Concurrent Objects and Correctness (Safety)

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University
Sequential Objects and Correctness

➢ An object is a data object (e.g., queue, variable, ...)
  ▪ Each object provides a set of methods which are the only way to manipulate that object
  ▪ An object has a well-defined state

➢ There are many ways to describe how an object’s methods behave
  ▪ Formal specification
  ▪ Plain English
  ▪ The application program interface (API) documentation
    • lies somewhere in between
Specification for Sequential Objects

➢ The following information is included
  ▪ Definition of the entity
  ▪ Description of inputs and where they come from
  ▪ Description of outputs and where they go to
  ▪ Indication of other entities required
  ▪ Precondition
    • A predicate expected to hold at a function call
    • A failure can be blamed on the caller
  ▪ Post conditions
    • A predicate expected to hold at a function return
    • A failure can be blamed on the callee
  ▪ The side effects
Concurrent Objects and Correctness

A concurrent object is a data object shared by concurrent threads
- Unfortunately, specifications for sequential objects do not work well for concurrent objects
- An object’s methods can be invoked by concurrent threads, then the method calls can overlap in time, and it no longer makes sense to talk about their order

What does it mean, in a multithreaded program, if x and y are enqueued on a FIFO queue during overlapping intervals?
- Which will be dequeued first?
- Can we continue to describe methods in isolation, via preconditions and postconditions, or must we provide explicit descriptions of every possible interaction among every possible collection of concurrent method calls?
Concurrent Objects and Correctness

Even the notion of an object’s state becomes confusing

- In single-threaded programs, an object must assume a meaningful state only between method calls
- For concurrent objects, however, overlapping method calls may be in progress at every instant, so the object may never be between method calls
- Any method call must be prepared to encounter an object state that reflects the incomplete effects of other concurrent method calls, a problem that simply does not arise in single-threaded programs
## Sequential vs. Concurrent

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each method described independently.</td>
<td>Need to describe all possible interactions between methods. (what if enq and deq overlap? …)</td>
</tr>
<tr>
<td>Object’s state is defined between method calls.</td>
<td>Because methods can overlap, the object may never be between method calls…</td>
</tr>
<tr>
<td>Adding new method does not affect older methods.</td>
<td>Need to think about all possible interactions with the new method.</td>
</tr>
</tbody>
</table>
Concurrency and Correctness

➢ Three concurrency models
  ▪ Quiescent consistency
  ▪ Sequential consistency
  ▪ Linearizability

➢ The above models permit us specify and reason about concurrent objects using known techniques from the sequential domain
  ▪ In general, there are two important issues about concurrency, safety and liveness
  ▪ Here, we will focus on safety (correctness)
Quiescent Consistency

➢ An object is quiescent if it has no pending method calls

➢ Quiescent consistency (correctness property)
  ▪ Method calls should appear to happen in a one-at-a-time, sequential order
  ▪ Method calls separated by a period of quiescence should appear to take effect in their real-time order

➢ Example
  ▪ Suppose A and B concurrently enqueue x and y in a FIFO queue
  ▪ The queue becomes quiescent, and then C enqueues z
  ▪ We may not be able to predict the relative order of x and y in the queue, but we know they are ahead of z
Sequential Consistency

➢ Program order
  ▪ The order in which a single thread issues method calls
  ▪ Method calls by different threads are unrelated by program order

➢ Sequential consistency
  ▪ Method calls should appear to happen in a one-at-a-time, sequential order
  ▪ Method calls should appear to take effect in “program order”
    • Sequential consistency is free to reorder method calls as long as the program order is preserved
FIFO Queue Example 1

➢ Two sequential orderings
  ▪ A enq x → B enq y → B deq x → A deq y
  ▪ B enq y → A enq x → A deq y → B deq x

➢ These two sequential orderings (1) preserve program order and (2) satisfy the sequential FIFO semantics
  ▪ So the above concurrent execution is sequentially consistent
This execution may violate our intuitive notion of how a FIFO queue should behave
- The call enqueuing \( x \) finishes before the call dequeuing \( y \) starts, so although \( y \) is enqueued after \( x \), it is dequeued before

Nevertheless, this execution is sequentially consistent
- It is acceptable to reorder method calls that occur in different threads
Notes on Sequential Consistency

- In most modern multiprocessor architectures, memory reads and writes are not sequentially consistent
  - They can be typically reordered in complex ways
  - In those specific cases where programmers need sequential consistency, they must ask for it explicitly
  - The architectures provide special instructions (usually called memory barriers or fences) that instruct the processor to propagate updates to and from memory as needed, to ensure that reads and writes interact correctly
  - In the end, the architectures do implement sequential consistency, but only on demand
Compositionality

➢ A correctness property $P$ is compositional if, whenever each object in the system satisfies $P$, the system as a whole satisfies $P$.

