1)Scheduling Algorithms Simulation: FCFS

Problem Explanation:
The task is to simulate the First-Come-First-Serve (FCFS) scheduling algorithm. FCFS is a non-preemptive scheduling algorithm where the processes are executed in the order they arrive. In this simulation, we need to calculate the waiting time and turnaround time for each process.

Algorithm/Flowchart:
- Start the program.
- Accept the number of processes, burst time, and arrival time from the user.
- Initialize variables for total waiting time and total turnaround time to zero.
- Calculate the waiting time and turnaround time for each process using the FCFS algorithm.
    - ➢ a. Set the first process waiting time as zero.
    - ➢ b. For each subsequent process, calculate the waiting time as the sum of the burst times of all previous processes.
    - ➢ c. Calculate the turnaround time by adding the burst time and waiting time.
- Calculate the average waiting time and average turnaround time.
- Display the process details, waiting time, and turnaround time for each process.
- Display the average waiting time and average turnaround time.
- End the program.

Code :
```
#include <iostream>
using namespace std;

void calculateTimes(int processes[], int n, int burstTime[]) {
   int waitingTime[n];
   int turnaroundTime[n];

   waitingTime[0] = 0;

   for (int i = 1; i < n; i++) {
      waitingTime[i] = burstTime[i - 1] + waitingTime[i - 1];
      turnaroundTime[i] = burstTime[i] + waitingTime[i];
   }

   cout << "Process\t\tBurst Time\tWaiting Time\tTurnaround Time" << endl;
   for (int i = 0; i < n; i++) {
      cout << processes[i] << "\t\t" << burstTime[i] << "\t\t" << waitingTime[i] << "\t\t" <<
turnaroundTime[i] << endl;
   }

   double averageWaitingTime = 0;
   double averageTurnaroundTime = 0;
   for (int i = 0; i < n; i++) {
      averageWaitingTime += waitingTime[i];
      averageTurnaroundTime += turnaroundTime[i];
   }
   averageWaitingTime /= n;
   averageTurnaroundTime /= n;

   cout << "Average Waiting Time: " << averageWaitingTime << endl;
   cout << "Average Turnaround Time: " << averageTurnaroundTime << endl;
}

int main() {
   int n;
   cout << "Enter the number of processes: ";
   cin >> n;

   int processes[n];
   int burstTime[n];

   cout << "Enter process IDs and burst times:" << endl;
   for (int i = 0; i < n; i++) {
      cout << "Process " << i + 1 << " ID: ";
      cin >> processes[i];
      cout << "Burst Time: ";
      cin >> burstTime[i];
   }

   calculateTimes(processes, n, burstTime);

   return 0;
}
```

2)Scheduling Algorithms Simulation: SJF

Problem Explanation:
The task is to simulate the Shortest Job First (SJF) scheduling algorithm. SJF is a non-preemptive scheduling algorithm where the process with the shortest burst time is executed first. In this simulation, we need to calculate the waiting time and turnaround time for each process.

Algorithm/Flowchart:

- Start the program.
- Accept the number of processes, burst time, and arrival time from the user.
- Sort the processes based on their burst time in ascending order.
- Initialize variables for total waiting time and total turnaround time to zero.
- Calculate the waiting time and turnaround time for each process using the SJF algorithm.
    - ➢ a. Set the first process waiting time as zero.

    - ➢ b. For each subsequent process, calculate the waiting time as the sum of the burst times of all previous processes.
    - ➢ c. Calculate the turnaround time by adding the burst time and waiting time.
- Calculate the average waiting time and average turnaround time.
- Display the process details, waiting time, and turnaround time for each process.
- Display the average waiting time and average turnaround time.
- End the program.

