Sistemi Operativi I

Corso di Laurea in Informatica

2025-2026

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

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 prevention
- Deadlock avoidance
- Deadlock detection and recovery

Our Journey

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

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 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 prevention (offline): imposes restrictions/rules on how to write deadlock-free programs
- Deadlock avoidance (online): scheduling threads to avoid deadlocks

Deadlock: Terminology

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

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 prevention
- Deadlock avoidance
- Deadlock detection and recovery

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 (e.g., only one thread grabs a lock)

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 (e.g., only one thread grabs a lock)
  - Hold-and-Wait → at least one thread is holding a non-sharable resource (e.g., a lock) and is waiting for other resource(s) to become available (e.g., other locks to acquire)

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 (e.g., only one thread grabs a lock)
  - Hold-and-Wait → at least one thread is holding a non-sharable resource (e.g., a lock) and is waiting for other resource(s) to become available (e.g., other locks to acquire)
  - 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 (e.g., only one thread grabs a lock)
  - Hold-and-Wait → at least one thread is holding a non-sharable resource (e.g., a lock) and is waiting for other resource(s) to become available (e.g., other locks to acquire)
  - 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 circular chain of waiting threads t_1, …, t_n where t_i holds a resource requested by $t_{(i+1)\%n}$

Our Journey

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

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 a lock when it requests another
    - Acquire all locks at once, atomically
    - Use a global lock that wraps the acquisition of all locks
    - Hard to predict all the resources a thread will need and inefficient!

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
    - Some thread libraries allow "trying" acquiring multiple locks
    - 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 prevention
- Deadlock avoidance
- Deadlock detection and recovery

Deadlock Avoidance via Scheduling

- An alternative to statically preventing deadlock upfront

Deadlock Avoidance via Scheduling

- An alternative to statically preventing deadlock upfront
- Avoidance requires some global knowledge of which locks various threads can grab

Deadlock Avoidance via Scheduling

- An alternative to statically preventing deadlock upfront
- Avoidance requires some global knowledge of which locks various threads can grab
- Based on that knowledge, the OS will schedule threads to guarantee that no deadlock occurs

Deadlock Avoidance via Scheduling

- An alternative to statically preventing deadlock upfront
- Avoidance requires some global knowledge of which locks various threads can grab
- Based on that knowledge, the OS will schedule threads to guarantee that no deadlock occurs
- Can be used only in limited environments where one has full knowledge of all tasks and locks needed

Deadlock Avoidance: Example

- 4 threads: T_1, T_2, T_3, T_4
- 2 CPUs: CPU_1, CPU_2
- knowledge: T_1 grabs locks L_1 and L_2 (in some order)
  T_2 grabs locks L_1 and L_2 (in some order)
  T_3 grabs locks L_2
  T_4 grabs no locks

Deadlock Avoidance: Example

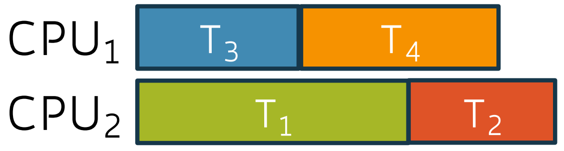
- 4 threads: T_1, T_2, T_3, T_4
- 2 CPUs: CPU_1, CPU_2
- knowledge: T_1 grabs locks L_1 and L_2 (in some order)
  T_2 grabs locks L_1 and L_2 (in some order)
  T_3 grabs locks L_2
  T_4 grabs no locks

A smart scheduler can avoid deadlock by not running T_1 and T_2 in parallel

Deadlock Avoidance: Example

- 4 threads: T_1, T_2, T_3, T_4
- 2 CPUs: CPU_1, CPU_2
- knowledge: T_1 grabs locks L_1 and L_2 (in some order)
  T_2 grabs locks L_1 and L_2 (in some order)
  T_3 grabs locks L_2
  T_4 grabs no locks

A smart scheduler can avoid deadlock by not running T_1 and T_2 in parallel



T_3 and T_1 (T_2) can run in parallel

Deadlock Avoidance: Safe State

- There exists **at least one** sequence of execution (safe sequence)

Deadlock Avoidance: Safe State

- There exists **at least one** sequence of execution (safe sequence)
- Each thread can obtain all the resources it needs, complete its execution, and release the resources

Deadlock Avoidance: Safe State

- There exists **at least one** sequence of execution (safe sequence)
- Each thread can obtain all the resources it needs, complete its execution, and release the resources
- An unsafe state does not necessarily mean deadlock (i.e., some threads may not request all the resources they declared)

Deadlock Avoidance: Safe State

- There exists **at least one** sequence of execution (safe sequence)
- Each thread can obtain all the resources it needs, complete its execution, and release the resources
- An unsafe state does not necessarily mean deadlock (i.e., some threads may not request all the resources they 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

- There exists **at least one** sequence of execution (safe sequence)
- Each thread can obtain all the resources it needs, complete its execution, and release the resources
- An unsafe state does not necessarily mean deadlock (i.e., some threads may not request all the resources they declared)
- Grant a resource to a thread if the new state is safe, otherwise make it wait even if the resource is available
- Given n threads, the naïve, brute-force approach would require analyzing **all the possible permutations** of them O(n!)

