Design of Parallel and High-Performance Computing
Fall 2017
Lecture: Linearizability

Motivational video: https://www.youtube.com/watch?v=qx2dRIQXnbs

Instructor: Torsten Hoefler & Markus Püschel
TAs: Salvatore Di Girolamo
Review of last lecture

- **Cache-coherence is not enough!**
  - Many more subtle issues for parallel programs!

- **Memory Models**
  - Sequential consistency
  - Why threads cannot be implemented as a library 😊
  - Relaxed consistency models
  - x86 TLO+CC case study

- **Complexity of reasoning about parallel objects**
  - Serial specifications (e.g., pre-/postconditions)
  - Started to lock things ...
Project: Reminder

- Count 50% of the grade (work, presentation, report)

- Teams of three
  - Important: organize yourselves
  - You may use the mailinglist

- Topic: Discussed in first lecture and recitation(s)

- Timeline:
  - Oct 12th: Announce project teams to TA
  - Oct 19th: Present your project in recitation
  - Nov 6th: Initial progress presentations during class
  - Last class (Dec 18th): Final project presentations

- Report:
  - 6 pages, template provided on webpage, due January
Peer Quiz

Instructions:
- Pick some partners (locally) and discuss each question for 2 minutes
- We then select a random student (team) to answer the question

What are the problems with sequential consistency?
- Is it practical? Explain!
- Is it sufficient for simple parallel programming? Explain!
- How would you improve the situation?

How could memory models of practical CPUs be described?
- Is Intel’s definition useful?
- Why would one need a better definition?
- Threads cannot be implemented as a library? Why does Pthreads work?
DPHPC Overview

concepts & techniques
- locality
  - caches
  - memory hierarchy
- parallelism
  - vector ISA
  - shared memory
  - distributed memory
- cache coherency
- memory models
  - locks
  - lock free
  - wait free
  - linearizability

models
- memory
  - $\alpha - \beta$
- Amdahl's and Gustafson's law
  - PRAM
  - LogP
- I/O complexity
- balance principles I
- Little's Law
- balance principles II
- scheduling
Goals of this lecture

■ Queue:
  ▪ Problems with the locked queue
  ▪ Wait-free two-thread queue

■ Linearizability
  ▪ Intuitive understanding (sequential order on objects!)
  ▪ Linearization points
  ▪ Linearizable executions
  ▪ Formal definitions (Histories, Projections, Precedence)

■ Linearizability vs. Sequential Consistency
  ▪ Modularity

■ Maybe: lock implementations
Lock-based queue

class Queue {
    private:
        int head, tail;
        std::vector<Item> items;
        std::mutex lock;
    
    public:
        Queue(int capacity) {
            head = tail = 0;
            items.resize(capacity);
        }
        ...
};

Queue fields protected by single shared lock!
Lock-based queue

```cpp
class Queue {
    ...

public:
    void enq(Item x) {
        std::lock_guard<std::mutex> l(lock);
        if((tail+1)%items.size()==head) {
            throw FullException;
        }
        items[tail] = x;
        tail = (tail+1)%items.size();
    }

    Item deq() {
        std::lock_guard<std::mutex> l(lock);
        if(tail == head) {
            throw EmptyException;
        }
        Item item = items[head];
        head = (head+1)%items.size();
        return item;
    }
};
```

Queue fields protected by single shared lock!

Class question: how is the lock ever unlocked?
C++ Resource Acquisition is Initialization

- RAlI – suboptimal name

- Can be used for locks (or any other resource acquisition)
  - Constructor grabs resource
  - Destructor frees resource

- Behaves as if
  - Implicit unlock at end of block!

- Main advantages
  - Always unlock/free lock at exit
  - No “lost” locks due to exceptions or strange control flow (goto 😐)
  - Very easy to use

```cpp
class lock_guard<typename mutex_impl> {
    mutex_impl & _mtx; // ref to the mutex

    public:
        scoped_lock(mutex_impl & mtx) : _mtx(mtx) {
            _mtx.lock(); // lock mutex in constructor
        }

        ~scoped_lock() {
            _mtx.unlock(); // unlock mutex in destructor
        }
};
```
Example execution

A: q.deq(): x

lock

update q

unlock

update q

lock

B: q.enq(x)

lock

update q

unlock

update q

“sequential behavior”
Correctness

- **Is the locked queue correct?**
  - Yes, only one thread has access if locked correctly
  - Allows us again to reason about pre- and postconditions
  - Smells a bit like sequential consistency, no?

