Sistemi Operativi

Corso di Laurea in Informatica

Sapienza Università di Roma

a.a. 2026/2027

Marco Raoul Marini

marini@di.uniroma1.it

Gabriele Tolomei

tolomei@di.uniroma1.it

Lecture 3.2: Concurrency II

Locks: Test-and-Set, CAS, Spinlock

Opening Thought

> *"A lock is just a variable ... It is either available (or unlocked or free) and thus no thread holds the lock, or acquired (or locked or held), and thus exactly one thread holds the lock and presumably is in a critical section."*
> *— Remzi & Andrea Arpaci-Dusseau, OSTEP Ch. 28*

In Lecture 3.1 we used `pthread_mutex_lock()` as a **black box**. Today we open the box. The storyline:

1. **Simple tricks fail:** turning interrupts off, or using only ordinary loads and stores.
2. **Hardware atomic instructions** (TAS, CAS, FAA) give us correct locks.
3. **But waiting threads spin** and waste CPU.
4. **OS-assisted sleeping locks** (park/unpark, `futex`, two-phase) fix the waste.

Lecture Roadmap

Eight sections, following the storyline *simple tricks fail → hardware atomics → spinning waste → sleeping locks*:

1. **Recap & Motivation: What a Lock Must Do** — correctness, fairness, performance
2. **Controlling Interrupts** — the uniprocessor idea and why it fails
3. **Failed Software-Only Attempts** — shared flag, alternation, Peterson in brief
4. **Hardware Atomics and the TAS Spinlock** — atomic read-modify-write, first working lock, its cost
5. **Compare-and-Swap (CAS)** — CAS spinlock, the retry loop
6. **Fetch-and-Add and the Ticket Lock** — a fair lock
7. **OS-Assisted Sleeping Locks & Two-Phase Locks** — park/unpark, `futex`, spin-then-sleep
8. **Putting It Together & Summary** — comparison of all locks

> ***Appendix A.1–A.7*** *holds the extended material (kernel interrupts, Peterson, LL/SC, traces and TTAS, ABA, futex code, exercises); dividers point to it.*

Prerequisite Check

Before proceeding, ensure you are comfortable with:

- **Critical section and race conditions** (Lecture 3.1): why `counter++` is unsafe.
- **Mutual exclusion, progress, bounded waiting** (Lecture 3.1): the three properties a solution must have.
- `pthread_mutex_lock()` / `unlock()` (Lecture 3.1): the API we implement today.
- **Preemption and thread states** (Lectures 2.2–2.4, 3.1): timer interrupt, READY / RUNNING / BLOCKED.
- **Kernel vs. user mode** (Lecture 1.2): privileged instructions.
- **Memory reordering, intuition only** (Lecture 3.1, App. A.8): the CPU may make stores visible out of order.

New today: atomic read-modify-write, TAS, CAS, FAA, spinlock, ticket lock, park/unpark, `futex`, two-phase lock.

§ 1 — Recap & Motivation: What a Lock Must Do

Goal: fix the criteria we will use to judge every lock in this lecture.

Recap: The Critical Section Problem

From Lecture 3.1, we need a mechanism that enforces safe access to shared state:

```
// Without a lock — BROKEN:
counter++;           // load, add, store — not atomic

// With a lock — CORRECT:
lock(&mutex);
counter++;           // now executes as if atomic
unlock(&mutex);
```

The lock enforces that only **one thread at a time** executes the critical section. Any other thread calling `lock()` while the lock is held must **wait** — by spinning (looping) or by sleeping — until the lock is released.

> ***Question for today:*** *how do we build* `lock()` *and* `unlock()` *correctly and efficiently?*

Three Desiderata for Evaluating a Lock

1. Correctness = mutual exclusion + progress

At most one thread is inside the critical section; and if the lock is free and someone wants it, someone gets it. **Non-negotiable.**

2. Fairness = bounded waiting

Can a waiting thread **starve**? *Unfair:* T1 releases and re-acquires again and again while T2 waits forever. *Strict FIFO:* nobody overtakes T2, so with n threads it waits for at most $n - 1$ acquisitions.

3. Performance

What does the lock cost? **Uncontended** (no competitor) vs. **contended** (many waiters).

> *Properties 1–2 are the requirements from Lecture 3.1; performance is new.* ***Preview:*** *TAS/CAS spinlocks are unfair, the ticket lock is FIFO,* `futex` *mutexes are approximately fair.*

✅ Checkpoint 1 — Recap & Motivation

Test your understanding:

1. A lock is FREE. Three threads call `lock()` simultaneously. What must happen? Must we say *which* thread wins?
2. State the three desiderata. Which one is non-negotiable?
3. Define **starvation** and give a scenario where it occurs.
4. A thread calls `lock()` on a held lock. Name the two ways it can wait. (We compare them in § 7.)

§ 2 — Controlling Interrupts

Goal: see why the most obvious idea, switching preemption off, is not enough.

📎 *Extended material → Appendix A.1*

Disabling Interrupts: Simple, but It Fails

Idea: on a uniprocessor a thread is preempted only when the timer interrupt fires, so switch interrupts off inside the critical section.

```
void lock(lock_t *l)   { disable_interrupts(); }   // CLI on x86
void unlock(lock_t *l) { enable_interrupts();  }   // STI on x86
```

Why it fails:

1. **Trust:** `cli` is privileged. Handing it to user code means a thread that loops forever (or crashes) in its critical section **freezes the machine**.
2. **Multicore:** interrupts are off on *one* core only; a thread on another core still enters the critical section.
3. **Long sections:** timer, disk and network interrupts are delayed or lost while they are off.

> *Verdict: only the trusted kernel, for a few instructions (Appendix A.1). Can threads themselves coordinate through ordinary shared variables?*

§ 3 — Failed Software-Only Attempts

Goal: see that ordinary loads and stores are not enough, even when cleverly combined.

📎 *Extended material → Appendix A.2*

Attempt 1: A Single Shared Flag

Idea: a variable `flag` says whether the lock is taken: wait until it is 0, then set it to 1.

```
int flag = 0;                                   // 0 = free, 1 = taken
void lock()   { while (flag == 1) ; flag = 1; }
void unlock() { flag = 0; }
```

Thread 1	flag	Thread 2
① while (flag == 1); reads 0 → exits loop	0	*(not running)*
(preempted between check and set)	0	② while (flag == 1); reads 0 → exits loop
(still preempted)	1	③ flag = 1; enters the critical section
④ flag = 1; enters the critical section	1	*(inside the CS)*

BOTH threads are in the critical section → mutual exclusion violated

Check and set are **two separate steps**. Testing a condition and acting on it must be **one atomic action**.

