Week 6: Avoiding Locks & Scalability

Non-blocking Synchronization

Read-Copy Update

Transactional Memory

Scalability

CSC 469 / CSC 2208

Fall 2018

(with thanks to Bogdan Simion, Tom Hart, Paul McKenney)

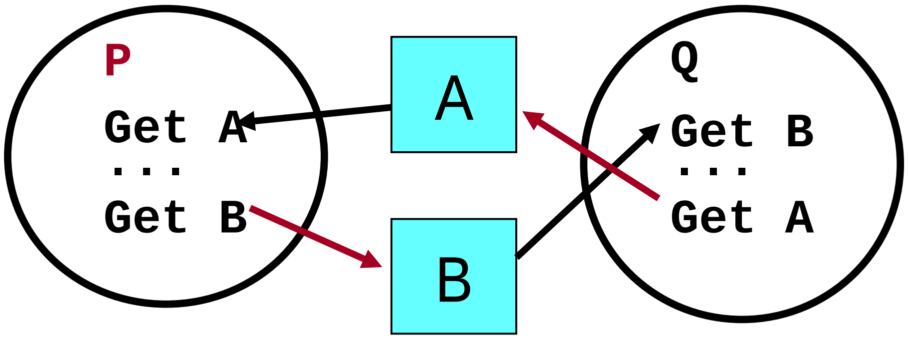
University of Toronto, Department of Computer Science

Locking: A necessary evil?

- Locks are an easy to understand solution to critical section problem
  - Protect shared data from corruption due to simultaneous updates
  - Protect against inconsistent views of intermediate states
- But locks have lots of problems
  - 1. Deadlock
  - 2. Priority inversion
  - 3. Not fault tolerant
  - 4. Convoying
  - 5. Expensive, even when uncontended
- *Not* easy to use correctly!

1. Deadlock

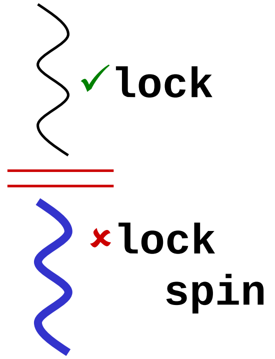
- Textbook definition: Set of threads blocked waiting for event that can only be caused by another thread in the same set
- Classic example:



- Self-deadlock also a big issue
  - Thread holds lock on shared data structure and is interrupted
  - Interrupt handler needs same lock!
- Solutions exist (e.g., specify lock order, disable interrupts while holding lock) but add complexity

2. Priority Inversion

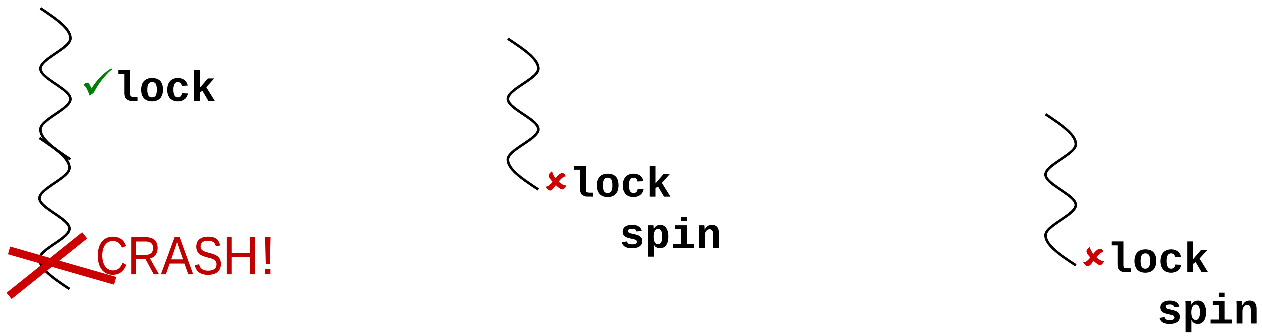
- Lower priority thread gets spinlock
- Higher priority thread becomes runnable and preempts it
  - Needs lock, starts spinning
  - Lock holder can't run and release lock



- Solutions exist (e.g. disable preemption while holding spinlock, implement priority inheritance, etc.), but add complexity

3. Not fault tolerant

- If lock holder crashes, or gets delayed, no one makes progress

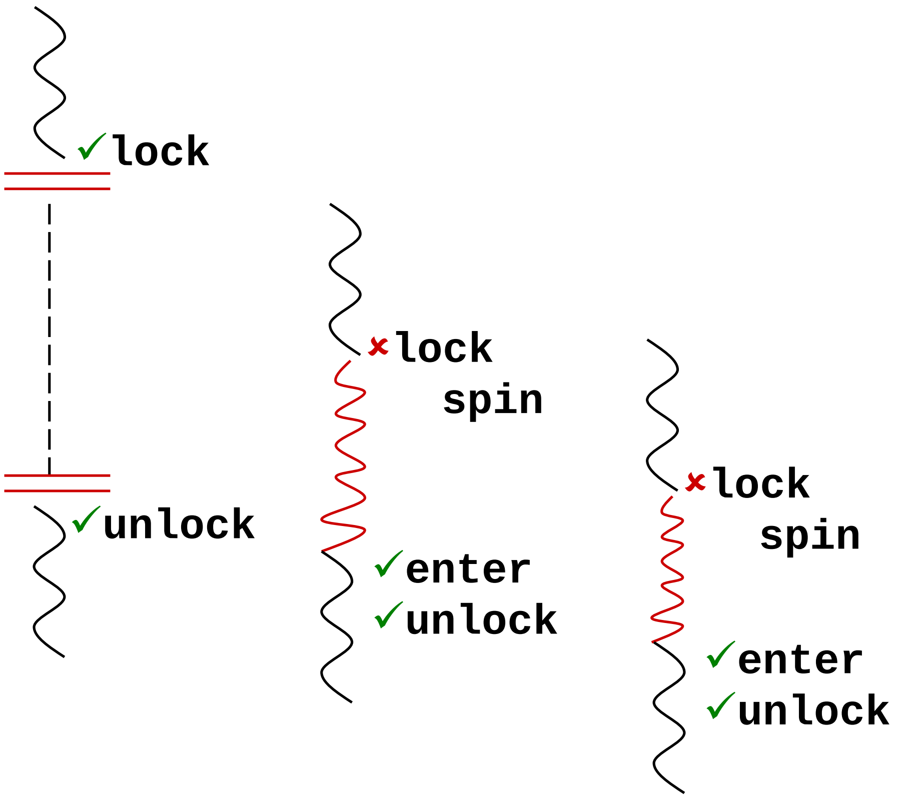


- Scheduler-conscious synchronization helps with delays (preemption, page faults)
    - Crashes require abort / restart

4. Convoying

- Threads doing similar work, started at different times, occasionally accessing shared data
  - e.g., multi-threaded web server
- Expect access to shared objects to be spread out over time
  - Lock contention should be low
- Delay of lock holder allows other threads to catch up
  - Lock becomes contended and tends to stay that way

    => Convoying

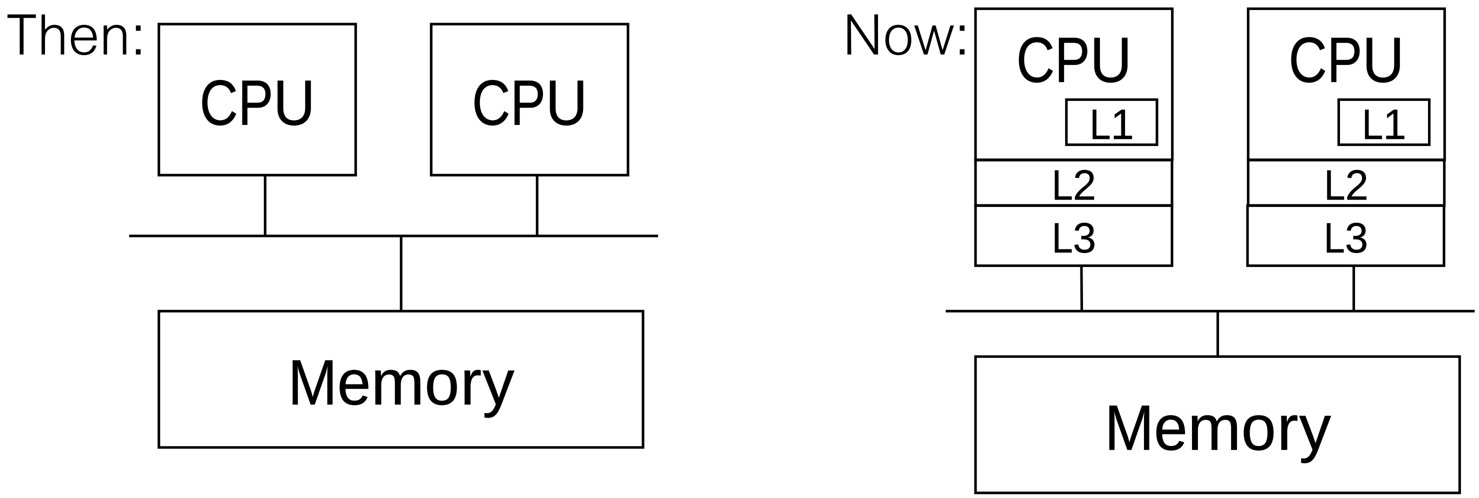


5. Expensive, even when uncontended

Operation	Nanoseconds
Instruction	0.24
Clock Cycle	0.69
Atomic Increment	42.09
Cmpxchg Blind Cache Transfer	56.80
Cmpxchg Cache Transfer and Invalidate	59.10
SMP Memory Barrier (eieio)	75.53
Full Memory Barrier (sync)	92.16
CPU-Local Lock	243.10

McKenney, 2005 – 8-CPU 1.45 GHz PPC

Causes: Deeper Memory Hierarchy



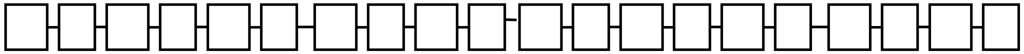
- Memory speeds have not kept up with CPU speeds
  - 1984: no caches needed, since instructions slower than memory accesses
  - after ~2005: 3-4 level cache hierarchies, since instructions orders of magnitude faster than memory accesses
- Synchronization ops typically execute at memory speed

Causes: Deeper Pipelines

Then:

Fetch — Execute — Retire

Now:



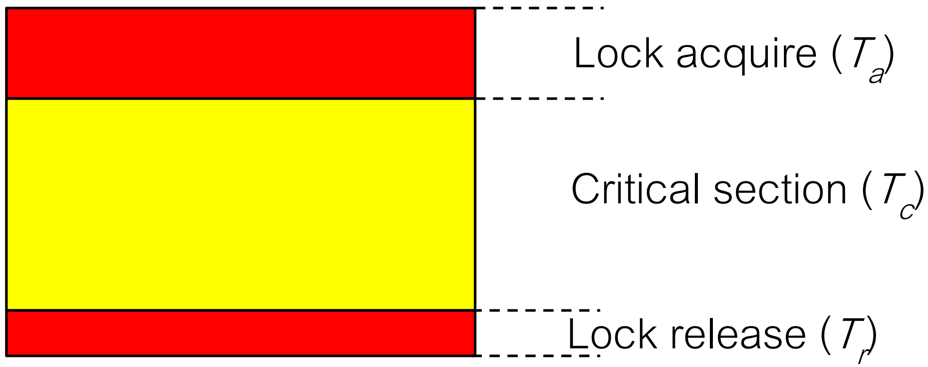
- 1984: Many cycles per instruction
- 2005: Many instructions per cycle
  - 20 stage pipelines
  - CPU logic executes instructions out-of-order to keep pipeline full
  - Synchronization instructions must not be reordered
  - => synchronization stalls the pipeline
- Deeper pipelines not always better and processors are changing

Performance

- Main issue with lock performance used to be contention
  - Techniques were developed to reduce overheads in contended case
  - And to reduce contention
- Today, issue is degraded performance even when locks are *always* available
  - Together with other concerns about locks

Critical section efficiency

- Assuming little to no contention, and no caching effects in CS



$$Efficiency = \frac{T_c}{T_c + T_a + T_r}$$

- Even if lock contention is negligible, critical section efficiency must be addressed!

Locks: ~~A necessary~~ evil?

Idea: Don’t lock if we don’t need to!

