Lecture 15: Mutual Exclusion

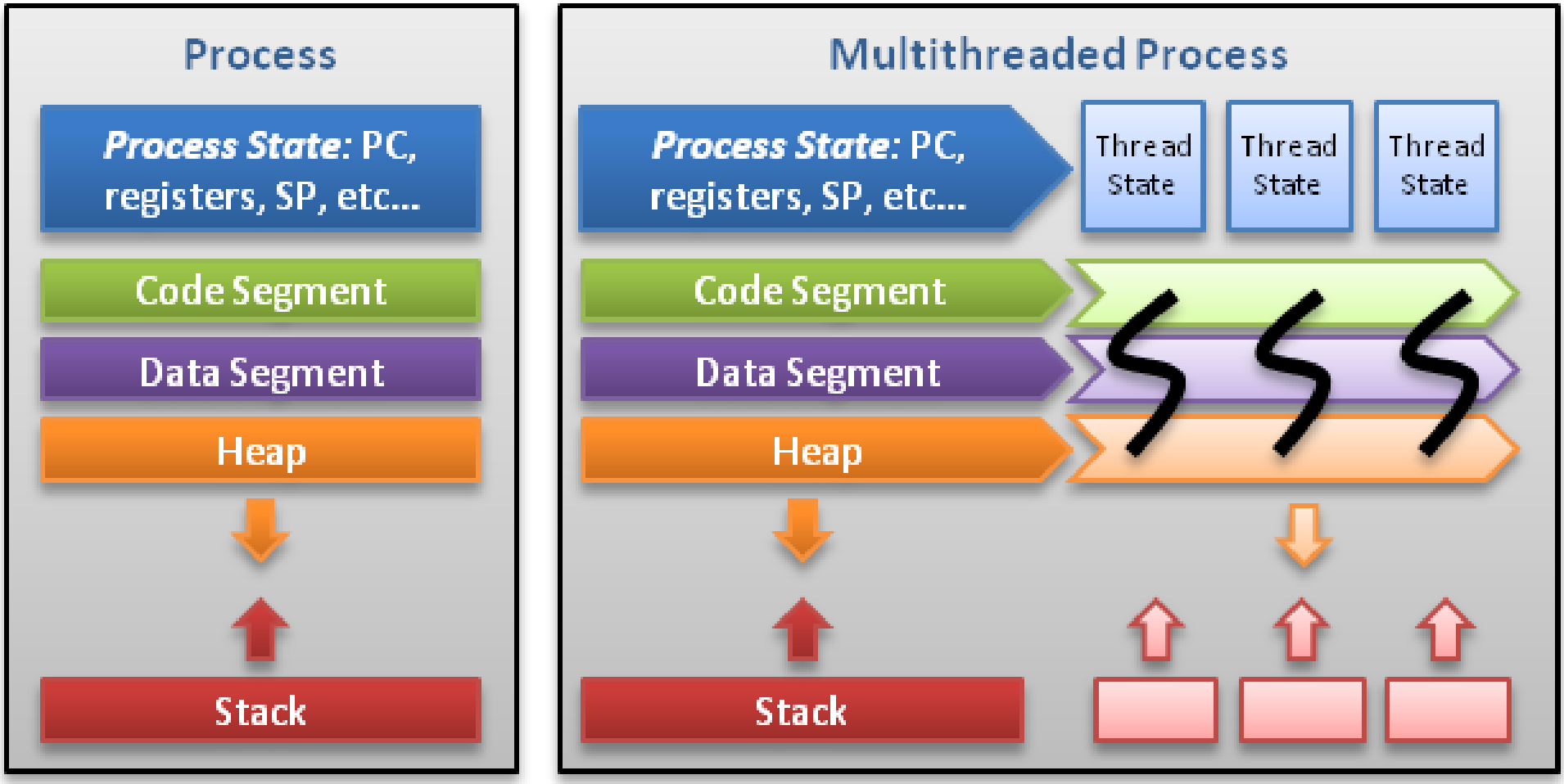
Peterson's Algorithm, Spinlocks, Mutexes, and Futexes

Xin Liu

Florida State University
xliu15@fsu.edu

COP 4610 Operating Systems
https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

Recap: Multithreading Model



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

Recap: How Many Threads Run at a Time?

- Threads in one process share the **same address space** (code, data, heap).
- The address space contains **multiple stacks**, one per thread.

However, a single CPU core executes **only one instruction stream (thread)** at any instant.

- **Because**, they also all share one set of **core hardware resources**: program counter (PC), registers, pipeline, and execution units.
- These resources can only hold the state of *one* execution stream at a time, only one thread can physically run at that instant.

Recap: Using the Pthreads Library

To create a thread in C, we use the POSIX (Pthreads) library.

```
#include <stdio.h>
#include <pthread.h>

// 1. The required function signature
void *hello() {
    printf("Hello\n");
    return NULL;
}

int main() {
    pthread_t t1;                          // 2. A variable to hold the thread

    pthread_create(&t1, NULL, hello, (void *)1L); // 3. Create the thread

    pthread_join(t1, NULL);                // 4. Wait for the thread to finish
    return 0;
}
```

Recap: From “Hello, Thread” to “I’m out.”

- Loss of Determinism in State Transitions
  - `load` → `add` → `store`
  - Any `load` may read a value written by another thread.
  - Even “1 + 1” does not equal 2.
- Loss of Sequential Consistency
  - The compiler and CPU can reorder, merge, or remove loads and stores.
  - We think `while(!flag)` rereads flag each time, but the compiler hoists one load so the loop never sees the update.
- Loss of Global Program Order
  - Hardware reality: Shared memory on multicore uses relaxed memory.
  - A store first updates a local cache, then propagates. Cores may read stale values due to propagation latency, so visibility times differ.

