Parallel Systems

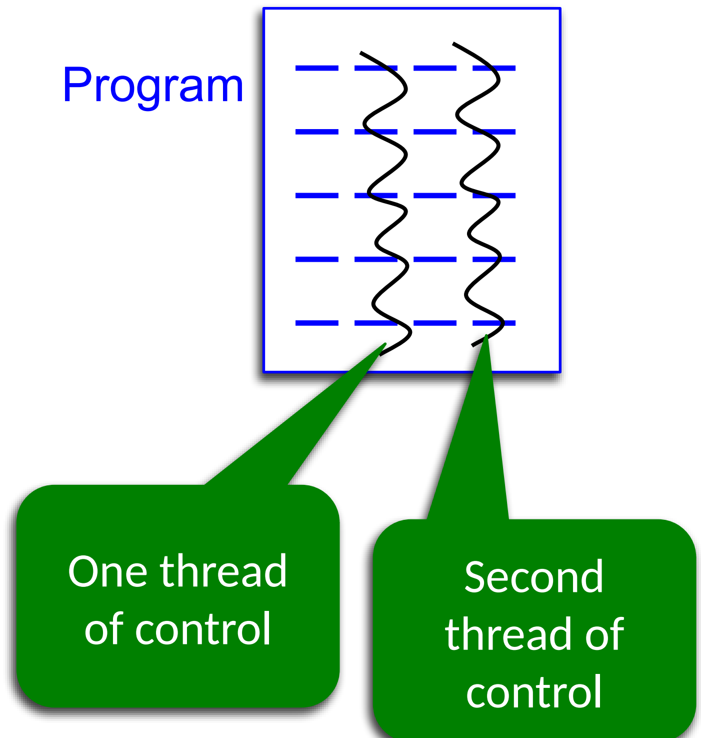
Chapter 12, Ramachandran and Leahy

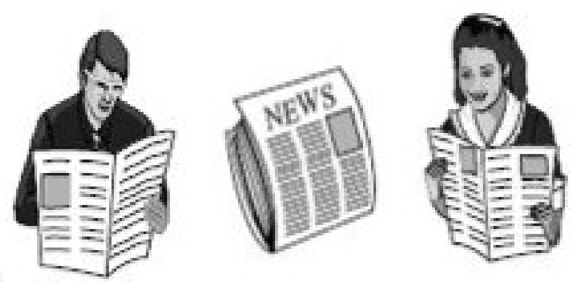
Abstractions

- Program
  - A static image loaded into memory
- Process
  - A program in execution
- In other words, process = program + state
  - The state evolves as the program executes
- Threads – an upgrade in the functionality of processes
  - In the modern interpretation, state is split into
    - data (main memory)
    - threads of control (PC and registers)
  - One memory space, but one or more threads of control

So then, what's a thread?

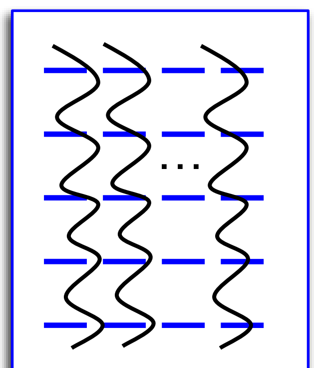
Program





In general:

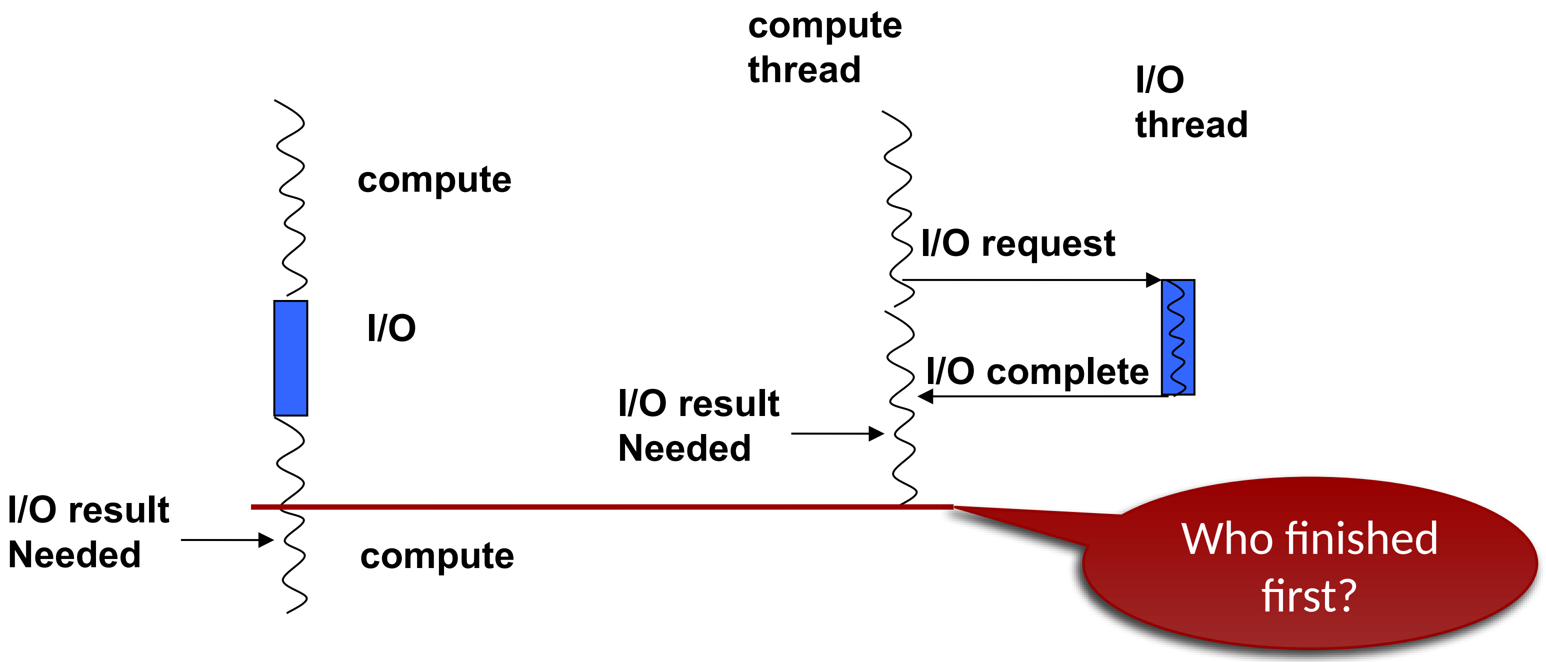
Program



n threads of control (n>0)

Process = program + state of all threads executing in the program

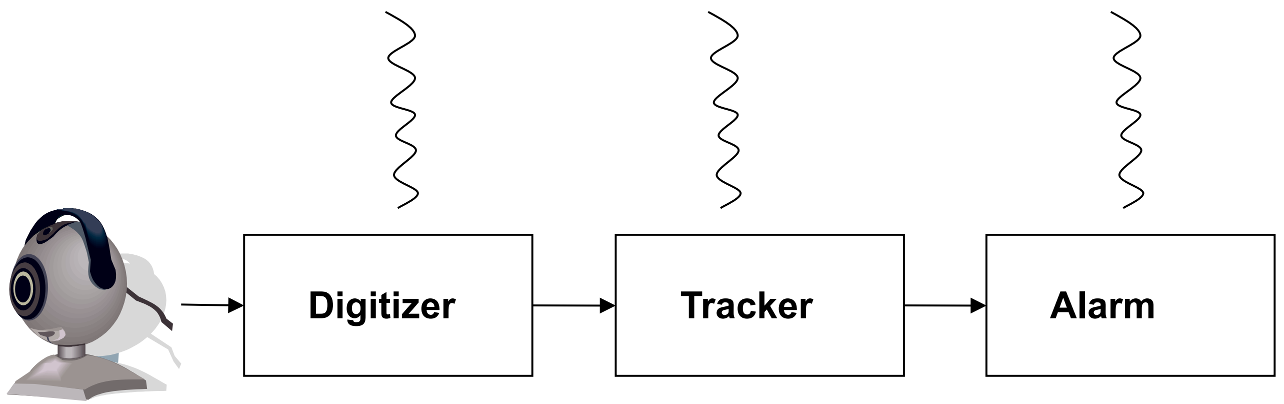
Example use of threads - 1



(a) Sequential process

(b) Multithreaded process

Example use of threads - 2



Digitize an image from the camera

Examine the image for motion

Send off a notification

Where are we headed?

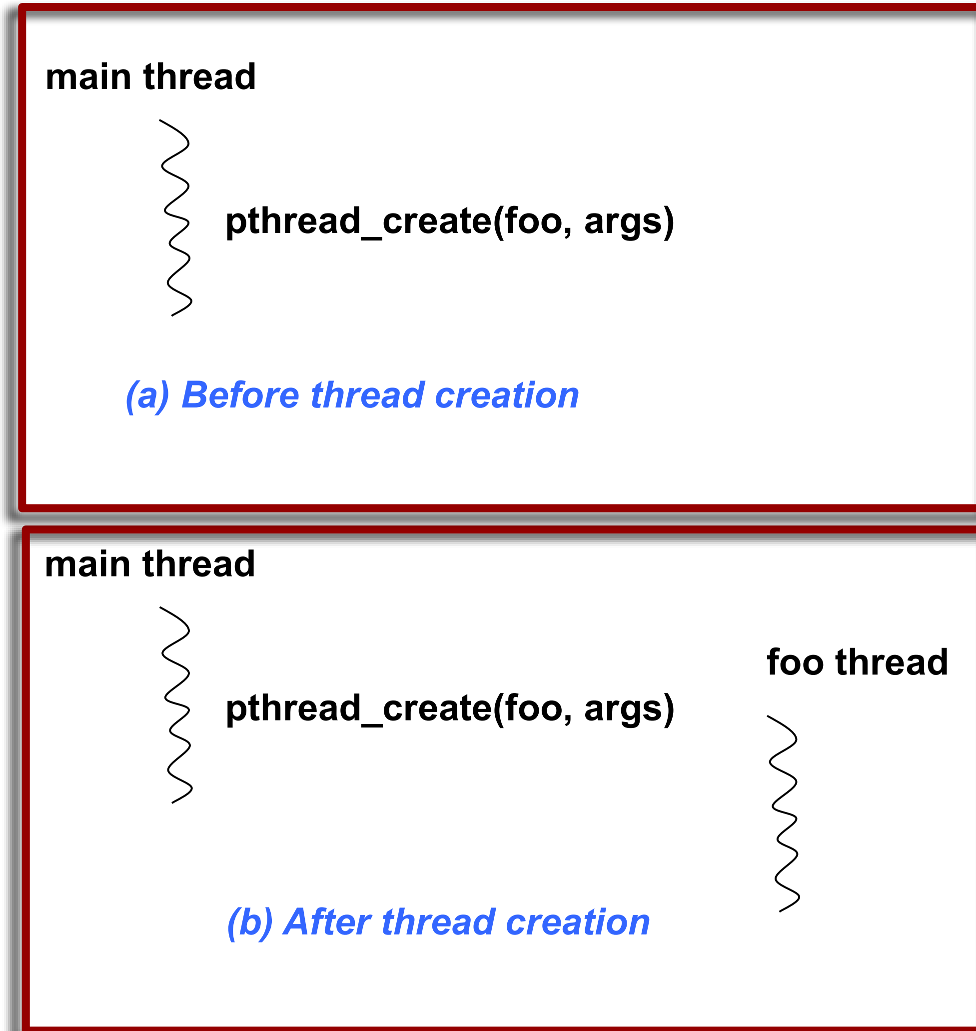
- Programming support for threads
- Synchronization and communication between threads
- Architecture and OS support for threads

C with pthreads

- Multithreaded program
  - Main and subroutines (procedures)
- Thread starts executing in some top-level procedure upon "thread create"
- Thread terminates
  - When main() terminates or
  - When the thread's top-level procedure terminates
- We are going to need synchronization and communication among threads
- Pthreads is a standard feature of a POSIX standards-compliant C library; implementations differ widely, but the behavior should be portable to any POSIX-compliant platform

Programming Support for Threads

- creation
  - pthread_create(top-level procedure, args)
- termination
  - return from top-level procedure
  - explicit kill
- rendezvous
  - creator can wait for children
    - pthread_join(child_tid)
- synchronization
  - mutex
  - condition variables



(a) Before thread creation

(b) After thread creation

Sample program – thread create/join

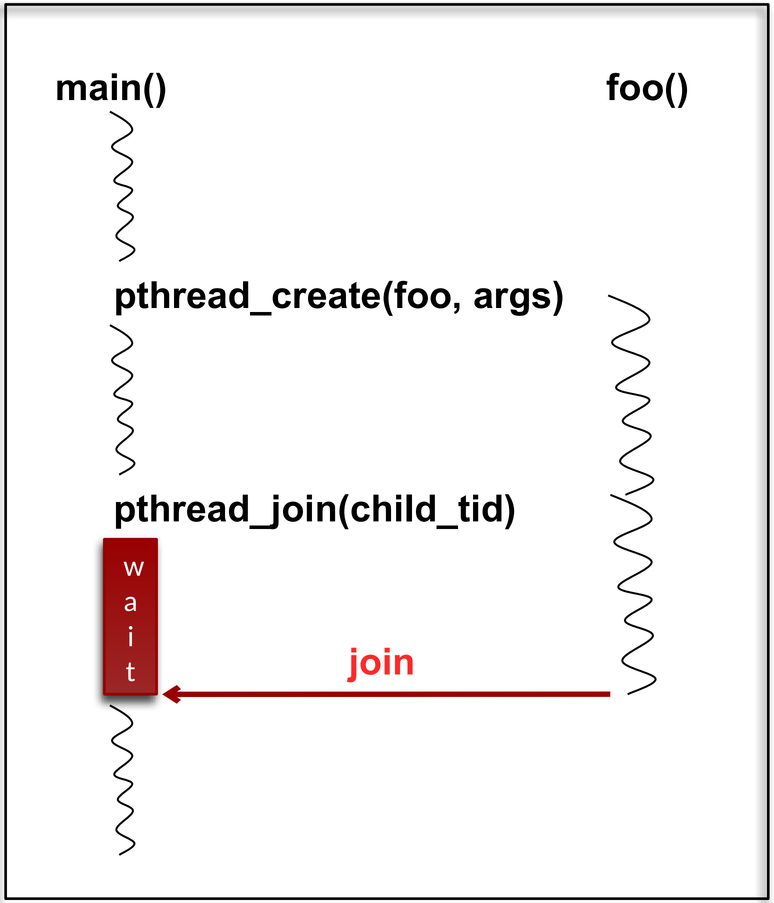
```
int foo(int n)
{
  .....
  return 0;
}
int main()
{
        int f;
        pthread_type child_tid;

        .....

        child_tid = pthread_create (foo,
        &f);

        .....

        pthread_join(child_tid);
}
```



