Operating System Concepts and Implementations

Betriebssystemkonzepte und -implementierungen

IV. Interrupts (Software)

November 3, 2026

Benedict Herzog

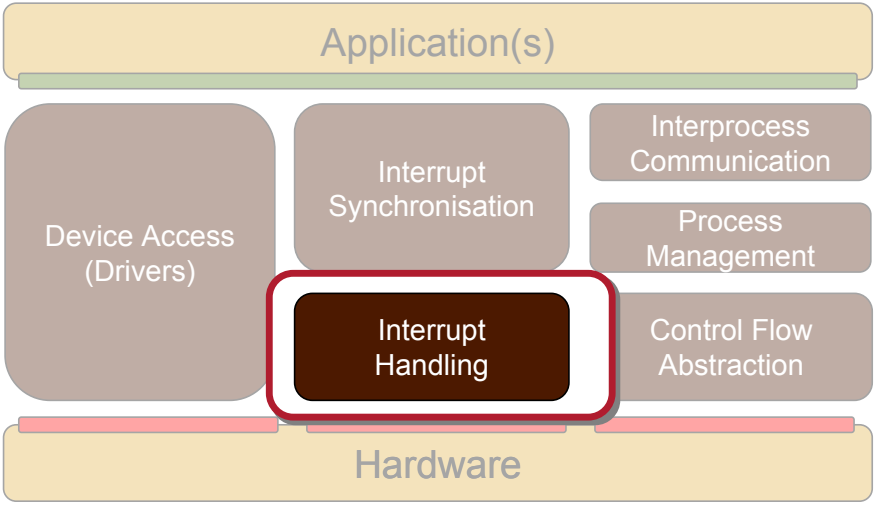
Material based on *Chair of Computer Science 4 (System Software)*
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)



RUHR
UNIVERSITÄT
BOCHUM

RUB

Overview: Today's Focus



Agenda



Context

Terminology and Basic Assumptions

State Preservation

State Modifications

Summary

Agenda



Context

Terminology and Basic Assumptions

- Interrupt, Exception, and Trap
- Basic Assumptions

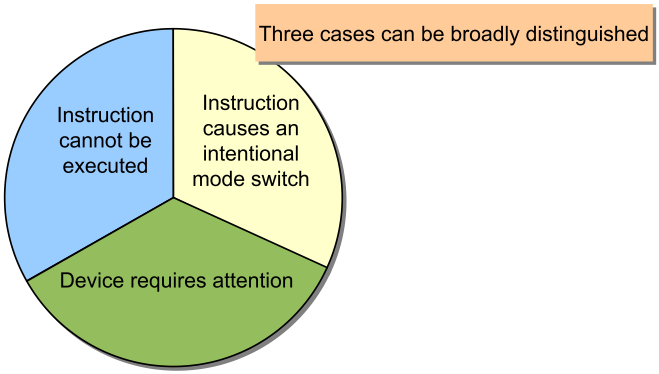
State Preservation

State Modifications

Summary

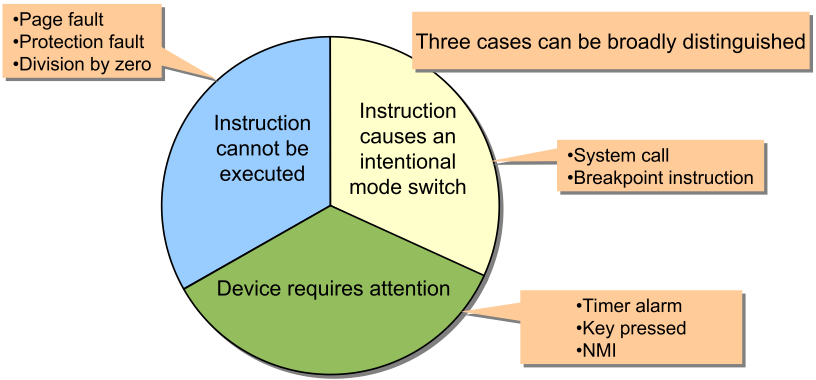
Terminology

- The terminology is interpreted differently …
  - For disambiguation, we take a technical perspective



