

CS2200

Systems and Networks

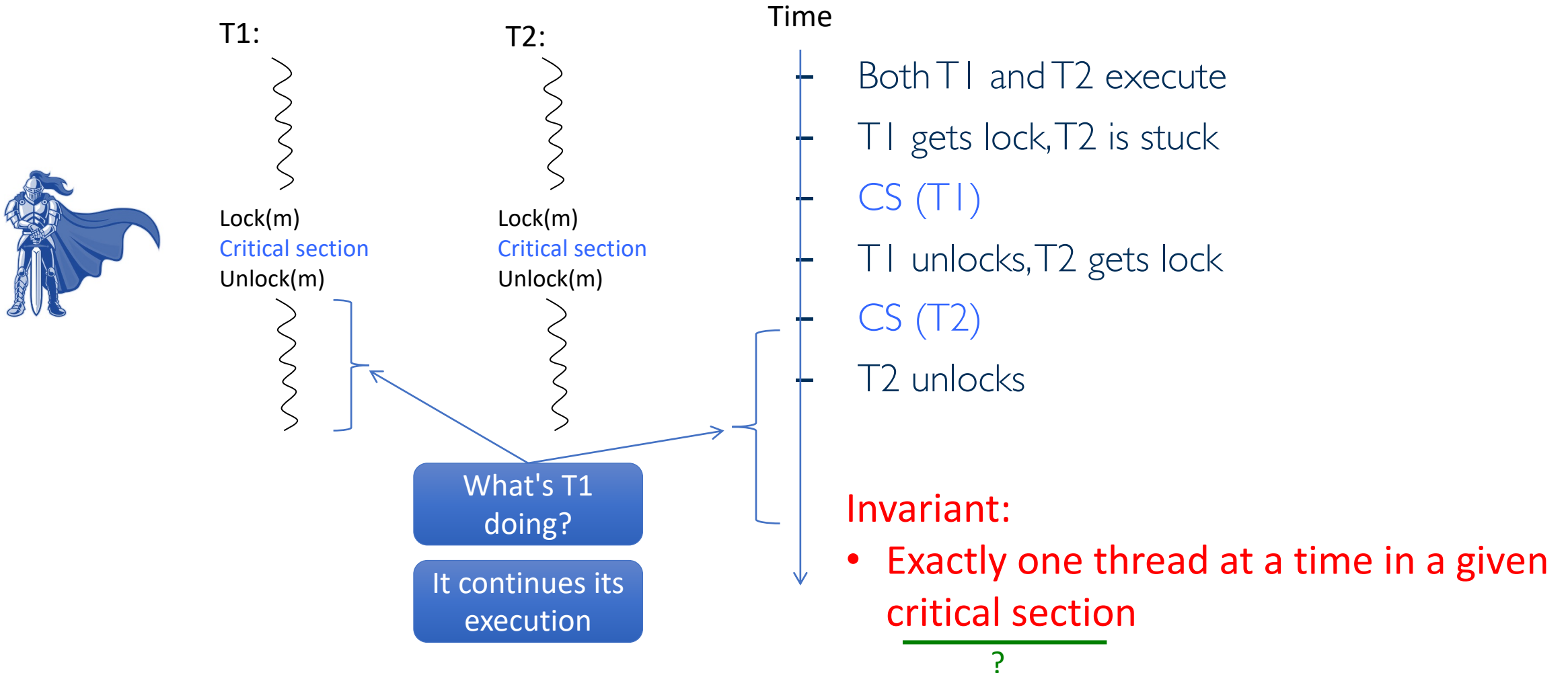
Spring 2022

Lecture 22:

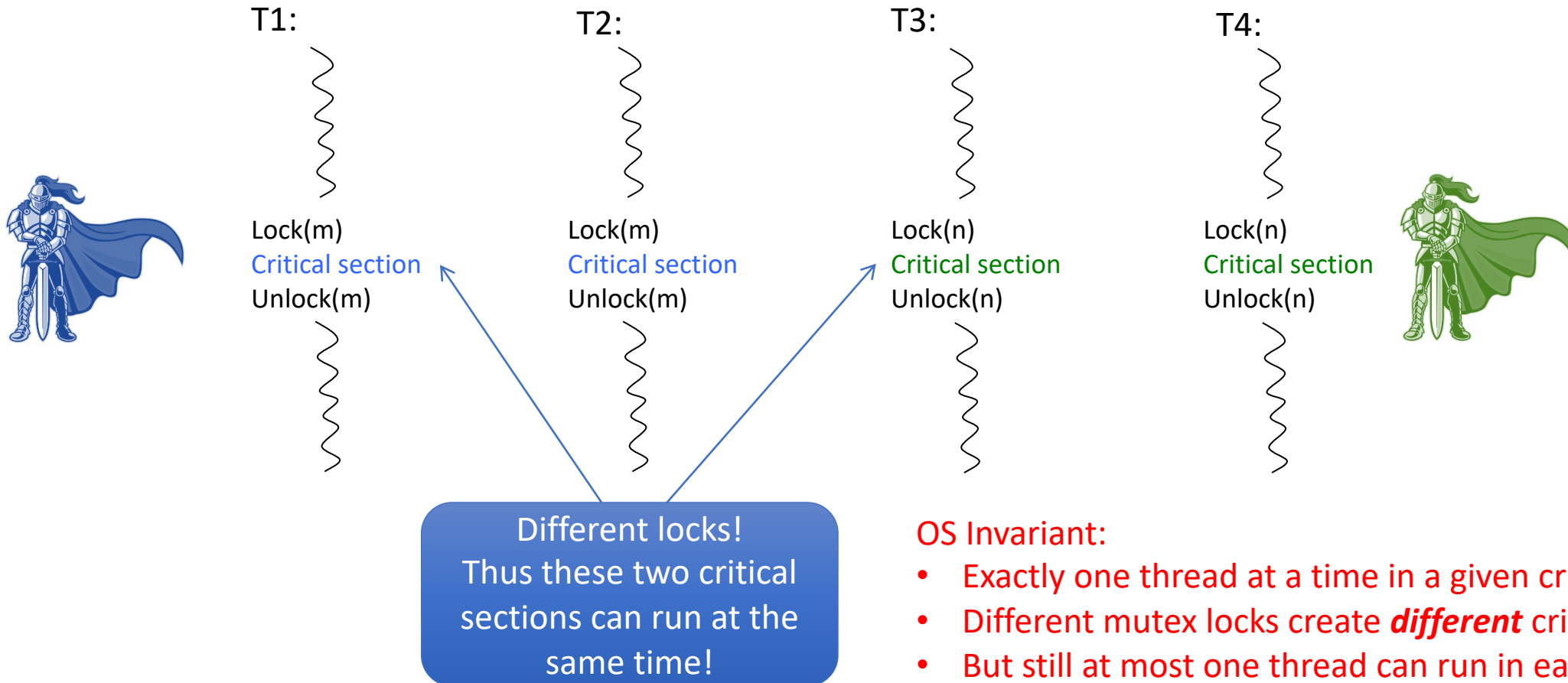
Parallel Systems pt 2

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

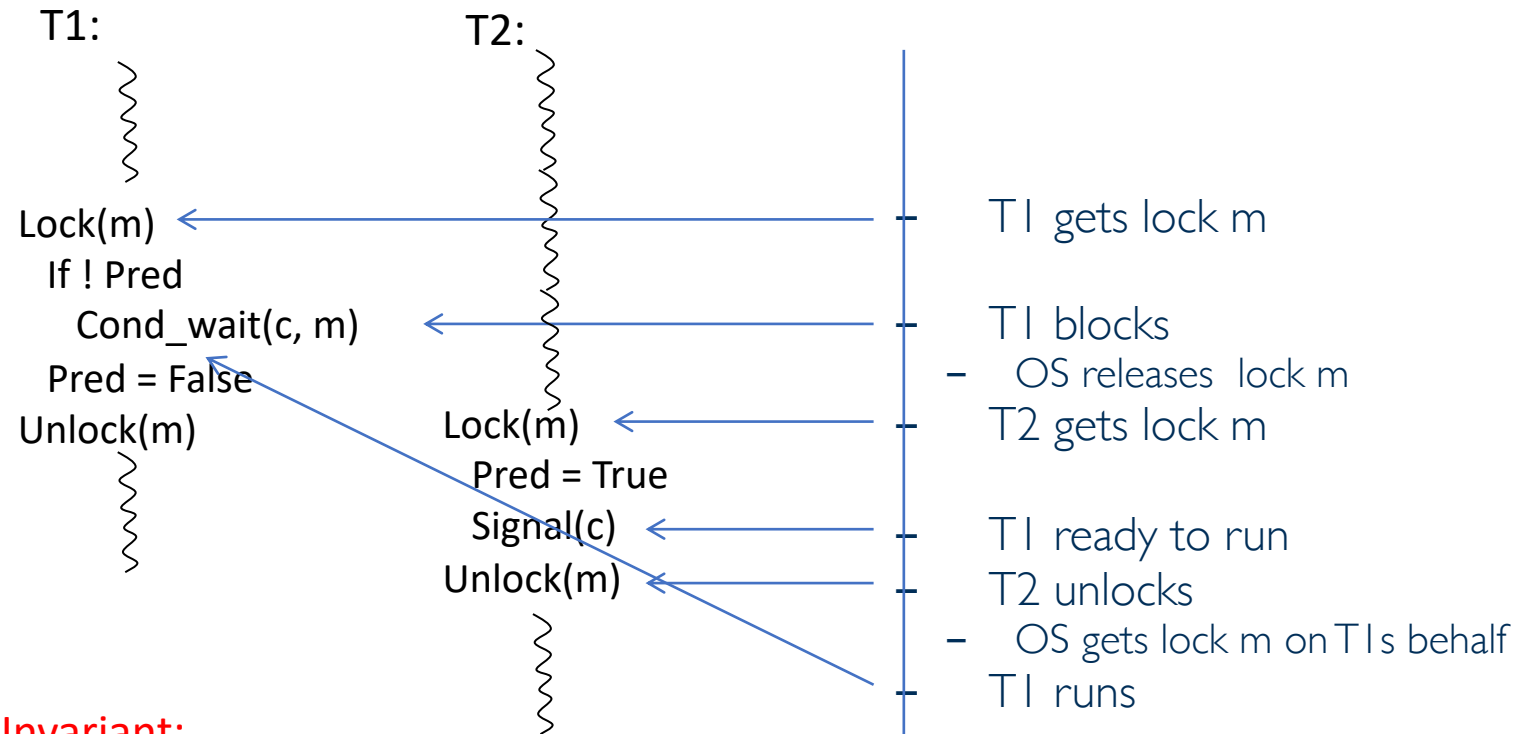
Recall: Mutex locks



More than one critical section?



Recall: Condition variables



Invariant:

- OS ensures T1 gets lock m back
- Anything else?
 - **Pred has to be True**

Who must ensure this?

The programmer!

Fix #4 – cond_var

```
cond_var buf_not_full, buf_not_empty;
```

```
digitizer()
```

```
{
```

```
    image_type dig_image;
```

```
    int tail = 0;
```

```
    loop {
```

```
        grab(dig_image);
```

```
        thread_mutex_lock(buflock);
```

```
        if (bufavail == 0)
```

```
            thread_cond_wait(buf_not_full, buflock);
```

```
        thread_mutex_unlock(buflock);
```

```
        frame_buf[tail] = dig_image;
```

```
        tail = (tail + 1) % MAX;
```

```
        thread_mutex_lock(buflock);
```

```
        bufavail = bufavail - 1;
```

```
        thread_cond_signal(buf_not_empty);
```

```
        thread_mutex_unlock(buflock);
```

```
}
```

Invariants:

Only 1 thread at a time in CS

bufavail != 0

```
tracker()
```

```
{
```

```
    image_type track_image;
```

```
    int head = 0;
```

```
    loop {
```

```
        thread_mutex_lock(buflock);
```

```
        if (bufavail == MAX)
```

```
            thread_cond_wait(buf_not_empty, buflock);
```

```
        thread_mutex_unlock(buflock);
```

```
        track_image = frame_buf[head];
```

```
        head = (head + 1) % MAX;
```

```
        thread_mutex_lock(buflock);
```

```
        bufavail = bufavail + 1;
```

```
        thread_cond_signal(buf_not_full);
```

```
        thread_mutex_unlock(buflock);
```

```
        analyze(track_image);
```

```
}
```

