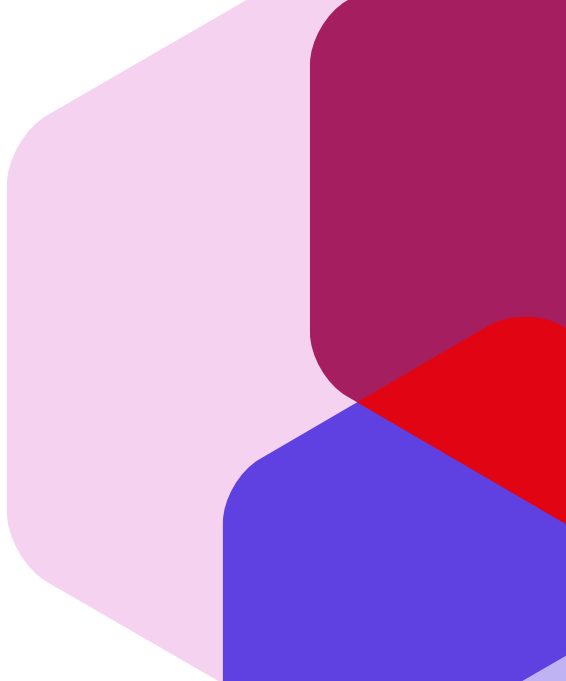
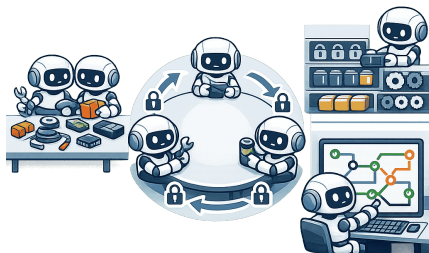


Principles, Algorithms and Models of Concurrency and Synchronization

05 Multiple Resources



- Problem introduction:
Multiple resources, Resource allocation,
Examples, Deadlock handling techniques,
Coffman conditions
- Deadlock Prevention:
Programmer avoids deadlocks
- Deadlock Avoidance:
System avoids deadlocks



Deadlock in the mutual exclusion problem:

- Deadlock: No process enters the critical section, even though at least one process wants to enter the critical section.
- This is equivalent to allocating **one** resource.

Now:

- We consider processes that want to allocate **multiple** such resources, where the individual resources are each protected by locks.
- We assume that there are only operations for requesting and releasing the resources.
- We ignore the exact implementation of the locks (these can be implemented using semaphores or monitors, for example).

Illustration of Resource Allocation

Process P_1 requests to allocate resource R_1 (but does not have it):



Process P_1 has allocated resource R_1 :



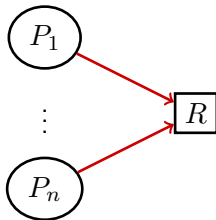
Deadlocks in general (with multiple resources)

Definition

A set of processes is **deadlocked** if **every** (unfinished) process in the set is waiting for an event that only **another** process in the set can cause.

The event usually corresponds to the release of a resource.

- Mutual exclusion problem: Deadlock if it is no longer possible for any process to enter the critical section, even though processes want to enter the critical section.



Examples (2)

- Two processes make transfers between account A and account B.
- Procedure:
 - Lock the first account.
 - Lock the second account.
 - Transfer funds from the first account to the second account.
 - Unlock the accounts.

- Process P
wait(AccountA);
wait(AccountB);
transfer from A to B
signal(AccountA);
signal(AccountB);
- Process Q
wait(AccountB);
wait(AccountA);
transfer from B to A
signal(AccountB);
signal(AccountA);



Examples (2)

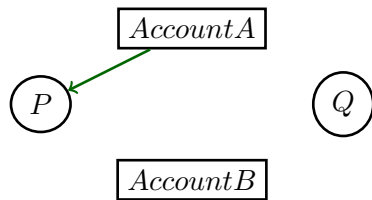
- Two processes make transfers between account A and account B.
- Procedure:
 - Lock the first account.
 - Lock the second account.
 - Transfer funds from the first account to the second account.
 - Unlock the accounts.

- Process P

```
wait(AccountA);  
wait(AccountB);  
transfer from A to B  
signal(AccountA);  
signal(AccountB);
```

- Process Q

```
wait(AccountB);  
wait(AccountA);  
transfer from B to A  
signal(AccountB);  
signal(AccountA);
```



Examples (2)

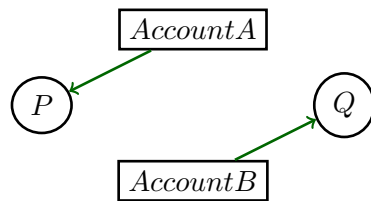
- Two processes make transfers between account A and account B.
- Procedure:
 - Lock the first account.
 - Lock the second account.
 - Transfer funds from the first account to the second account.
 - Unlock the accounts.

- Process P

wait(AccountA);
wait(AccountB);
transfer from A to B
signal(AccountA);
signal(AccountB);

- Process Q

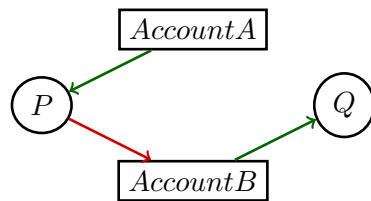
wait(AccountB);
wait(AccountA);
transfer from B to A
signal(AccountB);
signal(AccountA);



Examples (2)

- Two processes make transfers between account A and account B.
- Procedure:
 - Lock the first account.
 - Lock the second account.
 - Transfer funds from the first account to the second account.
 - Unlock the accounts.

