Laboratory Manual

(Operating System Lab and PCC-CSM491)

Table of Contents

Sujit Kumar Sadhukhan
Assistant Professor, CSE-CS&DS
Brainware University, Kolkata

Experiment 1

AIM:

Write a program on process, replacing a process image.

CODE:

```
#include<stdio.h>
#include<unistd.h>
int main()
{
   printf("Before execl\n");
   execl("/bin/ps","ps","-a",NULL);//
   printf("After execlp\n");
}
```

Output:

$gcc -o exec execl.c

$./exec

```
Before execl
  PID TTY          TIME CMD
  122 tty1     00:00:00 ps
```

Experiment 2

AIM:

Write a Program on duplicating a process image, waiting for a process, zombie process.

CODE:

- **Duplicating a Process Image**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#define BUFFER_SIZE 4096
void copy_process_image(pid_t pid, const char *output_path) {
```

```
    char mem_path[64];
    snprintf(mem_path, sizeof(mem_path), "/proc/%d/mem", pid);
    int mem_fd = open(mem_path, O_RDONLY);
    if (mem_fd == -1) {
        perror("Failed to open /proc/PID/mem");
        return;
    }
    int output_fd = open(output_path, O_WRONLY | O_CREAT | O_TRUNC, 0777);
    if (output_fd == -1) {
        perror("Failed to create output file");
        close(mem_fd);
        return;
    }
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;
    while ((bytes_read = pread(mem_fd, buffer, BUFFER_SIZE, 0)) > 0) {
        if (write(output_fd, buffer, bytes_read) != bytes_read) {
            perror("Failed to write to output file");
            close(mem_fd);
            close(output_fd);
            return;
        }
    }
    close(mem_fd);
    close(output_fd);
    printf("Process image copied to %s\n", output_path);
}
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <pid> <output_path>\n", argv[0]);
        return 1;
    }
    pid_t pid = atoi(argv[1]);
```

```
    const char *output_path = argv[2];
    copy_process_image(pid, output_path);
    return 0;
}
```

To compile this program, save it to a file (e.g., copy_process_image.c) and run:

gcc copy_process_image.c -o copy_process_image

Then, to copy the process image, execute the compiled program with the PID of the target process and the desired output path:

./copy_process_image <pid> <output_path>

Replace <pid> with the process ID of the target process you want to copy, and <output_path> with the desired path for the output file.

- **Waiting for a process**

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
    pid_t cpid;
    if (fork()== 0)
        exit(0);          /* terminate child */
    else
        cpid = wait(NULL); /* reaping parent */
    printf("Parent pid = %d\n", getpid());
    printf("Child pid = %d\n", cpid);
    return 0;
}
```

Output:

Parent pid = 12345678

Child pid = 89546848

- **Zombie Process**

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

```
#include <sys/types.h>
#include <unistd.h>
int main()
{
   // Fork returns process id
   // in parent process
   pid_t child_pid = fork();
   // Parent process
   if (child_pid > 0)
      sleep(50);
   // Child process
   else
      exit(0);
   return 0;
}
```

Output:

```
guest-glcbIs@ubuntu:~$gcc  zombie.c
guest-glcbIs@ubuntu:~$./a.out
```

Then command prompt will wait for some time (50 sec) and then again command prompt will appear later.

Experiment 3

AIM:

Write a Program on Preemptive and Non-preemptive, FCFS, SJF.

CODE:

First Come First Serve Scheduling

```
// C program for implementation of FCFS
// scheduling
#include<stdio.h>
// Function to find the waiting time for all
// processes
void findWaitingTime(int processes[], int n,
                     int bt[], int wt[])
{
   // waiting time for first process is 0
```

```
    wt[0] = 0;
    // calculating waiting time
    for (int  i = 1; i < n ; i++ )
        wt[i] =  bt[i-1] + wt[i-1] ;
}
// Function to calculate turn around time
void findTurnAroundTime( int processes[], int n,
                int bt[], int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int  i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}
//Function to calculate average time
void findavgTime( int processes[], int n, int bt[])
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    //Function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt);
    //Function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);
    //Display processes along with all details
    printf("Processes   Burst time   Waiting time   Turn around time\n");
    // Calculate total waiting time and total turn
    // around time
    for (int  i=0; i<n; i++)
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        printf("%d ",(i+1));
        printf("%d ", bt[i] );
        printf("%d",wt[i] );
```

```
        printf("%d\n",tat[i] );
    }
    float s=(float)total_wt / (float)n;
    float t=(float)total_tat / (float)n;
    printf("Average waiting time = %f",s);
    printf("\n");
    printf("Average turn around time = %f ",t);
}
// Driver code
int main()
{
    //process id's
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];
    //Burst time of all processes
    int  burst_time[] = {10, 5, 8};
    findavgTime(processes, n,  burst_time);
    return 0;
}
```

