Principles of Concurrency

Week 3
Simple Concurrent Data Structures

Material adapted from Herlihy and Shavit, Art of Multiprocessor Programming, Chapters 9 and 10
Desirable properties of a concurrent object:
- safety: accesses to these object should obey desired consistency properties
- liveness: all threads should be able to eventually access an object if they wish to do so

Simplest kind of object:
- read-write register (name is a historical artifact)

An object is said to be wait-free if:
- every method defined by its implementation is guaranteed to complete in a finite number of steps regardless of the number of other concurrent calls to the method made by other threads
- Guarantees progress without appealing to a scheduler implementation
 Atomic Register 

- A single-writer, multi-reader (SWMR) register implementation:
  - is safe if it ensures that a read() operation not overlapping with a write() operation returns the value of the most recently completed write() in a trace. It provides no guarantees when operations overlap.
  - is regular if it is safe but its writes are not atomic
    - readers can observe the write happening, “flickering” between old and new values

- A multi-writer, multi-reader (MWMR) register implementation:
  - is atomic if it guarantees that every read() returns the value of the most recently completed write().

Suppose \( R^1() \) returns 0. What can \( R^2 \) and \( R^3 \) return in a safe register? atomic register? regular register?
Histories

- A history is a trace of operations on an object
- Every read() call in a history must return the value written by some previously executed write()
  - no “out-of-thin-air” values
- A history can include method invocation and response events
  - A method call in a history is defined by the interval containing the method invocation and its response
  - The set of method calls in a history defines a partial order over the happens-before relation. Why is this not total?
- A register implementation defines a total order on writes
  - This is sometimes known as coherence
Properties

- A regular register does not allow reads to:
  - witness writes from the future in a history
  - witness a write that has been overwritten by another visible write

- An atomic register additionally mandates that an earlier read cannot return a value later than that returned by a later read

\[
\text{if } R^i \rightarrow R^j \text{ then } i \leq j. \quad (*)
\]

The superscripts indicate the index of the write in the history that the read observes.
Constructions

- Can build safe MRSW registers from safe SRSW registers
  - Maintain an array of SRSW registers
  - Writes update every element in the array (non-atomically)
  - Reads guaranteed to see latest non-overlapping write

- Can build atomic SRSW registers from regular SRSW registers
  - An SRSW register has no concurrent reads
  - Requirement that write order is reflected by reads in an atomic register can be violated by a regular register if:
    - two reads overlap the same write and read values out-of-order
  - Prevent this condition from happening by using timestamps that order write calls
    - Each read remembers latest (highest timestamp) value pair read
    - A read that reads a lower value than a previous read simply ignores it
    - Interesting challenge: need to read both timestamp and value atomically
      - E.g., store both in a single 64bit word
Constructions

- Can build an atomic MRSW register from an atomic SRSW register
  - As with the construction of safe MRMW registers from safe SRSW registers, use a table of atomic SRSW registers
  - Writes update table in increasing order; use a “helping” mechanism: earlier readers inform later readers of the values they’ve read
  - Importantly, this does not provide MRMW capability

- Can build an atomic MRMW register from atomic MRSW registers
  - To write to the register, first read all the elements in the table and choose a timestamp higher than any observed and use that to write to the appropriate slot in the array
  - To read, first read all the elements and return the element with the highest stamp (like the Bakery algorithm)
  - Break ties using thread ids
### An MRSW Atomic Register

#### 4.2 Register Constructions

**Figure 4.13** An execution of the MRSW atomic register. Each reader thread has an index between 0 and 4, and we refer to each thread by its index. Here, the writer writes a new value with timestamp $t^+$ to locations $a_{table}[0][0]$ and $a_{table}[1][1]$ and then halts. Then, thread 1 reads its corresponding column $a_{table}[i][1]$ for all $i$, and writes its corresponding row $a_{table}[1][i]$ for all $i$, returning the new value with timestamp $t^+$. Threads 0 and 3 both read completely after thread 1's read. Thread 0 reads $a_{table}[0][1]$ with value $t^+$. Thread 3 cannot read the new value with timestamp $t^+$ because the writer has yet to write $a_{table}[3][3]$. Nevertheless, it reads $a_{table}[3][1]$ and returns the correct value with timestamp $t^+$ that was read by the earlier thread 1.

