

Design of Parallel and High-Performance Computing

Fall 2013

Lecture: Languages and Locks

Instructor: Torsten Hoefler & Markus Püschel

TAs: Timo Schneider, Arnamoy Bhattacharyya



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Administrivia

- **You should have a project partner by now**
 - Think about a project
- **Initial project presentations: Thursday 10/30 during 2nd part of recitation**
 - ... may continue the following Monday in lecture (order will be randomized)
 - Send slides (ppt or pdf) by 10/29 11:59pm to Timo/Arnamoy!
 - 7 minutes per team (hard limit)
 - **Prepare!** This is your first impression, gather feedback from us!
 - Rough guidelines:
 - Present your plan*
 - Related work (what exists, literature review!)*
 - Preliminary results (not necessarily)*
 - Main goal is to gather feedback, so present some details*
 - Pick one presenter (make sure to switch for other presentations!)*
- **Intermediate (very short) presentation: Thursday 11/27 during recitation**
- **Final project presentation: Monday 12/15 during last lecture**

Review of last lecture

■ Locked Queue

- Correctness
- Lock-free two-thread queue

■ Linearizability

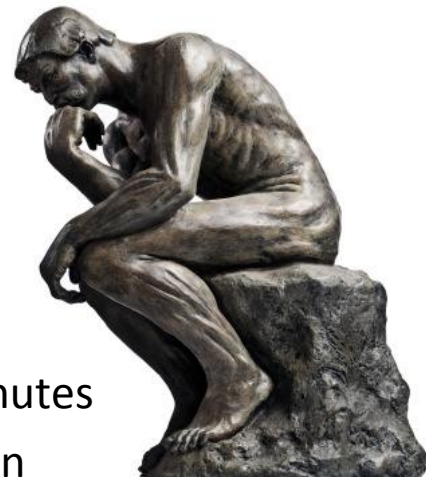
- Combine object pre- and postconditions with serializability
- Additional (semantic) constraints!

■ Histories

- Analyze given histories

Projections, Sequential/Concurrent, Completeness, Equivalence, Well formed, Linearizability (formal)

Peer Quiz



■ Instructions:

- Pick some partners (locally) and discuss each question for 4 minutes
- We then select a random student (team) to answer the question

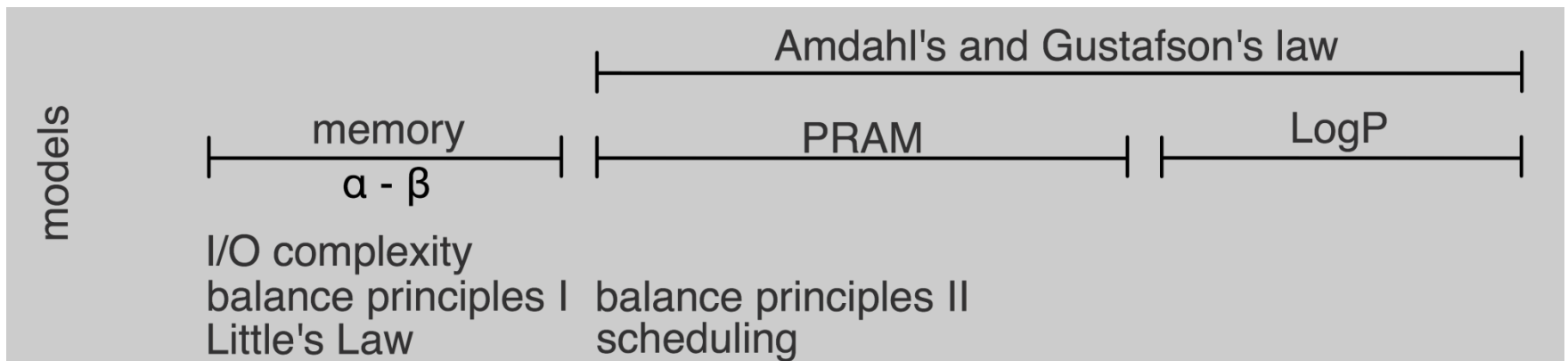
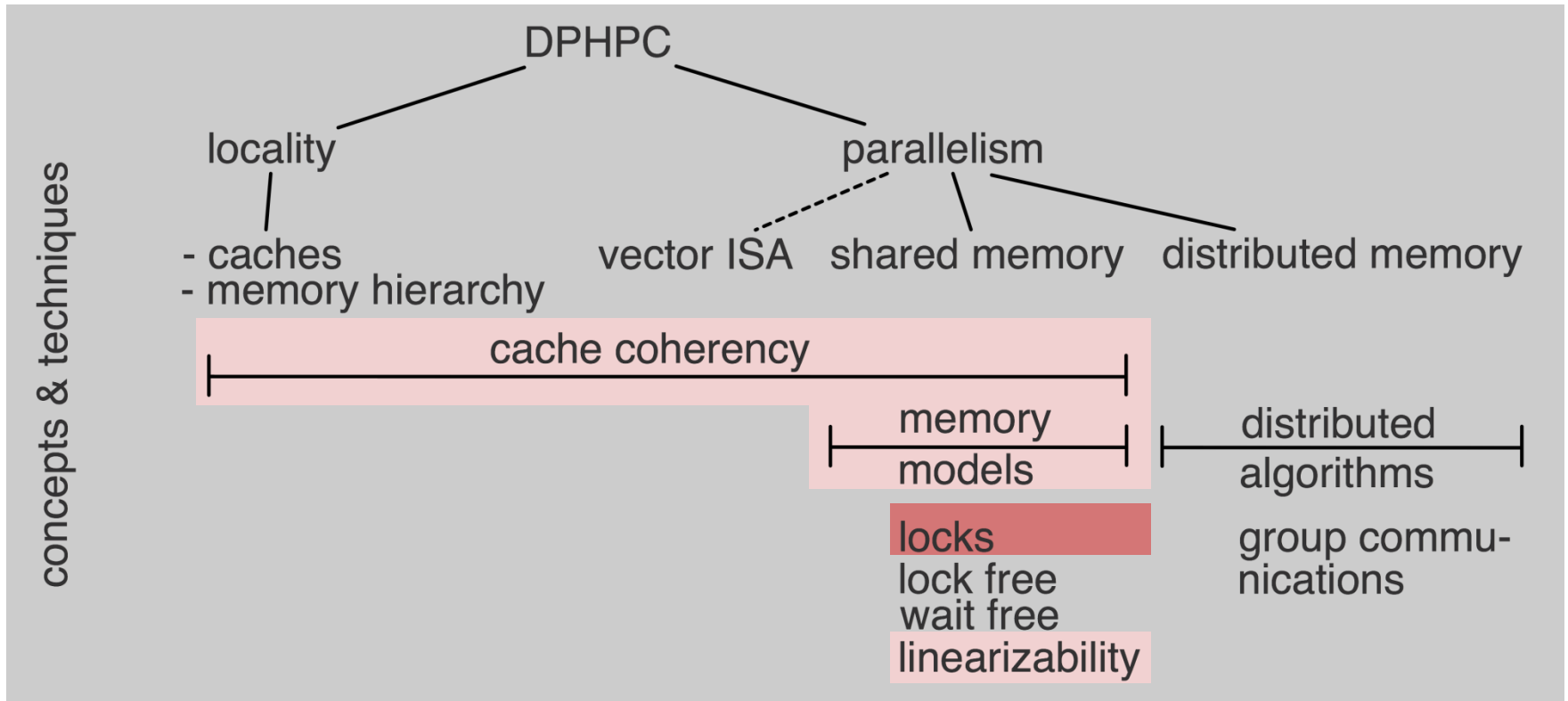
■ How can histories be used to proof a parallel program correct?

- How do histories relate to the source code?
- Can proofing be automated?

■ What are the practical limits of linearizability?

- Can it always be applied?
- Is there a performance tradeoff? Always? Sometimes? Never?

