

OPERATING-SYSTEM CONSTRUCTION

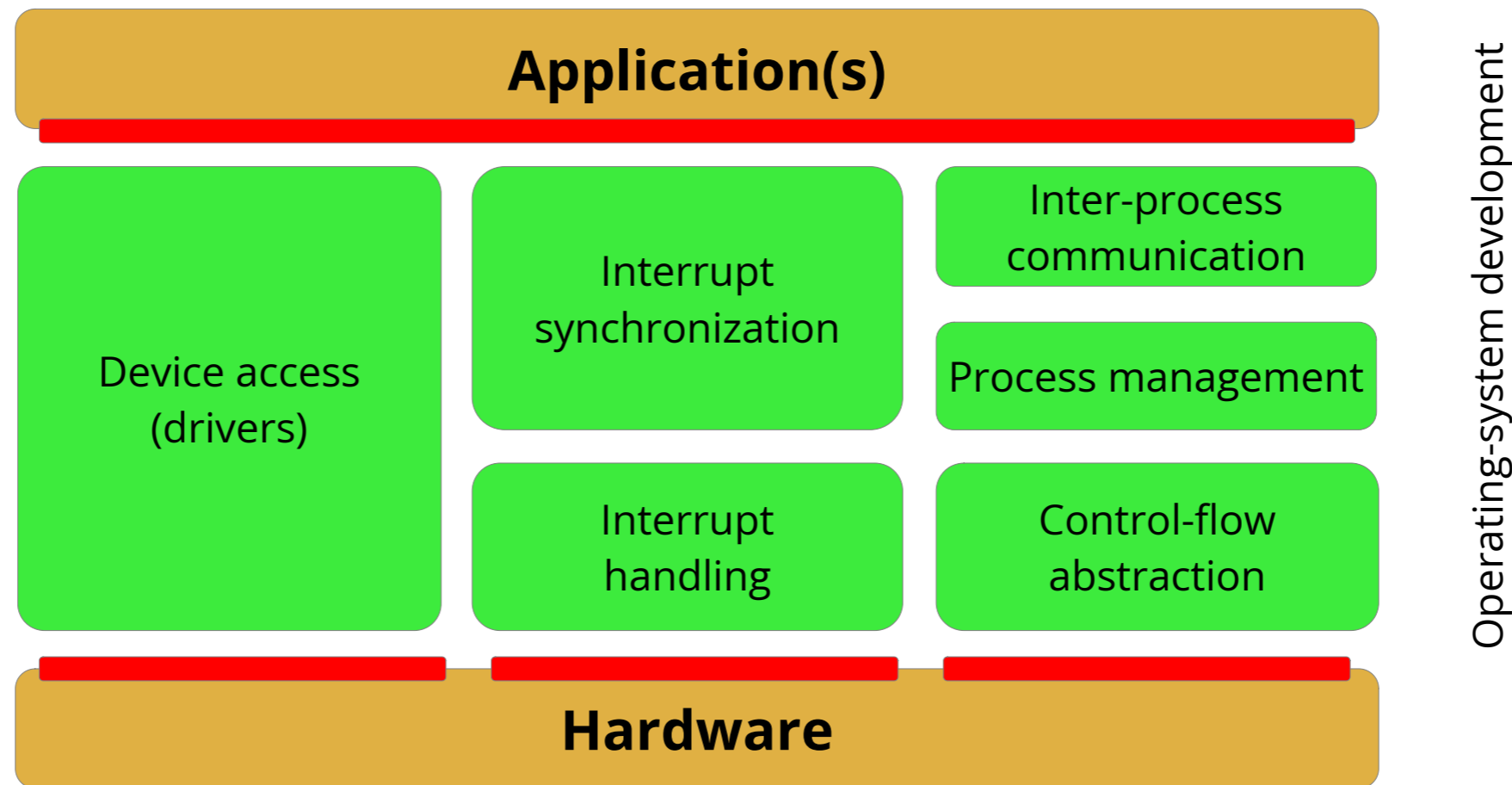
Interrupts – Synchronization

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

TILL SMEJKAL (slides by HORST SCHIRMEIER)