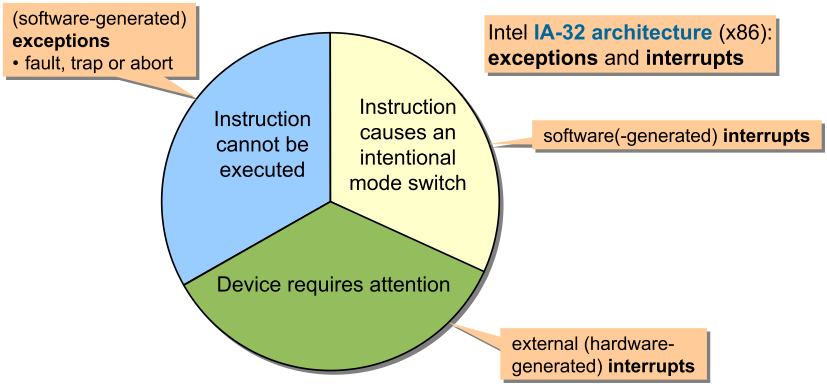
Terminology

- The terminology is interpreted differently …
  - For disambiguation, we take a technical perspective



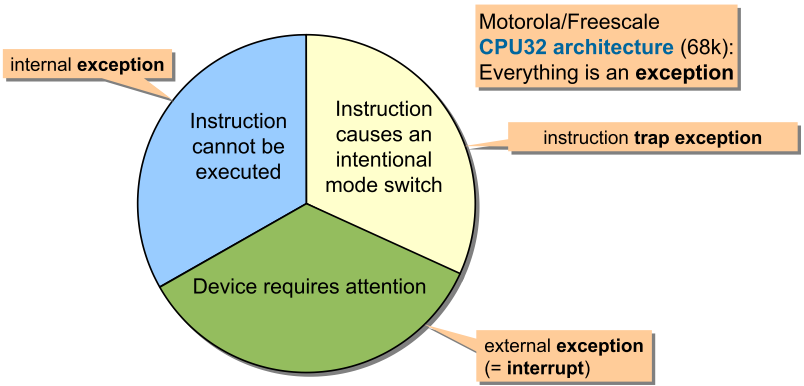
Terminology: Intel IA-32

- The terminology is interpreted differently …
  - For disambiguation, we take a technical perspective



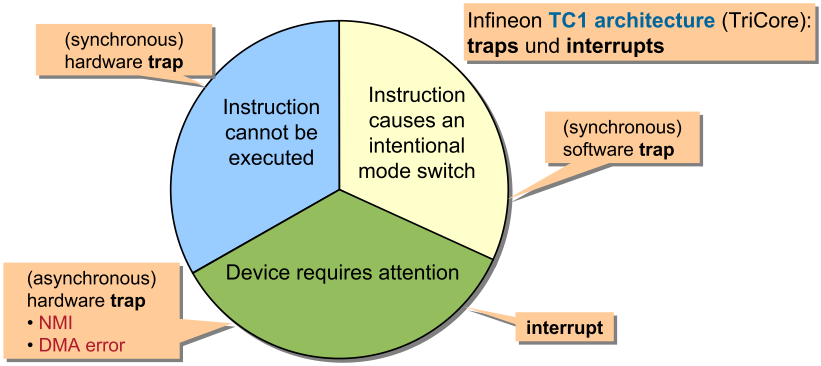
Terminology: Motorola/Freescale CPU32

- The terminology is interpreted differently …
  - For disambiguation, we take a technical perspective



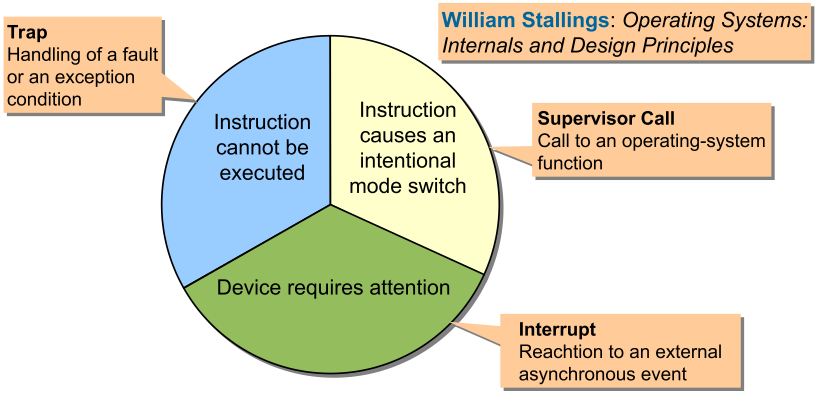
Terminology: Infineon TC1

- The terminology is interpreted differently …
  - For disambiguation, we take a technical perspective



