

USP LAB

Termwork 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

Theory:

IEEE formed a special task force called POSIX in 1980s to create a set of standards for OS interfacing.

POSIX.1 : standard for base operating system API

POSIX.1b : standard APIs for real time OS interfaces

POSIX.1c : standard for multithreading programming interfaces

POSIX.1 and POSIX.1b define a set of system configuration limits in the form of manifested constants in `<limits.h>` header.

`_POSIX_CHILD_MAX` : min. value = 6

desc: max. no. of child processes that may be created at any time by a process

`_POSIX_OPEN_MAX` : min. value = 16

desc: max. no. of files that may be opened simultaneously by a process

`_POSIX_NAME_MAX` : min. value = 14

desc: max. no. of characters allowed in a filename

To determine runtime limits, we can use functions [prototypes] in `<unistd.h>`

```
long sysconf(const int limit_name);
```

```
long pathconf(const char *pathname, int flimit_name);
```

```
long fpathconf(const int fdesc, int flimit_name);
```

Commands to execute:

```
$ gedit tw1.cpp
```

write your code in this editor given below, save it(Ctrl + s), and exit the editor once done

```
$ g++ tw1.cpp -o tw1
```

```
$ ./tw1
```

Program:

```
#define _POSIX_C_SOURCE 199309L
#include<iostream>
#include<unistd.h>
#include<limits.h>
using namespace std;
int main(){
    int choice,res=0;
    while(1){
        cout<<"1. Compile Time Values\n2. Run Time Values\n3.Exit\nEnter your choice:";
        cin>>choice;
        switch(choice){
            case 1: cout<<"Compile Time Values.\n";
                #ifdef _POSIX_CLK_TCK
                    cout<<"No. of Clock Ticks per second is: "<<_POSIX_CLK_TCK<<endl;
                #else
                    cout<<"_POSIX_CLK_TCK not defined\n";
                #endif
                #ifdef _POSIX_CHILD_MAX
                    cout<<"Max number of child processes at any time is:
"<<_POSIX_CHILD_MAX<<endl;
                #else
                    cout<<"_POSIX_CHILD_MAX not defined\n";
                #endif
                #ifdef _POSIX_PATH_MAX
                    cout<<"Max path name is: "<<_POSIX_PATH_MAX<<endl;
                #else
                    cout<<"_POSIX_PATH_MAX not defined\n";
                #endif
                #ifdef _POSIX_NAME_MAX
                    cout<<"Max no. of characters in file name: "<<_POSIX_NAME_MAX<<endl;
                #else
                    cout<<"_POSIX_NAME_MAX not defined\n";
                #endif
                #ifdef _POSIX_OPEN_MAX
                    cout<<"Max no. of files simultaneously opened is:
"<<_POSIX_OPEN_MAX<<endl;
```

```

        #else
            cout<<"_POSIX_OPEN_MAX not defined\n";
        #endif
        break;
    case 2: cout<<"Run Time Values.\n";
        if((res=sysconf(_SC_CLK_TCK))== -1)
            perror("sysconf");
        else
            cout<<"No. of Clock Ticks per second is: "<<res<<endl;
        if((res=sysconf(_SC_CHILD_MAX))== -1)
            perror("sysconf");
        else
            cout<<"Max number of child processes at any time is: "<<res<<endl;
        if((res=pathconf("/",_PC_PATH_MAX))== -1)
            perror("pathconf");
        else
            cout<<"Max path name is: "<<res<<endl;
        if((res=pathconf("/",_PC_NAME_MAX))== -1)
            perror("pathconf");
        else
            cout<<"Max no. of characters in file name: "<<res<<endl;
        if((res=sysconf(_SC_OPEN_MAX))== -1)
            perror("sysconf");
        else
            cout<<"Max no. of files simultaneously opened is:"<<res<<endl;
        break;
    case 3: exit(0);
    default : cout<<"Invalid Choice.\n";
    }
}
return 0;
}

```

Termwork 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.1 defines a set of feature test macros, which if defined on a system, the system has implemented the corresponding features. These can be found in `<unistd.h>` header.

-POSIX_JOB_CONTROL: System supports BSD style job control

-POSIX_SAVED_IDS: Each process running on system keeps the saved set UIDs and GIDs so that it can change its effective UIDs and GIDs to those values setuid and setgid APIs respectively.

-POSIX_CHOWN_RESTRICTED: if value is -1, users may change the ownerships of files owned by them, else only users with special privileges may change ownership of any file on system.

