memory management scheme

```c
#include <stdio.h>
void firstFit(int mem[], int n, int process[], int m, int allocation[]) {
for(int i = 0; i < m; i++) {
for(int j = 0; j < n; j++) {
if(mem[j] >= process[i]) {
allocation[i] = j;
mem[j] -= process[i];
break;}}}}
void bestFit(int mem[], int n, int process[], int m, int allocation[]) {
for(int i = 0; i < m; i++) {
int bestIdx = -1;
for(int j = 0; j < n; j++) {
if(mem[j] >= process[i]) {
if(bestIdx == -1 || mem[j] < mem[bestIdx]) {
bestIdx = j;}}}
if(bestIdx != -1) {
allocation[i] = bestIdx;
mem[bestIdx] -= process[i];}}}
void worstFit(int mem[], int n, int process[], int m, int allocation[]) {
for(int i = 0; i < m; i++) {
int worstIdx = -1;
for(int j = 0; j < n; j++) {
if(mem[j] >= process[i]) {
if(worstIdx == -1 || mem[j] > mem[worstIdx]) {
worstIdx = j;}}}
if(worstIdx != -1) {
allocation[i] = worstIdx;
mem[worstIdx] -= process[i];}}}
int main() {
int n, m, choice;
int mem[10], process[10], temp_mem[10], allocation[10];
printf("Enter number of memory blocks: ");
scanf("%d", &n);
printf("Enter size of each block:\n");
for(int i = 0; i < n; i++) {
scanf("%d", &mem[i]);}
printf("Enter number of processes: ");
scanf("%d", &m);
printf("Enter size of each process:\n");
for(int i = 0; i < m; i++) {
scanf("%d", &process[i]);
allocation[i] = -1;}
printf("\nMemory Allocation Algorithms:\n");
printf("1. First Fit\n2. Best Fit\n3. Worst Fit\n");
printf("Enter choice: ");
scanf("%d", &choice);
for(int i = 0; i < n; i++) {
temp_mem[i] = mem[i];}
switch(choice) {
case 1:
firstFit(temp_mem, n, process, m, allocation);
break;
case 2:
bestFit(temp_mem, n, process, m, allocation);
break;
case 3:
worstFit(temp_mem, n, process, m, allocation);
break;
default:
printf("Invalid choice!\n");
return 1;}
printf("\nProcess\tSize\tBlock Allocated\n");
for(int i = 0; i < m; i++) {
printf("%d\t%d\t", i+1, process[i]);
if(allocation[i] != -1) {
printf("%d\n", allocation[i] + 1);
} else {
printf("Not Allocated\n");}}
printf("\nRemaining Memory Blocks:\n");
for(int i = 0; i < n; i++) {
printf("Block %d: %d\n", i+1, temp_mem[i]);
}
return 0;
}
```

page replacement

```
#include <stdio.h>
#include <stdbool.h>
// Function prototypes
int fifo(int pages[], int n, int frames);
int lru(int pages[], int n, int frames);
int optimal(int pages[], int n, int frames);
int main() {
int choice, frames, n;
printf("Enter number of frames: ");
scanf("%d", &frames);
printf("Enter number of page references: ");
scanf("%d", &n);
int pages[n];
printf("Enter page reference string: ");
for(int i = 0; i < n; i++) {
scanf("%d", &pages[i]);}
printf("\nPage Replacement Algorithms:\n");
printf("1. FIFO\n2. LRU\n3. Optimal\n");
printf("Enter your choice: ");
scanf("%d", &choice);
int faults = 0;
switch(choice) {
case 1:
faults = fifo(pages, n, frames);
break;
case 2:
faults = lru(pages, n, frames);
break;
case 3:
faults = optimal(pages, n, frames);
break;
default:
printf("Invalid choice!\n");
return 1;}
printf("\nTotal Page Faults: %d\n", faults);
return 0;}
// FIFO Implementation
int fifo(int pages[], int n, int frames) {
int mem[frames], queue[frames];
int front = 0, faults = 0;
for(int i = 0; i < frames; i++) mem[i] = -1;
for(int i = 0; i < n; i++) {
bool found = false;
for(int j = 0; j < frames; j++) {
if(mem[j] == pages[i]) {
found = true;
break;}}
if(!found) {
mem[front] = pages[i];
front = (front + 1) % frames;
faults++;}}
return faults;}
// LRU Implementation
int lru(int pages[], int n, int frames) {
int mem[frames], counter[frames];
int faults = 0, time = 0;
for(int i = 0; i < frames; i++) {
mem[i] = -1;
counter[i] = 0;}
for(int i = 0; i < n; i++) {
bool found = false;
for(int j = 0; j < frames; j++) {
if(mem[j] == pages[i]) {
found = true;
counter[j] = ++time;
break;}}
if(!found) {
int lru = 0;
for(int j = 1; j < frames; j++) {
if(counter[j] < counter[lru]) lru = j;}
mem[lru] = pages[i];
counter[lru] = ++time;
faults++;}}
return faults;}
// Optimal Implementation
int optimal(int pages[], int n, int frames) {
int mem[frames], faults = 0;
for(int i = 0; i < frames; i++) mem[i] = -1;
for(int i = 0; i < n; i++) {
bool found = false;
for(int j = 0; j < frames; j++) {
if(mem[j] == pages[i]) {
found = true;
break;}}
if(!found) {
faults++;
int replace = -1, farthest = i;
for(int j = 0; j < frames; j++) {
int k;
for(k = i + 1; k < n; k++) {
if(mem[j] == pages[k]) break;}
if(k > farthest) {
farthest = k;
replace = j;}}
if(replace == -1) replace = 0;
mem[replace] = pages[i];}}
return faults;
}
```

