Sistemi Operativi

Corso di Laurea in Informatica

Sapienza Università di Roma

2026-2027

Gabriele Tolomei

tolomei@di.uniroma1.it

Lecture 3.5: Concurrency V

Deadlock

Opening Thought

> *"Deadlock occurs, for example, when a thread (say Thread 1) is holding a lock (L1) and waiting for another one (L2); unfortunately, the thread (Thread 2) that holds lock L2 is waiting for L1 to be released."*
> *— Arpaci-Dusseau & Arpaci-Dusseau, OSTEP Ch. 32*

In Lecture 3.4 two programs hung forever: the **naive Dining Philosophers** and the **bounded buffer with `wait(mutex)` before `wait(empty)`**. They were not two unrelated bugs: they are the **same bug**. Today we name it, draw it, and study four ways to deal with it: **prevent**, **avoid**, **detect & recover**, or **ignore**. We close with concurrency bugs that are *more common* than deadlock.

Lecture Roadmap

The story of this lecture: **what is deadlock → why does it happen → stop it by design → steer around it → repair it afterwards → live with it.**

1. **The Deadlock Problem** — a minimal example and its picture: the Resource Allocation Graph (RAG)
2. **The Four Coffman Conditions** — what must hold for deadlock; Lecture 3.4 revisited
3. **Deadlock Prevention** — break one condition by design (lock ordering, `trylock`)
4. **Deadlock Avoidance** — safe states and the Banker's Algorithm
5. **Detection & Recovery** — wait-for graph, victim selection, rollback
6. **Ignoring Deadlock** — the Ostrich algorithm, `lockdep`
7. **The Most Common Bugs Are Not Deadlocks** — atomicity and order violations

> *Extended material (history, RAG details, Coffman proof, backoff and lock-free code, multi-instance detection, `lockdep`, bug details, sanitizers, decision framework, Banker's pseudocode and complexity, exercises) is in the **Appendix** (A.1–A.9): section dividers point to their appendix counterpart.*

Prerequisite Check

To follow this lecture, ensure you recall:

- **Locks and mutual exclusion** (Lecture 3.2): `lock()` / `unlock()`, critical section, `trylock()`.
- **Condition variables** (Lecture 3.3): `wait()`, `signal()`, Mesa semantics.
- **Semaphores** (Lecture 3.4): `sem_wait()`, `sem_post()`; the **Dining Philosophers** and the **bounded-buffer wait order**.
- **Thread states** (Lecture 3.1): RUNNING, READY, BLOCKED.
- **Race conditions** (Lecture 3.1): why unprotected shared state is dangerous.

New concepts today: Resource Allocation Graph, Coffman conditions, lock ordering, safe state, Banker's Algorithm, wait-for graph, Ostrich algorithm, `lockdep`, atomicity violation, order violation.

§ 1 — The Deadlock Problem

Goal: recognise a deadlock in code and draw it as a graph.

📎 *Extended material → Appendix A.1, A.2*

A Minimal Deadlock

> ***So far:*** *a lock gives mutual exclusion (3.2).* ***Problem:*** *what if a thread needs two locks?*

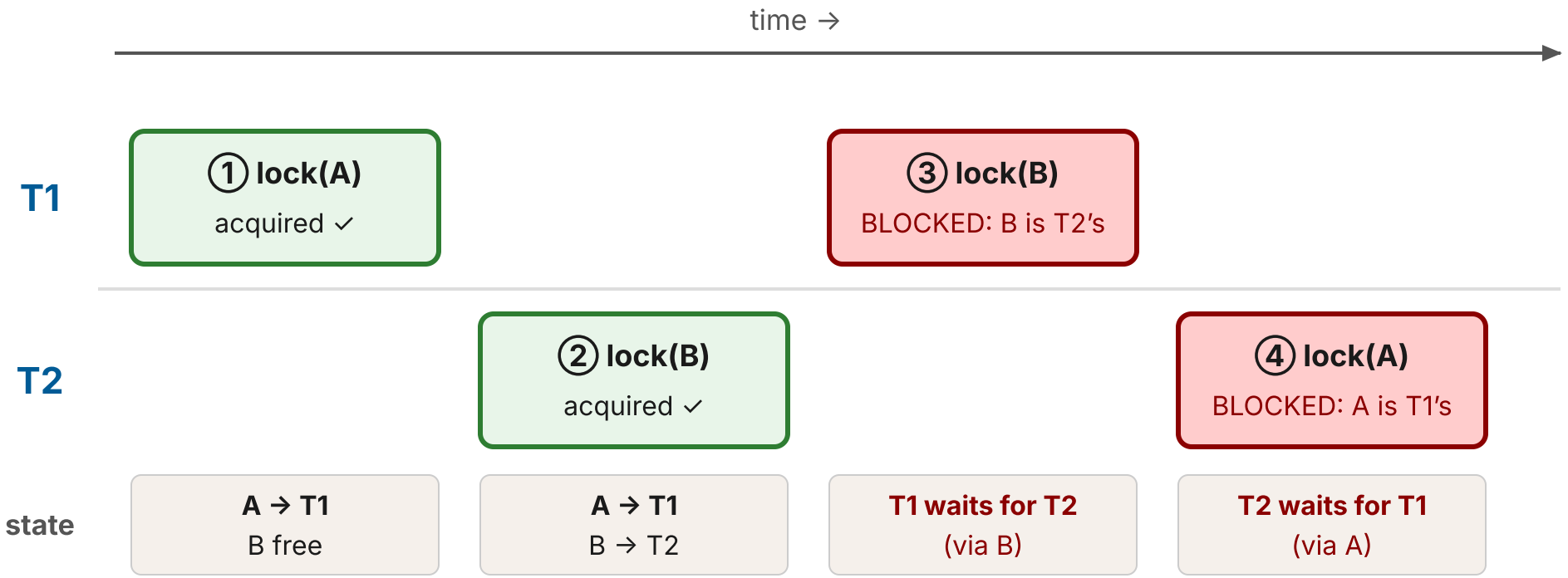
```
pthread_mutex_t lock_A = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock_B = PTHREAD_MUTEX_INITIALIZER;

void *thread1(void *arg) {
  pthread_mutex_lock(&lock_A);    // (1)
  sleep(1);                       // let T2 run
  pthread_mutex_lock(&lock_B);    // (3) BLOCKS
  /* ... critical section ... */
  pthread_mutex_unlock(&lock_B);
  pthread_mutex_unlock(&lock_A);
  return NULL;
}
```

```
void *thread2(void *arg) {
  pthread_mutex_lock(&lock_B);    // (2)
  sleep(1);                       // let T1 run
  pthread_mutex_lock(&lock_A);    // (4) BLOCKS
  /* ... critical section ... */
  pthread_mutex_unlock(&lock_A);
  pthread_mutex_unlock(&lock_B);
  return NULL;
}
```

Both threads block forever: no crash, no error message, CPU idle. Each lock is used correctly on its own; the bug is the **order** in which the two threads take them.

The Interleaving That Hangs



No thread can run again: each waits for a lock the other will never release.

Deadlock depends on **timing** (`sleep(1)` is only a demo trick that forces the bad interleaving; without it the hang is rare): if T1 finished before T2 started, the program would work. Tests may pass and production still hangs.

Drawing It: the Resource Allocation Graph (RAG)

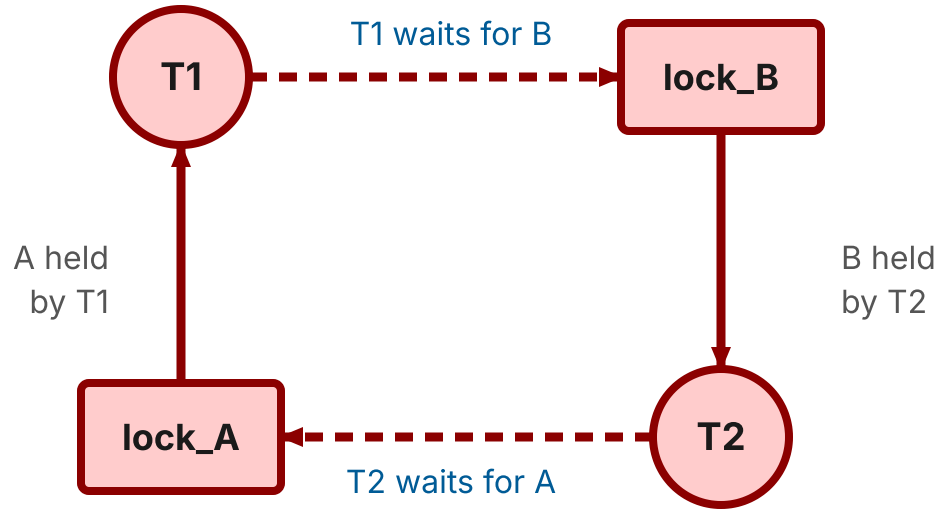
> ***Idea:*** *model "who holds what" and "who waits for what" as a directed graph $G = (V, E)$, so that a hang becomes a pattern we can see.*

- **Vertices $V = T \cup R$**: threads T_i (circles), resources R_j (squares; one dot per instance)
- **Request edge $T_i \to R_j$** (dashed): T_i is **waiting** for R_j
- **Assignment edge $R_j \to T_i$** (solid): an instance of R_j is **held** by T_i
- Reading rule: $T \to R \to T'$ means "T waits for T'"
- **Criterion:** no cycle $\Rightarrow$ no deadlock. Cycle with **single-instance** resources $\Rightarrow$ **deadlock**. Cycle with **multi-instance** resources $\Rightarrow$ deadlock *possible*, not certain.

