Os lab Manual

[Company name]

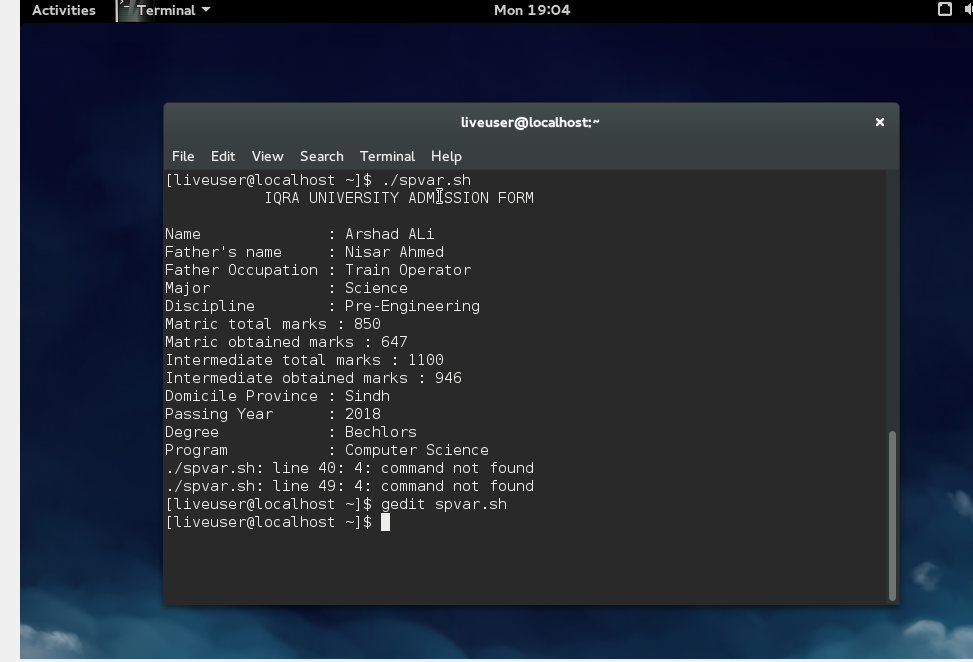
Name : Ali Abbas

Reg Id : 57451

Instructor:

Asif Ali

**Lab # 1**



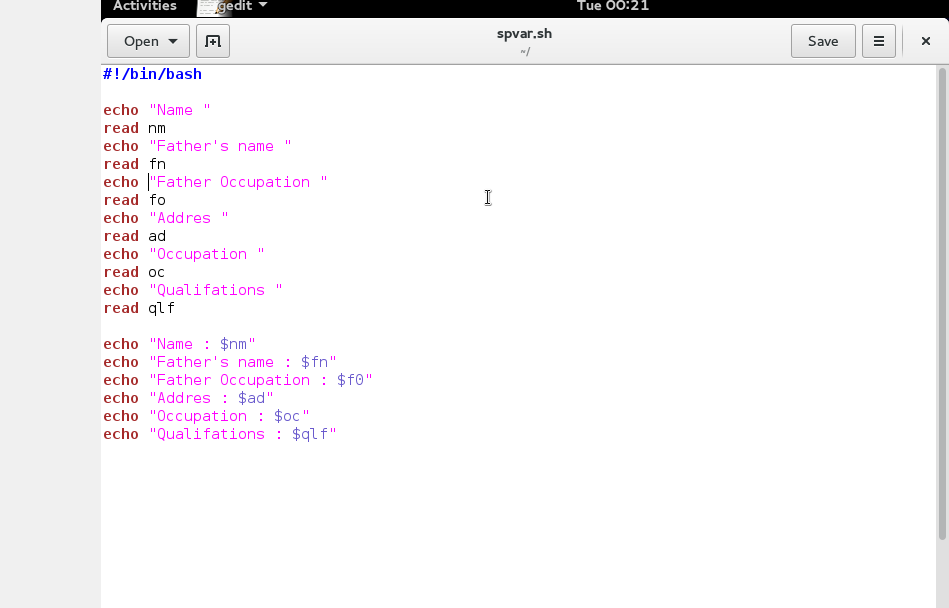
**Lab # 1**

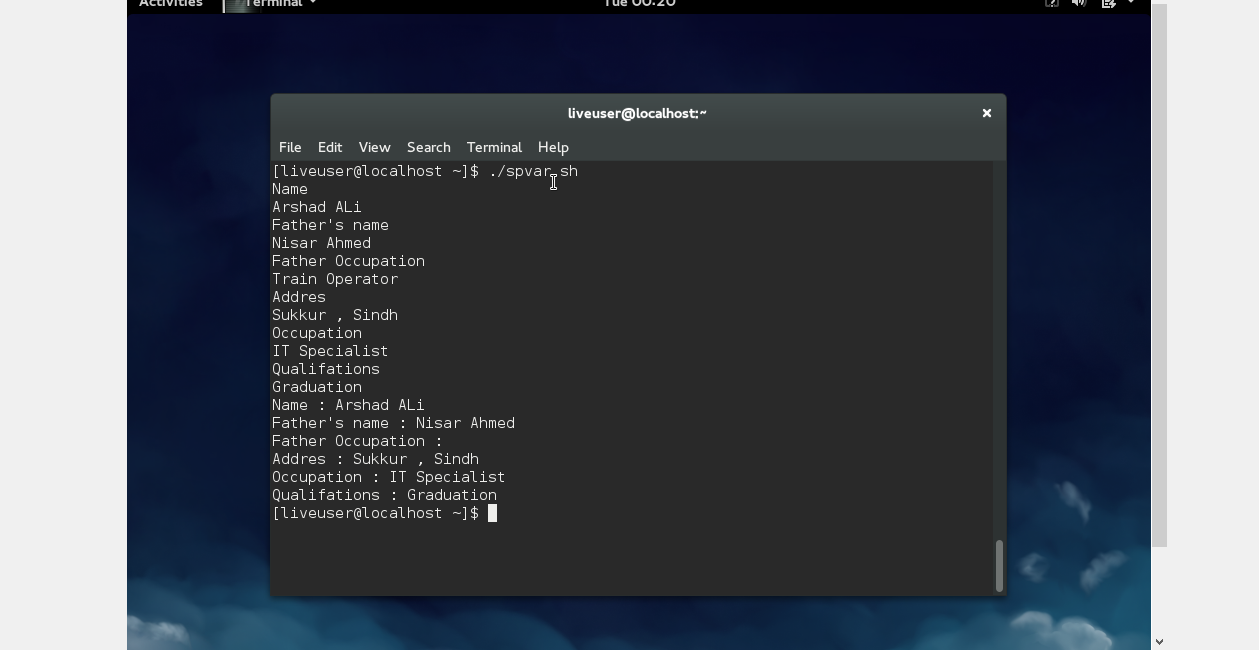
Ali Abbas



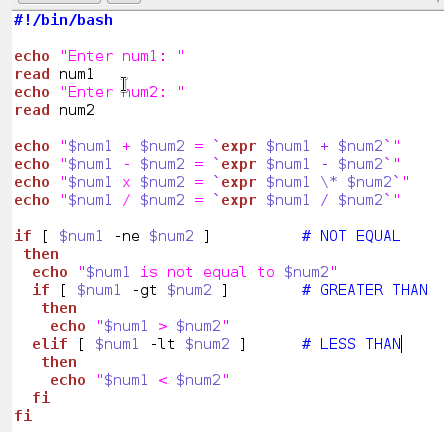
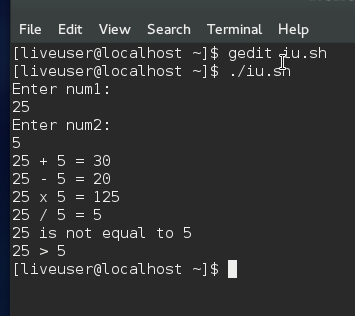
**Lab # 2**

Writing CV using special variables:



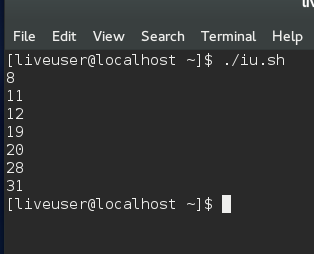


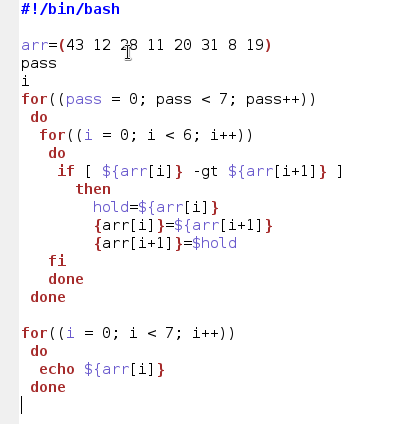
**LAB # 3**



**LAB # 4**

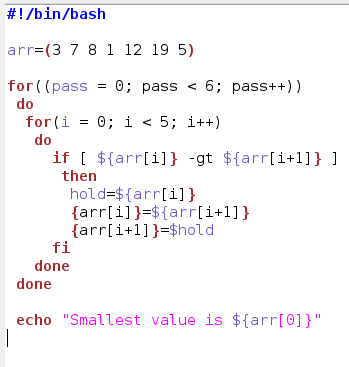
**Q1: Sort integer Array in ascending order.**