Output:

Processes Burst Time Waiting Time Turnaround Time

1 10 0 10

2 5 10 15

3 8 15 23

Average waiting time = 8.33333

Average turnaround time = 16

Shortest Job First Scheduling

```
/*
 * C Program to Implement SJF Scheduling
 */
#include<stdio.h>
int main()
{
```

```
int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,totalT=0,pos,temp;
float avg_wt,avg_tat;
printf("Enter number of process:");
scanf("%d",&n);
printf("\nEnter Burst Time:\n");
for(i=0;i<n;i++)
{
   printf("p%d:",i+1);
   scanf("%d",&bt[i]);
   p[i]=i+1;
}
//sorting of burst times
for(i=0;i<n;i++)
{
   pos=i;
   for(j=i+1;j<n;j++)
   {
      if(bt[j]<bt[pos])
         pos=j;
   }
   temp=bt[i];
   bt[i]=bt[pos];
   bt[pos]=temp;
   temp=p[i];
   p[i]=p[pos];
   p[pos]=temp;
}
wt[0]=0;
//finding the waiting time of all the processes
for(i=1;i<n;i++)
{
   wt[i]=0;
   for(j=0;j<i;j++)
```

```
        //individual WT by adding BT of all previous completed processes
        wt[i]+=bt[j];
      //total waiting time
      total+=wt[i];
   }
   //average waiting time
   avg_wt=(float)total/n;
   printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");
   for(i=0;i<n;i++)
   {
      //turnaround time of individual processes
      tat[i]=bt[i]+wt[i];
      //total turnaround time
      totalT+=tat[i];
      printf("\np%d\t\t %d\t\t %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
   }

   //average turnaround time
   avg_tat=(float)totalT/n;
   printf("\n\nAverage Waiting Time=%f",avg_wt);
   printf("\nAverage Turnaround Time=%f",avg_tat);
}
```

Output:

Enter number of process:4

Enter Burst Time:

p1:5

p2:4

p3:12

p4:7

Process	Burst Time	Waiting Time	Turnaround Time
p2	4	0	4
p1	5	4	9
p4	7	9	16

p3 12 16 28

Average Waiting Time=7.250000

Average Turnaround Time=14.250000

Experiment 4

AIM:

Write a Program on Priority & Round Robin CPU Scheduling Algorithm.

CODE:

Priority Scheduling

```
/*
 * C program to implement priority scheduling
 */

#include <stdio.h>

//Function to swap two variables
void swap(int *a,int *b)
{
   int temp=*a;
   *a=*b;
   *b=temp;
}
int main()
{
   int n;
   printf("Enter Number of Processes: ");
   scanf("%d",&n);

   // b is array for burst time, p for priority and index for process id
   int b[n],p[n],index[n];
   for(int i=0;i<n;i++)
   {
      printf("Enter Burst Time and Priority Value for Process %d: ",i+1);
      scanf("%d %d",&b[i],&p[i]);
```

```
    index[i]=i+1;
 }
 for(int i=0;i<n;i++)
 {
    int a=p[i],m=i;

    //Finding out highest priority element and placing it at its desired position
    for(int j=i;j<n;j++)
    {
       if(p[j] > a)
       {
          a=p[j];
          m=j;
       }
    }
    //Swapping processes
    swap(&p[i], &p[m]);
    swap(&b[i], &b[m]);
    swap(&index[i],&index[m]);
 }
 // T stores the starting time of process
 int t=0;
 //Printing scheduled process
 printf("Order of process Execution is\n");
 for(int i=0;i<n;i++)
 {
    printf("P%d is executed from %d to %d\n",index[i],t,t+b[i]);
    t+=b[i];
 }
 printf("\n");
 printf("Process Id     Burst Time   Wait Time    TurnAround Time\n");
 int wait_time=0;
 for(int i=0;i<n;i++)
```


```
  {
    printf("P%d        %d        %d        %d\n",index[i],b[i],wait_time,wait_time + b[i]);
    wait_time += b[i];
  }
  return 0;
}
```

Output:

