CS 477:

Advanced Operating Systems

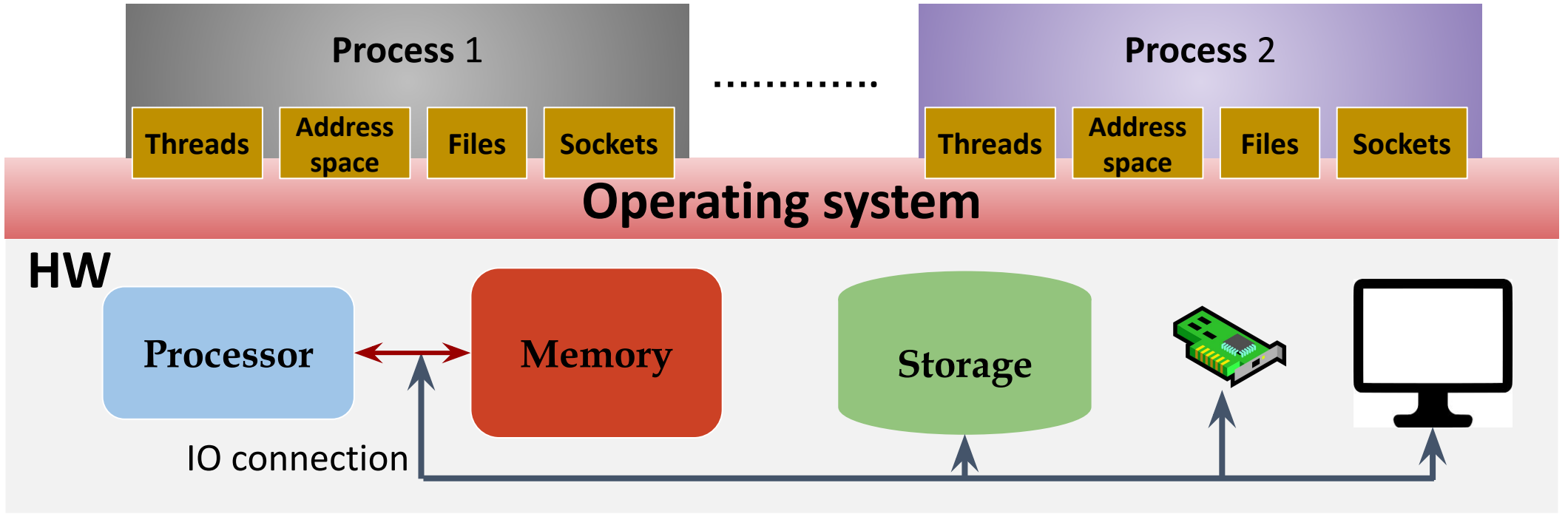
OS Concurrency

This week

- Concurrency primitives recap
- Advanced concurrency framework: Read-copy-update (RCU)
- Operating System Transactions

OS concurrency

- OS manages multiples of resources to execute multiples of tasks
  - Efficiently manage different **processes**
  - Efficiently manage various **hardware devices**



Need for mutual exclusion

Uncontrolled scheduling of threads on **shared memory**

```
1  #include<stdio.h>
2  #include<pthread.h>
3
4  int counter = 0;
5  void *incr(void *arg) {
6     printf("%s starts\n", (char *)arg);
7     for (int i=0; i < 100000; i++)
8           counter = counter + 1;
9      return NULL;
10 }
11
12 int main(int argc, char *argv[])
13 {
14     pthread_t t1, t2;
15     // Create two threads T1 and T2
16    pthread_create(&t1, NULL, incr, "T1");
17    pthread_create(&t2, NULL, incr, "T2");
18    pthread_join(t1, NULL); // Wait for T1 to finish
19    pthread_join(t2, NULL); // Wait for T2 to finish
20    printf("Counter: %d (expected: %d)\n", counter, 100000*2);
21    return 0;
22 }
```

Need for mutual exclusion

- Leads to uncontrolled scheduling of threads: **thread interleaving**
- Thread interleaving can introduce:
  - A **race condition** occurs when the timing or order of events affects the correctness of the program
  - A **data race** when one thread is accessing a mutable variable while another thread is writing to it without any synchronization
- Due to race condition, we observe different results for different executions
  - Thread interleavings leads to **non-deterministic behavior**

Ensuring atomicity through mutual exclusion

- Data race occurs on shared memory variables
- Code block that concurrently accesses shared region is called **critical section**

```
mov     0x2e95(%rip),%eax        # 4034 <counter>
add     $0x1,%eax
mov     %eax,0x2e8c(%rip)        # 4034 <counter>
```

- To ensure correct program, must mediate the access to the critical section
  - Achieve atomicity: Execute a critical section as an uninterruptible block
- **Mutual exclusion:** Only one thread can execute this critical section at any point in time, while others wait; Synchronizes the access to the critical section

Locks: basic idea

- Lock variables protect critical sections
- All threads competing for a critical section share a lock
- Only one thread succeeds at acquiring the lock (at a time): **Lock holder**
- Other threads must wait until the lock is released: **Lock waiter**

```
lock_t mutex;
...
lock(&mutex);
counter = counter + 1
unlock(&mutex)
```

A lock is a declared variable

Locks: basic idea

- Lock variables protect critical sections
- All threads competing for a critical section share a lock
- Only one thread succeeds at acquiring the lock (at a time)
- Other threads must wait until the lock is released

```
lock_t mutex;
...
lock(&mutex);
counter = counter + 1
unlock(&mutex)
```

It is either **available** (or **unlocked** or **free**)

Thus, if no thread holds/acquired the lock, the lock is **available** (or **unlocked** or **free**)

If the lock is acquired, exactly one thread holds the lock and enters the critical section

Locks: basic idea

- Lock variables protect critical sections
- All threads competing for a critical section share a lock
- Only one thread succeeds at acquiring the lock (at a time)
- Other threads must wait until the lock is released

```
lock_t mutex;
...
lock(&mutex);
counter = counter + 1
unlock(&mutex)
```

Releases the lock, which allows other threads to acquire it

Synchronization primitives rely on hw support

Hardware provides a set of instructions that provide some form of atomicity across all CPUs within a machine

- int xchg(int *ptr, int val): returns the old value and sets **val** at location ***ptr**

  int cas(int *ptr, int expected, int new): compares the value at **ptr**, and if it is equal to the **expected** value then the value is overwritten with **new**, while returning the old actual value at **ptr**

- int faa(int *ptr, int inc): fetch the old value from **ptr** and add **inc** to the value stored at **ptr**

Spin lock using compare-and-swap

```
bool lock1 = false;

void lock(bool *l) {
    while (*l); /* spin until we grab the lock */
    *l = true;
}
void unlock(bool *l) {
    *l = false
}
```

```
bool lock1 = false;

void lock(bool *l) {
    while (cas(l, false, true) == true);
    // spin and wait (do nothing)
}
void unlock(bool *l) {
    *l = false
}
```

We expect the lock value is `false` to acquire the lock. If free, we want to swap it with the new value `true` (acquired). If `cas` returns `true`, it means the lock was not free.

The behavior of test-and-set lock and compare-and-swap lock are the same:

Do not violate mutual exclusion but unfair!

