1. FCFS (with AT):

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

int main() {
  int n;

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

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

  float totalTAT = 0, totalWT = 0;

  for (int i = 0; i < n; i++) {
    process_id[i] = i + 1; // Assign process ID (starting from 1)
    printf("Arrival Time for process %d: ", (i + 1));
    scanf("%d", &at[i]); // Input arrival time
    printf("Burst Time for process %d: ", (i + 1));
    scanf("%d", &bt[i]); // Input burst time
  }

  // Sort processes based on arrival time using Bubble Sort
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
      if (at[i] > at[j]) {
        // Swap arrival times
        int temp = at[i];
        at[i] = at[j];
        at[j] = temp;
        // Swap burst times
        temp = bt[i];
        bt[i] = bt[j];
        bt[j] = temp;
        // Swap process IDs
        temp = process_id[i];
        process_id[i] = process_id[j];
        process_id[j] = temp;
      }
    }
  }

  int timePassed = 0; // Variable to track the total time passed
```

```
    // Calculate completion time for each process
    for (int i = 0; i < n; i++) {
        if (at[i] > timePassed) {
            timePassed = at[i]; // If the process arrives after the current time, update timePassed to the
arrival time
        }
        timePassed += bt[i]; // Add burst time to timePassed
        ct[i] = timePassed; // Completion time is the updated timePassed
    }

    // Calculate turnaround time and waiting time for each process
    for (int i = 0; i < n; i++) {
        tat[i] = ct[i] - at[i]; // Turnaround time = Completion time - Arrival time
        wt[i] = tat[i] - bt[i]; // Waiting time = Turnaround time - Burst time
        totalTAT += tat[i]; // Accumulate total turnaround time
        totalWT += wt[i]; // Accumulate total waiting time
    }

    printf("\nPID\tAT\tBT\tCT\tTAT\tWT");
    for (int i = 0; i < n; i++) {
        printf("\n%d\t%d\t%d\t%d\t%d\t%d", process_id[i], at[i], bt[i], ct[i], tat[i], wt[i]);
    }

    printf("\n\nAverage Turnaround Time: %.2f", totalTAT / n);
    printf("\nAverage Waiting Time: %.2f", totalWT / n);

    return 0;
}
```

2. **FCFS (without AT):**

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

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

    int process_id[n], bt[n], ct[n], tat[n], wt[n];
    float totalTAT = 0, totalWT = 0;

    for (int i = 0; i < n; i++) {
        process_id[i] = i + 1; // Assign process ID (starting from 1)
        printf("Burst Time for %d: ", (i + 1));
```

```c
        scanf("%d", &bt[i]); // Input burst time
    }

    int timePassed = 0; // Variable to track the total time passed

    // Calculate completion time for each process
    for (int i = 0; i < n; i++) {
        timePassed += bt[i]; // Add burst time to timePassed
        ct[i] = timePassed; // Completion time is the updated timePassed
    }

    // Calculate turnaround time and waiting time for each process
    for (int i = 0; i < n; i++) {
        tat[i] = ct[i]; // Turnaround time = Completion time (since arrival time is zero in this scenario)
        wt[i] = tat[i] - bt[i]; // Waiting time = Turnaround time - Burst time
        totalTAT += tat[i]; // Accumulate total turnaround time
        totalWT += wt[i]; // Accumulate total waiting time
    }

    // Print the results in tabular format
    printf("\nPID\tBT\tCT\tTAT\tWT");
    for (int i = 0; i < n; i++) {
        printf("\n%d\t%d\t%d\t%d\t%d", process_id[i], bt[i], ct[i], tat[i], wt[i]);
    }

    // Print the average turnaround time and average waiting time
    printf("\n\nAverage Turnaround Time: %.2f", totalTAT / n);
    printf("\nAverage Waiting Time: %.2f", totalWT / n);

    return 0;
}
```

3. **SJF:**

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

typedef struct {
    char process_name;       // Process name
    int arrival_time;        // Arrival time
    int burst_time;          // Burst time
    int completion_time;     // Completion time
    int turnaround_time;     // Turnaround time
    int waiting_time;        // Waiting time
    int response_time;       // Response time
```

```
} Process;

// Function to sort processes by burst time using Bubble Sort
void sort_by_burst_time(Process *processes, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            // Compare burst times of consecutive processes
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                // Swap if the current process has a longer burst time
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}

// Function to compute completion times of processes
void compute_completion_time(Process *processes, int n) {
    int current_time = 0; // Variable to keep track of the current time
    int index = 0;       // Index to iterate through processes

    // Loop until all processes are completed
    while (index < n) {
        int next_process = -1;

        // Find the next process to execute
        for (int i = 0; i < n; i++) {
            // Check if the process has arrived and is not yet completed
            if (processes[i].arrival_time <= current_time && processes[i].completion_time == 0) {
                // Choose the process with the shortest burst time
                if (next_process == -1 || processes[i].burst_time < processes[next_process].burst_time) {
                    next_process = i;
                }
            }
        }

        // If no process is ready, move to the next arrival time
        if (next_process == -1) {
            current_time = processes[index].arrival_time;
        } else {
            // Calculate completion time for the selected process
                                                processes[next_process].completion_time    =    current_time    +
processes[next_process].burst_time;
            current_time = processes[next_process].completion_time;
```

```
        }
        index++;
    }
}

// Function to compute turnaround and waiting times for each process
void compute_turnaround_waiting_time(Process *processes, int n) {
    for (int i = 0; i < n; i++) {
        // Turnaround time = Completion time - Arrival time
        processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
        // Waiting time = Turnaround time - Burst time
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
        // Response time = Waiting time (as it's a non-preemptive scheduling)
        processes[i].response_time = processes[i].waiting_time;
    }
}

// Function to display the process table with all times and average times
void display_table(Process *processes, int n) {
    int total_tat = 0, total_wt = 0, total_rt = 0;

    printf("Process\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf(" %c\t%d\t%d\t%d\t%d\t%d\t%d\n",
               processes[i].process_name,
               processes[i].arrival_time,
               processes[i].burst_time,
               processes[i].completion_time,
               processes[i].turnaround_time,
               processes[i].waiting_time,
               processes[i].response_time);

        total_tat += processes[i].turnaround_time;
        total_wt += processes[i].waiting_time;
        total_rt += processes[i].response_time;
    }

