

# CE 440 Introduction to Operating System

## Lecture 6: Synchronization Fall 2025

Prof. Yigong Hu



Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

# Administrivia

## Fill out project group form

- [https://docs.google.com/forms/d/e/1FAIpQLScqr0QdmoruMu\\_w7-FizeQ9OYaijg9-d9Y58zOV28wivnYp5A/viewform?usp=dialog](https://docs.google.com/forms/d/e/1FAIpQLScqr0QdmoruMu_w7-FizeQ9OYaijg9-d9Y58zOV28wivnYp5A/viewform?usp=dialog)

## Lab 1 released

- Lab 1 overview session this Friday
- Read the requirement now
- Start with exercise 2.1

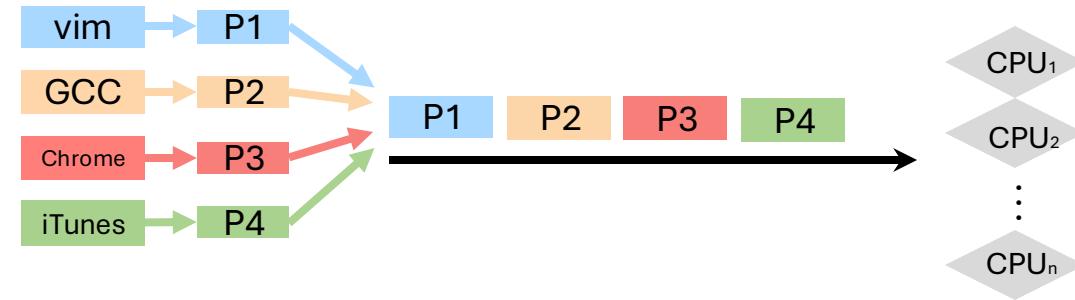
## GitHub classroom invitation link

- Used for the following lab assignments
- [https://classroom.github.com/a/MvQb\\_x8F](https://classroom.github.com/a/MvQb_x8F)

# Recap: Scheduling

## The scheduling problem:

- Have  $K$  jobs ready to run
- Have  $N \geq 1$  CPUs



## Many potential goals of scheduling algorithms

- Utilization, throughput, wait time, response time, etc.

## Various algorithms to meet these goals

- FCFS/FIFO, SJF, RR, Priority

# Recap: Single and Multithreaded Processes

## Process/Thread Separation

- The **thread** defines a sequential execution
- The **process** defines the address space and general process attributes

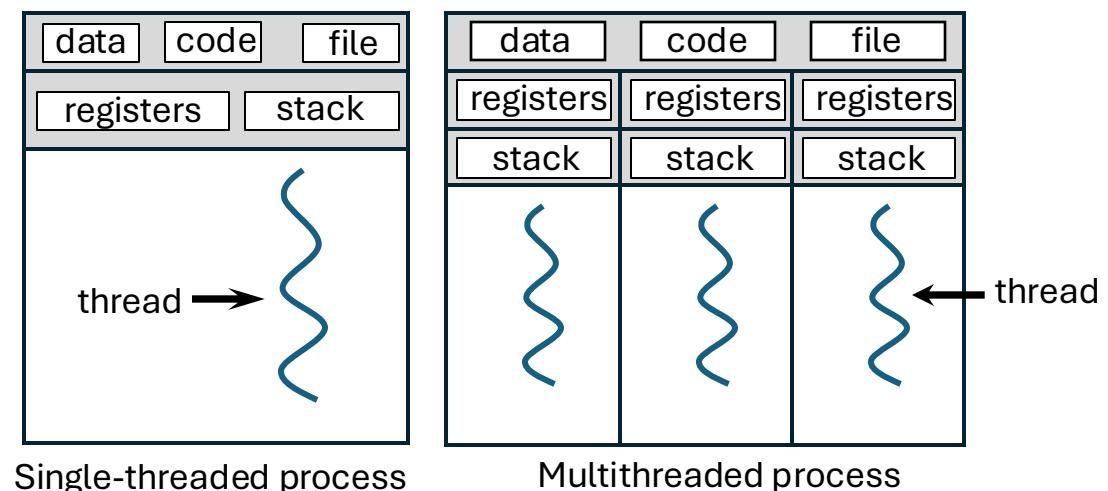
## A thread is bound to a single process

- Processes, however, can have multiple threads

## Threads become the unit of scheduling

**Now, how do we get our threads to correctly cooperate with each other?**

- Synchronization...



