

# **CSE 451: Operating Systems**

## **Winter 2026**

### **Module 7**

### **Synchronization**

**Gary Kimura**

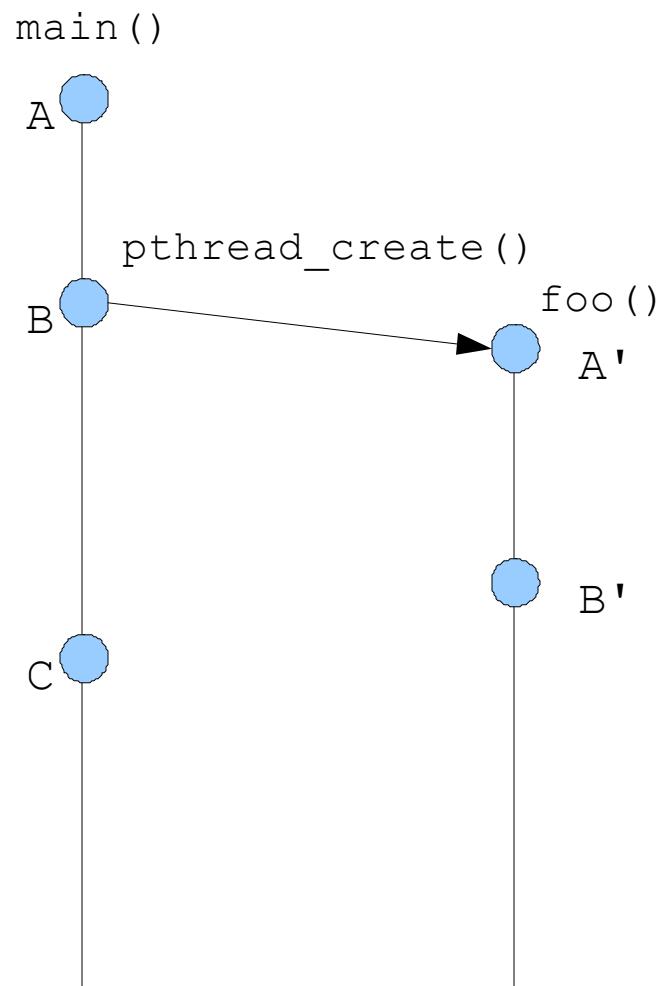
# What's to come

- **Synchronization**
  - Monday, define and provide motivation for synchronization, and look at fundamental locking primitives
  - Wednesday, advanced synchronization strategies
  - Friday, Pipes and locking in Windows
  - Possibly spill over into Monday of next week
- **Threads (kernel and user level)**

# Temporal relations

- Instructions executed by a single thread are totally ordered
  - $A < B < C < \dots$
- Absent **synchronization**, instructions executed by distinct threads must be considered unordered / simultaneous
  - Not  $X < X'$ , and not  $X' < X$

