Learning goals for today

- Finish introduction of the concept of Linearizability
  - how to make parallel software correct!
- (Re-)Introduce Sequential Consistency
  - how to argue about memory values
- Consensus and wait-freedom
  - The simplest parallel object that’s already too hard for many
- Begin discussion about transactional memory
  - Optimistic approach
  - Simplifies reasoning and programming
  - Still somewhat in development
  - Need to understand concepts
More formal

Split method calls into two events. Notation:

**Invocation**
A q.enq(x)

- thread
- object
- method
- arguments

**Response**
A q: void

- thread
- object
- result
History

History \( H \) = sequence of invocations and responses

\[
\begin{align*}
A & \text{ q.enq}(3) \quad \text{Invocations and response match, if thread names agree and object names agree} \\
A & \text{ q:void} \\
A & \text{ q.enq}(5) \\
B & \text{ p.enq}(4) \quad \text{An invocation is pending if it has no matching response.} \\
B & \text{ p:void} \\
B & \text{ q.deq()} \\
B & \text{ q:3} \quad \text{A subhistory is complete when it has no pending responses.}
\end{align*}
\]
Projections

Object projections

\[ H | q = \]

- A \texttt{q.enq}(3)
- A \texttt{q:void}
- A \texttt{q.enq}(5)
- B \texttt{q.deq}()
- B \texttt{q:3}

Thread projections

\[ H | B = \]

- B \texttt{p.enq}(4)
- B \texttt{p:void}
- B \texttt{q:3}
Complete subhistories

A q.enq(3)
A q:void
A q.enq(5)

complete (H) =
B p.enq(4)
B p:void
B q.deq()
B q:3

Complete subhistory
History H without its pending invocations.
Sequential histories

A q.enq(3)  ⇆
A q: void
B p.enq(4)  ⇆
B p: void
B q.deq()  ⇆
B q: 3
A q: enq(5)

Sequential history:

- Method calls of different threads do not interleave.
- A final pending invocation is ok.
Well formed histories

Well formed history:
Per thread projections sequential

H = A q.enq(3)
   B p.enq(4)
   B p:void
   B q.deq()
   A q:void
   B q:3

H | A = A q.enq(3)
   A q:void

H | B = B p.enq(4)
   B p:void
   B q.deq()
   B q:3
Equivalent histories

\[ H = \]
- A q.enq(3)
- B p.enq(4)
- B p:void
- B q.deq()
- A q:void
- B q:3

\[ G = \]
- A q.enq(3)
- A q:void
- B p.enq(4)
- B p:void
- B q.deq()
- B q:3

\[ H \text{ and } G \text{ equivalent:} \]
- \[ H|A = G|A \]
- \[ H|B = G|B \]
Legal histories

Sequential specification tells if a single-threaded, single object history is legal
Example: pre-/post conditions

A sequential history $H$ is legal, if
- for every object $x$
- $H|x$ adheres to the sequential specification of $x$
Precedence

A method call precedes another method call if the response event precedes the invocation event.

If no precedence then method calls overlap.
Notation

Given: history $H$ and method executions $m_0$ and $m_1$ on $H$

Definition: $m_0 \rightarrow_H m_1$ means $m_0$ precedes $m_1$

$\rightarrow_H$ is a relation and implies a partial order on $H$. The order is total when $H$ is sequential.
Linearizability

History $H$ is **linearizable** if it can be extended to a history $G$

- appending zero or more responses to pending invocations that took effect
- discarding zero or more pending invocations that did not take effect

such that $G$ is equivalent to a *legal sequential* history $S$ with

$$\rightarrow_G \subseteq \rightarrow_S$$
Invocations that took effect ... ?

A

q.enq(x)

cannot be removed because B already took effect into account

B

q.deq() \rightarrow x

can be removed, nobody relies on this

C

flag.read() \rightarrow ?
What does this mean?

→_G ⊆ →_S

→_G = \{a \rightarrow c, b \rightarrow c\}

→_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}

In other words: S respects the real-time order of G

Linearizability: limitation on the possible choice of S
Composability

**Composability Theorem**
History H is linearizable if and only if for every object x
H|x is linearizable

**Consequence:**

**Modularity**
- Linearizability of objects can be proven in isolation
- Independently implemented objects can be composed
Recall: Atomic Registers

Memory location for values of primitive type (boolean, int, ...)