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: Idea

- Keep track of which threads can finish given the current available resources

Banker's Algorithm: Idea

- Keep track of which threads can finish given the current available resources
- Whenever one thread that can finish is found, pretend it finishes and releases its resources

Banker's Algorithm: Idea

- Keep track of which threads can finish given the current available resources
- Whenever one thread that can finish is found, pretend it finishes and releases its resources
- Repeat the process until:
  - all processes can finish (→ **safe state**), or
  - no remaining process can finish (→ **unsafe state**)

Banker's Algorithm: Idea

- Keep track of which threads can finish given the current available resources
- Whenever one thread that can finish is found, pretend it finishes and releases its resources
- Repeat the process until:
  - all processes can finish ($\rightarrow$ **safe state**), or
  - no remaining process can finish ($\rightarrow$ **unsafe state**)
- This solution ensures that if any safe sequence exists, one will be found **without needing to test all permutations**

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: Pseudocode

```
work = available                    // m-dimensional vectors
finish[i] = false for all i

repeat                                         // outer loop
    found = false
    for i in 1..n:                             // inner loop
        if not finish[i] and need[i] <= work:
            work = work + allocation[i]
            finish[i] = true
            found = true
until not found

if all finish[i] == true   // safe state
     return true
return false               // unsafe state
```

Banker's Algorithm: Time Complexity

```
work = available                    // m-dimensional vectors
finish[i] = false for all i

repeat                                        // outer loop
    found = false
    for i in 1..n:                            // inner loop
        if not finish[i] and need[i] <= work:
            work = work + allocation[i]
            finish[i] = true
            found = true
until not found

if all finish[i] == true  // safe state
     return true
return false              // unsafe state
```

The **inner loop** scans all n threads to look for one that can finish → O(n)

Banker's Algorithm: Time Complexity

```
work = available                    // m-dimensional vectors
finish[i] = false for all i

repeat                                        // outer loop
    found = false
    for i in 1..n:                            // inner loop
        if not finish[i] and need[i] <= work:
            work = work + allocation[i]
            finish[i] = true
            found = true
until not found

if all finish[i] == true  // safe state
     return true
return false               // unsafe state
```

The **outer loop** runs as long as we find at least one process that can finish.
In the worst case, finding and completing only one thread at each iteration → O(n)

Banker's Algorithm: Time Complexity

```
work = available                  // m-dimensional vectors
finish[i] = false for all i

repeat                                     // outer loop
    found = false
    for i in 1..n:                         // inner loop
        if not finish[i] and need[i] <= work:
            work = work + allocation[i]
            finish[i] = true
            found = true
until not found

if all finish[i] == true   // safe state
     return true
return false               // unsafe state
```

Checking need[i] <= work may require up to m resource types → O(m) per thread

Banker's Algorithm: Time Complexity

```
work = available                  // m-dimensional vectors
finish[i] = false for all i

repeat                                     // outer loop
    found = false
    for i in 1..n:                         // inner loop
        if not finish[i] and need[i] <= work:
            work = work + allocation[i]
            finish[i] = true
            found = true
until not found

if all finish[i] == true  // safe state
     return true
return false             // unsafe state
```

Overall, $O(n^2 * m)$

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]

		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]

		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 = 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			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 = 0	
	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	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											
		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

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											
		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

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	$\boxed{9}$	1	5	$\boxed{2}$			

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

		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											
		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											
		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 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	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

		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									NEED		
		MAX			ALLOCATION			AVAILABLE					
		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			

Our Journey

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

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:

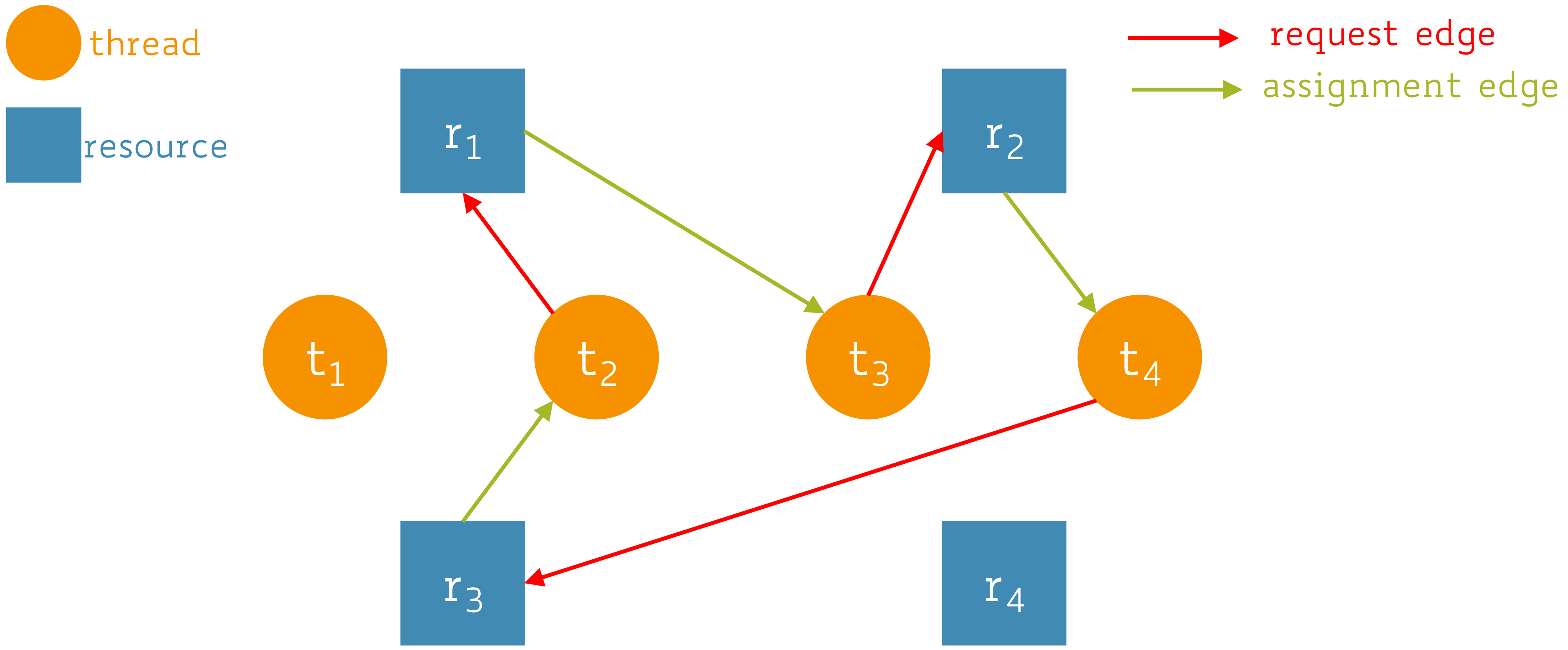
Deadlock Detection: Resource Allocation Graph

