Operating Systems

Timo Hönig
Bochum Operating Systems and System Software (BOSS)
Ruhr University Bochum (RUB)

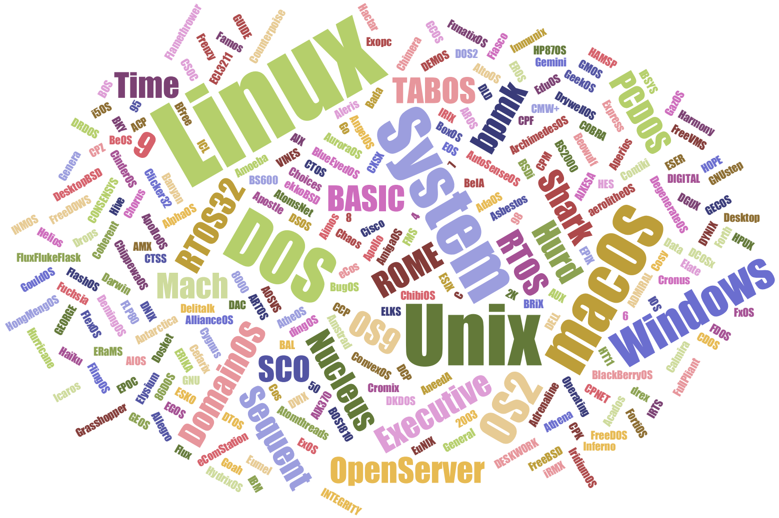
V. Deadlocks
May 10, 2023 (Summer Term 2023)

RUB Faculty of Computer Science

RUHR UNIVERSITÄT BOCHUM RUB

www.informatik.rub.de
Chair of Operating Systems and System Software

Agenda



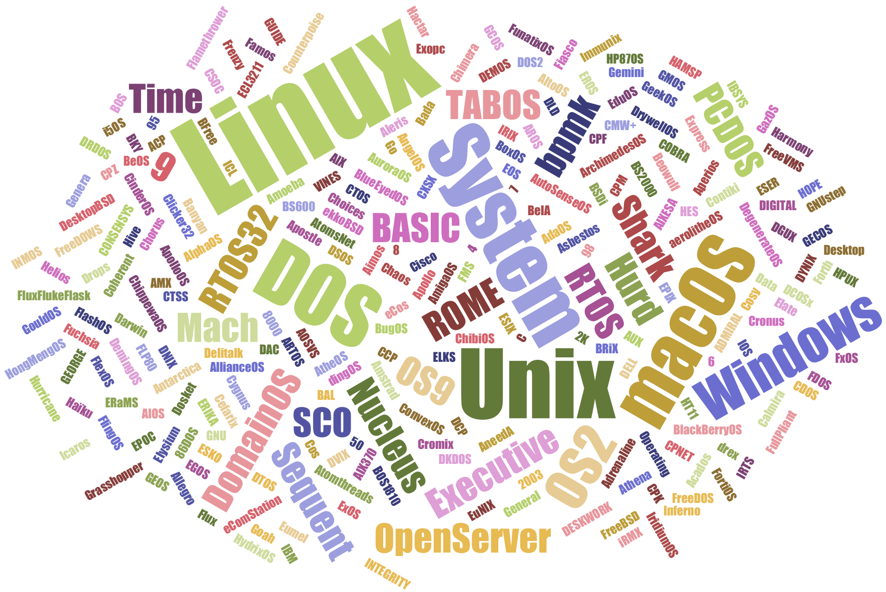
- Recap
- Organizational Matters
- Problem Scenario and Cause Study
- Process Deadlocks
  - Definition, Variants, and Conditions
  - Consumable and Non-consumable Resources
  - Resource Allocation Graphs
- Classic Deadlock Situation
  - The Dining Philosophers Problem
- Deadlock Counter Measures
  - Prevention, Avoidance, Detection and Recovery
- Summary and Outlook

Literature References

Silberschatz, Chapter 8

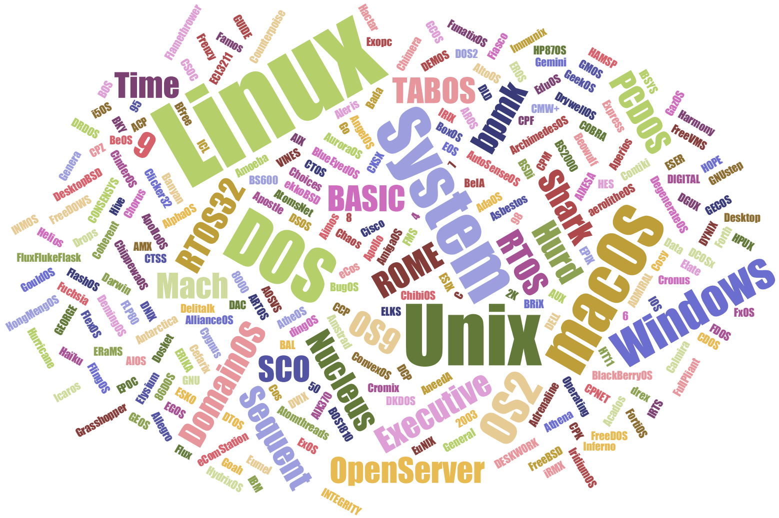
Tanenbaum, Chapter 6

⏮ Recap



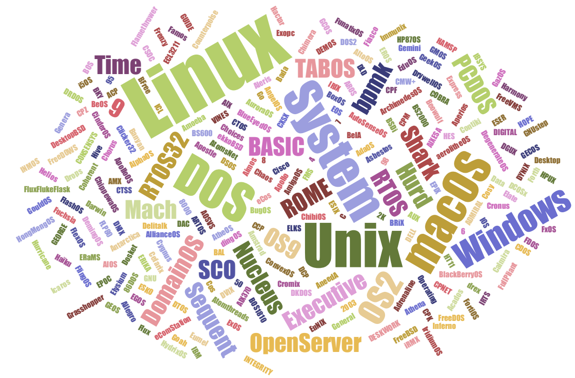
- **process synchronisation**
  - **critical sections, race conditions**
  - coordination methods to **avoid errors** due to uncontrolled, **concurrent data access**
- ad-hoc methods using **busy waiting**
  - **inefficient** solutions, dangerous illusion of correctness
  - **waste of resource** (i.e., CPU cycles)
- advanced methods to ensure mutual exclusion: **passive waiting**
  - exploit hardware support: **atomic operations**
  - operating system support: **semaphores** to implement **passive waiting**

Agenda



- Recap
- **Organizational Matters**
- Problem Scenario and Cause Study
- Process Deadlocks
  - Definition, Variants, and Conditions
  - Consumable and Non-consumable Resources
  - Resource Allocation Graphs
- Classic Deadlock Situation
  - The Dining Philosophers Problem
- Deadlock Counter Measures
  - Prevention, Avoidance, Detection and Recovery
- Summary and Outlook

