J.N.T.U.H. UNIVERSITY COLLEGE OF ENGINEERING ,

SCIENCE AND TECHNOLOGY HYDERABAD

KUKATPALLY, HYDERABAD – 500 085

Certificate

Certified that this is the bonafide record of the practical work done during

the academic year _____ by

Name _____

Roll Number _____ Class _____

in the Laboratory of _____

of the Department of _____

Signature of the Staff Member

Signature of the Head of the Department

Date of Examination_____

Signature of the Examiner/s

Internal Examiner

External Examiner

LIST OF PROGRAMS

1.Write a C program to simulate FCFS CPU scheduling algorithm

```
#include<stdio.h>

int main() {
   int p[10], at[10], bt[10], ct[10], tat[10], wt[10], i, j, temp = 0, n;
   float awt = 0, atat = 0;

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

   printf("Enter process IDs: ");
   for(i = 0; i < n; i++) {
      scanf("%d", &p[i]);
   }

   printf("Enter %d arrival times: ", n);
   for(i = 0; i < n; i++) {
      scanf("%d", &at[i]);
   }

   printf("Enter %d burst times: ", n);
   for(i = 0; i < n; i++) {
      scanf("%d", &bt[i]);
   }

   // Sorting at, bt, and process according to arrival time
   for(i = 0; i < n - 1; i++) {
      for(j = 0; j < n - i - 1; j++) {
         if(at[j] > at[j + 1]) {
            // Swap process IDs
            temp = p[j + 1];
            p[j + 1] = p[j];
            p[j] = temp;

            // Swap arrival times
            temp = at[j + 1];
            at[j + 1] = at[j];
            at[j] = temp;
```

```
            // Swap burst times
            temp = bt[j + 1];
            bt[j + 1] = bt[j];
            bt[j] = temp;
         }
      }
   }

   // Calculating completion time for the first process
   ct[0] = at[0] + bt[0];

   // Calculating completion times for remaining processes
   for(i = 1; i < n; i++) {
      temp = 0;
      if(ct[i - 1] < at[i]) {
         temp = at[i] - ct[i - 1];
      }
      ct[i] = ct[i - 1] + bt[i] + temp;
   }

   // Calculating turnaround time and waiting time
   printf("\np\t A.T\t B.T\t C.T\t TAT\t WT");
   for(i = 0; i < n; i++) {
      tat[i] = ct[i] - at[i];
      wt[i] = tat[i] - bt[i];
      atat += tat[i];
      awt += wt[i];
   }

   atat = atat / n;
   awt = awt / n;

   for(i = 0; i < n; i++) {
      printf("\nP%d\t %d\t %d\t %d \t %d \t %d", p[i], at[i], bt[i], ct[i], tat[i], wt[i]);
   }

   printf("\nAverage turnaround time is %f", atat);
   printf("\nAverage waiting time is %f", awt);

   return 0;
}
```

Output:

```
Enter number of processes you want: 3
Enter process IDs: 1
2
3
Enter 3 arrival times: 1
5
8
Enter 3 burst times: 2
4
8

p        A.T     B.T     C.T     TAT     WT
P1       1       2       3       2       0
P2       5       4       9       4       0
P3       8       8       17      9       1
Average turnaround time is 5.000000
Average waiting time is 0.333333
```

2.Write a C program to simulate SJF scheduling algorithm(non-preemptive).

```
#include<stdio.h>

void main()
    {
      int bt[20], p[20], at[20], wt[20], tat[20], ct[20], i, j, n, total_wt = 0, total_tat = 0;
      int temp, pos, current_time;
      float avg_wt, avg_tat;

      printf("Implementation of SJF (with arrival times):\n");
      printf("Enter number of processes: ");
      scanf("%d", &n);

      printf("\nEnter Arrival Time and Burst Time:\n");
      for(i = 0; i < n; i++)
      {
        printf("p%d Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("p%d Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
        p[i] = i + 1;  // contains process number
      }

      // Sorting based on arrival time
      for(i = 0; i < n; i++)
      {
        pos = i;
        for(j = i + 1; j < n; j++)
        {
          if(at[j] < at[pos])
            pos = j;
        }

        // Swap arrival time
        temp = at[i];
        at[i] = at[pos];
        at[pos] = temp;
```

```
    // Swap burst time
    temp = bt[i];
    bt[i] = bt[pos];
    bt[pos] = temp;

    // Swap process number
    temp = p[i];
    p[i] = p[pos];
    p[pos] = temp;
}

// Initialize the current time to the arrival time of the first process
current_time = at[0];

// Process the first job
wt[0] = 0; // waiting time for the first process is zero
ct[0] = current_time + bt[0];
tat[0] = bt[0]; // turnaround time for the first process is its burst time
total_tat = tat[0];
current_time = ct[0];

// Process the remaining jobs
for(i = 1; i < n; i++)
{
    pos = i;
    for(j = i + 1; j < n; j++)
    {
        if(at[j] <= current_time && bt[j] < bt[pos])
            pos = j;
    }

    // Swap arrival time
    temp = at[i];
    at[i] = at[pos];
    at[pos] = temp;

    // Swap burst time
    temp = bt[i];
    bt[i] = bt[pos];
    bt[pos] = temp;
```

```
        // Swap process number
        temp = p[i];
        p[i] = p[pos];
        p[pos] = temp;

        // Calculate waiting time and turnaround time
        if(current_time < at[i])
          current_time = at[i];

        wt[i] = current_time - at[i];
        current_time += bt[i];
        ct[i] = current_time;
        tat[i] = wt[i] + bt[i];

        total_wt += wt[i];
        total_tat += tat[i];
    }

    avg_wt = (float)total_wt / n; // average waiting time
    avg_tat = (float)total_tat / n; // average turnaround time

    printf("\nProcess\t Arrival Time\t Burst Time\t Waiting Time\t Turnaround
Time\t Completion Time");
    for(i = 0; i < n; i++)
    {
        printf("\np%d\t\t %d\t\t %d\t\t %d\t\t %d\t\t\t %d", p[i], at[i], bt[i], wt[i],
tat[i], ct[i]);
    }

    printf("\n\nAverage Waiting Time = %f", avg_wt);
    printf("\nAverage Turnaround Time = %f\n", avg_tat);
}
```