# Correctness with Concurrent Threads

## Threads cooperate in multithreaded programs

- To share resources, access shared data structures
- To coordinate their execution

## For correctness, we need to control this cooperation

- Thread schedule is non-deterministic (i.e., behavior changes when re-run program)
  - Scheduling is not under program control
  - Threads interleave executions arbitrarily and at different rates

# Motivated Example: Too Much Milk

**People need to coordinate:**

- Alice and Bob are roommate and they share milk
- Here is a story: they both thought they were buying one carton of milk, but they ended up with **two!**



Time	Alice	Bob
3:00	Look in Fridge. Out of milk.	
3:05	Leave for store.	
3:10	Arrive at store.	Look in fridge. Out of milk.
3:15	Buy milk.	Leave for store.
3:20	Arrive home, put milk away.	Arrive at store.
3:25		Buy milk.
3:30		Arrive home, put milk away. Oh no!

# Correctness with Concurrent Threads

## Threads cooperate in multithreaded programs

- To share resources, access shared data structures
- To coordinate their execution

## For correctness, we need to control this cooperation

- Thread schedule is non-deterministic (i.e., behavior changes when re-run program)
  - Scheduling is not under program control
  - Threads interleave executions arbitrarily and at different rates
- Multi-word operations are not atomic

# Too Much Milk... Operation?

x is a global variable initialized to 0

```
// Thread 1           // Thread 2  
  
void foo() {          void bar() {  
    x++;              x--;  
}  
}
```

**After thread 1 and thread 2 finishes, what is the value of x?**

- could be 0, 1, -1
- Why?
  - x++ and x-- are not atomic operations
  - Load x from memory
  - Modify value (add or subtract)
  - Store back to memory

# Correctness with Concurrent Threads

## Threads cooperate in multithreaded programs

- To share resources, access shared data structures
- To coordinate their execution

## For correctness, we need to control this cooperation

- Thread schedule is non-deterministic (i.e., behavior changes when re-run program)
  - Scheduling is not under program control
  - Threads interleave executions arbitrarily and at different rates
- Multi-word operations are not atomic
- Compiler/hardware instruction reordering

# One More Exercise

```
int p = 0, ready = 0;  
// Thread 1           // Thread 2  
  
p = 1000;           while (!ready);  
ready = 1;           use(p)
```

## What value of p is passed to use

- Could be 0, 1000
- Why?