• operations read and write

Linearizable with a single linearization point, i.e.

• sequentially consistent, every read operation yields most recently written value

• for non-overlapping operations, the realtime order is respected.
Reasoning About Linearizability (Locking)

```java
public T deq() throws EmptyException {
    lock.lock();
    try {
        if (tail == head)
            throw new EmptyException();
        T x = items[head % items.length];
        head++;
        return x;
    } finally {
        lock.unlock();
    }
}
```

Linearization points are when locks are released.
class WaitFreeQueue {
    volatile int head = 0, tail=0;
    AtomicReferenceArray<T>[] items =
        new AtomicReferenceArray<T>(capacity);

    public boolean enq (T x) {
        if (tail - head == capacity) return false;
        items.set((tail+2) % capacity, x);
        tail++;
        return true;
    }

    public T deq() {
        if (tail - head == 0) return null;
        int x = items.get((head+1) % capacity);
        head++;
        return x;
    }
}
Reasoning About Linearizability (Lock-free example)

```java
public T dequeue() {
    while (true) {
        Node first = head.get();
        Node last = tail.get();
        Node next = first.next.get();
        if (first == last) {
            if (next == null) return null;
            else tail.compareAndSet(last, next);
        } else {
            T value = next.item;
            if (head.compareAndSet(first, next))
                return value;
        }
    }
}
```

Linearization points:
- `if (first == last) {
  if (next == null) return null;
  else tail.compareAndSet(last, next);` (first linearization point)
- `T value = next.item;
  if (head.compareAndSet(first, next))
      return value;` (second linearization point)
- `while (true) {` (third linearization point)
Linearizability Strategy & Summary

Identify one atomic step where the method “happens”

- Critical section
- Machine instruction

Does not always work

- Might need to define several different steps for a given method

- Linearizability summary:
  - Powerful specification tool for shared objects
  - Allows us to capture the notion of objects being “atomic”
Sequential Consistency
Alternative: Sequential Consistency

History $H$ is **sequentially consistent** if it can be extended to a history $G$

- appending zero or more responses to pending invocations that took effect
- discarding zero or more pending invocations that did not take effect

such that $G$ is equivalent to a *legal sequential* history $S$.

(Note that $\rightarrow_G \subset \rightarrow_S$ is not required, i.e., no order across threads required)
(Sequential Consistency is weaker than Linearizability)
Alternative: Sequential Consistency

- Require that operations done by one thread respect program order
- No need to preserve real-time order
  - Cannot re-order operations done by the same thread
  - Can re-order non-overlapping operations done by different threads
- Often used to describe multiprocessor memory architectures
Example

A

q.enq(x)  q.deq() → y

B

q.enq(y)

time
Not linearizable

A

q.enq(x) → q.deq() → y

B

x is first in queue

q.enq(y)
Yet sequentially consistent!

A

q.enq(x)

B

q.enq(y)

\( \text{q.deq()} \rightarrow y \)

time
Sequential Consistency is not a local property

(and thus we lose composability...)

Can somebody remind me what “composability” meant?
Proof by Example: FIFO Queue

H =

A p.enq(x)
B q.enq(y)
A p:void
A q.enq(x)
B q:void
B p.enq(y)
A q:void
A p.deq()
B p:void
B q.deq();
A p:y
B q:x
H | q sequentially consistent

H =
A p.enq(x)
B q.enq(y)
A p: void
A q.enq(x)
B q: void
B p.enq(y)
A q: void
A p: deq()
B p: void
B q: deq();
A p: y
B q: x
H|p sequentially consistent

\[ H = \begin{align*}
A & \text{ p.enq(x)} \\
B & \text{ q.enq(y)} \\
A & \text{ p:void} \\
A & \text{ q.enq(x)} \\
B & \text{ q:void} \\
B & \text{ p.enq(y)} \\
A & \text{ q:void} \\
A & \text{ p.deq()} \\
B & \text{ p:void} \\
B & \text{ q.deq();} \\
A & \text{ p:y} \\
B & \text{ q:x}
\end{align*} \]
Ordering imposed by $H | q$ and $H | p$