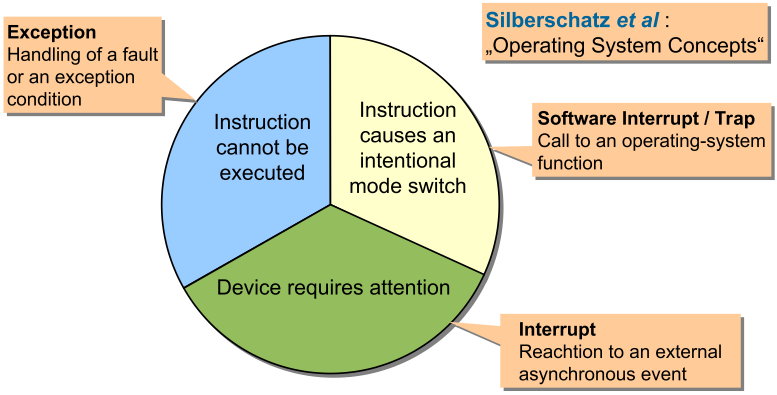
Terminology: Literature (Stallings)

- The terminology is interpreted differently …
  - For disambiguation, we take a technical perspective



Terminology: Literature (Silberschatz)

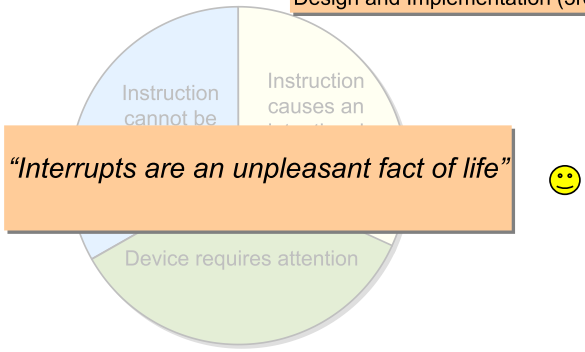
- The terminology is interpreted differently …
  - For disambiguation, we take a technical perspective



Terminology: Literature (Tanenbaum)

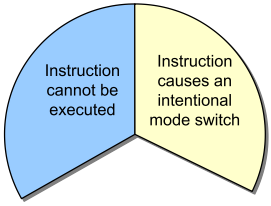
- The terminology is interpreted differently …
  - For disambiguation, we take a technical perspective

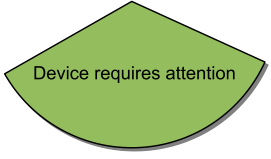
Andrew S. Tanenbaum: Operating Systems Design and Implementation (3rd ed., 2006)



Terminology: Understanding in OSCI

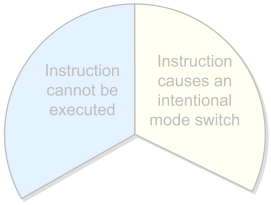
- **Trap**
  - Triggered by an **instruction**
    - … including the `trap` or `int` instruction for system calls
    - Undefined result (e.g. division by zero)
    - Hardware problem (e.g. bus error)
    - Operating system intervention required (e.g. page fault)
    - Invalid instruction (e.g. programming error)
  - Properties
    - Often predictable, often reproducible
    - **Restart or abort** the triggering activity
- **Interrupt**
  - Triggered by **hardware**
    - Hardware requires the attention of the software (timer, keyboard controller, disk controller, …)
  - Properties
    - Unpredictable, not reproducible
    - Usually **resume** the interrupted activity



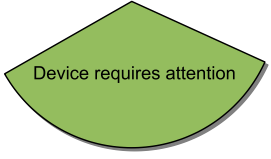


Terminology: Understanding in OSCI

- **Trap**
  - Triggered by an **instruction**
    - … including the `trap` or `int` instruction for system calls
    - Undefined result (e.g. division by zero)
    - Hardware problem (e.g. bus error)
    - Operating system intervention required (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 the attention of the software (timer, keyboard controller, disk controller, …)
  - Properties
    - Unpredictable, not reproducible
    - Usually **resume** the interrupted activity



