command differs from the **cp**(copy) command in that **cp** creates a duplicated copy of a file to another file with a different pathname, whereas **ln** command creates a new directory to reference a file.

- Let's visualize the content of a directory file after the execution of command **ln**.

Case 1: for hardlink file

ln /usr/divya/abc /usr/raj/xyz

The content of the directory files /usr/divya and /usr/raj are

Inode number	Filename
90	.
110	..
201	abc
150	xxx

Inode number	Filename
78	.
98	..
100	yyy
201	xyz

/usr/divya **/usr/raj**

Both /usr/divya/abcand /usr/raj/xyzrefer to the same inode number 201, thus type is no new file created.

Case 2: soft link : For the same operation, if ln –s command is used then a new inode will be created.

ln –s /usr/divya/abc /usr/raj/xyz

The content of the directory files divya and raj will be

Inode number	Filename
90	.
110	..
201	abc
150	xxx

/usr/divya

Inode number	Filename
78	.
98	..
100	yyy
450	xyz

/usr/raj

If cp command was used then the data contents will be identical and the 2 files will be separate objects in the file system, whereas in ln –s the data will contain only the path name.

Limitations of hard link:

1. User cannot create hard links for directories, unless he has super-user privileges.

2. User cannot create hard link on a file system that references files on a different file system, because inode number is unique to a file system.

Hard link	Symbolic link
Does not create a new inode.	It creates a new inode
It increases the hard link count of the file	Does not change the had link count of the file
It can t link directory files, unless it is done by superuser	It can link directory files.
It cant link files across different file system	It can link files across different file system
Eg: ln /urs/cse/abc /usr/cse/xyz	Eg: ln -s /urs/cse/abc /usr/cse/xyz

Program 6

Write a C/C++ program to illustrate the race condition.

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

using namespace std;
static void charatatime( char str[] );

int main()
{
    char msg1[]="output is from parent\n";
    char msg2[]="output is from child\n";
        pid_t pid;
        if((pid=fork())<0)
        {
                perror("fork error");
                return -1;
        }
        else if(pid==0)
                charatatime(msg1);
        else
                charatatime(msg2);
        exit(0);
}


static void charatatime( char str[])
{
      char *ptr;
      int i;
      setbuf(stdout,NULL);
      for(ptr=str; (i=*ptr++)!=0;)
           putc(i,stdout);
}
```

****

Program 6 output:

****

[root@localhost ~]# vim p6.cpp

[root@localhost ~]# g++ p6.cpp

[root@localhost ~]# ./a.out

Output is from child

Output is from parent

Explanation:

In some systems the output statements of both parent and the child are overlapped which shows the race condition.

Ex:

OOutput is from child

utput is from parent

OOutput is from parent

Output is from child

Output is from child

Output is from parent

All above examples illustrates race condition (Refer text book for more details)

Viva Questions

1. **What is the use of function setbuf() in unix?**
   Specifies the *buffer* to be used by the *stream* for I/O operations, which becomes a *fully buffered* stream. Or, alternatively, if *buffer* is a null pointer, buffering is disabled for the *stream*, which becomes an *unbuffered* stream. This function should be called once the *stream* has been associated with an open file, but before any input or output operation is performed with it.

2. **What is race condition?**
   A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.

3. **How to avoid race condition?**
   Using various forms of interprocess communication(IPC) and signals we can avoid race condition.

program modification to avoid race condition

```
#include "apue.h"
static void charatatime(char *);
    int main(void)
    {
        pid_t   pid;

+     TELL_WAIT();
+
        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {
+           WAIT_PARENT();      /* parent goes first */
            charatatime("output from child\n");
        } else {
```

