# Lecture 14: Shared-Memory Concurrency

## (Multithreading Model, Libraies, and Challenges)

Xin Liu

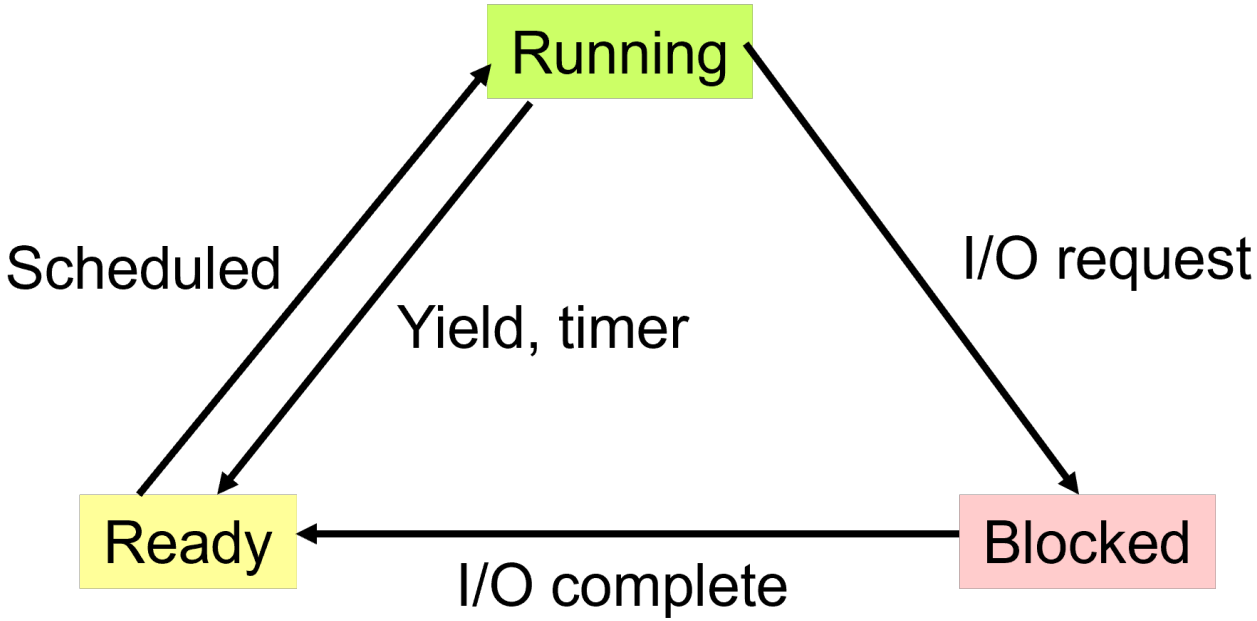
Florida State University
xliu15@fsu.edu

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

# Why Threads?

The UNIX process model, with its isolated address spaces, was a huge success. It provided a clean way to run multiple programs.
However, a critical limitation emerged: A process has only one flow of execution.
What happens if this single flow blocks on a slow I/O operation like `read()`?

Remember from last lecture: Per-thread State Diagram



# Why Threads?

The UNIX process model, with its isolated address spaces, was a huge success. It provided a clean way to run multiple programs. However, a critical limitation emerged: A process has only one flow of execution.
What happens if this single flow blocks on a slow I/O operation like `read()`?

- The **entire process freezes**. All its resources (memory, open files) sit idle.
- **Valuable CPU time is wasted** that could have been used for other tasks within the same program (e.g., updating a UI, performing calculations).

