# Lecture 15: Concurrency Control and Mutual Exclusion

## Peterson's Idea, Atomics, Mutexes, and Futexes

Xin Liu

Florida State University
xliu15@fsu.edu

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

# Concurrency: From Basics to Letting Go

**Part I: Threads 101**

- `spawn(fn)`: create a shared–memory thread (execution flow, state machine)
- `join()`: wait for threads to finish

**Part II: Letting go of determinism, execution order, and global consistency**

- Humans are sequential creatures; we simplify $A \to \cdots \to B$ as $A \to B$.
- 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.

# Don’t Panic: We Are Here to Solve Problems

**Threads = People**

- The brain performs local storage and computation.

**Shared Memory = The Physical World**

- The physical world is inherently parallel.

**Program = State Machine**

- C programs, machine instructions, and model checkers.
- We can model the physical world using state transitions.

# Shared-Memory Threading Model and Thread Libraries

# Concurrent Programming: Motivation

```
void http_server(int fd) {
  while (1) {
    ssize_t nread = read(fd, buf, 1024);
    handle_request(buf, nread);
  }
}
```

**What if the arrival time of `buf` is uncertain?**

- A burst of requests may arrive.
- The code waits for `handle_request` to finish before reading the next request.
- On a system with multiple CPUs this wastes opportunities.
- We want **shared-memory threads**.

# Solution: Add an OS API

**State-machine model of a C program**

- Initial state: `main(argc, argv, envp)`
- State transition: execute one statement (instruction)

**State-machine model of a multithreaded program**

- Add a special system call: `spawn()`
  - Add one more “state machine” with its own stack but shared global variables
- State transition: choose one state machine and execute one statement (instruction)
- Let’s do it with a model checker

# Concurrency vs. Parallelism

**Concurrency**

- “Simultaneous” execution in a logical sense.
- Can be interleaved by the OS or runtime using time slicing.
- May also be truly simultaneous on some hardware.

**Parallelism**

- Truly simultaneous execution in the strict sense.
- Requires multiple processors that share memory.
- Instructions run at the same time and can load or store to shared memory.

## A minimal thread API (`thread.h`)

- `spawn(fn)`
  - Create a thread whose entry function is `fn` and start it immediately.
  - Entry example:

```
void fn(int tid) { /* ... */ }
```

  - The parameter `tid` is numbered starting from 1.
- `join()`
  - Wait for all running threads to return.
  - `main` by default joins all threads.
  - Behavior:

```
while (num_done != num_threads) { /* ... */ }
```

# Multiprocessor Programming: As Simple As This

```
#include <thread.h>

int x = 0, y = 0;

void inc_x() { while (1) { x++; sleep(1); } }
void inc_y() { while (1) { y++; sleep(2); } }

int main() {
  spawn(inc_x);
  spawn(inc_y);
  while (1) {
    printf("\033[2J\033[H");
    printf("x = %d, y = %d", x, y);
    fflush(stdout);
  }
}
```

- This program shows that global variables are shared across threads.

# More Questions

**Do multithreaded programs really use multiple processors?**

- Concurrency can be confirmed, but is it true parallelism?

**Do threads have independent stacks?**

- If yes, what is the exact scope of a thread's stack?

**How to single-step a multithreaded program with `gdb`?**

- Use an LLM to help read "The Friendly Manual" and locate the right commands.
- Many system tasks used to be blocked by low-level tool know-how. With LLM help, these barriers are lower.

# Start with a Simple Problem

**1 + 1 should be easy, right?**

```
long sum = 0;

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

**What we want from an API**

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

# Stop the World

Could hardware give us a “The World” instruction that freezes all other threads?

# Stop the World (cont'd)

```
long sum = 0;

void T_sum() {
  stop_the_world();
  // "The World" state: all other threads are frozen
  sum++;
  resume_the_world();
}
```

**Seems like overkill**

- We only need to declare a small region that must not run concurrently.
- Unrelated code should still run in parallel.

# Mutual Exclusion: Blocking Concurrency

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

**Human view**

- Use `lock`/`unlock` to mark a code region.
- All marked regions are mutually exclusive.
- Once I enter a marked region others cannot enter.

**State-machine view**

- Execution of the marked region behaves like a single state transition.
- This holds under the lock correctness conditions.

**Pessimistic view: Amdahl's Law**

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

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

**Optimistic view: Gustafson's Law (refined form)**

- Parallel computing remains achievable by scaling the workload:

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

- $T_n$ denotes runtime on $n$ processors.

# Much of Real Life Is Parallelizable

**Classical physics: locality principle**

- An object's influence on its neighbors takes time to propagate.
- Even if not strictly true in all cases, it is a very good approximation.
- *Consequence:* simulations of the physical world can be massively parallel, so $T_\infty \ll T_1$.

**Examples of embarrassingly parallel tasks**

- Library vs. distributed data storage.
- Brain vs. deep neural networks.
- Search for NP-hard problems.

# Deterministic State Transitions

# Peterson's Algorithm: Where the Road Went Wrong

**Trying to implement mutual exclusion with only loads/stores**

- We design computer systems.
- We can change the rules to make mutual exclusion easier.
- This differs from natural science.
  - Many of the "game rules" are ours to define.

**When software is hard, ask hardware for help**

- Could we have a "stop the world" instruction?

# 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`.

**When applicable**

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

# Exclusion?

**Idea**

- Start from a wrong program and ask hardware to do what we intend.
- 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;
}
```