Alternation and Peterson: Close, but Not Enough

- **Attempt 2, strict alternation** (a shared `turn` variable): mutual exclusion holds, but threads can only enter in the order T0, T1, T0, T1… If T0 is busy elsewhere, T1 waits forever though the CS is free: **progress violated**.
- **Attempt 3, Peterson's algorithm** (`flag[i]` = "I want in" plus `turn` as tie-breaker; the thread that wrote `turn` last waits): correct for two threads **if** every write is seen in program order (sequential consistency). Code and trace: Appendix A.2.1–A.2.2.
- **But real CPUs** keep stores in a **store buffer**, so a later *load* can overtake an earlier *store* (store→load reordering, allowed on x86 and ARM). Both threads can read the other's flag as 0 and **both enter the CS**. A fence fixes it, but a fence is a hardware instruction.

> *No pure-software lock is both correct and practical on modern hardware (details: Appendix A.2.3–A.2.4).*

So What Do We Need From the Hardware?

> ***Correct, efficient mutual exclusion cannot be built from ordinary loads and stores alone on modern hardware.***

A useful atomic primitive must provide:

- **Atomicity:** read **and** write happen as one indivisible step, so no other CPU can slip in between.
- **Visibility:** once it completes, its effect is visible to all processors.
- **Ordering:** with the default sequentially consistent ordering (x86 locked instructions, C11 `seq_cst`) it acts as a barrier: memory operations are not reordered across it.

This is exactly what Attempt 1 was missing: *test* and *set* fused into **one** instruction.

✅ Checkpoint 2 — Failed Software-Only Attempts

Test your understanding:

1. Describe the exact interleaving that lets two threads enter the CS in the single-flag attempt.
2. Strict alternation violates **progress**. Give a concrete example.
3. Why can a software-only algorithm that is correct under sequential consistency (Peterson) fail on a modern CPU?
4. What property must a hardware primitive have that ordinary loads and stores lack?

§ 4 — Hardware Atomics and the TAS Spinlock

Goal: meet the atomic instructions, then build and judge the first correct lock.

📎 *Extended material → Appendix A.3 (LL/SC), A.4 (traces, cache effects, TTAS)*

Atomic Read-Modify-Write Primitives

An **atomic read-modify-write (RMW)** instruction reads a memory word, computes a new value, and writes it back as **one indivisible step**.

Primitive	What it does	x86-64	Natural use
Test-and-Set (TAS)	write 1, return the **old** value	`xchg`	simplest lock
Compare-and-Swap (CAS)	write *new* **only if** value $=$ *expected*	`cmpxchg`	conditional update, lock-free code
Fetch-and-Add (FAA)	add δ, return the **old** value	`lock xadd`	counters, tickets

Plan: TAS gives the first working lock (next slides). CAS builds the same lock and more (§ 5). FAA gives a *fair* lock (§ 6).

> *RISC CPUs (ARM, RISC-V) offer **LL/SC** instead of CAS: see **Appendix A.3**. On x86-64, `xchg` with a memory operand is implicitly locked.*

Test-and-Set: Definition

Test-and-Set atomically writes a new value and returns the old one:

```
// What the hardware does in ONE indivisible step:
int TestAndSet(int *ptr, int new_value) {
    int old = *ptr;      // read the old value
    *ptr = new_value;    // write the new value (1 for a lock)
    return old;          // report what was there before
}
// C11: atomic_exchange(ptr, new_value)      x86-64: xchg
```

How a lock uses it, with `flag` $\in \{0 = \text{FREE},\ 1 = \text{HELD}\}$:

- `TestAndSet(&flag, 1)` returns **0** $\Rightarrow$ it was FREE, we just made it HELD: **we own the lock**.
- returns **1** $\Rightarrow$ it was already HELD (we wrote 1 over 1, no change): **try again**.

The TAS Spinlock

```c
#include <stdatomic.h>
typedef struct { atomic_int flag; } spinlock_t;    // 0 = FREE, 1 = HELD

static inline int TestAndSet(atomic_int *p, int v) { return atomic_exchange(p, v); }

void spin_init(spinlock_t *l) { atomic_init(&l->flag, 0); }

void spin_lock(spinlock_t *l) {
    while (TestAndSet(&l->flag, 1) == 1)
        ;                           // returned 1: lock was HELD -> spin
}                                   // returned 0: lock was FREE, now ours

void spin_unlock(spinlock_t *l) {
    atomic_store_explicit(&l->flag, 0, memory_order_release);   // mark FREE
}
```

- A **release** store makes the critical section's writes visible *before* the lock looks FREE. On x86 it compiles to a plain `mov`.
- Usage: `spin_lock(&l); counter++; spin_unlock(&l);` Step-by-step trace: Appendix A.4.1.

Evaluating the TAS Spinlock

Correctness ✅ — If several threads call TAS simultaneously, the hardware serializes them: exactly **one** gets the old value 0; all others get 1. So at most one thread passes the loop until `unlock`. If the lock is free, someone gets it (progress).

Fairness ❌ — There is no queue: whoever's TAS happens to run next wins. A thread can lose again and again ⇒ **starvation**.

Performance ❌ **under contention** — Waiters burn CPU in the loop.

- **Uniprocessor:** the holder was preempted inside the CS; a spinner wastes its *entire time slice* with zero chance of progress. Always wasteful.
- **Multicore, short CS:** the holder runs on another core and releases soon, so spinning is acceptable.

> *Cache-line bouncing under many spinners:* ***Appendix A.4****.*

✅ Checkpoint 3 — Test-and-Set and the Spinlock

Test your understanding:

1. What does `TestAndSet(ptr, 1)` return, and what does it leave in `*ptr`? What does a return of 0 vs. 1 mean for the lock?
2. Argue informally that the TAS spinlock gives mutual exclusion. Which property of TAS is essential?
3. Why is spinning on a uniprocessor always suboptimal? What should the waiter do instead?
4. Why is the TAS spinlock unfair? Describe a starvation scenario.

