Orphan process:

Step 1: I create a new process using fork system call. Step 2: This splits the program into two, one running as the child and one as the parent. Step 3: I check the return value of fork system call: 3.1. If it's 0, I'm in the child. 3.2. If it's not 0, I'm in the parent. Step 4: In the child: 4.1. I clear the terminal. 4.2. I print that this is the child process and show its process ID. 4.3. I also print the child's parent process ID. Step 5: In the parent:5.1. I print that this is the parent process and show its process ID. 5.2. I also print the parent's parent process ID. Step 6: After printing, both processes end.

```
#include<stdio.h>

main()

{

 if(fork()==0)

 {

   system("clear");

   printf("CHILD:I am child & my ID is:%d\n",getpid());

   printf("CHILD:My parent is:%d\n",getppid());

 }

 else

 {

   printf("PARENT:I am parent %d\n",getpid());

   printf("PARENT:My parent is:%d\n",getppid());

 }

}
```

B)PARENT CHILD PROCESS

Step 1: I call fork system call to create a new process, so now there's a parent and a child running at the same time. Step 2: I check the return value of fork system call to see which process I'm in. Step 3: If it's 0, I'm in the child: 3.1. I clear the terminal for clean output. 3.2. I print that this is the child process and show its process ID. 3.3. I also print the process ID of the child's parent. Step 4: If it's greater than 0, I'm in the parent: 4.1. I print that this is the parent process along with its own process ID. 4.2. I print the parent's parent process ID. Step 5: After printing, both parent and child just finish execution

```
#include<stdio.h>
main()
{

 if(fork()==0)
 {
   system("clear");
   printf("CHILD:I am & my ID: %d \n",getpid());
   printf("CHILD:MY parent is:%d \n",getppid());
 }
 else
 {
   printf("PARENT:I am parent: %d \n",getpid());
   sleep(2);
   printf("PARENT:My parent is: %d\n",getppid());
 }

}
```

C) WAITING PROCESS

Step 1: I declare the variables needed to keep track of process IDs. Step 2: I call fork system call to create the first child. 2.1. If I'm in the first child, I print its process ID, say it's the first child, then print that it's terminating and exit. Step 3: If I'm still in the parent, I call fork system call again to create the second child. 3.1. If I'm in the second child, I print its process ID, say it's the second child, then print that it's terminating and exit. Step 4: If I'm in the parent after both forks, I use wait system call to wait for each child to finish. 4.1. For every child that ends, I print the process ID of the one that terminated. Step 5: Once both children are done, I let the parent finish execution cleanly and end the program.

```
#include<stdio.h>
main()
{
  int pid,dip,cpid;
  pid=fork();
  if(pid==0)
  {
    printf("1st Child Process ID is %d\n",getpid());
    printf("1st Child Process TERMINATING FROM THE MEMORY\n");
  }
  else
  {
    dip=fork();
    if(dip==0)
    {
      printf("2nd Child Process ID is %d\n",getpid());
      printf("2nd Child Process TERMINATING FROM THE MEMORY\n");
    }
    else
    {
      cpid=wait(0);
      printf("Child with pid:%d is dead\n",cpid);
```

```
        cpid=wait(0);
        printf("Child with pid:%d is dead\n",cpid);
      }
   }
}
```

D) ZOMBIE PROCESS

Step 1: I create a new process using fork system call. Step 2: I clear the terminal, so the output stays clean. Step 3: I print “My ID is …” showing the process ID of the running process. Step 4: If pid > 0, I’m in the parent process: 4.1- I print “My parent ID is …” showing the parent’s process ID. 4.2- I pause for 5 seconds using sleep system call for 5 seconds. Step 5: Both parent and child processes finish after printing their messages.

```
#include<stdio.h>
main()
{
   int pid = fork();
   system("clear");
   printf("My ID is %d\n",getpid());
   if(pid>0)
   {
      printf("My parent ID is %d\n",getpid());
      sleep(5);
   }
}
```

CHAT PROGRAM 1 (Sender Process): ALGORITHM: Step 1: First, I create a message queue using the system call for inter-process communication. Step 2: I check if it was created successfully. Step 3: If it fails, I print an error and stop the program. Step 4: If it's fine, I wait for the user to type the text they want to send. Step 5: I send that text into the queue using the message-send system call. Step 6: After sending, I show a confirmation that it was sent successfully. Step 7: I also display the same message back to the user, so they know exactly what went into the queue. Step 8: End the program once the sending part is done.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<unistd.h>
#include<string.h>
int main()
{
   int mid;
   char mess[20];
   printf("\n\n enter process communication using message queue: ");
   mid=msgget(27,IPC_CREAT|0777);
   if(mid==-1)
   {
      printf("\n\t invalid message id.");
      exit(1);
   }
   printf("\n enter the message text: ");
   scanf("%s",mess);
   msgsnd(mid,mess,strlen(mess),0);
   printf("\n\nthe message has been sent.\n");
   printf("\n the sent message is : %s\n",mess);
   return 0;
}
```

B) CHAT PROGRAM 2 (Receiver Process): ALGORITHM: Step 1: First, I try to access the message queue. If it's not there already, I create it. Step 2: I check if this worked. Step 3: If it didn't, I print an error message and exit the program. Step 4: If it's fine, I prepare to receive a message from the queue. Step 5: I use the receive system call to get the message. Step 6: Once I have it, I display the message to the user. Step 7: After confirming it's shown properly, I end the program.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <string.h>
int main()
{
   int mid;
   char mess[20];
   printf("\n\n process communication using message queue: ");
   mid=msgget(27,IPC_CREAT|0777);
   if(mid==-1)
   {
      printf("\n\t invalid message id.");
      exit(1);
   }
   msgrcv(mid,mess,100,0,0);
   printf("\n the received message is : %s\n",mess);
   return 0;
}
```

