<u>Condition Variables, Semaphores, Concurrency Bugs</u>

Questions:

+

Q1. A shared buffer of maximum capacity N = 5 is used by two types of threads:

- **Producers: generate integer data items and add them to the buffer.**
- **Consumers: remove integer data items from the buffer for processing.**

The system uses a single mutex lock (mutex) and two condition variables:

- **not_full - signals when there's at least one empty slot in the buffer.**
- **not_empty - signals when there's at least one filled slot in the buffer.**

Each thread follows the following pseudocode:

Producer Thread

```
while (true) {
   item = produce_item();

   pthread_mutex_lock(&mutex);
   while (count == N)
      pthread_cond_wait(&not_full, &mutex);

   buffer[in] = item;
   in = (in + 1) % N;
   count++;

   pthread_cond_signal(&not_empty);
   pthread_mutex_unlock(&mutex);
}
```

Consumer Thread

```
while (true) {
   pthread_mutex_lock(&mutex);
   while (count == 0)
      pthread_cond_wait(&not_empty, &mutex);

   item = buffer[out];
   out = (out + 1) % N;
   count--;

   pthread_cond_signal(&not_full);
   pthread_mutex_unlock(&mutex);

   consume_item(item);
}
```

Assume the following timeline of events, where P1, P2, C1, and C2 are threads:

Time (ms)	**Event Description**	**Buffer Count (count)**	**Condition Variable Actions**
0	**P1 starts producing**	**0**	**-**
10	**P1 tries to add item A**	**?**	**-**
20	**P2 tries to add item B**	**?**	**-**
30	**C1 starts consuming**	**?**	**-**
40	**C2 tries to consume**	**?**	**-**
50	**P1 adds item C**	**?**	**-**
60	**P2 adds item D**	**?**	**-**
70	**C1 consumes**	**?**	**-**
80	**C2 consumes**	**?**	**-**

At the start, the buffer is empty (count = 0), and both in and out are 0.

1. **Step Simulation (Numerical):**
   For each event (every 10 ms), determine:
   - **Whether the thread waits or proceeds.**
   - **The updated buffer count (count).**
   - **Which condition variable (if any) is signaled.**
2. **Fill in a table showing how count, not_full, and not_empty evolve.**
3. **Explain why condition variables are needed, and what would happen if they were replaced by busy waiting.**

Ans. Initial State

Variable	**Value**
count	0
in	0
out	0
not_full	no waiting threads

not_empty	no waiting threads

Step-by-Step Execution

Time = 0 ms

- P1 starts producing an item (A).
- It's still in produce_item() phase - no effect on shared state.

-> count = 0

Time = 10 ms

- P1 tries to add item A.
- It locks mutex. Since count < N (0 < 5), it proceeds.
- It adds A to buffer, increments count to 1, signals not_empty.

Result:

- count = 1
- Signal: pthread_cond_signal(&not_empty)any waiting consumer wakes.

Time = 20 ms

- P2 tries to add item B.
- Locks mutex. count = 1 < 5, proceeds.
- Adds B, count = 2.
- Signals not_empty.

Result:

- count = 2
- Signal: not_empty

Time = 30 ms

- C1 starts consuming.
- Locks mutex. Since count = 2 > 0, proceeds.
- Removes item A, count = 1.
- Signals not_full.

Result:

- count = 1
- Signal: not_full

Time = 40 ms

- C2 tries to consume.
- Locks mutex. Since count = 1 > 0, proceeds.
- Removes item B, count = 0.
- Signals not_full.

Result:

- count = 0
- Signal: not_full

Time = 50 ms

- P1 adds item C.
- Locks mutex. Since count = 0 < 5, proceeds.
- Adds C, count = 1.
- Signals not_empty.

Result:

- count = 1
- Signal: not_empty

Time = 60 ms

- P2 adds item D.
- Locks mutex. Since count = 1 < 5, proceeds.
- Adds D, count = 2.
- Signals not_empty.

Result:

- count = 2
- Signal: not_empty

Time = 70 ms

- C1 consumes again.
- Locks mutex. count = 2 > 0, proceeds.
- Removes C, count = 1.
- Signals not_full.

Result:

- count = 1
- Signal: not_full

Time = 80 ms

- C2 consumes again.
- Locks mutex. count = 1 > 0, proceeds.
- Removes D, count = 0.
- Signals not_full.

