Concurrency and Synchronization

CS 202: Advanced Operating Systems

UCR UNIVERSITY OF CALIFORNIA, RIVERSIDE

(some slides credit to Christo Wilson)

Classic Example

- Consider a bank application with a withdrawal function:

```
withdraw (account, amount) {
    local var = get_balance(account);
    var = var – amount;
    put_balance(account, var);
    return var;
}
```

  – Multi-threaded; each customer request is handled in a separate thread

- What happen if two people try to withdraw money from the same shared account at the same time?

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

Account balance = $100

Amount to withdraw = $50

Execution sequence

Thread A

```
local var = get_balance(account);
var = var – amount;
```

Thread B

```
local var = get_balance(account);
var = var – amount;
put_balance(account, var);
```

Context switch

Thread A

```
put_balance(account, var);
```

- What is the balance of the account now?

Race Conditions

- The previous example shows a race condition
  - Two threads “race” to execute code and update shared (dependent) data
  - Errors emerge based on the ordering of operations, and the scheduling of threads
  - Thus, errors are nondeterministic

Example: Linked List

```
elem pop(&list):
    tmp = list
    list = list->next
    tmp->next = NULL
    return tmp
```

```
push(&list, elem):
    elem->next = list
    list = elem
```

- What happens if one thread calls pop(), and another calls push() at the same time?

Thread 1 (pop)	**Thread 2 (push)**
1. tmp = list	
	2. elem->next = list
3. list = list->next	
	4. list = elem
5. tmp->next = NULL	

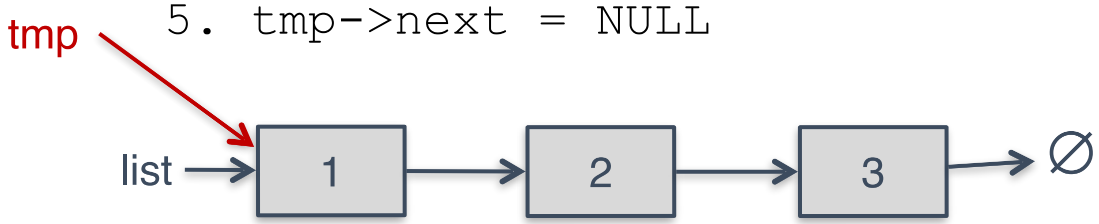
Example: Linked List

```
elem pop(&list):
    tmp = list
    list = list->next
    tmp->next = NULL
    return tmp
```

```
push(&list, elem):
    elem->next = list
    list = elem
```

- What happens if one thread calls pop(), and another calls push() at the same time?

Thread 1 (pop)	Thread 2 (push)
1. tmp = list	
	2. elem->next = list
3. list = list->next	
	4. list = elem
5. tmp->next = NULL	



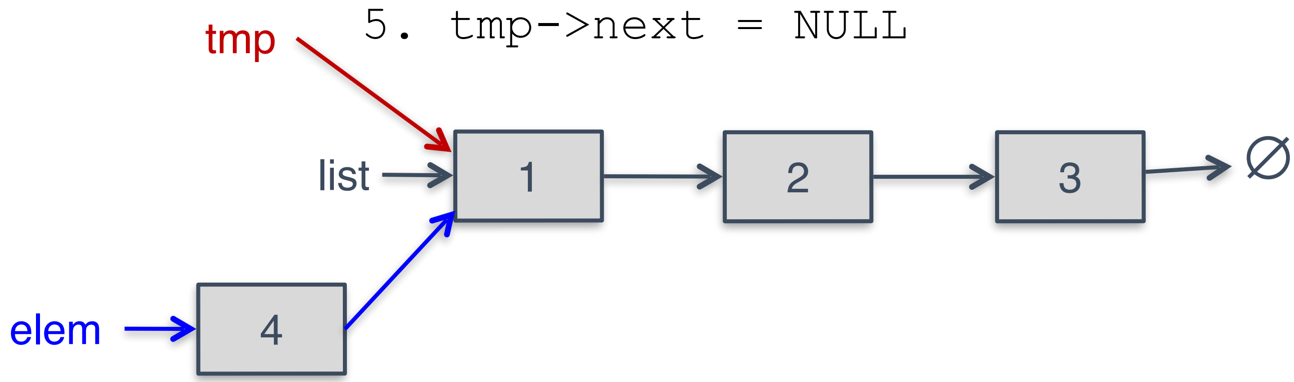
Example: Linked List

```
elem pop(&list):
    tmp = list
    list = list->next
    tmp->next = NULL
    return tmp
```

```
push(&list, elem):
    elem->next = list
    list = elem
```

- What happens if one thread calls pop(), and another calls push() at the same time?

Thread 1 (pop)	Thread 2 (push)
1. tmp = list	
	2. elem->next = list
3. list = list->next	
	4. list = elem
5. tmp->next = NULL	



(slides credit to Christo Wilson)

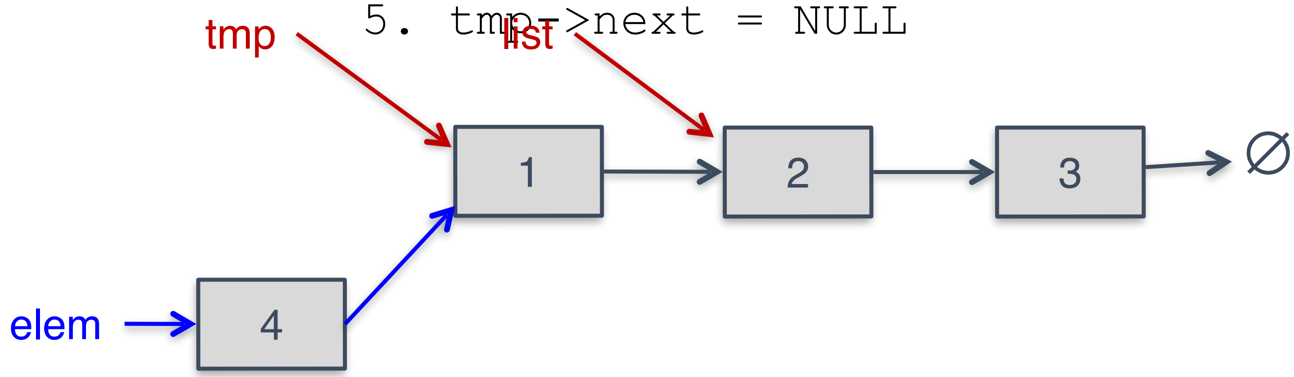
Example: Linked List

```
elem pop(&list):
    tmp = list
    list = list->next
    tmp->next = NULL
    return tmp
```

```
push(&list, elem):
    elem->next = list
    list = elem
```

- What happens if one thread calls pop(), and another calls push() at the same time?

Thread 1 (pop)	Thread 2 (push)
1. tmp = list	
	2. elem->next = list
3. list = list->next	
	4. list = elem
5. tmp->next = NULL	



(slides credit to Christo Wilson)

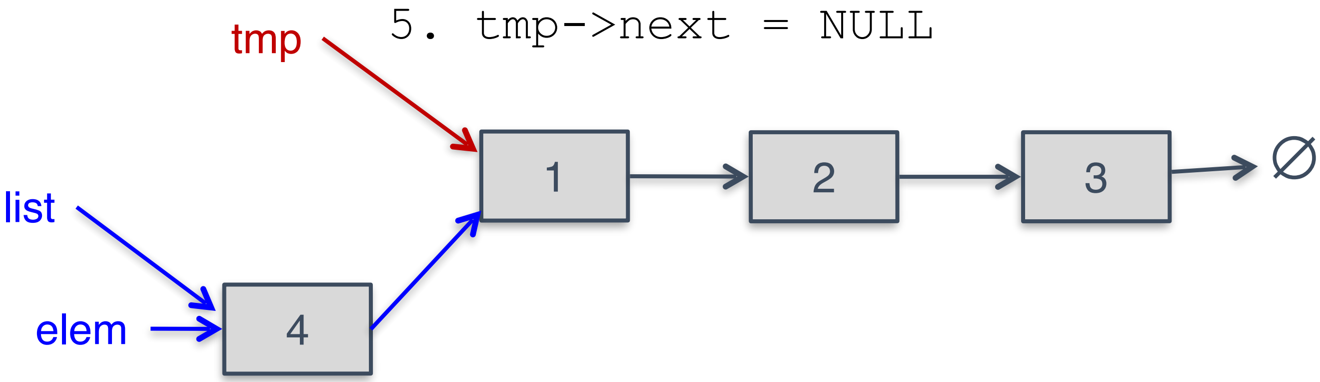
Example: Linked List

```
elem pop(&list):
    tmp = list
    list = list->next
    tmp->next = NULL
    return tmp
```

```
push(&list, elem):
    elem->next = list
    list = elem
```

- What happens if one thread calls pop(), and another calls push() at the same time?

