LAB-TASK
On
CSE-3632
Operating Systems Lab

SUBMITTED TO —

Mohammad Zainal Abedin

Assistant Professor, Dept of CSE

SUBMITTED BY—

Name	**:**	**Md. Najmus Sakib Rafi**
Metric No	**:**	**C221060**
Section	**:**	**6BM**
Semester	**:**	**6^{th}**
Date of Submission	**:**	**17-01-2025**

LAB - 01 : FCFS[First Come First Service Scheduling]

Code implementation in CPP:

```
#include <iostream>

using namespace std;


// Function to find the waiting time for all

// processes

void calculateWaitingTime(int processIds[], int numProcesses, int burstTimes[], int waitingTimes[]) {

   // Waiting time for the first process is 0

   waitingTimes[0] = 0;


   // Calculating waiting time for each process

   for (int i = 1; i < numProcesses; i++) {

      waitingTimes[i] = burstTimes[i - 1] + waitingTimes[i - 1];

   }

}


// Function to calculate turn around time

void calculateTurnaroundTime(int processIds[], int numProcesses, int burstTimes[], int waitingTimes[],
int turnaroundTimes[]) {

   // Turnaround time is the sum of burst time and waiting time

   for (int i = 0; i < numProcesses; i++) {

      turnaroundTimes[i] = burstTimes[i] + waitingTimes[i];

   }

}
```

```
// Function to calculate average times and display process details
void calculateAndDisplayTimes(int processIds[], int numProcesses, int burstTimes[]) {
    int waitingTimes[numProcesses], turnaroundTimes[numProcesses];
    int totalWaitingTime = 0, totalTurnaroundTime = 0;

    // Calculate waiting time for all processes
    calculateWaitingTime(processIds, numProcesses, burstTimes, waitingTimes);

    // Calculate turnaround time for all processes
    calculateTurnaroundTime(processIds, numProcesses, burstTimes, waitingTimes, turnaroundTimes);

    // Display processes along with details
    cout << "Process ID  " << "Burst Time  " << "Waiting Time  " << "Turnaround Time\n";

    for (int i = 0; i < numProcesses; i++) {
        totalWaitingTime += waitingTimes[i];
        totalTurnaroundTime += turnaroundTimes[i];
        cout << "   " << processIds[i] << "\t\t" << burstTimes[i] << "\t    " << waitingTimes[i] << "\t\t  " <<
turnaroundTimes[i] << "\n";
    }

    cout << "\nAverage Waiting Time = " << (float)totalWaitingTime / (float)numProcesses;
    cout << "\nAverage Turnaround Time = " << (float)totalTurnaroundTime / (float)numProcesses << "\n";
}

// Main function
int main() {
    // Process IDs
    int processIds[] = {1, 2, 3};
```

```
    int numProcesses = sizeof(processIds) / sizeof(processIds[0]);

    // Burst times for all processes
    int burstTimes[] = {10, 5, 8};

    // Call function to calculate and display times
    calculateAndDisplayTimes(processIds, numProcesses, burstTimes);
    return 0;
}
```

Output:

```
Process ID  Burst Time  Waiting Time  Turnaround Time
  1            10           0            10
  2             5          10            15
  3             8          15            23

Average Waiting Time = 8.33333
Average Turnaround Time = 16
```

LAB - 02 : SJF[Shortest Job First]

Code Implementation in CPP:

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

// Function to find the waiting time for all
// processes
void calculateWaitingTime(int processIds[], int numProcesses, int burstTimes[], int waitingTimes[]) {
   // Waiting time for the first process is 0
   waitingTimes[0] = 0;

   // Calculating waiting time for each process
   for (int i = 1; i < numProcesses; i++) {
      waitingTimes[i] = burstTimes[i - 1] + waitingTimes[i - 1];
   }
}

// Function to calculate turn around time
void calculateTurnaroundTime(int processIds[], int numProcesses, int burstTimes[], int waitingTimes[],
int turnaroundTimes[]) {
   // Turnaround time is the sum of burst time and waiting time
   for (int i = 0; i < numProcesses; i++) {
      turnaroundTimes[i] = burstTimes[i] + waitingTimes[i];
   }
}
```

