

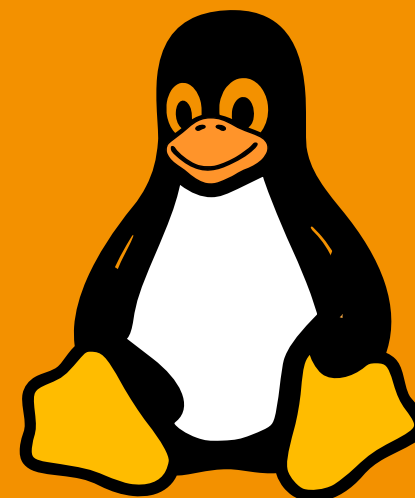
Start: 8:01



**Prof. Dr. Florian Künzner**

Technical University of Applied Sciences Rosenheim, Computer Science

## OS 10 – Deadlocks

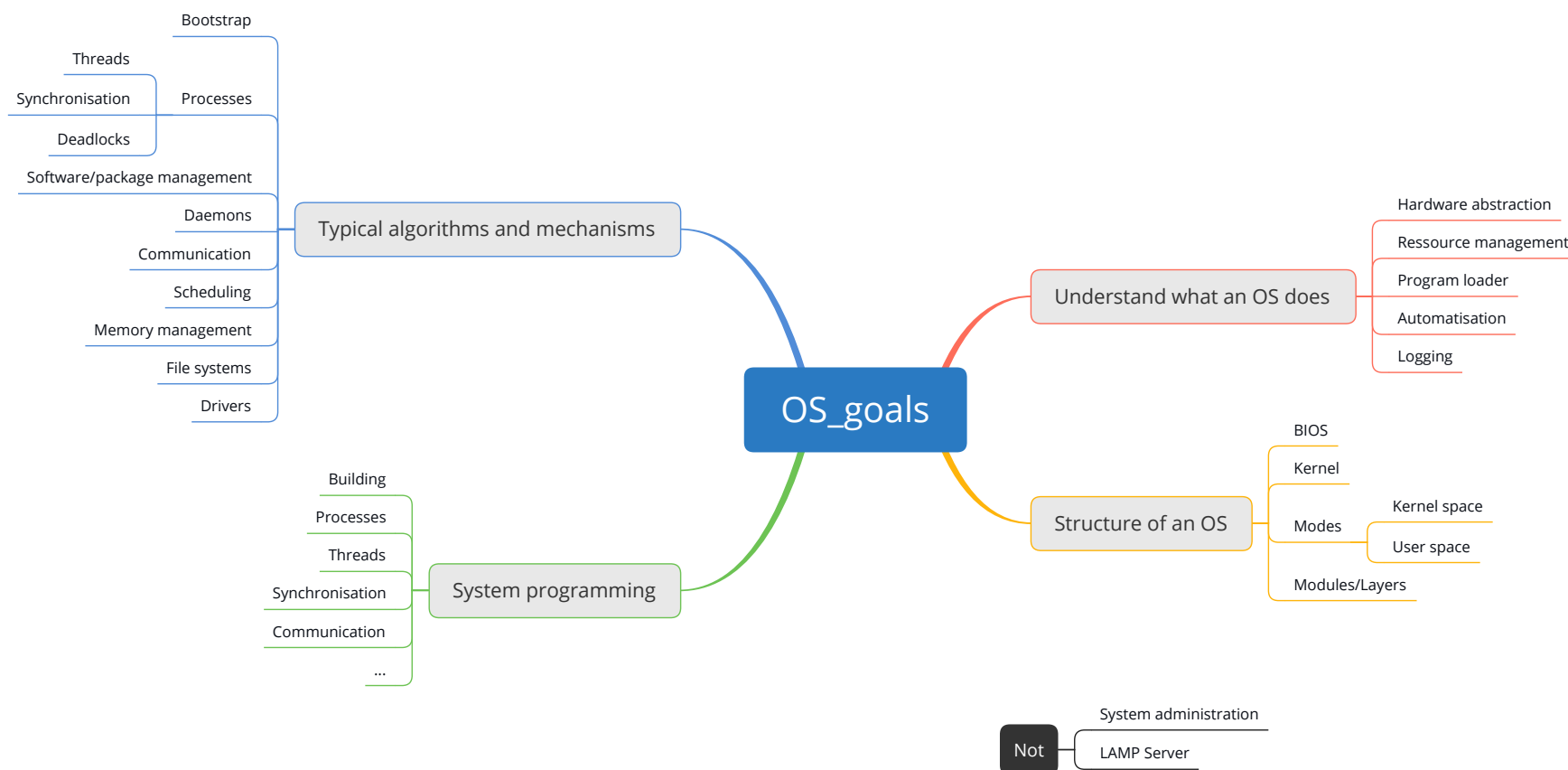


source: iconspng.com

The lecture is based on the work and the documents of Prof. Dr. Ludwig Frank



# Goal



# Goal

## OS::Deadlocks

- Intro
- Analysis
- Safe state
- Deadlock prevention
- Deadlock recovery

# Intro

Parallelisation with processes and threads and their synchronisation is nice, but...



# Intro

## Example 1



```
1 seminit(s, 0);
```

```
2 process1() {
```

```
3     P(s);
```

```
4     //critical area..,
```

```
5     V(s);
```

```
6 }
```

```
7 process2() {
```

```
8     P(s);
```

```
9     //critical area..,
```

```
10    V(s);
```

```
11 }
```

## Problem

### ■ Deadlock

- Reason: Critical area can't be accessed, because the semaphore is initialised with 0.

# Intro

## Example 1

```
1 seminit(s, 0);  
2 process1() {  
3     P(s);  
4     //critical area..  
5     V(s);  
6 }
```

```
7 process2() {  
8     P(s);  
9     //critical area..  
10    V(s);  
11 }
```

## Problem

- Deadlock
- Reason: Critical area can't be accessed, because the semaphore is initialised with 0.

# Intro

## Example 1

```
1 seminit(s, 0);  
2 process1() {  
3     P(s);  
4     //critical area..  
5     V(s);  
6 }
```

```
7 process2() {  
8     P(s);  
9     //critical area..  
10    V(s);  
11 }
```

## Problem

### ■ Deadlock

- Reason: Critical area can't be accessed, because the semaphore is initialised with 0.

# Intro

## Example 1

```
1 seminit(s, 0);
2 process1() {
3     P(s);
4     //critical
5     V(s);
6 }
```

```

7 process2() {
8     P(s);
9     //critical area..
10    V(s);
11 }

```

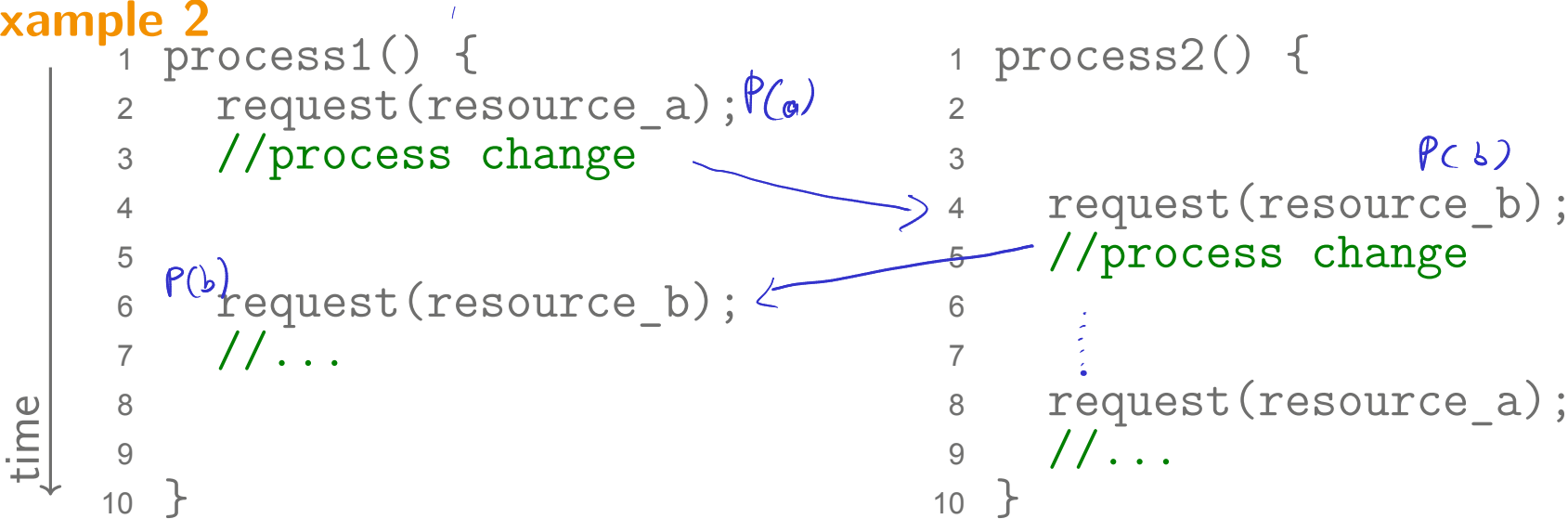
# Problem

- **Deadlock**
- Reason: Critical area can't be accessed, because the **semaphore is initialised with 0**.



# Intro

## Example 2



## Problem

- Deadlock
- Reason: Resource is already assigned to other process (both wait)

# Intro

## Example 2

```

1 process1() {
2     request(resource_a);
3     //process change
4
5
6     request(resource_b);
7     //...
8
9
10 }

```

```
1 process2() {
2
3
4     request(resource_b);
5     //process change
6
7
8     request(resource_a);
9     //...
10 }
```

# Problem

# Intro

## Example 2

```

1 process1() {
2     request(resource_a);
3     //process change
4
5
6     request(resource_b);
7     //...
8
9
10 }

```

```
1 process2() {
2
3
4     request(resource_b);
5     //process change
6
7
8     request(resource_a);
9     //...
10 }
```

# Problem

## Deadlock

# Intro

## Example 2

```

1 process1() {
2     request(resource_a);
3     //process change
4
5
6     request(resource_b);
7     //...
8
9
10 }

```

```
1 process2() {
2
3
4     request(resource_b);
5     //process change
6
7
8     request(resource_a);
9     //...
10 }
```

# Problem

- **Deadlock**
- Reason: Resource is **already assigned** to other process (**both wait**)

# Definition

A **deadlock** is a situation where one or ~~ore~~ more processes wait for resource(s) and no one is able to get its required resource(s), because they are **waiting**.

