Universität Bamberg

Operating Systems Engineering

Lecture 9: Deadlocks and Devices

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

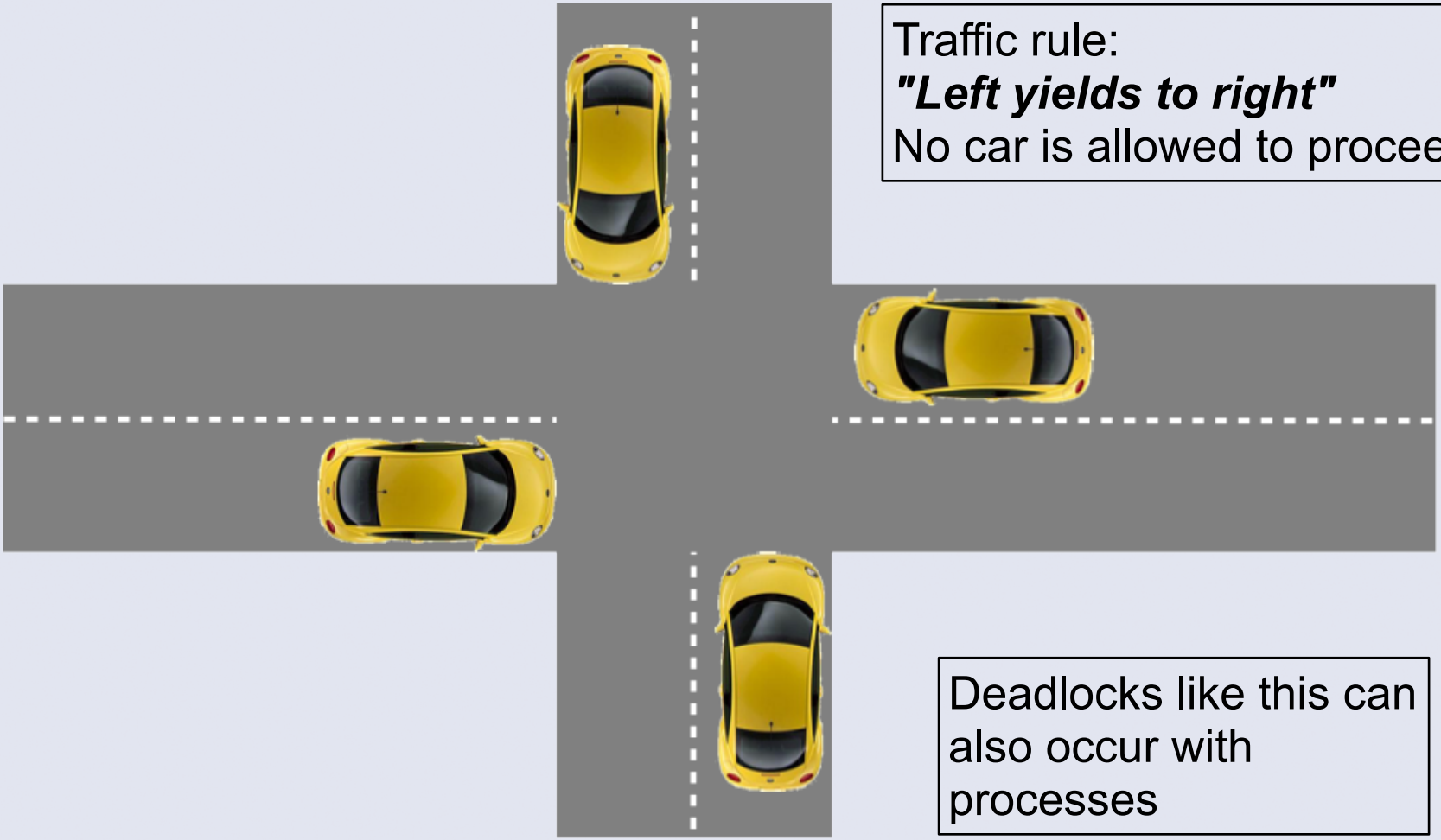
Concurrency revisited

- Processes run concurrently in a computer
  - This also includes interrupt handlers, which run *asynchronously* to the main OS kernel
- To coordinate processes, we use *synchronization primitives*

- Basic idea: *passive waiting*
- Semaphores enable...
  - mutual exclusion
  - one-sided synchronization
  - resource-oriented synchronization

