

Deadlocks and resource allocation

CSE513 · Week 4

Prof. Mehmet Göktürk

When can processes wait forever,
and what can we do about it?

The two halves of this lecture

First half: identify a deadlock, read resource graphs and explain prevention through lock ordering.

Second half: work through safe states, Banker's algorithm, detection and recovery.

Practice: distinguish a proof about a protocol from evidence obtained in one execution.

Exclusion does not guarantee progress

Week 3 protected shared state with mutexes, semaphores and monitor predicates.

A perfectly exclusive lock can still participate in a system where nobody can move.

Ask both: who is allowed to enter, and who must act before a waiting thread can proceed?

Two locks, opposite acquisition orders

Both locks initially available. Each worker needs both before doing its update.

Thread A	Thread B
lock(L1)	lock(L2)
lock(L2)	lock(L1)
update	update
unlock(L2)	unlock(L1)
unlock(L1)	unlock(L2)

If each holds its first lock, neither can obtain its second. Their release code is unreachable.

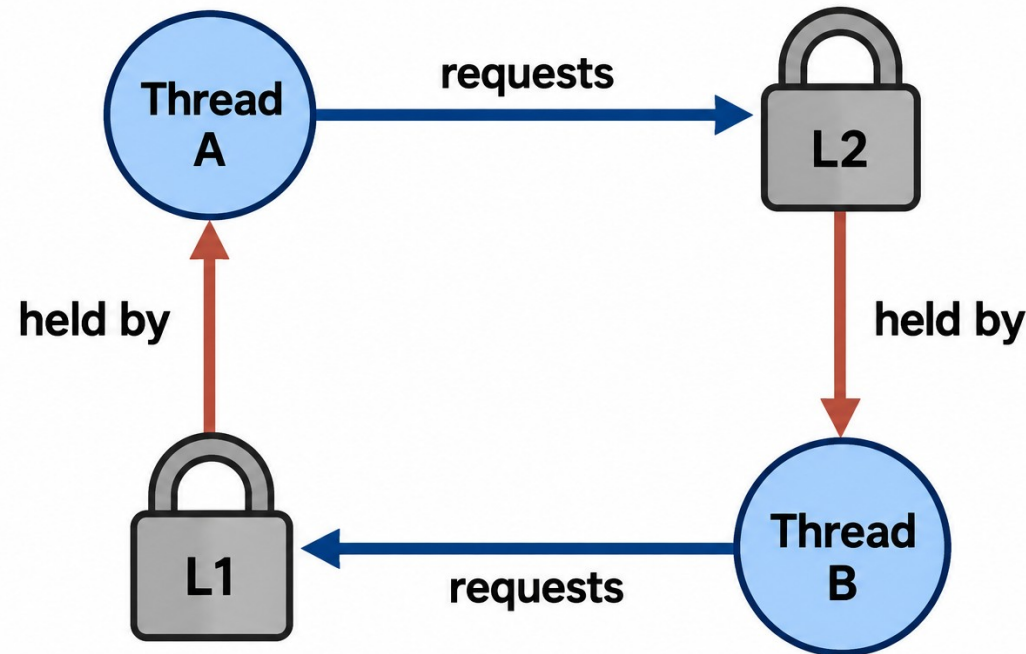
An interleaving that cannot finish

A holds L1; B holds L2. Both now make blocking requests for their second lock. More CPU time does not make either lock available: the required release depends on a blocked worker.

Step	Thread A	Thread B
1	Acquires L1	
2		Acquires L2
3	Waits for L2	
4		Waits for L1

The two-lock dependency cycle

Two locks, circular wait



A holds L1. B holds L2. Neither can reach its release.

Request arrows point to locks. Ownership arrows point to holders. Trace the complete cycle.

Resource types and instances

A resource type can have one or several interchangeable instances: one mutex, three identical devices, or a pool of connections.

The basic lifecycle is request, use, release. A request may block when its required instances are unavailable.

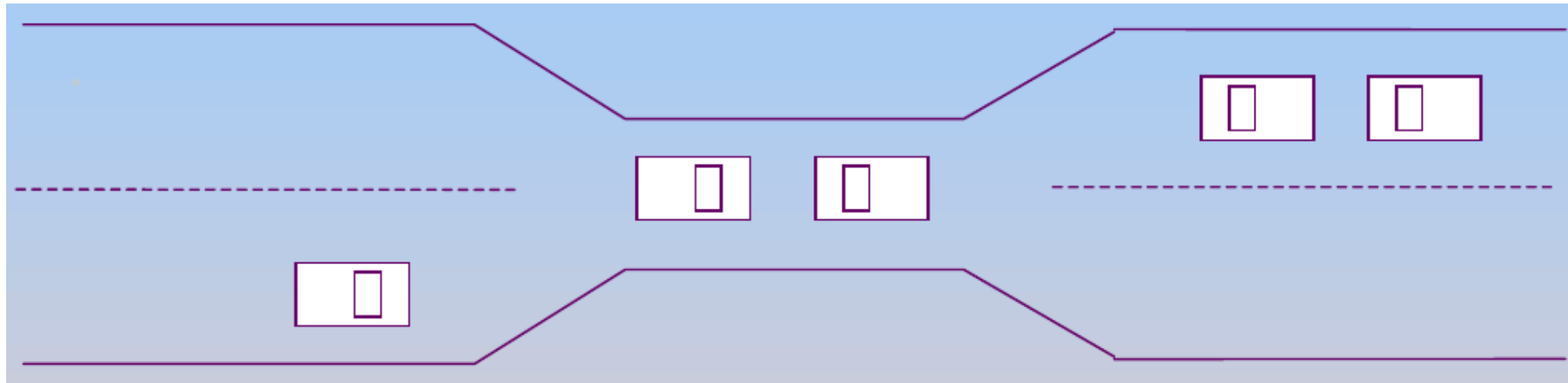
For the textbook model, resources are reusable and are released after completion. External intervention is excluded when deciding whether this state is deadlocked.

Bridge crossing: road sections as resources

Each occupied road section is a resource. Opposing cars can hold their sections while waiting for space ahead.

Backing cars out releases sections and rolls back progress. Several cars may have to retreat.

Admitting one direction at a time prevents this conflict. Fair turn-taking is still needed to avoid starvation.



Silberschatz, Galvin & Gagne · OSC 2002 · Slide 8.3

Four necessary conditions

Mutual exclusion: at least one relevant resource cannot be shared concurrently.

Hold and wait: a thread keeps an allocated resource while awaiting another.

No preemption: the resource cannot simply be taken from its current holder.

Circular wait: a closed chain of threads waits for resources held by the next member.

Mutual exclusion

A mutex has one owner at a time. A shared read-only object need not impose the same restriction.