- Non-Blocking Synchronization (NBS)
  - Use term “*lockless*” to describe strategies that avoid locking

NBS Basics

- Make change optimistically, roll back and retry if conflict detected

```
atomic_inc(int *counter) {
      int value;
      do {
            value = *counter;
      } while (!CAS(counter, value, value+1);
}
```

- Complex updates (e.g. modifying multiple values in a structure) are hidden behind a single commit point using atomic instructions

Example: Stack Data Structure

- Lock-based synchronization:

```
/* definitions */

typedef struct node_s
{
  int val;
  struct node_s *next;
} node_t;

typedef struct stack_s
{
  node_t *top;
  lock_t *stack_lock;
} stack_t;
```

```
void push(stack_t *S, node_t *n)
{
  lock(S->stack_lock);
  n->next = S->top; S->top=n;
  unlock(S->stack_lock);
}
node_t* pop(stack_t *S){
    node_t *n = NULL;
    lock(S->stack_lock);
    if (S->top != NULL) {
      n = S->top;
      S->top = S->top->next;
    }
    unlock(S->stack_lock);
    return n;
}
```

Non-blocking stack (take 1)

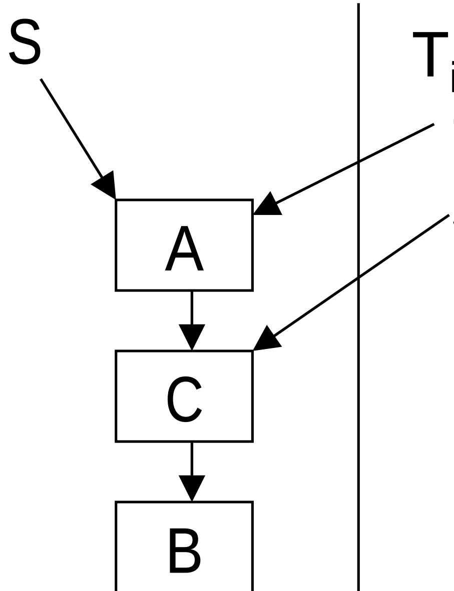
```
/* definitions */

typedef struct node_s {
  int val;
  struct node_s *next;
} node_t;

/* Stack type is just
 * a pointer to a node.
 */
typedef
     node_t* stack_t;
```

What's wrong?

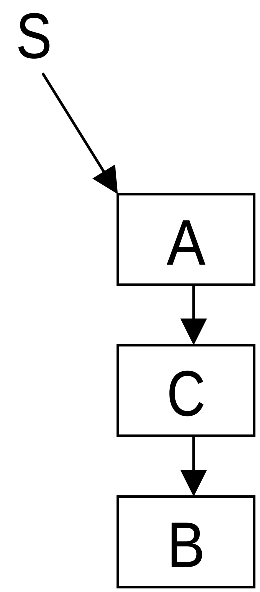
```
void push(stack_t *S, node_t *n)
{
  node_t *first;
  do {
    first = *S;
    n->next = first;
  } while (!CAS(S,first,n));
}
node_t* pop(stack_t *S) {
  node_t *first, *second;
  do {
    first = *S;
    if (first != NULL) {
      second = first->next;
    } else return NULL;
  } while (!CAS(S,first,second));
  return first;
}
```

ABA Problem

- T_i, T_j both doing pops and pushes, interleaved as follows:



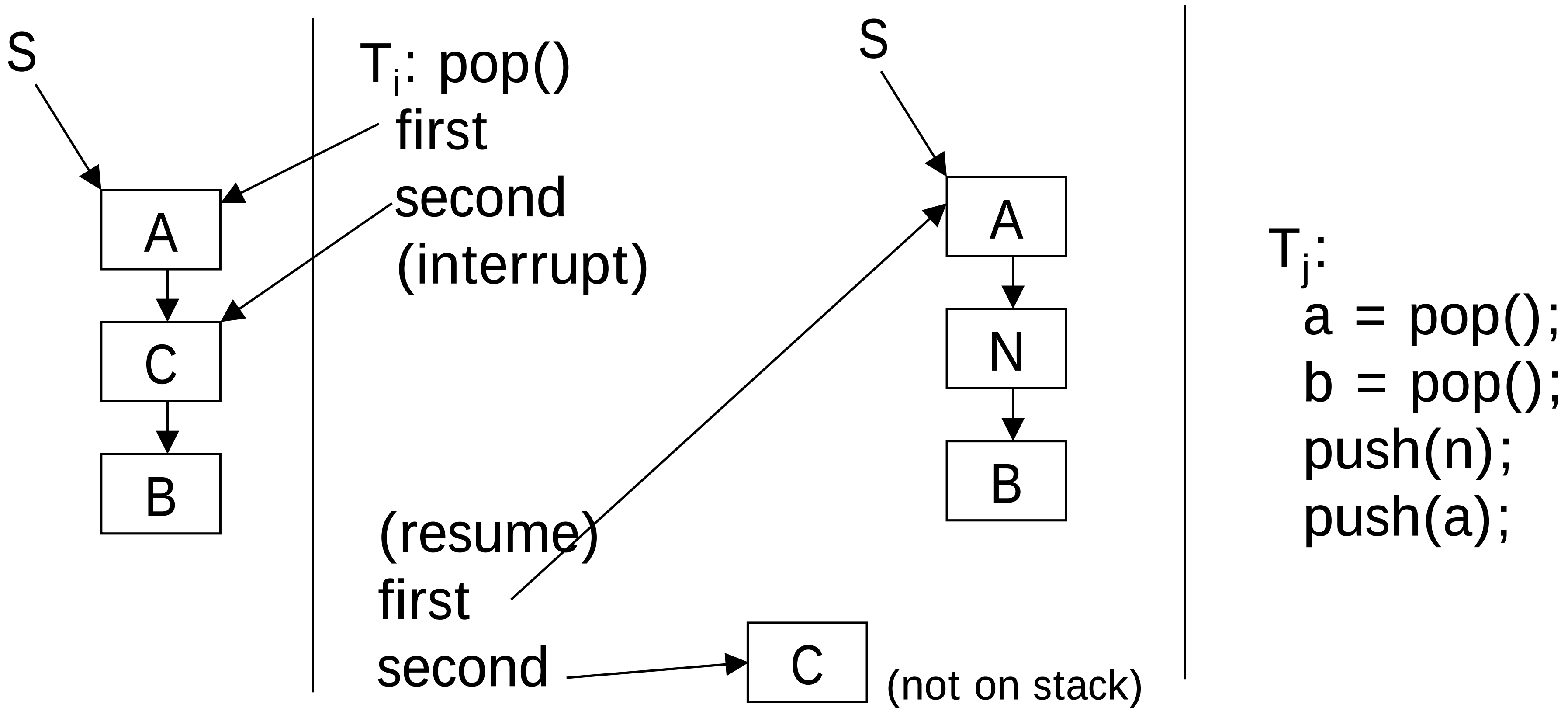
T_i: pop()
first
second
(interrupt)



T_j:
a = pop();
b = pop();

ABA Problem

- CAS(x, y, z) succeeds if value stored at x matches y



`CAS(S,first,second)`

One Solution

- Include a version number with every pointer
  - pointer_t = <pointer, version>
  - Increment version number (atomically) every time you modify pointer
  - Change to version number guarantees CAS will fail if pointer has changed
  - Requires double-word CAS operation (most architectures do not provide this)
  - Use garbage collection to reclaim memory later
    - May restrict reuse of memory

Using NBS

- Good for simple data structures, update heavy
- When you need NBS constraints/guarantees
  - Progress in face of failure
  - Linearizability
    - Everyone agrees on all intermediate states
- Both constraints are often irrelevant!

Constraints Irrelevant?

- Real systems don't fail the way theoretical ones do
  - Software bugs are not always fail-stop
  - Preemption/interrupt is not a failure
    - And can be controlled by system programmer or scheduler-conscious synchronization
  - Page fault is not a failure
    - Over-provision memory… if shared data really is paged out, it will have to be brought into memory before progress is made anyway
- Don't always need intermediate states, just final
  - Linearizability implies dependency → limits parallelism
  - If events are unrelated, asynchronous, does it matter which happened first?

Read-Copy Update (RCU)

- What is RCU?
  - Paul McKenney's PhD thesis
  - a key part of the Linux scalability effort
  - and one of the key technologies in the SCO lawsuit against IBM.
- Ok, what is it really?
  - Reader-writer synchronization mechanism
    - Readers use no locks; best for read-mostly data structures
    - Writers create new versions atomically
      - typically by locking out other writers
    - Readers can continue to access old versions
      - Old versions must be deleted at some point
      - "poor man's garbage collection"

RCU Basics

- From http://lwn.net/Articles/262464

1. Publish/Subscribe mechanism (for insertion)
2. Mechanism to wait for previous readers to complete (for deletion)
3. Maintain multiple versions of recently updated objects (for readers)

RCU Publish/Subscribe

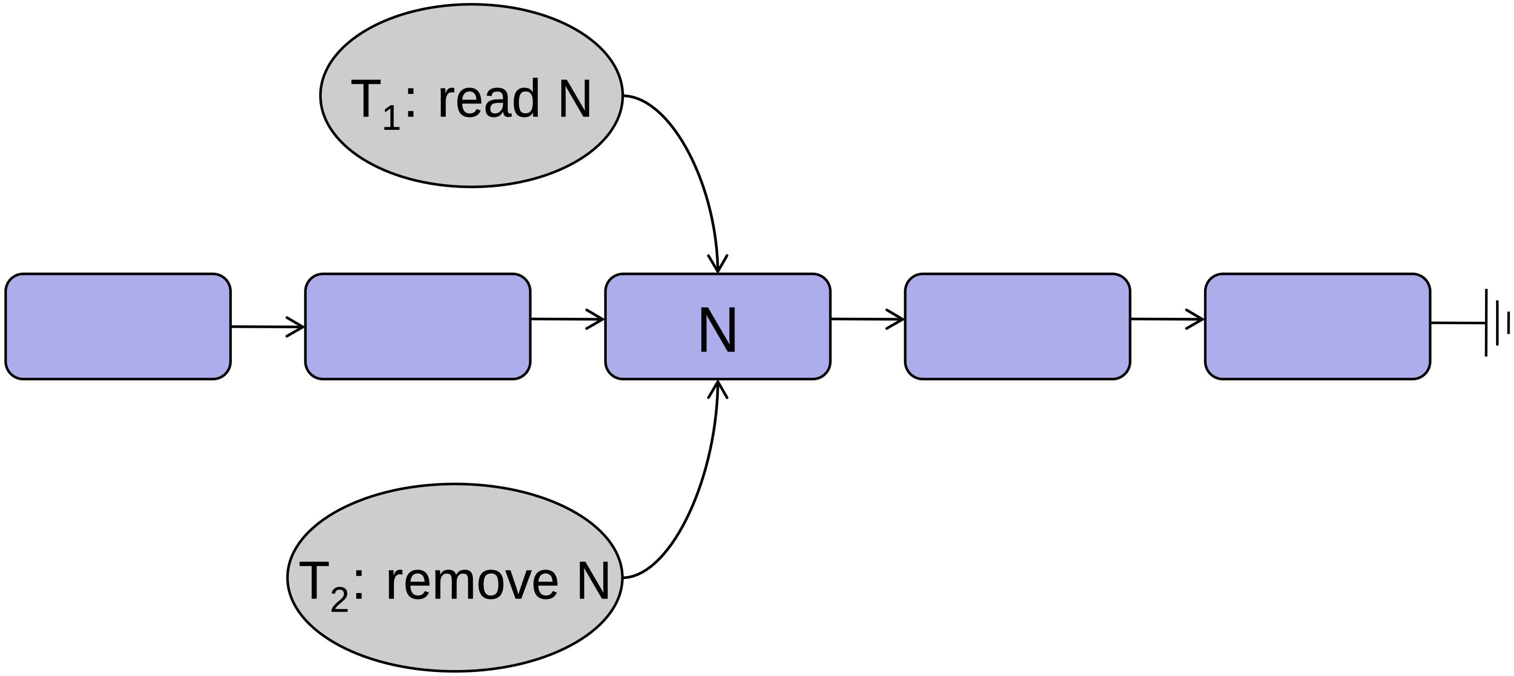
```
/* definitions */
struct foo {
  int a;
  int b;
  int c;
};

struct foo *gp = NULL;
```

```
T1: p = kmalloc(sizeof(*p),
                   GFP_KERNEL);
T1: p->a = 1;
T1: p->b = 2;
T1: p->c = 3;
T1: gp = p; rcu_assign_pointer(gp,p);
...
    rcu_read_lock();
T2: p = gp;  p = rcu_dereference(gp);
T2: if (p != NULL)
T2:    use(p->a, p->b, p->c);
    rcu_read_unlock();
```