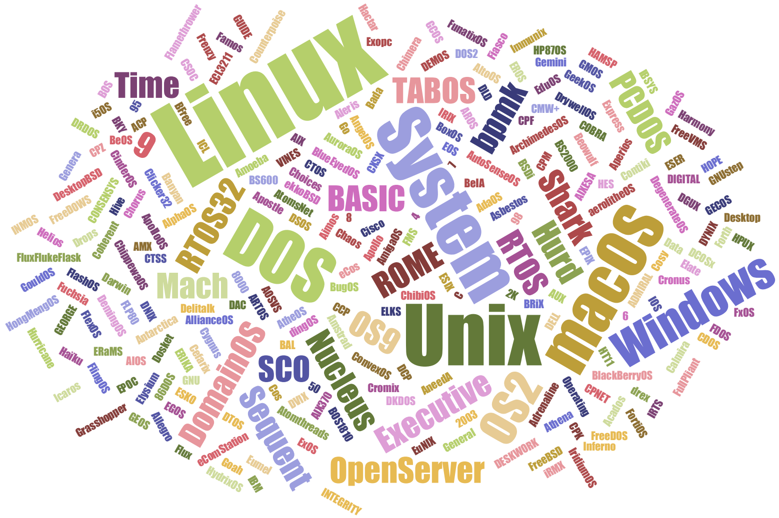
Organizational Matters



- lecture
  - Wednesday, 10:15 – 11:45
  - format: synchronous, **hybrid**
    - → in presence (Room HID, Building ID)
    - → online lecture (Zoom)
  - **exam:** August 7, 2023 (first appointment)
    September 25, 2023 (retest appointment)
- exercises: **group allocation ~~almost~~ complete**

  - make use of group work - for your own benefit!
  - remarks on Wednesday for Übung T03
- manage course material, asynchronous communication: Moodle
- `https://moodle.ruhr-uni-bochum.de/course/view.php?id=50698`

Agenda



- Recap
- Organizational Matters
- **Problem Scenario and Cause Study**
- Process Deadlocks
  - Definition, Variants, and Conditions
  - Consumable and Non-consumable Resources
  - Resource Allocation Graphs
- Classic Deadlock Situation
  - The Dining Philosophers Problem
- Deadlock Counter Measures
  - Prevention, Avoidance, Detection and Recovery
- Summary and Outlook

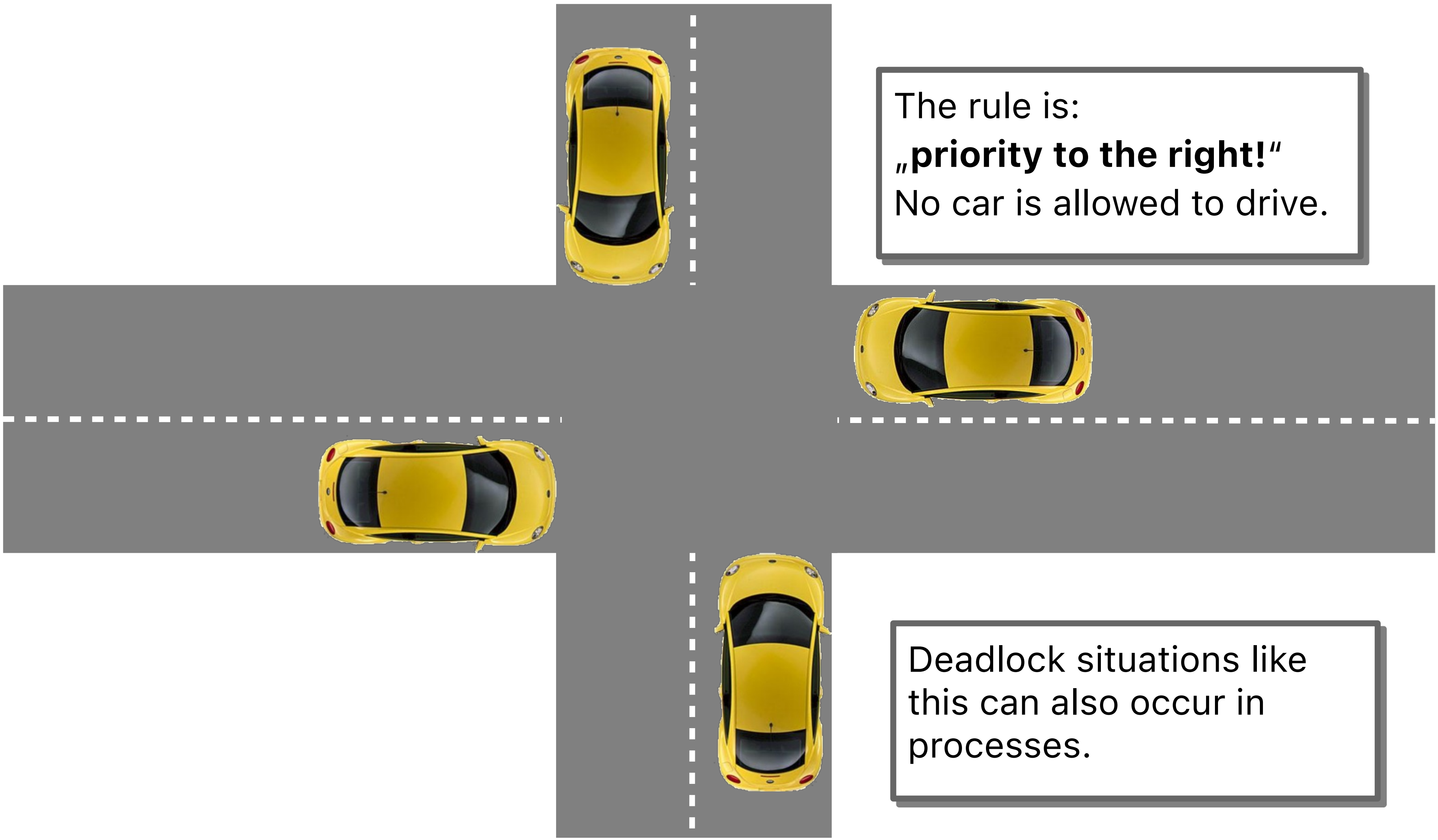
Problem Scenario and Cause Study

- **processes operate concurrently** (cooperation and competition)
  - increased utilization of shared resources (e.g., CPU)
- mandatory: **synchronisation** primitives are used to coordinate processes
  - data races induce the need of synchronisation methods
- concept: **passive waiting** by blocking and waking processes
  - resource usage and availability decides on process states
  - eliminates **busy waiting**, which is considered an anti-pattern (too expensive, error prone)

- solution with OS support: **semaphores** → implement **passive waiting** and enable:
  - unilateral synchronisation (→ e.g., producer/consumer)
  - multilateral synchronisation
    - interlaced producer/consumer, "prosumer"
    - mutual exclusion (e.g., exclusive access to a critical section)

- **waiting mechanisms** can lead to **deadlock** situations...

