TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf Spinczyk, Universität Osnabrück

Interrupts – Software

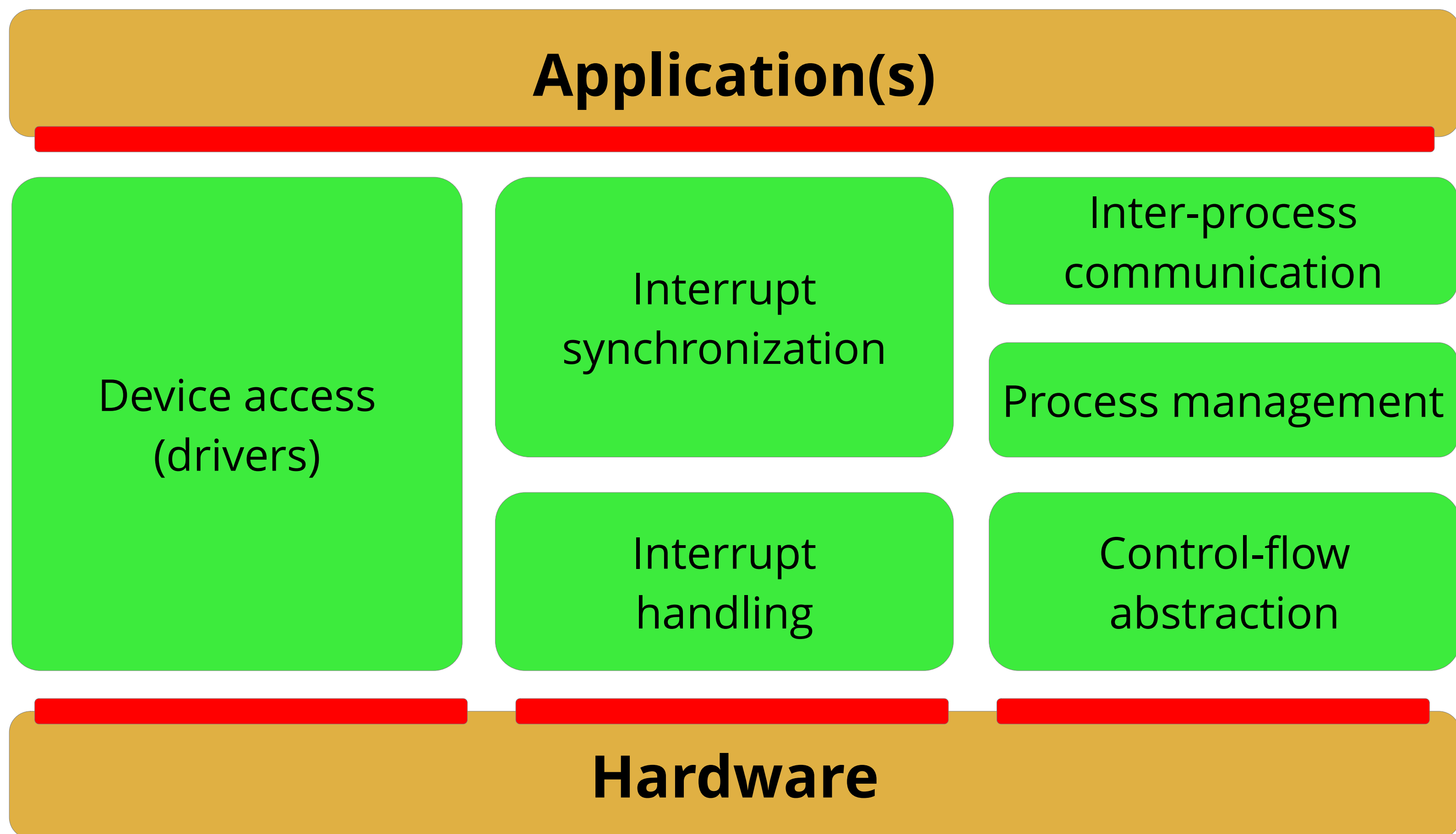
https://tud.de/inf/os/studium/vorlesungen/betriebssystembau

HORST SCHIRMEIER

TECHNISCHE UNIVERSITÄT DRESDEN

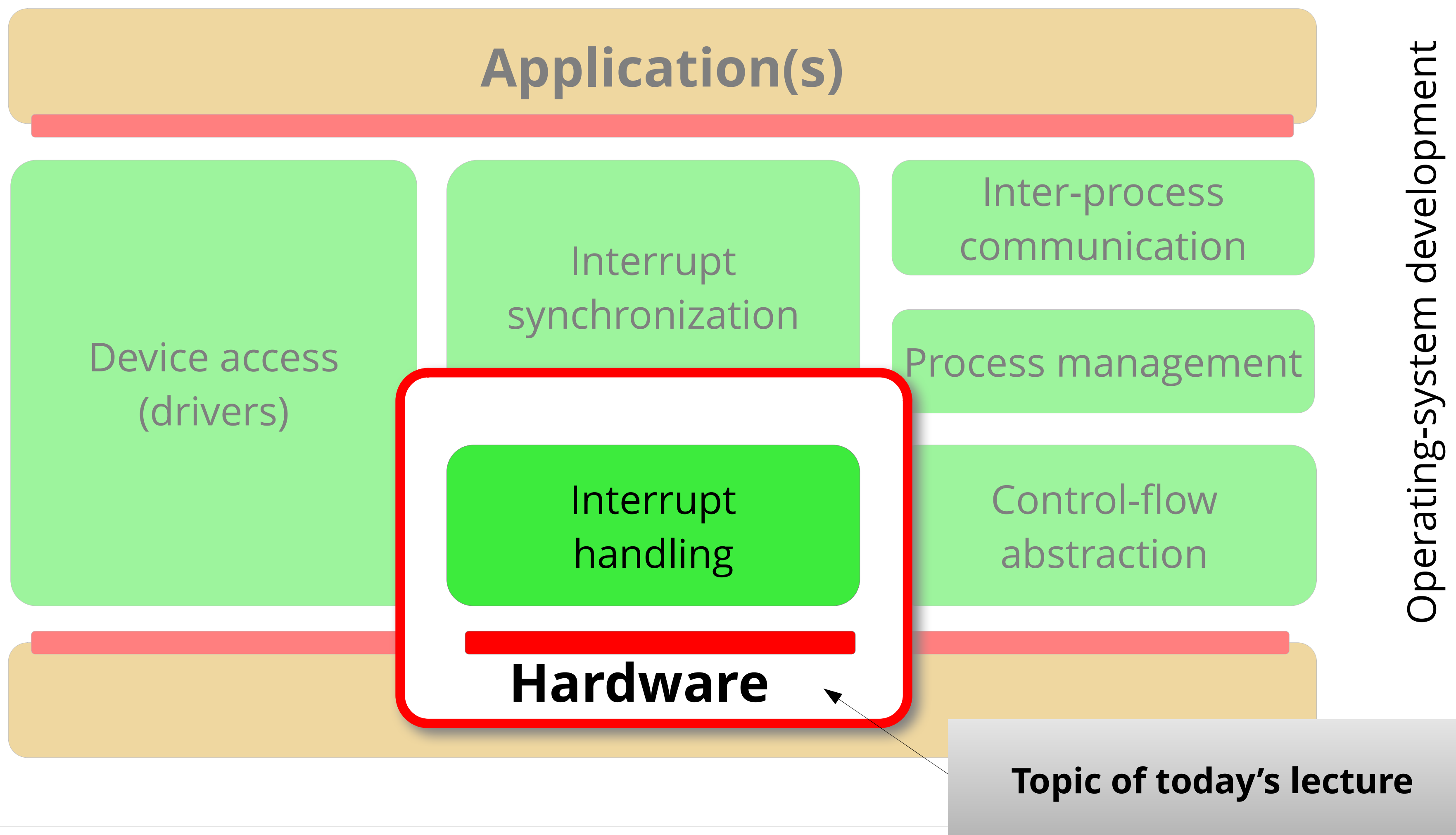
Overview: Lectures

Structure of the "OO-StuBS" operating system:



Overview: Lectures

Structure of the “OO-StuBS” operating system:



Overview

- Terminology and Assumptions
- Saving State
- Modifying State
- Synchronization Techniques
- Summary

