ECE 350 Real-time Operating Systems

Lecture 5: Multiprocessor Systems

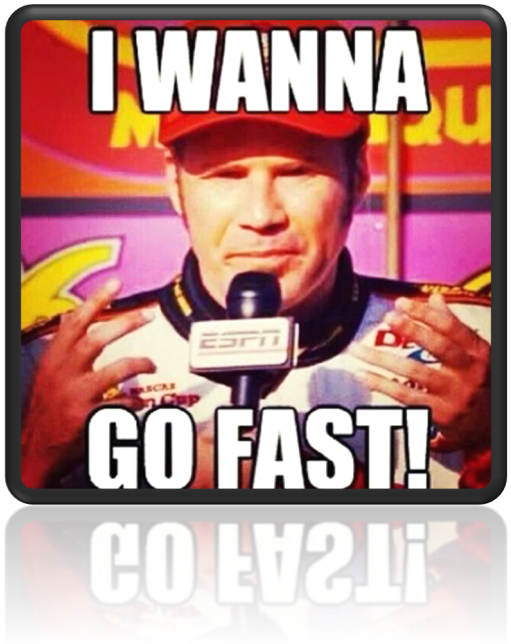
Prof. Seyed Majid Zahedi

https://ece.uwaterloo.ca/~smzahedi

Outline

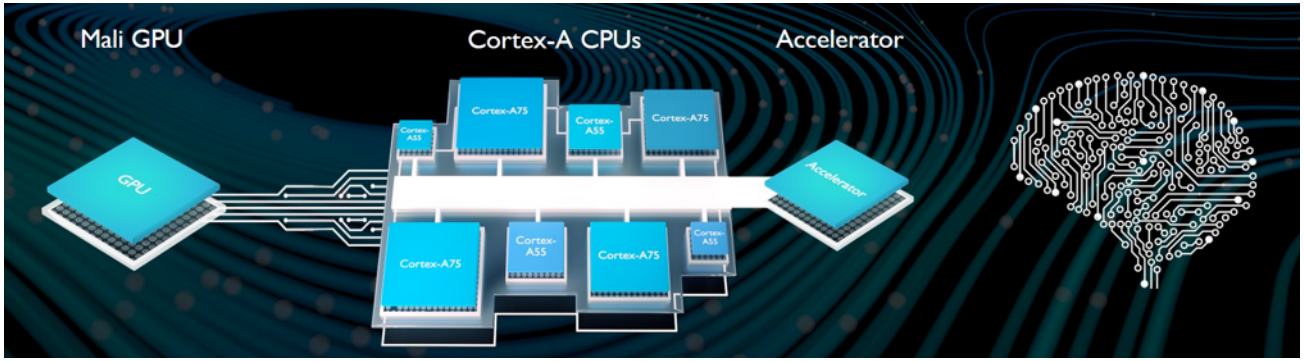
- Background on multiprocessor architecture
  - Heterogeneity, NUMA, cache hierarchy
- Multithreading on multiprocessors
  - Mitigating lock contention: MCS and RCU locks
- Load balancing in multiprocessors
  - Load tracking, NUMA domains, cache reuse, energy efficiency
- Scheduling policies for multithreaded programs
  - Oblivious scheduling, gang scheduling, …

Multiprocessor Scheduling



- Scheduling's simple invariant
  - Ready threads should be scheduled on available CPUs
- Properties of multiprocessors make implementing this invariant very hard
  - E.g., high cost of cache coherence and synchronization
  - E.g., non-uniform memory access latencies (NUMA)

Background: Symmetric vs. Asymmetric Multiprocessors

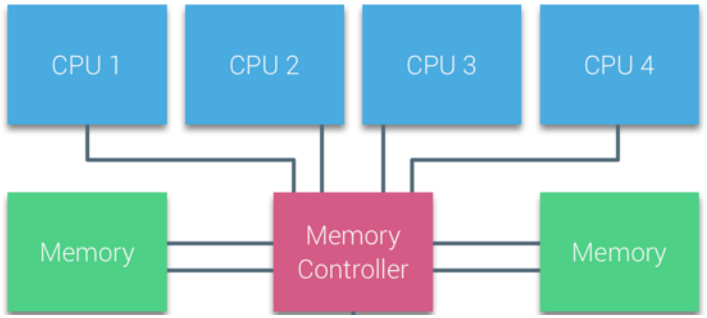


- In symmetric multiprocessor, all processors are identical
- Asymmetric multiprocessors could include processors with different performance and energy characteristics
  - Processors with different microarchitectures
  - Processors running at different voltage and frequencies
- ARM big.LITTLE architecture is example of both
  - Big CPUs have more pipeline stages, bigger caches, and smarter predictors than LITTLE CPUs
  - Big CPUs also reach higher voltage and frequencies than LITTLE ones can
- Asymmetric multiprocessors could provide better energy efficiency while meeting workloads' performance targets

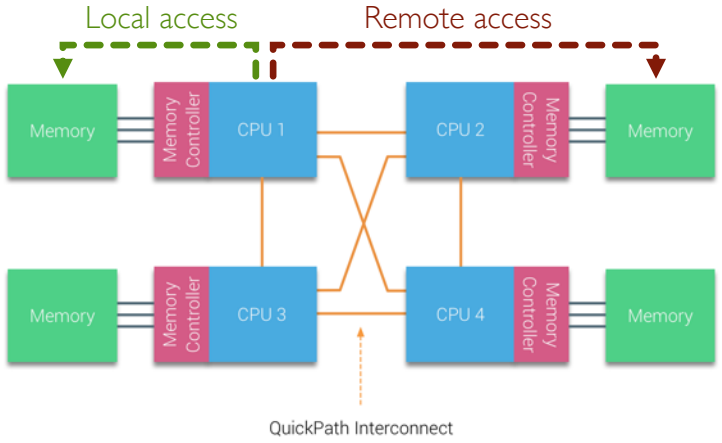
Fig. www.arm.com

Background: NUMA

- Uniform memory access (UMA): processors experience same, uniform access time to any memory block

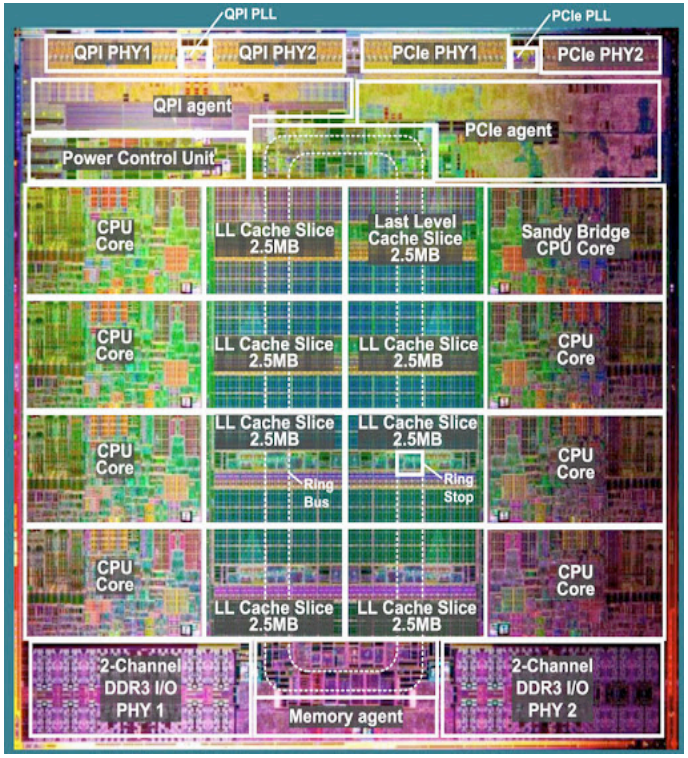


- Non-uniform memory access (NUMA): processors access their local memory blocks faster than remote memory blocks