The condition concerns resource semantics, not whether the machine has one CPU or many.

Removing locks without replacing the protection can remove waiting while introducing data corruption.



AI-generated illustration · Instructor bathroom-key analogy

A worker owns L1 while it asks for L2. The first resource is unavailable to others during the wait.

Requesting all required resources together can remove this pattern, provided the reservation itself is coordinated.

Waiting for a free connection while holding a database or application lock can create the same dependency across layers.

No preemption of the protected operation

The CPU scheduler can preempt a thread without releasing its mutex. Processor preemption and resource preemption are different.

Taking a lock away does not restore the half-updated data it protects.

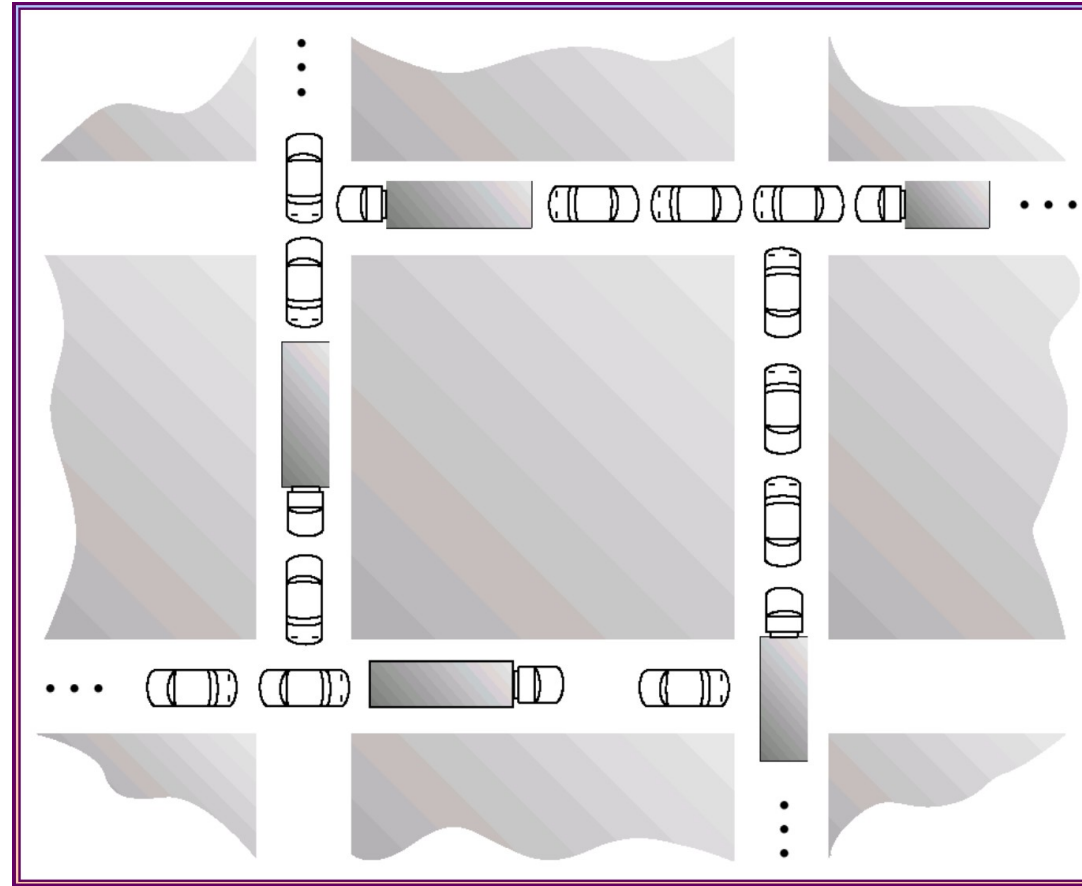
Preemption is practical only when the resource state can be saved, restored or safely rolled back.

A waits for a resource held by B; B waits for one held by C; C waits for one held by A.

A circular chain explains why each required release depends on progress elsewhere in the same chain.

All four conditions are necessary for deadlock in this model. Merely permitting them in a design does not mean every execution deadlocks.

Traffic deadlock in a street grid



Each vehicle holds road space and waits for space ahead. Trace the circular wait. Keeping an exit clear can prevent the gridlock.

Deadlock: a set of participants cannot progress because the events they need depend on blocked members of that set.

Starvation: one participant waits indefinitely while other work may continue, for example because admission always favors newcomers.

Breaking circular wait prevents this class of deadlock; it does not automatically provide bounded waiting or fair scheduling.

Two contenders may repeatedly release, retry and collide again. Their states change, but useful work never completes.

A try-lock failure followed by immediate synchronized retries can avoid blocking deadlock yet still fail to make progress.

Random backoff can reduce repeated collisions; a complete progress argument still needs an admission rule and stated scheduling assumptions.

Reading a resource-allocation graph

Circles are threads. Rectangles are resource types; dots represent individual instances.

$T \rightarrow R$ means a current request. $R \rightarrow T$ means an allocated instance. Read arrow direction before interpreting a cycle.

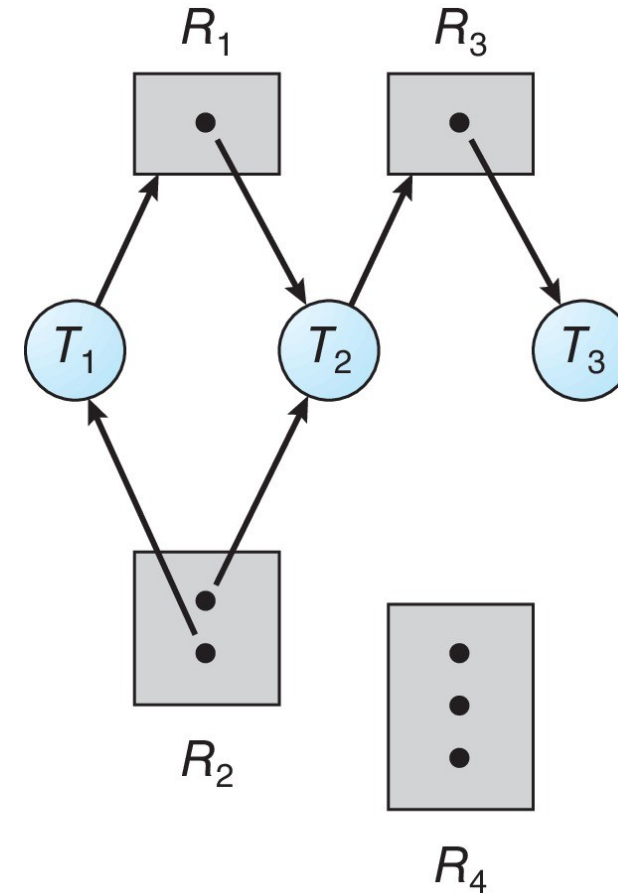
The graph is a snapshot of dependencies, not a CPU scheduling timeline.