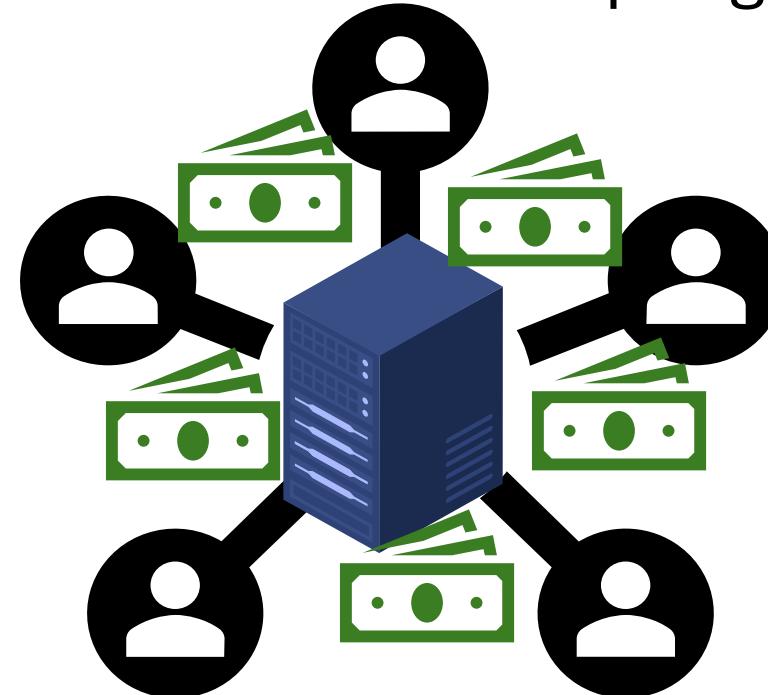
# Concurrency Is Important and Hard

## Therac-25: Radiation Therapy Machine with Unintended Overdose

- Concurrency errors caused the death of a number of patients

## ATM Bank:

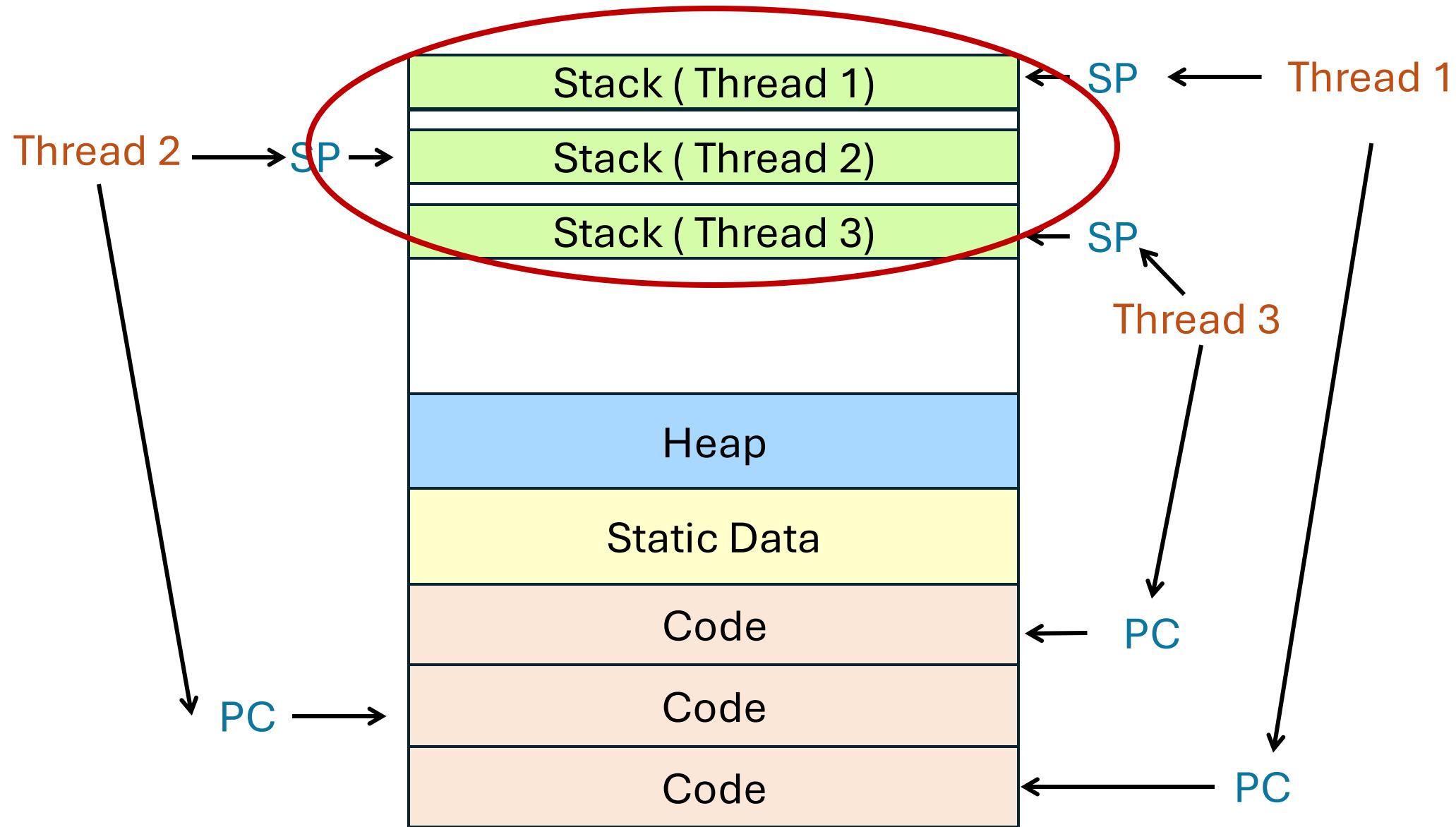
- Service a set of requests without corrupting database



