Assignment 2

Contents

C Synchronization

Approach

Here, we implemented 4 semaphores in throughout 4 different threads. Our goal is simple: to print a sequence of natural numbers from 1 to 8 in the correct order and it shouldn't matter with which thread the computer begins.

Here is what happens when we run the code:

- Initially, all threads are blocked except for A.
- A prints numbers, then releases B and blocks itself.
- B prints numbers, then releases C and blocks itself.
- C prints numbers, then releases D and blocks itself.
- D prints numbers, then releases A and blocks itself.
- This repeats until all numbers are printed.

The outcome is that all numbers are printed concurrently one after another in the right order.

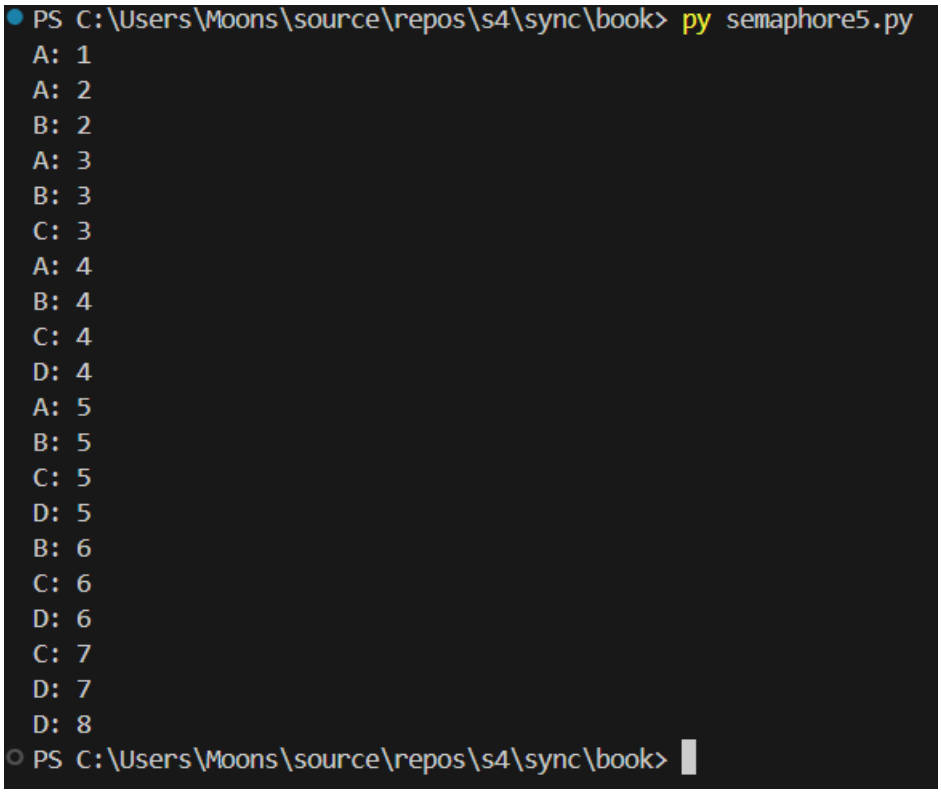
Implementation

```
import concurrent.futures
import logging
import threading
import time

import threading

# Semaphores to coordinate Thread A and Thread B
semaphore_A = threading.Semaphore(0)
semaphore_B = threading.Semaphore(0)
semaphore_C = threading.Semaphore(0)
semaphore_D = threading.Semaphore(0)

# Thread A, 1 to 5
def thread_A():
    print("A: 1")
    print("A: 2")
    semaphore_B.release()
    semaphore_A.acquire()

    print("A: 3")
    semaphore_B.release()
    semaphore_A.acquire()

    print("A: 4")
    semaphore_B.release()
    semaphore_A.acquire()
```

```
    print("A: 5")
    semaphore_B.release()

# Thread B, 2 to 6
def thread_B():
    semaphore_B.acquire()

    print("B: 2")
    semaphore_A.release()
    semaphore_B.acquire()

    print ("B: 3")
    semaphore_C.release()
    semaphore_B.acquire()

    print ("B: 4")
    semaphore_C.release()
    semaphore_B.acquire()

    print ("B: 5")
    semaphore_C.release()
    semaphore_B.acquire()

    print ("B: 6")
    semaphore_C.release()

# Thread C, 3 to 7
def thread_C():
    semaphore_C.acquire()

    print("C: 3")
    semaphore_A.release()
    semaphore_C.acquire()

    print ("C: 4")
    semaphore_D.release()
    semaphore_C.acquire()

    print ("C: 5")
    semaphore_D.release()
    semaphore_C.acquire()

    print ("C: 6")
    semaphore_D.release()
    semaphore_C.acquire()

    print ("C: 7")
```

```
    semaphore_D.release()

# # threadD: 4 to 8.
def thread_D():
    semaphore_D.acquire()
    print("D: 4")
    semaphore_A.release()
    semaphore_D.acquire()

    print("D: 5")
    semaphore_B.release()
    semaphore_D.acquire()

    print("D: 6")
    semaphore_C.release()
    semaphore_D.acquire()

    print("D: 7")
    print("D: 8")


# Create threads
thread1 = threading.Thread(target=thread_A)
thread2 = threading.Thread(target=thread_B)
thread3 = threading.Thread(target=thread_C)
thread4 = threading.Thread(target=thread_D)

# Start both threads
thread1.start()
thread2.start()
thread3.start()
thread4.start()

# Wait for both threads to finish
thread1.join()
thread2.join()
thread3.join()
thread4.join()
```

