Consistency
Transactions
Transactional Memory

Chris Rossbach

cs378h

Outline for Today

- Questions?
- Administrivia
  - Comments on Lab 2 due date
  - Comments on the changes to schedule
- Agenda
  - Consistency
  - Transactions
  - Transactional Memory

- Acks: Yoav Cohen for some STM slides

Faux Quiz questions

- How are promises and futures related? Since there is disagreement on the nomenclature, don’t worry about which is which—just describe what the different objects are and how they function.
- How does HTM resemble or differ from Load-linked Stored-Conditional?
- What are some pros and cons of HTM vs STM?
- What is Open Nesting? Closed Nesting? Flat Nesting?
- How does 2PL differ from 2PC?
- Define ACID properties: which, if any, of these properties does TM relax?

Memory Consistency

Memory Consistency

- Formal specification of memory semantics
  - Statement of how shared memory will behave with multiple CPUs
  - Ordering of reads and writes

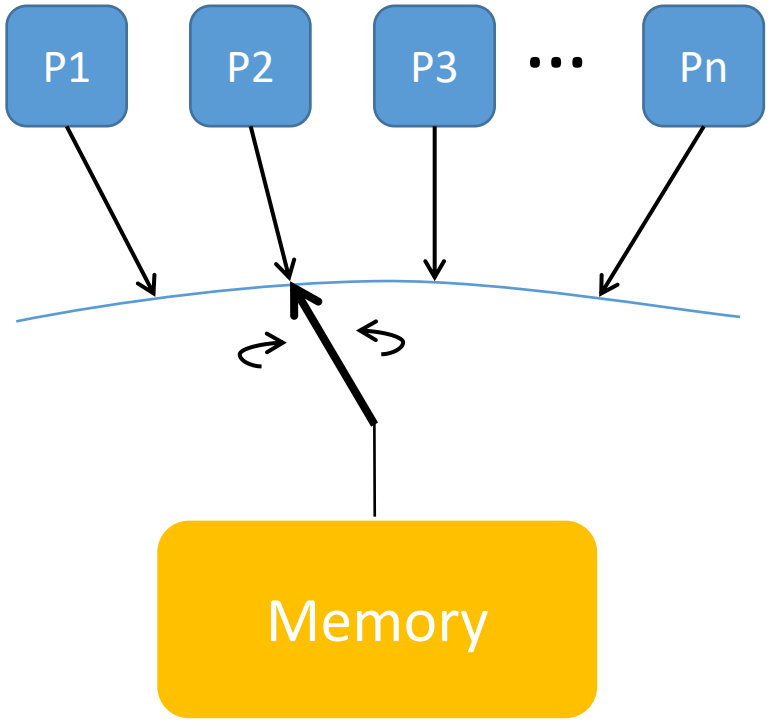
Memory Consistency

- Formal specification of memory semantics
  - Statement of how shared memory will behave with multiple CPUs
  - Ordering of reads and writes

- Memory Consistency != Cache Coherence
  - Coherence: propagate updates to cached copies
    - Invalidate vs. Update
  - Coherence vs. Consistency?
    - **Coherence:** ordering of ops. at a single location
    - **Consistency:** ordering of ops. at multiple locations