- When is it safe to read a pointer?
  - RCU Readers use no locks
  - Compiler, CPU may reorder assignments
  - Enforce ordering with rcu_assign_pointer/rcu_dereference

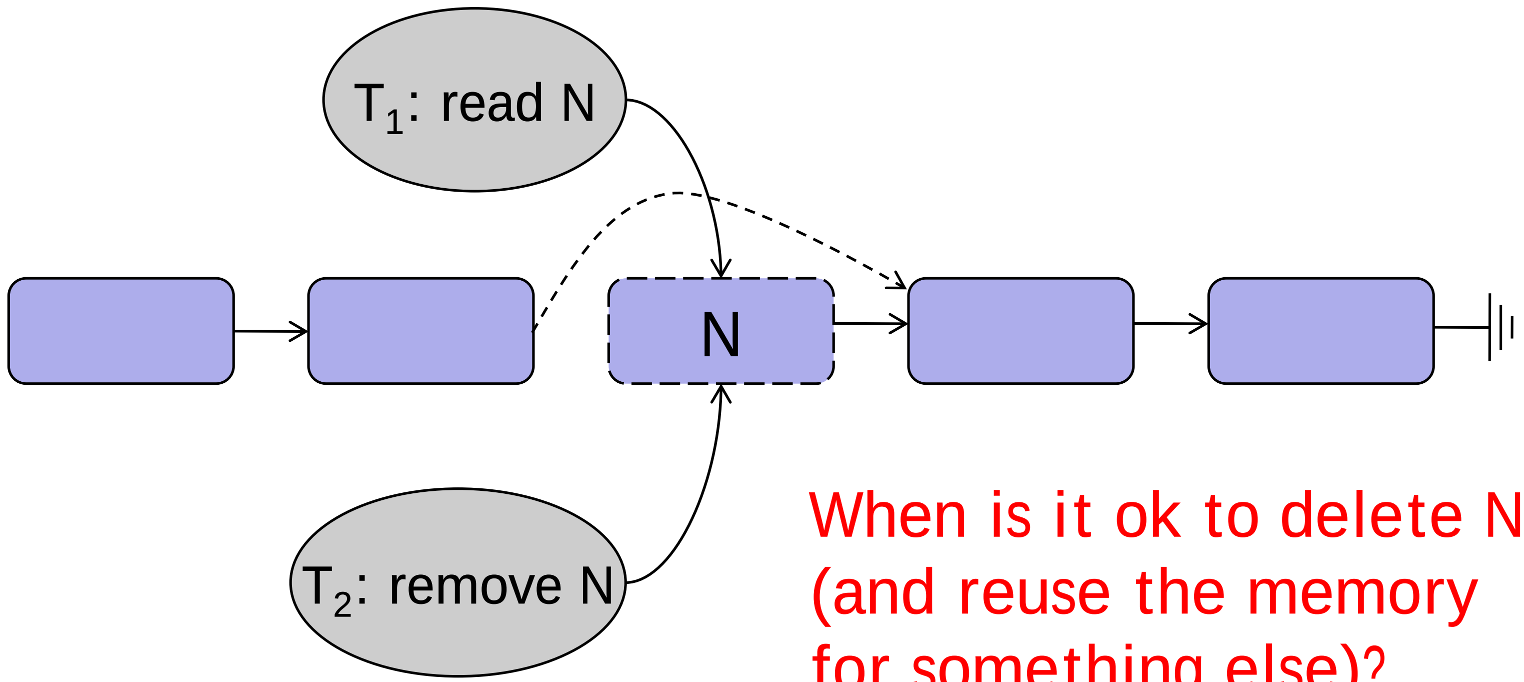
RCU Deletion Example

- T_1 traversing linked list, T_2 removes an element:



RCU Deletion Example (2)

- After removal – T_1 continues to use N and later nodes in the list

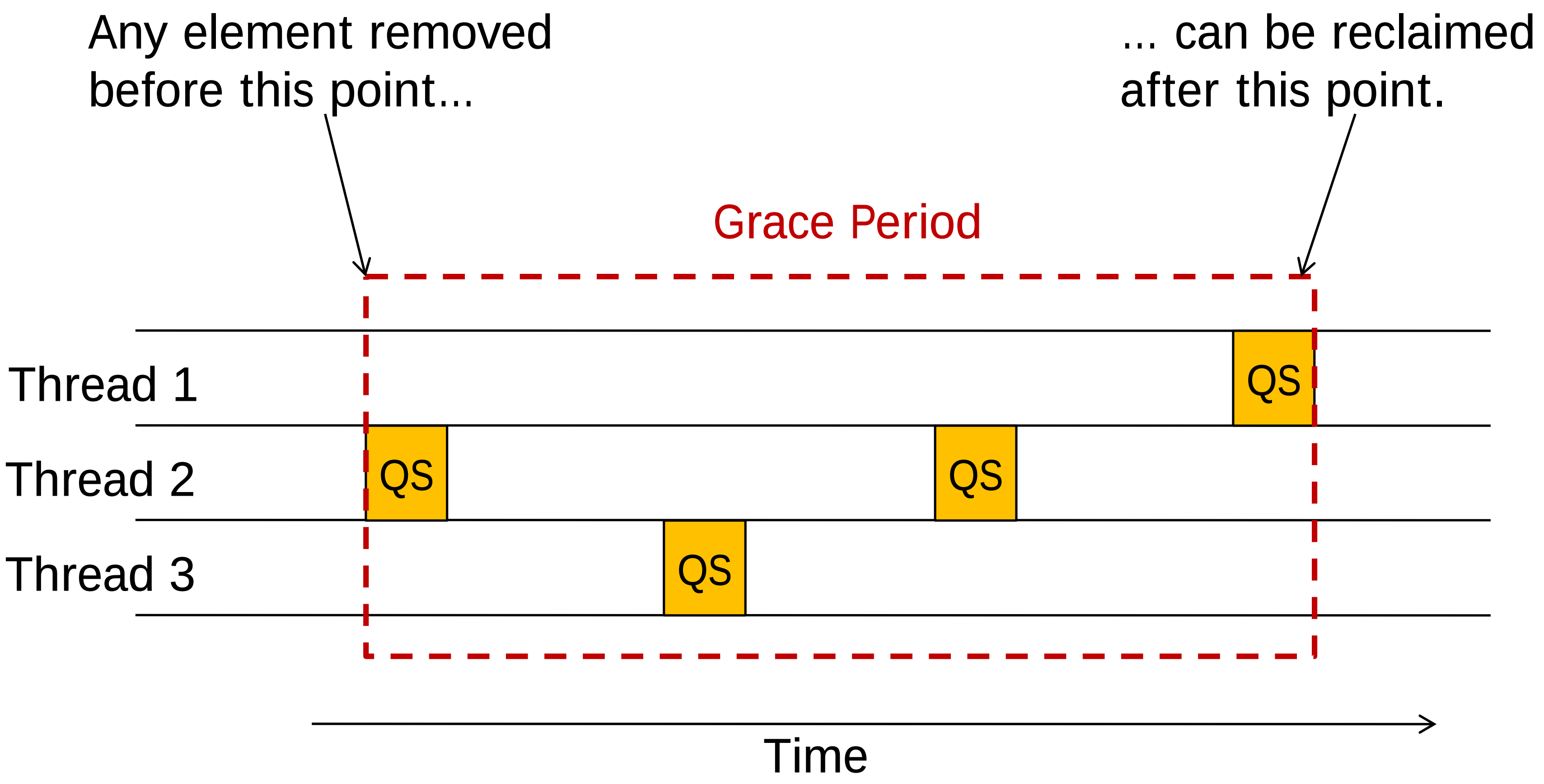


When is it ok to delete N (and reuse the memory for something else)?

Handling read-reclaim races

- RCU uses *quiescent state based reclamation* (QSBR)
- Defn: A *quiescent state* for a thread T is a state in which T holds no references to shared data
- Defn: A *grace period* is an interval in which every thread has passed through at least one quiescent state
- Basic Idea: elements removed from a data structure can be reclaimed after a grace period, since no thread can still be holding a reference to the old element at that point

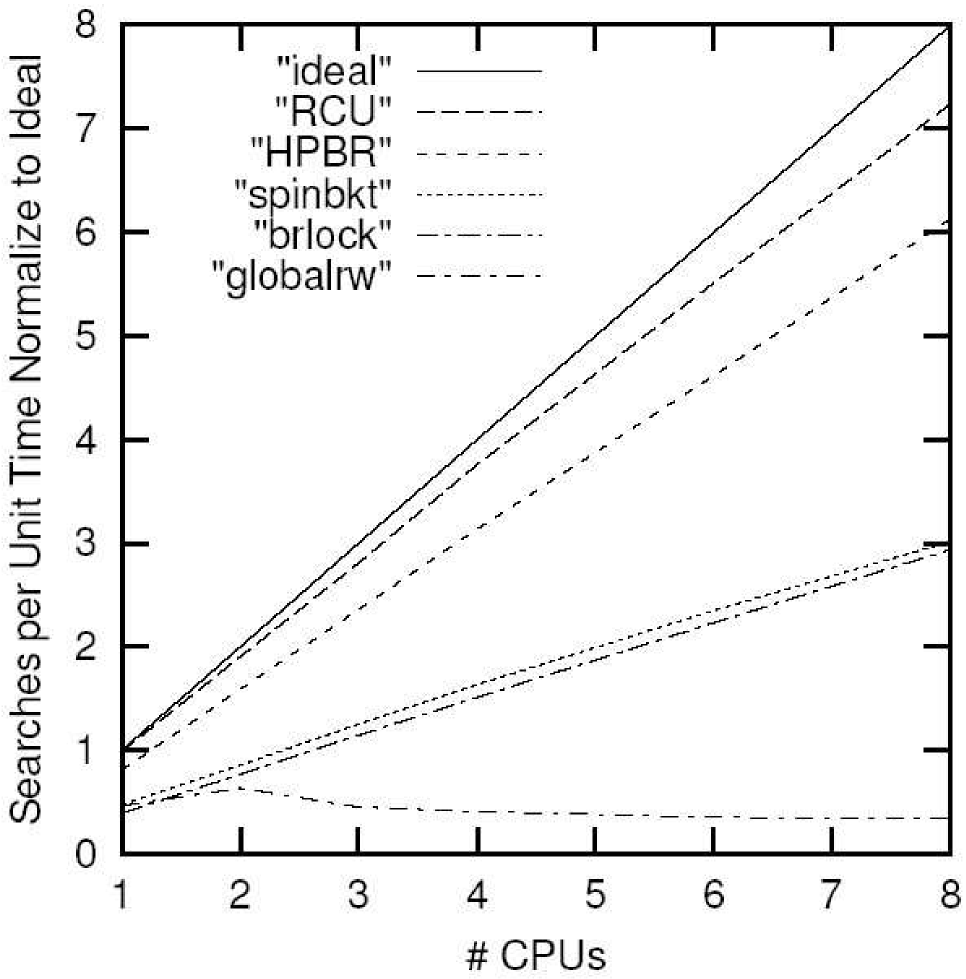
Illustration



How to define Quiescent States?

- Application dependent!
- For OS kernels, some natural ones exist
  - E.g. a context switch in a non-preemptive kernel
- RCU primitives
  - rcu_read_lock() and rcu_read_unlock()
    - Surround read-side critical sections
    - No overhead (#define'd as nothing) in non-preemptive kernels
    - Modest overhead in preemptive kernels (disable preemption)
  - synchronize_rcu()
    - Wait until all pre-existing RCU read-side critical sections complete
    - Force execution on all CPUs

PPC Hash Table with RCU



When to use which tool

- Read-mostly situations
  - RCU (if algorithm can tolerate concurrent reads and updates)
- Update-heavy situations
  - Simple data structures and algorithms: NBS
  - Complex data structures and algorithms: Locking

"When the only tool you have is a hammer, everything looks like a nail."

  - It's good to have lots of tools in your toolbox

Transactional Memory

Active research! Here be dragons…

Challenges of Synchronization

- Two major issues:
  - Performance
    - Scalability
    - Base cost
    - We have looked at some techniques that address this
      - Better spinlocks
      - Lockless strategies (NBS, RCU)
  - Programmability
    - Locks are hard to use correctly
    - Lockless data structures are hard to design

What's missing?

