Homework - 3

Sriamoghavarsha Beerangi Srinivasa

October 19, 2024

Quiescent consistency

Example : Consider a queue where multiple threads are performing `enqueue` and `dequeue` operations.

```
A: --- enq(1) ----------------------------
B: --------- enq(2) ------------------------
C: ---------------- deq() -> 2 -------------
D: ---------------------------- deq -> 1 ---
```

Here, the operations are not sequentially consistent because the dequeue operations do not follow the order of the enqueue operations. According to sequential consistency, `dequeue()` should return 1 first and then 2.

However, if the system enters a quiescent state after these operations (no further enqueue or dequeue operations), the system would still be quiescently consistent because the final state of the queue reflects the correct completion of all operations (the queue is empty).

Problem 2

Example 2: Consider a shared variable `x`, initialized to 0, where multiple threads perform `read` and `write` operations.

```
A: ---- W(x = 1) -----------------
B: --------- R(x) -> 0 -----------
C: -------------- W(x = 2) -------
D: ------------------ R(x) -> 2 --
```

Here, the operations are sequentially consistent because A writes $x = 1$, B reads $x = 0$ before the write takes effect, C writes $x = 2$, and D reads the latest value $x = 4$.

However, this execution is not quiescently consistent. After quiescence (no further writes or reads), `B` has seen the old value of `x (0)`, which violates quiescent consistency. After quiescence, all threads should observe the most recent updates to `x`. Therefore, `Thread B` should have seen either `x = 1` or `x = 2`, but not `x = 0`.

Sequential consistency

Problem 3

a) Sequentially consistent

```
P1 : ---------- W(x, 1) ---------
P2 : - R(x, 0) ---------- R(x, 1)
```

b) Not sequentially consistent: P2 reads $x = 1$ and then $x = 0$, which contradicts the idea of a consistent global order. Once $x = 1$ is read, there's no way to read $x = 0$ again.

```
P1 : W(x, 1) ----------------------
P2 : --------- R(x, 1) ---- R(x, 0)
```

c) Sequentially consistent

```
P1 : W(x, 1) ----------------------------
P2 : ------------------- W(x, 2) ---------
P3 : --------- R(x, 1) ----------- R(x, 2)
```

d) Sequentially consistent

```
P1 : ------------------- W(x, 1) ---------
P2 : W(x, 2) ----------------------------
P3 : --------- R(x, 2) ----------- R(x, 1)
```

e) Not sequentially consistent: Once P1 writes $x = 1$, P4 will not be able to read $x = 2$ afterward, hence violating sequential consistency.

```
P1 : ------------------- W(x, 1) ----------------------
P2 : W(x, 2) ------------------------------------------
P3 : --------- R(x, 2) ----------- R(x, 1) -------------
P4 : ---------------------------- R(x, 1) ----- R(x, 2)
```

f) Not sequentially consistent: P4 reads $x = 0$ and $y = 0$, which is inconsistent since both P1 and P2 have written $x = 1$ and $y = 1$.

```
P1 : W(x, 1) ---- R(x, 1) ---- R(y, 0)
P2 : W(y, 1) ---- R(y, 1) ---- R(x, 1)
P3 : ---- R(x, 1) ---- R(y, 0)
P4 : ---- R(y, 0) ---- R(x, 0)
```

g) Not sequentially consistent: P3 reads $x = 0$ after P1 has written $x = 1$, which violates sequential consistency.

```
P1 : W(x, 1) ---- R(x, 1) ---- R(y, 0)
P2 : W(y, 1) ---- R(y, 1) ---- R(x, 1)
P3 : ------------ R(y, 1) ---- R(x, 0)
```

