WEEK – 7

AIM: Write a program to demonstrate the use of exec functions.

DESCRIPTION:

The exec system call in operating systems is essential for process creation and program execution. It replaces the current process's memory space with a new program, allowing the loading and execution of a different executable within the existing process context. Often used alongside the fork system call, which creates a new process, the exec call enables a child process to adopt a new program, facilitating efficient and flexible process management in operating systems.

- execv(const char *path, char *const argv[]); - The execv function takes an array of pointers to null-terminated strings as arguments.
- execvp(const char *file, char *const argv[]); - Searches for the program specified by file in PATH and replaces the current process image with it, using an array of strings as arguments.
- execvpe(const char *file, char *const argv[], char *const envp[]);: Similar to execvp but allows specifying environment variables with an additional array.
- execl(const char *path, const char *arg0, ..., (char *)0);: Replaces the process image with a new one specified by path, with individual arguments listed, terminated by a null pointer.
- execle(const char *path, const char *arg0, ..., (char *)0, char *const envp[]);: Similar to execl but allows specifying environment variables.
- execlp(const char *file, const char *arg0, ... /*, (char *)0 */);: Replaces the current process image with a new one specified by file, searching in PATH and taking arguments individually.

CODE :

execv

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(){
	pid_t cpid = fork();
	char *args[]={"./fact","./p",NULL};
	if(cpid==-1){
		printf("FOrk failed");
		exit(EXIT_FAILURE);
	}
	else if(cpid==0){
		printf("Process ID of child : %d\n",getpid());
		execv(args[0],args);
	}
	else{
		printf("Process ID of parent : %d\n",getpid());
		execv(args[1],args);
	}
```

```
	printf("ENd of fork system call");
}
```

factorial.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(){
	int n,i,f=1;
	printf("Enter a number:");
	scanf("%d",&n);
	printf("Process ID of child process---executing factorial:%d\n\n",getpid());
	if(n==0 || n==1){
		printf("Factorial of %d is 1",n);
	}
	else if(n<0){
		printf("Factorial doesnt exists for a negative number");
	}
	else{
		for(i=1;i<=n;i++){
			f=f*i;
		}
		printf("factorial of %d is %d",n,f);
	}
}
```

prime.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(){
	int n,i;
	printf("Enter a number:");
	scanf("%d",&n);
	printf("Process ID of parent process -- executing prime number program -
	%d\n\n",getpid());
	int flag=0;
	for (i=2;i<=n/2;i++){
		if (n%i==0){
			flag++;
			break;
		}
	}
	if(flag==0){
		printf("%d is prime number\n\n",n);
	}
	else{
		printf("%d is not a prime number\n\n",n);
	}
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./execv
Process ID of parent : 1369
Process ID of child : 1370
Enter a number:Enter a number:11
Process ID of child process---executing factorial:1370

factorial of 11 is 39916800

11
Process ID of parent process -- executing prime number program - 1369

11 is prime number

sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

CODE :

execvp

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(){
        printf("execvp----\n");
        pid_t cpid = fork();
        char *args[]={"fact",NULL};
        char *args1[]={"p",NULL};
        if(cpid==-1){
                printf("FOrk failed");
        }
        else if(cpid==0){
                printf("Process ID of child : %d\n",getpid());
                execvp("fact", args);
        }
        else{
                printf("Process ID of parent : %d\n",getpid());
                execvp("p", args1);
        }
        printf("ENd of fork system call");
        }
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./execv
Process ID of parent : 1369
Process ID of child : 1370
Enter a number:Enter a number:11
Process ID of child process---executing factorial:1370

factorial of 11 is 39916800

11
Process ID of parent process -- executing prime number program - 1369

11 is prime number

sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ |
```

CODE :

execve

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(){
printf("execve----\n");
pid_t cpid = fork();
char *args[]={"fact",NULL};
char *args1[]={"p",NULL};
char *envp[] = {"VAR=value", NULL};
if(cpid==-1){
printf("FOrk failed");
exit(EXIT_FAILURE);
}
else if(cpid==0){
printf("Process ID of child : %d\n",getpid());
execve(args[0], args, envp);
}
else{
printf("Process ID of parent : %d\n",getpid());
execve(args1[0], args1, envp);
}
printf("ENd of fork system call");}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./execv
Process ID of parent : 1369
Process ID of child : 1370
Enter a number:Enter a number:11
Process ID of child process---executing factorial:1370

factorial of 11 is 39916800

11
Process ID of parent process -- executing prime number program - 1369

11 is prime number

sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ |
```