- **Class question: What is the problem with this approach?**
  - Same as for SC 😊

It does not scale!
What is the solution here?
Threads working at the same time?

- Same thing (concurrent queue)
- For simplicity, assume only two threads
  - Thread A calls only `enq()`
  - Thread B calls only `deq()`
Wait-free 2-Thread Queue

Diagram showing a circular queue with pointers to head and tail. The diagram includes labels for `deq()` and `enq(z)`. The queue is depicted with a circular structure, and the numbers 0 to 7 are placed in the sections of the circle, with `x` and `y` indicating intermediate positions.
Wait-free 2-Thread Queue

result = x

queue[tail] = z
Wait-free 2-Thread Queue

head++

tail++
Is this correct?

- Hard to reason about correctness
- What could go wrong?

```cpp
void enq(Item x) {
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

Item deq() {
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
}
```

- Nothing (at least no crash)
- Yet, the **semantics** of the queue are funny (define “FIFO” now)!
Serial to Concurrent Specifications

- Serial specifications are complex enough, so let's stick to them
  - Define invocation and response events (start and end of method)
  - Extend the sequential concept to concurrency: linearizability

- Each method should "take effect"
  - Instantaneously
  - Between invocation and response events

- A concurrent object is correct if its "sequential" behavior is correct
  - Called "linearizable"

Linearization point = when method takes effect
Linearizability

- Sounds like a property of an execution ...
- An object is called linearizable if all possible executions on the object are linearizable
- Says nothing about the order of executions!
Example

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock)
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

Item deq() {
    std::lock_guard<std::mutex> l(lock)
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
}
```

linearization points
Example

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

Item deq() {
    std::lock_guard<std::mutex> l(lock);
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
}
```

linearization points
Example

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock)
    if((tail+1) % items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1) % items.size();
}

Item deq() {
    std::lock_guard<std::mutex> l(lock)
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1) % items.size();
}
```

linearization points
Example

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

Item deq() {
    std::lock_guard<std::mutex> l(lock);
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
}
```

linearization points
Example

```
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

```
Item deq() {
    std::lock_guard<std::mutex> l(lock);
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
}
```

linearization points
Example

```
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

Item deq() {
    std::lock_guard<std::mutex> l(lock);
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
}
```

linearization points

```
| q.enq(x) | q.enq(y) | q.deq(x) | q.deq(y) |
```

linearizable

time
Example

```
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock)
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

Item deq() {
    std::lock_guard<std::mutex> l(lock)
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
}
```

linearization points

```
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock)
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}

Item deq() {
    std::lock_guard<std::mutex> l(lock)
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
}
Example 2

gg
Example 2

\[ \text{q.enq}(x) \]
Example 2

```
q.enq(x)
```

```
q.deq(y)
```

```
time
```
Example 2

q.enq(x) → q.deq(y) → q.enq(y)

Time

q.enq(x)
Example 2

q.enq(x)
q.enq(y)
q.deq(y)
q.enq(x)
q.enq(y)

time
Example 2

```
q.enq(x)
qu.enq(y)
qu.deq(y)
qu.enq(x)
qu.enq(y)
```

Not linearizable
Example 3
Example 3

\[q.\text{enq}(x)\]

Time

Diagram showing a queue with elements and the time line.
Example 3

```plaintext
| q.enq(x) | time | q.deq(x) |
```

- `q.enq(x)`: Enqueue operation
- `q.deq(x)`: Dequeue operation
Example 3

q.enq(x)
q.deq(x)
q.enq(x)
q.deq(x)
Example 3

- q.enq(x)
- q.deq(x)

linearizable
Example 4

q.enq(x)
Example 4

```
Example 4
```

```
q.enq(x)
```

```
q.enq(y)
```

```
38
```

```
time
```

```
  [ ]
  [ ]
  [x]
```
Example 4

q.enq(x)  q.enq(y)  q.deq(y)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

time
Example 4

```
q.enq(x)
q.enq(y)
q.deq(y)
q.deq(x)
```

```
time
```
Example 4

multiple orders OK linearizable

time
Is the lock-free queue linearizable?

A) Only two threads, one calls only \texttt{deq()} and one calls only \texttt{enq()}?

