Problem 01: Write a C program to simulate the CPU scheduling algorithm First Come First Serve(FCFS).

Source Code:

```
#include <stdio.h>

// Function to calculate waiting time and turnaround time for FCFS scheduling
void findWaitingTime(int n, int bt[], int wt[]) {
   wt[0] = 0; // Waiting time for the first process is always 0

   // Calculate waiting time for each
   process for (int i = 1; i < n; i++) {
      wt[i] = wt[i - 1] + bt[i - 1];
   }
}

void findTurnaroundTime(int n, int bt[], int wt[], int tat[]) {
   // Calculate turnaround time for each process
   for (int i = 0; i < n; i++) {
      tat[i] = bt[i] + wt[i];
   }
}

void findAverageTime(int n, int bt[]) {
   int wt[n], tat[n];

   // Calculate waiting time and turnaround time
   findWaitingTime(n, bt, wt);
   findTurnaroundTime(n, bt, wt, tat);

   // Calculate average waiting time and average turnaround
   time float total_wt = 0, total_tat = 0;
   for (int i = 0; i < n; i++)
      { total_wt += wt[i];
      total_tat += tat[i];
   }

   float avg_wt = total_wt / n;
   float avg_tat = total_tat / n;

   printf("Process\tBurst Time\tWaiting 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, bt[i], wt[i], tat[i]);
   }
```

```
    printf("Average Waiting Time: %.2f\n", avg_wt);
    printf("Average Turnaround Time: %.2f\n", avg_tat);
}

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

    int burst_time[n];

    printf("Enter the burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &burst_time[i]);
    }

    findAverageTime(n, burst_time);

    return 0;
}
```

Input and Output:

```
Enter the number of processes: 5
Enter the burst time for each process:
Process 1: 9
Process 2: 7
Process 3: 8
Process 4: 6
Process 5: 5
Process Burst Time  Waiting Time    Turnaround Time
P1   9       0       9
P2   7       9       16
P3   8       16      24
P4   6       24      30
P5   5       30      35
Average Waiting Time: 15.80
Average Turnaround Time: 22.80
```

Problem 02: Write a C program to simulate the CPU scheduling algorithm Shortest Job First(SJF).

Source Code:

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h> // Include the <limits.h> header for INT_MAX
// Structure to represent a process
struct Process {
    int id;        // Process ID
    int burst_time; // Burst time
};
// Function to perform non-preemptive SJF
scheduling void SJF(struct Process processes[], int n)
{
    int waiting_time[n], turnaround_time[n];
    bool completed[n];

   for (int i = 0; i < n; i++) {
      completed[i] = false;
    }

    int total_time = 0; // Total time elapsed
    int completed_count = 0; // Number of completed
    processes while (completed_count < n) {
       int shortest_index = -1;
       int shortest_burst = INT_MAX;
       // Find the process with the shortest burst time that has not completed yet
       for (int i = 0; i < n; i++) {
          if (!completed[i] && processes[i].burst_time < shortest_burst)
             { shortest_burst = processes[i].burst_time;
             shortest_index = i;
          }
       }
       if (shortest_index == -1) {
          // No eligible process found, increment the total time
          total_time++;
       } else {
          // Execute the shortest job
          waiting_time[shortest_index] = total_time;
          total_time += processes[shortest_index].burst_time;
          turnaround_time[shortest_index] = total_time;
          completed[shortest_index] = true;
          completed_count++;
       }
```

```
    }
    // Print the results
    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time,
waiting_ time[i], turnaround_time[i]);
    }
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
    printf("Enter the burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        processes[i].id = i + 1;
        scanf("%d", &processes[i].burst_time);
    }
    SJF(processes, n);
    return 0;
}
```

Input and Output:

```
Enter the number of processes: 5
Enter the burst time for each process:
Process 1: 9
Process 2: 7
Process 3: 8
Process 4: 1
Process 5: 3
Process Burst Time  Waiting Time    Turnaround Time
P1  9       19      28
P2  7       4       11
P3  8       11      19
P4  1       0       1
P5  3       1       4
```

Problem 03: Write a C program to simulate the CPU scheduling algorithm Round Robin (RR).
Source Code:

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

// Structure to represent a process
struct Process {
   int id;          // Process ID
   int burst_time;     // Burst time
   int remaining_time; // Remaining burst time
};
// Function to perform Round Robin scheduling
void RoundRobin(struct Process processes[], int n, int time_quantum)
   { int remaining_processes = n;
   int current_time = 0;
   // Create an array to store waiting times for each
   process int waiting_time[n];
   for (int i = 0; i < n; i++) { waiting_time[i] = 0;
      processes[i].remaining_time = processes[i].burst_time;
   } while (remaining_processes > 0)
      { for (int i = 0; i < n; i++) {
         if (processes[i].remaining_time > 0) {
            // Execute the process for the time quantum or its remaining time, whichever
is smaller
            int execution_time = (processes[i].remaining_time < time_quantum) ? pro-
cesses[i].remaining_time : time_quantum;
            // Update the remaining time for the process
            processes[i].remaining_time -= execution_time;
            // Update current time
            current_time += execution_time;
            // Update waiting time for the process
            waiting_time[i] += current_time;
            if (processes[i].remaining_time == 0)
               { remaining_processes--;
            }
         }
      }
   }
   // Calculate turnaround time and print results
   printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
   for (int i = 0; i < n; i++) {
```

```
        int turnaround_time = waiting_time[i] + processes[i].burst_time;
        printf("P%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, waiting_
time[i], turnaround_time);
    }
}
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
    printf("Enter the burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        processes[i].id = i + 1;
        scanf("%d", &processes[i].burst_time);
    }
    int time_quantum;
    printf("Enter the time quantum: ");
    scanf("%d", &time_quantum);
    RoundRobin(processes, n, time_quantum);
    return 0;
}
```

Input and Output:

```
Enter the number of processes: 5
Enter the burst time for each process:
Process 1: 9
Process 2: 4
Process 3: 4
Process 4: 9
Process 5: 5
Enter the time quantum: 3
Process Burst Time  Waiting Time    Turnaround Time
P1  9       49      58
P2  4       25      29
P3  4       29      33
P4  9       66      75
P5  5       40      45
```

Problem 04: Write a C program to simulate the CPU scheduling priority algorithm.
Source Code:

```
#include <stdio.h>

// Structure to represent a process
struct Process {
   int id;        // Process ID
   int burst_time; // Burst time
   int priority;   // Priority
};

// Function to perform Priority scheduling
void PriorityScheduling(struct Process processes[], int n)
   { int waiting_time[n], turnaround_time[n];

   // Sort the processes based on priority (higher priority has lower value)
   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
            struct Process temp = processes[j];
            processes[j] = processes[j + 1];
            processes[j + 1] = temp;
         }
      }
   }

   // Calculate waiting time and turnaround time
   waiting_time[0] = 0; turnaround_time[0] =
   processes[0].burst_time;

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

   // Print the results
   printf("Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
   for (int i = 0; i < n; i++) {
      printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time,
processes[i].priority, waiting_time[i], turnaround_time[i]);
   }
}
```

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

   struct Process processes[n];

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

   PriorityScheduling(processes, n);

   return 0;
}
```

Input and Output:

```
Enter the number of processes: 5
Enter the burst time and priority for each process:
Process 1 (Burst Time Priority): 6
9
Process 2 (Burst Time Priority): 6
1
Process 3 (Burst Time Priority): 3
8
Process 4 (Burst Time Priority): 9
2
Process 5 (Burst Time Priority): 3
5
Process Burst Time  Priority    Waiting Time    Turnaround Time
P2   6        1        0        6
P4   9        2        6        15
P5   3        5        15       18
P3   3        8        18       21
P1   6        9        21       27
```

Problem 05: Write a C program to simulate producer-consumer problem using semaphores.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
// Initialize a mutex to
1 int mutex = 1;
// Number of full slots as
0 int full = 0;
// Number of empty slots as size
// of buffer
int empty = 10, x = 0;
// Function to produce an item and
// add it to the buffer
void producer()
{
        // Decrease mutex value by
        1 --mutex;
        // Increase the number of full
        // slots by 1
        ++full;
        // Decrease the number of empty
        // slots by 1
        --empty;
        // Item produced x++;
        printf("\nProducer produces"

                "item %d
                ", x);
        // Increase mutex value by
        1 ++mutex;
}
// Function to consume an item and
// remove it from buffer
void consumer()
{
        // Decrease mutex value by
        1 --mutex;
        // Decrease the number of full
        // slots by 1
        --full;
        // Increase the number of empty
        // slots by 1
```

```
        ++empty;
        printf("\nConsumer consumes "
                "item %d",
                x);
        x--;
        // Increase mutex value by
        1 ++mutex;
}
// Driver
Code int
main()
{
        int n, i;
        printf("\n1. Press 1 for Producer"
                "\n2. Press 2 for Consumer"
                "\n3. Press 3 for Exit");

// Using '#pragma omp parallel for'
// can give wrong value due to
// synchronization issues.

// 'critical' specifies that code is
// executed by only one thread at a
// time i.e., only one thread enters
// the critical section at a given
time #pragma omp critical
        for (i = 1; i > 0; i++)
                { printf("\nEnter your
                choice:"); scanf("%d", &n);
                // Switch Cases
                switch (n) {
                case 1:
                        // If mutex is 1 and empty
                        // is non-zero, then it is
                        // possible to produce
                        if ((mutex == 1)
                                && (empty != 0)) {
                                producer();
                        }
                        // Otherwise, print buffer
                        // is full
                        else {
                              printf("Buffer is full!");
                        }
```

```
                break;

        case 2:
                // If mutex is 1 and full
                // is non-zero, then it is
                // possible to consume
                if ((mutex == 1)
                        && (full != 0))
                        { consumer();
                }
                // Otherwise, print Buffer
                // is empty
                else {
                        printf("Buffer is empty!");
                }
                break;
        // Exit Condition
        case 3:
                exit(0);
                break;
        }
    }
}
```

Input and Output:

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:2
Buffer is empty!
Enter your choice:1
Producer producesitem 1
Enter your choice:1
Producer producesitem 2
Enter your choice:2
Consumer consumes item 2
Enter your choice:1
Producer producesitem 2
Enter your choice:2
Consumer consumes item 2
Enter your choice:2
Consumer consumes item 1
Enter your choice:2
Buffer is empty!
Enter your choice:3
```

Problem 06: Write a C program to simulate the concept of dining-philosophers problem.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_PHILOSOPHERS 5
#define NUM_CHOPSTICKS 5

void dine(int n);
pthread_t philosopher[NUM_PHILOSOPHERS];
pthread_mutex_t chopstick[NUM_CHOPSTICKS];

int main()
{
 // Define counter var i and
 status_message int i, status_message;
 void *msg;

 // Initialise the semaphore array
 for (i = 1; i <= NUM_CHOPSTICKS; i++)
 {
  status_message = pthread_mutex_init(&chopstick[i], NULL);
  // Check if the mutex is initialised
  successfully if (status_message == -1)
  {
   printf("\n Mutex initialization failed");
   exit(1);
  }
 }

 // Run the philosopher Threads using *dine() function
 for (i = 1; i <= NUM_PHILOSOPHERS; i++)
 {
  status_message = pthread_create(&philosopher[i], NULL, (void *)dine, (int
  *)i); if (status_message != 0)
  {
   printf("\n Thread creation error \n");
   exit(1);
  }
 }
```

```
  // Wait for all philosophers threads to complete executing (finish dining) before
clos-ing the program
  for (i = 1; i <= NUM_PHILOSOPHERS; i++)
  {
    status_message = pthread_join(philosopher[i], &msg);
    if (status_message != 0)
    {
     printf("\n Thread join failed \n");
     exit(1);
    }
  }

  // Destroy the chopstick Mutex array
  for (i = 1; i <= NUM_CHOPSTICKS; i++)
  {
    status_message = pthread_mutex_destroy(&chopstick[i]);
    if (status_message != 0)
    {
     printf("\n Mutex Destroyed \n");
     exit(1);
    }
  }
  return 0;
}
void dine(int n)
{
  printf("\nPhilosopher % d is thinking ", n);
  // Philosopher picks up the left chopstick (wait)
  pthread_mutex_lock(&chopstick[n]);
  // Philosopher picks up the right chopstick (wait)
  pthread_mutex_lock(&chopstick[(n + 1) % NUM_CHOPSTICKS]);
  // After picking up both the chopstick philosopher starts eating
  printf("\nPhilosopher % d is eating ", n);
  sleep(3);
  // Philosopher places down the left chopstick (signal)
  pthread_mutex_unlock(&chopstick[n]);
  // Philosopher places down the right chopstick (signal)
  pthread_mutex_unlock(&chopstick[(n + 1) % NUM_CHOPSTICKS]);

  // Philosopher finishes eating
  printf("\nPhilosopher % d Finished eating ", n);
}
```