- Waiting mechanisms lead to ***deadlock problems***

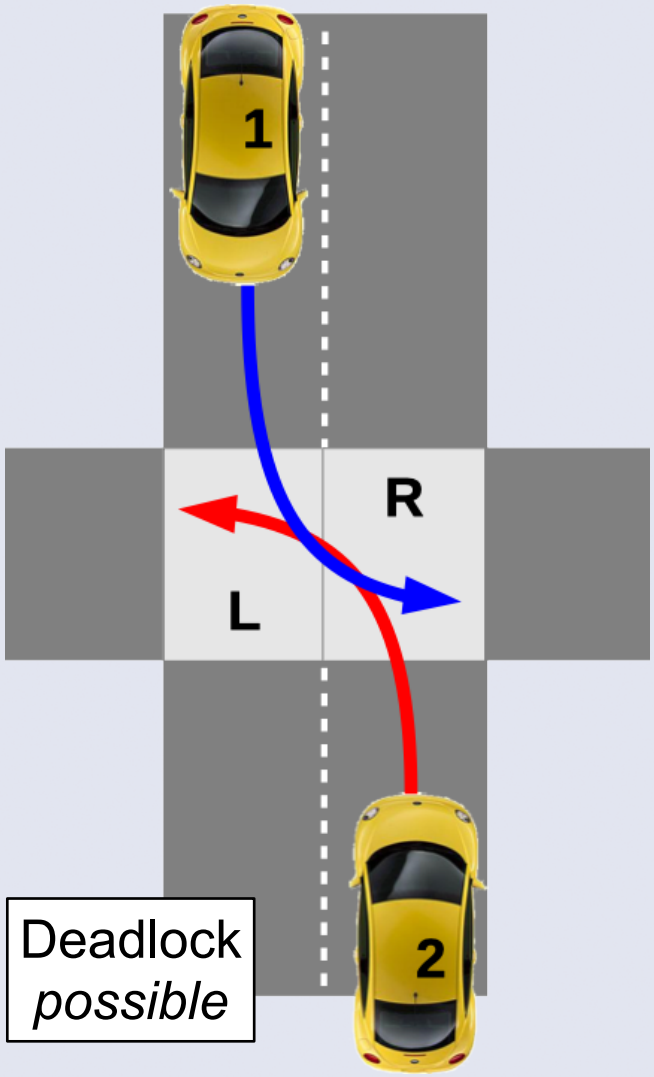
Deadlocks



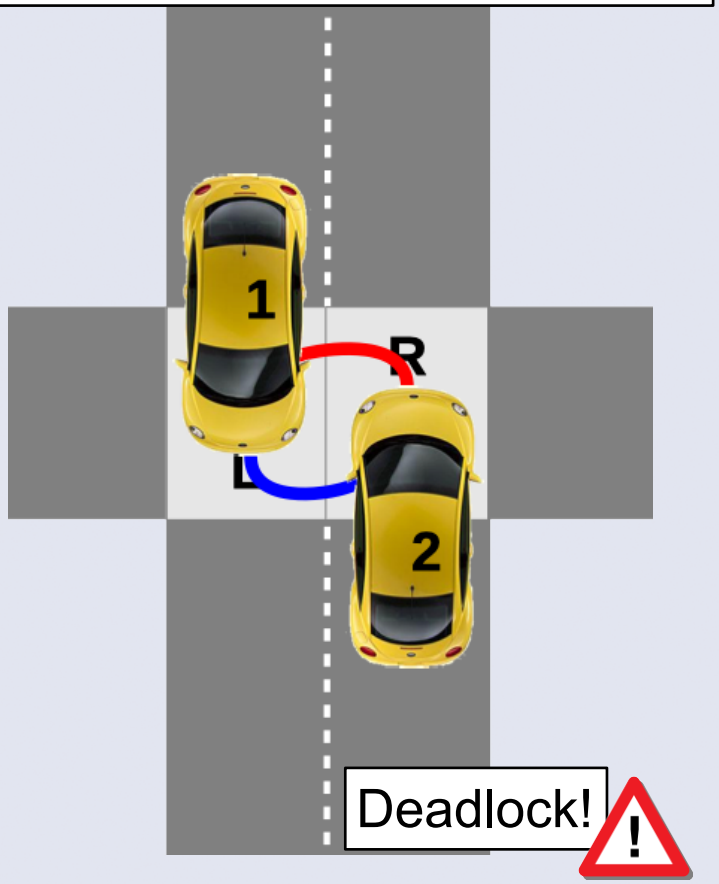
Traffic rule:
"Left yields to right"
No car is allowed to proceed

