Dr. D. Y. Patil Unitech Society's
DR. D. Y. PATIL ARTS, COMMERCE AND SCIENCE COLLEGE, PIMPRI, PUNE - 18
Department of Computer Science
Academic Year 2024 - 2025
M.Sc (CA) (Semester-1) Operating System Practical Assignments

Q1.Write a program to implement FCFS CPU scheduling algorithm. Take arrival time, burst time for n number of processes from the user. Calculate average waiting time.

```
#include <stdio.h>
void findWaitingTime(int n, int bt[], int at[], int wt[]) {
   // Waiting time for the first process is always 0
   wt[0] = 0;
      // Calculate waiting time for all other processes
   for (int i = 1; i < n; i++) {
      wt[i] = bt[i-1] + wt[i-1] - at[i];
      if (wt[i] < 0) {
         wt[i] = 0;  // Waiting time cannot be negative
      }
   }
}

void findTurnAroundTime(int n, int bt[], int at[], int wt[], int tat[]) {
   // Turnaround time is burst time + waiting time
   for (int i = 0; i < n; i++) {
      tat[i] = bt[i] + wt[i];
   }
}

void findAverageTimes(int n, int bt[], int at[]) {
   int wt[n], tat[n];
     // Calculate waiting time
   findWaitingTime(n, bt, at, wt);
      // Calculate turnaround time
   findTurnAroundTime(n, bt, at, wt, tat);
      // Calculate total waiting time and total turnaround time
   int total_wt = 0, total_tat = 0;
   for (int i = 0; i < n; i++) {
      total_wt += wt[i];
      total_tat += tat[i];
   }
      // Calculate and print average waiting time and average turnaround time
   printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
   for (int i = 0; i < n; i++) {
      printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i+1, at[i], bt[i], wt[i], tat[i]);
   }
   printf("\nAverage Waiting Time: %.2f", (float)total_wt / n);
  }
int main() {
   int n;
      // Take number of processes as input
   printf("Enter number of processes: ");
  scanf("%d", &n);
    int bt[n], at[n];
    // Take burst time and arrival time for each process
  for (int i = 0; i < n; i++) {
    printf("\nEnter Burst Time and Arrival Time for Process %d: ", i + 1);
    scanf("%d %d", &bt[i], &at[i]);
  }
    // Sort the processes based on arrival time for FCFS
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
      if (at[i] > at[j]) {
        // Swap burst time
        int temp = bt[i];
        bt[i] = bt[j];
        bt[j] = temp;
                // Swap arrival time
        temp = at[i];
        at[i] = at[j];
        at[j] = temp;
      }
    }
  }

  // Call function to calculate average times
  findAverageTimes(n, bt, at);

  return 0;
}
```

Q2.Write a program to implement FCFS CPU scheduling algorithm. Take arrival time, burst time for n number of processes from the user. Calculate average turnaround time.

```
#include <stdio.h>

// Function to calculate Turnaround Time
void findTurnaroundTime(int n, int bt[], int at[], int wt[], int tat[]) {
  // Turnaround time is burst time + waiting time
  for (int i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i];
  }
}

// Function to calculate Waiting Time
void findWaitingTime(int n, int bt[], int at[], int wt[]) {
  // Waiting time for the first process is always 0
  wt[0] = 0;

  // Calculate waiting time for all other processes
  for (int i = 1; i < n; i++) {
    wt[i] = bt[i-1] + wt[i-1] - at[i];
    if (wt[i] < 0) {
      wt[i] = 0;  // Waiting time cannot be negative
    }
    }
}

// Function to calculate average times
void findAverageTimes(int n, int bt[], int at[]) {
    int wt[n], tat[n];

    // Calculate waiting time
    findWaitingTime(n, bt, at, wt);

    // Calculate turnaround time
    findTurnaroundTime(n, bt, at, wt, tat);

    // Calculate total turnaround time
    int total_tat = 0;
    for (int i = 0; i < n; i++) {
        total_tat += tat[i];
    }

    // Calculate and print average turnaround time
    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i+1, at[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Turnaround Time: %.2f", (float)total_tat / n);
}

int main() {
    int n;

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

    int bt[n], at[n];

    // Take burst time and arrival time for each process
    for (int i = 0; i < n; i++) {
        printf("\nEnter Burst Time and Arrival Time for Process %d: ", i + 1);
        scanf("%d %d", &bt[i], &at[i]);
    }

    // Sort the processes based on arrival time for FCFS
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (at[i] > at[j]) {
                // Swap burst time
                int temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                // Swap arrival time
                temp = at[i];
                at[i] = at[j];
                at[j] = temp;
            }
        }
    }

    // Call function to calculate average turnaround time
    findAverageTimes(n, bt, at);

    return 0;
}
```

Q3.Write a program to simulate Pre-emptive Shortest Job First (SJF) CPU scheduling algorithm. Accept no. of processes, arrival time and burst time from user. Calculate and display the average waiting time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Function to calculate waiting time
void calculateWaitingTime(int n, int bt[], int at[], int wt[]) {
    int remaining_bt[n];  // Remaining burst time
    int complete = 0, time = 0, min_time, shortest, finish_time;
    int check[n];  // To track whether a process has been completed

    // Initialize remaining burst time and check array
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
        check[i] = 0;
    }

    // Execute the process until all processes are completed
    while (complete < n) {
        min_time = 9999;
        shortest = -1;

        // Find the process with the shortest remaining burst time
        for (int i = 0; i < n; i++) {
            if ((at[i] <= time) && (check[i] == 0) && (remaining_bt[i] < min_time) && (remaining_bt[i] > 0)) {
                min_time = remaining_bt[i];
                shortest = i;
            }
        }

        // If no process is found, increment the time and continue
        if (shortest == -1) {
            time++;
            continue;
        }

        // Process the selected shortest job
        remaining_bt[shortest]--;

        // If the process is completed
        if (remaining_bt[shortest] == 0) {
            complete++;
            finish_time = time + 1;
            wt[shortest] = finish_time - at[shortest] - bt[shortest];
            if (wt[shortest] < 0) {
                wt[shortest] = 0;  // Waiting time cannot be negative
            }
        }

        // Increment time
        time++;
    }
}

// Function to calculate the average waiting time
void calculateAverageWaitingTime(int n, int bt[], int at[]) {
    int wt[n];
    int total_wt = 0;

    // Calculate waiting time for each process
    calculateWaitingTime(n, bt, at, wt);

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

    // Calculate average waiting time
    float avg_wt = (float)total_wt / n;
    printf("\nAverage Waiting Time: %.2f\n", avg_wt);
}

int main() {
    int n;

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

    int bt[n], at[n];

    // Take burst time and arrival time for each process
    for (int i = 0; i < n; i++) {
        printf("\nEnter Burst Time and Arrival Time for Process %d: ", i + 1);
        scanf("%d %d", &bt[i], &at[i]);
    }

