

# Theory of Operating Systems

## Week 4: Threads and Concurrency

Faye (Chi Zhang)

CUNY Hunter

February 25, 2026

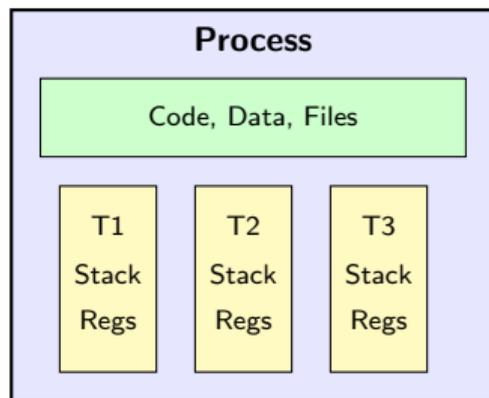
# Today's Agenda

- 1 Introduction to Threads
- 2 Thread Models
- 3 POSIX Threads (pthreads)
- 4 Thread Synchronization Preview
- 5 Thread Pools
- 6 Practical Exercises
- 7 Summary

# Why Threads?

## Limitations of Processes:

- Process creation is expensive (fork overhead)
- Context switching between processes is costly
- Inter-process communication requires special mechanisms
- Each process has separate address space



## Solution: Threads

- Lightweight execution units within a process
- Share the same address space
- Cheaper to create and switch

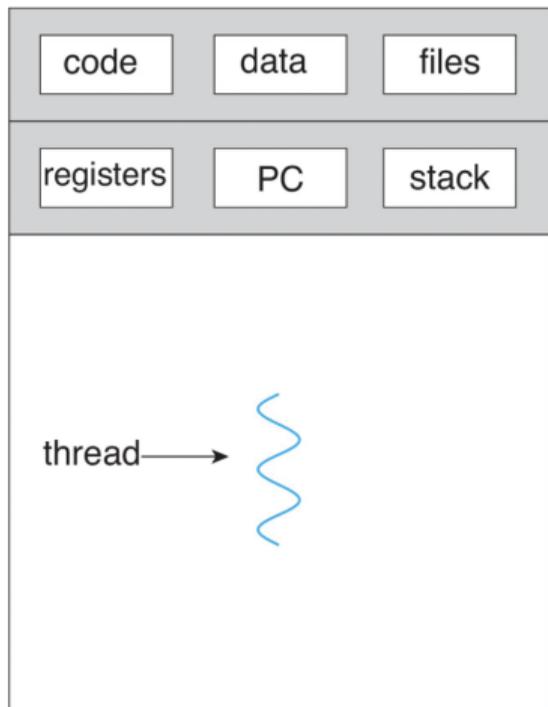
# Thread vs Process

Aspect	Process	Thread
Address Space	Separate	Shared
Creation Time	Heavy (ms)	Light ( $\mu s$ )
Context Switch	Expensive	Cheap
Communication	IPC required	Direct memory
Isolation	Strong	Weak
Crash Impact	Contained	May crash process

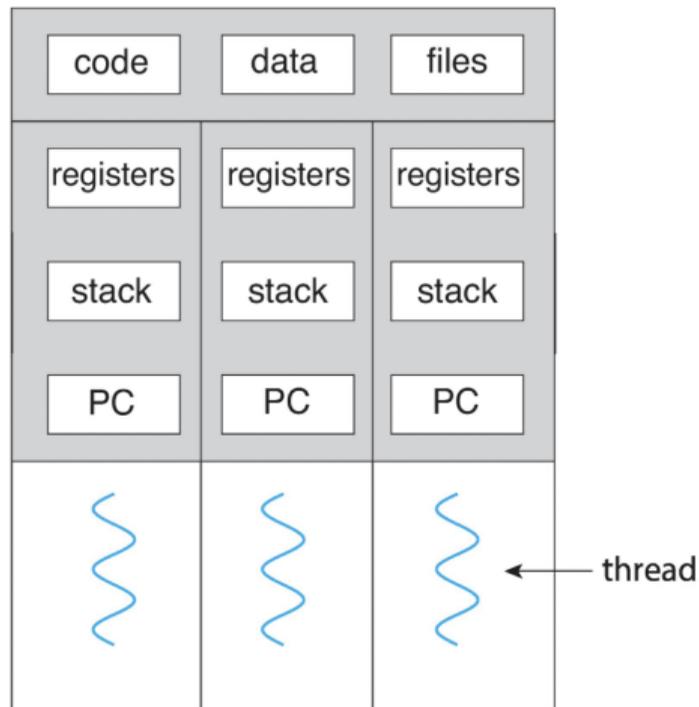
## Key Insight

Threads trade isolation for efficiency. Use threads when you need parallelism within a single task; use processes when you need fault isolation.