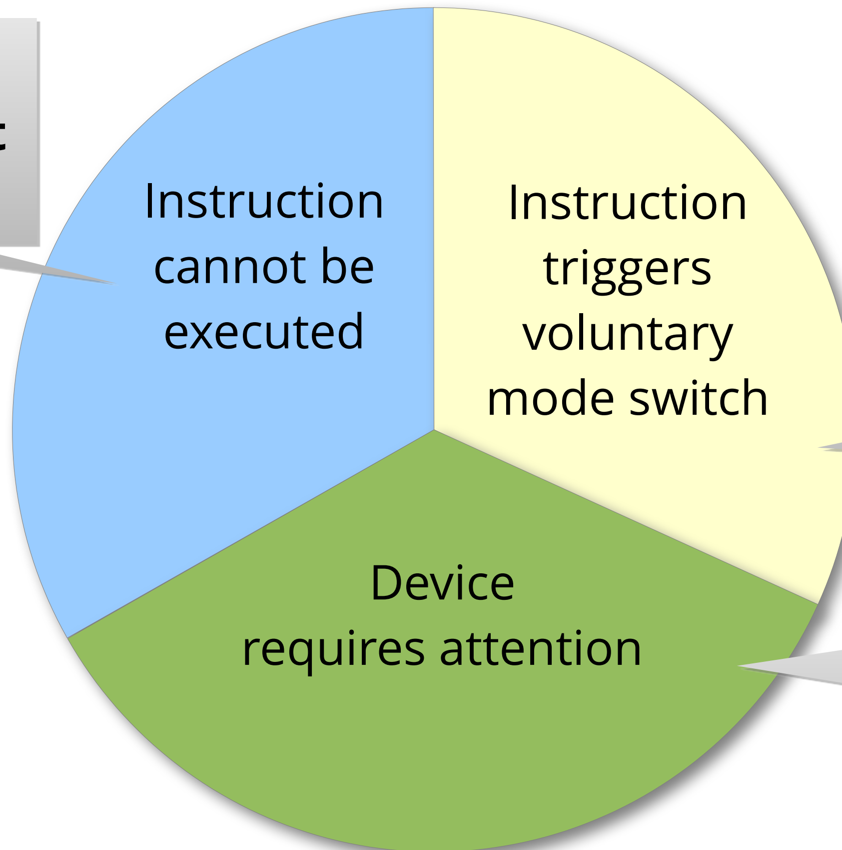
Overview

- **Terminology and Assumptions**
- Saving State
- Modifying State
- Synchronization Techniques
- Summary

Terminology

- Term(s) are understood differently …
  - For disambiguation, we take a technical perspective.

- **Page fault**
- **Protection fault**
- **Division by 0**



In general, we can distinguish three cases

- **System call**
- **Breakpoint instruction**

- **Timer alarm**
- **Key pressed**
- **“NMI”**

TECHNISCHE UNIVERSITÄT DRESDEN

Terminology: Intel IA-32

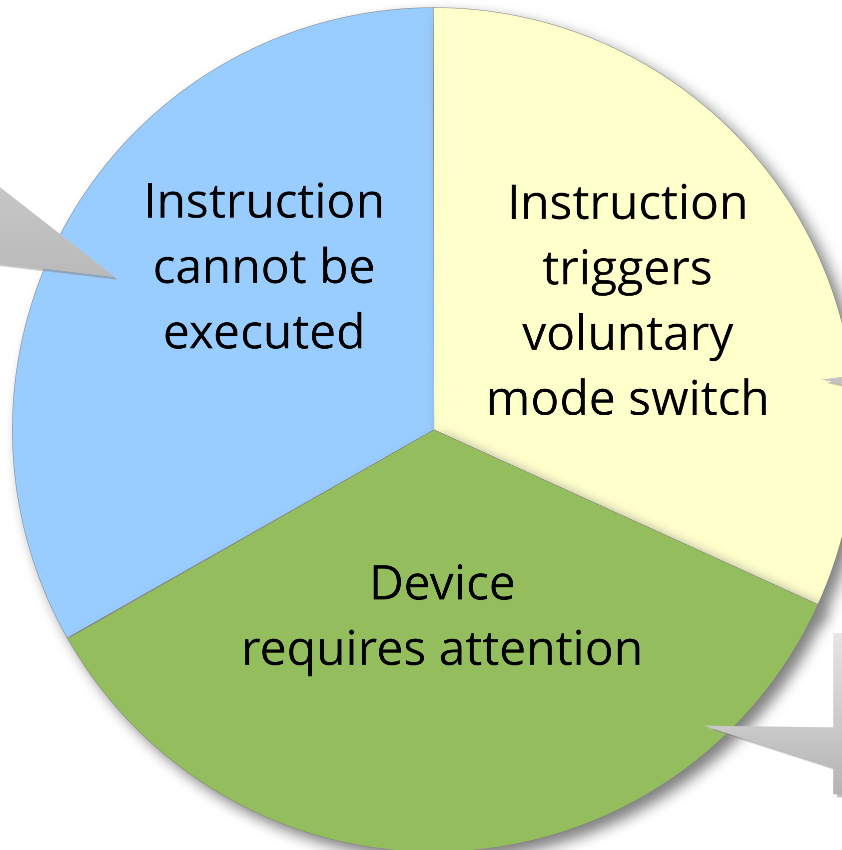
- Term(s) are understood differently …
  - For disambiguation, we take a technical perspective.