# **Problem with Shared Resources**

**We focus on controlling access to shared resources**

# What Resources Are Shared?

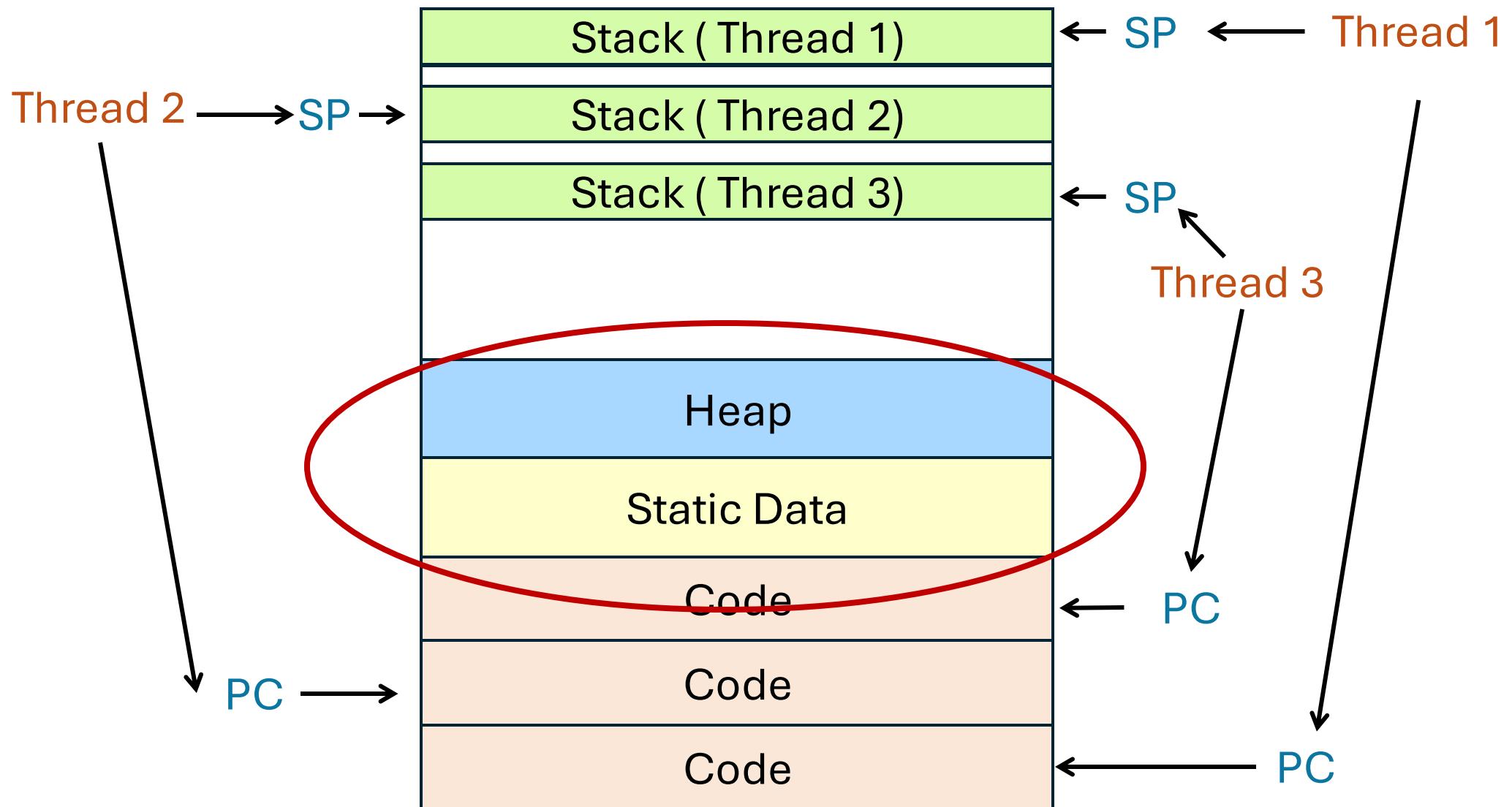


# What Resources Are Shared?

**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

# What Resources Are Shared?



# What Resources Are 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

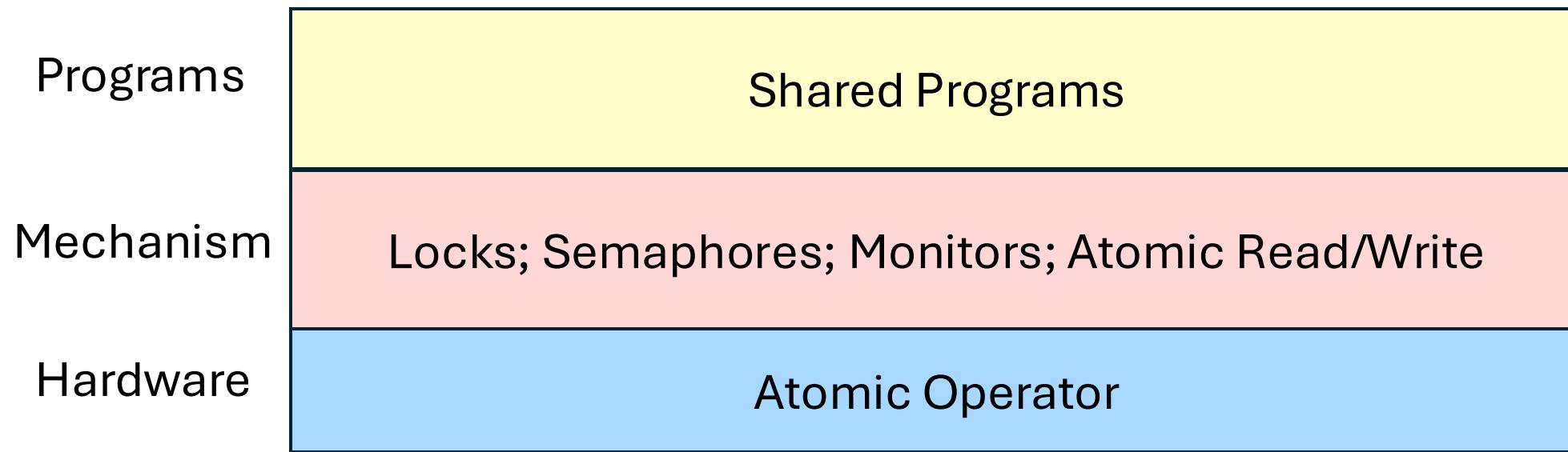
# Problem with Shared Resources

We focus on controlling 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.

# Where Are We Going with Synchronization?



**Coordination happens across all layers**

# Problem with Shared Resources

We focus on controlling 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 couple of 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
  - Reader-writer, producer-consumer, etc.

# Problem with Shared Resources

**Problem: concurrent threads accessed a **shared resource** without any **synchronization****

- Know as **a race condition**

# Race Condition Example: Bank Account

Implement a function to handle withdrawals from a bank account:

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

Suppose that you have a family account with a balance of \$10,000

Then you and your parent go to separate ATM machines and simultaneously withdraw \$1000 from the account