Do Not Give Up!

Humans are born to solve concurrency.

Shared Memory $\leftrightarrow$ The Physical World

- The world is inherently parallel.
- Many agents act at the same time.

Threads $\leftrightarrow$ People

- Each brain has local storage and computation.
- We cache facts, plan, and update our state.

Where problems appear

- Agents touch the same resource: a door, a desk, a fridge.
- Two people write the same note at once $\Rightarrow$ a race.
- Reads can be stale. Writes can arrive late.

This motivates Exclusion

- One-at-a-time access to a critical section.
- Real-life tools: a key, a line, a “busy” sign.

Exclusion

Start from a simple problem: 1 + 1

```
long sum = 0;

void T_sum() {
    sum++;
}
```

We want an API

- The result of sum is correct regardless of execution order.
- Mutual exclusion: prevent concurrent `sum++`.

Stop the World

Figure: Dio's stand, The World, can stop time, allowing only Dio and his stand to move during the frozen period.

Can OS give us a "Stop the World" instruction?

Stop the World (cont'd)

```
long sum = 0;

void T_sum() {
    stop_the_world();
    sum++;
    resume_the_world();
}
```

This may be overkill

- We only need to mark the critical section that must not run concurrently.
  - A critical section is the smallest code block that reads or writes shared data and therefore requires mutual exclusion.
- Other code unrelated to `sum` can still run at the same time.

Mutual Exclusion: prevent concurrent execution

```
// critical section
lock();
sum++;
// or any code
unlock();
```

Human view

- Use `lock`/`unlock` to mark onecritical section.
- Marked critical sections are mutually exclusive: **once I enter a marked critical section, others cannot enter.**

State-machine view

- Executing a marked block is a single state transition.

If we stop the world, do we still need threads?

Pessimistic view: Amdahl's Law

- If a fraction $1/k$ of the code is serial, then

$$T_\infty > \frac{T_1}{k}.$$

Optimistic view: Gustafson's Law (more precise form)

- Parallel computing is still achievable.

$$T_p < T_\infty + \frac{T_1}{p}.$$

(T_n denotes the running time on n processors.)

- Protect only the true critical section, keep it small.
- Parallelize the rest of the work outside the lock.
- Use fine-grained locks or lock-free where possible.
- Threads still help overlap I/O and use multiple cores.

Peterson's Algorithm: Incorrect Attempts of Mutual Exclusion

Dekker's Algorithm (1965)

Rule: A process *P* can enter the critical section if the other does not want to enter, otherwise it may enter only if it is its turn.

The original solution due to Dekker is discussed at length by Dijkstra in [1]. Of the many reformulations given since, perhaps the best appears in [3]. (Unfortunately the authors believe their correct solution is incorrect.) The solutions of Doran and Thomas are slight improvements which eliminate the 'loop inside a loop' structure of the previously published solutions. The solution presented here has an extremely simple structure and, as shown later, is easy to prove correct.

- Myths about the mutual exclusion problem (IPL, 1981)

Peterson's Algorithm (1981)

Rule: A process P can enter the critical section if the other does not want to enter, or it has indicated its desire to enter and has given the other process the turn.

Peterson's Protocol: A Better Analogy

Single Occupance Restroom

- Anyone can use the restroom, but only one car at a time.
- If two persons want to enter, a turn label decides whose **turn** it is.
- Using the restroom does not consume anything; others can use it later.

Peterson's Protocol

Model

- Three variables: *my flag*, *other's flag*, and *turn*.