Output:

```
Implementation of SJF (with arrival times):
Enter number of processes: 4

Enter Arrival Time and Burst Time:
p1 Arrival Time: 0
p1 Burst Time: 6
p2 Arrival Time: 2
p2 Burst Time: 2
p3 Arrival Time: 5
p3 Burst Time: 4
p4 Arrival Time: 6
p4 Burst Time: 5

Process  Arrival Time    Burst Time      Waiting Time    Turnaround Time
Completion Time
p1               0               6               0               6
6
p2               2               2               4               6
8
p3               5               4               3               7
12
p4               6               5               6               11
17

Average Waiting Time = 3.250000
Average Turnaround Time = 7.500000
```

3.Write a C program to simulate SJF scheduling algorithm(preemptive).

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

#define SIZE 5

void SJFPreemptive(int n, int processes[], int arrivalTime[], int burstTime[]) {
   int remainingTime[SIZE];
   int completionTime[SIZE];
   int waitingTime[SIZE];
   int turnaroundTime[SIZE];
   int totalWaitingTime = 0;
   int totalTurnaroundTime = 0;
   int currentTime = 0;
   int completed = 0;

   for (int i = 0; i < n; i++) {
      remainingTime[i] = burstTime[i];
   }

   while (completed != n) {
      int minTime = INT_MAX;
      int shortest = -1;

      for (int i = 0; i < n; i++) {
         if (arrivalTime[i] <= currentTime && remainingTime[i] > 0 && remainingTime[i] <
minTime) {
            minTime = remainingTime[i];
            shortest = i;
         }
      }

      if (shortest == -1) {
         currentTime++;
         continue;
      }

      remainingTime[shortest]--;
      currentTime++;
```

```
        if (remainingTime[shortest] == 0) {
            completionTime[shortest] = currentTime;
            turnaroundTime[shortest] = completionTime[shortest] - arrivalTime[shortest];
            waitingTime[shortest] = turnaroundTime[shortest] - burstTime[shortest];

            totalWaitingTime += waitingTime[shortest];
            totalTurnaroundTime += turnaroundTime[shortest];

            completed++;
        }
    }

    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], arrivalTime[i], burstTime[i],
waitingTime[i], turnaroundTime[i]);
    }

    printf("\nAverage Waiting Time: %.2f\n", (float)totalWaitingTime / n);
    printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);
}

int main() {
    int processes[SIZE];
    int arrivalTime[SIZE];
    int burstTime[SIZE];

    printf("Enter the process details (processId, arrivalTime, burstTime):\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Process %d:\n", i + 1);
        scanf("%d %d %d", &processes[i], &arrivalTime[i], &burstTime[i]);
    }

    SJFPreemptive(SIZE, processes, arrivalTime, burstTime);

    return 0;
}
```

Output:

```
Enter the process details (processId, arrivalTime, burstTime):
Process 1:
1 0 6
Process 2:
2 1 2
Process 3:
3 2 2
Process 4:
4 3 5
Process 5:
5 7 4
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       0               6               4               10
2       1               2               0               2
3       2               2               1               3
4       3               5               11              16
5       7               4               3               7

Average Waiting Time: 3.80
Average Turnaround Time: 7.60
```

4.Write a C program to simulate Priority scheduling algorithm(Non-Preemptive).

```
#include <stdio.h>

int main() {
   int bt[20], p[20], wt[20], tat[20], pri[20], i, j, n, total = 0, pos, temp;
   float avg_wt, avg_tat;

   printf("Implementation of Priority algorithm\n");
   printf("Enter number of processes: ");
   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; // Contains process number
   }

   printf("Enter Priority of the processes:\n");
   for (i = 0; i < n; i++) {
      printf("p%d: ", i + 1);
      scanf("%d", &pri[i]);
   }

   // Sorting based on priority (Selection Sort)
   for (i = 0; i < n; i++) {
      pos = i;
      for (j = i + 1; j < n; j++) {
         if (pri[j] < pri[pos]) {
            pos = j;
         }
      }

      // Swapping burst time
      temp = bt[i];
      bt[i] = bt[pos];
      bt[pos] = temp;
```

```
            // Swapping process number
            temp = p[i];
            p[i] = p[pos];
            p[pos] = temp;

            // Swapping priority
            temp = pri[i];
            pri[i] = pri[pos];
            pri[pos] = temp;
        }

        wt[0] = 0; // Waiting time for first process is zero

        // Calculate waiting time
        for (i = 1; i < n; i++) {
            wt[i] = 0;
            for (j = 0; j < i; j++) {
                wt[i] += bt[j];
            }
            total += wt[i];
        }

        avg_wt = (float)total / n; // Average waiting time
        total = 0;

        printf("\nProcess\t Burst Time \tPriority \tWaiting Time\tTurnaround Time\n");
        for (i = 0; i < n; i++) {
            tat[i] = bt[i] + wt[i]; // Calculate turnaround time
            total += tat[i];
            printf("p%d\t\t %d\t\t %d\t\t %d\t\t\t%d\n", p[i], bt[i], pri[i], wt[i], tat[i]);
        }

        avg_tat = (float)total / n; // Average turnaround time
        printf("\nAverage Waiting Time = %f", avg_wt);
        printf("\nAverage Turnaround Time = %f\n", avg_tat);

        return 0;
    }
```

Output:

