Operating Systems - Monsoon 2024

November 6, 2024

Assignment 2 (Total points: 100)

Instructions.

1. Follow the given instructions closely, not following any single instruction will result in a **heavy deduction of marks**.
2. Code should be written in C language.
3. There should be a **Makefile** present inside the submitted zip which compiles all the source files into their executable form.
4. Files should be named **q[question number].c** unless stated otherwise. Files with any other extension like .txt will not be considered.
5. Submit a .zip named **[RollNo].zip** file containing source code files. Improper naming will be awarded 0 marks.
6. There **should not be** any subfolders inside the zip file, nor there should be any files except the source code.
7. During the Assignment demos, questions about the code and how it works will be asked. Ensure that you know the code you are submitting.
8. After the deadline, for every 6 hrs, 5% of points will be deducted.

Question 1 **[25 points]**

Write a C program to simulate **thread scheduling** that ensures **deadlock avoidance** using a simplified resource allocation system. The goal is to avoid deadlock by controlling the order in which threads acquire locks, based on a predefined global knowledge of resources.

Design:

- You should create **3 threads** (T1, T2, T3), and each thread needs to acquire **2 locks** (Lock A and Lock B) in a specific order.
- The threads should not be able to acquire locks in a fashion such that the program deadlocks.
- The program should ensure **deadlock-free execution** by scheduling the acquisition of locks based on the given deadlock avoidance algorithm.
- Implement **deadlock avoidance** using **Resource Instance Ordering** as follows:
    - Enforce a strict lock acquisition order (e.g., Lock A must always be acquired before Lock B).

1. **Input:**
    - None
2. **Output:**
    - Print messages showing the order of lock acquisition and thread waiting (e.g., "T1 acquired Lock A", "T2 waiting for Lock A").
    - The program should terminate after each thread has acquired each lock 3 times.

Question 2 **[25 points]**

Write a multithreaded C program to simulate the **Networked Servers Problem**, a synchronization problem where **servers** are trying to access **shared network resources**. The goal is to ensure **mutual exclusion** and prevent **resource starvation** using **Semaphores**.

Design:

- There are **5 servers** in a data center, each trying to perform **data processing tasks** that require access to two **network channels** (left and right).
- Each network channel is shared by 2 adjacent servers. The last network channel is shared by the **last server** and the **first server**.
- A server can only process data when it has access to both its **left** and **right** network channels.
- Each network channel can only be used by **one server at a time**.
- Each server takes **1 second** to do data processing. (You can use **sleep()** for this)
- The challenge is to design a solution where:
    1. **Mutual exclusion** is maintained (no two servers can access the same network channel simultaneously).
    2. **Starvation** is prevented (every server should get a chance to access the network channels).

Use **Semaphores** to represent the network channels and ensure proper synchronization among the servers.

1. **Input:**
    - None
2. **Output:**
    - Each server should print messages indicating when they are **waiting** (attempting to acquire network channels), and **processing** (after acquiring both channels).
    - The program should terminate after each server has processed data 3 times.

    Example:

    ```
    Server 1 is processing
    Server 2 is waiting
    Server 4 is waiting
    ```

```
Server 2 is processing
Server 1 is waiting
```

Question 3 **[25 points]**

In a warehouse management system, products are delivered and stored in a limited number of storage spaces. The warehouse has a limited capacity, and multiple delivery trucks can arrive simultaneously, each bringing a different quantity of products. To ensure efficient handling of incoming products and prevent overloading of the storage area, the system needs to manage the delivery process effectively.

Implement a C program that simulates the inventory management of a warehouse. The program should consist of two types of threads: **Delivery Trucks** and **Storage Managers**.

- **Delivery Trucks:** Each truck delivers a certain number of products to the warehouse. If the warehouse is full, the truck must wait until space is available.
- **Storage Managers:** These workers are responsible for organizing and storing the products. If there are no products to store, they must wait until trucks arrive with new products.

Design :

1. Implement a circular buffer to represent the storage area in the warehouse. The buffer should have a maximum capacity that can be defined as a constant in the code.
2. Create multiple delivery truck threads that simulate trucks arriving at the warehouse at random intervals, delivering a random number of products (between 1 and a predefined maximum).
3. Create multiple storage manager threads that simulate the storage of products at random intervals, taking a random number of products from the warehouse (between 1 and a predefined maximum) until the buffer is empty.

You can use the following synchronization primitives: semaphores/mutexes to coordinate the access to the shared storage buffer and ensure that the delivery trucks and storage managers operate correctly without causing race conditions.
The program should run for a defined period or until a certain number of deliveries have been processed, then print the final state of the warehouse.

Inputs:

- The number of products delivered must be randomly generated using rand() by Delivery Truck threads.
- The number of products stored must be randomly generated using rand() by Storage Manager threads.
- Buffer size must be a constant defined in your program.

- The number of delivery trucks and storage managers must be passed as input to the program.

Output:

- Current Inventory Status: Messages printed to the console by the delivery truck and storage manager threads, indicating the current number of products in the buffer after each delivery and storage operation.
- Buffer State Messages: Messages indicating attempts to add products to the buffer when it's full or attempts to take products when it's empty, which would help demonstrate synchronization issues and buffer management.

Question 4 **[25 points]**

This question is divided into two parts (1 and 2) :

1. Implement a program in C that performs matrix multiplication using multiple threads. The goal here is to optimize the multiplication process by dividing the work among several threads.

   Input :
   - Matrix A (size: m x n)
   - Matrix B (size: n x p)
   - Resultant matrix C (m x p)

   Design :
   - Create a suitable number of threads based on the size of the resultant matrix allowing each thread to calculate one element of Matrix C.
   - The matrices must be allocated dynamically based on the user input

   Output :
   - Resultant matrix C along with speed up over sequential version.

2. In the previous question, you created a separate thread for computing each element of Matrix C which can hamper the performance if the size of Matrix C is large. Therefore in this part of the question, you are required to implement a thread pool that will reuse threads for computing each element of Matrix C.

   Design :
   - Implement a thread pool that can manage a fixed number of threads using the pthreads API (the number of threads allocated must be equal to the number of cores in your system).

- Each thread in the pool should be capable of performing a single task to compute one element of the resultant matrix C.
- Suggestion: Instead of hardcoding the thread pool API to work with matrix multiplication, you can also design it as a generic API, that accepts a callback with some data as input and runs the callback with data.

Input: Same as part 1
Output: Resultant matrix C along with speed up over sequential version and part 1 parallel implementation.