Result:

- count = 0
- Signal: not_full

Time (ms)	Action	Wait/Proceed	count After	Condition Signal
10	P1 adds A	Proceed	1	not_empty
20	P2 adds B	Proceed	2	not_empty
30	C1 consumes A	Proceed	1	not_full
40	C2 consumes B	Proceed	0	not_full

50	P1 adds C	Proceed	1	not_empty
60	P2 adds D	Proceed	2	not_empty
70	C1 consumes C	Proceed	1	not_full
80	C2 consumes D	Proceed	0	not_full

- Condition Variables (pthread_cond_wait, pthread_cond_signal) allow threads to block (sleep) efficiently until a condition becomes true.
- Without them, threads would use busy waiting:
  - Continuously polling (while(count == N);) would waste CPU cycles.
  - Performance drops drastically when multiple threads are competing.

Thus, condition variables provide a mechanism for efficient synchronization and CPU resource sharing.

Q2. At a small airport, there is one runway shared by planes that land and planes that take off.
To ensure safety:

1. **Only one plane may use the runway at any time.**
2. **Landing planes have priority over takeoff planes (since they have limited fuel).**
3. **If no landing planes are waiting, one takeoff plane may use the runway.**
4. **The system must prevent starvation - takeoff planes must eventually get to use the runway if landings stop arriving for a while.**

You are asked to design a synchronization scheme using semaphores to coordinate the runway usage between multiple landing and takeoff planes.
Given: Each plane is represented by a thread executing one of the following procedures:

```
void landing_plane(int id) {
   arrive_to_land(id);        // just prints arrival
   request_to_land(id);        // tries to acquire runway
   use_runway(id, "landing");   // safely land
   leave_runway(id);           // release runway
}

void takeoff_plane(int id) {
   arrive_to_takeoff(id);
   request_to_takeoff(id);
   use_runway(id, "takeoff");
   leave_runway(id);
}
```

Your task is to implement the synchronization logic for:

- **request_to_land()**
- **request_to_takeoff()**
- **leave_runway()**

using semaphores.
You may use the following semaphores and counters:
Semaphore runway = 1; // controls access to the runway
Semaphore mutex = 1; // protects shared counters
Semaphore landing_queue = 0;// planes waiting to land
Semaphore takeoff_queue = 0;// planes waiting to take off

int waiting_to_land = 0;
int waiting_to_takeoff = 0;

1. **Design the semaphore-based solution ensuring:**
   - **At most one plane is on the runway at a time.**
   - **Landing planes have priority.**
   - **Takeoff planes are not starved.**
2. **Provide pseudocode for each of the three critical functions:**
   - **request_to_land()**
   - **request_to_takeoff()**
   - **leave_runway()**

Explain step-by-step how your solution handles the following sequence of events:
L1 (landing), T1 (takeoff), L2 (landing), T2 (takeoff)

3. **arriving in this order at short intervals.**
4. **Discuss what could go wrong if semaphores were misused (e.g., forgetting wait(mutex) before updating counters).**

Ans. We'll use counting semaphores and counters to manage which type of plane can proceed.
Rules encoded:

- A plane can only wait(runway) when it is explicitly signaled that it may proceed.
- Landing planes increment waiting_to_land and block if the runway is in use.
- Takeoff planes increment waiting_to_takeoff but must also wait if there are any landing planes waiting or using the runway.

Pseudocode
Landing Plane

```
void request_to_land(int id) {
   wait(mutex);
   waiting_to_land++;
   wait(mutex);

   // Priority to landing planes
   if (runway == 0) {            // someone using runway
      wait(landing_queue);       // wait until signaled
   }

   wait(runway);                 // acquire runway
   wait(mutex);
```

```
    waiting_to_land--;
    signal(mutex);
}
```

Takeoff Plane

```
void request_to_takeoff(int id) {
    wait(mutex);
    waiting_to_takeoff++;

    // If any landing planes are waiting, must yield
    if (waiting_to_land > 0 || runway == 0) {
        signal(mutex);
        wait(takeoff_queue);       // wait until allowed
    } else {
        signal(mutex);
        wait(runway);              // acquire runway
    }

    wait(mutex);
    waiting_to_takeoff--;
    signal(mutex);
}
```

Leaving the Runway

```
void leave_runway(int id) {
    wait(mutex);

    if (waiting_to_land > 0) {
        signal(landing_queue);     // give priority to next landing
    } else if (waiting_to_takeoff > 0) {
        signal(takeoff_queue);     // allow one takeoff plane
    } else {
  ●         signal(runway);            // free the runway
    }

    signal(mutex);
}
```

Sequence Simulation

Let’s walk through the given sequence: L1, T1, L2, T2

Initial state: runway = 1, waiting_to_land = 0, waiting_to_takeoff = 0.