A graph in which progress is possible

T1 holds one R2 and requests R1. T2 holds R1 and the other R2, and requests R3.

T3 holds R3 and has no outgoing request. It can finish and release R3, enabling T2 and then T1.

R4 is unused. Its three free instances do not substitute for a missing R1 or R3.

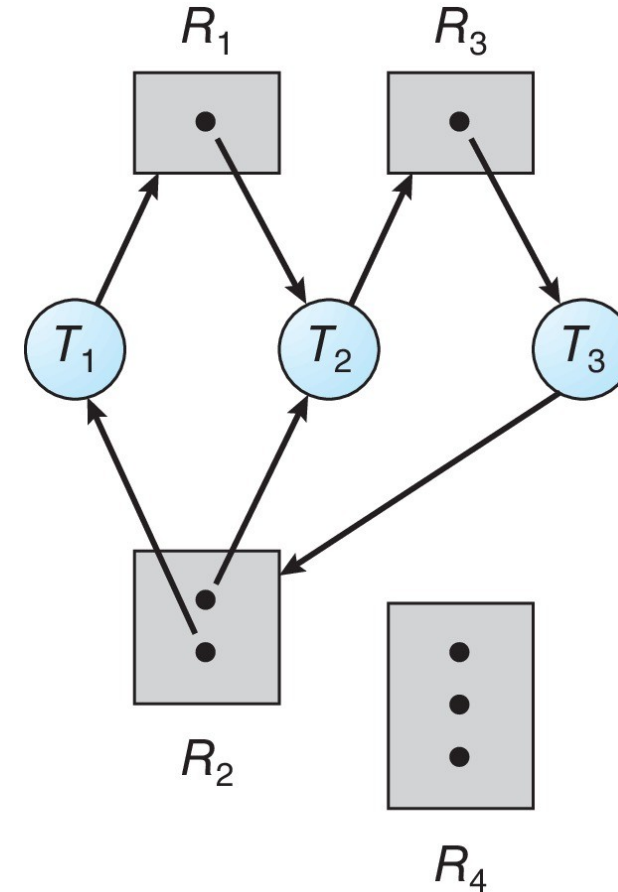


Adding one request creates deadlock

Now T3 requests R2. Both R2 instances are held by T1 and T2.

T1 waits for T2's R1; T2 waits for T3's R3; T3 needs an R2 release from this blocked set.

Identify the closed dependency set, rather than concluding from the existence of any free resource.

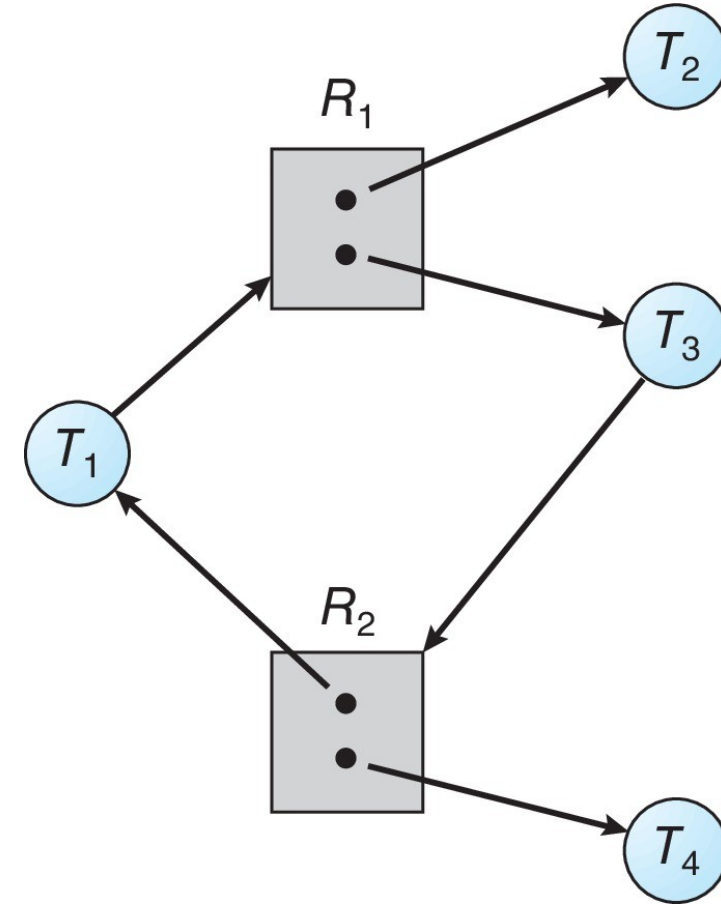


A cycle without deadlock

The cycle is $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$. But each resource type has two instances.

T_2 or T_4 can finish without waiting on that cycle. Their release can enable a blocked request and dissolve the cycle.

With multiple instances, a cycle alone is not a sufficient deadlock test.



What a cycle establishes

No cycle in the current reusable-resource graph: no resource deadlock in this model.

One instance per resource type: a cycle of current requests and allocations establishes deadlock.

Several instances: a cycle indicates possible deadlock; inspect counts and potential completion.

A dashed future claim is not a currently blocked request. Avoidance graphs require a different interpretation.

Graph exercise: decide before intervening

State A: one X and one Y. A holds X and requests Y; B holds Y and requests X.

State B: two X and one Y. A holds one X and requests Y; B holds Y and requests X; C holds the other X and requests nothing.

For each state, identify a cycle and decide whether an outside completion can break the wait.

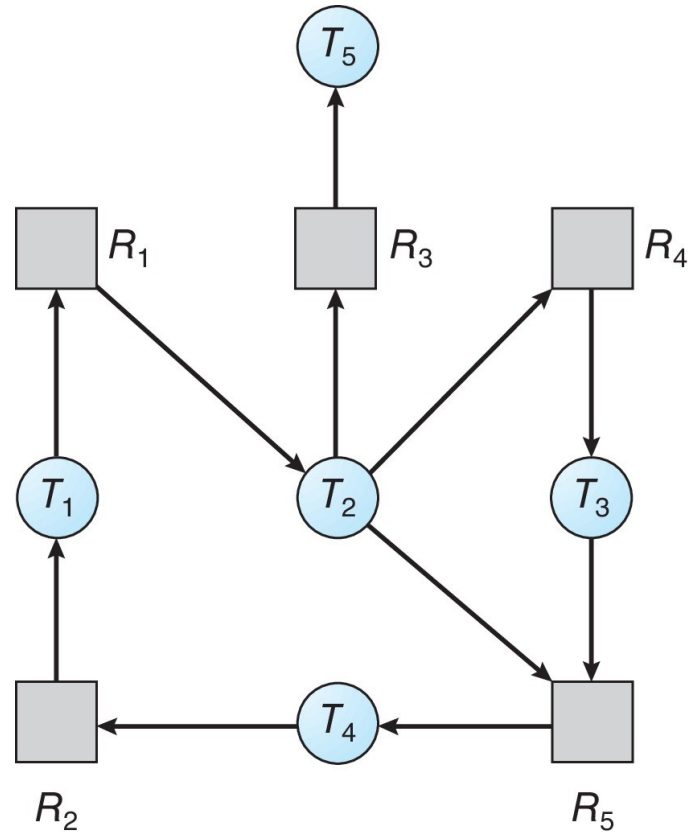
Graph exercise: worked answer

State A is deadlocked: neither A nor B can reach a release under the stated protocol.