# Multithreaded Process



single-threaded process



multithreaded process

# What Threads Share vs Own

## Shared (per-process):

- Code (text segment)
- Global variables (data segment)
- Heap memory
- Open file descriptors
- Current working directory
- User and group IDs
- Signal handlers

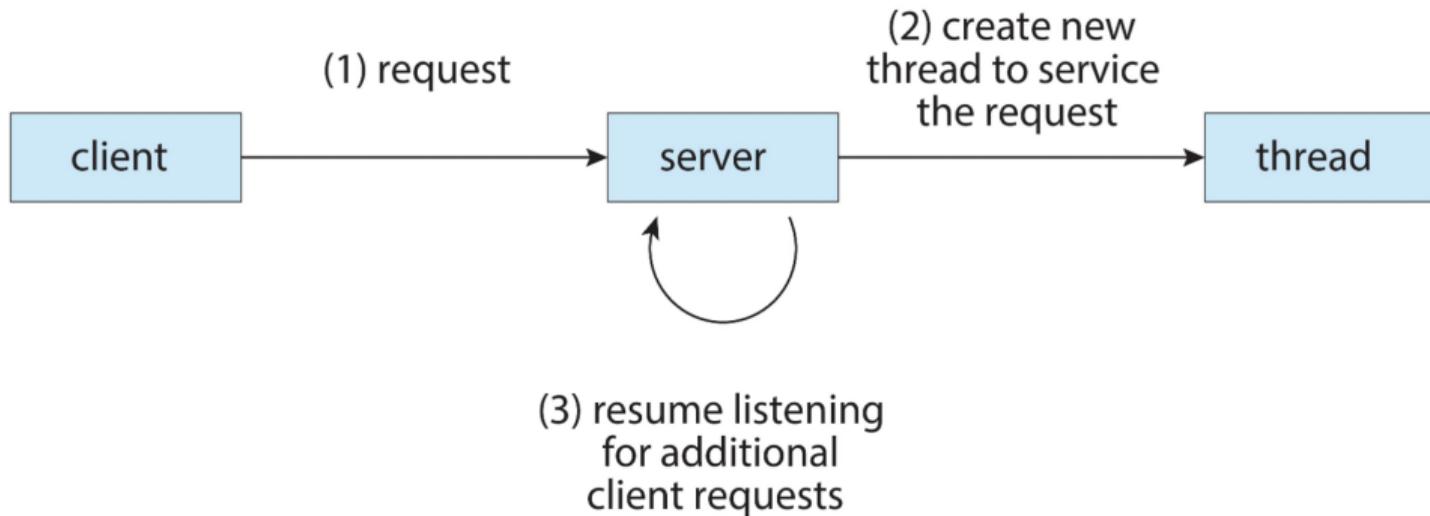
## Private (per-thread):

- Thread ID
- Stack (local variables)
- CPU registers (PC, SP, etc.)
- Stack pointer
- Scheduling priority
- Signal mask
- errno value

## Memory Layout Implication

Each thread needs its own stack for function calls and local variables, but they all access the same heap and global data.

# Multithreaded Server Architecture



# Benefits of Multithreading

## ① Responsiveness

- UI thread remains responsive while worker threads compute
- Non-blocking I/O operations

## ② Resource Sharing

- Threads share memory automatically
- No need for explicit IPC mechanisms

## ③ Economy

- Creating threads is 10-100x faster than processes
- Context switching is 5-10x faster

## ④ Scalability

- Can utilize multiple CPU cores
- Better throughput on multiprocessor systems

# Challenges of Multithreading

## Correctness Issues:

- Race conditions
- Deadlocks
- Data corruption
- Non-deterministic behavior

## Debugging Difficulty:

- Heisenbugs (disappear when observed)
- Hard to reproduce errors
- Timing-dependent bugs