Problem 4

Let the operations be represented as $W(a,1) \implies a = 1$ and $P(a,b) \implies print(a,b)$

a) 001011 :

```
P1 : - W(x, 1) --- P(y,z) ----------------------------------------
P2 : ---------------------- W(y, 1) --- P(x,z) ----------------------
P3 : ------------------------------------------ W(z, 1) --- P(x,y) -
```

b) 001111 :

```
P1 : ---------------------- W(x, 1) ----------- P(y,z) ----------
P2 : - W(y, 1) --- P(x,z) ---------------------------------------
P3 : ------------------------------- W(z, 1) ---------- P(x,y) -
```

c) 001110 : There is no valid interleaving of the instructions that might produce a legal output while maintaining sequential consistency as you can from the timeline below. All the processes must set their respective variables before executing their respective print statements and hence the last 0 in the sequence can never be realized.

```
P1 : ------------------------------- W(x, 1) ----------- P(y,z) -
P2 : - W(y, 1) --- P(x,z) ---------------------------------------
P3 : ---------------------- W(z, 1) ---------- P(x,y)-------------
```

Problem 5

a) $H|B$ is the subhistory of H consisting only of operations performed by process B. From H, the operations performed by B are:

```
r.write(1)
r:void
r.read()
r:1
```

b) $H|r$ is the subhistory of H consisting of operations on the register r. From H, the operations on register r are:

```
r.write(1)
r.read()
r.write(2)
r:1
r:void
r:void
r.read()
r:1
```

c) A complete subhistory H' includes all operations with both their invocation and response. The last operation by C, $r.read()$, should return $r:1$, as it reads the most recent write, which was by B. Thus, the complete subhistory H' is:

```
B: r.write(1)
A: r.read()
C: r.write(2)
A: r:1
B: r:void
C: r:void
B: r.read()
B: r:1
A: q.write(3)
C: r.read()
C: r:1
A: q:void
```

Here, C's final read returns $r:1$, conforming to the most recent write by B.

d) Yes, H' is sequential if we impose the following valid global order:

i) C : r.write(2) happens first.

ii) B : r.write(1) happens after C : r.write(2), overwriting C's value of 2 with 1.

iii) A : r.read() $\rightarrow r : 1$ happens after B : r.write(1), reading the most recent value of 1.

iv) B : r.read() $\rightarrow r : 1$ happens after B's own write, correctly returning 1.

v) C : r.read() $\rightarrow r : 1$ happens last, also returning the most recent write of $r : 1$.

This order is valid, respecting both program order and the values returned by reads.

e) Yes, H' is well-formed because:

i) Each process performs at most one operation at a time, with no overlapping operations.

ii) The operations within each process respect program order. For example, process A performs $r.read()$ before writing to q, and process B writes to r before reading from r.

f) H' is linearizable if it meets the following conditions:

i) Real-time order is respected: Operations must occur in a sequence consistent with their real-time invocation.

ii) Reads return the most recent write.

Checking the history:

- C : r.write(2) happens first, but B : r.write(1) overwrites this value.
- A $r : 1$, which is consistent with B's most recent write.
- B $r : 1$, also reading the value it just wrote.
- C $r : 1$, reading the same value written by B.

Since all reads return the correct, most recent value and the real-time order is preserved, H' is linearizable.

g) If we swap the first two events:

```
Original order: B: r.write(1), A: r.read()
Swapped order : A: r.read(), B: r.write(1)
```

In this case, if A's read happens before B's write, A would not be able to return $r : 1$, as the value has not been written yet. However, if the linearization point places A's read logically after B's write (even if A was invoked first), A can still return $r : 1$. Hence, the resulting history equivalent to H.

Using Porcupine Linerizability checker to verify Problem 5

Both the original history and the swapped history were deemed **linearizable**, as indicated by the grey lines for valid linearization points in the visualizations.

- **Original History Visualization**: In the original history, all operations (puts and gets) respect the linearizability constraints. The final `get()` operations from clients return the most recent value '1', which was written by **Client 1 (B)**.
- **Swapped History Visualization**: In the swapped history, where **Client 0 (A)**'s `get()` was invoked before **Client 1 (B)**'s `put('1')`, the checker still shows that the history is **linearizable**. This means that although the real-time invocation order was swapped, the **linearization points** can be arranged in a way where **Client 0 (A)**'s read happens logically after **Client 1 (B)**'s write, thus returning the correct value '1'.