Basic Assumptions

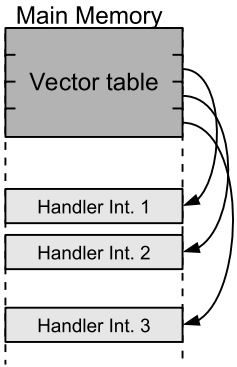


We consider interrupt handling under the following basic assumptions:

1. The CPU starts the interrupt handler **automatically**.
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: Interrupt Handler

1. **The CPU starts the interrupt handler automatically.**
   - Requires the assignment of an interrupt handler
   - Determine the cause of the interrupt (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
- Interrupt handler has own process context

Assumption: Supervisor Mode

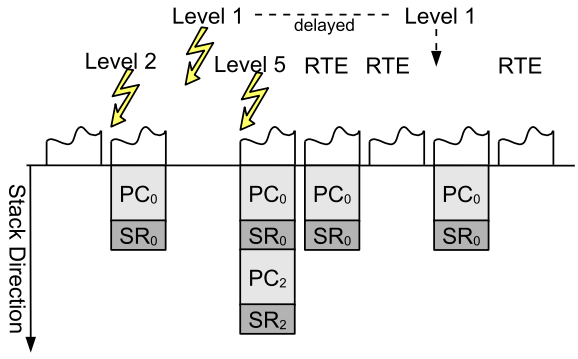
2. **Interrupt handling takes place in supervisor mode.**
   - Interrupts are the only mechanism to preempt non-cooperative applications
   - Only the OS is allowed unrestricted access to devices

     → The CPU switches to the privileged supervisor mode before interrupt handling

Variants:
- On **16-bit CPUs**, a distinction between user and supervisor mode is the exception rather than the rule
- On **8-bit CPUs** (or smaller), this distinction does not exist

Assumption: State Preservation

3. **The interrupted program can be resumed.**
   - Required state is saved automatically
   - Possibly nested; requires a stack



Variants:
- Additional information about cause/trigger is stored on the stack
- No priorities
- Dedicated *interrupt stack*
- Context saved in registers

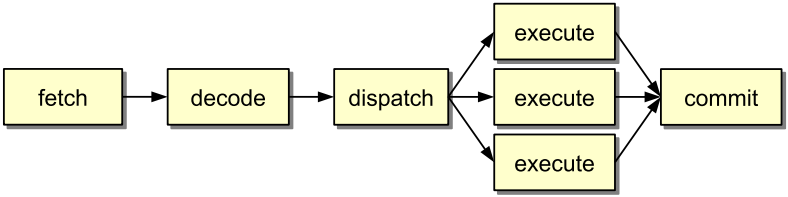
Assumption: Atomic Behaviour

4. **Machine instructions are atomic.**
   - Defined CPU state at the start of the interrupt handler
   - The state must be restorable

   - Trivial for CPUs with a classical von Neumann cycle (fetch-decode-execute)
   - Non-trivial for modern CPUs:
     - Pipelining: instructions must be flushed (thrown away)
     - Superscalar CPUs: must leave pipeline in well-defined state

Assumption: Atomic Behaviour

Instruction Processing in Superscalar Processors (Highly Simplified!):



Ideally, all stages are used at all times, i.e. multiple instructions are executed in parallel. When should the processor check whether an interrupt request is pending?

Assumption: Atomic Behaviour

Although nontrivial, most CPUs provide **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 inPipelined Processors",
IEEE Transactions on Computers, Vol. 37, No. 5, 1988

Assumption: Interrupt Suppression

5. **Interrupt handling can be disabled/suppressed.**
   - Examples:
     - Motorola 68x0: according to priority
     - Intel x86: globally using `sti`, `cli`
     - Interrupt controller: each source individually
   - Additionally: **Automatic suppression** by the CPU before starting the interrupt handler
     - Interrupts are unpredictable and can theoretically occur arbitrarily often
     - Without this automatism, a **stack overflow** would be possible

Assumption: Interrupt Suppression

- The hardware suppresses the handling of ...
  - All interrupts indiscriminately (very restrictive)
  - Interrupts of lower or equal priority (less restrictive)
    - → certain devices are given priority
- Further alternatives based on software (e.g. in Linux):
  - Suppress interrupts that are **currently being handled**
    - → low reaction latency without priority for individual devices

Agenda



Context
Terminology and Basic Assumptions
State Preservation
State Modifications
Summary