## Race Condition Example

Two threads incrementing a counter:

```
// Thread 1      // Thread 2
read count      read count
count++         count++
write count     write count
```

Expected: 2, Possible: 1

# User Threads vs. Kernel Threads

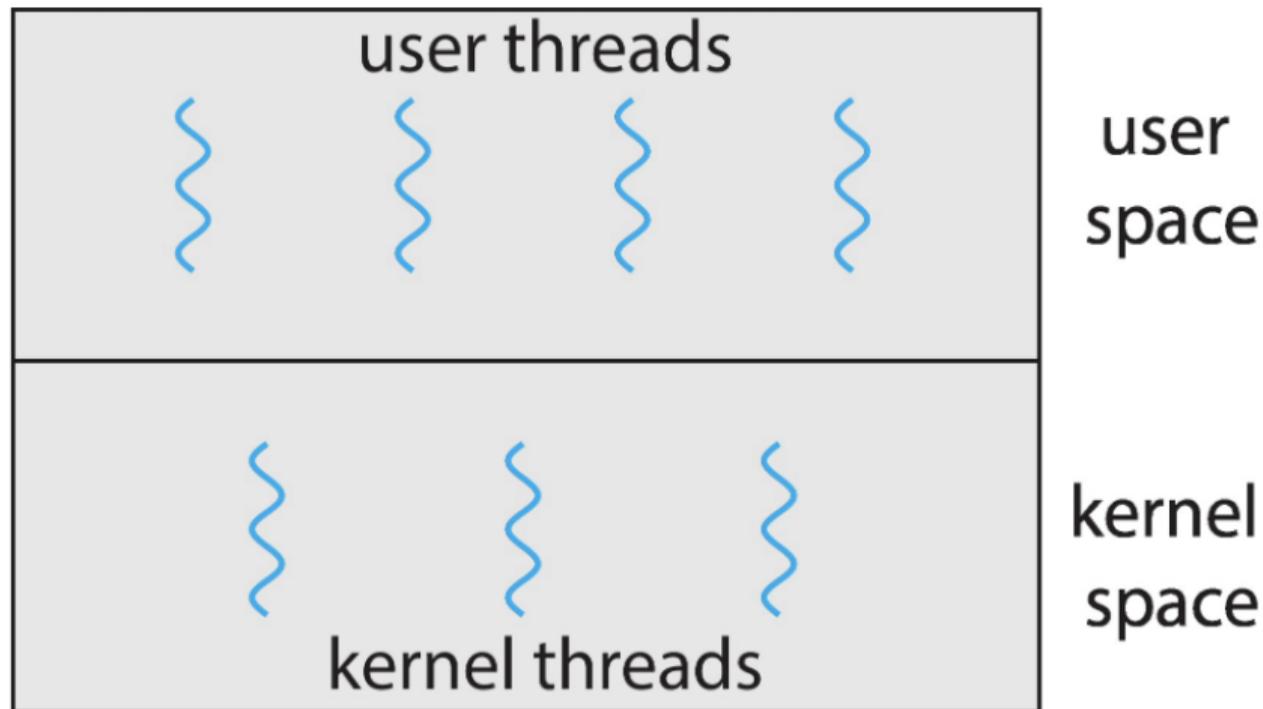
## User threads

- Managed by a user-level thread library
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads

## Kernel threads

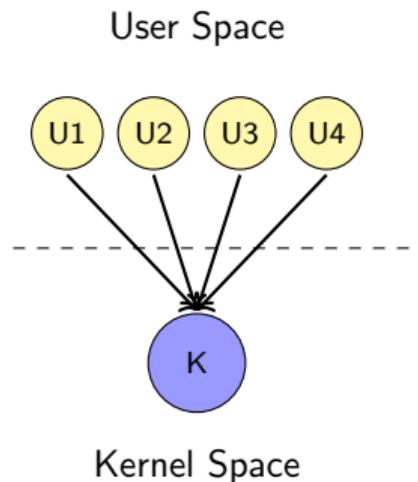
- Supported and scheduled by the OS kernel
- Examples (general-purpose OS):
  - Windows
  - Linux
  - macOS / Mac OS X
  - iOS
  - Android