Deadlocks like this can also occur with processes

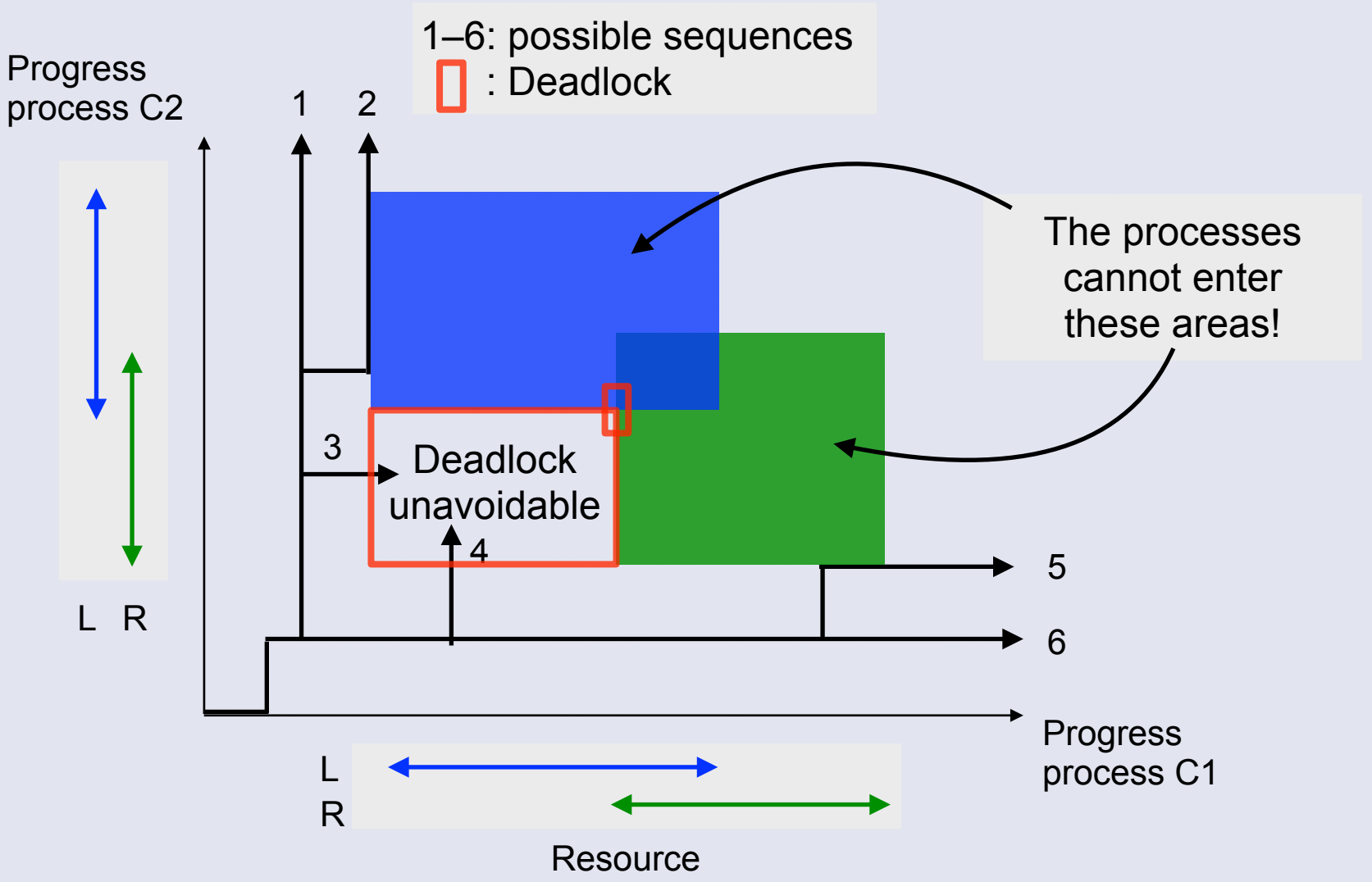
Why do deadlocks occur?



Car 1 occupies “**L**” and needs “**R**”
Car 2 occupies “**R**” and needs “**L**”