<u>CODE :</u>

execl

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(){
pid_t cpid = fork();
char *file1 = "./fact";
char *file2 = "./p";
char *arg1 = "Hello world!";
if(cpid==-1){
printf("FOrk failed");
}
else if(cpid==0){
printf("Process ID of child : %d\n",getpid());
execl(file1, arg1, NULL);
}
else{
printf("Process ID of parent : %d\n",getpid());
execl(file2, arg1,NULL);}
printf("ENd of fork system call");
}
```

<u>OUTPUT :</u>

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./execv
Process ID of parent : 1369
Process ID of child : 1370
Enter a number:Enter a number:11
Process ID of child process---executing factorial:1370

factorial of 11 is 39916800

11
Process ID of parent process -- executing prime number program - 1369

11 is prime number

sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

<u>CODE :</u>

execlp

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(){
printf("execlp----\n");
pid_t cpid = fork();
char *file1 = "fact";
char *file2 = "p";
char *arg1 = "Hello world!";
```

```
if(cpid==-1){
printf("FOrk failed");
}
else if(cpid==0){
printf("Process ID of child : %d\n",getpid());
execl(file1, arg1, NULL);
}
else{
printf("Process ID of parent : %d\n",getpid());
execl(file2, arg1,NULL);
}
printf("ENd of fork system call");
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./execv
Process ID of parent : 1369
Process ID of child : 1370
Enter a number:Enter a number:11
Process ID of child process---executing factorial:1370

factorial of 11 is 39916800

11
Process ID of parent process -- executing prime number program - 1369

11 is prime number

sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

CODE :

execle

```
#include <unistd.h>
#include<stdlib.h>
#include<stdio.h>
int main(void) {
char *file = "/usr/bin/bash";
char *arg1 = "-c";
char *arg2 = "echo $ENV1 $ENV2!";
char *const env[] = {"ENV1=Hello", "ENV2=World", NULL};
printf("execle----\n");
pid_t cpid = fork();
char *file1 = "./fact";
char *file2 = "./p";
if(cpid==-1){
printf("FOrk failed");
}
else if(cpid==0){
printf("Process ID of child : %d\n",getpid());
execle(file1,file, arg1, NULL,env);
}
```

```
else{
printf("Process ID of parent : %d\n",getpid());
execle(file2,file, arg1,NULL,env);
}
printf("ENd of fork system call");
        return 0;
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./execv
Process ID of parent : 1369
Process ID of child : 1370
Enter a number:Enter a number:11
Process ID of child process---executing factorial:1370

factorial of 11 is 39916800

11
Process ID of parent process -- executing prime number program - 1369

11 is prime number

sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ |
```

AIM : Write a program to demonstrate threads.

DESCRIPTION:

Threads in an OS are lightweight, independent units within a process, sharing resources like memory but having separate program counters. They enable parallelism and concurrency, executing tasks simultaneously. Commonly used to optimize multi-core processors, threads enhance efficiency by running on separate cores. While thread creation incurs overhead, the advantages include faster communication, data sharing, and improved system responsiveness and resource utilization.

ALGORITHM :

1. Initialize Shared Variable: Set shared to 1.
2. Main Function:
    - Print main thread process ID using getpid().
    - Create thread1 and thread2 using pthread_create.
    - Wait for both threads to finish using pthread_join.
    - Print a message indicating main thread execution.
3. Thread Functions (fun1 and fun2):
    - Print thread entry message.
    - Print thread-specific process ID using getpid().
    - Execute a loop (five iterations) in each thread.
    - Print thread-specific information.
    - Sleep for one second between iterations.

CODE :

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
void *fun1();
void *fun2();
int shared = 1;
int main(){
pthread_t thread1, thread2;
printf("Main thread process ID %d\n", getpid());
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
printf("In main thread\n");
}
void *fun1(){
        printf("Inside thread 1 with process ID: %d\n", getpid());
        for(int i=0; i<5;i++){
                printf("Thread one I-i = %d, Shared: %d\n", i, shared++);
                sleep(1);}
}
void *fun2(){
        printf("Inside thread 2 with process ID: %d\n", getpid());
        for(int j=0; j<5;j++){
                printf("Thread two II-j = %d, Shared: %d\n", j, shared++);
                sleep(1);}
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./thread
Main thread process ID 1503
Inside thread 1 with process ID: 1503
Thread one I-i = 0, Shared: 1
Inside thread 2 with process ID: 1503
Thread two II-j = 0, Shared: 2
Thread one I-i = 1, Shared: 3
Thread two II-j = 1, Shared: 4
Thread one I-i = 2, Shared: 5
Thread two II-j = 2, Shared: 6
Thread one I-i = 3, Shared: 7
Thread two II-j = 3, Shared: 8
Thread one I-i = 4, Shared: 9
Thread two II-j = 4, Shared: 10
In main thread
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

AIM : Write a program to demonstrate paging.

DESCRIPTION :

Paging is a memory management approach dividing physical memory into fixed-size pages and logical memory into corresponding frames. This allows flexible memory allocation. During process execution, logical address space is segmented into pages mapped to physical memory frames. Paging minimizes external fragmentation by supporting non-contiguous memory allocation. It enables virtual memory implementation, allowing parts of a process to reside in secondary storage and be brought into main memory as needed. Page tables track logical-to-physical address mapping, triggering page faults when needed pages are absent, and facilitating efficient memory use in modern operating systems.

ALGORITHM :

1. Initialization:
   - Divide physical memory into fixed-size frames.
   - Divide logical memory into fixed-size pages.
   - Create a page table to map logical pages to physical frames.

2.Page Table Handling:
   - Initialize page table entries.
   - Update page table during page faults.
   - Translate logical addresses to physical addresses using the page table.

3. Page Fault Handling:
   - Identify required page.
   - Replace a page in case of a page fault.
   - Load the required page into the freed frame.

4. Memory Access:
   - Translate logical addresses via the page table.
   - Access data in the physical memory location.

CODE :