pthread

Step 1: First, I decide how many worker threads I need and how many jobs each will do. I store these in variables. Step 2: Then I set up a barrier so all threads can wait for each other before moving ahead. Step 3: I create the threads, give each one a unique ID, and pass that along with the function it should run. Step 4: The main program waits for all threads to finish by joining them. Step 5: Inside the thread function, it gets its ID, processes jobs in a loop, prints which worker is doing which job, and waits at the barrier after each job. Step 6: Once everything's done, I destroy the barrier and end the program.

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <semaphore.h>

void handler(void*ptr);
pthread_barrier_t barrier;

int worker =2;
int job = 4;

int main()
{
   int i = 0;
   pthread_t thread_a;

   pthread_barrier_init(&barrier, NULL, worker);

   for (i; i < worker ; i++)
   {
```

```
      int *n_workers = malloc(sizeof(*n_workers));
      *n_workers = 4;

      pthread_create (&thread_a, NULL, (void *) &handler, n_workers);
   }
   pthread_join(thread_a, NULL);

   pthread_barrier_destroy(&barrier);pthread_exit(0);
}
void handler ( void *ptr )
{
   int x = *((int *) ptr);

   int i = 0;


// [Code from bottom of image]
   for (i; i< job; i++)
   {
      printf("worker %d: Doing Job %d\n", x, i);

      pthread_barrier_wait(&barrier);
   }
}
```

1.FCFS

Step 1. I ask for the number of processes, n. Step 2. I create space to store, for each process: an ID, its arrival time (AT), its burst time (BT), its completion time (CT), its turnaround time (TAT), and its waiting time (WT). Step 3. For each process i from 1 to n: 3.1 I assign a process ID (like 1, 2, 3, ...). 3.2 I read that process's arrival time and burst time and store them. Step 4. I arrange the processes by arrival time so the early starriving process comes first. 4.1 To keep things consistent, I always move the arrival time, burst time and process ID together when I swap two processes. Step 5. I set a time variable to 0 to represent the current clock. P a g e | 3 |DEVANSH POKHARIYA 24BCA0014 Step 6. For each process in the sorted order: 6.1 If time is less than the process's arrival time, I move time forward to that arrival time (the CPU is idle until the process arrives). 6.2 I start the process at the current time and run it for its burst time, so I add the burst time to time. 6.3 I record the completion time CT as the updated time. 6.4 I compute turnaround time: TAT = CT - AT. 6.5 I compute waiting time: WT = TAT - BT. Step 7. While processing, I keep running totals of all waiting times and all turnaround times so I can compute averages later. Step 8. After all processes are done, I print a table row for each process containing: PID, AT, BT, TAT and WT. Step 9 . I compute the average waiting time as (sum of all WT) divided by n. Step 10. I compute the average turnaround time as (sum of all TAT) divided by n. Step 11. I display both average values. Formulas summary (for each process): a. Completion time: set when the process finishes (the current clock). b. Turnaround time = Completion time − Arrival time. c. Waiting time = Turnaround time − Burst time.

```
#include<stdio.h>

int main()
{
   int n;
   printf("Enter number of processes: ");
   scanf("%d", &n);

   int pid[n], at[n], bt[n], ct[n], tat[n], wt[n];
   for (int i = 0; i < n; i++) {
      pid[i] = i+1;
      printf("Enter Arrival Time and Burst Time of P%d: ", i+1);
      scanf("%d %d", &at[i], &bt[i]);
   }

   for (int i = 0; i < n-1; i++) {
```

```
   for (int j = i+1; j < n; j++) {
      if (at[i] > at[j]) {
         int t;
         t = at[i]; at[i] = at[j]; at[j] = t;
         t = bt[i]; bt[i] = bt[j]; bt[j] = t;
         t = pid[i]; pid[i] = pid[j]; pid[j] = t;
      }
   }
}

int time = 0;
for (int i = 0; i < n; i++) {
   if (time < at[i]) time = at[i];
   ct[i] = time;
   tat[i] = ct[i] - at[i];
   wt[i] = tat[i] - bt[i];
}

