Slip 1

Q1. Write a program that demonstrates the use of nice() system call. After a child process is started using fork(), assign higher priority to the child using nice() system call.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
   pid_t pid;
   pid = fork();
   if(pid == 0)
   {
      //child process
      printf("\n I am child process, id=%d",getpid());
      printf("\n Priority : %d, id=%d\n",nice(-7));
   }
   else
   {
      //parent process
      printf("\n I am parent process, id=%d\n",getpid());
      printf("\n Priority : %d, id=%d\n",nice(15),getpid());
   }
   return 0;
}
```

Q3. Write a C program to simulate Banker's algorithm for the purpose of deadlock avoidance. Consider the following snapshot of system, A, B, C and D are the resource type. a) Calculate and display the content of need matrix? b) Is the system in safe state? If display the safe sequence. c) If a request from process P arrives for (0, 4, 2, 0) can it be granted immediately by keeping the system in safe state. Print a message

```
#include <stdio.h>
#define P 5  // Number of processes
#define R 4  // Number of resource types

void calculateNeed(int need[P][R], int max[P][R], int allocation[P][R]) {
   for (int i = 0; i < P; i++) {
      for (int j = 0; j < R; j++) {
         need[i][j] = max[i][j] - allocation[i][j];
      }
   }
}

int isSafe(int processes[], int available[], int max[][R], int allocation[][R]) {
   int need[P][R];
```

```
    calculateNeed(need, max, allocation);

    int finish[P] = {0};
    int safeSeq[P];
    int work[R];

    for (int i = 0; i < R; i++)
        work[i] = available[i];

    int count = 0;
    while (count < P) {
        int found = 0;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                if (j == R) {
                    for (int k = 0; k < R; k++)
                        work[k] += allocation[p][k];

                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = 1;
                }
            }
        }

        if (found == 0) {
            printf("System is not in a safe state\n");
            return 0;
        }
    }

    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < P; i++)
        printf("%d ", safeSeq[i]);
    printf("\n");
    return 1;
}

void requestResources(int process, int request[], int available[], int allocation[][R], int max[][R]) {
    int need[P][R];
    calculateNeed(need, max, allocation);

    for (int i = 0; i < R; i++) {
        if (request[i] > need[process][i]) {
            printf("Error: Process has exceeded its maximum claim.\n");
            return;
        }
        if (request[i] > available[i]) {
            printf("Process must wait, resources are not available.\n");
            return;
        }
    }

    for (int i = 0; i < R; i++) {
        available[i] -= request[i];
        allocation[process][i] += request[i];
        need[process][i] -= request[i];
    }

    if (isSafe((int[]){0, 1, 2, 3, 4}, available, max, allocation)) {
        printf("Request can be granted immediately.\n");
    } else {
        printf("Request cannot be granted immediately. Rolling back.\n");
        for (int i = 0; i < R; i++) {
            available[i] += request[i];
            allocation[process][i] -= request[i];
            need[process][i] += request[i];
        }
    }
}

int main() {
    int processes[] = {0, 1, 2, 3, 4};

    int allocation[P][R] = {{0, 0, 1, 2},
                            {1, 0, 0, 0},
                            {1, 3, 5, 4},
                            {0, 6, 3, 2},
                            {0, 0, 1, 4}};

    int max[P][R] = {{2, 0, 1, 2},
                     {1, 7, 5, 0},
                     {2, 3, 5, 6},
                     {0, 6, 5, 2},
                     {0, 6, 5, 6}};

    int available[R] = {1, 5, 2, 0};

    isSafe(processes, available, max, allocation);

    int request[] = {0, 4, 2, 0};
    requestResources(1, request, available, allocation, max);

    return 0;
}
```

Slip 2

Q.1 Create a child process using fork(), display parent and child process id. Child process will display the message “Hello World” and the parent process should display “Hi”.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
    int pid;
    pid=fork();
    if(pid<0)  //-ve value error
    {
        printf("\n Error in creation of child process");
        exit(1);
    }
    else if(pid==0)  //child process
    {
        printf("\n Hello World");
        exit(0);
    }
    else
    {
        printf("\n Hi");
        exit(1);
    }
}
```

Q.2 Write the simulation program using SJF (non-preemptive). The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times.

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

struct job
{
    int atime;  //arrival time
    int btime;  //brust time
    int ft;  //finish time
    int tat;  //turnaround time
    int wt;  //waiting time
}p[10];

int arr[10],brust[10],n,rq[10],no_rq=0,time=0;

void main()
{
    int i,j;
    printf("Enter the job number : ");
    scanf("%d",&n);
    printf("\n");

    for(i=0;i<n;i++)
    {
        printf("Enter the arrival time p%d : ",i);
        scanf("%d",&p[i].atime);  //assign arrival time
        arr[i]=p[i].atime;
    }
    printf("\n");

    for(i=0;i<n;i++)
    {
        printf("Enter the arrival time p%d : ",i);
        printf("%d\n",p[i].atime);  //printing arrival time
        arr[i]=p[i].atime;
    }
    printf("\n");

    for(i=0;i<n;i++)
    {
        printf("Enter the brust time p%d:",i);
        scanf("%d",&p[i].btime);  //assigning brust time
        brust[i]=p[i].btime;
    }
    printf("\n");

    for(i=0;i<n;i++)
    {
        printf("Enter the brust time p%d:",i);
        printf("%d\n",p[i].btime);  //printing brust time
        brust[i]=p[i].btime;
    }
    printf("\n");

    addrq();  //adding process
    //process start now
    printf("Gantt Chart is : ");
    while(1)
    {
        j=selectionjob();  //search process now
        if(j==-1)
        {
            printf("CPU is dead");
            time++;
            addrq();
        }
        else
        {
            while(brust[j]!=0)
            {
                printf("\t j %d",j);
                brust[j]--;
                time++;
                addrq();
            }
            p[j].ft=time;
        }
        if(fsahll()==1)
        break;
    }

    int Tat=0,Twt=0;

    printf("\n");
    printf("\nJob\tFT\tTAT\tWT");
    for(i=0;i<n;i++)
    {
        p[i].tat=p[i].ft-p[i].atime;
        p[i].wt=p[i].tat-p[i].btime;

        printf("\n Job %d \t %d \t %d \t %d",i,p[i].ft,p[i].tat,p[i].wt);
        Tat+=p[i].tat;
        Twt+=p[i].wt;
    }

    float avgtat=Tat/n;
    float avgwt=Twt/n;

    printf("\n Average trunaround time is : %f",avgtat);
    printf("\n Average waiting time is : %f",avgwt);
}

int addrq()
{
    int i;
    for(i=0;i<n;i++)
    {
        if(arr[i]==time)
        {
            rq[no_rq]=i;
            no_rq++;
        }
    }
}

int selectionjob()
{
    int i,j;
    if(no_rq==0)
    return 1;
    j=rq[0];

    for(i=1;i<no_rq;i++)
    if(brust[j]>brust[rq[i]])
    j=rq[i];

    deleteq(j);
    return j;
}

int deleteq(int j)
{
    int i;
    for(i=0;i<no_rq;i++)

    if(rq[i]==j)
    break;

    for(i=i+1;i<no_rq;i++)
    rq[i-1]=rq[i];
    no_rq--;
}

int fsahll()
{
    int i;
    for(i=0;i<n;i++)
    if(brust[i]!=0)
    return -1;
    return 1;
}
```

Slip 3

Q. 1 Creating a child process using the command exec(). Note down process ids of the parent and the child processes, check whether the control is given back to the parent after the child process terminates.

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

int main()
{
   pid_t pid;
   //create child process
   pid = fork();

   if(pid<0)
   {
      perror("Fork Failed");
      exit(1);
   }
   else if(pid==0)
   {
      //code executed by child process
      printf("Child Process : My PID is %d \n",getpid());
      printf("Child Process : Executing 'ls' command...\n");
      //execute ls command is child process
      execlp("ls", "ls", (char*)NULL);
      //if execlp fails it will print error msg
      perror("execlp");
      exit(1);
   }
   else
   {
      //code executed by parent process
      printf("Parent Process : My PID is %d \n",getpid());
      printf("Parent Process : Waiting for the child to finish...\n");
      //wait for child process to terminate
      wait(NULL);
      printf("Parent Process : Child has finished \n");
   }
   return 0;
}
```

Q2. Write the simulation program using FCFS. The arrival time and first CPU bursts of different jobs should be input to the system. Assume the fixed I/O waiting time (2 units). The next CPU burst should be generated using random function. The output should give the Gantt chart, Turnaround Time and Waiting time for each process and average times.