```
Implementation of Priority algorithm with Arrival Time
Enter number of processes: 4

Enter Arrival Time and Burst Time and Priority:
p1 Arrival Time: 0
p1 Burst Time: 6
p1 Priority: 3
p2 Arrival Time: 2
p2 Burst Time: 2
p2 Priority: 4
p3 Arrival Time: 5
p3 Burst Time: 4
p3 Priority: 1
p4 Arrival Time: 6
p4 Burst Time: 5
p4 Priority: 2

Process  Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
p1              0               6               3               0               6
p2              2               2               4               13              15
p3              5               4               1               1               5
p4              6               5               2               4               9

Average Waiting Time = 4.50
Average Turnaround Time = 8.75
```

5.Write a C program to simulate Priority scheduling algorithm (Preemptive).

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

#define SIZE 5

struct Process {
   int processId;
   int arrivalTime;
   int burstTime;
```

```
    int remainingTime;
    int completionTime;
    int priority;
    struct Process* next;
};

struct Process* head;

void insert() {
    struct Process* newProcess = (struct Process*)malloc(sizeof(struct Process));
    int processId, arrivalTime, burstTime, priority;
    printf("Enter the processId, arrivalTime, burstTime, priority:\n");
    scanf("%d %d %d %d", &processId, &arrivalTime, &burstTime, &priority);
    newProcess->processId = processId;
    newProcess->arrivalTime = arrivalTime;
    newProcess->burstTime = burstTime;
    newProcess->remainingTime = burstTime;
    newProcess->completionTime = 0;
    newProcess->priority = priority;
    newProcess->next = NULL;

    if (head == NULL) {
        head = newProcess;
    } else {
        struct Process* curr = head;
        while (curr->next != NULL) {
            curr = curr->next;
        }
        newProcess->next = curr->next;
        curr->next = newProcess;
    }
}

void PriorityPreemScheduling() {
    int currTime = 0;
    int processesDone = 0;
    float waitingTime = 0;
    printf("Process\tArrival Time\tBurst Time\tPriority\tWaiting Time\n");
    while (processesDone != SIZE) {
        struct Process* shortest = NULL;
        int highestPriority = INT_MAX;
        struct Process* temp = head;
```

```
      while (temp != NULL) {
         if (temp->arrivalTime <= currTime && temp->remainingTime > 0 && temp->priority
< highestPriority) {
            highestPriority = temp->priority;
            shortest = temp;
         }
         temp = temp->next;
      }
      if (shortest == NULL) {
         printf("No valid Process at time %d\n", currTime);
         currTime++;
         continue;
      }
      currTime++;
      shortest->remainingTime--;
      if (shortest->remainingTime == 0) {
         processesDone++;
         shortest->completionTime = currTime;
         waitingTime += shortest->completionTime - shortest->arrivalTime - shortest-
>burstTime;
         printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", shortest->processId, shortest->arrivalTime,
shortest->burstTime, shortest->priority, shortest->completionTime - shortest->arrivalTime
- shortest->burstTime);
      }
   }
   printf("The Average Waiting Time is :%f\n", waitingTime / SIZE);
}

int main() {
   int choice = 0;
   while (choice != 3) {
      printf("Enter the choice:\n");
      scanf("%d", &choice);
      switch (choice) {
         case 1:
            insert();
            break;
         case 2:
            PriorityPreemScheduling();
            break;
         case 3:
            exit(0);
```

```
            break;
        default:
            printf("Enter the valid choice!\n");
    }
  }
}
```

Output:

```
Enter the number of Processes:
4
Enter the process details (processId, arrivalTime, burstTime, priority):
Process 1:
1 0 6 3
Process 2:
2 2 2 4
Process 3:
3 5 4 1
Process 4:
4 6 5 2
Process Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
1       0               6               3               9               15
2       2               2               4               13              15
3       5               4               1               0               4
4       6               5               2               3               8

Average Waiting Time: 6.25
Average Turnaround Time: 10.50
```

6 .Write a C program to simulate round robin scheduling algorithm.

```
#include <stdio.h>

int main() {
        int st[10], bt[10], wt[10], tat[10], n, tq;
        int i, count = 0, swt = 0, stat = 0, temp, sq = 0;
        float awt, atat;

        printf("Implementation of Round Robin algorithm\n");
        printf("Enter the number of processes: ");
        scanf("%d", &n);
        printf("Enter the burst time of each process \n");

        for (i = 0; i < n; i++) {
           printf("p %d: ", i + 1);
           scanf("%d", &bt[i]);
           st[i] = bt[i];  // Initialize remaining burst time
        }

        printf("Enter the time quantum: ");
        scanf("%d", &tq);

        // Round Robin scheduling loop
        while (1) {
           for (i = 0, count = 0; i < n; i++) {
              temp = tq;

              if (st[i] == 0) {
                 count++;
                 continue;
              }

              if (st[i] > tq) {
                 st[i] = st[i] - tq;
              } else if (st[i] >= 0) {
                 temp = st[i];
                 st[i] = 0;
              }

              sq = sq + temp;
              tat[i] = sq;
           }
```

```
        if (n == count)  // If all processes are completed
            break;
    }

    // Calculate waiting time and turnaround time
    for (i = 0; i < n; i++) {
        wt[i] = tat[i] - bt[i];
        swt = swt + wt[i];
        stat = stat + tat[i];
    }

    awt = (float)swt / n;  // Average waiting time
    atat = (float)stat / n;  // Average turnaround time

    printf("Process No\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);
    }

    printf("Average Waiting Time = %f\n", awt);
    printf("Average Turnaround Time = %f\n", atat);

    return 0;
}
```