Problem with spinning of spin lock

- Currently, lock waiters keep spinning until they acquire the lock
  - Also known as **busy-waiting**
- Ends up wasting CPU cycles, which could be used by other threads or processes in a system
- Some approaches to avoid the situation:
  - Lock waiters give up the CPU by **yielding**
    - The scheduler will schedule the thread after some time
  - **Mutex**: waiters go to sleep and the lock holder wakes up the waiters at the time of releasing the lock

Bugs in concurrent programs

1. **Atomicity violation bugs:** concurrent, unsynchronized modification (locks)

2. **Order-violating bugs:** Data is accessed in the wrong order (use CV)

3. **Deadlock:** Program no longer makes progress (locking order)

Atomicity violation bugs

Atomicity violation happens when *a sequence of operations that are intended to be executed atomically (as an individual unit), are interrupted,* allowing other operations to interleave, *leading to inconsistent or unexpected states in a program*.

```
Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;
```

- Thread 1 checks the value for non-**NULL** and writes it
- Thread 2 sets the value to **NULL**
- Thread 2 can execute before **`fputs`** set the value to NULL

Solution: Use a common lock between both threads when accessing the shared resource

Order violation bugs

Order violation occurs when *the expected sequence of operations is not followed due to incorrect program execution order*, *leading to incorrect program* behavior.

```
Thread 1::
void init() {
  mThread = PR_CreateThread(mMain, …);
  mThread->State = …;
}

Thread 2::
void mMain(...) {
  mState = mThread->State
}
```

- Thread 2 assumes that **`mState`** is already initialized, not **`NULL`**
- If thread 2 runs before thread 1, this will crash the program due **`NULL`** pointer dereference

Solution: Use a CV to signal that `mState` has been initialized

Deadlock

A specific condition when two or more processes are unable to proceed with their execution because each one is waiting for the other to release a resource they need.

```
Thread 1::
  pthread_mutex_lock(lock1);
  pthread_mutex_lock(lock2);

Thread 2::
  pthread_mutex_lock(lock2);
  pthread_mutex_lock(lock1);
```

- Thread 1 and thread 2 will be stuck after acquiring **`lock1`** and **`lock2`** respectively

Deadlock conditions

1. **Mutual exclusion:** Threads claim exclusive control of resources that they require
2. **Hold and wait:** A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads
3. **No preemption:** Resources cannot be preempted; that is, resources cannot be taken away from a thread unless the thread voluntarily releases them
4. **Circular wait:** There must be a circular chain of two or more threads, each of which is waiting for a resource held by the next member in the chain

Solution: Impose total ordering + obtain all resources or nothing at once + release held resources if not all available at the same time

Types of mutual exclusion

- Lock/mutex
  - Treats all access to critical section the same
- Readers-writer lock
  - Only one writer enters the critical section

OR

  - Multiple readers can enter the critical section
    - Readers do not modify the critical section
  - API: read_lock()/read_unlock() for readers; write_lock()/write_unlock() for writers

This week

- Concurrency primitives recap
- **Advanced concurrency framework: Read-copy-update (RCU)**
- Operating System Transactions

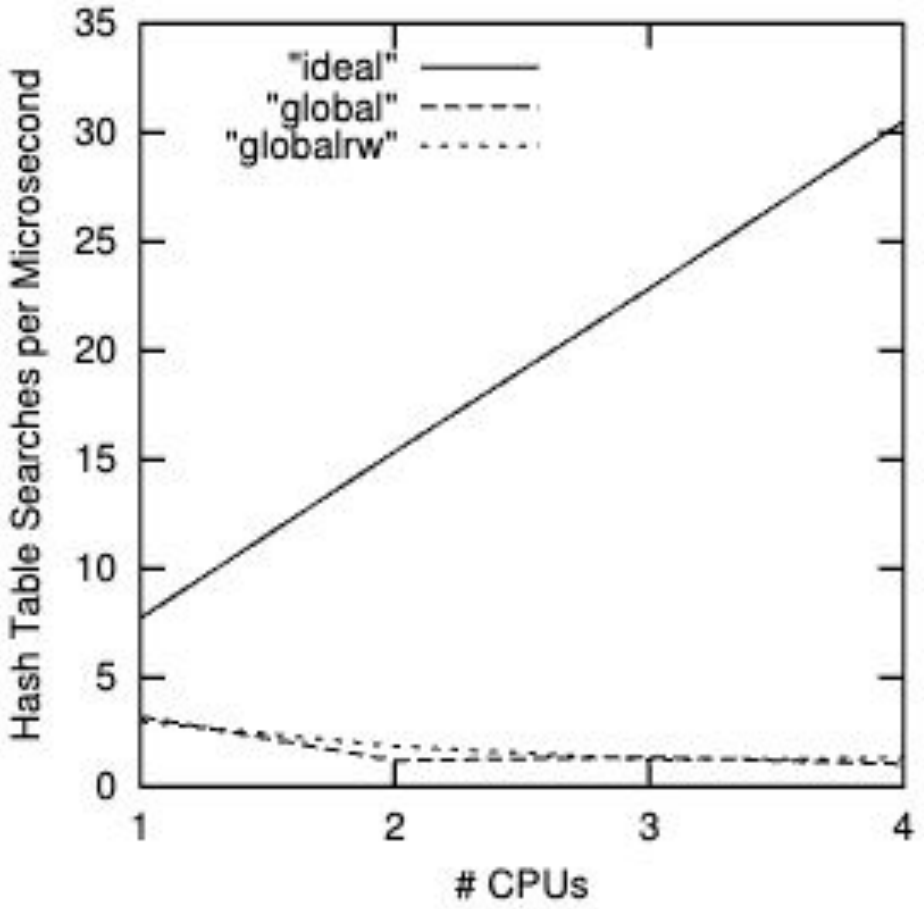
RCU in a nutshell

- Several data structures are **mostly read, occasionally written**
  - Ex: Linux dentry cache
- Readers-writer lock still allow concurrent reads
  - Still require an atomic decrement of a lock counter
  - Atomic ops are expensive

Idea: Only acquire locks for writers; carefully update data structure so readers see consistent views of data

RCU motivation

- Performance of readers-writer lock is marginally better than mutex



Issue with locks/rwlocks

- Locks have an acquire and release cost
  - Substantial, since atomic ops are expensive
- For short critical sections, the cost dominates performance
- Readers-writer locks may allow critical sections to execute in parallel
  - Serialize the increment and decrement of the read count with atomic instructions
  - Atomic instructions performance decreases with increasing CPU count
- **The read lock itself becomes the scalability bottleneck, even if the data it protects is read 99% of the time**