(Our locks are single-instance resources: one holder at a time.)

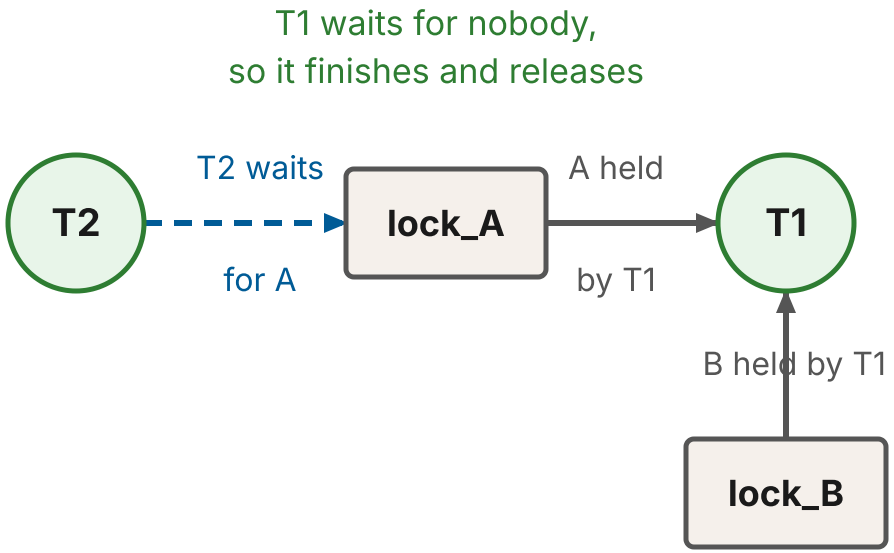
RAG: Deadlock vs. No Deadlock

DEADLOCK: the RAG has a cycle



cycle: T1 → lock_B → T2 → lock_A → T1

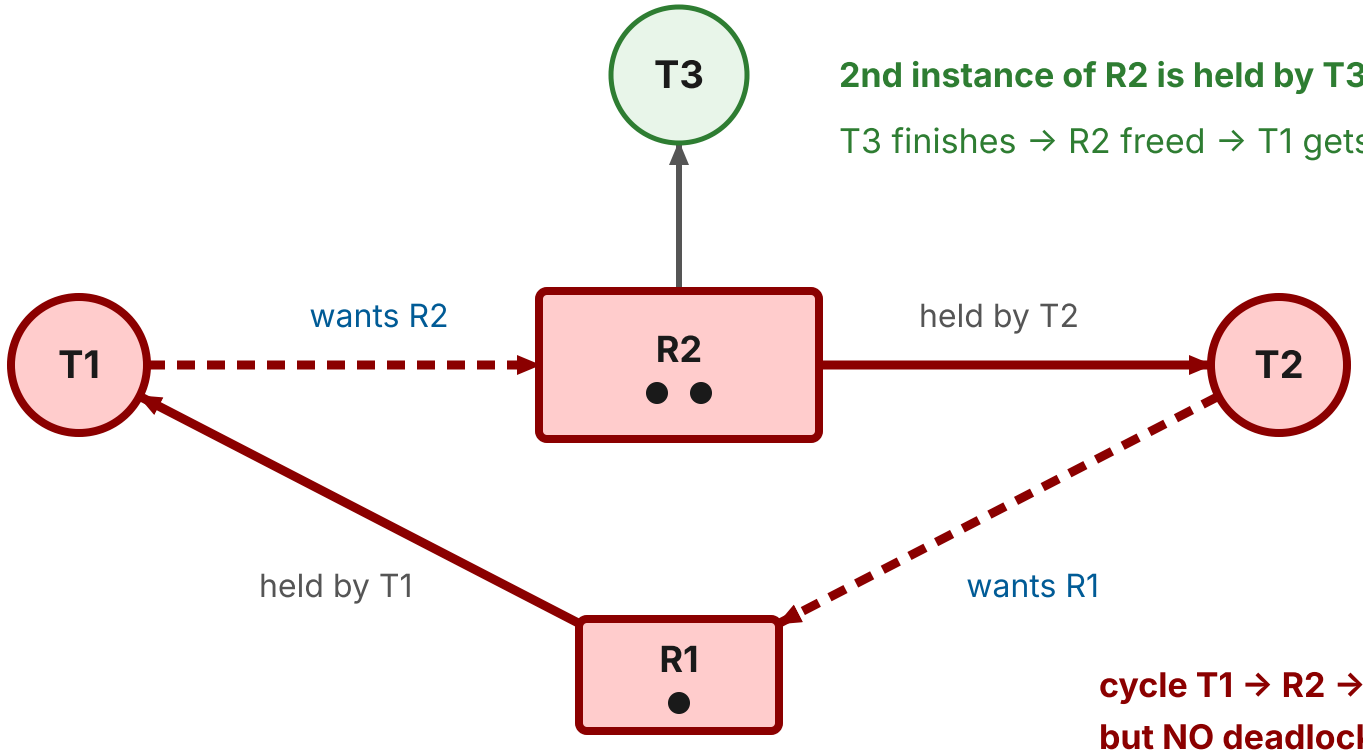
NO DEADLOCK: no cycle (both lock A first)



the chain ends at T1 (no arrow out of T1)

dashed = request (waits for) solid = assignment (held by) circle = thread square = resource

A Cycle Is Not Always a Deadlock



With **single-instance** resources, a cycle is a deadlock. With **multi-instance** resources it is only a *warning*: some thread outside the cycle may still release an instance.

✅ Checkpoint 1 — The Deadlock Problem

Test your understanding:

1. Define deadlock in one sentence. How does it differ from a thread that waits for slow I/O?
2. Draw the RAG of the two-lock program at the moment it hangs. Mark the cycle.
3. Change the program so that T2 also locks `lock_A` first. Draw the RAG when T2 is blocked. Is there a cycle?
4. Why can a deadlock-prone program pass every test and still hang in production?
5. Single-instance resources: does a cycle imply deadlock? And with multi-instance resources? Why?

Write your answers before moving on.

§ 2 — The Four Coffman Conditions

Goal: name the four ingredients of deadlock and see them in Lecture 3.4's examples.

📎 *Extended material → Appendix A.3*

The Four Coffman Conditions

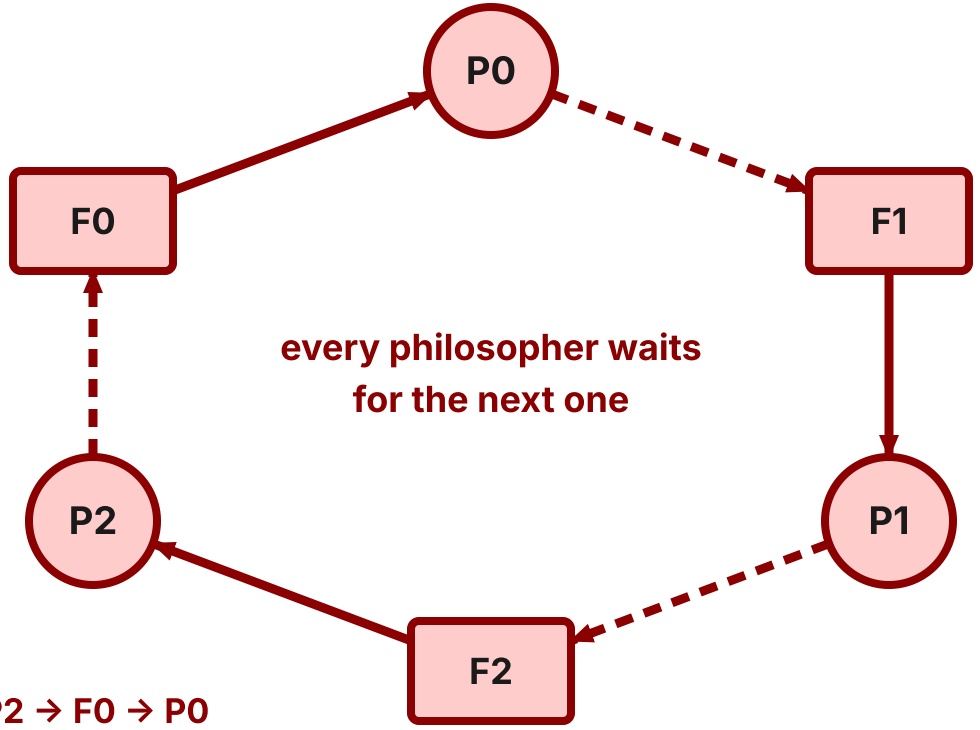
So far: *a cycle in the RAG means deadlock.* ***Question:*** *what must be true of a system for such a cycle to be possible at all?*