§ 5 — Compare-and-Swap (CAS)

Goal: learn a more general atomic instruction and what it adds over TAS.

📎 *Extended material → Appendix A.5*

Compare-and-Swap: Definition and Spinlock

TAS *always* writes. **CAS** writes only if the word still holds the value we expect:

```
int CompareAndSwap(int *ptr, int expected, int new_value) {   // ONE indivisible step
    if (*ptr == expected) { *ptr = new_value; return 1; }     // success
    return 0;                                                 // failure: nothing written
}   // C11: atomic_compare_exchange_strong(ptr, &expected, new)   x86-64: cmpxchg

void cas_lock(cas_lock_t *l) {                  // flag: 0 = FREE, 1 = HELD
    int expected;
    do { expected = 0; }                        // reset every time: on failure C11
    while (!atomic_compare_exchange_strong(&l->flag, &expected, 1));  // overwrites expected
}
void cas_unlock(cas_lock_t *l) { atomic_store_explicit(&l->flag, 0, memory_order_release); }
```

- `CAS(ptr, 0, 1)` emulates TAS, but TAS cannot emulate CAS: CAS is **more general**.
- As a lock, the CAS spinlock behaves exactly like the TAS spinlock: correct ✅, unfair ❌, spinning waste ❌.

What CAS Adds: The Read–Compute–CAS Retry Loop

CAS can update a word by **any function** of its old value, without holding a lock:

```
int cas_increment(atomic_int *p) {
    int old, new;
    do {
        old = atomic_load(p);                    // 1. read
        new = old + 1;                           // 2. compute (any function of old)
    } while (!atomic_compare_exchange_strong(p, &old, new));   // 3. commit if *p == old
    return new;
}
```

- CAS **fails** only if another thread changed `*p` between steps 1 and 3 $\Rightarrow$ retry with the fresh value.
- Two threads both read 5: one CAS succeeds ($p = 6$), the other fails, re-reads 6, and writes 7. No update is lost.

> *This pattern builds **lock-free** data structures (stack: Appendix A.5), but it has a trap: the **ABA problem**.*

✅ Checkpoint 4 — Compare-and-Swap (CAS)

Test your understanding:

1. Write the pseudocode of `CompareAndSwap(ptr, expected, new)`. What does it return on success and on failure?
2. In `cas_lock`, why must `expected = 0` be reset on every iteration? How does the CAS spinlock compare with the TAS one?
3. Two threads run `cas_increment` on $p = 5$ at the same time. Trace both and give the final value of p.
4. Name one thing CAS can do that TAS cannot.

§ 6 — Fetch-and-Add and the Ticket Lock

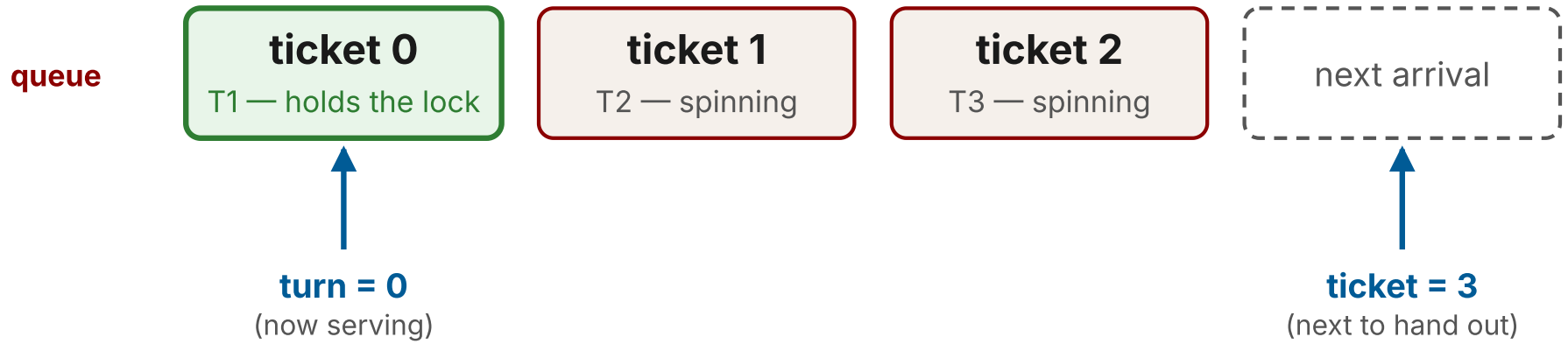
Goal: fix the unfairness of TAS/CAS locks using a third primitive.

📎 *Extended material → Appendix A.4.4–A.4.5 (trace, evaluation)*

Ticket Lock: The Idea

Problem: TAS/CAS waiters have no order. **Idea:** number the arrivals and serve in order, using **Fetch-and-Add**: `FetchAndAdd(ptr, d)` atomically does `*ptr += d` and returns the **old** value (C11 `atomic_fetch_add`, x86-64 `lock xadd`), so two callers never get the same number.

Two shared counters, both 0 at start: **`ticket`** (next number to hand out) and **`turn`** (number being served; `unlock()` increments it). A thread takes `my_ticket = FetchAndAdd(&ticket, 1)`, then spins until `turn == my_ticket`.



State after T1, T2, T3 arrived in this order. `unlock()` does `turn++`, so T2 is served next.

Ticket Lock: Implementation

```
typedef struct {
    atomic_uint ticket;   // next ticket to dispense (starts at 0)
    atomic_uint turn;     // ticket currently being served (starts at 0)
} ticket_lock_t;

void ticket_lock(ticket_lock_t *l) {
    unsigned my_ticket = atomic_fetch_add(&l->ticket, 1);   // take a ticket (FAA)
    while (atomic_load(&l->turn) != my_ticket) {
        /* spin until we are served */
    }
}                                   // turn == my_ticket: we hold the lock

void ticket_unlock(ticket_lock_t *l) {
    atomic_fetch_add(&l->turn, 1);  // serve the next ticket
}
```

FIFO ✅: nobody overtakes, so with n threads a waiter sees at most $n - 1$ acquisitions. **Still a spinlock** ❌: we fixed fairness, not the CPU waste. Trace and evaluation: Appendix A.4.4–A.4.5.