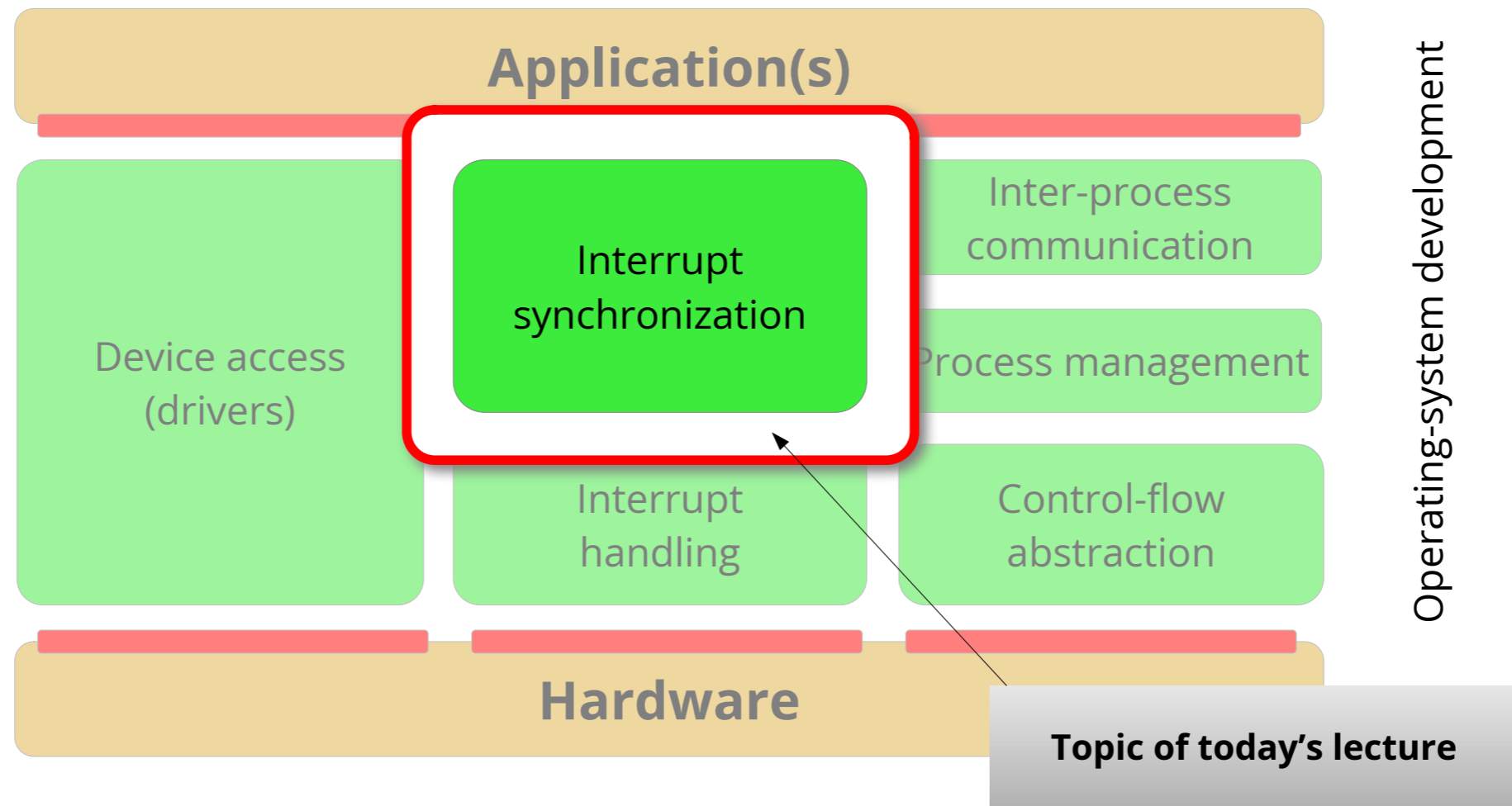
Overview: Lectures

Structure of the "OO-StuBS" operating system:



Overview: Lectures

Structure of the "OO-StuBS" operating system:



Agenda

- Recapitulation
- Control-Flow Level Model
- Hard Synchronization
- Nonblocking Synchronization
- Synchronization with the Prologue/Epilogue Model
- Summary

Agenda

- **Recapitulation**
- Control-Flow Level Model
- Hard Synchronization
- Nonblocking Synchronization
- Synchronization with the Prologue/Epilogue Model
- Summary