printf("\nFCFS Scheduling\n");
printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
float sumwt=0;
float sumtat=0;
for (int i = 0; i < n; i++) {
   sumwt = sumwt + wt[i];
   sumtat = sumtat + tat[i];
   printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], at[i], bt[i], ct[i], tat[i], wt[i]);
}

printf("Average Waiting time is : %.2f msec\n", sumwt/n);
printf("Average Turnaround time is : %.2f msec\n", sumtat/n);
return 0;
```

}

2.SJF (NON-PREEMPTIVE)

Step 1. First, I ask for the number of processes, n. Step 2. I create arrays to store for each process: 2.1 Process ID (PID) 2.2 Arrival Time (AT) 2.3 Burst Time (BT) 2.4 Completion Time (CT) 2.5 Turnaround Time (TAT) 2.6 Waiting Time (WT) 2.7 A flag done[] to check if a process is already completed or not. Step 3. For each process, I assign a PID (P1, P2, ...), and then I take its arrival time and burst time as input. Initially, I mark all processes as not done. Step 4. I start with time = 0 and completed = 0. Step 5. While the number of completed processes is less than n: 5.1 I look for the process that has already arrived (AT <= time), is not completed, and has the minimum burst time among all such processes. 5.2 If no process is ready at the current time (i.e., no process has arrived yet), I increase time by 1. 5.3 Otherwise, I pick the process with the shortest burst time and run it. 5.4 I increase time by the burst time of that process. 5.5 I set its Completion Time (CT) = current time. 5.6 I calculate its Turnaround Time (TAT) = CT – AT. 5.7 I calculate its Waiting Time (WT) = TAT – BT. 14 5.8 I mark the process as done. 5.9 I increase the completed counter by 1. Step 6. Once all processes are finished, I print the scheduling table showing PID, AT, BT, TAT, and WT for each process. Step 7. While printing, I also calculate the total waiting time and total turnaround time. Step 8. Finally, I calculate and display: 8.1 Average Waiting Time = (Sum of WT) / n 8.2 Average Turnaround Time = (Sum of TAT) / n

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

int main()
{
   int n;
   printf("Enter number of processes: ");
   scanf("%d", &n);

   int pid[n], at[n], bt[n], ct[n], tat[n], wt[n], done[n];
   for (int i = 0; i < n; i++) {
      pid[i] = i+1;
      printf("Enter Arrival Time and Burst Time of P%d: ", i+1);
      scanf("%d %d", &at[i], &bt[i]);
```

```
    done[i] = 0;
}

int time = 0, completed = 0;
while (completed < n) {
    int idx = -1, minBT = INT_MAX;
    for (int i = 0; i < n; i++) {
        if (at[i] <= time && !done[i] && bt[i] < minBT) {
            minBT = bt[i];
            idx = i;
        }
    }

    if (idx == -1) {
        time++;
    } else {
        time += bt[idx];
        ct[idx] = time;
        tat[idx] = ct[idx] - at[idx];
        wt[idx] = tat[idx] - bt[idx];
        done[idx] = 1;
        completed++;
    }
}

printf("\nSJF Scheduling\n");
printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
float sumwt=0;
float sumtat=0;
for (int i = 0; i < n; i++) {
    sumwt = sumwt + wt[i];
```

```
        sumtat = sumtat + tat[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], at[i], bt[i], ct[i], tat[i], wt[i]);
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], at[i], bt[i], ct[i], tat[i], wt[i]);
    }

    printf("Average Waiting time is : %.2f msec\n", sumwt/n);
    printf("Average Turnaround time is : %.2f msec\n", sumtat/n);
    return 0;
}
```

3.PRIORITY (NON-PREEMPTIVE)

Step 1. First, I ask for the number of processes, n. Step 2. I prepare arrays to store, for each process: 2.1 Process ID (PID) 2.2 Arrival Time (AT) 2.3 Burst Time (BT) 2.4 Priority (PR) 2.5 Completion Time (CT) 2.6 Turnaround Time (TAT) 2.7 Waiting Time (WT) 2.8 A flag done[] to check if the process is finished or not Step 3. For each process, I assign a PID (P1, P2, ...) and take its arrival time, burst time, and priority as input. Initially, I mark all processes as not done. Step 4. I set time = 0 and completed = 0. Step 5. While the number of completed processes is less than n: • I search through all processes and look for the one that has: 5.1 Already arrived (AT <= time) 5.2 Not yet completed (done = 0) 5.3 The highest priority (here, smaller number means higher priority). • If no such process is found at the current time, I increase time by 1 (CPU is idle). • Otherwise: 5.4 I select that process. 5.5 I increase time by its burst time (since non-preemptive → once started, it runs till completion). 5.6 I set its Completion Time (CT) = current time. 5.7 I calculate its Turnaround Time (TAT) = CT – AT. 5.8 I calculate its Waiting Time (WT) = TAT – BT. 5.9 I mark this process as done. 5.10I increase completed by 1. Step 6. After finishing all processes, I print a table showing PID, AT, BT, Priority, TAT, and WT. Step 7. While printing, I also add up the total Waiting Times and total Turnaround Times. Step 8. Finally, I compute and display: 8.11 Average Waiting Time = (Sum of all WT) / n 8.12 Average Turnaround Time = (Sum of TAT) / n