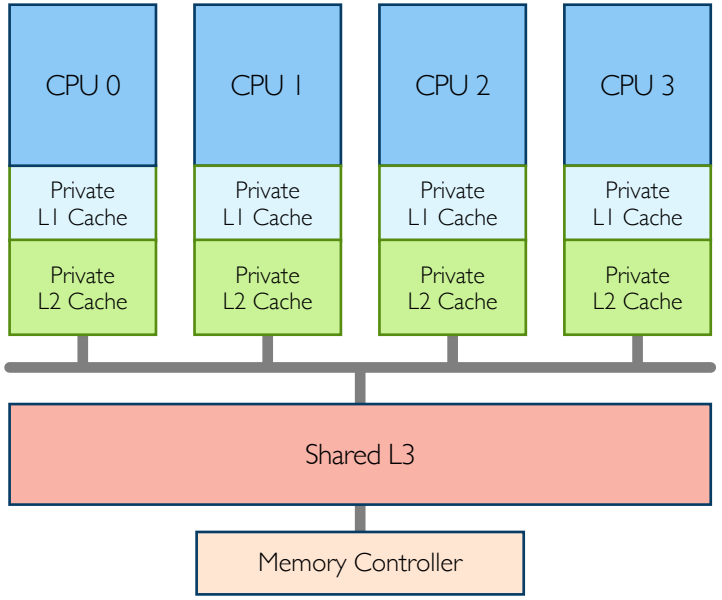


Photos from: http://www.evoventurepartners.com

Background: Cache Hierarchy in Multiprocessors



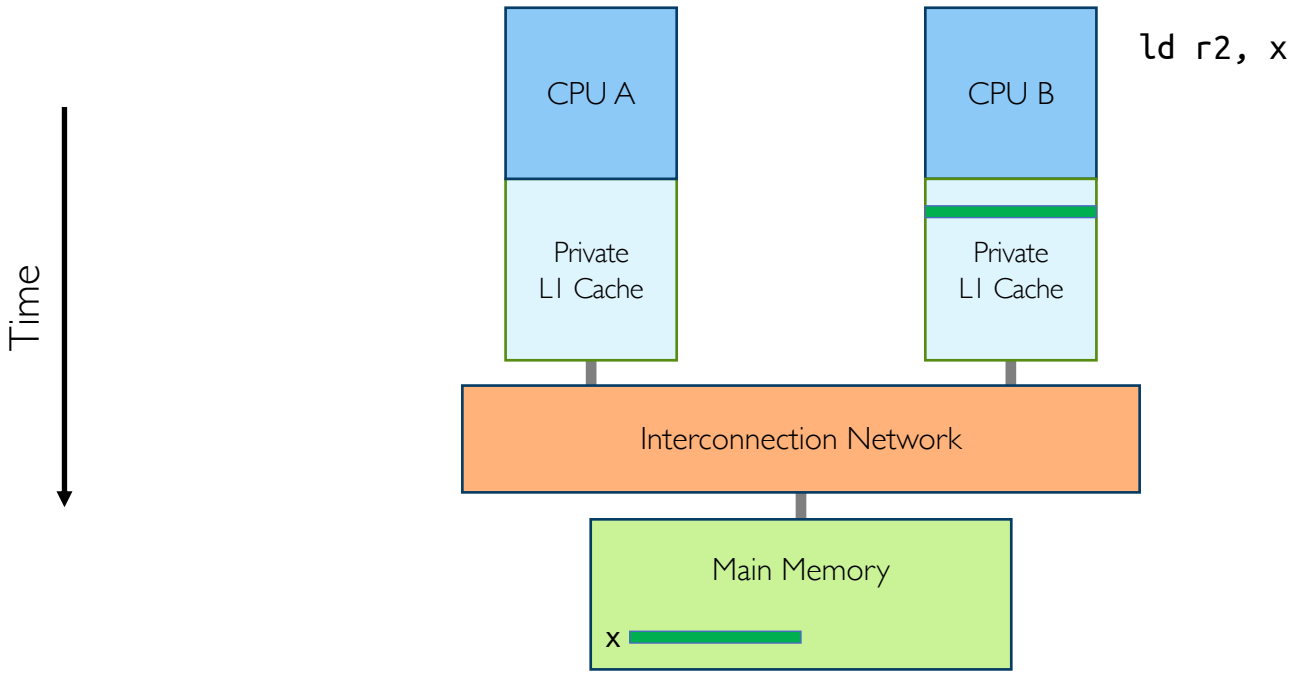
http://www.cpushack.com



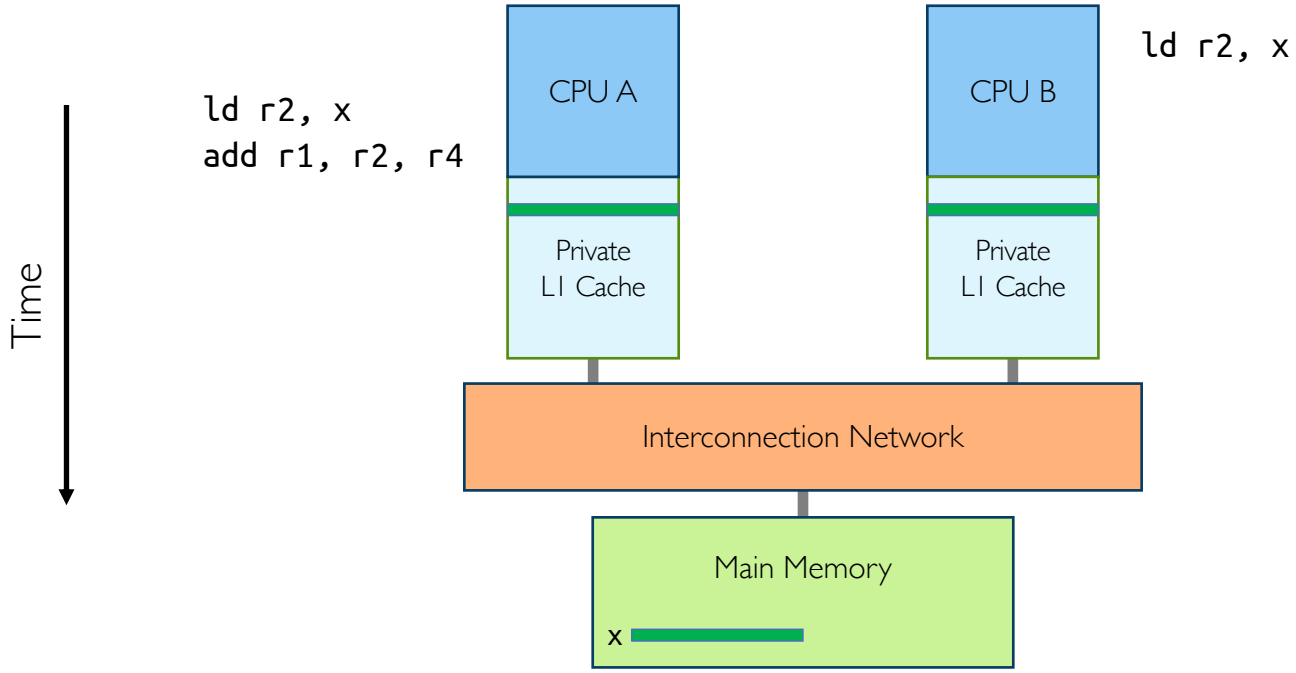
Background: Cache Coherence

- Scenario
  - Thread A modifies data inside critical section and unlocks mutex
  - Thread B locks mutex and reads data
- Easy if all accesses go to main memory
  - Thread A changes main memory; thread B reads it
- What if new data is cached at processor running thread A?
- What if old data is cached at processor running thread B?

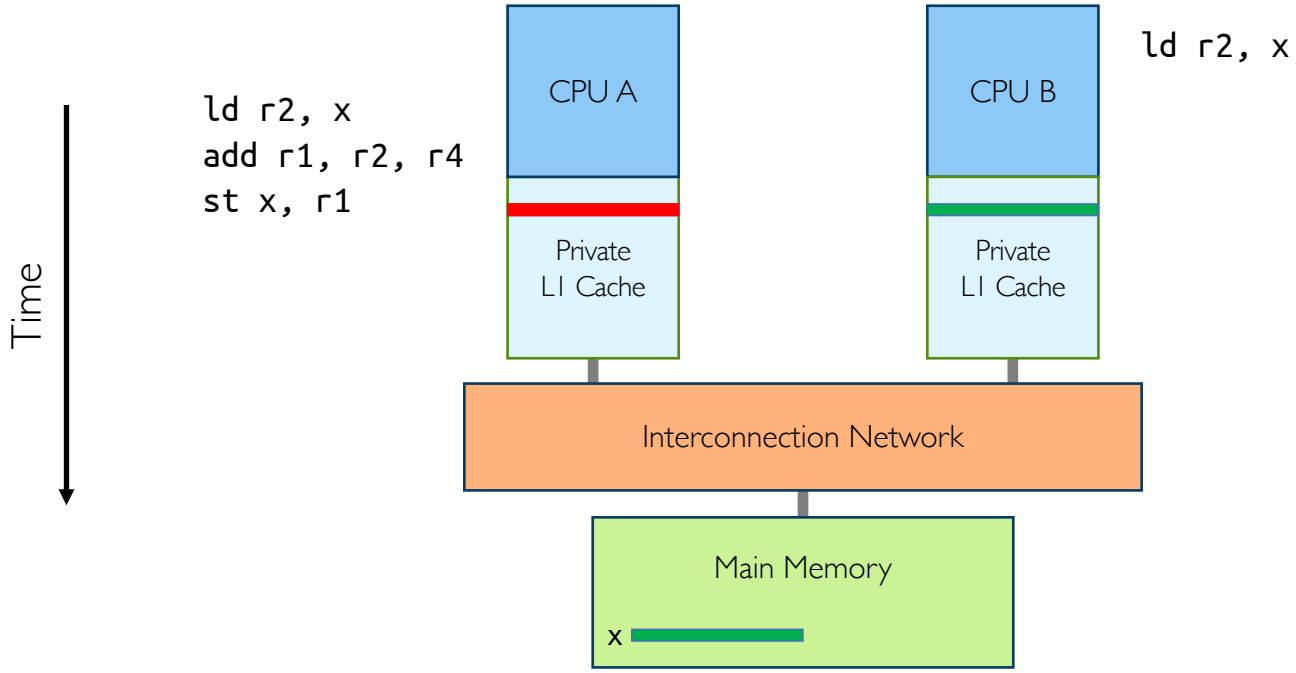
Cache-coherence Problem (Step 1)



Cache-coherence Problem (Step 2)

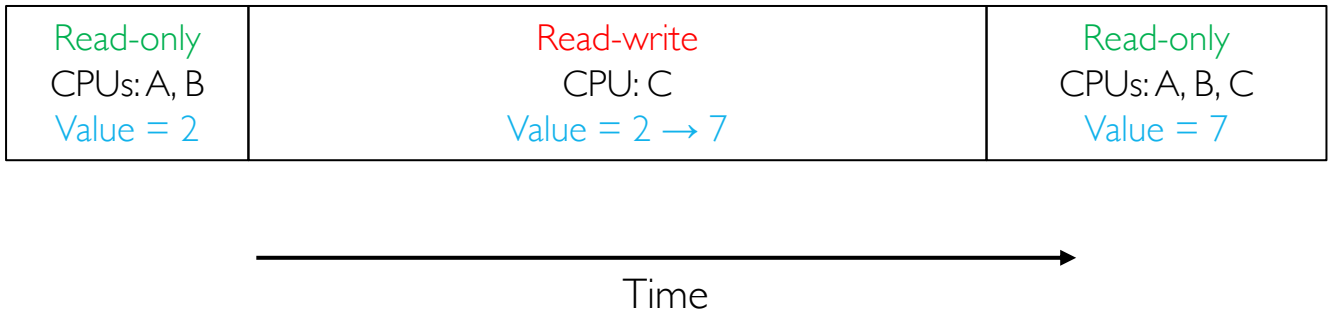


Cache-coherence Problem (Step 3)