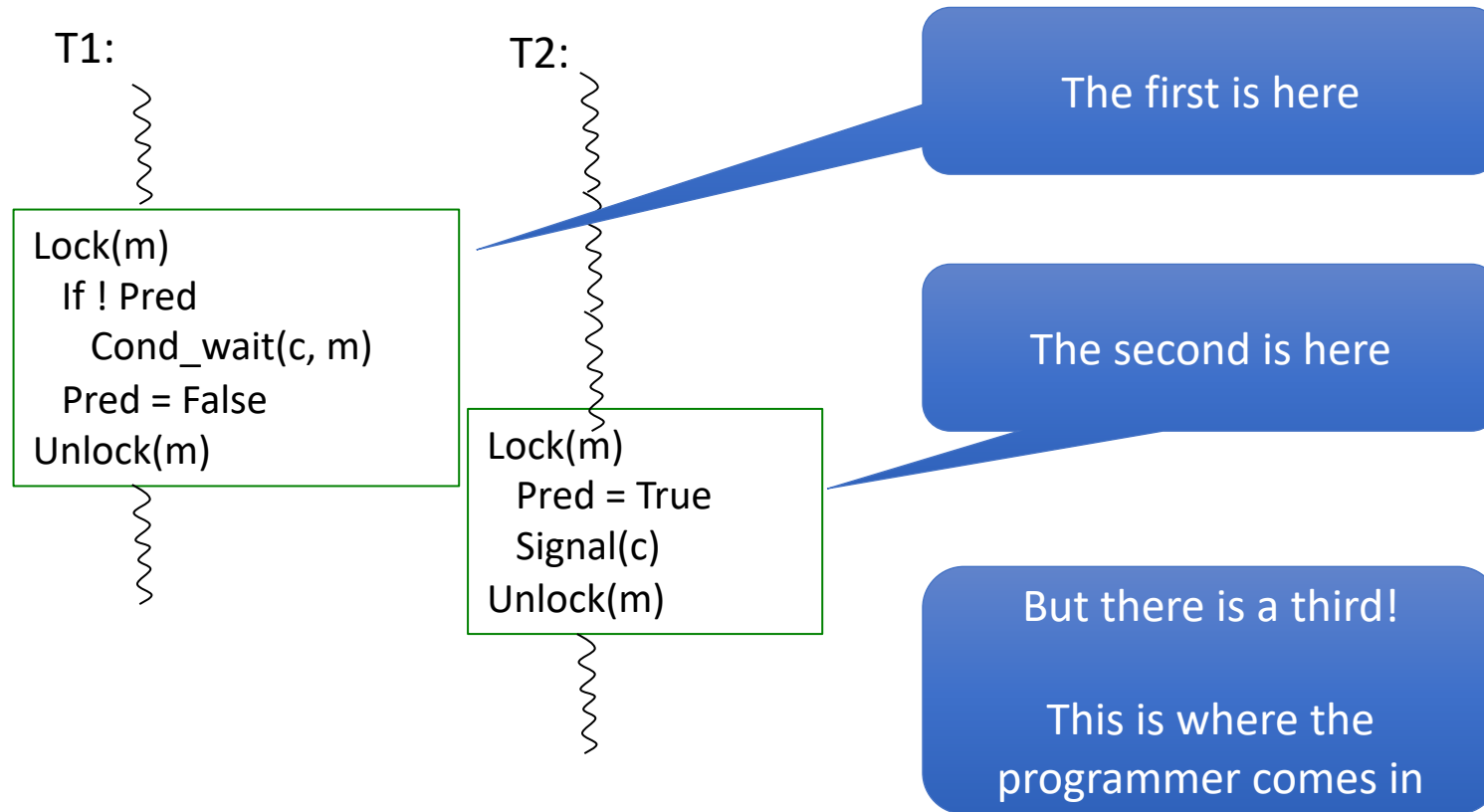
Only 1 thread at a time in CS

bufavail != MAX

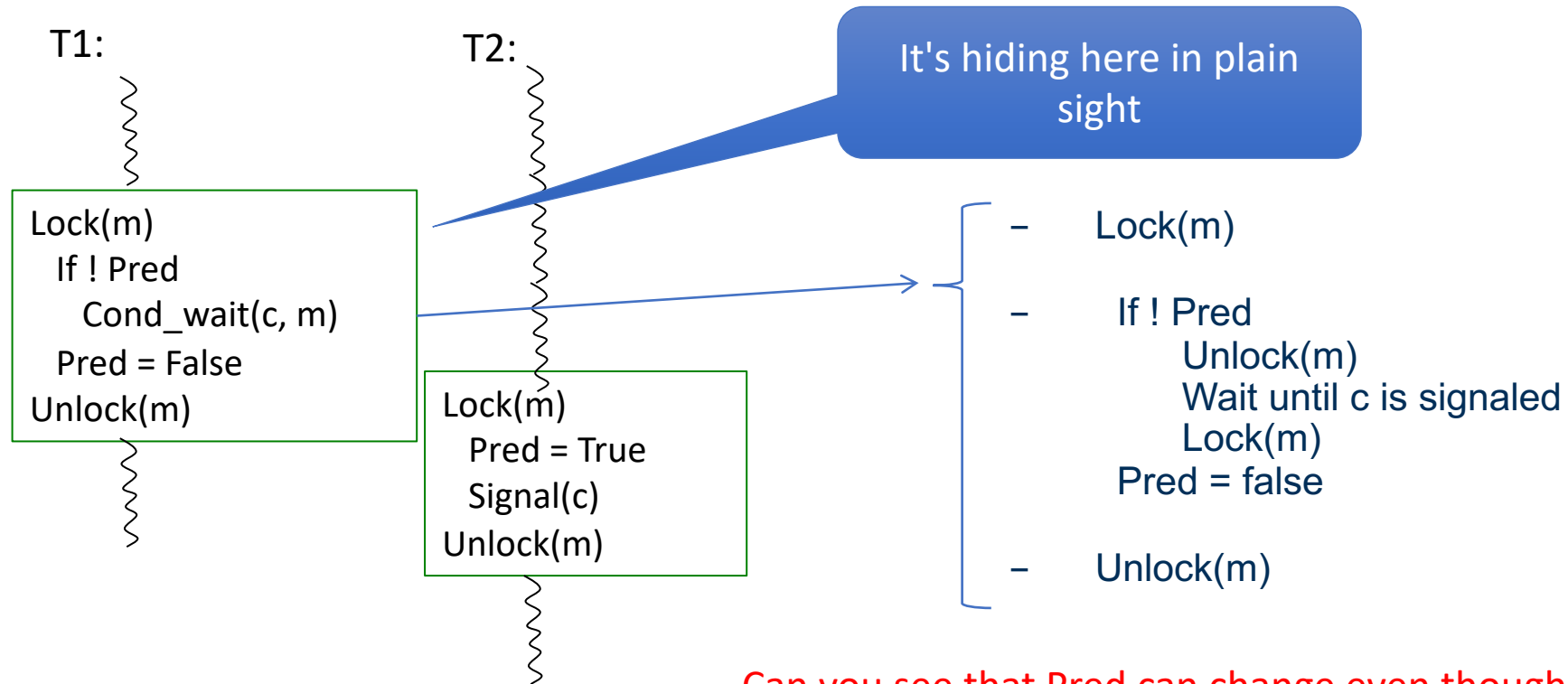
Ensured by OS

Programmer must ensure

Just how many code blocks are in the critical section here?



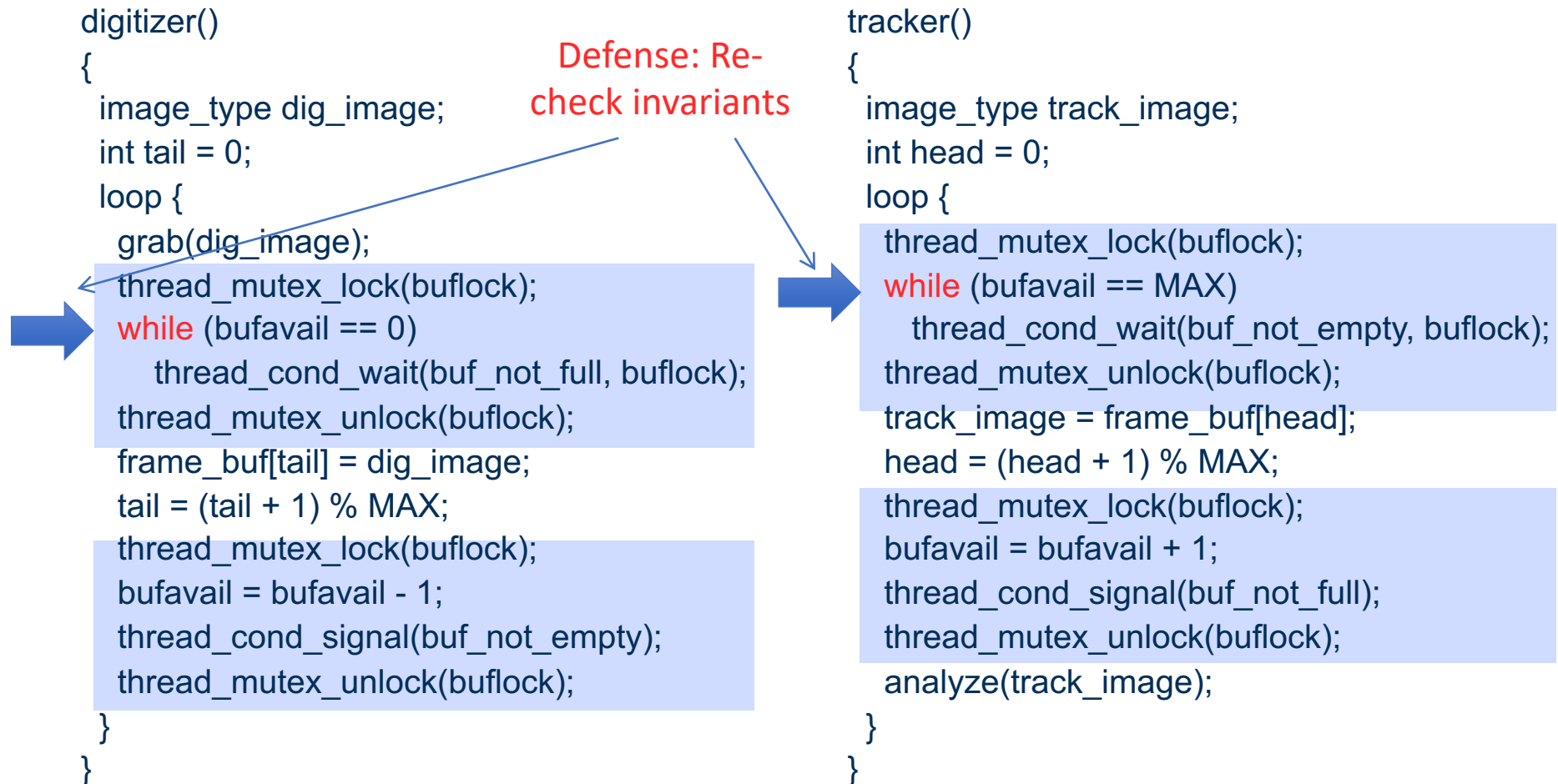
Where is that third block?



Can you see that Pred can change even though this block appears to be a single critical section?

This detail becomes a problem if there are more threads...

Fix # 5 – Defensive programming



The notion of re-checking a flag after waiting is important.



A condition variable...

40% A. I just want the participation credit

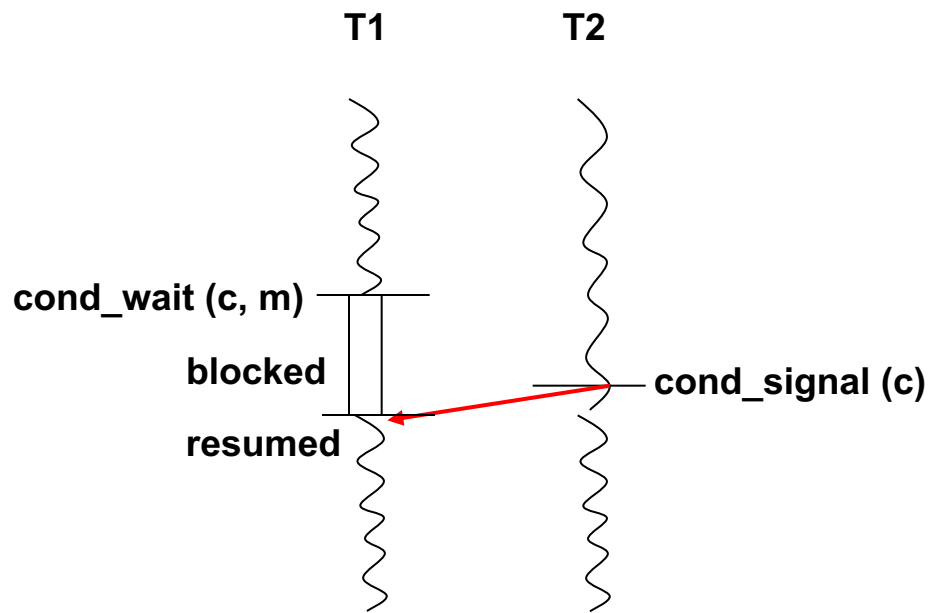
B. ... is just another name for a mutex lock

C. ...enables a thread to wait for a condition to become true without consuming processor cycles

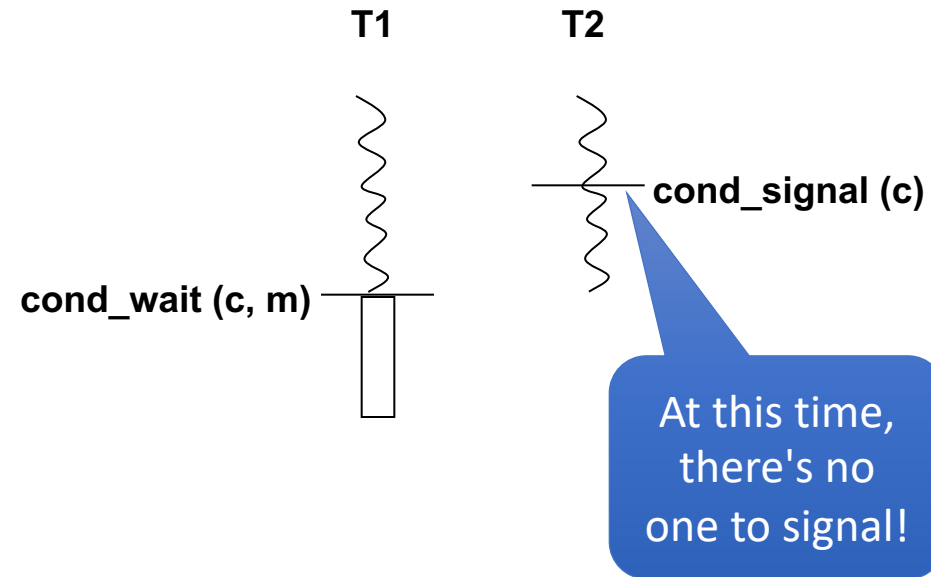
D. ...enables a thread to enter a critical section

E. ...none of the above (B to D)

Gotchas in programming with cond vars



(a) Wait before signal



(b) Wait after signal (T1 blocked forever)

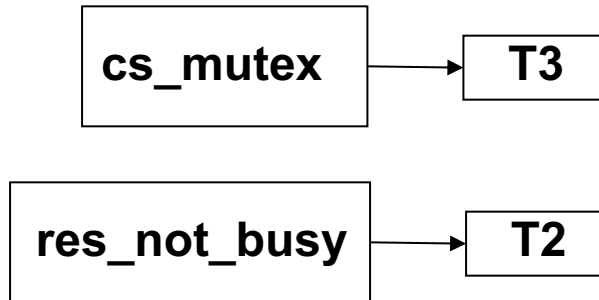
Gotchas in programming with cond vars

Say we have three threads that want to share a resource, perhaps a printer...