    // Call function to calculate average waiting time
    calculateAverageWaitingTime(n, bt, at);

    return 0;
```

```
}
```

Q4.Write a program to simulate Pre-emptive Shortest Job First (SJF) CPU scheduling algorithm. Accept no. of processes, arrival time and burst time from the user. Calculate and Display the average turnaround time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Function to calculate Turnaround Time
void calculateTurnaroundTime(int n, int bt[], int at[], int wt[], int tat[]) {
  // Turnaround time is burst time + waiting time
  for (int i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i];
  }
}

// Function to calculate Waiting Time
void calculateWaitingTime(int n, int bt[], int at[], int wt[]) {
  int remaining_bt[n];  // Remaining burst time
  int complete = 0, time = 0, min_time, shortest, finish_time;
  int check[n];  // To track whether a process has been completed

  // Initialize remaining burst time and check array
  for (int i = 0; i < n; i++) {
    remaining_bt[i] = bt[i];
    check[i] = 0;
  }

  // Execute the process until all processes are completed
  while (complete < n) {
    min_time = 9999;
    shortest = -1;

    // Find the process with the shortest remaining burst time
    for (int i = 0; i < n; i++) {
            if ((at[i] <= time) && (check[i] == 0) && (remaining_bt[i] < min_time) &&
(remaining_bt[i] > 0)) {
        min_time = remaining_bt[i];
        shortest = i;
      }
    }

    // If no process is found, increment the time and continue
    if (shortest == -1) {
      time++;
      continue;
    }

    // Process the selected shortest job
    remaining_bt[shortest]--;
```

```
        // If the process is completed
        if (remaining_bt[shortest] == 0) {
            complete++;
            finish_time = time + 1;
            wt[shortest] = finish_time - at[shortest] - bt[shortest];
            if (wt[shortest] < 0) {
                wt[shortest] = 0;  // Waiting time cannot be negative
            }
        }

        // Increment time
        time++;
    }
}

// Function to calculate the average turnaround time
void calculateAverageTurnaroundTime(int n, int bt[], int at[]) {
    int wt[n], tat[n];
    int total_tat = 0;

    // Calculate waiting time for each process
    calculateWaitingTime(n, bt, at, wt);

    // Calculate turnaround time for each process
    calculateTurnaroundTime(n, bt, at, wt, tat);

    // Calculate total turnaround time
    for (int i = 0; i < n; i++) {
        total_tat += tat[i];
    }

    // Calculate average turnaround time
    float avg_tat = (float)total_tat / n;
    printf("\nAverage Turnaround Time: %.2f\n", avg_tat);
}

int main() {
    int n;

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

    int bt[n], at[n];

    // Take burst time and arrival time for each process
    for (int i = 0; i < n; i++) {
        printf("\nEnter Burst Time and Arrival Time for Process %d: ", i + 1);
        scanf("%d %d", &bt[i], &at[i]);
    }

    // Call function to calculate average turnaround time
    calculateAverageTurnaroundTime(n, bt, at);
    return 0;
}
```

Q5.Write a program to simulate Non-Pre-emptive Shortest Job First (SJF) scheduling. Accept no. of processes, arrival time and burst time. Calculate and display the average waiting time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Function to calculate Waiting Time
void calculateWaitingTime(int n, int bt[], int at[], int wt[]) {
    int remaining_bt[n], completed[n];
    int time = 0, complete = 0, min_time, shortest = -1;
    int finish_time, total_wt = 0;

    // Initialize remaining burst times and completed status
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
        completed[i] = 0; // Not completed yet
    }

    while (complete < n) {
        min_time = 9999;
        // Select the shortest job (non-preemptive)
        for (int i = 0; i < n; i++) {
            if (at[i] <= time && completed[i] == 0 && remaining_bt[i] < min_time) {
                min_time = remaining_bt[i];
                shortest = i;
            }
        }

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

        // Process the shortest job
            time += remaining_bt[shortest];  // Increase current time by the burst time of the selected
process
        finish_time = time;
        wt[shortest] = finish_time - at[shortest] - bt[shortest];

        if (wt[shortest] < 0) {
            wt[shortest] = 0; // Waiting time can't be negative
        }

        // Mark process as completed
        completed[shortest] = 1;
        complete++;
    }
}
```

```
// Function to calculate Average Waiting Time
void calculateAverageWaitingTime(int n, int bt[], int at[]) {
    int wt[n], total_wt = 0;
    calculateWaitingTime(n, bt, at, wt);

    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
    }

    float avg_wt = (float)total_wt / n;
    printf("\nAverage Waiting Time: %.2f\n", avg_wt);
}

int main() {
    int n;

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

    int bt[n], at[n];

    // Take burst time and arrival time for each process
    for (int i = 0; i < n; i++) {
        printf("\nEnter Burst Time and Arrival Time for Process %d: ", i + 1);
        scanf("%d %d", &bt[i], &at[i]);
    }

    // Call function to calculate average waiting time
    calculateAverageWaitingTime(n, bt, at);

    return 0;
}
```

Q6.Write a program to simulate Non-Pre-emptive Shortest Job First (SJF) scheduling. Accept no. of processes, arrival time and burst time. Calculate and display the average turnaround time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Function to calculate Turnaround Time
void calculateTurnaroundTime(int n, int bt[], int at[], int wt[], int tat[]) {
    // Turnaround time is burst time + waiting time
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

// Function to calculate Waiting Time
void calculateWaitingTime(int n, int bt[], int at[], int wt[]) {
    int remaining_bt[n], completed[n];
    int time = 0, complete = 0, min_time, shortest = -1;
    int finish_time;

    // Initialize remaining burst times and completed status
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
        completed[i] = 0; // Not completed yet
    }

    while (complete < n) {
        min_time = 9999;
        // Select the shortest job (non-preemptive)
        for (int i = 0; i < n; i++) {
            if (at[i] <= time && completed[i] == 0 && remaining_bt[i] < min_time) {
                min_time = remaining_bt[i];
                shortest = i;
            }
        }

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

        // Process the shortest job
            time += remaining_bt[shortest];  // Increase current time by the burst time of the selected
process
        finish_time = time;
        wt[shortest] = finish_time - at[shortest] - bt[shortest];

        if (wt[shortest] < 0) {
            wt[shortest] = 0; // Waiting time can't be negative
        }

        // Mark process as completed
        completed[shortest] = 1;
        complete++;
    }
}

// Function to calculate Average Turnaround Time
void calculateAverageTurnaroundTime(int n, int bt[], int at[]) {
    int wt[n], tat[n], total_tat = 0;

    calculateWaitingTime(n, bt, at, wt);
    calculateTurnaroundTime(n, bt, at, wt, tat);

    // Calculate total turnaround time
    for (int i = 0; i < n; i++) {
        total_tat += tat[i];
    }

    float avg_tat = (float)total_tat / n;
    printf("\nAverage Turnaround Time: %.2f\n", avg_tat);
}

int main() {
    int n;

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

    int bt[n], at[n];

    // Take burst time and arrival time for each process
    for (int i = 0; i < n; i++) {
        printf("\nEnter Burst Time and Arrival Time for Process %d: ", i + 1);
        scanf("%d %d", &bt[i], &at[i]);
    }

    // Call function to calculate average turnaround time
    calculateAverageTurnaroundTime(n, bt, at);

    return 0;
}
```

Q7.Write a program for Round Robin (RR) scheduling for a given time quantum. Accept no. of processes, arrival time and burst time for every process and time quantum. Calculate the waiting time of every process and Display the average waiting time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Function to calculate Waiting Time for Round Robin Scheduling
void calculateWaitingTime(int n, int bt[], int at[], int wt[], int tq) {
    int remaining_bt[n], completed[n];
    int time = 0, complete = 0, turn_around_time;

    // Initialize remaining burst times and completed status
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
        completed[i] = 0; // Not completed yet
    }

    // Round Robin Scheduling
    while (complete < n) {
        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0 && at[i] <= time) {
                // Process execution
                if (remaining_bt[i] > tq) {
                    time += tq;
                    remaining_bt[i] -= tq;
                } else {
                    time += remaining_bt[i];
                    wt[i] = time - at[i] - bt[i];
                    if (wt[i] < 0) {
                        wt[i] = 0; // Waiting time can't be negative
                    }
                    remaining_bt[i] = 0;
                    completed[i] = 1;
                    complete++;
                }
            }
        }
    }
}