Condition	Meaning	In our program	Absent when…
1. Mutual exclusion	one holder at a time	`lock_A`, `lock_B` are mutexes	data is shared (read-only, rwlock in read mode)
2. Hold and wait	holds one resource, waits for another	T1 holds A, waits for B	asks for all at once, or releases first
3. No preemption	released only by its holder	OS cannot take A from T1	preemptable: CPU, memory pages (not a mutex, 3.1)
4. Circular wait	$T_1 \to \cdots \to T_n \to T_1$, each waits for the next	T1 waits for T2, T2 for T1	the chain ends at a thread waiting for nobody

All four are necessary: *remove any one and deadlock is impossible. Recipe: attack one condition (§ 3). Circular wait is the "final ingredient": the other three describe the system, the fourth the unlucky state.*

Lecture 3.4 Revisited: Dining Philosophers as a RAG

Recall (3.4): *N philosophers, N fork semaphores initialised to 1; each does* `sem_wait(&forks[i]); sem_wait(&forks[(i+1)%N]);` *. Take $N = 3$, all hungry at once.*



dashed: Pi waits for F(i+1)%3

solid: Fi is held by Pi

cycle: P0 → F1 → P1 → F2 → P2 → F0 → P0

Lecture 3.4 Revisited: The Same Bug Twice

Condition	Naive Dining Philosophers	Bounded buffer, `wait(mutex)` first
Mutual exclusion	each fork is a binary semaphore	`mutex` is a binary semaphore
Hold and wait	holds left fork, waits for right	Producer holds `mutex`, waits on `empty`
No preemption	a `sem_wait`ed fork cannot be taken	`mutex` cannot be taken away
Circular wait	$P_0 \to P_1 \to P_2 \to P_0$	Producer waits for a Consumer `post(empty)`; Consumer waits for the Producer's `mutex`
The 3.4 fix	last philosopher takes forks in **reverse order** $\Rightarrow$ breaks *circular wait*	"`wait(resource)` **before** `wait(mutex)`" $\Rightarrow$ a lock order (resources before the mutex): nobody ever waits for a resource while holding the mutex; breaks *circular wait*

> *Both fixes of 3.4 were an **instance of prevention**: they remove one Coffman condition. § 3 generalises them.*

✅ Checkpoint 2 — The Four Coffman Conditions

Test your understanding:

1. List the four Coffman conditions with a one-line description each.
2. A read-only shared file is read by many threads at once. Which condition is absent? Can they deadlock on it?
3. The CPU scheduler can preempt a running thread. Why does this mean CPU scheduling cannot deadlock, while mutexes can?
4. For the naive Dining Philosophers with $N = 3$, point at each of the four conditions.
5. In the bounded buffer, which Coffman condition does the rule "`wait(resource)` before `wait(mutex)`" remove? (Two answers are defensible: say why.)

Write your answers before moving on.

§ 3 — Deadlock Prevention

Goal: for each Coffman condition, know one technique that breaks it, and its price.

📎 *Extended material → Appendix A.4*

Prevention: Break One Condition

> ***So far:*** *deadlock needs all four conditions.* ***Idea:*** *design the program so that one of them can never hold. Then deadlock is impossible by construction.*

Break…	Technique	Price
1. Mutual exclusion	no lock at all: atomic operations (CAS, 3.2), read-only sharing (Appendix A.4)	only for simple data structures
2. Hold and wait	take all locks at once or none (`trylock`)	lower concurrency; starvation, livelock
3. No preemption	if a lock is busy, give back what you hold and retry	redone work; livelock
4. Circular wait	fixed global **lock order**	order must be known and obeyed: **the practical choice**

> ***Livelock:*** *threads keep running (retrying) but make no progress. Unlike deadlock, nobody is blocked.*

Breaking Hold and Wait (and No Preemption) with `trylock`

```
// Returns holding BOTH locks, or loops until it can. Never waits while holding one.
void acquire_both(pthread_mutex_t *a, pthread_mutex_t *b) {
    while (1) {
        if (pthread_mutex_trylock(a) == 0) {        // got A
            if (pthread_mutex_trylock(b) == 0)
                return;                              // got B too: success
            pthread_mutex_unlock(a);                 // B busy: give A back
        }
        sched_yield();                               // then retry
    }
}
```

- `trylock` returns `0` on success, `EBUSY` if held: it **never blocks**, so no hold-and-wait.
- Giving A back when B is busy is a **self-inflicted preemption** (the OS never takes a mutex): this breaks *no preemption* too.
- Price: two colliding threads may retry in lockstep forever (**livelock**). A **random backoff** (`nanosleep` of a random few µs, Appendix A.4) makes that unlikely.

Breaking Circular Wait: Lock Ordering

> ***Idea:*** *give every resource a number $f(R)$; every thread acquires locks in* ***increasing*** *order of f.*

Why it works. Suppose a cycle $T_1 \to T_2 \to \cdots \to T_n \to T_1$ existed, where T_i holds R_i and waits for R_{i+1}. The rule forces $f(R_i) < f(R_{i+1})$ for each i, so going around the cycle:

$$f(R_1) < f(R_2) < \cdots < f(R_n) < f(R_1)$$

which is impossible. No cycle can form.

> ***This is exactly what 3.4 did.*** *Number the forks $f(\text{fork}_i) = i$. Philosophers $0..N-2$ take fork_i then fork_{i+1} (increasing); the last one takes fork_0 before fork_{N-1} (also increasing). The "asymmetric" trick is lock ordering.*

Lock Ordering: Code

```
typedef struct { int id; pthread_mutex_t lock; int balance; } Account;

void transfer(Account *from, Account *to, int amount) {
    if (from == to) return;                          // else we would lock the same mutex twice
    Account *first  = (from->id < to->id) ? from : to;   // lower id first: f(account) = id
    Account *second = (from->id < to->id) ? to   : from;

    pthread_mutex_lock(&first->lock);
    pthread_mutex_lock(&second->lock);
    from->balance -= amount;
    to->balance   += amount;
    pthread_mutex_unlock(&second->lock);
    pthread_mutex_unlock(&first->lock);
}
```

`transfer(a,b)` and `transfer(b,a)` now both lock the lower id first: the opposite-order pattern of § 1 cannot happen.

✅ Checkpoint 3 — Deadlock Prevention

Test your understanding:

1. Which Coffman condition does each of the four prevention techniques target? What does each cost?
2. What is livelock? Why does `trylock`-based acquisition need random backoff?
3. Two threads call `acquire_both(a,b)` and `acquire_both(b,a)` in perfect lockstep. Describe the livelock. How does random backoff help?
4. A bank has 1000 accounts. Which lock order prevents deadlock between `transfer(a,b)` and `transfer(b,a)`? Why does the code need the `from == to` test?
5. Verify with $f(\text{fork}_i) = i$ that the asymmetric Dining Philosophers of 3.4 has no circular wait.

Write your answers before moving on.

§ 4 — Deadlock Avoidance: Banker's Algorithm

Goal: decide whether a state is *safe*, and grant a request only if it keeps the system safe.

