CS2200
Systems and Networks
Spring 2022

Lecture 21:
Parallel Systems

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

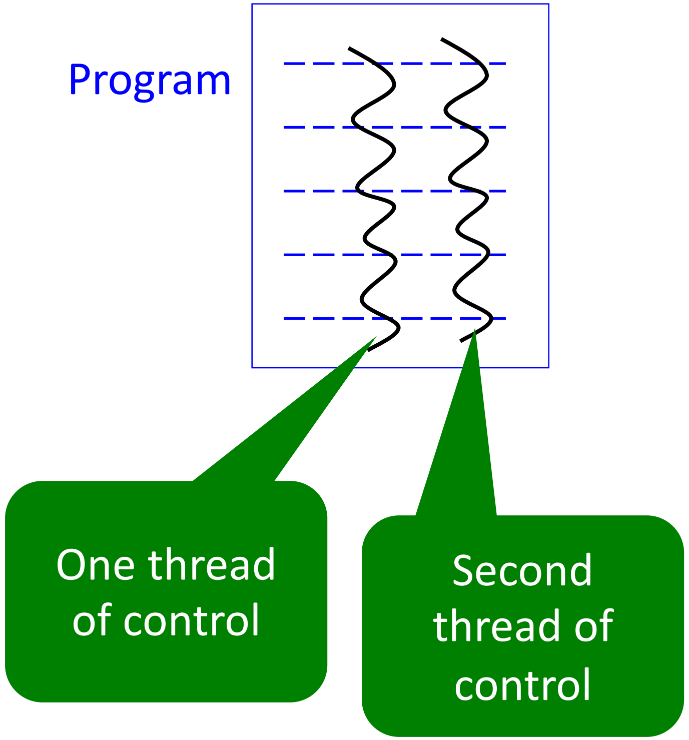
Lecture slides adapted from Bill Leahy and Charles Lively of Georgia Tech

Abstractions

- Program
  - A static image loaded into memory
- Process
  - A program in execution
- In other words, process = program + state
  - State evolves as the program executes
- Thread
  - We split the idea of state up into data (memory) and threads of control (PC and registers)
  - One memory space, but one or more contexts of control

So then, what's a thread?

Program

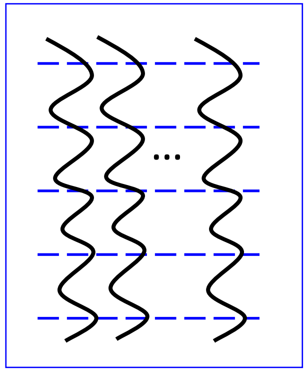


One thread of control

Second thread of control

In general:

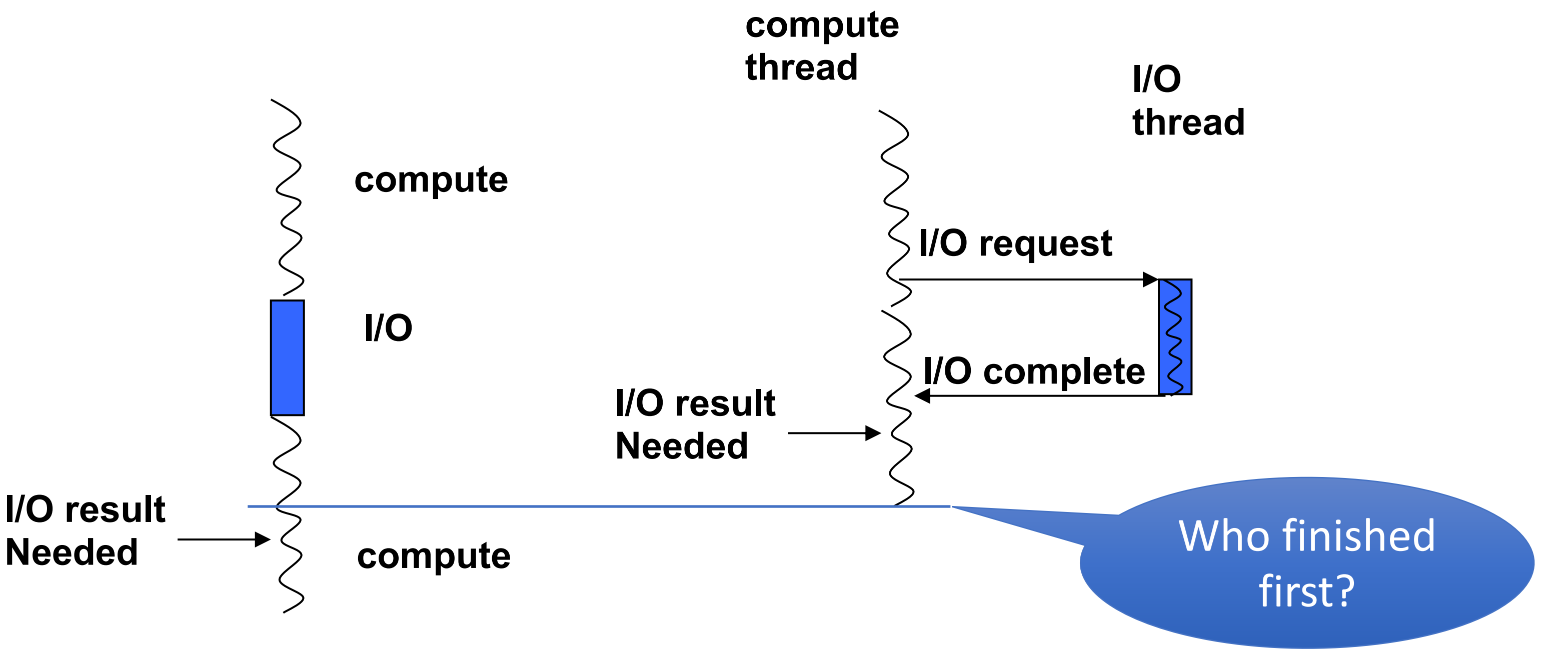
Program



n threads of control (n>0)

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

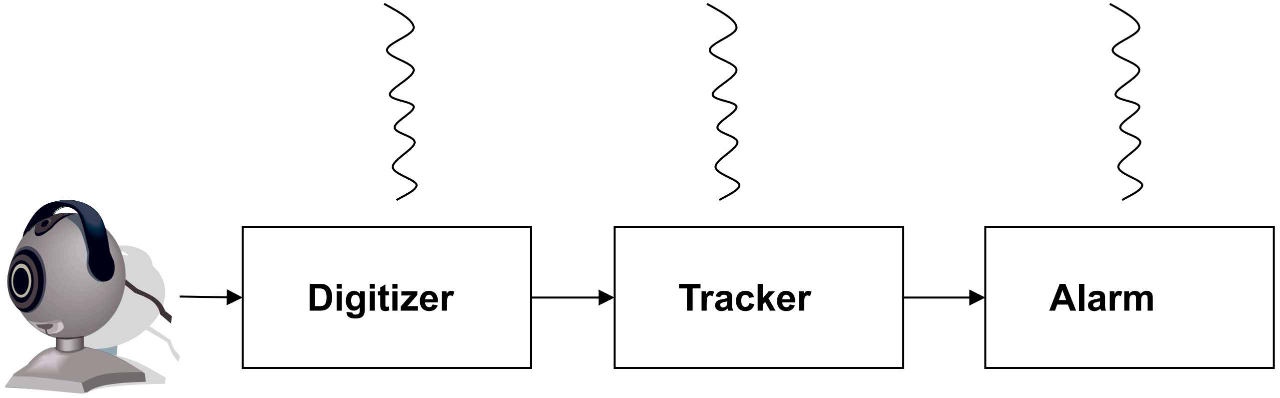
Example use of threads - 1



(a) Sequential process

(b) Multithreaded process

Example use of threads - 2



Digitizer

Tracker

Alarm

Digitize an image from the camera

Examine the image for motion

Send off a notification

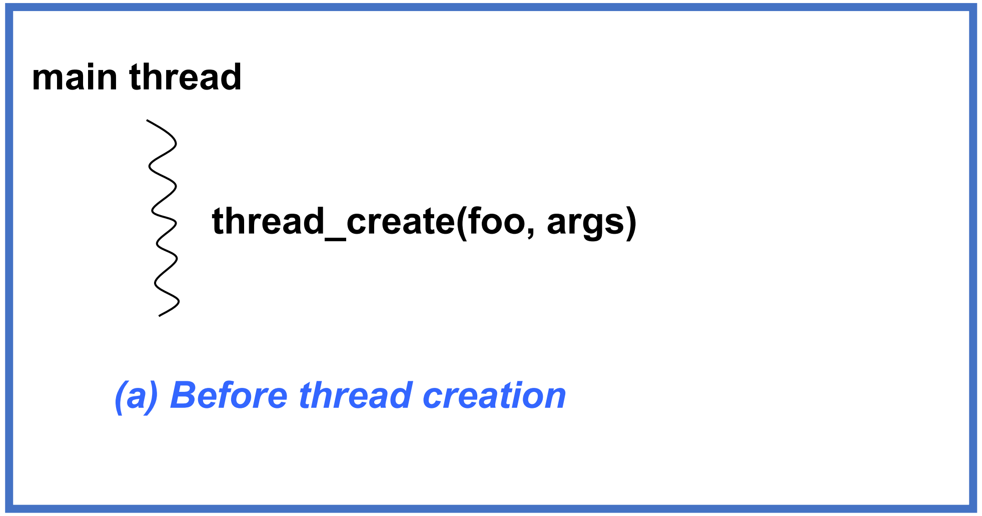
Where are we headed?