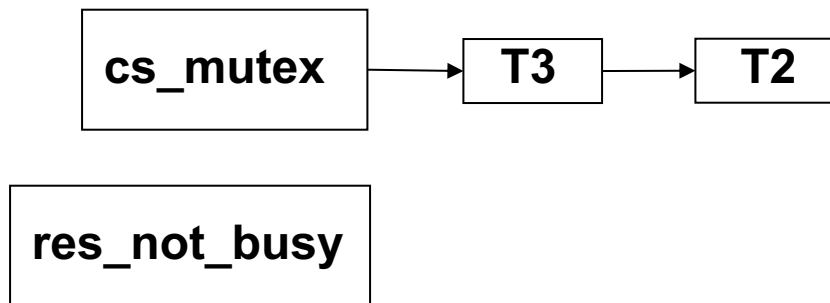
```
acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex); ← T3 is here
    if (res_state == BUSY)
        thread_cond_wait(res_not_busy, cs_mutex); ← T2 is here
    res_state = BUSY;
    thread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;
    thread_cond_signal(res_not_busy); ← T1 is here
    thread_mutex_unlock(cs_mutex);
}
```

State of waiting queues



(a) Waiting queues before T1 signals



(b) Waiting queues after T1 signals

```
acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex); ← T3 is here
    if (res_state == BUSY)
        thread_cond_wait(res_not_busy, cs_mutex); ← T2 is here
    res_state = BUSY;
    thread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;
    thread_cond_signal(res_not_busy); ← T1 is here
    thread_mutex_unlock(cs_mutex);
}
```

Gotchas -- what could go wrong?

T1 signals and unlocks
mutex

What if T3 wakes up and
locks the mutex?

T3 sets res_state to BUSY,
unlocks the mutex, and goes
off to use the resource

T2 then locks the mutex

T2 has already tested res_state, so it
unlocks the mutex and goes off to use
the resource(!)

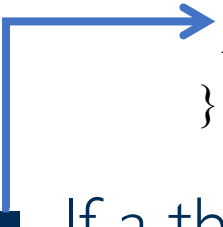
```
acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex); ← T3 is here
    if (res_state == BUSY)
        thread_cond_wait(res_not_busy, cs_mutex); ← T2 is here
    res_state = BUSY;
    thread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;
    thread_cond_signal(res_not_busy); ← T1 is here
    thread_mutex_unlock(cs_mutex);
}
```

Why did this happen?

- We violated invariants...

```
acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    if (res_state == BUSY)
        thread_cond_wait(res_not_busy, cs_mutex);
    res_state = BUSY;
    thread_mutex_unlock(cs_mutex);
}
```



- If a thread is here, what are the invariants?
 - The thread holds the mutex → the OS ensures that
 - `res_state == NOT_BUSY` → the programmer ensures that

Gotchas in programming - 3

- There's yet another surprise...
 - It's possible to have a spurious wake-up of threads by the OS
 - Even without a signal, a thread may be woken up
 - Documented behavior in Linux
 - Turns out to be very hard to avoid this in the kernel
 - Solution: Defensive programming

Gotchas— retest predicate

```
acquire_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    while (res_state == BUSY)
        thread_cond_wait(res_not_busy, cs_mutex);
    res_state = BUSY;
    thread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
    thread_mutex_lock(cs_mutex);
    res_state = NOT_BUSY;
    thread_cond_signal(res_not_busy);
    thread_mutex_unlock(cs_mutex);
}
```

Replace the
"if" with a
"while"

Make T2 recheck predicate

Avoids the "race condition"

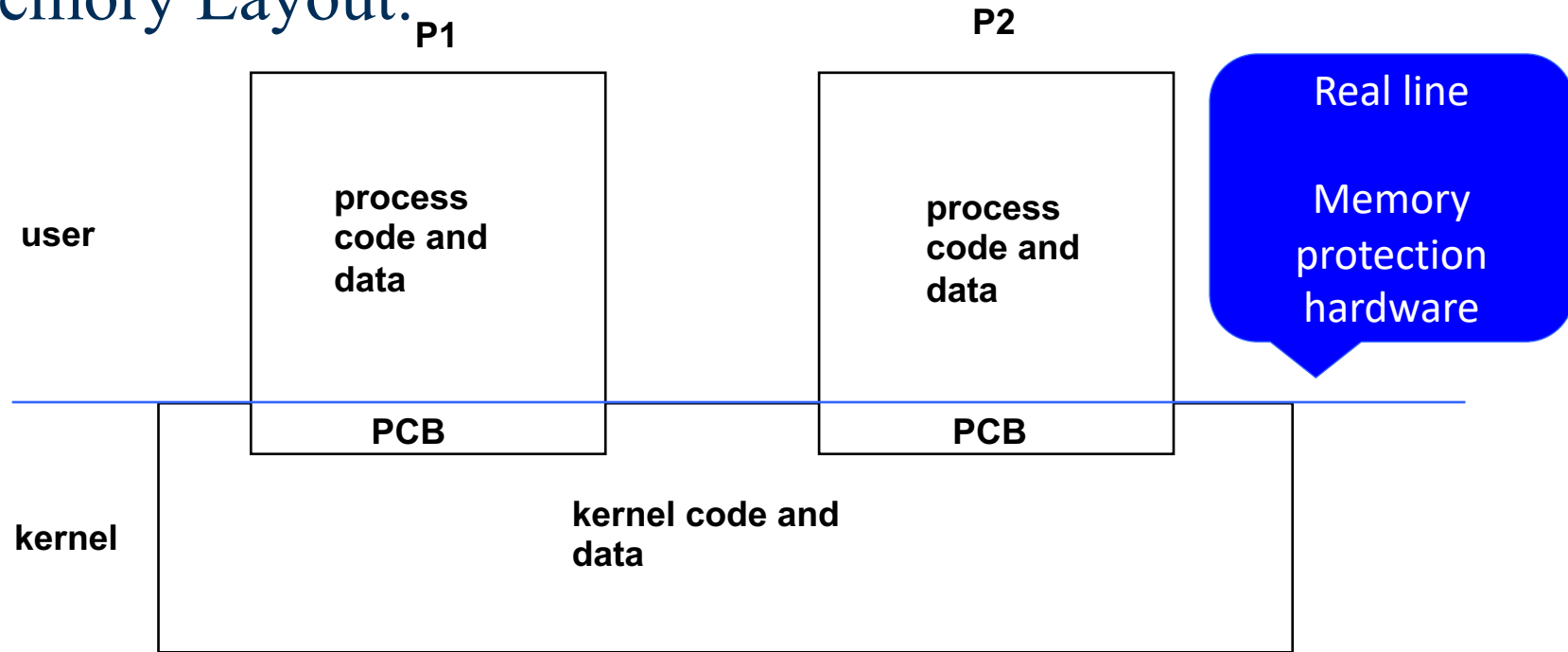
Prevents a "timing bug" or non-deterministic result in a parallel program!

Checkpoint

- ~~Pthreads programming~~
- OS issues with threads
- Hardware support for threads

Traditional OS: Unix

Memory Layout:



- Protection between user and kernel?
- Multiple processes, one thread each

Tradition

- Programs in traditional OS are single threaded
 - One PC per program (process), one stack, one set of CPU registers
 - If a process blocks (say disk I/O, network communication, etc.) then no progress for that program as a whole

Multi-Threaded Operating Systems

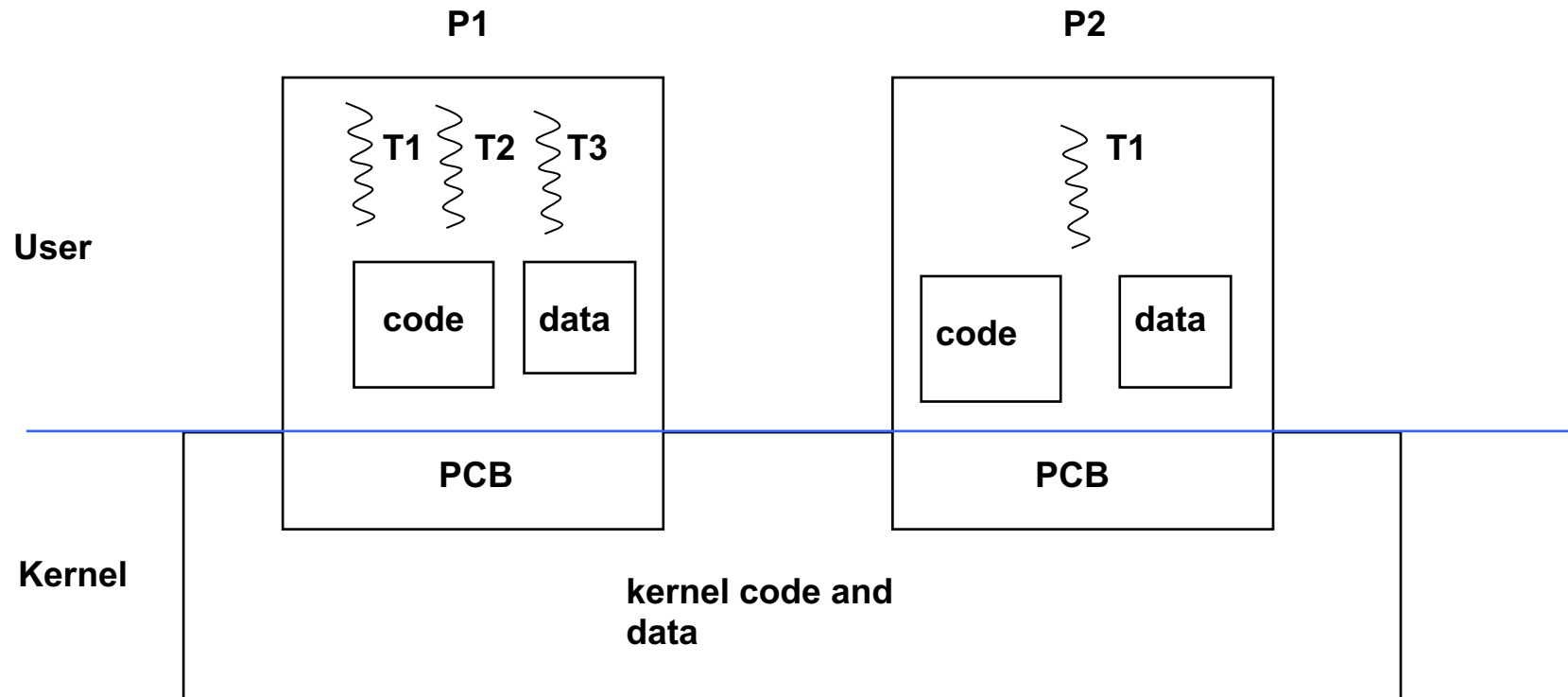
How widespread is support for threads in OS?

- Linux, MacOS, iOS, Android, Windows
- (In other words, every modern operating system)

Process Vs. Thread?

- In a single-threaded program, the state of the executing program is contained in a process
- In a MT program, the state of the executing program is contained in several 'concurrent' threads

Process Vs. Thread

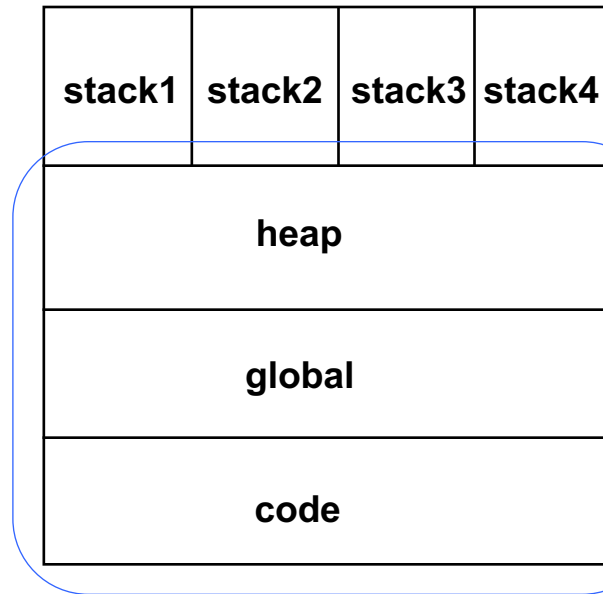


- Computational state (PC, regs, ...) for each thread
- How different from process state?
 - There's a lot of admin info in common

MT Bookkeeping



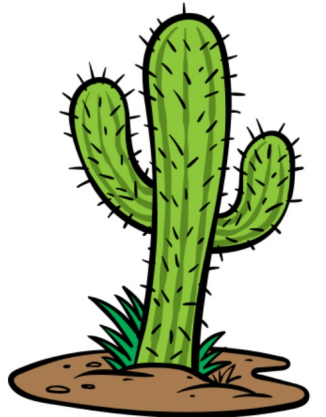
(a) ST program



(b) MT program

Common
for all
threads

- Can you see why the stack is sometimes called a "cactus stack"?