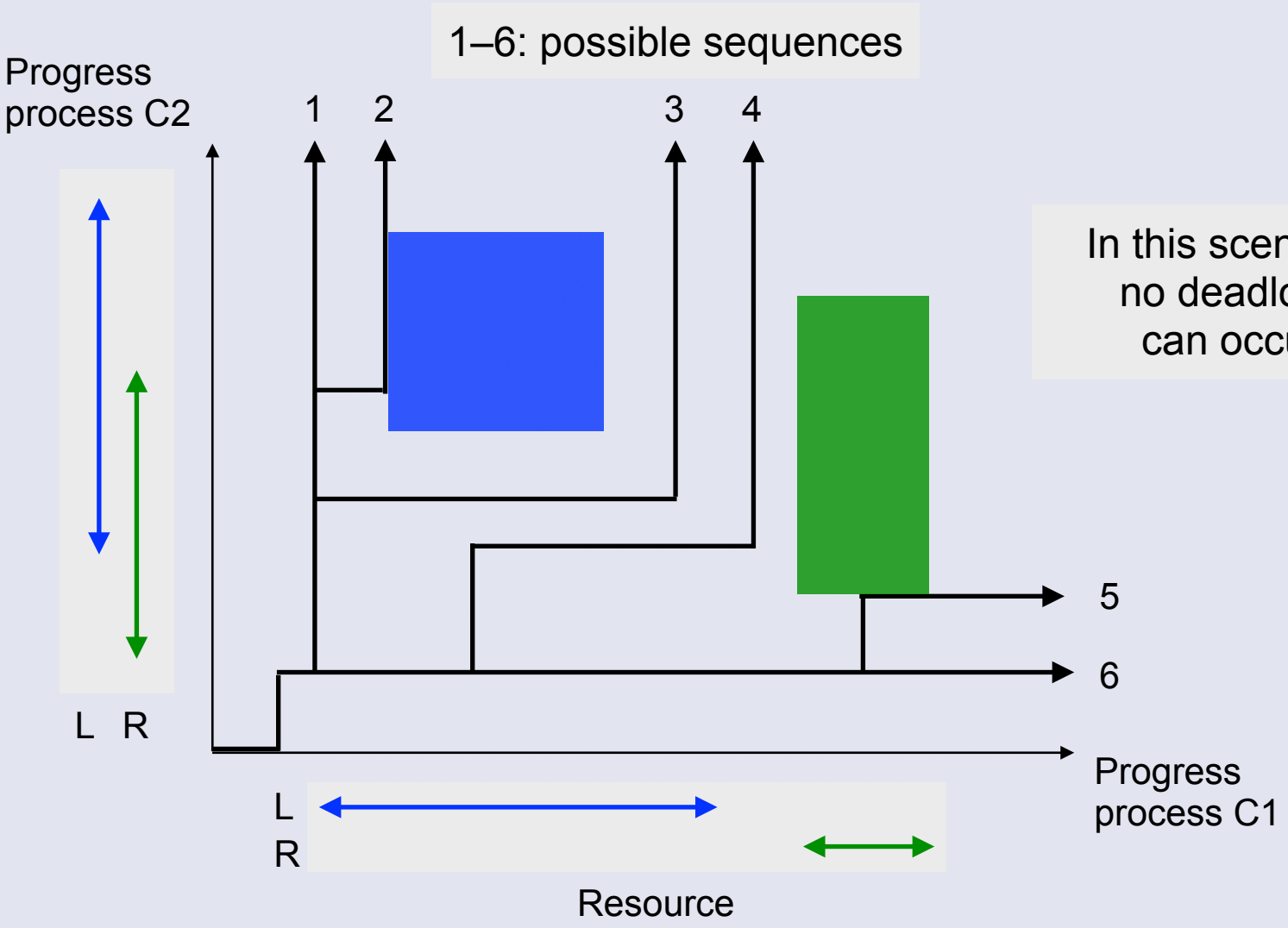


Abstract view of deadlocks



Abstract view of deadlocks

Universität Bamberg



Deadlocking processes

The term “deadlock” (in computer science) means:

> „[…] a situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.“
>
> William Stallings.
> Operating Systems: Internals and Design Principles.

Deadlock processing

- Alternative 1: **Deadlock**
  - Passive waiting
  - Process state is BLOCKED
- Alternative 2: **Livelock**
  - Active waiting (busy waiting or “lazy” busy waiting)
  - Arbitrary process state (including RUNNING), but none of the involved processes is able to proceed

- ***Deadlocks are the “lesser evil”***
  - This state is uniquely discoverable
    → Basis to ”resolve” deadlocks is available
- Active waiting results in an extremely high system load

Conditions for deadlocks

All of the following three conditions must be fulfilled for a deadlock to occur ("**necessary conditions**"):

1. **Exclusive allocation** of resources ("mutual exclusion")
   - Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process
2. Allocation of **additional resources** ("hold and wait")
   - A process may hold allocated resources while awaiting assignment of other resources
3. **No removing** of resources ("no preemption")
   - The OS is unable to forcibly remove a resource from a process once it is allocated

Conditions for deadlocks

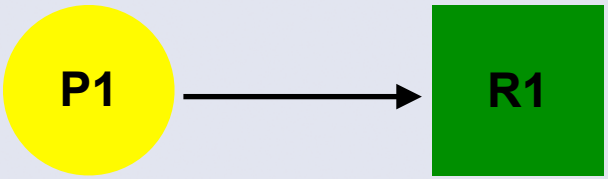
All of the following three conditions must be fulfilled for a deadlock to occur ("**necessary conditions**"):