> *Every lock so far makes waiters **burn CPU**. Time to let them sleep (§ 7).*

✅ Checkpoint 5 — Fetch-and-Add and the Ticket Lock

Test your understanding:

1. What does `FetchAndAdd(ptr, 1)` return? What does `*ptr` contain afterward?
2. In the ticket lock, what do `ticket` and `turn` represent?
3. T1 holds the lock; T2 and T3 arrived next (tickets 1 and 2). While T1 still holds it, T4 arrives. Which ticket does T4 get, and at which value of `turn` does it enter?
4. The ticket lock is fair but still a spinlock. What problem does it not solve?

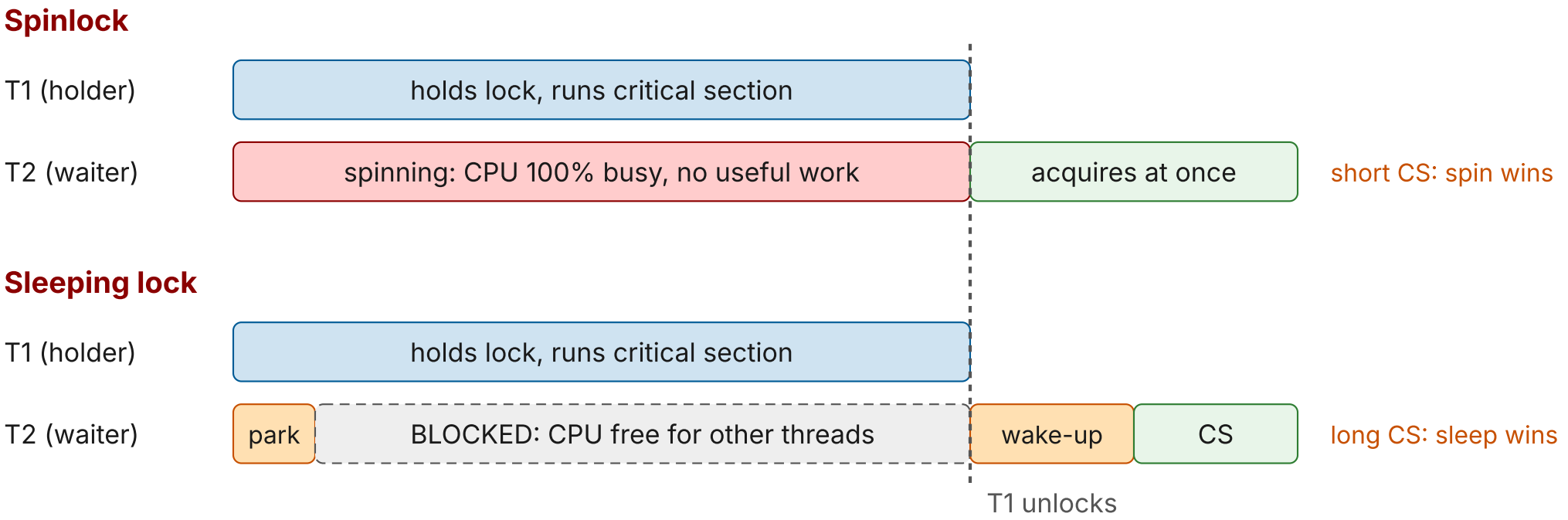
§ 7 — OS-Assisted Sleeping Locks & Two-Phase Locks

Goal: stop wasting CPU: let waiting threads sleep, with the kernel's help.

📎 *Extended material → Appendix A.6*

The Cost of Spinning vs. Sleeping

When `lock()` finds the lock held, a thread has three options: **(1) spin** (busy-wait), **(2) yield** (`sched_yield()` in a loop: a context switch per retry, Appendix A.6.1), **(3) sleep** (go BLOCKED until woken).



Sleeping costs a **park + wake-up** (a few µs); spinning costs CPU **for the whole wait**.

Sleeping Locks: `park()`, `unpark()` and a Wait Queue

The kernel moves a thread between `BLOCKED` and `READY` (primitives from Solaris: the same idea as Linux `futex`):

```
void park(void);              // put the CALLER to sleep: RUNNING -> BLOCKED
void unpark(pthread_t tid);   // wake thread tid:         BLOCKED -> READY

typedef struct {
    int     flag;         // 0 = FREE, 1 = HELD  (the real lock)
    int     guard;        // tiny TAS spinlock protecting flag + waitqueue
    queue_t waitqueue;    // FIFO of sleeping thread IDs (unlock() must know whom to wake)
} mutex_t;
```

- **`flag`** protects the user's critical section, possibly held for a *long* time.
- **`guard`** protects only the mutex's own fields, for a **few instructions**: spinning on it is cheap.
- Threads that find `flag == 1` enqueue themselves and `park()`: 0% CPU.