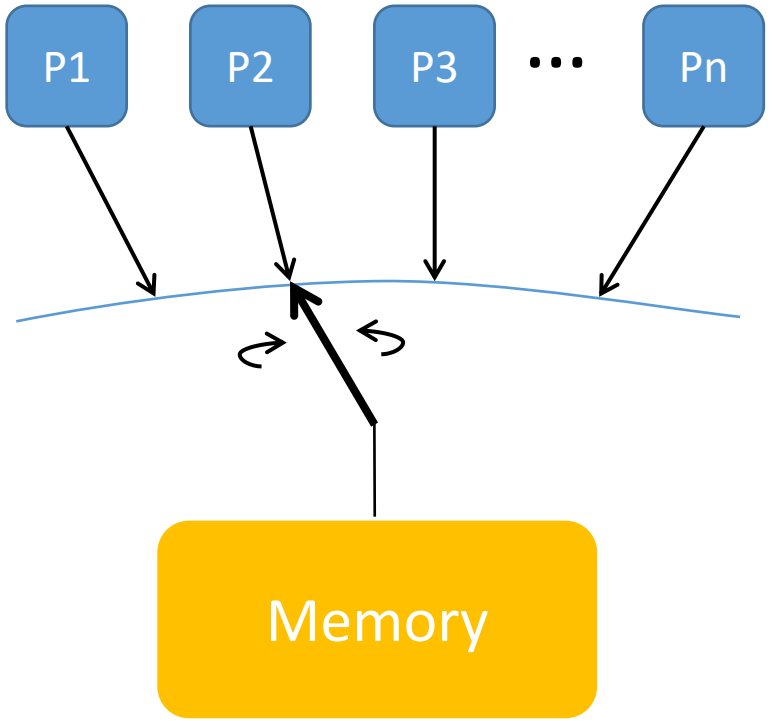
Sequential Consistency

Sequential Consistency



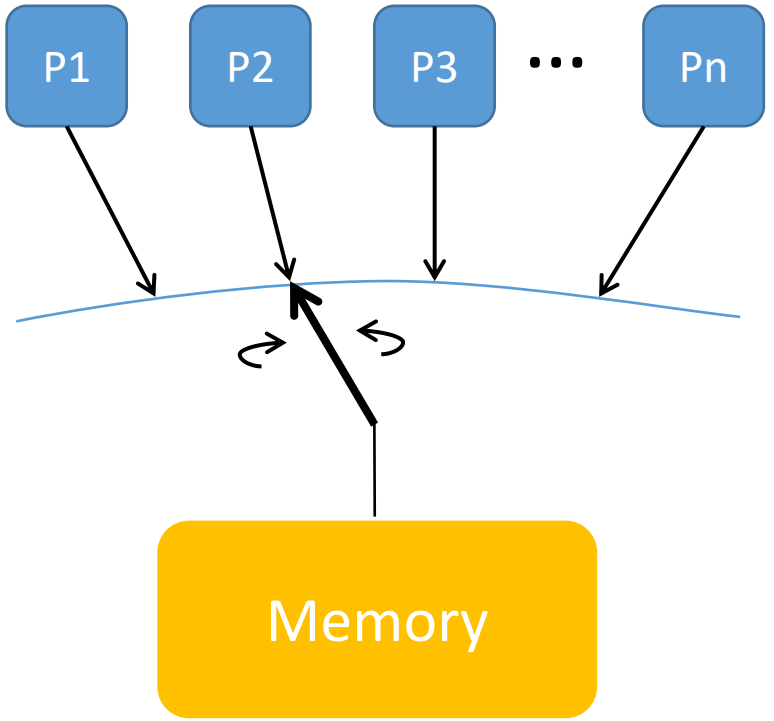
Sequential Consistency

- Result of *any* execution is same as if all operations execute on a uniprocessor



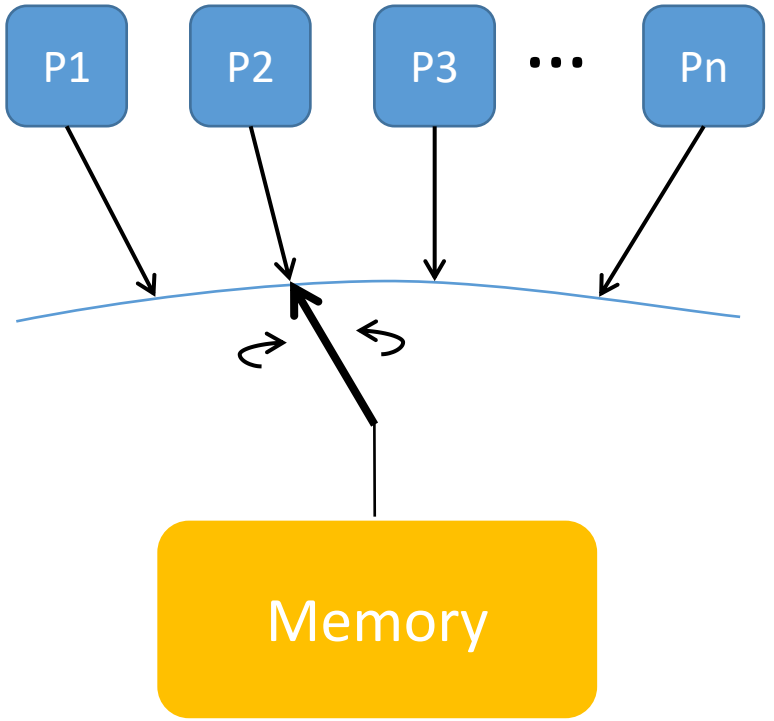
Sequential Consistency

- Result of *any* execution is same as if all operations execute on a uniprocessor
- Operations on each processor are *totally ordered* in the sequence and respect program order for each processor



Sequential Consistency

- Result of *any* execution is same as if all operations execute on a uniprocessor
- Operations on each processor are *totally ordered* in the sequence and respect program order for each processor