PROGRAM-9
PAGE REPLACEMENT ALGORITHM

AIM

To write a C program for implementation of FIFO, LRU, and optimal page replacement algorithm using switch.

ALGORITHM

1. Start the program
2. Declare the necessary variable
3. Enter the number of frames
4. Enter the reference string, ending with zero.
5. Display the menu.
6. Read user input into choices as 1, 2, 3, 4
7. If the user choice = 1, perform the following steps for FIFO page replacement.

7.1: The page that has been in memory the longest time is selected.

7.2: When a page must be replaced, the oldest page is chosen.

7.3: When a page is brought into memory, it is inserted at the tail of the queue.

7.4: Initially, all frames are empty.

7.5: The page fault rate increases as the number of allocated frames increases.

7.6: Print the total number of page fault.

8: If the user choice = 2 perform the following steps for LRU page replacement

8.1: Declare the size.

8.2: Get the number of pages to be inserted
8.3: Get the values.

8.4: Declare counter and stack.

8.5: Select the least recently used page by value.

8.6: Stack them according to the selection.
8.7: Display the values.

9: If the user choice = 3, perform the following steps for optimal page replacement

9.1: Declare the size.

9.2: Get the number of pages to be inserted.

9.3: Get the values.

9.4: Check future page references to determine which page will not be needed for longest time

9.5: Select that page for replacement.

9.6: Replace the selected page with new page.

9.7: Display the values.

10: If the user choice = 4, exit the program.

11: Stop the program

PROGRAM- 8
MEMORY ALLOCATION METHODS FOR FIXED PARTITION

Aim

To write a program to implement memory management scheme First fit, Best fit and Worst fit.

ALGORITHM

Step1: Start

Step2: Read numbers of memory blocks, n

Step3: Read sizes of memory blocks

Step4: Read numbers of processes, m

Step5: Read sizes of processes.

Step6: For each process i from 1 to m, set allocation[i] = -1

Step7: Display choices

1 → First fit
2 → Best fit
3 → Worst fit

Step8: Read choice.

Step9: For each block i from 1 to n, copy mem[i] to temp mem[i]

Step 10: if choice == 1 (First fit)
for each process i from 1 to m:
for each block j from 1 to n
if temp mem[j] is large enough for process [i]
Assign allocation [i] = j
Reduce temp mem[j] by process [i]
Break inner loop

Step 11: else if choice == 2 (Best fit)
for each process i from 1 to m
Set best Idx = -1
for each block j from 1 to n:
if temp mem[j] fits process [i] and is smaller than the current best Idx block:
Set best Idx = j
if best Idx is found, assign allocation [i] = best Idx
Reduce temp mem [best Idx] by process [i]

Step 12: else if choice == 3 (worst fit)
For each process i from 1 to m:
Set worst Idx = -1
for each block j from 1 to n:
if temp_mem [j] fits process [i] and is larger than the current worst Idx block, Set worst Idx = j
if worst Idx is found, assign allocation [i] = worst Idx
Reduce temp_mem [worst Idx] by process [i].

Step 13: else, print "Invalid Choice!" and exit.

Step 14: Print Process, Size, Block Allocated for each process i from 1 to m:
- if allocation[i] is assigned, print the Block number
- else, print "Not Allocated".

Step 15: Print updated sizes of all memory blocks temp_mem[1] to temp_mem[n].

