ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

ADVANCED FILE HANDLING OPERATIONS

CHAPTER 1: INTRODUCTION TO ADVANCED FILE HANDLING

File handling is a fundamental aspect of programming that allows applications to **store, retrieve, modify, and process data** efficiently. While basic file operations such as reading and writing to text and binary files are essential, **advanced file handling** techniques provide greater flexibility, efficiency, and control over data manipulation.

Advanced file handling includes operations such as **random access to files, file locking, buffering, handling large files, serialization, and working with different file formats**. These techniques are particularly useful in **database systems, real-time applications, logging mechanisms, and cloud-based storage systems**.

File handling in C++ is performed using the <fstream> library, which provides three primary classes:

- **ifstream** – For reading files.
- **ofstream** – For writing files.
- **fstream** – For both reading and writing files.

Advanced file handling techniques optimize file operations, ensuring **faster execution, efficient storage, and better data integrity**.

These techniques play a crucial role in **enterprise applications, operating systems, and high-performance computing**.

Chapter 2: Random File Access in C++

Understanding Random Access in Files

Unlike sequential file access, where data is read or written in order, **random access** allows a program to jump to any position in the file and read or modify data at that location. This is particularly useful in **databases, indexing systems, and multimedia applications** where efficient data retrieval is required.

Key Functions for Random File Access

C++ provides the following functions in the <fstream> library for moving within a file:

- **seekg(offset, direction)** – Moves the read pointer to a specified location.
- **seekp(offset, direction)** – Moves the write pointer to a specified location.
- **tellg()** – Returns the current read pointer position.
- **tellp()** – Returns the current write pointer position.

Example: Using Random Access to Modify a File

```
#include <iostream>

#include <fstream>

using namespace std;
```

```
int main() {
  fstream file("data.txt", ios::in | ios::out);

  if (!file) {
    cout << "File opening failed!" << endl;
    return 1;
  }

  file.seekp(5, ios::beg);  // Move write pointer to the 5th position
  file << "Hello";  // Overwrite data at this position

  file.seekg(5, ios::beg);  // Move read pointer to the 5th position
  string word;
  file >> word;
  cout << "Read from file: " << word << endl;

  file.close();
  return 0;
}
```

Output:

```
Read from file: Hello
```

Here, **random access** allows data to be written at a specific position, enabling efficient file modifications without rewriting the entire file.

CHAPTER 3: FILE LOCKING FOR DATA INTEGRITY

Why Use File Locking?

In multi-threaded or multi-user applications, simultaneous file access can cause **data corruption and inconsistency**. **File locking** prevents multiple processes from modifying a file concurrently, ensuring **data integrity** in applications like banking systems, log management, and networked applications.

Types of File Locking:

1. **Shared Lock (Read Lock)** – Multiple processes can read but not write.
2. **Exclusive Lock (Write Lock)** – Only one process can read or write.

Example: File Locking Using fcntl in Linux

```
#include <iostream>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

using namespace std;
```

```
int main() {
  int fd = open("log.txt", O_RDWR);

  if (fd == -1) {
    cout << "File opening failed!" << endl;
    return 1;
  }

  struct flock lock;
  lock.l_type = F_WRLCK;  // Write lock
  lock.l_whence = SEEK_SET;
  lock.l_start = 0;
  lock.l_len = 0;  // Lock entire file

  if (fcntl(fd, F_SETLK, &lock) == -1) {
    cout << "File is locked by another process!" << endl;
    return 1;
  }

  cout << "File locked for writing." << endl;
```

```
    sleep(10);  // Simulate long-running process
    lock.l_type = F_UNLCK;  // Unlock file
    fcntl(fd, F_SETLK, &lock);

    close(fd);
    return 0;
}
```

Here, file locking ensures that only one process can **write to the log file** at a time, preventing corruption in concurrent environments.

CHAPTER 4: HANDLING LARGE FILES EFFICIENTLY

Challenges in Handling Large Files

- **Memory Constraints:** Loading large files into memory can cause excessive RAM usage.
- **Performance Issues:** Reading or writing large files sequentially can be slow.
- **Concurrency Handling:** Multiple processes may need to access large files efficiently.

Optimizing Large File Processing

1. **Buffered Reading and Writing:** Read/write in chunks instead of loading the entire file.
2. **Compression Techniques:** Use algorithms like Gzip to store large files efficiently.

3. **Memory Mapping (mmap)**: Map files into memory for fast access.

Example: Buffered File Reading

```
#include <iostream>

#include <fstream>

using namespace std;


int main() {

  ifstream file("large_data.txt", ios::in);

  if (!file) {

    cout << "File not found!" << endl;

    return 1;

  }


  char buffer[1024];

  while (file.read(buffer, sizeof(buffer))) {

    cout.write(buffer, file.gcount());  // Process the chunk

  }


  file.close();

  return 0;
```

}

This approach reduces **memory consumption** while ensuring efficient file processing.

Chapter 5: Case Study – Log Management System

Problem Statement

A **network security system** logs **millions of entries per day**. The challenge is to **store, retrieve, and process logs efficiently** while ensuring data integrity.

Solution Approach

1. **Use Random Access:** Store logs in indexed files for fast lookup.
2. **Implement File Locking:** Prevent concurrent modifications.
3. **Use Buffered File Handling:** Process logs in chunks for efficiency.

Implementation

```
#include <iostream>

#include <fstream>

#include <ctime>

using namespace std;


void logMessage(const string& message) {

  fstream file("system_log.txt", ios::app);
  if (!file) {
    cout << "Log file cannot be opened!" << endl;
    return;
  }

  time_t now = time(0);
  file << "[" << ctime(&now) << "] " << message << endl;

  file.close();
}

int main() {
  logMessage("User login attempt detected.");
  logMessage("Security scan completed.");
  cout << "Logs recorded successfully." << endl;
  return 0;
}
```

OUTCOME

✅ **Logs are recorded efficiently without overwriting previous entries.**
✅ **Buffered writing ensures fast data entry.**
✅ **Timestamping helps in tracking security events.**

CHAPTER 6: EXERCISES

1. **Implement a file encryption system** that reads a text file, encrypts its contents, and writes it back.
2. **Modify the log management system** to store logs in different files based on the date.
3. **Implement a database system using indexed file storage** where records can be retrieved efficiently using random access.
4. **Use memory-mapped files (mmap)** to process large datasets efficiently.

CONCLUSION

Advanced file handling operations such as **random access, file locking, buffered reading, and memory mapping** enable efficient **data storage, retrieval, and processing** in real-world applications. These techniques **enhance performance, ensure data integrity, and support large-scale systems** in **database management, network security, and high-performance computing**.

EXCEPTION HANDLING & ERROR MANAGEMENT

CHAPTER 1: INTRODUCTION TO EXCEPTION HANDLING

Exception handling is a crucial concept in programming that ensures a program can deal with runtime errors gracefully without crashing. Errors can occur due to various reasons, such as invalid user inputs, file access issues, division by zero, network failures, or hardware malfunctions. Without proper exception handling, these errors can cause a program to terminate unexpectedly, leading to data loss, security vulnerabilities, and a poor user experience.

Exception handling provides a structured mechanism to detect, manage, and resolve errors dynamically during program execution. Most modern programming languages, such as Python, Java, C++, and C#, include built-in features to handle exceptions. The process typically involves enclosing the critical code in a "try" block and catching potential errors in an associated "catch" or "except" block. This allows developers to provide alternative actions, such as displaying error messages, logging errors, retrying operations, or gracefully exiting the program.

For instance, in Python, a simple exception handling implementation looks like this:

```
try:
   num1 = int(input("Enter numerator: "))
   num2 = int(input("Enter denominator: "))
   result = num1 / num2
   print("Result:", result)
```

```
except ZeroDivisionError:
   print("Error: Division by zero is not allowed.")
except ValueError:
   print("Error: Invalid input. Please enter numeric values.")
finally:
   print("Execution completed.")
```

In this example, the try block attempts to perform division, and the except blocks catch specific exceptions, preventing the program from crashing. The finally block executes regardless of whether an error occurs, ensuring that necessary cleanup operations (such as closing files or releasing resources) take place.

CHAPTER 2: TYPES OF ERRORS AND EXCEPTIONS

Errors in programming can be broadly classified into three categories:

Syntax Errors

Syntax errors occur when the code violates the grammatical rules of the programming language. These errors are detected at compile time and prevent the program from running.

Example:

```
print "Hello, World!"  # Missing parentheses in Python 3
```

This results in a syntax error because Python 3 requires parentheses for the print function.

Runtime Errors (Exceptions)

Runtime errors occur when the program encounters an unexpected issue during execution. These errors, also known as exceptions, do not appear until the program is run. Common runtime errors include division by zero, accessing an undefined variable, or opening a non-existent file.

Example:

```
num = 5 / 0  # Division by zero
```

This raises a ZeroDivisionError, which must be handled properly to prevent program failure.

Logical Errors

Logical errors occur when a program runs without syntax or runtime errors but produces incorrect results due to flawed logic. These errors are the hardest to detect because they do not halt execution.

Example:

```
def calculate_area(radius):
    return 2 * 3.14 * radius  # Incorrect formula for area (should be πr²)
```

Though the program runs smoothly, the incorrect formula leads to incorrect area calculations.

Chapter 3: Best Practices for Exception Handling

Effective exception handling is critical for writing robust, maintainable code. Following best practices can enhance program reliability and improve debugging efficiency.

Catch Only Necessary Exceptions

Instead of using a broad except statement, catch specific exceptions to avoid unintentionally suppressing errors.

Example:

```
try:
   file = open("data.txt", "r")
   content = file.read()
except FileNotFoundError:
   print("Error: The file was not found.")
```

Use Finally Blocks for Cleanup

The finally block ensures that necessary cleanup operations (such as closing database connections or files) execute regardless of whether an exception occurs.

Example:

```
try:
   file = open("data.txt", "r")
   content = file.read()
except FileNotFoundError:
   print("File not found.")
finally:
   file.close()
```

Logging Errors for Debugging

Logging error messages is crucial for diagnosing issues in production environments. Python's logging module can be used to store error details for later analysis.

Example:

```
import logging


logging.basicConfig(filename="error.log", level=logging.ERROR)


try:
    value = int("xyz")
except ValueError as e:
    logging.error(f"ValueError occurred: {e}")
```

Chapter 4: Case Study – Exception Handling in a Banking System

Scenario

A bank's online banking system allows customers to transfer funds between accounts. The system must handle exceptions such as insufficient balance, invalid account numbers, and network failures.

Solution

To ensure a smooth transaction experience, the banking application implements exception handling:

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance


    def withdraw(self, amount):
        try:
            if amount > self.balance:
                raise ValueError("Insufficient balance.")
            self.balance -= amount
            print(f"Withdrawal successful. Remaining balance: 
{self.balance}")
        except ValueError as e:
            print(f"Transaction error: {e}")


account = BankAccount(500)
account.withdraw(600)  # Raises an exception
```

This structured approach prevents unauthorized withdrawals and maintains financial integrity.

CHAPTER 5: EXERCISES

Exercise 1: Basic Exception Handling

Write a Python program that accepts two numbers from the user and divides them. Implement exception handling for ZeroDivisionError and ValueError.

Exercise 2: File Handling with Exception Management

Write a Python script to read a file’s contents. If the file does not exist, display an error message. Ensure the file is closed properly using the finally block.

Exercise 3: Custom Exception Handling

Create a Python program that raises a custom exception called NegativeValueError when a user enters a negative number for an age input.

MULTI-THREADING: THREAD CREATION & SYNCHRONIZATION

CHAPTER 1: INTRODUCTION TO MULTI-THREADING

Multi-threading is a programming technique that enables a process to execute multiple threads concurrently. A thread is the smallest unit of execution within a process, and multi-threading allows different parts of a program to run in parallel, improving performance and responsiveness. It is particularly useful in applications such as gaming, real-time systems, web servers, and data processing, where multiple tasks must be executed simultaneously without blocking the entire application.

Modern operating systems and programming languages, including Python, Java, and C++, support multi-threading, allowing developers to create efficient applications that leverage multi-core processors. While multi-threading can enhance performance, it also introduces challenges such as race conditions, deadlocks, and thread synchronization issues, which must be handled carefully to ensure program stability.

For example, a web server handling multiple client requests can use multi-threading to process each request independently without waiting for previous ones to complete. Similarly, a word processor can allow users to type text while simultaneously checking spelling in the background.

CHAPTER 2: THREAD CREATION

Creating threads in a program allows tasks to run concurrently, improving efficiency. Different programming languages provide various ways to create threads.

Thread Creation in Python

Python provides the threading module for implementing multi-threading. Threads can be created using two primary methods:

1. **Using the Thread class:**

```
import threading


def print_numbers():

   for i in range(5):

      print(f"Thread: {i}")


# Creating a thread

thread = threading.Thread(target=print_numbers)


# Starting the thread

thread.start()


# Waiting for the thread to complete

thread.join()


print("Main program completed")
```

In this example, a new thread is created to execute the print_numbers function separately from the main program.

2. **Using Thread Subclassing:**

```
import threading


class MyThread(threading.Thread):
    def run(self):
        for i in range(5):
            print(f"Thread: {i}")


# Creating and starting the thread
thread = MyThread()
thread.start()
thread.join()


print("Main program completed")
```

This method subclasses the Thread class and overrides the run() method, providing a more object-oriented approach to threading.

Thread Creation in Java

Java provides the Thread class and Runnable interface for thread creation.

1. **Extending the Thread class:**

```
class MyThread extends Thread {
    public void run() {
```

```
    for (int i = 0; i < 5; i++) {
      System.out.println("Thread: " + i);
    }
  }

  public static void main(String[] args) {
    MyThread thread = new MyThread();
    thread.start();
  }
}
```

2. **Implementing the Runnable interface:**

```
class MyRunnable implements Runnable {
  public void run() {
    for (int i = 0; i < 5; i++) {
      System.out.println("Thread: " + i);
    }
  }

  public static void main(String[] args) {
    Thread thread = new Thread(new MyRunnable());
    thread.start();
  }
}
```

Using Runnable is preferred as it allows the class to extend other classes while still using multi-threading.

CHAPTER 3: THREAD SYNCHRONIZATION

While multi-threading improves efficiency, it can lead to synchronization issues when multiple threads try to access shared resources simultaneously. Problems such as race conditions and inconsistent data states arise if proper synchronization mechanisms are not implemented.

Synchronization Using Locks

Locks ensure that only one thread can access a shared resource at a time. In Python, the threading module provides the Lock class for this purpose.

Example:

```
import threading


lock = threading.Lock()

counter = 0


def increment():

  global counter

  with lock:
        counter += 1

        print(f"Counter: {counter}")


threads = []

for _ in range(5):

    thread = threading.Thread(target=increment)

    threads.append(thread)

    thread.start()


for thread in threads:

    thread.join()
```

Here, the lock ensures that only one thread modifies the counter at a time, preventing race conditions.

Synchronization Using Synchronized Methods (Java)

Java provides the synchronized keyword to ensure thread safety.

Example:

```
class Counter {

    private int count = 0;


    public synchronized void increment() {

        count++;
        System.out.println("Counter: " + count);
    }
}

class MyThread extends Thread {
    Counter counter;

    MyThread(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        counter.increment();
    }

    public static void main(String[] args) {
        Counter counter = new Counter();
        MyThread t1 = new MyThread(counter);
        MyThread t2 = new MyThread(counter);

        t1.start();
        t2.start();
    }
}
```

The synchronized keyword ensures that only one thread can execute the increment method at a time, preventing inconsistent values.

CHAPTER 4: CASE STUDY – MULTI-THREADING IN E-COMMERCE SYSTEMS

Scenario

An e-commerce platform allows multiple users to purchase products simultaneously. When users add items to their cart, multiple threads handle product availability checks and order processing. Without proper synchronization, two users might purchase the last available item, leading to an incorrect inventory count.

Solution

Using thread synchronization techniques such as locks and semaphores ensures data integrity.

Example in Python:

```
import threading


class Inventory:
    def __init__(self, stock):
        self.stock = stock
        self.lock = threading.Lock()
    def purchase(self, quantity):
        with self.lock:
            if self.stock >= quantity:
                self.stock -= quantity
                print(f"Purchase successful. Remaining stock: {self.stock}")
            else:
                print("Not enough stock available")

inventory = Inventory(5)

threads = []
for _ in range(3):
    thread = threading.Thread(target=inventory.purchase, args=(2,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

Here, the lock prevents multiple threads from modifying the inventory simultaneously, ensuring accurate stock updates.

CHAPTER 5: EXERCISES

Exercise 1: Multi-threaded Counter

Write a Python program that creates five threads, each incrementing a shared counter variable. Ensure that race conditions do not occur.

Exercise 2: Bank Account Synchronization

Implement a bank account class where multiple threads try to withdraw money simultaneously. Use thread synchronization to ensure that the balance does not become negative.

Exercise 3: Multi-threaded File Processing

Write a Java program that reads multiple files in parallel using threads. Ensure that only one thread can write to a shared output file at a time.

MUTEX & DEADLOCKS IN MULTI-THREADED PROGRAMMING

CHAPTER 1: INTRODUCTION TO MUTEX & DEADLOCKS

Multi-threaded programming allows multiple threads to execute concurrently, improving application performance and responsiveness. However, when multiple threads access shared resources simultaneously, issues such as race conditions and inconsistent data can arise. To prevent such problems, synchronization mechanisms such as **mutex (mutual exclusion)** are used to control access to shared resources.

A **mutex (mutual exclusion)** is a synchronization primitive that allows only one thread to access a shared resource at a time. When a thread locks a mutex, other threads must wait until the mutex is released before accessing the resource. This prevents data corruption and ensures thread safety.

However, using mutexes incorrectly can lead to **deadlocks**, a situation where two or more threads are waiting indefinitely for resources locked by each other, causing the program to freeze. Deadlocks occur when multiple threads hold locks and wait for additional locks in a circular dependency.

Understanding mutexes and deadlocks is crucial for writing efficient and bug-free multi-threaded applications. Proper handling of mutexes, avoiding circular dependencies, and implementing deadlock prevention techniques can help ensure smooth execution.

CHAPTER 2: UNDERSTANDING MUTEX IN MULTI-THREADING

A **mutex (mutual exclusion)** ensures that only one thread can access a shared resource at a time. It works by using a locking mechanism:

1. A thread **locks** the mutex before accessing the resource.
2. Other threads attempting to access the same resource **wait** until the mutex is released.
3. The thread **unlocks** the mutex after finishing its operation.

Mutex Implementation in Python

Python's threading module provides a Lock class to implement a mutex.

Example:

```
import threading


mutex = threading.Lock()

shared_variable = 0


def increment():

  global shared_variable

  with mutex:

    temp = shared_variable

    temp += 1

    shared_variable = temp

    print(f"Updated value: {shared_variable}")
```

```
threads = []
for _ in range(5):
    thread = threading.Thread(target=increment)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

In this example, the mutex ensures that only one thread modifies shared_variable at a time, preventing race conditions.

Mutex Implementation in Java

Java provides the ReentrantLock class for implementing mutex-based synchronization.

Example:

```
import java.util.concurrent.locks.ReentrantLock;

class SharedResource {
    private int count = 0;
    private final ReentrantLock lock = new ReentrantLock();

    public void increment() {
        lock.lock();

        try {

            count++;

            System.out.println("Count: " + count);

        } finally {

            lock.unlock();

        }

    }

}


class MyThread extends Thread {

    SharedResource resource;


    MyThread(SharedResource resource) {

        this.resource = resource;

    }


    public void run() {

        resource.increment();

    }

}
```

```
public class MutexExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
        MyThread t1 = new MyThread(resource);
        MyThread t2 = new MyThread(resource);

        t1.start();
        t2.start();
    }
}
```

Here, the lock.lock() ensures that only one thread modifies the count variable at a time, preventing concurrent modification issues.

CHAPTER 3: UNDERSTANDING DEADLOCKS

A **deadlock** occurs when two or more threads hold locks and wait for each other's resources indefinitely, creating a circular dependency. This leads to program freezes and can severely impact performance.

Example of Deadlock in Python

```
import threading


lock1 = threading.Lock()

lock2 = threading.Lock()
```

```
def task1():
    with lock1:
        print("Task 1 acquired Lock 1")
        threading.Event().wait(1)
        with lock2:
            print("Task 1 acquired Lock 2")

def task2():
    with lock2:
        print("Task 2 acquired Lock 2")
        threading.Event().wait(1)
        with lock1:
            print("Task 2 acquired Lock 1")

thread1 = threading.Thread(target=task1)
thread2 = threading.Thread(target=task2)

thread1.start()
thread2.start()
```

```
thread1.join()

thread2.join()
```

In this example, task1 locks lock1 and waits for lock2, while task2 locks lock2 and waits for lock1. Since neither task can proceed, the program results in a deadlock.

Example of Deadlock in Java

```
class SharedResource {
   static final Object Lock1 = new Object();
   static final Object Lock2 = new Object();
}


class Thread1 extends Thread {
   public void run() {
      synchronized (SharedResource.Lock1) {
         System.out.println("Thread 1: Holding Lock 1...");
         try { Thread.sleep(100); } catch (InterruptedException e) {}
         synchronized (SharedResource.Lock2) {
            System.out.println("Thread 1: Holding Lock 2...");
         }
      }
   }
}
```

```
class Thread2 extends Thread {
    public void run() {
        synchronized (SharedResource.Lock2) {
            System.out.println("Thread 2: Holding Lock 2...");
            try { Thread.sleep(100); } catch (InterruptedException e) {}
            synchronized (SharedResource.Lock1) {
                System.out.println("Thread 2: Holding Lock 1...");
            }
        }
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();
        t1.start();
        t2.start();
    }
}
```

Here, Thread1 holds Lock1 and waits for Lock2, while Thread2 holds Lock2 and waits for Lock1, causing a deadlock.

CHAPTER 4: DEADLOCK PREVENTION & RESOLUTION

To avoid deadlocks, developers can use various strategies:

1. Avoid Nested Locks

Prevent locking multiple resources within the same thread. Instead, use a single lock at a time.

2. Lock Ordering

Ensure all threads acquire locks in a consistent order. If every thread acquires Lock1 before Lock2, deadlocks can be avoided.

3. Timeouts for Locks

Set a timeout when acquiring a lock so that a thread does not wait indefinitely.

Example in Python:

```
if lock1.acquire(timeout=1):
    if lock2.acquire(timeout=1):
        print("Locks acquired")
        lock2.release()
    lock1.release()
else:
    print("Could not acquire locks, avoiding deadlock")
```

4. Deadlock Detection

Use system monitoring tools or application-specific logic to detect deadlocks and take corrective action.

CHAPTER 5: CASE STUDY – AVOIDING DEADLOCKS IN A BANKING SYSTEM

Scenario

A banking application processes multiple transactions simultaneously. Each transaction locks a source and destination account before transferring funds. Without proper synchronization, deadlocks can occur if two transactions wait for each other's locks.

Solution

To prevent deadlocks, the system enforces a strict lock order, ensuring that all transactions lock accounts in a predetermined sequence.

Example in Java:

```
class BankAccount {
    private int balance;
    private final Object lock = new Object();

    public BankAccount(int balance) {
        this.balance = balance;
    }

    public void transfer(BankAccount target, int amount) {
        synchronized (this.lock) {
            synchronized (target.lock) {
                if (this.balance >= amount) {
                    this.balance -= amount;
                    target.balance += amount;
                    System.out.println("Transfer successful");
                } else {
                    System.out.println("Insufficient balance");
                }
            }
        }
    }
}
```

By ensuring a consistent lock order, deadlocks are avoided, and transactions complete successfully.

Chapter 6: Exercises

Exercise 1: Mutex Implementation

Write a Python program where multiple threads increment a shared counter using a mutex to prevent race conditions.

Exercise 2: Deadlock Simulation

Write a Java program that simulates a deadlock scenario and modifies it to prevent deadlocks using lock ordering.

Exercise 3: File Synchronization

Create a multi-threaded Python program where multiple threads write to a shared file. Use a mutex to ensure synchronized file access.

MULTI-THREADED TICKET BOOKING SYSTEM IN C++

ASSIGNMENT SOLUTION: DEVELOPING A MULTI-THREADED TICKET BOOKING SYSTEM IN C++

This assignment involves developing a multi-threaded ticket booking system in C++ using the **pthread** library. The application will handle multiple users trying to book tickets simultaneously while ensuring thread synchronization to avoid race conditions.

Step-by-Step Guide

STEP 1: UNDERSTANDING THE PROBLEM STATEMENT

We need to create a **multi-threaded** application that:

1. Manages a fixed number of available tickets.
2. Allows multiple users (threads) to attempt booking tickets simultaneously.
3. Ensures no overbooking occurs using thread synchronization (mutex).

STEP 2: SETTING UP THE ENVIRONMENT

To develop this application, you need:

- A C++ compiler (g++ for Linux/macOS, MinGW for Windows).
- The pthread library for multi-threading support.

To compile the program on Linux/macOS:

```
g++ ticket_booking.cpp -o ticket_booking -lpthread

./ticket_booking
```

Step 3: Implementing Multi-threading with Mutex

1. Include Required Headers

```
#include <iostream>

#include <pthread.h>

#include <unistd.h>  // For sleep()
```

2. Define Shared Resources

```
#define TOTAL_TICKETS 10  // Total available tickets

int available_tickets = TOTAL_TICKETS; // Shared variable


pthread_mutex_t ticket_mutex; // Mutex for synchronization
```

3. Create the Booking Function

Each thread simulates a user attempting to book a ticket.

```
void* book_ticket(void* user_id) {

   int user = *(int*)user_id; // Convert void* to int

   pthread_mutex_lock(&ticket_mutex);  // Lock the resource


   if (available_tickets > 0) {

      std::cout << "User " << user << " booked a ticket. Remaining: " <<
available_tickets - 1 << "\n";

      available_tickets--;  // Reduce ticket count
```

```
        sleep(1);  // Simulate processing time
    } else {
        std::cout << "User " << user << " failed to book. No tickets left.\n";
    }

    pthread_mutex_unlock(&ticket_mutex);  // Unlock the resource
    return nullptr;
}
```

4. Main Function to Create and Manage Threads

```
int main() {
    const int NUM_USERS = 15;  // Number of users attempting to book
    pthread_t users[NUM_USERS];  // Thread array
    int user_ids[NUM_USERS];  // User ID storage

    pthread_mutex_init(&ticket_mutex, nullptr);  // Initialize mutex

    // Create threads for users trying to book tickets
    for (int i = 0; i < NUM_USERS; i++) {
        user_ids[i] = i + 1;
        pthread_create(&users[i], nullptr, book_ticket, (void*)&user_ids[i]);
    }
    // Join threads to ensure execution completion
    for (int i = 0; i < NUM_USERS; i++) {
        pthread_join(users[i], nullptr);
    }

    pthread_mutex_destroy(&ticket_mutex);  // Destroy mutex

    std::cout << "Ticket booking process completed.\n";
    return 0;
}
```

STEP 4: EXPLANATION OF THE CODE

1. **Initialization**
   - We define a shared available_tickets variable to track ticket availability.
   - A **mutex (pthread_mutex_t ticket_mutex)** ensures only one thread accesses the shared variable at a time.
2. **Booking Function (book_ticket)**
   - The function locks the mutex before accessing available_tickets.
   - If tickets are available, the user books one; otherwise, a failure message is printed.
   - The mutex is unlocked after booking to allow other users to proceed.

3. **Thread Creation and Synchronization**

   - pthread_create is used to launch multiple threads simulating multiple users.
   - pthread_join ensures that the main function waits for all threads to finish execution.
   - pthread_mutex_destroy is called at the end to clean up the mutex.

Step 5: Expected Output

```
User 1 booked a ticket. Remaining: 9
User 2 booked a ticket. Remaining: 8
User 3 booked a ticket. Remaining: 7
User 4 booked a ticket. Remaining: 6
User 5 booked a ticket. Remaining: 5
User 6 booked a ticket. Remaining: 4
User 7 booked a ticket. Remaining: 3
User 8 booked a ticket. Remaining: 2
User 9 booked a ticket. Remaining: 1
User 10 booked a ticket. Remaining: 0
User 11 failed to book. No tickets left.
User 12 failed to book. No tickets left.
User 13 failed to book. No tickets left.
User 14 failed to book. No tickets left.
```

User 15 failed to book. No tickets left.

Ticket booking process completed.

STEP 6: HANDLING RACE CONDITIONS & DEADLOCKS

1. Without Mutex (Race Condition Issue)
If we remove pthread_mutex_lock and pthread_mutex_unlock, multiple threads might access available_tickets simultaneously, causing inconsistent results.

2. Avoiding Deadlocks

- Since we use a single mutex (ticket_mutex) and **unlock it immediately after use**, deadlocks do not occur.
- Deadlocks can happen when multiple mutexes are locked in an inconsistent order.

STEP 7: ENHANCEMENTS & EXTENSIONS

1. **Adding Payment Processing:**
   - Introduce a payment function that simulates payment confirmation before finalizing a booking.
2. **Database Integration:**
   - Store booking details in a database instead of printing to the console.
3. **Graphical User Interface (GUI):**
   - Create a simple GUI application using **Qt** or **GTK** to allow users to select tickets visually.

CONCLUSION

This multi-threaded ticket booking system successfully simulates concurrent ticket reservations while handling synchronization using mutexes. It prevents race conditions and ensures a thread-safe environment.

By following this guide, you have learned:
✓ How to create **threads** using pthread_create.
✓ How to **synchronize threads** using pthread_mutex_t.
✓ How to **avoid race conditions** and **prevent deadlocks**.

ASSIGNMENT SOLUTION: IMPLEMENTING A FILE-HANDLING SYSTEM FOR A STUDENT MANAGEMENT APPLICATION IN C++

Step-by-Step Guide

In this assignment, we will develop a **Student Management Application** using **file handling** in C++. The application will:

✓ Allow users to **add, view, search, update, and delete student records**.

✓ Store and retrieve student records using **file handling (text/binary file)**.

✓ Provide a **menu-driven interface** for user interaction.

STEP 1: UNDERSTANDING THE PROBLEM STATEMENT

We need to implement:

1. **File Handling** – Read and write student records to a file.
2. **Menu-based Interaction** – Provide options to **add, view, search, update, and delete student records**.
3. **Data Persistence** – Store data permanently in a file (students.txt).

STEP 2: SETTING UP THE ENVIRONMENT

You need:

- A **C++ compiler** (g++ for Linux/macOS, MinGW for Windows).

- Basic knowledge of **file handling** (fstream library) in C++.

To compile and run the program:

```
g++ student_management.cpp -o student_management
./student_management
```

STEP 3: IMPLEMENTING FILE HANDLING IN C++

1. Include Required Headers

```
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
using namespace std;
```

2. Define the Student Structure

```
struct Student {
   int id;
   string name;
   int age;
   string course;
};
```

3. Function to Add Student Records

This function writes student data to the file.

```
void addStudent() {
   Student student;
   ofstream file("students.txt", ios::app); // Open file in append mode

   cout << "Enter Student ID: ";
   cin >> student.id;
   cin.ignore(); // Clear buffer
   cout << "Enter Student Name: ";
   getline(cin, student.name);
   cout << "Enter Age: ";
   cin >> student.age;
   cin.ignore();
   cout << "Enter Course: ";
   getline(cin, student.course);

   file << student.id << "," << student.name << "," << student.age <<
"," << student.course << "\n";
   file.close();

   cout << "Student record added successfully.\n";
}
```

4. Function to View All Student Records

Reads the file and displays student records.

```
void viewStudents() {
    ifstream file("students.txt");
    string line;
    cout << "\nStudent Records:\n";
    cout << "-----------------\n";

    while (getline(file, line)) {
        stringstream ss(line);
        Student student;
        string temp;

        getline(ss, temp, ','); student.id = stoi(temp);
        getline(ss, student.name, ',');
        getline(ss, temp, ','); student.age = stoi(temp);
        getline(ss, student.course, ',');

        cout << "ID: " << student.id << ", Name: " << student.name
             << ", Age: " << student.age << ", Course: " << student.course
<< "\n";
```

```
  }
  file.close();
}
```

5. Function to Search for a Student Record

Searches for a student by ID.

```
void searchStudent() {
  ifstream file("students.txt");
  int searchID;
  bool found = false;
  cout << "Enter Student ID to search: ";
  cin >> searchID;

  string line;
  while (getline(file, line)) {
    stringstream ss(line);
    Student student;
    string temp;

    getline(ss, temp, ','); student.id = stoi(temp);
    getline(ss, student.name, ',');
    getline(ss, temp, ','); student.age = stoi(temp);
    getline(ss, student.course, ',');


    if (student.id == searchID) {
      cout << "Record Found: ID: " << student.id << ", Name: " <<
student.name
           << ", Age: " << student.age << ", Course: " << student.course
<< "\n";
      found = true;
      break;
    }
  }
  file.close();


  if (!found) {
    cout << "Student record not found.\n";
  }
}
```

6. Function to Update a Student Record

Modifies an existing record in the file.

```
void updateStudent() {
  ifstream file("students.txt");
  ofstream tempFile("temp.txt");
```

```
int searchID;
bool found = false;
cout << "Enter Student ID to update: ";
cin >> searchID;

string line;
while (getline(file, line)) {
    stringstream ss(line);
    Student student;
    string temp;

    getline(ss, temp, ','); student.id = stoi(temp);
    getline(ss, student.name, ',');
    getline(ss, temp, ','); student.age = stoi(temp);
    getline(ss, student.course, ',');

    if (student.id == searchID) {
        cout << "Enter New Name: ";
        cin.ignore();
        getline(cin, student.name);
        cout << "Enter New Age: ";
```

```
            cin >> student.age;

            cin.ignore();

            cout << "Enter New Course: ";

            getline(cin, student.course);

            found = true;

        }

        tempFile << student.id << "," << student.name << "," <<
student.age << "," << student.course << "\n";

    }

    file.close();

    tempFile.close();


    remove("students.txt");

    rename("temp.txt", "students.txt");


    if (found)

        cout << "Student record updated successfully.\n";

    else

        cout << "Student record not found.\n";

}
```

7. Function to Delete a Student Record

Removes a student from the file.

```
void deleteStudent() {
   ifstream file("students.txt");
   ofstream tempFile("temp.txt");
   int searchID;
   bool found = false;
   cout << "Enter Student ID to delete: ";
   cin >> searchID;

   string line;
   while (getline(file, line)) {
      stringstream ss(line);
      Student student;
      string temp;

      getline(ss, temp, ','); student.id = stoi(temp);
      getline(ss, student.name, ',');
      getline(ss, temp, ','); student.age = stoi(temp);
      getline(ss, student.course, ',');

      if (student.id == searchID) {
            found = true;
        } else {
            tempFile << student.id << "," << student.name << "," <<
student.age << "," << student.course << "\n";
        }
    }
    file.close();
    tempFile.close();

    remove("students.txt");
    rename("temp.txt", "students.txt");

    if (found)
        cout << "Student record deleted successfully.\n";
    else
        cout << "Student record not found.\n";
}
```

8. Creating the Main Menu

```
int main() {
    int choice;
    while (true) {
        cout << "\nStudent Management System\n";
        cout << "------------------------\n";
        cout << "1. Add Student\n2. View Students\n3. Search
Student\n4. Update Student\n5. Delete Student\n6. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1: addStudent(); break;
            case 2: viewStudents(); break;
            case 3: searchStudent(); break;
            case 4: updateStudent(); break;
            case 5: deleteStudent(); break;
            case 6: cout << "Exiting...\n"; return 0;
            default: cout << "Invalid choice, try again.\n";
        }
    }
}
```

STEP 4: EXPECTED OUTPUT

```
Student Management System
```

```
-------------------------

1. Add Student

2. View Students

3. Search Student

4. Update Student

5. Delete Student

6. Exit

Enter your choice: 1


Enter Student ID: 101

Enter Student Name: Alice

Enter Age: 20

Enter Course: Computer Science

Student record added successfully.
```

CONCLUSION

✓ Successfully implemented **file handling** in C++.
✓ Built a **student management system** with **CRUD operations**.
✓ Used **text files** for **persistent storage**.