- $p.\text{enq}(x)$
- $q.\text{enq}(x)$
- $p.\text{deq}() \rightarrow y$

- $q.\text{enq}(y)$
- $p.\text{enq}(y)$
- $q.\text{deq}() \rightarrow x$

$\Rightarrow$ $H$ is not sequentially consistent
Another example: Flags

Each object update (H|x and H|y) is sequentially consistent.
Entire history is not sequentially consistent.
Reminder: Consequence for Peterson Lock (Flag Principle)

flag[id] = true;
victim = id;
while (flag[1-id] && victim == id);

flag[0].write(true)  victim.write(0)  flag[1].read() \rightarrow ?  victim.read() \rightarrow ?
A

flag[1].write(true)  victim.write(1)  flag[0].read() \rightarrow ?  victim.read() \rightarrow ?
B

Sequential Consistency \rightarrow At least one of the processes A and B read flag[1-id] = true. If both processes read flag = true then both processes eventually read the same value for victim().
Side Remark: Quiescent Consistency

Another idea: Programs should respect real-time order of algorithms separated by periods of *quiescence*.

In other words: quiescent consistency requires non-overlapping methods to take effect in their real-time order!
Side Remark: Quiescent Consistency

Quiescent consistency is incomparable to Sequential Consistency

This example is sequentially consistent but not quiescently consistent
Side Remark: Quiescent Consistency

Quiescent consistency is incomparable to Sequential Consistency

This example is quiescently consistent but not sequentially consistent
(note that initially the queue is empty)
Discussion

This pattern
   Write mine, read yours
is exactly the flag principle
   Heart of mutual exclusion
   ▪ Peterson
   ▪ Bakery, etc.
Sequential Consistency seems non-negotiable!

... but:
Many hardware architects think that sequential consistency is too strong.
Too expensive to implement in modern hardware
Assume that flag principle
   Violated by default
   Honored by explicit request (e.g., volatile)

Recall our discussions at the beginning!
Recall our short discussion of caches
Recall: Memories and caches

Memory hierarchy

- On modern multiprocessors, processors do not read and write directly to memory.
- Memory accesses are very slow compared to processor speeds.
- Instead, each processor reads and writes directly to a cache.

While writing to memory

- A processor can execute hundreds, or even thousands of instructions.
- Why delay on every memory write?
- Instead, write back in parallel with rest of the program.
Recall: Memory operations

To **read** a memory location,
  load data into cache.

To **write** a memory location
  update cached copy,
  lazily write cached data back to memory

“Flag-violating” history is actually OK
  processors delay writing to memory
  until after reads have been issued.

Otherwise unacceptable delay between read
  and write instructions.

Writing to memory = mailing a letter

Vast majority of reads & writes
  Not for synchronization
  No need to idle waiting for post office

If you want to synchronize
  Announce it explicitly
  Pay for it only when you need it
Synchronization

Explicit
Memory barrier instruction
   Flush unwritten caches
   Bring caches up to date
Compilers often do this for you
   Entering and leaving critical sections

Implicit
In Java, can ask compiler to keep a variable up-to-date with volatile keyword
Also inhibits reordering, removing from loops & other optimizations
Real-World Hardware Memory

**Weaker than sequential consistency**

But you can get sequential consistency at a price [1]

Concept of linearizability more appropriate for high-level software

[1]: H. Schweizer, M. Besta, T. Hoefler: Evaluating the Cost of Atomic Operations on Modern Architectures, ACM PACT’15
Linearizability vs. Sequential Consistency

Linearizability
- Operation takes effect instantaneously between invocation and response
- Uses sequential specification, locality implies composability
- Good for high level objects

Sequential Consistency
- Not composable
- Harder to work with in software development
- Good way to think about hardware models
Consensus

Literature:
Herlihy: Chapter 5.1-5.4, 5.6-5.8
Consensus

Consider an object c with the following interface

```java
public interface Consensus<T> {
    T decide (T value);
}
```

A number of threads call `c.decide(v)` with an input value `v` each.
Consensus protocol

Requirements on consensus protocol

• **wait-free**: consensus returns in finite time for each thread
• **consistent**: all threads decide the same value
• **valid**: the common decision value is some thread's input

→ linearizability of consensus must be such that first thread's decision is adopted for all threads.
Consensus

c.decide(x) → x

c.decide(y) → x

c.decide(z) → x

A

B

c.decide(y) → x

C

time
Consensus number

A class C solves n-thread consensus if there exists a consensus protocol using any number of objects of class C and any number of atomic registers. Consensus number of C: largest n such that C solves n-thread consensus.
Atomic registers

**Theorem:** Atomic Registers have consensus number 1.

[Proof: Herlihy, Ch. 5, presented later if we have time!]

