Sistemi Operativi

Corso di Laurea in Informatica

Sapienza Università di Roma

a.a. 2026/2027

Marco Raoul Marini

marini@di.uniroma1.it

Gabriele Tolomei

tolomei@di.uniroma1.it

Lecture 3.1: Concurrency I

Threads and Shared Memory

Opening Thought

> *"... each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data."*
> *— Remzi & Andrea Arpaci-Dusseau, OSTEP Ch. 26*

Module 02 (Virtualization) gave every process the illusion of a private CPU and a private address space. **Module 03** asks a new question: what happens when several execution flows deliberately share *one* address space at the same time?

The answer leads to **concurrency** — where the difficulty is not in understanding individual instructions, but in understanding their interactions across time.

Lecture Roadmap

This lecture is organized into **7 major sections**:

1. **From Processes to Threads** — Why share an address space
2. **What Is a Thread?** — Thread vs. process, private vs. shared state, thread states
3. **Creating and Managing Threads: the `pthread` API** — `create`, `join`, arguments
4. **The Shared Memory Problem** — Why `counter++` is broken; race condition, atomicity, critical section
5. **The Critical Section and Mutual Exclusion** — Mutual exclusion, the lock abstraction
6. **The Mutex API** — `pthread_mutex_t`, using mutexes correctly
7. **Summary & Hand-off** — Key ideas, open question for Lecture 3.2

> *Extended material is in the **Appendix** (A.1–A.8): each section divider points to its appendix counterpart.*

Prerequisite Check

To follow this lecture comfortably, ensure you recall:

- **Processes and PCB** (Lecture 2.1): state, address space layout, context switch.
- `fork()` **and** `exec()` (Lecture 2.1): creating new processes in UNIX.
- **Address space layout** (Lecture 2.5): code, data, heap, stack.
- **Context switching** (Lecture 2.2): saving/restoring PC, SP, registers.
- **C programming basics** (Lectures 1.3–1.4): pointers, structs, `malloc`, global vs. local variables.
- **Scheduling** (Lectures 2.2–2.3): READY, RUNNING, BLOCKED states; timer-interrupt preemption.

New concepts introduced today: thread, race condition, data race, atomicity, critical section, mutual exclusion, lock (mutex).

§ 1 — From Processes to Threads

Goal: see why we want several flows of control inside one address space.

📎 *Extended material → Appendix A.1*

Why Threads? Two Goals

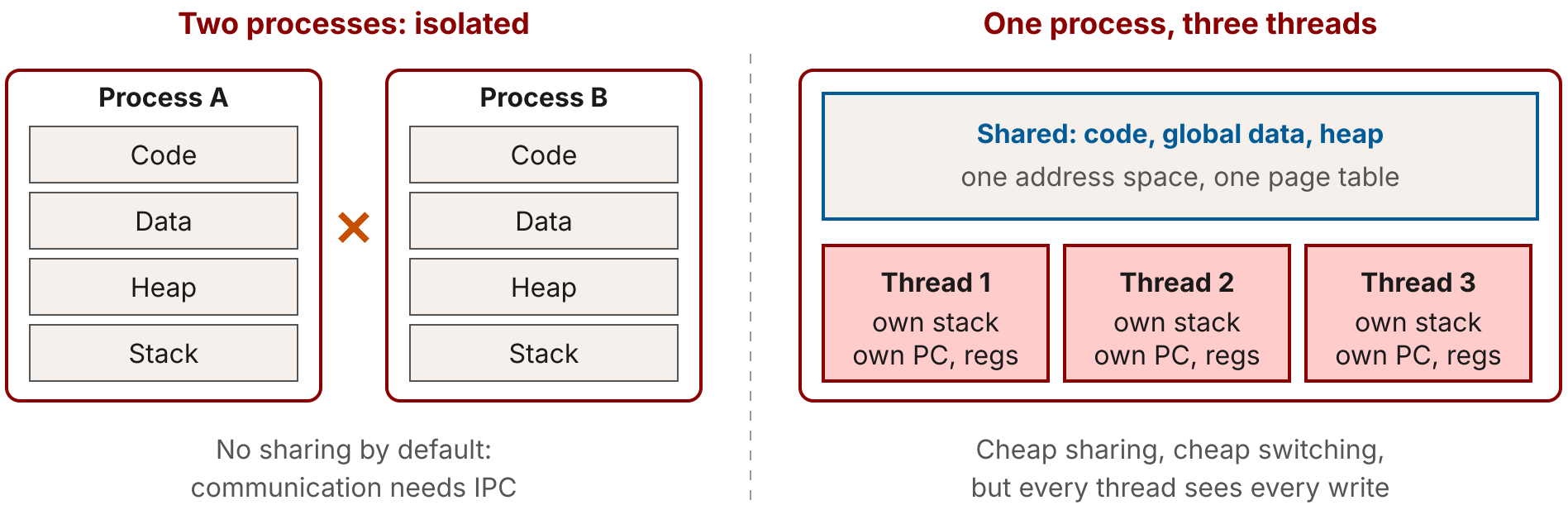
> ***So far:*** *a process is one flow of control in a private address space. Can one program do several things at once?*

- **Parallelism:** split independent work across cores. N equal chunks on N cores can finish up to $N\times$ faster.
- **I/O overlap:** while one flow waits for disk or network, another keeps the CPU busy. If a flow computes a fraction f of its time, about $1/f$ flows keep one core busy (e.g. $f = 0.1 \Rightarrow$ ~10 flows).

Both goals need several flows that **work on the same data**. Could we simply use several processes (`fork()`)?

Processes Isolate, Threads Share

Problem: *processes are isolated, so cooperating means IPC (pipes, sockets, shared-memory segments) and costly context switches (page-table switch, cold TLB and caches).*
Idea: *keep **one** address space; give each flow only its own stack and CPU state.*



✅ Checkpoint 1 — From Processes to Threads

Test your understanding:

1. Name the two motivations for using several threads in one program.
2. Why is `fork()` a poor solution for parallelism within a single program? Name three specific costs.
3. What must a programmer do to share memory between two *processes*? Why is this unnecessary between two *threads*?
4. A thread spends 90% of its time blocked on disk I/O and 10% computing. Roughly how many such threads are needed to keep one core 100% busy (ideal case)?

§ 2 — What Is a Thread?

Goal: state precisely what a thread owns privately and what it shares.