- 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

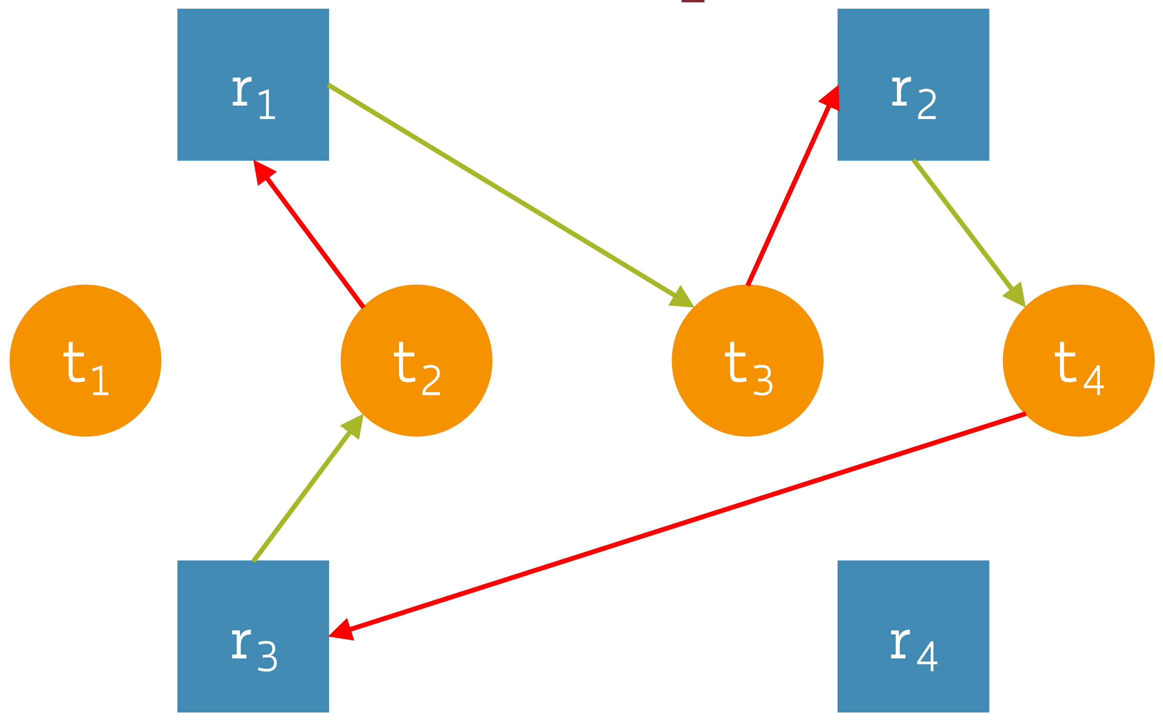
Deadlock Detection: Resource Allocation Graph