(assume each source-line is itself linearizable)

```java
void enq(Item x) {
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

```java
Item deq() {
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
}
```

B) Only two threads but both may call \texttt{enq()} or \texttt{deq()} independently

C) An arbitrary number of threads, but only one calls \texttt{enq()}

D) An arbitrary number of threads can call \texttt{enq()} or \texttt{deq()}

E) If it’s linearizable, where are the linearization points?

- Remark: typically executions are not constrained, so this is NOT linearizable
Read/Write Register Example

- Assume atomic update to a single read/write register!
Read/Write Register Example

- Assume atomic update to a single read/write register!
Read/Write Register Example

- Assume atomic update to a single read/write register!
Read/Write Register Example

- Assume atomic update to a single read/write register!

![Diagram showing read/write operations with notes on atomic update.]
Read/Write Register Example

- Assume atomic update to a single read/write register!

write(0) → read(1) → write(1) → write(2) → read(1)
Read/Write Register Example

- Assume atomic update to a single read/write register!

write(0)  read(1)  write(1)  write(2)  read(1)

write(1) already happened
Read/Write Register Example

- Assume atomic update to a single read/write register!

![Diagram showing read/write operations over time](image-url)
Read/Write Register Example

- Assume atomic update to a single read/write register!
Read/Write Register Example

- Assume atomic update to a single read/write register!
Read/Write Register Example

write(0) → read(1) → write(2) → write(1) → read(2)

time
Read/Write Register Example

write(0) → read(1) → write(2) → write(1) → read(2)

time
Read/Write Register Example

- write(0)
- read(1)
- write(1)
- write(2)
- write(1)
- read(2)

Time
Read/Write Register Example

- write(0)
- read(1)
- write(2)
- write(1)
- read(2)

Time

Linearizable
About Executions

- **Class-question: Why is this all defined for executions?**
  - Can’t we specify the linearization point of each operation statically (based on the program text) without describing an execution?

- **Not always**
  - In some cases, the linearization point depends on the execution
    
    ***Imagine a “check if one should lock” (not recommended)!***

- **Define a formal model for executions!**
Properties of concurrent method executions

- Method executions take time
  - May overlap

- Method execution = operation
  - Defined by invocation and response events

- Duration of method call
  - Interval between the events
Formalization - Notation

- Invocation

  A: q.enq(x)

  thread  object  method  arguments

- Response

  A: q:void

  thread  object  result

  A: q:FullException()

  thread  object  exception

- Question: why is the method name not needed in the response?

  *Method is implicit (correctness criterion)!*
Concurrency

- A concurrent system consists of a collection of sequential threads $P_i$.
- Threads communicate via shared objects.

*For now!*
History

- Describes an execution
  - Sequence of invocations and responses
  - $H =$
    - A: q.enq(a)
    - A: q:void
    - A: q.enq(b)
    - B: p.enq(c)
    - B: p:void
    - B: q.deq()
    - B: q:a

Invocation and response match if
- thread names are the same
- objects are the same

Remember: Method name is implicit!

Side Question: Is this history linearizable?
Projections on Threads

- Threads subhistory $H|P$ (“$H$ at $P$”)
  - Subsequences of all events in $H$ whose thread name is $P$

$H =$
- $A$: q.enq(a)
- $A$: q:void
- $A$: q.enq(b)
- $B$: p.enq(c)
- $B$: p:void
- $B$: q.deq()
- $B$: q:a

$H|A =$
- $A$: q.enq(a)
- $A$: q:void
- $A$: q.enq(b)

$H|B =$
- $B$: p.enq(c)
- $B$: p:void
- $B$: q.deq()
Projections on Objects

- Objects subhistory $H|o$ ("H at o")
  - Subsequence of all events in $H$ whose object name is $o$

$H = \begin{align*}
A: & \text{q.enq(a)} \\
A: & \text{q:void} \\
A: & \text{q.enq(b)} \\
B: & \text{p.enq(c)} \\
B: & \text{p:void} \\
B: & \text{q.deq()} \\
B: & \text{q:a}
\end{align*}$

$H|p = \begin{align*}
B: & \text{p.enq(c)} \\
B: & \text{p:void}
\end{align*}$

$H|q = \begin{align*}
A: & \text{q.enq(a)} \\
A: & \text{q:void} \\
A: & \text{q.enq(b)} \\
B: & \text{q.deq()} \\
B: & \text{q:a}
\end{align*}$
Sequential Histories

- A history $H$ is sequential if
  - The first event of $H$ is an invocation
  - Each invocation (except possibly the last) is immediately followed by a matching response
  - Each response is immediately followed by an invocation

- Method calls of different threads do not interleave

- A history $H$ is concurrent if
  - It is not sequential
Well-formed histories

- Per-thread projections must be sequential

A history is sequential if:

- The first event of H is an invocation
- Each invocation (except possibly the last) is immediately followed by a matching response
- Each response is immediately followed by an invocation

H =

A: q.enq(x)
B: p.enq(y)
B: p:void
B: q.deq()
A: q:void
B: q:x

H | A =

A: q.enq(x)
A: q:void

H | B =

B: p.enq(y)
B: p:void
B: q.deq()
B: q:x
Equivalent histories

- Per-thread projections must be the same