Outcome


PS C:\Users\Moons\source\repos\s4\sync\book> py semaphore5.py
A: 1
A: 2
B: 2
A: 3
B: 3
C: 3
A: 4
B: 4
C: 4
D: 4
A: 5
B: 5
C: 5
D: 5
B: 6
C: 6
D: 6
C: 7
D: 7
D: 8
PS C:\Users\Moons\source\repos\s4\sync\book>

D Deadlock

Approach

In the implemented system, we use three semaphores **(A, B, C)**

and three threads **(Thread A, Thread B, Thread C)**

Each thread attempts to enter a critical section by acquiring all three semaphores, but in a unique order. The design intentionally introduces a deadlock risk, still under certain conditions, allowing the threads to run for hours without a deadlock.

This pattern can lead to a situation where each thread holds one semaphore and waits for another, forming a cyclic dependency. For example, Thread A could hold Semaphore C and wait for B, which is held by Thread C, which in turn waits for A, held by Thread B, creating a deadlock loop.

Despite the setup's inherent risk of deadlock, the system might operate flawlessly for hours due to non-deterministic scheduling and timing variations.

Implementation

```
A = Semaphore(1)
B = semaphore(1)
C = Semaphore(1)

#Thread A
c.wait()
b.wait()
a.wait()

#critical section

a.signal()
c.signal()
b.signal()


#Thread B
a.wait()
c.wait()
b.wait()

#critical section

b.signal()
a.signal()
c.signal()


#Thread C
b.wait()
a.wait()
c.wait()

#critical section

c.signal()
b.signal()
a.signal()
```

Deadlock Risk

The potential for deadlock arises from the order in which semaphores are acquired:

Action	*Thread A*	*Thread B*	*Thread C*
A4	Crit Section		
A5	Releasse A		
B1		Holds A	
A6	Release C		
B2		Holds C	
A6	Releasse B		
C1			Holds B

So at the end Thread C takes Semaphore B first then Thread B so now it is common circular wait scenario where both threads wait each other to release Semaphore after their critical sections. But will never get there becouse stuck on the barier.

