1. **Simulate the following CPU scheduling algorithms**
   1. **FCFS**

**#include <stdio.h>**

**int main() {**

**int n;**

**printf("Enter the number of processes: ");**

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

**int bt[n], at[n], wt[n], tat[n], ct[n];**

**float total\_wt = 0, total\_tat = 0;**

**// Input arrival times**

**printf("Enter the arrival time for each process:\n");**

**for (int i = 0; i < n; i++) {**

**printf("Arrival time for P%d: ", i + 1);**

**scanf("%d", &at[i]);**

**}**

**// Input burst times**

**printf("Enter the burst time for each process:\n");**

**for (int i = 0; i < n; i++) {**

**printf("Burst time for P%d: ", i + 1);**

**scanf("%d", &bt[i]);**

**}**

**// Sort by arrival time**

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

**for (int j = i + 1; j < n; j++) {**

**if (at[i] > at[j]) {**

**int temp = at[i];**

**at[i] = at[j];**

**at[j] = temp;**

**temp = bt[i];**

**bt[i] = bt[j];**

**bt[j] = temp;**

**}**

**}**

**}**

**// Calculate completion time, turnaround time and waiting time**

**int time = 0;**

**for (int i = 0; i < n; i++) {**

**if (time < at[i]) {**

**time = at[i]; // CPU idle time**

**}**

**ct[i] = time + bt[i];**

**tat[i] = ct[i] - at[i];**

**wt[i] = tat[i] - bt[i];**

**time = ct[i];**

**total\_wt += wt[i];**

**total\_tat += tat[i];**

**}**

**// Output table**

**printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");**

**for (int i = 0; i < n; i++) {**

**printf("P%d\t\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);**

**}**

**// Averages**

**printf("\nAverage Turnaround Time: %.2f", total\_tat / n);**

**printf("\nAverage Waiting Time: %.2f\n", total\_wt / n);**

**return 0;**

**}**

* 1. **SJF**

**#include <stdio.h>**

**int main() {**

**int n, i, j;**

**int at[10], bt[10], wt[10], tat[10];**

**int executed[10]; // To mark executed processes**

**int pid[10]; // For output order**

**int time = 0, count = 0;**

**float totalWT = 0, totalTAT = 0;**

**printf("Enter number of processes: ");**

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

**// Input AT**

**printf("Enter Arrival Times:\n");**

**for(i = 0; i < n; i++) {**

**printf("AT for P%d: ", i+1);**

**scanf("%d", &at[i]);**

**executed[i] = 0; // Initially, none executed**

**}**

**// Input BT**

**printf("Enter Burst Times:\n");**

**for(i = 0; i < n; i++) {**

**printf("BT for P%d: ", i+1);**

**scanf("%d", &bt[i]);**

**}**

**// SJF Execution Loop**

**while(count < n) {**

**int min = -1;**

**// Find shortest job among arrived and not executed**

**for(i = 0; i < n; i++) {**

**if(!executed[i] && at[i] <= time) {**

**if(min == -1 || bt[i] < bt[min])**

**min = i;**

**}**

**}**

**// If no process arrived yet, jump time**

**if(min == -1) {**

**time++;**

**continue;**

**}**

**// Process min is selecte**

**wt[min] = time - at[min];**

**tat[min] = wt[min] + bt[min];**

**time += bt[min];**

**executed[min] = 1;**

**pid[count] = min;**

**count++;**

**totalWT += wt[min];**

**totalTAT += tat[min];**

**}**

**// Final Output**

**printf("\nPID\tAT\tBT\tWT\tTAT\n");**

**for(i = 0; i < n; i++) {**

**int p = pid[i];**

**printf("P%d\t%d\t%d\t%d\t%d\n", p+1, at[p], bt[p], wt[p], tat[p]);**

**}**

**printf("\nAverage Waiting Time = %.2f", totalWT / n);**

**printf("\nAverage Turnaround Time = %.2f\n", totalTAT / n);**

**return 0;**

**}**

1. **Simulate the following CPU scheduling algorithms**
