

## UNIX SYSTEMS PROGRAMMING AND COMPILER DESIGN LABORATORY

**Subject Code: 10CSL68**  
**Exam Hours: 03**

**I.A. Marks : 25**  
**Total Hours : 42**

**Hours/Week : 03**  
**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 interposes 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*

**Aim:1**

To 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

**Theory:**

**sysconf** - *Get configuration information at runtime*

**SYNOPSIS**

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

```
long sysconf(int name);
```

**DESCRIPTION**

POSIX allows an application to test at compile or run time whether certain options are supported, or what the value is of certain configurable constants or limits. At compile time this is done by including <unistd.h> and/or <limits.h> and testing the value of certain macros.

At run time, one can ask for numerical values using the present function sysconf(). One can ask for numerical values that may depend on the file system a file is in using the calls fpathconf(3) and pathconf(3). One can ask for string values using confstr(3). The values obtained from these functions are system configuration constants. They do not change during the lifetime of a process.

clock ticks - \_SC\_CLK\_TCK

The number of clock ticks per second. The corresponding variable is obsolete. It was of course called CLK\_TCK.

#### CHILD\_MAX - \_SC\_CHILD\_MAX

The max number of simultaneous processes per user ID. Must not be less than \_POSIX\_CHILD\_MAX (25).

#### OPEN\_MAX - \_SC\_OPEN\_MAX

The maximum number of files that a process can have open at any time. Must not be less than \_POSIX\_OPEN\_MAX (20).

[fpathconf](#), [pathconf](#) - *Get configuration values for files*

#### SYNOPSIS

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

```
long fpathconf(int fd, int name);
```

```
long pathconf(char *path, int name);
```

#### DESCRIPTION

fpathconf() gets a value for the configuration option name for the open file descriptor fd.

pathconf() gets a value for configuration option name for the filename path.

The corresponding macros defined in <unistd.h> are minimum values; if an application wants to take advantage of values which may change, a call to fpathconf() or pathconf() can be made, which may yield more liberal results.

#### \_PC\_PATH\_MAX

returns the maximum length of a relative pathname when path or fd is the current working directory. The corresponding macro is \_POSIX\_PATH\_MAX.

#### \_PC\_NAME\_MAX

returns the maximum length of a filename in the directory path or fd that the process is allowed to create. The corresponding macro is `_POSIX_NAME_MAX`.