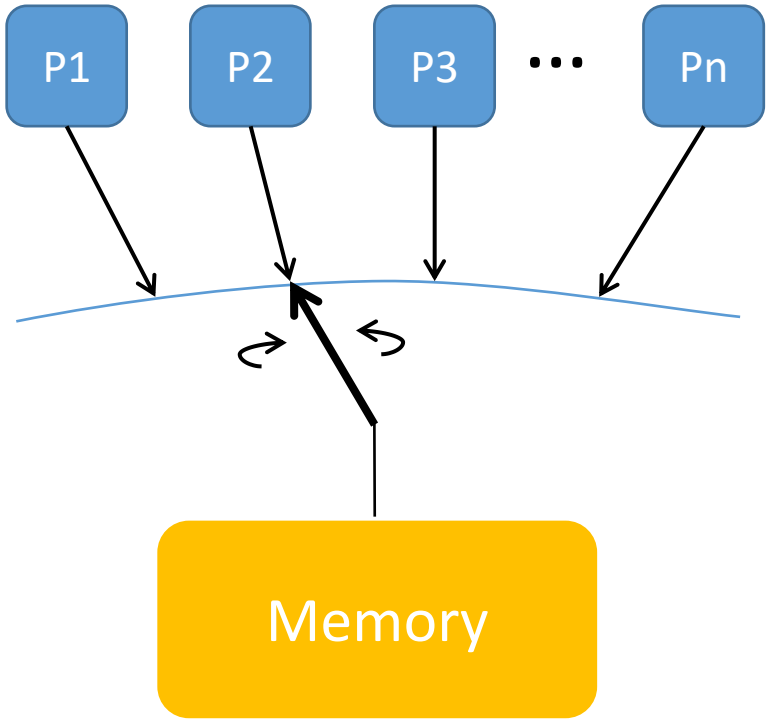


Trying to mimic Uniprocessor semantics:

- Memory operations occur:
  - One at a time
  - In program order
- Read returns value of last write

Sequential Consistency

- Result of *any* execution is same as if all operations execute on a uniprocessor
- Operations on each processor are *totally ordered* in the sequence and respect program order for each processor



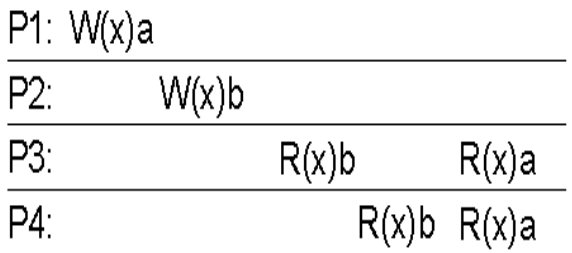
Trying to mimic Uniprocessor semantics:

- Memory operations occur:
  - One at a time
  - In program order
- Read returns value of last write

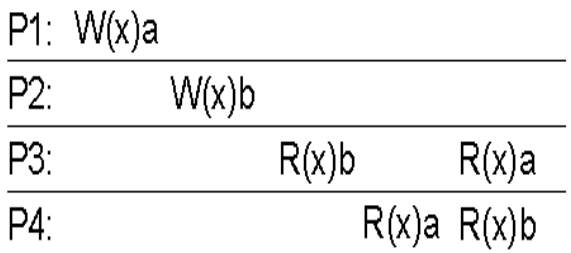
- How is this different from coherence?
- Why do modern CPUs not implement SC?
- Requirements: ***program order, write atomicity***

Sequential Consistency

- All operations are executed in *some* sequential order
- each process issues operations in program order
  - *Any* valid interleaving is allowed
  - All *agree* on the same interleaving
  - Each process preserves its program order



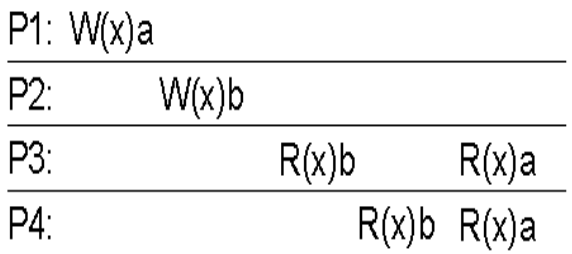
(a)



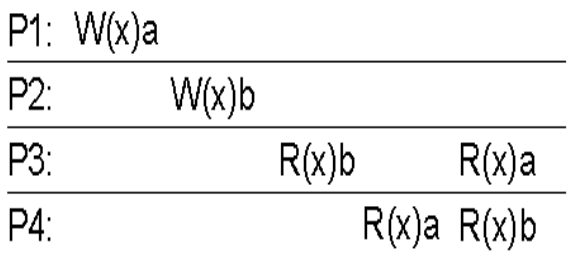
(b)

Sequential Consistency

- All operations are executed in *some* sequential order
- each process issues operations in program order
  - *Any* valid interleaving is allowed
  - All *agree* on the same interleaving
  - Each process preserves its program order



(a)



(b)

Are either of these SC?

Sequential Consistency: Canonical Example

```
Initially, Flag1 = Flag2 = 0
```

P1

```
Flag1 = 1
if (Flag2 == 0)
  enter CS
```

P2

```
Flag2 = 1
if (Flag1 == 0)
  enter CS
```

Sequential Consistency: Canonical Example

```
Initially, Flag1 = Flag2 = 0
```

P1

```
Flag1 = 1
if (Flag2 == 0)
  enter CS
```

P2

```
Flag2 = 1
if (Flag1 == 0)
  enter CS
```

Can both P1 and P2 wind up in the critical section at the same time?

Do we need Sequential Consistency?

Initially, Flag1 = Flag2 = 0

P1	**P2**
Flag1 = 1	
	Flag2 = 1
	if (Flag1 == 0)
	shared_data++
if (Flag2 == 0)	
shared_data++	

Do we need Sequential Consistency?

Initially, Flag1 = Flag2 = 0

P1	**P2**
Flag1 = 1	
	Flag2 = 1
	if (Flag1 == 0)
	shared_data++
if (Flag2 == 0)	
shared_data++	

Key issue:

- P1 and P2 may not see each other's writes in the same order
- Implication: both in critical section, which is incorrect
- Why would this happen?