2. **Priority**

**#include <stdio.h>**

**int main() {**

**int n;**

**printf("Enter number of processes: ");**

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

**int at[n], bt[n], pr[n], p\_id[n];**

**int wt[n], tat[n], ct[n], completed[n];**

**float total\_wt = 0, total\_tat = 0;**

**// Input**

**for (int i = 0; i < n; i++) {**

**printf("Enter Arrival Time, Burst Time, and Priority for P%d: ", i + 1);**

**scanf("%d %d %d", &at[i], &bt[i], &pr[i]);**

**p\_id[i] = i + 1;**

**completed[i] = 0;**

**}**

**int time = 0, count = 0;**

**printf("\nExecution Order:\n");**

**printf("Process\tAT\tBT\tPR\tWT\tTAT\n");**

**while (count < n) {**

**int idx = -1;**

**int min\_priority = 9999;**

**for (int i = 0; i < n; i++) {**

**if (!completed[i] && at[i] <= time) {**

**if (pr[i] < min\_priority || (pr[i] == min\_priority && at[i] < at[idx])) {**

**min\_priority = pr[i];**

**idx = i;**

**}**

**}**

**}**

**if (idx != -1) {**

**time += bt[idx];**

**ct[idx] = time;**

**tat[idx] = ct[idx] - at[idx];**

**wt[idx] = tat[idx] - bt[idx];**

**completed[idx] = 1;**

**total\_tat += tat[idx];**

**total\_wt += wt[idx];**

**count++;**

**// Print as they execute**

**printf("P%d\t\t%d\t%d\t%d\t%d\t%d\n", p\_id[idx], at[idx], bt[idx], pr[idx], wt[idx], tat[idx]);**

**} else {**

**time++; // No process ready yet**

**}**

**}**

**printf("\nAverage Waiting Time: %.2f", total\_wt / n);**

**printf("\nAverage Turnaround Time: %.2f\n", total\_tat / n);**

**return 0;**

**}**

1. **Round Robin**

**#include <stdio.h>**

**int main() {**

**int n, tq;**

**printf("Enter number of processes: ");**

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

**printf("Enter time quantum: ");**

**scanf("%d", &tq);**

**int bt[n], at[n], rt[n]; // bt = burst time, at = arrival time, rt = remaining time**

**int wt[n], tat[n]; // wt = waiting time, tat = turnaround time**

**int total\_wt = 0, total\_tat = 0;**

**int time = 0, completed = 0;**

**// Input process info**

**for (int i = 0; i < n; i++) {**

**printf("Enter Arrival Time and Burst Time for P%d: ", i + 1);**

**scanf("%d %d", &at[i], &bt[i]);**

**rt[i] = bt[i]; // remaining time initially equals burst time**

**}**

**// Round Robin logic**

**while (completed < n) {**

**int done = 1;**

**for (int i = 0; i < n; i++) {**

**if (rt[i] > 0 && at[i] <= time) {**

**done = 0;**

**if (rt[i] > tq) {**

**time += tq;**

**rt[i] -= tq;**

**} else {**

**time += rt[i];**

**wt[i] = time - at[i] - bt[i]; // WT = current time - arrival - burst**

**tat[i] = wt[i] + bt[i]; // TAT = WT + BT**

**rt[i] = 0;**

**completed++;**

**}**

**}**

**}**

**if (done) {**

**time++; // if no process was ready, increment time**

**}**

**}**

**// Output**

**printf("\nProcess\tAT\tBT\tWT\tTAT\n");**

**for (int i = 0; i < n; i++) {**

**printf("P%d\t\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], wt[i], tat[i]);**

**total\_wt += wt[i];**

**total\_tat += tat[i];**

**}**

**printf("\nAverage Waiting Time: %.2f\n", (float)total\_wt / n);**

**printf("Average Turnaround Time: %.2f\n", (float)total\_tat / n);**

**return 0;**

**}**

1. **Implement the following memory allocation methods for fixed partition**
   1. **First fit**