**Code:**

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <iostream>
#include <unistd.h>
using namespace std;
int main()
{
    cout<<"No of clock ticks:"<<sysconf(_SC_CLK_TCK)<<endl;
    cout<<"Maximum no of child processes:"<<sysconf(_SC_CHILD_MAX)<<endl;
    cout<<"Maximum path length:"<<pathconf("/",_PC_PATH_MAX)<<endl;
    cout<<"Maximum characters in a file name:"<<pathconf("/",_PC_NAME_MAX)<<endl;
    cout<<"Maximum no of open files:"<<sysconf(_SC_OPEN_MAX)<<endl;
    return 0;
}
```

**Output:**

*Commands for execution:-*

- Open a terminal.
- Change directory to the file location in the terminal.
- Run `g++ usp01.c -o usp01.out` in the terminal.
- If no errors, run `./usp01.out`

## Aim :2

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

## Theory :

POSIX allows an application to test at compile or run time whether certain options are supported, or what the value is of certain configurable constants or limits.

- `_POSIX_SOURCE`:If you define this macro, then the functionality from the POSIX.1 standard (IEEE Standard 1003.1) is available, as well as all of the ISO C facilities.
- `_POSIX_C_SOURCE`:Define this macro to a positive integer to control which POSIX functionality is made available. The greater the value of this macro, the more functionality is made available.
- `_POSIX_JOB_CONTROL`:If this symbol is defined, it indicates that the system supports job control. Otherwise, the implementation behaves as if all processes within a session belong to a single process group. See section Job Control.
- `_POSIX_SAVED_IDS`:If this symbol is defined, it indicates that the system remembers the effective user and group IDs of a process before it executes an executable file with the set-user-ID or set-group-ID bits set, and that explicitly changing the effective user or group IDs back to these values is permitted. If this option is not defined, then if a nonprivileged process changes its effective user or group ID to the real user or group ID of the process, it can't change it back again.
- `_POSIX_CHOWN_RESTRICTED`:If this option is in effect, the chown function is restricted so that the only changes permitted to nonprivileged processes is to change the group owner of a file to either be the effective group ID of the process, or one of its supplementary group IDs.
- `int _POSIX_NO_TRUNC`:If this option is in effect, file name components longer than `NAME_MAX` generate an `ENAMETOOLONG` error. Otherwise, file name components that are too long are silently truncated.
- `_POSIX_VDISABLE`:This option is only meaningful for files that are terminal devices. If it is enabled, then handling for special control characters can be disabled individually.

## Code:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include "iostream"
#include <unistd.h>
```

```

using namespace std;
int main()
{
    #ifdef _POSIX_JOB_CONTROL
        cout<<"System supports POSIX job control:"<<_POSIX_JOB_CONTROL<<endl;
    #else
        cout<<"System does not support POSIX job control"<<endl;
    #endif
    #ifdef _POSIX_SAVED_IDS
        cout<<"System supports saved set UID and GID:"<<_POSIX_SAVED_IDS<<endl;
    #else
        cout<<"System does not support saved set GID and UID"<<endl;
    #endif
    #ifdef _POSIX_CHOWN_RESTRICTED
        cout<<"Chown restricted option is : "<<_POSIX_CHOWN_RESTRICTED<<endl;
    #else
        cout<<"Chown Restricted not defined"<<endl;
    #endif
    #ifdef _POSIX_NO_TRUNC
        cout<<"Truncation option is : "<<_POSIX_NO_TRUNC<<endl;
    #else
        cout<<"Truncation Option not defined"<<endl;
    #endif
    #ifdef _POSIX_VDISABLE
        cout<<"disable char for terminal files"<<_POSIX_VDISABLE<<endl;
    #else
        cout<<"char for terminal device files will not be diasbled"<<endl;
    #endif
    return 0;
}

```

### Output:

- Open a terminal.
- Change directory to the file location in the terminals.
- Open a file using command followed by program\_name

vi 02\_posix\_configuration.cpp

and then enter the source code and save it.

- Then compile the program using

g++ 02\_posix\_configuration.cpp

- If there are no errors after compilation execute the program using

./a.out

**Aim: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.

### Theory:

File locking provides a very simple yet incredibly useful mechanism for coordinating file accesses.

flock - manage locks from shell scripts.

fcntl(used to manipulate the file descriptors) file commands can be used to support record locking, which permits multiple cooperating programs to prevent each other from simultaneously accessing parts of a file in error-prone ways.

fcntl() performs the operations on the open file descriptor fd.

F\_GETLK, F\_SETLK and F\_SETLKW are used to acquire, release, and test for the existence of record locks. F\_UNLK to remove the existing lock.

### Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
int main(int argc, char *argv[])
{
    int fd;
    char buffer[255];
    struct flock fvar;
    if(argc==1)
    {
        printf("usage: %s filename\n", argv[0]);
        return -1;
    }
    if((fd=open(argv[1], O_RDWR))== -1)
    {
        perror("open");
        exit(1);
    }
}
```

