CSE 506: Operating Systems

Threading

Threading Review

- Multiple threads of execution in one address space
- x86 hardware:
  – One CR3 register and set of page tables
    - Shared by 2+ different contexts (each has RIP, RSP, etc.)
- Linux:
  – One mm_struct shared by several task_structs

Threading Libraries

- Kernel provides basic functionality
  – Ex.: create new thread
- Threading library (e.g., libpthread) provides nice API
  – Thread management (join, cleanup, etc.)
  – Synchronization (mutex, condition variables, etc.)
  – Thread-local storage
- Part of design is division of labor
  – Between kernel and library

User vs. Kernel Threading

- Kernel threading
  - Every application-level thread is kernel-visible
    - Has its own `task struct`
  - Called ***1:1***
- User threading
  - Multiple application-level threads (m)
    - Multiplexed on n kernel-visible threads (m > n)
  - Context switching can be done in user space
    - Just a matter of saving/restoring all registers (including RSP!)
  - Called ***m:n***
    - Special case: ***m:1*** (no kernel support)

Tradeoffs of Threading Approaches

- Context switching overheads
- Finer-grained scheduling control
- Blocking I/O
- Multi-core

Context Switching Overheads

- Forking a thread halves your time slice
  - Takes a few hundred cycles to get in/out of kernel
    - Plus cost of switching a thread
  - Time in the scheduler counts against your timeslice
- 2 threads, 1 CPU
  - Run the context switch code in user space
    - Avoids trap overheads, etc.
    - Get more time from the kernel

Finer-Grained Scheduling Control

- Thread 1 has lock, Thread 2 waiting for lock
  - Thread 1’s quantum expired
  - Thread 2 spinning until its quantum expires
  - Can donate Thread 2’s quantum to Thread 1?
    - Both threads will make faster progress!
- Many examples (producer/consumer, barriers, etc.)
- Deeper problem:
  - Application’s data and synchronization unknown to kernel

Blocking I/O

- I/O requires going to the kernel
- When one user thread does I/O
  - All other user threads in same kernel thread wait
  - Solvable with async I/O
    - Much more complicated to program

Multi-core

- Kernel can schedule threads on different cores
  - Higher performance through parallelism
- User-level threads unknown to kernel
  - Restricted to switching within one core
  - m:n libraries can help here
    - User code can expect kernel threads to run on different cores
    - Make things a lot more complicated

User-level threading

- User scheduler creates:
  - Analog of task struct for each thread
    - Stores register state when preempted
  - Stack for each thread
  - Some sort of run queue
    - Simple list in the (optional) paper
    - Application free to use O(1), CFS, round-robin, etc.

User-threading in practice

- Has come in and out of vogue
  - Correlated to efficiency of OS thread create and switch
- Linux 2.4 – Threading was really slow
  - User-level thread packages were hot
    - Code is really complicated
      - Hard to maintain
      - Hard to tune
- Linux 2.6 – Substantial effort into tuning threads
  - Most JVMs abandoned user-threads
    - Tolerable performance at low complexity

CSE 506: Operating Systems

Kernel Synchronization

What is Synchronization?

- Code on multiple CPUs coordinate their operations
- Examples:
  - Locking provides mutual exclusion
    - CPU A locks CPU B’s run queue to steal tasks
      - Otherwise CPU B may start running a task that CPU A is stealing
  - Threads wait at barrier for completion of computation
  - Coordinating which CPU handles an interrupt

Lock Frequency

- Modern OS kernel is a complex parallel program
  - Arguably the most complex
    - Database community would likely be the only ones to argue
- Includes most common synchronization patterns
  - And a few interesting, uncommon ones

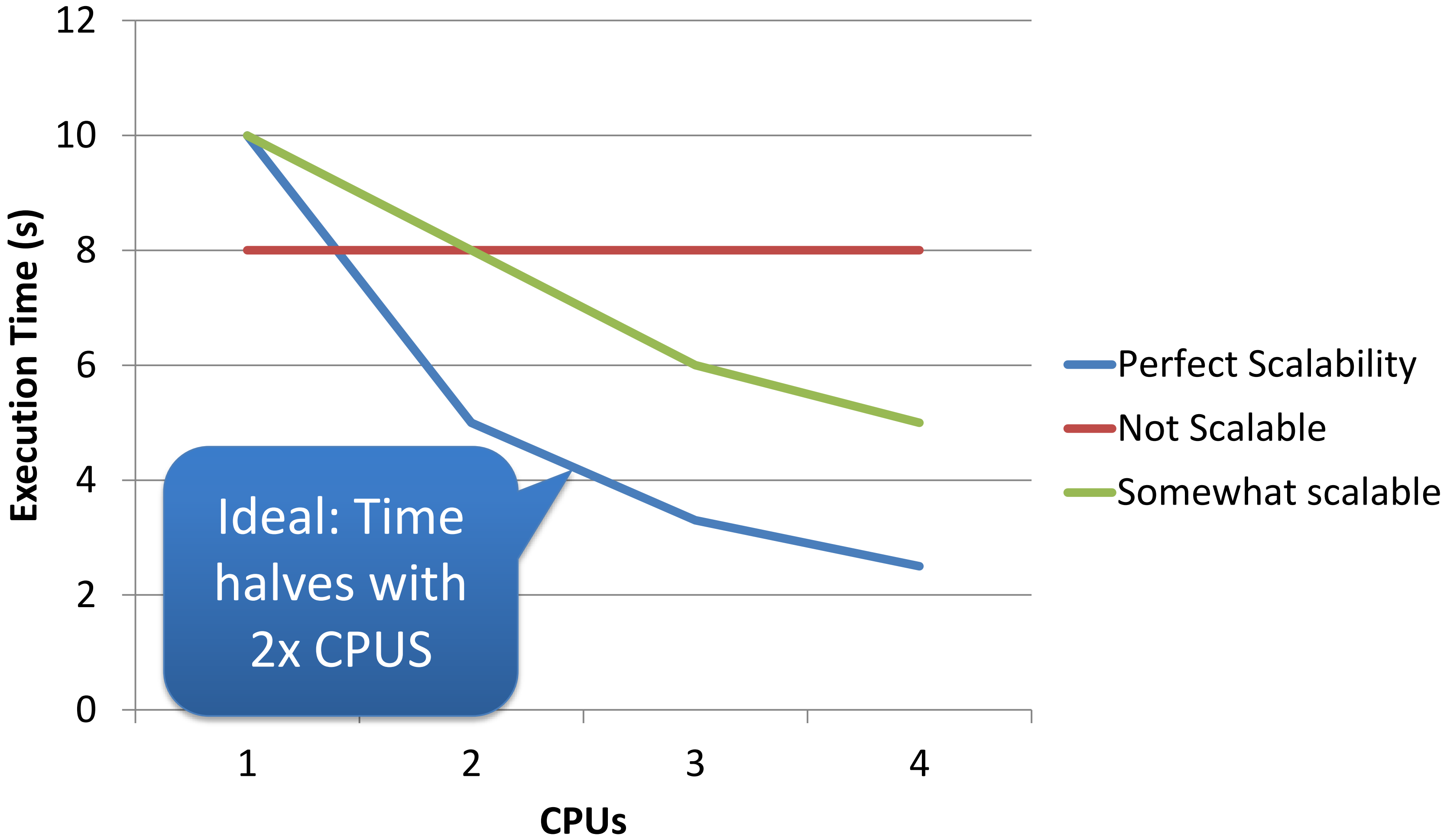
Kernel Locking History

- Traditionally, didn't worry about it
  - Most machines were single processor
- Eventually started supporting multi-processors
  - Called kernels "SMP" around this time
  - Typically had a few (one?) lock
    - Called "***Giant***" lock
- Giant lock became a bottleneck
  - Switched to fine-grained locking
    - With many different types of locks
- Grew tools to dynamically detect/fix locking bugs
  - E.g., FreeBSD "WITNESS" infrastructure