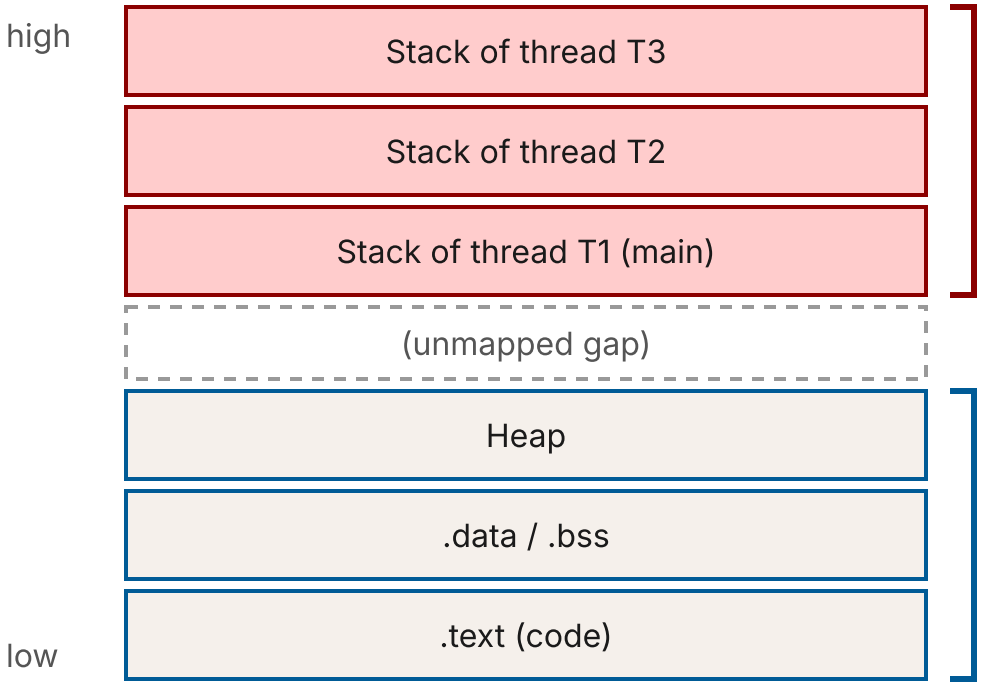
📎 *Extended material → Appendix A.2*

What Is a Thread? Thread vs. Process

> ***A thread is the smallest unit of execution scheduled by the OS. It lives inside a process, shares its address space and resources, and has its own PC, registers and stack.*** *A single-threaded process is the special case of one thread; the OS schedules threads (Linux: 1:1, see A.2).*

Property	Process	Thread
Address space, code, heap, open files	Private (own page table)	Shared by all threads
Stack	One	One **per thread**
Registers / PC	Saved in PCB	Saved in TCB
Creation / context switch	High (`fork` + COW; page-table switch, TLB/cache effects)	Low (stack + TCB; register save/restore)
Communication	IPC (pipes, sockets, shmem)	Direct: shared memory
Failure isolation	A crash stays local	One crashing thread can kill the process

Private vs. Shared: One Address Space, Many Stacks



PRIVATE to each thread

Local variables (e.g. loop index i), call frames

CPU state: PC, SP, registers (saved in the TCB)

SHARED by all threads

Globals (e.g. int counter), static data

Heap data (p = malloc(...)), code

Open files, signal handlers, working dir

The Thread Control Block (TCB)

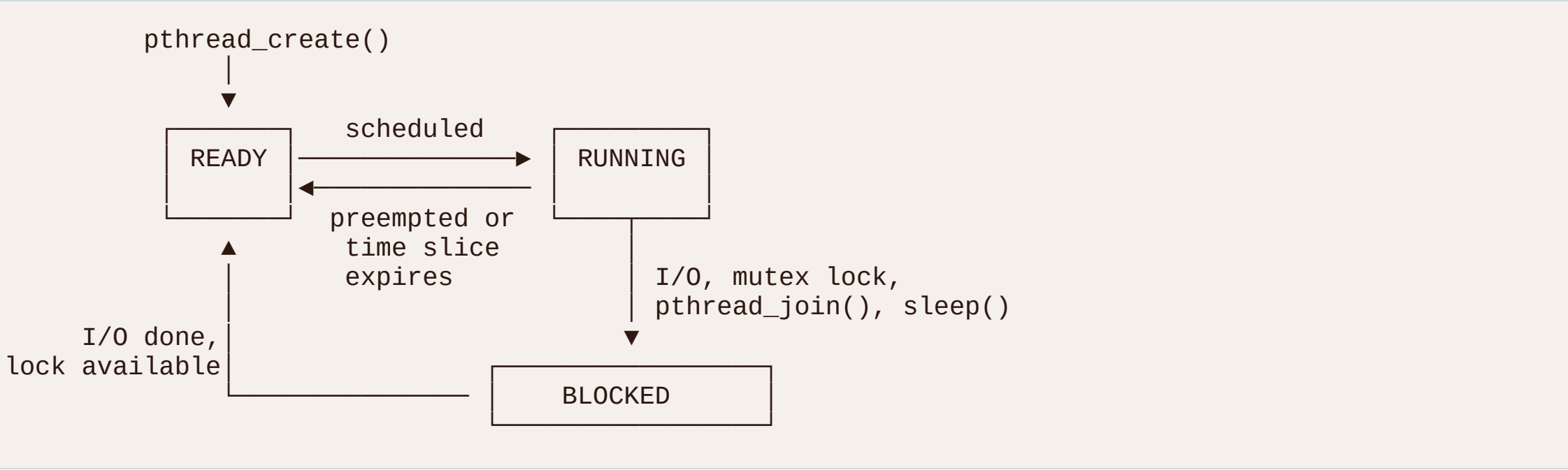
> *The OS schedules threads individually, so it needs a per-thread record, just as the PCB serves processes.*

```
// Simplified TCB structure:
struct thread_control_block {
    int          tid;          // Thread ID
    enum state   state;        // RUNNING, READY, BLOCKED
    uint64_t     pc;           // Saved program counter
    uint64_t     sp;           // Saved stack pointer
    uint64_t     regs[16];     // Saved general-purpose registers
    uint64_t     stack_base;   // Bottom of this thread's stack
    uint64_t     stack_size;   // Size of this thread's stack
    void        *ret_value;    // Value returned / passed to pthread_exit()
    struct pcb  *process;      // Pointer to the owning process's PCB
};
```

The TCB holds only per-thread execution state; the address-space description stays in the (shared) PCB.

Thread States: Running, Ready, Blocked

Threads go through the same lifecycle states as processes:



When a thread calls `pthread_mutex_lock()` on a held lock, it moves to BLOCKED, freeing the CPU for other threads. When the lock is released, the blocked thread moves back to READY.

✅ Checkpoint 2 — What Is a Thread?

Test your understanding:

1. List three things a thread has **privately** and three things it **shares** with all other threads in its process.
2. Why is context-switching between two threads of the **same** process cheaper than switching between two **different** processes?
3. Draw the multi-threaded address space for a process with 2 threads. Label all shared and private regions.
4. What is a TCB? How does it differ from a PCB?
5. Four equal-priority threads of process P and one thread of process Q are all READY on a 4-core Linux machine (threads are scheduled individually). Roughly what share of the CPU does each process receive?

§ 3 — Creating and Managing Threads: the `pthread` API

Goal: write a program that creates threads, passes them arguments, and waits for them.

📎 *Extended material → Appendix A.3*

`pthread_create()`: Creating a Thread

```
gcc -pthread -o program program.c      # compile and link; code: #include <pthread.h>
```

```
int pthread_create(
    pthread_t           *thread,    // output: thread ID
    const pthread_attr_t *attr,    // NULL for default attributes
    void *(*start_routine)(void*), // function the thread will run
    void                *arg       // argument passed to the function
);                                 // returns 0 on success, error code on failure
```

- The new thread starts `start_routine(arg)` as soon as the scheduler runs it (it begins READY, like any new thread).
- `thread` receives the new thread's ID, used later by `join()`; `attr = NULL` means defaults.
- The `pthreads` API also offers mutexes (§ 6), condition variables, barriers (later lectures).