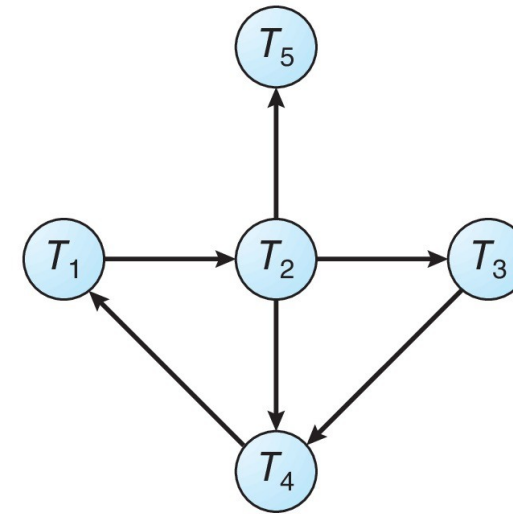
State B need not be deadlocked. C can finish and release X; B obtains X, completes and releases Y; A can then complete.

A cycle is present in both states. The second X instance changes the conclusion.

Resource graph to wait-for graph



(a)



(b)

Single-instance resources: replace $T_i \rightarrow R \rightarrow T_j$ with $T_i \rightarrow T_j$.
The cycle $T_1 \rightarrow T_2 \rightarrow T_4 \rightarrow T_1$ remains even if T_5 finishes.

Dining philosophers revisited

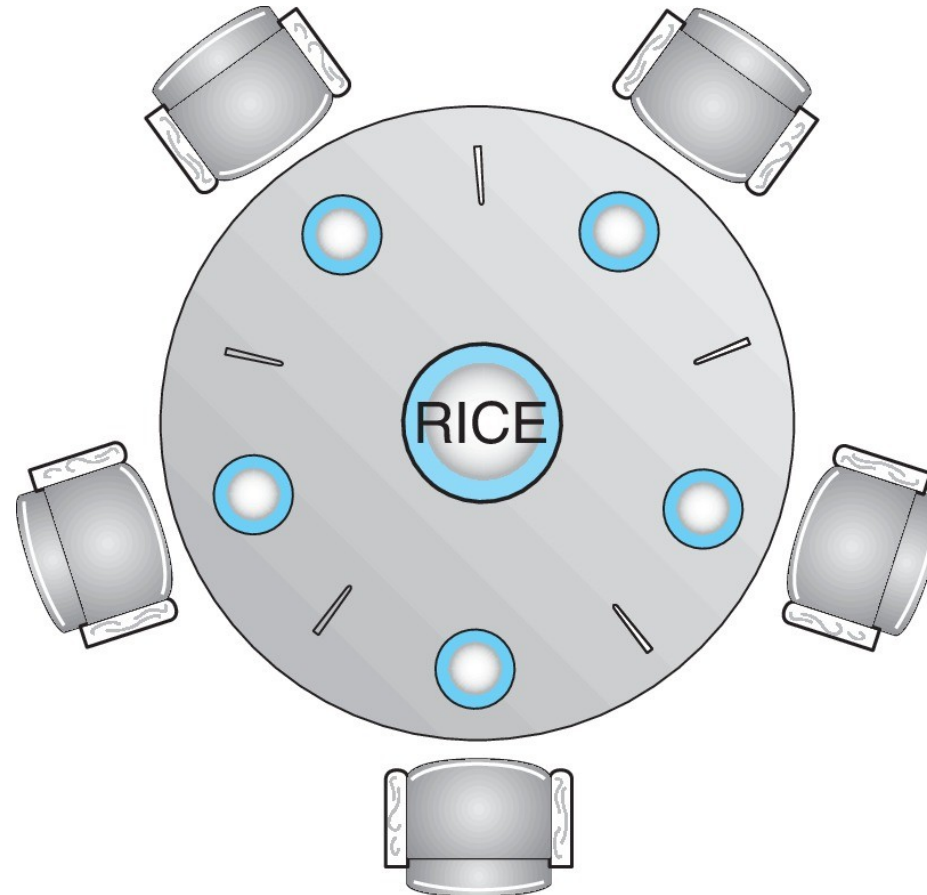
Each utensil has one instance. If all five philosophers hold one and wait for the other, all four deadlock conditions hold.

Number utensils 0–4. Acquire the lower number first, then the higher; release both after eating. This removes circular acquisition.



AI-generated illustration · Instructor classroom analogy

The textbook's shared-utensil model



If every philosopher holds one chopstick and waits for the other, no one can finish eating.

Four approaches to deadlock

Approach	Decision	Main cost
Prevention	Forbid a necessary condition	Restrictions on access
Avoidance	Grant only if state stays safe	Maximum-demand information
Detection + recovery	Allow, detect, then repair	Monitoring and lost work
Accept the risk	No general mechanism here	Application must handle failures

Prevention by sharing or reservation

Remove mutual exclusion only where sharing is valid: immutable data or a suitable spooler can change the resource model.

Remove hold-and-wait by reserving the complete set before starting, or releasing current resources before requesting a new set.

Whole-set reservation may leave resources idle and can starve a large request. Removing one condition has a cost.

Reservation must itself be atomic

“Check that X is free; then take Y; then take X” is not an atomic whole-set reservation.

Use a coordinator or a protocol that prevents partial reservations from accumulating into another wait cycle.

Resources needed later may be unknown. Reserving a maximum set for an entire job can sacrifice substantial concurrency.

Prevention through release and rollback

If a new request cannot be met, release previously held resources and retry only when the required set can be obtained.

This works only if partial work is reversible or has not become externally visible. A sent message or physical action may not be undoable.

Repeated rollback can cause starvation. Track retry cost and who repeatedly loses.