Step 16: Stop

Algorithm for FCFS and SCAN Disk Scheduling using Switch Case
 	
1. Start
2. Declare variables: n, head, disk_size, choice, and an integer array requests[]
3. Input the number of disk requests
4. Create an array of size n to store the disk request sequence
5. Input the request sequence from the user
   5.1 Store each request into the array
6. Input the initial position of the disk head
7. Input the total size of the disk
8. Display algorithm options to the user
   8.1 Option 1 → FCFS
   8.2 Option 2 → SCAN
9. Take user's choice as input
10. Use switch-case to perform the selected scheduling
   10.1 If choice = 1, call FCFS function
   10.2 If choice = 2, call SCAN function
   10.3 If choice is invalid, print error message
11. In FCFS function:
   11.1 Declare and initialize seek_time = 0
   11.2 Print initial head position
   11.3 For each request in order:
      11.3.1 Calculate absolute difference between head and current request
      11.3.2 Add difference to seek_time
      11.3.3 Update head to current request
   11.4 Print total seek time and movement sequence
12. In SCAN function:
   12.1 Create a new array sorted_requests[] of size n + 2
   12.2 Add 0 and disk_size - 1 as boundary values to the array
   12.3 Copy all disk requests into the array
   12.4 Sort the array in ascending
	order
	   12.5 Find the position where head fits in the sorted array
	   12.6 Move head from current position to the highest track
	      12.6.1 For each movement, update seek_time and head
	   12.7 After reaching the end, move head back toward 0
	      12.7.1 For each movement, update seek_time and head
	   12.8 Print total seek time and complete movement path
	13. Stop

Disk Scheduling C program,

```c
#include <stdio.h>
#include <stdlib.h>
void fcfs(int requests[], int n, int head);
void scan(int requests[], int n, int head, int
disk_size);
 int main() {
int n, head, disk_size, choice;
printf("Enter the number of requests: ");
scanf("%d", &n);
int requests[n];
printf("Enter the request sequence: ");
for (int i = 0; i < n; i++) {
scanf("%d", &requests[i]);}
printf("Enter the initial position of the disk head:");
scanf("%d", &head);
printf("Enter the total size of the disk: ");
scanf("%d", &disk_size);
printf("\nDisk Scheduling Algorithms:\n");
printf("1. FCFS\n2. SCAN\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
case 1:
fcfs(requests, n, head);
break;
case 2:
scan(requests, n, head, disk_size);
break;
default:
printf("Invalid choice!\n");}
return 0;}
void fcfs(int requests[], int n, int head) {
int seek_time = 0;
printf("\nFCFS Disk Scheduling\n");
printf("Sequence of movement: %d", head);
for (int i = 0; i < n; i++) {
seek_time += abs(requests[i] - head);
head = requests[i];
printf(" -> %d", head);}
printf("\nTotal Seek Time: %d\n", seek_time);}
void scan(int requests[], int n, int head, int
disk_size) {
int seek_time = 0;
int sorted_requests[n + 2];
sorted_requests[0] = 0;
sorted_requests[n + 1] = disk_size - 1;
for (int i = 0; i < n; i++) {
sorted_requests[i + 1] = requests[i];}
for (int i = 0; i < n + 2; i++) {
for (int j = i + 1; j < n + 2; j++) {
if (sorted_requests[i] > sorted_requests[j]) {
int temp = sorted_requests[i];
sorted_requests[i] = sorted_requests[j];
sorted_requests[j] = temp;}}}
int pos;
for (pos = 0; pos < n + 2; pos++) {
if (sorted_requests[pos] >= head) {
break;}}
printf("\nSCAN Disk Scheduling\n");
printf("Sequence of movement: %d", head);
for (int i = pos; i < n + 2; i++) {
seek_time += abs(sorted_requests[i] - head);
head = sorted_requests[i];
printf(" -> %d", head);}
for (int i = pos - 1; i >= 0; i--) {
seek_time += abs(sorted_requests[i] - head);
head = sorted_requests[i];
printf(" -> %d", head);}
printf("\nTotal Seek Time: %d\n", seek_time);}
```

Algorithm for FCFS Disk Scheduling

1. Start the program.
2. Declare variables:
   - n for total number of disk requests.
   - head for initial disk head position.
   - requests[n] to store the disk request queue.
3. Read input values from the user:
   3.1 Ask and read the value of n.
   3.2 Declare an integer array requests[n].
   3.3 Ask the user to enter n disk requests.
   3.4 Use a loop from i = 0 to i < n to read and store each request into requests[i].
   3.5 Ask and read the initial value of head.
4. Call the FCFS function by passing requests, n, and head as arguments.
5. Inside the FCFS function:
   5.1 Declare seek_time = 0 to hold the total seek time.
   5.2 Declare prev = head to remember the previous head position.
   5.3 Print the message for FCFS Order.
   5.4 Use a loop from i = 0 to i < n:
    - 5.4.1 Print the current request requests[i].
      - 5.4.2 Find the absolute difference between requests[i] and prev, and add it to seek_time.
      - 5.4.3 Update prev to requests[i].
 5.5 After the loop, print "End" to complete the order.
   5.6 Print the total seek time value.
6. End the program.