Enter Number of Processes: 3

Enter Burst Time and Priority Value for Process 1: 10 2

Enter Burst Time and Priority Value for Process 2: 5 0

Enter Burst Time and Priority Value for Process 3: 8 1

Order of process Execution is

P1 is executed from 0 to 10

P3 is executed from 10 to 18

P2 is executed from 18 to 23

Process Id	Burst Time	Wait Time	TurnAround Time
P1	10	0	10
P3	8	10	18
P2	5	18	23

Round Robin CPU Scheduling Algorithm

```
#include<stdio.h>
int main()
{
 int cnt,j,n,t,remain,flag=0,tq;
 int wt=0,tat=0,at[10],bt[10],rt[10];
 printf("Enter Total Process:\t ");
 scanf("%d",&n);
 remain=n;
 for(cnt=0;cnt<n;cnt++)
 {
  printf("Enter Arrival Time and Burst Time for Process Process Number %d :",cnt+1);
  scanf("%d",&at[cnt]);
  scanf("%d",&bt[cnt]);
  rt[cnt]=bt[cnt];
 }
 printf("Enter Time Quantum:\t");
 scanf("%d",&tq);
 printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
 for(t=0,cnt=0;remain!=0;)
 {
  if(rt[cnt]<=tq && rt[cnt]>0)
  {
   t+=rt[cnt];
   rt[cnt]=0;
   flag=1;
  }
  else if(rt[cnt]>0)
  {
   rt[cnt]-=tq;
   t+=tq;
  }
  if(rt[cnt]==0 && flag==1)
```

```
  {
    remain--;
    printf("P[%d]\t|\t%d\t|\t%d\n",cnt+1,t-at[cnt],t-at[cnt]-bt[cnt]);
    wt+=t-at[cnt]-bt[cnt];
    tat+=t-at[cnt];
    flag=0;
  }
  if(cnt==n-1)
    cnt=0;
  else if(at[cnt+1]<=t)
    cnt++;
  else
    cnt=0;
 }
 printf("\nAverage Waiting Time= %f\n",wt*1.0/n);
 printf("Avg Turnaround Time = %f",tat*1.0/n);
 return 0;
}
```

Output:

```
Enter Total Process:     4
Enter Arrival Time and Burst Time for Process Process Number 1 :0  5
Enter Arrival Time and Burst Time for Process Process Number 2 :1  4
Enter Arrival Time and Burst Time for Process Process Number 3 :2  2
Enter Time Quantum:     2
Process |Turnaround Time|Waiting Time
P[3]    |      4       |      2
P[4]    |      3       |      2
P[2]    |      10      |      6
P[1]    |      12      |      7

Average Waiting Time= 4.250000
Avg Turnaround Time = 7.250000
```

Experiment 5

AIM:

Write a Program on multilevel Queue and Multilevel Feedback Queue Scheduling.

CODE:

```
#include<stdio.h>
int main()
{
	int p[20],bt[20], su[20], wt[20],tat[20],i, k, n, temp;
	float wtavg, tatavg;
	printf("Enter the number of processes:");
	scanf("%d",&n);
	for(i=0;i<n;i++)
	{
		p[i] = i;
		printf("Enter the Burst Time of Process%d:", i);
		scanf("%d",&bt[i]);
		printf("System/User Process (0/1) ? ");
		scanf("%d", &su[i]);
	}
	for(i=0;i<n;i++)
		for(k=i+1;k<n;k++)
			if(su[i] > su[k])
			{
			temp=p[i];
			p[i]=p[k];
			p[k]=temp;
			temp=bt[i];
			bt[i]=bt[k];
			bt[k]=temp;
			temp=su[i];
			su[i]=su[k];
			su[k]=temp;
			}
```

```
        wtavg = wt[0] = 0;
        tatavg = tat[0] = bt[0];
        for(i=1;i<n;i++)
        {
                wt[i] = wt[i-1] + bt[i-1];
                tat[i] = tat[i-1] + bt[i];
                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }
        printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
        for(i=0;i<n;i++)
                printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],su[i],bt[i],wt[i],tat[i]);
        printf("\nAverage Waiting Time is --- %f",wtavg/n);
        printf("\nAverage Turnaround Time is --- %f",tatavg/n);
        return 0;
}
```

Output:

```
Enter the number of processes:3
Enter the Burst Time of Process0:12
System/User Process (0/1) ? 0
Enter the Burst Time of Process1:18
System/User Process (0/1) ? 0
Enter the Burst Time of Process2:15
System/User Process (0/1) ? 1

PROCESS          SYSTEM/USER PROCESS    BURST TIME      WAITING TIME    TURNAROUND TIME
0                0                12              0               12
1                0                18              12              30
2                1                15              30              45
Average Waiting Time is --- 14.000000
Average Turnaround Time is --- 29.000000
```