#### 4.2.6 An Atomic MRMW Register

Here is how to construct an atomic MRMW register from an array of atomic MRSW registers, one per thread.

- **To write to the register,** a reads all the array elements, chooses a timestamp higher than any it has observed, and writes a stamped value to array element $A$.  

- **To read the register,** a thread reads all the array elements, and returns the one with the highest timestamp. This is exactly the timestamp algorithm used by the Bakery algorithm of Section 2.6. As in the Bakery algorithm, we resolve ties in favor of the thread with the lesser index; in other words, we use a lexicographic order on pairs of timestamp and thread ids.

**Lemma 4.2.7.** The construction in Fig. 4.14 is an atomic MRMW register.

**Proof:** Define the write order among \( write() \) calls based on the lexicographic order of their timestamps and thread ids so that the \( write() \) call by \( A \) with timestamp \( t_A \) precedes the \( write() \) call by \( B \) with timestamp \( t_B \) if \( t_A < t_B \), or if \( t_A = t_B \) and \( A < B \).  

Clearly a \( read() \) call cannot read a value written in $a_{table}[i][j]$ after it is completed, and any \( write() \) call completely preceded by the read has a
Other kinds of locks ...

- Why are approaches like Peterson’s or the Bakery algorithm not sufficient:
  - cost in space and time
  - unexpected interactions with compiler optimizations
  - strong assumptions on multiprocessor hardware behavior
    - all writes are sequentially consistent, i.e., program behaves as a sequential interleaving of concurrent actions

- Consider lower-level approaches that are closer to features supported by the architecture
  - robust to compiler optimizations and underlying hardware
  - implemented as part of the processor’s ISA
Test-and-Set

value : Boolean

fun getAndSet (newVal : Boolean) = {
  prior = value
  value = newVal
  return prior
}

execute this atomically

- value represents a lock
- When getAndSet() returns false, the lock is free
- When getAndSet() returns true, the lock is held
- Acquire a lock by calling getAndSet() until it returns true
- Release lock by calling getAndSet with false
Test-and-set

- Compared with Peterson’s, this lock has a small ($O(1)$) footprint (compared with $O(n)$ for Peterson or Bakery)
- Key difference:
  - It relies on an atomic Read-Modify-Write (RMW) instruction
- Should have clear scaling advantages

The diagram shows a graph with time on the y-axis and threads on the x-axis. The TAS lock line is compared against the ideal line, indicating the performance of the lock relative to an ideal scenario.
Variant of test-and-set that repeatedly loops waiting for the lock to become free before trying to acquire it:

```javascript
fun acquire() = {
    while (true) {
        while (value == true) {}
        if (!getAndSet(true)) { return }
    }
}
```
Compare-and-Swap

Three operands:
- a memory location (v)
- an expected value (old)
- a new value (new)

Atomically update v with new if its value is old, and return old

Use this for synchronization:
- read value A from v
- perform some computation to derive new value B
- use CAS to write B back to V

```scala
val oldVal = counter.val
while (counter.CAS (oldVal, oldVal + 1) != oldVal))
  oldVal = counter.val
return counter.val
```

Lock-free counter:
Taxonomy

- An algorithm is said to be wait-free if every thread makes process in the face of arbitrary delay (or even failure) of other threads

- An algorithm is said to be lock-free if some thread always makes progress
  - starvation possible

- An algorithm is said to be obstruction-free if at every point in the program’s execution, there exists some thread that if executed in isolation for a bounded number of steps will complete