|       | H=                          | G=                          | H|A=G|A=                      |
|-------|-----------------------------|-----------------------------|-----------------------------|
| A: q.enq(x) | B: p.enq(y)               | A: q.enq(x)                 | A: q.enq(x)                 |
| B: q.deq()      | B: q.deq()              | B: q.deq()                  | B: q.deq()                  |
| A: q: void       | B: q: x               | A: q: void                  | B: q: x                     |
| B: q: x           |                         |                            |                            |
Legal Histories

- Sequential specification allows to describe what behavior we expect and tolerate
  - When is a single-thread, single-object history legal?

- Recall: Example
  - Preconditions and Postconditions
  - Many others exist!

- A sequential (multi-object) history $H$ is legal if
  - For every object $x$
  - $H|x$ adheres to the sequential specification for $x$

- Example: FIFO queue
  - Correct internal state
    - *Order of removal equals order of addition*
  - Full and Empty Exceptions
Precedence

A: q.enq(x)
B: q.enq(y)
B: q: void
A: q: void
B: q.deq()
B: q:x

A method execution **precedes** another if response event precedes invocation event
Precedence vs. Overlapping

- Non-precedence = overlapping

Some method executions overlap with others

A: q.enq(x)
B: q.enq(y)
B: q:void
A: q:void
B: q.deq()
B: q:x

Side Question: Is this a correct linearization order?
Complete Histories

- A history $H$ is complete
  - If all invocations are matched with a response

$H = \begin{align*}
A: & \text{ q.enq(x)} \\
B: & \text{ p.enq(y)} \\
B: & \text{ p:void} \\
B: & \text{ q.deq()} \\
A: & \text{ q:void} \\
B: & \text{ q:x} 
\end{align*}$

$G = \begin{align*}
A: & \text{ q.enq(x)} \\
B: & \text{ p.enq(y)} \\
B: & \text{ p:void} \\
B: & \text{ q.deq()} \\
A: & \text{ q:void} \\
A: & \text{ q.enq(z)} \\
B: & \text{ q:x} 
\end{align*}$

$I = \begin{align*}
A: & \text{ q.enq(x)} \\
B: & \text{ p.enq(y)} \\
B: & \text{ p:void} \\
B: & \text{ q.deq()} \\
B: & \text{ q.deq()} \\
A: & \text{ q:void} \\
B: & \text{ q:x} 
\end{align*}$

Which histories are complete and which are not?
Precedence Relations

■ Given history H

■ Method executions $m_0$ and $m_1$ in H
  ▪ $m_0 \rightarrow_H m_1$ ($m_0$ precedes $m_1$ in H) if
  ▪ Response event of $m_0$ precedes invocation event of $m_1$

■ Precedence relation $m_0 \rightarrow_H m_1$ is a
  ▪ Strict partial order on method executions
    * Irreflexive, antisymmetric, transitive

■ Considerations
  ▪ Precedence forms a total order if H is sequential
  ▪ Unrelated method calls $\rightarrow$ may overlap $\rightarrow$ concurrent
Definition Linearizability

- A history $H$ induces a strict partial order $<_H$ on operations
  - $m_0 <_H m_1$ if $m_0 \rightarrow_H m_1$

- A history $H$ is **linearizable** if
  - $H$ can be extended to a complete history $H'$ by appending responses to pending operations or dropping pending operations
  - $H'$ is equivalent to some legal sequential history $S$ and
  - $<_H \subseteq <_S$

- $S$ is a **linearization** of $H$

- Remarks:
  - For each $H$, there may be many valid extensions to $H'$
  - For each extension $H'$, there may be many $S$
  - Interleaving at the granularity of methods
Ensuring $\prec_{H'} \subseteq \prec_S$

- Find an $S$ that contains $H'$

$\prec_{H'} = \{a \rightarrow c, b \rightarrow c\}$

$\prec_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$

$S$ respects the “real time” order of $H'$
Example

A. \texttt{q.enq(3)}
B. \texttt{q.enq(4)}
B. \texttt{q: void}
B. \texttt{q.deq()}
B. \texttt{q: 4}
B. \texttt{q: enq(6)}
Example

A \texttt{q.enq(3)}
B \texttt{q.enq(4)}
B \texttt{q: void}
B \texttt{q.deq()}
B \texttt{q: 4}
B \texttt{q.enq(6)}

Complete this pending invocation

A. \texttt{q.enq(3)}
B. \texttt{q.enq(4)}
B. \texttt{q.deq(): 4}
B. \texttt{q.enq(6)}

\textbf{time}
Example

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q: 4
B q: enq(6)
A q: void

Complete this pending invocation

A.q.enq(3)
B.q.enq(4)
B.q.deq(): 4
B. q.enq(6)

time
Example

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q: 4
B q: enq(6)
A q: void

class disc this one
Example

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

discard this one

A.q.enq(3)
B.q.enq(4)
B.q.deq(): 4

time
Example

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

What would be an equivalent sequential history?
Example

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

B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4
Example

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

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

Equivalent sequential history
Remember: Linearization Points

- Identify one atomic step where a method “happens” (effects become visible to others)
  - Critical section
  - Machine instruction (atomics, transactional memory, …)

- Does not always succeed
  - One may need to define several different steps for a given method
  - If so, extreme care must be taken to ensure pre-/postconditions

- All possible executions on the object must be linearizable