# Example



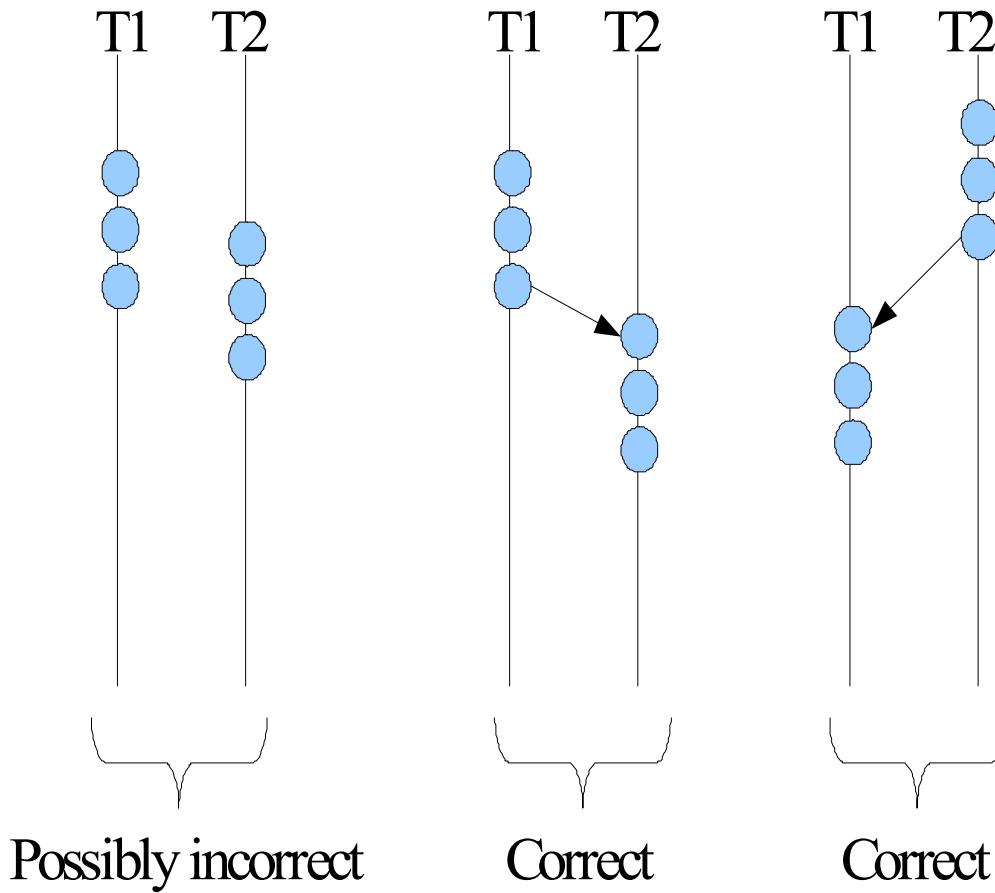
- $A < B < C$
- $A' < B'$
- $A < A'$
- $C == A'$
- $C == B'$

# Critical Sections / Mutual Exclusion

- Sequences of instructions that may get incorrect results if executed simultaneously are called **critical sections**
- (We also use the term **race condition** to refer to a situation in which the results depend on timing)
- **Mutual exclusion** means “not simultaneous”
  - $A < B$  or  $B < A$
  - We don’t care which
- Forcing mutual exclusion between two critical section executions is sufficient to ensure correct execution – guarantees ordering
- One way to guarantee mutually exclusive execution is using **locks**