FCFS CODE

```
#include <stdio.h>
#include <stdlib.h>
void fcfs(int requests[], int n, int head) {
int seek_time = 0, prev = head;
printf("\nFCFS Order: ");
for (int i = 0; i < n; i++) {
printf("%d -> ", requests[i]);
seek_time += abs(requests[i] - prev);
prev = requests[i];}
printf("End\nTotal Seek Time: %d\n", seek_time);}
int main() {
int n, head;
printf("Enter total number of disk requests: ");
scanf("%d", &n);
int requests[n];
printf("Enter disk requests: ");
for (int i = 0; i < n; i++)
scanf("%d", &requests[i]);
printf("Enter initial head position: ");
scanf("%d", &head);
fcfs(requests, n, head);
return 0;}
```

Algorithm for Round Robin Scheduling

1. Start the program execution.
2. Declare structure process with:
   - pname[4] to store the name of the process.
   - bt for burst time.
   - wt for waiting time.
   - tt for turnaround time.
   - rjob for remaining job time.
3. Declare integer variables:
   - ts for time slice.
   - a as a cumulative time tracker.
   - n for number of processes.
   - i for loop control.
   - avgwt and avgtt for total waiting and turnaround time.
   - c for counting completed processes.
4. Ask the user to enter number of processes n and read it.
5. Create an array p[n] of type process.
6. Loop through all n processes:
   6.1 Ask the user to enter the process name and read it into p[i].pname.
6.2 Ask the user to enter the burst time and read it into p[i].bt.
   6.3 Initialize p[i].rjob with the burst time (p[i].bt).
7. Ask the user to enter the time slice ts and read it.
8. Display all process names and their burst times in a table format.
9. Print the Gantt chart label.
10. Use a loop to simulate the Round Robin scheduling:
10.1 Repeat the scheduling until c (completed process count) becomes equal to n.
   10.2 For each process in the loop:
      10.2.1 If p[i].rjob <= ts and p[i].rjob > 0:
         - Add p[i].rjob to a.
         - Set p[i].rjob = 0.
         - Increment c by 1.
         - Set p[i].tt = a.
         - Compute waiting time: p[i].wt = p[i].tt - p[i].bt.
         - Print Gantt chart movement: p[i].pname --> (a).
      10.2.2 Else if p[i].rjob > ts:
         - Add ts to a.
         - Decrease p[i].rjob by ts.
         - Print Gantt chart movement: p[i].pname --> (a).
11. After the loop ends, print the final table with:
   - Process name
   - Burst time
   - Waiting time
   - Turnaround time
12. Loop through each process:
   12.1 Add p[i].wt to avgwt.
   12.2 Add p[i].tt to avgtt.
13. Compute and print average waiting time: avgwt / n.
14. Compute and print average turnaround time: avgtt / n.
15. End the program.

CODE FOR ROUnd robin

```
#include<stdio.h>
struct process
{
char pname[4];
int bt,wt,tt,rjob;
}p[10];
void main()
{
int ts,a=0,n,i,avgwt=0,avgtt=0,c=0;
printf("How many processes do you need? : ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter the Process name : ");
scanf("%s",p[i].pname);
printf("Enter the Burst time : ");
scanf("%d",&p[i].bt);
p[i].rjob=p[i].bt;
}
printf("\nEnter the time slice:");
scanf("%d",&ts);
printf("\nPname\tBT\t\n");
for(i=0;i<n;i++){
printf("%s\t%d\t\n",p[i].pname,p[i].bt);
}
printf("\nGantt chart:\n");
do{
for(i=0;i<n;i++){
if((p[i].rjob<=ts) &&(p[i].rjob>0)){
a=a+p[i].rjob;
p[i].rjob=0;
c++;
p[i].tt=a;
p[i].wt=p[i].tt-p[i].bt;
printf("%s-->(%d)",p[i].pname,a);
}
else if(p[i].rjob>ts){
a=a+ts;
p[i].rjob=p[i].rjob-ts;
printf("%s-->(%d)",p[i].pname,a);
}
}
}while(c<n);
printf("\n\nPname\tBT\tWT\tTT\n");
for(i=0;i<n;i++){
printf("%s\t%d\t%d\t%d\n",p[i].pname,p[i].bt,p[i].w
t,p[i].tt);
}
for(i=0;i<n;i++)
{
avgwt=avgwt+p[i].wt;
avgtt=avgtt+p[i].tt;
}
printf("\n\nAverage Waiting Time =
%f",(float)avgwt/n);
printf("\nAverage Turnaround Time =
%f\n",(float)avgtt/n);
}
```

priority non preemptive

Algorithm: Priority Scheduling (Non-preemptive)