📎 *Extended material → Appendix A.9 (Banker's pseudocode and complexity), A.8 (decision framework)*

Safe States and the Safety Algorithm

So far: *prevention forbids whole patterns, even harmless ones.* ***Idea:*** *if each thread declares its* ***maximum*** *need in advance, grant a request only when everybody can still finish.*

Safe state: *a* ***safe sequence*** $\langle T_1, \ldots, T_n \rangle$ *exists: each* T_i*'s remaining need can be met by the free resources* ***plus*** *those released by earlier* T_j $(j < i)$. ***Unsafe*** $\neq$ ***deadlock****, but deadlock can no longer be ruled out (Appendix A.8).*

Per thread and resource type: **Max** (declared up front), **Allocation** (held), **Need** = Max − Allocation; **Available** (free). $X \leq Y$ is component-wise.

```
1. Work = Available;  Finish[i] = false for all i
2. Find i with Finish[i] == false AND Need[i] <= Work     // Ti can run to completion now
3. If found: Work = Work + Allocation[i]; Finish[i] = true; go to 2   // Ti releases all
4. All Finish[i] true -> SAFE (the order of the picks is a safe sequence); else UNSAFE
```

Banker's Example (1/3): The State

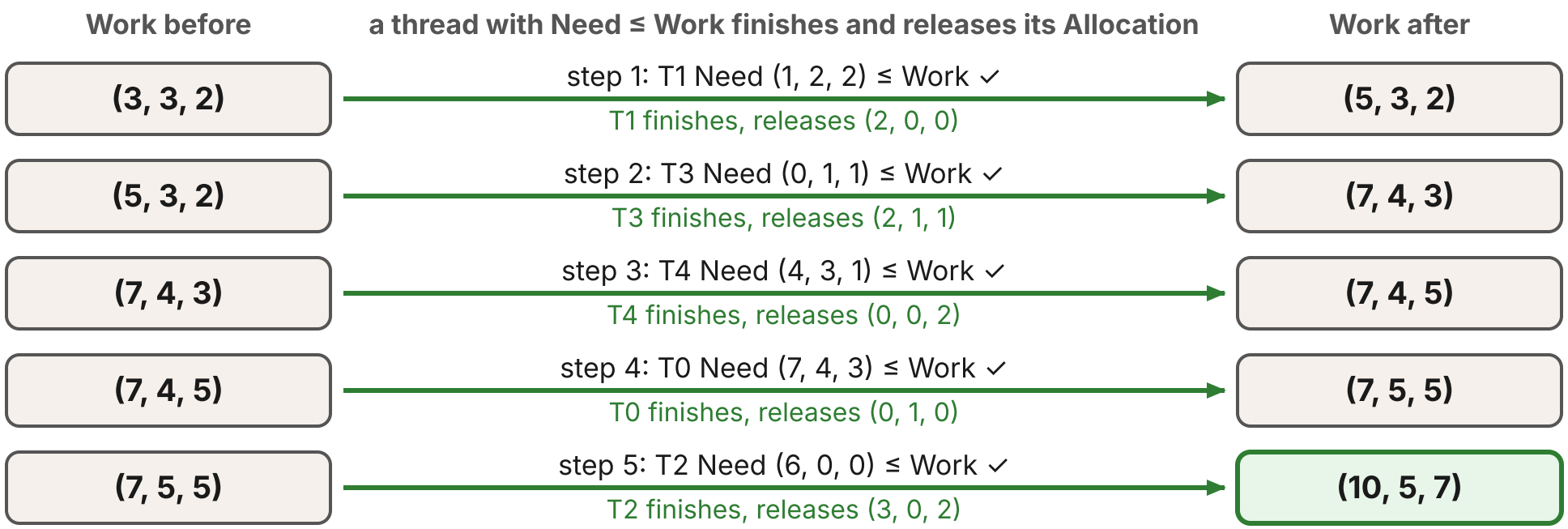
Five threads T_0–T_4, three resource *types* A, B, C (not the locks of § 1), $\text{Total} = (10, 5, 7)$.

Thread	Allocation (A B C)	Max (A B C)	Need = Max − Alloc
T_0	0 1 0	7 5 3	7 4 3
T_1	2 0 0	3 2 2	1 2 2
T_2	3 0 2	9 0 2	6 0 0
T_3	2 1 1	2 2 2	0 1 1
T_4	0 0 2	4 3 3	4 3 1

$\sum \text{Allocation} = (7, 2, 5)$, so $\text{Available} = \text{Total} - \sum \text{Allocation} = (10, 5, 7) - (7, 2, 5) = (3, 3, 2)$.

> ***Question:*** *is this state safe? Only T_1 and T_3 have $\text{Need} \leq (3, 3, 2)$ right now.*

Banker's Example (2/3): Finding a Safe Sequence



Work = Total (10, 5, 7): all finished → SAFE, with safe sequence ⟨T1, T3, T4, T0, T2⟩

Other safe sequences exist (e.g. starting with T_3); one is enough to prove the state is safe.

Banker's Example (3/3): Granting a Request

Resource-Request algorithm for a request by T_i:

1. Request $>$ Need: **error** (exceeds the declared maximum).
2. Request $>$ Available: the thread must **wait** (not enough free resources).
3. Otherwise **pretend** to grant it (Available $-$, Allocation $+$, Need $-$ by Request) and run the safety algorithm: **safe** $\Rightarrow$ **grant**; **unsafe** $\Rightarrow$ **undo and wait**.

- T_1 **requests** $(1, 0, 2)$: within Need $(1, 2, 2)$ and Available $(3, 3, 2)$. Pretend: Available $(2, 3, 0)$, T_1 has Alloc $(3, 0, 2)$, Need $(0, 2, 0)$. Safe sequence $\langle T_1, T_3, T_4, T_0, T_2 \rangle \Rightarrow$ **granted**.
- T_4 **requests** $(3, 3, 0)$ (original state): within Need $(4, 3, 1)$ and Available. Pretend: Available $(0, 0, 2)$, T_4 Need $(1, 0, 1)$. No thread has Need $\leq (0, 0, 2) \Rightarrow$ **unsafe, denied**.

> ***Why rarely used:*** *needs every thread's maximum up front, a fixed set of threads and resources, and a safety check on every request.*

✅ Checkpoint 4 — Deadlock Avoidance: Banker's Algorithm

Test your understanding:

1. What is the difference between a *safe* and an *unsafe* state? Can an unsafe state guarantee deadlock?
2. What must each thread declare before it starts? Give two reasons why that is a practical limitation.
3. $n = 3$, $m = 2$ (Total $A = 4$, $B = 3$). Allocation: $T_0 = (1, 1)$, $T_1 = (2, 1)$, $T_2 = (1, 0)$. Max: $T_0 = (3, 2)$, $T_1 = (2, 2)$, $T_2 = (3, 1)$. Compute Available and Need, then run the safety algorithm. Is the state safe?
4. In that state: (a) T_1 requests $(0, 1)$; (b) T_0 requests $(0, 1)$. For each, apply the Resource-Request algorithm: grant or wait?
5. Prevention vs. avoidance: which one needs the maximum claims, and which one restricts *how* locks are taken?

Write your answers before moving on.

§ 5 — Deadlock Detection & Recovery

Goal: detect a deadlock with a wait-for graph and know how to recover from it.

📎 *Extended material → Appendix A.5*

Detection: Cycle in the Wait-For Graph

> ***So far:*** *prevention and avoidance both restrict the program in advance.* ***Idea:*** *be optimistic. Let deadlocks happen,* ***detect*** *them, then* ***repair****. Attractive when deadlock is rare and repair is cheap (databases).*

For **single-instance** resources, collapse the RAG into a **Wait-For Graph (WFG)**: edge $T_i \to T_j$ means "T_i waits for a resource held by T_j" (the path $T_i \to R \to T_j$ of the RAG).

```
Two-lock deadlock:     RAG  T1 -> lock_B -> T2 -> lock_A -> T1
                       WFG  T1 -> T2 -> T1                    cycle: DEADLOCK

Three threads:         WFG  T1 -> T2 -> T3 -> T1              cycle: DEADLOCK
```

- Run a DFS on the WFG: a **back edge** means a cycle. Cost $O(V + E)$.
- Run it periodically, or when a request has been blocked "too long".
- Multi-instance resources need a matrix algorithm similar to Banker's (Appendix A.5).

Recovery: Terminate or Preempt

Once a cycle is found, the system must break it.

- **Terminate:** abort *all* deadlocked threads (simple, all work lost) or *one at a time*, re-running detection after each (less waste, more detection cost).
- **Preempt and roll back:** take **resources** from a victim, roll it back (*total*: restart; *partial*: to a saved checkpoint), give the resources to another thread, restart the victim.
- **Which victim?** A cost function: priority, work done, resources held and still needed, number of threads killed, interactive vs. batch.
- **Starvation:** the same cheap victim forever never finishes: add *how often it was a victim* to the cost.
- **Challenge:** preemption needs saveable state. CPU registers, pages: yes. A mutex held mid-update, a half-written file: no. So databases (undo log) do it routinely, general OSes do not (Appendix A.5).

§ 6 — Ignoring Deadlock: What Real Systems Do

Goal: explain why most systems just ignore deadlock, and which safety nets they add.