First Example: Creating a Thread

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

void *thread_function(void *arg) {
    int id = *(int *)arg;
    printf("Thread %d: hello from a new thread!\n", id);
    return NULL;
}

int main(void) {
    pthread_t tid;
    int arg = 1;      // lives on main's stack: valid because main joins below

    pthread_create(&tid, NULL, thread_function, &arg);

    // Main thread continues here concurrently with the new thread
    printf("Main thread: created thread %lu\n", (unsigned long)tid);

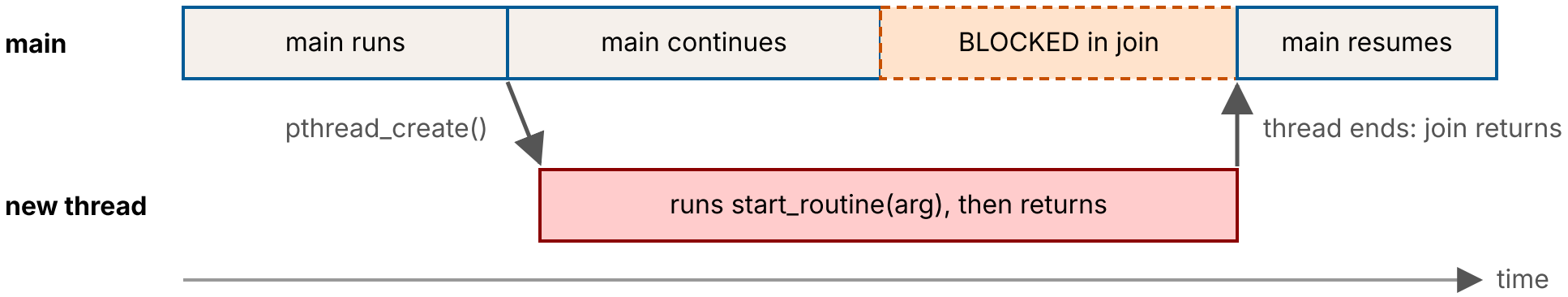
    pthread_join(tid, NULL);   // wait for the thread to finish
    printf("Main thread: thread has finished.\n");
    return 0;
}
```

Output order is **non-deterministic**: the "hello" line may appear before or after "created thread".

`pthread_join()`: Waiting for a Thread

```
int pthread_join(pthread_t thread, void **retval);  // blocks until `thread` ends; 0 on success
```

`pthread_join()` is the thread analogue of `waitpid()`. If `retval` is not `NULL`, it receives the value the thread returned (or passed to `pthread_exit()`). Join every thread: if `main` returns first, the whole process ends and all threads are killed.



Passing Arguments to Threads

> *The new thread runs **later**: its argument must still be valid and unchanged by then.*

```c
pthread_t tids[4];
// WRONG: passing the address of a loop variable
for (int i = 0; i < 4; i++) {
    pthread_create(&tids[i], NULL, func, &i);  // ⚠️ main keeps modifying i!
}

// CORRECT: pass the value itself (cast through void*)
for (long i = 0; i < 4; i++) {
    pthread_create(&tids[i], NULL, func, (void *)i);
    // In the thread: long id = (long)arg;
}
// ALSO CORRECT: a unique heap object per thread
for (int i = 0; i < 4; i++) {
    int *arg = malloc(sizeof(int));
    *arg = i;
    pthread_create(&tids[i], NULL, func, arg);   // thread must free(arg)
}
```

In the WRONG version main and the threads access the same `i` concurrently: a first taste of the problem of § 4.

✅ Checkpoint 3 — Creating and Managing Threads

Test your understanding:

1. Write the correct `pthread_create()` call to start a thread that runs `void *compute(void *arg)` with argument `42` (passed by value).
2. What is the risk of passing `&i` from a `for` loop to `pthread_create()`? Show how to fix it.
3. What does `pthread_join()` do? What happens to the other threads if `main` returns without joining them?
4. Why is the output order of the first example non-deterministic?

§ 4 — The Shared Memory Problem

Goal: explain exactly how two threads running `counter++` can lose an update.

📎 *Extended material → Appendix A.4, A.5*

A Simple Shared Counter: Two Threads Incrementing

```c
#include <stdio.h>
#include <pthread.h>
int counter = 0;   // global shared variable

void *increment(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        counter++;
    }
    return NULL;
}
int main(void) {
    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("Final counter value: %d\n", counter);
}
```

Now we use sharing: two threads update one global. Expected: `2000000`. Actual: **different on every run, almost always less** (compiled without optimization, `-O0`; see the next slides).

Disassembling `counter++`: Three Machine Instructions

> ***Why is the result wrong?*** *Look below the C level.*

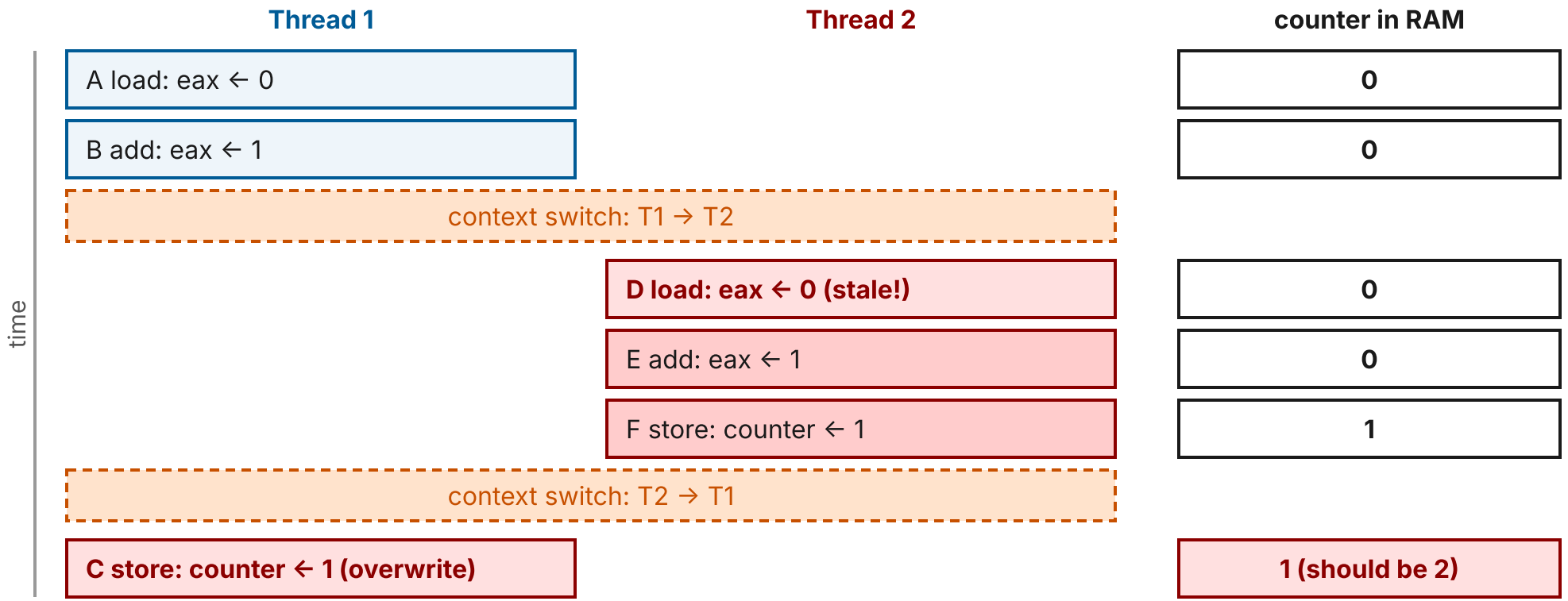
`counter++` looks atomic but is **not**. Unoptimized x86-64 code:

```
mov    eax, [counter]    ; (1) LOAD:  read counter from memory into a register
add    eax, 1            ; (2) ADD:   increment the register value
mov    [counter], eax    ; (3) STORE: write new value back to memory
```

- The three steps are **not atomic** as a group: the OS can preempt a thread between any two of them (the register `eax` is saved per thread).
- On a multi-core machine no preemption is even needed: two cores can interleave these steps truly in parallel.
- Even a single `add [counter], 1` instruction is not atomic across cores: the hardware still performs a separate read and write.

Interleaving: One Lost Update

Initially `counter = 0`; each thread executes load (A/D), add (B/E), store (C/F).



Both increments ran, but T1's stale value overwrote T2's: one increment LOST.

The Race Condition Defined

> *A race condition occurs when the correctness of a program depends on the relative timing or interleaving of multiple threads, and at least one possible interleaving produces an incorrect result.*

Properties of race conditions:

- **Non-deterministic:** it appears only if the scheduler switches at a bad moment. Rare per increment, near-certain over millions (A.4.1).
- **A "Heisenbug":** observing it changes timing and can hide it (adding `printf`, using a debugger, or changing the optimization level: here `-O2` folds the loop and hides the bug).
- **Dangerous:** data corruption, security holes, crashes.

A **data race** (unsynchronized conflicting accesses, undefined behavior in C) is the usual low-level cause; the two notions differ (A.5.1).

In the counter example the result depends on whether T2 loads `counter` before or after T1 stores it — decided entirely by the scheduler.

Atomicity and the Critical Section

> ***Atomicity:*** *an operation is atomic if it appears to execute as a single, indivisible step — no other thread can observe it half-done.*

> ***Critical section:*** *a region of code that accesses shared data and must execute atomically with respect to other threads. In our program:* `counter++` *.*

	What we want	What we have
`counter++`	one atomic step: the new value is visible, or the old one	three instructions; any boundary is a possible switch point
Consequence	no lost updates	T2 can load `counter` between T1's LOAD and STORE

We need a way to make the critical section behave atomically. This is what **locks** provide (§ 5).

✅ Checkpoint 4 — The Shared Memory Problem

Test your understanding:

1. `counter++` compiles to three machine instructions. List them and explain why this makes `counter++` unsafe in a multi-threaded context.
2. Draw a timeline showing the specific interleaving of Thread 1 and Thread 2 that causes the counter to be incremented only once instead of twice. Label each instruction.
3. Why does the counter bug manifest **sometimes** but not always? Why is such a bug called a Heisenbug?
4. Define **critical section**. Identify the critical section in the counter example.
5. What does **atomicity** mean? Why doesn't C guarantee that `counter++` is atomic?

§ 5 — The Critical Section and Mutual Exclusion

Goal: state what any solution to the critical-section problem must guarantee, and meet the lock abstraction.

📎 *Extended material → Appendix A.6*

Formalizing the Critical Section Problem

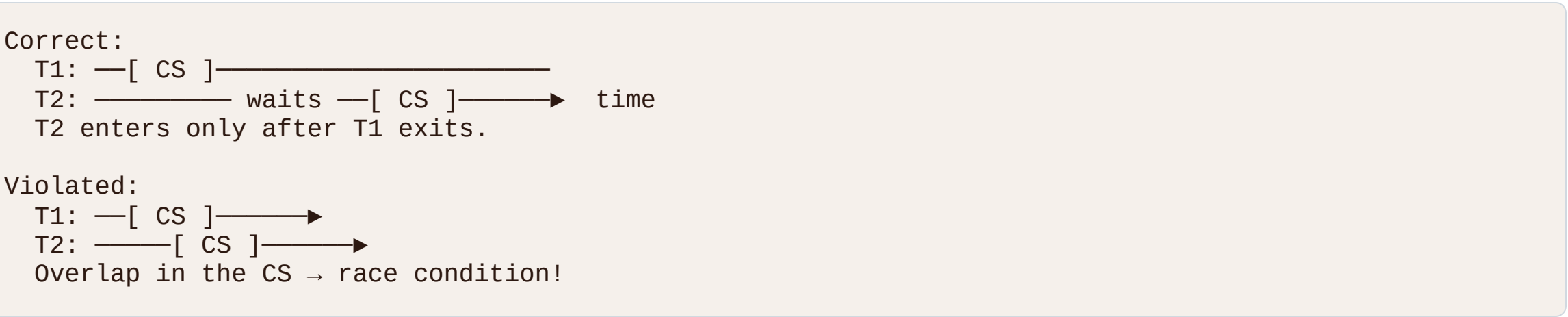
> ***Problem:*** *the critical section (§ 4) must run atomically.* ***Idea:*** *wrap it in a protocol that every thread follows.*

```
while (true) {
    entry_section      // request permission to enter
    critical_section   // access shared data
    exit_section       // signal departure
    remainder_section  // everything else (non-critical work)
}
```

A correct entry/exit protocol must guarantee **mutual exclusion** (next slide). Mutual exclusion alone is not enough: a protocol that lets *nobody* in satisfies it trivially but is useless. Hence two more requirements, **progress** and **bounded waiting** (A.6.1; Lecture 3.2 builds on them).

Mutual Exclusion

> ***Mutual Exclusion:*** *if thread T_i is executing in its critical section, then no other thread T_j is executing in its critical section.*



This is the most fundamental requirement: without it, all the races of § 4 can occur.

The Lock Abstraction

> ***Idea:*** *package the entry and exit sections into an object, the* ***lock****.*

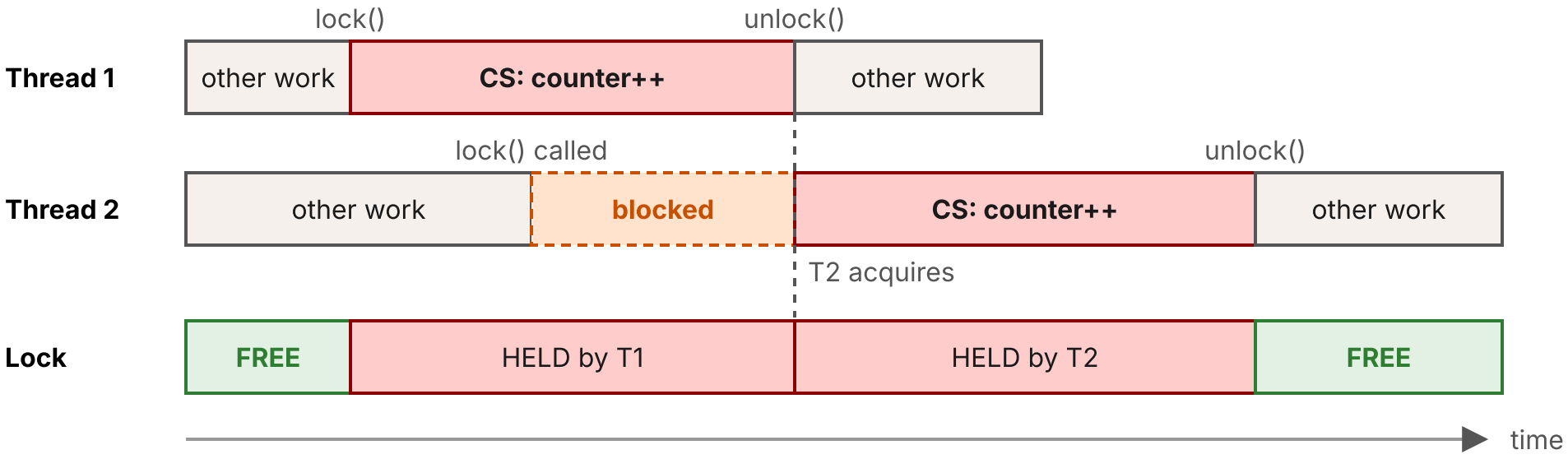
```
typedef struct { ... } lock_t;