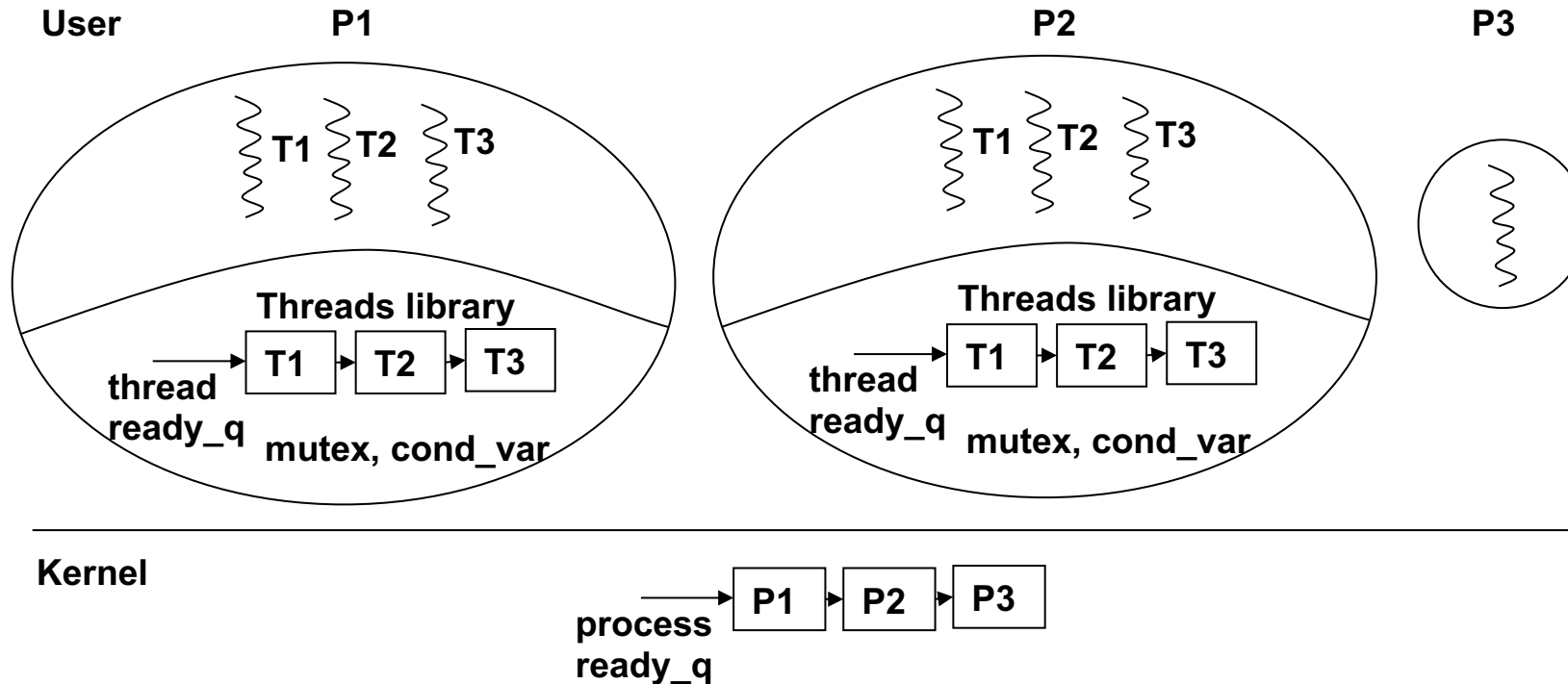
Thread properties

- Threads
 - Share address space of process
 - Cooperate to get job done
- Threads concurrent?
 - Truly parallel if the machine is a true multiprocessor
 - Share (time-multiplex) the same CPU on a uniprocessor
- Threaded code different from non-threaded?
 - Protection for data shared among threads
 - Synchronization among threads

User-Level Threads

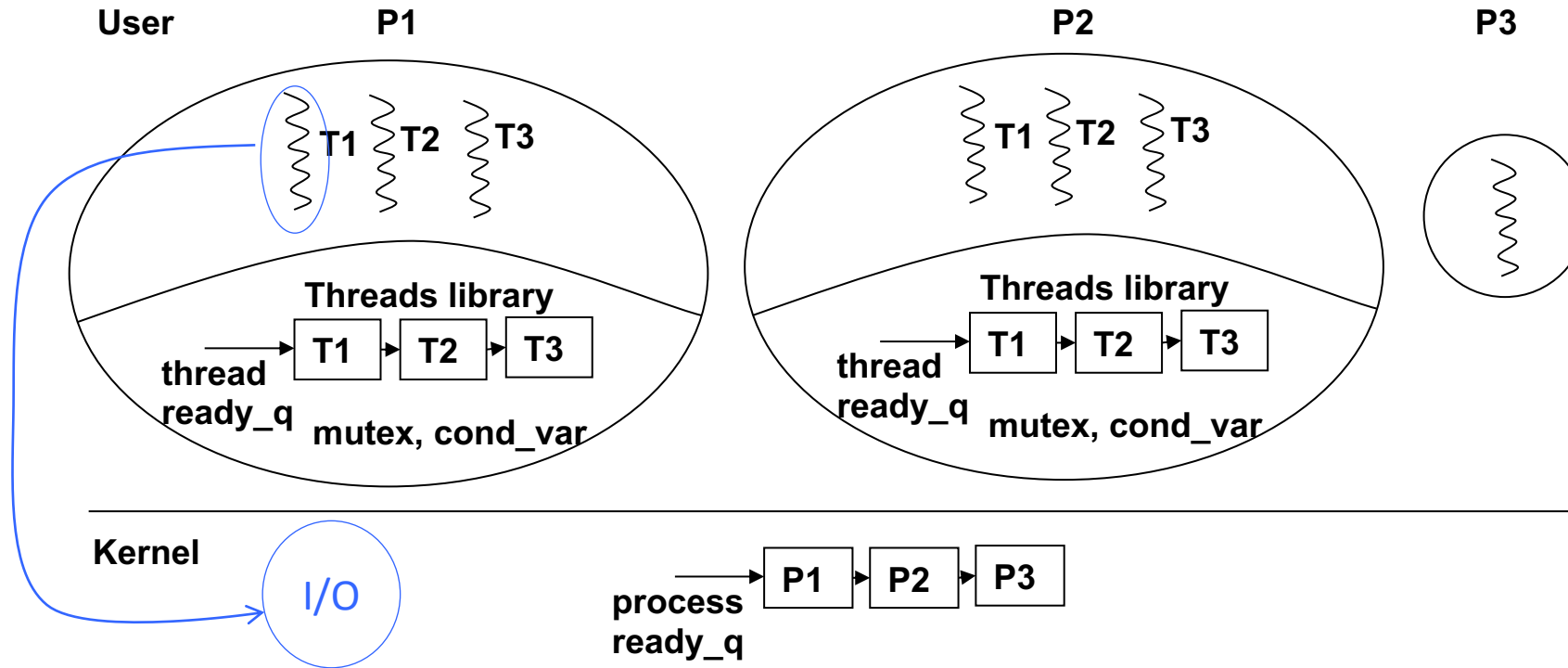
- OS independent
- Scheduler is part of the user space runtime system
- Thread switching is cheap (just save PC, SP, regs)
- Scheduling is customizable, i.e., more app control
- Blocking call by a thread blocks entire process

User-level threads



- OS independent
- Thread library part of application runtime
- Thread switching is cheap
- User-customizable thread scheduling

User-level threads



- Problem?
- Unfortunately, I/O blocks the entire process



User-level threads with process level scheduling...

10%

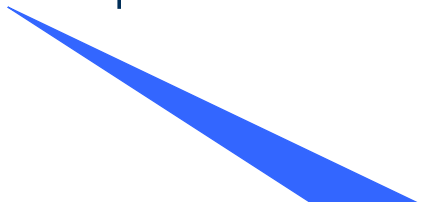
- A. I just want the participation credit
- B. ...serves no purpose since the operating system does not schedule at the thread level
- C. ...is useful for overlapping computation with I/O
- D. ...is useful as a software structuring mechanism at the user level
- E. All of the above

Kernel-level threads

- The norm in most modern operating systems
- Thread switch is more expensive
- Makes sense for blocking calls by threads
- Kernel becomes more complicated dealing with process and thread scheduling
- Thread packages become OS-dependent and non-portable

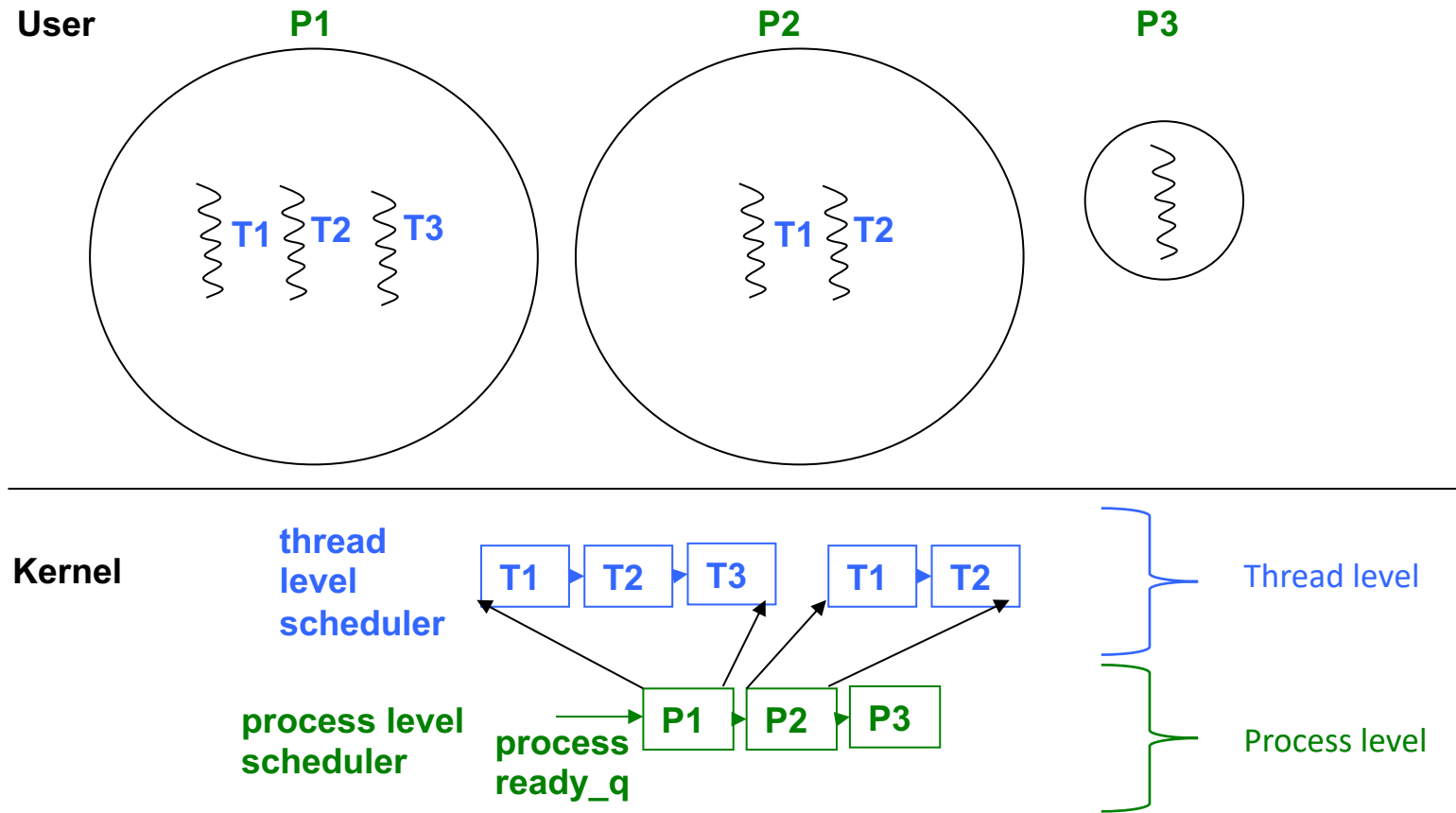


Compelling
reason for
kernel-level
threads



Reason for the
existence of
pthreads (POSIX)
standard

Two-level OS scheduler



- Threading in the application is visible to the OS
- OS provides the thread library

Thread-safe libraries

- Library functions (methods) have concurrency issues when used by user and kernel-level threads
 - All threads in a process share the heap and static data areas
 - Library routines that use static data or the heap are **very likely to implicitly share data** with other threads!
 - Solution is to have thread-safe wrappers to such library calls

Thread safe libraries

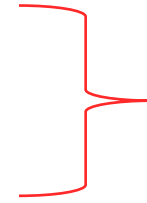
<code>/* original version */</code>		<code>/* thread safe version */</code>
		<code>mutex_lock_type cs_mutex;</code>
<code>void *malloc(size_t size)</code>		<code>void *malloc(size_t size)</code>
<code>{</code>		<code>{</code>
		<code> thread_mutex_lock(cs_mutex);</code>
<code> </code>		<code> memory_pointer = malloc(size);</code>
<code> </code>		<code> </code>
		<code> thread_mutex_unlock(cs_mutex);</code>
<code> return(memory_pointer);</code>		<code> return (memory_pointer);</code>
<code>}</code>		<code>}</code>

Checkpoint

- ~~Pthreads programming~~
- ~~OS issues with threads~~
- Hardware support for threads

Synchronization support in a uniprocessor

- Thread creation/termination
- Communication among threads
- Synchronization among threads
 - How do we implement mutex_lock?