Code :

```
#include <iostream>
#include <algorithm>
using namespace std;

struct Process {
    int processId;
    int burstTime;
};

bool compareByBurstTime(const Process& a, const Process& b) {
    return a.burstTime < b.burstTime;
}

void calculateTimes(Process processes[], int n) {
    int waitingTime[n];
    int turnaroundTime[n];

    waitingTime[0] = 0;

    for (int i = 1; i < n; i++) {
        waitingTime[i] = processes[i - 1].burstTime + waitingTime[i - 1];
        turnaroundTime[i] = processes[i].burstTime + waitingTime[i];
    }

    cout << "Process\tBurst Time\tWaiting Time\tTurnaround Time" << endl;
    for (int i = 0; i < n; i++) {
        cout << processes[i].processId << "\t\t" << processes[i].burstTime << "\t\t" <<
waitingTime[i] << "\t\t" << turnaroundTime[i] << endl;
    }

    double averageWaitingTime = 0;
    double averageTurnaroundTime = 0;
    for (int i = 0; i < n; i++) {
        averageWaitingTime += waitingTime[i];
        averageTurnaroundTime += turnaroundTime[i];
    }
    averageWaitingTime /= n;
    averageTurnaroundTime /= n;
    cout << "Average Waiting Time: " << averageWaitingTime << endl;
    cout << "Average Turnaround Time: " << averageTurnaroundTime << endl;
}

int main() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    Process processes[n];

    cout << "Enter process IDs and burst times:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Process " << i + 1 << " ID: ";
        cin >> processes[i].processId;
        cout << "Burst Time: ";
        cin >> processes[i].burstTime;
    }

    sort(processes, processes + n, compareByBurstTime);

    calculateTimes(processes, n);

    return 0;
}
```

3)Scheduling Algorithms Simulation: Priority

Problem Explanation:
The task is to simulate the Priority scheduling algorithm. Priority scheduling is a non-preemptive scheduling algorithm where each process is assigned a priority value. The process with the highest priority is executed first. In this simulation, we need to calculate the waiting time and turnaround time for each process.

Algorithm/Flowchart:

- Start the program.
- Accept the number of processes, burst time, and priority from the user.
- Initialize variables for total waiting time and total turnaround time to zero.
- Calculate the waiting time and turnaround time for each process using the Priority algorithm.
    - ➢ a. Set the first process waiting time as zero.
    - ➢ b. For each subsequent process, calculate the waiting time as the sum of the burst times of all previous processes.
    - ➢ c. Calculate the turnaround time by adding the burst time and waiting time.
- Calculate the average waiting time and average turnaround time.
- Display the process details, waiting time, and turnaround time for each process.

- Display the average waiting time and average turnaround time.
- End the program.

Code :

```
#include <iostream>
#include <algorithm>
using namespace std;

struct Process {
    int processId;
    int burstTime;
    int priority;
};

bool compareByPriority(const Process& a, const Process& b) {
    return a.priority < b.priority;
}

void calculateTimes(Process processes[], int n) {
    int waitingTime[n];
    int turnaroundTime[n];

    waitingTime[0] = 0;

    for (int i = 1; i < n; i++) {
        waitingTime[i] = processes[i - 1].burstTime + waitingTime[i - 1];
        turnaroundTime[i] = processes[i].burstTime + waitingTime[i];
    }

    cout << "Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time" << endl;
    for (int i = 0; i < n; i++) {
        cout << processes[i].processId << "\t\t" << processes[i].burstTime << "\t\t" <<
processes[i].priority << "\t\t" << waitingTime[i] << "\t\t" << turnaroundTime[i] << endl;
    }

    double averageWaitingTime = 0;
    double averageTurnaroundTime = 0;
    for (int i = 0; i < n; i++) {
        averageWaitingTime += waitingTime[i];
        averageTurnaroundTime += turnaroundTime[i];
    }
    averageWaitingTime /= n;
    averageTurnaroundTime /= n;

    cout << "Average Waiting Time: " << averageWaitingTime << endl;
    cout << "Average Turnaround Time: " << averageTurnaroundTime << endl;
}
```

```
int main() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    Process processes[n];

    cout << "Enter process IDs, burst times, and priorities:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Process " << i + 1 << " ID: ";
        cin >> processes[i].processId;
        cout << "Burst Time: ";
        cin >> processes[i].burstTime;
        cout << "Priority: ";
        cin >> processes[i].priority;
    }

    sort(processes, processes + n, compareByPriority);

    calculateTimes(processes, n);

    return 0;
}
```

<u>4)Scheduling Algorithms Simulation: RR</u>

Problem Explanation:
The task is to simulate the Round Robin (RR) scheduling algorithm. RR is a preemptive scheduling algorithm where each process is assigned a fixed time quantum (slice) for execution. If a process does not complete within the time quantum, it is moved to the back of the queue. In this simulation, we need to calculate the waiting time and turnaround time for each process.

