Computing platforms (Spring 2025) week 5

Exercise 1

Synchronous and asynchronous communication. In synchronous communication both communicating parties are expected to be present at the same time, while in asynchronous communication they can run separate times.

(a) What if your application logic needs synchronous communication, but the environment only provide asynchronous communication? How can you use asynchronous communication to implement synchronous communication?

- **Sending a Request:** The sender dispatches a message asynchronously to the receiver.
- **Attaching an Identifier:** Include a unique identifier with the message so that the corresponding response can be recognized.
- **Blocking for the Response:** The sender then waits (i.e., blocks) until it receives a reply matching the identifier. This wait can be implemented using mechanisms such as callbacks or condition variables.
- **Resuming Execution:** Once the response is received, the sender unblocks and continues execution.

This approach creates request-response (synchronous) pattern over asynchronous channel, ensuring the sender does not proceed until the required information is available.

(b) Think about a reverse situation. You need asynchronous communication, but only synchronous communication is available? How can you implement asynchronous communication using synchronous communication?

- **Using Concurrency:** Spawn separate thread or process to handle the synchronous communication. Main thread can continue execution without waiting.
- **Buffering Messages:** Create local message queue or buffer where outgoing messages are stored. Dedicated worker thread performs synchronous send/receive operations behalf of main application.
- **Event-Driven Notification:** On completing synchronous operation, worker thread notify main thread (e.g., via event or callback mechanism) that communication is complete, thus decoupling the communication from main control flow.

By offloading the blocking operations to separate threads and using buffering, overall system behaves in an asynchronous manner despite underlying synchronous primitives.

Exercise 2

You have an existing program that uses pipes (named or unnamed) as communication mechanism between two processes. Now you need to move one of the processes to another computer and you need to change the communication mechanism, because pipes can be used only within one computer. Which of the mechanisms (shared memory, signal, file, socket, message queue, remote procedure call) would you be able to use and why? Please also give reasons for the rest, why they would not be suitable in this case.

Suitable mechanisms

- **Socket:** Sockets are designed for network communication. They allow data to be transmitted between processes on different computers reliably and efficiently.
- **Message Queue:** Distributed message queue systems enable asynchronous communication across machines. They provide buffering, reliability, and decoupling of processes.
- **Remote Procedure Call (RPC):** RPC abstracts network communication details and allows a process to invoke functions on a remote machine. It encapsulates the network protocols and provides a clean interface for inter-machine communication.

Not suitable mechanisms:

- **Shared Memory:** Shared memory is based on idea of multiple processes accessing common memory space, which is only feasible on the same physical computer.
- **Signal:** Signals are primarily used for simple notifications or process control within a single machine. They do not carry amount of data or offer reliability needed for communication across different machines.
- **File:** While files can be used for communication via a shared network file system, this approach is prone to synchronization issues. File-based communication is less suitable for real-time or interactive inter-process communication.

Exercise 3

Race condition. Alice and Bob come to run 4000 meters on a running track that is 400 meters long. The python code “race-condition.py” shows how they are counting the laps. Download the code separately from Moodle.

(a) Run the code several times on your computer. What do you observe?

The Python program creates two threads (Alice and Bob) both running 10 iterations. In each iteration the thread:

- Read shared variable **laps**.
- Compute **mylap = laps + 1**.
- Assign **laps = mylap**.

Because there is no synchronization, two threads can interleave their operations in a way that leads to **lost updates**.

(b) Determine the proper lower and upper bound on the final value of the shared variable laps when this concurrent program terminates. Assume threads can execute at any relative speed.

When you run the code several times, you may observe that:

- The printed lap numbers may not increase sequentially as expected.
- The final value of **laps** is not always 20 (i.e. 10 iterations per thread). It often appears to be lower.

This is a manifestation of a *race condition* where both threads read and update the shared variable without coordination, causing some increments to be “lost.”

Upper Bound: The best-case scenario occurs when the threads never interfere (i.e. their operations happen sequentially without overlap). In that case every increment is preserved and

$$\text{Final value} = 10 + 10 = 20.$$

Lower Bound: A worst-case scenario is possible if a thread, due to scheduling delays, reads *stale* value before any updates occur and then later writes that stale increment, thereby *overwriting* progress made by other thread.