(software-generated) **exceptions**
- fault, trap, or abort

Intel IA-32 architecture (x86): **exceptions** and **interrupts**

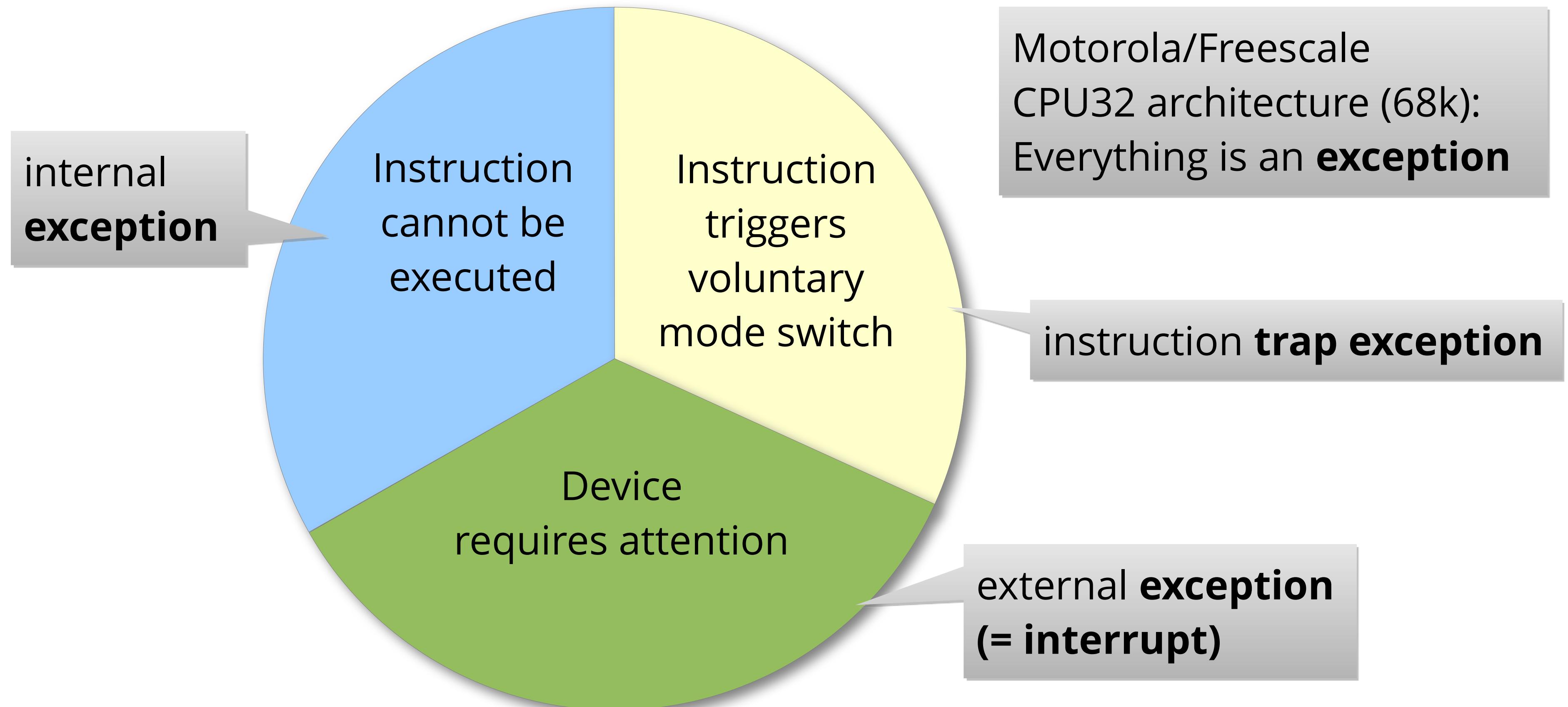
software(-generated) **interrupts**

external (hardware-generated) **interrupts**



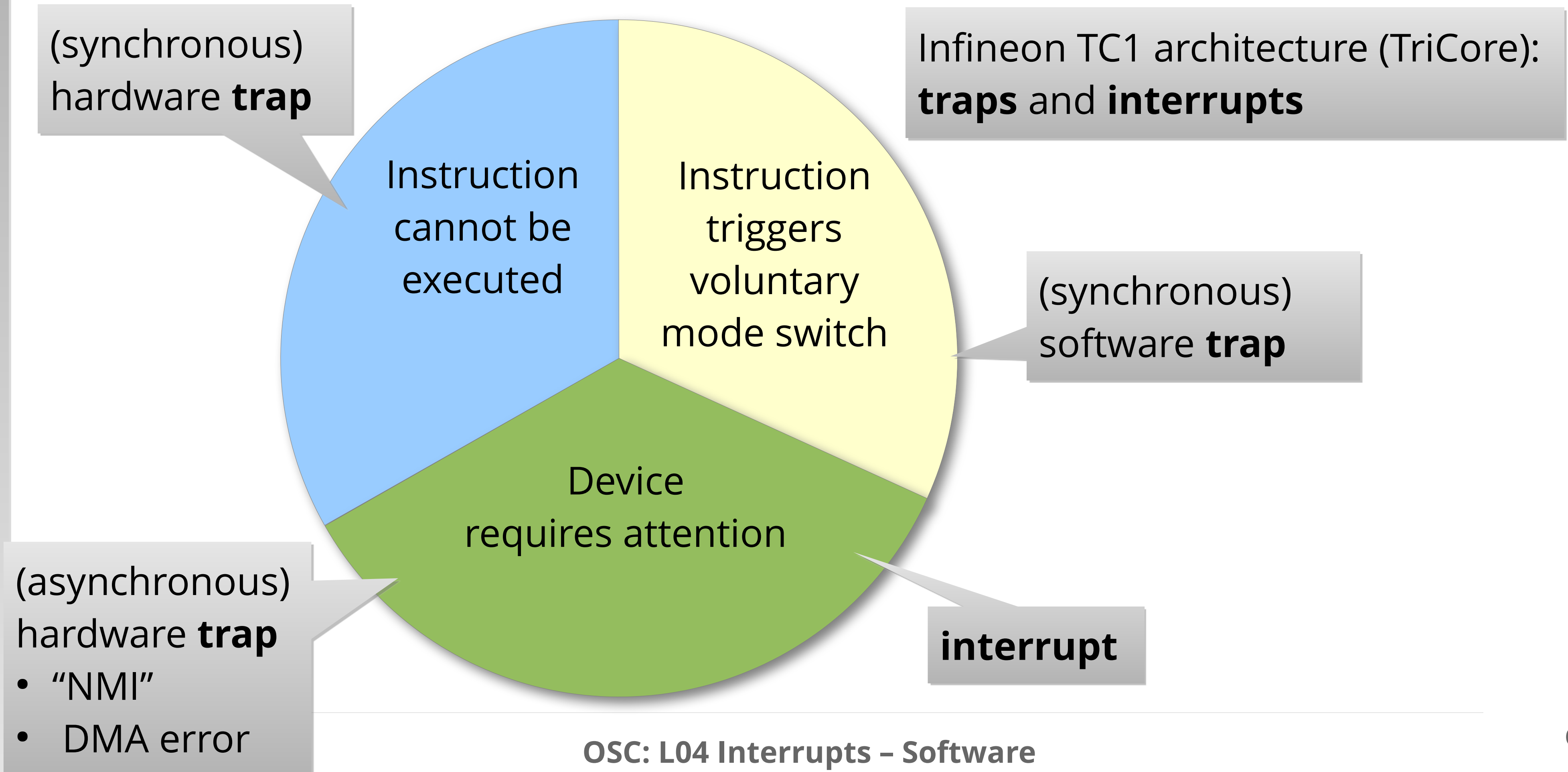
Terminology: Motorola/Freescale CPU32

- Term(s) are understood differently ...
  - For disambiguation, we take a technical perspective.



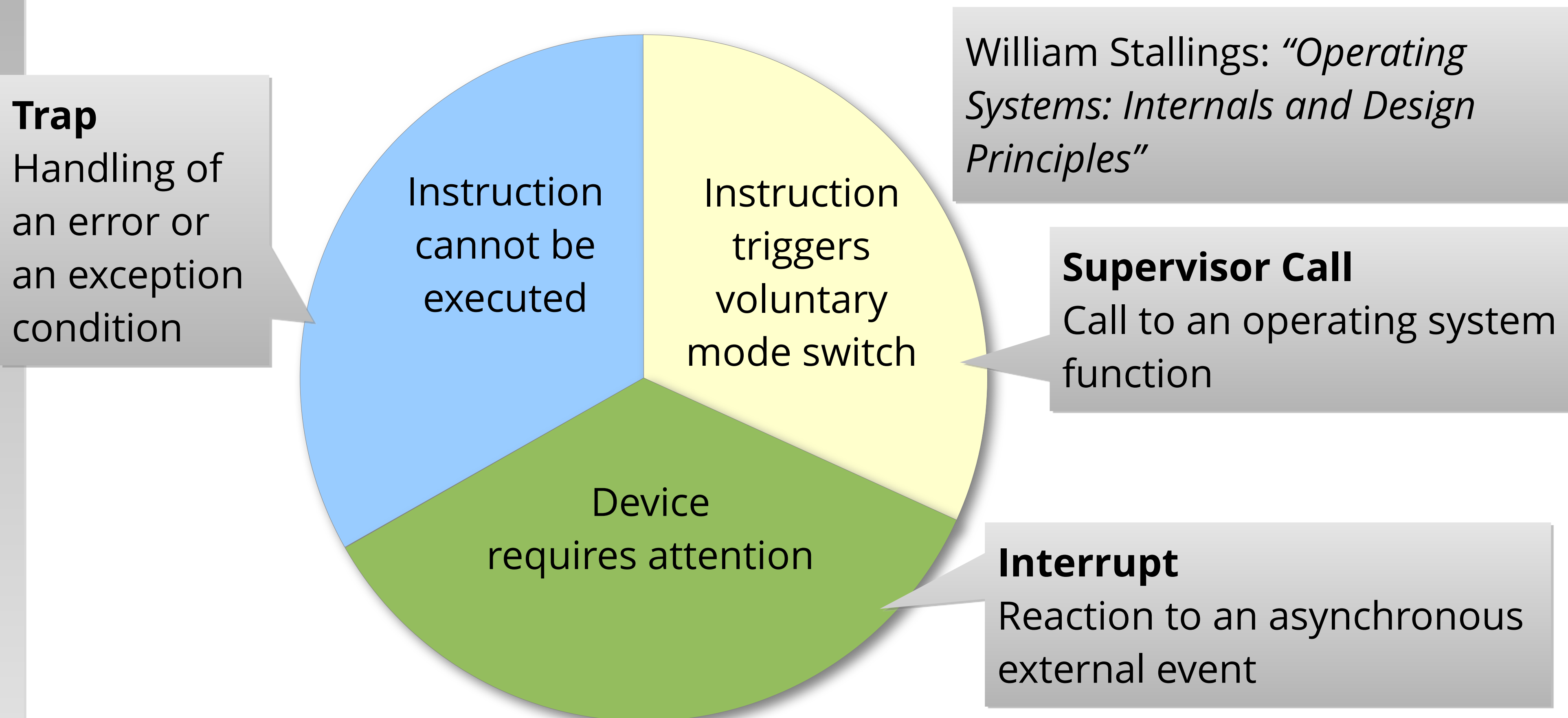
Terminology: Infineon TC1

- Term(s) are understood differently ...
  - For disambiguation, we take a technical perspective.