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

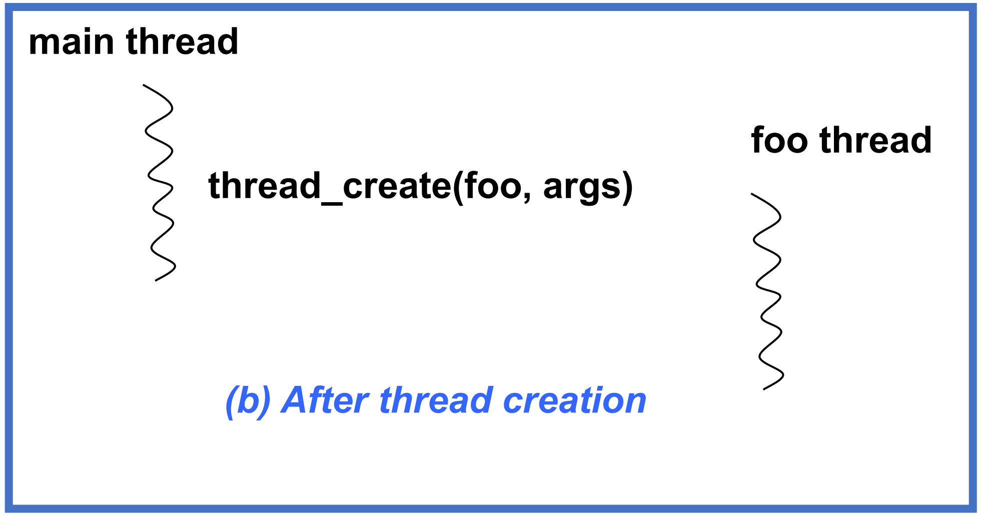
- Multithreaded program
  - Main and subroutines (procedures)
- Thread starts executing in some top-level procedure upon “thread create”
- Thread terminates
  - When main() terminates or
  - When the thread’s top-level procedure terminates
- We are going to need synchronization and communication among threads

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



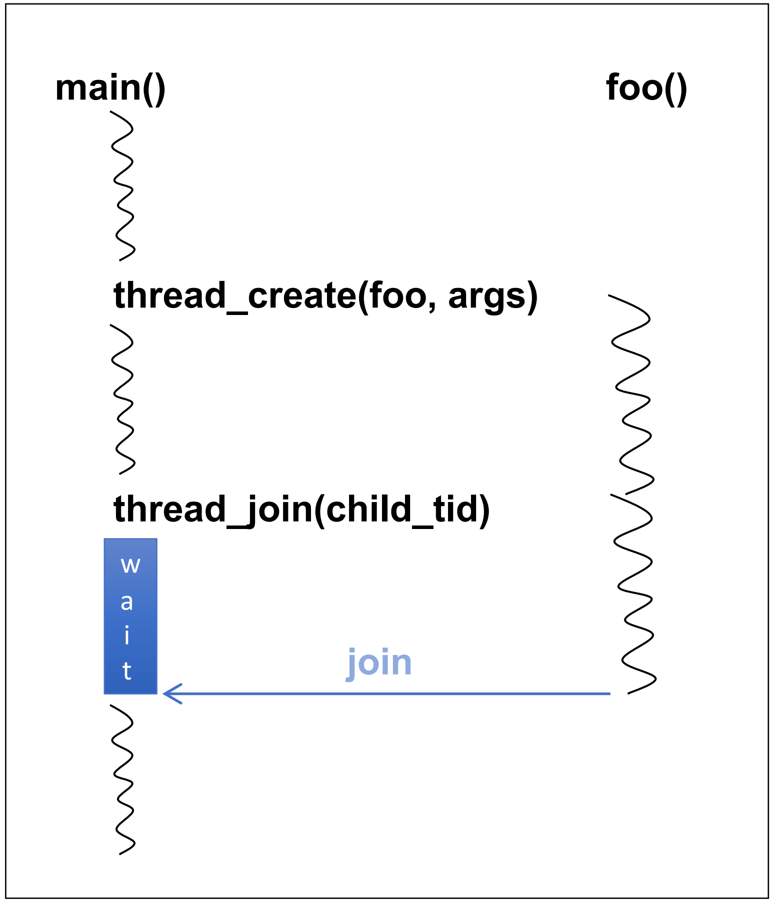
(a) Before thread creation



(b) After thread creation

Sample program – thread create/join

```
int foo(int n)
{
  .....
  return 0;
}

int main()
{
        int f;
        thread_type child_tid;

        .....

        child_tid = thread_create (foo, &f);

        .....

        thread_join(child_tid);
}
```



Threads within the same process

Session ID: dag2200

A thread…

7% A. I just want the participation credit

B. …is the same as a process

C. …is usually part of a process

D. …has nothing to do with a process

E. …usually refers to a set of processes

F. …is part of the memory hierarchy

G. …often involves a needle

Session ID: dag2200

A thread starts its execution …

67% A. I just want the participation credit

0% B. In main()

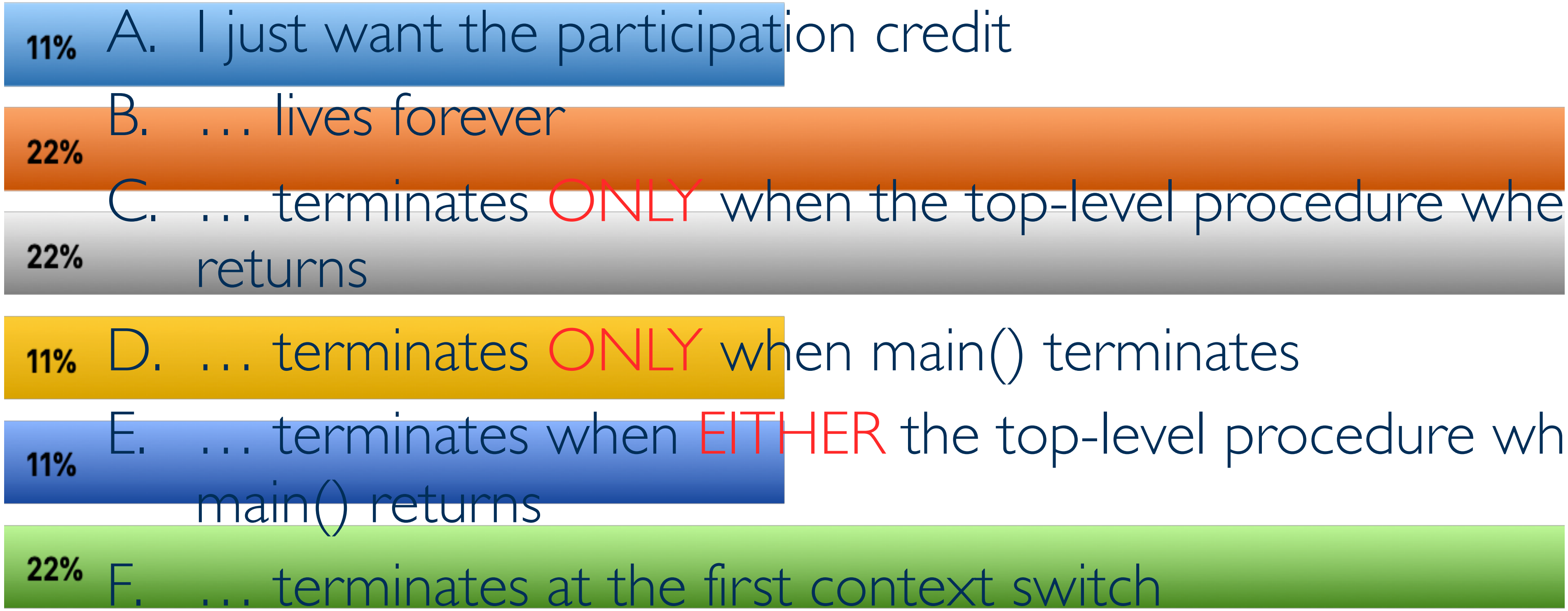
0% C. At some top-level procedure that is part of the same program

33% D. At some top-level procedure that is part of a different program

E. None of the above (B to D)

Session ID: dag2200

A thread



A. I just want the participation credit

B. … lives forever

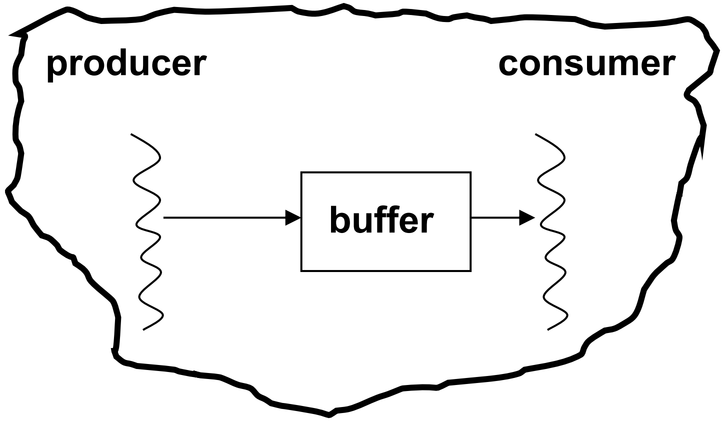
C. … terminates ONLY when the top-level procedure where it started returns

D. … terminates ONLY when main() terminates

E. … terminates when EITHER the top-level procedure where it started or main() returns

F. … terminates at the first context switch

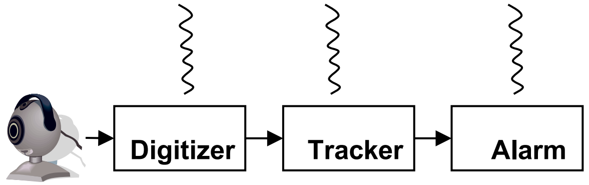
Programming with Threads



- synchronization
  - for coordination of the threads
- communication
  - for inter-thread sharing of data
  - threads can be executing on 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)

