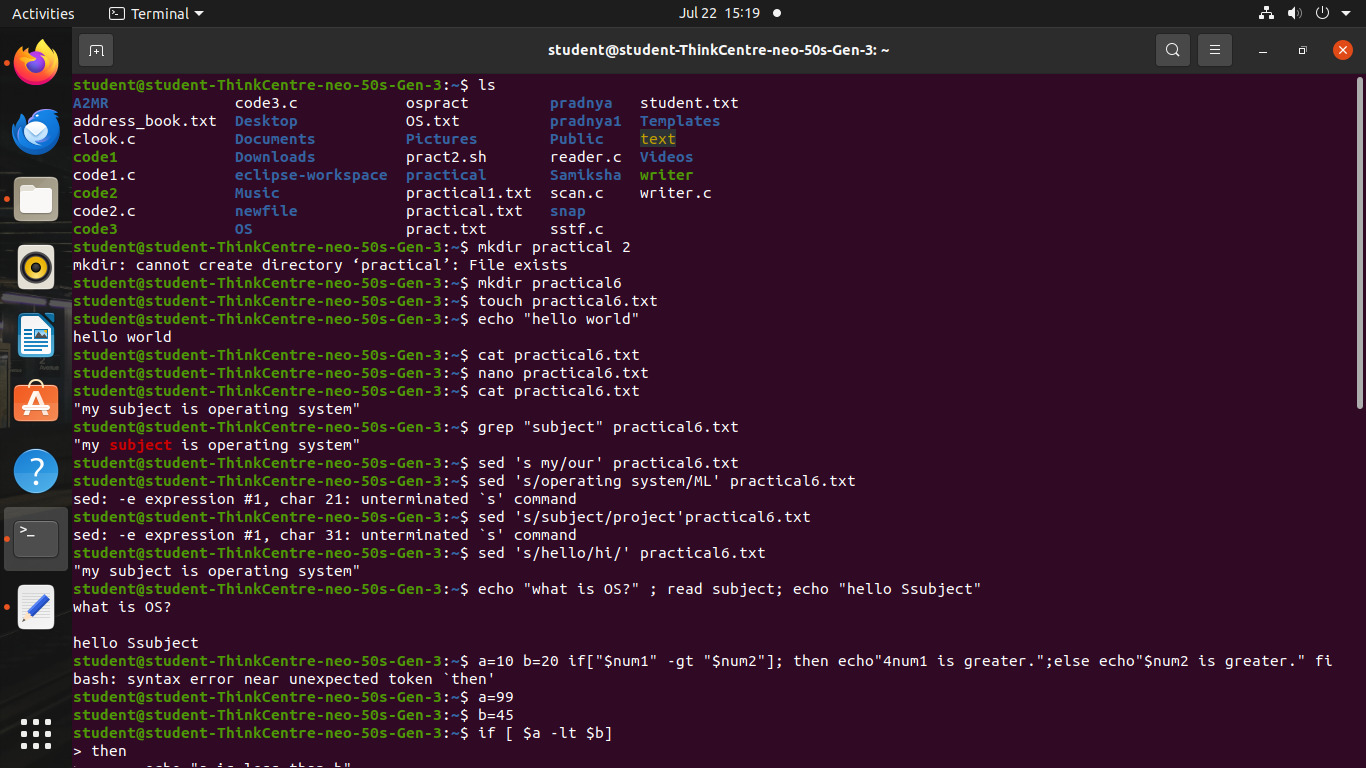
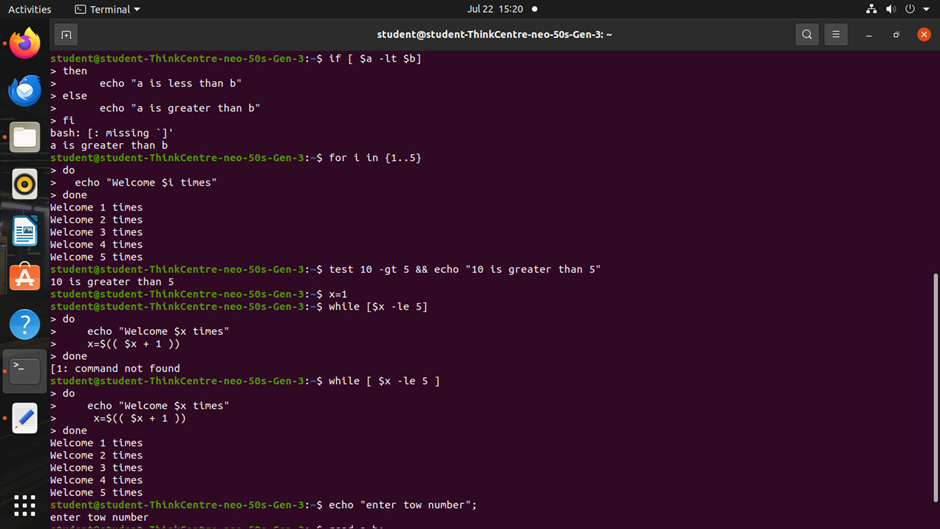
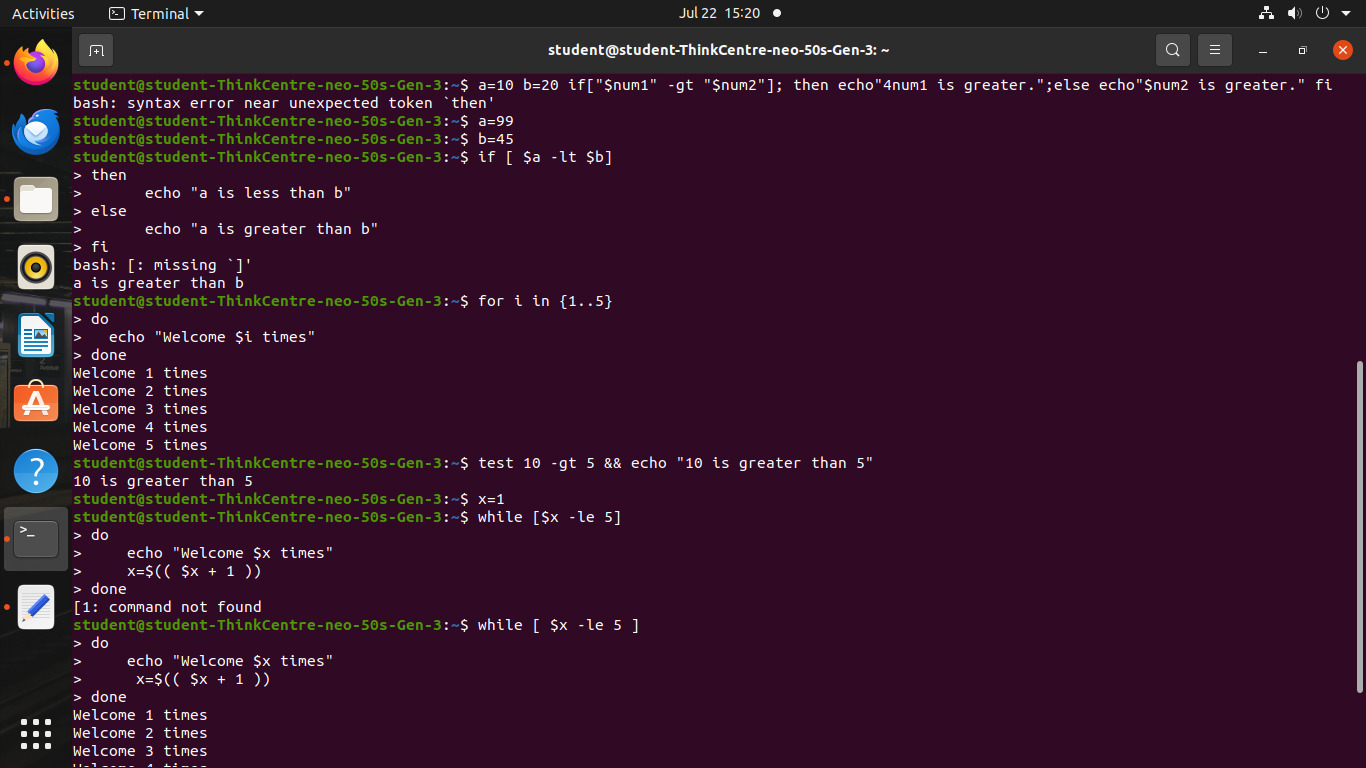
**Practical no 1 :**

1. **Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep,sed etc.**

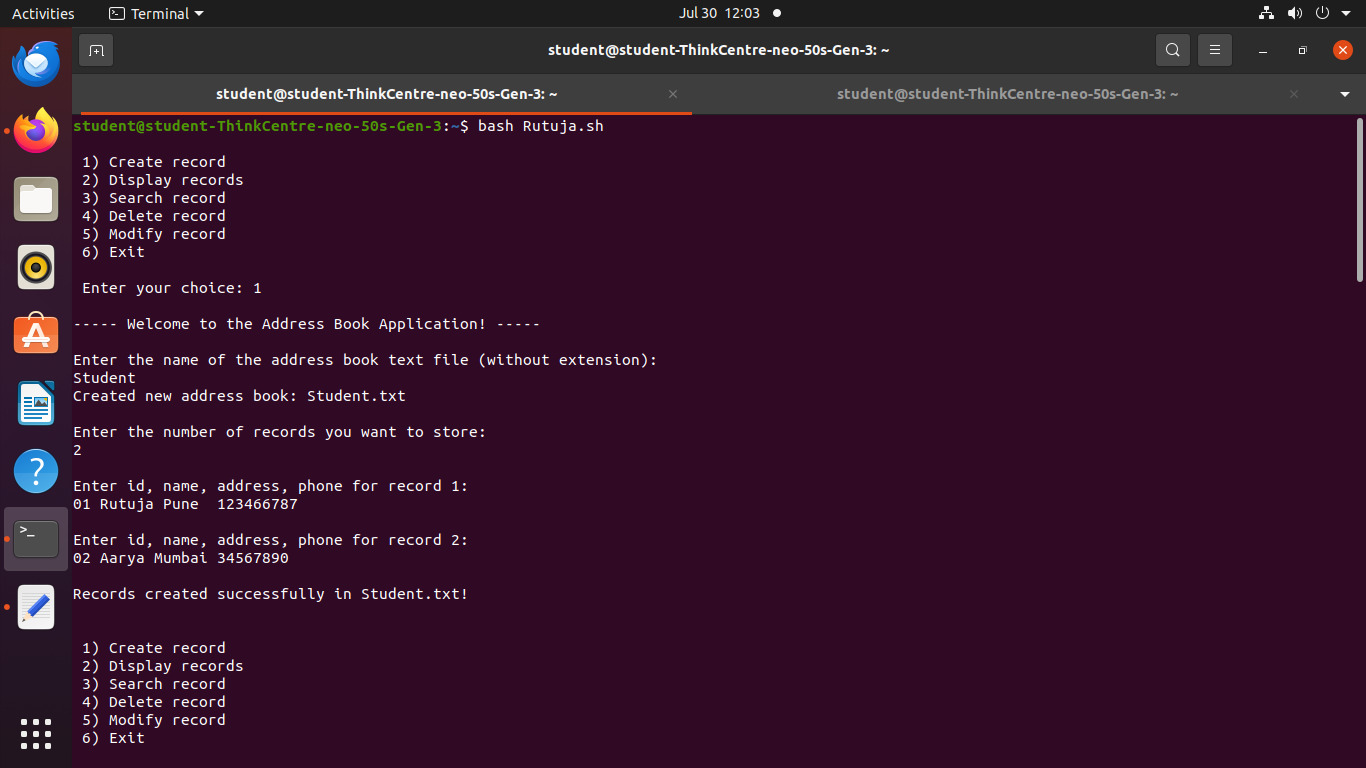
****

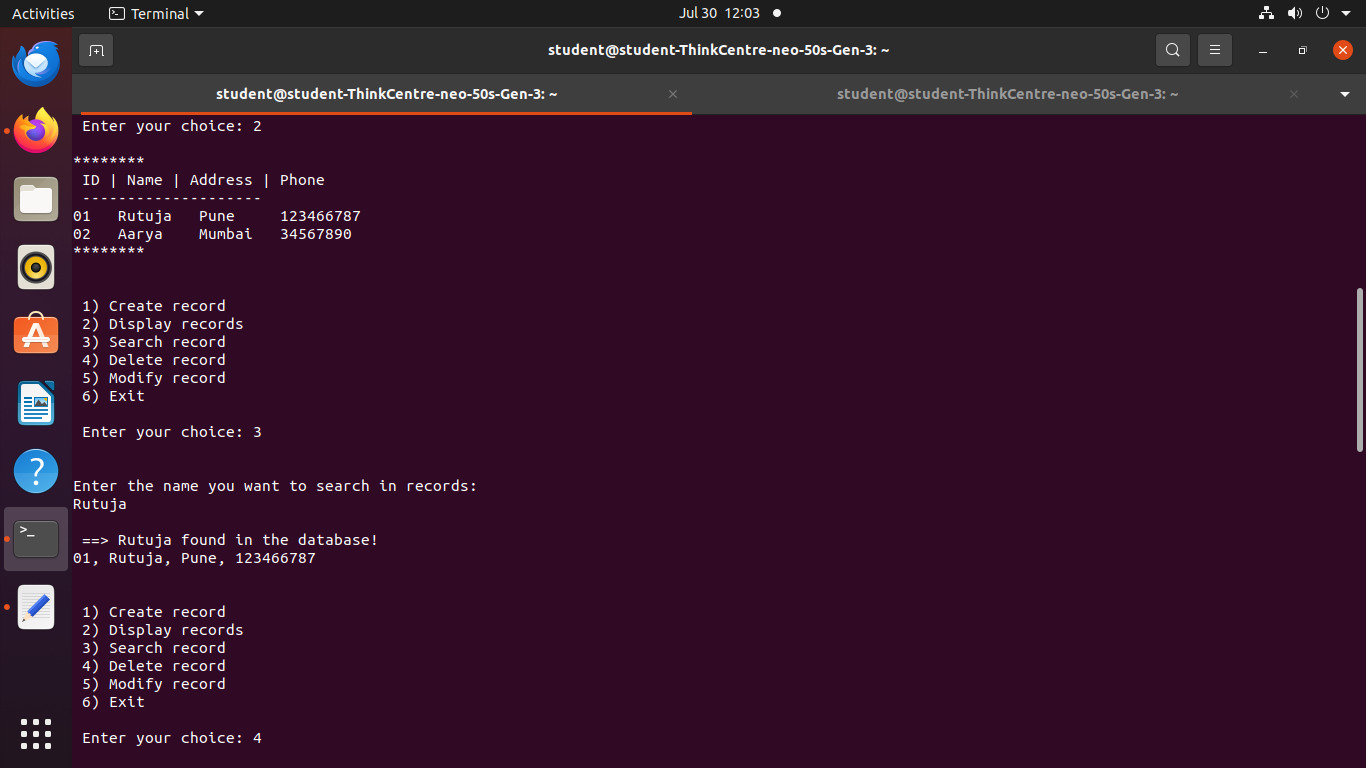
****

****

****

1. **Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit**

****

****

**Practical no 2:**

**A. Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

// Function to perform Bubble Sort

void bubbleSort(int arr[], int n) {

int i, j, temp;

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

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

if (arr[j] > arr[j+1]) {

temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

}

// Function to perform Insertion Sort

void insertionSort(int arr[], int n) {

int i, key, j;

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

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

// Function to print the array

void printArray(int arr[], int n) {

int i;

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

printf("%d ", arr[i]);

printf("\n");

}

int main() {

pid\_t pid;

int status;

int n, i;

// Accepting input

printf("Enter the number of integers: ");

scanf("%d", &n);

int arr[n];

printf("Enter the integers:\n");

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

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

}

// Demonstrating Zombie State

printf("\n--- Demonstrating Zombie State ---\n");

pid = fork();

if (pid < 0) {

// Fork failed

perror("fork");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

// Child process

printf("Child process (PID: %d) is sorting the array for zombie state.\n", getpid());

bubbleSort(arr, n);

printf("Child process sorted array:\n");

printArray(arr, n);

// Simulate some processing time

printf("Child process will now sleep for 30 seconds...\n");

sleep(30);

printf("Child process is exiting now.\n");

exit(EXIT\_SUCCESS);

} else {

// Parent process

printf("Parent process (PID: %d) is sorting the array for zombie state.\n", getpid());

bubbleSort(arr, n);

printf("Parent process sorted array:\n");

printArray(arr, n);

// Wait before checking zombie state

sleep(10); // Ensure that child process has time to become a zombie

// Add space before checking zombie state

printf("\nChecking if the child process is a zombie...\n");

if (waitpid(pid, &status, WNOHANG) == 0) {

printf("The child process is still alive and in a zombie state if parent exits next.\n");

} else {

printf("The child process is not a zombie.\n");

}

// Exit to observe the zombie state

printf("Parent process is exiting now. The child process should be a zombie.\n");

sleep(10); // Wait to observe the zombie state

}

// Clear screen

printf("\n--- Clearing Screen ---\n");

sleep(5); // Simulate some delay before the orphan demonstration

// Demonstrating Orphan State

printf("\n--- Demonstrating Orphan State ---\n");

pid = fork();

if (pid < 0) {

// Fork failed

perror("fork");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

// Child process

printf("Child process (PID: %d) is sorting the array for orphan state.\n", getpid());

insertionSort(arr, n);

printf("Child process sorted array:\n");

printArray(arr, n);

// Simulate some processing time

printf("Child process will now sleep for 30 seconds...\n");

sleep(30);

printf("Child process is exiting now.\n");

exit(EXIT\_SUCCESS);

} else {

// Parent process

printf("Parent process (PID: %d) is sorting the array for orphan state.\n", getpid());

insertionSort(arr, n);

printf("Parent process sorted array:\n");

printArray(arr, n);

// Sleep before the parent process exits

printf("Parent process will now sleep for 5 seconds before exiting...\n");

sleep(5);

// Add space before checking orphan state

printf("\nChecking if the child process has become an orphan...\n");

// Print statement to indicate parent exit and orphan state

printf("Parent process is exiting now. The child process will become an orphan.\n");

// Check if child is still running (to conβirm orphan state)

sleep(1); // Allow a moment for the child process to become an orphan

if (waitpid(pid, &status, WNOHANG) == 0) {

printf("The child process is still running, indicating it has become an orphan.\n");

} else {

printf("The child process is not running; it might have βinished already.\n");

}

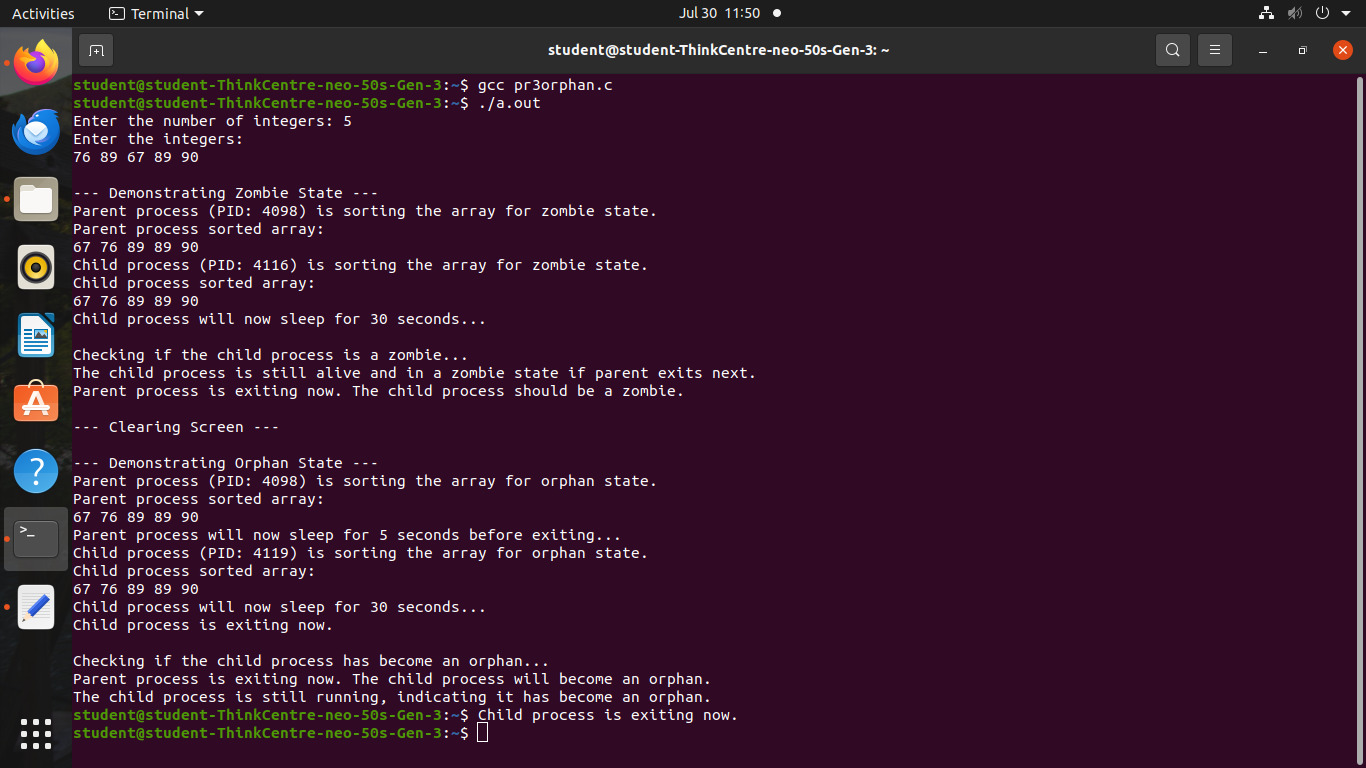
exit(EXIT\_SUCCESS);

}

return 0;

}

**Output:-**

****

**Practical no 2:**

**B. Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system callto load new program which display array in reverse order.**

**Parent code:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <string.h>

#define MAX\_SIZE 100

#define MAX\_ARG\_LEN 20

int compare(const void \*a, const void \*b) {

return ((int)a - (int)b);

}

int main() {

int arr[MAX\_SIZE];

int n;

int pipefd[2];

printf("Enter number of elements: ");

scanf("%d", &n);

if (n <= 0 || n > MAX\_SIZE) {

printf("Invalid size\n");

return 1;

}

printf("Enter %d integers:\n", n);

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

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

}

if (pipe(pipefd) == -1) {

perror("pipe");

exit(1);

}

pid\_t pid = fork();

if (pid < 0) {

perror("fork");

exit(1);

}

if (pid > 0) {

// Parent process

close(pipefd[0]); // Close read end

// Sort the array

qsort(arr, n, sizeof(int), compare);

// Write sorted array to pipe as space-separated string

char buffer[4096] = {0};

char temp[MAX\_ARG\_LEN];

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

snprintf(temp, MAX\_ARG\_LEN, "%d ", arr[i]);

strcat(buffer, temp);

}

write(pipefd[1], buffer, strlen(buffer));

close(pipefd[1]); // Close write end

wait(NULL); // Wait for child

} else {

// Child process

close(pipefd[1]); // Close write end

char buffer[4096];

int len = read(pipefd[0], buffer, sizeof(buffer) - 1);

if (len < 0) {

perror("read");

exit(1);

}

buffer[len] = '\0';

close(pipefd[0]);

// Tokenize buffer to get numbers and prepare argv for execve

char \*tokens[MAX\_SIZE + 2];

int count = 0;

tokens[count++] = "./child";

char \*token = strtok(buffer, " ");

while (token != NULL) {

tokens[count++] = token;

token = strtok(NULL, " ");

}

tokens[count] = NULL;

// execve call to load child program

if (execve("./child", tokens, NULL) == -1) {

perror("execve");

exit(1);

}

}

return 0;

}

**child.c 2(b)**

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[]) {

printf("Child received sorted numbers:\n");

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

printf("%s ", argv[i]);

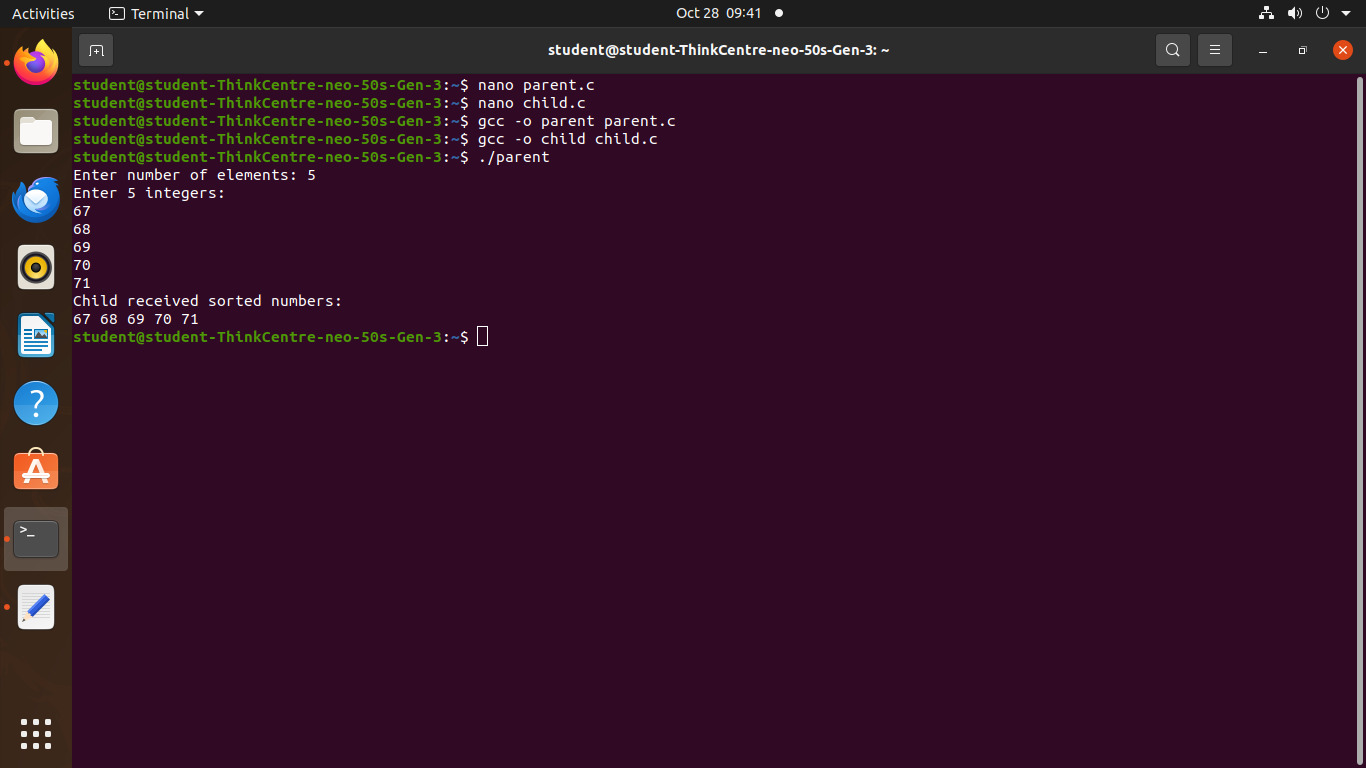
}

printf("\n");

return 0;

}

**Output:-**

****

**Practical no 3**

**Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.**

#include <stdio.h>

int main() {

int n, i, j;

printf("Enter number of processes: ");

scanf("%d", &n);

int process[n], burst[n], waiting[n], turnaround[n];

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

process[i] = i + 1;

printf("Enter burst time for process P%d: ", i + 1);

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

}

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

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

if(burst[i] > burst[j]) {

int temp = burst[i];

burst[i] = burst[j];

burst[j] = temp;

temp = process[i];

process[i] = process[j];

process[j] = temp;

}

}

}

waiting[0] = 0; // First process has no waiting time

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

waiting[i] = waiting[i - 1] + burst[i - 1];

}

// Calculate turnaround time

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

turnaround[i] = waiting[i] + burst[i];

}

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

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

printf("P%d\t%d\t\t%d\t\t%d\n", process[i], burst[i], waiting[i], turnaround[i]);

}

float avg\_wt = 0, avg\_tat = 0;

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

avg\_wt += waiting[i];

avg\_tat += turnaround[i];

}

avg\_wt /= n;

avg\_tat /= n;

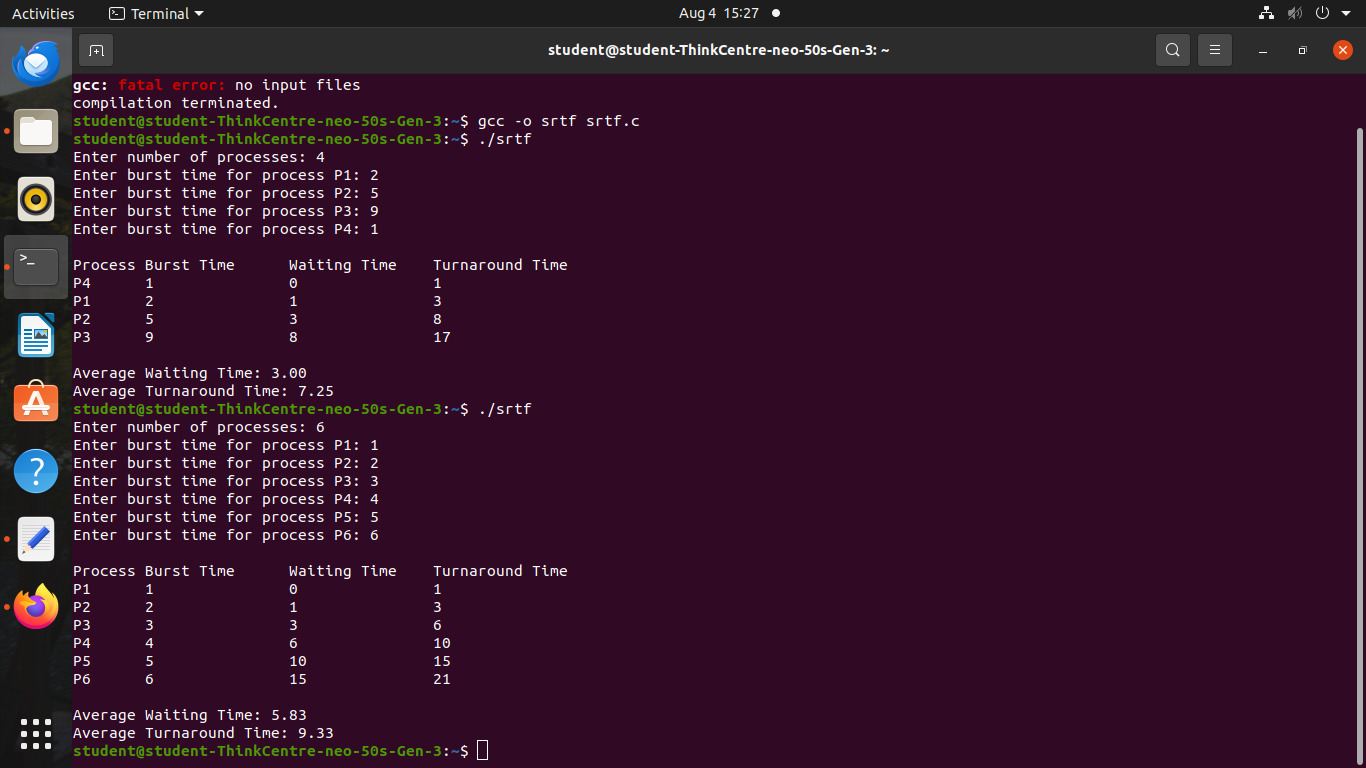
printf("\nAverage Waiting Time: %.2f", avg\_wt);

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

return 0;

}

**Output:-**

****

**Round Robin:-**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 100

typedef struct {

int pid;

int arrival;

int burst;

int remaining;

int completion;

int turnaround;

int waiting;

bool done;

} Process;

void sortByArrival(Process p[], int n) {

Process temp;

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

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

if (p[j].arrival > p[j+1].arrival) {

temp = p[j];

p[j] = p[j+1];

p[j+1] = temp;

}

}

//=========================== SJF PREEMPTIVE ==============================

void SJF\_Preemptive(Process p[], int n) {

int time = 0, completed = 0, min\_index = -1;

float avg\_tat = 0, avg\_wt = 0;

printf("\n--- SJF (Preemptive) Scheduling ---\n");

sortByArrival(p, n);

while (completed != n) {

int min\_burst = 1e9;

min\_index = -1;

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

if (p[i].arrival <= time && p[i].remaining > 0 && p[i].remaining < min\_burst) {

min\_burst = p[i].remaining;

min\_index = i;

}

}

if (min\_index != -1) {

p[min\_index].remaining--;

time++;

if (p[min\_index].remaining == 0) {

p[min\_index].completion = time;

p[min\_index].turnaround = p[min\_index].completion - p[min\_index].arrival;

p[min\_index].waiting = p[min\_index].turnaround - p[min\_index].burst;

completed++;

}

} else {

time++;

}

}

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");

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

avg\_tat += p[i].turnaround;

avg\_wt += p[i].waiting;

printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival, p[i].burst,

p[i].completion, p[i].turnaround, p[i].waiting);

}

printf("\nAverage Turnaround Time = %.2f\n", avg\_tat / n);

printf("Average Waiting Time = %.2f\n", avg\_wt / n);

}

//============================= ROUND ROBIN ===============================

void RoundRobin(Process p[], int n, int quantum) {

int time = 0, completed = 0;

float avg\_tat = 0, avg\_wt = 0;

int queue[MAX], front = 0, rear = 0;

bool in\_queue[MAX] = {false};

printf("\n--- Round Robin Scheduling ---\n");

sortByArrival(p, n);

queue[rear++] = 0;

in\_queue[0] = true;

while (completed != n) {

int idx = queue[front++];

if (p[idx].remaining > 0) {

int exec\_time = (p[idx].remaining > quantum) ? quantum : p[idx].remaining;

p[idx].remaining -= exec\_time;

time += exec\_time;

// Enqueue processes that have arrived in this time

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

if (p[i].arrival <= time && !in\_queue[i] && p[i].remaining > 0) {

queue[rear++] = i;

in\_queue[i] = true;

}

}

// If not finished, put back into queue

if (p[idx].remaining > 0)

queue[rear++] = idx;

else {

p[idx].completion = time;

p[idx].turnaround = p[idx].completion - p[idx].arrival;

p[idx].waiting = p[idx].turnaround - p[idx].burst;

completed++;

}

// If queue is empty, add next arriving process

if (front == rear) {

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

if (p[i].remaining > 0) {

queue[rear++] = i;

in\_queue[i] = true;

break;

}

}

}

}

}

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");

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

avg\_tat += p[i].turnaround;

avg\_wt += p[i].waiting;

printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival, p[i].burst,

p[i].completion, p[i].turnaround, p[i].waiting);

}

printf("\nAverage Turnaround Time = %.2f\n", avg\_tat / n);

printf("Average Waiting Time = %.2f\n", avg\_wt / n);

}

//============================= MAIN FUNCTION ==============================

int main() {

Process p[MAX], p1[MAX], p2[MAX];

int n, choice, quantum;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter Arrival Time and Burst Time for each process:\n");

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

p[i].pid = i + 1;

printf("Process P%d Arrival Time: ", i + 1);

scanf("%d", &p[i].arrival);

printf("Process P%d Burst Time: ", i + 1);

scanf("%d", &p[i].burst);

p[i].remaining = p[i].burst;

p[i].done = false;

}

// Copy for both algorithms

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

p1[i] = p[i];

p2[i] = p[i];

}

SJF\_Preemptive(p1, n);

printf("\nEnter Time Quantum for Round Robin: ");

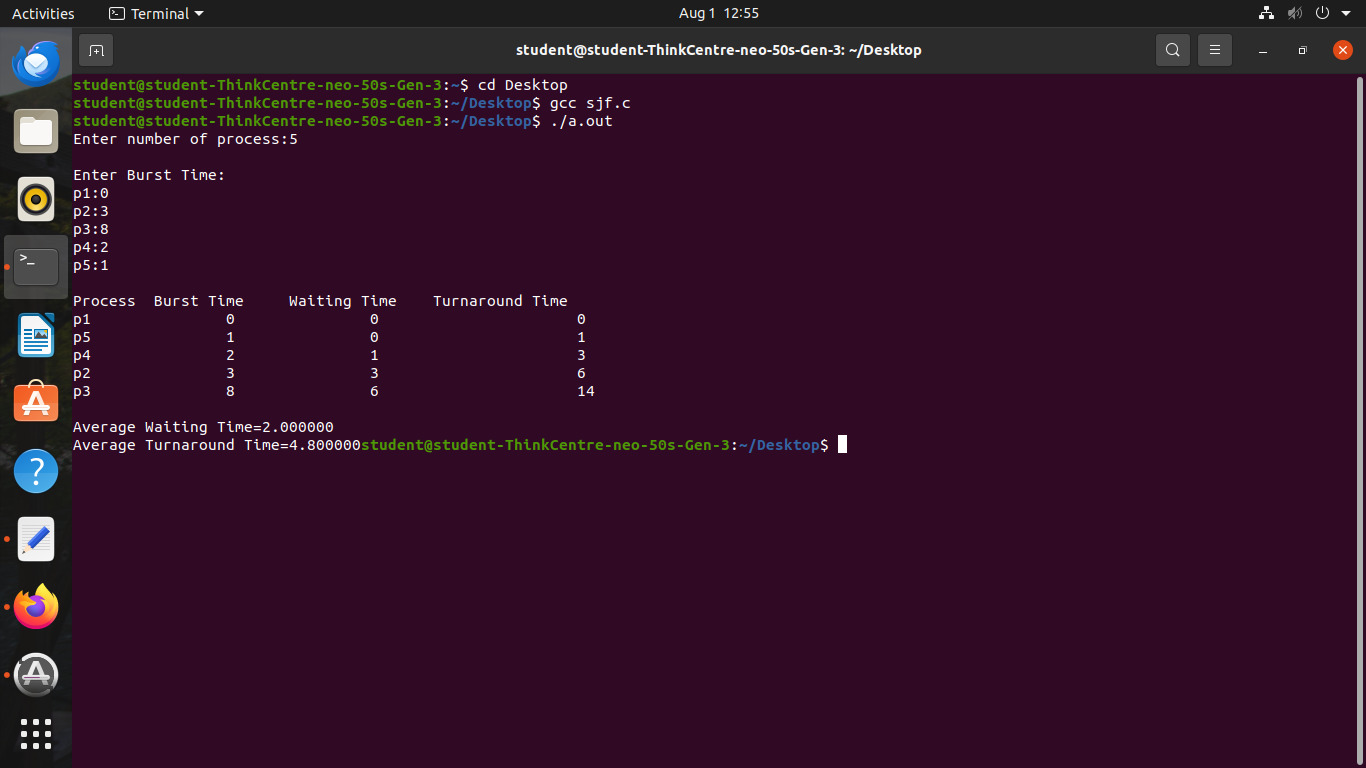
scanf("%d", &quantum);

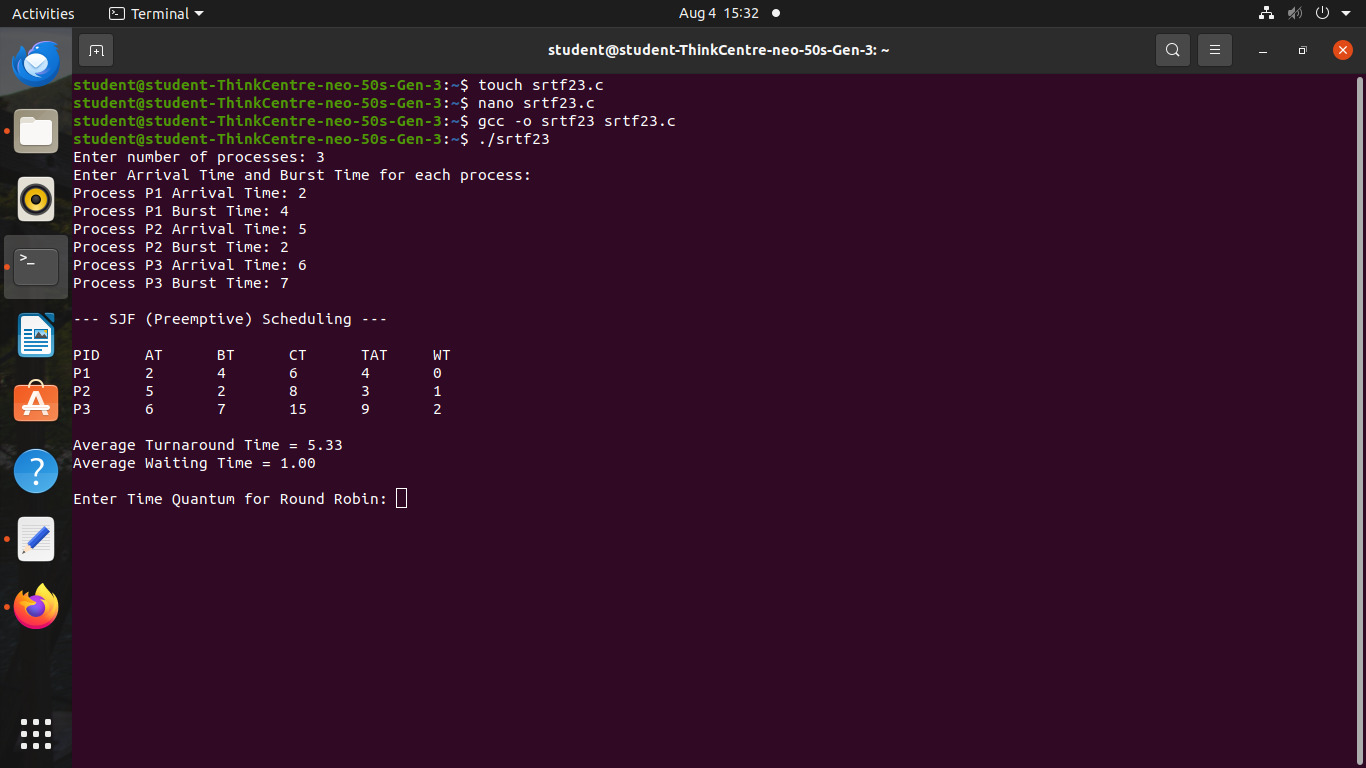
RoundRobin(p2, n, quantum);

return 0;

}

**Output:-**

****

****

**Practical no 4**

1. **Thread synchronization using counting semaphores. Application to demonstrate: producer consumer problem with counting semaphores and mutex.**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#deϐine MAX 5 // Maximum buffer size

int buffer[MAX]; // Shared buffer

int in = 0, out = 0; // Indices for in and out

sem\_t empty, full; // Counting semaphores

pthread\_mutex\_t mutex; // Mutex for critical section

void\* producer(void\* arg) {

int items\_to\_produce = \*(int\*)arg; // Get the number of items to produce

for (int i = 0; i < items\_to\_produce; i++) {

sem\_wait(&empty); // Wait for an empty slot

pthread\_mutex\_lock(&mutex); // Lock the buffer

buffer[in] = i; // Produce an item

printf("Number of items Produced: %d\n", i);

in = (in + 1) % MAX; // Update the in index

pthread\_mutex\_unlock(&mutex); // Unlock the buffer

sem\_post(&full); // Signal that an item is produced

}

return NULL;

}

void\* consumer(void\* arg) {

int items\_to\_consume = \*(int\*)arg; // Get the number of items to consume

for (int i = 0; i < items\_to\_consume; i++) {

sem\_wait(&full); // Wait for a ϐilled slot

pthread\_mutex\_lock(&mutex); // Lock the buffer

int item = buffer[out]; // Consume an item

printf("Number of items Consumed: %d\n", item);

out = (out + 1) % MAX; // Update the out index

pthread\_mutex\_unlock(&mutex); // Unlock the buffer

sem\_post(&empty); // Signal that an item is consumed

}

return NULL;

}

int main() {

pthread\_t prod, cons;

int items\_to\_process;

sem\_init(&empty, 0, MAX); // Set empty slots to MAX

sem\_init(&full, 0, 0); // No ϐilled slots initially

pthread\_mutex\_init(&mutex, NULL); // Initialize the mutex

printf("Enter the number of items to produce and consume: ");

scanf("%d", &items\_to\_process);

pthread\_create(&prod, NULL, producer, &items\_to\_process);

pthread\_create(&cons, NULL, consumer, &items\_to\_process);

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

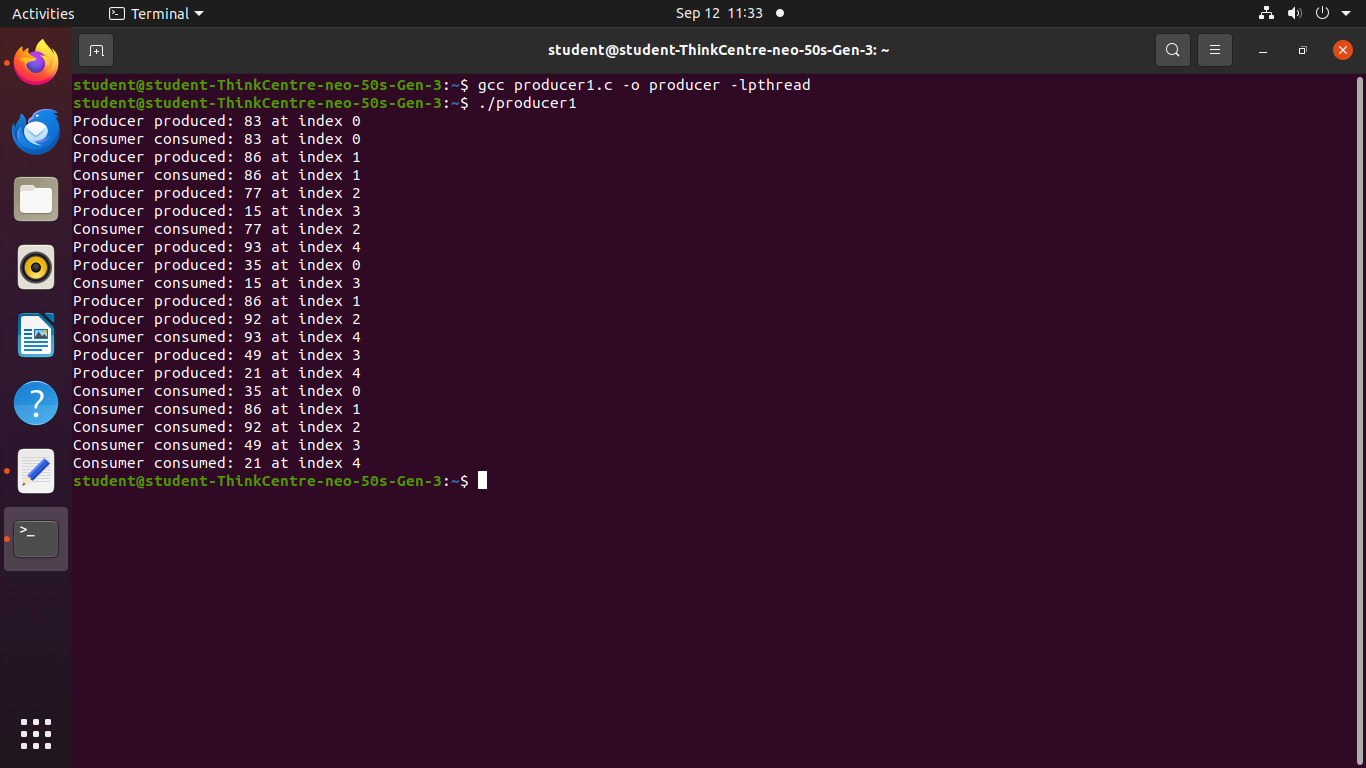
sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

**Output:**

1. **Thread synchronization and mutual exclusion using mutex. Application to demonstrate: ReaderWriter problem with reader priority.**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

pthread\_mutex\_t readWriteMutex;

pthread\_mutex\_t mutex;

int readCount = 0;

int sharedData = 0;

int iterations; // Number of iterations for readers and writers

void\* reader(void\* arg) {

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

for (int i = 0; i < iterations; i++) { // Run for a fixed number of iterations

pthread\_mutex\_lock(&mutex);

readCount++;

if (readCount == 1) {

pthread\_mutex\_lock(&readWriteMutex);

}

pthread\_mutex\_unlock(&mutex);

printf("Reader %d: Reading data %d\n", id, sharedData);

sleep(1); // Simulate reading time

pthread\_mutex\_lock(&mutex);

readCount--;

if (readCount == 0) {

pthread\_mutex\_unlock(&readWriteMutex);

}

pthread\_mutex\_unlock(&mutex);

sleep(1); // Simulate some delay before next read

}

return NULL;

}

void\* writer(void\* arg) {

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

for (int i = 0; i < iterations; i++) { // Run for a fixed number of iterations

pthread\_mutex\_lock(&readWriteMutex);

sharedData++;

printf("Writer %d: Writing data %d\n", id, sharedData);

sleep(1); // Simulate writing time pthread\_mutex\_unlock(&readWriteMutex);

sleep(3); // Simulate some delay before next write

}return NULL;

}int main() {

int numReaders, numWriters;

printf("Enter the number of readers: ");

scanf("%d", &numReaders);

printf("Enter the number of writers: ");

scanf("%d", &numWriters);

printf("Enter the number of iterations for each reader/writer: ");

scanf("%d", &iterations);

pthread\_t\* readers = malloc(numReaders \* sizeof(pthread\_t));

pthread\_t\* writers = malloc(numWriters \* sizeof(pthread\_t));

int\* readerIDs = malloc(numReaders \* sizeof(int));

int\* writerIDs = malloc(numWriters \* sizeof(int));

pthread\_mutex\_init(&readWriteMutex, NULL);

pthread\_mutex\_init(&mutex, NULL);

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

readerIDs[i] = i + 1;

pthread\_create(&readers[i], NULL, reader, &readerIDs[i]);

}

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

writerIDs[i] = i + 1;

pthread\_create(&writers[i], NULL, writer, &writerIDs[i]);

}

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

pthread\_join(readers[i], NULL);

} for (int i = 0; i < numWriters; i++) {

pthread\_join(writers[i], NULL);

}

free(readers);

free(writers);

free(readerIDs);

free(writerIDs);

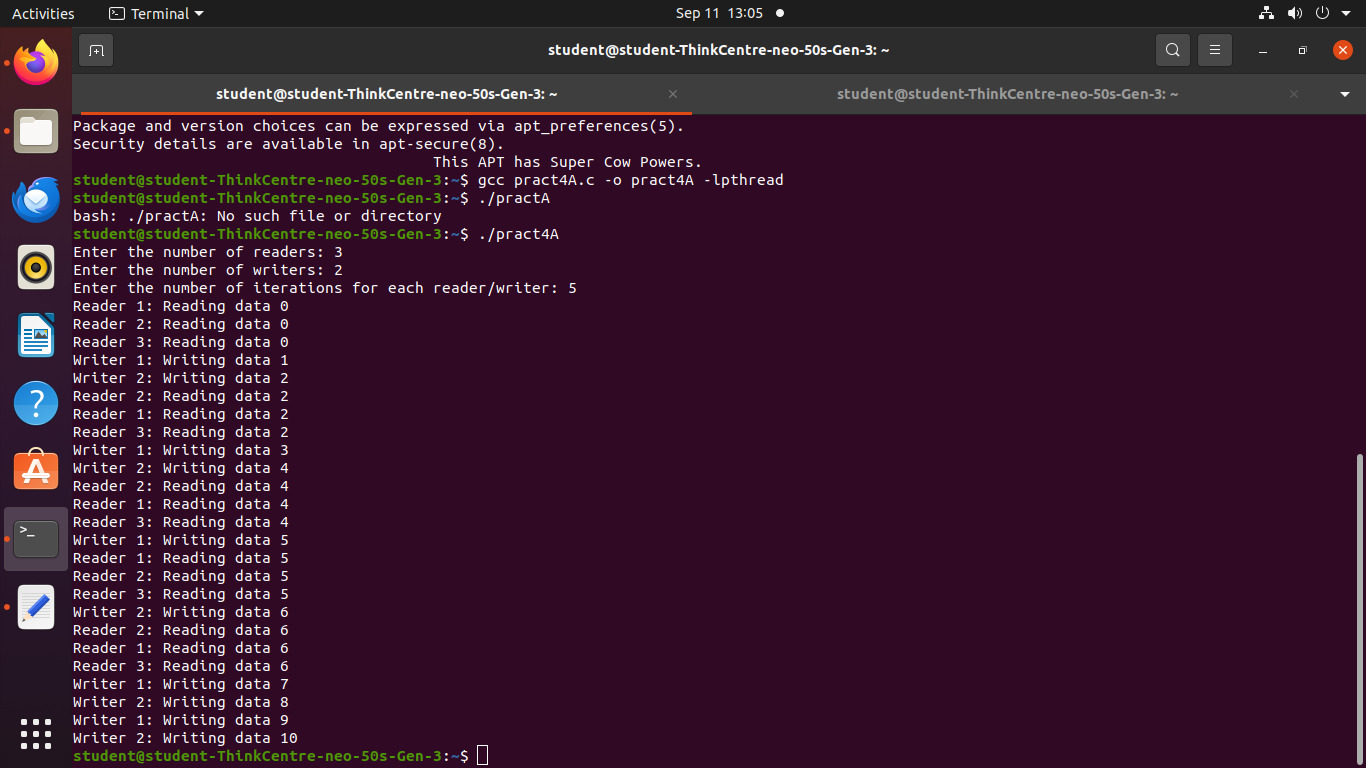
pthread\_mutex\_destroy(&readWriteMutex);

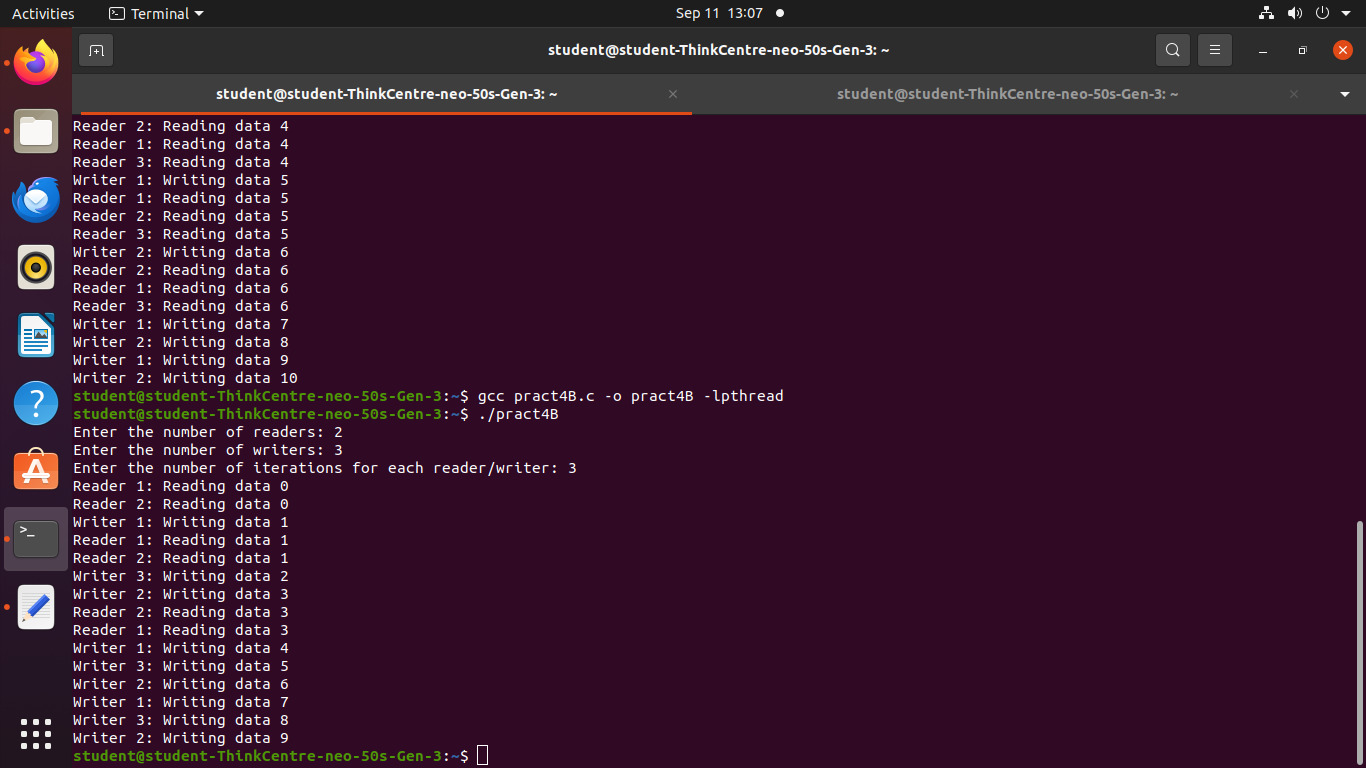
pthread\_mutex\_destroy(&mutex);

return 0;

}

**Output:-**

****

****

**Practical no:5**

**Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm**

#include <stdio.h>

#include <stdbool.h> // For using bool type

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

int main() {

int num\_processes, num\_resources;

int available[MAX\_RESOURCES];

int max[MAX\_PROCESSES][MAX\_RESOURCES];

int allocation[MAX\_PROCESSES][MAX\_RESOURCES];

int need[MAX\_PROCESSES][MAX\_RESOURCES];

bool finish[MAX\_PROCESSES];

int safe\_sequence[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

printf("Enter the number of processes: ");

scanf("%d", &num\_processes);

printf("Enter the number of resources: ");

scanf("%d", &num\_resources);

printf("Enter the available resources (e.g., R1 R2 R3): ");

for (int i = 0; i < num\_resources; i++) {

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

}

printf("Enter the allocation matrix (P x R): \n");

for (int i = 0; i < num\_processes; i++) {

printf("For Process P%d: ", i);

for (int j = 0; j < num\_resources; j++) {

scanf("%d", &allocation[i][j]); } }

printf("Enter the maximum matrix (P x R): \n");

for (int i = 0; i < num\_processes; i++) {

printf("For Process P%d: ", i);

for (int j = 0; j < num\_resources; j++) {

scanf("%d", &max[i][j]); } }

for (int i = 0; i < num\_processes; i++) {

for (int j = 0; j < num\_resources; j++) {

need[i][j] = max[i][j] - allocation[i][j]; } }

for (int i = 0; i < num\_resources; i++) {

work[i] = available[i]; }

for (int i = 0; i < num\_processes; i++) {

finish[i] = false; }

int count = 0;

int safe\_seq\_index = 0;

while (count < num\_processes) {

bool found\_process = false;

for (int p = 0; p < num\_processes; p++) {

if (finish[p] == false) {

bool can\_execute = true;

for (int r = 0; r < num\_resources; r++) {

if (need[p][r] > work[r]) {

can\_execute = false;

break } }

if (can\_execute) {

for (int r = 0; r < num\_resources; r++) {

work[r] += allocation[p][r];

}

finish[p] = true;

safe\_sequence[safe\_seq\_index++] = p;

count++;

found\_process = true }} }

if (!found\_process && count < num\_processes) {

printf("\nSystem is in an UNSAFE state. No safe sequence exists.\n");

return 0; } }

printf("\nSystem is in a SAFE state. Safe sequence: ");

for (int i = 0; i < num\_processes; i++) {

printf("P%d", safe\_sequence[i]);

if (i < num\_processes - 1) {

printf(" -> ");

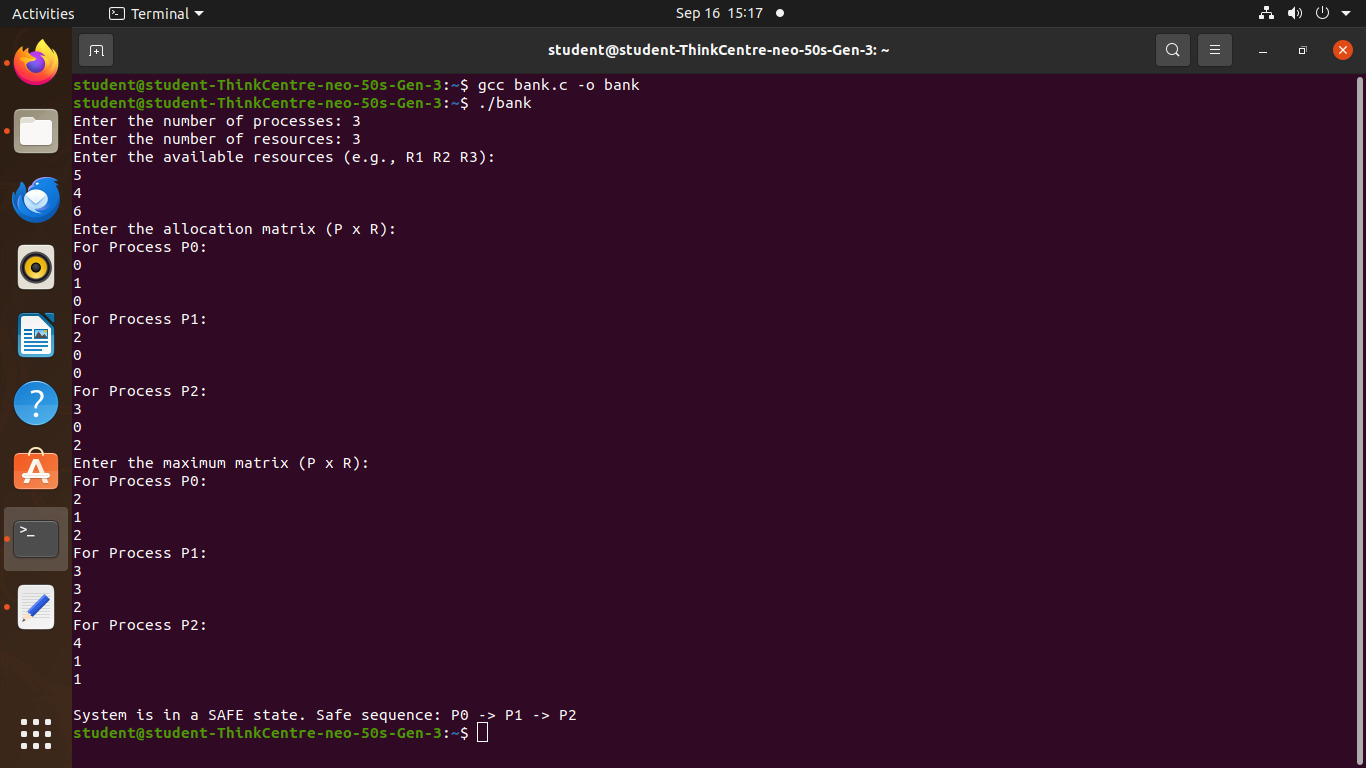
}

}

printf("\n");

return 0;

}

**Output:-**

**Practical no 6**

**Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

void printFrames(int frames[], int frame\_size) {

for (int i = 0; i < frame\_size; i++) {

if (frames[i] == -1) {

printf(" - ");

} else {

printf(" %d ", frames[i]);

}

}

printf("\n");

}

void fcfs(int pages[], int n\_pages, int frame\_size) {

int frames[frame\_size];

for (int i = 0; i < frame\_size; i++) {

frames[i] = -1; // Initialize frames as empty

}

int page\_faults = 0;

int frame\_index = 0; // Pointer to the oldest page

printf("\n--- FCFS Page Replacement ---\n");

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

int current\_page = pages[i];

bool found = false;

for (int j = 0; j < frame\_size; j++) {

if (frames[j] == current\_page) {

found = true;

break;

}

}

if (!found) {

page\_faults++;

frames[frame\_index] = current\_page;

frame\_index = (frame\_index + 1) % frame\_size;

}

printf("Page %d: ", current\_page);

printFrames(frames, frame\_size);

}

printf("Total FCFS Page Faults: %d\n", page\_faults);

}

void lru(int pages[], int n\_pages, int frame\_size) {

int frames[frame\_size];

int recency[frame\_size]; // Stores the last access time (index in pages array)

for (int i = 0; i < frame\_size; i++) {

frames[i] = -1;

recency[i] = -1;

}

int page\_faults = 0;

printf("\n--- LRU Page Replacement ---\n");

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

int current\_page = pages[i];

bool found = false;

int found\_index = -1;

for (int j = 0; j < frame\_size; j++) {

if (frames[j] == current\_page) {

found = true;

found\_index = j;

break;

}

}

if (!found) {

page\_faults++;

int lru\_index = -1;

int min\_recency = INT\_MAX;

for (int j = 0; j < frame\_size; j++) {

if (frames[j] == -1) { // Empty frame available

lru\_index = j;

break;

}

if (recency[j] < min\_recency) {

min\_recency = recency[j];

lru\_index = j;

}

}

frames[lru\_index] = current\_page;

recency[lru\_index] = i; // Update recency

} else {

recency[found\_index] = i; }

printf("Page %d: ", current\_page);

printFrames(frames, frame\_size);

}

printf("Total LRU Page Faults: %d\n", page\_faults);

}

void optimal(int pages[], int n\_pages, int frame\_size) {

int frames[frame\_size];

for (int i = 0; i < frame\_size; i++) {

frames[i] = -1;

}

int page\_faults = 0;

printf("\n--- Optimal Page Replacement ---\n");

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

int current\_page = pages[i];

bool found = false;

for (int j = 0; j < frame\_size; j++) {

if (frames[j] == current\_page) {

found = true;

break; } }

if (!found) {

page\_faults++;

int replace\_index = -1;

bool empty\_frame\_found = false;

for (int j = 0; j < frame\_size; j++) {

if (frames[j] == -1) {

replace\_index = j;

empty\_frame\_found = true;

break;

}

}

if (!empty\_frame\_found) {

int farthest\_future\_index = -1;

int max\_future\_use = -1;

for (int j = 0; j < frame\_size; j++) {

int next\_use = INT\_MAX; // Initialize with a large value

for (int k = i + 1; k < n\_pages; k++) {

if (frames[j] == pages[k]) {

next\_use = k;

break;

}

}

if (next\_use > max\_future\_use) {

max\_future\_use = next\_use;

farthest\_future\_index = j;

}

}

replace\_index = farthest\_future\_index;

}

frames[replace\_index] = current\_page;

}

printf("Page %d: ", current\_page);

printFrames(frames, frame\_size);

}

printf("Total Optimal Page Faults: %d\n", page\_faults);

}

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int n\_pages = sizeof(pages) / sizeof(pages[0]);

int frame\_size = 3;

fcfs(pages, n\_pages, frame\_size);

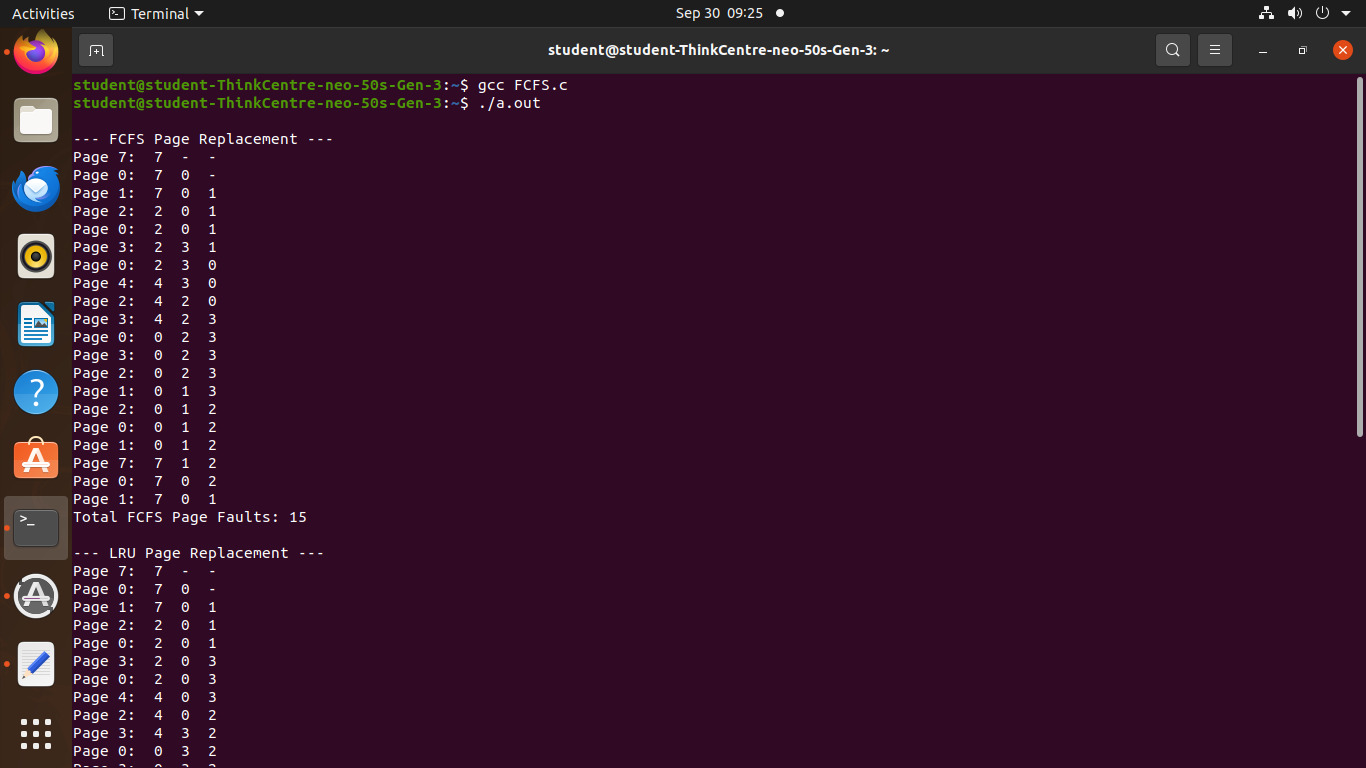
lru(pages, n\_pages, frame\_size);

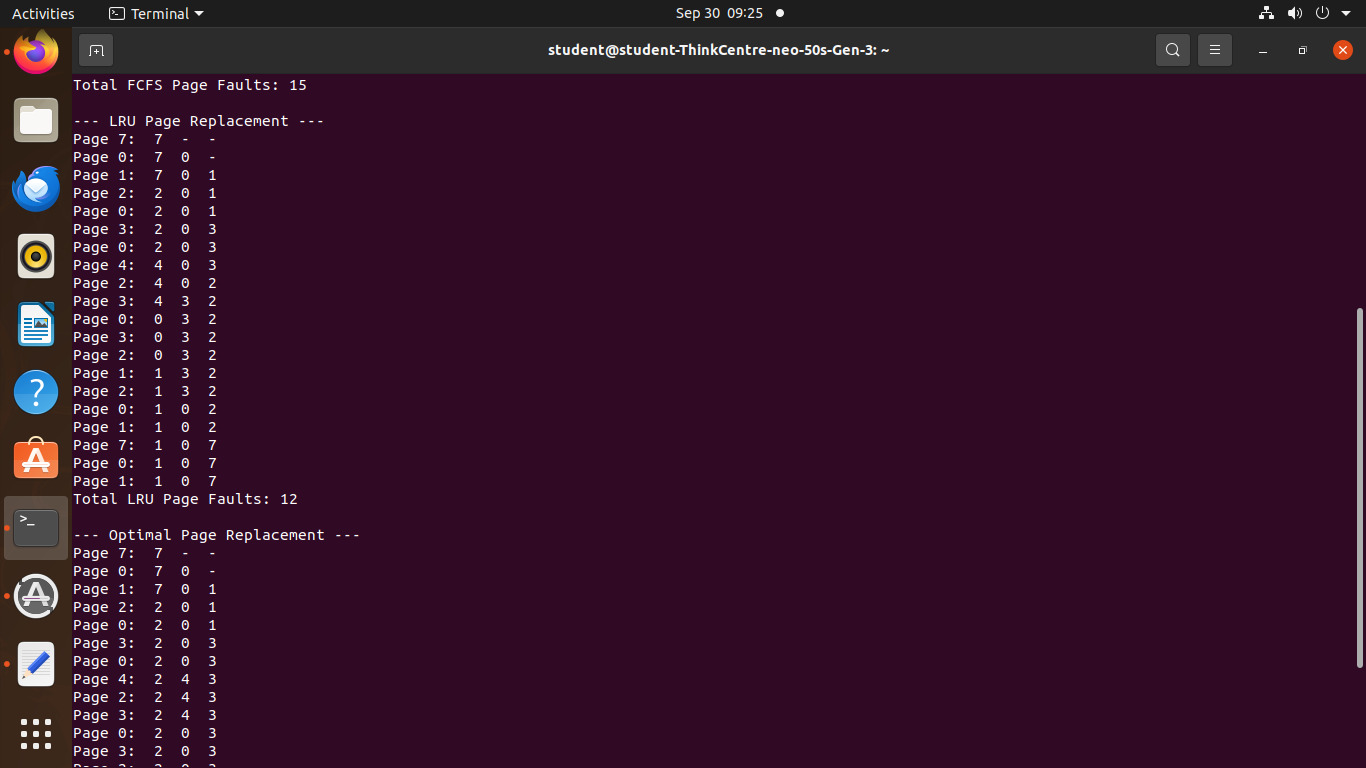
optimal(pages, n\_pages, frame\_size);

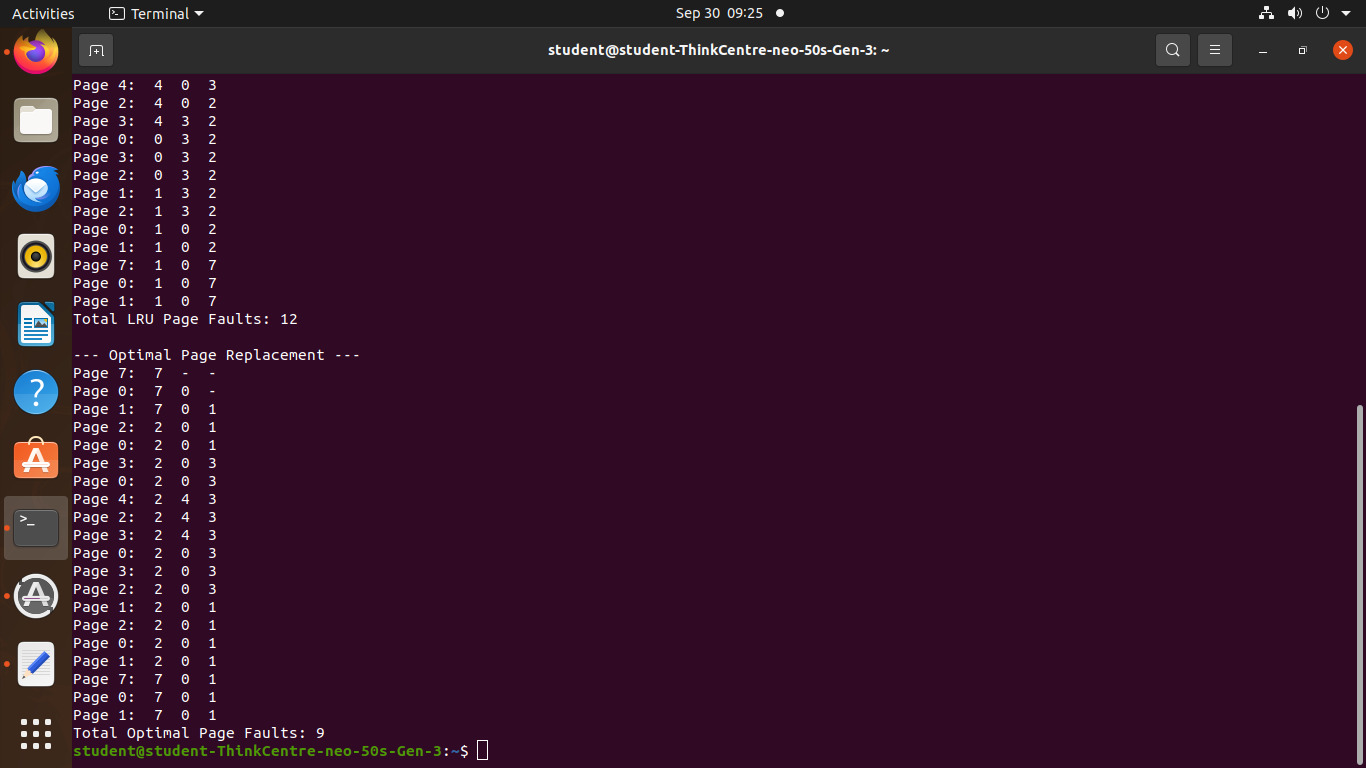
return 0;

}

**Output:-**

****

****

****

**Practical no 7**

1. **FIFOS: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays onstandard output.**

**Reader:**

#include <stdio.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

void countWordsLinesChars(char \*sentence, int \*wordCount, int \*lineCount, int \*charCount) {

char \*token = strtok(sentence, " \n");

\*wordCount = 0;

\*lineCount = 1; // Start with one line

\*charCount = strlen(sentence);

while (token != NULL) {

(\*wordCount)++;

token = strtok(NULL, " \n");

}

}

int main() {

int fd;

char buffer[100];

ssize\_t bytesRead;

fd = open(FIFO\_PATH, O\_RDONLY);

if (fd == -1) {

perror("Error opening FIFO for reading");

return 1;

}

while (1) {

bytesRead = read(fd, buffer, sizeof(buffer) - 1);

if (bytesRead == -1) {

perror("Error reading from FIFO");

break;

}

if (bytesRead == 0) {

break;

}

buffer[bytesRead] = '\0';

if (strcmp(buffer, "exit\n") == 0) {

printf("Exiting reader...\n");

break;

}

printf("Received: %s\n", buffer);

char temp[100];

strncpy(temp, buffer, sizeof(temp));

temp[sizeof(temp) - 1] = '\0';

int wordCount, lineCount, charCount;

countWordsLinesChars(temp, &wordCount, &lineCount, &charCount);

printf("Characters: %d\n", charCount);

printf("Words: %d\n", wordCount);

printf("Lines: %d\n", lineCount);

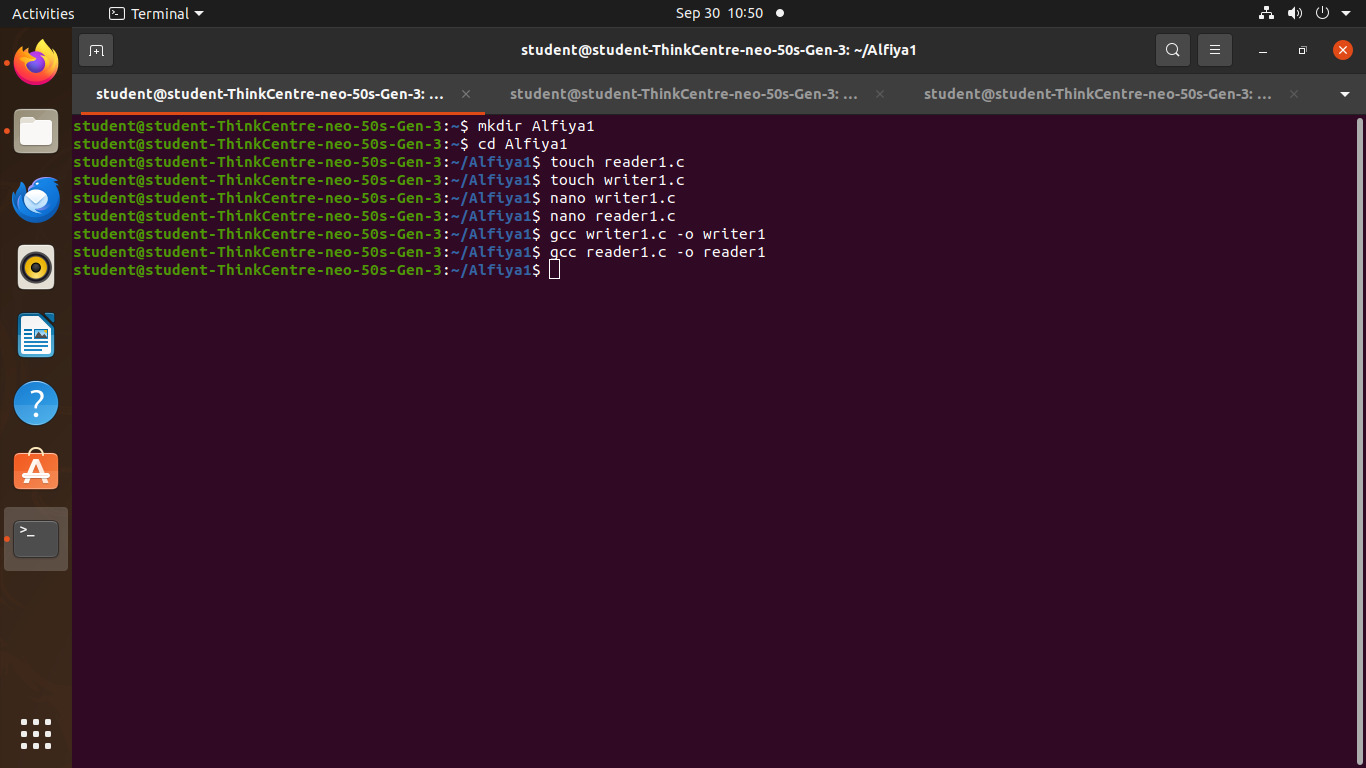
}

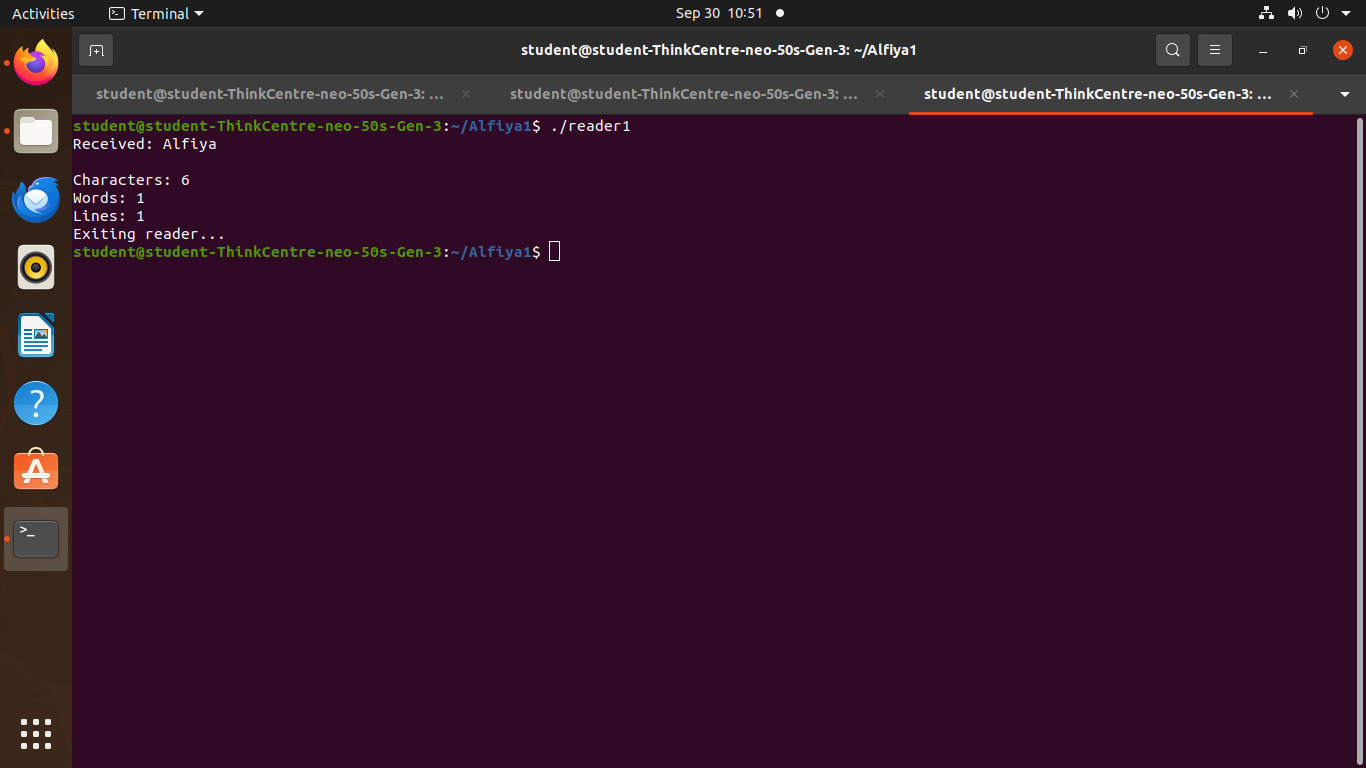
close(fd);

return 0;

}

**Output:-**

****

****

**Writer:**

#include <stdio.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#include <errno.h>

#define FIFO\_PATH "/home/student/prajwal/text"

int main() {

int fd;

char buffer[100]; // Buffer for user input

// Create FIFO if it doesn't already exist

if (mkfifo(FIFO\_PATH, 0666) == -1 && errno != EEXIST) {

perror("Error creating FIFO");

return 1;

}

// Open FIFO for writing

fd = open(FIFO\_PATH, O\_WRONLY);

if (fd == -1) {

perror("Error opening FIFO for writing");

return 1;

}

while (1) {

printf("Enter a sentence (or 'exit' to quit): ");

if (fgets(buffer, sizeof(buffer), stdin) == NULL) {

perror("Error reading input");

break;

}

if (strcmp(buffer, "exit\n") == 0) {

write(fd, buffer, strlen(buffer) + 1);

break;

}

if (write(fd, buffer, strlen(buffer) + 1) == -1) {

perror("Error writing to FIFO");

break;

}

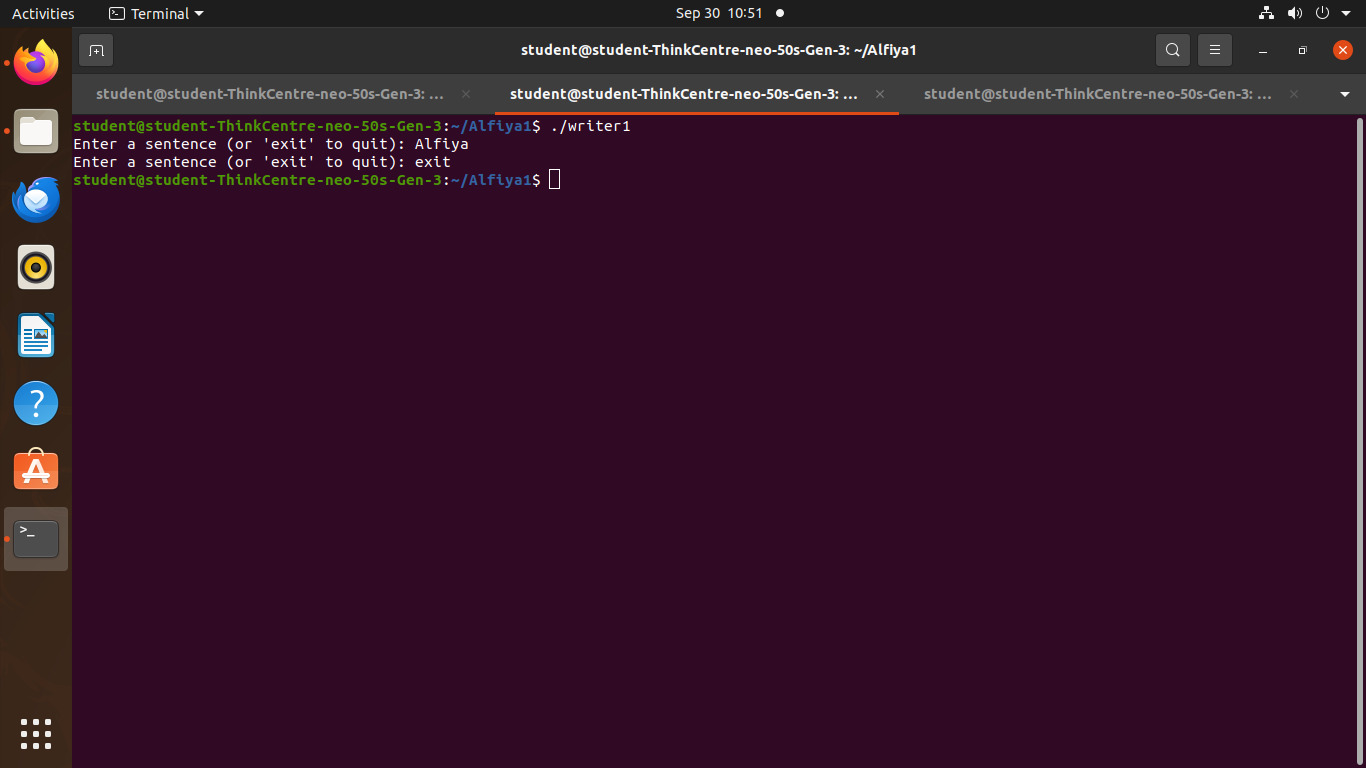
}

close(fd);

return 0;

}

**Ouput:-**

****

**B. Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.**

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#define SHM\_SIZE 1024 // Define the size of shared memory

int main() {

int shmid;

key\_t key = 1234; // Shared memory key

char \*shm;

// Access the shared memory segment

shmid = shmget(key, SHM\_SIZE, 0666);

if (shmid < 0) {

perror("shmget");

exit(1);

}

shm = shmat(shmid, NULL, 0);

if (shm == (char \*) -1) {

perror("shmat");

exit(1);

}

// Read the message from shared memory

printf("Client: Read message from shared memory: \"%s\"\n", shm);

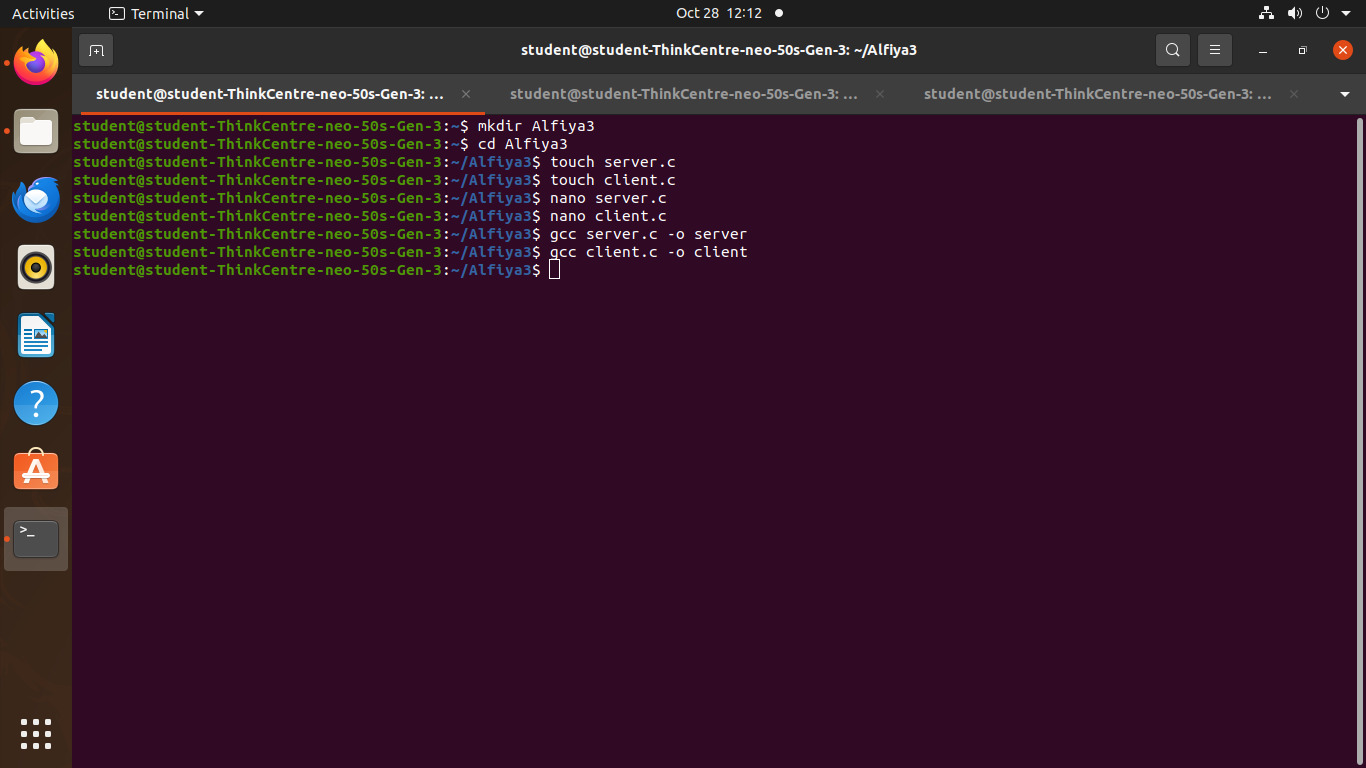
// Detach from shared memory

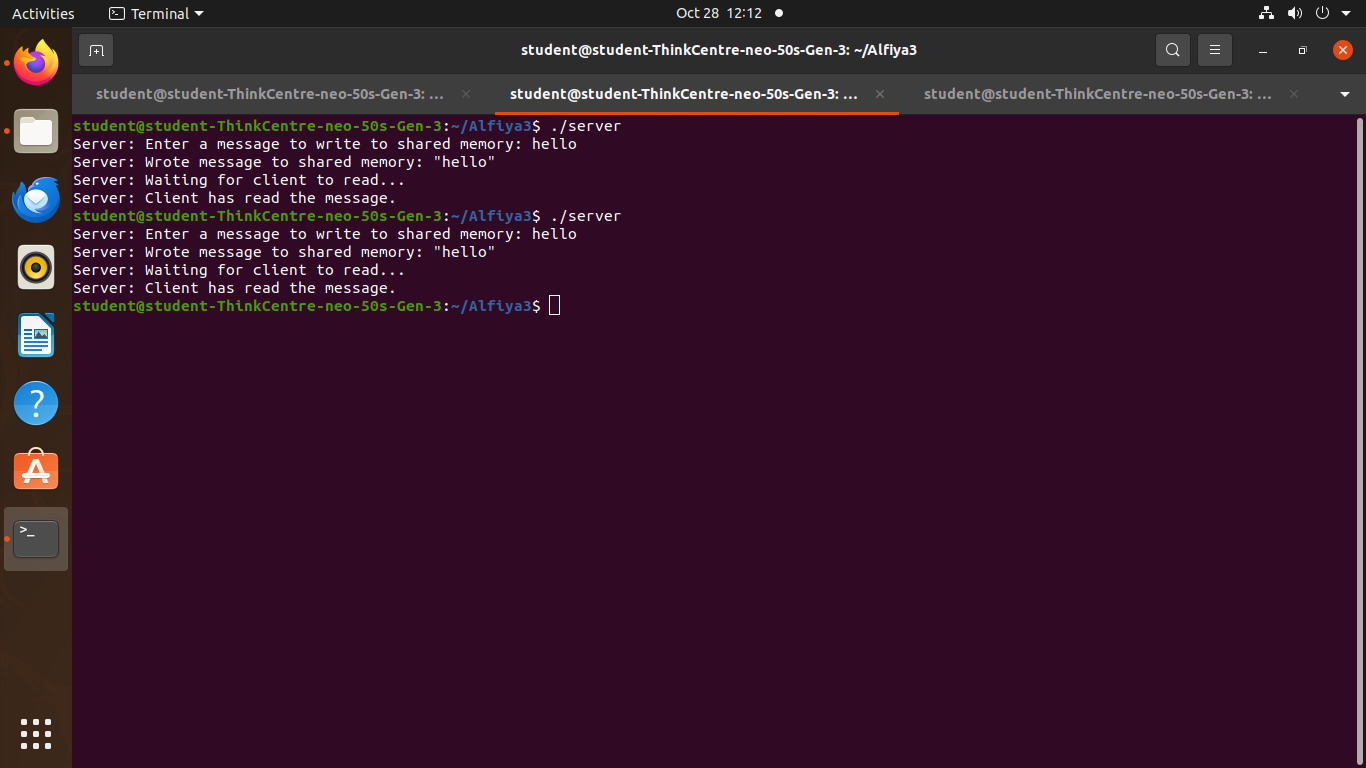
shmdt(shm);

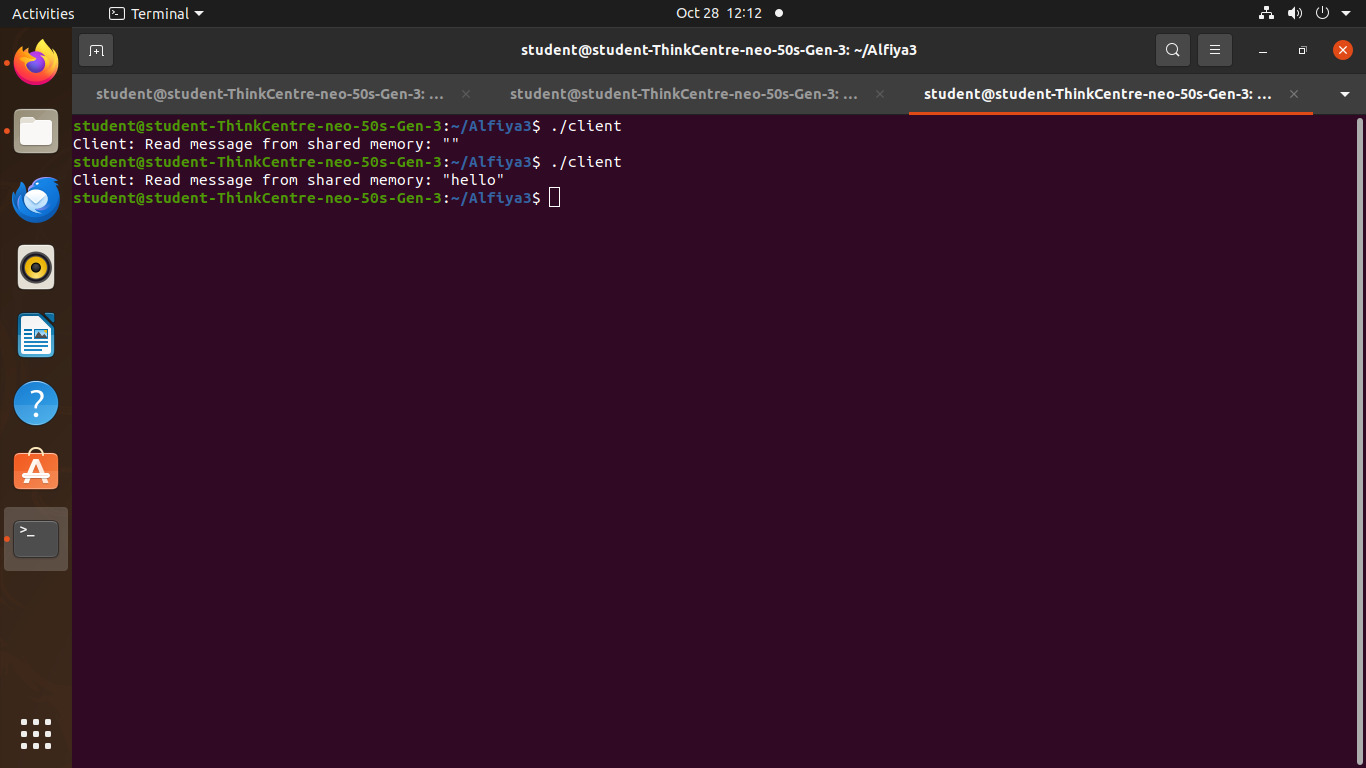
return 0;

}

**Output:-**

****

****

****

**Practical no 8 Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.**

**SSTF:-**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 100

void SSTF(int requests[], int num\_requests, int initial\_head) {

int completed[MAX\_REQUESTS] = {0};

int current\_head = initial\_head;

int total\_distance = 0;

printf("\n\n Seek Sequence: ");

for (int i = 0; i < num\_requests; i++) {

int min\_index = -1;

int min\_distance = 1000000; // Large number

// Find the closest request

for (int j = 0; j < num\_requests; j++) {

if (!completed[j]) {

int distance = abs(requests[j] - current\_head);

if (distance < min\_distance) {

min\_distance = distance;

min\_index = j;

}

}

}

// Service the closest request

if (min\_index != -1) {

total\_distance += min\_distance;

current\_head = requests[min\_index];

completed[min\_index] = 1;

printf("%d ", current\_head);

}

}

printf("\n Total Number of Seek Operations = %d\n", total\_distance);

}

int main() {

int requests[MAX\_REQUESTS];

int num\_requests, initial\_head;

printf("\nEnter number of disk requests: ");

scanf("%d", &num\_requests);

printf("Enter disk requests: ");

for (int i = 0; i < num\_requests; i++) {

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

}

printf("Enter initial head position: ");

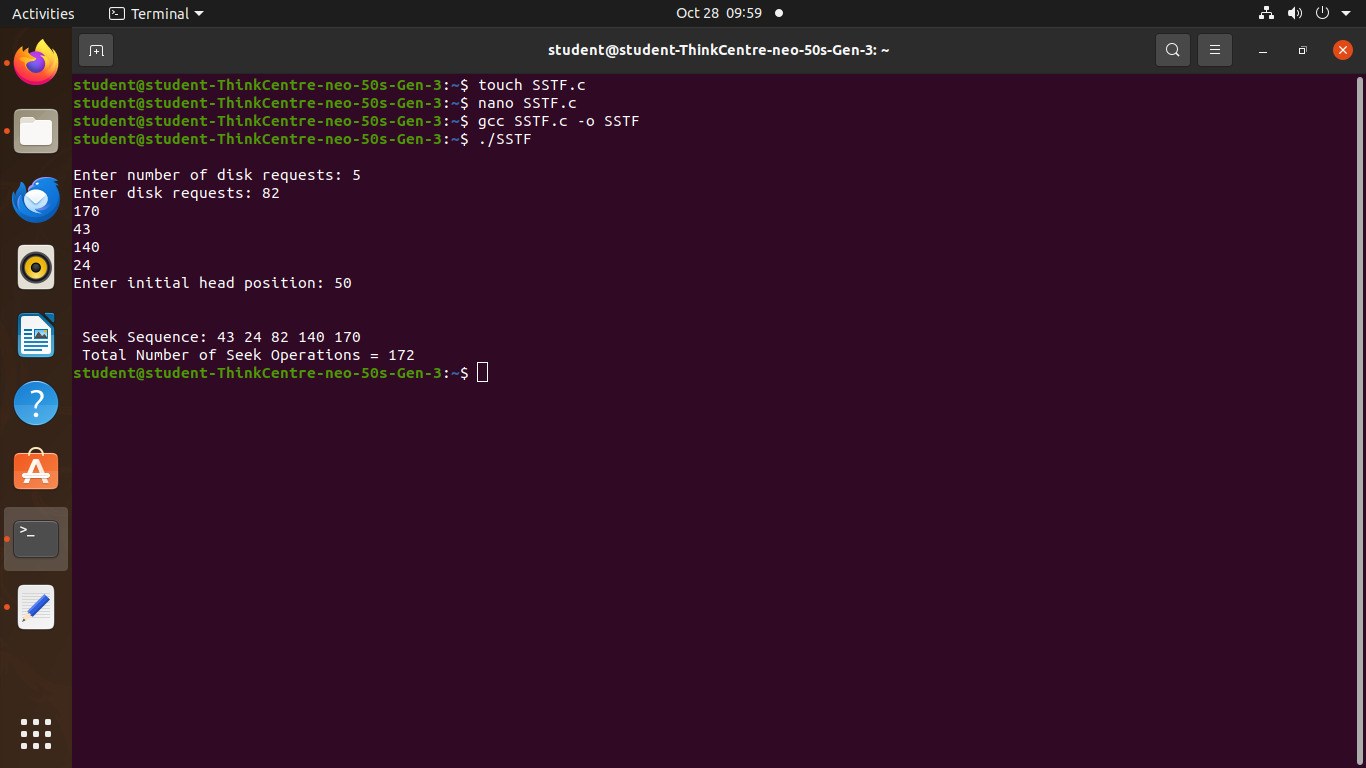
scanf("%d", &initial\_head);

SSTF(requests, num\_requests, initial\_head);

return 0;

}

**Output:-**

****

**SCAN.c**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 100

void SCAN(int requests[], int num\_requests, int initial\_head, int direction) {

int total\_distance = 0;

int current\_head = initial\_head;

// Sort the requests

for (int i = 0; i < num\_requests - 1; i++) {

for (int j = i + 1; j < num\_requests; j++) {

if (requests[i] > requests[j]) {

int temp = requests[i];

requests[i] = requests[j];

requests[j] = temp;

}

}

}

printf("\n\n Seek Sequence: ");

if (direction == 0) { // Moving left

// Service requests moving left

for (int i = num\_requests - 1; i >= 0; i--) {

if (requests[i] <= current\_head) {

total\_distance += abs(requests[i] - current\_head);

current\_head = requests[i];

printf("%d ", current\_head);

}

}

// Move to the start of the disk (0)

total\_distance += abs(current\_head - 0);

current\_head = 0;

printf("0 "); // Move to the start

// Then service all requests > 0

for (int i = 0; i < num\_requests; i++) {

if (requests[i] > 0) {

total\_distance += abs(requests[i] - current\_head);

current\_head = requests[i];

printf("%d ", current\_head);

}

}

} else { // Moving right

// Service requests moving right

for (int i = 0; i < num\_requests; i++) {

if (requests[i] >= current\_head) {

total\_distance += abs(requests[i] - current\_head);

current\_head = requests[i];

printf("%d ", current\_head);

}

}

// Move to the end of the disk (disk size = 200)

total\_distance += abs(current\_head - 200);

current\_head = 200;

printf("200 "); // Move to the end

// Then service all requests < disk size

for (int i = num\_requests - 1; i >= 0; i--) {

if (requests[i] < current\_head) {

total\_distance += abs(requests[i] - current\_head);

current\_head = requests[i];

printf("%d ", current\_head);

}

}

}

printf("\n Total number of seek operations = %d\n", total\_distance);

}

int main() {

int requests[MAX\_REQUESTS];

int num\_requests, initial\_head, direction;

printf("\nEnter number of disk requests: ");

scanf("%d", &num\_requests);

printf("Enter disk requests: ");

for (int i = 0; i < num\_requests; i++) {

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

}

printf("Enter initial head position: ");

scanf("%d", &initial\_head);

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

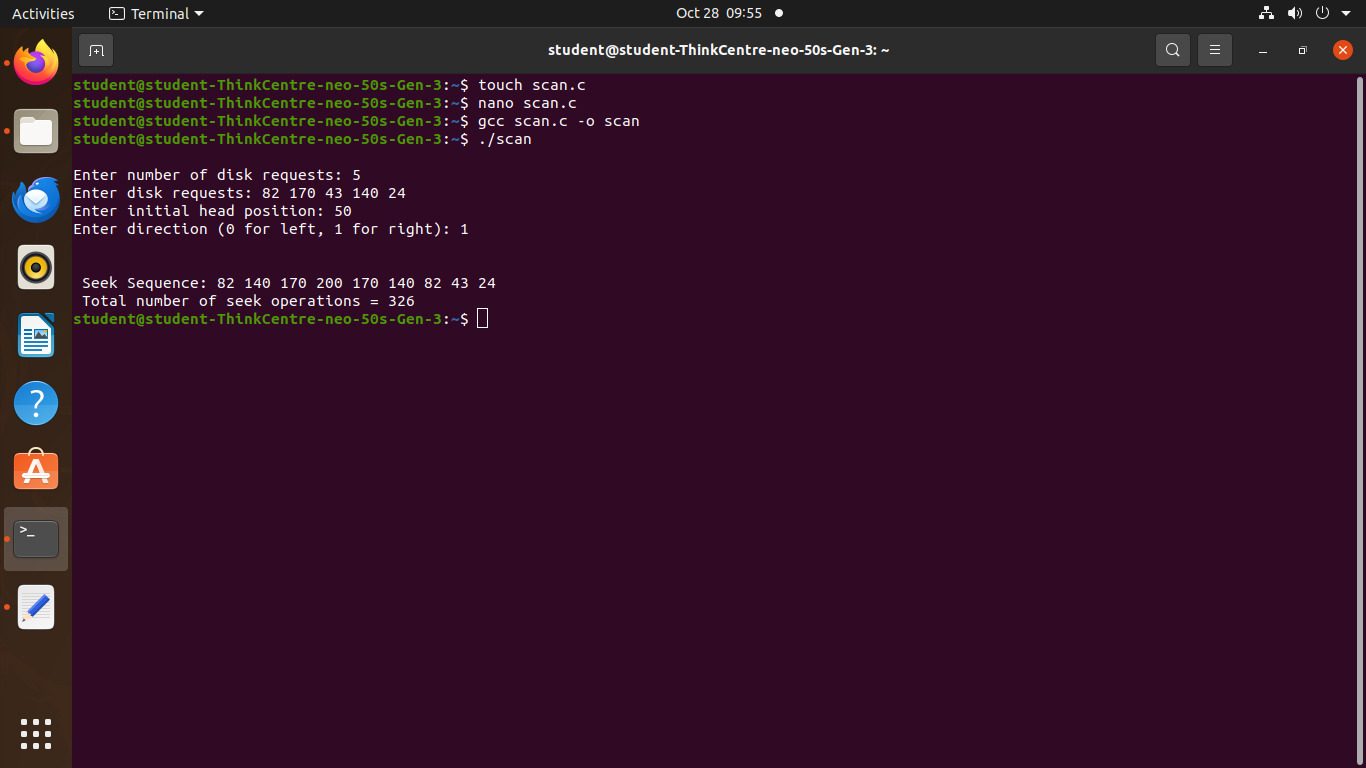
scanf("%d", &direction);

SCAN(requests, num\_requests, initial\_head, direction);

return 0;

}

**Output:-**

****

**CLOCK.c**

#include <stdio.h>

#include <stdlib.h>

void c\_look(int requests[], int num\_requests, int head, int direction) {

int i, j;

// Sort the requests

for (i = 0; i < num\_requests - 1; i++) {

for (j = i + 1; j < num\_requests; j++) {

if (requests[i] > requests[j]) {

int temp = requests[i];

requests[i] = requests[j];

requests[j] = temp;

}

}

}

printf("\n\n Seek Sequence: ");

int total\_seek\_operations = 0;

int current\_position = head;

// Move based on direction

if (direction == 1) { // Moving to the right

// Service requests greater than the current head position

for (i = 0; i < num\_requests; i++) {

if (requests[i] >= current\_position) {

total\_seek\_operations += abs(current\_position - requests[i]);

current\_position = requests[i];

printf("%d ", current\_position);

}

}

// Jump to the smallest request (circular movement)

if (current\_position != requests[0]) {

total\_seek\_operations += abs(current\_position - requests[0]);

current\_position = requests[0];

printf("%d ", current\_position);

}

// Service the remaining smaller requests

for (i = 0; i < num\_requests; i++) {

if (requests[i] < head) {

total\_seek\_operations += abs(current\_position - requests[i]);

current\_position = requests[i];

printf("%d ", current\_position);

}

}

} else { // Moving to the left

// Service requests smaller than the current head position

for (i = num\_requests - 1; i >= 0; i--) {

if (requests[i] <= current\_position) {

total\_seek\_operations += abs(current\_position - requests[i]);

current\_position = requests[i];

printf("%d ", current\_position);

}

}

// Jump to the largest request (circular movement)

if (current\_position != requests[num\_requests - 1]) {

total\_seek\_operations += abs(current\_position -

requests[num\_requests - 1]);

current\_position = requests[num\_requests - 1];

printf("%d ", current\_position);

}

// Service the remaining larger requests

for (i = num\_requests - 1; i >= 0; i--) {

if (requests[i] > head) {

total\_seek\_operations += abs(current\_position - requests[i]);

current\_position = requests[i];

printf("%d ", current\_position);

}

}

}

printf("\n Total number of seek operations = %d\n",

total\_seek\_operations);

}

int main() {

int num\_requests, head, direction;

printf("\nEnter the number of requests: ");

scanf("%d", &num\_requests);

int \*requests = (int \*)malloc(num\_requests \* sizeof(int));

printf("Enter the disk requests: ");

for (int i = 0; i < num\_requests; i++) {

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

}

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

scanf("%d", &head);

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

scanf("%d", &direction);

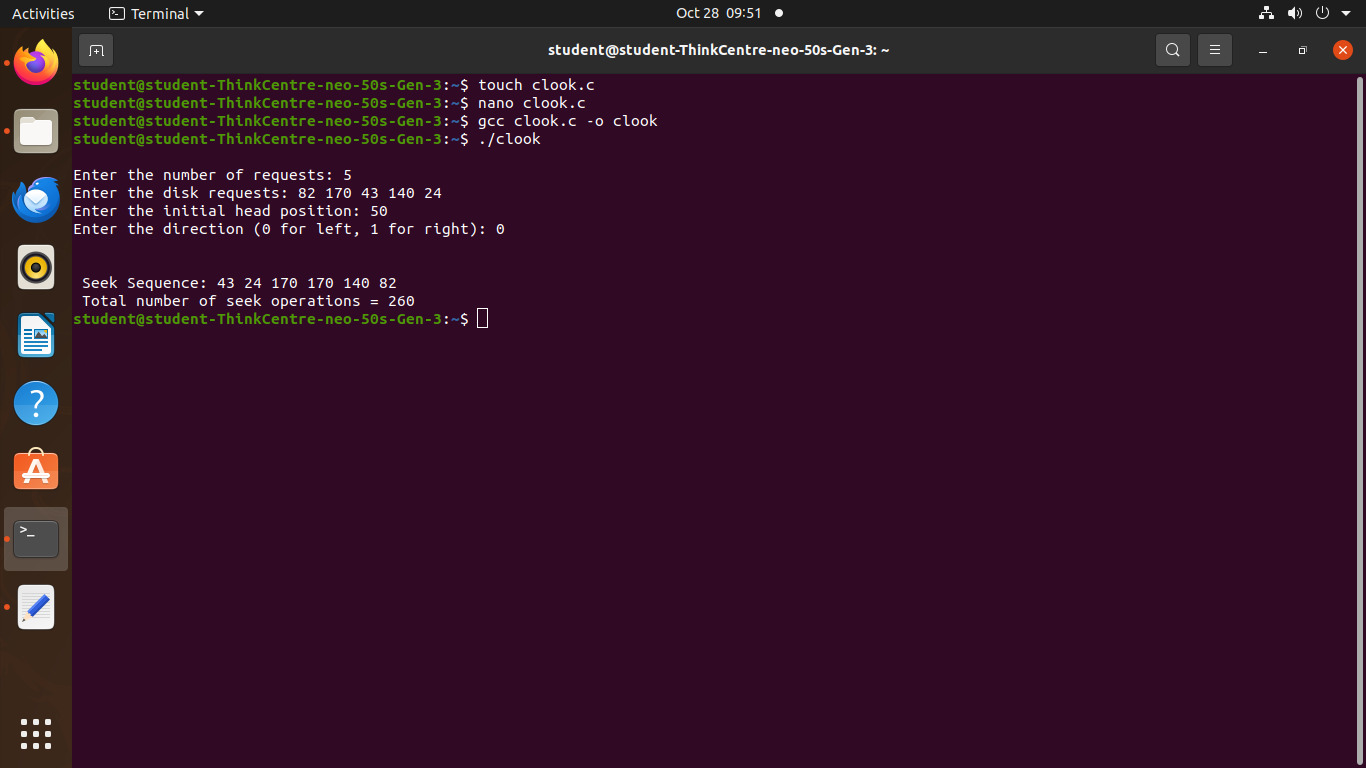
c\_look(requests, num\_requests, head, direction);

free(requests);

return 0;

}

**Output:-**

****