Lecture 14: Shared-Memory Concurrency

(Threads, Compilers, and Memory Models)

Xin Liu

Florida State University
xliu15@fsu.edu

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

Why Threads

After UNIX provided basic system calls for processes, address spaces, and object access, adoption surged. New needs appeared. A process blocked on `read()` could still do other work. Hardware gained multiple CPUs. Process-level parallelism felt limited. We needed multiple execution flows that share memory. Threads were introduced.

- Multithreading Model
- Libraries
- Challenges

Shared-Memory Threading Model and Thread Libraries

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 a special system call: `spawn()`
  - Add one more “state machine” with its own stack but shared global variables
- State transition: choose one state machine and execute one statement (instruction)
- Let’s do it with a model checker

Concurrency vs. Parallelism

Concurrency

- “Simultaneous” execution in a logical sense.
- Can be interleaved by the OS or runtime using time slicing.
- May also be truly simultaneous on some hardware.

Parallelism

- Truly simultaneous execution in the strict sense.
- Requires multiple processors that share memory.
- Instructions run at the same time and can load or store to shared memory.

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

Multiprocessor 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.

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?

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 lower.

Deterministic State Transitions

Loss of Determinism

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

- Except for system calls, program behavior is deterministic.
- With the same initial state (`argv`, `envp`) and the same syscalls, repeated runs yield the same result.

Concurrency breaks this

- Each step may nondeterministically choose a thread to run.
- A load may observe another thread's store, or it may not.
- Nondeterministic programs are hard to reason about.

Loss of Determinism: Example

```
unsigned int balance = 100;

int T_alipay_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?
- The account can overspend due to a race condition.
- Real systems have suffered large losses from bugs like this.

A Real Example: Diablo I (1996)

Picking up an item at the same instant you grab 1 gold

- A race between pickup and state update creates duplication.
- Different threads or components observe inconsistent state.
- A simple concurrency bug turns into a player exploit.

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);
}
```

- What result will we get?

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);
  }
}
```

Answer: the minimum possible final value is **3**.

- Wave 1: all threads do `load(sum)` seeing 0, then three `store` writes of 1 → `sum = 1`.
- Wave 2: all threads load 1, then three stores write 2 → `sum = 2`.
- Wave 3: all threads load 2, then three stores write 3 → `sum = 3`.

What is the answer?

Model checker: `sum = 2`

- Not 1, because the loop runs three times.
- Trace recovery is NP-complete.

Value of a mathematical view

- Nondeterminism is fundamentally hard for humans.
- Proof is the way to resolve it.
  - $\forall$ thread schedules, the program satisfies property P.

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. The classic `fork` example shows why buffering matters.
- Two threads doing `buf[pos++] = ch` at the same time is dangerous.
- See `man 3 printf`.

Deterministic State Transitions

Let's Look at the Compiler

Virtualization: a process only sees itself and the OS

- Determinism: except for system calls, no one can interfere with the program state.

Compiler: optimizes as if only system calls are observable

- Statements do not have to execute in the written order.
- Typical optimization: dead code elimination.
- As long as behavior at system call boundaries is preserved, the compiler may reorder or remove code.

But this clashes with nondeterminism

- A load may read a value written by another thread.
- If code assumes single-thread semantics, optimizations can break correctness.

A “Clever” Example

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

- This seems to implement waiting for a thread.
- “Wait until another thread raises the flag, then I continue?”

A “Clever” Example, but the compiler is smarter

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

- This looks like waiting until another thread sets `flag`.
- In a sequential view the compiler may optimize it.
  - It may hoist the load of `flag` out of the loop or assume it never changes.
  - This is a common concurrency bug pattern.
- Without atomics or proper synchronization the write may never become visible.
- Use `std::atomic<bool>` with acquire–release, a condition variable, or pthread mutex/cond.
- See “Ad hoc synchronization considered harmful.”

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.
- Use atomics or proper synchronization if you need correctness.

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) ;
```

Not recommended for an OS course

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

Deterministic State Transitions

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.

