Design of Parallel and High-Performance Computing

Fall 2014

Lecture: Memory Models

Motivational video: https://www.youtube.com/watch?v=tW2hT0g4OUs

Instructor: Torsten Hoefler & Markus Püschel

TA: Timo Schneider

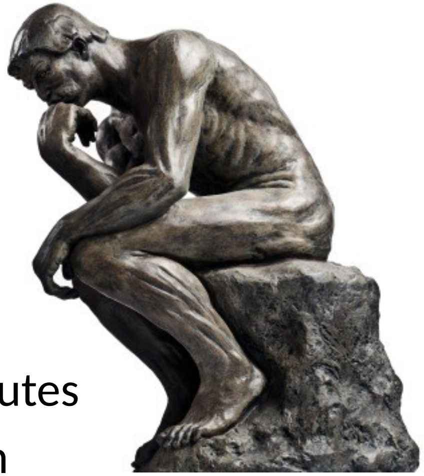
ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Review of last lecture

- **Architecture case studies**
  - Memory performance is often the bottleneck
  - Parallelism grows with compute performance
  - Caching is important
  - Several issues to address for parallel systems
- **Cache Coherence**
  - Hardware support to aid programmers
  - Two guarantees:
    - *Write propagation (updates are eventually visible to all readers)*
    - *Write serialization (writes to the same location are observed in order)*
  - Two major mechanisms:
    - *Snooping*
    - *Directory-based*
  - Protocols: MESI (MOESI, MESIF)

Peer Quiz

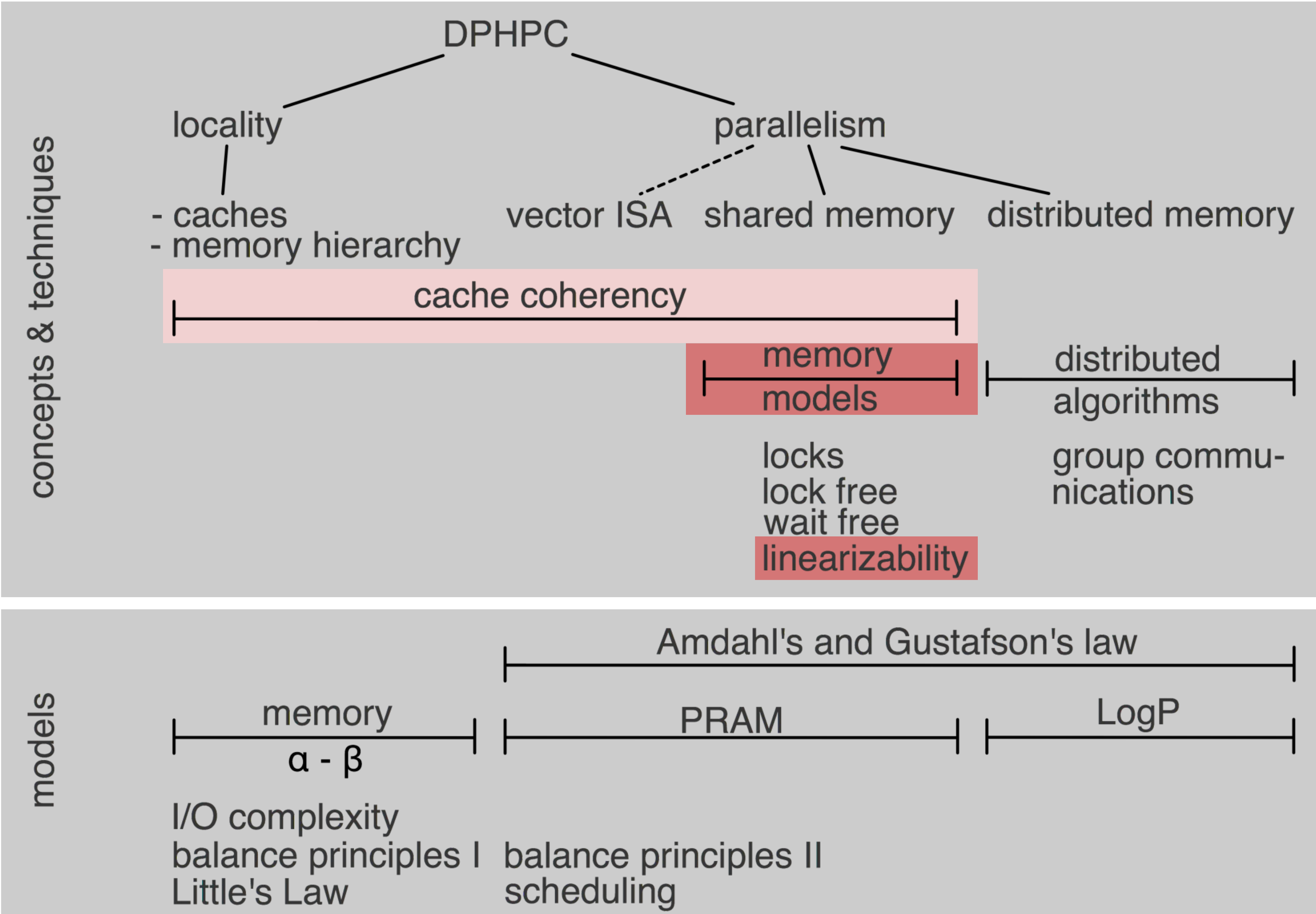


- **Instructions:**
  - Pick some partners (locally) and discuss each question for 2 minutes
  - We then select a random student (team) to answer the question

- **Discuss the MESI protocol – what would be a possible extension to improve it’s performance**
  - Try something we didn’t discuss last week ^^
  - Argue why is it an improvement!

- **Directory-based Cache Coherence?**
  - What are the pros/cons of directory-based CC?
  - Can this be mixed with broadcast-based?
  - If yes, how and why?

DPHPC Overview



Goals of this lecture

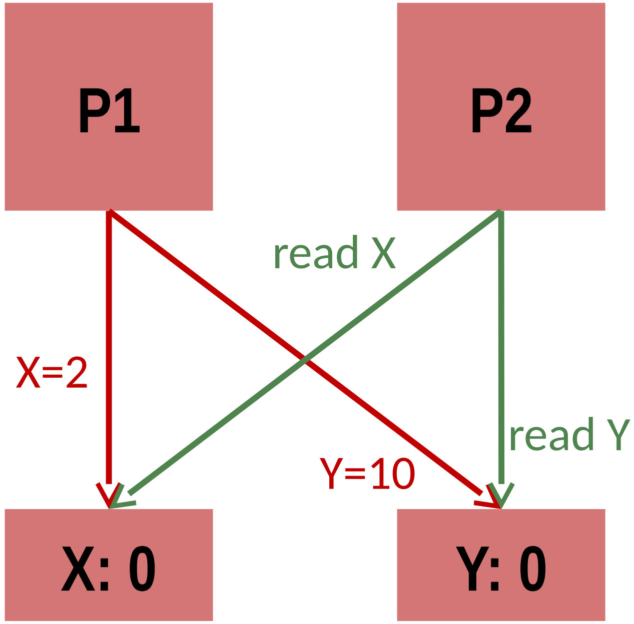
- **Don’t forget the projects!**
  - Groups to be defined by end of this week (send email to Timo)
  - Project progress presentations on 11/2 (<1 month from now)!
- **Cache-coherence is not enough!**
  - Many more subtle issues for parallel programs!
- **Memory Models**
  - Sequential consistency
  - Why threads cannot be implemented as a library ㅆ
  - Relaxed consistency models
- **Linearizability**
  - More complex objects

Is coherence everything?

- **Coherence is concerned with behavior of *individual* locations**
- **Consider the program (initial X,Y,Z = 0)**

P1	**P2**
Y=10 X=2	while (X==0); Z=Y

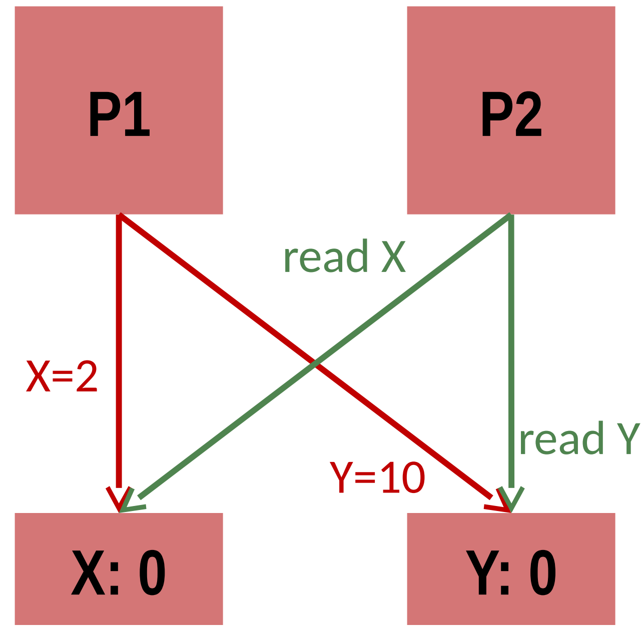
  - Class question: what value will Z on P2 have?



Is coherence everything?