# Why Threads?

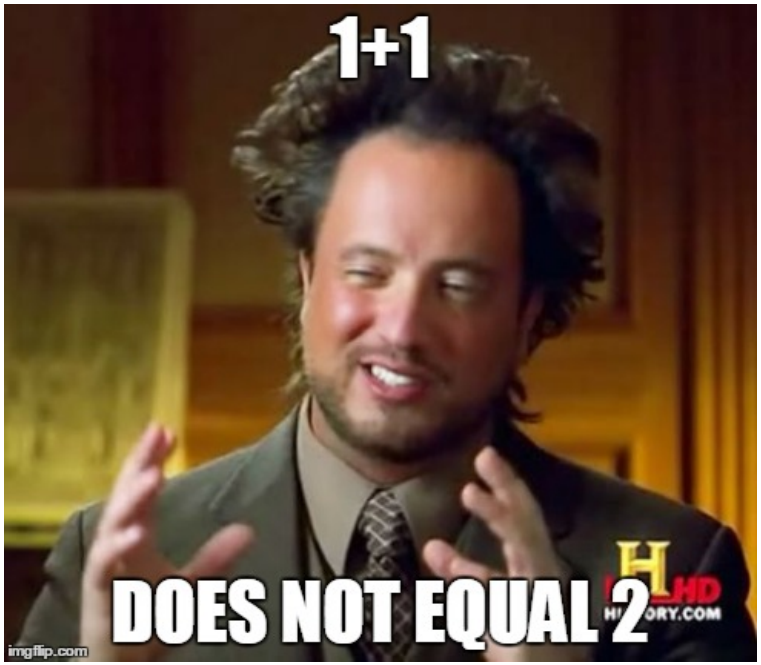
The UNIX process model, with its isolated address spaces, was a huge success. It provided a clean way to run multiple programs. However, a critical limitation emerged: A process has only one flow of execution.
What happens if this single flow blocks on a slow I/O operation like `read()`?

- The **entire process freezes**. All its resources (memory, open files) sit idle.
- **Valuable CPU time is wasted** that could have been used for other tasks within the same program (e.g., updating a UI, performing calculations).

Furthermore, hardware evolved to have multiple CPUs. How could a single program utilize all cores simultaneously?
Process-level parallelism felt too heavyweight and inefficient for these new needs. We needed multiple execution flows that could **share memory and resources**.

# Introduction of Threads

- Multithreading Model
- Libraries
- Challenges: **Why Doesn't 1+1 Equal 2?**



# Shared-Memory Multithreading Model

# 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)

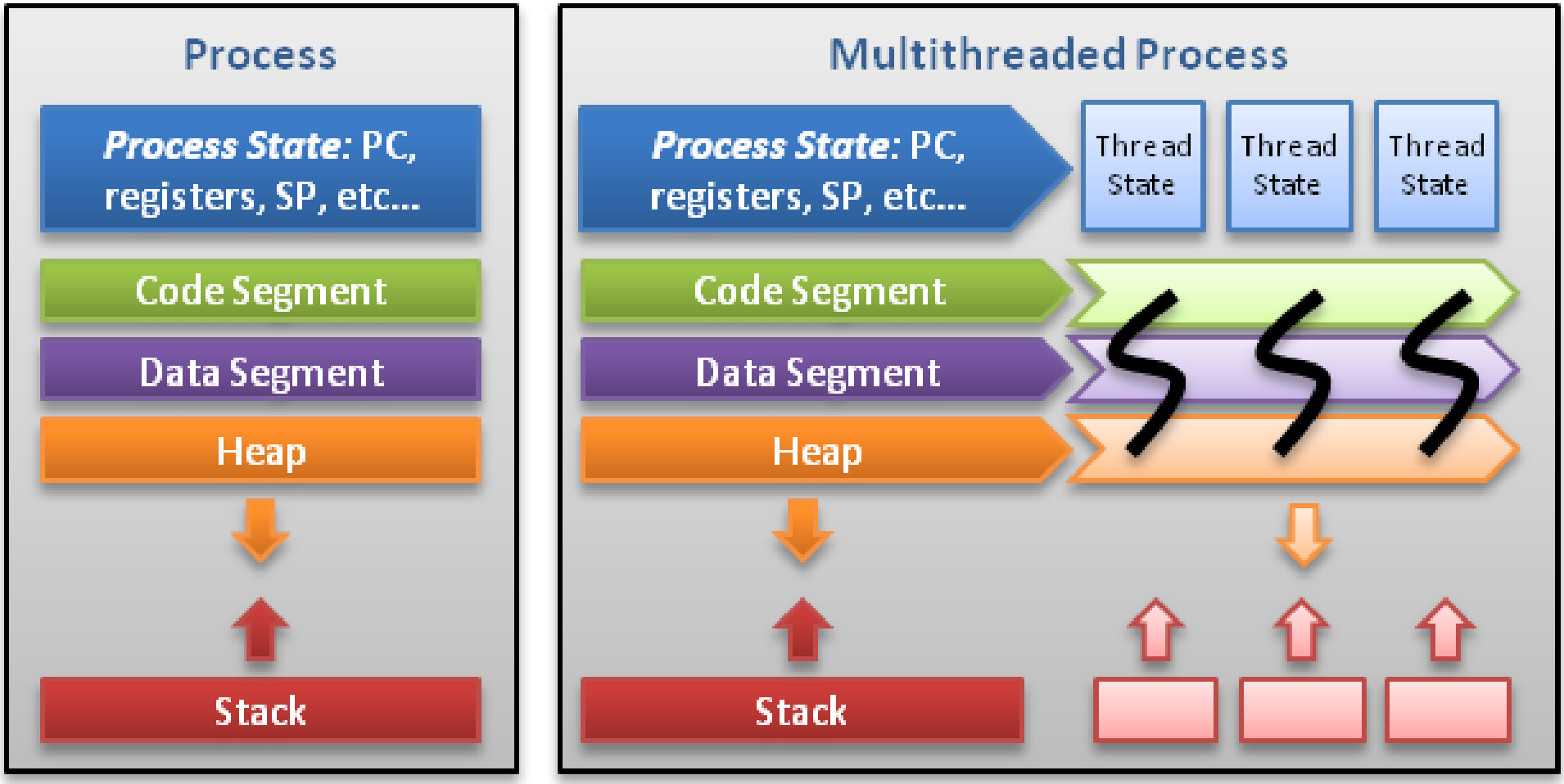
**State-machine model of a multithreaded program**