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

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

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

    // Dynamically allocate memory for processes
    Process *processes = (Process *)malloc(n * sizeof(Process));

    for (int i = 0; i < n; i++) {
        printf("Enter details for process %d (Name Arrival Burst): ", i + 1);
                      scanf(" %c %d %d", &processes[i].process_name, &processes[i].arrival_time,
&processes[i].burst_time);
        processes[i].completion_time = 0;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
        processes[i].response_time = 0;
    }

    sort_by_burst_time(processes, n);
    compute_completion_time(processes, n);
    compute_turnaround_waiting_time(processes, n);
    display_table(processes, n);
    free(processes);
    return 0;
}
```

4. **SRTF (SJF Preemptive):**

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

struct Process {
    int pid;              // Process ID
    int at;               // Arrival time
    int bt;               // Burst time
    int rem;              // Remaining burst time
    int ct;               // Completion time
    int tat;              // Turnaround time
    int wt;               // Waiting time
    int rt;               // Response time
};

// Function to find the index of the shortest job based on remaining time
int findShortestJob(struct Process processes[], int n, int current_time) {
    int sj_index = -1;
    int sj = INT_MAX;
    for (int i = 0; i < n; i++) {
```

```
        // Check if process has arrived, has remaining burst time, and has the shortest remaining burst
time
      if (processes[i].at <= current_time && processes[i].rem > 0 &&
        processes[i].rem < sj) {
        sj_index = i;
        sj = processes[i].rem;
      }
  }
  return sj_index;  // Return index of the shortest job found
}

// Function to perform Shortest Job First scheduling
void SJF(struct Process processes[], int n) {
  int current_time = 0;   // Initialize current time
  int completed = 0;      // Initialize completed processes counter

  // Process until all processes are completed
  while (completed < n) {
    // Find the index of the shortest job
    int sj_index = findShortestJob(processes, n, current_time);

    // If no job is found, move to the next time unit
    if (sj_index == -1) {
      current_time++;
    } else {
      // Record response time when the process starts execution
      if (processes[sj_index].rt == -1) {
        processes[sj_index].rt = current_time - processes[sj_index].at;
      }

      // Execute the shortest job for one time unit
      processes[sj_index].rem--;
      current_time++;

      // If the job is completed
      if (processes[sj_index].rem == 0) {
        // Record completion time, turnaround time, and waiting time
        processes[sj_index].ct = current_time;
        processes[sj_index].tat = processes[sj_index].ct - processes[sj_index].at;
        processes[sj_index].wt = processes[sj_index].tat - processes[sj_index].bt;
        completed++;  // Increment completed processes counter
      }
    }
  }
}
```

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

    struct Process processes[n];

    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].at);
        printf("Burst Time: ");
        scanf("%d", &processes[i].bt);
        processes[i].rem = processes[i].bt;  // Initialize remaining time
        processes[i].pid = i + 1;           // Assign process ID
        processes[i].rt = -1;               // Initialize response time
    }

    SJF(processes, n);

    int total_tat = 0, total_wt = 0, total_rt = 0;
    printf("\nP\tAT\tBT\tCT\tWT\tTAT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].at,
               processes[i].bt, processes[i].ct, processes[i].wt,
               processes[i].tat, processes[i].rt);
        // Accumulate total times
        total_tat += processes[i].tat;
        total_wt += processes[i].wt;
        total_rt += processes[i].rt;
    }

    printf("\nAverage Turnaround Time: %.2f\n", (float)total_tat / n);
    printf("Average Waiting Time: %.2f\n", (float)total_wt / n);
    printf("Average Response Time: %.2f\n", (float)total_rt / n);

    return 0;
}
```

5. **Priority (Non-preemptive):**

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

```
#define MAX_PROCESSES 10

// Define the Process structure with necessary attributes
struct Process {
   int pid;      // Process ID
   int at;       // Arrival time
   int bt;       // Burst time
   int priority;  // Priority of the process
   int tat;      // Turnaround time
   int wt;       // Waiting time
   int ct;       // Completion time
   int rt;       // Response time
};

// Function to perform Priority Non-preemptive Scheduling
void priority_nonpreemptive(struct Process processes[], int n) {
   // Sort processes by priority for non-preemptive scheduling
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (processes[j].priority > processes[j + 1].priority) {
            // Swap processes based on priority
            struct Process temp = processes[j];
            processes[j] = processes[j + 1];
            processes[j + 1] = temp;
         }
      }
   }

   // Initialize total time and variables for calculating averages
   int total_time = 0;
   double total_tat = 0;
   double total_wt = 0;
   double total_rt = 0;

   // Calculate completion time, turnaround time, waiting time, and response time
   for (int i = 0; i < n; i++) {
      processes[i].ct = total_time + processes[i].bt;  // Completion time
      processes[i].tat = processes[i].ct - processes[i].at;  // Turnaround time
      processes[i].wt = processes[i].tat - processes[i].bt;  // Waiting time
      processes[i].rt = total_time - processes[i].at;  // Response time

      total_time = processes[i].ct;  // Update total time to current process completion time

      // Accumulate totals for calculating averages
```

```
        total_tat += processes[i].tat;
        total_wt += processes[i].wt;
        total_rt += processes[i].rt;
    }

    // Print results
    printf("Process\tPriority\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].priority,
               processes[i].at, processes[i].bt, processes[i].ct,
               processes[i].tat, processes[i].wt, processes[i].rt);
    }

    // Print average turnaround time, average waiting time, and average response time
    printf("\nAverage Turnaround Time: %.2f\n", total_tat / n);
    printf("Average Waiting Time: %.2f\n", total_wt / n);
    printf("Average Response Time: %.2f\n", total_rt / n);
}

// Main function to input processes and call scheduling function
int main() {
    int n;
    struct Process processes[MAX_PROCESSES];

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

    // Input details for each process
    for (int i = 0; i < n; i++) {
        printf("Process %d\n", i + 1);
        printf("Enter arrival time: ");
        scanf("%d", &processes[i].at);
        printf("Enter burst time: ");
        scanf("%d", &processes[i].bt);
        printf("Enter priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].tat = 0;
        processes[i].wt = 0;
        processes[i].ct = 0;
        processes[i].rt = 0;
    }

    // Perform Priority Non-preemptive Scheduling
```

```
    printf("\nPriority Non-preemptive Scheduling:\n");
    priority_nonpreemptive(processes, n);

    return 0;
}
```

6. **Priority (Preemptive):**

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

#define MAX_PROCESSES 10

struct Process {
    int pid;            // Process ID
    int at;     // Arrival time
    int bt;      // Burst time
    int priority;       // Priority of the process
    int rem;    // Remaining time to complete execution
    int tat;   // Turnaround time
    int wt;     // Waiting time
    int ct;   // Completion time
    int rt;     // Response time
};

// Function to perform Priority Preemptive Scheduling
void priority_preemptive(struct Process processes[], int n) {
    int total_time = 0;
    int completed = 0;

    while (completed < n) {
        int highest_priority = -1;
        int next_process = -1;

        // Find the process with the highest priority that has arrived and still needs processing
        for (int i = 0; i < n; i++) {
            if (processes[i].at <= total_time && processes[i].rem > 0) {
                if (highest_priority == -1 || processes[i].priority < highest_priority) {
                    highest_priority = processes[i].priority;
                    next_process = i;
                }
            }
        }
        // If no process is ready to execute, move time forward
        if (next_process == -1) {
```

```
            total_time++;
            continue;
        }
        // Calculate response time if the process is starting execution for the first time
        if (processes[next_process].rem == processes[next_process].bt) {
            processes[next_process].rt = total_time - processes[next_process].at;
        }
        // Execute the process for one unit of time
        processes[next_process].rem--;
        total_time++;

        // If process completes its execution
        if (processes[next_process].rem == 0) {
            completed++;
            processes[next_process].ct = total_time;
            processes[next_process].tat = processes[next_process].ct - processes[next_process].at;
            processes[next_process].wt = processes[next_process].tat - processes[next_process].bt;
        }
    }

    // Calculate total and average turnaround time, waiting time, and response time
    double total_tat = 0;
    double total_wt = 0;
    double total_rt = 0;

    // Print results
    printf("Process\tPriority\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].priority,
               processes[i].at, processes[i].bt, processes[i].ct,
               processes[i].tat, processes[i].wt, processes[i].rt);

        // Accumulate totals for calculating averages
        total_tat += processes[i].tat;
        total_wt += processes[i].wt;
        total_rt += processes[i].rt;
    }

    // Print average turnaround time, average waiting time, and average response time
    printf("\nAverage Turnaround Time: %.2f\n", total_tat / n);
    printf("Average Waiting Time: %.2f\n", total_wt / n);
    printf("Average Response Time: %.2f\n", total_rt / n);
}

int main() {
```

```
    int n;
    struct Process processes[MAX_PROCESSES];

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

    for (int i = 0; i < n; i++) {
        printf("Process %d\n", i + 1);
        printf("Enter arrival time: ");
        scanf("%d", &processes[i].at);
        printf("Enter burst time: ");
        scanf("%d", &processes[i].bt);
        printf("Enter priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].rem = processes[i].bt;
        processes[i].tat = 0;
        processes[i].wt = 0;
        processes[i].ct = 0;
        processes[i].rt = 0;
    }

    printf("\nPriority Preemptive Scheduling:\n");
    priority_preemptive(processes, n);

    return 0;
}
```