- 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

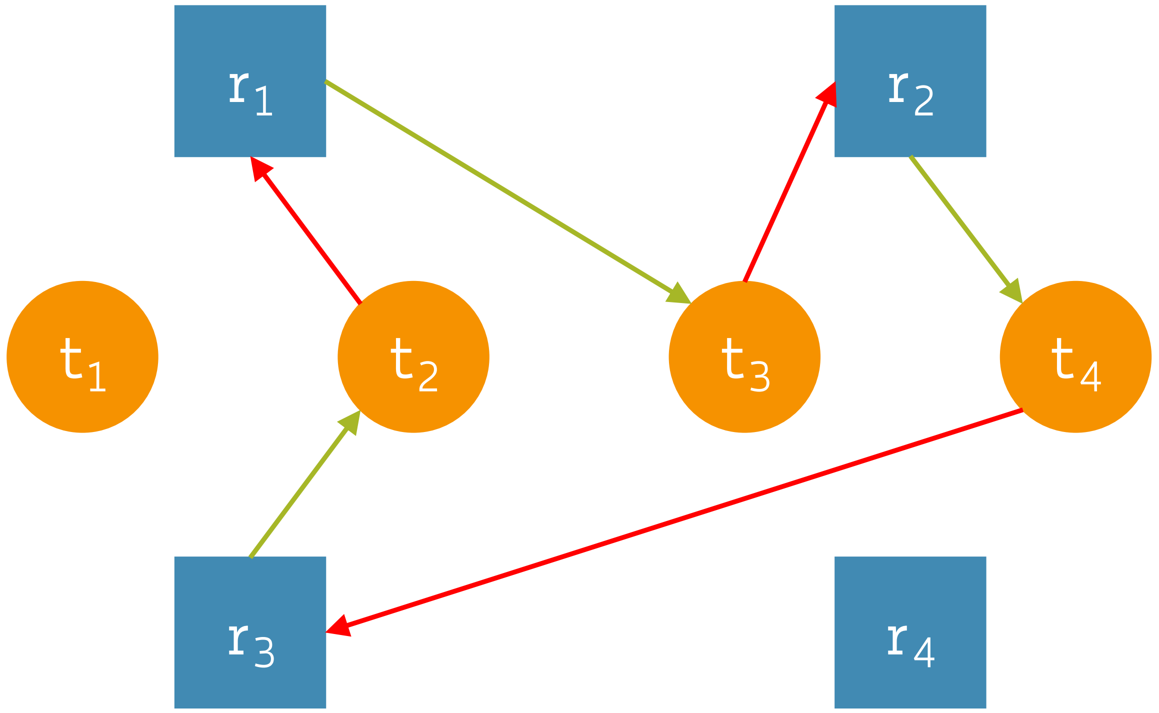


Deadlock Detection: Resource Allocation Graph



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

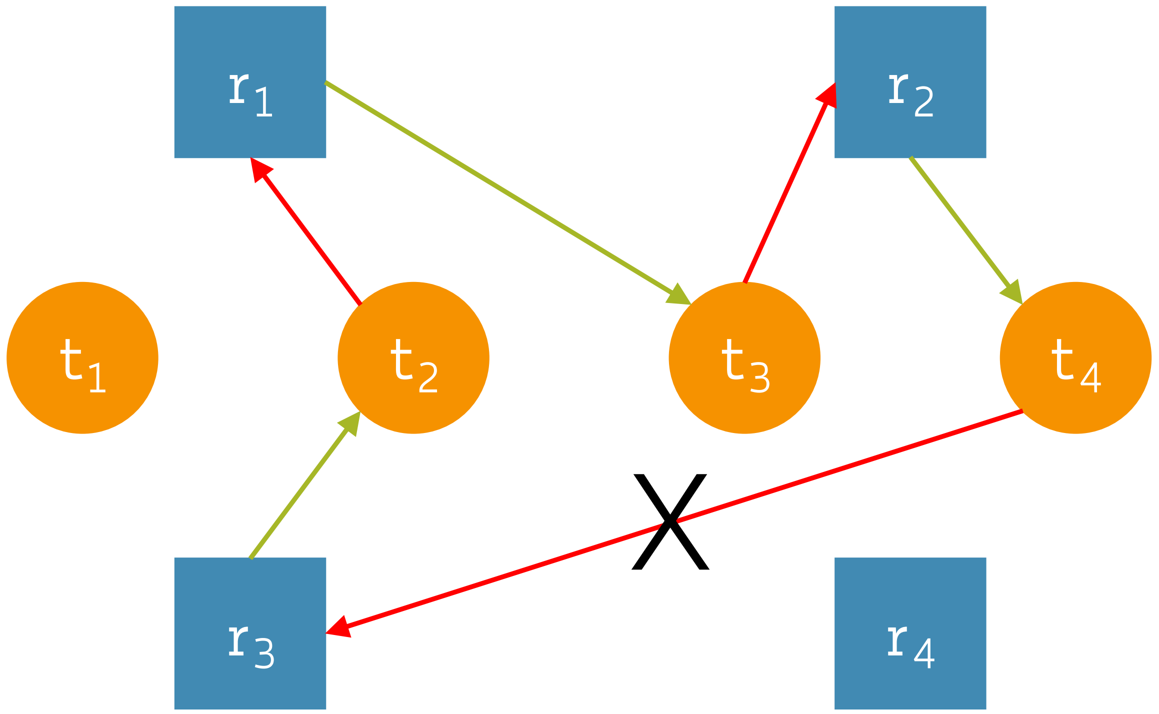
Deadlock Detection: Resource Allocation Graph



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

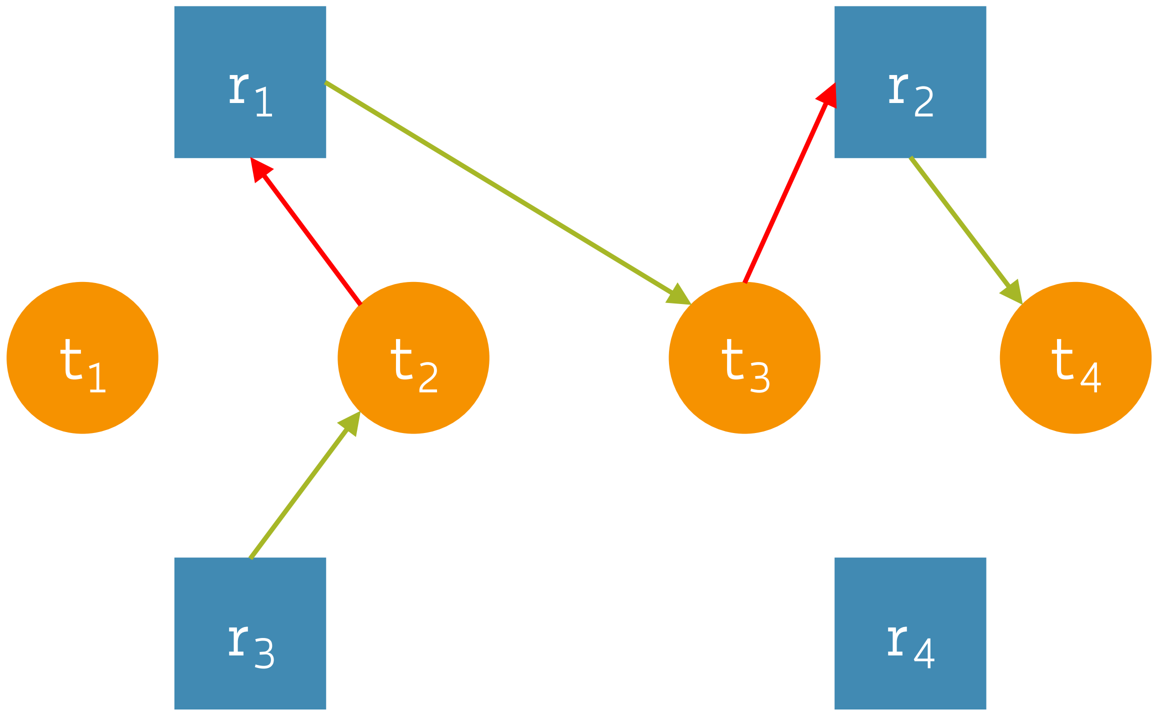
Why?