1. Start the program.
2. Declare integer variable `n` to store the number of processes.
3. Prompt the user to enter `n`.
4. Define a structure `Process` with the following members:
4.1. `id` – process identifier
4.2. `burstTime` – burst time of the process
4.3. `priority` – priority of the process (lower number = higher priority)
4.4. `arrivaltime` – arrival time of the process
4.5. `completiontime` – time when the process completes
4.6. `turnAroundTime` – turnaround time = completion time − arrival time
4.7. `waitTime` – waiting time = turnaround time − burst time
4.8. `completed` – flag to mark if the process is finished (1 = completed, 0 = not completed)
5. Declare an array `p[n]` of type `struct Process` to store all processes.
6. Loop `i` from 0 to `n-1`:
6.1. Set `p[i].id = i + 1`.
6.2. Set `p[i].completed = 0`.
6.3. Prompt the user to enter values: `burstTime`, `priority`, and `arrivaltime` for `p[i]`.
6.4. Store them into `p[i].burstTime`, `p[i].priority`, and `p[i].arrivaltime` respectively.
7. Call the function `priorityScheduling(p, n)` to compute scheduling.
8. Inside `priorityScheduling(p, n)`, do the following:
8.1. Declare integer `completedProcesses = 0` to count completed processes.
8.2. Declare integer `currentTime = 0` to represent system time.
8.3. Declare float variables `totalWaitTime = 0` and `totalTurnAroundTime = 0` to calculate averages later.
8.4. While `completedProcesses < n`, repeat steps:
8.4.1. Declare integer `highindex = -1` to track index of the next process to execute.
8.4.2. Declare integer `highestpriority = 999` (assumed to be very low priority).
8.4.3. Loop `i` from 0 to `n-1`:
8.4.3.1. If `p[i].completed == 0` AND `p[i].arrivaltime <= currentTime`:
8.4.3.1.1. If `p[i].priority < highestpriority`:
8.4.3.1.1.1. Set `highindex = i`.
8.4.3.1.1.2. Set `highestpriority = p[i].priority`.
8.4.4. After the loop, check if `highindex != -1`:
8.4.4.1. Add `p[highindex].burstTime` to `currentTime`.
8.4.4.2. Set `p[highindex].completiontime = currentTime`.
8.4.4.3. Calculate `p[highindex].turnAroundTime = completiontime − arrivaltime`.
8.4.4.4. Calculate `p[highindex].waitTime = turnaroundTime − burstTime`.
8.4.4.5. Add `p[highindex].waitTime` to `totalWaitTime`.
8.4.4.6. Add `p[highindex].turnAroundTime` to `totalTurnAroundTime`.
8.4.4.7. Mark `p[highindex].completed = 1`.
8.4.4.8. Increment `completedProcesses` by 1.
8.4.5. Else (no process arrived yet), increment `currentTime` by 1.
9. After all processes are completed, display results:
9.1. Print header for output: Process ID, Burst Time, Priority, Waiting Time, Turnaround Time, Completion Time.
9.2. Loop `i` from 0 to `n-1`, print all values of `p[i]`.
10. Calculate and print average waiting time = `totalWaitTime / n`.
11. Calculate and print average turnaround time = `totalTurnAroundTime / n`.
12. End the program.

non preemptive (priority)

```c
#include <stdio.h>
struct Process {
int id;
int burstTime;
int completiontime;
int priority;
int waitTime;
int arrivaltime;
int turnAroundTime;
int completed;};
void priorityScheduling(struct Process p[], int n) {
int completedProcesses = 0, currentTime = 0;
float totalWaitTime = 0, totalTurnAroundTime = 0;
printf("\nProcess Execution Order:\n");
while (completedProcesses < n) {
int highindex = -1;
int highestpriority = 999;
for (int i = 0; i < n; i++) {
if (!p[i].completed && p[i].arrivaltime <=
currentTime) {
if (p[i].priority < highestpriority) {
highindex = i;
highestpriority = p[i].priority;}}}
if (highindex != -1) {
currentTime += p[highindex].burstTime;
p[highindex].completiontime = currentTime;
p[highindex].turnAroundTime =
p[highindex].completiontime -
p[highindex].arrivaltime;
p[highindex].waitTime =
p[highindex].turnAroundTime -
p[highindex].burstTime;
totalTurnAroundTime +=
p[highindex].turnAroundTime;
totalWaitTime += p[highindex].waitTime;
p[highindex].completed = 1;
completedProcesses++;
} else {
currentTime++;}}
printf("\n\nProcess\tBurst Time\tPriority\tWaiting
Time\tTurnaround Time\tCompletion Time\n");
for (int i = 0; i < n; i++) {
printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
p[i].id, p[i].burstTime, p[i].priority, p[i].waitTime,
p[i].turnAroundTime, p[i].completiontime);}
printf("\nAverage Waiting Time: %.2f",
totalWaitTime / n);
printf("\nAverage Turnaround Time: %.2f\n",
totalTurnAroundTime / n);}
```

```c
int main() {
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);
struct Process p[n];
for (int i = 0; i < n; i++) {
p[i].id = i + 1;
p[i].completed = 0;
printf("Enter Burst Time, Priority, and Arrival Time
for Process P%d: ", p[i].id);
scanf("%d %d %d", &p[i].burstTime, &p[i].priority,
&p[i].arrivaltime);}
priorityScheduling(p, n);
return 0;}
```