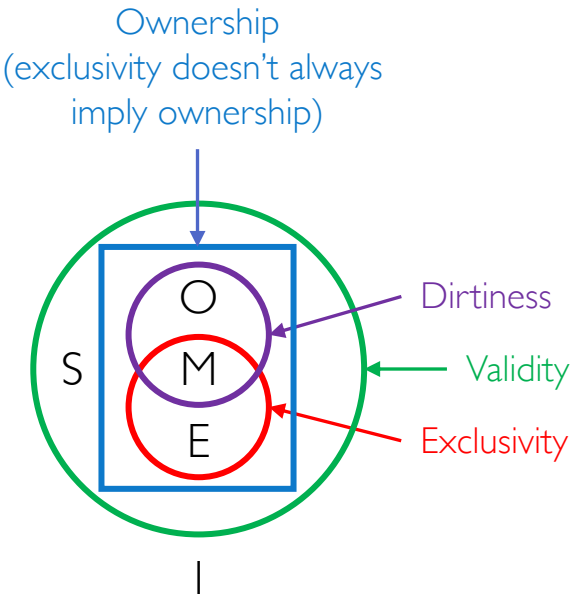
Background: Cache-coherence Protocols

- Enforce two invariants (time divided into intervals)
  - Single-writer-multiple-readers: at any time, any given block has either
    - One writer: one CPU has read-write access
    - Zero or more readers: some CPUs have read-only access
  - Up-to-date data: value at the beginning of each interval is equal to value at the end of the most recently completed read-write interval



Cache-coherence States

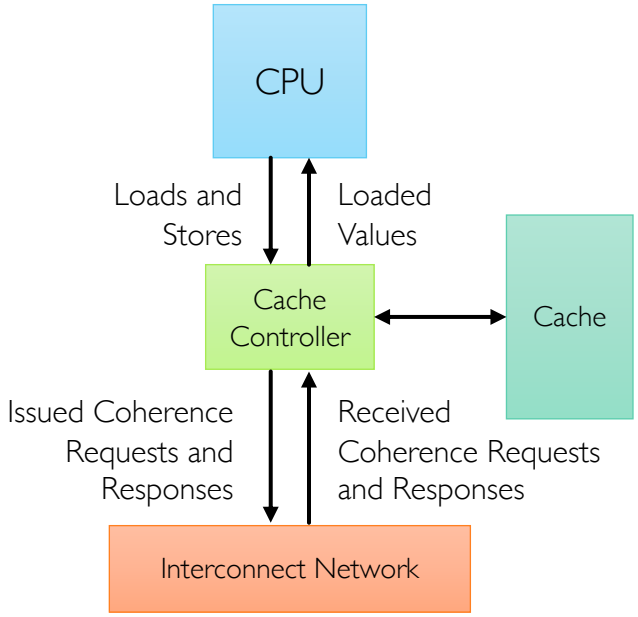
- Each cache block has coherence state denoting
  - Validity: invalid or invalid
  - Dirtiness: dirty or clean
  - Exclusivity: exclusive or not exclusive
  - Ownership: owned or not owned
  - Permission: read-only or read-write
- Five (common) states are
  - Modified (M): read-write, exclusive, owned, dirty
  - Owned (O): read-only, owned, dirty
  - Exclusive (E): read-only, exclusive, clean
  - Shared (S): read-only, clean
  - Invalid (I): invalid



- Different cache coherence protocols use subset of these states
  - E.g., MSI, MESI, MOSI, MOESI

Coherence Protocols: Big Picture

- CPUs issue `ld/st/rmw` request to caches
- If insufficient permissions, cache controller issues coherence requests
  - To where? Depends on protocol
  - *Get* requests block and *Put* writes block back to memory
  - E.g., for `rmw`, get exclusive access, prevent any other cache from reading block until instruction completes
- Owner responds to request with data

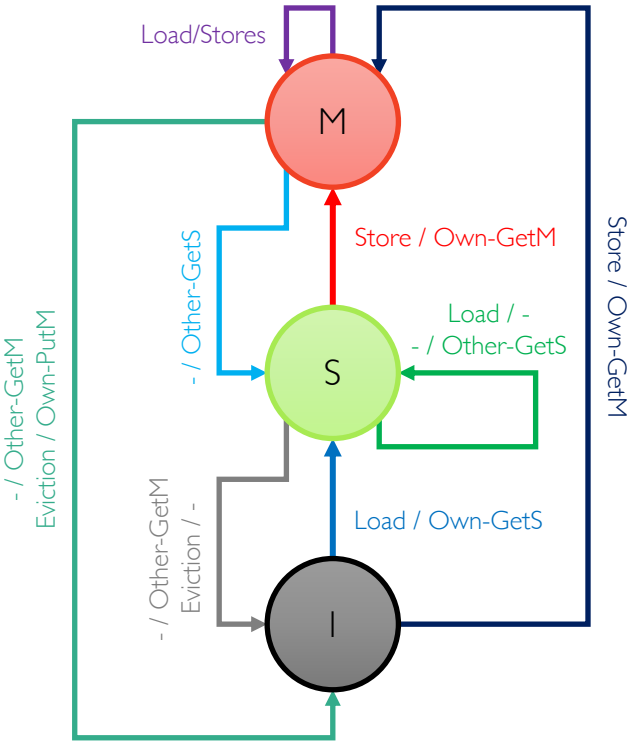


Common Coherence Requests

Request	Goal of Requestor
GetShared (GetS)	Obtain block in shared (read-only) state
GetModified (GetM)	Obtain block in Modified (read-only) state
PutShared (PutS)	Evict block in Shared state
PutExclusive (PutE)	Evict block in Exclusive state
PutOwned (PutO)	Evict block in Owned state
PutModified (PutM)	Evict block in Modified state
Upgrade (Upg)	Upgrade block state from read-only (Shared or Owned) to read-write (Modified); Upg (unlike GetM) does not require data to be sent to requestor

Snooping Cache-coherence Protocol

- Any CPU that wants block broadcasts to all CPUs
- Each cache controller "snoops" all bus transactions
- Example: MSI protocol



Directory-based Cache-coherence Protocols

- Limitations of snooping
  - Broadcasting uses lots of "bus" bandwidth
  - Snooping every transaction uses lots of controller bandwidth
  - Shared wire buses are EE nightmare
- Snooping is OK for small or medium-sized machines
  - Largest snooping system had 128 processors
- Directory-based cache-coherence protocol
  - Avoids broadcasting requests to all nodes on cache misses
  - Maintains directory of which nodes have cached copies of block
  - On misses, cache controller sends message to directory
  - Directory determines what (if any) protocol action is required
    - E.g., sending invalidations to Shared CPUs

Aside: Coherence vs. Consistency

- Coherence concerns only one memory location
  - Remember: coherence = the two invariants
  - Are not visible to software
- Consistency concerns apparent ordering for all locations
  - Defines contract between system and programmers
  - Restricts ordering of all loads and stores
  - Are visible to software

Memory Consistency Example

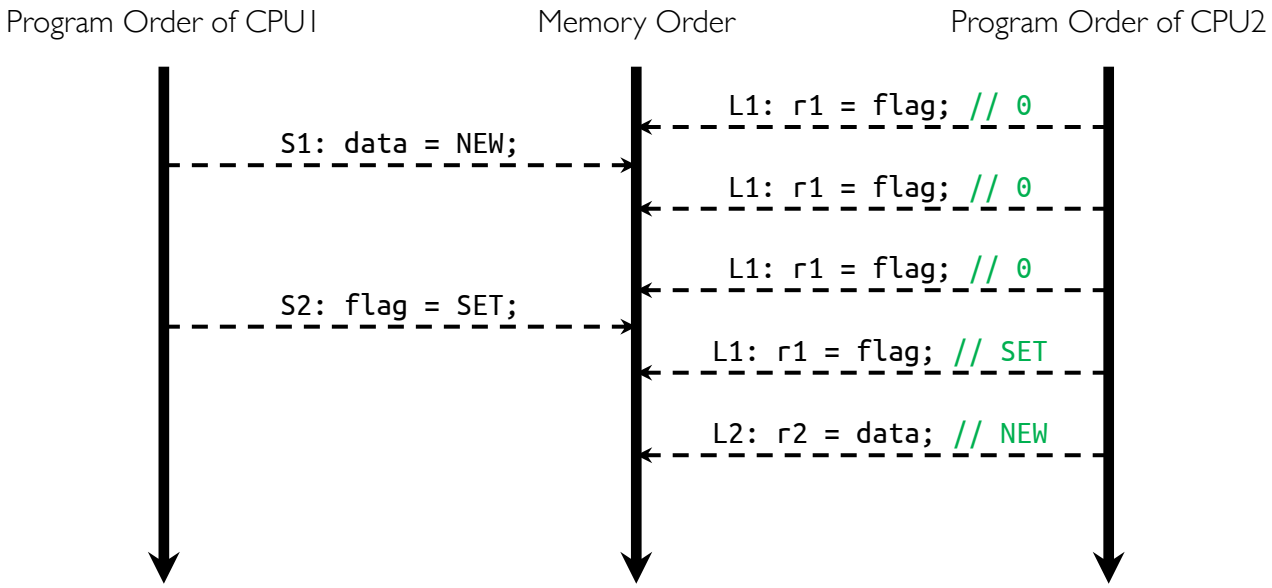
```
// initially flag = data = r1 = r2 = 0

CPU1                          CPU2
S1: data = NEW;               L1: r1 = flag;
S2: flag = SET;               B1: if (r1 != SET) goto L1;
                              L2: r2 = data;
```

- Intuition says we should print `r2 = NEW`
- Yet, in some consistency models, this isn't required!
- Coherence doesn't say anything … why?

Sequential Consistency

"The result of any execution is the same as if the operations of all processors (CPUs) were executed in some sequential order, and the operations of each individual processor (CPU) appear in this sequence in the order specified by its program."