Threads within the same process

A thread…

A. …is the same as a process
B. …is usually part of a process
C. …has nothing to do with a process
D. …usually refers to a set of processes
E. …is part of the memory hierarchy
F. …often involves a needle

Today’s number of interest: 80,008

A thread starts its execution …

A. In main()
B. At some top-level procedure that is part of the same program
C. At some top-level procedure that is part of a different program
D. None of the above

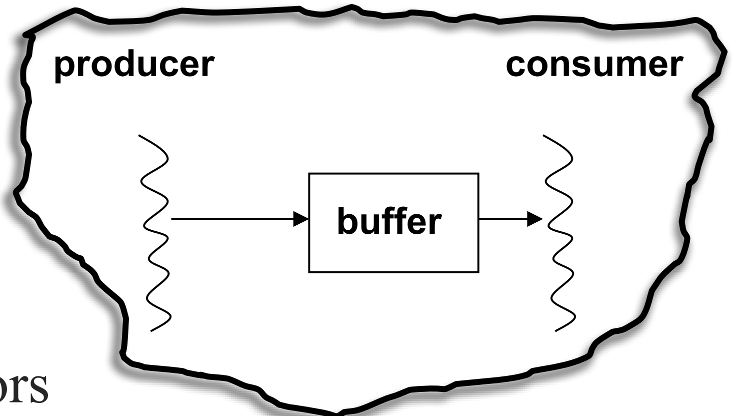
A thread …

A. … lives forever
B. … terminates ONLY when the top-level procedure where it started returns
C. … terminates ONLY when main() terminates
D. … terminates when EITHER the top-level procedure where it started or main() returns
E. … terminates at the first context switch

Programming with Threads

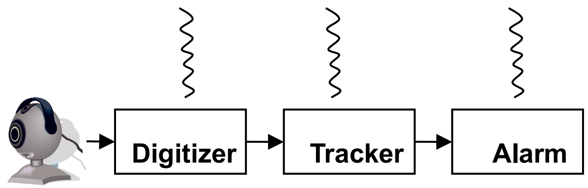
- synchronization
  - for coordination of the threads
- communication
  - for inter-thread sharing of data
  - threads can be in different processors
  - how to achieve sharing?
    - software: accomplished by keeping all threads in the same address space by the OS
    - hardware: accomplished by hardware shared memory and coherent caches (we will see this later)

Hardware software partnership

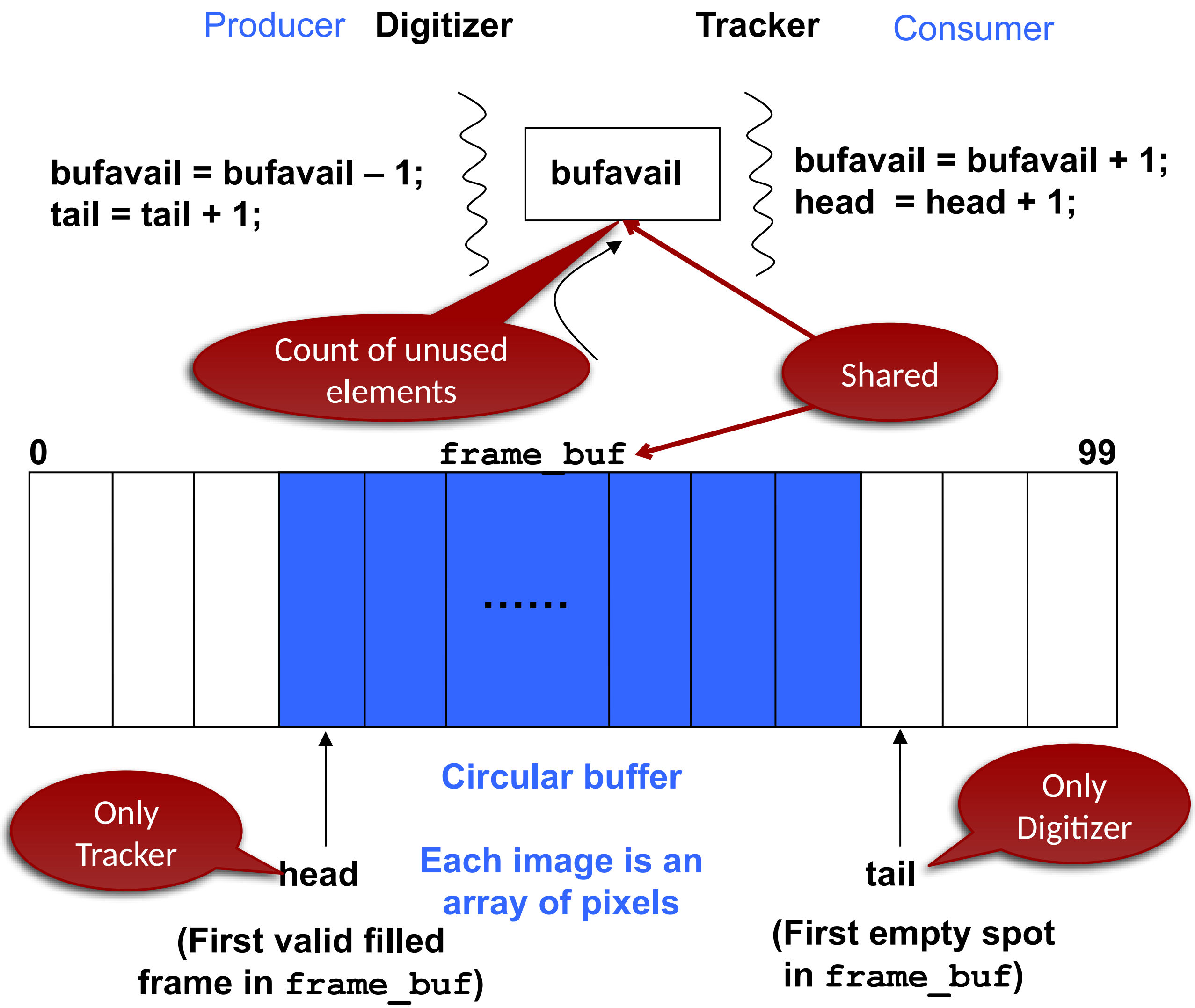


Same address space

Recall: our producer/consumer app



Global variables:
int bufavail = MAX;
image_type frame_buf[MAX];



Need for Synchronization

```
int bufavail = MAX; // global
image_type frame_buf[MAX]; // global

digitizer()
{
  image_type dig_image;
  int tail = 0; // private

  loop {
    if (bufavail > 0) {
      grab(dig_image);
      frame_buf[tail] = dig_image;
      tail = (tail + 1) % MAX;
      bufavail = bufavail - 1;
    }
  }
}
```

```
tracker()
{
  image_type track_image;
  int head = 0; // private

  loop {
    if (bufavail < MAX) {
      track_image = frame_buf[head];
      head = (head + 1) % MAX;
      bufavail = bufavail + 1;
      analyze(track_image);
    }
  }
}
```

Problem?

Need for Synchronization

```
int bufavail = MAX; // global
image_type frame_buf[MAX]; // global

digitizer()
{
  image_type dig_image;
  int tail = 0; // private

  loop {
    if (bufavail > 0) {
      grab(dig_image);
      frame_buf[tail] = dig_image;
      tail = (tail + 1) % MAX;
      bufavail = bufavail - 1;
    }
  }
}
```

```
tracker()
{
  image_type track_image;
  int head = 0; // private

  loop {
    if (bufavail < MAX) {
      track_image = frame_buf[head];
      head = (head + 1) % MAX;
      bufavail = bufavail + 1;
      analyze(track_image);
    }
  }
}
```

Problem?

Manipulating shared variables(!)

What's the issue?

Say that both threads happen to be executing at the red arrows…

Tracker

bufavail = bufavail – 1

```
LD      R1,bufavail
ADDI    R1,R1,-1
ST      R1,bufavail
```

Digitizer

bufavail = bufavail + 1

```
LD      R1,bufavail
ADDI    R1,R1,1
ST      R1,bufavail
```

R1
~~8~~ 7

bufavail
8

R1
~~8~~ 9

R1
~~8~~ 7

bufavail
is it 7 or 9?

R1
~~8~~ 9

But it should be 8! – FAIL!

And this is just one of many issues with this implementation!

Need for Synchronization

```
int bufavail = MAX; // global
image_type frame_buf[MAX]; // global

digitizer()
{
  image_type dig_image;
  int tail = 0; // private

  loop {
    if (bufavail > 0) {
      grab(dig_image);
      frame_buf[tail] = dig_image;
      tail = (tail + 1) % MAX;
      bufavail = bufavail - 1;
    }
  }
}
```

```
tracker()
{
  image_type track_image;
  int head = 0; // private

  loop {
    if (bufavail < MAX) {
      track_image = frame_buf[head];
      head = (head + 1) % MAX;
      bufavail = bufavail + 1;
      analyze(track_image);
    }
  }
}
```

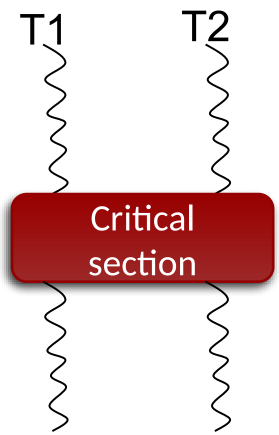
Problem?

Manipulating
shared variables(!)

Synchronization Primitives

- lock and unlock
  - mutual exclusion among threads
  - busy-waiting vs. blocking
  - pthread_mutex_trylock: no blocking
  - pthread_mutex_lock: blocking
  - pthread_mutex_unlock

Usage: pthread_mutex_t lock; // data structure
pthread_mutex_lock(&lock); // acquire lock
pthread_mutex_unlock(&lock); //release lock



- OS has no idea what you do in the critical section
- OS guarantees only 1 thread runs the critical section at a given time (we call this guarantee an OS invariant)

Critical Section

- "Code that is executed in a mutually exclusive manner"
- Shared access to data that must be synchronized
  - so we implement a mutual exclusion lock
  - which is honored by one or more segments of code that access the shared data
- A critical section is not necessarily a single piece of code. Any segment of code that honors the same mutual exclusion lock is called a critical section.

Fix number 1 – with locks

```
int bufavail = MAX;
image_type frame_buf[MAX];
pthread_mutex_t buflock;

digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop {
    pthread_mutext_lock(buflock);
    if (bufavail > 0) {
      grab(dig_image);
      frame_buf[tail] = dig_image;
      tail = (tail + 1) % MAX;
      bufavail = bufavail - 1;
    }
    pthread_mutex_unlock(buflock);
  }
}
```

```
tracker()
{
  image_type track_image;
  int head = 0;

  loop {
    pthread_mutext_lock(buflock);
    if (bufavail < MAX) {
      track_image = frame_buf[head];
      head = (head + 1) % MAX;
      bufavail = bufavail + 1;
      analyze(track_image);
    }
    pthread_mutex_unlock(buflock);
  }
}
```

A pthreads mutex lock

A. Allows exactly one thread to acquire it at a time
B. Allows any number of threads to acquire it at a time
C. Allows a defined number of threads to acquire it at a time
D. None of the above

Today's number is 10,987

Fix number 1 – with locks

```
int bufavail = MAX;
image_type frame_buf[MAX];
pthread_mutex_t buflock;

digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop {
   pthread_mutext_lock(buflock);
   if (bufavail > 0) {
     grab(dig_image);
     frame_buf[tail] = dig_image;
     tail = (tail + 1) % MAX;
     bufavail = bufavail - 1;
   }
   pthread_mutex_unlock(buflock);
  }
}
```

```
tracker()
{
  image_type track_image;
  int head = 0;

  loop {
   pthread_mutext_lock(buflock);
   if (bufavail < MAX) {
     track_image = frame_buf[head];
     head = (head + 1) % MAX;
     bufavail = bufavail + 1;
     analyze(track_image);
   }
   pthread_mutex_unlock(buflock);
  }
}
```

Critical section is far too coarse!

Problem?

No concurrency!
No performance improvement.

Fix number 1 – with locks

```
int bufavail = MAX;
image_type frame_buf[MAX];
pthread_mutex_t buflock;
```

```
digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop {
   pthread_mutext_lock(buflock);
   if (bufavail > 0) {
     grab(dig_image);
     frame_buf[tail] = dig_image;
     tail = (tail + 1) % MAX;
     bufavail = bufavail - 1;
   }
   pthread_mutex_unlock(buflock);
  }
}
```

No need for mutex

```
tracker()
{
  image_type track_image;
  int head = 0;

  loop {
   pthread_mutext_lock(buflock);
   if (bufavail < MAX) {
     track_image = frame_buf[head];
     head = (head + 1) % MAX;
     bufavail = bufavail + 1;
     analyze(track_image);
   }
   pthread_mutex_unlock(buflock);
  }
}
```

Fix number 2 – with locks

```
int bufavail = MAX;
image_type frame_buf[MAX];
pthread_mutex_t buflock;

digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop {
    grab(dig_image);
    pthread_mutex_lock(buflock);
      while (bufavail == 0) ; // do nothing
    pthread_mutex_unlock(buflock);
    frame_buf[tail] = dig_image;
    tail = (tail + 1) % MAX;
    pthread_mutex_lock(buflock);
      bufavail = bufavail - 1;
    pthread_mutex_unlock(buflock);
  }
}
```

```
tracker()
{
  image_type track_image;
  int head = 0;

  loop {
    pthread_mutex_lock(buflock);
      while (bufavail == MAX) ; // do nothing
    pthread_mutex_unlock(buflock);
    track_image = frame_buf[head];
    head = (head + 1) % MAX;
    pthread_mutex_lock(buflock);
      bufavail = bufavail + 1;
    pthread_mutex_unlock(buflock);
    analyze(track_image);
  }
}
```

Problem?

Deadlock!

Fix number 3

```
int bufavail = MAX;
image_type frame_buf[MAX];
pthread_mutex_t buflock;

digitizer()
{
  image_type dig_image;
  int tail = 0;

  loop {
    grab(dig_image);

    while (bufavail == 0) ; // do nothing

    frame_buf[tail] = dig_image;
    tail = (tail + 1) % MAX;
    pthread_mutex_lock(buflock);
      bufavail = bufavail - 1;
    pthread_mutex_unlock(buflock);
  }
}
```

We're only reading so no need for mutex

```
tracker()
{
 image_type track_image;
 int head = 0;

 loop {

  while (bufavail == MAX) ; // do nothing

  track_image = frame_buf[head];
  head = (head + 1) % MAX;
  pthread_mutex_lock(buflock);
    bufavail = bufavail + 1;
  pthread_mutex_unlock(buflock);
  analyze(track_image);
 }
}
```

Problem?