- Lack of support for *abstraction* and *composition*
- E.g. Suppose we have thread-safe stack with (abstract) push and pop operations
  - In sequential programs, can use these operations without regard to their implementation
  - In parallel programs, internal details may be needed
    - Consider task of moving an item from one stack to another
    - Need to expose stack locking mechanism

“Magic” Wish List

- Let programmers express desired outcome
  - “This block of code should appear atomic”
- Let run-time system or hardware support make it happen
- Allow abstractions to hide implementation and be composable

☞ A new programming model is needed

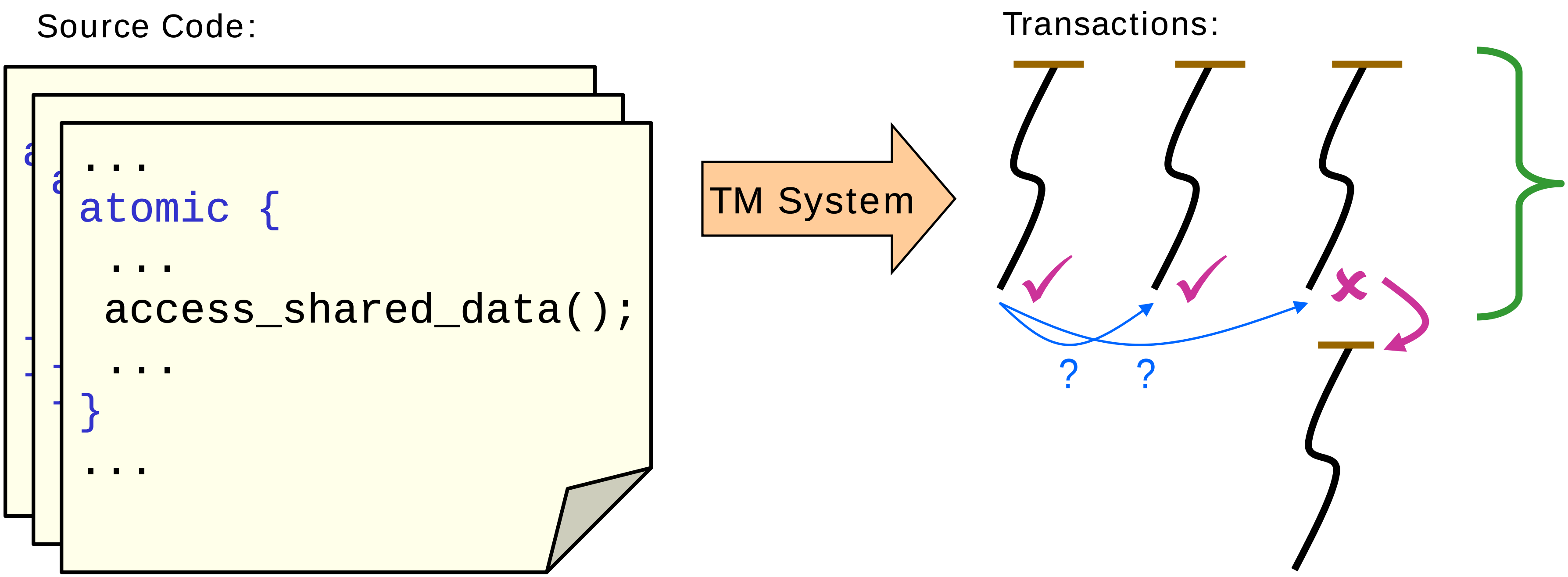
Database Transactions

- Database systems allow multiple queries to run in parallel
- Query authors don't worry about concurrency
- Complex queries can be composed out of simpler ones
- Can we use the DB programming model as a general parallel model?
- Key Programming Model: everything is a transaction
  - A transaction executes as if it were the only computation accessing the database
  - Restricted interactions, serializability
  - Hide complex implementation detail, programmer only sees a simple interface
  - Atomic – all updates become visible, or none
  - Consistent – transactions leave database in consistent state
  - Isolated – no interference with or from other transactions
  - Durable – once committed, updates are permanent

Transactional Memory: Some History

- 1977 – D.B. Lomet (IBM Research, now at Microsoft Research) suggests database transaction model for concurrent programming
  - No practical implementation provided
- 1983 – Kung & Robinson propose *optimistic concurrency control* for databases
- 1988 – Chang & Mergen describe IBM 801 storage manager
  - HW provided lock bits for each 128 byte range of a page; page tables & TLB extended
- 1993 – Herlihy & Moss describe a hardware proposal for *transactional memory*

Transactional Memory (TM)



Programmer: Specifies threads/transactions in source code

TM System: Executes transactions optimistically in parallel

1) Checkpoints execution
2) Detects conflicts
3) Commits or aborts and re-executes

Differences from DB Transactions

- Memory vs. disk
  - Disk access takes 100X longer than memory access

  → database systems can use relatively heavy-weight software solutions
- No need for durability
  - Memory is transient anyway

  → simplifies TM implementations
- Existing languages, libraries and systems
  - Databases are closed systems in which all code executes as a transaction, BUT programs using TM must coexist with libraries and OSs that do not

TM Implementations

- Hardware TM (HTM)
  - Changes to computer system and ISA
  - Extra cache to buffer writes, extended coherence protocol to track conflicts, special transaction instructions
  - Support for limited number of memory locations
- Software TM (STM)
  - Language runtime (or library) + extensions to specify transaction
  - Exploit current commodity hardware (multicores)
  - Get experience with transactional programming model
  - Java: DSTM (Marathe et al.), ASTM (Herlihy et al.)
  - C/C++: McRT-STM (Saha et al.), TL2 (Dice et al.), RSTM
  - Intel's C++ STM compiler
- Hybrid TM (HyTM)

Programming Constructs

- Atomic block

```
atomic {
  if (x!=null) x.foo();
  y = true;
}
```

- Delimits code that should execute in a transaction
- Dynamically-scoped – code in foo() executes in transaction as well
- Does not name shared resources (unlike monitors or lock-based programming)
- 3 possible outcomes – commits, aborts, non-termination

Caution!

- Programmers can still use *atomic* incorrectly

```
bool flagA=false; bool flagB=false;
```

Thread 1:

```
atomic {
  while (!flagA);
  flagB = true;
}
```

Thread 2:

```
atomic {
  flagA = true;
  while (!flagB);
}
```

- What's wrong?
  - Deadlock results

Semantics

- Not yet formally specified!
- Useful ways to reason about TM:
  - Database correctness criteria: serializability
    - Useful for understanding transaction behaviour
    - Says nothing about interaction of transactions with code outside of transactions
  - Operational semantics – single-lock atomicity (SLA)
    - Program executes as if all atomic blocks were protected by single global lock
    - Attractive, but may be problematic conceptually
    - SLA does not support failure atomicity, forms of nesting, etc.

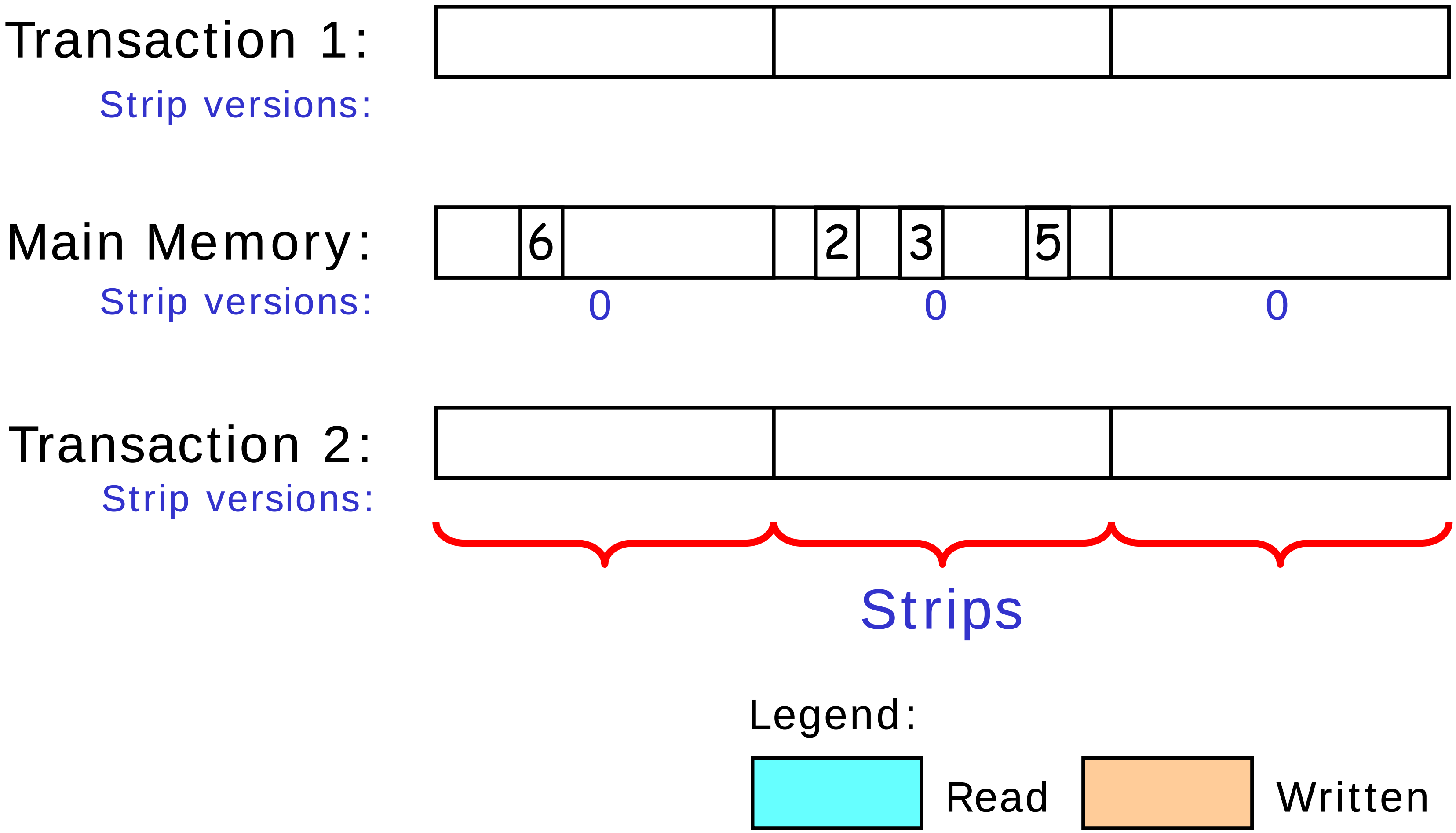
Implementation Basics

- For all (non-stack) write instructions:
  - Track write addresses and values (*write set*)
- For all (non-stack) read instructions:
  - track read addresses and values (*read set*)
- When a transaction completes:
  - Atomically
    - Validate read set (conflict detection)
    - Commit write set