Motivation: Consistency Issues

Example 1: System Time

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

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

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

Examples from the previous lecture

Example 2: Ring Buffer

Execution

```
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'
}
```

State

buf [0] [1] [2] occ. nextin nextout

buf [0] [1] [2] occ. nextin nextout

buf [0] [1] [2] occ. nextin nextout

buf [0] [1] [2] occ. nextin nextout

First Approach

- One-sided synchronization
 - Suppress interrupts on the consumer side
 - Operations `disable_interrupts()` / `enable_interrupts()`
(in the following without loss of generality in "Intel" speak: `cli()` / `sti()`)

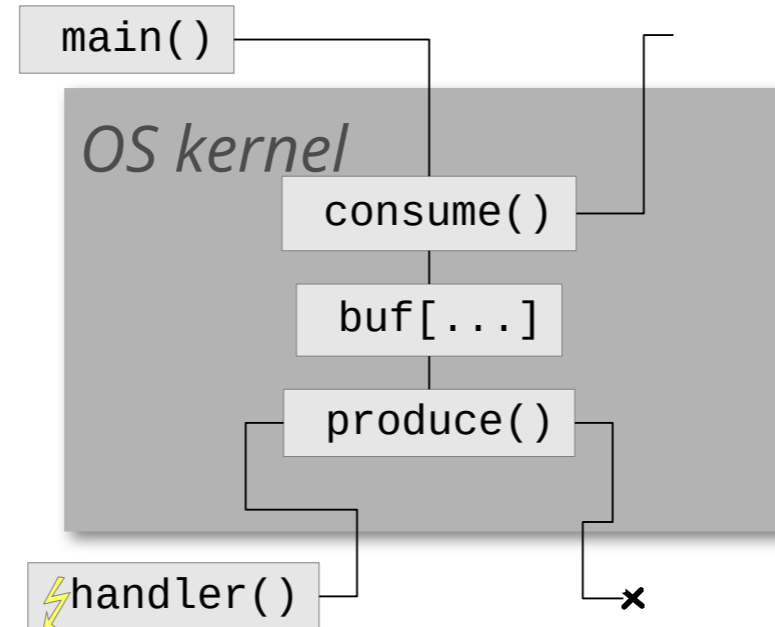
It works with **one-sided** synchronization ...

(why?)

```
char consume() {
    cli();
    ...
    char result = buf[nextout++];
    ...
    sti();
    return result;
}
```

```
void produce(char data) {
    // nothing to do here
    ...
    buf[nextin++] = data;
    ...
    // nothing to do here
}
```

Application control flow (A)



Interrupt handler (IH)

First Conclusions

- Ensuring consistency between an application control flow (A) and an interrupt handler (IH) works differently than between processes.
- Relationship between A and IH is **asymmetric**
 - “Different kinds” of control flows
 - IH **interrupts** A
 - implicitly, at an arbitrary point
 - always higher priority, runs to completion
 - A **can suppress** (better: **delay**) IH
 - explicitly, with `cli / sti` (assumption #5 from previous lecture)
- Synchronization / maintenance of consistency is **one-sided**