DPHPC Overview



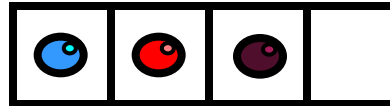
Goals of this lecture

- **Languages and Memory Models**
 - Java/C++ definition
- **Recap serial consistency**
 - Races (now in practice)
- **Mutual exclusion**
- **Locks**
 - Two-thread
 - Peterson
 - N-thread
 - Many different locks, strengths and weaknesses
 - Lock options and parameters
- **Problems and outline to next class**

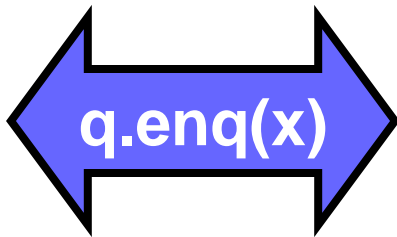
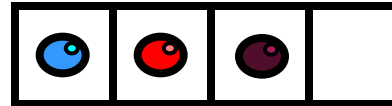
Map linearizability to sequential consistency

- **Variables with read and write operations**
 - Sequential consistency (enforces sequential visibility order)
- **Objects with a type and methods**
 - Linearizability (stronger than SC, considers invocation and response)
- **Map sequential consistency \leftrightarrow 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 (i.e., program order is maintained)
 - Note: Precedence order ($<_H \subseteq <_S$) does not need to be maintained

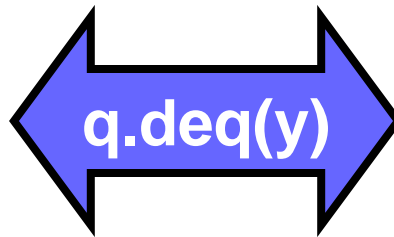
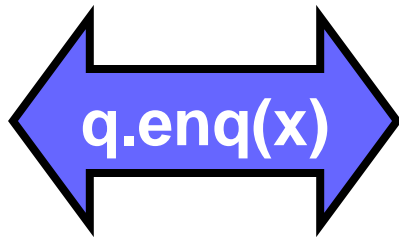
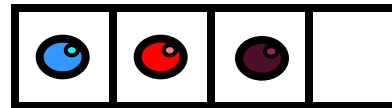
Example



Example



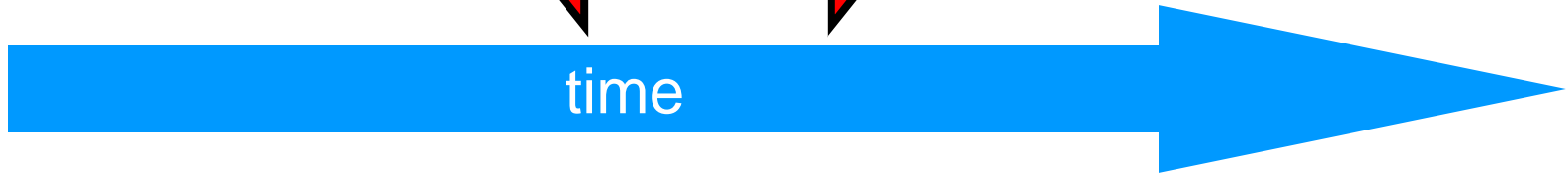
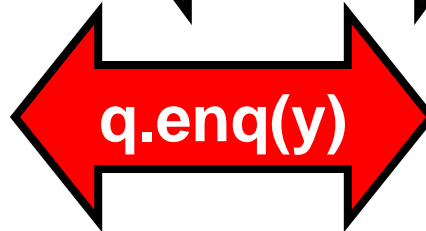
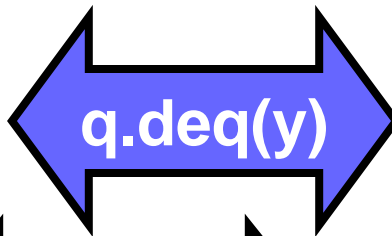
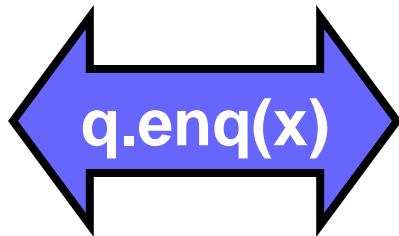
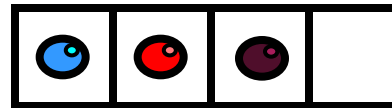
Example



Example



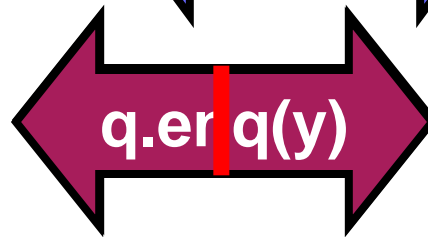
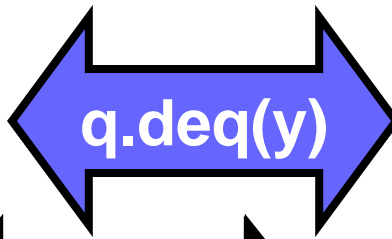
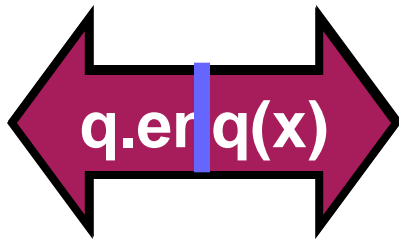
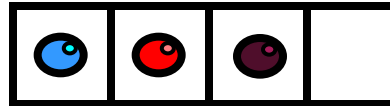
Linearizable?



Example



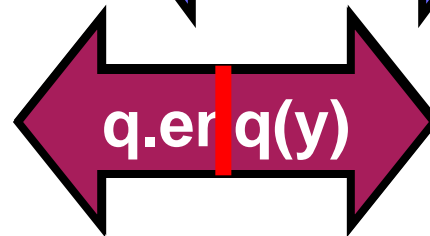
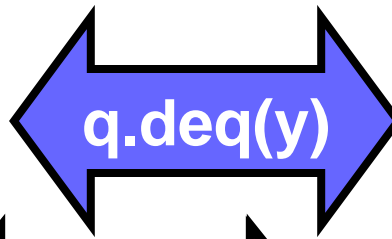
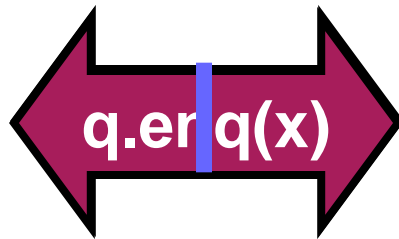
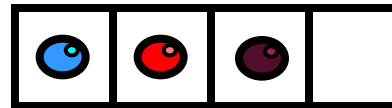
Linearizable?



Example



Linearizable?

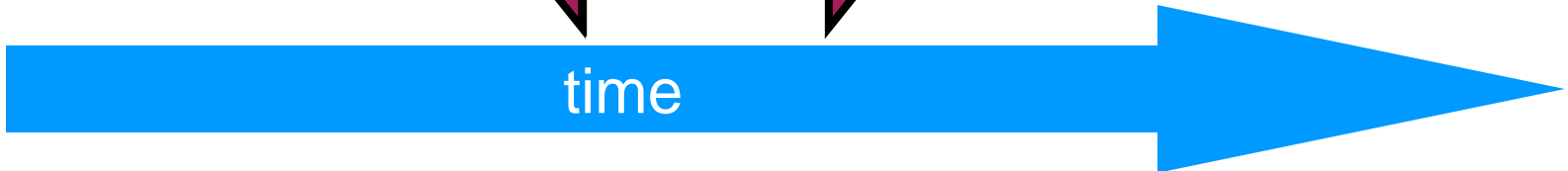
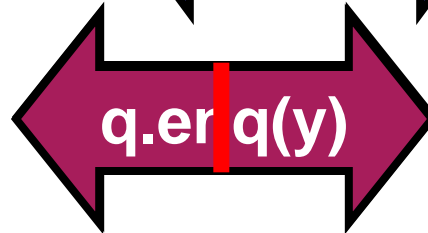
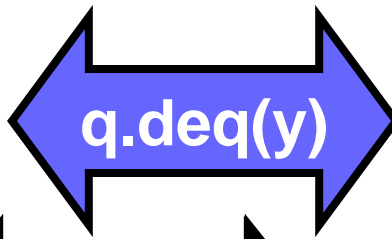
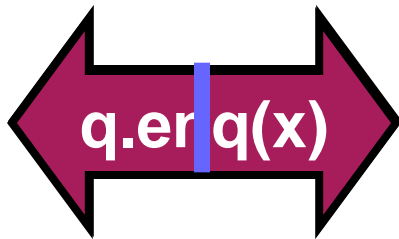
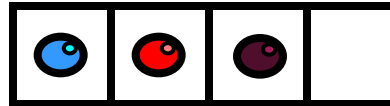


not linearizable

Example



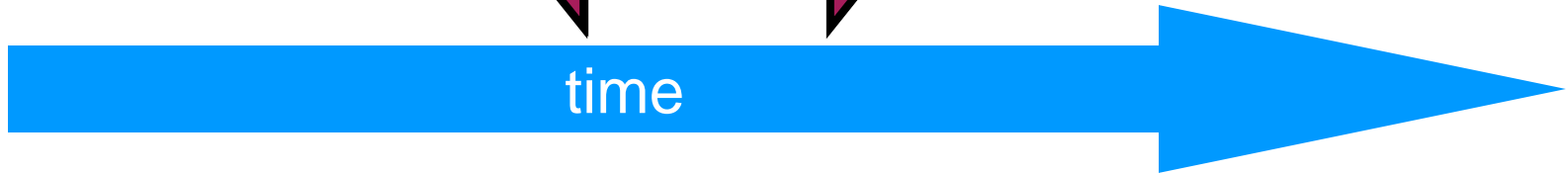
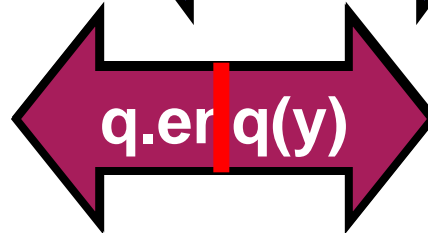
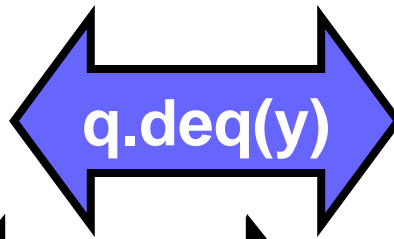
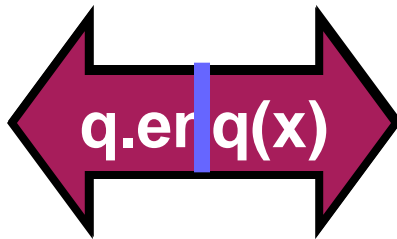
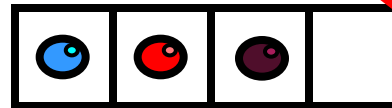
Sequentially consistent?