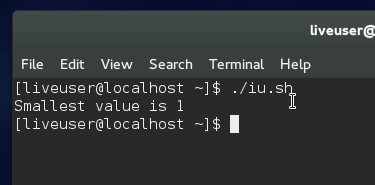
****

****

**LAB # 4**

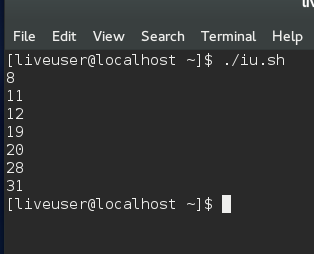
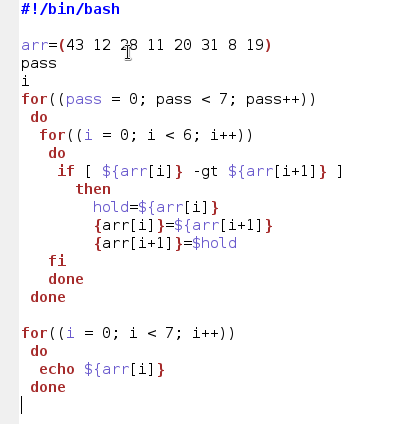
**Q2: Find shortest value in integer array.**





**LAB # 3**

**Q4:**

****

**LAB # 10**

**TASK2:**

Multi-level queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on property like process type, CPU time, IO access, memory size, etc. In a multi-level queue scheduling algorithm, there will be 'n' number of queues, where 'n' is the number of groups the processes are classified into. Each queue will be assigned a priority and will have its own scheduling algorithm like round-robin scheduling or FCFS. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the process in those high priority queues should have completed its execution. In this scheduling algorithm, once assigned to a queue, the process will not move to any other queues.

**PROGRAM**

main() {

int p[20],bt[20], su[20], wt[20],tat[20],i, k, n, temp; float wtavg, tatavg;

clrscr();

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

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

p[i] = i;

printf("Enter the Burst Time of Process %d --- ", i); scanf("%d",&bt[i]);

printf("System/User Process (0/1) ? --- "); scanf("%d", &su[i]);

}

for(i=0;i<n;i++)

for(k=i+1;k<n;k++)

if(su[i] > su[k]) {

temp=p[i];

p[i]=p[k];

p[k]=temp;

temp=bt[i];

bt[i]=bt[k];

bt[k]=temp;

temp=su[i];

su[i]=su[k];

su[k]=temp;

}

wtavg = wt[0] = 0; tatavg = tat[0] = bt[0];

for(i=1;i<n;i++) {

wt[i] = wt[i-1] + bt[i-1];tat[i] = tat[i-1] + bt[i]

wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i];

}

printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME\tTURNAROUND TIME"); for(i=0;i<n;i++)

printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],su[i],bt[i],wt[i],tat[i]);

printf("\nAverage Waiting Time is --- %f",wtavg/n);

printf("\nAverage Turnaround Time is --- %f",tatavg/n);

getch();

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***INPUT*** |  |  |  |  |
| Enter the number of processes --- 4 | | |  |  |
| Enter the Burst Time of Process 0 --- 3 | | |  |  |
| System/User Process (0/1) ? --- 1 | | |  |  |
| Enter the Burst Time of Process 1 --- 2 | | |  |  |
| System/User Process (0/1) ? --- 0 | | |  |  |
| Enter the Burst Time of Process 2 --- 5 | | |  |  |
| System/User Process (0/1) ? --- 1 | | |  |  |
| Enter the Burst Time of Process 3 --- 1 | | |  |  |
| System/User Process (0/1) ? --- 0 | | |  |  |
| ***OUTPUT*** |  |  |  |  |
| PROCESS | SYSTEM/USER PROCESS BURST TIME | | WAITING TIME | TURNAROUND TIME |
| 1 | 0 | 2 | 0 | 2 |
| 3 | 0 | 1 | 2 | 3 |
| 2 | 1 | 5 | 3 | 8 |
| 0 | 1 | 3 | 8 | 11 |
| Average Waiting Time is --- | | 3.250000 |  |  |
| Average Turnaround Time is | | --- 6.000000 |  |  |

**LAB # 11**

MEMORY MANAGEMENT TECHNIQUE (JAVA CODE)