~~XXXXXXXXXX~~ #ifdef, #else, #endif is used to check these macros.

-POSIX_NO_TRUNC: if value is -1, any long pathname passed to an API is silently truncated to NAME_MAX bytes, else error is generated.

-POSIX_VDISABLE: if value is -1, there's no disabling character for special characters for all terminal device files, else the value is the disabling character value

Commands to execute:

```
$ gedit tw2.cpp
```

write your code in this editor given below, save it (Ctrl + s), and exit the editor once done

```
$ g++ tw2.cpp -o tw2
```

```
$ ./tw2
```

Program:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include<unistd.h>
#include<iostream>
using namespace std;
int main(){

    #ifdef _POSIX_JOB_CONTROL

        cout<<"System Supports Job Control feature"<<endl;

    #else

        cout<<"System does not support job control\n";

    #endif

    #ifdef _POSIX_SAVED_IDS

        cout<<"System Supports saved set-UID and saved set-GID"<<endl;

    #else

        cout<<"System does not support saved set-UID\n";

    #endif

    #ifdef _POSIX_CHOWN_RESTRICTED

        cout<<"System Supports change ownership feature"<<endl;

    #else

        cout<<"System does not support change ownership feature\n";

    #endif

    #ifdef _POSIX_NO_TRUNC

        cout<<"System Supports path truncation option."<<endl;

    #else

        cout<<"System does not support path truncation\n";

    #endif

    #ifdef _POSIX_VDISABLE

        cout<<"System Supports disable character for files."<<endl;

    #else

        cout<<"System does not support disable character\n";

    #endif

    return 0;

}
```

Termwork 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:

UNIX systems allows multiple processes to read and write the same file concurrently. It also renders difficulty for any process in determining when data in a file can be overridden by another processes.

Hence UNIX and POSIX systems supports a file and record locking mechanism which is applicable only for regular files

The `fcntl` API is used for file locking. Prototype of `fcntl` API is:

```
#include <fcntl.h>
```

```
int fcntl(int fdesc, int cmd-flag, ...);
```

`fdesc` is file descriptor of the file to be locked & `cmd-flag` can have values as follows:

`F_SETLK` → Set a file lock, doesn't block if not succeed immediately

`F_SETLKW` → Set a file lock and block the calling process until released

`F_GETLK` → Query as to which process locked

third argument to `fcntl` is address of a struct `flock` type variable.

```
struct flock {
```

```
    short l_type; // what lock to set or to unlock file
```

```
    short l_whence; // 0 reference address for next field
```

```
    off_t l_start; // offset from l_whence reference address
```

```
    off_t l_len; // how many bytes in locked region
```

```
    pid_t l_pid; // pid of the process which has locked the file
```

```
};
```

Commands to execute:

Create an input.txt file using command:

```
$ gedit input.txt
```

add some content more than 100 bytes in this editor, save it(Ctrl + s), and exit the editor once done then execute these commands:

```
$ gedit tw3.c
```

write your code in this editor given below, save it(Ctrl + s), and exit the editor once done

```
$ gcc tw3.c -o tw3
```

```
$ ./tw3 input.txt
```

**open another terminal while file is locked and try to access this file using `./tw3 input.txt` command, same command that's mentioned above. If the terminal says cannot access, file is locked, then your code is working properly.

Program:

```
#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[256];
    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("trying to get lock ... \n");
    if((fcntl(fd, F_SETLK, &fvar)) == -1){

        fcntl(fd, F_GETLK, &fvar);
        printf("\n File already locked by process with pid = %d\n", fvar.l_pid);
        return -1;

    }

    printf("\nLOCKED..\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("\nData Read from he file: ");
    puts(buffer);
    printf("\nPress 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);