Example



Sequentially consistent?



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

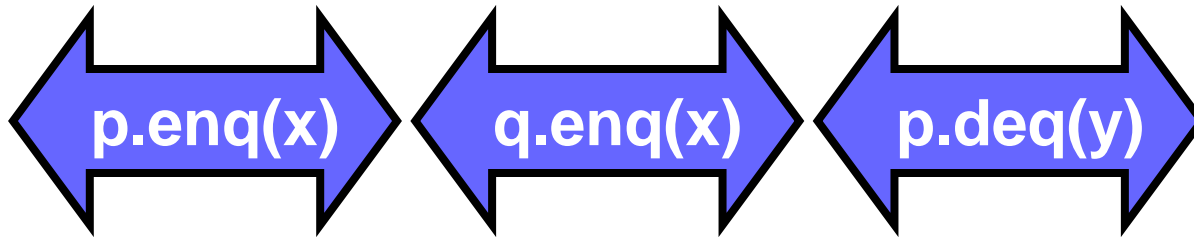
Compositional would mean:

“If $H|p$ and $H|q$ are sequentially consistent, then H is sequentially consistent!”

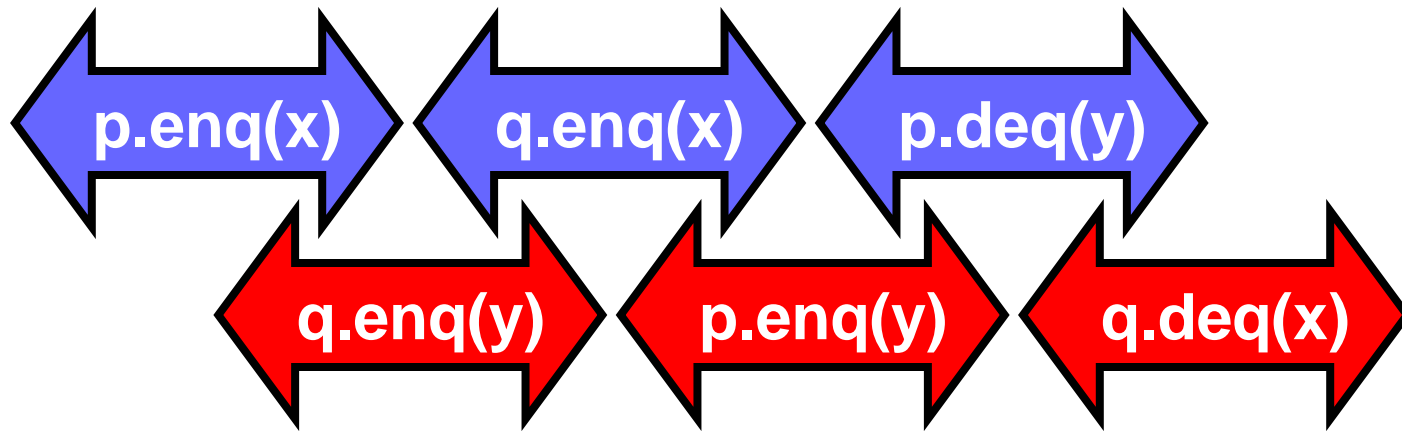
This is not guaranteed for SC schedules!

See following example!