📎 *Extended material → Appendix A.6, A.7*

The Ostrich Algorithm

> ***So far:*** *three active strategies, each with a cost (restrictions, declared maxima, detection).* ***Fourth option:*** *do nothing. The* ***Ostrich algorithm*** *"sticks its head in the sand": assume deadlock never happens; if it does, the user kills the process or reboots.*

Why this can be rational (illustrative numbers):

- Application code and kernels: deadlock is rare, and `lockdep` (next slide) catches most ordering bugs *before* release.
- Full prevention/avoidance costs *every* run: ordering rules, declared maxima, reduced concurrency.
- Suppose a deadlock costs 1 minute of downtime per year, while avoidance slows the system by 10% all the time (≈ 876 h per year). The cheap, rare event wins.
- **Databases differ:** many transactions lock many rows, deadlocks are routine, and rollback is cheap, so they use *detection + recovery* (§ 5).

Safety Nets Without Prevention

- `lockdep` (Linux debug kernels, `CONFIG_PROVE_LOCKING`): records the order in which lock *classes* are taken. If it sees A→B in one place and B→A in another, it warns of a **possible** circular dependency **even if no hang occurred yet**. Too slow for production.
- **Error-checking mutex** (`PTHREAD_MUTEX_ERRORCHECK`): re-locking a mutex you already hold returns `EDEADLK` instead of hanging. It detects **only self-deadlock**, not a cycle between two threads.
- **Timeouts** (`pthread_mutex_timedlock`): give up after x seconds and log, retry or abort.
- **Sanitizers** (TSan, Helgrind): report lock-order inversions in tests (Appendix A.7).

> *These are the **detective** counterpart of lock ordering: they enforce it by checking, not by construction.*

✅ Checkpoint 5 — Detection, Recovery & Ignoring Deadlock

Test your understanding:

1. Derive the WFG from your RAG of Checkpoint 1. Which algorithm finds the cycle, and at what cost?
2. Name the two recovery strategies. List four victim-selection criteria and say how to avoid starving a victim.
3. What is the Ostrich algorithm? Why do most general-purpose OSes use it, while databases use detection and recovery?
4. What does `lockdep` check, in which builds, and why not in production?
5. What does `PTHREAD_MUTEX_ERRORCHECK` detect, and what does it *not* detect?

Write your answers before moving on.

§ 7 — The Most Common Bugs Are Not Deadlocks

Goal: know the two bug classes that outnumber deadlocks, and their fixes from 3.2–3.4.

📎 *Extended material → Appendix A.7*

Atomicity and Order Violations

> ***So far:*** *deadlock = threads waiting forever. A study of 105 real bugs (Lu et al., OSTEP Ch. 32):* ***74 non-deadlock vs. 31 deadlocks****; about* ***97%*** *of the non-deadlock bugs are atomicity or order violations.*

- **Atomicity violation:** a region assumed to run as one unit is interleaved with another thread. Fix: a **lock** around check *and* use.
- **Order violation:** "A before B" is not enforced (a worker reads `data` before the init thread set it). A mutex cannot fix it; a **semaphore initialised to 0** can (3.4): it works even if the post comes first, since the semaphore remembers it. Details: Appendix A.7.

```
// MySQL (simplified)   Thread 1:                    Thread 2:
if (thd->proc_info) {              // (1) check       thd->proc_info = NULL;   // (3)
    fputs(thd->proc_info, ...);    // (2) use         // crash if (3) runs between (1) and (2)
}
// FIX: lock around (1)+(2), and Thread 2 takes the SAME lock around (3).
pthread_mutex_lock(&thd->lock);  if (thd->proc_info) fputs(thd->proc_info, ...);  pthread_mutex_unlock(&thd->lock);
```

✅ Checkpoint 6 — The Most Common Bugs Are Not Deadlocks

Test your understanding:

1. In the study of 105 real bugs, how many were deadlocks and how many were not? What does this say about where to spend debugging effort?
2. Explain the MySQL interleaving that crashes. Which operations belong inside the critical section, and why must Thread 2 take the lock too?
3. Describe an order violation. Why can a mutex alone not fix it?
4. Why does a semaphore initialised to 0 fix it, even if the init thread posts before the worker waits?

Write your answers before moving on.

Summary

Strategy	Idea	Condition attacked	Cost / who uses it
Prevention	design so one condition never holds	any one; **lock ordering** (4) is the practical one	restricts the program; application code
Avoidance	grant a request only if the state stays **safe** (Banker's)	none; steers around deadlock	needs max claims; rare
Detection & recovery	let it happen, find the cycle (WFG), abort/rollback	none	detection overhead, lost work; **databases**
Ignore (Ostrich)	assume it is rare; reboot if it happens	none	cheapest; most OSes, plus `lockdep` in debug builds

- Deadlock needs **all four** Coffman conditions; RAG: cycle + single-instance ⇒ deadlock. 3.4's fixes were prevention.
- More common than deadlock: **atomicity** violations (fix: lock) and **order** violations (fix: semaphore/CV).

📎 *Appendix A.8: decision framework, cheat sheet, common mistakes, exercises.*

Lecture 3.5 Complete

Recommended Reading

- **OSTEP Ch. 32:** *Common Concurrency Problems*
- 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	Historical context: Dijkstra, "deadly embrace", Coffman et al.	1
A.2	§ 1	RAG edges for semaphores; RAG of $N = 5$ Dining Philosophers	2
A.3	§ 2	Proof sketch: why the Coffman conditions are necessary	1
A.4	§ 3	All-or-nothing hazards, kernel/DB lock order, `trylock` backoff, lock-free	4
A.5	§ 5	Multi-instance detection, checkpoint-and-rollback, database rollback	4
A.6	§ 6	Kernel deadlocks, `lockdep`, POSIX error checking, case studies	5
A.7	§ 6–7	Atomicity/order violations (MySQL), error codes, TSan, Helgrind	6
A.8	§ 4	Decision framework, safe/unsafe diagram, cheat sheet, mistakes, exercises	5
A.9	§ 4	Banker's pseudocode, correctness, complexity	4

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

📎 A.1 — Historical Context of Deadlock

A.1.1 · Historical Note: Deadlock and the "Deadly Embrace"

```
1965: Dijkstra ("Cooperating Sequential Processes") names the problem
      "deadly embrace" and describes the Banker's Algorithm,
      the first deadlock-avoidance scheme.

1971: Coffman, Elphick and Shoshani, "System Deadlocks" (ACM Computing Surveys):
      the four necessary conditions (Coffman conditions) are formalized.

1970s: relational database systems (e.g. IBM System R) introduce
      deadlock detection with transaction rollback.

Today: Linux uses `lockdep` for kernel lock-order checking.
       Every major RDBMS has a built-in deadlock detector.
       Java has ThreadMXBean.findDeadlockedThreads().
```

The deadlock problem is as old as concurrent programming itself. Our understanding of it is mature, but writing deadlock-free code remains one of the hardest challenges in systems programming.

📎 A.2 — Extended Resource Allocation Graphs

A.2.1 · Mapping Semaphores and Multi-Unit Resources to Graph Edges

```
// How sem_wait() and sem_post() map to RAG edges:

sem_t s;
sem_init(&s, 0, 1);  // binary semaphore (1 instance)

// Thread T1:
sem_wait(&s);  // If s.value == 0: adds REQUEST edge T1 → s.
               // If s.value > 0: adds ASSIGNMENT edge s → T1.

sem_post(&s);  // Removes ASSIGNMENT edge s → T1.
               // If T2 is waiting: moves ASSIGNMENT edge to T2,
               //   removes REQUEST edge T2 → s.

// The RAG evolves dynamically as threads call sem_wait/sem_post.
// Deadlock = cycle appears in the RAG and never disappears.
```

A.2.2 · RAG for $N = 5$ Dining Philosophers

```
Setup: 5 philosophers (P0–P4), 5 forks (F0–F4), each fork: 1 instance.
Naive strategy: each philosopher picks up LEFT fork, then RIGHT fork.

Deadlock scenario (all pick up the left fork before anyone picks the right):
  P0 holds F0, requests F1
  P1 holds F1, requests F2
  P2 holds F2, requests F3
  P3 holds F3, requests F4
  P4 holds F4, requests F0

RAG cycle:
  P0 → F1 → P1 → F2 → P2 → F3 → P3 → F4 → P4 → F0 → P0

Cycle of length 10 (all threads and all resources).
All philosophers wait, all forks are held. DEADLOCK.
(Same picture as the N=3 slide of § 2, with two more philosophers.)
```