Deadlocks (dt. Verklemmungen)

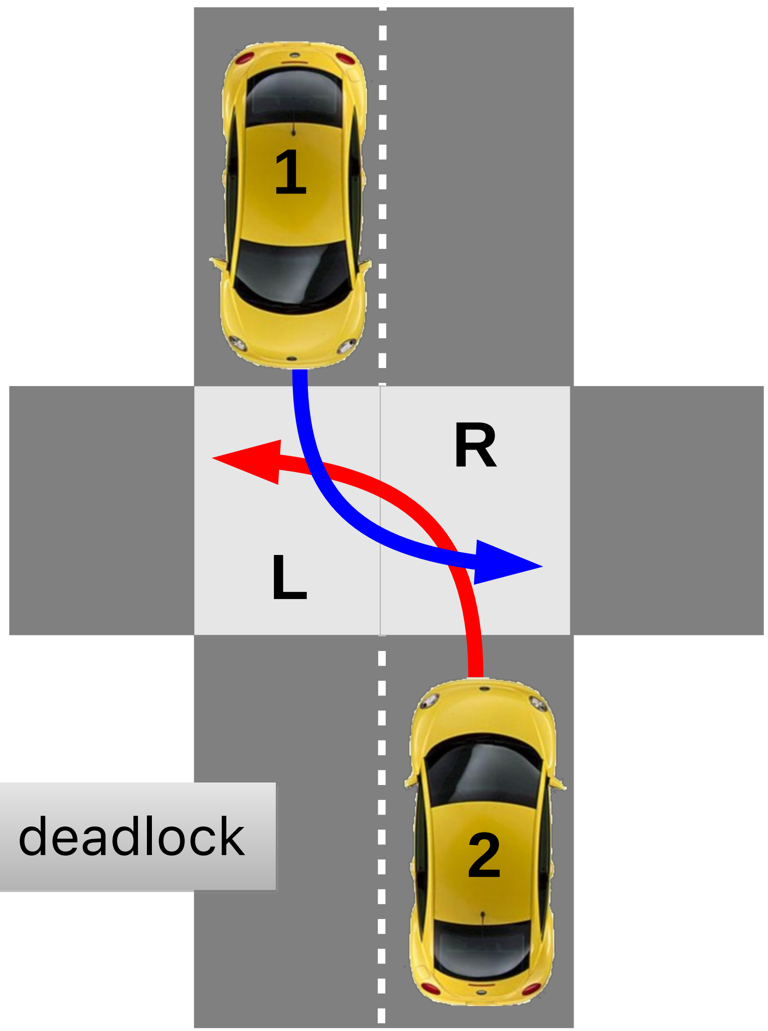


The rule is:
„priority to the right!"
No car is allowed to drive.

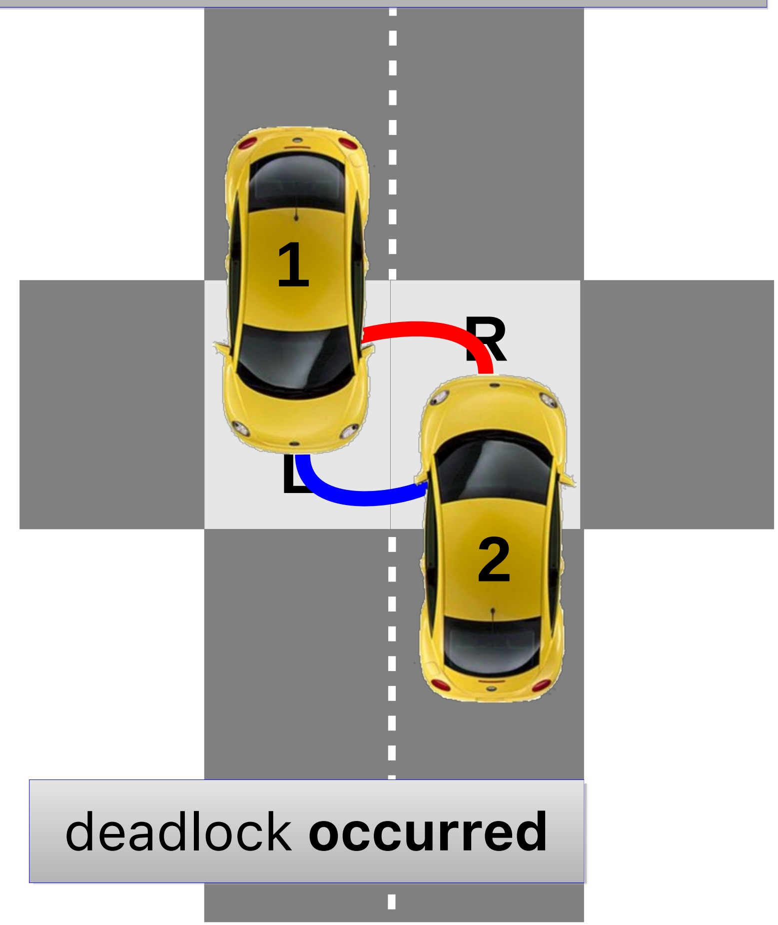
Deadlock situations like this can also occur in processes.

Deadlocks Cause Study - Example

car **1** occupies **L** and requires **R**
car **2** occupies **R** and requires **L**