```
// Function to calculate average times and display process details
void calculateAndDisplayTimes(int processIds[], int numProcesses, int burstTimes[]) {
    int waitingTimes[numProcesses], turnaroundTimes[numProcesses];
    int totalWaitingTime = 0, totalTurnaroundTime = 0;

    // Sort processes based on burst time for SJF
    for (int i = 0; i < numProcesses - 1; i++) {
        for (int j = 0; j < numProcesses - i - 1; j++) {
            if (burstTimes[j] > burstTimes[j + 1]) {
                swap(burstTimes[j], burstTimes[j + 1]);
                swap(processIds[j], processIds[j + 1]);
            }
        }
    }

    // Calculate waiting time for all processes
    calculateWaitingTime(processIds, numProcesses, burstTimes, waitingTimes);

    // Calculate turnaround time for all processes
    calculateTurnaroundTime(processIds, numProcesses, burstTimes, waitingTimes, turnaroundTimes);

    // Display processes along with details
    cout << "Process ID  " << "Burst Time  " << "Waiting Time  " << "Turnaround Time\n";
    for (int i = 0; i < numProcesses; i++) {
        totalWaitingTime += waitingTimes[i];
        totalTurnaroundTime += turnaroundTimes[i];
        cout << "   " << processIds[i] << "\t\t" << burstTimes[i] << "\t    " << waitingTimes[i] << "\t\t  " <<
turnaroundTimes[i] << "\n";
    }
```

```
    cout << "\nAverage Waiting Time = " << (float)totalWaitingTime / (float)numProcesses;
    cout << "\nAverage Turnaround Time = " << (float)totalTurnaroundTime / (float)numProcesses << "\n";
}

// Main function
int main() {
    // Process IDs
    int processIds[] = {1, 2, 3};
    int numProcesses = sizeof(processIds) / sizeof(processIds[0]);

    // Burst times for all processes
    int burstTimes[] = {10, 5, 8};

    // Call function to calculate and display times
    calculateAndDisplayTimes(processIds, numProcesses, burstTimes);
    return 0;
}
```

Output:

```
Process ID  Burst Time  Waiting Time  Turnaround Time
  2             5          0              5
  3             8          5              13
  1             10         13             23

Average Waiting Time = 6
Average Turnaround Time = 13.6667
```

LAB – 03 : SRTF[Shortest Remaining Time First Scheduling]

Algorithm:

- Traverse until all process gets completely executed.
    - Find process with minimum remaining time at every single time lap.
    - Reduce its time by 1.
    - Check if its remaining time becomes 0
    - Increment the counter of process completion.
    - Completion time of current process = current_time + 1;
    - Calculate waiting time for each completed process.
        - **wt[i]= Completion time – arrival_time-burst_time**
    - Increment time lap by one.
- Find turnaround time (waiting_time + burst_time).

Code implementation in CPP:

```
#include<bits/stdc++.h>
using namespace std;
/// Class to hold process with arrival time(arrivalTime), burst time (burstTime), and Name(name)
class Process
{
public:
   int arrivalTime, burstTime;
   string name;
```

```
};
/// Compare function to sort priority queue in ascending order
/// If burst time of two processes are the same, place the process with lower arrival time
class Compare
{
public:
    bool operator()(Process p1, Process p2)
    {
        if (p2.burstTime < p1.burstTime)
        {
            return true;
        }
        else if (p2.burstTime == p1.burstTime
                && p2.arrivalTime < p1.arrivalTime)
        {
            return true;
        }
     return false;
    }
};
int main()
{
    cout << "Enter the number of processes: ";
    int numProcesses;
    cin >> numProcesses;
    vector<Process> processes(numProcesses);
    map<int, int> arrivalMap;
    int i = 0, maxArrivalTime = 0;
    for (auto &process : processes)
```

