Read-Copy Update (RCU)

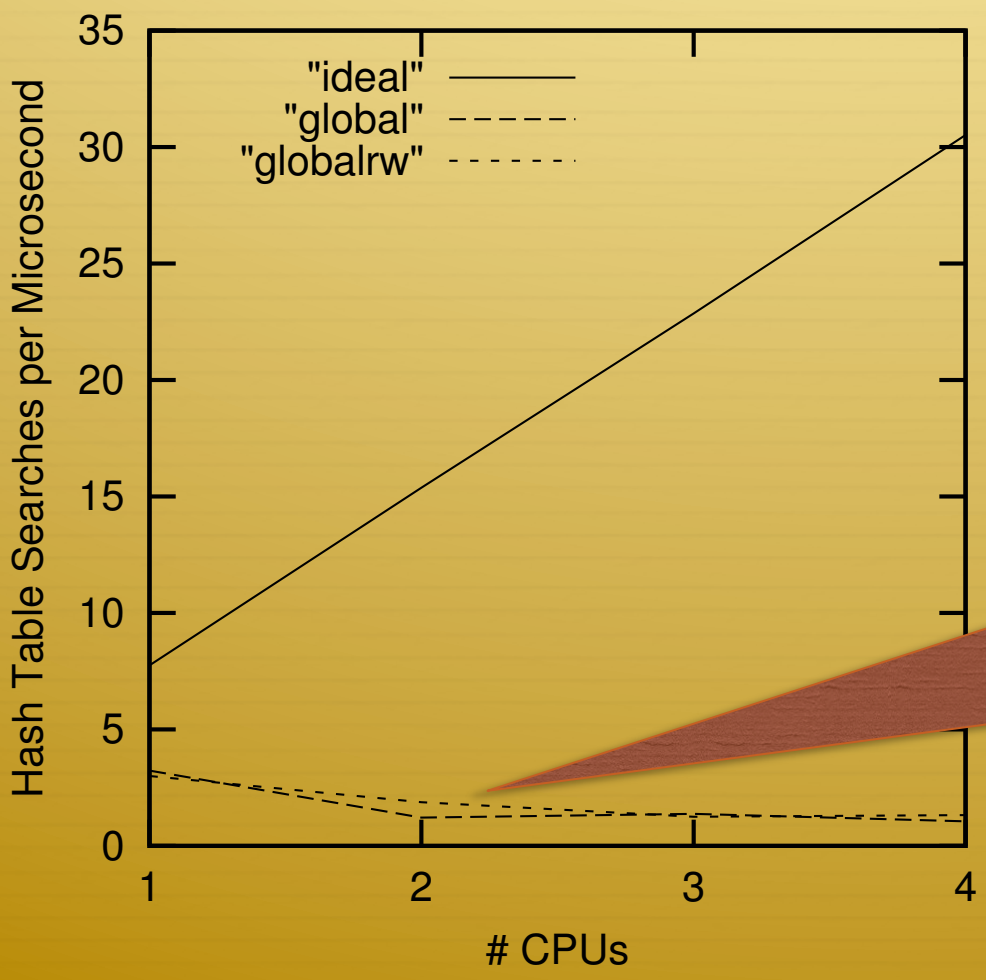
Don Porter
CSE 506

RCU in a nutshell

- Think about data structures that are mostly read, occasionally written
  - Like the Linux dcache
- RW locks allow concurrent reads
  - Still require an atomic decrement of a lock counter
  - Atomic ops are expensive
- Idea: Only require locks for writers; carefully update data structure so readers see consistent views of data

Motivation

(from Paul McKenney's Thesis)



Performance of RW lock only marginally better than mutex lock

Principle (1/2)

- Locks have an acquire and release cost
  - Substantial, since atomic ops are expensive
- For short critical regions, this cost dominates performance

Principle (2/2)

- Reader/writer locks may allow critical regions to execute in parallel
- But they still 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 scalability bottleneck, even if the data it protects is read 99% of the time**