// Function to calculate Average Waiting Time
void calculateAverageWaitingTime(int n, int bt[], int at[], int tq) {
    int wt[n], total_wt = 0;

    calculateWaitingTime(n, bt, at, wt, tq);

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

    float avg_wt = (float)total_wt / n;
    printf("\nAverage Waiting Time: %.2f\n", avg_wt);
}

int main() {
    int n, tq;

    // Take number of processes and time quantum as input
    printf("Enter number of processes: ");
    scanf("%d", &n);

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

    int bt[n], at[n];

    // Take burst time and arrival time for each process
    for (int i = 0; i < n; i++) {
        printf("\nEnter Burst Time and Arrival Time for Process %d: ", i + 1);
        scanf("%d %d", &bt[i], &at[i]);
    }

    // Call function to calculate average waiting time
    calculateAverageWaitingTime(n, bt, at, tq);

    return 0;
}
```

Q8.Write a program to implement Bankers algorithm. Mention no. of processes and

available resources. Calculate the need matrix based on max and allocation matrix.

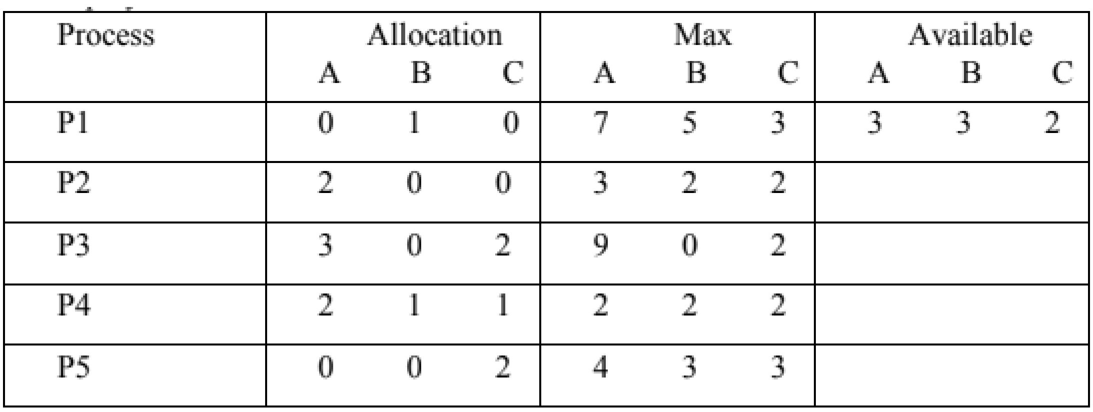
```
#include <stdio.h>

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

// Function to calculate Need matrix
void calculateNeedMatrix(int n, int m, int max[][MAX_RESOURCES], int
alloc[][MAX_RESOURCES], int need[][MAX_RESOURCES]) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         need[i][j] = max[i][j] - alloc[i][j];
      }
   }
}

// Function to display matrix
void displayMatrix(int n, int m, int matrix[][MAX_RESOURCES]) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
}

int main() {
   int n, m;

   // Number of processes (n) and resources (m)
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   printf("Enter the number of resources: ");
   scanf("%d", &m);

         int available[MAX_RESOURCES], max[MAX_PROCESSES][MAX_RESOURCES],
alloc[MAX_PROCESSES][MAX_RESOURCES],
need[MAX_PROCESSES][MAX_RESOURCES];

   // Input Available Resources
   printf("\nEnter available resources:\n");
   for (int i = 0; i < m; i++) {
      printf("Resource %d: ", i + 1);
      scanf("%d", &available[i]);
   }

   // Input Maximum Demand (Max) matrix
   printf("\nEnter Maximum Resource Requirement for each process:\n");
   for (int i = 0; i < n; i++) {
      printf("\nProcess %d:\n", i + 1);
      for (int j = 0; j < m; j++) {
         printf("Resource %d: ", j + 1);
         scanf("%d", &max[i][j]);
```

```
        }
    }

    // Input Allocation Matrix
    printf("\nEnter Allocation Matrix:\n");
    for (int i = 0; i < n; i++) {
        printf("\nProcess %d:\n", i + 1);
        for (int j = 0; j < m; j++) {
            printf("Resource %d: ", j + 1);
            scanf("%d", &alloc[i][j]);
        }
    }

    // Calculate the Need Matrix
    calculateNeedMatrix(n, m, max, alloc, need);

    // Display the Need Matrix
    printf("\nNeed Matrix (Max - Allocation):\n");
    displayMatrix(n, m, need);

    return 0;
}
```

Q9.Consider a system with ‘m’ processes and ‘n’ resource types. Accept number of instances for each resource type. For each process accept the allocation and maximum requirement matrices.Write a program to display the contents of the need matrix.

```
#include <stdio.h>

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

// Function to calculate the Need Matrix
void calculateNeedMatrix(int n, int m, int max[][MAX_RESOURCES], int
alloc[][MAX_RESOURCES], int need[][MAX_RESOURCES]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            need[i][j] = max[i][j] - alloc[i][j]; // Need = Max - Allocation
        }
    }
}

// Function to display a matrix
void displayMatrix(int n, int m, int matrix[][MAX_RESOURCES]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int n, m;

    // Input the number of processes (n) and resource types (m)
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resource types: ");
    scanf("%d", &m);

                                        int        max[MAX_PROCESSES][MAX_RESOURCES],
alloc[MAX_PROCESSES][MAX_RESOURCES],
need[MAX_PROCESSES][MAX_RESOURCES];

    // Input the Allocation Matrix for each process
    printf("\nEnter Allocation Matrix:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        for (int j = 0; j < m; j++) {
            printf("Resource %d: ", j + 1);
            scanf("%d", &alloc[i][j]);
        }
    }

    // Input the Maximum Resource Requirement (Max) Matrix for each process
    printf("\nEnter Maximum Resource Requirement Matrix (Max):\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        for (int j = 0; j < m; j++) {
            printf("Resource %d: ", j + 1);
            scanf("%d", &max[i][j]);
        }
    }

    // Calculate the Need Matrix
    calculateNeedMatrix(n, m, max, alloc, need);

    // Display the Need Matrix
    printf("\nNeed Matrix (Max - Allocation):\n");
    displayMatrix(n, m, need);

    return 0;
}
```

Q10.Write a Program to implement following functionality.
Accept Available
Display Allocation, Max
Display the contents of need matrix



Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