```
#include<stdio.h>
#define MAX 50
int main()
{
int page[MAX],i,n,f,ps,off,pno;
int choice=0;
printf("\nEnter the no of pages in memory: ");
scanf("%d",&n);
printf("Enter page size: ");
scanf("%d",&ps);
printf("Enter no of frames: ");
scanf("%d",&f);
for(i=0;i<n;i++)
page[i]=-1;
printf("\nEnter the page table\n");
```

```
printf("(Enter frame no as -1 if that page is not present in any frame)\n\n");
printf("\npageno\tframeno\n-------\t-------");
for(i=0;i<n;i++)
{
printf("\n\n%d\t\t",i);
scanf("%d",&page[i]);
}
do
{
printf("\nEnter the logical address(i.e,page no & offset):");
scanf("%d%d",&pno,&off);
if(page[pno]==-1)
printf("\nThe required page is not available in any of frames");
else
printf("\nPhysical address(i.e,frame no & offset):%d,%d",page[pno],off);
printf("\nDo you want to continue(1/0)?:");
scanf("%d",&choice);
}while(choice==1);
return 1;
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ vi paging.c
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ gcc paging.c -o paging
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./paging
Enter the no of pages in memory: 7
Enter page size: 3
Enter no of frames: 3

Enter the page table
(Enter frame no as -1 if that page is not present in any frame)


pageno  frameno
------- -------

0               4

1               -1

2               2

3               3

4               0

5               1

6               3
Enter the logical address(i.e,page no & offset):1 200

The required page is not available in any of frames
Do you want to continue(1/0)?:1

Enter the logical address(i.e,page no & offset):3 300

Physical address(i.e,frame no & offset):3,300
Do you want to continue(1/0)?:1

Enter the logical address(i.e,page no & offset):0 400

Physical address(i.e,frame no & offset):4,400
Do you want to continue(1/0)?:0
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

AIM : Write a program to demonstrate segmentation.

DESCRIPTION :

Segmentation is an alternative memory management technique that divides memory into logically meaningful segments, each representing a distinct unit such as code, data, or stack. Each segment is assigned a base address and a limit, defining its range in the logical address space. The mapping between logical and physical addresses is maintained through a segment table. Segmentation offers flexibility and intuitive organization of memory, facilitating the sharing and protection of segments. However, it may lead to internal fragmentation within segments, and its management can be more complex compared to paging.

ALGORITHM :

1. Initialization:
   - Divide logical memory into segments (code, data, stack).
   - Assign each segment a base address and a limit.
2. Initialize segment table entries.
   - Update segment table dynamically.
   - Translate logical addresses to physical addresses using the segment table.
3. Segmentation Fault Handling:
   - Terminate a process attempting illegal segment access.
4. Dynamic Memory Allocation:
   - Allow dynamic allocation of memory segments.
   - Adjust segment table entries dynamically.
5. Memory Access:
   - Translate logical addresses via the segment table.
   - Access data in the physical memory location

CODE :

```
#include<stdio.h>
int main() {
    int a[10][10], b[100], i, j, n, x, base, size, seg, off;
    printf("Enter the segments count\n");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter the %d size \n", i + 1);
        scanf("%d", &size);
        a[i][0] = size;
        printf("Enter the base address\n");
        scanf("%d", &base);
        a[i][1] = base;
        printf("Start entering the elements:\n");
        for (j = 0; j < size; j++) {
            x = 0;
            scanf("%d", &x);
```

```
            base++;
            b[base] = x;
        }
    }
    printf("Enter the segment number and offset value \n");
    scanf("%d%d", &seg, &off);
    if (off < a[seg][0]) {
        int abs = a[seg][1] + off;
        printf("The offset is less than %d\n", a[seg][0]);
        printf("%d + %d = %d\n", a[seg][1], off, abs);
        printf("The element %d is at %d\n", b[abs], abs);
    } else {
        printf("Error in locating\n");
    }
    return 0;
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ vi segmentation.c
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ gcc segmentation.c -o segmentation.sricharan_n
ama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./segmentation
Enter the segments count
5
Enter the 1 size
4
Enter the base address
2
Start entering the elements:
9
7
1
2
Enter the 2 size
3
Enter the base address
14
Start entering the elements:
25
14
17
Enter the 3 size
1
Enter the base address
1
Start entering the elements:
3
Enter the 4 size
7
Enter the base address
4
Start entering the elements:
1
2
3
5
4
5
1
Enter the 5 size
2
Enter the base address
2
Start entering the elements:
2
2
Enter the segment number and offset value
1
3
Error in locating
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ |
```

WEEK – 8

AIM : Write a program to demonstrate signals.

DESCRIPTION :

Signals are software interrupts notifying processes or threads about events, from user input to errors or termination requests. They facilitate inter-process communication, aiding process management and control. Each signal has a unique identifier and default action, modifiable by processes using signal handlers—functions defining custom actions for specific signals.

- sigint(signal interupt) - SIGINT interrupts a process and is typically generated by the user, often triggered by pressing Ctrl+C.The default action of SIGINT is to gracefully terminate the process, allowing it to clean up resources before exiting.
- sigchld(signal child) - SIGCHLD signals that a child process has terminated, notifying the parent process of the child's completion. The default action for SIGCHLD is to be ignored, as the operating system retains information about terminated child processes.