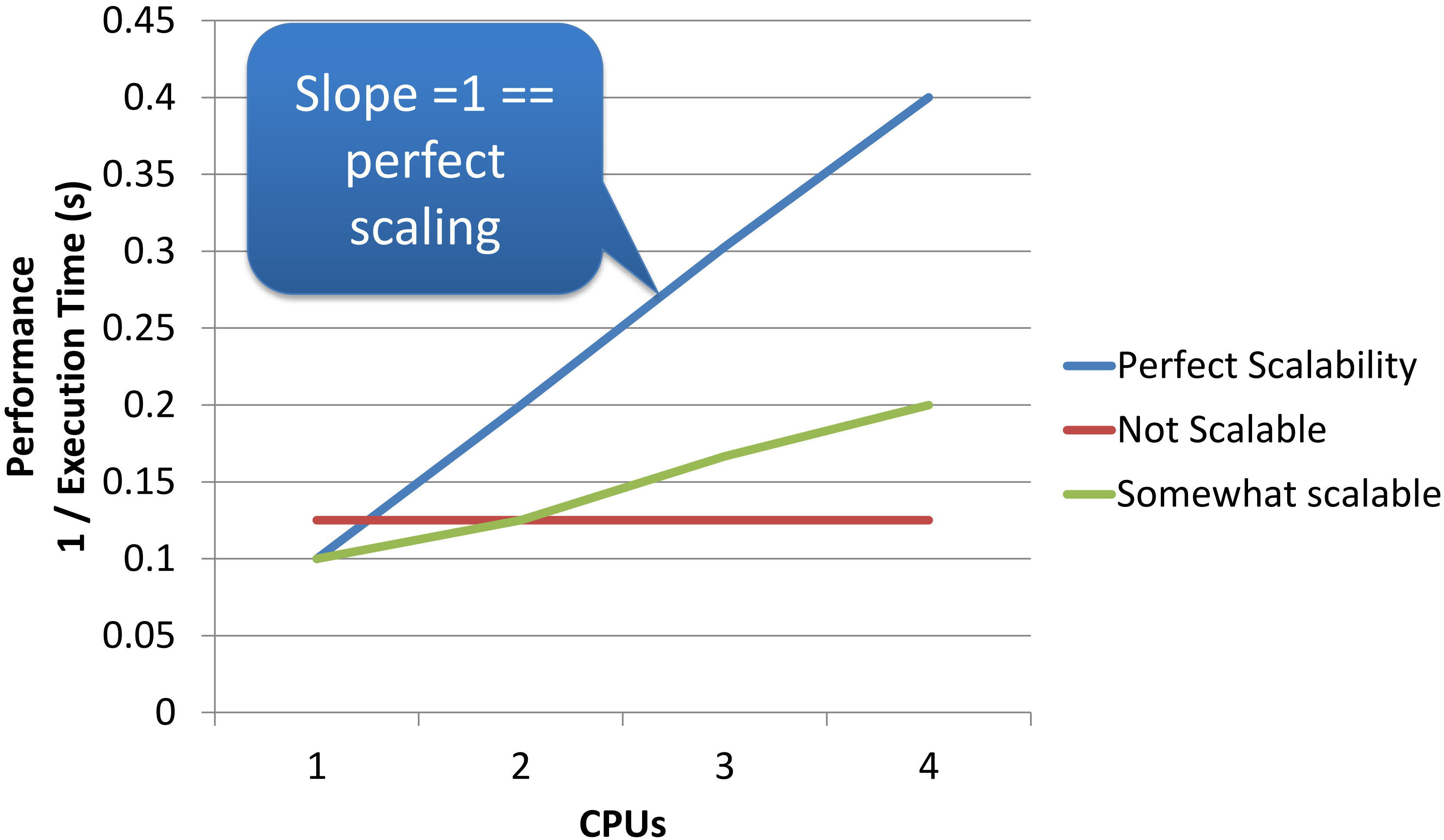
Performance Scalability

- How much performance do additional CPUs give?
  - None: extra CPU is wasted: No scalability
  - 2x CPUs doubles work done per time: Perfect scalability
- Most software isn't scalable
- Most scalable software isn't perfectly scalable
- Hardware matters for scalability
  - When OS people say "2x CPUs"
    - Did they add a chip to another socket with its own memory?
    - Did they double cores that shares cache with other cores?
    - Did they enable hyper threads in all cores?

Performance Scalability (Time)



Performance Scalability (Throughput)



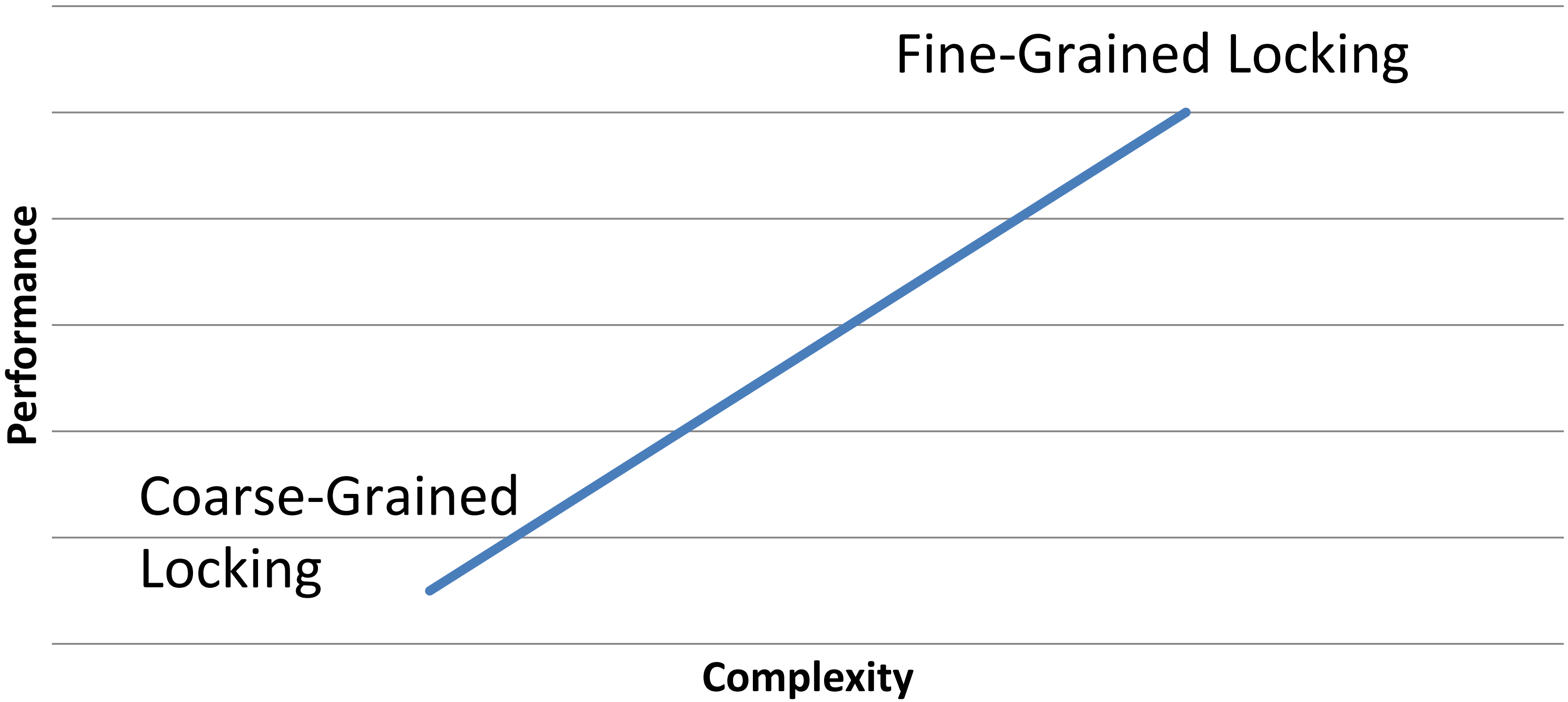
Coarse-Grained Locking

- A single lock for everything
  - Idea: Before touching any shared data, grab the lock
  - Problem: completely unrelated operations ***serialized***
    - Adding CPUs doesn’t improve performance

Fine-Grained Locking

- Many “little” locks for individual data structures
  - Goal: Unrelated activities hold different locks
    - Hence, adding CPUs improves performance
  - Cost: complexity of coordinating locks

Current Reality



- Unsavory trade-off between complexity & scalability

How Do Locks Work?

- Locks are addresses in ***shared memory***
  - To check if locked, read value from location
  - To unlock, write value to location to indicate unlocked
  - To lock, write value to location to indicate locked
    - If already locked, keep reading value until unlock observed
- Use hardware-provided ***atomic instruction***
  - Determines who wins under contention
  - Requires waiting strategy for the loser(s)

Atomic Instructions

- Regular memory accesses don't work

```
lock: movq [lock],%rax
cmpq %rax,1
je lock                 ;# "spin" lock
movq 1,[lock]
```

Other CPU may "movq 1,[lock]" at same time

- ***Atomic Instructions*** guarantee atomicity
  - Perform ***Read, Modify, and Write*** together (RMW)
  - Many flavors in the real world (***lock*** prefix on x86)
    - ***Compare and Swap*** (CAS)
    - ***Fetch and Add***
    - ***Test and Set***
    - ***Load Linked / Store Conditional***

Waiting Strategies

- Spinning
  - Poll lock in a busy loop
  - When lock is free, try to acquire it
- Blocking
  - Put process on wait queue and go to sleep
    - CPU may do useful work
  - Winner (lock holder) wakes up loser(s)
    - After releasing lock
  - Same thing as used to wait on I/O

Which strategy to use?