Busy waiting ® Wastes CPU

Deadlock and Livelock

- **Deadlock**, a.k.a "deadly embrace", is present when a thread waits on a condition that can never occur.
- It is often manifested by a "circular wait" where a thread A holds a lock and then waits on thread B which then tries to acquire the lock held by thread A.
- **Livelock** is a special case of deadlock in which the threads are changing state while waiting (i.e. they are wasting CPU resources while waiting infinitely).
- There is much literature on preventing, avoiding, and detecting deadlocks which is beyond our scope right now.
- However, one scheme for avoiding deadlock is to create a hierarchy of locks. Then by convention, a thread may only request a lock at a higher level than any lock the thread already holds. It must first release its higher-level locks if it is to wait on a lower-level lock.

We have deadlock when thread A is waiting on thread B and

A. Thread B then waits on thread A
B. Thread B then waits on thread C which then waits on thread A
C. Thread B then tries to claim a mutex lock which is held by thread A
D. All of the above

What should really happen?

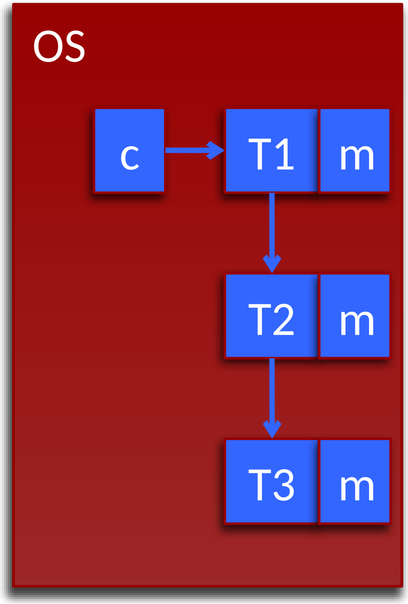
- Prevent the busy-waiting
- If frame_buf is full
  - Tracker is slow, so digitizer is waiting for space in frame_buf
  - Tracker should let digitizer know when it makes room in frame_buf
- If frame_buf is empty
  - Digitizer is slow, so tracker is waiting for an image in frame_buf
  - Digitizer should let tracker know when it adds an image to frame_buf

Add a condition variable

- Condition variable functions
  - pthread_cond_wait: block for a signal
  - pthread_cond_signal: signal one waiting thread
  - pthread_cond_broadcast: signal all waiting threads
- Semantics (OS invariants)
  - pthread_cond_wait (pthread_cond_t **c**, pthread_mutex_t **m**)
    - Atomically release mutex **m**
    - Put thread to sleep waiting on a signal to pthread_cond_t **c**
    - Atomically re-lock mutex **m** on awakening

Say we have 3 threads, T1-T3 that all wait on cond_var c.

This is what the OS does:

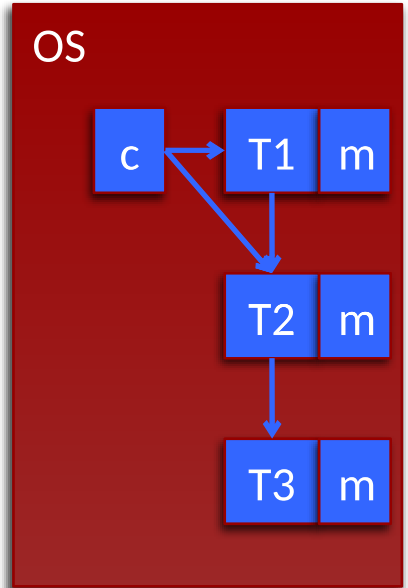


Add a condition variable

- Condition variable functions
  - pthread_cond_wait: block for a signal
  - pthread_cond_signal: signal one waiting thread
  - pthread_cond_broadcast: signal all waiting threads

- Semantics (OS invariants)
  - pthread_cond_signal (pthread_cond_t **c**)
    - Wake up one thread waiting on pthread_cond_t **c**
    - [The signaled thread will then go on to reclaim its mutex before proceeding.]

We have our 3 threads, T1-T3 all waiting on cond_var c when signal is called.



Condition variable

- We use condition variables to avoid busy waiting before entering critical sections
- They are a method of inter-process (or inter-thread) communication
- Condition variables *represent* a particular condition involving shared data, but despite their name, they don't actually test for it
- **We** must actually write the code to test for the condition
- And we must make sure our code doesn't enter the critical section until the condition is true
- Since we can't enter the critical section if the condition is false, we can be certain that **we** can't make the condition true; some other thread must do it
- We depend on a notification from the code in another thread that can **make** the condition true to wake us up when the condition is true!
- You will hear us call this condition an *invariant* for entering the critical section

Back to our surveillance app

- What were we waiting for?
  - Digitizer waits for frame_buf to be not full
  - Tracker waits for frame_buf to be not empty

Fix number 4 – cond_var

```
pthread_cond_t buf_not_full, buf_not_empty;
digitizer()
{
  image_type dig_image;
  int tail = 0;
  loop {
    grab(dig_image);
    pthread_mutex_lock(buflock);
    if (bufavail == 0)
      pthread_cond_wait(buf_not_full,
           buflock);
    pthread_mutex_unlock(buflock);
    frame_buf[tail] = dig_image;
    tail = (tail + 1) % MAX;
    pthread_mutex_lock(buflock);
    bufavail = bufavail - 1;
    pthread_cond_signal(buf_not_empty);
    pthread_mutex_unlock(buflock);
  }
}
```

```
tracker()
{
  image_type track_image;
  int head = 0;
  loop {
    pthread_mutex_lock(buflock);
    if (bufavail == MAX)
      pthread_cond_wait(buf_not_empty,
buflock);
    pthread_mutex_unlock(buflock);
    track_image = frame_buf[head];
    head = (head + 1) % MAX;
    pthread_mutex_lock(buflock);
    bufavail = bufavail + 1;
    pthread_cond_signal(buf_not_full);
    pthread_mutex_unlock(buflock);
    analyze(track_image);
  }
}
```

Recall: Mutex locks

T1:

Lock(m)
Critical section
Unlock(m)

T2:

Lock(m)
Critical section
Unlock(m)

Time

Both T1 and T2 execute

T1 gets lock, T2 is stuck

CS (T1)

T1 unlocks, T2 gets lock

CS (T2)

T2 unlocks

What's T1 doing?

It continues its execution

Invariant:

- Exactly one thread at a time in a given critical section

?

More than one critical section?

T1:

Lock(m)
Critical section
Unlock(m)

T2:

Lock(m)
Critical section
Unlock(m)

T3:

Lock(n)
Critical section
Unlock(n)

T4:

Lock(n)
Critical section
Unlock(n)

Different locks!
Thus these two critical sections can run at the same time!

OS Invariant:

- Exactly one thread at a time in a given critical section
- But different mutex locks create ***different*** critical sections?
- But still at most one thread can run in each!

Recall: Condition variables

T1:

```
Lock(m)
  If ! Pred
    Cond_wait(c, m)
  Pred = False
Unlock(m)
```

T2:

```
Lock(m)
  Pred = True
  Signal(c)
Unlock(m)
```

T1 gets lock m

T1 blocks

- OS releases lock m

T2 gets lock m

T1 ready to run

T2 unlocks

- OS gets lock m on T1s behalf

T1 runs

Invariant:

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

Who must ensure this?

The programmer!

Back again to our surveillance app

- What were we waiting for?
  - Digitizer waits for frame_buf to be not full
    - Predicate is (bufavail != 0)
  - Tracker waits for frame_buf to be not empty
    - Predicate is (bufavail != MAX)
- So we need two condition variables
  - buf_not_empty and buf_not_full
  - And we know how to test for these conditions using the predicates

Fix number 4 – cond_var

```
pthread_cond_t buf_not_full, buf_not_empty;
digitizer()
{
  image_type dig_image;
  int tail = 0;
  loop {
    grab(dig_image);
    pthread_mutex_lock(buflock);
      if (bufavail == 0)
        pthread_cond_wait(buf_not_full,
              buflock);
    pthread_mutex_unlock(buflock);
    frame_buf[tail] = dig_image;
    tail = (tail + 1) % MAX;
    pthread_mutex_lock(buflock);
      bufavail = bufavail - 1;
      pthread_cond_signal(buf_not_empty);
    pthread_mutex_unlock(buflock);
  }
}
```

```
tracker()
{
  image_type track_image;
  int head = 0;
  loop {
    pthread_mutex_lock(buflock);
      if (bufavail == MAX)
        pthread_cond_wait(buf_not_empty,
buflock);
    pthread_mutex_unlock(buflock);
    track_image = frame_buf[head];
    head = (head + 1) % MAX;
    pthread_mutex_lock(buflock);
      bufavail = bufavail + 1;
      pthread_cond_signal(buf_not_full);
    pthread_mutex_unlock(buflock);
    analyze(track_image);
  }
}
```

Invariants: Only 1 thread at a time in CS
bufavail != 0

Only 1 thread at a time in CS
bufavail != MAX

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

T1:

```
Lock(m)
  If ! Pred
    Cond_wait(c, m)
  Pred = False
Unlock(m)
```

T2:

```
Lock(m)
  Pred = True
  Signal(c)
Unlock(m)
```

The first is here

The second is here

But there is a third!

This is where the programmer comes in

Where is that third block?

T1:

```
Lock(m)
  If ! Pred
    Cond_wait(c, m)
  Pred = False
Unlock(m)
```

T2:

```
Lock(m)
  Pred = True
  Signal(c)
Unlock(m)
```

It's hiding here in plain sight

- Lock(m)
- If ! Pred
      Unlock(m)
  Wait until c is signaled
  Lock(m)
    Pred = false
- Unlock(m)

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

This detail becomes a problem if there are more threads…

Fix number 5 – Defensive programming

Defense:
Re-check
invariants

```
digitizer()
{
  image_type dig_image;
  int tail = 0;
  loop {
    grab(dig_image);
    pthread_mutex_lock(buflock);
    while (bufavail == 0)
      pthread_cond_wait(buf_not_full, buflock);
    pthread_mutex_unlock(buflock);
    frame_buf[tail] = dig_image;
    tail = (tail + 1) % MAX;
    pthread_mutex_lock(buflock);
    bufavail = bufavail - 1;
    pthread_cond_signal(buf_not_empty);
    pthread_mutex_unlock(buflock);
  }
}
```

```
tracker()
{
  image_type track_image;
  int head = 0;
  loop {
    pthread_mutex_lock(buflock);
    while (bufavail == MAX)
      pthread_cond_wait(buf_not_empty, buflock);
    pthread_mutex_unlock(buflock);
    track_image = frame_buf[head];
    head = (head + 1) % MAX;
    pthread_mutex_lock(buflock);
    bufavail = bufavail + 1;
    pthread_cond_signal(buf_not_full);
    pthread_mutex_unlock(buflock);
    analyze(track_image);
  }
}
```

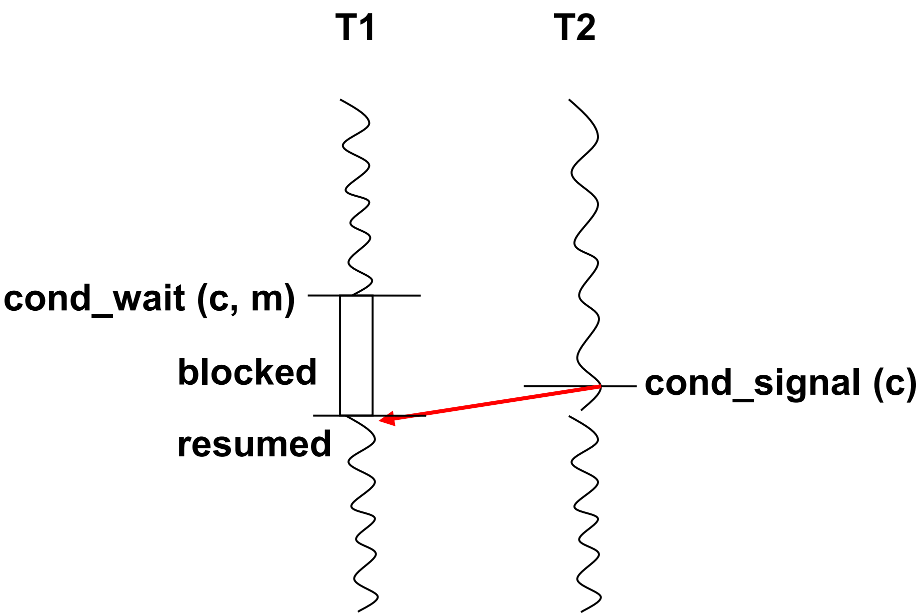
The notion of re-checking a flag after waiting is important.

A condition variable...