Nothing special needed...

PT, TLB, Cache all the same

Proposed implementation of mutex

- Lock():
 while (mem_lock != 0)
 block the thread
 mem_lock = 1;

Oops! These instructions aren't atomic

How did we deal with that earlier?

Unlock():
 mem_lock = 0

We used OS mutex calls to make instructions inseparable.

- What could go wrong?

But now we ARE the OS!

Lock() using machine instructions

T1

➔ Lock()
START LD R1,mem_lock
BZ SET
JSR block_thread
B START
SET ADDI R1,#1
ST R1,mem_lock
RET

T2

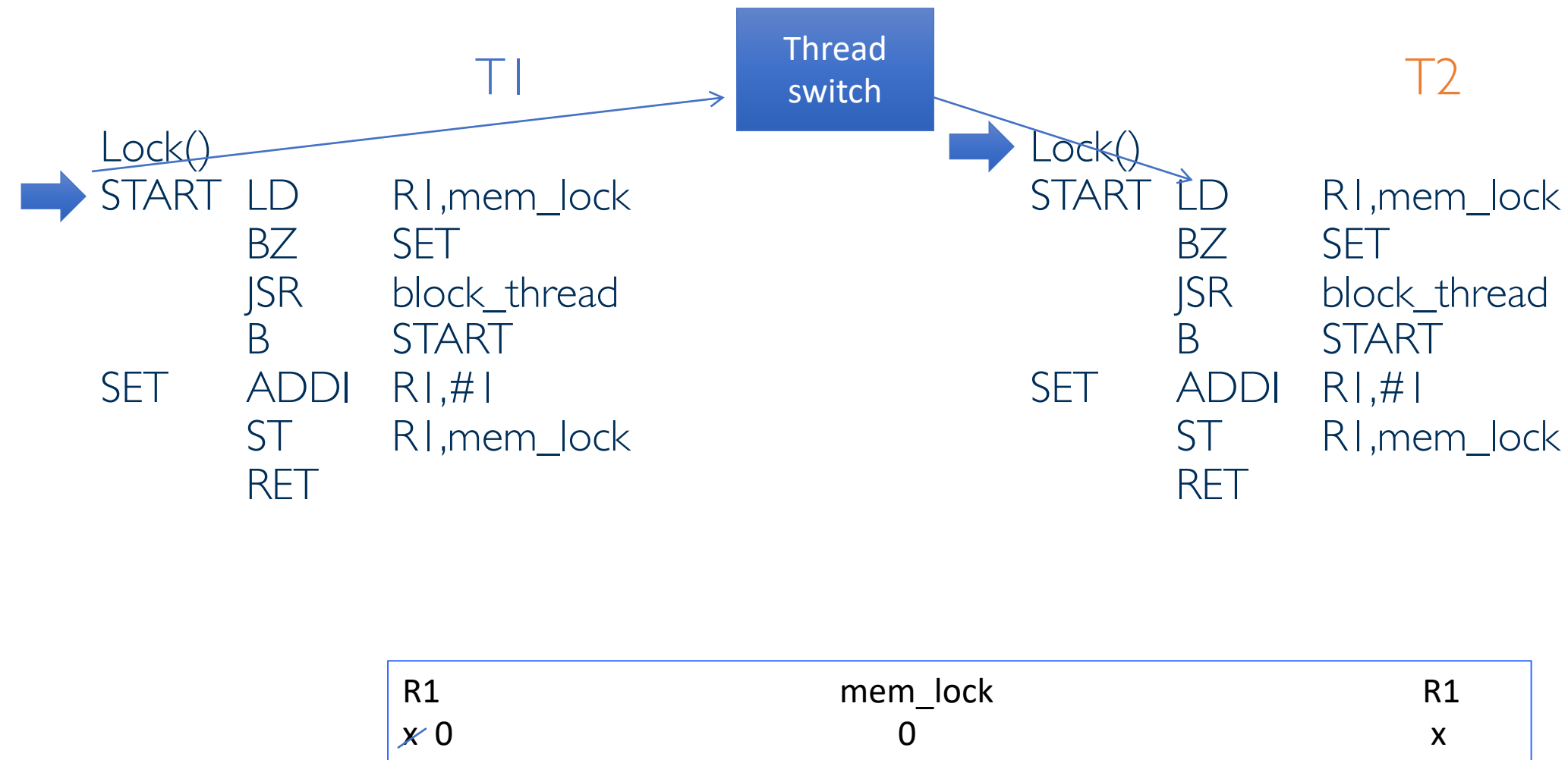
➔ Lock()
START LD R1,mem_lock
BZ SET
JSR block_thread
B START
SET ADDI R1,#1
ST R1,mem_lock
RET

R1
x

mem_lock
0

R1
x

Lock() using machine instructions



Lock() using machine instructions

T1

→ Lock()
START LD R1,mem_lock
BZ SET
JSR block_thread
B START
SET ADDI R1,#1
ST R1,mem_lock
RET

T2

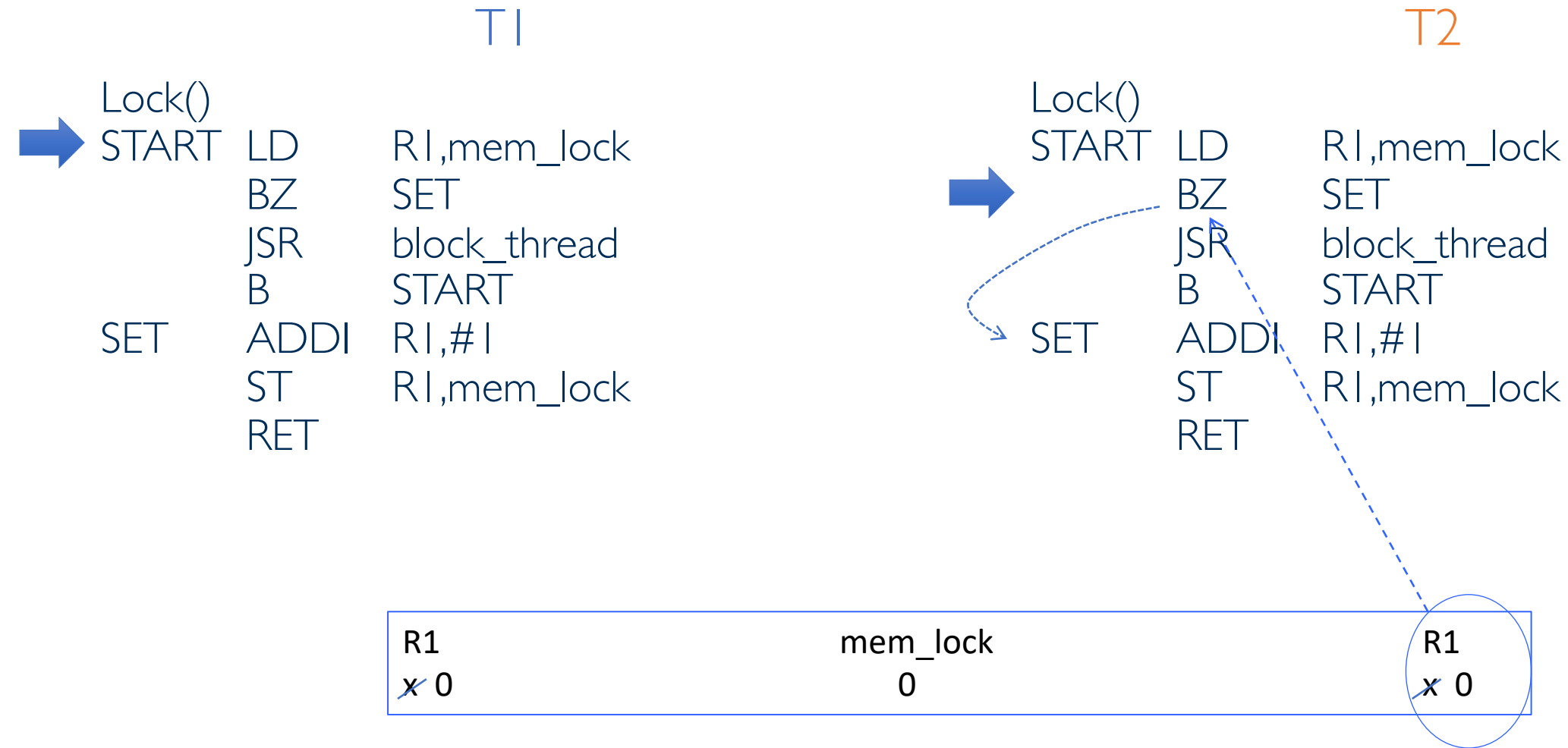
→ Lock()
START LD R1,mem_lock
BZ SET
JSR block_thread
B START
SET ADDI R1,#1
ST R1,mem_lock
RET

