Lecture 17:

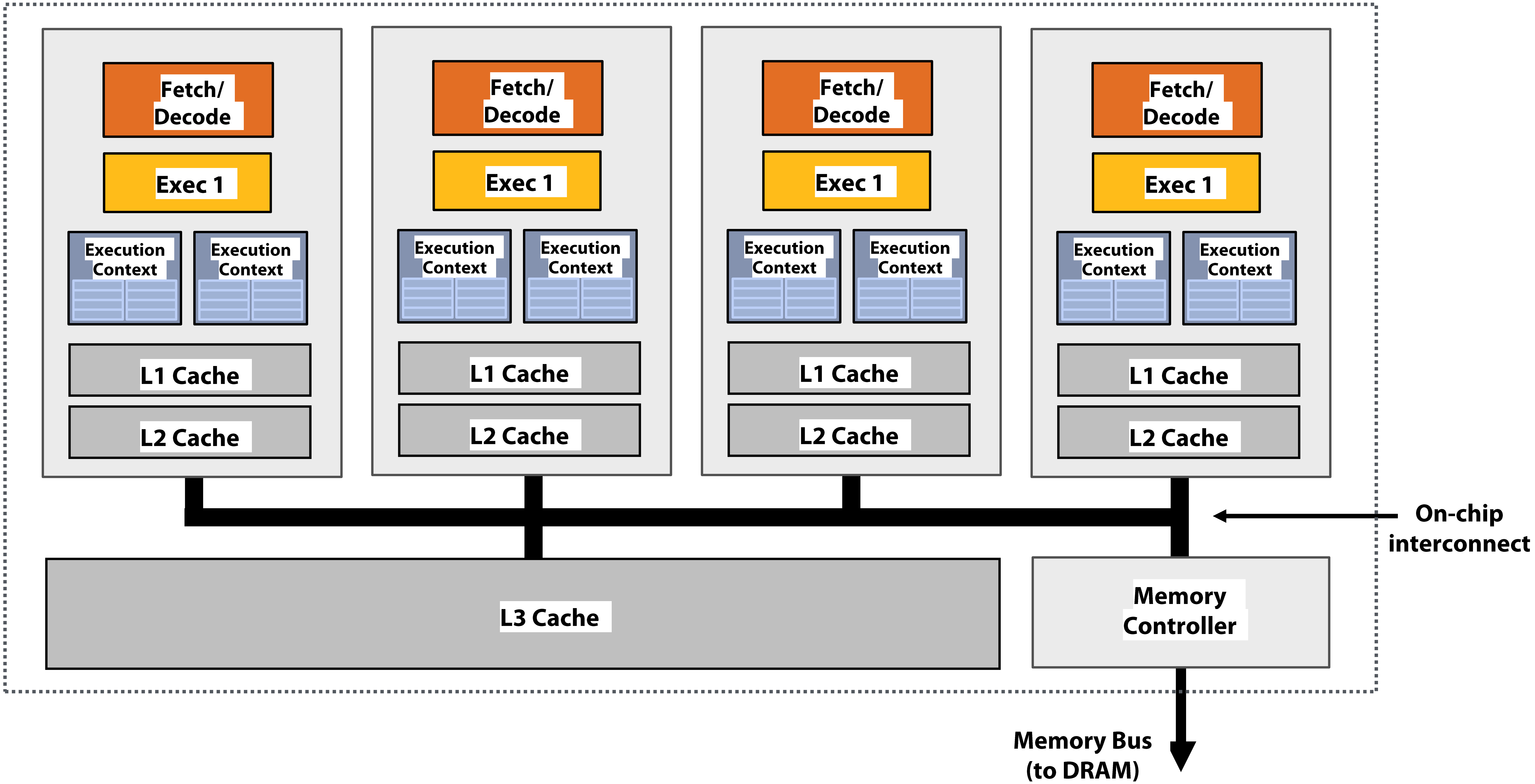
Implementing Synchronization

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2018

Review: how threads map to cores… again!

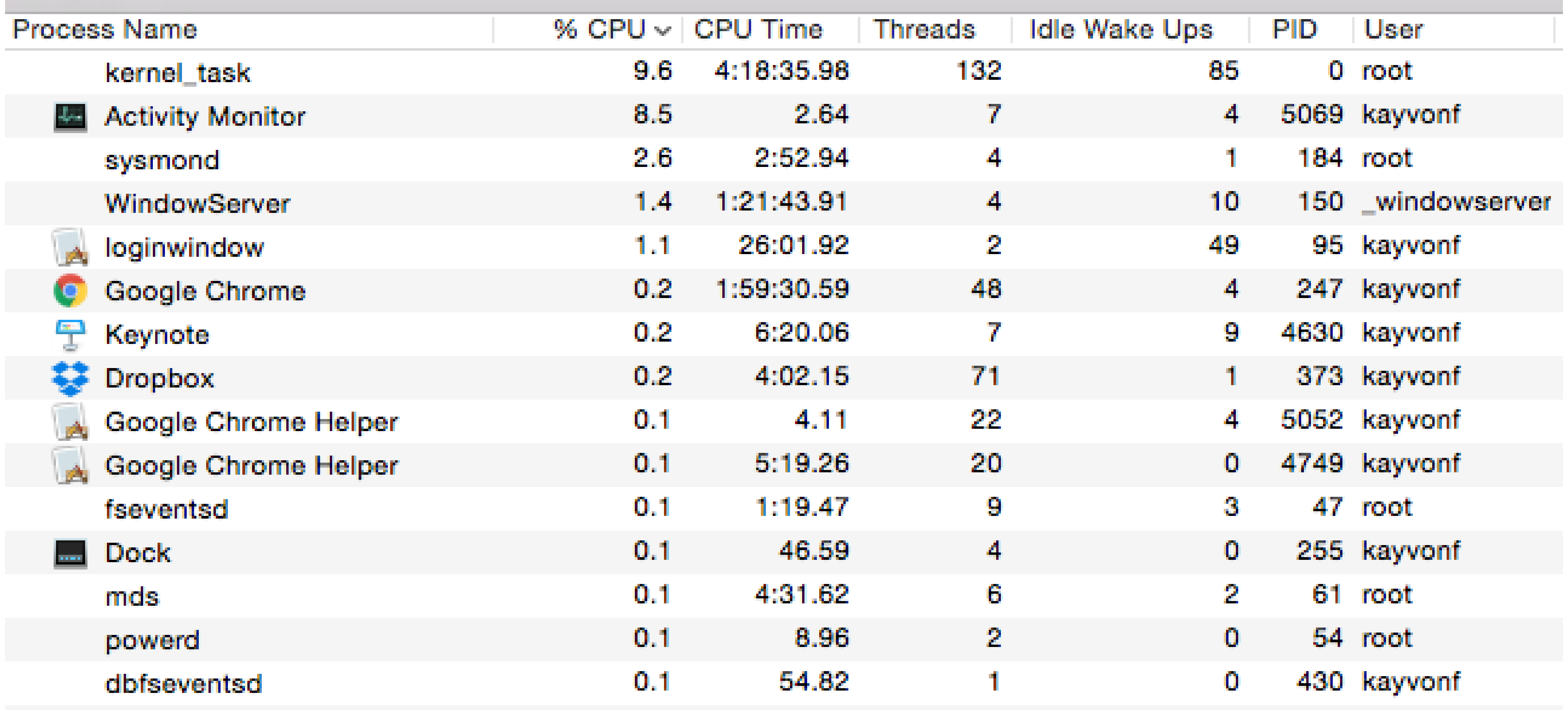
Let's say I have a processor with 4 cores, with support for 2 execution contexts per core.