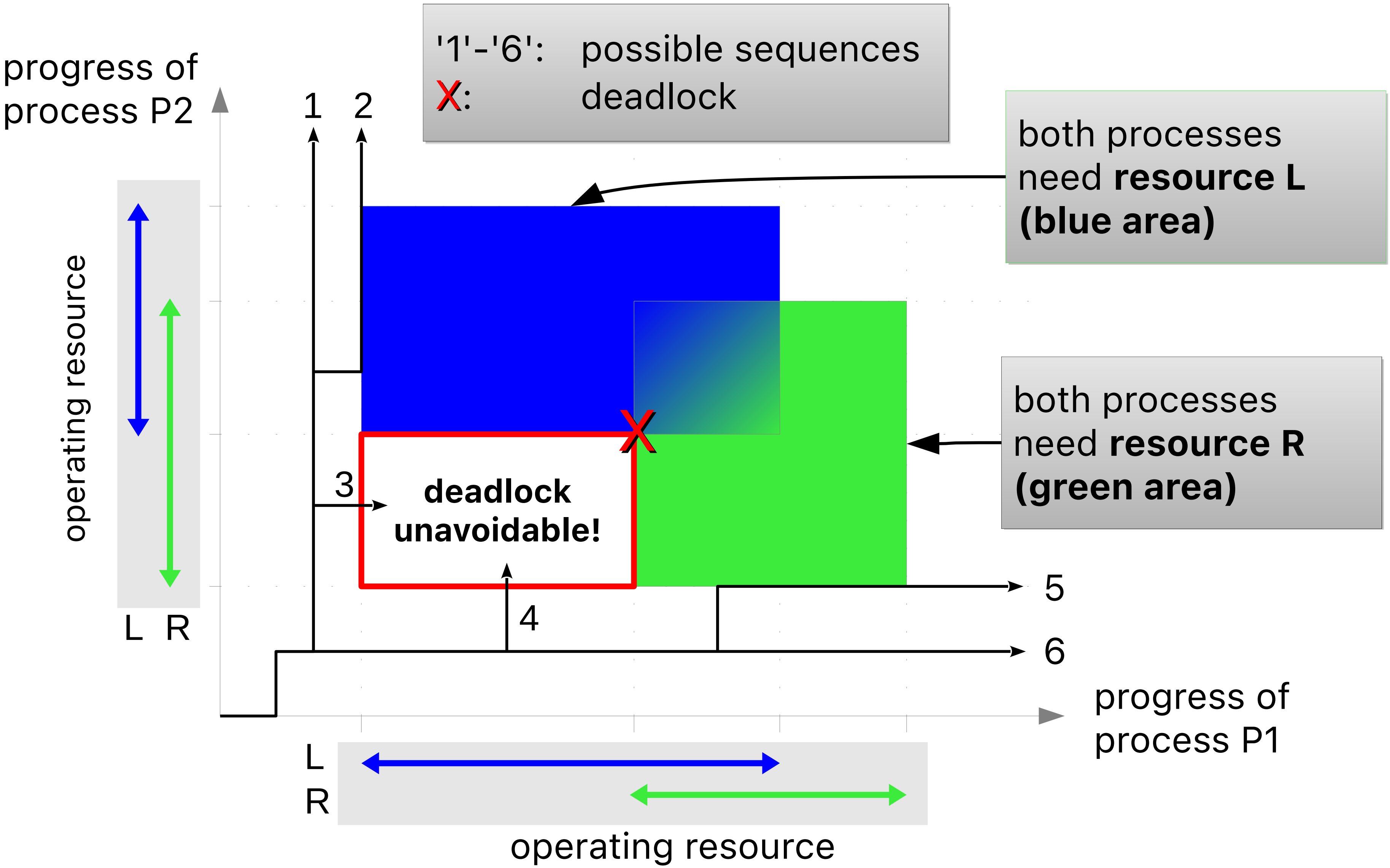


potential deadlock

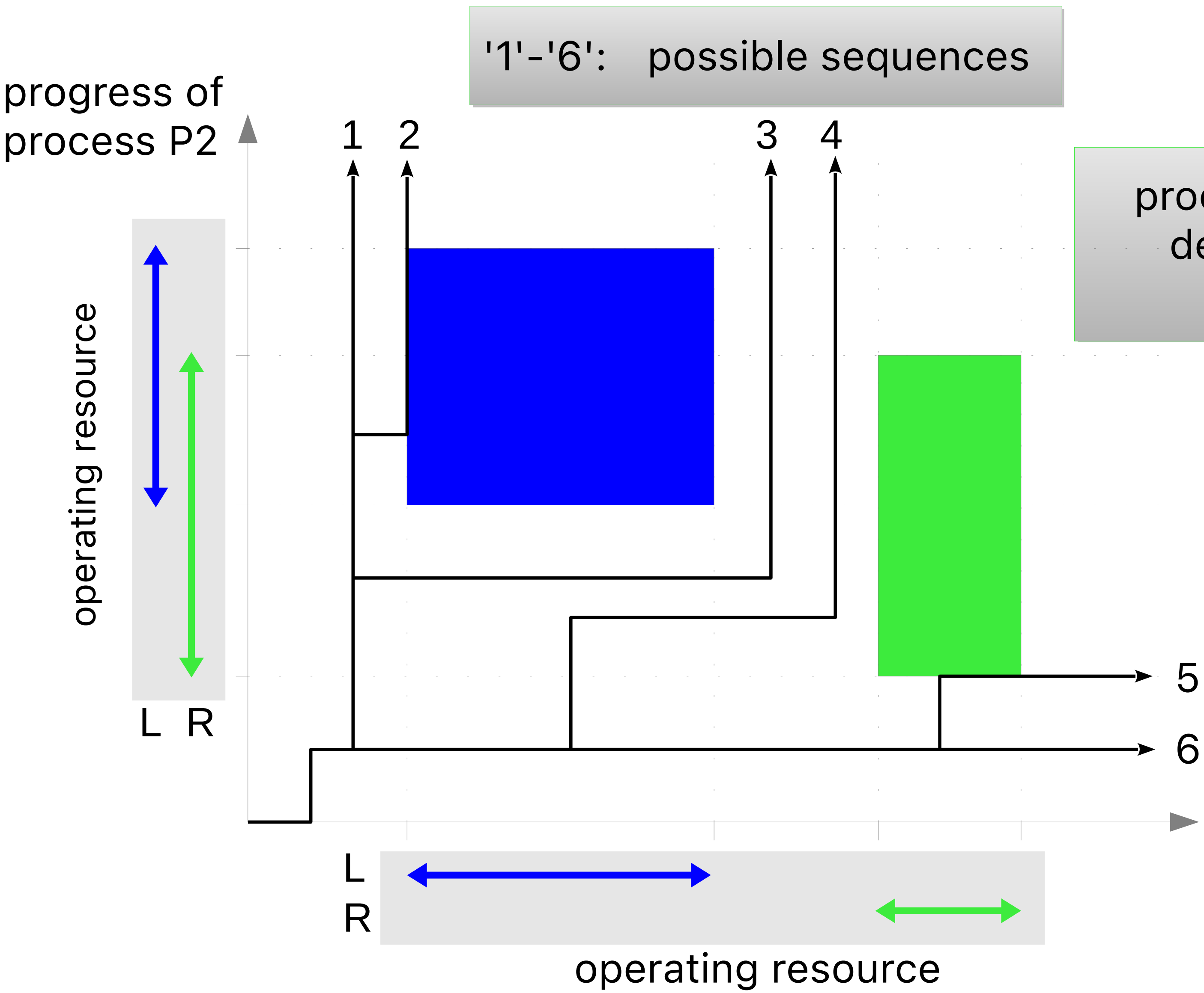


deadlock **occurred**

Deadlocks Cause Study - Abstract

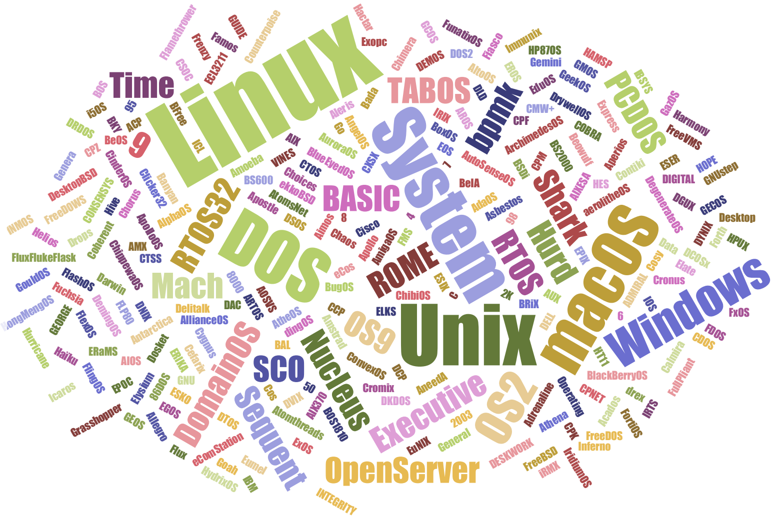


Deadlocks Cause Study - Abstract



processes **can not** deadlock in this scenario

Agenda



- Recap
- Organizational Matters
- Problem Scenario and Cause Study
- **Process Deadlocks**
  - Definition, Variants, and Conditions
  - Consumable and Non-consumable Resources
  - Resource Allocation Graphs
- Classic Deadlock Situation
  - The Dining Philosophers Problem
- Deadlock Counter Measures
  - Prevention, Avoidance, Detection and Recovery
- Summary and Outlook

Deadlocks of Processes: Definition

- Deadly Embrace

Photo: David Maitland, National Geographic

- Definition (Tanenbaum and Woodhull):

„*A* ***set*** *of* ***processes*** *is* ***deadlocked***
if ***each process*** *in the set*
is waiting *for an event*
that only ***another process*** *in the set* ***can cause***.“

Deadlocks of Processes: Variants

- **deadlock** (first variant)
  - passive waiting
  - process state: **BLOCKED**
- **livelock** (second variant)
  - busy waiting (or *„lazy“ busy waiting*)
  - arbitrary process state (including **RUNNING**), but: no progress!
- by comparison, deadlocks are the lesser evil
  - condition can be identified → prerequisite for „resolution“
  - extremely high system load due to busy waiting

Deadlocks of Processes: Conditions

For deadlocks to occur, **all** of the following conditions must be met ("necessary conditions"):

1. **mutual exclusion condition:** exclusive allocation of resources
   - only one process may use a resource at a time
2. **hold-and-wait condition:** follow-up request for resources
   - processes currently holding resources can request new resources
3. **no preemption condition:** no withdrawal of resources
   - granted resources must be explicitly released by the process holding them
4. **circular wait condition:** processes wait for each other circularly
   - several processes, each of which is waiting for a resource held by the next member of the chain