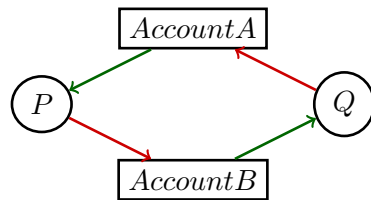
- | | |
|----------------------|----------------------|
| • <u>Process P</u> | <u>Process Q</u> |
| wait(AccountA); | wait(AccountB); |
| wait(AccountB); | wait(AccountA); |
| transfer from A to B | transfer from B to A |
| signal(AccountA); | signal(AccountB); |
| signal(AccountB); | signal(AccountA); |



Examples (2)

- Two processes make transfers between account A and account B.
- Procedure:
 - Lock the first account.
 - Lock the second account.
 - Transfer funds from the first account to the second account.
 - Unlock the accounts.

- Process P
wait(AccountA);
wait(AccountB);
transfer from A to B
signal(AccountA);
signal(AccountB);
- Process Q
wait(AccountB);
wait(AccountA);
transfer from B to A
signal(AccountB);
signal(AccountA);

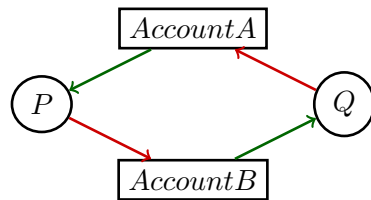


Examples (2)

- Two processes make transfers between account A and account B.
- Procedure:
 - Lock the first account.
 - Lock the second account.
 - Transfer funds from the first account to the second account.
 - Unlock the accounts.

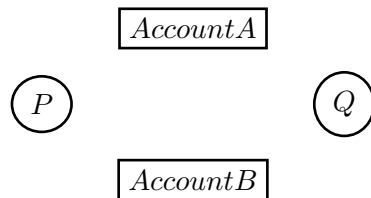
- | | |
|----------------------|----------------------|
| • <u>Process P</u> | <u>Process Q</u> |
| wait(AccountA); | wait(AccountB); |
| wait(AccountB); | wait(AccountA); |
| transfer from A to B | transfer from B to A |
| signal(AccountA); | signal(AccountB); |
| signal(AccountB); | signal(AccountA); |

Deadlock!



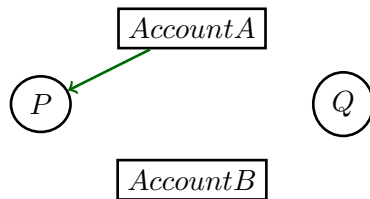
Examples (3)

- Working solution (deadlock-free)
- | <u>Process P</u> | <u>Process Q</u> |
|----------------------|----------------------|
| wait(AccountA); | wait(AccountA); |
| wait(AccountB); | wait(AccountB); |
| transfer from A to B | transfer from B to A |
| signal(AccountA); | signal(AccountA); |
| signal(AccountB); | signal(AccountB); |



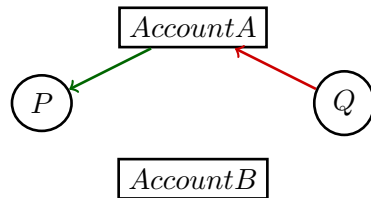
Examples (3)

- Working solution (deadlock-free)
- Process P
`wait(AccountA);`
`wait(AccountB);`
`transfer from A to B`
`signal(AccountA);`
`signal(AccountB);`
- Process Q
`wait(AccountA);`
`wait(AccountB);`
`transfer from B to A`
`signal(AccountA);`
`signal(AccountB);`



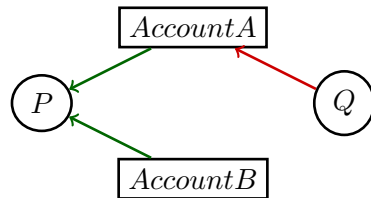
Examples (3)

- Working solution (deadlock-free)
- Process P
wait(AccountA);
wait(AccountB);
transfer from A to B
signal(AccountA);
signal(AccountB);
- Process Q
wait(AccountA);
wait(AccountB);
transfer from B to A
signal(AccountA);
signal(AccountB);



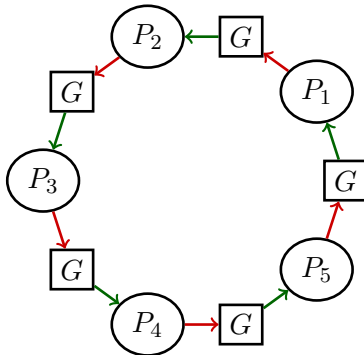
Examples (3)

- Working solution (deadlock-free)
- Process P
wait(AccountA);
wait(AccountB);
transfer from A to B
signal(AccountA);
signal(AccountB);
- Process Q
wait(AccountA);
wait(AccountB);
transfer from B to A
signal(AccountA);
signal(AccountB);



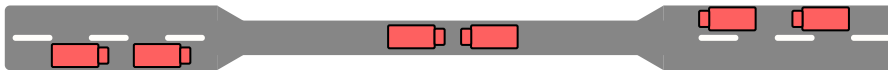
Examples (4)

- Similar problem: Dining philosophers
- Deadlock possible if all philosophers are unsynchronized
- All have the left fork, none has the right fork.