```
            charatatime("output from parent\n");
+           TELL_CHILD(pid);
        }
        exit(0);
    }


    static void charatatime(char *str)
    {
        char    *ptr;
        int     c;

        setbuf(stdout, NULL);           /* set unbuffered */
        for (ptr = str; (c = *ptr++) != 0; )
            putc(c, stdout);
    }
```

Program 7

7. Write a C/C++ program that creates a zombie and then calls system to execute the ps command to verify that the process is zombie.

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

using namespace std;

int main()
{
	pid_t pid;
	if((pid=fork())<0)
	{
		perror("fork error");
		return -1;
	}
	else if(pid>0)
	{
	  cout<<"\n parent process id :"<<getpid()<<endl;
	  sleep(2);
	}
	else
	{
	    cout<<"\n child process id :"<<getpid()<<endl;
	      exit(0);
}
cout<<"\n";
system("ps -a -o pid,ppid,s,comm " );
 return 0;
	}
```

****

Program 7 output:

****

[cs@localhost shalini]$ ./a.out

Parent process id : 19057.

Child process id : 19058.

PID	PPID	S	TT	COMMAND
19057	18840	S	pts/0	a.out
19058	19057	Z	pts/0	a.out <defunct>

Explanation:

In the output if the STAT column shows Z or Z+ then the process is a zombie process. Here 19057 is the child process which has Z+ as the STAT conforming it a zombie.

Viva Questions

1. **What is zombie process?**
   A zombie process or defunct process is a process that has completed execution but still has an entry in the process table.

2. **What is an orphan process?**

   An orphan process is a computer process whose parent process has finished or terminated. i.e a process that does not have a parent, called as orphan process. Generally the init process(process id is 1) will become the parent for orphan process.

A process can become orphaned during remote invocation when the client process crashes after making a request of the server.

3. How to kill a zombie process?

We can avoid zombie process by forking twice.

To remove zombies from a system, the SIGCHLD signal can be sent to the parent manually, using the kill command. If the parent process still refuses to reap the zombie, the next step would be to remove the parent process. When a process loses its parent, init becomes its new parent. Init periodically executes the wait system call to reap any zombies with init as parent.

4. What is the function of ps command?

ps - report process status
ps gives a snapshot of the current processes

SIMPLE PROCESS SELECTION

Switch	Description
-A	select all processes
-c	Display the output in columns
-a	select all with a tty except session leaders
-d	select all, but omit session leaders
-e	select all processes

Program 8

8. Write a C/C++ program to avoid zombie process by forking twice.

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

using namespace std;

int main()
{
pid_t pid;

if((pid=fork())<0)
cout<<"fork error";

else if(pid = = 0)
{
   if((pid=fork())<0)
   cout<<"fork error";
   else if(pid>0)
   exit(0);
   sleep(2);
   cout<<"second child, parent pid:"<< getppid()<<endl;
  exit(0);
}

if(waitpid(pid,NULL,0)!=pid)
cout<<"waitpid error";
exit(0);
}
```

****

Program 8 output:

****

```
 [root@localhost ~]# ./a.out
```

second child, parent id = 1

Viva Questions

1. What is the use of fork() system call?

System call **fork()** is used to create processes.

```
#include <unistd.h>

pid_t fork(void);
```

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

There are numerous other properties of the parent that are inherited by the child:

- Real user ID, real group ID, effective user ID, effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

The differences between the parent and child are

- The return value from fork
- The process IDs are different
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change

- The child's tms_utime, tms_stime, tms_cutime, and tms_cstimevalues are set to 0
- File locks set by the parent are not inherited by the child
- Pending alarms are cleared for the child

The set of pending signals for the child is set to the empty set

The two main reasons for forkto fail are

(a) if too many processes are already in the system, which usually means that something else is wrong, or

(b) if the total number of processes for this real user ID exceeds the system's limit.

There are two uses for fork:

- When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers, the parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
- When a process wants to execute a different program. This is common for shells. In this case, the child does an execright after it returns from the fork.

2. What is the function of waitpid() system call?

The **waitpid**() system call suspends execution of the calling process until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

< -1 meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.

-1 meaning wait for any child process.

0 meaning wait for any child process whose process group ID is equal to that of the calling process.

> 0 meaning wait for the child whose process ID is equal to the value of *pid*.

3. What is the function of getppid() system call?

getppid() returns the process ID of the parent of the calling process. **getpid**() returns the process ID of the calling process.

Program 9

9. Write a C/C++ program to implement the system function.

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
using namespace std;

int mysystem(const char *strcmd)
{
	int status;
	pid_t pid;
	if(strcmd==NULL)
		return -1;
	if((pid=fork())<0)
	{
		perror("fork");
		status=-1;
	}
	 else if(pid==0)
	{
	    execl("/bin/sh","sh","-c",strcmd,(char *)0);
	}
	  else
	{
		while(waitpid(pid,&status,0)==-1)
		{
			perror("waitpid");
			return -1;
		}
	}
	return status;
    }


int main(int argc, char *argv[])
{
	int status;
	char cmd[50];
	do
	{     cout<<"myshell> enter any unix command";
		fgets(cmd,20,stdin);
		if((status=mysystem(cmd))<0)
```

```
            cout<<"Error Encountered\n";
       }
     while(strcmp(cmd,"exit"));
      return 0;
}
```