**WORST FIT**

**package** OS\_LAB\_MANUAL;

**import** java.util.\*;

**class** MemoryBlock {

**int** startAddress;

**int** endAddress;

**int** processId;

**public** MemoryBlock(**int** startAddress, **int** endAddress) {

**this**.startAddress = startAddress;

**this**.endAddress = endAddress;

**this**.processId = -1;

}

}

**class** WorstFitMemoryAllocator {

List<MemoryBlock> memory;

**int** nextProcessId;

**public** WorstFitMemoryAllocator(**int** totalMemorySize) {

memory = **new** ArrayList<>();

memory.add(**new** MemoryBlock(0, totalMemorySize - 1));

nextProcessId = 1;

}

**public** **void** allocate(**int** processSize) {

MemoryBlock worstFitBlock = **null**;

**for** (MemoryBlock block : memory) {

**if** (block.processId == -1 && block.endAddress - block.startAddress + 1 >= processSize) {

**if** (worstFitBlock == **null** || block.endAddress - block.startAddress + 1 > worstFitBlock.endAddress - worstFitBlock.startAddress + 1) {

worstFitBlock = block;

}

}

}

**if** (worstFitBlock == **null**) {

System.***out***.println("Cannot allocate memory for process size: " + processSize);

**return**;

}

worstFitBlock.processId = nextProcessId;

nextProcessId++;

**if** (worstFitBlock.endAddress - worstFitBlock.startAddress + 1 > processSize) {

MemoryBlock newBlock = **new** MemoryBlock(worstFitBlock.startAddress + processSize, worstFitBlock.endAddress);

worstFitBlock.endAddress = worstFitBlock.startAddress + processSize - 1;

memory.add(memory.indexOf(worstFitBlock) + 1, newBlock);

}

System.***out***.println("Memory allocated for process " + (nextProcessId - 1) + " with size " + processSize);

}

**public** **void** deallocate(**int** processId) {

**boolean** deallocated = **false**;

**for** (MemoryBlock block : memory) {

**if** (block.processId == processId) {

block.processId = -1;

deallocated = **true**;

**break**;

}

}

**if** (deallocated) {

System.***out***.println("Memory deallocated for process " + processId);

} **else** {

System.***out***.println("Process " + processId + " not found in memory");

}

}

**public** **void** printMemoryState() {

System.***out***.println("Memory State:");

**for** (MemoryBlock block : memory) {

System.***out***.print(block.startAddress + "-" + block.endAddress + ": ");

**if** (block.processId == -1) {

System.***out***.println("Free");

} **else** {

System.***out***.println("Process " + block.processId);

}

}

System.***out***.println();

}

}

**package** OS\_LAB\_MANUAL;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

WorstFitMemoryAllocator allocator = **new** WorstFitMemoryAllocator(1000);

allocator.printMemoryState();

allocator.allocate(200);

allocator.printMemoryState();

allocator.allocate(400);

allocator.printMemoryState();

allocator.allocate(300);

allocator.printMemoryState();

allocator.deallocate(1);

allocator.printMemoryState();

allocator.allocate(600);

allocator.printMemoryState();

allocator.deallocate(2);

allocator.printMemoryState();

}

}

OUTPUT

Memory State:

0-999: Free

Memory allocated for process 1 with size 200

Memory State:

0-199: Process 1

200-999: Free

Memory allocated for process 2 with size 400

Memory State:

0-199: Process 1

200-599: Process 2

600-999: Free

Memory allocated for process 3 with size 300

Memory State:

0-199: Process 1

200-599: Process 2

600-899: Process 3

900-999: Free

Memory deallocated for process 1

Memory State:

0-199: Free

200-599: Process 2

600-899: Process 3

900-999: Free

Cannot allocate memory for process size: 600

Memory State:

0-199: Free

200-599: Process 2

600-899: Process 3

900-999: Free

Memory deallocated for process 2

Memory State:

0-199: Free

200-599: Free

600-899: Process 3

900-999: Free

**LAB # 11**

MEMORY MANAGEMENT TECHNIQUE (JAVA CODE)

**BEST FIT**

**package** OS\_LAB\_MANUAL;

**import** java.util.ArrayList;

**import** java.util.Comparator;

**import** java.util.List;

**import** java.util.Scanner;

**import** java.util.ArrayList;

**import** java.util.Comparator;

**import** java.util.List;

**import** java.util.Scanner;

**public** **class** BestFitAllocation {

**private** List<MemoryBlock> memory;

**public** BestFitAllocation() {

memory = **new** ArrayList<>();

}

**public** **void** initializeMemory() {

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.print("Enter the total memory size: ");

**int** totalSize = scanner.nextInt();

System.***out***.print("Enter the number of memory blocks: ");

