Channabasaveshwara Institute of Technology
(An ISO 9001:2008 certified Institution)
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216.Karnataka.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

LAB MANUAL

(2016 - `17)

10CSL68 UNIX SYSTEMS PROGRAMMING AND COMPILER DESIGN LAB

VI SEMESTER CSE

Name : _____

USN : _____

Batch : _____ Section: _____

UNIX SYSTEMS PROGRAMMING AND COMPILER DESIGN LABORATORY

Subject Code: 10CSL68 **I.A. Marks : 25**
Hours/Week : 03 **Exam Hours: 03**
Total Hours : 42 **Exam Marks: 50**

List of Experiments for USP: Design, develop, and execute the following programs

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

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

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.

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

5. a) Write a C/C++ program that outputs the contents of its Environment list
   b) Write a C / C++ program to emulate the unix **ln** command

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

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.

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

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

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

List of Experiments for Compiler Design: Design, develop, and execute the following programs.

11. Write a C program to implement the syntax-directed definition of "if E then S1" and "if E then S1 else S2". (Refer Fig. 8.23 in the text book prescribed for 06CS62 Compiler Design, Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman: Compilers- Principles, Techniques and Tools, 2nd Edition, Pearson Education, 2007).

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

Note: In the examination *each* student picks one question from the lot of *all* 12 questions.

Channabasaveshwara Institute of Technology
(Accredited by NBA & ISO 9001:2008 certified institution)
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CONTENTS

INTRODUCTION TO UNIX

UNIX: UNIX is an operating system.

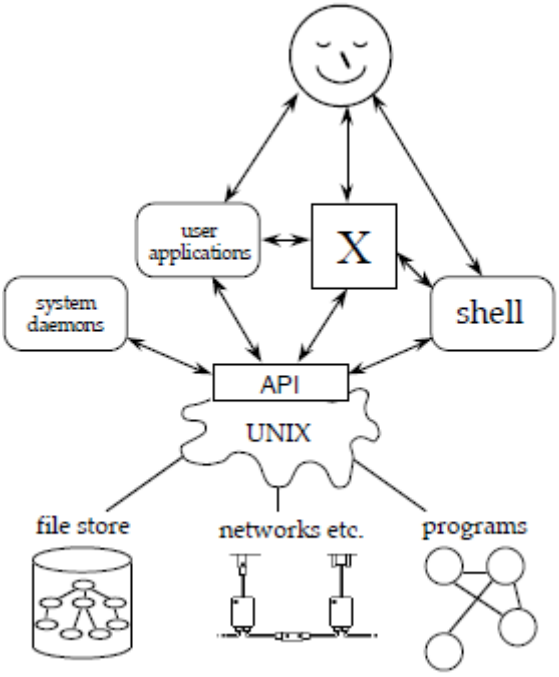


Figure 1: User interaction in UNIX.

It manages:

- Files and data.
- Running programs.
- Networks and other resources.

It is defined by

- Its behavior (on files etc.).
- Its application programmers interface API.
- Ultimately everything works through system calls.
- System calls are executed by the operating system and perform simple single operations.

UNIX System Programming

UNIX system programming means: writing a shell, writing a daemon for your UNIX/UNIX-like OS, writing your version of ps. So it means writing code that heavily depends on the system calls, that does things close related to the UNIX system.

The POSIX standards

- POSIX or "Portable Operating System Interface" is the name of a family of related standards specified by the IEEE to define the application-programming interface (API), along with shell and utilities interface for the software compatible with variants of the UNIX operating system.
- Some of the subgroups of POSIX are POSIX.1, POSIX.1b & POSIX.1c are concerned with the development of set of standards for system developers.

#define _POSIX_SOURCE Or specify the -D_POSIX_SOURCE option to a C++ compiler (CC) in a compilation.

% 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 compliance
199309L	POSIX.1 and POSIX.1b compliance

Sample programs:

1. Program to illustrate the use of cpp(C preprocessor) symbols:

```
#include<stdio.h>
int main()
{
        #if _STDC_==0
        printf("cc is not ANSI C compliant");
        #else
        printf("%s compiled at %s:%s. This statement is at line %d\n", _FILE_ ,
        _DATE_ , _TIME_ , _LINE_ );
        #endif
        return 0;
 }
```

2. Program to check and display _POSIX_VERSION constant of the system on which it is run.

```
#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
        cout<<"_POSIX_VERSION undefined\n";
        #endif return 0;
}
```

EXPERIMENTS FOR USP

Experiment No: 01

AIM:

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

Description:

Limits:

There are three types of limits. They are,

1. Compile-time limits (headers).
2. Runtime limits that are not associated with a file or directory (the sysconf function).
3. Runtime limits that are associated with a file or directory (the pathconf and fpathconf functions).

sysconf, pathconf, and fpathconf Functions:

The runtime limits are obtained by calling one of the following three functions.

```
#include <unistd.h>

long sysconf(int name);

long pathconf(const char *pathname, int name);

Long fpathconf(int filedes, int name);
```

All three return: corresponding value if OK, 1 on error

The difference between the last two functions is that one takes a pathname as its argument and the other takes a file descriptor argument.