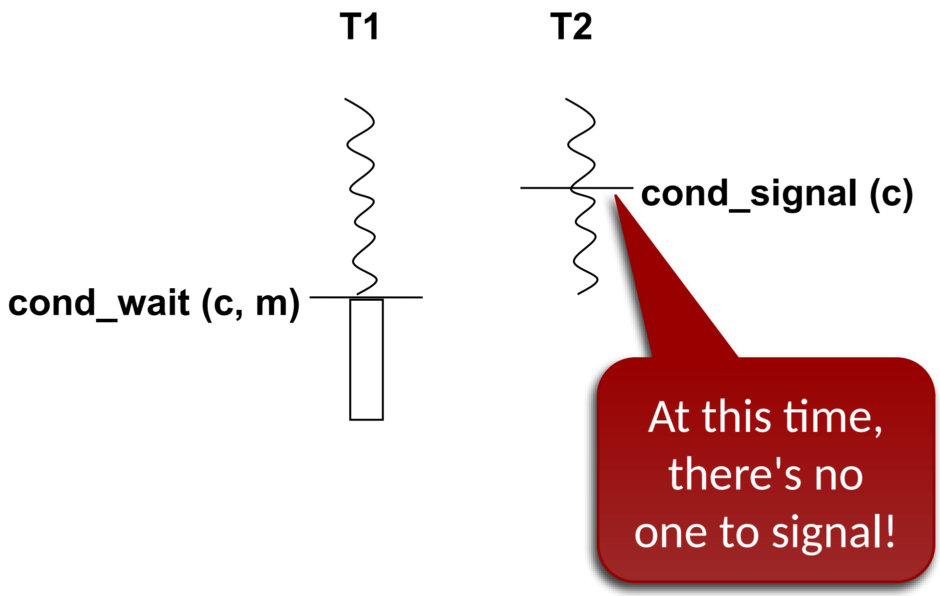
A. ... is just another name for a mutex lock
B. ...enables a thread to wait for a condition to become true without consuming processor cycles
C. ...enables a thread to enter a critical section
D. ...none of the above

Today's number is 27,045

Gotchas in programming with cond vars



(a) Wait before signal



(b) Wait after signal (T1 blocked forever)

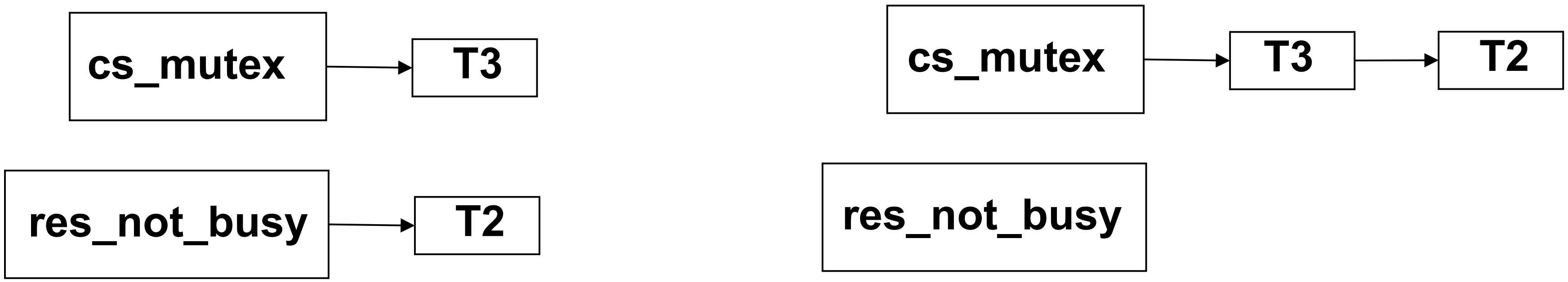
Gotchas in programming with cond vars

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

```
acquire_shared_resource()
{
 pthread_mutex_lock(cs_mutex);   <-- T3 is here
   if (res_state == BUSY)
     pthread_cond_wait (res_not_busy, cs_mutex);   <-- T2 is here
   res_state = BUSY;
 pthread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
 pthread_mutex_lock(cs_mutex);
   res_state = NOT_BUSY;
   pthread_cond_signal(res_not_busy);   <-- T1 is here
 pthread_mutex_unlock(cs_mutex);
}
```

State of waiting queues



(a) Waiting queues before T1 signals

(a) Waiting queues after T1 signals

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

```
acquire_shared_resource()
{
 pthread_mutex_lock(cs_mutex);        <-- T3 is here
   if (res_state == BUSY)
     pthread_cond_wait (res_not_busy, cs_mutex);        <-- T2 is here
   res_state = BUSY;
 pthread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
 pthread_mutex_lock(cs_mutex);
   res_state = NOT_BUSY;
   pthread_cond_signal(res_not_busy);        <-- T1 is here
 pthread_mutex_unlock(cs_mutex);
}
```

T2 then locks the mutex

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

Why did this happen?

- We violated invariants…

```
acquire_shared_resource()
{
 pthread_mutex_lock(cs_mutex);
   if (res_state == BUSY)
     pthread_cond_wait (res_not_busy, cs_mutex);
   res_state = BUSY;
 pthread_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 waked up
    - Documented behavior in Linux
    - Turns out to be very hard to avoid this in the kernel
  - Upshot: Defensive programming

Gotchas– retest predicate

```
acquire_shared_resource()
{
 pthread_mutex_lock(cs_mutex);
   while (res_state == BUSY)
     pthread_cond_wait (res_not_busy, cs_mutex);
   res_state = BUSY;
 pthread_mutex_unlock(cs_mutex);
}