If I want to enter, do these in order

1. Raise my flag (store *my flag* = up).
2. Set *turn* to the other side (store *turn* = other).

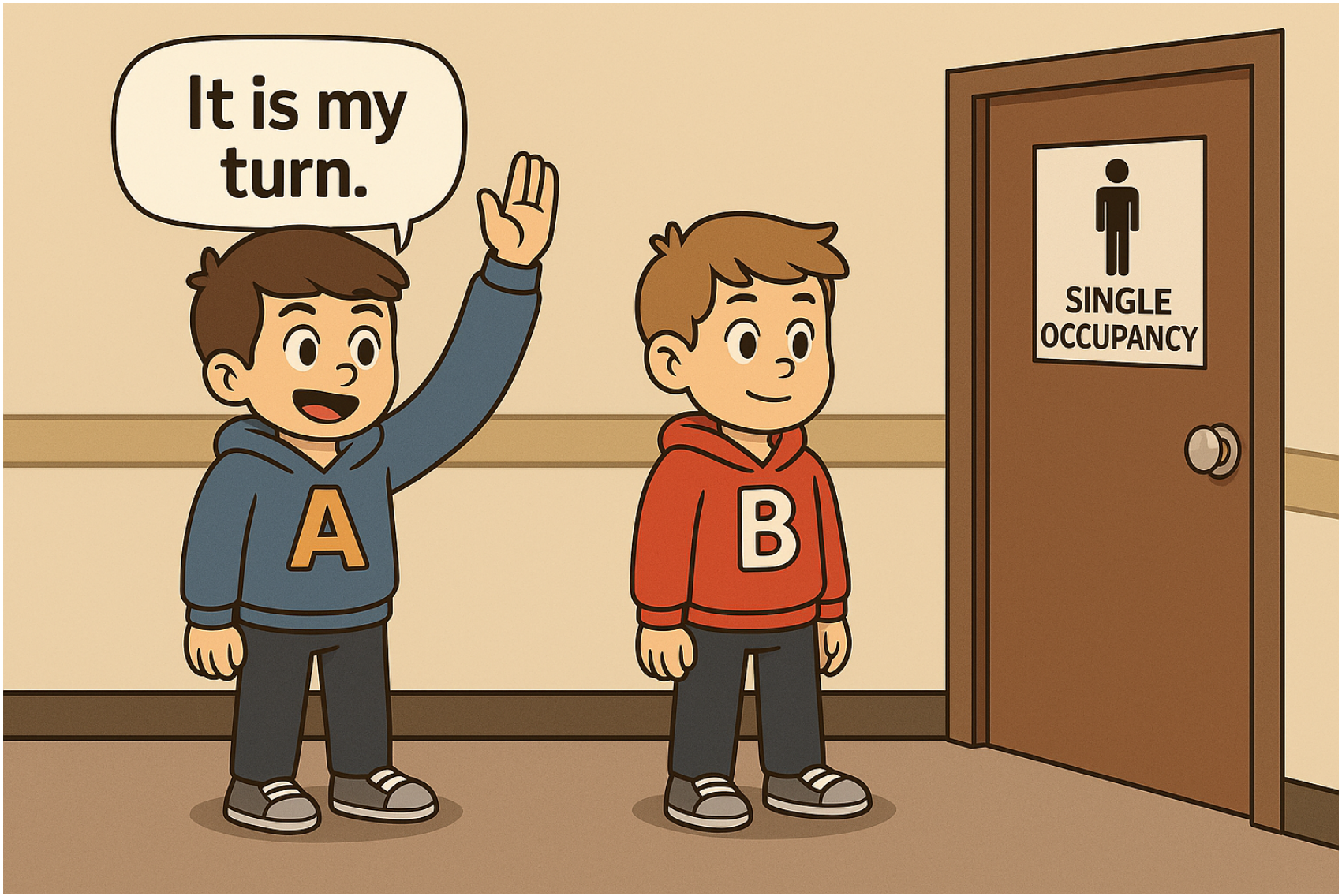
Then spin and observe

1. Read the other's flag (load *other's flag*).
2. Read the turn label (load *turn*).
3. If the other is **not** raising a flag **or** *turn* is **me**, enter; otherwise keep waiting.

On exit

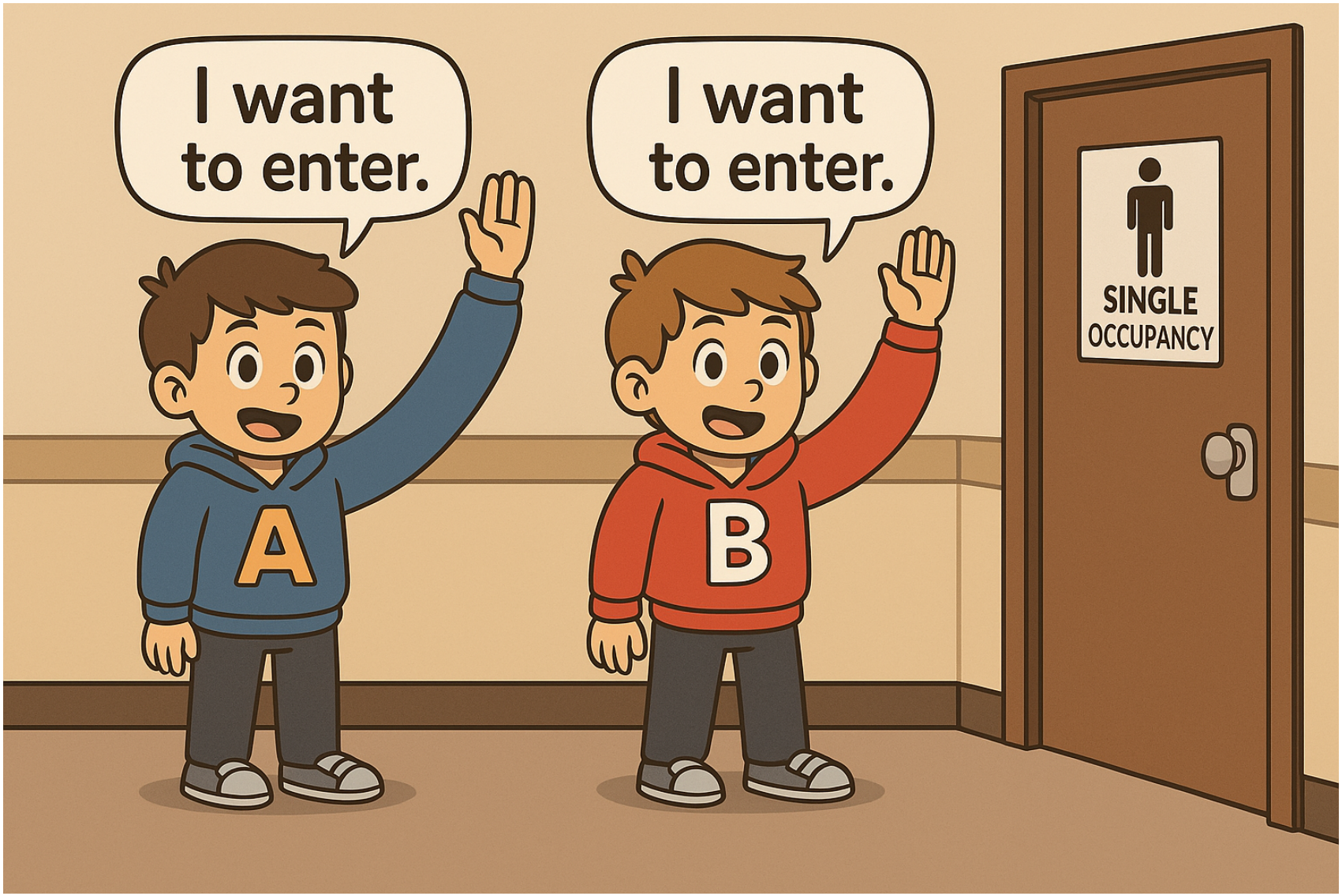
- Lower my flag (store *my flag* = down).