CODE :

sigint.c

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
void handle_signal(int sig)// Handler
{
  printf("\n Signal Caught %d \n",sig);}
int main()
{
   signal(SIGINT,handle_signal);
   int i=0;
   while(i<30){
     printf("Hello World \n");
     sleep(1);
     i++;}
    return 0;
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./sigint
Hello World
Hello World
Hello World
Hello World
Hello World
^C
 Signal Caught 2
Hello World
Hello World
^C
 Signal Caught 2
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
^[Hello World
Hello World
Hello World
^Z
[1]+  Stopped                 ./sigint
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

CODE :

sigchld.c

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<signal.h>
void handle_signal(int sig)// Handler
{
  printf("\n Inside Handler: child is terminated  %d \n",sig);
}
int main()
{
   signal(SIGCHLD,handle_signal);// Registration process
   int i=fork(), w;
          if(i==0)
          {
          printf("Child process");
          }
          else
          {
          printf("Inside parent process \n");
          wait(&w);
          printf("End of parent process \n");
          }
          return 0;
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ vi sigchild.c
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ gcc sigchild.c -o sigchild
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./sigchild
Inside parent process
Child process
 Inside Handler: child is terminated  17
End of parent process
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

AIM : WAP to demonstrate the use of Shared Memory -IPC(Inter Process Communication)

DESCRIPTION:

Shared memory in Inter-Process Communication (IPC) allows multiple processes to access a common region of memory, enabling efficient data exchange. By sharing this memory space, processes can communicate and synchronize more quickly than with other IPC mechanisms, like message passing. Proper synchronization mechanisms, such as semaphores, are crucial to prevent conflicts and ensure data integrity.

ALGORITHM :

1. Start
2. Include necessary headers: stdio.h, stdlib.h, unistd.h, sys/shm.h, string.h.
3. Declare variables i, shared_memory(Pointer to the shared memory segment), buff(Character array to store data read from shared memory) ,shmid(Integer variable to store the shared memory identifier)
4. Call shmget to create or access a shared memory segment with key 2345.
5. Print the obtained key for reference.
6. Call shmat to attach the process to the shared memory segment identified by shmid.
7. Obtain the starting address of the shared memory segment.
8. Print the address where the process is attached.
9. For smread :
    - Copy the data from the shared memory to the local buffer (buff).
10. For smsend :
    - Prompt the user to enter the data to be written to shared memory.
    - Use read to read up to 100 characters from standard input into the local buffer (buff).
    - Copy the data from the local buffer (buff) to the shared memory segment using strcpy.
11. Print the data read from the shared memory.
12. Detach the process from the shared memory using shmdt if necessary.
13. End

CODE :

smsend

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
```

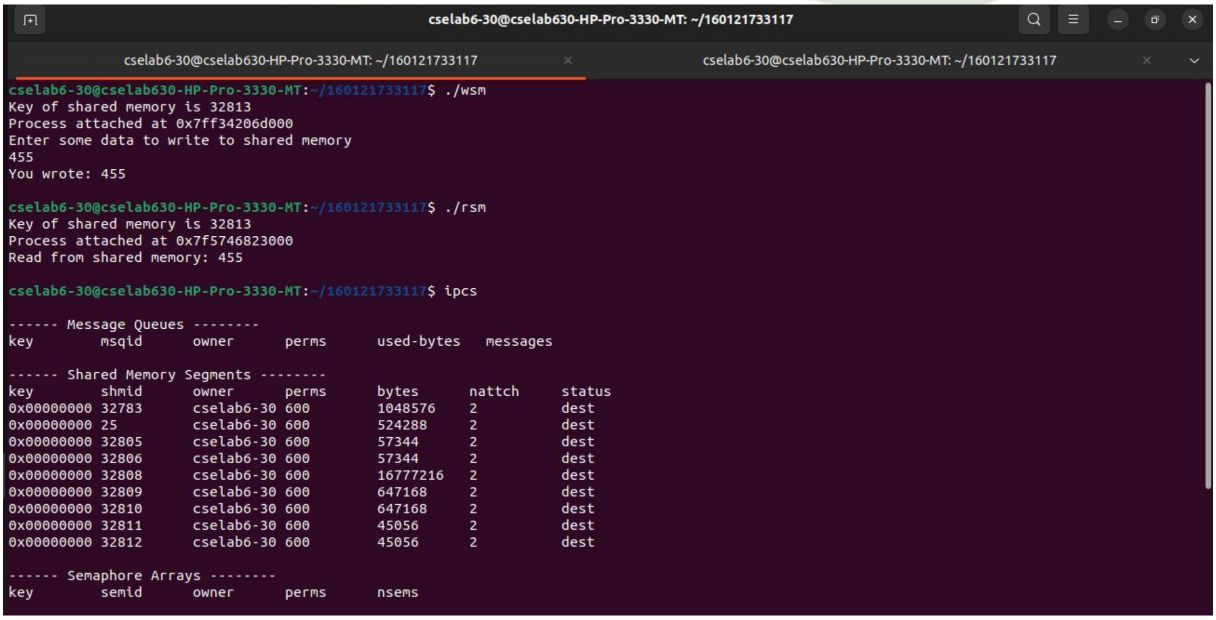
```
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0);
printf("Process attached at %p\n",shared_memory);
printf("Enter some data to write to shared memory\n");
read(0,buff,100);
strcpy(shared_memory,buff);
printf("You wrote : %s\n",(char *)shared_memory);
}
```

CODE :

smread

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0);
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

OUTPUT :