Time	Event	Action	Waiting Queues	Runway Holder
1	L1 arrives	wait(runway)succeeds	-	L1
2	T1 arrives	sees waiting_to_land > 0waits	takeoff_queue = [T1]	L1
3	L2 arrives	increments waiting_to_land, waits	landing_queue = [L2]	L1
4	L1 leaves	waiting_to_land > 0signals landing_queue	landing_queue = [ ]	L2
5	L2 uses runway	proceeds	-	L2
6	L2 leaves	waiting_to_land == 0 but waiting_to_takeoff > 0signals takeoff_queue	-	T1
7	T1 uses runway	proceeds	-	T1
8	T1 leaves	waiting_to_land == 0, waiting_to_takeoff == 1signals takeoff_queue	-	T2
9	T2 uses runway	proceeds	-	T2

Correct behavior:

- Only one plane uses runway at a time.
- Landing planes get priority.
- Takeoff planes eventually proceed (no starvation).

Q3. Consider the following pseudo-C program using two threads, T1 and T2. Assume that both threads share a global variable x.

```
int x = 0;

void* thread1(void* arg) {
    x = x + 1;
    printf("T1: x = %d\n", x);
    return NULL;
```

```
}

void* thread2(void* arg) {
    x = x + 2;
    printf("T2: x = %d\n", x);
    return NULL;
}

int main() {
    create_thread(thread1);
    create_thread(thread2);
    join_all();
    printf("Main: final x = %d\n", x);
}
```

The operations x = x + 1 and x = x + 2 each consist of three atomic steps:

1. **Read the value of x from memory into a register.**
2. **Add the constant to the register.**
3. **Write the register value back to memory.**

Assume that printf() happens immediately after the write.

1. **Enumerate two possible interleavings of the two threads' operations, step by step, showing the value of x after each atomic step.**
2. **For each interleaving, show what lines could appear on the screen, and what the final value of x printed by main would be.**
3. **Explain why the outputs are inconsistent across different runs.**
   What type of concurrency bug is this?
4. **Rewrite the pseudocode using a mutex lock to make the result deterministic.**
   Then state what the only possible output would be after applying the fix.

Ans.

Each thread performs:

Thread	**Step**	**Operation**	**Description**
T1	1	R1	read x
T1	2	A1	add 1
T1	3	W1	write result
T2	4	R2	read x
T2	5	A2	add 2

T2	6	W2	write result

Possible Interleavings

Interleaving 1 - Sequential (T1 fully before T2)

Step	Operation	x Value	Output
1	T1 reads x=0	0	-
2	T1 adds 1	-	-
3	T1 writes 1	1	T1: x = 1
4	T2 reads x=1	1	-
5	T2 adds 2	-	-
6	T2 writes 3	3	T2: x = 3
-	End	x=3	Main: final x = 3

Output (one possible run):
T1: x = 1
T2: x = 3
Main: final x = 3

Interleaving 2 - Overlapping (Race Condition)

Step	Operation	x Value	Output
1	T1 reads x=0	0	-
4	T2 reads x=0	0	-
2	T1 adds 1	-	-
5	T2 adds 2	-	-
3	T1 writes 1	1	T1: x = 1
6	T2 writes 2	2	T2: x = 2
-	End	x=2	Main: final x = 2

Output (another possible run):
T1: x = 1
T2: x = 2
Main: final x = 2

- If T2 executes first fully: final x = 3 again, but outputs may be reordered.
- If both read x=0 before either writes, the increment from one thread is lost.

Explanation:

- The outputs differ because both threads read the same initial value of x (0) before either wrote back the updated result.
- Therefore, one update overwrites the other - a classic race condition.
- The final x depends on the timing and interleaving of the two threads' atomic steps.

Use a mutex lock to ensure only one thread modifies x at a time.

```
mutex lock;

void* thread1(void* arg) {
   lock_acquire(&lock);
   x = x + 1;
   printf("T1: x = %d\n", x);
   lock_release(&lock);
}

void* thread2(void* arg) {
   lock_acquire(&lock);
   x = x + 2;
   printf("T2: x = %d\n", x);
   lock_release(&lock);
}
```

Now, the interleaving no longer matters:

- Whichever thread runs first sets x to 1 or 2.
- The second adds on top of that.
- Final value is always 3.

Deterministic Output (in any order):
T1: x = 1
T2: x = 3
Main: final x = 3

or
T2: x = 2
T1: x = 3
Main: final x = 3
Both are correct and consistent.