1. **Exclusive allocation** of resources ("mutual exclusion")
   - Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process
2. Allocation of **additional resources** ("hold and wait")
   - A process may hold allocated resources while awaiting assignment of other resources
3. **No removing** of resources ("no preemption")
   - The OS is unable to forcibly remove a resource from a process once it is allocated
4. Only if an **additional condition** occurs at runtime, we really have a **deadlock**: **"circular wait"**
   - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Resource allocation graphs

... are used to visualize and automatically detect deadlock situations

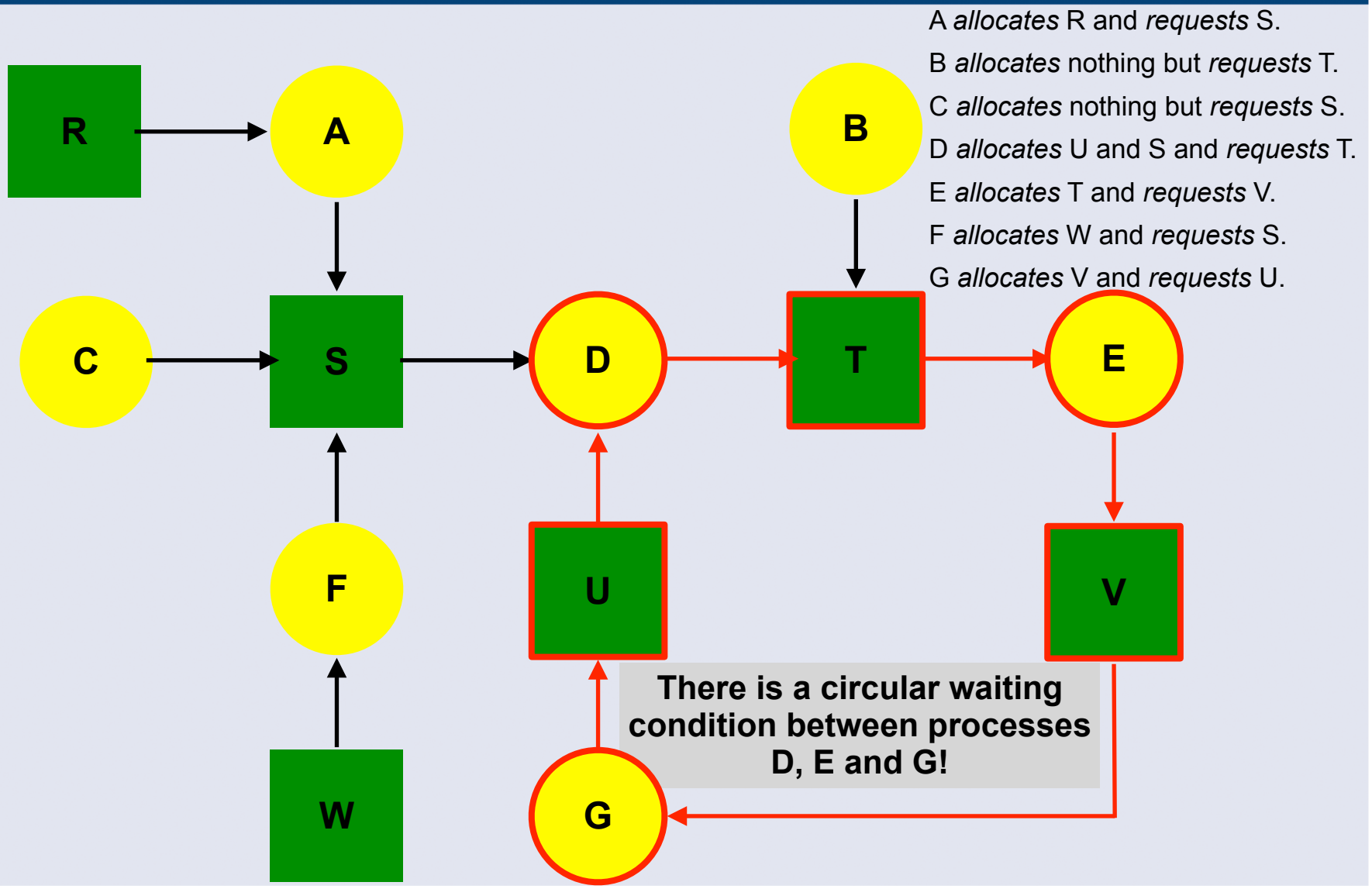
- They describe the current system state
- The nodes are processes and resources
- The edges show an allocation or a request