void lock_init(lock_t *l);   // initialize to FREE
void lock(lock_t *l);        // entry section: acquire (may wait)
void unlock(lock_t *l);      // exit section: release
```

- `lock()`: if FREE, acquire it (now HELD by this thread); if HELD, wait until FREE, then acquire.
- `unlock()`: release it; one waiting thread (if any) acquires it.
- At most **one** holder at any time ⇒ mutual exclusion. *How* to wait (spin or sleep) is up to the implementation.

```
lock(&mylock);     // acquire — only one thread proceeds
counter++;         // CRITICAL SECTION — now safe
unlock(&mylock);   // release — next waiter can proceed
```

Lock Timeline: Two Threads, One Lock



The critical sections never overlap: T2 waits at `lock()` until T1 calls `unlock()`.

✅ Checkpoint 5 — The Critical Section and Mutual Exclusion

Test your understanding:

1. Define **critical section** and **mutual exclusion**. Where are the entry and exit sections in the counter program with a lock?
2. A "lock" that never grants entry to anybody satisfies mutual exclusion. Why is it still useless?
3. Draw the timeline when T2 calls `lock()` while T1 holds the lock. If the mutex puts T2 to sleep, what state is T2 in (§ 2) meanwhile?
4. Does a lock make `counter++` atomic for a thread that does *not* call `lock()`?

§ 6 — The Mutex API

Goal: protect shared data with `pthread_mutex_t` and avoid the common misuses.

📎 *Extended material → Appendix A.7*

Mutual Exclusion: `pthread_mutex_t`

`pthread_mutex_t` is the POSIX implementation of the lock abstraction of § 5.

```c
#include <pthread.h>

// Static initialization:
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

// Dynamic initialization (e.g. for a heap-allocated mutex):
pthread_mutex_t *pm = malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(pm, NULL);       // NULL = default attributes

pthread_mutex_lock(&m);             // acquire — blocks if held by another thread
pthread_mutex_unlock(&m);           // release

int rc = pthread_mutex_trylock(&m); // non-blocking:
// rc == 0: acquired.  rc == EBUSY: held by someone else, not acquired.

pthread_mutex_destroy(pm);          // cleanup (also for &m when no longer needed)
```

The Corrected Counter Example: Full Code with Mutex

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

int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *increment(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        counter++;              // critical section: now safe
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main(void) {
    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("Final counter value: %d\n", counter);  // always 2000000
    return 0;
}
```

Using Mutexes Correctly

- **Every** access to the shared data must hold the **same** mutex. One unprotected access brings the race back.
- The critical section must cover the **whole** read-modify-write. Locking the load and the store separately is **not** enough:

```
pthread_mutex_lock(&m);   tmp = counter;       pthread_mutex_unlock(&m);
pthread_mutex_lock(&m);   counter = tmp + 1;   pthread_mutex_unlock(&m);
// No data race (every access is locked), yet updates are still lost:
// a race condition without a data race (A.5.1).
```

- **Unlock on every path** (early `return`, `break`): a forgotten unlock leaves all other threads blocked forever.
- Local variables (the loop index `i`) live on the thread's private stack: they need no lock.

> *Other pitfalls, build flags, and thread-safe library functions:* ***Appendix A.7****.*

✅ Checkpoint 6 — The Mutex API

Test your understanding:

1. Write a complete, correct multi-threaded program that creates 4 threads, each adding 250,000 to a shared counter, and prints the final value (which must always be 1,000,000).
2. What is the difference between `pthread_mutex_lock()` and `pthread_mutex_trylock()`? When would you prefer the latter?
3. Why does the corrected counter program always print `2000000`, whereas the broken version did not?
4. In the corrected program, which variables are shared and need the mutex, and which do not? Why?
5. A thread returns from its function while still holding the mutex. What happens to the other threads?

§ 7 — Summary & Hand-off

Goal: collect the key ideas and the open question for Lecture 3.2.

📎 *Extended material → Appendix A.8 (memory ordering: optional, for the curious)*

Quick Reference: `pthread` API

```
// Thread lifecycle:
pthread_create(&tid, NULL, func, arg);   // create
pthread_join(tid, &retval);              // wait for completion
pthread_detach(tid);                     // fire and forget (details: A.3.2)

// Mutual exclusion:
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&m);
  /* critical section */
pthread_mutex_unlock(&m);
```

Summary

- **Why threads:** parallelism across cores and I/O overlap, with *sharing by default* of one address space.
- **What is private:** stack, PC, registers (in the TCB). **What is shared:** code, globals, heap, open files.
- **The price:** unsynchronized shared access causes **race conditions**; `counter++` is load-add-store, so updates get lost.
- A **critical section** needs **mutual exclusion**; a **lock** (`pthread_mutex_t`) provides it, provided *every* access takes the same lock around the *whole* read-modify-write.
- Deeper: data races, progress and bounded waiting, memory ordering are in the appendix (A.5, A.6, A.8).

Lecture 3.1 Complete

Recommended Reading

- **OSTEP Ch. 26:** *Concurrency: An Introduction*
- **OSTEP Ch. 27:** *Interlude: Thread API*
- Available free at: https://ostep.org

Contact & Q&A

- **Lecturer:** Gabriele Tolomei
- **Email:** tolomei@di.uniroma1.it

- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!

§ Appendix

Appendix Map

Section	Extends	What you will find	Slides
A.1	§ 1	IPC costs: `fork()` overhead, shared memory complexities, real-world threaded systems	3
A.2	§ 2	Thread models: Kernel threads (1:1), User threads (N:1), and Hybrid (M:N)	2
A.3	§ 3	Thread lifetime rules, detached threads, table of `pthread` pitfalls	3
A.4	§ 4	Counting interleavings; extra worked trace (exam style)	2
A.5	§ 4	Data race definition, outcomes of a race, read-modify-write; anomalies (lists, bank, TOCTOU), TSan/Helgrind	7
A.6	§ 5	Progress and bounded waiting; lock granularity	2
A.7	§ 6	Compilation commands, sanitizer flags, error-checking wrappers, thread-safety	3
A.8	§ 7	Memory models, `volatile` misconceptions, store buffers, barriers, happens-before	10

Supplementary material for deeper study. Not required during the main lecture.

