# 7. Deadlocks (1)

/home/esser/Daten/Dozent/Folien/bs-esser-21-english.odp

---

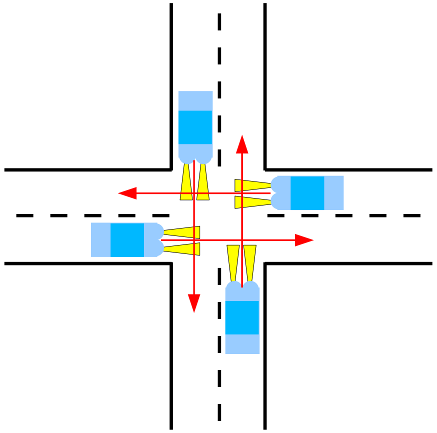
# What is a Deadlock?

- a set of processes / threads is in a **deadlock situation** when:
  - each process waits for a resource which is locked by another process of this set
  - none of the resources can be released because the locking process is blocked (waiting itself)
- processes in a deadlock situation will sit there forever
- deadlocks must be avoided!

---

# Deadlock: right before left (1)

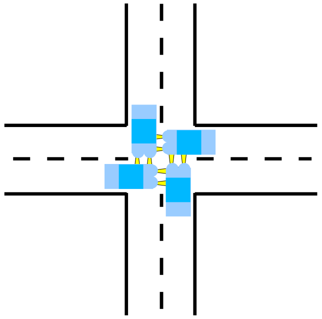
- classical example: right before left crossing



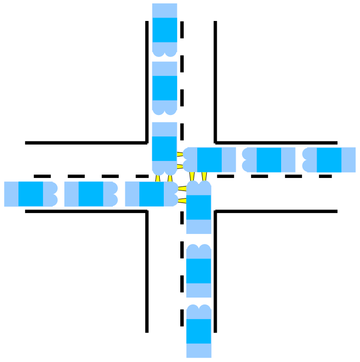
who can go first?
potential deadlock

---

# Deadlock: right before left (2)

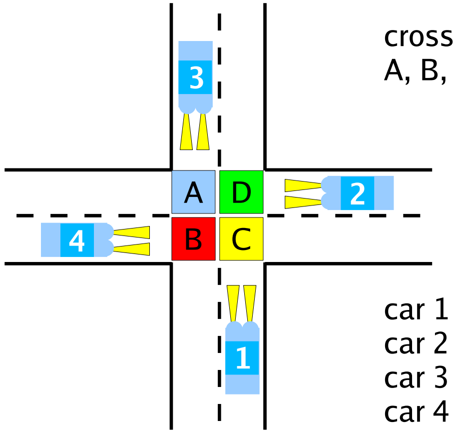


Deadlock, but recoverable: one or more cars can go backwards



Deadlock, not recoverable: participating cars cannot go back

# Deadlock: right before left (3)



**Analysis:**

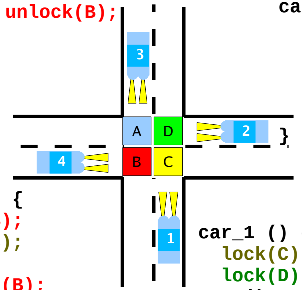
crossing consists of quadrants A, B, C, D

car 1 requires C, D
car 2 requires D, A
car 3 requires A, B
car 4 requires B, C

---

# Deadlock: right before left (4)

```
car_3 () {
  lock(A);
  lock(B);
  go();
  unlock(A);
  unlock(B);
}
```



```
car_2 () {
  lock(D);
  lock(A);
  go();
  unlock(D);
  unlock(A);
}
```

```
car_4 () {
  lock(B);
  lock(C);
  go();
  unlock(B);
  unlock(C);
}
```

```
car_1 () {
  lock(C);
  lock(D);
  go();
  unlock(C);
  unlock(D);
}
```

**Problematic order of execution:**

car1: lock(C)
car2: lock(D)
car3: lock(A)
car4: lock(B)
car1: lock(D) <- blocks
car2: lock(A) <- blocks
car3: lock(B) <- blocks
car4: lock(C) <- blocks

---

# Deadlock: smallest example (1)

- two locks A and B
  - e.g. A = scanner, B = printer, processes P, Q both want to create a xerox copy
- locking in differing orders

Process P

```
lock (A);
lock (B);

/* crit. area */

unlock (A);
unlock (B);
```

