Lecture 14: Shared-Memory Concurrency

(Multithreading Model, Libraies, and Challenges)

Xin Liu

Florida State University
xliu15@fsu.edu

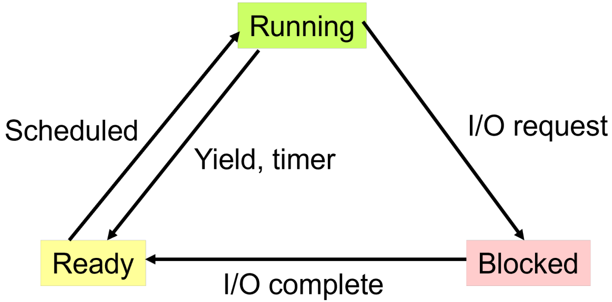
COP 4610 Operating Systems
https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

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?

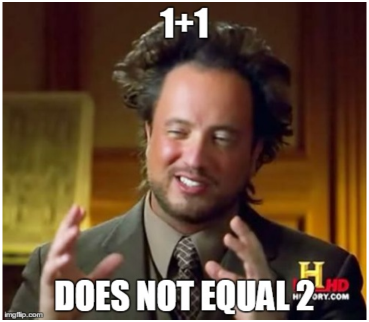
The UNIX process model, with its isolated address spaces, was a huge success. It provided a clean way to run multiple programs. However, a critical limitation emerged: A process has only one flow of execution.
What happens if this single flow blocks on a slow I/O operation like `read()`?

- 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)

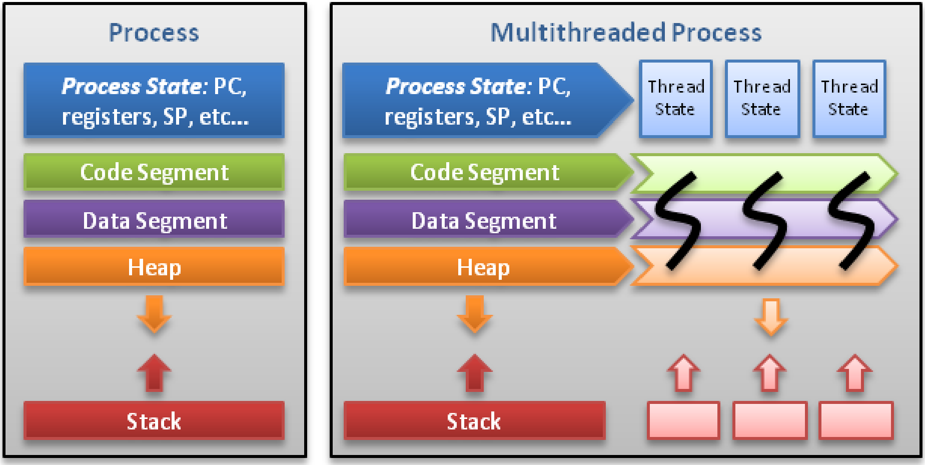
State-machine model of a multithreaded program

- 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.

One CPU Core: How Many Threads Run at a Time?

- Threads in one process share the **same address space** (code, data, heap).
- The address space contains **multiple stacks**, one per thread.

However, a single CPU core executes **only one instruction stream (thread)** at any instant.

- **Because**, they also all share one set of **core hardware resources**: program counter (PC), registers, pipeline, and execution units.
- These resources can only hold the state of *one* execution stream at a time, only one thread can physically run at that instant.

Concurrency vs. Parallelism

- The OS makes threads **take turns** by *context switching* (saving and restoring the PC, registers, and stack pointer) to create the illusion of concurrency.
- With one core you get *concurrency* (by switching). With two cores both threads can truly run at once (true *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)

On Linux, `pthread_create()` is built on `clone()`: `clone()` lets the caller supply the child's stack and select which resources to share (via `CLONE_*` flags), enabling thread-like or process-like semantics with a single primitive.

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.

```c
#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);
```

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.

Have A Try: thread-lib

Global Variables Are Shared Across Threads

- Let us verify this. Have A Try: share-memory

Threads Have Independent Stacks

If yes, what is the exact scope of a thread's stack?
Have A Try: independent-stack

Code Principle

- We create 4 threads (T1, T2, T3, T4).
- Each thread executes an infinite recursion function (`probe`).
- Every recursive call pushes a new stack frame onto its **own stack**, causing its stack to grow continuously.
- Each thread prints its own current stack size by tracking the highest and lowest memory addresses (high/low) it has touched.