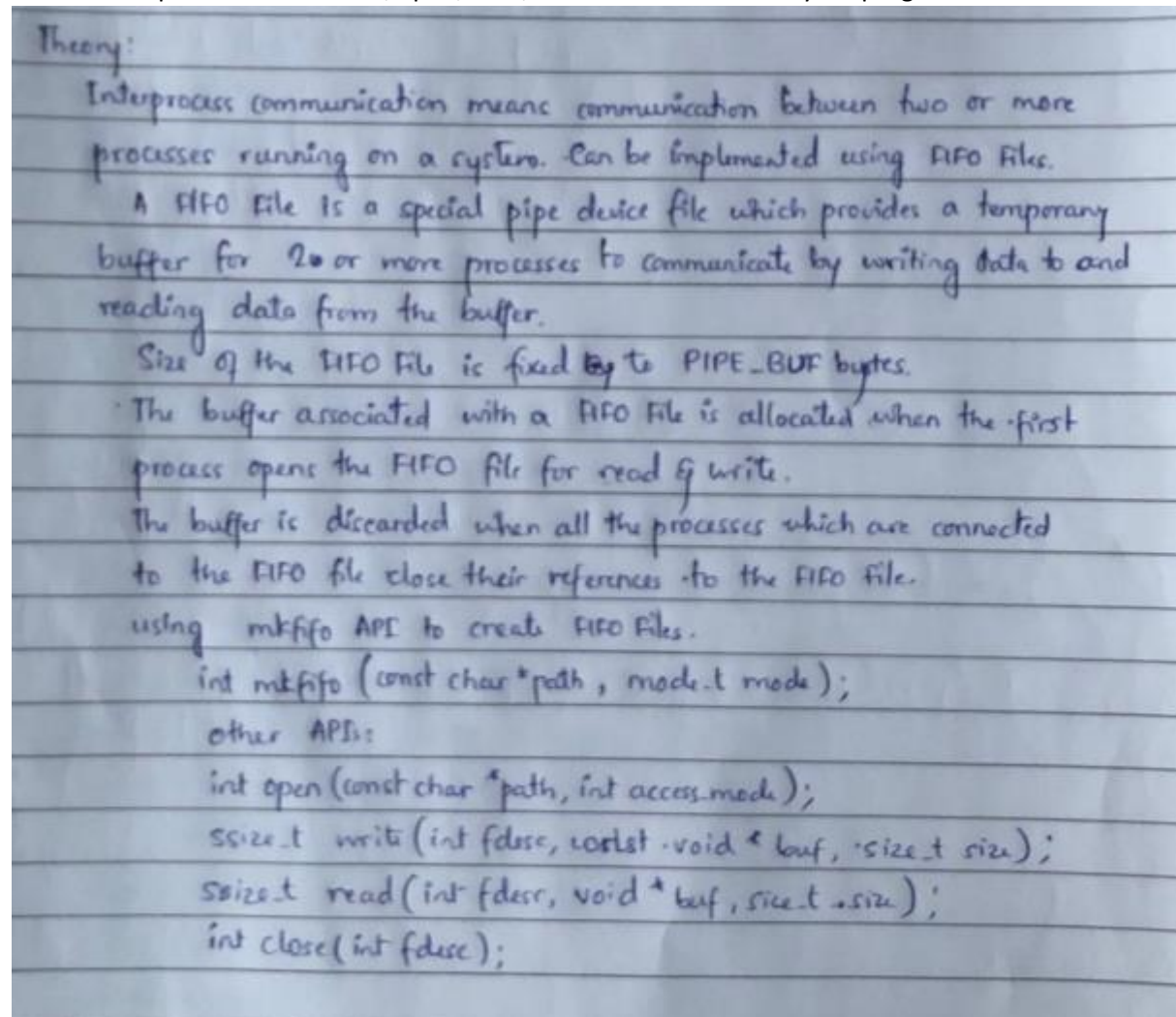
    }

    printf("\nUNLOCKED\n");
    close(fd);
    return 0;

}
```


Termwork 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.



Commands to execute:

```
$ gedit tw4c.c
```

*write your code in this editor given below as **client***, save it(Ctrl + s), and exit the editor once done

```
$ gedit tw4s.c
```

*write your code in this editor given below as **server***, save it(Ctrl + s), and exit the editor once done

```
$ gedit test_file.txt
```

add some content less than 50 bytes in this editor, save it(Ctrl + s), and exit the editor once done

open two terminals:

in first terminal, execute

```
$ gcc tw4c.c -o tw4c
```

in second terminal, execute

```
$ gcc tw4s.c -o tw4s
```

****make sure you run the server program first in second terminal i.e.,

```
$ ./tw4s
```

and then run the client program in first terminal

```
$ ./tw4c
```

request the file created(test_file.txt**) from the first terminal. Content inside file should be printed in **client terminal**.

Program:

Client:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<string.h>
#define FIFO1 "fifo1"
#define FIFO2 "fifo2"
#define PERMS 0666
char fname[256];
int main(){

    ssize_t n;
    char buff[512];
    int readfd, writefd;
    printf("Trying to connect to server..\n");
    writefd = open(FIFO1, O_WRONLY,0);
    readfd = open(FIFO2,O_RDONLY,0);
    printf("Connected..\n");
    printf("Enter the filename to request from server: ");
    scanf("%s", fname);
    write(writefd, fname, strlen(fname)+1);
    printf("Waiting for server to reply...\n");
    while((n=read(readfd,buff,512))>0)

        write(1,buff,n);

    close(readfd);
    close(writefd);
    return 0;
}
```