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

//structure to represent a job
typedef struct
{
   int arrival_time;
   int first_cpu_burst;
   int next_cpu_burst;
   int turnaround_time;
   int waiting_time;
}Job;

//function to calculate average turnaround & waiting time
void calculateAverages(Job jobs[], int n, float *avg_turnaround, float *avg_waiting)
{
   int total_turnaround = 0;
   int total_waiting = 0;

   for(int i=0;i<n;i++)
   {
      total_turnaround += jobs[i].turnaround_time;
      total_waiting += jobs[i].waiting_time;
   }
   *avg_turnaround = (float)total_turnaround/n;
   *avg_waiting = (float)total_waiting/n;
}

int main()
{
   int n; //number of jobs
   srand(time(NULL)); //send the random no. generator with the current time
   printf("Enter the number of jobs : ");
   scanf("%d",&n);

   Job jobs[n];

   //input arrival time & first CPU burst for each job
   for(int i=0;i<n;i++)
   {
      printf("Job %d\n",i+1);
      printf("Arrival Time : ");
```

```
        scanf("%d",&jobs[i].arrival_time);
        printf("First CPU Burst Time : ");
        scanf("%d",&jobs[i].first_cpu_burst);

        //initialize other job attributes
        jobs[i].next_cpu_burst=0;
        jobs[i].turnaround_time=0;
        jobs[i].waiting_time=0;
    }

    int current_time=0;

    //process jobs in FCFS order
    for(int i=0;i<n;i++)
    {
        //wait for the job to arrive
        if(current_time < jobs[i].arrival_time)
        {
            current_time = jobs[i].arrival_time;
        }

        //update next CPU burst using a random function (e.g., rand())
        jobs[i].next_cpu_burst = rand() % 10 + 1; //generating random burst time between 1 & 10 units

        //calculate turnaround & waiting times
        {
            jobs[i].turnaround_time = current_time + jobs[i].first_cpu_burst + jobs[i].next_cpu_burst -
jobs[i].arrival_time;
            jobs[i].waiting_time = jobs[i].turnaround_time - jobs[i].first_cpu_burst -
jobs[i].next_cpu_burst;

            //update the current time
            current_time += jobs[i].first_cpu_burst + jobs[i].next_cpu_burst;
        }

        //calculate & display the Gantt chart
        printf("\n Gantt Chart : \n");
        printf("0");
        for(int i=0;i<n;i++)
        {
            printf(" -> Job %d -> %d", i+1, current_time);
        }
        printf("\n");

        //display turnaround & waiting times for each job
        printf("\n Turnaround Time and Waiting Time : \n");
        for(int i=0;i<n;i++)
        {
            printf("Job %d - Turnaround Time : %d, Waiting Time : %d\n",i+1,jobs[i].turnaround_time,
jobs[i].waiting_time);
        }

        //calculate & display average turnaround & waiting time
        float avg_turnaround, avg_waiting;
        calculateAverages(jobs, n, &avg_turnaround, &avg_waiting);
        printf("\n Average Turnaround Time : %.2fn", avg_turnaround);
        printf("\n Average Waiting Time : %.2fn", avg_waiting);
    }
    return 0;
}
```

Slip 4

Q1. Write a program to illustrate the concept of orphan process (Using fork() and sleep())

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

int main()
{
    int pid = fork();
    if(pid>0)
    {
        printf("Parent process \n");
        printf("ID : %d \n \n",getpid());
    }
    else if(pid == 0)
    {
        printf("Child process \n");
        printf("ID : %d \n",getpid());
        sleep(10);
        printf("\nChild process\n");
        printf("ID : %d \n",getpid());
    }
    else
    {
        printf("Failed to create child process");
    }
    return 0;
}
```

Q2. Write the program to simulate Non-preemptive Priority scheduling. The arrival time and first CPU burst and priority for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

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

struct job
{
	int atime; // arraival time.
	int btime; //brust time.
	int ft; //finish time.
	int tat; //Turn around time.
	int wt; // waiting time.
	int pri; //Priority varilable is add.
}p[10];

int arr[10],brust[10],n,rq[10],no_rq=0,time=0,j=-1;

void main()
{
	int i,j;
	printf("Enter the job number:");
	scanf("%d",&n);
	printf("\n");

	for(i = 0;i<n;i++)
	{
		printf("Enter the arrival time p%d:",i);
		scanf("%d",&p[i].atime); //Assigning the arrival time.
		arr[i] = p[i].atime;
	}
	printf("\n");

	for(i = 0;i<n;i++)
	{
	printf("Enter the arrival time p%d:",i);
	printf("%d",p[i].atime); //Assigning the arrival time.
	arr[i] = p[i].atime;
	}
	printf("\n");

	for(i = 0;i<n;i++)
	{
	printf("Enter the brust time p%d:",i);
	scanf("%d",&p[i].btime); //Assigning the brust time.
	brust[i] = p[i].btime;
	}
	printf("\n");
```

```
for(i = 0;i<n;i++)
{
printf("Enter the brust time p%d:",i);
printf("%d",p[i].btime); //Assigning the brust time.
brust[i] = p[i].btime;
}
printf("\n");

for(i = 0;i<n;i++)
{
printf("Enter the Priority p%d:",i);
scanf("%d",&p[i].pri); //Assigning the Priority.
}
printf("\n");

for(i = 0;i<n;i++)
{
printf("Enter the Priority p%d:",i);
printf("%d",p[i].pri); //Assigning the Priority.
}
printf("\n");


addrq();  //Adding the process.
// Process start now.
while(1)
{
   j = selectionjob(); //sercah the process now.

            if(j == -1)
            {
                        printf("CPU is ideal");
                        time++;
                        addrq();
            }
            else
            {
                        while(brust[j]!=0)
                        {
                                    printf("\t j %d",j);
                                    brust[j]--;
                                    time++;
                                    addrq();
                        }
                        p[j].ft = time;
            }
            if(fsahll() == 1)
```

```
                break;
        }
        int Tat = 0,Twt =0;

        printf("\nJob \t FT \t TAT \t WT");
        for(i=0;i<n;i++)
        {
                p[i].tat = p[i].ft-p[i].atime;
                p[i].wt = p[i].tat-p[i].btime;

                printf("\n JOb %d \t %d \t %d \t %d",i,p[i].ft,p[i].tat,p[i].wt);
                Tat += p[i].tat;
                Twt += p[i].wt;
        }

        float avgtat = Tat /n;
        float avgwt = Twt /n;

        printf("\nAverage of trun around time is:%f",avgtat);
        printf("\nAverage of wait time is:%f",avgwt);
}

int addrq()
{
        int i;
        for(i=0;i<n;i++)
        {
          if(arr[i] == time)
                {
                        if(j!=-1 && p[i].pri>p[j].pri)
                        {
                           rq[no_rq] = i;
                           j = i;
                        }
                        else
                        {
                             rq[no_rq++] = i;
                        }
                }
   }
}

int selectionjob()
{
        int i,k;
        if(no_rq == 0)
                return -1;
                k = rq[0];
```

```
            for(i=1;i<no_rq;i++)
            if(p[k].pri<p[rq[i]].pri)
            {
                        k = rq[i];
            }
            deleteq(k);
            return k;
}

int deleteq(int k)
{
            int i;
            for(i=0;i<no_rq;i++)

                        if(rq[i] == k)
                                    break;