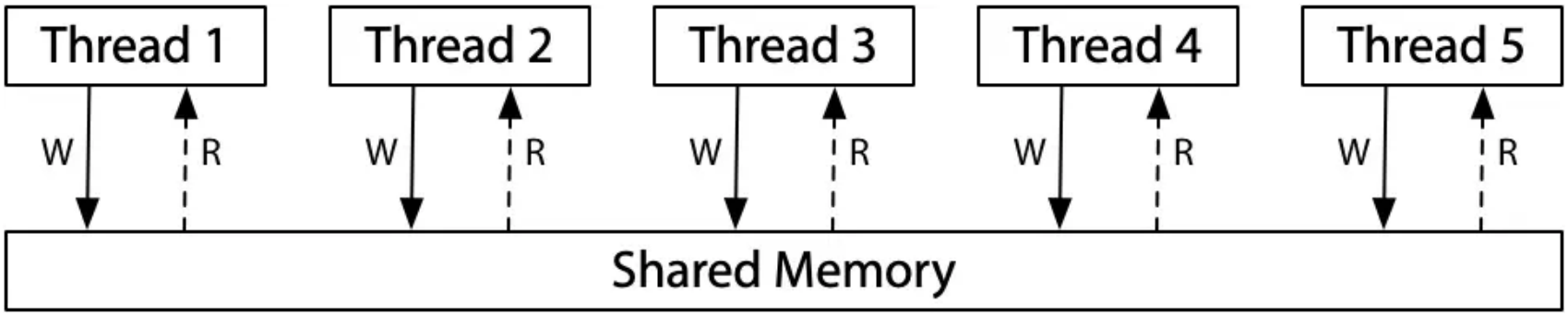
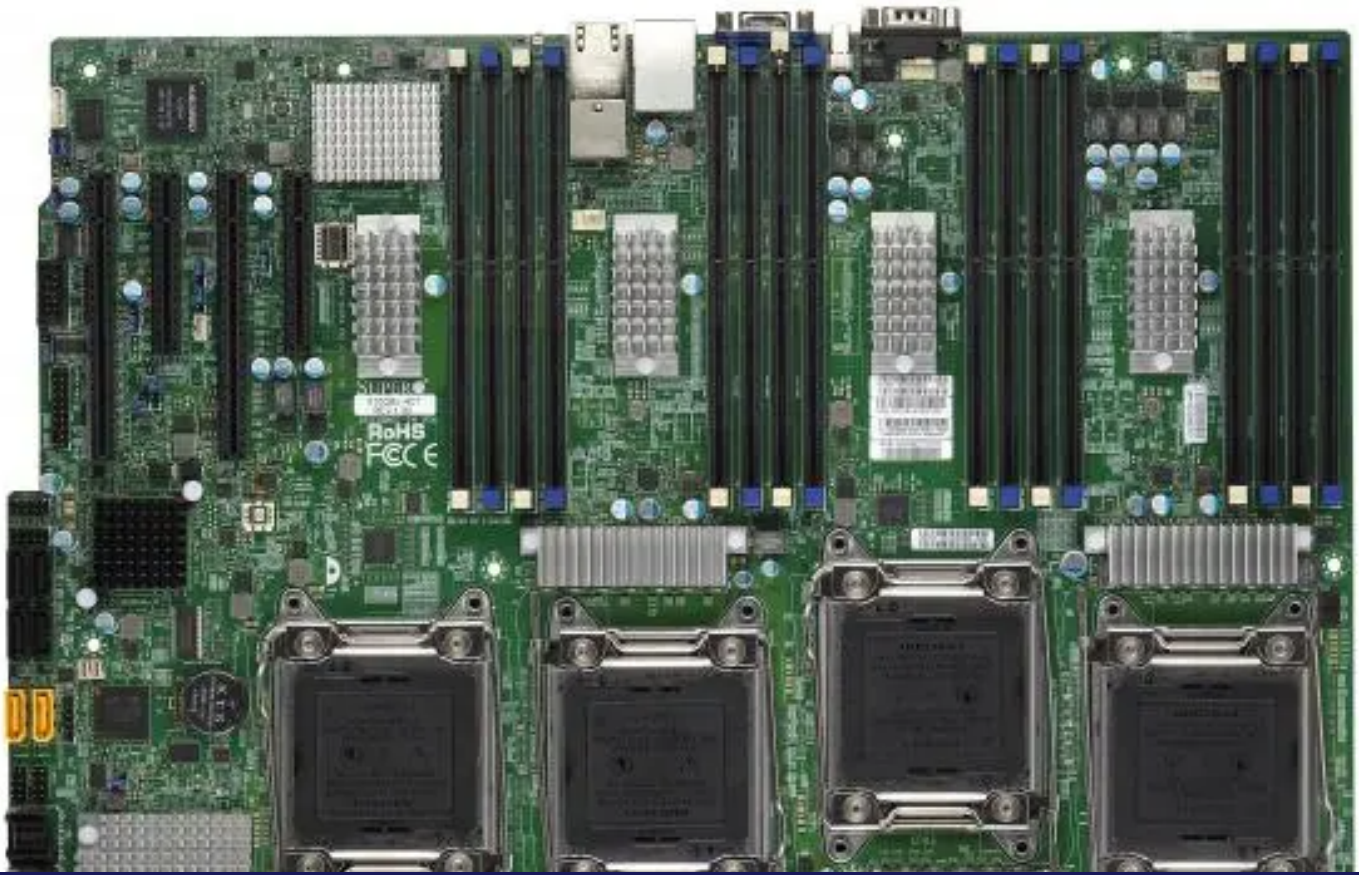