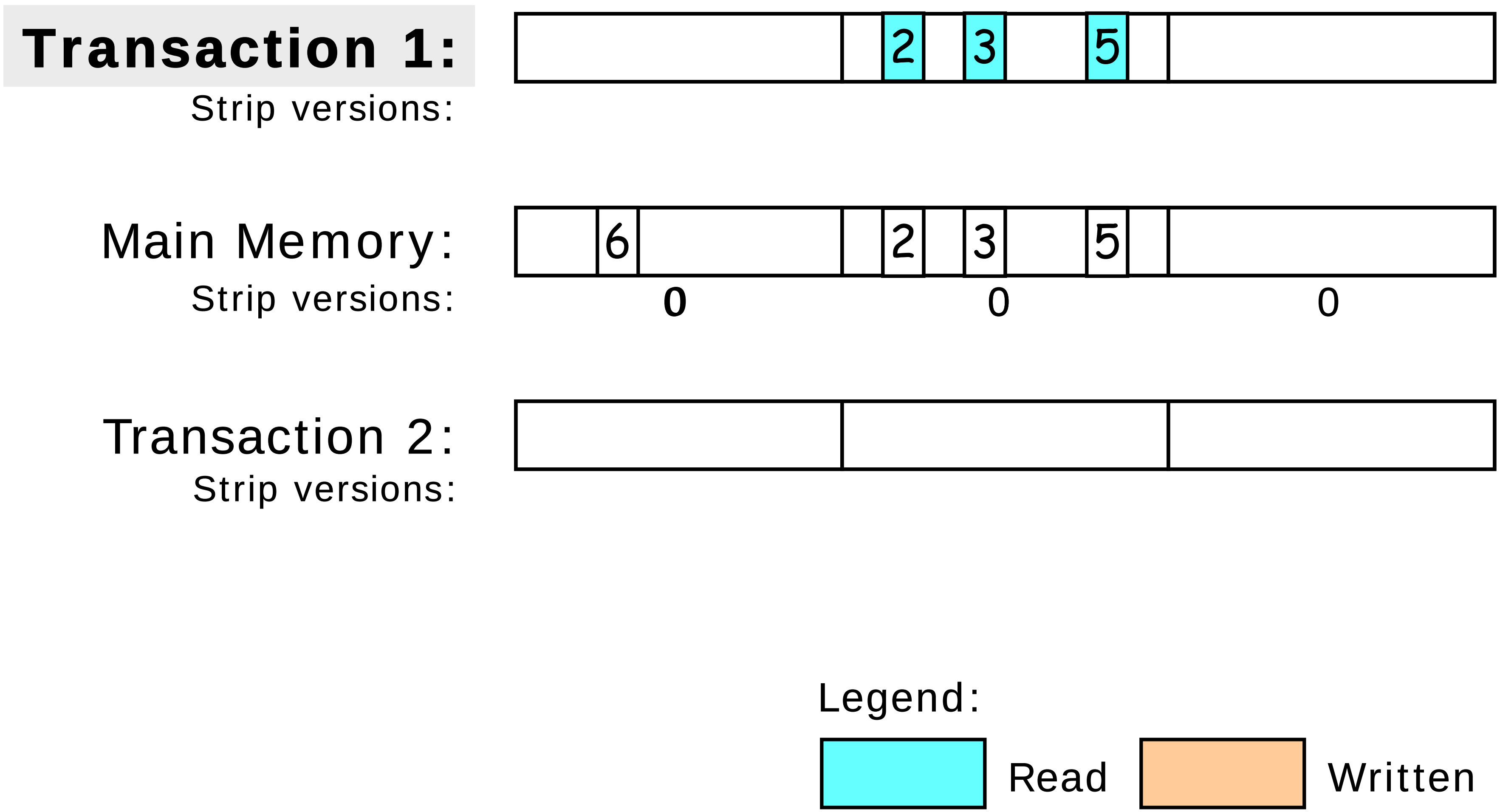
Implementation Options

- Transaction Granularity
  - Unit of storage over which TM system detects conflicts
  - Similar to notion of cache coherence
  - Word or block typical for HTM, object common for STMs that extend OO language
- Direct or Deferred Update
  - Direct – transaction directly modifies the object itself
    - Must log previous value for undo in case of abort
  - Deferred – modify private copy, propagate at commit
  - Both get complicated in the presence of data races
- Optimistic or Pessimistic Concurrency Control
  - TM typically optimistic; need to detect and resolve conflict

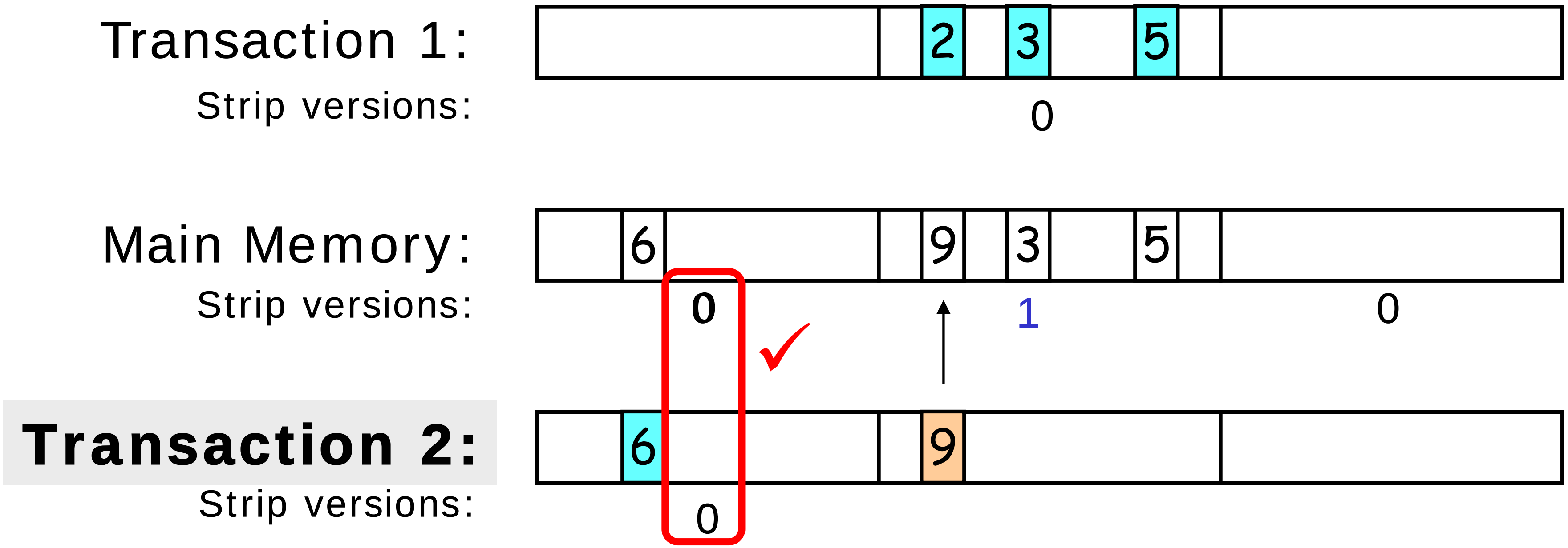
Location-Based Conflict Detection



Location-Based Conflict Detection

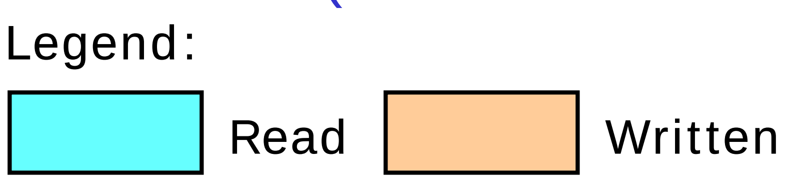


Location-Based Conflict Detection

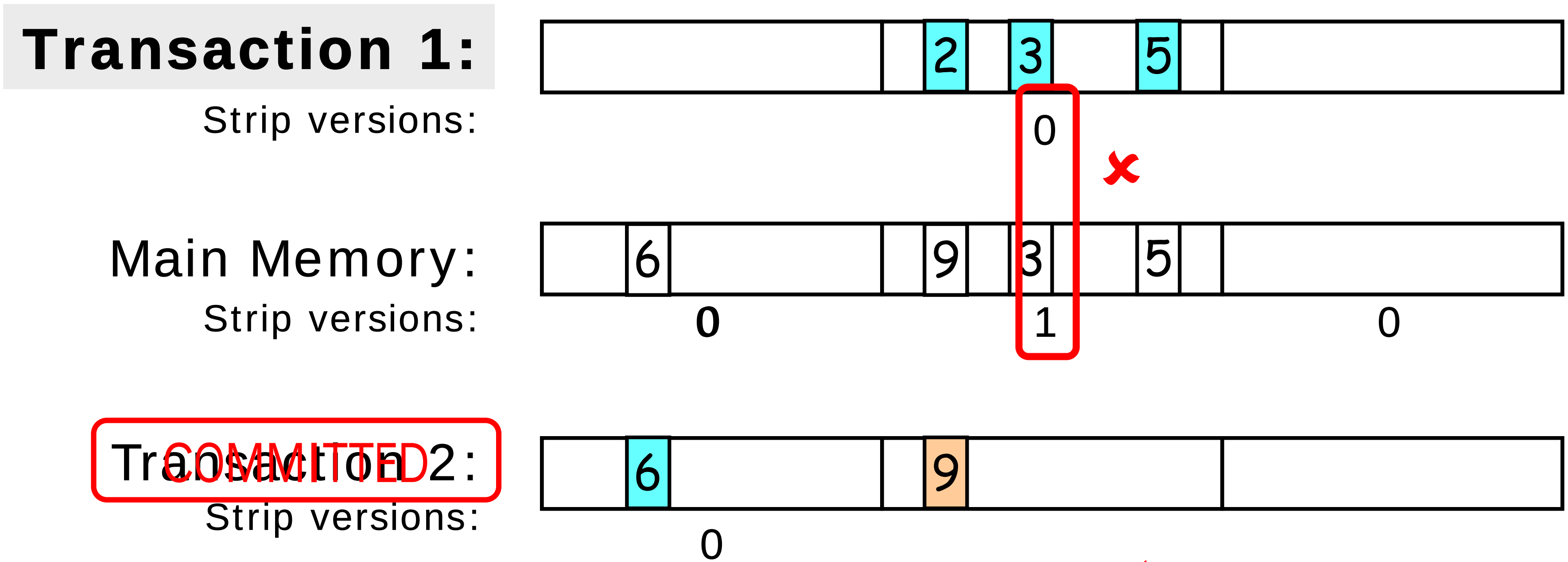


Commit step 1) Validate Read Set ✓

Commit step 2) Publish Writes (and inc version #s)



Location-Based Conflict Detection



Commit step 1) Validate Read Set ✘ Abort!

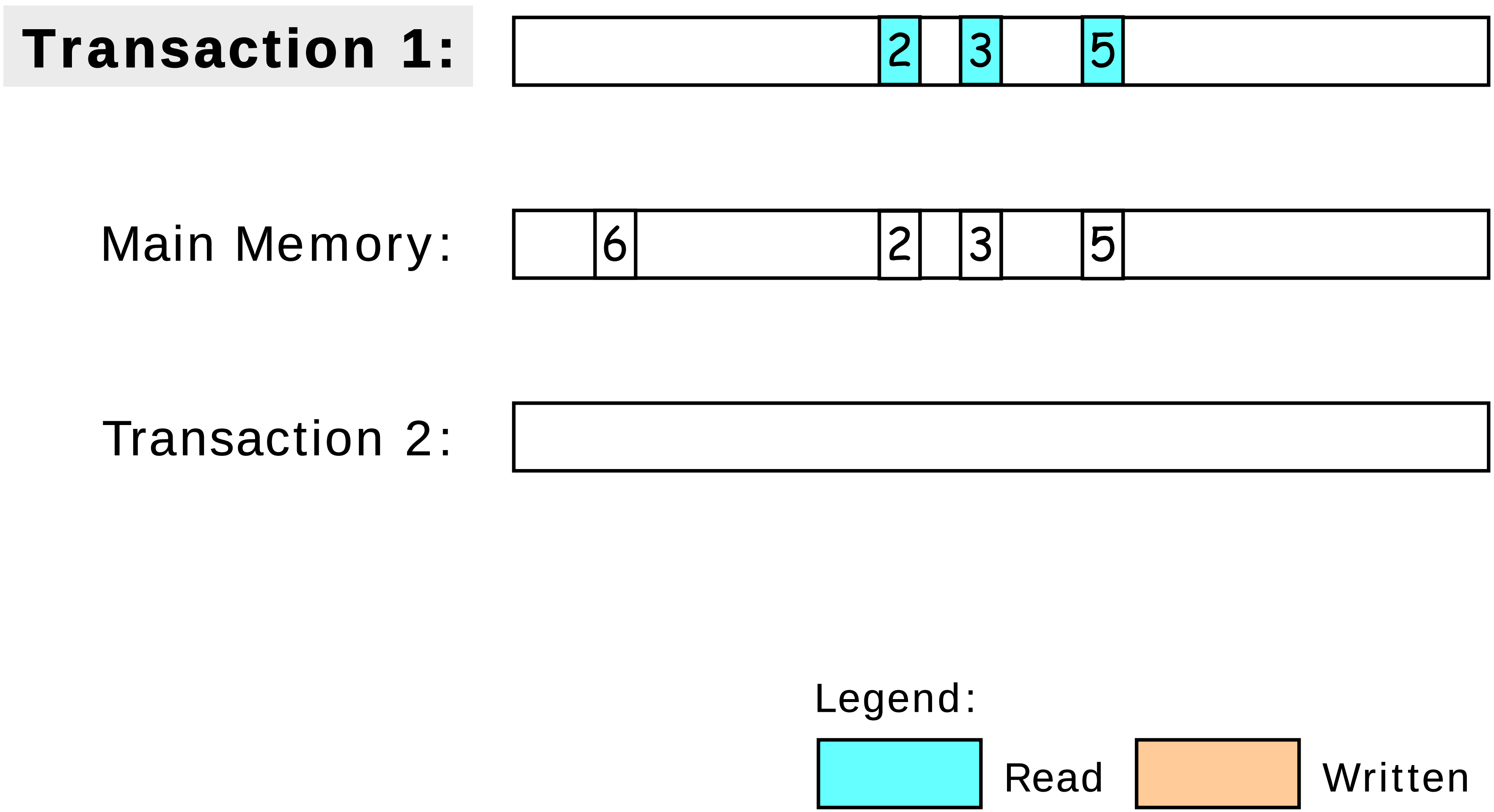
Note: all transactions must maintain strip version #s