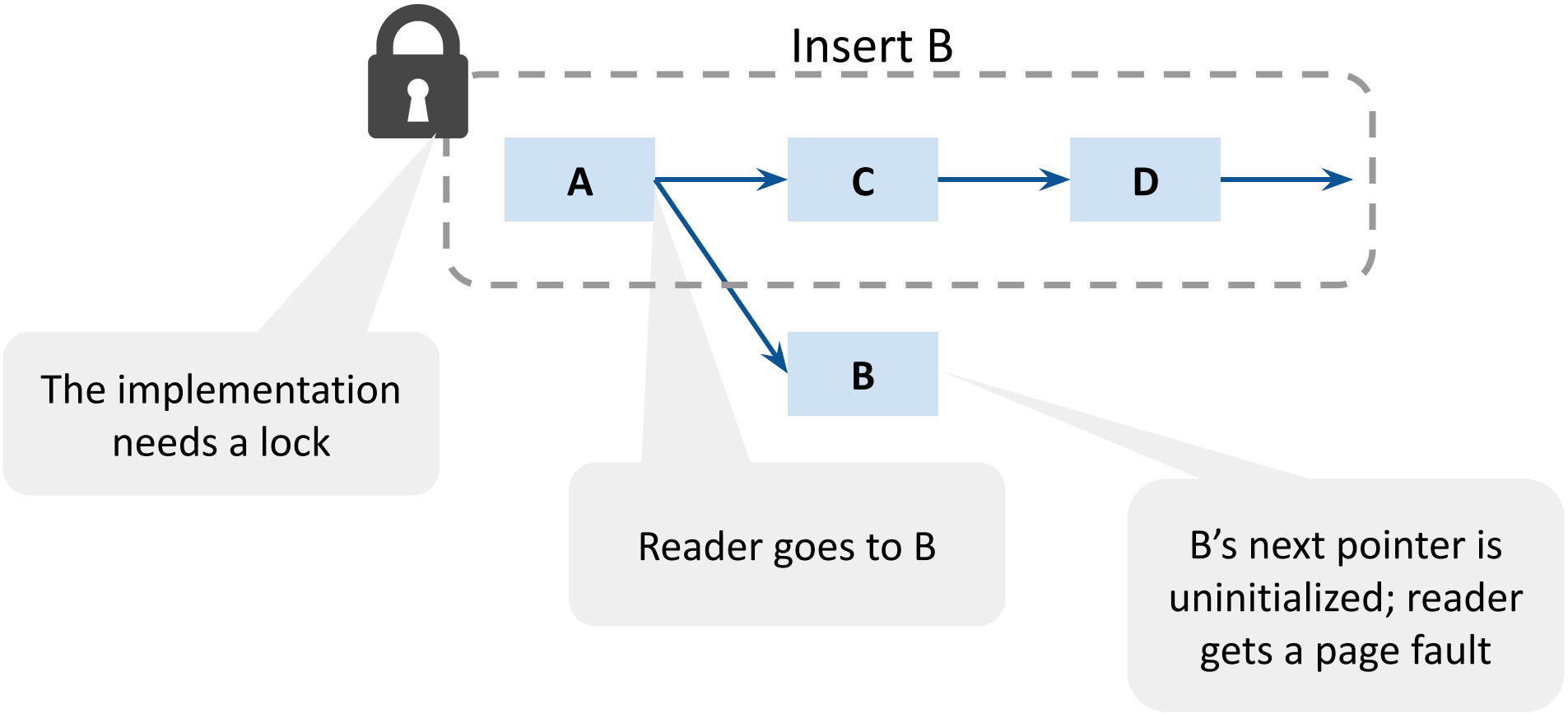
An alternative: Lock-free data structures

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

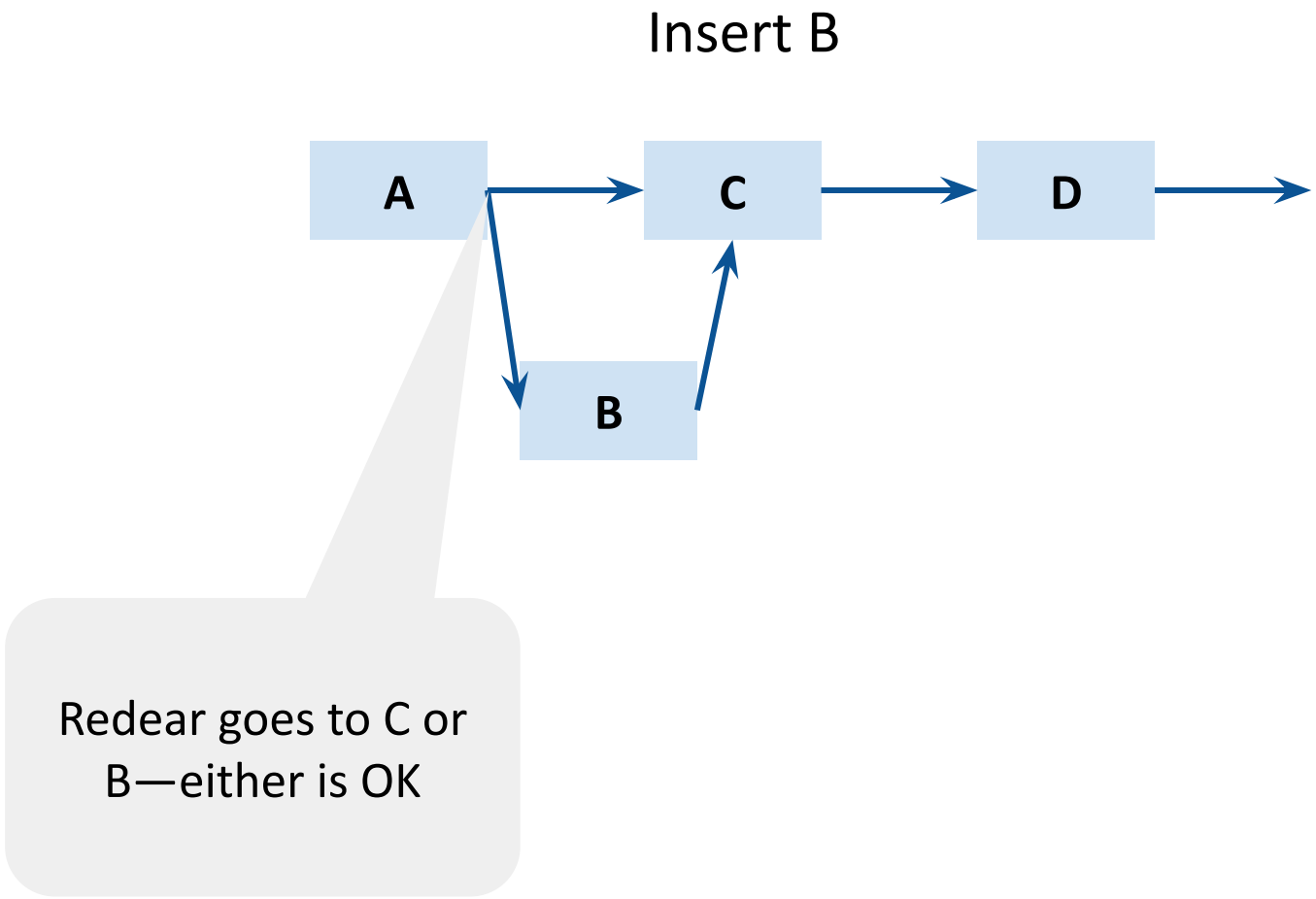
RCU: Split the difference

- One of the most difficult parts of lock-free algorithm is **concurrent changes to pointers**
  - So just use locks and make writers go one-at-a-time
- But, make writers wait be a bit careful so readers see a consistent view of the data structures
- If 99% of access are readers, avoid performance-killing read lock in the common case