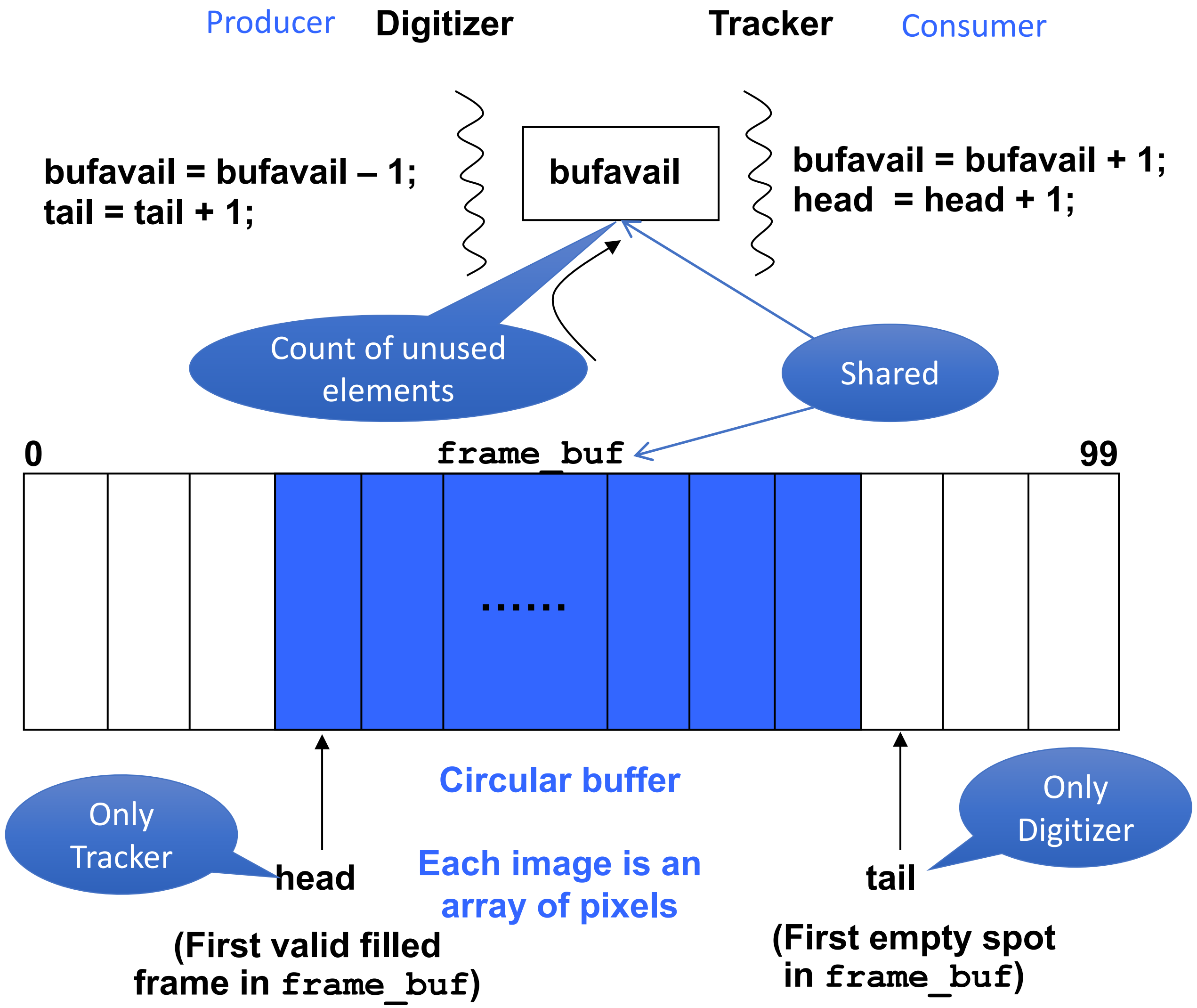
Hardware software partnership

Recall: our producer/consumer app



Global variables:

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



```
bufavail = bufavail – 1;
tail = tail + 1;
```

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

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);
    }
  }
}
```

Manipulating shared variables(!)

Problem?

What's the issue?

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

Tracker

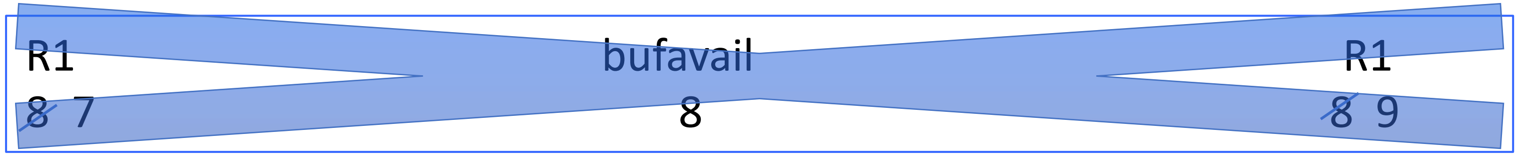
bufavail = bufavail – 1

```
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

is it 7 or 9?

R1 ~~8~~ 9

But it should be 8 ! – FAIL!

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);
    }
  }
}
```

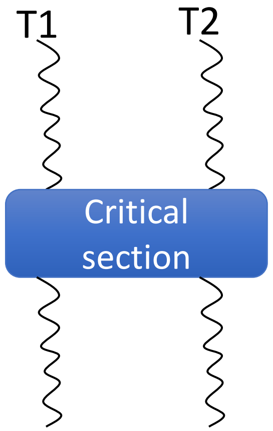
Problem?

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:  mutex lock; // data structure
        mutex_lock(lock); // acquire lock
        mutex_unlock(lock); //release lock
```



- OS has no idea what you do in the critical section
- OS guarantees only 1 thread in the critical section (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 #1 – with locks

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

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

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

Session ID: dag2200

A pthreads mutex lock

30% A. I just want the participation credit

B. Allows exactly one thread to acquire it at a time

C. Allows any number of threads to acquire it at a time

D. Allows a defined number of threads to acquire it at a time

E. None of the above (B to D)

Fix #1 – with locks

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

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

  loop {
    pthread_mutex_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_mutex_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 #1 – with locks

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

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

  loop {
   pthread_mutex_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_mutex_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 #2 – with locks

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

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

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

Problem?

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

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

Deadlock!

Fix #3

```
int bufavail = MAX;
image_type frame_buf[MAX];
mutex 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;
  thread_mutex_lock(buflock);
   bufavail = bufavail - 1;
  thread_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;
  thread_mutex_lock(buflock);
   bufavail = bufavail + 1;
  thread_mutex_unlock(buflock);
  analyze(track_image);
 }
}
```

Problem?

Busy waiting → Wastes CPU

Session ID: dag2200

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

30% A. I just want the participation credit

B. Thread B then waits on thread A

C. Thread B then waits on thread C which then waits on thread A

D. Thread B then tries to claim a mutex lock which is held by thread A

E. All of the above

What should really happen?

- 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

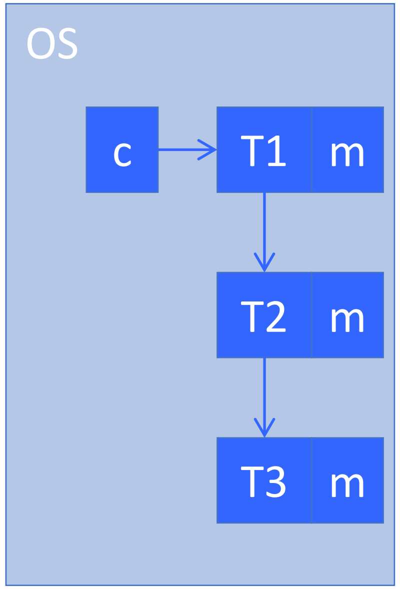
- That would stop the busy-waiting

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 (cond_var c, mutex m)
    - Atomically release mutex m
    - Put thread to sleep waiting on a signal to cond_var c
    - Atomically re-lock mutex m on awakening

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

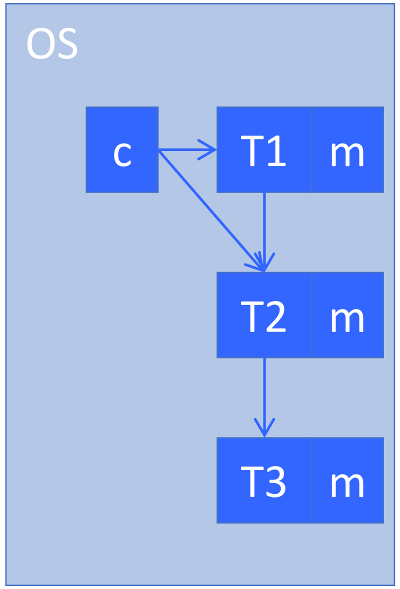
This is what the OS does:



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 (cond_var c)
    - Wake up one thread waiting on cond_var 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!
- We will call this condition an *invariant* for entering the critical section

Back to our surveillance app

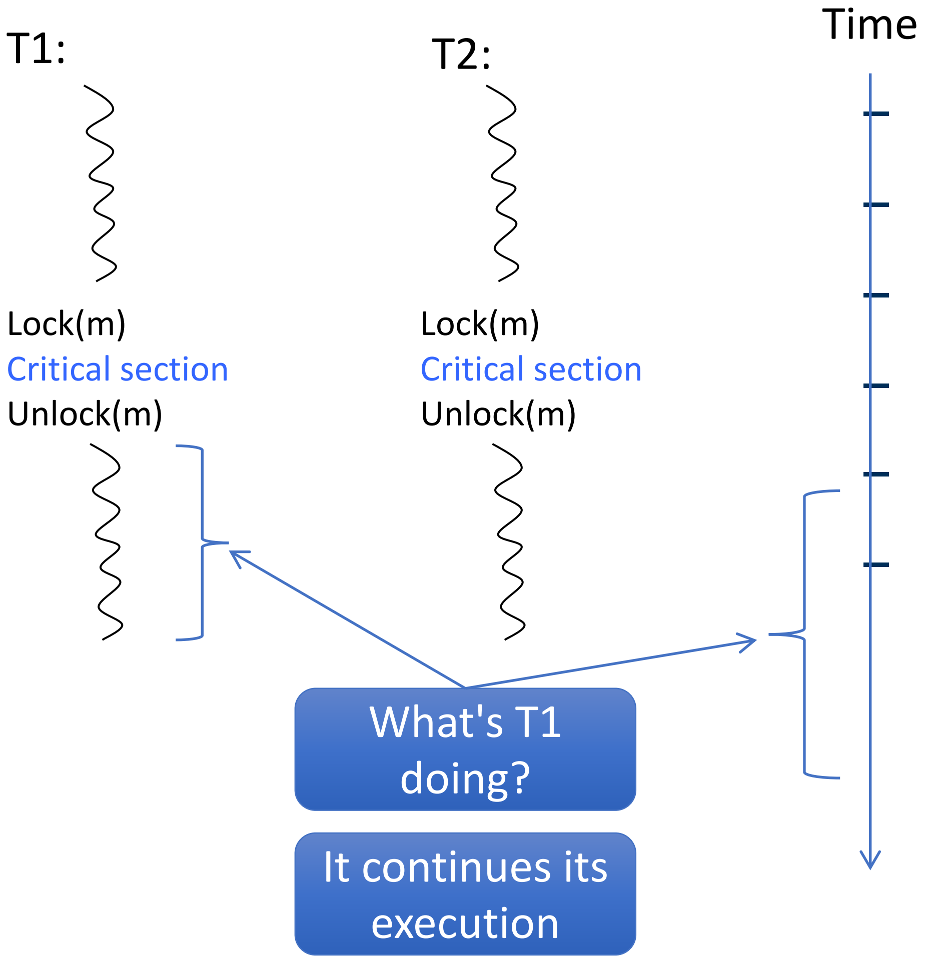
- What were we waiting for?
  - Digitizer waits for frame_buf to be not full
  - Tracker waits for frame_buf to be not empty