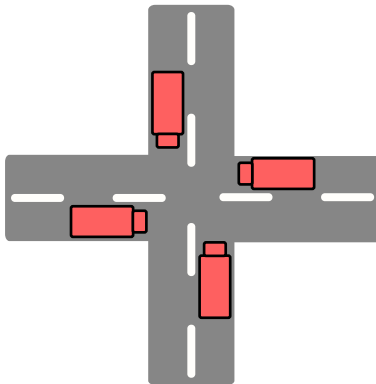


Examples (5)

- Bottleneck
- Only one vehicle can pass

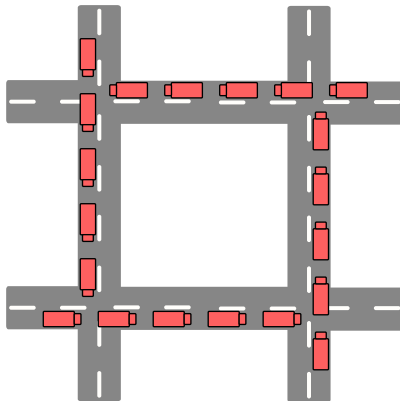


Examples (6)



- Everyone waits until the right lane is clear

Examples (7)



- Same problem, but slightly more complicated

Four approaches

- 1 **Ignore:** No precautions, hope that deadlocks occur only rarely.

Four approaches

- ① **Ignore:** No precautions, hope that deadlocks occur only rarely.
- ② **Deadlock detection and elimination:** Runtime system detects deadlocks and eliminates them. Problem: Find an algorithm that detects deadlocks.

Four approaches

- ① **Ignore:** No precautions, hope that deadlocks occur only rarely.
- ② **Deadlock detection and elimination:** Runtime system detects deadlocks and eliminates them. Problem: Find an algorithm that detects deadlocks.
- ③ **Deadlock avoidance:** Algorithm handles resources and does not allow situations that could lead to a deadlock.

Four approaches

- ① **Ignore:** No precautions, hope that deadlocks occur only rarely.
- ② **Deadlock detection and elimination:** Runtime system detects deadlocks and eliminates them. Problem: Find an algorithm that detects deadlocks.
- ③ **Deadlock avoidance:** Algorithm handles resources and does not allow situations that could lead to a deadlock.
- ④ **Deadlock prevention:** The programmer designs the programs so that deadlocks cannot occur.

Deadlock Handling (2)

- Obviously: Best method: Deadlock prevention
- To do this, you need to know:
[Under which circumstances can a deadlock occur?](#)
- We will consider conditions for deadlock and deadlock prevention.

Four Necessary Conditions for a Deadlock

the conditions are also called **Coffman conditions** (from Edward G. Coffman, Jr.)

- 1 **Mutual exclusion:** Only one process can access a resource at a time.



Four Necessary Conditions for a Deadlock

the conditions are also called **Coffman conditions** (from Edward G. Coffman, Jr.)

- ① **Mutual exclusion:** Only one process can access a resource at a time.
- ② **Hold and wait:** A process can request a resource (wait for a resource) while it already has allocated another resource.



Four Necessary Conditions for a Deadlock

the conditions are also called **Coffman conditions** (from Edward G. Coffman, Jr.)

- ① **Mutual exclusion:** Only one process can access a resource at a time.
- ② **Hold and wait:** A process can request a resource (wait for a resource) while it already has allocated another resource.
- ③ **No Preemption:** A resource can only be released by the allocating process.



Four Necessary Conditions for a Deadlock

the conditions are also called **Coffman conditions** (from Edward G. Coffman, Jr.)

- ① **Mutual exclusion:** Only one process can access a resource at a time.
- ② **Hold and wait:** A process can request a resource (wait for a resource) while it already has allocated another resource.
- ③ **No Preemption:** A resource can only be released by the allocating process.
- ④ **Circular Wait:** There is a cyclic dependency between waiting processes: Each process requests resource that is allocated by the next process in the cycle.

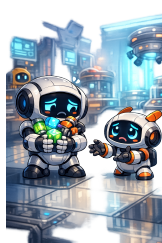


Illustration of the Four Conditions

❶ Mutual exclusion:

Only one outgoing (green) arrow per resource

Prohibited:

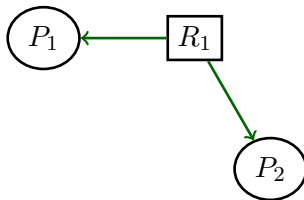


Illustration of the Four Conditions

- 1 **Mutual exclusion:**
Only one outgoing (green) arrow per resource
- 2 **Hold and wait:**
Red and green arrows possible for one process

Allowed:

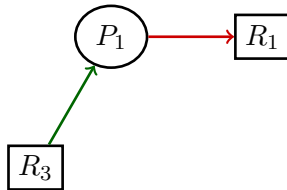


Illustration of the Four Conditions

❶ Mutual exclusion:

Only one outgoing (green) arrow per resource

❷ Hold and wait:

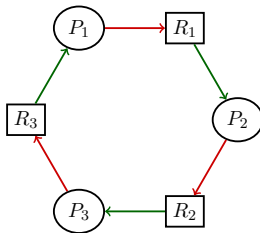
Red and green arrows possible for one process