Example: linked list



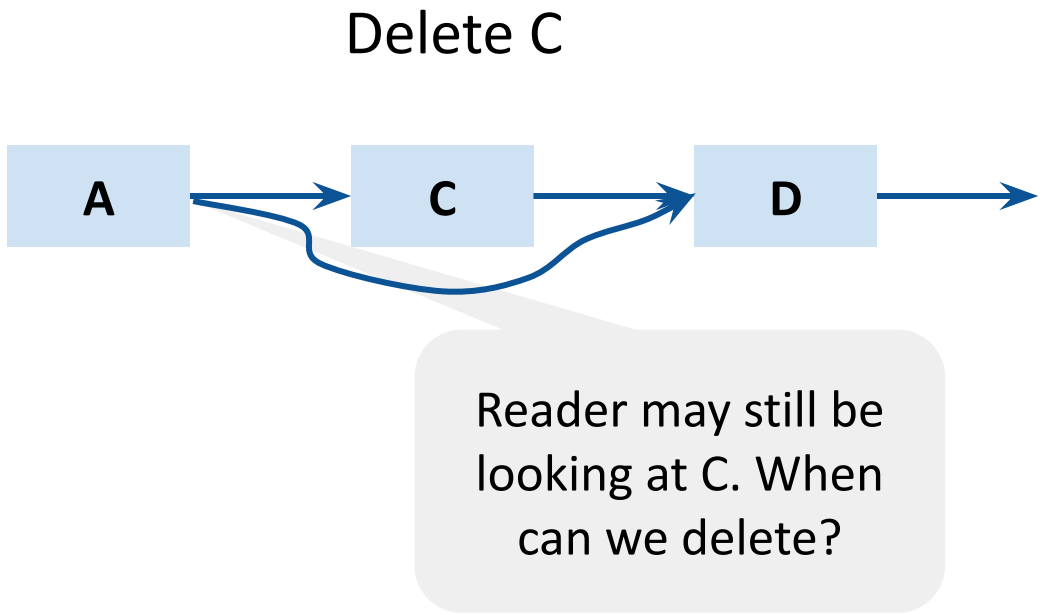
Example: linked list



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 or reader lock needed
  - Either traversal is safe
  - In some cases, we many need a memory barrier
- Key idea: carefully update the data structure so that a reader never follow a bad pointer
  - Writers still serialize using a lock

Example: linked list



Problem

- We logically remove a node by making it unreachable to future readers
  - No pointers to this node in the list
- We eventually need to free the node's memory
  - Leaks in kernel are bad!
- When is this safe?
  - Note that we have to wait for readers to "move on" down the list

Worst-case scenario

- Reader follows pointer to node X (about to be freed)
- Another thread frees X
- X is reallocated and overwritten with other data
- Reader interprets bytes in X->next as pointer, segmentation fault

Quiescence is the answer!

Trick: Linux does not allow a process to sleep while traversing an RCU-protected data structure

- Includes kernel preemption, IO 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 schedule, do the free

Note on RCU

- No doubly-linked list
- 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
  - Linked list are the workhorse of the Linux kernel
  - Improved performance

RCU big picture

- Carefully designed data structures
  - Readers always see consistent view

- Low-level “helper” functions encapsulate complex issues
  - Memory barriers
  - Quiescence

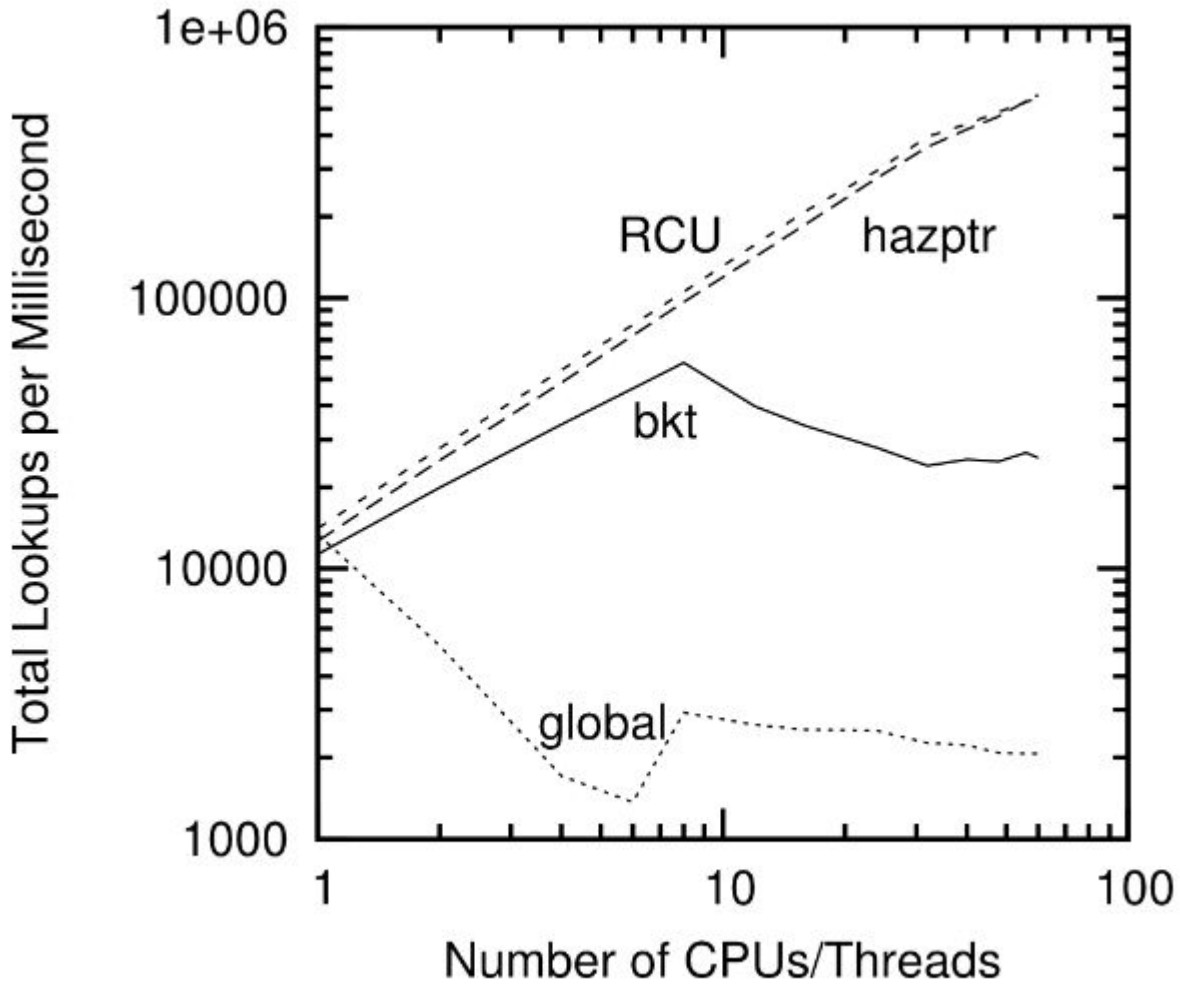
RCU API

- Drop in replacement for read_lock: **rcu_read_lock()**

- Publishing of new data: **rcu_assign_pointer()**
- Subscribing to the current version of data: **rcu_dereference()**
  - Wrappers such as **rcu_assign_pointer()** and **rcu_dereference()** include memory barriers

- Rather than immediately freeing an object, use **call_rcu(object, delete_func)** to do a deferred deletion