Do we need Sequential Consistency?

Initially, Flag1 = Flag2 = 0

P1

Flag1 = 1

if (Flag2 == 0)

shared_data++

P2

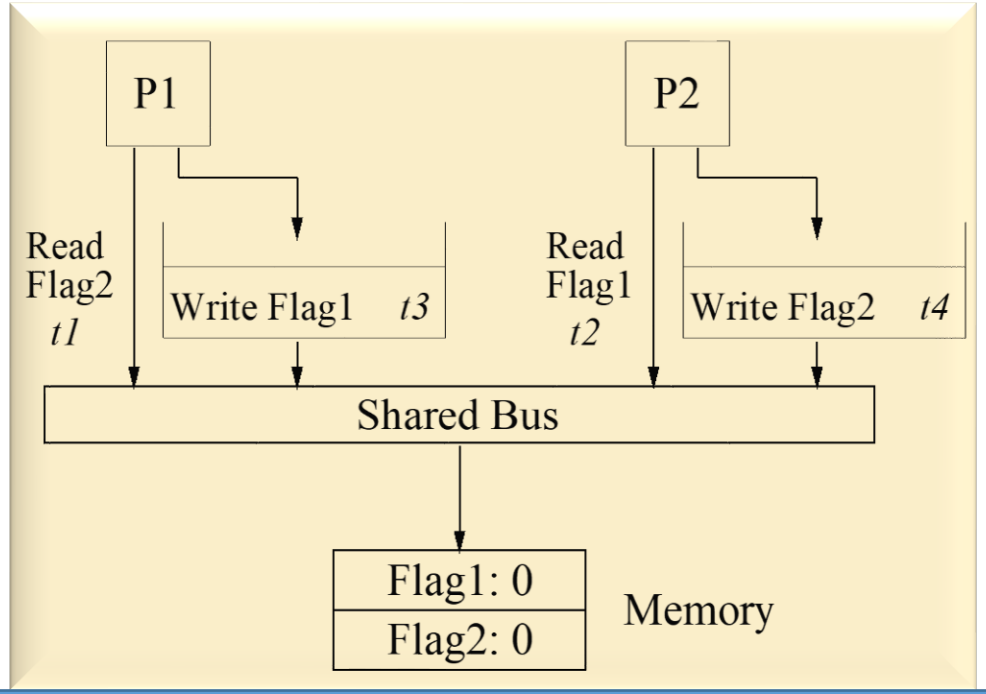
Flag2 = 1

if (Flag1 == 0)

shared_data++

Key issue:

- P1 and P2 may not see each other's writes in the same order
- Implication: both in critical section, which is incorrect
- Why would this happen?



Write Buffers

- P_0 write → queue op in write buffer, proceed
- P_0 read → look in write buffer,
- P_(x != 0) read → old value: write buffer hasn't drained

Requirements for Sequential Consistency

Requirements for Sequential Consistency

- *Program Order*
  - Processor's memory operations must complete in program order

Requirements for Sequential Consistency

- *Program Order*
  - Processor's memory operations must complete in program order
- *Write Atomicity*
  - Writes to the same location seen by all other CPUs
  - Subsequent reads must not return value of a write until propagated to all

Requirements for Sequential Consistency

- *Program Order*
  - Processor's memory operations must complete in program order
- *Write Atomicity*
  - Writes to the same location seen by all other CPUs
  - Subsequent reads must not return value of a write until propagated to all
- Write acknowledgements are necessary
  - Cache coherence provides these properties for a cache-only system

Requirements for Sequential Consistency

- *Program Order*
  - Processor's memory operations must complete in program order
- *Write Atomicity*
  - Writes to the same location seen by all other CPUs
  - Subsequent reads must not return value of a write until propagated to all
- Write acknowledgements are necessary
  - Cache coherence provides these properties for a cache-only system

Disadvantages:

- Difficult to implement!
  - Coherence to (e.g.) write buffers is hard
- Sacrifices many potential optimizations
  - Hardware (cache) and software (compiler)
  - Major performance hit

Relaxed Consistency Models

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's Write early

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's Write early
- *Requirement:* synchronization primitives for safety
  - Fence, barrier instructions etc

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's V
- *Requirement:* synchronization pri
  - Fence, barrier instructions etc

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Relaxed Consis

```
static inline void arch_write_lock(arch_rwlock_t *rw) {
   asm volatile(LOCK_PREFIX WRITE_LOCK_SUB(%1) "(%0)\n\t"
        "jz 1f\n"
         "call __write_lock_failed\n\t"
        "1:\n"
        ::LOCK_PTR_REG (&rw->write), "i" (RW_LOCK_BIAS) : "memory"); }
```

x86

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's V
- *Requirement:* synchronization pri
  - Fence, barrier instructions etc

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's V
- *Requirement:* synchronization pri
  - Fence, barrier instructions etc