```
// 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], 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);
}
```

Q11.Write a program to simulate Non-pre-emptive Shortest Job First (SJF) – scheduling. Accept no. of processes, arrival time and burst time from user. The output should be the waiting time for each process. Also find the average waiting time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Function to calculate the waiting time and turnaround time for all processes
void calculateWaitingTime(int n, int arrival[], int burst[], int waiting[], int turnaround[]) {
    int completion[MAX_PROCESSES], total_wt = 0, total_tat = 0;

    // Calculate completion times for each process
    completion[0] = arrival[0] + burst[0];  // First process completes after its burst time
    for (int i = 1; i < n; i++) {
        completion[i] = completion[i - 1] + burst[i]; // Completion time of each process
    }

    // Calculate waiting time and turnaround time for each process
    for (int i = 0; i < n; i++) {
        turnaround[i] = completion[i] - arrival[i];
        waiting[i] = turnaround[i] - burst[i];
        total_wt += waiting[i];
        total_tat += turnaround[i];
    }

    // Display waiting time and turnaround time for each process
    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, arrival[i], burst[i], waiting[i], turnaround[i]);
    }

    // Display the average waiting time
    printf("\nAverage Waiting Time: %.2f\n", (float)total_wt / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_tat / n);
}

// Function to sort processes by burst time (Shortest Job First)
void sortByBurstTime(int n, int arrival[], int burst[], int index[]) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (burst[index[i]] > burst[index[j]]) {
                // Swap burst times
                temp = index[i];
                index[i] = index[j];
                index[j] = temp;
            }
        }
    }
}

int main() {
     int n, arrival[MAX_PROCESSES], burst[MAX_PROCESSES], waiting[MAX_PROCESSES],
turnaround[MAX_PROCESSES];

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

    int index[MAX_PROCESSES]; // To keep track of process indices

    // Input arrival times and burst times
    for (int i = 0; i < n; i++) {
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &arrival[i]);
        printf("Burst Time: ");
        scanf("%d", &burst[i]);
        index[i] = i; // Initialize the index array
    }

    // Sort processes based on burst time (SJF)
    sortByBurstTime(n, arrival, burst, index);

    // Calculate the waiting time and turnaround time for each process
    calculateWaitingTime(n, arrival, burst, waiting, turnaround);

    return 0;
}
```

Q12.Write a program to simulate Non-pre-emptive Shortest Job First (SJF) CPU–scheduling.Accept no. of processes, arrival time and burst time from user. The output should be the turnaround time for each process. Also find the average turnaround time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Function to calculate the turnaround time for all processes
void calculateTurnaroundTime(int n, int arrival[], int burst[], int turnaround[]) {
    int completion[MAX_PROCESSES], total_tat = 0;

    // Calculate completion times for each process
    completion[0] = arrival[0] + burst[0];  // First process completes after its burst time
    for (int i = 1; i < n; i++) {
        completion[i] = completion[i - 1] + burst[i]; // Completion time of each process
    }

    // Calculate turnaround time for each process
    for (int i = 0; i < n; i++) {
        turnaround[i] = completion[i] - arrival[i];
        total_tat += turnaround[i];
    }

    // Display turnaround time for each process
    printf("\nProcess\tArrival Time\tBurst Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", i + 1, arrival[i], burst[i], turnaround[i]);
    }

    // Display the average turnaround time
    printf("\nAverage Turnaround Time: %.2f\n", (float)total_tat / n);
}

// Function to sort processes by burst time (Shortest Job First)
void sortByBurstTime(int n, int arrival[], int burst[], int index[]) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (burst[index[i]] > burst[index[j]]) {
                // Swap burst times
                temp = index[i];
                index[i] = index[j];
                index[j] = temp;
            }
        }
    }
}

int main() {
                        int      n,      arrival[MAX_PROCESSES],      burst[MAX_PROCESSES],
turnaround[MAX_PROCESSES];

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

    int index[MAX_PROCESSES]; // To keep track of process indices

    // Input arrival times and burst times
    for (int i = 0; i < n; i++) {
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &arrival[i]);
        printf("Burst Time: ");
        scanf("%d", &burst[i]);
        index[i] = i; // Initialize the index array
    }

    // Sort processes based on burst time (SJF)
    sortByBurstTime(n, arrival, burst, index);

    // Calculate the turnaround time for each process
    calculateTurnaroundTime(n, arrival, burst, turnaround);

    return 0;
}
```

Q13.Write a program to simulate FCFS CPU-scheduling. Accept no. of Processes, arrival time and burst time from user. The output should give Gantt chart, and waiting time for each process.Also find the average waiting time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Function to calculate the waiting time for each process
void calculateWaitingTime(int n, int arrival[], int burst[], int waiting[], int startTime[]) {
    int total_wt = 0;

    // The start time for the first process is its arrival time
    startTime[0] = arrival[0];
    waiting[0] = 0;  // The first process does not have to wait

    // Calculate the start time and waiting time for the rest of the processes
    for (int i = 1; i < n; i++) {
            // Start time of the process is the maximum of (completion of previous process, arrival of
current process)
            startTime[i] = (arrival[i] > startTime[i - 1] + burst[i - 1]) ? arrival[i] : startTime[i - 1] +
burst[i - 1];
        waiting[i] = startTime[i] - arrival[i];  // Waiting time = Start time - Arrival time
        total_wt += waiting[i];
    }

    // Calculate the average waiting time
    printf("\nAverage Waiting Time: %.2f\n", (float)total_wt / n);
}

// Function to display the Gantt chart
void printGanttChart(int n, int arrival[], int burst[], int startTime[]) {
    printf("\nGantt Chart:\n");
    printf("\n|");

    for (int i = 0; i < n; i++) {
        printf(" P%d |", i + 1);
    }

    printf("\n");

    // Print the timeline below the Gantt chart
    printf(" 0 ");
    for (int i = 0; i < n; i++) {
        printf("   %d", startTime[i] + burst[i]);
    }
    printf("\n");
}

int main() {
     int n, arrival[MAX_PROCESSES], burst[MAX_PROCESSES], waiting[MAX_PROCESSES],
startTime[MAX_PROCESSES];

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

    // Input arrival times and burst times
    for (int i = 0; i < n; i++) {
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &arrival[i]);
        printf("Burst Time: ");
        scanf("%d", &burst[i]);
    }

    // Calculate the waiting times for each process
    calculateWaitingTime(n, arrival, burst, waiting, startTime);
    // Display the Gantt chart
    printGanttChart(n, arrival, burst, startTime);

    // Display the waiting time for each process
    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\n", i + 1, arrival[i], burst[i], waiting[i]);
    }

    return 0;
}
```

Q14.Write a program to simulate FCFS CPU-scheduling. Accept no. of Processes, arrival time and burst time from user. The output should give Gantt chart, and turn around time for each process. Also find the average turn around time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Function to calculate the turnaround time for each process
void calculateTurnaroundTime(int n, int arrival[], int burst[], int turnaround[], int completion[]) {
    int total_tat = 0;

    // Calculate completion times for each process
    completion[0] = arrival[0] + burst[0];  // First process completes after its burst time
    for (int i = 1; i < n; i++) {
        completion[i] = completion[i - 1] + burst[i]; // Completion time of each process
    }

    // Calculate turnaround time for each process
    for (int i = 0; i < n; i++) {
        turnaround[i] = completion[i] - arrival[i];
        total_tat += turnaround[i];
    }

    // Display the average turnaround time
    printf("\nAverage Turnaround Time: %.2f\n", (float)total_tat / n);
}

// Function to display the Gantt chart
void printGanttChart(int n, int arrival[], int burst[], int completion[]) {
    printf("\nGantt Chart:\n");
    printf("\n|");

    for (int i = 0; i < n; i++) {
        printf(" P%d |", i + 1);
    }

    printf("\n");

    // Print the timeline below the Gantt chart
    printf(" 0 ");
    for (int i = 0; i < n; i++) {
        printf("   %d", completion[i]);
    }
    printf("\n");
}

int main() {
    int n, arrival[MAX_PROCESSES], burst[MAX_PROCESSES],
turnaround[MAX_PROCESSES], completion[MAX_PROCESSES];

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

    // Input arrival times and burst times
    for (int i = 0; i < n; i++) {
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &arrival[i]);
        printf("Burst Time: ");
        scanf("%d", &burst[i]);
    }

    // Calculate the turnaround times for each process
    calculateTurnaroundTime(n, arrival, burst, turnaround, completion);

    // Display the Gantt chart
    printGanttChart(n, arrival, burst, completion);

    // Display the turnaround time for each process
    printf("\nProcess\tArrival Time\tBurst Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\n", i + 1, arrival[i], burst[i], turnaround[i]);
    }

    return 0;
}
```

Q15.Write a program to simulate Pre-emptive Shortest Job First (SJF) – scheduling. Accept no.of processes, arrival time and burst time from user.The output should be the waiting time for each process. Also find the average waiting time.