Output:

```
Implementation of Round Robin algorithm
Enter the number of processes: 4
Enter the burst time of each process
p 1: 6
p 2: 2
p 3: 4
p 4: 5
Enter the time quantum: 3
Process No      Burst Time      Waiting Time    Turnaround Time
1               6               8               14
2               2               3               5
3               4               11              15
4               5               12              17
Average Waiting Time = 8.500000
Average Turnaround Time = 12.750000
```

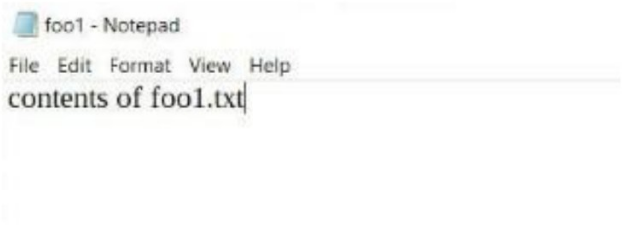
7.Write a C program to illustrate the system calls of UNIX/LINUX operatingsystem.

#opening and reading from a file and displaying its content

```
#include<stdio.
h>
#include<sys/t
ypes.h>
#include<sys/f
cntl.h>
#include<sys/s
tat.h>
#include<sys/u
nistd.h>
#include<string
.h>
 int
   m
   a
   i
   n
   (
   )
   {
   i
   n
   t
   s
   z
   ;
   char C[100];
   int

   fd=open("foo1.txt",O_RDO
```

```
NLY);if(fd>0){

   sz=read(f
   d,C,10);
   C[sz]='\0'
   ;
   printf("called read(%d,C,10) it reads 10 bytes into buffer
   \"%s\"",fd,C);close(fd);
}

return 0;
}
```

Output:



8.Write the c program using the I/O system calls of UNIX/LINUX operatingsystem

#reading and writing into file

```
#include<stdio.
h>
#include<sys/t
ypes.h>
#include<sys/f
cntl.h>
#include<sys/s
tat.h>
#include<sys/u
nistd.h>
#include<string
.h>

int
   m
   a
   i
   n
   (
   )
   {
   i
   n
   t
   s
   z
   ;
   char buff[100];
   int fd=open("foo1.txt",O_RDWR);
```

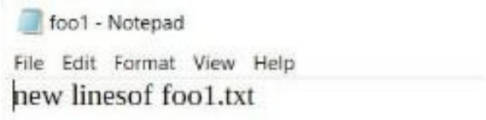
```
    if(fd>0){
       int size=strlen("new lines");
       sz=write(fd, "new lines", strlen("new lines"));
       printf("called write(%d, %s, %d)\n", fd, "new
       lines", size);sz=read(fd,buff,10);
       buff[sz]=0;
       printf("called read(%d,buff,10) it reads 10 bytes into
       buffer=\"%s\"",fd,buff);close(fd);
    }

    return 0;
}
```

Output:

File before execution:


```
D:\c++>writenew
called write(3, new lines, 9)
called read(3,buff,10) it reads 10 bytes into buffer="of foo1.tx"
D:\c++>
```

File after execution:



9.Write C programs using the I/O system calls of UNIX/LINUX operatingsystem

fcntl:

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

int main() {
   int fd;
   struct flock lock;
```

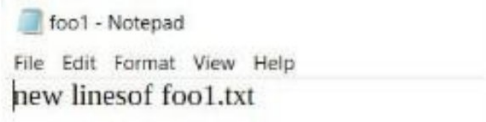
```
    fd = open("example.txt", O_RDWR | O_CREAT, 0666);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Set up the lock structure
    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;

    // Apply the write lock
    if (fcntl(fd, F_SETLK, &lock) == -1) {
        perror("fcntl");
        close(fd);
        exit(EXIT_FAILURE);
    }

    printf("Write lock acquired.\n");

    // Do some file operations here

    // Release the lock
    lock.l_type = F_UNLCK;
    if (fcntl(fd, F_SETLK, &lock) == -1) {
        perror("fcntl");
        close(fd);
        exit(EXIT_FAILURE);
    }

    printf("Write lock released.\n");

    close(fd);
    return 0;
}
```

Output:

File before execution:


```
D:\c++>writenew
called write(3, new lines, 9)
called read(3,buff,10) it reads 10 bytes into buffer="of foo1.tx"
D:\c++>
```

File after execution:



10.Write programs using the I/O system calls of UNIX/LINUX operatingsystem

lseek:

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

int main() {
   int fd;
```

```
    off_t offset;
    char buffer[20];

    fd = open("example.txt", O_RDWR | O_CREAT, 0666);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Write some data
    write(fd, "Hello, world!", 13);

    // Move the file offset to the beginning
    offset = lseek(fd, 0, SEEK_SET);
    if (offset == -1) {
        perror("lseek");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // Read the data back
    read(fd, buffer, 13);
    buffer[13] = '\0';
    printf("Read from file: %s\n", buffer);

    close(fd);
    return 0;
}
```

Output:

File before execution:

```
D:\c++>writenew
called write(3, new lines, 9)
called read(3,buff,10) it reads 10 bytes into buffer="of foo1.tx"
D:\c++>
```

File after execution

11.Write programs using the I/O system calls of UNIX/LINUX operating system.