Fix #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);
  }
}
```

```
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);
  }
}
```

Recall: Mutex locks



T1:

Lock(m)
Critical section
Unlock(m)

T2:

Lock(m)
Critical section
Unlock(m)

What's T1 doing?

It continues its execution

Time

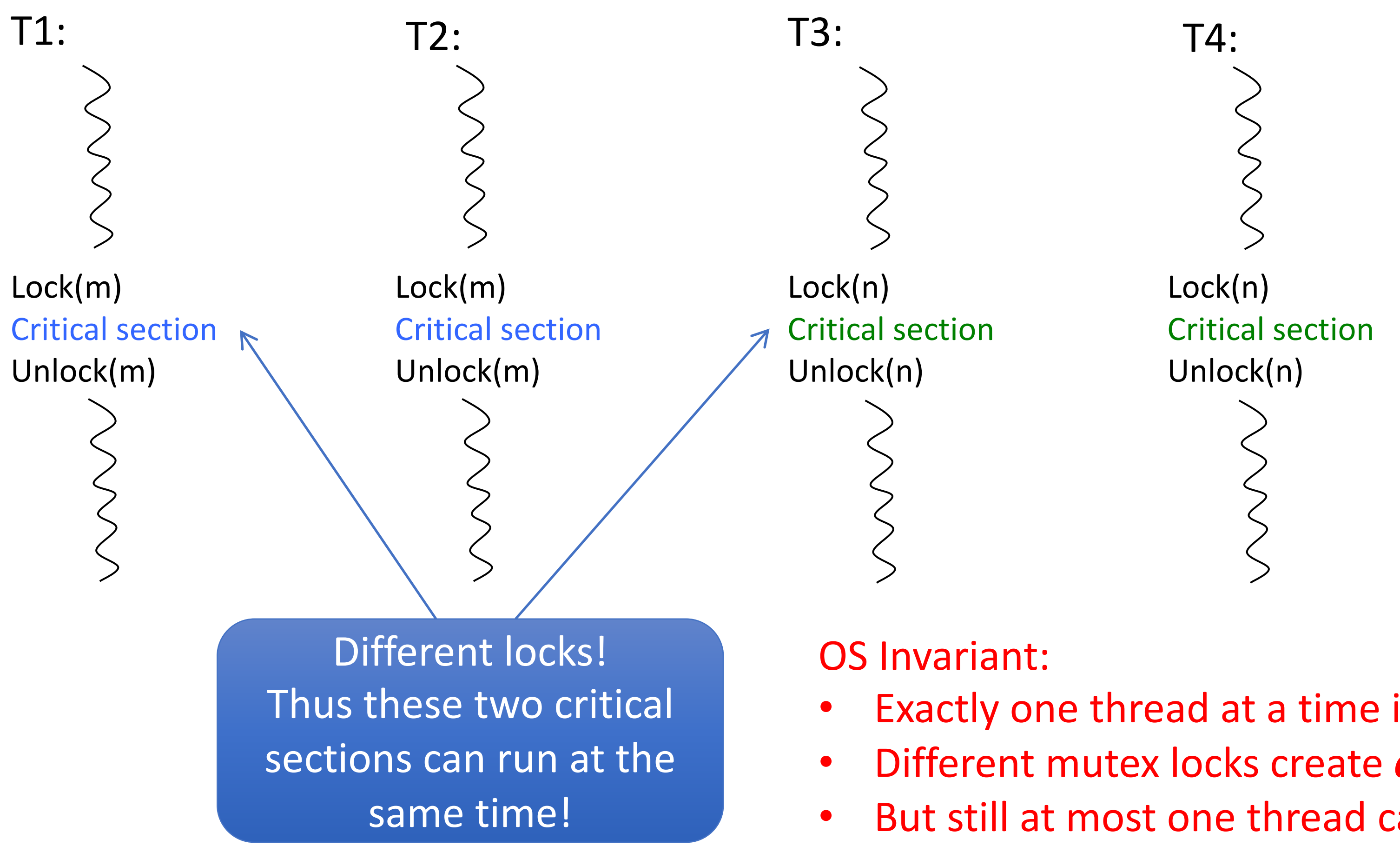
- Both T1 and T2 execute
- T1 gets lock, T2 is stuck
- CS (T1)
- T1 unlocks, T2 gets lock
- CS (T2)
- T2 unlocks

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:

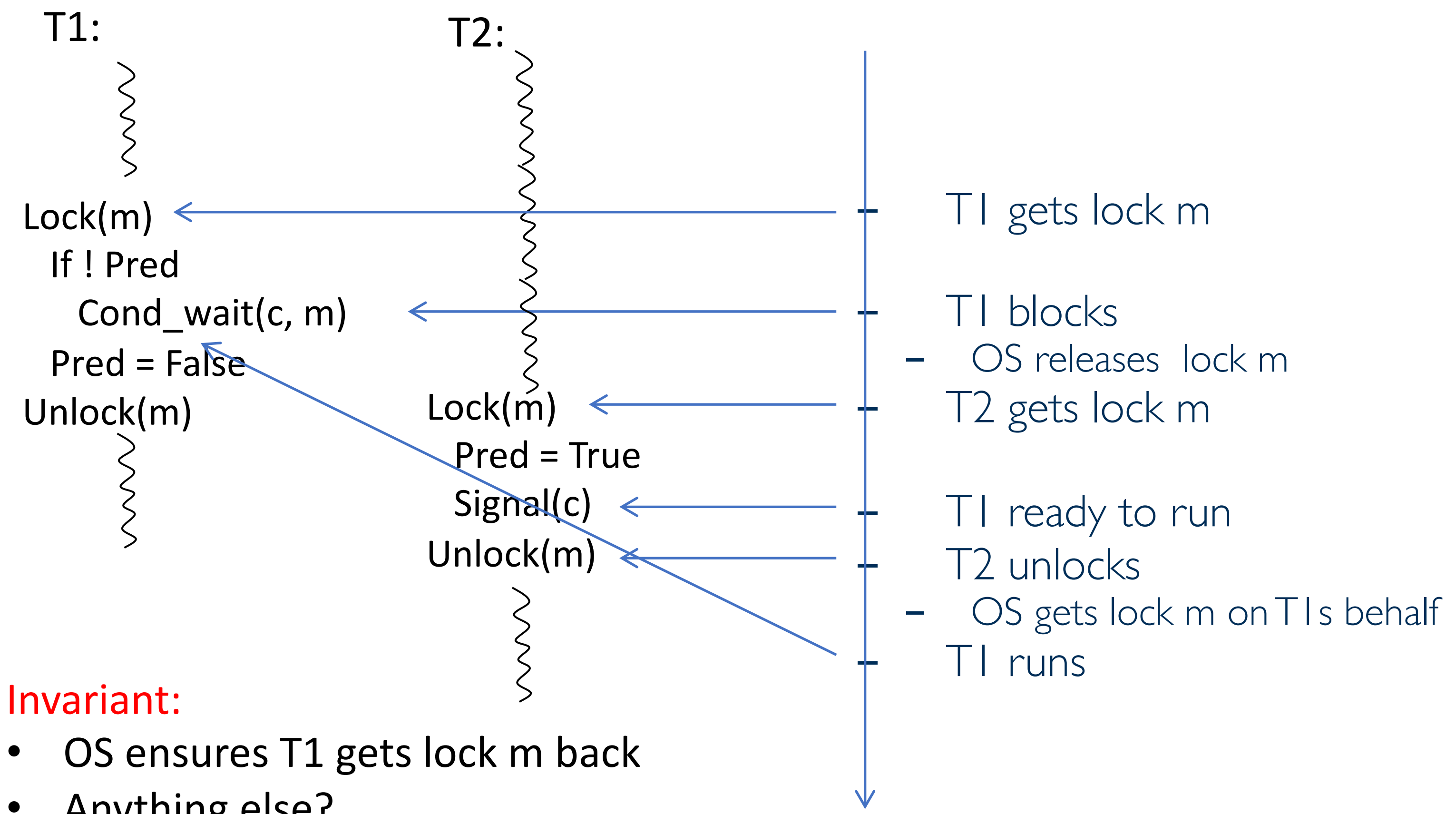
Lock(n)
Critical section
Unlock(n)

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

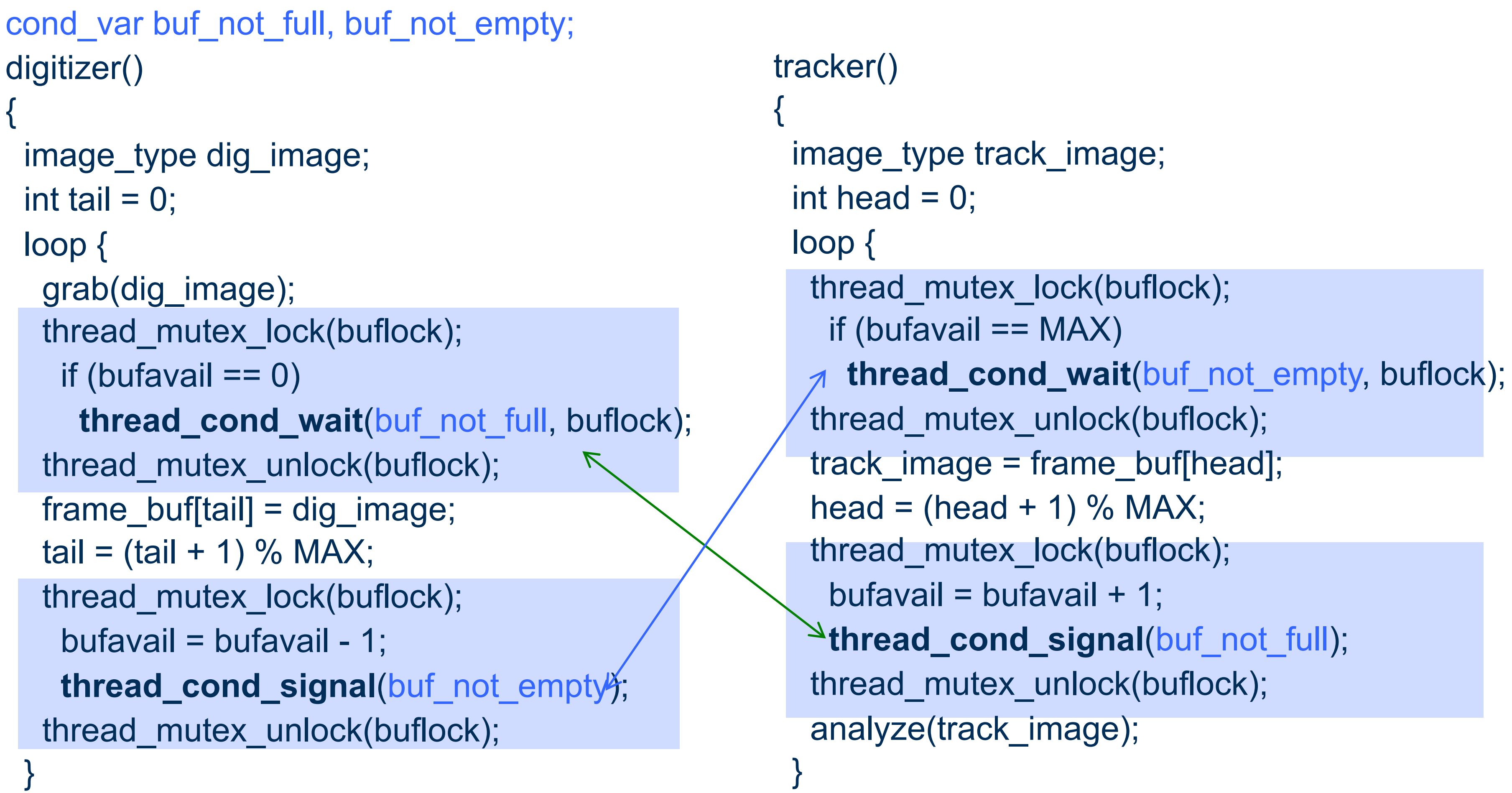
Who must ensure this?

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 #4 – cond_var



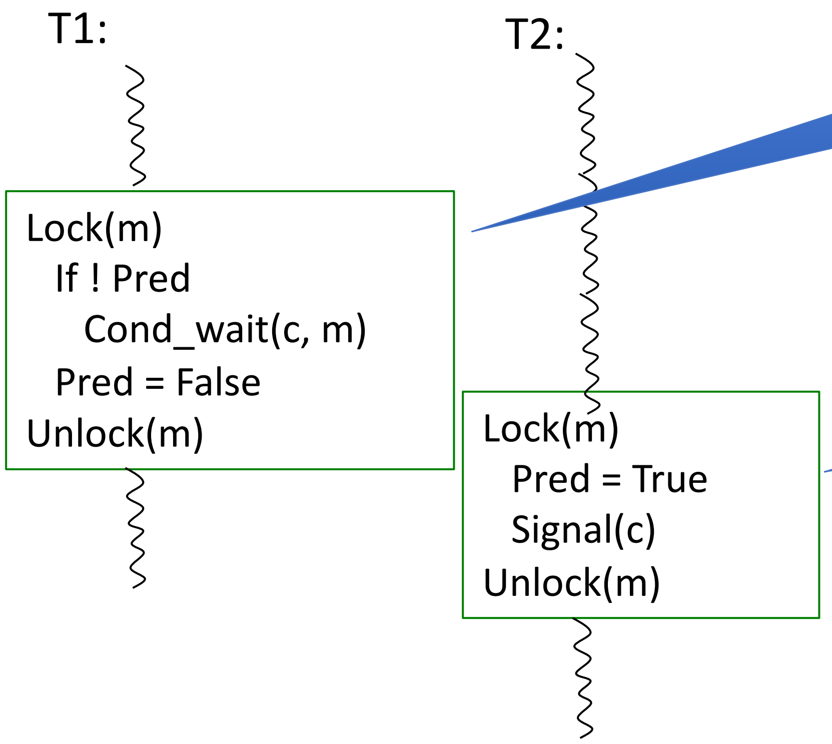
```
cond_var buf_not_full, buf_not_empty;
digitizer()
{
  image_type dig_image;
  int tail = 0;
  loop {
    grab(dig_image);
    thread_mutex_lock(buflock);
      if (bufavail == 0)
        thread_cond_wait(buf_not_full, buflock);
    thread_mutex_unlock(buflock);
    frame_buf[tail] = dig_image;
    tail = (tail + 1) % MAX;
    thread_mutex_lock(buflock);
      bufavail = bufavail - 1;
      thread_cond_signal(buf_not_empty);
    thread_mutex_unlock(buflock);
  }
}
```