- Add APIs to create new threads, e.g., pthread_create()
  - Provided by the POSIX Threads (Pthreads) library on UNIX-like systems
  - On Linux, this library call uses the clone() system call
- Adds another "state machine" with its own stack but shared global variables

**Modeling the state transition:**

- Multiple state machines can now appear to run concurrently.
- The system (OS scheduler) **interleaves** their execution.
- At any moment, it **chooses one** state machine and executes **one** of its statements.

# Multithreading Model




Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

# Concurrency vs. Parallelism

## Concurrency

Dealing with many things at once.

- A design property (structure).
- Tasks are *interleaved*.
- Can be on a single CPU core.

## Parallelism

Doing many things at once.

- An execution property (hardware).
- Tasks run *simultaneously*.
- Requires multiple CPU cores.

# Comparison: Creating Execution Flows

| Call | Primary Goal | Level | Address Space | Portability |
|---|---|---|---|---|
| `fork()` | Create a Process (a "neighbor") | System Call | Separate (Copy-on-Write) | High (POSIX) |
| `pthread_create()` | Create a Thread (a "roommate") | Library Call | Shared | High (POSIX) |
| `posix_spawn()` | Create a Process | Library Call | Separate | High (POSIX) |
| `clone()` | Create a Task (low-level) | System Call | Configurable | Low (Linux-specific) |

# Comparison: Creating Execution Flows

## Key Takeaways

- **Process vs. Thread:** `fork` / `posix_spawn` create isolated processes. `pthread_create` creates threads that share memory.
- **Library vs. Syscall:** `pthread_create` and `posix_spawn` are convenient, portable library functions. `fork` and `clone` are the low-level system calls that do the actual work in the kernel.
- `clone()` is the powerful, low-level Linux primitive that underlies both modern process and thread creation.

# Thread Libraries

# The Standard Way: Using the Pthreads Library

To create a thread in C, we use the POSIX (Pthreads) library.

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

// 1. The required function signature
void *hello() {
    printf("Hello\n");
    return NULL;
}

int main() {
    pthread_t t1;                          // 2. A variable to hold the thread

    pthread_create(&t1, NULL, hello, (void *)1L); // 3. Create the thread

    pthread_join(t1, NULL);                // 4. Wait for the thread to finish
    return 0;
}
```

# The Standard Way: Using the Pthreads Library

While powerful, this involves several mandatory, verbose steps for even the simplest tasks.

# The Standard Way: Using the Pthreads Library

While powerful, this involves several mandatory, verbose steps for even the simplest tasks.

- **Step 1: The Signature**
  Your function *must* accept a ‘void*‘ and return a ‘void*‘. This forces you to constantly cast your data back and forth.