Stat:

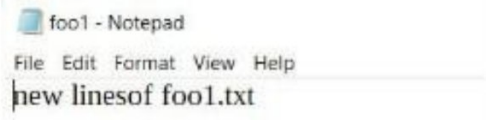
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main() {
   struct stat file_stat;

   if (stat("example.txt", &file_stat) == -1) {
      perror("stat");
      exit(EXIT_FAILURE);
   }

   printf("File size: %ld bytes\n", file_stat.st_size);
   printf("Number of links: %ld\n", file_stat.st_nlink);
   printf("File inode: %ld\n", file_stat.st_ino);

   return 0;
}
```

Output:

File before execution:


```
D:\c++>writenew
called write(3, new lines, 9)
called read(3,buff,10) it reads 10 bytes into buffer="of foo1.tx"
D:\c++>
```

File after execution:



12.Write programs using the I/O system calls of UNIX/LINUX operatingsystem

Open_Read_dir:

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

int main() {
   DIR *dir;
   struct dirent *entry;

   dir = opendir(".");
   if (dir == NULL) {
      perror("opendir");
```

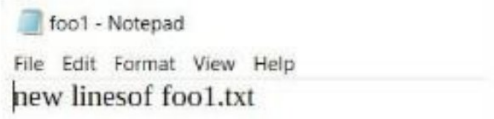
```
        exit(EXIT_FAILURE);
    }

    while ((entry = readdir(dir)) != NULL) {
        printf("File: %s\n", entry->d_name);
    }

    closedir(dir);
    return 0;
}
```

Output:

File before execution:


```
D:\c++>writenew
called write(3, new lines, 9)
called read(3,buff,10) it reads 10 bytes into buffer="of foo1.tx"
D:\c++>
```

File after execution:



13.Write a C program to simulate Bankers Algorithm for deadlock prevention and avoidance.

```
#include <stdio.h>
#include <stdbool.h>
#include<stdlib.h>
int i;
struct Node {
   int pid;
   int A;
   int B;
   int C;
};

struct Node* Allocation;
struct Node* Max;
struct Node* Need;
struct Node work;
bool complete[100];

void NeedCalculation(int numProcesses) {
   Need = (struct Node*)malloc(numProcesses * sizeof(struct Node));
   for (i = 0; i < numProcesses; i++) {
      int pid = Max[i].pid;
      int A = Max[i].A - Allocation[i].A;
      int B = Max[i].B - Allocation[i].B;
      int C = Max[i].C - Allocation[i].C;
      Need[i].pid = pid;
      Need[i].A = A;
      Need[i].B = B;
      Need[i].C = C;
   }
}

void BankersAlgo(int numProcesses) {
   bool allfinish = false;
   bool atleastonefinish;
     printf("The Process Excecution order is:\n");
   while (!allfinish) {
      atleastonefinish = false;
```

```
    for (i = 0; i < numProcesses; i++) {
      if (complete[i]) {
        continue;
      } else if (!complete[i] && Need[i].A <= work.A && Need[i].B <= work.B &&
Need[i].C <= work.C) {
        atleastonefinish = true;
        complete[i] = true;
        printf("P%d ",i);
        work.A += Allocation[i].A;
        work.B += Allocation[i].B;
        work.C += Allocation[i].C;
      }
    }

    if (!atleastonefinish) {
      printf("Deadlock Detected Unsafe state!\n");
      return;
    }

    allfinish = true;
    for (i = 0; i < numProcesses; i++) {
      if (!complete[i]) {
        allfinish = false;
        break;
      }
    }
  }

  printf("\nAll Processes are finished without leading to Deadlock--Safe State\n");
}

int main() {
  int numProcesses;

  printf("Enter the number of processes: ");
  scanf("%d", &numProcesses);
```

```
Allocation = (struct Node*)malloc(numProcesses * sizeof(struct Node));
Max = (struct Node*)malloc(numProcesses * sizeof(struct Node));

for (i = 0; i < numProcesses; i++) {
   printf("Enter Allocation for Process %d: ", i);
   int pid, A, B, C;
   scanf("%d %d %d %d", &pid, &A, &B, &C);
   Allocation[i].pid = pid;
   Allocation[i].A = A;
   Allocation[i].B = B;
   Allocation[i].C = C;

   printf("Enter MaxResources for Process %d: ", i);
   scanf("%d %d %d %d", &pid, &A, &B, &C);
   Max[i].pid = pid;
   Max[i].A = A;
   Max[i].B = B;
   Max[i].C = C;
}

NeedCalculation(numProcesses);

printf("Enter Available Resources: ");
int availableA, availableB, availableC;
scanf("%d %d %d", &availableA, &availableB, &availableC);
work.pid = -1;
work.A = availableA;
work.B = availableB;
work.C = availableC;

BankersAlgo(numProcesses);

free(Allocation);
free(Max);
free(Need);
```

```
    return 0;
}
```

Output:

```
Enter the number of processes: 5
Enter Allocation for Process 0: 0 0 1 0
Enter MaxResources for Process 0: 0 7 5 3
Enter Allocation for Process 1: 1 2 0 0
Enter MaxResources for Process 1: 1 3 2 2
Enter Allocation for Process 2: 2 3 0 2
Enter MaxResources for Process 2: 2 9 0 2
Enter Allocation for Process 3: 3 2 1 1
Enter MaxResources for Process 3: 3 2 2 2
Enter Allocation for Process 4: 4 0 0 2
Enter MaxResources for Process 4: 4 4 3 3
Enter Available Resources: 3 3 2
The Process Excecution order is:
P1 P3 P4 P0 P2
All Processes are finished without leading to Deadlock--Safe State
```