CODE OF SJF	Algorithm: Shortest Job First (SJF)

CODE OF SJF

```
#include<stdio.h>
#include<stdlib.h>
struct sjf{
   int pid;
   int bttime;
   int tttime;
   int wttime;
}p[10];
int main()
{ int i,n;
int totalwtime=0,totaltime=0;
printf("Enter the number of processes: ");
scanf("%d", &n);
for(i=0;i<n;i++)
{p[i].pid = i+1;
printf("enter the burst time of the process
%d:",p[i].pid);
scanf("%d",&p[i].bttime);}
struct sjf temp;
for(int i = 0; i < n - 1; i++) {
for(int j = 0; j < n - i - 1; j++) {
if(p[j].bttime > p[j + 1].bttime) {
temp = p[j];
p[j] = p[j + 1];
p[j + 1] = temp;
}}}
p[0].wttime = 0;
p[0].tttime = p[0].bttime;
totaltime += p[0].tttime;
for(i = 1; i < n; i++) {
 p[i].wttime = p[i-1].wttime + p[i-1].bttime;
 p[i].tttime = p[i].wttime + p[i].bttime;
 totalwtime = totalwtime+p[i].wttime;
 totaltime =totaltime + p[i].tttime;   }

printf("\nPID\tBurstTime\tWaitingTime\tTurnaroun
dTime\n");
 for(i = 0; i < n; i++) {
  printf("\n%d\t%d\t\t\t%d\t%d\t\n", p[i].pid,
p[i].bttime, p[i].wttime, p[i].tttime);}
printf("\nTotal Waiting Time: %d", totalwtime);
   printf("\nAverage Waiting Time: %.2f",
(float)totalwtime/n);
   printf("\nTotal Turnaround Time: %d",
totaltime);
   printf("\nAverage Turnaround Time: %.2f",
(float)totaltime/n);
   return 0;}
```

Algorithm: Shortest Job First (SJF)

1. Define a structure for each process with:
   - pid (process ID), bttime (burst time)
   - wttime (waiting time), tttime (turnaround time)

2. Take input for number of processes (n).
   For each process, assign pid and input burst time.

3. Sort the processes based on burst time in ascending order using bubble sort.

4. Set the waiting time of first process to 0.
   Its turnaround time = burst time.

5. For all remaining processes (i = 1 to n-1):
   - waiting time = waiting time of previous + previous burst
   - turnaround time = waiting time + current burst
   - add to total waiting and total turnaround

6. Display PID, Burst Time, Waiting Time and Turnaround Time for all.

7. Finally, calculate and print average waiting time and average turnaround time.

End.

prority code

```
#include<stdio.h>
#include<stdlib.h>
typedef struct
{int pno;
int pri;
int btime;
int wtime;}
sp;
int main(){
int i,j,n;
int tbm=0,totwtime=0,totttime=0;
sp *p,t;
printf("\n PRIORITY SCHEDULING \n");
printf("\n Enter the no of process\n");
scanf("%d",&n);
p=(sp*)malloc(sizeof(sp));
printf("enter the burst time and priority\n");
for(i=0;i<n;i++){
printf("process%d",i+1);
scanf("%d%d",&p[i].btime,&p[i].pri);
p[i].pno=i+1;
p[i].wtime=0;
}
for(i=0;i<n-1;i++)
for(j=j+1;j<n;j++){
if(p[i].pri>p[j].pri){
t=p[i];
p[i]=p[j];
p[j]=t;
}}
printf("\n process\tpriority\tbursttime\ttwaiting
time\tturnaround time\n");
for(i=0;i<n;i++)
{
totwtime+=p[i].wtime=tbm;
tbm+=p[i].btime;
printf("\n%d\t%d\t%d",p[i].pno,p[i].pri,p[i].btime);
printf("\t\t%d\t\t%d",p[i].wtime,p[i].wtime+p[i].btime
);
}
totttime=tbm+totwtime;
printf("\n total waiting time:%d",totwtime);
printf("\naverage waiting
time:%f",(float)totwtime/n);
printf("\n total turnaround time:%d",totttime);
printf("\n avg turnaround
time:%f",(float)totttime/n);}
```

algorithm for prority

Algorithm: Priority Scheduling (Non-Preemptive)