- **Coherence is concerned with behavior of *individual* locations**
- **Consider the program (initial X,Y,Z = 0)**

P1	P2
Y=10	while (X==0)
X=2	Z=Y



- **Y=10 does not need to have completed before X=2 is visible to P2!**
  - This allows P2 to exit the loop and read Y=0
  - This may not be the intent of the programmer!
  - This may be due to congestion (imagine X is pushed to a remote cache while Y misses to main memory) and or due to write buffering, or …
- **Bonus class question: what happens when Y and X are on the same cache line (assume simple MESI)?**

Memory Models

- **Need to define what it means to "read a location" and "to write a location" and the respective ordering!**
  - What values should be seen by a processor
- **First thought: extend the abstractions seen by a sequential processor:**
  - **Compiler** and **hardware** maintain data and control dependencies at all levels:

Two operations to the same location

```
Y=10
....
T = 14
Y=15
```

One operation controls execution of others

```
Y = 5
X = 5
T = 3
Y = 3
If (X==Y)
  Z = 5
....
```

Sequential Processor

- **Correctness condition:**
  - The result of the execution is the same as if the operations had been executed in the order specified by the program

    "program order"
  - A read returns the value last written to the same location

    "last" is determined by program order!
- **Consider only memory operations (e.g., a trace)**
- **N Processors**
  - P1, P2, …., PN
- **Operations**
  - Read, Write on shared variables (initial state: all 0)
- **Notation:**
  - P1: R(x):3 P1 reads x and observes the value 3
  - P2: W(x,5) P2 writes 5 to variable x

Terminology

- **Program order**
  - Deals with a *single* processor
  - Per-processor order of memory accesses, determined by program 's

    Control flow
  - Often represented as trace

- **Visibility order**
  - Deals with operations on *all* processors
  - Order of memory accesses observed by one or more processors
  - E.g., "every read of a memory location returns the value that was written last"

    Defined by memory model

Memory Models

- Contract at each level between programmer and processor

Programmer

High-level language API

Compiler Frontend — Optimizing transformations

Intermediate Language

Compiler Backend/JIT — Reordering

Machine code

Processor — Operation overlap, OOO Execution, VLIW, Vector ISA

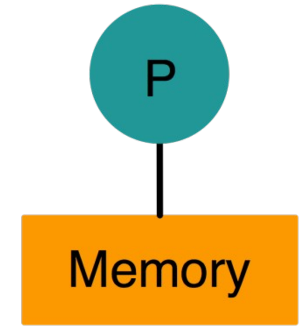
Sequential Consistency

- **Extension of sequential processor model**

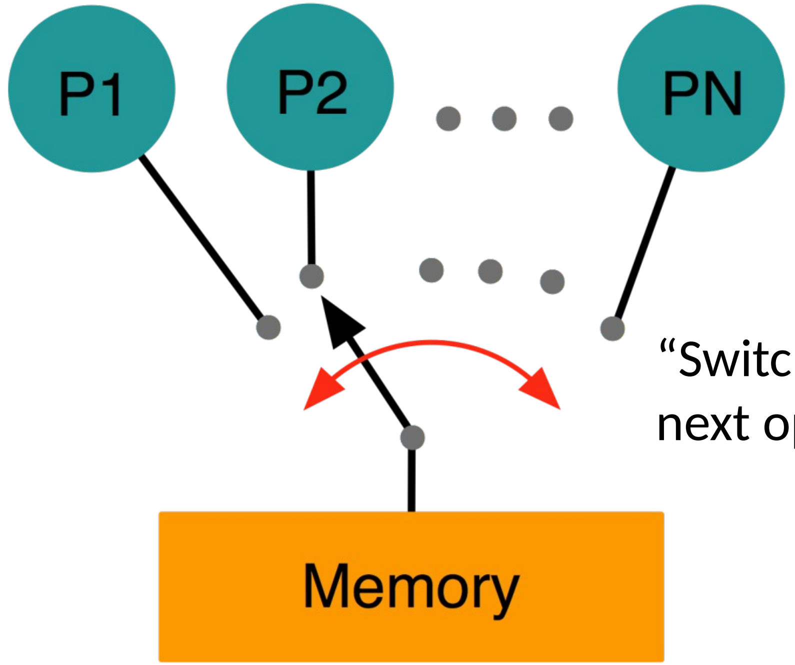
- **The execution happens as if**
  - The operations of all processes were executed in some sequential order (atomicity requirement), and
  - The operations of each individual processor appear in this sequence in the order specified by the program (program order requirement)

- **Applies to all layers!**
  - Disallows many compiler optimizations (e.g., reordering of *any* memory instruction)
  - Disallows many hardware optimizations (e.g., store buffers, nonblocking reads, invalidation buffers)

Illustration of Sequential Consistency



Processors issue in program order



"Switch" selects arbitrary next operation

- **Globally consistent view of memory operations (atomicity)**
- **Strict ordering in program order**

Original SC Definition

"The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program"

(Lamport, 1979)

Good read: Gharachorloo et al.: "Memory consistency and event ordering in scalable shared-memory multiprocessors."

Alternative SC Definition

- **Textbook: Hennessy/Patterson Computer Architecture**

- **A sequentially consistent system maintains three invariants:**
  1. A load L from memory location A issued by processor P_i obtains the value of the previous store to A by P_i, unless another processor has to stored a value to A in between
  2. A load L from memory location A obtains the value of a store S to A by another processor P_k if S and L are "sufficiently separated in time" and if no other store occurred between S and L
  3. Stores to the same location are serialized (defined as in (2))

- **"Sufficiently separated in time" not precise**
  - Works but is not formal (a formalization must include all possibilities)

Example Operation Reordering

- **Recap: “normal” sequential assumption:**
  - Compiler and hardware can reorder instructions as long as control and data dependencies are met
- **Examples:**

Compiler

  - Register allocation
  - Code motion
  - Common subexpression elimination
  - Loop transformations

Hardware

  - Pipelining
  - Multiple issue (OOO)
  - Write buffer bypassing
  - Nonblocking reads

Simple compiler optimization

- **Initially, all values are zero**

P1

```
input = 23
ready = 1
```

P2

```
while (ready == 0) {}
compute(input)
```

  - Assume P1 and P2 are compiled separately!
  - What optimizations can a compiler perform for P1?

Simple compiler optimization

- **Initially, all values are zero**

P1	P2
input = 23 ready = 1	while (ready == 0) {} compute(input)

  - Assume P1 and P2 are compiled separately!
  - What optimizations can a compiler perform for P1?

    Register allocation or even replace with constant, or

    Switch statements
  - What happens?

Simple compiler optimization

- **Initially, all values are zero**

P1	**P2**
input = 23 ready = 1	while (ready == 0) {} compute(input)

  - Assume P1 and P2 are compiled separately!
  - What optimizations can a compiler perform for P1?

    Register allocation or even replace with constant, or

    Switch statements
  - What happens?

    P2 may never terminate, or

    Compute with wrong input

Sequential Consistency Examples

- **Relying on program order: Dekker's algorithm**
  - Initially, all zero

P1
```
a = 1
if(b == 0)
  critical section
  a = 0
```

P2
```
b = 1
if(a == 0)
  critical section
  b = 0
```

  - What can happen at compiler and hardware level?

- **Rel … e seque … atomic … arers**

P1
```
a = 5
a = 1
```

P2
```
if (a == 1)
  b = 1
```

P3
```
if(b == 1)
  print(a)
```

Optimizations violating program order

- **Analyzing P1 and P2 in isolation!**
  - Compiler can reorder

P1

```
a = 1
if(b == 0)
  critical section
  a = 0
```

P2

```
b = 1
if(a == 0)
  critical section
  b = 0
```

P1

```
if(b == 0)
critical section
  a = 0
else
  a = 1
```

P2

```
if(a == 0)
  critical section
  b = 0
else
  b = 1
```

are can … ne a writes g … er or sp

P1

```
a = 1
if(b == 0)
  critical section
  a = 0
```

P2

```
b = 1
if(a == 0)
  critical section
  b = 0
```

P1

```
if(b == 0)
  a = 1
  critical section
  a = 0
```

P2

```
if(a == 0)
  b = 1
  critical section
  b = 0
```

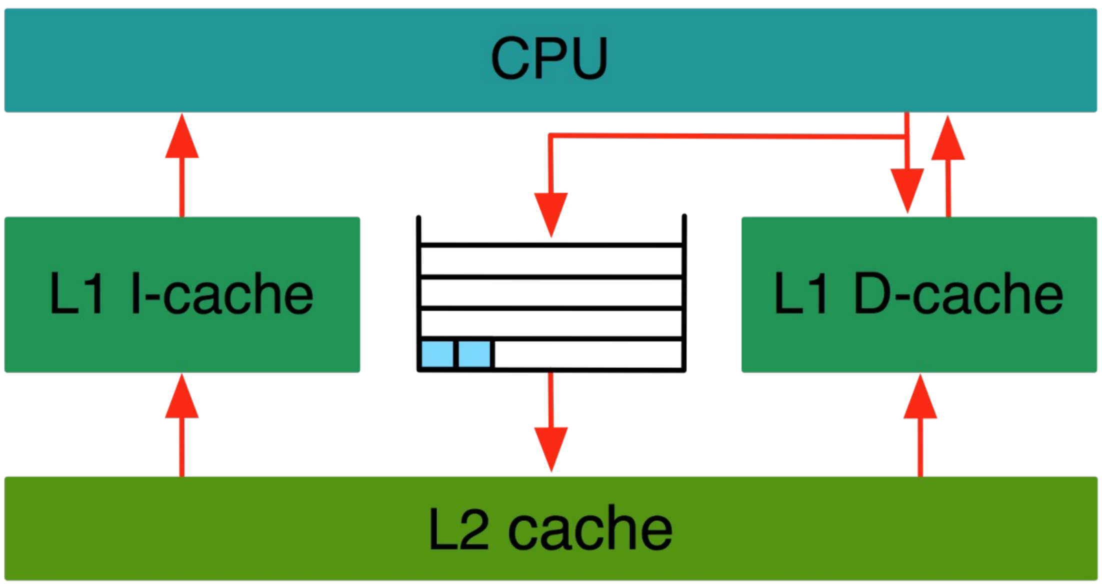
Considerations

- **Define partial order on memory requests A □ B**
  - If P_i issues two requests A and B and A is issued before B in program order, then A □ B
  - A and B are issued to the same variable, and A is entered first, then A □ B (on all processors)
- **These partial orders can be interleaved, define a total order**
  - Many total orders are sequentially consistent!
- **Example:**
  - P1: W(a), R(b), W(c)
  - P2: R(a), W(a), R(b)
  - Are the following schedules (total orders) sequentially consistent?