Threads Have Independent Stacks

If yes, what is the exact scope of a thread's stack?
Have A Try: independent-stack

- **Concurrent Execution:** The output is interleaved, showing that the 4 threads are running concurrently and competing for CPU time.
- **Independent Stacks:** Each thread's stack size (T1, T2, T3, T4) grows **independently**. This proves they are completely separate, non-contiguous blocks of memory.
- **Stack Overflow:** When thread T3's stack grew to approx. 8185 KB ($\approx$ 8MB), the program crashed with a Segmentation fault.
- **Conclusion:**
    - By default on Linux, `pthread_create` allocates a fixed-size ($\approx$ 8MB) independent stack for **each thread**.
    - The OS places a special Guard Page at the end of this 8MB region (marked as non-readable/non-writable).
    - When a thread's stack exhausted its space and tried to write into this Guard Page, it triggered the segfault. This is the mechanism for detecting stack overflow.

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.

Have A Try: order

Loss of Determinism: Example

```
unsigned int balance = 100;

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

Have A Try: eBay

A "Check-Then-Act" Race

- We start with `balance = 100`.
- Two threads (T1 and T2) both try to withdraw $100.
- The function `eBay_withdraw()` has a critical flaw:
  1. **Check:** `if (balance >= amount)`
  2. **Act:** `balance -= amount;`

Exposing the Race with `usleep(1)`

- **Outcome 1 (No `usleep`): `balance = 0`**
  - The race is still possible, but *unlikely*.
  - One thread (e.g., T1) might win the race, running the *entire* block before T2 starts.
  - T1: checks ($100 \geq 100$) $\rightarrow$ True. T1: subtracts $\rightarrow$ `balance = 0`.
  - T2: checks ($0 \geq 100$) $\rightarrow$ False. T2: does nothing.

Exposing the Race with `usleep(1)`

- **Outcome 1 (No `usleep`): `balance = 0`**
  - The race is still possible, but *unlikely*.
  - One thread (e.g., T1) might win the race, running the *entire* block before T2 starts.
  - T1: checks ($100 \geq 100$) $\rightarrow$ True. T1: subtracts $\rightarrow$ `balance = 0`.
  - T2: checks ($0 \geq 100$) $\rightarrow$ False. T2: does nothing.
- **Outcome 2 (With `usleep`): `balance = 18446744073709551516`**
  - `usleep(1)` forces a context switch between the Check and the Act:
  - T1: checks ($100 \geq 100$) $\rightarrow$ True.
  - T1: calls `usleep(1)` and gets suspended.
  - T2: checks ($100 \geq 100$) $\rightarrow$ True (balance is still 100!).
  - T2: calls `usleep(1)` and gets suspended.
  - T1: wakes up, subtracts $\rightarrow$ `balance = 0`.
  - T2: wakes up, subtracts $\rightarrow$ `balance = 0 - 100`.

The ”Endless Money” Bug

- `balance` is an `unsigned long`.
- $0 - 100$ causes an integer underflow.
- The result is $2^{64} - 100$, which is the huge number you saw.

$$0 = 0xFFFFFFFFFFFFFF = 2^{64} = 18{,}446{,}744{,}073{,}709{,}551{,}616$$

$$0 - 100 = 2^{64} - 100 = 18{,}446{,}744{,}073{,}709{,}551{,}516$$

This perfectly matches your C program’s output!

The Similar Bugs in Read-World

- Bugs and vulnerabilities are no joke: The Mt. Gox Hack resulted in the loss of 650,000 BTC, worth approximately $28 billion.
- Blockchain system such as 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

1. Drop the item you want to duplicate on the ground, walk a few steps away.
2. Click to **pick it up** (start walking toward it).
3. **At the exact moment** the pickup happens, click a belt potion slot.
4. The game mis-binds the “just-picked-up” item ID to the potion slot, so the potion **turns into a copy** of that item.
5. Drop the “potion” on the ground — it appears as the duplicated item; pick both up and repeat.

Intuition: a timing race between the ground-pickup handler and the belt-click handler causes the belt slot to adopt the item being picked up (shared cursor/item ID updated in the same tick).

Diablo I Bug: 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;
}
```

- Shared variable `hoveredItem` is overwritten by a late click.
- Pickup then reads the new value, not the hovered one.
- Inventory and hand both get `"$99"`.