Table 1 lists the *name* arguments that sysconf uses to identify system limits. Constants beginning with _SC_ are used as arguments to sysconf to identify the runtime limit. Table 2 lists the *name* arguments that are used by pathconf and fpathconf to identify system limits. Constants beginning with _PC_ are used as arguments to pathconf and fpathconf to identify the runtime limit

Name of limit	Description	*name* argument
CHILD_MAX	maximum number of processes per real user ID	_SC_CHILD_MAX
clock ticks/second	number of clock ticks per second	_SC_CLK_TCK
OPEN_MAX	maximum number of open files per process	_SC_OPEN_MAX
SAVED_IDS	The _POSIX_SAVED_IDS value	_SC_SAVED_IDS

Table 1: Limits and *name* arguments to `sysconf`

Name of limit	Description	*name* argument
CHOWN_RESTRICTED	The _POSIX_CHOWN_RESTRICTED value	_PC_CHOWN_RESTRICTED
NAME_MAX	maximum number of bytes in a filename (does not include a null at end)	_PC_NAME_MAX
PATH_MAX	maximum number of bytes in a relative pathname, including the terminating null	_PC_PATH_MAX

Table 2: Limits and *name* arguments to `pathconf` and `fpathconf`

To Open the Editor with filename

[root@localhost /]# gedit limit.c

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include<stdio.h>
#include<unistd.h>
int main()
{
        int res;
        if((res = sysconf(_SC_OPEN_MAX)) == -1)
            perror("sysconf");
        else
            printf("OPEN_MAX:%d\n",res);
        if((res = pathconf("/", _PC_NAME_MAX)) == -1)
            perror("pathconf");
        else
            printf("max_path name:%d\n",res);
        if((res = sysconf(_SC_CLK_TCK))== -1)
            perror("sysconf");
        else
            printf("clock ticks:%d\n",res);
        if((res = sysconf (_SC_CHILD_MAX)) == -1)
            perror("sysconf");
        else
            printf("max_childs:%d\n",(res));
        if((res = pathconf("/",_PC_PATH_MAX)) == -1)
            perror("pathconf");
        else
            printf("max path name length:%d\n",res);
```

```
        return 0;
}
```

To Run the Program:

```
[root@localhost /]# cc limit.c
[root@localhost /]# ./a.out
```

Output:

```
OPEN_MAX:1024
max_path name:256
clock ticks:100
max_childs:999
max path name length:4096
[root@localhost uspl]#
```

Experiment No: 02

AIM:

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

Description:

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 on as system
_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 respectively.
_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. If this constant is undefined in <unistd.h> header, user must use the pathconf or fpathconf function to check the permission for changing ownership on a per-file basis.
_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. If this constant is undefined in <unistd.h> header, user must use the pathconf or fpathconf function to check the path name truncation option on a per-directory basis.
_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. If this constant is undefined in <unistd.h> header, user must use the pathconf or fpathconf function to check the disabling character option on a per-terminal device file basis.

Table 3: POSIX feature test macros

To Open the Editor with filename

[root@localhost /]# gedit testmacro.c

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include<stdio.h>
#include<unistd.h>
int main()
{
        #ifdef _POSIX_JOB_CONTROL
                printf("System supports job control\n");
        #else
                printf("System does not support job control \n");
        #endif
        #ifdef _POSIX_SAVED_IDS
                printf("System supports saved set-UID and saved set-GID\n");
        #else
                printf("System does not support saved set-UID and saved set-GID \n");
        #endif
        #ifdef _POSIX_CHOWN_RESTRICTED
                printf("chown_restricted option is %d\n",
                _POSIX_CHOWN_RESTRICTED);
        #else
                printf("System does not support chown_restricted option \n");
        #endif
        #ifdef _POSIX_NO_TRUNC
                printf("Pathname trunc option is %d\n",_POSIX_NO_TRUNC);
        #else
                printf("System does not support system-wide pathname trunc option \n");
        #endif
        #ifdef _POSIX_VDISABLE
                printf("Disable character for terminal files is %d\n",
                _POSIX_VDISABLE);
        #else
                printf(" System does not support _POSIX_VDISABLE \n");
        #endif
        return 0;
}
```

To Run the Program:

```
[root@localhost /]# cc testmacro.c
[root@localhost /]# ./a.out
```

Output:

```
System supports job control
System supports saved set-UID and saved set-GID
chown_restricted option is 1
Pathname trunc option is 1
Disable character for terminal files is 0
```

Experiment No: 03

AIM:

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.

Description:

- 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 for regular files.
- 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.

The prototype of fcntl is:

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd_flag, ....);
```

- The first argument specifies the file descriptor for a file to be processed.
- The second argument cmd_flag specifies what operation has to be performed.
- The possible *cmd_flag* values are defined in the <fcntl.h> header. The specific values for file locking and their uses are:

cmd_flag	**Use**
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. The *struct flock* is declared in the <fcntl.h> as:

```
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 l_whence field defines a reference address to which the l_start byte offset value is added. 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

To Open the Editor with filename

[root@localhost /]# gedit lock.cpp

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

using namespace std;

int main(int argc, char *argv[])
{
        int fd,res;
        struct flock fl,fvar;
        char ch,type,fname[20];
        char buf[100];
        cout<<"Enter the file to be locked"<<endl;
        cin>>fname;
        cout<<"CURRENTLY RUNNING PROCESS is :"<< getpid()<<endl;
        fd=open(fname, O_RDWR);
        cout<<"FILE DESCRIPTOR="<<fd;
        if (fd!= -1)
        {
```

```
        /* Make a non-blocking request to place a write lock on last 100 bytes */
        fl.l_pid=getpid();
        fl.l_type = F_WRLCK;
        fl.l_whence = SEEK_END;
        fl.l_start = 0;
        fl.l_len = 100;

        while ((fcntl(fd, F_SETLK, &fl) == -1))
        {
                if (errno == EACCES || errno == EAGAIN)
                {
                        cout<<"Already locked by another process\n";
                                /* We can't get the lock at the moment */
                        (void) fcntl(fd, F_GETLK, &fvar);
                        cout<<"\nFIle Locked by : "<<fvar.l_pid<<endl;
                }
        }
                        /* Lock granted... */
        cout<<"\n\n************LOCKGRANTED**************\n";
        if (lseek(fd,-50,SEEK_END)!=-1)
        {
                cout<<"Located  the file pointer to 50th byte from end of file";
                cout<<"\n File ptr Loc:"<<lseek(fd,(off_t)0,SEEK_CUR)<<endl;
                if (read(fd,&buf,50)!=-1)
                {
                        cout<<"\nContent of FIle [ last 50 bytes ] "<<endl;
                        cout<<"-----------------------------------"<<endl;
                        cout<<buf;
                }
        }//end of lseek
        cout<<"\nPress <RETURN> to UNLOCK: ";
        getchar(); getchar();