14.Write a C program to implement the Producer-Consumer problem using semaphores using UNIX/LINUX system calls.

```
#include<stdio.h>
#include<stdlib.h>
 int mutex = 1, full = 0, empty =
 3, x = 0;int main () {
  int n;
  void
  producer
  (); void
  consume
```

```
 r ();int
 wait (int);
 int signal (int);
 printf
 ("\n1.Producer\n2.Consumer\n3.Exit");
 while (1)
{
 printf ("\nEnter your
 choice:");scanf ("%d",
 &n);
 swi
tch
(n) {
cas
e 1:
 if ((mutex == 1) &&
 (empty != 0))producer ();
 else
 printf ("Buffer is
 full!!");break;
case 2:
 if ((mutex == 1) &&
 (full != 0))consumer
 ();
 else
 printf ("Buffer is
 empty!!");break;
case 3:
 e
 x
 i
 t
 (
 0
 )
 ;
 b
```

```
 r
 e
 a
 k
 ;
 }

}
 return 0;
 }

int wait (int s)
{
 return (--s);
}
int signal (int s) {
 return (++s);
}
void producer ()
{
 mutex = wait
 (mutex);full =
 signal (full);
 empty = wait
 (empty);x++;
 printf ("\nProducer produces the item
 %d", x);mutex = signal (mutex);
}
void consumer ()
{
 mutex = wait
 (mutex);full =
 wait (full);
 empty = signal (empty);
 printf ("\nConsumer consumes item
 %d", x);x--;
```

```
 mutex = signal (mutex);
}
```

Output:

```
[188r1a0501@localhost ~]$ vi pc.c
[188r1a0501@localhost ~]$ gcc  pc.c
[188r1a0501@localhost ~]$ ./a.out

1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:1

Producer produces the item 3
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3
[188r1a0501@localhost ~]$
```

15.Write a C program to simulate Paging memory management technique.

```
#include <stdio.h>

 int main() {
    int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;
    int s[10], fno[10][20];

    // Input memory size
    printf("Enter the memory size: ");
    scanf("%d", &ms);

    // Input page size
    printf("Enter the page size: ");
    scanf("%d", &ps);

    // Calculate number of pages available in memory
    nop = ms / ps;
    printf("The number of pages available in memory are: %d\n", nop);

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

    rempages = nop; // Remaining pages initially is total pages

    // Loop for each process to input page requirements and page table
    for(i = 1; i <= np; i++) {
       printf("Enter the number of pages required for process %d: ", i);
       scanf("%d", &s[i]);

       if(s[i] > rempages) {
          printf("Memory is Full\n");
          break;
       }

       rempages -= s[i]; // Decrement remaining pages

       printf("Enter the page table for process %d:\n", i);
       for(j = 0; j < s[i]; j++) {
```

```
            scanf("%d", &fno[i][j]);
        }
    }

    // Input logical address details to find physical address
    printf("Enter Logical Address to find Physical Address\n");
    printf("Enter process number, page number, and offset: ");
    scanf("%d %d %d", &x, &y, &offset);

    // Check for valid input
    if(x > np || y >= s[x] || offset >= ps) {
        printf("Invalid Process or Page Number or Offset\n");
    } else {
        // Calculate physical address
        pa = fno[x][y] * ps + offset;
        printf("The Physical Address is: %d\n", pa);
    }

    return 0;
}
```

Output:

```
Enter the memory size -- 1000

Enter the page size -- 100

The no. of pages available in memory are -- 10
Enter number of processes -- 3

Enter no. of pages required for p[1]-- 4

Enter pagetable for p[1] --- 8 6
9 5

Enter no. of pages required for p[2]-- 5

Enter pagetable for p[2] --- 1 4 5 7 3

Enter no. of pages required for p[3]-- 5

Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 2
3
60

The Physical Address is -- 760
--------------------------------
Process exited after 87.72 seconds with return value 0
Press any key to continue . . .
```

16.Write a C program to simulate Segmentation memory managementtechnique.

```
#include <stdio.h>

int main() {
        int i, base = 0, offset, np, ns, msize, ad = 0, ss[10], seg_num, logical_offset;

        // Input total memory size
        printf("Enter the memory size: ");
        scanf("%d", &msize);

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

        // Input segment sizes and check for memory allocation
        for (i = 0; i < ns; i++) {
           printf("Enter the segment %d size: ", i + 1);
           scanf("%d", &ss[i]);

           ad += ss[i];  // Update allocated memory

           if (ad > msize) {
              printf("Segment %d cannot be allocated due to insufficient memory\n", i + 1);
              ad -= ss[i];  // Revert allocation if it exceeds memory size
              ss[i] = 0;    // Mark segment as not allocated
           }
        }

        // Print Segment Table
        printf("\nSEGMENT TABLE\n");
        ad = 0;
        printf("\t\tBASE\tOFFSET\n");
        for (i = 0; i < ns; i++) {
           printf("Segment %d: %d\t%d\n", i + 1, ad, ad + ss[i]);
           ad += ss[i];  // Update base address for next segment
        }

        // Input logical address details to find physical address
        printf("Enter Segment number and Offset to find Physical Address: ");
```

```
    scanf("%d %d", &seg_num, &logical_offset);

    // Check for valid input
    if (seg_num > ns || logical_offset >= ss[seg_num - 1]) {
        printf("Invalid Segment Number or Offset\n");
    } else {
        // Calculate physical address
        base = 0;
        for (i = 0; i < seg_num - 1; i++) {
            base += ss[i];
        }
        int physical_address = base + logical_offset;
        printf("The Physical Address is: %d\n", physical_address);
    }

    return 0;
}
```

Output:

```
Enter the memory size: 50
Enter the number of segments: 5
Enter the segment 1 size: 20
Enter the segment 2 size: 15
Enter the segment 3 size: 10
Enter the segment 4 size: 5
Enter the segment 5 size: 13
Segment 5 cannot be allocated due to insufficient memory

SEGMENT TABLE
                BASE    OFFSET
Segment 1: 0    20
Segment 2: 20   35
Segment 3: 35   45
Segment 4: 45   50
Segment 5: 50   50
Enter Segment number and Offset to find Physical Address: 2 20
Invalid Segment Number or Offset
```

17.Write the C programs to illustrate the IPC mechanisms.