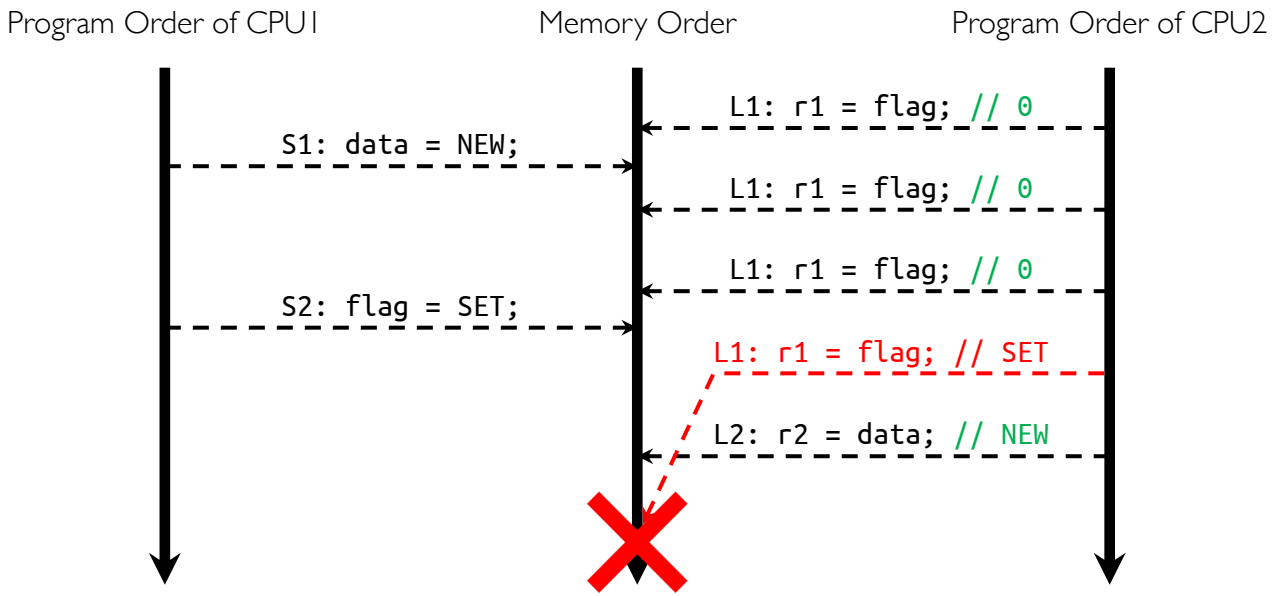
Lamport, 1979



Sequential Consistency (cont.)

"The result of any execution is the same as if the operations of all processors (CPUs) were executed in some sequential order, and the operations of each individual processor (CPU) appear in this sequence in the order specified by its program."

Lamport, 1979



x86 Memory Consistency Model

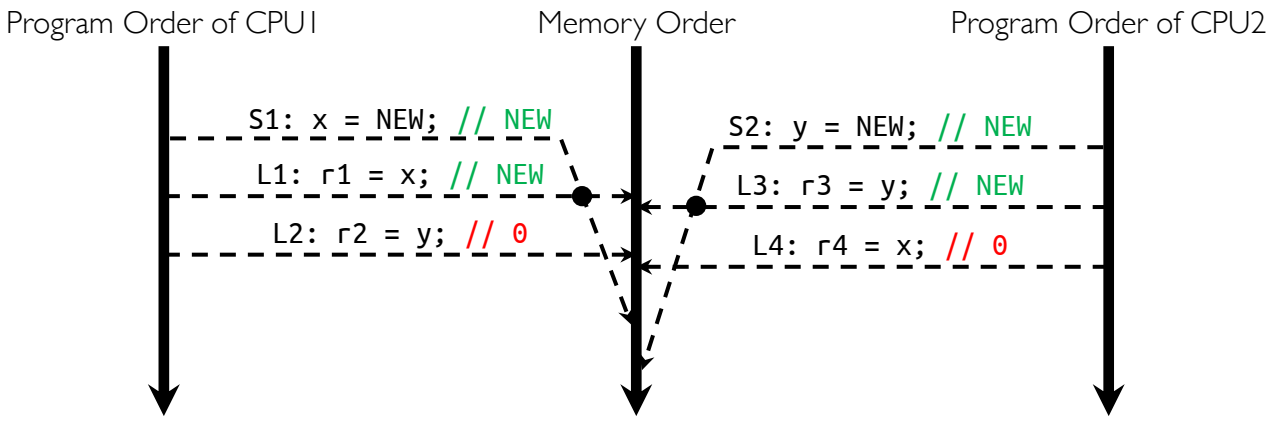
```
// initially x = y = r1 = r2 = r3 = r4 = 0
```

CPU1

```
S1: x = NEW;
L1: r1 = x;
L2: r2 = y;
```

CPU2

```
S2: y = NEW;
L3: r3 = y;
L4: r4 = x;
```

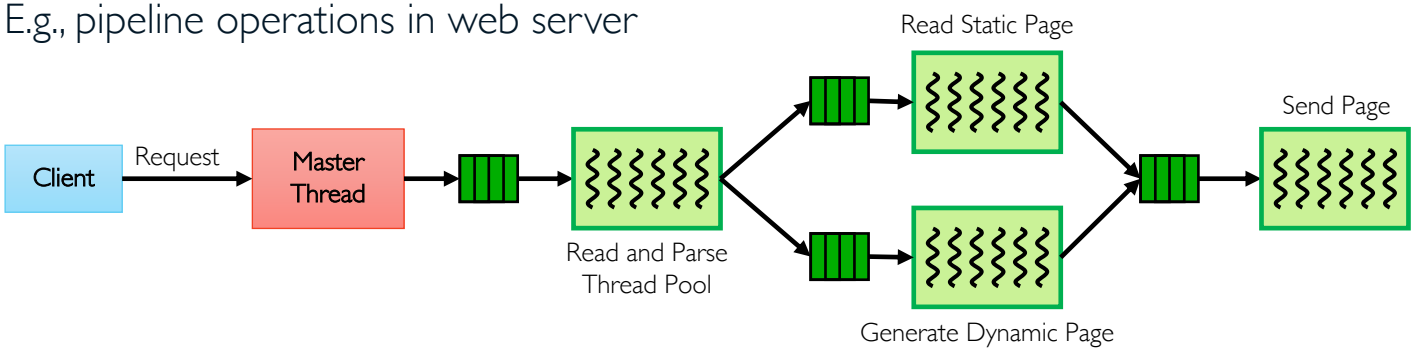


Multithreading on Multiprocessors

- Multithreading doesn't guarantee performance
  - Overheads: creating/managing threads introduces costs (memory/computation)
  - Lock contention: only one thread can hold lock at any time
  - Communication of shared data: shared data might ping back and forth between CPUs' caches (enforced by cache coherence protocol)
  - False sharing: CPUs may still have to communicate for data that is not shared because coherence protocols track blocks at fixed granularity (e.g., 64 byte)

Reducing Lock Contention

- Fine-grained locking
  - Partition object into subsets, each protected by its own lock
  - E.g., divide hash table key space
- Per-processor data structures
  - Partition object so that most/all accesses are made by one processor
  - E.g., partition heap into per-processor memory regions
- Staged architecture
  - Each stage includes private state and set of worker threads
  - Different stages communicate by sending messages via producer/consumer queues
  - Each worker thread pulls next message and processes it
  - E.g., pipeline operations in web server



What If Locks are Still Mostly Busy?

- MCS (Mellor-Crummey-Scott) lock
  - Optimize lock implementation for when lock is contended
- RCU (read-copy-update) lock
  - Efficient readers/writers lock used in Linux kernel
  - Readers proceed without first acquiring lock
  - Writer ensures that readers are done
- Both rely on atomic read-modify-write instructions

Lock Contention Example

```
// test-and-set
Counter::Increment() {
    while (test&set(&lock));
    value++;
    lock = FREE;
    memory_barrier();
}
```

- What happens if many processors run this code?
  - All processors will try to gain exclusive access to memory location of **`lock`** to execute **`test&set`**
  - Processor that wants to set value of **`lock`** to `FREE` also needs exclusive access
  - Value of **`lock`** pings between caches

Lock Contention Example (cont.)

```
// test and test-and-set
Counter::Increment() {
    while (lock == BUSY ||
                test_and_set(&lock));
    value++;
    lock = FREE;
    memory_barrier();
}
```

- Would this solve the problem?
  - Once **lock** is **FREE**, its value must be communicated to all processors
  - Eventually one processor reads **FREE** and sets it to **BUSY**
  - Other processors will still be busy fetching **FREE** value of **lock**, trying to execute **test&set**, preventing **lock** value to be set to **FREE** again

MCS Lock

- Maintain list of threads waiting for lock
  - Front of list holds the lock
  - Tail is last thread in list
  - New thread uses `compare&swap` to add itself to list

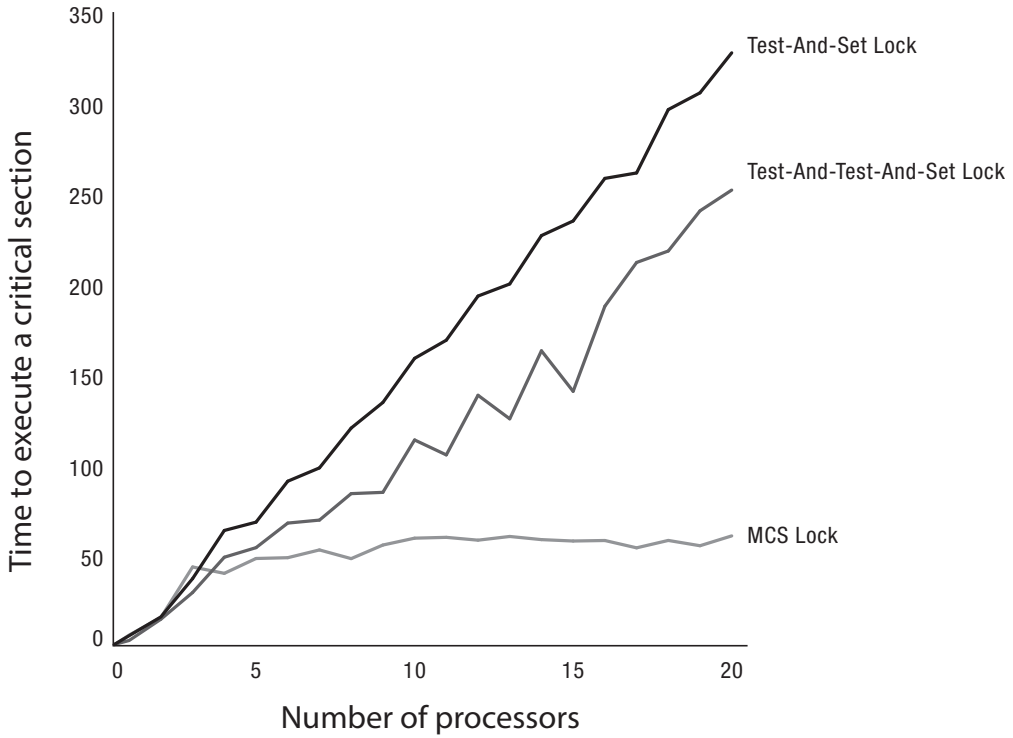
- Lock is passed by setting `flag` for next thread in line
  - Next thread spins while its flag is `FALSE`