                        for(i= i+1;i<no_rq;i++)
                        rq[i-1] = rq[i];
                        no_rq--;
}

int fsahll()
{
            int i;
            for(i=0;i<n;i++)
            if(brust[i]!=0)
            return -1;
            return 1;
}
```

Slip 5

Q1. Write a program that demonstrates the use of nice () system call. After a child process is started using fork (), assign higher priority to the child using nice () system call.

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

int main()
{
   pid_t pid;
```

```
    // Fork a child process
    pid = fork();

    if (pid < 0)
    {  // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0)
    {  // Child process
        printf("Child process (PID: %d) before changing priority\n", getpid());

        // Set a higher priority (decreasing the nice value)
        int new_nice_value = nice(-5);  // Decrease nice value by 5

        if (new_nice_value == -1 && errno != 0)
        {
            perror("Nice failed");
            exit(EXIT_FAILURE);
        }

        printf("Child process (PID: %d) after changing priority. New nice value: %d\n", getpid(),
new_nice_value);

        // Simulate some work
        for (int i = 0; i < 5; i++)
        {
            printf("Child process (PID: %d) is working...\n", getpid());
            sleep(1);
        }

        exit(EXIT_SUCCESS);
    }
    else
    {  // Parent process
        printf("Parent process (PID: %d)\n", getpid());

        // Wait for the child process to complete
        wait(NULL);

        printf("Parent process (PID: %d) after child has completed.\n", getpid());
    }

    return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames. Reference String: 3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6. Implement FIFO

```
#include<stdio.h>

int main()
{
int page_reference_string[]={3,4,5,6,3,4,7,3,4,5,6,7,2,4,6};
int n,i,j;
printf("Enter the numbers of memory frames:");
scanf("%d",&n);
int memory_frames[n];
int page_queue[n];
int page_faults=0;

for(i=0;i<n;i++)
{
 memory_frames[i]=-1;
page_queue[i]=-1;
}

for(i=0;i<sizeof(page_reference_string)/sizeof(page_reference_string[0]);i++)
{
 int page=page_reference_string[i];
 int found=0;
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==page)
{
 found=1;
 break;
}
}

if(!found)
{
 page_faults++;

 if(page_queue[0]!=-1)
{
 int replaced_page=page_queue[0];
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==replaced_page)
{
  memory_frames[j]=page;
 break;
}
}
for(j=0;j<n-1;j++)
{
 page_queue[j]=page_queue[j+1];
}
}
else
{
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==-1)
{
 memory_frames[j]=page;
break;
}
}
}
for(j=0;j<n-1;j++)
{
 page_queue[j]=page_queue[j+1];
}
page_queue[n-1]=page;
}
printf("Page reference:%d\n",page);
printf("Memory Frames:");
for(j=0;j<n;j++)
{
 printf("%d",memory_frames[j]);
}
printf("\n Page Queue:");
 for(j=0;j<n;j++)
{
 printf("%d",page_queue[j]);
}
printf("\n\n");
}
printf("Total Page Faults:%d\n",page_faults);
return 0;
}
```

Slip 6

Q1. Write a program to find the execution time taken for execution of a given set of instructions (use clock() function)

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

```
#include<time.h>
int main()
{
   int i;
   clock_t start_time = clock();
   for(i=1;i<1000000;i++)
   {

   }
   clock_t end_time = clock();
   double execution_time = (double)(end_time-start_time)/CLOCKS_PER_SEC;
   printf("Execution time : %f seconds\n",execution_time);
   return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames.
Reference String :3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6
Implement FIFO

```
#include<stdio.h>

int main()
{
int page_reference_string[]={3,4,5,6,3,4,7,3,4,5,6,7,2,4,6};
int n,i,j;
printf("Enter the numbers of memory frames:");
scanf("%d",&n);
int memory_frames[n];
int page_queue[n];
int page_faults=0;

for(i=0;i<n;i++)
{
 memory_frames[i]=-1;
page_queue[i]=-1;
}

for(i=0;i<sizeof(page_reference_string)/sizeof(page_reference_string[0]);i++)
{
 int page=page_reference_string[i];
 int found=0;
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==page)
{
 found=1;
 break;
}
}

if(!found)
{
 page_faults++;

 if(page_queue[0]!=-1)
{
 int replaced_page=page_queue[0];
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==replaced_page)
{
  memory_frames[j]=page;
 break;
}
}
for(j=0;j<n-1;j++)
{
 page_queue[j]=page_queue[j+1];
}
}
else
{
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==-1)
{
 memory_frames[j]=page;
break;
}
}
}
for(j=0;j<n-1;j++)
{
 page_queue[j]=page_queue[j+1];
}
page_queue[n-1]=page;
}
printf("Page reference:%d\n",page);
printf("Memory Frames:");
for(j=0;j<n;j++)
{
 printf("%d",memory_frames[j]);
}
printf("\n Page Queue:");
```

```
 for(j=0;j<n;j++)
{
 printf("%d",page_queue[j]);
}
printf("\n\n");
}
printf("Total Page Faults:%d\n",page_faults);
return 0;
}
```

Slip 7

Q1. Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

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

int main()
{
   pid_t pid;
   // Fork a child process
   pid = fork();

   if (pid < 0)
   {  // Fork failed
      perror("Fork failed");
      exit(EXIT_FAILURE);
   }
   else if (pid == 0)
   {  // Child process
      printf("Child process (PID: %d) is executing 'ls' command using execl().\n", getpid());

      // Execute the "ls" command
      execl("/bin/ls", "ls", "-l", (char *)NULL);

      // If execl() fails
      perror("execl() failed");
      exit(EXIT_FAILURE);
   }
   else {  // Parent process
      printf("Parent process (PID: %d) is going to sleep.\n", getpid());
```

```
        // Parent process goes to sleep
        sleep(5);

        // Wait for the child process to complete
        wait(NULL);

        printf("Parent process (PID: %d) has woken up and child process completed.\n", getpid());
    }
    return 0;
}
```

Q3. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames. Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Implement LRU

```
#include <stdio.h>

int findLRU(int time[], int n) {
    int min = time[0], pos = 0, i;
    for (i = 1; i < n; ++i) {
        if (time[i] < min) {
            min = time[i];
            pos = i;
        }
    }
    return pos;
}

int lru_page_replacement(int pages[], int n, int frames) {
    int memory_frames[frames], time[frames], counter = 0, page_faults = 0;
    int i, j, pos;

    // Initialize memory frames and time array
    for (i = 0; i < frames; i++) {
        memory_frames[i] = -1;
        time[i] = 0;
    }

    // Processing the reference string
    for (i = 0; i < n; i++) {
        int found = 0;

        // Check if the page is already in the memory frames
        for (j = 0; j < frames; j++) {
            if (memory_frames[j] == pages[i]) {
                counter++;
                time[j] = counter; // Update time for the recently used page
                found = 1;
                break;
            }
        }

        if (!found) {
            // If the page is not found, handle the page fault
            page_faults++;
            if (i < frames) {
                // Insert page if there's still room
                memory_frames[i] = pages[i];
                counter++;
                time[i] = counter;
            } else {
                // Find the least recently used page and replace it
                pos = findLRU(time, frames);
                memory_frames[pos] = pages[i];
                counter++;
                time[pos] = counter;
            }
        }

        // Print the current state of the memory frames
        printf("Page %d -> Memory: ", pages[i]);
        for (j = 0; j < frames; j++) {
            if (memory_frames[j] == -1)
                printf("- ");
            else
                printf("%d ", memory_frames[j]);
        }
        printf("\n");
    }

    return page_faults;
}

int main() {
    int frames, page_faults;
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2}; // Reference string
    int n = sizeof(pages) / sizeof(pages[0]); // Number of pages

    // Input number of memory frames
    printf("Enter the number of memory frames: ");
    scanf("%d", &frames);

    // Call LRU page replacement and get the number of page faults
    page_faults = lru_page_replacement(pages, n, frames);
    // Output the total page faults
    printf("Total Page Faults: %d\n", page_faults);

    return 0;
}
```

Slip 8

Q1. Write a C program to accept the number of process and resources and find the need matrix content and display it.

