Systems and Networking I

Applied Computer Science and Artificial Intelligence

2024-2025

SAPIENZA
UNIVERSITÀ DI ROMA

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

Recap from Last Lecture

- Synchronization primitives:
  - Locks
  - Semaphores
  - Monitors

Recap from Last Lecture

- Synchronization primitives:
  - Locks
  - Semaphores
  - Monitors
- 2 fundamental synchronization problems:
  - Producers-Consumers
  - Readers-Writers

Our Journey

- What is deadlock?
- Conditions for deadlock to happen
- Deadlock detection
- Deadlock prevention
- Deadlock avoidance

Our Journey

- What is deadlock?
- Conditions for deadlock to happen
- Deadlock detection
- Deadlock prevention
- Deadlock avoidance

What is Deadlock?

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Kansas legislation early 1900's

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

Thread A

```
printer.wait();
disk.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

Thread B

```
disk.wait();
printer.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

Thread A

```
printer.wait();
disk.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

A starts first

Thread B

```
disk.wait();
printer.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

Thread A

```
printer.wait();
disk.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

Acquires printer and context switch

Thread B

```
disk.wait();
printer.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

Thread A

```
printer.wait();
disk.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

B takes over

Thread B

```
disk.wait();
printer.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

Thread A

```
printer.wait();
disk.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

Thread B

```
disk.wait();
printer.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

Acquires disk and context switch

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

Thread A

```
printer.wait();
disk.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

A executes again and blocks

Thread B

```
disk.wait();
printer.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

Thread A

```
printer.wait();
disk.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

Thread B

```
disk.wait();
printer.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

B executes again and blocks

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

Thread A

```
printer.wait();
disk.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

Thread B

```
disk.wait();
printer.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

A waits B to release the disk

What is Deadlock?

Intuitively, a condition where two or more threads are waiting for an event that can only be generated by the very same threads

Thread A

```
printer.wait();
disk.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

Thread B

```
disk.wait();
printer.wait();