Multilevel Feedback Queue Scheduling

```
#include<stdio.h>
#define MAX_PROCESSES 10
// Structure to represent a process
struct Process {
   int id;
```

```
    int burst_time;
    int remaining_time;
};
// Function to simulate multilevel feedback queue scheduling
void multilevel_feedback_queue(struct Process processes[], int n) {
    int time_quantum[3] = {4, 8, 16}; // Time quantum for each queue
    int current_time = 0;
    // Iterate over each time quantum
    for (int q = 0; q < 3; q++) {
        for (int i = 0; i < n; i++) {
            // If process is not completed and its remaining time is greater than 0
            if (processes[i].remaining_time > 0) {
                // Execute process for time quantum or remaining time, whichever is smaller
                int execution_time = (processes[i].remaining_time < time_quantum[q]) ? processes[i].remaining_time :
time_quantum[q];
                processes[i].remaining_time -= execution_time;
                current_time += execution_time;
        printf("Process %d executed for %d units at time %d\n", processes[i].id, execution_time, current_time);
                // If process is not completed, move it to the end of the queue
                if (processes[i].remaining_time > 0) {
                    printf("Process %d moved to the end of the queue\n", processes[i].id);
                }
            }
        }
    }
}
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[MAX_PROCESSES];
    // Input process details
    printf("Enter process details (id burst_time):\n");
```

```
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &processes[i].id, &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }
    multilevel_feedback_queue(processes, n); // Perform MLFQ scheduling
    return 0;
}
```

Output:

```
Enter the number of processes: 5
Enter process details (id burst_time):
1 2
2 3
3 1
4 5
5 6
Process 1 executed for 2 units at time 2
Process 2 executed for 3 units at time 5
Process 3 executed for 1 units at time 6
Process 4 executed for 4 units at time 10
Process 4 moved to the end of the queue
Process 5 executed for 4 units at time 14
Process 5 moved to the end of the queue
Process 4 executed for 1 units at time 15
Process 5 executed for 2 units at time 17
```

Experiment 6

AIM:

Write a Program on Signal handling, sending signals, signal interface, and signal sets.

CODE:

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

```
// Signal handler function
void sig_handler(int signum) {
    printf("Received signal %d\n", signum);
}

int main() {
    // Register signal handlers
    if (signal(SIGINT, sig_handler) == SIG_ERR) {
        perror("Signal registration failed");
        return 1;
    }

    if (signal(SIGTERM, sig_handler) == SIG_ERR) {
        perror("Signal registration failed");
        return 1;
    }

    // Sending signals
    printf("Sending SIGINT\n");
    kill(getpid(), SIGINT);
    sleep(2);
    printf("Sending SIGTERM\n");
    kill(getpid(), SIGTERM);

    // Signal set operations
    sigset_t set;
    sigemptyset(&set); // Initialize an empty signal set
    sigaddset(&set, SIGUSR1); // Add SIGUSR1 to the set

    if (sigismember(&set, SIGUSR1)) {
        printf("SIGUSR1 is in the set\n");
    } else {
        printf("SIGUSR1 is not in the set\n");
```

```
    }

    return 0;
}
```

Output:

Sending SIGINT

Received signal 2

Sending SIGTERM

Received signal 15

SIGUSR1 is in the set

Experiment 7

AIM:

Write a Program on semaphores use functions semctl, semget, semop.

CODE:

7.1

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

#define SEM_KEY 1234 // Key for semaphore set

int main() {
    int sem_id;
    struct semid_ds sem_info;

    // Create a semaphore set with one semaphore
    sem_id = semget(SEM_KEY, 1, IPC_CREAT | 0666);
    if (sem_id == -1) {
        perror("semget");
        exit(1);
```

```
    }

    // Get semaphore information
    if (semctl(sem_id, 0, IPC_STAT, &sem_info) == -1) {
        perror("semctl IPC_STAT");
        exit(1);
    }

    printf("Semaphore Information:\n");
    printf("Semaphore ID: %d\n", sem_id);
    printf("Owner's UID: %d\n", sem_info.sem_perm.uid);
    printf("Owner's GID: %d\n", sem_info.sem_perm.gid);
    printf("Mode: %#o\n", sem_info.sem_perm.mode);
    printf("Semaphore Value: %d\n", semctl(sem_id, 0, GETVAL));

    // Set semaphore value
    if (semctl(sem_id, 0, SETVAL, 5) == -1) {
        perror("semctl SETVAL");
        exit(1);
    }
    printf("Semaphore Value set to 5\n");
    printf("New Semaphore Value: %d\n", semctl(sem_id, 0, GETVAL));