7. **Round Robin:**

```
#include<stdio.h>

// Function to sort processes based on arrival time
void sort(int proc_id[], int at[], int bt[], int b[], int n) {
    int min, temp;
    for (int i = 0; i < n; i++) {
        min = at[i];
        for (int j = i; j < n; j++) {
            if (at[j] < min) {
                // Swap arrival times
                temp = at[i];
                at[i] = at[j];
                at[j] = temp;
                // Swap burst times
                temp = bt[i];
```

```
            bt[i] = bt[j];
            bt[j] = temp;
            // Swap remaining burst times
            temp = b[i];
            b[i] = b[j];
            b[j] = temp;
            // Swap process IDs
            temp = proc_id[i];
            proc_id[i] = proc_id[j];
            proc_id[j] = temp;
          }
        }
    }
}

void main() {
    int n, t;   // Number of processes and time quantum
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter Time Quantum: ");
    scanf("%d", &t);

    // Arrays to store process attributes
    int proc_id[n], at[n], bt[n], ct[n], tat[n], wt[n], b[n], rt[n], m[n];
    int f = -1, r = -1;   // Front and rear indices of the queue
    int q[100];          // Queue to store process IDs
    int c = 0;           // Current time
    int count = 0;       // Count of completed processes
     double avg_tat = 0.0, avg_wt = 0.0, ttat = 0.0, twt = 0.0;  // Variables for average turnaround and
waiting times

    // Initialize process IDs
    for (int i = 0; i < n; i++)
        proc_id[i] = i + 1;

    // Input arrival times
    printf("Enter arrival times:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &at[i]);

    // Input burst times and initialize other arrays
    printf("Enter burst times:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &bt[i]);
        b[i] = bt[i];
```

```
    m[i] = 0;
    rt[i] = -1;
}

// Sort processes based on arrival time
sort(proc_id, at, bt, b, n);

// Initialize the queue
f = r = 0;
q[0] = proc_id[0];
int p = 0, i = 0;

// Process scheduling using Round Robin algorithm
while (f >= 0) {
    p = q[f++];
    i = 0;
    while (p != proc_id[i])
        i++;

    // Execute the process for time quantum or until it finishes
    if (b[i] >= t) {
        if (rt[i] == -1)
            rt[i] = c;
        b[i] -= t;
        c += t;
        m[i] = 1;
    } else {
        if (rt[i] == -1)
            rt[i] = c;
        c += b[i];
        b[i] = 0;
        m[i] = 1;
    }

    m[0] = 1;

    // Add processes to the queue that have arrived and not yet executed
    for (int j = 0; j < n; j++) {
        if (at[j] <= c && proc_id[j] != p && m[j] != 1) {
            q[++r] = proc_id[j];
            m[j] = 1;
        }
    }

    // Check if the process has completed or needs to be re-added to the queue
```

```
        if (b[i] == 0) {
            count++;
            ct[i] = c;
        } else
            q[++r] = proc_id[i];

        if (f > r)
            f = -1;
    }

    // Calculate turnaround time and waiting time for each process
    for (int i = 0; i < n; i++) {
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
    }

    // Output the results
    printf("\nRRS scheduling:\n");
    printf("PID\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], at[i], bt[i], ct[i], tat[i], wt[i], rt[i]);

    // Calculate average turnaround time and average waiting time
    for (int i = 0; i < n; i++) {
        ttat += tat[i];
        twt += wt[i];
    }
    avg_tat = ttat / (double)n;
    avg_wt = twt / (double)n;
    printf("\nAverage turnaround time: %lfms\n", avg_tat);
    printf("Average waiting time: %lfms\n", avg_wt);
}
```

8. Multilevel (FCFS both system and user):

```
#include <stdio.h>

// Function to sort processes based on arrival time using Selection Sort
void sort(int proc_id[], int at[], int bt[], int n) {
    int min, temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (at[j] < at[i]) {
                // Swap arrival times
                temp = at[i];
```

```
                at[i] = at[j];
                at[j] = temp;
                // Swap burst times
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;
                // Swap process IDs
                temp = proc_id[i];
                proc_id[i] = proc_id[j];
                proc_id[j] = temp;
            }
        }
    }
}

// Function to simulate FCFS scheduling for given processes
void simulateFCFS(int proc_id[], int at[], int bt[], int n, int start_time) {
    int c = start_time, ct[n], tat[n], wt[n];
    double ttat = 0.0, twt = 0.0;

    // Calculate completion times
    for (int i = 0; i < n; i++) {
        if (c >= at[i])
            c += bt[i];
        else
            c = at[i] + bt[i];
        ct[i] = c;
    }

    // Calculate turnaround time and waiting time
    for (int i = 0; i < n; i++) {
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
    }

    // Output the results
    printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], at[i], bt[i], ct[i], tat[i], wt[i]);
        ttat += tat[i];
        twt += wt[i];
    }

    // Calculate average turnaround time and average waiting time
    printf("Average Turnaround Time: %.2lf ms\n", ttat / n);
```

```c
    printf("Average Waiting Time: %.2lf ms\n", twt / n);
}

void main() {
    int n;

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

    int proc_id[n], at[n], bt[n], type[n];
    int sys_proc_id[n], sys_at[n], sys_bt[n], user_proc_id[n], user_at[n], user_bt[n];
    int sys_count = 0, user_count = 0;

    // Input arrival time, burst time, and type for each process
    for (int i = 0; i < n; i++) {
        proc_id[i] = i + 1;
        printf("Enter arrival time, burst time and type (0 for system, 1 for user) for process %d: ", i + 1);
        scanf("%d %d %d", &at[i], &bt[i], &type[i]);

        // Separate processes into system and user based on type
        if (type[i] == 0) {
            sys_proc_id[sys_count] = proc_id[i];
            sys_at[sys_count] = at[i];
            sys_bt[sys_count] = bt[i];
            sys_count++;
        } else {
            user_proc_id[user_count] = proc_id[i];
            user_at[user_count] = at[i];
            user_bt[user_count] = bt[i];
            user_count++;
        }
    }

    // Sort system processes by arrival time
    sort(sys_proc_id, sys_at, sys_bt, sys_count);

    // Sort user processes by arrival time
    sort(user_proc_id, user_at, user_bt, user_count);

    // Perform FCFS scheduling for system processes
    printf("System Processes Scheduling:\n");
    simulateFCFS(sys_proc_id, sys_at, sys_bt, sys_count, 0);

    // Determine the end time of system processes
```

```
    int system_end_time = 0;
    if (sys_count > 0) {
        system_end_time = sys_at[sys_count - 1] + sys_bt[sys_count - 1];
        for (int i = 0; i < sys_count - 1; i++) {
            if (sys_at[i + 1] > system_end_time) {
                system_end_time = sys_at[i + 1];
            }
            system_end_time += sys_bt[i];
        }
    }

    // Perform FCFS scheduling for user processes
    printf("\nUser Processes Scheduling:\n");
    simulateFCFS(user_proc_id, user_at, user_bt, user_count, system_end_time);
}
```