// copy from disk to printer

printer.signal();
disk.signal();
```

B waits A to release the printer

Deadlock: Terminology

- Deadlock: it can occur when multiple threads compete for a finite number of resources

Deadlock: Terminology

- Deadlock: it can occur when multiple threads compete for a finite number of resources
- Deadlock detection: finds instances of deadlocks and tries to recover

Deadlock: Terminology

- Deadlock: it can occur when multiple threads compete for a finite number of resources
- Deadlock detection: finds instances of deadlocks and tries to recover
- Deadlock prevention (offline): imposes restrictions/rules on how to write deadlock-free programs

Deadlock: Terminology

- Deadlock: it can occur when multiple threads compete for a finite number of resources
- Deadlock detection: finds instances of deadlocks and tries to recover
- Deadlock prevention (offline): imposes restrictions/rules on how to write deadlock-free programs
- Deadlock avoidance (online): runtime support checks resource requests made by threads to avoid deadlocks

Deadlock vs. Starvation

- Not to be confused with each other!

Deadlock vs. Starvation

- Not to be confused with each other!
- Related terms but each one refers to a specific situation

Deadlock vs. Starvation

- Not to be confused with each other!
- Related terms but each one refers to a specific situation
- Starvation occurs when a thread waits indefinitely for some resource but other threads are actually making progress using that resource

Deadlock vs. Starvation

- Not to be confused with each other!
- Related terms but each one refers to a specific situation
- Starvation occurs when a thread waits indefinitely for some resource but other threads are actually making progress using that resource
- The main difference with deadlock is that the system is not completely stuck!

Our Journey

- What is deadlock?
- Conditions for deadlock to happen
- Deadlock detection
- Deadlock prevention
- Deadlock avoidance

Necessary Conditions for Deadlock

- Deadlock can happen if all the 4 conditions below hold

Necessary Conditions for Deadlock

- Deadlock can happen if all the 4 conditions below hold
  - Mutual Exclusion → at least one thread must hold a non-sharable resource (only one thread holds the resource)

Necessary Conditions for Deadlock

- Deadlock can happen if all the 4 conditions below hold
  - Mutual Exclusion → at least one thread must hold a non-sharable resource (only one thread holds the resource)
  - Hold and Wait → at least one thread is holding a non-sharable resource and is waiting for other resource(s) to become available (another thread holds the resource(s))

Necessary Conditions for Deadlock

- Deadlock can happen if all the 4 conditions below hold
  - Mutual Exclusion → at least one thread must hold a non-sharable resource (only one thread holds the resource)
  - Hold and Wait → at least one thread is holding a non-sharable resource and is waiting for other resource(s) to become available (another thread holds the resource(s))
  - No Preemption → a thread can only release a resource voluntarily; neither another thread nor the OS can force it to release the resource

Necessary Conditions for Deadlock

- Deadlock can happen if all the 4 conditions below hold
  - Mutual Exclusion → at least one thread must hold a non-sharable resource (only one thread holds the resource)
  - Hold and Wait → at least one thread is holding a non-sharable resource and is waiting for other resource(s) to become available (another thread holds the resource(s))
  - No Preemption → a thread can only release a resource voluntarily; neither another thread nor the OS can force it to release the resource
  - Circular Wait → a set of waiting threads $t_1, \ldots, t_n$ where t_i is waiting on $t_{(i+1)\%n}$

Our Journey

- What is deadlock?
- Conditions for deadlock to happen
- Deadlock detection
- Deadlock prevention
- Deadlock avoidance

Deadlock Detection: Resource Allocation Graph

- We define a directed graph G=(V, E) where:

Deadlock Detection: Resource Allocation Graph

- We define a directed graph G=(V, E) where:
  - V is the set of vertices representing both resources $\{r_1, \ldots, r_m\}$ and threads $\{t_1, \ldots, t_n\}$

Deadlock Detection: Resource Allocation Graph

- We define a directed graph G=(V, E) where:
  - V is the set of vertices representing both resources $\{r_1, \ldots, r_m\}$ and threads $\{t_1, \ldots, t_n\}$
  - E is the set of edges between resources and threads

Deadlock Detection: Resource Allocation Graph

- We define a directed graph G=(V, E) where:
  - V is the set of vertices representing both resources $\{r_1, \ldots, r_m\}$ and threads $\{t_1, \ldots, t_n\}$
  - E is the set of edges between resources and threads
- Edges can be of 2 types:

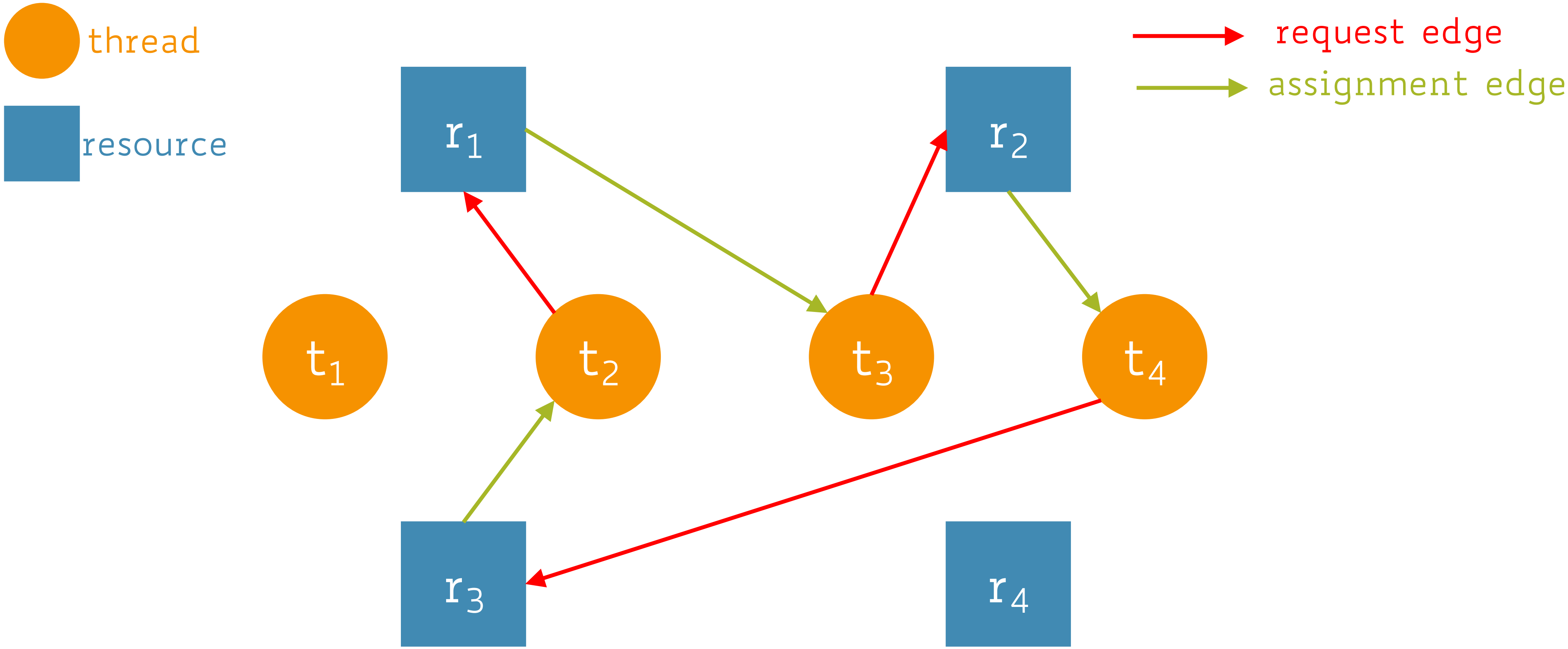
Deadlock Detection: Resource Allocation Graph