📎 A.1 — From Processes to Threads: Extended Motivation

A.1.1 · The Process Approach to Parallelism: `fork()` and Its Costs

One approach: use `fork()` to create worker processes.

```
for (int i = 0; i < NUM_WORKERS; i++) {
    pid_t pid = fork();
    if (pid == 0) {
        do_work(i);
        exit(0);
    }
}
for (int i = 0; i < NUM_WORKERS; i++) wait(NULL);
```

Problems with `fork()` for parallelism:

- **Separate address space (even with COW):** children cannot write to the parent's heap — shared state needs explicit IPC.
- **Context switch cost:** switching between processes requires a page-table switch and loses TLB and cache locality.
- **Creation overhead:** `fork()` (+ `exec()`) is much heavier than creating a thread.

A.1.2 · Shared Memory Between Processes: Complex and Expensive

Processes *can* share memory, but it requires explicit setup:

```
// POSIX shared memory (simplified):
int fd = shm_open("/myshm", O_CREAT | O_RDWR, 0666);
ftruncate(fd, sizeof(shared_data_t));
shared_data_t *shared = mmap(NULL, sizeof(shared_data_t),
                             PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Every access to shared state still requires explicit synchronization (`sem_*`, `pthread_mutex_*`) because the region is visible to all attached processes. The setup — `shm_open`, `ftruncate`, `mmap`, naming, permissions — is substantial.

The key question: is there a way to get shared memory *by default*, without any of this setup? Yes: threads (§ 1).

A.1.3 · Real-World Examples of Threaded Programs

Threads are ubiquitous in production software:

- **Web servers (Apache, nginx):** one thread (or process) per connection; while one waits for a client, others serve active requests.
- **Databases (PostgreSQL, MySQL):** one thread (or process) per client connection, plus background threads for checkpointing, vacuuming, replication.
- **GUI applications:** one UI thread (must never block); separate threads for computation, network I/O, file loading.
- **JVM:** every Java program runs application, garbage-collector, JIT-compiler, and finalizer threads.
- **Scientific computing:** `OpenMP`, `pthreads` divide a large computation across all cores for near-linear speedup.

📎 A.2 — Thread Models: Kernel, User, and Hybrid

A.2.1 · Kernel Threads (1:1) vs. User Threads (N:1)

Kernel Threads (1:1 model)

- Each user thread maps to one OS kernel thread.
- The OS schedules each thread independently on any core.
- Blocking one thread does not block the others.
- Used by: Linux (`pthreads` via `clone()`), Windows, macOS.

User Threads (N:1 model)

- Many user threads multiplexed over one kernel thread.
- Thread switching done entirely in user space (no syscall).
- Very fast creation and switching, but only one core is used.
- If one thread blocks in a syscall, all block.
- Used by: early Java green threads.

A.2.2 · Hybrid Threading Models (M:N)

Possible alternative:

Hybrid (M:N model)

- M user threads mapped to N kernel threads (M > N).
- Best of both worlds in theory; complex to implement correctly.

📎 A.3 — Advanced `pthread` Management & Pitfalls

A.3.1 · Thread Termination and Lifetime Rules

- A thread ends when its function **returns** (or calls `pthread_exit(value)`); the value is collected by `pthread_join`.
- **Join each thread exactly once** — or **detach** it (`pthread_detach`, A.3.2) if nobody needs its result. Otherwise its resources are never reclaimed ("zombie thread").
- When `main` returns, or any thread calls `exit()`, the **whole process ends** and all other threads are killed. `main` must join its workers first.
- A thread's stack dies with it: **never return the address of a local variable**; return heap data and let the joiner `free()` it.

```
void *compute(void *arg) {
    int *res = malloc(sizeof(int));   // heap: survives the thread
    *res = 42;
    return res;                       // NOT: int x = 42; return &x;
}
// main: void *r; pthread_join(tid, &r); printf("%d\n", *(int *)r); free(r);
```

A.3.2 · Detached Threads: `pthread_detach()`

Sometimes we want to create a thread and forget about it — no `join()` needed:

```
pthread_t tid;
pthread_create(&tid, NULL, background_task, NULL);
pthread_detach(tid);   // OS will automatically free resources on completion
// Never call pthread_join(tid) after detaching!

// Alternative: create detached from the start using attributes:
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, background_task, NULL);
pthread_attr_destroy(&attr);
```

Detached threads are appropriate for "fire and forget" tasks: logging, background cleanup, periodic maintenance. The OS automatically frees the thread's resources when it finishes.

A.3.3 · Common `pthread` Pitfalls & Solutions

Pitfall	Description	Fix
Passing `&i` from a loop	All threads share the same `i`; race condition	Pass `(void *)i` by value or allocate per thread
Returning stack address	Stack is invalid after thread function returns	Return heap-allocated data; caller frees
Forgetting `join()`	Zombie threads; main exits before threads finish	Always `join()` or `detach()`
Double-joining	Undefined behavior after the thread has been joined	Join exactly once
Not checking return codes	Silently missed errors (e.g., out of resources)	Always check `pthread_create()` return value
Stack overflow	Deep recursion in a thread with small stack	Increase stack size via `pthread_attr_setstacksize()`

📎 A.4 — Counting Interleavings

A.4.1 · How Many Interleavings Lose an Update?

Two threads, each running three instructions in program order (T1: A, B, C; T2: D, E, F). The number of possible interleavings is

$$\binom{6}{3} = 20.$$

- The result is **2** (correct) only if one thread's STORE precedes the other's LOAD: **ABCDEF** and **DEFABC**. That is 2 of 20.
- In the other **18**, both threads load the old value (0) and the final value is **1**.

So most orderings are bad, yet the bug is rare *per increment* in practice: a thread usually runs far longer than three instructions before being preempted, so almost all executions look like the two "serial" orderings. Over 1,000,000 iterations per thread, however, the bad ones add up to many lost updates.

A.4.2 · Worked Example: Assembly-Level Interleaving Trace

Task: `counter` starts at 0. Two threads run `counter++`. Find the final value for the execution order **A D B E F C**.

```
T1: (A) mov eax,[counter]   (B) add eax,1   (C) mov [counter],eax
T2: (D) mov eax,[counter]   (E) add eax,1   (F) mov [counter],eax
(each thread has its own eax)
```

Step	Runs	eax of T1	eax of T2	counter
1	A: T1 load	0	–	0
2	D: T2 load	0	0	0
3	B: T1 add	1	0	0
4	E: T2 add	1	1	0
5	F: T2 store	1	1	1
6	C: T1 store	1	1	**1**

Final: 1 (expected 2): both loads read 0, so one increment is lost. Check yourself: order **A B C D E F** (no overlap) gives 2.

📎 A.5 — Race Conditions in Depth, Anomalies & Detection Tools

A.5.1 · Data Race vs. Race Condition

A **data race** occurs when:

1. Two or more threads access the **same memory location** concurrently.
2. At least one of the accesses is a **write**.
3. The accesses are **not protected by synchronization** (no lock, no atomic operation).

- Two threads that only *read* a variable, or that use *different* variables, have no data race.
- **Data race** = property of the memory accesses. **Race condition** = property of the program's behavior (result depends on timing).
- All data races are bugs: in C11 they are **undefined behavior**; there is no "benign" data race.

A.5.2 · Example: Many Possible Outcomes