- Expected waiting time vs. time of 2 context switches
  - If lock will be held a long time, blocking makes sense
  - If the lock is only held momentarily, spinning makes sense
- Adaptive sometimes works
  - Try to spin a bit
    - If successful, great
    - If unsuccessful, block
  - Can backfire (if spin is never successful)

Reader/Writer Locks

- If everyone is reading, no need to block
  - Everyone reads at the same time
- Writers require mutual exclusion
  - For anyone to write, wait for all readers to give up lock

Linux RW-Spinlocks

- Low 24 bits count active readers
  - Unlocked: 0x01000000
  - To read lock: atomic_dec_unless(count, 0)
    - 1 reader: 0x:00ffffff
    - 2 readers: 0x00fffffe
    - Etc.
    - Readers limited to 2^24
- 25th bit for writer
  - Write lock – CAS 0x01000000 -> 0
    - Readers will fail to acquire the lock until we add 0x1000000

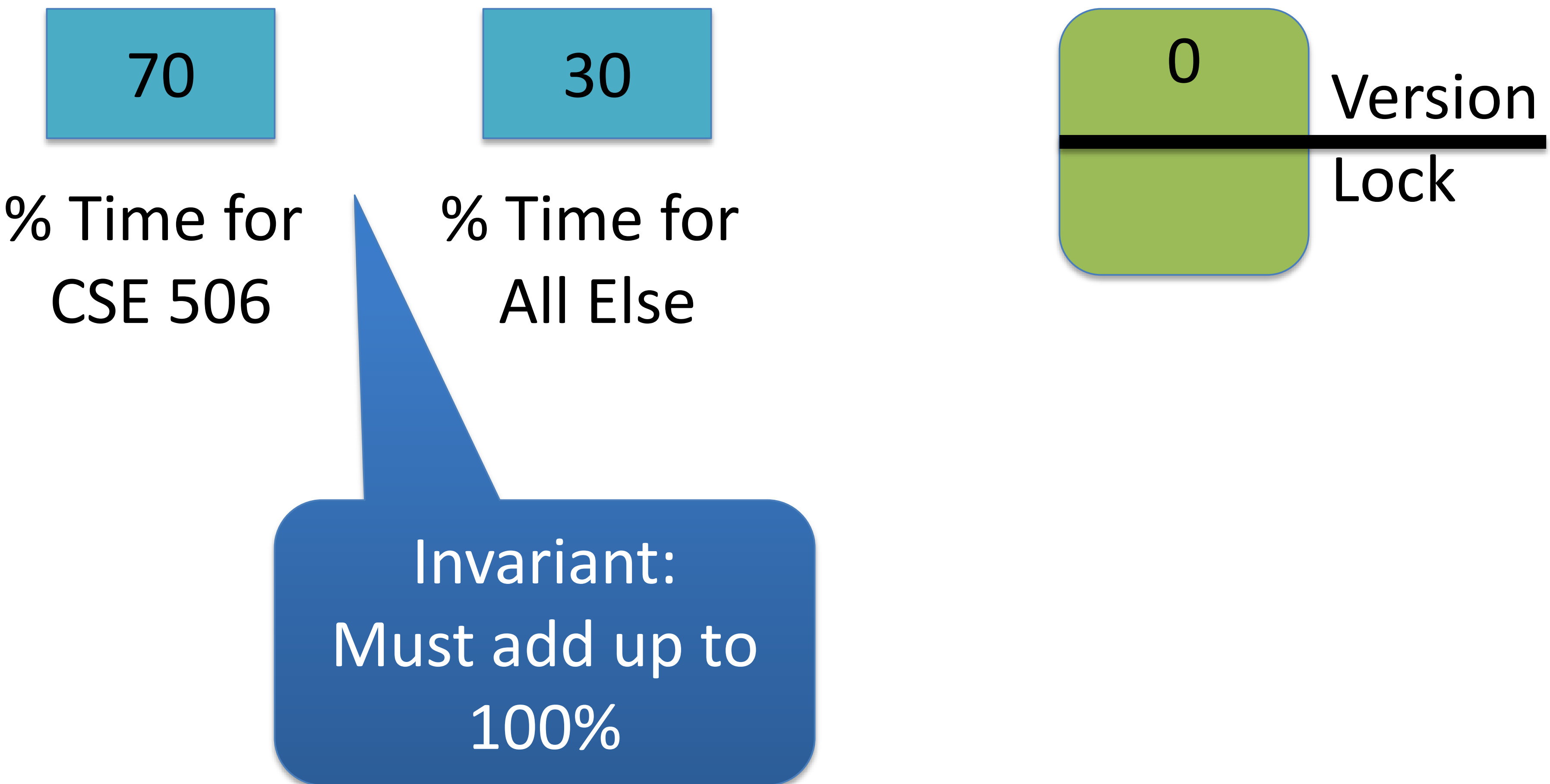
Readers Starving Writers

- Constant stream of readers starves writer
- We may want to prioritize writers over readers
  - For instance, when readers are polling for the write

Linux Seqlocks

- Explicitly favor writers, potentially starve readers
- Idea:
  - An explicit write lock (one writer at a time)
  - Plus a version number
    - Each writer increments at beginning ***and*** end of critical section
- Readers: Check version, read data, check again
  - If version changed, try again in a loop
  - If version hasn't changed and is even, data is safe to use

Seqlock Example



Seqlock Example

80	20
% Time for CSE 506	% Time for All Else

2

Version

Lock

What if reader executed now?

```
Reader:
do {
        v = version;
        a = cse506;
        b = other;
} while (v % 2 == 1 ||
              v != version);
```

```
Writer:
        lock();
        version++;
        other = 20;
        cse506 = 80;
        version++;
        unlock();
```

Lock Composition

- Need to touch two data structures (A and B)
  - Each is protected by its own lock
- What could go wrong?
  - Deadlock!
  - Thread 0: lock(a); lock(b)
  - Thread 1: lock(b); lock(a)
- How to solve?
  - Lock ordering

Lock Ordering

- A code convention
- Developers gather, eat lunch, plan order of locks
  - Potentially worse: gather, drink beer, plan order of locks
- Nothing prevents violating convention
  - Research topics on making this better:
    - Finding locking bugs
    - Automatically locking things properly
    - Transactional memory

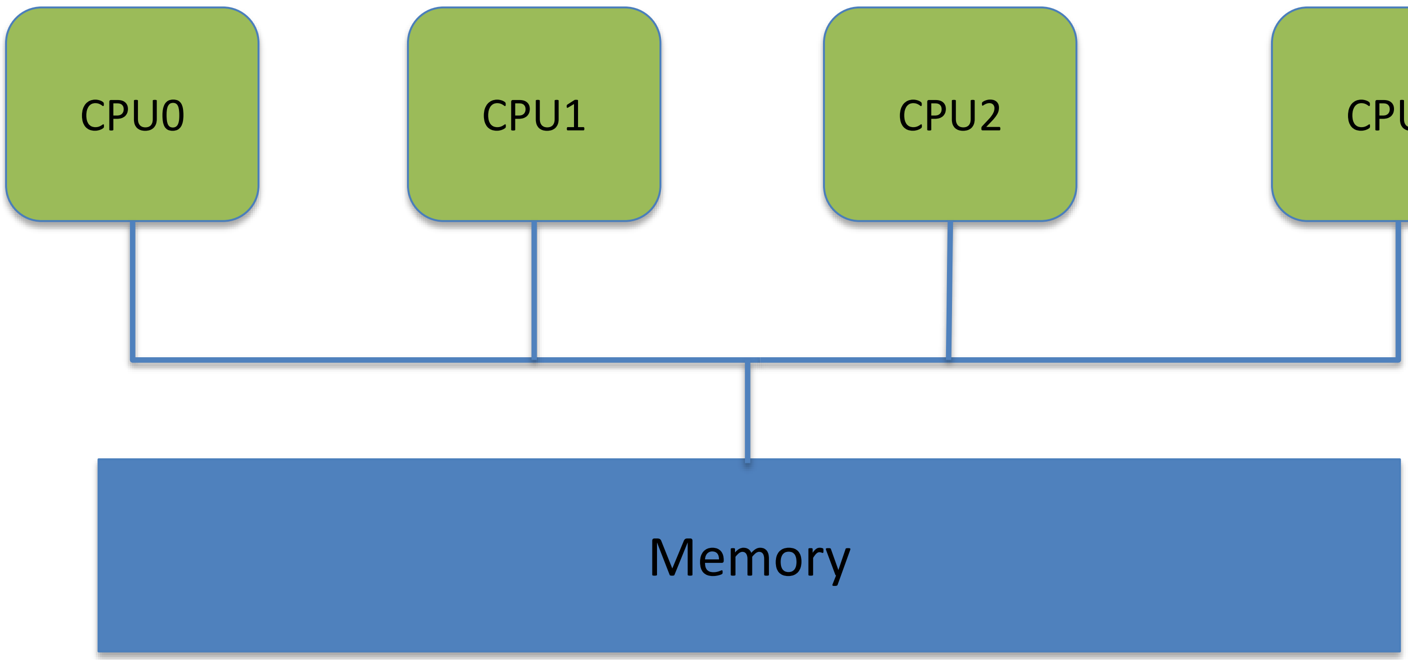
mm/filemap.c lock ordering

```
/*
 * Lock ordering:
 *  ->i_mmap_lock                (vmtruncate)
 *    ->private_lock             (__free_pte->__set_page_dirty_buffers)
 *      ->swap_lock              (exclusive_swap_page, others)
 *        ->mapping->tree_lock
 *  ->i_mutex
 *    ->i_mmap_lock              (truncate->unmap_mapping_range)
 *  ->mmap_sem
 *    ->i_mmap_lock
 *      ->page_table_lock or pte_lock   (various, mainly in memory.c)
 *        ->mapping->tree_lock   (arch-dependent flush_dcache_mmap_lock)
 *  ->mmap_sem
 *    ->lock_page                (access_process_vm)
 *  ->mmap_sem
 *    ->i_mutex                  (msync)
 *  ->i_mutex
 *    ->i_alloc_sem              (various)
 *  ->inode_lock
 *    ->sb_lock                  (fs/fs-writeback.c)
 *    ->mapping->tree_lock       (__sync_single_inode)
 *  ->i_mmap_lock
 *    ->anon_vma.lock            (vma_adjust)
 *  ->anon_vma.lock
 *    ->page_table_lock or pte_lock     (anon_vma_prepare and various)
 *  ->page_table_lock or pte_lock
 *    ->swap_lock                (try_to_unmap_one)
 *    ->private_lock             (try_to_unmap_one)
 *    ->tree_lock                (try_to_unmap_one)
 *    ->zone.lru_lock            (follow_page->mark_page_accessed)
 *    ->zone.lru_lock            (check_pte_range->isolate_lru_page)
 *    ->private_lock             (page_remove_rmap->set_page_dirty)
 *    ->tree_lock                (page_remove_rmap->set_page_dirty)
 *    ->inode_lock               (page_remove_rmap->set_page_dirty)
 *    ->inode_lock               (zap_pte_range->set_page_dirty)
 *    ->private_lock             (zap_pte_range->__set_page_dirty_buffers)
 *  ->task->proc_lock
 *    ->dcache_lock              (proc_pid_lookup)
 */
```