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

int main()
{
    int n;
    printf("Enter number of processes: ");
```

```
scanf("%d", &n);

int pid[n], at[n], bt[n], pr[n], ct[n], tat[n], wt[n], done[n];
for (int i = 0; i < n; i++) {
   pid[i] = i+1;
   printf("Enter Arrival Time, Burst Time and Priority of P%d: ", i+1);
   scanf("%d %d %d", &at[i], &bt[i], &pr[i]);
   done[i] = 0;
}

int time = 0, completed = 0;
while (completed < n) {
   int idx = -1, bestPr = INT_MAX;
   for (int i = 0; i < n; i++) {
      if (at[i] <= time && !done[i] && pr[i] < bestPr) {
         bestPr = pr[i];
         idx = i;
      }
   }

   if (idx == -1) {
      time++;
   } else {
      time += bt[idx];
      ct[idx] = time;
      tat[idx] = ct[idx] - at[idx];
      wt[idx] = tat[idx] - bt[idx];
      done[idx] = 1;
      completed++;
   }
}
```

```
    printf("\nPriority Scheduling\n");
    printf("PID\tAT\tBT\tPR\tCT\tTAT\tWT\n");
    float sumwt=0;
    float sumtat=0;
    for (int i = 0; i < n; i++) {
        sumwt = sumwt + wt[i];
        sumtat = sumtat + tat[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], at[i], bt[i], pr[i], ct[i], tat[i], wt[i]);
    }

    printf("Average Waiting time is : %.2f msec\n", sumwt/n);
    printf("Average Turnaround time is : %.2f msec\n", sumtat/n);

    return 0;
}
```

4.RR (ROUND ROBIN)

Step 1. First, I ask for the number of processes n. Step 2. I prepare arrays to store for each process: 2.1 Process ID (PID) 2.2 Arrival Time (AT) 2.3 Burst Time (BT) 2.4 Remaining Time (RT) → initially equal to burst time 2.5 Completion Time (CT) 2.6 Turnaround Time (TAT) 2.7 Waiting Time (WT) Step 3. For each process, I assign a PID (P1, P2, ...), then I take its arrival time and burst time as input. I also set RT = BT. Step 4. I then take the quantum time (QT) as input. Step 5. I initialise: 5.1 time = 0 (current time on the CPU) 5.2 completed = 0 (number of processes finished) Step 6. While completed < n (until all processes are done): 6.1 I set a flag to check if at least one process executes in this cycle. 6.2 I loop through all processes one by one (like a circular queue). ▪ If a process has already arrived (AT <= time) and still has remaining time (RT > 0): 6.3 I set the flag to true (means something executed this cycle). 6.4 If its remaining time is less than or equal to the quantum time: 6.5 I run it completely. 6.6 I increase the clock time by its remaining time. 6.7 I set RT = 0 (process finished). 6.8 I record its Completion Time = current time. 6.9 I calculate TAT = CT – AT. 6.10 I calculate WT = TAT – BT. 6.11 I increase completed by 1. 6.12 Else (its remaining time is more than the quantum time): 6.13 I run it only for QT. 6.14 I decrease its RT by QT. 6.15 I increase the clock time by QT. 6.16 If no process was ready in this cycle (flag = 0), I simply increase time by 1 (CPU idle). Step 7. After all processes are finished, I print a table showing: 6.17PID, AT, BT, TAT, and WT Step 8. While printing, I also keep track of the total waiting time and total turnaround time. Step 9. Finally, I calculate and display: 6.18Average Waiting Time = (Sum of all WT) / n 6.19Average Turnaround Time = (Sum of all TAT) / n

```
#include <stdio.h>

int main()
{
   int n, tq;
   printf("Enter number of processes: ");
   scanf("%d", &n);

   int pid[n], at[n], bt[n], rt[n], ct[n], tat[n], wt[n];
   for (int i = 0; i < n; i++) {
      pid[i] = i+1;
      printf("Enter Arrival Time and Burst Time of P%d: ", i+1);
      scanf("%d %d", &at[i], &bt[i]);
```

```
    rt[i] = bt[i];
}

printf("Enter Time Quantum: ");
scanf("%d", &tq);

int time = 0, completed = 0;
while (completed < n) {
    int flag = 0;
    for (int i = 0; i < n; i++) {
        if (at[i] <= time && rt[i] > 0) {
            flag = 1;
            if (rt[i] <= tq) {
                time += rt[i];
                rt[i] = 0;
                ct[i] = time;
                tat[i] = ct[i] - at[i];
                wt[i] = tat[i] - bt[i];
                completed++;
            } else {
                rt[i] -= tq;
                time += tq;
            }
        }
    }
    if (!flag) time++;
}

printf("\nRound Robin Scheduling\n");
printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
float sumwt=0;
```