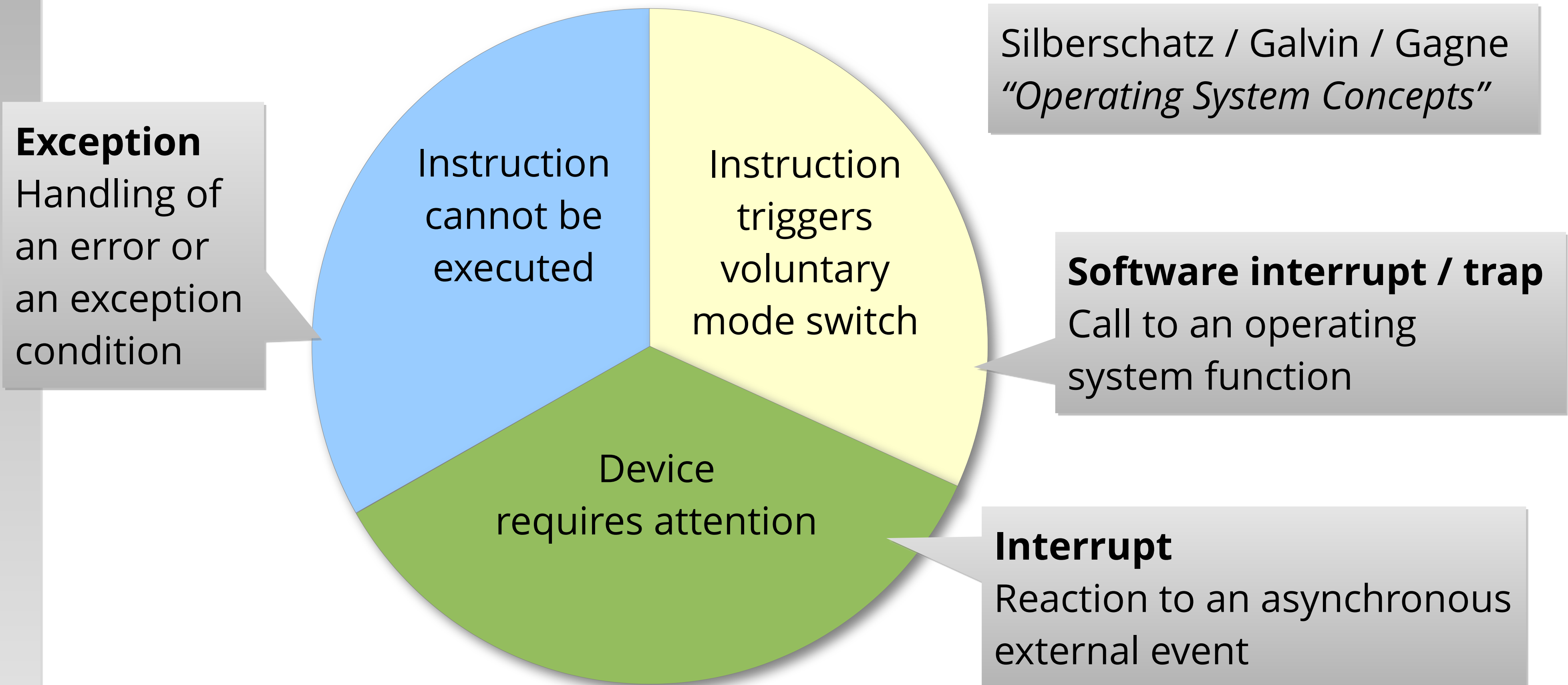
Terminology: Literature (Stallings)

- Term(s) are understood differently …
  - For disambiguation, we take a technical perspective.



Terminology: Literature (Silberschatz)

- Term(s) are understood differently …
  - For disambiguation, we take a technical perspective.



Terminology: Literature (Tanenbaum)

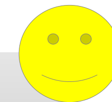
- Term(s) are understood differently ...
  - For disambiguation, we take a technical perspective.

Andrew S. Tanenbaum: *“Operating Systems Design and Implementation”* (3rd ed., 2006)

Instruction

Instruction

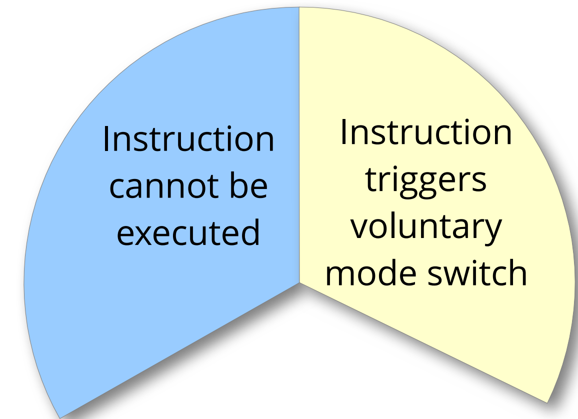
“Interrupts are an unpleasant fact of life”



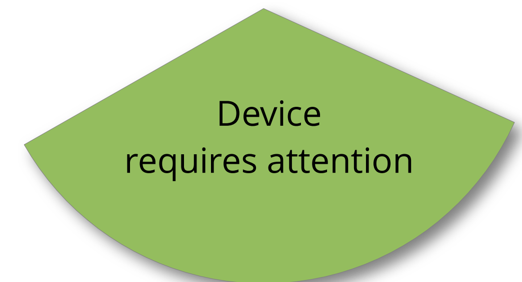
Device
requires attention

Terminology: Understanding in OSC

- **"Trap"**
  - Triggered by an **instruction**
    - … including the `trap` or `int` instruction for system calls
    - Undefined result (e.g. division by 0)
    - Hardware problem (e.g. bus error)
    - OS must do something (e.g. page fault)
    - Invalid instruction (e.g. programming error)
  - Properties:
    - often predictable, often reproducible
    - **Restart or abort** the triggering activity



- **"Interrupt"**
  - Triggered by **hardware**
    - Hardware requires attention by software
      (Timer, Keyboard controller, Hard-disk controller, …)
  - Properties
    - not predictable, not reproducible
    - Usually **resume** the interrupted activity



Basic Assumptions

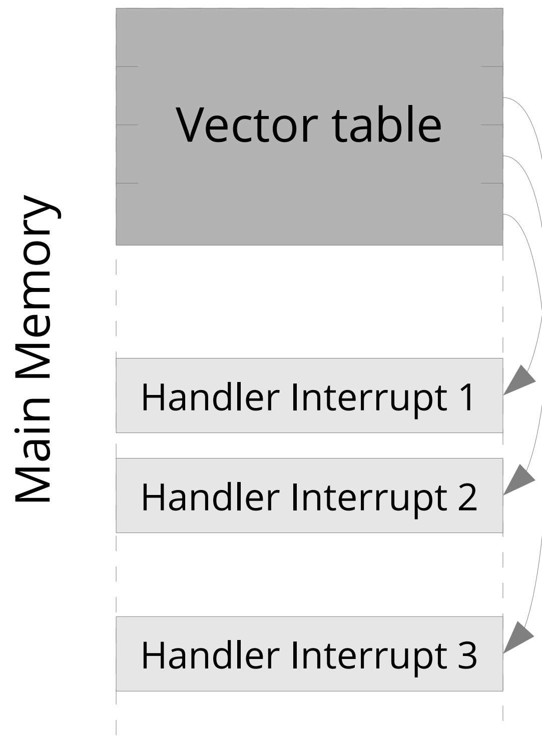
In the discussion on interrupt handling, we assume the following:

1. The CPU **automatically** starts the handler routine.
2. Interrupt handling takes place in **supervisor mode**.
3. The interrupted program can be **resumed**.
4. Machine instructions are **atomic**.
5. Interrupt handling can be **disabled/suppressed**.

Assumption: Handler Routine

1. The CPU automatically starts the handler routine.

- Necessitates dispatch to a handler
- Determine which device triggered the interrupt



Variants:
- Register contains vector-table start address
- Table entries contain code
- Programmable “event controller” handles interrupt in hardware
- Table contains descriptors
- Handler routine has own process context

Assumption: Supervisor mode

2. Interrupt handling takes place in supervisor mode.

- Interrupts are the only mechanism to preempt non-cooperative applications.
- Only the OS may access devices without restrictions.
  - Before interrupt handling, the CPU switches to the privileged supervisor mode.