MCS Lock Implementation

```
class MCSLock {
  private:
    TCB *tail = NULL;
}


MCSLock::unlock() {
    if (compare_and_swap(&tail,
          myTCB, NULL)) {
        // If tail == myTCB, no one
        // is waiting, lock is free
    } else {
        // Someone is waiting,
        // spin until next is set
        while (myTCB->next == NULL);
        // Tell next thread to proceed
        myTCB->next->MCS_flag = TRUE;
    }
}
```

```
MCSLock::lock() {
    TCB *oldTail = tail;
    myTCB->next = NULL;
    while (!compare_and_swap(&tail,
              oldTail, &myTCB)) {
        // Try again if someone else
        // changed tail
        oldTail = tail;
    }
    // If oldTail == NULL, lock acquire,
    // otherwise, wait!
    if (oldTail != NULL) {
        myTCB->MCS_flag = FALSE;
        memory_barrier();
        oldTail->next = myTCB;
        while (myTCB->needToWait);
    }
}
```

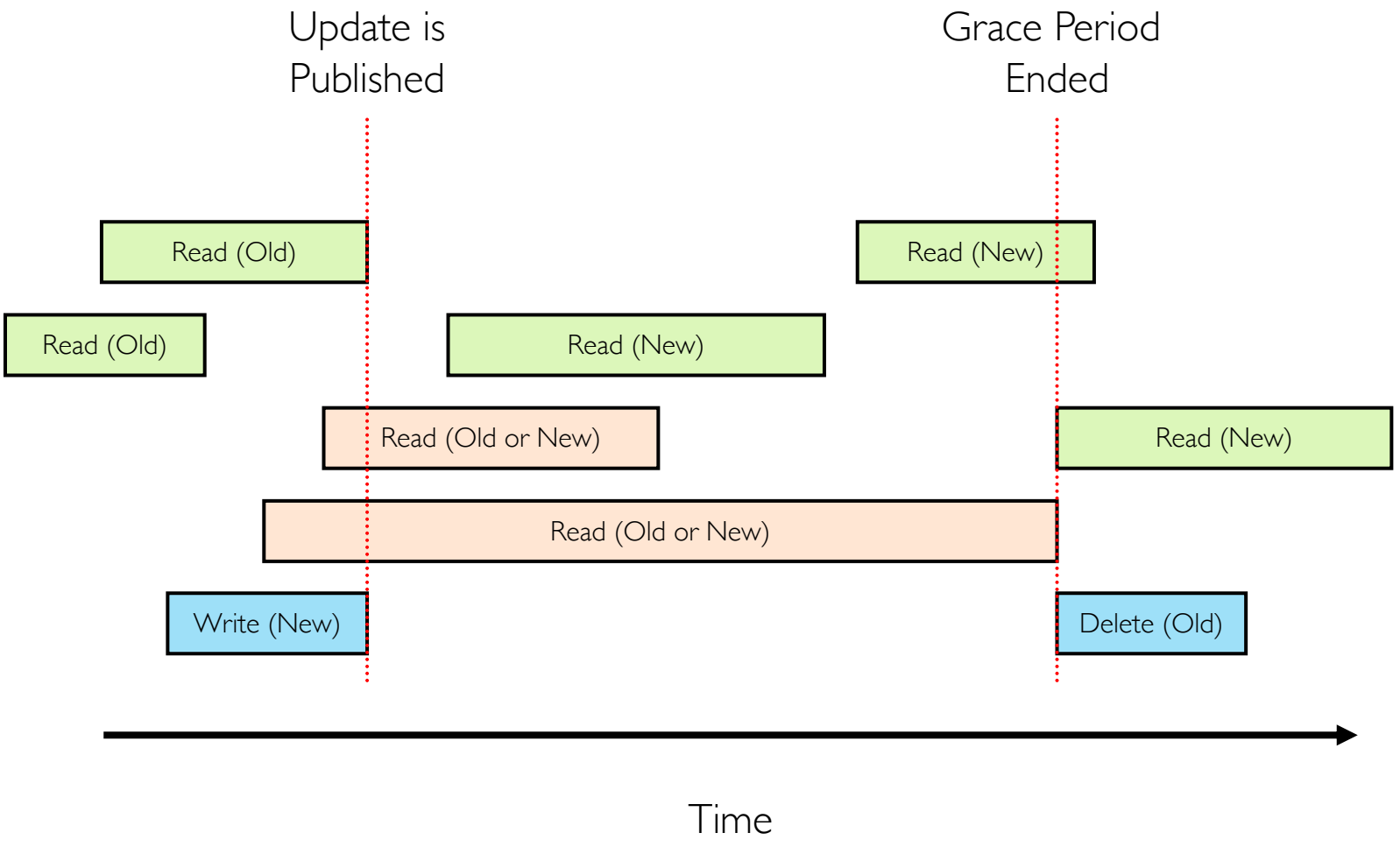
Performance Comparison



RCU Lock

- Goal is to allow very fast reads to shared data
  - Reads proceed without first acquiring a lock
- Writers are only allowed to make restricted updates
  - Make a copy of old version of data
  - Updates the copy
  - Publishes pointer to updated copy with single atomic instruction
- Multiple concurrent versions of data can coexist
  - Readers may see old or new version
- RCU needs integration with thread scheduler
  - To guarantee all readers complete within grace period, and then garbage collect old version

RCU Lock in Action

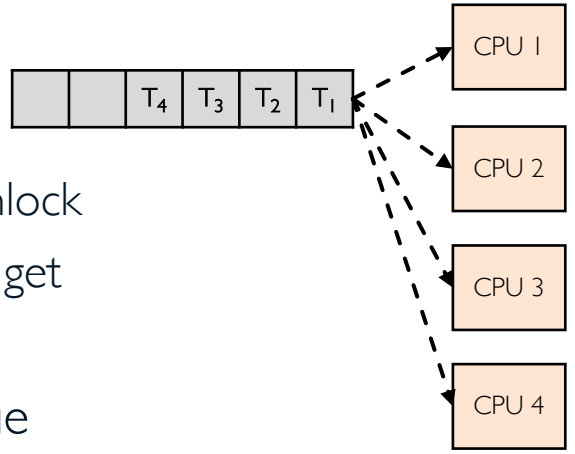


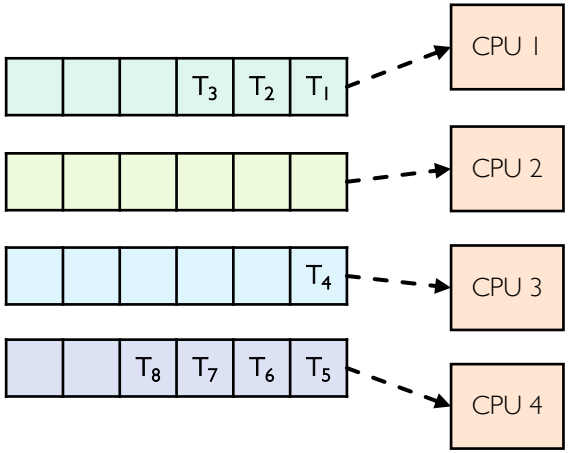
RCU Lock Implementation

- Readers disable interrupts on entry
  - Guarantees they complete critical section in a timely fashion
  - No read or write lock
- Writer
  - Acquire write lock
  - Compute new data structure
  - Publish new version with atomic instruction
  - Release write lock
  - Wait for time slice on each CPU
  - Only then, garbage collect old version of data structure

Scheduling in Multiprocessors

- There could be one ready queue for all CPUs
- Notice any problems?
  - Single bottleneck: contention for ready queue's spinlock
  - Limited cache reuse: lack of data locality as threads get scheduled on different CPUs
- Solution: each CPU has its own private ready queue
- Notice any problems?
  - Load balancing: some CPUs might be idle while threads pile up on others ready queues
  - Priorities: one queue has one low-priority thread and another has ten high-priority ones
- Solution: load balancing!





Load Tracking Metric

- Option 1: balance queues based on number of threads
  - Two queues could have same number of threads, in one all are high-priority, and in other all are low-priority
  - Low-priority threads get the same amount of CPU time as high-priority ones
- Option 2: balance queues based on threads' weights
  - One queue has 1 thread with weight 9, other has 9 threads with weight 1
  - High-priority thread often sleeps, its CPU must frequently steal work from other queue
- Option 3: balance queues based on threads' weights and CPU utilization
  - Metric called *load* used in Linux's completely fair scheduling (CFS)
  - If thread does not use much of CPU, its *load* will be decreased accordingly
  - CFS further uses `autogroup` feature to normalize *load* based on number of threads per process

Load Balancing is Simple, isn't it?

"I suspect that making the scheduler use per-CPU queues together with some inter-CPU load balancing logic is probably trivial . Patches already exist, and I don't feel that people can screw up the few hundred lines too badly."