```
    float sumtat=0;
    for (int i = 0; i < n; i++) {
        sumwt = sumwt + wt[i];
        sumtat = sumtat + tat[i];
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", pid[i], at[i], bt[i], ct[i], tat[i], wt[i]);
    }

    printf("Average Waiting time is : %.2f msec\n", sumwt/n);
    printf("Average Turnaround time is : %.2f msec\n", sumtat/n);
    return 0;
}
```

semaphore

STEP 1: First, I initialise a semaphore variable called mutex with an initial value of 1. This ensures mutual exclusion so that only one thread can enter the critical section at a time. STEP 2: Then, I declare two thread variables t1 and t2. STEP 3: I create the first thread t1 which will run the function thread. STEP 4: After starting the first thread, I make the main program sleep for 2 seconds before creating the second thread. STEP 5: Once the 2 seconds pass, I create the second thread t2, which also runs the same thread function. STEP 6: Inside the thread function, the first step is that the thread tries to acquire the semaphore lock using sem_wait. STEP 7: If the semaphore is available, the thread enters the critical section and prints "Entered..". STEP 8: Then the thread goes to sleep for 4 seconds while holding the lock. STEP 9: After the sleep, the thread prints "Just Exiting..." to indicate that it is leaving the critical section. STEP 10: Finally, the thread releases the semaphore lock using sem_post, so that another waiting thread can enter. STEP 11: Back in the main program, I wait for both threads to complete by using pthread_join for t1 and t2. STEP 12: Once both threads finish execution, I destroy the semaphore to free the resources. STEP 13: The program then exits successfully.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;
void* thread(void* arg)
{
```

```
    sem_wait(&mutex);
    printf("\nEntered..\n");

    sleep(4);

    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}

int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

BANKERS ALGORITHMS

STEP 1: First, I ask the user to enter the number of processes. STEP 2: Then, I take the names of all the processes. STEP 3: Next, I ask the user to enter the number of resources in the system. STEP 4: I also take the names of the resources one by one. STEP 5: After that, I ask the user to enter how many instances of each resource are available in the system. STEP 6: Then, I ask the user to input the Allocation Matrix, which shows how many instances of each resource are currently allocated to every process. STEP 7: After that, I take the Maximum Matrix, which shows the maximum demand of each process for every resource. STEP 8: Once I have both matrices, I calculate the Need Matrix by subtracting allocation from maximum for every process and resource. STEP 9: I then display this Need Matrix so that it is clear what each process still requires. STEP 10: Next, I prepare a temporary array called Work, which is initially set equal to the available resources. STEP 11: I also prepare a Finish array, which keeps track of which processes have finished execution. Initially, I mark all processes as unfinished. STEP 12: Now I start the safety check loop: 12.1 I look for any process that has not yet finished. 12.2 For each unfinished process, I check if its remaining needs can be satisfied by the currently available resources in the Work array. 12.3 If its needs can be satisfied, I assume the process runs to completion. 12.4 Once it finishes, I release its allocated resources back to the Work array (by adding allocation to work). 12.5 I then mark this process as finished. STEP 13: I repeat this checking for all processes until I can no longer find any more processes that can finish. STEP 14: After this loop, I check the Finish array: 14.1 If all processes are marked as finished, I declare that the system is in a Safe State. 14.2 If even one process remains unfinished, I declare that the system is Not Safe.

```
#include<stdio.h>
#include<string.h>
int main()
{
   int i,j,k,l,ava[100],maxi[100][100],allo[100][100],need[100][100],np,n,work[100];
   char p[10][10],tem[10],ar[10][10];
   int fin[100];
   printf("NENTER NO OF PROCESS:");
   scanf("%d",&np);
   printf("ENTER PROCESS NAME:");
   for(i=0;i<np;i++)
   {
      scanf("%s",p[i]);
   }
```

```
printf("\nEnter no.of Resources:");
scanf("%d",&n);
printf("\nEnter Resources name:");
for(i=0;i<n;i++)
{
   scanf("%s",ar[i]);
}
printf("\nEnter Available:");
for(i=0;i<n;i++)
{
   scanf("%d",&ava[i]);
}
printf("\nEnter Allocation:");
for(i=0;i<np;i++)
{
   printf("\nEnter %s :",p[i]);
   for(j=0;j<n;j++)
   {
      scanf("%d",&allo[i][j]);
   }
}
printf("\nEnter Maximum:");
for(i=0;i<np;i++)
{
   printf("\nEnter %s :",p[i]);
   for(j=0;j<n;j++)
   {
      scanf("%d",&maxi[i][j]);
   }
}
for(i=0;i<np;i++)
```

```
{
   for(j=0;j<n;j++)
   {
      need[i][j]=maxi[i][j]-allo[i][j];
   }
}
printf(" ");
for(i=0;i<n;i++)
{
   printf("%s\t",ar[i]);
}
for(i=0;i<np;i++)
{
   printf("\n%s:",p[i]);
   for(j=0;j<n;j++)
   {
      printf("%d\t",need[i][j]);
   }
}
for(i=0;i<n;i++)
{
   work[i]=ava[i];
}
for(i=0;i<n;i++)
{
   fin[i]=0;
}
for(l=0;l<np;l++)
{
   for(i=0;i<np;i++)
   {
```

