Operating System Lab Manual

# Lab 1

## Program to create Single and Multi Threaded Process

**Objective:**

• To study the implementation on single thread and multi-threaded process in Linux using C

**Background**

**Algorithm**

**Source code:**

*#include <stdio.h>*

*void printNumbers() {*

*for (int i = 1; i <= 5; i++) {*

*printf("%d\n", i);*

*}*

*}*

*int main() {*

*printf("Single-threaded program:\n");*

*printNumbers();*

*printf("Program execution completed.\n");*

*return 0;*

*}*

**Actual Output:**

**Findings:**

In the above example, the program simply prints numbers from 1 to 5 using a printNumbers function. It's a single-threaded program because there is only one execution flow, and the numbers are printed sequentially.

**Source code:**

*#include <stdio.h>*

*#include <pthread.h>*

*void \*printNumbers(void \*arg) {*

*int threadID = \*(int \*)arg;*

*for (int i = 1; i <= 5; i++) {*

*printf("Thread %d: %d\n", threadID, i);*

*}*

*return NULL;*

*}*

*int main() {*

*printf("Multi-threaded program:\n");*

*pthread\_t thread1, thread2;*

*int threadID1 = 1, threadID2 = 2;*

*pthread\_create(&thread1, NULL, printNumbers, &threadID1);*

*pthread\_create(&thread2, NULL, printNumbers, &threadID2);*

*pthread\_join(thread1, NULL);*

*pthread\_join(thread2, NULL);*

*printf("Program execution completed.\n");*

*return 0;*

*}*

**Actual Output:**

**Findings**

In the multi-threaded program, we use the POSIX threads library (pthread) to create two threads. Each thread executes the printNumbers function, which prints numbers from 1 to 5 along with the associated thread ID. The pthread\_join function is used to wait for both threads to complete before the program exits. As a result, the numbers may be printed out of order due to the concurrent execution of multiple threads.

Please note that the actual output of multi-threaded programs can vary due to the nature of concurrent execution.

# Lab 2

## Implementation of Priority Scheduling algorithm

**Objective**

• To write a C program for implementation of Priority scheduling algorithm

**Background**

**Algorithm**

**Source Code**

*#include <stdio.h>*

*#define MAX\_PROCESS 10*

*struct Process {*

*int processId;*

*int burstTime;*

*int priority;*

*};*

*void swapProcesses(struct Process \*p1, struct Process \*p2) {*

*struct Process temp = \*p1;*

*\*p1 = \*p2;*

*\*p2 = temp;*

*}*

*void priorityScheduling(struct Process processes[], int n) {*

*// Sort processes based on priority (in ascending order)*

*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) {*

*swapProcesses(&processes[j], &processes[j + 1]);*

*}*

*}*

*}*

*int waitingTime[MAX\_PROCESS], turnaroundTime[MAX\_PROCESS], completionTime[MAX\_PROCESS];*

*float averageWaitingTime = 0, averageTurnaroundTime = 0;*

*// Calculate waiting time, turnaround time, and completion time for each process*

*completionTime[0] = processes[0].burstTime;*

*turnaroundTime[0] = completionTime[0];*

*waitingTime[0] = 0;*

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

*waitingTime[i] = completionTime[i - 1];*

*completionTime[i] = waitingTime[i] + processes[i].burstTime;*

*turnaroundTime[i] = completionTime[i];*

*}*

*// Calculate total waiting time and total turnaround time*

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

*averageWaitingTime += waitingTime[i];*

*averageTurnaroundTime += turnaroundTime[i];*

*}*

*// Calculate average waiting time and average turnaround time*

*averageWaitingTime /= n;*

*averageTurnaroundTime /= n;*

*// Print the scheduling results*

*printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");*

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

*printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].processId, processes[i].burstTime,*