Server:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<string.h>
#define FIFO1 "fifo1"
#define FIFO2 "fifo2"
#define PERMS 0666
char fname[256];
int main(){

    int readfd,writefd,fd;
    ssize_t n;
    char buff[512];
    if(mkfifo(FIFO1,PERMS)<0)
        printf("Can't create FIFO files\n");
    if(mkfifo(FIFO2, PERMS)<0)
        printf("Can't create FIFO files\n");
    printf("Waiting for connection request.....\n");
    readfd = open(FIFO1, O_RDONLY,0);
    writefd = open(FIFO2, O_WRONLY,0);
    printf("Connection established...\n");
    read(readfd,fname,255);
    printf("Client has requested file %s\n", fname);
    if((fd= open(fname,O_RDWR))<0){
        strcpy(buff,"File does not exist!\n");
        write(writefd,buff,strlen(buff));
    }
    else{
        while(n=read(fd,buff,512)>0)
            write(writefd,buff,n);
    }
    close(readfd);
    unlink(FIFO1);
    close(writefd);
    unlink(FIFO2);

}
```

Termwork 5:

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

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

Theory:

5a: Environment variables have details of environment functionality built into the OS and an environment variable is of the form name = value.

Eg: HOME = /usr/Sagar

These environment variables help programs know what directory to install files in, where to store temporary files, where to find user portal settings, etc.. These variables are null terminated C-strings. can be accessed as follows:

- Using environment list passed to main function as argument
- Using `extern char **environ`
- Using `getenv()` function

5b: **ln** is a link command used to create new links for existing files. Implemented using link API. If successful, hard link count attribute of the file is increased by 1.

A link can be Hard link or Symbolic link.

Hard links cannot be created across files but symbolic links can be used to do so.

Commands to execute:

For Termwork 5a:

```
$ gedit tw5a.c
```

write your code in this editor given below, save it(Ctrl + s), and exit the editor once done

```
$ gcc tw5a.c -o tw5a
```

```
$ ./tw5a
```

For Termwork 5b:

```
$ gedit test_file.txt
```

add some content less than 50 bytes in this editor, save it(Ctrl + s), and exit the editor once done

```
$ gedit tw5b.c
```

write your code in this editor given below, save it(Ctrl + s), and exit the editor once done

```
$ gcc tw5b.c -o tw5b
```

hard link creation:

```
$ ./tw5b test_file.txt testH
```

symbolic link creation:

```
$ ./tw5b -s test_file.txt testS
```

Execute these commands after creating hard and soft links:

```
$ ls -li test_file.txt testH testS
```

this should result in **test_file.txt and **testH** to have same inode numbers and **testS** to point to **test_file.txt**(**testS -> test_file.txt**)

```
$ cat test_file.txt testH testS
```

this command should display your file content the exact same as **test_file.txt by **testH** and **testS**

Program:**5a:**

```
#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);

}
```

5b:

```
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
int main(int argc,char *argv[]){

    if(argc==3){

        printf("Hard Link Created\n");
        return link(argv[1],argv[2]);

    }
    else if(argc == 4){

        if(strcmp(argv[1],"-s") == 0){

            printf("Symbolic link created\n");
            return symlink(argv[2],argv[3]);

        }

        else
            printf("Option must be -s for symbolic link\n");

    }
    else
        printf("Invalid number of arguments\n");

    return 0;

}
```

Termwork 6:

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

Theory:

- Race condition occurs when there are multiple processes accessing / manipulating the shared data / resources and the final outcome depends on the order in which the processes are executed.
- To illustrate race condition, we can use fork system call via. fork API

```
#include <unistd.h>
pid_t fork(void);
```

 - creates a new process called child process.
 - this function returns twice :
 - 1) returns 0 in child process
 - 2) returns pid of child in parent process.
 - if there's an error, it returns -1

Commands to execute:

```
$ gedit tw6.c
```

write your code in this editor given below, save it(Ctrl + s), and exit the editor once done

```
$ gcc tw6.c -o tw6
```

```
$ ./tw6
```

**output should be jumbled

to kill the execution, press **Ctrl + c** together

Program:

```
#define POSIX_SOURCE
#define POSIX_C_SOURCE 199309L
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
static void charatime(char *);
int main(){
    int pid,i;
    for(i=0;i<10;i++){
        if((pid=fork())<0)
            printf("fork error.\n");
        else if(pid==0)
            charatime("output from child\n");
        else
            charatime("output from parent\n");
    }
    return 0;
}

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

Termwork 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:

- A zombie process is the leftover bits of dead processes that haven't been cleaned up properly. We can't kill a zombie process because it's already dead like an actual zombie.
- In UNIX, a zombie process is a process that has completed execution but still has an entry in process table.
- Every process has an entry in process table and these entries are called process control and contains information like process state, memory state, resource state etc.
- These processes simply fill up entries in the process table and as a result, if the table gets completely filled, we cannot run any new processes because no entries can be made in the table.
- We can check if a process is a zombie by executing ps command.

Commands to execute:

```
$ gedit tw7.c
```

write your code in this editor given below, save it(Ctrl + s), and exit the editor once done

```
$ gcc tw7.c -o tw7
```

```
$ ./tw7
```

wait until you see a zombie process with **state Z and **tw7 <defunct>** as command

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#define PS "ps -eo pid,ppid,state,tty,command"
int main(){
    pid_t child_pid;
    if((child_pid=fork())<0)

        perror("fork error");
    else if (child_pid==0)

        exit(0); //child

    sleep(4); // parent
    system(PS);
    return 0;
}
```


Termwork 8:

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

Theory:

- A zombie process is one that has completed its execution but still has an entry in the process table.
- A process table contains all the information that must be saved when CPU switches from process to process in a multitasking system.
- We can use `ps` system call to check whether a process is zombie or not.
- To avoid creation of zombie process:
 - use `wait()` or `waitpid()`
 - parent process can register a `SIGCHLD` handler with `signal()`
 - forking twice.
- How forking works:
 - ~~A process~~ 'A' process creates 'B' process and assigns some work. now 'A' waits till 'B' is completed
 - 'B' creates 'C' child process and assigns a work given by 'A' and terminates
 - termination of 'B' results in:
 - 'C' Becoming orphan and adopted by init process
 - 'A' resumes and entry of 'B' is closed for the process table
 - when 'C' terminates, its entry is closed and 'C' does not become a zombie.

Commands to execute:

```
$ gedit tw8.c
```

write your code in this editor given below, save it(Ctrl + s), and exit the editor once done

```
$ gcc tw8.c -o tw8
```

```
$ ./tw8
```

if you see **NO zombie process with **state Z** and **tw8 <defunct>** as command in your output, then your code is working.

Program:

```
#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;

}
```

Termwork 9:

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

Theory:

- System function is a part of C/C++ standard library.
- It's used to pass the commands that can be executed in the command processor or terminal.

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

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

- To implement system function, we use `execl()` function.
- This function replaces the current process image with a new process image specified by path.

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

```
int execl(const char* path, const char* arg0, ..., const char*  
          argn, NULL);
```

→ path is file to execute.

→ arg0...argn are NULL terminated C-strings which constitute the argument list available to the new process image.

arg0 must point to a filename that's associated with the process being started and cannot be NULL.

Commands to execute:

```
$ gedit tw9.c
```

write your code in this editor given below, save it(Ctrl + s), and exit the editor once done

```
$ gcc tw9.c -o tw9
```

```
$ ./tw9 ls pwd ps whoami who date cal
```

you can put any number of **system commands as arguments to output **./tw9**

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/types.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);
}
```

Termwork 10:

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

Theory:

- An Interval timer can be used to schedule a process to do some tasks at a fixed time interval, to time the execution of some operation or to limit the time allowed for the execution of some tasks
- The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a specified number of real clock seconds.

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

```
unsigned int alarm(unsigned int seconds);
```

- For signal handling we use sigaction API and it allows us to examine or modify the action associated with a particular signal.

```
#include <signal.h>
```

```
int sigaction(int sig-no, const struct sigaction *restrict_act,  
              struct sigaction *restrict_oact);
```

```
struct sigaction {  
    void (*sa_handler)(int);  
    sigset_t sa_mask;  
    int sa_flag;  
};
```

Commands to execute:

```
$ gedit tw10.c
```

write your code in this editor given below, save it(Ctrl + s), and exit the editor once done

```
$ gcc tw10.c -o tw10
```

```
$ ./tw10
```

**after 2 sec, you will see Hello! as output.

Program:

```
#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); //interrupt
    sleep(5);
    return 0;

}
```