- We define a directed graph G=(V, E) where:
  - V is the set of vertices representing both resources $\{r_1, \ldots, r_m\}$ and threads $\{t_1, \ldots, t_n\}$
  - E is the set of edges between resources and threads
- Edges can be of 2 types:
  - Request Edge → a directed edge (t_i, r_j) indicates that t_i has requested r_j, but not yet acquired

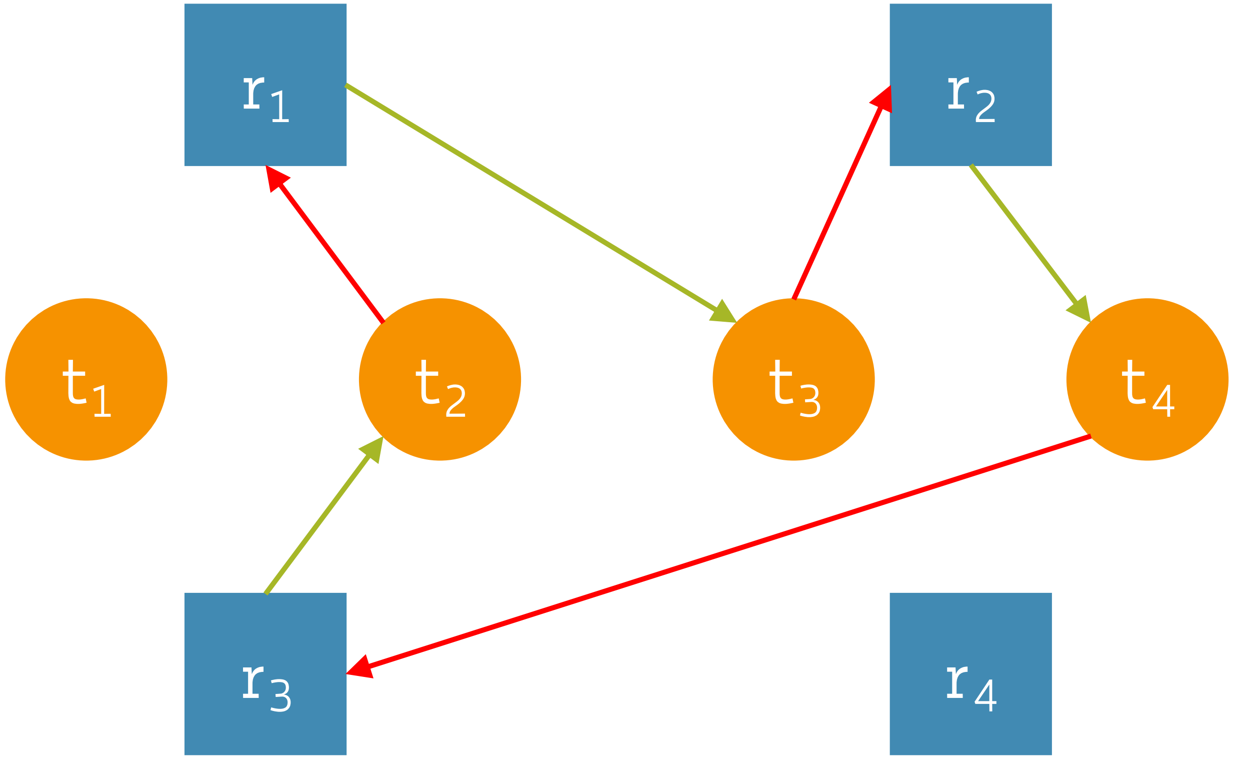
Deadlock Detection: Resource Allocation Graph

- We define a directed graph G=(V, E) where:
  - V is the set of vertices representing both resources $\{r_1, \ldots, r_m\}$ and threads $\{t_1, \ldots, t_n\}$
  - E is the set of edges between resources and threads
- Edges can be of 2 types:
  - Request Edge → a directed edge (t_i, r_j) indicates that t_i has requested r_j, but not yet acquired
  - Assignment Edge → a directed edge (r_j, t_i) indicates that the OS has allocated r_j to t_i

Deadlock Detection: Resource Allocation Graph

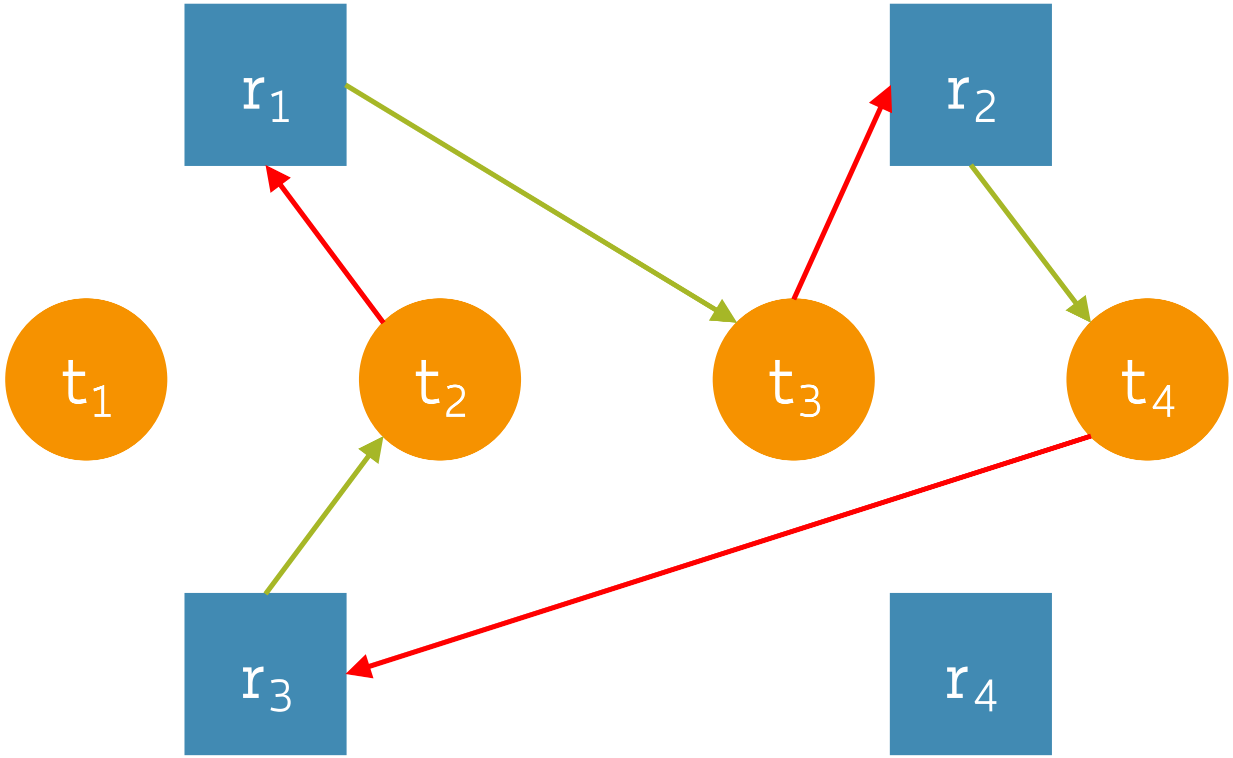


Deadlock Detection: Resource Allocation Graph



If the graph has no cycles, no deadlock will ever exist

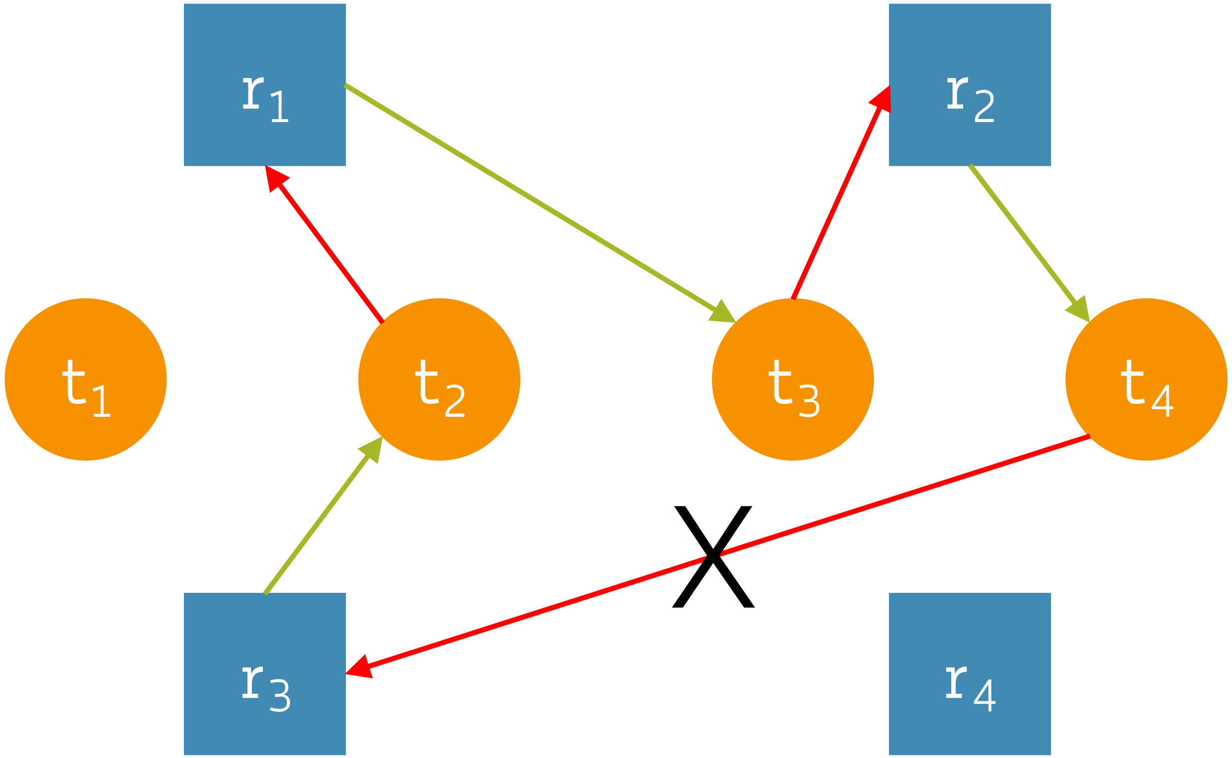
Deadlock Detection: Resource Allocation Graph



If the graph has no cycles, no deadlock will ever exist

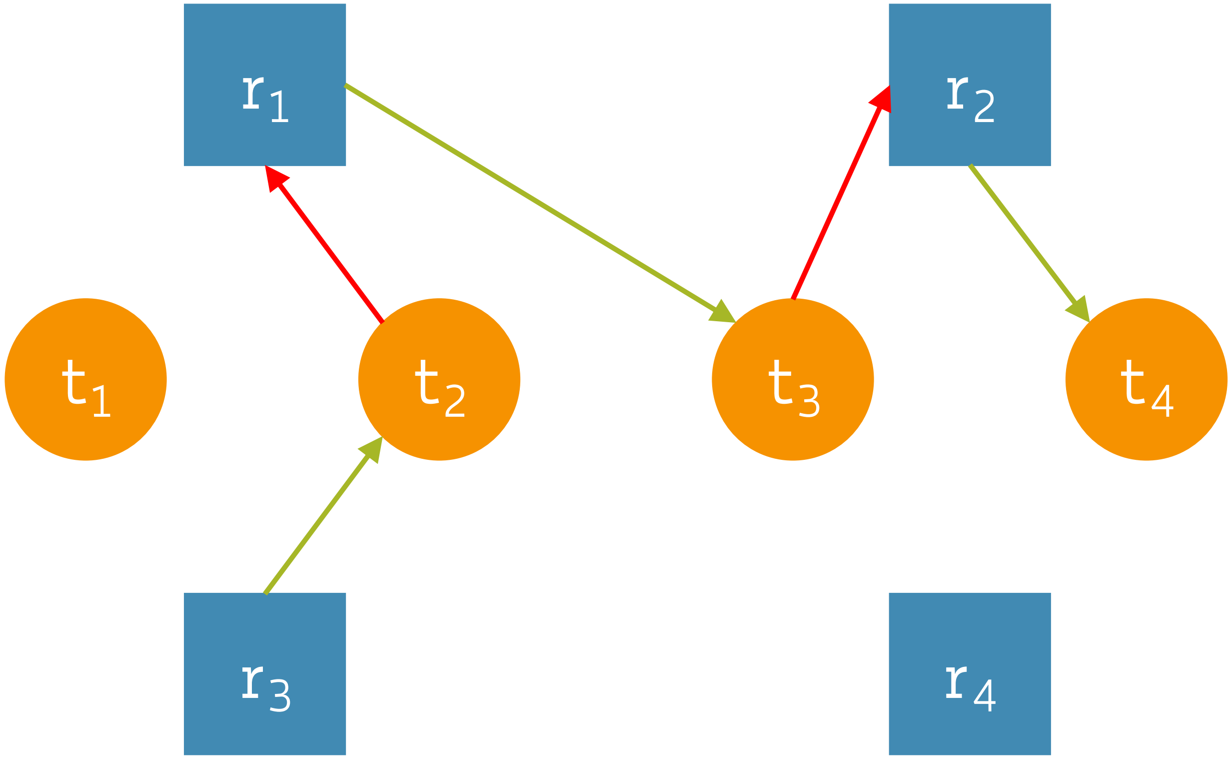
Why?

Deadlock Detection: Resource Allocation Graph



Suppose we remove the edge (t_4, r_3) so as to remove the cycle

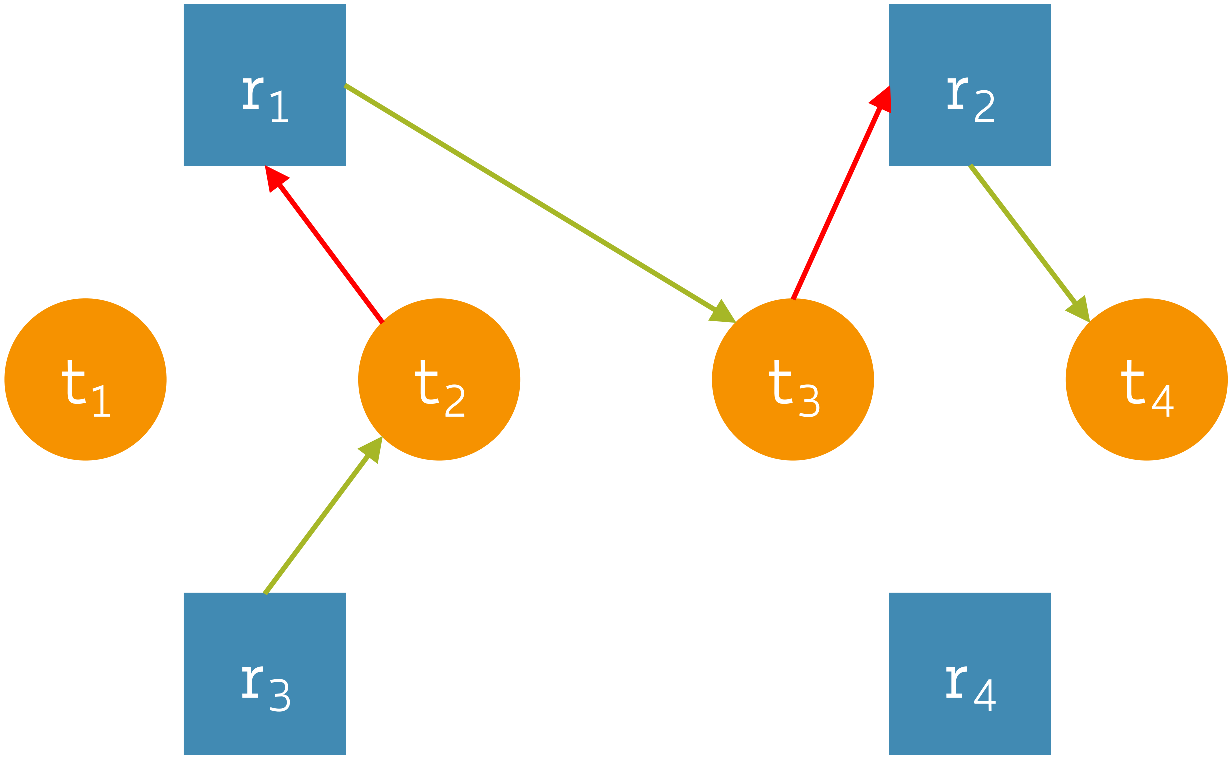
Deadlock Detection: Resource Allocation Graph



Suppose we remove the edge (t_4, r_3) so as to remove the cycle

No deadlock can occur as t_4 is not waiting on anything…

Deadlock Detection: Resource Allocation Graph

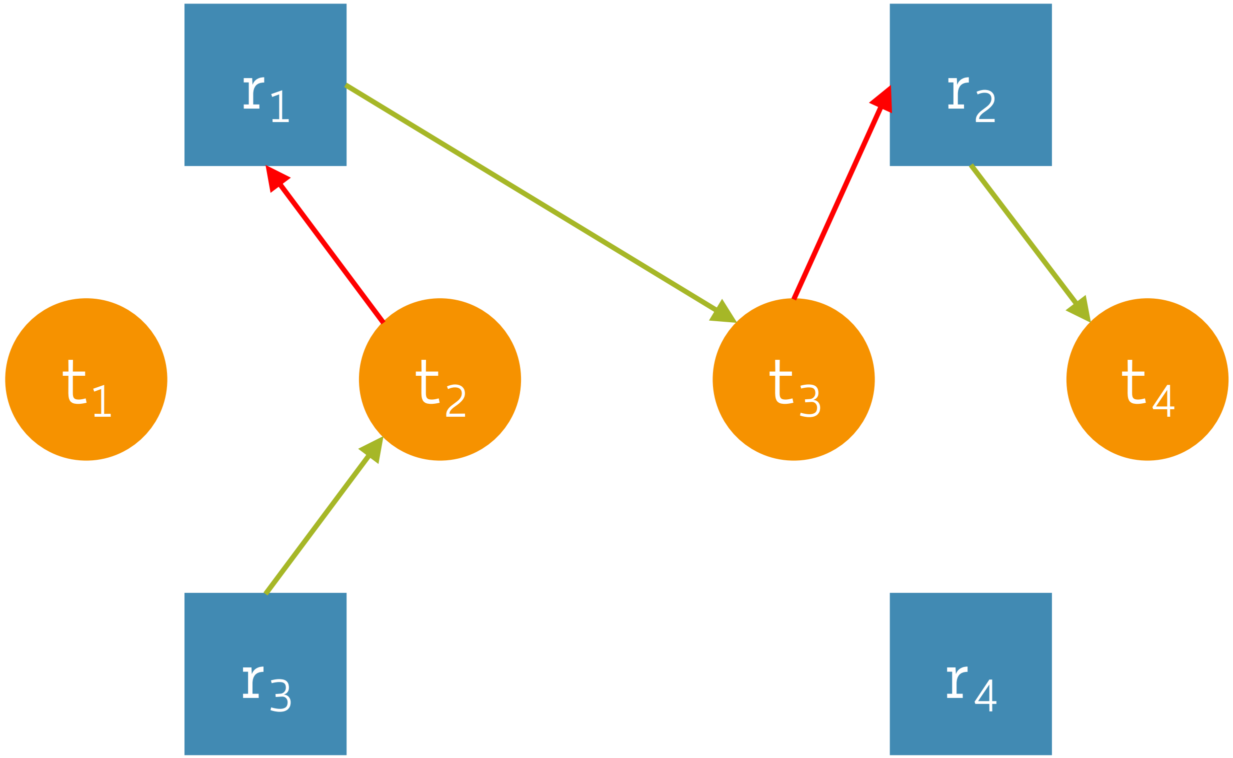


Suppose we remove the edge (t_4, r_3) so as to remove the cycle

No deadlock can occur as t_4 is not waiting on anything…

Therefore, t_4 can run and eventually will release r_2, which wakes up t_3

Deadlock Detection: Resource Allocation Graph



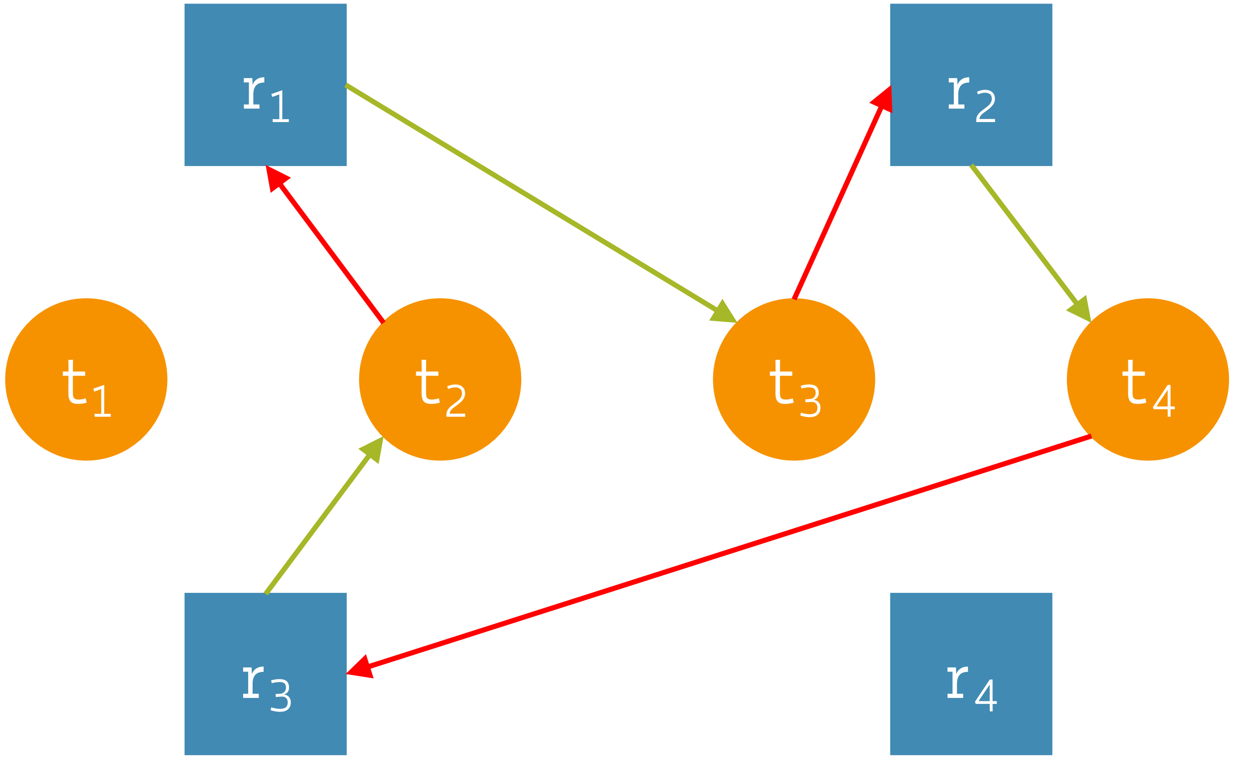
Suppose we remove the edge (t_4, r_3) so as to remove the cycle

No deadlock can occur as t_4 is not waiting on anything…

Therefore, t_4 can run and eventually will release r_2, which wakes up t_3

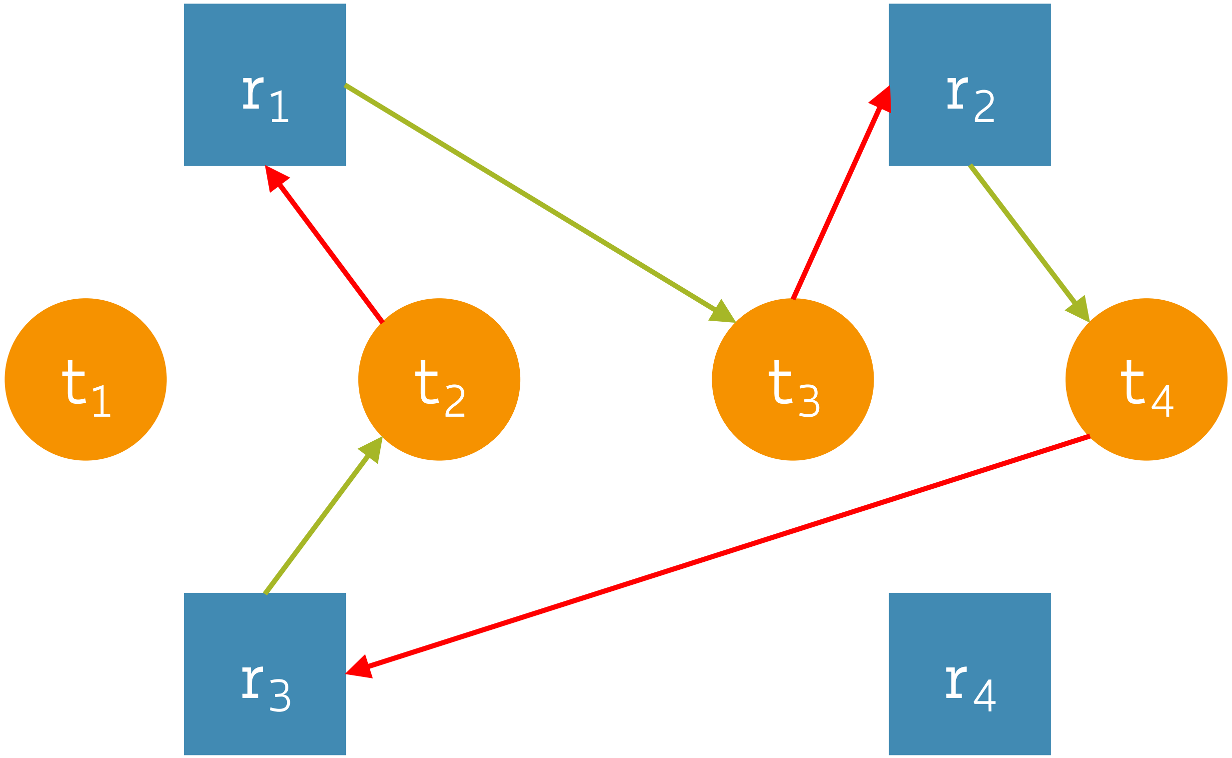
And so on and so forth…

Deadlock Detection: Resource Allocation Graph



If the graph has cycles, deadlock might exist

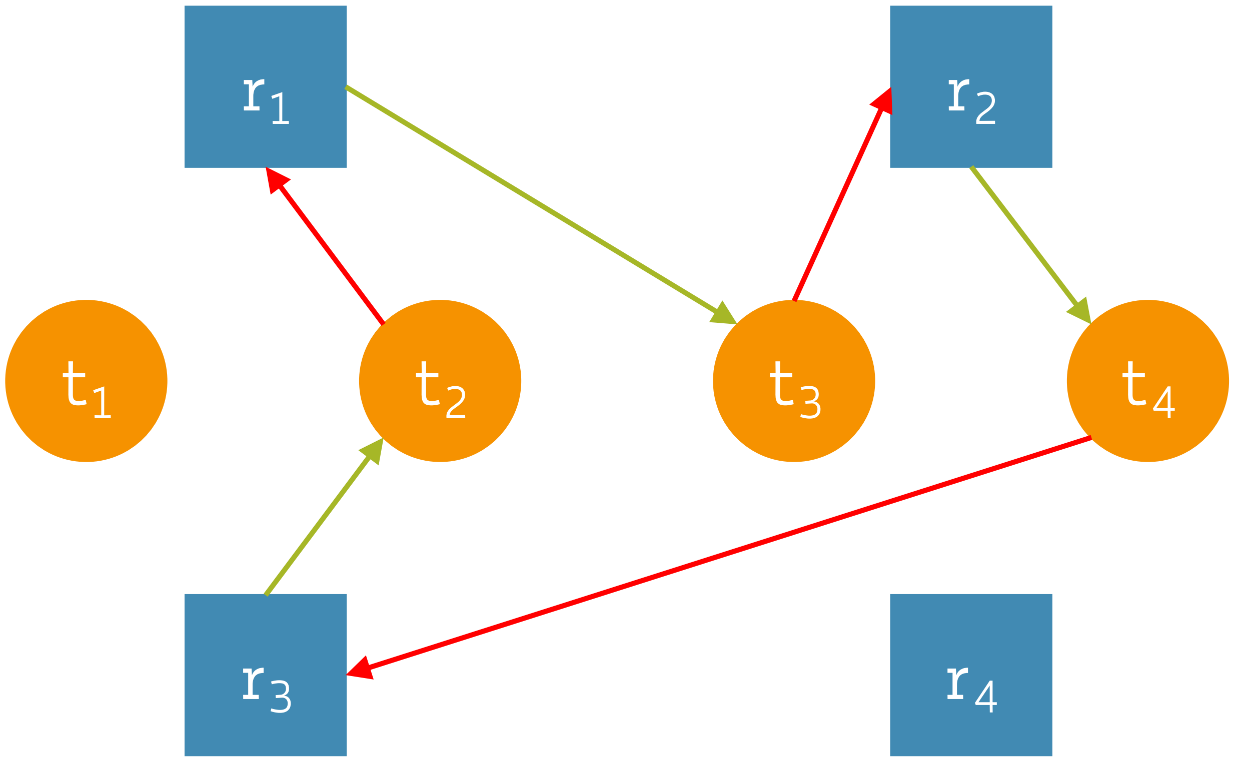
Deadlock Detection: Resource Allocation Graph



If the graph has cycles, deadlock might exist

Why?

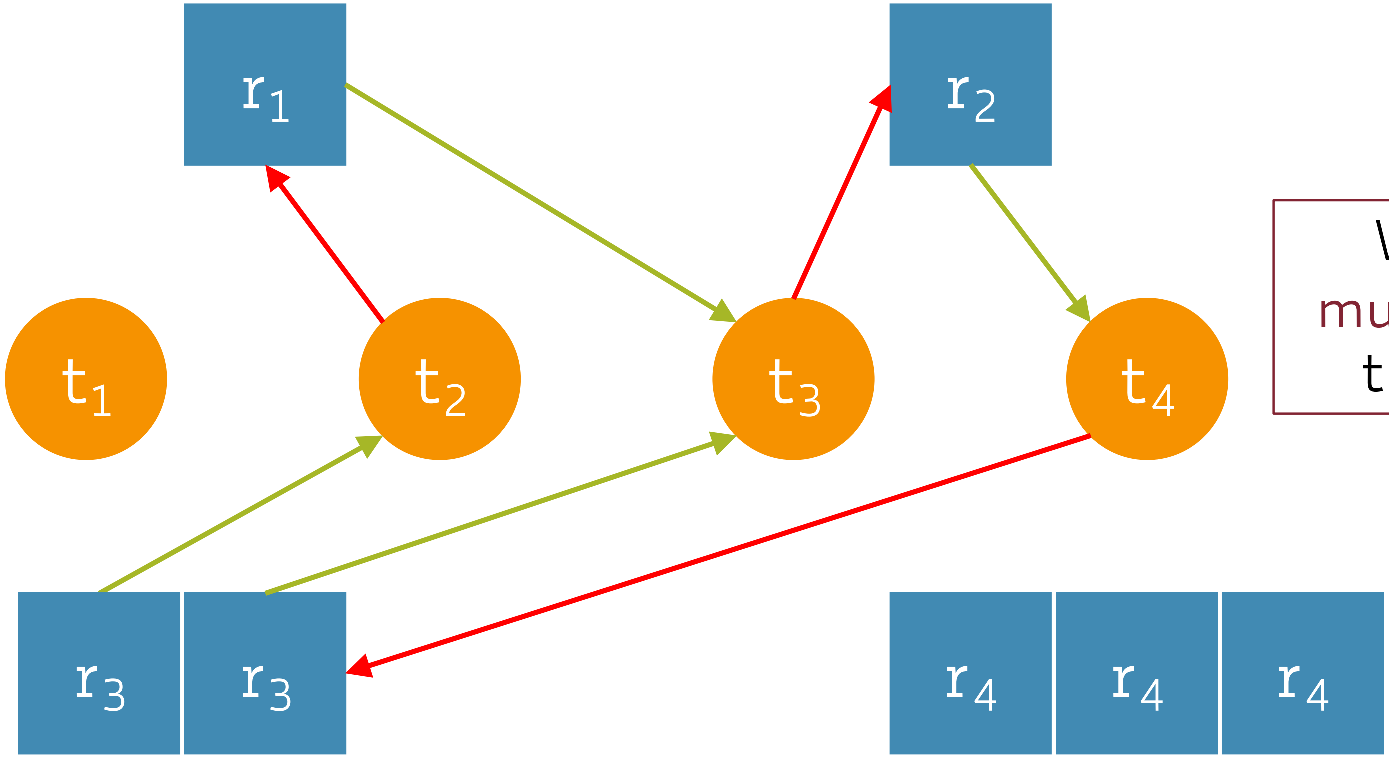
Deadlock Detection: Resource Allocation Graph



If the graph has cycles, deadlock might exist

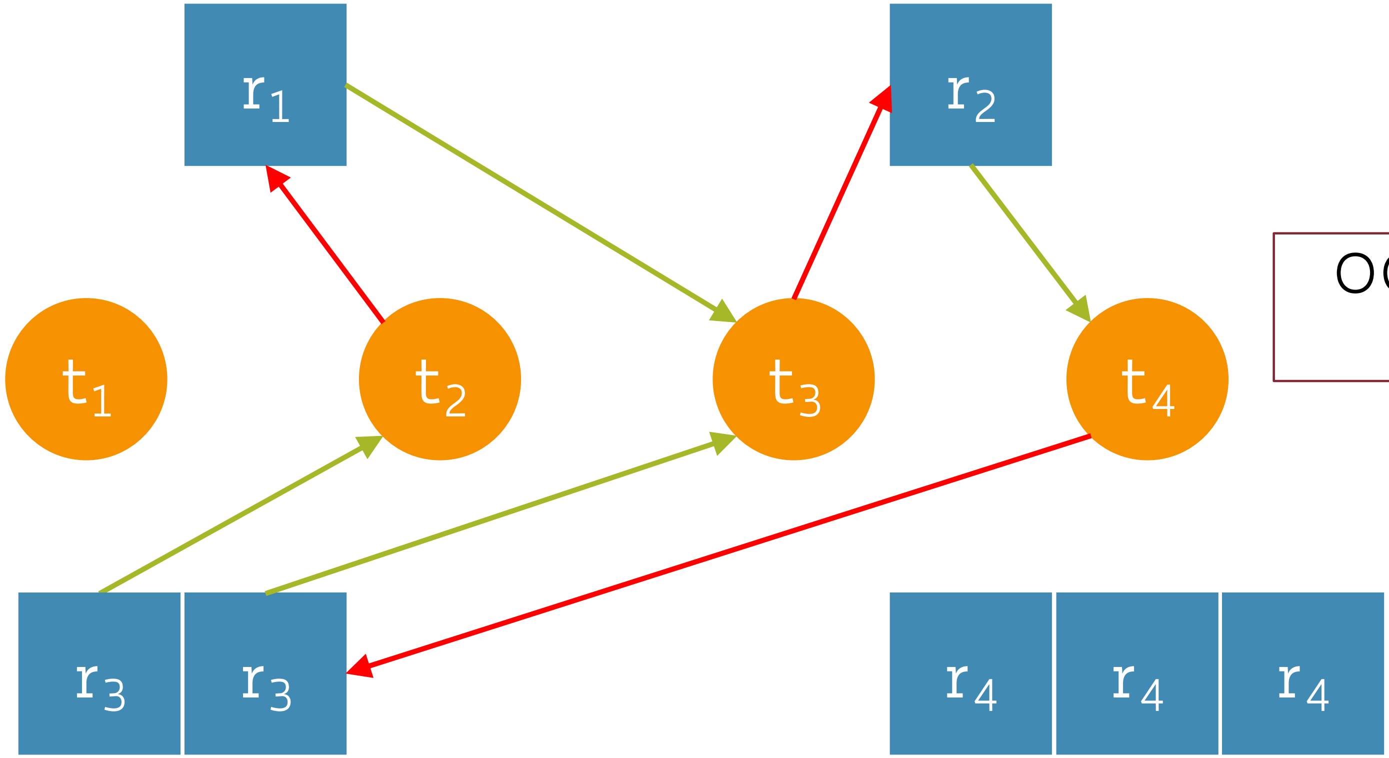
We are assuming the multiplicity of each resource is 1 (i.e., one r_1, one r_2, etc.)

Deadlock Detection: Resource Allocation Graph



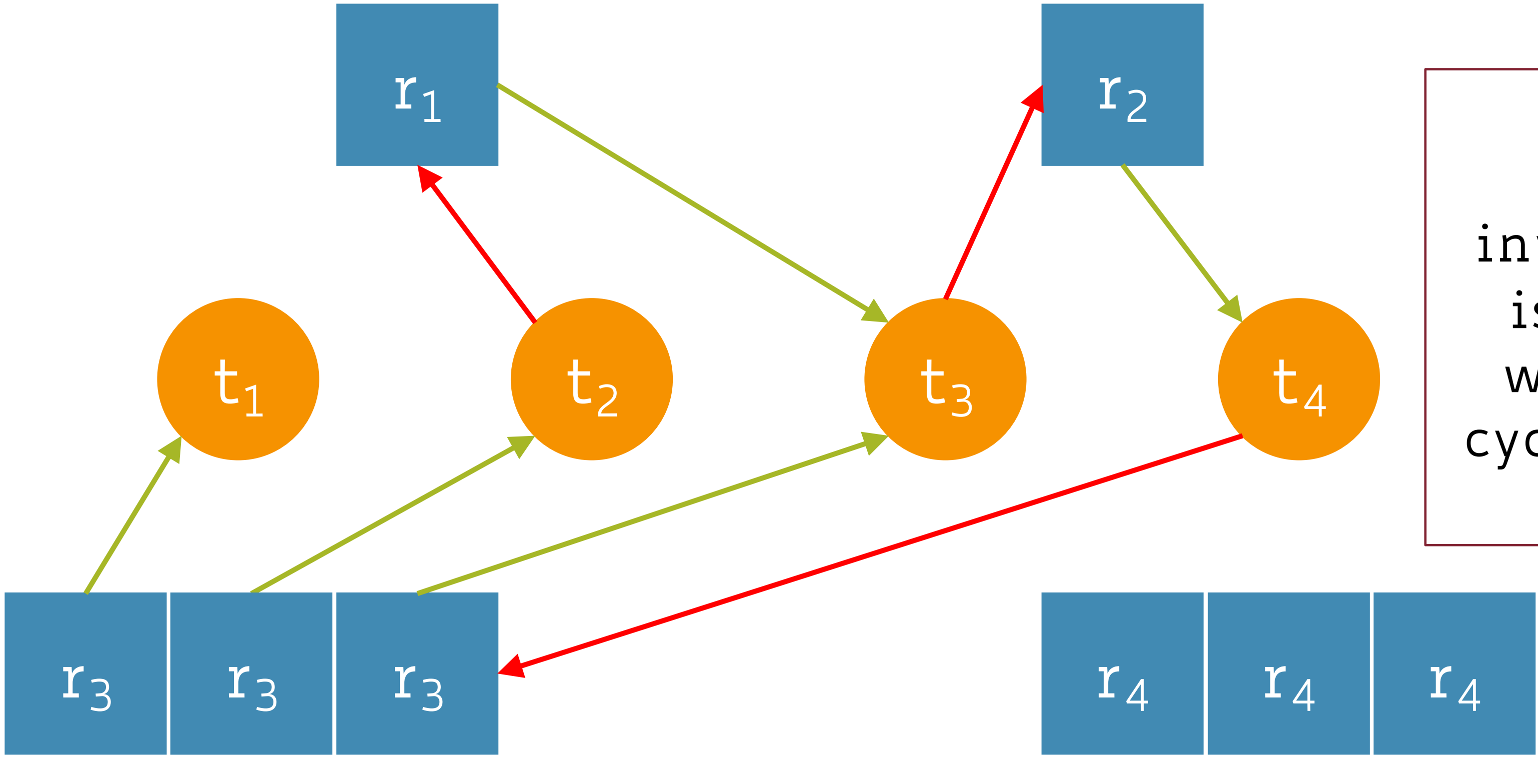
What if there are multiple instances of the same resource?

Deadlock Detection: Resource Allocation Graph



OOPS! This is still a deadlock

Deadlock Detection: Resource Allocation Graph



This works!
If *any* resource involved in the cycle is held by a thread which is not in the cycle (t_1) then we can make progress

Deadlock: Detect and Correct It!

- Scan the Resource Allocation Graph (RAG) for cycles, and then break those!

Deadlock: Detect and Correct It!

- Scan the Resource Allocation Graph (RAG) for cycles, and then break those!
- How? Several ways of doing it:

Deadlock: Detect and Correct It!

- Scan the Resource Allocation Graph (RAG) for cycles, and then break those!
- How? Several ways of doing it:
  - Kill all the threads in the cycle (quite harsh, ugh?)

Deadlock: Detect and Correct It!

- Scan the Resource Allocation Graph (RAG) for cycles, and then break those!
- How? Several ways of doing it:
  - Kill all the threads in the cycle (quite harsh, ugh?)
  - Kill all the threads one at a time, forcing each one of them to release resource(s)

Deadlock: Detect and Correct It!

- Scan the Resource Allocation Graph (RAG) for cycles, and then break those!
- How? Several ways of doing it:
  - Kill all the threads in the cycle (quite harsh, ugh?)
  - Kill all the threads one at a time, forcing each one of them to release resource(s)
  - Preempt resources one at a time rolling back to a consistent status (e.g., common in database transactions)

Deadlock: Detect and Correct It!

- Scan the Resource Allocation Graph (RAG) for cycles, and then break those!
- How? Several ways of doing it:
  - Kill all the threads in the cycle (quite harsh, ugh?)
  - Kill all the threads one at a time, forcing each one of them to release resource(s)
  - Preempt resources one at a time rolling back to a consistent status (e.g., common in database transactions)
- We would like to be more precise than that…

Deadlock: Detect and Correct It!

- Detecting cycles on a directed graph $G=(V, E)$ is a quite costly operation

Deadlock: Detect and Correct It!