# Critical sections

→ *is the "happens-before" relation*

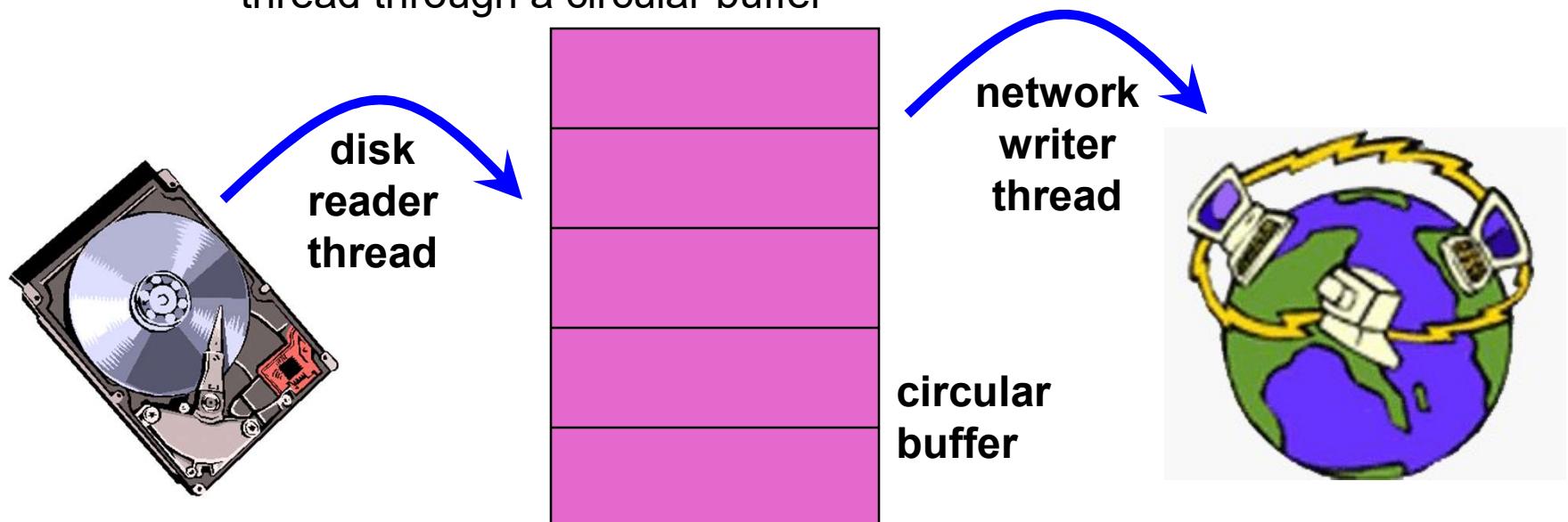


# When do critical sections arise?

- One common pattern:
  - read-modify-write of
  - a shared value (variable)
  - in code that can be executed concurrently
    - (Note: There may be only one copy of the code (e.g., a procedure), but it can be executed by more than one thread at a time)
- Shared variable:
  - Globals and heap-allocated variables
  - NOT local variables (which are on the stack)
    - (Note: Never give a reference to a stack-allocated (local) variable to another thread, unless you're superhumanly careful ...)

# Example: buffer management

- Threads cooperate in multithreaded programs
  - to **share** resources, access shared data structures
    - e.g., threads accessing a memory cache in a web server
  - also, to **coordinate** their execution
    - e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer



# Example: shared bank account

- Suppose we have to implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {  
    int balance = get_balance(account);      // read  
    balance -= amount;                      // modify  
    put_balance(account, balance);          // write  
    spit out cash;  
}
```

- Now suppose that you and your partner share a bank account with a balance of \$100.00
  - what happens if you both go to separate ATM machines, and simultaneously withdraw \$10.00 from the account?

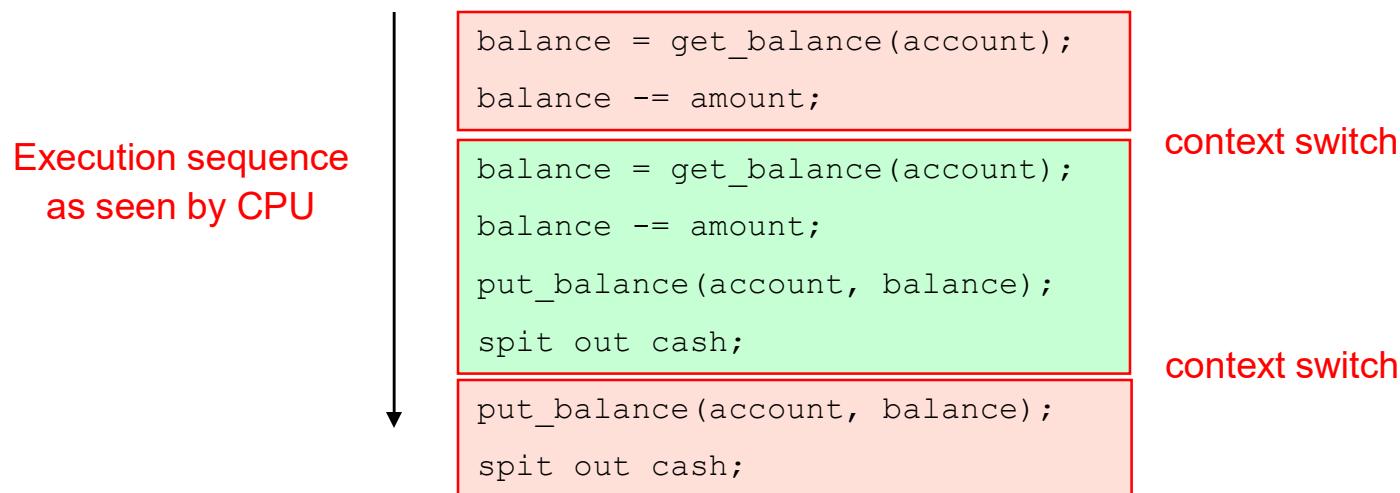
- Assume the bank's application is multi-threaded
- A random thread is assigned a transaction when that transaction is submitted