        /* Unlock the file */
        fvar.l_type = F_UNLCK;
        fvar.l_pid=getpid();
        fvar.l_whence = SEEK_SET;
        fvar.l_start = 0;
        fvar.l_len=0;

        if(fcntl(fd,F_SETLKW,&fvar)==-1)
                perror("FCNTL UNLOCK ERROR");
        else
                cout<<"\nSuccessfully Unlocked the file\n";
        close(fd);
        //end of if apply lock
}
        else
```

```
                {
                        perror("UNABLE TO open a file ...Please try again");
                        return 0;
                }
} /* main */
```

To Run the Program:

Create a file with filename '**temp'** and type some content with at least 100 bytes.

UNIX system programming means things like these: writing a shell, writing a daemon for your UNIX/UNIX-like OS, writing your version of ps. So it means writing code that heavily depends on the system calls, that does things close related to the UNIX system.

```
[root@localhost /]# g++  lock.cpp
[root@localhost /]# ./a.out
```

Sample Output:

Case 1: If file is not locked

```
Enter the file to be locked
temp
CURRENTLY RUNNING PROCESS is :3860
FILE DESCRIPTOR=3

**********LOCKGRANTED**********
Located  the file pointer to 50th byte from end of file
 File ptr Loc:263

Content of FIle [ last 50 bytes ]
-----------------------------------
hat does things close related to the UNIX system.

Press <RETURN> to UNLOCK:

Successfully Unlocked the file
```

Case 2: If the file is already locked another process

Terminal 1:

```
Enter the file to be locked
temp
CURRENTLY RUNNING PROCESS is :3933
FILE DESCRIPTOR=3

**********LOCKGRANTED**********
```

Located the file pointer to 50th byte from end of file
File ptr Loc:263

Content of FIle [last 50 bytes]
---
hat does things close related to the UNIX system.

Press <RETURN> to UNLOCK:

Terminal 2:(In second terminal try to lock 'temp' file)
[root@localhost /]# g++ lock.cpp
[root@localhost /]# ./a.out
Enter the file to be locked
temp
FIle Locked by : 3933
Already locked by another process

FIle Locked by : 3933
Already locked by another process
.
.
.
FIle Locked by : 3933
Already locked by another process

FIle Locked by : 3933
Already locked by another process

[1]+ Stopped ./a.out
[root@localhost uspl]#

Return to Terminal 1 and unlock the 'temp' file:

Press <RETURN> to UNLOCK:

Successfully Unlocked the file

Experiment No: 04

AIM:

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

Description:

FIFO file:

It 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. The size of the buffer is fixed to PIPE_BUF. Data in the buffer is accessed in a first-in-first-out manner. The buffer is allocated when the first process opens the FIFO file for read or write. The buffer is discarded when all processes close their references (stream pointers) to the FIFO file. Data stored in a FIFO buffer is temporary.

The prototype of mkfifo is:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
int mkfifo(const char *path_name, mode_t mode);
```

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.

open:

This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file. The returned value of the open system call is the file descriptor (row number of the file table), which contains the inode information.

The prototype of open function is,

```
#include<sys/types.h>
#include<sys/fcntl.h>
int open(const char *pathname, int accessmode, mode_t permission);
```

- If successful, open returns a nonnegative integer representing the open file descriptor.
- If unsuccessful, open returns –1.
- The first argument is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.
- If the given pathname is symbolic link, the open function will resolve the symbolic link reference to a non symbolic link file to which it refers.

- The second argument is access modes, which is an integer value that specifies how actually the file should be accessed by the calling process.
- Generally the access modes are specified in <fcntl.h>. Various access modes are:

O_RDONLY - open for reading file only
O_WRONLY - open for writing file only
O_RDWR - opens for reading and writing file.

There are other access modes, which are termed as access modifier flags, and one or more of the following can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file.

O_APPEND - Append data to the end of file.
O_CREAT - Create the file if it doesn't exist
O_EXCL - Generate an error if O_CREAT is also specified and the file already exists.
O_TRUNC - If file exists discard the file content and set the file size to zero bytes.
O_NONBLOCK - Specify subsequent read or write on the file should be non-blocking.
O_NOCTTY - Specify not to use terminal device file as the calling process control terminal.

read:

The read function fetches a fixed size of block of data from a file referenced by a given file descriptor.

The prototype of read function is:

#include<sys/types.h>
#include<unistd.h>
size_t ***read***(**int** fdesc, **void** *buf, **size_t** nbyte);

- If successful, read returns the number of bytes actually read.
- If unsuccessful, read returns –1.
- The first argument is an integer, fdesc that refers to an opened file.
- The second argument, buf is the address of a buffer holding any data read.
- The third argument specifies how many bytes of data are to be read from the file.
- The size_t data type is defined in the <sys/types.h> header and should be the same as unsigned int.
- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.
-

write:

The write system call is used to write data into a file. The write function puts data to a file in the form of fixed block size referred by a given file descriptor.

