COMP2212 Programming Language Concepts

Barriers and latches, performance and scalability, CAS and fine-grained concurrency
Latches

- A latch acts as a gate
  - until the latch reaches its terminal state no thread can pass
  - when the terminal state is reached the gate opens and all threads can pass
- `CountDownLatch` class in Java
  - initialised with some positive integer value
  - `await()` and `countDown()`
```java
public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task)
            throws InterruptedException {
        final CountDownLatch startGate = new CountDownLatch(1);
        final CountDownLatch endGate = new CountDownLatch(nThreads);

        for (int i = 0; i < nThreads; i++) {
            Thread t = new Thread() {
                public void run() {
                    try {
                        startGate.await();
                        try {
                            task.run();
                        } finally {
                            endGate.countDown();
                        }
                    } catch (InterruptedException ignored) { }
                }
            }
            t.start();
        }

        long start = System.nanoTime();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end - start;
    }
}
```
Barriers

- Sometimes it is useful to force several threads to synchronise.

- All threads must come together at the barrier point at the same time in order for them to proceed.

- Java `CyclicBarrier` class - initialised with number of processes.

- `await()` on barrier blocks the calling thread until all threads call `await()`.

- Latches wait for events, barriers wait for other threads.
Performance

❖ How to measure: throughput, responsiveness
❖ Avoid premature optimisations: make it correct, then make it fast
❖ Threading increases performance
   ❖ Independent tasks can be run in parallel
❖ Threading introduces overheads
   ❖ thread coordination (locking, signalling, memory synchronisation)
   ❖ context switching, thread creation, OS scheduling overheads
Scalability

❖ Scalability means the ability to improve performance when additional resources are added

❖ If I double clock speed then clearly most programs run faster

❖ But if I add another CPU will it increase performance?

❖ clearly if all my code is sequential, there is no performance benefit since the extra hardware parallelism cannot be used
Amdahl’s Law

- Some tasks are naturally parallel: throwing extra cores at the problem improves performance
- Sequential tasks do not benefit at all from extra processors
- **Amdahl’s law**: suppose that $F$ is the fraction of task that is sequential. Let $N$ be the number of processors available.

\[
speedup \leq \frac{1}{F + \frac{1-F}{N}}
\]

- e.g. suppose that half of task is sequential. Then maximal speedup is 2, regardless of number of processors thrown at the problem!
Improving scalability

- **Reduce lock contention** (multiple threads competing for the same lock) is a problem
  - reduce the duration for which locks are held (reducing lock granularity)
  - reduce the frequency with which locks are requested
  - replace exclusive locks with *other coordination mechanisms that permit greater concurrency*
  - eg. fine-grained concurrency with CAS
Compare and Set (CAS)

- Supported by most processor architectures
  - CAS(V, A, B)
    - V - memory reference
    - A - old value
    - B - new value
- There are two possible executions
  - If V contains A then it is atomically changed to B and CAS returns true
  - If V does not contain A then nothing happens and CAS returns false
- When multiple threads attempt to modify a memory location using CAS, one wins, the others lose
- From Java 5.0, CAS is available on int, long and object references: it is used in implementations of many thread safe data structures in java.util.concurrent
When there are many competing threads, CAS can repeatedly fail, in that case can use \textit{exponential backoff}

```java
import java.util.concurrent.atomic.*;
import net.jcip.annotations.*;

/**
 * ConcurrentStack
 * <p/>
 * Nonblocking stack using Treiber's algorithm
 * 
 * @author Brian Goetz and Tim Peierls
 */
@ThreadSafe
public class ConcurrentStack <E> {
    AtomicReference<Node<E>> top = new AtomicReference<Node<E>>(null);

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = top.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead, newHead));
        return oldHead.item;
    }

    private static class Node <E> {
        public final E item;
        public Node<E> next;

        public Node(E item) {
            this.item = item;
        }
    }
}
Stacks are a relatively simple to implement in fine-grained fashion because there is a single point of contention - the head of the stack.

- for scalability, even this can be improved by adding elimination arrays.

Surprisingly, there are also beautiful solutions for other data structures.

- e.g. Michael-Scott queue, in java as java.util.concurrent.LinkedBlockingQueue
@ThreadSafe
public class LinkedQueue<E> {

    private static class Node<E> {
        final E item;
        final AtomicReference<LinkedQueue.Node<E>> next;

        public Node(E item, LinkedQueue.Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<>(next);
        }
    }

    private final LinkedQueue.Node<E> dummy = new LinkedQueue.Node<E>(null, null);
    private final AtomicReference<LinkedQueue.Node<E>> head
        = new AtomicReference<>(dummy);
    private final AtomicReference<LinkedQueue.Node<E>> tail
        = new AtomicReference<>(dummy);

    public boolean put(E item) {
        LinkedQueue.Node<E> newNode = new LinkedQueue.Node<E>(item, null);
        while (true) {
            LinkedQueue.Node<E> curTail = tail.get();
            LinkedQueue.Node<E> tailNext = curTail.next.get();
            if (curTail == tail.get()) {
                if (tailNext != null) {
                    // Queue in intermediate state, advance tail
                    tail.compareAndSet(curTail, tailNext);
                } else {
                    // In quiescent state, try inserting new node
                    if (curTail.next.compareAndSet(null, newNode)) {
                        // Insertion succeeded, try advancing tail
                        tail.compareAndSet(curTail, newNode);
                        return true;
                    }
                }
            }
        }
    }
}

How can the last CAS fail?