    // Remove the semaphore set
    if (semctl(sem_id, 0, IPC_RMID, 0) == -1) {
        perror("semctl IPC_RMID");
        exit(1);
    }

    return 0;
}
```

Output:

Semaphore Information:

Semaphore ID: 1

Owner's UID: 1000

Owner's GID: 1000

Mode: 0666

Semaphore Value: 0

Semaphore Value set to 5

New Semaphore Value: 5

7.2

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

#define SEM_KEY 1234 // Key for semaphore set

int main() {
    int sem_id;

    // Create a semaphore set with one semaphore
    sem_id = semget(SEM_KEY, 1, IPC_CREAT | IPC_EXCL | 0666);
    if (sem_id == -1) {
        perror("semget");
        exit(1);
    }

    printf("Semaphore created with ID: %d\n", sem_id);

    // Remove the semaphore set
    if (semctl(sem_id, 0, IPC_RMID, 0) == -1) {
        perror("semctl IPC_RMID");
        exit(1);
    }
```

```
    printf("Semaphore removed\n");

    return 0;
}
```

Output:

Semaphore created with ID: 0

Semaphore removed

7.3

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

#define SEM_KEY 1234 // Key for semaphore set

int main() {
    int sem_id;
    struct sembuf sem_op;

    // Create a semaphore set with one semaphore
    sem_id = semget(SEM_KEY, 1, IPC_CREAT | 0666);
    if (sem_id == -1) {
        perror("semget");
        exit(1);
    }

    // Set initial semaphore value
    if (semctl(sem_id, 0, SETVAL, 1) == -1) {
        perror("semctl SETVAL");
        exit(1);
    }
```

```
// Perform semaphore operations
sem_op.sem_num = 0;
sem_op.sem_op = -1; // Wait operation (decrement semaphore value)
sem_op.sem_flg = 0; // No flags

printf("Waiting for semaphore...\n");

if (semop(sem_id, &sem_op, 1) == -1) {
    perror("semop wait");
    exit(1);
}

printf("Semaphore acquired!\n");

// Perform some critical section operations here

// Release the semaphore
sem_op.sem_op = 1; // Signal operation (increment semaphore value)

if (semop(sem_id, &sem_op, 1) == -1) {
    perror("semop signal");
    exit(1);
}

printf("Semaphore released!\n");

// Remove the semaphore set
if (semctl(sem_id, 0, IPC_RMID, 0) == -1) {
    perror("semctl IPC_RMID");
    exit(1);
}
```

```
    return 0;
}
```

Output:

Waiting for semaphore...

Semaphore acquired!

Semaphore released!

Experiment 8

AIM:

Write a Program on semaphores use functions semctl, semget, semop, set_semvalue, del_semvalue, semaphore_p, semaphore_v.

CODE (set_semvalue and del_semvalue):

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

#define SEM_KEY 1234 // Key for semaphore set

// Function to set semaphore value
int set_semvalue(int sem_id) {
    union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
    } sem_union;
    sem_union.val = 1; // Set the semaphore value to 1

    if (semctl(sem_id, 0, SETVAL, sem_union) == -1) {
        perror("semctl SETVAL");
        return 0;
    }
    return 1;
```

```
}

// Function to delete semaphore
void del_semvalue(int sem_id) {
    if (semctl(sem_id, 0, IPC_RMID, 0) == -1) {
        perror("semctl IPC_RMID");
        exit(1);
    }
}

int main() {
    int sem_id;

    // Create a semaphore set with one semaphore
    sem_id = semget(SEM_KEY, 1, IPC_CREAT | 0666);
    if (sem_id == -1) {
        perror("semget");
        exit(1);
    }

    // Set semaphore value
    if (!set_semvalue(sem_id)) {
        fprintf(stderr, "Failed to set semaphore value\n");
        exit(1);
    }

    printf("Semaphore value set\n");

    // Delete semaphore
    del_semvalue(sem_id);
    printf("Semaphore deleted\n");

    return 0;
```

```
}
```

Output:

Semaphore value set

Semaphore deleted

CODE (semaphore_p and semaphore_v)

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

#define SEM_KEY 1234 // Key for semaphore set

// Function for semaphore P (wait) operation
void semaphore_p(int sem_id) {
    struct sembuf sem_op;
    sem_op.sem_num = 0; // Semaphore index in the set
    sem_op.sem_op = -1; // P operation (decrement semaphore value)
    sem_op.sem_flg = 0; // No flags
    if (semop(sem_id, &sem_op, 1) == -1) {
        perror("semop P");
        exit(1);
    }
}

// Function for semaphore V (signal) operation
void semaphore_v(int sem_id) {
    struct sembuf sem_op;
    sem_op.sem_num = 0; // Semaphore index in the set
    sem_op.sem_op = 1; // V operation (increment semaphore value)
    sem_op.sem_flg = 0; // No flags
    if (semop(sem_id, &sem_op, 1) == -1) {
        perror("semop V");
```

