Lecture 9: Avoiding Locks

CSC 469H1F / CSC 2208H1F

Fall 2007

Angela Demke Brown

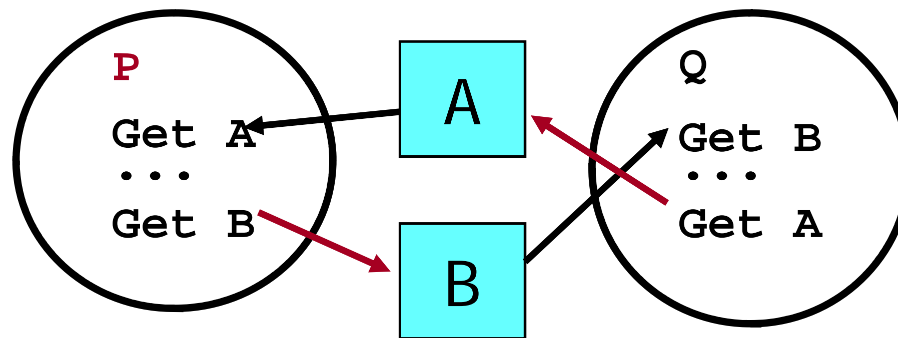
(with thanks to Paul McKenney)

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
  - Deadlock
  - Priority inversion
  - Not fault tolerant
  - Convoying
  - Expensive, even when uncontended
- *Not* easy to use correctly!

Deadlock

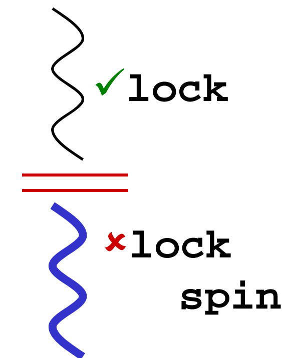
- Textbook definition: Set of threads blocked waiting for event that can only be caused by another thread in the same set
- Classic example:



- 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., disable interrupts while holding lock), but add complexity

Priority Inversion

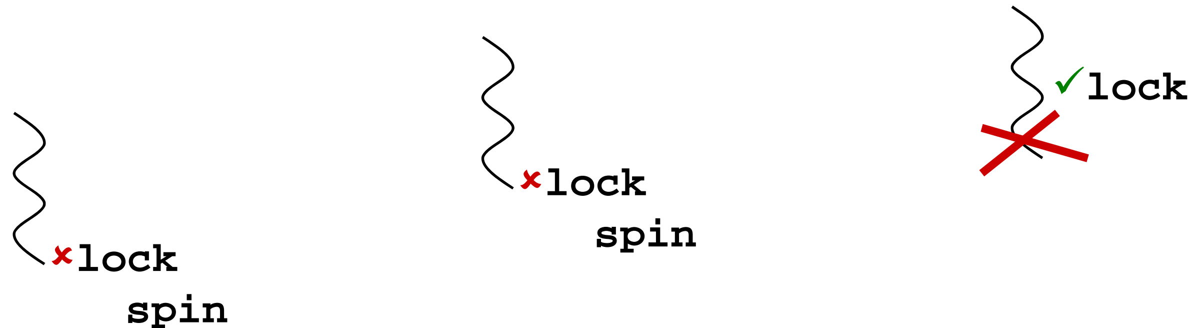
- Lower priority thread gets spinlock
- Higher priority thread becomes runnable and preempts it
  - needs lock, starts spinning
  - Lock holder can't run and release lock
    - May get to run on another CPU