```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```

```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```

# Interleaved schedules

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:



- What's the account balance after this sequence?
  - who's happy, the bank or you?
- How often is this sequence likely to occur?

# Other Execution Orders

- Which interleavings are ok? Which are not?

```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```

```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```

# How About Now?

```
int xfer(from, to, amt) {  
    withdraw( from, amt );  
    deposit( to, amt );  
}
```

```
int xfer(from, to, amt) {  
    withdraw( from, amt );  
    deposit( to, amt );  
}
```

- **Morals:**
  - Interleavings are hard to reason about
    - We make lots of mistakes
    - Control-flow analysis is hard for tools to get right
  - Identifying critical sections and ensuring mutually exclusive access is ... “easier”

# Another example

```
i++;
```

```
i++;
```

# Correct critical section requirements

- Correct critical sections have the following requirements
  - mutual exclusion
    - at most one thread is in the critical section
  - progress
    - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
  - bounded waiting (no starvation)
    - if thread T is waiting on the critical section, then T will eventually enter the critical section
      - assumes threads eventually leave critical sections
  - performance
    - the overhead of entering and exiting the critical section is small with respect to the work being done within it

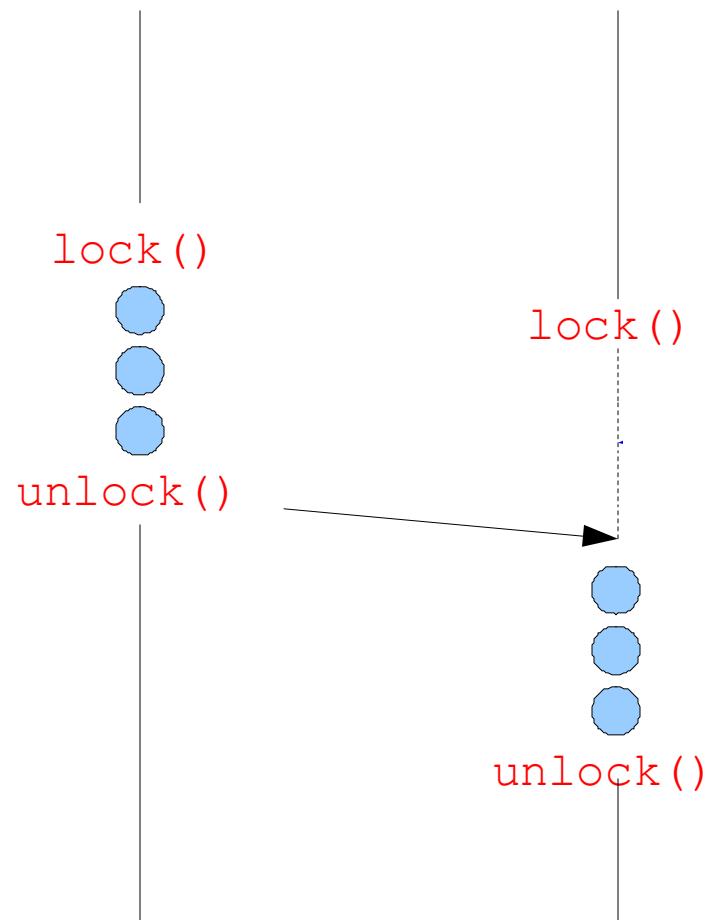
# Mechanisms for building critical sections

- Spinlocks
  - primitive, minimal semantics; used to build others
- Semaphores (and non-spinning locks)
  - basic, easy to get the hang of, somewhat hard to program with
- Monitors
  - higher level, requires language support, implicit operations
  - easier to program with; Java “`synchronized()`” as an example
- Messages
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems

# Locks

- A lock is a memory object with two operations:
  - `acquire()`: obtain the right to enter the critical section
  - `release()`: give up the right to be in the critical section
- `acquire()` prevents progress of the thread until the lock can be acquired
- (Note: terminology varies: `acquire/release`, `lock/unlock`)

# Locks: Example



# Acquire/Release

- Threads pair up calls to `acquire()` and `release()`
  - between `acquire()` and `release()`, the thread **holds** the lock
  - `acquire()` does not return until the caller “owns” (holds) the lock
    - at most one thread can hold a lock at a time
  - What happens if the calls aren’t paired (I acquire, but neglect to release)?
  - What happens if the two threads acquire different locks (I think that access to a particular shared data structure is mediated by lock A, and you think it’s mediated by lock B)?
    - (granularity of locking)

# Using locks