Algorithm/Flowchart:

- Start the program.
- Accept the number of processes, burst time, and time quantum from the user.
- Initialize variables for total waiting time and total turnaround time to zero.
- Calculate the waiting time and turnaround time for each process using the RR algorithm.
    - ➢ a. Create a queue to store the processes.
    - ➢ b. Set the waiting time for all processes as zero.
    - ➢ c. Set the remaining burst time for all processes as their burst time.
    - ➢ d. Create a variable to track the current time.
    - ➢ e. While there are processes in the queue:
    - ➢ i. Dequeue the front process from the queue.
    - ➢ ii. If the remaining burst time of the process is less than or equal to the time quantum:

        - ❖ - Update the waiting time and turnaround time for the process.
        - ❖ - Increment the current time by the remaining burst time.
        - ❖ - Set the remaining burst time for the process as zero.
    - ➢ iii. If the remaining burst time of the process is greater than the time quantum:
        - ❖ - Update the waiting time for the process.
        - ❖ - Decrement the remaining burst time by the time quantum.
        - ❖ - Increment the current time by the time quantum.
        - ❖ - Enqueue the process back to the queue.
    - ➢ f. Calculate the turnaround time by adding the burst time and waiting time for each process.
- Calculate the average waiting time and average turnaround time.
- Display the process details, waiting time, and turnaround time for each process.
- Display the average waiting time and average turnaround time.
- End the program.

Code:

```
#include <iostream>
#include <queue>
using namespace std;

struct Process {
    int processId;
    int burstTime;
};

void calculateTimes(Process processes[], int n, int timeQuantum) {
    queue<int> processQueue;
    int waitingTime[n];
    int turnaroundTime[n];
    int remainingBurstTime[n];

    for (int i = 0; i < n; i++) {
        processQueue.push(i);
        waitingTime[i] = 0;
        remainingBurstTime[i] = processes[i].burstTime;
    }

    int currentTime = 0;
    while (!processQueue.empty()) {
        int frontProcess = processQueue.front();
        processQueue.pop();

        if (remainingBurstTime[frontProcess] <= timeQuantum) {
            waitingTime[frontProcess] += currentTime;
            currentTime += remainingBurstTime[frontProcess];
            remainingBurstTime[frontProcess] = 0;
        } else {
            waitingTime[frontProcess] += currentTime;
            currentTime += timeQuantum;
            remainingBurstTime[frontProcess] -= timeQuantum;
            processQueue.push(frontProcess);
        }

        if (remainingBurstTime[frontProcess] > 0) {
            processQueue.push(frontProcess);
        }
    }

    cout << "Process\tBurst Time\tWaiting Time\tTurnaround Time" << endl;
    for (int i = 0; i < n; i++) {
        turnaroundTime[i] = processes[i].burstTime + waitingTime[i];
        cout << processes[i].processId << "\t\t" << processes[i].burstTime << "\t\t" <<
waitingTime[i] << "\t\t" << turnaroundTime[i] << endl;
    }

    double averageWaitingTime = 0;
    double averageTurnaroundTime = 0;
    for (int i = 0; i < n; i++) {
        averageWaitingTime += waitingTime[i];
        averageTurnaroundTime += turnaroundTime[i];
    }
    averageWaitingTime /= n;
    averageTurnaroundTime /= n;

    cout << "Average Waiting Time: " << averageWaitingTime << endl;
    cout << "Average Turnaround Time: " << averageTurnaroundTime << endl;
}

int main() {
    int n;
    cout << "Enter the number of processes: ";
    cin >> n;

    Process processes[n];

    cout << "Enter process IDs and burst times:" << endl;
    for (int i = 0; i < n; i++) {
        cout << "Process " << i + 1 << " ID: ";
        cin >> processes[i].processId;
        cout << "Burst Time: ";
        cin >> processes[i].burstTime;
    }

    int timeQuantum;
    cout << "Enter the time quantum: ";
    cin >> timeQuantum;

    calculateTimes(processes, n, timeQuantum);

    return 0;
}
```

5)Bankers Algorithm

Problem Explanation:
The task is to simulate the Banker's algorithm, which is a deadlock avoidance algorithm used in resource allocation. The algorithm ensures that resources are allocated in a safe manner to avoid deadlocks. In this simulation, we need to determine if a system is in a safe state based on the available resources, maximum resource allocation, and current resource allocation.