Thread 1 (pop)	Thread 2 (push)
1. tmp = list	
	2. elem->next = list
3. list = list->next	
	4. list = elem
5. tmp->next = NULL	



(slides credit to Christo Wilson)

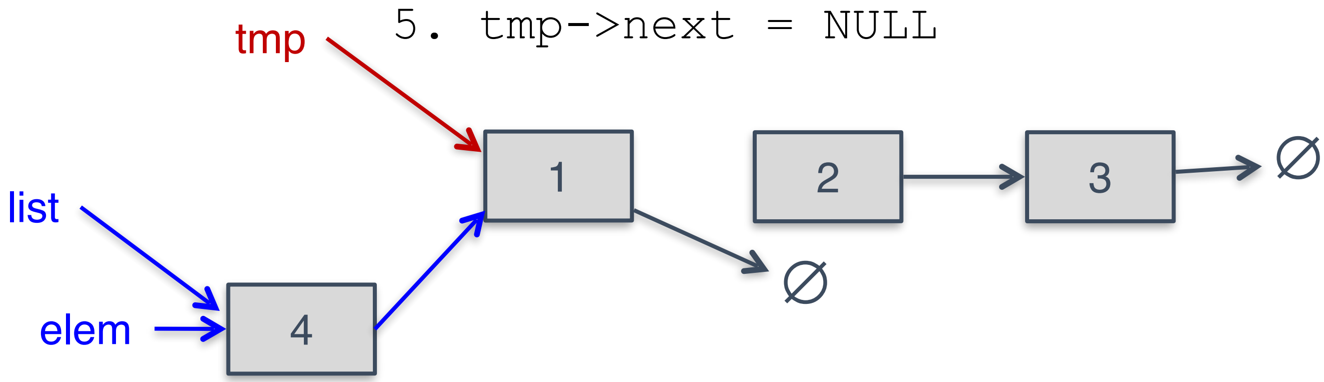
Example: Linked List

```
elem pop(&list):
    tmp = list
    list = list->next
    tmp->next = NULL
    return tmp
```

```
push(&list, elem):
    elem->next = list
    list = elem
```

- What happens if one thread calls pop(), and another calls push() at the same time?

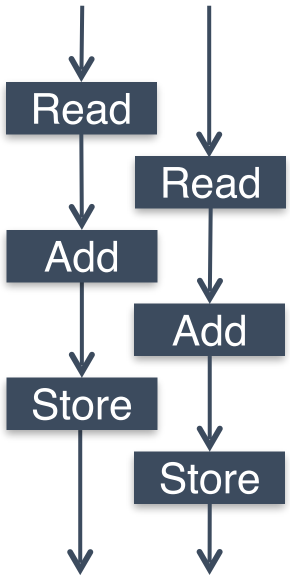
```
Thread 1 (pop)                Thread 2 (push)
1. tmp = list
                              2. elem->next =
                              list
3. list = list-
>next
                              4. list = elem
5. tmp->next = NULL
```



(slides credit to Christo Wilson)

Atomicity

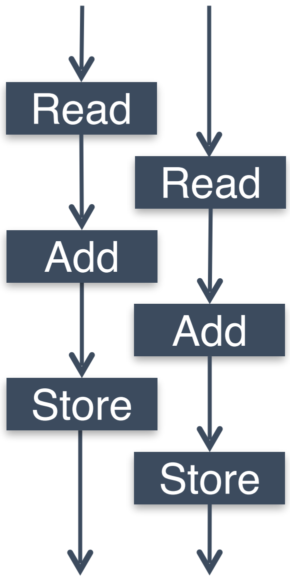
- Race conditions lead to errors when sections of code are interleaved



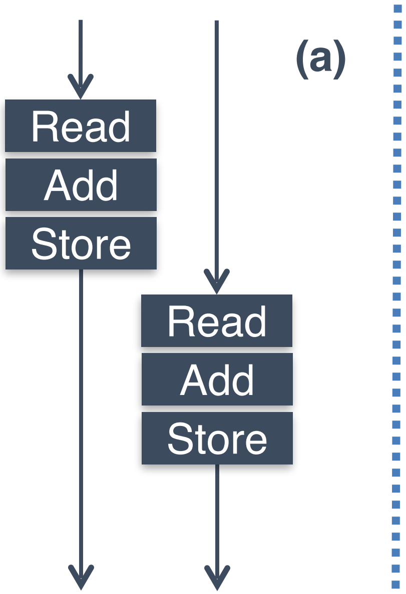
Interleaved Execution

Atomicity

- Race conditions lead to errors when sections of code are interleaved
- These errors can be prevented by ensuring code executes atomically



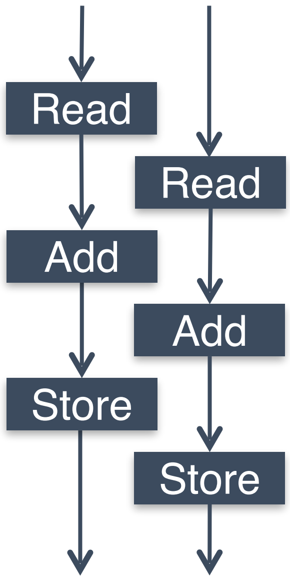
Interleaved Execution



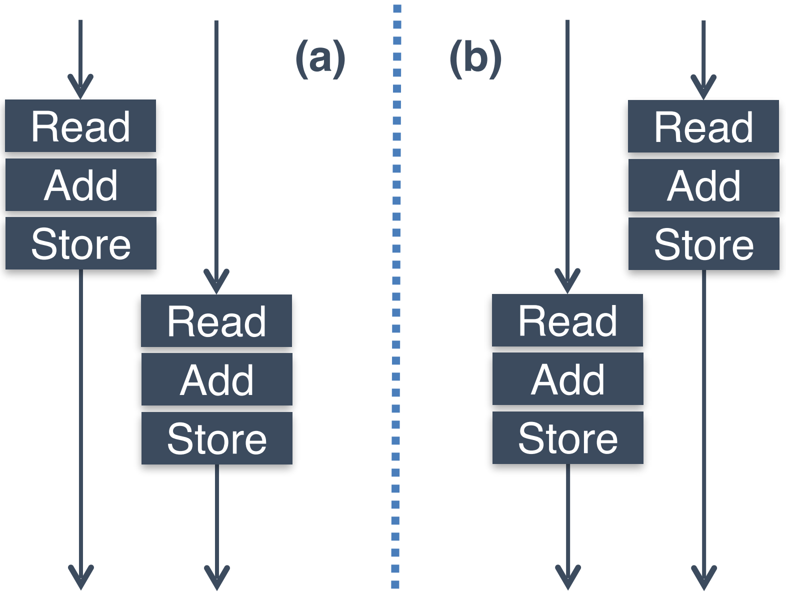
Non-Interleaved (Atomic) Execution

Atomicity

- Race conditions lead to errors when sections of code are interleaved
- These errors can be prevented by ensuring code executes atomically



Interleaved Execution



Non-Interleaved (Atomic) Execution

Discussion Questions

Which of the following best describes the root cause of a race condition?