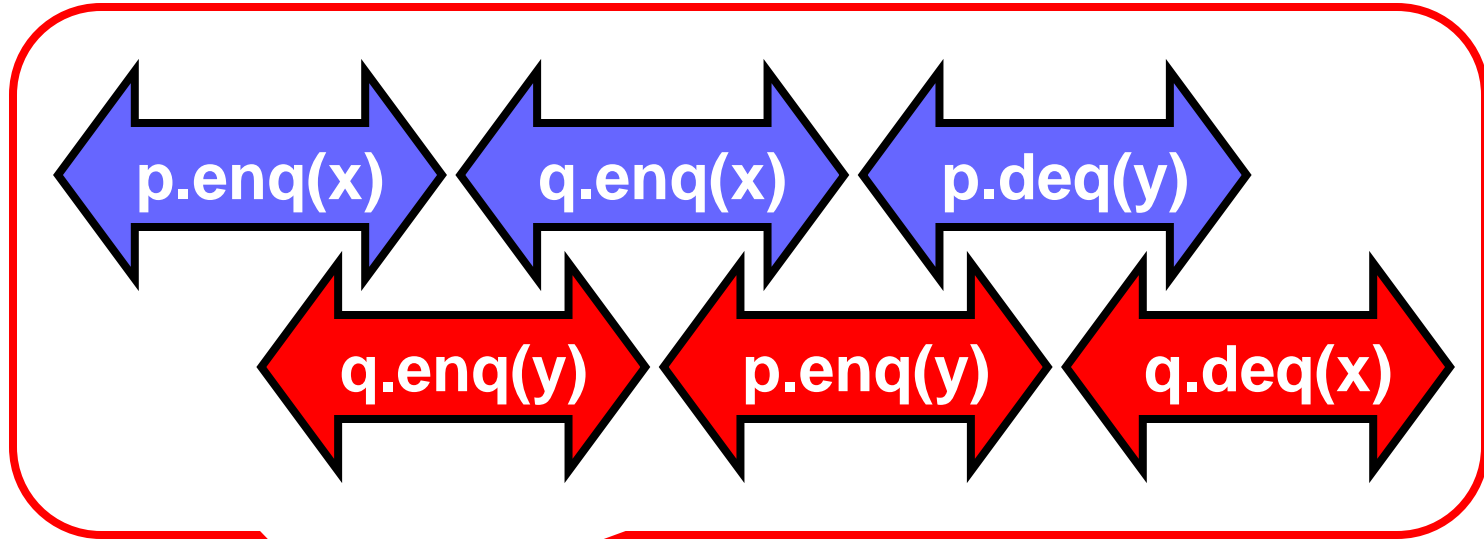
FIFO Queue Example



FIFO Queue Example



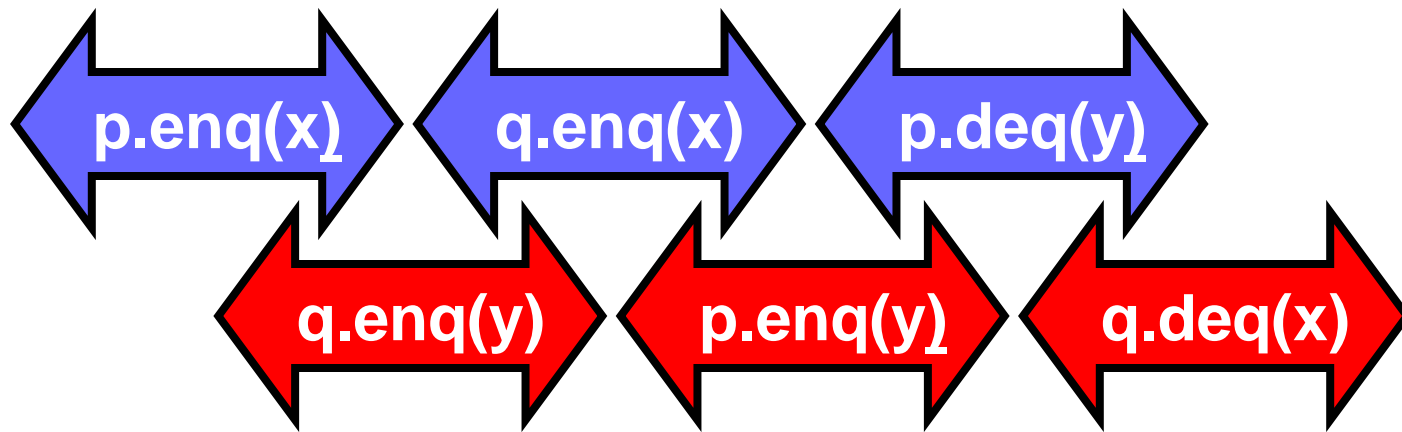
FIFO Queue Example



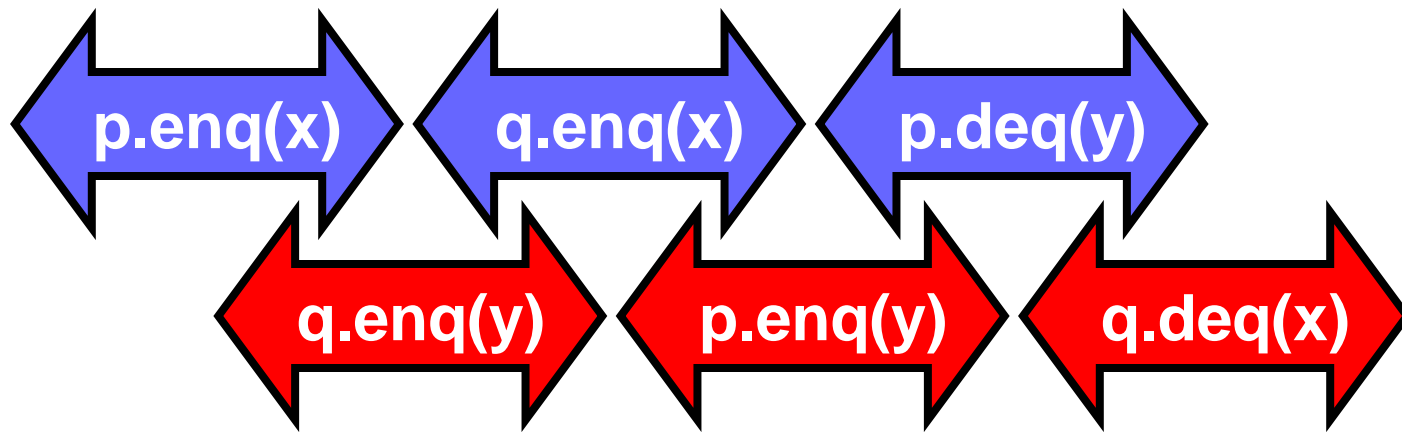
History H



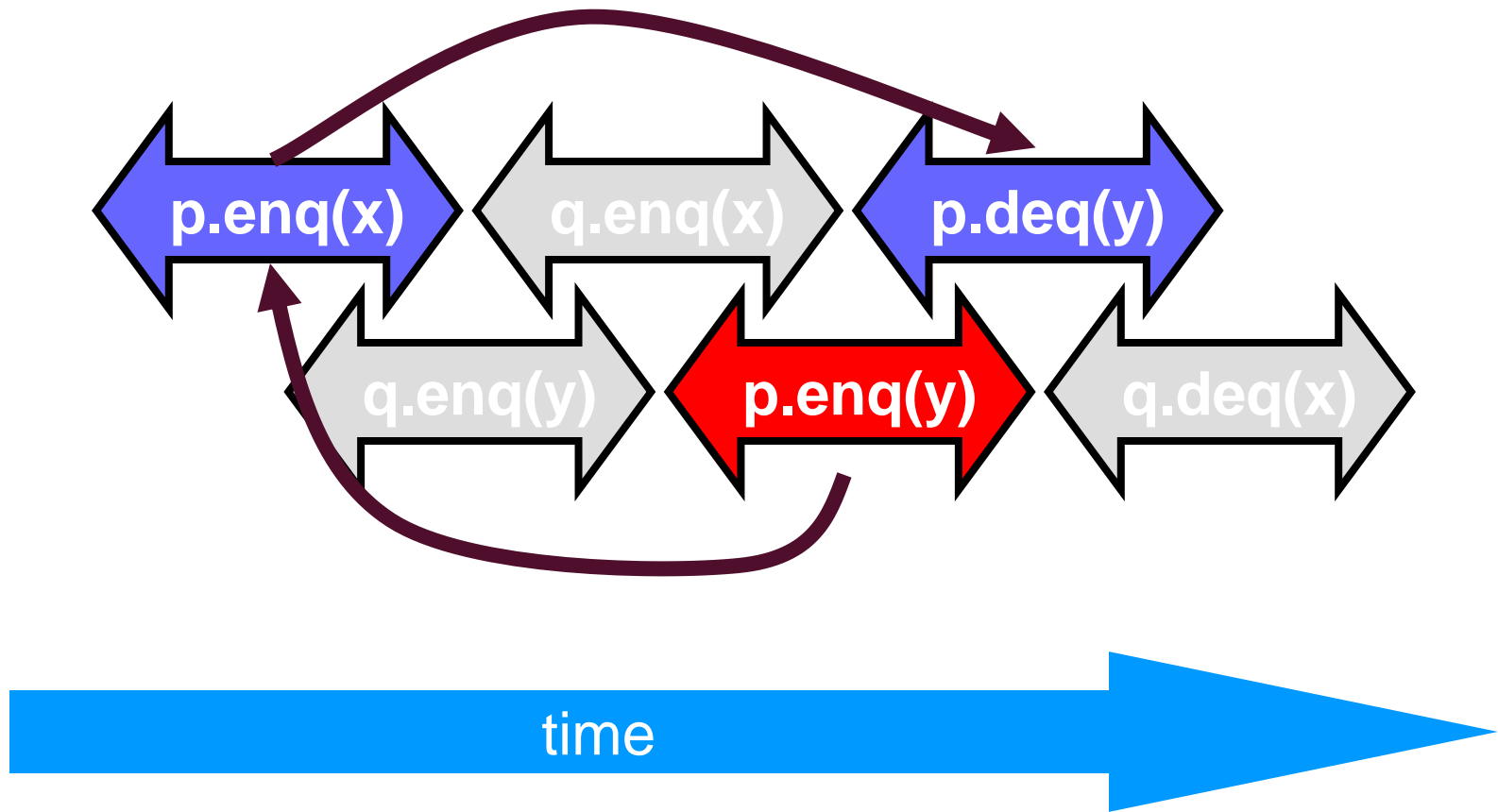
H/p Sequentially Consistent



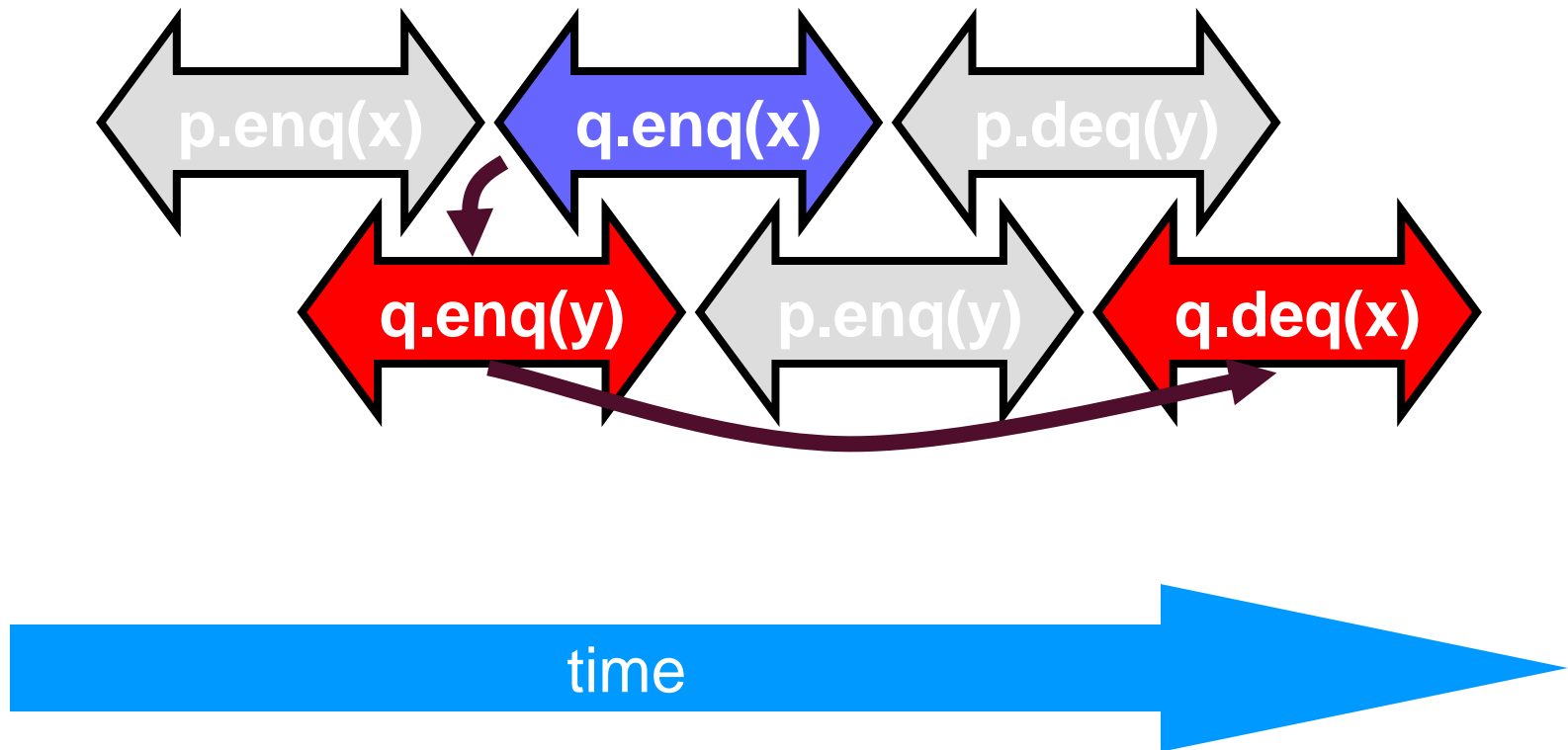
H|q Sequentially Consistent



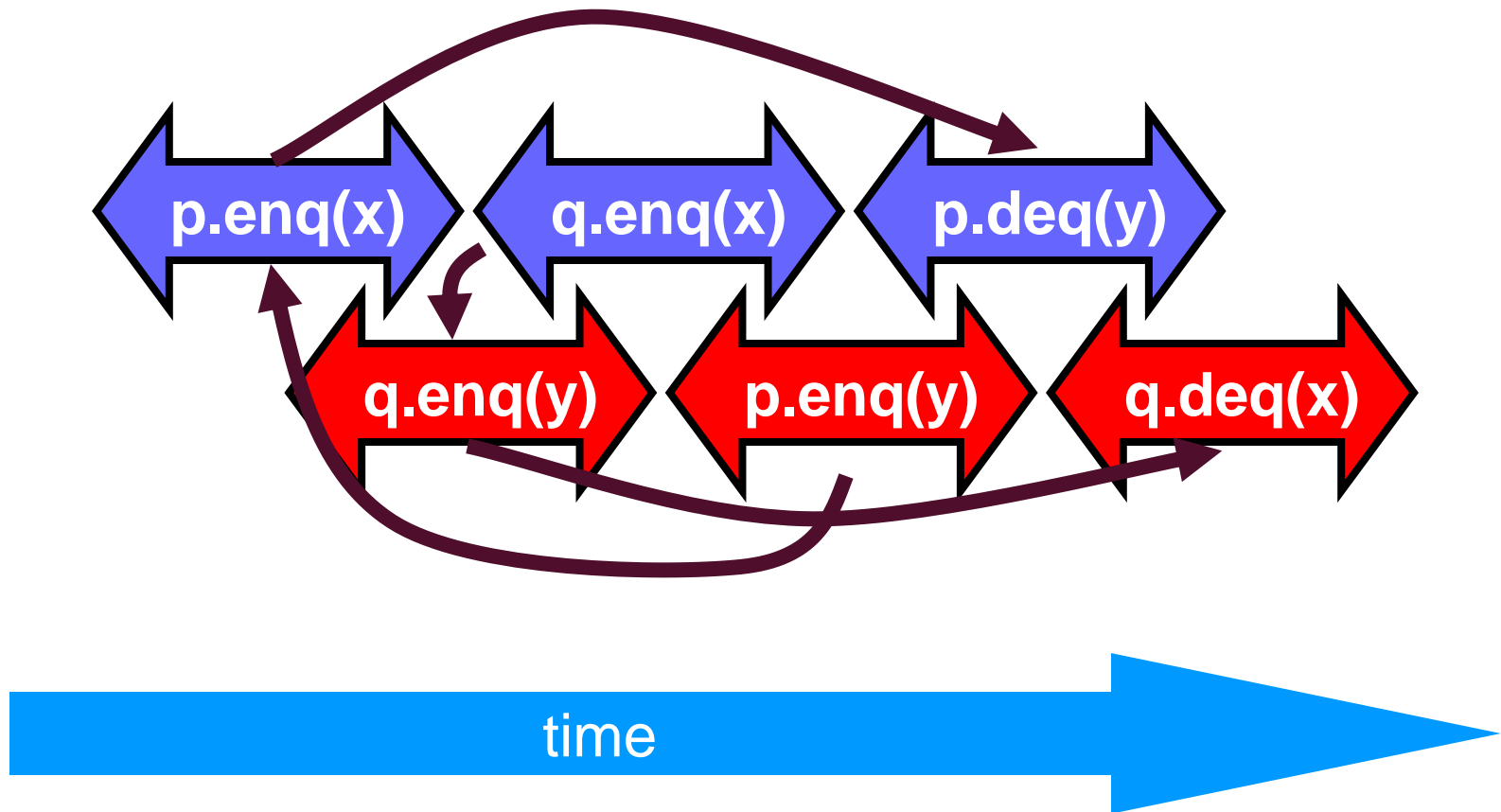
Ordering imposed by p



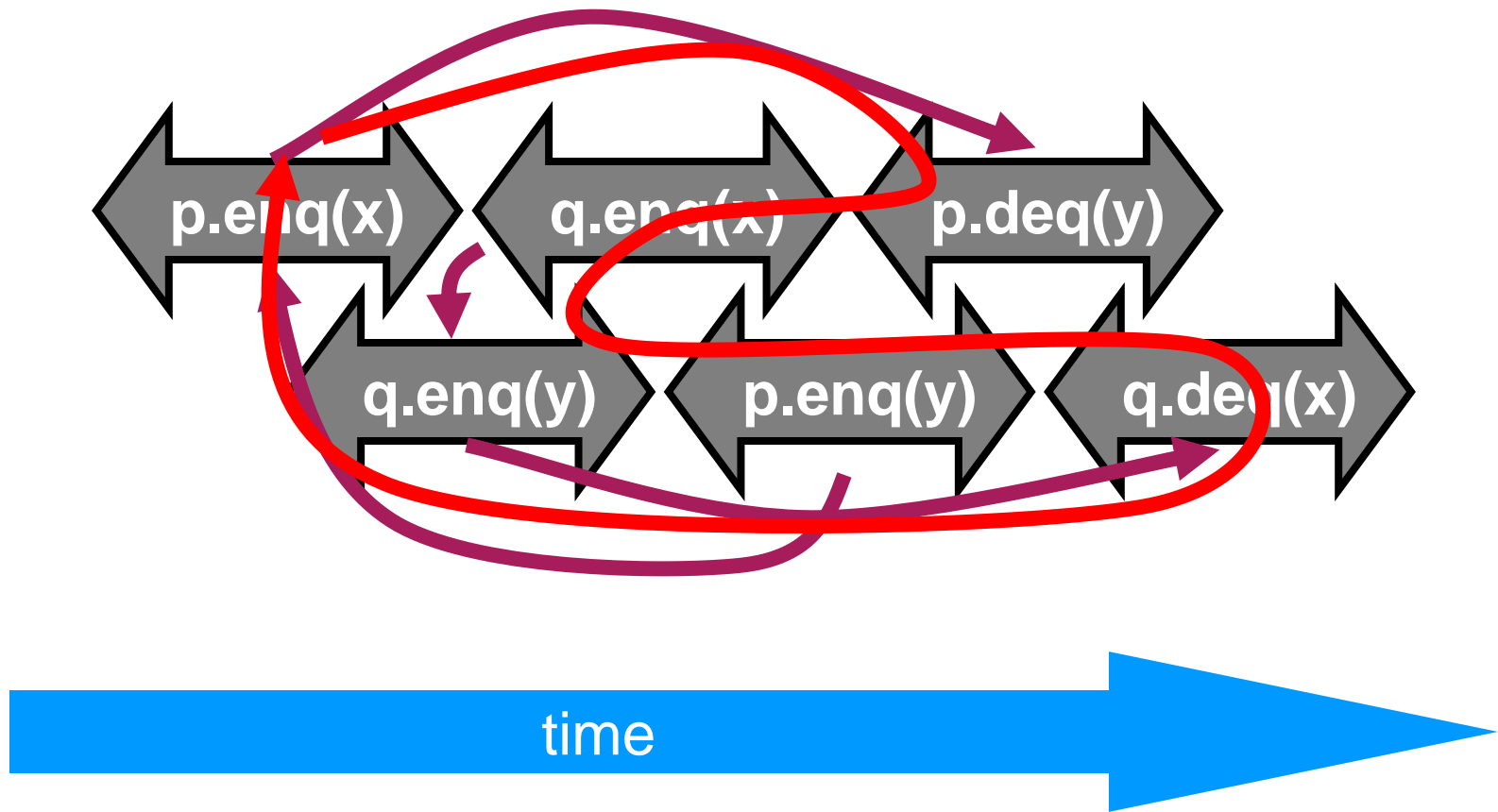
Ordering imposed by q



Ordering imposed by both



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

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|A=

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

H|B=

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

H is not sequentially consistent!

Correctness: Linearizability

- **Sequential Consistency**

- Not composable
- Harder to work with
- Good way to think about hardware models

- **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 an 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**
 - 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**
 - Java 1.5 (JSR-133)
 - Still some issues (operational semantics definition)
- **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`, ...)