# Analysis

How can a deadlock occur? We try a systematic analysis.

# Deadlock characterisation

**A deadlock can occur under these conditions**

Condition	Description
-----------	-------------

Original paper: (E. G. COFFMAN et al., 1971) [coffman\\_deadlocks.pdf](#)

More details: <https://en.wikipedia.org/wiki/Deadlock>

# Deadlock characterisation

## A deadlock can occur under these conditions

Condition	Description
Mutual exclusion	Tasks <b>claim exclusive control</b> of the resources they require ("mutual exclusion" condition).

Original paper: (E. G. COFFMAN et al., 1971) [coffman\\_deadlocks.pdf](#)

More details: <https://en.wikipedia.org/wiki/Deadlock>



# Deadlock characterisation

## A deadlock can occur under these conditions

Condition	Description
<b>Mutual exclusion</b>	Tasks <b>claim exclusive control</b> of the resources they require ("mutual exclusion" condition).
<b>Hold and wait</b>	Tasks <b>hold resources already allocated</b> to them while waiting for additional resources ("wait for" condition).

Original paper: (E. G. COFFMAN et al., 1971) [coffman\\_deadlocks.pdf](#)

More details: <https://en.wikipedia.org/wiki/Deadlock>

# Deadlock characterisation

A deadlock can occur under these conditions

Condition	Description
Mutual exclusion	Tasks <b>claim exclusive control</b> of the resources they require (" <b>mutual exclusion</b> " condition).
Hold and wait	Tasks <b>hold resources already allocated</b> to them while waiting for additional resources (" <b>wait for</b> " condition).
No preemption	Resources <b>cannot be forcibly removed</b> from the tasks holding them until the resources are used to completion (" <b>no preemption</b> " condition).

Original paper: (E. G. COFFMAN et al., 1971) [coffman\\_deadlocks.pdf](#)

More details: <https://en.wikipedia.org/wiki/Deadlock>

# Deadlock characterisation

## A deadlock can occur under these conditions

Condition	Description
Mutual exclusion	Tasks <b>claim exclusive control</b> of the resources they require (" <b>mutual exclusion</b> " condition).
Hold and wait	Tasks <b>hold resources already allocated</b> to them while waiting for additional resources (" <b>wait for</b> " condition).
No preemption	Resources <b>cannot be forcibly removed</b> from the tasks holding them until the resources are used to completion (" <b>no preemption</b> " condition).
Circular wait	A circular chain of tasks exist, such that <b>each task holds one or more resources that are being requested by the next task</b> in the chain (" <b>circular wait</b> " condition).

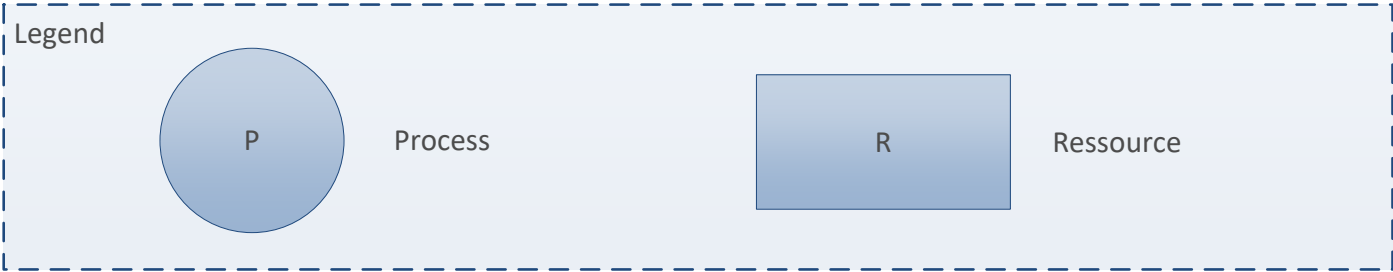
Original paper: (E. G. COFFMAN et al., 1971) [coffman\\_deadlocks.pdf](#)

More details: <https://en.wikipedia.org/wiki/Deadlock>

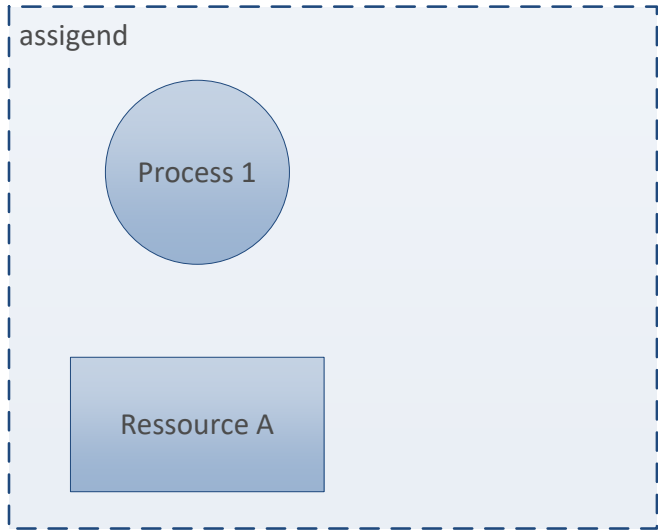
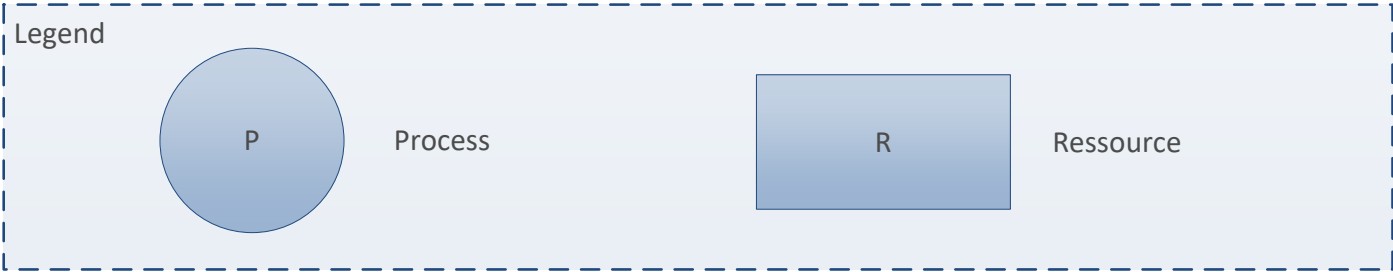
# Analysis: notation



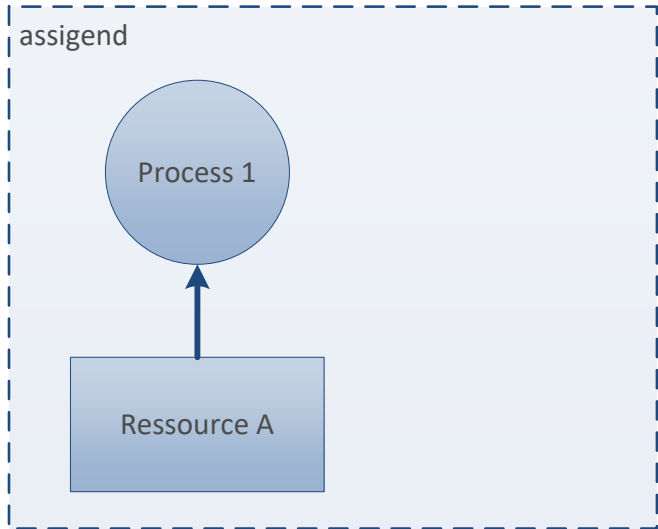
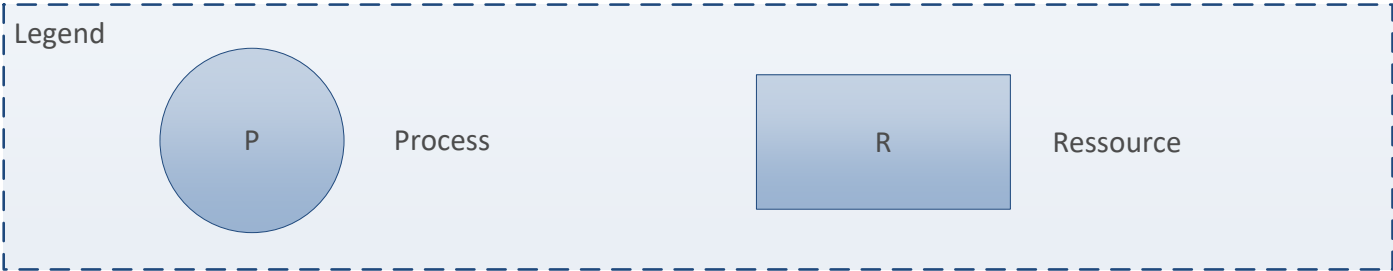
# Analysis: notation