RCU performance



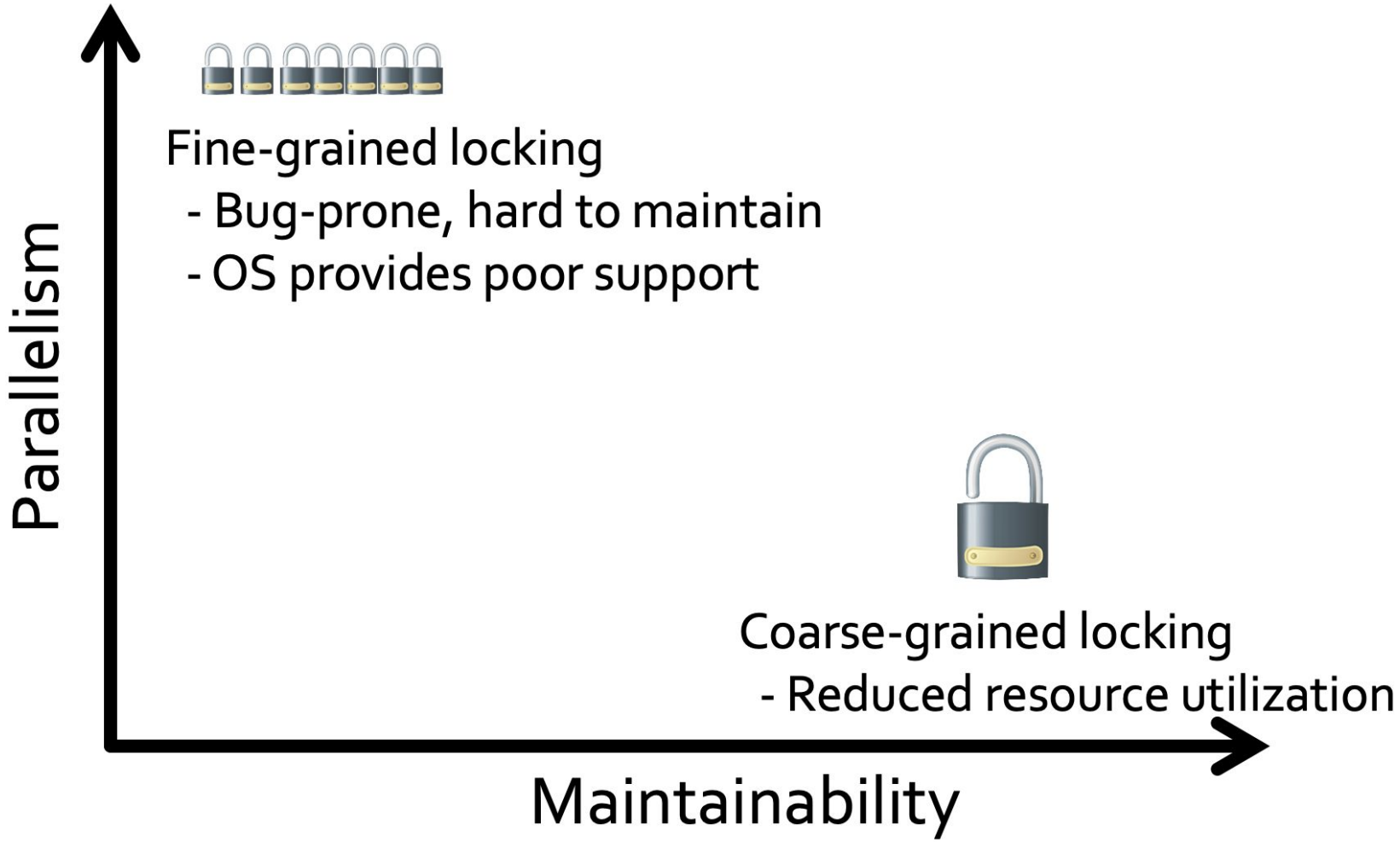
RCU area of applicability

https://lwn.net/Articles/262464/

This week

- Concurrency primitives recap
- Advanced concurrency framework: Read-copy-update (RCU)
- **Operating System Transactions**

Poor OS support for OS concurrency



Poor OS support for OS concurrency

- OS is weak link in concurrent programming model
- Can’t make consistent updates to system resources across multiple system calls
  - Race conditions for resources such as the file system
  - No simple work-around
- Applications can’t express consistency requirements
- OS can’t infer requirements

System transactions

- System transactions ensure consistent updates by concurrent applications
  - Prototype called TxOS
- Solve problems
  - System level race conditions (TOCTTOU)
- Build better applications
  - LDAP directory server
  - Software installation

System-level races

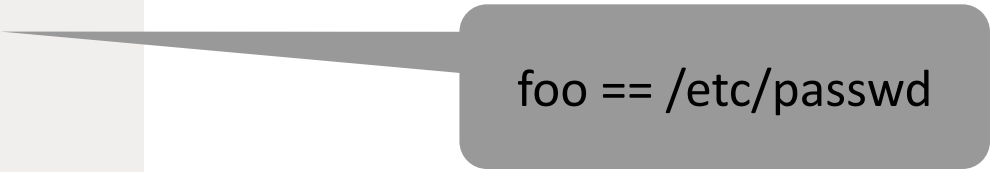
```
if (access(“foo”)) {

    fd = open(“foo”);
    write(fd, …);
}
```

Time-of-check-to-time-of-use (TOCTTOU) race condition

System-level races

```
if (access(“foo”)) {
    symlink(“/etc/passwd”, “foo”);
    fd = open(“foo”);
    write(fd, …);
}
```



Time-of-check-to-time-of-use (TOCTTOU) race condition

Eliminating TOCTTOU race

```
sys_xbegin();
if (access(“foo”)) {

    fd = open(“foo”);
    write(fd, …);
}
sys_xend();
```

Time-of-check-to-time-of-use (TOCTTOU) race condition

Eliminating TOCTTOU race

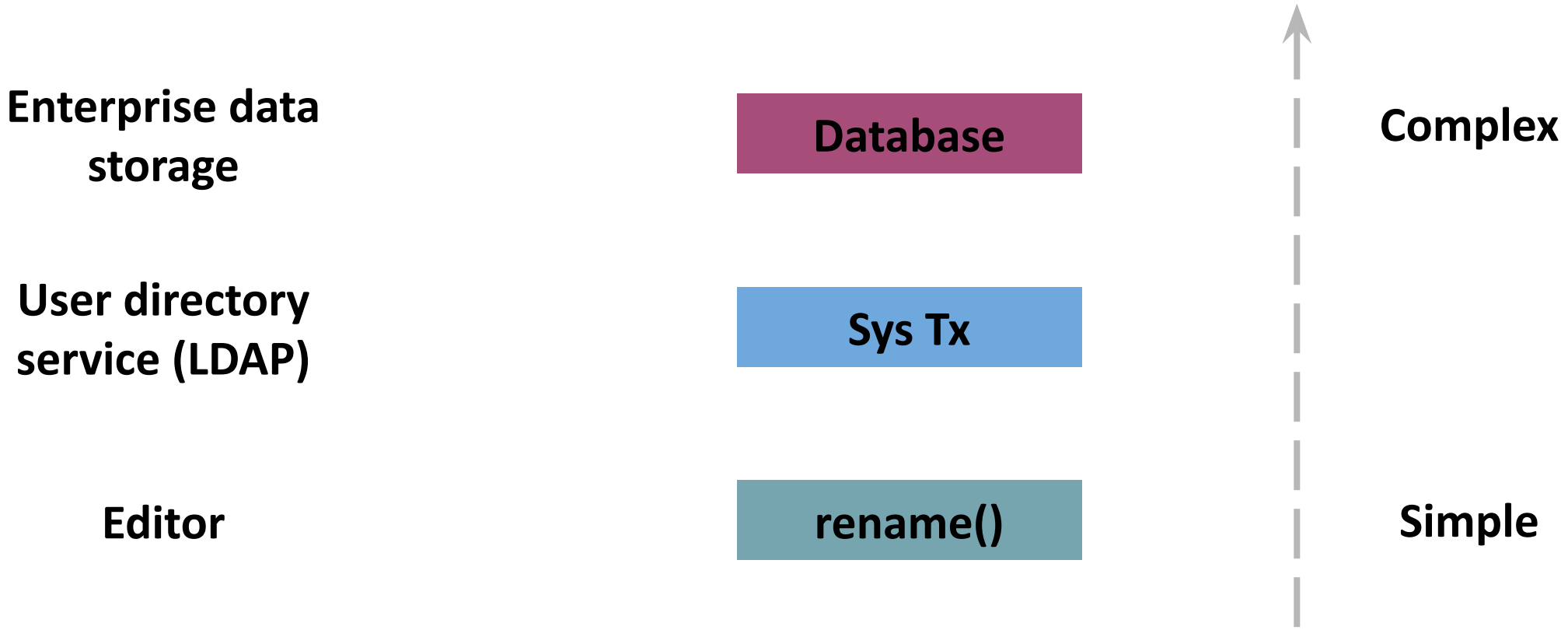
```
sys_xbegin();
if (access(“foo”)) {

    fd = open(“foo”);
    write(fd, …);
}
sys_xend();
```

Time-of-check-to-time-of-use (TOCTTOU) race condition

Example: Better application design

- How to make consistent updates to stable storage?



