Chapter 7: Deadlocks

Operating Systems

Resources | The four Coffman conditions | Resource-allocation graph | Wait-for graph | The ostrich algorithm | Detection algorithm | Recovery | Safe states | Banker's algorithm | Prevention and lock ordering | 2PL, livelock, starvation | Java, jstack, lockdep

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_07_Deadlocks.html

Learning Outcomes

Definition and resources

Define deadlock; distinguish preemptable from nonpreemptable resources and describe the request–use–release cycle.

Four conditions

Illustrate the Coffman conditions (mutual exclusion, hold and wait, no preemption, circular wait) with an example.

Modeling

Draw resource-allocation and wait-for graphs; state when a cycle means deadlock.

Detection and recovery

Run the matrix-based detection algorithm by hand; compare recovery by preemption, rollback, and termination.

Avoidance

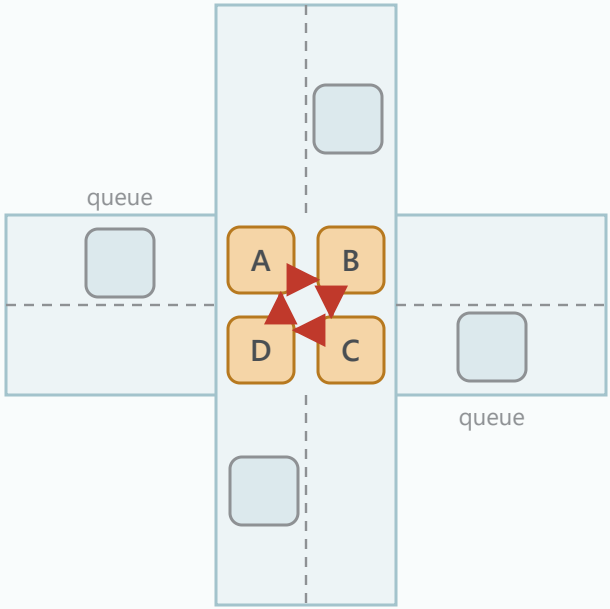
Distinguish safe from unsafe states; use the banker's algorithm to decide whether to grant a request.

Prevention and practice

Recognize lock ordering, two-phase locking, livelock, and starvation; find a deadlock in Java with `jstack`.

What Is a Deadlock?

- **Definition (Tanenbaum):** a set of processes is **deadlocked** if *each* process in the set is waiting for an event that only *another* process in the set can cause.
- The event is usually the **release of a resource**; since none of them runs, none can release anything.
- Example: P1 holds R1 and waits for R2; P2 holds R2 and waits for R1.
- Everyday examples: four cars at an intersection block each other (*gridlock*); two trains facing each other on a single track.
- Deadlocked processes **use no CPU** (they are blocked), but the resources they hold are unusable forever.



It isn't only an OS issue: database transactions, multithreaded applications, and distributed systems deadlock too.

Resources: Preemptable and Nonpreemptable

Preemptable

Can be taken from its owner *without harm*. Examples: the CPU (context is saved), main memory (a page is written to disk). Deadlock can be resolved by reallocating it.

Nonpreemptable

Taking it away breaks the computation. Examples: a Blu-ray being burned or a print job, a `mutex`, a database record lock. Deadlocks involve **this kind** of resource.

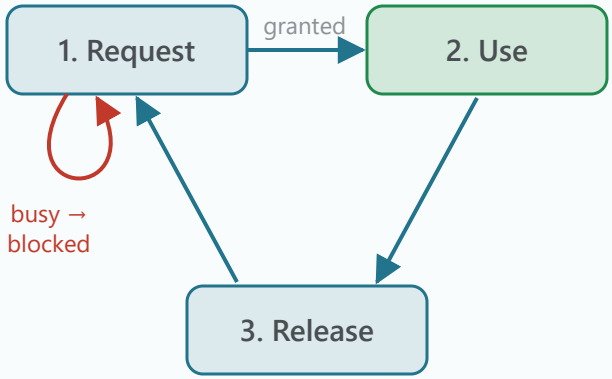
Resource	Instances	Type	Note
CPU core	many	preemptable	handed over by the timer interrupt
Memory page	many	preemptable	reclaimed by swapping/paging
Mutex, semaphore, file lock	1	nonpreemptable	only the owner releases it
Database row lock	1	nonpreemptable*	*released if the transaction aborts

Each resource type R_i may have W_i identical instances; instances are identical if a request can be satisfied by *any* instance of that type.

The Resource-Use Cycle

- **Request:** granted if the resource is free; otherwise the process **blocks** (or gets an error code and retries later).
- **Use:** the process operates on the resource.
- **Release:** the resource becomes free and one of the waiters is woken.
- The OS keeps a **system table** recording which resource is allocated to whom and who is waiting.

Request	Release
`open()`	`close()`
`malloc()` / `mmap()`	`free()` / `munmap()`
`pthread_mutex_lock()`	`pthread_mutex_unlock()`
`sem_wait()` (down)	`sem_post()` (up)



Same Order and Reverse Order

Each resource is guarded by a binary semaphore: `down` before access, `up` when done. With one process or one resource there is no problem.

Deadlock-free: both take 1, then 2

```
void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
void process_B(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
```

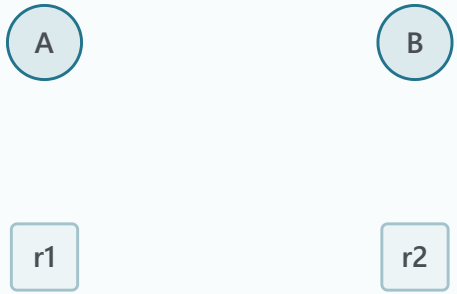
Can deadlock: B takes 2, then 1

```
void process_A(void) {
  down(&resource_1);
  down(&resource_2);
  use_both_resources();
  up(&resource_2);
  up(&resource_1);
}
void process_B(void) {
  down(&resource_2);  /* reverse order */
  down(&resource_1);
  use_both_resources();
  up(&resource_1);
  up(&resource_2);
}
```

If two processes **acquire resources in the same order**, the code cannot deadlock, even if they release in different orders. Deadlock becomes possible when the **acquisition (down) order** is reversed (Tanenbaum Figure 6-2b).

Acquiring in Reverse Order, Step by Step

```
▶ semaphore r1 = 1, r2 = 1;
2 void A(void) {
3   down(&r1);
4   down(&r2);
5   use_both();
6   up(&r2); up(&r1);
7 }
8 void B(void) {
9   down(&r2);
10  down(&r1);
11  use_both();
12  up(&r1); up(&r2);
13 }
```



Two binary semaphores start at 1: both resources are free.

Step 1 / 6

VARIABLES

`r1 = 1` `r2 = 1` `A = ready` `B = ready`

Deadlock doesn't occur on every run: if A finishes both `down`s before B starts, nothing goes wrong. That's why it rarely shows up in testing and surfaces in production.

Acquiring in Reverse Order, Step by Step

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
▶    down(&r1);
4    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
9    down(&r2);
10   down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



A calls `down(&r1)` and gets r1 (edge r1 → A).

Step 2 / 6

VARIABLES

`r1 = 0` `r2 = 1` `A = holds r1` `B = ready`

Deadlock doesn't occur on every run: if A finishes both `down`s before B starts, nothing goes wrong. That's why it rarely shows up in testing and surfaces in production.

Acquiring in Reverse Order, Step by Step

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
3    down(&r1);
4    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
▶    down(&r2);
10   down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



The scheduler switches to B; B gets r2 with `down(&r2)`.

Step 3 / 6

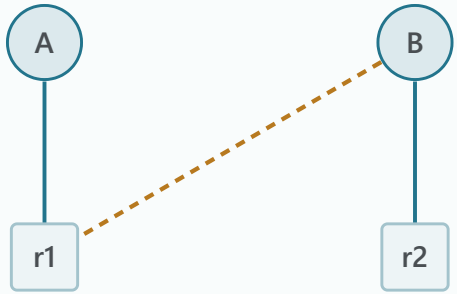
VARIABLES

`r1 = 0` `r2 = 0` `A = holds r1` `B = holds r2`

Deadlock doesn't occur on every run: if A finishes both `down`s before B starts, nothing goes wrong. That's why it rarely shows up in testing and surfaces in production.

Acquiring in Reverse Order, Step by Step

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
3    down(&r1);
4    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
9    down(&r2);
▶    down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



B requests r1; r1 is held by A, so B **blocks** (dashed request edge).

Step 4 / 6

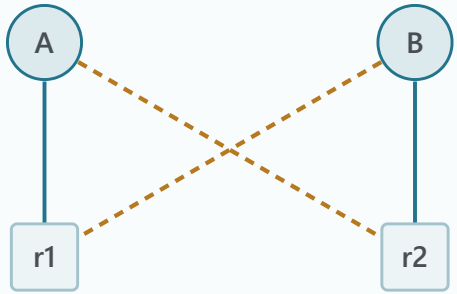
VARIABLES

`r1 = 0` `r2 = 0` `A = holds r1` `B = blocked (r1)`

Deadlock doesn't occur on every run: if A finishes both `down`s before B starts, nothing goes wrong. That's why it rarely shows up in testing and surfaces in production.

Acquiring in Reverse Order, Step by Step

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
3    down(&r1);
▶    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
9    down(&r2);
10   down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



A requests r2; r2 is held by B, so A blocks too.

Step 5 / 6

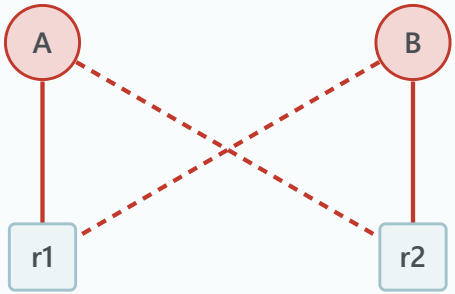
VARIABLES

`r1 = 0` `r2 = 0` `A = blocked (r2)` `B = blocked (r1)`

Deadlock doesn't occur on every run: if A finishes both `down`s before B starts, nothing goes wrong. That's why it rarely shows up in testing and surfaces in production.

Acquiring in Reverse Order, Step by Step

```
1  semaphore r1 = 1, r2 = 1;
2  void A(void) {
3    down(&r1);
▶    down(&r2);
5    use_both();
6    up(&r2); up(&r1);
7  }
8  void B(void) {
9    down(&r2);
▶    down(&r1);
11   use_both();
12   up(&r1); up(&r2);
13 }
```



cycle: A → r2 → B → r1 → A

The graph now has a cycle. Neither can reach its `up` lines: **deadlock**.

Step 6 / 6

VARIABLES

`r1 = 0` `r2 = 0` `A = blocked (r2)` `B = blocked (r1)`

Deadlock doesn't occur on every run: if A finishes both `down`s before B starts, nothing goes wrong. That's why it rarely shows up in testing and surfaces in production.

The Four Conditions for Deadlock (Coffman, 1971)

1. Mutual exclusion

Each resource is either assigned to exactly one process or free; it cannot be shared at the same time.

2. Hold and wait

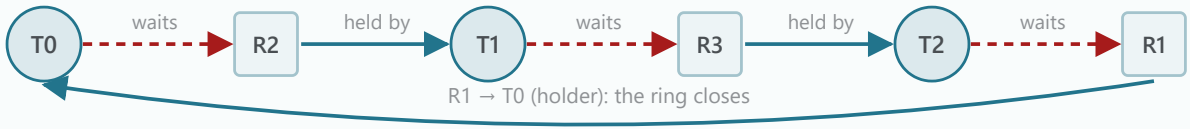
A process *holding* at least one resource requests new resources held by others, and waits.

3. No preemption

A granted resource cannot be taken away by force; only the holding process can release it voluntarily.

4. Circular wait

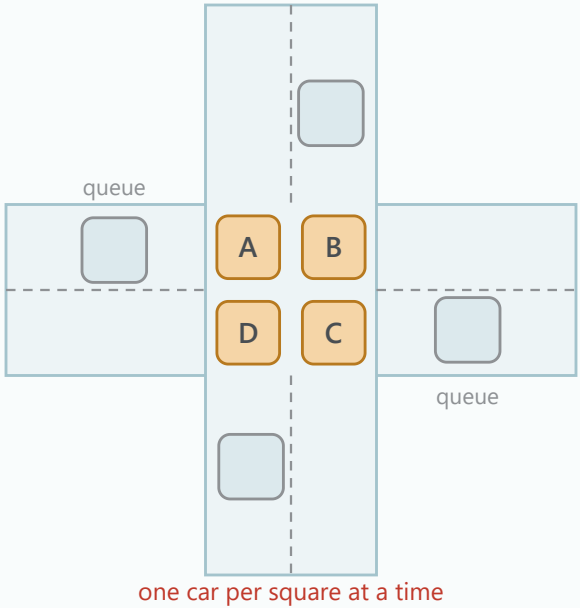
T_0 waits for a resource held by T_1, T_1 for one held by T_2, …, and T_n for one held by T_0.



All four conditions are **necessary together**: if even one is absent, deadlock cannot occur. This is the basis of the prevention strategy. The first three are system *policies*; the fourth is the current *state*. With a single instance of each type, the four conditions are also **sufficient**.

The Four Conditions Together

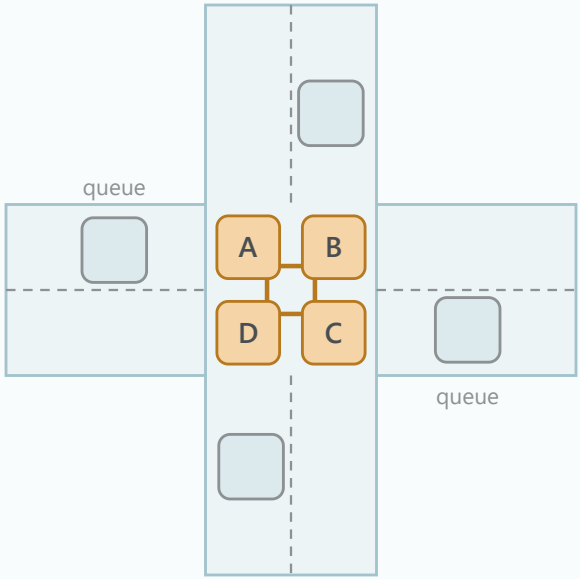
- The intersection's four squares = four **nonpreemptable** resources; cars = processes.
- Each step highlights one condition.
- Breaking any condition is a solution:
  - a traffic officer backs a car out (*preemption*),
  - a "don't enter until the junction is clear" rule (no *hold and wait*),
  - a traffic light (ordering, no *cycle*).



Mutual exclusion: only one car can be in an intersection square at a time.

The Four Conditions Together

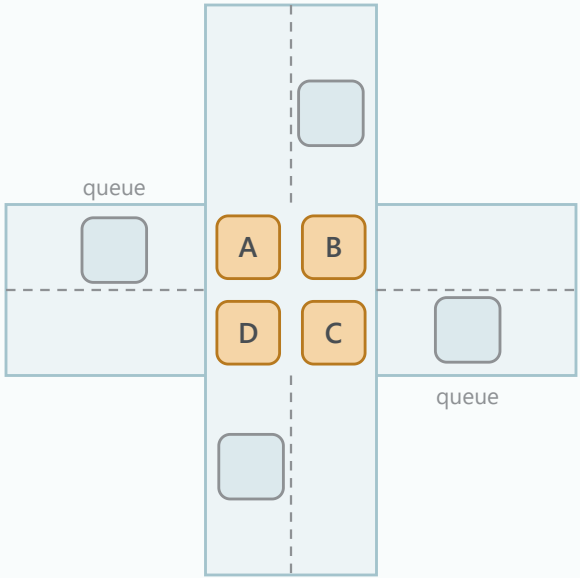
- The intersection's four squares = four **nonpreemptable** resources; cars = processes.
- Each step highlights one condition.
- Breaking any condition is a solution:
  - a traffic officer backs a car out (*preemption*),
  - a "don't enter until the junction is clear" rule (no *hold and wait*),
  - a traffic light (ordering, no *cycle*).



Hold and wait: each car holds its square while waiting for the one ahead (orange arrows).

The Four Conditions Together

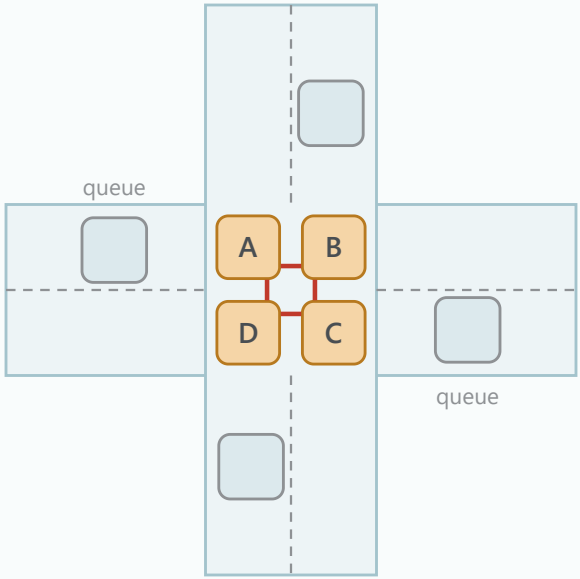
- The intersection's four squares = four **nonpreemptable** resources; cars = processes.
- Each step highlights one condition.
- Breaking any condition is a solution:
  - a traffic officer backs a car out (*preemption*),
  - a "don't enter until the junction is clear" rule (no *hold and wait*),
  - a traffic light (ordering, no *cycle*).



No preemption: there are queues behind; no car can back up to free its square.

The Four Conditions Together

- The intersection's four squares = four **nonpreemptable** resources; cars = processes.
- Each step highlights one condition.
- Breaking any condition is a solution:
  - a traffic officer backs a car out (*preemption*),
  - a "don't enter until the junction is clear" rule (no *hold and wait*),
  - a traffic light (ordering, no *cycle*).

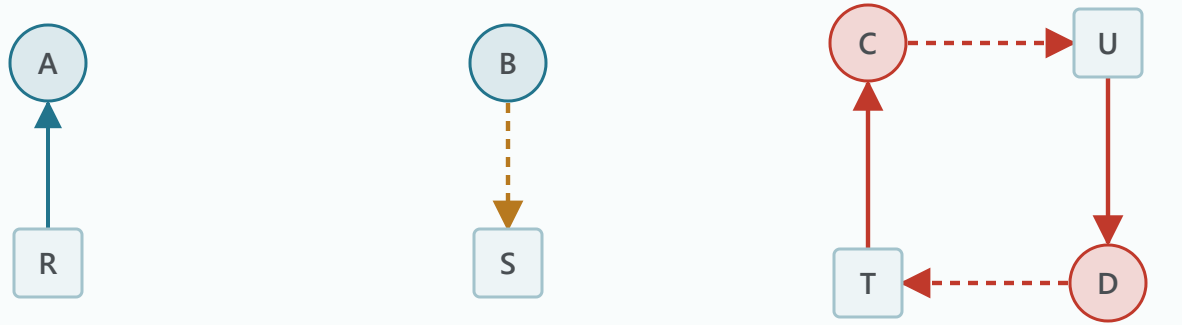


Circular wait: A → B → C → D → A.
All four conditions together: gridlock.

MODELING

Resource-Allocation Graph

- Holt (1972): two kinds of nodes; **processes are circles**, **resources are squares**. A multi-instance resource has one dot inside its square per instance.
- **Assignment edge** R → P: R is allocated to P. **Request edge** P → R: P is waiting for R (dashed orange in these slides).
- When a request is granted, the request edge becomes an assignment edge; on release it is deleted.



(a) A holds R

(b) B requests S

(c) Deadlock: C → U → D → T → C

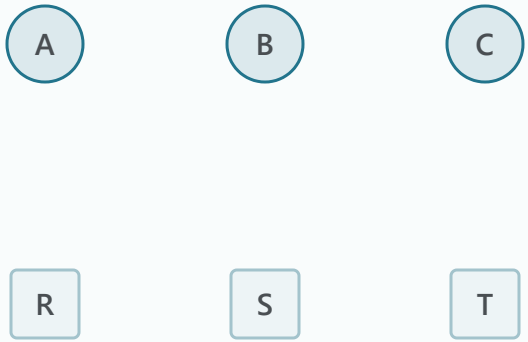
Tanenbaum MOS Figure 6-3. Silberschatz draws the same model as a "resource-allocation graph" with T (thread) nodes.

MODELING

How Deadlock Happens

```
A           B           C
request R   request S   request T
request S   request T   request R
release R   release S   release T
release S   release T   release R
```

If the OS grants requests **in arrival order**: R, S, and T are granted, then each process requests the next one's resource.



Initially all three resources are free. Round-robin scheduling is assumed.

Step 1 / 7

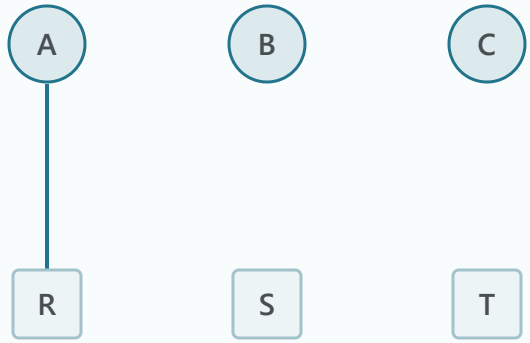
Tanenbaum MOS Figure 6-4 (a)–(j). The scheduler could have picked another order; deadlock depends on the interleaving.

MODELING

How Deadlock Happens

```
A          B          C
request R  request S  request T
request S  request T  request R
release R  release S  release T
release S  release T  release R
```

If the OS grants requests **in arrival order**: R, S, and T are granted, then each process requests the next one's resource.



1. A requests R and gets it.

Step 2 / 7

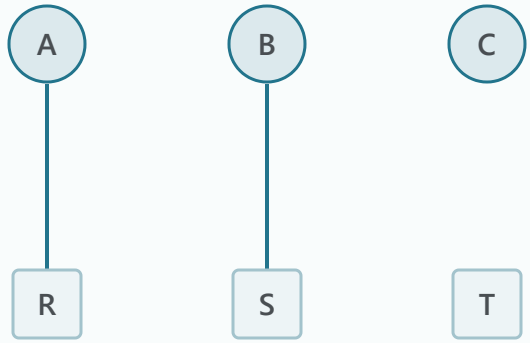
Tanenbaum MOS Figure 6-4 (a)–(j). The scheduler could have picked another order; deadlock depends on the interleaving.

MODELING

How Deadlock Happens

```
A          B          C
request R  request S  request T
request S  request T  request R
release R  release S  release T
release S  release T  release R
```

If the OS grants requests **in arrival order**: R, S, and T are granted, then each process requests the next one's resource.



2. B requests S and gets it.

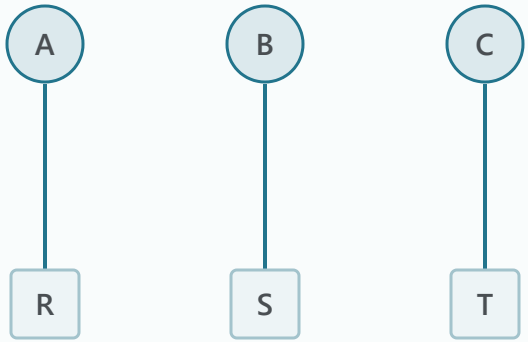
Tanenbaum MOS Figure 6-4 (a)–(j). The scheduler could have picked another order; deadlock depends on the interleaving.

MODELING

How Deadlock Happens

```
A           B           C
request R   request S   request T
request S   request T   request R
release R   release S   release T
release S   release T   release R
```

If the OS grants requests **in arrival order**: R, S, and T are granted, then each process requests the next one's resource.