Legend:

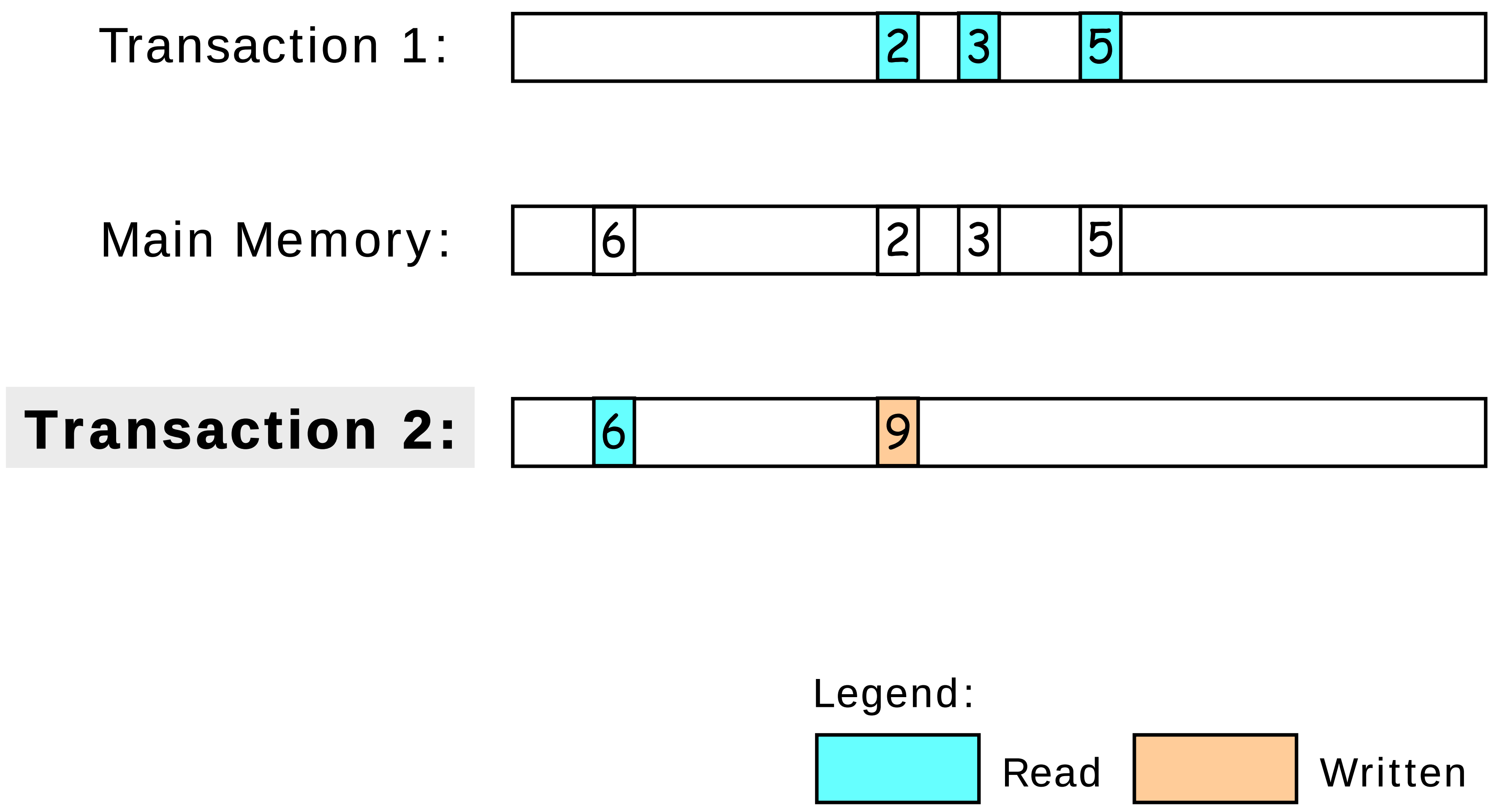
Read

Written

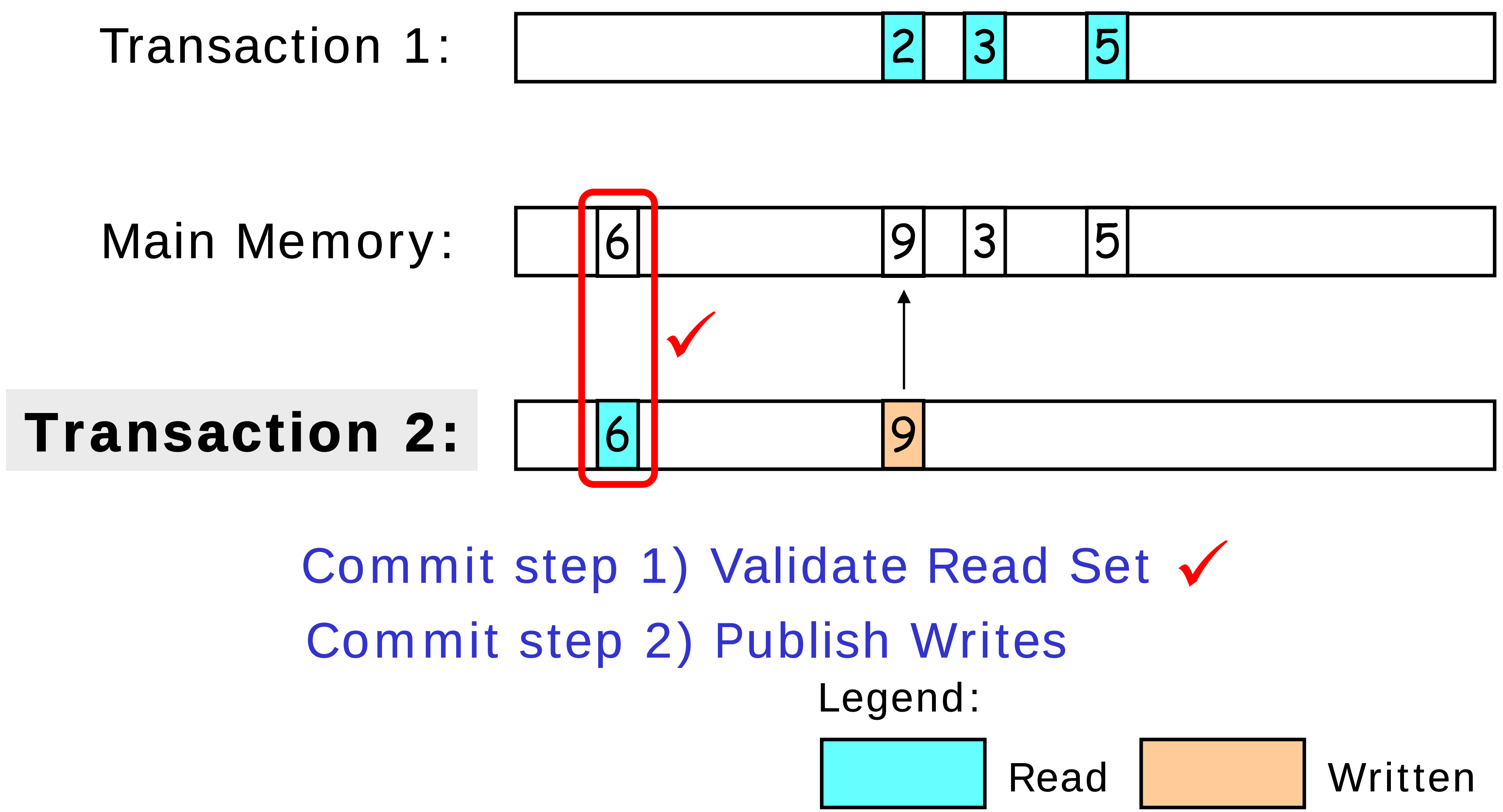
Value-Based Conflict Detection



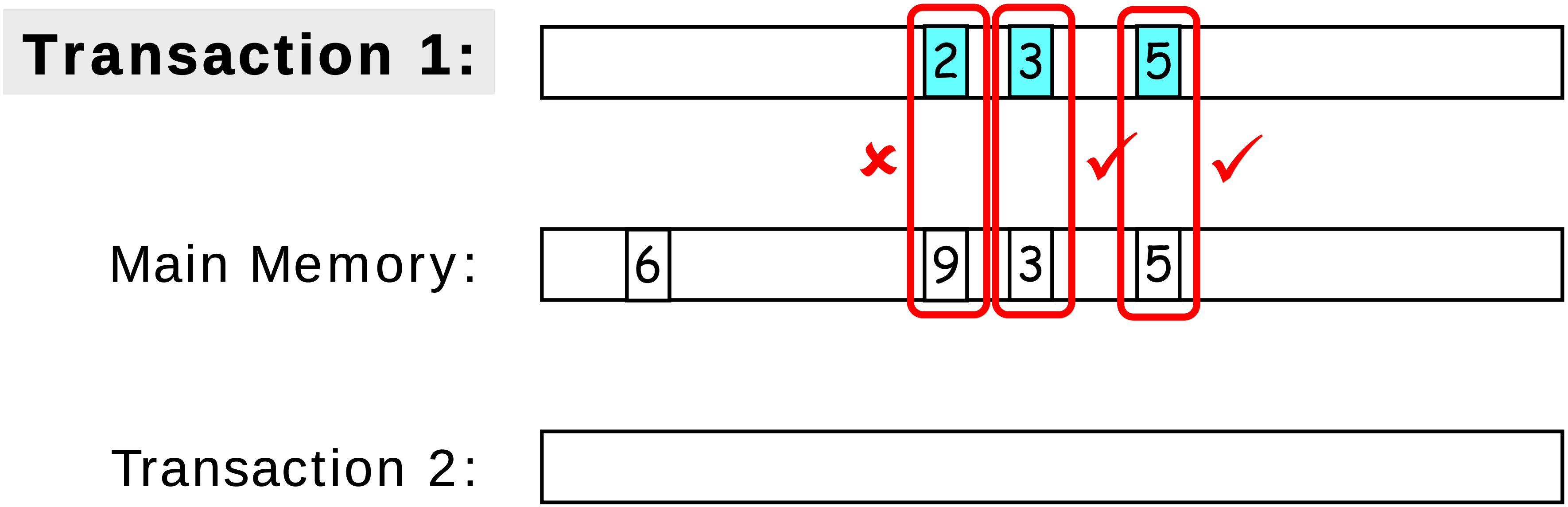
Value-Based Conflict Detection



Value-Based Conflict Detection



Value-Based Conflict Detection



Commit step 1) Validate Read Set ✗ Abort!

Note: no version information to maintain

Legend:

Read

Written

TM Weaknesses

- Some operations are hard to abort/retry
  - Essentially anything not idempotent, e.g. I/O
- In practice, TM does not interact well with locking
- Some variables are prone to high conflict rates (frequent true sharing & dependences)
- Conflict resolution needs to avoid starving long-running, large transactions
- Poor interaction with standard software tools like debuggers
  - Getting better though ...