CSE 506: Operating Systems

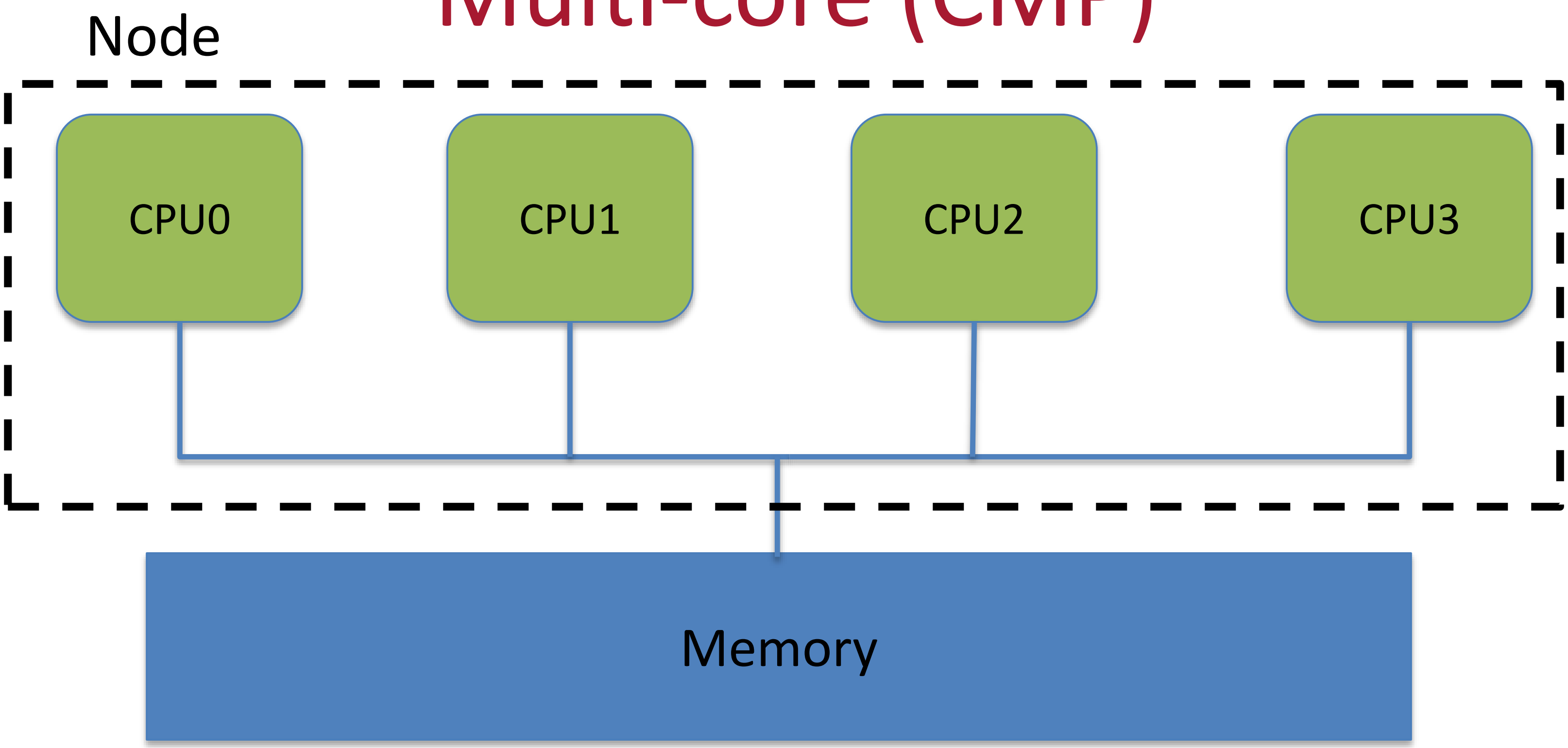
MP Scheduling

Symmetric Multi-Processing (SMP)



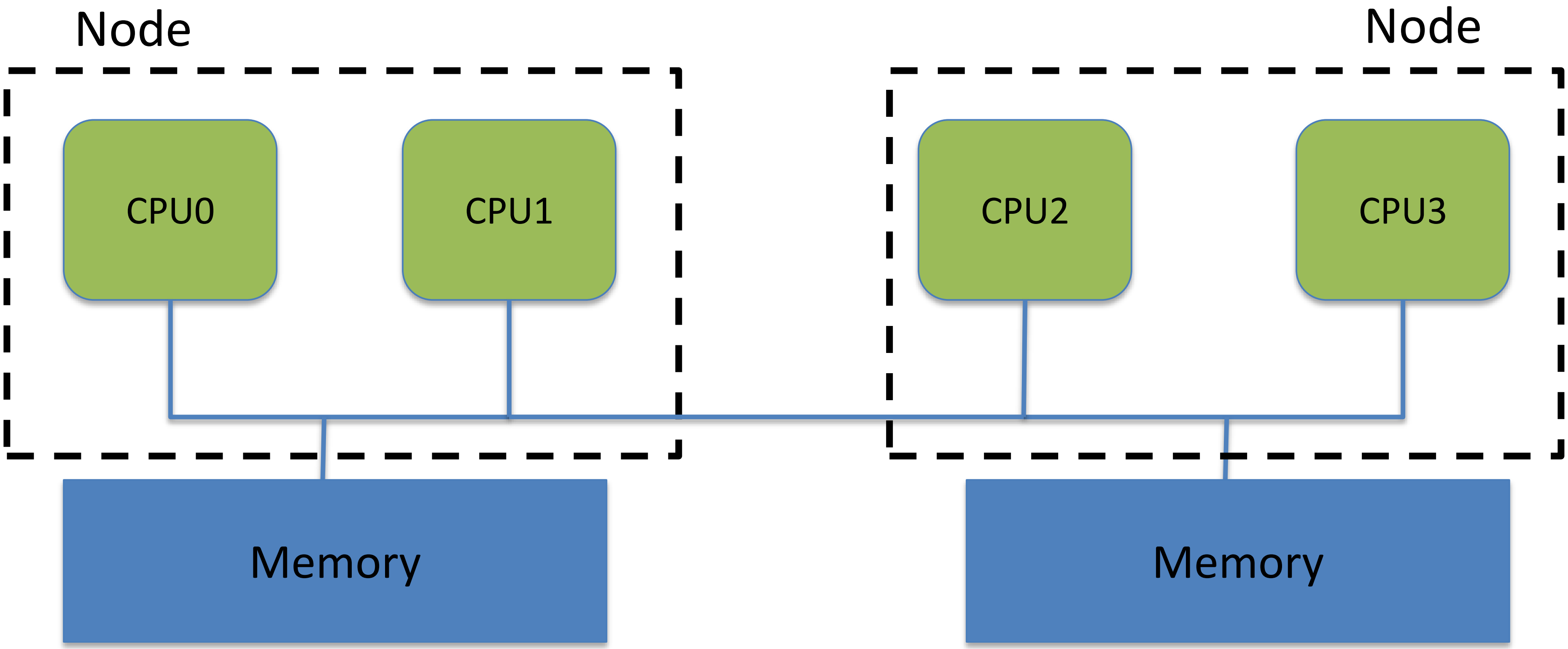
- All CPUs similar, equally “close” to memory
- Horribly abused name by software community
  - Use “SMP” for anything with more than 1 “context”