3. C requests T and gets it.

Step 4 / 7

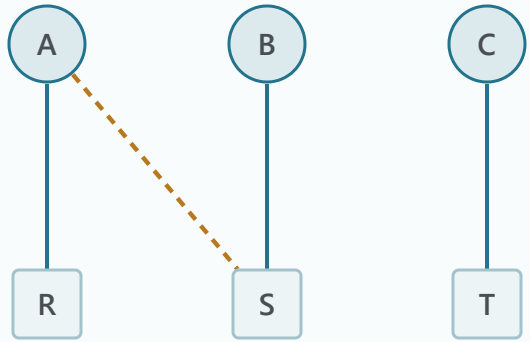
Tanenbaum MOS Figure 6-4 (a)–(j). The scheduler could have picked another order; deadlock depends on the interleaving.

MODELING

How Deadlock Happens

```
A           B           C
request R   request S   request T
request S   request T   request R
release R   release S   release T
release S   release T   release R
```

If the OS grants requests **in arrival order**: R, S, and T are granted, then each process requests the next one's resource.



4. A requests S: S is held by B → A blocks.

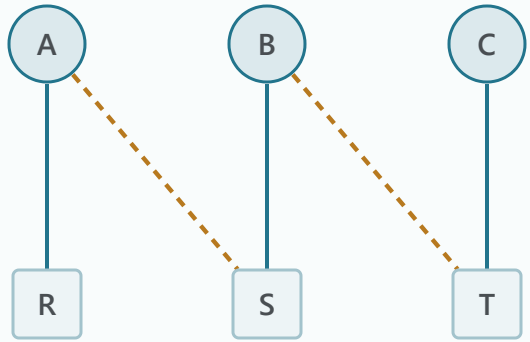
Step 5 / 7

Tanenbaum MOS Figure 6-4 (a)–(j). The scheduler could have picked another order; deadlock depends on the interleaving.

How Deadlock Happens

```
A           B           C
request R   request S   request T
request S   request T   request R
release R   release S   release T
release S   release T   release R
```

If the OS grants requests **in arrival order**: R, S, and T are granted, then each process requests the next one's resource.



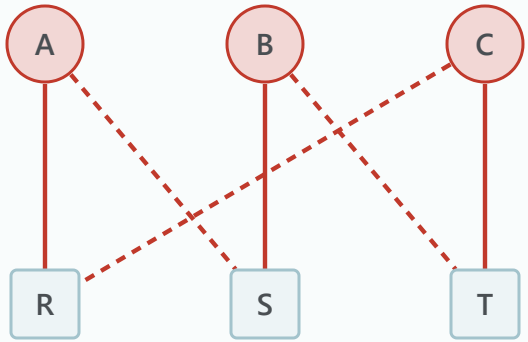
5. B requests T: T is held by C → B blocks.

Tanenbaum MOS Figure 6-4 (a)–(j). The scheduler could have picked another order; deadlock depends on the interleaving.

How Deadlock Happens

A	B	C
request R	request S	request T
request S	request T	request R
release R	release S	release T
release S	release T	release R

If the OS grants requests **in arrival order**: R, S, and T are granted, then each process requests the next one's resource.



cycle A→S→B→T→C→R→A

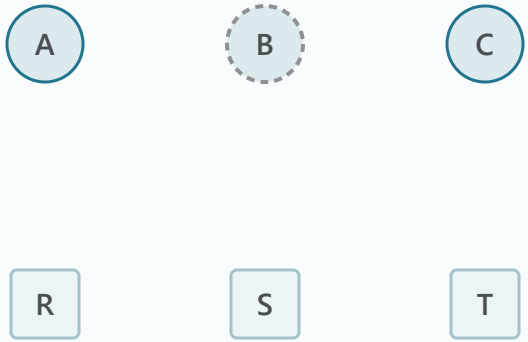
6. C requests R: R is held by A → C blocks. The cycle closes: **deadlock**.

Tanenbaum MOS Figure 6-4 (a)–(j). The scheduler could have picked another order; deadlock depends on the interleaving.

How Could Deadlock Be Avoided?

A	B	C
request R	request S	request T
request S	request T	request R
release R	release S	release T
release S	release T	release R

If the OS **knew in advance** that granting a request would lead to deadlock, it could hold B back and avoid the dangerous order.



The OS **suspends** B's requests for a while (doesn't schedule B). Only A and C run.

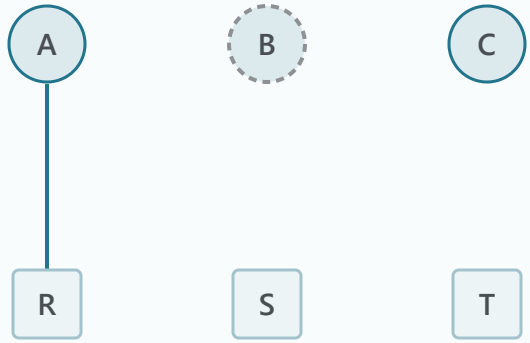
Step 1 / 7

Tanenbaum MOS Figure 6-4 (k)–(q). "Which request should be deferred?" is the heart of **avoidance** (the banker's algorithm), coming up later.

How Could Deadlock Be Avoided?

A	B	C
request R	request S	request T
request S	request T	request R
release R	release S	release T
release S	release T	release R

If the OS **knew in advance** that granting a request would lead to deadlock, it could hold B back and avoid the dangerous order.



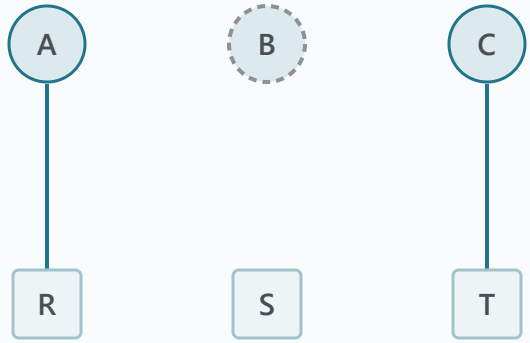
1. A requests R and gets it.

Tanenbaum MOS Figure 6-4 (k)–(q). "Which request should be deferred?" is the heart of **avoidance** (the banker's algorithm), coming up later.

How Could Deadlock Be Avoided?

A	B	C
request R	request S	request T
request S	request T	request R
release R	release S	release T
release S	release T	release R

If the OS **knew in advance** that granting a request would lead to deadlock, it could hold B back and avoid the dangerous order.



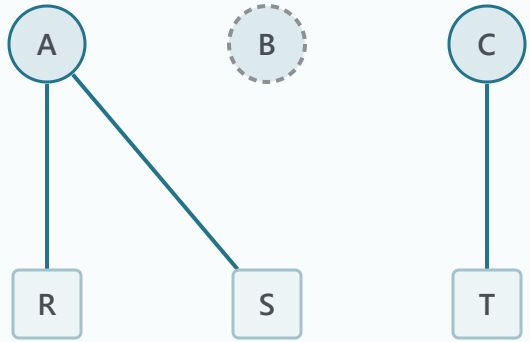
2. C requests T and gets it.

Tanenbaum MOS Figure 6-4 (k)–(q). "Which request should be deferred?" is the heart of **avoidance** (the banker's algorithm), coming up later.

How Could Deadlock Be Avoided?

A	B	C
request R	request S	request T
request S	request T	request R
release R	release S	release T
release S	release T	release R

If the OS **knew in advance** that granting a request would lead to deadlock, it could hold B back and avoid the dangerous order.



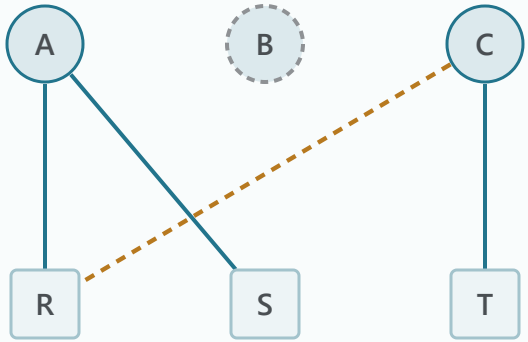
3. A requests S and gets it (S is free).

Tanenbaum MOS Figure 6-4 (k)–(q). "Which request should be deferred?" is the heart of **avoidance** (the banker's algorithm), coming up later.

How Could Deadlock Be Avoided?

A	B	C
request R	request S	request T
request S	request T	request R
release R	release S	release T
release S	release T	release R

If the OS **knew in advance** that granting a request would lead to deadlock, it could hold B back and avoid the dangerous order.



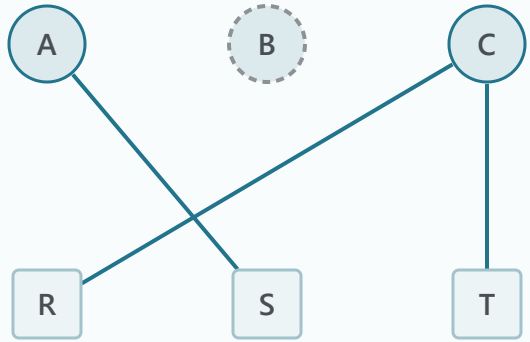
4. C requests R: R is held by A → C blocks. No cycle; A can proceed.

Tanenbaum MOS Figure 6-4 (k)–(q). "Which request should be deferred?" is the heart of **avoidance** (the banker's algorithm), coming up later.

How Could Deadlock Be Avoided?

A	B	C
request R	request S	request T
request S	request T	request R
release R	release S	release T
release S	release T	release R

If the OS **knew in advance** that granting a request would lead to deadlock, it could hold B back and avoid the dangerous order.



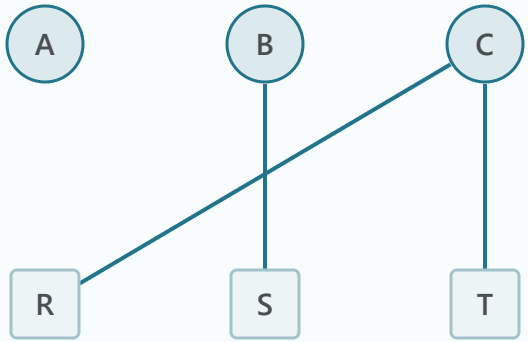
5. A releases R → R goes to C; C proceeds.

Tanenbaum MOS Figure 6-4 (k)–(q). "Which request should be deferred?" is the heart of **avoidance** (the banker's algorithm), coming up later.

How Could Deadlock Be Avoided?

A	B	C
request R	request S	request T
request S	request T	request R
release R	release S	release T
release S	release T	release R

If the OS **knew in advance** that granting a request would lead to deadlock, it could hold B back and avoid the dangerous order.

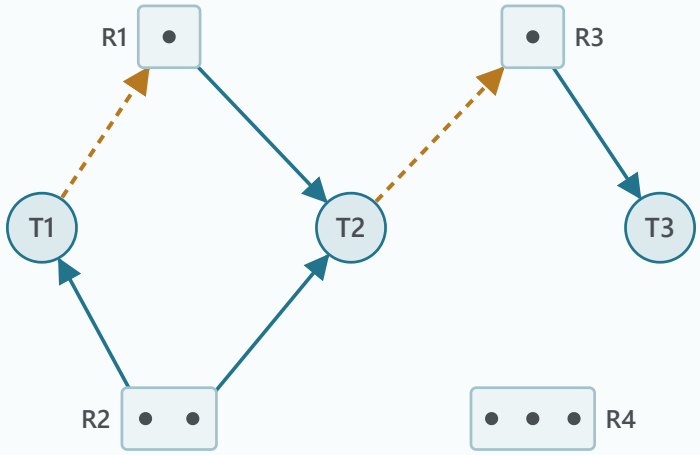


6. A releases S and finishes. Now B can run: it gets S, and T once C finishes. No deadlock.

Tanenbaum MOS Figure 6-4 (k)–(q). "Which request should be deferred?" is the heart of **avoidance** (the banker's algorithm), coming up later.

Multiple-Instance Resources

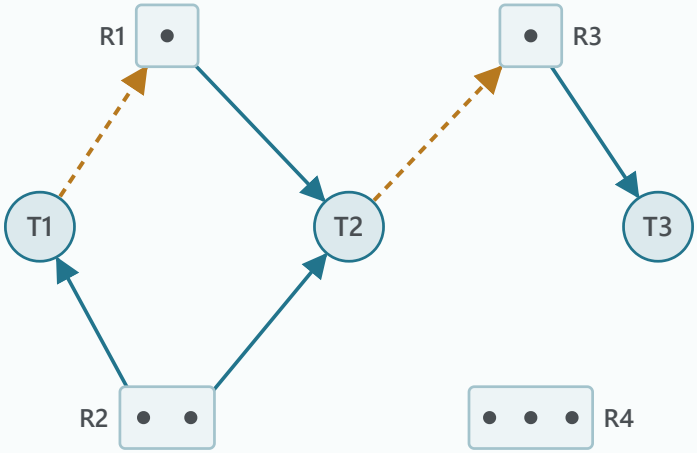
- Resource types: R1 ×1, **R2 ×2**, R3 ×1, **R4 ×3**.
- **T1:** holds one instance of R2, waits for R1.
- **T2:** holds R1 and one instance of R2, waits for R3.
- **T3:** holds R3.
- An assignment edge leaves from a **dot** (an instance) in the square; a request edge goes to the square itself.
- No cycle in the graph → **no deadlock**: when T3 finishes it releases R3, then T2 and T1 proceed.



Silberschatz 10e Figure 8.4. All three instances of R4 are free.

Cycles and Deadlock

- A single request edge (T3 → R2) creates two cycles.
- T3 waits for T1 or T2 to release an R2; T2 waits for T3's R3; T1 waits for T2's R1.
- Every process in the cycle waits for another one in the cycle, and there is no outside "rescuer".



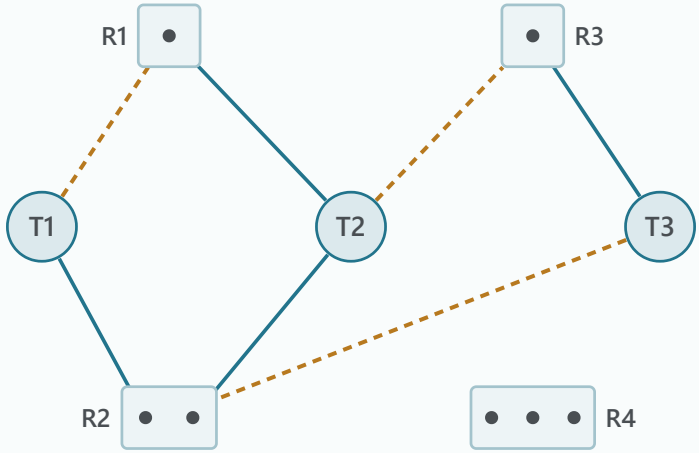
Previous state: no cycle.

Step 1 / 4

Silberschatz 10e Figure 8.5.

Cycles and Deadlock

- A single request edge (T3 → R2) creates two cycles.
- T3 waits for T1 or T2 to release an R2; T2 waits for T3's R3; T1 waits for T2's R1.
- Every process in the cycle waits for another one in the cycle, and there is no outside "rescuer".

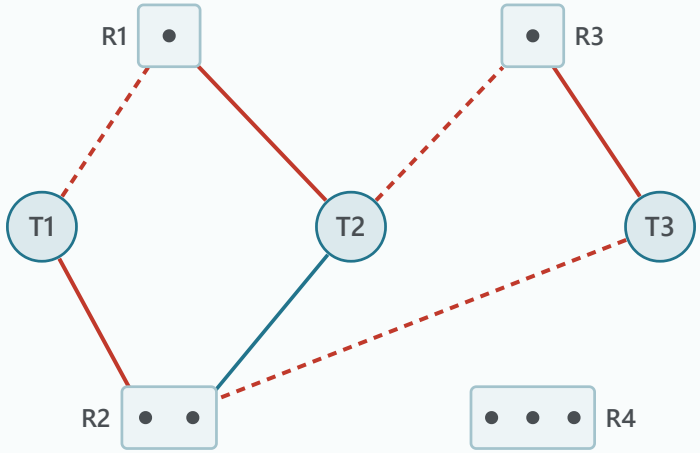


T3 requests an instance of R2. Both instances are taken (by T1 and T2) → T3 blocks.

Silberschatz 10e Figure 8.5.

Cycles and Deadlock

- A single request edge (T3 → R2) creates two cycles.
- T3 waits for T1 or T2 to release an R2; T2 waits for T3's R3; T1 waits for T2's R1.
- Every process in the cycle waits for another one in the cycle, and there is no outside "rescuer".

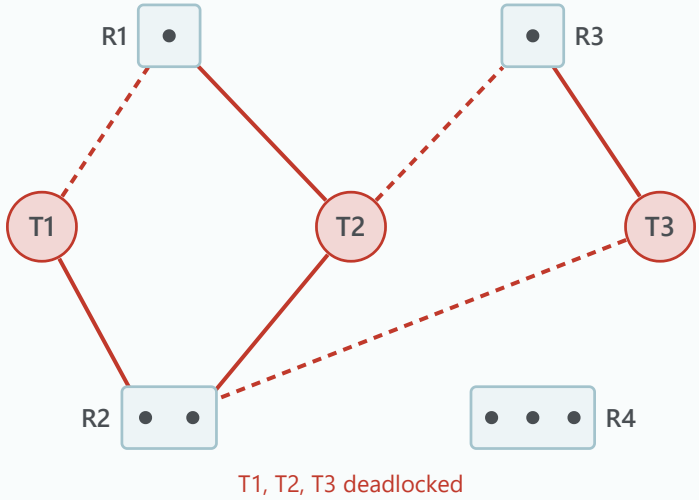


Cycle 1: T1 → R1 → T2 → R3 → T3 → R2 → T1.

Silberschatz 10e Figure 8.5.

Cycles and Deadlock

- A single request edge (T3 → R2) creates two cycles.
- T3 waits for T1 or T2 to release an R2; T2 waits for T3's R3; T1 waits for T2's R1.
- Every process in the cycle waits for another one in the cycle, and there is no outside "rescuer".

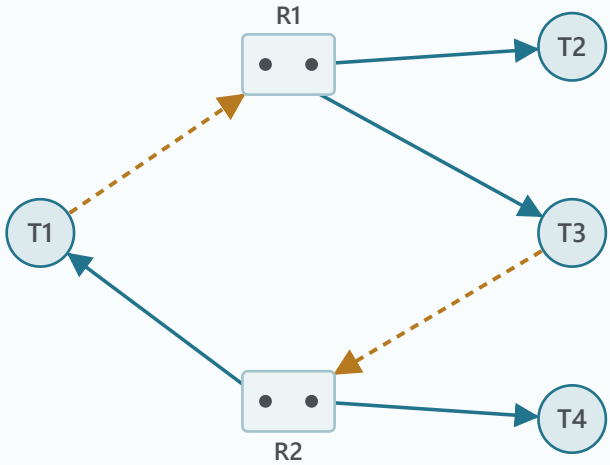


Cycle 2: T2 → R3 → T3 → R2 → T2. The holders of *both* R2 instances are in the cycle → **deadlock**.

Silberschatz 10e Figure 8.5.

A Cycle Without Deadlock

- When resource types have several instances, a cycle is **necessary but not sufficient**.
- When a process outside the cycle (T2 or T4) finishes and releases its instance, the cycle dissolves.
- So with multiple instances, searching for cycles is not enough; a **matrix-based** algorithm is needed.

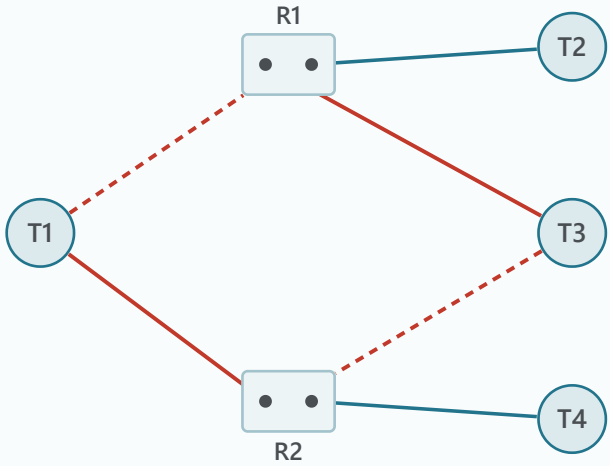


T1 requests an R1; both R1 instances are held by T2 and T3. T3 requests an R2; R2's instances are held by T1 and T4.

Silberschatz 10e Figure 8.6.

A Cycle Without Deadlock

- When resource types have several instances, a cycle is **necessary but not sufficient**.
- When a process outside the cycle (T2 or T4) finishes and releases its instance, the cycle dissolves.
- So with multiple instances, searching for cycles is not enough; a **matrix-based** algorithm is needed.

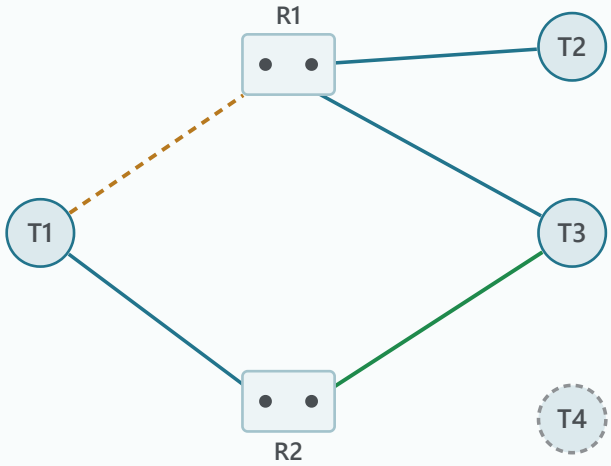


There is a cycle: T1 → R1 → T3 → R2 → T1. But T2 and T4 are *outside* it and wait for nothing.

Silberschatz 10e Figure 8.6.

A Cycle Without Deadlock

- When resource types have several instances, a cycle is **necessary but not sufficient**.
- When a process outside the cycle (T2 or T4) finishes and releases its instance, the cycle dissolves.
- So with multiple instances, searching for cycles is not enough; a **matrix-based** algorithm is needed.



T4 finishes and releases its R2 instance → it goes to T3. The cycle is broken: **there was no deadlock**.

Silberschatz 10e Figure 8.6.

MODELING

The Cycle Rule

Graph	Single instance of each resource type	Multiple instances of some types
No cycle	No deadlock	No deadlock
Cycle	**Deadlock** (cycle ⇔ deadlock)	Deadlock **possible**; a cycle is necessary but not sufficient

Single instance

Cycle detection suffices: DFS in O(V + E). The processes on the cycle are deadlocked.

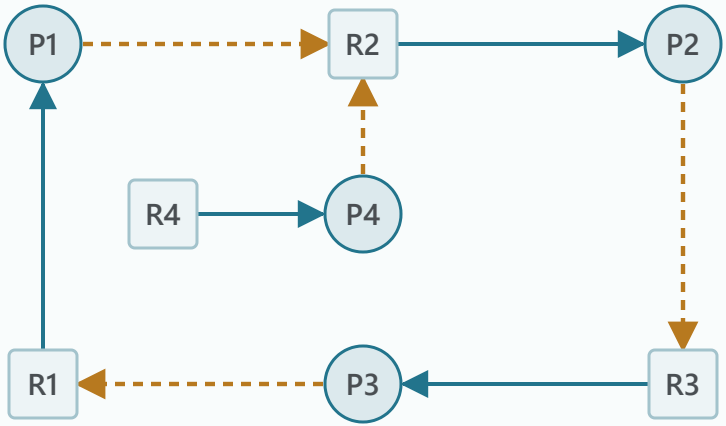
Multiple instances

A cycle is only a warning. For a definite answer, ask "can everyone finish?" using the allocation/request matrices.