# User-Level vs Kernel-Level Threads



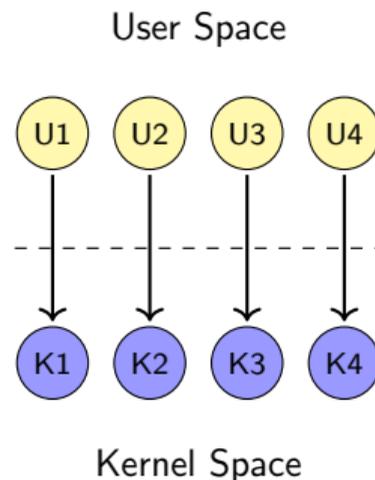
# Many-to-One Model (N:1)

- Many user threads  $\rightarrow$  one kernel thread
- Thread library in user space
- **Advantages:**
  - Very fast thread operations
  - Portable across OS
- **Disadvantages:**
  - No true parallelism
  - Blocking syscall blocks all
- Examples: Green threads (early Java), GNU Portable Threads



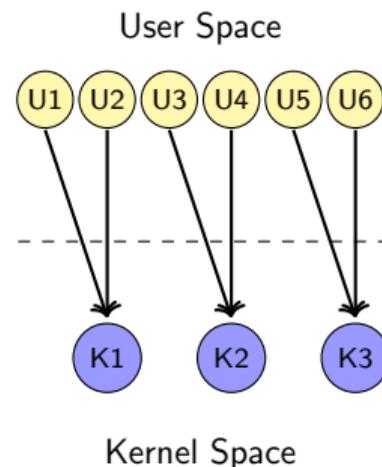
# One-to-One Model (1:1)

- Each user thread  $\rightarrow$  one kernel thread
- Most common modern approach
- **Advantages:**
  - True parallelism on multi-core
  - Blocking doesn't affect others
- **Disadvantages:**
  - Higher overhead (syscalls)
  - Limited by kernel resources
- Examples: Linux (NPTL), Windows, macOS



# Many-to-Many Model (M:N)

- M user threads  $\rightarrow$  N kernel threads
- Hybrid approach
- **Advantages:**
  - True parallelism (up to N cores)
  - Flexible scheduling
  - Can have many user threads
- **Disadvantages:**
  - Complex implementation
  - Scheduling coordination
- Examples: Solaris (historical), Go runtime



# Thread Model Comparison

Feature	N:1	1:1	M:N
Parallelism	None	Full	Partial
Thread Creation	Fast	Slow	Medium
Context Switch	Fast	Slow	Medium
Blocking Syscall	Blocks all	Individual	Depends
Complexity	Low	Low	High
Scalability	Limited	By kernel	Flexible

## Modern Trend

Most modern systems use 1:1 model (Linux NPTL, Windows). The M:N model sees renewed interest in languages like Go (goroutines) and Rust (async/await).

# Linux Threading: NPTL

- **NPTL** = Native POSIX Thread Library
- Replaced older LinuxThreads implementation
- Uses 1:1 threading model
- Key characteristics:
  - Uses `clone()` syscall with specific flags
  - Each thread is a `task_struct` in kernel
  - Shared: address space, file descriptors, signal handlers
  - Private: stack, thread ID, signal mask

## Thread Creation via `clone()`

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD, ...)
```

- `CLONE_VM`: Share virtual memory
- `CLONE_THREAD`: Same thread group

# Introduction to pthreads

- **POSIX Threads:** Standardized threading API (IEEE 1003.1c)
- Portable across UNIX-like systems
- Header: `#include <pthread.h>`
- Link with: `-lpthread` (or `-pthread`)

## Core Data Types

<code>pthread_t</code>	Thread identifier
<code>pthread_attr_t</code>	Thread attributes
<code>pthread_mutex_t</code>	Mutex (mutual exclusion)
<code>pthread_cond_t</code>	Condition variable
<code>pthread_key_t</code>	Thread-specific data key

# Creating Threads: pthread\_create()

```
#include <pthread.h>

int pthread_create(
    pthread_t *thread,          // Output: thread ID
    const pthread_attr_t *attr, // Thread attributes (NULL for default)
    void *(*start_routine)(void *), // Function to execute
    void *arg                  // Argument to function
);
// Returns: 0 on success, error number on failure
```

- Creates a new thread executing start\_routine(arg)
- Thread ID stored in \*thread
- attr = NULL uses default attributes
- Returns immediately (thread runs concurrently)

# Simple Thread Example