1. *P1:W(a), P2:R(a), P2:W(a), P1:R(b), P2:R(b), P1:W(c)*
2. *P1:W(a), P2:R(a), P1:R(b), P2:R(b), P1:W(c), P2:W(a)*
3. *P2:R(a), P2:W(a), P1:R(b), P1:W(a), P1:W(c), P2:R(b)*

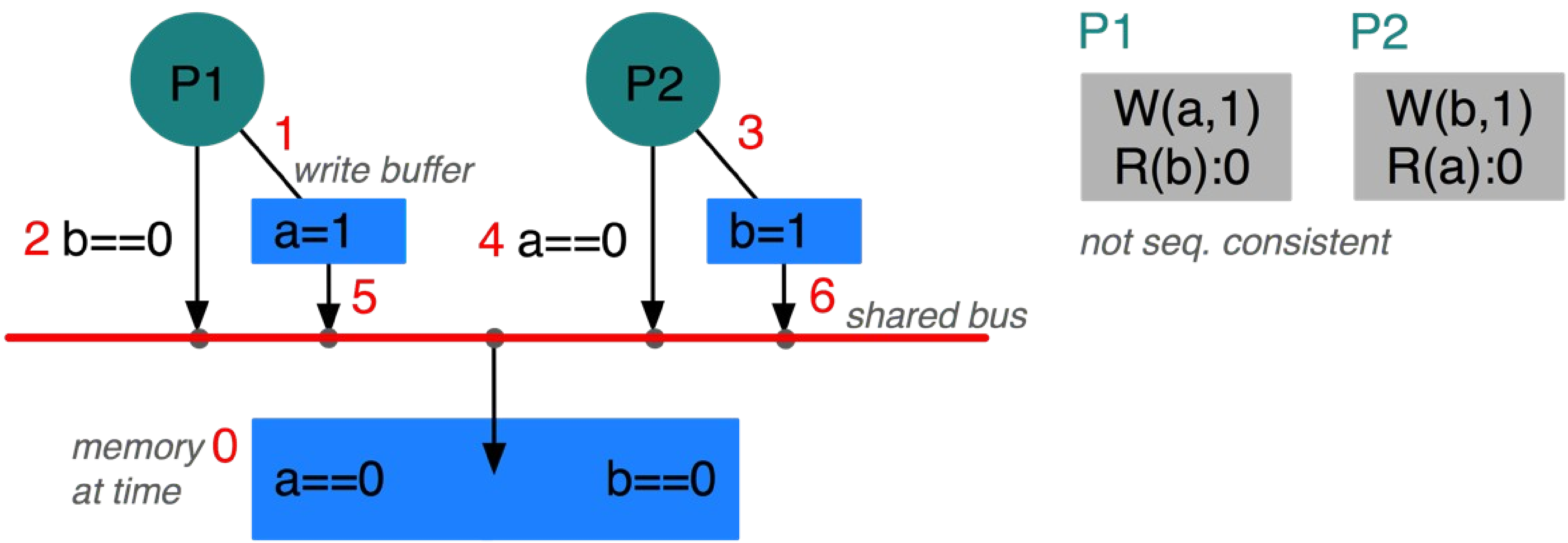
Write buffer example

- **Write buffer**
  - Absorbs writes faster than the next cache ▭ prevents stalls
  - Aggregates writes to the same cache block ▭ reduces cache traffic



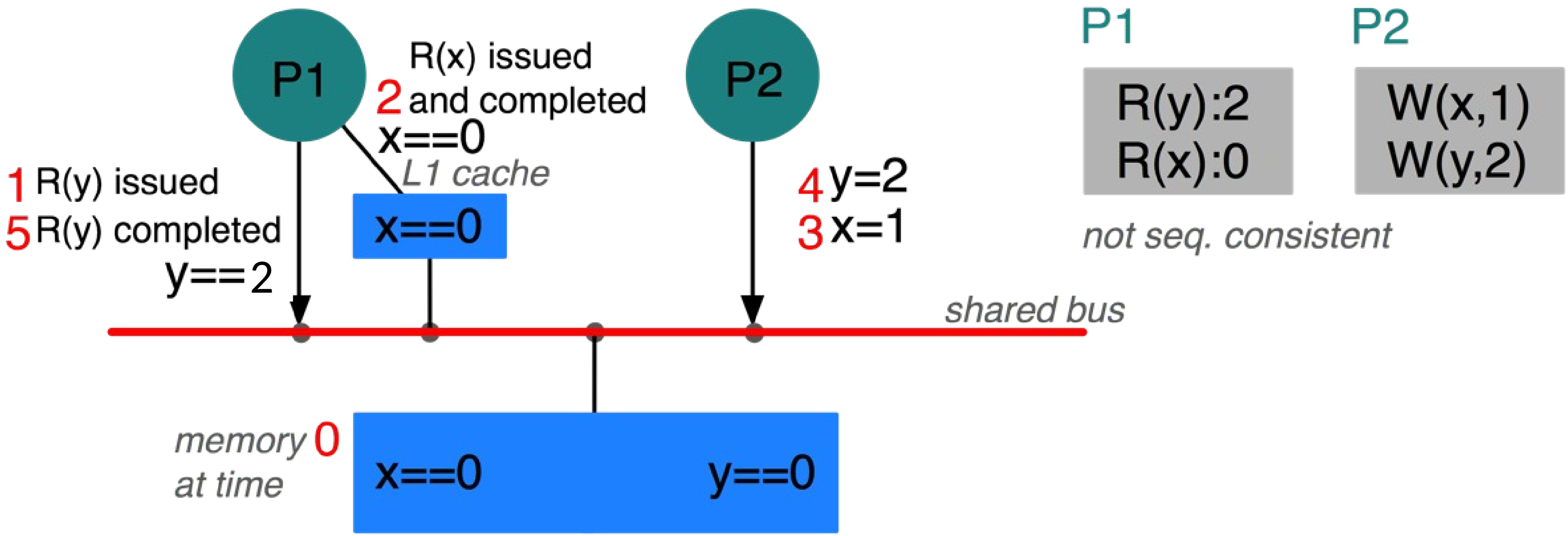
Write buffer example

- **Reads can bypass previous writes for faster completion**
  - If read and write access different locations
  - No order between write and following read (W $\not\to$ R)



Nonblocking read example

- **W $\not\rightarrow$ W: OK**
- **R $\not\rightarrow$ W, R $\not\rightarrow$ R: No order between read and following read/write**



Discussion

- **Programmer's view:**
  - Prefer sequential consistency
  - Easiest to reason about

- **Compiler/hardware designer's view:**
  - Sequential consistency disallows many optimizations!
  - Substantial speed difference
  - ➢ Most architectures and compilers don't adhere to sequential consistency!

- **Solution: synchronized programming**
  - Access to shared data (aka. "racing accesses") are ordered by synchronization operations
  - Synchronization operations guarantee memory ordering (aka. fence)
  - More later!