```
int withdraw(account, amount) {  
    acquire(lock);  
    balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    release(lock);  
    spit out cash;  
}
```

} critical section

```
acquire(lock)  
balance = get_balance(account);  
balance -= amount;
```

```
acquire(lock)  
put_balance(account, balance);  
release(lock);
```

```
balance = get_balance(account);  
balance -= amount;  
put_balance(account, balance);  
release(lock);  
spit out cash;
```

```
spit out cash;
```

- What happens when green tries to acquire the lock?

# Roadmap ...

- Where we are eventually going:
  - The OS and/or the user-level thread package will provide some sort of efficient primitive for user programs to utilize in achieving mutual exclusion (for example, *locks* or *semaphores*, used with *condition variables*)
  - There may be higher-level constructs provided by a programming language to help you get it right (for example, *monitors* – which also utilize condition variables)
- But somewhere, underneath it all, there needs to be a way to achieve “hardware” mutual exclusion (for example, *test-and-set* used to implement *spinlocks*)
  - This mechanism will not be utilized by user programs
  - But it will be utilized in implementing what user programs see

# Spinlocks

- How do we implement spinlocks? Here's one attempt:

```
struct lock_t {  
    int held = 0;  
}  
  
void acquire(lock) {  
    while (lock->held);  
    lock->held = 1;  
}  
  
void release(lock) {  
    lock->held = 0;  
}
```

the caller “busy-waits”,  
or spins, for lock to be  
released  $\Rightarrow$  hence spinlock

- Why doesn't this work?
  - where is the race condition?

# Implementing spinlocks (cont.)

- Problem is that implementation of spinlocks has critical sections, too!
  - the acquire/release must be **atomic**
    - atomic == executes as though it could not be interrupted
    - code that executes “all or nothing”
- Need help from the hardware
  - atomic instructions
    - test-and-set, compare-and-swap, ...
  - disable/enable interrupts
    - to prevent context switches

# Spinlocks redux: Hardware Test-and-Set

- CPU provides the following as **one atomic instruction**:

```
bool test_and_set(bool *flag) {  
    bool old = *flag;  
    *flag = True;  
    return old;  
}
```

- Remember, this is a single **atomic** instruction ...

# Compare and Exchange

- Compare and Exchange replaces Test and Set
- It takes three parameters
  - Pointer to a memory location
  - Comparand
  - Value
- Atomically
  - If the Comparand is equal to what is stored in the memory location then replace it with the new Value and return what was previously stored in the memory location