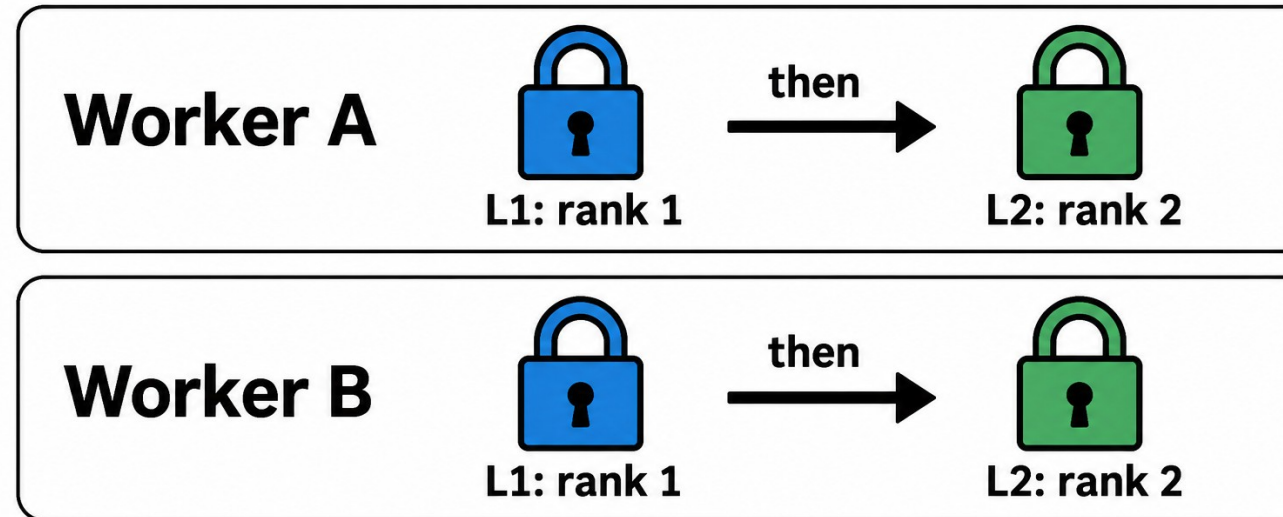
Why a global lock order works

Give every relevant lock a unique rank. Acquire only in increasing rank while holding other locks.

A cycle would require $\text{rank}(L1) < \text{rank}(L2) < \dots < \text{rank}(Lk) < \text{rank}(L1)$, which is impossible.

The proof assumes every path follows the same order, including callbacks and error paths. It does not prove fairness.

One shared lock order



$$1 < 2$$

Both workers request lower rank first.

These rows describe acquisition plans. They do not show simultaneous ownership of L1.

The textbook's conflicting lock orders

The first worker takes `first_mutex` then `second_mutex`; the second worker reverses that order.

When both hold their first lock, each blocks on the other. Fix the second acquisition sequence to follow the shared ranking.

This retained fragment omits error handling; use the checked practice program for execution.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

Ordering dynamically chosen resources

For a transfer between accounts, acquire the lock of the smaller stable account ID before the larger, regardless of transfer direction.

If source and destination are the same account, handle it explicitly; do not acquire a nonrecursive lock twice.

Keep ordering rules in shared helpers. Inconsistent local “exceptions” invalidate the global proof.

Try-lock and backoff

trylock returns without blocking if the next lock is unavailable. On failure, release the first lock before a retry.

This removes the permanent hold-and-wait cycle in that attempt, provided the release actually occurs.

Bounded retries, asymmetry or an admission policy may be needed for progress. Retrying forever is not a fairness guarantee.

A timeout is evidence, not diagnosis

A timeout proves that an operation did not finish by a chosen deadline. Slow computation, starvation and external I/O can also time out.

Diagnose deadlock by reconstructing owners, requests and the events required for release.

A watchdog may stop a demonstration process; that is not a general method for repairing shared application state.

First-half checkpoint

Draw the two-lock state after each acquisition. Mark current ownership and current requests.

Explain why a cycle proves deadlock with one instance, but can be harmless with two.

State a global lock order and identify exactly which necessary condition it removes.

Avoidance uses future-demand information

Prevention changes the permitted protocol. Avoidance inspects each proposed allocation against a declared maximum demand.

Available resources are not enough: granting them may leave no guaranteed completion sequence.

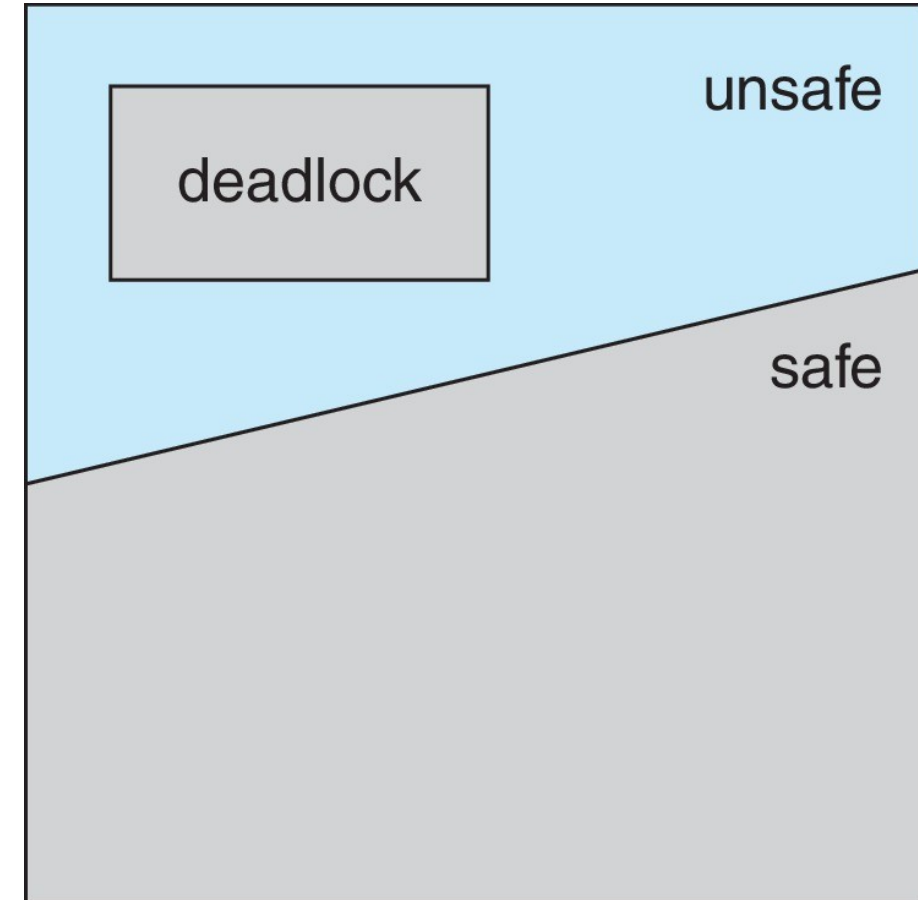
Maximum claims must be accurate, resources interchangeable within a type, and completion must release resources in finite time.