❸ No Preemption:

Green arrows can only be changed by the process that has the arrow

Illustration of the Four Conditions

- ① **Mutual exclusion:**
Only one outgoing (green) arrow per resource
- ② **Hold and wait:**
Red and green arrows possible for one process
- ③ **No Preemption:**
Green arrows can only be changed by the process that has the arrow
- ④ **Circular Wait:** Cycle in the graph



Example

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

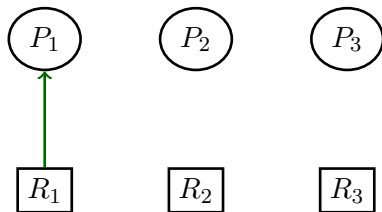
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

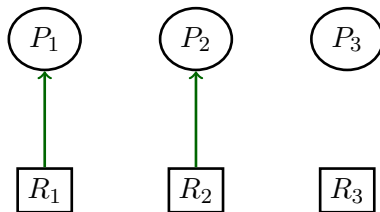
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

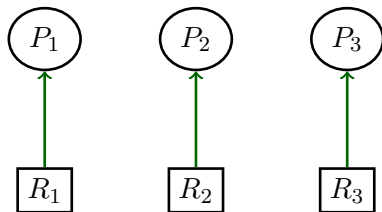
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

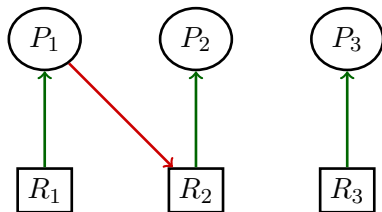
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

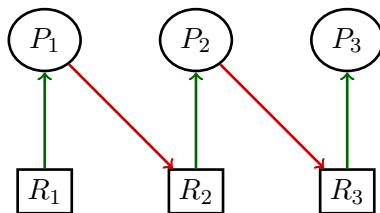
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

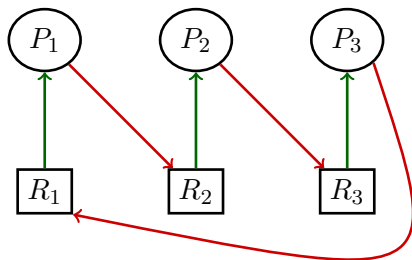
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

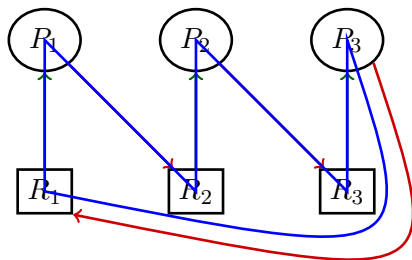
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

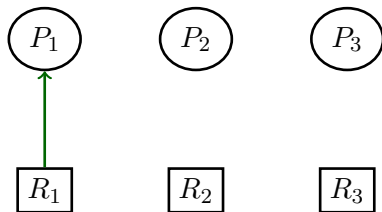
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

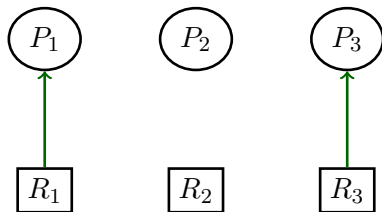
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

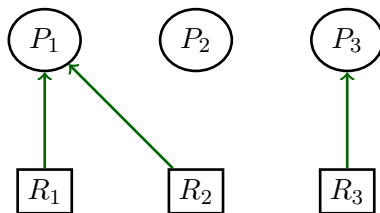
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

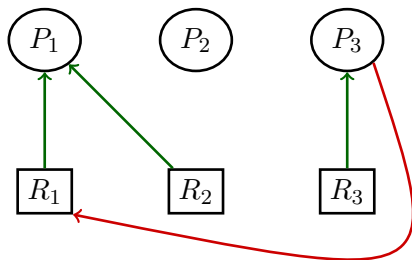
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

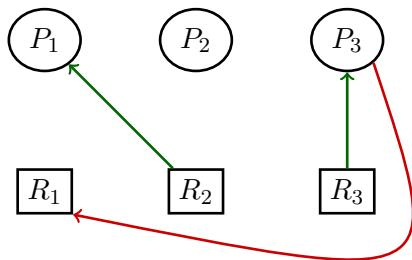
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

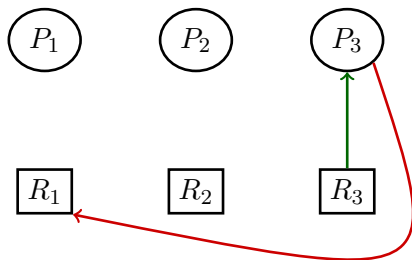
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

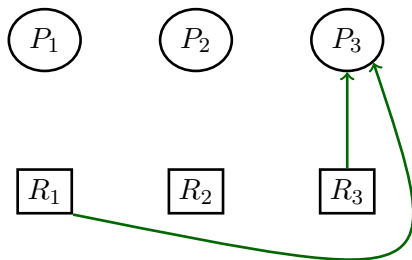
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