```
{
    cout << "Enter the arrival time, burst time, and process name of process no: " << i + 1 << ": ";
    cin >> process.arrivalTime >> process.burstTime >> process.name;
    maxArrivalTime = max(maxArrivalTime, process.arrivalTime);
    /// Maintain arrival time index
    arrivalMap[process.arrivalTime] = i + 1;
    ++i;
}
i = 0;
priority_queue<Process, vector<Process>, Compare> pq;
/// Track which process executed in each second
map<int, string> executionTimeline;
while (1)
{
    string executedProcess = "none";
    /// Push a process if it arrives at time i
    if (arrivalMap[i])
    {
        pq.push(processes[arrivalMap[i] - 1]);
    }
    if (!pq.empty())
    {
        Process currentProcess = pq.top();
        executedProcess = currentProcess.name;
        pq.pop();
        currentProcess.burstTime -= 1;
        if (currentProcess.burstTime)
            pq.push(currentProcess);
    }
```

```
        executionTimeline[i] = executedProcess;

        if (i >= maxArrivalTime && pq.empty())
            break;

        ++i;
    }
    cout << "Gantt chart: \n";
    for (auto entry : executionTimeline)
    {
        string time = to_string(entry.first);
        cout << time;
        int space = 4 - time.size();
        for (int j = 0; j < space; j++)
            cout << ' ';
    }
    cout << endl;
    for (auto entry : executionTimeline)
    {
        string processName = entry.second;
        cout << processName;
        int space = 4 - processName.size();
        for (int j = 0; j < space; j++)
            cout << ' ';
    }
    cout << endl;
    return 0;
}
```

Output:

Enter the number of processes: 3

Enter the arrival time, burst time, and process name of process no: 1: 2 3 P1

Enter the arrival time, burst time, and process name of process no: 2: 5 6 P2

Enter the arrival time, burst time, and process name of process no: 3: 5 7 P3

Gantt chart:

0	1	2	3	4	5	6	7	8	9	10	11
None	None	P1	P1	P1	P3	P3	P3	P3	P3	P3	P3

LAB - 04 : Round Robin

Algorithm:

- *Create an array **rem_bt[]** to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array)*
- *Create another array **wt[]** to store waiting times of processes. Initialize this array as 0.*
- *Initialize time : t = 0*
- *Keep traversing all the processes while they are not done. Do following for **i'th** process if it is not done yet.*
    - *If rem_bt[i] > quantum*
        - *t = t + quantum*
        - *rem_bt[i] -= quantum;*
    - *Else // Last cycle for this process*
        - *t = t + rem_bt[i];*
        - *wt[i] = t – bt[i]*
        - *rem_bt[i] = 0; // This process is over*

Once we have waiting times, we can compute turn around time tat[i] of a process as sum of waiting and burst times, i.e., wt[i] + bt[i]

Code Implementation in CPP:

```
#include<bits/stdc++.h>

using namespace std;

/// Class to hold process with arrival time(arrivalTime), burst time(burstTime), and name(processName)

class Process

{

public:

   int arrivalTime, burstTime;

   string processName;
```

```
};
int main()
{
   cout << "Enter the number of processes: ";
   int numProcesses, timeQuantum;
  cin >> numProcesses;
   vector<Process> processList(numProcesses);
   map<int, int> arrivalIndexMap;
   int currentTime = 0, maxArrivalTime = 0;

  for (int i = 0; i < numProcesses; i++)
   {
       cout << "Enter the arrival time, burst time, and process name of process no: " << i + 1 << ": ";
       cin >> processList[i].arrivalTime >> processList[i].burstTime >> processList[i].processName;
       maxArrivalTime = max(maxArrivalTime, processList[i].arrivalTime);
       /// Maintain arrival time index
       arrivalIndexMap[processList[i].arrivalTime] = i;
   }
  cout << "Enter Time Quantum: ";
   cin >> timeQuantum;

   map<int, string> ganttChart;
   queue<Process> readyQueue;
while (true)
   {
       string currentProcessName = "none";