Multi-core (CMP)



- All CPUs inside a single chip

Non-Uniform Memory Access (NUMA)

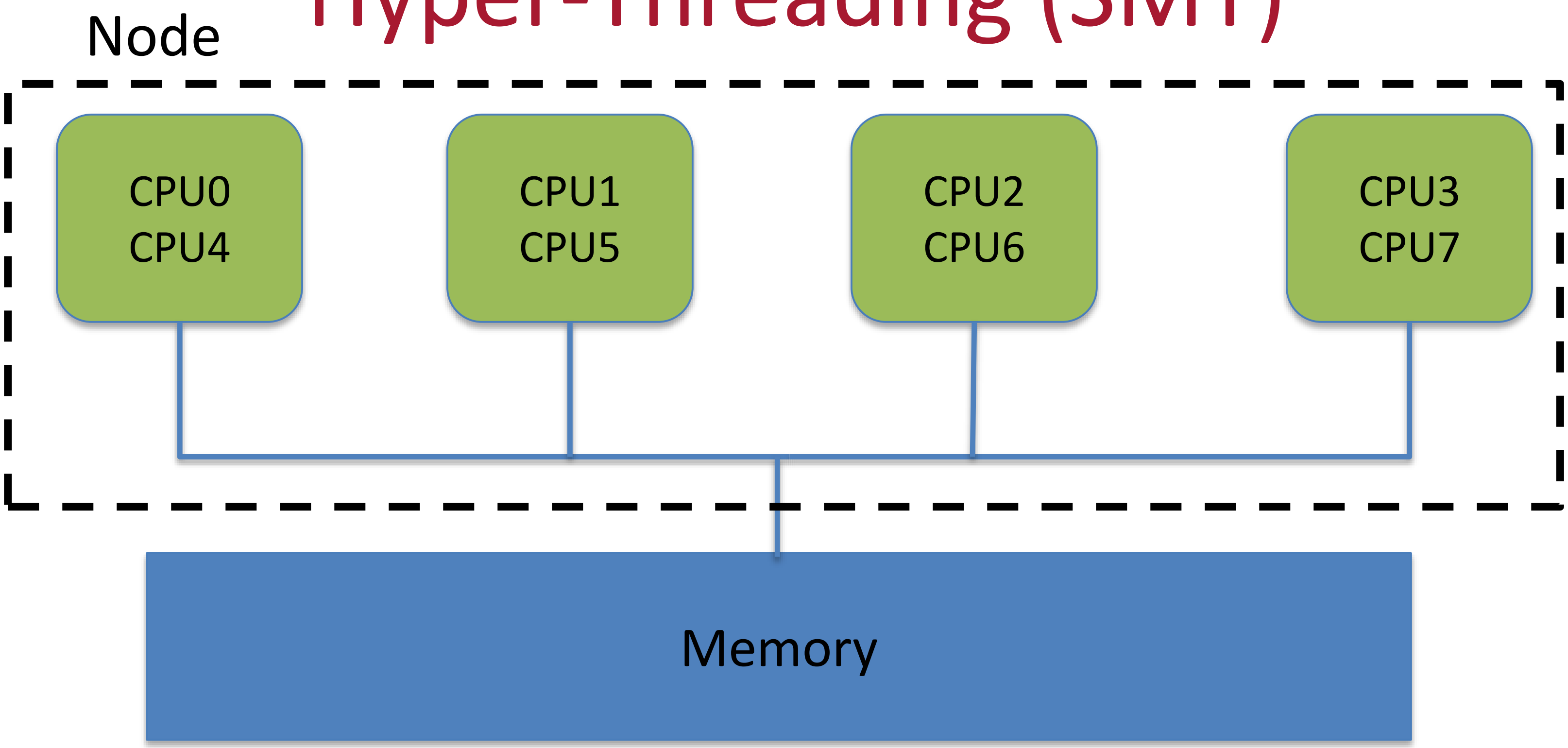


- Want to keep execution near memory
  - Accessing “remote” memory is more expensive

Hyper-Threading (SMT)

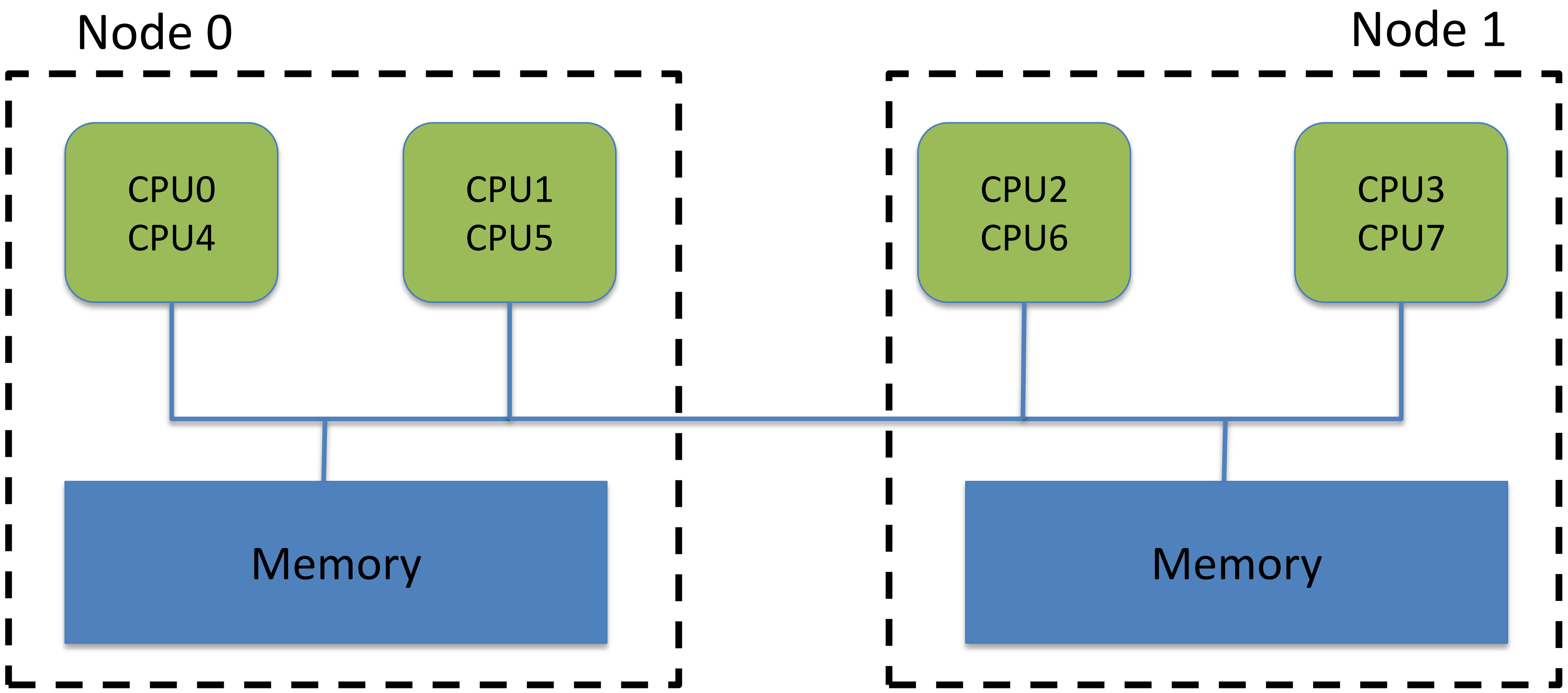
- One core, but multiple contexts
  - What's a context?
    - A set of register values (including ones like CR3)
- OS view: 2 logical CPUs
  - "CPU" is also horribly abused
    - Really should be "hardware context" or "hardware thread"
  - Does not duplicate execution resources
  - Programs on same core may interfere with each other
    - But both may run
      - 2x slow threads may be better than 1x fast one

Hyper-Threading (SMT)