The circular-wait condition corresponds to a graph with a cycle; with single-instance resources, the four conditions together guarantee deadlock.

MODELING

Wait-For Graph



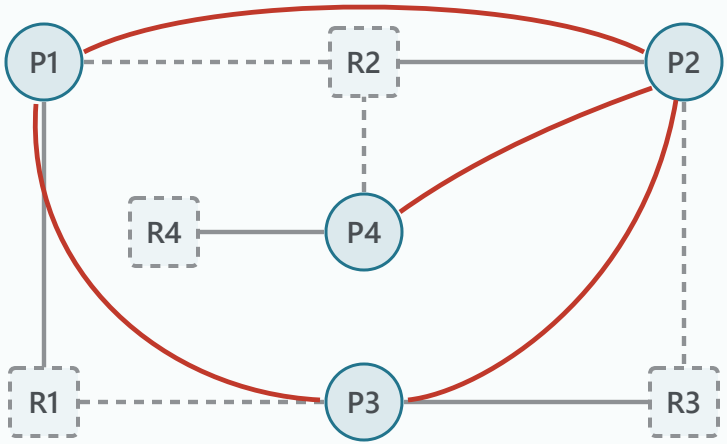
An allocation graph with single-instance resources: P1 holds R1 and requests R2, P2 holds R2 and requests R3, P3 holds R3 and requests R1, P4 holds R4 and requests R2.

Step 1 / 4

Usable only when there is **a single instance of each type**. Databases (e.g., PostgreSQL, MySQL InnoDB) detect deadlock by searching for cycles in the lock wait-for graph.

MODELING

Wait-For Graph

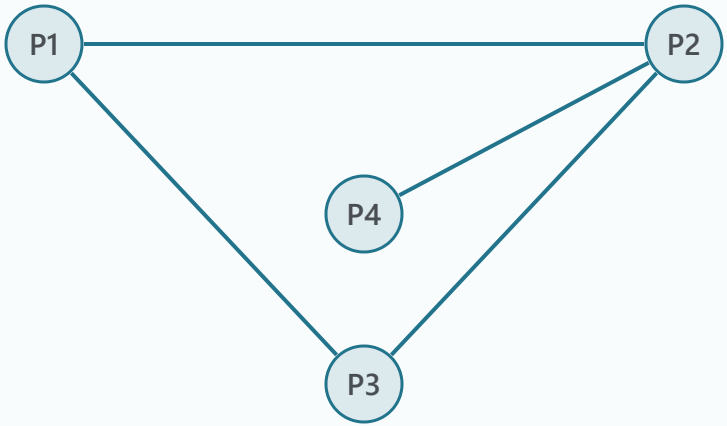


Each path $P_i \rightarrow R \rightarrow P_j$ collapses into one edge $P_i \rightarrow P_j$ (red): "P_i is waiting for P_j".

Usable only when there is **a single instance of each type**. Databases (e.g., PostgreSQL, MySQL InnoDB) detect deadlock by searching for cycles in the lock wait-for graph.

MODELING

Wait-For Graph



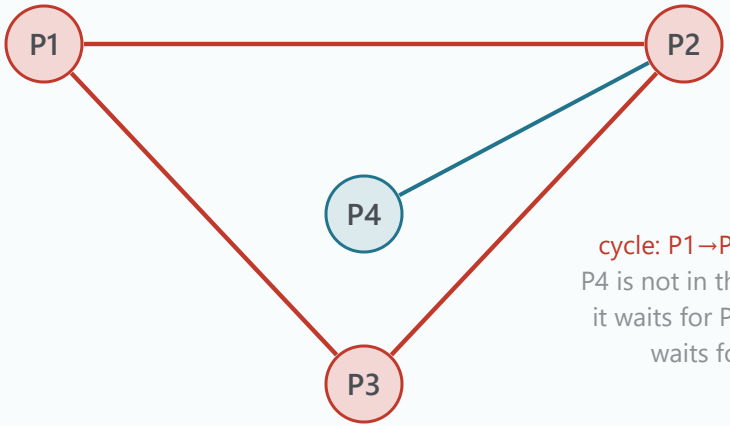
The resource nodes are removed: the **wait-for graph**.

Step 3 / 4

Usable only when there is **a single instance of each type**. Databases (e.g., PostgreSQL, MySQL InnoDB) detect deadlock by searching for cycles in the lock wait-for graph.

MODELING

Wait-For Graph



cycle: P1→P2→P3→P1
P4 is not in the cycle, but it waits for P2, so it too waits forever

A cycle in the wait-for graph ⇔ deadlock. The algorithm scans the graph periodically; the cost is $O(n^2)$ (n: number of processes).

Step 4 / 4

Usable only when there is **a single instance of each type**. Databases (e.g., PostgreSQL, MySQL InnoDB) detect deadlock by searching for cycles in the lock wait-for graph.

STRATEGIES

Strategies for Handling Deadlocks

1. Ignore it

"The ostrich algorithm." If deadlock is rare and prevention is expensive, pretend the problem doesn't exist and reboot when needed.

2. Detect and recover

Reactive. Deadlocks are allowed, detected periodically, then resolved by preemption, rollback, or termination.

3. Dynamic avoidance

Proactive, at run time. A request that would put the system in an unsafe state is deferred (banker's). Requires knowing maximum needs in advance.

4. Prevention

Proactive, at design time. At least one of the four conditions is made structurally impossible (e.g., lock ordering).

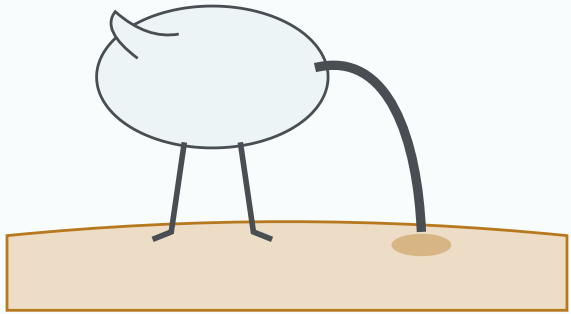
"Recovery" is not a separate technique; it is the second half of detection.

STRATEGIES

The Ostrich Algorithm

- Strategy: **do nothing**. If a deadlock occurs, the user or administrator kills the process or reboots the system.
- An engineering trade-off: if deadlocks happen once a year while crashes and hardware faults are far more frequent, expensive and restrictive measures are not worth it.
- General-purpose OSes (Linux, Windows, macOS) take this approach for user-process locks, file locks, and pipe waits; the responsibility lies with the application developer.
- But the OS protects itself internally: kernel lock ordering and `lockdep` checking.
- Unacceptable in real-time and safety-critical systems.

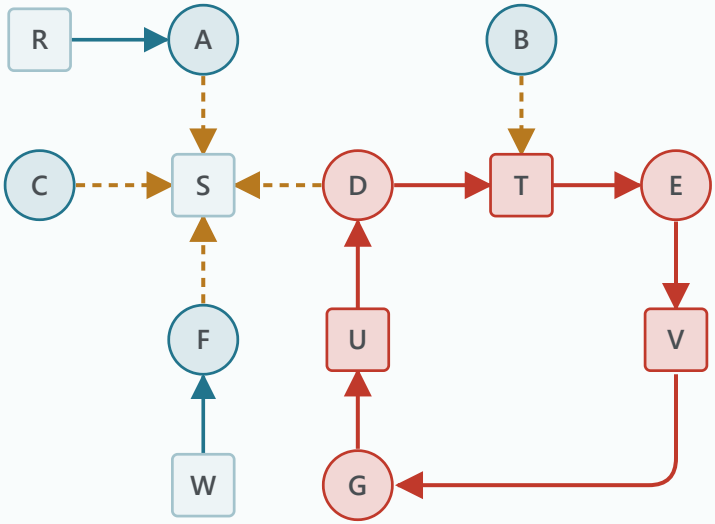
"Deadlock? What deadlock?"



The ostrich approach is not a compromise between detection and avoidance; it does neither, and is purely a cost–probability comparison.

Detection: One Resource of Each Type

- A: holds R, requests S. B: requests T.
- C: requests S. D: holds U, requests S and T.
- E: holds T, requests V. F: holds W, requests S.
- G: holds V, requests U.
- **Question:** is the system deadlocked, and if so, which processes?
- Cycle in the graph: **D** → **T** → **E** → **V** → **G** → **U** → **D**. D, E, and G are deadlocked.
- A, C, and F are not deadlocked: S is free; one of them takes it, finishes, and releases it. B cannot proceed, since it waits for T on the cycle (stuck, though not on the cycle).



Tanenbaum MOS Figure 6-5. Easy by eye; a graph with hundreds of nodes needs a systematic algorithm.

Cycle-Detection Algorithm (DFS)

1. For each node N in the graph, perform steps 2–6 with N as the starting node.
2. Set L to the empty list; mark all edges as unmarked.
3. Add the current node to the end of L. If the node appears in L **twice**, the graph has a cycle: report the cycle in L and stop.
4. Are there unmarked outgoing edges from the current node? If so, go to 5; if not, go to 6.
5. Pick an unmarked edge at random, mark it, follow it to the new current node; go to 3.
6. If this is the starting node, no cycle is reachable from N. Otherwise we've hit a dead end: remove the node from L, back up to the previous node, and continue at step 4 (that node is already in L).

In essence

A *depth-first search* from each node; the list L serves as the stack.

Cost

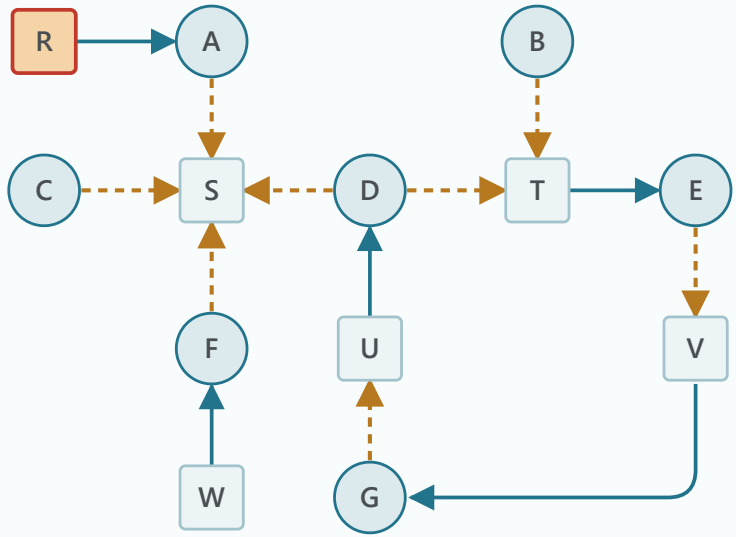
The simple version restarts from every node, so it is O(V·(V+E)); a colored DFS (white/gray/black) needs a single O(V+E) pass.

Output

The part of L after the repeated node is the cycle; the processes in it are deadlocked.

Cycle Detection, Step by Step

- Starting nodes are taken in the order shown (left to right, top to bottom): R, A, B, C, …
- Orange nodes are in the current list L; the red border marks the current node.
- Edge choice may be "random"; here S is tried first, then T.

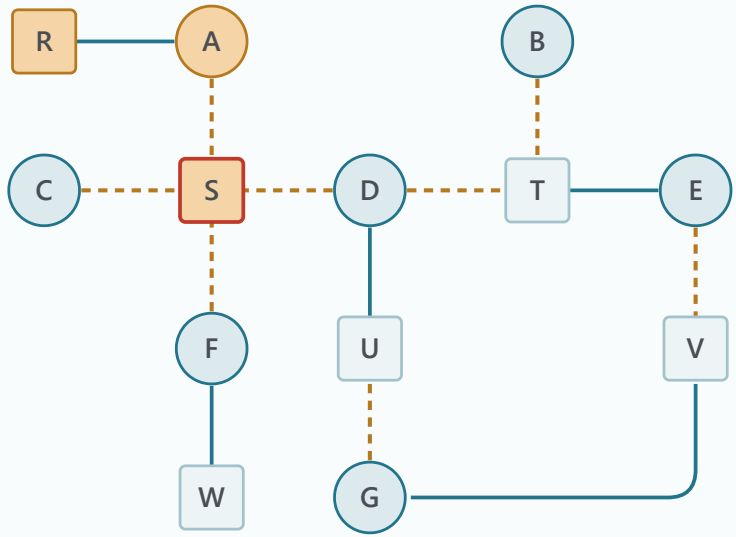


`L = [R]`

Start at R. L = [R]. The only exit is R → A.

Cycle Detection, Step by Step

- Starting nodes are taken in the order shown (left to right, top to bottom): R, A, B, C, …
- Orange nodes are in the current list L; the red border marks the current node.
- Edge choice may be "random"; here S is tried first, then T.

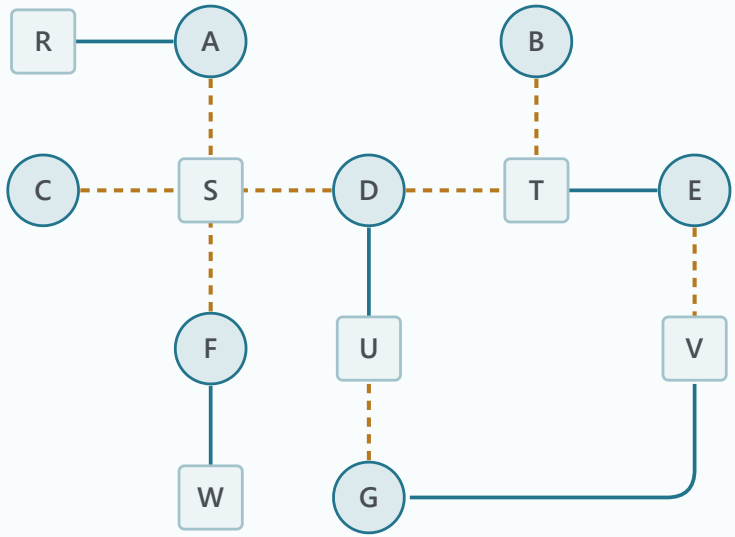


`L = [R, A, S]`

R → A → S. S has no outgoing edge: a dead end.

Cycle Detection, Step by Step

- Starting nodes are taken in the order shown (left to right, top to bottom): R, A, B, C, …
- Orange nodes are in the current list L; the red border marks the current node.
- Edge choice may be "random"; here S is tried first, then T.

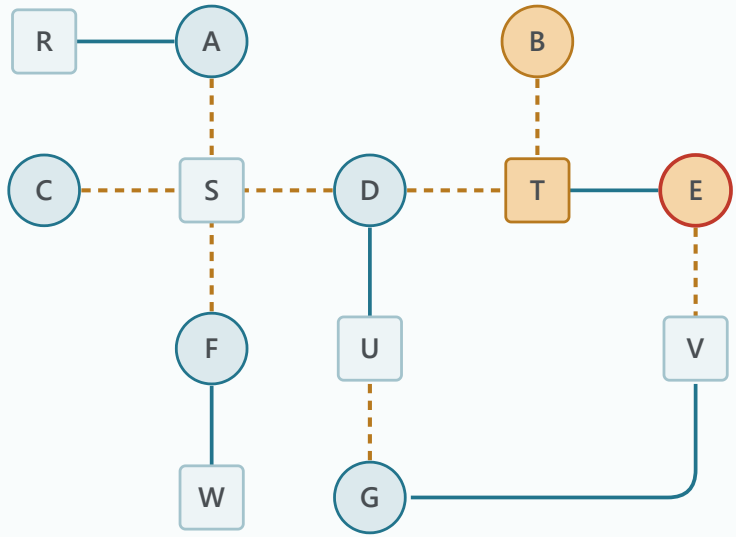


`L = []`

S, then A and R, are removed from L; we're back at R: no cycle from R. Next we start from A (none again)…

Cycle Detection, Step by Step

- Starting nodes are taken in the order shown (left to right, top to bottom): R, A, B, C, …
- Orange nodes are in the current list L; the red border marks the current node.
- Edge choice may be "random"; here S is tried first, then T.

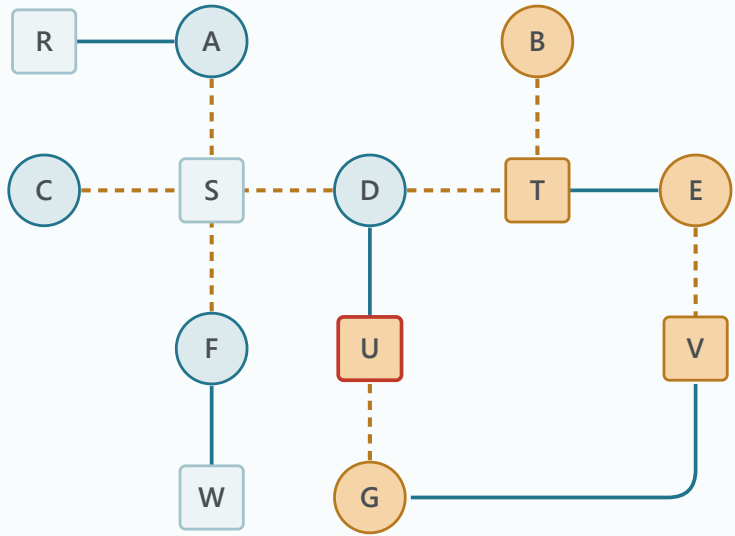


`L = [B, T, E]`

Start from B: B → T → E.

Cycle Detection, Step by Step

- Starting nodes are taken in the order shown (left to right, top to bottom): R, A, B, C, …
- Orange nodes are in the current list L; the red border marks the current node.
- Edge choice may be "random"; here S is tried first, then T.

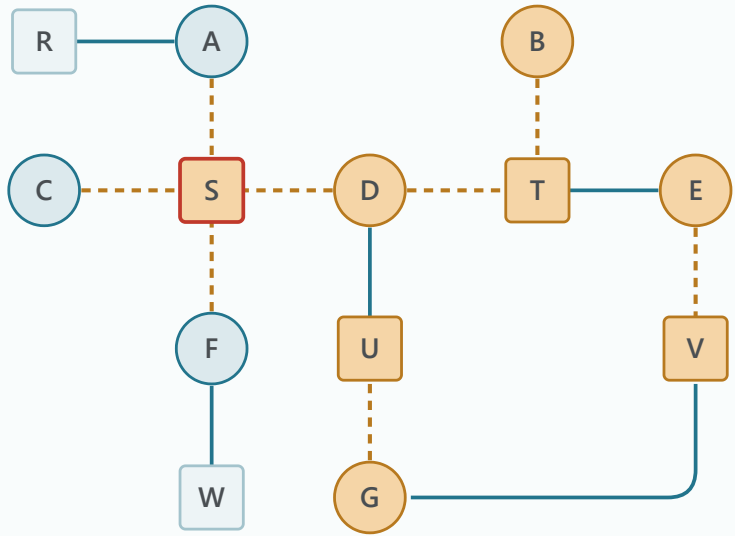


`L = [B, T, E, V, G, U]`

E → V → G → U. Still no repeat.

Cycle Detection, Step by Step

- Starting nodes are taken in the order shown (left to right, top to bottom): R, A, B, C, …
- Orange nodes are in the current list L; the red border marks the current node.
- Edge choice may be "random"; here S is tried first, then T.

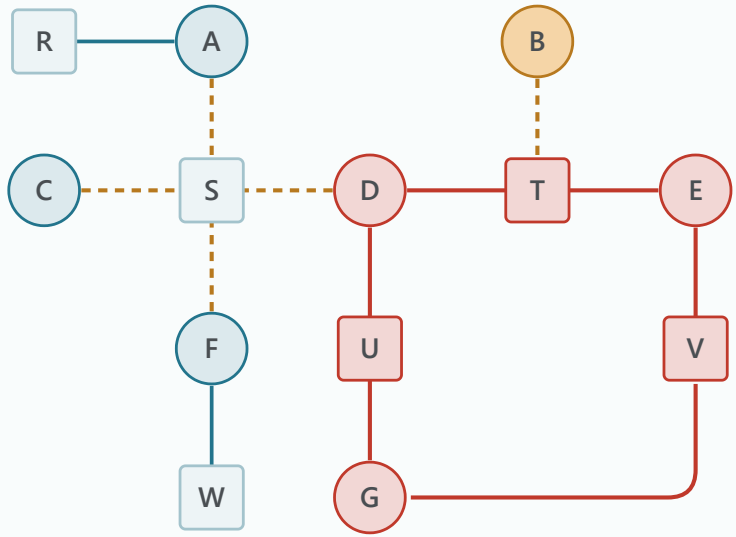


`L = [B, T, E, V, G, U, D, S]`

U → D. From D, S is tried first: a dead end, so S leaves L.

Cycle Detection, Step by Step

- Starting nodes are taken in the order shown (left to right, top to bottom): R, A, B, C, …
- Orange nodes are in the current list L; the red border marks the current node.
- Edge choice may be "random"; here S is tried first, then T.

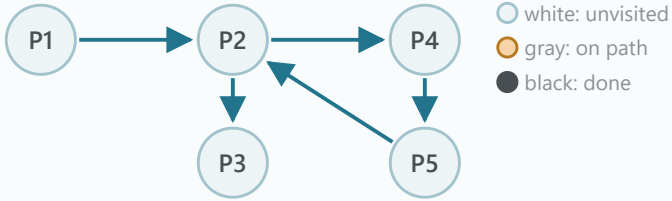


`L = [B, T, E, V, G, U, D, T]`

D → T: T appears in L **a second time** → cycle: T, E, V, G, U, D. The algorithm stops.

Cycle Detection: Colored DFS Code

```
1  enum { WHITE, GRAY, BLACK };
2  int color[N];          /* all WHITE */
3  bool cycle(int u) {
4    color[u] = GRAY;     /* on path */
5    for (int v = 0; v < N; v++) {
6      if (!waits[u][v]) continue;
7      if (color[v] == GRAY)
8        return true;     /* back edge */
9      if (color[v] == WHITE && cycle(v))
10       return true;
11   }
12   color[u] = BLACK;    /* done */
13   return false;
14 }
```



P2 tries P3, then P4

Wait-for graph: P1 → P2, P2 → P3 and P4, P4 → P5, P5 → P2. All **white**; the search starts at P1.

Step 1 / 9

VARIABLES

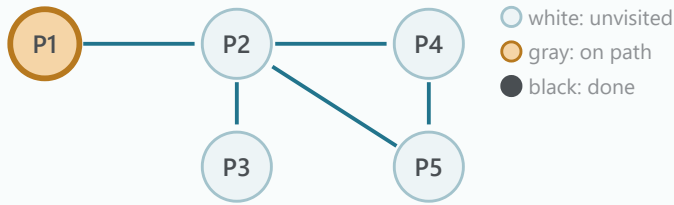
P1 = W P2 = W P3 = W P4 = W P5 = W

CALL STACK

main

Cycle Detection: Colored DFS Code

```
1  enum { WHITE, GRAY, BLACK };
2  int color[N];          /* all WHITE */
3  bool cycle(int u) {
▶    color[u] = GRAY;     /* on path */
5    for (int v = 0; v < N; v++) {
6      if (!waits[u][v]) continue;
7      if (color[v] == GRAY)
8        return true;     /* back edge */
9      if (color[v] == WHITE && cycle(v))
10       return true;
11   }
12   color[u] = BLACK;    /* done */
13   return false;
14 }
```



P2 tries P3, then P4

P1 turns gray: it is on the current path.