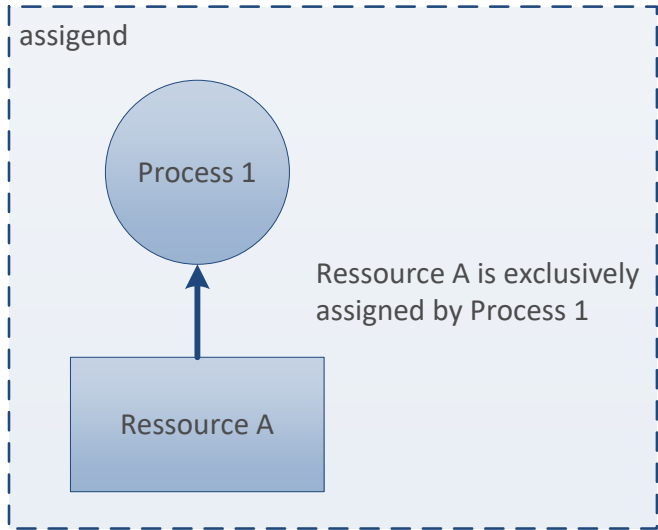
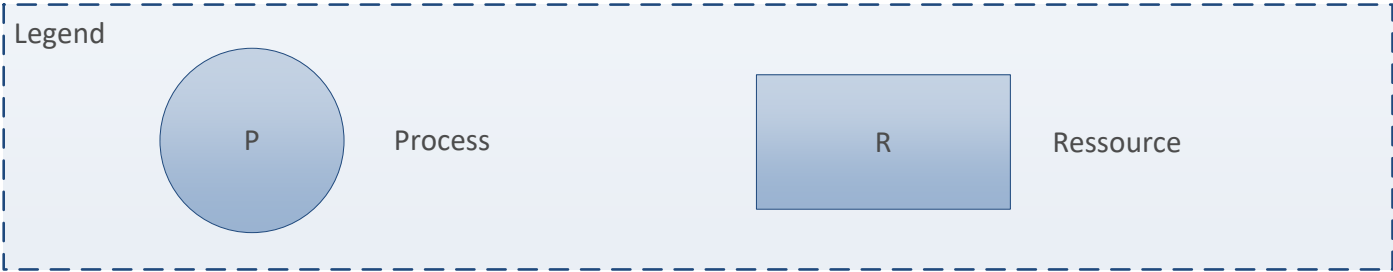
# Analysis: notation