```
#include <stdio.h>
#include <pthread.h>

void *print_message(void *arg) {
    char *msg = (char *)arg;
    printf("Thread says: %s\n", msg);
    return NULL;
}

int main() {
    pthread_t thread;
    char *message = "Hello from thread!";

    pthread_create(&thread, NULL, print_message, message);
    pthread_join(thread, NULL); // Wait for thread to finish

    printf("Main thread done.\n");
    return 0;
}
```

# Waiting for Threads: pthread\_join()

```
int pthread_join(  
    pthread_t thread,    // Thread to wait for  
    void **retval        // Output: thread's return value (or NULL)  
);  
// Returns: 0 on success, error number on failure
```

## Behavior:

- Blocks until thread terminates
- Retrieves thread's return value
- Releases thread resources

## Important:

- Can only join once per thread
- Must join or detach all threads
- Failure to join = resource leak

# Thread Return Values

```
void *compute_sum(void *arg) {
    int *nums = (int *)arg;
    int *result = malloc(sizeof(int));
    *result = nums[0] + nums[1];
    return result; // Return pointer to result
}

int main() {
    pthread_t thread;
    int data[] = {10, 20};
    void *retval;

    pthread_create(&thread, NULL, compute_sum, data);
    pthread_join(thread, &retval);

    int *sum = (int *)retval;
    printf("Sum = %d\n", *sum); // Output: Sum = 30
    free(sum);
    return 0;
}
```

# Terminating Threads

## Three ways a thread can terminate:

- 1 Return from `start_routine`
- 2 Call `pthread_exit()`
- 3 Cancelled by another thread

```
void pthread_exit(void *retval);  
// Does not return. Thread terminates immediately.  
  
// Example:  
void *worker(void *arg) {  
    if (error_condition) {  
        pthread_exit(NULL); // Terminate early  
    }  
    // ... normal work ...  
    return result;  
}
```

## Warning

Calling `exit()` from any thread terminates the **entire process**.

# Detached Threads

```
int pthread_detach(pthread_t thread);  
// Returns: 0 on success, error number on failure
```

- Detached thread: resources automatically released on termination
- Cannot be joined after detaching
- Useful for "fire and forget" threads

```
void *background_task(void *arg) {  
    // Do work, then exit  
    return NULL;  
}  
  
int main() {  
    pthread_t thread;  
    pthread_create(&thread, NULL, background_task, NULL);  
    pthread_detach(thread); // No need to join  
    // Main continues without waiting  
    // ...  
}
```

# Creating Multiple Threads

```
1 #define NUM_THREADS 5
2
3 void *worker(void *arg) {
4     int id = *(int *)arg;
5     printf("Thread %d running\n", id);
6     return NULL;
7 }
8
9 int main() {
10    pthread_t threads[NUM_THREADS];
11    int ids[NUM_THREADS];
12
13    for (int i = 0; i < NUM_THREADS; i++) {
14        ids[i] = i;
15        pthread_create(&threads[i], NULL, worker, &ids[i]);
16    }
17
18    for (int i = 0; i < NUM_THREADS; i++) {
19        pthread_join(threads[i], NULL);
20    }
21    return 0;
22 }
```

# Common Pitfall: Passing Arguments

## WRONG:

```
1 for (int i = 0; i < N; i++) {  
2     pthread_create(&t[i], NULL,  
3         worker, &i); // BUG!  
4 }  
5 // Problem: i changes before  
6 // thread reads it  
7
```

## CORRECT:

```
1 int ids[N];  
2 for (int i = 0; i < N; i++) {  
3     ids[i] = i;  
4     pthread_create(&t[i], NULL,  
5         worker, &ids[i]); // OK  
6 }  
7 // Each thread has own copy
```

## Alternative: Cast to void\*

```
pthread_create(&t[i], NULL, worker, (void*)(intptr_t)i);  
// In worker: int id = (int)(intptr_t)arg;
```

# Thread Attributes

```
pthread_attr_t attr;
pthread_attr_init(&attr);

// Set detach state
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

// Set stack size
pthread_attr_setstacksize(&attr, 2 * 1024 * 1024); // 2 MB

// Set scheduling policy
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

// Create thread with attributes
pthread_create(&thread, &attr, worker, NULL);