Thread 1

```
x = 10  
y = 5  
flag = true
```

synchronization

Thread 2

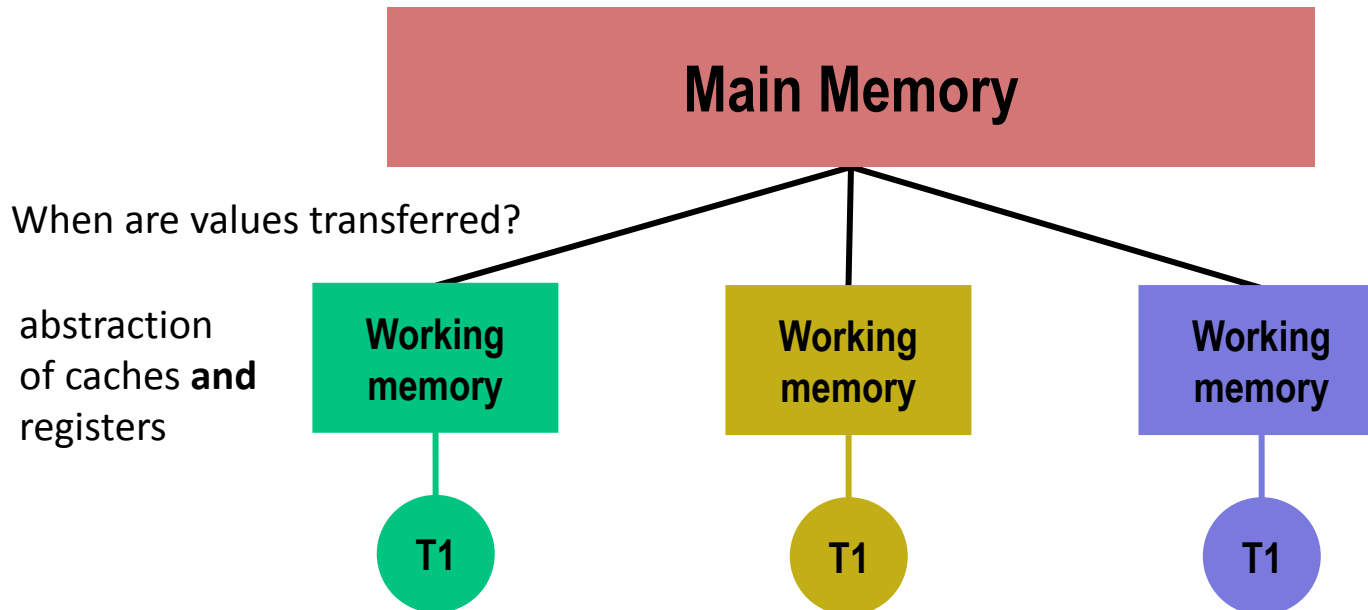
```
if(flag)  
print(x+y)
```

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*!

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 (lock) {  
    // critical region  
}
```

- Synchronized methods as syntactic sugar

■ C++

```
{  
    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

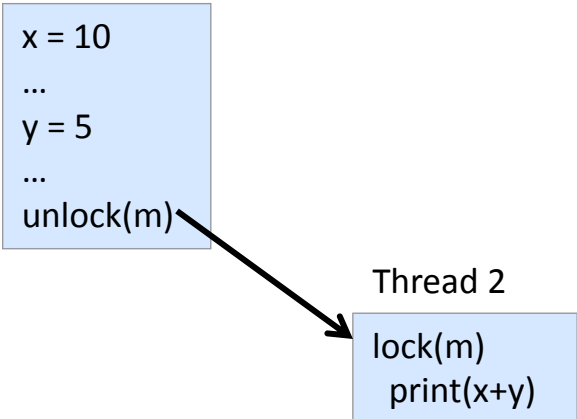
- Similar to synchronization variables

Thread 1

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

Thread 2

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



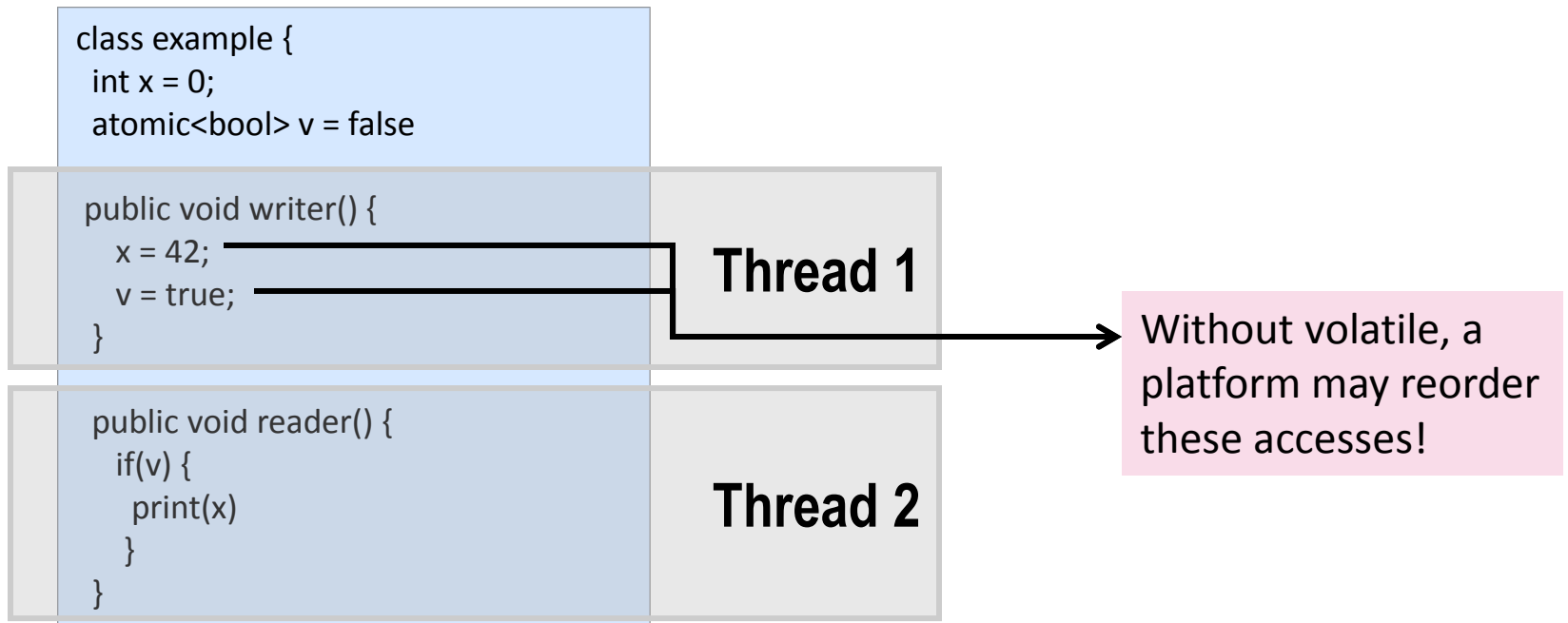
- All memory accesses **before** an unlock ...
- are ordered before and are visible to ...
- any memory access **after** a matching lock!

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
 - ...

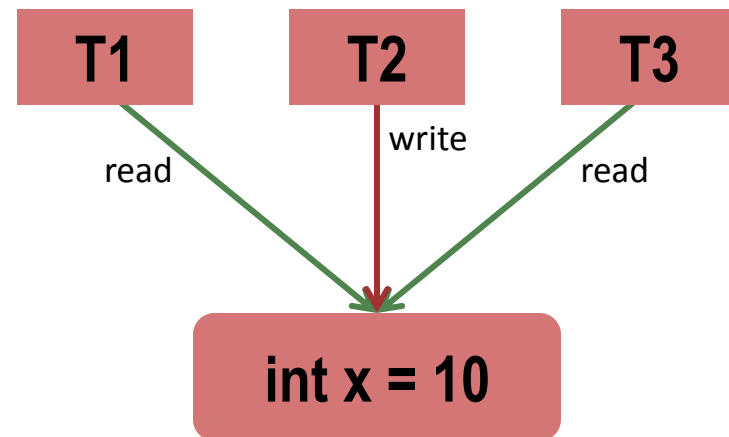
Synchronization Variables

- **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!)



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



Case Study: Locks - Lecture Goals

- **Among the simplest concurrency constructs**
 - Yet, complex enough to illustrate many optimization principles
- **Goal 1: You understand locks in detail**
 - Requirements / guarantees
 - Correctness / validation
 - Performance / scalability
- **Goal 2: Acquire the ability to design your own locks**
 - Understand techniques and weaknesses/traps
 - Extend to other concurrent algorithms
 - Issues are very much the same*
- **Goal 3: Feel the complexity of shared memory!**

Preliminary Comments

- **All code examples are in C/C++ style**

- Neither C nor C++ <11 have a clear memory model
- C++ is one of the languages of choice in HPC
- Consider source as exemplary (and pay attention to the memory model)!

In fact, many/most of the examples are incorrect in anything but sequential consistency!

In fact, you'll most likely not need those algorithms, but the principles will be useful!

