Principles of Concurrency

Lecture 6
Concurrent Data Structures (cont)

Material adapted from Herlihy and Shavit, Art of Multiprocessor Programming, Chapters 10 and 11
The Problem

- Data structures like queues and stacks differ from sets:
  - no contains() method
  - an item can appear more than once

- These structures can be:
  - bounded or unbounded
  - total, partial, or synchronous:
    - total: every operation is non-blocking e.g., attempting to retrieve an element from an empty stack simply returns failure
    - partial: certain conditions may need to hold before an operation is allowed to complete, e.g., adding an element to a full bounded queue must wait until an element is removed
    - synchronous: one method waits for another to overlap its call/return interval, e.g., a method that adds an element to a queue blocks until a request to remove an element is received. These implementations implement a form of rendezvous protocol.
A bounded partial queue

- enq() and deq() operations operate over different parts of the queue
- As long as the queue is neither full nor empty, these operations can proceed without interference
- Use locks for enq() and deq() to regulate concurrent enq() and deq() operations

fun <T> enq(x : T) = {
  var mustWakeDequeuers = false
  enq.acquire()
  while (size.get() == capacity {
    wait(notFullCondition)
  }
  var node = new Node(x)
  tail.next = tail = node
  if (size.getAndIncrement() == 0) {
    mustWakeDequeuers = true
  enq.release()
  if (mustWakeDequeuers) {
    deq.acquire()
    signal(notEmptyCondition)
  deq.release()
  }
}
An unbounded total queue always successfully enqueues and item, with deq() throwing an exception if the queue is empty.

This implementation does not requiring checking any conditions before proceeding with an operation.

```java
fun <T> enq(item : T) = {
  var node = new Node(value)
  while (true) {
    val last = tail.get()
    val next = last.next.get()
    if (last == tail.get()) {
      if (next == null) {
        if (last.next.CAS(next, node)) {
          tail.CAS(last, node)
          return
        } else {
          tail.CAS(last, next)
        }
      } else {
        tail.CAS(last, node)
        return
      }
    } else {
      tail.CAS(last, next)
    }
  }
}
```

Appending a node to the list, and setting the tail to point to that node are not atomic. Use a “helping” mechanism to allow other nodes to finish the enq() of an operation that was preempted between these two actions.
fun <T>deq() {
    while (true) {
        val first = head.get()
        val last = tail.get()
        val next = first.next.get()
        if (first == head.get()) {
            if (first == last) {
                if (next == null) {
                    throw EmptyExn()
                } else {
                    tail.CAS(last, next)
                }
            } else {
                val value = next.value
                if (head.CAS(first, next) {
                    return value
                }
            }
        } else {
            tail.CAS(last, next)
        }
    }
}
Unbounded Lock-free Queue

Basic operation

Dequeuer’s help fix-up lagging tail from concurrent enqueue operations
The ABA Problem

How do we deal with dequeued nodes? In a language without garbage collection, we need to provide our own memory management mechanism.

Idea: every thread maintains its own local node freelist. An enqueuing operation removes a node from the freelist (or allocates a new node if the list is empty). A dequeueing thread adds the node back to the freelist.

Subtle problem:
- A \texttt{deq()} operation observes a node \( a \) followed by \( b \)
- It attempts to update head to point to \( b \) using CAS and is preempted
- Concurrently, other \texttt{deq()} operations remove both \( a \) and \( b \)
- Node \( a \) is recycled
- The preempted node resumes and (incorrectly) observes that head still points to \( a \) and completes the CAS operation so that head now points to \( b \) (a recycled node)

![Diagram of ABA problem](image)
fun tryPush(n : Node) = {
    val oldTop = top.get()
    n.next = oldTop
    top.CAS(oldTop, node)
}

fun <T>push(value : T) = {
    var node = new Node(value)
    while (true) {
        if (tryPush(node)) {
            return
        }
    }
}

fun tryPop() = {
    val oldTop = top.get()
    if (oldTop == null) {
        throw EmptyExn();
    }
    val newTop = oldTop.next()
    if (top.CAS(oldTop, newTop) {
        return oldTop
    } else {
        return null
    }
}

fun <T>pop() = {
    while (true) {
        val returnNode = tryPop()
        if (returnNode != null) {
            return returnNode.value
        }
    }
}

When can the CAS operations in tryPush() and tryPop() fail?
Lock-free Stack

(a)

A:push()

top

A:pop()

top

Figure 11.1
A Lock-free stack. In Part (a) a thread pushes value \textit{a} into the stack by applying a \texttt{compareAndSet()} to the \texttt{top} field. In Part (b) a thread pops value \textit{a} from the stack by applying a \texttt{compareAndSet()} to the \texttt{top} field.

```java
public class LockFreeStack<T> {
    AtomicReference<Node> top = new AtomicReference<Node>(null);
    static final int MIN_DELAY = ...;
    static final int MAX_DELAY = ...;
    Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);

    protected boolean tryPush(Node node) {
        Node oldTop = top.get();
        node.next = oldTop;
        return (top.compareAndSet(oldTop, node));
    }

    public void push(T value) {
        Node node = new Node(value);
        while (true) {
            if (tryPush(node)) {
                return;
            } else {
                backoff.backoff();
            }
        }
    }
}
```

Figure 11.2
The \texttt{LockFreeStack<T>} class: in the \texttt{push()} method, threads alternate between trying to alter the \texttt{top} reference by calling \texttt{tryPush()}, and backing off using the \texttt{Backoff} class from Fig. 7.5 of Chapter 7. The \texttt{tryPop()} method is called until it succeeds, at which point \texttt{push()} returns the value from the removed node.

As we have seen in Chapter 7, one can significantly reduce contention at the \texttt{top} field using exponential backoff (see Fig. 7.5 of Chapter 7). Accordingly, both
The lockfree stack has poor scalability:
- CAS operations sequentialize access to the stack’s top field

Alternative approach:
- Pair concurrent pushes and pops - threads calling `push()` exchange values with threads calling `pop()`
- This exchange happens without modifying the stack

**LockFreeExchanger:**
- permits two threads to atomically exchange values
- First thread spins waiting for the second until a timeout

**Three state automaton:**
1. Slot initially EMPTY
2. First thread sets it to WAITING using CAS
   - if not successful, retries
   - if successful, spins
3. Another thread that accesses this slot sets the state to BUSY
4. Item can be consumed and the state reset to EMPTY