R1
~~x~~ 0

mem_lock
0

R1
~~x~~ 0

Lock() using machine instructions



Lock() using machine instructions



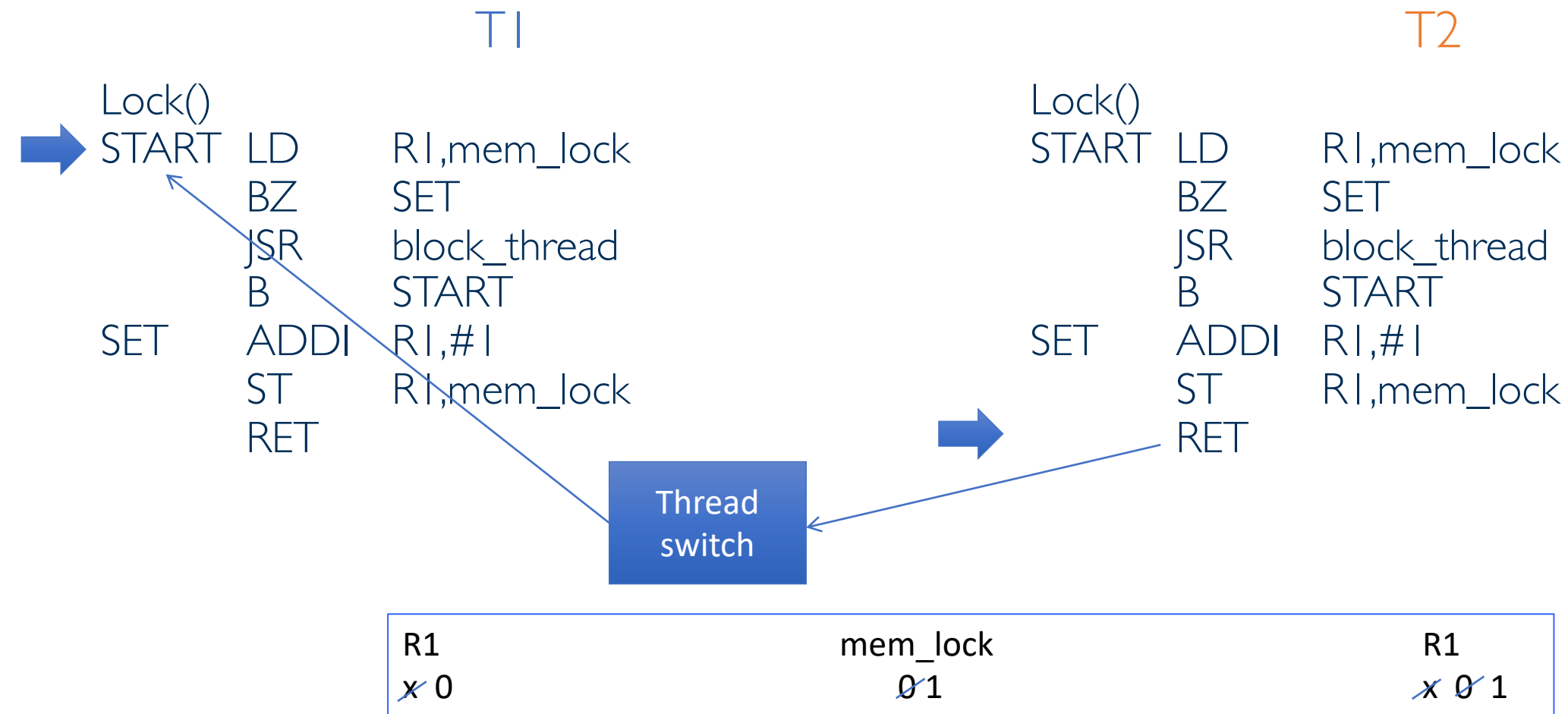
R1	mem_lock	R1
x 0	0	x 0 1

Lock() using machine instructions

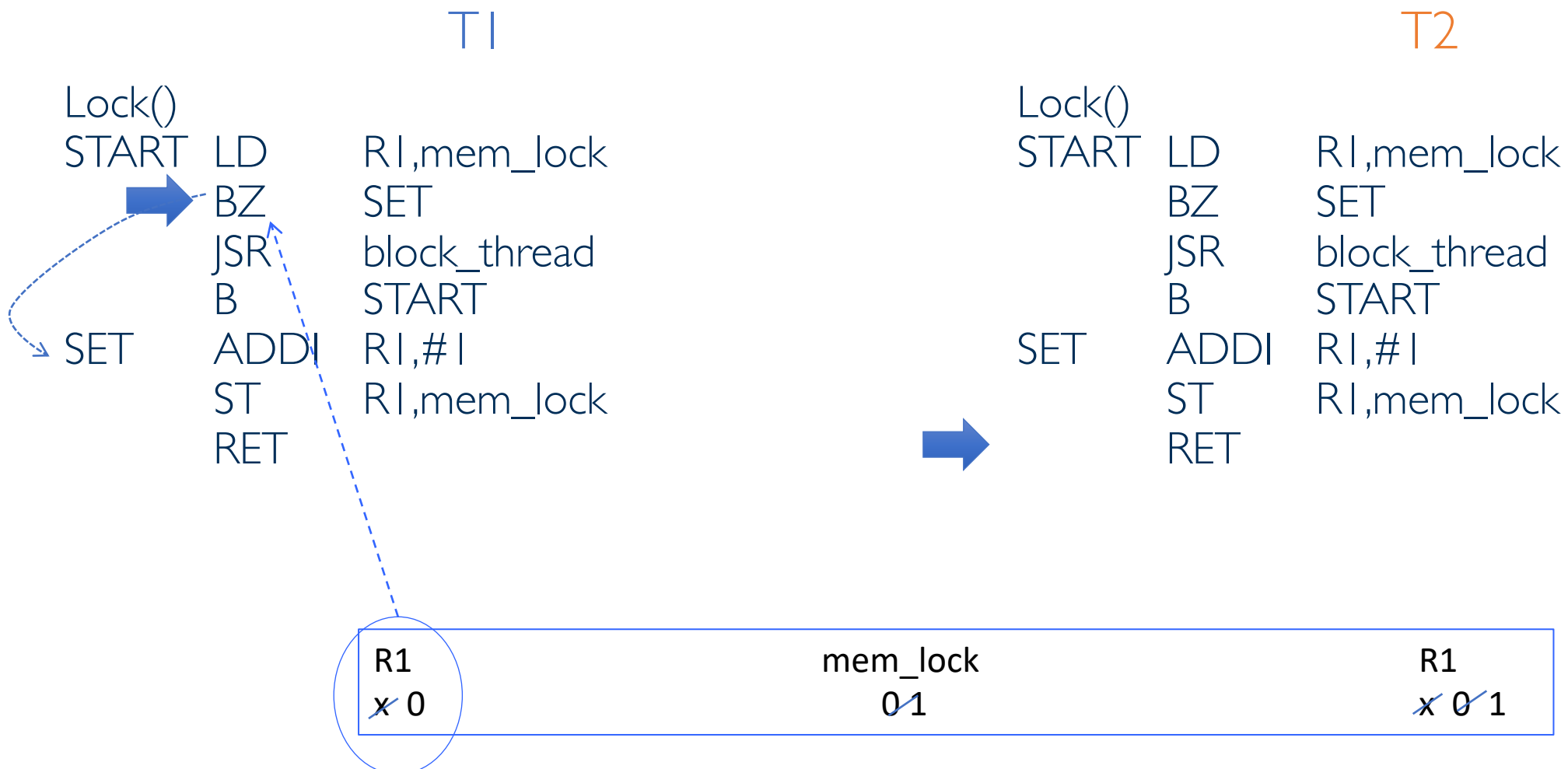


R1	mem_lock	R1
x 0	0 1	x 0 1

Lock() using machine instructions



Lock() using machine instructions



Lock() using machine instructions

T1

Lock()
START LD R1,mem_lock
BZ SET
JSR block_thread
B START
→ SET ADDI R1,#1
ST R1,mem_lock
RET

T2

Lock()
START LD R1,mem_lock
BZ SET
JSR block_thread
B START
SET ADDI R1,#1
→ ST R1,mem_lock
RET

R1
~~x~~ 0 1

mem_lock
~~0~~ 1

R1
~~x~~ 0 1

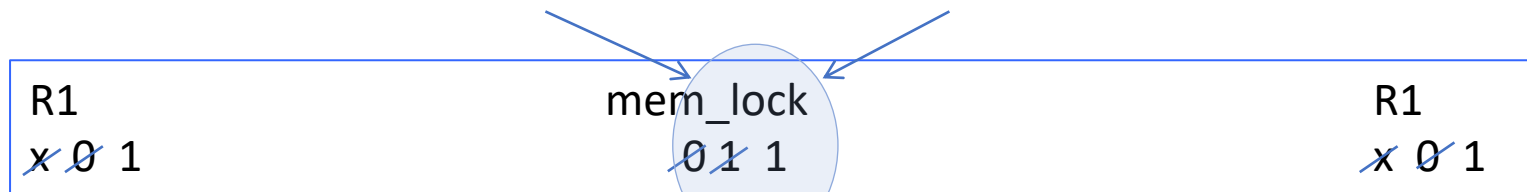
Lock() using machine instructions

TI

Lock()
START LD R1,mem_lock
BZ SET
JSR block_thread
B START
SET ADDI R1,#1
ST R1,mem_lock
RET

T2

Lock()
START LD R1,mem_lock
BZ SET
JSR block_thread
B START
SET ADDI R1,#1
ST R1,mem_lock
RET



What happened?!?

- We weren't in a critical section (We're implementing the lock for a critical section – how could we be in one??)
- Interrupts **can** happen between instructions

Lock():

```
while (mem_lock != 0)
    block the thread
mem_lock = 1;
```

Unlock():

```
mem_lock = 0
```

- We need the test and assignment to be atomic/indivisible!

Hardware to the rescue!

- We need an atomic Read-Modify-Write instruction
- TEST-AND-SET <memory-location>
load current value in <memory-location>
store 1 in <memory-location>
- Atomically:
 - Test L and set it to 1
 - If L tested originally as 0, we've claimed the lock
 - If L tested as 1, we need to try again
- Work-alikes
 - Compare-and-swap (IBM 370)
 - Fetch-and-add (Intel x86)
 - Load-linked/store-conditional (MIPS, ARM, ...)

Our new implementation of mutex

- Lock():
 while (test-and-set (&mem_lock))
 block the thread

Unlock():
 mem_lock = 0

Example: Implementing Mutex Lock

```
static int shared-lock = 0; /* global variable to
                             both T1 and T2 */
/* shared procedure for T1 and T2 */
int binary-semaphore(int *L)
{
    int X;

    X = test-and-set(L);

    /* X = 0 for successful return */
    return(X);
}
```

Two threads T1 and T2 execute the following statement simultaneously:

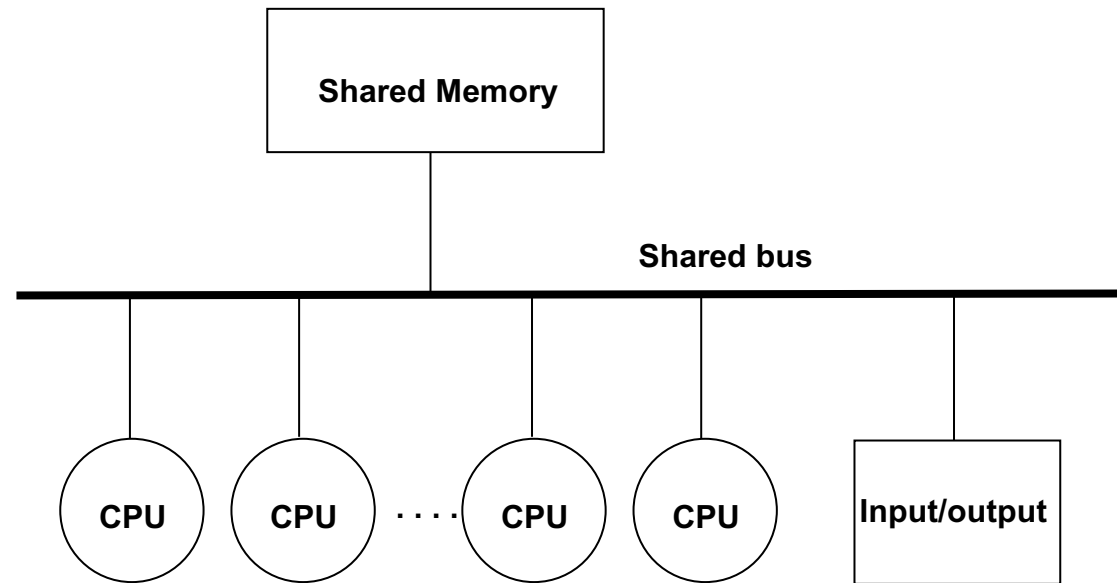
MyX = binary_semaphore(&shared-lock);

where MyX is a local variable in each of T1 and T2.

What are the possible values returned to T1 and T2?

Getting 0 0 isn't possible!

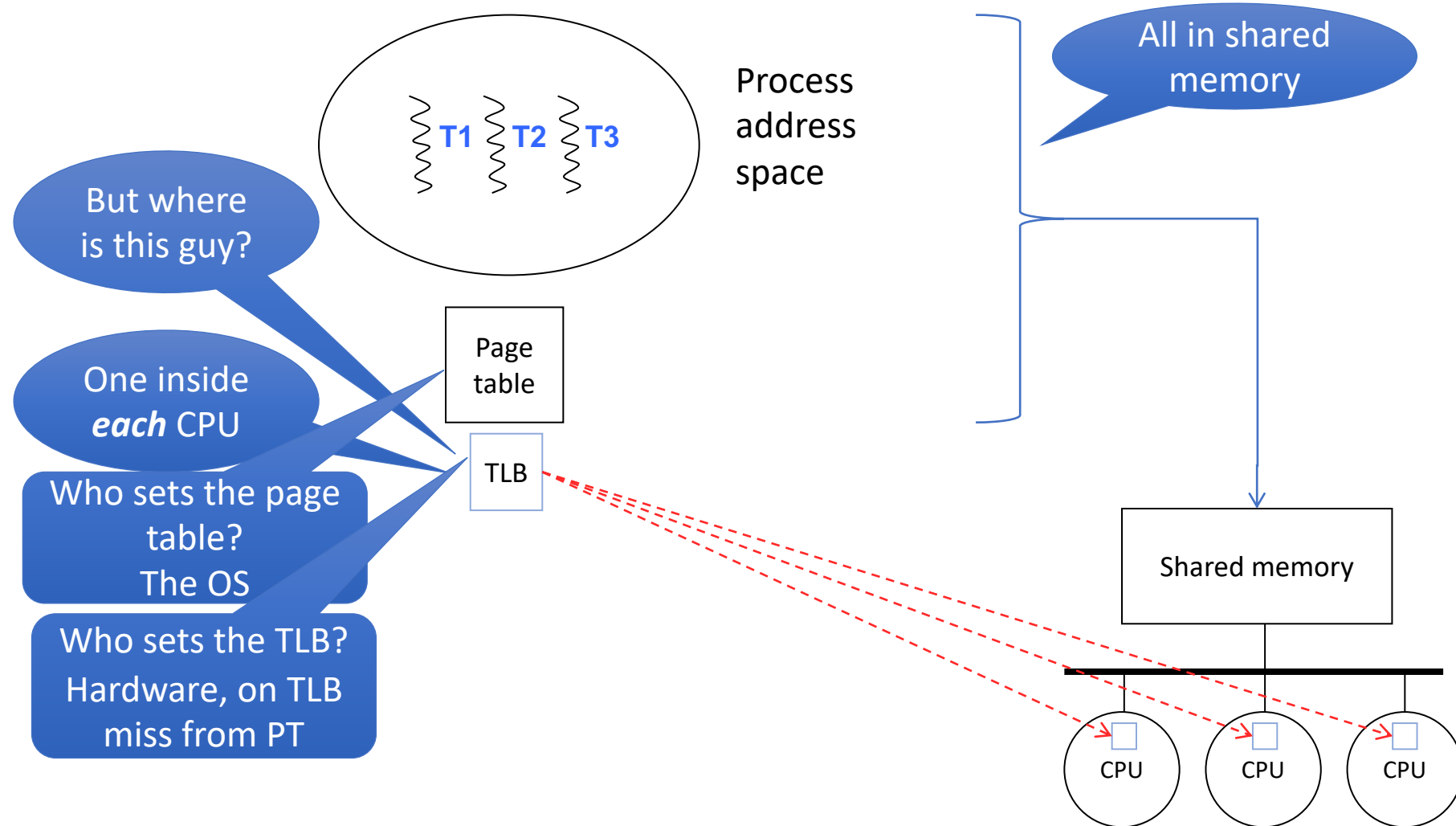
How do we implement Symmetric Multi Processing (SMP)?