- Detecting cycles on a directed graph $G=(V, E)$ is a quite costly operation
- Known algorithms based on depth-first search (DFS) take $O(|V|+|E|)$ time

Deadlock: Detect and Correct It!

- Detecting cycles on a directed graph $G=(V, E)$ is a quite costly operation
- Known algorithms based on depth-first search (DFS) take $O(|V|+|E|)$ time
- $O(|V|+|E|) \sim O(|V|^2)$ as $|E| = O(|V|^2)$ for dense graphs, and $|V|$ = #threads + #resources

Deadlock: Detect and Correct It!

- When to run such a detection algorithm?

Deadlock: Detect and Correct It!

- When to run such a detection algorithm?
  - Before granting a resource → each granted request will take $O(|V|^2)$

Deadlock: Detect and Correct It!

- When to run such a detection algorithm?
  - Before granting a resource → each granted request will take $O(|V|^2)$
  - When a request cannot be fulfilled → each failed request will take $O(|V|^2)$

Deadlock: Detect and Correct It!

- When to run such a detection algorithm?
  - Before granting a resource → each granted request will take $O(|V|^2)$
  - When a request cannot be fulfilled → each failed request will take $O(|V|^2)$
  - On a regular schedule or when the CPU is under-utilized

Our Journey

- What is deadlock?
- Conditions for deadlock to happen
- Deadlock detection
- Deadlock prevention
- Deadlock avoidance

Deadlock Prevention

- Ensure that at least one of the 4 necessary conditions doesn't hold
  - Mutual Exclusion → make all resources sharable
    - Not all can be shared
    - E.g., disks, printers, etc.

Deadlock Prevention

- Ensure that at least one of the 4 necessary conditions doesn't hold
  - Hold and Wait → a thread cannot hold one resource when it requests another
    - Enforce requests to be made all at once
    - Hard to predict all the resources a thread will need

Deadlock Prevention

- Ensure that at least one of the 4 necessary conditions doesn't hold
  - No Preemption → if a thread requests a resource that cannot be allocated to it, the OS preempts (releases) all the resources that the thread is already holding
    - Not all resources can be easily preempted (e.g., printers)

Deadlock Prevention

- Ensure that at least one of the 4 necessary conditions doesn't hold
  - Circular Wait → impose an ordering (i.e., numbering) on resources and enforce to request them in such order
    - Hard to establish such an order

Our Journey

- What is deadlock?
- Conditions for deadlock to happen
- Deadlock detection
- Deadlock prevention
- Deadlock avoidance

Deadlock Avoidance: Resource Reservation

Each thread provides information about the maximum number of resources it might need during execution

Deadlock Avoidance: Resource Reservation

Each thread provides information about the maximum number of resources it might need during execution

$$m_i = \textit{maximum} \text{ number of resources that thread } i \textit{ might} \text{ request}$$

$$c_i = \textit{current} \text{ number of resources that thread } i \text{ is holding}$$

$$C = \sum_{i=1}^{n} c_i = \textit{total} \text{ number of resources currently allocated}$$

$$R = \textit{maximum} \text{ number of resources overall available}$$

Deadlock Avoidance: Resource Reservation

Each thread provides information about the maximum number of resources it might need during execution

$$m_i = \textit{maximum} \text{ number of resources that thread } i \textit{ might} \text{ request}$$
$$c_i = \textit{current} \text{ number of resources that thread } i \text{ is holding}$$
$$C = \sum_{i=1}^{n} c_i = \textit{total} \text{ number of resources currently allocated}$$
$$R = \textit{maximum} \text{ number of resources overall available}$$

Any thread sequence is safe if for each thread it holds that:

$$\underbrace{m_i - c_i}_{\text{resources } t_i \text{ might still request}} \leq \underbrace{R - C}_{\text{resources currently available}} + \underbrace{\sum_{j=1}^{i-1} c_j}_{\text{resources currently allocated up to } t_j, j<i}$$

Deadlock Avoidance: Safe State

- A state in which there is a safe sequence for the threads

Deadlock Avoidance: Safe State

- A state in which there is a safe sequence for the threads
- An unsafe state does not necessarily mean deadlock (i.e., some threads may not request the maximum number of resources as declared)

Deadlock Avoidance: Safe State

- A state in which there is a safe sequence for the threads
- An unsafe state does not necessarily mean deadlock (i.e., some threads may not request the maximum number of resources as declared)
- Grant a resource to a thread if the new state is safe, otherwise make it wait even if the resource is available

Deadlock Avoidance: Safe State

- A state in which there is a safe sequence for the threads
- An unsafe state does not necessarily mean deadlock (i.e., some threads may not request the maximum number of resources as declared)
- Grant a resource to a thread if the new state is safe, otherwise make it wait even if the resource is available