Wait-Queue Mutex: `lock()` and `unlock()`

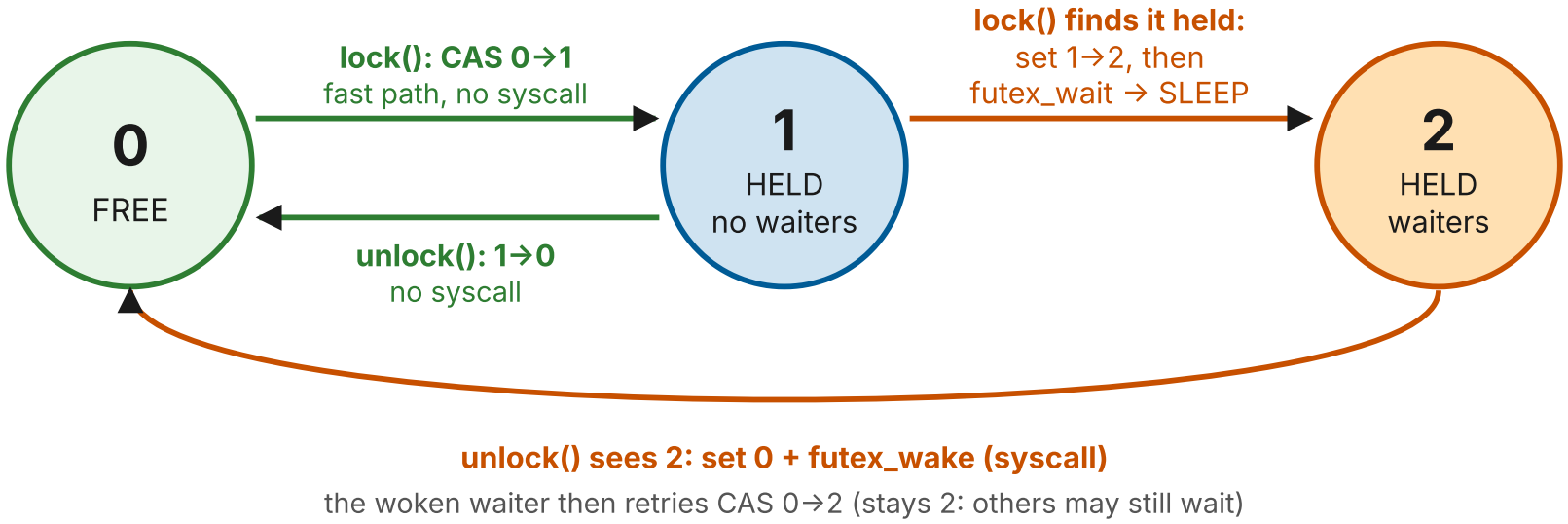
```
void mutex_lock(mutex_t *m) {
    guard_acquire(m);                          // spin on m->guard (TAS), briefly
    if (m->flag == 0) { m->flag = 1; guard_release(m); }          // free: take it
    else {
        queue_add(&m->waitqueue, gettid()); // held: join the queue
        setpark();                          // "I am about to park" (see below)
        guard_release(m);                   // release guard BEFORE sleeping
        park();                             // sleep until unpark(): lock handed to us
    }
}
void mutex_unlock(mutex_t *m) {
    guard_acquire(m);
    if (queue_empty(&m->waitqueue)) m->flag = 0;                  // nobody waits: FREE
    else unpark(queue_remove(&m->waitqueue));  // hand-off: flag stays 1
    guard_release(m);
}
```

*The guard is released before `park()`. If the holder unparks us in that window, the wake-up would be **lost**; `setpark()` makes the following `park()` return immediately. Trace: **Appendix A.6.2**.*

Linux `futex`: A Sleeping Lock Done Fast

A kernel wait queue **per memory address**; no syscall unless contended (orange arrows). Code: **Appendix A.6.3**.

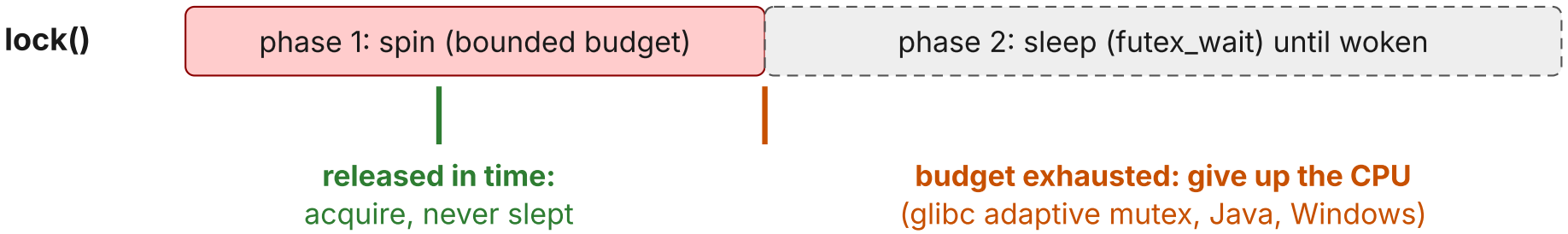
- `futex(addr, FUTEX_WAIT, expected)`: sleep only if `*addr == expected`, checked **atomically in the kernel** (no lost wake-up, like `setpark()`).
- `futex(addr, FUTEX_WAKE, n)`: wake up to n sleepers.



The Two-Phase Lock

A **two-phase lock** combines the best of spinning and sleeping.

- **Phase 1 — spin** for a short, bounded time: if the critical section is short, the lock is released during the spin and we acquire it with no sleep/wake-up cost.
- **Phase 2 — sleep** if the lock is still held: the wait is long, so spinning would only waste CPU.
- **Spin budget:** roughly the cost of one context switch (a few µs), so we never waste much more than sleeping would have cost.



> *glibc's default `pthread_mutex_t` goes straight from the fast-path CAS to `futex_wait` (no spin phase).*

✅ Checkpoint 6 — Sleeping Locks, `futex` and Two-Phase Locks

Test your understanding:

1. Describe the two phases of a two-phase lock and the rationale for each.
2. When does a `futex`-based mutex make **no** syscall? When does it fall back to `futex(FUTEX_WAIT)`?
3. The wait-queue mutex exists to avoid spinning, yet it uses a `guard` spinlock. Why is that acceptable?
4. Which lock would you choose for: (a) a short critical section (say, updating two related fields) in a hot loop on 8 cores; (b) a transaction that may hold the lock for 100 ms; (c) a short critical section where no thread may starve?

§ 8 — Putting It Together & Summary

Goal: compare every lock against our three desiderata and consolidate.

Evaluation Table: All Lock Implementations

Lock	Correct	Fair	Uncontended	Contended
Interrupt disable	✅ 1 CPU only	n/a	✅	❌ no use on multicore
Single flag	❌	—	—	—
Peterson's	⚠️ SC only	✅	✅	❌ spins
TAS / CAS spinlock	✅	❌	✅	❌ spins, cache traffic
Ticket lock	✅	✅ FIFO	✅	❌ spins
Yield lock (A.6.1)	✅	❌	✅	⚠️ switch per retry
Wait-queue mutex	✅	✅ FIFO	⚠️ guard + queue	✅ no CPU waste
`futex` mutex	✅	⚠️ approx.	✅	✅ no CPU waste
Two-phase lock	✅	⚠️ approx.	✅	✅ spin if short, else sleep

Summary

- **Simple tricks fail:** disabling interrupts (trust, multicore) and ordinary loads/stores (Peterson needs sequential consistency) cannot give general mutual exclusion.
- **Hardware atomics work:** TAS and CAS give a correct (but unfair) spinlock; FAA gives the fair **ticket lock**. CAS also enables lock-free retry loops.
- **Spinning wastes CPU:** worst on a uniprocessor or with long critical sections.
- **Sleeping locks** keep a wait queue and use `park` / `unpark`; `futex` adds a user-space fast path, entering the kernel only under contention.
- **Two-phase locks** spin briefly, then sleep. **Rule of thumb:** short CS and cores to spare $\Rightarrow$ spin; long CS or contention $\Rightarrow$ sleep.