```
cselab6-30@cselab630-HP-Pro-3330-MT: ~/160121733117
cselab6-30@cselab630-HP-Pro-3330-MT:~/160121733117$ ./wsm
Key of shared memory is 32813
Process attached at 0x7ff34206d000
Enter some data to write to shared memory
455
You wrote: 455

cselab6-30@cselab630-HP-Pro-3330-MT:~/160121733117$ ./rsm
Key of shared memory is 32813
Process attached at 0x7f5746823000
Read from shared memory: 455

cselab6-30@cselab630-HP-Pro-3330-MT:~/160121733117$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 32783      cselab6-30 600        1048576    2          dest
0x00000000 25         cselab6-30 600        524288     2          dest
0x00000000 32805      cselab6-30 600        57344      2          dest
0x00000000 32806      cselab6-30 600        57344      2          dest
0x00000000 32808      cselab6-30 600        16777216   2          dest
0x00000000 32809      cselab6-30 600        647168     2          dest
0x00000000 32810      cselab6-30 600        647168     2          dest
0x00000000 32811      cselab6-30 600        45056      2          dest
0x00000000 32812      cselab6-30 600        45056      2          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

AIM : WAP to demonstrate race condition

DESCRIPTION :

A race condition in operating systems arises when multiple processes access shared resources simultaneously, leading to unpredictable outcomes depending on execution order. Without proper synchronization, conflicts may occur, compromising data integrity and system stability. Synchronization mechanisms like locks are crucial to prevent race conditions and ensure orderly access to shared resources.

ALGORITHM :

1. Start
2. Import necessary libraries: pthread.h, stdio.h, unistd.h.
3. Declare a global variable shared initialized to 1.
4. Create two threads (thread1 and thread2) using pthread_create.
5. Thread 1 (fun1):
   - Read the current value of shared.
   - Print the read value.
   - Increment the local variable.
   - Print the locally updated value.
   - Sleep for 1 second.
   - Update the shared variable with the local value.
   - Print the final updated value of shared.
6. Thread 2 (fun2):
   - Read the current value of shared.
   - Print the read value.
   - Decrement the local variable.
   - Print the locally updated value.
   - Sleep for 1 second.
   - Update the shared variable with the local value.
   - Print the final updated value of shared.
7. Wait for both threads to complete using pthread_join.
8. Print the final value of the shared variable.
9. end

CODE :

```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1;
int main()
 {
 pthread_t thread1, thread2;
```

```
 pthread_create(&thread1, NULL, fun1, NULL);
 pthread_create(&thread2, NULL, fun2, NULL);
 pthread_join(thread1, NULL);
 pthread_join(thread2,NULL);
 printf("Final value of shared is %d\n",shared);
 }
void *fun1()
{
   int x;
   x=shared;
   printf("Thread1 reads the value of shared variable as %d\n",x);
   x++;
   printf("Local updation by Thread1: %d\n",x);
   sleep(1);
   shared=x;
   printf("Value of shared variable updated by Thread1 is: %d\n",shared);
}
void *fun2()
 {
    int y;
    y=shared;
    printf("Thread2 reads the value as %d\n",y);
    y--;
    printf("Local updation by Thread2: %d\n",y);
    sleep(1);
    shared=y;
    printf("Value of shared variable updated by Thread2 is: %d\n",shared);
 }
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ gcc race.c -o race
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./race
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Thread2 reads the value as 1
Local updation by Thread2: 0
Value of shared variable updated by Thread1 is: 2
Value of shared variable updated by Thread2 is: 0
Final value of shared is 0
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ |
```

AIM : WAP to demonstrate the use of Semaphore -IPC (Critical section Problem)

DESCRIPTION :

Semaphores in Inter-Process Communication (IPC) are synchronization mechanisms used to control access to shared resources among multiple processes. They help prevent race conditions by allowing processes to signal each other when a resource is available or when an operation is complete. Semaphores can be used to coordinate and synchronize concurrent processes, ensuring orderly access to shared resources and avoiding conflicts in a multi-process environment.

CODE :

```
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1;
sem_t s;
int main()
{
sem_init(&s,0,1);
pthread_t thread1, thread2;
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2,NULL);
printf("Final value of shared is %d\n",shared);
}
void *fun1()
{
   int x;
   sem_wait(&s);
   x=shared;
   printf("Thread1 reads the value as %d\n",x);
   x++;
   printf("Local updation by Thread1: %d\n",x);
   sleep(1);
   shared=x;
   printf("Value of shared variable updated by Thread1 is: %d\n",shared);
   sem_post(&s);
}
void *fun2()
{
   int y;
```

```
    sem_wait(&s);
    y=shared;
    printf("Thread2 reads the value as %d\n",y);
    y--;
    printf("Local updation by Thread2: %d\n",y);
    sleep(1);
    shared=y;
    printf("Value of shared variable updated by Thread2 is: %d\n",shared);
    sem_post(&s);
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./semaphore
Thread2 reads the value as 1
Local updation by Thread2: 0
Value of shared variable updated by Thread2 is: 0
Thread1 reads the value as 0
Local updation by Thread1: 1
Value of shared variable updated by Thread1 is: 1
Final value of shared is 1
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ |
```