// Clean up
pthread_attr_destroy(&attr);
```

# Thread Attributes Summary

Attribute	Functions	Description
Detach state	setdetachstate	Joinable or detached
Stack size	setstacksize	Thread stack size
Stack address	setstackaddr	Custom stack location
Guard size	setguardsize	Overflow protection
Scheduling	setschedpolicy	FIFO, RR, OTHER
Priority	setschedparam	Scheduling priority
Inherit sched	setinheritsched	Inherit from parent
Scope	setscope	System or process

## Best Practice

Use default attributes (NULL) unless you have specific requirements. Customize only when necessary.

# Thread-Specific Data (TSD)

**Problem:** Global variables are shared by all threads.

**Solution:** Thread-specific data provides per-thread storage.

```
1 pthread_key_t key;
2
3 void destructor(void *value) {
4     free(value); // Called when thread exits
5 }
6
7 // Once at program start
8 pthread_key_create(&key, destructor);
9
10 // In each thread
11 int *mydata = malloc(sizeof(int));
12 *mydata = thread_id;
13 pthread_setspecific(key, mydata);
14
15 // Later in same thread
16 int *data = pthread_getspecific(key);
```

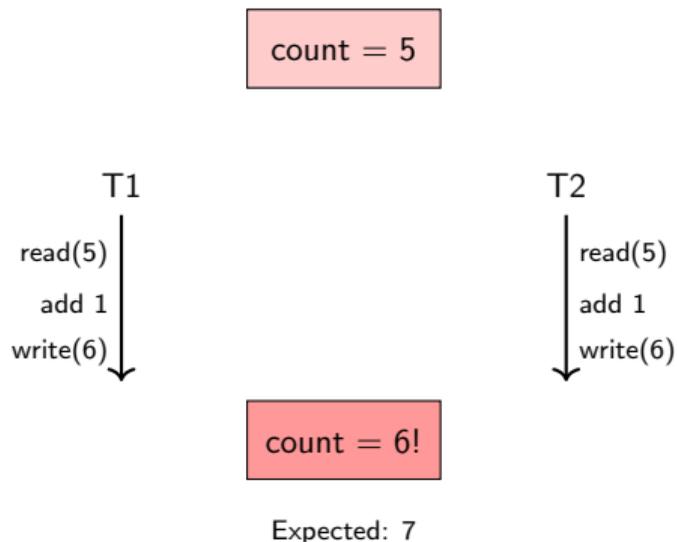
# The Need for Synchronization

## The Problem:

- Threads share memory
- Concurrent access to shared data
- Operations may interleave
- Results become unpredictable

## Race Condition:

- Outcome depends on timing
- Non-deterministic behavior
- Hard to debug



# Race Condition Example

```
#include <pthread.h>
#include <stdio.h>

int counter = 0; // Shared variable

void *increment(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        counter++; // Not atomic!
    }
    return NULL;
}

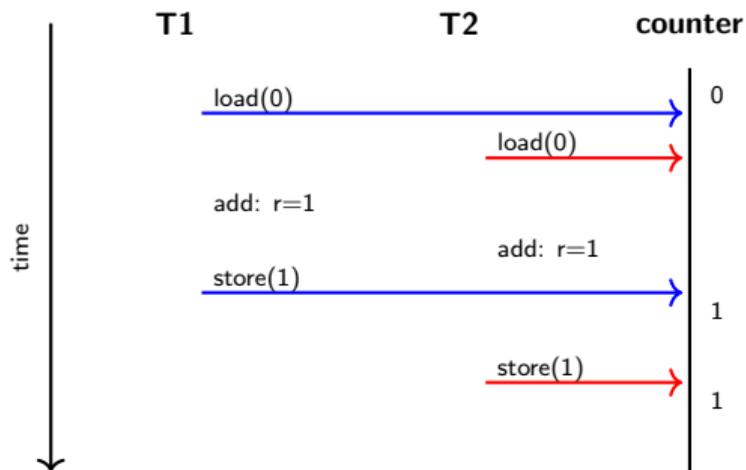
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter = %d\n", counter); // Should be 2000000
    return 0; // Actually: random < 2000000
}
```

# Why counter++ is Not Atomic

counter++ compiles to:

- 1 LOAD counter → register
- 2 ADD 1 to register
- 3 STORE register → counter

**Threads can interleave at any point!**



## Result

Both threads read 0, both increment to 1, both store 1. We lost one increment!

# Critical Section

## Definition

A **critical section** is a code segment that accesses shared resources and must not be executed by more than one thread at a time.

## Requirements for correct critical section handling:

- 1 **Mutual Exclusion**: Only one thread in CS at a time
- 2 **Progress**: If no thread is in CS, one waiting thread must be allowed to enter
- 3 **Bounded Waiting**: Limit on how long a thread waits to enter CS

## General Pattern