Resource R1 is *requested* by process P1

Process P2 *allocates* resource B2

Resource allocation graphs



Preventing deadlocks

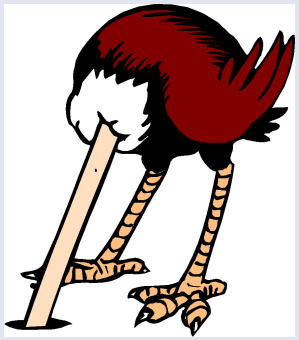
- **Indirect methods** invalidate one of the conditions 1–3
  1. use non blocking approaches
  2. only allow atomic resource allocations
  3. enable the preemption of resources using virtualization
     - virtual memory, virtual devices, virtual processors

- **Direct methods** invalidate condition 4
  4. introduce a linear/total order of resource classes:
     - Resource R_i can only be successfully allocated before R_j if i is ordered linear before j (i.e. $i < j$)

- **Rules** that prevent deadlocks
  - Methods at design or implementation time

Detecting deadlocks

- Deadlocks are (silently) accepted („ostrich algorithm“)...
  - Nothing in the system tries to avoid the occurrence of waiting cycles!
  - None of our four conditions is invalidated



- Approach: **create waiting graph** and search for **cycles** → O(n)
  - Checking too frequently wastes resource and compute time
  - Checking too infrequently wastes unused resources
- **Cycle search** take place in large time intervals only, if…
  - Resource requests take too much time
  - The CPU load decreases even though the number of processes increases
  - The CPU is already idle for a long time

Resolving deadlocks

Recovery phase after the detection phase

- **Terminate processes** to release resources
  - Terminate deadlocked processes step by step (lots of effort)
    - Start with the ”most effective victim” (**?**)
  - Terminate all deadlocked processes (large possible damage)
- **Preempt resources** and start with the “most effective victim” (**?**)
  - *Roll back* or *restart* the affected process
    - Use transactions, checkpointing/recovery (lots of effort)
  - A starvation of the rolled back processes has to be avoided
  - Also: *take care of livelocks!*
- **Balance** between damage and effort:
  - Damages are unavoidable, so we need to consider what the consequence is

Discussion of prevention methods

- Methods to avoid/detect deadlocks have little practically relevance in the context of operating systems
  - They are very difficult to implement, require too much overhead and are thus not useable
  - Since sequential programming is still the predominant approach, avoidance and detection methods are rarely required
- The risk of deadlock can be solved by virtualizing resources
  - Processes only request/allocate logical resources
  - Using virtualization, physical resources can be removed (preempted) from a process (without the process noticing) in critical moments
  - Accordingly, the “no preemption” condition is invalidated

⇒ Prevention methods more commonly used & relevant in practice

Device interrupts on RISC-V

Interrupts

Internal: CLINT

External: PLIC

Devices control their interrupts in addition

UART registers

16550 UART registers: two related to interrupts

Register	READ MODE	WRITE MODE
0	Receive Holding Register	Transmit Holding Register
	N/A	LSB of Divisor Latch when Enabled
1	N/A	**Interrupt Enable Register**
	N/A	MSB of Divisor Latch when Enabled
2	**Interrupt Status Register**	FIFO control Register
3	N/A	Line Control Register
4	N/A	Modem Control Register
5	Line Status Register	N/A
6	Modem Status Register	N/A
7	Scratchpad Register Read	Scratchpad Register Write

16550 Interrupt Enable Register (IER)

The **UART IER register** masks the incoming interrupts from receiver ready, transmitter empty, line status and modem status registers to the INT output pin.

- bit 0:
  0=disable the receiver ready interrupt
  1=enable the receiver ready interrupt
- bit 1:
  0=disable the transmitter empty interrupt
  1=enable the transmitter empty interrupt
- bit 2:
  0=disable the receiver line status interrupt
  1=enable the receiver line status interrupt
- bit 3:
  0=disable the modem status register interrupt
  1=enable the modem status register interrupt
- bits 4–7:
  All these bits are set to logic zero

Bit	Interrupt configuration
0	1 = enable receive int.
1	1 = enable transmit int.
2	1 = enable receiver line status int.
3	1 = enable receiver tramsmit status int.
4–7	unused

16550 Interrupt Status Register (ISR)

- Four level prioritized interrupt conditions to minimize software overhead during data character transfers
- The Interrupt Status Register (**ISR**) provides the source of the interrupt in prioritized manner: **bit 0 = 0 → interrupt occured**
  - It provides the highest interrupt level to be serviced by CPU
  - No other interrupts are acknowledged until the particular interrupt is serviced