The prototype of write is

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

ssize_t ***write***(**int** fdesc, **const void** ***buf, **size_t** size);

- If successful, write returns the number of bytes actually written.
- If unsuccessful, write returns –1.
- The first argument, fdesc is an integer that refers to an opened file.
- The second argument, buf is the address of a buffer that contains data to be written.
- The third argument, size specifies how many bytes of data are in the buf argument.
- The return value is usually equal to the number of bytes of data successfully written to a file. (*size* value)

close:

The close system call is used to terminate the connection to a file from a process.

The prototype of the close is

#include<unistd.h>
int ***close***(**int** fdesc);

- If successful, close returns 0.
- If unsuccessful, close returns –1.
- The argument fdesc refers to an opened file.
- Close function frees the unused file descriptors so that they can be reused to reference other files. This is important because a process may open up to OPEN_MAX files at any time and the close function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution.
- The close function de-allocates system resources like file table entry and memory buffer allocated to hold the read/write.

To Open the Editor with filename

[root@localhost /]# gedit interproc.c

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

```
int main(int argc, char* argv[])
{
        int fd;
        char buf[256];
        if(argc != 2 && argc != 3)
        {
                printf("USAGE %s <file> [<arg>]\n",argv[0]);
                return 0;
        }
        mkfifo(argv[1],S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO );
        if(argc == 2)
        {
                fd = open(argv[1], O_RDONLY|O_NONBLOCK);
                while(read(fd, buf, sizeof(buf)) > 0)
                        printf("%s",buf);
        }
        else
        {
                fd = open(argv[1], O_WRONLY);
                write(fd,argv[2],strlen(argv[2]));
        }
        close(fd);
}
```

To Run the Program:

```
[root@localhost /]# cc interproc.c
[root@localhost /]# ./a.out
```

Output:

```
 /* Terminal 1 – writer process */
[root@localhost uspl]#./a.out FIFO1 “This is USP & CD lab”

 After this Open New Terminal by pressing shift+ctrl+N or Go to File->Open Terminal
/* Terminal 2 – reader process */
[root@localhost /]# ./a.out FIFO1
 This is USP & CD lab
```

Experiment No: 05

AIM:

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

Description:

Each program is also passed an environment list. Like the argument 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;

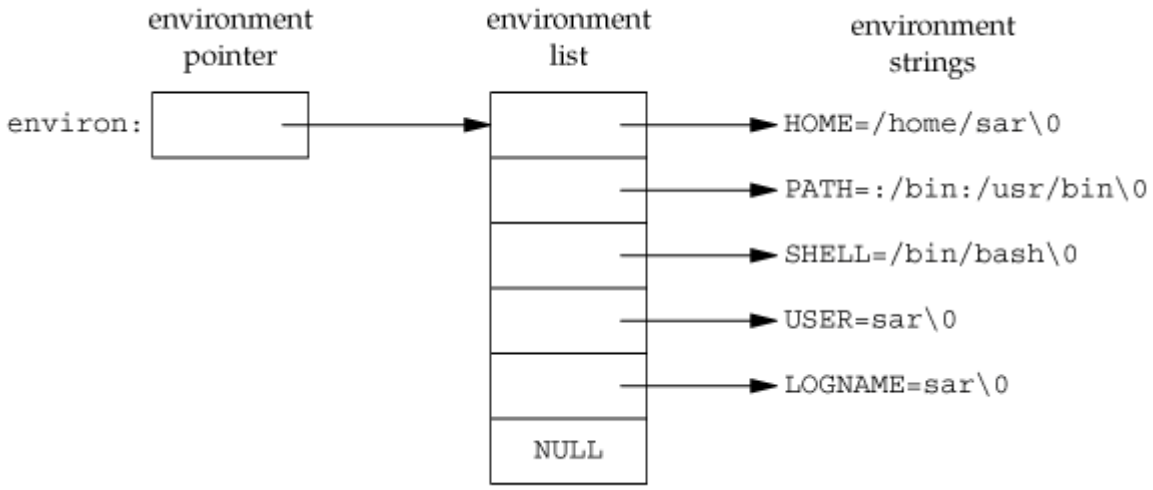


Figure 2: Environment consisting of five C character strings

In figure 3 *environ* is the environment pointer, the array of pointers as environment list, and the strings they point to as environment strings.

By convention the environment consists of *name=value* strings as shown in above figure. Historically, most Unix systems have provided a third argument to the main function that is the address of the environment list:

int main(int argc, char * argv[ ], char *envp[ ]);

Since ANSI C specifies that the main function be written with two arguments, and since this third argument provides no benefit over the global variable environ, POSIX.1 specifies that environ should be used instead of the third argument. Access to specific environment variables is normally through the getenv and putenv functions instead of through the environ variable.

The environment strings are usually of the form: ***name=value.*** The UNIX kernel never looks at these strings; their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some, such as HOME and USER, are set automatically at login, and others are for us to set. We normally set environment variables in a shell start-up file to control the shell's actions. 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.

Note that this function returns a pointer to the value of a **name=value** string. 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 putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
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
  - if rewrite is nonzero, the existing definition for name is first removed;
  - 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.