**int** numBlocks = scanner.nextInt();

**for** (**int** i = 0; i < numBlocks; i++) {

System.***out***.print("Enter the size of memory block " + (i + 1) + ": ");

**int** blockSize = scanner.nextInt();

memory.add(**new** MemoryBlock(-1, blockSize));

}

memory.sort(Comparator.*comparingInt*(MemoryBlock::getSize));

memory.get(0).allocate();

memory.get(0).setStartAddress(0);

memory.get(0).setEndAddress(memory.get(0).getSize() - 1);

memory.sort(Comparator.*comparingInt*(MemoryBlock::getStartAddress));

System.***out***.println("Memory initialized successfully!");

scanner.close();

}

**public** **void** allocateMemory() {

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.print("Enter the process ID: ");

**int** processId = scanner.nextInt();

System.***out***.print("Enter the size of the process: ");

**int** processSize = scanner.nextInt();

MemoryBlock bestFitBlock = **null**;

**int** minFragmentation = Integer.***MAX\_VALUE***;

**for** (MemoryBlock block : memory) {

**if** (!block.isAllocated() && block.getSize() >= processSize) {

**int** fragmentation = block.getSize() - processSize;

**if** (fragmentation < minFragmentation) {

minFragmentation = fragmentation;

bestFitBlock = block;

}

}

}

**if** (bestFitBlock != **null**) {

bestFitBlock.allocate();

bestFitBlock.setEndAddress(bestFitBlock.getStartAddress() + processSize - 1);

System.***out***.println("Process " + processId + " allocated successfully in Memory Block " + memory.indexOf(bestFitBlock));

} **else** {

System.***out***.println("Insufficient memory to allocate Process " + processId);

}

scanner.close();

}

**public** **void** deallocateMemory() {

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.print("Enter the process ID to deallocate: ");

**int** processId = scanner.nextInt();

**for** (MemoryBlock block : memory) {

**if** (block.isAllocated() && block.getStartAddress() == processId) {

block.deallocate();

System.***out***.println("Process " + processId + " deallocated successfully.");

**return**;

}

}

System.***out***.println("No allocated process found with ID " + processId);

scanner.close();

}

**public** **void** printMemoryStatus() {

System.***out***.println("Memory Status:");

**for** (MemoryBlock block : memory) {

System.***out***.println(block);

}

}

**public** **static** **void** main(String[] args) {

BestFitAllocation allocator = **new** BestFitAllocation();

Scanner scanner = **new** Scanner(System.***in***);

**int** choice;

**do** {

System.***out***.println("\n----- Best-Fit Memory Allocation -----");

System.***out***.println("1. Initialize Memory");

System.***out***.println("2. Allocate Memory");

System.***out***.println("3. Deallocate Memory");

System.***out***.println("4. Print Memory Status");

System.***out***.println("0. Exit");

System.***out***.print("Enter your choice: ");

choice = scanner.nextInt();

**switch** (choice) {

**case** 1:

allocator.initializeMemory();

**break**;

**case** 2:

allocator.allocateMemory();

**break**;

**case** 3:

allocator.deallocateMemory();

**break**;

**case** 4:

allocator.printMemoryStatus();

**break**;

**case** 0:

System.***out***.println("Exiting...");

**break**;

**default**:

System.***out***.println("Invalid choice. Please try again.");

}

} **while** (choice != 0);

scanner.close();

}

}

**package** OS\_LAB\_MANUAL;

**import** java.util.ArrayList;

**import** java.util.Comparator;

**import** java.util.List;

**import** java.util.Scanner;

**class** MemoryBlock {

**private** **int** startAddress;

**private** **int** endAddress;

**private** **int** size;

**private** **boolean** allocated;

**public** MemoryBlock(**int** startAddress, **int** size) {

**this**.startAddress = startAddress;

**this**.size = size;

**this**.endAddress = startAddress + size - 1;

**this**.allocated = **false**;

}

**public** **int** getStartAddress() {

**return** startAddress;

}

**public** **int** getEndAddress() {

**return** endAddress;

}

**public** **int** getSize() {

**return** size;

}

**public** **boolean** isAllocated() {

**return** allocated;

}

**public** **void** allocate() {

allocated = **true**;

}

**public** **void** deallocate() {

allocated = **false**;

}

**public** String toString() {

**return** "Start Address: " + startAddress + ", Size: " + size + ", Allocated: " + allocated;

}

}

**LAB # 11**

MEMORY MANAGEMENT TECHNIQUE (JAVA CODE)