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W

ons

cessor's V

tion pri

etc

```
static inline unsigned long
__arch_spin_trylock(arch_spinlock_t *lock)
{
  unsigned long tmp, token;
  token = LOCK_TOKEN;
  __asm__ __volatile__(
    "1: "  PPC_LWARX(%0,0,%2,1) "\n\
           cmpwi 0,%0,0\n\
           bne- 2f\n\
           stwcx. %1,0,%2\n\
           bne- 1b\n"
           PPC_ACQUIRE_BARRIER
    "2:“ : "=&r" (tmp)
         : "r" (token), "r" (&lock->slock)
         : "cr0", "memory");
    return tmp;
}
```

PowerPC

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Relaxed Consistency Models

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's V
- *Requirement:* synchronization pri
  - Fence, barrier instructions etc

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Transactions and Transactional Memory

Transactions and Transactional Memory

- 3 Programming Model Dimensions:

Transactions and Transactional Memory

- 3 Programming Model Dimensions:
  - How to specify computation

Transactions and Transactional Memory

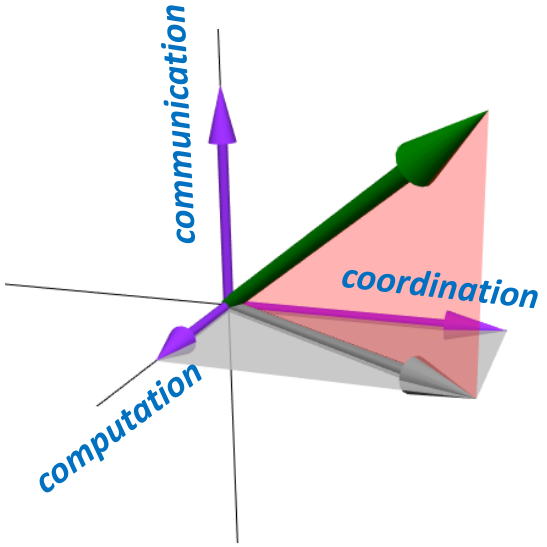
- 3 Programming Model Dimensions:
  - How to specify computation
  - How to specify communication

Transactions and Transactional Memory

- 3 Programming Model Dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer

Transactions and Transactional Memory

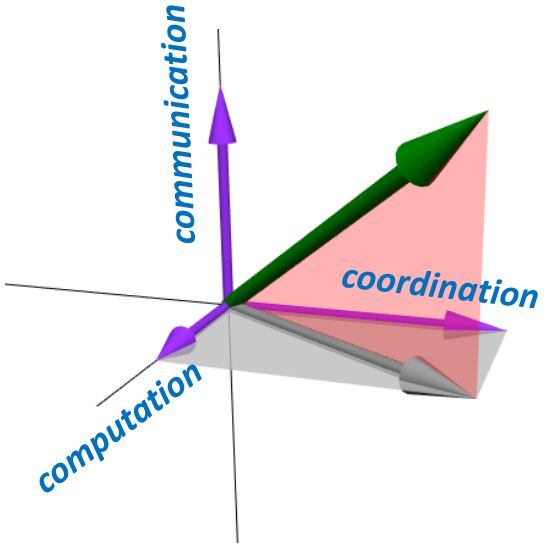
- 3 Programming Model Dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer



Transactions and Transactional Memory

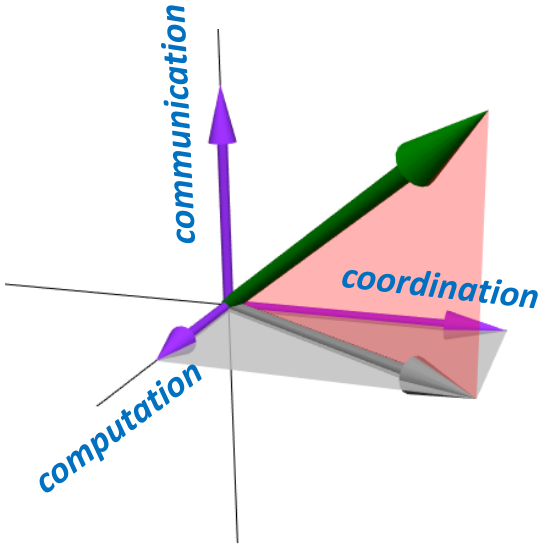
- 3 Programming Model Dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer

- Threads, Futures, Events etc.
  - *Mostly about how to express control*



Transactions and Transactional Memory

- 3 Programming Model Dimensions:
  - How to specify computation
  - How to specify communication
  - How to specify coordination/control transfer
- Threads, Futures, Events etc.
  - *Mostly about how to express control*
- Transactions
  - *Mostly about how to deal with shared state*



Transactions

Core issue: multiple updates

Canonical examples:

```
move(file, old-dir, new-dir) {
   delete(file, old-dir)
   add(file, new-dir)
}
```

```
create(file, dir) {
   alloc-disk(file, header, data)
   write(header)
   add (file, dir)
}
```

Transactions

Core issue: multiple updates

Canonical examples:

```
move(file, old-dir, new-dir) {
   delete(file, old-dir)
   add(file, new-dir)
}
```

```
create(file, dir) {
   alloc-disk(file, header, data)
   write(header)
   add (file, dir)
}
```

- Modified data in memory/caches
- Even if in-memory data is durable, multiple disk updates