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` $\to$ `add` $\to$ `store`.
- The threads overwrite each other's updates. Have A Try: sum

Why `sum++` Loses Updates

Two threads (`T1`, `T2`) on shared `sum`: `sum++` = `load` → `add` → `store`.

- **Serial, OK**:

```
T1: load 0; add 1; store 1
T2: load 1; add 1; store 2   // final = 2
```

Why `sum++` Loses Updates

Two threads (`T1`, `T2`) on shared `sum`: `sum++` = `load` → `add` → `store`.

- **Serial, OK**:

```
T1: load 0; add 1; store 1
T2: load 1; add 1; store 2    // final = 2
```

- **Lost update** (both read old 0):

```
T1: load 0; add 1
T2: load 0; add 1
T1: store 1
T2: store 1 // final = 1  (one increment lost)
```

Why `sum++` Loses Updates

Two threads (`T1`, `T2`) on shared `sum`: `sum++` = `load` → `add` → `store`.

- **Serial, OK**:

```
T1: load 0; add 1; store 1
T2: load 1; add 1; store 2   // final = 2
```

- **Lost update** (both read old 0):

```
T1: load 0; add 1
T2: load 0; add 1
T1: store 1
T2: store 1 // final = 1  (one increment lost)
```

- **Interleaved store overwrites**:

```
T1: load 0; add 1
T2: load 0; add 1; store 1
T1: store 1 // final = 1 (one increment lost)
```

Many more interleavings

`load/add/store` from different threads happen in any order.
`load/add/store` from different threads overlap and overwrite.

Even One Assembly Instruction Fails

Idea: Maybe we can make `sum++` atomic by using one assembly instruction?

```
asm volatile("incq %0" : "+m"(sum));
```

- `asm volatile` only prevents compiler reordering or removal.
- `incq %0` still performs a **read-modify-write** on memory.
- Without the `lock` prefix, CPUs can interleave updates.
- Two threads can both load the same old value and overwrite each other.
- Correct version needs hardware atomicity, e.g.:

```
asm volatile("lock incq %0" : "+m"(sum));
```

- Or better: use C11 atomics or `__sync_fetch_and_add()`.

Have A Try: sum. You can use `objdump -d` to find the assembly instruction.

Consequences of Losing Determinism

Run three `T_sum` threads in parallel. What is the minimum final `sum`?

- 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`

- **Model checker:** Have A Try: sum-model
- 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 2*N*.
- 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!

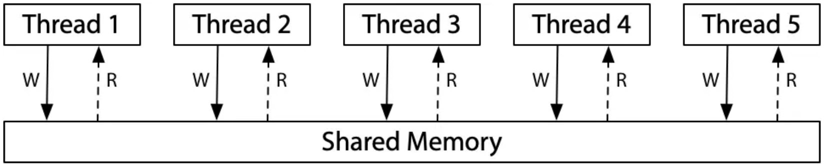
- Prefer proper synchronization primitives.
- Do not play with shared memory without atomics.

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

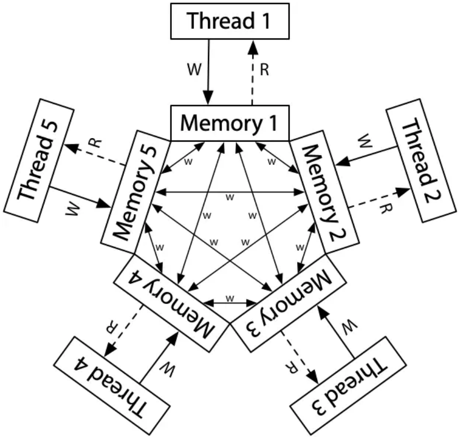
Reading: *Memory Models* by Russ Cox

- 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.

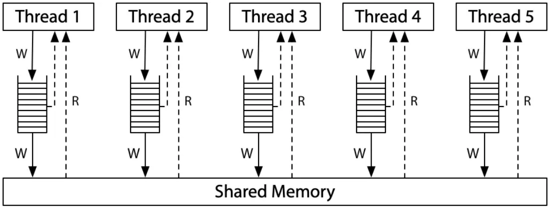
Observing the Effects of “Disorder” (cont’d)

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 $\times$ 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.