# 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.

**Turn the racy code into an atomic step**

```
// try to set can_go from true to false atomically
bool lock() {
  bool expected = true;
  return atomic_compare_exchange_strong(&can_go, &
    expected, false);
}

void unlock() {
  atomic_store(&can_go, true);
}
```

**Example on x86 (sketch)**

```
// EAX = expected (true), [can_go] is the memory location
```

# Finally We Can Do 1 + 1

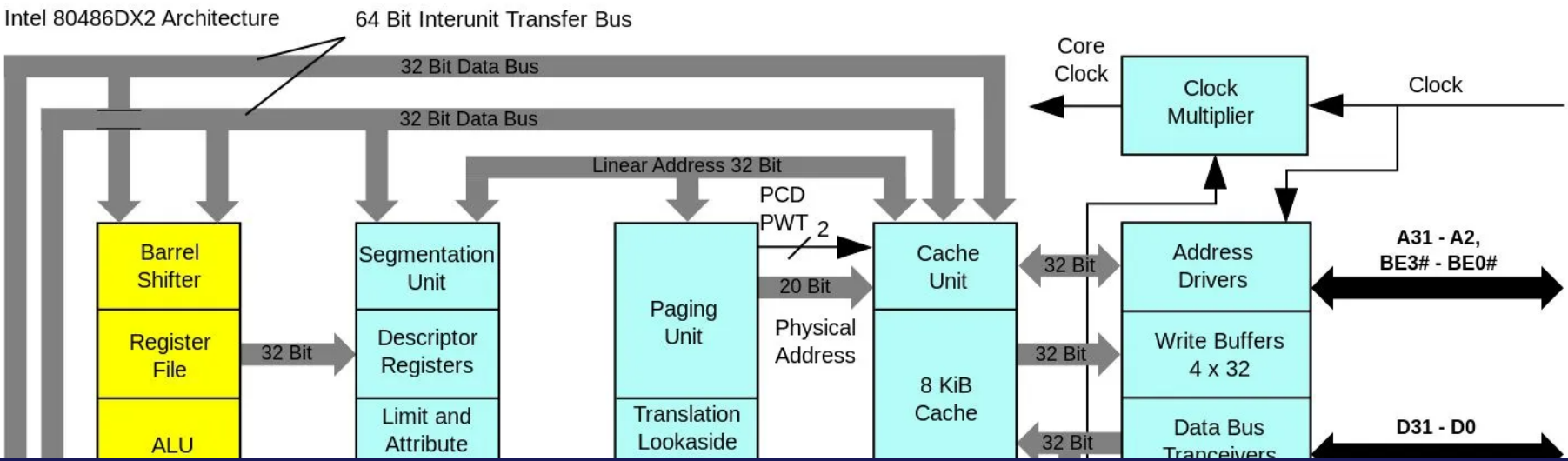
## Atomic increment on x86 with a locked instruction

```
long sum = 0;

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

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

Optional diagram



# Spin Lock: API and Implementation

## API

```c
#include <stdatomic.h>

typedef struct {
  atomic_bool locked;    // false = free, true = held
} lock_t;