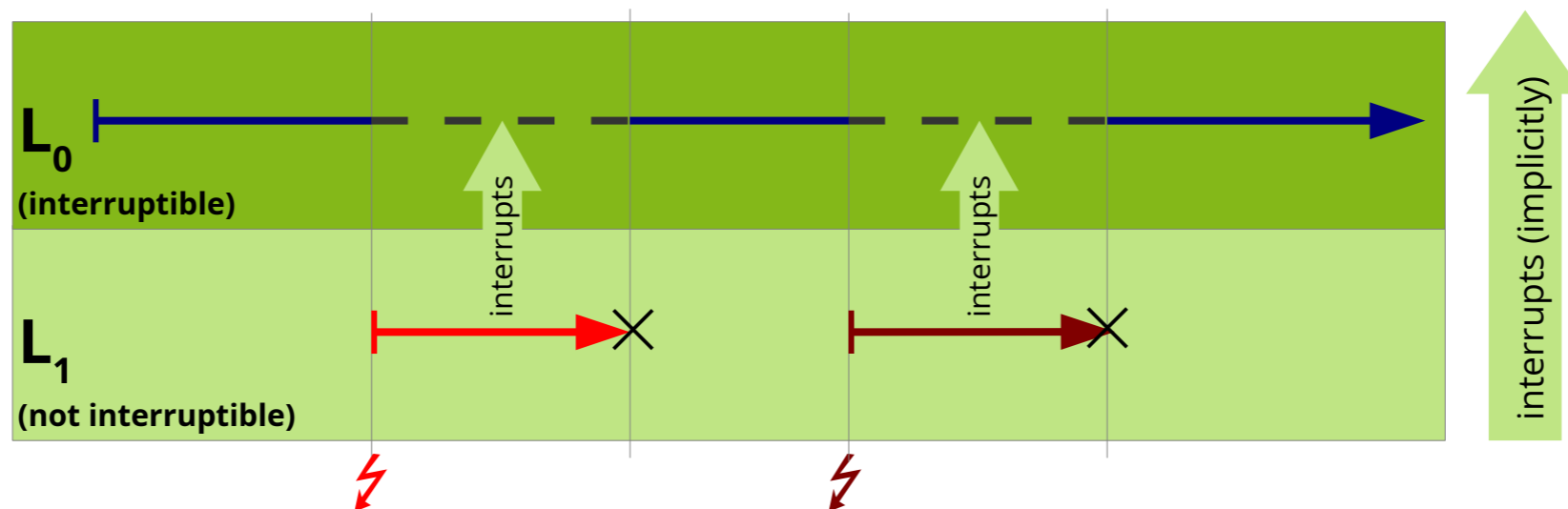
We must take these facts **into account!**
(This also means: We can **exploit** them.)

Agenda

- Recapitulation
- **Control-Flow Level Model**
- Hard Synchronization
- Nonblocking Synchronization
- Synchronization with the Prologue/Epilogue Model
- Summary

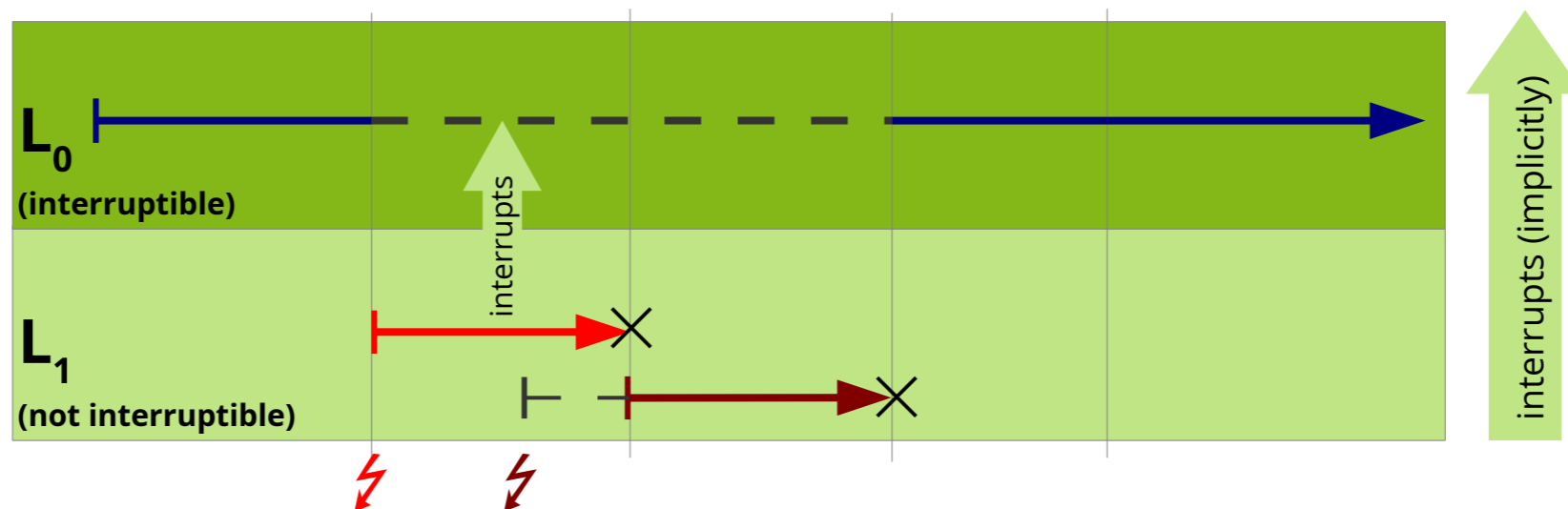
Control-Flow Level Model

- Be L_0 the application control-flow level (A)
 - Control flows on this level are **interruptible at any time** (by L_1 control flows, implicitly)
- Be L_1 the interrupt handling level (IH)
 - Control flows on this level are **not interruptible** (by other $L_{0/1}$ control flows)
 - L_1 control flows have priority over L_0 control flows