# Analysis: notation

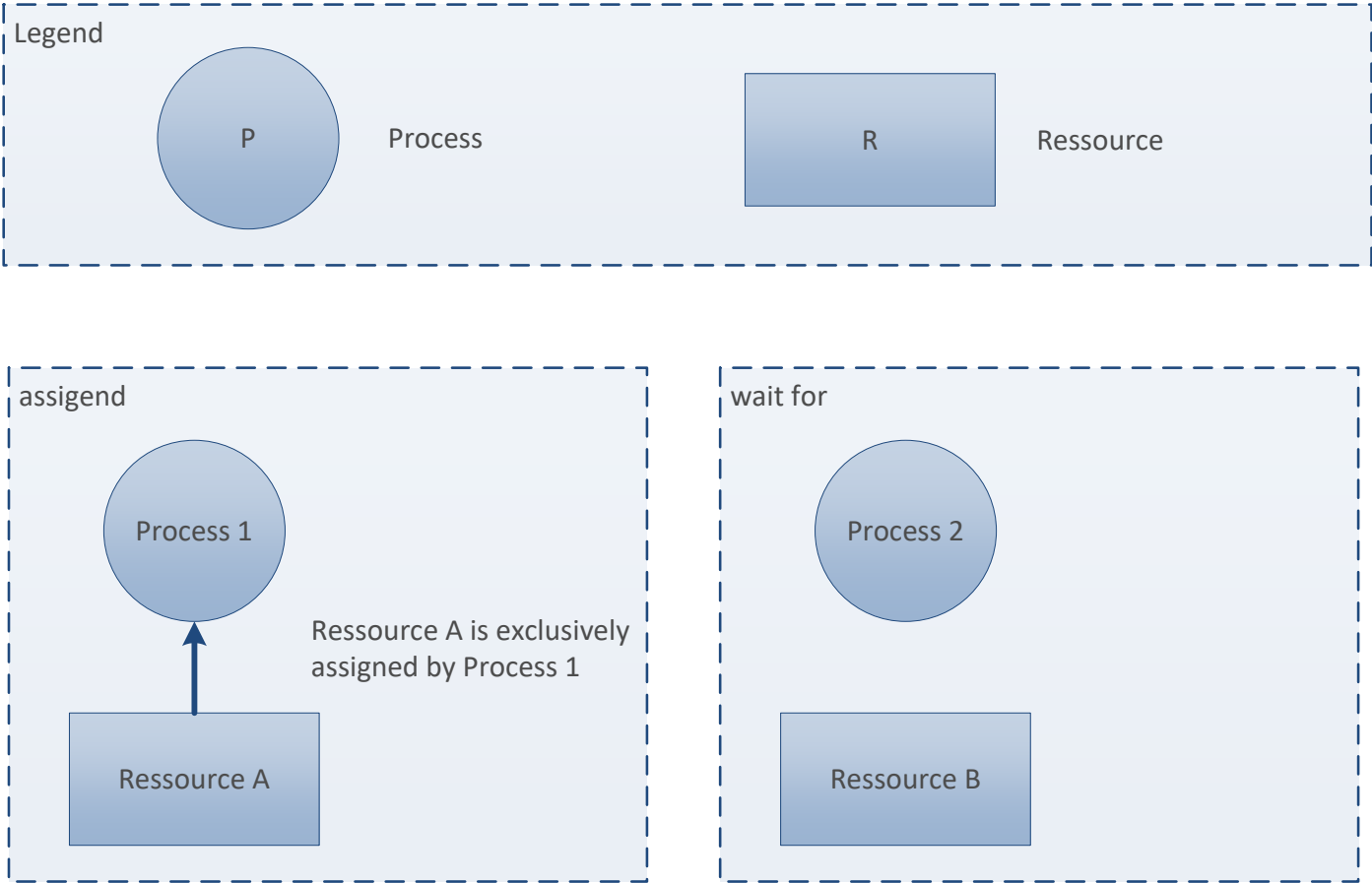


# Analysis: notation

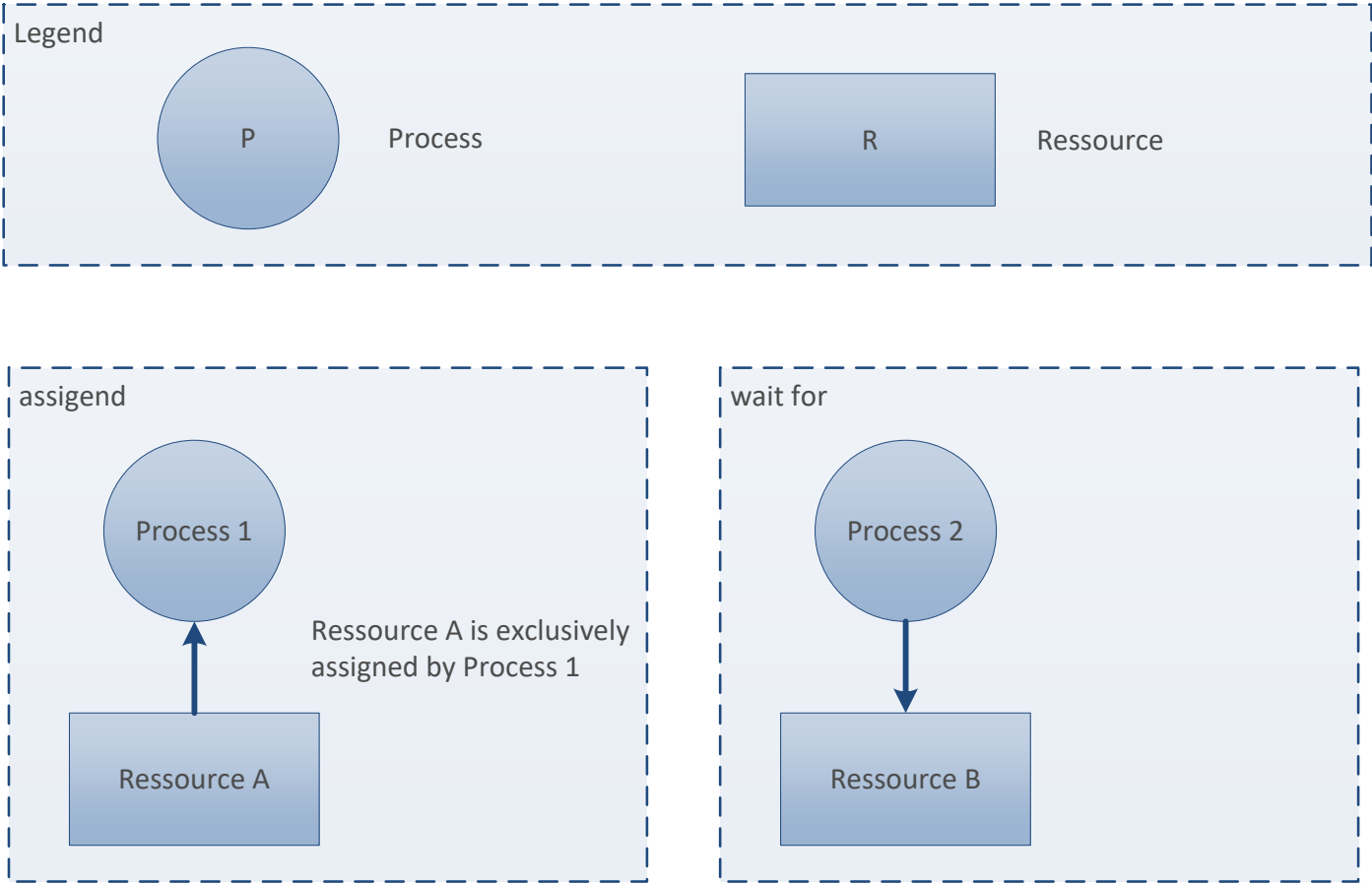




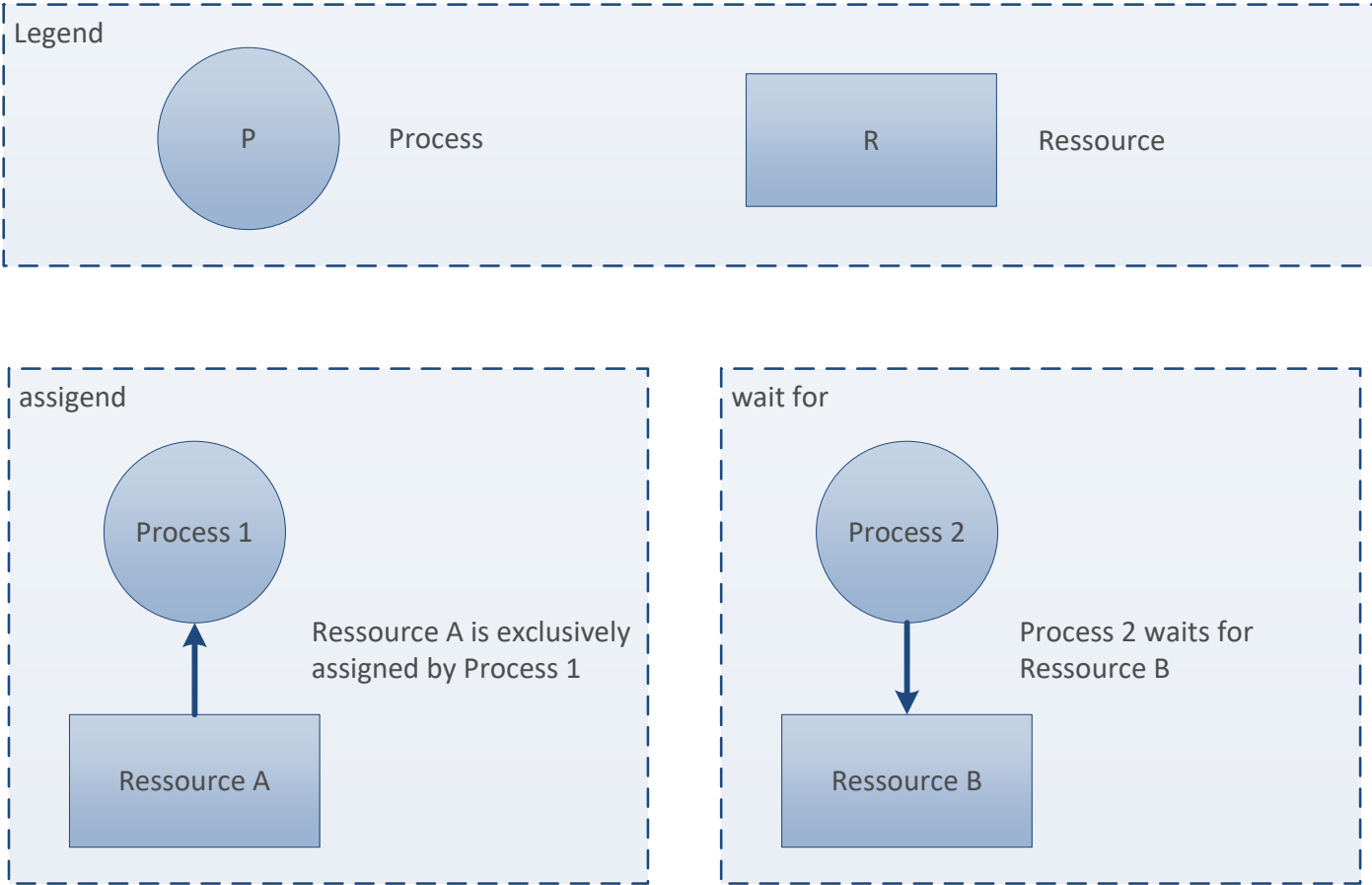
# Analysis: notation



# Analysis: notation

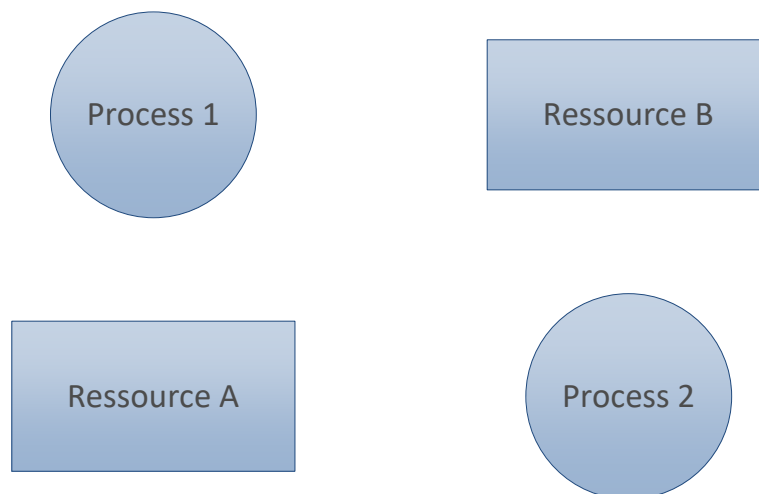


# Analysis: notation



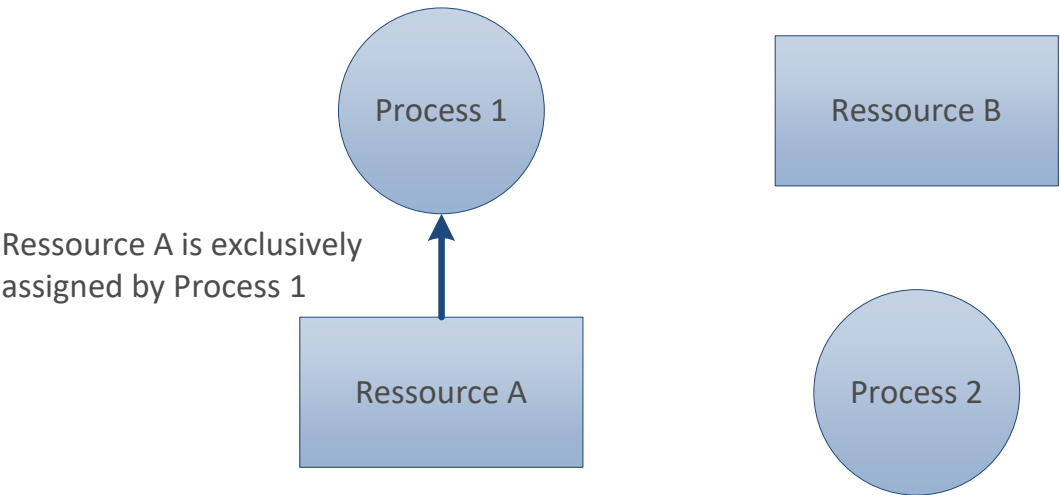


# Analysis: example 1 (graphically)



- $P1 \leftarrow A, P1 \rightarrow B, P2 \leftarrow B, P2 \rightarrow A$
- **Circular wait: deadlock.**

# Analysis: example 1 (graphically)

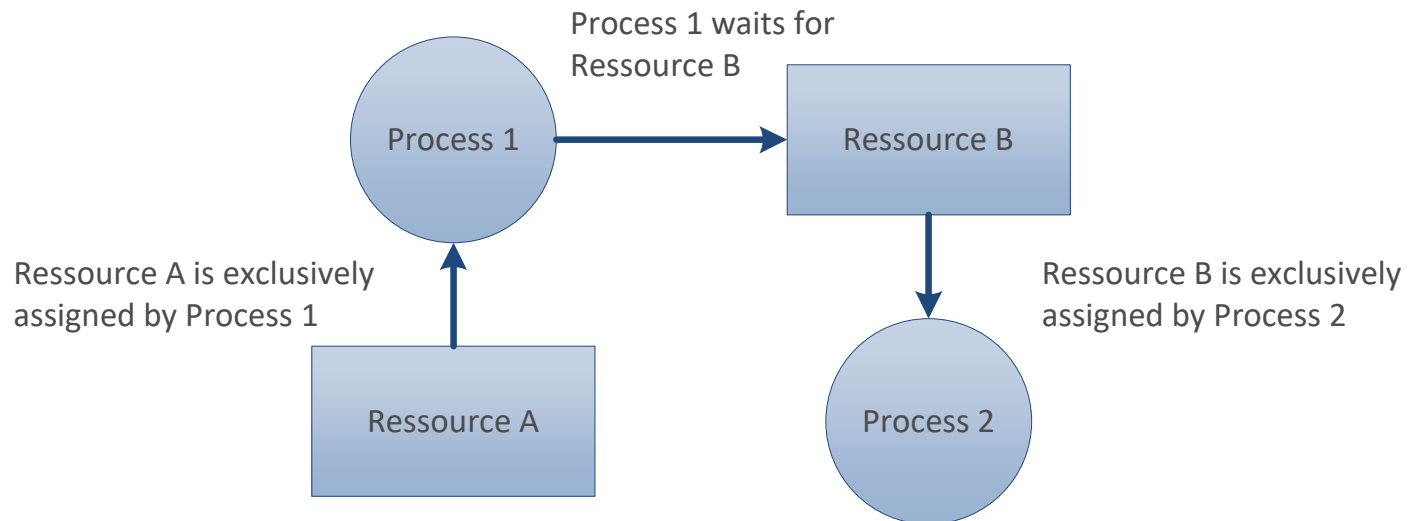


- $P1 \leftarrow A, P1 \rightarrow B, P2 \leftarrow B, P2 \rightarrow A$
- **Circular wait: deadlock.**



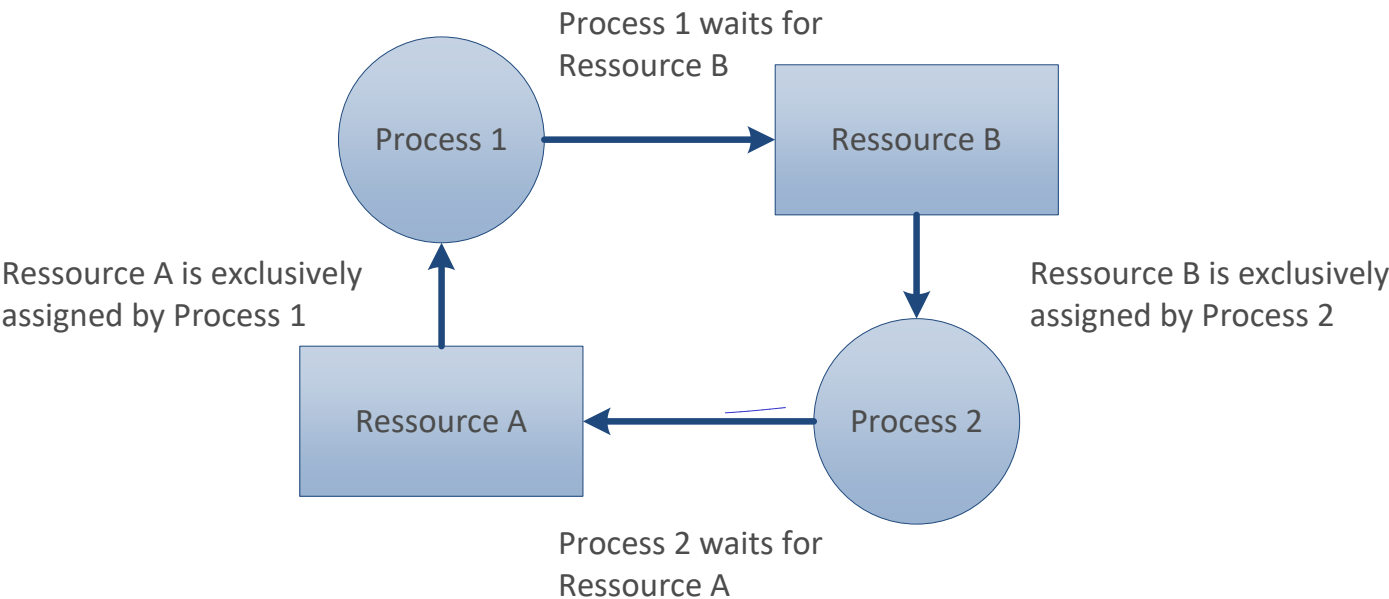


# Analysis: example 1 (graphically)



- $P1 \leftarrow A, P1 \rightarrow B, P2 \leftarrow B, P2 \rightarrow A$
- **Circular wait: deadlock.**