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

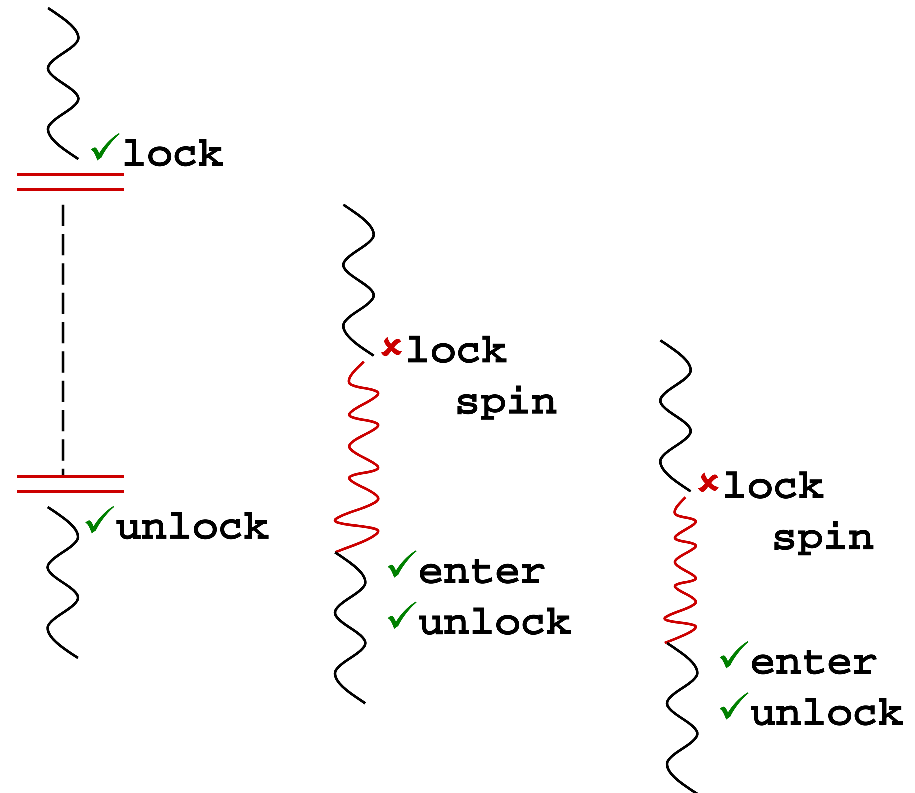
Not fault tolerant

- Lock holder crashes, or suffers indefinite delay, no one makes progress



Convoying

- Suppose we have set of threads, similar work per thread, but started at different times, occasionally accessing shared data
  - E.g. multi-threaded web server
- Expect access to shared objects (and hence times when locks are needed) to be spread out over time
  - Delay of lock holder allows other threads to catch up
  - Lock becomes contended and tends to stay that way

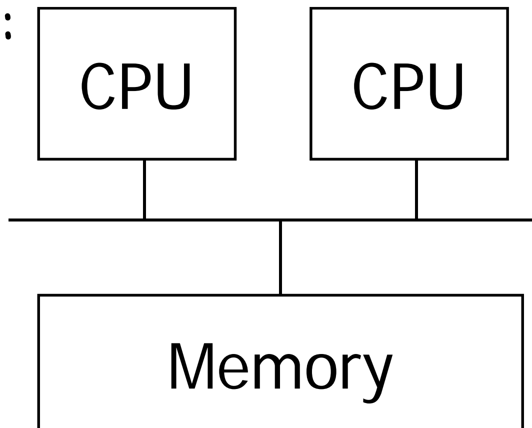


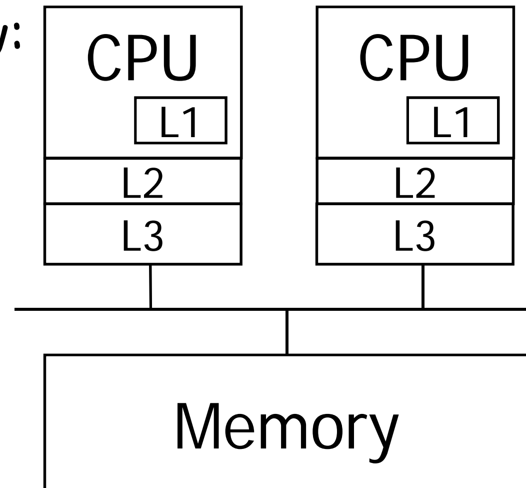
Expensive, even when uncontended

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



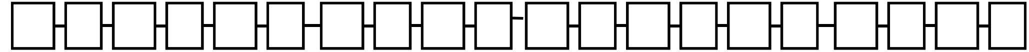


- Memory speeds have not kept up with CPU speeds
  - 1984: no caches needed, since instructions slower than memory accesses
  - 2005: 3-4 level cache hierarchies, since instructions orders of magnitude faster than memory accesses
- Synch. ops typically execute at memory speed

Causes: Deeper Pipelines

Then:

Now:

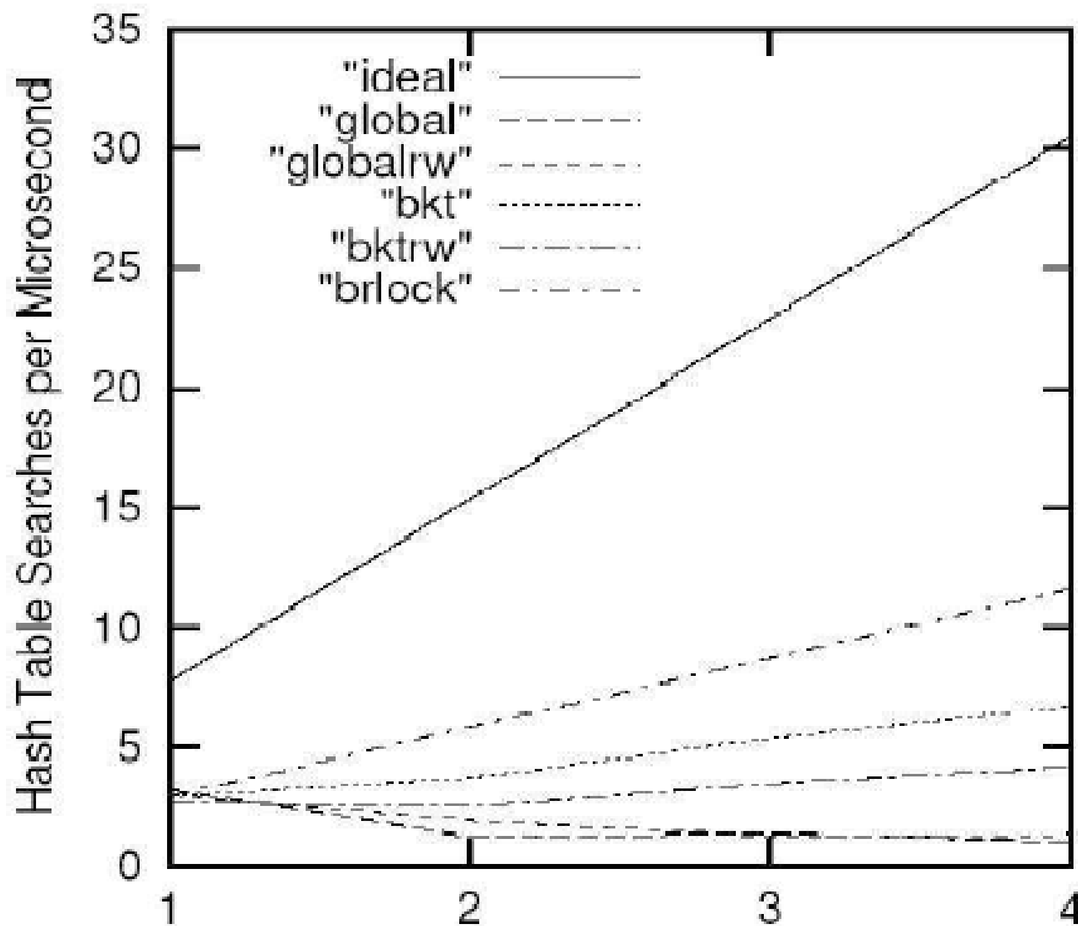


- 1984: Many cycles per instruction
- 2005: Many instructions per cycle
  - 20 stage pipelines
  - CPU logic executes instructions out-of-order to keep pipeline full
  - Synchronization instructions must not be reordered
    - Or you could execute instructions inside c.s. without completing entry instructions
  - So synchronization stalls the pipeline

Performance

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

Hash Table Microbenchmark



McKenney, 2005

- Read only
- Best case with brlock gets only ~1.5X speedup on 4 CPUs
  - Linux "Big Reader Lock", per-cpu reader lock, writers must acquire *all*
  - 4X speedup relative to degraded single CPU performance

Locks: ~~A necessary~~ evil?

- Idea: Don't lock if we don't need to
- Non-Blocking Synchronization (NBS)
  - "non blocking" refers to progress guarantees in the presence of thread failures; it does *not* mean that individual threads do not sleep or get interrupted
    - Wait-free → everyone makes progress
    - Lock-free → someone makes progress
    - Obstruction-free → someone makes progress in the absence of contention
  - We won't worry about these distinctions
    - Use *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:

```
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 *rnode = NULL;
    lock(S->stack_lock);
    if (S->top != NULL) {
      rnode = S->top;
      S->top = S->top->next;
    }
    unlock(S->stack_lock);
    return rnode;
}
```

Non-blocking stack (take 1)