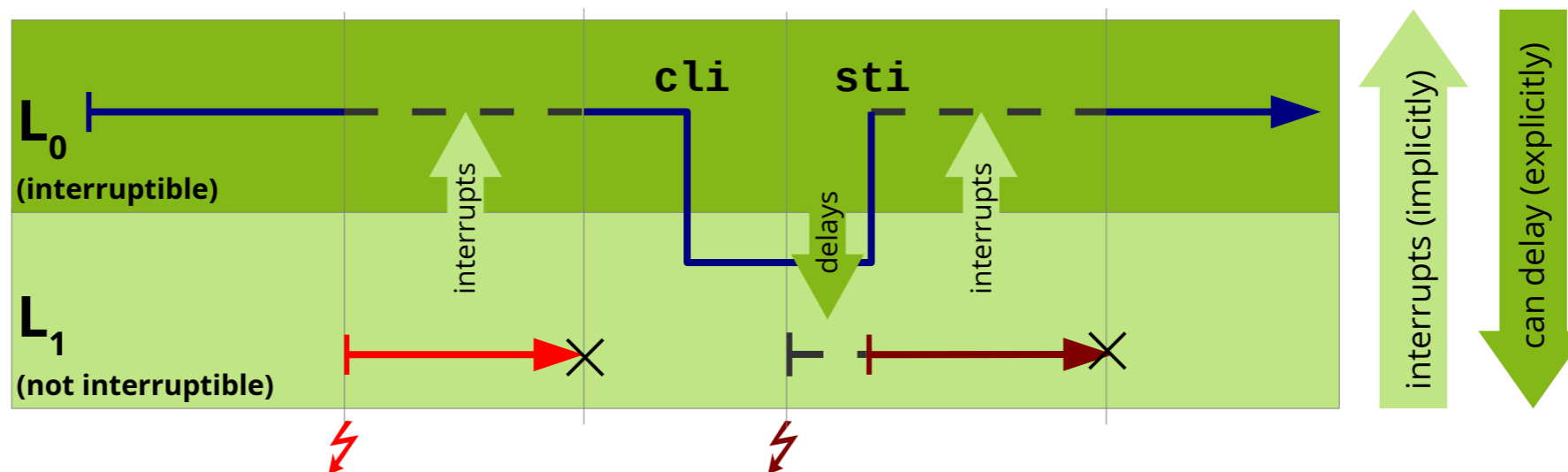
Control-Flow Level Model

- Control flows of the same level are **sequentialized**
 - If multiple control flows on one level are ready, they are **executed sequentially** (*run-to-completion*).
 - Consequence: max. 1 control flow active on each level
 - Arbitrary **sequentialization strategy**
 - FIFO, LIFO, with priorities, random, ...
 - For L_1 control flows on the PC, the PIC implements this strategy.