Safe, unsafe and deadlocked states

Safe: at least one order can satisfy each thread's remaining maximum demand and allow all to finish.

Unsafe: that guarantee is absent. The actual future may use fewer resources and still finish.

Every deadlocked state is unsafe; an unsafe state need not already be deadlocked.



A small safe-state example

One type has 10 units. A holds 4 and may need at most 6 in total; B holds 3 and may need at most 8.
Available = 3.

A needs at most 2 more, so it can finish first. Its completion leaves 7 available; B's remaining need of 5 can then be met.

The sequence A, B is a witness of safety. It is a possible completion order, not a promised CPU schedule.

Ten reusable units

Available: 3



A holds 4



**Needs at most
2 more**

B holds 3



**Needs at most
5 more**

Maximum totals: A = 6, B = 8

A can borrow 2, finish, and return 6. The pool then contains 7, enough for B's remaining 5.

Available does not always mean safe

From that same baseline, tentatively give B two more units. Available = 1; A still needs 2 and B may still need 3.

Neither can be guaranteed to finish from one available unit. The tentative state is unsafe.

It is not necessarily already deadlocked: B could finish without requesting its full maximum. Banker's algorithm nevertheless refuses this grant.

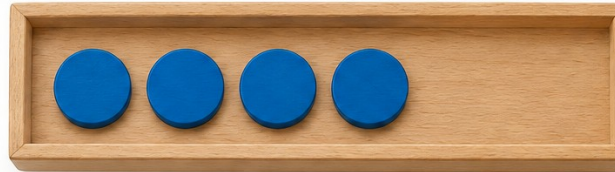
After lending two more units to B

Available, but unsafe

Available: 1



A holds 4



Needs at most 2 more

B holds 5



Needs at most 3 more

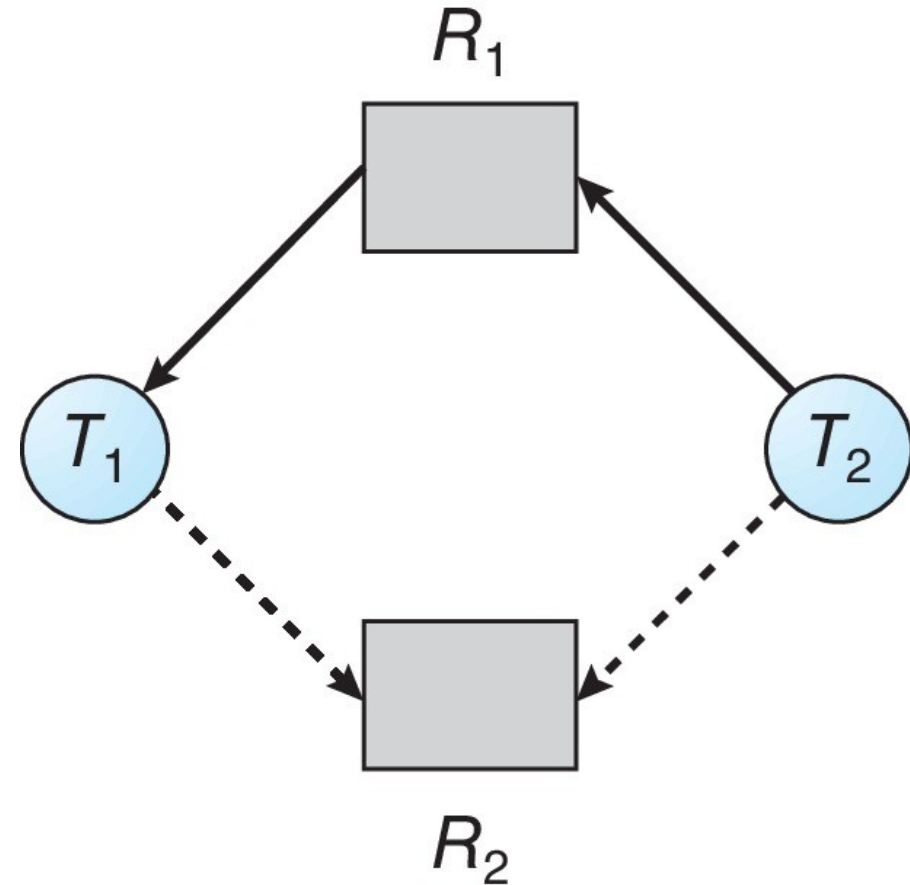
Neither maximum remaining need fits the free pool.

One free unit cannot satisfy either maximum remaining need. This is unsafe, not necessarily deadlocked.

Future claims in a single-instance graph

R_1 is assigned to T_1 ; T_2 currently requests R_1 .
Dashed edges to R_2 describe possible future requests.

The avoidance model requires advance claims.
A dashed edge is not evidence that the thread is blocked now.

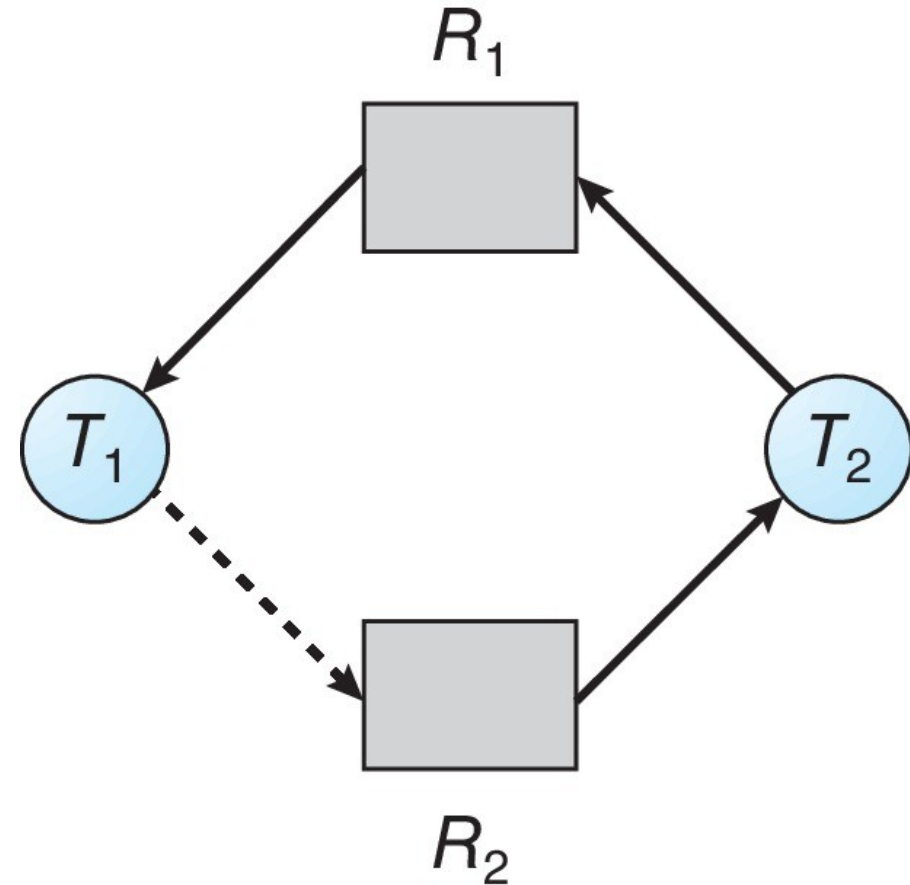


An unsafe tentative assignment

Assigning R_2 to T_2 closes a cycle containing T_1 's future claim on R_2 .