Example: transactional software install

```
sys_xbegin();
apt-get upgrade
sys_xend();
```

- A failed install is automatically rolled back
  - Concurrent, unrelated operations are unaffected
- System crash: reboot to entire upgrade or none

System transactions

- Simple API: sys_xbegin(), sys_xend(), sys_xabort()
- Transactions wraps group of system calls
  - Results isolated from other threads until commit
- Conflicting transactions must serialize for safety
  - Conflict must often read and write of same data
  - Too much serialization hurts performance

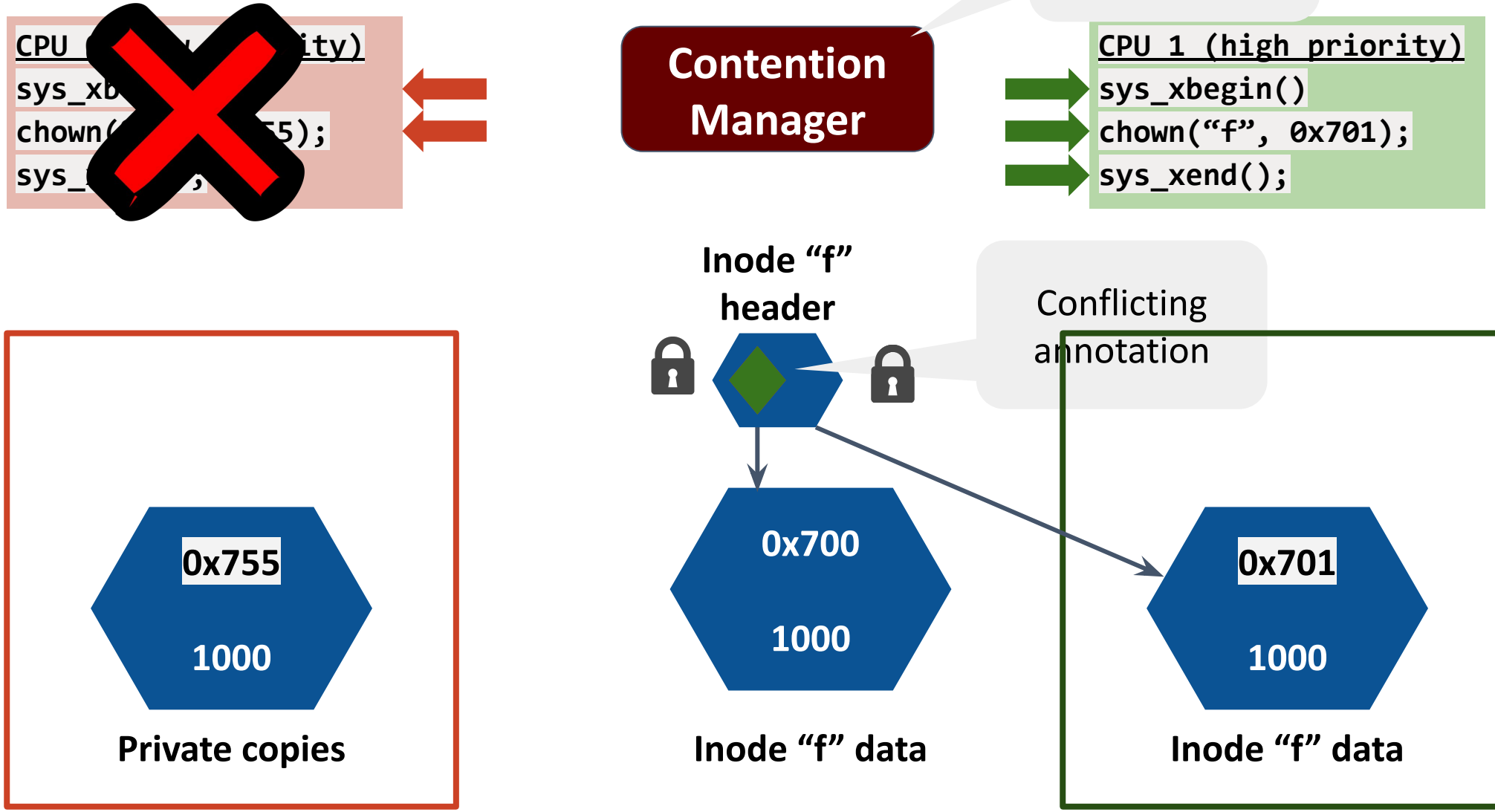
System transactions

- Provide ACID semantics:
  - Atomicity (A): all or nothing
  - Consistency (C): one consistent state to another
  - Isolated (I): updates as if only one concurrent transaction
  - Durable (D): committed transactions on disk

Building a transactional system

- Version management
  - Private copies instead of undo log
- Detect conflicts
  - Minimize performance impact of true conflicts
  - Eliminate false conflicts
- Resolve conflicts
  - Non-transactional code must respect transactional code

TxOS in action



System comparison

	Previous systems	TxOS
Speculative write location	Shared data structures	
Isolation mechanism	Two-phase locking	
Rollback mechanism	Undo log	
Commit mechanism	Discard undo log, release locks	

Deadlock prone

Can cause priority inversion

System comparison

	Previous systems	TxOS
Speculative write location	Shared data structures	Private copies of data structures
Isolation mechanism	Two-phase locking	Private copies + annotations
Rollback mechanism	Undo log	Discard private copies
Commit mechanism	Discard undo log, release locks	Publish private copies by pointer swap

Minimizing false conflicts

	Read	Write
Read	✓	✗
Write	✗	✗

```
sys_xbegin();
create(“/tmp/foo”);
sys_xend();
```

```
sys_xbegin();
create(“/tmp/bar”);
sys_xend();
```

Minimizing false conflicts

	Read	Add/del
Read	✓	✗
Add/del	✗	✗

OK if different files created, and directory is not being read

```
sys_xbegin();
create("/tmp/foo");
sys_xend();
```

```
sys_xbegin();
create("/tmp/bar");
sys_xend();
```

Minimizing false conflicts

	Read	Add/del
Read	✓	✗
Add/del	✗	✗

OK if different files created, and directory is not being read

- Insight: Object semantics allow more permissive conflict definition and therefore more concurrency
- TxOS supports precise conflict definitions per object type

Minimizing false conflicts

	Read	Add/del	Add/del + Read
Read	✔	✘	✘
Add/del	✘	✔	✘
Add/del + Read	✘	✘	✘

- Insight: Object semantics allow more permissive conflict definition and therefore more concurrency
- TxOS supports precise conflict definitions per object type
- Increases concurrency without relaxing isolation

Serializing txns and non-txns (strong iso.)