```
#include <stdio.h>

int main()
{
    int num_processes, num_resources;

    // Accept the number of processes and resources
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);

    printf("Enter the number of resources: ");
    scanf("%d", &num_resources);

    int max[num_processes][num_resources];
    int allocation[num_processes][num_resources];
    int need[num_processes][num_resources];

    // Accept the Max matrix
    printf("Enter the Max matrix (%d x %d):\n", num_processes, num_resources);
    for (int i = 0; i < num_processes; i++)
    {
        for (int j = 0; j < num_resources; j++)
        {
            scanf("%d", &max[i][j]);
        }
    }

    // Accept the Allocation matrix
    printf("Enter the Allocation matrix (%d x %d):\n", num_processes, num_resources);

    for (int i = 0; i < num_processes; i++)
    {
        for (int j = 0; j < num_resources; j++)
        {
            scanf("%d", &allocation[i][j]);
        }
    }

    // Calculate the Need matrix
    for (int i = 0; i < num_processes; i++)
    {
        for (int j = 0; j < num_resources; j++)
        {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }

    // Display the Need matrix
    printf("The Need matrix is:\n");
    for (int i = 0; i < num_processes; i++)
    {
        for (int j = 0; j < num_resources; j++)
        {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Q2. AI

Q3. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n =3 as the number of memory frames. Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8 Implement OPT

```
#include <stdio.h>

#define MAX 100

// Function to find the index of the page to be replaced using the Optimal algorithm
int find_optimal(int frames[], int ref_string[], int ref_size, int current_pos, int num_frames) {
    int i, j, farthest = current_pos, page_index = -1;

    for (i = 0; i < num_frames; i++) {
        int found = 0;
        for (j = current_pos + 1; j < ref_size; j++) {
            if (frames[i] == ref_string[j]) {
                if (j > farthest) {
                    farthest = j;
                    page_index = i;
                }
                found = 1;
                break;
            }
        }

        if (!found) {
            return i;
        }
    }

    return (page_index == -1) ? 0 : page_index;
}

// Function to implement the Optimal Page Replacement algorithm
void optimal_page_replacement(int ref_string[], int ref_size, int num_frames) {
    int frames[num_frames];
    int i, j, page_faults = 0;

    // Initialize frames with -1 to indicate empty slots
    for (i = 0; i < num_frames; i++) {
        frames[i] = -1;
    }

    printf("Page scheduling:\n");
    for (i = 0; i < ref_size; i++) {
        int page = ref_string[i];
        int found = 0;

        // Check if the page is already in the frames
        for (j = 0; j < num_frames; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurs
            page_faults++;

            if (i < num_frames) {
                // If there is space in frames, insert the page into the next empty frame
                frames[i] = page;
            } else {
                // Replace a page using the Optimal algorithm
                int replace_index = find_optimal(frames, ref_string, ref_size, i, num_frames);
                frames[replace_index] = page;
            }
            // Print the current state of frames
            printf("Page %d: ", page);
            for (j = 0; j < num_frames; j++) {
                if (frames[j] == -1)
                    printf("[ ] ");
                else
                    printf("[%d] ", frames[j]);
            }
            printf("\n");
        }
    }

    printf("Total number of page faults: %d\n", page_faults);
}

int main() {
    int ref_string[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int num_frames = 3;
    int ref_size = sizeof(ref_string) / sizeof(ref_string[0]);

    optimal_page_replacement(ref_string, ref_size, num_frames);

    return 0;
}
```

Slip 9

Q1. Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

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

int main()
{
    pid_t pid;
    // Fork a child process
    pid = fork();

    if (pid < 0)
    {  // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0)
    {  // Child process
        printf("Child process (PID: %d) is executing 'ls' command using execl().\n", getpid());

        // Execute the "ls" command
        execl("/bin/ls", "ls", "-l", (char *)NULL);

        // If execl() fails
        perror("execl() failed");
        exit(EXIT_FAILURE);
    }
    else {  // Parent process
        printf("Parent process (PID: %d) is going to sleep.\n", getpid());

        // Parent process goes to sleep
        sleep(5);

        // Wait for the child process to complete
        wait(NULL);

        printf("Parent process (PID: %d) has woken up and child process completed.\n", getpid());
    }
    return 0;
}
```

Q3. Write the program to simulate Round Robin (RR) scheduling. The arrival time and first CPU- burst for different n number of processes should be input to the algorithm. Also give the time quantum as input. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

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

#define MAX 100

typedef struct {
    int id, arrival, burst, remaining, waiting, turnaround;
} Process;

typedef struct {
    int processId, startTime, endTime;
} GanttChart;

int main() {
```

```
int n, quantum, time = 0, completed = 0;
Process p[MAX];
GanttChart gantt[MAX];
int ganttIndex = 0;  // Track the Gantt chart index

printf("Enter number of processes: ");
scanf("%d", &n);
printf("Enter time quantum: ");
scanf("%d", &quantum);

for (int i = 0; i < n; i++) {
   p[i].id = i + 1;
   printf("Enter arrival time and CPU burst for P%d: ", p[i].id);
   scanf("%d %d", &p[i].arrival, &p[i].burst);
   p[i].remaining = p[i].burst;
   p[i].waiting = 0;
   p[i].turnaround = 0;
}

printf("\nGantt Chart:\n");

while (completed < n) {
   bool idle = true;
   for (int i = 0; i < n; i++) {
      if (p[i].remaining > 0 && p[i].arrival <= time) {
         idle = false;
         int execTime = (p[i].remaining > quantum) ? quantum : p[i].remaining;
         p[i].remaining -= execTime;

         // Store Gantt chart details
         gantt[ganttIndex].processId = p[i].id;
         gantt[ganttIndex].startTime = time;
         gantt[ganttIndex].endTime = time + execTime;
         ganttIndex++;

         time += execTime;

         if (p[i].remaining == 0) {
            p[i].turnaround = time - p[i].arrival;
            p[i].waiting = p[i].turnaround - p[i].burst;
            completed++;
         }
      }
   }
   if (idle) time++;  // Handle idle time
}

// Print the Gantt chart
```

```
    for (int i = 0; i < ganttIndex; i++) {
        printf("| P%d (%d-%d) ", gantt[i].processId, gantt[i].startTime, gantt[i].endTime);
    }
    printf("|\n\n");

    // Process results
    printf("Process\tArrival\tBurst\tWaiting\tTurnaround\n");

    float totalWait = 0, totalTurnaround = 0;
    for (int i = 0; i < n; i++) {
        totalWait += p[i].waiting;
        totalTurnaround += p[i].turnaround;
        printf("P%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].arrival, p[i].burst, p[i].waiting, p[i].turnaround);
    }

    printf("\nAverage waiting time: %.2f\n", totalWait / n);
    printf("Average turnaround time: %.2f\n", totalTurnaround / n);

    return 0;
}
```

Slip 10

Q1. Write a program to illustrate the concept of orphan process (Using fork() and sleep())