Cache Coherence vs. Memory Model

- **Varying definitions!**

- **Cache coherence: a mechanism that propagates writes to other processors/caches if needed, recap:**
  - Writes are eventually visible to all processors
  - Writes to the same location are observed in (one) order

- **Memory models: define the bounds on when the value is propagated to other processors**
  - E.g., sequential consistency requires *all* reads and writes to be ordered in program order

Good read: McKenney: “Memory Barriers: a Hardware View for Software Hackers”

Relaxed Memory Models

- **Sequential consistency**
  - R → R, R → W, W → R, W → W (all orders guaranteed)

- **Relaxed consistency (varying terminology):**
  - Processor consistency (aka. TSO)

    Relaxes W → R
  - Partial write (store) order (aka. PSO)

    Relaxes W → R, W → W
  - Weak consistency and release consistency (aka. RMO)

    Relaxes R → R, R → W, W → R, W → W
  - Other combinations/variants possible

    There are even more types of orders (above is a simplification)

Architectures

Memory ordering in some architectures[2][3]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent Instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

Source: Wikipedia

Some older x86 and AMD systems have weaker memory ordering[4]

Case Study: Memory ordering on Intel

- **Intel® 64 and IA-32 Architectures Software Developer's Manual**
  - Volume 3A: System Programming Guide
  - Chapter 8.2 Memory Ordering
  - http://www.intel.com/products/processor/manuals/

- **Google Tech Talk: IA Memory Ordering**
  - Richard L. Hudson

    http://www.youtube.com/watch?v=WUfvvFD5tAA

x86 Memory model: TLO + CC

- **Total lock order (TLO)**
  - Instructions with “lock” prefix enforce total order across all processors
  - Implicit locking: xchg (locked compare and exchange)

- **Causal consistency (CC)**
  - Write visibility is transitive

- **Eight principles**
  - After some revisions ^^

The Eight x86 Principles

1. "Reads are not reordered with other reads." (Rㅂ R)
2. "Writes are not reordered with other writes." (Wㅂ W)
3. "Writes are not reordered with older reads." (Rㅂ W)
4. "Reads may be reordered with older writes to different locations but not with older writes to the same location." (NO Wㅂ R!)
5. "In a multiprocessor system, memory ordering obeys causality." (memory ordering respects transitive visibility)
6. "In a multiprocessor system, writes to the same location have a total order." (implied by cache coherence)
7. "In a multiprocessor system, locked instructions have a total order." (enables synchronized programming!)
8. "Reads and writes are not reordered with locked instructions. " (enables synchronized programming!)

Principle 1 and 2

Reads are not reordered with other reads. (R ㅂ R)

Writes are not reordered with other writes. (W ㅂ W)

All values zero initially

P1	P2
a = 1	r1 = b
b = 2	r2 = a

- If r1 == 2, then r2 must be 1!
- Not allowed: r1 == 2, r2 == 0

Question: is r1=0, r2=1 allowed?

- Reads and writes observed in program order
- Cannot be reordered!

Principle 3

Writes are not reordered with older reads. (R $\sqsubset$ W)

All values zero initially

P1	**P2**
r1 = a b = 1	r2 = b a = 1

- If r1 == 1, then P2:W(a) $\sqsubset$ P1:R(a), thus r2 must be 0!
- If r2 == 1, then P1:W(b) $\sqsubset$ P1:R(b), thus r1 must be 0!

Question: is r1==1 and r2==1 allowed?

- Not allowed: r1 == 1 and r2 == 1

Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W ㅂ R!)

All values zero initially

P1	P2
a = 1	b = 1
r1 = b	r2 = a

- Allowed: r1=0, r2=0

Question: is r1=1, r2=0 allowed?

- Sequential consistency can be enforced with mfence
- Attention: may allow reads to move into critical sections!

Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

All values zero initially

P1	**P2**	**P3**
a = 1	r1 = a	
	b = 1	r2 = b
		r3 = a

- If r1 == 1 and r2==1, then r3 must read 1
- Not allowed: r1 == 1, r2 == 1, and r3 == 0

Question: is r1==1, r2==0, r3==1 allowed?

- Provides some form of atomicity

Principle 6

In a multiprocessor system, writes to the same location have a total order. (implied by cache coherence)

All values zero initially

P1	P2	P3	P4
a=1	a=2		
		r1 = a	r3 = a
		r2 = a	r4 = a

- Not allowed: r1 == 1, r2 == 2, r3 == 2, r4 == 1

  Question: is r1=0, r2=2, r3=0, r4=1 allowed?
- If P3 observes P1's write before P2's write, then P4 will also see P1's write before P2's write
- Provides some form of atomicity

Principle 7

In a multiprocessor system, locked instructions have a total order.
(enables synchronized programming!)

All values zero initially, registers r1==r2==1

P1	P2	P3	P4
xchg(a,r1)	xchg(b,r2)	r3 = a	r5 = b
		r4 = b	r6 = a

- Not allowed: r3 == 1, r4 == 0, r5 == 1, r6 ==0
- If P3 observes ordering P1:xchg ▭ P2:xchg, P4 observes the same ordering

Question: is r3=1, r4=0, r5=0, r6=1 allowed?

- (xchg has implicit lock)

Principle 8

Reads and writes are not reordered with locked instructions.
(enables synchronized programming!)

All values zero initially but r1 = r3 = 1

P1	**P2**
xchg(a,r1) r2 = b	xchg(b,r3) r4 = a

- Not allowed: r2 == 0, r4 == 0
- Locked instructions have total order, so P1 and P2 agree on the same order
- If volatile variables use locked instructions ⇒ practical sequential consistency

An Alternative View: x86-TSO

- **Sewell el al.: “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”, CACM May 2010**

“[…] **real multiprocessors typically do not provide the sequentially consistent memory** that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, **the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion**. […] We present a new x86-TSO programmer’s model that, to the best of our knowledge, suffers from none of these problems. **It is mathematically precise** (rigorously defined in HOL4) but can be presented as an **intuitive abstract machine which should be widely accessible to working programmers**. […]”

Notions of Correctness

- **We discussed so far:**
  - Read/write of the same location

    Cache coherence (write serialization and atomicity)
  - Read/write of multiple locations

    Memory models (visibility order of updates by cores)

- **Now: objects (variables/fields with invariants defined on them)**
  - Invariants "tie" variables together
  - Sequential objects
  - Concurrent objects

Sequential Objects

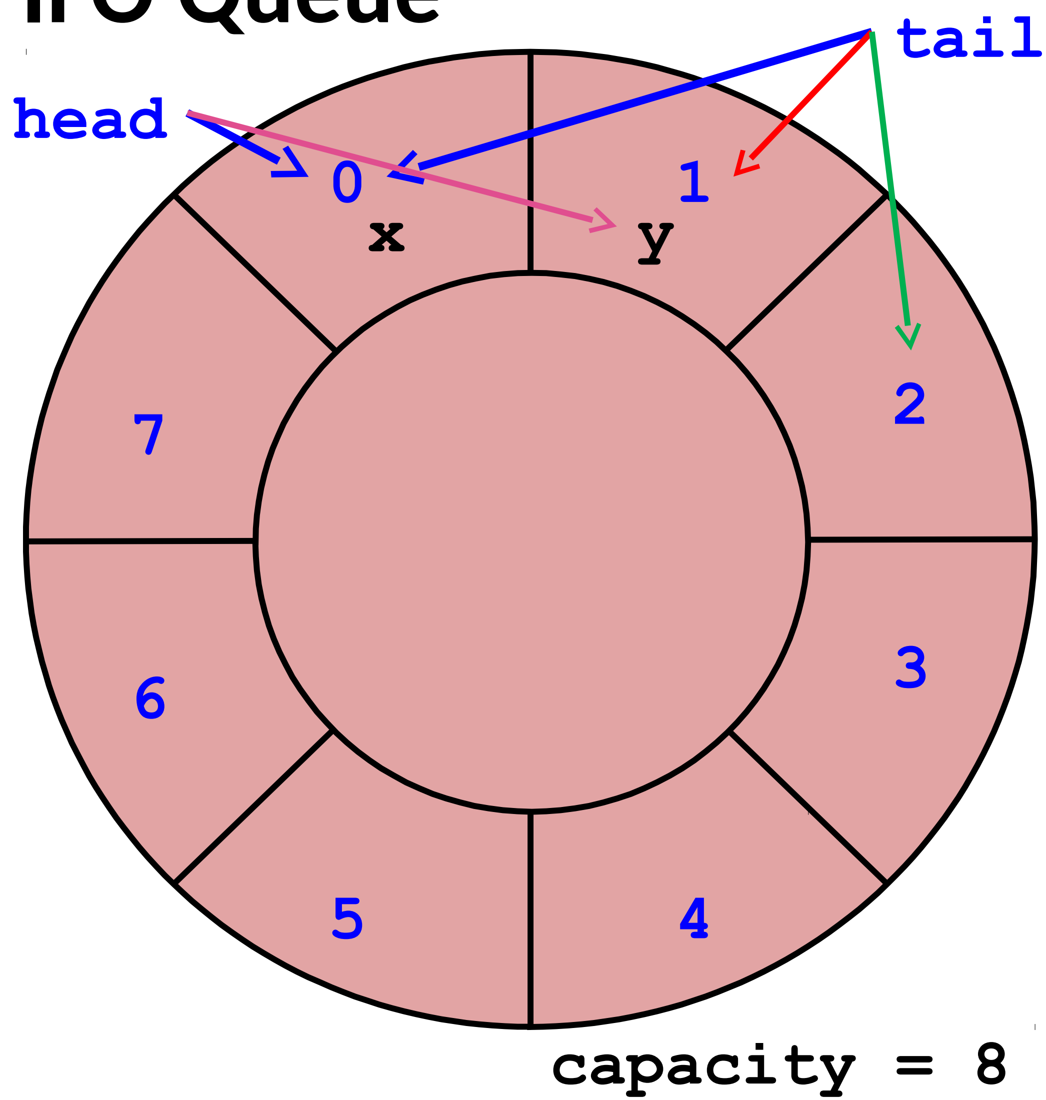
- **Each object has a type**
- **A type is defined by a class**
  - Set of fields forms the state of an object
  - Set of methods (or free functions) to manipulate the state