CODE :

semaphoresexample

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <unistd.h>
void *withdraw();
void *deposite();
int amount = 1;
sem_t s;
int main()
{
    sem_init(&s, 0, 1);
    pthread_t thread1, thread2;
    int damount, wamount;
    printf("Enter the amount for Deposite");
    scanf("%d", &damount);
    printf("Enter the wthdraw amount");
    scanf("%d", &wamount);
    pthread_create(&thread1, NULL, withdraw, (void *)wamount);
    pthread_create(&thread2, NULL, deposite, (void *)damount);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Final value of shared is %d\n", amount);
}
void *withdraw()
```

```
{
   int x;
   sem_wait(&s);
   x = amount;
   printf("Thread1 reads the value as %d\n", x);
   x = x + 1000;
   printf("Local updation by Thread1: %d\n", x);
   sleep(1);
   amount = x;
   printf("Value of shared variable updated by Thread1 is: %d\n", amount);
   sem_post(&s);
}
void *deposite()
{
   int y;
   sem_wait(&s);
   y = amount;
   printf("Thread2 reads the value as %d\n", y);
   y = y - 500;
   printf("Local updation by Thread2: %d\n", y);
   sleep(1);
   amount = y;
   printf("Value of shared variable updated by Thread2 is: %d\n", amount);
   sem_post(&s);
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./semaphoreex
Enter the amount for Deposite700
Enter the wthdraw amount300
Thread1 reads the value as 1
Local updation by Thread1: 1001
Value of shared variable updated by Thread1 is: 1001
Thread2 reads the value as 1001
Local updation by Thread2: 501
Value of shared variable updated by Thread2 is: 501
Final value of shared is 501
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ |
```

AIM : WAP to demonstrate dining philosopher

DESCRIPTION :

The Dining Philosophers problem is often used as an illustration of synchronization issues in operating systems and concurrent programming. In this scenario, the philosophers represent processes, and the forks represent resources that these processes need to execute their tasks. The goal is to design a solution that avoids deadlock(if each philosopher picks up one fork and waits for the other, a deadlock can occur where no philosopher can proceed with eating) and starvation(If a philosopher is unable to acquire the required forks due to the actions of their neighbors, they may starve and never get a chance to eat).

CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t chopstick[5];
void *philos(void *);
void eat(int);
int main()
{
	int i, n[5];
	pthread_t T[5];
	for (i = 0; i < 5; i++)
		sem_init(&chopstick[i], 0, 1);
	for (i = 0; i < 5; i++)
	{
		n[i] = i;
		pthread_create(&T[i], NULL, philos, (void *)&n[i]);
	}
	for (i = 0; i < 5; i++)
		pthread_join(T[i], NULL);
}
void *philos(void *n)
{
	int ph = *(int *)n;
	printf("Philosopher %d wants to eat\n", ph);
	printf("Philosopher %d tries to pick left chopstick\n", ph);
	sem_wait(&chopstick[ph]);
	printf("Philosopher %d picks the left chopstick\n", ph);
	printf("Philosopher %d tries to pick the right chopstick\n", ph);
	sem_wait(&chopstick[(ph + 1) % 5]);
	printf("Philosopher %d picks the right chopstick\n", ph);
	eat(ph);
```

```
        sleep(2);
        printf("Philosopher %d has finished eating\n", ph);
        sem_post(&chopstick[(ph + 1) % 5]);
        printf("Philosopher %d leaves the right chopstick\n", ph);
        sem_post(&chopstick[ph]);
        printf("Philosopher %d leaves the left chopstick\n", ph);
}
void eat(int ph)
{
        printf("Philosopher %d begins to eat\n", ph);
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ vi dining.c
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ gcc dining.c -o dining
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./dining
Philosopher 0 wants to eat
Philosopher 0 tries to pick left chopstick
Philosopher 3 wants to eat
Philosopher 3 tries to pick left chopstick
Philosopher 3 picks the left chopstick
Philosopher 3 tries to pick the right chopstick
Philosopher 3 picks the right chopstick
Philosopher 3 begins to eat
Philosopher 2 wants to eat
Philosopher 2 tries to pick left chopstick
Philosopher 2 picks the left chopstick
Philosopher 2 tries to pick the right chopstick
Philosopher 0 picks the left chopstick
Philosopher 0 tries to pick the right chopstick
Philosopher 0 picks the right chopstick
Philosopher 0 begins to eat
Philosopher 1 wants to eat
Philosopher 1 tries to pick left chopstick
Philosopher 4 wants to eat
Philosopher 4 tries to pick left chopstick
Philosopher 3 has finished eating
Philosopher 3 leaves the right chopstick
Philosopher 4 picks the left chopstick
Philosopher 4 tries to pick the right chopstick
Philosopher 3 leaves the left chopstick
Philosopher 2 picks the right chopstick
Philosopher 2 begins to eat
Philosopher 0 has finished eating
Philosopher 0 leaves the right chopstick
Philosopher 0 leaves the left chopstick
Philosopher 1 picks the left chopstick
Philosopher 1 tries to pick the right chopstick
Philosopher 4 picks the right chopstick
Philosopher 4 begins to eat
Philosopher 2 has finished eating
Philosopher 2 leaves the right chopstick
Philosopher 2 leaves the left chopstick
Philosopher 1 picks the right chopstick
Philosopher 1 begins to eat
Philosopher 4 has finished eating
Philosopher 4 leaves the right chopstick
Philosopher 4 leaves the left chopstick
Philosopher 1 has finished eating
Philosopher 1 leaves the right chopstick
Philosopher 1 leaves the left chopstick
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

AIM : WAP to demonstrate the use of Mutex -IPC (Critical section Problem)

DESCRIPTION :

In Inter-Process Communication (IPC), a Mutex (short for mutual exclusion) is a synchronization mechanism used to control access to shared resources among multiple processes. It ensures that only one process at a time can access the critical section of code, preventing data corruption or race conditions. Mutexes help maintain data integrity by allowing processes to acquire and release locks on shared resources, enforcing a mutually exclusive access pattern.

CODE :

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
void *fun1();
void *fun2();
int shared = 1;
pthread_mutex_t l;
int main()
{
   pthread_mutex_init(&l, NULL);
   pthread_t thread1, thread2;
   pthread_create(&thread1, NULL, fun1, NULL);
   pthread_create(&thread2, NULL, fun2, NULL);
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);
   printf("Final value of shared is %d\n", shared);
}
void *fun1()
{
   int x;
   printf("Thread1 trying to acquire lock\n");
   pthread_mutex_lock(&l);
   printf("Thread1 acquired lock\n");
   x = shared;
   printf("Thread1 reads the value of shared variable as %d\n", x);
   x++;
   printf("Local updation by Thread1: %d\n", x);
   sleep(1);
   shared = x;
   printf("Value of shared variable updated by Thread1 is: %d\n", shared);
   pthread_mutex_unlock(&l);
   printf("Thread1 released the lock\n");
}
```

```
void *fun2()
{
   int y;
   printf("Thread2 trying to acquire lock\n");
   pthread_mutex_lock(&l);
   printf("Thread2 acquired lock\n");
   y = shared;
   printf("Thread2 reads the value as %d\n", y);
   y--;
   printf("Local updation by Thread2: %d\n", y);
   sleep(1);
   shared = y;
   printf("Value of shared variable updated by Thread2 is: %d\n", shared);
   pthread_mutex_unlock(&l);
   printf("Thread2 released the lock\n");
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./mutex
Thread1 trying to acquire lock
Thread1 acquired lock
Thread1 reads the value of shared variable as 1
Local updation by Thread1: 2
Thread2 trying to acquire lock
Value of shared variable updated by Thread1 is: 2
Thread1 released the lock
Thread2 acquired lock
Thread2 reads the value as 2
Local updation by Thread2: 1
Value of shared variable updated by Thread2 is: 1
Thread2 released the lock
Final value of shared is 1
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ |
```