```
        exit(1);
    }
}
int main() {
    int sem_id;
    // Create a semaphore set with one semaphore
    sem_id = semget(SEM_KEY, 1, IPC_CREAT | IPC_EXCL | 0666);
    if (sem_id == -1) {
        perror("semget");
        exit(1);
    }
    // Set initial semaphore value to 1
    if (semctl(sem_id, 0, SETVAL, 1) == -1) {
        perror("semctl SETVAL");
        exit(1);
    }
    // Perform semaphore P (wait) operation
    semaphore_p(sem_id);
    printf("Semaphore acquired!\n");
    // Perform some critical section operations here
    // Perform semaphore V (signal) operation
    semaphore_v(sem_id);
    printf("Semaphore released!\n");
    // Remove the semaphore set
    if (semctl(sem_id, 0, IPC_RMID, 0) == -1) {
        perror("semctl IPC_RMID");
        exit(1);
    }
    return 0;
}
```

Output:

```
Semaphore acquired!
Semaphore released!
```

Experiment 9

AIM:

Write a Program on pipes (use functions pipe, popen, pclose), named pipes (FIFOs, accessing FIFO).

CODE:

Program 1: Using pipe function

This program showcases how a parent process and a child process communicate using the pipe function:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
   int pipefds[2]; // Array to store file descriptors for the pipe
   pid_t pid;
   // Create the pipe
   if (pipe(pipefds) == -1) {
      perror("pipe");
      exit(EXIT_FAILURE);
   }
   // Fork a child process
   pid = fork();
   if (pid == -1) {
      perror("fork");
      exit(EXIT_FAILURE);
   }
   // Parent process
   if (pid > 0) {
      printf("Parent process writing to pipe...\n");
      // Close the read end of the pipe in the parent
      close(pipefds[0]);
      // Write data to the pipe
      const char *data = "Hello from the parent!\n";
      if (write(pipefds[1], data, strlen(data)) == -1) {
         perror("write");
         exit(EXIT_FAILURE);
```

```
    }
    // Close the write end of the pipe in the parent
    close(pipefds[1]);
    printf("Parent process waiting for child...\n");
    wait(NULL); // Wait for the child process to finish
  } else { // Child process
    printf("Child process reading from pipe...\n");
    // Close the write end of the pipe in the child
    close(pipefds[1]);
    // Read data from the pipe
    char buffer[100];
    int bytes_read = read(pipefds[0], buffer, sizeof(buffer));
    if (bytes_read == -1) {
      perror("read");
      exit(EXIT_FAILURE);
    }
    // Print the data received
    printf("Child process received: %s", buffer);
    // Close the read end of the pipe in the child
    close(pipefds[0]);
  }
  return 0;
}
```

Program 2: Using popen function

This program demonstrates how to use the popen function to execute a command and capture its output:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  FILE *fp;
  char buffer[1024];
  // Open a pipe to the `ls` command
  fp = popen("ls", "r");
  if (fp == NULL) {
```

```
    perror("popen");
    exit(EXIT_FAILURE);
  }
  // Read the output of the command line by line
  while (fgets(buffer, sizeof(buffer), fp) != NULL) {
    printf("%s", buffer);
  }
  // Close the pipe
  pclose(fp);
  return 0;
}
```

Experiment 10

AIM:

Write a Program on Deadlock Prevention, Deadlock Avoidance: Banker's algorithm and Deadlock detection.

CODE:

```
// Banker's Algorithm (Deadlock Prevention and Avoidance)
#include<stdio.h>
int main()
{
  // P0 , P1 , P2 , P3 , P4 are the Process names here
  int n , m , i , j , k;
  n = 5; // Number of processes
  m = 3; // Number of resources
  int alloc[ 5 ] [ 3 ] = { { 0 , 1 , 0 }, // P0 // Allocation Matrix
                { 2 , 0 , 0 } , // P1
                { 3 , 0 , 2 } , // P2
                { 2 , 1 , 1 } , // P3
                { 0 , 0 , 2 } } ; // P4
  int max[ 5 ] [ 3 ] = { { 7 , 5 , 3 } , // P0 // MAX Matrix
              { 3 , 2 , 2 } , // P1
              { 9 , 0 , 2 } , // P2
              { 2 , 2 , 2 } , // P3
              { 4 , 3 , 3 } } ; // P4
```