A. Insufficient CPU scheduling priority for threads
B. Too many threads reading a variable simultaneously
C. Interleaving of thread operations modifying shared data without proper synchronization
D. Overuse of locking mechanisms reducing performance

Locks

- **Locks**: enforces ***atomicity*** in code
  - Used to synchronize access to shared resources

Locks

- **Locks**: enforces ***atomicity*** in code
  - Used to synchronize access to shared resources

- Critical section: code block that requires mutual exclusion
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: Banking application

Locks

- **Locks**: enforces ***atomicity*** in code
  - Used to synchronize access to shared resources
- Critical section: code block that requires mutual exclusion
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: Banking application
- *What requirements would you place on locks?*

Required Properties

1. Mutual exclusion
   - No two tasks may be simultaneously in critical sections accessing the same shared resource
2. Progress
   - If there is at least one process in a trying state, then eventually some process enters the critical section
3. Bounded waiting (no starvation)
   - Waiting time for a task to enter its critical section should be bounded

Using Locks

```
withdraw (account, amount) {
    acquire(lock);
    local var = get_balance(account);
    var = var – amount;
    put_balance(account, var);
    release(lock);
    return var;
}
```

Critical section

Thread A

```
acquire(lock);
local var = get_balance(account);
balance = var – amount;
```

Thread B: *wait for lock*

```
acquire(lock);
```

```
put_balance(account, var);
release(lock);
```

```
local var = get_balance(account);
var = var – amount;
put_balance(account, var);
release(lock);
```

Implementing Locks

- Typically, developers don't write their own locking-primitives
  - You use an API from the OS or a library

- Why don't people write their own locks?
  - Much more complicated than they at-first appear
  - Very, very difficult to get correct
  - May require access to privileged instructions
  - May require specific assembly instructions
    - Instruction architecture dependent

Lock-based synchronization

- Low-level synchronization primitives
  - Primitive, minimal semantics, used to build others
  1. Disabling interrupts
     - Prevent context switches in single-core systems
  2. Hardware atomic instructions (spinlock)
     - Using test-and-set, compare-and-swap instructions
  3. Software-only solutions (spinlock)
     - Dekker's algorithm, Peterson's algorithm, …
- High-level synchronization methods
  - Operation System (& Programming Language) solutions
    - Sleeping & queues to avoid starvation, priority inheritance, etc.
  - Provide some functions and data structures to the programmer
    - Semaphore, monitor, …

Disabling Interrupts

- Enabling mutual exclusion by disabling interrupts
  - Prevent preemption/context switching
  - Example: implemented by cli or sti instruction (in x86)

```
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

Disabling Interrupts

- Enabling mutual exclusion by disabling interrupts
  - Prevent preemption/context switching
  - Example: implemented by cli or sti instruction (in x86)

- Problems
  - Only available to kernel (why?)
  - What if the critical section is long?
    - Mutual exclusion is preserved but efficiency of execution is degraded
    - Can miss or delay important events
  - Works only on a single processor (how about multi-core?)
  - Not a general solution to use
    - Used to implement higher-level synchronization primitives as with spinlocks

Instruction-level Atomicity

- Modern CPUs have atomic instruction(s)
  - Enable you to build high-level synchronized objects
- Example: test-and-set instruction
  - Write (set) 1 to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation
  - The caller can then "test" the result to see if the state was changed by the call

Using Test-And-Set

- Spin lock implementation with test-and-set:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held) == 1);
}
void release (lock) {
    lock->held = 0;
}
```

Write 1 to the memory location and return its old value atomically

- When will the while return? What is the value of 'held'?

Using Test-And-Set

- Spin lock implementation with test-and-set:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held) == 1);
}
void release (lock) {
    lock->held = 0;
}
```

Write 1 to the memory location and return its old value atomically

No bounded waiting (potential starvation)

- When will the while return? What is the value of 'held'?

Notes on spin-based locks

- Good for **short** critical sections
- Potentially, waste of resources
  - Spinning wastes processor cycles and can increase contention for the lock
  - The longer the critical section, the longer the spin
- Used as building block
  - Use spinlock as primitives to build high-level synchronization constructs
  - Mutex, semaphore: suspension-based (blocking) locks
    - Overhead can be larger than spinlocks. Why?
  - Monitor, conditional variables, etc.

Semaphores

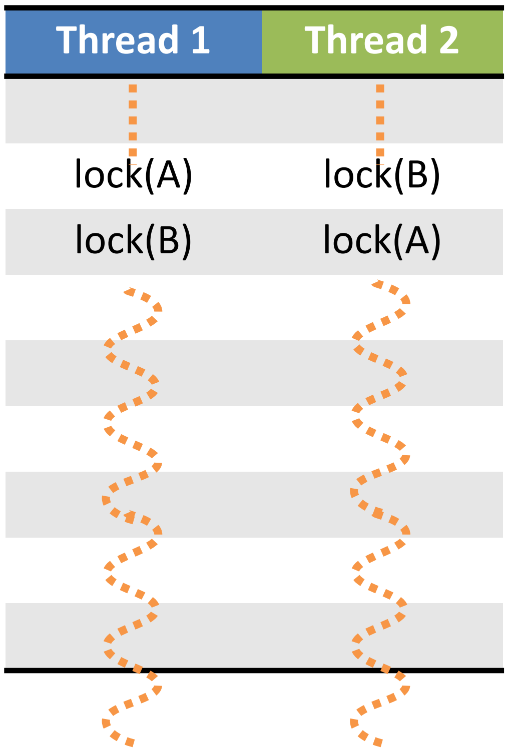
- Block waiters & leave interrupts enabled inside the critical section
  - Associated with a positive integer N (locked by up to N threads)
- wait(s): block until semaphore s is open; also called P()
- signal(s): allow another to enter; also called V()