The System (hardware+OS) has to ensure 3 things:

1. Threads of the same process share the same PT
2. Threads have synchronization atomicity
3. Threads have identical views of memory

1) All threads share the same page table



SMP context switch handling

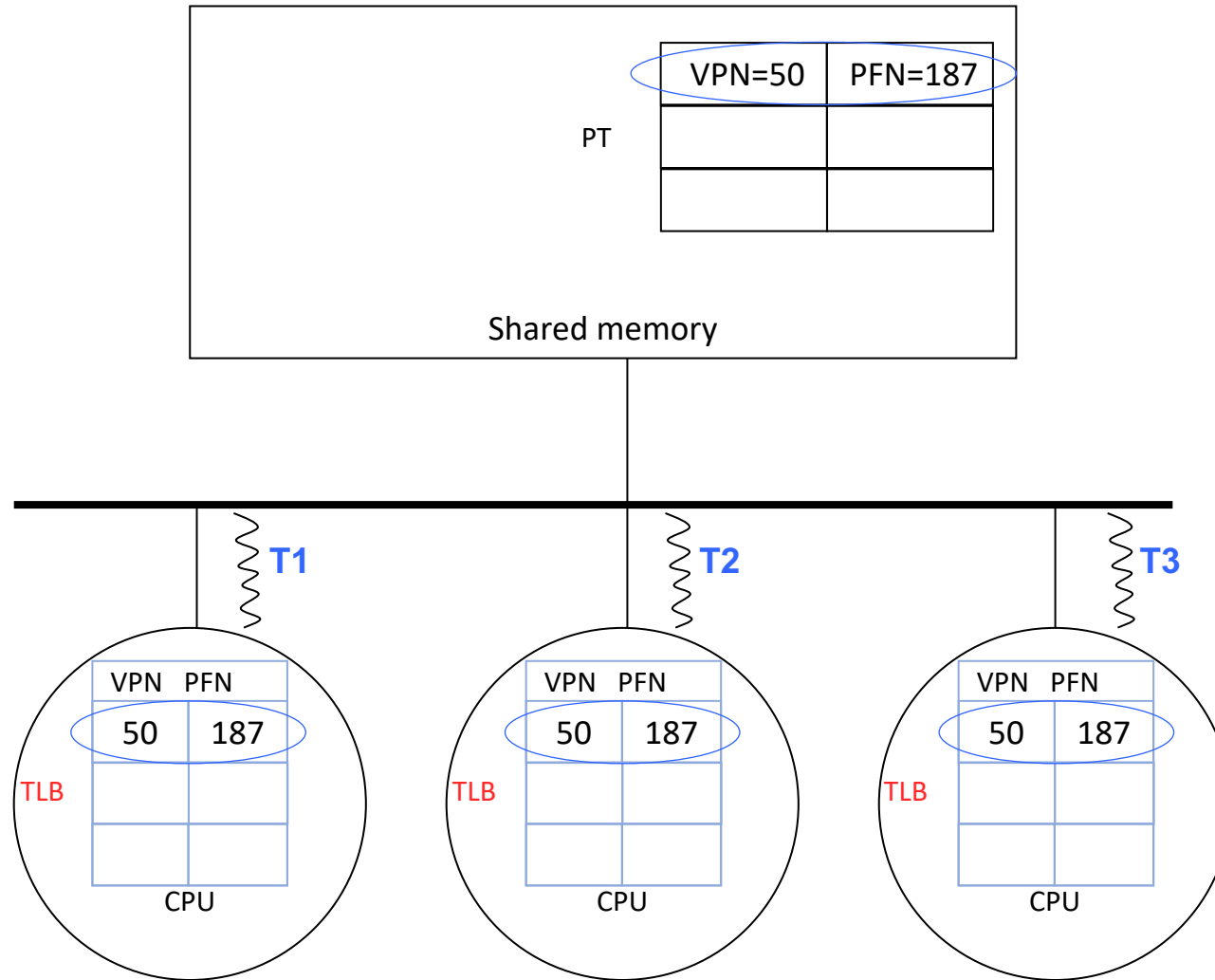
- As in the single-CPU case, the TLB must be flushed of user-space addresses on context switch
- How do multiple processors complicate this?
 - Basically, they don't
 - Any time a CPU is switched to a new thread, the OS flushes user entries from that CPU's TLB
 - There's no need to affect other TLBs

SMP page replacement handling

- On page replacement by the OS (which can happen on any of the CPUs)
- OS must
 - Evict the TLB entry for that page
(must happen even on a uniprocessor)
 - Tell the OS on other CPUs to evict the corresponding entry (if present) using software interrupts
 - This is called **TLB Shootdown**
- All of this happens in software by the OS
 - ➔ Another partnership of hardware and software

Example: Handling an SMP page fault

Note that the TLBs have each pulled in VPN=50 because the threads each referenced that page



Example: Handling an SMP page fault

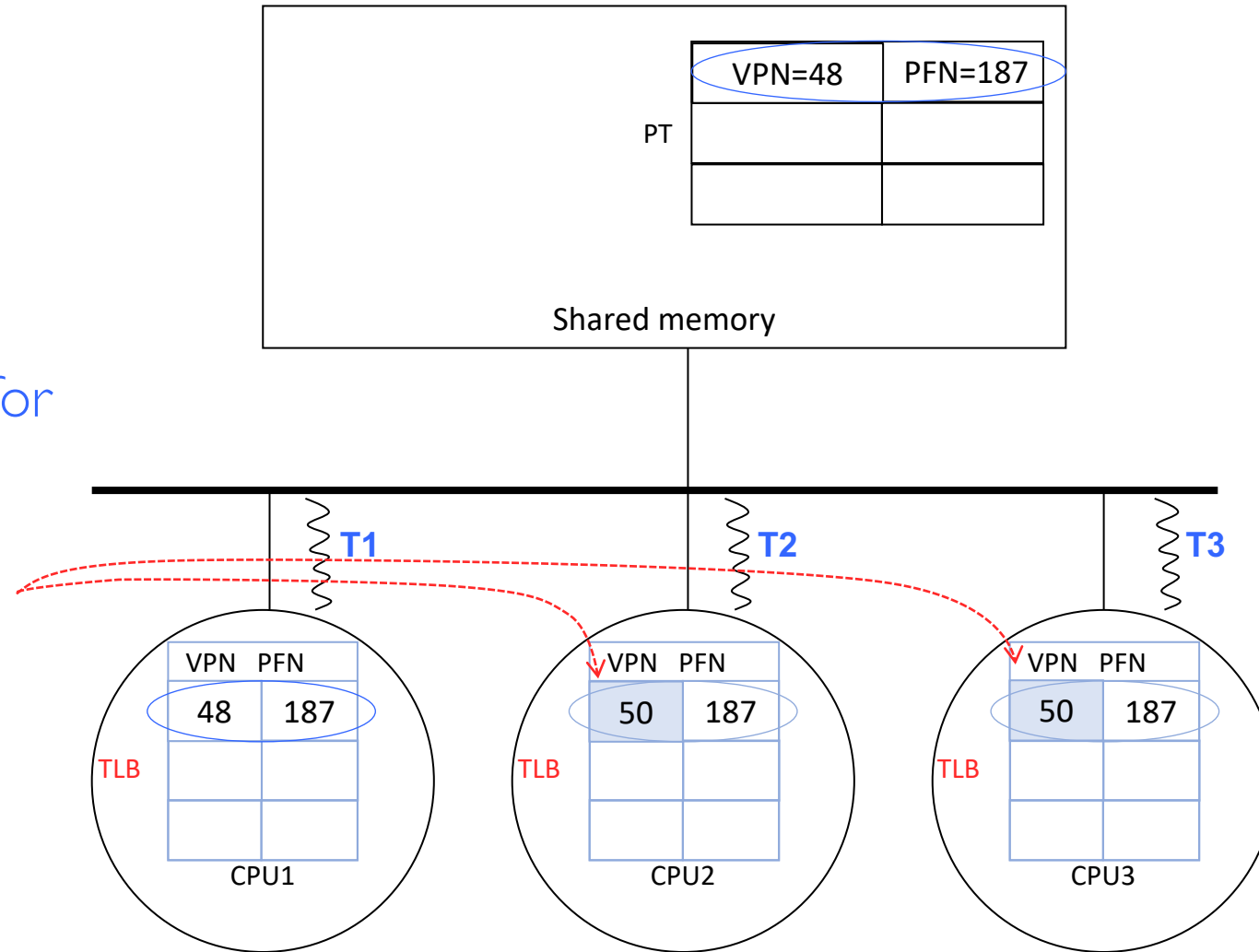
- Assume
 - TL encounters a page fault on VPN=48
 - OS decides to evict VPN=50
 - And use PFN=187 for hosting VPN=48

Example: Handling an SMP page fault

OS changes the page table entry for VPN=50

Then because it's running on CPU1, it evicts the TLB entry for VPN 50

Now we've got stale TLB entries in CPU2 and CPU3

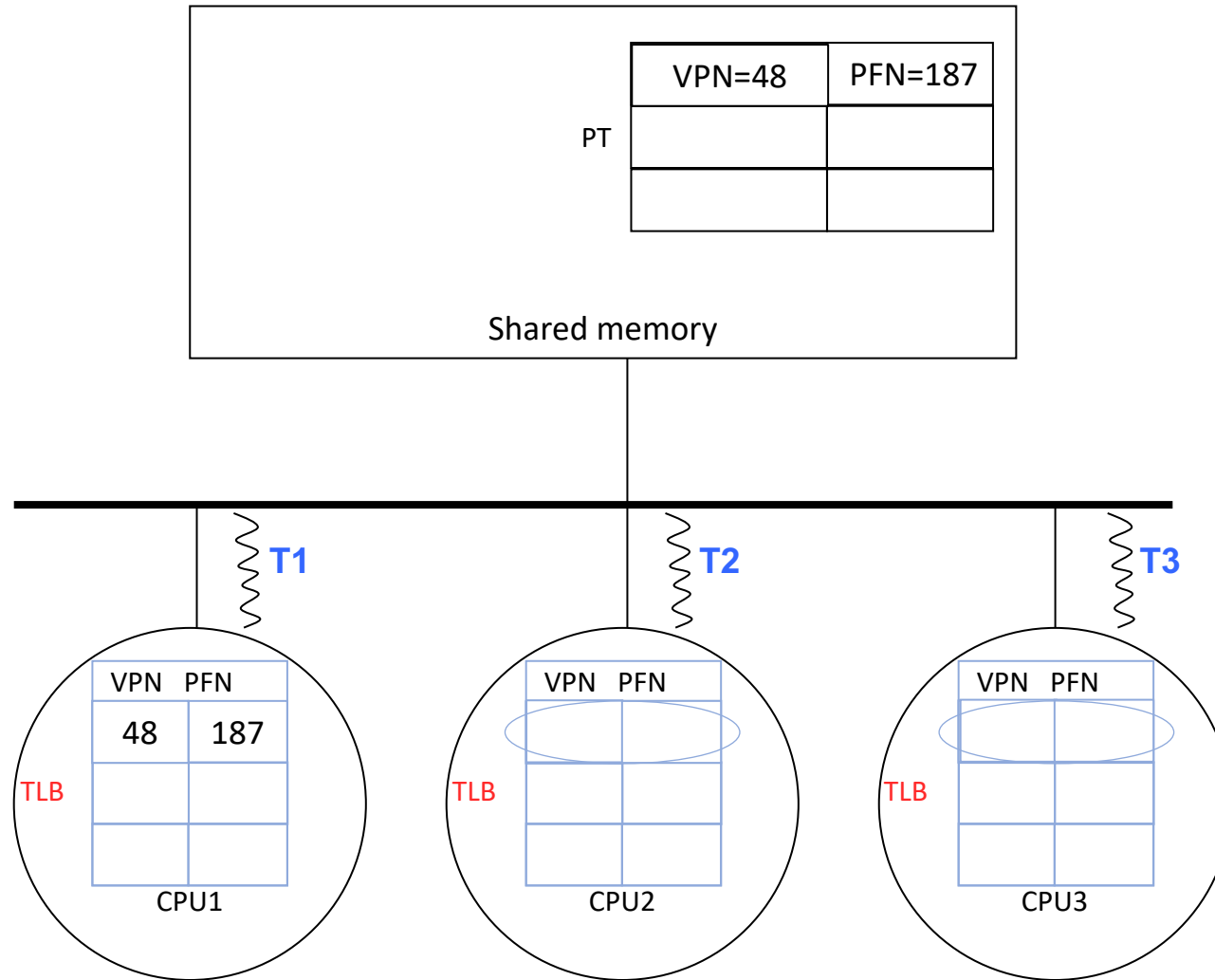


Example: Handling an SMP page fault

Then we have the TLB
Shutdown

The OS arranges to
invalidate the
corresponding TLB
entries on the other
CPUs

And the CPUs can
pull in the PTE when
they next reference
VPN=48





Ensuring that all threads of a process share an address space in an SMP is

20% A. I just want the participation credit

20% B. Impossible

20% C. Trivially achieved since the page table resides in shared memory

20% D. Achieved by careful replication of the page table by the operating system for each thread

E. Achieved by special-purpose hardware that no one has told us about yet

Keeping the TLBs consistent in an SMP

- 19% A. I just want the participation credit
- 13% B. Is the responsibility of the programmer
- 38% C. Is the responsibility of the hardware
- 19% D. Is the responsibility of the operating system
- 13% E. Is not possible

2) Threads have synchronization atomicity

- We already introduced the TEST-AND-SET instruction
- It should be easy on a multiprocessor, right?
- The location we use for synchronization is in shared memory, so no sweat.
- What could go wrong?