- **Remark**
  - An Interface is an abstract type that defines behavior

    A class implementing an interface defines several types

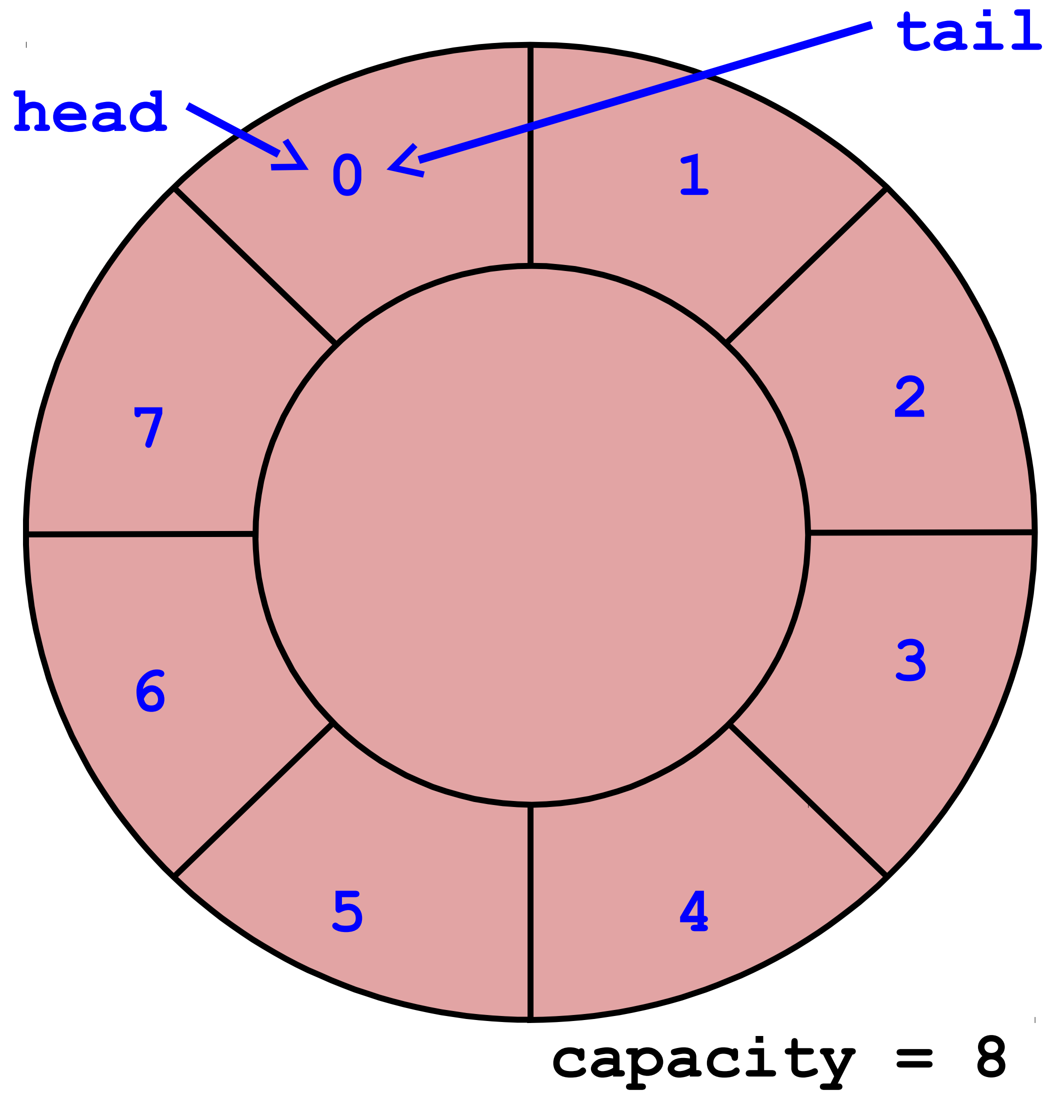
Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
  - deq() [x]
  - ...



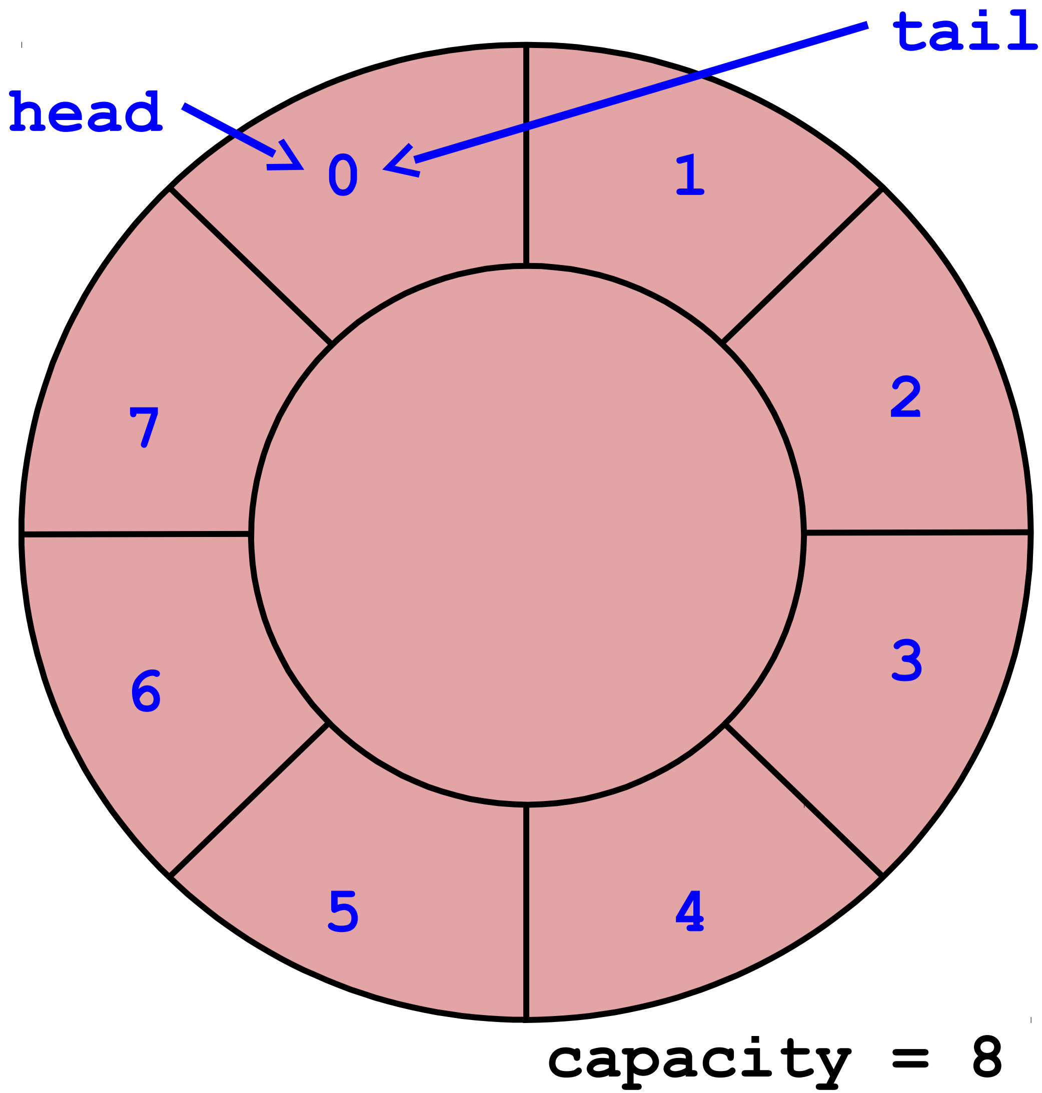
Sequential Queue

```
class Queue {

  private:
  int head, tail;
  std::vector<Item> items;

  public:
  Queue(int capacity) {
    head = tail = 0;
    items.resize(capacity);
  }
  ...
};
```