```
void *hello(void *arg)
{
   // ...
}
```

# The Standard Way: Using the Pthreads Library

While powerful, this involves several mandatory, verbose steps for even the simplest tasks.

- **Step 1: The Signature**
  Your function *must* accept a ‘void*‘ and return a ‘void*‘. This forces you to constantly cast your data back and forth.
- **Step 2: The Handle**
  You must manually declare a ‘pthread_t‘ variable for every thread you want to manage.

```
pthread_t t1;
```

# The Standard Way: Using the Pthreads Library

While powerful, this involves several mandatory, verbose steps for even the simplest tasks.

- **Step 1: The Signature**
  Your function *must* accept a ‘void*‘ and return a ‘void*‘. This forces you to constantly cast your data back and forth.
- **Step 2: The Handle**
  You must manually declare a ‘pthread_t‘ variable for every thread you want to manage.
- **Step 3: The Creation**
  ‘pthread_create()‘ takes four arguments, including the address of the handle and casting the argument to ‘void*‘.

```
pthread_create(&t1, NULL, hello, (void *)1L);
```

# The Standard Way: Using the Pthreads Library

While powerful, this involves several mandatory, verbose steps for even the simplest tasks.

- **Step 1: The Signature**
  Your function *must* accept a ‘void*‘ and return a ‘void*‘. This forces you to constantly cast your data back and forth.
- **Step 2: The Handle**
  You must manually declare a ‘pthread_t‘ variable for every thread you want to manage.
- **Step 3: The Creation**
  ‘pthread_create()‘ takes four arguments, including the address of the handle and casting the argument to ‘void*‘.
- **Step 4: The Cleanup**
  You must manually call ‘pthread_join()‘ for each thread to ensure your main program waits for it to complete.

```
pthread_join(t1, NULL);
```

# Our Solution: A Simple Wrapper ('pthread.h')

Our custom header file simplifies this entire process dramatically

```
void hello() {
      printf("Hello\n");
      return NULL;
}

int main() {
    spawn(hello); // join() is called automatically!
}
```

## Key Benefits:

- Write simple functions: `void my_func()`.
- Use a clean, intuitive command: `spawn(my_func)`.
- No need to manually manage `pthread_t` variables or call `join()`.

Have A Try: thread-lib

# How thread.h Works

## 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) { /* ... */ }
```

# How It Works: Automatic Cleanup with 'join()'

How does the program wait for threads to finish without a 'join()' call in 'main()'?

```
__attribute__((constructor))
static void startup() {
    atexit(join);
}
```

## The Magic: `atexit()` and `__attribute__((constructor))`

1. A 'startup()' function is marked with `__attribute__((constructor))`. This is a GCC/Clang trick that makes it run **before** 'main()'.
2. This 'startup()' function calls `atexit(join)`.
3. `atexit()` is a standard C function that registers a function (our 'join()') to be called automatically when the program exits normally.

# Multithreaded 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. Have A Try: thread-examples/test_shm.c

# 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?
  Have A Try: thread-examples/test_stack.c

**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 not issues anymore.

# Challenge I: Loss of Determinism in State Transitions

# Loss of Determinism

**Virtualization makes a process believe "the world is only itself"**

- **A program = computation + system calls**
- Pure computation is deterministic.
- With the same initial state (`argv`, `envp`) and the same syscalls, repeated runs yield the same result.

**Concurrency breaks this**

- The scheduler may pick different threads on each run.
- A thread's load may read another thread's store.
- Nondeterministic programs are difficult to debug and fix, because the same bug may not appear in every run.

# Loss of Determinism: Example

```
unsigned int balance = 100;

int T_eBay_withdraw(int amount) {
  if (balance >= amount) {
    balance -= amount;
    return SUCCESS;
  } else {
    return FAIL;
  }
}
```

- What happens if two threads both withdraw $100 at the same time? An account with endless money! Have A Try: eBay
- Bugs and vulnerabilities are no joke: The Mt. Gox Hack resulted in the loss of 650,000 BTC, worth approximately $28 billion.
  - Blockchain Ethereum provides its own execution environment and language (EVM and Solidity) to avoid concurrency, so smart contracts always execute deterministically.