Step 2 / 9

VARIABLES

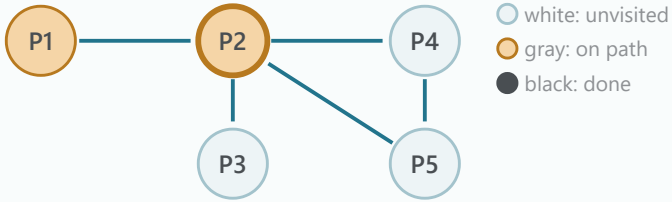
P1 = G P2 = W P3 = W P4 = W P5 = W

CALL STACK

cycle(P1) main

Cycle Detection: Colored DFS Code

```
1  enum { WHITE, GRAY, BLACK };
2  int color[N];          /* all WHITE */
3  bool cycle(int u) {
▶    color[u] = GRAY;     /* on path */
5    for (int v = 0; v < N; v++) {
6      if (!waits[u][v]) continue;
7      if (color[v] == GRAY)
8        return true;     /* back edge */
9      if (color[v] == WHITE && cycle(v))
10       return true;
11   }
12   color[u] = BLACK;    /* done */
13   return false;
14 }
```



P2 tries P3, then P4

P1 waits for P2 and P2 is white → `cycle(P2)` is called; P2 turns gray.

Step 3 / 9

VARIABLES

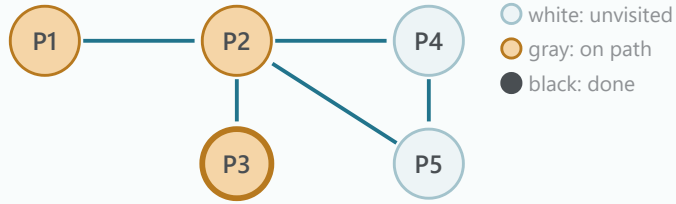
P1 = G | P2 = G | P3 = W | P4 = W | P5 = W

CALL STACK

`cycle(P2)` | `cycle(P1)` | `main`

Cycle Detection: Colored DFS Code

```
1  enum { WHITE, GRAY, BLACK };
2  int color[N];          /* all WHITE */
3  bool cycle(int u) {
▶    color[u] = GRAY;     /* on path */
5    for (int v = 0; v < N; v++) {
6      if (!waits[u][v]) continue;
7      if (color[v] == GRAY)
8        return true;     /* back edge */
9      if (color[v] == WHITE && cycle(v))
10       return true;
11   }
12   color[u] = BLACK;    /* done */
13   return false;
14 }
```



P2 tries P3, then P4

P2 waits for P3 and P3 is white → `cycle(P3)` is called; P3 turns gray.

Step 4 / 9

VARIABLES

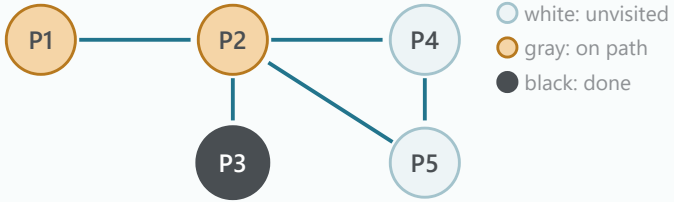
P1 = **G** P2 = **G** P3 = **G** P4 = **W** P5 = **W**

CALL STACK

`cycle(P3)` `cycle(P2)` `cycle(P1)` `main`

Cycle Detection: Colored DFS Code

```
1  enum { WHITE, GRAY, BLACK };
2  int color[N];          /* all WHITE */
3  bool cycle(int u) {
4    color[u] = GRAY;     /* on path */
5    for (int v = 0; v < N; v++) {
6      if (!waits[u][v]) continue;
7      if (color[v] == GRAY)
8        return true;     /* back edge */
9      if (color[v] == WHITE && cycle(v))
10       return true;
11   }
▶    color[u] = BLACK;    /* done */
▶    return false;
14 }
```



P2 tries P3, then P4

P3 waits for no one: the `for` loop ends empty, P3 turns **black** and returns `false`; back to P2.

Step 5 / 9

VARIABLES

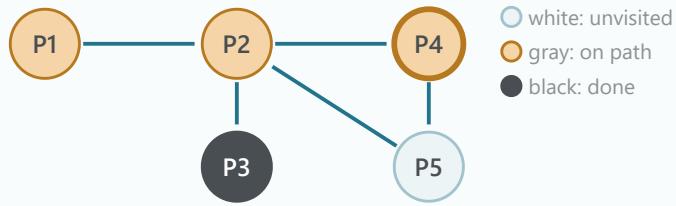
`P1 = G` `P2 = G` `P3 = B` `P4 = W` `P5 = W`

CALL STACK

`cycle(P3)` `cycle(P2)` `cycle(P1)` `main`

Cycle Detection: Colored DFS Code

```
1  enum { WHITE, GRAY, BLACK };
2  int color[N];          /* all WHITE */
3  bool cycle(int u) {
▶    color[u] = GRAY;     /* on path */
5    for (int v = 0; v < N; v++) {
6      if (!waits[u][v]) continue;
7      if (color[v] == GRAY)
8        return true;     /* back edge */
9      if (color[v] == WHITE && cycle(v))
10       return true;
11   }
12   color[u] = BLACK;    /* done */
13   return false;
14 }
```



P2 tries P3, then P4

P2's next neighbor, P4, is white → `cycle(P4)`; P4 turns gray.

Step 6 / 9

VARIABLES

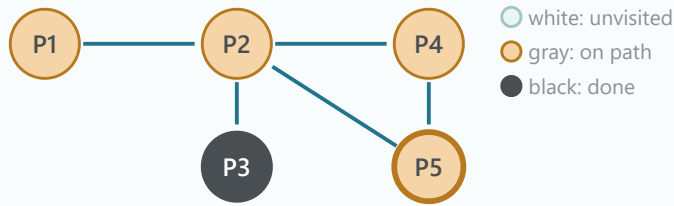
P1 = **G** P2 = **G** P3 = **B** P4 = **G** P5 = **W**

CALL STACK

`cycle(P4)` `cycle(P2)` `cycle(P1)` `main`

Cycle Detection: Colored DFS Code

```
1  enum { WHITE, GRAY, BLACK };
2  int color[N];          /* all WHITE */
3  bool cycle(int u) {
▶    color[u] = GRAY;     /* on path */
5    for (int v = 0; v < N; v++) {
6      if (!waits[u][v]) continue;
7      if (color[v] == GRAY)
8        return true;     /* back edge */
9      if (color[v] == WHITE && cycle(v))
10       return true;
11   }
12   color[u] = BLACK;    /* done */
13   return false;
14 }
```



P2 tries P3, then P4

P4 waits for P5 and P5 is white → `cycle(P5)` is called; P5 turns gray.

Step 7 / 9

VARIABLES

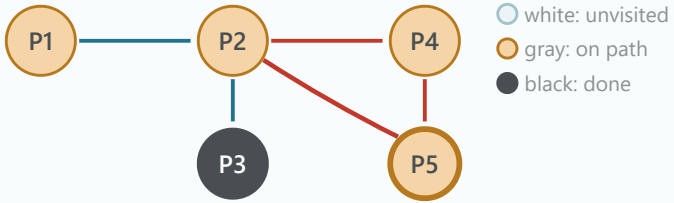
P1 = G P2 = G P3 = B P4 = G P5 = G

CALL STACK

`cycle(P5)` `cycle(P4)` `cycle(P2)` `cycle(P1)` `main`

Cycle Detection: Colored DFS Code

```
1  enum { WHITE, GRAY, BLACK };
2  int color[N];          /* all WHITE */
3  bool cycle(int u) {
4    color[u] = GRAY;     /* on path */
5    for (int v = 0; v < N; v++) {
6      if (!waits[u][v]) continue;
▶      if (color[v] == GRAY)
▶        return true;     /* back edge */
9      if (color[v] == WHITE && cycle(v))
10       return true;
11   }
12   color[u] = BLACK;    /* done */
13   return false;
14 }
```



P2 tries P3, then P4

On edge P5 → P2, P2 is **gray**: still open on the stack. A back edge = a cycle: P2 → P4 → P5 → P2.

Step 8 / 9

VARIABLES

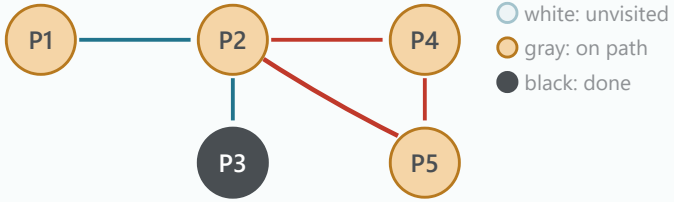
P1 = **G** P2 = **G** P3 = **B** P4 = **G** P5 = **G**

CALL STACK

`cycle(P5)` `cycle(P4)` `cycle(P2)` `cycle(P1)` `main`

Cycle Detection: Colored DFS Code

```
1  enum { WHITE, GRAY, BLACK };
2  int color[N];          /* all WHITE */
3  bool cycle(int u) {
4    color[u] = GRAY;     /* on path */
5    for (int v = 0; v < N; v++) {
6      if (!waits[u][v]) continue;
7      if (color[v] == GRAY)
8        return true;     /* back edge */
▶      if (color[v] == WHITE && cycle(v))
▶        return true;
11   }
12   color[u] = BLACK;    /* done */
13   return false;
14 }
```



P2 tries P3, then P4

`true` propagates up the stack. P2, P4, and P5 on the cycle are deadlocked; P1 is not on the cycle but is stuck because it waits for P2.

Step 9 / 9

VARIABLES

P1 = **G** P2 = **G** P3 = **B** P4 = **G** P5 = **G**

CALL STACK

`main`

Detection: Multiple Resources of Each Type

Existing resources

$E = (E_1, E_2, E_3, \ldots, E_m)$

Available resources

$A = (A_1, A_2, A_3, \ldots, A_m)$

Allocation matrix C (row i: what Pi holds)

$$\begin{matrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & & & \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{matrix}$$

Request matrix R (row i: what Pi requests)

$$\begin{matrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & & & \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{matrix}$$

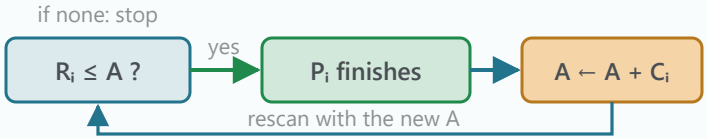
Invariant: $\Sigma_i C_{ij} + A_j = E_j$ (for every resource type j)

- n processes, m resource types. Row i of C is what P_i *currently holds*; row i of R is what P_i *currently requests*.
- Vector comparison: X ≤ Y if and only if $X_j \le Y_j$ for **every** j. E.g., (1 0 1 0) ≤ (2 1 0 0) is **false**.

Tanenbaum MOS Figure 6-6. Silberschatz names the same structures Available, Allocation, and Request.

Detection Algorithm

```
A ← available vector
all Pi unmarked
repeat:
  find an unmarked Pi with Ri ≤ A
  if none: exit
  A ← A + Ci   /* Pi ends, releases */
  mark Pi
unmarked Pi's are deadlocked
```



- **Idea:** a process whose request can be met with the currently available resources *can finish*; when it does, it returns what it holds.
- This is an optimistic assumption: the process is assumed to finish without requesting more. If that's wrong, the next run will catch it.
- Whatever remains unmarked at the end is **deadlocked**.
- Complexity: $O(m \cdot n^2)$.

Row i of C is added to A: $\mathbf{A \leftarrow A + C_i}$ (what the finished process held returns to the free pool).

Detection Example

```
▶ A ← (2 1 0 0)
▶ all Pi unmarked
3 repeat:
4   find unmarked Pi with Ri ≤ A
5   if none: exit
6   A ← A + Ci
7   mark Pi
8 unmarked ones deadlocked
```

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	1	0	0

Tape drives (Tp), plotters (Pl), scanners (Sc), CD. E = (4 2 3 1), available A = (2 1 0 0).

Step 1 / 9

VARIABLES

A = (2 1 0 0) marked = –

Tanenbaum MOS Figure 6-7. The order found (P3, P2, P1) depends on the order in which rows are scanned; the outcome does not.

Detection Example

```
1 A ← (2 1 0 0)
2 all Pi unmarked
3 repeat:
▶   find unmarked Pi with Ri ≤ A
5   if none: exit
6   A ← A + Ci
7   mark Pi
8 unmarked ones deadlocked
```

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	1	0	0

(2 0 0 1) ≤ (2 1 0 0)? no CD ✗

P1: R_1 = (2 0 0 1). No free CD → it can't finish yet.

Step 2 / 9

VARIABLES

A = **(2 1 0 0)** marked = **–**

Tanenbaum MOS Figure 6-7. The order found (P3, P2, P1) depends on the order in which rows are scanned; the outcome does not.

Detection Example

```
1 A ← (2 1 0 0)
2 all Pi unmarked
3 repeat:
▶   find unmarked Pi with Ri ≤ A
5   if none: exit
6   A ← A + Ci
7   mark Pi
8 unmarked ones deadlocked
```

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	1	0	0

(1 0 1 0) ≤ (2 1 0 0)? no scanner ✗

P2: R_2 = (1 0 1 0). No free scanner ✗.

Step 3 / 9

VARIABLES

A = **(2 1 0 0)** marked = **–**

Tanenbaum MOS Figure 6-7. The order found (P3, P2, P1) depends on the order in which rows are scanned; the outcome does not.

Detection Example

```
1 A ← (2 1 0 0)
2 all Pi unmarked
3 repeat:
▶   find unmarked Pi with Ri ≤ A
5   if none: exit
6   A ← A + Ci
7   mark Pi
8 unmarked ones deadlocked
```

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	1	0	0

(2 1 0 0) ≤ (2 1 0 0) ✓

P3: R_3 = (2 1 0 0) ≤ A ✓. P3 can run and finish.

Step 4 / 9

VARIABLES

A = **(2 1 0 0)** marked = **–**

Tanenbaum MOS Figure 6-7. The order found (P3, P2, P1) depends on the order in which rows are scanned; the outcome does not.

Detection Example

```
1 A ← (2 1 0 0)
2 all Pi unmarked
3 repeat:
4   find unmarked Pi with Ri ≤ A
5   if none: exit
▶   A ← A + Ci
▶   mark Pi
8 unmarked ones deadlocked
```

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	2	2	0

P3 finishes and returns its C_3 = (0 1 2 0): A = (2 2 2 0).

Step 5 / 9

VARIABLES

`A = (2 2 2 0)` `marked = P3`

Tanenbaum MOS Figure 6-7. The order found (P3, P2, P1) depends on the order in which rows are scanned; the outcome does not.

Detection Example

```
1 A ← (2 1 0 0)
2 all Pi unmarked
3 repeat:
▶   find unmarked Pi with Ri ≤ A
5   if none: exit
6   A ← A + Ci
7   mark Pi
8 unmarked ones deadlocked
```

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	2	2	2	0

still no CD ✗

New round. P1 still waits (CD = 0).

Step 6 / 9

VARIABLES

`A = (2 2 2 0)` `marked = P3`

Tanenbaum MOS Figure 6-7. The order found (P3, P2, P1) depends on the order in which rows are scanned; the outcome does not.

Detection Example

```
1 A ← (2 1 0 0)
2 all Pi unmarked
3 repeat:
▶   find unmarked Pi with Ri ≤ A
5   if none: exit
▶   A ← A + Ci
▶   mark Pi
8 unmarked ones deadlocked
```

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	4	2	2	1

P2: (1 0 1 0) ≤ (2 2 2 0) ✓ → finishes, returning C_2 = (2 0 0 1): A = (4 2 2 1).

Step 7 / 9

VARIABLES

`A = (4 2 2 1)` `marked = P3, P2`

Tanenbaum MOS Figure 6-7. The order found (P3, P2, P1) depends on the order in which rows are scanned; the outcome does not.

Detection Example

```
1 A ← (2 1 0 0)
2 all Pi unmarked
3 repeat:
▶   find unmarked Pi with Ri ≤ A
5   if none: exit
▶   A ← A + Ci
▶   mark Pi
8 unmarked ones deadlocked
```

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	4	2	3	1

P1: (2 0 0 1) ≤ (4 2 2 1) ✓ → finishes: A = (4 2 3 1) = E.

Step 8 / 9

VARIABLES

A = (4 2 3 1) marked = P3, P2, P1

Tanenbaum MOS Figure 6-7. The order found (P3, P2, P1) depends on the order in which rows are scanned; the outcome does not.

Detection Example

```
1 A ← (2 1 0 0)
2 all Pi unmarked
3 repeat:
4   find unmarked Pi with Ri ≤ A
5   if none: exit
6   A ← A + Ci
7   mark Pi
▶ unmarked ones deadlocked
```

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	0

E = (4 2 3 1)

A	4	2	3	1

order: P3 → P2 → P1

Every process is marked: **no deadlock**.

Step 9 / 9

VARIABLES

`A = (4 2 3 1)` `marked = P3, P2, P1`

Tanenbaum MOS Figure 6-7. The order found (P3, P2, P1) depends on the order in which rows are scanned; the outcome does not.

One Request Too Many: Deadlock

- Changing a single cell (the CD in R_3) changes the outcome completely.
- If no row can be satisfied in the first round, the algorithm stops at once.
- With multi-instance resources this scan replaces the graph; cycle information comes for free.

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	1

E = (4 2 3 1)

A	2	1	0	0

Same state, one difference: in addition to two tape drives and a plotter, P3 also wants **a CD**: R_3 = (2 1 0 1).

One Request Too Many: Deadlock

- Changing a single cell (the CD in R_3) changes the outcome completely.
- If no row can be satisfied in the first round, the algorithm stops at once.
- With multi-instance resources this scan replaces the graph; cycle information comes for free.

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	1

E = (4 2 3 1)

A	2	1	0	0

P1 ✗ (no CD)

P1 ✗ (needs a CD, none free).

One Request Too Many: Deadlock

- Changing a single cell (the CD in R_3) changes the outcome completely.
- If no row can be satisfied in the first round, the algorithm stops at once.
- With multi-instance resources this scan replaces the graph; cycle information comes for free.

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	1

E = (4 2 3 1)

A	2	1	0	0

P2 ✗ (no scanner)

P2 ✗ (needs a scanner, none free).

One Request Too Many: Deadlock

- Changing a single cell (the CD in R_3) changes the outcome completely.
- If no row can be satisfied in the first round, the algorithm stops at once.
- With multi-instance resources this scan replaces the graph; cycle information comes for free.

C (held)

	Tp	Pl	Sc	CD
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	2	0

R (request)

	Tp	Pl	Sc	CD
P1	2	0	0	1
P2	1	0	1	0
P3	2	1	0	1

E = (4 2 3 1)

A	2	1	0	0

P3 ✗ → P1, P2, P3 deadlocked

P3 ✗ (the only CD is held by P2). No row can be satisfied: **all three processes are deadlocked**.

When Should Detection Run?

On every request

After every request that must wait. The deadlock is found the moment it forms, along with the culprit request; but the CPU cost is high.

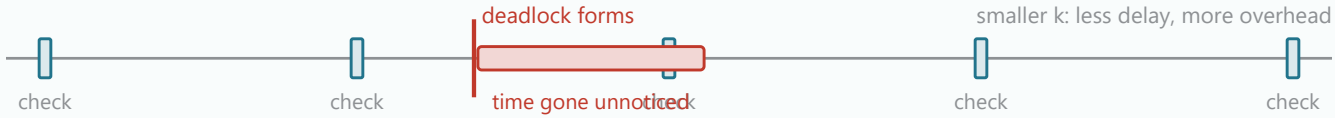
Periodically

Every k minutes. Cheap; a deadlock may go unnoticed for a while, and more processes may get caught in it.

On symptoms

When CPU utilization drops below some threshold, or a lock wait grows long.

- How often? It depends on **how likely deadlock is** and how many processes it would affect.
- Example: PostgreSQL scans the wait-for graph for a transaction that has waited for a lock for `deadlock_timeout` (default 1 s); if it finds a cycle, it aborts a transaction (`ERROR: deadlock detected`).
- The overhead of detection is not negligible and grows with how often it runs; the cost of recovery (lost work) must be counted too.



Recovery from Deadlock

Resource preemption

Temporarily take a resource from its owner and give it to another process (e.g., set a print job aside and separate the pages). Depends on the resource; often manual or impossible.

Rollback

Processes periodically save *checkpoints*. The victim is rolled back to a checkpoint before it acquired the resource; later work is lost. This is a transaction abort in databases.

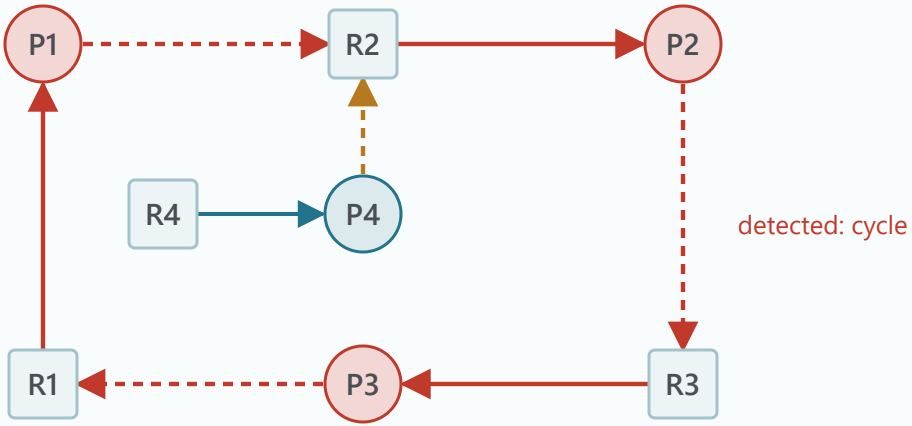
Termination (killing)

The crudest method. Kill either all processes in the cycle, or *one at a time* until the cycle breaks (rerunning detection after each kill). A process outside the cycle may also be chosen.

Victim selection (minimum cost): priority, how long it has run and how much remains, how many resources it holds, how many more it needs to finish, interactive or batch, whether it can be restarted (a compile, yes; a database update, carefully). If the same process is always chosen, it **starves**: count its rollbacks in the cost.

Prevention and detection are not recovery methods. No recovery is free: every method costs lost work or a rerun.

Recovery, Step by Step

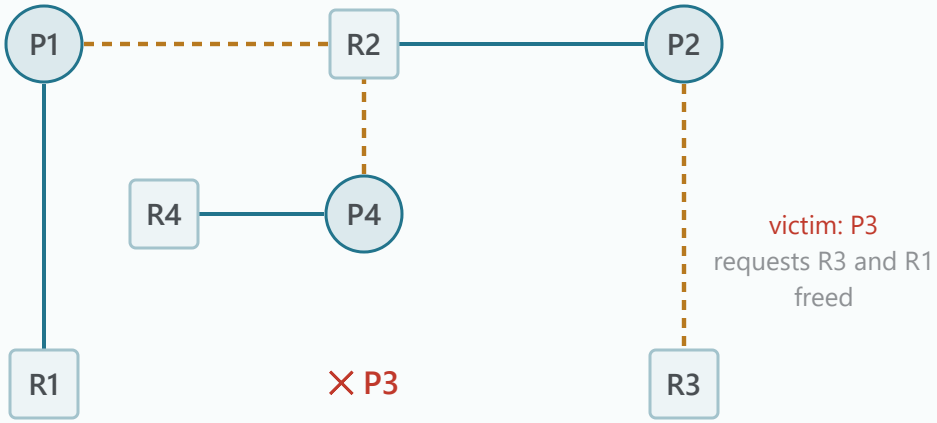


The detection algorithm found the cycle P1 → R2 → P2 → R3 → P3 → R1 → P1. P4 is stuck too, since it waits for R2.