```
      if(fin[i])
      {
         int f=1;
         for(j=0;j<n;j++)
         {
            if(work[j]<need[i][j])
            f=0;
         }
         if(f)
         {
            fin[i]=1;
            for(k=0;k<n;k++)
            {
               work[k]=allo[i][k];
            }
         }
      }
   }
}

int f1=1;
for(i=0;i<np;i++)
{
   if(fin[i]==0)
   {
      printf("\nNot Safe\n");
      f1=0;
      break;
   }
}
if(f1)
```

```
    printf("\nSystem is Safe\n");
}
```

First Fit b. Worst Fit c. Best Fit

Step 1: Input the number of memory partitions. Step 2: Input the sizes of all partitions. Step 3: Store the original partition sizes in another array. Step 4: Input the number of processes. Step 5: Input the sizes of all processes. Step 6: Reset allocation details and free space. Step 7: Display menu for memory allocation methods — First Fit, Best Fit, and Worst Fit. Step 8: Input the user's choice. Step 9: If choice is First Fit, then Step 9.1: For each process, check partitions from the beginning. Step 9.2: Allocate the process to the first partition large enough to hold it. Step 9.3: Reduce that partition's remaining size. Step 10: If choice is Best Fit, then Step 10.1: For each process, find the partition with the smallest space that can fit it. Step 10.2: Allocate the process to that partition. Step 10.3: Reduce that partition's remaining size. Step 11: If choice is Worst Fit, then Step 11.1: For each process, find the partition with the largest space that can fit it. Step 11.2: Allocate the process to that partition. Step 11.3: Reduce that partition's remaining size. Step 12: If an invalid choice is entered, display "Invalid choice". Step 13: Display the allocation table showing process number, process size, and allocated partition size or "Not Allocated". Step 14: Calculate the total free space left by adding all unallocated memory from partitions. Step 15: Display the total free space. Step 16: Ask the user if they want to try another allocation method. Step 17: If the answer is yes, repeat from Step 6. Step 18: If the answer is no, stop execution.

```
#include <stdio.h>
int main()
{
    int m, n, i, j, choice;
    int process[20], partition[20], originalPartition[20];
    int allocated[20];
    int freeSpace = 0;
    char ans;

    printf("Enter number of memory partitions: ");
    scanf("%d", &m);
```

```
printf("Enter sizes of each partition:\n");
for(i = 0; i < m; i++)
{
   scanf("%d", &partition[i]);
   originalPartition[i] = partition[i];
}

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

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

do
{
   // Reset partition and allocation data for a new simulation
   for(i=0; i<n; i++) allocated[i] = -1;
   for(i=0; i<m; i++) partition[i] = originalPartition[i];
   freeSpace = 0;

   printf("\nChoose Memory Allocation Method:\n");
   printf("1. First Fit\n2. Best Fit\n3. Worst Fit\n");
   printf("Enter choice: ");
   scanf("%d", &choice);

   switch(choice)
   {
      case 1: // First Fit
```

```
    for(i = 0; i < n; i++) {
        for(j = 0; j < m; j++) {
            if(process[i] <= partition[j]) {
                allocated[i] = j;
                partition[j] -= process[i];
                break; // Move to the next process
            }
        }
    }
    break;

case 2: // Best Fit
    for(i = 0; i < n; i++) {
        int bestIndex = -1;
        for(j = 0; j < m; j++) {
            if(process[i] <= partition[j]) {
                if(bestIndex == -1 || partition[j] < partition[bestIndex]) {
                    bestIndex = j;
                }
            }
        }
        if(bestIndex != -1) {
            allocated[i] = bestIndex;
            partition[bestIndex] -= process[i];
        }
    }
    break;

case 3: // Worst Fit
    for(i = 0; i < n; i++) {
        int worstIndex = -1;
```

```
        for(j = 0; j < m; j++) {
            if(process[i] <= partition[j]) {
                if(worstIndex == -1 || partition[j] > partition[worstIndex]) {
                    worstIndex = j;
                }
            }
        }
        if(worstIndex != -1) {
            allocated[i] = worstIndex;
            partition[worstIndex] -= process[i];
        }
    }
    break;

  default:
    printf("Invalid choice!\n");
    continue; // Skip the rest of the loop and restart
}

// Print the results of the allocation
printf("\nProcess\tSize\tAllocated Partition Size\n");
for(i = 0; i < n; i++)
{
  if(allocated[i] != -1)
  {
    // Print the original size of the partition it was placed in
    printf("%d\t%d\t%d\n", i+1, process[i], originalPartition[allocated[i]]);
  }
  else
  {
    printf("%d\t%d\tNot Allocated\n", i+1, process[i]);
```

```
        }
      }

      // Calculate total remaining free space (fragmentation)
      for(i = 0; i < m; i++)
      {
        freeSpace += partition[i];
      }

      printf("\nTotal Free Space Left = %d\n", freeSpace);

      printf("\nDo you want to choose another allocation method? (y/n): ");
      scanf(" %c", &ans); // Note the space before %c to consume the newline