9. Multilevel (P1 RR and P2 FCFS):

Didn’t get :))

10. Rate Monotonic CPU Scheduling (doesn’t handle decimal values):

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

struct P {
    int id;        // Process ID
    float et;      // Execution time
    int tp;        // Time period (periodicity)
    int v;         // Flag indicating if process is executing or not (0 or 1)
    int b;         // Backup of execution time (initial execution time)
};

// Function to find the greatest common divisor (GCD)
int gcd(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

// Function to find the least common multiple (LCM) of an array of integers
int findlcm(int arr[], int n) {
    int ans = arr[0];

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

```
        ans = (((arr[i] * ans)) / (gcd(arr[i], ans)));

    return ans;
}

void main() {
    int n, ct = 0, f = 0;
    float awt = 0, atat = 0, art = 0, tp;

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

    struct P p[n];      // Array of processes
    struct P temp;
    int a[n];           // Array to store time periods

    // Input details for each process
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        p[i].v = 0;     // Initially set all processes to not executing
        printf("Enter Execution Time and Time Period of P%d: ", i + 1);
        scanf("%f %d", &p[i].et, &p[i].tp);
        p[i].b = p[i].et;   // Backup initial execution time
        a[i] = p[i].tp;     // Store time period in an array for LCM calculation
    }

    // Sort processes based on time period (using a simple selection sort)
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].tp > p[j].tp) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }

    // Find the LCM of all time periods
    int ans = findlcm(a, n);

    // Simulate RMS scheduling for the calculated LCM time frame
    for (int i = 0; i < ans; i++) {
        f = 0;
```

```
        // Check for processes arriving at their time periods
        for (int j = 0; j < n; j++) {
            if (i % p[j].tp == 0) {
                p[j].v = 0;         // Reset flag to indicate arrival
                p[j].et = p[j].b;   // Reset execution time to initial
            }
        }

        // Execute processes according to RMS priority
        for (int j = 0; j < n; j++) {
            if (p[j].v == 0) {      // If process is ready to execute
                f = 1;
                p[j].et -= 1;       // Execute process for 1 unit of time
                printf("%d to %d P%d\n", i, i + 1, p[j].id);
                if (p[j].et == 0) {
                    p[j].v = 1;     // Set flag to indicate completion
                }
                break;
            }
        }

        // If no process is executing, print idle time
        if (f == 0) {
            printf("%d to %d -\n", i, i + 1);
        }
    }
}
```

11. Earliest Deadline First CPU Scheduling:

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

#define MAX_TSKS 10

typedef struct {
    int p;      // Period of the task
    int c;      // Execution time of the task
    int d;      // Deadline of the task
    int rt;     // Remaining execution time of the task
    int nd;     // Next deadline of the task
    int id;     // Task ID
} Task;

// Function to input task details
```

```
void Input(Task tsks[], int *n_tsk) {
   printf("Enter number of tasks (max %d): ", MAX_TSKS);
   scanf("%d", n_tsk);

   if (*n_tsk > MAX_TSKS) {
      printf("Number of tasks exceeds the maximum limit of %d.\n", MAX_TSKS);
      exit(EXIT_FAILURE);
   }

   for (int i = 0; i < *n_tsk; i++) {
      tsks[i].id = i + 1;
      printf("Enter period (p) of task %d: ", i + 1);
      scanf("%d", &tsks[i].p);
      printf("Enter execution time (c) of task %d: ", i + 1);
      scanf("%d", &tsks[i].c);
      printf("Enter deadline (d) of task %d: ", i + 1);
      scanf("%d", &tsks[i].d);

      tsks[i].rt = tsks[i].c;     // Initialize remaining time to execution time
      tsks[i].nd = tsks[i].d;     // Initialize next deadline to the specified deadline
   }
}

// Function to perform Earliest-Deadline First (EDF) scheduling
void EDF(Task tsks[], int n_tsk, int tf) {
   printf("\nEarliest-Deadline First Scheduling:\n");
   for (int t = 0; t < tf; t++) {
      int s_tsk = -1; // Index of the selected task to execute

      // Update next deadlines of tasks that are ready to execute
      for (int i = 0; i < n_tsk; i++) {
         if (t % tsks[i].p == 0) {
            tsks[i].rt = tsks[i].c;        // Reset remaining time to execution time
            tsks[i].nd = t + tsks[i].d;     // Calculate next deadline
         }
      }

      // Select the task with the earliest next deadline and non-zero remaining time
      for (int i = 0; i < n_tsk; i++) {
         if (tsks[i].rt > 0 && (s_tsk == -1 || tsks[i].nd < tsks[s_tsk].nd)) {
            s_tsk = i;
         }
      }

      // Execute the selected task or idle if no task is ready
```

```
        if (s_tsk != -1) {
            printf("Time %d: Task %d\n", t, tsks[s_tsk].id);
            tsks[s_tsk].rt--;   // Decrement remaining time of the selected task
        } else {
            printf("Time %d: Idle\n", t);
        }
    }
}

int main() {
    Task tsks[MAX_TSKS];
    int n_tsk;
    int tf;     // Time frame for simulation

    // Input task details
    Input(tsks, &n_tsk);

    // Input simulation time frame
    printf("Enter time frame for simulation: ");
    scanf("%d", &tf);

    // Perform EDF scheduling
    EDF(tsks, n_tsk, tf);

    return 0;
}
```

12. Proportional CPU Scheduling:

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

int main() {
    int n, totalTickets = 0;

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

    int pid[n];     // Array to store process IDs
    int tickets[n]; // Array to store number of tickets for each process
    tickets[0] = 0; // Initialize the cumulative ticket count

    // Input the number of tickets for each process
```

```
    printf("\nEnter the number of tickets for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("PID%d: ", i + 1);
        scanf("%d", &pid[i]);
        totalTickets += pid[i];    // Calculate total number of tickets
        tickets[i + 1] = totalTickets; // Store cumulative tickets up to this process
    }

    // Output the probability of each process being serviced
    printf("\nProbability of servicing each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: %d%%\n", i + 1, (pid[i] * 100) / totalTickets);
    }

    // Initialize random number generator
    srand(time(NULL));

    int t = 1; // Time unit
    int sum = totalTickets; // Total remaining tickets
    while (sum > 0) {
        // Generate a random number between 0 and totalTickets - 1
        int random = rand() % totalTickets;

        // Find the process whose ticket range includes the random number
        int j;
        for (j = 0; j < n; j++) {
            if (random < tickets[j + 1]) {
                printf("%d ms: Servicing Ticket of process %d\n", t, j + 1);
                pid[j]--; // Decrement the number of tickets for the selected process
                sum--;    // Decrease the total number of remaining tickets
                t++;      // Increment time unit
                break;    // Exit the loop once a process is selected
            }
        }
    }

    // Output processes that have finished executing
    for (int i = 0; i < n; i++) {
        if (pid[i] == 0) {
            printf("PID%d has finished executing\n", i + 1);
        }
    }

    return 0;
}
```