*processes[i].priority, waitingTime[i], turnaroundTime[i]);*

*}*

*printf("\nAverage Waiting Time: %.2f", averageWaitingTime);*

*printf("\nAverage Turnaround Time: %.2f\n", averageTurnaroundTime);*

*}*

*int main() {*

*int n;*

*struct Process processes[MAX\_PROCESS];*

*printf("Enter the number of processes (up to %d): ", MAX\_PROCESS);*

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

*if (n > 0 && n <= MAX\_PROCESS) {*

*printf("Enter process details:\n");*

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

*printf("Process %d\n", i + 1);*

*processes[i].processId = i + 1;*

*printf("Enter Burst Time: ");*

*scanf("%d", &processes[i].burstTime);*

*printf("Enter Priority: ");*

*scanf("%d", &processes[i].priority);*

*}*

*priorityScheduling(processes, n);*

*} else {*

*printf("Invalid number of processes. Please try again.\n");*

*}*

*return 0;*

*}*

**User Input**

*Enter the number of processes (up to 10): 3*

*Enter process details:*

*Process 1*

*Enter Burst Time: 3*

*Enter Priority: 5*

*Process 2*

*Enter Burst Time: 1*

*Enter Priority: 1*

*Process 3*

*Enter Burst Time: 3*

*Enter Priority: 3*

**Expected Output**

*Process Burst Time Priority Waiting Time Turnaround Time*

*2 1 1 0 1*

*3 3 3 1 4*

*1 3 5 4 7*

*Average Waiting Time: 1.67*

*Average Turnaround Time: 4.00*

**Actual Output**

**Findings**

In this program, we define a Process structure that represents a process with its ID, priority, and burst time. The priorityScheduling function implements the priority scheduling algorithm. It sorts the processes based on their priority in ascending order and then calculates the waiting time and turnaround time for each process. Finally, it calculates and prints the average waiting time and average turnaround time.

In the main function, we create an array of processes with their respective IDs, priorities, and burst times. We calculate the number of processes in the array (n) and then call the priorityScheduling function, passing the array and the number of processes as arguments.

# Lab 3

## Implemantation of the FCFS Scheduling Algorithm

**Objective**

* To write a C program for implementation of FCFS scheduling algorithm

**Background**

**Algorithm**

**Source Code**

*#include<stdio.h>*

*void findWaitingTime(int n, int bt[], int wt[])*

*{*

*wt[0] = 0;*

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

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

*}*

*void findTurnAroundTime(int n, int bt[], int wt[], int tat[])*

*{*

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

*tat[i] = bt[i] + wt[i];*

*}*

*void findAverageTime(int n, int bt[])*

*{*

*int wt[n], tat[n], total\_wt = 0, total\_tat = 0;*

*findWaitingTime(n, bt, wt);*

*findTurnAroundTime(n, bt, wt, tat);*

*printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");*

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

*total\_wt = total\_wt + wt[i];*

*total\_tat = total\_tat + tat[i];*

*printf("%d\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);*

*}*

*float avg\_wt = (float)total\_wt / n;*

*float avg\_tat = (float)total\_tat / n;*

*printf("\nAverage Waiting Time: %.2f", avg\_wt);*

*printf("\nAverage Turnaround Time: %.2f", 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;*

*}*

**User Input**

*Enter the number of*

*processes: 3*

*Enter the burst time for each process:*

*Process 1: 3*

*Process 2: 4*

*Process 3: 5*

**Expected Output**

*Process Burst Time Waiting Time Turnaround Time*

*1 3 0 3*

*2 4 3 7*

*3 5 7 12*

*Average Waiting Time: 3.33*

*Average Turnaround Time: 7.33%*

**Findings**

# Lab 4

## Implementation of the SJF Scheduling Algorithm

**Objective**

* To wite a C program for the implementation of the SJF Scheduling Algorithm

**Background**

**Algorithm**

**Source Code**

*#include <stdio.h>*

*#define MAX\_PROCESS 10*

*struct Process {*

*int processId;*

*int burstTime;*

*};*

*void swapProcesses(struct Process \*p1, struct Process \*p2) {*

*struct Process temp = \*p1;*

*\*p1 = \*p2;*

*\*p2 = temp;*

*}*

*void priorityScheduling(struct Process processes[], int n) {*

*// Sort processes based on priority (in ascending order)*

*for (int i = 0; i < n - 1; i++) {*

*for (int j = 0; j < n - i - 1; j++) {*

*if (processes[j].burstTime > processes[j + 1].burstTime) {*

*swapProcesses(&processes[j], &processes[j + 1]);*

*}*

*}*

*}*

*int waitingTime[MAX\_PROCESS], turnaroundTime[MAX\_PROCESS], completionTime[MAX\_PROCESS];*

*float averageWaitingTime = 0, averageTurnaroundTime = 0;*

*// Calculate waiting time, turnaround time, and completion time for each process*

*completionTime[0] = processes[0].burstTime;*

*turnaroundTime[0] = completionTime[0];*

*waitingTime[0] = 0;*

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

*waitingTime[i] = completionTime[i - 1];*

*completionTime[i] = waitingTime[i] + processes[i].burstTime;*

*turnaroundTime[i] = completionTime[i];*

*}*

*// Calculate total waiting time and total turnaround time*

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

*averageWaitingTime += waitingTime[i];*

*averageTurnaroundTime += turnaroundTime[i];*

*}*

*// Calculate average waiting time and average turnaround time*

*averageWaitingTime /= n;*

*averageTurnaroundTime /= n;*

*// Print the scheduling results*

*printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");*

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

*printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i].processId, processes[i].burstTime, waitingTime[i], turnaroundTime[i]);*