1. Define a structure for each process with:
   - pno (process number)
   - pri (priority)
   - btime (burst time)
   - wtime (waiting time)

2. Take input: number of processes (n).

3. For each process:
   - Input burst time and priority
   - Assign process number
   - Initialize waiting time to 0

4. Sort the processes in ascending order of priority using bubble sort.

5. Initialize tbm (total burst time passed) to 0.

6. For each process (in sorted order):
   - Set waiting time = tbm
   - Update tbm by adding current process's burst time
   - Add waiting time to total waiting time
   - Calculate turnaround time = waiting time + burst time

7. Print details: Process No, Priority, Burst Time, Waiting Time, Turnaround Time.

8. Compute total turnaround time = total waiting time + total burst time

9. Print:
   - Total and average waiting time
   - Total and average turnaround time

End.

code of producer

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1, full = 0, empty = 3, x = 0;
void main()
{
	int n;
	void producer();
	void consumer();
	int wait(int);
	int signal (int);
	printf("\n 1. PRODUCER\n 2.
CONSUMER\n 3. EXIT\n");
	while(1)
	{
		printf("\nENTERT YOUR CHOICE:
");
		scanf("%d", &n);
		switch (n)
		{
			case 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:
			exit(0);
			break;
		}
	}
}

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("Producer Produced %d items", x);
	mutex = signal(mutex);
}
void consumer()
{
	mutex = wait(mutex);
	full = wait(full);
	empty = signal(empty);
	x--;
	printf("Consumer consumed, remaining:
%d", x);
	mutex = signal(mutex);
}
```

algorithm for producer

1. Initialize:
   - mutex = 1 (for mutual exclusion)
   - full = 0 (no items produced yet)
   - empty = 3 (buffer size = 3)
   - x = 0 (count of items in buffer)

2. Show menu:
   1. Producer
   2. Consumer
   3. Exit

3. Loop infinitely:
   - Ask user for choice (n)

4. If n == 1 (Producer):
   - Check if mutex == 1 and empty != 0
     - If yes, call producer()
     - Else, print "BUFFER IS FULL"

5. If n == 2 (Consumer):
   - Check if mutex == 1 and full != 0
     - If yes, call consumer()
     - Else, print "BUFFER IS EMPTY"

6. If n == 3, exit the program.

7. Function wait(s):
   - Decrease semaphore value → return s - 1

8. Function signal(s):
   - Increase semaphore value → return s + 1

9. Function producer():
   - wait(mutex) → enter critical section
   - signal(full), wait(empty)
   - Increment x (produce an item)
   - Print "Producer produced x items"
   - signal(mutex) → exit critical section

10. Function consumer():
    - wait(mutex) → enter critical section
    - wait(full), signal(empty)
    - Decrement x (consume an item)
    - Print "Consumer consumed, remaining: x"
    - signal(mutex) → exit critical section

End.

BANK CODE(s)

```
#include<stdio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n,r,cl;
void input();
void show();
void cal();
int main(){
int i,j;
printf("********** Banker's Algo ************\n");
input();
show();
cal();
return 0;}
void input(){
int i,j;
printf("Enter the no of Processes\t");
scanf("%d",&n);
printf("Enter the no of resources instances\t");
scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++) {
for(j=0;j<r;j++) {
scanf("%d",&max[i][j]);}}
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++) {
for(j=0;j<r;j++) {
scanf("%d",&alloc[i][j]);}}
printf("Enter the available Resources\n");
for(j=0;j<r;j++) {
scanf("%d",&avail[j]);}}
void show() {
int i,j;
printf("Process\t Allocation\t Max\t Available\t");
for(i=0;i<n;i++) {
printf("\nP%d\t ",i+1);
for(j=0;j<r;j++) {
printf("%d ",alloc[i][j]); }
printf("\t");
for(j=0;j<r;j++) {
printf("%d ",max[i][j]); }
printf("\t");
if(i==0) {
for(j=0;j<r;j++)
printf("%d ",avail[j]);
}}}
void cal()
{int
finish[100],temp,need[100][100],flag=1,k,c1=0;
int safe[100];
int i,j;
for(i=0;i<n;i++) {
finish[i]=0; }
for(i=0;i<n;i++) {
for(j=0;j<r;j++) {
need[i][j]=max[i][j]-alloc[i][j];
}}
printf("\n");
while(flag) {
flag=0;
for(i=0;i<n;i++) {
int c=0;
for(j=0;j<r;j++) {
if((finish[i]==0)&&(need[i][j]<=avail[j])) {
c++;
if(c==r) {
for(k=0;k<r;k++) {
avail[k]+=alloc[i][j];
finish[i]=1;
flag=1; }
printf("P%d->",i);
if(finish[i]==1) {
i=n;
}}}}}}
for(i=0;i<n;i++) {
if(finish[i]==1) {
cl++;}
else{
printf("P%d->",i);
}}if(cl==n){
printf("\n The system is in safe state");
}else{
printf("\n Process are in dead lock");
printf("\n System is in unsafe state");
}}
```

BANKER(F)