Figures:

Below are the visualizations from the Porcupine checker:

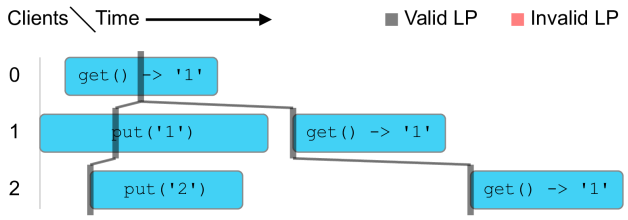

Figure 1: Original History H' Visualization For Register R - Linearizable

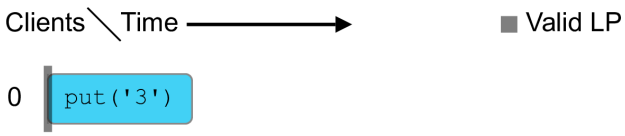

Figure 2: Swapped History H' Visualization For Shared Register Q - Linearizable

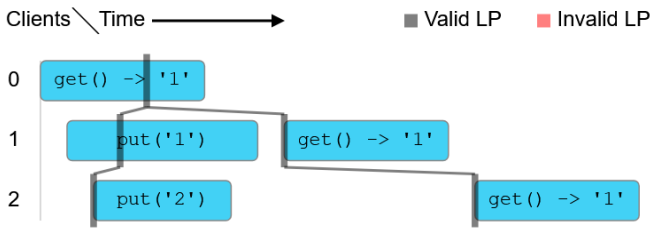

Figure 3: Swapped History H' Visualization For Shared Register R - Linearizable

Problem 6

a) To check if the history is **linearizable**, we need to verify whether the operations can be reordered to respect both **real-time order** and the **specification of a FIFO queue**.

The definition of a FIFO queue dictates that the element enqueued first should be the first to be dequeued. Since **A** enqueued x first and **B** enqueued y second, when **A** dequeues, it should receive x, not y. However, in the given history, **A** receives y after the dequeue, which violates the FIFO order. Thus, the history is **not linearizable**.

b) For the history to be **sequentially consistent**, we need to verify if the operations respect the program order of each process and produces the same result as some sequential execution.

Even though the FIFO order is violated, the operations are consistent with the program order of each process, hence the history is **sequentially consistent**.

Using Porcupine Linerizability checker to verify Problem 6

Checker shows that the history is **not linearizable**, as indicated by grey lines for valid points and red lines for invalid points.

Client 0 (A) enqueues x and then performs a dequeue operation, expecting to receive x, the value it enqueued first. However, instead of dequeuing x, **A** receives y, which was enqueued by **Client 1 (B)** afterward. This violates the FIFO property because the first value enqueued (i.e., x) should be the first value dequeued. This shows that the history cannot be linearized in a way that respects both the real-time order and the FIFO specification.

Figures:

Below are the visualizations from the Porcupine checker:

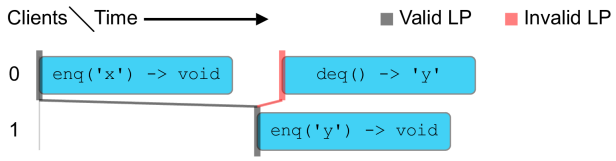

Figure 4: History H' Visualization For FIFO Queue - Not Linearizable

Problem 7

This execution violates the fundamental property of a FIFO queue, where the first element enqueued should be the first element dequeued. In this case:

- x was enqueued first by Task A.
- y was enqueued later by Task B.
- However, both tasks dequeue y, even though x should have been dequeued first.

The fact that **both Task A and Task B return y as the result of their dequeue operations** shows an inconsistency in the queue's behavior. A FIFO queue should ensure that x is dequeued before y, but this is not happening in the observed history.

Hence, This execution is **not linearizable** because operations violate the real-time order of enqueue and dequeue operations, and the queue does not maintain the expected FIFO ordering.