To Open the Editor with filename

[root@localhost /]# gedit environ.c

```
#include<stdio.h>
int main(int argc, char* argv[ ])
{
        int i;
        char **ptr;
        extern char **environ;
                for( ptr = environ; *ptr != 0; ptr++ )  /*echo all env strings*/
                printf("%s\n", *ptr);
        return 0;
}
```

To Run the Program:

```
[root@localhost /]# cc environ1.c
[root@localhost /]# ./a.out
```

Output:

```
SSH_AGENT_PID=3207
HOSTNAME=localhost.localdomain
DESKTOP_STARTUP_ID=
SHELL=/bin/bash
TERM=xterm
HISTSIZE=1000
KDE_NO_IPV6=1
GTK_RC_FILES=/etc/gtk/gtkrc:/root/.gtkrc-1.2-gnome2
WINDOWID=44040273
OLDPWD=/root/tan
QTDIR=/usr/lib/qt-3.3
QTINC=/usr/lib/qt-3.3/include
USER=root
LS_COLORS=no=00:fi=00:di=00;34:l
GNOME_KEYRING_SOCKET=/tmp/keyring-vsDBVL/socket
SSH_AUTH_SOCK=/tmp/ssh-SEwJHJ3149/agent.3149
KDEDIR=/usr
SESSION_MANAGER=local/localhost.localdomain:/tmp/.ICE-unix/3149
MAIL=/var/spool/mail/root
DESKTOP_SESSION=default
PATH=/usr/lib/qt-
3.3/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sb
in:/usr/bin
GDM_XSERVER_LOCATION=local
INPUTRC=/etc/inputrc
PWD=/root/tan/uspl
XMODIFIERS=@im=none
KDE_IS_PRELINKED=1
LANG=en_US.UTF-8
GDMSESSION=default
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
HOME=/root
SHLVL=2
GNOME_DESKTOP_SESSION_ID=Default
LOGNAME=root
QTLIB=/usr/lib/qt-3.3/lib
CVS_RSH=ssh
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
i0dVljt8MQ,guid=f47759511fe6adb91b249b482809fa00
LESSOPEN=|/usr/bin/lesspipe.sh %s
DISPLAY=:0.0
```

```
G_BROKEN_FILENAMES=1
COLORTERM=gnome-terminal
XAUTHORITY=/tmp/.gdm5X71UW
_=./a.out
```

Note: Output for this program is system dependent

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

Description:

Hard and Symbolic Links
link:

- The link function creates a new hard link for the existing file.

The prototype of the link function is

#include <unistd.h>
int ***link***(**const char** *cur_link, **const char** *new_link);

- If successful, the link function returns 0. If unsuccessful, link returns –1.
- The first argument cur_link, is the pathname of existing file.
- The second argument new_link is a new pathname to be assigned to the same file.
- If this call succeeds, the hard link count will be increased by 1.
- The UNIX ln command is implemented using the link API.

Symlink:

- A symbolic link is created with the symlink function.

The prototype of the symlink function is

#include <unistd.h>
int ***symlink***(**const char** *actualpath , **const char** *sympath);

- If successful, the link function returns 0. If unsuccessful, link returns –1.
- A new directory entry, sympathy, is created that points to actualpath.
- It is not required that actualpath exists when the symboliclink is created.
- Actualpath and sympath need not reside in the same file system.

To Open the Editor with filename

[root@localhost /]# gedit link.c

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<string.h>
int main(int argc, char * argv[])
{
        if(argc < 3 || argc > 4 || (argc == 4 && strcmp(argv[1],"-s")))
        {
                printf("Usage: ./a.out [-s] <org_file> <new_link>\n");
                return 1;
        }
        if(argc == 4)
```

```
        {
                if((symlink(argv[2], argv[3])) == -1)
                        printf("Cannot create symbolic link\n") ;
                else
                        printf("Symbolic link created\n") ;
        }
        else
        {
                if((link(argv[1], argv[2])) == -1)
                        printf("Cannot create  hard link\n") ;
                else
                        printf("Hard link created\n") ;
        }
        return 0;
}
```

To Run the Program:

```
[root@localhost /]# cc link.c
[root@localhost /]# ./a.out
```

Sample Output:

```
Usage: ./a.out [-s] <org_file> <new_link>

[root@localhost uspl]# ./a.out 1 2 3 4
Usage: ./a.out [-s] <org_file> <new_link>

[root@localhost uspl]# ./a.out 1.c z
Hard link created
[root@localhost uspl]# ls -l
-rw-r--r-- 2 root root  657 Mar 27 16:44 1a.c
-rw-r--r-- 2 root root  657 Mar 27 16:44 z          (Bolded columns are hardlink count
                                                    & inode number respectively)
[root@localhost uspl]# ./a.out 1a.c z
Cannot create hard link                             (Because z already exists)
[root@localhost uspl]# ./a.out -s 1a.c zz
Symbolic link created
[root@localhost uspl]# ls -l
-rw-r--r--      2 root root  657 Mar 27 16:44 1a.c
lrwxrwxrwx 1 root root    4  Apr  1   18:32  zz ->
1a.c
[root@localhost uspl]# readlink zz
1a.c
```

Experiment No: 06
AIM:

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

Description:

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. It is also defined as; an execution ordering of concurrent flows that results in undesired behavior is called a race condition-a software defect and frequent source of vulnerabilities.

Race condition is possible in runtime environments, including operating systems that must control access to shared resources, especially through process scheduling.

Avoid Race Condition:

If a process wants to wait for a child to terminate, it must call one of the wait functions. If a process wants to wait for its parent to terminate, a loop of the following form could be used

```
while( getppid() != 1 )
        sleep(1);
```

The problem with this type of loop (called polling) is that it wastes CPU time, since the caller is woken up every second to test the condition. To avoid race conditions and to avoid polling, some form of signaling is required between multi processes.

fork() function:
An existing process can create a new one by calling the fork function.

```
#include <unistd.h>
pid_t fork(void);
```
Returns: 0 in child, process ID of child in parent, 1 on error.

- The new process created by fork is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)
- Both the child and the parent continue executing with the instruction that follows the call to fork.
- The child is a copy of the parent. Example, the child gets a copy of the parent's data space, heap, and stack.

- Note that this is a copy for the child; the parent and the child do not share these portions of memory.
- The parent and the child share the text segment.

To Open the Editor with filename

[root@localhost /]# gedit race.c

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

static void charatatime(char *);

int main(void)
{
        pid_t pid;
        if ((pid = fork()) < 0)
        {
                printf("fork error");
        }
        else if (pid == 0)
        {
                charatatime("output from child\n");
        }
        else
        {
                charatatime("output from parent\n");
         }
        return 0;
 }
 static void charatatime(char *str)
 {
        char *ptr;
        int c;
        setbuf(stdout, NULL); /* set unbuffered */
        for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
 }
```

To Run the Program:

```
[root@localhost /]# cc race.c
[root@localhost /]# ./a.out
```

Output:

```
output from child
```

```
output from parent
[root@localhost /]# ./a.out
output from cohuitlpdu
t from parent
[root@localhost /]# ./a.out
oouuttppuutt ffrroomm pcahrieldnt
```

Experiment No: 07

AIM:

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.