```cpp
void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size() == head) {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
}
```

```cpp
Item deq() {
    std::lock_guard<std::mutex> l(lock);
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
}
```
Composition

- H is linearizable iff for every object x, H|x is linearizable!
  - Corollary: Composing linearizable objects results in a linearizable system

- Reasoning
  - Consider linearizability of objects in isolation

- Modularity
  - Allows concurrent systems to be constructed in a modular fashion
  - Compose independently-implemented objects
Linearizability vs. Sequential Consistency

- **Sequential consistency**
  - Correctness condition
  - For describing hardware memory interfaces
  - Remember: not *actual* ones!

- **Linearizability**
  - Stronger correctness condition
  - For describing higher-level systems composed from linearizable components
    
    Requires understanding of object semantics
Map linearizability to sequential consistency

- **Variables with read and write operations**
  - Sequential consistency

- **Objects with a type and methods**
  - Linearizability

- **Map sequential consistency \(\leftrightarrow\) linearizability**
  - \(\leftarrow\) Reduce data types to variables with read and write operations
  - \(\rightarrow\) Model variables as data types with read() and write() methods

- **Remember: Sequential consistency**
  - A history \(H\) is sequential if it can be extended to \(H'\) and \(H'\) is equivalent to some sequential history \(S\)
  - Note: Precedence order \((<_H \subseteq <_S)\) does not need to be maintained
Example
Example

q.enq(x)

time
Example

```
q.enq(x)
q.deq(y)
```
Example

Linearizable?

q.enq(x)

q.deq(y)

q.enq(y)

time
Example

Linearizable?