       /// Add process to the queue if it arrives at the current time
       if (arrivalIndexMap.count(currentTime))
```

```
    {
        readyQueue.push(processList[arrivalIndexMap[currentTime]]);
    }
 if (!readyQueue.empty())
    {
        Process currentProcess = readyQueue.front();
        readyQueue.pop();

        ganttChart[currentTime] = currentProcess.processName;

        for (int t = 0; t < timeQuantum && currentProcess.burstTime > 0; t++)
        {
            currentProcess.burstTime--;
            currentTime++;
            if (arrivalIndexMap.count(currentTime))
            {
                readyQueue.push(processList[arrivalIndexMap[currentTime]]);
            }
        }
if (currentProcess.burstTime > 0)
        {
            readyQueue.push(currentProcess);
        }
    }
    else
    {
        ganttChart[currentTime] = currentProcessName;
        currentTime++;
    }
```

```
        if (currentTime > maxArrivalTime && readyQueue.empty())
        {
            break;
        }
    }

    cout << "Gantt chart: \n";
    for (auto entry : ganttChart)
    {
        string time = to_string(entry.first);
        cout << time;
        int space = 4 - time.size();
        for (int j = 0; j < space; j++)
            cout << ' ';
    }
    cout << endl;
    for (auto entry : ganttChart)
    {
        string processName = entry.second;
        cout << processName;
        int space = 4 - processName.size();
        for (int j = 0; j < space; j++)
            cout << ' ';
    }
    cout << endl;
    return 0;
}
```

Output:

Enter the number of processes: 3

Enter the arrival time, burst time, and process name of process no: 1: 3 2 P1

Enter the arrival time, burst time, and process name of process no: 2: 2 5 P2

Enter the arrival time, burst time, and process name of process no: 3: 4 5 P3

Enter Time Quantum: 2

Gantt chart:

0 1 2 4 6 8 10 12 14 15 17 18

None none P2 P1 P3 P2 P3 P3 P2 P3 P3 P3

LAB – 05 : Bankers Algorithm

Algorithm:

1. **Active**:= Running U Blocked;
   for k=1...r
   New_ request[k]:= Requested_ resources[requesting_ process, k];

2. **Simulated_ allocation**:= Allocated_ resources;
   for k=1.....r //Compute projected allocation state
   Simulated_ allocation [requesting _process, k]:= Simulated_ allocation [requesting _process, k] + New_ request[k];

3. **feasible**:= true;
   for k=1....r // Check whether projected allocation state is feasible
   if Total_ resources[k]< Simulated_ total_ alloc [k] then feasible:= false;

4. **if feasible**= true
   then // Check whether projected allocation state is a safe allocation state
   while set Active contains a process P1 such that
   For all k, Total _resources[k] - Simulated_ total_ alloc[k]>= Max_ need
   [l ,k]-Simulated_ allocation[l, k]
   Delete Pl from Active;
   for k=1.....r
   Simulated_ total_ alloc[k]:= Simulated_ total_ alloc[k]- Simulated_ allocation[l, k];

5. **If set Active is empty**
   then // Projected allocation state is a safe allocation state
   for k=1....r // Delete the request from pending requests
   Requested_ resources[requesting_ process, k]:=0;
   for k=1....r // Grant the request
   Allocated_ resources[requesting_ process, k]:= Allocated_ resources
   [requesting_ process, k] + New_ request[k];
   Total_ alloc[k]:= Total_ alloc[k] + New_ request[k];

Code implementation in CPP:

```
#include <bits/stdc++.h>
#define pb push_back
using namespace std;

int main()
{
   ///assume there is only three resources (A,B,C) and 5 processes.
int A , B , C, totall_A = 0, totall_B = 0, totall_C = 0;
cout<<"Input totall resources for A : ";
cin>>A;
cout<<"Input totall resources for B : ";
cin>>B;
cout<<"Input totall resources for C : ";
cin>>C;
vector<int> allocation[10], Max[10],Current[10];
cout<<"Allocation input : \n";
for( int i = 1; i <= 5; i++)
{
   int a , b, c;
   cout<<"Enter allocation A B C for process no "<<i<<": ";
     cin>>a>>b>>c;
     allocation[i].pb(a);
     allocation[i].pb(b);
     allocation[i].pb(c);
     totall_A += a;
     totall_B += b;
     totall_C += c;
```