Deadlocks of Processes: Conditions

For deadlocks to occur, **all** of the following conditions must be met ("necessary conditions"):

1. **mutual exclusion condition:** exclusive allocation of resources
   - only one process may use a resource at a time
2. **hold-and-wait condition:** follow-up request for resources
   - processes currently holding resources can request new resources
3. **no preemption condition:** no withdrawal of resources
   - granted resources must be explicitly released by the process holding them

A deadlock **only occurs** when a **fourth condition** holds true at runtime:

4. **circular wait condition:** processes wait for each other circularly
   - several processes, each of which is waiting for a resource held by the next member of the chain

Operating Resources (dt. Betriebsmittel)

- operating resources are managed by the operating system and are made accessible to the processes
- types of operating resources:
  - **consumable operating resources**
    - → are generated (produced) and destroyed (consumed) at runtime
    - → examples: interrupt requests, signals, messages, data from input devices
    - → typical access synchronisation: **unilateral synchronisation**
  - **reusable operating resources**
    - → are allocated by processes for a certain time and then released
    - → examples: CPU, main and background memory, I/O devices, system data structures such as files, process table entries
    - → typical access synchronisation: **multilateral synchronisation, mutual exclusion**
- both resource types are prone to deadlocks

Operating Resources - Reusable

- a deadlock occurs when two processes use a **reusable operating resource** that is requested by the other process
- example: a computer system has 200 GiB of main memory, two processes allocate the memory step by step (memory allocation is done block by block)

Process 1	Process 2
`...`	`...`
`allocate 80 GiB;`	`allocate 70 GiB;`
`...`	`...`
`allocate 60 GiB;`	`allocate 80 GiB;`

If both processes issue their first request before memory is requested, a deadlock is unavoidable.

Operating Resources - Consumable

- a deadlock occurs when two processes are waiting for a **consumable operating resource** to be produced by the other
- example: synchronisation signals are "sent" between two processes using the **wait** and **signal** semaphore operations

Process P1

```
semaphore s1 = {0, NULL};
wait (&s1);
… // consume (from P2)
… // produce (for P2)
signal (&s2);
```

Process P2

```
semaphore s2 = {0, NULL};
wait (&s2);
… // consume (from P1)
… // produce (for P1)
signal (&s1);
```

Each process waits for a synchronisation signal from the other, but this cannot be sent because it is blocked itself.

Resource Allocation Graphs (RAGs)

(dt. **Betriebsmittelbelegungsgraphen**)

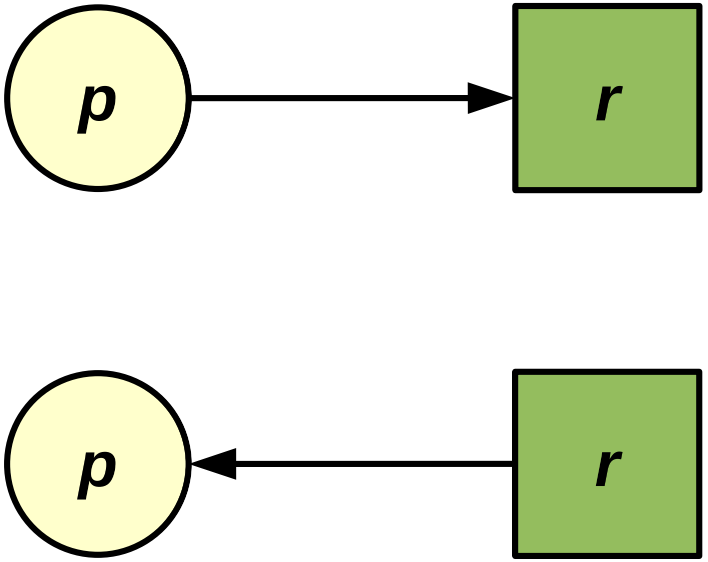
- to visualise and automatically detect deadlock situations Resource Allocation Graphs (RAGs) are used
- RAGs describe a current system state
  - **nodes:** processes and resources
  - **edges:** indicate an occupation or a request

operating resource R_1 is **requested** by process P_1

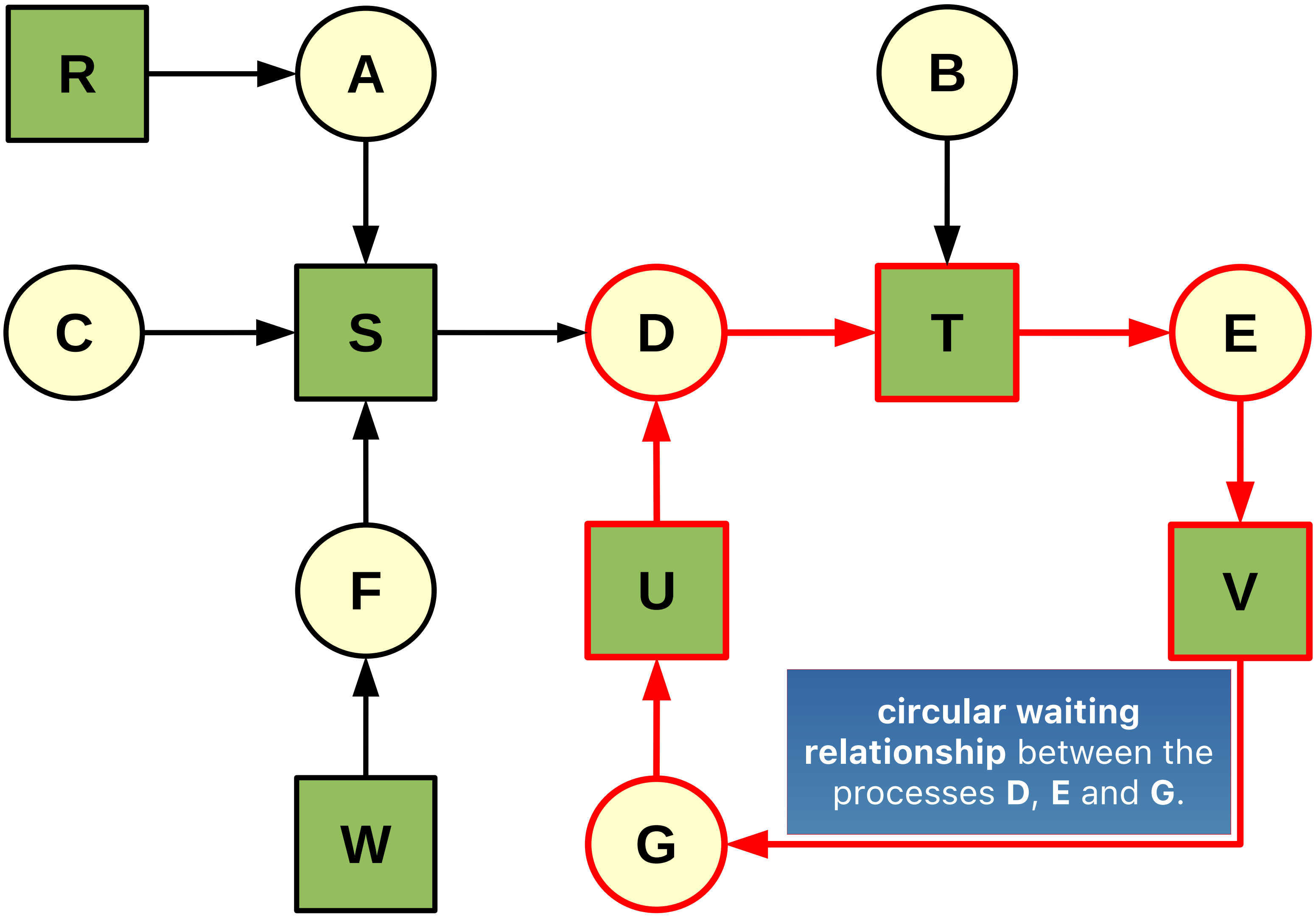
process P_2 **has occupied** the operating resource R_2