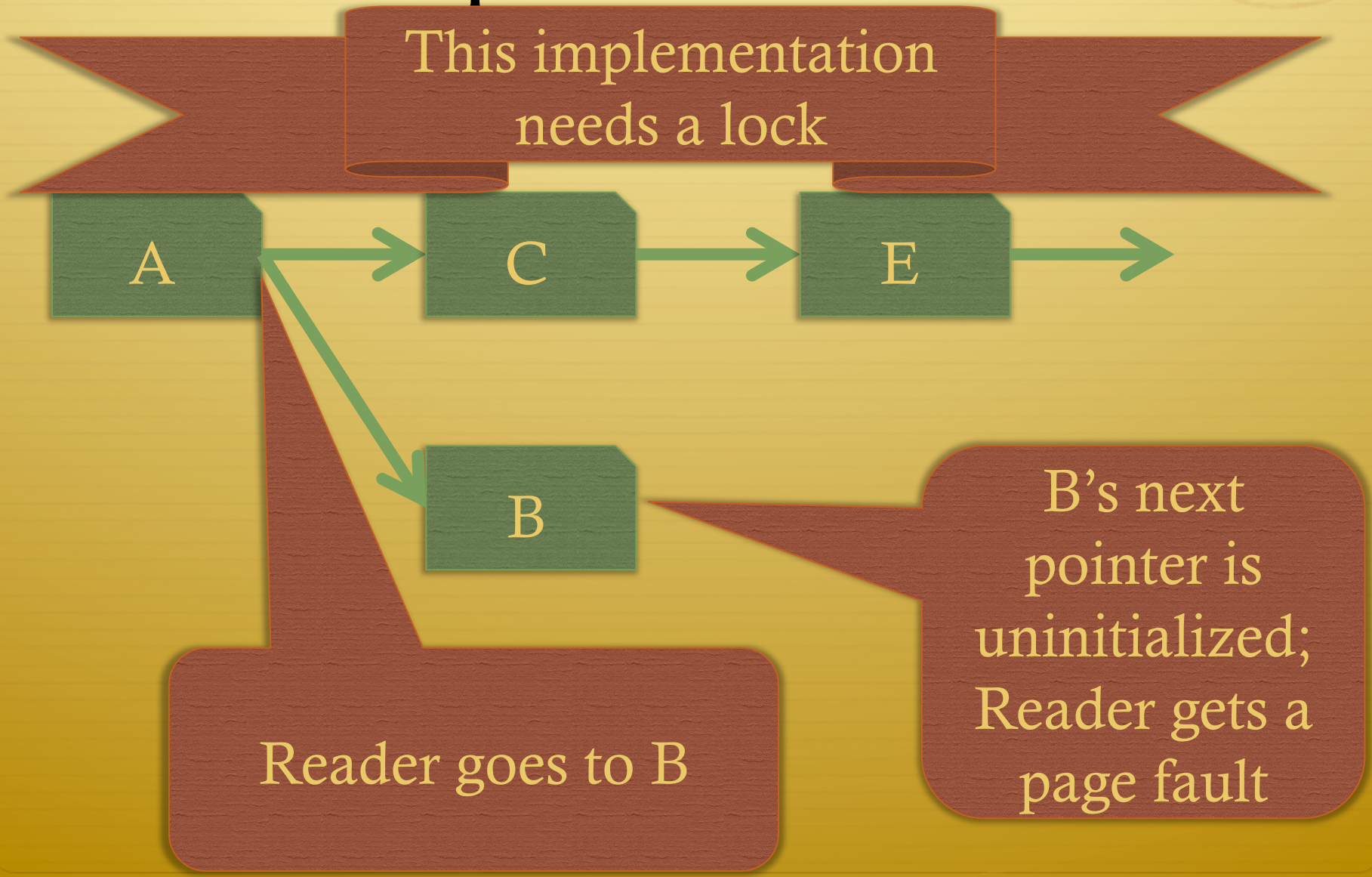
Lock-free data structures

- Some concurrent data structures have been proposed that don't require locks
- They are difficult to create if one doesn't already suit your needs; highly error prone
- Can eliminate these problems