- This policy ensures no circular-wait condition exists

Deadlock Avoidance: Example

- 3 threads: t_1, t_2, and t_3 are competing for 12 tape drives (resources)

Deadlock Avoidance: Example

- 3 threads: t_1, t_2, and t_3 are competing for 12 tape drives (resources)
- Currently, 11 drives are allocated to the threads, leaving 1 available

Deadlock Avoidance: Example

- 3 threads: t_1, t_2, and t_3 are competing for 12 tape drives (resources)
- Currently, 11 drives are allocated to the threads, leaving 1 available

Thread	m_i	c_i	$m_i - c_i$
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

Deadlock Avoidance: Example

- 3 threads: t_1, t_2, and t_3 are competing for 12 tape drives (resources)
- Currently, 11 drives are allocated to the threads, leaving 1 available

Thread	m_i	c_i	$m_i - c_i$
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

Is the current state safe?

Deadlock Avoidance: Example

Thread	m_i	c_i	$m_i - c_i$
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

The current state is safe: there exists a sequence of threads (t_1, t_2, t_3) where each one gets the maximum number of resources without waiting

Deadlock Avoidance: Example

Thread	m_i	c_i	$m_i - c_i$
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

The current state is safe: there exists a sequence of threads (t_1, t_2, t_3) where each one gets the maximum number of resources without waiting

t_1 can complete using the current allocation and the 1 drive left

Deadlock Avoidance: Example

Thread	m_i	c_i	$m_i - c_i$
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

The current state is safe: there exists a sequence of threads (t_1, t_2, t_3) where each one gets the maximum number of resources without waiting

t_1 can complete using the current allocation and the 1 drive left

t_2 can use the current allocation, plus t_1's resources and 1 drive left (4 drives)

Deadlock Avoidance: Example

Thread	m_i	c_i	$m_i - c_i$
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

The current state is safe: there exists a sequence of threads (t_1, t_2, t_3) where each one gets the maximum number of resources without waiting

t_1 can complete using the current allocation and the 1 drive left

t_2 can use the current allocation, plus t_1's resources and 1 drive left (4 drives)

t_3 can use the current allocation, plus t_1's & t_2's resources and 1 drive left (8 drives)

Deadlock Avoidance: Example

Thread	m_i	c_i	$m_i - c_i$
t_1	4	3	1
t_2	8	4	4
t_3	12	5	7

Suppose t_3 requests one more drive, then now there are no more available drives

Deadlock Avoidance: Example

Thread	m_i	c_i	$m_i - c_i$
t_1	4	3	1
t_2	8	4	4
t_3	12	5	7

Suppose t_3 requests one more drive, then now there are no more available drives

Theoretically, everything might still work (e.g., t_1 may never request another drive)

Deadlock Avoidance: Example

Thread	m_i	c_i	$m_i - c_i$
t_1	4	3	1
t_2	8	4	4
t_3	12	5	7

Suppose t_3 requests one more drive, then now there are no more available drives

Theoretically, everything might still work (e.g., t_1 may never request another drive)

However, t_3 must wait because allocating that extra drive would lead to an unsafe state, which in turn might lead to deadlock

Deadlock Avoidance: Resource Allocation Graph

- An extension of the original resource allocation graph

Deadlock Avoidance: Resource Allocation Graph

- An extension of the original resource allocation graph
- Edges can now be of 3 types:
  - Request Edge → a directed edge (t_i, r_j) indicates that t_i has requested r_j, but not yet acquired
  - Claim (dotted) Edge → a directed edge (t_i, r_j) indicates that t_i might request r_j in the future
  - Assignment Edge → a directed edge (r_j, t_i) indicates that the OS has allocated r_j to t_i

Deadlock Avoidance: Resource Allocation Graph

- An extension of the original resource allocation graph
- Edges can now be of 3 types:
  - Request Edge → a directed edge (t_i, r_j) indicates that t_i has requested r_j, but not yet acquired
  - Claim (dotted) Edge → a directed edge (t_i, r_j) indicates that t_i might request r_j in the future
  - Assignment Edge → a directed edge (r_j, t_i) indicates that the OS has allocated r_j to t_i
- Satisfying a request means converting a claim into an assignment edge

Deadlock Avoidance: Resource Allocation Graph

- A cycle in this extended RAG indicates an unsafe state

Deadlock Avoidance: Resource Allocation Graph

- A cycle in this extended RAG indicates an unsafe state
- If the allocation results in an unsafe state, this will be denied even if the resource is actually available

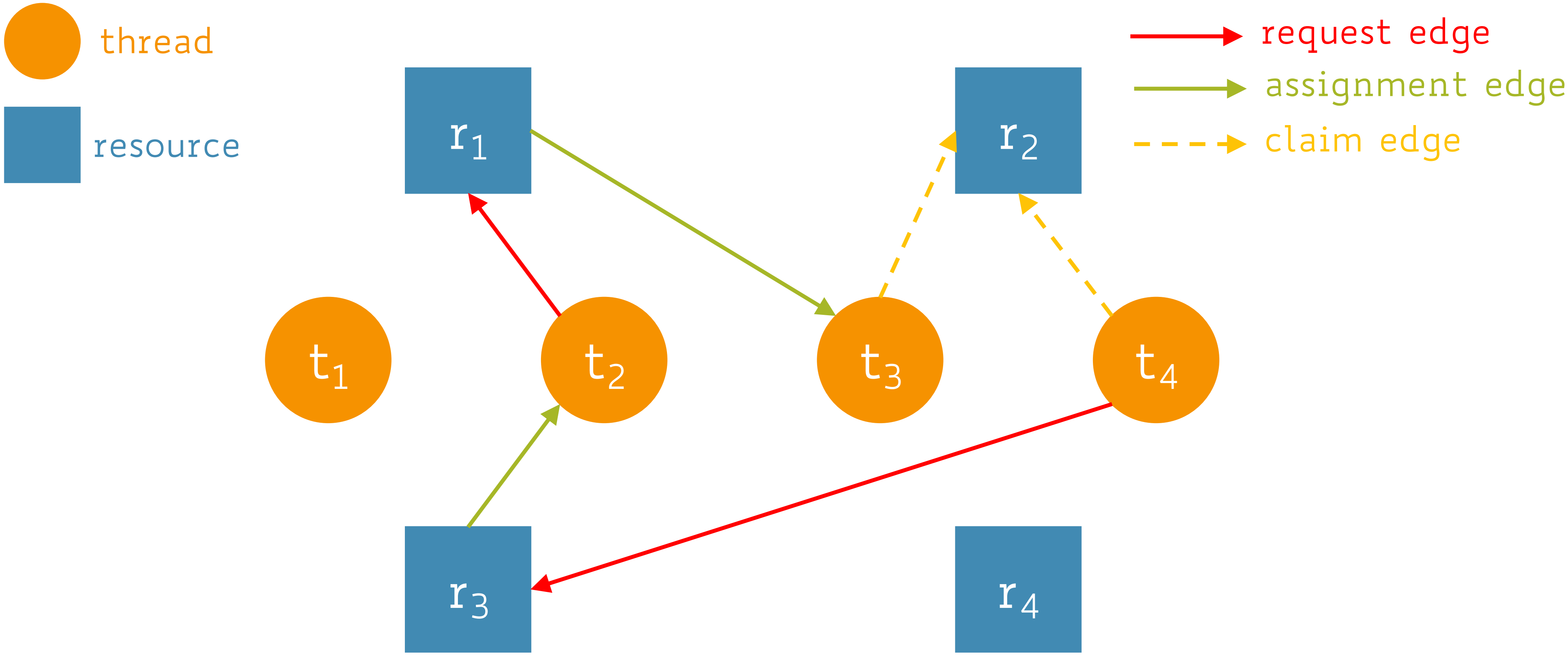
Deadlock Avoidance: Resource Allocation Graph

- A cycle in this extended RAG indicates an unsafe state
- If the allocation results in an unsafe state, this will be denied even if the resource is actually available
- In other words, the claim edge is converted into a request edge and the thread will wait