Description:

A zombie process or defunct process is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status. The term zombie process derives from the common definition of zombie - an undead person.

System() function:

It is convenient to execute a command string from within a program. ANSI C defines the system function, but its operation is strongly system dependent. The system function is not defined by POSIX.1 because it is not an interface to the operating system, but really an interface to a shell.

The prototype of system function is:

```
#include <stdlib.h>
int system(const char *cmdstring);
```

It executes a command specified in string by calling /bin/sh -c string, and returns after the command has been completed. During execution of the command, SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored.

If cmdstring 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 is always available. Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

- If either the fork fails or waitpid returns an error other than EINTR, system returns 1 with errno set to indicate the error.
- If the exec fails, 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 system is the termination status of the shell, in the format specified for waitpid.

To Open the Editor with filename

[root@localhost /]# gedit zombie.c

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

```
int main()
{
        pid_t pid;
        pid = fork();
        if(pid == 0)
        {
            exit(0);
        }
        else
            sleep(3);
            system("ps -ax");
        return 0;
}
```

To Run the Program:

```
[root@localhost /]# cc zombie.c
[root@localhost /]# ./a.out
```

Sample Output:

```
PID TTY      STAT   TIME COMMAND
   1 ?        Ss     0:00 init [5]
   2 ?        S      0:00 [migration/0]
   3 ?        SN     0:00 [ksoftirqd/0]
   4 ?        S      0:00 [watchdog/0]
   5 ?        S      0:00 [migration/1]
   6 ?        SN     0:00 [ksoftirqd/1]
   7 ?        S      0:00 [watchdog/1]
  13 ?        S<     0:00 [xenwatch]
  14 ?        S<     0:00 [xenbus]
  17 ?        S<     0:00 [kblockd/0]
  18 ?        S<     0:00 [kblockd/1]
  19 ?        S<     0:00 [kacpid]
 112 ?        S<     0:00 [cqueue/0]
 113 ?        S<     0:00 [cqueue/1]
3379 ?        Sl     0:03 /usr/lib/openoffice.org2.0/program/swriter.bin -write
3390 ?        Ss     0:00 gnome-screensaver
3421 ?        S      0:00 /usr/libexec/notification-daemon
3510 pts/1    S+     0:00 ./a.out
3511 pts/1    Z+     0:00 [a.out] <defunct>
3512 pts/1    R+     0:00 ps –ax
```

Experiment No: 08
AIM:

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

Description:

wait and waitpid Functions:

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event - it can happen at any time while the parent is running - this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. A process that calls wait or waitpid can:

- Block, if all of its children are still running
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- Return immediately with an error, if it doesn't have any child processes.

The prototype of wait and waitpid function is:

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
Both return: process ID if OK, 0 (see later), or 1 on error.
```

The differences between these two functions are as follows.

- The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates. For both functions, the argument statloc is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

To Open the Editor with filename

[root@localhost /]# gedit forktwice.c

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

```
int main()
{
        pid_t pid;

        if ((pid = fork()) < 0)
        {
                printf("fork error");
        }
        else if (pid == 0)
        { /* first child */
                if ((pid = fork()) < 0)
                printf("fork error");
                else if (pid > 0)
                exit(0);
                sleep(2);
                printf("second child, parent pid = %d\n", getppid()); exit(0);
        }
        if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
        printf("waitpid error");
        exit(0);
}
```

To Run the Program:

```
[root@localhost /]# cc forktwice.c
[root@localhost /]# ./a.out
```

Output:

```
 [root@localhost uspl]# second child, parent pid = 1
```

Experiment No: 09

AIM:

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

Description:

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 six 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 [ ]);
```

All six return: -1 on error, no return on success.

- The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a filename argument is specified
- If filename contains a slash, it is taken as a pathname.
- Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.
- The next difference concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.
- The final difference is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

To Open the Editor with filename

[root@localhost /]# gedit system.c

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

int system(const char *cmdstring)
{
        pid_t pid;
        int status;
        if (cmdstring == NULL)
        return(1);
        if ((pid = fork()) < 0)
        {
                status = -1;
        }
        else if (pid == 0)
        {
        /* child */
                execl("/bin/sh", "sh", "-c", cmdstring, (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()
{
        int status;
        if ((status = system("date")) < 0)
                printf("system() error");
        if ((status = system("who")) < 0)
                printf("system() error");
        exit(0);
}
```

To Run the Program:

```
[root@localhost /]# cc system.c
[root@localhost /]# ./a.out
```

Sample Output:

```
Mon Apr  1 18:03:31 IST 2013
root     pts/1        2013-04-01 17:35 (:0.0)
```

Experiment No: 10

AIM:

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

Description:

Alarm() function:

The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. The function prototype of the API is:

#include<signal.h>
unsigned int **alarm**(unsigned int time_interval);

Returns: 0 or number of seconds until previously set alarm

pause() function: It waits for signal.

#include <unistd.h>
int pause(void);

The pause() library function causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function. The pause() function only returns when a signal was caught and the signal-catching function returned. In this case pause() returns -1, and *errno* is set to EINTR.

Sigaction() function:

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);

Returns: 0 if OK, 1 on error 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;
}
```