```
#include <stdio.h>

#define MAX_PROCESSES 10
#define MAX_TIME 100

// Structure to represent a process
struct Process {
     int id, arrival_time, burst_time, remaining_burst_time, start_time, finish_time, waiting_time,
turnaround_time;
};
```

```
// Function to find the process with the smallest remaining burst time
int findShortestJob(struct Process processes[], int n, int time) {
   int min_time = MAX_TIME;
   int index = -1;

   for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= time && processes[i].remaining_burst_time > 0 &&
processes[i].remaining_burst_time < min_time) {
         min_time = processes[i].remaining_burst_time;
         index = i;
      }
   }
   return index;
}

// Function to calculate waiting times and turnaround times
void calculateTimes(struct Process processes[], int n) {
   int total_waiting_time = 0, total_turnaround_time = 0;

   // Calculate turnaround time and waiting time for each process
   for (int i = 0; i < n; i++) {
      processes[i].turnaround_time = processes[i].finish_time - processes[i].arrival_time;
      processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
      total_waiting_time += processes[i].waiting_time;
      total_turnaround_time += processes[i].turnaround_time;
   }

   // Display the results
   printf("\nProcess ID | Arrival Time | Burst Time | Waiting Time | Turnaround Time\n");
   for (int i = 0; i < n; i++) {
               printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n",  processes[i].id,  processes[i].arrival_time,
processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time);
   }

   // Display the average waiting time and average turnaround time
   printf("\nAverage Waiting Time: %.2f", (float)total_waiting_time / n);
   printf("\nAverage Turnaround Time: %.2f", (float)total_turnaround_time / n);
}

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

   struct Process processes[MAX_PROCESSES];

   // Input arrival time and burst time for each process
   for (int i = 0; i < n; i++) {
      printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
      processes[i].id = i + 1;
      printf("Arrival Time: ");
      scanf("%d", &processes[i].arrival_time);
      printf("Burst Time: ");
```

```
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_burst_time = processes[i].burst_time;
    }

    // Sort processes by arrival time
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].arrival_time > processes[j].arrival_time) {
                struct Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    int time = 0, completed = 0;
    while (completed < n) {
        int index = findShortestJob(processes, n, time);

        if (index == -1) {
            time++; // No process is ready to execute at this time
        } else {
            if (processes[index].start_time == 0) {
                processes[index].start_time = time; // First time the process is being executed
            }
            processes[index].remaining_burst_time--;
            time++;

            if (processes[index].remaining_burst_time == 0) {
                processes[index].finish_time = time;
                completed++;
            }
        }
    }

    // Calculate and display waiting time and turnaround time
    calculateTimes(processes, n);

    return 0;
}
```

Q16.Write a program to simulate Pre-emptive Shortest Job First (SJF) – scheduling. Accept no. of processes, arrival time and burst time from user.The output should be the turn around time for each process. Also find the average turn around time.

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

#define MAX_PROCESSES 10

// Structure to represent a process
struct Process {
        int id, arrival_time, burst_time, remaining_burst_time, start_time, finish_time,
turnaround_time;
};

// Function to find the process with the smallest remaining burst time
int findShortestJob(struct Process processes[], int n, int time) {
    int min_time = INT_MAX;
    int index = -1;

    for (int i = 0; i < n; i++) {
                if (processes[i].arrival_time <= time && processes[i].remaining_burst_time > 0 &&
processes[i].remaining_burst_time < min_time) {
            min_time = processes[i].remaining_burst_time;
            index = i;
        }
    }
    return index;
}

// Function to calculate turnaround times
void calculateTurnaroundTimes(struct Process processes[], int n) {
    int total_turnaround_time = 0;

    // Calculate turnaround time for each process
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].finish_time - processes[i].arrival_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    // Display the results
    printf("\nProcess ID | Arrival Time | Burst Time | Turnaround Time\n");
    for (int i = 0; i < n; i++) {
                            printf("P%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival_time,
processes[i].burst_time, processes[i].turnaround_time);
    }

    // Display the average turnaround time
    printf("\nAverage Turnaround Time: %.2f", (float)total_turnaround_time / n);
}

int main() {
    int n;

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

    struct Process processes[MAX_PROCESSES];

    // Input arrival time and burst time for each process
    for (int i = 0; i < n; i++) {
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        processes[i].id = i + 1;
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_burst_time = processes[i].burst_time;
    }

    // Sort processes by arrival time
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].arrival_time > processes[j].arrival_time) {
                struct Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    int time = 0, completed = 0;
    while (completed < n) {
        int index = findShortestJob(processes, n, time);

        if (index == -1) {
            time++; // No process is ready to execute at this time
        } else {
            if (processes[index].start_time == 0) {
                processes[index].start_time = time; // First time the process is being executed
            }
            processes[index].remaining_burst_time--;
            time++;

            if (processes[index].remaining_burst_time == 0) {
                processes[index].finish_time = time;
                completed++;
            }
        }
    }

    // Calculate and display turnaround times
    calculateTurnaroundTimes(processes, n);

    return 0;
}
```

Q17.Write a program for Round Robin scheduling for given time quantum. Accept no. of processes, arrival time and burst time for each process and time quantum. The output should give the waiting time for each process. Also display the average waiting time.

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

#define MAX_PROCESSES 10

// Structure to represent a process
struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int start_time;
    int completion_time;
    int waiting_time;
    int turnaround_time;
};

// Function to calculate waiting time and turnaround time
void calculateWaitingAndTurnaroundTimes(struct Process processes[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Calculate waiting time and turnaround time for each process
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;

        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    // Display the results
    printf("\nProcess ID | Arrival Time | Burst Time | Waiting Time | Turnaround Time\n");
    for (int i = 0; i < n; i++) {
                    printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n",   processes[i].id,   processes[i].arrival_time,
processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time);
    }

    // Display average waiting time and average turnaround time
    printf("\nAverage Waiting Time: %.2f", (float)total_waiting_time / n);
    printf("\nAverage Turnaround Time: %.2f", (float)total_turnaround_time / n);
}

// Function to implement Round Robin Scheduling
void roundRobin(struct Process processes[], int n, int time_quantum) {
    int time = 0;
    int completed = 0;
    int queue[MAX_PROCESSES];
    int front = 0, rear = 0;

    // Initialize queue
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time <= time) {
            queue[rear++] = i;  // Add process to queue if it's arrived
        }
    }

    // Main loop for Round Robin scheduling
    while (completed < n) {
        int idx = queue[front++];  // Get the process at the front of the queue
        if (processes[idx].remaining_time > 0) {
            // Process execution for time quantum
            if (processes[idx].remaining_time <= time_quantum) {
                time += processes[idx].remaining_time;
                processes[idx].remaining_time = 0;
                processes[idx].completion_time = time;
                completed++;
            } else {
                processes[idx].remaining_time -= time_quantum;
                time += time_quantum;
            }

            // Add process back to queue if it's not completed
            if (processes[idx].remaining_time > 0) {
                queue[rear++] = idx;
            }
        }