13. **Producer-consumer problem using semaphores:**

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

int mutex = 1, full = 0, empty = 3, x = 0; // Shared variables: mutex for mutual exclusion, full and
empty for counting items in buffer, x for item number

// Function prototypes
int wait(int s);
int signal(int s);
void producer();
void consumer();

// Function to decrement semaphore
int wait(int s) {
  return (--s);
}

// Function to increment semaphore
int signal(int s) {
  return (++s);
}

// Producer function
void producer() {
  mutex = wait(mutex); // Acquire mutex lock

  if (empty > 0) { // Check if there is space in the buffer
    empty--; // Decrease empty count
    x++; // Increment item number
    printf("\nProducer produces item %d\n", x); // Print produced item number
    full++; // Increase full count (item added to buffer)
  } else {
    printf("\nBuffer is full. Producer cannot produce.\n");
  }

  mutex = signal(mutex); // Release mutex lock
}

// Consumer function
void consumer() {
  mutex = wait(mutex); // Acquire mutex lock

  if (full > 0) { // Check if there are items in the buffer
```

```
        full--; // Decrease full count
        x--; // Decrement item number (consume item)
        printf("\nConsumer consumes item %d\n", x + 1); // Print consumed item number
        empty++; // Increase empty count (item removed from buffer)
    } else {
        printf("\nBuffer is empty. Consumer cannot consume.\n");
    }

    mutex = signal(mutex); // Release mutex lock
}

// Main function
int main() {
    int n;

    printf("\n1.Producer\n2.Consumer\n3.Exit\n");

    while (1) {
        printf("Enter your choice: ");
        scanf("%d", &n);

        switch (n) {
            case 1:
                if (mutex == 1 && empty != 0)
                    producer();
                else
                    printf("Buffer is full\n");
                break;
            case 2:
                if (mutex == 1 && full != 0)
                    consumer();
                else
                    printf("Buffer is empty\n");
                break;
            case 3:
                exit(0); // Exit the program
                break;
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

14. Dining-Philosophers problem:

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

#define MAX_PHILOSOPHERS 5

// Function to allow one philosopher to eat
void allow_one_to_eat(int hungry[], int n) {
   int isWaiting[MAX_PHILOSOPHERS];

   // Initialize isWaiting array to true (1)
   for (int i = 0; i < n; i++) {
      isWaiting[i] = 1;
   }

   // Grant permission for each philosopher in hungry[]
   for (int i = 0; i < n; i++) {
      printf("P %d is granted to eat\n", hungry[i]);
      isWaiting[hungry[i]] = 0;

      // Print waiting philosophers
      for (int j = 0; j < n; j++) {
         if (isWaiting[hungry[j]]) {
            printf("P %d is waiting\n", hungry[j]);
         }
      }

      // Reset isWaiting array for next iteration
      for (int k = 0; k < n; k++) {
         isWaiting[k] = 1;
      }
      isWaiting[hungry[i]] = 0; // Mark current philosopher as not waiting
   }
}

// Function to allow two philosophers to eat simultaneously
void allow_two_to_eat(int hungry[], int n) {
   if (n < 2 || n > MAX_PHILOSOPHERS) {
      printf("Invalid number of philosophers.\n");
      return;
   }

   // Grant permission for pairs of philosophers
   for (int i = 0; i < n - 1; i++) {
```

```
        for (int j = i + 1; j < n; j++) {
            printf("P %d and P %d are granted to eat\n", hungry[i], hungry[j]);

            // Print waiting philosophers
            for (int k = 0; k < n; k++) {
                if (k != i && k != j) {
                    printf("P %d is waiting\n", hungry[k]);
                }
            }
        }
    }
}

int main() {
    int total_philosophers, hungry_count;
    int hungry_positions[MAX_PHILOSOPHERS];

    // Input number of philosophers
    printf("DINING PHILOSOPHER PROBLEM\n");
    printf("Enter the total number of philosophers: ");
    scanf("%d", &total_philosophers);

    // Validate number of philosophers
    if (total_philosophers > MAX_PHILOSOPHERS || total_philosophers < 2) {
        printf("Invalid number of philosophers.\n");
        return 1;
    }

    // Input number of hungry philosophers
    printf("How many are hungry: ");
    scanf("%d", &hungry_count);

    // Validate number of hungry philosophers
    if (hungry_count < 1 || hungry_count > total_philosophers) {
        printf("Invalid number of hungry philosophers.\n");
        return 1;
    }

    // Input positions of hungry philosophers
    for (int i = 0; i < hungry_count; i++) {
        printf("Enter position of philosopher %d (0 to %d): ", i + 1, total_philosophers - 1);
        scanf("%d", &hungry_positions[i]);

        // Validate philosopher position
        if (hungry_positions[i] < 0 || hungry_positions[i] >= total_philosophers) {
```

```
            printf("Invalid philosopher position.\n");
            return 1;
        }
    }

    int choice;
    while (1) {
        // Menu for selecting dining strategy
        printf("\n1. One can eat at a time\n");
        printf("2. Two can eat at a time\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                allow_one_to_eat(hungry_positions, hungry_count);
                break;
            case 2:
                allow_two_to_eat(hungry_positions, hungry_count);
                break;
            case 3:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

15. Bankers algorithm for deadlock avoidance:

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

// Function to calculate the Need matrix
void calculateNeed(int P, int R, int need[P][R], int max[P][R], int allot[P][R]) {
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            need[i][j] = max[i][j] - allot[i][j];
}

// Function to check if the system is in a safe state
bool isSafe(int P, int R, int processes[], int avail[], int max[][R], int allot[][R]) {
```

```
int need[P][R];
calculateNeed(P, R, need, max, allot); // Calculate the Need matrix

bool finish[P]; // Array to track if a process has finished
for (int i = 0; i < P; i++) {
  finish[i] = false;
}

int safeSeq[P]; // Safe sequence of processes
int work[R]; // Work array to store available resources
for (int i = 0; i < R; i++) {
  work[i] = avail[i];
}

int count = 0; // Counter to track the number of processes finished
while (count < P) {
  bool found = false;
  for (int p = 0; p < P; p++) {
    if (finish[p] == false) {
      int j;
      for (j = 0; j < R; j++)
        if (need[p][j] > work[j])
          break;

      if (j == R) { // If all needs of process p can be satisfied
        printf("P%d is visited (", p);
        for (int k = 0; k < R; k++) {
          work[k] += allot[p][k]; // Release resources
          printf("%d ", work[k]);
        }
        printf(")\n");
        safeSeq[count++] = p; // Add process p to safe sequence
        finish[p] = true; // Mark process p as finished
        found = true;
      }
    }
  }

  if (found == false) { // If no process could be found in this iteration
    printf("System is not in safe state\n");
    return false;
  }
}

printf("SYSTEM IS IN SAFE STATE\nThe Safe Sequence is -- (");
```

```
    for (int i = 0; i < P; i++) {
        printf("P%d ", safeSeq[i]);
    }
    printf(")\n");

    return true;
}

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

    int processes[P];
    int avail[R];
    int max[P][R];
    int allot[P][R];

    for (int i = 0; i < P; i++) {
        processes[i] = i; // Initialize process numbers
    }