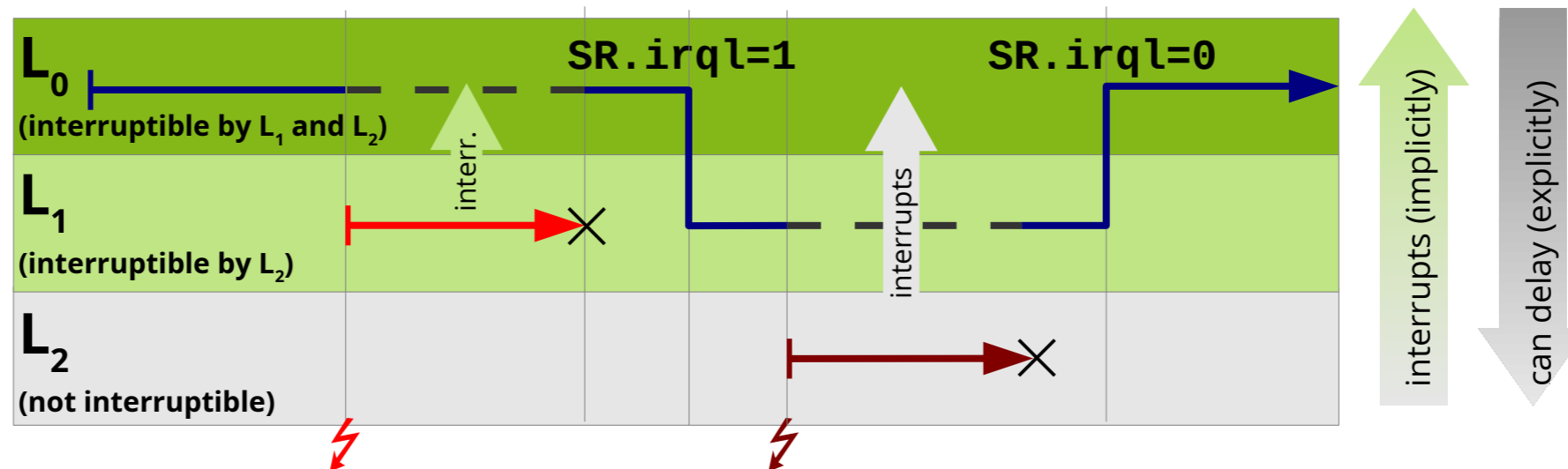
Control-Flow Level Model

- Control flows can switch levels
 - With `cli` an L_0 control flow explicitly **switches** to L_1
 - from then on no longer interruptible
 - other L_1 control flows are delayed (\rightarrow sequentialization)
 - With `sti` an L_1 control flow explicitly **switches** to L_0
 - from then on interruptible (again)
 - delayed/pending L_1 control flows get their turn now (\rightarrow sequentialization)



Control-Flow Level Model

- Generalization to multiple interrupt levels:
 - Control flows on L_f are
 - **interrupted anytime** by control flows on L_g (for $f < g$)
 - **never interrupted** by control flows on L_e (for $e \leq f$)
 - **sequentialized** with other control flows on L_f
 - Control flows can switch levels
 - by special operations (here: modifying the status register)



Control-flow Levels: Maintaining Consistency

- Each state variable is (logically) assigned to **exactly one level** L_f
 - Accesses from L_f implicitly consistent (\rightarrow sequentialization)
 - For accesses from higher/lower levels, we must **explicitly maintain consistency**.
- Measures for maintaining consistency:
 - “from above” (from L_e with $e < f$) with **hard synchronization**
 - **explicitly switch to L_f** for the access (delay)
 - Thereby, the access comes from the same level. (\rightarrow sequentialization)
 - “from below” (from L_g with $f < g$) with **nonblocking synchronization**
 - make sure algorithmically that interrupts do not endanger consistency
 - necessitates **interrupt-transparent** algorithms

Agenda

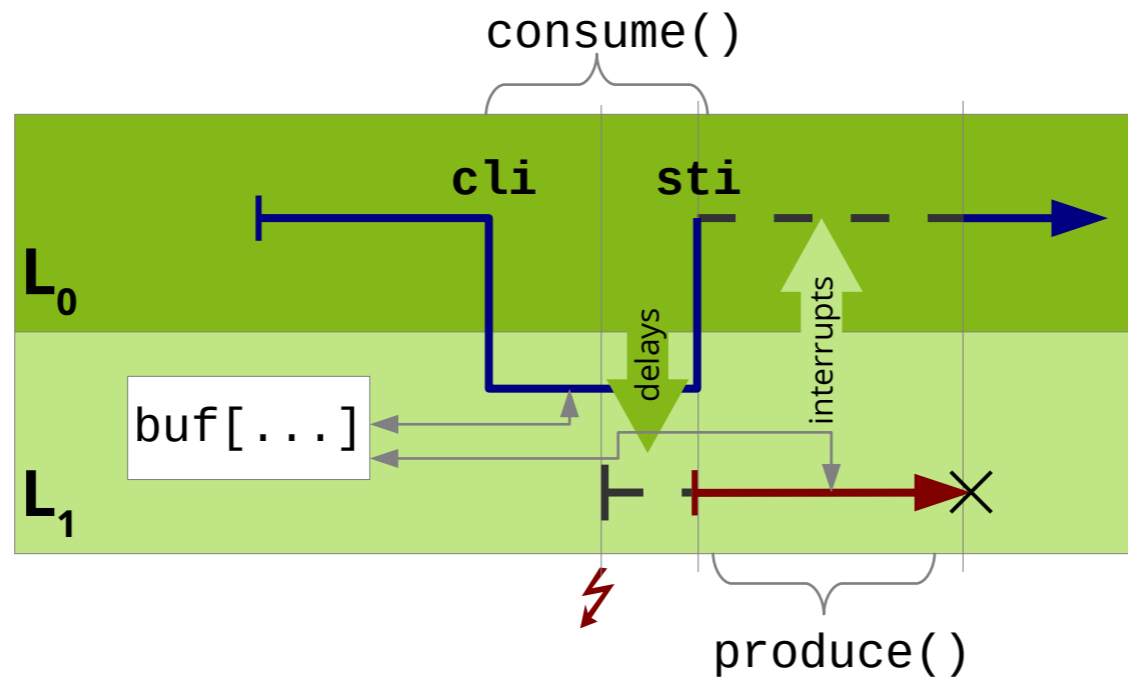
- Recapitulation
- Control-Flow Level Model
- **Hard Synchronization**
- Nonblocking Synchronization
- Synchronization with the Prologue/Epilogue Model
- Summary