```
semaphore s = 1; // binary semaphore; also called mutex
void wait (s) { // lock
    while (s <= 0) sleep;
    s--;
}
void signal (s) { // unlock
    s++;
    if (s > 0) wake up a waiter;
}
```

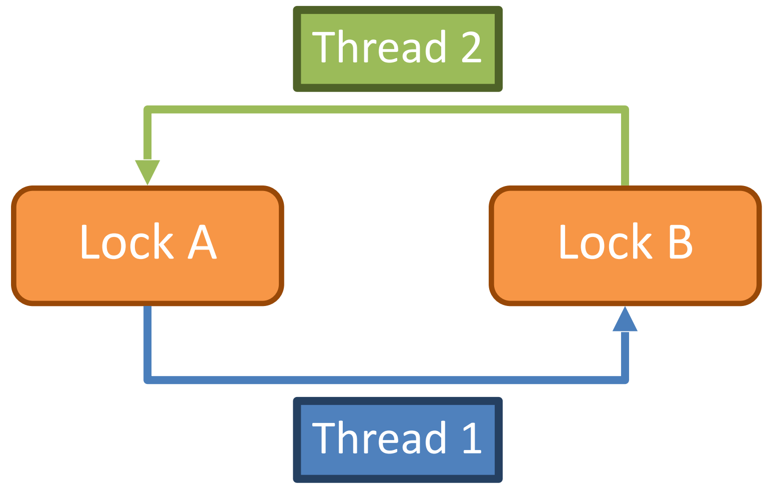
When Can Deadlocks Occur?

- Classic conditions for deadlock
  1. Mutual exclusion: resources can be exclusively held by one process
  2. Hold and wait: A process holding a resource can block, waiting for another resource
  3. No preemption on resource: one process cannot force another to give up a resource
  4. Circular wait: given conditions 1-3, if there is a circular wait then there is potential for deadlock
- Another issue:
  5. Buggy programming: programmer forgets to release one or more resources

Circular Waiting



- Simple example of circular waiting
  - Thread 1 holds lock *a*, waits on lock *b*
  - Thread 2 holds lock *b*, waits on lock *a*



Avoiding Deadlock

- If circular waiting can be prevented, no deadlocks can occur
- Technique to prevent circles: lock ranking
  1. Locate all locks in the program
  2. Number the locks in the order (rank) they should be acquired
  3. Add assertions that trigger if a lock is acquired out-of-order
- No automated way of doing this analysis
  – Requires careful programming by the developer(s)

Lock Ranking Example

	Thread 1	Thread 2

`#1: mutex A`
`#2: mutex B`
	`lock A`
	`assert(islocked(A))`
	`lock B`
	`// do something`
	`unlock B`
	`unlock A`
		`assert(islocked(A))`
		`lock B`
		`lock A`
		`// do something`
		`unlock A`
		`unlock B`

- Rank the locks
- Add assertions to enforce rank ordering
- In this case, Thread 2 assertion will fail at runtime

Read Copy Update

Review: Lock-based synchronization

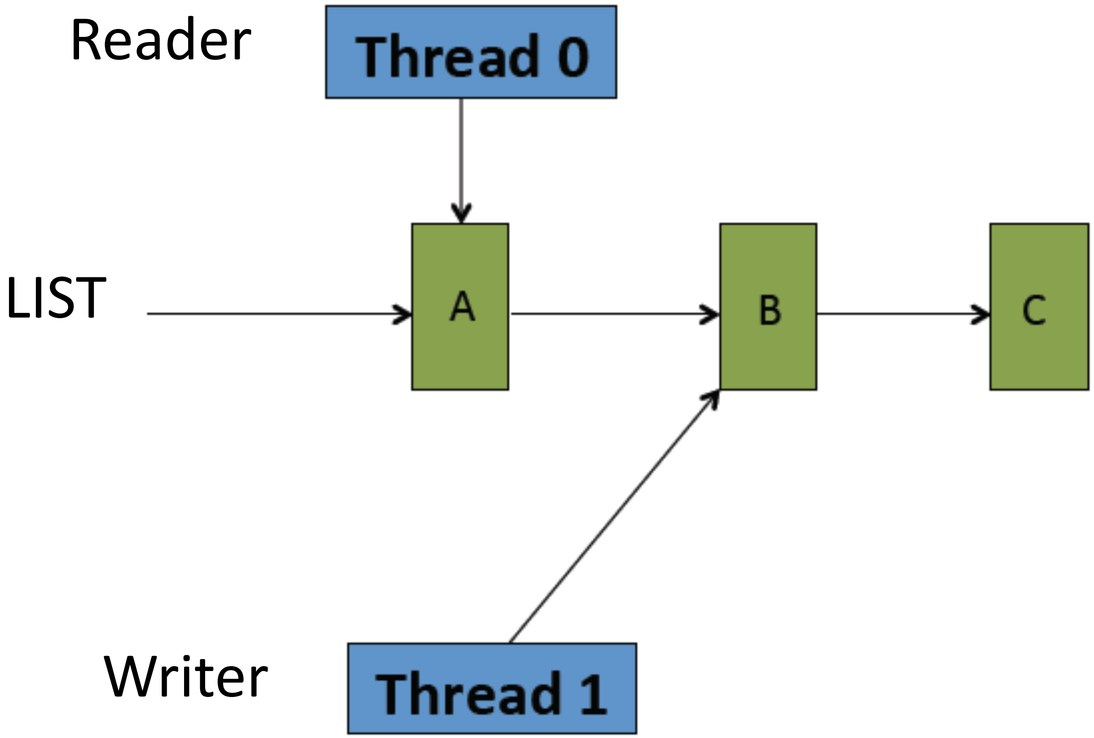
- Low-level synchronization primitives
  - Primitive, minimal semantics, used to build others
  1. Software-only solutions (spinlock)
     - Dekker's algorithm, Peterson's algorithm, …
  2. Hardware atomic instructions (spinlock)
     - Using test-and-set, compare-and-swap instructions
