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/

Part I: Threads 101

- `spawn(fn)`: create a shared–memory thread
- Model: state machine $\rightarrow$ execution flow

Part II: Loss of determinism, execution order, and global consistency

- Humans are sequential creatures; we simplify $A \rightarrow \cdots \rightarrow B$ as $A \rightarrow B$.
- Compilers (and CPUs acting like compilers) are built around that sequential intuition.
- Multiprocessors change what "execute" means:
  - Any `load` may read a value written by another thread.
  - Even "$1 + 1$" can fail without proper synchronization.

Humans Are Born to Solve Concurrency Problems.

Shared Memory = The Physical World

- The physical world is inherently parallel.

Threads = People

- The brain performs local storage and computation.

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 code block that must not run concurrently.
- 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 one code block.
- Marked blocks are mutually exclusive: once I enter a marked block 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.

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

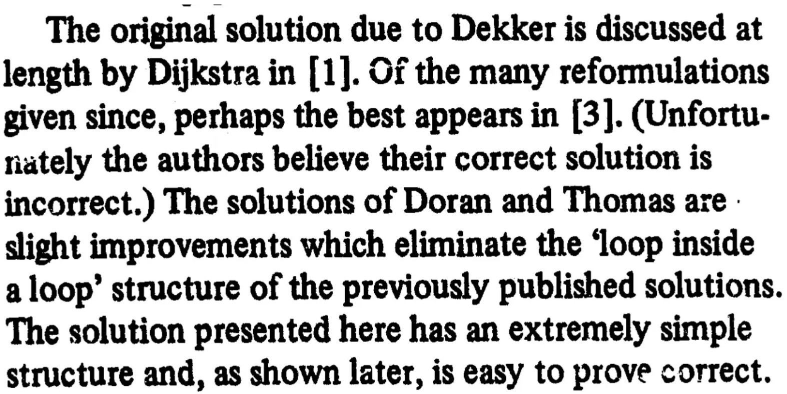
(T_n denotes the running time on n processors.)

- 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-lane road with flagger control

- Anyone can use the road, but only one car at a time.
- If both sides want to enter, the flagger decides whose **turn** it is.
- Using the road 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).

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.

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?

Idea

- Bug cause: the condition may become false by the time we write `can_go = false`.

```
void lock() {
retry:
  if (can_go == true) {
    can_go = false;    // race: another thread can change
   it here
    return;
  } else {
    goto retry;
  }
}

void unlock() {
  can_go = true;
}
```

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

Hardware: a Small “Stop the World” Operation

Atomos = indivisible primitive

- One uninterruptible read + compute + write.
- x86: `lock`ed instructions, e.g., `lock cmpxchg`.
- RISC−V: LR/SC pair and the A extension.
- ARM: `ldxr`/`stxr` pair, or `stadd` in Atomics.

```
if (can_go == True) {
    can_go = False;  // then function returns
}

// The corresponding x86 instructions:
// movl $True, %eax
// movl $False, %edx
// lock cmpxchgl %edx, (can_go)
```

Finally We Can Do 1 + 1 !

Atomic increment on x86 with a locked instruction

```
long sum = 0;

// increment sum atomically on all CPUs
asm volatile("lock incq %0" : "+m"(sum));
```

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.

Spinlock: API

API

```
typedef struct {
   ...
} lock_t;
void spin_lock(lock_t *lk);
void spin_unlock(lock_t *lk);
```

Spinlock: Implementation

```
void spin_lock(lock_t *lk) {
retry:
    if (!atomic_cmpxchg(&lk->status, True, False)) {
        goto retry;
    }
}

void spin_unlock(lock_t *lk) {
    lk->status = True;
    __sync_synchronize();
}
```

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: spin_lock(&l); sum++; spin_unlock(&l);
T2: spin_lock(&I); sum++; spin_unlock(&I);
```

Do not laugh. This will be you.

Mutexes & Futexes: The OS Helps Implementing Mutual Exclusion, Too.

Another Dimension of Performance: Scalability

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.)

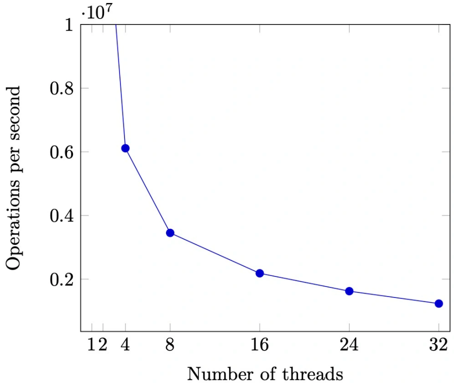


Figure: Throughput drops as the number of threads increases.

A spin lock is already a performance bug.

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)

- The application cannot preempt a spinning thread.
- If the lock holder is descheduled, other threads spin and waste 100% of a CPU.
- If the app can tell the OS, it should block instead of spin.

When Threads Cannot Solve It, Let the OS Help

Move the lock into the operating system

- `syscall(SYSCALL_acquire, &lk);`
  - Try to acquire `lk`. If it fails, switch to another thread.
- `syscall(SYSCALL_release, &lk);`
  - Release `lk`. Wake a waiting thread if any.

Let the kernel handle the hard parts

- Use short spin inside the kernel and control preemption or interrupts.
- Use spin only for tiny critical sections in kernel space.
- On success return to user space quickly.
- On failure mark the thread not runnable and schedule another one.

pthread Mutex Lock

Same API as a spin lock, and performance is good enough

```
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

Have A Try: sum-mutex

Futex: Fast Userspace muTexes

The OS wants both

- A common performance trick: optimize the fast path

Fast Path: spin once

- One atomic instruction, then enter the critical section

Slow Path: spin failed

- Invoke the syscall `futex_wait`
- Let the kernel emulate the effect of spinning
    - (not actually spinning)

Futex: Fast Userspace muTexes

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?

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.