Algorithm/Flowchart:

- Start the program.
- Accept the number of processes, number of resource types, maximum resource allocation, and current resource allocation from the user.
- Initialize data structures and variables for available resources, maximum resource allocation, current resource allocation, and a boolean array to track the status of processes.
- Calculate the need matrix by subtracting the current resource allocation from the maximum resource allocation for each process.
- Check the safety of the system using the Banker's algorithm.
    - ➢ a. Initialize a work array with the available resources.
    - ➢ b. Initialize a finish array to track the completion status of each process (initialize all as false).
    - ➢ c. Find a process that satisfies the condition: need <= work for all resource types.
    - ➢ d. If such a process is found, update the work array and mark the process as finished.
    - ➢ e. Repeat steps c and d until all processes are marked as finished or no process satisfies the condition.
    - ➢ f. If all processes are marked as finished, the system is in a safe state. Otherwise, it is not in a safe state.
- Display the results indicating whether the system is in a safe state or not.
- End the program.

Code :

```
#include <iostream>
#include <vector>
using namespace std;
```

```
bool isSafeState(vector<vector<int>>& need, vector<int>& available, vector<vector<int>>&
allocation) {
    int numProcesses = need.size();
    int numResources = available.size();

    vector<bool> finish(numProcesses, false);
    vector<int> work = available;

    int count = 0;
    while (count < numProcesses) {
        bool found = false;
        for (int i = 0; i < numProcesses; i++) {
            if (!finish[i]) {
                bool canExecute = true;
                for (int j = 0; j < numResources; j++) {
                    if (need[i][j] > work[j]) {
                        canExecute = false;
                        break;
                    }
                }
                if (canExecute) {
                    for (int j = 0; j < numResources; j++) {
                        work[j] += allocation[i][j];
                    }
                    finish[i] = true;
                    found = true;
                    count++;
                }
            }
        }
        if (!found) {
            break;
        }
    }

    for (bool f : finish) {
        if (!f) {
            return false;
        }
    }
    return true;
}

int main() {
    int numProcesses, numResources;
    cout << "Enter the number of processes: ";
    cin >> numProcesses;
    cout << "Enter the number of resource types: ";
    cin >> numResources;

    vector<vector<int>> maxAllocation(numProcesses, vector<int>(numResources, 0));
    vector<vector<int>> currentAllocation(numProcesses, vector<int>(numResources, 0));
    vector<vector<int>> need(numProcesses, vector<int>(numResources, 0));
    vector<int> available(numResources, 0);

    cout << "Enter the maximum resource allocation for each process:" << endl;
    for (int i = 0; i < numProcesses; i++) {
        cout << "Process " << i + 1 << ":" << endl;
        for (int j = 0; j < numResources; j++) {
            cout << "Resource " << j + 1 << ": ";
            cin >> maxAllocation[i][j];
        }
    }

    cout << "Enter the current resource allocation for each process:" << endl;
    for (int i = 0; i < numProcesses; i++) {
        cout << "Process " << i + 1 << ":" << endl;
        for (int j = 0; j < numResources; j++) {
            cout << "Resource " << j + 1 << ": ";
            cin >> currentAllocation[i][j];
        }
    }

    cout << "Enter the available resources:" << endl;
    for (int i = 0; i < numResources; i++) {
        cout << "Resource " << i + 1 << ": ";
        cin >> available[i];
    }

    // Calculate the need matrix
    for (int i = 0; i < numProcesses; i++) {
        for (int j = 0; j < numResources; j++) {
            need[i][j] = maxAllocation[i][j] - currentAllocation[i][j];
        }
    }

    bool safeState = isSafeState(need, available, currentAllocation);

    if (safeState) {
        cout << "The system is in a safe state." << endl;
    } else {
        cout << "The system is not in a safe state." << endl;
    }

    return 0;
}
```

6)Threads: Creation, Run, Synchronize

Problem Explanation:
The task is to simulate the creation, running, and synchronization of threads in a program. Threads are lightweight processes that can run concurrently within a program. In this simulation, we need to create multiple threads, run them concurrently, and synchronize their execution using synchronization mechanisms.