Sequential Queue

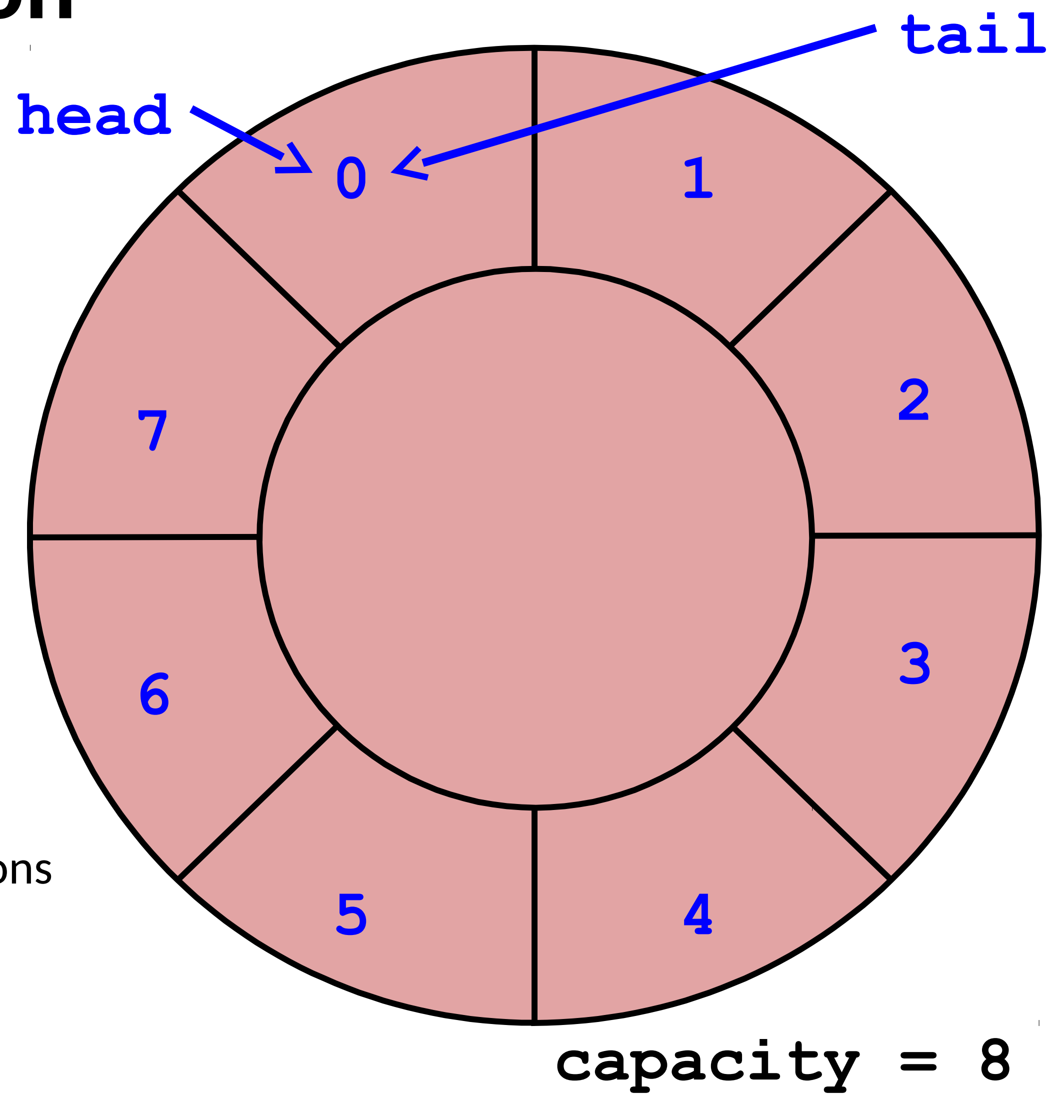
```
class Queue {
 ...

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

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



Sequential Execution

- **(The) one process executes operations one at a time**
  - Sequential ㅆ

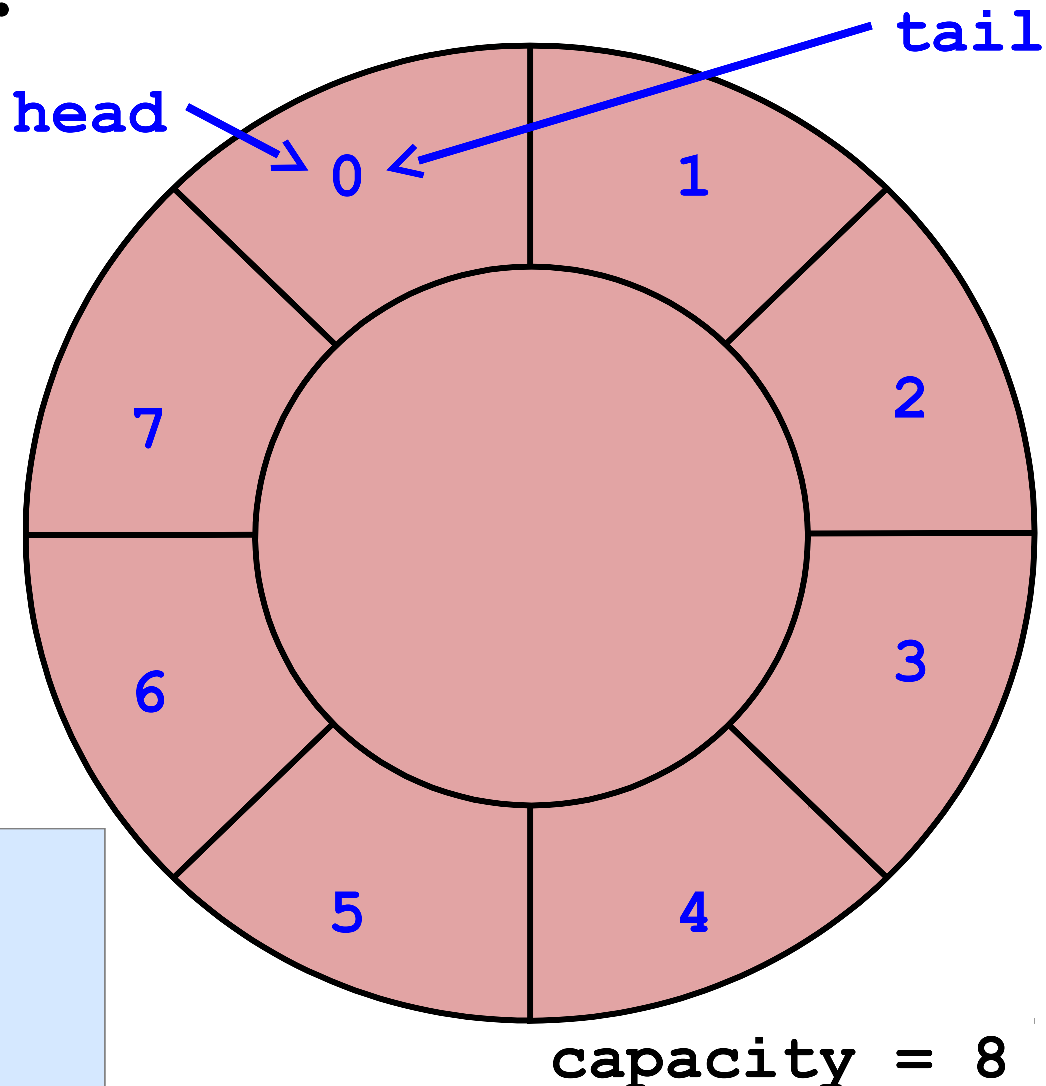
- **Semantics of operation defined by specification of the class**
  - Preconditions and postconditions



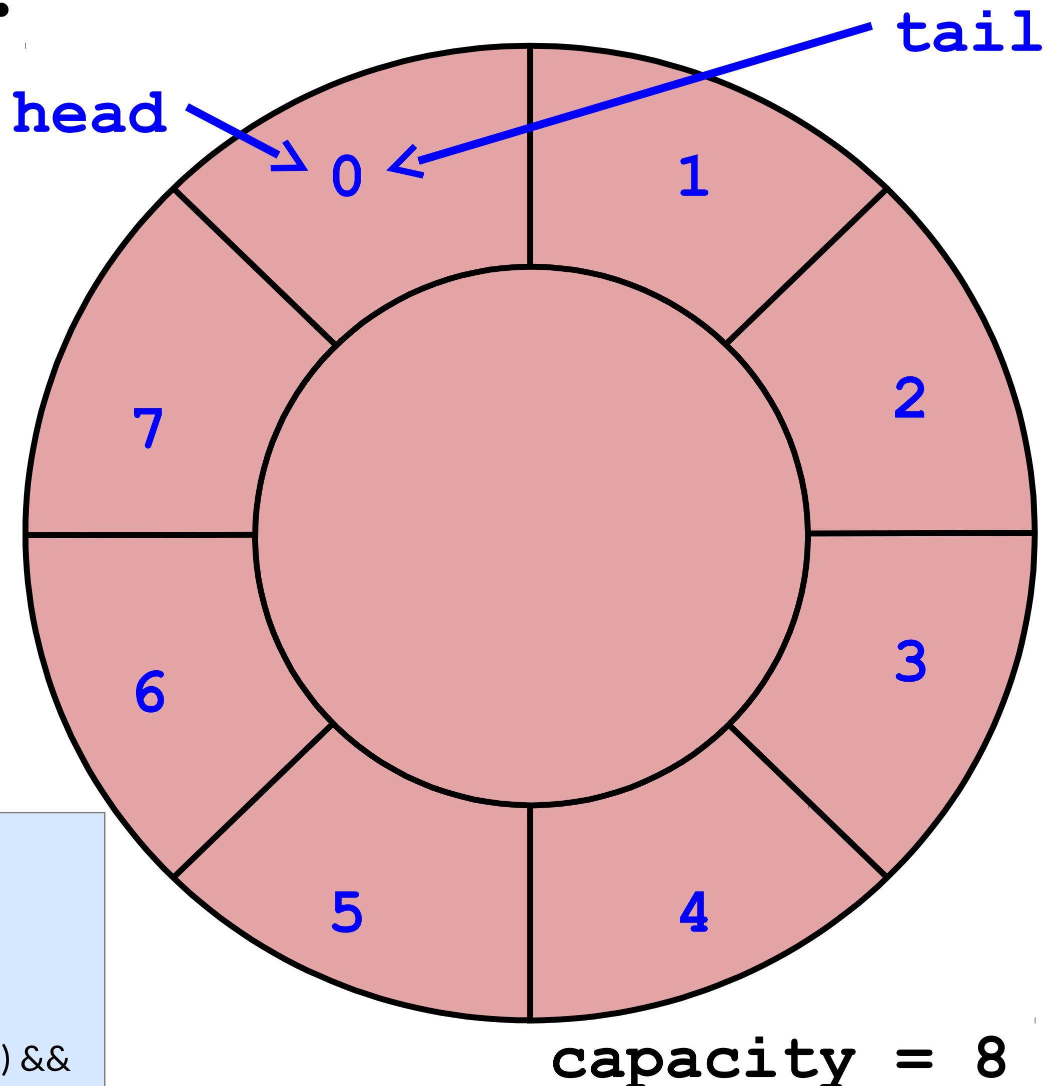
Design by Contract!

- **Preconditions:**
  - Specify conditions that must hold before method executes
  - Involve state and arguments passed
  - Specify obligations a client must meet before calling a method
- **Example: enq()**
  - Queue must not be full!