```
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);
  }
}
```

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)
```

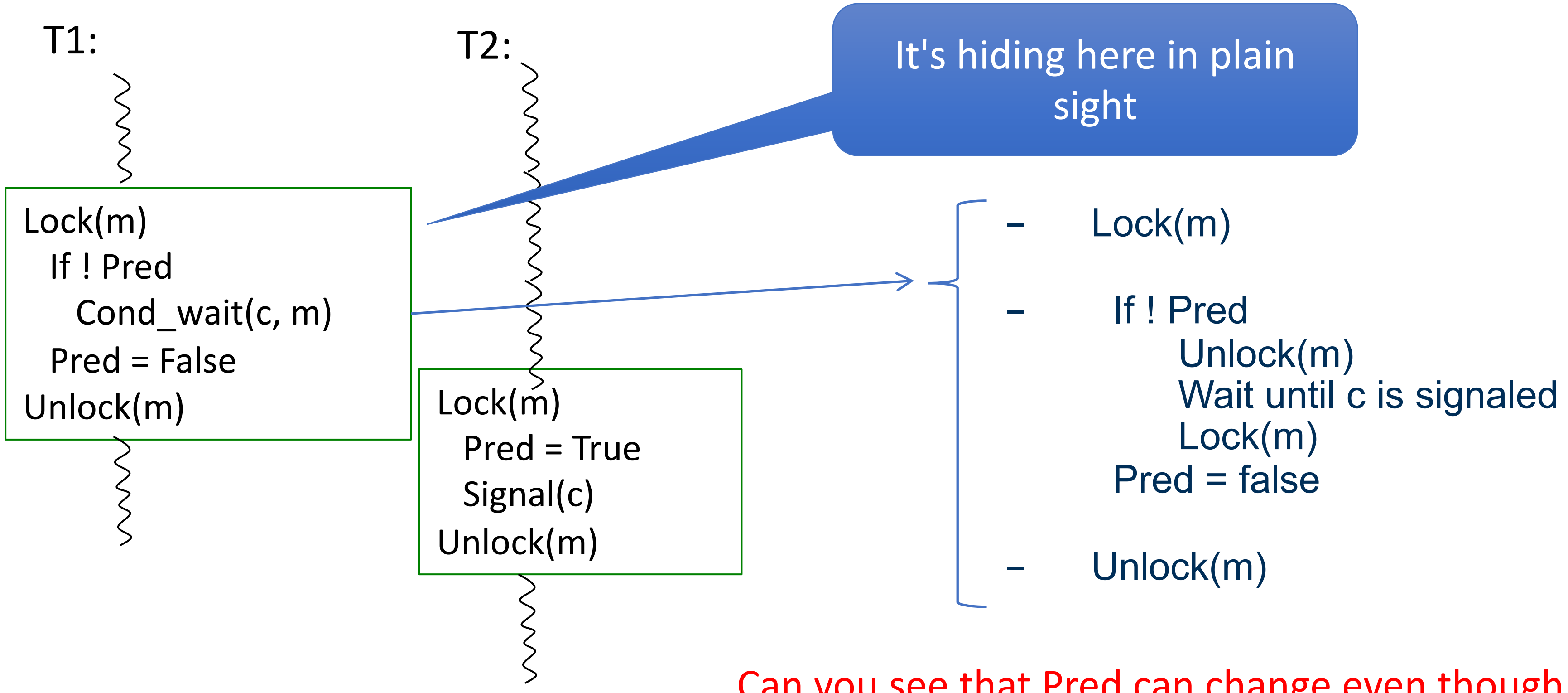
The first is here

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 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

```
digitizer()
{
  image_type dig_image;
  int tail = 0;
  loop {
    grab(dig_image);
    thread_mutex_lock(buflock);
    while (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);
  }
}
```

Defense: Re-check invariants

```
tracker()
{
  image_type track_image;
  int head = 0;
  loop {
    thread_mutex_lock(buflock);
    while (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);
  }
}
```

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

Session ID: dag2200

A condition variable…

13% A. I just want the participation credit

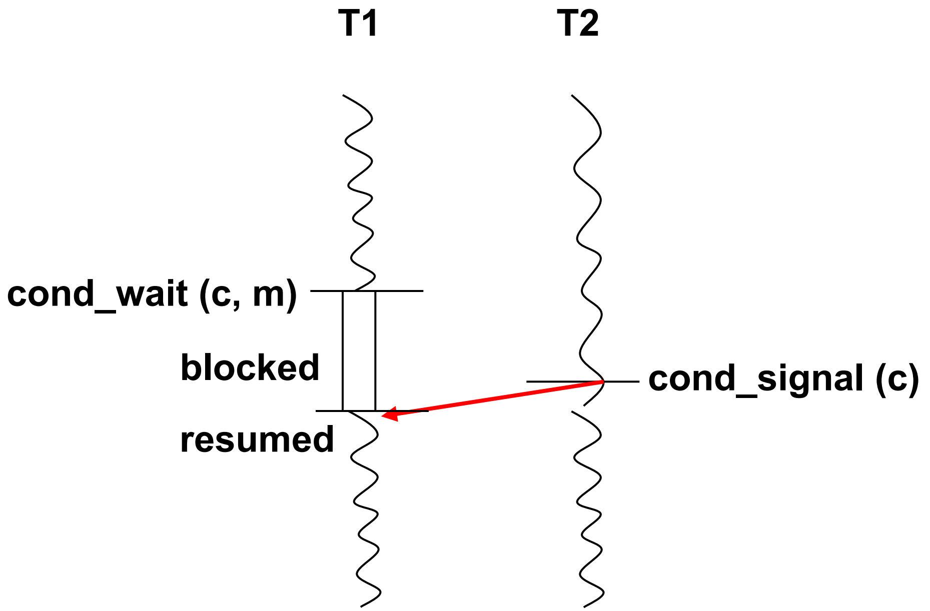
B. … is just another name for a mutex lock