➢ Sequential consistency is not compositional in that a correct sequential execution may not be found.
Example 1 (Composition of Two Queues)

➢ FIFO specification
   ▪ \(<p.enq(y) B> \rightarrow <p.enq(x) A>\), since A dequeues y from p
   ▪ \(<q.enq(x) A> \rightarrow <q.enq(y) B>\), since B dequeues x from q

➢ Program order
   ▪ \(<p.enq(x) A> \rightarrow <q.enq(x) A>\)
   ▪ \(<q.enq(y) B> \rightarrow <p.enq(y) B>\)

➢ Together, these orderings form a cycle
   ▪ \(<p.enq(y) B> \rightarrow <p.enq(x) A> \rightarrow <q.enq(x) A> \rightarrow <q.enq(y) B> \rightarrow <p.enq(y) B>\)
Motivation for Linearizability

- Sequential consistency arguably the most widely used consistency definition
  - E.g. Almost all commercial databases
  - Many multi-processors

- But sometimes not strong enough

```
P: write(x,1) ok() Q: read(x) ok(0) ≡ P: write(x,1) ok() Q: read(x) ok(0)
```

Sequentially consistent, but does not respect precedence order
Linearizability

- The principal drawbacks of sequential consistency
  - It is not compositional
  - It does not respect precedence order

- We propose the following way out of this dilemma
  - Let us replace the requirement that method calls appear to happen in program order with the following stronger restriction
  - “Each method call should appear to take effect instantaneously at some moment between its invocation and response”
History (Definition)

- A history $H$ is a sequence of invocations and responses ordered by wall clock time.
- A history $H$ is sequential if any invocation is always immediately followed by its response (atomicity).
- A sequential history $H$ is legal if all responses satisfies the sequential semantics of the object.

Sequential history:
\[\text{inv}(p, \text{read}, X) \quad \text{resp}(p, \text{read}, X, 0) \quad \text{inv}(q, \text{write}, X, 1) \quad \text{resp}(q, \text{write}, X, \text{OK})\]

Concurrent history:
\[\text{inv}(p, \text{read}, X) \quad \text{inv}(q, \text{write}, X, 1) \quad \text{resp}(p, \text{read}, X, 0) \quad \text{resp}(q, \text{write}, X, \text{OK})\]
History (Equivalence)

Two histories are equivalent if they have the exactly same set of events

- Same events imply all responses are the same
- Ordering of the events may be different
Linearizability

➢ Intuitive definition
  ▪ Each method call should appear to take effect instantaneously at some moment (linearization point) between its invocation and response

➢ Formal definition
  ▪ It is equivalent to some legal sequential history S, and
  ▪ The operation partial order induced by H, i.e., program order, is a subset of the operation partial order induced by S

➢ Are the two definitions the same?
Linearizability

- Linearizability is stronger than sequential consistency
  - Linearizable $\implies$ sequentially consistent

- Linearization point
  - For lock-based implementations, each method’s critical section can serve as its linearization point
  - For implementations that do not use locking, the linearization point is typically a single step where the effects of the method call become visible to other method calls
Example: FIFO Queue

(a) $H_1$ (linearizable)

(b) $H_2$ (not linearizable)
Example: FIFO Queue

(c) \( H_3 \) (linearizable)

(d) \( H_4 \) (not linearizable)

Fig. 1. FIFO queue histories.
Compositionality and Nonblocking Property

➢ Linearizability is compositional (local property)
  ▪ If an execution of a concurrent object X is linearizable and an execution of a concurrent object Y is linearizable, then we can construct a new linearizable object from X and Y
    • H is linearizable if and only if, for each object X, H | X is linearizable

➢ Linearizability is a nonblocking property
  ▪ A pending invocation of a totally defined operation is never required to wait for another pending invocation to complete
    • A method is total if, like enq, it is defined for every object value, otherwise it is partial, like deq, which left undefined for the empty queue
  ▪ Blocking may occur as artifacts of particular implementations of linearizability, but it is not inherent to the correctness property itself (we may construct nonblocking implementations of linearizable objects)
Example 1 (Lock-based Implementation)

➢ Is this counter implementation linearizable?