Variants:

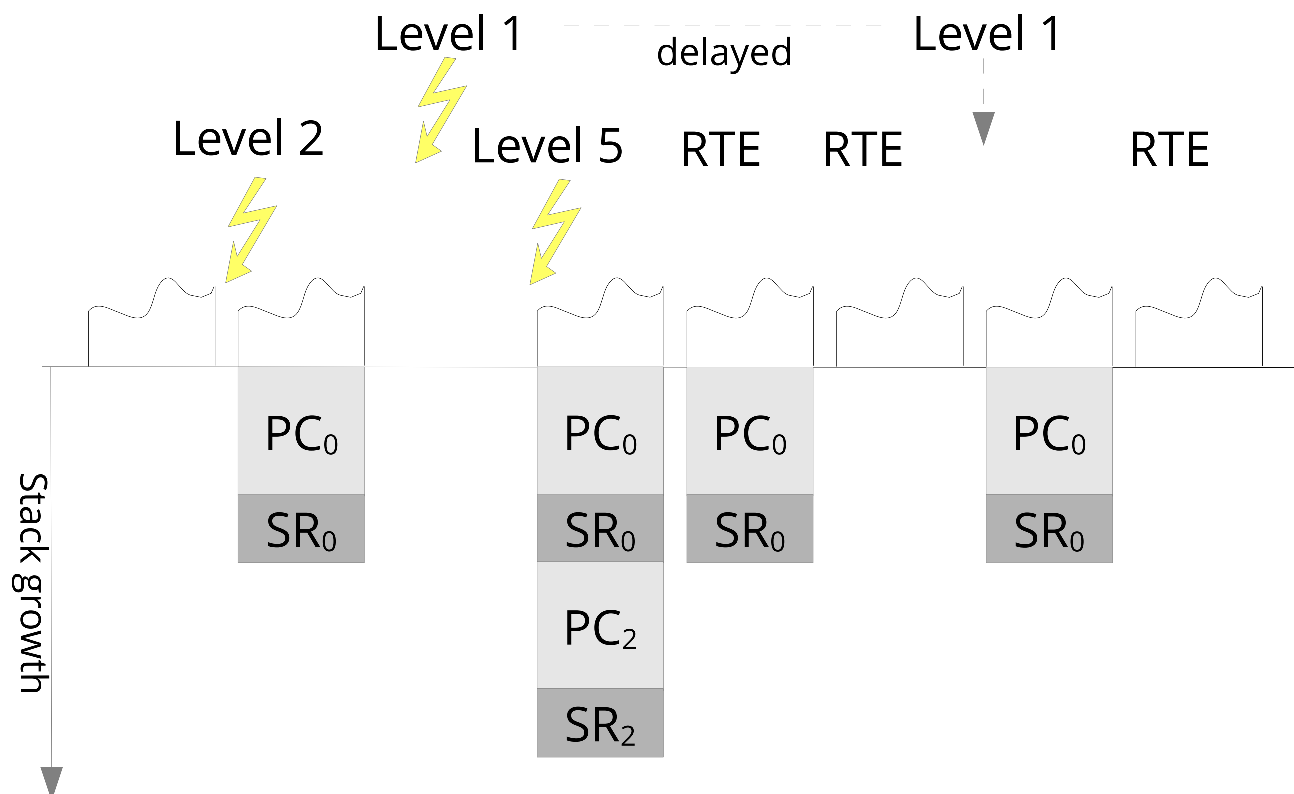
- For **16-bit CPUs**, a subdivision into user and supervisor mode is rather the exception.
- For **8-bit CPUs** (or smaller) this subdivision does not exist.

TECHNISCHE UNIVERSITÄT DRESDEN

Assumption: State Save

3. The interrupted program can be resumed.

- necessary state is automatically saved
- possibly nested, requires a stack



Variants:

- More information on cause/trigger on the stack
- No priorities
- Special “interrupt stack”
- State save in registers

Assumption: Atomic Behavior

4. Machine instructions are atomic.

- Defined CPU state when handler routine starts
- State restorable

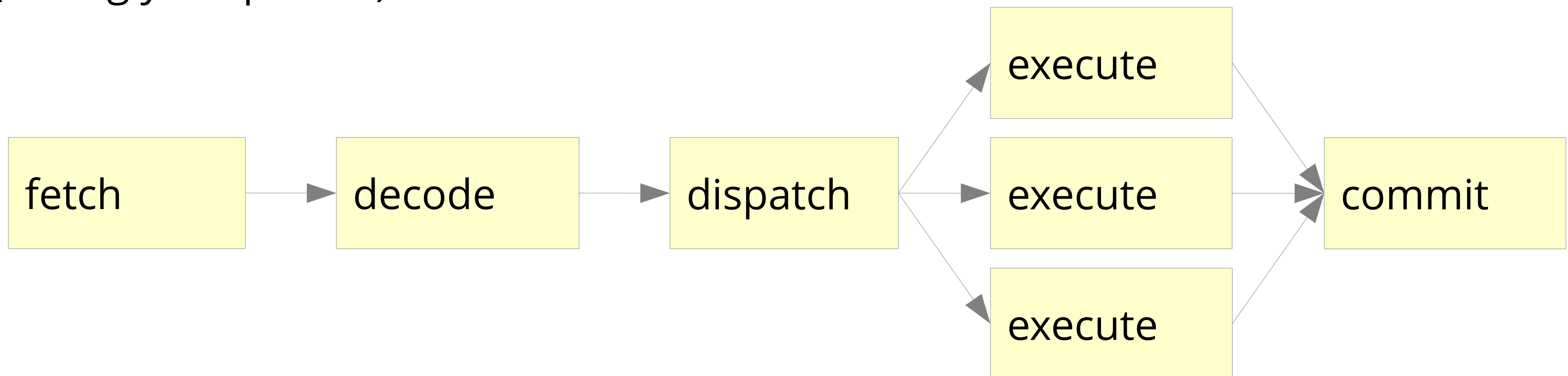
- Trivial for CPUs with classic von Neumann (fetch-decode-execute) cycle
- Nontrivial for modern CPUs:
  - Pipelining: Instructions must be flushed (thrown away)
  - Superscalar CPUs: Must leave pipeline in well-defined state

Assumption: Atomic Behavior

4. Machine instructions are atomic.

Instruction execution in superscalar CPUs:

(strongly simplified!)



Ideally, all stages are always in use, i.e. multiple instructions are executed in parallel. When should we check whether an IRQ has been issued?

Assumption: Atomic Behavior

4. Machine instructions are atomic.

- Although nontrivial, most CPUs implement **precise interrupts**:
  - *„All instructions **preceding** the instruction indicated by the saved program counter **have been executed** and have modified the process state correctly.“*
  - *„All instructions **following** the instruction indicated by the saved program counter are **unexecuted** and have not modified the process state.“*
  - *„If the interrupt is caused by an **exception condition** raised by an instruction in the program, the saved program counter **points to the interrupted instruction**. The interrupted instruction may or may not have been executed, depending on the definition of the architecture and the cause of the interrupt. Whichever is the case, the interrupted instruction has either completed, or has not started execution.“*

J. E. Smith and A. R. Pleszkun,
„Implementing Precise Interrupts in Pipelined Processors“,
IEEE Transactions on Computers, Vol. 37, No. 5, 1988

Assumption: Interrupt Suppression

5. Interrupt handling can be disabled/suppressed.

- Examples:
  - Motorola 680x0: according to priority
  - Intel x86: globally with `sti`, `cli`
  - Interrupt Controller: each source individually
- Additionally: **Automatic suppression** by the CPU before starting the handler routine
  - Interrupts not predictable (theoretically arbitrarily frequent!)
  - Without this automatism, a **stack overflow** would be possible

Assumption: Interrupt Suppression

5. Interrupt handling can be disabled/suppressed.

- The hardware suppresses ...
  - across the board **all interrupts** (very restrictive)
  - interrupts with **lower or same priority** (less restrictive)
    - preference for particular devices
- Further alternatives based on software, e.g. in Linux:
  - Suppress interrupts that are **currently being handled**
    - low reaction latency without preference for individual devices

Overview

- Terminology and Assumptions
- **Saving State**
- Modifying State
- Synchronization Techniques
- Summary

Saving State

- A computer's state is large:
  - All CPU registers
    - Instruction pointer, stack pointer, general-purpose registers, status register, ...
  - Main-memory contents, caches
  - I/O registers / ports, hard-disk contents, ...
- Any state the interrupted program
  does **not expect to asynchronously change** ...
  - must **not be modified** in an interrupt handler
    or
  - must be **saved, and restored** afterwards.
- Depending on its architecture/type, the CPU **automatically saves** ...
  - a minimum number of bytes (e.g., only instruction pointer and status register)
    or
  - all registers