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

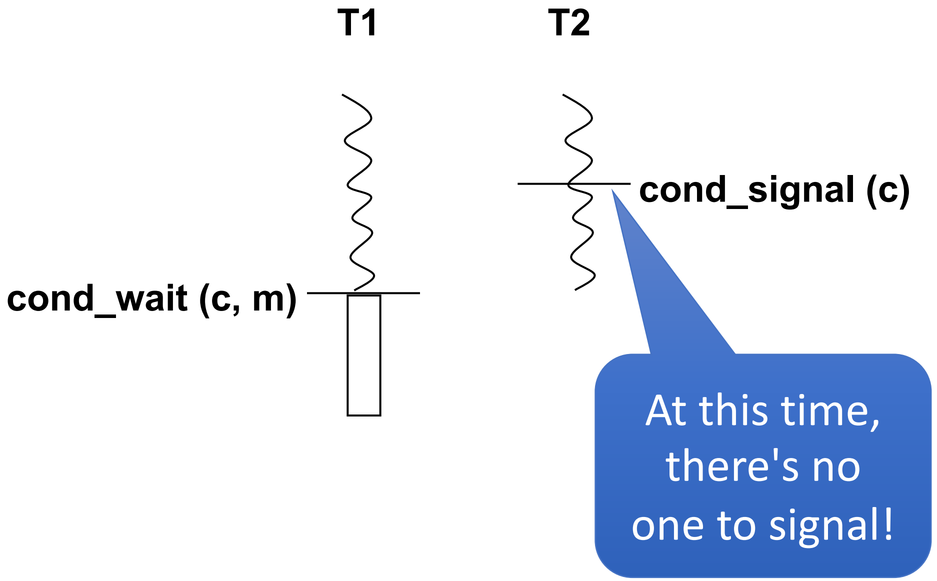
D. …enables a thread to enter a critical section

E. …none of the above (B to D)

Gotchas in programming with cond vars



(a) Wait before signal



At this time, there's no one to signal!

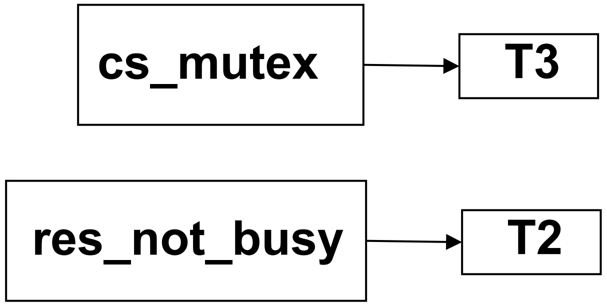
(b) Wait after signal (T1 blocked forever)

Gotchas in programming with cond vars

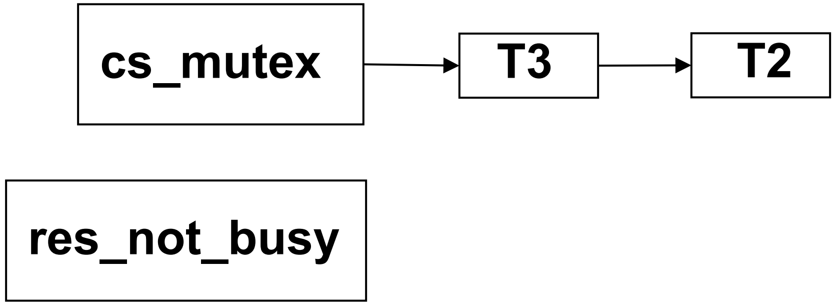
Say we have three threads that want to share a resource, perhaps a printer…

```
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?