- TxOS mixess transactional and non-transactional code
  - In databases, everything is transaction
  - Semantically murky in historical systems
- Critical to correctness
  - Allows incremental adoption of transactions
  - TOCTTOU attacker will not use a transaction
- Problem: Can’t roll-back non-transactional system calls
  - Always aborting transaction undermines fairness

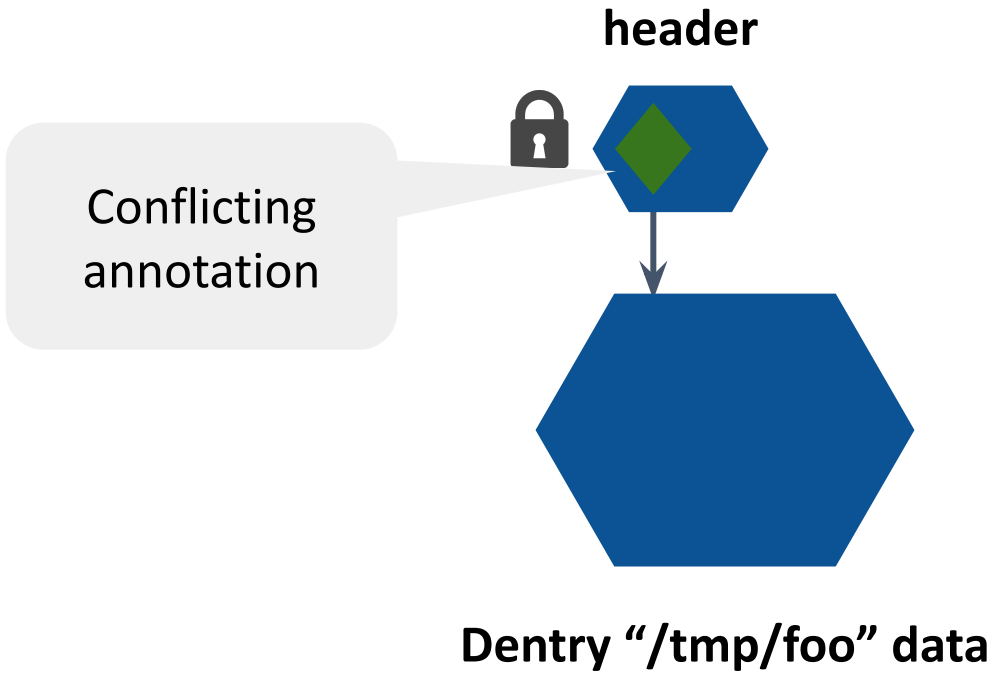
Strong isolation example in TxOS

CPU 0

```
symlink("/etc/passwd",
"/tmp/foo");
```

CPU 1

```
sys_xbegin()
if (access("/tmp/foo"))
    open("/tmp/foo"));
sys_xend();
```



Contention Manager

Options:

- **Abort CPU1**
- **Deschedule CPU0**

Transactions for application state

- System transactions only manage system state
- Applications can select their approach
  - Copy-on-write paging
  - Hardware or software transactional memory (TM)
  - Application-specific compensation code

Transactions: a core OS abstraction

- Easy to make kernel subsystems transactional
- Transactional file system in TxOS
  - Transactional implemented in VFS or higher
  - FS responsible for atomic updates to stable store
- Journal + TxOS = Transactional file system
  - 1 developer-month transactional ext3 prototype

TxOS prototype

- Extended Linux 2.6.22 to support system transactions
- Runs on commodity hardware
  - Added 8,600 LoC to Linux
  - Minor modification to 14,000 LoC
- Transactional semantics for a range of resources
  - File system, signals, processes, pipes

Transactional software install

```
sys_xbegin();
dpkg -i openssh;
sys_xend();
```

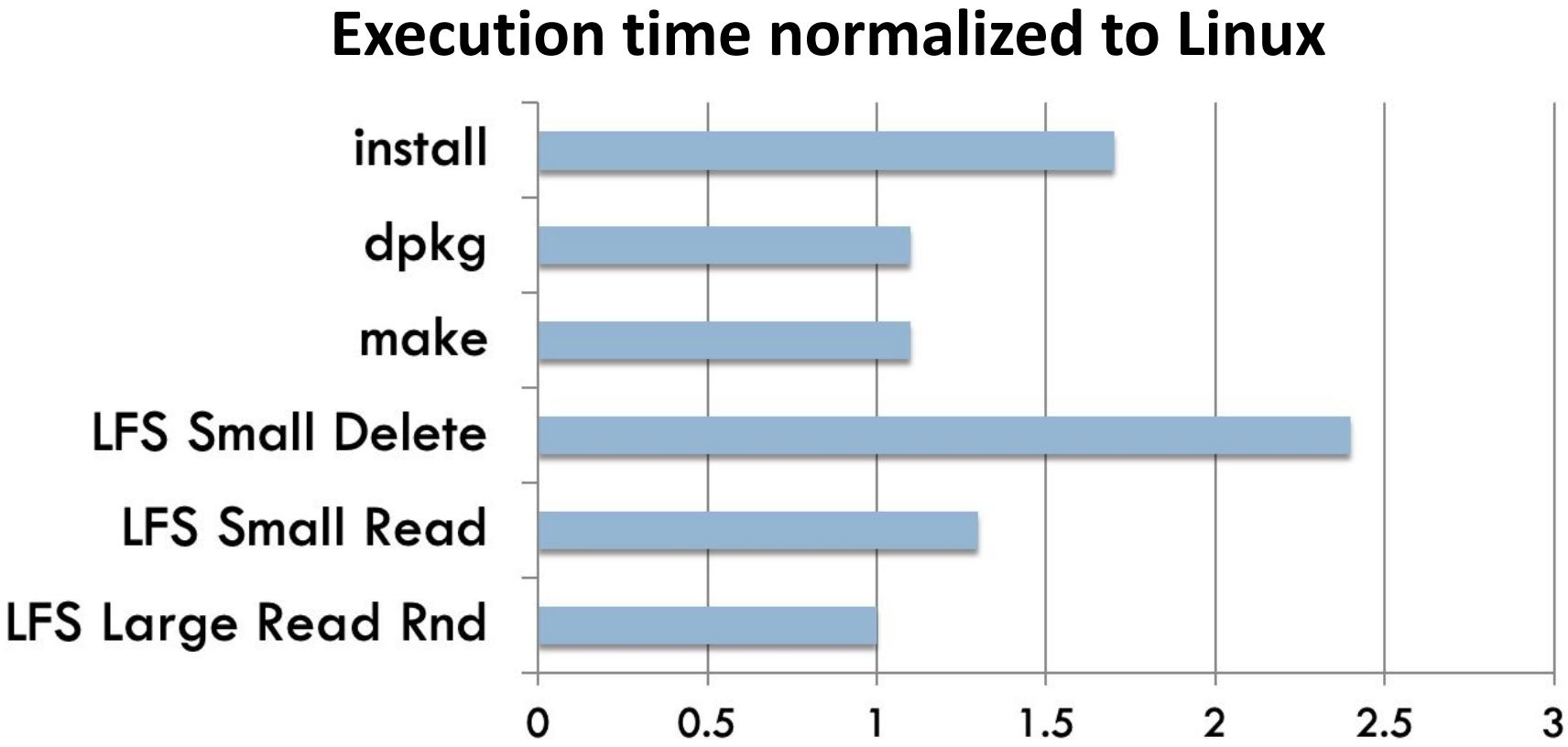
10% overhead

```
sys_xbegin();
install svn;
sys_xend();
```