# Item Duplication in Diablo I (1996)

How to dupe items in Diablo I

# Event-Level Concurrency

```
Event(doMouseMove) {
    hoveredItem = Item("$1");
}

// Unexpected interleaved event
Event(clickEvent) {
    hoveredItem = Item("$99");  // <- Shared state
    Inventory.append(hoveredItem);
}

Event(doPickUp) {
    InHand = hoveredItem;
}
```

# You realize even 1 + 1 is hard...

**Task:** Compute $1 + 1 + \cdots + 1$ with a total of $2N$ ones, split across two threads.

```
#define N 100000000
long sum = 0;

void T_sum() {
  for (int i = 0; i < N; i++) sum++;
}

int main() {
  create(T_sum);
  create(T_sum);
  join();
  printf("sum = %ld\n", sum);
}
```

- `sum++` is not atomic: `load` → `add` → `store`.
- The threads overwrite each other's updates. Have A Try: sum

# Consequences of Losing Determinism

Run three `T_sum` threads in parallel. What is the minimum final `sum`? Have A Try: sum-model

- Initial `sum = 0`. Assume each single statement executes atomically.

```
void T_sum() {
  for (int i = 0; i < 3; i++) {
    int t = load(sum);
    t += 1;
    store(sum, t);
  }
}
```

**Both ChatGPT and Gemini answered 3 — they're wrong!**

# What is the answer?

`sum = 2`

- Not 1, because the loop runs three times.

**Value of a mathematical view**

- Trace recovery is NP-complete.
  - Even if each thread behaves deterministically, reconstructing the actual execution order from partial observations is computationally intractable.
- Nondeterminism is fundamentally hard for humans.

# Consequences of Losing Determinism

**Implementing concurrent "1 + 1" is harder than it looks**

- In the 1960s people raced to achieve atomicity on shared memory (mutual exclusion).
- Almost every early solution was wrong.
- Even Dekker's algorithm only proves mutual exclusion for two threads.

**Concurrency touches everything in a computing system**

- Are `libc` functions safe to call in multithreaded programs?
- `printf` is buffered.
  - We showed it using a `fork` example. Have A Try: forkHello.c
- Is two threads doing `buf[pos++] = ch` at the same time dangerous?
- See `man 3 printf`.

# In-Class Quiz

# Challenge II: Loss of Sequential Consistency

# What the Compiler Assumes

**As-if rule and observability**

- Only system calls and I/O are observable.
- Pure loads and stores are not observable to the outside.
- The compiler can reorder, merge, or remove these loads and stores.

**Visibility in Concurrency Is Not Guaranteed by the Compiler**

- Threads communicate through memory.
- Without locks, atomics, or fences, a load may not see another thread's recent store.
- The compiler and the CPU may reorder or cache these accesses without synchronization.
- Code that assumes single-thread order can fail after valid optimizations.

**Data races are undefined behavior**

- In C and C++ a data race gives the compiler freedom.
- The result can be any value or any reordering.

# Spin Wait Looks Clever but is Unsafe

```
while (!flag) ;
```

**What we think it does**

- Wait until another thread sets `flag`.

**What the compiler is allowed to do**

- Hoist the load of `flag` out of the loop.
- Treat `flag` as constant if it sees no visible writes.
- Keep the value in a register and never re-read memory.

**Result**

- The loop may never see the update.
- The program can hang.

# Two Classic Failure Patterns

## 1) Load hoisted out of the loop

```
while (!flag) { /* empty */ } // flag is non-atomic
puts("go");
```

- The compiler can read `flag` once before the loop.
- The loop becomes infinite if the first read is 0.

## 2) Dead code elimination removes your signal

```
flag = 1;                          // writer
if (flag) do_something();          // same thread, no external
    use
```

- The compiler can remove or reorder these statements.
- Another thread cannot rely on this write without a happens-before edge.