**#include<stdio.h>  
int main() {  
    int m[100],p[100],n1,n2;  
    printf("Enter size of memory block: ");  
    scanf("%d",&n1);  
    printf("Enter no of processors: ");  
    scanf("%d",&n2);  
    printf("Enter Memories: ");  
    for(int i=0;i<n1;i++){  
        scanf("%d",&m[i]);  
    }  
    printf("Enter Processor sizes: ");  
    for(int i=0;i<n2;i++){  
        scanf("%d",&p[i]);  
    }  
    for(int i=0;i<n2;i++){  
        int c=0;  
        for(int j=0;j<n1;j++){  
            if(p[i]<=m[j]){  
                printf("Allocated for Process P%d\n",i);  
                m[j]-=p[i];  
                c++;  
                break;  
            }  
        }  
        if(c==0){  
            printf("No space to allocate memory for P%d\n",i);  
        }  
    }  
    return 0;  
}**

**b) Worst fit**

**#include<stdio.h>  
int main() {  
    int m[100], p[100], n1, n2;  
    printf("Enter number of memory blocks: ");  
    scanf("%d", &n1);  
    printf("Enter number of processes: ");  
    scanf("%d", &n2);  
     
    printf("Enter memory block sizes:\n");  
    for (int i = 0; i < n1; i++) {  
        scanf("%d", &m[i]);  
    }  
     
    printf("Enter process sizes:\n");  
    for (int i = 0; i < n2; i++) {  
        scanf("%d", &p[i]);  
    }  
  
    for (int i = 0; i < n2; i++) {  
        int worstIndex = -1;  
        for (int j = 0; j < n1; j++) {  
            if (m[j] >= p[i]) {  
                if (worstIndex == -1 || m[j] > m[worstIndex]) {  
                    worstIndex = j;  
                }  
            }  
        }  
  
        if (worstIndex != -1) {  
            printf("Allocated Process P%d to block of size %d\n", i, m[worstIndex]);  
            m[worstIndex] -= p[i];  
        } else {  
            printf("No space to allocate memory for Process P%d\n", i);  
        }  
    }  
  
    return 0;  
}**

1. **A)Implement the BEST FIT memory allocation methods for fixed partition**

**#include <stdio.h>  
  
int main() {  
    int m[100], p[100], n1, n2;  
    printf("Enter number of memory blocks: ");  
    scanf("%d", &n1);  
    printf("Enter number of processes: ");  
    scanf("%d", &n2);  
  
    printf("Enter memory block sizes: ");  
    for (int i = 0; i < n1; i++) {  
        scanf("%d", &m[i]);  
    }  
  
    printf("Enter process sizes: ");  
    for (int i = 0; i < n2; i++) {  
        scanf("%d", &p[i]);  
    }  
  
    for (int i = 0; i < n2; i++) {  
        int bestIndex = -1;  
        for (int j = 0; j < n1; j++) {  
            if (m[j] >= p[i]) {  
                if (bestIndex == -1 || m[j] < m[bestIndex]) {  
                    bestIndex = j;  
                }  
            }  
        }  
        if (bestIndex != -1) {  
            printf("Process P%d allocated to block %d\n", i, bestIndex);  
            m[bestIndex] -= p[i];  
        } else {  
            printf("No space to allocate memory for P%d\n", i);  
        }  
    }  
  
    return 0;  
}**

**B) Simulate the FIFO page replacement algorithms**

**#include<stdio.h>  
int main() {  
    int l = 0, a[100], f[100],n,fr;  
    printf("Enter no of pages: ");  
    scanf("%d",&n);  
    printf("Enter no of frames: ");  
    scanf("%d",&fr);  
    printf("Enter page numbers:");  
    for(int i = 0; i < n; i++) {  
        scanf("%d", &a[i]);  
    }  
    for(int i = 0; i < fr; i++) {  
        f[i] = -1;  
    }  
    int m = 0;  
    printf("Pgno\tf1\tf2\tf3\n");  
    for(int i = 0; i < n; i++) {  
        int found = 0; // Track if page is found in the frame  
        for(int j = 0; j < fr; j++) {  
            if(f[j] == a[i]) {  
                found = 1; // Page is already in frame  
                break;  
            }  
        }  
        if(found == 0) { // If page is not found  
            f[m] = a[i]; // Replace page in frame  
            m=(m+1)%fr;  
            l++;  
        }  
        printf("%d\t",a[i]);  
        for(int m= 0; m < fr; m++){  
            printf("%d\t",f[m]);  
        }  
        printf("\n");  
    }  
    printf("Faults: %d\n", l);  
    return 0;  
}**