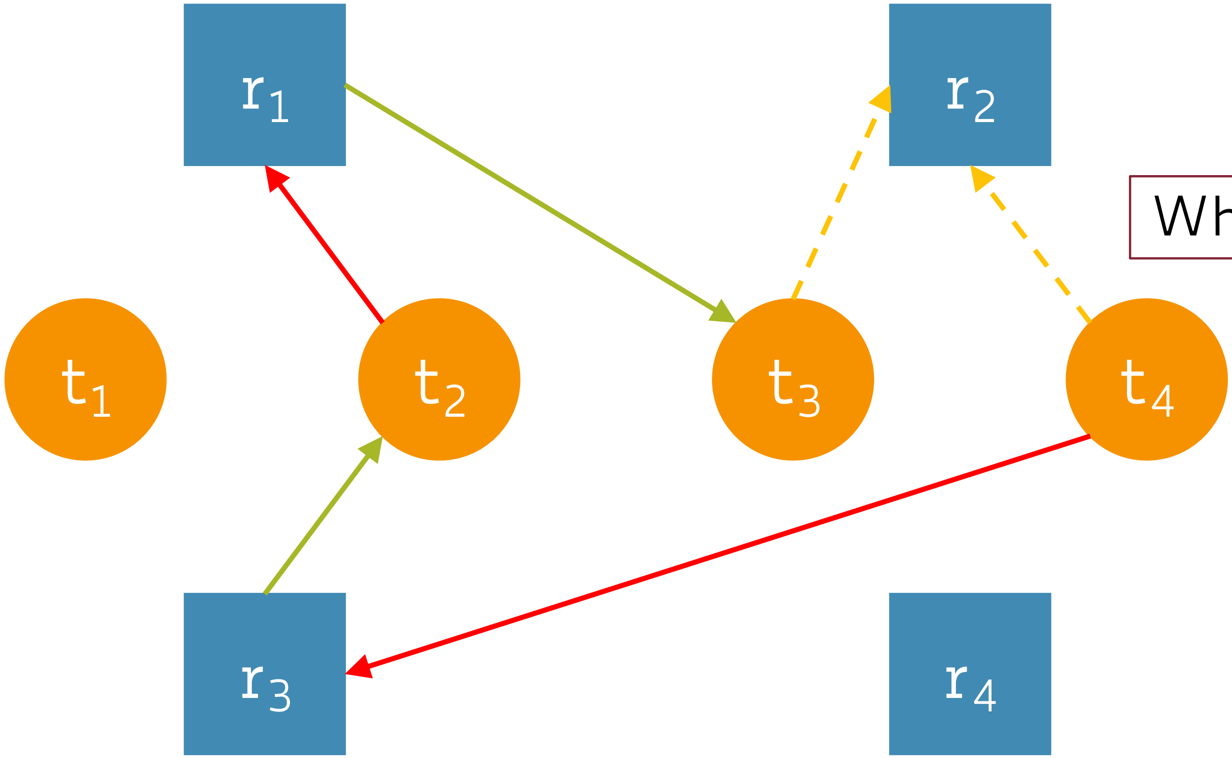
Deadlock Avoidance: Resource Allocation Graph

- A cycle in this extended RAG indicates an unsafe state
- If the allocation results in an unsafe state, this will be denied even if the resource is actually available
- In other words, the claim edge is converted into a request edge and the thread will wait
- NOTE: This solution does not work when there are multiple instances of the same resource

Deadlock Avoidance: Resource Allocation Graph

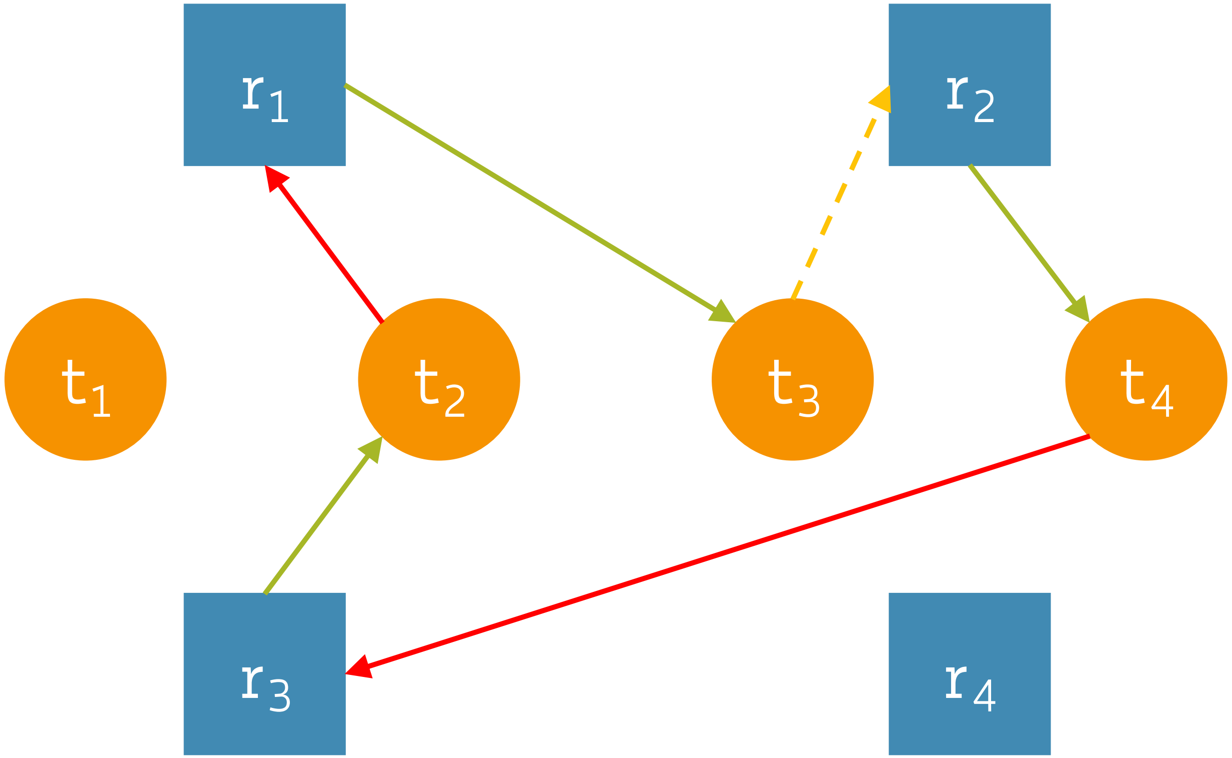


Deadlock Avoidance: Resource Allocation Graph



What happens if t_4 is given r_2?

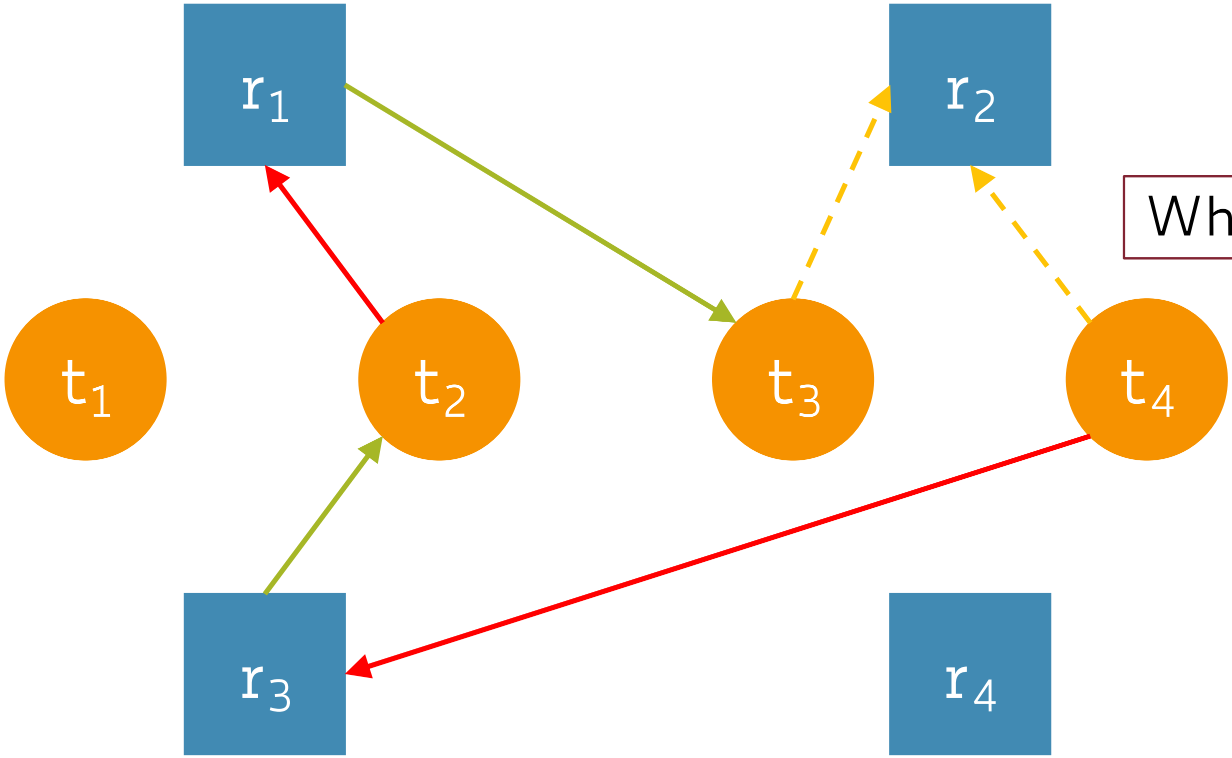
Deadlock Avoidance: Resource Allocation Graph



We are introducing a potential cycle (t_3 requests r_2), which in turn might cause deadlock

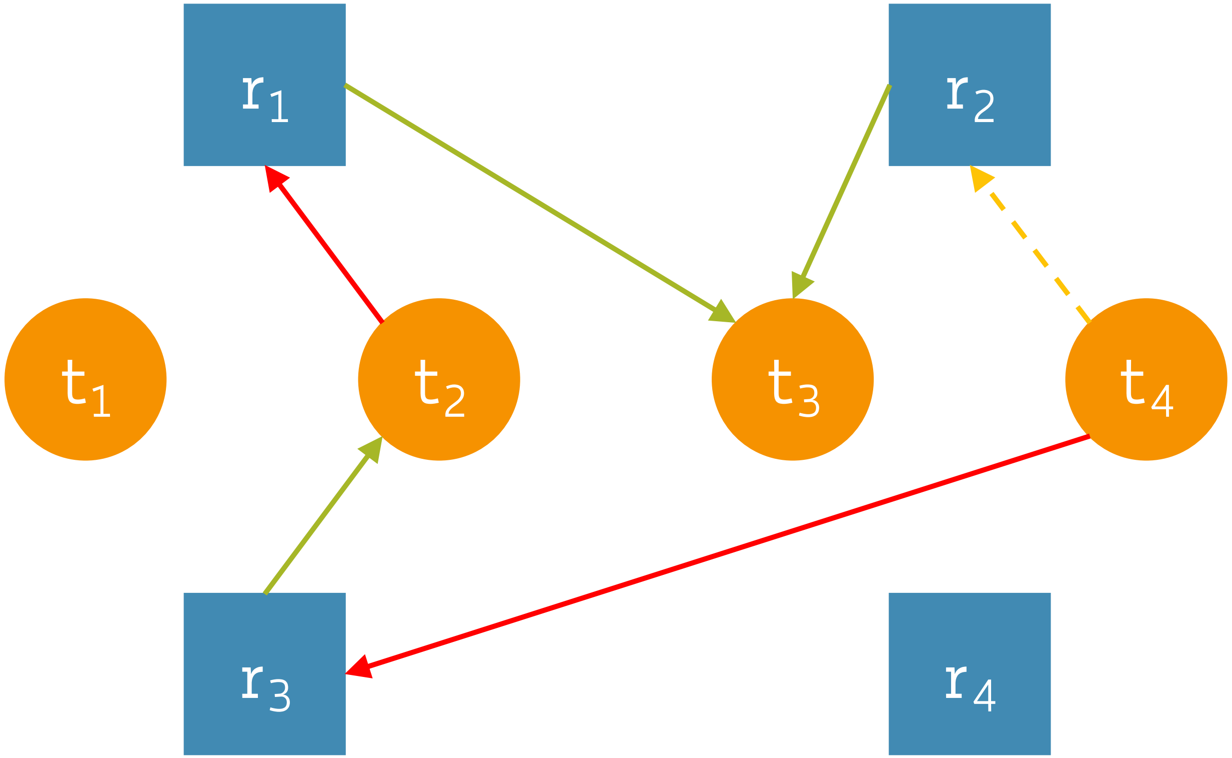
unsafe state

Deadlock Avoidance: Resource Allocation Graph



What happens if t_3 is given r_2?

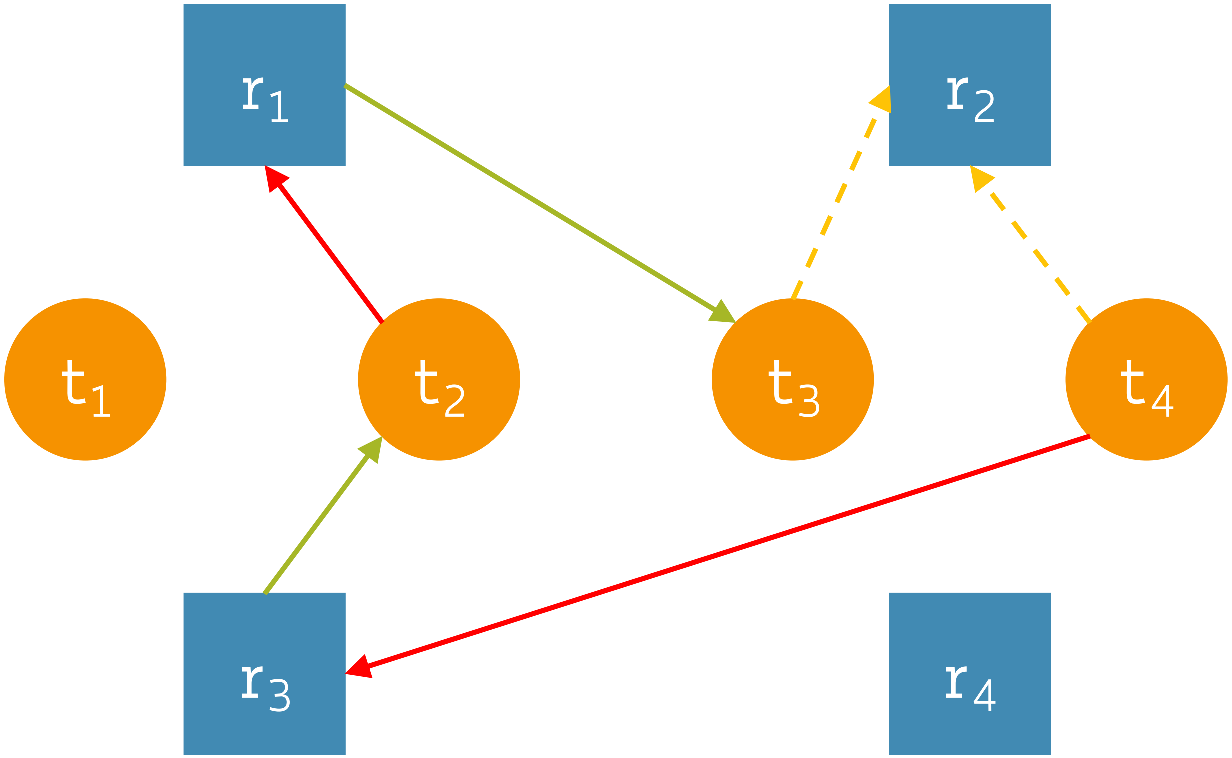
Deadlock Avoidance: Resource Allocation Graph



We are not introducing any potential cycle (t_4 requests r_2)

safe state

Deadlock Avoidance: Resource Allocation Graph



Start from a safe state

Invariant
Accept a request iff we move from a safe state to another

Banker's Algorithm

- Handles multiple instances of the same resource
- Forces threads to provide information on what resource they might need, in advance
- The resources requested must not exceed the total available in the system
- The algorithm allocates resources to a requesting thread if the allocation leaves the system in a safe state, otherwise the thread waits

Banker's Algorithm: Data Structures

- n = number of threads; m = number of resource types
- available[1..m]: m-dimensional vector
  - available[j] = k means there are k resources of type j available
- max[1..n, 1..m]: n x m matrix
  - max[i, j] = k means thread i may require at most k resources of type j
- allocation[1..n, 1..m]: n x m matrix
  - allocation[i, j] = k means thread i has allocated k resources of type j
- need[1..n, 1..m]: n x m matrix
  - need[i, j] = max[i, j] – allocation[i, j] = k means thread i may need k more resources of type j to complete its task

Banker's Algorithm: Idea

- The algorithm is divided in 2 tasks:
  - isSafeState → given the current status of allocation of resources, tests if this is a safe state
  - resourceRequest → given a thread and its resource request decides if such a request can be satisfied

Banker's Algorithm: Idea

- The algorithm is divided in 2 tasks:
  - isSafeState → given the current status of allocation of resources, tests if this is a safe state
  - resourceRequest → given a thread and its resource request decides if such a request can be satisfied
- A request can be satisfied iff this leads to a safe state!

Banker's Algorithm: Idea

- The algorithm is divided in 2 tasks:
  - isSafeState → given the current status of allocation of resources, tests if this is a safe state
  - resourceRequest → given a thread and its resource request decides if such a request can be satisfied
- A request can be satisfied iff this leads to a safe state!
- In other words, the second tasks uses the output of the first one in order to make a decision

Banker's Algorithm: isSafeState

1. Let work and finish be vectors of length m and n, respectively

   Initialize: work = available; finish[i] = false; for all i

2. Find an i such that:

   finish[i] = false && need[i] ≤ work

   If no such i exists, go to step 4.

3. Assume thread i executes:

   work = work + allocation[i]; finish[i] = true; go to step 2.

4. If finish[i] == true for all i, the system is in a safe state

Banker's Algorithm: requestResource

Input: i (thread) and request an m-dimensional vector of requests

1. If request > need[i] raise an error as thread i is attempting to request more resources that it claimed, otherwise go to step 2.
2. If request > available thread i must wait since resources are not available, otherwise go to step 3.
3. Even if resources are available, test if this allocation will lead to a safe state by simulating it

   available -= request; allocation[i] += request; need[i] -= request;

   isSafeState() ? OK : rollback() and wait()

Banker's Algorithm: Example

A snapshot of the current state of the system

		RESOURCES								
		MAX			ALLOCATION			AVAILABLE		
		A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1			
	T_1	1	7	5	1	0	0			
	T_2	2	3	5	1	3	5			
	T_3	0	6	5	0	6	3			
	Total				2	9	9	1	5	2

Banker's Algorithm: Example

Q1: How many resources of type A, B, and C are there overall?

		RESOURCES								
		MAX			ALLOCATION			AVAILABLE		
		A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1			
	T_1	1	7	5	1	0	0			
	T_2	2	3	5	1	3	5			
	T_3	0	6	5	0	6	3			
	Total				2	9	9	1	5	2

Banker's Algorithm: Example

Q1: How many resources of type A, B, and C are there overall?