Input and Output:

```
Philosopher  1 is thinking
Philosopher  1 is eating
Philosopher  5 is thinking
Philosopher  2 is thinking
Philosopher  3 is thinking
Philosopher  3 is eating
Philosopher  4 is thinking
Philosopher  1 Finished eating
Philosopher  5 is eating
Philosopher  3 Finished eating
Philosopher  2 is eating
Philosopher  4 is eating
Philosopher  5 Finished eating
Philosopher  2 Finished eating
Philosopher  4 Finished eating
```

Problem 07: Write a C program to simulate the the following contiguous memory allocation- (a) Worst Fit (b) Best Fit (c) First Fit.

Source Code:

```
#include <stdio.h>
#define MAX_MEMORY_SIZE 1000 // Maximum available memory size

struct Block {
   int size;
   int allocated; // 1 if allocated, 0 if free
};

struct Block memory[MAX_MEMORY_SIZE]; // Memory blocks

void initializeMemory() {
   for (int i = 0; i < MAX_MEMORY_SIZE; i++) {
      memory[i].size = 0;
      memory[i].allocated = 0;
   }
}

// Function to allocate memory using Worst
Fit void worstFit(int processSize) {
   int worstFitIdx = -1;
   int worstFitSize = -1;

   for (int i = 0; i < MAX_MEMORY_SIZE; i++) {
   if (!memory[i].allocated && memory[i].size >= processSize) { if
   (memory[i].size > worstFitSize) { worstFitSize = memory[i].size;
            worstFitIdx = i;
         }
      }
   }

   if (worstFitIdx != -1) {
      memory[worstFitIdx].allocated = 1;
      printf("Allocated %d units of memory at index %d (Worst Fit).\n",
processSize, worstFitIdx);
   } else {
      printf("Not enough memory available for allocation (Worst Fit).\n");
   }
}
```

```
// Function to allocate memory using Best
Fit void bestFit(int processSize) {
   int bestFitIdx = -1;
   int bestFitSize = MAX_MEMORY_SIZE + 1;

   for (int i = 0; i < MAX_MEMORY_SIZE; i++) {
    if (!memory[i].allocated && memory[i].size >= processSize) { if
      (memory[i].size < bestFitSize) { bestFitSize = memory[i].size;
            bestFitIdx = i;
         }
      }
   }

   if (bestFitIdx != -1) {
      memory[bestFitIdx].allocated = 1;
      printf("Allocated %d units of memory at index %d (Best Fit).\n", processSize,
best-FitIdx);
   } else {
      printf("Not enough memory available for allocation (Best Fit).\n");
   }
}

// Function to allocate memory using First
Fit void firstFit(int processSize) {
   for (int i = 0; i < MAX_MEMORY_SIZE; i++) {
      if (!memory[i].allocated && memory[i].size >= processSize)
         { memory[i].allocated = 1;
         printf("Allocated %d units of memory at index %d (First Fit).\n", processSize,
         i); return;
      }
   }
   printf("Not enough memory available for allocation (First Fit).\n");
}

int main() {
   initializeMemory();

   // Initialize memory blocks with
   sizes memory[0].size = 200;
   memory[1].size = 300;
   memory[2].size = 100;
   memory[3].size = 150;
```

```
    memory[4].size = 250;

    int choice;
    int processSize;

    do {
        printf("\nMemory Allocation Algorithms:\n");
        printf("1. Worst Fit\n"); printf("2. Best Fit\n");
        printf("3. First Fit\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        if (choice == 4) {
            break;
        }

        printf("Enter process size: ");
        scanf("%d", &processSize);

        switch (choice) {
            case 1:
                worstFit(processSize);
                break;
            case 2:
                bestFit(processSize);
                break;
            case 3:
                firstFit(processSize);
                break;
            default:
                printf("Invalid choice.\n");
                break;
        }
    } while (choice !=

    4); return 0;

}
```