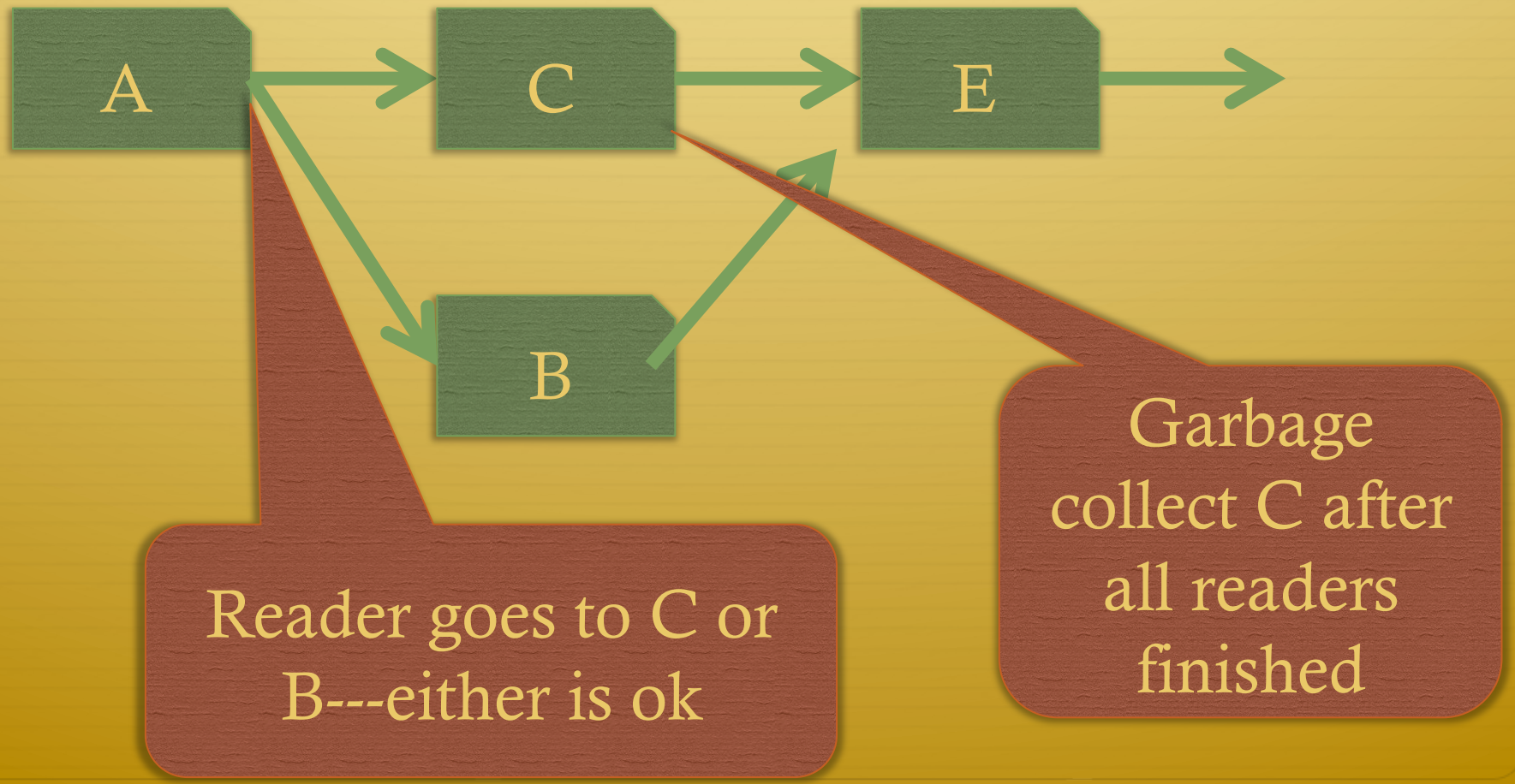
RCU: Split the difference

- One of the hardest parts of lock-free algorithms 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
- If 99% of accesses are readers, avoid performance-killing read lock in the common case

Example: Linked lists



Example: Linked lists



Example recap

- Notice that we first created node B, and set up all outgoing pointers
- Then we overwrite the pointer from A
  - No atomic instruction needed
  - Either traversal is safe
  - In some cases, we may need a memory barrier
- Key idea: Carefully update the data structure so that a reader can never follow a bad pointer

Garbage collection

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

Quiescence

- Trick: Linux doesn't allow a process to sleep while traversing an RCU-protected data structure
  - Includes kernel preemption, I/O waiting, etc.
- Idea: If every CPU has called schedule() (quiesced), then it is safe to free the node
  - Each CPU counts the number of times it has called schedule()
  - Put a to-be-freed item on a list of pending frees
  - Record timestamp on each CPU
  - Once each CPU has called schedule, do the free

Quiescence, cont

- There are some optimizations that keep the per-CPU counter to just a bit
  - Intuition: All you really need to know is if each CPU has called schedule() once since this list became non-empty
  - Details left to the reader

Limitations

- No doubly-linked lists
- Can't immediately reuse embedded list nodes
  - Must wait for quiescence first
  - So only useful for lists where an item's position doesn't change frequently
- Only a few RCU data structures in existence

Nonetheless

- Linked lists are the workhorse of the Linux kernel
- RCU lists are increasingly used where appropriate
- Improved performance!