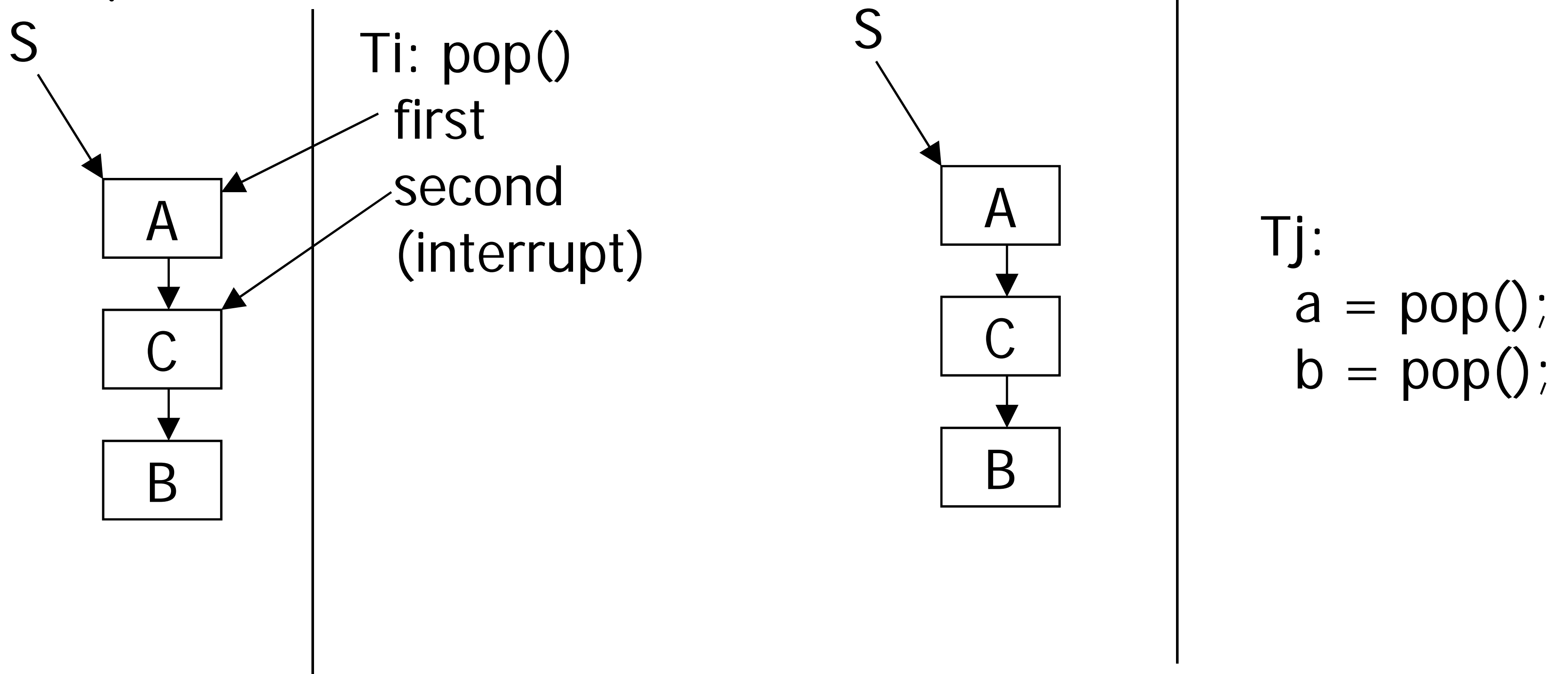
```
typedef struct node_s {
  int val;
  struct node_s *next;
} node_t;

typedef node_t *stack_t;

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;
}
```

What's wrong?