State Preservation

- The state of a computer is enormous
  - All processor registers
    - Instruction pointer, stack pointer, general-purpose registers, status register, ...
  - The entire contents of main memory and caches
  - The contents of I/O registers or *ports*, hard disk contents, ...
- Any state the interrupt program does not expected to change asynchronously ...
  - Must not be modified during interrupt handling

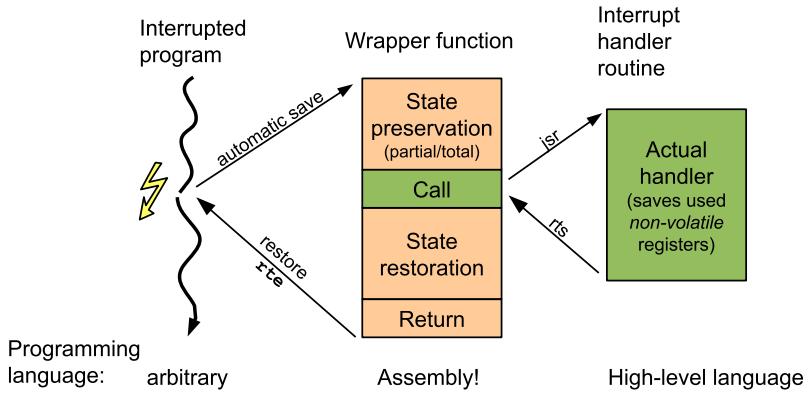
    or
  - Must be saved and subsequently restored
- Depending on the CPU type, the CPU automatically saves ...
  - A minimal number of bytes (e.g. only instruction pointer and status register)
  - All registers

State Preservation Concepts

- **Total state preservation**
  - The interrupt handler saves all registers that were not saved automatically
  - Disadvantage: more state than necessary may be saved
  - Advantage: the saved state is easily accessible (one coherent data structure)
- **Partial state preservation**
  - The interrupt handler saves only those registers that will subsequently be modified or that are not otherwise saved and restored
  - Feasible if the actual interrupt handling is implemented in a high-level language such as C or C++
  - Advantages:
    - Only state that is modified is saved
    - Potentially fewer instructions are required for saving and restoring state
  - Disadvantage: the saved state is “scattered” (not in one place; hard to access)

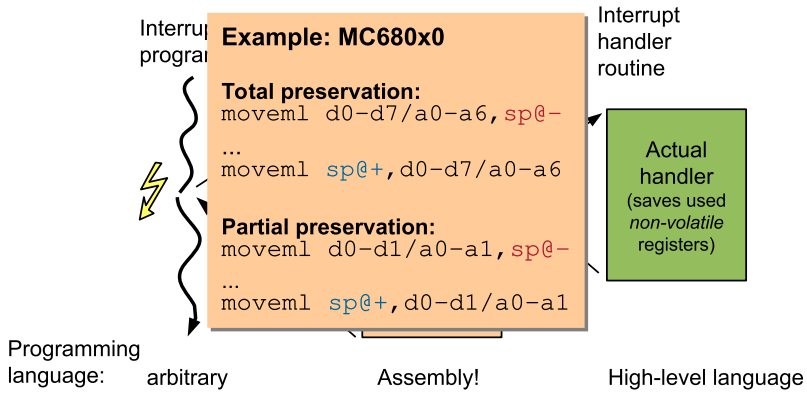
Transition to High-Level Language

- Non-portable machine code should be minimised
- The actual interrupt handling is performed in a high-level language function



Transition to High-Level Language

- Non-portable machine code should be minimised
- The actual interrupt handling is performed in a high-level language function



Excursus: Volatile and Non-Volatile Registers

- A distinction made by the **(C/C++) compiler**
- **non-volatile** (aka *callee-saved* registers)
  - Compiler guarantees that values of these registers are preserved across function calls
  - If necessary, they are saved and restored by the **called function**
- **volatile** (aka *caller-saved* or *scratch* registers)
  - If the **calling function** still requires a value after the function call, it must save and restore the register itself
  - Usually used for intermediate results
- Usually defined in a standard all compilers adhere to (why?)
  - Example for 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, ...

State Restoration

- As its last duty, the wrapper must restore the contents of all saved registers
  - … and must not modify them afterwards!
- A special instruction, e.g. `rte` (68k) or `iret` (x86-64), restores the previous state
  - Reads automatically saved state from supervisor stack
  - Restores the saved operating mode (user/supervisor mode) and jumps to saved address

The OS can also modify the state before **`rte`**/**`iret`**. This is useful to execute OS code in user mode.