```

}
fvar.l_type=F_WRLCK;
fvar.l_whence=SEEK_END;
fvar.l_start=SEEK_END-100;
fvar.l_len=100;
printf("press enter to set lock\n");
getchar();
printf("trying to get lock..\n");
if((fcntl(fd,F_SETLK,&fvar))== -1)
{
    fcntl(fd,F_GETLK,&fvar);
    printf("\nFile already locked by process (pid):  \t%d\n",fvar.l_pid);
    return -1;
}
printf("locked\n");
if((lseek(fd,SEEK_END-50,SEEK_END))== -1)
{
    perror("lseek");
    exit(1);
}
if((read(fd,buffer,100))== -1)
{
    perror("read");
    exit(1);
}
printf("data read from file..\n");
puts(buffer);
printf("press enter to release lock\n");
getchar();
fvar.l_type = F_UNLCK;
fvar.l_whence = SEEK_SET;
fvar.l_start = 0;
fvar.l_len = 0;
if((fcntl(fd,F_UNLCK,&fvar))== -1)
{
    perror("fcntl");
    exit(0);
}
printf("Unlocked\n");
close(fd);
return 0;
}

```

## Output:

*Commands for instructions:*

1. Compile the program

2. Run the program using `./a.out filename`
3. Note: Do not stop execution
4. Open another terminal
5. Run the program using `./a.out filename`

Note: Both the filename's should be same

#### **Aim: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.

#### **Algorithm:**

## Theory:

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. 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.

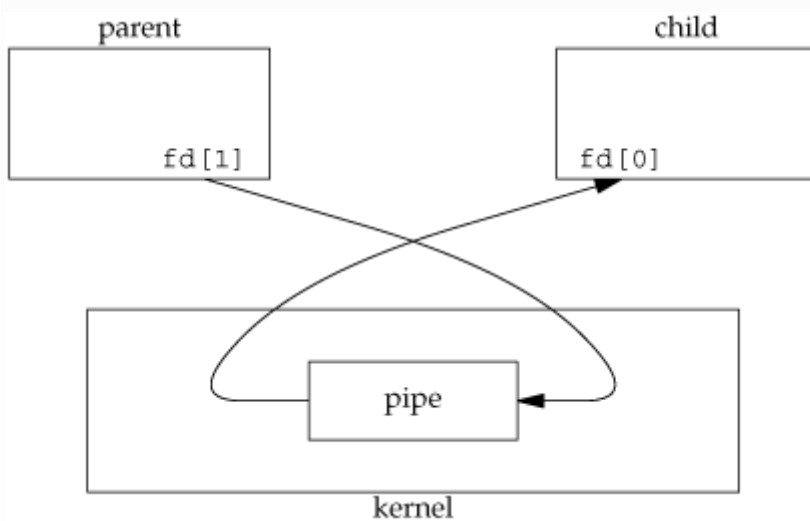
A pipe is created by calling the pipe function.

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

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

Returns: 0 if OK, 1 on error

Figure "Pipe from parent to child"



For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`. When one end of a pipe is closed, the following two rules apply.

- 1.If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read. (Technically, we should say that this end of file is not generated until there are no more writers for the pipe. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing. Normally, however, there is a single reader and a single writer for a pipe. When we get to FIFOs in the next section, we'll see that often there are multiple writers for a single FIFO.)
- 2.If we write to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, write returns 1 with `errno` set to `EPIPE`.

When we're writing to a pipe (or FIFO), the constant `PIPE_BUF` specifies the kernel's pipe buffer size. A write of `PIPE_BUF` bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we write more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers. We can determine the value of `PIPE_BUF` by using `pathconf` or `fpathconf`.

**Code:**

[4read.cpp](#)

```
#include<stdio.h>
#include<stdlib.h>
#include<iostream>
#include<unistd.h>
#include<limits.h>
#include<fcntl.h>
using namespace std;
#define BUFFER_SIZE PIPE_BUF
int main(int argc,char *argv[])
{
    int pipe_fd,res=0;
    char buffer[BUFFER_SIZE+1];
    if(argc!=2)
    {
        cout<<"usage:./a.out pipe_name\n";
```

```

    return -1;
}
cout<<"\nFD of fifo in read mode:"<<pipe_fd<<endl;

if((pipe_fd=open(argv[1],O_RDONLY))!=-1)
{
    res=read(pipe_fd,buffer,BUFFER_SIZE);
    cout<<"\ndata read..\n";
    cout<<buffer;
    close(pipe_fd);
}
else
{
    perror("\nfifo read\n");
}

cout<<"\nprocess "<<getpid()<<" finished reading\n"<<endl;
return 0;
}

```

4write.cpp