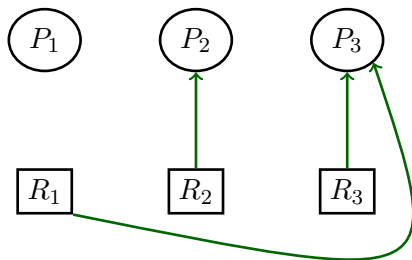
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

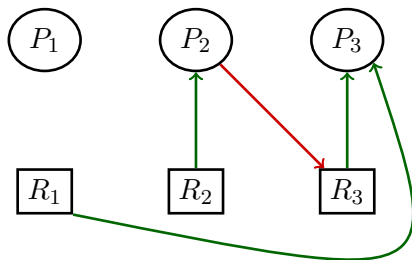
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

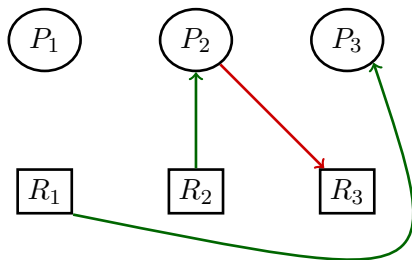
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

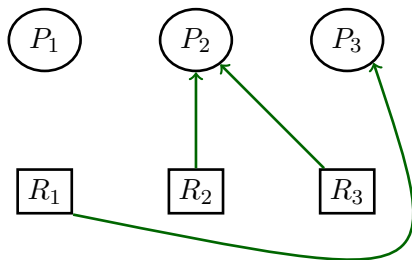
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

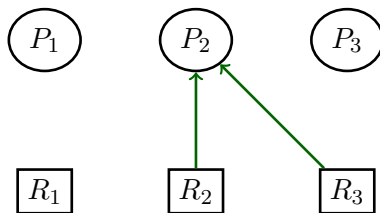
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

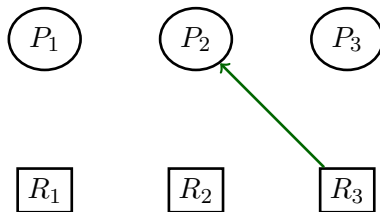
Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:

Request R_1

Request R_2

Release R_1

Release R_2

Process 2:

Request R_2

Request R_3

Release R_2

Release R_3

Process 3:

Request R_3

Request R_1

Release R_3

Release R_1



Example: Different Scheduling

Process 1:	Process 2:	Process 3:
Request R_1	Request R_2	Request R_3
Request R_2	Request R_3	Request R_1
Release R_1	Release R_2	Release R_3
Release R_2	Release R_3	Release R_1

Deadlock prevention: Program created in such a way that no deadlock occurs regardless of scheduling

Deadlock avoidance: Scheduler recognizes deadlock risk and excludes these schedules

Approach: Address one of the four conditions so that it never becomes true.

- Mutual exclusion: Generally difficult, but sometimes possible:
E.g., printer access is controlled by a spooler.
- No preemption: Difficult; for example, you cannot revoke access to the printer if the process has not finished printing, etc.

- Possibility: At the beginning, the process requests all the resources it needs.
- Philosophers: Exclusive access to all forks.
- 1st problem: Possibly too sequential.
- 2nd problem: Often unclear which resources each process needs.
- Variation of this solution: 2-phase locking protocol

2-Phase Locking Protocol

The processes work in two phases

Each process performs the following:

- **Phase 1:** The process attempts to allocate all required resources. If a required resource **is not available**, the process returns **all** allocated resources and starts again with phase 1.
- **Phase 2:** The process has allocated all the necessary resources. Once it has finished its computation, it releases all resources.



Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

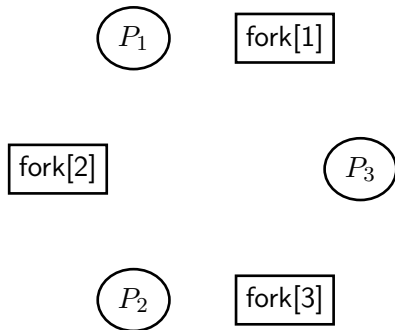
Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop

```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

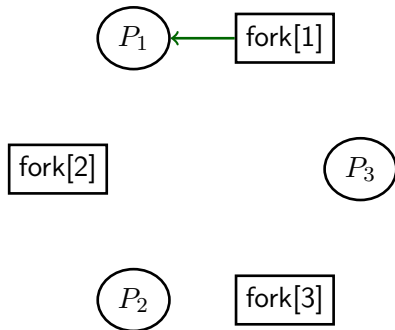
Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop

```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

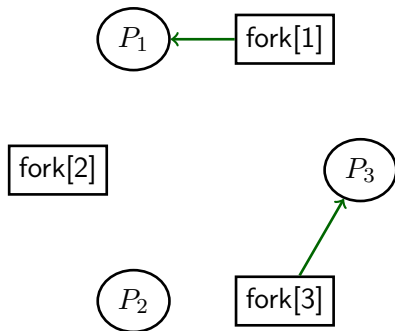
Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop

```



- Deadlock not possible!

Example: Dining Philosophers

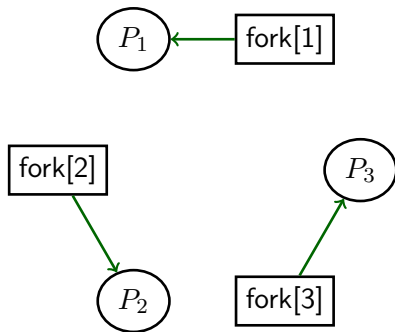
- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

Philosopher i