70% overhead

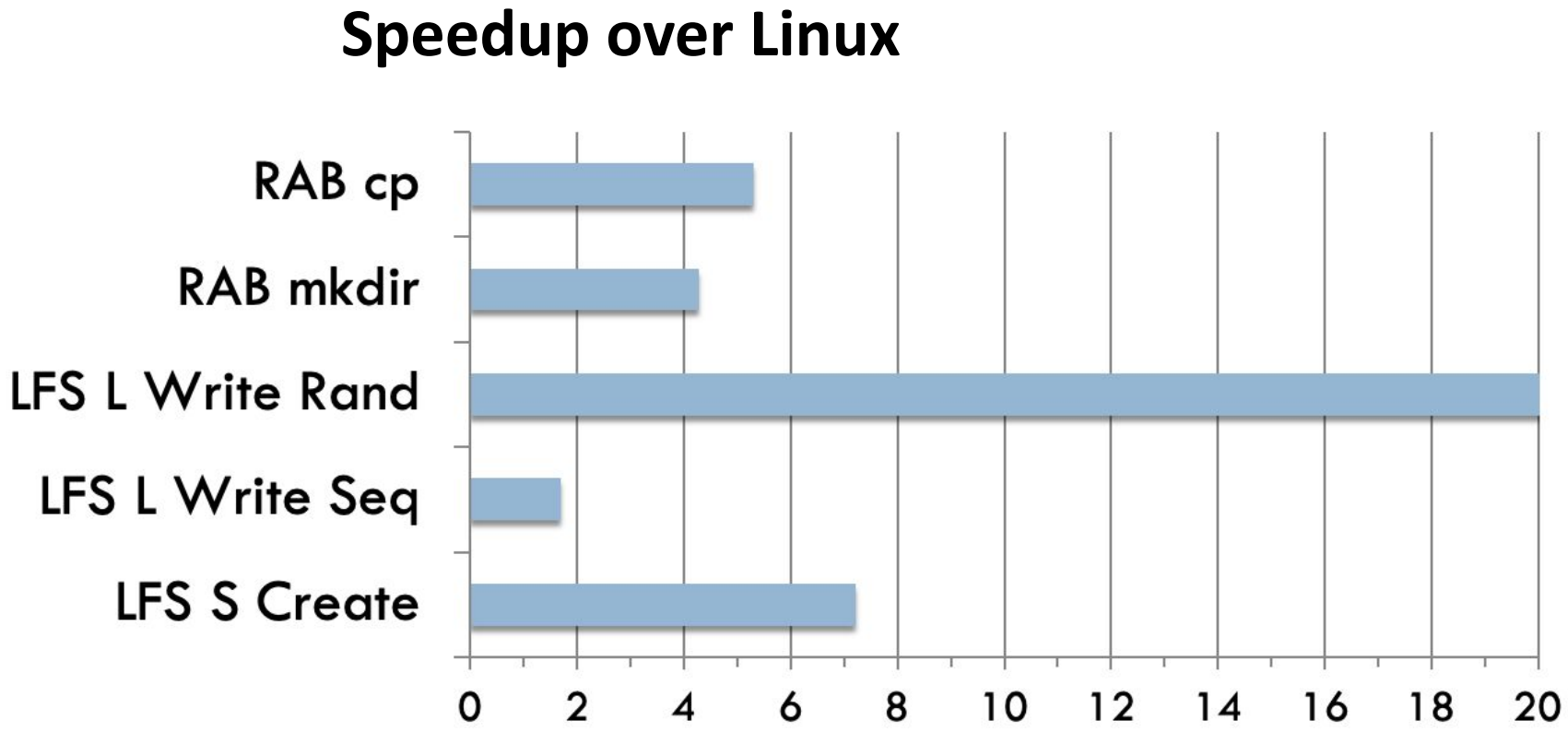
- A failed install is automatically rolled back
  - Concurrent, unrelated operations are unaffected
- System crash: reboot to entire upgrade or none

Transaction overheads



- Memory overhead on LFS large:
  - 13% high, 5% low (kernel)

Write speedups



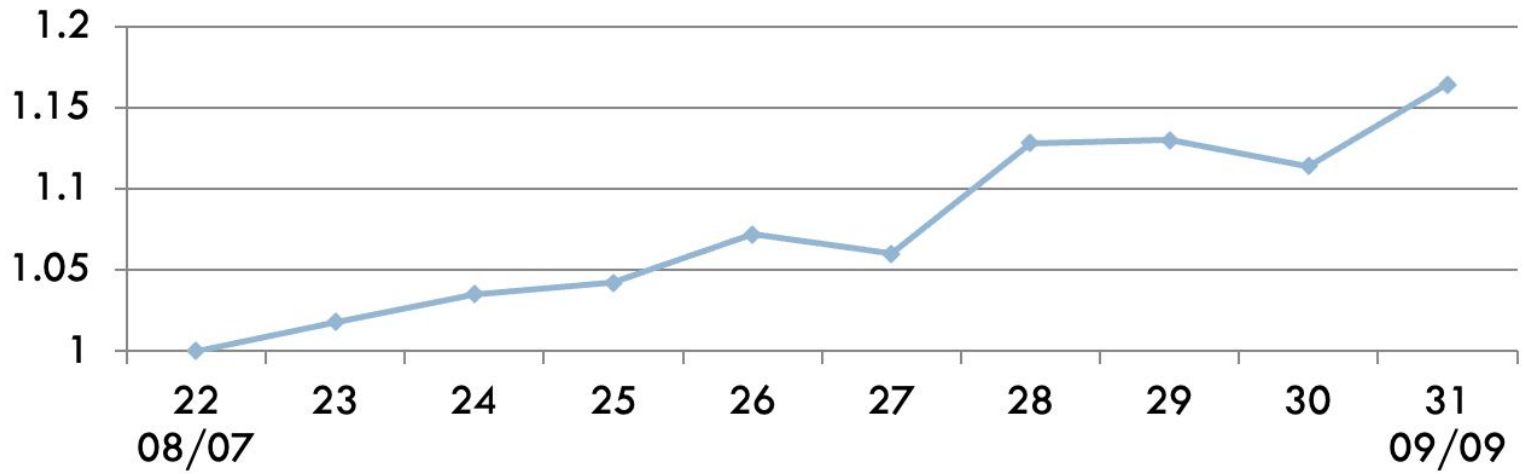
- Better IO scheduling – not luck
- Tx boundaries provide IO scheduling hints to the OS

Non-transactional overheads

- Non-transactional Linux compile: <2% on TxOS
  - Transactions are “pay-to-play”
- Single system call: 42% geometric mean
  - With additional optimization: 14% geomean
  - Optimizations approximated by eliding checks

What is practical?

Mean Linux syscall overhead, normalized to 2.6.22



- Feature creep over 2 years costs 16%
- Developers are willing to give up performance for useful features
- Transactions are in the same range (14%), more powerful

Summary

- RCU designed for handling read-mostly workloads
- RCU follows a publish-subscribe model with only single pointer update possible

- Transactions solve long-standing problems (TOCTTOU)
  - Replace ad-hoc solutions
- Transactions enable better concurrent programs