Process Q

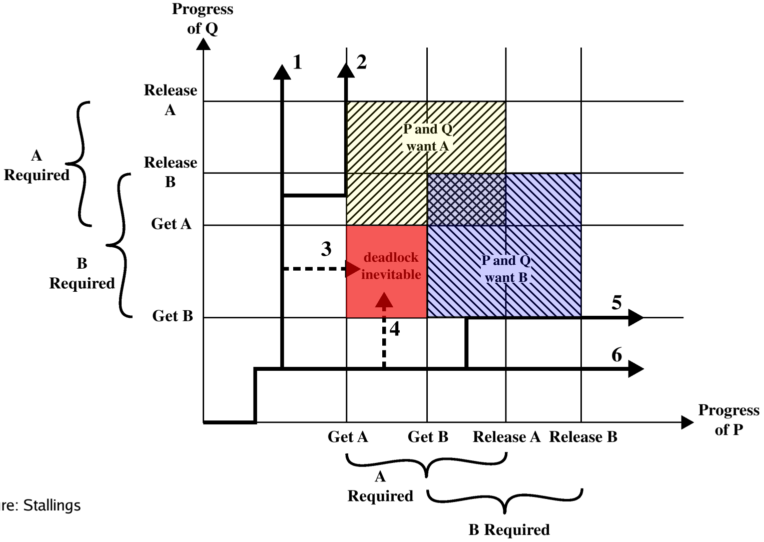
```
lock (B);
lock (A);

/* crit. area */

unlock (B);
unlock (A);
```

**Problematic order of execution:**

P: lock(A)
Q: lock(B)
P: lock(B) <- blocks
Q: lock(A) <- blocks

---

# Deadlock: smallest example (2)



Picture: Stallings

# Deadlock: smallest example (3)

- one possible solution:
  P does not need both locks simultaneously

Process P

```
lock (A);
/* crit. area */
unlock (A);

lock (B);
/* crit. area */
unlock (B);
```

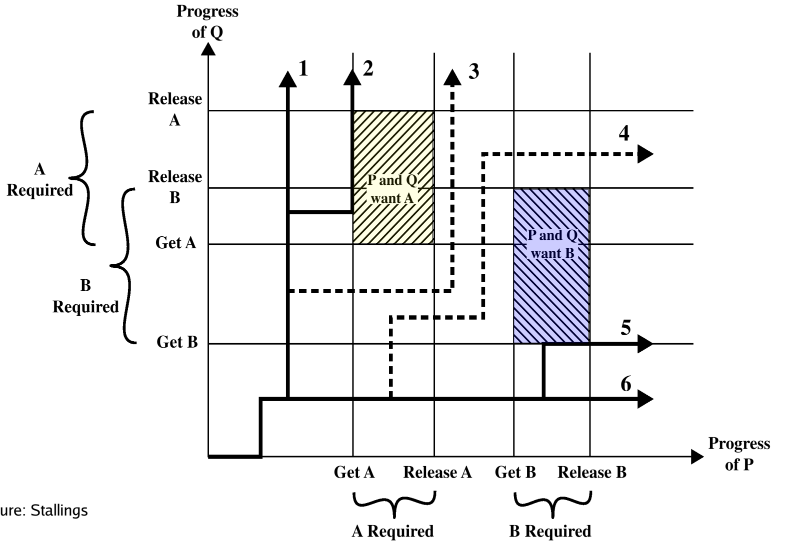
Process Q

```
lock (B);
lock (A);

/* crit. area */

unlock (B);
unlock (A);
```

- now deadlock is impossible

---

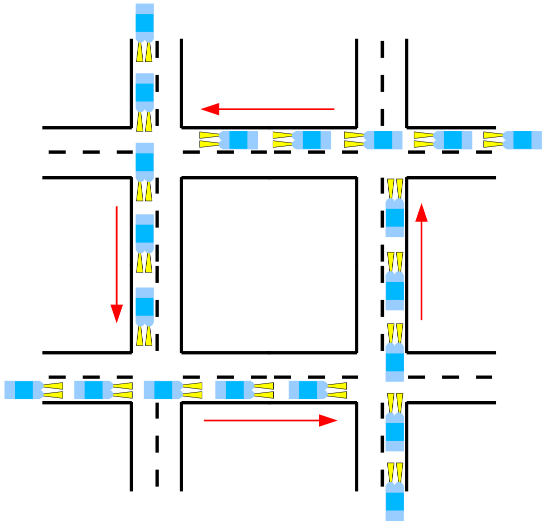
# Deadlock: smallest example (4)