**FIRST FIT**

**package** OS\_LAB\_MANUAL;

**import** java.util.\*;

**class** MemoryBlock {

**int** startAddress;

**int** endAddress;

**int** processId;

**public** MemoryBlock(**int** startAddress, **int** endAddress) {

**this**.startAddress = startAddress;

**this**.endAddress = endAddress;

**this**.processId = -1;

}

}

**class** WorstFitMemoryAllocator {

List<MemoryBlock> memory;

**int** nextProcessId;

**public** WorstFitMemoryAllocator(**int** totalMemorySize) {

memory = **new** ArrayList<>();

memory.add(**new** MemoryBlock(0, totalMemorySize - 1));

nextProcessId = 1;

}

**public** **void** allocate(**int** processSize) {

MemoryBlock worstFitBlock = **null**;

**for** (MemoryBlock block : memory) {

**if** (block.processId == -1 && block.endAddress - block.startAddress + 1 >= processSize) {

**if** (worstFitBlock == **null** || block.endAddress - block.startAddress + 1 > worstFitBlock.endAddress - worstFitBlock.startAddress + 1) {

worstFitBlock = block;

}

}

}

**if** (worstFitBlock == **null**) {

System.***out***.println("Cannot allocate memory for process size: " + processSize);

**return**;

}

worstFitBlock.processId = nextProcessId;

nextProcessId++;

**if** (worstFitBlock.endAddress - worstFitBlock.startAddress + 1 > processSize) {

MemoryBlock newBlock = **new** MemoryBlock(worstFitBlock.startAddress + processSize, worstFitBlock.endAddress);

worstFitBlock.endAddress = worstFitBlock.startAddress + processSize - 1;

memory.add(memory.indexOf(worstFitBlock) + 1, newBlock);

}

System.***out***.println("Memory allocated for process " + (nextProcessId - 1) + " with size " + processSize);

}

**public** **void** deallocate(**int** processId) {

**boolean** deallocated = **false**;

**for** (MemoryBlock block : memory) {

**if** (block.processId == processId) {

block.processId = -1;

deallocated = **true**;

**break**;

}

}

**if** (deallocated) {

System.***out***.println("Memory deallocated for process " + processId);

} **else** {

System.***out***.println("Process " + processId + " not found in memory");

}

}

**public** **void** printMemoryState() {

System.***out***.println("Memory State:");

**for** (MemoryBlock block : memory) {

System.***out***.print(block.startAddress + "-" + block.endAddress + ": ");

**if** (block.processId == -1) {

System.***out***.println("Free");

} **else** {

System.***out***.println("Process " + block.processId);

}

}

System.***out***.println();

}

}

**package** OS\_LAB\_MANUAL;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

WorstFitMemoryAllocator allocator = **new** WorstFitMemoryAllocator(1000);

allocator.printMemoryState();

allocator.allocate(200);

allocator.printMemoryState();

allocator.allocate(400);

allocator.printMemoryState();

allocator.allocate(300);

allocator.printMemoryState();

allocator.deallocate(1);

allocator.printMemoryState();

allocator.allocate(600);

allocator.printMemoryState();

allocator.deallocate(2);

allocator.printMemoryState();

}

}

OUTPUT

Memory State:

0-999: Free

Memory allocated for process 1 with size 200

Memory State:

0-199: Process 1

200-999: Free

Memory allocated for process 2 with size 400

Memory State:

0-199: Process 1

200-599: Process 2

600-999: Free

Memory allocated for process 3 with size 300

Memory State:

0-199: Process 1

200-599: Process 2

600-899: Process 3

900-999: Free

Memory deallocated for process 1

Memory State:

0-199: Free

200-599: Process 2

600-899: Process 3

900-999: Free

Cannot allocate memory for process size: 600

Memory State:

0-199: Free

200-599: Process 2

600-899: Process 3

900-999: Free

Memory deallocated for process 2

Memory State:

0-199: Free

200-599: Free

600-899: Process 3

900-999: Free

**LAB # 12**

PAGING MEMORY MANAGEMENT TECHNIQUE (JAVA CODE)

**package** OS\_LAB\_MANUAL;

**import** java.util.\*;

**class** PagingSimulator {

**public** **static** **void** main(String[] args) {

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.print("Enter the total number of frames: ");

**int** totalFrames = scanner.nextInt();

System.***out***.print("Enter the total number of pages: ");

**int** totalPages = scanner.nextInt();

System.***out***.print("Enter the page reference string (comma-separated): ");

String input = scanner.next();

String[] references = input.split(",");

// Create a frame table to store the current state of frames

**int**[] frameTable = **new** **int**[totalFrames];

Arrays.*fill*(frameTable, -1); // Initialize with -1 indicating an empty frame

**int** pageFaults = 0;

// Process each page reference

**for** (String reference : references) {

**int** page = Integer.*parseInt*(reference);

// Check if the page is already in a frame

**boolean** pageFound = **false**;

**for** (**int** i = 0; i < totalFrames; i++) {

**if** (frameTable[i] == page) {

pageFound = **true**;