Additional Problems

6.2.1 E 3.7 Reusable Barrier (I)

Implement the Reusable Barrier of paragraph 3.7, but only with the use of semaphores (so no counters). The number of threads is known at compile time, e.g. 4.

```
n = 4
barier1_check = Semaphore(n)
barier2_check = Semaphore(n)

barrier1 = Semaphore(0)
barrier2 = Semaphore(1)

## Thread A
barier1_check.wait()
if barier1_check.n == 0:
    barrier2.wait()
    barrier1.signal()
barrier1.wait()
barrier1.signal()
barier1_check.signal()
# critical section
barier2_check.wait()
if barier2_check.n == 0:
    barrier1.wait()
    barrier2.signal()

barrier2.wait()
barrier2.signal()
barier2_check.signal()
```

In the designed symetrical system, I utilize two sets of semaphores—**barrier1_check**, **barrier2_check**, **barrier1**, **barrier2** to implement a reusable barrier synchronization mechanism for ***n*** threads. This setup is intended to manage thread synchronization without the use of counters, using only semaphores.

Semaphores and Their Purposes:

- **barrier1_check and barrier2_check:** These are initialized to the number of threads (n = 4) and are used to track whether all threads have reached the barrier before any can proceed.
- **barrier1 and barrier2:** These control the passage of threads through each phase of the barrier. **barrier1** is initialized to 0 to block threads initially, and **barrier2** is initialized to 1 to allow passage after the first barrier is lifted.

Thread Operation Details:

Each thread follows this procedure to synchronize at two points (two barriers), ensuring all threads reach the same execution point before any can proceed:

1. **Entering Barrier 1:**
    - Acquire **mutex** to ensure exclusive access.

- Perform a wait operation on **barrier1_check**. This semaphore decrementing to 0 indicates all threads have reached this point.
- If the last thread arrives (**barrier1_check.n == 0**), it allows all threads to pass the first barrier by signaling **barrier1** after ensuring **barrier2** is locked.
- Wait on **barrier1** to ensure all threads synchronize at this barrier, then signal **barrier1** to allow following threads to proceed.
- Signal **barrier1_check** to reset for potential reuse.

2. **Critical Section:**
   - Threads perform required operations that must be synchronized.
3. **Entering Barrier 2:**
   - Similar to Barrier 1, perform a wait on **barrier2_check**.
   - If the last thread arrives at Barrier 2 (**barrier2_check.n == 0**), it switches the roles of the barriers: waiting for all threads to complete interactions with **barrier1** and then signaling **barrier2** to reset the state for future reuse.
   - Wait on **barrier2**, and signal it to maintain synchronization, followed by signaling **barier2_check** for reset.

6.2.2 F 3.7 Reusable Barrier (II)

Re-implement the solution of the Reusable Barrier of paragraph 3.7, but don't use turnstile.wait() for locking (aka closing) a turnstile (see the rectangles in the following picture

```
n = 4
count = 0

barrier1 = Semaphore(0)
barrier2 = Semaphore(0)
mutex = Semaphore(1)

## Thread A
mutex.wait()
count += 1
if count == n:
    barrier1.signal()
    mutex.signal()
    barrier1.wait()
else:
    mutex.signal()
    barrier1.wait()
    barrier1.signal()

# critical section
mutex.wait()
count -= 1
if count == 0:
    barrier2.signal()
    mutex.signal()
    barrier2.wait()
else:
    mutex.signal()
    barrier2.wait()
    barrier2.signal()
```

In this revised implementation of a reusable barrier using semaphores, the setup includes three semaphores (**barrier1**, **barrier2**, and **mutex**) and a shared counter (**count**). This mechanism is designed to synchronize four threads (as indicated by **n = 4**). The threads use this barrier to ensure that all threads have reached a certain point in their execution before any of them can proceed, which is crucial for maintaining correct program flow when multiple threads need to operate in lockstep.

Semaphores and Their Purposes:

- **barrier1 and barrier2:** These semaphores are used to control the flow of threads through the barrier. Initially, both are set to 0, effectively blocking any thread trying to pass through until they are explicitly signaled.
- **mutex:** A binary semaphore used to protect the shared **count** variable, ensuring that only one thread can modify it at a time to prevent race conditions.