release_shared_resource()
{
 pthread_mutex_lock(cs_mutex);
   res_state = NOT_BUSY;
   pthread_cond_signal(res_not_busy);
 pthread_mutex_unlock(cs_mutex);
}
```

Replace the "if" with a "while"

Make T2 recheck predicate

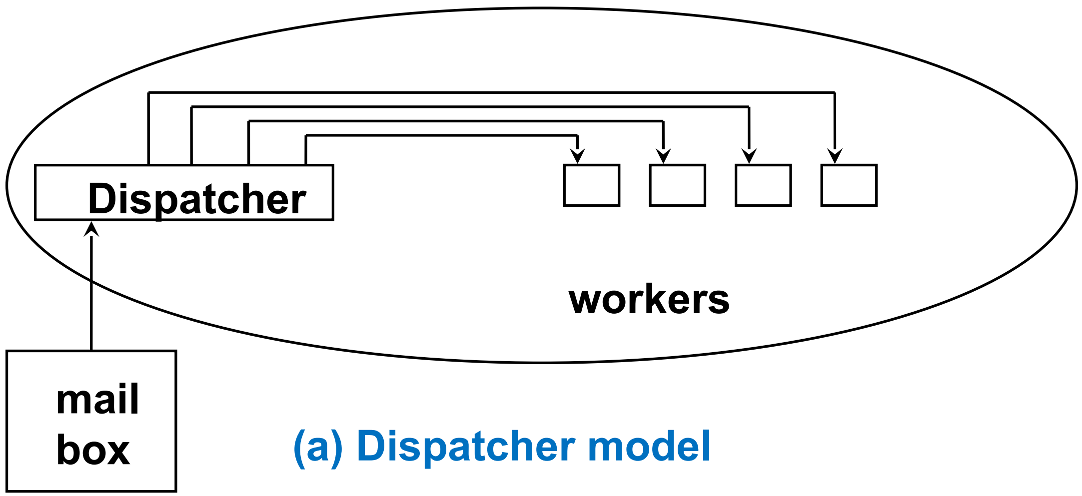
Avoids the "race condition"

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

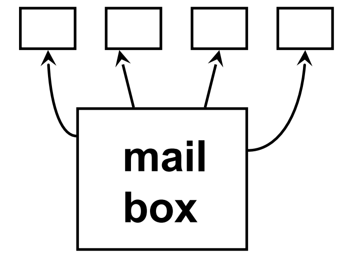
Checkpoint

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

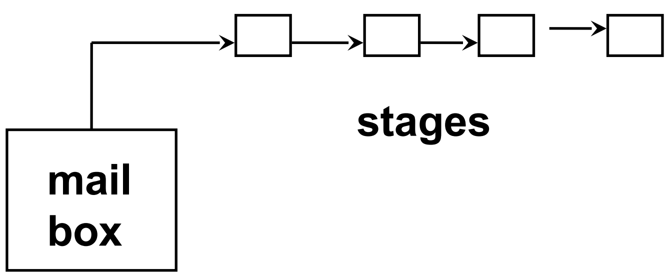
Threads as software structuring abstractions



(a) Dispatcher model



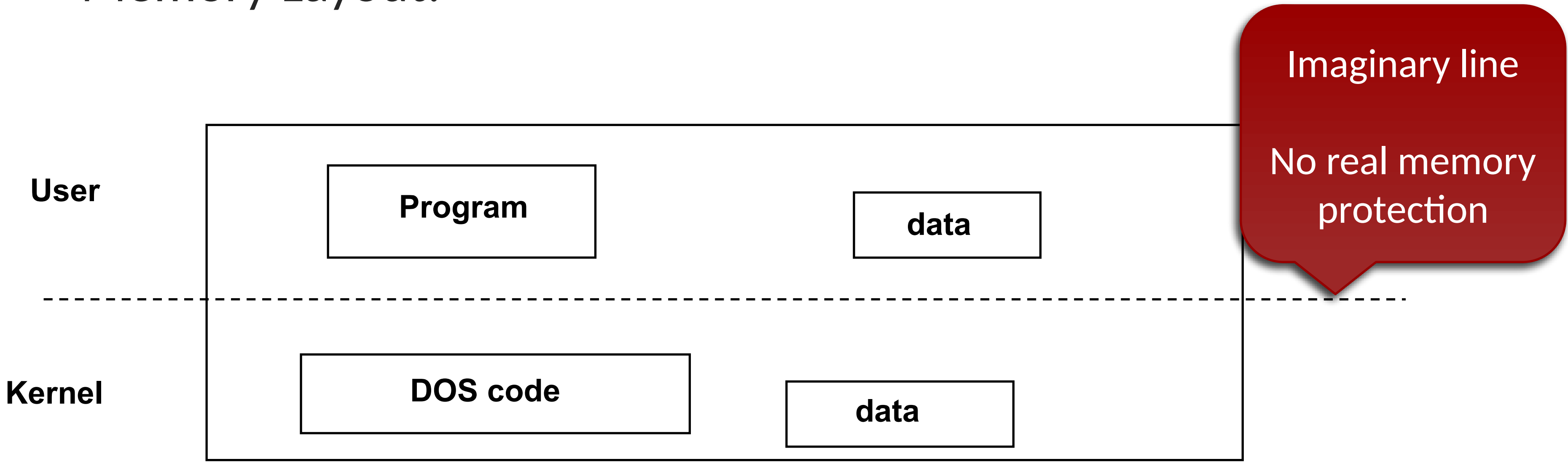
(b) Team model



(c) Pipelined model

Traditional OS: DOS

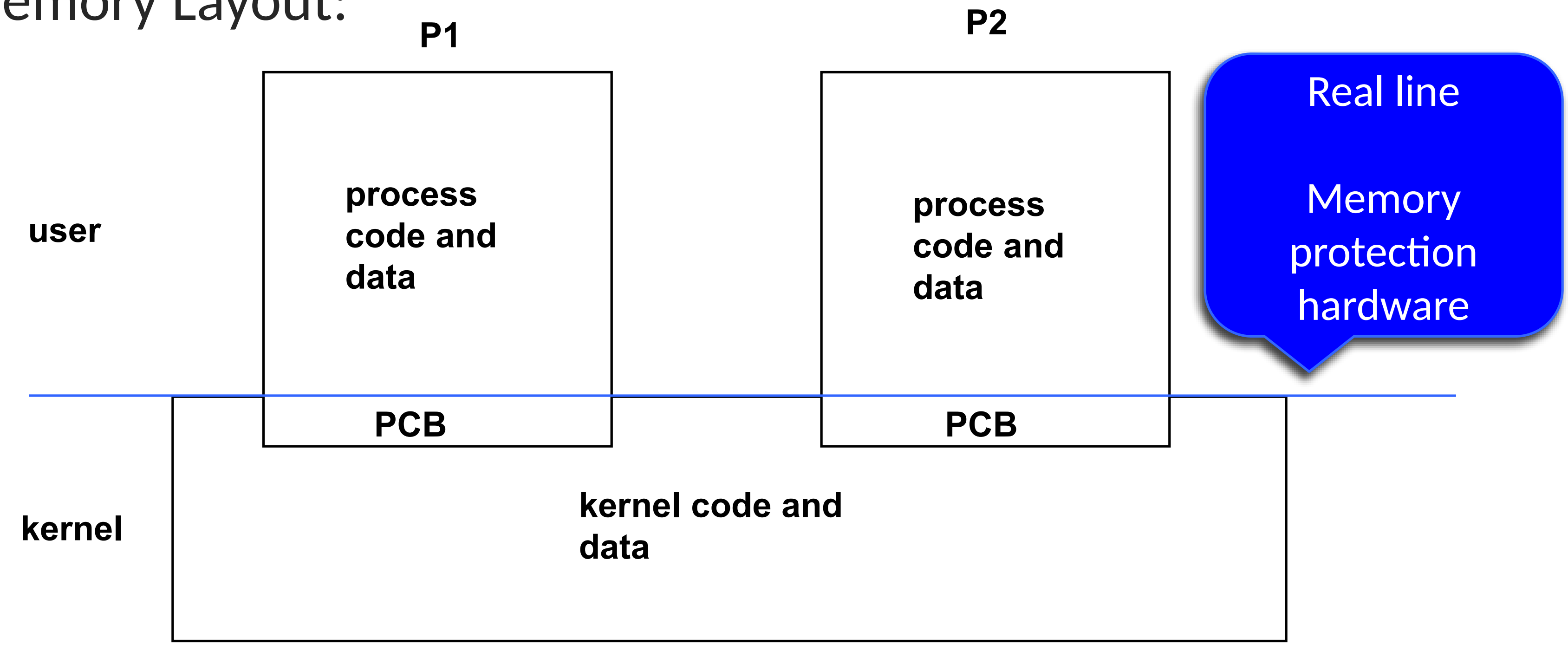
Memory Layout:



- Protection between user and kernel?
- Single process, single thread

Traditional OS: Unix

Memory Layout:



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

Tradition

- Programs in these 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

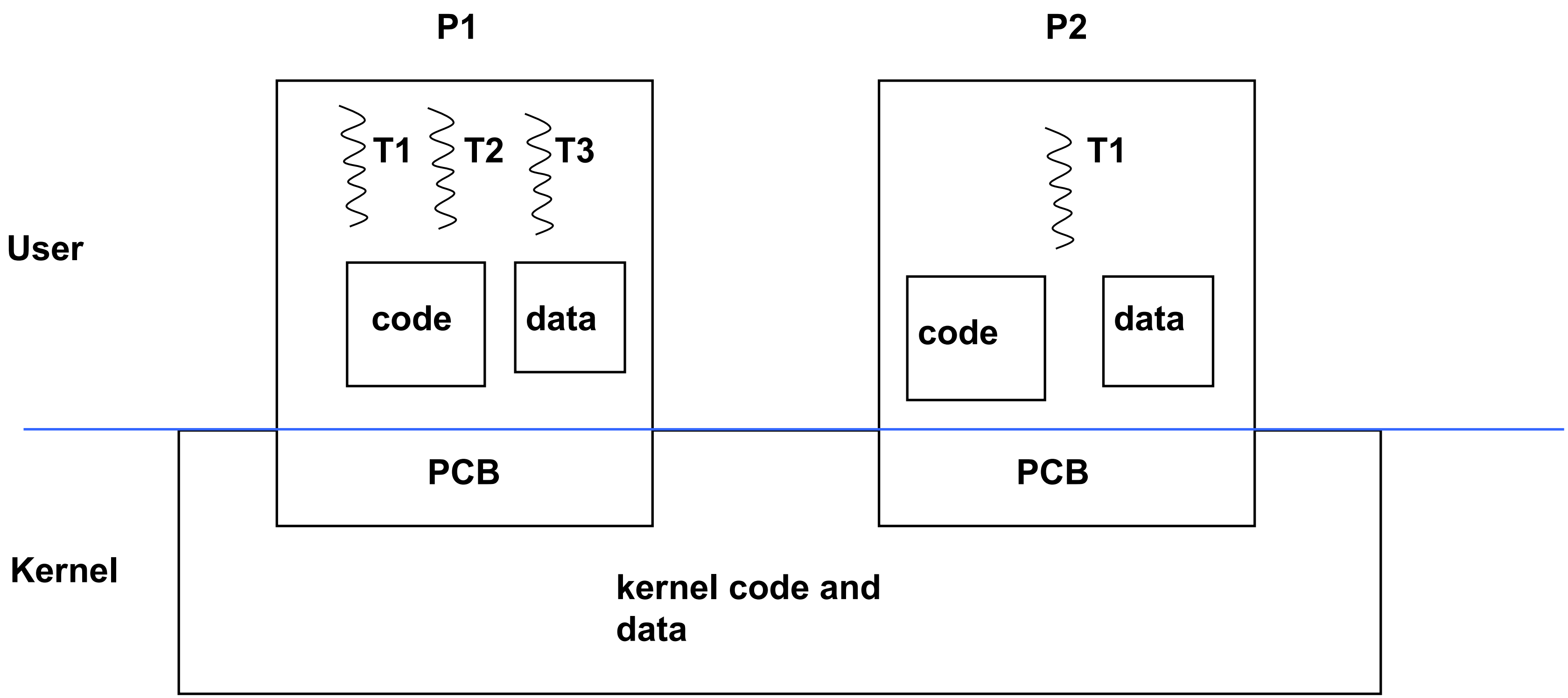
How widespread is support for threads in operating systems?

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

Process Vs. Thread?

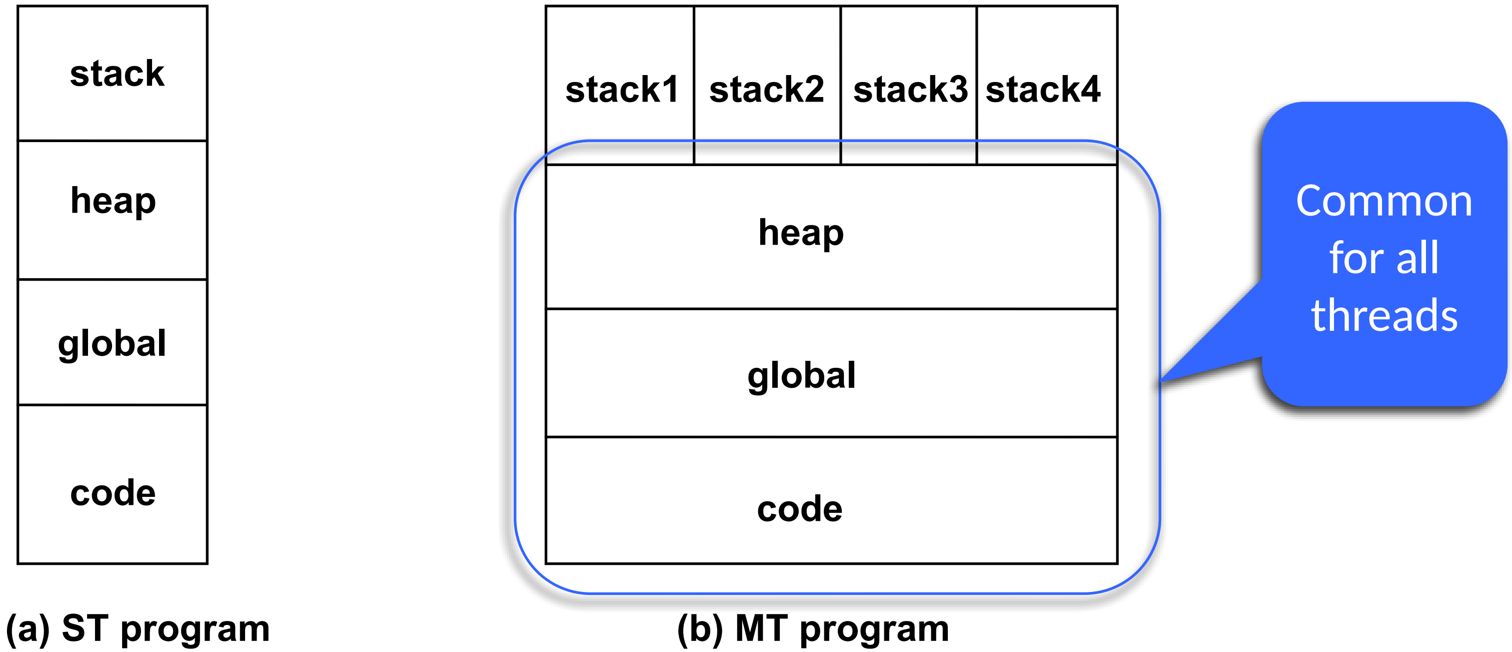
- In a single threaded program, the state of the executing program is contained in a process
- In a MT program, the state of the executing program is contained in several 'concurrent' threads

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



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

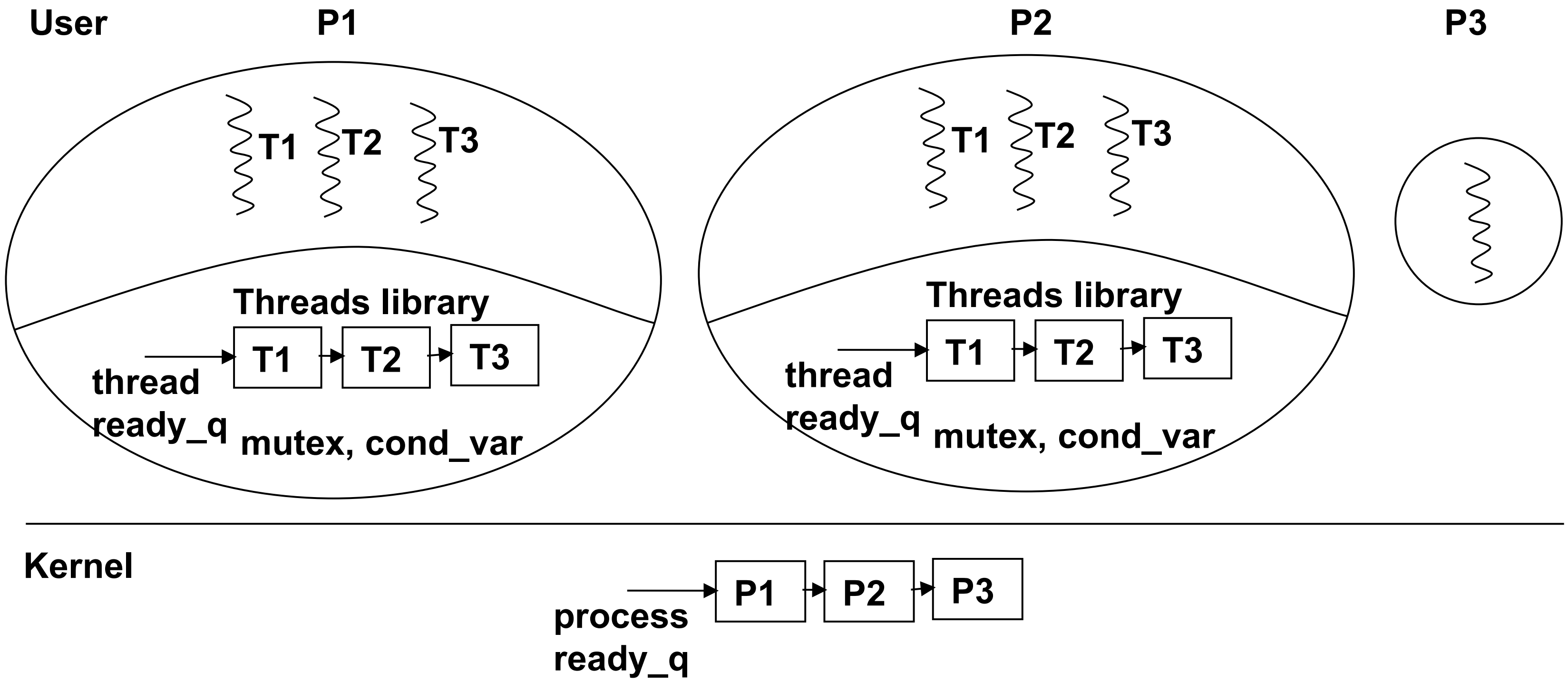
Thread properties

- Threads
  - Share address space of process
  - Cooperate to get job done
- Are threads concurrent?
  - Maybe if the box is a true multiprocessor
  - Share the same CPU on a uniprocessor
- Is threaded code different from non-threaded?
  - Protection for data shared among threads
  - Synchronization among threads

User-Level Threads

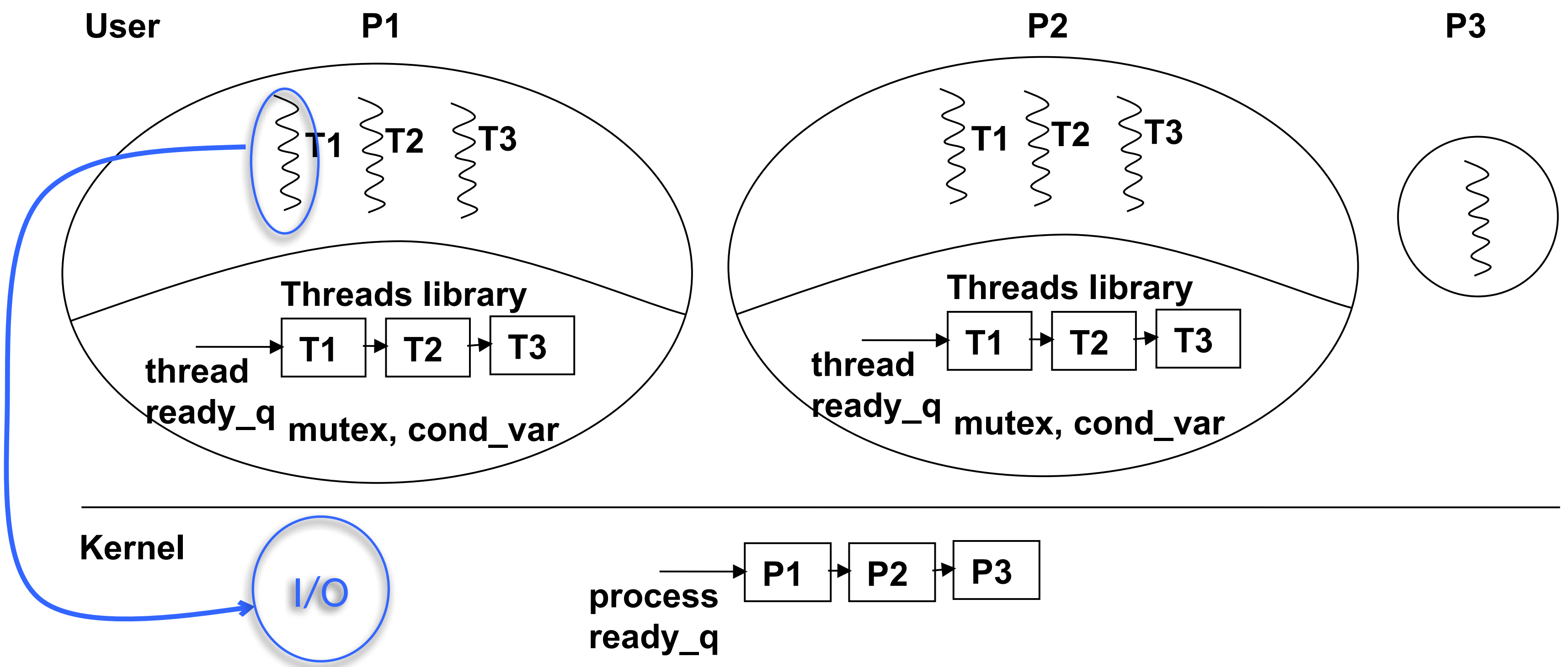
- OS independent
- Scheduler is part of the user space runtime system
- Thread switching is cheap (just save PC, SP, regs)
- Scheduling is customizable, i.e., more app control
- Blocking call by a thread blocks a process

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…

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

Today's number is 31,956

User-level threads with process level scheduling…

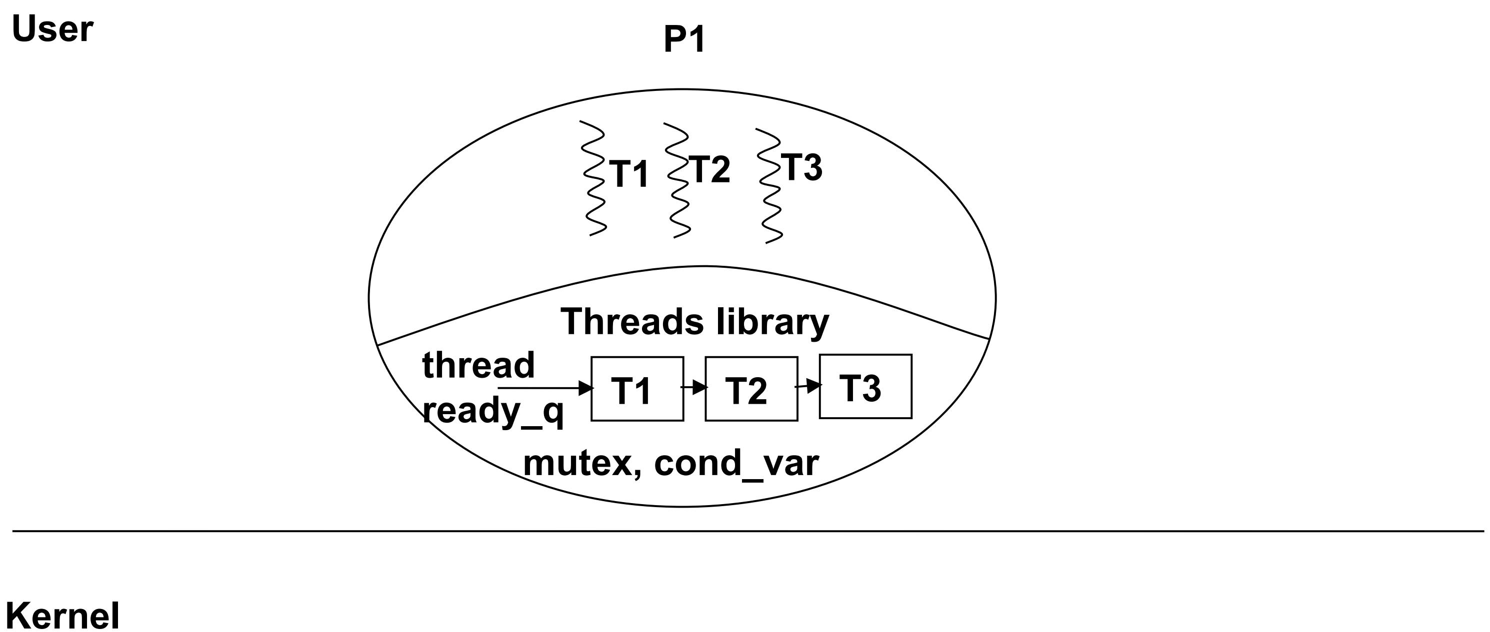
A. …can take advantage of the hardware concurrency available in a multiprocessor

B. …is impossible to implement on a true multiprocessor

C. …will have no performance advantage on a multiprocessor compared to a uniprocessor

D. None of the above

User-level threads

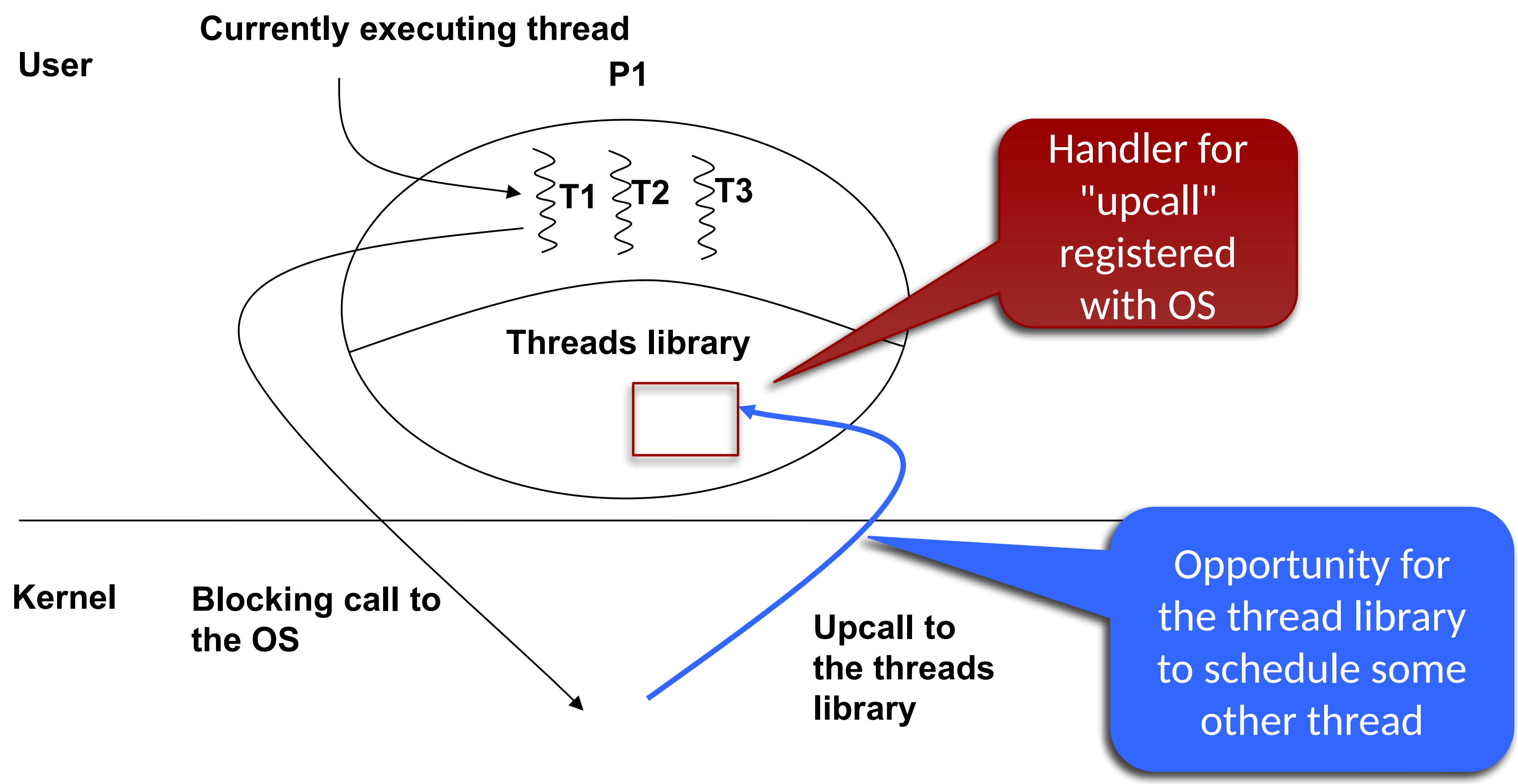


- Switching among user-level threads
  - Yield voluntarily
  - How can we make them preemptive?
    - Use timer interrupts from kernel to switch

User-level threads

- Solutions to blocking problem in user-level threads
  - Implement a non-blocking version of all system calls
    - Not particularly feasible
  - Polling wrapper in the scheduler for such system calls
    - All blocking system calls go through the thread library
    - Thread library queues the system calls and will issue them to the OS only if it has no runnable threads
  - OS support to deal with blocking I/O
    - How might we do this?...

OS support for user-level threads blocking calls



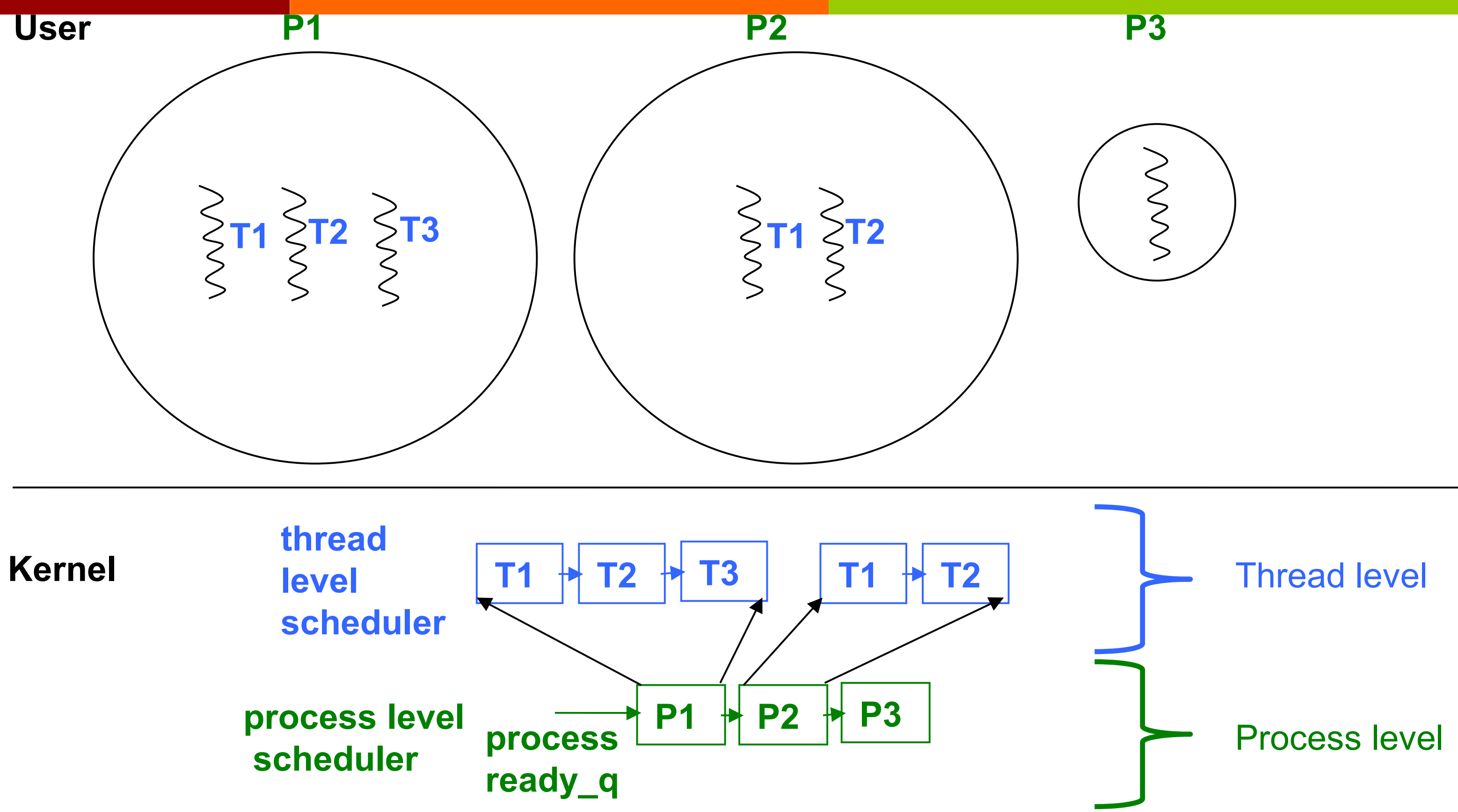
Kernel-level threads

- 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

Tw0-level OS scheduler



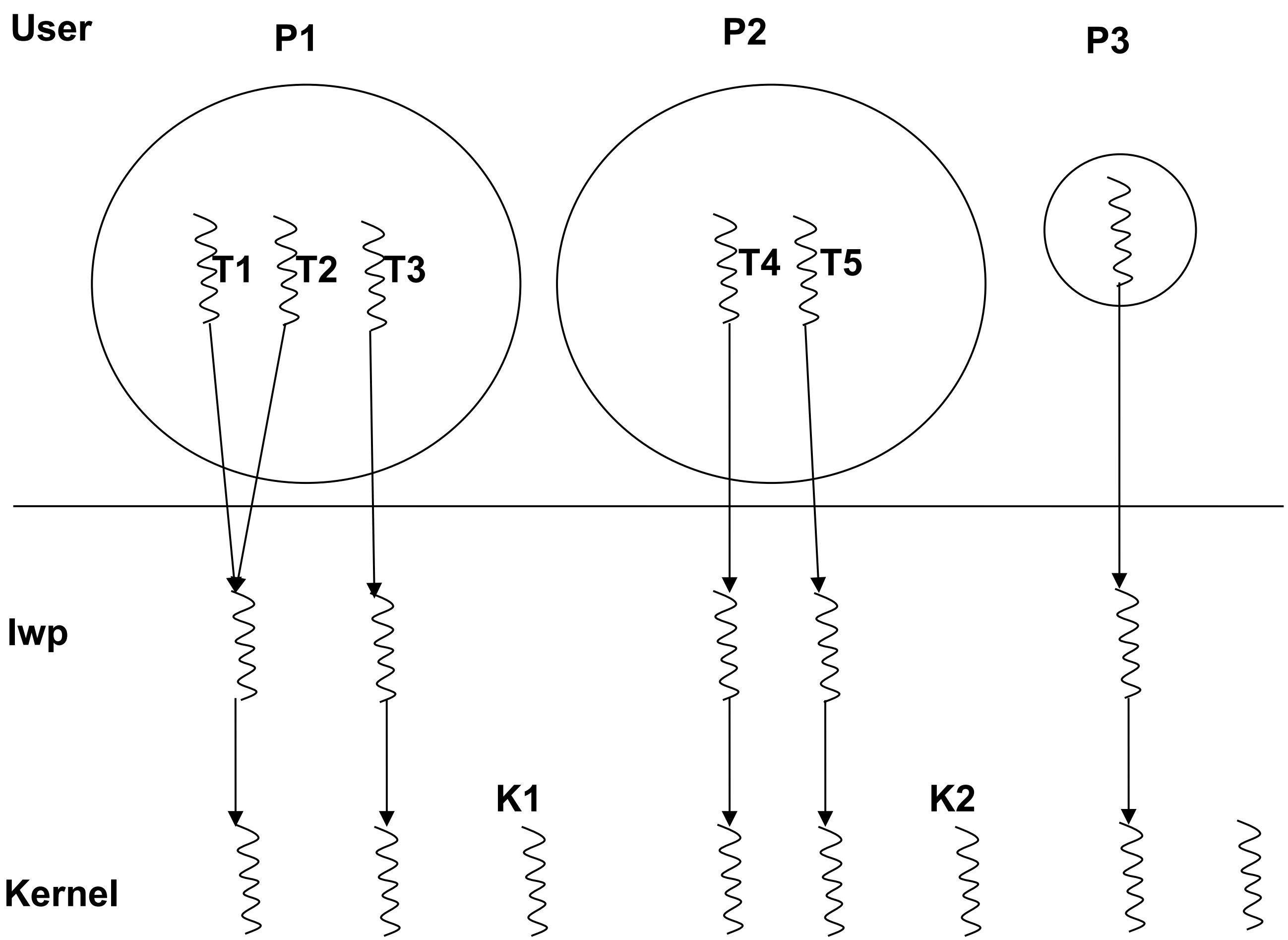
- Threading in the application is visible to the OS
- OS provides the thread library

Example: Solaris Threads

- Three kinds of threads
  - user, lwp (light-weight process), kernel
- User thread: any number can be created and attached to lwp's
- One to one mapping between lwp and kernel threads (each lwp is mapped to exactly one kernel thread)
- Only kernel threads are known to the OS scheduler
- If a kernel thread blocks, the associated lwp, and user-level threads block as well

Solaris threads

- The unit of scheduling is the kernel thread; user-level threads are scheduled by the user space threads library
- T1 and T2 operate as co-routines with lowest thread-switching cost
- T3 can run concurrently with T1 or T2, but incurs the higher lwp switching cost.
- T4 and T5 can also run concurrently, but pay a higher switching cost with T3 because they are switch address spaces too
- K1 and K2 are kernel threads and don't have to change address spaces when switching to each other



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 for such library calls

Thread safe libraries