```

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

using namespace std;
#define BUFFER_SIZE PIPE_BUF
int main(int argc,char *argv[])
{
    int pipe_fd;
    char buffer[BUFFER_SIZE+1];
    if(argc!=2)
    {
        cout<<"usage:./a.out pipe_name\n";
        return 1;
    }
    if(access(argv[1],F_OK)==-1)
    {
        if(mkfifo(argv[1],0777))
        {
            perror("\nmkfifo error\n");
            exit(0);
        }
    }
}

```

```

}

cout<<"Process "<<getpid()<<"opening fifo in write mode"<<endl;
pipe_fd=open(argv[1],O_WRONLY);
cout<<"FD of fifo in write mode:"<<pipe_fd<<endl;
if(pipe_fd!=-1)
{
    cout<<"enter data\n";
    gets(buffer);
    res=write(pipe_fd,buffer,BUFFER_SIZE);
    if(res==-1)
    {
        perror("write error\n");
        exit(0);
    }
    close(pipe_fd);
}
else
    perror("fifo write");

cout<<"\nprocess "<<getpid()<<" finished writing\n"<<endl;
unlink(argv[1]);
return 0;
}

```

### Output:

*Commands for execution:-*

- Open a terminal.
- Change directory to the file location in the terminal.
- Run "g++ 4read.cpp -o 4read.out" in the terminal.
- If no errors run "g++ 4write.cpp -o 4write.out"
- If no errors, run ./4write.out pipe1"
- Open another terminal without closing the first one
- Change directory into the file location in the terminal
- Run "./4read.out pipe1"
- Shift to the original terminal and enter some string
- The same string should be displayed on the other terminal

### **Aim :5a**

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

### **Theory :**

Environment variables are a set of dynamic named values that can affect the way running processes will behave on a computer.

They are part of the operating environment in which a process runs. For example, a running process can query the value of the TEMP environment variable to discover a suitable location to store temporary files, or the HOME or USERPROFILE variable to find the directory structure owned by the user running the process.

### **Code :**

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(int argc,char *argv[])
{
    int i;
```

```
char **ptr;
extern char **environ;
for(ptr=environ; *ptr; ptr++)
    printf("%s\n", *ptr);
exit(0);
}
```

#### Output:

- Open a terminal. Change directory to the file location in both the terminals.
- Open a file using command followed by program\_name

vi 5a\_environlist.c and then enter the source code and save it.

- Then compile the program using

gcc 5a\_environlist.c

- If there are no errors after compilation execute the program using

./a.out

#### Aim :5b

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

#### Theory :

Links are created by giving alternate names to the original file. The use of links allows a large file, such as a database or mailing list, to be shared by several users without making copies of that file. Not only do links save disk space, but changes made to one file are automatically reflected in all the linked files. The ln command links the file designated in the SourceFile parameter to the file designated by the TargetFile parameter or to the same file name in another directory specified by the TargetDirectory parameter. By default, the ln command creates hard links.

To create a link to a file named chap1, type the following:

ln -f chap1 intro

This links chap1 to the new name, intro. When the -f flag is used, the file name intro is created if it does not already exist. If intro does exist, the file is replaced by a link to chap1. Both the chap1 and intro file names refer to the same file.

To link a file named index to the same name in another directory named manual, type the following:

In index manual

This links index to the new name, manual/index. To link several files to names in another directory, type the following:

In chap2 jim/chap3 /home/manual

This links chap2 to the new name /home/manual/chap2 and jim/chap3 to /home/manual/chap3.

**Code :**

```
#include<stdio.h>
#include<unistd.h>
int main(int argc, char *argv[])
{
    if(argc!=3)
    {
        printf("Usage: %s <src_file><dest_file>\n",argv[0]);
        return 0;
    }
    if(link(argv[1],argv[2])==-1)
    {
        printf("Link Error\n");
        return 1;
    }
    return 0;
}
```

**Output :**

- Open a terminal.
- Change directory to the file location in both the terminals.
- Open a file using command followed by program\_name

```
vi 5b_unix_ln-command.c
```

and then enter the source code and save it.