Operation Details:

Each thread follows a specific sequence to ensure synchronized progression at two distinct points in their execution:

1. **Before the Critical Section:**
    - **Mutex Acquisition:** Each thread starts by acquiring the **mutex** to gain exclusive access to the shared **count**.
    - **Count Increment:** The thread increments the **count** to indicate that it has reached the barrier.
    - **Barrier Check:** If the incremented **count** equals **n** (the total number of threads), this indicates that all threads have reached this point. The thread then signals **barrier1** to allow all waiting threads to proceed and releases the **mutex**.
    - **Waiting and Signaling:** If the **count** is not yet **n**, the thread releases the **mutex** and waits on **barrier1**. Once it is signaled (either by itself or another thread), it signals **barrier1** again to ensure that the next thread can also proceed.
2. **Critical Section:**
    - **Perform Critical Operations:** This is where threads perform their synchronized tasks.
3. **After the Critical Section:**
    - **Mutex Acquisition Again:** Similar to the pre-critical section, the thread again acquires the **mutex** to safely decrement the **count**.
    - **Count Decrement:** The thread decrements the **count** to indicate it has passed the critical section.
    - **Final Barrier Check:** If the decremented **count** reaches zero, indicating all threads have completed their critical section, it signals **barrier2**. This releases the first waiting thread, which then waits and signals **barrier2** to pass the control back to subsequent threads.
    - **Waiting and Signaling for Completion:** If the **count** is not zero, the thread releases the **mutex** and waits on **barrier2**, signaling it after passing through to allow the next thread to also proceed.

G 3.8 Queue: followers & leaders

Make a symmetric implementation of the 3.8 problem with a pipet; without counters.

Ensure that an arbitrary number of follower and leader threads can be started (e.g. N=5)

```
leader = Semaphore(1)
leader_Queue = Semaphore(1)
follower = Semaphore(1)
follower_Queue = Semaphore(1)

## Thread L1
leader_Queue.wait()
follower.signal()
leader.wait()
# dance
leader_Queue.signal()

##Thread F1
follower_Queue.wait()
leader.signal()
follower.wait()
# dance
follower_Queue.signal()
```

To implement this solutionI use a symmetric solution with semaphores. Each type of thread—leader and follower—runs the same code within their respective groups, ensuring that they can only proceed to the dance task when paired correctly with a thread from the other group. This approach leverages four semaphores to control entry into the critical section where the dance occurs and to ensure that threads pair up properly.

Semaphores and Their Purposes:

- **leader and follower:** These semaphores are used to signal the availability of a leader or a follower to the opposite group. Initially, both are set to 1 to indicate that one leader and one follower can enter the pairing area.
- **leader_Queue and follower_Queue:** These semaphores control access to the critical section where the dance occurs, ensuring that no more than one thread of each type is attempting to pair at any one time.

Operation Details:

Each thread follows a simple sequence designed to ensure that each leader pairs with a follower:

Leader Thread (e.g., Thread L1):

1. **Queue Entry:** Waits for access to the leader queue (**leader_Queue.wait()**) to ensure that it is the only leader attempting to pair.
2. **Signal Availability:** Signals a follower (**follower.signal()**) to indicate a leader is ready to pair.
3. **Wait for Pair:** Waits for a follower to be ready (**leader.wait()**) before proceeding.

4. **Perform Task (Dance):** After pairing, the leader can proceed to the critical section to perform the dance.
5. **Exit Queue:** Signals the next leader that the queue is free (**leader_Queue.signal()**).

Follower Thread (e.g., Thread F1):

1. **Queue Entry:** Waits for access to the follower queue (**follower_Queue.wait()**) to ensure that it is the only follower attempting to pair.
2. **Signal Availability:** Signals a leader (**leader.signal()**) to indicate a follower is ready to pair.
3. **Wait for Pair:** Waits for a leader to be ready (**follower.wait()**) before proceeding.
4. **Perform Task (Dance):** After pairing, the follower can proceed to the critical section to perform the dance.
5. **Exit Queue:** Signals the next follower that the queue is free (**follower_Queue.signal()**).