Step 1 / 4

When killing one at a time, detection reruns after each victim; stop once the cycle is broken.

Recovery, Step by Step

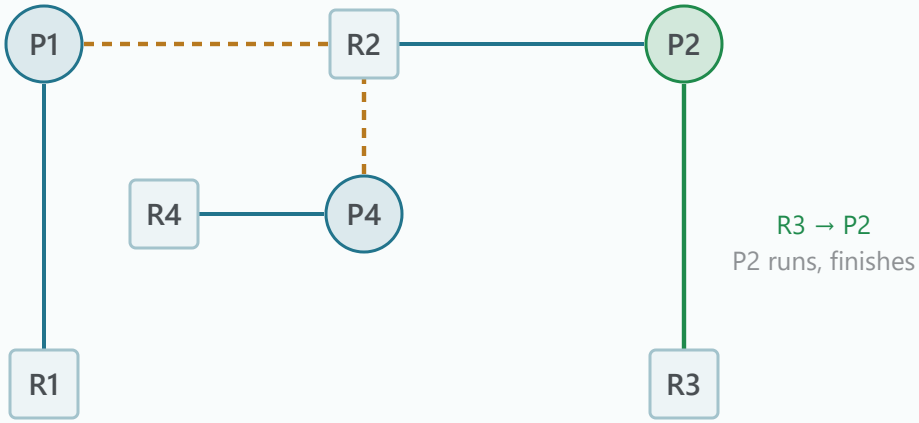


The victim is the process whose loss costs least: P3. It is terminated (or rolled back to a checkpoint); the R3 it held becomes free.

Step 2 / 4

When killing one at a time, detection reruns after each victim; stop once the cycle is broken.

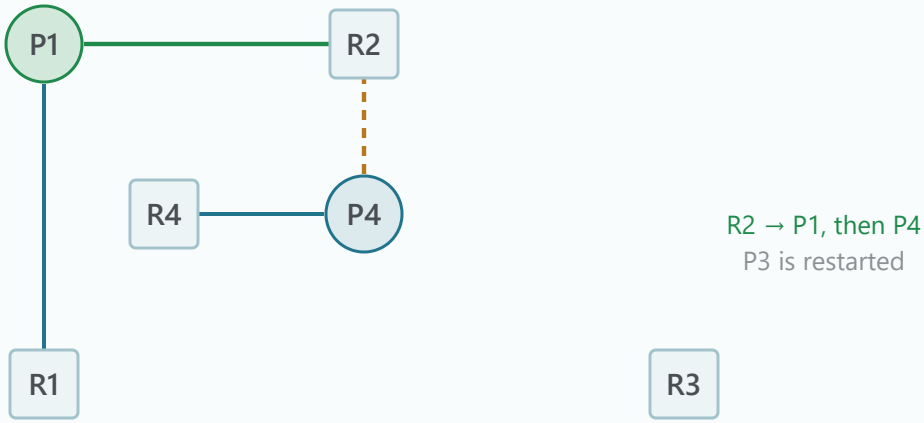
Recovery, Step by Step



R3 goes to the waiting P2. P2 finishes its work and releases R2 and R3.

When killing one at a time, detection reruns after each victim; stop once the cycle is broken.

Recovery, Step by Step

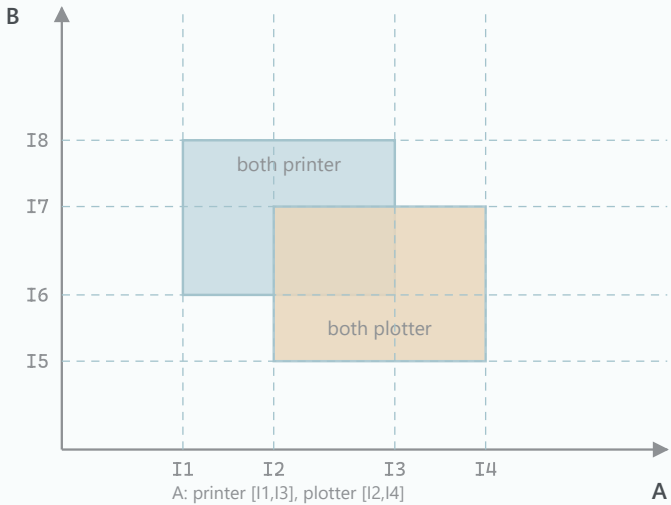


R2 goes to the next waiter (P1); when P1 finishes, to P4. P3 is rerun from the start (or from its checkpoint).

When killing one at a time, detection reruns after each victim; stop once the cycle is broken.

Resource Trajectories

- A uses the printer from I1 to I3 and the plotter from I2 to I4.
- B uses the plotter from I5 to I7 and the printer from I6 to I8.
- The shaded regions **cannot be entered** because of mutual exclusion.
- Avoidance = **deferring** any request that would steer the trajectory into the unsafe region.

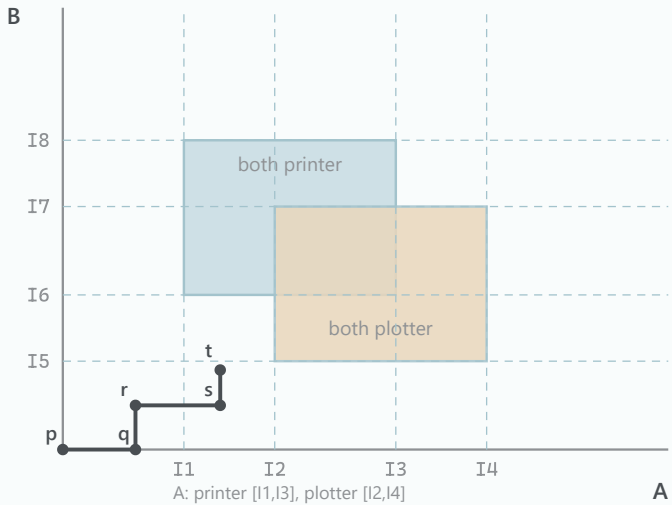


The horizontal axis is A's executed instructions, the vertical axis B's. With one CPU the trajectory moves only **right** (A runs) or **up** (B runs).

Tanenbaum MOS Figure 6-8 (Dijkstra's two-process diagram).

Resource Trajectories

- A uses the printer from I1 to I3 and the plotter from I2 to I4.
- B uses the plotter from I5 to I7 and the printer from I6 to I8.
- The shaded regions **cannot be entered** because of mutual exclusion.
- Avoidance = **deferring** any request that would steer the trajectory into the unsafe region.

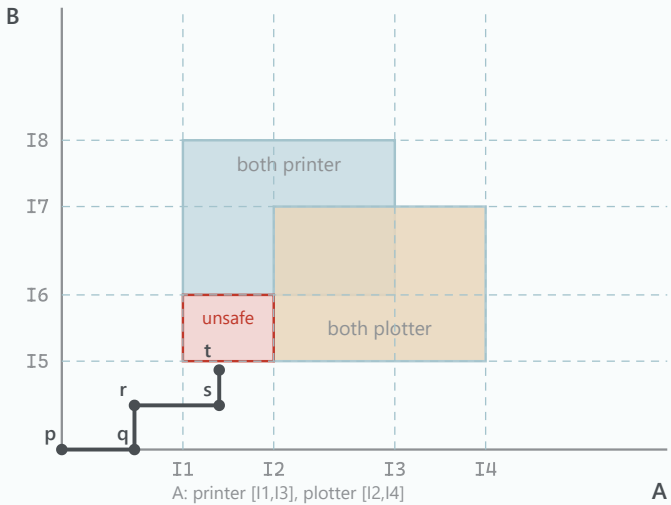


p → q: A runs; q → r: B runs; r → s: A acquires the printer (I1). s → t: B advances and will request the plotter at I5.

Tanenbaum MOS Figure 6-8 (Dijkstra's two-process diagram).

Resource Trajectories

- A uses the printer from I1 to I3 and the plotter from I2 to I4.
- B uses the plotter from I5 to I7 and the printer from I6 to I8.
- The shaded regions **cannot be entered** because of mutual exclusion.
- Avoidance = **deferring** any request that would steer the trajectory into the unsafe region.

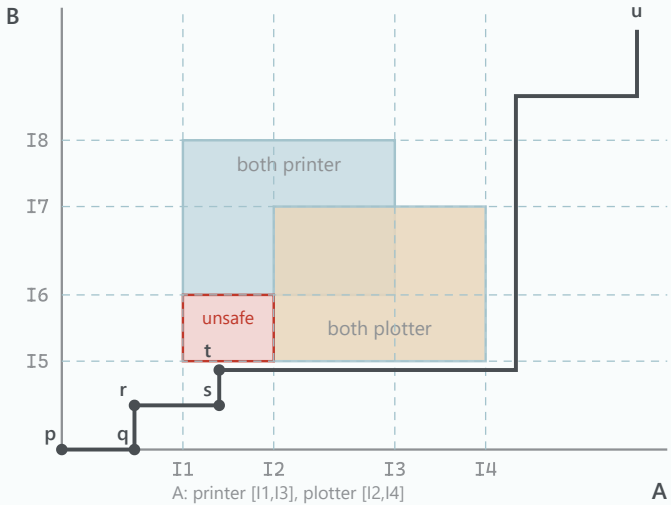


Once inside the red box, A holds the printer and B the plotter, and both must enter a shaded region: deadlock is **inevitable**. This region is **unsafe**.

Tanenbaum MOS Figure 6-8 (Dijkstra's two-process diagram).

Resource Trajectories

- A uses the printer from I1 to I3 and the plotter from I2 to I4.
- B uses the plotter from I5 to I7 and the printer from I6 to I8.
- The shaded regions **cannot be entered** because of mutual exclusion.
- Avoidance = **deferring** any request that would steer the trajectory into the unsafe region.



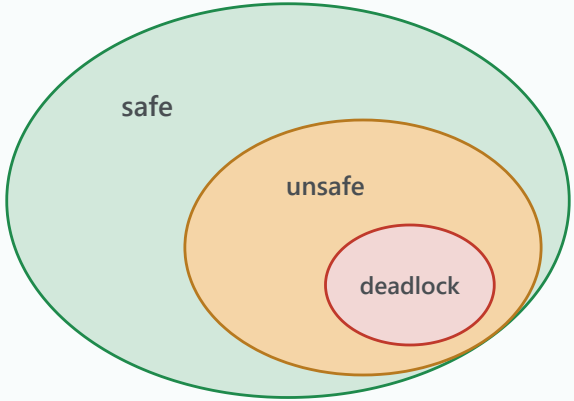
At point t the OS defers B's plotter request; A runs to I4 and releases both. Then B continues: no deadlock.

Step 4 / 4

Tanenbaum MOS Figure 6-8 (Dijkstra's two-process diagram).

Safe and Unsafe States

- **State:** determined by E, A, C, and the maximum needs (Max).
- **Safe state:** there is a **safe sequence** $<P_1, ..., P_n>$ that runs all processes to completion *in some order*, even if every process immediately requests its maximum.
- P_i's remaining need ≤ available + held by all P_j ($j < i$).
- **Unsafe ≠ deadlocked.** An unsafe state is not yet a deadlock; if processes don't make their maximum requests, all may go well. But the system can no longer *guarantee* completion.
- An avoidance algorithm keeps the system in a safe state at all times.



A Safe State Example

- One resource type, 10 instances.
- **Has:** currently held; **Max:** the most it will need before finishing.
- Remaining need = Max − Has.
- At every step the OS is in a position to choose the order of requests.

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a) 10 instances in total; 7 allocated, 3 free. Is it safe? Let's look for a safe sequence.

Tanenbaum MOS Figure 6-9.

A Safe State Example

- One resource type, 10 instances.
- **Has:** currently held; **Max:** the most it will need before finishing.
- Remaining need = Max − Has.
- At every step the OS is in a position to choose the order of requests.

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(b) B's remaining need is 2 ≤ 3: B gets 2 and reaches its maximum.

Tanenbaum MOS Figure 6-9.

A Safe State Example

- One resource type, 10 instances.
- **Has:** currently held; **Max:** the most it will need before finishing.
- Remaining need = Max − Has.
- At every step the OS is in a position to choose the order of requests.

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(c) B finishes and releases 4 instances: 5 free.

Tanenbaum MOS Figure 6-9.

A Safe State Example

- One resource type, 10 instances.
- **Has:** currently held; **Max:** the most it will need before finishing.
- Remaining need = Max − Has.
- At every step the OS is in a position to choose the order of requests.

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(d) C's remaining need is 5 ≤ 5: C gets 5.

Tanenbaum MOS Figure 6-9.

A Safe State Example

- One resource type, 10 instances.
- **Has:** currently held; **Max:** the most it will need before finishing.
- Remaining need = Max − Has.
- At every step the OS is in a position to choose the order of requests.

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

safe sequence: B, C, A

(e) C finishes: 7 free. A's remaining 6 can be met. The safe sequence **B** → **C** → **A** exists: the state is safe.

Tanenbaum MOS Figure 6-9.

An Unsafe State Example

- The difference is a single decision: giving A one extra instance in (a).
- (b) is not yet a deadlock. If A releases a resource before requesting more, there may be no problem.
- But the OS assumes the worst case: if everyone requests its maximum, completion is not guaranteed.

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

The same (safe) start. Now A requests one more instance and the OS *grants* it.

Step 1 / 4

Tanenbaum MOS Figure 6-10.

An Unsafe State Example

- The difference is a single decision: giving A one extra instance in (a).
- (b) is not yet a deadlock. If A releases a resource before requesting more, there may be no problem.
- But the OS assumes the worst case: if everyone requests its maximum, completion is not guaranteed.

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b) A: 4/9, 2 free. Is there a safe sequence?

Tanenbaum MOS Figure 6-10.

An Unsafe State Example

- The difference is a single decision: giving A one extra instance in (a).
- (b) is not yet a deadlock. If A releases a resource before requesting more, there may be no problem.
- But the OS assumes the worst case: if everyone requests its maximum, completion is not guaranteed.

	Has	Max
A	4	9
B	4	4
C	2	7

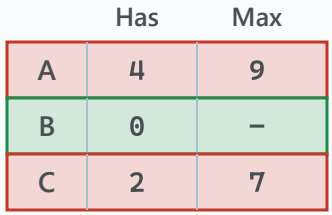
Free: 0

(c) Only B can finish (need 2 ≤ 2): B gets 2.

Tanenbaum MOS Figure 6-10.

An Unsafe State Example

- The difference is a single decision: giving A one extra instance in (a).
- (b) is not yet a deadlock. If A releases a resource before requesting more, there may be no problem.
- But the OS assumes the worst case: if everyone requests its maximum, completion is not guaranteed.



(d) B finishes: 4 free. A needs 5 and C needs 5: neither can be guaranteed. State (b) was **unsafe**; A's request should have been deferred.

Tanenbaum MOS Figure 6-10.

Banker's Algorithm: A Single Resource

- On every request: "If I grant this, will the state remain **safe**?"
- Yes → grant. No → **defer** (the process waits and the request is reconsidered later).
- Safety: find a customer whose remaining need can be met from what's free, assume it finishes, add its credit back to the vault; repeat.

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

Max sums to 22 > 10, yet safe

(a) The banker (Dijkstra, 1965): 10 units in the vault, 4 customers whose credit limits total 22. Not everyone asks for their maximum at once.

Step 1 / 3

A state is safe only if **all** processes, not just *one*, can finish in some order. The banker doesn't resolve deadlocks after the fact: if every request goes through the algorithm and maximum needs were declared correctly, the system never enters deadlock.

Banker's Algorithm: A Single Resource

- On every request: "If I grant this, will the state remain **safe**?"
- Yes → grant. No → **defer** (the process waits and the request is reconsidered later).
- Safety: find a customer whose remaining need can be met from what's free, assume it finishes, add its credit back to the vault; repeat.

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

safe sequence: C, D, A, B

(b) 2 free. C's need 2 ≤ 2 → C finishes (4 free), then D (3 ≤ 4 → 8 free), A, B. **Safe.**

A state is safe only if **all** processes, not just *one*, can finish in some order. The banker doesn't resolve deadlocks after the fact: if every request goes through the algorithm and maximum needs were declared correctly, the system never enters deadlock.

Banker's Algorithm: A Single Resource

- On every request: "If I grant this, will the state remain **safe**?"
- Yes → grant. No → **defer** (the process waits and the request is reconsidered later).
- Safety: find a customer whose remaining need can be met from what's free, assume it finishes, add its credit back to the vault; repeat.

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

no need ≤ 1: unsafe

(c) If B asks for one more unit in (b): 1 free, smallest need 2. **Unsafe** → the request must be refused and B made to wait.

A state is safe only if **all** processes, not just *one*, can finish in some order. The banker doesn't resolve deadlocks after the fact: if every request goes through the algorithm and maximum needs were declared correctly, the system never enters deadlock.

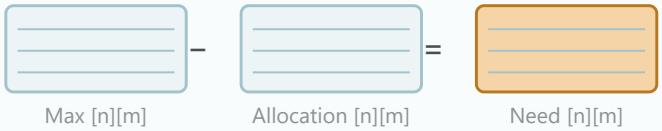
Banker's Algorithm: Multiple Resources

- `Available[m]` : free instances of each type.
- `Max[n][m]` : each process's maximum demand, **declared in advance**.
- `Allocation[n][m]` : currently held.
- `Need = Max − Allocation` : remaining need.

```
/* safety algorithm */
Work ← Available
Finish[i] ← false  (all i)
while some i has Finish[i] = false
      and Need[i] ≤ Work:
  Work ← Work + Allocation[i]
  Finish[i] ← true
SAFE if Finish[i] for all i
```

```
/* Ti requests Request[i] */
if Request[i] > Need[i]:
  error  /* exceeds Max */
if Request[i] > Available:
  Ti waits  /* not available */
/* pretend to allocate: */
Available  -= Request[i]
Allocation[i] += Request[i]
Need[i]    -= Request[i]
if safe: grant the request
else: restore, Ti waits
```

Complexity: the safety test is $O(m \cdot n^2)$ and runs on every request.



The Safety Algorithm: An Example

```
Work ← Available
Finish[i] ← false
for i = T0 … T4 (repeatedly):
  if Finish[i] = false and
     Need[i] ≤ Work:
    Work ← Work + Allocation[i]
    Finish[i] ← true
if all Finish[i]: SAFE
```

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

3	3	2

Order: —

A=10, B=5, C=7 instances. Available = (3 3 2). Need = Max − Allocation has been computed.

Step 1 / 9

VARIABLES

`Work = (3 3 2)` `Order = –`

The example from Silberschatz 10e §8.6.3.3. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

The Safety Algorithm: An Example

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (repeatedly):
▶   if Finish[i] = false and
▶      Need[i] ≤ Work:
6     Work ← Work + Allocation[i]
7     Finish[i] ← true
8 if all Finish[i]: SAFE
```

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

3	3	2

Order: —

(7 4 3) ≤ (3 3 2)? ✗

T0: Need (7 4 3) ≤ (3 3 2)? No, skip.

Step 2 / 9

VARIABLES

`Work = (3 3 2)` `Order = –`

The example from Silberschatz 10e §8.6.3.3. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

The Safety Algorithm: An Example

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (repeatedly):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
▶     Work ← Work + Allocation[i]
▶     Finish[i] ← true
8 if all Finish[i]: SAFE
```

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

5	3	2

Order: T1

(1 2 2) ≤ (3 3 2) ✓ Work += (2 0 0)

T1: (1 2 2) ≤ (3 3 2) ✓ → finishes, Work = (3 3 2) + (2 0 0) = (5 3 2).

Step 3 / 9

VARIABLES

`Work = (5 3 2)` `Order = T1`

The example from Silberschatz 10e §8.6.3.3. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

The Safety Algorithm: An Example

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (repeatedly):
▶   if Finish[i] = false and
▶      Need[i] ≤ Work:
6     Work ← Work + Allocation[i]
7     Finish[i] ← true
8 if all Finish[i]: SAFE
```

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

5	3	2

Order: T1

(6 0 0) ≤ (5 3 2)? ✗

T2: (6 0 0) ≤ (5 3 2)? Not enough A ✗.

Step 4 / 9

VARIABLES

`Work = (5 3 2)` `Order = T1`

The example from Silberschatz 10e §8.6.3.3. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

The Safety Algorithm: An Example

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (repeatedly):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
▶     Work ← Work + Allocation[i]
▶     Finish[i] ← true
8 if all Finish[i]: SAFE
```

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

7	4	3

Order: T1 → T3

(0 1 1) ≤ (5 3 2) ✓ Work += (2 1 1)

T3: (0 1 1) ✓ → Work = (7 4 3).

Step 5 / 9

VARIABLES

`Work = (7 4 3)` `Order = T1, T3`

The example from Silberschatz 10e §8.6.3.3. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

The Safety Algorithm: An Example

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (repeatedly):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
      Work ← Work + Allocation[i]
      Finish[i] ← true
8 if all Finish[i]: SAFE
```

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

7	4	5

Order: T1 → T3 → T4

(4 3 1) ≤ (7 4 3) ✓ Work += (0 0 2)

T4: (4 3 1) ✓ → Work = (7 4 5).

Step 6 / 9

VARIABLES

`Work = (7 4 5)` `Order = T1, T3, T4`

The example from Silberschatz 10e §8.6.3.3. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

The Safety Algorithm: An Example

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (repeatedly):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
      Work ← Work + Allocation[i]
      Finish[i] ← true
8 if all Finish[i]: SAFE
```

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

7	5	5

Order: T1 → T3 → T4 → T0

second round: T0 (7 4 3) ≤ (7 4 5) ✓

Second round. T0: (7 4 3) ≤ (7 4 5) ✓ → Work = (7 5 5).