Linus Torvalds, 2001[1]

- In 2001, server systems typically had only few CPUs
- Today, datacenter servers could have hundreds of CPUs
- Load balancing has become expensive procedure
  - Computation: requires iterating over dozens of ready queues
  - Communication: involves modifying remotely cached data structures, causing extremely expensive cache misses and synchronization
- Scheduler try to avoid running load-balancing procedure often
  - Results in cases where CPU becomes idle while others are busy
- Schedule runs "emergency" load balancing (*work stealing*) when CPUs become idle

[1] L. Torvalds. The Linux Kernel Mailing List. http://tech-insider.org/linux/research/2001/1215.html, Feb. 2001.

Processor Affinity

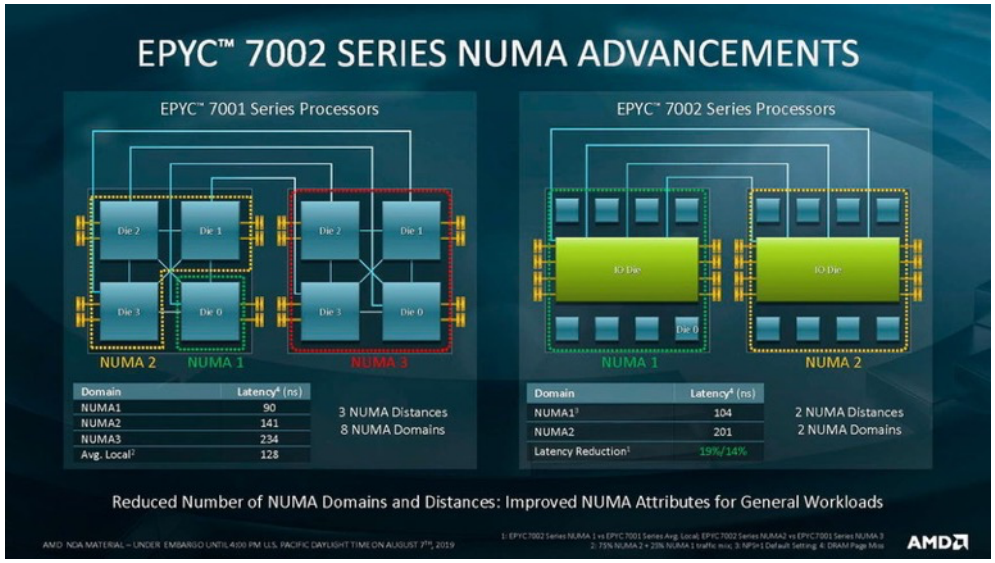
- Processor affinity: when thread runs on one processor, cache contents of that processor stores recent memory accesses by that thread

- Migration: load balancing may affect affinity as threads move between processors
  - Performance of migrated thread suffers because it loses its cached contents
  - Migration is justified only if performance loss is less than waiting time

- Soft affinity: OS tries to keep threads on same CPU, but no guarantees

- Hard affinity: OS allows threads to specify set of CPUs they may run on

NUMA Nodes and Scheduling Domains



- Group of processors sharing last level cache (LLC) form NUMA node
- NUMA nodes are grouped according to their level of connectivity
- Each level of hierarchy is called scheduling domain
  - Scheduling domain differ from each processor's perspective
  - E.g., above photo shows scheduling domains from perspective of processors in NUMA1 node
- Load balancing is run for each scheduling domain

Load Balancing Newly-ready Threads

- Where should OS schedule newly-created threads?
  - Linux typically schedules new threads on the same processor that runs parent thread!
- Where should OS schedule awakened threads?
  - For cache reuse, OS might want to schedule thread on same node it was put to sleep
  - But this is not optimal if all processors of the same node are busy while other nodes are idle

- One strategy
  - If there are no idle processors, schedule newly-ready threads on original node
  - If there are idle processors, pick one that has been idle for longest period
    - Processor with short idle time could still be overloaded with lots of threads that frequently sleep due to synchronization or I/O

Energy Efficiency Scheduling (EAS)

- Goal of EAS is to maximize performance per watt
  - Performance is instruction per second
  - Maximizing performance per watt = minimizing energy per instruction

- EAS needs to know each processor's
  - Capacity: amount of work it can do when running at its highest frequency compared to the most capable processor in system
  - Energy model: power cost table per *performance domain*

- Using utilization and capacity, EAS estimates how big/busy each task/CPU is

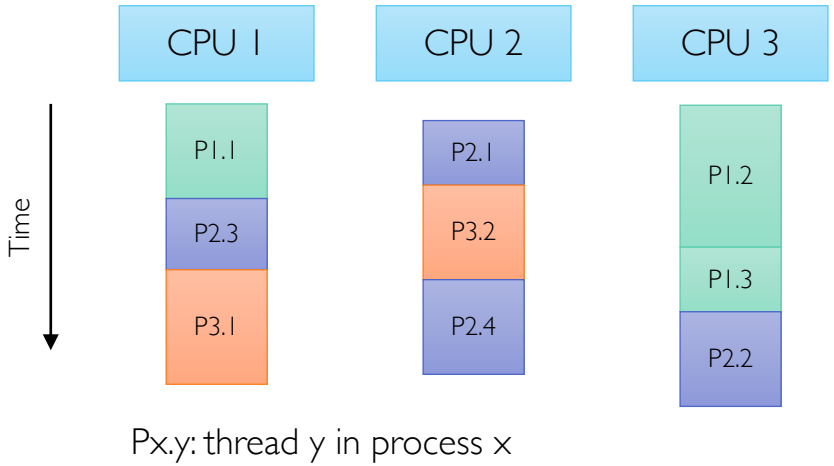
- EAS assigns awakened threads to processor that is predicted to yield best energy consumption without harming performance

Fig. www.arm.com

Aside: History of Linux CPU Scheduler

- Linux kernel between v2.4 and v2.6 used *O(n) Scheduler*
  - Only one ready queue for all CPUs implementing MFQ
  - High algorithmic complexity, performed poorly for highly multithreaded workloads
- In 2003, it was replaced by *O(1) Scheduler* (from v2.6.0 to v2.6.22)
  - Supporting constant *O(1)* scheduling complexity
  - Better scalability, not so friendly with interactive and audio applications
  - Initially successful, soon required lots of patches for new architectures (e.g., NUMA and SMT)
- In 2007, *Completely Fair Scheduler* was introduced replacing *O(1) Scheduler* (from v2.6.23)
  - Sacrificing *O(1)* complexity for *O(log n)* (read-black tree)
  - Implementing fair scheduling with lots of heuristics and optimizations for corner cases
- In 2009, *Brain Fuck Scheduler* was introduced as alternative to CFS
  - Not intended for mainline kernel, providing simple scheduler not needing heuristics and tunning
  - Only one ready queue, behavior that resembles weighted RR
  - Eventually retired in favor of *Multiple Queue Skiplist Scheduler*

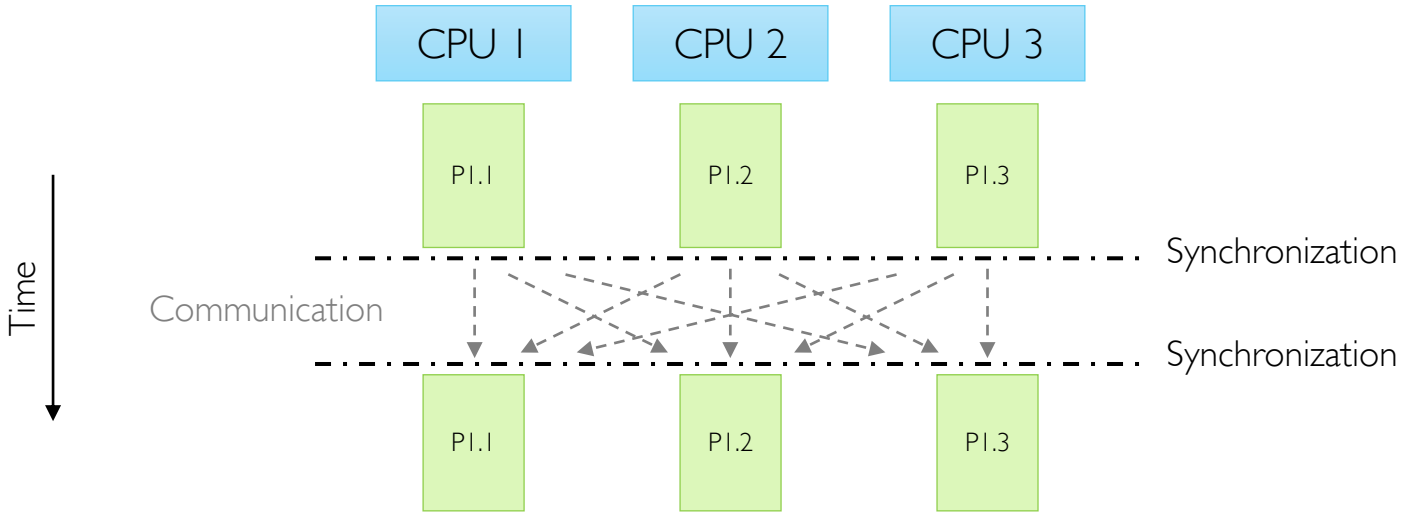
Scheduling Multithreaded Programs: Oblivious Scheduler