**Corollary:** There is no wait-free implementation of n-thread consensus, n>1, from read-write registers
Compare and swap/set

Theorem: Compare-And-Swap has infinite consensus number.

How to prove this?
class CASConsensus {
    private final int FIRST = -1;
    private AtomicInteger r = new AtomicInteger(FIRST); // supports CAS
    private AtomicIntegerArray proposed; // suffices to be atomic register

    ... // constructor (allocate array proposed etc.)

    public Object decide (Object value) {
        int i = ThreadID.get();
        proposed.set(i, value);
        if (r.compareAndSet(FIRST, i)) // I won
            return proposed.get(i); // = value
        else
            return proposed.get(r.get());
    }
}
How to use this? Wait-free FIFO queue

Theorem: There is no wait-free implementation of a FIFO queue with atomic registers

How to prove this now?

Proof follows.
Can a FIFO queue implement two-thread consensus?

proposed array

FIFO queue with red and black balls

red ball

black ball
Protocol: Write value to array
Protocol: Take next item from queue
Protocol: Take next Item from Queue

I got the red ball, so I will decide my value

I got the black ball, so I will decide the other’s value from the array
Why does this work?

If one thread gets the red ball
Then the other gets the black ball
Winner decides her own value
Loser can find winner’s value in array
   Because threads write array
   Before dequeueing from queue
Wait-free queue implementation from atomic registers?

Given
   A consensus protocol from queue and registers
Assume there exists
   A queue implementation from atomic registers
Substitution yields:
   A wait-free consensus protocol from atomic registers

However: atomic registers have consensus number 1
Why consensus is important

We know

• Wait-free FIFO queues have consensus number 2
• Test-And-Set, getAndSet, getAndIncrement have consensus number 2
• CAS has consensus number $\infty$

→ wait-free FIFO queues, wait-free RMW operations and CAS cannot be implemented with atomic registers!
# The Consensus Hierarchy

<table>
<thead>
<tr>
<th></th>
<th>Read/Write Registers</th>
<th>FIFO Queue LIFO Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>getAndSet, getAndIncrement, ...</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>CompareAndSet, ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Importance of Consensus by Analogy

Squaring the circle

Geometric way to construct a square with the same area as a given circle with compass and straightedge using a finite number of steps.

There is an algebraic proof that no such construction exists.

People tried it for hundreds of years, some still try it today. Apparently they do not believe the mathematical proof.

Let's not do the same mistake in our field...: provably there is no way to construct certain wait-free algorithms with atomic registers. Don't even try.
Motivation for

Transactional Memory
**Transactional Memory in a nutshell**

**Motivation**: programming with locks is too difficult
Lock-free programming is even more difficult...

**Goal**: remove the burden of synchronization from the programmer and place it in the system (hardware / software)

**Literature:**
- Herlihy Chapter 18.1 – 18.2.
- Herlihy Chapter 18.3. interesting but too detailed for this course.
What is wrong with locking?

**Deadlocks:** threads attempt to take common locks in different orders
What is wrong with locking?

**Convoying**: thread holding a resource R is descheduled while other threads queue up waiting for R
What is wrong with locking?

**Priority Inversion**: lower priority thread holds a resource R that a high priority thread is waiting on
What is wrong with locking?

Association of locks and data established by convention. The best you can do is reasonably document your code!
What is wrong with CAS?

Example: Unbounded Queue (FIFO)

public class LockFreeQueue<T> {
    private AtomicReference<Node> head;
    private AtomicReference<Node> tail;

    public void enq(T item);
    public T deq();
}

public class Node {
    public T value;
    public AtomicReference<Node> next;
    public Node(T v) {
        value = v;
        next = new AtomicReference<Node>(null);
    }
}
Enqueue

Two CAS operations → half finished enqueue visible to other processes
Dequeue

1. Read the value from the head node.
2. CAS the head value to the current node.

S

head

node

value

node

value

node

value

node

value

node

tail
public class LockFreeQueue<T> {
    ..
    public void enq(T item) {
        Node node = new Node(item);
        while(true){
            Node last = tail.get();
            Node next = last.next.get();
            if (last == tail.get()) {
                if (next == null)
                    if (last.next.compareAndSet(next, node)) {
                        tail.compareAndSet(last, node);
                        return;
                    }
                else
                    tail.compareAndSet(last, next);
            }
        }
    }
}
public class LockFreeQueue<T> {
    ..
    public void enq(T item) {
        Node node = new Node(item);
        while(true) {
            Node last = tail.get();
            Node next = last.next.get();
            if (multiCompareAndSet({last.next, tail},{next, last},{node, node})
                return;
        }
    }
}
More problems: Bank account

class Account {
    private final Integer id;  // account id
    private Integer balance;   // account balance