Picture: Stallings

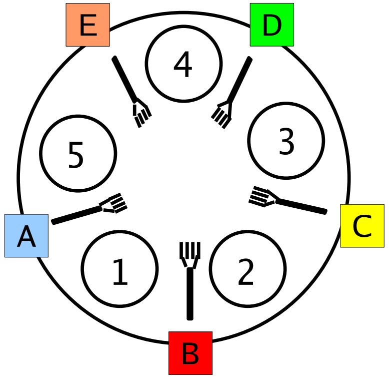
---

# Deadlock: Grid Lock



---

# Five Philosophers Problem



Philosopher 1 needs forks A, B
Philosopher 2 needs forks B, C
Philosopher 3 needs forks C, D
Philosopher 4 needs forks D, E
Philosopher 5 needs forks E, A

**Problematic order of execution:**
p1: lock (B)
p2: lock (C)
p3: lock (D)
p4: lock (E)
p5: lock (A)
p1: lock (A) <- blocks
p2: lock (B) <- blocks
p3: lock (C) <- blocks
p4: lock (D) <- blocks
p5: lock (E) <- blocks

# Contents of this chapter

- resource types
- sufficient and necessary deadlock conditions
- deadlock detection and removal
- deadlock avoidance: banker algorithm
- deadlock prevention

---

# Resource Types (1)

**two categories of resources:**
**preemptible / non-preemptible**

- preemptible resources
  - operating system can preempt such a resource and assign it to another process
  - examples:
    CPU (scheduler),
    main memory (memory management system)
  - these resources will not lead to deadlocks

---

# Resource Types (2)

- **non-preemptible resources**
  - operating system cannot preempt such a resource (without causing a program failure) – process must release it freely
  - examples:
    - DVD writer (preemption → destroyed DVD)
    - tape streamer (preemption → useless data on tape or cancellation of backup due to timeout)
- only the *non*-preemptible ones are of interest, because only they can cause deadlocks

---

# Resource Types (3)

- reusable vs. consumable resources
  - **reusable:** resource is accessed exclusively, but after releasing it, it can be reused by a different process (disk, RAM, CPU, ...)
  - **consumable:** created by one process and consumed by another process (messages, interrupts, signals, ...)

# Deadlock Conditions (1)

1. **mutual exclusion**
   - resource is exclusive: at any given moment, only one process can access the resource
2. **hold and wait**
   - a process already holds one or several resources,
   - and it can request further resources
3. **non-preemptiveness of resources**
   - the resource cannot be preempted (taken away from the process) by the operating system

---

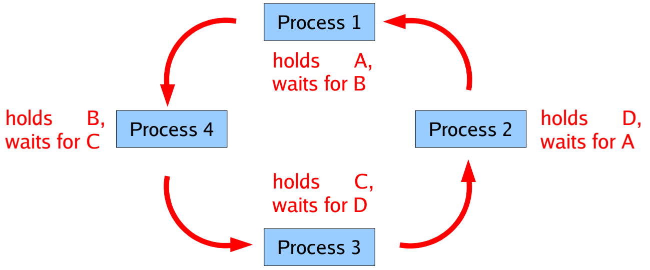
# Deadlock Conditions (2)

- (1) – (3) are **necessary** conditions for a deadlock
- (1) – (3) are also desirable properties of an operating system, because:
  - mutual exclusion is needed for proper synchronization
  - hold & wait is needed when a process simultaneously needs more than one resource exclusively
  - for some resources preemption makes no sense (e.g. DVD writer, streamer)

---

# Deadlock Conditions (3)

4. **cyclic wait**
   - processes can be ordered in a circle, where each process waits for a resource that is currently blocked by the next process in the circle



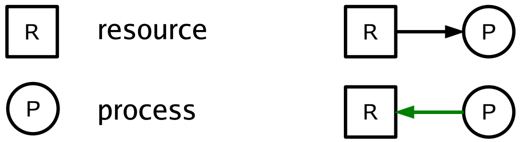
---

# Deadlock Conditions (4)

- (1) – (4) are **necessary and sufficient** conditions for a deadlock
- cyclic wait (4) (and its irresolvability) are consequences of (1) – (3)
- (4) is the most promising point of attack in order to avoid deadlocks

# Resource Allocation Graph (1)

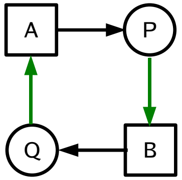
- show allocations and (not yet granted) requests graphically:



resource — R is allocated to P

process — P has requested R

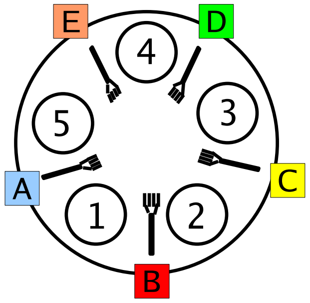
- P, Q from the minimal example:
- deadlock = circle in graph