```
/* original version */


void *_malloc(size_t size)
{




  ......
  ......


  return(memory_pointer);
}
```

```
/* thread safe wrapper */

mutex_lock_type m_mutex;
void *malloc(size_t size)
{
void *p, *_malloc(size_t);
pthread_mutex_lock(m_mutex);

p = _malloc(size);

pthread_mutex_unlock(m_mutex);

return p;
}
```

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;

  Unlock():

  mem_lock = 0

- What could go wrong?

Oops! These instructions aren't atomic

How did we deal with that earlier?

We used OS mutex calls to make instructions inseparable.

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	mem_lock	R1
x	0	x

Lock() using machine instructions

Thread switch

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	mem_lock	R1
~~x~~ 0	0	x

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	mem_lock	R1
~~x~~ 0	0	~~x~~ 0

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	mem_lock	R1
~~x~~ 0	0	~~x~~ 0

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	mem_lock	R1
~~x~~ 0	0	~~x~~ ~~0~~
1		

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	mem_lock	R1
~~x~~ 0	~~0~~ 1	~~x~~ ~~0~~
1		

Lock() using machine instructions

T1	T2
`` ```	

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
```

Thread switch

R1	mem_lock	R1
~~x~~ 0	~~0~~ 1	~~x~~ ~~0~~
1		

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	mem_lock	R1
~~x~~ 0 1	~~0~~ 1	~~x~~ ~~0~~

Lock() using machine instructions

T1			T2		
Lock()			Lock()		
START	LD	R1,mem_lock	START	LD	R1,mem_lock
	BZ	SET		BZ	SET
	JSR	block_thread		JSR	block_thread
	B	START		B	START
SET	ADDI	R1,#1	SET	ADDI	R1,#1
	ST	R1,mem_lock		ST	R1,mem_lock
	RET			RET	