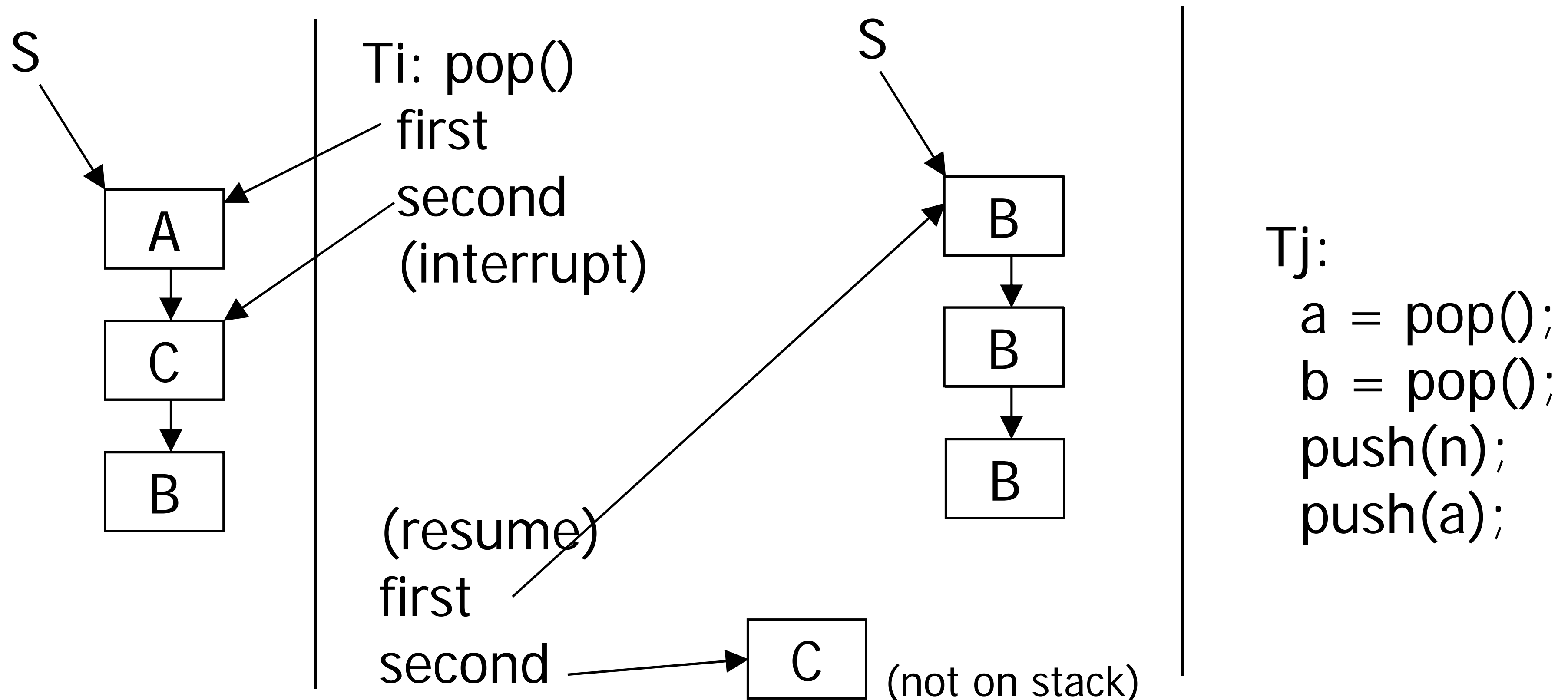
ABA Problem

- Ti, Tj both doing pops and pushes, interleaved as follows:



ABA Problem

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



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 (not every architecture provides this)
  - 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?