Resource Allocation Graphs (RAGs)

- situation: there are 7 processes (**A** to **G**) and 6 resources (**R** to **W**)
  - **is there circular waiting?**
  - **which processes are involved?**
- **current state**:
  - **A** occupies **R** and requests **S**
  - **B** does not occupy anything, but requests **T**
  - **C** does not occupy anything, but requests **S**
  - **D** occupies **U** and **S** and requests **T**
  - **E** occupies **T** and requests **V**
  - **F** occupies **W** and requests **S**
  - **G** occupies **V** and requests **U**



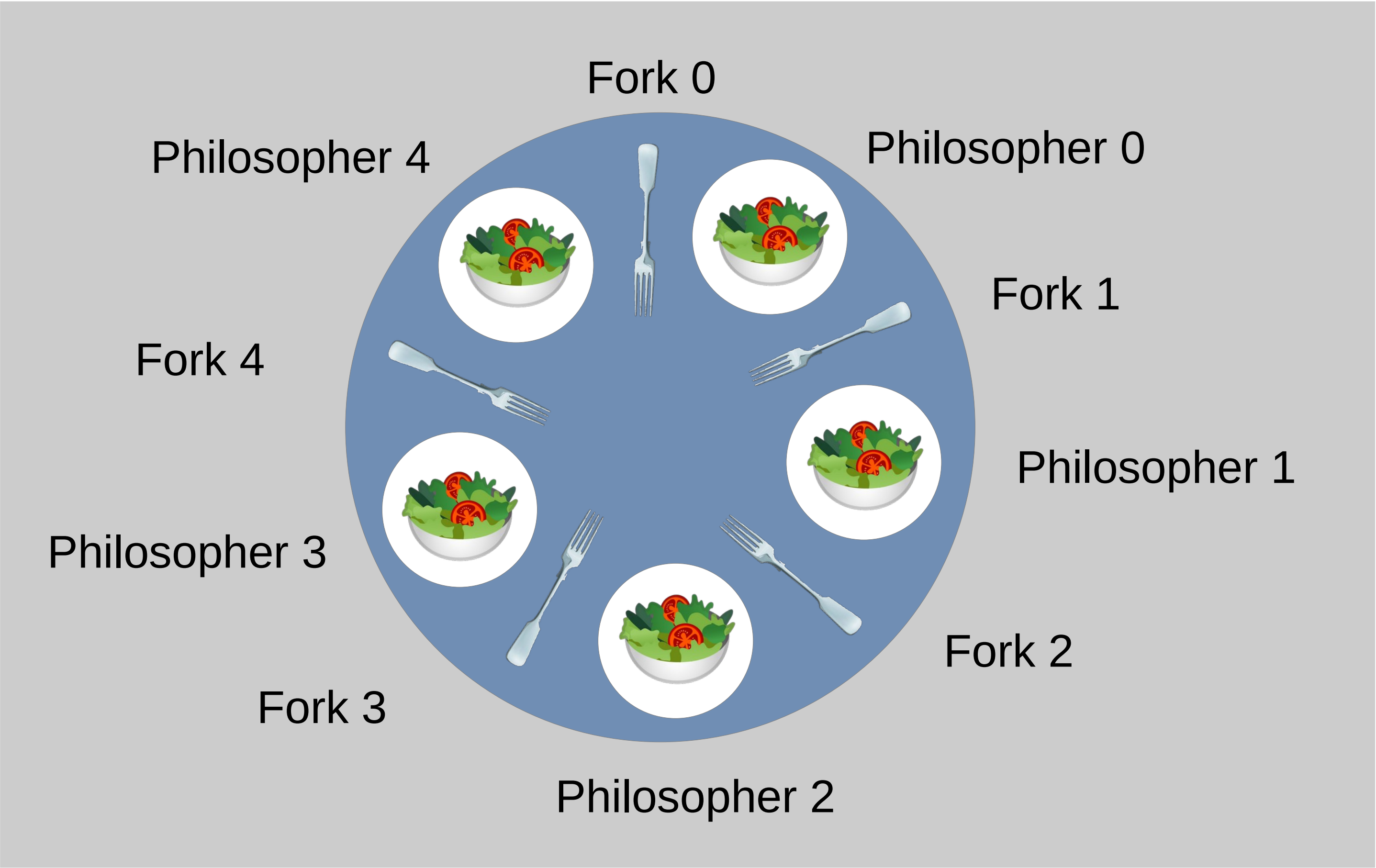
Resource Allocation Graphs (RAGs)



Agenda

- Recap
- Organizational Matters
- Problem Scenario and Cause St
- Process Deadlocks
  - Definition, Variants, and Conditions
  - Consumable and Non-consumable Resources
  - Resource Allocation Graphs
- Classic Deadlock Situation
  - The Dining Philosophers Problem
- Deadlock Counter Measures
  - Prevention, Avoidance, Detection and Recovery
- Summary and Outlook

The Dining Philosophers Problem



The Dining Philosophers Problem (by E. W. Dijkstra)

Five philosophers, who have nothing else to do but think and eat, sit at a round table.

Thinking makes you hungry - so every philosopher will also eat. For this, however, a philosopher always needs **both** forks lying next to his plate.

Philosopher → Process
Fork → Operating Resource

Deadlocked Philosophers?

The first three necessary conditions are met:

✓ ■ **mutual exclusion**

- for reasons, the philosophers are not allowed to share forks

✓ ■ **hold and wait**

- the philosophers are so preoccupied with their thoughts before eating that they can neither really grasp their forks at the same time, nor do they think of putting a fork down again

✓ ■ **no preemption**

- snatching the fork from another philosopher is out of the question

? **But:** does a deadlock really happen?

Dining Philosophers - Attempt 1

```
void phil (int who) {
    while (1) {
        think();
        grab(who);
        eat();
        drop(who);
    }
}

void think () { ... }
void eat   () { ... }
```

```
semaphore fork[NPHIL] = {
  {1, NULL}, ...
};

void grab (int who) {
    wait(&fork[who]);
    wait(&fork[(who+1)%NPHIL]);
}

void drop (int who) {
    signal(&fork[who]);
    signal(&fork[(who+1)%NPHIL]);
}
```

With the help of a semaphore, **mutual exclusion** is guaranteed when accessing the forks. Each philosopher first takes his right fork and then his left fork.