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

int main()
{
    int pid = fork();
    if(pid>0)
    {
        printf("Parent process \n");
        printf("ID : %d \n \n",getpid());
    }
    else if(pid == 0)
    {
        printf("Child process \n");
        printf("ID : %d \n",getpid());
        sleep(10);
        printf("\nChild process\n");
        printf("ID : %d \n",getpid());
    }
    else
    {
        printf("Failed to create child process");
    }
    return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n=3 as the number of memory frames. Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8 Implement OPT

```
#include <stdio.h>

#define MAX 100

// Function to find the index of the page to be replaced using the Optimal algorithm
int find_optimal(int frames[], int ref_string[], int ref_size, int current_pos, int num_frames) {
    int i, j, farthest = current_pos, page_index = -1;

    for (i = 0; i < num_frames; i++) {
        int found = 0;
        for (j = current_pos + 1; j < ref_size; j++) {
            if (frames[i] == ref_string[j]) {
                if (j > farthest) {
                    farthest = j;
                    page_index = i;
                }
                found = 1;
                break;
            }
        }

        if (!found) {
            return i;
        }
    }

    return (page_index == -1) ? 0 : page_index;
}

// Function to implement the Optimal Page Replacement algorithm
void optimal_page_replacement(int ref_string[], int ref_size, int num_frames) {
    int frames[num_frames];
    int i, j, page_faults = 0;

    // Initialize frames with -1 to indicate empty slots
    for (i = 0; i < num_frames; i++) {
        frames[i] = -1;
    }
    printf("Page scheduling:\n");
    for (i = 0; i < ref_size; i++) {
        int page = ref_string[i];
        int found = 0;

        // Check if the page is already in the frames
        for (j = 0; j < num_frames; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurs
            page_faults++;

            if (i < num_frames) {
                // If there is space in frames, insert the page into the next empty frame
                frames[i] = page;
            } else {
                // Replace a page using the Optimal algorithm
                int replace_index = find_optimal(frames, ref_string, ref_size, i, num_frames);
                frames[replace_index] = page;
            }

            // Print the current state of frames
            printf("Page %d: ", page);
            for (j = 0; j < num_frames; j++) {
                if (frames[j] == -1)
                    printf("[ ] ");
                else
                    printf("[%d] ", frames[j]);
            }
            printf("\n");
        }
    }

    printf("Total number of page faults: %d\n", page_faults);
}

int main() {
    int ref_string[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int num_frames = 3;
    int ref_size = sizeof(ref_string) / sizeof(ref_string[0]);

    optimal_page_replacement(ref_string, ref_size, num_frames);
```

```
   return 0;
}
```

Slip 11

Q1. Create a child process using fork(), display parent and child process id. Child process will display the message “Hello World” and the parent process should display “Hi”.

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

int main()
{
   int pid;
   pid=fork();
   if(pid<0)  //-ve value error
   {
      printf("\n Error in creation of child process");
      exit(1);
   }
   else if(pid==0)  //child process
   {
      printf("\n Hello World");
      exit(0);
   }
   else
   {
      printf("\n Hi");
      exit(1);
   }
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames. Reference String: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1 Implement FIFO

```
#include<stdio.h>

int main()
{
int page_reference_string[]={3,4,5,6,3,4,7,3,4,5,6,7,2,4,6};
int n,i,j;
printf("Enter the numbers of memory frames:");
scanf("%d",&n);
int memory_frames[n];
int page_queue[n];
int page_faults=0;

for(i=0;i<n;i++)
{
 memory_frames[i]=-1;
page_queue[i]=-1;
}

for(i=0;i<sizeof(page_reference_string)/sizeof(page_reference_string[0]);i++)
{
 int page=page_reference_string[i];
 int found=0;
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==page)
{
 found=1;
 break;
}
}

if(!found)
{
 page_faults++;

 if(page_queue[0]!=-1)
{
 int replaced_page=page_queue[0];
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==replaced_page)
{
  memory_frames[j]=page;
 break;
}
}
for(j=0;j<n-1;j++)
{
 page_queue[j]=page_queue[j+1];
}
}
else
{
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==-1)
{
 memory_frames[j]=page;
break;
}
}
}
for(j=0;j<n-1;j++)
{
 page_queue[j]=page_queue[j+1];
}
page_queue[n-1]=page;
}
printf("Page reference:%d\n",page);
printf("Memory Frames:");
for(j=0;j<n;j++)
{
 printf("%d",memory_frames[j]);
}
printf("\n Page Queue:");
 for(j=0;j<n;j++)
{
 printf("%d",page_queue[j]);
}
printf("\n\n");
}
printf("Total Page Faults:%d\n",page_faults);
return 0;
}
```

Slip 12

Q1. Write a program to illustrate the concept of orphan process (Using fork() and sleep())

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

int main()
{
   int pid = fork();
   if(pid>0)
   {
      printf("Parent process \n");
      printf("ID : %d \n \n",getpid());
   }
   else if(pid == 0)
   {
      printf("Child process \n");
      printf("ID : %d \n",getpid());
```

```
        sleep(10);
        printf("\nChild process\n");
        printf("ID : %d \n",getpid());
    }
    else
    {
        printf("Failed to create child process");
    }
    return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames. Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8 Implement OPT

```
#include <stdio.h>

#define MAX 100

// Function to find the index of the page to be replaced using the Optimal algorithm
int find_optimal(int frames[], int ref_string[], int ref_size, int current_pos, int num_frames) {
    int i, j, farthest = current_pos, page_index = -1;

    for (i = 0; i < num_frames; i++) {
        int found = 0;
        for (j = current_pos + 1; j < ref_size; j++) {
            if (frames[i] == ref_string[j]) {
                if (j > farthest) {
                    farthest = j;
                    page_index = i;
                }
                found = 1;
                break;
            }
        }

        if (!found) {
            return i;
        }
    }

    return (page_index == -1) ? 0 : page_index;
}

// Function to implement the Optimal Page Replacement algorithm
void optimal_page_replacement(int ref_string[], int ref_size, int num_frames) {
    int frames[num_frames];
    int i, j, page_faults = 0;
    // Initialize frames with -1 to indicate empty slots
    for (i = 0; i < num_frames; i++) {
        frames[i] = -1;
    }

    printf("Page scheduling:\n");
    for (i = 0; i < ref_size; i++) {
        int page = ref_string[i];
        int found = 0;

        // Check if the page is already in the frames
        for (j = 0; j < num_frames; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurs
            page_faults++;

            if (i < num_frames) {
                // If there is space in frames, insert the page into the next empty frame
                frames[i] = page;
            } else {
                // Replace a page using the Optimal algorithm
                int replace_index = find_optimal(frames, ref_string, ref_size, i, num_frames);
                frames[replace_index] = page;
            }

            // Print the current state of frames
            printf("Page %d: ", page);
            for (j = 0; j < num_frames; j++) {
                if (frames[j] == -1)
                    printf("[ ] ");
                else
                    printf("[%d] ", frames[j]);
            }
            printf("\n");
        }
    }

    printf("Total number of page faults: %d\n", page_faults);
}

int main() {
    int ref_string[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int num_frames = 3;
    int ref_size = sizeof(ref_string) / sizeof(ref_string[0]);

    optimal_page_replacement(ref_string, ref_size, num_frames);

    return 0;
}
```

Slip 13

Q1. Write a program that demonstrates the use of nice() system call. After a child process is started using fork(), assign higher priority to the child using nice() system call.

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

int main()
{
    pid_t pid;

    // Fork a child process
    pid = fork();

    if (pid < 0)
    {  // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0)
    {  // Child process
        printf("Child process (PID: %d) before changing priority\n", getpid());

        // Set a higher priority (decreasing the nice value)
        int new_nice_value = nice(-5);  // Decrease nice value by 5

        if (new_nice_value == -1 && errno != 0)
        {
            perror("Nice failed");
            exit(EXIT_FAILURE);
        }
        printf("Child process (PID: %d) after changing priority. New nice value: %d\n", getpid(),
new_nice_value);

        // Simulate some work
        for (int i = 0; i < 5; i++)
        {
            printf("Child process (PID: %d) is working...\n", getpid());
            sleep(1);
        }

        exit(EXIT_SUCCESS);
    }
    else
    {  // Parent process
        printf("Parent process (PID: %d)\n", getpid());

        // Wait for the child process to complete
        wait(NULL);

        printf("Parent process (PID: %d) after child has completed.\n", getpid());
    }

    return 0;
}
```

Q2. Write a C program to simulate Banker's algorithm for the purpose of deadlock avoidance. Consider the following snapshot of system, A, B, C and D are the resource type. a) Calculate and display the content of need matrix? b) Is the system in safe state? If display the safe sequence. c) If a request from process P arrives for (0, 4, 2, 0) can it be granted immediately by keeping the system in safe state. Print a message.