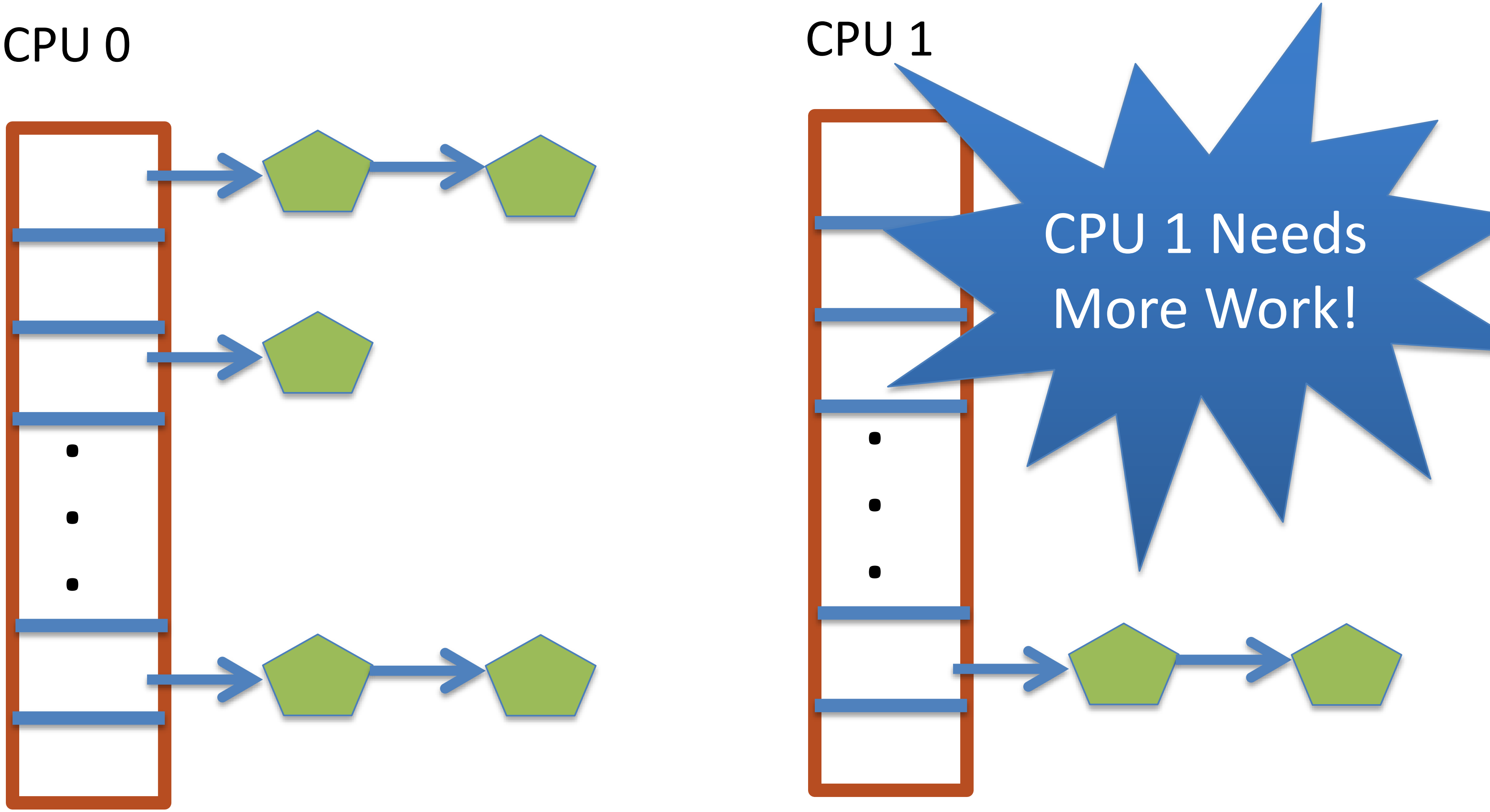
- All CPUs inside a single chip

All Kinds of Parallelism Together



- 2-socket NUMA, w/2 dual-threaded cores per socket

One set of Run Queues per “CPU”



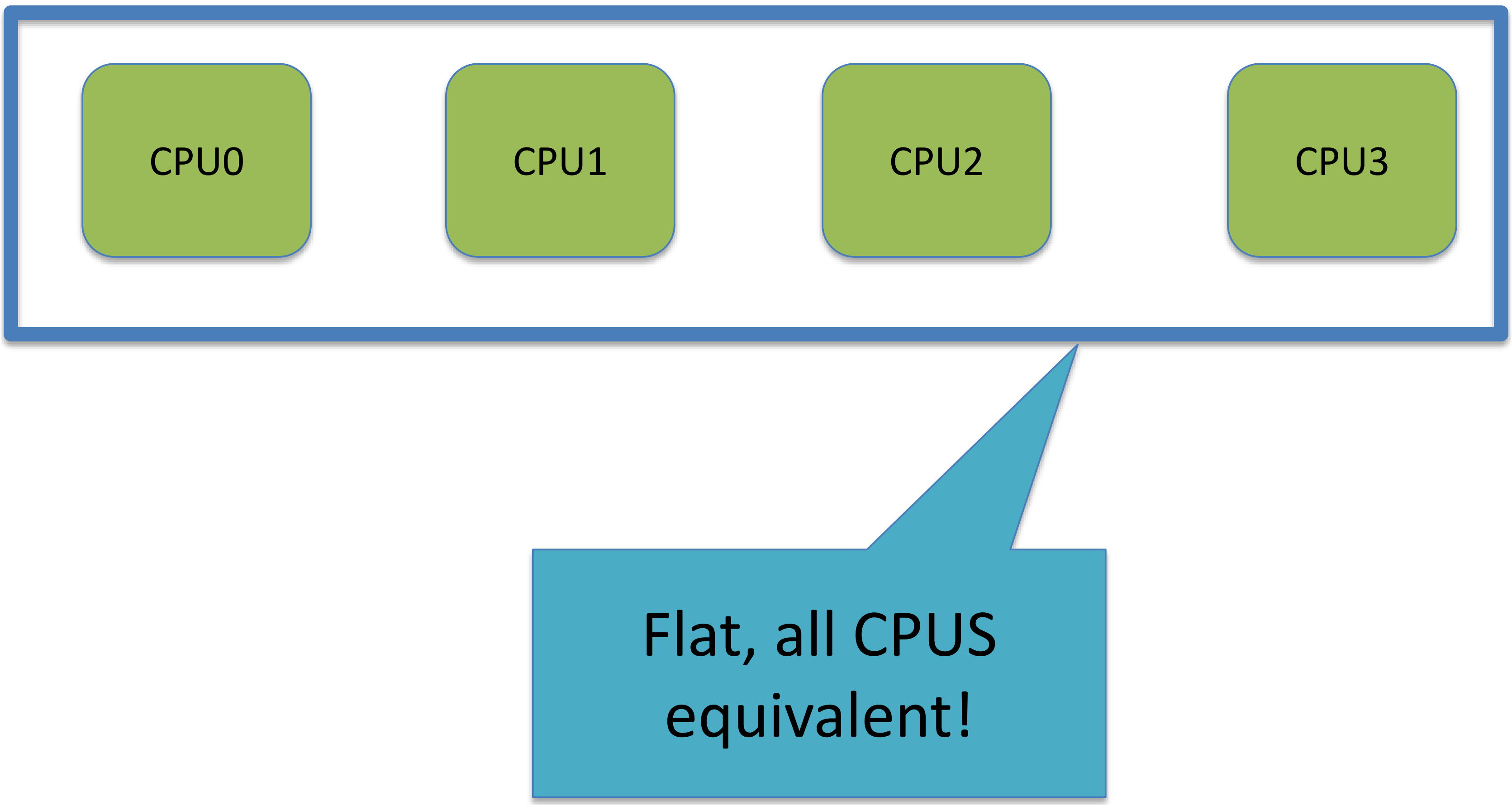
Rebalancing Tasks

- Once task in one CPU's runqueue
  - It stays on that CPU?
- What if all processes on CPU 0 exit
  - But all of the processes on CPU 1 fork more children?
- We need to periodically rebalance
  - CPU that runs out of work does the rebalance
    - ***work stealing***
- Balance overheads against benefits
  - Figuring out where to move tasks isn't free