```

loop forever
(1)  l := tryWait(fork[i]);
(2)  if l then
(3)    r := tryWait(fork[i+1]);
(4)    if r then
(5)      Philosopher eats
(6)      signal(fork[i + 1]);
(7)      signal(fork[i]);
(8)    else signal(fork[i]);
(9)    Philosopher thinks;
end loop
  
```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

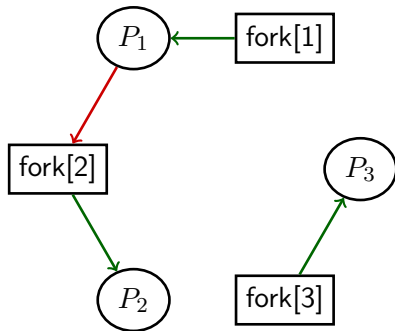
Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop

```



- Deadlock not possible!

Example: Dining Philosophers

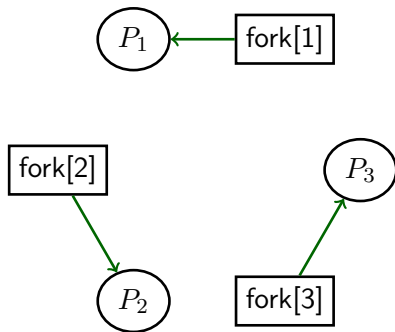
- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

Philosopher i

```

loop forever
(1)  l := tryWait(fork[i]);
(2)  if l then
(3)    r := tryWait(fork[i+1]);
(4)    if r then
(5)      Philosopher eats
(6)      signal(fork[i + 1]);
(7)      signal(fork[i]);
(8)    else signal(fork[i]);
(9)    Philosopher thinks;
end loop
  
```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

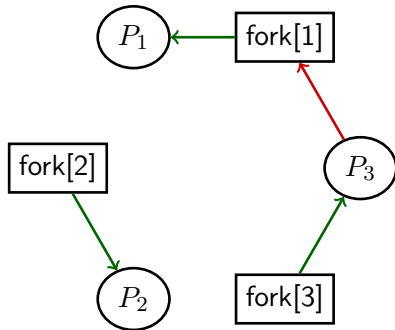
$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop
  
```



- Deadlock not possible!

Example: Dining Philosophers

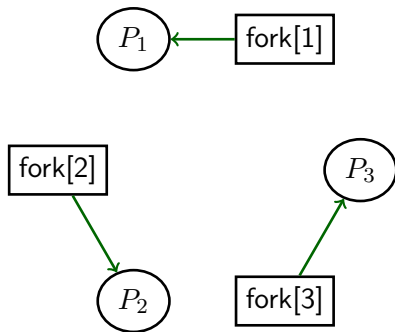
- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

Philosopher i

```

loop forever
(1)  l := tryWait(fork[i]);
(2)  if l then
(3)    r := tryWait(fork[i+1]);
(4)    if r then
(5)      Philosopher eats
(6)      signal(fork[i + 1]);
(7)      signal(fork[i]);
(8)    else signal(fork[i]);
(9)    Philosopher thinks;
end loop
  
```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

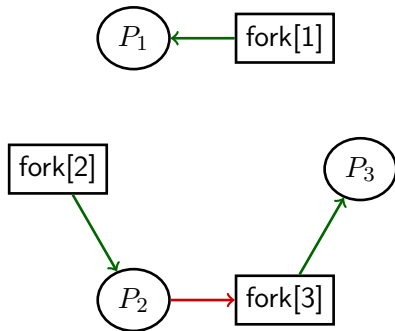
Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop

```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

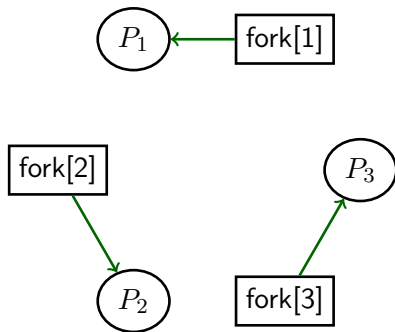
$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop
  
```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

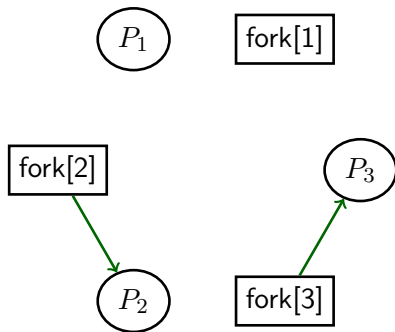
$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop
  
```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

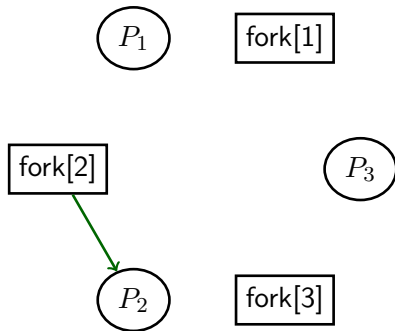
Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop

```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

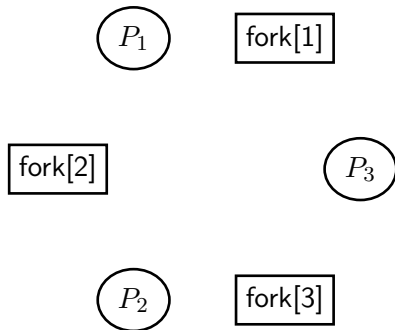
Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop

```



- Deadlock not possible!