Algorithm/Flowchart:

- Start the program.
- Define the tasks or operations to be performed by each thread.
- Initialize any shared resources or variables that will be accessed by multiple threads.
- Create the threads and assign tasks to them.
- Implement synchronization mechanisms to coordinate the execution of threads.
    - ➢ a. Use locks, mutexes, or semaphores to control access to shared resources.
    - ➢ b. Implement thread-safe operations to avoid race conditions.
- Start the threads and let them run concurrently.
- Wait for the threads to complete their tasks.
- Perform any necessary cleanup or final operations.
- End the program.

Code :

```
#include <iostream>
#include <pthread.h>
using namespace std;

// Define shared resources or variables
int sharedVariable = 0;
pthread_mutex_t mutexLock;

// Define tasks to be performed by each thread
void* threadTask(void* threadId) {
   long tid = (long)threadId;

   // Perform thread-specific operations

   // Synchronize access to shared resources
   pthread_mutex_lock(&mutexLock);

   // Access and modify shared resources
   sharedVariable++;
   cout << "Thread " << tid << ": Shared Variable = " << sharedVariable << endl;

   // Release the lock
   pthread_mutex_unlock(&mutexLock);
```

```
    // Perform other operations

    pthread_exit(NULL);
}

int main() {
    int numThreads;
    cout << "Enter the number of threads: ";
    cin >> numThreads;

    pthread_t threads[numThreads];
    pthread_mutex_init(&mutexLock, NULL);

    // Create threads
    for (long i = 0; i < numThreads; i++) {
        int rc = pthread_create(&threads[i], NULL, threadTask, (void*)i);
        if (rc) {
            cout << "Error creating thread." << endl;
            return -1;
        }
    }

    // Wait for threads to complete
    for (int i = 0; i < numThreads; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutexLock);

    cout << "All threads have completed their tasks." << endl;

    return 0;
}
```

7)Page replacement algorithms: FIFO

Problem Explanation:
The task is to simulate the First-In-First-Out (FIFO) page replacement algorithm. Page replacement algorithms are used in operating systems to manage the allocation of memory pages. FIFO is a non-optimal page replacement algorithm where the page that has been in memory the longest is replaced. In this simulation, we need to calculate the number of page faults that occur during the execution.

Algorithm/Flowchart:
- Start the program.
- Accept the number of frames and the sequence of page references from the user.
- Initialize variables for the number of page faults, a frame array to represent the frames in memory, and a page table to track the pages in memory.

- Initialize the frame array and page table to -1 to indicate an empty frame.
- Iterate through the sequence of page references.
    - ➢ a. Check if the current page is already in memory by searching in the page table.
    - ➢ b. If the page is not found in memory (a page fault), replace the page in the oldest frame with the current page.
    - ➢ c. Update the page table and frame array with the new page.
    - ➢ d. Increment the number of page faults.
- Display the number of page faults.
- End the program.