Lecture 3.2 Complete

Recommended Reading

- **OSTEP Ch. 28:** *Locks*
- Available free at: https://ostep.org

Contact & Q&A

- **Lecturer:** Gabriele Tolomei
- **Email:** tolomei@di.uniroma1.it
- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!

§ Appendix

Appendix Map

Section	Extends	What you will find	Slides
A.1	§ 2	Legitimate use of interrupt disabling in OS kernels	1
A.2	§ 3	Peterson's algorithm: code, interleaving trace, scalability, store-buffer failure	4
A.3	§ 4	Load-Linked / Store-Conditional (LL/SC) and a multi-architecture instruction table	2
A.4	§ 4, § 6	TAS trace, cache-line thrashing, TTAS, ticket-lock trace and evaluation	5
A.5	§ 5	CAS lock-free stack, the ABA problem, version-tag solution	3
A.6	§ 7	Yield overhead, the lost wake-up race, `futex` code, Amdahl's Law	4
A.7	§ 8	Exercises: Peterson on weak memory, ABA hazard	2

Supplementary material for deeper study. Not required during the main lecture.

📎 A.1 — Interrupt Control: Kernel Use Cases

A.1.1 · Where Interrupt Disabling Is Still Used in Operating Systems

Despite its limitations, interrupt disabling has one legitimate use: **short critical sections inside the kernel**.

When the kernel updates its own data structures (scheduler run queue, allocator free list, interrupt vector table), it may briefly disable interrupts so a re-entrant interrupt handler cannot corrupt the structure.

Example (xv6 kernel):

```
acquire(lock):
    push_off();               // disable interrupts on this CPU (nestable)
    acquire_spinlock(lock);   // mutual exclusion across CPUs
    ...
```

This is safe because:

1. The kernel is trusted (no malicious code).
2. The critical section is extremely short (a few instructions).
3. On multicore, the kernel uses a spinlock **in addition** to disabling interrupts.

📎 A.2 — Peterson's Algorithm: In-Depth Analysis

A.2.1 · Peterson's Algorithm

Idea: each thread announces interest with `flag[i]`. If both want in, each *politely gives way* by writing `turn = j`: the thread that wrote `turn` **last** waits.

```
int flag[2] = {0, 0};   // flag[i] = 1: thread i wants to enter the CS
int turn = 0;            // tie-breaker

void lock(int i) {
    int j = 1 - i;       // the other thread
    flag[i] = 1;         // declare intent
    turn = j;            // give priority to the peer
    while (flag[j] == 1 && turn == j) { /* spin */ }
}
void unlock(int i) { flag[i] = 0; }
```

Under **sequential consistency** it satisfies mutual exclusion, progress and bounded waiting for two threads.

A.2.2 · Peterson's Algorithm: Interleaving Trace

Both threads want in. T0 writes `turn` first, T1 writes it **last** (so T1 gives way):

```
T0: flag[0] = 1;  turn = 1;          // T0 gives way to T1
T1: flag[1] = 1;  turn = 0;          // T1 overwrites turn: T1 gives way to T0

T0 checks: flag[1] == 1 && turn == 1 ?   turn is 0  -> false -> T0 ENTERS the CS
T1 checks: flag[0] == 1 && turn == 0 ?   both true  -> true  -> T1 SPINS
T0 unlock: flag[0] = 0                   T1's condition becomes false -> T1 enters
```

Only one thread is ever inside: **the thread that wrote `turn` last waits**. If T1 is not interested (`flag[1] == 0`), T0 enters immediately.

Under **sequential consistency**, Peterson's algorithm satisfies mutual exclusion, progress and bounded waiting.

A.2.3 · Scalability: Why Peterson Does Not Generalize Easily

Peterson's algorithm as shown works only for **two threads**. Generalizing it to N threads needs more complex variants (e.g., Lamport's Bakery Algorithm), which are:

- Harder to implement correctly.
- Costly: $O(N)$ shared variables, and $O(N)$ reads per acquisition.
- Still dependent on sequential consistency.

No software-only lock scales cleanly to arbitrary numbers of threads.

A.2.4 · Peterson's Failure Under Out-of-Order Execution

Peterson's algorithm assumes every write becomes visible to the other thread in program order (sequential consistency). Real CPUs and compilers break this assumption for performance.

```
Thread 0:                Thread 1:
flag[0] = 1;             flag[1] = 1;
turn = 1;                turn = 0;
// read flag[1]          // read flag[0]

x86 store buffer: each core holds its stores locally before they reach the cache.
  T0 reads flag[1] while its own stores are still buffered, and T1's store is not visible yet.
  T1 reads flag[0] likewise. Both read 0 -> both skip the loop -> both enter the CS!
```

- **Fix:** a full memory fence (`mfence`) between the writes and the reads.
- But the fence is a hardware operation, so the solution is no longer "pure software". Full walk-through: **Exercise 1 (A.7.1)**.

📎 A.3 — Load-Linked / Store-Conditional (LL/SC)

A.3.1 · Load-Linked / Store-Conditional (LL/SC) Mechanics

Used on RISC architectures as an instruction pair:

- **LL (Load-Linked):** loads a value and marks the address as "reserved" by this CPU.
- **SC (Store-Conditional):** stores to the reserved address. **Succeeds** (returns 1) only if no other CPU has written the address since the LL; **fails** (returns 0) otherwise, and we retry.

