

# **CSE 120**

# **Principles of Operating Systems**

**Fall 2025**

**Lecture 5: Synchronization**

Geoffrey M. Voelker

# Administrivia

---

- Make sure you accept your group github invite!
- Homework 2 out

# Synchronization

---

- Threads cooperate in multithreaded programs
  - ♦ To share resources, access shared data structures
    - » Threads accessing a memory cache in a Web server
  - ♦ To coordinate their execution
    - » One thread executes relative to another (recall ping-pong)
- For correctness, we need to control this cooperation
  - ♦ Threads **interleave executions arbitrarily** and at **different rates**
  - ♦ Scheduling is not under program control
- We control cooperation using **synchronization**
  - ♦ Synchronization enables us to restrict the possible interleavings of thread executions
- Discuss in terms of threads, also applies to processes

# Shared Resources

---

We initially focus on coordinating access to shared resources

- **Basic problem**
  - ◆ If two concurrent threads (processes) are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior
- Over the next few lectures, we will look at
  - ◆ **Mechanisms to control access to shared resources**
    - » Locks, mutexes, semaphores, monitors, condition variables, etc.
  - ◆ **Patterns for coordinating accesses to shared resources**
    - » Bounded buffer, producer-consumer, etc.

# Classic Example

---

- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {  
    int balance = get_balance(account);  
    balance = balance – amount;  
    put_balance(account, balance);  
    return balance;  
}
```

- Now suppose that you and your significant other share a bank account with a balance of \$1000
- Then you each go to separate ATM machines and simultaneously withdraw \$100 from the account

# Example Continued

---

- We'll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run on the same bank server:

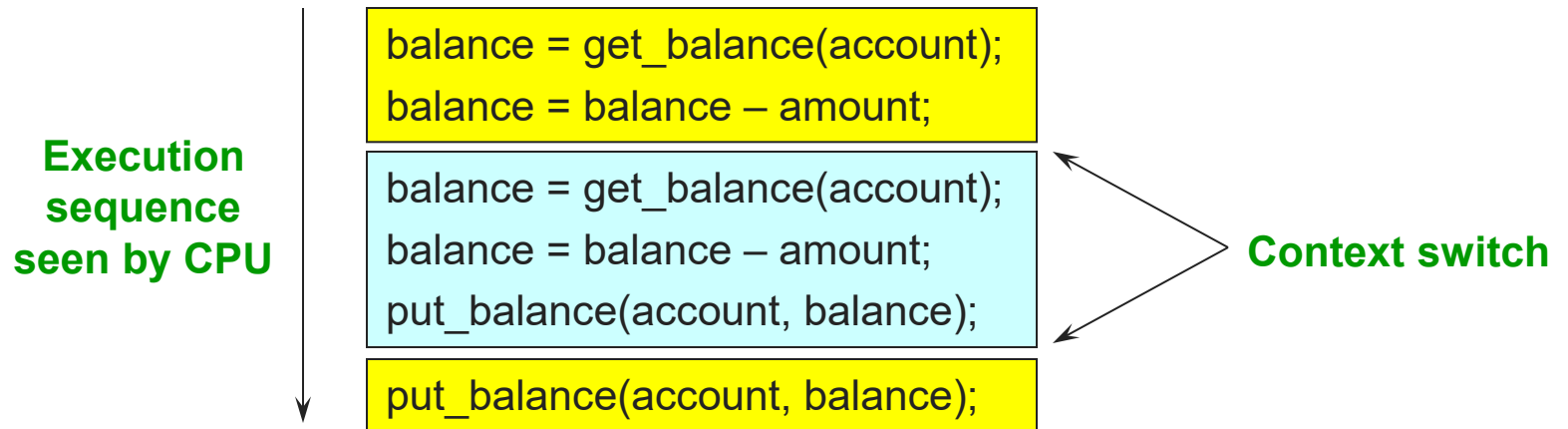
```
withdraw (account, amount) {  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    return balance;  
}
```