R1	mem_lock	R1
~~x~~ ~~0~~ 1	~~0~~ 1	~~x~~ ~~0~~
1		

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

1

mem_lock

~~0~~ ~~1~~ 1

R1

~~x~~ ~~0~~

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 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:

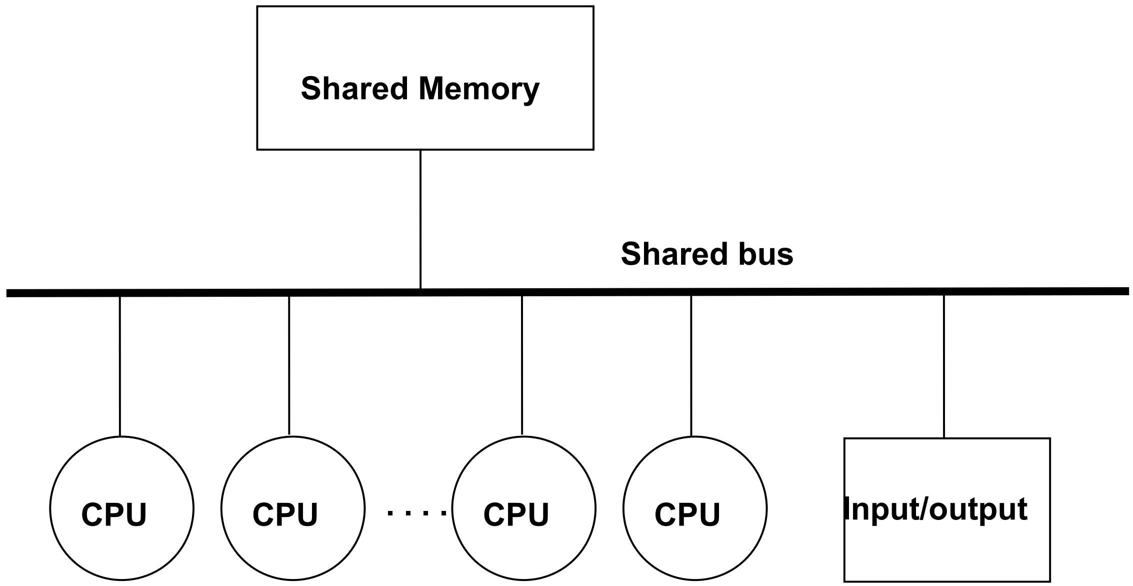
MyX = binary_semaphore(&shared_lock);

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 or 1 1 isn't possible!

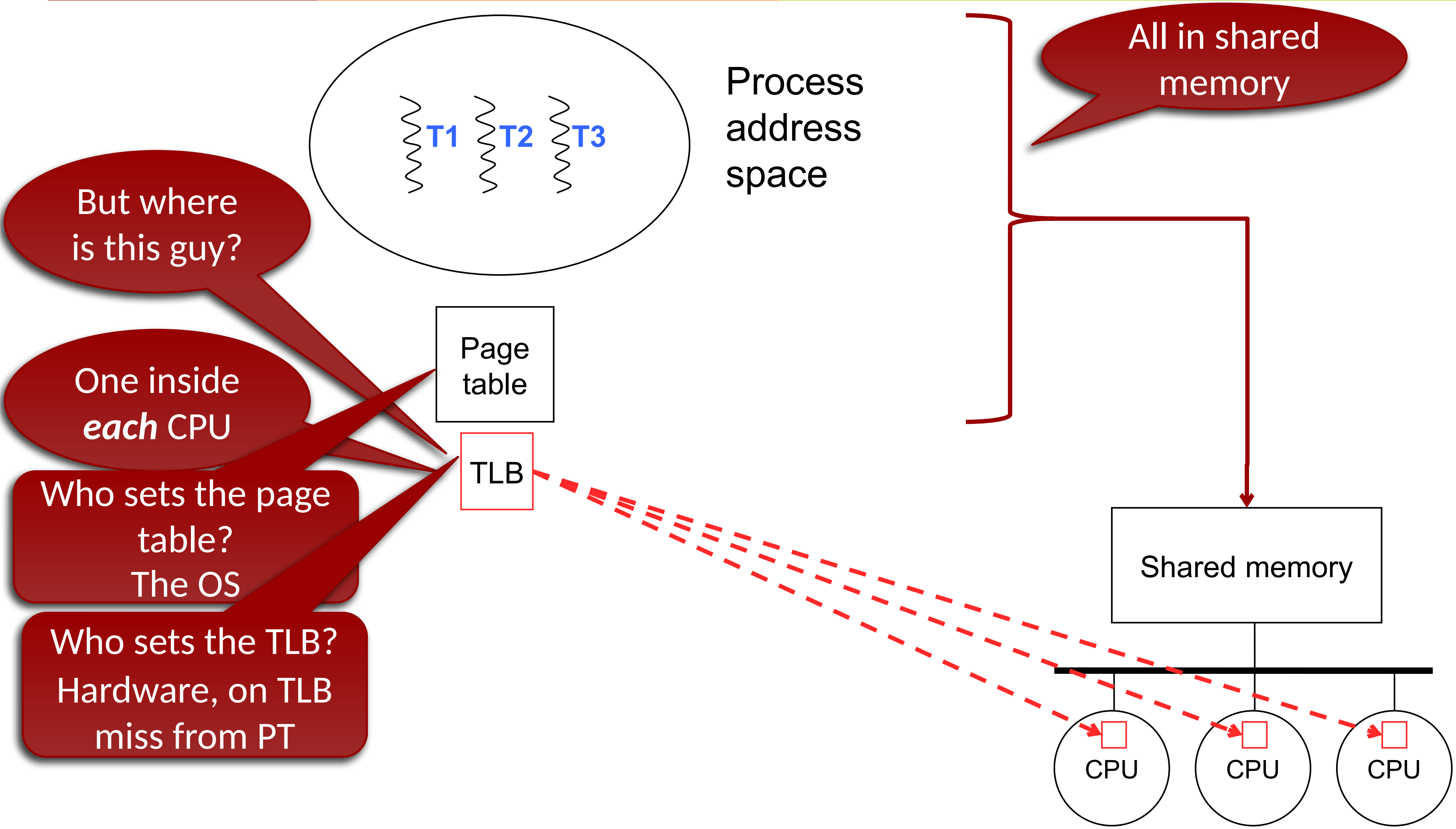
How do we implement Symmetric Multi Processing (SMP)?



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

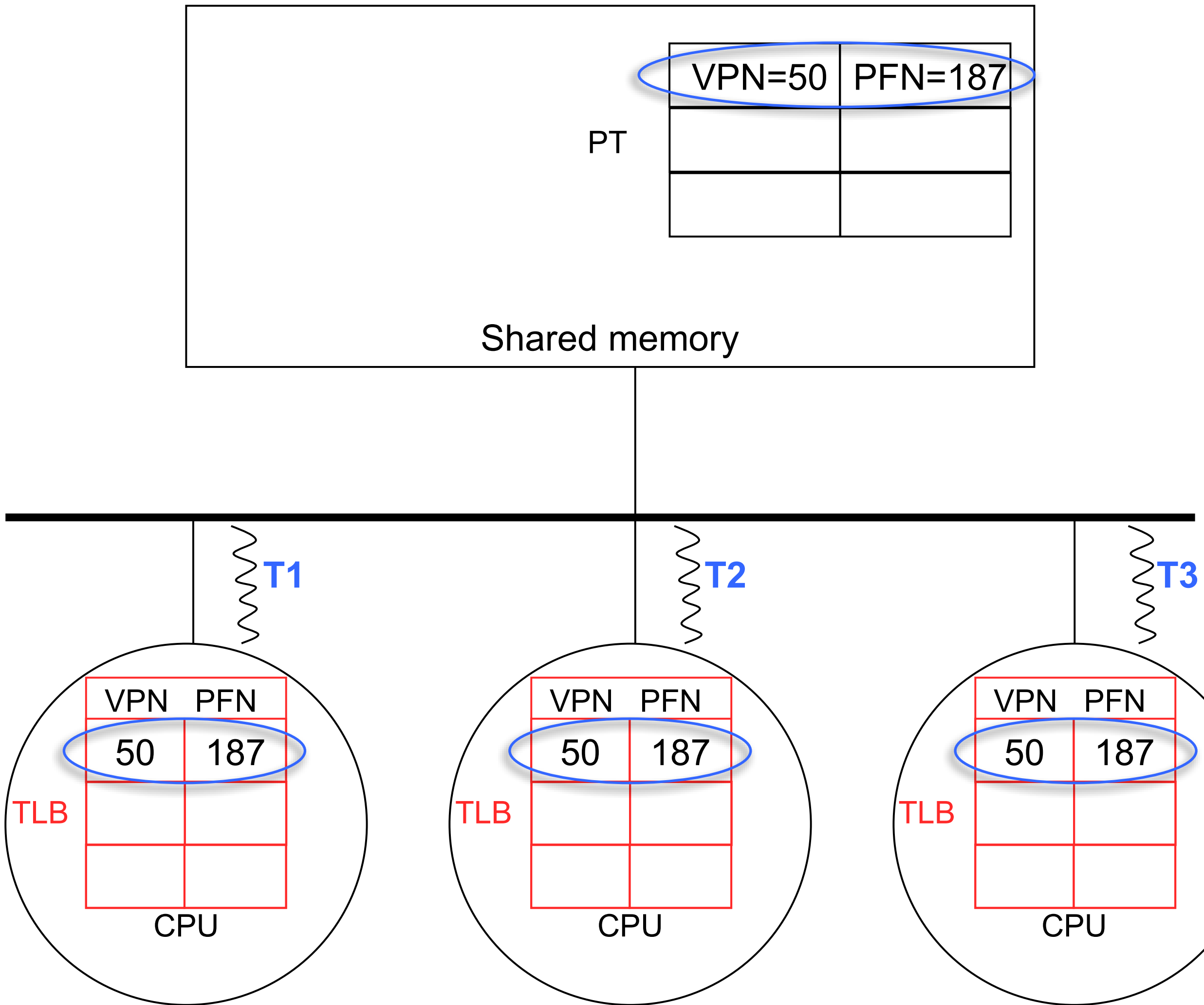
- As in the single-CPU case, the TLB must be flushed of user-space addresses on context switch
- How do multiple processors complicate this?
  - Basically, they don't
  - Any time a CPU is switched to a new thread, the OS flushes user entries from that CPU's TLB
  - There's no need to affect other TLBs

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

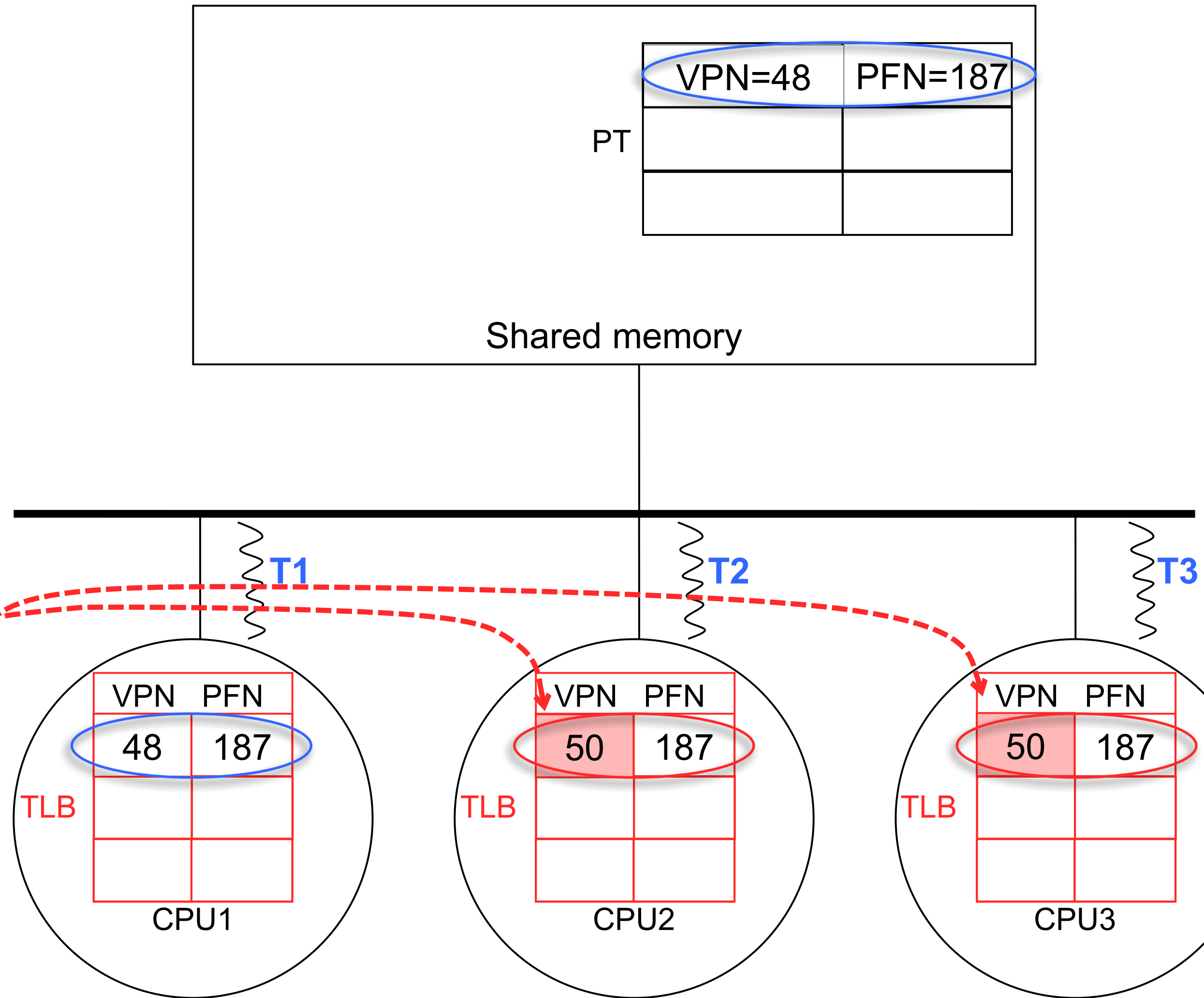
- Assume
  - T1 encounters a page fault on VPN=48
  - OS decides to evict VPN=50 from PFN=187
  - And use PFN=187 for hosting VPN=48

