FCFS

```
#include<stdio.h>
int main()
{
   int wt[10],bt[10],tat[10],rt[10];
   int n;
   float
sum_wt=0,sum_bt=0,sum_tat=0,sum_rt=0;
printf("Enter the number of processes:\n");
scanf("%d",&n);    printf("Enter  the burst time for
the processes:");    for(int i=0;i<n;i++)
  {
      printf("\nProcess[%d]:",i+1);
scanf("%d",&bt[i]);
   }
   for(int i=0;i<n;i++)
  {
      wt[i]=wt[i-1] + bt[i-1];
rt[i]=wt[i];
   }

   for(int i=0;i<n;i++)
   {
      tat[i]=wt[i]+bt[i];
printf("\nProcess[%d]:",i+1);
```

```
printf("\nBurst time:%d",bt[i]);
printf("\nwaiting time:%d",wt[i]);
printf("\nTurn around time:%d",tat[i]);
printf("\nResponse time:%d",rt[i]);
sum_wt=sum_wt+wt[i];
sum_bt=sum_bt+bt[i];
sum_tat=sum_tat+tat[i];
sum_rt=sum_rt+rt[i];       printf("\n");
   }

   printf("\navgerage waiting time:%.2f",sum_wt/n);
printf("\naverage burst time:%.2f",sum_bt/n);
printf("\naverage turn around time:%.2f",sum_tat/n);
printf("\naverage response time:%.2f",sum_rt/n);

   return 0;
}
```

SJF

```
#include <stdio.h>

struct Process {
   int id;
int bT;
int wT;
int tAT;
int rT;
};

void sortpBybT(struct Process p[], int n) {
   for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++) {
if (p[j].bT > p[j + 1].bT) {
struct Process temp = p[j];
p[j] = p[j + 1];              p[j + 1] =
temp;
         }
      }
   }
}
```

```
void calculateTimes(struct Process p[], int n)
{    p[0].wT = 0;    p[0].rT = 0;

   for (int i = 1; i < n; i++) {      p[i].wT
= p[i - 1].wT + p[i - 1].bT;       p[i].rT
= p[i].wT;
   }

   for (int i = 0; i < n; i++) {
p[i].tAT = p[i].wT + p[i].bT;
   }
}

void display(struct Process p[], int n)
{    int totalbT = 0;    int totalwT = 0;
int totaltAT = 0;    int totalrT = 0;

   printf("Process ID\tBurst Time\tWaiting Time\tTurnaround Time\tResponse
Time\n");    for (int i = 0; i < n; i++) {       printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].id,
p[i].bT, p[i].wT, p[i].tAT, p[i].rT);       totalbT += p[i].bT;       totalwT += p[i].wT;
totaltAT += p[i].tAT;       totalrT += p[i].rT;
   }

   printf("\nAverage Burst Time: %.2f\n", (float)totalbT / n);
printf("Average Waiting Time: %.2f\n", (float)totalwT / n);
```

```
printf("Average Turnaround Time: %.2f\n", (float)totaltAT / n);
printf("Average Response Time: %.2f\n", (float)totalrT / n);
}

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;
printf("Enter burst time for process %d: ", i + 1);
scanf("%d", &p[i].bT);
   }

   sortpBybT(p, n);
calculateTimes(p, n);
display(p, n);

   return 0;
}
```

PRIORITY SCHEDULING

```
#include <stdio.h>

#define MAX_PRIORITY 100

struct Process {
   int id;    int
bT;    int
remainT;
int priority;
int wT;    int
tAT;    int cT;
int rT;    int
started;
};

void findwT(struct Process p[], int n) {    int completed = 0, t = 0, minPriority =
MAX_PRIORITY, shortest = 0, finishTime;    int check = 0;    while (completed
!= n) {
      for (int j = 0; j < n; j++) {          if
((p[j].priority < minPriority) &&
            (p[j].remainT > 0)) {
minPriority = p[j].priority;
shortest = j;
```

```
check = 1;

        }
    }


    if (check == 0) {
t++;
continue;
    }


    if (p[shortest].started == 0) {
p[shortest].rT = t;
p[shortest].started = 1;
    }


    p[shortest].remainT--;
minPriority = p[shortest].priority;


    if (p[shortest].remainT == 0) {
minPriority = MAX_PRIORITY;
completed++;          finishTime = t + 1;
p[shortest].cT = finishTime;          p[shortest].wT
= finishTime - p[shortest].bT;
p[shortest].tAT = finishTime;          if
(p[shortest].wT < 0) p[shortest].wT = 0;
    }
```

```
      t++;
    }
}

void findAverageTimes(struct Process p[], int n) {
int totalwT = 0, totaltAT = 0, totalrT = 0;

    findwT(p, n);

    printf("Process ID\tBurst Time\tPriority\tWaiting Time\tTurnaround
Time\tResponse Time\n");
    for (int i = 0; i < n; i++) {
totalwT += p[i].wT;
totaltAT += p[i].tAT;
totalrT += p[i].rT;

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
p[i].id, p[i].bT, p[i].priority,              p[i].wT,
p[i].tAT, p[i].rT);
    }

    printf("\nAverage Waiting Time: %.2f", (float)totalwT / n);
printf("\nAverage Turnaround Time: %.2f", (float)totaltAT / n);
printf("\nAverage Response Time: %.2f", (float)totalrT / n);
}
```

```
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;
printf("Enter burst time for process %d: ", i + 1);
scanf("%d", &p[i].bT);        printf("Enter priority for
process %d: ", i + 1);        scanf("%d",
&p[i].priority);        p[i].remainT = p[i].bT;
p[i].started = 0;
   }

   findAverageTimes(p, n);

   return 0;
}
```