# Race Condition Example Continued

The bank server will create separate threads 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;  
}
```

Let's examine the schedules of these two threads together

# Interleaved Schedules

The execution of the two threads can be interleaved

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

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

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



After withdrawing \$2000 from \$10,000, balance of the account is...

- \$ 9,000



- The banker would be very unhappy about it



# How Interleaved Can It Get?

## How many possible interleaving?

- Only instructions are atomic
- A context switch can occur at any time
- OS can delay a thread for any time as long as it's not delayed forever

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

# Shared Resources

**Problem: concurrent threads accessed a **shared resource** without any **synchronization****

- Know as **a race condition**

**Although our example was updating a shared bank account, it is apply to any shared data structure**

- Buffers, queues, lists, hash tables, etc.

**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

# What do We Need for Controlling Concurrency

## Mutual Exclusion

- When one thread access shared resource, other thread can not access it

**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: sharing your bathroom with housemates

**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

# Critical Section Requirements

## 4. Performance

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

## In summary:

- **Safety property**: nothing bad happens
  - Mutex
- **Liveness property**: something good happens
  - Progress, Bounded Waiting
- **Performance requirement**
  - Performance

**Note: correctness of concurrent is guarantee by design**

# Too Much Milk: Solution #1

How about leave a note?

```
if (milk == 0) {          // if no milk
    if (note == 0) {      // if no note
        note = 1;         // leave note
        milk++;            // buy milk
        note = 0;           // remove note
    }
}
```



Does it solve the problem?

# Too Much Milk: Solution #1

## Problem with leave a note

Alice

```
if (milk == 0) {  
    if (note == 0) {  
        note = 1;  
        milk++;  
        note = 0;  
    }  
}
```

Bob

```
if (milk == 0) {  
    if (note == 0) {  
        note = 1;  
        milk++;  
        note = 0;  
    }  
}
```

# Too Much Milk: Solution #2

**Clearly the Note is not quite blocking enough**

- Let's try to fix this by placing two note: alice note and bob note

Alice

```
noteA = 1;  
if (noteB == 0) {  
    if (milk == 0) {  
        milk++;  
    }  
}  
noteA = 0;
```

Bob

```
noteB = 1;  
if (noteA == 0) {  
    if (milk == 0) {  
        milk++;  
    }  
}  
noteB = 0;
```

**Is this safe?**

- Yes
- What if Alice executes noteA = 1. At the same time, Bob executes noteB = 1?
  - I'm not getting milk, You're getting milk
  - Starvation

# Too Much Milk: Solution #3

**Monitoring note:**

Alice

```
noteA = 1;  
while (noteB == 1);  
if (noteB == 0) {  
    if (milk == 0) {  
        milk++;  
    }  
}  
noteA = 0;
```

Bob

```
noteB = 1;  
if (noteA == 0) {  
    if (milk == 0) {  
        milk++;  
    }  
}  
noteB = 0;
```

**Is this safe?**

- Yes

**Do it ensure liveness?**

# Mechanisms For Building Critical Sections

## Atomic read/write

- Can it be done?

## Locks

- Primitive, minimal semantics, used to build others

## Semaphores

- Basic, easy to get the hang of, but hard to program with

## Monitors

- High-level, requires language support, operations implicit

# Mutex with Atomic R/W: Try #1

int turn = 1;

$T_1$

```
while (true) {  
    while (turn != 1);  
    critical section  
    turn = 2;  
    outside of critical section  
}
```

$T_2$

```
while (true) {  
    while (turn != 2);  
    critical section  
    turn = 1;  
    outside of critical section  
}
```

This is called alternation

Does it satisfy the safety requirement?

- Yes

Does it satisfy the liveness requirement?

- No, T1 can go into infinite loop outside of the critical section preventing T2 from entering

# Mutex with Atomic R/W: Peterson's Algorithm

```
int turn = 1;  
bool try1 = false, try2 = false;
```

$T_1$

```
while (true) {  
    try1 = true;  
    turn = 2;  
    while (try2 && turn != 1);  
    critical section  
    try1 = false;  
    outside of critical section  
}
```

$T_2$

```
while (true) {  
    try2 = true;  
    turn = 1;  
    while (try1 && turn != 2);  
    critical section  
    try2 = false;  
    outside of critical section  
}
```

**Does it satisfy the liveness requirement?**

**Does it satisfy the safety requirement?**

# Proof Sketch of Peterson's Algorithm