📎 A.3 — Formal Proof of the Coffman Conditions

A.3.1 · Proof Sketch: Necessity of the Coffman Conditions

```
Claim: deadlock cannot occur if any one condition is missing.

Proof by contradiction (for Circular Wait):
  Suppose deadlock occurs but there is no circular wait.
  Then the wait-for graph is acyclic (a DAG).
  In any finite DAG, there exists at least one node with no outgoing edges.
  That node = a thread that is NOT waiting for anyone.
  That thread can proceed, complete, and release its resources.
  The next thread in the chain can now proceed. And so on.
  → All threads eventually complete. No deadlock. Contradiction. ✓

Analogous arguments exist for each condition:
  Without Mutual Exclusion: no thread blocks on a resource → no wait graph.
  Without Hold-and-Wait: blocked threads hold nothing → no "held" edges.
  Without No-Preemption: resources can be taken → graph edges can be removed.
```

📎 A.4 — Advanced Deadlock Prevention Mechanics

A.4.1 · Pitfalls of All-or-Nothing Lock Acquisition

```
Problem 1: Reduced Concurrency
  If a thread needs 10 resources but only 9 are free,
  it acquires NONE and waits, even though it could start with 9.
  Resource utilization drops; throughput suffers.

Problem 2: Starvation
  A thread that needs many popular resources may wait indefinitely
  because all of them are never simultaneously free.
  → Starvation (not deadlock, but still a liveness problem).

Problem 3: Encapsulation
  A thread must know in advance ALL resources it will need.
  In modular code, a function call may acquire locks internally.
  The caller cannot know about inner locks at the outer level.
  → Breaks abstraction.

Problem 4: Practical Infeasibility
  In dynamic systems (databases, file systems), the set of needed
  resources is often not known until mid-execution.
```

A.4.2 · Hierarchical Lock Ordering in Kernels and Databases

```
Linux kernel:
  The kernel documents a hierarchy of lock classes
  (e.g. in the memory-management code: mmap_lock is taken before
  page-table locks). Violating the order is a kernel bug,
  flagged by lockdep.

Database systems (typical lock hierarchies):
  PostgreSQL: relation lock → page lock → row lock (never in reverse).
  MySQL InnoDB: tablespace → table → page → row.

Rule of thumb for application code:
  Assign a global integer to every mutex at initialization time.
  Always acquire in ascending order of this integer.
  Never acquire a mutex with a lower ID than one you already hold.

Limitation:
  The ordering must be known at design time.
  Dynamic resource graphs (e.g., graph databases) make this hard.
```

A.4.3 · `trylock` with Random Backoff

```
void acquire_both_backoff(pthread_mutex_t *a, pthread_mutex_t *b) {
    while (1) {
        pthread_mutex_lock(a);
        if (pthread_mutex_trylock(b) == 0)
            return;                                  // success: hold A and B
        pthread_mutex_unlock(a);                     // B busy: voluntarily give A back
        struct timespec ts = { .tv_sec = 0, .tv_nsec = 1000 + rand() % 10000 };
        nanosleep(&ts, NULL);                        // random backoff avoids lockstep (livelock)
    }
}
```

Releasing A is a **self-inflicted preemption**. Without the random delay, two threads can retry in lockstep forever.

A.4.4 · Breaking Mutual Exclusion: Lock-Free Operations

```c
#include <stdatomic.h>
atomic_int counter = 0;

void increment(void) {
    int old = atomic_load(&counter);
    // CAS (3.2): if counter == old, set it to old+1 atomically.
    // On failure, 'old' is refreshed with the current value: just retry.
    while (!atomic_compare_exchange_weak(&counter, &old, old + 1))
        ;
}
```

- No mutex ⇒ no lock to wait for ⇒ no deadlock on it.
- Used in Linux (`atomic_t`), Java (`AtomicInteger`), C++ (`std::atomic`), lock-free queues and stacks.
- **Limit:** a single word or carefully designed structures, not "update these three fields together".

📎 A.5 — Multi-Instance Detection & Rollback Systems

A.5.1 · Multi-Instance Deadlock Detection Algorithm

For systems with multiple instances per resource, we use a matrix algorithm (similar to Banker's, but with the **current Request** instead of the future Need):

```
Data structures:
  Allocation[n][m]: current allocation (same as before).
  Request[n][m]:    current outstanding request for each thread.
  Available[m]:     currently free instances.

Algorithm:
  1. Work = Available.copy()
     Finish[i] = (Allocation[i] == 0) for all i.
     (A thread holding nothing cannot be part of a deadlock.)

  2. Find i where Finish[i] == false AND Request[i] <= Work.
     (Thread i's current request can be satisfied right now.)

  3. If found:
       Work = Work + Allocation[i]  (simulate Ti completing)
       Finish[i] = true
       Go to step 2.

  4. Every i with Finish[i] == false is DEADLOCKED.
```

A.5.2 · Multi-Instance Detection: Worked Matrix Example

```
n=5 (T0–T4), m=3 (A,B,C). Total: A=7, B=2, C=6.

         Allocation    Request      Available
         A  B  C       A  B  C      A  B  C
  T0:    0  1  0       0  0  0       0  0  0
  T1:    2  0  0       2  0  2
  T2:    3  0  3       0  0  1
  T3:    2  1  1       1  0  0
  T4:    0  0  2       0  0  2

Check: sum of Allocation = (7,2,6) = Total, so Available = (0,0,0).
Everyone holds something, so Finish = [F,F,F,F,F]; Work = (0,0,0).

Step 1: T0: Request (0,0,0) <= Work  -> finishes, Work = (0,1,0).
Step 2: T1 (2,0,2)? no.  T2 (0,0,1)? no (C: 1 > 0).
        T3 (1,0,0)? no.  T4 (0,0,2)? no.

No more progress: T1, T2, T3, T4 are DEADLOCKED.  T0 is not.
(If T2 requested (0,0,0) instead: T0,T2,T1,T3,T4 all finish, no deadlock.)
```

A.5.3 · Checkpoint-and-Rollback Architecture

```
Concept: periodically save the state of a thread/transaction.
  If the thread must be rolled back, restore from the last checkpoint.

Implementation in databases (UNDO log):
  Before modifying any data, write the OLD value to the UNDO log.
  If a transaction is chosen as deadlock victim:
    Read the UNDO log in reverse.
    Undo each modification (restore old values).
    Release all locks held by the transaction.
    The transaction is effectively "rolled back" to its start.
  The row locks are freed → the other deadlocked transactions can proceed.

Implementation in OS processes:
  Much harder than databases:
    Processes may have side effects (network, file writes) that can't be undone.
    Only memory state can be restored from a checkpoint image.
  Used in HPC (High-Performance Computing) for fault tolerance:
    BLCR (Berkeley Lab Checkpoint/Restart) saves and restores process images.
```

A.5.4 · Database Transaction Rollback (WAL & ARIES)

```
PostgreSQL deadlock recovery (example):

  ERROR:  deadlock detected
  DETAIL: Process 42 waits for ShareLock on transaction 99;
          blocked by process 99.
          Process 99 waits for ShareLock on transaction 42;
          blocked by process 42.
  HINT:   See server log for query details.

Actions taken automatically by the database:
  1. Selects one transaction as victim.
  2. Rolls the victim transaction back.
  3. Releases all row locks held by the victim.
  4. The surviving transaction proceeds.
  5. The application receives a "deadlock detected" error code.
  6. The application retries the rolled-back transaction.

Application responsibility:
  Catch the deadlock error.
  Retry the transaction (often in a loop with backoff).
```

📎 A.6 — Kernel Lockdep Internals & Production Cases

A.6.1 · Deadlock Scenarios in the Linux Kernel

```
How Linux handles kernel-level deadlocks:

1. lockdep (development builds only):
   CONFIG_PROVE_LOCKING=y
   Detects lock order violations at runtime during kernel development.
   Every violation is a kernel BUG — developers must fix it before release.

2. Lock ordering conventions (documentation):
   The kernel documents lock hierarchies, e.g. in memory management
   mmap_lock is taken before page-table locks.
   Developers are expected to follow these conventions.

3. RCU (Read-Copy-Update):
   A non-locking concurrency mechanism for read-mostly data structures.
   Readers take no locks → cannot deadlock.
   Writers use a specific protocol that avoids holding multiple locks.

4. Production kernels: no general deadlock detection at runtime.
   If deadlock occurs: the kernel hangs → user must reboot.
   Watchdogs ("soft lockup", "hung task") log a warning before the system freezes.
```