# Analysis: example 1 (graphically)



- $P1 \leftarrow A, P1 \rightarrow B, P2 \leftarrow B, P2 \rightarrow A$
- **Circular wait: deadlock.**



# Questions?

All right?  $\Rightarrow$  

Question?  $\Rightarrow$   and use **chat**

or

**speak** *after* I  
ask you to

# Safe state

Is the system in a safe state?



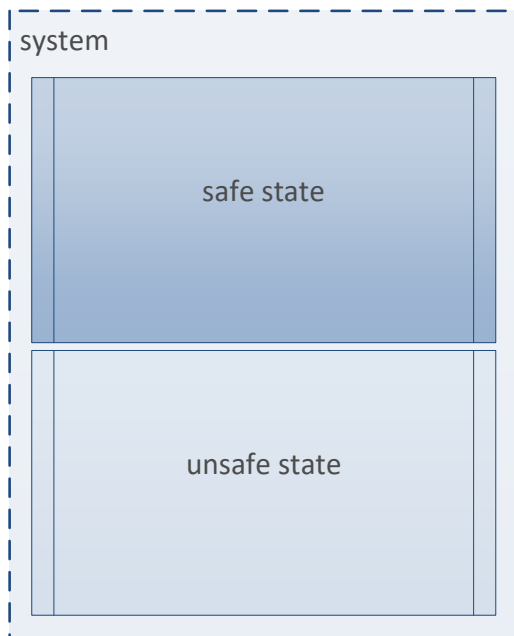
# Safe state



## Safe state

- A state is **safe**, if there is **no deadlock**, and there **exists at least one sequence** of processes that **ends not in a deadlock**.
- It is **guaranteed** that all processes can finish.

# Safe state



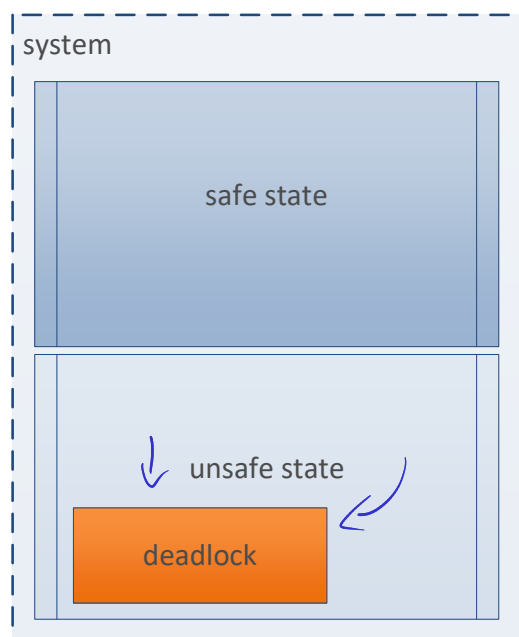
## Safe state

- A state is **safe**, if there is **no deadlock**, and there **exists at least one sequence** of processes that **ends not** in a **deadlock**.
- It is **guaranteed** that all processes can finish.

## Unsafe state

- A state is **unsafe** if there is a **deadlock**, or there exists only sequences that ends in a **deadlock**.
- It is **not guaranteed** that all processes can finish.
- A deadlock is an unsafe state.

# Safe state



## Safe state

- A state is **safe**, if there is **no deadlock**, and there **exists at least one sequence** of processes that **ends not** in a **deadlock**.
- It is **guaranteed** that all processes can finish.

## Unsafe state

- A state is **unsafe** if there is a **deadlock**, or there **exists only sequences** that **ends** in a **deadlock**.
- It is **not guaranteed** that all processes can finish.
- A deadlock is an unsafe state.

# Safe state: terms

## Safe sequence of processes

A *sequence of processes*  $(P_1, \dots, P_n)$  is **safe** for a certain state of acquired resources, if *for every process*  $P_i$  ( $1 \leq i \leq n$ ) it is **guaranteed** that *all required resources*  $(R_1, \dots, R_m)$  can be **granted** including the resources that are acquired by  $P_j$  ( $j < i$ ).





# Safe state: example 1

There are 12 resources in total available.

Process	Max. need	Acquisition	$T_0$	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$
$P_1$	10	5	5	5	10	-	-	-	-
$P_2$	4	2	4	-	-	-	-	-	-
$P_3$	9	2	2	2	2	2	9	-	-
<b>free</b>		3	1	5	0	10	3	12	

A **safe sequence** of processes exists  $\Rightarrow$  the system is in a **safe state** 😊

# Safe state: example 2

There are 12 resources in total available.

Process	Max. need	Acquisition	$T'_0$	$T_1$	$T_2$
$P_1$	10	5		5	
$P_2$	4	2		4	1
$P_3$	9	3		3	
free		2		0	4

Is the system in a safe state?


## Safe state: example 2


There are 12 resources in total available.

Process	Max. need	Acquisition	$T'_0$	$T_1$	$T_2$
$P_1$	10	5		5	5
$P_2$	4	2		4	-
$P_3$	9	3		3	3
<b>free</b>		2		0	4

No safe sequence of processes exists  $\Rightarrow$  the system is in an **unsafe state** 😞

# Questions?

All right?  $\Rightarrow$  

Question?  $\Rightarrow$   and use **chat**

or

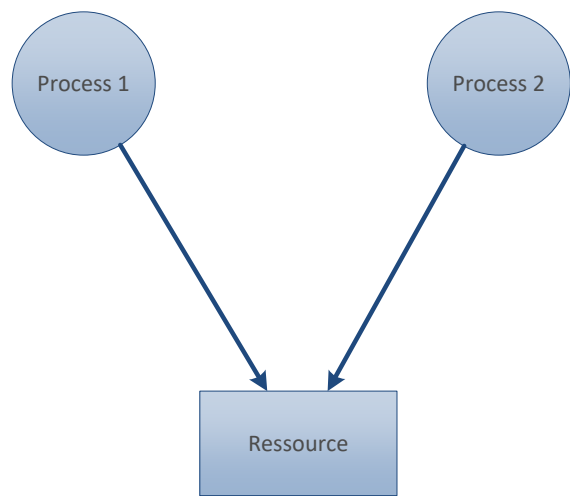
**speak** *after* I ask you to

# Deadlock prevention

We try to **further investigate the conditions** under these a deadlock can occur, in order to **avoid it as much as possible.**

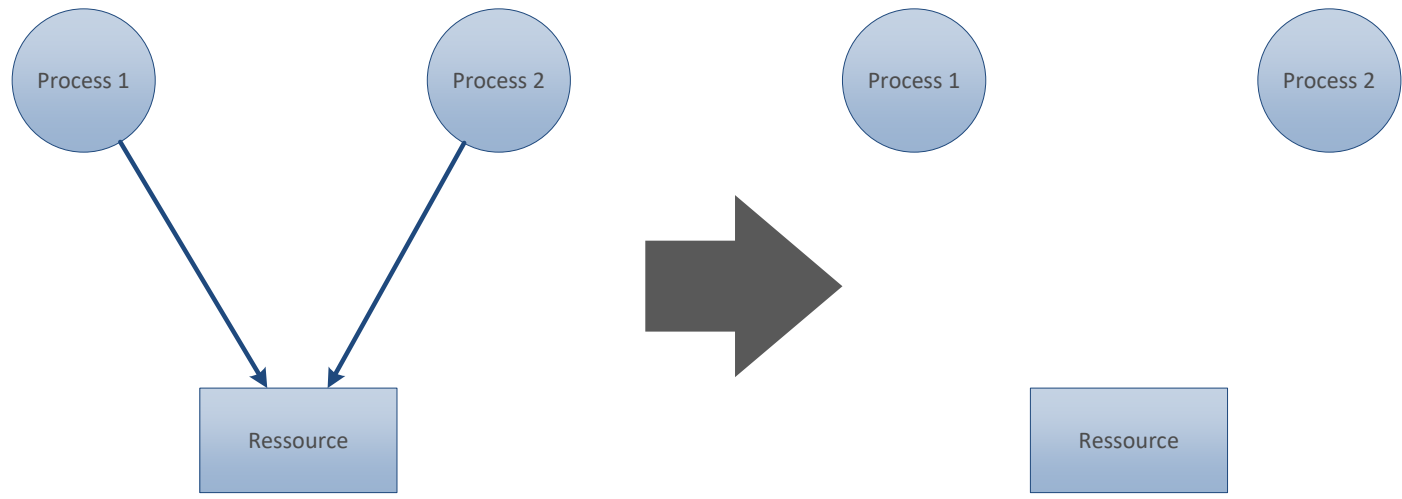
# Prevent mutual exclusion

Idea: try to avoid the mutual exclusion.