```
int turn = 1;  
 $T_1$  bool try1 = false, try2 = false;  $T_2$ 
```

```
while (true) {  
{¬ try1 ∧ (turn == 1 ∨ turn == 2)}  
1   try1 = true;  
{ try1 ∧ (turn == 1 ∨ turn == 2)}  
2   turn = 2;  
{ try1 ∧ (turn == 1 ∨ turn == 2)}  
3   while (try2 && turn != 1);  
{ try1 ∧ (turn == 1 ∨ ¬ try2 ∨  
(try2 ∧ (line at 6 or at 7))) }  
    critical section  
4   try1 = false;  
{¬ try1 ∧ (turn == 1 ∨ turn == 2)}  
    outside of critical section  
}
```

```
while (true) {  
{¬ try2 ∧ (turn == 1 ∨ turn == 2)}  
5   try2 = true;  
{ try2 ∧ (turn == 1 ∨ turn == 2)}  
6   turn = 1;  
{ try2 ∧ (turn == 1 ∨ turn == 2)}  
7   while (try1 && turn != 2);  
{ try2 ∧ (turn == 2 ∨ ¬ try1 ∨  
(try1 ∧ (line at 2 or at 3))) }  
    critical section  
8   try2 = false;  
{¬ try2 ∧ (turn == 1 ∨ turn == 2)}  
    outside of critical section  
}
```

Safety property: (line 4)  $\wedge$  (line 8)  $\Rightarrow$  (turn == 1  $\wedge$  turn == 2)

# Locks

**A lock is an object in memory providing two operations**

- `acquire()`: wait until lock is free, then take it to enter a C.S
- `release()`: release lock to leave a C.S, waking up anyone waiting for it

# Too Much Milk: Solution #4

## Solution #4: lock

Alice

```
lock.acquire();
if (milk == 0) {
    milk++;
}
lock.release();
```

Bob

```
lock.acquire();
if (milk == 0) {
    milk++;
}
lock.release();
```

# Locks

**A lock is an object in memory providing two operations**

- `acquire()`: wait until lock is free, then take it to enter a C.S
- `release()`: release lock to leave a C.S, waking up anyone waiting for it

**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?

# Fix Banking Problem with Lock

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

# Locks

**A lock is an object in memory providing two operations**

- `acquire()`: wait until lock is free, then take it to enter a C.S
- `release()`: release lock to leave a C.S, waking up anyone waiting for it

**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 (a mutex)**

- Can break apart Peterson's to implement a spinlock

# 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

Called a **spinlock** because a thread spins waiting for the lock to be released

# Implementing Locks (2)

## The while is not atomic:

- 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/enable interrupts (prevents context switches)

# Atomic Operations

**Atomic Operation:** an operation that always runs to completion or not at all

- cannot be stopped in the middle
- cannot be modified by someone else in the middle
- fundamental building block for synchronization

On most machines, memory references and assignments are atomic

Many instructions are not atomic

- Double-precision floating point store often not atomic

# Atomic Instructions: Test-And-Set

The semantics of test-and-set are:

- Record the old value
- Set the value to indicate available
- Return the old value

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

**Hardware executes it atomically!**

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?

Other similar flavor atomic instructions: xchg, CAS

# Using Test-And-Set to Implement Lock

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?

Implement it with xchg, Compare-And-Swap

# **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?**

- Lock holder calls yield or sleep
- Involuntary 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 {  
    int held = 0;  
}  
void acquire (lock) {  
    disable interrupts;  
}  
  
void release (lock) {  
    enable 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 Pintos 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** (either 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

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

## Disabling Interrupts:

- Disabling 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

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

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

Pintos [threads/synch.c](#): sema\_down/up

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

Interrupts Disabled  
Interrupts Enabled  
Interrupts Disabled

# **Summary**

**Why we need synchronizations**

**Critical sections**

**Simple algorithms to implement critical sections**

**Locks**

**Lock implementations**

# **Next Time...**

**Read Chapters 30,31**