CODE :

mutexevenodd

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define LOCK(m) pthread_mutex_lock(m)
#define UNLOCK(m) pthread_mutex_unlock(m)
#define WAIT(c,m) pthread_cond_wait(c, m)
#define SIGNAL(c) pthread_cond_signal(c)
#define MAX_COUNT 200
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
int count = 0;
void* printEven(void* data)
{
   int tid = *((int*)data);
   while (1)
   {
```

```
        LOCK(&lock);
        while (count%2 == 0)
            WAIT(&condition, &lock);
        if (count < MAX_COUNT)
        {
            count++;
            printf("Thread %d, count %d\n", tid, count);
        }
        SIGNAL(&condition);
        if (count >= MAX_COUNT)
        {
            UNLOCK(&lock);
            return NULL;
        }
        UNLOCK(&lock);
    }
}
void* printOdd(void* data)
{
    int tid = *((int*)data);
    while (1)
    {
        LOCK(&lock);
        while (count%2 != 0)
            WAIT(&condition, &lock);
        if (count < MAX_COUNT)
        {
            count++;
            printf("Thread %d, count %d\n", tid, count);
        }
        SIGNAL(&condition);
        if (count >= MAX_COUNT)
        {
            UNLOCK(&lock);
            return NULL;
        }
        UNLOCK(&lock);
    }
}
int main(int argc, char** argv)
{
    pthread_t thread1, thread2;
    int tid[] = {1, 2};
    pthread_create(&thread1, NULL, printOdd, &tid[0]);
    pthread_create(&thread2, NULL, printEven, &tid[1]);
    pthread_join(thread1, NULL);
```