FIFO

```
#include <stdio.h>
int main()
 {
  int incomingStream[] = {4, 1, 2, 4, 5};
  int pageFaults = 0;
  int frames = 3;
  int m, n, s, pages;
  pages = sizeof(incomingStream) / sizeof(incomingStream[0]);
  printf(" Incoming \t\t Frame 1 \t\t Frame 2 \t\t Frame 3 ");
  int temp[frames];
  for (m = 0; m < frames; m++)
  {
     temp[m] = -1;
  }
  for (m = 0; m < pages; m++)
  {
     s = 0;
     for (n = 0; n < frames; n++)
     {
        if (incomingStream[m] == temp[n])
        {
           s++;
           pageFaults--;
        }
     }
     pageFaults++;
     if ((pageFaults <= frames) && (s == 0))
     {
        temp[m] = incomingStream[m];
     }
     else if (s == 0)
     {
        temp[(pageFaults - 1) % frames] = incomingStream[m];
     }
     printf("\n");
     printf("%d\t\t\t", incomingStream[m]);
     for (n = 0; n < frames; n++)
```

```
    {
      if (temp[n] != -1)
        printf(" %d\t\t\t", temp[n]);
      else
        printf(" - \t\t\t");
    }
  }
  printf("\nTotal Page Faults:\t%d\n", pageFaults);
  return 0;
}
```

```
 Incoming            Frame 1             Frame 2             Frame 3
4                    4                   -                   -
1                    4                   1                   -
2                    4                   1                   2
4                    4                   1                   2
5                    5                   1                   2
Total Page Faults:   4
```

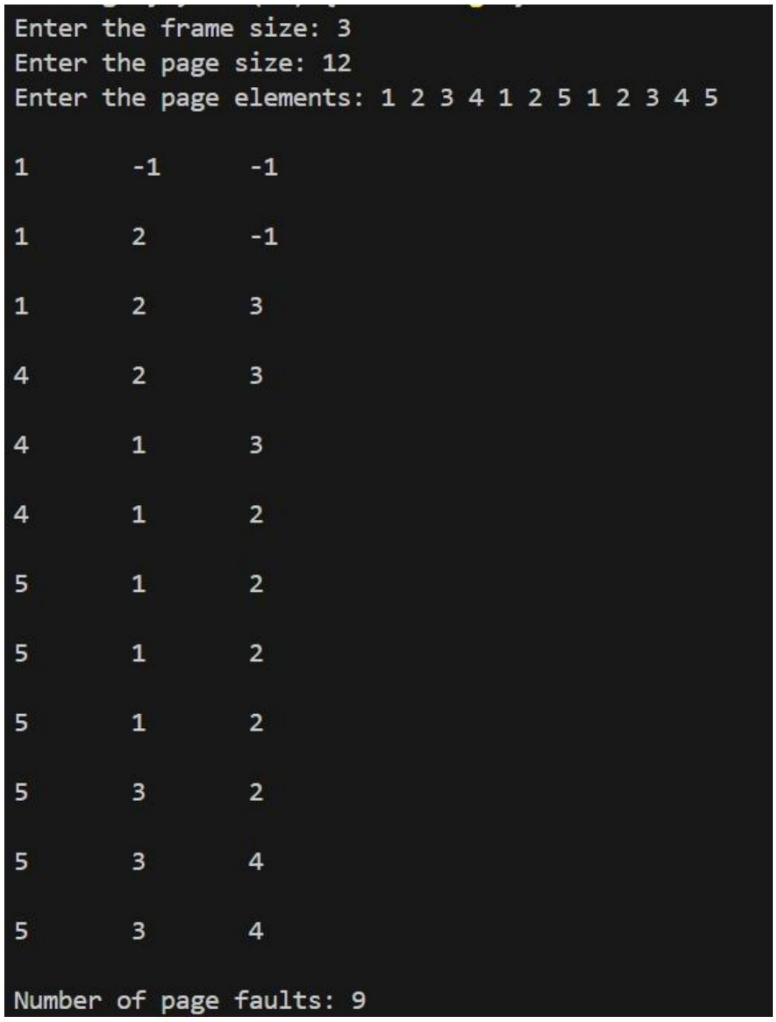
18. Write C programs to implement Page Replacement algorithms.

FIFO

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

void display(int* fr, int frame_size); // Function prototype for displaying page
frames

int main() {
   int frame_size, page_size;
   printf("Enter the frame size: ");
   scanf("%d", &frame_size);

   int* fr = (int*)malloc(frame_size * sizeof(int));
   for (int i = 0; i < frame_size; i++) {
      fr[i] = -1; // Initialize page frames with -1
   }

   printf("Enter the page size: ");
   scanf("%d", &page_size);

   int page[page_size];
   printf("Enter the page elements: ");
   for (int i = 0; i < page_size; i++) {
      scanf("%d", &page[i]);
   }

   int pf = 0; // Page fault count
   int flag1 = 0, flag2 = 0, top = 0; // Flags and top pointer

   // Iterate over the page reference sequence
   for (int j = 0; j < page_size; j++) {
      flag1 = 0;
      flag2 = 0;

      // Check if the page is already in a frame
      for (int i = 0; i < frame_size; i++) {
         if (fr[i] == page[j]) {
            flag1 = 1; // Page found in frame
            flag2 = 1; // No page fault
            break;
         }
```

```
        }

        // If page is not in a frame, find an empty frame
        if (flag1 == 0) {
            for (int i = 0; i < frame_size; i++) {
                if (fr[i] == -1) { // Find empty frame
                    fr[i] = page[j];
                    flag2 = 1; // Page fault
                    pf++; // Increment page fault count
                    break;
                }
            }
        }