## Key point

- Memory communication must be synchronized.
- Otherwise the compiler can reorder and the CPU can reorder.

# Summation (again)

```
#define N 100000000
long sum = 0;

void T_sum() { for (int i = 0; i < N; i++) sum++; }

int main() {
  create(T_sum);
  create(T_sum);
  join();
  printf("sum = %ld\n", sum);
}
```

**What if we enable compiler optimizations?**

- With `-O1` you may observe $N$.
- With `-O2` you may observe $2N$.
- Reordering, common subexpression elimination, or vectorization can change the outcome.

# What does the compiler do?

**Behavior of `T_sum`:** perform `n` increments on `sum`.

- Equivalent rewrite 1
  - `t = load(sum);`
  - `while (n--) t++;`
  - `store(sum, t);`
- Equivalent rewrite 2
  - `t = load(sum);`
  - `store(sum, t + n);`
- Optimizations assume determinism.
  - Without that assumption performance suffers.

# Controlling Compiler Optimizations

## Method 1: insert a “non-optimizable” block

```
while (!flag) {
  asm volatile ("" ::: "memory");
}
```

## Method 2: mark loads/stores as non-optimizable

```
int volatile flag;
while (!flag) ;
```

## They Are Not recommended for an OS course!

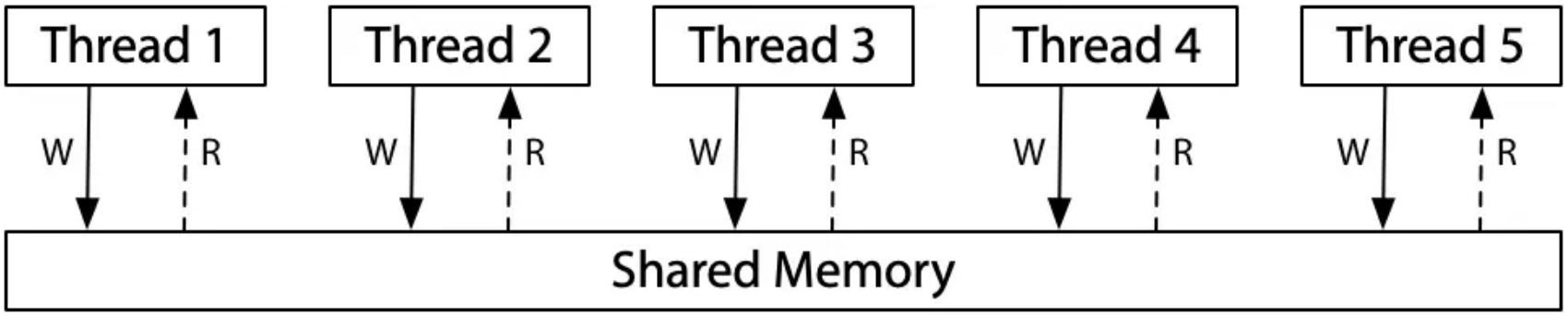
- Prefer proper synchronization primitives.
- Do not play with shared memory without atomics.

# Challenge III: Loss of Global Program Order

# State-Machine Model of a Concurrent Program

**State transition:** choose one thread and execute one instruction.

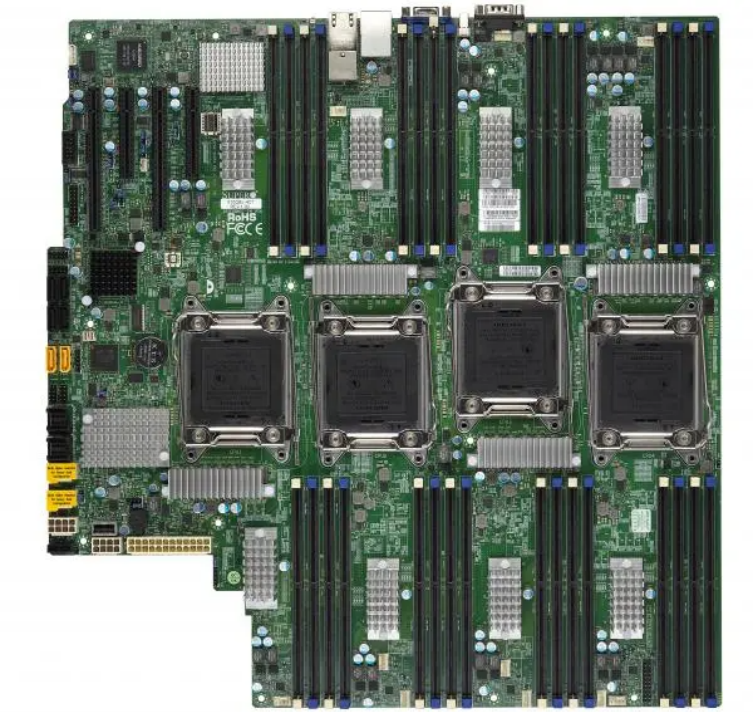
- Shared memory provides immediate writes and immediate reads.
- Therefore there is a single global order of instruction execution.