```
withdraw (account, amount) {  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    return balance;  
}
```

- What's the problem with this implementation?
  - ◆ Think about potential schedules of these two threads

# Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:



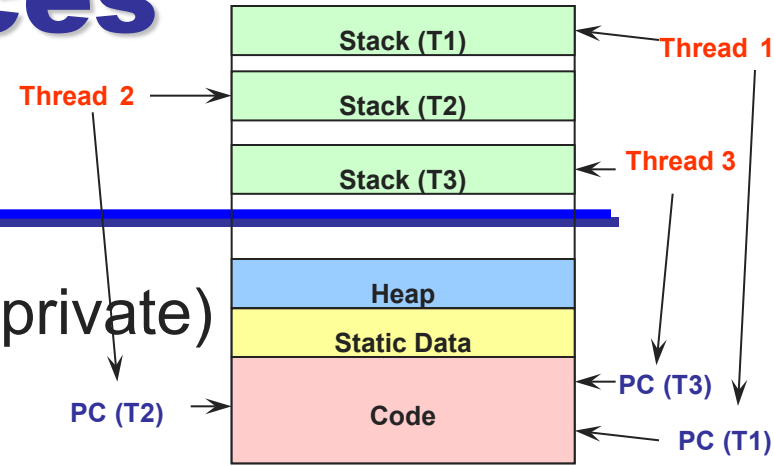
- What is the balance of the account now?
- Is the bank happy with our implementation?

# Shared Resources

---

- The problem is that two concurrent threads (or processes) accessed a **shared resource** (account) without any **synchronization**
  - ◆ Known as a **race condition** (memorize this buzzword)
- We need mechanisms to control access to these shared resources in the face of concurrency
  - ◆ So we can reason about how the program will operate
- Our example was updating a shared bank account
- Also necessary for synchronizing access to **any shared data structure**
  - ◆ Buffers, queues, lists, hash tables, counters, etc.

# When Are Resources Shared?



- Local variables are **not shared** (private)
  - ◆ Refer to data on the stack
  - ◆ Each thread has its own stack
  - ◆ **Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2**
- Global variables and static objects are **shared**
  - ◆ Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects are **shared**
  - ◆ Allocated from heap with malloc/free or new/delete

# How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are instructions (e.g., reads and writes of words)
  - ◆ (Some early architectures didn't even give you that)
- We'll assume that a **context switch can occur at any time**
  - ◆ Examples may show code
  - ◆ But actually at instruction granularity
- We'll assume that **you can delay a thread as long as you like as long as it's not delayed forever**

```
..... get_balance(account);
balance = get_balance(account);
balance = .....
balance = balance - amount;
balance = balance - amount;
put_balance(account, balance);
put_balance(account, balance);
```

# Mutual Exclusion

---

- We want to use **mutual exclusion** to synchronize access to shared resources
  - ◆ This allows us to have larger atomic blocks
- Code that uses mutual exclusion to synchronize its execution is called a **critical section**
  - ◆ Only one thread at a time can execute in the critical section
  - ◆ All other threads are forced to wait on entry
  - ◆ When a thread leaves a critical section, another can enter
  - ◆ Example: bathrooms on airplanes
- What requirements would you place on a critical section?

# Critical Section Requirements

---

## 1) Mutual exclusion (mutex)

- ◆ If one thread is in the critical section, then no other is

## 2) Progress

- ◆ If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
- ◆ A thread in the critical section will eventually leave it

## 3) Bounded waiting (no starvation)

- ◆ If some thread T is waiting on the critical section, then T will eventually enter the critical section

## 4) Performance

- ◆ The overhead of entering and exiting the critical section is small with respect to the work being done within it

# About Requirements

---

Requirements also expressed as three properties:

- **Safety property**: nothing bad happens
  - ◆ Mutex
- **Liveness property**: something good happens
  - ◆ Progress, Bounded Waiting
- **Performance property**
  - ◆ Performance
- Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!)
  - ◆ Performance ultimately only matters if it's correct

# Mechanisms For Building Critical Sections

---

- **Atomic read/write**
  - ◆ Can provide atomicity but not practical for programming
- **Locks**
  - ◆ Primitive, minimal semantics, used to build others
- **Semaphores**
  - ◆ Basic, easy to get the hang of, but harder to program with
- **Monitors**
  - ◆ High-level, requires language support, operations implicit
- **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 an object in memory providing two operations
  - ♦ `acquire()`: to enter a critical section
  - ♦ `release()`: to leave a critical section
- Threads **pair calls** to acquire and release
  - ♦ Between `acquire/release`, the thread **holds** the lock
  - ♦ `acquire` does not return until any previous holder releases
  - ♦ **What can happen if the calls are not paired?**
- Locks can spin (a spinlock) or block

# Using Locks

```
withdraw (account, amount) {  
    acquire(lock);  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    release(lock);  
    return balance;  
}
```

**Critical  
Section**

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

```
acquire(lock);
```

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

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

- ◆ What happens when blue tries to acquire the lock?
- ◆ Why is the “return” outside the critical section? Is this ok?
- ◆ What happens when a third thread calls acquire?

# Implementing Locks (1)

---

- How do we implement locks? Here is one attempt:

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

busy-wait (spin-wait)  
for lock to be released

- This is called a **spinlock** because a thread spins waiting for the lock to be released
- Does this work?

# Implementing Locks (2)

- No. Two independent threads may both notice that a lock has been released and thereby acquire it.

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

A context switch can occur here, causing a race condition

# Implementing Locks (3)

---

- The problem is that the implementation of locks has critical sections, too
- How do we stop the recursion?
- The implementation of acquire/release must be **atomic**
  - ◆ An atomic operation is one which executes as though it could not be interrupted
  - ◆ Code that executes “all or nothing”
- How do we make them atomic?
- Need help from hardware
  - ◆ Atomic instructions (e.g., test-and-set)
  - ◆ Disable/restore interrupts (prevents context switches)

# Atomic: Test-And-Set

---

- The semantics of test-and-set are:
  - ◆ Record the old value
  - ◆ Set the value to true
  - ◆ Return the old value
- Hardware executes it atomically!

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

- When executing test-and-set on “flag”
  - ◆ What is **value of flag** afterwards if it was initially False? True?
  - ◆ What is the **return result** if flag was initially False? True?

# Using Test-And-Set

---

- Here is our lock implementation with test-and-set:

```
struct lock {  
    int held = 0;  
}  
void acquire (lock) {  
    while (test-and-set(&lock→held));  
}  
void release (lock) {  
    lock→held = 0;  
}
```

- When will the while return? What is the value of held?
- What about multiprocessors?

# Problems with Spinlocks

---

- The problem with spinlocks is that they are wasteful
  - ◆ If a thread is spinning on a lock, then the thread holding the lock cannot make progress (on a uniprocessor)
- How did the lock holder give up the CPU in the first place?
  - ◆ Voluntary: Lock holder calls yield or sleep
  - ◆ Involuntary: Timer causes context switch
- Only want to use spinlocks as primitives to build higher-level synchronization constructs

# Disabling Interrupts

---

- Another implementation of acquire/release is to disable interrupts:

```
struct lock {  
}  
void acquire (lock) {  
    disable interrupts;  
}  
void release (lock) {  
    restore interrupts;  
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?

# On Disabling Interrupts

---

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
  - ◆ This is what Nachos uses as its primitive
- In a “real” system, this is only available to the kernel
  - ◆ Why?
- **Disabling interrupts is insufficient on a multiprocessor**
  - ◆ Interrupts are only disabled on a per-core basis
  - ◆ Back to atomic instructions
- Like spinlocks, only want to disable interrupts to implement higher-level synchronization primitives
  - ◆ Don't want interrupts disabled between acquire and release

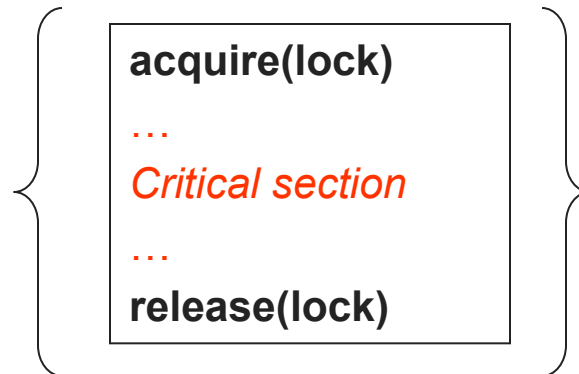
# Summarize Where We Are

---

- Goal: Use **mutual exclusion** to protect **critical sections** of code that access **shared resources**
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections (CS) can be long

## Spinlocks:

- ◆ Threads waiting to acquire lock spin in test-and-set loop
- ◆ Wastes CPU cycles
- ◆ Longer the CS, the longer the spin
- ◆ Greater the chance for lock holder to be interrupted



## Disabling Interrupts:

- ◆ Should not disable interrupts for long periods of time
- ◆ Can miss or delay important events (e.g., timer, I/O)

# Higher-Level Synchronization

---

- Spinlocks and disabling interrupts are useful only for very short and simple critical sections
  - ◆ Wasteful otherwise
  - ◆ These primitives are “primitive” – don’t do anything besides mutual exclusion
- Need higher-level synchronization primitives that:
  - ◆ **Block waiters**
  - ◆ **Leave interrupts enabled within the critical section**
- All synchronization requires atomicity
- So we’ll use our “atomic” locks as primitives to implement them

# Implementing Locks (4)

- Block waiters, interrupts enabled in critical sections
  - ◆ How Nachos implements locks (see threads/Lock.java)

```
struct lock {
    int held = 0;
    queue Q;
}
void acquire (lock) {
    Disable interrupts;
    while (lock→held) {
        put current thread on lock Q;
        sleep current thread;
    }
    lock→held = 1;
    Restore interrupts;
}
```

```
void release (lock) {
    Disable interrupts;
    if (Q) remove waiting thread;
    ready waiting thread;
    lock→held = 0;
    Restore interrupts;
}
```

```
acquire(lock)
...
Critical section
...
release(lock)
```

**Interrupts Disabled**

**Interrupts Enabled**

**Interrupts Disabled**

# Implementing Locks (5)

- Using test-and-set
  - ♦ Interrupts always enabled, can be used at user level
  - ♦ Works on multiprocessors

```
struct lock {  
    int held = 0;  
}  
void acquire (lock) {  
    while (test-and-set(&lock→held));  
}
```

```
void release (lock) {  
    lock→held = 0;  
}
```

**acquire(lock)**

...

*Critical section*

...

**release(lock)**

**Interrupts Enabled**

# Cornucopia of Locks

---

- OSes are very sensitive to overhead of locking
  - ◆ Want to minimize overhead, optimize for common case
- Many different kinds of locks have been invented
  - ◆ test-and-test-and-set (avoid cache, bus contention)
  - ◆ test-and-yield (allow another thread to run)
  - ◆ test-and-sleep (avoid spinning)
  - ◆ reader-writer locks (allow multiple readers)
    - » Variants optimized for reads as common case, many readers
  - ◆ read-copy-update (optimize for reads)
  - ◆ distributed locks (avoid cache, bus contention)
  - ◆ ...

# Next time...

---

- Read Chapters 30, 31