****

Program 9 Output:

****

[root@localhost ~]# vi p9.cpp

[root@localhost ~]# g++ p9.cpp

[root@localhost ~]# ./a.out

myshell> enter any unix command: ls

1.cpp 2.cpp 3.c 5.cpp

myshell> enter any unix command: ctrl +z

Viva Questions

1. **What is the use of execl() system call?**

- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk.

There are 6 exec functions:

```
#include <unistd.h>


int execl(const char *pathname, const char *arg0,... /* (char *)0 */ );

int execv(const char *pathname, char *const argv []);

int execle(const char *pathname, const char *arg0,... /*(char *)0, char *const envp */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv []);
```

2. What is _exit ( )?

Terminate the calling process.

3. What is EINTR?

If a read system call returns a -1 with errno set to EINTR, then it was interrupted by a signal. The default action of every signal is to either be completely ignored or to kill the process. Thus it had to be some signal that was caught.

4. What is the syntax of the following? a. wait b.waitpid c. waitid

```
#include<sys/wait.h>


pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

5.What is EINTR?

If a read system call returns a -1 with errno set to EINTR, then it was interrupted by a signal. The default action of every signal is to either be completely ignored or to kill the process. Thus it had to be some signal that was caught.

SYSTEM FUNCTION:

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

If cmd string is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system. Under the UNIX System, system functionis always available.

Because system is implemented by calling fork, exec, and waitpid, there are three types of return values:

- If either the forkfails or waitpidreturns an error other than EINTR, systemreturns 1 with errnoset to indicate the error.

- If the execfails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).

- Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from systemis the termination status of the shell, in the format specified for waitpid.

Program 10

Write a C/C++ program to set up a real-time clock interval timer using the alarm API.

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

void alarm_handler(int signum)
{
cout<<"\nAlarm time elapsed!!!\n";
}


void func(int dur)
{
 signal(SIGALRM,alarm_handler);
alarm(dur);
pause();
}


int main()
{
int dur;
cout<<"\n Enter duration in seconds:";
cin>>dur;
```

```
func(dur);
return 0;
}
```

****

Program 10 Output:

****

```
[root@localhost ~]# vi p10.cpp
[root@localhost ~]# g++  p10.cpp
[root@localhost ~]# ./a.out

Enter duration in seconds:5
Alarm time alapsed. !!
```

Viva Questions:

Signals:

Signals are software interrupts. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

When a signal is sent to a process, it is pending on the process to handle it. The process can react to pending signals in one of three ways:

- Accept the **default action** of the signal, which for most signals will terminate the process.
- **Ignore** the signal. The signal will be discarded and it has no affect whatsoever on the recipient process.
- Invoke a **user-defined** function. The function is known as a **signal handler routine** and the signal is said to be ***caught*** when this function is called.