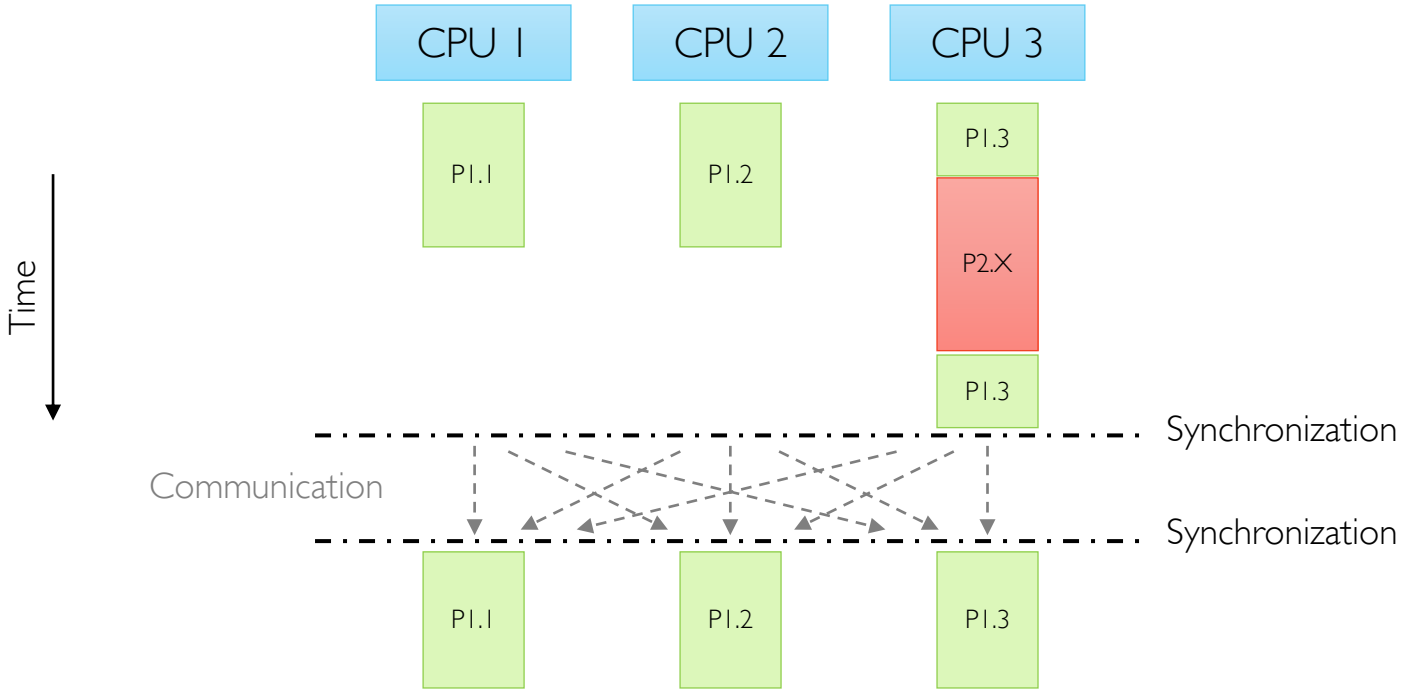
- Oblivious scheduling: CPUs independently schedule threads in their queue
  - Each thread is treated as independent thread
- What happens if one thread gets time-sliced while others are still running?
  - Assuming program uses mutexes and condition variables, it will still be correct
  - Performance, however, could suffer if threads actually depend on one another

Problem with Oblivious Scheduling: Bulk Synchronous Delay

- Data parallelism is common programming design pattern
  - Data is split into roughly equal sized chunks
  - Chunks are processed independently by threads
  - Once all chunks are processed, threads synchronize and communicate their results to next stage of computation
  - E.g., Google MapReduce

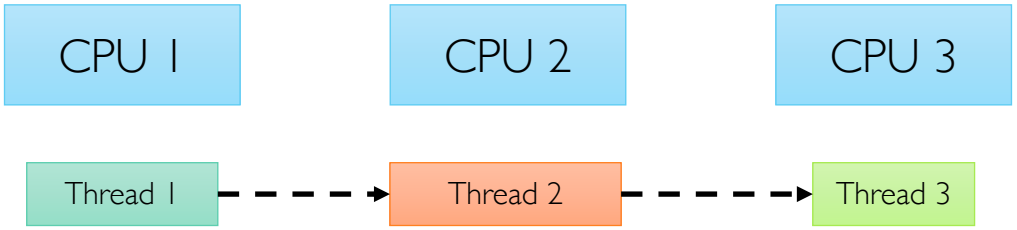


Problem with Oblivious Scheduling: Bulk Synchronous Delay



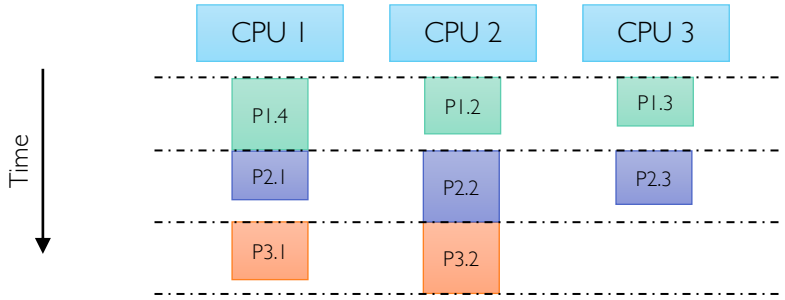
- At each step, computation is limited by the slowest thread
- If thread is preempted on one CPU, it stalls all other threads

Problem with Oblivious Scheduling: Producer-Consumer Delay



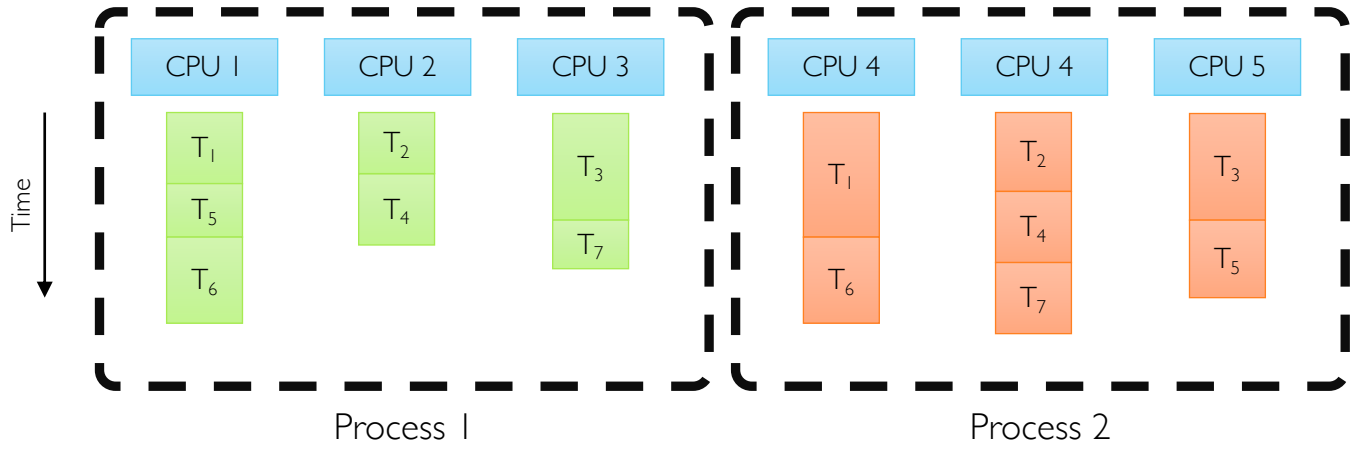
- Producer-consumer design patter is also very common
- Preempting a thread on one CPU stalls all others in the chain
- Some other problems with oblivious scheduling
  - Preempting a thread on the critical path will slow down the entire process
  - Preempting thread that has locked mutex stalls others until it is re-scheduled

Gang Scheduling



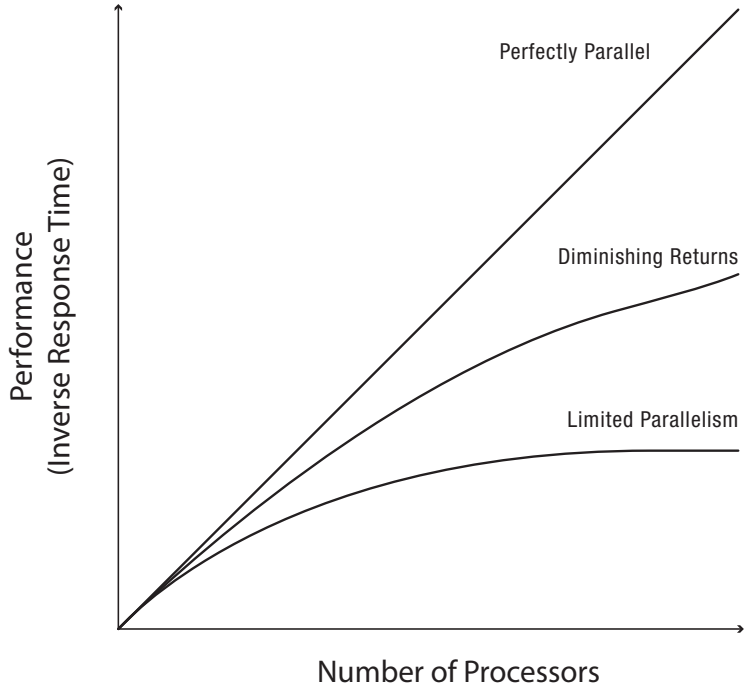
- Time is divided into equal intervals
- Threads from same process are scheduled at beginning of each interval
- Notice any problems?
  - CPU cycles are waisted when threads have different lengths
  - Some CPUs remain idle when a process doesn't have enough threads for all CPUs

Space Sharing



- Each process is assigned a subset of CPUs
  - Minimizes processor context switches

How Many CPUs Does a Process Need?