Example: Dining Philosophers

- Initialize all forks (semaphores) with 1
- tryWait: Non-blocking implementation of wait:

$$\text{tryWait}(S) = \begin{cases} \text{True,} & \text{if semaphore could be allocated} \\ \text{False,} & \text{otherwise} \end{cases}$$
- Phase 1, Phase 2

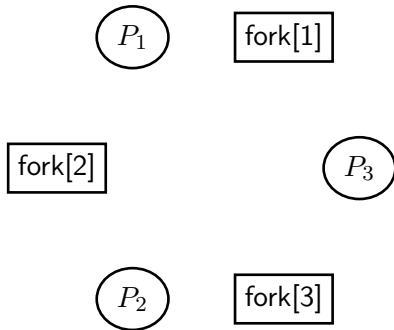
Philosopher i

loop forever

```

(1) l := tryWait(fork[i]);
(2) if l then
(3)   r := tryWait(fork[i+1]);
(4)   if r then
(5)     Philosopher eats
(6)     signal(fork[i + 1]);
(7)     signal(fork[i]);
(8)   else signal(fork[i]);
(9)   Philosopher thinks;
end loop

```



- Deadlock not possible!
- But **Livelock!**

Timestamp Ordering

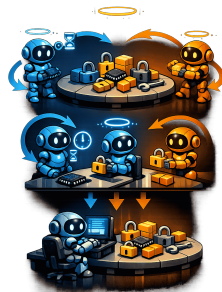
Two-phase locking protocol with timestamping

Processes get **unique timestamps** when they start.

Each process proceeds as follows:

- Phase 1: The process attempts to allocate all required resources. If a required resource is allocated by a process with a **smaller timestamp**, the process releases **all** resources and starts again. If **its own** timestamp is **smaller**, the process waits for the remaining resources.
- Phase 2: If the process entered this phase, it has all the resources it needs. It uses them and then releases them again.

Note: New timestamps are only assigned after successful completion of both phases.



- Deadlock-free
- Livelock not possible
- Starvation-free

Definition

Starvation is a situation in which a process is never (after any number of computation steps) able to allocate all the resources it needs.

- Attempt to prevent circular waiting.
- Philosophers' problem: The N th process picks up the right fork first.
- Then circular waiting cannot occur.
- This is because the forks are **totally ordered**: $1 < 2 < \dots < N$.
- And the philosophers allocate the resources according to this order.

In general:

Total-Order Theorem

If all shared resources are totally ordered and each process allocates its required resources in ascending order with respect to the total order, then a deadlock is impossible.

Proof: By contradiction. Assumption: There is a deadlock:

There are resources $R_1, \dots, R_n$ and processes $P_1, \dots, P_n$ with

- Process P_i has allocated resource R_{i-1}
- Process P_i is waiting for resource R_i

Let R_j be the smallest resource from $\{R_1, \dots, R_n\}$ with respect to the total order. Then process P_j is waiting for resource R_j , although it has already allocated resource R_{j-1} . Contradiction.

Total Order Theorem (2)

The following can be shown:

Lemma

A deadlock-free system in which the allocation of (all) *individual* resources is starvation-free is also starvation-free overall.

With the Total Order Theorem, we can conclude:

If there is a total order of resources, the resources are allocated according to this order, and the allocation of individual resources is starvation-free, then the overall system is starvation-free.

Reminder:

- **Deadlock avoidance:** Algorithm handles resources and does not allow situations that could lead to a deadlock.

In the following:

- Example by Dijkstra on deadlock avoidance:
- Problem of the **Deadly Embrace**
- Solution: Banker's algorithm

Assumption:

- There are m identical resources.
- Each process P_i requires a certain number $m_i \leq m$ of this resource.
- Processes request resources step-wise.
- The maximum number required m_i is known at the start.
- Once a process has reached its maximum number, it terminates and returns the resources.
- Problem: Implement a resource manager.

Dijkstra's Illustration

- Resource: money
- Processes are bank customers
- Resource manager: banker



Deadlock Avoidance: Example

- Available resources at the beginning: EUR 98
- 2 customers, both need EUR 50
- Both customers have already received EUR 48
- Both customers request EUR 1.
- Should the banker grant both requests?

Deadlock Avoidance: Example

- Available resources at the beginning: EUR 98
- 2 customers, both need EUR 50
- Both customers have already received EUR 48
- Both customers request EUR 1.
- Should the banker grant both requests?
- No: Then both customers have EUR 49 and the bank has EUR 0
- A deadlock has occurred.

Deadlock Avoidance: Example

- Available resources at the beginning: EUR 98
- 2 customers, both need EUR 50
- Both customers have already received EUR 48
- Both customers request EUR 1.
- Should the banker grant both requests?
- No: Then both customers have EUR 49 and the bank has EUR 0
- A deadlock has occurred.
- One customer requests EUR 1
- Should he get the money?