```
q.enq(x)
q.enq(y)
q.deq(y)
q.enq(x)
q.enq(y)
```
Example

Linearizable?

not linearizable

\[ \text{q.enq(x)} \]
\[ \text{q.enq(y)} \]
\[ \text{q.deq(y)} \]

\[ \text{q.enq(x)} \]
\[ \text{q.enq(y)} \]
Example

Sequentially consistent?

\[ q.enq(x) \]
\[ q.enq(y) \]
\[ q.deq(y) \]
\[ q.enq(x) \]
\[ q.enq(y) \]
Example

Sequentially consistent?

q.enq(x)
q.enq(y)
q.deq(y)
q.enq(x)
q.enq(y)

Yet sequentially consistent
Properties of sequential consistency

- Theorem: Sequential consistency is not compositional

H =

A: p.enq(x)
A: p:void
B: q.enq(y)
B: q:void
A: q.enq(x)
A: q:void
B: p.enq(y)
B: p:void
A: p.deq()
A: p:y
B: q.deq()
B: q:x

Compositional would mean:
“If H|p and H|q are sequentially consistent, then H is sequentially consistent!”

This is not guaranteed for SC schedules!

See following example!
FIFO Queue Example

\[ p.\text{enq}(x) \quad q.\text{enq}(x) \quad p.\text{deq}(y) \]

\text{time}
FIFO Queue Example

p.enq(x)  q.enq(x)  p.deq(y)  q.enq(y)  p.enq(y)  q.deq(x)

time
FIFO Queue Example

History $H$
H|=p Sequentially Consistent

\[\begin{align*}
\text{p.enq}(x) & \quad \text{q.enq}(x) & \quad \text{p.deq}(y) \\
\text{q.enq}(y) & \quad \text{p.enq}(y) & \quad \text{q.deq}(x)
\end{align*}\]
H|q Sequentially Consistent

p.enq(x) q.enq(x) p.deq(y) q.enq(y) p.enq(y) q.deq(x)

time
Ordering imposed by p (linearizability)
Ordering imposed by q (linearizability)
Ordering imposed by both

- p.enq(x)
- q.enq(x)
- p.deq(y)
- q.enq(y)
- p.enq(y)
- q.deq(x)

Time
Combining orders
Example in our notation

Sequential consistency is not compositional – $H|p$

$H =$

A: p.enq(x)
A: p:void
B: q.enq(y)
B: q:void
A: q.enq(x)
A: q:void
B: p.enq(y)
B: p:void
A: p.deq()
A: p:y
B: q.deq()
B: q:x

$H|p =$

A: p.enq(x)
A: p:void
B: p.enq(y)
B: p:void
A: p.deq()
A: p:y

$(H|p)|A =$

A: p.enq(x)
A: p:void
A: p.deq()
A: p:y

$(H|p)|B =$

B: p.enq(y)
B: p:void

$H|p$ is sequentially consistent!
Example in our notation

- Sequential consistency is not compositional – H|q

H=
A: p.enq(x)
A: p: void
B: q.enq(y)
B: q: void
A: q.enq(x)
A: q: void
B: p.enq(y)
B: p: void
A: p.deq()
A: p: y
B: q.deq()
B: q: x

H|q=
B: q.enq(y)
B: q: void
A: q.enq(x)
A: q: void
B: q.deq()
B: q: x

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

(H|q)|B=
B: q.enq(y)
B: q: void
B: q.deq()
B: q: x

H|q is sequentially consistent!
Example in our notation

- Sequential consistency is not compositional

<table>
<thead>
<tr>
<th>H=</th>
<th>H</th>
<th>A=</th>
<th>H</th>
<th>B=</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: p.enq(x)</td>
<td>A: p.enq(x)</td>
<td>A: p.enq(y)</td>
<td>B: q.enq(y)</td>
<td></td>
</tr>
<tr>
<td>A: p:void</td>
<td>A: p:void</td>
<td>A: q.enq(x)</td>
<td>B: q:void</td>
<td></td>
</tr>
<tr>
<td>B: q.enq(y)</td>
<td>A: q.enq(void)</td>
<td>A: q.enq(x)</td>
<td>B: p.enq(y)</td>
<td></td>
</tr>
<tr>
<td>B: p.enq(y)</td>
<td>A: p:empty</td>
<td>B: q.deq()</td>
<td>B: q:x</td>
<td></td>
</tr>
<tr>
<td>B: p:empty</td>
<td>A: p:empty</td>
<td>B: q:deq()</td>
<td>B: q:x</td>
<td></td>
</tr>
</tbody>
</table>

H is not sequentially consistent!
Correctness: Linearizability

- **Sequential Consistency**
  - Not composable
  - Harder to work with
  - Good (simple) way to think about hardware models
    - *Few assumptions (no semantics or time)*

- **We will use** *linearizability* **in the remainder of this course unless stated otherwise**
  - *Consider routine entry and exit*
Study Goals (Homework)

- Define linearizability with your own words!
- Describe the properties of linearizability!
- Explain the differences between sequential consistency and linearizability!

- Given a history H
  - Identify linearization points
  - Find equivalent sequential history S
  - Decide and explain whether H is linearizable
  - Decide and explain whether H is sequentially consistent
  - Give values for the response events such that the execution is linearizable
Language Memory Models

- Which transformations/reorderings can be applied to a program

- Affects platform/system
  - Compiler, (VM), hardware

- Affects programmer
  - What are possible semantics/output
  - Which communication between threads is legal?

- Without memory model
  - Impossible to even define “legal” or “semantics” when data is accessed concurrently

- A memory model is a contract
  - Between platform and programmer
History of Memory Models

- **Java's original memory model was broken [1]**
  - Difficult to understand => widely violated
  - Did not allow reorderings as implemented in standard VMs
  - Final fields could appear to change value without synchronization
  - Volatile writes could be reordered with normal reads and writes
    => counter-intuitive for most developers

- **Java memory model was revised [2]**
  - Java 1.5 (JSR-133)
  - Still some issues (operational semantics definition [3])

- **C/C++ didn’t even have a memory model until recently**
  - Not able to make any statement about threaded semantics!
  - Introduced in C++11 and C11
  - Based on experience from Java, more conservative

Everybody wants to optimize

- **Language constructs for synchronization**
  - Java: volatile, synchronized, ...
  - C++: atomic, *(NOT volatile!)*, mutex, ...

- **Without synchronization (defined language-specific)**
  - Compiler, (VM), architecture
  - Reorder and appear to reorder memory operations
  - Maintain *sequential semantics* per thread
  - Other threads may observe any order (have seen examples before)
Java and C++ High-level overview

- **Relaxed memory model**
  - No global visibility ordering of operations
  - Allows for standard compiler optimizations

- **But**
  - Program order for each thread (sequential semantics)
  - Partial order on memory operations (with respect to synchronizations)
  - Visibility function defined

- **Correctly synchronized programs**
  - Guarantee sequential consistency

- **Incorrectly synchronized programs**
  - Java: maintain safety and security guarantees
    - *Type safety etc. (require behavior bounded by causality)*
  - C++: undefined behavior
    - *No safety (anything can happen/change)*
Communication between threads: Intuition

- **Not guaranteed unless by:**
  - Synchronization
  - Volatile/atomic variables
  - Specialized functions/classes (e.g., `java.util.concurrent`, ...)

---

Flag is a synchronization variable (atomic in C++, volatile in Java), i.e., all memory written by T1 must be visible to T2 after it reads the value true for `flag`!
Recap: Memory Model (Intuition)

- Abstract relation between threads and memory
  - Local thread view!

- Does not talk about classes, objects, methods, ...
  - Linearizability is a higher-level concept!
Lock synchronization

- **Java**
  
  ```java
  synchronized (lock) {
   // critical region
  }
  