API

- Drop in replacement for read_lock:
  - rcu_read_lock()
- Wrappers such as rcu_assign_pointer() and rcu_dereference_pointer() include memory barriers
- Rather than immediately free an object, use call_rcu(object, delete_fn) to do a deferred deletion

From McKenney and Walpole, Introducing Technology into the Linux Kernel: A Case Study

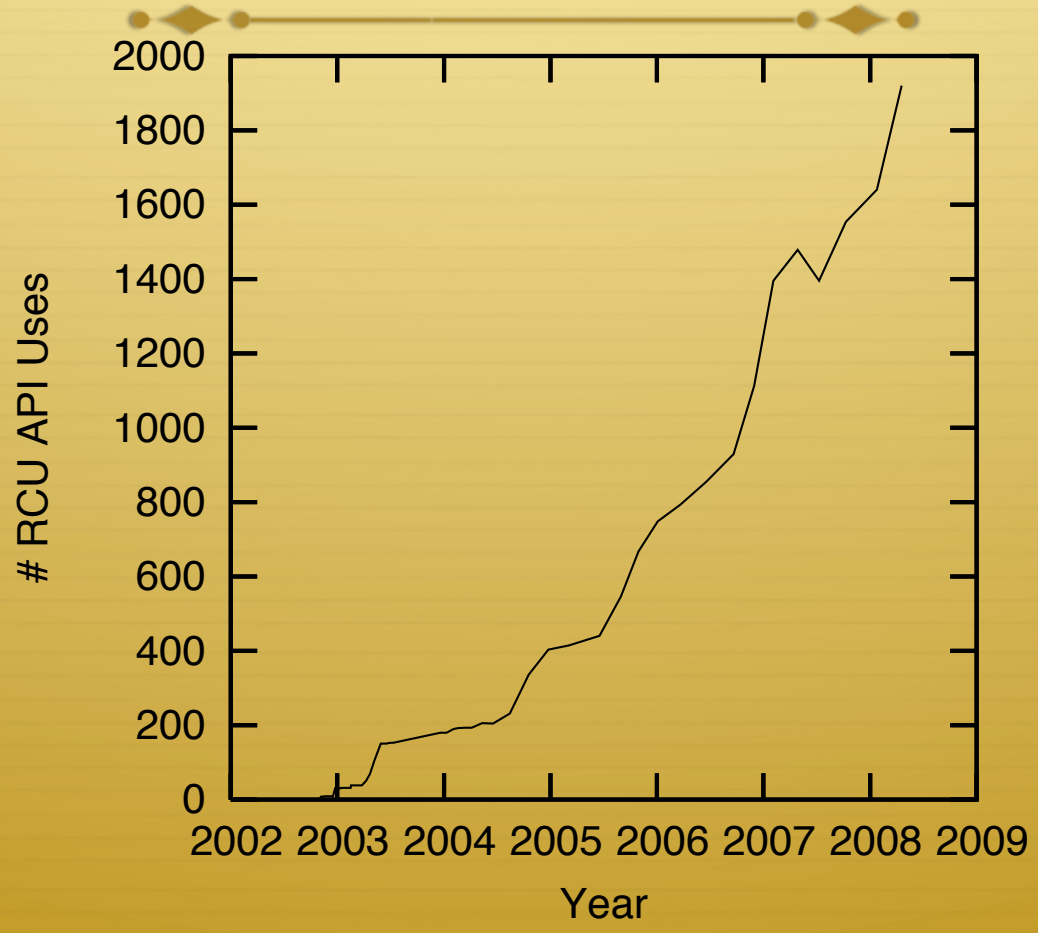


Figure 2: RCU API Usage in the Linux Kernel

Summary

- Understand intuition of RCU
- Understand how to add/delete a list node in RCU
- Pros/cons of RCU