TECHNISCHE UNIVERSITÄT DRESDEN

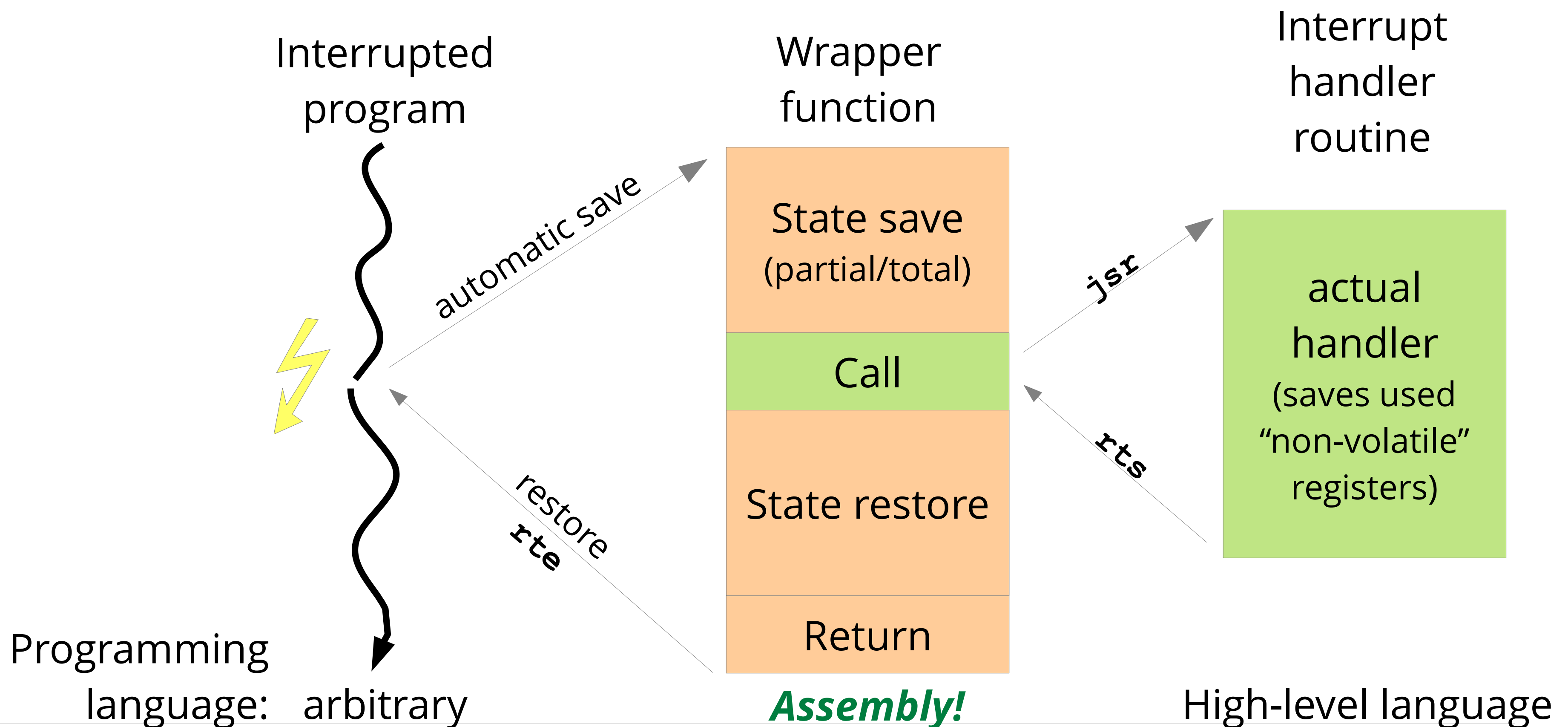
State-Saving Concepts

- **Total save**
  - Handler routine saves all registers that were not automatically saved
    – probably saves too much
    + saved state easily accessible (one coherent data structure)
- **Partial save**
  - Handler only saves registers that **1) are modified** in the interrupt handler* and **2) are not saved/restored by other parts** of the handler*
    (*) or the functions it calls directly or indirectly
  - Feasible if actual handler is implemented in a high-level language, e.g. C/C++
    + only state that actually gets modified is saved/restored
    + possibly less instructions for save/restore necessary
    – saved state is "scattered" (not in one place, hard to access)

TECHNISCHE UNIVERSITÄT DRESDEN

Transition to High-Level Language

- Minimize non-portable machine code
- Do actual interrupt handling in high-level language function



Transition to High-Level Language

- Minimize non-portable machine code
- Do actual in[...] language function

Example: MC680x0

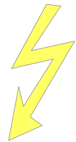
Total save:

```
moveml d0-d7/a0-a6,sp@-
...
moveml sp@+,d0-d7/a0-a6
```

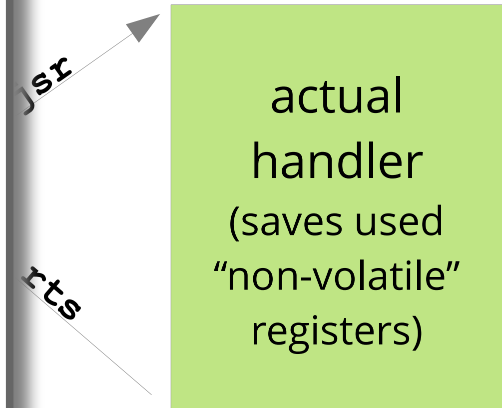
Partial save:

```
moveml d0-d1/a0-a1,sp@-
...
moveml sp@+,d0-d1/a0-a1
```

Inter[...] pr[...]



Interrupt handler routine



Programming language: arbitrary

Assembly!

High-level language

Excursus: Volatile and Non-Volatile Registers

- Partitioning of CPU registers **relevant for the (C/C++) compiler** in the context of **function calls**
  - **non-volatile** (aka *callee-saved*) registers
    - Compiler guarantees that the stored **value is conserved across function calls**
    - **Callee** (=called function) is responsible: If it uses the register, it saves/restores value.
  - **volatile** (aka *caller-saved* or *scratch*) registers
    - If the **caller** (=calling function) still needs the value after a function call, it **must save/restore the register itself**.
    - Usually used for intermediate results
- Usually defined in a standard all compilers adhere to (why?)
  - e.g. x86-64 ("System V AMD64 ABI"):
    - **non-volatile**: `rbx`, `rbp`, `r12-r15`
    - all others are **volatile**: `rax`, `rcx`, `rdx`, `rdi`, `rsi`, `r8-r11`, `eflags`, FPU/SSE registers, …

Restoring State

- As its last duty, wrapper must restore saved register contents
  - … and must not again modify them afterwards!
- A special instruction, e.g. **`rte`** (68k) or **`iret`** (x86-64) completes the restore procedure:
  - Reads automatically saved state from supervisor stack
  - Sets the saved CPU mode (user/supervisor), jumps to saved address

> The OS can **modify saved state** before **`rte`**/**`iret`**.
> This is useful for running OS code in user mode.

Overview

- Terminology and Assumptions
- Saving State
- **Modifying State**
- Synchronization Techniques
- Summary

State Modifications …

- are the **main purpose of interrupt handling**, e.g.
  - Inform device driver about completed I/O operation
  - Notify scheduler that time slice has run out
- must be performed with care:
  - Interrupts can occur **at any time**
  - **critical:** Data/data structures shared between regular control flow and interrupt handling

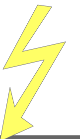
Example 1: System Time

- Timer interrupt is used to increment global system time
  - e.g. once per second
- An application can read the system time using the OS function **`time()`**

```
/* global var. with current time */
extern volatile time_t global_time;
```

```
/* Read current time */
time_t time () {
  return global_time;
}
```

```
/* Interrupt handler */
void timerHandler () {
  global_time++;
}
```



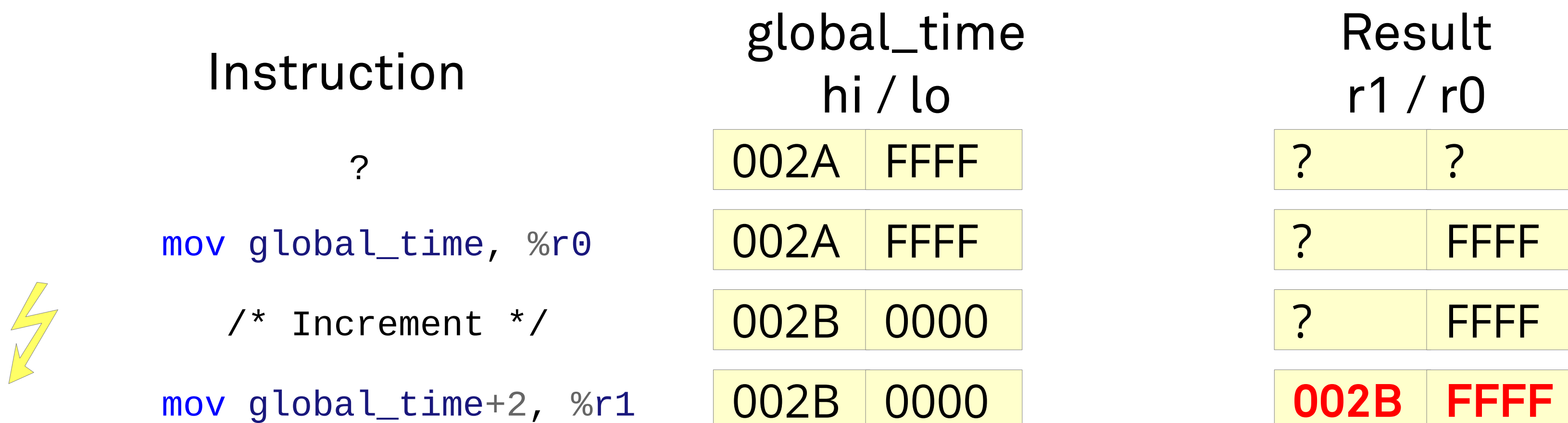
Example 1: System Time

- Here, a possible bug is hiding in plain sight …
  - Reading global_time is not necessarily an atomic operation!

```
; time() on a 32-bit CPU
mov global_time, %eax
```

```
; 16-bit CPU (little endian)
mov global_time, %r0   ; lo
mov global_time+2, %r1 ; hi
```