    Account(int id, int balance) {
        this.id = new Integer(id);
        this.balance = new Integer(balance);
    }

    synchronized void withdraw(int amount) {
        // assume that there are always sufficient funds...
        this.balance = this.balance - amount;
    }

    synchronized void deposit(int amount) {
        this.balance = this.balance + amount;
    }
}
Bank account transfer (unsafe)

```cpp
void transfer_unsafe(Account a, Account b, int amount) {
    a.withdraw(amount);
    b.deposit(amount);
}
```

Transfer does not happen atomically

A thread might observe the withdraw, but not the deposit
Bank account transfer (can cause a deadlock)

void transfer_deadlock(Account a, Account b, int amount) {
    synchronized (a) {
        synchronized (b) {
            a.withdraw(amount);
            b.deposit(amount);
        }
    }
}

Concurrently executing:
- transfer_deadlock(a, b)
- transfer_deadlock(b, a)

Might lead to a deadlock
void transfer(Account a, Account b, int amount) {
    if (a.id < b.id) {
        synchronized (a) {
            synchronized (b) {
                a.withdraw(amount);
                b.deposit(amount);
            }
        }
    } else {
        synchronized (b) {
            synchronized (a) {
                a.withdraw(amount);
                b.deposit(amount);
            }
        }
    }
}
Bank account transfer (slightly better ordering version)

```java
void transfer_elegant(Account a, Account b, int amount) {
    Account first, second;
    if (a.id < b.id) {
        first = a;
        second = b;
    } else {
        first = b;
        second = a;
    }
    synchronized (first) {
        synchronized (second) {
            a.withdraw(amount);
            b.deposit(amount);
        }
    }
}
```
Ensuring ordering (and correctness) is **really hard** (even for advanced programmers)

- rules are ad-hoc, and not part of the program
- (documented in comments at best-case scenario)

Locks are **not composable**

- how can you combine *n* thread-safe operations?
- internal details about locking are required
- big problem, especially for programming “in the large”
Problems using locks (cont'd)

Locks are pessimistic

- worst is assumed
- performance overhead paid every time

Locking mechanism is hard-wired to the program

- synchronization / rest of the program cannot be separated
- changing synchronization scheme → changing all of the program
Solution: atomic blocks (or transactions)

What the programmer actually meant to say is:

```c
atomic {
    a. withdraw(amount);
    b. deposit(amount);
}
```

→ This is the idea behind transactional memory also behind locks, isn’t it? The difference is the *execution*!
Transactional Memory (TM)

Programmer explicitly defines **atomic code sections**

Programmer is concerned with:

- **what**: what operations should be atomic

  but, **not how**: e.g., via locking

  the how is left to the system (software, hardware or both)

  (declarative approach)
TM benefits

- simpler and less error-prone code
- higher-level (declarative) semantics (what vs. how)
- composable
- analogy to garbage collection
  (Dan Grossman. 2007. "The transactional memory / garbage collection analogy". SIGPLAN Not. 42, 10 (October 2007), 695-706.)
- optimistic by design
  (does not require mutual exclusion)
TM semantics: **Atomicity**

changes made by a transaction are
made visible atomically

other threads preserve either the initial or the final state, but not any
intermediate states

Note: locks enforce atomicity via mutual exclusion, while transactions
do not require mutual exclusion
Transactions run in isolation

- while a transaction is running, effects from other transactions are not observed
- as if the transaction takes a snapshot of the global state when it begins and then operates on that snapshot

TM semantics: Isolation
Serializability

Thread 0

Thread 1

as if:
Executed Sequentially

(transactions appear serialized)
Transactions in databases

Transactional Memory is heavily inspired by database transactions.

**ACID** properties in database transactions:

- Atomicity
- Consistency (database remains in a consistent state)
- Isolation (no mutual corruption of data)
- Durability (e.g., transaction effects will survive power loss → stored in disk)