Usage pattern (the same shape as the CAS retry loop):

```
loop:
    value = LL(addr);              // load and reserve
    new_value = compute(value);
    if (SC(addr, new_value))       // store only if still reserved
        break;                     // success
    // else: retry
```

A.3.2 · Hardware Atomic Primitives Across Architectures

Architecture	Test-and-Set	CAS	Fetch-and-Add	LL/SC
x86-64	`xchg`	`cmpxchg`	`lock xadd`	—
ARM (AArch64)	`swp` (ARMv8.1) or `ldxr` / `stxr`	`cas` (ARMv8.1)	`ldadd` (ARMv8.1)	`ldxr` / `stxr`
RISC-V	`amoswap`	`lr` / `sc` loop (or Zacas extension)	`amoadd`	`lr` / `sc`
MIPS	via `ll` / `sc`	via `ll` / `sc`	via `ll` / `sc`	`ll` / `sc`
IBM POWER	via `lwarx` / `stwcx`	via `lwarx` / `stwcx`	via `lwarx` / `stwcx`	`lwarx` / `stwcx`

On x86-64, the `LOCK` prefix makes read-modify-write instructions atomic: `lock cmpxchg`, `lock xadd`, `lock inc`, … (`xchg` with a memory operand is implicitly locked).

📎 A.4 — Spinlocks in Depth: Traces, Cache Effects & TTAS

A.4.1 · TAS Spinlock Trace: Two Threads Competing

```
Initially: flag = 0 (FREE)

Thread 1                                Thread 2                          flag
──────────────────────────────────────────────────────────────────────────────
spin_lock(): TAS → 0  (acquired)                                          1
[T1 in CS: counter++]                   spin_lock(): TAS → 1  (denied)    1
                                        TAS → 1  (denied, spinning...)    1
spin_unlock(): flag = 0                                                   0
                                        TAS → 0  (acquired)               1
                                        [T2 in CS: counter++]             1
                                        spin_unlock(): flag = 0           0
```

Each failed TAS rewrites a 1 over a 1: harmless, but it is a *write* on every iteration.

A.4.2 · The Spin-Wait Problem: Cache Line Thrashing

A subtle cost of spinning under contention is **cache-line thrashing**.

Scenario: 16 threads spin on `TAS(flag, 1)`. Every TAS is a *write*, so each core must obtain the cache line in **exclusive** state:

- Core i gets the line and **invalidates** all other copies.
- Core j misses, re-fetches the line, and invalidates the others in turn.
- All 16 cores do this continuously.

> ***Effect:*** *the line holding the lock bounces between cores. Coherence traffic explodes, and it slows even the lock holder, who needs that line (or its neighbors) to make progress.*

The ticket lock avoids the writes but not the invalidations: every `turn++` invalidates all spinners' copies.

A.4.3 · Test-and-Test-and-Set (TTAS)

Idea: spin with plain *reads* (cache hits, no traffic) and attempt the expensive TAS only when the lock looks free.

```
void ttas_lock(spinlock_t *l) {
    for (;;) {
        while (atomic_load_explicit(&l->flag, memory_order_relaxed) == 1)
            ;                                   // read-only spin: stays in local cache
        if (TestAndSet(&l->flag, 1) == 0)       // looks free: try the atomic exchange
            return;                             // got it
    }                                           // lost the race: back to read-only spin
}
```

- While the lock is held, all spinners read **their own cached copy**: no bus traffic.
- Only when the holder writes 0 is the line invalidated; then spinners race with TAS, and one wins.

A.4.4 · Ticket Lock Trace: T1, T2, T3

Step	Event	`my_ticket`	`ticket`	`turn`	Holder
0	initial state	—	0	0	—
1	T1 calls `lock()`	0	1	0	**T1** $(0 = \text{turn})$
2	T2 calls `lock()`	1	2	0	T1 (T2 spins)
3	T3 calls `lock()`	2	3	0	T1 (T2, T3 spin)
4	T1 calls `unlock()`	—	3	1	**T2** $(1 = \text{turn})$
5	T2 calls `unlock()`	—	3	2	**T3**
6	T3 calls `unlock()`	—	3	3	— (free)

Service order = arrival order: **FIFO**.

A.4.5 · Evaluating the Ticket Lock

Correctness ✅ — Each thread gets a **unique** ticket (FAA is atomic) and only the thread with `my_ticket == turn` proceeds, so at most one thread is in the CS.

Fairness ✅ — Threads are served in the exact order of their `lock()` calls: nobody can overtake. With n threads, a thread waits for at most $n - 1$ acquisitions by others.

Performance ❌ **(still a spinlock)** — All waiters spin on `turn`. Worse: if the next-in-line thread is *preempted*, everyone behind it waits for it.

> *We fixed fairness, not the spinning. Every lock so far makes waiters **burn CPU**. Time to let them sleep (§ 7).*

📎 A.5 — Lock-Free Structures & The ABA Problem

A.5.1 · CAS-Based Lock-Free Stack: Push & Pop

```c
typedef struct Node { int val; struct Node *next; } Node;
typedef struct { _Atomic(Node *) top; } Stack;

void push(Stack *s, Node *n) {
    Node *old;
    do {
        old = atomic_load(&s->top);
        n->next = old;                                  // link to the current top
    } while (!atomic_compare_exchange_strong(&s->top, &old, n));  // top still == old?
}

Node *pop(Stack *s) {
    Node *old, *next;
    do {
        old = atomic_load(&s->top);
        if (old == NULL) return NULL;
        next = old->next;                               // read BEFORE the CAS
    } while (!atomic_compare_exchange_strong(&s->top, &old, next));
    return old;
}
```

If two threads race, one CAS succeeds and the other retries. No lock is needed.

A.5.2 · The ABA Problem: A Fundamental CAS Hazard

CAS asks "is the value still what I read?", but equal does not mean *unchanged*.

```
Stack: top -> A -> B -> C
T1 pop():  reads old = A, next = A->next = B.      [T1 preempted before its CAS]

T2: pop A    -> top -> B -> C
T2: pop B    -> top -> C        (B is freed!)
T2: push A   -> top -> A -> C   (A is reused; now A->next = C)

T1 resumes:  CAS(top, expected = A, new = B)
  top == A  ✓  (the address matches) -> CAS SUCCEEDS, top = B
  B was freed: top points to freed memory (use-after-free), and C is lost.
```

The value went $A \to B \to C \to A$ (equivalently: A was removed and later re-inserted), so CAS cannot tell the "new A" from the "old A".