Figure: One step picks a thread and executes one instruction; reads/writes are immediately visible.

But this is an oversimplified illusion

Reading: *Memory Models* by Russ Cox

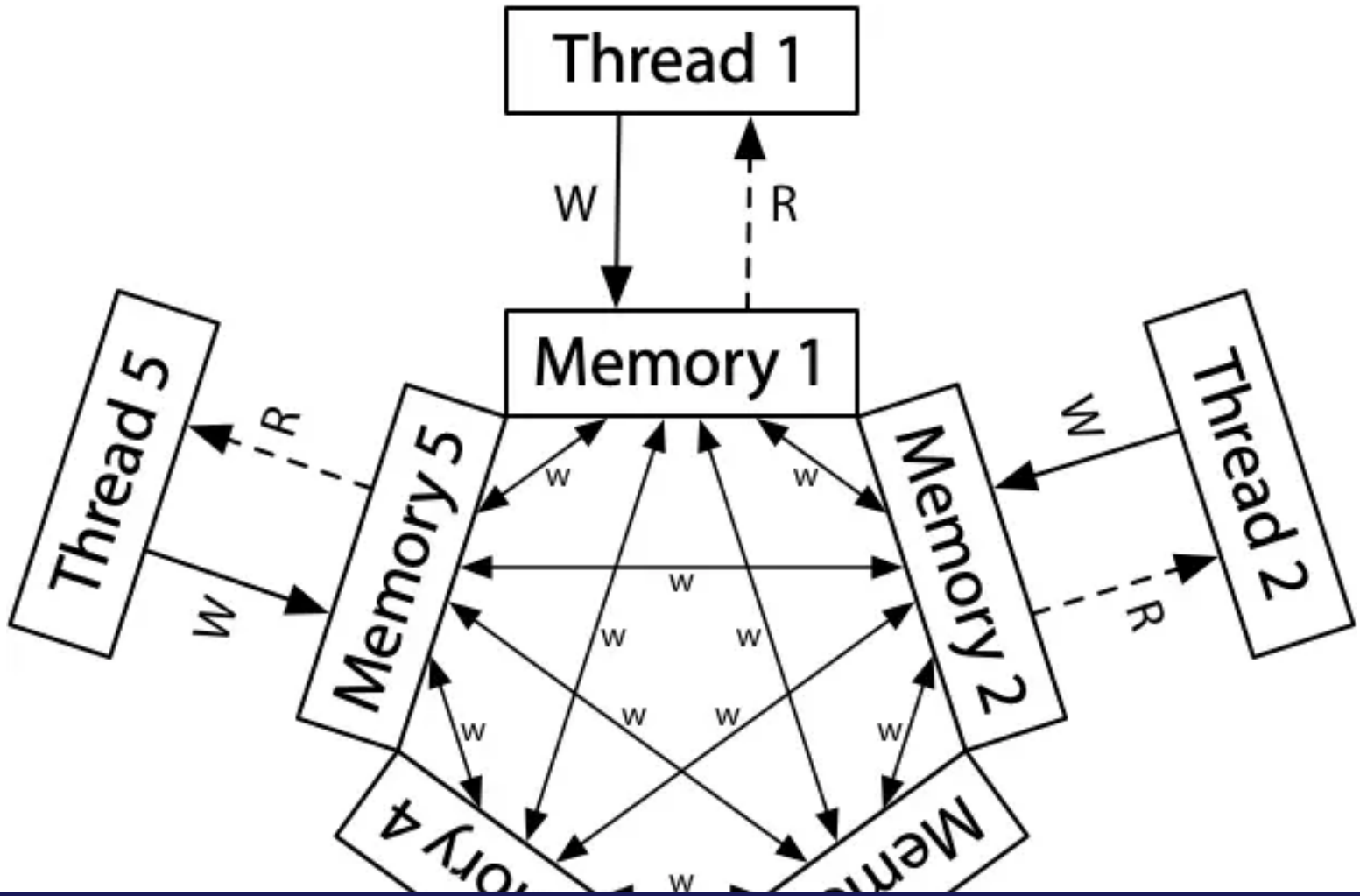
- Multiprocessor systems work hard to preserve this illusion.
- The illusion is unreliable in practice; think about sending data between planets.
- Non-Uniform Memory Access and even disaggregated memory are at the core.