```
int x = 0;              // shared variable

// Thread 1:            // Thread 2:
x = x + 1;              x = x * 2;
```

Both threads read and write `x` with no lock, so there are three possible final values:

Interleaving	Result
T1 completes, then T2	$x = (0 + 1) \cdot 2 = 2$
T2 completes, then T1	$x = (0 \cdot 2) + 1 = 1$
T1 and T2 both read 0; T1 writes 1, then T2 writes $0 \cdot 2$	$x = 0$

(Also $x = 1$ if both read 0 and T2 writes first.) The result depends entirely on scheduling order.

A.5.3 · Read-Modify-Write: The Fundamental Pattern

Almost all race conditions follow the **read-modify-write (RMW)** pattern:

1. **READ** the current value of a shared variable.
2. **MODIFY**: compute a new value from the value read.
3. **WRITE** the new value back (another thread may have written in between: its result is overwritten).

Common RMW operations that are unsafe without synchronization:

- `counter++` (load, add 1, store)
- `balance -= amount` (read balance, subtract, write)
- `list.insert(x)` (read head, link new node, write head)

> *Real-world anomalies and detection tools follow (A.5.4–A.5.7).*

A.5.4 · Lost Update Anomaly: Two Threads on a Shared List

A more complex race: two threads simultaneously inserting into a linked list:

```
// Shared linked list (simplified):
struct Node { int val; struct Node *next; };
struct Node *head = NULL;

void insert(int val) {
    struct Node *new = malloc(sizeof(struct Node));
    new->val = val;
    new->next = head;   // (1) read head
    head = new;          // (2) write head
}
```

```
Thread 1 (insert 10):          Thread 2 (insert 20):
(1) new1->next = head = NULL
                               (1) new2->next = head = NULL
                               (2) head = new2   [list: 20→NULL]
(2) head = new1 [list: 10→NULL]
    ↑ overwrites Thread 2's insertion!

Result: list contains only {10}, not {10, 20}.
Node 20 is leaked (malloc'd but never reachable).
```

A.5.5 · Partial Update Anomaly: Inconsistent State

Another class of race: leaving a data structure in an **inconsistent intermediate state** that another thread observes:

```
// Thread-unsafe bank transfer:
void transfer(Account *from, Account *to, int amount) {
    from->balance -= amount;   // (1) deduct from sender
    // ← context switch here ←
    to->balance += amount;     // (2) credit receiver
}

// Thread 2 reads account balances between step 1 and step 2:
int check_total(Account *a, Account *b) {
    return a->balance + b->balance;  // reads INCONSISTENT STATE!
    // Money has left 'from' but not yet arrived at 'to'.
    // Total appears lower than it should be.
}
```

This class of bug causes data corruption in databases, file systems, and financial systems — and can result in real financial loss or data loss.

A.5.6 · Races Beyond Counters: Files, Allocators, Signals

Race conditions affect virtually every shared resource:

File System

```
// TOCTOU (Time-Of-Check-Time-Of-Use) race:
if (access("secret.txt", R_OK) == 0) {  // (1) check permission
    // context switch: attacker replaces file with symlink
    open("secret.txt", O_RDONLY);        // (2) open different file!
}
```

Memory Allocator

`malloc()` uses internal free lists. Without internal locking, two threads calling `malloc()` simultaneously could receive the same block — heap corruption. (Real allocators are thread-safe precisely because they lock internally.)

Signal Handlers

A signal handler that modifies a global variable also read by the main code can introduce races — `volatile sig_atomic_t` is required.

A.5.7 · Dynamic Race Detection Tools (TSan & Helgrind)

Two essential tools for finding race conditions dynamically:

ThreadSanitizer (TSan)

```
gcc -g -fsanitize=thread -o program program.c -pthread
./program
# Output: DATA RACE on counter at address 0x...
#   Write of size 4 at 0x601060 by thread T2:
#     #0 increment counter.c:7
#   Previous write of size 4 at 0x601060 by thread T1:
#     #0 increment counter.c:7
```

Helgrind (Valgrind)

```
gcc -g -o program program.c -pthread
valgrind --tool=helgrind ./program
# Detects happens-before violations and lock ordering issues.
```

Both tools work by **instrumenting** all memory accesses at runtime and checking for concurrent conflicting accesses without proper synchronization.

📎 A.6 — Critical Section Requirements & Lock Granularity

A.6.1 · Progress and Bounded Waiting

> ***Progress:*** *if no thread is in the critical section and some threads want to enter, one of them must be able to enter in finite time. Threads in their remainder sections cannot block the others.*

> ***Bounded waiting:*** *once a thread has requested entry, there is a bound on how many times other threads can enter before it does. No thread starves.*

Why mutual exclusion alone is not enough:

- A protocol that **never lets anyone in** satisfies mutual exclusion trivially, but violates **progress**.
- A protocol that always prefers T_1 lets T_2 **starve**: it violates **bounded waiting**.

A.6.2 · Lock Granularity: Coarse vs. Fine-Grained Trade-offs

The granularity of locking affects both correctness and performance:

Coarse-Grained Locking

One lock protects many data structures:

```
lock(&global_lock);  /* access ANY shared structure */  unlock(&global_lock);
```

Simple to reason about. Poor scalability: all threads serialize on one lock.

Fine-Grained Locking

One lock per data structure or even per element:

```
lock(&list_A_lock);  /* access list A only */  unlock(&list_A_lock);
```

More complex, better parallelism (threads can use list B, tree C concurrently), but adds deadlock risk (Lecture 3.5).

📎 A.7 — Build Tools, Error Handling & Thread Safety

A.7.1 · Compiling and Linking Threaded Programs

```
# Compile: -pthread sets the compiler and linker flags
gcc -Wall -Wextra -g -o program program.c -pthread

# With sanitizers (highly recommended during development):
gcc -Wall -g -fsanitize=thread -o program program.c -pthread
# ThreadSanitizer detects data races at runtime.

# With AddressSanitizer (for memory errors):
gcc -Wall -g -fsanitize=address -o program program.c -pthread

# Check thread count (Linux):
cat /proc/$(pgrep program)/status | grep Threads
```

On Linux, pthreads are implemented via the `clone()` system call (with flags such as `CLONE_VM` and `CLONE_THREAD`), which creates a new kernel thread sharing the address space.

A.7.2 · Error Checking Wrappers: `Pthread_*` Macros

```c
// WRONG: ignoring return values
pthread_create(&tid, NULL, func, arg);
pthread_mutex_lock(&lock);

// CORRECT: check and handle errors
int rc;
rc = pthread_create(&tid, NULL, func, arg);
if (rc != 0) {
    fprintf(stderr, "pthread_create failed: %s\n", strerror(rc));
    exit(EXIT_FAILURE);
}

rc = pthread_mutex_lock(&lock);
if (rc != 0) {
    fprintf(stderr, "pthread_mutex_lock failed: %s\n", strerror(rc));
    exit(EXIT_FAILURE);
}

// Helper macro (common pattern in OSTEP examples):
#define Pthread_create(t, a, f, arg) \
    do { int rc = pthread_create(t,a,f,arg); assert(rc == 0); } while(0)
```

A.7.3 · Thread Safety in the C Standard Library

A function is **thread-safe** if it can be called from multiple threads simultaneously without data races or incorrect results:

```
// NOT thread-safe: strtok() uses internal static state
char *token = strtok(str, " ");   // modifies internal static char*

// Thread-safe alternative: strtok_r() uses caller-provided state
char *saveptr;
char *token = strtok_r(str, " ", &saveptr);

// NOT guaranteed thread-safe: rand() uses a hidden global seed
int r = rand();

// Thread-safe alternative: rand_r() uses caller-provided state
unsigned int seed = 12345;
int r2 = rand_r(&seed);
```

Always check the documentation: C library functions that use global/static state are typically **not** thread-safe.

📎 A.8 — Memory Models, Reordering & Hardware Fences

A.8.1 · Beyond Mutual Exclusion: The Ordering Problem

Mutual exclusion protects **simultaneous access** to shared data. But sometimes the problem is about **ordering**: Thread A must complete an action before Thread B observes its result.