```
    pthread_join(thread2, NULL);
    return 0;
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./mutexevenodd
Thread 1, count 1
Thread 2, count 2
Thread 1, count 3
Thread 2, count 4
Thread 1, count 5
Thread 2, count 6
Thread 1, count 7
Thread 2, count 8
Thread 1, count 9
Thread 2, count 10
Thread 1, count 11
Thread 2, count 12
Thread 1, count 13
Thread 2, count 14
Thread 1, count 15
Thread 2, count 16
Thread 1, count 17
Thread 2, count 18
Thread 1, count 19
Thread 2, count 20
Thread 1, count 21
Thread 2, count 22
Thread 1, count 23
Thread 2, count 24
Thread 1, count 25
Thread 2, count 26
Thread 1, count 27
Thread 2, count 28
Thread 1, count 29
Thread 2, count 30
Thread 1, count 31
Thread 2, count 32
Thread 1, count 33
Thread 2, count 34
Thread 1, count 35
Thread 2, count 36
Thread 1, count 37
Thread 2, count 38
Thread 1, count 39
```

 Signature of the Faculty……………………...

AIM : WAP to demonstrate the use of Socket programming

DESCRIPTION :

Socket programming in operating systems facilitates inter-process communication across a network. In Python, a server establishes a socket, binds it to an address, and listens for connections. Clients connect to the server, enabling the exchange of data. This mechanism, often utilizing TCP or UDP protocols, forms the backbone of networked applications, providing a foundation for robust communication between processes in an operating system, vital for distributed computing and internet-based services.

CODE :

Server.c

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
void func(int connfd)
{
char buff[MAX];
int n;
for (;;) {
bzero(buff, MAX);
read(connfd, buff, sizeof(buff));
printf("From client: %s\t To client : ", buff);
bzero(buff, MAX);
n = 0;
while ((buff[n++] = getchar()) != '\n')
;
write(connfd, buff, sizeof(buff));
if (strncmp("exit", buff, 4) == 0) {
printf("Server Exit...\n");
break;
}
}
}
int main()
{
int sockfd, connfd, len;
```

```
struct sockaddr_in servaddr, cli;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
printf("socket creation failed...\n");
exit(0);
}
else
printf("Socket successfully created..\n");
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(PORT);
if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
printf("socket bind failed...\n");
exit(0);
}
else
printf("Socket successfully binded..\n");
if ((listen(sockfd, 5)) != 0) {
printf("Listen failed...\n");
exit(0);
}
else
printf("Server listening..\n");
len = sizeof(cli);
connfd = accept(sockfd, (SA*)&cli, &len);
if (connfd < 0) {
printf("server accept failed...\n");
exit(0);
}
else
printf("server accept the client...\n");
func(connfd);
close(sockfd);
}
```

Client.c

```
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <unistd.h>
#define MAX 80
```

```
#define PORT 8080
#define SA struct sockaddr
void func(int sockfd)
{
char buff[MAX];
int n;
for (;;) {
bzero(buff, sizeof(buff));
printf("Enter the string : ");
n = 0;
while ((buff[n++] = getchar()) != '\n')
;
write(sockfd, buff, sizeof(buff));
bzero(buff, sizeof(buff));
read(sockfd, buff, sizeof(buff));
printf("From Server : %s", buff);
if ((strncmp(buff, "exit", 4)) == 0) {
printf("Client Exit...\n");
break;
}
}
}
int main()
{
int sockfd, connfd;
struct sockaddr_in servaddr, cli;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
printf("socket creation failed...\n");
exit(0);
}
else
printf("Socket successfully created..\n");
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
servaddr.sin_port = htons(PORT);
if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr))
!= 0) {
printf("connection with the server failed...\n");
exit(0);
}
else
printf("connected to the server..\n");
func(sockfd);
close(sockfd);
```

}

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./server
Socket successfully created..
Socket successfully binded..
Server listening..
server accept the client...
From client: hello Sricharan
         To client : hi
From client: what is your age
         To client : 21
From client: faviurite cricket player
         To client : MS Dhoni
From client:     To client : ^C
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./client
Socket successfully created..
connected to the server..
Enter the string : hello Sricharan
From Server : hi
Enter the string : what is your age
From Server : 21
Enter the string : faviurite cricket player
From Server : MS Dhoni
Enter the string : ^C
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```

WEEK – 9

AIM : WAP program to demonstrate the use of setuid(), setgid() system calls

DESCRIPTION :

In Unix-like operating systems, the setuid and setgid system calls are used to change the effective user ID (UID) and effective group ID (GID) of a process, respectively. These system calls are often employed for security and privilege management in certain scenarios.

- setuid: The setuid system call in Unix-like systems changes a process's effective user ID, often used by privileged programs to relinquish superuser privileges. It allows temporary elevation of permissions, demanding cautious use to prevent security vulnerabilities and unauthorized access.
- setgid: Unix-like systems employ the setgid system call to alter a process's effective group ID. Similar to setuid, it facilitates controlled elevation of privileges, crucial for processes needing temporary access to specific group-related resources, demanding careful handling to ensure security integrity.

CODE :

```
#include<stdio.h>
#include<unistd.h>
int main()
{
uid_t uid;
printf("Before setting the uid and gid\nreal user id: %d\nreal grp id: %d\neffective user id:
%d\neffective grp id:%d\n",getuid(),getgid(),geteuid(),getegid());
setuid(1003);
setgid(1002);
printf("After setting the uid and gid\nreal user id: %d\nreal grp id: %d\neffective user id:
%d\neffective grp id: %d\n",getuid(),getgid(),geteuid(),getegid());
return 0;
}
```

OUTPUT :

```
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$ ./getuid
Before setting the uid and gid
real user id: 1000
real grp id: 1000
effective user id: 1000
effective grp id:1000
After setting the uid and gid
real user id: 1000
real grp id: 1000
effective user id: 1000
effective grp id: 1000
sricharan_nama@Sricharan:/mnt/d/Operating Systems/160121733117/lab$
```