    // Input allocation and max matrices for each process
    for (int i = 0; i < P; i++) {
        printf("Enter details for P%d\n", i);
        printf("Enter allocation -- ");
        for (int j = 0; j < R; j++) {
            scanf("%d", &allot[i][j]);
        }
        printf("Enter Max -- ");
        for (int j = 0; j < R; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    // Input available resources
    printf("Enter Available Resources -- ");
    for (int i = 0; i < R; i++) {
        scanf("%d", &avail[i]);
    }

    // Check if the system is in a safe state
    isSafe(P, R, processes, avail, max, allot);
```

```
    // Print the Allocation, Max, and Need matrices
    printf("\nProcess\tAllocation\tMax\tNeed\n");
    for (int i = 0; i < P; i++) {
        printf("P%d\t", i);
        for (int j = 0; j < R; j++) {
            printf("%d ", allot[i][j]);
        }
        printf("\t");
        for (int j = 0; j < R; j++) {
            printf("%d ", max[i][j]);
        }
        printf("\t");
        for (int j = 0; j < R; j++) {
            printf("%d ", max[i][j] - allot[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

16. Deadlock detection:

```
#include <stdio.h>

int main() {
    int n, m, i, j, k;

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

    int alloc[n][m], request[n][m], avail[m];

    // Input allocation and request matrices for each process
    for (i = 0; i < n; i++) {
        printf("Enter details for P%d\n", i);
        printf("Enter allocation -- ");
        for (j = 0; j < m; j++) {
            scanf("%d", &alloc[i][j]);
        }
        printf("Enter Request -- ");
```

```
    for (j = 0; j < m; j++) {
        scanf("%d", &request[i][j]);
    }
}

// Input available resources
printf("Enter Available Resources -- ");
for (i = 0; i < m; i++) {
    scanf("%d", &avail[i]);
}

int finish[n], safeSeq[n], work[m], flag, f = 0;

// Initialize finish array to 0, indicating all processes are unfinished
for (i = 0; i < n; i++) {
    finish[i] = 0;
}

// Initialize work array with available resources
for (j = 0; j < m; j++) {
    work[j] = avail[j];
}

int count = 0;

// Loop until all processes are either finished or a deadlock is detected
while (count < n) {
    flag = 0;
    f = 0;
    for (i = 0; i < n; i++) {
        if (finish[i] == 0) {
            // Check if process i has any allocated resources
            for (j = 0; j < m; j++) {
                if (alloc[i][j] != 0) {
                    f = 1;
                    break;
                }
            }

            if (f) {
                int canProceed = 1;

                // Check if the request for resources can be satisfied
                for (j = 0; j < m; j++) {
                    if (request[i][j] > work[j]) {
```

```
                    canProceed = 0;
                    break;
                }
            }

            // If request can be satisfied, allocate resources and update work array
            if (canProceed) {
                for (k = 0; k < m; k++) {
                    work[k] += alloc[i][k];
                }
                safeSeq[count++] = i;
                finish[i] = 1;
                flag = 1;
            }
        } else {
            // If process has no allocated resources, mark it as finished
            safeSeq[count++] = i;
            finish[i] = 1;
            flag = 1;
        }
    }
}

// If no process was able to proceed, break the loop
if (flag == 0) {
    break;
}
}

int deadlock = 0;

// Check if any processes are unfinished, indicating a deadlock
for (i = 0; i < n; i++) {
    if (finish[i] == 0) {
        deadlock = 1;
        printf("\nSystem is in a deadlock state.\n");
        printf("The deadlocked processes are: ");
        for (j = 0; j < n; j++) {
            if (finish[j] == 0) {
                printf("P%d ", j);
            }
        }
        printf("\n");
        break;
    }
```

```
    }

    // If no deadlock is detected, print the safe sequence
    if (deadlock == 0) {
        printf("\nSystem is not in a deadlock state.\n");
        printf("Safe Sequence is: ");
        for (i = 0; i < n; i++) {
            printf("P%d ", safeSeq[i]);
        }
        printf("\n");
    }

    return 0;
}
```

17. **Worst-fit contiguous memory allocation technique:**
18. **Best-fit contiguous memory allocation technique:**
19. **First-fit contiguous memory allocation technique:**

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

#define MAX 25

// First Fit algorithm
void firstFit(int nb, int nf, int b[], int f[]) {
    int ff[MAX] = {0};        // Array to store allocated block index for each file
    int allocated[MAX] = {0};  // Array to track allocated blocks

    // Allocate each file to the first suitable block
    for (int i = 0; i < nf; i++) {
        ff[i] = -1;  // Initialize with -1 indicating no allocation
        for (int j = 0; j < nb; j++) {
            if (allocated[j] == 0 && b[j] >= f[i]) {  // Check if block is free and can accommodate the file
                ff[i] = j;
                allocated[j] = 1;  // Mark block as allocated
                break;
            }
        }
    }

    // Print allocation result
    printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:");
    for (int i = 0; i < nf; i++) {
```

```
        if (ff[i] != -1)
            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], ff[i] + 1, b[ff[i]]);
        else
            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);
    }
}

// Best Fit algorithm
void bestFit(int nb, int nf, int b[], int f[]) {
    int ff[MAX] = {0};        // Array to store allocated block index for each file
    int allocated[MAX] = {0};  // Array to track allocated blocks

    // Allocate each file to the best fitting block
    for (int i = 0; i < nf; i++) {
        int best = -1;
        ff[i] = -1;  // Initialize with -1 indicating no allocation
        for (int j = 0; j < nb; j++) {
            if (allocated[j] == 0 && b[j] >= f[i]) {  // Check if block is free and can accommodate the file
                if (best == -1 || b[j] < b[best])     // Find the best fitting block
                    best = j;
            }
        }
        if (best != -1) {
            ff[i] = best;
            allocated[best] = 1;  // Mark block as allocated
        }
    }

    // Print allocation result
    printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:");
    for (int i = 0; i < nf; i++) {
        if (ff[i] != -1)
            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], ff[i] + 1, b[ff[i]]);
        else
            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);
    }
}

// Worst Fit algorithm
void worstFit(int nb, int nf, int b[], int f[]) {
    int ff[MAX] = {0};        // Array to store allocated block index for each file
    int allocated[MAX] = {0};  // Array to track allocated blocks

    // Allocate each file to the worst fitting block
    for (int i = 0; i < nf; i++) {
```

```
        int worst = -1;
        ff[i] = -1;  // Initialize with -1 indicating no allocation
        for (int j = 0; j < nb; j++) {
            if (allocated[j] == 0 && b[j] >= f[i]) {  // Check if block is free and can accommodate the file
                if (worst == -1 || b[j] > b[worst])   // Find the worst fitting block
                    worst = j;
            }
        }
        if (worst != -1) {
            ff[i] = worst;
            allocated[worst] = 1;  // Mark block as allocated
        }
    }

    // Print allocation result
    printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:");
    for (int i = 0; i < nf; i++) {
        if (ff[i] != -1)
            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], ff[i] + 1, b[ff[i]]);
        else
            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);
    }
}

int main() {
    int nb, nf, choice;

    // Input number of blocks and files
    printf("Memory Management Scheme");
    printf("\nEnter the number of blocks: ");
    scanf("%d", &nb);
    printf("Enter the number of files: ");
    scanf("%d", &nf);

    int b[nb], f[nf];

    // Input size of each block
    printf("\nEnter the size of the blocks:\n");
    for (int i = 0; i < nb; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &b[i]);
    }