```
// Thread A produces a result:
result = compute_answer();
done = 1;   // signal completion

// Thread B waits for the result:
while (done == 0) { /* spin */ }
use(result);   // should always see the computed answer
```

Is this correct? Intuitively yes — but with modern CPUs and compilers, it may **silently fail**. The culprit: **memory reordering**.

A.8.2 · `volatile`: A Common Misconception

Many programmers believe `volatile` prevents race conditions. It does **not**:

```
volatile int counter = 0;   // does NOT fix the race!

// volatile guarantees:
//   1. The compiler will NOT cache the variable in a register.
//   2. Every access generates an actual load/store instruction.

// volatile does NOT guarantee:
//   1. Atomicity of read-modify-write operations.
//   2. Ordering with respect to other threads.
//   3. Any protection against CPU reordering.

// counter++ is STILL three instructions even with volatile.
// The race condition is STILL present.

// Correct use of volatile: hardware registers in device drivers,
// signal handlers (sig_atomic_t), setjmp/longjmp interactions.
// NOT for inter-thread communication.
```

A.8.3 · Sequential Consistency vs. Relaxed Memory Models

A **memory model** defines the order in which memory operations appear to execute as seen by different threads:

Sequential Consistency (SC)

The ideal: all memory operations from all threads appear to execute in some global sequential order consistent with each thread's program order.

```
If Thread 1 writes X then Y, all other threads see X written before Y.
Simple to reason about, but very conservative.
```

Relaxed Memory Models (real hardware)

Modern CPUs (x86, ARM, RISC-V) may execute operations **out of order** to improve performance. Different threads may observe writes in different orders.

```
x86: store-load reordering is allowed.
ARM: all reorderings are possible (weakest model in common CPUs).
```

A.8.4 · Compiler Reordering and Optimizations

The compiler transforms code for performance, assuming single-threaded execution:

```
// Source code (as in A.8.1):
result = compute_answer();
done = 1;

// Compiler may reorder to:
done = 1;
result = compute_answer();
// In a single thread: equivalent (done and result are independent).
// In a multi-thread context: Thread B may see done=1 but result not yet set!
```

The C11 standard explicitly states: **without proper synchronization, data races are undefined behavior**. The compiler is allowed to assume they never happen, and may optimize in ways that break concurrent code.

A.8.5 · Hardware Reordering & Store Buffers

Modern CPUs use **store buffers** to decouple store operations from the cache:

```
CPU 0 writes X=1:
  → Write goes into CPU 0's store buffer (not yet visible to other CPUs).
  → CPU 0 continues executing subsequent instructions immediately.

CPU 1 reads X:
  → May read from cache (sees X=0, the old value).
  → CPU 0's store buffer content is not yet visible to CPU 1.

Result: CPU 1 sees X=0 even though CPU 0 has "written" X=1.
This is called a store-load reordering.

On x86-64: this is the ONLY reordering allowed for ordinary memory.
On ARM: all four load/store reorderings are allowed → more surprises.
```

A.8.6 · Memory Barriers and CPU Fences

A **memory barrier** is an instruction that prevents the CPU from reordering memory operations across it:

```
// GCC built-in full barrier:
__sync_synchronize();

// x86-64 assembly: mfence instruction
asm volatile("mfence" ::: "memory");

// C11 atomic fence:
#include <stdatomic.h>
atomic_thread_fence(memory_order_seq_cst);

// Effect: all loads and stores BEFORE the barrier are completed
// (globally visible) before any load or store AFTER it begins.

// Usage in the ordering example of A.8.1 (Thread A):
result = compute_answer();
__sync_synchronize();   // make result visible BEFORE done
done = 1;
// (Thread B needs a matching barrier between reading done and reading result.)
```

A.8.7 · The Happens-Before Relationship

The **happens-before** relation is the formal foundation of correct concurrent reasoning:

> ***A happens-before B*** *means that all effects of A are visible to B before B executes.*

```
Happens-before is established by:
  1. Program order within a thread:
     If A appears before B in the same thread → A happens-before B.

  2. Thread creation:
     pthread_create() → first instruction of new thread.

  3. Thread join:
     Last instruction of joined thread → instruction after pthread_join().

  4. Mutex:
     pthread_mutex_unlock() → subsequent pthread_mutex_lock() on same mutex.

  5. Condition variable (Lecture 3.3):
     pthread_cond_signal() → pthread_cond_wait() that it wakes.

If there is NO happens-before between two conflicting accesses → DATA RACE.
```

A.8.8 · Happens-Before Analysis of the Counter

```
// Thread 1:                    // Thread 2:
lock(&mutex);                    lock(&mutex);
counter++;                       counter++;
unlock(&mutex);                  unlock(&mutex);

Happens-before chain:
  T1: lock() → counter++ → unlock()
  If T2's lock() comes after T1's unlock():
    T1's unlock() happens-before T2's lock()
    → T1's counter++ happens-before T2's counter++
    → T2 sees T1's result: counter = 1 when T2 loads it.
    → T2 stores counter = 2.

  If T1's lock() comes after T2's unlock():
    Same reasoning, result is still counter = 2.

In both cases: counter = 2 (correct).
The mutex establishes the happens-before relationship that prevents the race.
```

A.8.9 · Self-Check: Memory Ordering and Concurrency

Test your understanding:

1. Does declaring a variable `volatile` prevent data races? Explain precisely what `volatile` does and does not guarantee.
2. What is a **memory model**? How does x86-64's memory model differ from ARM's?
3. The compiler reorders `done=1` before `result=compute_answer()`. In a single-threaded program, is this reordering observable? What about in a multi-threaded program?
4. What does `__sync_synchronize()` do? Where in the ordering example should it be placed to ensure correctness?
5. State the **happens-before** relationship established by a mutex. How does it prevent the counter race condition?

A.8.10 · Summary: Memory Ordering & Hardware Reality

- **Sequential consistency** is the programmer's mental model — reality is more complex.
- Modern CPUs and compilers **reorder** memory operations for performance.
- `volatile` prevents register caching but does NOT prevent reordering or races.
- **Memory barriers** (`__sync_synchronize()`, `mfence`) prevent CPU reordering.
- The **happens-before** relation (established by locks, `join()`, condition variables) is the formal guarantee of correct inter-thread visibility.
- The **C11 memory model** defines data races as undefined behavior — the compiler may assume they never occur.