Condition 1: A thread requests the critical section



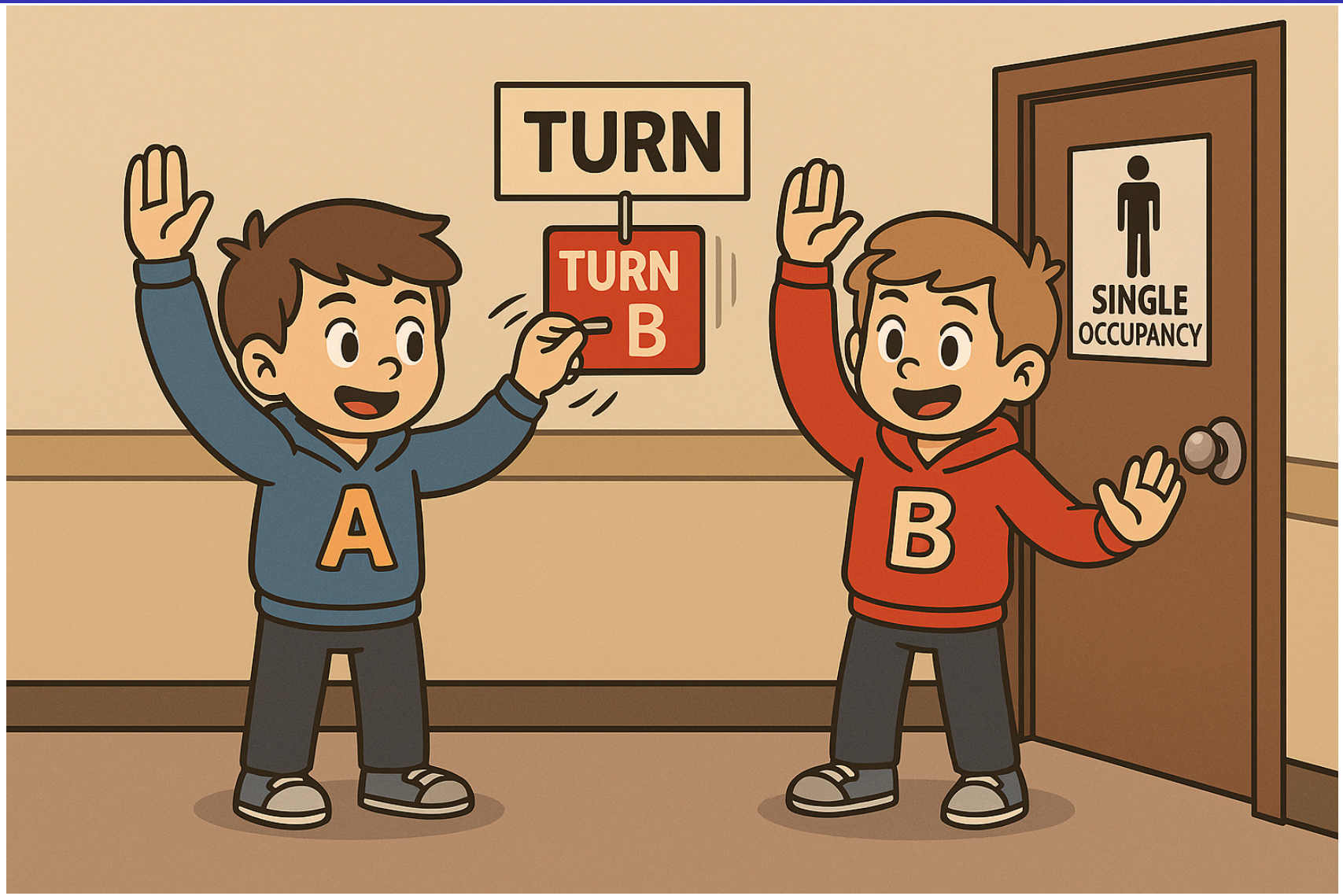
Rule: Whoever raises its flag owns turn.