```
int avail[3] = { 3 , 3 , 2 } ; // Available Resources
int f[n] , ans[n] , ind = 0 ;
for (k = 0; k < n; k++) {
   f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
   for (j = 0; j < m; j++)
      need[i][j] = max[i][j] - alloc[i][j] ;
}
int y = 0;
for (k = 0; k < 5; k++){
   for (i = 0; i < n; i++){
      if (f[i] == 0){
         int flag = 0;
         for (j = 0; j < m; j++) {
            if(need[i][j] > avail[j]){
               flag = 1;
               break;
            }
         }
         if ( flag == 0 ) {
            ans[ind++] = i;
            for (y = 0; y < m; y++)
               avail[y] += alloc[i][y] ;
            f[i] = 1;
         }
      }
   }
}
int flag = 1;
for(int i=0;i<n;i++)
{
```

```
  if(f[i] == 0)
  {
     flag = 0;
     printf(" The following system is not safe ");
     break;
  }
  }
  if (flag == 1)
  {
  printf(" Following is the SAFE Sequence \ n ");
  for (i = 0; i < n - 1; i++)
     printf(" P%d -> " , ans[i]);
  printf(" P%d ", ans[n - 1]);
  }
  return(0);
}
```

Output

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
.....................................................
Process execute din 1.33 seconds
Press any key to continue.
```

SOURCE CODE (Deadlock Detection)

```
#include<stdio.h>
static int mark[20];
int i,j,np,nr;
int main()
{
int alloc[10][10],request[10][10],avail[10],r[10],w[10];
printf("\nEnter the no of process: ");
scanf("%d",&np);
printf("\nEnter the no of resources: ");
scanf("%d",&nr);
```

```
for(i=0;i<nr;i++)
{
printf("\nTotal Amount of the Resource R%d: ",i+1);
scanf("%d",&r[i]);
}
printf("\nEnter the request matrix:");
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&request[i][j]);
printf("\nEnter the allocation matrix:");
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&alloc[i][j]);
/*Available Resource calculation*/
for(j=0;j<nr;j++)
{
avail[j]=r[j];
for(i=0;i<np;i++)
{
avail[j]-=alloc[i][j];
}
}
//marking processes with zero allocation
for(i=0;i<np;i++)
{
int count=0;
 for(j=0;j<nr;j++)
  {
    if(alloc[i][j]==0)
      count++;
    else
      break;
  }
```

```
 if(count==nr)
 mark[i]=1;
}
// initialize W with avail
for(j=0;j<nr;j++)
   w[j]=avail[j];
//mark processes with request less than or equal to W
for(i=0;i<np;i++)
{
int canbeprocessed=0;
 if(mark[i]!=1)
{
  for(j=0;j<nr;j++)
   {
    if(request[i][j]<=w[j])
      canbeprocessed=1;
    else
      {
       canbeprocessed=0;
       break;
       }
   }
if(canbeprocessed)
{
mark[i]=1;
for(j=0;j<nr;j++)
w[j]+=alloc[i][j];
}
}
}
//checking for unmarked processes
int deadlock=0;
for(i=0;i<np;i++)
```

```
if(mark[i]!=1)
deadlock=1;
if(deadlock)
printf("\n Deadlock detected");
else
printf("\n No Deadlock possible");
}
```

OUTPUT:

Enter the no of process: 4

Enter the no of resources: 5

Total Amount of the Resource R1: 2

Total Amount of the Resource R2: 1

Total Amount of the Resource R3: 1

Total Amount of the Resource R4: 2

Total Amount of the Resource R5: 1

Enter the request matrix:0 1 0 0 1

0 0 1 0 1

0 0 0 0 1

1 0 1 0 1

Enter the allocation matrix:1 0 1 1 0

1 1 0 0 0

0 0 0 1 0

0 0 0 0 0

Deadlock detected

Experiment 11

AIM:

Write a Program on Resource Request Algorithm. Wait for Graph.

