Maryam Soleimani 401222075

Q1:

buffer_size: This variable defines the size of the shared buffer, which is 30 in this case. The buffer will be used to store items produced by the producer thread and consumed by the consumer threads.

buffer: This is a list that represents the shared buffer. Initially, it is empty.

consumer_semaphore: This is a semaphore object that is used to control access to the shared buffer by consumer threads. It is initialized with a value of 0, which means that no consumer threads can acquire the semaphore initially because the buffer is empty.

producer_semaphore: This is a semaphore object that is used to control access to the shared buffer by producer threads. It is initialized with a value equal to the buffer size (30), which means that all producer threads can acquire the semaphore initially because the buffer is empty.

items_produced: This variable is used to keep track of the number of items that have been produced by the producer thread.

The consumer thread runs in a loop forever.

Inside the loop, the consumer thread first tries to acquire the consumer_semaphore. If the semaphore cannot be acquired, it means that the buffer is empty, and the consumer thread logs a message indicating that it is waiting.

If the semaphore can be acquired, it means that there is at least one item in the buffer. The consumer thread then removes the first item from the buffer using buffer.pop(0) and stores it in the item variable. It then logs a message indicating that it has consumed an item and prints the item number.

After processing the item, the consumer thread releases the producer_semaphore. This indicates that the consumer has consumed an item and that the producer can add a new item to the buffer.

Finally, the consumer thread sleeps for a random amount of time between 0.01 and 0.01 seconds to simulate some processing time.

- The producer function defines the behavior of the producer thread (1 in this case).
- It runs in a loop until items_produced reaches 30 (total items to produce).
- The producer first tries to acquire the producer_semaphore.
    - If acquired, it means there's space in the buffer.
    - If not acquired (semaphore at 0), it logs a message indicating it's waiting.
- If the semaphore is acquired, the producer generates a random number between 0 and 1000 to represent an item.
- It logs a message indicating it has produced an item and prints the item number.
- The producer then checks if there's space in the buffer (using len(buffer)).

    - If there's space (less than buffer_size items), it adds the generated item to the buffer using buffer.append(item). It also increments items_produced.
    - If there's no space, it logs a message indicating it's waiting.
- The producer then releases the consumer_semaphore. This signals that a new item is available in the buffer for consumers to process.
- Finally, it sleeps for a random time (between 1 and 3 seconds) to simulate some production time for the item.

Main Function:

- Creates a list threads to store all the threads.
- Creates 5 consumer threads using a loop, starts each thread, and adds it to the threads list.
- Creates 1 producer thread, starts it, and adds it to the threads list.
- Waits for all threads in the threads list to finish using a loop and the join method on each thread.
- Prints a message indicating the program has terminated.

Overall Functionality:

- The program simulates producer-consumer interaction.
- The producer generates items and adds them to the buffer.
- Consumers wait for items in the buffer, consume them (remove them from the buffer), and simulate some processing time.
- Semaphores are used for synchronization to ensure consumers don't try to consume from an empty buffer and producers don't add items to a full buffer.

The first question consists of 3 parts. If we fix the sleep time of consumer too much less than the sleep time of producer it means that consumer works more than the producer in an amount of fixed time so it is more likely to empty the buffer. If the producer sleep time is too much less than consumer sleep time so producer does more work than the consumer so it is ,ore likely to full the buffer. And for the same time the sleep time for both is almost the same.

Q2:

The code is like in Q1 but we create 3 producers in a loop.

Q3:

Semaphore Creation:

- sem1 and sem2: Two threading.Semaphore objects are created.
    - Semaphores are used to control access to shared resources.

    - Each semaphore is initialized with a value of 1, meaning only one thread can acquire it at a time.

Thread Classes:

- DeadlockTest1:
    - This class inherits from threading.Thread, making it a thread class.
    - In the run() method:
        - sem1.acquire(): The thread acquires semaphore sem1.
        - time.sleep(1): The thread simulates some work by pausing for 1 second.
        - sem2.acquire(): The thread attempts to acquire semaphore sem2. This is where a potential deadlock can occur.
- DeadlockTest2:
    - This class also inherits from threading.Thread.
    - In the run() method:
        - sem2.acquire(): The thread acquires semaphore sem2.
        - time.sleep(1): The thread simulates some work.
        - sem1.acquire(): The thread attempts to acquire semaphore sem1. This is another potential deadlock point.

Thread Instantiation and Execution:

- tester1 and tester2: Instances of the DeadlockTest1 and DeadlockTest2 classes are created.
- tester1.start() and tester2.start(): The start() method of each thread object is called to begin their execution concurrently.
- tester1.join() and tester2.join(): The join() method ensures that the main thread waits for both tester1 and tester2 to complete their execution before proceeding.

Output:

- print("Threads finished execution."): This message is printed after all threads have finished executing.

Potential Deadlock:

- The key to understanding the potential deadlock lies in the order in which the threads attempt to acquire the semaphores.
    - If Thread 1 acquires sem1 and Thread 2 acquires sem2, they both wait for the other thread to release the semaphore they need.

    - This creates a circular dependency, where neither thread can proceed, leading to a deadlock.

this code demonstrates a scenario where two threads can potentially deadlock while trying to acquire resources (semaphores) in a conflicting order.

Q4:

Semaphore Creation:

- semaphore = threading.Semaphore(1): A threading.Semaphore object named semaphore is created.
    - Semaphores are used to control access to shared resources.
    - The Semaphore(1) initialization creates a binary semaphore, meaning only one thread can acquire it at a time.

Thread Functions:

- process_p1():
    - semaphore.acquire(): This line attempts to acquire the semaphore. If the semaphore is available (not held by another thread), the thread acquires it and proceeds. Otherwise, the thread waits until the semaphore is released.
    - print("Process P1 is executing..."): This line prints a message indicating that process P1 has started execution.
    - time.sleep(2): This line simulates some work being done by process P1 by pausing the thread for 2 seconds.
    - print("Process P1 has finished execution."): This line prints a message indicating that process P1 has completed execution.
    - semaphore.release(): This line releases the semaphore, allowing other threads to acquire it.
- process_p2():
    - Similar to process_p1(), this function defines the actions of process P2.
    - It acquires the semaphore, performs some work, prints a message, and releases the semaphore.

Thread Creation and Execution:

- thread1 = threading.Thread(target=process_p1): A thread object named thread1 is created, targeting the process_p1() function.

- thread2 = threading.Thread(target=process_p2): A thread object named thread2 is created, targeting the process_p2() function.
- thread1.start(): This line starts the execution of thread1 concurrently with the main thread.
- thread1.join(): This line makes the main thread wait for thread1 to complete its execution before proceeding.
- semaphore.release(): This line releases the semaphore. This is crucial because thread1 did not explicitly release the semaphore within its function.
- thread2.start(): This line starts the execution of thread2.
- thread2.join(): This line makes the main thread wait for thread2 to complete its execution.

Output:

- print("Both processes have completed execution."): This message is printed after both thread1 and thread2 have finished executing.

Key Points:

- This code demonstrates how to use a semaphore to control the execution order of two threads.
- By acquiring and releasing the semaphore, we ensure that only one thread can execute its critical section (the part of the code that accesses or modifies shared resources) at a time.
- In this case, the semaphore prevents both threads from executing concurrently, ensuring that they execute sequentially.

Here we can make sure that P1 always is executed before P2 because first P1acquire semaphore and doesn’t release it until it finishes then it releases it and P2 can acquire it.