A.6.2 · Linux `lockdep`: Example Output

```
======================================================
WARNING: possible circular locking dependency detected
(abridged and illustrative)
------------------------------------------------------
kworker/u4:2/123 is trying to acquire lock:
ffff88801234abcd (&mm->mmap_lock){++++}-{3:3}, at:
                 do_mmap+0x123/0x456

but task is already holding lock:
ffff88807890ef01 (&inode->i_rwsem){++++}-{3:3}, at:
                 ext4_file_write_iter+0x78/0x200

which lock already depends on the new lock.
the existing dependency chain (in reverse order) is:

 -> #1 (&inode->i_rwsem){++++}-{3:3}:
        down_read+0x22/0x80
        filemap_fault+0x123/0x345
        do_fault+0x45/0x90

 -> #0 (&mm->mmap_lock){++++}-{3:3}:
        down_write+0x22/0x80
        do_mmap+0x123/0x456
```

A.6.3 · Deadlock Detection in POSIX Threads

```
// POSIX mutex error detection for self-deadlock:
pthread_mutex_t m;
pthread_mutexattr_t attr;

pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK);
pthread_mutex_init(&m, &attr);

// Thread T1:
pthread_mutex_lock(&m);    // OK
pthread_mutex_lock(&m);    // Returns EDEADLK (does not block forever):
                            // ERRORCHECK detects the same thread re-locking.

// A cross-thread deadlock is NOT detected by default.
// pthread_mutex_timedlock() blocks with a timeout (absolute CLOCK_REALTIME time):
struct timespec ts = { .tv_sec = time(NULL) + 5 };  // 5-second timeout
int ret = pthread_mutex_timedlock(&m, &ts);
if (ret == ETIMEDOUT) {
    // Potential deadlock: took too long to acquire the lock.
    // Application can log, retry, or abort.
}
```

A.6.4 · Case Study: Dining Philosophers Prevention by Ordering

```
// As in Lecture 3.4: the last philosopher picks the RIGHT fork first.

void philosopher(int id) {
    int left  = id;
    int right = (id + 1) % NUM_PHILOSOPHERS;

    while (1) {
        think();
        if (id == NUM_PHILOSOPHERS - 1) {
            sem_wait(&forks[right]);   // reversed order breaks the cycle
            sem_wait(&forks[left]);
        } else {
            sem_wait(&forks[left]);
            sem_wait(&forks[right]);
        }
        eat();
        sem_post(&forks[left]);
        sem_post(&forks[right]);
    }
}

// P0–P3: pick left then right (increasing fork number).
// P4:    picks right (fork 0) then left (fork 4): also increasing.
// All threads acquire forks in increasing order: no cycle. No deadlock. ✓
```

A.6.5 · Case Study: Two-Lock Deadlock in Production Systems

```
Real bug anatomy (simplified from open-source code base):

Module A: handles network packets.
  lock_order: acquire net_lock, then buf_lock.

Module B: handles buffer recycling.
  lock_order: acquire buf_lock, then net_lock.

Introduced by: developer in Module B didn't know about Module A's order.
When triggered: rare race between a network burst and buffer GC.
Symptom: server hangs after ~72 hours of load testing.
          CPU at 0%, no progress, no error messages.
Detection: lockdep (in dev build) caught it on first test run:
           "WARNING: possible circular locking dependency detected"
           buf_lock → net_lock vs. net_lock → buf_lock.
Fix: standardize to always acquire net_lock before buf_lock everywhere.
     Added comment in both modules explaining the ordering rule.

Lesson: deadlock bugs are often caused by independent code
        making independent reasonable decisions that conflict.
        Global lock ordering must be documented and enforced.
```

📎 A.7 — Concurrency Sanitizers & POSIX Error Codes

A.7.1 · Atomicity Violations: Pattern and MySQL Case

Atomicity violation: *a code region assumed to execute as an atomic unit is interrupted by another thread, invalidating the assumption.*

```
// Pattern:  Thread 1:  CHECK(x)   [preempted]   USE(x)
//           Thread 2:              MODIFY(x)
// MySQL bug (simplified):
if (thd->proc_info) {                       // (1) check
    fputs(thd->proc_info, ...);             // (2) use
}
thd->proc_info = NULL;                      // (3) Thread 2: between (1) and (2) -> crash

// Fix: Thread 1 holds thd->lock around (1)+(2); Thread 2 holds it around (3).
```

A.7.2 · Order Violations: Pattern

> ***Order violation:*** *the desired order between two threads (A before B) is not enforced, so B can run before A is ready.*

```c
int *data = NULL;                          // shared
void *init_thread(void *arg)   { data = malloc(sizeof(int)); *data = 42; return NULL; }  // (1)
void *worker_thread(void *arg) { printf("%d\n", *data);                  return NULL; }  // (2)
// If the worker runs first: data is NULL -> segfault.
// Mozilla variant: a helper thread read mThread before main finished
//   mThread = PR_CreateThread(...).
```

The programmer assumes (1) before (2). A mutex cannot enforce *which* comes first.

A.7.3 · Fixing an Order Violation: Semaphore Initialised to 0

```
sem_t ready;                        // sem_init(&ready, 0, 0) in main, before creating threads

void *init_thread(void *arg) {
    data = malloc(sizeof(int));  *data = 42;
    sem_post(&ready);               // "initialization is complete"
    return NULL;
}
void *worker_thread(void *arg) {
    sem_wait(&ready);               // blocks until init has posted
    printf("%d\n", *data);          // safe now
    return NULL;
}
```

The ordering pattern of 3.4 (parent-child join). It works even if `init_thread` posts first: the semaphore keeps the value 1, so the later `sem_wait` returns at once.

A.7.4 · POSIX Thread Concurrency Error Codes (`errno`)

```
pthread_mutex_lock():
  EDEADLK    — Lock already held by calling thread (ERRORCHECK mutex only).
  EINVAL     — Mutex not properly initialized.
  EOWNERDEAD — Mutex owner died (robust mutex only).

pthread_mutex_trylock():
  0          — Lock acquired successfully.
  EBUSY      — Lock already held by another thread.

pthread_mutex_timedlock():
  ETIMEDOUT  — Lock not acquired within the specified timeout.
               Useful for deadlock detection with a time bound.

sem_wait():
  EINTR      — Interrupted by a signal (retry after checking EINTR).

sem_timedwait():
  ETIMEDOUT  — Semaphore not decremented within the timeout.
               Useful for detecting potential deadlock in sem-based code.
```

A.7.5 · Detecting Data Races with ThreadSanitizer (TSan)

```
ThreadSanitizer (TSan): a dynamic data-race and lock-order detector.

Supported: GCC and Clang/LLVM; Linux and macOS.
Usage:
  gcc   -fsanitize=thread -g program.c -o program
  clang -fsanitize=thread -g program.c -o program

What it detects:
  - Data races: two threads access the same memory, at least one write,
                no synchronization. Reports: variable, file, line number.
  - Lock-order inversions: potential deadlocks due to inconsistent ordering.
  - Misuse of pthread API (unlock without lock, etc.).

Example output (data race):
  WARNING: ThreadSanitizer: data race
    Write of size 4 at 0x... by thread T1:
      #0 increment counter.c:15
    Previous read of size 4 at 0x... by main thread:
      #0 main       counter.c:30

Overhead: ~5–15× slowdown, ~5–10× memory overhead.
          Only for testing — not production.
```

A.7.6 · Detecting Lock Inversions with Helgrind and DRD

```
Helgrind:
  Valgrind tool for detecting synchronization errors.
  Detects: data races, lock-order violations, pthread API misuse.
  Usage: valgrind --tool=helgrind ./program
  Output: "Thread #1 is trying to acquire lock at ..."
          "But lock was previously acquired in opposite order in thread #2 ..."
  Overhead: very high (10–100× slowdown): thorough but slow.

DRD (Data Race Detector):
  Similar to Helgrind but with lower memory overhead.
  Better for programs with many threads.
  Usage: valgrind --tool=drd ./program

Static analysis tools:
  Clang Static Analyzer: finds potential deadlocks without running the code.
  Coverity: commercial static analyzer; used by many OS projects.
  LLVM's ThreadSafetyAnalysis: annotations in code → compile-time checks.
    e.g.:  int counter GUARDED_BY(mutex);
           void increment() EXCLUSIVE_LOCKS_REQUIRED(mutex) { counter++; }
```