    // Input size of each file
    printf("Enter the size of the files:\n");
```

```
    for (int i = 0; i < nf; i++) {
        printf("File %d: ", i + 1);
        scanf("%d", &f[i]);
    }

    // Menu for choosing allocation scheme
    while (1) {
        printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\n\tMemory Management Scheme - First Fit\n");
                firstFit(nb, nf, b, f);  // Execute First Fit algorithm
                break;
            case 2:
                printf("\n\tMemory Management Scheme - Best Fit\n");
                bestFit(nb, nf, b, f);  // Execute Best Fit algorithm
                break;
            case 3:
                printf("\n\tMemory Management Scheme - Worst Fit\n");
                worstFit(nb, nf, b, f);  // Execute Worst Fit algorithm
                break;
            case 4:
                printf("\nExiting...\n");
                exit(0);  // Exit the program
                break;
            default:
                printf("\nInvalid choice.\n");
                break;
        }
    }

    return 0;
}
```

20. FIFO page replacement algorithm:

```
#include <stdio.h>

// Global variables
int n, nf;              // n: length of page reference sequence, nf: number of frames
int in[100];            // array to store the page reference sequence
int p[50];              // array to store the pages in frames
```

```c
int pgfaultcnt = 0;     // page fault counter

// Function to get data from the user
void getData()
{
   printf("\nEnter length of page reference sequence: ");
   scanf("%d", &n);  // Input length of page reference sequence
   printf("\nEnter the page reference sequence: ");
   for (int i = 0; i < n; i++)
      scanf("%d", &in[i]);  // Input page reference sequence
   printf("\nEnter no of frames: ");
   scanf("%d", &nf);  // Input number of frames
}

// Function to initialize frames with a large number (indicating empty frame)
void initialize()
{
   pgfaultcnt = 0;  // Reset page fault counter
   for (int i = 0; i < nf; i++)
      p[i] = 9999;  // Initialize frames with 9999
}

// Function to check if a page is a hit
int isHit(int data)
{
   for (int j = 0; j < nf; j++)
   {
      if (p[j] == data)  // If page is found in frames
         return 1;
   }
   return 0;  // Return hit status
}

// Function to display the pages in frames
void dispPages()
{
   for (int k = 0; k < nf; k++)
   {
      if (p[k] != 9999)
         printf(" %d", p[k]);  // Display pages in frames
   }
}

// Function to display the total number of page faults
void dispPgFaultCnt()
```

```
{
   printf("\nTotal no of page faults: %d", pgfaultcnt);  // Display page fault count
}

// FIFO page replacement algorithm
void fifo()
{
   initialize();  // Initialize frames
   for (int i = 0; i < n; i++)
   {
      printf("\nFor %d :", in[i]);

      if (isHit(in[i]) == 0)  // If page is not a hit
      {
         // Shift frames to the left
         for (int k = 0; k < nf - 1; k++)
            p[k] = p[k + 1];

         p[nf - 1] = in[i];  // Add new page at the end
         pgfaultcnt++;  // Increment page fault counter
         dispPages();  // Display pages in frames
      }
      else
         printf("No page fault");  // If page is a hit, no page fault
   }
   dispPgFaultCnt();  // Display total number of page faults
}

int main()
{
   getData();  // Get data from user
   fifo();  // Execute FIFO algorithm
   return 0;
}
```

21. LRU page replacement algorithm:

```
#include <stdio.h>

// Global variables
int n, nf;             // n: length of page reference sequence, nf: number of frames
int in[100];           // array to store the page reference sequence
int p[50];             // array to store the pages in frames
```

```c
int pgfaultcnt = 0;     // page fault counter

// Function to get data from the user
void getData()
{
   printf("\nEnter length of page reference sequence: ");
   scanf("%d", &n);  // Input length of page reference sequence
   printf("\nEnter the page reference sequence: ");
   for (int i = 0; i < n; i++)
      scanf("%d", &in[i]);  // Input page reference sequence
   printf("\nEnter no of frames: ");
   scanf("%d", &nf);  // Input number of frames
}

// Function to initialize frames with a large number (indicating empty frame)
void initialize()
{
   pgfaultcnt = 0;  // Reset page fault counter
   for (int i = 0; i < nf; i++)
      p[i] = 9999;  // Initialize frames with 9999
}

// Function to check if a page is a hit
int isHit(int data)
{
   for (int j = 0; j < nf; j++)
   {
      if (p[j] == data)  // If page is found in frames
         return 1;
   }
   return 0;  // Return hit status
}

// Function to display the pages in frames
void dispPages()
{
   for (int k = 0; k < nf; k++)
   {
      if (p[k] != 9999)
         printf(" %d", p[k]);  // Display pages in frames
   }
}

// Function to display the total number of page faults
void dispPgFaultCnt()
```

```
{
   printf("\nTotal no of page faults: %d", pgfaultcnt);  // Display page fault count
}

// LRU page replacement algorithm
void lru()
{
   initialize();  // Initialize frames

   int least[50];  // Array to store the last occurrence of pages
   for (int i = 0; i < n; i++)
   {
      printf("\nFor %d :", in[i]);

      if (isHit(in[i]) == 0)  // If page is not a hit
      {
         // Calculate the last occurrence of each page in frames
         for (int j = 0; j < nf; j++)
         {
            int pg = p[j];
            int found = 0;
            for (int k = i - 1; k >= 0; k--)
            {
               if (pg == in[k])
               {
                  least[j] = k;
                  found = 1;
                  break;
               }
            }
            if (!found)
               least[j] = -9999;  // If page is not found in the past, set to a large negative number
         }
         int min = 9999;
         int repindex;
         // Find the page with the farthest last occurrence
         for (int j = 0; j < nf; j++)
         {
            if (least[j] < min)
            {
               min = least[j];
               repindex = j;
            }
         }
         p[repindex] = in[i];  // Replace the page
```

```
            pgfaultcnt++;  // Increment page fault counter

            dispPages();  // Display pages in frames
        }
        else
            printf("No page fault");  // If page is a hit, no page fault
    }
    dispPgFaultCnt();  // Display total number of page faults
}

int main()
{
    getData();  // Get data from user
    lru();  // Execute LRU algorithm
    return 0;
}
```

22. Optimal page replacement algorithm:

```
#include <stdio.h>

// Global variables
int n, nf;              // n: length of page reference sequence, nf: number of frames
int in[100];            // array to store the page reference sequence
int p[50];              // array to store the pages in frames
int pgfaultcnt = 0;     // page fault counter

// Function to get data from the user
void getData()
{
    printf("\nEnter length of page reference sequence: ");
    scanf("%d", &n);  // Input length of page reference sequence
    printf("\nEnter the page reference sequence: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &in[i]);  // Input page reference sequence
    printf("\nEnter no of frames: ");
    scanf("%d", &nf);  // Input number of frames
}

// Function to initialize frames with a large number (indicating empty frame)
void initialize()
{
    pgfaultcnt = 0;  // Reset page fault counter
```

```c
    for (int i = 0; i < nf; i++)
        p[i] = 9999;  // Initialize frames with 9999
}

// Function to check if a page is a hit
int isHit(int data)
{
    for (int j = 0; j < nf; j++)
    {
        if (p[j] == data)  // If page is found in frames
            return 1;
    }
    return 0;  // Return hit status
}