*}*

*printf("\nAverage Waiting Time: %.2f", averageWaitingTime);*

*printf("\nAverage Turnaround Time: %.2f\n", averageTurnaroundTime);*

*}*

*int main() {*

*int n;*

*struct Process processes[MAX\_PROCESS];*

*printf("Enter the number of processes (up to %d): ", MAX\_PROCESS);*

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

*if (n > 0 && n <= MAX\_PROCESS) {*

*printf("Enter process details:\n");*

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

*printf("Process %d\n", i + 1);*

*processes[i].processId = i + 1;*

*printf("Enter Burst Time: ");*

*scanf("%d", &processes[i].burstTime);*

*}*

*priorityScheduling(processes, n);*

*} else {*

*printf("Invalid number of processes. Please try again.\n");*

*}*

*return 0;*

*}*

**User Input**

*Enter the number of processes (up to 10): 3*

*Enter process details:*

*Process 1*

*Enter Burst Time: 2*

*Process 2*

*Enter Burst Time: 1*

*Process 3*

*Enter Burst Time: 5*

**Expected Output**

*Process Burst Time Waiting Time Turnaround Time*

*2 1 0 1*

*1 2 1 3*

*3 5 3 8*

*Average Waiting Time: 1.33*

*Average Turnaround Time: 4.00*

**Actual Output**

**Findings**

# Lab 5

## Implementation of the Round Robin Scheduling Algorithm

**Objective**

* To write a C program to implement the Round Robin scheduling Algorithm

**Background**

**Algorithm**

**Source Code**

*#include <stdio.h>*

*#define MAX\_PROCESS 10*

*void roundRobinScheduling(int bt[], int n, int quantum) {*

*int remainingTime[MAX\_PROCESS];*

*int currentTime = 0;*

*// Initialize remaining time for each process*

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

*remainingTime[i] = bt[i];*

*}*

*// Execute processes in a round-robin manner*

*while (1) {*

*int allProcessesComplete = 1;*

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

*if (remainingTime[i] > 0) {*

*allProcessesComplete = 0;*

*if (remainingTime[i] <= quantum) {*

*currentTime += remainingTime[i];*

*remainingTime[i] = 0;*

*} else {*

*currentTime += quantum;*

*remainingTime[i] -= quantum;*

*}*

*printf("Process %d executed for %d units.\n", i + 1, currentTime);*

*}*

*}*

*if (allProcessesComplete) {*

*break;*

*}*

*}*

*}*

*int main() {*

*int n;*

*int bt[MAX\_PROCESS];*

*int quantum;*

*printf("Enter the number of processes (up to %d): ", MAX\_PROCESS);*

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

*printf("Enter burst time for each process:\n");*

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

*printf("Process %d: ", i + 1);*

*scanf("%d", &bt[i]);*

*}*

*printf("Enter the quantum time: ");*

*scanf("%d", &quantum);*

*roundRobinScheduling(bt, n, quantum);*

*return 0;*

*}*

**User Input**

*Enter the number of processes (up to 10): 3*

*Enter burst time for each process:*

*Process 1: 2*

*Process 2: 3*

*Process 3: 4*

*Enter the quantum time:*

*1*

**Expected Output**

*Process 1 executed for 1 units.*

*Process 2 executed for 2 units.*

*Process 3 executed for 3 units.*

*Process 1 executed for 4 units.*

*Process 2 executed for 5 units.*

*Process 3 executed for 6 units.*

*Process 2 executed for 7 units.*

*Process 3 executed for 8 units.*

*Process 3 executed for 9 units.*

**Findings**

**Lab 6**

**First Fit***#include<stdio.h>*

*#define max 25*

*int main()*

*{*

*int frag[max], b[max], f[max], i, j, nb, nf, temp;*

*static int bf[max], ff[max];*

*printf("\n\tMemory Management Scheme - First Fit");*

*printf("\nEnter the number of blocks:");*

*scanf("%d", &nb);*

*printf("Enter the number of files:");*

*scanf("%d", &nf);*

*printf("\nEnter the size of the blocks:\n");*

*for(i = 1; i <= nb; i++)*

*{*

*printf("Block %d:", i);*

*scanf("%d", &b[i]);*

*}*

*printf("Enter the size of the files:\n");*

*for(i = 1; i <= nf; i++)*

*{*

*printf("File %d:", i);*

*scanf("%d", &f[i]);*

*}*

*for(i = 1; i <= nf; i++)*

*{*

*for(j = 1; j <= nb; j++)*

*{*

*if(bf[j] != 1)*

*{*

*temp = b[j] - f[i];*

*if(temp >= 0)*

*{*

*ff[i] = j;*

*break;*

*}*

*}*

*}*

*frag[i] = temp;*

*bf[ff[i]] = 1;*

*}*

*printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragmentation");*

*for(i = 1; i <= nf; i++)*

*printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);*

*return 0;*

*}*

**Lab7  
Bestfit***#include<stdio.h>*

*#define max 25*

*int main()*

*{*

*int frag[max], b[max], f[max], i, j, nb, nf, temp, best;*

*static int bf[max], ff[max];*

*printf("\n\tMemory Management Scheme - Best Fit");*

*printf("\nEnter the number of blocks:");*

*scanf("%d", &nb);*

*printf("Enter the number of files:");*

*scanf("%d", &nf);*

*printf("\nEnter the size of the blocks:\n");*

*for(i = 1; i <= nb; i++)*

*{*

*printf("Block %d:", i);*

*scanf("%d", &b[i]);*

*}*

*printf("Enter the size of the files:\n");*

*for(i = 1; i <= nf; i++)*

*{*

*printf("File %d:", i);*

*scanf("%d", &f[i]);*

*}*

*for(i = 1; i <= nf; i++)*

*{*

*best = -1;*

*for(j = 1; j <= nb; j++)*

*{*

*if(bf[j] != 1)*

*{*

*if(b[j] >= f[i])*

*{*

*if(best == -1 || b[j] < b[best])*

*{*

*best = j;*

*}*

*}*

*}*

*}*

*if(best != -1)*

*{*

*frag[i] = b[best] - f[i];*

*ff[i] = best;*

*bf[best] = 1;*

*}*

*}*

*printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragmentation");*

*for(i = 1; i <= nf; i++)*

*printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);*

*return 0;*

*}*

**Lab8  
WorstFit**

*#include<stdio.h>*

*#define max 25*

*int main()*

*{*

*int frag[max], b[max], f[max], i, j, nb, nf, temp, worst;*

*static int bf[max], ff[max];*

*printf("\n\tMemory Management Scheme - Worst Fit");*

*printf("\nEnter the number of blocks:");*

*scanf("%d", &nb);*

*printf("Enter the number of files:");*

*scanf("%d", &nf);*

*printf("\nEnter the size of the blocks:\n");*

*for(i = 1; i <= nb; i++)*

*{*

*printf("Block %d:", i);*

*scanf("%d", &b[i]);*

*}*

*printf("Enter the size of the files:\n");*

*for(i = 1; i <= nf; i++)*

*{*

*printf("File %d:", i);*

*scanf("%d", &f[i]);*

*}*

*for(i = 1; i <= nf; i++)*

*{*

*worst = -1;*

*for(j = 1; j <= nb; j++)*

*{*

*if(bf[j] != 1)*

*{*

*if(b[j] >= f[i])*

*{*

*if(worst == -1 || b[j] > b[worst])*

*{*

*worst = j;*

*}*

*}*

*}*

*}*

*if(worst != -1)*

*{*

*frag[i] = b[worst] - f[i];*

*ff[i] = worst;*

*bf[worst] = 1;*

*}*

*}*

*printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragmentation");*

*for(i = 1; i <= nf; i++)*

*printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);*

*return 0;*

*}*

**Lab 9**

**FCFS Disk Scheduling**

*#include <stdio.h>*

*int main()*

*{*

*int t[20], n, i, tohm[20], tot = 0;*

*float avhm;*

*printf("Enter the number of tracks: ");*

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

*printf("Enter the tracks to be traversed: ");*

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

*scanf("%d", &t[i]);*

*}*

*for (i = 0; i < n - 1; i++) {*

*tohm[i] = t[i + 1] - t[i];*

*if (tohm[i] < 0)*

*tohm[i] = tohm[i] \* (-1);*

*}*

*for (i = 0; i < n - 1; i++) {*

*tot += tohm[i];*

*}*

*avhm = (float)tot / n;*

*printf("Tracks traversed\tDifference between tracks\n");*

*for (i = 0; i < n - 1; i++) {*

*printf("%d\t\t\t%d\n", t[i], tohm[i]);*

*}*

*printf("\nAverage header movements: %.2f\n", avhm);*

*return 0;*

*}*

**Lab 10**

**C-SCAN***#include <stdio.h>*

*#include <stdlib.h>*

*int main()*

*{*

*int diskQueue[20]; // Maximum number of disk requests*

*int diskQueueSize;*

*int initialHeadPosition;*

*printf("Enter the number of disk requests: ");*

*scanf("%d", &diskQueueSize);*

*printf("Enter the disk requests: ");*

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

*scanf("%d", &diskQueue[i]);*

*}*

*printf("Enter the initial head position: ");*

*scanf("%d", &initialHeadPosition);*

*// Sort the disk requests in ascending order*

*for (int i = 0; i < diskQueueSize - 1; i++) {*

*for (int j = 0; j < diskQueueSize - i - 1; j++) {*

*if (diskQueue[j] > diskQueue[j + 1]) {*

*int temp = diskQueue[j];*

*diskQueue[j] = diskQueue[j + 1];*

*diskQueue[j + 1] = temp;*

*}*

*}*

*}*

*int headMovement = 0;*

*int currentHeadPosition = initialHeadPosition;*

*// Find the position of initialHeadPosition in the sorted disk queue*

*int initialPosition = 0;*

*while (diskQueue[initialPosition] < initialHeadPosition && initialPosition < diskQueueSize) {*

*initialPosition++;*

*}*

*// C-SCAN (Circular SCAN)*

*for (int i = initialPosition; i < diskQueueSize; i++) {*

*headMovement += abs(currentHeadPosition - diskQueue[i]);*

*currentHeadPosition = diskQueue[i];*

*}*

*headMovement += abs(currentHeadPosition - 99); // Move to the rightmost track (99 is the maximum track)*

*currentHeadPosition = 0; // Move to the leftmost track (0 is the minimum track)*

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

*headMovement += abs(currentHeadPosition - diskQueue[i]);*

*currentHeadPosition = diskQueue[i];*

*}*

*printf("Total head movement: %d\n", headMovement);*

*return 0;*

*}*

**Lab-11**

**Scan**

*#include <stdio.h>*

*#include <stdlib.h>*

*int main()*

*{*

*int diskQueue[20]; // Maximum number of disk requests*

*int diskQueueSize;*

*int initialHeadPosition;*

*int direction; // 0 for left, 1 for right*

*printf("Enter the number of disk requests: ");*

*scanf("%d", &diskQueueSize);*

*printf("Enter the disk requests: ");*

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

*scanf("%d", &diskQueue[i]);*

*}*

*printf("Enter the initial head position: ");*

*scanf("%d", &initialHeadPosition);*

*printf("Enter the direction (0 for left, 1 for right): ");*

*scanf("%d", &direction);*

*// Sort the disk requests in ascending order*

*for (int i = 0; i < diskQueueSize - 1; i++) {*

*for (int j = 0; j < diskQueueSize - i - 1; j++) {*

*if (diskQueue[j] > diskQueue[j + 1]) {*

*int temp = diskQueue[j];*

*diskQueue[j] = diskQueue[j + 1];*

*diskQueue[j + 1] = temp;*

*}*

*}*

*}*

*int headMovement = 0;*

*int currentHeadPosition = initialHeadPosition;*

*// Find the position of initialHeadPosition in the sorted disk queue*

*int initialPosition = 0;*

*while (diskQueue[initialPosition] < initialHeadPosition && initialPosition < diskQueueSize) {*

*initialPosition++;*

*}*

*// SCAN in the requested direction*

*if (direction == 0) { // Left*

*for (int i = initialPosition; i >= 0; i--) {*

*headMovement += abs(currentHeadPosition - diskQueue[i]);*

*currentHeadPosition = diskQueue[i];*

*}*

*headMovement += currentHeadPosition; // Move to the leftmost track*

*currentHeadPosition = 0;*

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

*headMovement += abs(currentHeadPosition - diskQueue[i]);*

*currentHeadPosition = diskQueue[i];*

*}*

*} else { // Right*

*for (int i = initialPosition; i < diskQueueSize; i++) {*

*headMovement += abs(currentHeadPosition - diskQueue[i]);*

*currentHeadPosition = diskQueue[i];*

*}*

*headMovement += abs(currentHeadPosition - 99); // Move to the rightmost track (99 is the maximum track)*

*currentHeadPosition = 99;*

*for (int i = initialPosition - 1; i >= 0; i--) {*

*headMovement += abs(currentHeadPosition - diskQueue[i]);*

*currentHeadPosition = diskQueue[i];*

*}*

*}*

*printf("Total head movement: %d\n", headMovement);*

*return 0;*

*}*