        // Add newly arrived processes to the queue
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= time && processes[i].remaining_time > 0) {
                int alreadyInQueue = 0;
                for (int j = front; j < rear; j++) {
                    if (queue[j] == i) {
                        alreadyInQueue = 1;
                        break;
                    }
                }
                if (!alreadyInQueue) {
                    queue[rear++] = i;
                }
            }
        }
    }
}

int main() {
    int n, time_quantum;

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

    struct Process processes[MAX_PROCESSES];

    // Input arrival time, burst time for each process
    for (int i = 0; i < n; i++) {
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        processes[i].id = i + 1;
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }

    // Input time quantum
    printf("\nEnter the time quantum: ");
    scanf("%d", &time_quantum);

    // Initialize start time to -1 for all processes
    for (int i = 0; i < n; i++) {
        processes[i].start_time = -1;
    }

    // Call Round Robin scheduling
    roundRobin(processes, n, time_quantum);

    // Calculate and display waiting time and turnaround time
    calculateWaitingAndTurnaroundTimes(processes, n);

    return 0;
}
```

Q18.Write a program for Round Robin scheduling for given time quantum. Accept no. of processes, arrival time and burst time for each process and time quantum. Calculate the turn around time for each process. Display the average turn around time.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Structure to represent a process
struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int start_time;
    int completion_time;
    int turnaround_time;
};

// Function to calculate turnaround time
void calculateTurnaroundTime(struct Process processes[], int n) {
    int total_turnaround_time = 0;

    // Calculate turnaround time for each process
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    // Display the results
    printf("\nProcess ID | Arrival Time | Burst Time | Turnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival_time,
processes[i].burst_time, processes[i].turnaround_time);
    }

    // Display average turnaround time
    printf("\nAverage Turnaround Time: %.2f", (float)total_turnaround_time / n);
}

// Function to implement Round Robin Scheduling
void roundRobin(struct Process processes[], int n, int time_quantum) {
    int time = 0;
    int completed = 0;
    int queue[MAX_PROCESSES];
    int front = 0, rear = 0;

    // Initialize queue
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time <= time) {
            queue[rear++] = i;  // Add process to queue if it's arrived
        }
    }

    // Main loop for Round Robin scheduling
    while (completed < n) {
        int idx = queue[front++];  // Get the process at the front of the queue
        if (processes[idx].remaining_time > 0) {
            // Process execution for time quantum
            if (processes[idx].remaining_time <= time_quantum) {
                time += processes[idx].remaining_time;
                processes[idx].remaining_time = 0;
                processes[idx].completion_time = time;
                completed++;
            } else {
                processes[idx].remaining_time -= time_quantum;
                time += time_quantum;
            }

            // Add process back to queue if it's not completed
            if (processes[idx].remaining_time > 0) {
                queue[rear++] = idx;
            }
        }

        // Add newly arrived processes to the queue
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= time && processes[i].remaining_time > 0) {
                int alreadyInQueue = 0;
                for (int j = front; j < rear; j++) {
                    if (queue[j] == i) {
                        alreadyInQueue = 1;
                        break;
                    }
                }
                if (!alreadyInQueue) {
                    queue[rear++] = i;
                }
            }
        }
    }
```

```
 }

 int main() {
    int n, time_quantum;

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

    struct Process processes[MAX_PROCESSES];

    // Input the arrival time and burst time for each process
    for (int i = 0; i < n; i++) {
       printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
       processes[i].id = i + 1;
       printf("Arrival Time: ");
       scanf("%d", &processes[i].arrival_time);
       printf("Burst Time: ");
       scanf("%d", &processes[i].burst_time);
         processes[i].remaining_time = processes[i].burst_time;  // Initial remaining time is the burst
 time
    }

    // Input the time quantum
    printf("\nEnter the time quantum: ");
    scanf("%d", &time_quantum);

    // Initialize start time to -1 for all processes (not yet started)
    for (int i = 0; i < n; i++) {
       processes[i].start_time = -1;
    }

    // Call Round Robin scheduling
    roundRobin(processes, n, time_quantum);

    // Calculate and display turnaround time
    calculateTurnaroundTime(processes, n);

    return 0;
 }
```

Q19.Write a program to simulate FCFS CPU-scheduling. Accept no. of Processes, arrival time and burst time from user. The output should give Gantt chart.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Structure to represent a process
struct Process {
   int id;
   int arrival_time;
   int burst_time;
```

```
    int start_time;
    int completion_time;
    int waiting_time;
    int turnaround_time;
};

// Function to calculate waiting time
void calculateWaitingTime(struct Process processes[], int n) {
    processes[0].waiting_time = 0;

    for (int i = 1; i < n; i++) {
        processes[i].waiting_time = processes[i - 1].completion_time - processes[i].arrival_time;
        if (processes[i].waiting_time < 0)
            processes[i].waiting_time = 0;
    }
}

// Function to calculate turnaround time
void calculateTurnaroundTime(struct Process processes[], int n) {
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].burst_time + processes[i].waiting_time;
    }
}

// Function to display the Gantt chart
void displayGanttChart(struct Process processes[], int n) {
    printf("\nGantt Chart:\n");
    printf("------------------------------------------------\n");

    // Display the process sequence in Gantt chart
    for (int i = 0; i < n; i++) {
        printf("| P%d ", processes[i].id);
    }
    printf("|\n");

    // Display the time axis
    printf("0    ");
    for (int i = 0; i < n; i++) {
        printf("%d    ", processes[i].completion_time);
    }
    printf("\n");
}

// Function to implement FCFS Scheduling
void fcfsScheduling(struct Process processes[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Sort the processes by their arrival time
    struct Process temp;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].arrival_time > processes[j].arrival_time) {
                temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    // Calculate start time, completion time, waiting time, and turnaround time
    processes[0].start_time = processes[0].arrival_time;
    processes[0].completion_time = processes[0].start_time + processes[0].burst_time;

    for (int i = 1; i < n; i++) {
        processes[i].start_time = processes[i - 1].completion_time;
        processes[i].completion_time = processes[i].start_time + processes[i].burst_time;
    }

    calculateWaitingTime(processes, n);
    calculateTurnaroundTime(processes, n);

    // Calculate total waiting time and total turnaround time
    for (int i = 0; i < n; i++) {
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    // Display the results
     printf("\nProcess ID | Arrival Time | Burst Time | Waiting Time | Turnaround Time | Completion
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", processes[i].id, processes[i].arrival_time,
               processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time,
               processes[i].completion_time);
    }

    // Display average waiting time and average turnaround time
    printf("\nAverage Waiting Time: %.2f", (float)total_waiting_time / n);
    printf("\nAverage Turnaround Time: %.2f", (float)total_turnaround_time / n);

    // Display Gantt chart
    displayGanttChart(processes, n);
}

int main() {
    int n;

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

    struct Process processes[MAX_PROCESSES];

    // Input the arrival time and burst time for each process
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
    }