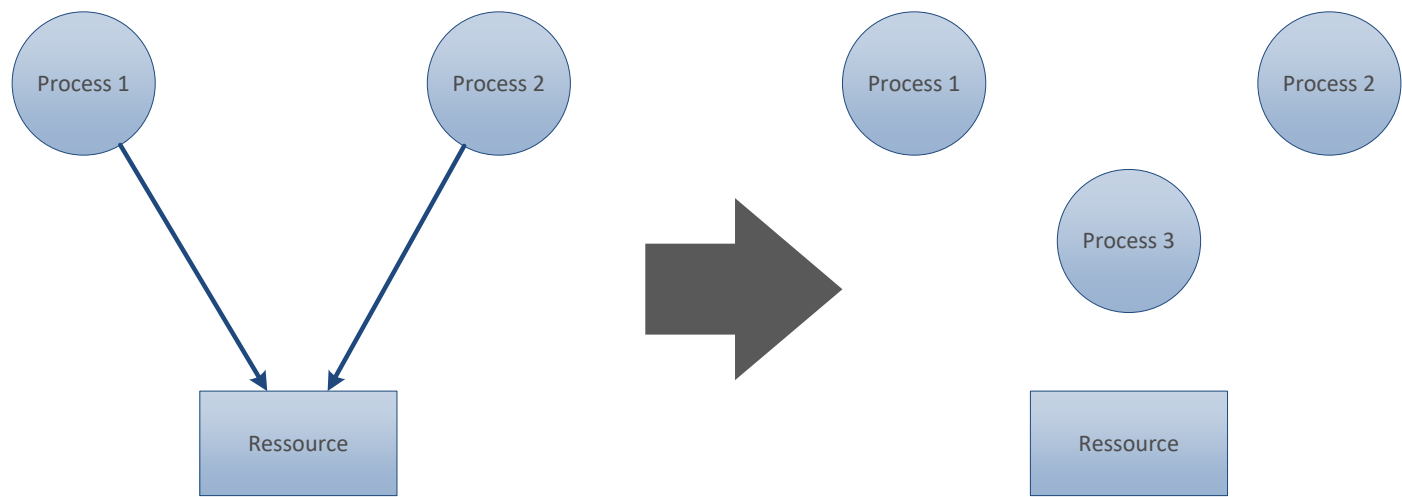
# Prevent mutual exclusion

Idea: try to avoid the mutual exclusion.



# Prevent mutual exclusion

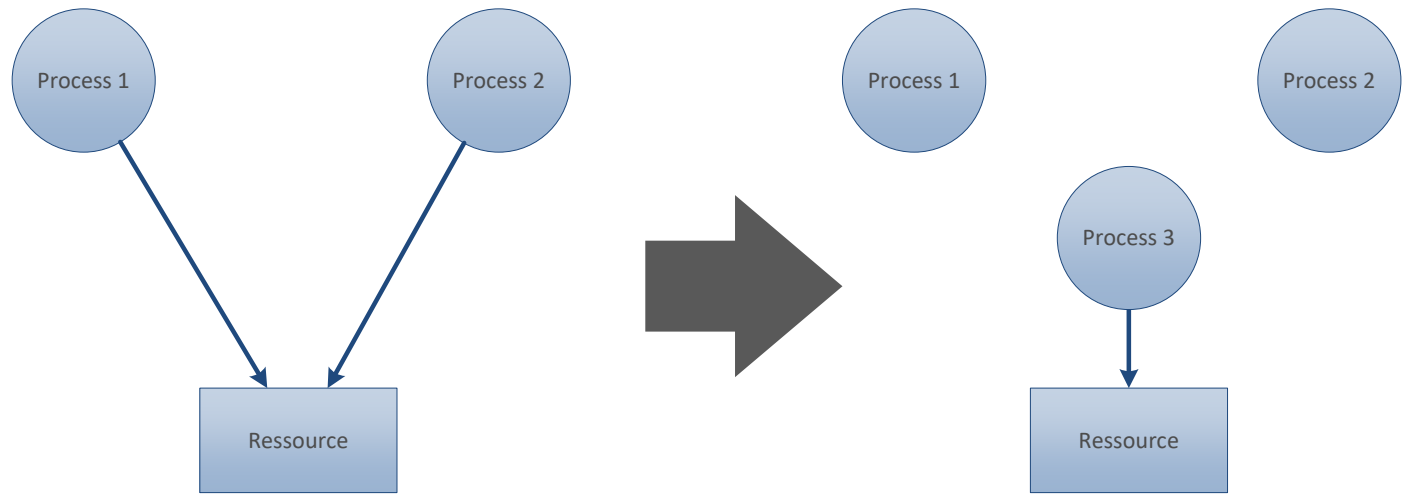
Idea: try to avoid the mutual exclusion.





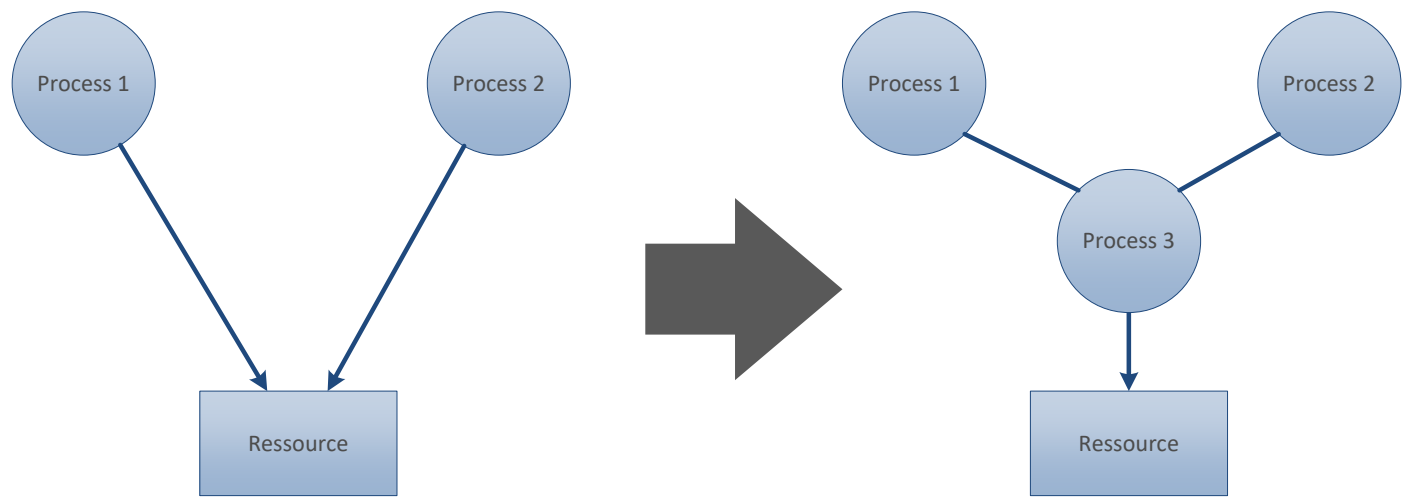
# Prevent mutual exclusion

Idea: try to avoid the mutual exclusion.



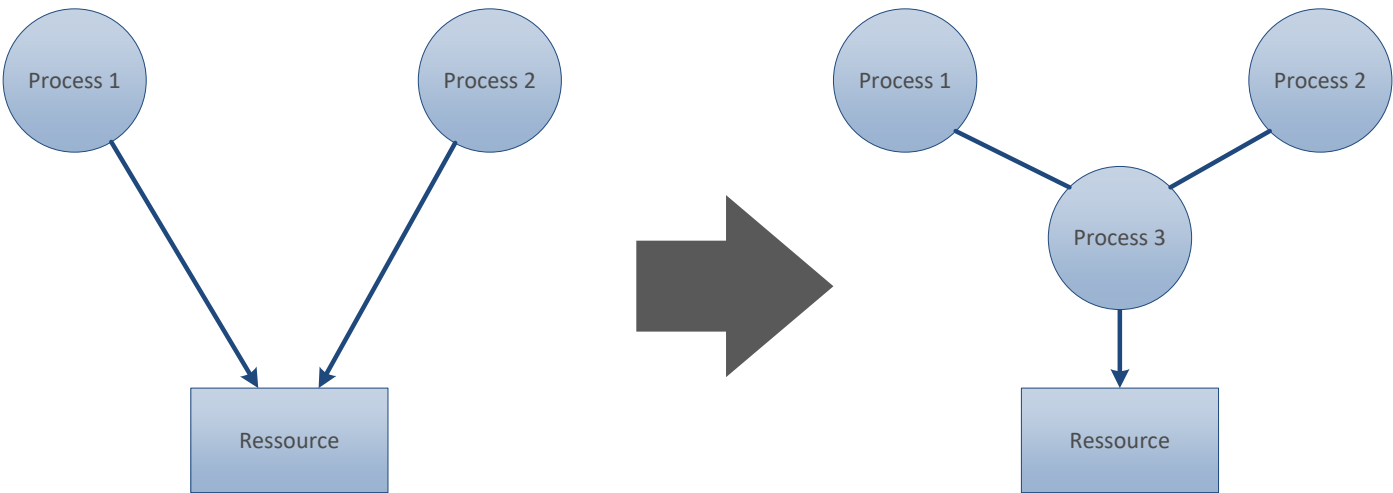
# Prevent mutual exclusion

Idea: try to avoid the mutual exclusion.



# Prevent mutual exclusion

Idea: try to avoid the mutual exclusion.