Attempt 1: Prone to Deadlock Situations

context switch

every philosopher grabs his **right fork** first

```
void grab(0) {
  wait(&fork[0]);
```

```
void grab(1) {
  wait(&fork[1]);
```

```
void grab(2) {
  wait(&fork[2]);
```

```
void grab(3) {
  wait(&fork[3]);
```

```
void grab(4) {
  wait(&fork[4]);
  wait(&fork[0]);
}
```

Attempt 1: Prone to Deadlock Situations

context switch

```
void grab(0) {
  wait(&fork[0]);
```

```
void grab(1) {
  wait(&fork[1]);
```

```
void grab(2) {
  wait(&fork[2]);
```

```
void grab(3) {
  wait(&fork[3]);
```

```
void grab(4) {
  wait(&fork[4]);
  wait(&fork[0]);
}
```

```
  wait(&fork[4]);
}
```

```
  wait(&fork[3]);
}
```

```
  wait(&fork[2]);
}
```

```
  wait(&fork[1]);
}
```

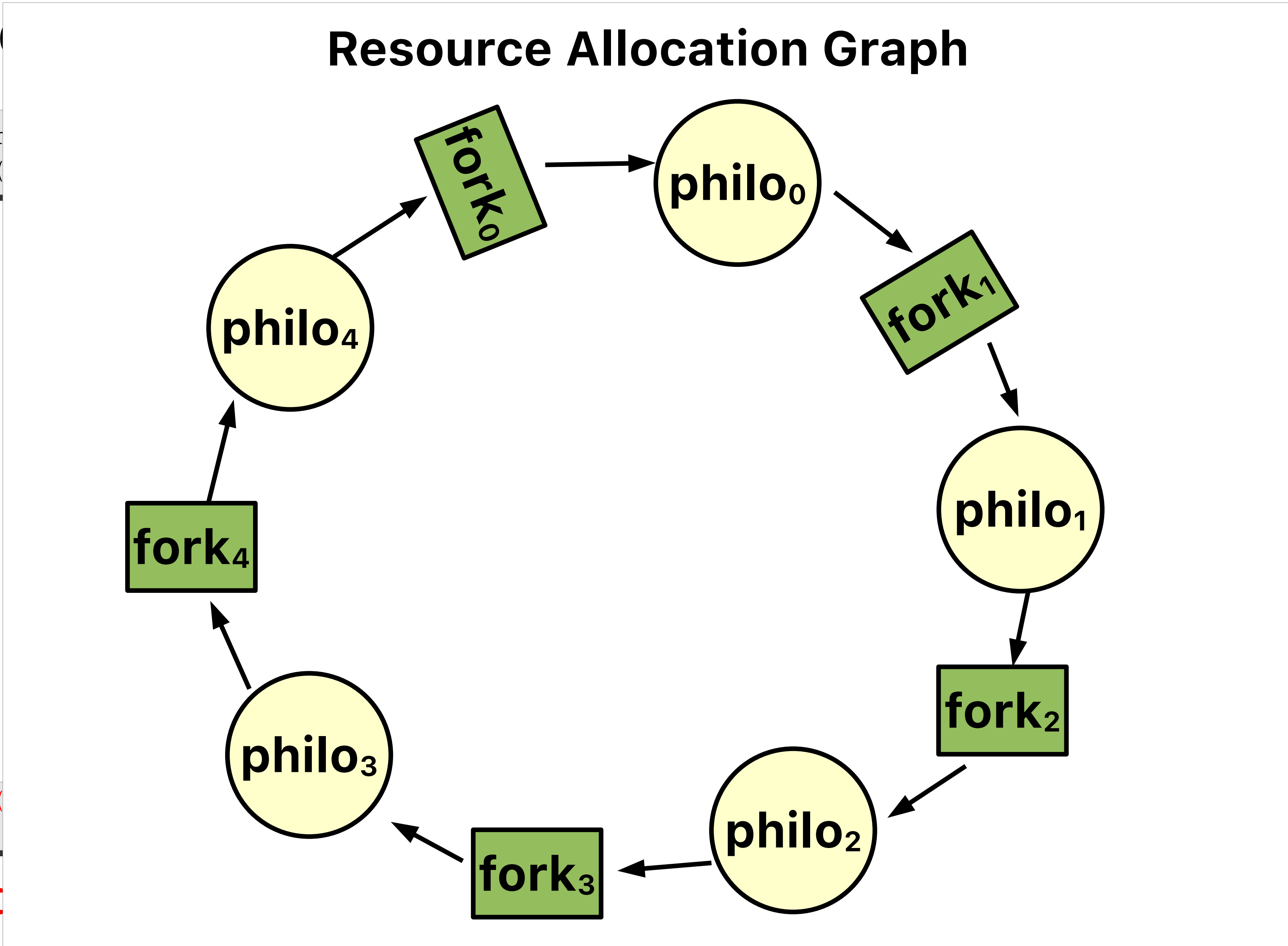
every philosopher grabs his **right fork** first

all philosophers block at the second wait

Deadlock Situation

Att

```
void gr
  wait(
```




```
);
);
```

he

```
  wait(
}
```

Dead

Dining Philosophers - Attempt 2

```
semaphore mutex = {1, NULL};

void grab (int who) {
  wait(&mutex);
  wait(&fork[who]);
  wait(&fork[(who+1) % NPHIL]);
  signal(&mutex);
}
```

The problem of the first implementation were context switches between the 1st and 2nd wait, in fact, a **critical section**.

The second implementation protects this critical section by mutual exclusion.

- Is it deadlock free?

- Is the second attempt a solid solution?

Dining Philosophers - Attempt 2

```
semaphore mutex = {1, NULL};

void grab (int who) {
  wait(&mutex);
  wait(&fork[who]);
  wait(&fork[(who+1) % NPHIL]);
  signal(&mutex);
}
```

The problem of the first implementation were context switches between the 1st and 2nd wait, in fact, a **critical section**.

The second implementation protects this critical section by mutual exclusion.

✓ - Is it deadlock free? **Yes**
  - max. 1 process can wait for a fork (cycle needs 2!)
  - a process waiting for **mutex** has no fork

! - Is the second attempt a solid solution? - No!
  - when $\mathbf{philo}_{who}$ eats, $\mathbf{philo}_{who+1}$ blocks a the entry of the critical section and **all other** philosophers **block**, too.
  - **result: low degree of concurrency** and **inefficient use of available resources**

Dining Philosophers - Attempt 3

```
const int N = 5;                        /* number of dining philosophers     */
semaphore mutex = {1, NULL};            /* mutual exclusion                  */
semaphore s[N] = {{0, NULL},...};       /* one semaphor for each philosopher */
enum {THINKING, EATING, HUNGRY } state[N]; /* philosopher states             */

int left(i) { return (i+N-1)%N; } /* index left neighbor  */
int right(i) { return (i+1)%N; }  /* index right neighbor */
```

```
void test (int i) {
  if (state[i] == HUNGRY && state[left(i)]  != EATING &&
                            state[right(i)] != EATING) {
    state[i] = EATING;
    signal(&s[i]);
  }
}
```

```
void grab(int i) {
  wait(&mutex);
  state[i] = HUNGRY;
  test(i);
  signal(&mutex);
  wait(&s[i]);
}
```

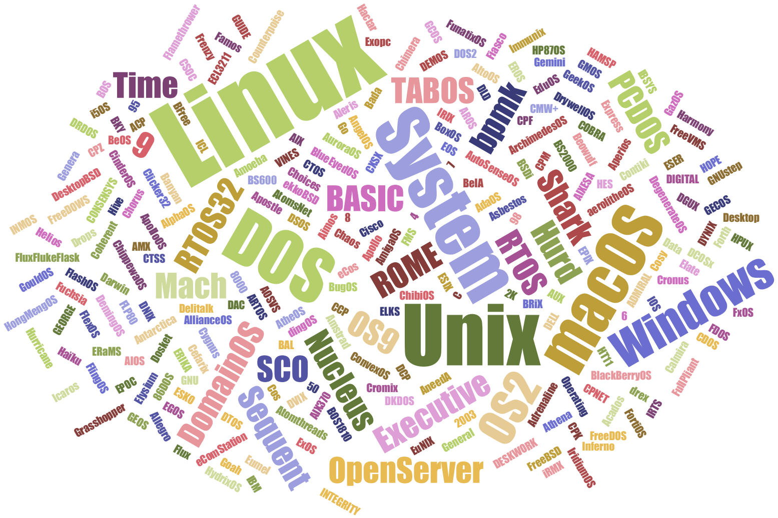
```
void drop(int i) {
  wait(&mutex);
  state[i] = THINKING;
  test(left(i));
  test(right(i));
  signal(&mutex);
}
```