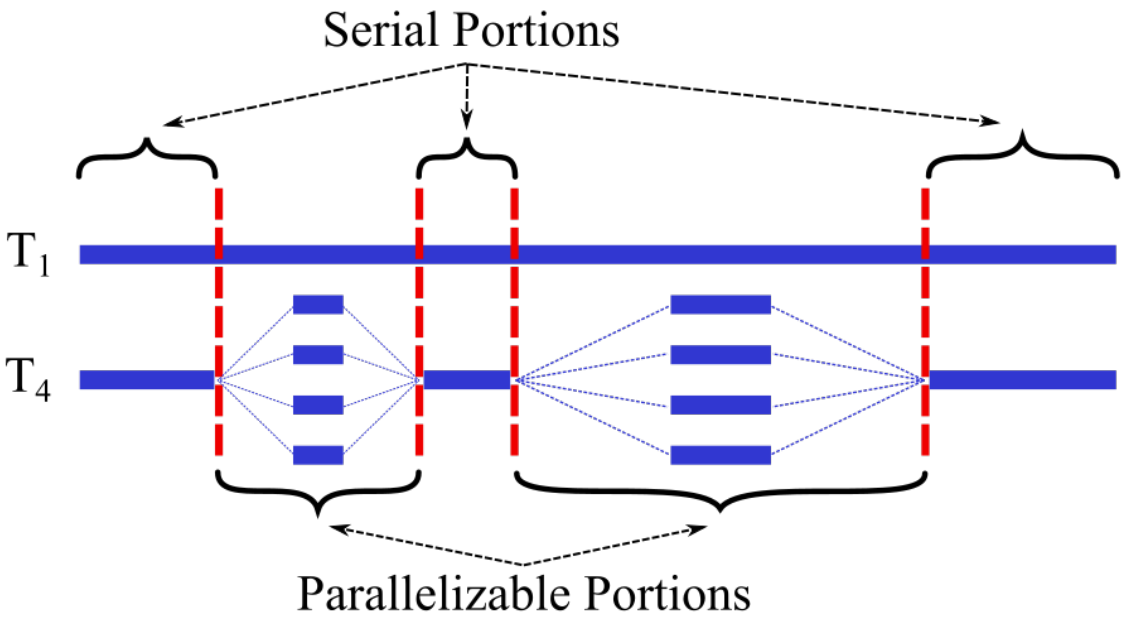


- There are overheads
  - E.g., creating extra threads, synchronization, communication
- Overheads shift the curve down

Amdahl's Law

[G. Amdahl 1967]

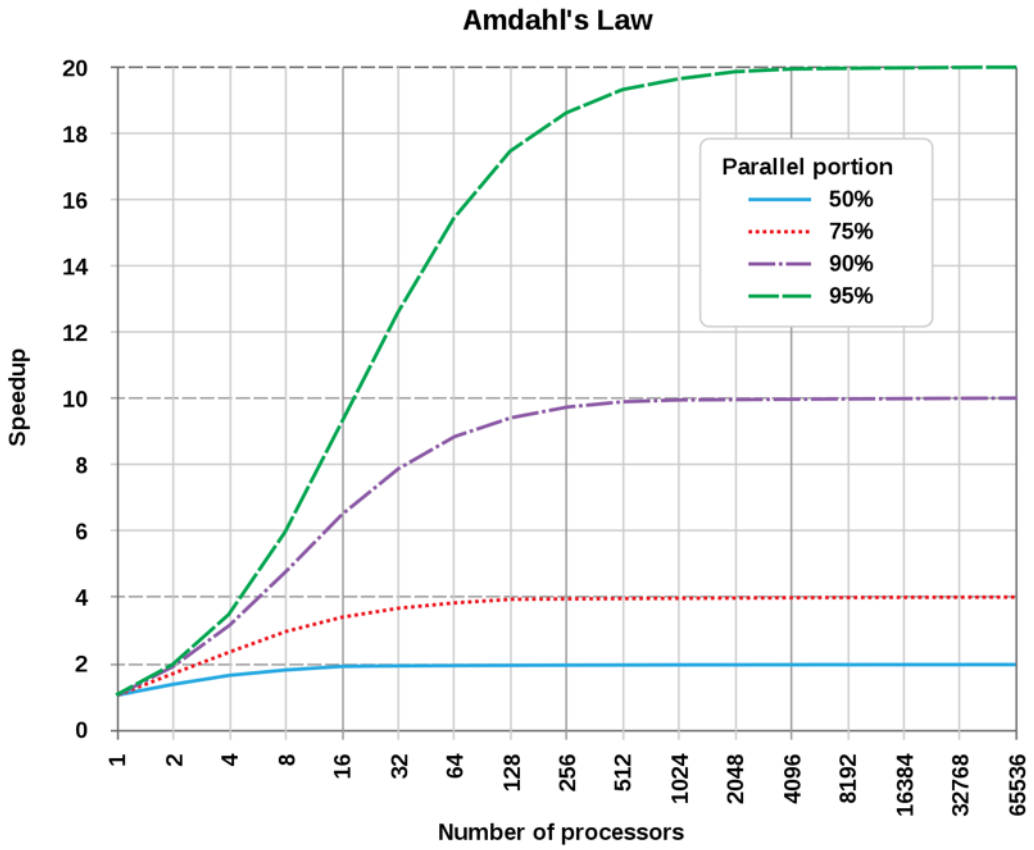
- Architects use it to estimate upper bounds on speedups



$$\text{Speedup(x)} = \frac{T_1}{T_x} = \frac{T_1}{(1-F)T_1 + \frac{FT_1}{x}} = \frac{x}{x(1-F)+F}$$

Amdahl's Law (cont.)

[G. Amdahl 1967]



$$\text{Speedup(x)} = \frac{T_1}{T_x} = \frac{T_1}{(1-F)T_1 + \frac{FT_1}{x}} = \frac{x}{x(1-F)+F}$$

What Portion of Code is Parallelizable?

[Allen Karp and Horace Flatt 1990]

- Expert programmers may not know!
- Fortunately, we can measure speedup

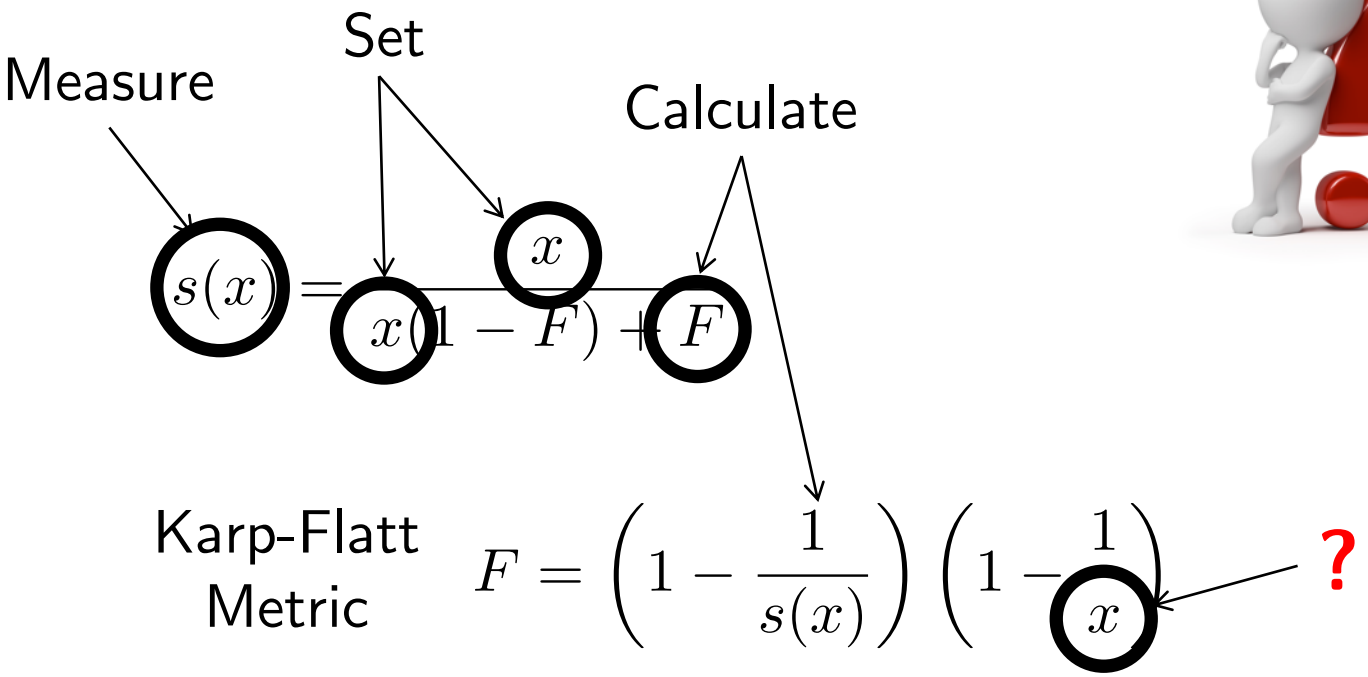
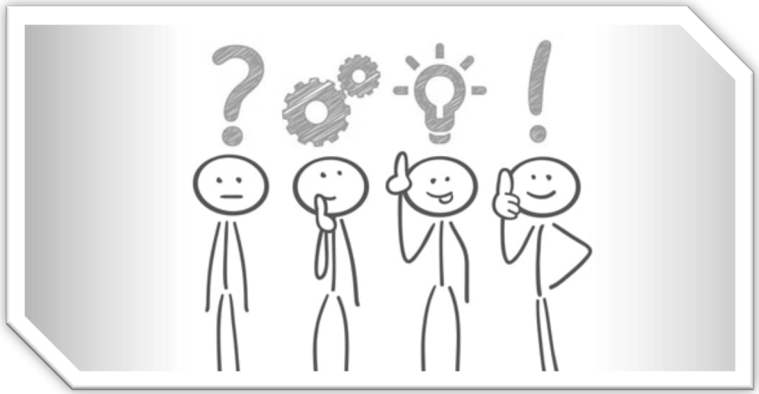


Fig. criticallyrated.files.wordpress.com

Summary

- Properties of multiprocessors make scheduling them complex
  - Symmetric vs. asymmetric multiprocessors
  - Uniform vs. non-uniform memory access
  - Cache coherence problem
- Concurrency doesn't always mean higher performance
  - Lock contention solutions: MCS and RCU locks
- Scheduling multiprocessors typically boils down to efficient load balancing
  - Load balancing newly-ready threads based on different goals
- Orchestrating threads when scheduling multiple multithreaded programs could have great impact on performance
  - Gang scheduling, space sharing

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, Canny, Sorin