		RESOURCES								
		MAX			ALLOCATION			AVAILABLE		
		A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1			
	T_1	1	7	5	1	0	0			
	T_2	2	3	5	1	3	5			
	T_3	0	6	5	0	6	3			
	Total				2	9	9	1	5	2

A = 2 + 1 = 3
B = 9 + 5 = 14
C = 9 + 2 = 11

Banker's Algorithm: Example

Q2: What is the content of the NEED matrix?

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1						
	T_1	1	7	5	1	0	0						
	T_2	2	3	5	1	3	5						
	T_3	0	6	5	0	6	3						
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

Q2: What is the content of the NEED matrix?

NEED[i, j] = MAX[i, j] – ALLOCATION[i, j]

THREADS		RESOURCES									NEED		
		MAX			ALLOCATION			AVAILABLE					
		A	B	C	A	B	C	A	B	C	A	B	C
	T_0	0	0	1	0	0	1						
	T_1	1	7	5	1	0	0						
	T_2	2	3	5	1	3	5						
	T_3	0	6	5	0	6	3						
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

Q2: What is the content of the NEED matrix?

NEED[i, j] = MAX[i, j] – ALLOCATION[i, j]

		RESOURCES								
		MAX			ALLOCATION			AVAILABLE		
		A	B	C	A	B	C	A	B	C
THREADS	T_0	$\boxed{0}$	0	1	$\boxed{0}$	0	1			
	T_1	1	7	5	1	0	0			
	T_2	2	3	5	1	3	5			
	T_3	0	6	5	0	6	3			
	Total				2	9	9	1	5	2

NEED		
A	B	C
0-0 = 0		

Banker's Algorithm: Example

Q2: What is the content of the NEED matrix?

NEED[i, j] = MAX[i, j] – ALLOCATION[i, j]

		RESOURCES								
		MAX			ALLOCATION			AVAILABLE		
		A	B	C	A	B	C	A	B	C
THREADS	T_0	0	$\boxed{0}$	1	0	$\boxed{0}$	1			
	T_1	1	7	5	1	0	0			
	T_2	2	3	5	1	3	5			
	T_3	0	6	5	0	6	3			
	Total				2	9	9	1	5	2

NEED		
A	B	C
0	0–0 = 0	

Banker's Algorithm: Example

Q2: What is the content of the NEED matrix?

NEED[i, j] = MAX[i, j] – ALLOCATION[i, j]

		RESOURCES								
		MAX			ALLOCATION			AVAILABLE		
		A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1			
	T_1	1	7	5	1	0	0			
	T_2	2	3	5	1	3	5			
	T_3	0	6	5	0	6	3			
	Total				2	9	9	1	5	2

NEED		
A	B	C
0	0	1–1 = 0

Banker's Algorithm: Example

Q2: What is the content of the NEED matrix?

NEED[i, j] = MAX[i, j] – ALLOCATION[i, j]

		RESOURCES									NEED		
		MAX			ALLOCATION			AVAILABLE					
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

Q3: Is the system in a safe state? Why?

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

Let's start with T_0

		RESOURCES									NEED		
		MAX			ALLOCATION			AVAILABLE					
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

Eventually, T_0 finishes and releases all its resources

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

T_1 can't execute as it still might NEED (0, 7, 5) and AVAILABLE = (1, 5, 3)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	-	-	-				-	-	-
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	8	1	5	3			

Banker's Algorithm: Example

T_2 can execute as it still might NEED (1, 0, 0) and AVAILABLE = (1, 5, 3)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	-	-	-				-	-	-
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	8	1	5	3			

Banker's Algorithm: Example

T_2 can execute as it still might NEED (1, 0, 0) and AVAILABLE = (1, 5, 3)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	-	-	-				-	-	-
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	2	3	5				0	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				3	9	8	0	5	3			

Banker's Algorithm: Example

T_2 eventually finishes and releases all its resources

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	-	-	-				-	-	-
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	-	-	-				-	-	-
	T_3	0	6	5	0	6	3				0	0	2
	Total				1	6	3	2	8	8			

Banker's Algorithm: Example

T_3 can execute as it still might NEED (0, 0, 2) and AVAILABLE = (2, 8, 8)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	0	6	3				0	0	2
	Total				1	6	3	2	8	8			

Banker's Algorithm: Example

T_3 can execute as it still might NEED (0, 0, 2) and AVAILABLE = (2, 3, 6)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	0	6	5				0	0	0
	Total				1	6	5	2	8	6			

Banker's Algorithm: Example

T_3 eventually finishes and releases all its resources

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	–	–	–				–	–	–
	Total				1	0	0	2	14	11			

Banker's Algorithm: Example

T_1 can now execute since NEED (0, 7, 5) and AVAILABLE = (2, 14, 11)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	-	-	-				-	-	-
	T_1	1	7	5	1	7	5				0	0	0
	T_2	2	3	5	-	-	-				-	-	-
	T_3	0	6	5	-	-	-				-	-	-
	Total				1	7	5	2	7	6			

Banker's Algorithm: Example

We have found a sequence of execution T_0, T_2, T_3, T_1 which leads to safe state!

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	–	–	–				–	–	–
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	–	–	–				–	–	–
	Total				–	–	–	3	14	11			

Banker's Algorithm: Example

Q4: If T_1 issues a REQUEST (0, 5, 2), can this be granted immediately?

		RESOURCES									NEED		
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

We have to ask ourselves: 1. if the request can be satisfied;
2. if it will lead to a safe state

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