Deadlock Avoidance: Example

- Available resources at the beginning: EUR 98
- 2 customers, both need EUR 50
- Both customers have already received EUR 48
- Both customers request EUR 1.
- Should the banker grant both requests?
- No: Then both customers have EUR 49 and the bank has EUR 0
- A deadlock has occurred.
- One customer requests EUR 1
- Should he get the money?
- Yes, because after that the state is still **safe**
- **safe** = Deadlock still avoidable (does not have to occur)

Attempted Solution

Naive solution:

- Customers receive an ordering
- Banker always serves one customer until he has received his maximum amount
- All other customers must wait

Naive solution:

- Customers receive an ordering
- Banker always serves one customer until he has received his maximum amount
- All other customers must wait
- Bad, because it is a sequential algorithm

Naive solution:

- Customers receive an ordering
- Banker always serves one customer until he has received his maximum amount
- All other customers must wait
- Bad, because it is a sequential algorithm

Therefore:

- Additional requirement: Allow as much concurrency as possible
- Only reject a request if the state would then become unsafe, i.e., a deadlock must occur

- Assumption: Various resources, e.g., multiple currencies
- Vector $\vec{A}$: Current stock in the bank.
Each component of $\vec{A}$ is a non-negative integer and represents the stock of a currency

Set of customers (processes) $\mathcal{P}$. For $P \in \mathcal{P}$ let:

- $\vec{M}_P$ is the vector of the maximum resources that can be requested by process P
- $\vec{C}_P$ is the vector of the resources already allocated to process P

Banker's Algorithm: Safe State

safe = deadlock is avoidable in the future

A state (with all its vectors) is **safe**,
if there is a permutation π of the processes $P_1, \dots, P_n \in \mathcal{P}$,
such that for each process P_i according to the order of the permutation, there are
sufficient resources to fulfill the full request of P_i

Sufficient resources in this context means:

$$\vec{A} + \left(\sum_{\pi(j) < \pi(i)} \vec{C}_{P_{\pi(j)}} \right) - \vec{M}_{P_i} + \vec{C}_{P_i} \geq \vec{0}$$

To the currently available resources $\vec{A}$ add the resources that are in use by processes
that are ordered in the permutation **before** P_i may be added to the already allocated
resources, whose associated processes are fully served according to the permutation
before P_i .

Banker's Algorithm: Basic Structure

- Banker receives a resource request $\vec{L}_P$ from a process $P \in \mathcal{P}$.
- Consistency condition: $\vec{L}_P + \vec{C}_P \leq \vec{M}_P$
- Banker calculates the subsequent state, **as if P receives the resources**, i.e.

$$\begin{aligned}\vec{A} &:= \vec{A} - \vec{L}_P \\ \vec{C}_P &:= \vec{C}_P + \vec{L}_P\end{aligned}$$

- Then: Test whether the state is still safe.
- If unsafe, restore the original state and let P wait

If the customer receives the maximum resources, $\vec{A}$ is automatically adjusted.

Banker's Algorithm: Test for Safety

```
function testState( $\mathcal{P}$ ,  $\vec{A}$ ):  
  if  $\mathcal{P} = \emptyset$  then  
    return "safe"  
  else  
    if  $\exists P \in \mathcal{P}$  with  $\vec{M}_P - \vec{C}_P \leq \vec{A}$  then  
       $\vec{A} := \vec{A} + \vec{C}_P$ ;  
       $\mathcal{P} := \mathcal{P} \setminus \{P\}$ ;  
      testState( $\mathcal{P}$ ,  $\vec{A}$ )  
    else  
      return "unsafe"
```

- Algorithm calculates one of the sought permutations
- Runtime $O(|\mathcal{P}|^2)!$
- The criterion is exclusively “No deadlock”; no further “optimization” is applied
- Small improvement:
If $\vec{A} \geq \vec{M}_P - \vec{C}_P$ (sufficient resources available to completely serve P) then, after request $\vec{L}_P$, the state is always safe (test does not need to be performed)

Example

4 resources (EUR, USD, JYN, SFR)

4 processes A, B, C, D

Current state

Maximum values	Values received	Available resources
$\vec{M}_A = (4, 7, 1, 1)$	$\vec{C}_A = (1, 1, 0, 0)$	$\vec{A} = (2, 2, 3, 3)$
$\vec{M}_B = (0, 8, 1, 5)$	$\vec{C}_B = (0, 5, 0, 3)$	
$\vec{M}_C = (2, 2, 4, 2)$	$\vec{C}_C = (0, 2, 1, 0)$	
$\vec{M}_D = (2, 0, 0, 2)$	$\vec{C}_D = (1, 0, 0, 1)$	

Is the state safe?

Example (2)

We now consider the request $\vec{L}_A = (2, 2, 0, 0)$, i.e., process A wants to allocate two more EUR and two more USD.

After updating ($\vec{A} := \vec{A} - \vec{L}_A$ and $\vec{C}_A := \vec{C}_A + \vec{L}_A$) we obtain the state:

Maximum values	Values received	Available resources
$\vec{M}_A = (4, 7, 1, 1)$	$\vec{C}_A = (3, 3, 0, 0)$	$\vec{A} = (0, 0, 3, 3)$
$\vec{M}_B = (0, 8, 1, 5)$	$\vec{C}_B = (0, 5, 0, 3)$	
$\vec{M}_C = (2, 2, 4, 2)$	$\vec{C}_C = (0, 2, 1, 0)$	
$\vec{M}_D = (2, 0, 0, 2)$	$\vec{C}_D = (1, 0, 0, 1)$	

Does the state remain safe?