- **Critical:** Interrupt **between** the two read instructions for the 16-bit CPU



Example 1: System Time

- Here, a possible bug is hiding in plain sight ...
  - Reading global_time is not necessarily an atomic operation!

```
; time(
mov glo
                                                  endian)
                                                     ; lo
                                                 r1 ; hi
```

- **Critical** ... for the 16-bit C...

Problem

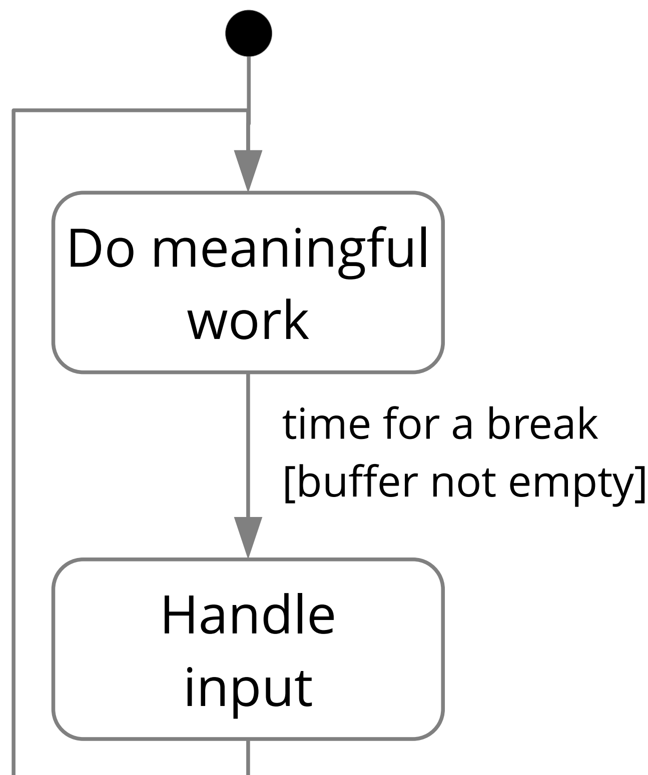
Every 18.2 hours, the system time can appear to be 18.2 hours ahead (for a short moment). Unfortunately, this is **not reliably reproducible**.

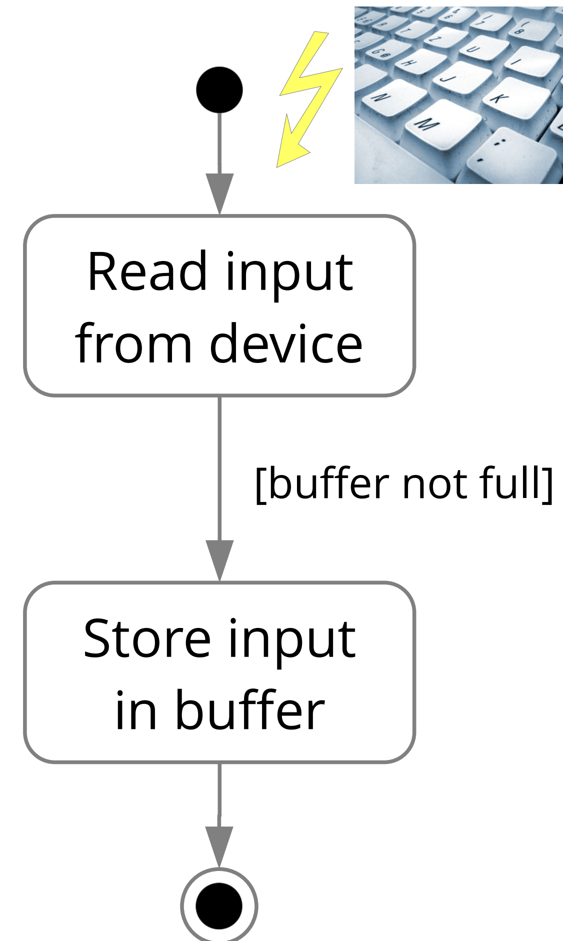
Instruction	hi	lo	r1	r0
?	002A	FFFF	?	?
`mov global_time, %r0`	002A	FFFF	?	FFFF
`/* Increment */`	002B	0000	?	FFFF
`mov global_time+2, %r1`	002B	0000	**002B**	**FFFF**

TECHNISCHE UNIVERSITÄT DRESDEN

Example 2: Ring Buffer

- Interrupts were introduced to **avoid busy waiting** for input
  - While an application is doing meaningful work, the interrupt handler can store input in a buffer.





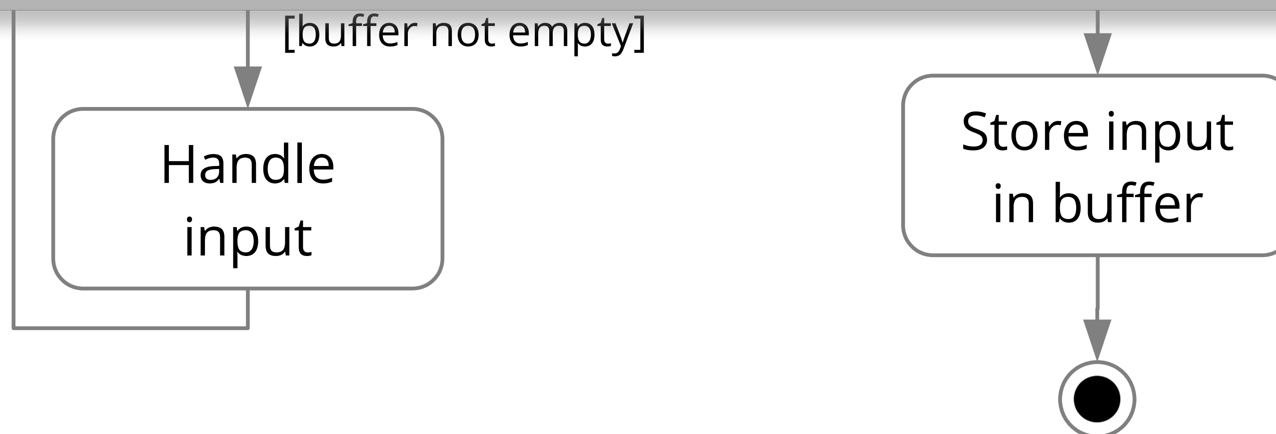
Example 2: Ring Buffer

- Interrupts were introduced to **avoid busy waiting** for input
  - While an application is doing meaningful work, the interrupt handler

Problem #1:

If input is not handled/consumed fast enough, the **buffer can fill up**. The interrupt handler routine cannot store further input there. This **input is lost**.

ot full]



Example 2: Ring Buffer

Problem #2: The buffer implementation itself is critical …

```
// Bounded ring buffer in C++
class BoundedBuffer {
  char buf[SIZE]; int occupied; int nextin, nextout;
public:
  BoundedBuffer(): occupied(0), nextin(0), nextout(0) {}
  void produce(char data) {        // Interrupt handler:
    int elements = occupied;       // Local copy of element counter
    if (elements == SIZE) return;  // Full? Drop this element.
    buf[nextin] = data;            // Write element
    nextin++; nextin %= SIZE;      // Advance write index
    occupied = elements + 1;       // Increase element counter
  }
  char consume() {                 // Regular control flow:
    int elements = occupied;       // Local copy of element counter
    if (elements == 0) return 0;   // Buffer empty, no result
    char result = buf[nextout];    // Read element
    nextout++; nextout %= SIZE;    // Advance read index
    occupied = elements – 1;       // Decrease element counter
    return result;                 // Return result
} };
```