This solution is deadlock-free and has a maximum degree of concurrency

Dining Philosophers - Discussion

- **for specific problems:** there are usually many different ways to ensure deadlock freedom
  - solutions differ in the degree of possible concurrency
  - if a solution is too restrictive, operating resources will temporarily lie idle unnecessarily
- **in general:** representative example of deadlock problems in the management of indivisible resources
  - based on E.W. Dijkstra (1965)
  - established standard scenario for evaluating and illustrating operating system and language mechanisms for concurrent programming

Agenda



- Recap
- Organizational Matters
- Problem Scenario and Cause Study
- Process Deadlocks
  - Definition, Variants, and Conditions
  - Consumable and Non-consumable Resources
  - Resource Allocation Graphs
- Classic Deadlock Situation
  - The Dining Philosophers Problem
- **Deadlock Counter Measures**
  - **Prevention, Avoidance, Detection and Recovery**
- Summary and Outlook

Deadlock Prevention

(dt. **Verklemmungsvorbeugung**)

- **indirect methods** invalidate (at least) one of the conditions 1-3
  1. use of non-blocking process synchronisation methods
  2. design resource requests in an indivisible (atomic) way
  3. withdrawal of operating resources through **virtualisation**
     - ➔ virtual memory, virtual devices, virtual processor cores
- **direct methods** invalidate condition 4
  4. linear/total order of operating resources:
     - ➔ operating resource R_i can only be allocated before resource R_j, if i is in a linear order directly before j (i.e., $i < j$).

➔ indirect/direct methods: rules, that prevent the occurrence of deadlocks

- methods that take effect at **design/implementation time**

Deadlock Avoidance

(dt. **Verklemmungsvermeidung**)

- preventing circular waiting (**at run-time**) through strategic measures:
  - none of the first three necessary conditions is invalidated
  - ongoing **resource requirement analysis** eliminates circular waiting
- requests for operating resources of the processes are to be controlled:
  - **„safe state“** must always be maintained:
    - → there is a process sequence in which each process can cover its maximum resource requirement
  - **„unsafe states“** must be **avoided**:
    - → allocation rejection in the event of uncovered operating resource requirements
    - → do not serve requesting processes or suspend them at an early stage
- problem: a priori knowledge of maximum operating resource requirements of processes is necessary

Deadlock Avoidance: Safe and Unsafe States

(usi

- i ... ents
  nents
- c
  F ... ch,
  2

„Banker's Algorithm“

- management of process/operating resource matrices for **current occupancy** and **maximum occupancy**
- function for **finding a process sequence** in which the which the resources do not run out even if the "credit limit" is exhausted completely
- anticipatory application of this function in the event of fulfilling resource allocation requests

(c.f., Tanenbaum)

Deadlock Avoidance: Safe and Unsafe States

(using the example of multiple existing resources)

- **initial state**: a UNIX system with **12 shared memory segments**
  - process P1 requires max. 10 segments, P2: 4 segments, P3: 9 segments
- **current state**:
  P1 occupies 6 segments, P2 and P3 occupy 2 segments each,
  2 segments are still available (free)
  - P3 requests 1 segment, 1 would remain free → **unsafe state**
    - ➔ P3's request is rejected, P3 has to wait
  - P1 requests 2 segments, no free segment remains → **unsafe state**
    - ➔ the request of P1 is rejected, P1 has to wait
  - **safe** process **sequence**: P2 → P1 → P3

Deadlock Detection

(dt. **Deadlockerkennung**)

- deadlocks are (silently) accepted (**„ostrich algorithm“**) . . .
  - nothing in the system prevents the occurrence of circular wait conditions
  - *none* of the four conditions is invalidated
- approach: create **waiting graph** and search cycles → **O(n)**
  - too frequent verification wastes resources/computational power
  - too infrequent verification leaves operating resources lying idle
- **search for cycles** occurs mostly at large time intervals when:
  - serving operating resource requests takes too long
  - the utilisation of the CPU decreases despite process increase
  - the CPU has already been idle for a very long period of time

Deadlock Recovery

recovery phase *after* the detection phase

- **terminate processes** and thus free up operating resources
  - stopping deadlocked processes step by step (huge efforts)
    - ➔ start with the "most effective victim" – who is it?
  - terminate **all** deadlocked processes (huge damage)
- **revoke operating resources** and begin with the „most effective victim“ – again, who is it?
  - reverse or restart the identified process
    - ➔ transactions, checkpointing/recovery (huge effort)
  - starvation of the processes that have been reversed must be avoided
  - also, beware of livelocks!
- walking a tightrope between harm and expense:
  - damage is unavoidable, potential for (severe) follow-up damage

Discussion on Countermeasures

- deadlock *prevention*, *avoidance*, *detection*, *recovery*
- methods for avoidance/detection are less relevant in practice in the context of operating systems
  - methods are hard to implement, too costly and therefore not applicable
  - moreover, the prevalence of sequential programming makes these methods little necessary
- risk of deadlocks can be solved by virtualisation of operating equipment
  - processes use/occupy **logical operating resources** only
  - goal: withdraw **physical operating resources** from the processes (without their knowledge) at critical moments
  - with this, condition of preemption is defused
- deadlock prevention methods are more relevant

Agenda

- Recap
- Organizational Matters
- Problem Scenario and Cause Study
- Process Deadlocks
  - Definition, Variants, and Conditions
  - Consumable and Non-consumable Resources
  - Resource Allocation Graphs
- Classic Deadlock Situation
  - The Dining Philosophers Problem
- Deadlock Counter Measures
  - Prevention, Avoidance, Detection and Recovery
- Summary and Outlook

⏭ Summary and Outlook

- **summary**
  - **process deadlocks:** mutual blocking of concurrent but independent processes
    - → deadly embrace: deadlocks and livelocks
    - → four deadlock conditions must hold true – invalidate (at least) a single condition to avoid deadlocks
    - → occurs with reusable and consumable resources
  - **deadlock handling**
    - → dining philosophers and discussion of (potential) solutions
    - → countermeasures: deadlock prevention, avoidance, detection and resolution
- **outlook: memory management**
  - main memory is a fundamental operating resource to processes
  - memory allocation and memory management for multi-program operation, address mapping

References and Acknowledgments

Lecture

- Systemnahe Programmierung in C (SPiC), Betriebssysteme (Jürgen Kleinöder, Wolfgang Schröder-Preikschat)
- Betriebssysteme und Rechnernetze (Olaf Spinczyk, Embedded Software Systems Group, Universität Osnabrück)

Teaching Books and Reference Book

[1] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*, John Wiley & Sons*, 2018*.

[2] Andrew Tanenbaum, Herbert Bos: *Modern Operating Systems*, Pearson, 2015.

[3] Wolfgang Schröder-Preikschat: *Grundlage von Betriebssystemen – Sachwortverzeichnis*, 2023.
`https://www4.cs.fau.de/~wosch/glossar.pdf`