Code :

```
#include <iostream>
#include <vector>
using namespace std;

int fifoPageReplacement(int numFrames, vector<int>& pageReferences) {
    int numPageFaults = 0;
    vector<int> frame(numFrames, -1);
    vector<int> pageTable(pageReferences.size(), -1);
    int nextFrameIndex = 0;

    for (int i = 0; i < pageReferences.size(); i++) {
        int currentPage = pageReferences[i];

        // Check if page is already in memory
        bool pageFound = false;
        for (int j = 0; j < numFrames; j++) {
            if (frame[j] == currentPage) {
                pageFound = true;
                break;
            }
        }

        if (!pageFound) {
            // Page fault: Replace page in the oldest frame
            frame[nextFrameIndex] = currentPage;
            pageTable[i] = nextFrameIndex;
            nextFrameIndex = (nextFrameIndex + 1) % numFrames;
            numPageFaults++;
        } else {
            // Page hit: Update page table
            for (int j = 0; j < numFrames; j++) {
                if (frame[j] == currentPage) {
                    pageTable[i] = j;
                    break;
                }
            }
        }
    }

    return numPageFaults;
}

int main() {
    int numFrames;
    cout << "Enter the number of frames: ";
    cin >> numFrames;

    int numPageReferences;
    cout << "Enter the number of page references: ";
    cin >> numPageReferences;

    vector<int> pageReferences(numPageReferences);
    cout << "Enter the sequence of page references: ";
    for (int i = 0; i < numPageReferences; i++) {
        cin >> pageReferences[i];
    }

    int numPageFaults = fifoPageReplacement(numFrames, pageReferences);

    cout << "Number of Page Faults: " << numPageFaults << endl;

    return 0;
}
```

<u>8)Page replacement algorithms: LIFO</u>

Problem Explanation:
The task is to simulate the Last-In-First-Out (LIFO) page replacement algorithm. Page replacement algorithms are used in operating systems to manage the allocation of memory pages. LIFO is a non-optimal page replacement algorithm where the page that has been most recently added to memory is replaced. In this simulation, we need to calculate the number of page faults that occur during the execution.

Algorithm/Flowchart:
- Start the program.
- Accept the number of frames and the sequence of page references from the user.
- Initialize variables for the number of page faults, a frame array to represent the frames in memory, and a page table to track the pages in memory.
- Initialize the frame array and page table to -1 to indicate an empty frame.
- Iterate through the sequence of page references.
    - ➢ a. Check if the current page is already in memory by searching in the page table.

    - ➢ b. If the page is not found in memory (a page fault), replace the page that was most recently added to memory with the current page.
    - ➢ c. Update the page table and frame array with the new page.
    - ➢ d. Increment the number of page faults.
- Display the number of page faults.
- End the program.

Code :

```
#include <iostream>
#include <vector>
using namespace std;

int lifoPageReplacement(int numFrames, vector<int>& pageReferences) {
    int numPageFaults = 0;
    vector<int> frame(numFrames, -1);
    vector<int> pageTable(pageReferences.size(), -1);
    int topFrameIndex = 0;

    for (int i = 0; i < pageReferences.size(); i++) {
        int currentPage = pageReferences[i];

        // Check if page is already in memory
        bool pageFound = false;
        for (int j = 0; j < numFrames; j++) {
            if (frame[j] == currentPage) {
                pageFound = true;
                break;
            }
        }

        if (!pageFound) {
            // Page fault: Replace the most recently added page
            frame[topFrameIndex] = currentPage;
            pageTable[i] = topFrameIndex;
            topFrameIndex = (topFrameIndex + 1) % numFrames;
            numPageFaults++;
        } else {
            // Page hit: Update page table
            for (int j = 0; j < numFrames; j++) {
                if (frame[j] == currentPage) {
                    pageTable[i] = j;
                    break;
                }
            }
        }
    }
    return numPageFaults;
}

int main() {
    int numFrames;
    cout << "Enter the number of frames: ";
    cin >> numFrames;

    int numPageReferences;
    cout << "Enter the number of page references: ";
    cin >> numPageReferences;

    vector<int> pageReferences(numPageReferences);
    cout << "Enter the sequence of page references: ";
    for (int i = 0; i < numPageReferences; i++) {
        cin >> pageReferences[i];
    }

    int numPageFaults = lifoPageReplacement(numFrames, pageReferences);

    cout << "Number of Page Faults: " << numPageFaults << endl;

    return 0;
}
```

<u>**9)Page replacement algorithms: LRU**</u>

Problem Explanation:
The task is to simulate the Least Recently Used (LRU) page replacement algorithm. Page replacement algorithms are used in operating systems to manage the allocation of memory pages. LRU is an optimal page replacement algorithm where the least recently used page is replaced. In this simulation, we need to calculate the number of page faults that occur during the execution.