Agenda



Context

Terminology and Basic Assumptions

State Preservation

State Modifications

- Examples
- Problem Analysis
- Solutions

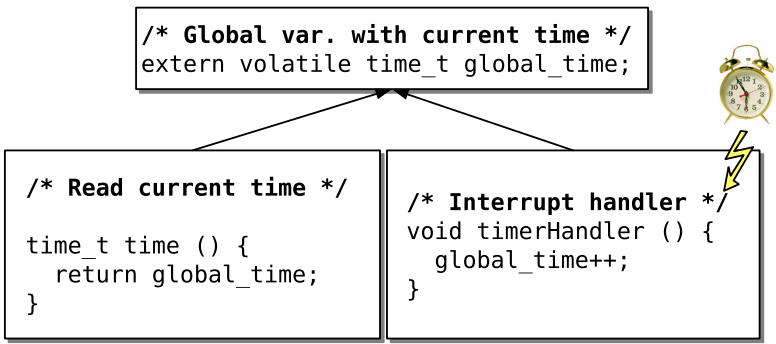
Summary

State Modifications …

- … are the main purpose of interrupt handling
  - Device drivers must be informed when an I/O operation has completed
  - The scheduler must be notified when a time slice has expired
- … must be performed with care
  - Interrupts can occur at any time
  - **Critical sections**: Data and data structures shared between the normal control flow and interrupt handling

Example 1: System Time

- The global system time is incremented by a timer interrupt
  - e.g. once per second
- The system time can be queried using the operating system function `time()`



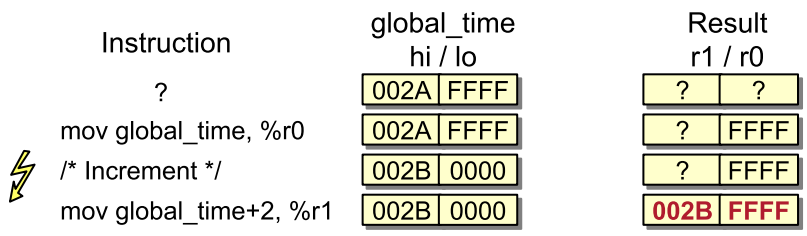
Example 1: System Time

- There may be a bug lurking here ...
  - Reading `global_time` is not necessarily atomic!

```
; 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 load instructions on the 16-bit CPU



Example 1: System Time

- There may be a bug lurking here …
  - Re

Problem

Every 18.2 hours (0x10000 s), the system time may (briefly) jump ahead by approximately the same amount (0x*FFFF* s). Unfortunately, the problem cannot be reliably reproduced.

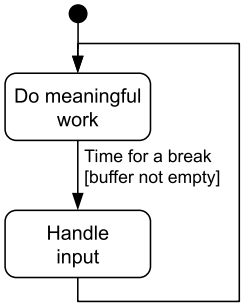
- **Critical**: Interrupt between the two load instructions on the 16-bit CPU

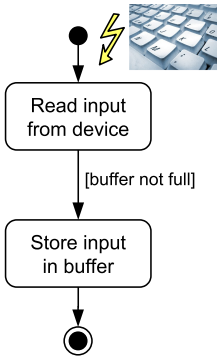


Instruction	global_time hi	global_time lo	Result r1	Result r0
?	002A	FFFF	?	?
mov global_time, %r0	002A	FFFF	?	FFFF
/* Increment */	002B	0000	?	FFFF
mov global_time+2, %r1	002B	0000	002B	FFFF

Example 2: Ring Buffer

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

Problem #1

If input is not handled/consumed quickly enough, the **buffer may become full**. The interrupt handler can then no longer store the input in the buffer. In this case, this **input is lost**.



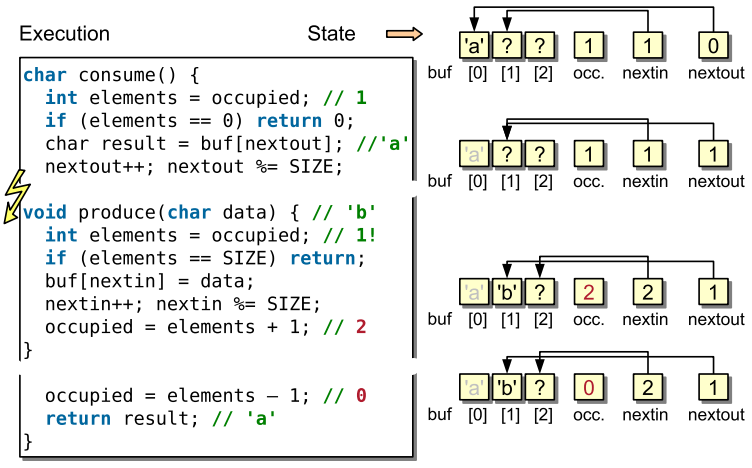
Example 2: Ring Buffer

The buffer implementation itself is also 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 element counter
    if (elements == SIZE) return; // Full? Drop element
    buf[nextin] = data;            // Write element
    nextin++; nextin %= SIZE;      // Advance write index
    occupied = elements + 1;       // Increment element counter
  }
  char consume() {                 // Regular control flow:
    int elements = occupied;       // Local copy 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;       // Decrement element counter
    return result;                 // Return result
} };
```