Transactions

Core issue: multiple updates

Canonical examples:

```
move(file, old-dir, new-dir) {
   delete(file, old-dir)
   add(file, new-dir)
}
```

```
create(file, dir) {
   alloc-disk(file, header, data)
   write(header)
   add (file, dir)
}
```

Problems: crash in the middle / visibility of intermediate state

- Modified data in memory/caches
- Even if in-memory data is durable, multiple disk updates

Problem: Unreliability

Problem: Unreliability

- Want reliable update of two resources (e.g. in two disks, machines…)
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA account)
  - Move directory from server A to B

Problem: Unreliability

- Want reliable update of two resources (e.g. in two disks, machines…)
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA account)
  - Move directory from server A to B
- Machines can crash, messages can be lost

Problem: Unreliability

- Want reliable update of two resources (e.g. in two disks, machines…)
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA account)
  - Move directory from server A to B
- Machines can crash, messages can be lost

Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?

Problem: Unreliability

- Want reliable update of two resources (e.g. in two disks, machines…)
  - Move file from A to B
  - Create file (update free list, inode, data block)
  - Bank transfer (move $100 from my account to VISA account)
  - Move directory from server A to B
- Machines can crash, messages can be lost

Can we use messages? E.g. with retries over unreliable medium to synchronize with guarantees?

No.
Not even if all messages get through!

General’s paradox

General’s paradox

- Two generals on separate mountains

General’s paradox

- Two generals on separate mountains
- Can only communicate via messengers

General’s paradox

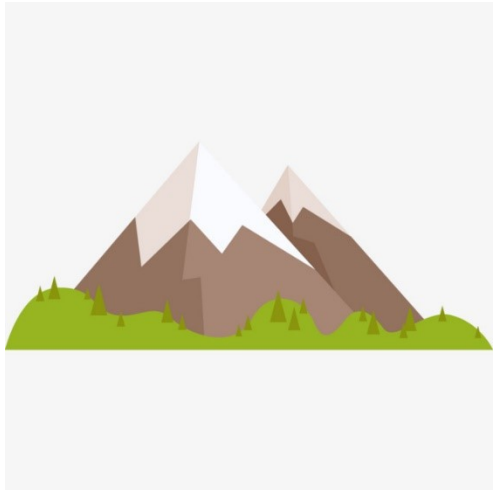
- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured

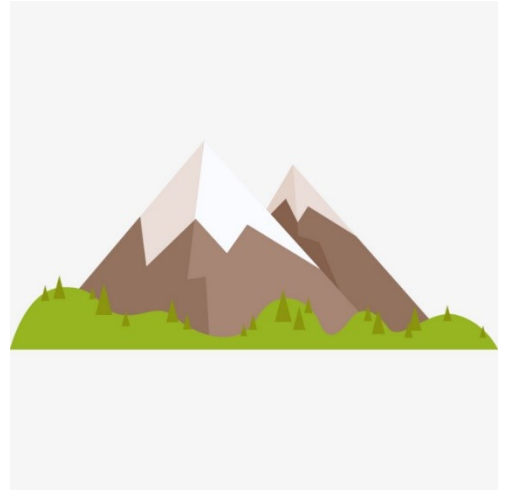
General’s paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

General’s paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!





General’s paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

General's paradox

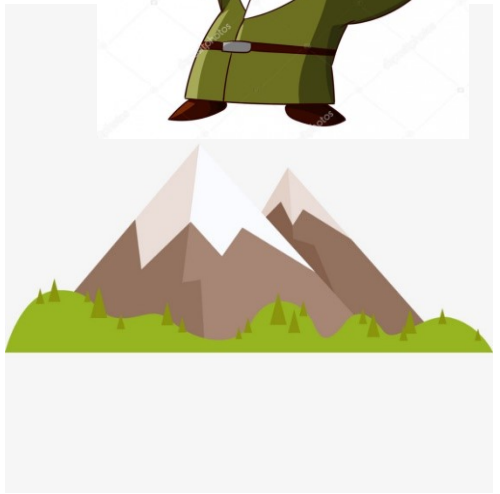
- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

General A → General B: let's attack at dawn

General's paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

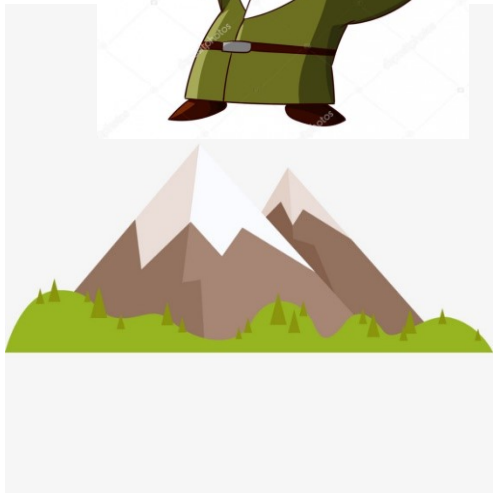
General A → General B: let's attack at dawn
General B → General A: OK, dawn.