Condition 2: Two threads request the critical section



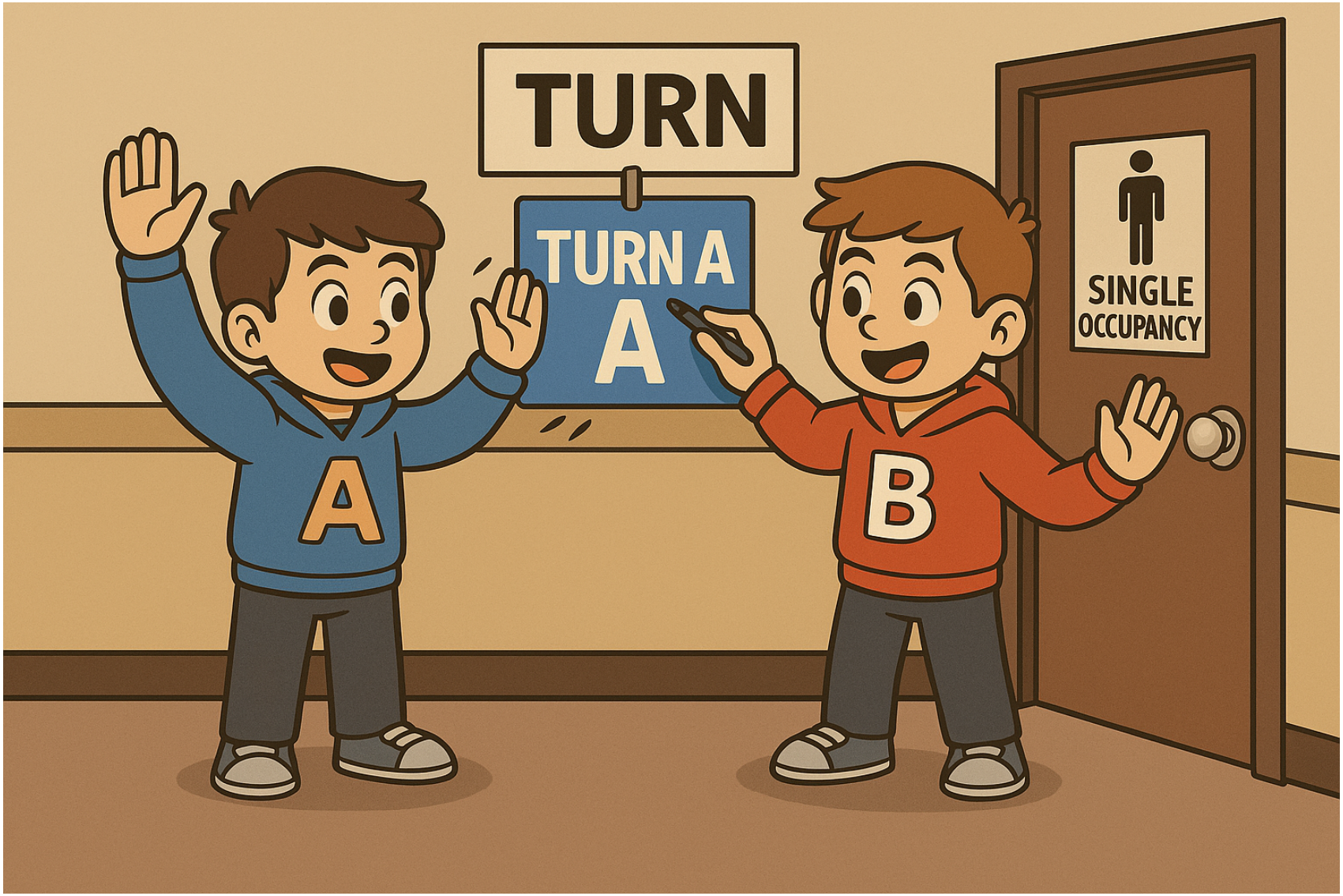
Step 1: Raise flags

Condition 2: Two threads request the critical section



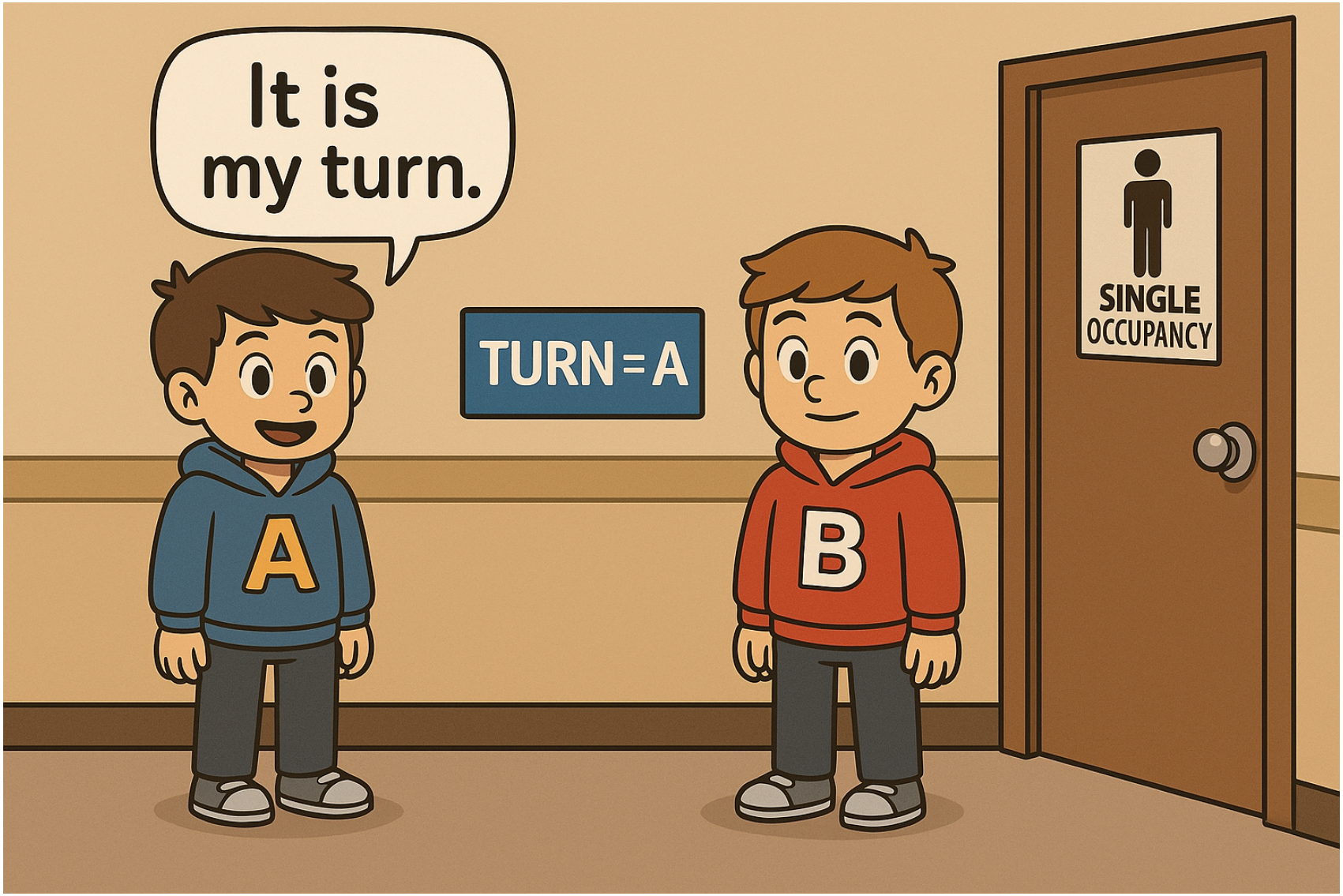
Step 2: Compete to set `turn = other`. (In the figure, A is faster than B.)