To Open the Editor with filename

[root@localhost /]# gedit alarm.c

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

void wakeup()
{
        printf("Hello\n");
        printf("---------------------------------\n");
}
int main()
{
        int i;
        struct sigaction action;
        action.sa_handler = wakeup();
        action.sa_flags = SA_RESTART;
        sigemptyset(&action.sa_mask);
        if(sigaction(SIGALRM, &action, 0) == -1)
        {
                perror("sigaction");
        }
        while(1)
        {
                alarm(5);
                pause();
                printf("Waiting for alarm\n");
        }
        return 0;
}
```

To Run the Program:

```
[root@localhost /]# cc alarm.c
[root@localhost /]# ./a.out
```

Sample Output:

```
Hello
---------------------------------
Waiting For Alarm
Hello
--------------------------------- (After 5 CPU Clockcycle)
Waiting For Alarm
```

Hello
---------- (After 5 CPU Clockcycle)
Waiting For Alarm
Hello

Waiting For Alarm
Hello
---------- (After 5 CPU Clockcycle)
Waiting For Alarm
Hello
---------- (After 5 CPU Clockcycle)
Waiting For Alarm

INTRODUCTION TO COMPILER DESIGN

Compiler Design:

Compiler is a System Software that converts High level language to low level language. We human beings can't program in machine language (low level language) understood by computers. In high level language and compiler is the software which bridges the gap between user and computer. It's a very complicated piece of software which took 18 years to build first compiler .This is divided into six phases, they are:

1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate Code Generation
5) Code Optimization
6) Code Generation

Regular Expression:

A regular expression is a specific pattern that provides concise and flexible means to "match" (specify and recognize) strings of text, such as particular characters, words, or patterns of characters. A regular expression provides a grammar for a formal language; this specification can be interpreted by a regular expression processor, which is a program that either serves as a parser generator or examines text and identifies substrings that are members of the specified (again, formal) language.

Derivations:

To check whether a sequence of tokens is legal or not, we start with a nonterminal called the start symbol. We apply productions, rewriting nonterminals, until only terminals remain. A derivation replaces a nonterminal on LHS of a production with RHS.

Leftmost and Rightmost Derivations:

When deriving a sequence of tokens. More than one nonterminal may be present and can be expanded. A leftmost derivation chooses the leftmost nonterminal to expand. A rightmost derivation chooses the rightmost nonterminal to expand.

Parse Tree:

Is a graphical representation for a derivation. It filters out choice regarding replacement order. It is rooted by the start symbol S, interior nodes represent nonterminals in N, leaf nodes are terminals in T or node A can have children X1, X2,...Xn if a rule A->X1, X2... Xn exists.

Syntax Directed Definition:

A Syntax directed definition is a generalization of a context free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol.

An attribute can represent anything we choose: a string, a number, a type, a memory location, or whatever. The value of an attribute at a parse tree node is defined by a semantic rule associated with a production used at that node. The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in

the parse tree; the value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node.

Semantic rules set up dependencies between attributes that will be represented by a graph. From the dependency graph, we derive an evaluation order for the semantic rules. Evaluation of the semantic rules defines the values of the attributes at the nodes in parse tree for the input string.

A parse tree showing the values of attributes at each node is called an annotated parse tree. The process of computing the attributes at the nodes is called annotating or decorating the parse tree.

EXPERIMENTS FOR COMPILER DESIGN

Experiment No: 11

AIM:

Write a C program to implement the syntax-directed definition of "if E then S1" and "if E then S1 else S2". (Refer Fig. 8.23 in the text book prescribed for 06CS62 Compiler Design, Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman: Compilers-Principles, Techniques and Tools, 2nd Edition, Pearson Education, 2007).

To Open the Editor with filename

[root@localhost /]# gedit sdd.c

```
#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[54];
        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");
        scanf("%[^\n]",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");
                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\tgoto %d",Bf);
        printf("\n %d:",Bt);
        printf("%s",S1);
        if(!elsepart)
        printf("\n%d\n",Bf);
        else
        {
                printf("\n\t goto %d",Sn);
                printf("\n %d:%s",Bf,S2);
                printf("\n%d\n",Sn);
        }
}
```

To Run the Program:

```
[root@localhost /]# cc sdd.c
[root@localhost /]# ./a.out
```

Output:

```
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 (x=a);

 parsing the input statement
        if (a<b) goto101
       goto 102
 101:(x=a)
102

[root@localhost uspl]# ./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(x=a) else (x=b);

 parsing the input statement
        if (a<b) goto101
       goto 102
 101:(x=a)
        goto 103
 102:(x=b)
103
```

Experiment No: 12

AIM:

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

Description:

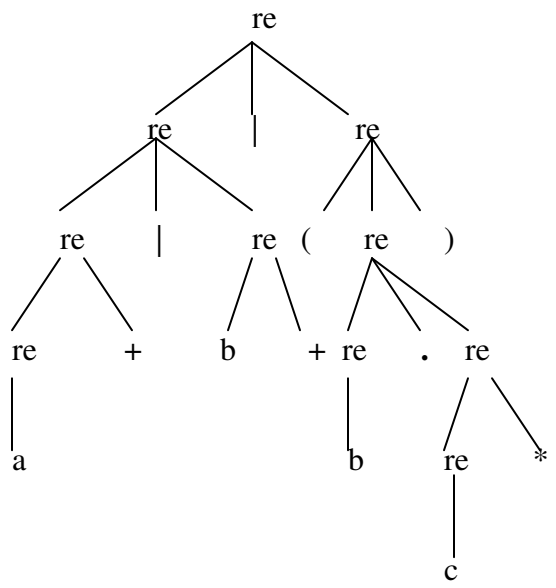
Productions used in this program are:

S -> re
re -> re* | re+ | (re) | re|re | re.re | ALPHABET

Some of the valid regular expressions are:

a+ | b* | (b.c*)
(a.b)* | a+
(a | b*) | a+
(a) | b+

Parse tree for regular expression: a+ | b* | (b.c*)



Some of the invalid regular expressions are:

(ab*) | a+
a* + b*
(a+b)*

To Open the Editor with filename

[root@localhost /]# gedit parse.y