General's paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

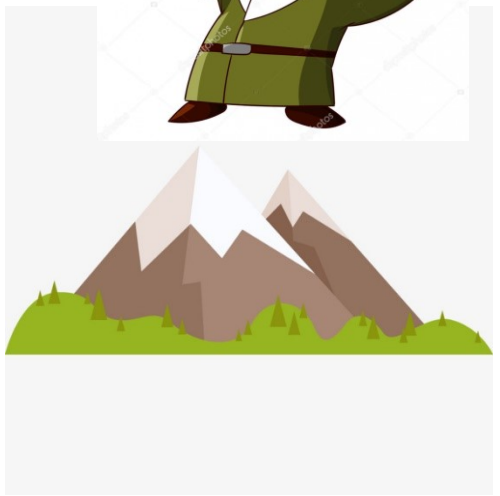
General A → General B: let's attack at dawn
General B → General A: OK, dawn.
General A → General B: Check. Dawn it is.



General's paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

General A → General B: let's attack at dawn
General B → General A: OK, dawn.
General A → General B: Check. Dawn it is.
General B → General A: Alright already—dawn.

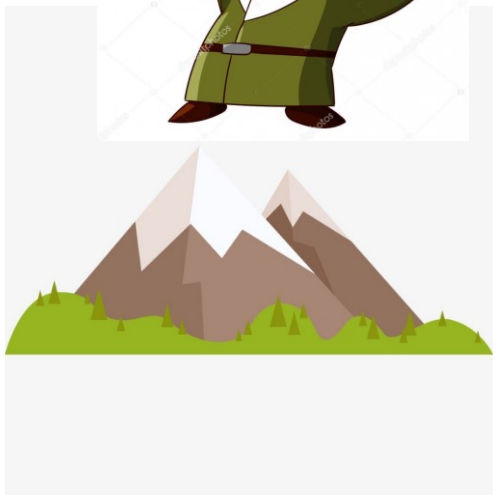


General's paradox

- Two generals on separate mountains
- Can only communicate via messengers
- Messengers can get lost or captured
- Need to coordinate attack
  - attack at same time good, different times bad!

- Even if all messages delivered, can't assume– maybe some message didn't get through.
- No solution: one of the few CS impossibility results.

General A → General B: let's attack at dawn
General B → General A: OK, dawn.
General A → General B: Check. Dawn it is.
General B → General A: Alright already—dawn.



Transactions can help

(*but can't solve it*)

Transactions can help
(*but can't solve it*)

- Solves weaker problem:
  - 2 things will either happen or not
  - not necessarily at the same time

Transactions can help
(*but can't solve it*)

- Solves weaker problem:
  - 2 things will either happen or not
  - not necessarily at the same time
- Core idea: one entity has the power to say yes or no for all
  - Local txn: one final update (TxEND) irrevocably triggers several
  - Distributed transactions
    - 2 phase commit
    - One machine has final say for all machines
    - Other machines bound to comply

Transactions can help
(*but can't solve it*)

- Solves weaker problem:
  - 2 things will either happen or not
  - not necessarily at the same time
- Core idea: one entity has the power to say yes or no for all
  - Local txn: one final update (TxEND) irrevocably triggers several
  - Distributed transactions
    - 2 phase commit
    - One machine has final say for all machines
    - Other machines bound to comply

What is the role of synchronization here?

Transactional Programming Model

```
begin transaction;
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
commit transaction;
```

Transactional Programming Model

```
begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;
```

What has changed from previous programming models?

ACID Semantics

ACID Semantics

What are they?

- A
- C
- I
- D

ACID Semantics

ACID Semantics

```
begin transaction;
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
commit transaction;
```

ACID Semantics

- Atomic – all updates happen or none do

```
begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;
```

ACID Semantics

- Atomic – all updates happen or none do
- Consistent – system invariants maintained across updates

```
begin transaction;
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
commit transaction;
```

ACID Semantics

- Atomic – all updates happen or none do
- Consistent – system invariants maintained across updates
- Isolated – no visibility into partial updates

```
begin transaction;
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
commit transaction;
```

ACID Semantics

- Atomic – all updates happen or none do
- Consistent – system invariants maintained across updates
- Isolated – no visibility into partial updates
- Durable – once done, stays done

```
begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;
```

ACID Semantics

- Atomic – all updates happen or none do
- Consistent – system invariants maintained across updates
- Isolated – no visibility into partial updates
- Durable – once done, stays done
- Are subsets ever appropriate?
  - When would ACI be useful?
  - ACD?
  - Isolation only?

```
begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;
```

Transactions: Implementation

Transactions: Implementation

- Key idea: turn multiple updates into a single one

Transactions: Implementation

- Key idea: turn multiple updates into a single one
- Many implementation Techniques
  - Two-phase locking
  - Timestamp ordering
  - Optimistic Concurrency Control
  - Journaling
  - 2,3-phase commit
  - Speculation-rollback
  - Single global lock
  - Compensating transactions

Transactions: Implementation

- Key idea: turn multiple updates into a single one
- Many implementation Techniques
  - Two-phase locking
  - Timestamp ordering
  - Optimistic Concurrency Control
  - Journaling
  - 2,3-phase commit
  - Speculation-rollback
  - Single global lock
  - Compensating transactions