Input and Output:

```
Memory Allocation Algorithms:
1. Worst Fit
2. Best Fit
3. First Fit
4. Exit
Enter your choice: 1
Enter process size: 5
Allocated 5 units of memory at index 1 (Worst Fit).

Memory Allocation Algorithms:
1. Worst Fit
2. Best Fit
3. First Fit
4. Exit
Enter your choice: 2
Enter process size: 10
Allocated 10 units of memory at index 2 (Best Fit).

Memory Allocation Algorithms:
1. Worst Fit
2. Best Fit
3. First Fit
4. Exit
Enter your choice: 3
Enter process size: 6
Allocated 6 units of memory at index 0 (First Fit).

Memory Allocation Algorithms:
1. Worst Fit
2. Best Fit
3. First Fit
4. Exit
Enter your choice:
```

Problem 08: Write a C program to implement page replacement techniques - (a) First in Frist Out (b) Least Recently Used (c) Optimal Source Code:

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

#define MAX_FRAMES 3 // Maximum number of frames in memory

// Function to find the index of the page that will be replaced in the future
(Optimal Algorithm)
int findOptimalReplacement(int pages[], int frames[], int n, int current)
  { int index = -1, farthest = -1;
  for (int i = 0; i < MAX_FRAMES; i++)
    { int j;
    for (j = current + 1; j < n; j++)
      { if (frames[i] == pages[j]) {
        if (j > farthest) {
          farthest = j;
          index = i;
        }
        break;
      }
    }
     if (j == n)
       return i;
  }
  return (index == -1) ? 0 : index;
}

// Function to simulate the FIFO page replacement algorithm
void fifoPageReplacement(int pages[], int n)
  { int frames[MAX_FRAMES];
  int front = 0, rear = -1;
  int pageFaults = 0;

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

  for (int i = 0; i < n; i++) {
    int currentPage = pages[i];
    int isPageInFrames = 0;
```

```
        // Check if the current page is already in frames
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == currentPage)
                { isPageInFrames = 1;
                break;
            }
        }

        // If the page is not in frames, replace the page at the front and add the new
        page if (!isPageInFrames) {
            pageFaults++;
            rear = (rear + 1) % MAX_FRAMES;
            frames[rear] = currentPage;
        }
    }

    printf("FIFO Page Replacement Algorithm:\n");
    printf("Total Page Faults: %d\n", pageFaults);
}

// Function to simulate the LRU page replacement
algorithm void lruPageReplacement(int pages[], int n) {
    int frames[MAX_FRAMES];
    int pageFaults = 0;

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

    for (int i = 0; i < n; i++) {
        int currentPage = pages[i];
        int isPageInFrames = 0;

        // Check if the current page is already in frames
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == currentPage)
                { isPageInFrames = 1;
                // Move the current page to the front (most recently used)
                for (int k = j; k > 0; k--) {
                    frames[k] = frames[k - 1];
                }
                frames[0] = currentPage;
                break;
```

```
        }
      }

      // If the page is not in frames, replace the least recently used
      page if (!isPageInFrames) {
        pageFaults++;
        for (int j = MAX_FRAMES - 1; j > 0; j--)
          { frames[j] = frames[j - 1];
        }
        frames[0] = currentPage;
      }
    }

    printf("LRU Page Replacement Algorithm:\n");
    printf("Total Page Faults: %d\n", pageFaults);
}
// Function to simulate the Optimal page replacement algorithm
void optimalPageReplacement(int pages[], int n) {
    int frames[MAX_FRAMES];
    int pageFaults = 0;
    for (int i = 0; i < MAX_FRAMES; i++)
      { frames[i] = -1; // Initialize frames as
      empty
    }
    for (int i = 0; i < n; i++) { int
      currentPage = pages[i]; int
      isPageInFrames = 0;
      // Check if the current page is already in frames
      for (int j = 0; j < MAX_FRAMES; j++) {
        if (frames[j] == currentPage) {
          isPageInFrames = 1;
          break;
        }
      }
      // If the page is not in frames, find the page to replace using the Optimal Algo-
rithm
      if (!isPageInFrames)
        { pageFaults++;
        int replaceIndex = findOptimalReplacement(pages, frames, n,
        i); frames[replaceIndex] = currentPage;
      }
    }
    printf("Optimal Page Replacement Algorithm:\n");
    printf("Total Page Faults: %d\n", pageFaults);
```