  3. Disabling interrupts
     - Prevent context switches in single-core systems
- High-level synchronization methods
  - Operation System (& Programming Language) solutions
    - Sleeping & queues to avoid starvation, priority inheritance, etc.
  - Provide some functions and data structures to the programmer
    - Semaphore, Monitor, …

Traditional OS locking designs

- Poor concurrency
  - Accesses to critical sections are serialized
- Locks have acquire and release cost
  - Each uses atomic operations which are expensive
  - Can dominate cost for short critical regions
  - Locks become the bottleneck
  - Other issues: deadlocks and priority inversion
- Common pattern in OS kernel
  - A lot of reads
  - Writes are rare
  - Ok to read a slightly stale copy
    - But that can be fixed too

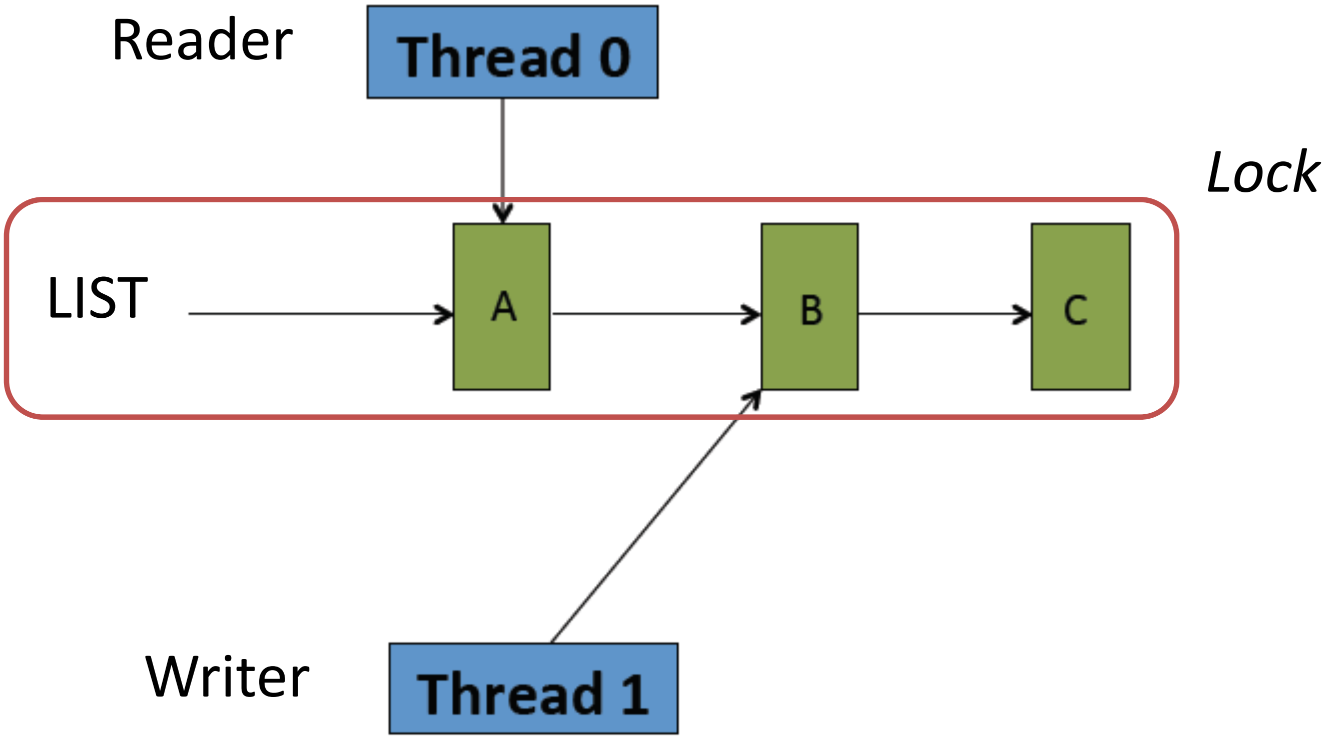
Mutex/Semaphore example

- A singly linked-list



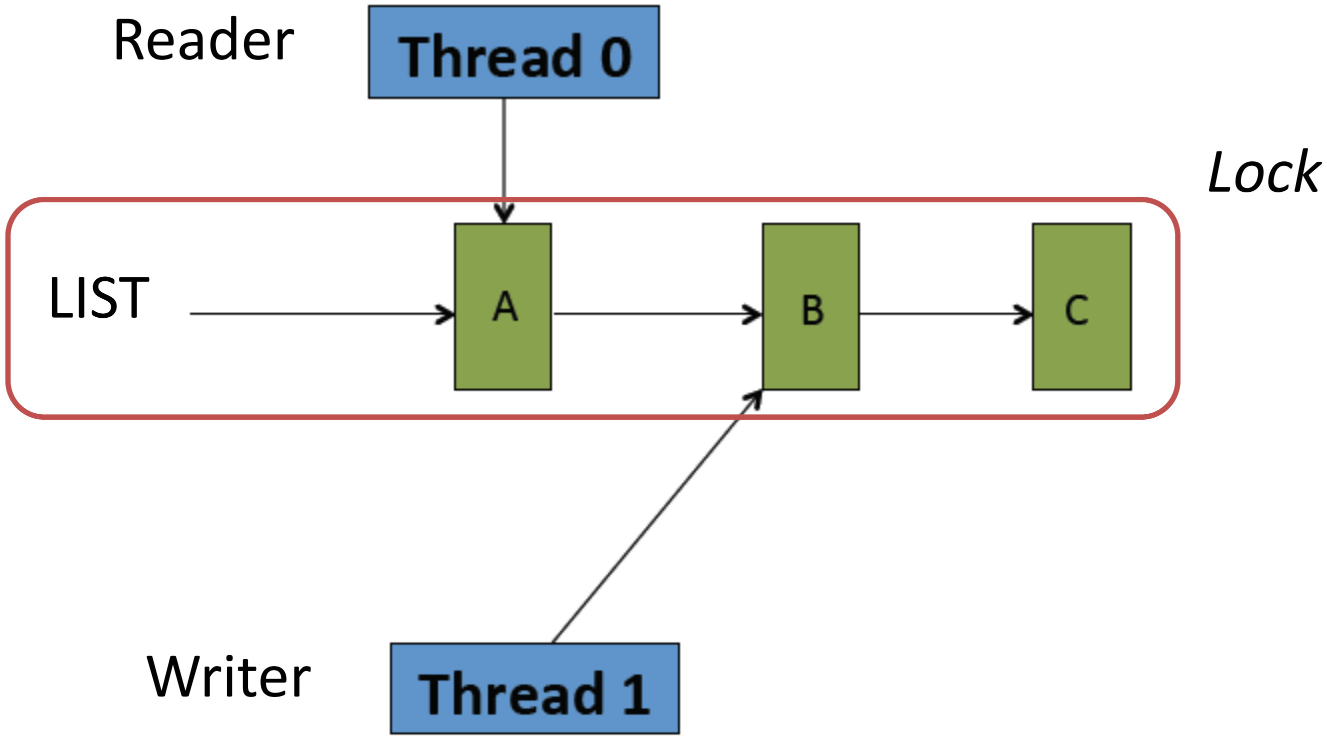
Mutex/Semaphore example

- A singly linked-list



Mutex/Semaphore example

- A singly linked-list

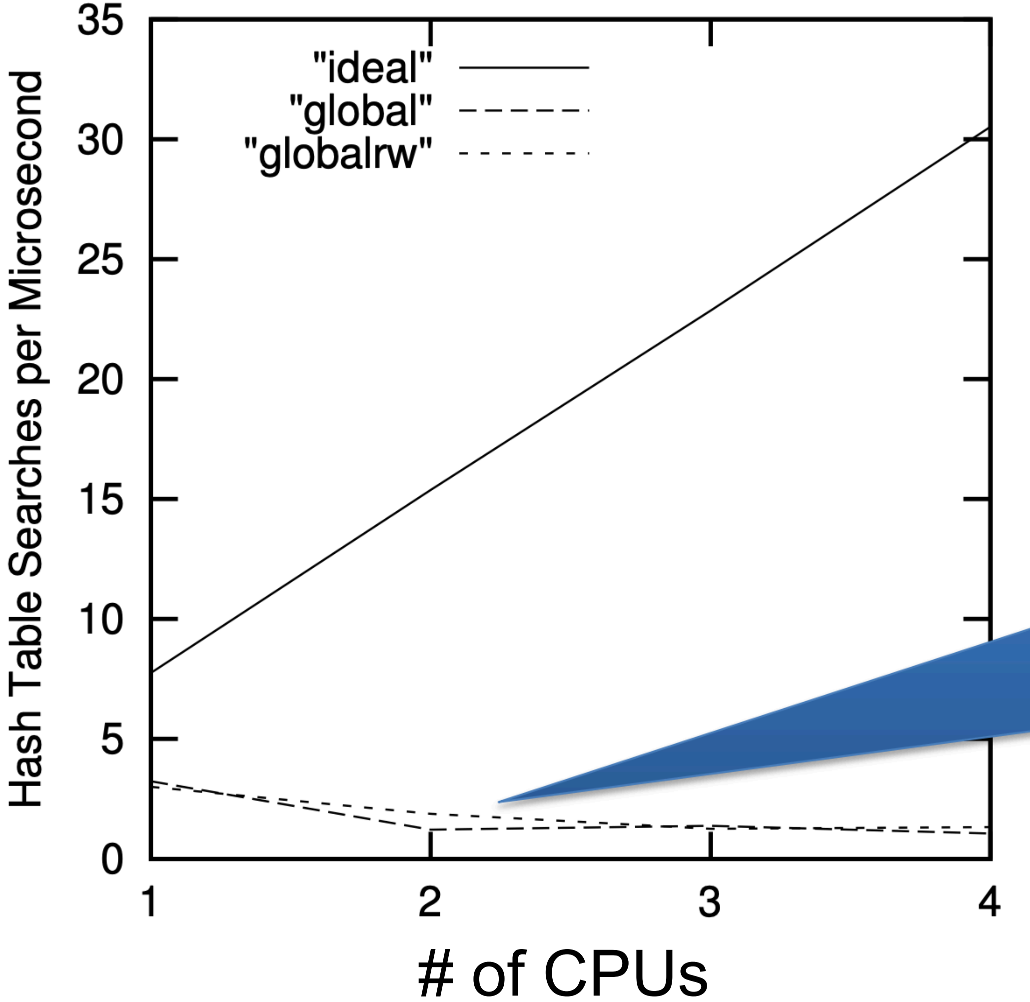


May be inefficient if it is mostly read only...

RW Locks: R/W problem

- Consider a shared database with readers & writers
  - Using a single lock is clearly inefficient
  - Like to have multiple readers at the same time & only one writer at a time
- First R/W problem (favoring reader):
  - No reader will wait even if a writer is waiting
  - Writer starvation!
  - Solutions: semaphore (mutex used to lock CS for R/W; binary Wrt lock used to block writers from entering the CS; read count lock used to count # of readers in CS and permits writer to enter when it becomes 0)
- Second R/W problem (favoring writer):
  - No new readers allowed once a writer has asked for access

Motivation behind RCU (from Paul McKenney's Thesis)



Performance of RW lock only marginally better than mutex lock

Reader-Writer Locks Limitation

- Locks have an acquire and release cost
  - Expensive atomic operations
  - May dominate performance, particularly for short critical sections