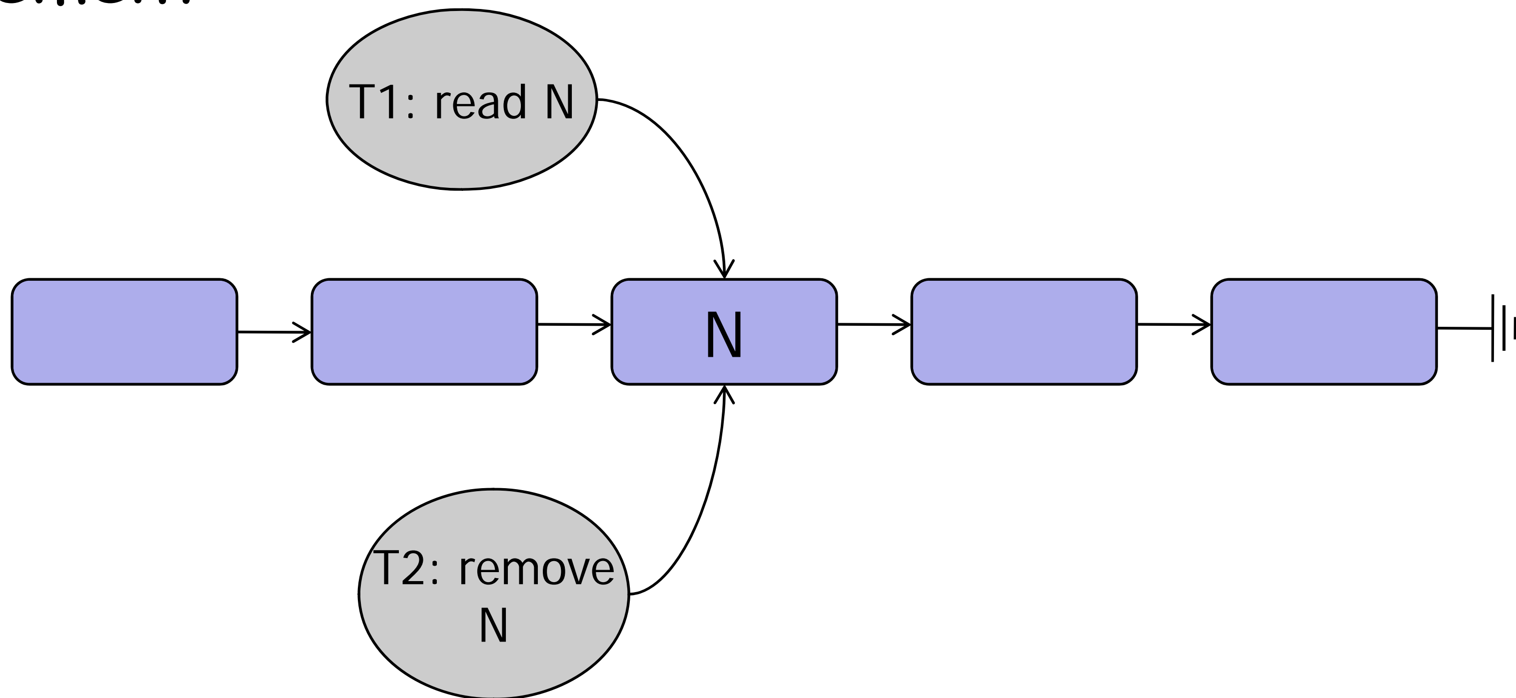
Read-Copy Update (RCU)

- 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")