📎 A.8 — Decision Framework, Cheat Sheet & Common Mistakes

A.8.1 · Decision Framework: Prevention vs. Avoidance vs. Detection

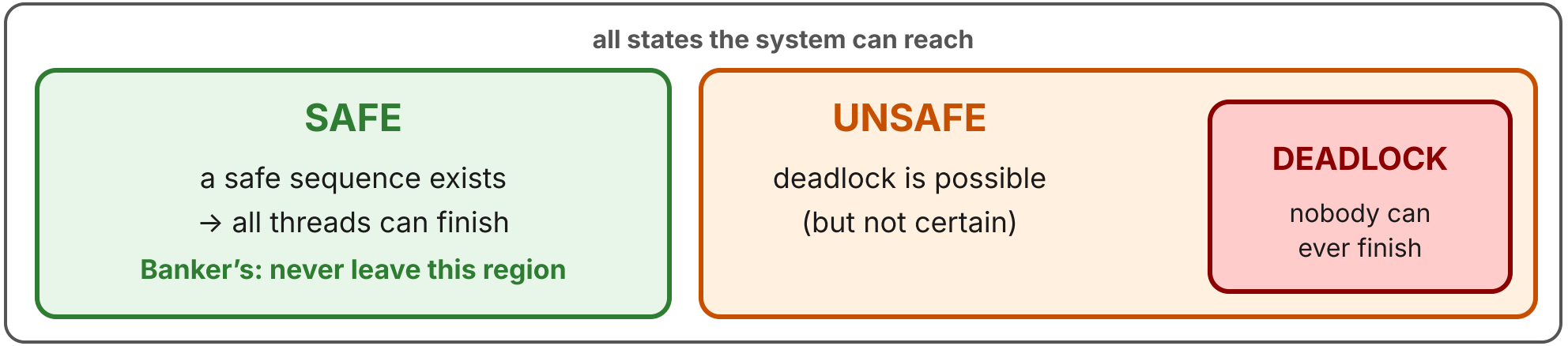
```
Use PREVENTION (lock ordering) when:
  → Code is modular and lock order can be determined at design time.
  → Critical section is simple and resources are known.
  → Best practice for all application-level multithreaded code.

Use AVOIDANCE (Banker's) when:
  → Resource needs are known in advance and fixed.
  → System is embedded or real-time with bounded threads.
  → Academic/exam context. :-)

Use DETECTION + RECOVERY when:
  → Resources are dynamic (DB rows, network connections).
  → Deadlock is acceptable occasionally; recovery is cheap (rollback).
  → Databases: PostgreSQL, MySQL, Oracle all do this.

Use OSTRICH when:
  → General-purpose OS kernel or application.
  → Deadlock is extremely rare; complexity of prevention outweighs benefit.
  → Complement with lockdep (dev builds) to catch violations early.
```

A.8.2 · Safe, Unsafe and Deadlocked States



Banker's Algorithm keeps the system inside the **safe** region: from an unsafe state deadlock *may* follow, but it is not certain.

A.8.3 · Coffman Conditions Cheat Sheet

```
Deadlock requires ALL four:

  1. Mutual Exclusion:  resource is exclusive → one holder at a time.
  2. Hold and Wait:     hold ≥1 resource while waiting for more.
  3. No Preemption:     held resources cannot be forcibly taken.
  4. Circular Wait:     chain T1→T2→...→T1 of waiting threads.

Elimination strategies:
  1. Share resources (read-only, lock-free atomics).
  2. Acquire all-or-nothing; release before requesting more.
  3. Allow preemption (trylock + backoff / rollback).
  4. Enforce total lock ordering (most practical).
```

A.8.4 · Common Mistakes to Avoid

Mistake	Correct approach
Assuming a cycle in a multi-instance RAG means deadlock	A cycle is necessary, not sufficient: run the detection algorithm
Treating an unsafe state as a deadlock	Unsafe only means deadlock is *possible*
Comparing $\mathrm{Need} \leq \mathrm{Work}$ on sums	Every component must satisfy the inequality
Mixing up Need and Request	Banker's uses future **Need**; detection uses the current **Request**
Threads with no allocation in detection	They start with $\mathrm{Finish}[i] = \mathrm{true}$
`wait(mutex)` before `wait(resource)` (3.4)	`wait(resource)` first: never wait while holding the mutex
`trylock` retry without randomised backoff	Livelock is possible
Thinking Ostrich = bad engineering	A legitimate trade-off when deadlock is rare

A.8.5 · Recommended Exercises

1. **Two-lock deadlock:** write and compile the minimal example. Verify it hangs. Apply lock ordering. Verify it no longer hangs.
2. **Banker's Algorithm:** take the example of this lecture, change one Allocation row, and run the safety algorithm and two resource requests by hand.
3. **Dining Philosophers:** implement 5 philosophers with the asymmetric solution of 3.4. Run it for 30 seconds and verify nobody starves.
4. **Atomicity violation:** write a check-then-use bug, run it under ThreadSanitizer (`-fsanitize=thread`), observe the report, fix it.
5. **Order violation:** write a two-thread program where one thread sets a global and the other uses it; add a semaphore to enforce the order; verify with TSan.

📎 A.9 — Banker's Algorithm: Pseudocode & Complexity

A.9.1 · The Safety Algorithm in Full

Notation: n threads, m resource types; $X \leq Y$ means $X[j] \leq Y[j]$ for **every** j.

```
SAFETY(Available[m], Allocation[n][m], Need[n][m]):
    Work[0..m-1] <- Available                  // copy:                O(m)
    Finish[0..n-1] <- false;  seq <- [ ]       //                      O(n)
    repeat                                     // at most n rounds
        found <- false
        for i <- 0 to n-1:                     // scan for a candidate
            if Finish[i] == false and Need[i] <= Work:     // up to m comparisons
                Work <- Work + Allocation[i]   // Ti runs to completion, releases all
                Finish[i] <- true;  append i to seq
                found <- true
                break                          // rescan: Work has grown
    until found == false
    return ( all Finish[i] == true,  seq )     // SAFE iff everyone could finish
```

On the lecture's example, scanning from T_0 each time gives $\langle T_1, T_3, T_0, T_2, T_4 \rangle$: a different safe sequence from slide 27, same verdict.

A.9.2 · The Resource-Request Algorithm in Full

```
REQUEST(i, Request[m]):                          // thread Ti asks for Request
    if Request > Need[i]:       error             // more than its declared maximum   O(m)
    if Request > Available:     return WAIT       // not enough free right now        O(m)

    Available     <- Available     - Request      // pretend to grant                 O(m)
    Allocation[i] <- Allocation[i] + Request
    Need[i]       <- Need[i]       - Request

    (safe, seq) <- SAFETY(Available, Allocation, Need)                          // O(m n^2)
    if safe:  return GRANT                       // keep the new state
    else:     undo the three updates;  return WAIT   // state unchanged           O(m)
```

- **Only a safe state is ever kept:** the system never leaves the safe region.
- A *waiting* thread simply asks again later; no state was changed by the denied request.
- Lecture example: T_1 asks $(1, 0, 2) \Rightarrow$ safe, **GRANT**. T_4 asks $(3, 3, 0) \Rightarrow$ pretend state has no eligible thread, **WAIT**.

A.9.3 · Why Any Eligible Thread Can Be Picked

The safety algorithm is **greedy**: it takes *any* thread with $\text{Need}[i] \leq \text{Work}$. Why does the pick order not matter?

1. **Finishing a thread never hurts:** Work only grows ($\text{Work} + \text{Allocation}[i] \geq \text{Work}$), so a thread that is eligible now stays eligible later.
2. **Claim:** if *some* safe sequence S exists, greedy finds a complete one.
3. **Proof sketch:** suppose greedy gets stuck with unfinished set $U \neq \emptyset$ where nobody is eligible. Let T_j be the first thread of S that is in U. Every thread before T_j in S has finished in greedy, so greedy's Work is at least what T_j has in S. Hence $\text{Need}[j] \leq \text{Work}$: T_j *is* eligible. Contradiction.

> ***Consequence:*** *one pass of the greedy search decides safety; no backtracking over orders is needed.*

A.9.4 · Time and Space Complexity

Cost	Bound	Why
Space: system state	$O(nm)$	Max, Allocation (and Need $=$ Max $-$ Allocation) are $n \times m$ matrices; Available is m
Space: SAFETY extra	$O(n+m)$	Work (m), Finish (n), the sequence (n)
Time: SAFETY	$O(m\,n^2)$	at most n rounds; a round examines up to n threads; each examination compares $\leq m$ components
Time: REQUEST	$O(m\,n^2)$	two $O(m)$ checks, $O(m)$ pretend-grant and undo, plus one SAFETY call

- **Tighter count:** rounds examine $n, n-1, \ldots, 1$ threads, so at most $m\,\frac{n(n+1)}{2}$ comparisons (reachable): still $\Theta(m\,n^2)$.
- **Paid on every request:** $n = 1000$, $m = 10$ gives about $5 \cdot 10^6$ comparisons per request; with the up-front maxima, this is why general-purpose kernels do not run it.
- **Detection (A.5.1)** has the same shape ($\mathrm{Request}$ instead of Need): also $O(m\,n^2)$.