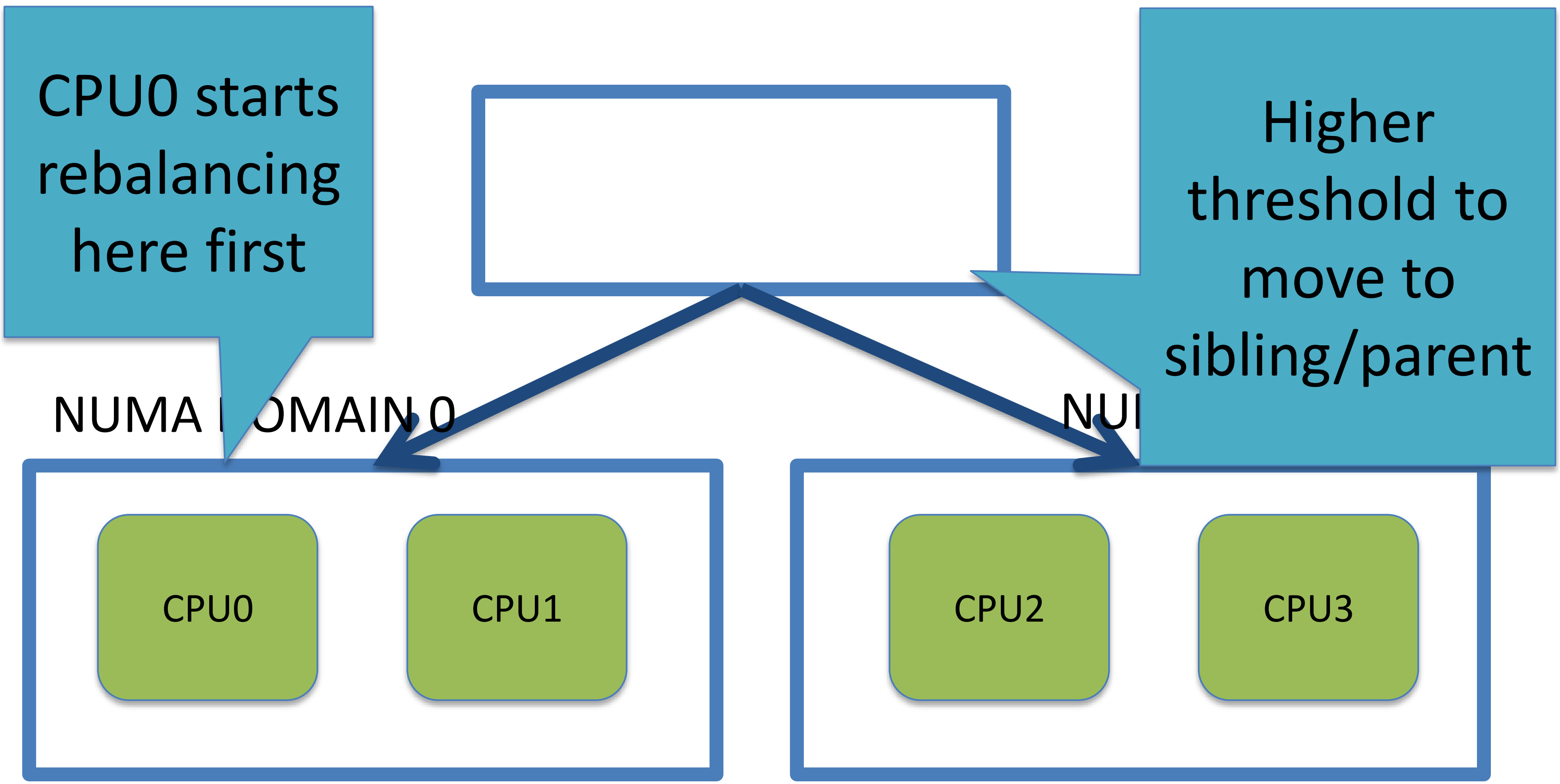
Scheduling Domains

- General abstraction for CPU topology
- “Tree” of CPUs
  - Each leaf node contains a group of “close” CPUs
- When a CPU is idle, it triggers rebalance
  - Most rebalancing within the leaf
  - Higher threshold to rebalance across a parent
- What if all CPUs are busy
  - But some have fewer running tasks than others?
    - Might still want to rebalance
      - Heuristics in scheduler to decide when to trigger rebalance

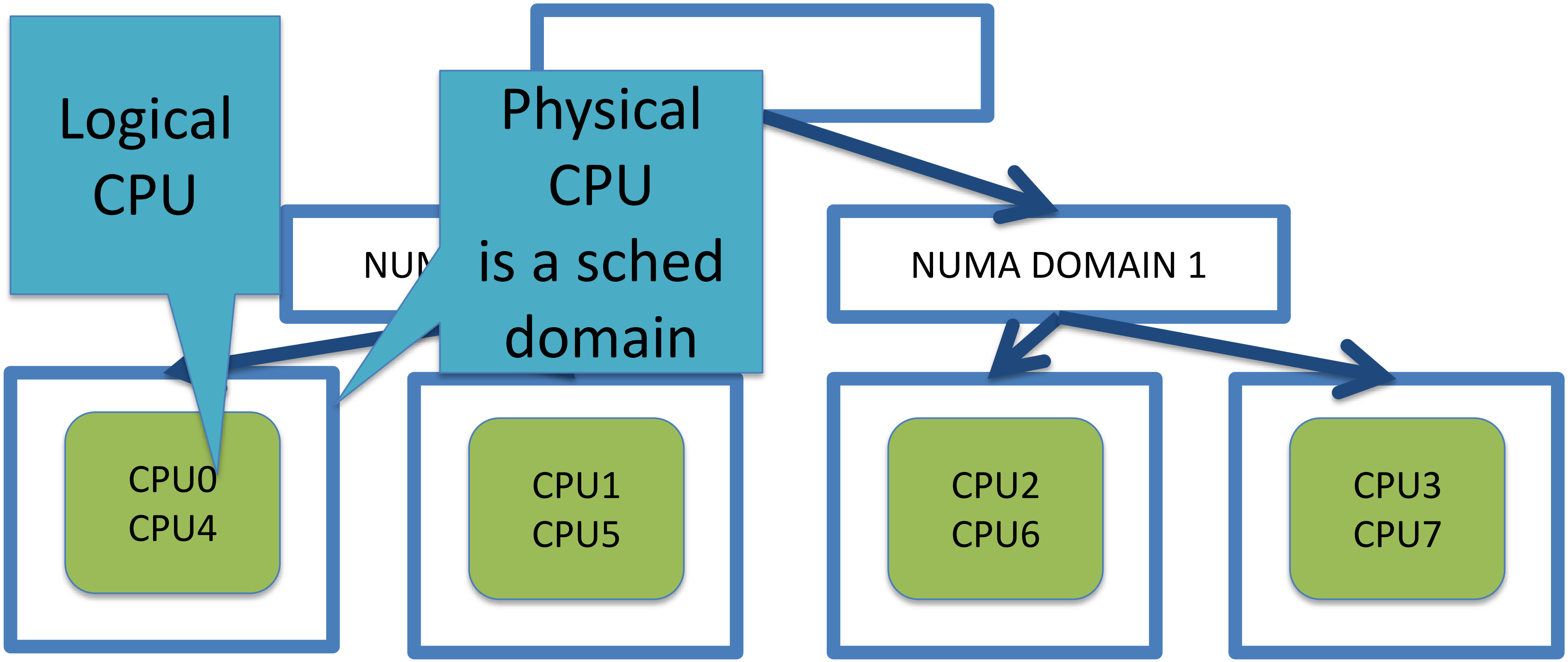
SMP Scheduling Domain



NUMA Scheduling Domains



NUMA + Hyperthreading



Rebalancing Strategy

- Read the loadavg of each CPU
  - Find the one with the highest loadavg
- Figure out how many tasks we should take
  - If worth it, take tasks
    - Need to lock runqueue
  - If not, try again later

CSE 506: Operating Systems

Read-Copy Update

RCU in a nutshell

- Many structures mostly read, occasionally written
- 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
    - Readers see consistent views of data

Principle (1/2)

- Locks have an acquire and release cost
  - Substantial, since atomic ops are expensive
- For short critical sections, cost dominates perf.

Principle (2/2)

- Reader/writer locks allow parallel execution
  - Still serialize increment/decrement of read count
    - Atomic instructions inherently “serializing”
  - Atomic instructions contend on addresses
    - Contention resolution not free, even in hardware
- Read lock becomes a scalability bottleneck
  - Even if data it protects is read 99% of time

Lock-free data structures

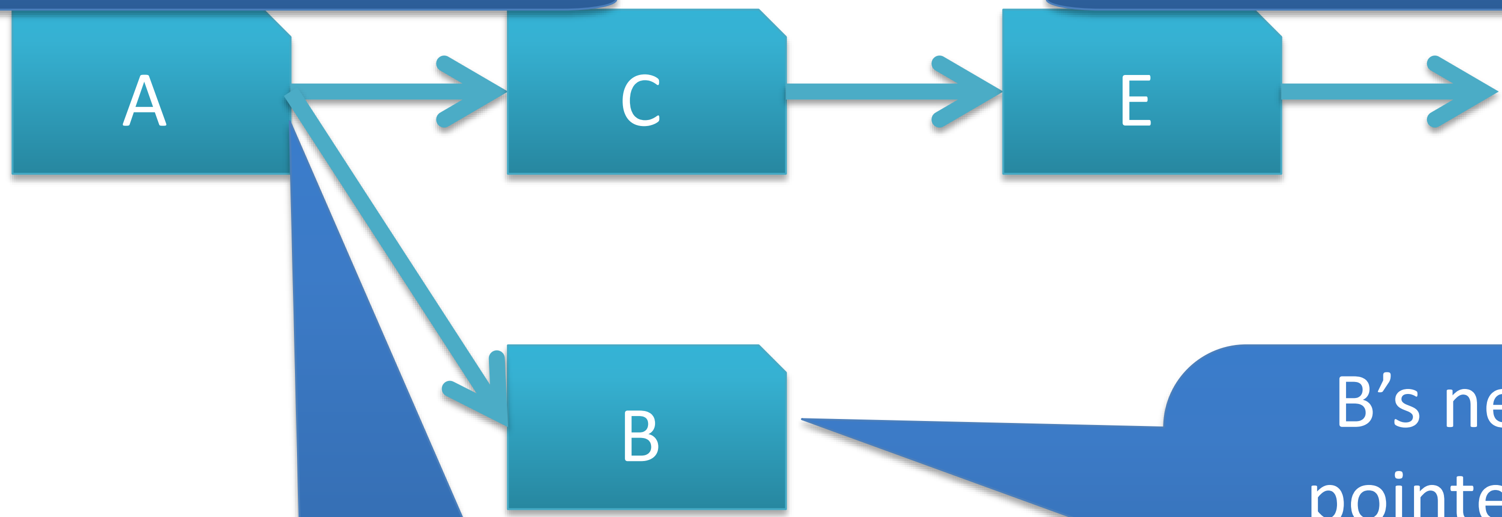
- Some data structures don’t require locks
- They are difficult to create
  - Highly error prone
  - Try to use existing ones if needed
- Can eliminate R/W locks and atomic ops