```
}
int main() {
    int n, pages[MAX_FRAMES];
    printf(“Enter the number of pages: “);
    scanf(“%d”, &n);
    printf(“Enter the page sequence: “);
    for (int i = 0; i < n; i++) {
        scanf(“%d”, &pages[i]);
    }
    fifoPageReplacement(pages, n);
    lruPageReplacement(pages, n);
    optimalPageReplacement(pages, n);
    return 0;
}
```

Input and Output:

```
Enter the number of pages: 10
Enter the page sequence: 5
6
8
2
8
9
3
FIFO Page Replacement Algorithm:
Total Page Faults: 2
LRU Page Replacement Algorithm:
Total Page Faults: 2
Optimal Page Replacement Algorithm:
Total Page Faults: 2
```

Problem 09: Write a C program to simulate all file allocation strategies - (a) Sequential (b) Indexed (c)

Linked Source Code:

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

#define MAX_BLOCKS 1000 // Maximum number of blocks

// Structure for block in linked
allocation struct LinkedBlock {
   int data;
   struct LinkedBlock *next;
};

// Function to simulate sequential file allocation
void sequentialFileAllocation(int blocks[], int n, int fileSize)
   { int startBlock = -1;
   for (int i = 0; i < n; i++) {
      if (blocks[i] == 0) { // Check if the block is
         free if (startBlock == -1) {
            startBlock = i;
         }
         fileSize--;
         blocks[i] = 1; // Allocate the block
         if (fileSize == 0) {
            break;
         }
      }
   }

   if (fileSize == 0) {
      printf("Sequential File Allocation: File allocated from block %d to
block %d.\n", startBlock, startBlock + fileSize);
   } else {
      printf("Sequential File Allocation: Not enough contiguous space available.\n");
   }
}

// Function to simulate indexed file allocation
void indexedFileAllocation(int blocks[], int n, int indexBlock, int fileSize)
   { if (blocks[indexBlock] == 0) { // Check if the index block is free
      blocks[indexBlock] = 1; // Allocate the index block
```

```
    int allocatedBlocks[fileSize];
    for (int i = 0; i < fileSize; i++) {
      allocatedBlocks[i] = -1; // Initialize allocated blocks to -1
    }

    for (int i = 0; i < fileSize; i++) {
      for (int j = 0; j < n; j++) {
        if (blocks[j] == 0) { // Check if the block is free
          blocks[j] = 1; // Allocate the block
          allocatedBlocks[i] = j; // Store the allocated
          block break;
        }
      }
    }

    printf("Indexed File Allocation: File allocated with index block %d and data
blocks: ", indexBlock);
   for (int i = 0; i < fileSize; i++) {
     if (allocatedBlocks[i] != -1) {
        printf("%d ", allocatedBlocks[i]);
      }
    }
    printf("\n");
  } else {
    printf("Indexed File Allocation: Index block is already allocated.\n");
  }
}

// Function to simulate linked file allocation
void linkedFileAllocation(struct LinkedBlock *blocks[], int n, int fileSize)
  { struct LinkedBlock *startBlock = NULL;
  struct LinkedBlock *currentBlock =

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

    if (blocks[i] == NULL) { // Check if the block is
      free if (startBlock == NULL) {
        startBlock = (struct LinkedBlock *)malloc(sizeof(struct
        LinkedBlock)); currentBlock = startBlock;
      } else {
       currentBlock->next = (struct LinkedBlock *)malloc(sizeof(struct Linked-
Block));
        currentBlock = currentBlock->next;
      }
```

```
            currentBlock->data = i;
            currentBlock->next = NULL;
            fileSize--;
            if (fileSize == 0) {
               break;
            }
         }
      }

   if (fileSize == 0) {
      printf("Linked File Allocation: File allocated with blocks: ");
      currentBlock = startBlock;
      while (currentBlock != NULL) {
         printf("%d ", currentBlock->data);
         currentBlock = currentBlock->next;
      }
      printf("\n");
   } else {
      printf("Linked File Allocation: Not enough blocks available.\n");
   }
}

int main() {
   int blocks[MAX_BLOCKS] = {0}; // Initialize blocks as
   free int n; // Total number of blocks
   int fileSize; // Size of the file to be allocated
   int indexBlock; // Index block for indexed allocation

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

   printf("Enter the size of the file to be allocated:
   "); scanf("%d", &fileSize);

   printf("Enter the index block for indexed allocation (0-%d): ", n -
   1); scanf("%d", &indexBlock);

   if (n <= 0 || fileSize <= 0 || indexBlock < 0 || indexBlock >= n)
      { printf("Invalid input.\n");
      return 1;
   }

   sequentialFileAllocation(blocks, n, fileSize);
```

```
    indexedFileAllocation(blocks, n, indexBlock, fileSize);

    // Linked allocation requires an array of pointers to linked
    blocks struct LinkedBlock *linkedBlocks[MAX_BLOCKS] =
    {NULL}; linkedFileAllocation(linkedBlocks, n, fileSize);

    return 0;
}
```