```
class Queue {
  …
  void enq(Item x) {
    assert((tail+1)%items.size() != head);
    …
  }
};
```



Design by Contract!



- **Postconditions:**
  - Specify conditions that must hold after method executed
  - Involve old state and arguments passed
- **Example: enq()**
  - Queue must contain element!

```
class Queue {
  ...
  void enq(Item x) {
    ...                          creative assertion ^^
    assert( (tail == (old_tail + 1)%items.size()) &&
              (items[old_tail] == x) );
  }
};
```

Sequential specification

- **if(precondition)**
  - Object is in a **specified state**
- **then(postcondition)**
  - The method returns a particular value or
  - Throws a particular exception **and**
  - Leaves the object in a specified state
- **Invariants**
  - Specified conditions (e.g., object state) must hold **anytime** a client could invoke an objects method!

Advantages of sequential specification

- **State between method calls is defined**
  - Enables reasoning about objects
  - Interactions between methods captured by side effects on object state

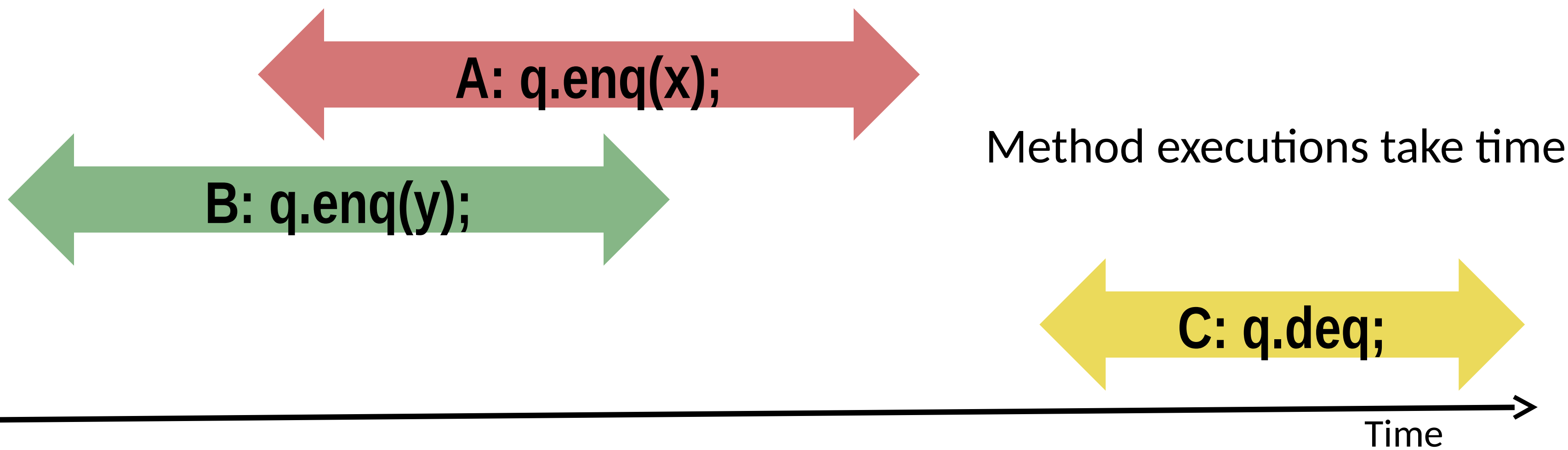
- **Enables reasoning about each method in isolation**
  - Contracts for each method
  - Local state changes global state

- **Adding new methods**
  - Only reason about state changes that the new method causes
  - If invariants are kept: **no need to check old methods**
  - **Modularity!**

Concurrent execution - State

- **Concurrent threads invoke methods on possibly shared objects**
  - At overlapping time intervals!

Property	Sequential	Concurrent
State	Meaningful only between method executions	Overlapping method executions object may never be “between method executions”



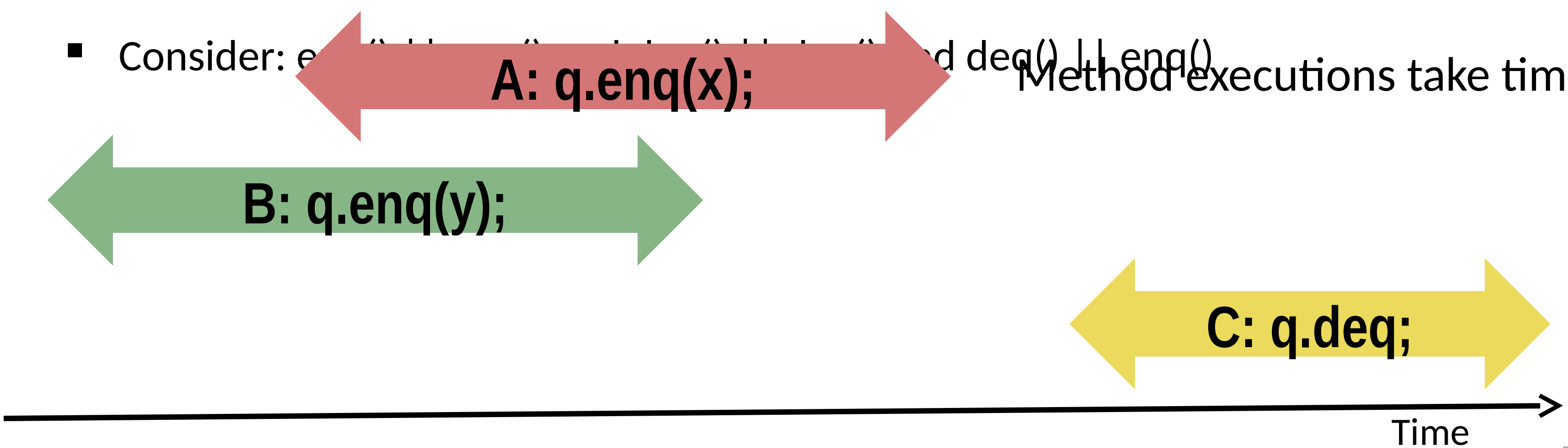
Concurrent execution - Reasoning

- **Reasoning must now include all possible interleavings**
  - Of changes caused by methods themselves

Property	Sequential	Concurrent
Reasoning	Consider each method in isolation; invariants on state before/after execution.	Need to consider all possible interactions; all intermediate states during execution

  - Consider: e… d deq() || enq()

Method executions take time!



Concurrent execution - Method addition

- **Reasoning must now include all possible interleavings**
  - Of changes caused by and methods themselves

Property	Sequential	Concurrent
Add Method	Without affecting other methods; invariants on state before/after execution.	Everything can potentially interact with everything else

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

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