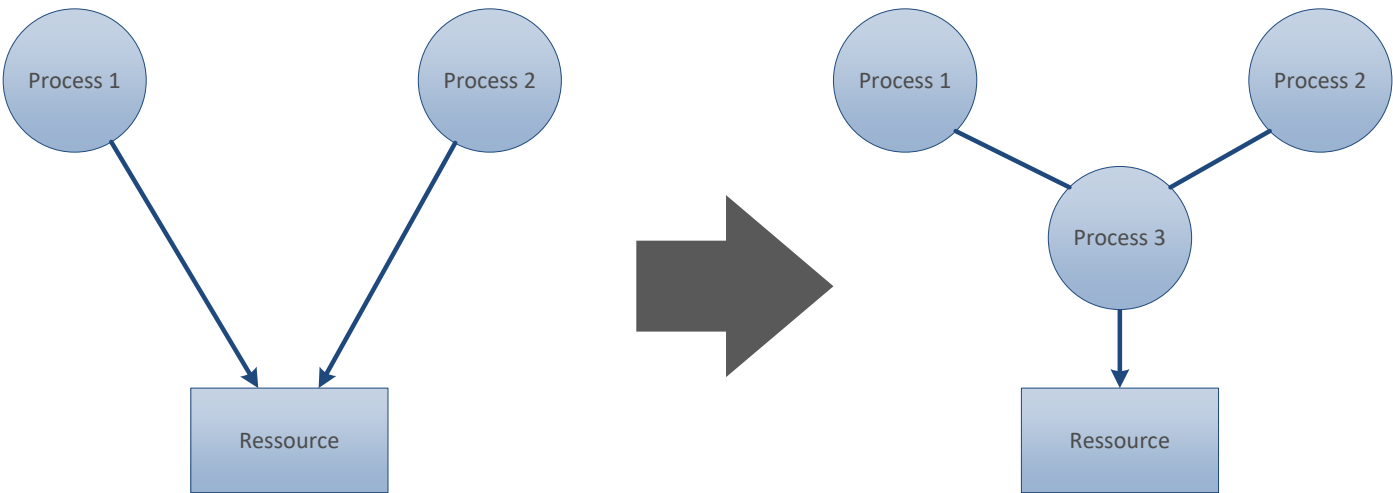


## Problems

- Not always possible
- Requires redesign of the application

# Prevent mutual exclusion

Idea: try to avoid the mutual exclusion.

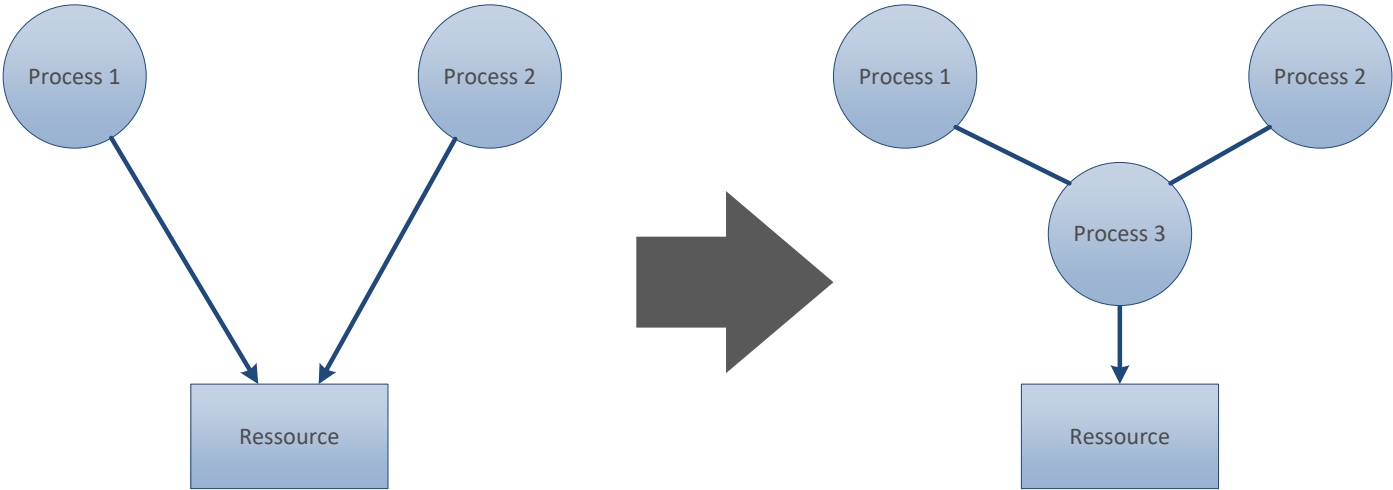


## Problems

- Not always possible
- Requires redesign of the application

# Prevent mutual exclusion

Idea: try to avoid the mutual exclusion.



## Problems

- Not always possible
- Requires redesign of the application

# Prevent hold-on-wait

**Idea: try to avoid the hold-on-wait.**

A process tries to requests all resources at startup. If even one resource is not available, all resources are released, and the process waits.

## Problems

- Often a process doesn't know at startup which resources it needs.
- This wastes resources, because they are acquired, but by no means used.
- It can happen that a process has to try it very often until it can acquire all resources.

# Prevent hold-on-wait

**Idea: try to avoid the hold-on-wait.**

A process tries to requests all resources at startup. If even one resource is not available, all resources are released, and the process waits.

## Problems

- Often a process doesn't know at startup which resources it needs.
- This wastes resources, because they are acquired, but by no means used.
- It can happen that a process has to try it very often until it can acquire all resources.

# Prevent hold-on-wait

**Idea: try to avoid the hold-on-wait.**

A process tries to requests all resources at startup. If even one resource is not available, all resources are released, and the process waits.

## Problems

- Often a process doesn't know at startup which resources it needs.
- This wastes resources, because they are acquired, but by no means used.
- It can happen that a process has to try it very often until it can acquire all resources.



# Prevent hold-on-wait

**Idea: try to avoid the hold-on-wait.**

A process tries to requests all resources at startup. If even one resource is not available, all resources are released, and the process waits.

## Problems

- Often a process doesn't know at startup which resources it needs.
- This wastes resources, because they are acquired, but by no means used.
- It can happen that a process has to try it very often until it can acquire all resources.

# Prevent hold-on-wait

**Idea: try to avoid the hold-on-wait.**

A process tries to requests all resources at startup. If even one resource is not available, all resources are released, and the process waits.

## Problems

- Often a process doesn't know at startup which resources it needs.
- This wastes resources, because they are acquired, but by no means used.
- It can happen that a process has to try it very often until it can acquire all resources.

# Prevent non-preemption

**Idea: try to avoid the non-preemption.**

Release all resources if a required resource isn't immediately available.

## Problems

- With the release of a resource, often the whole work is lost and has to be repeated.
- It can happen that a process has to release the resources very often until it can complete its work.

# Prevent non-preemption

**Idea: try to avoid the non-preemption.**

Release all resources if a required resource isn't immediately available.

## Problems

- With the release of a resource, often the whole work is lost and has to be repeated.
- It can happen that a process has to release the resources very often until it can complete its work.

# Prevent non-preemption

**Idea: try to avoid the non-preemption.**

Release all resources if a required resource isn't immediately available.

## Problems

- With the release of a resource, often the whole work is lost and has to be repeated.
- It can happen that a process has to release the resources very often until it can complete its work.

# Prevent non-preemption

**Idea: try to avoid the non-preemption.**

Release all resources if a required resource isn't immediately available.

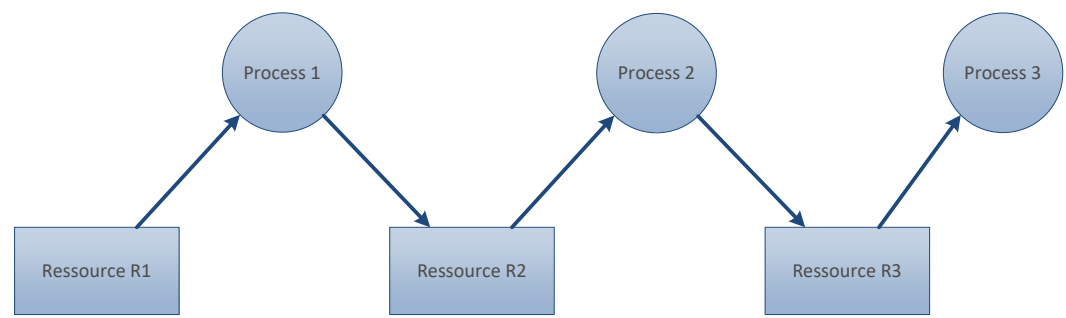
## Problems

- With the release of a resource, often the whole work is lost and has to be repeated.
- It can happen that a process has to release the resources very often until it can complete its work.

# Prevent circular wait

Idea: try to avoid circles.

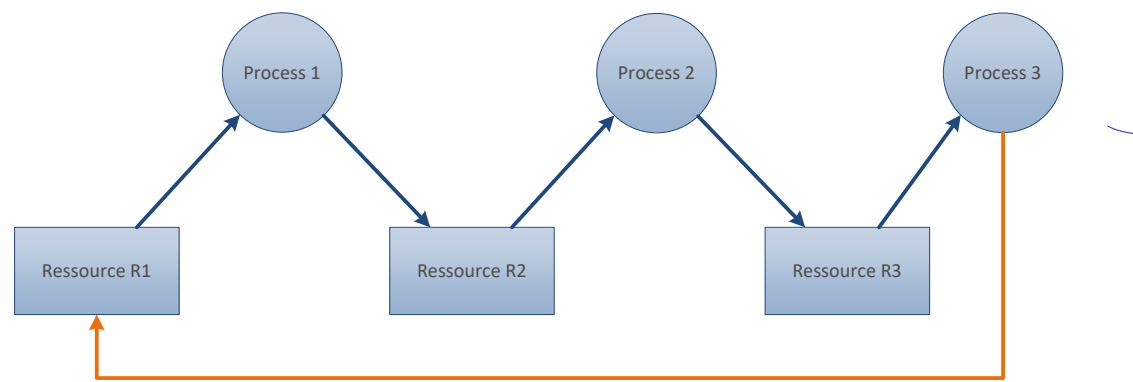
All resources are numbered. If a process has already acquired resource  $i$ , it can only acquire resource  $k$  (if  $k > i$ ).



# Prevent circular wait

Idea: try to avoid circles.

All resources are numbered. If a process has already acquired resource  $i$ , it can only acquire resource  $k$  (if  $k > i$ ).

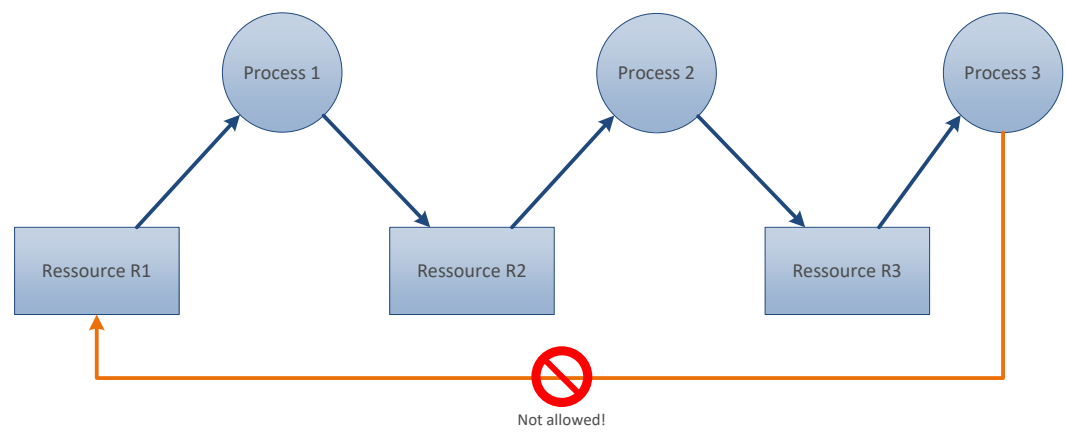




# Prevent circular wait

Idea: try to avoid circles.

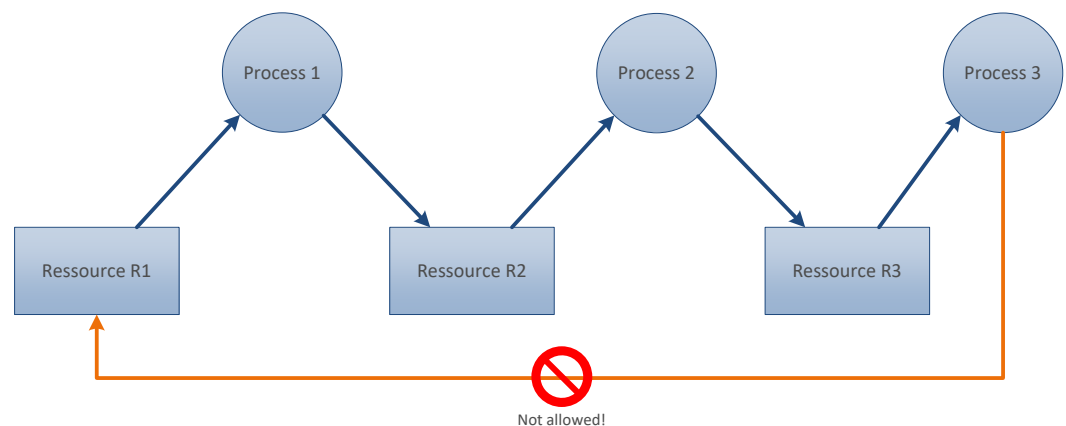
All resources are numbered. If a process has already acquired resource  $i$ , it can only acquire resource  $k$  (if  $k > i$ ).



# Prevent circular wait

Idea: try to avoid circles.

All resources are numbered. If a process has already acquired resource  $i$ , it can only acquire resource  $k$  (if  $k > i$ ).



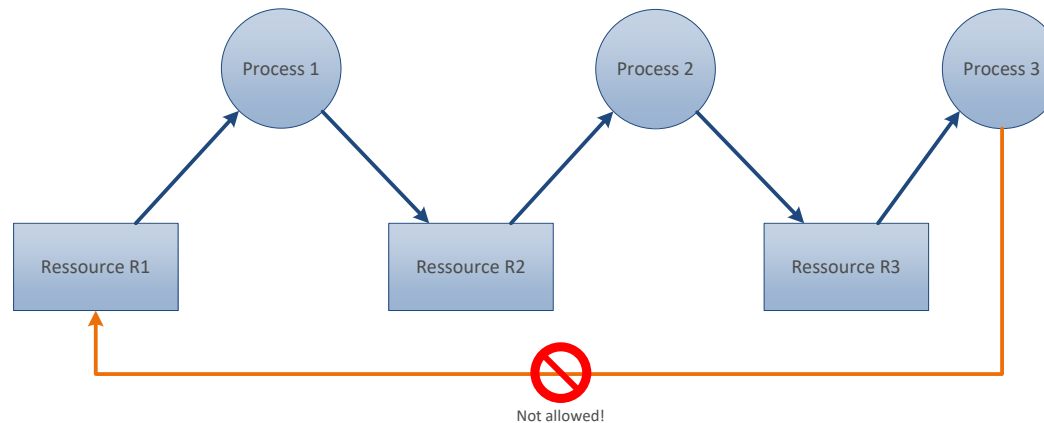
## Problems

- Finding an order in which everyone is satisfied is not easy and not always possible.
- Some resources are created and removed at runtime. Numbering not possible.
- But: With this it is possible to proof the **deadlock free** acquisition: **All processes have to acquire the resources in the same order.**

# Prevent circular wait

**Idea: try to avoid circles.**

All resources are numbered. If a process has already acquired resource  $i$ , it can only acquire resource  $k$  (if  $k > i$ ).



## Problems

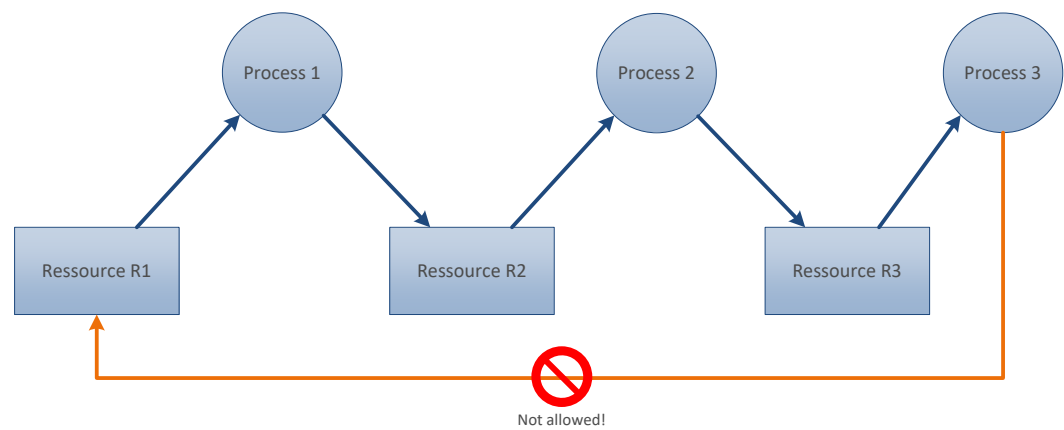
- Finding an order in which everyone is satisfied is not easy and not always possible.
- Some resources are created and removed at runtime. Numbering not possible.
- But: With this it is possible to prove the **deadlock free** acquisition: **All processes have to acquire the resources in the same order.**



# Prevent circular wait

**Idea: try to avoid circles.**

All resources are numbered. If a process has already acquired resource  $i$ , it can only acquire resource  $k$  (if  $k > i$ ).



## Problems

- Finding an order in which everyone is satisfied is not easy and not always possible.
- Some resources are created and removed at runtime. Numbering not possible.
- But: With this it is possible to proof the **deadlock free** acquisition: **All processes have to acquire the resources in the same order.**

# Deadlock prevention

## The banker's algorithm

- For every resource that is acquired by a process it checks if the system stays in a safe state.
  - True: The resource is given to the process.
  - False: The resource is not given to the process.

Cost:  $C = \text{num\_resources} * \text{num\_processes}^2$

# Deadlock prevention

## The banker's algorithm

- For every resource that is acquired by a process it checks if the system stays in a safe state.
  - True: The resource is given to the process.
  - False: The resource is not given to the process.

Cost:  $C = \text{num\_resources} * \text{num\_processes}^2$

# Deadlock prevention

## The banker's algorithm

- For every resource that is acquired by a process it checks if the system stays in a safe state.
  - **True:** The **resource is given** to the process.
  - **False:** The **resource is not given** to the process.

Cost:  $C = \text{num\_resources} * \text{num\_processes}^2$



# Deadlock prevention

## The banker's algorithm

- For every resource that is acquired by a process it checks if the system stays in a safe state.
  - **True:** The **resource is given** to the process.
  - **False:** The **resource is not given** to the process.

Cost:  $C = \text{num\_resources} * \text{num\_processes}^2$


# Deadlock prevention


## The banker's algorithm

- For every resource that is acquired by a process it checks if the system stays in a safe state.
  - **True:** The **resource is given** to the process.
  - **False:** The **resource is not given** to the process.

Cost:  $C = \text{num\_resources} * \text{num\_processes}^2$

# Questions?

All right?  $\Rightarrow$  

Question?  $\Rightarrow$   and use **chat**

or

**speak** *after* I  
ask you to



# Deadlock recovery

## Deadlock recovery strategies

- Terminate one or more processes involved in the deadlock
- Inform the system operator and give him instructions how to proceed (e.g. manual restart)
- Watchdog: automatically restart processes/device

source (compare): [https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7\\_Deadlocks.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html)

# Deadlock recovery

## Deadlock recovery strategies

- Terminate one or more processes involved in the deadlock
- Inform the system operator and give him instructions how to proceed (e.g. manual restart)
- Watchdog: automatically restart processes/device

source (compare): [https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7\\_Deadlocks.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html)

# Deadlock recovery

## Deadlock recovery strategies

- Terminate one or more processes involved in the deadlock
- Inform the system operator and give him instructions how to proceed (e.g. manual restart)
- Watchdog: automatically restart processes/device

source (compare): [https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7\\_Deadlocks.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html)

# Deadlock recovery


## Deadlock recovery strategies


- Terminate one or more processes involved in the deadlock
- Inform the system operator and give him instructions how to proceed (e.g. manual restart)
- Watchdog: automatically restart processes/device

source (compare): [https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7\\_Deadlocks.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html)



# Questions?

All right?  $\Rightarrow$  

Question?  $\Rightarrow$   and use **chat**

or

**speak** *after* I ask you to

# Summary and outlook

## Summary

- Deadlock intro
- Deadlock analysis
- Deadlock Safe state
- Deadlock prevention
- Deadlock recovery

## Outlook

- Scheduling theory
- Scheduling algorithms

# Summary and outlook

## Summary

- Deadlock intro
- Deadlock analysis
- Deadlock Safe state
- Deadlock prevention
- Deadlock recovery

## Outlook

- Scheduling theory
- Scheduling algorithms