- Then compile the program using

```
g++ 5b_unix_ln-command.c
```

- Then create a dummy file with any of the name like abc.c .
- If there are no errors after compilation execute the program using

```
./a.out abc.c out.c
```

where abc.c is the source file and out.c is the new destination file to be given.

- Then verify the creation of hard link using

```
ls -l
```

by checking the inode number of both the input files.

**Aim:6**

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

**Algorithm:****Theory:**

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. The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork. In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

**Code:**

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
static void charatotime(char *);
int main()
{
    int pid;
    if((pid=fork())<0)
        printf("fork error\n");
    else if(pid==0)
        charatotime("output from child\n");
    else
        charatotime("output from parent\n");
    _exit(0);
}
```

```

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

```

### Output:

*Commands for execution:-*

- Open a terminal.
- Change directory to the file location in the terminal.
- Run gcc usp06.c -o usp06.out in the terminal.
- If no errors, run ./usp06.out

### Aim:11

Write a C program for a syntax directed definition of a "if E then S1" and "if E then S1 else S2"

### Description:

A SYNTAX-DIRECTED DEFINITION is a context-free grammar in which each grammar symbol X is associated with two finite sets of values: the synthesized attributes of X and the inherited attributes of X, each production  $A \rightarrow \alpha$  is associated with a finite set of expressions of the form

$$b := f(c_1, \dots, c_k)$$

called semantic rules where f is a function and either b is a synthesized attribute of A and the values  $c_1, \dots, c_k$  are attributes of the grammar symbols of  $\alpha$  or A, or b is an inherited attribute of a grammar symbol of  $\alpha$  and the values  $c_1, \dots, c_k$  are attributes of the grammar symbols of  $\alpha$  or A. Each terminal symbol has no inherited attributes.

It is usual to denote the attributes of a grammar symbol in the form X.name where name is an meaningful name for the attribute.

### Algorithm:

1. Start

2. Output the if, if-else statement to the user for reference.
3. Manipulate the input string such that the if and if-else conditions are stored separately.
4. Generate the format of the if, if-else statements and output the same.
5. End.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char input[60],stmt[3][60];
int len,cur,i,j;
void gen()/*used for generation of if, if-else format statements*/
{
    int l1=101,l2=102,l3=103;
    printf("if %s goto %d\n",stmt[0],l1);
    printf("goto %d\n",l2);
    printf("%d:%s\n",l1,stmt[1]);
    if(cur<3)/*if statement*/
        printf("%d:STOP\n",l2);
    else/*if-else statement*/
    {
        printf("goto %d\n",l3);
        printf("%d:%s\n",l2,stmt[2]);
        printf("%d:STOP\n",l3);
    }
}
int main()
{
    printf("Format of if stmt\nExample\n");
    printf("if(a<b)then(s=a);\n");
    printf("if(a<b)then(s=a)else(s=b);\n");
    printf("enter stmt:");
    gets(input);
    len=strlen(input);
    int index=0;
    for(i=0;i<len&&input[i]!=';';i++)
        if(input[i]=='(')
        {
            index=0;
            for(j=i;input[j-1]!=')';j++)
                stmt[cur][index++]=input[j];
            cur++;
            i=j;
        }
    gen();
}
```

```
    return 0;  
}
```

Output:

*Commands for execution:-*

- Open a terminal
- Change the directory to the file location
- Use `gcc filename.c` for compilation
- Run `./a.out` for execution

**Aim: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.

**Theory:**

In unix terminology, a process that has terminated, but whose parent has not yet waited for it, is called a zombie.

*fork()*

Syntax:

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

```
pid_t fork(void);
```

fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent.

*Sleep()* : Delay for a specified amount of time.