```
Item deq() {
   Item item = items[head];
   head = (head+1)%items.size();
}
```

- peek() || enq(): what if tail has not yet been incremented?

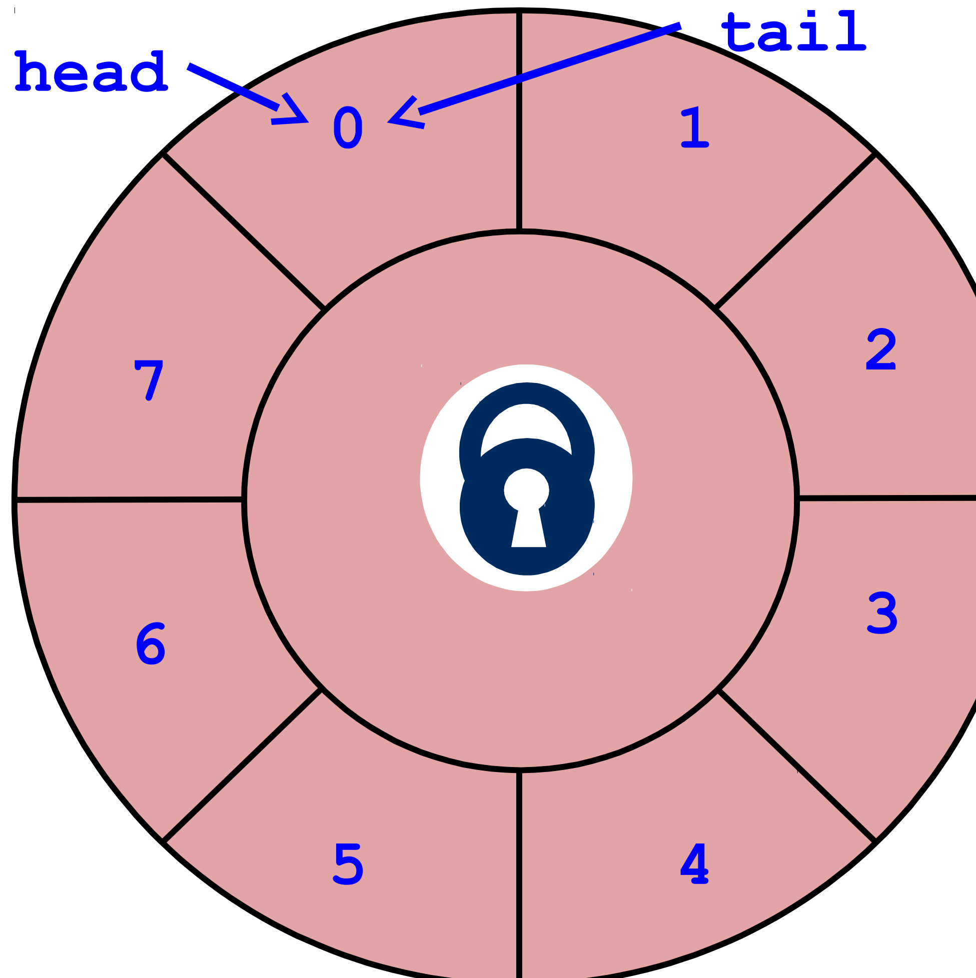
Concurrent objects

- **How do we describe one?**
  - No pre-/postconditions
- **How do we implement one?**
  - Plan for exponential number of interactions
- **How do we tell if an object is correct?**
  - Analyze all exponential interactions

Is it time to panic for software engineers?
Who has a solution?

Lock-based queue

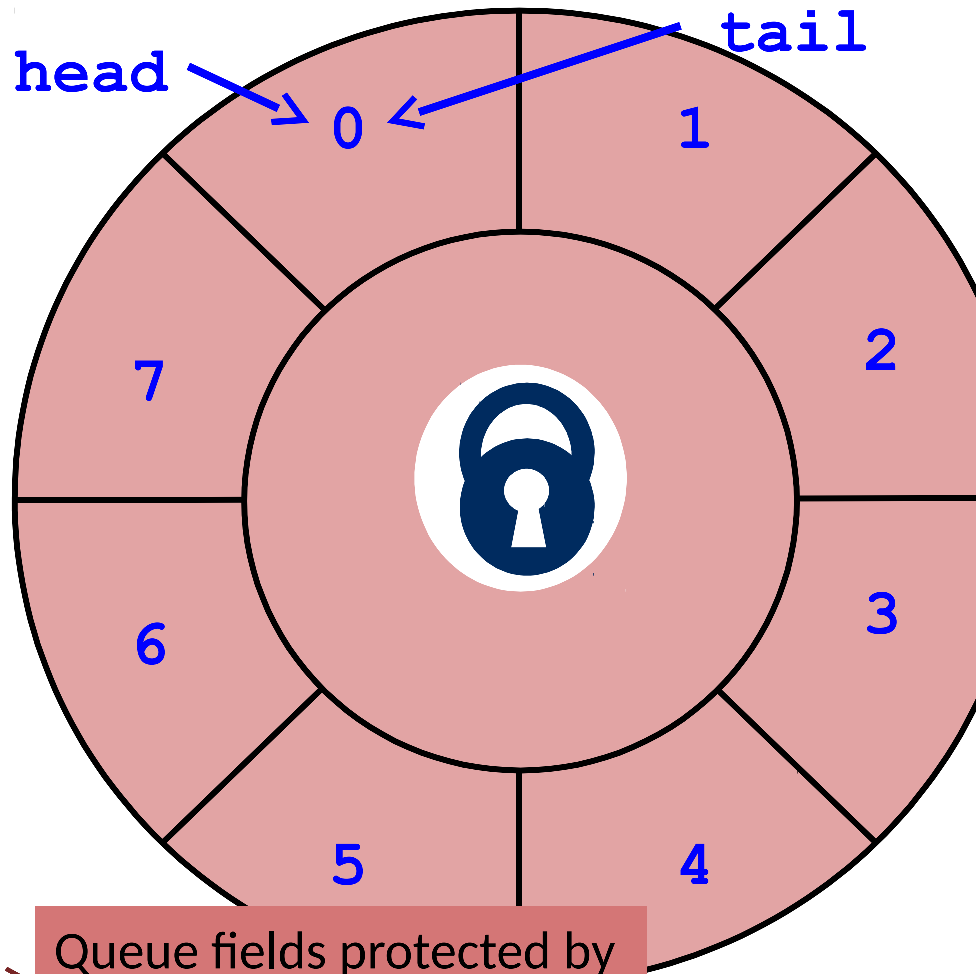
```
class Queue {

  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

```
class Queue {
 ...

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

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



Queue fields protected by single shared lock!

Class question: how is the lock ever unlocked?

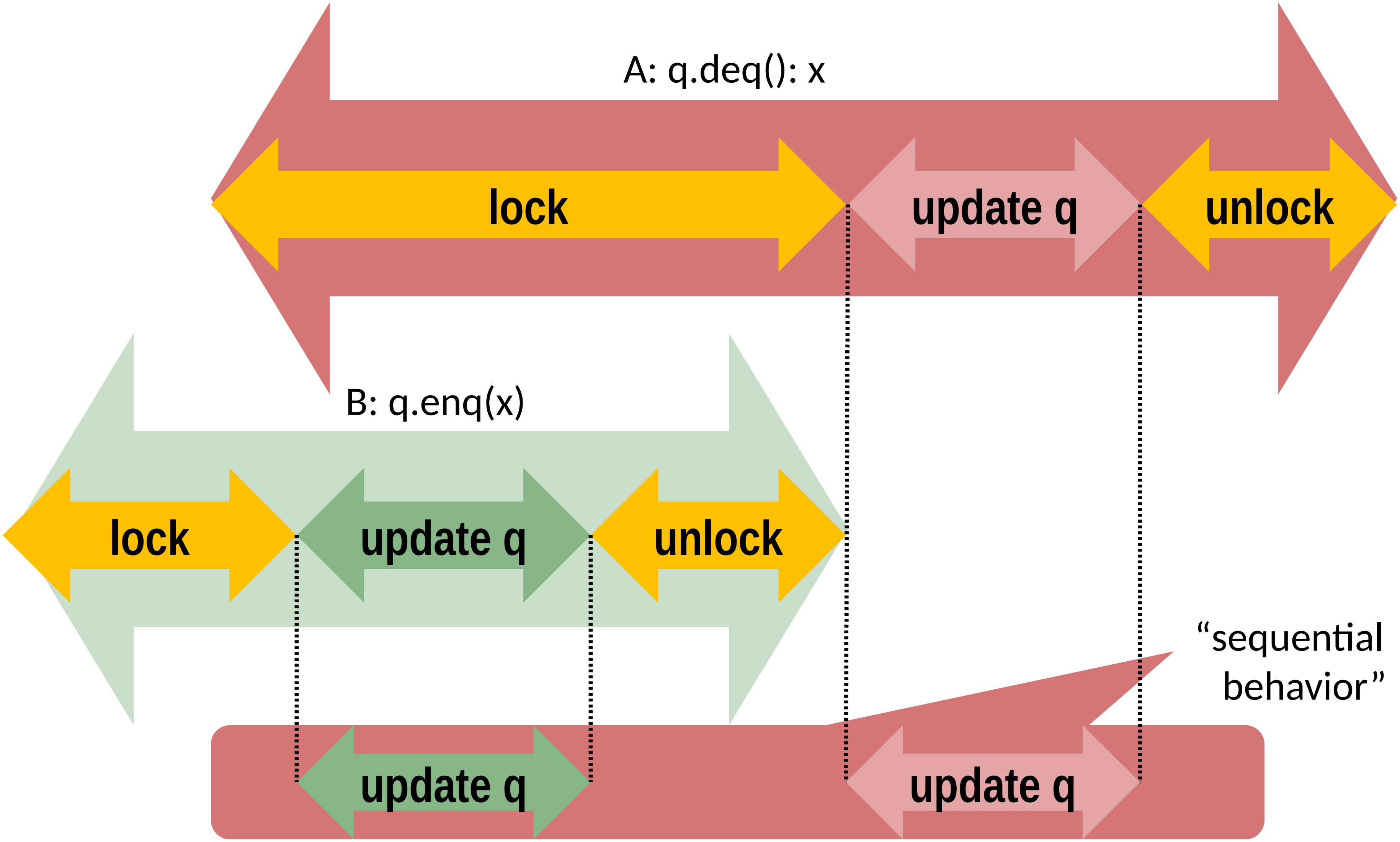
C++ Resource Acquisition is Initialization

- **RAII – suboptimal name**
- **Can be used for locks (or any other resource acquisition)**
  - Constructor grabs resource
  - Destructor frees resource

```
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
    }
};
```

- **Behaves as if**
  - Implicit unlock at end of block!
- **Main advantages**
  - Always free lock at exit
  - No “lost” locks due to exceptions or strange control flow (goto ^^)
  - Very easy to use

Example execution



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?