```
}

cout<<"Max input : \n";
for( int i = 1; i <= 5; i++)
{
   int a , b, c;
   cout<<"Enter Max A B C for process no "<<i<<": ";
     cin>>a>>b>>c;
     Max[i].pb(a);
     Max[i].pb(b);
     Max[i].pb(c);
}
for( int i = 1; i <= 5; i++)
{
   int a , b, c;
   a =Max[i][0] - allocation[i][0];
   b =Max[i][1] - allocation[i][1];
   c =Max[i][2] - allocation[i][2];
   Current[i].pb(a);
   Current[i].pb(b);
   Current[i].pb(c);
}
int available[10] = {0};
map<int , bool>m;
available[1] = A - totall_A;
available[2] = B - totall_B;
available[3] = C - totall_C;
vector<int> ans;
int cnt = 0;
```

```
while( 1)
{
    int f = 0;
  for( int i = 1 ; i <= 5;i++)
  {
    if(m[i])
    continue;
    if(Current[i][0] <= available[1] and Current[i][1] <= available[2] and
    Current[i][2] <= available[3]  )
    {
      m[i] = 1 ;
      available[1]+= allocation[i][0];
      available[2]+= allocation[i][1];
      available[3]+= allocation[i][2];
      f = 1;
      ++cnt;
      ans.pb(i);
    }
  }

  if(!f )
  break;

}
if(cnt == 5)
{
  cout<<"answer is : ";
  for(auto a : ans)
  cout<<a<<' ';
```

```
   cout<<endl;
}
Else
cout<<"Not safe\n";
}
```

Result and output:

Input total resources for A: 3

Input total resources for B: 4

Input total resources for C: 5

Input total resources for B: Input total resources for C: Allocation input:

Enter allocation A B C for process no 1: 1 2 3

Enter allocation A B C for process no 2: 1 2 4

Enter allocation A B C for process no 3: 1 2 4

Enter allocation A B C for process no 4: 1 2 3

Enter allocation A B C for process no 5: 1 2 3 4

Max input:

Enter Max A B C for process no 1: 1 2 3

Enter Max A B C for process no 2: 1 2 4

Enter Max A B C for process no 3: 1 2 4

Enter Max A B C for process no 4: 1 4 5

Enter Max A B C for process no 5: 1 5 2

The system is not in a safe state.

LAB – 06 : [FIFO Page replacement Algorithm]

Code implementation in cpp:

```cpp
#include <iostream>

#include <queue>

#include <vector>

#include <unordered_map>


using namespace std;


class FIFO {

public:

  // Function to simulate the FIFO page replacement algorithm

  void pageReplacement(vector<int>& pages, int capacity) {

    unordered_map<int, bool> pageInMemory;

    queue<int> fifoQueue;

    int pageFaults = 0;


    for (int page : pages) {

      // If the page is not in memory, it's a page fault

      if (pageInMemory.find(page) == pageInMemory.end()) {

        pageFaults++;


        // If memory is full, remove the oldest page

        if (fifoQueue.size() == capacity) {

          int oldestPage = fifoQueue.front();

          fifoQueue.pop();

          pageInMemory.erase(oldestPage);

        }
```

```
            // Insert the new page into memory
            fifoQueue.push(page);
            pageInMemory[page] = true;
          }
        }

        cout << "Total Page Faults: " << pageFaults << endl;
    }
};

int main() {
    FIFO fifo;

    // Reference string (sequence of pages to be loaded)
    vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 4};