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);
}
```

Model checker says 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.
- Memory fences and atomics are needed if you require ordering guarantees

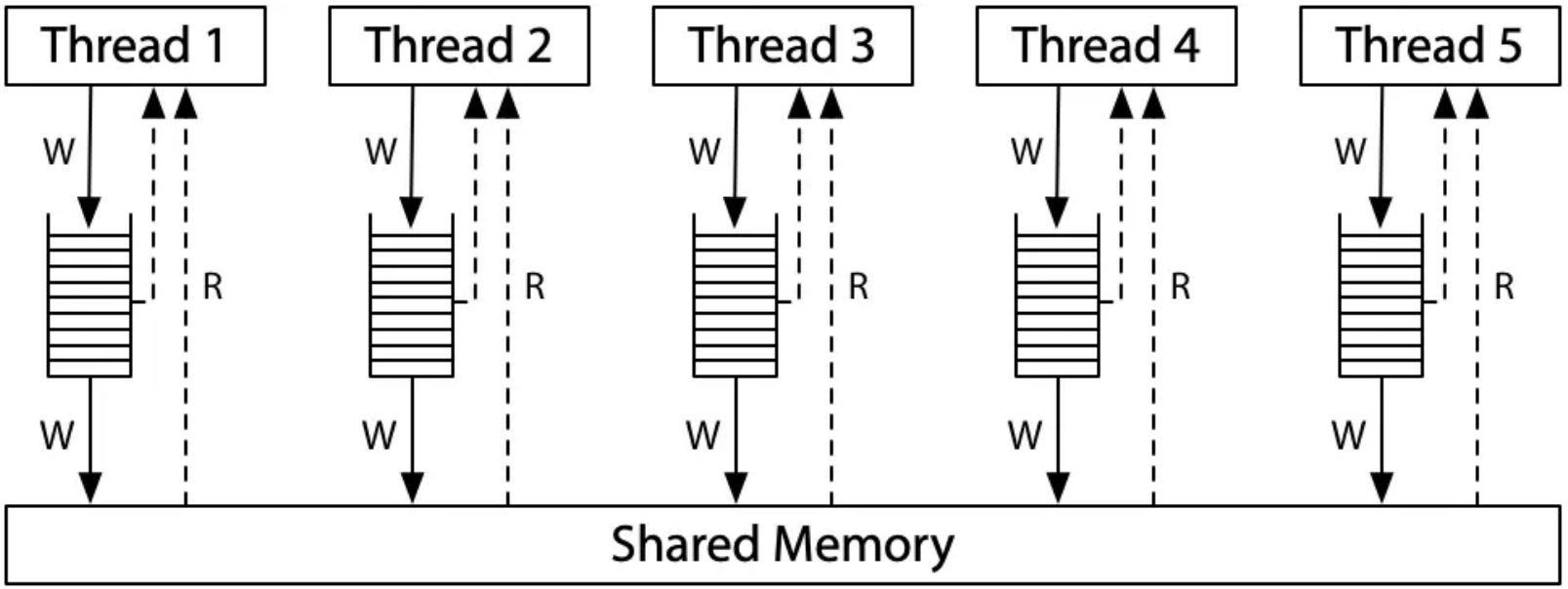
Observing the Effects of "Disorder" (cont'd)

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 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**.

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, spawn 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.