    // Call FCFS scheduling
    fcfsScheduling(processes, n);

    return 0;
}
```

Q20.Write a program for Round Robin scheduling for given time quantum. Accept no. of processes, arrival time and burst time for each process and time quantum from user. Display the content of Gantt Chart.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Structure to represent a process
struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
    int start_time;
    int completion_time;
};

// Function to calculate waiting time
void calculateWaitingTime(struct Process processes[], int n) {
    int total_time = 0;
    int completed = 0;
    int current_time = 0;
    int queue[MAX_PROCESSES];
    int front = 0, rear = 0;

    // Initialize queue
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival_time <= current_time) {
            queue[rear++] = i;
        }
    }

    while (completed < n) {
        if (front < rear) {
            int index = queue[front++];
            if (processes[index].remaining_time > 0) {
                                        int time_slice = (processes[index].remaining_time < time_quantum) ?
processes[index].remaining_time : time_quantum;
                current_time += time_slice;
                processes[index].remaining_time -= time_slice;

                if (processes[index].remaining_time == 0) {
                    processes[index].completion_time = current_time;
                    processes[index].turnaround_time = processes[index].completion_time -
processes[index].arrival_time;
                    processes[index].waiting_time = processes[index].turnaround_time -
processes[index].burst_time;
                    completed++;
                }
            }

            for (int i = 0; i < n; i++) {
                if (processes[i].arrival_time <= current_time && processes[i].remaining_time > 0) {
                    queue[rear++] = i;
                }
            }
        }
    }
}

// Function to display the Gantt Chart
void displayGanttChart(struct Process processes[], int n) {
    printf("\nGantt Chart:\n");
    printf("------------------------------------------------\n");

    // Display the process sequence in Gantt chart
    for (int i = 0; i < n; i++) {
        printf("| P%d ", processes[i].id);
    }
    printf("|\n");

    // Display the time axis
    printf("0    ");
    for (int i = 0; i < n; i++) {
        printf("%d    ", processes[i].completion_time);
    }
    printf("\n");
}

// Round Robin scheduling function
void roundRobinScheduling(struct Process processes[], int n, int time_quantum) {
    int total_waiting_time = 0, total_turnaround_time = 0;

    // Initialize the remaining time for each process
    for (int i = 0; i < n; i++) {
        processes[i].remaining_time = processes[i].burst_time;
    }

    calculateWaitingTime(processes, n);

    // Calculate total waiting time and total turnaround time
    for (int i = 0; i < n; i++) {
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    // Display the results
     printf("\nProcess ID | Arrival Time | Burst Time | Waiting Time | Turnaround Time | Completion
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", processes[i].id, processes[i].arrival_time,
               processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time,
               processes[i].completion_time);
    }

    // Display average waiting time and average turnaround time
    printf("\nAverage Waiting Time: %.2f", (float)total_waiting_time / n);
    printf("\nAverage Turnaround Time: %.2f", (float)total_turnaround_time / n);

    // Display Gantt chart
    displayGanttChart(processes, n);
}

int main() {
    int n, time_quantum;

    // Input the number of processes and time quantum
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the time quantum: ");
    scanf("%d", &time_quantum);

    struct Process processes[MAX_PROCESSES];

    // Input the arrival time and burst time for each process
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
    }

    // Call Round Robin scheduling
    roundRobinScheduling(processes, n, time_quantum);

    return 0;
}
```

Q21.Write a program to simulate Pre-emptive Shortest Job First (SJF) – scheduling. Accept no. of processes, arrival time and burst time from user. Display the content of Gantt Chart.

```
#include <stdio.h>

#define MAX_PROCESSES 10
```

```
// Structure to represent a process
struct Process {
   int id;              // Process ID
   int arrival_time;    // Arrival time of the process
   int burst_time;      // Initial burst time
   int remaining_time;  // Remaining burst time (for preemptive scheduling)
   int completion_time; // Completion time of the process
   int start_time;      // Start time of the process
   int turnaround_time; // Turnaround time of the process
   int waiting_time;    // Waiting time of the process
};

// Function to calculate the Gantt chart
void calculateGanttChart(struct Process processes[], int n) {
   int current_time = 0, completed = 0;
   int min_remaining_time = 9999;
   int shortest = -1;
    int is_completed[MAX_PROCESSES] = {0}; // Keeps track of whether a process is completed
or not

   printf("\nGantt Chart: \n");

   while (completed < n) {
      // Find the process with the shortest remaining time
      for (int i = 0; i < n; i++) {
         if (processes[i].arrival_time <= current_time && !is_completed[i] &&
               processes[i].remaining_time < min_remaining_time && processes[i].remaining_time >
0) {
            min_remaining_time = processes[i].remaining_time;
            shortest = i;
         }
      }

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

      // Execute the selected process for 1 unit of time
      processes[shortest].remaining_time--;
      printf("P%d ", processes[shortest].id);

      // If the process is completed, calculate its turnaround and waiting time
      if (processes[shortest].remaining_time == 0) {
         is_completed[shortest] = 1;
         completed++;
         processes[shortest].completion_time = current_time + 1;
                   processes[shortest].turnaround_time = processes[shortest].completion_time -
processes[shortest].arrival_time;
                      processes[shortest].waiting_time = processes[shortest].turnaround_time -
processes[shortest].burst_time;
      }
```

```
        min_remaining_time = 9999;
        current_time++;
    }

    printf("\n");
}

// Function to display the results of the scheduling
void displayResults(struct Process processes[], int n) {
    int total_turnaround_time = 0, total_waiting_time = 0;

    printf("\nProcess ID | Arrival Time | Burst Time | Waiting Time | Turnaround Time | Completion
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", processes[i].id, processes[i].arrival_time,
               processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time,
               processes[i].completion_time);
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }

    // Display average waiting time and average turnaround time
    printf("\nAverage Waiting Time: %.2f", (float)total_waiting_time / n);
    printf("\nAverage Turnaround Time: %.2f", (float)total_turnaround_time / n);
}

int main() {
    int n;

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

    struct Process processes[MAX_PROCESSES];

    // Input arrival time and burst time for each process
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
            processes[i].remaining_time = processes[i].burst_time; // Initialize remaining time as burst
time
    }

    // Call Preemptive Shortest Job First scheduling
    calculateGanttChart(processes, n);

    // Display the results
    displayResults(processes, n);

    return 0;
```

```
}
```

Q22.Write a program to simulate Non-pre-emptive Shortest Job First (SJF) CPU–scheduling.Accept no. of processes, arrival time and burst time from user. Display the content of Gantt Chart.

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

#define MAX_PROCESSES 10

// Structure to represent a process
struct Process {
    int id;             // Process ID
    int arrival_time;     // Arrival time of the process
    int burst_time;       // Burst time of the process
    int completion_time;  // Completion time of the process
    int waiting_time;     // Waiting time of the process
    int turnaround_time;  // Turnaround time of the process
};

// Function to calculate the Gantt chart for Non-preemptive SJF
void calculateGanttChart(struct Process processes[], int n) {
    int current_time = 0;
    int completed = 0;
    bool is_completed[MAX_PROCESSES] = {0}; // To keep track of completed processes

    // Process the Gantt chart
    printf("\nGantt Chart: \n");

    // While there are uncompleted processes
    while (completed < n) {
        int shortest = -1;
        int min_burst_time = 9999;

        // Find the process with the smallest burst time that is ready to execute
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time <= current_time && !is_completed[i]) {
                if (processes[i].burst_time < min_burst_time) {
                    min_burst_time = processes[i].burst_time;
                    shortest = i;
                }
            }
        }

        // If no process is ready to execute, increment the current time
        if (shortest == -1) {
            current_time++;
            continue;
        }

        // Execute the selected process (update its completion time)
        is_completed[shortest] = 1;
        processes[shortest].completion_time = current_time + processes[shortest].burst_time;
        current_time += processes[shortest].burst_time;

        // Calculate the turnaround and waiting times for the selected process
                      processes[shortest].turnaround_time = processes[shortest].completion_time -
processes[shortest].arrival_time;
                        processes[shortest].waiting_time = processes[shortest].turnaround_time -
processes[shortest].burst_time;