Deadlock Detection: Resource Allocation Graph



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

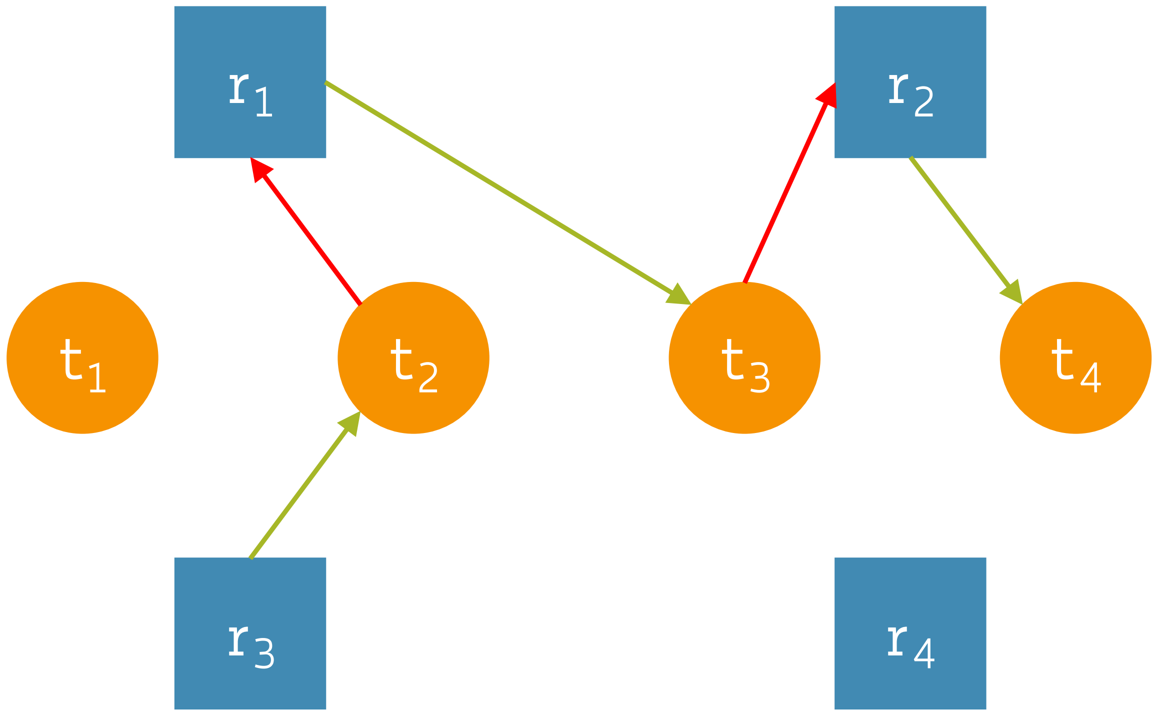
Deadlock Detection: Resource Allocation Graph