Algorithm/Flowchart:
- Start the program.
- Accept the number of frames and the sequence of page references from the user.
- Initialize variables for the number of page faults, a frame array to represent the frames in memory, and a page table to track the pages in memory along with their last used timestamps.
- Initialize the frame array and page table to -1 and -1 respectively to indicate an empty frame and an unused timestamp.
- Iterate through the sequence of page references.
    - ➢ a. Check if the current page is already in memory by searching in the page table.
    - ➢ b. If the page is not found in memory (a page fault), replace the least recently used page with the current page.

    - ➢ c. Update the page table and frame array with the new page and update its timestamp as the current timestamp.
    - ➢ d. Increment the number of page faults.
- Display the number of page faults.
- End the program.

Code :

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

int lruPageReplacement(int numFrames, vector<int>& pageReferences) {
   int numPageFaults = 0;
   vector<int> frame(numFrames, -1);
   unordered_map<int, int> pageTable;
   int currentTimestamp = 0;

   for (int i = 0; i < pageReferences.size(); i++) {
      int currentPage = pageReferences[i];

      // Check if page is already in memory
      if (pageTable.find(currentPage) == pageTable.end()) {
         // Page fault: Replace the least recently used page
         int lruPageIndex = 0;
         int lruTimestamp = pageReferences.size() + 1;

         // Find the least recently used page in memory
         for (int j = 0; j < numFrames; j++) {
            int framePage = frame[j];
            if (pageTable.find(framePage) != pageTable.end() && pageTable[framePage] <
lruTimestamp) {
               lruPageIndex = j;
               lruTimestamp = pageTable[framePage];
            }
         }

         // Replace the least recently used page
         frame[lruPageIndex] = currentPage;
         pageTable[currentPage] = currentTimestamp;
         numPageFaults++;
      } else {
         // Page hit: Update page table with the current timestamp
         pageTable[currentPage] = currentTimestamp;
      }

      currentTimestamp++;
```

```
    }

    return numPageFaults;
}

int main() {
    int numFrames;
    cout << "Enter the number of frames: ";
    cin >> numFrames;

    int numPageReferences;
    cout << "Enter the number of page references: ";
    cin >> numPageReferences;

    vector<int> pageReferences(numPageReferences);
    cout << "Enter the sequence of page references: ";
    for (int i = 0; i < numPageReferences; i++) {
        cin >> pageReferences[i];
    }

    int numPageFaults = lruPageReplacement(numFrames, pageReferences);

    cout << "Number of Page Faults: " << numPageFaults << endl;

    return 0;
}
```

<u>10)Able and Baker problem (Two server)</u>

Problem Explanation:
The Able and Baker problem, also known as the Two Server Problem, involves simulating the service process of two servers (Able and Baker) to handle incoming tasks or customers. The goal is to track the total waiting time and service time for the tasks as they are processed by the servers.

Algorithm/Flowchart:

- Start the program.
- Accept the number of tasks and their arrival times from the user.
- Initialize variables for the total waiting time, total service time, current time, and queues for each server (Able and Baker).
- Iterate through the tasks:
    - ➢ a. Check if the current task has arrived or if any server has completed its previous task.
    - ➢ b. If the current task has arrived, add it to the appropriate server's queue.
    - ➢ c. If any server is idle, assign the next task from its queue and update the service time.
    - ➢ d. Update the waiting time for all tasks in the queues.

    - e. Increment the current time.
- Calculate the average waiting time and average service time.
- Display the average waiting time and average service time.
- End the program.

Code :
```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

struct Task {
    int id;
    int arrivalTime;
    int serviceTime;
};

void twoServerSimulation(vector<Task>& tasks, int numTasks) {
    int currentTime = 0;
    int totalWaitingTime = 0;
    int totalServiceTime = 0;

    queue<Task> ableQueue, bakerQueue;
    Task ableServer = { -1, -1, -1 };
    Task bakerServer = { -1, -1, -1 };

    for (int i = 0; i < numTasks; i++) {
        while (currentTime < tasks[i].arrivalTime || (!ableQueue.empty() && ableServer.id == -1)
||
            (!bakerQueue.empty() && bakerServer.id == -1)) {
            if (ableServer.id == -1 && !ableQueue.empty()) {
                ableServer = ableQueue.front();
                ableQueue.pop();
                totalServiceTime += ableServer.serviceTime;
            }