Condition 2: Two threads request the critical section



Since B is slower than A, B writes the turn label last, setting `turn = A`, so the final visible value is `turn = A`.

Condition 2: Two threads request the critical section



Peterson's Algorithm: Where the Road Went Wrong

Assumptions to abandon in concurrent programming

- Loads and stores are instantaneous and immediately visible
- Instructions execute strictly in the written program order
  - This was a reasonable belief before modern compilers and multicore CPUs.

Counterexample 1: Reordering Breaks Peterson

Two threads T0 and T1

What we write

- T0: raise my flag; set turn = 1; then spin.
- T1: raise my flag; set turn = 0; then spin.

What can happen

- Compiler hoists the read of the other flag out of the loop.
- Each thread sees the other flag as *down* once and caches that view.
- Both conclude “safe to enter” and step into the critical section.

Takeaway

- Peterson assumes fresh reads and program order.
- Reordering breaks the safety check.

Counterexample 2: Visibility Delay on Multicore

Two threads on different cores

Setup

- T0 raises its flag. The write sits in T0's cache.
- T1 raises its flag. The write sits in T1's cache.

Interleaving

- T0 reads T1's flag before propagation. Sees *down*.
- T1 reads T0's flag before propagation. Sees *down*.
- Each checks *turn*. Each finds a path to "enter".

Result

- Both enter the critical section at the same time.
- The protocol assumed instant visibility. Reality has latency.

Implementing Peterson correctly

Compiler barrier

- Prevents the compiler from reordering memory accesses.
- Example: `asm volatile("" ::: "memory")` or using `volatile` variables.

Memory barrier

- `__sync_synchronize()` = compiler barrier + hardware fence.
- ISA mappings:
  - x86: `mfence`
  - ARM: `dmb ish`
  - RISC-V: `fence rw, rw`
- Orders loads and stores so both threads observe a consistent order.
- With per-operation atomic loads/stores, this is enough to make Peterson work.

Have A Try: peterson

Spinlock: Implementing Mutual Exclusion Using Hardware

There Really Is Such an Instruction

`cli` (x86)

- Clear Interrupt Flag.
- IF bit in `EFLAGS` is 0x200.
- On a single-processor system a tight loop after `cli` can freeze the machine.

`csrci mstatus, 8` (RISC-V)

- Control and Status Register Clear Immediate.
- Clears the `MIE` bit in `mstatus`.

But

- Single-processor systems.
- Operating system kernel code only, not user space.

What Instruction Do We Need?

- Bug cause: the condition may become false by the time we write `*lock = 1;`.

```
int lock = 0;   // 0 means unlocked, 1 means locked

void acquire_lock(int *lock) {
    while (*lock != 0) {}
    *lock = 1;
}

void release_lock(int *lock) {*lock = 0;}

void *foo(void *arg) {
    acquire_lock(&lock);
    // Critical section: Do work here ...
    release_lock(&lock);
    return NULL;
}
```

We need an **atomic read–compute–write**.

Finally We Can Do 1 + 1 !

A thread-safe atomic addition operation on x86 with a locked instruction

```
asm volatile("lock addq $1, %0": "+m"(x));
```

Have A Try: sum-atomic

- `lock` prefixes the instruction to make the read–compute–write indivisible.
- All cores see a single atomic update to `sum`.
- Use such primitives to build locks and counters.

Recall. In the last lecture on loss of determinism, we showed that even if sum++ is lowered to a single assembly instruction, the result can still be incorrect.

```
// Missing lock prefixes cause loss of determinism.
asm volatile("addq $1, %0": "+m"(x))
```

An API for Lock Management

- x86: `lock`ed instructions, e.g., `lock xchg`.
- RISC–V: LR/SC pair and the A extension.
- ARM: `ldxr`/`stxr` pair, or `stadd` in Atomics.

```
void acquire_lock(int *lock) {
    while (*lock != 0) {}
    *lock = 1;
}
// Atomically write newval to *addr and return the
   previous value of *addr.
// int xchg(int *addr, int newval) {
//     int result;
//     asm volatile (
//         "lock xchg %0, %1"
//         : "+m" (*addr), "=a" (result)
//         : "1" (newval)
//         : "cc"
//     );
//     return result;
// }
```

Spinlock: API and Implementation

```
int lock = 0;   // 0 means unlocked, 1 means locked

int xchg(int *addr, int newval) {
    int result;
    asm volatile (
        "lock xchg %0, %1"
        : "+m" (*addr), "=a" (result)
        : "1" (newval)
        : "cc"
    );
    return result;
}
void acquire_lock(int *lock) {while (xchg(lock, 1)) {}}
void release_lock(int *lock) {xchg(lock, 0);}
void *foo(void *arg) {
    acquire_lock(&lock);
    // Critical section: Do work here ...
    release_lock(&lock);
    return NULL;
}
```

Have A Try: sum-spinlock

Caveat: lock/unlock as a Source of Bugs

From the moment we chose this API...

- Humans repeat the same mistakes.
- `lock` and `unlock` put the burden on the programmer.
  - You must know exactly when and with whom data is shared.
- Recall Tony Hoare's "billion-dollar mistake." Programmers will make errors.
  - Forgetting to mark a region with a lock.
  - Forgetting `unlock` on rare control paths, for example a `return` between `lock` and `unlock`.