Input and Output:

```
Enter the total number of blocks: 10
Enter the size of the file to be allocated: 5
Enter the index block for indexed allocation (0-9): 2
Sequential File Allocation: File allocated from block 0 to block 0.
Indexed File Allocation: Index block is already allocated.
Linked File Allocation: File allocated with blocks: 0 1 2 3 4
```

Problem 10: Write a C program for banker's algorithm using deadlock method.
Source Code:

```
#include <stdio.h>

// Maximum number of processes and
resources #define MAX_PROCESSES 10
#define MAX_RESOURCES 10

// Function to check if the system is in a safe state
int isSafe(int available[], int max[][MAX_RESOURCES], int allocation[][MAX_RE-
SOURCES], int need[][MAX_RESOURCES], int processes, int resources) {
   int work[MAX_RESOURCES];
   int finish[MAX_PROCESSES] = {0}; // Initialize all processes as unfinished

   // Initialize the work array to available
   resources for (int i = 0; i < resources; i++) {
      work[i] = available[i];
   }

   int safeSequence[MAX_PROCESSES];
   int count = 0; // Count of safe processes

   while (count < processes) {
      int found = 0;

      for (int i = 0; i < processes; i++) {
         if (!finish[i]) {
            int j;
            for (j = 0; j < resources; j++) {
               if (need[i][j] > work[j]) {
                  break;
               }
            }

            if (j == resources) {
               // Process i can complete its execution
               for (int k = 0; k < resources; k++) {
                  work[k] += allocation[i][k];
               }
               safeSequence[count++] = i;
               finish[i] = 1;
               found = 1;
            }
```

```
        }
    }

    if (!found) {
        // No process found that can be executed, meaning the system is not in a safe
state
        return 0;
    }
  }

  // If all processes can complete execution, the system is in a safe
  state printf("Safe Sequence: ");
    for (int i = 0; i < processes; i++)
                          { printf("%d ",
                        safeSequence[i]);
  }
  printf("\n");

  return 1;
}

int main() {
  int processes, resources;
  int max[MAX_PROCESSES][MAX_RESOURCES];
  int allocation[MAX_PROCESSES][MAX_RESOURCES];
  int available[MAX_RESOURCES];
  int need[MAX_PROCESSES][MAX_RESOURCES];

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

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

  printf("Enter the maximum resource instances for each process:\n");
  for (int i = 0; i < processes; i++) {
    for (int j = 0; j < resources; j++) {
      scanf("%d", &max[i][j]);
    }
  }

  printf("Enter the allocated resource instances for each
  process:\n"); for (int i = 0; i < processes; i++) {
    for (int j = 0; j < resources; j++) {
```

```
            scanf(“%d”, &allocation[i][j]);
            need[i][j] = max[i][j] - allocation[i][j]; // Calculate the need matrix
        }
    }

    printf(“Enter the available resource
    instances:\n”); for (int i = 0; i < resources; i++) {
        scanf(“%d”, &available[i]);
    }

    if (isSafe(available, max, allocation, need, processes, resources))
        { printf(“System is in a safe state.\n”);
    } else {
        printf(“System is in an unsafe state (potential deadlock).\n”);
    }

    return 0;
}
```

Input and Output:

```
Terminal
Enter the number of processes: 2
Enter the number of resources: 2
Enter the maximum resource instances for each process:
1
5
6
3
Enter the allocated resource instances for each process:
3
6
4
8
Enter the available resource instances:
3
3
Safe Sequence: 0 1
System is in a safe state.
```