            if (bakerServer.id == -1 && !bakerQueue.empty()) {
                bakerServer = bakerQueue.front();
                bakerQueue.pop();
                totalServiceTime += bakerServer.serviceTime;
            }

            currentTime++;
        }

        if (ableServer.id == -1) {
            ableServer = tasks[i];
            totalServiceTime += ableServer.serviceTime;
        } else if (bakerServer.id == -1) {
            bakerServer = tasks[i];
            totalServiceTime += bakerServer.serviceTime;
        } else {
            ableQueue.push(tasks[i]);
        }

        totalWaitingTime += (currentTime - tasks[i].arrivalTime);
    }

    double avgWaitingTime = static_cast<double>(totalWaitingTime) / numTasks;
    double avgServiceTime = static_cast<double>(totalServiceTime) / numTasks;

    cout << "Average Waiting Time: " << avgWaitingTime << endl;
    cout << "Average Service Time: " << avgServiceTime << endl;
}

int main() {
    int numTasks;
    cout << "Enter the number of tasks: ";
    cin >> numTasks;

    vector<Task> tasks(numTasks);
    cout << "Enter the arrival time and service time for each task:" << endl;
    for (int i = 0; i < numTasks; i++) {
        tasks[i].id = i + 1;
        cout << "Task " << tasks[i].id << ":" << endl;
        cout << "Arrival Time: ";
        cin >> tasks[i].arrivalTime;
        cout << "Service Time: ";
        cin >> tasks[i].serviceTime;
    }

    twoServerSimulation(tasks, numTasks);

    return 0;
}
```

11)Mutual exclusion (Critical Section simulation)

Problem Explanation:
The task is to simulate the mutual exclusion problem or critical section problem. In this problem, multiple processes or threads compete for access to a critical section of code, which should only be executed by one process at a time. The goal is to implement a solution that ensures mutual exclusion, i.e., only one process can access the critical section at any given time.

Algorithm/Flowchart:

- Start the program.
- Create a shared variable or flag to represent the critical section.
- Create a shared variable or flag for each process to indicate their intent to enter the critical section.
- Initialize the shared variables and flags accordingly.
- Implement the solution for achieving mutual exclusion. One possible solution is the Peterson's algorithm:
    - ➢ a. Each process takes turns entering the critical section.
    - ➢ b. When a process wants to enter the critical section, it sets its intent flag and gives the turn to the other process.
    - ➢ c. The process only enters the critical section if the other process doesn't intend to enter or it is the process's turn.
- Simulate the execution of the processes by repeatedly trying to enter the critical section and performing some operations within it.
- Display appropriate messages to indicate when a process enters and exits the critical section.
- End the program.

Code :

```
#include <iostream>
#include <thread>
#include <atomic>
using namespace std;

atomic<bool> flag1 = false;
atomic<bool> flag2 = false;
atomic<int> turn = 1;

void process1() {
   while (true) {
      // Non-critical section
      flag1 = true;
      turn = 2;

      // Wait until process 2 is not in its critical section or it is process 1's turn
      while (flag2 && turn == 2) {
         // Busy waiting
      }

      // Critical section
      cout << "Process 1 entered the critical section" << endl;
      // Perform operations within the critical section
      cout << "Process 1 exited the critical section" << endl;

      // End condition
      flag1 = false;
```

```
        // Continue with other computations
    }
}

void process2() {
    while (true) {
        // Non-critical section
        flag2 = true;
        turn = 1;

        // Wait until process 1 is not in its critical section or it is process 2's turn
        while (flag1 && turn == 1) {
            // Busy waiting
        }

        // Critical section
        cout << "Process 2 entered the critical section" << endl;
        // Perform operations within the critical section
        cout << "Process 2 exited the critical section" << endl;

        // End condition
        flag2 = false;
        // Continue with other computations
    }
}

int main() {
    thread t1(process1);
    thread t2(process2);

    t1.join();
    t2.join();

    return 0;
}
```