```
#include <stdio.h>

#define P 5  // Number of processes
#define R 4  // Number of resource types

void calculateNeed(int need[P][R], int max[P][R], int allocation[P][R]) {
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

int isSafe(int processes[], int available[], int max[][R], int allocation[][R]) {
    int need[P][R];
    calculateNeed(need, max, allocation);
    int finish[P] = {0};
    int safeSeq[P];
    int work[R];

    for (int i = 0; i < R; i++)
        work[i] = available[i];

    int count = 0;
    while (count < P) {
        int found = 0;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                if (j == R) {
                    for (int k = 0; k < R; k++)
                        work[k] += allocation[p][k];

                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = 1;
                }
            }
        }

        if (found == 0) {
            printf("System is not in a safe state\n");
            return 0;
        }
    }

    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < P; i++)
        printf("%d ", safeSeq[i]);
    printf("\n");
    return 1;
}

void requestResources(int process, int request[], int available[], int allocation[][R], int max[][R]) {
    int need[P][R];
    calculateNeed(need, max, allocation);

    for (int i = 0; i < R; i++) {
        if (request[i] > need[process][i]) {
            printf("Error: Process has exceeded its maximum claim.\n");
            return;
        }
        if (request[i] > available[i]) {
            printf("Process must wait, resources are not available.\n");
            return;
        }
    }

    for (int i = 0; i < R; i++) {
        available[i] -= request[i];
        allocation[process][i] += request[i];
        need[process][i] -= request[i];
    }

    if (isSafe((int[]){0, 1, 2, 3, 4}, available, max, allocation)) {
        printf("Request can be granted immediately.\n");
    } else {
        printf("Request cannot be granted immediately. Rolling back.\n");
        for (int i = 0; i < R; i++) {
            available[i] += request[i];
            allocation[process][i] -= request[i];
            need[process][i] += request[i];
        }
    }
}

int main() {
    int processes[] = {0, 1, 2, 3, 4};

    int allocation[P][R] = {{0, 0, 1, 2},
                            {1, 0, 0, 0},
                            {1, 3, 5, 4},
                            {0, 6, 3, 2},
                            {0, 0, 1, 4}};

    int max[P][R] = {{2, 0, 1, 2},
                     {1, 7, 5, 0},
                     {2, 3, 5, 6},
                     {0, 6, 5, 2},
                     {0, 6, 5, 6}};

    int available[R] = {1, 5, 2, 0};

    isSafe(processes, available, max, allocation);

    int request[] = {0, 4, 2, 0};
    requestResources(1, request, available, allocation, max);
```

```
   return 0;
}
```

Slip 14

Q1. Write a program to find the execution time taken for execution of a given set of instructions (use clock() function)

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

int main()
{
   int i;
   clock_t start_time = clock();
   for(i=1;i<1000000;i++)
   {

   }
   clock_t end_time = clock();
   double execution_time = (double)(end_time-start_time)/CLOCKS_PER_SEC;
   printf("Execution time : %f seconds\n",execution_time);
   return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n =3 as the number of memory frames. Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1 Implement FIFO

```
#include<stdio.h>

int main()
{
int page_reference_string[]={3,4,5,6,3,4,7,3,4,5,6,7,2,4,6};
int n,i,j;
printf("Enter the numbers of memory frames:");
scanf("%d",&n);
int memory_frames[n];
int page_queue[n];
int page_faults=0;

for(i=0;i<n;i++)
{
 memory_frames[i]=-1;
page_queue[i]=-1;
```

```
}

for(i=0;i<sizeof(page_reference_string)/sizeof(page_reference_string[0]);i++)
{
 int page=page_reference_string[i];
 int found=0;
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==page)
{
 found=1;
 break;
}
}

if(!found)
{
 page_faults++;

 if(page_queue[0]!=-1)
{
 int replaced_page=page_queue[0];
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==replaced_page)
{
  memory_frames[j]=page;
 break;
}
}
for(j=0;j<n-1;j++)
{
 page_queue[j]=page_queue[j+1];
}
}
else
{
 for(j=0;j<n;j++)
{
 if(memory_frames[j]==-1)
{
 memory_frames[j]=page;
break;
}
}
}
for(j=0;j<n-1;j++)
{
 page_queue[j]=page_queue[j+1];
}
page_queue[n-1]=page;
}
printf("Page reference:%d\n",page);
printf("Memory Frames:");
for(j=0;j<n;j++)
{
 printf("%d",memory_frames[j]);
}
printf("\n Page Queue:");
 for(j=0;j<n;j++)
{
 printf("%d",page_queue[j]);
}
printf("\n\n");
}
printf("Total Page Faults:%d\n",page_faults);
return 0;
}
```

Slip 15

Q1. Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

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

int main()
{
    pid_t pid;
    // Fork a child process
    pid = fork();

    if (pid < 0)
    {  // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0)
    {  // Child process
        printf("Child process (PID: %d) is executing 'ls' command using execl().\n", getpid());
        // Execute the "ls" command
        execl("/bin/ls", "ls", "-l", (char *)NULL);

        // If execl() fails
        perror("execl() failed");
        exit(EXIT_FAILURE);
    }
    else {  // Parent process
        printf("Parent process (PID: %d) is going to sleep.\n", getpid());

        // Parent process goes to sleep
        sleep(5);

        // Wait for the child process to complete
        wait(NULL);

        printf("Parent process (PID: %d) has woken up and child process completed.\n", getpid());
    }
    return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames. Reference String :7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 Implement LRU

```
#include <stdio.h>

int findLRU(int time[], int n) {
    int min = time[0], pos = 0, i;
    for (i = 1; i < n; ++i) {
        if (time[i] < min) {
            min = time[i];
            pos = i;
        }
    }
    return pos;
}

int lru_page_replacement(int pages[], int n, int frames) {
    int memory_frames[frames], time[frames], counter = 0, page_faults = 0;
    int i, j, pos;

    // Initialize memory frames and time array
    for (i = 0; i < frames; i++) {
        memory_frames[i] = -1;
        time[i] = 0;
    }
    // Processing the reference string
    for (i = 0; i < n; i++) {
        int found = 0;

        // Check if the page is already in the memory frames
        for (j = 0; j < frames; j++) {
            if (memory_frames[j] == pages[i]) {
                counter++;
                time[j] = counter; // Update time for the recently used page
                found = 1;
                break;
            }
        }

        if (!found) {
            // If the page is not found, handle the page fault
            page_faults++;
            if (i < frames) {
                // Insert page if there's still room
                memory_frames[i] = pages[i];
                counter++;
                time[i] = counter;
            } else {
                // Find the least recently used page and replace it
                pos = findLRU(time, frames);
                memory_frames[pos] = pages[i];
                counter++;
                time[pos] = counter;
            }
        }

        // Print the current state of the memory frames
        printf("Page %d -> Memory: ", pages[i]);
        for (j = 0; j < frames; j++) {
            if (memory_frames[j] == -1)
                printf("- ");
            else
                printf("%d ", memory_frames[j]);
        }
        printf("\n");
    }

    return page_faults;
}

int main() {
    int frames, page_faults;
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2}; // Reference string
    int n = sizeof(pages) / sizeof(pages[0]); // Number of pages

    // Input number of memory frames
    printf("Enter the number of memory frames: ");
    scanf("%d", &frames);

    // Call LRU page replacement and get the number of page faults
    page_faults = lru_page_replacement(pages, n, frames);

    // Output the total page faults
    printf("Total Page Faults: %d\n", page_faults);

    return 0;
}
```

Slip 16

Q1. Write a program to find the execution time taken for execution of a given set of instructions (use clock() function)

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

int main()
{
    int i;
    clock_t start_time = clock();
    for(i=1;i<1000000;i++)
    {

    }
    clock_t end_time = clock();
    double execution_time = (double)(end_time-start_time)/CLOCKS_PER_SEC;
    printf("Execution time : %f seconds\n",execution_time);
    return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n =3 as the number of memory frames. Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8 Implement OPT

```
#include <stdio.h>

#define MAX 100
```

```
// Function to find the index of the page to be replaced using the Optimal algorithm
int find_optimal(int frames[], int ref_string[], int ref_size, int current_pos, int num_frames) {
    int i, j, farthest = current_pos, page_index = -1;

    for (i = 0; i < num_frames; i++) {
        int found = 0;
        for (j = current_pos + 1; j < ref_size; j++) {
            if (frames[i] == ref_string[j]) {
                if (j > farthest) {
                    farthest = j;
                    page_index = i;
                }
                found = 1;
                break;
            }
        }

        if (!found) {
            return i;
        }
    }

    return (page_index == -1) ? 0 : page_index;
}

// Function to implement the Optimal Page Replacement algorithm
void optimal_page_replacement(int ref_string[], int ref_size, int num_frames) {
    int frames[num_frames];
    int i, j, page_faults = 0;