```
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);
}
```

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(!)

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

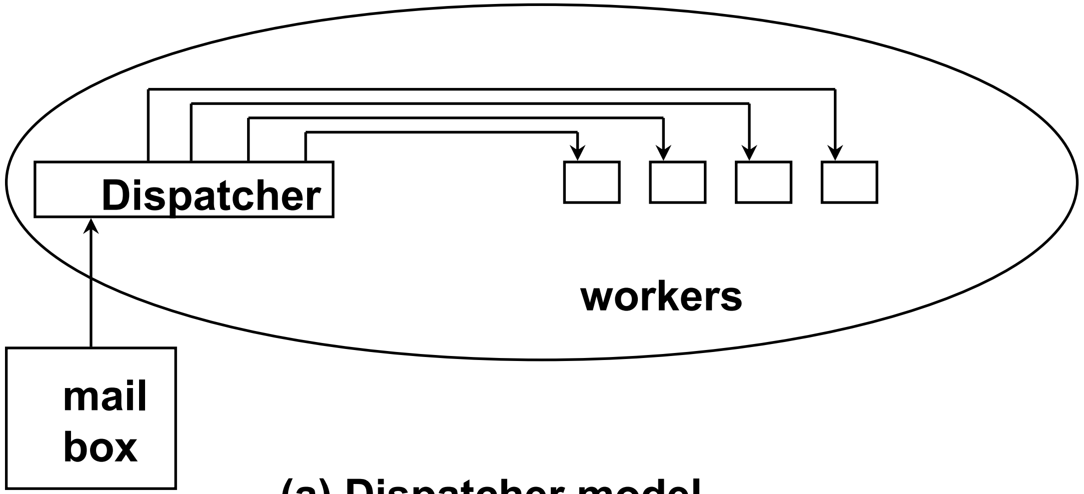
Avoids the "race condition"

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

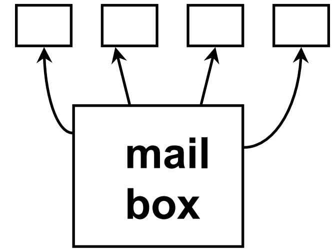
Checkpoint

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

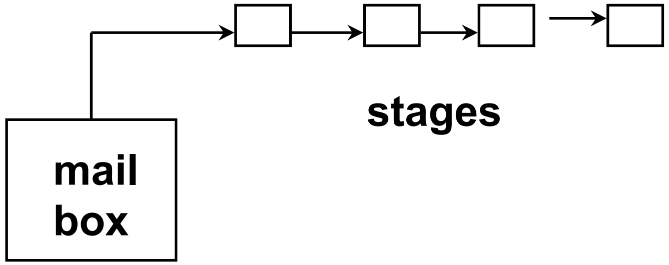
Threads as software structuring abstraction



(a) Dispatcher model



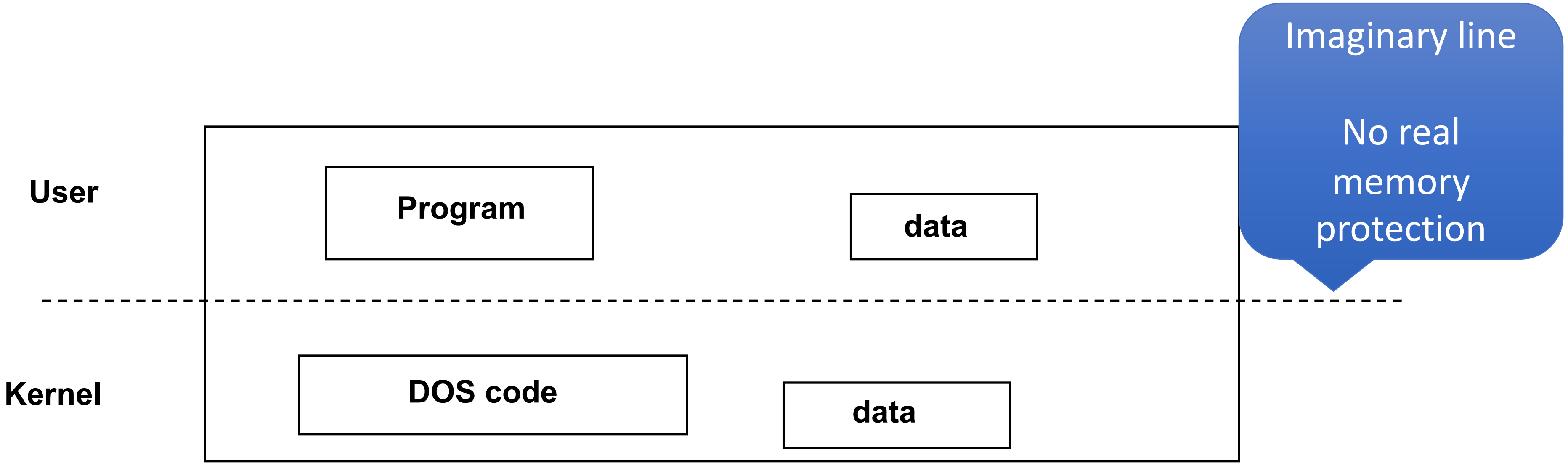
(b) Team model



(c) Pipelined model

Traditional OS: DOS

Memory Layout:



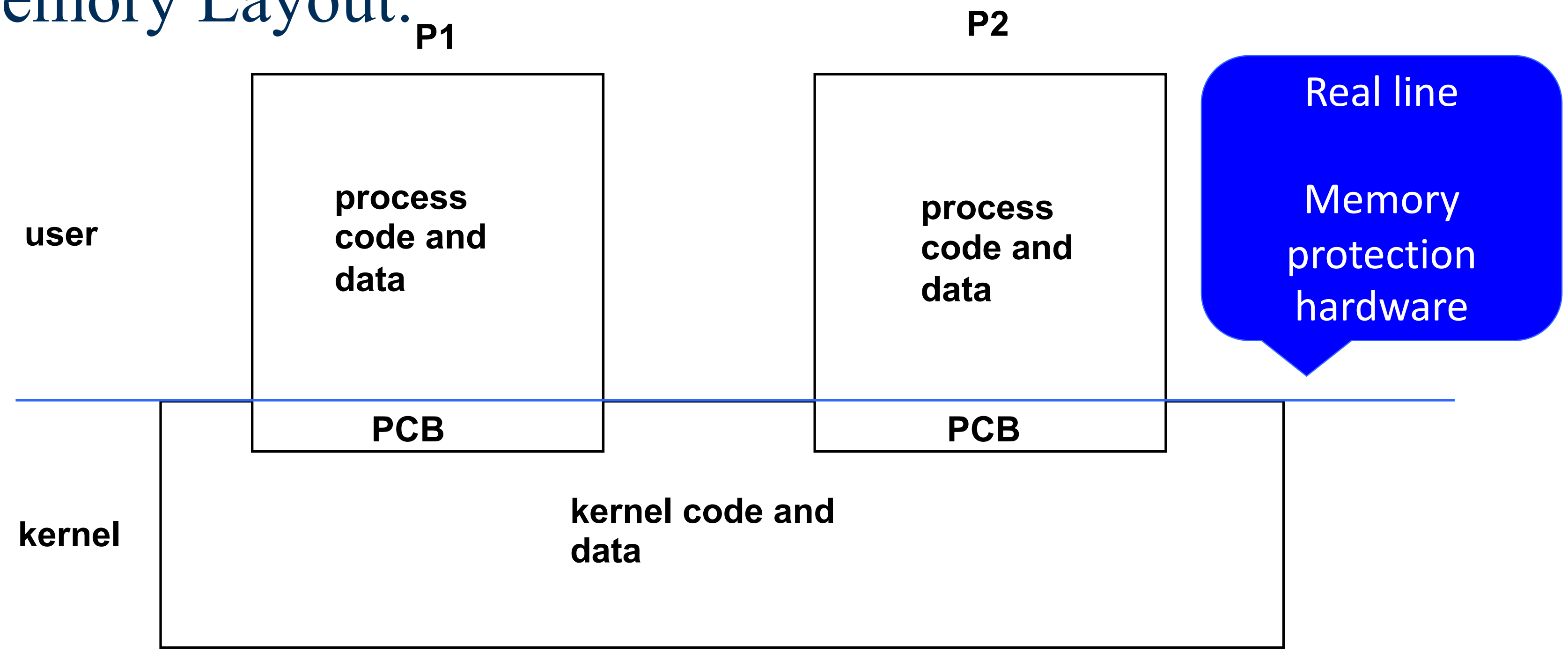
Imaginary line

No real memory protection

- 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

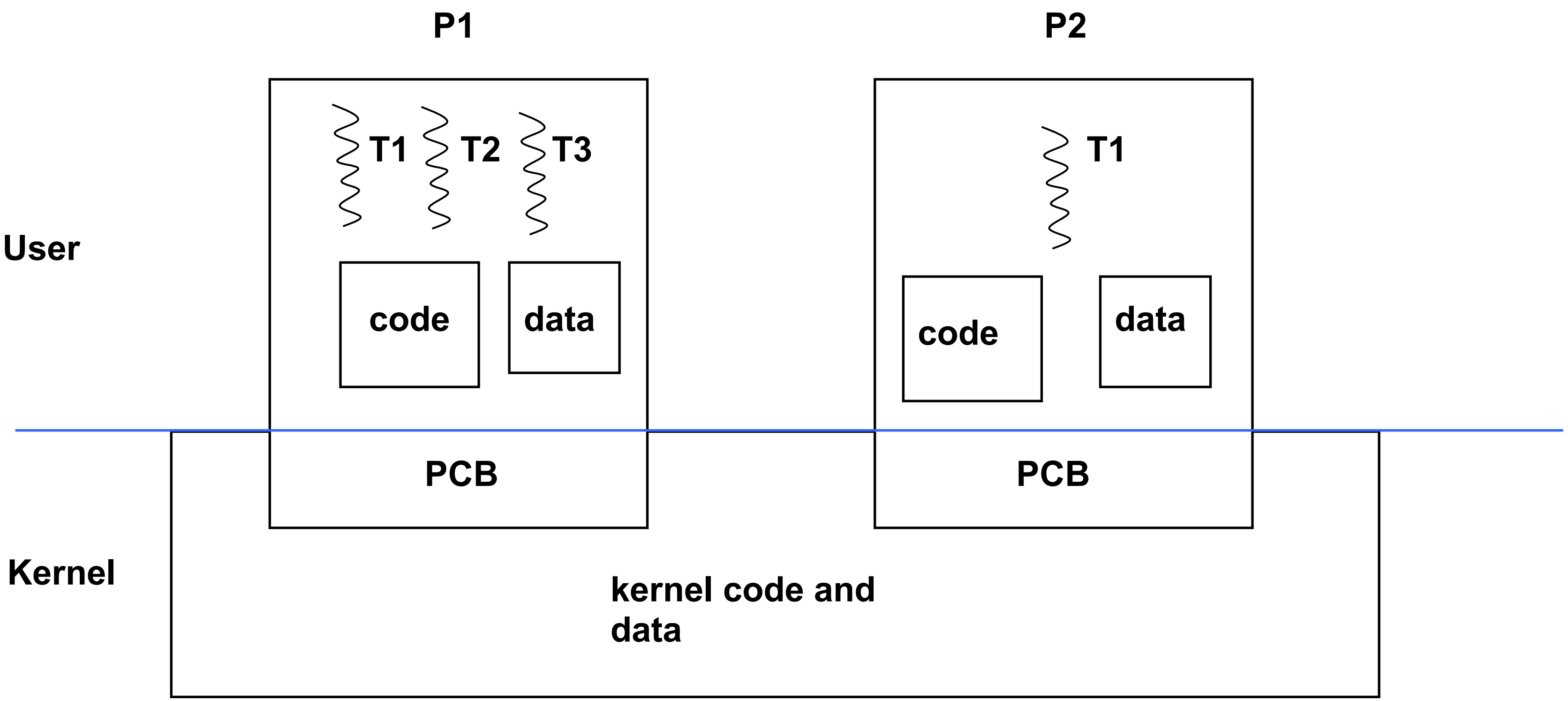
How widespread is support for threads in OS?

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

Process Vs. Thread?

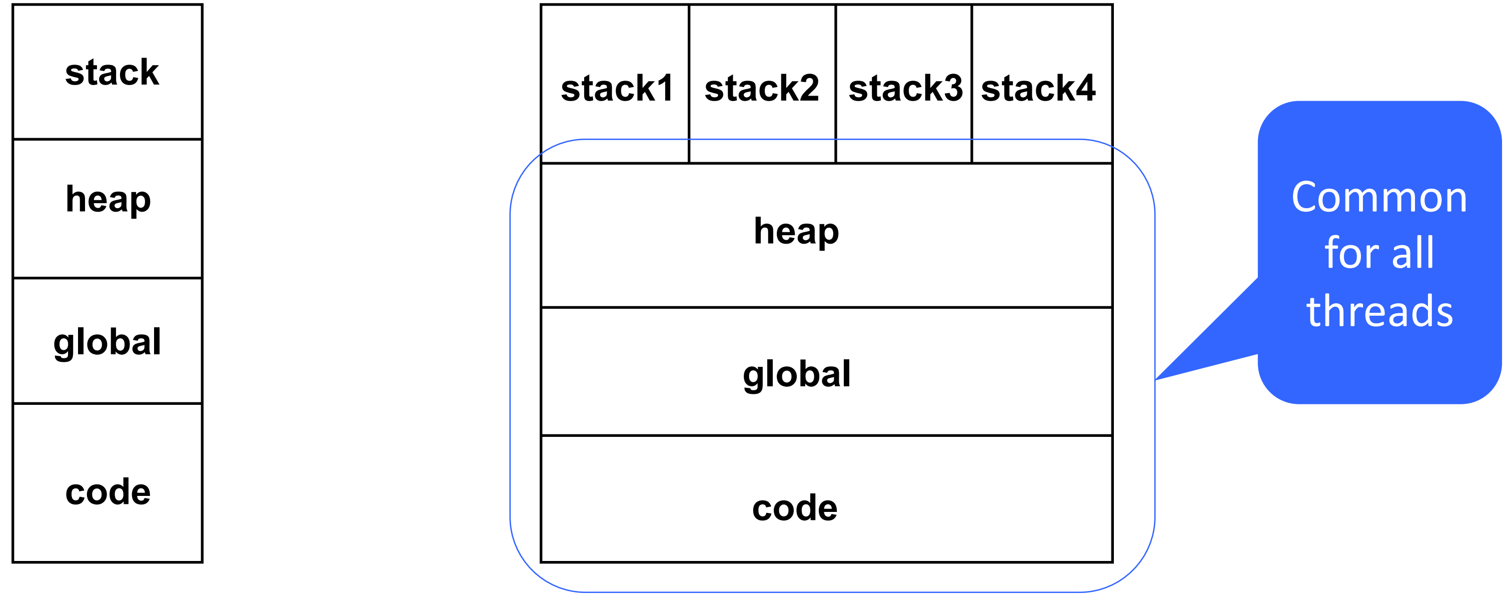
- In a single-threaded program, the state of the executing program is contained in a process
- In a MT program, the state of the executing program is contained in several 'concurrent' threads

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

- 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

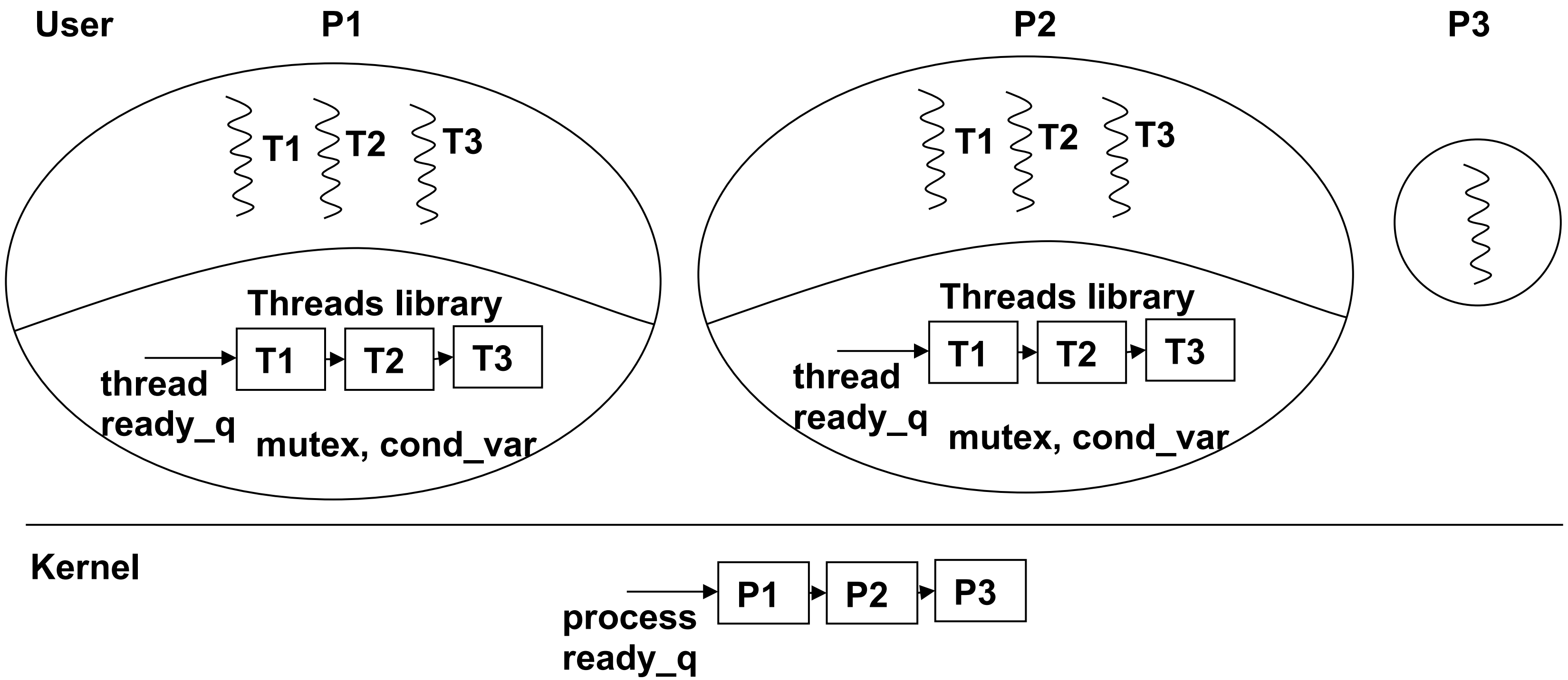
- Threads concurrent?