**break**;

}

}

**if** (!pageFound) {

// Page fault occurred, find an empty frame or replace an existing page

**int** emptyFrameIndex = -1;

**for** (**int** i = 0; i < totalFrames; i++) {

**if** (frameTable[i] == -1) {

emptyFrameIndex = i;

**break**;

}

}

**if** (emptyFrameIndex != -1) {

// Found an empty frame, assign the page to it

frameTable[emptyFrameIndex] = page;

} **else** {

// No empty frame available, replace a page using a page replacement algorithm (e.g., FIFO, LRU, etc.)

// Here, we replace the page at the first frame (index 0)

frameTable[0] = page;

}

pageFaults++;

}

// Display the current state of frame table after processing the page reference

System.***out***.println("Page Reference: " + page);

System.***out***.println("Frame Table: " + Arrays.*toString*(frameTable));

System.***out***.println();

}

System.***out***.println("Total Page Faults: " + pageFaults);

scanner.close();

}

}

OUTPUT

Enter the total number of frames: 4

Enter the total number of pages: 8

Enter the page reference string (comma-separated): 12

Page Reference: 12

Frame Table: [12, -1, -1, -1]

Total Page Faults: 1

**LAB # 13**

DEADLOCK MANAGEMENT TECHNIQUE (JAVA CODE)

FCFS

package OS\_LAB\_MANUAL;

import java.util.ArrayList;

import java.util.List;

class Resource {

private int id;

private boolean isAllocated;

public Resource(int id) {

this.id = id;

this.isAllocated = false;

}

public synchronized void allocate() {

while (isAllocated) {

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

isAllocated = true;

System.out.println("Resource " + id + " allocated.");

}

public synchronized void release() {

isAllocated = false;

System.out.println("Resource " + id + " released.");

notify();

}

}

class Process implements Runnable {

private int id;

private List<Resource> resources;

public Process(int id, List<Resource> resources) {

this.id = id;

this.resources = resources;

}

@Override

public void run() {

for (Resource resource : resources) {

resource.allocate();

// Simulate some work being done with the allocated resource

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

resource.release();

}

}

}

**package** OS\_LAB\_MANUAL;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** DeadlockManagementFCFS {

**public** **static** **void** main(String[] args) {

// Create resources

List<Resource> resources = **new** ArrayList<>();

**for** (**int** i = 1; i <= 3; i++) {

resources.add(**new** Resource(i));

}

// Create processes

Thread process1 = **new** Thread(**new** Process(1, resources));

Thread process2 = **new** Thread(**new** Process(2, resources));

// Start processes

process1.start();

process2.start();

// Wait for processes to finish

**try** {

process1.join();

process2.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

OUTPUT

Resource 1 allocated.

Resource 1 released.

Resource 2 allocated.

Resource 1 allocated.

Resource 2 released.

Resource 3 allocated.

Resource 1 released.

Resource 2 allocated.

Resource 3 released.

Resource 2 released.

Resource 3 allocated.

Resource 3 released.

**LAB # 14**

PAGE REPLACEMENT TECHNIQUE (JAVA)

FIFO

**package** OS\_LAB\_MANUAL;

**import** java.util.\*;

**public** **class** FIFOPageReplacement {

**public** **static** **void** main(String[] args) {

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.print("Enter the number of frames: ");

**int** numFrames = scanner.nextInt();

System.***out***.print("Enter the length of the reference string:");

**int** length = scanner.nextInt();

**int**[] referenceString = **new** **int**[length];

System.***out***.print("Enter the reference string: ");

**for** (**int** i = 0; i < length; i++) {

referenceString[i] = scanner.nextInt();

}

**int**[] frames = **new** **int**[numFrames];

Arrays.*fill*(frames, -1); // Initialize frames with -1 to indicate an empty frame

**int** pageFaults = 0;

**int** currentIndex = 0;

**for** (**int** i = 0; i < length; i++) {

**int** currentPage = referenceString[i];

// Check if the current page is already present in a frame

**boolean** pageHit = **false**;

**for** (**int** j = 0; j < numFrames; j++) {

**if** (frames[j] == currentPage) {

pageHit = **true**;

**break**;

}

}

// If the current page is not present in a frame, replace the oldest page (First-In)

**if** (!pageHit) {

frames[currentIndex] = currentPage;

currentIndex = (currentIndex + 1) % numFrames; // Update the index to the next frame

pageFaults++;

}

// Display the current state of frames after each reference

System.***out***.print("Frames: ");

**for** (**int** j = 0; j < numFrames; j++) {

System.***out***.print(frames[j] + " ");

}

System.***out***.println();

}

System.***out***.println("Total Page Faults: " + pageFaults);

scanner.close();

}

}

OUTPUT

Enter the number of frames: 4

Enter the length of the reference string: 2

Enter the reference string: 1

2

Frames: 1 -1 -1 -1

Frames: 1 2 -1 -1

Total Page Faults: 2

**LAB # 14**

PAGE REPLACEMENT ALGORITHM (JAVA)

LRU

**package** OS\_LAB\_MANUAL;

**import** java.util.\*;

**public** **class** LRUPageReplacement {

**public** **static** **void** main(String[] args) {

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.print("Enter the number of frames: ");

**int** numFrames = scanner.nextInt();

System.***out***.print("Enter the number of pages: ");

**int** numPages = scanner.nextInt();

System.***out***.print("Enter the page reference string: ");

**int**[] pages = **new** **int**[numPages];

**for** (**int** i = 0; i < numPages; i++) {

pages[i] = scanner.nextInt();

}

**int** numPageFaults = *performLRU*(pages, numFrames);

System.***out***.println("Number of page faults: " + numPageFaults);

scanner.close();

}

**public** **static** **int** performLRU(**int**[] pages, **int** numFrames) {

Set<Integer> frameSet = **new** HashSet<>();

Queue<Integer> frameQueue = **new** LinkedList<>();

**int** numPageFaults = 0;

**for** (**int** i = 0; i < pages.length; i++) {

**int** page = pages[i];

**if** (!frameSet.contains(page)) {

**if** (frameSet.size() == numFrames) {

**int** lruPage = frameQueue.poll();

frameSet.remove(lruPage);

}

frameSet.add(page);

frameQueue.offer(page);

numPageFaults++;

} **else** {

frameQueue.remove(page);

frameQueue.offer(page);

}

}

**return** numPageFaults;

}

}

OUTPUT

Enter the number of frames: 4

Enter the number of pages: 8

Enter the page reference string: 2

1

3

4

5

2

1

1

Number of page faults: 7

**LAB # 14**

PAGE REPLACEMENT ALGORITHM (JAVA)

LFU

package OS\_LAB\_MANUAL;

import java.util.HashMap;

import java.util.LinkedHashSet;

import java.util.Map;

import java.util.Set;

class LFUCache {

private Map<Integer, Integer> keyToVal;

private Map<Integer, Integer> keyToFreq;

private Map<Integer, LinkedHashSet<Integer>> freqToKeys;

private int capacity;

private int minFreq;

public LFUCache(int capacity) {

this.capacity = capacity;

this.minFreq = 0;

keyToVal = new HashMap<>();

keyToFreq = new HashMap<>();

freqToKeys = new HashMap<>();

}

public int get(int key) {

if (!keyToVal.containsKey(key)) {

return -1;

}

int freq = keyToFreq.get(key);

keyToFreq.put(key, freq + 1);

freqToKeys.get(freq).remove(key);

if (freq == minFreq && freqToKeys.get(freq).isEmpty()) {

minFreq++;

}

freqToKeys.computeIfAbsent(freq + 1, k -> new LinkedHashSet<>()).add(key);

return keyToVal.get(key);

}

public void put(int key, int value) {

if (capacity <= 0) {

return;

}

if (keyToVal.containsKey(key)) {

keyToVal.put(key, value);

get(key); // Update the frequency of the key

return;

}

if (keyToVal.size() >= capacity) {

int evictKey = freqToKeys.get(minFreq).iterator().next();

freqToKeys.get(minFreq).remove(evictKey);

keyToVal.remove(evictKey);

keyToFreq.remove(evictKey);

}

keyToVal.put(key, value);

keyToFreq.put(key, 1);

freqToKeys.computeIfAbsent(1, k -> new LinkedHashSet<>()).add(key);

minFreq = 1;

}

}

**package** OS\_LAB\_MANUAL;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

LFUCache cache = **new** LFUCache(2);

cache.put(1, 1);

cache.put(2, 2);

System.***out***.println(cache.get(1)); // Output: 1

cache.put(3, 3);

System.***out***.println(cache.get(2)); // Output: -1 (not found)

System.***out***.println(cache.get(3)); // Output: 3

cache.put(4, 4);

System.***out***.println(cache.get(1)); // Output: -1 (not found)

System.***out***.println(cache.get(3)); // Output: 3

System.***out***.println(cache.get(4)); // Output: 4

}

}

OUTPUT

1

-1

3

-1

3

4