```
int Cmpxchg(int *Ptr, int Comparand, int NewValue) {  
    int OldValue = *Ptr;  
    if (OldValue == Comparand) *Ptr = NewValue;  
    return OldValue;  
}
```

- A gotcha with multiprocessor systems is the need to restrict access to the memory location while doing the operation.

# Implementing spinlocks using CMPXCHG

- So, to fix our broken spinlocks:

```
struct lock {
    int held = 0;
}
void acquire(lock) {
    while(cmpxchg(&lock->held, 0, MyID));
}
void release(lock) {
    lock->held = 0;
}
```

- **mutual exclusion?** (at most one thread in the critical section)
- **progress?** (T outside cannot prevent S from entering)
- **bounded waiting?** (waiting T will eventually enter)
- **performance?** (low overhead (modulo the spinning part ...))

# Reminder of use ...

```
int withdraw(account, amount) {  
    acquire(lock);  
    balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    release(lock);  
    spit out cash;  
}
```

} critical section {

```
acquire(lock)  
balance = get_balance(account);  
balance -= amount;
```

```
acquire(lock)  
put_balance(account, balance);  
release(lock);
```

```
balance = get_balance(account);  
balance -= amount;  
put_balance(account, balance);  
release(lock);  
spit out cash;
```

```
spit out cash;
```

- How does a thread blocked on an “acquire” (that is, stuck in a test-and-set loop) yield the CPU?
  - calls `yield()` (*spin-then-block*)
  - there’s an involuntary context switch (e.g., timer interrupt)

# Problems with spinlocks

- Spinlocks work, but are wasteful!
  - if a thread is spinning on a lock, the thread holding the lock cannot make progress
    - You'll spin for a scheduling quantum
- Therefore, only want spinlocks as primitives to build higher-level synchronization constructs
  - Why is this okay?
- We'll see later how to build blocking locks
  - But blocking locks have more overhead so it is sometimes cheaper to spin

# Another approach: Disabling interrupts

```
struct lock {  
}  
void acquire(lock) {  
    cli();    // disable interrupts  
}  
void release(lock) {  
    sti();    // reenable interrupts  
}
```

# Problems with disabling interrupts

- Only available to the kernel
  - Can't allow user-level to disable interrupts!
- Insufficient on a multiprocessor
  - Each processor has its own interrupt mechanism
- “Long” periods with interrupts disabled can wreak havoc with devices
- Just as with spinlocks, you only want to use disabling of interrupts to build higher-level synchronization constructs

# Race conditions

- Informally, we say a program has a **race condition** (aka “data race”) if the result of an executing depends on timing
  - i.e., is non-deterministic
- Typical symptoms
  - I run it on the same data, and sometimes it prints 0 and sometimes it prints 4
  - I run it on the same data, and sometimes it prints 0 and sometimes it crashes
- Do we want A before B, or B before A?
  - It is synchronized but the results may not be what we want
- So locks in general do not address race conditions

# Quick Recap

- Most basic locking primitives are **acquire()** and **release()**
- Most common Hardware support for doing locks
  - Fiddle with **interrupts**, Only works for the kernel and not multiprocessors (so pretty irrelevant)
  - Storing something in **memory**, The idea is that the last writer wins.
- **Spinlocks**
  - The most rudimentary building block used to implement higher level locks
  - Easy to implement but wasteful if held too long. Why? Because other threads will end up spinning instead of sleeping for the lock
- Now onto higher level synchronization constructs

# Summary

- Synchronization introduces temporal ordering
- Adding synchronization can eliminate races
- Synchronization can be provided by locks, semaphores, monitors, messages ...
- Spinlocks are the lowest-level mechanism
  - primitive in terms of semantics – error-prone
  - implemented by spin-waiting (crude) or by disabling interrupts (also crude, and can only be done in the kernel)
- In our next exciting episode ...
  - semaphores are a slightly higher level abstraction
    - Importantly, they are implemented by blocking, not spinning
    - Locks can also be implemented in this way
  - monitors are significantly higher level
    - utilize programming language support to reduce errors