# But this is an oversimplified illusion

**Reading:** *Memory Models* by Russ Cox

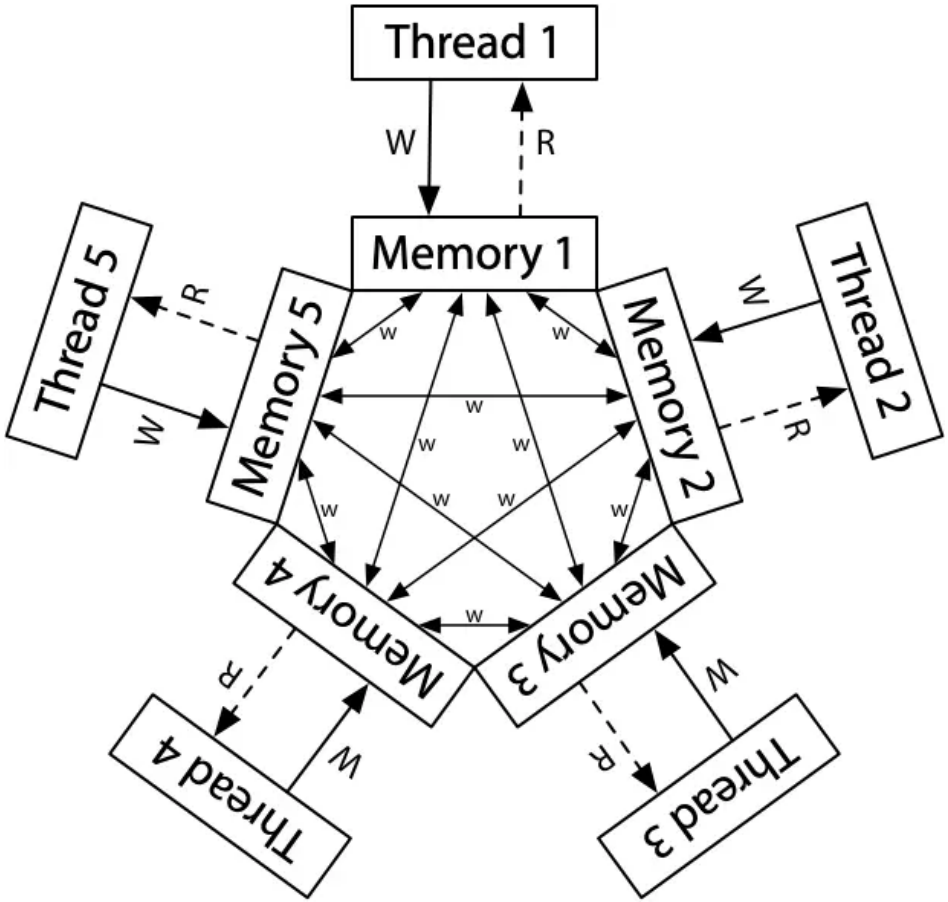
- Multiprocessor systems work hard to preserve this illusion.
- The illusion is unreliable in practice; think about sending data between planets.
- Non-Uniform Memory Access and even disaggregated memory are at the core.



# The Real State-Machine Model

**For performance: a relaxed memory model**

- A `store` writes to local memory (cache) first, then slowly propagates to other processors.
- A `load` may read an old value from its local memory (cache).