```
%{
  /**
     Yacc program to recognise a regular expression
     and produce a parse tree as output
   */
  #include <stdio.h>
  #include <ctype.h>
  #include <stdlib.h>
  #include <string.h>

  /* To store the productions */
  #define MAX 100
  int getREindex ( const char* );
    signed char productions[MAX][MAX];
  int count = 0 , i , j;
  char temp[MAX + MAX] , temp2[MAX + MAX];
%}
%token ALPHABET
%left '|'
%left '.'
%nonassoc '*' '+'

%%
S : re '\n'    {
  printf ( "This is the rightmost derivation--\n" );
  for ( i = count - 1 ; i >= 0 ; --i ) {
     if ( i == count - 1 ) {
        printf ( "\nre => " );
        strcpy ( temp , productions[i] );
        printf ( "%s" , productions[i] );
     }
     else {
        printf ( "\n   => " );
        j = getREindex ( temp );
        temp[j] = '\0';
        sprintf ( temp2 , "%s%s%s" , temp ,
productions[i] , (temp + j + 2) );
        printf ( "%s" , temp2 );
        strcpy ( temp , temp2 );
     }
  }
  printf ( "\n" );
  exit ( 0 );
}
```

```
re : ALPHABET {
            temp[0] = yylval; temp[1] = '\0';
            strcpy ( productions[count++] , temp );
            }
  | '(' re ')'
      { strcpy ( productions[count++] , "(re)" ); }
  | re '*'
      { strcpy ( productions[count++] , "re*" ); }
  | re '+'
      { strcpy ( productions[count++] , "re+" ); }
  | re '|' re
     {strcpy ( productions[count++] , "re | re" );}
  | re '.' re
     {strcpy ( productions[count++] , "re . re" );}
  ;
%%
int main ( int argc , char **argv )
{
/*
    Parse and output the rightmost derivation,
    from which we can get the parse tree
*/
 yyparse();

 return 0;
}
yylex()
{
 signed char ch = getchar();
 yylval = ch;
 if ( isalpha ( ch ) )
   return ALPHABET;
 return ch;
}
yyerror()
{
 fprintf(stderr , "Invalid Regular Expression!!\n");
 exit ( 1 );
}
int getREindex ( const char *str )
{
 int i = strlen ( str ) - 1;
 for ( ; i >= 0 ; --i ) {
   if ( str[i] == 'e' && str[i-1] == 'r' )
     return i-1;
 }
}
```

To Run the Program:

```
[root@localhost /]# cc y.tab.c -ll
[root@localhost /]# ./a.out
```

Output :

```
a+|b*|(b.c*)
This is the rightmost derivation--

re => re | re
=> re | (re)
=> re | (re . re)
=> re | (re . re*)
=> re | (re . c*)
=> re | (b . c*)
=> re | re | (b . c*)
=> re | re* | (b . c*)
=> re | b* | (b . c*)
=> re+ | b* | (b . c*)
=> a+ | b* | (b . c*)

[root@localhost uspl]# ./a.out
(a.b)*|a+
This is the rightmost derivation--

re => re | re
   => re | re+
   => re | a+
   => re* | a+
   => (re)* | a+
   => (re . re)* | a+
   => (re . b)* | a+
   => (a . b)* | a+
```

VIVA QUESTIONS

1. What are the differences between ANSI C and K & R C?
2. Explain feature test macros. Write a C/C++ POSIX compliant program that is supported on any given system using feature test macros.
3. Explain the common characteristics of API and describe the error status codes.
4. Describe the characteristics of POSIX.1 FIPS standard and X/Open standard.
5. Differentiate between ANSI C and C++ standards.
6. Explain the different file types available in UNIX and POSIX systems.
7. Differentiate between hard links and symbolic links with an example.
8. Describe Inode table entries.
9. Write a C/C++ program to emulate the UNIX mv command.
10. Explain how fcntl API is used for file and record locking.
11. Write the code segment in C that reads utmost 100 bytes into a variable but from standard input.
12. List and explain the access mode flags and access modifier flags. Also explain how the permission value specified in an 'open' call is modified by its calling process 'umask' value.
13. Explain the process of changing user and group ID of files.
14. What are named pipes? Explain with an example the use of lseek, link, access with their prototypes and argument values.
15. Describe the structure of stat system call along with declarations. Also write struct stat structure.
16. Write a C program to implement the UNIX chown functions
17. Explain Open, Creat, read and fcntl APIs with example
18. With an example program, explain the use of setjmp and longjmp functions.
19. Explain how a C program can be started and various ways of termination.
20. Briefly explain the memory layout of a C program
21. What is fork and vfork? Explain with an example program for each.
22. What is a zombie process?
23. What is pipe?
24. Explain wait, waitpid, wait3 and wait4 functions with their prototypes and uses.
25. Explain different exec functions? Describe how their functioning differs from each other.
26. What is Race condition?
27. Write a program that execs an interpreter file.
28. What is process Identifier? Mention the commands for getting different ID's of calling process.

REFERENCES

1. Terrence Chan: UNIX System Programming Using C++, Prentice Hall India, 1999.

2. W. Richard Stevens: Advanced Programming in the UNIX Environment, 2nd Edition, Pearson Education, 2005.