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

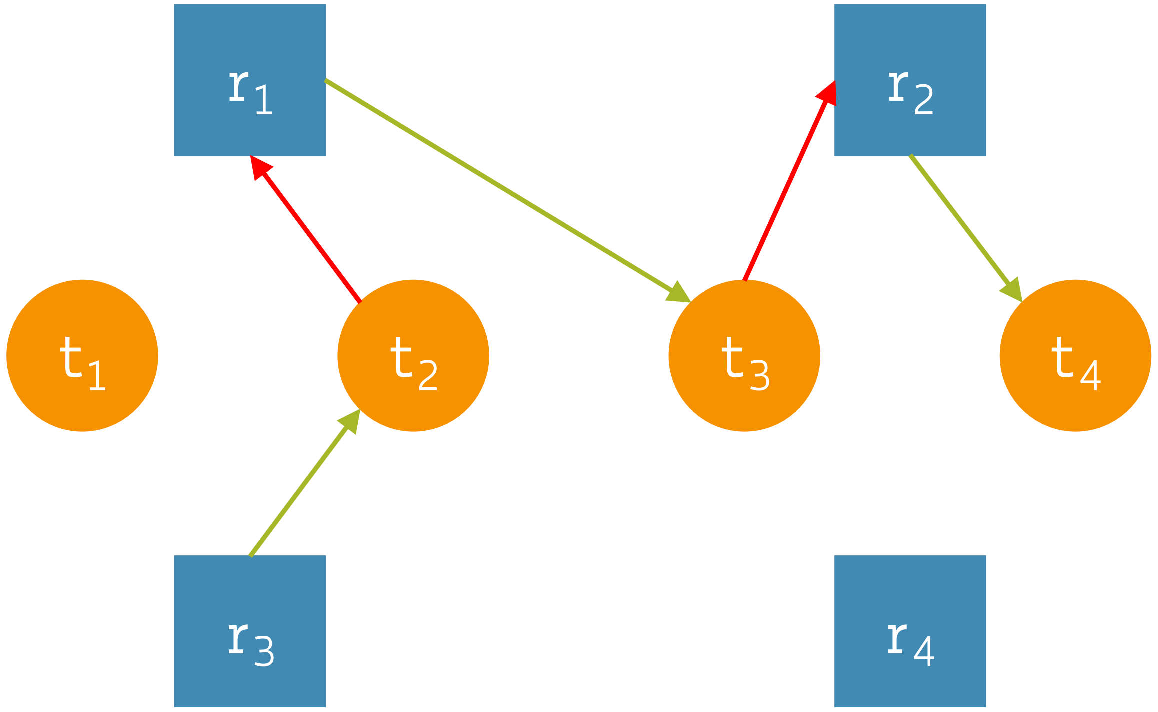


Suppose we remove the edge (t_4, r_3) so as to remove the cycle

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



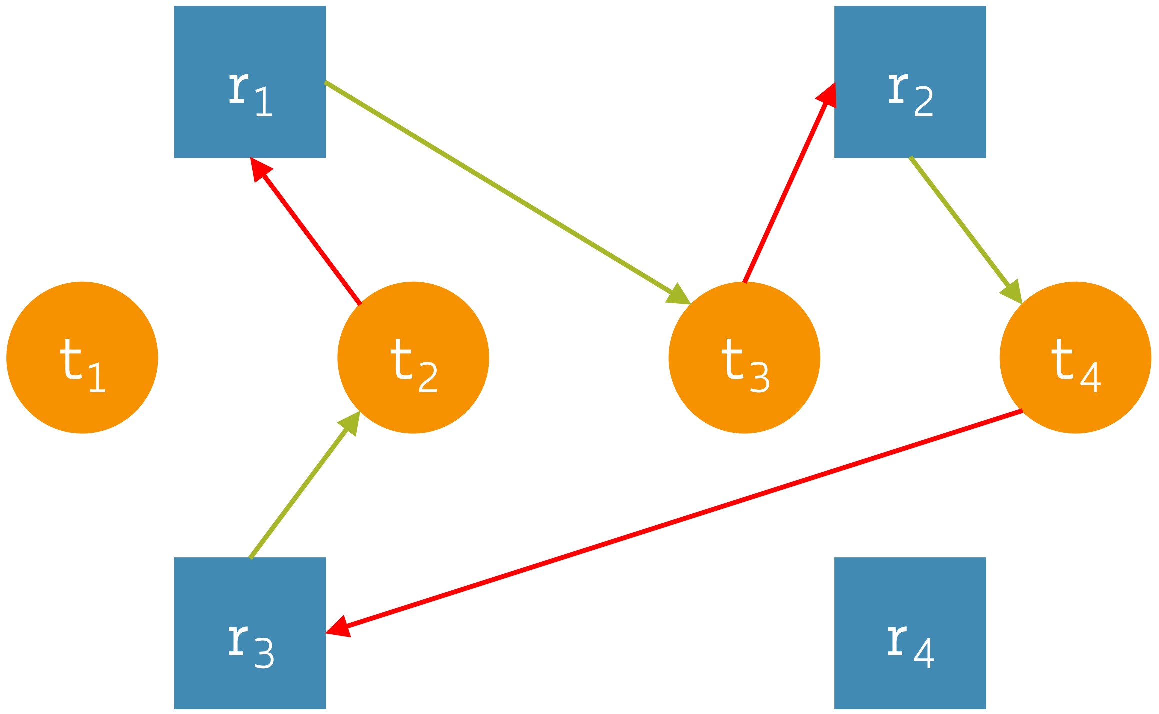
Suppose we remove the edge (t_4, r_3) so as to remove the cycle