        // Print the process ID for the Gantt chart
        printf("P%d ", processes[shortest].id);
        completed++;
    }
    printf("\n");
}

// Function to display the results of the scheduling
void displayResults(struct Process processes[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;

     printf("\nProcess ID | Arrival Time | Burst Time | Waiting Time | Turnaround Time | Completion
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", processes[i].id, processes[i].arrival_time,
               processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time,
               processes[i].completion_time);
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    // Calculate average waiting time and average turnaround time
    printf("\nAverage Waiting Time: %.2f", (float)total_waiting_time / n);
    printf("\nAverage Turnaround Time: %.2f", (float)total_turnaround_time / n);
}

int main() {
    int n;

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

    struct Process processes[MAX_PROCESSES];

    // Input arrival time and burst time for each process
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
    }
```

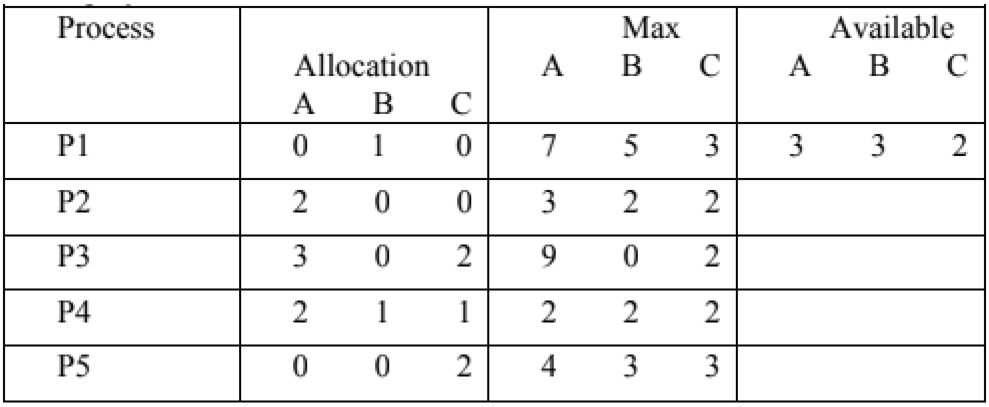
Q23.Write a program for Round Robin scheduling for given time quantum. Accept no. of processes, arrival time and burst time for each process and time quantum Display the content of Gantt Chart.

```
#include <stdio.h>

#define MAX_PROCESSES 10

// Structure to represent a process
struct Process {
    int id;              // Process ID
    int arrival_time;     // Arrival time of the process
    int burst_time;       // Burst time of the process
    int remaining_time;   // Remaining burst time
    int completion_time;  // Completion time of the process
    int waiting_time;     // Waiting time of the process
    int turnaround_time;  // Turnaround time of the process
};

// Function to calculate the Gantt chart for Round Robin scheduling
void calculateGanttChart(struct Process processes[], int n, int time_quantum) {
    int current_time = 0;
    int completed = 0;
    int i;

    // Gantt chart display
    printf("\nGantt Chart: \n");

    // Continue until all processes are completed
    while (completed < n) {
        for (i = 0; i < n; i++) {
            // If the process has arrived and is not yet completed
            if (processes[i].arrival_time <= current_time && processes[i].remaining_time > 0) {
                // If burst time is greater than time quantum, process will run for time quantum
                if (processes[i].remaining_time > time_quantum) {
                    current_time += time_quantum;
                    processes[i].remaining_time -= time_quantum;
                }
                // If burst time is less than or equal to time quantum, process will run to completion
                else {
                    current_time += processes[i].remaining_time;
                    processes[i].remaining_time = 0;
                    processes[i].completion_time = current_time;
                                    processes[i].turnaround_time = processes[i].completion_time -
processes[i].arrival_time;
                    processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
                    completed++;
                }

                // Display the process id for the Gantt chart
                printf("P%d ", processes[i].id);
            }
        }
    }

    printf("\n");
}

// Function to display the results of the scheduling
void displayResults(struct Process processes[], int n) {
    int total_waiting_time = 0, total_turnaround_time = 0;

     printf("\nProcess ID | Arrival Time | Burst Time | Waiting Time | Turnaround Time | Completion
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", processes[i].id, processes[i].arrival_time,
               processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time,
               processes[i].completion_time);
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    // Calculate average waiting time and average turnaround time
    printf("\nAverage Waiting Time: %.2f", (float)total_waiting_time / n);
    printf("\nAverage Turnaround Time: %.2f", (float)total_turnaround_time / n);
}

int main() {
    int n, time_quantum;

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

    struct Process processes[MAX_PROCESSES];

    // Input arrival time and burst time for each process
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("\nEnter arrival time and burst time for Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time; // Initialize remaining time
    }

    // Input the time quantum
    printf("\nEnter the time quantum: ");
    scanf("%d", &time_quantum);

    // Call Round Robin scheduling to calculate the Gantt Chart
    calculateGanttChart(processes, n, time_quantum);

    // Display the results
    displayResults(processes, n);

    return 0;
}
```

Q24.Write a program to implement Bankers algorithm. Mention no. of processes and available resources. Calculate need matrix based on max and allocation matrix.

```
#include <stdio.h>

#define MAX 10
#define RESOURCE_TYPES 3

// Function to calculate the Need matrix
void calculateNeed(int need[MAX][RESOURCE_TYPES], int max[MAX][RESOURCE_TYPES],
int allocation[MAX][RESOURCE_TYPES], int n, int m) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

// Function to display the matrix
void displayMatrix(int matrix[MAX][RESOURCE_TYPES], int n, int m, char *title) {
    printf("%s\n", title);
    printf("Process | ");
    for (int i = 0; i < m; i++) {
        printf("R%d ", i + 1);
    }
    printf("\n");
    for (int i = 0; i < n; i++) {
        printf("P%d     | ", i + 1);
        for (int j = 0; j < m; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int n, m;  // n = number of processes, m = number of resource types
                  int     allocation[MAX][RESOURCE_TYPES],     max[MAX][RESOURCE_TYPES],
need[MAX][RESOURCE_TYPES];
    int available[RESOURCE_TYPES];

    // Input number of processes and resource types
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the number of resource types: ");
    scanf("%d", &m);

    // Input available resources
    printf("\nEnter the available instances of each resource:\n");
    for (int i = 0; i < m; i++) {
        printf("R%d: ", i + 1);
        scanf("%d", &available[i]);
    }

    // Input the Allocation Matrix
    printf("\nEnter the Allocation Matrix:\n");
    for (int i = 0; i < n; i++) {
        printf("For Process P%d:\n", i + 1);
        for (int j = 0; j < m; j++) {
            printf("R%d: ", j + 1);
            scanf("%d", &allocation[i][j]);
        }
    }

    // Input the Max Matrix
    printf("\nEnter the Max Matrix:\n");
    for (int i = 0; i < n; i++) {
        printf("For Process P%d:\n", i + 1);
        for (int j = 0; j < m; j++) {
            printf("R%d: ", j + 1);
            scanf("%d", &max[i][j]);
        }
    }

    // Calculate the Need Matrix
    calculateNeed(need, max, allocation, n, m);

    // Display the matrices
    displayMatrix(allocation, n, m, "Allocation Matrix:");
    displayMatrix(max, n, m, "Max Matrix:");
    displayMatrix(need, n, m, "Need Matrix:");

    return 0;
}
```

Q25.Write a Program to implement following functionality
Accept Available
Display Allocation, Max
Display the contents of need matrix



Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

```
// 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], 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);

  }
```