RCU: Split the difference

- Hard part of lock-free data is parallel pointer updates
  – Concurrent changes to pointers are hard
- RCU: Use locks for hard case
  – Writes take a lock
  – Reads don’t take a lock
    - But writes are careful to preserve consistency
  – Avoid performance-killing read lock (the common case)

Example: Linked lists

This implementation needs a lock

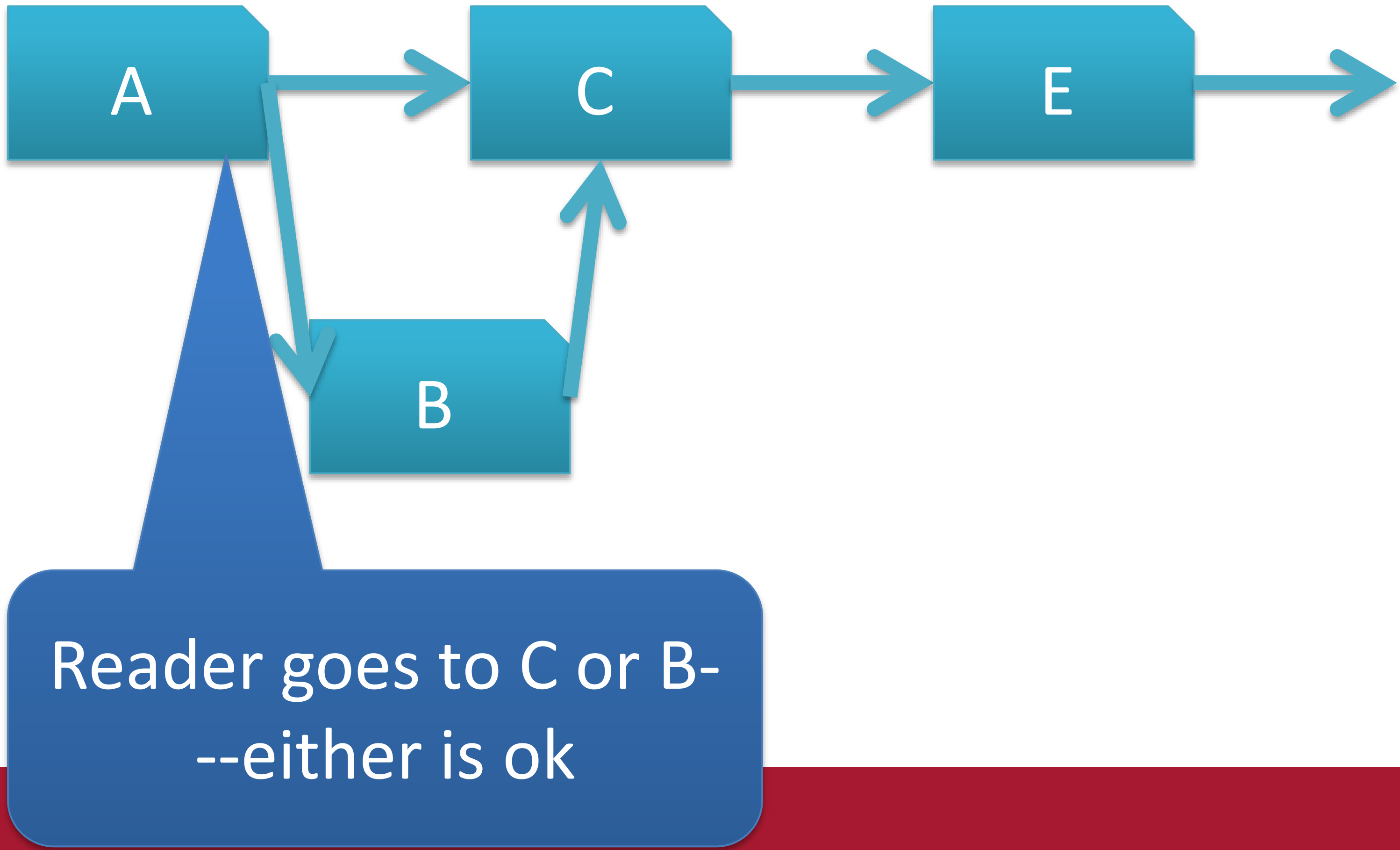


B’s next pointer is uninitialized; Reader gets a page fault

Reader goes to B

Example: Linked lists

`Insert(B)`

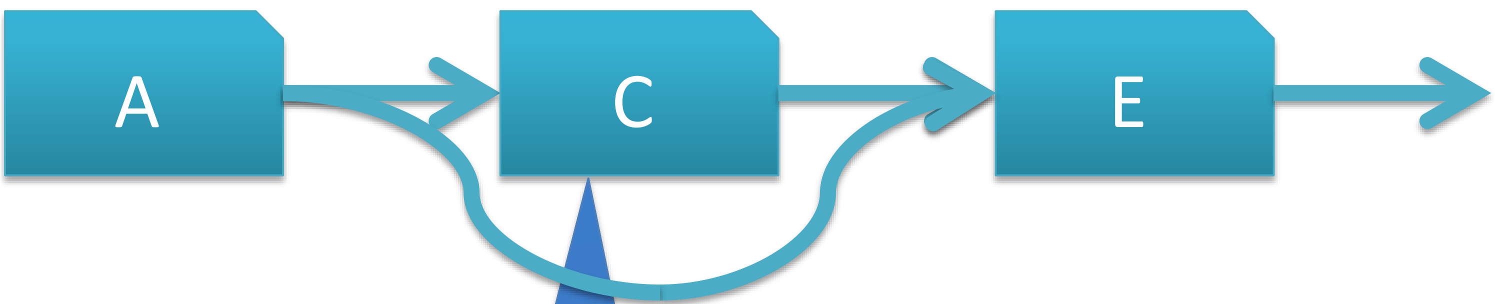


Example recap

- First create node B
  - Set up all outgoing pointers
- Then we overwrite pointer from A
  - No atomic instruction or reader lock needed
    - Either traversal is safe
- Reader can never follow a bad pointer
  - Writers still serialize using a lock

Example 2: Linked lists

`Delete (C)`



Reader may still be looking at C. When can we delete?

Problem

- Logically remove node by making it unreachable
  - No pointers to this node in the list
- Eventually need to free the node's memory
  - When is this safe?

Worst-case scenario

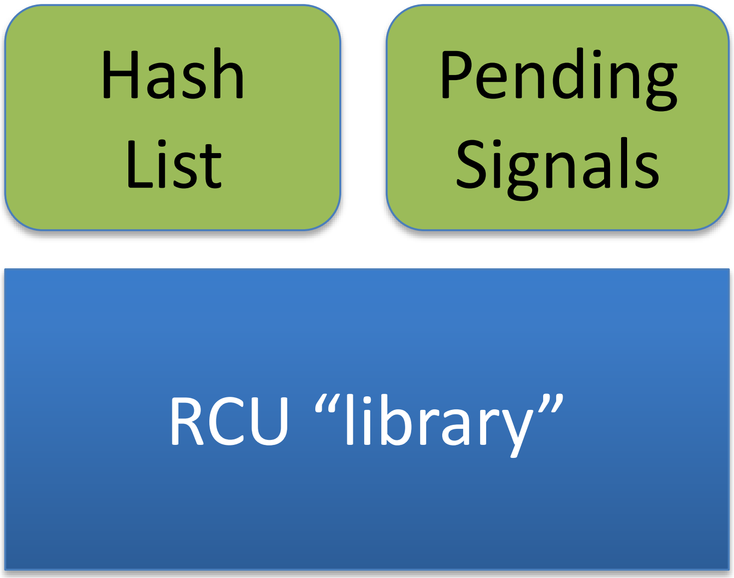
- Reader follows pointer to node X (about to be freed)
- Another CPU frees X
- X is reallocated and overwritten with other data
- Reader interprets bytes in X->next as pointer
  - Page fault in kernel

Quiescence

- Trick: Don’t allow process to sleep in RCU traversal
  – Includes kernel preemption, I/O waiting, etc.
- If every CPU has called schedule() (quiesced)
  – It is safe to free the node
    - Because schedule() can’t be called in the middle of traversal
- Each CPU counts number of schedule() calls
  – Maintain list of items to free
    - Record timestamp on each CPU
  – Wait for each CPU to call schedule
    - Do the free

Big Picture

- Carefully designed data structures
  - Readers always see consistent view
- Low-level “helper” functions encapsulate complexity
  - Memory barriers
  - Quiescence



Linux API

- Drop in replacement for read_lock:
  - rcu_read_lock()
- rcu_assign_pointer() and rcu_dereference_pointer()
  - Still need special assignment to ensure consistency
- call_rcu(object, delete_fn) to do deferred deletion