TECHNISCHE UNIVERSITÄT DRESDEN

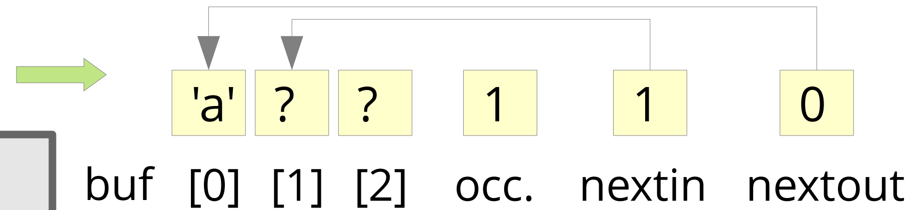
Example 2: Ring Buffer

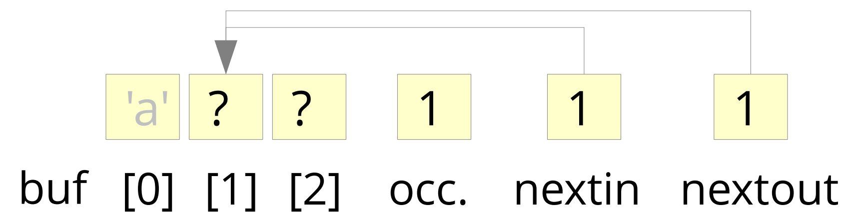
Execution State

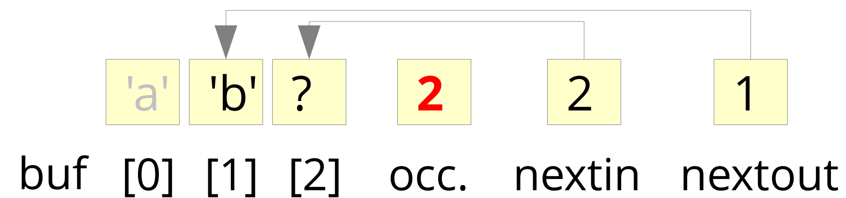
```
char consume() {
  int elements = occupied; // 1
  if (elements == 0) return 0;
  char result = buf[nextout]; // 'a'
  nextout++; nextout %= SIZE;
```

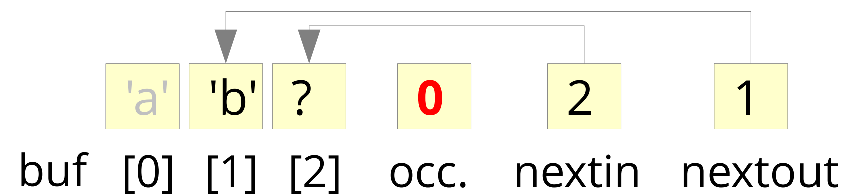
```
void produce(char data) { // 'b'
  int elements = occupied; // 1!
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1; // 2
}
```

```
  occupied = elements – 1; // 0
  return result; // 'a'
}
```







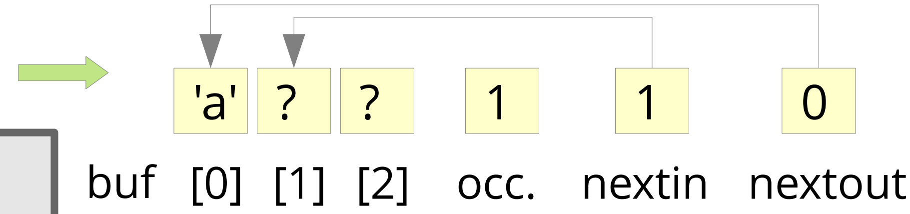


Example 2: Ring Buffer

Execution

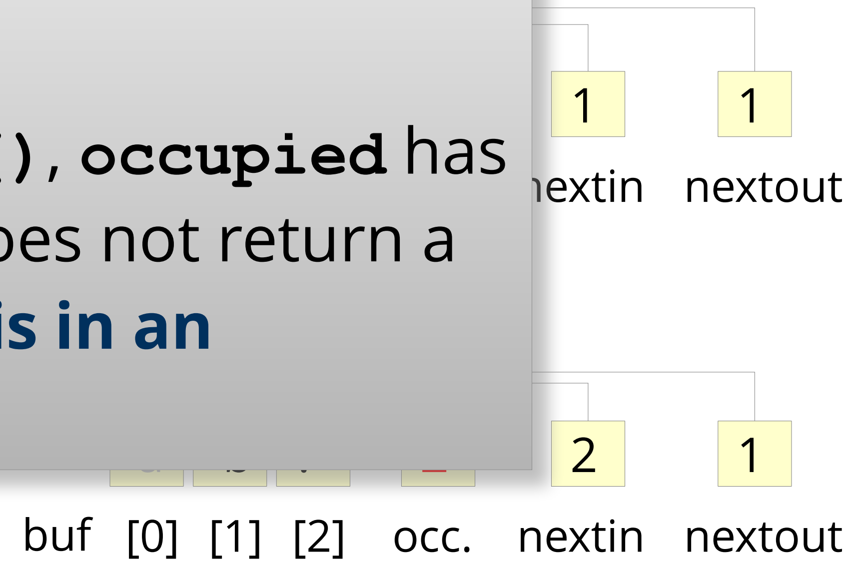
State

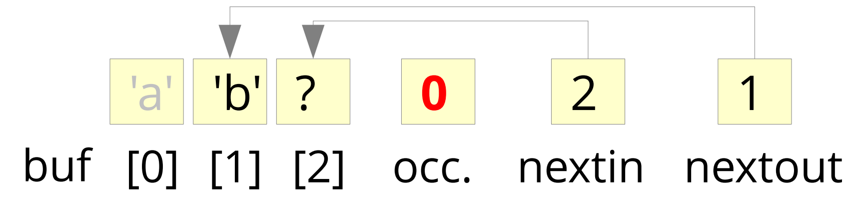
```
char consume() {
   int
   if (
   char
   next

void p
   int
   if (
   buf[
   next
   occupied = elements + 1; // 2
}

   occupied = elements - 1; // 0
   return result; // 'a'
}
```



Problem #2: Result

In the next call to **consume()**, **occupied** has a value of 0. The function does not return a result. **The data structure is in an inconsistent state.**





State Modification: Analysis

- Even **single variable assignments** are not necessarily atomic.
  - Depends on CPU type, compiler, code optimizations
- Buffer memory is finite
  - Handler routine cannot wait! (Why?)
  - Data might get lost
- Buffer data structure can “break”, caused by ...
  - **inconsistent intermediate states** during modifications by regular control flow
  - state modifications **while reading** (inconsistent copy!)
  - modifications based on a **copy** that does not correspond to original anymore
- The problem is not symmetric:
  - Regular control flow does not “interrupt” the interrupt handler
  - We can exploit this fact!

Overview

- Terminology and Assumptions
- Saving State
- Modifying State
- **Synchronization Techniques**
- Summary

"Hard" Synchronization

- By suppressing interrupts, we can avoid **race conditions**.
  - Operations on shared data are made atomic this way.

```
  char consume() {                         // Regular control flow:
    disable_interrupts();                  // Inhibit interrupts
    int elements = occupied;               // Local copy of element ctr
    if (elements == 0) return 0;           // Buffer empty, no results
    char result = buf[nextout];            // Read element
    nextout++; nextout %= SIZE;            // Advance read index
    occupied = elements – 1;               // Decrease element counter
    enable_interrupts();                   // Allow interrupts
    return result;                         // Return result
}
```

- **Problems:**
  - Hazard of losing interrupt requests
  - High and difficult to predict **"interrupt latency"**

More Techniques in the next Lecture

- "Smart" (optimistic) solutions
  - Clever data-structure choice
    - as few as possible **shared elements**
    - work with **weak** consistency conditions
  - Optimistic approach
    - Usually we **aren't interrupted** in the critical section
    - However, if we are, examine the damage and **repair**
    - Potentially **repeat/restart** the operation
- Prologue/epilogue model
  - Partition the interrupt handler in two phases
    - **Delay the critical part** by a software mechanism
    - **Low-latency reaction** still possible

Overview

- Terminology and Assumptions
- Saving State
- Modifying State
- Synchronization Techniques
- **Summary**

Summary

- **Correct interrupt handling** is one of the hardest tasks in operating-system construction
  - Source of non-determinism
    - … both a blessing and a curse!
  - State save on register level
    - Assembly programming!
    - Dependence on compiler (e.g. volatile/non-volatile registers)
  - Different models (priorities etc.)
- **State modifications** in an interrupt handler must be well-considered
  - Protect critical sections
  - Hard to debug (not reliably reproducible!)