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

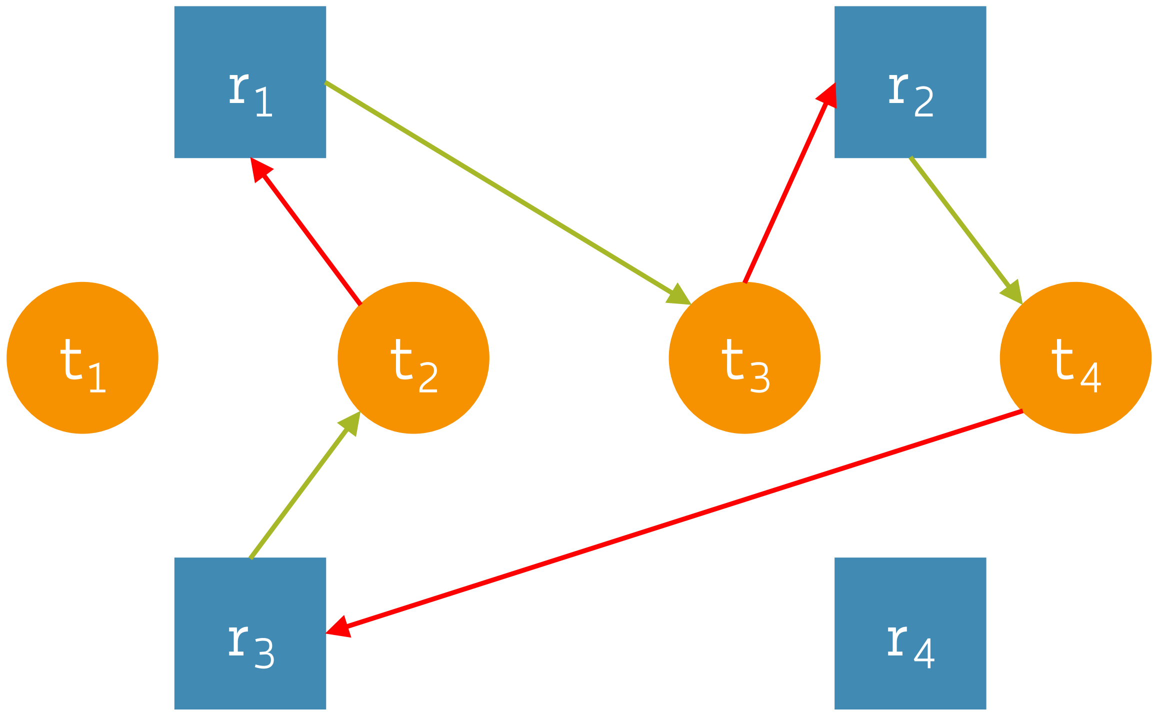
And so on and so forth…

Deadlock Detection: Resource Allocation Graph



If the graph has cycles, deadlock might exist

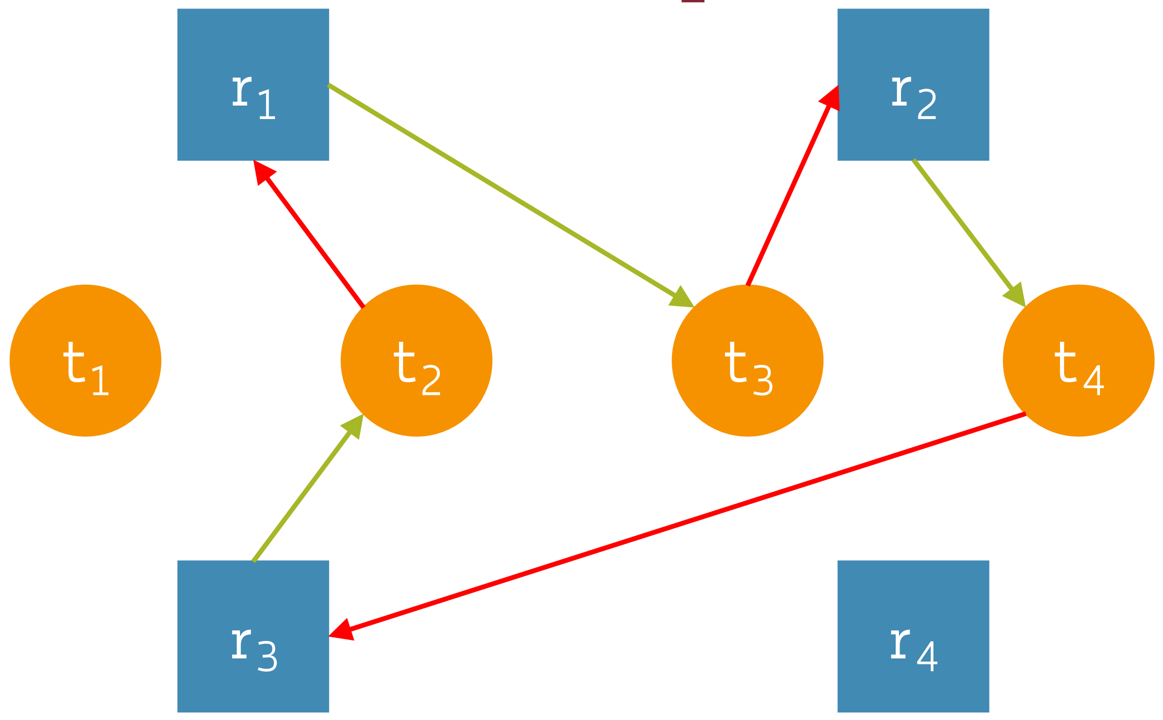
Deadlock Detection: Resource Allocation Graph



If the graph has cycles, deadlock might exist

Why?

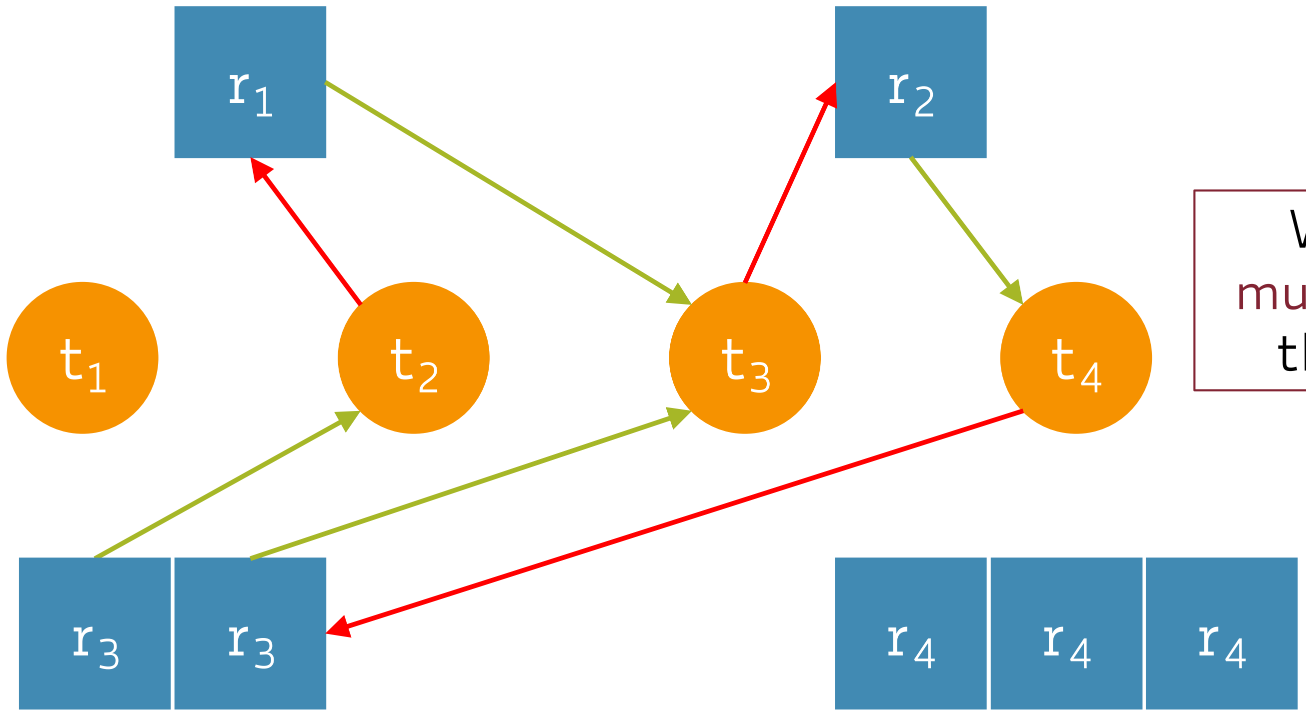
Deadlock Detection: Resource Allocation Graph