Key problems:

- output commit
- synchronization

Transactions: Implementation

- Key idea: turn multiple updates into a single one
- Many implementation Techniques
  - Two-phase locking
  - Timestamp ordering
  - Optimistic Concurrency Control
  - Journaling
  - 2,3-phase commit
  - Speculation-rollback
  - Single global lock
  - Compensating transactions

Key problems:

- output commit
- synchronization

Implementing Transactions

```
BEGIN_TXN();
    x = read(“x-values”, ....);
    y = read(“y-values”, ....);
    z = x+y;
    write(“z-values”, z, ....);
COMMIT_TXN();
```

Implementing Transactions

```
BEGIN_TXN();
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
COMMIT_TXN();
```

```
BEGIN_TXN() {

}
```

```
COMMIT_TXN() {

}
```

Implementing Transactions

```
BEGIN_TXN();
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
COMMIT_TXN();
```

```
BEGIN_TXN() {
   LOCK(single-global-lock);
}
```

```
COMMIT_TXN() {
   UNLOCK(single-global-lock);
}
```

Implementing Transactions

```
BEGIN_TXN();
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
COMMIT_TXN();
```

```
BEGIN_TXN() {
   LOCK(single-global-lock);
}
```

```
COMMIT_TXN() {
   UNLOCK(single-global-lock);
}
```

Pros/Cons?

Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```

Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```

BEGIN_TXN() {

}

COMMIT_TXN() {

}

Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```

```
BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
    LOCK(x);
}
```

```
COMMIT_TXN() {
  forall x in rwset
    UNLOCK(x);
}
```

Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```

```
BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
    LOCK(x);
}
```

```
COMMIT_TXN() {
  forall x in rwset
    UNLOCK(x);
}
```

Pros/Cons?

Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```

```
BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
    LOCK(x);
}
```

```
COMMIT_TXN() {
  forall x in rwset
    UNLOCK(x);
}
```

Pros/Cons?

What happens on failures?

Two-phase locking

- Phase 1: only acquire locks in order
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```

A: grab locks
A: modify x, y,
A: unlock y, x
B: grab locks
B: update x, y
B: unlock y, x
B: COMMIT
A: CRASH

```
BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
    LOCK(x);
}
```

```
COMMIT_TXN() {
  forall x in rwset
    UNLOCK(x);
}
```

Pros/Cons?

What happens on failures?

Two-phase locking

- Phase 1: only acquire locks in
- Phase 2: unlock at commit
- avoids deadlock

```
BEGIN_TXN();
Lock x, y
x = x + 1
y = y - 1
unlock y, x
COMMIT_TXN();
```

B commits changes that depend on A's updates

A: grab locks
A: modify x, y,
A: unlock y, x
B: grab locks
B: update x, y
B: unlock y, x
B: COMMIT
A: CRASH

```
BEGIN_TXN() {
  rwset = Union(rset, wset);
  rwset = sort(rwset);
  forall x in rwset
    LOCK(x);
}
```

```
COMMIT_TXN() {
  forall x in rwset
    UNLOCK(x);
}
```

Pros/Cons?

What happens on failures?

Two-phase commit

- N participants agree or don’t (atomicity)
- Phase 1: everyone “prepares”
- Phase 2: Master decides and tells everyone to actually commit
- What if the master crashes in the middle?

2PC: Phase 1

1. Coordinator sends REQUEST to all participants
2. Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Example—move: C→S1: delete foo from /, C→S2: add foo to /

Failure case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 decides permission problem
S2 writes/sends VOTE_ABORT

Success case:
S1 writes rm /foo, VOTE_COMMIT to log
S1 sends VOTE_COMMIT
S2 writes add foo to /
S2 writes/sends VOTE_COMMIT

2PC: Phase 2

- Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Participants receive decision, write GLOBAL_* to log

2PC corner cases

Phase 1

1. Coordinator sends REQUEST to all participants
2. X Participants receive request and
3. Execute locally
4. Write VOTE_COMMIT or VOTE_ABORT to local log
5. Send VOTE_COMMIT or VOTE_ABORT to coordinator

Phase 2

- Y Case 1: receive VOTE_ABORT or timeout
  - Write GLOBAL_ABORT to log
  - send GLOBAL_ABORT to participants
- Case 2: receive VOTE_COMMIT from all
  - W Write GLOBAL_COMMIT to log
  - send GLOBAL_COMMIT to participants
- Z Participants recv decision, write GLOBAL_* to log

- What if participant crashes at X?
- Coordinator crashes at Y?
- Participant crashes at Z?
- Coordinator crashes at W?

2PC limitation(s)

2PC limitation(s)

- Coordinator crashes at W, never wakes up

2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!

2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!
- Can participants ask each other what happened?

2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!
- Can participants ask each other what happened?
- 2PC: always has risk of indefinite blocking

2PC limitation(s)

- Coordinator crashes at W, never wakes up
- All nodes block forever!
- Can participants ask each other what happened?
- 2PC: always has risk of indefinite blocking
- Solution: (yes) 3 phase commit!
  - Reliable replacement of crashed “leader”
  - 2PC often good enough in practice

Questions?