Step 7 / 9

VARIABLES

`Work = (7 5 5)` `Order = T1, T3, T4, T0`

The example from Silberschatz 10e §8.6.3.3. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

The Safety Algorithm: An Example

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (repeatedly):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
▶     Work ← Work + Allocation[i]
▶     Finish[i] ← true
8 if all Finish[i]: SAFE
```

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

10	5	7

Order: T1 → T3 → T4 → T0 → T2

T2 (6 0 0) ≤ (7 5 5) ✓ Work = total

T2: (6 0 0) ✓ → Work = (10 5 7), i.e., all resources.

Step 8 / 9

VARIABLES

`Work = (10 5 7)` `Order = T1, T3, T4, T0, T2`

The example from Silberschatz 10e §8.6.3.3. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

The Safety Algorithm: An Example

```
1 Work ← Available
2 Finish[i] ← false
3 for i = T0 … T4 (repeatedly):
4   if Finish[i] = false and
5      Need[i] ≤ Work:
6     Work ← Work + Allocation[i]
7     Finish[i] ← true
▶ if all Finish[i]: SAFE
```

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

10	5	7

Order: T1 → T3 → T4 → T0 → T2

SAFE

All Finish true: the state is **safe**. Other safe sequences exist too, such as <T1, T3, T4, T2, T0>.

Step 9 / 9

VARIABLES

`Work = (10 5 7)` `Order = T1, T3, T4, T0, T2`

The example from Silberschatz 10e §8.6.3.3. Max: T0 (7 5 3), T1 (3 2 2), T2 (9 0 2), T3 (2 2 2), T4 (4 3 3).

Granting a Request: T1 (1 0 2)

- Three steps: the **Need** check, the **Available** check, and the **safety** test.
- The safety test runs on the "pretend" state; had the state been unsafe, the changes would be rolled back.
- Calculations verified with a Python script.

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Avail

3	3	2

$\text{Request}_1 = (1\ 0\ 2) \leq \text{Need}_1 = (1\ 2\ 2)$ ✓

T1 requests (1 0 2). Check 1: it doesn't exceed its declared maximum need ✓.

Granting a Request: T1 (1 0 2)

- Three steps: the **Need** check, the **Available** check, and the **safety** test.
- The safety test runs on the "pretend" state; had the state been unsafe, the changes would be rolled back.
- Calculations verified with a Python script.

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Avail

3	3	2

(1 0 2) ≤ Available = (3 3 2) ✓

Check 2: the resources are free right now ✓. But that alone is not enough.

Granting a Request: T1 (1 0 2)

- Three steps: the **Need** check, the **Available** check, and the **safety** test.
- The safety test runs on the "pretend" state; had the state been unsafe, the changes would be rolled back.
- Calculations verified with a Python script.

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Avail

2	3	0

pretend: Avail (2 3 0), $Alloc_1$ (3 0 2), $Need_1$ (0 2 0)

The new state is computed **as if** the request were granted.

Granting a Request: T1 (1 0 2)

- Three steps: the **Need** check, the **Available** check, and the **safety** test.
- The safety test runs on the "pretend" state; had the state been unsafe, the changes would be rolled back.
- Calculations verified with a Python script.

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

10	5	7

Order: T1 → T3 → T4 → T0 → T2

new state safe → request GRANTED

The safety algorithm finds the sequence <T1, T3, T4, T0, T2> in the new state: **safe → the request is granted**.

Refusing Requests: T4 (3 3 0) and T0 (0 2 0)

- **Reason to wait 1:** resources unavailable (T4).
- **Reason to wait 2:** resources available, but granting would make the state unsafe (T0).
- This is the price of avoidance: a request is deferred even while resources sit idle, lowering utilization.



T4: (3 3 0) ≤ Avail (2 3 0)? ✗ → wait

After T1's request is granted, T4 requests (3 3 0). It doesn't exceed Need, but only 2 instances of type A are free: **T4 waits**.

Refusing Requests: T4 (3 3 0) and T0 (0 2 0)

- **Reason to wait 1:** resources unavailable (T4).
- **Reason to wait 2:** resources available, but granting would make the state unsafe (T0).
- This is the price of avoidance: a request is deferred even while resources sit idle, lowering utilization.

Allocation

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

Need

	A	B	C
T0	7	4	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Avail

2	3	0

T0: (0 2 0) ≤ $Need_0$ ✓ and ≤ Avail (2 3 0) ✓

T0 requests (0 2 0). Both checks pass; the resources *are* free.

Refusing Requests: T4 (3 3 0) and T0 (0 2 0)

- **Reason to wait 1:** resources unavailable (T4).
- **Reason to wait 2:** resources available, but granting would make the state unsafe (T0).
- This is the price of avoidance: a request is deferred even while resources sit idle, lowering utilization.

Allocation

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

Need

	A	B	C
T0	7	2	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

2	1	0

pretend: Avail (2 1 0), $Need_0$ (7 2 3)

Pretend: Available = (2 1 0). Which Need ≤ (2 1 0)? T1 (0 2 0): not enough B; T3 (0 1 1): not enough C…

Refusing Requests: T4 (3 3 0) and T0 (0 2 0)

- **Reason to wait 1:** resources unavailable (T4).
- **Reason to wait 2:** resources available, but granting would make the state unsafe (T0).
- This is the price of avoidance: a request is deferred even while resources sit idle, lowering utilization.

Allocation

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

Need

	A	B	C
T0	7	2	3
T1	0	2	0
T2	6	0	0
T3	0	1	1
T4	4	3	1

Work

2	1	0

Order: —

no row fits → UNSAFE → REFUSE

No process can finish: the state is **unsafe**. The changes are rolled back and T0 waits, even though the resources are free.

Banker's Request Algorithm: Code Trace

```
1  /* Ti requests Request */
▶  if Request > Need[i]: error /* exceeds Max */
3  if Request > Available: Ti waits
4  Available -= Request  /* pretend */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if safe(): GRANT the request
8  else: restore, Ti waits
9  safe():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (repeatedly):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
15   return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 1 1	0 1 1	—
T4	0 0 2	4 3 1	—

A safe initial state: Available = (3 3 2). T3 requests **Request = (0 1 1)**. Need[3] = (0 1 1): the request doesn't exceed the declared maximum.

Step 1 / 10

VARIABLES

`Request = (0 1 1)` `Need[3] = (0 1 1)` `Available = (3 3 2)`

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Banker's Request Algorithm: Code Trace

```
1  /* Ti requests Request */
2  if Request > Need[i]: error /* exceeds Max */
▶  if Request > Available: Ti waits
4  Available -= Request  /* pretend */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if safe(): GRANT the request
8  else: restore, Ti waits
9  safe():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (repeatedly):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
15   return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 1 1	0 1 1	—
T4	0 0 2	4 3 1	—

(0 1 1) ≤ Available (3 3 2): the resources are there now. Still, it isn't granted immediately.

Step 2 / 10

VARIABLES

`Request = (0 1 1)` `Need[3] = (0 1 1)` `Available = (3 3 2)`

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Banker's Request Algorithm: Code Trace

```
1  /* Ti requests Request */
2  if Request > Need[i]: error /* exceeds Max */
3  if Request > Available: Ti waits
▶  Available -= Request  /* pretend */
▶  Allocation[i] += Request
▶  Need[i] -= Request
7  if safe(): GRANT the request
8  else: restore, Ti waits
9  safe():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (repeatedly):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
15   return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	—
T4	0 0 2	4 3 1	—

Pretend to allocate: Available = **(3 2 1)**, Allocation[3] = (2 2 2), Need[3] = (0 0 0).

Step 3 / 10

VARIABLES

`Available = (3 2 1)`

`Allocation[3] = (2 2 2)`

`Need[3] = (0 0 0)`

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Banker's Request Algorithm: Code Trace

```
1  /* Ti requests Request */
2  if Request > Need[i]: error /* exceeds Max */
3  if Request > Available: Ti waits
4  Available -= Request  /* pretend */
5  Allocation[i] += Request
6  Need[i] -= Request
▶  if safe(): GRANT the request
8  else: restore, Ti waits
9  safe():
▶    Work ← Available; Finish ← false
11   for i = T0 … T4 (repeatedly):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
15   return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	—
T4	0 0 2	4 3 1	—

Work (3 2 1)

Safety test: Work = (3 2 1). T0 (7 4 3), T1 (1 2 2), T2 (6 0 0) can't finish with this Work.

Step 4 / 10

VARIABLES

`Work = (3 2 1)` `Order = –`

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Banker's Request Algorithm: Code Trace

```
1  /* Ti requests Request */
2  if Request > Need[i]: error /* exceeds Max */
3  if Request > Available: Ti waits
4  Available -= Request  /* pretend */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if safe(): GRANT the request
8  else: restore, Ti waits
9  safe():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (repeatedly):
▶      if !Finish[i] and Need[i] ≤ Work:
▶        Work ← Work + Allocation[i]
▶        Finish[i] ← true
15   return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	—

Work (5 4 3)

T3: Need (0 0 0) ≤ Work (3 2 1) ✓ → finishes, Work = (3 2 1) + (2 2 2) = **(5 4 3)**.

Step 5 / 10

VARIABLES

`Work = (5 4 3)` `Order = T3`

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Banker's Request Algorithm: Code Trace

```
1  /* Ti requests Request */
2  if Request > Need[i]: error /* exceeds Max */
3  if Request > Available: Ti waits
4  Available -= Request  /* pretend */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if safe(): GRANT the request
8  else: restore, Ti waits
9  safe():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (repeatedly):
▶      if !Finish[i] and Need[i] ≤ Work:
▶        Work ← Work + Allocation[i]
▶        Finish[i] ← true
15   return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	—
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (5 4 5)

T4: Need (4 3 1) ≤ Work (5 4 3) ✓ → finishes, Work = (5 4 3) + (0 0 2) = **(5 4 5)**.

Step 6 / 10

VARIABLES

`Work = (5 4 5)` `Order = T3, T4`

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Banker's Request Algorithm: Code Trace

```
/* Ti requests Request */
if Request > Need[i]: error /* exceeds Max */
if Request > Available: Ti waits
Available -= Request  /* pretend */
Allocation[i] += Request
Need[i] -= Request
if safe(): GRANT the request
else: restore, Ti waits
safe():
  Work ← Available; Finish ← false
  for i = T0 … T4 (repeatedly):
    if !Finish[i] and Need[i] ≤ Work:
      Work ← Work + Allocation[i]
      Finish[i] ← true
  return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	✓
T2	3 0 2	6 0 0	—
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (7 4 5)

Second round. Not enough A for T0 (7 4 3); T1: Need (1 2 2) ≤ Work (5 4 5) ✓ → finishes, Work = (5 4 5) + (2 0 0) = **(7 4 5)**.

Step 7 / 10

VARIABLES

`Work = (7 4 5)` `Order = T3, T4, T1`

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Banker's Request Algorithm: Code Trace

```
1  /* Ti requests Request */
2  if Request > Need[i]: error /* exceeds Max */
3  if Request > Available: Ti waits
4  Available -= Request  /* pretend */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if safe(): GRANT the request
8  else: restore, Ti waits
9  safe():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (repeatedly):
▶      if !Finish[i] and Need[i] ≤ Work:
▶        Work ← Work + Allocation[i]
▶        Finish[i] ← true
15   return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	—
T1	2 0 0	1 2 2	✓
T2	3 0 2	6 0 0	✓
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (10 4 7)

T2: Need (6 0 0) ≤ Work (7 4 5) ✓ → finishes, Work = (7 4 5) + (3 0 2) = **(10 4 7)**.

Step 8 / 10

VARIABLES

`Work` = **(10 4 7)** `Order` = **T3, T4, T1, T2**

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Banker's Request Algorithm: Code Trace

```
1  /* Ti requests Request */
2  if Request > Need[i]: error /* exceeds Max */
3  if Request > Available: Ti waits
4  Available -= Request  /* pretend */
5  Allocation[i] += Request
6  Need[i] -= Request
7  if safe(): GRANT the request
8  else: restore, Ti waits
9  safe():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (repeatedly):
▶      if !Finish[i] and Need[i] ≤ Work:
▶        Work ← Work + Allocation[i]
▶        Finish[i] ← true
15   return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	✓
T1	2 0 0	1 2 2	✓
T2	3 0 2	6 0 0	✓
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (10 5 7)

Third round: T0: Need (7 4 3) ≤ Work (10 4 7) ✓ → finishes, Work = (10 4 7) + (0 1 0) = **(10 5 7)**.

Step 9 / 10

VARIABLES

`Work = (10 5 7)`

`Order = T3, T4, T1, T2, T0`

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Banker's Request Algorithm: Code Trace

```
1  /* Ti requests Request */
2  if Request > Need[i]: error /* exceeds Max */
3  if Request > Available: Ti waits
4  Available -= Request  /* pretend */
5  Allocation[i] += Request
6  Need[i] -= Request
▶  if safe(): GRANT the request
8  else: restore, Ti waits
9  safe():
10   Work ← Available; Finish ← false
11   for i = T0 … T4 (repeatedly):
12     if !Finish[i] and Need[i] ≤ Work:
13       Work ← Work + Allocation[i]
14       Finish[i] ← true
▶    return all Finish[i]
```

	Allocation	Need	Finish
T0	0 1 0	7 4 3	✓
T1	2 0 0	1 2 2	✓
T2	3 0 2	6 0 0	✓
T3	2 2 2	0 0 0	✓
T4	0 0 2	4 3 1	✓

Work (10 5 7)

All Finish true → safe sequence <T3, T4, T1, T2, T0>. The request is **granted** and the state becomes permanent.

Step 10 / 10

VARIABLES

`Work = (10 5 7)`

`Order = T3, T4, T1, T2, T0`

`result = GRANT`

The same Silberschatz state; only the request differs. Values verified by a Python simulation.

Multiple Resources: A Second Example

Assigned

	Tp	Pl	Pr	CD
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Still needed

	Tp	Pl	Pr	CD
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

```
E = (6 3 4 2)
P = (5 3 2 2)
A = (1 0 2 0)
```

E = existing, P = total assigned, A = E − P (available)

Five processes, four resource types (Tp, Pl, Pr = printer, CD). A = E − P = (1 0 2 0).

Step 1 / 4

Tanenbaum MOS Figure 6-12. The "find a row ≤ A" steps here are the safety test; the same logic is used with the *request* matrix in detection and with the *remaining-need* matrix in avoidance.

Multiple Resources: A Second Example

Assigned

	Tp	Pl	Pr	CD
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Still needed

	Tp	Pl	Pr	CD
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

```
E = (6 3 4 2)
P = (5 3 2 2)
A = (1 0 2 0)
```

safe sequence: D, E, A, B, C

D (0 0 1 0) ≤ A → finishes, A = (2 1 2 1); E, A, B, C then finish in turn. The state is safe.

Tanenbaum MOS Figure 6-12. The "find a row ≤ A" steps here are the safety test; the same logic is used with the *request* matrix in detection and with the *remaining-need* matrix in avoidance.

Multiple Resources: A Second Example

Assigned

	Tp	Pl	Pr	CD
A	3	0	1	1
B	0	1	1	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Still needed

	Tp	Pl	Pr	CD
A	1	1	0	0
B	0	1	0	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

```
E = (6 3 4 2)
P = (5 3 3 2)
A = (1 0 1 0)
```

B requests a printer → A = (1 0 1 0), sequence D, E, A, B, C still exists → GRANT

B requests a printer. If granted, A = (1 0 1 0); D can still finish and the sequence survives: **granted**.

Tanenbaum MOS Figure 6-12. The "find a row ≤ A" steps here are the safety test; the same logic is used with the *request* matrix in detection and with the *remaining-need* matrix in avoidance.

Multiple Resources: A Second Example

Assigned

	Tp	Pl	Pr	CD
A	3	0	1	1
B	0	1	1	0
C	1	1	1	0
D	1	1	0	1
E	0	0	1	0

Still needed

	Tp	Pl	Pr	CD
A	1	1	0	0
B	0	1	0	2
C	3	1	0	0
D	0	0	1	0
E	2	1	0	0

E = (6 3 4 2)
P = (5 3 4 2)
A = (1 0 0 0)

then E requests the last printer → A = (1 0 0 0), no one can finish → DEFER

If E then requests the last printer, A becomes (1 0 0 0) and no row can be satisfied: unsafe → **deferred**.

Tanenbaum MOS Figure 6-12. The "find a row ≤ A" steps here are the safety test; the same logic is used with the *request* matrix in detection and with the *remaining-need* matrix in avoidance.

The Banker's Algorithm in Practice

Assumptions

Each process declares its maximum need *in advance*; the number of processes is fixed; the number of resources doesn't change (no device fails); processes release resources in finite time.

Real systems

These assumptions don't hold in general-purpose OSes, and the algorithm is almost never used. Related "admission control" ideas: real-time scheduling, cloud quotas, pooled connection limits.

- Because of the $O(m \cdot n^2)$ test on every request and unnecessary deferrals, avoidance is usually **more expensive** than detection and recovery; its advantage is that no recovery is needed.
- The algorithm doesn't run continuously in the background: the test runs only **when a request arrives** (and, for waiters, when a resource is released).

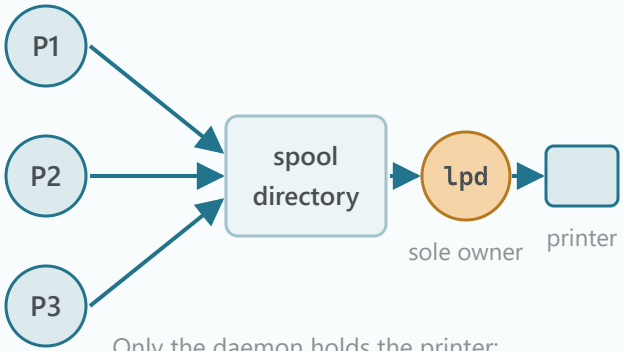
Prevention: Attacking the Conditions

Condition	Attack	Cost
Mutual exclusion	Make the resource shareable: spooling, read-only sharing, lock-free structures	Most resources are inherently unshareable (a mutex!)
Hold and wait	Request all resources at once up front, or release what you hold before a new request	Low utilization, starvation; needs must be known in advance
No preemption	Take back a waiting process's resources; `tryLock` + back off; virtualize	Only for resources whose state can be saved; risk of livelock
Circular wait	Impose a total order on resources and request only in increasing order (lock ordering)	Following the order takes programmer discipline

Eliminating mutual exclusion does **not** mean "only one process may request a resource at a time" (that is exclusion itself). Eliminating "no preemption" doesn't mean "keep resources until the process finishes"; on the contrary, it means **resources can be taken back**.

Attacking Mutual Exclusion

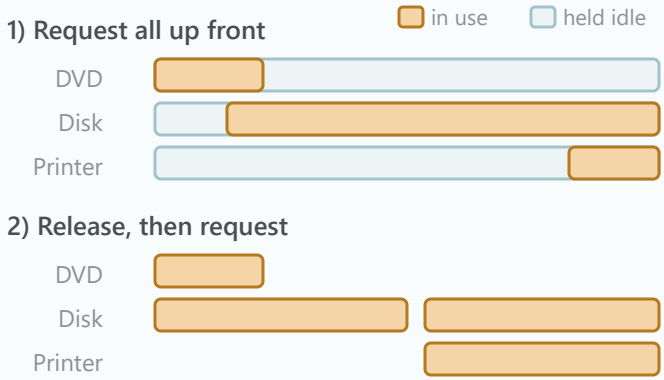
- **Spooling:** processes write their output to disk; a single daemon uses the printer in turn. Contention for the printer disappears.
- **Read-only** files and code pages are shared simultaneously; reader locks don't exclude each other.
- **Lock-free** structures progress with CAS; with no locks there is no deadlock (starvation is still possible).
- The limit: the spool disk is itself a resource. If it fills with half-written output, deadlock can still occur (Tanenbaum).
- General principle: **don't assign** a resource unless it is really needed, and **minimize** the number of processes that may claim it.



Only the daemon holds the printer: user processes never wait for it

Attacking Hold and Wait

- **Protocol 1:** a process acquires *all* the resources it will need in a single request before it starts (all or nothing).
- **Protocol 2:** a process may request resources only when it holds none: it **releases** what it holds before requesting more.
- Example: copy from DVD to disk, sort the file, print it. Under Protocol 1 the printer is held idle from start to finish.
- Problems: **low resource utilization**; **starvation** for a process that needs many resources; needs are often unknown in advance.
- In practice: "all or nothing" acquisition with `pthread_mutex_trylock`; conservative 2PL in databases, which takes all locks up front.



Silberschatz 10e §8.5.2.

Attacking No Preemption

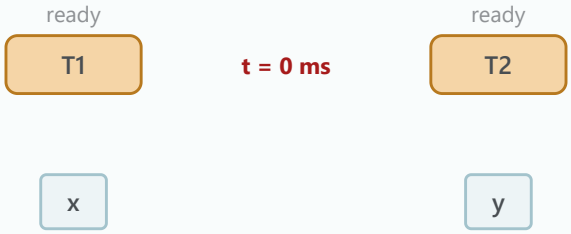
- **Preemption protocol:** if a process holding resources makes a request that cannot be granted at once, all its resources are *implicitly released*; it restarts only when it can get them all together.
- Suitable only for resources whose state can be saved and restored: CPU registers, memory, database transactions (abort).
- **Virtualization:** the printer is spooled, memory is paged.
- For mutexes: **timed attempts**. A thread that can't acquire a lock releases what it holds itself and retries after a random delay.
- Run on JDK 21: 2 threads × 1000 transfers in opposite directions, result `x=1000 y=1000 total=2000`.

```
boolean tryXfer(Account a, Account b, int m)
        throws InterruptedException {
  if (!a.lock.tryLock(50, MILLISECONDS))
    return false;
  try {
    if (!b.lock.tryLock(50, MILLISECONDS))
      return false;  /* a released too */
    try { a.balance -= m; b.balance += m;
          return true; }
    finally { b.lock.unlock(); }
  } finally { a.lock.unlock(); }
}
/* caller: while (!tryXfer(x, y, 1))
              Thread.sleep(random); */
```

The random delay matters: if two threads try and release in lockstep, a **livelock** results.

Backing Off with tryLock: Code Trace

```
▶ boolean tryXfer(Account a, Account b, int m)
2         throws InterruptedException {
3   if (!a.lock.tryLock(50, MILLISECONDS))
4     return false;
5   try {
6     if (!b.lock.tryLock(50, MILLISECONDS))
7       return false;  /* a released too */
8     try { a.balance -= m; b.balance += m;
9           return true; }
10    finally { b.lock.unlock(); }
11  } finally { a.lock.unlock(); }
12 }
13 /* caller: while (!tryXfer(x, y, 1))
14               Thread.sleep(random); */
```



T1 moves 1 from x to y, T2 from y to x: they request the locks in **opposite order**. Both locks are free.

Step 1 / 10

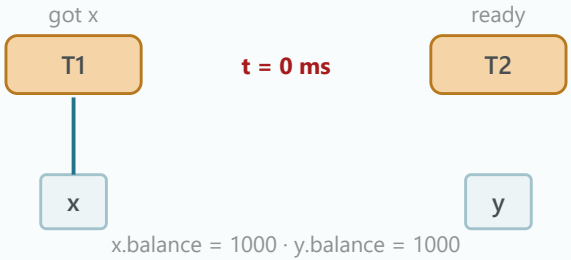
VARIABLES

`t = 0 ms` `x = free` `y = free` `x.balance = 1000` `y.balance = 1000`

OUTPUT

Backing Off with tryLock: Code Trace

```
1  boolean tryXfer(Account a, Account b, int m)
2          throws InterruptedException {
3    if (!a.lock.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!b.lock.tryLock(50, MILLISECONDS))
7        return false;  /* a released too */
8      try { a.balance -= m; b.balance += m;
9            return true; }
10     finally { b.lock.unlock(); }
11   } finally { a.lock.unlock(); }
12 }
13 /* caller: while (!tryXfer(x, y, 1))
14               Thread.sleep(random); */
```



T1 takes `x` at once.

Step 2 / 10

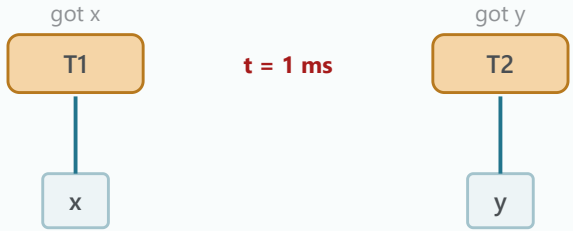
VARIABLES

`t = 0 ms` `x = T1` `y = free` `x.balance = 1000` `y.balance = 1000`

OUTPUT

Backing Off with tryLock: Code Trace

```
1  boolean tryXfer(Account a, Account b, int m)
2          throws InterruptedException {
3    if (!a.lock.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!b.lock.tryLock(50, MILLISECONDS))
7        return false;  /* a released too */
8      try { a.balance -= m; b.balance += m;
9            return true; }
10     finally { b.lock.unlock(); }
11   } finally { a.lock.unlock(); }
12 }
13 /* caller: while (!tryXfer(x, y, 1))
14               Thread.sleep(random); */
```



T2 takes `y`. With a plain `lock()`, the next step would make deadlock certain.