Prio	bit 3	bit 2	bit 1	bit 0	SOURCE OF THE INTERRUPT
1	0	1	1	0	LSR (Receiver Line Status Register)
2	0	1	0	0	RXRDY (Received Data Ready)
2	1	1	0	0	RXRDY (Received Data Timeout)
3	0	0	1	0	TXRDY (Transmitter Holding Register Empty)
4	0	0	0	0	MSR (Modem Status Register)

External (device) interrupts on RISC-V

- External interrupts inform the CPU of a service request by a device other than the timer
  - Signal that some external, or platform interrupt has occurred
  - For example, the UART device could have just filled its buffer

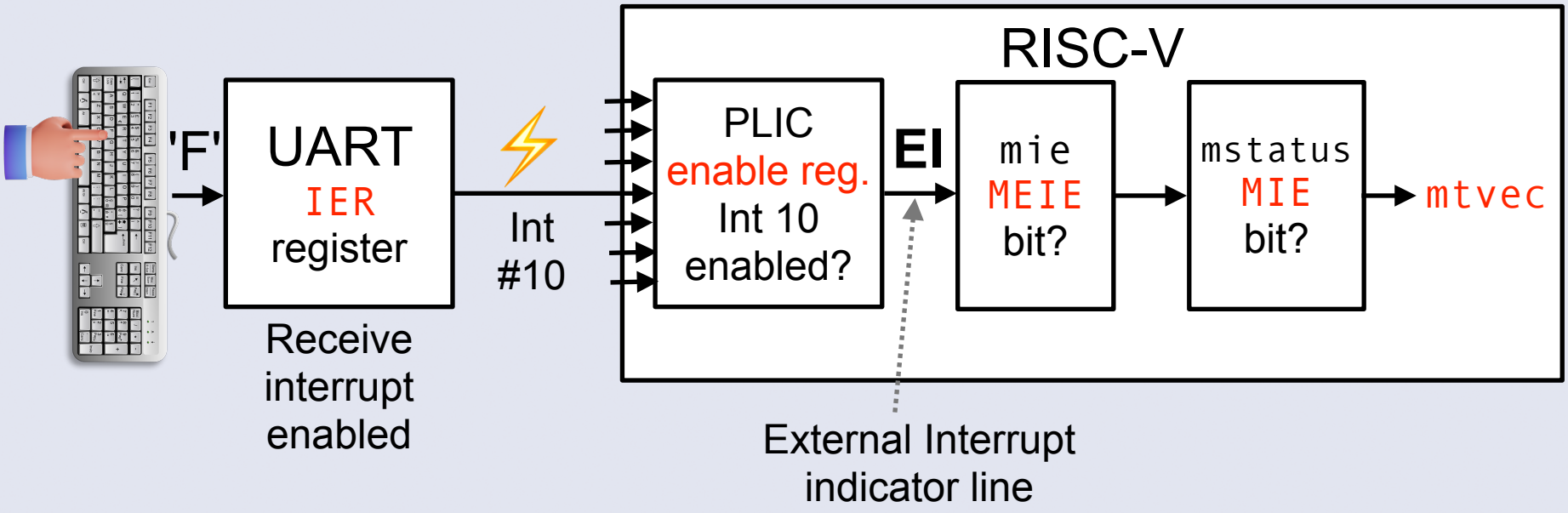
- The platform-level interrupt controller (**PLIC**) routes all signals through one pin on the CPU–the **EI** (external interrupt) pin
  - This pin can be enabled via the machine external interrupt enable (`meie`) bit in the `mie` register
- Whenever we see that this pin has been triggered (an external interrupt is pending), we can query the PLIC to see what caused it
- We can configure the PLIC to prioritize interrupt sources or to completely disable some sources, while enabling others

The way of an interrupt

Universität Bamberg

An external interrupt has to go through four steps to arrive at the CPU:

- UART interrupt enable in UART IER
- PLIC interrupt enable in PLIC enable register
- MEIE enable in `mie` CSR
- Global interrupt enable in `mstatus` CSR