- Suppose Alice starts first iteration and reads **laps** when it is **0** but gets delayed before writing.
- Meanwhile Bob performs all of his 10 iterations correctly, updating **laps** from 0 to 10.
- Then Alice resumes and, still holding stale value from her first iteration, writes **laps = 0 + 1 = 1**.

Even if only one such *rollback* occurs, it can drastically reduce the final count. In the extreme, similar delays in several iterations could cause many increments to be lost. By careful interleaving it is possible for final value to be as low as **1** (i.e. only one effective update is preserved).

Conclusion:

$$1 \leq \texttt{laps} \leq 20.$$

One might expect ”pairing” of increments to yield lower bound of 10, but possibility of *stale reads* and *rollbacks* means that in worst-case final value can be as low as 1.

Note: Printed output is not necessarily precise reflection of actual execution order, since print statements are unsynchronized and may interleave arbitrarily with thread operations.

Exercise 4

Locks and condition variables. In this exercise we will continue with the python code of Alice and Bob counting the laps they are running. Here we will modify it so that Alice and Bob synchronize with each other properly.

(a) Modify the python code so that the critical sections are protected properly by using locks. Make your solution as simple as possible but still correct in all possible scenarios.

```
from threading import Thread, current_thread, Lock

# Alice and Bob will both run n laps
n=10
# Total number of laps
laps=0
lock = Lock()

# Function executed by both Alice and Bob threads
def runlaps():
    global laps
    name = current_thread().name # Is this Alice or Bob?
    for _ in range(n):
        with lock:
            # Which lap is starting?
            mylap = laps+1
            print(name + "␣starting␣lap:␣" + str(mylap))
            # Store the lap number that was finished
            laps = mylap
            print(name + "␣finished␣lap:␣" + str(laps))

# Create the threads for Alice and Bob
alice_thread = Thread(target=runlaps, name="Alice")
bob_thread = Thread(target=runlaps, name="Bob")

# Start the threads
alice_thread.start()
bob_thread.start()

# Wait for Alice and Bob threads to finish
alice_thread.join()
bob_thread.join()

# Print the total number of laps
print("Total␣laps:␣" + str(laps))
```

(b) After each lap, Alice waits for Bob to catch up with her (on the same lap, so that both have run equal number of laps). If Bob happens to be running ahead, he will keep on running and lets Alice try to catch him. So Bob just runs for his 10 laps. On a good day Bob could pass Alice many times and finish his run when Alice still has many laps to go. Use condition variables to have Alice wait for Bob after each lap.

To implement this we need separate counters for Alice and Bob. We also use a `Condition` variable, which is associated with a lock, to allow Alice to wait until Bob's counter catches up.

```
from threading import Thread, current_thread, Condition

# Alice and Bob will both run n laps
n=10
# Separate lap counters for Alice and Bob
alice_laps = 0
bob_laps = 0

# Condition variable with its internal lock
cond = Condition()

# Function executed by Alice
def runlaps_alice():
    global alice_laps, bob_laps
    for _ in range(n):
        with cond:
            # Which lap is starting?
            alice_laps += 1
            print("Alice␣finished␣lap:␣" + str(alice_laps))
            cond.notify_all()
            while bob_laps < alice_laps:
                cond.wait()

# Function executed by Bob
def runlaps_bob():
    global bob_laps
    for _ in range(n):
        with cond:
            bob_laps += 1
            print("Bob␣finished␣lap:␣" + str(bob_laps))
            cond.notify_all()

# Create the threads for Alice and Bob
alice_thread = Thread(target=runlaps_alice, name="Alice")
bob_thread = Thread(target=runlaps_bob, name="Bob")

# Start the threads
alice_thread.start()
bob_thread.start()

# Wait for Alice and Bob threads to finish
alice_thread.join()
bob_thread.join()

# Print the total number of laps
print("Final␣laps:␣Alice␣=␣" + str(alice_laps) + ",␣Bob␣=␣" + str(bob_laps))
```