1. **Implement Banker’s Algorithm for Dead Lock avoidance and prevention**

**#include<stdio.h>**

**#include<stdlib.h>**

**int main(){**

**int al[100][100],need[100][100],max[100][100],av[100],c=0,f[100];**

**int sum[100];**

**int m,n;**

**printf("Enter no of processes: ");**

**scanf("%d",&m);**

**printf("Enter no of resources: ");**

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

**printf("Enter allocation values: \n");**

**for(int i=0;i<m;i++){**

**for(int j=0;j<n;j++){**

**scanf("%d",&al[i][j]);**

**sum[j]+=al[i][j];**

**}**

**}**

**printf("Enter Need values: \n");**

**for(int i=0;i<m;i++){**

**for(int j=0;j<n;j++)**

**scanf("%d",&need[i][j]);**

**}**

**printf("Enter Max values: \n");**

**for(int i=0;i<m;i++){**

**for(int j=0;j<n;j++)**

**scanf("%d",&max[i][j]);**

**}**

**printf("Enter available values: \n");**

**for(int i=0;i<n;i++){**

**scanf("%d",&av[i]);**

**sum[i]+=av[i];**

**}**

**for(int i=0;i<m;i++){**

**f[i]=0;**

**}**

**printf("Total resources: \n");**

**for(int i=0;i<n;i++){**

**printf("%d ",sum[i]);**

**}**

**printf("\nSafe sequence: ");**

**while(c<m){**

**for(int i=0;i<m;i++){**

**int fi=1;**

**if(f[i]==0){**

**for(int j=0;j<n;j++){**

**if(need[i][j]>av[j]){**

**fi=0;**

**break;**

**}**

**}**

**if(fi==1){**

**printf("p%d ",i);**

**c++;**

**for(int t=0;t<n;t++){**

**av[t]+=al[i][t];**

**}**

**f[i]=1;**

**}**

**}**

**}**

**}**

**}**

1. **Write a program to solve producer consumer problem using semaphores**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <pthread.h>**

**#include <semaphore.h>**

**#include <unistd.h>**

**#define MAX 5**

**#define NUM\_ITEMS 5**

**sem\_t empty, full, mutex;**

**int buffer[MAX];**

**int in = 0, out = 0;**

**int produced\_items = 0, consumed\_items = 0;**

**void\* producer(void\* arg) {**

**while (produced\_items < NUM\_ITEMS) {**

**int item = rand() % 100;**

**sem\_wait(&empty);**

**sem\_wait(&mutex);**

**buffer[in] = item;**

**printf("Produced: %d at position %d\n", item, in);**

**in = (in + 1) % MAX;**

**produced\_items++;**

**sem\_post(&mutex);**

**sem\_post(&full);**

**usleep(1000000);**

**}**

**return NULL;**

**}**

**void\* consumer(void\* arg) {**

**while (consumed\_items < NUM\_ITEMS) {**

**sem\_wait(&full);**

**sem\_wait(&mutex);**

**int item = buffer[out];**

**printf("Consumed: %d from position %d\n", item, out);**

**out = (out + 1) % MAX;**

**consumed\_items++;**

**sem\_post(&mutex);**

**sem\_post(&empty);**

**usleep(1000000);**

**}**

**return NULL;**

**}**

**int main() {**

**pthread\_t prod, cons;**

**sem\_init(&empty, 0, MAX);**

**sem\_init(&full, 0, 0);**

**sem\_init(&mutex, 0, 1);**

**pthread\_create(&prod, NULL, producer, NULL);**

**pthread\_create(&cons, NULL, consumer, NULL);**

**pthread\_join(prod, NULL);**

**pthread\_join(cons, NULL);**

**sem\_destroy(&empty);**

**sem\_destroy(&full);**

**sem\_destroy(&mutex);**

**return 0;**

**}**