void spin_lock(lock_t *lk);
void spin_unlock(lock_t *lk);
```

## Implementation

```c
void spin_lock(lock_t *lk) {
  bool expected = false;
  // Try to set locked from false -> true
  while (!atomic_compare_exchange_weak_explicit(
              &lk->locked, &expected, true,
              memory_order_acquire,      // on success
              memory_order_relaxed)) {   // on failure
    expected = false;                    // reset
    expectation
```

# 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**

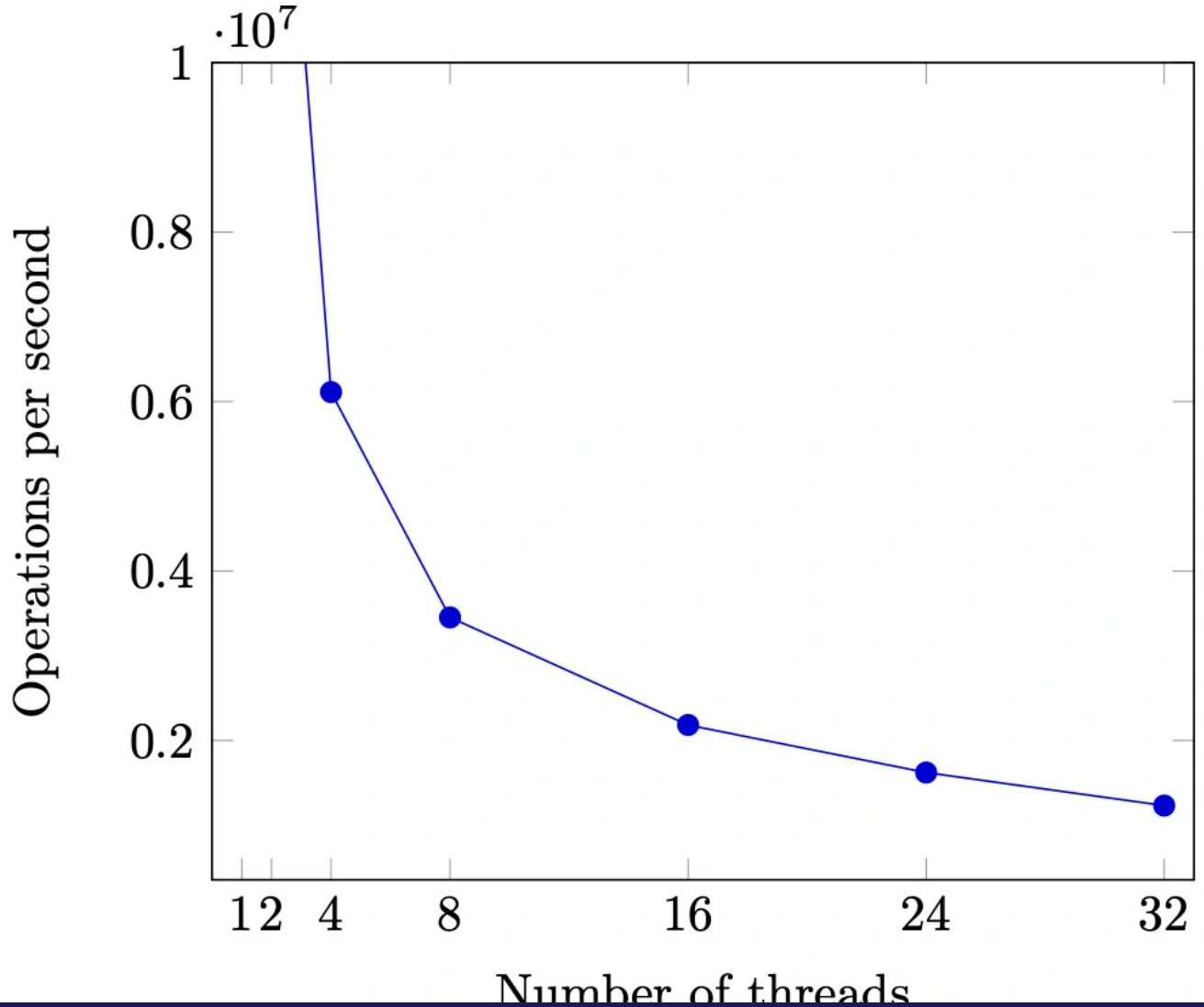
```
T1: spin_lock(&l); sum++; spin_unlock(&l);
T2: spin_lock(&l); sum++; spin_unlock(&l);
```

**Do not laugh.** This will be you unless the API is safer.

# Mutex

# 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.)*



# 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.

**Implications**

- Prefer blocking locks (mutex/futex) or adaptive spin then park.
- Use backoff to reduce contention.

# 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
- Sounds almost too good

# 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

# Concurrency is hard

- Concurrency is hard. A simple way to control complexity is to fall back to non-concurrent execution.
- We can use `lock`/`unlock` in threads to enforce mutual exclusion.
- Code protected by the same lock loses concurrency, although the exact order is still uncontrolled.
- Implementing mutual exclusion is challenging. Modern systems use thread-level atomic operations, kernel interrupt control, and primitives such as atomics and spinning.
- If enough of the program can run in parallel, serializing a small part will not hurt performance much.