```
entry_section(); // Acquire lock
// Critical Section
exit_section(); // Release lock
```

# Mutex: Mutual Exclusion Lock

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

void *increment(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&mutex);    // Enter CS
        counter++;                      // Critical section
        pthread_mutex_unlock(&mutex); // Exit CS
    }
    return NULL;
}
```

- `pthread_mutex_lock()`: Blocks if mutex is held
- `pthread_mutex_unlock()`: Releases mutex
- Now counter will correctly be 2000000

We will cover these in detail in Week 5:

## ① Mutex (Mutual Exclusion)

- Binary lock: locked or unlocked
- Only owner can unlock

## ② Semaphore

- Counting synchronization primitive
- Can allow N threads simultaneously

## ③ Condition Variable

- Wait for a condition to become true
- Used with mutex for complex synchronization

## ④ Read-Write Lock

- Multiple readers OR one writer
- Optimizes read-heavy workloads

# Motivation for Thread Pools

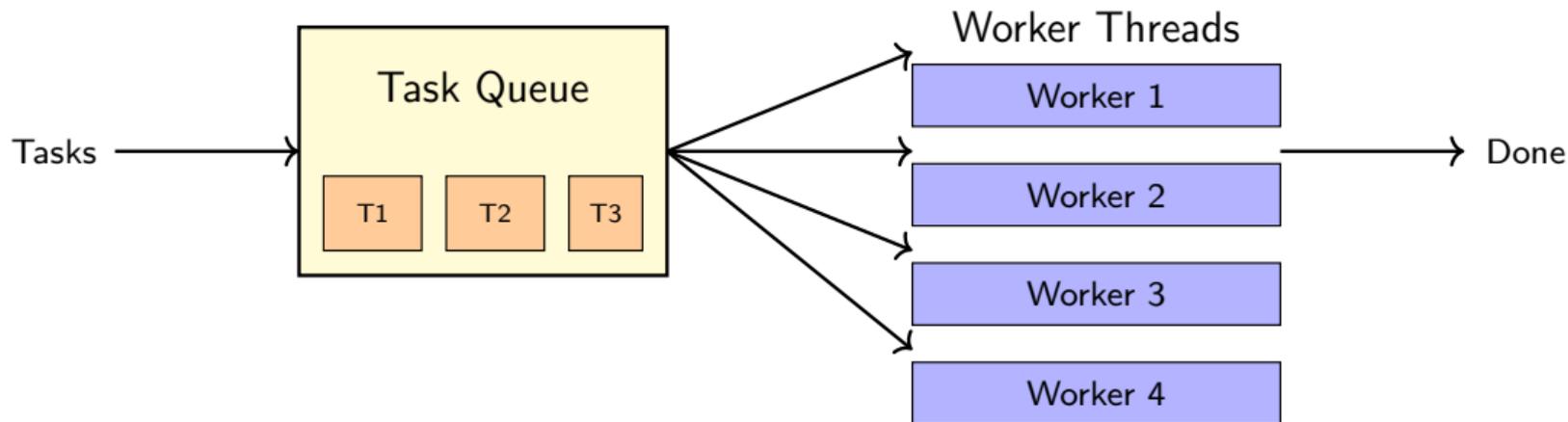
## Problem with creating threads per task:

- Thread creation has overhead (stack allocation, kernel structures)
- Many short tasks = many create/destroy cycles
- Unbounded thread creation can exhaust resources

## Solution: Thread Pool

- Create a fixed number of worker threads at startup
- Tasks submitted to a queue
- Workers pick tasks from queue and execute them
- Threads are reused for multiple tasks

# Thread Pool Architecture



# Thread Pool Benefits

## Performance:

- Eliminates thread creation overhead
- Reduced memory footprint
- Better cache utilization