  - Synchronized methods as syntactic sugar
  
- **C++ (RAII)**
  
  ```cpp
  {
   unique_lock<mutex> l(lock);
   // critical region
  }
  
  - Many flexible variants

- **Semantics:**
  
  - mutual exclusion
  - at most one thread may own a lock
  - a thread B trying to acquire a lock held by thread A blocks until thread A releases lock
  - note: threads may wait forever (no progress guarantee!)
Memory semantics

- Similar to synchronization variables

Thread 1

```
x = 10
...
y = 5
...
unlock(m)
```

Thread 2

```
lock(m)
print(x+y)
```

- All memory accesses **before** an unlock ...
- are ordered before and are visible to ...
- any memory access **after** a matching lock!
Synchronization variables

- Variables can be declared volatile (Java) or atomic (C++)

- Reads and writes to synchronization variables
  - Are totally ordered with respect to all threads
  - Must not be reordered with normal reads and writes

- Compiler
  - Must not allocate synchronization variables in registers
  - Must not swap variables with synchronization variables
  - May need to issue memory fences/barriers
  - ...

Synchronization variables

- **Write to a synchronization variable**
  - Similar memory semantics as unlock (no process synchronization!)

- **Read from a synchronization variable**
  - Similar memory semantics as lock (no process synchronization!)

```java
class example {
    int x = 0;
    atomic<bool> v = false

    public void writer() {
        x = 42;
        v = true;
    }

    public void reader() {
        if(v) {
            print(x)
        }
    }
}
```

Without atomic or volatile, a platform may reorder these accesses!
Intuitive memory model rules

- Java/C++: Correctly synchronized programs will execute sequentially consistent

- Correctly synchronized = data-race free
  - iff all sequentially consistent executions are free of data races

- Two accesses to a shared memory location form a data race in the execution of a program if
  - The two accesses are from different threads
  - At least one access is a write and
  - The accesses are not synchronized

```
int x = 10
T1  read
T2  write
T3  read
```