The function prototype of the signal API is:

```
#include <signal.h>

Void(*signal (int signal_num, void (*handler) ( int ) ) ) (int);
```

The formal argument of the API are: sig_no is a signal identifier like SIGINT or SIGTERM. The handler argument is the function pointer of a user-defined signal handler function.

SIGINT Interrupt from keyboard

SIGQUIT Quit from keyboard

SIGILL Illegal Instruction

SIGABRT Abort signal from abort

SIGFPE Floating point exception

SIGKILL Kill signal

SIGSEGV Invalid memory reference

SIGPIPE write to pipe with no readers

SIGALRM Alarm timer times out.

SIGTERM Termination signal

SIGCHLD Child stopped or terminated

SIGCONT Continue if stopped

SIGSTOP Stop process execution

SIGTSTP Stop typed at terminal

SIGTTIN Stop background process when it tries to read from controlling terminal

SIGTTOU Stop background process when it tries to Write to its controlling terminal

SIGACTION API:

The sigaction API is called by a process to setup a signal handler method for each

signal it wants to deal with. This function allow us to examine or modify(or both) the action associated with the signal. Futhermore, the sigaction API blocks the signal it is catching allowing a process to specify additional signals to be blocked when the API is handling a signal.

The sigaction API prototype is:

```
#include<signal.h>

int sigaction(int signal_num, struct sigaction* action, struct sigaction* old_action);
```

The struct sigaction data type is defined in the <signal.h> header as:

```
struct sigaction

{

        void        (*sa_handler)(int);

        sigset_t     sa_mask;

        int         sa_flag;

}
```

Sigemptyset:

This function clears all signal flags in the argument**.**

SA__RESTART:

System calls(like open,read etc) interrupted by the signal are automatically restarted.

ALARM FUNCTION:

This function allows us to set a timer that will expire at a specified time in the future. When the timer expires SIGALRM signal is generated.

SIGALRM:

This signal is generated when a timer set with the alarm function expires.

What is the use of following functions?

sigaction()--Examine and Change Signal Action

sigaddset()--Add Signal to Signal Set

sigdelset()--Delete Signal from Signal Set

sigfillset()--Initialize and Fill Signal Set

sigismember()--Test for Signal in Signal Set

sigprocmask()--Examine and Change Blocked Signals

sigpending()--Examine Pending Signals

sigsuspend()--Wait for Signal

sigtimedwait()--Synchronously Accept a Signal for Interval of Time

sigwait()--Synchronously Accept a Signal

sigwaitinfo()--Synchronously Accept a Signal and Signal Data

For C program in UNIX

To open/edit/write a C program	**:**	**vi filename.c**
To compile	**:**	**cc filename.c**
To see output	**:**	**./a.out**

For C++ program in UNIX

To open/edit/write a C++ program	**:**	**vi filename.cpp**
To compile	**:**	**g++ filename.cpp**
To see output	**:**	**./a.out**

For yacc program

To open/edit/write a C++ program	**:**	**vi filename.y**
To compile yacc	**:**	**yacc –d filename.y**
To compile C program with yacc	**:**	**cc y.tab.c -ll**
To see output	**:**	**./a.out**

Program 11

/* 11. Write a C program to implement the syntax-directed definition of "if E then S1" and "if E then S1 else S2" */

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>


int parsecondition(char[],int,char*,int);

void gen(char[],char[],char[],int);


int main()

{

int counter=0,stlen=0,elseflag=0;

char stmt[60];

char strb[50];

char strs1[50];

char strs2[45];

printf("format of if statement\n example---\n");

printf("if (a<b)then (s=a);\n");

printf("if (a<b)then (s=a) else (s=b);\n\n");

printf("enter the statement:\n");

gets(stmt);

stlen=strlen(stmt);
```

```
counter=counter+2;
counter=parsecondition(stmt,counter,strb,stlen);
if(stmt[counter]==')')
counter++;
counter=counter+3;
counter=parsecondition(stmt,counter,strs1,stlen);
if(stmt[counter+1]==';')
{
printf("\n parsing the input statement---");
gen(strb,strs1,strs2, elseflag);
return 0;
}
if(stmt[counter]==')')
counter++;
counter=counter+3;
counter=parsecondition(stmt,counter,strs2,stlen);
counter=counter+2;
if(counter==stlen)
{
elseflag=1;
printf("\n parsing the input statement\n");
gen(strb,strs1,strs2,elseflag);
```