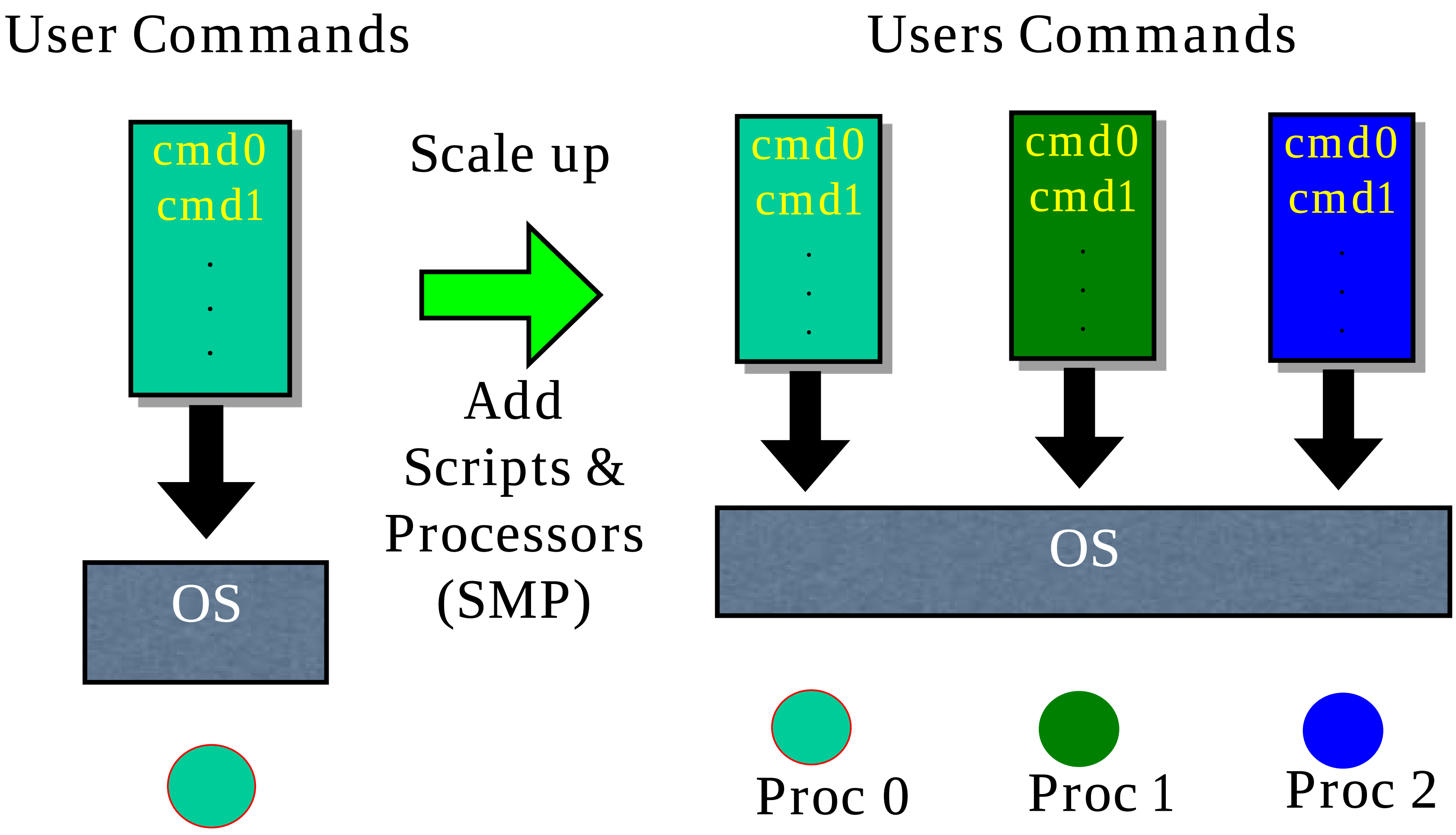
TM Status

- Hardware TM is now a reality
  - Sun's Rock processor was killed after acquisition by Oracle (2009)
  - Azul Systems has HTM in their Java appliance hardware (circa 2009)
  - IBM BlueGene/Q (2011)
  - Intel Haswell's *Transactional Sync Extensions (TSX)*
- Software TM has performance problems
  - But some applications are a nice fit
    - E.g. parallel game server

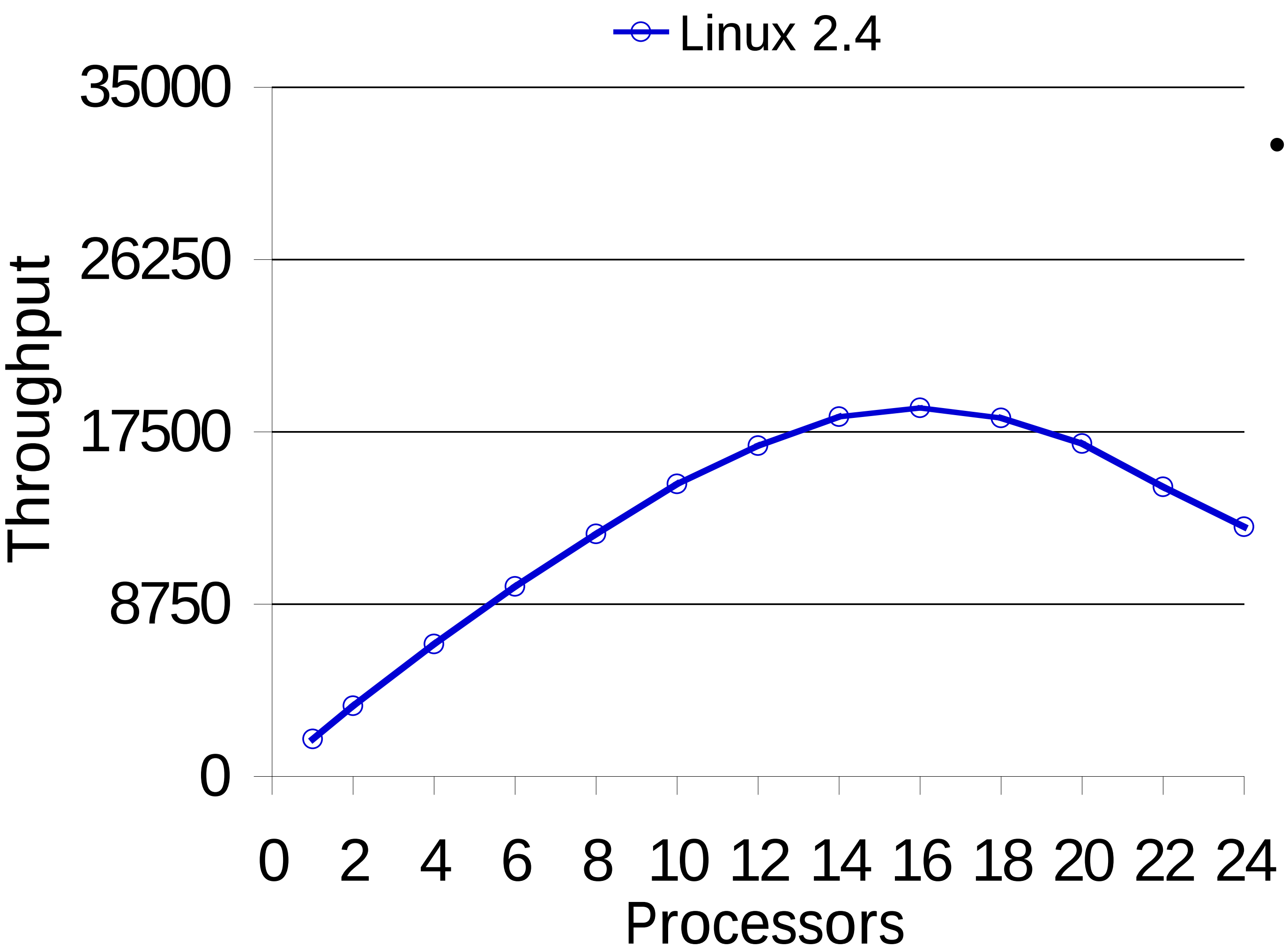
Operating System Scalability

- We have looked at various synchronization strategies
  - Scalability has been a key concern
- Most user applications actually aren't very scalable
- Most exceptions use few OS services anyway
  - E.g. scientific computing
- Most multiprocessor systems support independent processes (multi-user workloads)
- Why does OS scalability matter?