  - Truly parallel if the box is a true multiprocessor
  - Share (time-multiplex) the same CPU on a uniprocessor

- Threaded code different from non-threaded?
  - Protection for data shared among threads
  - Synchronization among threads

User-Level Threads

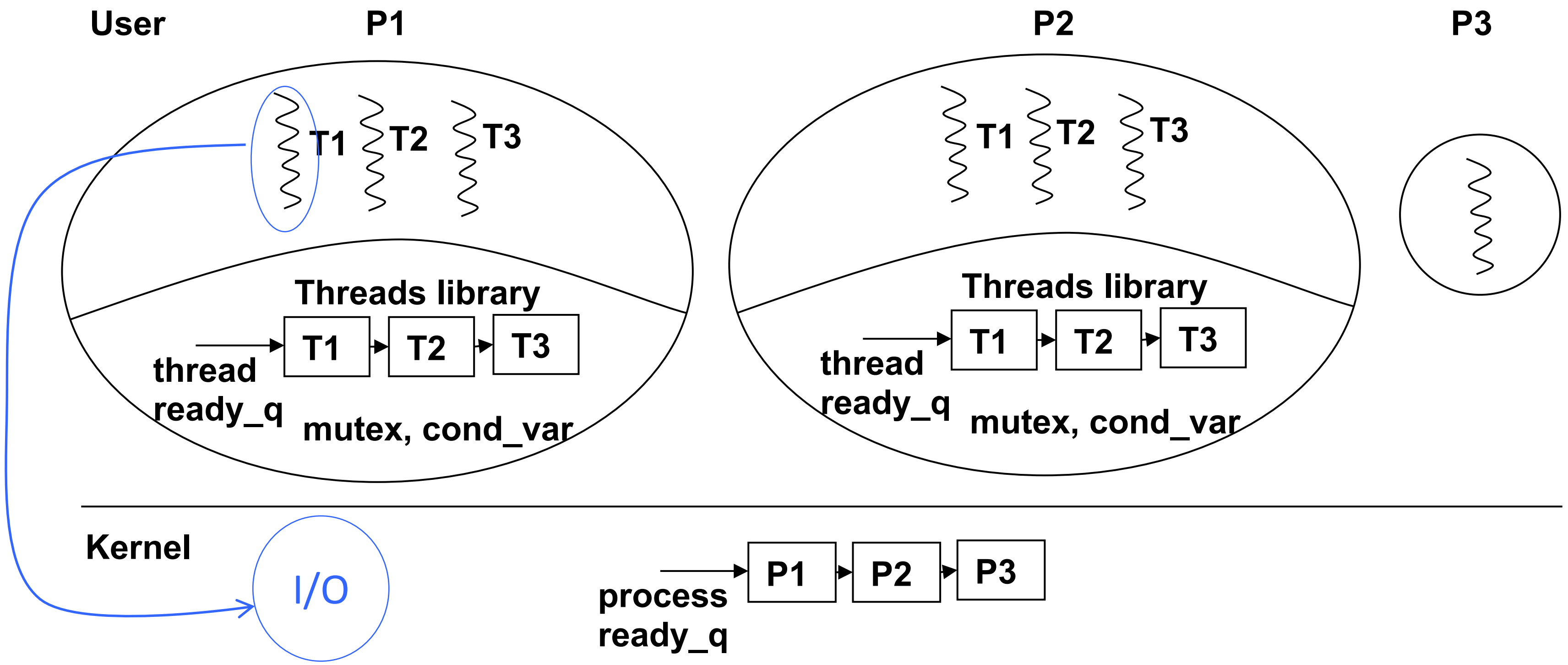
- OS independent
- Scheduler is part of the user space runtime system
- Thread switching is cheap (just save PC, SP, regs)
- Scheduling is customizable, i.e., more app control
- Blocking call by a thread blocks a process

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

Session ID: dag2200

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

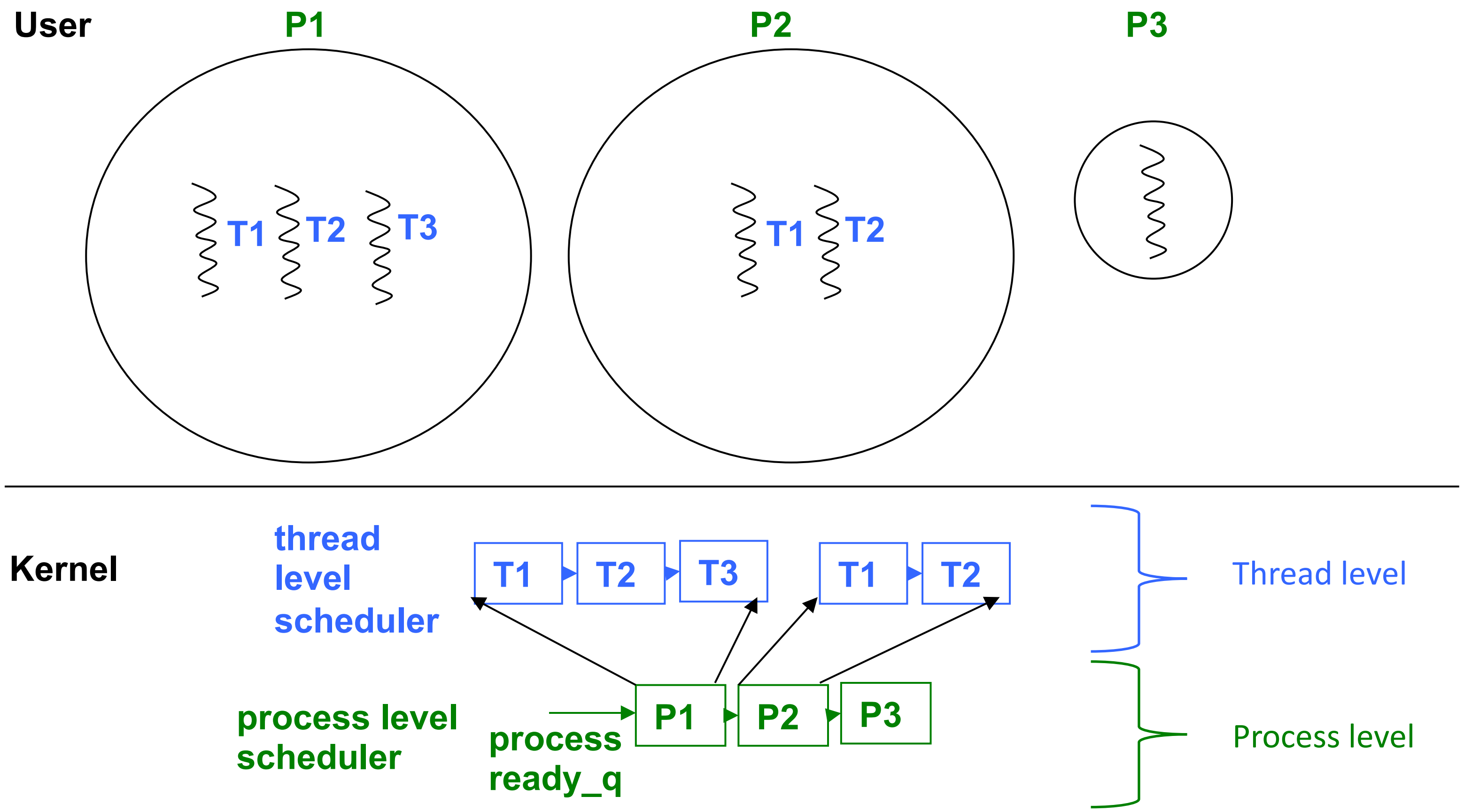
Kernel-level threads

- 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

```
/* original version */   |  /* thread safe version */
                         |
                         |  mutex_lock_type cs_mutex;
void *malloc(size_t size)|  void *malloc(size_t size)
{                        |  {
                         |   thread_mutex_lock(cs_mutex);
                         |
  ......                 |   memory_pointer = malloc(size);
  ......                 |   ......
                         |
                         |   thread_mutex_unlock(cs_mutex);
                         |
  return(memory_pointer);|   return (memory_pointer);
}                        |  }
```