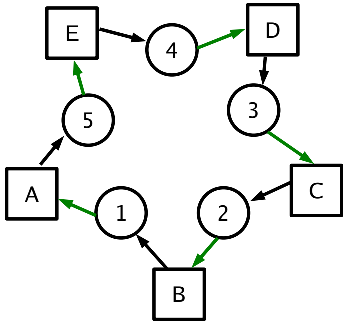
---

# Resource Allocation Graph (2)

Philosophers Example



Situation after all philosophers have taken their right forks

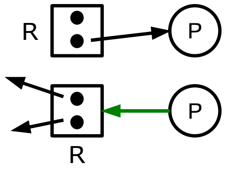


---

# Resource Allocation Graph (3)

- variant for resources which occur multiple times

resource (with two instances) — process



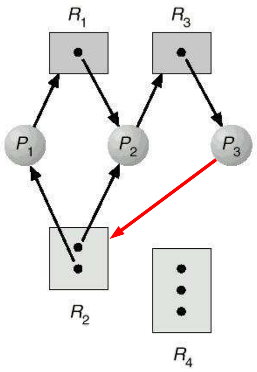
(one) R was allocated to P

P has requested (some) R

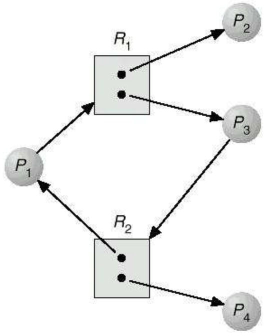
---

# Resource Allocation Graph (4)

- examples with several resource instances



with the red edge ($P_3$→$R_2$) there is a deadlock (without it there isn't)



circle, but no deadlock – the circle condition is only **necessary**, not sufficient!

images: Silberschatz/Galvin

# Deadlock Detection (1)

- idea: allow deadlocks to occur
- regularly check whether the system has run into a deadlock state – then remove any deadlock found
- uses three data structures:
  - allocation matrix
  - available vector
  - request matrix

---

# Deadlock Detection (2)

- *n* processes $P_1, \dots, P_n$
- *m* resource classes $R_1, \dots, R_m$
  of type $R_i$ there are $E_i$ resource instances ($i=1,\dots,m$)
  -> **resource vector** $E = (E_1\ E_2 \dots E_m)$
- **available vector** $A$ (how many are free?)
- **allocation matrix** $C$
  $C_{ij}$ = number of resources of type $j$, which are allocated to process $i$
- **request matrix** $R$
  $R_{ij}$ = number of resources of type $j$, which process $i$ needs (additionally to what it already has)

---

# Deadlock Detection (3)

example

$$\begin{array}{ccccc} & \text{ISDN adapter} & \text{streamer} & \text{scanner} & \text{DVD writer} \\ E = ( & 1 & 3 & 2 & 4 & ) \end{array}$$

resource vector

$$\begin{array}{ccccc} & \text{ISDN adapter} & \text{streamer} & \text{scanner} & \text{DVD writer} \\ A = ( & 0 & 0 & 1 & 2 & ) \end{array}$$

available vector

$$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 \end{pmatrix} \qquad R = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix} \quad \begin{array}{l} \text{Process 1} \\ \text{Process 2} \\ \text{Process 3} \end{array}$$

allocation matrix request matrix

---

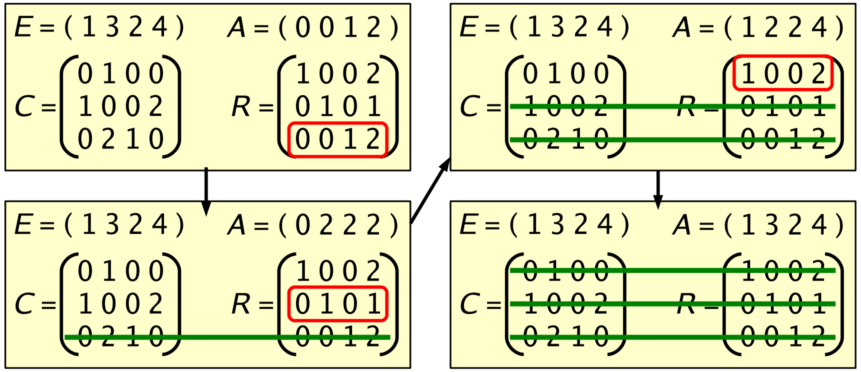
# Deadlock Detection (4)

