Design of Parallel and High-Performance Computing

Lecture: Linearizability

Instructor: Torsten Hoefler & Markus Püschel
TAs: Salvatore Di Girolamo

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

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 mailing list
- 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

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!

C++ Resource Acquisition is Initialization

- RAII – 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

Example execution

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

"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

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()

It does not scale!
What is the solution here?
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"

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!

Is this correct?

- Hard to reason about correctness
- What could go wrong?
  - Nothing (at least no crash)
  - Yet, the **semantics** of the queue are funny (define “FIFO” now)!

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

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

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

Linearization points

Example

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();
}
```

```cpp
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

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

Example 2

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

Example 2

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

Example 2

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

Example 2

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

Example 2

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

Example 2

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

Example 2

```cpp
q.enq(x)
q.deq(y)
```
Example 2

Example 3

Example 3

Example 3
Example 4

q.enq(x)

time

Example 4

q.enq(x)

q.enq(y)

time

Example 4

q.enq(x)

q.enq(y)

q.deq(y)

q.deq(y)

q.deq(x)

Example 4

q.enq(x)

q.enq(y)

q.deq(y)

q.deq(x)

Is the lock-free queue linearizable?

- A) Only two threads, one calls only deq() and one calls only enq()? (assume each source line is itself linearizable)
- B) Only two threads but both may call enq() or deq() independently
- C) An arbitrary number of threads, but only one calls enq()
- D) An arbitrary number of threads can call enq() or deq()
- E) If it’s linearizable, where are the linearization points?
  - Remark: typically executions are not constrained, so this is NOT linearizable

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

```c
Item deq() {
    if(tail == head) {
        throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
}
```
Read/Write Register Example
- Assume atomic update to a single read/write register!

Write(0)  |  Read(1)  |  Write(2)
---------|-----------|---------
Write(1)  |       | Read(0)

Time

Write(0)  |  Read(1)  |  Write(2)
---------|-----------|---------
Write(1)  |       | Write(1) already happened
Write(2)  |       | 

Read(0)

Write(1) already happened

Write(1) already happened

Not linearizable

Write(0)  |  Read(1)  |  Write(2)
---------|-----------|---------
Write(1)  |       | Write(1) already happened
Write(2)  |       | 

Read(1)

Write(1) already happened

Not linearizable
Read/Write Register Example

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

\[ \text{write}(0) \rightarrow \text{write}(1) \rightarrow \text{write}(2) \]

\[ \text{read}(1) \]

\[ \text{linearizable} \]
Read/Write Register Example

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 \rightarrow object \rightarrow method \rightarrow arguments
- Response
  - A: q: void
  - A: q: FullException()
  - 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
  - His:
    - 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 $P$

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

Projections on Objects

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

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

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

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 method execution precedes another if response event precedes invocation event.

Precedence vs. Overlapping

- Non-precedence = overlapping

Some method executions overlap with others.

Side Question: Is this a correct linearization order?

Complete Histories

- A history H is complete if all invocations are matched with a response.

Precedence Relations

- Given history H
- Method executions $m_0$ and $m_1$ in H
  - $m_0 \rightarrow H m_1$ (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 may overlap

Definition Linearizability

- A history H induces a strict partial order $<_H$ on operations
- 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 $<_H \subseteq <_S$

- Find an S that contains $H'$
  $$<_H = \{a \rightarrow c, b \rightarrow c\}$$
  $$<_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$$

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

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

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

Example

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

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
```

Example

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

Example

```
A q.enq(3)
B q.enq(4)
B 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
```

Example

```
A q.enq(3)
B q.enq(4)
B 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
```

Example

```
A q.enq(3)
B q.enq(4)
B 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
```

Example

```
A q.enq(3)
B q.enq(4)
B 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
```

Example

```
A q.enq(3)
B q.enq(4)
B 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
```

Example

```
A q.enq(3)
B q.enq(4)
B 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
```

Example

```
A q.enq(3)
B q.enq(4)
B 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
```

Example

```
A q.enq(3)
B q.enq(4)
B 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
```

Example
Example

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

Example

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

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

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;
}

Composition

- H is linearizable iff for every object x, H|x is linearizable!
  - Corollary: Composing linearizable objects results in a linearizable system
- Reasoning
  - Consider linearity 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 ↔ linearizability
  - Reduce data types to variables with read and write operations
  - 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 (<, ≤, <) does not need to be maintained
Example

Sequentially consistent?

Properties of sequential consistency

- Theorem: Sequential consistency is not compositional

H=

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

History \( H \)
H\|p Sequentially Consistent

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

time

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)

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

time

Ordering imposed by q (linearizability)

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

time

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

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

time
Sequential consistency is not compositional

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

Example in our notation
- Sequential consistency is not compositional – \( H|p \)
- Sequential consistency is not compositional – \( H|q \)

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 \textit{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!

H |p is sequentially consistent!

H |q is sequentially consistent!

H is not sequentially consistent!
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, ...)

---

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
  - Synchronized methods as syntactic sugar

- C++ (RAII)
  - 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
  
  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

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

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

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

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