- Reader/writer locks may allow critical sections to execute in parallel
  - Still, need to serialize the increment and decrement of the read count with atomic instructions
  - Atomic instructions performance decreases as more CPUs try to do them at the same time
- The read lock itself becomes a major scalability bottleneck
- R/W lock still requires that writers wait for readers to finish

Lock-free data structures

- Do not require locks
- Good if contention is rare
- But difficult to create and error prone

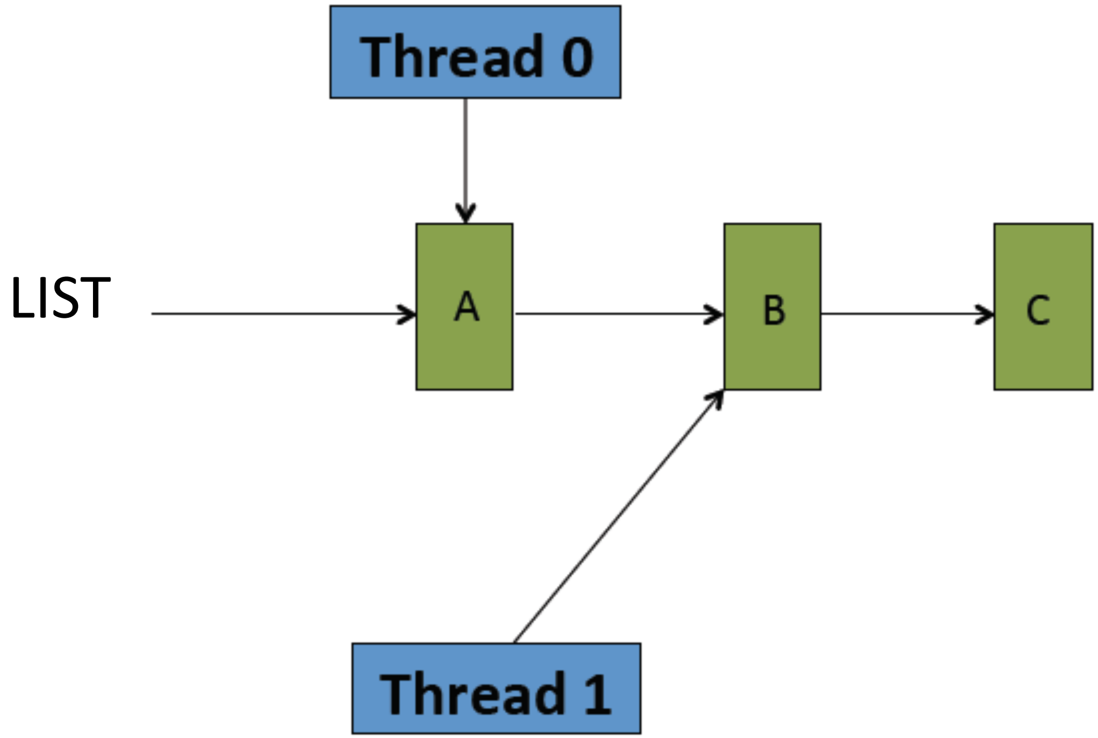
- RCU (Read-Copy Update)
  - Useful for read-mostly data structures (rare writes)
  - Read is what readers do & Copy update is what writers do
  - Replace locking in time vs. locking in space
    - Writer creates a copy (new version) of data structure offline
    - Then swaps in the new version atomically
  - RCU serializes writers using locks
    - Win if most of our accesses are reads

RCU is *not* a lock

- Readers read latest published data
  - Readers are block-free
  - No deadlock
- Writers update on a copied data and publish the new version
  - Update without blocking (if there is one writer at a time)
  - Existing readers can continue with older version
- Need garbage collection for old versions of data
- Represents a way of thinking more than a specific algorithm

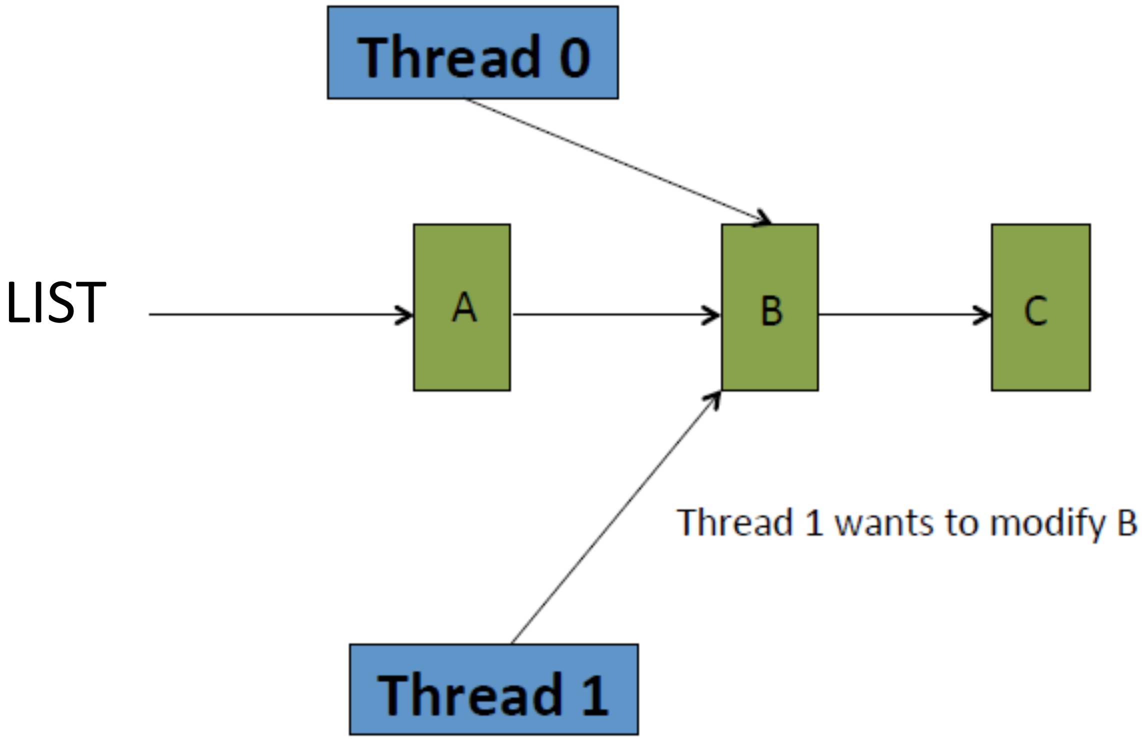
RCU Example

- A singly linked-list



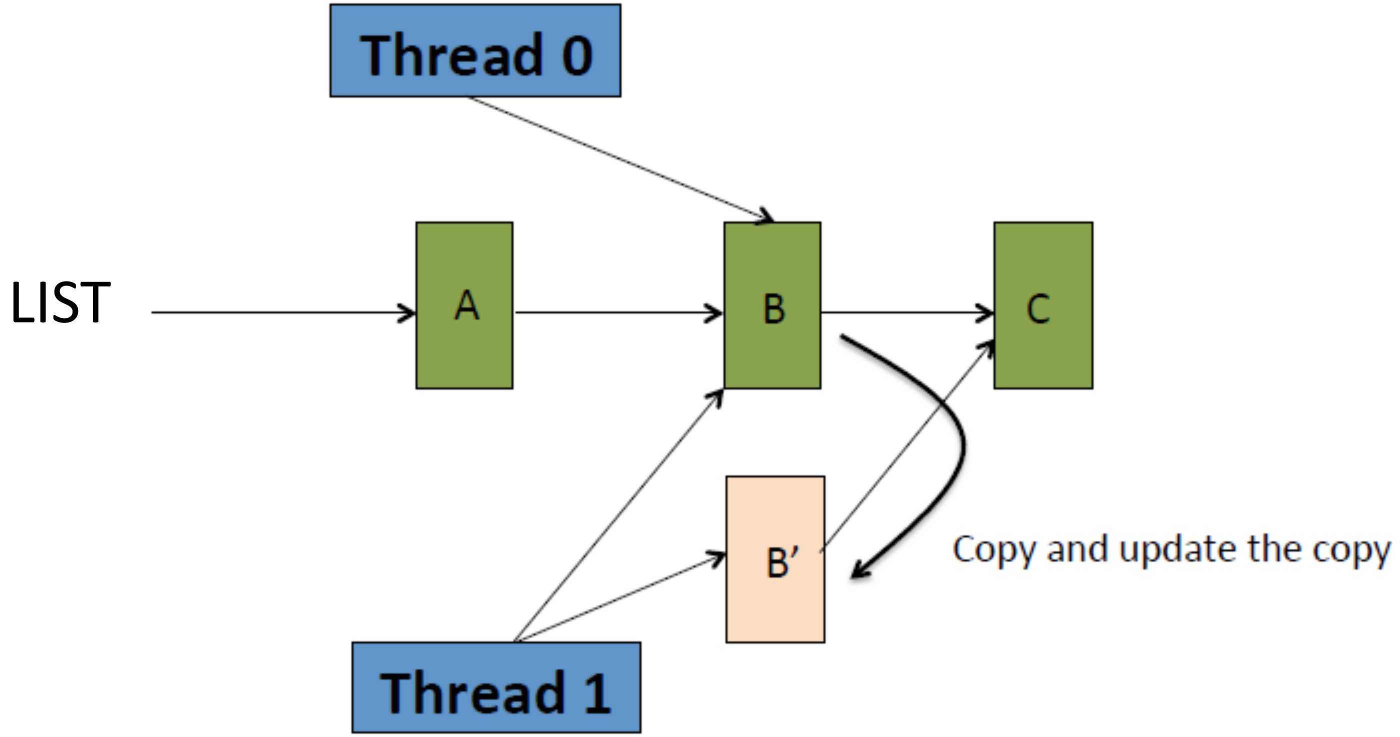
RCU Example

- A singly linked-list



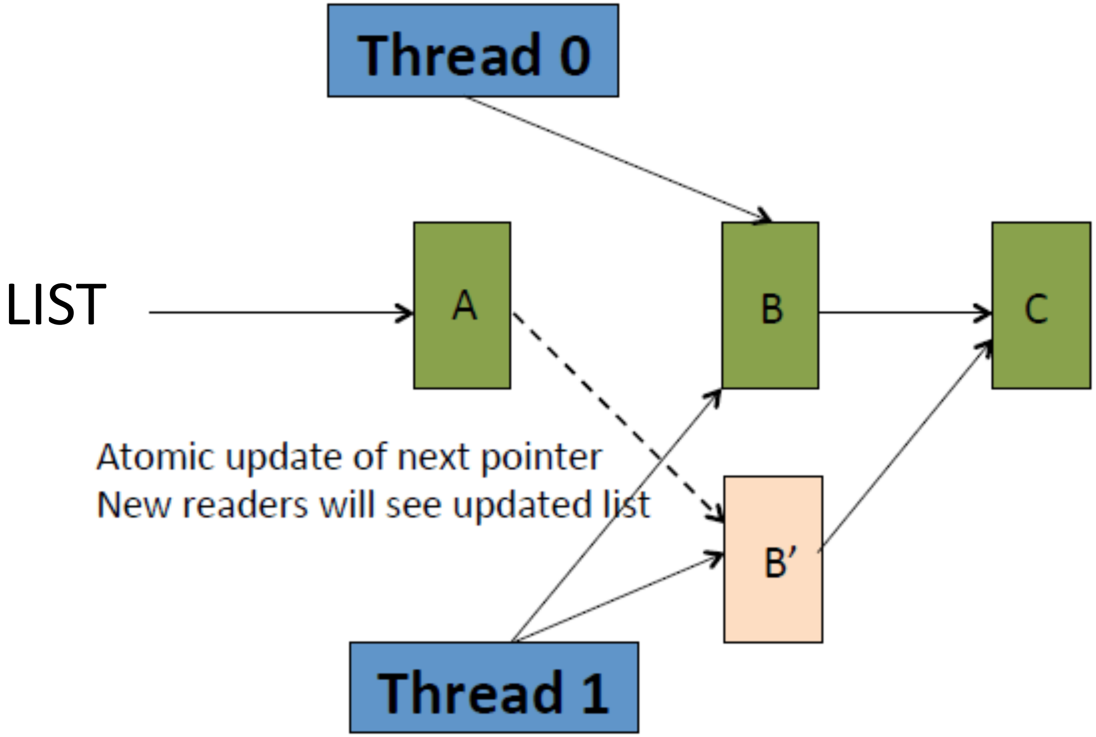
RCU Example

- A singly linked-list



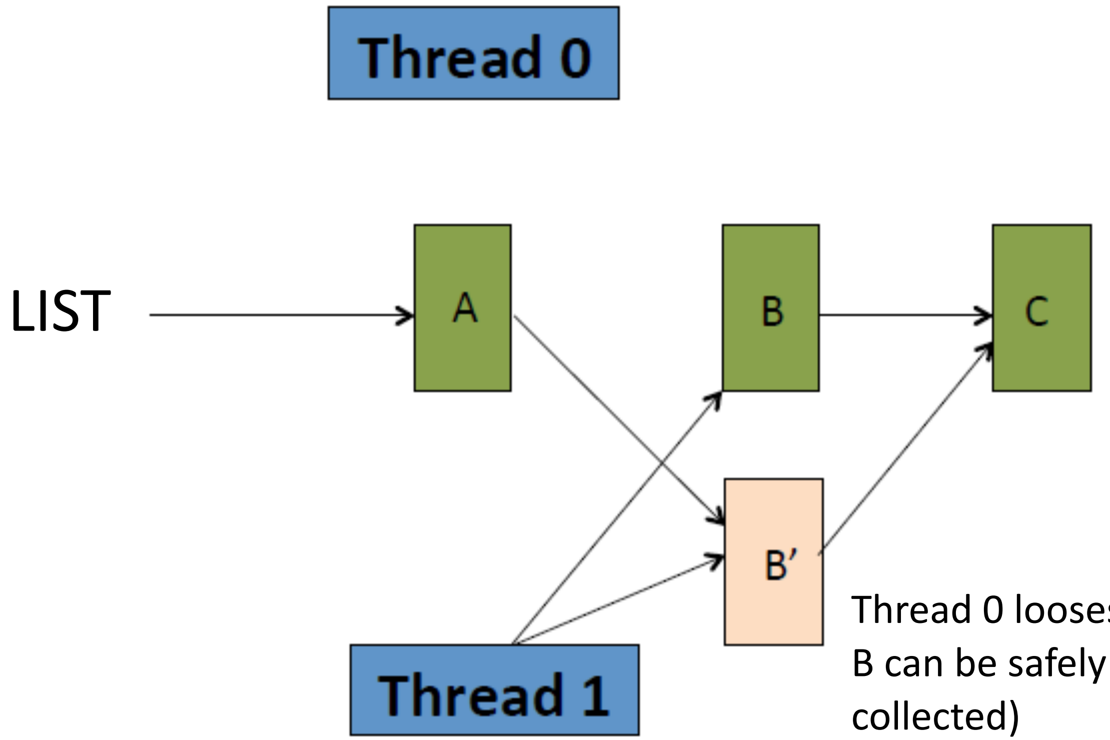
RCU Example

- A singly linked-list



RCU Example

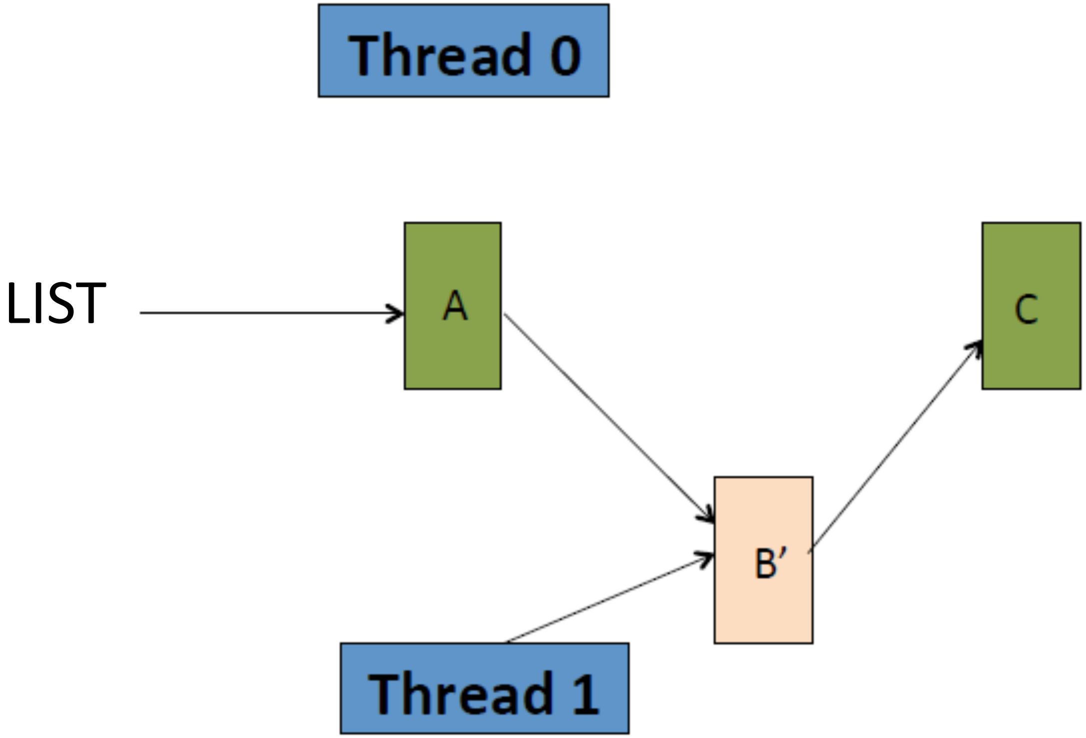
- A singly linked-list



Thread 0 looses reference to B.
B can be safely removed (garbage collected)

RCU Example

- A singly linked-list



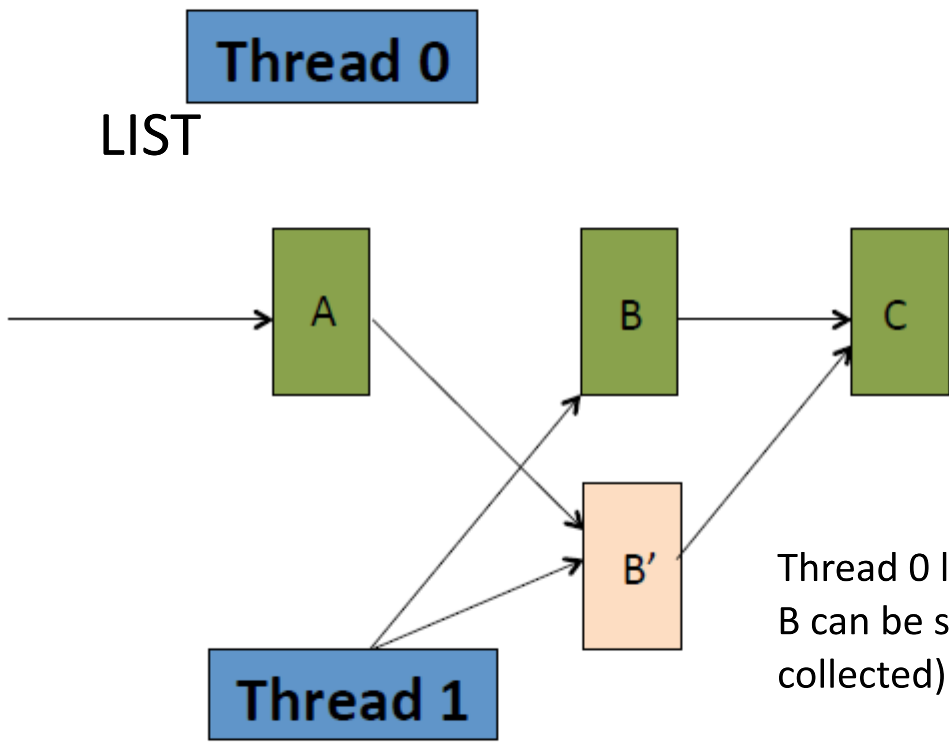
Challenges under lock-free algorithms

- One of the hardest parts of lock-free algorithms, including RCU, is concurrent changes to pointers