2) Threads have synchronization atomicity

We have memory location L for TEST-AND-SET

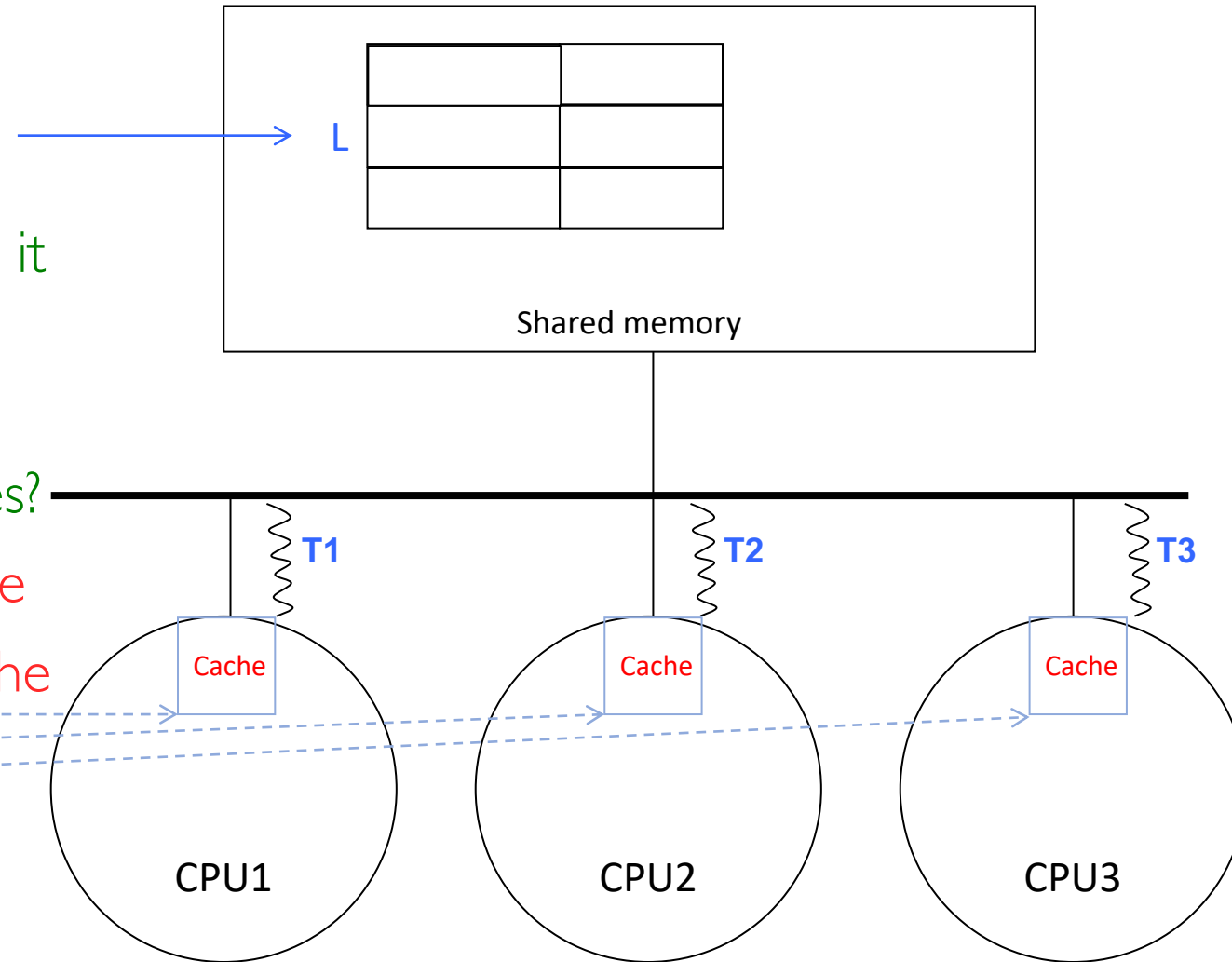
Each CPU can access it

But...what did we forget?

Where are the caches?

In the CPUs of course

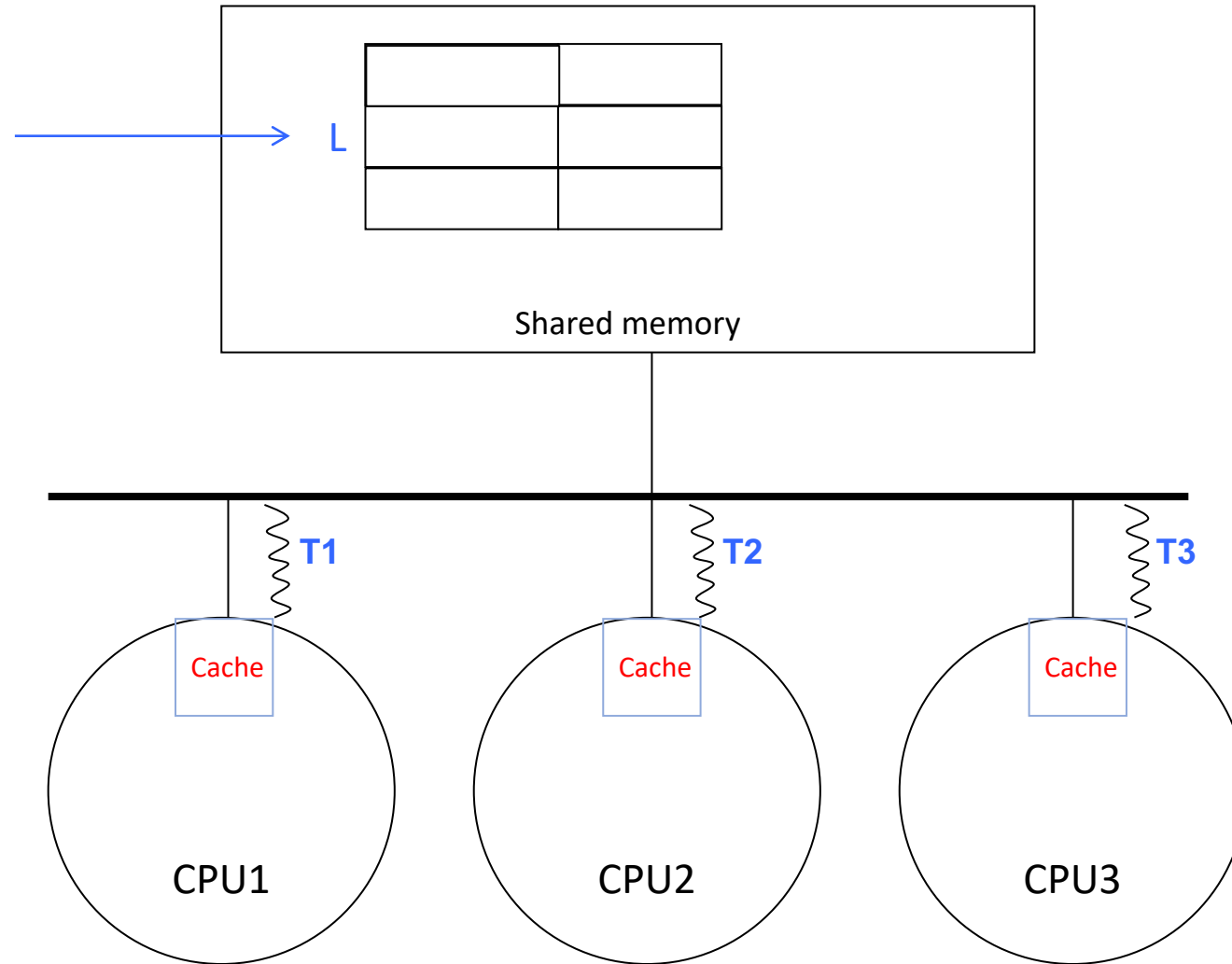
If L is cached, the cache is wrong!



2) Threads have synchronization atomicity

What shall we do?

One solution is to
bypass the cache for
the T&S instruction

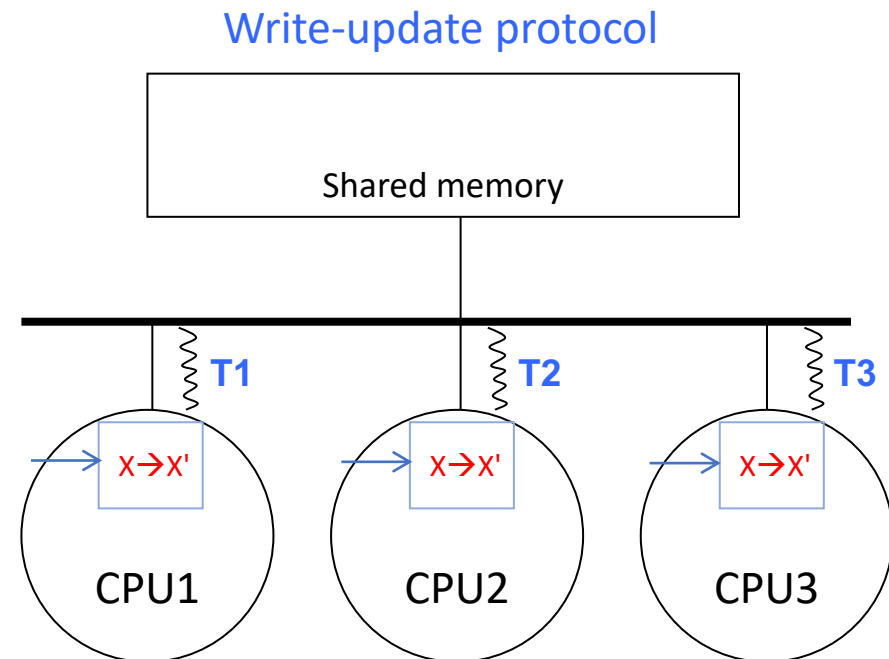
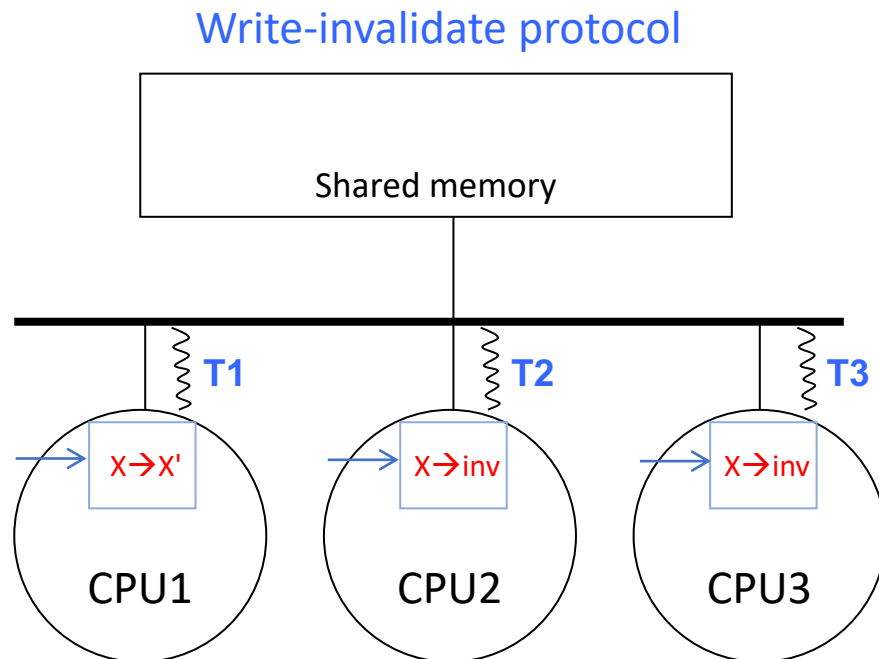


Requirements for SMP

- ~~1. Threads of the same process share the same PT~~
- ~~2. Threads have synchronization atomicity~~
3. Threads have identical views of memory
 - ➔ This implies that access to a memory location returns the same value on all CPUs
 - ➔ We'll refer to the method of keeping all the copies of the same data across caches as a **cache coherence protocol**

3) Threads have identical views of memory

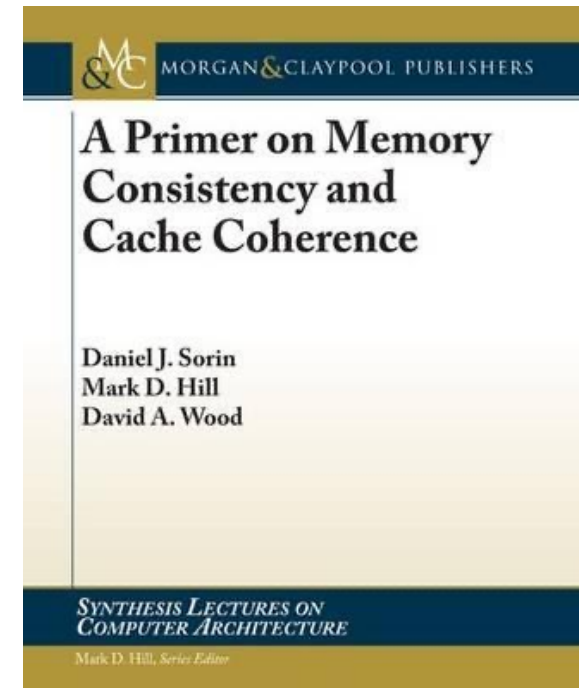
- Two possible solutions, **in hardware**
- Both: Cache becomes active and monitors or *snoops* the bus
- Let's watch a memory location change value from **X** to **X'**





Keeping the caches coherent in an SMP

- A. I just want the participation credit
- B. ...is the responsibility of the user program
- C. ...is the responsibility of the hardware
- D. ...is the responsibility of the operating system
- E. ...is impossible
- F. ...is why we don't allow caches in SMP systems



Summary

- Page tables in shared memory
 - Set up by the OS
 - Used by the hardware
- TLB consistency in software by the OS
 - Hardware brings PTE into the TLB from the PT
 - Page replacement algorithm changes the PT and does the TLB shoot-down
- Synchronized atomicity
 - Test-and-set instruction serialized by the shared bus
 - Atomic read-modify-write transaction
- Cache coherence in hardware
 - Invalidation based or update based