- **x86 is really only used because it's common**

- This does not mean that we consider the ISA or memory model elegant!
- We assume atomic memory (or registers)!

Usually given on x86 (easy to enforce)

- **Number of threads/processes is p , tid is the thread id**

Recap Concurrent Updates

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
    a++;
```

gcc -O3

```
movl $1000, %eax // i=n=1000
.L2:
movl (%rdx), %ecx // ecx = *a
addl $1, %ecx // ecx++
subl $1, %eax // i--
movl %ecx, (%rdx) // *a = ecx
jne .L2 // loop if i>0
```

■ Multi-threaded execution!

- Value of a for p=1?
- Value of a for p>1?

Why? Isn't it a single instruction?

```
const int n=1000;
std::atomic<int> a;
a=0;
for (int i=0; i<n; ++i)
    a++;
```

g++ -O3

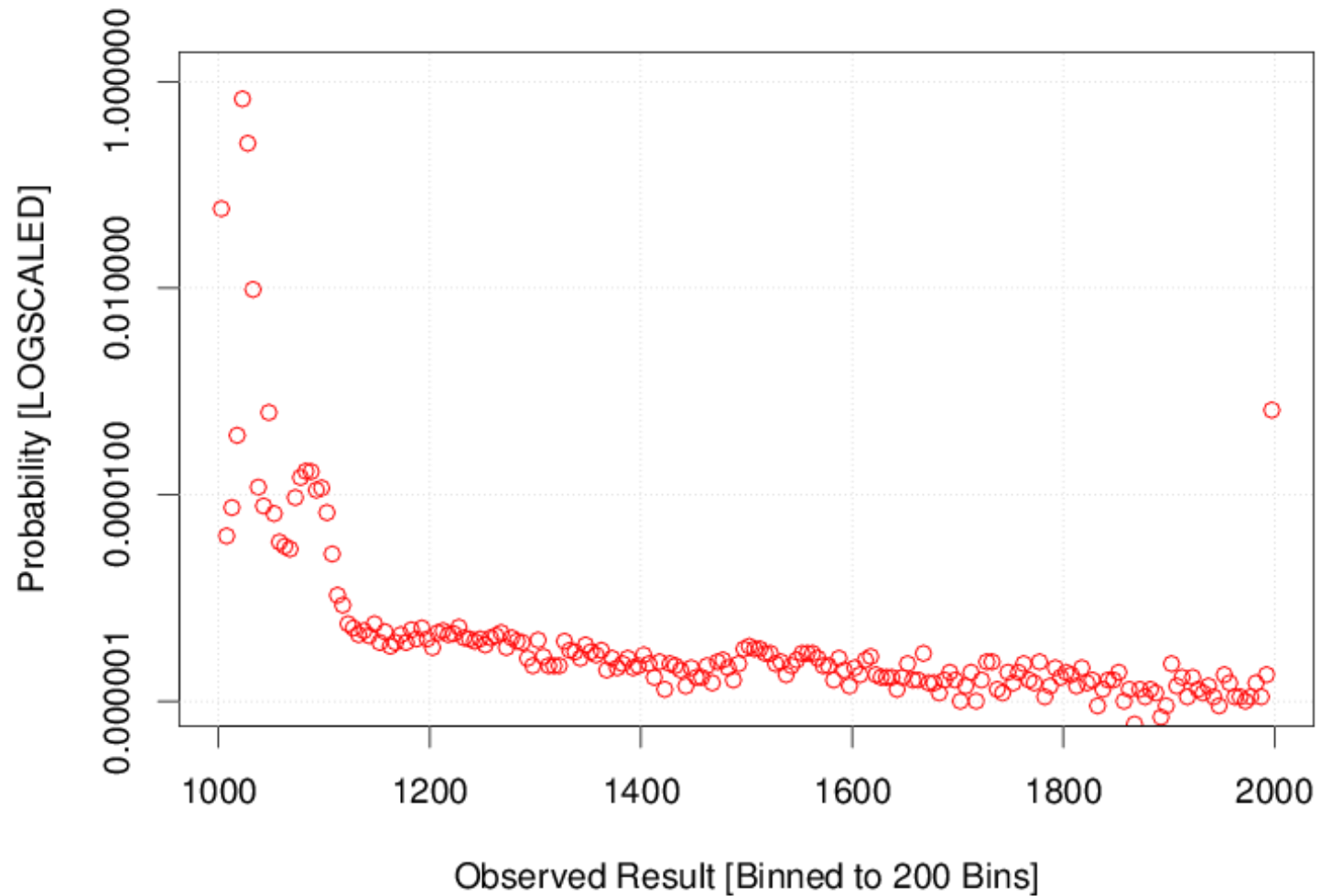
```
movl $1000, %eax // i=n=1000
movl $0, -24(%rsp) // a = 0
mfence // a is visible!
.L2:
lock addl $1, -24(%rsp) // (*a)++
subl $1, %eax // i--
jne .L2 // loop if i>0
```

Some Statistics

- **Nondeterministic execution**
 - Result depends on timing (probably not desired)
- **What do you think are the most significant results?**
 - Running two threads on Core i5 dual core
 - a=1000? 2000? 1500? 1223? 1999?

```
const int n=1000;  
volatile int a=0;  
for (int i=0; i<n; ++i)  
    a++;
```

Some Statistics



Conflicting Accesses

- (recap) two memory accesses conflict if they can happen at *the same time* (in happens-before) and one of them is a write (store)
- Such a code is said to have a “race condition”
 - Also data-race
 - Trivia around races:
 - The Therac-25 killed three people due to a race*
 - A data-race led to a large blackout in 2003, leaving 55 million people without power causing \$1bn damage*
- Can be avoided by critical regions
 - Mutually exclusive access to a set of operations



Mutual Exclusion

■ Control access to a critical region

- Memory accesses of all processes happen in program order (a partial order, many interleavings)

An execution history defines a total order of memory accesses

- Some subsets of memory accesses (issued by the same process) need to happen **atomically** (thread a's memory accesses may **not** be **interleaved** with other thread's accesses)

To achieve linearizability!

We need to restrict the valid executions

■ → Requires synchronization of some sort

- Many possible techniques (e.g., TM, CAS, T&S, ...)
- We discuss locks which have wait semantics

```
movl    $1000, %eax    // i=1000
.L2:
    movl (%rdx), %ecx    // ecx = *a
    addl $1, %ecx        // ecx++
    subl $1, %eax        // i--
    movl %ecx, (%rdx)    // *a = ecx
    jne  .L2             // loop if i>0
                        [sub sets ZF]
```


Fixing it with locks

```
const int n=1000;
volatile int a=0;
omp_lock_t lck;
for (int i=0; i<n; ++i) {
    omp_set_lock(&lck);
    a++;
    omp_unset_lock(&lck);
}
```



gcc -O3

```
    movl  $1000,%ebx    // i=1000
.L2:
    movq  0(%rbp),%rdi  // (SystemV CC)
    call  omp_set_lock  // get lock
    movq  0(%rbp),%rdi  // (SystemV CC)
    movl  (%rax),%edx   // edx = *a
    addl  $1,%edx       // edx++
    movl  %edx,(%rax)   // *a = edx
    call  omp_unset_lock // release lock
    subl  $1,%ebx      // i--
    jne   .L2           // repeat if i>0
```

- **What must the functions lock and unlock guarantee?**
 - **#1: prevent two threads from simultaneously entering CR**
i.e., accesses to CR must be mutually exclusive!
 - **#2: ensure consistent memory**
i.e., stores must be globally visible before new lock is granted!

Lock Overview

- **Lock/unlock or acquire/release**

- Lock/acquire: **before** entering CR
- Unlock/release: **after** leaving CR

- **Semantics:**

- Lock/unlock pairs have to match
- Between lock/unlock, a thread **holds** the lock

Lock Properties

- **Mutual exclusion**
 - Only one thread is on the critical region
- **Consistency**
 - Memory operations are visible when critical region is left
- **Progress**
 - If any thread a is not in the critical region, it cannot prevent another thread b from entering
- **Starvation-freedom (implies deadlock-freedom)**
 - If a thread is requesting access to a critical region, then it will eventually be granted access
- **Fairness**
 - A thread a requested access to a critical region before thread b. Did it also get access to this region before b?
- **Performance**
 - Scaling to large numbers of contending threads

Simplified Notation (cf. Histories)

- **Time defined by precedence (a total order on events)**
 - Events are instantaneous (linearizable)
 - Threads produce sequences of events a_0, a_1, a_2, \dots
 - Program statements may be repeated, denote i -th instance of a as a^i
 - Event a occurs before event b : $a \rightarrow b$
 - An interval (a, b) is the duration between events $a \rightarrow b$
 - Interval $I_1=(a, b)$ precedes interval $I_2=(c, d)$ iff $b \rightarrow c$
- **Critical regions**
 - A critical region CR is an interval $a \rightarrow b$, where a is the first operation in the CR and b the last
- **Mutual exclusion**
 - Critical regions CR_A and CR_B are mutually exclusive if:
Either $CR_A \rightarrow CR_B$ or $CR_B \rightarrow CR_A$ for all instances!
- **Assume atomic registers (for now)**

Simple Two-Thread Locks

- A first simple spinlock