T_1 may later request R_2 while T_2 waits for R_1 .
The avoidance rule rejects the tentative allocation.

Distinguish a potential dependency cycle including claims from a deadlock of current requests.



Banker's algorithm: the snapshot

Three resource types A, B, C. Totals = (10, 5, 7); Available = (3, 3, 2). All vectors use this component order.

Thread	Allocation A B C	Max A B C
T0	0 1 0	7 5 3
T1	2 0 0	3 2 2
T2	3 0 2	9 0 2
T3	2 1 1	2 2 2
T4	0 0 2	4 3 3

Compute Need = Max – Allocation

Need is the maximum additional demand, not the total claim and not the currently outstanding request.

Thread	Max	Allocation	Need
T0	7 5 3	0 1 0	7 4 3
T1	3 2 2	2 0 0	1 2 2
T2	9 0 2	3 0 2	6 0 0
T3	2 2 2	2 1 1	0 1 1
T4	4 3 3	0 0 2	4 3 1

Set Work = Available and mark all threads unfinished.

Find an unfinished thread whose Need is no larger than Work in every component. Pretend it completes; set $Work = Work + Allocation[i]$.

Repeat until every thread is finished, or no unfinished thread qualifies. A complete sequence proves safety; getting stuck means unsafe.

Work is scratch data. This simulation neither runs threads nor changes live allocation.

A safe completion sequence

Choose T1, T3, T4, T2, T0. Each row shows Work after the selected thread finishes.

Step	Thread and remaining need	Work after completion
Start	—	3 3 2
1	T1: Need 1 2 2	5 3 2
2	T3: Need 0 1 1	7 4 3
3	T4: Need 4 3 1	7 4 5
4	T2: Need 6 0 0	10 4 7
5	T0: Need 7 4 3	10 5 7

Why Work gains Allocation, not Max

For T1, Work starts at (3, 3, 2), Need is (1, 2, 2), and Allocation is (2, 0, 0).

Lend its remaining need, then reclaim everything at completion: $\text{Work} - \text{Need} + \text{Max} = \text{Work} + \text{Allocation}$.

So the new Work is (5, 3, 2). Adding Max directly would invent the units just lent to T1.

When all threads finish, Work must equal the total resource vector (10, 5, 7).

Checking a new request

1. $\text{Request} \leq \text{Need}$? Otherwise it exceeds the declared maximum.
2. $\text{Request} \leq \text{Available}$? Otherwise it must wait for resources.
3. Tentatively subtract Request from Available, add it to Allocation, and subtract it from Need.
4. Run the safety test. Commit only if safe; otherwise restore the original state. Serialize this decision with other allocation decisions.

T1 requests (1, 0, 2)

Use the original five-thread snapshot. The request is within both Need and Available.

Tentative Available = (2, 3, 0); T1 Allocation = (3, 0, 2); T1 Need = (0, 2, 0).

T1 can finish first. A full safe sequence still exists, so this request can be granted.

Other rows are unchanged. The test must use the tentative state, not the old one.

Safety after granting T1's request

Sequence T1, T3, T4, T0, T2 is safe. Each completion returns its current allocation.

Step	Completion	Work
Start	—	2 3 0
1	T1	5 3 2
2	T3	7 4 3
3	T4	7 4 5
4	T0	7 5 5
5	T2	10 5 7

An unavailable request

Continue from the state after granting T1. Available is (2, 3, 0).

T4 requests (3, 3, 0). The request is within its Need of (4, 3, 1), but requires more A than is available.

It must wait at the availability check. No tentative allocation or safety simulation is needed.

Always state which snapshot a request exercise uses; resetting to the original snapshot is a different problem.

An available but unsafe request

Again start after granting T1, without applying T4's refused request. T0 asks for (0, 2, 0).

Tentative Available = (2, 1, 0); T0 Need becomes (7, 2, 3).

No thread's Need fits: T1 lacks B; T3 lacks C; T0, T2 and T4 lack A or other resources.

The safety test cannot start. Refuse the request and restore the state with Available = (2, 3, 0).

Common mistakes in safety calculations

Compare vectors component by component. A surplus of A cannot pay for a shortage of C.

Use remaining Need for avoidance; keep original Allocation for the completion update.

Refusing an unsafe grant does not mean the pre-request state was deadlocked.

A safe sequence is a witness. Several valid witnesses may exist, and choosing another eligible thread is not an error.

Detection asks a different question

Avoidance asks whether all declared future demands can be handled. Detection asks whether current outstanding waits can be resolved.

The detection matrix Request is not Max – Allocation. A thread may currently request less than its possible future need.

The snapshot must be consistent. Resource ownership changing halfway through collection can produce a misleading graph or matrix.

Detection with multiple instances

Set Work = Available. Initially finish threads holding no resources; they cannot retain an instance needed by the set.

Repeatedly find an unfinished holder whose current Request $\leq$ Work; model its completion and add its Allocation to Work.

When no candidate remains, unfinished resource holders form the deadlocked set under the model's completion assumption.

A zero-allocation waiter can still be blocked behind that set. Report affected waiters separately from holders retaining resources.

Detection snapshot

This is a different system: totals (7, 2, 6), Available (0, 0, 0). Request means currently outstanding demand.

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

Zero availability does not prove deadlock

T0 and T2 have no outstanding requests. Model their completion and follow the release chain.

Completion	Work before	Work after
T0	0 0 0	0 1 0
T2	0 1 0	3 1 3
T3	3 1 3	5 2 4
T1	5 2 4	7 2 4
T4	7 2 4	7 2 6

One extra current request changes the result

Change only T2's Request from $(0, 0, 0)$ to $(0, 0, 1)$. Keep the detection snapshot's Allocation and Available.

T0 can finish, giving Work = $(0, 1, 0)$. Every remaining holder needs A or C that cannot be obtained.