# A “Disordered” Real World

**Shared memory introduces disorder**

- The time when a `store` becomes visible to other processors can differ.

**Processors are disordered internally, too**

- For the same address, a `store`/`load` may bypass in hardware.
- Loads and stores to different addresses may be reordered.
- Out-of-order execution is a key feature of modern high-performance CPUs.
- In a sense the processor acts like a compiler.

# Observing the Effects of "Disorder"

```
int x = 0, y = 0;

void T1() {
  x = 1; int t = y;    // Store(x); Load(y)
  __sync_synchronize();
  printf("%d", t);
}

void T2() {
  y = 1; int t = x;    // Store(y); Load(x)
  __sync_synchronize();
  printf("%d", t);
}
```

Have A Try: mem-model

**The possible outputs:** `01`, `10`, `11`

- **In practice you may see:** `00` (surprising)
- Hardware and caches can delay visibility. Each load may read the old cached value.

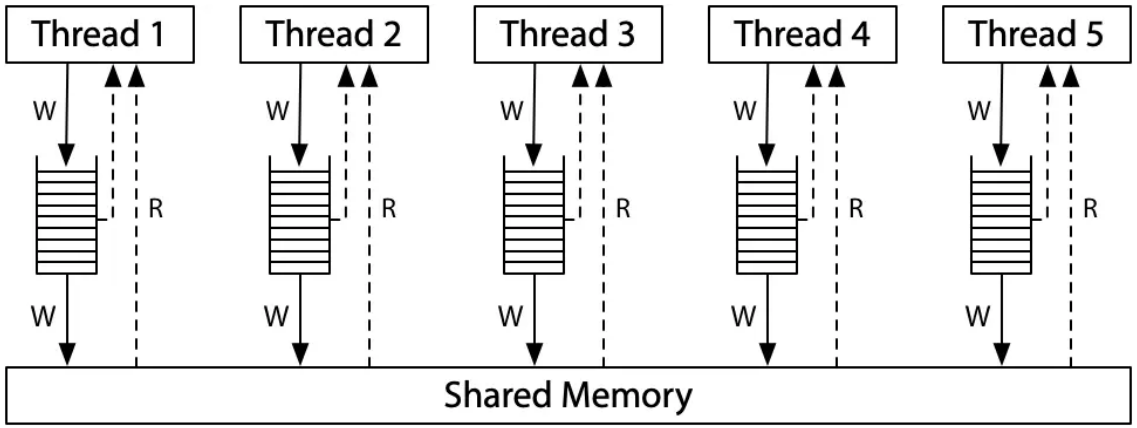
# Observing the Effects of "Disorder" (cont'd)

**CPU designers face a trade-off**

- A more ordered memory model is easier to program but can hurt performance.
- x86 provides a strong model (TSO). ARM and RISC-V are more relaxed.

**Implication: emulating x86 on ARM is hard**

- Emulators must insert fences and barriers to match x86 TSO.
- Some systems (Apple) add hardware assists to reduce the overhead.



# Shared Memory × Address Space

**Recall: the Translation Lookaside Buffer (TLB)**

- Caches mappings from virtual addresses to physical addresses.
- Every instruction fetch and memory access consults the TLB (including $\mathrm{M}[\mathrm{PC}]$, which is a virtual address).

**What if we change a region with `munmap`/`mprotect`?**

- Another thread may be running on another CPU.
- Its TLB may still hold stale translations.
- The OS must invalidate those entries on all CPUs: **TLB shootdown**.

# Takeaways

We can extend the state-machine model to shared-memory multithreading with little effort. At each step we choose one state machine to execute. With two APIs, create and join, we can use the shared-memory power of today's multiprocessor systems.

However, compiler optimizations are everywhere and CPUs act like compilers, so behavior under shared memory concurrency is complex. Humans think in physical time and tend to be "sequential creatures." Programming languages also build our intuition around sequence, selection, and iteration.

As a result, shared-memory concurrency is a challenging low-level craft.

In this Operating Systems course we do not encourage "playing with fire." We will introduce control techniques that let us avoid concurrency when needed, reduce concurrent programs back to sequential behavior, and make them understandable and controllable.