## Resource Management:

- Bounded number of threads
- Prevents resource exhaustion
- Predictable memory usage

## Flexibility:

- Queue provides backpressure
- Can prioritize tasks
- Graceful degradation under load

## Use Cases:

- Web servers
- Database connection pools
- Parallel computation
- Event processing

# Thread Pool Sizing

## How many threads should a pool have?

- **CPU-bound tasks:** threads  $\approx$  number of CPU cores
  - More threads = context switch overhead
  - Example: Image processing, encryption
- **I/O-bound tasks:** threads  $>$  number of CPU cores
  - Threads spend time waiting for I/O
  - Can overlap I/O with computation
  - Example: Web requests, file I/O

## Rule of Thumb

For mixed workloads:  $N_{threads} = N_{cores} \times (1 + \frac{W}{C})$   
where  $W$  = wait time,  $C$  = compute time

# Thread Pool in Practice

## Real-world implementations:

- **Java:** `ExecutorService`, `ThreadPoolExecutor`
- **C++11:** `std::async`, custom implementations
- **Python:** `concurrent.futures.ThreadPoolExecutor`
- **Go:** Goroutines with channel-based scheduling
- **POSIX:** Manual implementation or libraries (`libuv`, `GLib`)

## Design Considerations

- Task queue: bounded vs unbounded
- Rejection policy when queue is full
- Thread lifecycle: core vs max threads
- Work stealing for load balancing

# Exercise 1: Basic Thread Creation

## Task

Write a program that creates 4 threads. Each thread should:

- 1 Print its thread ID
- 2 Sleep for a random time (1-3 seconds)
- 3 Print a completion message

The main thread should wait for all threads to complete.

## Expected Output (order may vary):

```
Thread 0 starting
```

```
Thread 1 starting
```

```
Thread 2 starting
```

```
Thread 3 starting
```

```
Thread 1 finished
```

```
Thread 0 finished
```

## Exercise 2: Parallel Sum

### Task

Given an array of 1,000,000 integers, compute the sum using multiple threads:

- 1 Divide the array into N equal parts
- 2 Create N threads, each computing partial sum
- 3 Collect partial sums and compute total

Compare performance with single-threaded version.

### Hints:

- Pass array bounds as thread argument
- Use `pthread_join` to get return values
- Measure time with `clock_gettime()`
- Try different thread counts: 1, 2, 4, 8

## Exercise 3: Producer-Consumer Preview

### Task

Implement a simple producer-consumer pattern:

- 2 producer threads adding items to a shared buffer
- 2 consumer threads removing items
- Buffer size: 10 items
- Each producer adds 50 items
- Each consumer removes 50 items

**This exercise previews Week 5 topics:**

- Mutex for mutual exclusion
- Condition variables for signaling
- Bounded buffer synchronization

# Key Takeaways

## 1 Threads vs Processes

- Threads share address space, processes don't
- Threads are lighter but less isolated

## 2 Thread Models

- N:1, 1:1, M:N models with different tradeoffs
- Modern systems mostly use 1:1 (Linux NPTL)

## 3 POSIX Threads

- `pthread_create`, `pthread_join`, `pthread_exit`
- Thread attributes for customization

## 4 Synchronization

- Shared data requires protection
- Mutexes prevent race conditions

## 5 Thread Pools

- Reuse threads for efficiency
- Size based on workload characteristics

## Mistakes to Watch For

- 1 Passing stack variables to threads (use heap or static)
- 2 Forgetting to join or detach threads (resource leak)
- 3 Accessing shared data without synchronization
- 4 Calling `exit()` instead of `pthread_exit()`
- 5 Creating too many threads (exhausting resources)
- 6 Deadlocks from improper lock ordering

# Next Week: Synchronization

## Week 5 Preview:

- Mutex in depth: types, attributes, error checking
- Condition variables: wait, signal, broadcast
- Semaphores: counting and binary
- Read-write locks
- Classic synchronization problems:
  - Producer-Consumer
  - Readers-Writers
  - Dining Philosophers
- Deadlock detection and prevention

## Preparation

Review mutex usage from today. Try Exercise 3 (producer-consumer) before next class.

- **Quiz:** Available now, covers threads and concurrency basics
- **Assignment:** TBD
- **Reading:** Chapter on threads in your textbook

**Questions?**