BANKERS ALGORITHM

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
   // P0, P1, P2, P3, P4 are the Process names
here

   int n, m, i, j, k;
   n = 5; // Number of processes     m = 3; // Number
of resources     int alloc[5][3] = { { 0, 1, 0 }, // P0    //
Allocation Matrix
                  { 2, 0, 0 }, // P1
                  { 3, 0, 2 }, // P2
                  { 2, 1, 1 }, // P3
                  { 0, 0, 2 } }; // P4

   int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
                 { 3, 2, 2 }, // P1
                 { 9, 0, 2 }, // P2
                 { 2, 2, 2 }, // P3
                 { 4, 3, 3 } }; // P4

   int avail[3] = { 3, 3, 2 }; // Available Resources
```

```
   int f[n], ans[n];     int
ind = 0;     for (k = 0; k <
n; k++) {        f[k] = 0;

   }
   int need[n][m];     for (i = 0; i < n;
i++) {        for (j = 0; j < m; j++)
need[i][j] = max[i][j] - alloc[i][j];
   }
   int y = 0;     for (k = 0;
k < 5; k++) {        for (i =
0; i < n; i++) {           if
(f[i] == 0) {

            int flag = 0;              for
(j = 0; j < m; j++) {                 if
(need[i][j] > avail[j]){
flag = 1;                     break;
               }
            }

            if (flag == 0) {
ans[ind++] = i;                  for (y =
0; y < m; y++)
avail[y] += alloc[i][y];
f[i] = 1;
```

```
            }
          }
        }
      }

    int flag = 1;

    for(int i=0;i<n;i++)
  {
    if(f[i]==0)
    {
      flag=0;        printf("The following system
is not safe");       break;
    }
  }

  if(flag==1)
  {
    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n - 1; i++)
printf(" P%d ->", ans[i]);
printf(" P%d", ans[n - 1]);
  }

  return 0;
}
```

FIFO PAGE REPLACEMENT

```
#include <stdio.h>

#define MAX_FRAMES 3 // Number of frames in physical memory

int main() {    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2}; // Sequence of
page references    int n = sizeof(pages) / sizeof(pages[0]); // Number of pages
int frames[MAX_FRAMES]; // Array to hold the pages in the frames    int
page_faults = 0; // Counter for page faults    int page_hits = 0;   // Counter for
page hits

  for (int i = 0; i < MAX_FRAMES; i++) {
frames[i] = -1; // Initialize frames as empty
  }

  int front = 0; // Front of the FIFO queue

  for (int i = 0; i < n; i++) {
int   page   =   pages[i];
int page_found = 0;

     // Check if the page is already in a frame
for (int j = 0; j < MAX_FRAMES; j++) {          if
(frames[j] == page) {             page_found = 1;
```

```
page_hits++; // Increment page hits if found
break;
        }
      }

      // If not found, replace the oldest page
(FIFO)       if (!page_found) {          frames[front]
= page;          front = (front + 1) %
MAX_FRAMES;          page_faults++;
      }
   }

   double fault_ratio = (double)page_faults / n;
double hit_ratio = (double)page_hits / n;

   printf("Total page faults: %d\n", page_faults);
printf("Total page hits: %d\n", page_hits);
printf("Fault ratio: %.2f\n", fault_ratio);
printf("Hit ratio: %.2f\n", hit_ratio);

   return 0;
}
```

SYSTEM CALLS

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

int main() {
   pid_t pid;
   int status;

   // Create a new process
   pid = fork();

   if (pid == -1) {
      // Fork failed
      perror("fork");
      exit(EXIT_FAILURE);
   } else if (pid == 0) {
      // Child process
      printf("Child process (PID: %d) executing...\n", getpid());

      // Replace the current process with a new process
      execlp("/bin/ls", "ls", NULL);

      // If execlp fails
```

```
        perror("execlp");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
        printf("Parent process (PID: %d) waiting for child to complete...\n", getpid());

        // Wait for the child process to finish
        if (waitpid(pid, &status, 0) == -1) {
            perror("waitpid");
            exit(EXIT_FAILURE);
        }

        // Check if the child process terminated normally
        if (WIFEXITED(status)) {
            printf("Child process terminated with exit status: %d\n",
WEXITSTATUS(status));
        } else {
            printf("Child process did not terminate normally.\n");
        }

        printf("Parent process exiting...\n");
    }

    return 0;
}
```