CODE:

```
#include <stdio.h>
#define MAX_PROCESSES 10
#define MAX_RESOURCES 10
int available[MAX_RESOURCES];
int maximum[MAX_PROCESSES][MAX_RESOURCES];
```

```
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];
// Function prototypes
void request_resources(int pid, int request[]);
int check_request(int pid, int request[]);
int check_safety();
int main() {
   int num_processes, num_resources;
   printf("Enter the number of processes: ");
   scanf("%d", &num_processes);
   printf("Enter the number of resources: ");
   scanf("%d", &num_resources);
   // Initialize available resources
   printf("Enter the number of available instances of each resource:\n");
   for (int i = 0; i < num_resources; i++) {
      scanf("%d", &available[i]);
   }
   // Initialize maximum resources needed by each process
   printf("Enter the maximum number of instances of each resource needed by each process:\n");
   for (int i = 0; i < num_processes; i++) {
      printf("Process %d: ", i);
      for (int j = 0; j < num_resources; j++) {
         scanf("%d", &maximum[i][j]);
         need[i][j] = maximum[i][j];
      }
   }
   // Initialize allocation of resources to each process
   printf("Enter the number of instances of each resource currently allocated to each process:\n");
   for (int i = 0; i < num_processes; i++) {
      printf("Process %d: ", i);
      for (int j = 0; j < num_resources; j++) {
         scanf("%d", &allocation[i][j]);
         need[i][j] -= allocation[i][j];
```

```
        }
    }
    // Perform resource requests and check safety
    int pid, request[MAX_RESOURCES];
    printf("Enter the process ID and the requested resources for that process (-1 to exit):\n");
    while (1) {
        printf("Process ID: ");
        scanf("%d", &pid);
        if (pid == -1)
            break;
        printf("Requested resources: ");
        for (int i = 0; i < num_resources; i++) {
            scanf("%d", &request[i]);
        }
        request_resources(pid, request);
        if (check_safety()) {
            printf("Request granted safely.\n");
        } else {
            printf("Request cannot be granted safely. Try again later.\n");
        }
    }
    return 0;
}
void request_resources(int pid, int request[]) {
    if (check_request(pid, request)) {
        for (int i = 0; i < MAX_RESOURCES; i++) {
            available[i] -= request[i];
            allocation[pid][i] += request[i];
            need[pid][i] -= request[i];
        }
    } else {
        printf("Invalid request.\n");
    }
```

```
}
int check_request(int pid, int request[]) {
    for (int i = 0; i < MAX_RESOURCES; i++) {
        if (request[i] > need[pid][i] || request[i] > available[i])
            return 0;
    }
    return 1;
}
int check_safety() {
    int work[MAX_RESOURCES];
    int finish[MAX_PROCESSES] = {0};
    // Initialize work to available
    for (int i = 0; i < MAX_RESOURCES; i++) {
        work[i] = available[i];
    }
    // Check if process can finish
    for (int i = 0; i < MAX_PROCESSES; i++) {
        if (!finish[i]) {
            int can_allocate = 1;
            for (int j = 0; j < MAX_RESOURCES; j++) {
                if (need[i][j] > work[j]) {
                    can_allocate = 0;
                    break;
                }
            }
            if (can_allocate) {
                for (int j = 0; j < MAX_RESOURCES; j++) {
                    work[j] += allocation[i][j];
                }
                finish[i] = 1;
                i = -1; // Start from the beginning again
            }
        }
```

```
    }
    // If all processes can finish, return true
    for (int i = 0; i < MAX_PROCESSES; i++) {
        if (!finish[i])
            return 0;
    }
    return 1;
}
```

Experiment 12

AIM:

Write a Program on creating a script.

CODE:

Open the terminal. Go to the directory where you want to create your script.

Create a file with .sh extension.

Write the script in the file using an editor.

Make the script executable with command chmod +x <fileName>.

Run the script using ./<fileName>.

```
#!/bin/sh
echo "what is your name?"
read name
echo "How do you do, $name?"
read remark
echo "I am $remark too!"
```

Experiment 13

AIM:

Write a Program on making a script executable.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    // Execute the script with output capture
    FILE *fp = popen("sh script.sh", "r");
    if (fp == NULL) {
```

```
    perror("popen");
    return 1;
  }
  char buffer[1024];
  while (fgets(buffer, sizeof(buffer), fp) != NULL) {
    printf("%s", buffer);
  }
  pclose(fp);
  return 0;
}
```

Experiment 14

AIM:

Write a Program on shell syntax (variables, conditions).

CODE:

Example 1:

Implementing if.else statement

```
#Initializing two variables
a=20
b=20
if [ $a == $b ]
then
  #If they are equal then print this
  echo "a is equal to b"
else
  #else print this
  echo "a is not equal to b"
fi
```

Example 2:

Implementing switch statement

```
CARS="bmw"
#Pass the variable in string
case "$CARS" in
  #case 1
```


```
  "mercedes") echo "Headquarters - Affalterbach, Germany" ;;
  #case 2
  "audi") echo "Headquarters - Ingolstadt, Germany" ;;
  #case 3
  "bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;
esac
```

Experiment 15

AIM:

Write a Program on shell syntax (control structures, functions, commands).

CODE:

```
# Define your function here
Hello () {
  echo "Hello World"
}
# Invoke your function
Hello
# Define your function here
Hello () {
  echo "Hello World $1 $2"
}
# Invoke your function
Hello Zara Ali
```