    // Capacity of the memory (number of pages it can hold at a time)
    int capacity = 3;

    fifo.pageReplacement(pages, capacity);

    return 0;
}
```

Output:

Total Page Faults: 11

LAB – 07 : LRU Page Replacement Algorithm

Code implementation in CPP:

```
#include <iostream>

#include <list>

#include <unordered_map>

#include <vector>


using namespace std;


class LRU {

public:

  // Function to simulate the LRU page replacement algorithm

  void pageReplacement(vector<int>& pages, int capacity) {

    unordered_map<int, list<int>::iterator> pageMap;

    list<int> lruList;

    int pageFaults = 0;


    for (int page : pages) {

      // If the page is already in memory, move it to the front (most recently used)

      if (pageMap.find(page) != pageMap.end()) {

        lruList.erase(pageMap[page]); // Remove the old occurrence

        lruList.push_front(page); // Insert at the front

        pageMap[page] = lruList.begin(); // Update the map with the new position

      } else {

        // If memory is full, remove the least recently used page

        if (lruList.size() == capacity) {

          int lruPage = lruList.back();
```

```
                lruList.pop_back(); // Remove from the back (least recently used)
                pageMap.erase(lruPage); // Remove from the map
            }
            // Insert the new page at the front (most recently used)
            lruList.push_front(page);
            pageMap[page] = lruList.begin();
            pageFaults++;
        }
    }

    cout << "Total Page Faults: " << pageFaults << endl;
  }
};

int main() {
  LRU lru;

  // Reference string (sequence of pages to be loaded)
  vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 4};

  // Capacity of the memory (number of pages it can hold at a time)
  int capacity = 3;

  lru.pageReplacement(pages, capacity);

  return 0;
}
```

Output:

Total Page Faults: 9

LAB - 08 :Second Chance Page Replacement Algorithm

Code Implementation in CPP:

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
class SecondChance {
public:
  // Function to simulate the Second Chance page replacement algorithm
  void pageReplacement(vector<int>& pages, int capacity) {
    vector<int> memory(capacity, -1);   // Memory to hold the pages
    vector<bool> referenceBit(capacity, false);  // Reference bits for the pages
    int pageFaults = 0;
    int pointer = 0;  // Pointer to track the page to be replaced

    for (int page : pages) {
      bool pageFound = false;

      // Check if the page is already in memory
      for (int i = 0; i < capacity; ++i) {
        if (memory[i] == page) {
          pageFound = true;
          referenceBit[i] = true;  // Set reference bit to true if page is found
          break;
```

```
            }
        }

        if (!pageFound) {
            // Page fault occurs, find the page to replace using the second chance algorithm
            pageFaults++;

            // Keep searching until we find a page with reference bit 0
            while (referenceBit[pointer] == true) {
                referenceBit[pointer] = false;  // Reset the reference bit
                pointer = (pointer + 1) % capacity;  // Move pointer in a circular manner
            }

            // Replace the page at pointer
            memory[pointer] = page;
            referenceBit[pointer] = true;  // Set reference bit for the new page
            pointer = (pointer + 1) % capacity;  // Move pointer in a circular manner
        }
    }

    cout << "Total Page Faults: " << pageFaults << endl;
  }
};

int main() {
  SecondChance sc;

  // Reference string (sequence of pages to be loaded)
  vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 4};
```

```
    // Capacity of the memory (number of pages it can hold at a time)
    int capacity = 3;
    sc.pageReplacement(pages, capacity);
    return 0;
}
```

Output:

Total Page Faults: 9

LAB – 09 : Memory Allocation [Best fit, First fit,Worst fit]