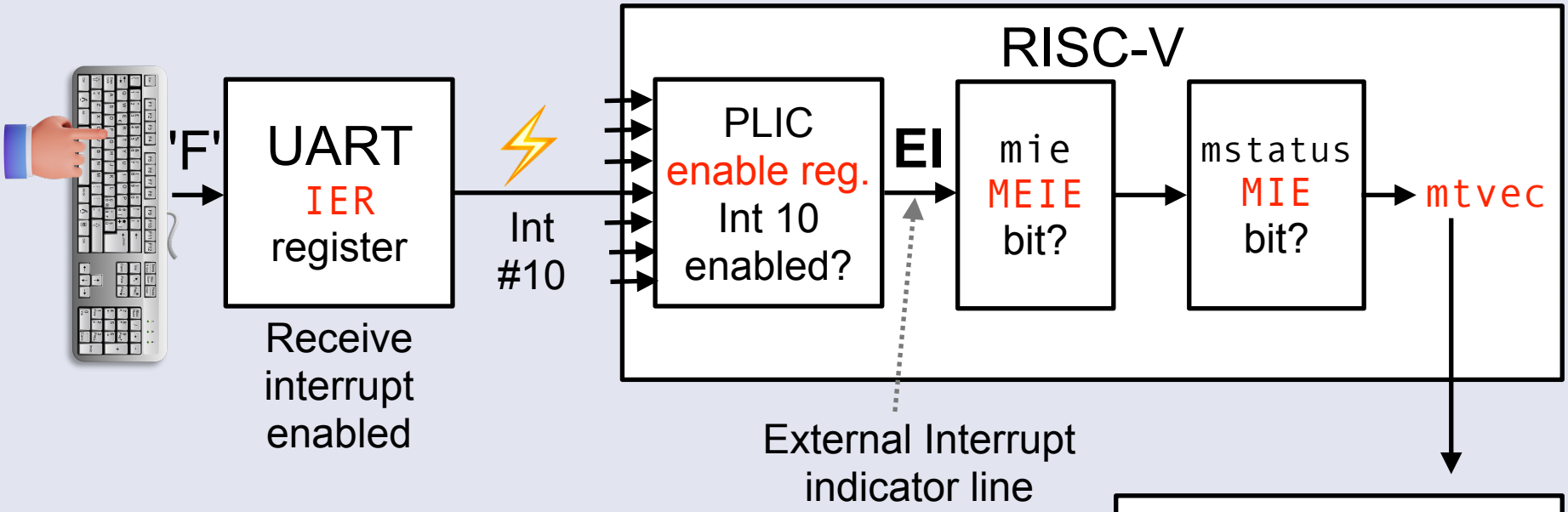
PLIC registers and initialization

Initializing the PLIC requires:

- Init PLIC threshold value (e.g. to 0)
- Set M-mode enable bit for given interrupt to 1
- Init M-mode priority of given interrupt to value >= threshold

Register	Address	Description
Priority	0x0c00_0000	Sets the priority of a particular interrupt source
Pending	0x0c00_1000	Contains a list of interrupts that have been triggered (are pending)
Enable	0x0c00_2000	Enable/disable certain interrupt sources (1 bit per int.)
Threshold	0x0c20_0000	Sets the threshold that interrupts must meet before being able to trigger (1 32-bit word per int.)
Claim (read)	0x0c20_0004	Returns the next interrupt in priority order
Complete (write)	0x0c20_0004	Completes handling of a particular interrupt

Handling a PLIC interrupt



Steps to handle an UART interrupt:

- **check** for `mstatus` bit 63 = '1' (async exception)
- **claim** the interrupt: read PLIC `claim` reg. → 10
- **handle** the int.: read UART `ISR` → RXRD bit
- **complete**: write 10 to PLIC `complete` register

ex.S→exception

```
check mstatus
if (mstatus bit 63==1)
  read claim → irq = 10
  if (irq == 10) {
    read UART→ISR
    if (RXRD) read RDR
    write irq → complete
```

Interrupt handling pseudo code


```
exception(…) {
  if (mcause bit 63==1) {              // async interrupt?
    if (mcause bit 7…0==11) {          // M-mode external interrupt?
      uint64 irq = PLIC->claim;        // get highest prio interrupt nr
      switch (irq) {
        case UART_IRQ:                 // #define UART_IRQ 10
          uint8 uart_irq = UART->ISR; // read UART interrupt source
          if (RXRD) {c = UART->RDR;}  // read received char from UART
          break;                       // (clears UART interrupt)
        case …: …; break; // handle other external interrupts
        default: print("Unknown interrupt!"); break;
    }
    PLIC->complete = irq;    // announce to PLIC that IRQ was handled
  } else {
    // sync interrupt: ecalls, access faults…
  }
  // set mepc etc.
}
```

Conclusion

- Problems with **deadlocks** and **livelocks**
  - livelocks are the bigger problem of the two
- For a dead/lifelock, four conditions have to occur simultaneously:
  - **Exclusive allocation**, **hold and wait**, **no preemption** of resources
  - Circular waiting of the processes requesting the resources
- Handling dead/lifelocks implies:
  - **prevent**, **avoid**, **detect/resolve**
  - the discussed approaches can also be combined
- External interrupts on RISC-V
  - Programming the interrupt chain
  - PLIC configuration and interrupt enabling
  - Handling external interrupts

References

1. *RISC-V Platform-Level Interrupt Controller Specification*, https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc