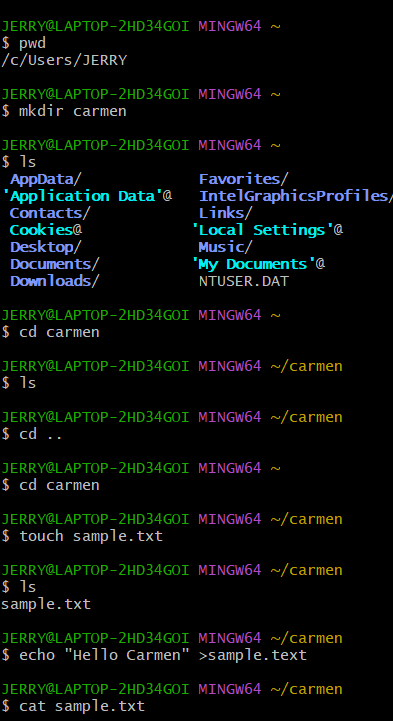
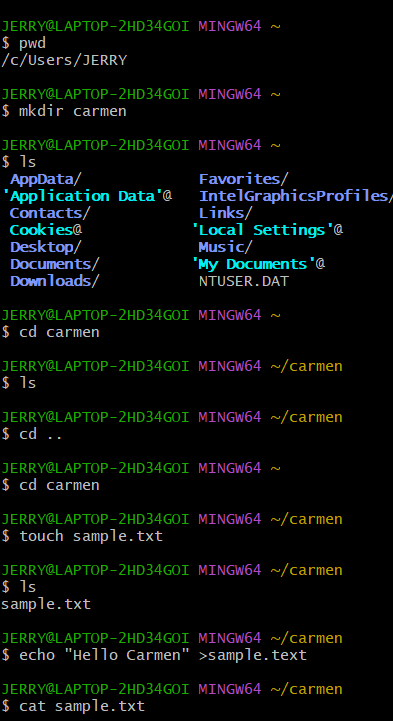
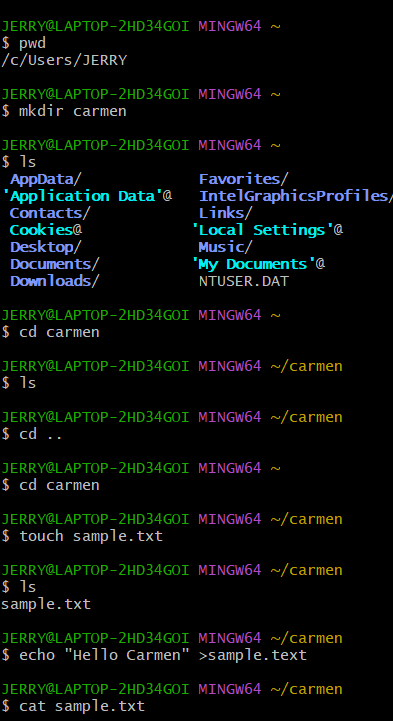
1. **Execution of basic and advanced Unix commands**
2. **pwd - Prints the current working directory**

****

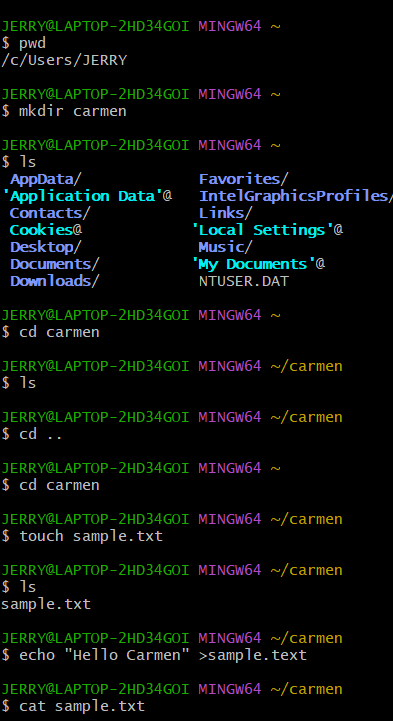
1. **mkdir - Creates a new directory**

****

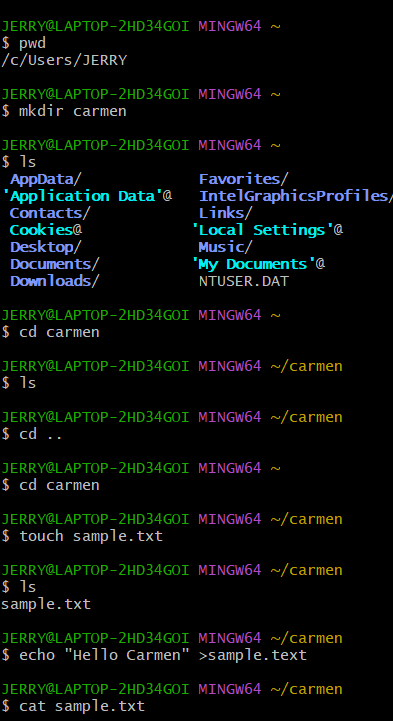
1. **ls - Lists files and directories in the current directory**

****

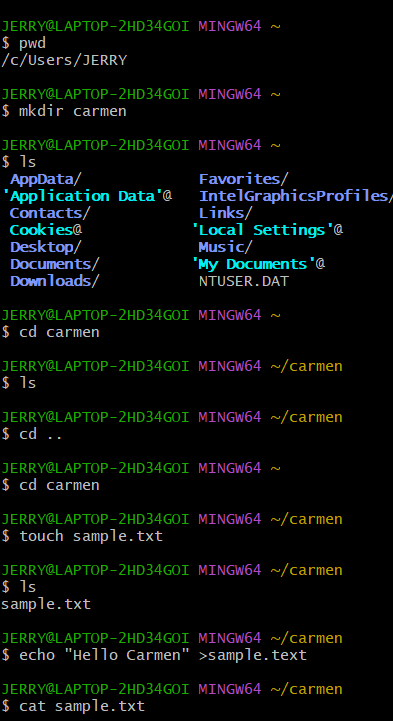
1. **cd - Changes the current working directory**

****

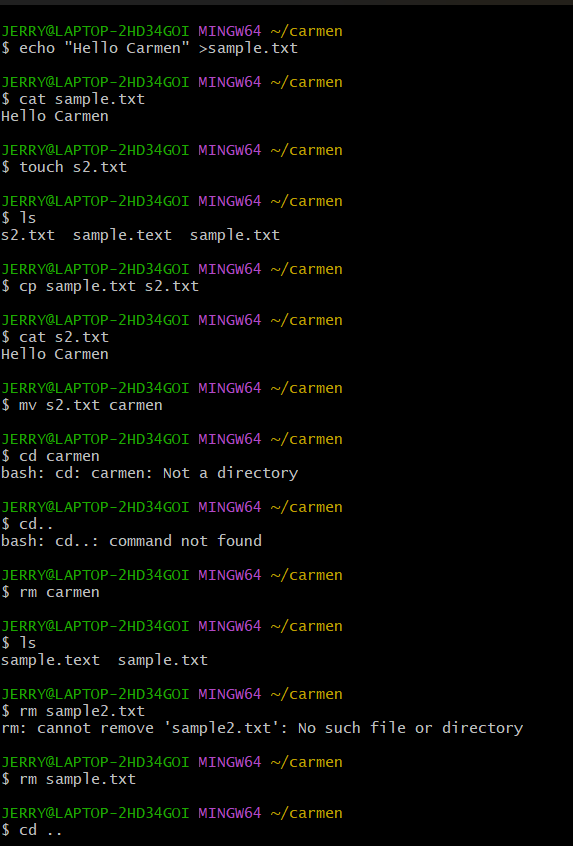
1. **cd .. – changes the current working directory to the parent directory of the current directory.**

****

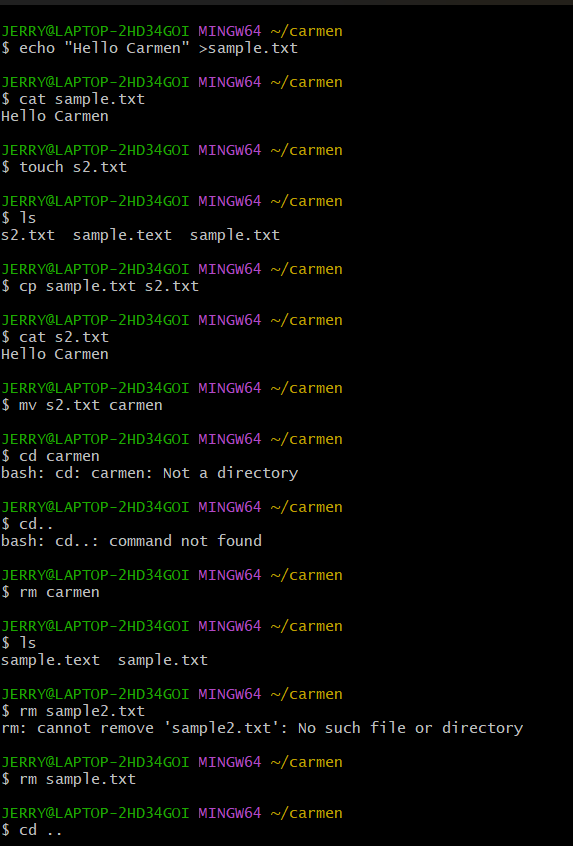
1. **touch - Creates an empty file or updates the access and modification times**

****

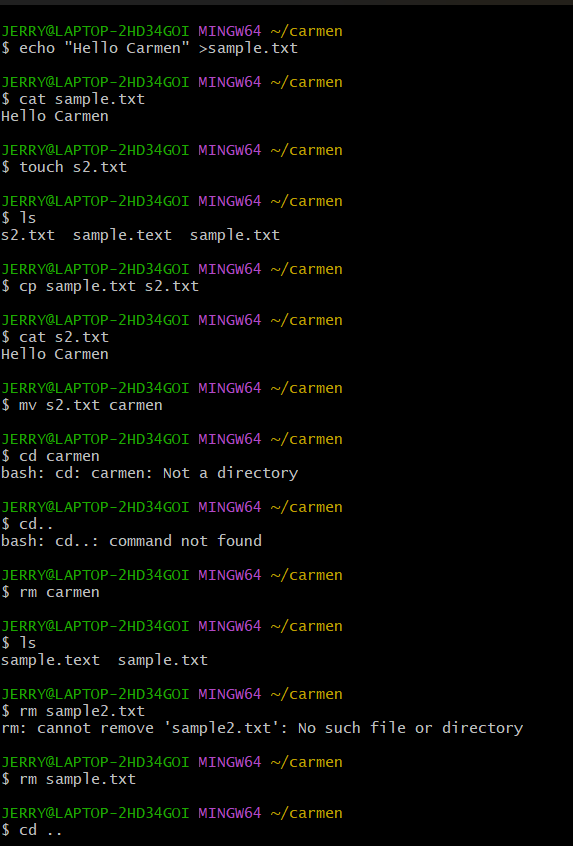
1. **echo - displaying lines of text or string which are passed as arguments on the command line**

****

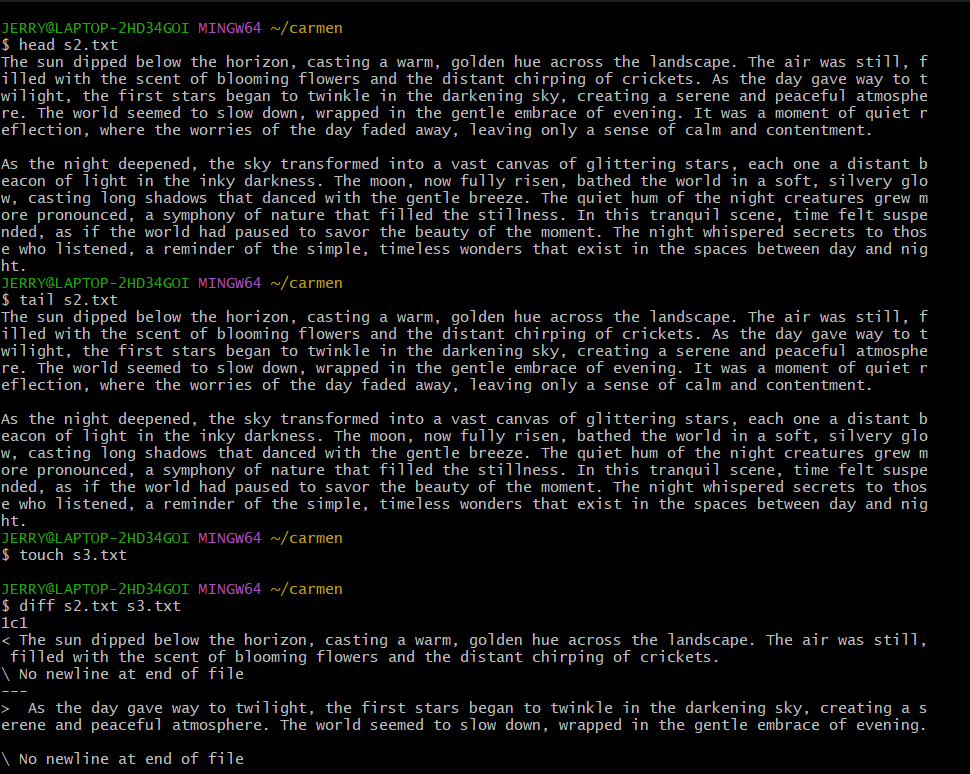
1. **cp - Copies files or directories**

****

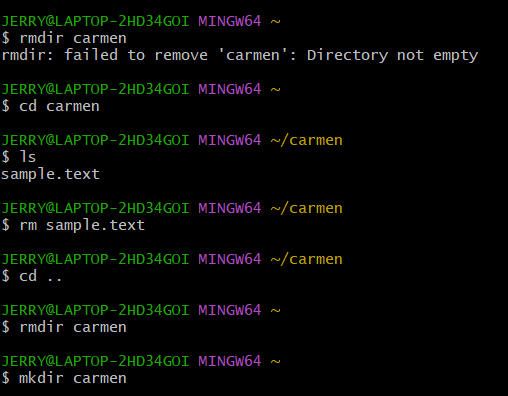
1. **cat - Concatenates files and displays their contents**

****

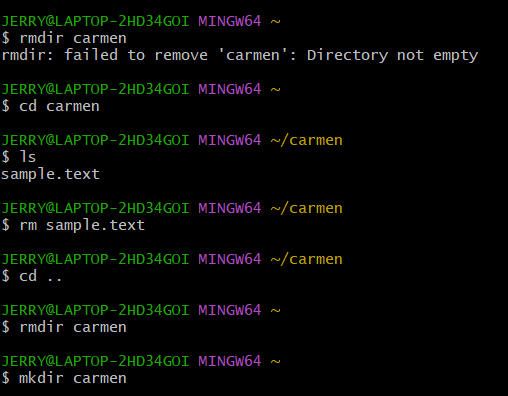
1. **diff - Compares files line by line**

****

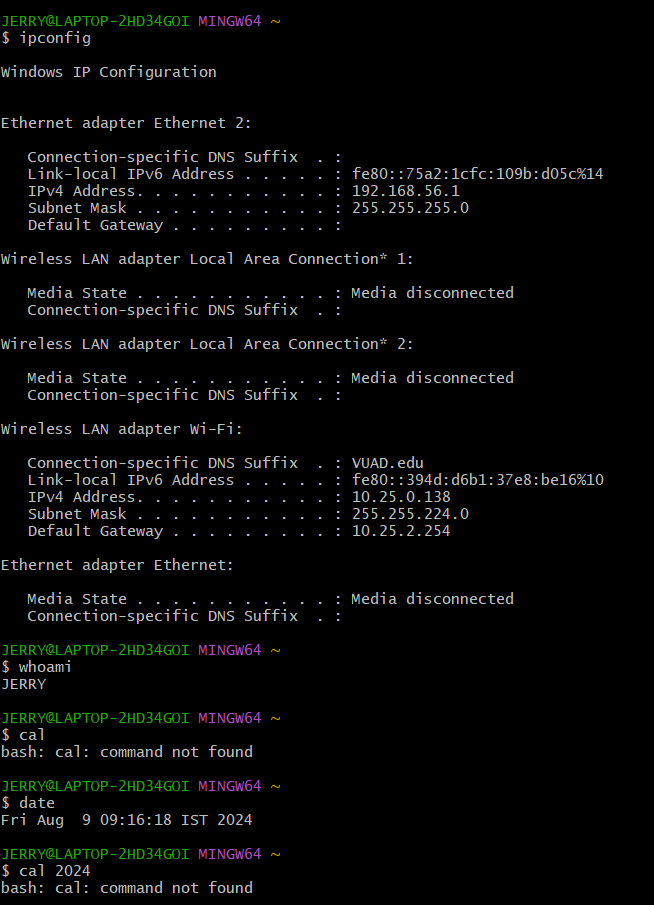
1. **rm - Remove files or directories**

****

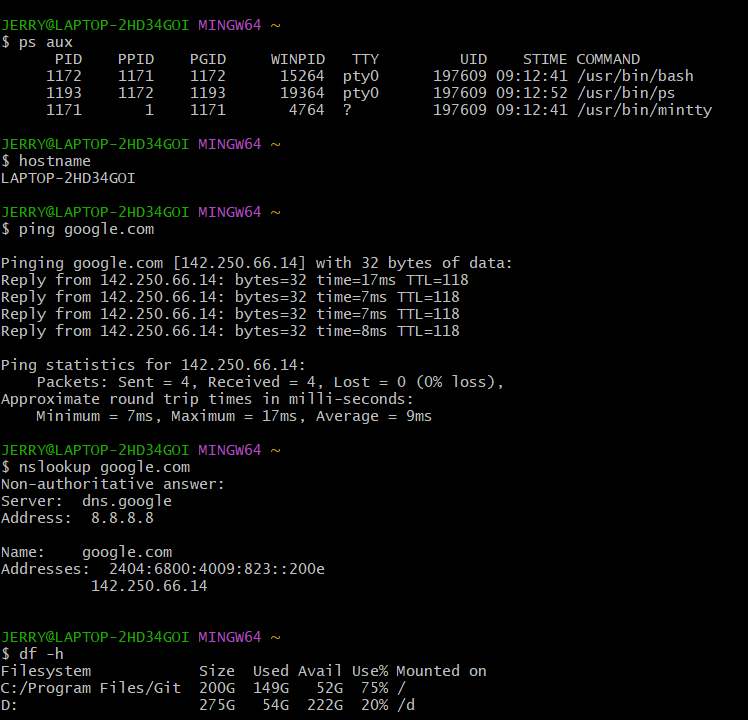
1. **rmdir - Removes an empty directory**

****

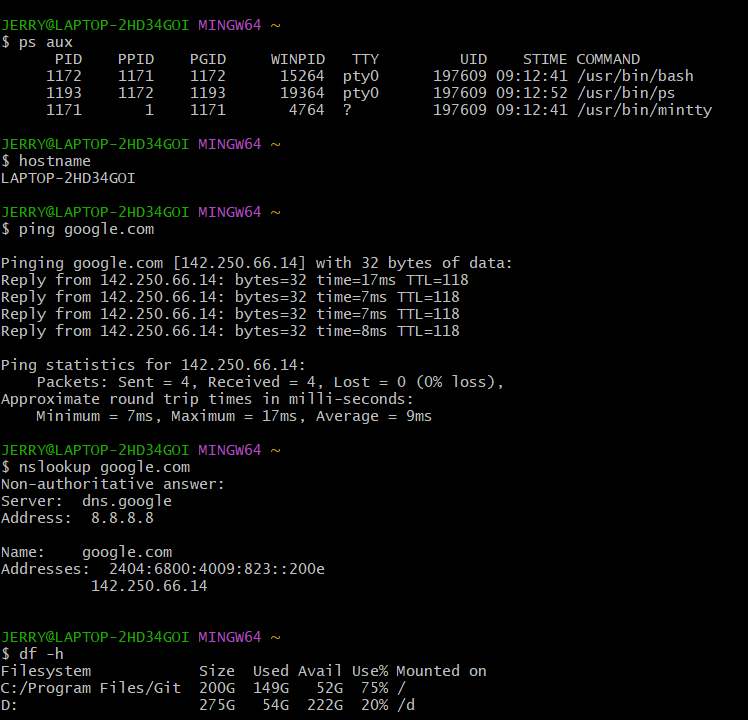
1. **ifconfig - Displays or configures network interfaces and their settings**

****

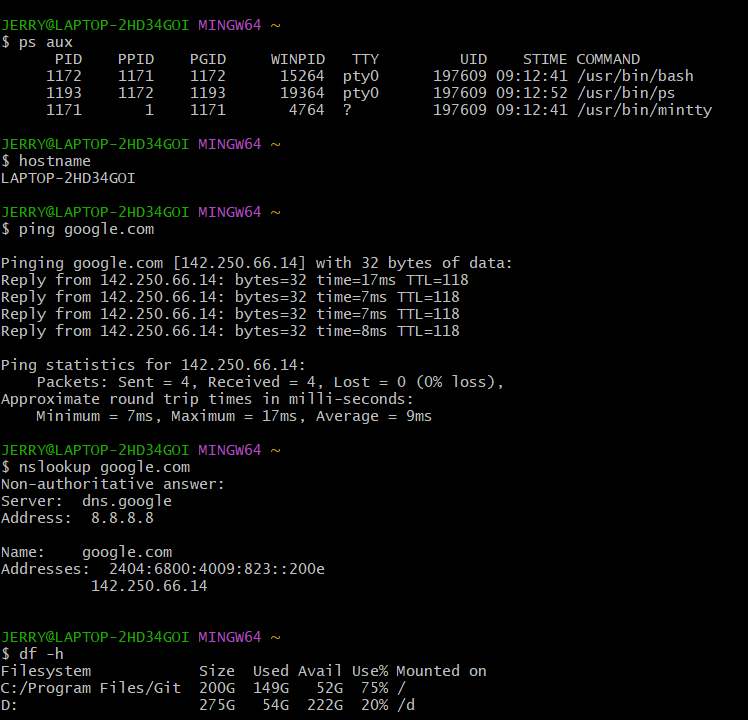
1. **df -h - Displays disk space usage**

****

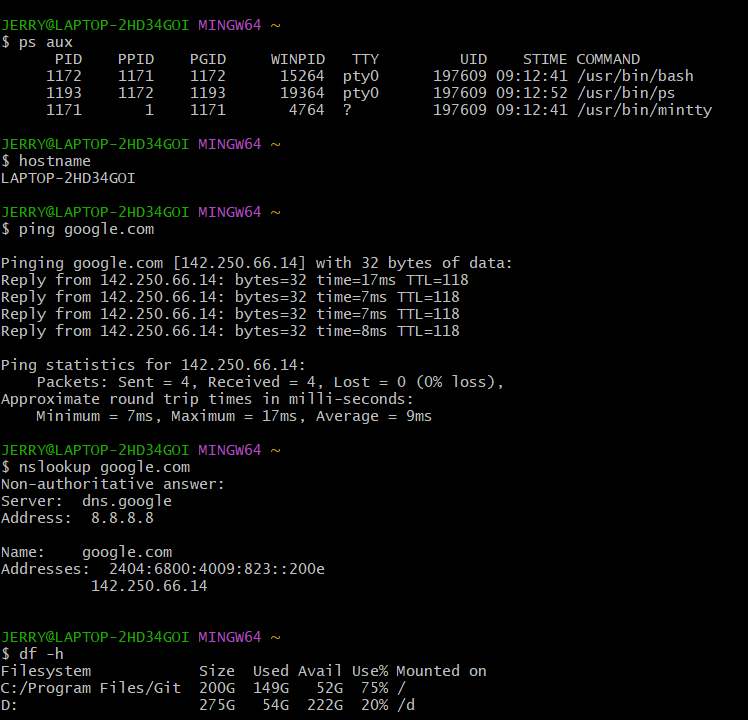
1. **hostname - Displays the system’s hostname’**

****

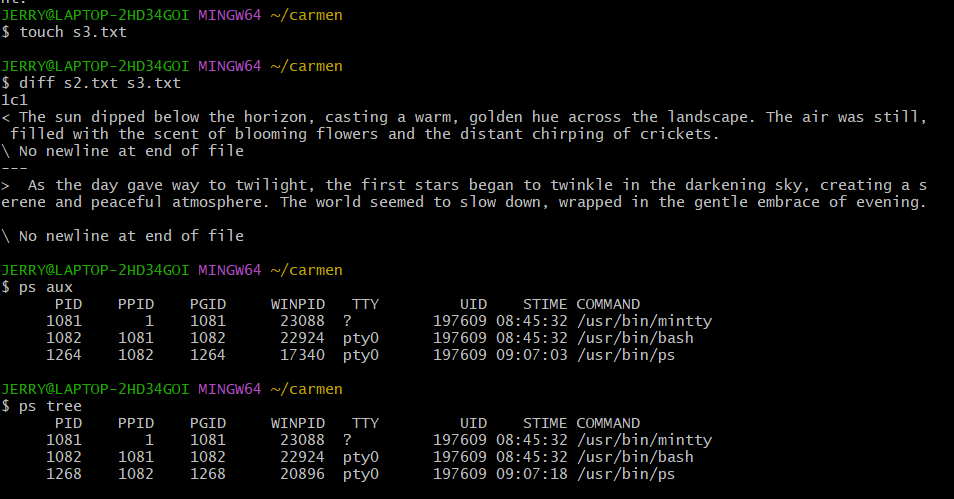
1. **ping - Tests connectivity with another host using ICMP echo requests**

****

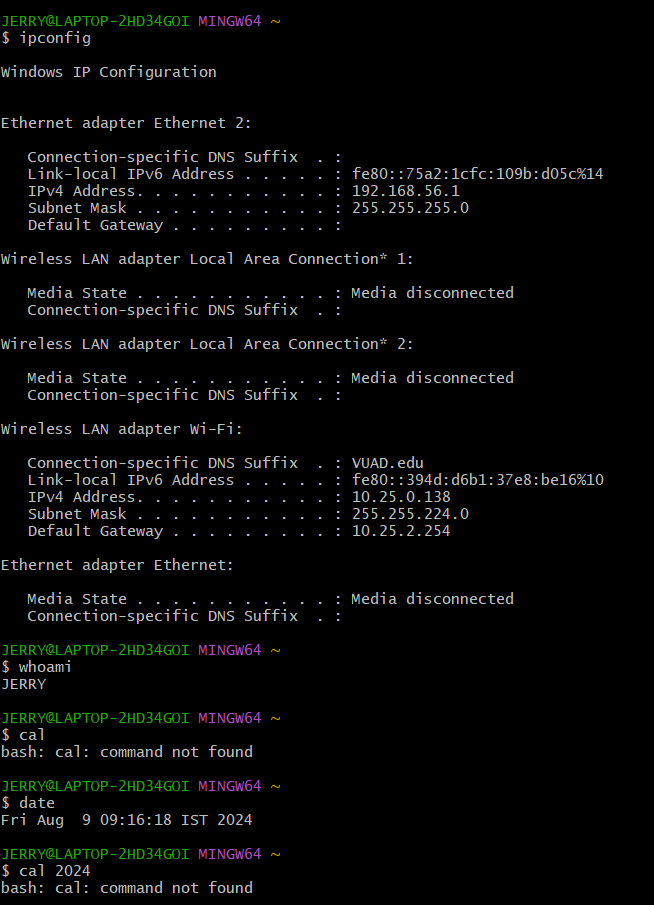
1. **nslookup - Queries DNS servers for domain name resolution and IP address information**

****

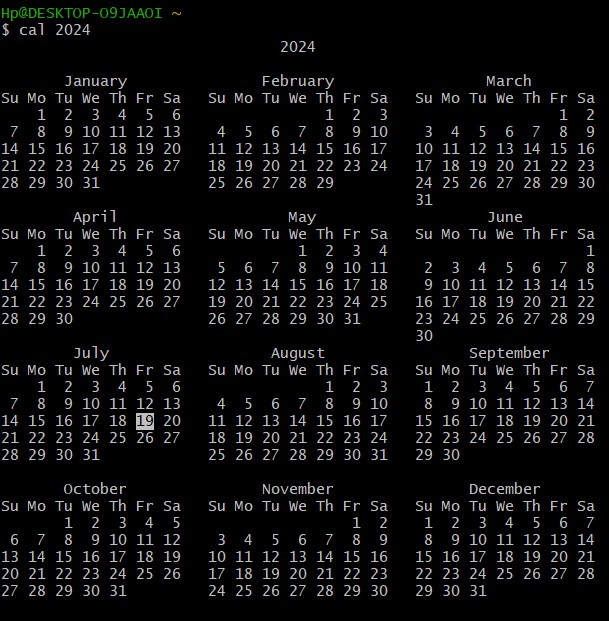
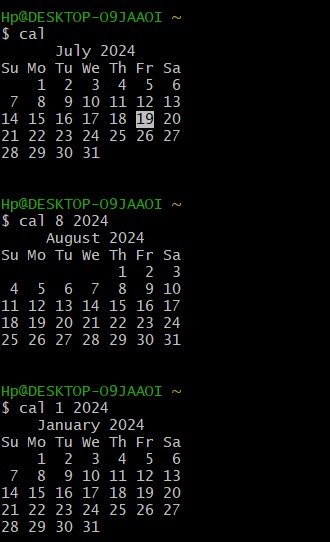
1. **ps aux/ ps tree - a common combination of options that provides detailed information about all processes running on the system**

****

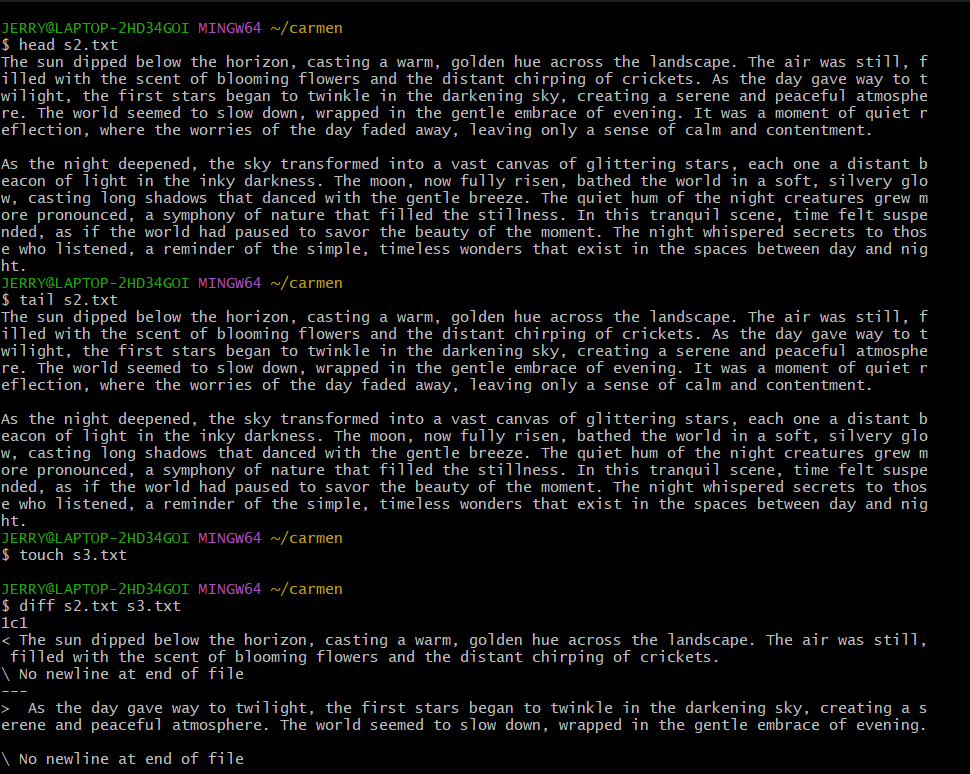
1. **whoami - displaying the username of the current user who is executing the command**

****

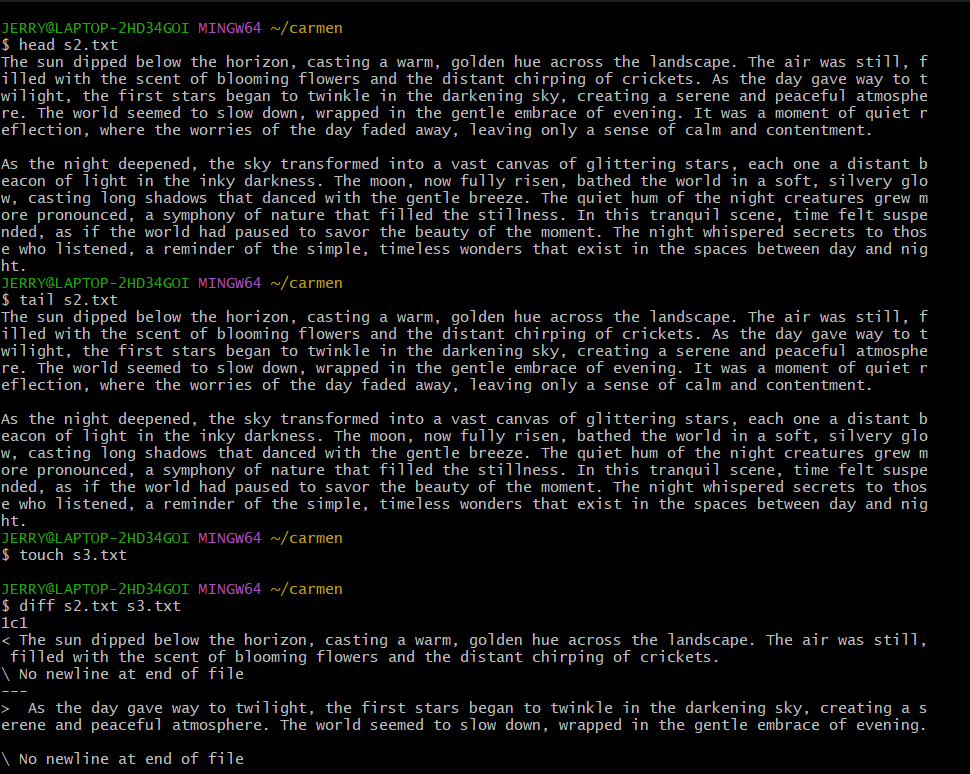
1. **cal - displaying a calendar in the terminal**



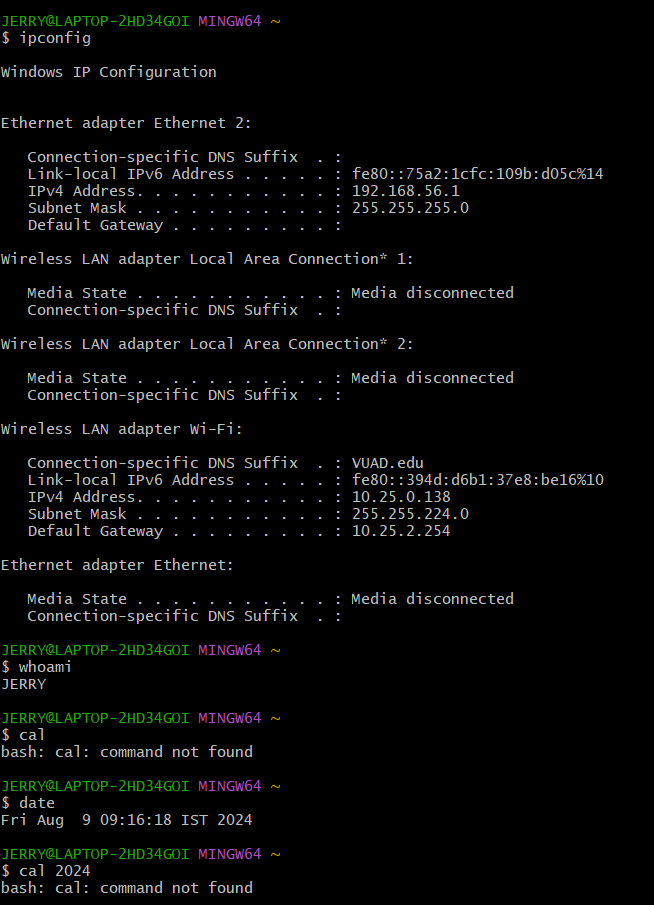
1. **head - Displays the first few lines of a file**

****

1. **tail - Displays the last few lines of a file**

****

1. **date - displays the date in the time zone on which unix operating system is configured**

****

1. **kill - Terminates processes using their process IDs (PIDs)**

****

**4.** **Implement following shell programs**

**a. To find out if a given string is a palindrome or not**

#!/bin/bash

# Function to check if a string is a palindrome

is\_palindrome() {

local str="$1"

local reversed\_str=""

# Reverse the string using a for loop

len=${#str}

for (( i=$len-1; i>=0; i-- )); do

reversed\_str="${reversed\_str}${str:$i:1}"

done

# Compare original string with reversed string

if [ "$str" == "$reversed\_str" ]; then

echo "The string '$str' is a palindrome."

else

echo "The string '$str' is not a palindrome."

fi

}

# Read input from the user

read -p "Enter a string: " input\_string

is\_palindrome "$input\_string"

chmod +x palindrome.sh

./palindrome.sh

**5. Implement following shell programs**

**b. To sort numbers of a given array using bubble sort**

#!/bin/bash

# Function to perform bubble sort

bubble\_sort() {

local arr=("$@")

local n=${#arr[@]}

# Bubble sort algorithm

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

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

if [ ${arr[j]} -gt ${arr[j+1]} ]; then

# Swap if the element is greater than the next

temp=${arr[j]}

arr[j]=${arr[j+1]}

arr[j+1]=$temp

fi

done

done

# Print the sorted array

echo "Sorted Array: ${arr[@]}"

}

# Read input numbers from the user

read -p "Enter numbers separated by space: " -a numbers

bubble\_sort "${numbers[@]}"

chmod +x bubble.sh

./bubble.sh

**6.** **Generate a student report using Awk programming**

**Sample Input File: students.txt**

txt

Copy code

Name GPA Year Major Gender State

Alice 3.8 Freshman Computer\_Science Female CA

Bob 3.2 Junior Mathematics Male TX

Charlie 3.6 Senior Physics Male NY

Diana 4.0 Sophomore Computer\_Science Female CA

Eve 3.1 Freshman Biology Female FL

Frank 2.9 Junior English Male TX

Grace 3.9 Senior Computer\_Science Female WA

Awk ‘{print}’ student.txt

**1. Print Students with GPA Greater than 3.5**

bash

awk '$2 > 3.5 {print $1, $2}' students.txt

**2. Print Student Names and Their GPAs**

bash

awk '{print $1, $2}' students.txt

**3. Print Students from a Specific State**

bash

awk '$6 == "CA" {print $1, $6}' students.txt

**4. Print Students' Year and Major**

bash

awk '{print $3, $4}' students.txt

**5. Print Students' Names and Their Grades**

Assuming GPA corresponds to grades:

* >= 3.7: A
* 3.0-3.6: B
* < 3.0: C

bash

awk '{

grade = ($2 >= 3.7) ? "A" : ($2 >= 3.0) ? "B" : "C";

print $1, grade;

}' students.txt

**6. Print the Total Number of Students**

bash

Copy code

awk 'END {print "Total Students:", NR - 1}' students.txt

**7. Print the Names of All Female Students**

bash

Copy code

awk '$5 == "Female" {print $1}' students.txt

**8. Print Students Who Are Freshmen**

bash

Copy code

awk '$3 == "Freshman" {print $1, $3}' students.txt

**9. Print Students Whose Major Is Computer Science**

bash

Copy code

awk '$4 == "Computer\_Science" {print $1, $4}' students.txt

**10. Print the Student with the Highest GPA**

bash

Copy code

awk 'NR == 1 {next} $2 > max {max = $2; student = $1} END {print "Highest GPA:", student, max}' students.txt

**11. Print the Student with the Lowest GPA**

bash

Copy code

awk 'NR == 1 {next} $2 < min || NR == 2 {min = $2; student = $1} END {print "Lowest GPA:", student, min}' students.txt

**12. Print the Average GPA of All Students**

bash

Copy code

awk 'NR > 1 {sum += $2; count++} END {print "Average GPA:", sum / count}' students.txt

**13. Print the Number of Students in Each Year**

bash

Copy code

awk 'NR > 1 {years[$3]++} END {for (year in years) print year, years[year]}' students.txt

**14. Print the Number of Students in Each Major**

bash

Copy code

awk 'NR > 1 {majors[$4]++} END {for (major in majors) print major, majors[major]}' students.txt

**15. Print the Number of Students from Each State**

bash

Copy code

awk 'NR > 1 {states[$6]++} END {for (state in states) print state, states[state]}' students.txt

**16. Print All Students in Junior Year**

bash

Copy code

awk '$3 == "Junior" {print $1, $3}' students.txt

**17. Print Students Whose Name Starts with 'A'**

bash

Copy code

awk '$1 ~ /^A/ {print $1}' students.txt

**18. Print Students and Their GPAs, Sorted by GPA (Descending)**

bash

awk 'NR > 1 {print $0}' students.txt | sort -k2,2nr

**19. Print Students Grouped by Gender**

bash

Copy code

awk '{genders[$5] = genders[$5] $1 " "} END {for (gender in genders) print gender ":", genders[gender]}' students.txt

**7.** **Solve the Readers-Writers problem**

**a. using threads and semaphores**

**ALGORITHM:**

1. Initialize read\_count = 0

2. Initialize resource\_lock, read\_count\_lock (Critical Sections)

3. Function reader(id):

while True:

Acquire read\_count\_lock

Increment read\_count

if read\_count == 1:

Acquire resource\_lock

Release read\_count\_lock

Print "Reader {id} is reading"

Sleep for reading time

Acquire read\_count\_lock

Decrement read\_count

if read\_count == 0:

Release resource\_lock

Release read\_count\_lock

Sleep for non-critical tim

4. Function writer(id):

while True:

Acquire resource\_lock

Print "Writer {id} is writing"

Sleep for writing time

Release resource\_lock

Sleep for non-critical time

5. In main():

Initialize resource\_lock, read\_count\_lock

Create reader threads (reader function)

Create writer threads (writer function)

Run for predefined time (e.g., 10 seconds)

Terminate threads

Delete resource\_lock, read\_count\_lock

**CODE;**

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>  // For Windows threads

#include <time.h>     // For sleep functionality

// Global variables

CRITICAL\_SECTION resource\_lock;  // Mutex for resource access

CRITICAL\_SECTION read\_count\_lock;  // Mutex for read\_count access

int read\_count = 0;  // Number of readers

// Reader thread function

DWORD WINAPI reader(LPVOID param) {

    int id = \*(int \*)param;

    while (1) {

        // Entry section for readers: update read\_count

        EnterCriticalSection(&read\_count\_lock);

        read\_count++;

        if (read\_count == 1) {

            EnterCriticalSection(&resource\_lock); // First reader locks the resource

        }

        LeaveCriticalSection(&read\_count\_lock);

        // Critical section: reader reads the resource

        printf("Reader %d is reading the resource.\n", id);

        Sleep(1000);  // Simulate reading time

        // Exit section for readers: update read\_count

        EnterCriticalSection(&read\_count\_lock);

        read\_count--;

        if (read\_count == 0) {

            LeaveCriticalSection(&resource\_lock); // Last reader unlocks the resource

        }

        LeaveCriticalSection(&read\_count\_lock);

        Sleep(1000);  // Simulate time between reads

    }

    return 0;

}

// Writer thread function

DWORD WINAPI writer(LPVOID param) {

    int id = \*(int \*)param;

    while (1) {

        // Entry section for writers: lock the resource

        EnterCriticalSection(&resource\_lock);

        // Critical section: writer writes to the resource

        printf("Writer %d is writing to the resource.\n", id);

        Sleep(1000);  // Simulate writing time

        // Exit section for writers: unlock the resource

        LeaveCriticalSection(&resource\_lock);

        Sleep(2000);  // Simulate time between writes

    }

    return 0;

}

int main() {

    int num\_readers = 3, num\_writers = 2;

    HANDLE readers[num\_readers], writers[num\_writers];

    DWORD reader\_ids[num\_readers], writer\_ids[num\_writers];

    // Initialize the critical sections

    InitializeCriticalSection(&resource\_lock);

    InitializeCriticalSection(&read\_count\_lock);

    // Create reader threads

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

        reader\_ids[i] = i + 1;

        readers[i] = CreateThread(NULL, 0, reader, &reader\_ids[i], 0, NULL);

    }

    // Create writer threads

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

        writer\_ids[i] = i + 1;

        writers[i] = CreateThread(NULL, 0, writer, &writer\_ids[i], 0, NULL);

    }

    // Run for a certain period (e.g., 10 seconds)

    Sleep(10000);

    // Clean up: terminate threads and delete critical sections

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

        TerminateThread(readers[i], 0);

        CloseHandle(readers[i]);

    }

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

        TerminateThread(writers[i], 0);

        CloseHandle(writers[i]);

    }

    DeleteCriticalSection(&resource\_lock);

    DeleteCriticalSection(&read\_count\_lock);

    printf("Program terminated.\n");

    return 0;

}

**8. Solve the Readers-Writers problem**

**b. using threads and mutex**

**ALGORITHM:**

// Initialize global variables

read\_count = 0

resource\_mutex = Mutex() // Mutex to control access to shared resource

read\_count\_mutex = Mutex() // Mutex to protect read\_count variable

// Reader thread function

function reader(reader\_id):

while True:

// Entry Section

acquire(read\_count\_mutex) // Lock read\_count mutex

read\_count += 1

if read\_count == 1: // If this is the first reader

acquire(resource\_mutex) // Lock the resource to prevent writers from writing

release(read\_count\_mutex) // Unlock read\_count mutex

// Critical Section

print("Reader " + reader\_id + " is reading the resource.")

sleep(random time) // Simulate reading for a random time

// Exit Section

acquire(read\_count\_mutex) // Lock read\_count mutex

read\_count -= 1

if read\_count == 0: // If this is the last reader

release(resource\_mutex) // Unlock the resource for writers

release(read\_count\_mutex) // Unlock read\_count mutex

// Non-Critical Section

sleep(random time) // Simulate time between read operations

// Writer thread function

function writer(writer\_id):

while True:

// Entry Section

acquire(resource\_mutex) // Lock the resource to ensure exclusive write access

// Critical Section

print("Writer " + writer\_id + " is writing to the resource.")

sleep(random time) // Simulate writing for a random time

// Exit Section

release(resource\_mutex) // Unlock the resource after writing

// Non-Critical Section

sleep(random time) // Simulate time between write operations

// Main program

function main():

// Create threads

create reader\_threads for 3 readers

create writer\_threads for 2 writers

// Let threads run for a fixed time (e.g., 5 seconds)

sleep(5 seconds)

// Stop the threads after the specified time

stop threads

// Cleanup resources

close all threads

close resource\_mutex

close read\_count\_mutex

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

HANDLE resource\_mutex; // Mutex to protect the resource

HANDLE read\_count\_mutex; // Mutex to protect read\_count

int read\_count = 0; // Variable to track number of readers

// Reader thread function

DWORD WINAPI reader(LPVOID param) {

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

while (1) {

// Entry Section

WaitForSingleObject(read\_count\_mutex, INFINITE);

read\_count++;

if (read\_count == 1) {

WaitForSingleObject(resource\_mutex, INFINITE); // First reader locks the resource

}

ReleaseMutex(read\_count\_mutex);

// Critical Section (Reading)

printf("Reader %d is reading the resource.\n", id);

Sleep(500); // Simulate reading time (500ms)

// Exit Section

WaitForSingleObject(read\_count\_mutex, INFINITE);

read\_count--;

if (read\_count == 0) {

ReleaseMutex(resource\_mutex); // Last reader releases the resource

}

ReleaseMutex(read\_count\_mutex);

// Non-critical Section

Sleep(500); // Simulate time between reads

}

return 0;

}

// Writer thread function

DWORD WINAPI writer(LPVOID param) {

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

while (1) {

// Entry Section: Writer locks the resource

WaitForSingleObject(resource\_mutex, INFINITE);

// Critical Section (Writing)

printf("Writer %d is writing to the resource.\n", id);

Sleep(500); // Simulate writing time (500ms)

// Exit Section

ReleaseMutex(resource\_mutex); // Writer releases the resource

// Non-critical Section

Sleep(500); // Simulate time between writes

}

return 0;

}

int main() {

// Initialize mutexes

resource\_mutex = CreateMutex(NULL, FALSE, NULL);

read\_count\_mutex = CreateMutex(NULL, FALSE, NULL);

// Create reader and writer threads

HANDLE readers[3], writers[2];

int reader\_ids[3] = {1, 2, 3};

int writer\_ids[2] = {1, 2};

for (int i = 0; i < 3; i++) {

readers[i] = CreateThread(NULL, 0, reader, &reader\_ids[i], 0, NULL);

}

for (int i = 0; i < 2; i++) {

writers[i] = CreateThread(NULL, 0, writer, &writer\_ids[i], 0, NULL);

}

// Run for a certain period (e.g., 5 seconds)

Sleep(5000); // Sleep for 5 seconds

// Cleanup: Close the threads and mutexes

for (int i = 0; i < 3; i++) {

CloseHandle(readers[i]);

}

for (int i = 0; i < 2; i++) {

CloseHandle(writers[i]);

}

CloseHandle(resource\_mutex);

CloseHandle(read\_count\_mutex);

printf("Program terminated.\n");

return 0;

}

**9.** **Solve the Producer Consumer problem**

**a. using threads and semaphores**

**ALGORITHM:**

**INITIALIZE buffer[BUFFER\_SIZE] TO EMPTY**

**INITIALIZE empty TO BUFFER\_SIZE, full TO 0, mutex TO UNLOCKED**

**SET items\_to\_produce TO 10, items\_produced TO 0, items\_consumed TO 0**

**CREATE THREAD PRODUCER**

**WHILE items\_produced < items\_to\_produce DO**

**item ← GENERATE RANDOM ITEM**

**WAIT(empty) // Ensure there’s an empty slot**

**WAIT(mutex) // Lock access to buffer**

**ADD item TO BUFFER**

**INCREMENT items\_produced**

**SIGNAL(mutex) // Unlock buffer**

**SIGNAL(full) // Signal an item is available**

**SLEEP(SHORT\_DELAY) // Simulate production time**

**END WHILE**

**END THREAD**

**CREATE THREAD CONSUMER**

**WHILE items\_consumed < items\_to\_produce DO**

**WAIT(full) // Ensure there’s an item to consume**

**WAIT(mutex) // Lock access to buffer**

**item ← REMOVE FROM BUFFER**

**INCREMENT items\_consumed**

**SIGNAL(mutex) // Unlock buffer**

**SIGNAL(empty) // Signal a slot is available**

**SLEEP(SHORT\_DELAY) // Simulate consumption time**

**END WHILE**

**END THREAD**

**WAIT FOR PRODUCER AND CONSUMER TO FINISH**

**CLEANUP semaphores AND mutex**

**PRINT "Production and Consumption completed."**

**Code**

#include <stdio.h>

#include <stdlib.h>

#include <windows.h>

#define BUFFER\_SIZE 10

int buffer[BUFFER\_SIZE] = {0}; // Shared buffer

// Variables to track items produced and consumed

int items\_to\_produce = 10;  // Total items to produce and consume

int items\_produced = 0;

int items\_consumed = 0;

// Handles for semaphores and mutex

HANDLE empty;

HANDLE full;

HANDLE mutex;

// Producer thread function

DWORD WINAPI producer(LPVOID param) {

    int item;

    while (1) {

        if (items\_produced >= items\_to\_produce) {

            break; // Stop producing after the required items

        }

        item = rand() % 100; // Generate a random item

        // Wait for an empty slot and acquire the mutex

        WaitForSingleObject(empty, INFINITE);

        WaitForSingleObject(mutex, INFINITE);

        // Add the item to the buffer

        printf("Produced: %d\n", item);

        for (int i = 0; i < BUFFER\_SIZE; i++) {

            if (buffer[i] == 0) {

                buffer[i] = item;

                break;

            }

        }

        items\_produced++; // Increment produced count

        // Release the mutex and signal that a slot is filled

        ReleaseMutex(mutex);

        ReleaseSemaphore(full, 1, NULL);

        Sleep(500); // Simulate production time

    }

    return 0;

}

// Consumer thread function

DWORD WINAPI consumer(LPVOID param) {

    int item;

    while (1) {

        if (items\_consumed >= items\_to\_produce) {

            break; // Stop consuming after the required items

        }

        // Wait for a filled slot and acquire the mutex

        WaitForSingleObject(full, INFINITE);

        WaitForSingleObject(mutex, INFINITE);

        // Remove the item from the buffer

        for (int i = 0; i < BUFFER\_SIZE; i++) {

            if (buffer[i] != 0) {

                item = buffer[i];

                buffer[i] = 0;

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

                break;

            }

        }

        items\_consumed++; // Increment consumed count

        // Release the mutex and signal that a slot is empty

        ReleaseMutex(mutex);

        ReleaseSemaphore(empty, 1, NULL);

        Sleep(500); // Simulate consumption time

    }

    return 0;

}

int main() {

    // Initialize semaphores and mutex

    empty = CreateSemaphore(NULL, BUFFER\_SIZE, BUFFER\_SIZE, NULL);

    full = CreateSemaphore(NULL, 0, BUFFER\_SIZE, NULL);

    mutex = CreateMutex(NULL, FALSE, NULL);

    // Create producer and consumer threads

    HANDLE producer\_thread = CreateThread(NULL, 0, producer, NULL, 0, NULL);

    HANDLE consumer\_thread = CreateThread(NULL, 0, consumer, NULL, 0, NULL);

    // Wait for both threads to finish

    WaitForSingleObject(producer\_thread, INFINITE);

    WaitForSingleObject(consumer\_thread, INFINITE);

    // Cleanup resources

    CloseHandle(empty);

    CloseHandle(full);

    CloseHandle(mutex);

    printf("Production and consumption of %d items completed.\n", items\_to\_produce);

    return 0;

}

**10. Solve the Producer Consumer problem**

**b. using threads and mutex**

**ALGORITHM:**

**Initialize BUFFER\_SIZE = 10, buffer[BUFFER\_SIZE], in = 0, out = 0**

**Initialize mutex, buffer\_not\_full, buffer\_not\_empty**

**Producer Function:**

**for i = 1 to 10:**

**item = generate random item**

**acquire(mutex) // Lock the buffer**

**while (buffer is full):**

**wait(buffer\_not\_full, mutex) // Wait for space in the buffer**

**buffer[in] = item // Produce item**

**print "Produced:", item**

**in = (in + 1) % BUFFER\_SIZE**

**signal(buffer\_not\_empty) // Notify consumer that an item is available**

**release(mutex) // Unlock the buffer**

**sleep(1) // Simulate production delay**

**Consumer Function:**

**for i = 1 to 10:**

**acquire(mutex) // Lock the buffer**

**while (buffer is empty):**

**wait(buffer\_not\_empty, mutex) // Wait for item to be available**

**item = buffer[out] // Consume item**

**print "Consumed:", item**

**buffer[out] = 0 // Optional: clear consumed item**

**out = (out + 1) % BUFFER\_SIZE**

**signal(buffer\_not\_full) // Notify producer that space is available**

**release(mutex) // Unlock the buffer**

**sleep(1) // Simulate consumption delay**

**Main Function:**

**initialize mutex and condition variables**

**create producer and consumer threads**

**wait for both threads to finish**

**clean up resources (mutex, condition variables)**

**print "Production and consumption completed."**

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <windows.h> // For Windows-specific thread and synchronization

// Define the buffer size and the buffer itself

#define BUFFER\_SIZE 10

int buffer[BUFFER\_SIZE];

int in = 0; // Index for the producer to insert

int out = 0; // Index for the consumer to remove

// Mutex and condition variables for synchronization

HANDLE mutex;

HANDLE buffer\_not\_full;

HANDLE buffer\_not\_empty;

// Function for producer thread

DWORD WINAPI producer(LPVOID param) {

for (int i = 0; i < 10; i++) { // Produce 10 items

int item = rand() % 100; // Generate a random item

WaitForSingleObject(mutex, INFINITE); // Lock the buffer

while ((in + 1) % BUFFER\_SIZE == out) { // Buffer is full

ReleaseMutex(mutex);

Sleep(100); // Sleep for a while to allow consumer to consume

WaitForSingleObject(mutex, INFINITE);

}

// Add the item to the buffer

buffer[in] = item;

printf("Produced: %d\n", item);

in = (in + 1) % BUFFER\_SIZE;

// Signal consumer that buffer is not empty

ReleaseSemaphore(buffer\_not\_empty, 1, NULL);

ReleaseMutex(mutex); // Unlock the buffer

Sleep(1000); // Simulate production delay

}

return 0;

}

// Function for consumer thread

DWORD WINAPI consumer(LPVOID param) {

for (int i = 0; i < 10; i++) { // Consume 10 items

WaitForSingleObject(mutex, INFINITE); // Lock the buffer

while (in == out) { // Buffer is empty

ReleaseMutex(mutex);

WaitForSingleObject(buffer\_not\_empty, INFINITE); // Wait until buffer has an item

WaitForSingleObject(mutex, INFINITE); // Lock the buffer again

}

// Remove the item from the buffer

int item = buffer[out];

buffer[out] = 0; // Optional: clear the buffer slot

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

out = (out + 1) % BUFFER\_SIZE;

// Signal producer that buffer has space

ReleaseSemaphore(buffer\_not\_full, 1, NULL);

ReleaseMutex(mutex); // Unlock the buffer

Sleep(1000); // Simulate consumption delay

}

return 0;

}

int main() {

// Initialize mutex and condition variables (semaphores in this case)

mutex = CreateMutex(NULL, FALSE, NULL);

buffer\_not\_full = CreateSemaphore(NULL, BUFFER\_SIZE, BUFFER\_SIZE, NULL);

buffer\_not\_empty = CreateSemaphore(NULL, 0, BUFFER\_SIZE, NULL);

// Create producer and consumer threads

HANDLE producer\_thread = CreateThread(NULL, 0, producer, NULL, 0, NULL);

HANDLE consumer\_thread = CreateThread(NULL, 0, consumer, NULL, 0, NULL);

// Wait for threads to finish

WaitForSingleObject(producer\_thread, INFINITE);

WaitForSingleObject(consumer\_thread, INFINITE);

// Cleanup

CloseHandle(mutex);

CloseHandle(buffer\_not\_full);

CloseHandle(buffer\_not\_empty);

printf("Production and Consumption completed.\n");

return 0;

}

**11.** **Simulate the following CPU scheduling algorithms:**

**A. First come First serve:**

**ALGORITHM:**

FCFS Scheduling:

1. Sort the processes based on their arrival times (if not already sorted).

2. For each process, calculate the waiting time and turnaround time:

- Waiting Time for P1 = 0

- Waiting Time for P2 = Burst Time of P1

- Waiting Time for P3 = Burst Time of P1 + Burst Time of P2

...

3. Calculate the Turnaround Time:

- Turnaround Time for P1 = Waiting Time + Burst Time

4. Display the results for each process:

- Process ID, Burst Time, Waiting Time, Turnaround Time

5. Calculate the average waiting and turnaround times.

**CODE:**

#include <stdio.h>

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

wt[0] = 0; // The first process has no waiting time

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

wt[i] = bt[i - 1] + wt[i - 1]; // Waiting time of each process

}

}

void findTurnaroundTime(int bt[], int wt[], int tat[], int n) {

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

tat[i] = bt[i] + wt[i]; // Turnaround time = Burst time + Waiting time

}

}

void findAvgTime(int bt[], int n) {

int wt[n], tat[n];

findWaitingTime(bt, wt, n);

findTurnaroundTime(bt, wt, tat, n);

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

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

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

total\_wt += wt[i];

total\_tat += tat[i];

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

}

printf("\nExplanation:\n");

printf("FCFS executes processes in the order they arrive, without preemption.\n");

printf("For each process, the waiting time is the sum of the burst times of all previous processes.\n");

printf("Turnaround time is the sum of burst time and waiting time for each process.\n");

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

printf("Average Turnaround Time: %.2f\n", total\_tat / n);

}

int main() {

int n;

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

scanf("%d", &n);

int bt[n];

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

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

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

}

findAvgTime(bt, n);

return 0;

}

**B.** **Shortest Job First (Non-preemptive):**

**ALGORITHM:**

SJF (Non-preemptive) Scheduling:

1. Sort the processes by burst time.

2. For each process, calculate:

- Waiting Time for P1 = 0

- Waiting Time for P2 = Burst Time of P1

- Waiting Time for P3 = Burst Time of P1 + Burst Time of P2

...

3. Calculate the Turnaround Time:

- Turnaround Time for P1 = Waiting Time + Burst Time

4. Display the results for each process:

- Process ID, Burst Time, Waiting Time, Turnaround Time

5. Calculate the average waiting and turnaround times.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

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

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

}

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

wt[0] = 0; // The first process has no waiting time

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

wt[i] = bt[i - 1] + wt[i - 1]; // Waiting time of each process

}

}

void findTurnaroundTime(int bt[], int wt[], int tat[], int n) {

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

tat[i] = bt[i] + wt[i]; // Turnaround time = Burst time + Waiting time

}

}

void findAvgTime(int bt[], int n) {

int wt[n], tat[n];

findWaitingTime(bt, wt, n);

findTurnaroundTime(bt, wt, tat, n);

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

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

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

total\_wt += wt[i];

total\_tat += tat[i];

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

}

printf("\nExplanation:\n");

printf("SJF (Non-preemptive) schedules processes by burst time.\n");

printf("Processes with the shortest burst time are executed first.\n");

printf("Waiting time is calculated by summing the burst times of all previous processes.\n");

printf("Turnaround time is the sum of burst time and waiting time.\n");

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

printf("Average Turnaround Time: %.2f\n", total\_tat / n);

}

int main() {

int n;

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

scanf("%d", &n);

int bt[n];

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

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

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

}

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

findAvgTime(bt, n);

return 0;

}

**C.** **Shortest Job First (Preemptive)**

**Algorithm:**

**CODE:**

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {

int rem\_bt[n], t = 0;

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

rem\_bt[i] = bt[i];

}

while (1) {

int done = 1;

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

if (rem\_bt[i] > 0) {

done = 0;

if (rem\_bt[i] > quantum) {

rem\_bt[i] -= quantum;

t += quantum;

} else {

t += rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

}

}

}

if (done == 1) break;

}

}

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

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

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

}

}

void findAvgTime(int processes[], int n, int bt[], int quantum) {

int wt[n], tat[n];

findWaitingTime(processes, n, bt, wt, quantum);

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

printf("\nProcess Burst Time Waiting Time Turnaround Time\n");

int total\_wt = 0, total\_tat = 0;

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

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

total\_wt += wt[i];

total\_tat += tat[i];

}

printf("\nAverage Waiting Time = %.2f", (float)total\_wt / n);

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

}

int main() {

int n, quantum;

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

scanf("%d", &n);

int processes[n], burst\_time[n];

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

processes[i] = i + 1; // Process IDs

}

printf("Enter the time quantum: ");

scanf("%d", &quantum);

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

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

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

}

findAvgTime(processes, n, burst\_time, quantum);

return 0;

}

**D.** **Round Robin CODE:**

**ALGORITHM:**

Round Robin Scheduling:

1. Initialize all processes in the ready queue.

2. For each process:

- If the burst time <= time quantum, execute the process completely.

- If the burst time > time quantum, execute the process for the quantum time and put it back in the queue with the remaining burst time.

3. Calculate the waiting time and turnaround time for each process:

- Waiting Time = Time spent in the ready queue before execution.

- Turnaround Time = Waiting Time + Burst Time.

4. Display the results for each process.

5. Calculate the average waiting and turnaround times.

**CODE:**

#include <stdio.h>

void findWaitingTime(int bt[], int wt[], int n, int q) {

int rem\_bt[n];

for (int i = 0; i < n; i++) rem\_bt[i] = bt[i];

int t = 0;

while (1) {

int done = 1;

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

if (rem\_bt[i] > 0) {

done = 0;

if (rem\_bt[i] > q) {

rem\_bt[i] -= q;

t += q;

} else {

t += rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

}

}

}

if (done == 1) break;

}

}

void findTurnaroundTime(int bt[], int wt[], int tat[], int n) {

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

tat[i] = bt[i] + wt[i]; // Turnaround time = Burst time + Waiting time

}

}

void findAvgTime(int bt[], int n, int q) {

int wt[n], tat[n];

findWaitingTime(bt, wt, n, q);

findTurnaroundTime(bt, wt, tat, n);

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

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

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

total\_wt += wt[i];

total\_tat += tat[i];

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

}

printf("\nExplanation:\n");

printf("Round Robin executes processes in a cyclic order.\n");

printf("Each process is given a fixed time quantum (time slice) to execute.\n");

printf("If a process doesn't finish in the quantum, it's put back in the ready queue.\n");

printf("Waiting time is calculated as the time a process spends waiting in the ready queue.\n");

printf("Turnaround time is the total time taken for the process to complete.\n");

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

printf("Average Turnaround Time: %.2f\n", total\_tat / n);

}

int main() {

int n, q;

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

scanf("%d", &n);

printf("Enter the time quantum: ");

scanf("%d", &q);

int bt[n];

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

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

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

}

findAvgTime(bt, n, q);

return 0;

}

**E.** **Priority (Non-preemptive) CODE:**

**ALGORITHM:**

Priority Scheduling (Non-preemptive):

1. Sort the processes based on their priority (highest priority first).

2. Execute each process in the sorted order:

- Waiting Time for P1 = 0

- Waiting Time for P2 = Burst Time of P1

- Waiting Time for P3 = Burst Time of P1 + Burst Time of P2

...

3. Calculate the Turnaround Time:

- Turnaround Time for P1 = Waiting Time + Burst Time

4. Display the results for each process:

- Process ID, Priority, Burst Time, Waiting Time, Turnaround Time

5. Calculate the average waiting and turnaround times.

**CODE:**

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int priority[]) {

int completed[n];

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

completed[i] = 0; // Process not completed

}

int t = 0, min\_priority, min\_index;

while (1) {

min\_priority = 9999;

min\_index = -1;

// Find process with the highest priority

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

if (!completed[i] && priority[i] < min\_priority) {

min\_priority = priority[i];

min\_index = i;

}

}

if (min\_index == -1) break; // All processes completed

completed[min\_index] = 1;

wt[min\_index] = t;

t += bt[min\_index];

}

}

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

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

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

}

}

void findAvgTime(int processes[], int n, int bt[], int priority[]) {

int wt[n], tat[n];

findWaitingTime(processes, n, bt, wt, priority);

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

printf("\nProcess Burst Time Priority Waiting Time Turnaround Time\n");

int total\_wt = 0, total\_tat = 0;

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

printf("%d %d %d %d %d\n", processes[i], bt[i], priority[i], wt[i], tat[i]);

total\_wt += wt[i];

total\_tat += tat[i];

}

printf("\nAverage Waiting Time = %.2f", (float)total\_wt / n);

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

}

int main() {

int n;

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

scanf("%d", &n);

int processes[n], burst\_time[n], priority[n];

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

processes[i] = i + 1; // Process IDs

}

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

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

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

}

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

printf("Enter priority for process %d: ", i + 1);

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

}

findAvgTime(processes, n, burst\_time, priority);

return 0;

}

**F.** **Priority (preemptive)**

**ALGORITHM:**

Priority Scheduling (Preemptive):

1. For each process, assign a priority.

2. Start execution with the highest priority process.

3. If a new process with higher priority arrives, preempt the currently running process.

4. Calculate the waiting time for each process:

- Waiting Time = Time spent waiting in the ready queue.

5. Calculate the turnaround time:

- Turnaround Time = Waiting Time + Burst Time.

6. Display the results for each process.

7. Calculate the average waiting and turnaround times.

**CODE:**

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int priority[]) {

int completed[n];

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

completed[i] = 0; // Process not completed

}

int t = 0;

while (1) {

int min\_priority = 9999;

int min\_index = -1;

// Find the process with the highest priority that is not completed

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

if (!completed[i] && priority[i] < min\_priority) {

min\_priority = priority[i];

min\_index = i;

}

}

if (min\_index == -1) break; // All processes completed

// Execute the selected process

wt[min\_index] = t;

t += bt[min\_index];

completed[min\_index] = 1;

}

}

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

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

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

}

}

void findAvgTime(int processes[], int n, int bt[], int priority[]) {

int wt[n], tat[n];

findWaitingTime(processes, n, bt, wt, priority);

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

printf("\nProcess Burst Time Priority Waiting Time Turnaround Time\n");

int total\_wt = 0, total\_tat = 0;

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

printf("%d %d %d %d %d\n", processes[i], bt[i], priority[i], wt[i], tat[i]);

total\_wt += wt[i];

total\_tat += tat[i];

}

printf("\nAverage Waiting Time = %.2f", (float)total\_wt / n);

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

}

int main() {

int n;

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

scanf("%d", &n);

int processes[n], burst\_time[n], priority[n];

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

processes[i] = i + 1; // Process IDs

}

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

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

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

}

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

printf("Enter priority for process %d: ", i + 1);

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

}

findAvgTime(processes, n, burst\_time, priority);

return 0;

}

**12.** **Implement Banker’s Safety algorithm for Deadlock Avoidance**

**Algorithm:**

Algorithm Banker's Algorithm (Safe State Check):

Input:

n = Number of processes

m = Number of resources

max[n][m] = Maximum demand for each process and resource

allot[n][m] = Currently allocated resources for each process

avail[m] = Available resources

Output:

safeSequence[] = Sequence of processes in the safe state (if found)

Steps:

1. Calculate the need matrix:

need[i][j] = max[i][j] - allot[i][j]

2. Initialize:

work[] = avail[] // Available resources initially

finish[] = false // All processes are unfinished initially

count = 0 // Counter for finished processes

index = 0 // Index for storing safe sequence

3. While count < n (not all processes finished):

found = false

For each process p:

If finish[p] = false:

canFinish = true

For each resource j:

If need[p][j] > work[j]:

canFinish = false

break

If canFinish:

For each resource j:

work[j] = work[j] + allot[p][j] // Release allocated resources

finish[p] = true // Mark process as finished

safeSequence[index] = p // Add process to safe sequence

count++

found = true

Break

If not found, return unsafe state

4. If count == n:

Print "System is in a safe state."

Print safeSequence

Else:

Print "System is in an unsafe state."

**Code:**

#include <stdio.h>

#include <stdbool.h>

// Define constants for the number of processes and resource types

#define MAX\_PROCESSES 3 // Number of processes

#define MAX\_RESOURCES 3 // Number of resource types

// Function to check if the system is in a safe state and find the safe sequence

bool isSafe(int processes[], int avail[], int max[][MAX\_RESOURCES], int allot[][MAX\_RESOURCES], int n, int m, int safeSequence[]) {

int need[n][m]; // Need matrix (max demand - allocated resources)

bool finish[n]; // Array to track which processes are finished

int work[m]; // Work array (available resources)

// Calculate the 'Need' matrix, which shows the remaining resources each process needs

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

for (int j = 0; j < m; j++) {

need[i][j] = max[i][j] - allot[i][j]; // Need = Max - Allot

}

}

// Initialize the work array (resources available in the system)

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

work[i] = avail[i]; // Work = Available resources

}

// Initialize the finish array to false (no process is finished initially)

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

finish[i] = false;

}

int count = 0; // Counter to track how many processes have been completed

int index = 0; // Index for storing the safe sequence

// Loop until all processes are finished or no safe sequence is found

while (count < n) {

bool found = false; // Flag to track if a process can be finished

// Try to find a process that can finish

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

if (!finish[p]) { // Process not finished yet

bool canFinish = true; // Flag to check if process can finish

// Check if all the resources required by process p can be allocated from the available resources

for (int j = 0; j < m; j++) {

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

canFinish = false; // If need is more than available, can't finish the process

break;

}

}

// If the process can finish, simulate it and update the resources

if (canFinish) {

for (int j = 0; j < m; j++) {

work[j] += allot[p][j]; // Add allocated resources back to the available pool

}

finish[p] = true; // Mark the process as finished

safeSequence[index++] = p; // Add process to the safe sequence

found = true;

count++; // Increment the count of finished processes

break; // Exit the loop to start checking for another process

}

}

}

// If no process can be finished, it means the system is in an unsafe state

if (!found) {

return false; // Return false indicating unsafe state

}

}

return true; // Return true indicating the system is in a safe state

}

int main() {

int n = MAX\_PROCESSES, m = MAX\_RESOURCES; // Set the number of processes and resources

// Initialize the process IDs (for display purposes)

int processes[n];

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

processes[i] = i;

}

// Available resources in the system

int avail[MAX\_RESOURCES] = {3, 3, 2};

// Maximum demand for each process

int max[MAX\_PROCESSES][MAX\_RESOURCES] = {

{7, 5, 3}, // Process 0 requires a maximum of 7 units of resource 0, 5 units of resource 1, and 3 units of resource 2

{3, 2, 2}, // Process 1 requires a maximum of 3 units of resource 0, 2 units of resource 1, and 2 units of resource 2

{9, 0, 2} // Process 2 requires a maximum of 9 units of resource 0, 0 units of resource 1, and 2 units of resource 2

};

// The resources currently allocated to each process

int allot[MAX\_PROCESSES][MAX\_RESOURCES] = {

{3, 2, 2}, // Process 0 has 3 units of resource 0, 2 units of resource 1, and 2 units of resource 2 allocated

{2, 1, 1}, // Process 1 has 2 units of resource 0, 1 unit of resource 1, and 1 unit of resource 2 allocated

{3, 1, 2} // Process 2 has 3 units of resource 0, 1 unit of resource 1, and 2 units of resource 2 allocated

};

// Array to store the safe sequence if the system is in a safe state

int safeSequence[n];

// Run the safety algorithm to check if the system is in a safe state and find the safe sequence

if (isSafe(processes, avail, max, allot, n, m, safeSequence)) {

// If the system is in a safe state, print the safe sequence

printf("System is in a safe state.\n");

printf("Safe Sequence: ");

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

printf("P%d ", safeSequence[i]); // Display the safe sequence

}

printf("\n");

} else {

// If the system is not in a safe state, print an unsafe state message

printf("System is in an unsafe state.\n");

}

return 0;

}

**13.** **Simulate the following page replacement algorithms FIFO**

**ALGORITHM:**

FIFO(pages, n):

// Initialize the frames with -1 indicating that the frame is empty

Initialize frames[FRAME\_SIZE] to -1

page\_faults = 0 // Counter for the number of page faults

// Iterate through each page in the reference string

for each page in pages:

// Check if the current page is already present in the frames

if page not in frames:

// If the page is not in the frames, replace the oldest page

Replace the oldest page in frames with the current page

Increment page\_faults // Increase the page fault count

// Print the current contents of the frames after each page reference

Print current frame contents

// After processing all pages, print the total number of page faults

Print total page faults

**CODE:**

#include <stdio.h>

#define FRAME\_SIZE 4

void FIFO(int pages[], int n) {

int frames[FRAME\_SIZE] = {-1, -1, -1, -1};

int page\_faults = 0, index = 0;

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

int page = pages[i], found = 0;

for (int j = 0; j < FRAME\_SIZE; j++) {

if (frames[j] == page) {

found = 1;

break;

}

}

if (!found) {

frames[index] = page;

index = (index + 1) % FRAME\_SIZE;

page\_faults++;

}

printf("Page %d -> ", page);

for (int j = 0; j < FRAME\_SIZE; j++) {

printf("%d ", frames[j] == -1 ? 0 : frames[j]);

}

printf("\n");

}

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

}

int main() {

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

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

FIFO(pages, n);

return 0;

}

**14.** **Simulate the following page replacement algorithms LRU (Least Recently Used)**

**ALGORITHM:**

LRU(pages, n):

// Initialize the frames with -1 indicating that the frame is empty

Initialize frames[FRAME\_SIZE] to -1

page\_faults = 0 // Counter for the number of page faults

// Iterate through each page in the reference string

for each page in pages:

// Check if the current page is already present in the frames

if page not in frames:

// If the page is not in the frames, we need to replace an existing page

page\_faults++ // Increase the page fault count

// Find the Least Recently Used (LRU) page in the frames

lru\_index = Find the LRU page (the one with the oldest access time)

// Replace the LRU page with the current page

Replace LRU page with current page

// Print the current contents of the frames after each page reference

Print current frame contents

// After processing all pages, print the total number of page faults

Print total page faults

**CODE:**

#include <stdio.h>

#define FRAME\_SIZE 4

void LRU(int pages[], int n) {

int frames[FRAME\_SIZE] = {-1, -1, -1, -1};

int page\_faults = 0;

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

int page = pages[i], found = 0, lru\_index = -1;

for (int j = 0; j < FRAME\_SIZE; j++) {

if (frames[j] == page) {

found = 1;

break;

}

}

if (!found) {

page\_faults++;

int lru\_time = n;

for (int j = 0; j < FRAME\_SIZE; j++) {

int last\_used = n;

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

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

last\_used = k;

break;

}

}

if (last\_used < lru\_time) {

lru\_time = last\_used;

lru\_index = j;

}

}

frames[lru\_index] = page;

}

printf("Page %d -> ", page);

for (int j = 0; j < FRAME\_SIZE; j++) {

printf("%d ", frames[j] == -1 ? 0 : frames[j]);

}

printf("\n");

}

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

}

int main() {

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

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

LRU(pages, n);

return 0;

}

**15.** **Simulate the following page replacement algorithms OPT**

**ALGORITHM:**OPT(pages, n):

// Initialize the frames with -1 indicating that the frame is empty

Initialize frames[FRAME\_SIZE] to -1

page\_faults = 0 // Counter for the number of page faults

// Iterate through each page in the reference string

for each page in pages:

// Check if the current page is already present in the frames

if page not in frames:

// If the page is not in the frames, check for empty space in the frames

// If there is space, place the current page in the empty frame

If space available in frames:

Place the current page in the empty space

// If no space is available, find the page that will not be used for the longest time

else:

// Find the page with the farthest next use in the future

Find the page with the farthest next occurrence in the future

// Replace that page with the current page

Replace that page with the current page

// Increment the page fault count

Increment page\_faults

// Print the current contents of the frames after each page reference

Print current frame contents

// After processing all pages, print the total number of page faults

Print total page faults

**CODE:**

#include <stdio.h>

#define FRAME\_SIZE 4

void OPT(int pages[], int n) {

int frames[FRAME\_SIZE] = {-1, -1, -1, -1};

int page\_faults = 0;

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

int page = pages[i], found = 0;

for (int j = 0; j < FRAME\_SIZE; j++) {

if (frames[j] == page) {

found = 1;

break;

}

}

if (!found) {

page\_faults++;

int empty\_index = -1;

for (int j = 0; j < FRAME\_SIZE; j++) {

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

empty\_index = j;

break;

}

}

if (empty\_index != -1) {

frames[empty\_index] = page;

} else {

int farthest\_index = -1, farthest\_time = -1;

for (int j = 0; j < FRAME\_SIZE; j++) {

int next\_use = n;

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

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

next\_use = k;

break;

}

}

if (next\_use > farthest\_time) {

farthest\_time = next\_use;

farthest\_index = j;

}

}

frames[farthest\_index] = page;

}

}

printf("Page %d -> ", page);

for (int j = 0; j < FRAME\_SIZE; j++) {

printf("%d ", frames[j] == -1 ? 0 : frames[j]);

}

printf("\n");

}

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

}

int main() {

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

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

OPT(pages, n);

return 0;

}