```c
#include<stdio.h>
void main()
{
	int n, m, i, j, k, y, flag, index = 0, safesequence[100], finish[100], alloc[100][100], max[100][100], need[100][100], avail[100], work[100];
	printf("Enter the number of processes: ");
	scanf("%d", &n);
	printf("Enter the number of resources: ");
	scanf("%d", &m);
	printf("Enter the allocation matrix; ");
	for (i = 0; i < n; i++)
	{
		for (j = 0; j < m; j++)
		{
			scanf("%d", &alloc[i][j]);
		}
	}
	printf("\n");
	printf("Enter the max matrix: ");
	for (i = 0; i < n; i++)
	{
		for (j = 0; j < m; j++)
		{
			scanf("%d", &max[i][j]);
		}
	}
	printf("\n");
	printf("Enter the avail matrix: ");
	for (i = 0; i < m; i++)
	{

		scanf("%d", &avail[i]);
	}
	printf("\n");
	for (i = 0; i < n; i++)
	{
		for (j = 0; j < m; j++)
		{
			need[i][j] = max[i][j] - alloc[i][j];
		}
	}
	printf("The need matrix is : \n");
	for (i = 0; i < n; i++)
	{
		for (j = 0; j < m; j++)
		{
			printf("%d\t", need[i][j]);
		}
		printf("\n");
	}
	for (i = 0; i < n ; i++)
		work[i] = avail[i];
	for (i = 0; i < n ; i++)
		finish[i] = 0;
	for (k = 0; k < n; k++)
	{
		for (i = 0; i < n; i++)
		{
			if (finish[i] == 0)
			{
				flag = 0;
				for (j = 0; j < m; j ++)
				{
					if (need [i][j] > work[j])

					{
						flag = 1;
						break;
					}
				}
				if (flag == 0)
				{
					safesequence[index] = i;
					index++;
					for (y = 0; y < m; y++)

					{
						work[y] = alloc[i][y] + work[y];
					}
					finish[i] = 1;
				}
			}
		}
	}
	printf("\n following is the safesequence: ");
	for (i = 0; i < n; i++)
	{
		printf("\tp%d",safesequence[i]);
	}
	printf("\n");
}
```

BANK ALGORITHM
Algorithm: Banker's Algorithm for Safe State Detection**

1. Start
2. Declare the following:
   - Integer arrays: `finish[n]`, `need[n][r]`, `safe[n]`, `alloc[n][r]`, `max[n][r]`, `avail[r]`
   - Integers: `flag = 1`, `cl = 0`, `i`, `j`, `k`, `c`
3. Initialize all `finish[i] = 0` for i = 0 to n-1
4. Calculate need matrix:
   4.1. For i = 0 to n-1
   4.2. For j = 0 to r-1
   4.3. `need[i][j] = max[i][j] - alloc[i][j]`
5. Repeat while `flag == 1`
   5.1. Set `flag = 0`
   5.2. For i = 0 to n-1
   5.3. If `finish[i] == 0` then
      5.3.1. Set `c = 0`
      5.3.2. For j = 0 to r-1
         If `need[i][j] <= avail[j]` then increment `c`
      5.3.3. If `c == r` then
         For k = 0 to r-1
            `avail[k] += alloc[i][k]`
         Set `finish[i] = 1`, `flag = 1`
         Print `P[i] ->`
6. After loop, check system status:
   6.1. For i = 0 to n-1
   6.2. If `finish[i] == 1` then increment `cl`, else print `P[i] ->`
7. If `cl == n` then print “System is in safe state”
   Else print “System is in unsafe state” and “Processes are in deadlock”
8. Stop