Problem 8

In this case:

- Task A enqueues x first and completes this before y is enqueued by Task B.
- Task B enqueues y after x, and the enqueue operations respect the real-time order.
- Task B dequeues x, which maintains the FIFO order as x was the first element enqueued.

Since the operations respect the real-time order and FIFO behavior, the execution is **linearizable**.

Hence, The execution is **linearizable** because the operations follow the correct real-time order and the queue behaves according to FIFO semantics.

Problem 9

This problem asks to give an example showing that the implementation in Figure 1 is **not linearizable**.

The issue with this queue implementation lies in the non-atomic nature of the `enq()` method. The problem is that `tail` is incremented, reserving a slot, but the actual item placement in the array is done separately. This can lead to incorrect behavior in concurrent scenarios, where operations do not appear to be linearizable.

Example :

We have two tasks $TaskA$ and $TaskB$ concurrently enqueuing items and another task (Task C) dequeuing items. Below is a timeline example:

- **Time 0:**
  - Task A calls `enq(x)`.
  - Task A increments `tail` to 1 (reserving slot 0), but hasn't stored `x` in the array yet. (The assignment to `items[0]` hasn't occurred yet.)
- **Time 1:**
  - Task B calls `enq(y)`.
  - Task B increments `tail` to 2 (reserving slot 1).
  - Task B stores `y` in `items[1]`. Now `y` is stored in slot 1, but `x` is still not stored in slot 0.
- **Time 2:**
  - Task C calls `deq()`.
  - Task C reads from slot 0, but it sees `items[0] == null` because Task A has not yet stored `x` in that slot.
  - Task C throws an `EmptyException` since it thinks the queue is empty or reads the wrong value.
- **Time 3:**
  - Task A stores `x` in `items[0]`, but the `deq()` operation has already completed incorrectly.

The queue is **not linearizable** because the order in which items are enqueued and dequeued does not follow a real-time ordering that preserves consistency.

Problem 10

Example 1: Failure at Line 15

At line 15, the thread reserves a slot in the queue by incrementing the `tail`, but it has not yet placed the value in the array. Consider the following timeline:

- **Time 0: Thread P1** calls `enq(x)` and increments `tail` to 1, reserving slot 0, but `x` is not yet stored in the array.
- **Time 1: Thread P2** calls `enq(y)` and increments `tail` to 2, reserving slot 1. **P2** immediately stores `y` in `items[1]`.
- **Time 2: P1** finishes its operation by storing `x` in `items[0]`.

```
Time 0: items = [null, null, null, ...]
Time 1: items = [null, y, null, ...]
Time 2: items = [x, y, null, ...]
```

If we consider line 15 as the linearization point, it would imply that **P1**'s `enq(x)` completes before **P2**'s `enq(y)`, but **P2**'s value appears in the queue first, violating real-time order. Therefore, line 15 cannot be the linearization point.

Example 2: Failure at Line 16

At line 16, the value is stored in the queue, but other threads might already be interacting with the queue. Consider this scenario:

- **Time 0: Thread P1** calls `enq(x)` and increments `tail` to 1, reserving slot 0.
- **Time 1: Thread P3** calls `deq()` and checks `items[0]`, which is still `null`.
- **Time 2: P1** finishes its `enq(x)` operation by storing `x` in `items[0]`.

The queue states at each step are:

```
Time 0: items = [null, null, null, ...]
Time 1: items = [null, y, null, ...] (P3 sees an empty queue)
Time 2: items = [x, y, null, ...]
```

If we consider line 16 as the linearization point, it would imply that **P1**'s `enq(x)` completes after **P3**'s `deq()`, but **P3** did not see the value `x`, violating the real-time order. Therefore, line 16 cannot be the linearization point.

The `enq()` operation is **not linearizable** because it lacks a single atomic point where the operation occurs. The slot is reserved (line 15) before the value is placed in the queue (line 16), leaving an intermediate state visible to other threads. This can cause inconsistent views, where operations like `deq()` might see an empty queue or miss an enqueued value. These inconsistencies violate the real-time ordering required for linearizability, making the operation **not linearizable**.