    // Initialize frames with -1 to indicate empty slots
    for (i = 0; i < num_frames; i++) {
        frames[i] = -1;
    }

    printf("Page scheduling:\n");
    for (i = 0; i < ref_size; i++) {
        int page = ref_string[i];
        int found = 0;

        // Check if the page is already in the frames
        for (j = 0; j < num_frames; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }
        if (!found) {
            // Page fault occurs
            page_faults++;

            if (i < num_frames) {
                // If there is space in frames, insert the page into the next empty frame
                frames[i] = page;
            } else {
                // Replace a page using the Optimal algorithm
                int replace_index = find_optimal(frames, ref_string, ref_size, i, num_frames);
                frames[replace_index] = page;
            }

            // Print the current state of frames
            printf("Page %d: ", page);
            for (j = 0; j < num_frames; j++) {
                if (frames[j] == -1)
                    printf("[ ] ");
                else
                    printf("[%d] ", frames[j]);
            }
            printf("\n");
        }
    }

    printf("Total number of page faults: %d\n", page_faults);
}

int main() {
    int ref_string[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int num_frames = 3;
    int ref_size = sizeof(ref_string) / sizeof(ref_string[0]);

    optimal_page_replacement(ref_string, ref_size, num_frames);

    return 0;
}
```

Slip 17

Q1. Write the program to calculate minimum number of resources needed to avoid deadlock.

```
#include <stdio.h>

int main() {
    int processes, max_resources, available, i, j;
    // Input the number of processes and maximum resources
    printf("Enter number of processes: ");
    scanf("%d", &processes);
    printf("Enter the number of resource instances: ");
    scanf("%d", &max_resources);

    int max[processes], allocated[processes];

    // Input maximum and allocated resources for each process
    printf("Enter the maximum resources required by each process:\n");
    for (i = 0; i < processes; i++) {
        printf("Process %d: ", i);
        scanf("%d", &max[i]);
    }

    printf("Enter the currently allocated resources for each process:\n");
    for (i = 0; i < processes; i++) {
        printf("Process %d: ", i);
        scanf("%d", &allocated[i]);
    }

    // Calculate the minimum number of resources needed to avoid deadlock
    int total_max = 0, total_allocated = 0;
    for (i = 0; i < processes; i++) {
        total_max += max[i];
        total_allocated += allocated[i];
    }

    int min_resources_needed = total_max - total_allocated;
    printf("Minimum additional resources needed to avoid deadlock: %d\n", min_resources_needed);

    return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n=3 as the number of memory frames. Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8 Implement OPT

```
#include <stdio.h>

#define MAX 100
```

```
// Function to find the index of the page to be replaced using the Optimal algorithm
int find_optimal(int frames[], int ref_string[], int ref_size, int current_pos, int num_frames) {
   int i, j, farthest = current_pos, page_index = -1;

   for (i = 0; i < num_frames; i++) {
      int found = 0;
      for (j = current_pos + 1; j < ref_size; j++) {
         if (frames[i] == ref_string[j]) {
            if (j > farthest) {
               farthest = j;
               page_index = i;
            }
            found = 1;
            break;
         }
      }

      if (!found) {
         return i;
      }
   }

   return (page_index == -1) ? 0 : page_index;
}

// Function to implement the Optimal Page Replacement algorithm
void optimal_page_replacement(int ref_string[], int ref_size, int num_frames) {
   int frames[num_frames];
   int i, j, page_faults = 0;

   // Initialize frames with -1 to indicate empty slots
   for (i = 0; i < num_frames; i++) {
      frames[i] = -1;
   }

   printf("Page scheduling:\n");
   for (i = 0; i < ref_size; i++) {
      int page = ref_string[i];
      int found = 0;

      // Check if the page is already in the frames
      for (j = 0; j < num_frames; j++) {
         if (frames[j] == page) {
            found = 1;
            break;
         }
      }
```

```
        if (!found) {
            // Page fault occurs
            page_faults++;

            if (i < num_frames) {
                // If there is space in frames, insert the page into the next empty frame
                frames[i] = page;
            } else {
                // Replace a page using the Optimal algorithm
                int replace_index = find_optimal(frames, ref_string, ref_size, i, num_frames);
                frames[replace_index] = page;
            }

            // Print the current state of frames
            printf("Page %d: ", page);
            for (j = 0; j < num_frames; j++) {
                if (frames[j] == -1)
                    printf("[ ] ");
                else
                    printf("[%d] ", frames[j]);
            }
            printf("\n");
        }
    }

    printf("Total number of page faults: %d\n", page_faults);
}

int main() {
    int ref_string[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int num_frames = 3;
    int ref_size = sizeof(ref_string) / sizeof(ref_string[0]);

    optimal_page_replacement(ref_string, ref_size, num_frames);

    return 0;
}
```

Slip 18

Q1. Write a C program to accept the number of process and resources and find the need matrix content and display it.

```
#include <stdio.h>

int main()
{
```

```
int num_processes, num_resources;

// Accept the number of processes and resources
printf("Enter the number of processes: ");
scanf("%d", &num_processes);

printf("Enter the number of resources: ");
scanf("%d", &num_resources);

int max[num_processes][num_resources];
int allocation[num_processes][num_resources];
int need[num_processes][num_resources];

// Accept the Max matrix
printf("Enter the Max matrix (%d x %d):\n", num_processes, num_resources);
for (int i = 0; i < num_processes; i++)
{
   for (int j = 0; j < num_resources; j++)
   {
      scanf("%d", &max[i][j]);
   }
}

// Accept the Allocation matrix
printf("Enter the Allocation matrix (%d x %d):\n", num_processes, num_resources);

for (int i = 0; i < num_processes; i++)
{
   for (int j = 0; j < num_resources; j++)
   {
      scanf("%d", &allocation[i][j]);
   }
}

// Calculate the Need matrix
for (int i = 0; i < num_processes; i++)
{
   for (int j = 0; j < num_resources; j++)
   {
      need[i][j] = max[i][j] - allocation[i][j];
   }
}

// Display the Need matrix
printf("The Need matrix is:\n");
for (int i = 0; i < num_processes; i++)
{
   for (int j = 0; j < num_resources; j++)
```

```
        {
            printf("%d ", need[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n as the number of memory frames. Reference String : 12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8 Implement OPT

```
#include <stdio.h>

#define MAX 100

// Function to find the index of the page to be replaced using the Optimal algorithm
int find_optimal(int frames[], int ref_string[], int ref_size, int current_pos, int num_frames) {
    int i, j, farthest = current_pos, page_index = -1;

    for (i = 0; i < num_frames; i++) {
        int found = 0;
        for (j = current_pos + 1; j < ref_size; j++) {
            if (frames[i] == ref_string[j]) {
                if (j > farthest) {
                    farthest = j;
                    page_index = i;
                }
                found = 1;
                break;
            }
        }

        if (!found) {
            return i;
        }
    }

    return (page_index == -1) ? 0 : page_index;
}

// Function to implement the Optimal Page Replacement algorithm
void optimal_page_replacement(int ref_string[], int ref_size, int num_frames) {
    int frames[num_frames];
    int i, j, page_faults = 0;

    // Initialize frames with -1 to indicate empty slots
    for (i = 0; i < num_frames; i++) {
        frames[i] = -1;
    }

    printf("Page scheduling:\n");
    for (i = 0; i < ref_size; i++) {
        int page = ref_string[i];
        int found = 0;

        // Check if the page is already in the frames
        for (j = 0; j < num_frames; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            // Page fault occurs
            page_faults++;

            if (i < num_frames) {
                // If there is space in frames, insert the page into the next empty frame
                frames[i] = page;
            } else {
                // Replace a page using the Optimal algorithm
                int replace_index = find_optimal(frames, ref_string, ref_size, i, num_frames);
                frames[replace_index] = page;
            }