```
volatile int flag=0;
```

```
void lock(lock) {  
    while(flag);  
    flag = 1;  
}
```

```
void unlock(lock) {  
    flag = 0;  
}
```

**Busy-wait to acquire lock
(spinning)**

Is this lock correct?

**Why does this not guarantee
mutual exclusion?**

Proof Intuition

- **Construct a sequentially consistent history that permits both processes to enter the CR**

Simple Two-Thread Locks

- Another two-thread spin-lock: LockOne

```
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

**When and why does this
guarantee mutual exclusion?**

Correctness Proof

- **In sequential consistency!**
- **Intuitions:**
 - Situation: both threads are ready to enter
 - Show that situation that allows both to enter leads to a schedule violating sequential consistency

Using transitivity of program and synchronization orders

Simple Two-Thread Locks

- Another two-thread spin-lock: LockOne

```
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

When and why does this guarantee mutual exclusion?

Does it work in practice?

Simple Two-Thread Locks

- A third attempt at two-thread locking: LockTwo

```
volatile int victim;  
  
void lock() {  
    victim = tid; // grant access  
    while (victim == tid) {} // wait  
}  
  
void unlock() {}
```

**Does this guarantee
mutual exclusion?**

Correctness Proof

- **Intuition:**
 - Victim is only written once per lock()
 - A can only enter after B wrote
 - B cannot enter in any sequentially consistent schedule

Simple Two-Thread Locks

- A third attempt at two-thread locking: LockTwo

```
volatile int victim;  
  
void lock() {  
    victim = tid; // grant access  
    while (victim == tid) {} // wait  
}  
  
void unlock() {}
```

**Does this guarantee
mutual exclusion?**

Does it work in practice?

Simple Two-Thread Locks

- **The last two locks provide mutual exclusion**
 - LockOne succeeds iff lock attempts do not overlap
 - LockTwo succeeds iff lock attempts do overlap
- **Combine both into one locking strategy!**
 - Peterson's lock (1981)

Peterson's Two-Thread Lock (1981)

- Combines the first lock (request access) with the second lock (grant access)

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1;    // I'm interested
    victim = tid;    // other goes first
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

Proof Correctness

- **Intuition:**
 - Victim is written once
 - Pick thread that wrote victim last
 - Show thread must have read $\text{flag}==0$
 - Show that no sequentially consistent schedule permits that

Starvation Freedom

- **(recap) definition: Every thread that calls lock() eventually gets the lock.**
 - Implies deadlock-freedom!
- **Is Peterson's lock starvation-free?**

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1;    // I'm interested
    victim = tid;    // other goes first
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```


Proof Starvation Freedom

■ Intuition:

- Threads can only wait/starve in while()
Until flag==0 or victim==other
- Other thread enters lock() → sets victim to other
Will definitely “unstuck” first thread
- So other thread can only be stuck in lock()
Will wait for victim==other, victim cannot block both threads → one must leave!

Peterson in Practice ... on x86

- Implement and run our little counter on x86
- 100000 iterations
 - $1.6 \cdot 10^{-6}$ % errors
 - What is the problem?

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1;    // I'm interested
    victim = tid;    // other goes first
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice ... on x86

- Implement and run our little counter on x86
- 100000 iterations

- $1.6 \cdot 10^{-6}$ % errors
- What is the problem?

No sequential consistency for $W(v)$ and $R(flag[j])$

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1;    // I'm interested
    victim = tid;    // other goes first
    asm ("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice ... on x86

- Implement and run our little counter on x86
- 100000 iterations

- $1.6 \cdot 10^{-6}\%$ errors
- What is the problem?

No sequential consistency for $W(v)$ and $R(flag[j])$

- Still $1.3 \cdot 10^{-6}\%$ Why?

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1;    // I'm interested
    victim = tid;    // other goes first
    asm ("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice ... on x86

- Implement and run our little counter on x86
- 100000 iterations

- $1.6 \cdot 10^{-6}\%$ errors
- What is the problem?

No sequential consistency for $W(v)$ and $R(flag[j])$

- Still $1.3 \cdot 10^{-6}\%$ Why?

Reads may slip into CR!

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1;    // I'm interested
    victim = tid;    // other goes first
    asm ("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    asm ("mfence");
    flag[tid] = 0; // I'm not interested
}
```

Correct Peterson Lock on x86

- Unoptimized (naïve sprinkling of mfences)

- Performance:

- No mfence
375ns
- mfence in lock
379ns
- mfence in unlock
404ns
- Two mfence
427ns (+14%)

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1;    // I'm interested
    victim = tid;    // other goes first
    asm ("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    asm ("mfence");
    flag[tid] = 0; // I'm not interested
}
```

Locking for N threads

- **Simple generalization of Peterson's lock, assume n levels $l = 0 \dots n-1$**
 - Is it correct?

```
volatile int level[n] = {0,0,...,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
    for (int i = 1; i < n; i++) { //attempt level i
        level[tid] = i;
        victim[i] = tid;
        // spin while conflicts exist
        while (( $\exists k \neq tid$ ) (level[k] >= i && victim[i] == tid )) {}
    }
}

void unlock() {
    level[tid] = 0;
}
```

Filter Lock - Correctness

- **Lemma: For $0 < j < n-1$, there are at most $n-j$ threads at level j !**
- **Intuition:**
 - Recursive proof (induction on j)
 - By contradiction, assume $n-j+1$ threads at level $j-1$ and j
 - Assume last thread to write victim
 - Any other thread writes level before victim
 - Last thread will stop at spin due to other thread's write
- **$j=n-1$ is critical region**

Locking for N threads

- **Simple generalization of Peterson's lock, assume n levels $l = 0 \dots n-1$**
 - Is it starvation-free?

```
volatile int level[n] = {0,0,...,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
    for (int i = 1; i < n; i++) { //attempt level i
        level[tid] = i;
        victim[i] = tid;
        // spin while conflicts exist
        while (( $\exists k \neq tid$ ) (level[k] >= i && victim[i] == tid )) {}
    }
}

void unlock() {
    level[tid] = 0;
}
```

Filter Lock Starvation Freedom

■ Intuition:

- Inductive argument over j (levels)
- Base-case: level $n-1$ has one thread (not stuck)
- Level j : assume thread is stuck

Eventually, higher levels will drain (induction)

Last entering thread is victim, it will wait

Thus, only one thread can be stuck at each level

Victim can only have one value \rightarrow older threads will advance!

Filter Lock

- What are the disadvantages of this lock?

```
volatile int level[n] = {0,0,...,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
    for (int i = 1; i < n; i++) { // attempt level i
        level[tid] = i;
        victim[i] = tid;
        // spin while conflicts exist
        while (( $\exists k \neq tid$ ) (level[k] >= i && victim[i] == tid )) {};
    }
}

void unlock() {
    level[tid] = 0;
}
```

Lock Fairness

- Starvation freedom provides no guarantee on how long a thread waits or if it is “passed”!
- To reason about fairness, we define two sections of each lock algorithm:
 - Doorway D (bounded # of steps)
 - **Waiting W (unbounded # of steps)**
- **FIFO locks:**
 - If T_A finishes its doorway before T_B the $CR_A \rightarrow CR_B$
 - Implies fairness

```
void lock() {  
    int j = 1 - tid;  
    flag[tid] = true; // I'm interested  
    victim = tid;    // other goes first  
    while (flag[j] && victim == tid) {}  
}
```

Lamport's Bakery Algorithm (1974)

- Is a FIFO lock (and thus fair)
- Each thread takes number in doorway and threads enter in the order of their number!

```
volatile int flag[n] = {0,0,...,0};
volatile int label[n] = {0,0,...,0};

void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
    while (( $\exists k \neq tid$ )(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}

public void unlock() {
    flag[tid] = 0;
}
```

Lamport's Bakery Algorithm

- **Advantages:**

- Elegant and correct solution
- Starvation free, even FIFO fairness

- **Not used in practice!**

- Why?
- Needs to read/write N memory locations for synchronizing N threads
- Can we do better?

Using only atomic registers/memory

A Lower Bound to Memory Complexity

- Theorem 5.1 in [1]: *“If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress [deadlock-freedom], then S must have at least as many variables as processes”*
- **So we’re doomed! Optimal locks are available and they’re fundamentally non-scalable. Or not?**

[1] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993

Hardware Support?

■ Hardware atomic operations:

■ Test&Set

Write const to memory while returning the old value

■ Atomic swap

Atomically exchange memory and register

■ Fetch&Op

Get value and apply operation to memory location

■ Compare&Swap

Compare two values and swap memory with register if equal

■ Load-linked/Store-Conditional LL/SC

Loads value from memory, allows operations, commits only if no other updates committed → mini-TM

■ Intel TSX (transactional synchronization extensions)

Hardware-TM (roll your own atomic operations)