A.5.3 · Solving ABA: Version Tags

Attach a **version counter** to the pointer and make CAS compare both:

```
typedef struct { uintptr_t ptr; uintptr_t version; } TaggedPtr;

// Instead of  CAS(top, A, B)  use  CAS((top, version), (A, v), (B, v+1)).
// Every successful update increments the version.

// T1 reads top = (A, 3) and prepares CAS((top,ver), (A,3), (B,4)).
// T2: pop A  -> top = (B, 4);  pop B -> top = (C, 5);  push A -> top = (A, 6).
// T1's CAS: current (A, 6) != expected (A, 3)  -> FAILS. Bug prevented. ✅

// x86-64: CMPXCHG16B (128-bit CAS) for the (ptr, version) pair,
// or encode the version in unused pointer bits.
```

Alternative: **hazard pointers** (or epochs) never reuse a node while another thread may still read it.

📎 A.6 — Advanced OS Lock Support & Scalability

A.6.1 · Problem with Yield: Context Switch Overhead

Yielding wastes less CPU than spinning, but costs context switches:

```
Scenario: 100 threads wait for a lock held by T0 for 10 ms.

With a yield lock:
  T1 tries TAS -> fails -> yields   (context switch #1)
  T2 tries TAS -> fails -> yields   (context switch #2)
  ...
  T99 tries TAS -> fails -> yields  (context switch #99)
  ... T0 releases; the next waiter that runs finds the lock free.

  99 context switches x ~5 us = ~495 us of pure overhead per round,
  and the threads keep cycling while T0 is still inside the CS.
  For a 10 us CS, the same ~495 us of switching dwarfs the useful work.

Yield also gives no fairness guarantee (no FIFO order).
```

A.6.2 · The Lost Wake-Up Race (why `setpark()`)

In `mutex_lock()` the guard must be released **before** `park()`, otherwise the holder could never take the guard to unlock. That opens a window:

```
T2 (waiter)                              T1 (holder)
-------------------------------------------------------------------
guard_acquire; queue_add(T2);
guard_release;                           mutex_unlock(): guard_acquire
   <-- window -->                        queue not empty: unpark(T2)
                                            (T2 is not parked yet: wake-up LOST)
park();  sleeps forever                  (flag stays 1; nobody will wake T2 again)
```

Fix: T2 calls `setpark()` *before* releasing the guard. An `unpark()` that arrives early leaves a **pending wake-up**, and the following `park()` returns immediately.

`futex` solves the same race differently: `FUTEX_WAIT(addr, expected)` re-checks `*addr` atomically inside the kernel.

A.6.3 · Linux `futex` Fast Path and Slow Path Internals

Simplified `pthread_mutex_lock()` on a three-state word (`0` free, `1` held, `2` held with waiters). Here `cmpxchg` **returns the old value**, like the x86 instruction.

```
void mutex_lock(int *flag) {
    int c;
    if ((c = cmpxchg(flag, 0, 1)) == 0)     // fast path: single CAS, no syscall
        return;                              // was FREE: acquired
    do {                                     // slow path: lock was held
        if (c == 2 || cmpxchg(flag, 1, 2) != 0)      // mark "held with waiters"
            futex(flag, FUTEX_WAIT, 2, NULL, NULL, 0);   // sleep if still 2
    } while ((c = cmpxchg(flag, 0, 2)) != 0);  // woken: retry, keep state 2
}

void mutex_unlock(int *flag) {
    if (atomic_dec(flag) != 1) {             // atomic_dec returns the OLD value
        *flag = 0;                           // old value was 2: waiters exist
        futex(flag, FUTEX_WAKE, 1, NULL, NULL, 0);   // wake one waiter
    }                                        // old value was 1: now 0, no syscall
}
```

A.6.4 · Scalability: Amdahl's Law Applied to Synchronization

Even with a perfect lock, the serial portion limits the parallel speedup:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

where p = fraction of work that can run in parallel, n = number of threads.

```
If 10% of the program is serialized by locks (p = 0.9):
  2 threads:  S = 1 / (0.1 + 0.45)   = 1.82x
  4 threads:  S = 1 / (0.1 + 0.225)  = 3.08x
  8 threads:  S = 1 / (0.1 + 0.1125) = 4.71x
  infinite:   S = 1 / 0.1            = 10x   (the maximum ever)

If 50% is serialized (p = 0.5): the maximum is 2x.
```

Reducing the locked fraction matters: fine-grained locking, lock-free structures, read-write locks.

Quick exercise: 25% serialized code ($p = 0.75$) $\Rightarrow S(\infty) = 1/0.25 = 4\times$.

📎 A.7 — Exercises

A.7.1 · Exercise 1: Peterson's Algorithm on Weak Memory Models

Question: two threads run Peterson's algorithm on a CPU with a store buffer. Show how both can enter the CS, and fix it.

```
// Thread 0:                // Thread 1:
flag[0] = 1;                flag[1] = 1;
turn = 1;                   turn = 0;
// read flag[1]             // read flag[0]
```

```
Solution (x86 allows store-load reordering):
  T0 puts flag[0]=1 and turn=1 in its store buffer (not yet in the cache).
  T0 reads flag[1]: not in its own buffer -> reads cache: 0 (T1's write not visible yet).
  T1 reads flag[0]: 0 (T0's write is still in T0's store buffer).
  Both see the other's flag as 0 -> both enter the CS!
Fix: insert mfence between the writes and the reads (hardware-dependent, no longer pure software).
```

A.7.2 · Exercise 2: Tracing and Fixing the ABA Hazard

Question: lock-free stack, initially `top -> A -> B -> NULL`. Trace the ABA failure of `pop()` and show how version tags prevent it.

```
T1 pop():  reads top = A, A->next = B; will run CAS(top, A, B).  [T1 preempted]

T2:  pop()   removes A  (top -> B).       A is freed or pooled.
T2:  pop()   removes B  (top -> NULL).    B is freed.
T2:  push(A) reuses A, sets A->next = NULL.  top -> A.

T1 resumes:  CAS(top, A, B): top == A ✓ (same address) -> sets top = B.
  B was FREED: top now points to freed memory. Use-after-free, crash or heap corruption.

Fix with version tags (start at top = (A,3)):
  T1 prepares CAS((top,v), (A,3), (B,4)).
  T2's three updates give versions 4, 5, 6: top = (A,6).
  T1's CAS: (A,3) != (A,6) -> FAILS, T1 retries with the fresh state. ✅
```