            // Print the current state of frames
            printf("Page %d: ", page);
            for (j = 0; j < num_frames; j++) {
                if (frames[j] == -1)
                    printf("[ ] ");
                else
                    printf("[%d] ", frames[j]);
            }
            printf("\n");
        }
    }

    printf("Total number of page faults: %d\n", page_faults);
}

int main() {
    int ref_string[] = {12, 15, 12, 18, 6, 8, 11, 12, 19, 12, 6, 8, 12, 15, 19, 8};
    int num_frames = 3;
    int ref_size = sizeof(ref_string) / sizeof(ref_string[0]);

    optimal_page_replacement(ref_string, ref_size, num_frames);

    return 0;
}
```

Slip 19

Q1. Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

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

int main()
{
    pid_t pid;
    // Fork a child process
    pid = fork();

    if (pid < 0)
    {  // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0)
    {  // Child process
        printf("Child process (PID: %d) is executing 'ls' command using execl().\n", getpid());

        // Execute the "ls" command
        execl("/bin/ls", "ls", "-l", (char *)NULL);

        // If execl() fails
        perror("execl() failed");
        exit(EXIT_FAILURE);
    }
    else {  // Parent process
        printf("Parent process (PID: %d) is going to sleep.\n", getpid());

        // Parent process goes to sleep
        sleep(5);
        // Wait for the child process to complete
        wait(NULL);

        printf("Parent process (PID: %d) has woken up and child process completed.\n", getpid());
    }
    return 0;
}
```

Q3. Write a C program to simulate Banker's algorithm for the purpose of deadlock avoidance. Consider the following snapshot of system, A, B, C and D are the resource type.
a) Calculate and display the content of need matrix?
b) Is the system in safe state? If display the safe sequence.
c) If a request from process P arrives for (0, 4, 2, 0) can it be granted immediately by keeping the system in safe state. Print a message

```
#include <stdio.h>

#define P 5  // Number of processes
#define R 4  // Number of resource types

void calculateNeed(int need[P][R], int max[P][R], int allocation[P][R]) {
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

int isSafe(int processes[], int available[], int max[][R], int allocation[][R]) {
    int need[P][R];
    calculateNeed(need, max, allocation);

    int finish[P] = {0};
    int safeSeq[P];
    int work[R];

    for (int i = 0; i < R; i++)
        work[i] = available[i];

    int count = 0;
    while (count < P) {
        int found = 0;
        for (int p = 0; p < P; p++) {
            if (finish[p] == 0) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                if (j == R) {
                    for (int k = 0; k < R; k++)
                        work[k] += allocation[p][k];

                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = 1;
                }
            }
        }

        if (found == 0) {
            printf("System is not in a safe state\n");
            return 0;
        }
    }

    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < P; i++)
        printf("%d ", safeSeq[i]);
    printf("\n");
    return 1;
}

void requestResources(int process, int request[], int available[], int allocation[][R], int max[][R]) {
    int need[P][R];
    calculateNeed(need, max, allocation);

    for (int i = 0; i < R; i++) {
        if (request[i] > need[process][i]) {
            printf("Error: Process has exceeded its maximum claim.\n");
            return;
        }
        if (request[i] > available[i]) {
            printf("Process must wait, resources are not available.\n");
            return;
        }
    }

    for (int i = 0; i < R; i++) {
        available[i] -= request[i];
        allocation[process][i] += request[i];
        need[process][i] -= request[i];
    }
```

```
    if (isSafe((int[]){0, 1, 2, 3, 4}, available, max, allocation)) {
        printf("Request can be granted immediately.\n");
    } else {
        printf("Request cannot be granted immediately. Rolling back.\n");
        for (int i = 0; i < R; i++) {
            available[i] += request[i];
            allocation[process][i] -= request[i];
            need[process][i] += request[i];
        }
    }
}

int main() {
    int processes[] = {0, 1, 2, 3, 4};

    int allocation[P][R] = {{0, 0, 1, 2},
                            {1, 0, 0, 0},
                            {1, 3, 5, 4},
                            {0, 6, 3, 2},
                            {0, 0, 1, 4}};

    int max[P][R] = {{2, 0, 1, 2},
                     {1, 7, 5, 0},
                     {2, 3, 5, 6},
                     {0, 6, 5, 2},
                     {0, 6, 5, 6}};

    int available[R] = {1, 5, 2, 0};

    isSafe(processes, available, max, allocation);

    int request[] = {0, 4, 2, 0};
    requestResources(1, request, available, allocation, max);

    return 0;
}
```

Slip 20

Q1. Write a program to create a child process using fork().The parent should goto sleep state and child process should begin its execution. In the child process, use execl() to execute the “ls” command.

```
#include <stdio.h>

int main() {
    int processes, max_resources, available, i, j;
    // Input the number of processes and maximum resources
    printf("Enter number of processes: ");
    scanf("%d", &processes);
    printf("Enter the number of resource instances: ");
    scanf("%d", &max_resources);

    int max[processes], allocated[processes];

    // Input maximum and allocated resources for each process
    printf("Enter the maximum resources required by each process:\n");
    for (i = 0; i < processes; i++) {
        printf("Process %d: ", i);
        scanf("%d", &max[i]);
    }

    printf("Enter the currently allocated resources for each process:\n");
    for (i = 0; i < processes; i++) {
        printf("Process %d: ", i);
        scanf("%d", &allocated[i]);
    }

    // Calculate the minimum number of resources needed to avoid deadlock
    int total_max = 0, total_allocated = 0;
    for (i = 0; i < processes; i++) {
        total_max += max[i];
        total_allocated += allocated[i];
    }

    int min_resources_needed = total_max - total_allocated;
    printf("Minimum additional resources needed to avoid deadlock: %d\n", min_resources_needed);

    return 0;
}
```

Q2. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults for the following given page reference string. Give input n=3 as the number of memory frames. Reference String : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2. Implement LRU

```
#include <stdio.h>

int findLRU(int time[], int n) {
    int min = time[0], pos = 0, i;
    for (i = 1; i < n; ++i) {
        if (time[i] < min) {
            min = time[i];
            pos = i;
        }
    }
    return pos;
}

int lru_page_replacement(int pages[], int n, int frames) {
    int memory_frames[frames], time[frames], counter = 0, page_faults = 0;
    int i, j, pos;

    // Initialize memory frames and time array
    for (i = 0; i < frames; i++) {
        memory_frames[i] = -1;
        time[i] = 0;
    }

    // Processing the reference string
    for (i = 0; i < n; i++) {
        int found = 0;

        // Check if the page is already in the memory frames
        for (j = 0; j < frames; j++) {
            if (memory_frames[j] == pages[i]) {
                counter++;
                time[j] = counter; // Update time for the recently used page
                found = 1;
                break;
            }
        }

        if (!found) {
            // If the page is not found, handle the page fault
            page_faults++;
            if (i < frames) {
                // Insert page if there's still room
                memory_frames[i] = pages[i];
                counter++;
                time[i] = counter;
            } else {
                // Find the least recently used page and replace it
                pos = findLRU(time, frames);
                memory_frames[pos] = pages[i];
                counter++;
                time[pos] = counter;
            }
        }

        // Print the current state of the memory frames
        printf("Page %d -> Memory: ", pages[i]);
        for (j = 0; j < frames; j++) {
            if (memory_frames[j] == -1)
                printf("- ");
            else
                printf("%d ", memory_frames[j]);
        }
        printf("\n");
    }

    return page_faults;
}

int main() {
    int frames, page_faults;
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2}; // Reference string
    int n = sizeof(pages) / sizeof(pages[0]); // Number of pages

    // Input number of memory frames
    printf("Enter the number of memory frames: ");
    scanf("%d", &frames);

    // Call LRU page replacement and get the number of page faults
    page_faults = lru_page_replacement(pages, n, frames);

    // Output the total page faults
    printf("Total Page Faults: %d\n", page_faults);

    return 0;
}
```
1

11