Code Implementation in CPP:

```cpp
#include<bits/stdc++.h>
#define all(hole) sort(hole.begin(), hole.end())
#define rall(hole) sort(hole.rbegin(), hole.rend())

using namespace std;

int main() {
   vector<int> holes, processes, best_fit_holes, first_fit_holes, worst_fit_holes;
   int num_holes;

   cout << "Enter the number of holes: ";
   cin >> num_holes;

   for (int i = 1; i <= num_holes; i++) {
      int hole_size;
      cin >> hole_size;
      holes.push_back(hole_size);
   }

   first_fit_holes = holes;
   all(holes);  // Sort holes for best fit
   worst_fit_holes = holes;
   rall(worst_fit_holes);  // Sort holes for worst fit

   int num_processes;
```

```
    cout << "Enter the number of processes: ";
    cin >> num_processes;

    for (int i = 1; i <= num_processes; i++) {
        int process_size;
        cin >> process_size;
        processes.push_back(process_size);
    }

    // First Fit
    for (int i = 0; i < processes.size(); i++) {
        cout << "Process no: " << i + 1 << endl;
        bool allocated = false;

        for (int j = 0; j < first_fit_holes.size(); j++) {
            if (processes[i] <= first_fit_holes[j]) {
                cout << "First Fit\n";
                cout << "Process " << i + 1 << " used hole of size " << first_fit_holes[j] << " and the new hole size
is " << first_fit_holes[j] - processes[i] << endl;
                allocated = true;
                first_fit_holes[j] -= processes[i];  // Update the hole size
                cout << "Available holes: \n";
                for (int k = 0; k < first_fit_holes.size(); k++) {
                    if (first_fit_holes[k] > 0)
                        cout << first_fit_holes[k] << endl;
                }
                break;
            }
        }
```

```
        if (!allocated) {
            cout << "First Fit allocation failed\n";
        }
        cout << '\n';
    }


    // Best Fit
    for (int i = 0; i < processes.size(); i++) {
        cout << "Process no: " << i + 1 << endl;
        bool allocated = false;


        for (int j = 0; j < best_fit_holes.size(); j++) {
            if (processes[i] <= best_fit_holes[j]) {
                cout << "Best Fit\n";
                cout << "Process " << i + 1 << " used hole of size " << best_fit_holes[j] << " and the new hole size
is " << best_fit_holes[j] - processes[i] << endl;
                allocated = true;
                best_fit_holes[j] -= processes[i];  // Update the hole size
                all(best_fit_holes);  // Sort holes after allocation
                cout << "Available holes: \n";
                for (int k = 0; k < best_fit_holes.size(); k++) {
                    if (best_fit_holes[k] > 0)
                        cout << best_fit_holes[k] << endl;
                }
                break;
            }
        }
        if (!allocated) {
            cout << "Best Fit allocation failed\n";
```

```
        }
        cout << '\n';
    }

    // Worst Fit
    for (int i = 0; i < processes.size(); i++) {
        cout << "Process no: " << i + 1 << endl;
        bool allocated = false;

        for (int j = 0; j < worst_fit_holes.size(); j++) {
            if (processes[i] <= worst_fit_holes[j]) {
                cout << "Worst Fit\n";
                cout << "Process " << i + 1 << " used hole of size " << worst_fit_holes[j] << " and the new hole
size is " << worst_fit_holes[j] - processes[i] << endl;
                allocated = true;
                worst_fit_holes[j] -= processes[i];  // Update the hole size
                rall(worst_fit_holes);  // Sort holes after allocation
                cout << "Available holes: \n";
                for (int k = 0; k < worst_fit_holes.size(); k++) {
                    if (worst_fit_holes[k] > 0)
                        cout << worst_fit_holes[k] << endl;
                }
                break;
            }
        }
        if (!allocated) {
            cout << "Worst Fit allocation failed\n";
        }
        cout << '\n';
```

```
    }

    return 0;
}
```

Output:

```
Enter the number of holes: 3
10 20 30
Enter the number of processes: 4
5 10 15 25

Process no: 1
First Fit
Process 1 used hole of size 10 and the new hole size is 5
Available holes:
5
20
30

Process no: 2
First Fit
Process 2 used hole of size 20 and the new hole size is 10
Available holes:
5
10
30
```

Process no: 3

First Fit

Process 3 used hole of size 30 and the new hole size is 15

Available holes:

5

10

15

Process no: 4

First Fit

Process 4 used hole of size 15 and the new hole size is 0

Available holes:

5

10