To answer 1. check if: a. REQUEST <= NEED and b. REQUEST <= AVAILABLE

		RESOURCES									NEED		
		MAX			ALLOCATION			AVAILABLE					
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

1.a. REQUEST <= NEED?

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

1.a. REQUEST <= NEED? YES! (0, 5, 2) <= (0, 7, 5)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

1.b. REQUEST <= AVAILABLE?

		RESOURCES									NEED		
		MAX			ALLOCATION			AVAILABLE					
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

1.b. REQUEST <= AVAILABLE? YES! (0, 5, 2) <= (1, 5, 2)

		RESOURCES									NEED		
		MAX			ALLOCATION			AVAILABLE					
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

To answer 2. we simulate the request is granted and see if we are still in a safe state

		RESOURCES									NEED		
		MAX			ALLOCATION			AVAILABLE					
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	0	0				0	7	5
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	9	9	1	5	2			

Banker's Algorithm: Example

To answer 2. we simulate the request is granted and see if we are still in a safe state

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	14	11	1	0	0			

Banker's Algorithm: Example

Let's start with T_0

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	0	0	1				0	0	0
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	14	11	1	0	0			

Banker's Algorithm: Example

Eventually, T_0 finishes and releases all its resources

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	14	10	1	0	1			

Banker's Algorithm: Example

T_1 can't execute as it still might NEED (0, 2, 3) and AVAILABLE = (1, 0, 1)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	-	-	-				-	-	-
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	14	10	1	0	1			

Banker's Algorithm: Example

T_2 can execute as it still might NEED (1, 0, 0) and AVAILABLE = (1, 0, 1)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	1	3	5				1	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				2	14	10	1	0	1			

Banker's Algorithm: Example

T_2 can execute as it still might NEED (1, 0, 0) and AVAILABLE = (1, 0, 1)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	-	-	-				-	-	-
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	2	3	5				0	0	0
	T_3	0	6	5	0	6	3				0	0	2
	Total				3	14	10	0	0	1			

Banker's Algorithm: Example

T_2 eventually finishes and releases all its resources

		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	0	6	3				0	0	2
	Total				1	11	5	2	3	6			

Banker's Algorithm: Example

T_3 can execute as it still might NEED (0, 0, 2) and AVAILABLE = (2, 3, 6)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	0	6	3				0	0	2
	Total				1	11	5	2	3	6			

Banker's Algorithm: Example

T_3 can execute as it still might NEED (0, 0, 2) and AVAILABLE = (2, 3, 6)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	0	6	5				0	0	0
	Total				1	11	7	2	3	4			

Banker's Algorithm: Example

T_3 eventually finishes and releases all its resources

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	5	2				0	2	3
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	–	–	–				–	–	–
	Total				1	5	2	2	9	9			

Banker's Algorithm: Example

T_1 can now execute since NEED (0, 2, 3) and AVAILABLE = (2, 9, 9)

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	1	7	5				0	0	0
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	–	–	–				–	–	–
	Total				1	7	5	2	7	6			

Banker's Algorithm: Example

We have found a sequence of execution T_0, T_2, T_3, T_1 which leads to safe state!

		RESOURCES											
		MAX			ALLOCATION			AVAILABLE			NEED		
		A	B	C	A	B	C	A	B	C	A	B	C
THREADS	T_0	0	0	1	–	–	–				–	–	–
	T_1	1	7	5	–	–	–				–	–	–
	T_2	2	3	5	–	–	–				–	–	–
	T_3	0	6	5	–	–	–				–	–	–
	Total				–	–	–	3	14	11			

Summary

- Deadlock → a situation in which a set of threads/processes cannot proceed because each one requires resources held by another

Summary

- Deadlock → a situation in which a set of threads/processes cannot proceed because each one requires resources held by another
- Detection and Recovery → recognize deadlock after it has occurred and break it

Summary

- Deadlock → a situation in which a set of threads/processes cannot proceed because each one requires resources held by another
- Detection and Recovery → recognize deadlock after it has occurred and break it
- Prevention → design resource allocation strategies which guarantee at least one of the 4 necessary deadlock conditions never holds

Summary

- Deadlock → a situation in which a set of threads/processes cannot proceed because each one requires resources held by another
- Detection and Recovery → recognize deadlock after it has occurred and break it
- Prevention → design resource allocation strategies which guarantee at least one of the 4 necessary deadlock conditions never holds
- Avoidance → runtime checks to avoid deadlock online

Summary

In practice, most OSs don't do anything and leave it all to applications