```
return 0;
}
return 0;
}

int parsecondition(char input[],int cntr,char *dest,int totallen)
{
int index=0,pos=0;
while(input[cntr]!='(' && cntr <=totallen)
cntr++;
if(cntr>=totallen)
return 0;
index=cntr;
while(input[cntr]!=')')
cntr++;
if(cntr>=totallen)
return 0;
while(index<=cntr)
dest[pos++]=input[index++];
dest[pos]='\0';
return cntr;
}
```

```
void gen(char b[],char s1[],char s2[],int elsepart)
{
int bt=101,bf=102,sn=103;
printf("\n\t if %s goto %d",b,bt);
printf("\n\t goto %d",bf);
printf("\n %d:",bt);
printf("%s",s1);
if(!elsepart)
printf("\n%d:\n\n",bf);
else
{
printf("\n\t goto %d",sn);
printf("\n %d:%s",bf,s2);
printf("\n %d:\n",sn);
}
```

OUTPUT 1:

```
[cs@localhost ~]$ vi filename.c
[cs@localhost ~]$ cc filename.c
[cs@localhost ~]$ ./a.out
format of if statement
 example---
if (a<b)then (s=a)
if (a<b)then (s=a) else (s=b);

Enter the statement:
if (a<b)then (s=a);
```

```
 parsing the input statement---
          if (a<b) goto 101
          goto 102
 101:(s=a)
 102:
```

OUTPUT 2:

cs@localhost ~]$ ./a.out

format of if statement
example---
if (a<b)then (s=a);
if (a<b)then (s=a) else (s=b);

Enter the statement:
if (a<b)then (s=a) else (s=b);

```
 parsing the input statement

          if (a<b) goto 101
          goto 102
 101:(s=a)
          goto 103
 102:(s=b)
 103:
```

Viva Questions

1. **Write a regular expression to recognize identifier.**

   Ans: A regular expression is a set of pattern matching rules encoded in a string according to certain syntax rules.

2. **Define syntax directed definition/translation.**

   Ans: Syntax-directed translation (SDT) is a method of translating a string into a sequence of actions by attaching one such action to each rule of a grammar.

3. **Define ambiguous grammar?**

   Ans: Any grammar generates more than one parse tree for the same input statement.

4. **When to use intermediate code?**

Ans: To represent the intermediate three address code.

5. Define context free grammar?

Ans: a context-free grammar (CFG) is a formal grammar every production rules of the form $V \rightarrow w$

where V is a *single* nonterminal , and w is a string of terminals and/or nonterminals (w can be empty).

6. Write the three address code for the expression a + a * (b – c) + (b - c) * d

Ans:

t1= b-c
t2=a*t1
t3=t1*d
t4=a+t2
t5=t4+t3

7. What do you mean by optimal intermediate code?

Ans: Optimal code from the feasible set. By reducing the number of instructions.

Program12

/* 12. Write a yacc program that accepts a regular expression as input and produce its parse tree as output */

```
%{
 #include<ctype.h>
char str[20];
int i=0;
%}

%token id
%left  '+'  '-'  '*'  '/'
%%
E:S    { infix_postfix(str); }
S:S '+'T |S '-' T
 |T
T:T '*'F |T '/' F
 |F
F: id | '('S')'
  ;
%%