Systems View of Scalability

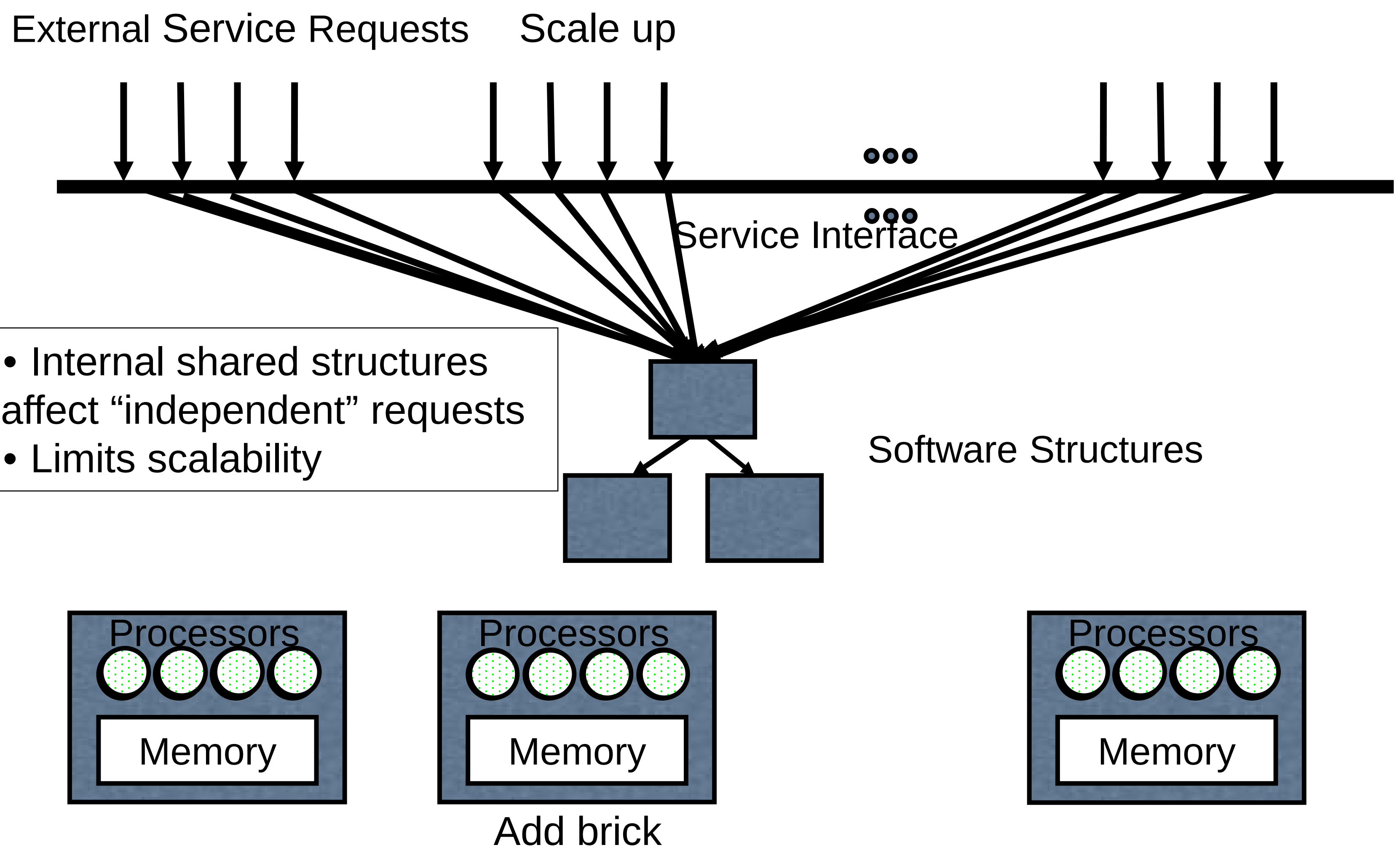


The Problem



- SDET benchmark

Scaling Existing OSes

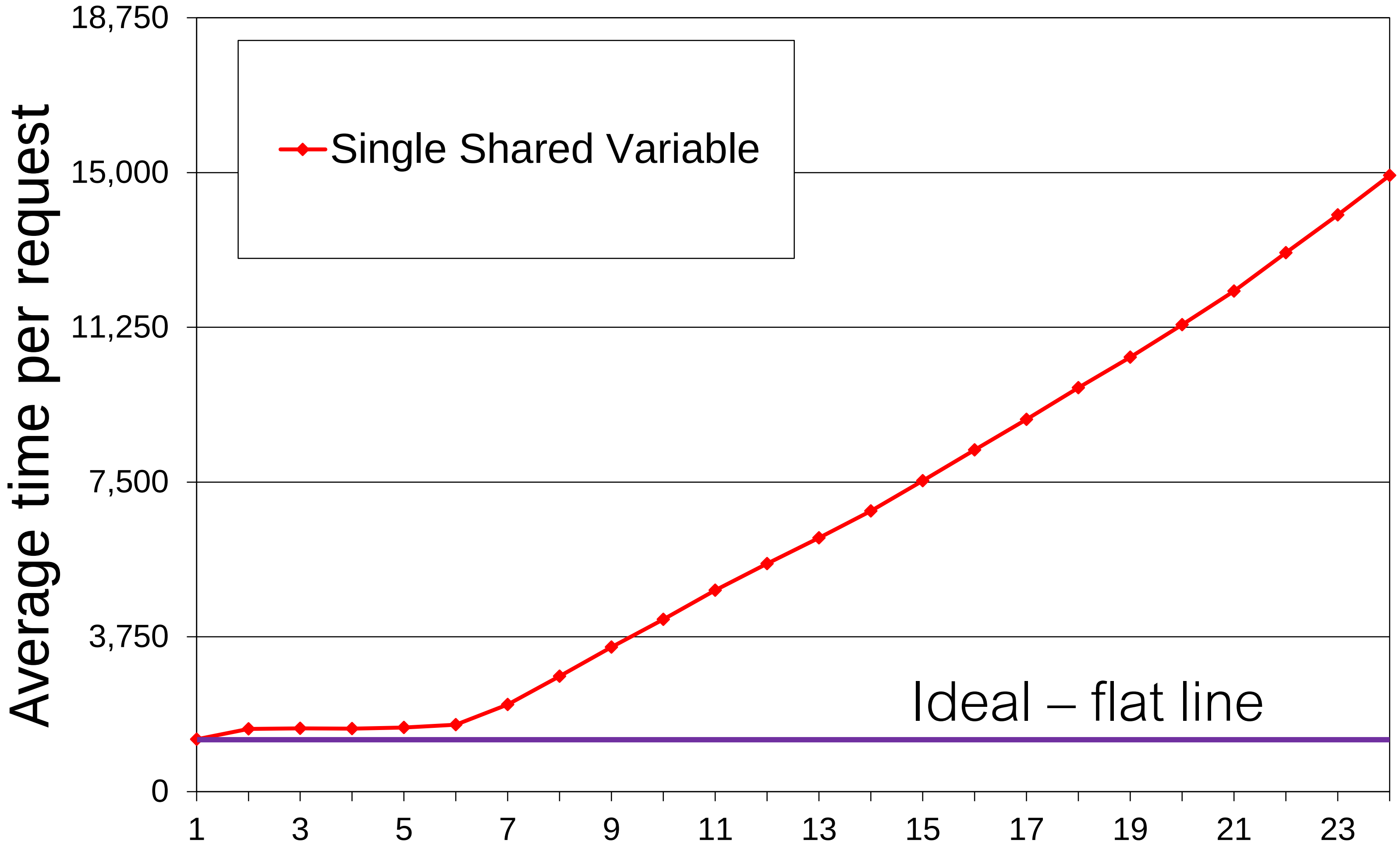


- Internal shared structures affect "independent" requests
- Limits scalability

Areas of Concern

- Statistical counters
  - Widely used to track variety of system properties
  - Frequently updated, rarely read
- Processor scheduling
  - We'll look at this closely in the next 2 lectures
- Memory management
  - In tutorial this week (also, Assignment 2!)

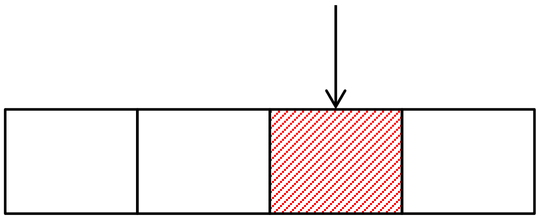
Simple Shared Counter Example



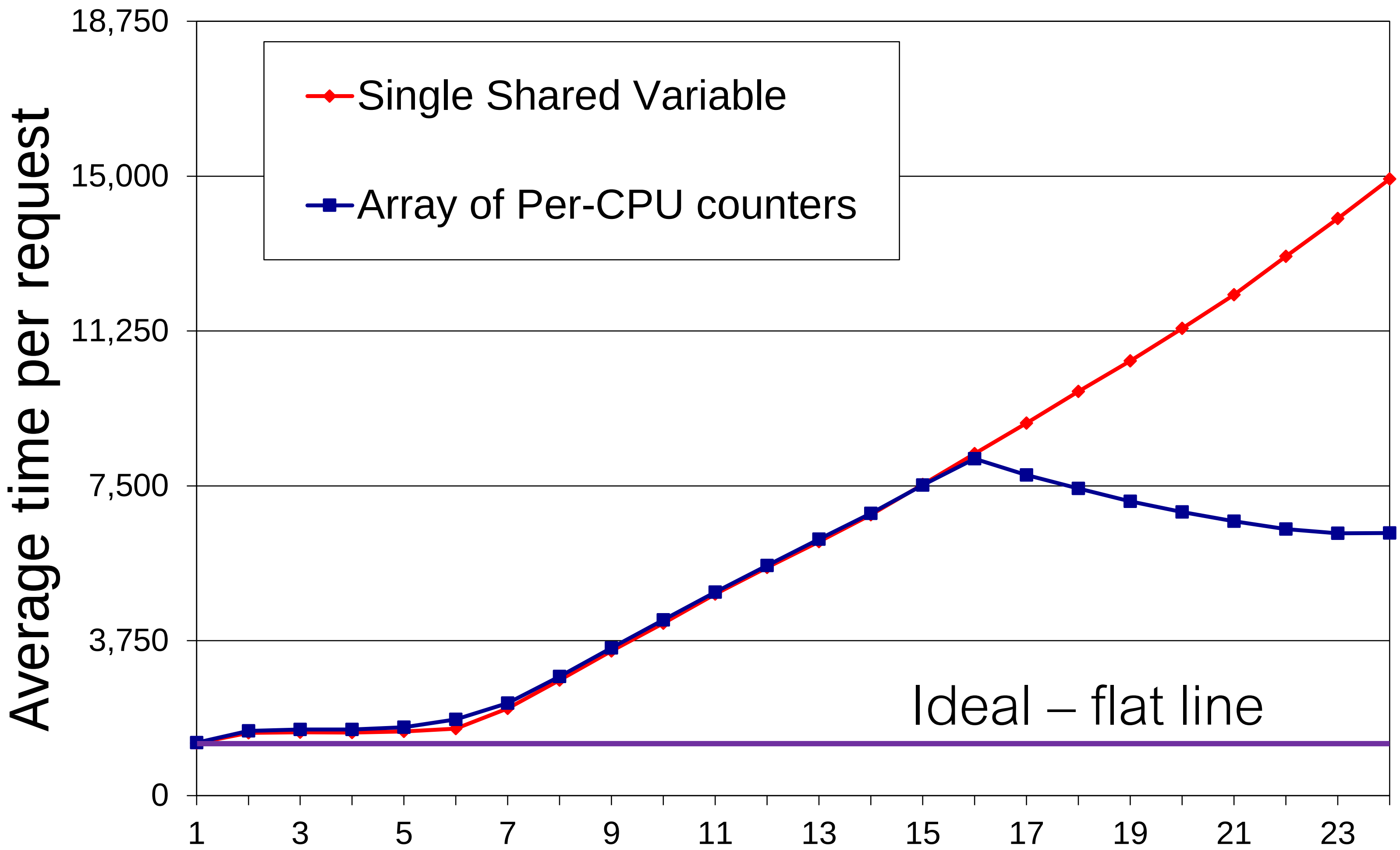
Solution: Per-CPU data

- OS assigns each CPU an integer *id* at boot time
  - Linux: access with smp_processor_id()

- Basic data structure is array with entry for each CPU
  - counter[smp_processor_id()] is data structure for current CPU



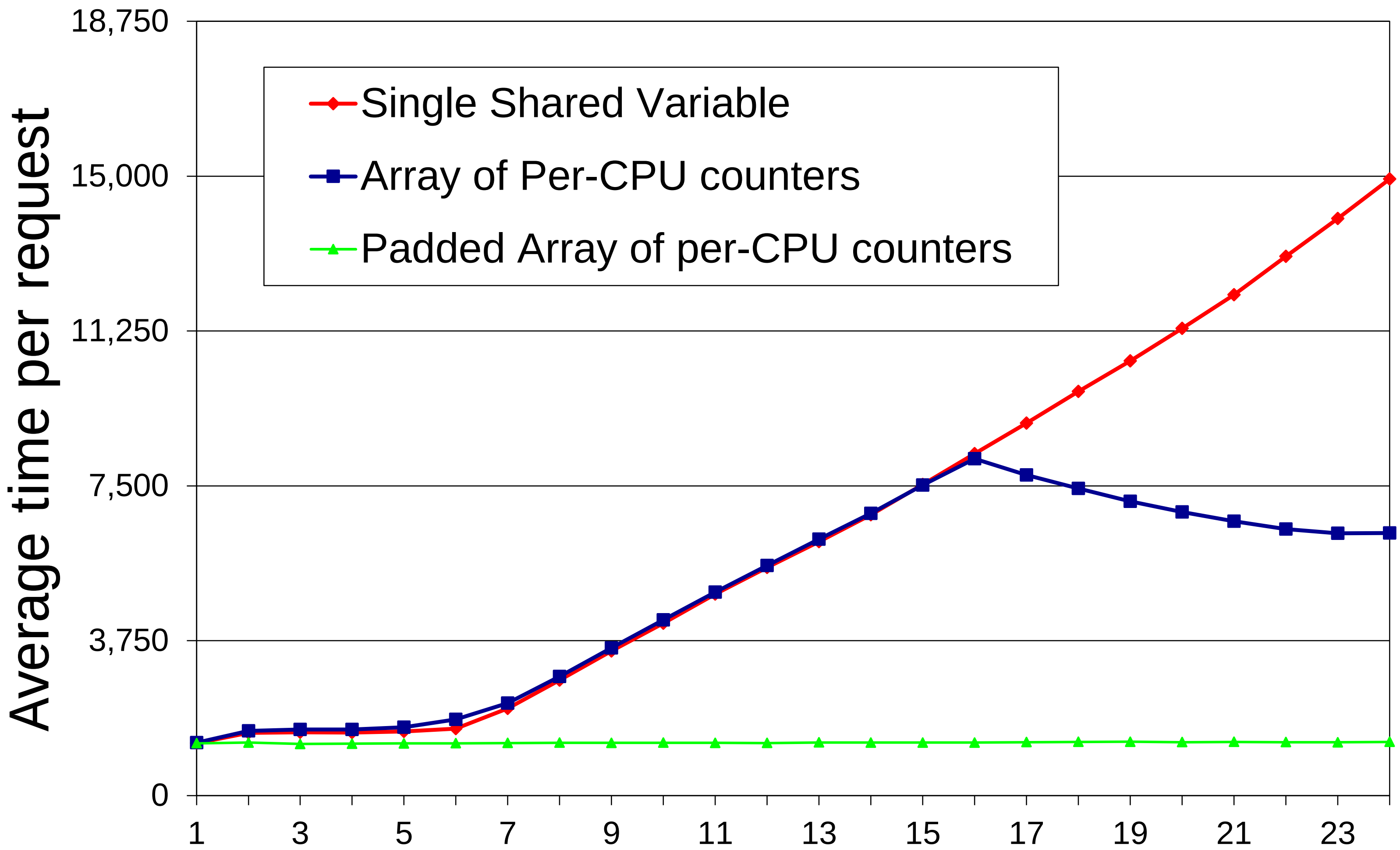
Simple Shared Counter Example



What went wrong?

- Per-CPU array can lead to *false sharing* problem
  - Each CPU has own variable
  - Several per-CPU variables are on same cache line
  - Modification of one causes invalidates in other CPUs' caches
- Solutions?
  - Use *padding* so each per-CPU variable lies on different cache line

Simple Shared Counter Example



Summary

- Taking a traditional OS and making it scale well on shared memory multiprocessors is hard
  - Fast uniprocessor solutions typically don't scale
  - Designing for scalability can hurt uniprocessor performance
  - Maintaining scalability with every change is hard

=> Must design a system from the ground up, with scalability in mind

Insights and Approaches

- Scalability must be considered in system design
- Shared data is the enemy
  - Distribute data structures
  - Use per-cpu data whenever possible
    - With padding to cache lines!
- Minimize locking and expensive atomic ops
- Ideas from research have been adopted by mainstream
  - UofT/IBM Tornado/K42 projects showed techniques to improve scalability
    - Some applied to Linux scalability project
  - More recently, MIT Corey project and OSDI paper on improving Linux scalability further