U of T

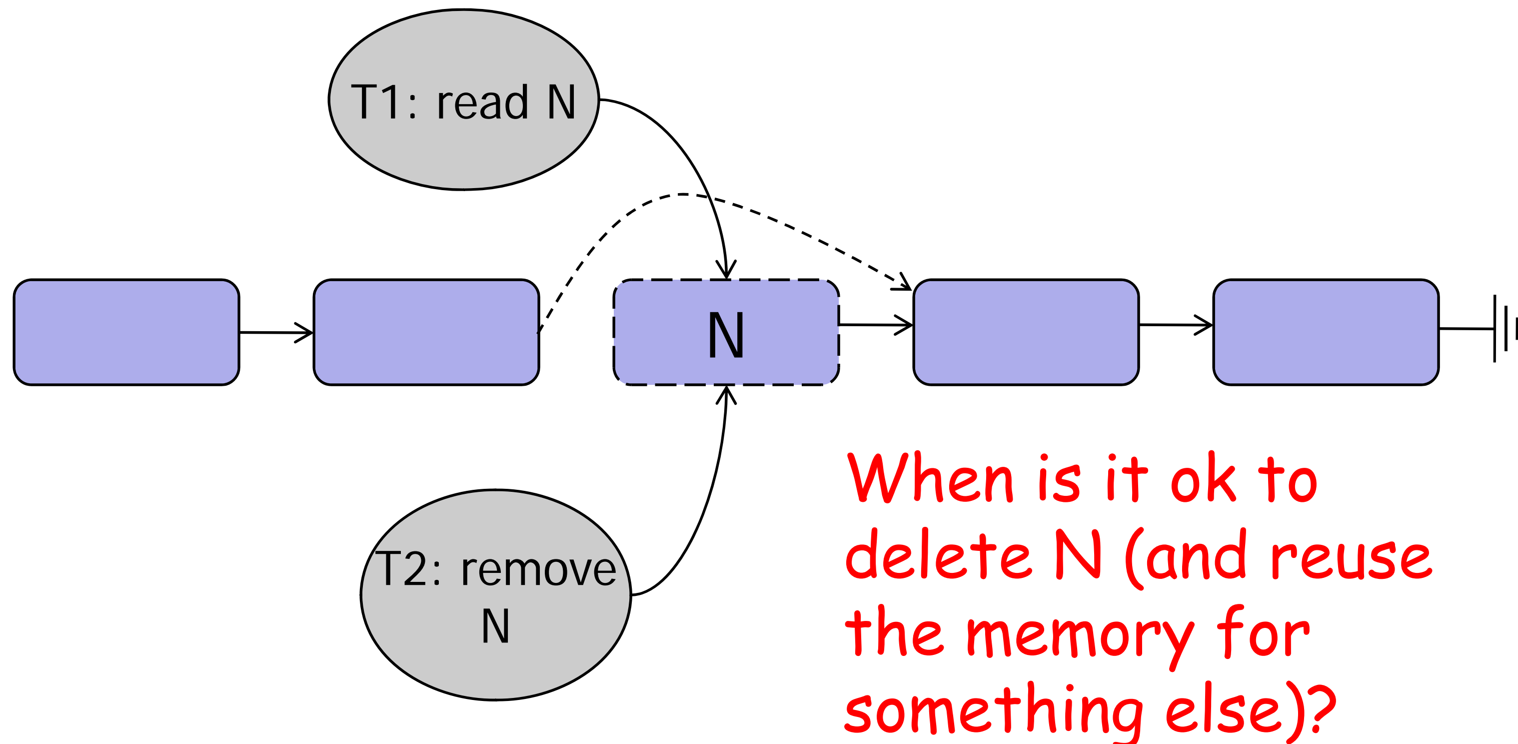
RCU Example

- T1 traversing linked list, T2 removes an element:



RCU Example (2)

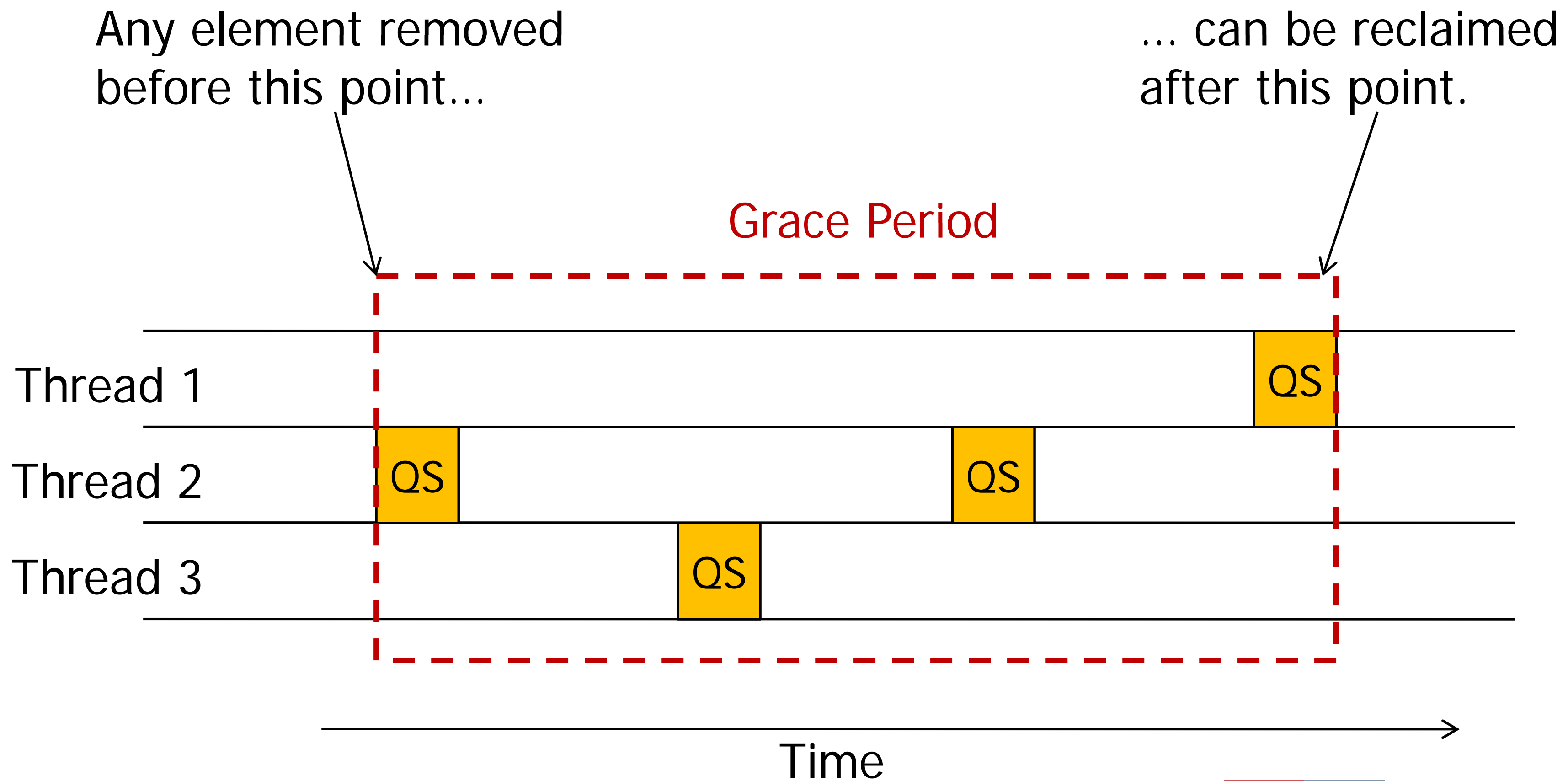
- After removal – T1 continues to use N and later nodes in the list