Typical pattern

```
T1: acquire_lock(&l); sum++; release_lock(&l);
T2: acquire_lock(&I); sum++; release_lock(&I);
```

Do not laugh. This will be you.

Drawbacks (Scalability Problems) of Spin Locks

Performance issue (1)

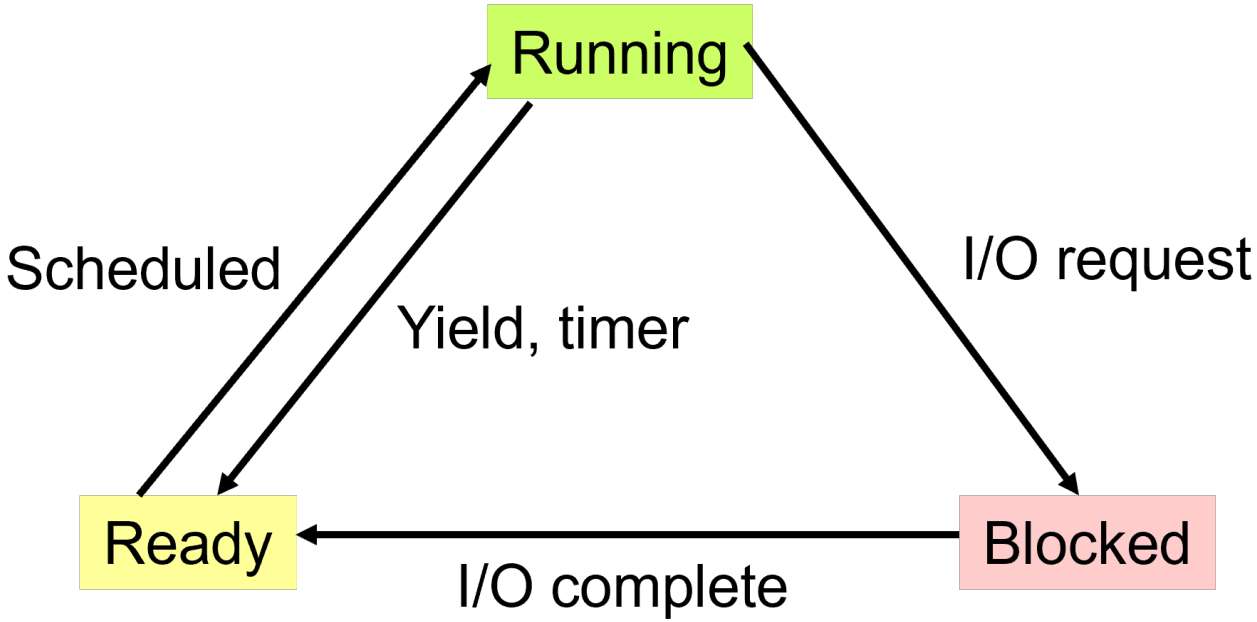
- Threads on other CPUs busy–wait while only one holds the lock.
- One core works, many cores spin.
- If the critical section is long, it is better to give the CPU to other threads.

Performance issue (2)

- Preemption is controlled by the OS, not the application.
- The thread holding the spinlock might be switched out by the operating system.
- This leads to a disastrous scenario: all other waiting threads (on other CPUs) continue to spin and waste 100% of their resources, while the only thread that can release the lock is now asleep (Ready to Run) and unable to make progress.

Explanation of Performance Issue 2

- The OS is unaware of the thread's activity
- But why can't it be?.
- Each thread can be in one of three states:
  1. *Running*: has the CPU
  2. *Blocked*: waiting for I/O or another thread
  3. *Ready to run*: on the ready list, waiting for the CPU



Explanation of Performance Issue 2 (Cont.)

- A thread waiting for a spinlock is busy-waiting (spinning) and keeps checking the lock.
- From the OS view this thread is Running (it uses the CPU) even if it does no useful work.
- When its time slice expires it moves to Ready to Run and will be rescheduled.
- The OS does not know it is busy-waiting. It treats it like any other CPU-using thread.
- This causes 100% resource waste.

Recap: Use Cases for Spinlocks

Low Competition

- The critical section is rarely contended.
- Example: only two threads occasionally touch one global for a very short time; a spinlock acquires/releases quickly with little chance of simultaneous contention.

Short Time

- Very short critical section: only a few instructions on shared data.
- A lock is held just briefly to prevent races; duration is minimal.

No Switch-Out

- The OS does not context-switch the holder during the short hold time.
- The OS (in kernels) can disable interrupts/preemption so the holder can release quickly.

In-Class Quiz

Mutexes & Futexes: When Threads Cannot Solve Busy-Waiting, Let the OS Help.

Mutex: Implementing the lock in the OS

Problem

- When a thread waits for a lock, it wastes CPU cycles by spinning.
- Let other threads use the CPU instead of busy waiting.

Solution

- If the lock is unavailable, block the thread.
- When the lock becomes available, wake the thread. This saves CPU time.

- User-space C code cannot block or wake threads by itself. It only computes.
- We need to use system calls to interact with the operating system.

Understanding Mutex with a Library Key Analogy

Analogy

- Operating System = **Library Desk Manager**
- Critical Section = **Study Room**
- Lock = **Key to the Study Room**

1. The first person (Thread 1) arrives and requests the key

- Thread 1 makes a **system call** to request the lock (the key).
- If the lock is available, the OS (desk manager) **hands over the key** immediately.
- The syscall **returns at once**: Thread 1 has acquired the lock and may enter the study room.
- Lock acquired, syscall completed.

Understanding Mutex with an Analogy (Cont.)

2. The second person (Thread 2) arrives but the lock is occupied

- Thread 2 makes a system call to request the lock, but Thread 1 already holds it.
- The OS places Thread 2 on the lock's **wait queue**; the syscall does not return.
- Thread 2 is blocked, waiting for the lock to be released.
- Thread 2 is queued, waiting for the key.