**Algorithm**

1. find an unmarked process $P_i$, whose remaining request can be fully granted, i.e. $R_{ij} \leq A_j$ for all $j$
2. if there is no such process, terminate the algorithm
3. such a process could terminate successfully. Simulate that this process returns all its resources:
   $A := A + C_i$ (*i*-te Zeile von *C*)
   mark the process – it is not part of a deadlock
4. continue with step 1

# Deadlock Detection (5)

- all process that this algorithm leaves unmarked, are part of a deadlock
- example



$$E = (1\ 3\ 2\ 4) \quad A = (0\ 0\ 1\ 2) \quad C = \begin{pmatrix} 0\ 1\ 0\ 0 \\ 1\ 0\ 0\ 2 \\ 0\ 2\ 1\ 0 \end{pmatrix} \quad R = \begin{pmatrix} 1\ 0\ 0\ 2 \\ 0\ 1\ 0\ 1 \\ \boxed{0\ 0\ 1\ 2} \end{pmatrix}$$

$$E = (1\ 3\ 2\ 4) \quad A = (0\ 2\ 2\ 2) \quad C = \begin{pmatrix} 0\ 1\ 0\ 0 \\ 1\ 0\ 0\ 2 \\ \cancel{0\ 2\ 1\ 0} \end{pmatrix} \quad R = \begin{pmatrix} 1\ 0\ 0\ 2 \\ \boxed{0\ 1\ 0\ 1} \\ \cancel{0\ 0\ 1\ 2} \end{pmatrix}$$

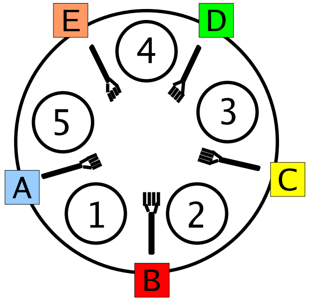
$$E = (1\ 3\ 2\ 4) \quad A = (1\ 2\ 2\ 4) \quad C = \begin{pmatrix} 0\ 1\ 0\ 0 \\ \cancel{1\ 0\ 0\ 2} \\ \cancel{0\ 2\ 1\ 0} \end{pmatrix} \quad R = \begin{pmatrix} \boxed{1\ 0\ 0\ 2} \\ \cancel{0\ 1\ 0\ 1} \\ \cancel{0\ 0\ 1\ 2} \end{pmatrix}$$

$$E = (1\ 3\ 2\ 4) \quad A = (1\ 3\ 2\ 4) \quad C = \begin{pmatrix} \cancel{0\ 1\ 0\ 0} \\ \cancel{1\ 0\ 0\ 2} \\ \cancel{0\ 2\ 1\ 0} \end{pmatrix} \quad R = \begin{pmatrix} \cancel{1\ 0\ 0\ 2} \\ \cancel{0\ 1\ 0\ 1} \\ \cancel{0\ 0\ 1\ 2} \end{pmatrix}$$

---

# Deadlock Detection (6)

- example: five philosophers



$$E = \begin{array}{c} A\ B\ C\ D\ E \\ (1\ 1\ 1\ 1\ 1) \end{array} \quad A = \begin{array}{c} A\ B\ C\ D\ E \\ (0\ 0\ 0\ 0\ 0) \end{array}$$

$$C = \begin{pmatrix} 1\ 0\ 0\ 0\ 0 \\ 0\ 1\ 0\ 0\ 0 \\ 0\ 0\ 1\ 0\ 0 \\ 0\ 0\ 0\ 1\ 0 \\ 0\ 0\ 0\ 0\ 1 \end{pmatrix} \quad R = \begin{pmatrix} 0\ 0\ 0\ 0\ 1 \\ 1\ 0\ 0\ 0\ 0 \\ 0\ 1\ 0\ 0\ 0 \\ 0\ 0\ 1\ 0\ 0 \\ 0\ 0\ 0\ 1\ 0 \end{pmatrix}$$

- algorithm terminates at once
- all processes are part of a deadlock

---

# Deadlock Detection (7): Recovery

**when to do when a deadlock was detected?**

- **preemption** of a resource?
  in the cases which we look at, this is impossible (non-preemptible resources)
- **abortion** of a process that is part of the deadlock
- **resetting** a process to an earlier process state in which it did not hold the resource
  - requires regularly saving the process states