  - So just use locks and make writers go one at a time

- But, make writers be a bit careful so readers see a consistent view of the data structures
  - **Readers never see a half-modified or partially updated data structure, should see either before or after the write**
  - **Readers traverse valid memory and pointers**
  - **All invariants of the data structure hold during a read**
  - If 99% of accesses are readers, avoid performance-killing read lock in the common case

RCU Example

- Key idea: Carefully update the data structure so that a reader can never follow a bad pointer

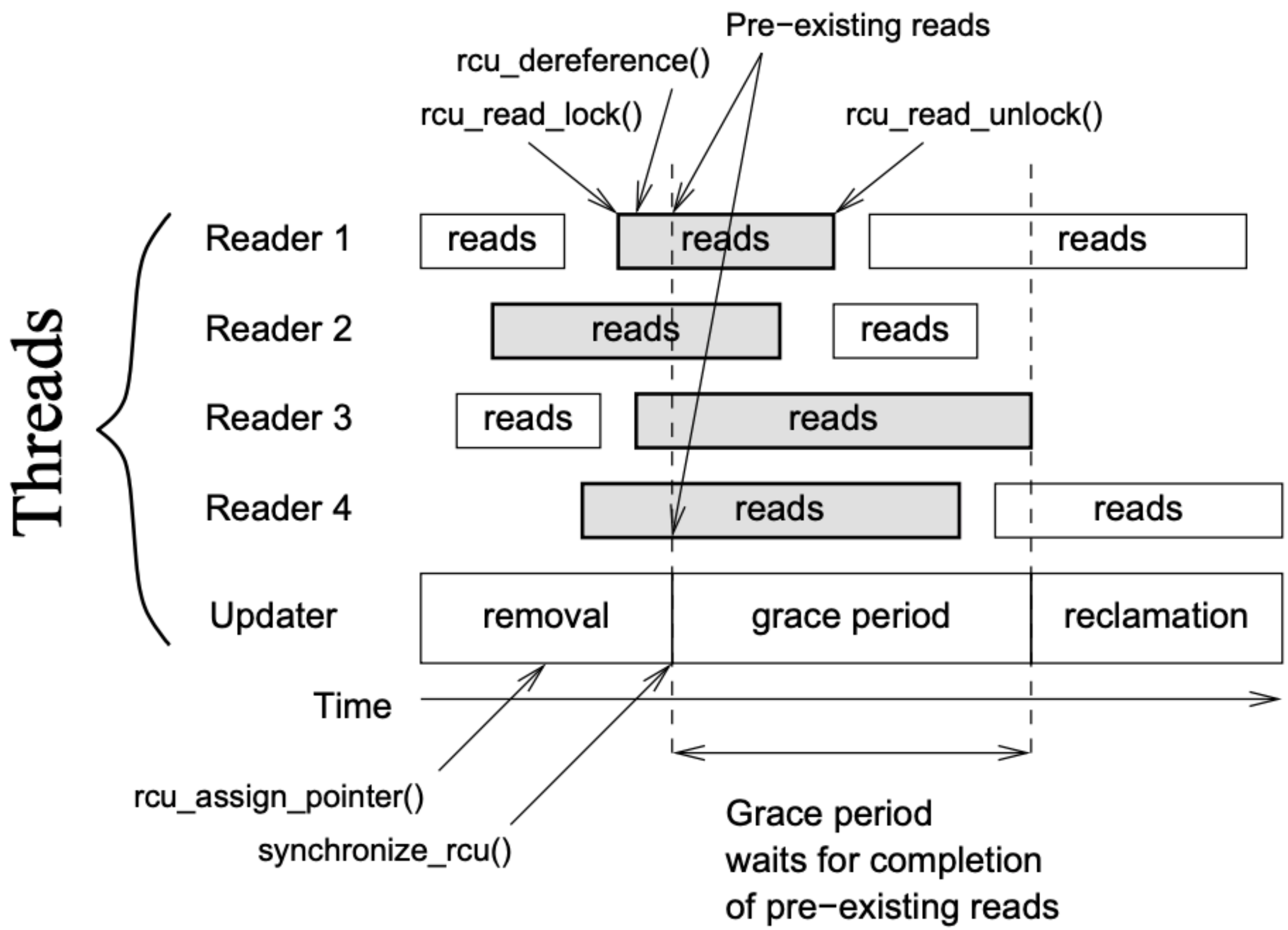


Thread 0 looses reference to B.
B can be safely removed (garbage collected)

Garbage Collection

- Part of what makes this safe is that we don't immediately free node B
  - A reader could be looking at this node
  - If we free/overwrite the node, the reader tries to follow the 'next' pointer!
- How do we know when all readers are finished using it?
  - Hint: No new readers can access this node: it is now unreachable

Grace Period



Grace Period

- Reference counting:
  - RCU employs reference counting to track how many readers are currently in their read-side critical sections
- Grace Period:
  - After a writer thread updates the shared data structure, it initiates a "grace period.”
  - The grace period is a period of time or a specific event during which RCU ensures that no new readers enter their read-side critical sections
  - The writer waits until the grace period is over.
- Reference Count Decrement:
  - After the grace period has passed, no new readers are allowed to enter their critical sections
  - The reference counter is decremented as readers exit their critical sections
- Reclamation of Old Version:
  - Safe when the reference counter associated with the old version of the data structure drops to zero

RCU Applicability

- Only a few RCU data structures in existence
  - Can RCU handle a doubly-linked list?
- Works well for singly-linked lists