#include<stdio.h>
main( )
{
printf("\n Enter an identifier:");
yyparse( );
}
```

```
yyerror( )
 {
printf("invalid");
}
yylex( )
 {
char ch=' ';
while(ch!='\n')
 {
  ch=getchar();
str[i++]=ch;
if(isalpha(ch)) return id;
if(ch=='+' || ch=='*' || ch=='-' || ch=='/')
 return ch;
}
str[--i]='\0';
return 0;
exit(0);
}

void push(char stack[],int *top,char ch)
 {
 stack[++(*top)]=ch;
}
char pop(char stack[],int *top)
 {
 return (stack[(*top)--]);
}

int prec(char ch)
 {
```

```
 switch(ch)
  {
     case '/':
     case '*': return 2;
     case '+':
     case '-': return 1;
     case '(': return 0;
     default : return -1;
  }
}

void infix_postfix(char infix[])
 {
   int top=-1,iptr=-1,pptr=-1;
   char postfix[20],stack[20],stksymb,cursymb;
   push(stack, &top,'\0');
   while((cursymb=infix[++iptr])!='\0')
    {
     switch(cursymb)
       {
        case'(' : push(stack,&top,cursymb);
                  break;
        case ')': stksymb = pop(stack,&top);
                  while(stksymb!='(')
                    {
                      postfix[++pptr] = stksymb;
                      stksymb = pop(stack,&top);
                    }
                     break;
        case '*' :
        case '/' :
```

```
 case '+' :
 case '-' : while(prec(stack[top]) >= prec(cursymb))
            postfix[++pptr]=pop(stack,&top);
            push(stack,&top,cursymb);
            break;
default   : if(isalnum(cursymb)==0)
             {
               printf("error in input!");
               exit(0);
             }
            postfix[++pptr]=cursymb;
             }
           }
          while(top!=-1)
          postfix[++pptr]=pop(stack,&top);
          printf("%s \n",postfix);
         }
```

OUTPUT 1:

cs@localhost ~]$ vi filename.y
[cs@localhost ~]$ yacc –d filename.y
[cs@localhost ~]$ cc y.tab.c -ll
[cs@localhost ~]$ ./a.out

Enter an identifier: a+b*c/d
abc*d/+

OUTPUT 2:

Enter an identifier: (a+b)*c/d
ab+c*d/

Viva Questions

1. **Define parse tree.**

   It is an ordered rooted tree that represents the syntactic of a string according to some formal grammar Or Graphical representation of an expression.

2. **Write the notation used in regular expression.**

   Upper case A,B,C , X,Y,Z are used for Non terminals similarly lower case a,b,c,x,y,z are used terminals, Greek letters are used for terminal and non terminals

   .

3. **List the difference between regular expression and production.**

   Regular expression is used to recognize an input string and the production rules that expands nodes in formal grammar.

4. **Write the precedence values used to convert inorder expression to postorder expression.**

Symbol	Input precedence	Stack precedence
+, -	1	2
*, /	3	4
#		-1
Operand	9	0

5. **What is the role lexical analyzer?**

   To generate the tokens from the input file.

6. **What do you mean dependency graph?**

   If an attribute *b* at a node in a parse tree depends on an attribute *c*, then the semantic rule for *b* at that node must be evaluated after the semantic rule that defines *c*. The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a dependency graph.

7. **What is the role of symbol table?**

   To keep the information about variable, array, functions names etc.

8. **What is annotated tree?**

   Graphical representation of a tree by using a formal grammar with return values of each node.

9. **Define DAG.**

   Directed acyclic graph is a directed graph with no direct cycles.

10. **List the different types of parsers.**

    Top down parser and bottom up parser.

Other possible viva questions:

1. **List the difference between parse tree and abstract parse tree.**
2. **Write three address codes for control statement.**
3. **List the different types of addressing statements**
4. **Define three address codes.**
5. **Difference between fork( ) and vfork( )**
6. **Difference between exit( ) and _exit(( )**
7. **What is the use of static and extern keywords?**
8. **How can we terminate a process.**
9. **What are the different types of files and how can we create these files?**
10. **What are merits of UNIX OS?**
11. **What are different environment variables functions?**
12. **Define shell and kernel.**
13. **What is the use of extern and static keyword?**
14. **Define (i)API (ii) process (iii)file descriptor (iv)inode.**
15. **What is the use of static keyword?**