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 any 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

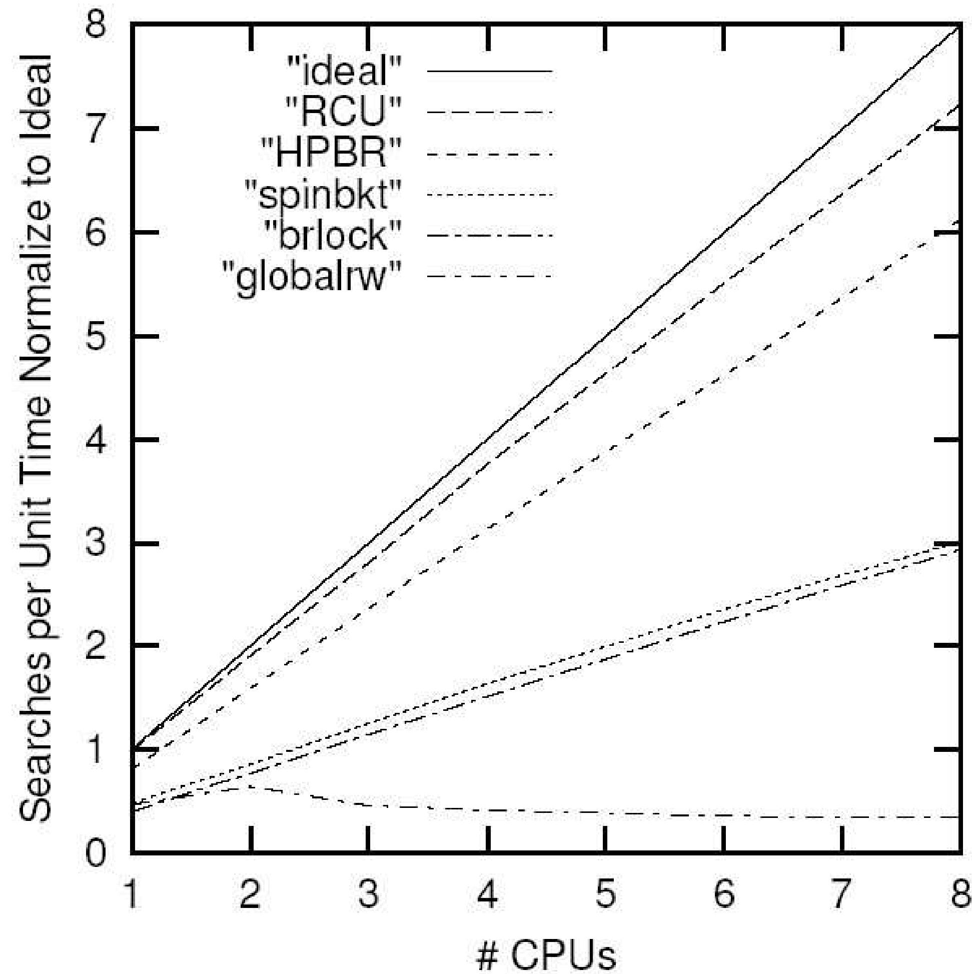
Illustration



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
    - Quiescent state may not occur inside read-side critical section
  - synchronize_rcu()
    - Wait until all pre-existing RCU read-side critical sections complete

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