If the graph has cycles, deadlock might exist

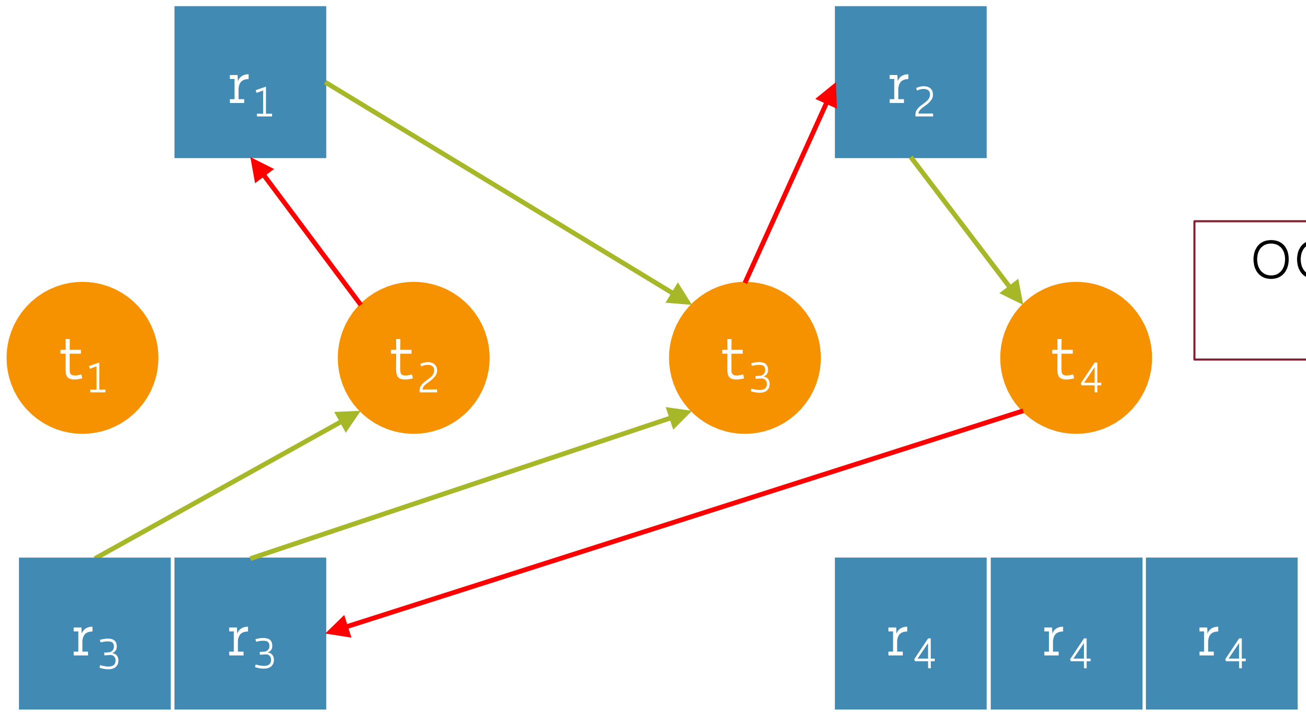
We are assuming the multiplicity of each resource is 1 (i.e., one r_1, one r_2, etc.)

Deadlock Detection: Resource Allocation Graph



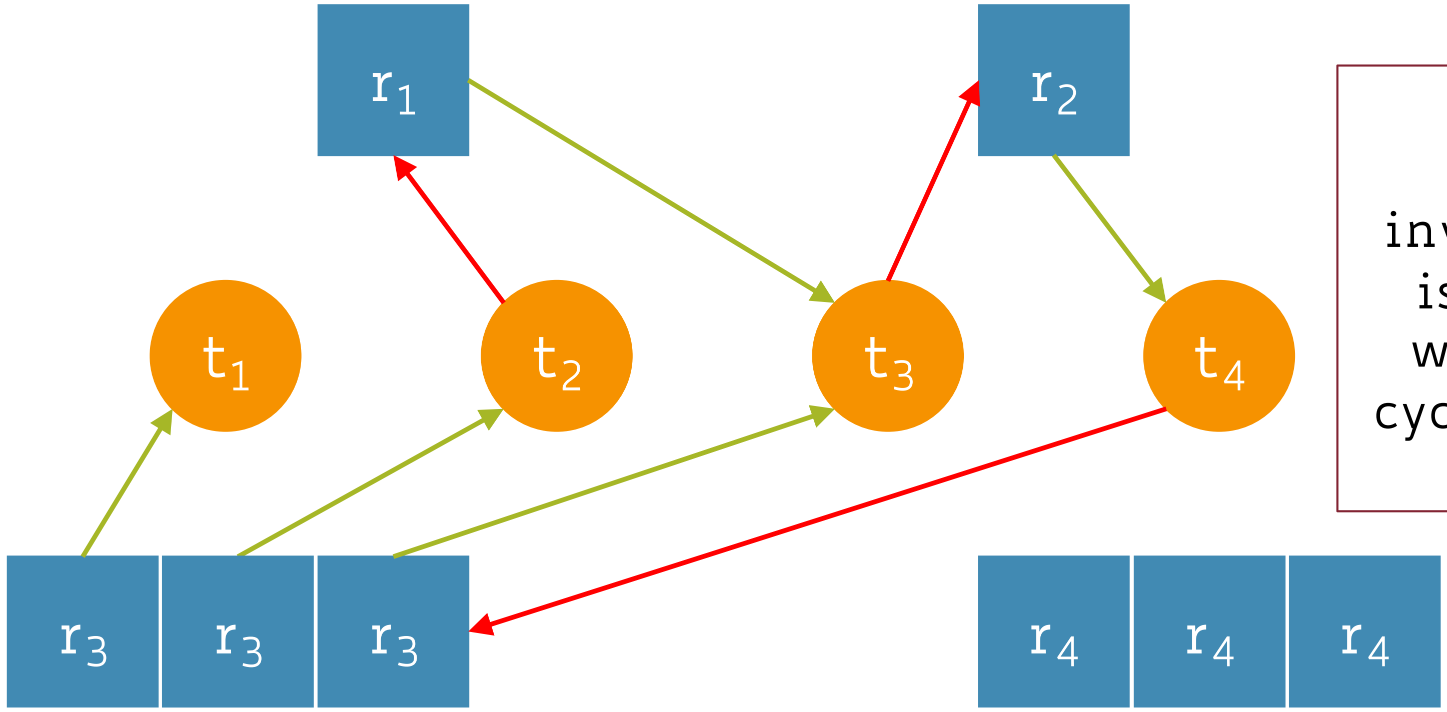
What if there are multiple instances of the same resource?

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

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
- Prevention → design resource allocation protocols that 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
- Prevention → design resource allocation protocols that guarantee at least one of the 4 necessary deadlock conditions never holds
- Avoidance → scheduling threads so as to avoid deadlock

Summary

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

Summary

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

Summary

After all, if deadlocks are rare, a non-solution like a hard reboot is often the best!