// Function to display the pages in frames
void dispPages()
{
    for (int k = 0; k < nf; k++)
    {
        if (p[k] != 9999)
            printf(" %d", p[k]);  // Display pages in frames
    }
}

// Function to display the total number of page faults
void dispPgFaultCnt()
{
    printf("\nTotal no of page faults: %d", pgfaultcnt);  // Display page fault count
}

// Optimal page replacement algorithm
void optimal()
{
    initialize();  // Initialize frames
    int near[50];  // Array to store the next occurrence of pages
    for (int i = 0; i < n; i++)
    {
        printf("\nFor %d :", in[i]);

        if (isHit(in[i]) == 0)  // If page is not a hit
        {
            // Calculate the next occurrence of each page in frames
            for (int j = 0; j < nf; j++)
            {
```

```c
            int pg = p[j];
            int found = 0;
            for (int k = i; k < n; k++)
            {
               if (pg == in[k])
               {
                  near[j] = k;
                  found = 1;
                  break;
               }
            }
            if (!found)
               near[j] = 9999;  // If page is not found in the future, set to a large number
         }
         int max = -9999;
         int repindex;
         // Find the page with the farthest next occurrence
         for (int j = 0; j < nf; j++)
         {
            if (near[j] > max)
            {
               max = near[j];
               repindex = j;
            }
         }
         p[repindex] = in[i];  // Replace the page
         pgfaultcnt++;  // Increment page fault counter

         dispPages();  // Display pages in frames
      }
      else
         printf("No page fault");  // If page is a hit, no page fault
   }
   dispPgFaultCnt();  // Display total number of page faults
}

int main()
{
   getData();  // Get data from user
   optimal();  // Execute Optimal algorithm
   return 0;
}
```

23. All page replacement algorithms

```
#include<stdio.h>

// Global variables
int n, nf;              // n: length of page reference sequence, nf: number of frames
int in[100];            // array to store the page reference sequence
int p[50];              // array to store the pages in frames
int hit = 0;            // flag to check if page is hit
int i, j, k;            // loop control variables
int pgfaultcnt = 0;     // page fault counter

// Function to get data from the user
void getData()
{
   printf("\nEnter length of page reference sequence:");
   scanf("%d", &n);  // Input length of page reference sequence
   printf("\nEnter the page reference sequence:");
   for(i = 0; i < n; i++)
      scanf("%d", &in[i]);  // Input page reference sequence
   printf("\nEnter no of frames:");
   scanf("%d", &nf);  // Input number of frames
}

// Function to initialize frames with a large number (indicating empty frame)
void initialize()
{
   pgfaultcnt = 0;  // Reset page fault counter
   for(i = 0; i < nf; i++)
      p[i] = 9999;  // Initialize frames with 9999
}

// Function to check if a page is a hit
int isHit(int data)
{
   hit = 0;
   for(j = 0; j < nf; j++)
   {
      if(p[j] == data)  // If page is found in frames
      {
         hit = 1;
         break;
      }
   }
   return hit;  // Return hit status
```

```
}

// Function to get the index of the hit page
int getHitIndex(int data)
{
   int hitind;
   for(k = 0; k < nf; k++)
   {
      if(p[k] == data)  // If page is found in frames
      {
         hitind = k;
         break;
      }
   }
   return hitind;  // Return index of the hit page
}

// Function to display the pages in frames
void dispPages()
{
   for (k = 0; k < nf; k++)
   {
      if(p[k] != 9999)
         printf(" %d", p[k]);  // Display pages in frames
   }
}

// Function to display the total number of page faults
void dispPgFaultCnt()
{
   printf("\nTotal no of page faults: %d", pgfaultcnt);  // Display page fault count
}

// FIFO page replacement algorithm
void fifo()
{
   initialize();  // Initialize frames
   for(i = 0; i < n; i++)
   {
      printf("\nFor %d :", in[i]);

      if(isHit(in[i]) == 0)  // If page is not a hit
      {
         // Shift frames to the left
         for(k = 0; k < nf - 1; k++)
```

```
            p[k] = p[k + 1];

          p[k] = in[i];  // Add new page at the end
          pgfaultcnt++;  // Increment page fault counter
          dispPages();  // Display pages in frames
        }
        else
          printf("No page fault");  // If page is a hit, no page fault
    }
    dispPgFaultCnt();  // Display total number of page faults
}

// Optimal page replacement algorithm
void optimal()
{
    initialize();  // Initialize frames
    int near[50];  // Array to store the next occurrence of pages
    for(i = 0; i < n; i++)
    {
        printf("\nFor %d :", in[i]);

        if(isHit(in[i]) == 0)  // If page is not a hit
        {
          // Calculate the next occurrence of each page in frames
          for(j = 0; j < nf; j++)
          {
            int pg = p[j];
            int found = 0;
            for(k = i; k < n; k++)
            {
              if(pg == in[k])
              {
                near[j] = k;
                found = 1;
                break;
              }
              else
                found = 0;
            }
            if(!found)
              near[j] = 9999;  // If page is not found in the future, set to a large number
          }
          int max = -9999;
          int repindex;
          // Find the page with the farthest next occurrence
```

```
        for(j = 0; j < nf; j++)
        {
          if(near[j] > max)
          {
            max = near[j];
            repindex = j;
          }
        }
        p[repindex] = in[i];  // Replace the page
        pgfaultcnt++;  // Increment page fault counter

        dispPages();  // Display pages in frames
      }
      else
        printf("No page fault");  // If page is a hit, no page fault
    }
    dispPgFaultCnt();  // Display total number of page faults
}

// LRU page replacement algorithm
void lru()
{
    initialize();  // Initialize frames

    int least[50];  // Array to store the last occurrence of pages
    for(i = 0; i < n; i++)
    {
      printf("\nFor %d :", in[i]);

      if(isHit(in[i]) == 0)  // If page is not a hit
      {
        // Calculate the last occurrence of each page in frames
        for(j = 0; j < nf; j++)
        {
          int pg = p[j];
          int found = 0;
          for(k = i - 1; k >= 0; k--)
          {
            if(pg == in[k])
            {
              least[j] = k;
              found = 1;
              break;
            }
            else
```

```
                found = 0;
            }
            if(!found)
                least[j] = -9999;  // If page is not found in the past, set to a large negative number
        }
        int min = 9999;
        int repindex;
        // Find the page with the farthest last occurrence
        for(j = 0; j < nf; j++)
        {
            if(least[j] < min)
            {
                min = least[j];
                repindex = j;
            }
        }
        p[repindex] = in[i];  // Replace the page
        pgfaultcnt++;  // Increment page fault counter

        dispPages();  // Display pages in frames
    }
    else
        printf("No page fault!");  // If page is a hit, no page fault
  }
  dispPgFaultCnt();  // Display total number of page faults
}

// Main function to execute the program
int main()
{
  int choice;
  while(1)
  {
    // Display menu options
                                        printf("\nPage        Replacement        Algorithms\n1.Enter
data\n2.FIFO\n3.Optimal\n4.LRU\n7.Exit\nEnter your choice:");
    scanf("%d", &choice);
    switch(choice)
    {
    case 1:
        getData();  // Get data from user
        break;
    case 2:
        fifo();  // Execute FIFO algorithm
        break;
```

```
        case 3:
          optimal();  // Execute Optimal algorithm
          break;
        case 4:
          lru();  // Execute LRU algorithm
          break;
        default:
          return 0;  // Exit the program
          break;
        }
    }
}
```