Bounded Buffer – Hard Synchronization

Access “from a higher layer” is synchronized hard.
 (For the execution of consume(), the control flow switches to L₁.)

```
char consume() {
    cli();
    ...
    char result = buf[nextout++];
    ...
    sti();
    return result;
}
```

```
void produce(char data) {
    // nothing to do here
    ...
    buf[nextin++] = data;
    ...
    // nothing to do here
}
```



State (logically) resides on L₁.

Hard Synchronization: Assessment

- **Advantages:**
 - Maintains consistency
 - also for complex data structures and access patterns
 - We're (largely) independent from what our compiler does.
 - Simple to use (for the developer), always works
 - In doubt, put all state in the highest-priority level.
- **Disadvantages:**
 - **Broadband effect**
 - Across-the-board all interrupt handlers (control flows) on and below the state level are delayed.
 - **Priority violation**
 - We delay control flows with a higher priority.
 - **Pessimism**
 - We put up with disadvantages although the probability of a relevant interrupt is tiny.

Hard Synchronization: Assessment

- Whether disadvantages become significant depends on the delays'
 - frequency,
 - mean duration, and
 - maximum duration.
- **Maximum duration** is the most critical one:
 - directly influences the (to be expected) **latency**
 - Latency too high → data can get lost
 - Interrupts aren't noticed
 - Data is picked up too slowly from I/O devices

Conclusion: Hard synchronization is rather **unsuitable** for maintaining consistency of **complex data structures**.

Agenda

- Recapitulation
- Control-Flow Level Model
- Hard Synchronization
- **Nonblocking Synchronization**
- Synchronization with the Prologue/Epilogue Model
- Summary

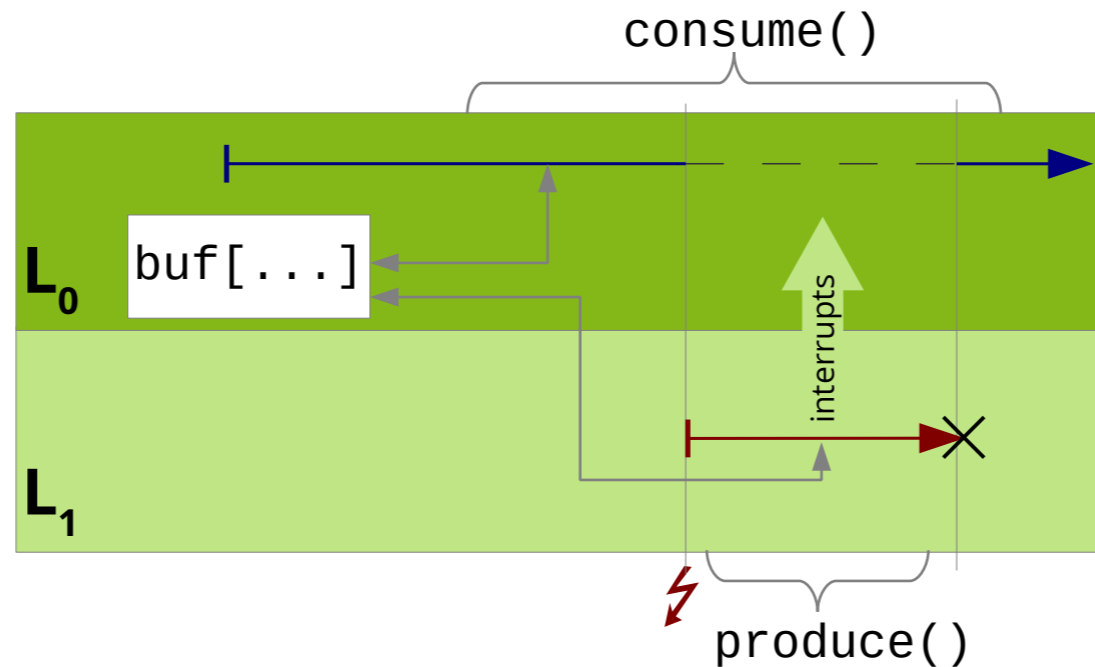
Bounded Buffer – Nonblocking Sync.