In each clock, each core executes one instruction (from one execution context)



We can run many programs on a computer concurrently

For example, let’s take a look at what’s running on a Mac.



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
kernel_task	9.6	4:18:35.98	132	85	0	root
Activity Monitor	8.5	2.64	7	4	5069	kayvonf
sysmond	2.6	2:52.94	4	1	184	root
WindowServer	1.4	1:21:43.91	4	10	150	_windowserver
loginwindow	1.1	26:01.92	2	49	95	kayvonf
Google Chrome	0.2	1:59:30.59	48	4	247	kayvonf
Keynote	0.2	6:20.06	7	9	4630	kayvonf
Dropbox	0.2	4:02.15	71	1	373	kayvonf
Google Chrome Helper	0.1	4.11	22	4	5052	kayvonf
Google Chrome Helper	0.1	5:19.26	20	0	4749	kayvonf
fseventsd	0.1	1:19.47	9	3	47	root
Dock	0.1	46.59	4	0	255	kayvonf
mds	0.1	4:31.62	6	2	61	root
powerd	0.1	8.96	2	0	54	root
dbfseventsd	0.1	54.82	1	0	430	kayvonf

Many processes, many of which has spawned many logical threads.

Many more logical threads than cores (and more threads than HW execution contexts)

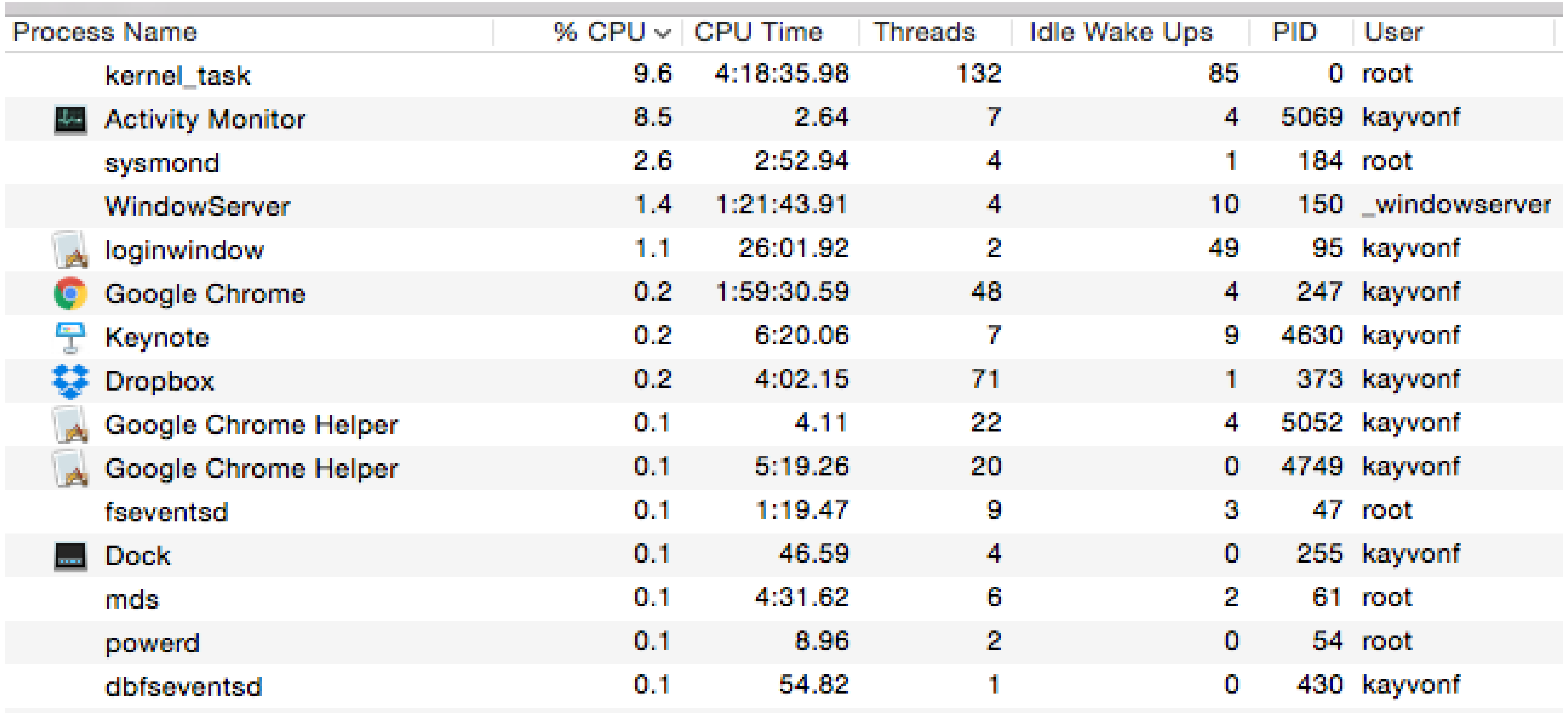
Who is responsible for choosing what threads execute on the processor?

What does running one thread entail?

- **A processor runs a logical thread by executing its instructions within a hardware execution context.**
- **If the operating system wants thread T of process P to run, it:**

1. **Chooses a CPU execution context**
2. **It sets the register values in that context to the last state of the thread (e.g., sets PC to point to next instruction the thread must run, sets stack pointer, VM mappings, etc.)**
3. **Then the processor starts running… It grabs the next instruction according to the PC, and executes it:**
    - **If the instruction is:** `add r0, r1, r2;` **then the processor adds the contexts of r1 and r2 and stores the result in r0**
    - **If the instruction is:** `ld r0 mem[r1];` **then the processor takes contents of r1, translates it to a physical address according to the page tables referenced by the execution context, and loads the value at that address into r0**
    - **Etc…**

The operating system maps logical threads to execution contexts



Process Name	% CPU ⌄	CPU Time	Threads	Idle Wake Ups	PID	User
kernel_task	9.6	4:18:35.98	132	85	0	root
Activity Monitor	8.5	2.64	7	4	5069	kayvonf
sysmond	2.6	2:52.94	4	1	184	root
WindowServer	1.4	1:21:43.91	4	10	150	_windowserver
loginwindow	1.1	26:01.92	2	49	95	kayvonf
Google Chrome	0.2	1:59:30.59	48	4	247	kayvonf
Keynote	0.2	6:20.06	7	9	4630	kayvonf
Dropbox	0.2	4:02.15	71	1	373	kayvonf
Google Chrome Helper	0.1	4.11	22	4	5052	kayvonf
Google Chrome Helper	0.1	5:19.26	20	0	4749	kayvonf
fseventsd	0.1	1:19.47	9	3	47	root
Dock	0.1	46.59	4	0	255	kayvonf
mds	0.1	4:31.62	6	2	61	root
powerd	0.1	8.96	2	0	54	root
dbfseventsd	0.1	54.82	1	0	430	kayvonf

Since there are more threads than execution contexts, the operating system must interleave execution of threads on the processor

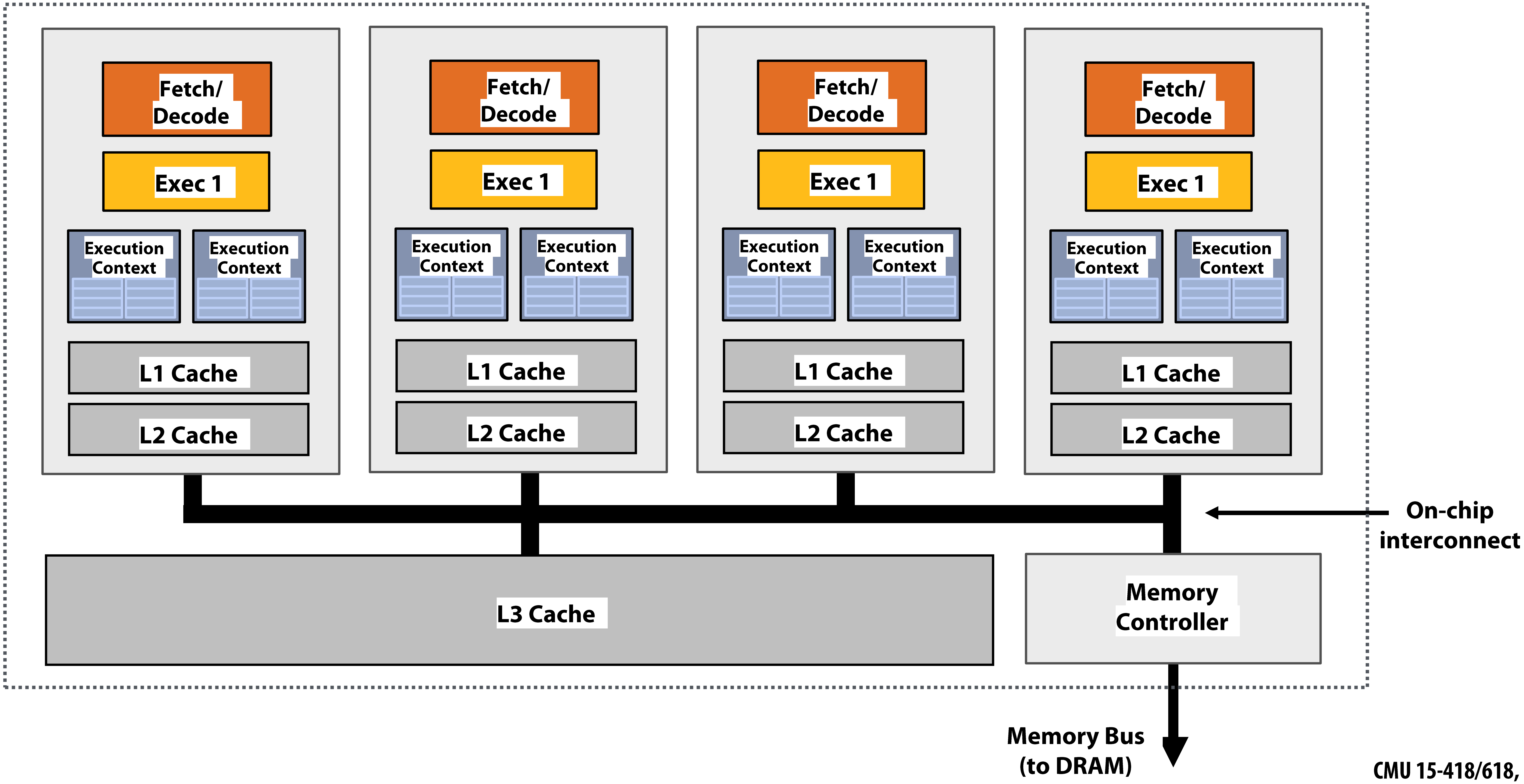
Periodically… the OS will:

1. Interrupts the processor
2. Copies the register state of threads currently mapped to execution contexts to OS data structures in memory
3. Copies the register state of other threads it now wants to run onto the processors execution context registers
4. Tell the processor to continue
   - Now these logical threads are running on the processor

But how do 2 execution contexts run on a core that can only run one instruction per clock?

It is the responsibility of the processor (without OS intervention) to choose how to interleave execution of instructions from multiple execution contexts on the resources of a single core.

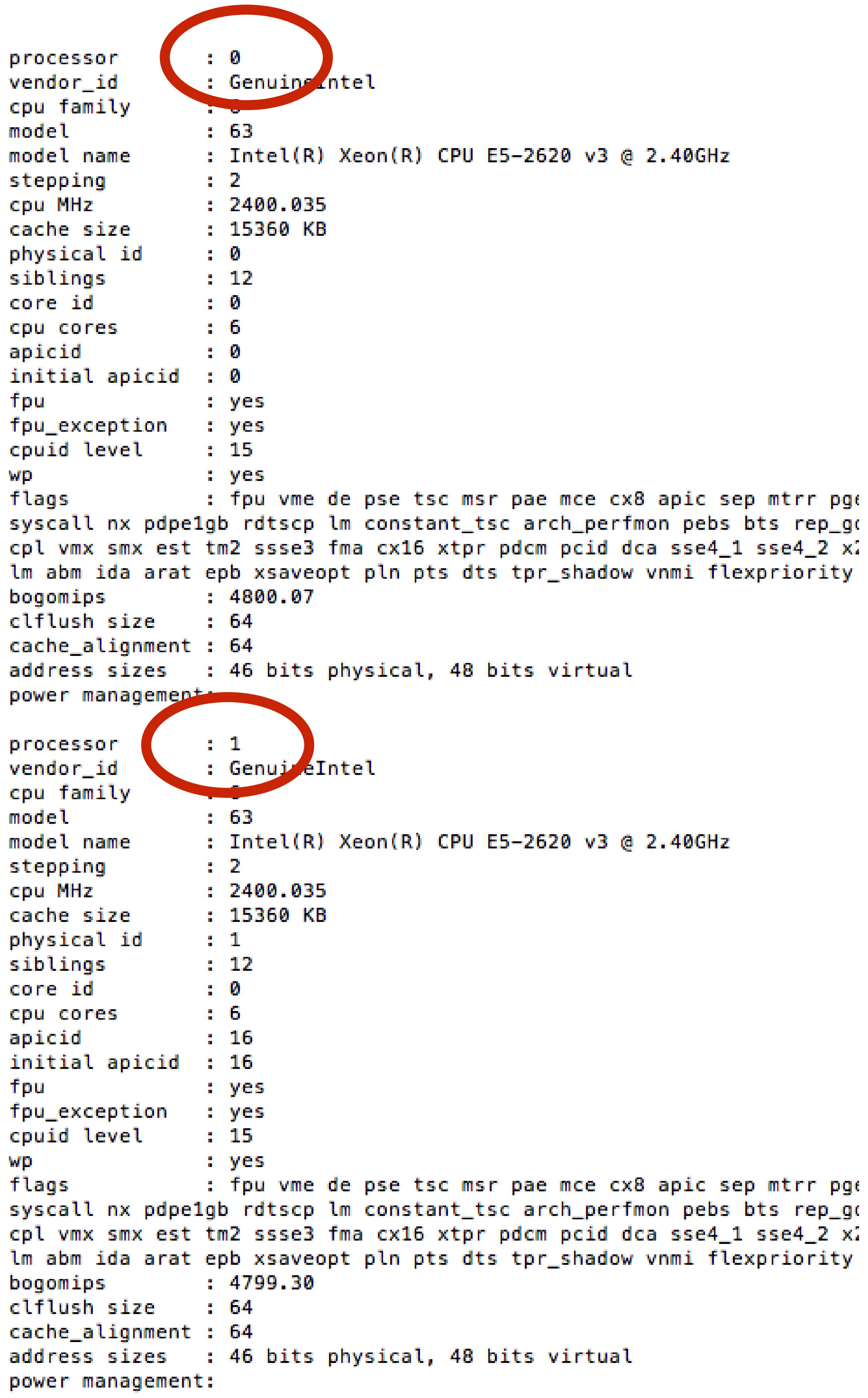
This is the idea of hardware multi-threading from Lecture 2.



Output of ‘less /proc/cpuinfo’ on latedays

- Dual CPU (two socket)
- Six-cores per CPU, two threads per core
- Linux has 24 execution contexts to fill

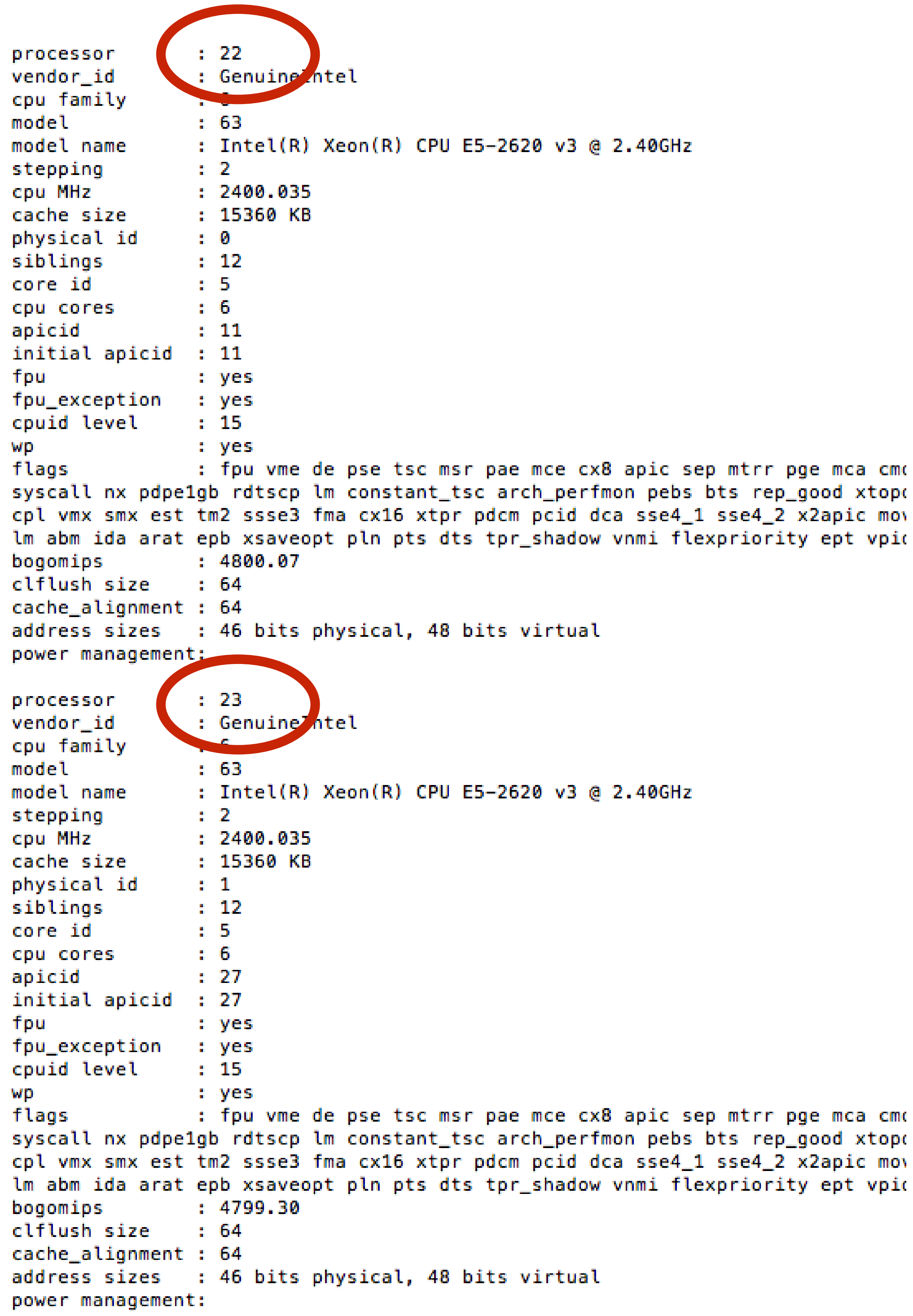
Linux reports it is running on a machine with 24 “logical processors” (corresponding to the 24 execution contexts available on the machine)



```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
stepping        : 2
cpu MHz         : 2400.035
cache size      : 15360 KB
physical id     : 0
siblings        : 12
core id         : 0
cpu cores       : 6
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pg
syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_g
cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x
lm abm ida arat epb xsaveopt pln pts dts tpr_shadow vnmi flexpriority
bogomips        : 4800.07
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
stepping        : 2
cpu MHz         : 2400.035
cache size      : 15360 KB
physical id     : 1
siblings        : 12
core id         : 0
cpu cores       : 6
apicid          : 16
initial apicid  : 16
fpu             : yes
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pg
syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_g
cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x
lm abm ida arat epb xsaveopt pln pts dts tpr_shadow vnmi flexpriority
bogomips        : 4799.30
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

. . .



```
processor       : 22
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
stepping        : 2
cpu MHz         : 2400.035
cache size      : 15360 KB
physical id     : 0
siblings        : 12
core id         : 5
cpu cores       : 6
apicid          : 11
initial apicid  : 11
fpu             : yes
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cm
syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good xtop
cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic mo
lm abm ida arat epb xsaveopt pln pts dts tpr_shadow vnmi flexpriority ept vpi
bogomips        : 4800.07
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:

processor       : 23
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
stepping        : 2
cpu MHz         : 2400.035
cache size      : 15360 KB
physical id     : 1
siblings        : 12
core id         : 5
cpu cores       : 6
apicid          : 27
initial apicid  : 27
fpu             : yes
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cm
syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good xtop
cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic mo
lm abm ida arat epb xsaveopt pln pts dts tpr_shadow vnmi flexpriority ept vpi
bogomips        : 4799.30
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

Today’s topic: efficiently implementing synchronization primitives

- **Primitives for ensuring mutual exclusion**
  - **Locks**
  - **Atomic primitives (e.g., atomic_add)**
  - **Transactions (later in the course)**

- **Primitives for event signaling**
  - **Barriers**
  - **Flags**

Three phases of a synchronization event

1. **Acquire method**
   - How a thread attempts to gain access to protected resource
2. **Waiting algorithm**
   - How a thread waits for access to be granted to shared resource
3. **Release method**
   - How thread enables other threads to gain resource when its work in the synchronized region is complete

Busy waiting

- **Busy waiting (a.k.a. “spinning”)**

```
while (condition X not true) {}
logic that assumes X is true
```

- **In classes like 15-213 or in operating systems, you have certainly also talked about synchronization**
  - **You might have been taught busy-waiting is bad: why?**

“Blocking” synchronization

- **Idea: if progress cannot be made because a resource cannot be acquired, it is desirable to free up execution resources for another thread (preempt the running thread)**

```
if (condition X not true)
   block until true;  // OS scheduler de-schedules thread
                      // (let’s another thread use the processor)
```

- **pthreads mutex example**

```
pthread_mutex_t mutex;
pthread_mutex_lock(&mutex);
```

Busy waiting vs. blocking

- **Busy-waiting can be preferable to blocking if:**
  - Scheduling overhead is larger than expected wait time
  - Processor's resources not needed for other tasks
    - This is often the case in a parallel program since we usually don't oversubscribe a system when running a performance-critical parallel app (e.g., there aren't multiple CPU-intensive programs running at the same time)
    - Clarification: be careful to not confuse the above statement with the value of multi-threading (interleaving execution of multiple threads/tasks to hiding long latency of memory operations) with other work within the same app.

- **Examples:**

```
pthread_spinlock_t spin;              int lock;
pthread_spin_lock(&spin);             OSSpinLockLock(&lock);   // OSX spin lock
```

Implementing Locks

Warm up: a simple, but incorrect, lock

```
lock:        ld   R0, mem[addr]      // load word into R0
             cmp  R0, #0             // compre R0 to 0
             bnz  lock               // if nonzero jump to top
             st   mem[addr], #1


unlock:      st   mem[addr], #0      // store 0 to address
```

Problem: data race because LOAD-TEST-STORE is not atomic!

Processor 0 loads address X, observes 0

Processor 1 loads address X, observes 0

Processor 0 writes 1 to address X

Processor 1 writes 1 to address X

Test-and-set based lock

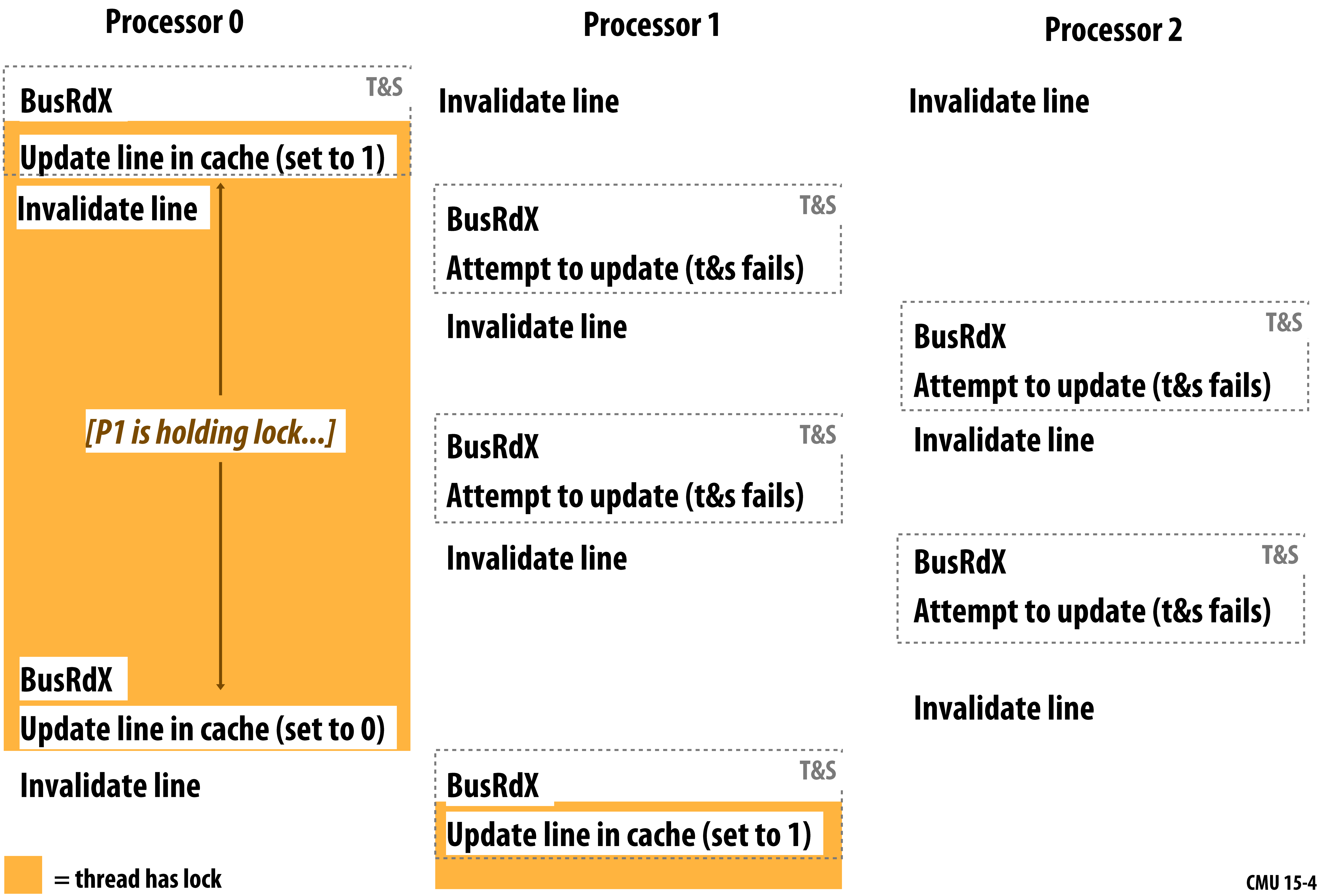
Atomic test-and-set instruction:

```
ts R0, mem[addr]       // load mem[addr] into R0
                       // if mem[addr] is 0, set mem[addr] to 1
```

```
lock:       ts   R0, mem[addr]       // load word into R0
            bnz  R0, lock            // if 0, lock obtained


unlock:     st   mem[addr], #0       // store 0 to address
```

Test-and-set lock: consider coherence traffic



Processor 0

T&S

BusRdX

Update line in cache (set to 1)

Invalidate line

[P1 is holding lock...]

BusRdX

Update line in cache (set to 0)

Invalidate line

Processor 1

Invalidate line

T&S

BusRdX

Attempt to update (t&s fails)

Invalidate line

T&S

BusRdX

Attempt to update (t&s fails)

Invalidate line

T&S

BusRdX

Update line in cache (set to 1)

Processor 2

Invalidate line

T&S

BusRdX

Attempt to update (t&s fails)

Invalidate line

T&S

BusRdX

Attempt to update (t&s fails)

Invalidate line

🟧 = thread has lock

Check your understanding

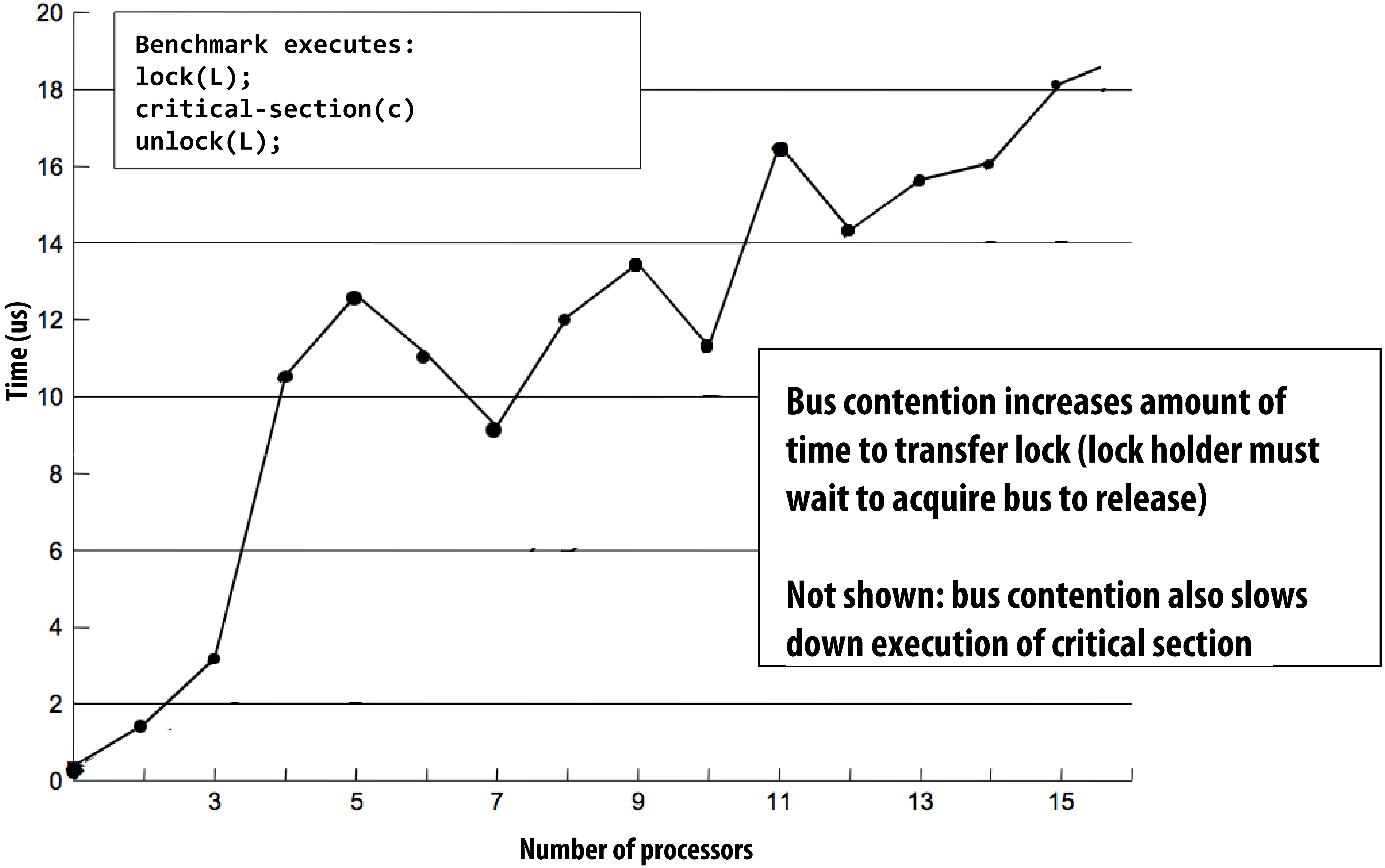
- On the previous slide, what is the duration of time the thread running on P0 holds the lock?

- At what points in time does P0's cache contain a valid copy of the cache line containing the lock variable?

Test-and-set lock performance

Benchmark: execute a total of N lock/unlock sequences (in aggregate) by P processors

Critical section time removed so graph plots only time acquiring/releasing the lock



```
Benchmark executes:
lock(L);
critical-section(c)
unlock(L);
```

Bus contention increases amount of time to transfer lock (lock holder must wait to acquire bus to release)

Not shown: bus contention also slows down execution of critical section

Figure credit: Culler, Singh, and Gupta

x86 cmpxchg

- **Compare and exchange (atomic when used with lock prefix)**

```
lock cmpxchg dst, src
```

often a memory address

lock prefix (makes operation atomic)

x86 accumulator register

```
if (dst == EAX)
    ZF = 1
    dst = src
else
    ZF = 0
    EAX = dst
```

flag register

Self-check: Can you implement ASM for atomic test-and-set using `cmpxchg`?

Desirable lock performance characteristics

- Low latency
  - If lock is free and no other processors are trying to acquire it, a processor should be able to acquire the lock quickly
- Low interconnect traffic
  - If all processors are trying to acquire lock at once, they should acquire the lock in succession with as little traffic as possible
- Scalability
  - Latency / traffic should scale reasonably with number of processors
- Low storage cost
- Fairness
  - Avoid starvation or substantial unfairness
  - One ideal: processors should acquire lock in the order they request access to it

Simple test-and-set lock: low latency (under low contention), high traffic, poor scaling, low storage cost (one int), no provisions for fairness

Test-and-test-and-set lock

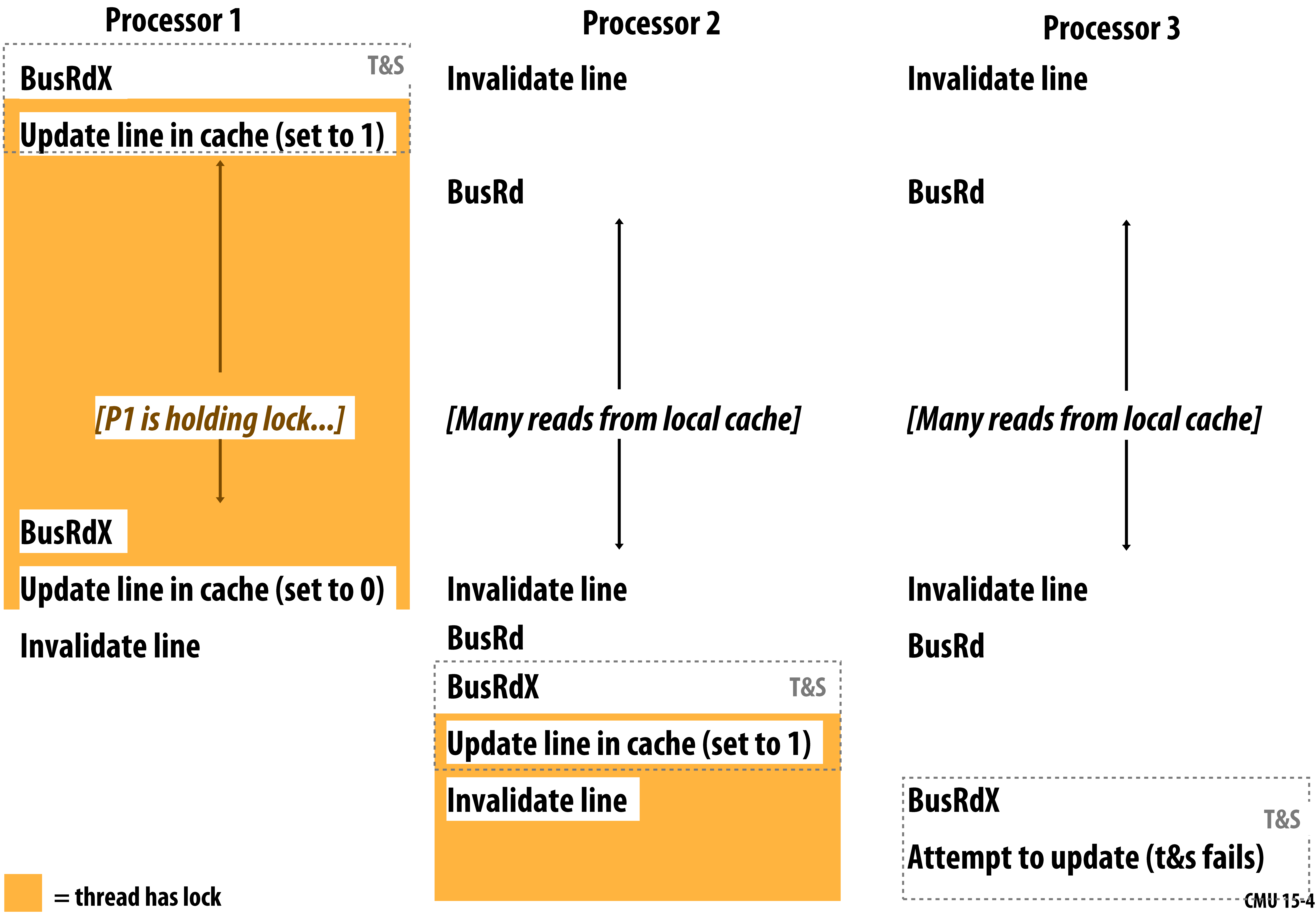
```
void Lock(int* lock) {
  while (1) {

    while (*lock != 0);            // while another processor has the lock...

    if (test_and_set(*lock) == 0)  // when lock is released, try to acquire it
      return;
  }
}

void Unlock(volatile int* lock) {
   *lock = 0;
}
```

Test-and-test-and-set lock: coherence traffic



Processor 1

BusRdX
Update line in cache (set to 1)
[P1 is holding lock...]
BusRdX
Update line in cache (set to 0)
Invalidate line

Processor 2

Invalidate line
BusRd
[Many reads from local cache]
Invalidate line
BusRd
BusRdX
Update line in cache (set to 1)
Invalidate line

Processor 3

Invalidate line
BusRd
[Many reads from local cache]
Invalidate line
BusRd
BusRdX
Attempt to update (t&s fails)

🟧 = thread has lock

Test-and-test-and-set characteristics

- Slightly higher latency than test-and-set in <u>uncontended</u> case
  - Must test... then test-and-set
- Generates much less interconnect traffic
  - One invalidation, per waiting processor, per lock release (O(P) invalidations)
  - This is $O(P^2)$ interconnect traffic if all processors have the lock cached
  - Recall: test-and-set lock generated one invalidation per waiting processor <u>per test</u>
- More scalable (due to less traffic)
- Storage cost unchanged (one int)
- Still no provisions for fairness

Test-and-set lock with back off

Upon failure to acquire lock, delay for awhile before retrying

```
void Lock(volatile int* l) {
  int amount = 1;
  while (1) {
    if (test_and_set(*l) == 0)
      return;
    delay(amount);
    amount *= 2;
  }
}
```

- Same uncontended latency as test-and-set, but potentially higher latency under contention. Why?
- Generates less traffic than test-and-set (not continually attempting to acquire lock)
- Improves scalability (due to less traffic)
- Storage cost unchanged (still one int for lock)
- Exponential back-off can cause severe unfairness
  - Newer requesters back off for shorter intervals

Ticket lock

Main problem with test-and-set style locks: upon release, all waiting processors attempt to acquire lock using test-and-set

```
struct lock {
   volatile int next_ticket;
   volatile int now_serving;
};

void Lock(lock* l) {
  int my_ticket = atomic_increment(&l->next_ticket);   // take a “ticket”
  while (my_ticket != l->now_serving);                 // wait for number
}                                                      // to be called

void unlock(lock* l) {
  l->now_serving++;
}
```

No atomic operation needed to acquire the lock (only a read)

Result: only one invalidation per lock release (O(P) interconnect traffic)

Array-based lock

Each processor spins on a different memory address

Utilizes atomic operation to assign address on attempt to acquire

```
struct lock {
   volatile padded_int status[P];    // padded to keep off same cache line
   volatile int head;
};

int my_element;

void Lock(lock* l) {
  my_element = atomic_circ_increment(&l->head);    // assume circular increment
  while (l->status[my_element] == 1);
}

void unlock(lock* l) {
  l->status[my_element] = 1;
  l->status[circ_next(my_element)] = 0;            // next() gives next index
}
```

O(1) interconnect traffic per release, but lock requires space linear in P

Also, the atomic circular increment is a more complex operation (higher overhead)

Queue-based Lock (MCS lock)

- Create a queue of waiters
  - Each thread allocates a local space on which to wait
- Pseudo-code:
  - glock – global lock
  - mlock – my lock (state, next pointer)

```
AcquireQLock(*glock, *mlock)
{
  mlock->next = NULL;
  mlock->state = UNLOCKED;
  ATOMIC();
  prev = glock
  *glock = mlock
  END_ATOMIC();
  if (prev == NULL) return;
  mlock->state = LOCKED;
  prev->next = mlock;
  while (mlock->state == LOCKED)
    ; // SPIN
}
```

```
ReleaseQLock(*glock, *mlock)
{
  do {
    if (mlock->next == NULL) {
      x = CMPXCHG(glock, mlock, NULL);
      if (x == mlock) return;
    }
    else
    {
      mlock->next->state = UNLOCKED;
      return;
    }
  } while (1);
}
```

Implementing Barriers

Implementing a centralized barrier

(Based on shared counter)

```
struct Barrier_t {
  LOCK lock;
  int counter;   // initialize to 0
  int flag;      // the flag field should probably be padded to
                 // sit on its own cache line. Why?
};

// barrier for p processors
void Barrier(Barrier_t* b, int p) {
  lock(b->lock);
  if (b->counter == 0) {
    b->flag = 0;       // first thread arriving at barrier clears flag
  }
  int num_arrived = ++(b->counter);
  unlock(b->lock);

  if (num_arrived == p) {  // last arriver sets flag
    b->counter = 0;
    b->flag = 1;
  }
  else {
    while (b->flag == 0);  // wait for flag
  }
}
```

Does it work? Consider:

```
do stuff ...
Barrier(b, P);
do more stuff ...
Barrier(b, P);
```

Correct centralized barrier

```
struct Barrier_t {
  LOCK lock;
  int arrive_counter;   // initialize to 0 (number of threads that have arrived)
  int leave_counter;    // initialize to P (number of threads that have left barrier)
  int flag;
};

// barrier for p processors
void Barrier(Barrier_t* b, int p) {
  lock(b->lock);
  if (b->arrive_counter == 0) {   // if first to arrive...
    if (b->leave_counter == P) {  // check to make sure no other threads “still in barrier”
       b->flag = 0;               // first arriving thread clears flag
    } else {
      unlock(lock);
      while (b->leave_counter != P);  // wait for all threads to leave before clearing
      lock(lock);
      b->flag = 0;                // first arriving thread clears flag
    }
  }
  int num_arrived = ++(b->arrive_counter);
  unlock(b->lock);

  if (num_arrived == p) {  // last arriver sets flag
    b->arrive_counter = 0;
    b->leave_counter = 1;
    b->flag = 1;
  }
  else {
    while (b->flag == 0);  // wait for flag
    lock(b->lock);
    b->leave_counter++;
    unlock(b->lock);
  }
}
```

Main idea: wait for all processes to leave first barrier, before clearing flag for entry into the second

Centralized barrier with sense reversal

```
struct Barrier_t {
  LOCK lock;
  int  counter;   // initialize to 0
  int  flag;      // initialize to 0
};

int local_sense = 0;  // private per processor. Main idea: processors wait for flag
                      // to be equal to local sense

// barrier for p processors
void Barrier(Barrier_t* b, int p) {
  local_sense = (local_sense == 0) ? 1 : 0;
  lock(b->lock);
  int num_arrived = ++(b->counter);
  if (b->counter == p) {  // last arriver sets flag
    unlock(b->lock);
    b->counter = 0;
    b->flag = local_sense;
  }
  else {
    unlock(b->lock);
    while (b.flag != local_sense);  // wait for flag
  }
```

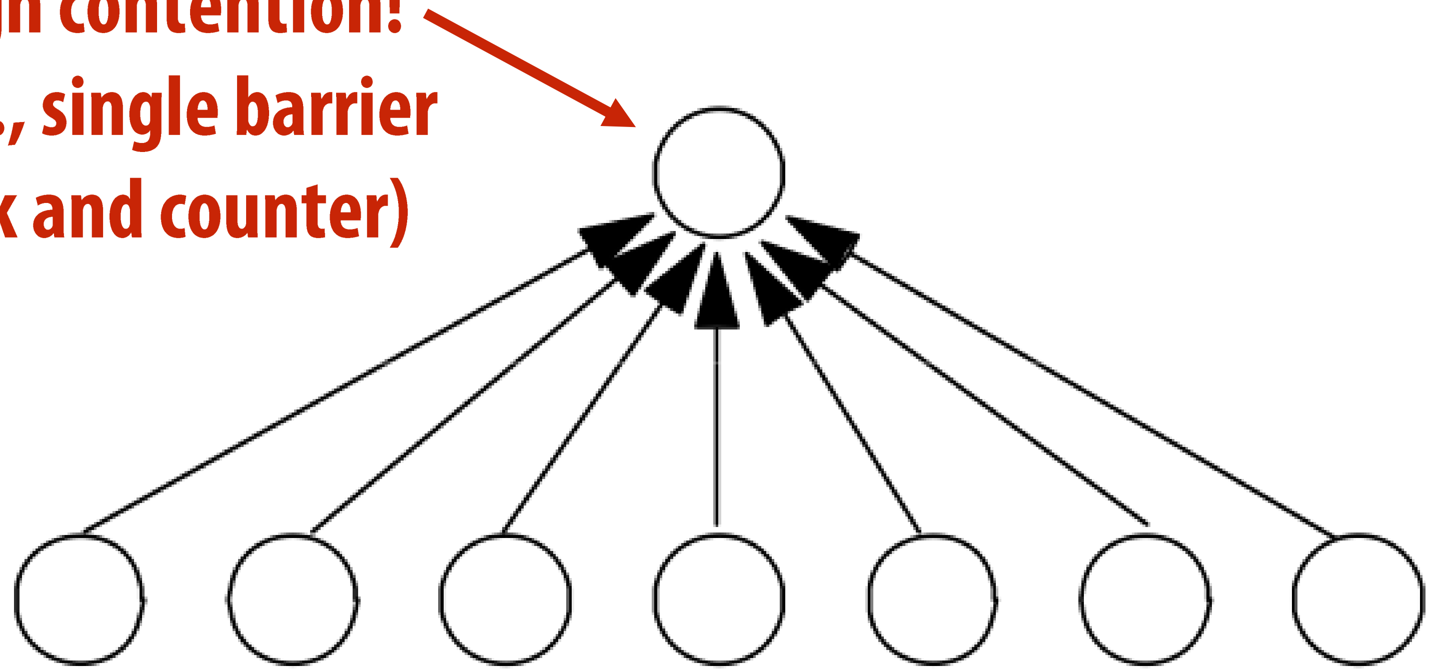
Sense reversal optimization results in one spin instead of two

Centralized barrier: traffic

- **O(P) traffic on interconnect per barrier:**
  - All threads: 2P write transactions to obtain barrier lock and update counter (O(P) traffic assuming lock acquisition is implemented in O(1) manner)
  - Last thread: 2 write transactions to write to the flag and reset the counter (O(P) traffic since there are many sharers of the flag)
  - P-1 transactions to read updated flag

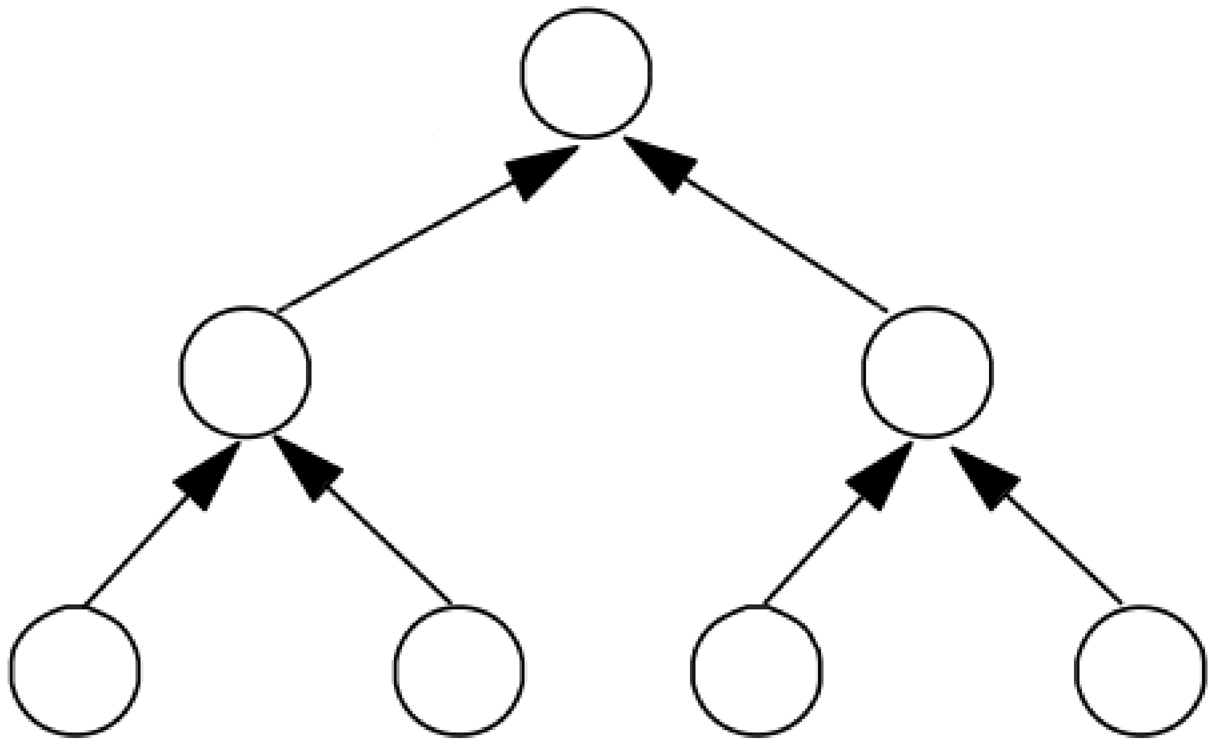
- **But there is still serialization on a single shared lock**
  - So span (latency) of entire operation is O(P)
  - Can we do better?

Combining tree implementation of barrier



High contention!
(e.g., single barrier
lock and counter)

Centralized Barrier



Combining Tree Barrier

- Combining trees make better use of parallelism in interconnect topologies
  - lg(P) span (latency)
  - Strategy makes less sense on a bus (all traffic still serialized on single shared bus)
- Barrier acquire: when processor arrives at barrier, performs increment of parent counter
  - Process recurses to root
- Barrier release: beginning from root, notify children of release

Coming up…

- Imagine you have a shared variable for which contention is low. So it is unlikely that two processors will enter the critical section at the same time?
- You could hope for the best, and avoid the overhead of taking the lock since it is likely that mechanisms for ensuring mutual exclusion are not needed for correctness
    - Take a "optimize-for-the-common-case" attitude
- What happens if you take this approach and you're wrong: in the middle of the critical region, another process enters the same region?

Preview: transactional memory

```
atomic
{   // begin transaction

   perform atomic computation here ...

}   // end transaction
```

Instead of ensuring mutual exclusion via locks, system will proceed as if no synchronization was necessary. (it speculates!)

System provides hardware/software support for "rolling back" all loads and stores in the critical region if it detects (at run-time) that another thread has entered same region at the same time.