Detection stops with T1, T2, T3 and T4 unfinished. They form the deadlocked resource-holding set.

The change is a current blocked request, not merely an increased maximum claim.

Run on blocked requests, periodically, or after a delay: detection frequency trades overhead against accumulated lost work.

A simple repeated-scan matrix algorithm costs $O(m n^2)$. Graph cycle detection can use $O(V + E)$ with adjacency lists.

A detector sees only modeled dependencies. A database lock, callback or remote wait omitted from its view can hide the real cycle.

Cycles can overlap. Recovering one chosen cycle does not guarantee that every cycle has disappeared.

Recovery by terminating work

Aborting the whole deadlocked set is simple but discards the most work. Aborting selected participants requires checking again.

Choose a victim using priority, work already performed, restart cost, resources held and consequences for users.

Stopping an arbitrary thread can leave in-process shared state inconsistent. Prefer an application-defined transaction or task boundary that supports recovery.

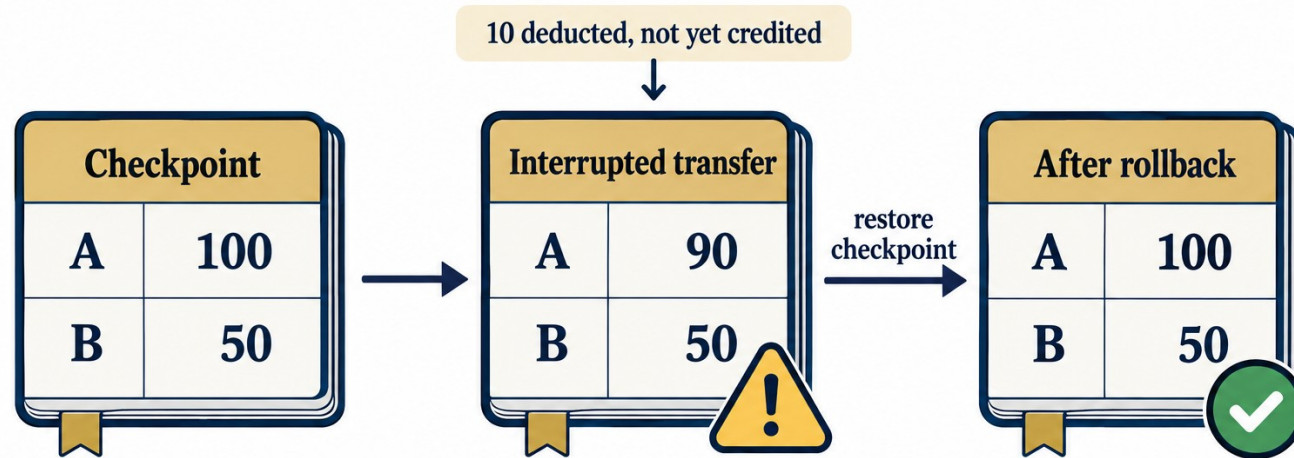
Recovery by preemption and rollback

Select a resource or task that can be reclaimed safely. Restore a consistent checkpoint and restart according to a defined protocol.

Releasing ownership is not enough if partially completed updates remain visible. Coordinate state repair with the release.

Database transaction abort is a useful comparison, but ordinary mutexes do not automatically undo the protected application operation.

Rollback restores a consistent state



Repair data before retrying

Illustrative transfer of 10 from A to B: rollback restores the balances before a controlled retry.

Recovery must not repeatedly punish one task

A cheapest-victim rule can repeatedly choose the same participant and starve it. Include retry or rollback history in the policy.

After each intervention, rebuild the state and confirm that the required waits can now resolve.

Prevent duplicate external effects when retrying. A lost response does not imply that an earlier operation had no effect.

Lock experiment: three modes

deadlock: both workers acquire different first locks, synchronize, then block on the other lock. Run under an external timeout.

ordered: both acquire L1 then L2. No rendezvous occurs while one waits for the first lock.

try: synchronize first ownership, try the second lock once, report EBUSY, then release. It demonstrates failed admission without permanent blocking.

Run and predict on Ubuntu

Compile with warnings and pthread support. The first command should be bounded; the other two should complete.

Before running, draw the expected ownership/request state. Do not infer correctness solely from an exit status.

```
cc -std=c11 -Wall -Wextra -O2 \  
-pthread lock-order.c -o lock-order  
timeout 3s ./lock-order deadlock  
echo $?  
./lock-order ordered  
./lock-order try
```

Interpreting the lock experiment

The deliberate rendezvous ensures both first locks are held before second requests. It creates the adverse state without relying on a sleep.

The ordered mode completes because all acquisition edges follow L1 before L2; a successful run illustrates that argument.

In try mode, both attempts finish before either worker releases its first lock. Each second acquisition therefore reports EBUSY.

The try mode makes one attempt, not an endless retry loop. It does not demonstrate starvation freedom for arbitrary retries.

Record OS/compiler versions, exact commands, output, exit status and the ownership/request graph.

GNU timeout normally reports 124 when its deadline expires. This is a deadline result, not a deadlock detector.

Explain why the deliberate deadlock has no internal release path. Explain why the ordered program cannot close a lock-order cycle.

AI assistance is permitted: use it to challenge your graph and assumptions, then check the claims against source and observations.

Lock dependency checking in Linux

Kernel lockdep tracks dependencies between lock classes and reports inconsistent locking patterns seen through its instrumentation.

An observed ordering dependency can reveal a potential deadlock even if a particular run has not frozen.

This is a kernel debugging facility, not a promise that ordinary pthread mutexes diagnose every application cycle. Coverage and configuration matter.

Safety practice: verify the calculation

Run the provided banker.c checker against the exact five-thread snapshot. Match its sequence to the displayed vectors.

Change one Available component, or make one request, and predict the result before execution.
 $\text{Preserve total} = \text{Available} + \text{allocated instances}.$

Check safe, unsafe, unavailable and above-maximum requests separately. Confirm that a refused request leaves the original state unchanged.

Review: explain the model and the remedy

Why can unused R4 instances coexist with deadlock on R1, R2 and R3? Why does a second instance weaken the cycle test?

Give an unsafe state that is not currently deadlocked. Explain why Banker's algorithm still refuses the risky grant.

Compare removing circular wait, refusing an unsafe request, and aborting a deadlocked transaction. What information and costs differ?

For each answer, state the assumptions, show a graph or vector trace, and identify what the proposed method does not guarantee.

References and the next lecture

Adapted from Silberschatz, Galvin and Gagne, Operating System Concepts, 10th edition, Chapter 8.

POSIX mutex semantics, GNU timeout and Linux lockdep documentation support the practical distinctions. Source links are in the reading and slide notes.

Next: memory management and address translation. Resource accounting continues, but the central object becomes the process address space.