Step 3 / 10

VARIABLES

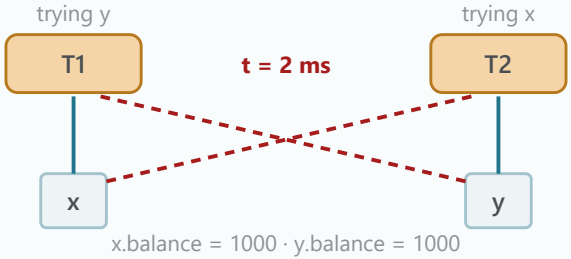
`t = 1 ms` `x = T1` `y = T2`

`x.balance = 1000` `y.balance = 1000`

OUTPUT

Backing Off with tryLock: Code Trace

```
1  boolean tryXfer(Account a, Account b, int m)
2          throws InterruptedException {
3    if (!a.lock.tryLock(50, MILLISECONDS))
4      return false;
5    try {
▶      if (!b.lock.tryLock(50, MILLISECONDS))
7        return false;  /* a released too */
8      try { a.balance -= m; b.balance += m;
9            return true; }
10     finally { b.lock.unlock(); }
11   } finally { a.lock.unlock(); }
12 }
13 /* caller: while (!tryXfer(x, y, 1))
14               Thread.sleep(random); */
```



Each waits **at most 50 ms** for its second lock: a circular wait has formed, but a timed one.

Step 4 / 10

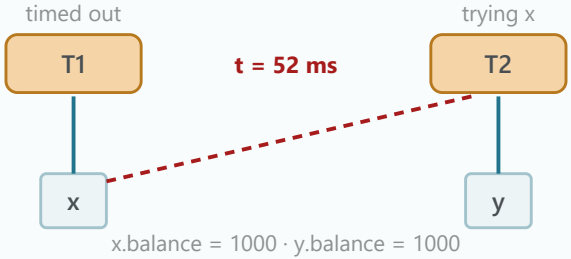
VARIABLES

`t = 2 ms` `x = T1` `y = T2` `x.balance = 1000` `y.balance = 1000`

OUTPUT

Backing Off with tryLock: Code Trace

```
1  boolean tryXfer(Account a, Account b, int m)
2          throws InterruptedException {
3    if (!a.lock.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!b.lock.tryLock(50, MILLISECONDS))
▶        return false;  /* a released too */
8      try { a.balance -= m; b.balance += m;
9            return true; }
10     finally { b.lock.unlock(); }
11   } finally { a.lock.unlock(); }
12 }
13 /* caller: while (!tryXfer(x, y, 1))
14               Thread.sleep(random); */
```



T1's 50 ms are up: `tryLock` returns false → `return false`.

Step 5 / 10

VARIABLES

`t = 52 ms` `x = T1` `y = T2`

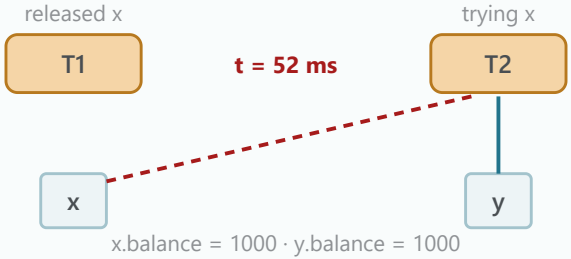
`x.balance = 1000` `y.balance = 1000`

OUTPUT

```
T1: false
```

Backing Off with tryLock: Code Trace

```
1  boolean tryXfer(Account a, Account b, int m)
2          throws InterruptedException {
3    if (!a.lock.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!b.lock.tryLock(50, MILLISECONDS))
7        return false;  /* a released too */
8      try { a.balance -= m; b.balance += m;
9            return true; }
10     finally { b.lock.unlock(); }
▶    } finally { a.lock.unlock(); }
12 }
13 /* caller: while (!tryXfer(x, y, 1))
14               Thread.sleep(random); */
```



`finally` makes T1 **release** the `x` it holds: the no-preemption condition is broken voluntarily.

Step 6 / 10

VARIABLES

`t = 52 ms` `x = free` `y = T2` `x.balance = 1000` `y.balance = 1000`

OUTPUT

```
T1: false
```

Backing Off with tryLock: Code Trace

```
1  boolean tryXfer(Account a, Account b, int m)
2          throws InterruptedException {
3    if (!a.lock.tryLock(50, MILLISECONDS))
4      return false;
5    try {
▶      if (!b.lock.tryLock(50, MILLISECONDS))
7        return false;  /* a released too */
8      try { a.balance -= m; b.balance += m;
9            return true; }
10     finally { b.lock.unlock(); }
11   } finally { a.lock.unlock(); }
12 }
13 /* caller: while (!tryXfer(x, y, 1))
14               Thread.sleep(random); */
```



T2 is still within its own 50 ms: it gets `x` once it is free.

Step 7 / 10

VARIABLES

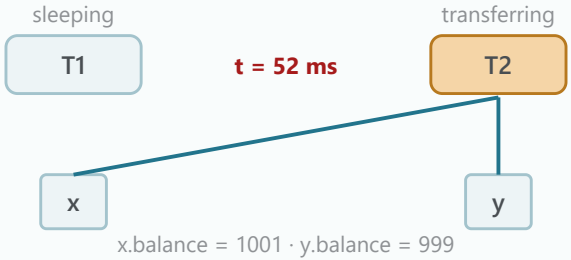
`t = 52 ms` `x = T2` `y = T2` `x.balance = 1000` `y.balance = 1000`

OUTPUT

```
T1: false
```

Backing Off with tryLock: Code Trace

```
1  boolean tryXfer(Account a, Account b, int m)
2          throws InterruptedException {
3    if (!a.lock.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!b.lock.tryLock(50, MILLISECONDS))
7        return false;  /* a released too */
▶      try { a.balance -= m; b.balance += m;
9            return true; }
10     finally { b.lock.unlock(); }
11   } finally { a.lock.unlock(); }
12 }
13 /* caller: while (!tryXfer(x, y, 1))
14               Thread.sleep(random); */
```



T2 transfers: y.balance = 999, x.balance = 1001.

Step 8 / 10

VARIABLES

`t = 52 ms` `x = T2` `y = T2`

`x.balance = 1001` `y.balance = 999`

OUTPUT

```
T1: false
```

Backing Off with tryLock: Code Trace

```
1  boolean tryXfer(Account a, Account b, int m)
2          throws InterruptedException {
3    if (!a.lock.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!b.lock.tryLock(50, MILLISECONDS))
7        return false;  /* a released too */
8      try { a.balance -= m; b.balance += m;
9            return true; }
▶      finally { b.lock.unlock(); }
11   } finally { a.lock.unlock(); }
12 }
13 /* caller: while (!tryXfer(x, y, 1))
14               Thread.sleep(random); */
```



T2 releases `x`, then `y`, and returns true.

Step 9 / 10

VARIABLES

`t = 53 ms` `x = free` `y = free`

`x.balance = 1001` `y.balance = 999`

OUTPUT

```
T1: false
T2: true
```

Backing Off with tryLock: Code Trace

```
1  boolean tryXfer(Account a, Account b, int m)
2          throws InterruptedException {
3    if (!a.lock.tryLock(50, MILLISECONDS))
4      return false;
5    try {
6      if (!b.lock.tryLock(50, MILLISECONDS))
7        return false;  /* a released too */
8      try { a.balance -= m; b.balance += m;
9            return true; }
10     finally { b.lock.unlock(); }
11   } finally { a.lock.unlock(); }
12 }
▶ /* caller: while (!tryXfer(x, y, 1))
▶                Thread.sleep(random); */
```



After a random sleep T1 retries, gets both locks, and transfers: the total is still 2000.

Step 10 / 10

VARIABLES

`t = 70 ms` `x = T1` `y = T1`

`x.balance = 1000` `y.balance = 1000`

OUTPUT

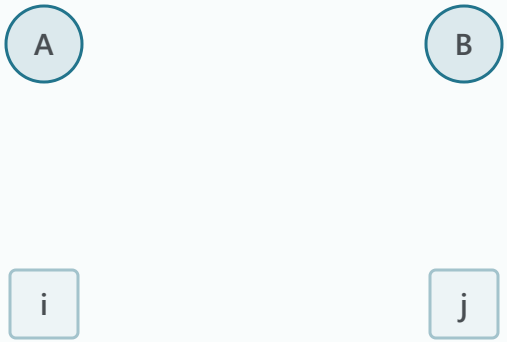
```
T1: false
T2: true
T1: true
```

PREVENTION

Attacking Circular Wait

#	Resource
1	Imagesetter
2	Printer
3	Plotter
4	Tape drive
5	Blu-ray drive

If a printer and a tape drive are needed, request 2 first, then 4. Finding a suitable ordering is the hardest part.



Every resource type gets a number. Rule: a process may request only resources numbered **higher than any it holds**.

Step 1 / 3

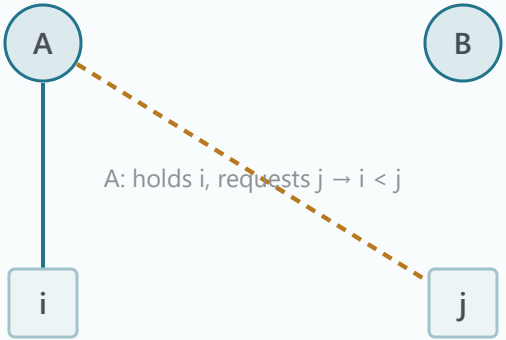
Tanenbaum MOS Figure 6-13. A variant: before requesting R_j, a process must release every R_i it holds with $i \geq j$.

PREVENTION

Attacking Circular Wait

#	Resource
1	Imagesetter
2	Printer
3	Plotter
4	Tape drive
5	Blu-ray drive

If a printer and a tape drive are needed, request 2 first, then 4. Finding a suitable ordering is the hardest part.



If A holds i and requests j, the rule implies i < j.

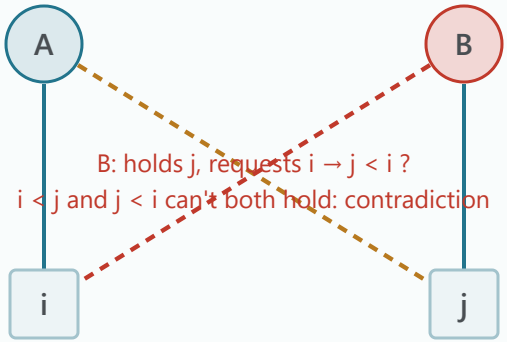
Step 2 / 3

Tanenbaum MOS Figure 6-13. A variant: before requesting R_j, a process must release every R_i it holds with $i \geq j$.

Attacking Circular Wait

#	Resource
1	Imagesetter
2	Printer
3	Plotter
4	Tape drive
5	Blu-ray drive

If a printer and a tape drive are needed, request 2 first, then 4. Finding a suitable ordering is the hardest part.

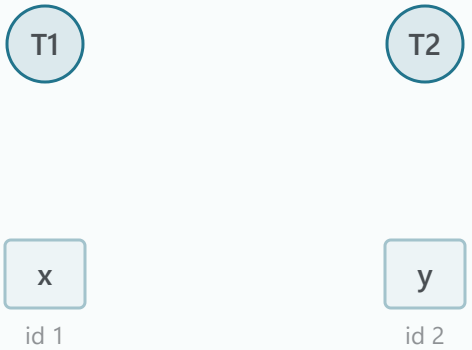


If B held j and requested i, then j < i: a contradiction. This edge **can never form**. In general, the process holding the highest-numbered resource in a cycle could not request a lower-numbered one.

Tanenbaum MOS Figure 6-13. A variant: before requesting R_j, a process must release every R_i it holds with $i \geq j$.

Lock Ordering in Practice

```
1  void transfer(Account a, Account b, int m) {
2    Account first = a.id < b.id ? a : b;
3    Account second = a.id < b.id ? b : a;
4    first.lock.lock();
5    try {
6      second.lock.lock();
7      try { a.balance -= m; b.balance += m; }
8      finally { second.lock.unlock(); }
9    } finally { first.lock.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T1 moves x→y and T2 moves y→x: the "natural" order is reversed between the two threads. The `id` order of the locks, however, is shared.

Step 1 / 6

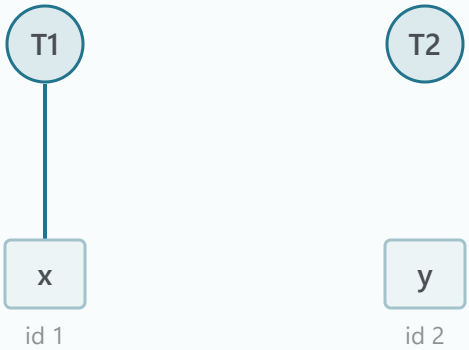
VARIABLES

`T1.first = ?` `T2.first = ?`

The Linux kernel takes two locks of the same type in address order (e.g., `double_rq_lock`); lock order is documented and checked by `lockdep`.

Lock Ordering in Practice

```
1  void transfer(Account a, Account b, int m) {
2    Account first = a.id < b.id ? a : b;
3    Account second = a.id < b.id ? b : a;
4    first.lock.lock();
5    try {
6      second.lock.lock();
7      try { a.balance -= m; b.balance += m; }
8      finally { second.lock.unlock(); }
9    } finally { first.lock.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T1: first = x (1 < 2). It acquires lock x.

Step 2 / 6

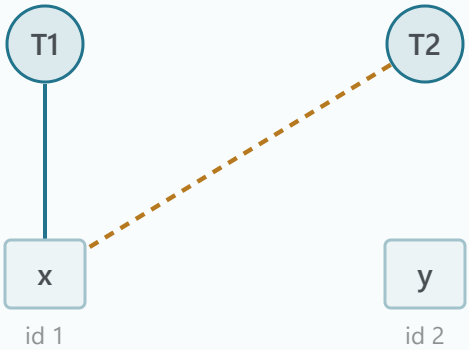
VARIABLES

`T1.first = x` `T2.first = ?`

The Linux kernel takes two locks of the same type in address order (e.g., `double_rq_lock`); lock order is documented and checked by `lockdep`.

Lock Ordering in Practice

```
1  void transfer(Account a, Account b, int m) {
2    Account first = a.id < b.id ? a : b;
3    Account second = a.id < b.id ? b : a;
▶    first.lock.lock();
5    try {
6      second.lock.lock();
7      try { a.balance -= m; b.balance += m; }
8      finally { second.lock.unlock(); }
9    } finally { first.lock.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T2: a = y, but first is still x (id 1). It waits for x **holding no lock at all**.

Step 3 / 6

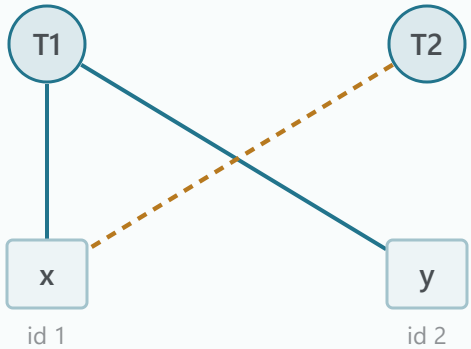
VARIABLES

`T1.first = x` `T2.first = x`

The Linux kernel takes two locks of the same type in address order (e.g., `double_rq_lock`); lock order is documented and checked by `lockdep`.

Lock Ordering in Practice

```
1  void transfer(Account a, Account b, int m) {
2    Account first = a.id < b.id ? a : b;
3    Account second = a.id < b.id ? b : a;
4    first.lock.lock();
5    try {
▶      second.lock.lock();
▶      try { a.balance -= m; b.balance += m; }
8      finally { second.lock.unlock(); }
9    } finally { first.lock.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T1 also gets y (no one holds it) and performs the transfer.

Step 4 / 6

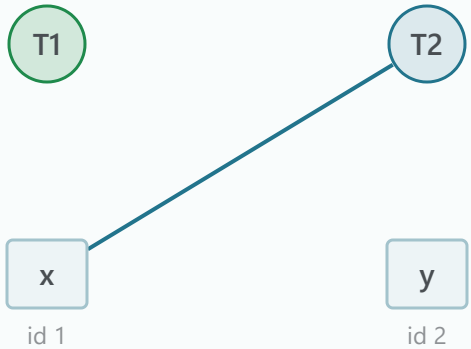
VARIABLES

`T1.first = x` `T2.first = x`

The Linux kernel takes two locks of the same type in address order (e.g., `double_rq_lock`); lock order is documented and checked by `lockdep`.

Lock Ordering in Practice

```
1  void transfer(Account a, Account b, int m) {
2    Account first = a.id < b.id ? a : b;
3    Account second = a.id < b.id ? b : a;
4    first.lock.lock();
5    try {
6      second.lock.lock();
7      try { a.balance -= m; b.balance += m; }
▶      finally { second.lock.unlock(); }
▶    } finally { first.lock.unlock(); }
10 }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



T1 releases its locks; T2 takes x, then y. No cycle can form.

Step 5 / 6

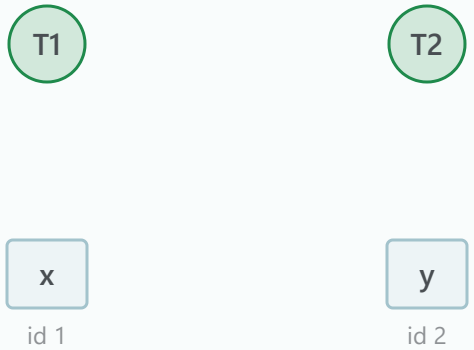
VARIABLES

`T1 = done` `T2.first = x`

The Linux kernel takes two locks of the same type in address order (e.g., `double_rq_lock`); lock order is documented and checked by `lockdep`.

Lock Ordering in Practice

```
1  void transfer(Account a, Account b, int m) {
2    Account first = a.id < b.id ? a : b;
3    Account second = a.id < b.id ? b : a;
4    first.lock.lock();
5    try {
6      second.lock.lock();
7      try { a.balance -= m; b.balance += m; }
8      finally { second.lock.unlock(); }
9    } finally { first.lock.unlock(); }
▶  }
11 /* T1: transfer(x, y, 1)   x.id = 1
12    T2: transfer(y, x, 1)   y.id = 2 */
```



Run on JDK 21: 100,000 opposite transfers each, result `x=1000 y=1000`, no deadlock.

Step 6 / 6

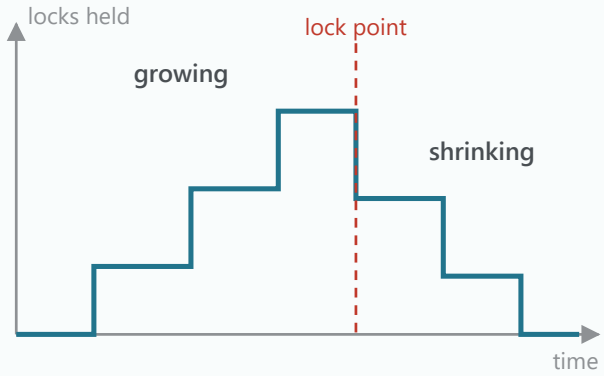
VARIABLES

`T1 = done` `T2 = done`

The Linux kernel takes two locks of the same type in address order (e.g., `double_rq_lock`); lock order is documented and checked by `lockdep`.

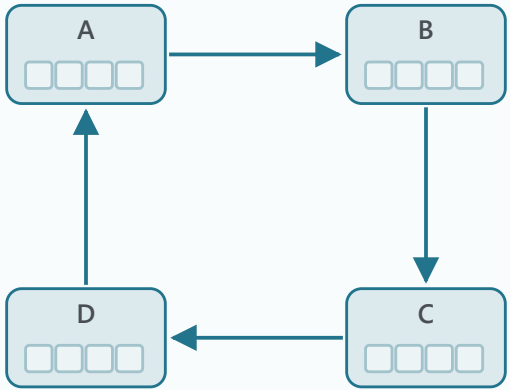
Two-Phase Locking (2PL)

- A protocol for database transactions: in the **growing** phase locks are only acquired, in the **shrinking** phase only released; after the first release, no new lock may be taken.
- Its goal is **serializability** (as if transactions ran one after another), not deadlock prevention. **Basic 2PL can deadlock:** T1 locks a, T2 locks b, then each requests the other's.
- Tanenbaum's variant: in phase 1, try to lock all records; if any is locked, **release them all** and start over. This removes hold and wait.
- Strict 2PL: locks are released at transaction end (commit/abort). The DBMS detects deadlocks with a wait-for graph and rolls back a transaction.



Communication Deadlocks

- **Communication deadlock:** there is no resource; processes wait for *messages* from each other. A sends B a request and waits for the reply; if the request is lost, A waits for the reply and B for the request, forever.
- Resource ordering and avoidance don't help; the cure is a **timeout** and retransmission. The protocol needs sequence numbers to handle duplicate messages.
- In network hardware, though, buffers are real resources; the ring below is a **resource deadlock**.



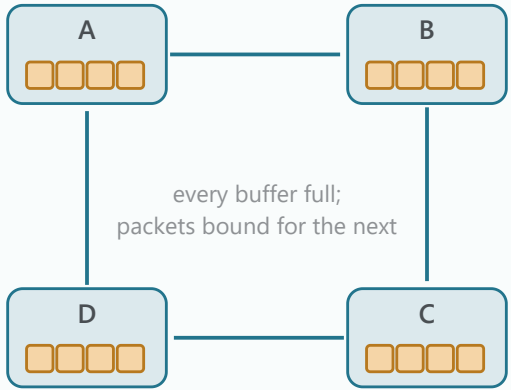
Four routers in a ring, each with limited buffer memory.

Step 1 / 3

Tanenbaum MOS §6.7.2, Figure 6-15.

Communication Deadlocks

- **Communication deadlock:** there is no resource; processes wait for *messages* from each other. A sends B a request and waits for the reply; if the request is lost, A waits for the reply and B for the request, forever.
- Resource ordering and avoidance don't help; the cure is a **timeout** and retransmission. The protocol needs sequence numbers to handle duplicate messages.
- In network hardware, though, buffers are real resources; the ring below is a **resource deadlock**.

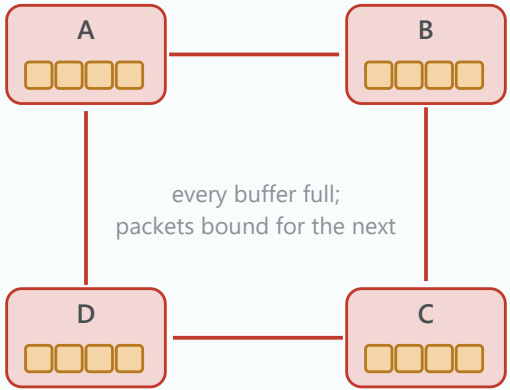


Under heavy traffic every buffer fills with packets bound for the next router clockwise.

Tanenbaum MOS §6.7.2, Figure 6-15.

Communication Deadlocks

- **Communication deadlock:** there is no resource; processes wait for *messages* from each other. A sends B a request and waits for the reply; if the request is lost, A waits for the reply and B for the request, forever.
- Resource ordering and avoidance don't help; the cure is a **timeout** and retransmission. The protocol needs sequence numbers to handle duplicate messages.
- In network hardware, though, buffers are real resources; the ring below is a **resource deadlock**.



A waits for a free buffer at B, B at C, C at D, D at A: circular wait. Remedies: dropping packets on timeout, virtual channels, credit-based flow control.

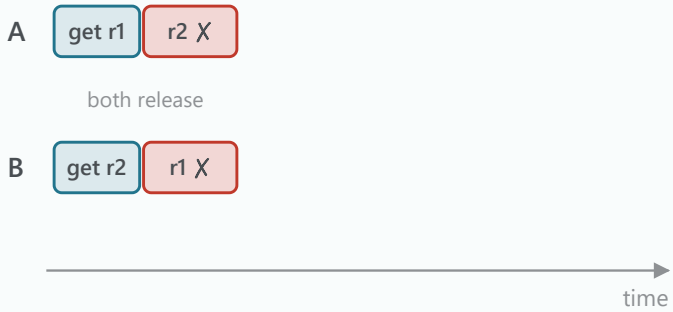
Tanenbaum MOS §6.7.2, Figure 6-15.

Livelock

- Two people meeting in a corridor both step to the same side at the same time: neither gets past.
- In deadlock, processes **sleep**; in livelock they **run** but do no useful work.

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: resource_2 first, then 1 */
```

Tanenbaum Figure 6-16: if busy-waiting `enter_region` is called in reverse order, the processes spin forever without blocking.