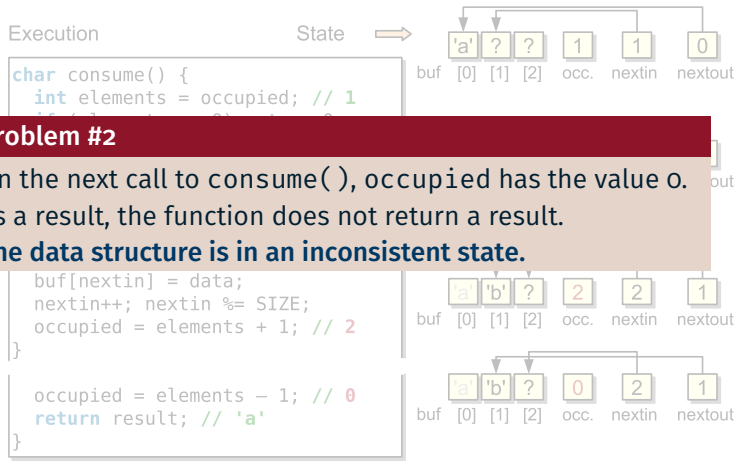
Example 2: Ring Buffer

The buffer implementation itself is also critical …



Example 2: Ring Buffer

The buffer implementation itself is also critical ...



Problem #2

On the next call to `consume()`, `occupied` has the value 0.

As a result, the function does not return a result.

The data structure is in an inconsistent state.

State Modification: Analysis

- Even individual assignments are not necessarily atomic
  - Depends on CPU type, compiler, and code optimisations
- Buffer space is finite
  - Interrupt handlers cannot wait
  - Data might get lost
- The buffer data structure may become corrupted due to …
  - Inconsistent intermediate states during modifications by regular control flow
  - State modifications while reading data (inconsistent copy!)
  - Modifications based on a copy that no longer corresponds to the original
- The problem is not symmetric
  - The regular control flow does not *interrupt* interrupt handling
  - We can exploit this fact!

“Hard” Synchronisation

- Race conditions can be avoided by suppressing interrupts
  → operations on shared data are made atomic this way

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

- Problems:
  - Risk of losing interrupt requests
  - High and difficult-to-predict **interrupt latencies**

Further Techniques in the Next Lecture



- *Clever* (optimistic) approaches
  - Clever data structure choice
    - Minimise shared elements
    - Use relaxed consistency requirements
  - Optimistic approach
    - Usually, no interrupt occurs within the critical section
    - If one does occur, the resulting inconsistency is detected and repaired
    - If necessary, the operation is repeated/restarted
- Prologue/Epilogue Model
  - Divide interrupt handling into two phases
    - The critical part is deferred by a software mechanism
    - A low-latency response remains possible

Agenda



Context
Terminology and Basic Assumptions
State Preservation
State Modifications
Summary

Summary



- Correct interrupt handling is one of the most difficult tasks in OS design
  - source of asynchronity
    - Both a blessing and a curse
    - Introduces non-determinism into the system
  - State preservation at register level
    - Assembly programming!
    - Dependency on the compiler (e.g. volatile/non-volatile registers)
  - Different models (priorities, etc.)
- State modifications during interrupt handling must be carefully considered
  - Protect critical sections
  - Errors are difficult to find (not reliably reproducible!)