        // If no empty frame, use FIFO to replace the oldest page
        if (flag2 == 0) {
            fr[top] = page[j]; // Replace page at the top
            top = (top + 1) % frame_size; // Update top pointer to rotate frames
            pf++; // Increment page fault count
        }

        display(fr, frame_size); // Display current page frames
    }

    printf("\nNumber of page faults: %d\n", pf); // Print total page faults

    free(fr); // Free dynamically allocated memory

    return 0; // Indicate successful execution
}

// Function to display current page frames
void display(int* fr, int frame_size) {
    printf("\n");
    for (int i = 0; i < frame_size; i++) {
        printf("%d\t", fr[i]);
    }
    printf("\n");
}
```

Output:



LRU

```
#include <stdio.h>

#define N 3
int pg_index,i,j;
void lruAlgo(int pageArr[], int length) {
   int fr[N][2] = {{-1, 0}, {-1, 0}, {-1, 0}};
   int pf = 0;

   for (pg_index = 0; pg_index < length; pg_index++) {
      int pg = pageArr[pg_index];
      int flag1 = 0;
      int flag2 = 0;

      // Check if page is already in frames
      for (i = 0; i < N; i++) {
         if (fr[i][0] == pg) {
            fr[i][1]++; // Increment time stamp
```

```
            flag1 = 1;
            flag2 = 1;
            break;
        }
    }

    if (flag1 == 0) {
        // If page not in frames, find an empty frame
        for (j = 0; j < N; j++) {
            if (fr[j][0] == -1) {
                fr[j][0] = pg;
                fr[j][1]++; // Increment time stamp
                flag2 = 1;
                pf++;
                break;
            }
        }
    }

    if (flag2 == 0) {
        // Find the least recently used page not currently in memory
        int leastUsedPageIndex = 0;
        int leastTime = fr[0][1];

        for (i = 0; i < N; i++) {
            if (fr[i][1] < leastTime) {
                leastUsedPageIndex = i;
                leastTime = fr[i][1];
            }
        }

        // Replace the least recently used page not currently in memory
        fr[leastUsedPageIndex][0] = pg;
        fr[leastUsedPageIndex][1]++; // Increment time stamp
        pf++;
    }

    // Display current page frames
    printf("Page frames after reference %d: ", pg);
    for (i = 0; i < N; i++) {
        printf("%d\t", fr[i][0]);
    }
    printf("\n");
  }

  // Print total page faults
  printf("Number of page faults: %d\n", pf);
}
```

```
int main() {
   // int pageArr[] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
   int pageArr[] = {1,2,3,4,1,1,5};
   int length = sizeof(pageArr) / sizeof(pageArr[0]);

   lruAlgo(pageArr, length);

   return 0;
}
```

Output:

```
Enter the number of frames:
3
Page frames after reference 1:  1 -1 -1
Page frames after reference 2:  1 2 -1
Page frames after reference 3:  1 2 3
Page frames after reference 4:  4 2 3
Page frames after reference 1:  4 1 3
Page frames after reference 2:  4 1 2
Page frames after reference 5:  5 1 2
Page frames after reference 1:  5 1 2
Page frames after reference 2:  5 1 2
Page frames after reference 3:  3 1 2
Page frames after reference 4:  3 4 2
Page frames after reference 5:  3 4 5
Number of page faults: 10
```

OPTIMAL

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

#define N 3 // Number of frames

bool isContains(int fr[], int pg) {
   for (int i = 0; i < N; i++) {
      if (fr[i] == pg) {
         return true;
      }
   }
   return false;
}

int findOptimalReplacement(int fr[], int pageArr[], int pg_index, int length) {
   int max_distance = -1; // Maximum distance in the future
   int replace_idx = 0; // Index to replace

   for (int i = 0; i < N; i++) {
      bool found = false;
      for (int j = pg_index + 1; j < length; j++) {
         if (fr[i] == pageArr[j]) {
            found = true;
            if (j > max_distance) {
               max_distance = j; // Farthest in the future
               replace_idx = i;
            }
            break; // We found the position, no need to look further for this frame
         }
      }

      if (!found) {
         // If this page is not found in the future, it's the best to replace
         return i;
      }
   }

   return replace_idx;
}

void optimalAlgorithm(int pageArr[], int length) {
   int fr[N] = {-1, -1, -1}; // Frame array
   int pf = 0; // Page faults count
```

```
    for (int pg_index = 0; pg_index < length; pg_index++) {
        int pg = pageArr[pg_index];
        int frameIndex = -1;

        if (isContains(fr, pg)) {
            // If page is already in frames, no page fault
            continue;
        } else {
            pf++; // Increment page fault count
            int emptyFrameIdx = -1;
            // Check for an empty frame to place the new page
            for (int i = 0; i < N; i++) {
                if (fr[i] == -1) {
                    emptyFrameIdx = i;
                    break;
                }
            }

            if (emptyFrameIdx != -1) {
                // Insert into empty frame
                fr[emptyFrameIdx] = pg;
            } else {
                // Find the optimal page to replace
                int replace_idx = findOptimalReplacement(fr, pageArr, pg_index, length);
                fr[replace_idx] = pg; // Replace with new page
            }
        }

        // Display current page frames
        printf("Page frames after reference %d: ", pg);
        for (int i = 0; i < N; i++) {
            printf("%d ", fr[i]);
        }
        printf("\n");
    }

    // Display the total page faults
    printf("Number of page faults: %d\n", pf);
}

int main() {
    int pageArr[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1}; // Test input
    int length = sizeof(pageArr) / sizeof(pageArr[0]);
```

```
    optimalAlgorithm(pageArr, length);

    return 0;
}
```

Output:

```
Page frames after reference 7: 7 -1 -1
Page frames after reference 0: 7 0 -1
Page frames after reference 1: 7 0 1
Page frames after reference 2: 2 0 1
Page frames after reference 3: 2 0 3
Page frames after reference 4: 2 4 3
Page frames after reference 0: 2 0 3
Page frames after reference 1: 2 0 1
Page frames after reference 7: 7 0 1
Number of page faults: 9
```