   } while(ans == 'y' || ans == 'Y');

   return 0;
}
```

FIFO b. Optimal c. LRU

Step 1: Input total number of pages. Step 2: Input all page numbers. Step 3: Display menu – FIFO, LRU, OPTIMAL, EXIT. Step 4: Input user choice. Step 5: Input number of frames. Step 6: Initialise all frames to empty and set page fault count = 0. Step 7: For each page in the reference string, check if it is already in the frames. Step 8: If not found, replace a page according to the selected method: - FIFO: Replace the oldest page. - LRU: Replace the least recently used page. - OPTIMAL: Replace the page not used for the longest time. Step 9: Increase page fault count for every replacement. Step 10: Display page number, current frames, and whether a fault occurred. Step 11: After all pages, display total page faults and page fault frequency. Step 12: Ask the user if they want to continue. Step 13: If yes, repeat the process; otherwise, stop.

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

// Global variables
int pages[20], frames[10], n, frameCount, pageFault;

// Function prototypes
void FIFO();
void LRU();
void OPTIMAL();
void initialize();

int main()
{
   printf("Enter total number of pages: ");
   scanf("%d", &n);

   printf("Enter the page numbers (reference string):\n");
   for(int i = 0; i < n; i++)
   {
      scanf("%d", &pages[i]);
   }
```

```
char ans;
do
{
   printf("\n--- Reference String: ");
   for (int i = 0; i < n; i++)
   {
      printf("%d ", pages[i]);
   }
   printf("---\n");

   printf("\nSelect Page Replacement Algorithm:\n");
   printf("1. FIFO (First-In, First-Out)\n");
   printf("2. LRU (Least Recently Used)\n");
   printf("3. OPTIMAL (Optimal Replacement)\n");
   printf("4. EXIT\n");
   printf("Choice: ");

   int choice;
   scanf("%d", &choice);

   switch(choice)
   {
      case 1: FIFO(); break;
      case 2: LRU(); break;
      case 3: OPTIMAL(); break;
      case 4: exit(0);
      default: printf("Invalid Choice!\n"); break;
   }

   printf("\nWant to continue (y/n): ");
   scanf(" %c", &ans); // Space before %c is important
```

```
    } while(ans == 'y' || ans == 'Y');

    return 0;
}

// Resets frames to -1 (empty) and gets the frame count
void initialize()
{
    printf("\nEnter number of frames: ");
    scanf("%d", &frameCount);

    // Check for array bounds
    if(frameCount > 10) {
        printf("Error: Max frames is 10.\n");
        frameCount = 10;
    }

    pageFault = 0;
    for(int i = 0; i < frameCount; i++)
        frames[i] = -1;
}

// Helper function to print the current state of frames
void printFrames(int page, int fault)
{
    printf("%d\t", page);
    for(int k = 0; k < frameCount; k++) {
        if(frames[k] != -1)
            printf("%d ", frames[k]);
        else
```

```
            printf("- "); // Show empty frames
        }
        printf("\t%c\n", (fault ? 'Y' : 'N'));
    }

    // 1. First-In, First-Out
    void FIFO() {
        initialize();
        int front = 0; // Pointer to the oldest frame

        printf("\nFIFO Page Replacement\n");
        printf("Page\tFrames\tFault\n");

        for (int i = 0; i < n; i++)
        {
            int found = 0;
            for (int j = 0; j < frameCount; j++)
            {
                if (frames[j] == pages[i]) {
                    found = 1;
                    printFrames(pages[i], 0); // No fault
                    break;
                }
            }

            if (!found) {
                frames[front] = pages[i];
                front = (front + 1) % frameCount; // Move pointer to next frame
                pageFault++;
                printFrames(pages[i], 1); // Page fault
            }
```

```
    }
    printf("\nTotal Page Faults: %d\n", pageFault);
    printf("Page Fault Frequency: %.3f\n", (float)pageFault / n);
}

// 2. Least Recently Used
void LRU() {
    initialize();
    // 'recent' will store the "timestamp" (index i) of the last use
    int recent[10] = {0};
    int counter = 0; // Tracks usage for timestamp

    printf("\nLRU Page Replacement\n");
    printf("Page\tFrames\tFault\n");

    for (int i = 0; i < n; i++)
    {
        int found = 0;
        // Check if page is already in a frame
        for (int j = 0; j < frameCount; j++)
        {
            if (frames[j] == pages[i]) {
                found = 1;
                recent[j] = ++counter; // Update its timestamp
                printFrames(pages[i], 0); // No fault
                break;
            }
        }

        if (!found) {
            pageFault++;
```

```
        int lru_index = -1;

        // First, check for an empty frame
        for(int j = 0; j < frameCount; j++) {
          if(frames[j] == -1) {
            lru_index = j;
            break;
          }
        }

        // If no empty frame, find the least recently used
        if (lru_index == -1) {
          int min = recent[0];
          lru_index = 0;
          for(int j = 1; j < frameCount; j++) {
            if (recent[j] < min) {
              min = recent[j];
              lru_index = j;
            }
          }
        }

        // Replace the page (either in empty slot or LRU slot)
        frames[lru_index] = pages[i];
        recent[lru_index] = ++counter; // Give it the newest timestamp
        printFrames(pages[i], 1); // Page fault
      }
    }
    printf("\nTotal Page Faults: %d\n", pageFault);
    printf("Page Fault Frequency: %.3f\n", (float)pageFault / n);
}
```

```
// 3. Optimal Page Replacement
void OPTIMAL() {
   initialize();

   printf("\nOPTIMAL Page Replacement\n");
   printf("Page\tFrames\tFault\n");

   for (int i = 0; i < n; i++)
   {
      int found = 0;
      // Check if page is already in a frame
      for (int j = 0; j < frameCount; j++)
      {
         if (frames[j] == pages[i]) {
            found = 1;
            printFrames(pages[i], 0); // No fault
            break;
         }
      }

      if (!found) {
         pageFault++;
         int replaceIndex = -1;

         // First, check for an empty frame
         for(int j = 0; j < frameCount; j++) {
            if(frames[j] == -1) {
               replaceIndex = j;
               break;
            }
```

```
}

// If no empty frame, find the optimal page to replace
if (replaceIndex == -1) {
   int farthest = -1; // Index in the *future*
   replaceIndex = 0;

   // Find which frame holds the page used farthest in the future
   for (int j = 0; j < frameCount; j++) {
      int k;
      // Look ahead in the reference string
      for (k = i + 1; k < n; k++) {
         if (frames[j] == pages[k]) {
            // Found a future use
            if (k > farthest) {
               farthest = k;
               replaceIndex = j;
            }
            break;
         }
      }

      // If page 'k' is never used again (loop finished)
      if (k == n) {
         replaceIndex = j; // This is the best page to replace
         goto replace; // Exit both loops
      }
   }
}

replace: // Label for the goto
```

```
            frames[replaceIndex] = pages[i];
            printFrames(pages[i], 1); // Page fault
        }
    }
    printf("\nTotal Page Faults: %d\n", pageFault);
    printf("Page Fault Frequency: %.3f\n", (float)pageFault / n);
}
```