Understanding Mutex with an Analogy (Cont.)

3. The first person (Thread 1) finishes and releases the lock

- Thread 1 finishes and makes a syscall to **release** the lock (return the key).
- The OS checks the lock's wait queue and finds Thread 2 waiting.
- The OS hands the lock to Thread 2 and the syscall returns, **waking up** Thread 2, which now acquires the lock and continues.

Understanding Mutex with an Analogy (Cont.)

4. If no one is waiting

- If no other thread is in the wait queue, the OS marks the lock as available.
- Lock released.

5. We still need spinlocks

- When many threads request a mutex via system calls, the OS must control access to the lock metadata.
- During allocation or release, the kernel uses a short spinlock to keep these updates atomic and to stop two threads from changing the lock state at the same time.
- The spinlock protects only this brief kernel critical section, then it is released right after the lock state is updated, which saves CPU time.
- Classic use case: a *short* critical section inside the kernel.

pthread Mutex Lock

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);

pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);
```

Use this in practice:

- Very good performance when there is no contention
  - It does not even need to trap into the OS kernel
- Scales well when more threads contend

Scalability Performance

A system should keep performance and stability as demand or load grows. It should scale resources flexibly. (Another view: as resources increase, it should maintain or improve throughput.)

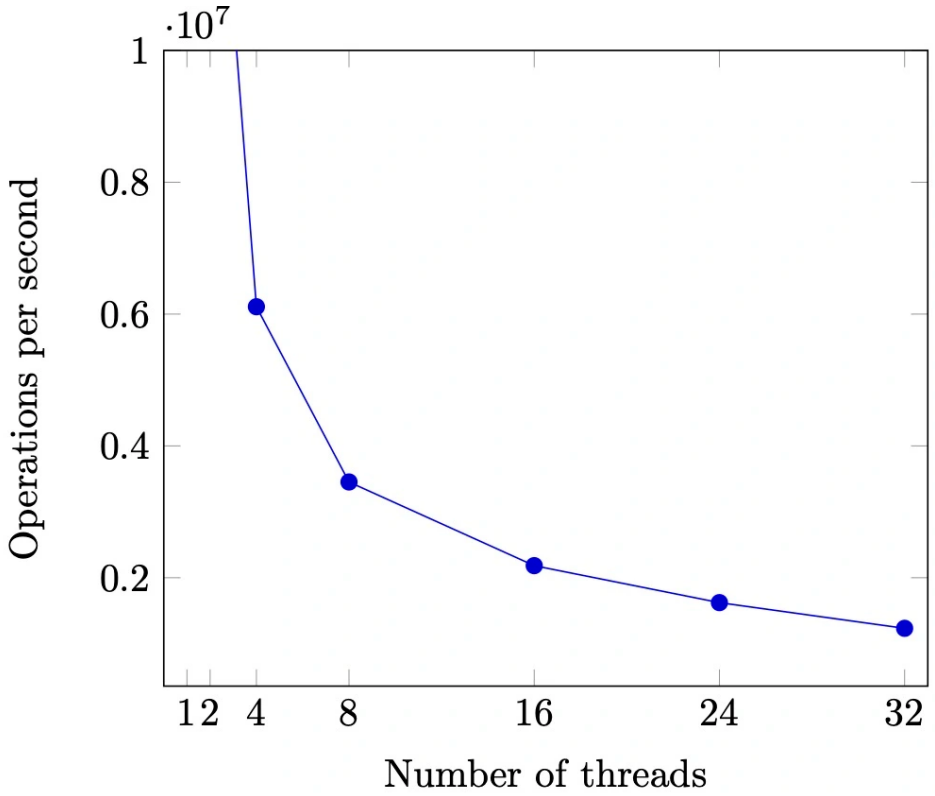


Figure: Throughput drops as the number of threads increases.

Have A Try: scalability

Some Analysis on Mutual Exclusion

Spinlock (threads share the locked variable directly)

- **Faster fast path:** if `xchg` succeeds, the thread enters the critical section immediately with minimal overhead.
- **Slower slow path:** if `xchg` fails, the thread spins in a loop and wastes CPU cycles.

Mutex (access via system calls)

- **Faster slow path:** if locking fails, the thread does not use the CPU. It sleeps.
- **Slower fast path:** even on success, entering and exiting the kernel adds syscall overhead.

Futex: Fast Userspace muTexes

Why choose when you can have both?

- **Fast path:** one atomic instruction in user space; if it succeeds, return immediately.
- **Slow path:** if it fails, call a **system call** to sleep.

Futex = Spin + Mutex

Common performance idea

- Optimize for the average (frequent) case, not the worst case.

The POSIX threads mutex (`pthread_mutex`) is implemented with futexes.
Have A Try: scalability using `strace -f -c ./mutex`

More On Futex

More complex than you think

- If there is no contention the fast path must not call `futex_wake`
- When spinning fails → call `futex_wait` → the thread sleeps
  - What if the lock is released right after the syscall begins?
  - What if an interrupt can occur at any time?
- But ChatGPT can help us!

Concurrency: the iceberg below the surface

- LWN: A futex overview and update
- Futexes are tricky by Ulrich Drepper

Takeaways

Concurrency programming is hard. A practical way to handle this complexity is to fall back to non-concurrency. We can use lock and unlock inside threads to enforce mutual exclusion. Any code protected by the same lock loses the opportunity to run in parallel (the execution order is still uncontrolled). Implementing mutual exclusion is challenging. Modern systems build it with atomic operations in threads, interrupt control in the kernel, atomic primitives, and spinning. Note that as long as the parallelizable portion of a program is large enough, serializing a small part will not cause a fatal performance loss.