A gets r1, B gets r2. A tries r2 (`tryLock` fails), B tries r1 (fails).

Step 1 / 4

Livelock

- Two people meeting in a corridor both step to the same side at the same time: neither gets past.
- In deadlock, processes **sleep**; in livelock they **run** but do no useful work.

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: resource_2 first, then 1 */
```

Tanenbaum Figure 6-16: if busy-waiting `enter_region` is called in reverse order, the processes spin forever without blocking.



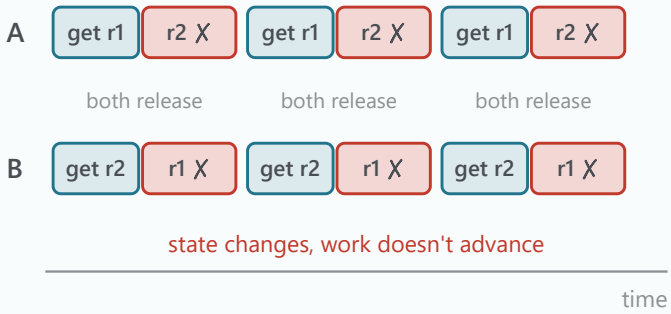
They are polite: both release what they hold and retry after *the same delay*. The same thing repeats.

Livelock

- Two people meeting in a corridor both step to the same side at the same time: neither gets past.
- In deadlock, processes **sleep**; in livelock they **run** but do no useful work.

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: resource_2 first, then 1 */
```

Tanenbaum Figure 6-16: if busy-waiting `enter_region` is called in reverse order, the processes spin forever without blocking.



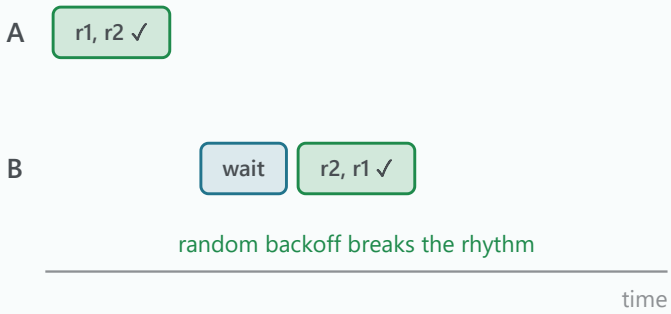
The processes are **not blocked**; they burn CPU and their state keeps changing, yet neither makes progress: **livelock**.

Livelock

- Two people meeting in a corridor both step to the same side at the same time: neither gets past.
- In deadlock, processes **sleep**; in livelock they **run** but do no useful work.

```
void process_A(void) {
  enter_region(&resource_1);
  enter_region(&resource_2);
  use_both_resources();
  leave_region(&resource_2);
  leave_region(&resource_1);
}  /* B: resource_2 first, then 1 */
```

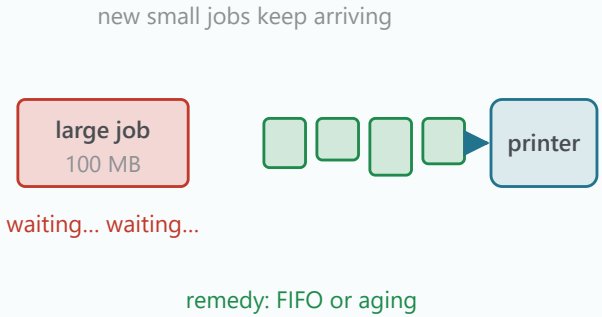
Tanenbaum Figure 6-16: if busy-waiting `enter_region` is called in reverse order, the processes spin forever without blocking.



Remedies: random (exponential) backoff, priorities/sequence numbers, or lock ordering. Ethernet's CSMA/CD backoff is the same idea.

Starvation

- A process waits indefinitely because it **never gets its turn**, even though the resource becomes available. Others make progress: the system is not deadlocked.
- It results from the allocation **policy**. Example: always giving the printer to the smallest file reduces average waiting time but can starve a large file.
- Other examples: strict priority scheduling, a reader-preferring reader–writer lock, always choosing the same victim in recovery.
- Remedies: first-come, first-served (FIFO), or raising priority the longer a process waits (**aging**).



	Deadlock	Livelock	Starvation
Process state	blocked	running	ready/blocked
Who is affected?	everyone in the cycle	the interacting group	one or a few processes

Non-Deadlock Bugs: Atomicity and Order

Deadlock isn't the most common concurrency bug: of 105 bugs studied in MySQL, Apache, Mozilla, and OpenOffice, 74 were non-deadlock, and nearly all fit two patterns (Lu et al., 2008).

Atomicity violation

Another thread slips in between two accesses that should be atomic together.

```
/* T1 */
if (sess->name != NULL)
    puts(sess->name);  /* NULL now? */
/* T2 */
sess->name = NULL;
```

Fix: put the test-and-use and T2's write under the *same* mutex.

Order violation

"A happens before B" is assumed, but nothing enforces that order.

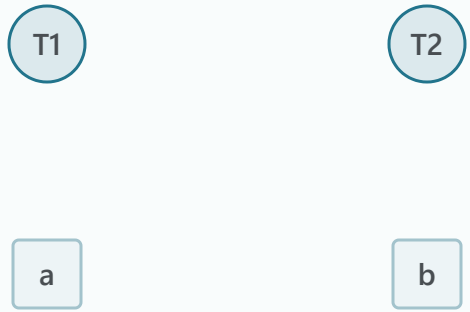
```
/* T1: init() */
worker = thread_create(body);
/* T2: body(), the new thread */
s = worker->state;  /* worker set yet? */
```

Fix: a `ready` flag and a condition variable. T2 does `while (!ready) wait`; T1 does `ready = 1; signal` after the assignment (or use a semaphore starting at 0).

A fallacy: "if every access is locked, the code is correct." Even if accesses are locked *individually*, releasing the lock between them leaves the atomicity violation; data-race tools like ThreadSanitizer may miss it.

Deadlock in Java

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
4    synchronized (a) {
5      nap(10);   /* widen the window */
6      synchronized (b) { work(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
10   synchronized (b) {
11     nap(10);
12     synchronized (a) { work(); }
13   }
14 }, "T2");
▶  t1.start(); t2.start();
```



Two threads will acquire two monitors (`synchronized` locks) in **opposite order**.

Step 1 / 5

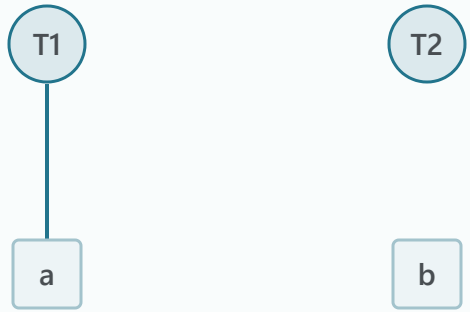
VARIABLES

`T1 = RUNNABLE` `T2 = RUNNABLE`

`nap(10)` makes the timing very likely but doesn't guarantee it. A version that used a `CountDownLatch` to wait until both first locks were held deadlocked on every run under JDK 21.

Deadlock in Java

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
▶    synchronized (a) {
▶      nap(10);   /* widen the window */
6      synchronized (b) { work(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
10   synchronized (b) {
11     nap(10);
12     synchronized (a) { work(); }
13   }
14 }, "T2");
15 t1.start(); t2.start();
```



T1 enters a's monitor and sleeps.

Step 2 / 5

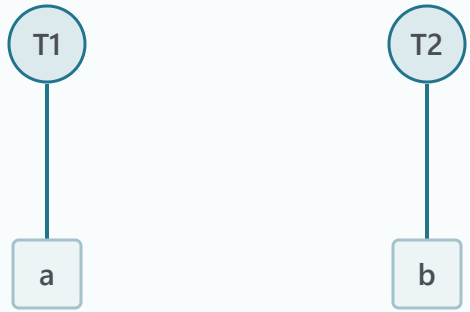
VARIABLES

`T1 = TIMED_WAITING` `T2 = RUNNABLE`

`nap(10)` makes the timing very likely but doesn't guarantee it. A version that used a `CountDownLatch` to wait until both first locks were held deadlocked on every run under JDK 21.

Deadlock in Java

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
4    synchronized (a) {
5      nap(10);   /* widen the window */
6      synchronized (b) { work(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
▶    synchronized (b) {
▶      nap(10);
12     synchronized (a) { work(); }
13   }
14 }, "T2");
15 t1.start(); t2.start();
```



Meanwhile T2 enters b's monitor.

Step 3 / 5

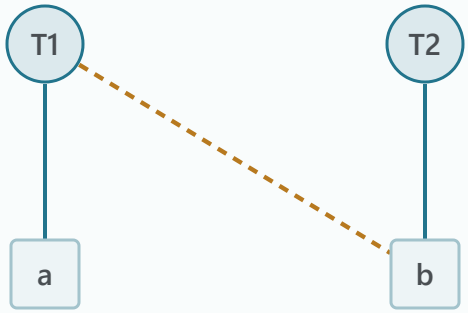
VARIABLES

`T1 = TIMED_WAITING` `T2 = TIMED_WAITING`

`nap(10)` makes the timing very likely but doesn't guarantee it. A version that used a `CountDownLatch` to wait until both first locks were held deadlocked on every run under JDK 21.

Deadlock in Java

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
4    synchronized (a) {
5      nap(10);   /* widen the window */
▶      synchronized (b) { work(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
10   synchronized (b) {
11     nap(10);
12     synchronized (a) { work(); }
13   }
14 }, "T2");
15 t1.start(); t2.start();
```



T1 wakes and requests b: b is held by T2 → T1 is `BLOCKED`.

Step 4 / 5

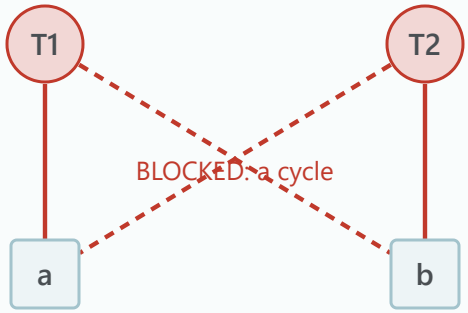
VARIABLES

`T1 = BLOCKED` `T2 = TIMED_WAITING`

`nap(10)` makes the timing very likely but doesn't guarantee it. A version that used a `CountDownLatch` to wait until both first locks were held deadlocked on every run under JDK 21.

Deadlock in Java

```
1  Object a = new Object(), b = new Object();
2
3  Thread t1 = new Thread(() -> {
4    synchronized (a) {
5      nap(10);   /* widen the window */
▶      synchronized (b) { work(); }
7    }
8  }, "T1");
9  Thread t2 = new Thread(() -> {
10   synchronized (b) {
11     nap(10);
▶      synchronized (a) { work(); }
13   }
14 }, "T2");
15 t1.start(); t2.start();
```



T2 requests a → `BLOCKED`. JVM monitors cannot be preempted: the program never ends.

Step 5 / 5

VARIABLES

`T1 = BLOCKED` `T2 = BLOCKED`

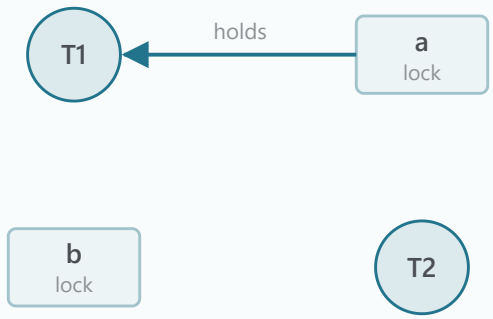
`nap(10)` makes the timing very likely but doesn't guarantee it. A version that used a `CountDownLatch` to wait until both first locks were held deadlocked on every run under JDK 21.

Encapsulation Hides Lock Order

```
var a = Collections.synchronizedList(
          new ArrayList<Integer>());
var b = Collections.synchronizedList(
          new ArrayList<Integer>());
/* inside a.addAll(b):                */
/* synchronized (a) { b.toArray() }  */
/*   → synchronized (b) { ... }      */
new Thread(() -> a.addAll(b)).start();
new Thread(() -> b.addAll(a)).start();

/* Fix: take the copy outside the lock */
Integer[] k = b.toArray(new Integer[0]);
a.addAll(Arrays.asList(k));
```

Cross call	JDK 8	JDK 21
`Vector` : `a.addAll(b)` ‖ `b.addAll(a)`	10/10	0/10
`synchronizedList` : same	10/10	10/10
Fixed version	0/10	0/10



T1: `a.addAll(b)` first takes **a**'s lock in the wrapper.

Step 1 / 3

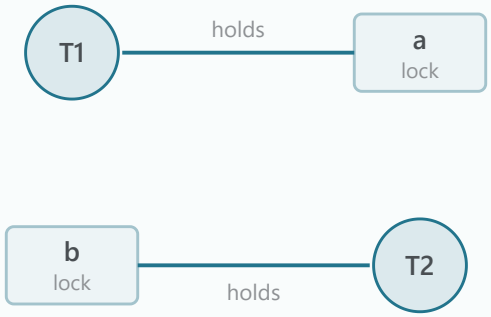
Table: deadlocks out of 10 trials, 2 threads × 200,000 rounds; results may vary between runs (e.g., 9/10). In JDK 21, `Vector.addAll` calls the argument's `toArray()` without the lock. Rule: never call alien methods while holding a lock.

Encapsulation Hides Lock Order

```
var a = Collections.synchronizedList(
          new ArrayList<Integer>());
var b = Collections.synchronizedList(
          new ArrayList<Integer>());
/* inside a.addAll(b):                */
/* synchronized (a) { b.toArray() }  */
/*   → synchronized (b) { ... }      */
new Thread(() -> a.addAll(b)).start();
new Thread(() -> b.addAll(a)).start();

/* Fix: take the copy outside the lock */
Integer[] k = b.toArray(new Integer[0]);
a.addAll(Arrays.asList(k));
```

Cross call	JDK 8	JDK 21
`Vector` : `a.addAll(b)` ‖ `b.addAll(a)`	10/10	0/10
`synchronizedList` : same	10/10	10/10
Fixed version	0/10	0/10



At the same time T2:
`b.addAll(a)` first takes **b**'s lock.

Step 2 / 3

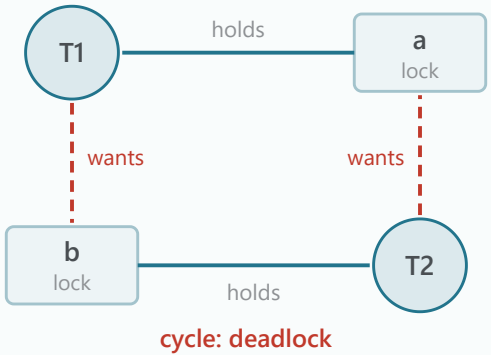
Table: deadlocks out of 10 trials, 2 threads × 200,000 rounds; results may vary between runs (e.g., 9/10). In JDK 21, `Vector.addAll` calls the argument's `toArray()` without the lock. Rule: never call alien methods while holding a lock.

Encapsulation Hides Lock Order

```
var a = Collections.synchronizedList(
          new ArrayList<Integer>());
var b = Collections.synchronizedList(
          new ArrayList<Integer>());
/* inside a.addAll(b):                */
/* synchronized (a) { b.toArray() }  */
/*   → synchronized (b) { ... }      */
new Thread(() -> a.addAll(b)).start();
new Thread(() -> b.addAll(a)).start();

/* Fix: take the copy outside the lock */
Integer[] k = b.toArray(new Integer[0]);
a.addAll(Arrays.asList(k));
```

Cross call	JDK 8	JDK 21
`Vector` : `a.addAll(b)` ‖ `b.addAll(a)`	10/10	0/10
`synchronizedList` : same	10/10	10/10
Fixed version	0/10	0/10



Inside, `b.toArray()` and `a.toArray()` are called: each wants the other's lock.

Table: deadlocks out of 10 trials, 2 threads × 200,000 rounds; results may vary between runs (e.g., 9/10). In JDK 21, `Vector.addAll` calls the argument's `toArray()` without the lock. Rule: never call alien methods while holding a lock.

Kilitlenmeyi Bulmak: jstack

```
$ jstack <pid>
Found one Java-level deadlock:
=============================
"T1":
  waiting to lock monitor 0x… (object 0x…, a java.lang.Object),
  which is held by "T2"
"T2":
  waiting to lock monitor 0x… (object 0x…, a java.lang.Object),
  which is held by "T1"

Java stack information for the threads listed above:
"T1":
        at Hang.run(Hang.java:7)
        - waiting to lock <0x…d0> (a java.lang.Object)
        - locked <0x…c0> (a java.lang.Object)
…
Found 1 deadlock.
```

Real JDK 21 output (addresses shortened). The same report: `jcmd <pid> Thread.print`, JConsole/VisualVM "Detect Deadlock".

```
ThreadMXBean mx =
    ManagementFactory.getThreadMXBean();
long[] ids = mx.findDeadlockedThreads();
if (ids != null)
  for (ThreadInfo ti : mx.getThreadInfo(ids))
    System.out.println(ti.getThreadName()
      + " waits, lock owner "
      + ti.getLockOwnerName());
```

```
T1 waits, lock owner T2
T2 waits, lock owner T1
```

- The JVM builds a **wait-for graph** from monitor and `ReentrantLock` ownership and searches it for cycles.
- It detects but **does not recover**: the process must be restarted.



Tools: lockdep and Others

Environment	Tool / behavior
Linux kernel	`lockdep` (`CONFIG_PROVE_LOCKING`): lock-order validator; hung-task detector
C/C++	ThreadSanitizer: *lock-order-inversion*; Valgrind Helgrind
Java	`jstack` , `jcmd` , `ThreadMXBean`
Go	`fatal error: all goroutines are asleep - deadlock!`
Database	PostgreSQL, InnoDB: wait-for graph + rolling back a victim transaction



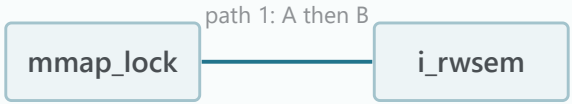
lockdep ties every lock to a **class** (all instances of the same kind).

Step 1 / 3

lockdep and TSan find *potential* deadlocks (order violations); jstack and the Go runtime report deadlocks that *have happened*.

Tools: lockdep and Others

Environment	Tool / behavior
Linux kernel	`lockdep` (`CONFIG_PROVE_LOCKING`): lock-order validator; hung-task detector
C/C++	ThreadSanitizer: *lock-order-inversion*; Valgrind Helgrind
Java	`jstack` , `jcmd` , `ThreadMXBean`
Go	`fatal error: all goroutines are asleep - deadlock!`
Database	PostgreSQL, InnoDB: wait-for graph + rolling back a victim transaction



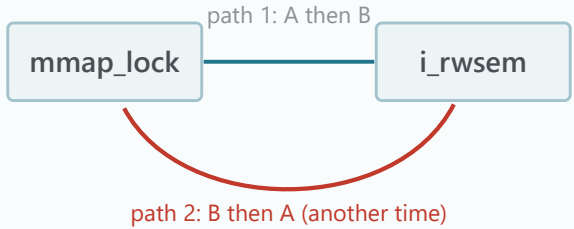
When a task takes B while holding A, the edge "A → B" is recorded.

Step 2 / 3

lockdep and TSan find *potential* deadlocks (order violations); jstack and the Go runtime report deadlocks that *have happened*.

Tools: lockdep and Others

Environment	Tool / behavior
Linux kernel	`lockdep` (`CONFIG_PROVE_LOCKING`): lock-order validator; hung-task detector
C/C++	ThreadSanitizer: *lock-order-inversion*; Valgrind Helgrind
Java	`jstack` , `jcmd` , `ThreadMXBean`
Go	`fatal error: all goroutines are asleep - deadlock!`
Database	PostgreSQL, InnoDB: wait-for graph + rolling back a victim transaction



If another path takes A while holding B, the graph gets a cycle: the deadlock is reported **without ever happening**.

lockdep and TSan find *potential* deadlocks (order violations); jstack and the Go runtime report deadlocks that *have happened*.

Summary

Approach	When?	Information needed	Cost	Used in
Ignore	—	none	an occasional reboot	general-purpose OSes
Detection + recovery	after the fact	current allocations and requests	periodic $O(m \cdot n^2)$, lost work	databases, JVM tools
Avoidance	on every request	maximum needs (Max)	deferrals, $O(m \cdot n^2)$ per request	rarely, special systems
Prevention	at design time	resource order	lost flexibility and utilization	kernel, application code

- The four conditions are necessary together; with single-instance resources, cycle ⇔ deadlock.
- An unsafe state is not a deadlock, but completion is not guaranteed.
- In practice the most effective tools: **lock ordering** + timed attempts + checking with tools (lockdep, jstack, TSan).

EXERCISES

Exercise: Classify the Bug

Classify each case as **deadlock**, **livelock**, **atomicity violation**, or **order violation**, and give the fix.

1. Two consumers on a thread-safe queue: `if (!q.isEmpty()) x = q.remove();`
2. The main thread first starts the worker, then does `buf = malloc(N)`; the worker writes `buf[0]` on its first line.
3. `transfer(x, y)` and `transfer(y, x)` run at once; each locks the source account first, then the destination.
4. Two threads do `lock(A); if (!trylock(B)) { unlock(A); usleep(10); retry; }`; the other takes B first. The delay is always 10 µs.
5. 2 CPUs and 4 threads: T1 and T2 use both L1 and L2, T3 uses only L2, T4 uses no locks. How can a lock-aware scheduler prevent deadlock?

▶ **Answers**

Review Questions

1. State the four conditions for deadlock. Which one is a state rather than a policy?
2. How can a graph have a cycle without deadlock? Give an example.
3. With E = (4 2 3 1), A = (2 1 0 0), what if P3's request were (2 1 0 1)?
4. What is the difference between an unsafe state and a deadlocked state?
5. In the Silberschatz example, what if T0 requests (0 2 0) after T1's (1 0 2) request? Why?
6. Why isn't the banker's algorithm used in general-purpose OSes?
7. Why does numbering resources make circular wait impossible?
8. How does livelock differ from deadlock? How is livelock broken?

▶ **Answers**

References

- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 (Chapter 8: Deadlocks).
- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson (Chapter 6: Deadlocks).
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org: "Common Concurrency Problems".
- E. G. Coffman, M. J. Elphick, A. Shoshani, "System Deadlocks", *ACM Computing Surveys* 3(2), 1971.
- E. W. Dijkstra, "Cooperating Sequential Processes", EWD-123, 1965 (the banker's algorithm); R. C. Holt, "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys*, 1972.
- Linux kernel documentation: `Documentation/locking/lockdep-design.rst`.
- Java SE 21: `java.lang.management.ThreadMXBean`, `java.util.concurrent.locks.ReentrantLock`; the `jstack` manual.
- PostgreSQL documentation: "Explicit Locking: Deadlocks", `deadlock_timeout`.

The End

Chapter 7: Deadlocks

Previous: Chapter 6: Synchronization · Next: Chapter 8: Memory Management