disk scheduling FCFS: Step 1: Input the current head position. Step 2: Input the number of disk requests. Step 3: Input all the request sequence values. Step 4: Calculate the movement from the current position to the first request. Step 5: Add this movement to the total head movement. Step 6: Print the movement path (current position → first request). Step 7: For each remaining request, calculate the movement between consecutive requests. Step 8: Add each movement to the total head movement. Step 9: Print each movement path (previous request → current request). Step 10: After all requests are processed, display the total head movement.

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

int main()
{
    int i, n, req[50], mov = 0, cp;
    printf("enter the current position\n");
    scanf("%d", &cp);
    printf("enter the number of requests\n");
    scanf("%d", &n);
    printf("enter the request order\n");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &req[i]);
    }
    mov = mov + abs(cp - req[0]);
```

```
    printf("%d -> %d", cp, req[0]);
    for(i = 1; i < n; i++)
    {
        mov = mov + abs(req[i] - req[i-1]);
        printf(" -> %d", req[i]);
    }
    printf("\n");
    printf("total head movement = %d\n", mov);
}
```

CONTINUOUS ALLOCATION:

Step 1: Take input for the number of files. Step 2: For each file, do the following: Step 2.1: Enter the file name. Step 2.2: Enter the number of blocks occupied by the file. Step 2.3: Enter the starting block of the file. Step 2.4: Store the starting block and total blocks. Step 2.5: Allocate the blocks one by one for the file. Step 3: Display the file details such as filename, start block, and length. Step 4: Display all the block numbers occupied by each file.

```
#include<stdio.h>
struct
{
    char fname[50];
} f[50];

int main()
{
    int n, i, j, b[50], sb[50], t[50], c[50][50];
    printf("Enter no. of files: ");
    scanf("%d", &n);

    for(i = 0; i < n; i++)
    {
        printf("Enter file %d name: ", i + 1);
```

```
        scanf("%s", f[i].fname);
        printf("Enter no. of blocks occupied by file %s: ", f[i].fname);
        scanf("%d", &b[i]);
        printf("Enter the starting block of file %s: ", f[i].fname);
        scanf("%d", &sb[i]);
        t[i] = sb[i];
        for(j = 0; j < b[i]; j++)
            c[i][j] = sb[i]++;
    }

    printf("\nFilename\tStart block\tLength\n");
    for(i = 0; i < n; i++)
        printf("%s\t\t%d\t\t%d\n", f[i].fname, t[i], b[i]);

    printf("\nBlocks occupied:\n");
    for(i = 0; i < n; i++)
    {
        printf("File name %s: ", f[i].fname);
        for(j = 0; j < b[i]; j++)
            printf("%d ", c[i][j]);
        printf("\n");
    }

    return 0;
}
```