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 PFN 50

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

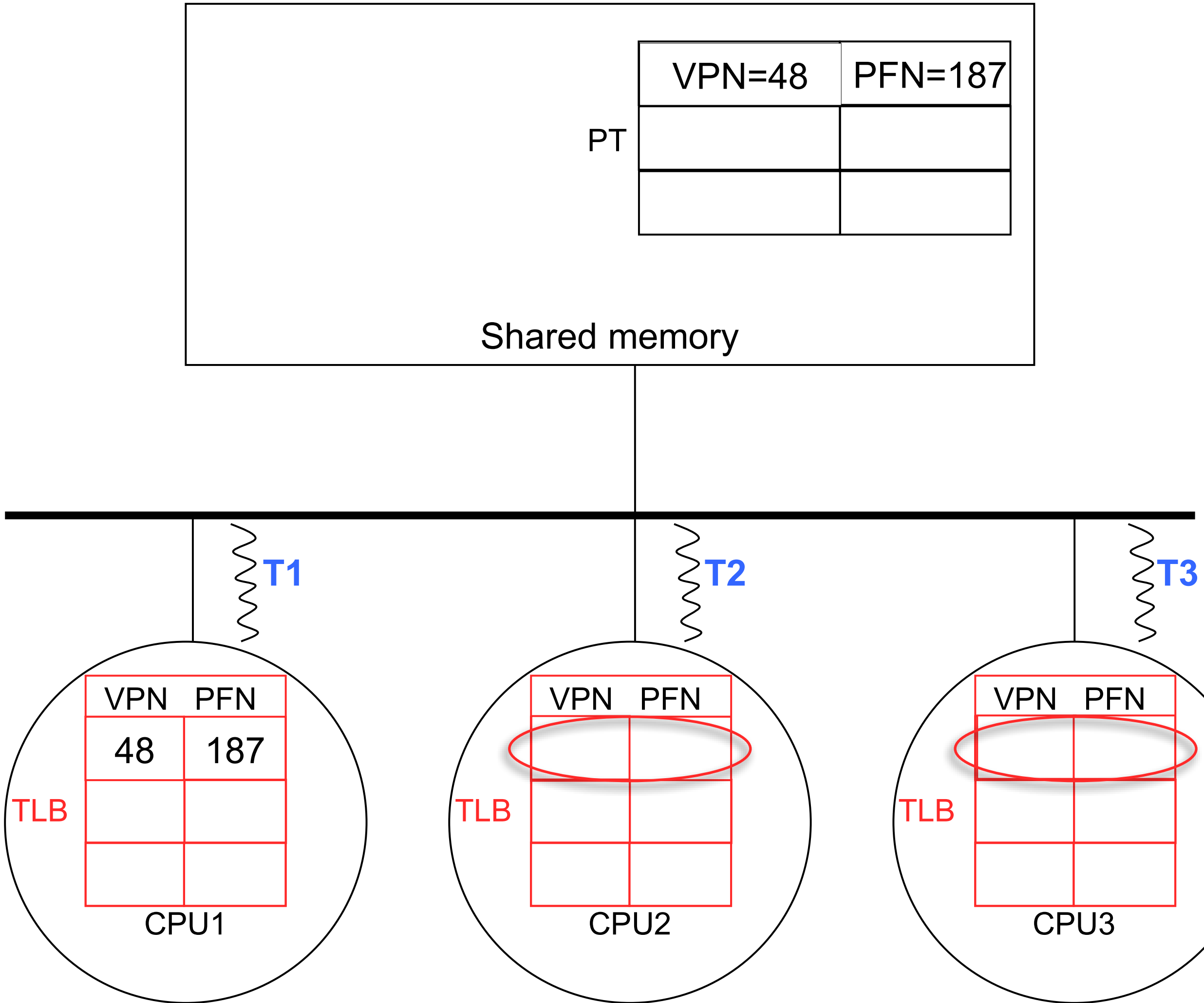


Example: Handling an SMP page fault

Then we have the TLB Shootdown

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

A. Impossible
B. Trivially achieved since the page table resides in shared memory
C. Achieved by careful replication of the page table by the operating system for each thread
D. Achieved by special-purpose hardware that no one has told us about yet
E. What is a thread?

Today's number is 43,312

Keeping the TLBs consistent in an SMP

A. Is the responsibility of the programmer
B. Is the responsibility of the hardware
C. Is the responsibility of the operating system
D. Is not possible
E. What is a TLB?

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

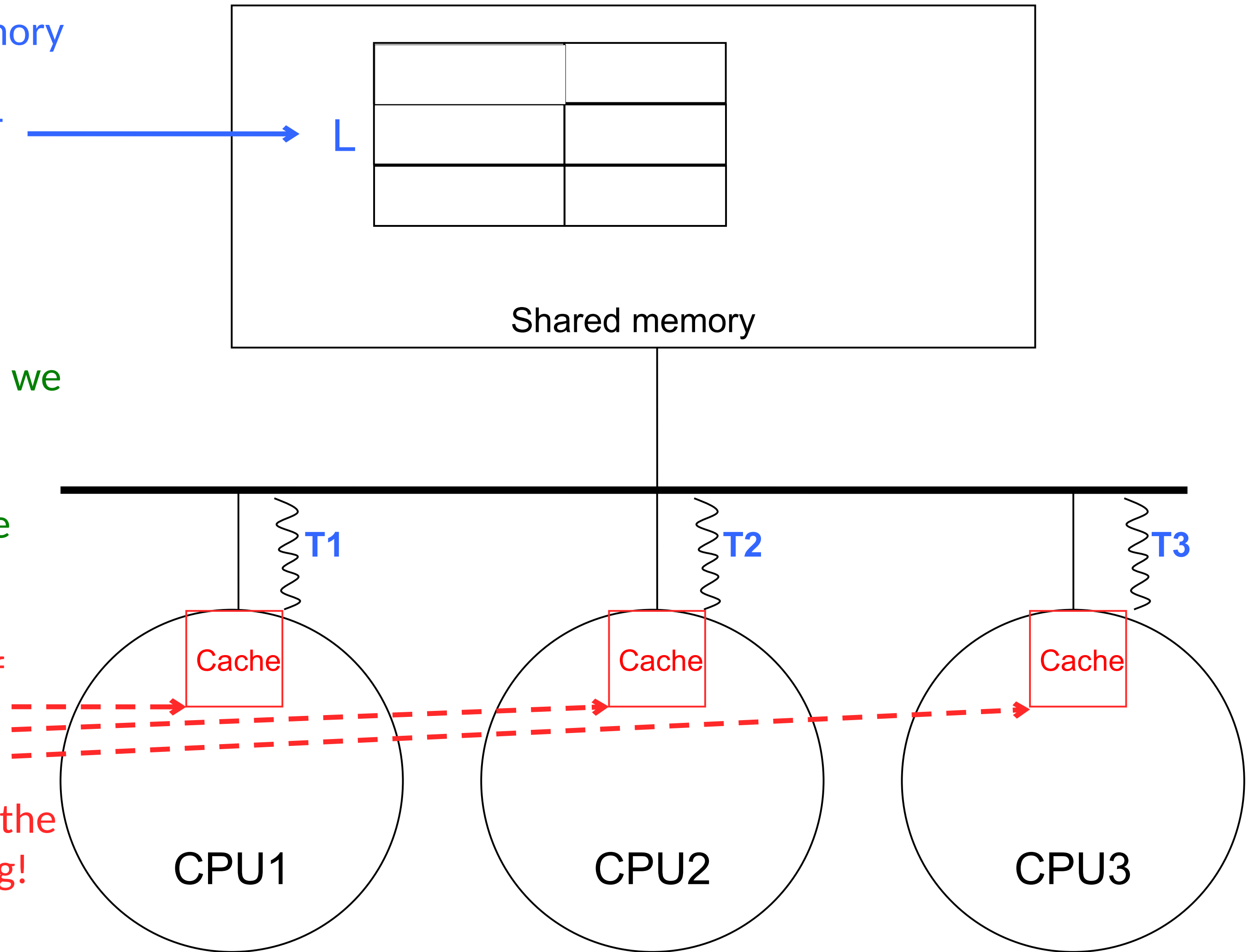
Each CPU can access it

But…what did we forget?

Where are the caches?

In the CPUs of course

If L is cached, the cache is wrong!

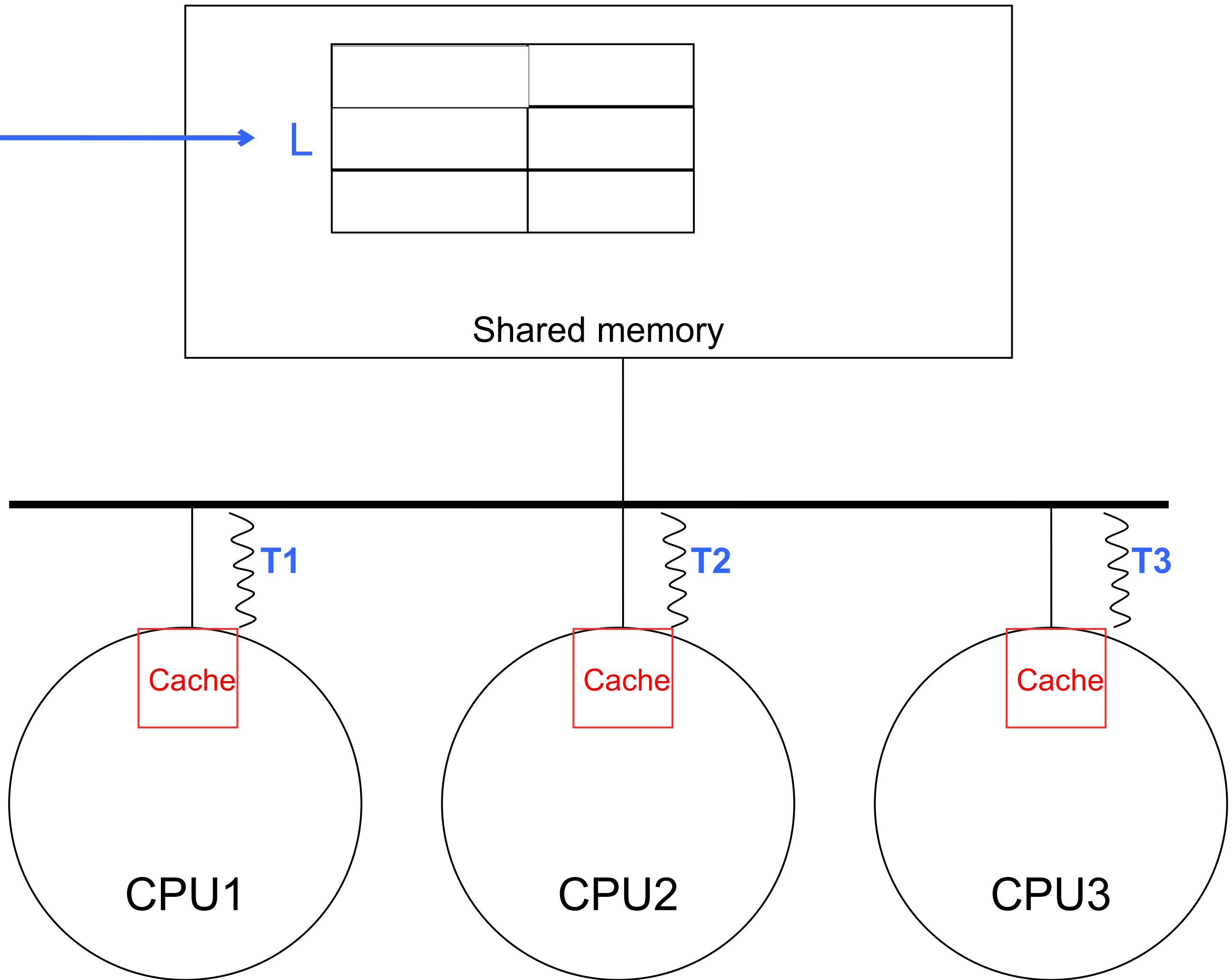


2) Threads have synchronization atomicity

What ever shall we do?

Fortunately, this one is easy

Bypass the cache for the T&S instruction

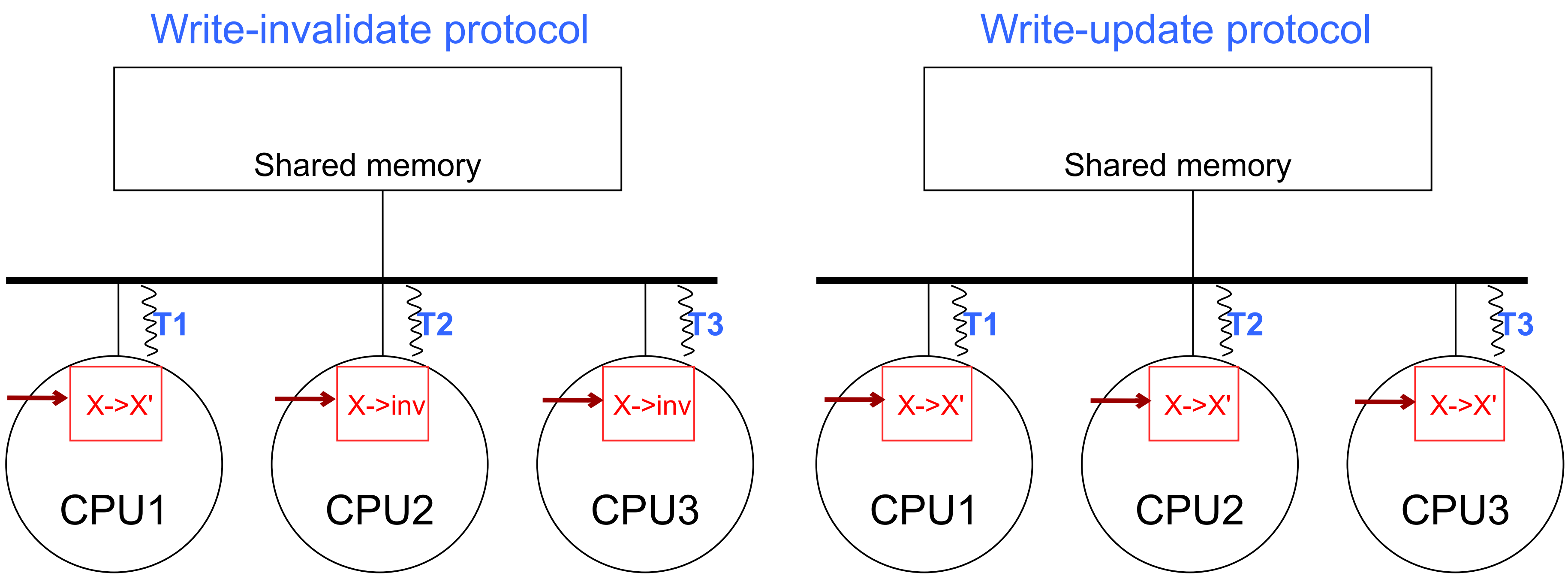


Requirements for SMP

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 synchronizing the caches as a cache consistency or cache coherency 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'



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 consistency in hardware
  - Invalidation based or update based