Exercise 5

One-lane bridge with locks and condition variables. There is a river between two villages A and B and the villages are connected by a narrow one-lane bridge, where cars are allowed to drive only in one direction in a time. There can be many cars on the bridge at a time, but only in one direction. When cars are passing the bridge from A to B (westbound), then the cars willing to cross the bridge from B to A (eastbound) have to wait. The python code “one-lane-bridge.py” contains skeleton code for the one-lane bridge. The car processes call bridge.enter_<direction> before using the bridge, and bridge.exit_<direction>, when they leave the bridge. A car process using the bridge executes the drive_<direction> functions. Complete the python code by defining the bridge.enter_west, bridge.exit_west, bridge.enter_east, and bridge.exit_east functions. Remember to define all your locks, condition variables and counters. The solution does not need to be fair, which means that the waiting times on the other end may be very long.

```
from threading import Thread, current_thread, Lock, Condition

# An object representing a one-lane bridge
class Bridge():
    def __init__(self):
        # Keep track of cars currently on the bridge
        # Add here the variables, locks and condition variables needed for synchronization
        self.cars = set()
        self.lock = Lock()
        self.cond = Condition(self.lock)
        self.west_count = 0
        self.east_count = 0

    # A car wishing to cross the bridge westbound will call this when entering the bridge
    def enter_west(self, name):
        # Add code here to synchronize the cars on the bridge properly
        with self.cond:
            while self.east_count > 0:
                self.cond.wait()
            self.west_count += 1
            self.cars.add(name)
            print("Cars␣on␣the␣bridge:␣" + str(self.cars))
        return

    # A car wishing to cross the bridge westbound will call this when exiting the bridge
    def exit_west(self, name):
        # Add code here to synchronize the cars on the bridge properly
        with self.cond:
            self.west_count -= 1
            self.cars.remove(name)
            print("Cars␣on␣the␣bridge:␣" + str(self.cars))
            if self.west_count == 0:
                self.cond.notify_all()
        return

    # A car wishing to cross the bridge eastbound will call this when entering the bridge
    def enter_east(self, name):
        # Add code here to synchronize the cars on the bridge properly
        with self.cond:
            while self.west_count > 0:
                self.cond.wait()
            self.east_count += 1
            self.cars.add(name)
            print("Cars␣on␣the␣bridge:␣" + str(self.cars))
        return

    # A car wishing to cross the bridge eastbound will call this when exiting the bridge
    def exit_east(self, name):
        # Add code here to synchronize the cars on the bridge properly
        with self.cond:
            self.east_count -= 1
            self.cars.remove(name)
            print("Cars␣on␣the␣bridge:␣" + str(self.cars))
            if self.east_count == 0:
                self.cond.notify_all()
        return


# A car driving westbound over the bridge
def drive_westbound(bridge):
    name = current_thread().name
    print(name + ":␣Driving␣westbound...")
    bridge.enter_west(name)
    print(name + ":␣Westbound␣on␣the␣bridge")
    bridge.exit_west(name)
    print(name + ":␣Crossed␣the␣bridge␣westbound")

# A car driving eastbound over the bridge
def drive_eastbound(bridge):
    name = current_thread().name
    print(name + ":␣Driving␣eastbound...")
    bridge.enter_east(name)
    print(name + ":␣Eastbound␣on␣the␣bridge")
    bridge.exit_east(name)
    print(name + ":␣Crossed␣the␣bridge␣eastbound")

# Initialize the bridge and the car threads
bridge = Bridge()
Twest = [Thread(target=drive_westbound, kwargs={"bridge": bridge}, name="W" + str(i)) for i in
    range(10)]
Teast = [Thread(target=drive_eastbound, kwargs={"bridge": bridge}, name="E" + str(i)) for i in
    range(10)]

# Start the car threads
for i in range(10):
    Twest[i].start()
    Teast[i].start()
```

This solution is not fair. If cars continuously enter one direction, waiting cars on other side can be left waiting indefinitely.