*System()*

Syntax:

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

```
int system(const char *command);
```

system() executes a command specified in command by calling /bin/sh -c command, and returns after the command has been completed.

**Code:**

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

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

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

```
int main()
```

```
{
```

```
    int pid;
```

```
    if((pid=fork())<0)
```

```
        printf("fork error\n");
```

```
    else if(pid==0)
```

```
        _exit(0);
```

```
    sleep(2);
```

```
    system( "ps -o pid,ppid,state,TTY,command");
```

```
    _exit(0);
```

```
}
```

**Output:**

- Open a terminal
- Change directory to the file location in the terminal
- Compile the program by using the command `cc usp-lab-07.c -o usp07.out`
- Run `usp07.out`

## Aim:8

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

## Algorithm:

## Theory:

If we want to write a process so that it forks a child but we don't want to wait for the child to complete and we don't want the child to become a zombie until we terminate, the trick is to call fork twice.

We call sleep in the second child to ensure that the first child terminates before printing the parent process ID. After a fork, either the parent or the child can continue executing; we never know which will resume execution first. If we didn't put the second child to sleep, and if it resumed execution after the fork before its parent, the parent process ID that it printed would be that of its parent, not process ID 1.

## Code:

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

int main()
{
    int pid;
    pid = fork();

    if (pid == 0) {
        // First child
        pid = fork();
        if (pid == 0) {
            // Second child
            sleep(1);
            printf("Second child: My parent PID is %d\n", getppid());
        }
    }
    else {
        // Parent process
        wait(NULL);
        sleep(2);
        system("ps -o pid,ppid,state,tty,command");
    }
    return 0;
}
```

**Output:**

*Commands for execution:-*

- Open a terminal.
- Change directory to the file location in the terminal.
- Run "gcc usp06.c -o usp08.out" in the terminal.
- If no errors, run "./usp08.out"

**Aim:9**

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

## Theory:

fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent.

system() executes a command specified in command by calling /bin/sh -c command, and returns after the command has been completed. The exec() family of functions replaces the current process image with a new process image. The execl() function is one among the exec() family of functions. The waitpid() system call suspends execution of the calling process until a child specified by pid argument has changed state.

## Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/wait.h>
void sys(const char *cmdstr)
{
    int pid;
    pid=fork();
    if(pid==0)
        execl("/bin/bash","bash","-c",cmdstr,NULL);
    else
        waitpid(pid,NULL,0);
}
int main(int argc,char *argv[])
{
    int i;
    for(i=1;i< argc;i++)
    {
        sys(argv[i]);
        printf("\n");
    }
    _exit(0);
}
```

## Output:

- Open a terminal
- Change the present working directory to the location where the program exists using the cd command in the terminal

- Compile the program using the command `cc <program name> -o usp09.out`
- run the program using `./usp09.out`

Note: To run use `./a.out command1 command2 ,..., commandn` where each command is a shell command. Example : `./a.out ps date who`

**Aim :10**

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

**Theory :**

- First, every signal has a name. These names all begin with the three characters SIG .For example,SIGABRT is the abort signal that is generated when a process calls the abort function.
- SIGALRM is the alarm signal that is generated when the timer set by the alarm function goes off.
- Use the alarm API for generating a signal after certain time interval as specified by the user.

#### Code :

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<signal.h>
#define INTERVAL 5
void callme(int sig_no)
{
    alarm(INTERVAL);
    printf("Hello!!\n");
}
int main()
{
    struct sigaction action;
    action.sa_handler=(void (*)(int))callme;
    sigaction(SIGALRM,&action,0);
    alarm(2);
    sleep(5);
    return 0;
}
```

#### Output :

1. Open a terminal.
2. Change directory to the file location in both the terminals.
3. Open a file using command followed by program\_name

```
vi 10_alarm_signal_handler.cpp
```

and then enter the source code and save it.

4. Then compile the program using

```
g++ 10_alarm_signal_handler.cpp
```

5. If there are no errors after compilation execute the program using

```
./a.out
```