Thread A

```c
item nextProduced;

if (user_wants_to_write == 1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    mutex_lock(&mutex);
    counter++;
    mutex_unlock(&mutex);
}
```

Thread B

```c
item nextConsumed;

If (user_wants_to_read == 1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    mutex_lock(&mutex);
    counter--;
    mutex_unlock(&mutex);
}
```

➢ Linearizable
  - The critical sections are linearization points
Example 2 (Wait-free and Race)

➢ Is this counter implementation linearizable?

Thread A

item nextProduced;

if (user_wants_to_write == 1) {
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++; 
}

Thread B

item nextConsumed;

If (user_wants_to_read == 1) {
    while (counter == 0) 
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--; 
}

➢ Not linearizable
Example 2 (Wait-free and Race)

- "counter++"
  - read 1
  - Increment
  - write 2

- "counter--"
  - read 1
  - Decrement
  - write 0

Not linearizable
(We cannot find a legal sequential history)
Linearizability vs. Serializability

In concurrency control of databases, a concurrent transaction schedule is serializable if its outcome is equivalent to one that executes the transactions serially in some order, i.e. without overlapping in time.

- Serializability is not compositional
- Serializability is inherently a blocking property
Concurrent Shared Memory
Concurrent Shared memory

➢ Shared-memory computation
  ▪ consists of multiple threads, each of which is a sequential program in its own right
  ▪ These threads communicate by calling methods of objects that reside in a shared memory
  ▪ Threads are asynchronous, meaning that they run at different speeds, and any thread can halt for an unpredictable duration at any time

➢ The simplest form of shared-memory computation
  ▪ Concurrent threads apply simple read–write operations to shared memory locations, called registers for historical reasons
Registers

- A register is an object that encapsulates a value that can be observed by a read() method and modified by a write() method
  - In real systems these method calls are often called load and store

```java
public class SequentialRegister<T> implements Register<T> {
    private T value;
    public T read() {
        return value;
    }
    public void write(T v) {
        value = v;
    }
}
```

*Figure 4.2 The SequentialRegister class.*
Concurrent Method Calls

- If method calls do not overlap, a register implementation should behave as shown in the previous figure.
- On a multiprocessor, however, we expect method calls to overlap all the time, so we need to specify what the concurrent method calls mean:
  - One approach is to rely on mutual exclusion: protect each register with a mutex lock acquired by each read() and write() call.
  - However, mutual exclusion can be accomplished by using registers, so it makes little sense to implement registers using mutual exclusion.
Wait-Free Implementation

➢ Here is a different approach

➢ Recall that an object implementation is wait-free if each method call finishes in a finite number of steps, independently of how its execution is interleaved with steps of other concurrent method calls
  ▪ The wait-free condition may seem simple and natural, but it has far-reaching consequences
  ▪ In particular, it rules out any kind of mutual exclusion, and guarantees independent progress, that is, without relying on an operating system scheduler

➢ We therefore require our register implementations to be wait-free
Safe Register

➢ Behavior

▪ A read() call that does not overlap a write() call returns the value written by the most recent write() call

▪ Otherwise, if a read() call overlaps a write() call, then the read() call may return any value within the register’s allowed range of values

➢ The term “safe” is a historical accident

▪ Because they provide such weak guarantees, “safe” registers are actually quite unsafe
Example

➢ R1 returns 0, the most recently-written value
➢ R2 and R3 are concurrent with W(1), so they could return any value in the range of the register
Regular Register

Behavior

- A read() call that does not overlap a write() call returns the value written by the most recent write() call.
- Otherwise, if a read() call overlaps one or more write() calls, then the read() call may return one of the values written by the last write() call or overlapping write() calls.

- Regular registers are quiescently consistent, but not vice versa.
  - It is an intermediate level of consistency between safe and atomic (we will see).
Example

- $R^1()$ returns 0, the most recently-written value
- $R^2()$ and $R^3()$ each return 0 or 1

$W(0)$

$W(1)$
Atomic Register

➢ An atomic register is a linearizable implementation of the sequential register class

➢ Informally, an atomic register behaves exactly as we would expect
  ▪ Each read returns the “last” value written
  ▪ A model in which threads communicate by reading and writing to atomic registers is intuitively appealing, and for a long time was the standard model of concurrent computation
Example

- R1 returns 0, the most recently-written value
- If R2 returns 1, then R3 also returns 1
- If R2 returns 0, then R3 could return 0 or 1