  - Linked lists are the workhorse of the Linux kernel

RCU performance

- Significantly better performance in settings with many readers and few writers
- Performance highly depending on specific use cases and implementation details
  - E.g., R/W locks performs better when there are many writers

RCU usage in Linux ((from Paul McKenney)

Discussion Questions

Why is RCU able to provide excellent scalability for read-mostly workloads without using traditional mutual exclusion?

A. It relies on speculative reads that are validated later
B. It uses per-CPU reader queues and global reader clocks
C. Readers access data directly without acquiring locks, while writers defer deletion and publish updates using pointer replacement
D. Writers block until all readers complete

Discussion Questions

Why can applying RCU to complex data structures like trees or hash tables be significantly harder than applying it to simple linked lists?

A. RCU only works when data is accessed sequentially
B. Trees and hash tables cannot be copied atomically
C. Maintaining consistency and traversability for readers becomes non-trivial when writers update or restructure internal nodes
D. RCU cannot manage memory in multi-level structures

RCU Pros and Cons

- Pros
  - Readers never block
  - Updates never block
  - Extremely scalable for large number of cores
  - No deadlocks
- Widely used in Linux kernel for scalability

RCU Pros and Cons

- Cons
  - Still need to synchronize multiple concurrent writers
  - Need to maintain multiple versions – can get complex
  - A lot of implementations do not support multiple writers, even if those writers work on different parts of data without blocking each other
- Research built upon RCU
  - RCU is just the beginning
  - RLU: read log update allows multiple changes to a data to be combined into a transaction which is not seen by any reader until completion
  - Transactional memory
    - Transparently support regions of code marked as transactions by enforcing atomicity, consistency, and isolation