Access “from a lower layer” is synchronized in a nonblocking manner. (consume() yields a correct result even if during its execution produce() was executed.)

```
void produce(char data) {
    ?
}
```

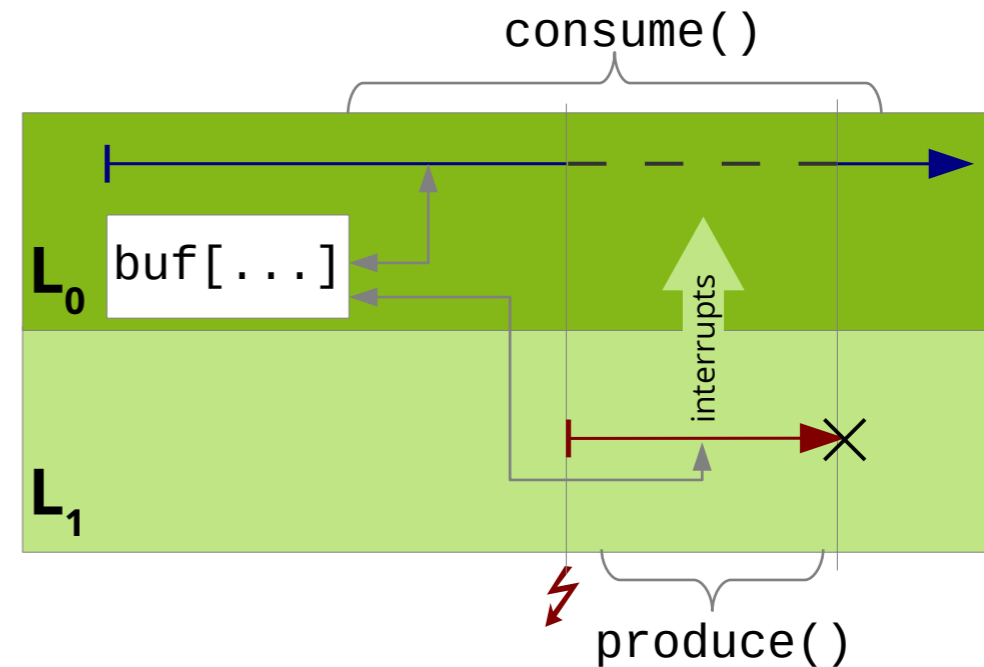
```
char consume() {
    ?
}
```

State (logically) resides on L₀.



Bounded Buffer – Nonblocking Sync.

- Consistency condition: **Sequential consistency**
 - The result of an interrupted execution must be equivalent to an arbitrary sequential execution of the operations
 - *either* consume() before produce()
or consume() after produce()
- Assumptions:
 - produce() interrupts consume()
 - all other combinations do not occur
 - produce() always runs to completion



Bounded Buffer – Code from previous lecture

- Shared state is critical

Especially state that
both sides write.

```
// 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
    }
};
```

Bounded Buffer – New Code

```
// Bounded ring buffer in C++ (alternative)
class BoundedBuffer {
    char buf[SIZE]; int nextin, nextout;
public:
    BoundedBuffer(): nextin(0), nextout(0) {}
    void produce(char data) {

        if ((nextin + 1) % SIZE == nextout) return;
        buf[nextin] = data;
        nextin = (nextin + 1) % SIZE;

    }
    char consume() {

        if (nextout == nextin) return 0;
        char result = buf[nextout];
        nextout = (nextout + 1) % SIZE;

        return result;
    } };
```

This implementation alternative goes **without shared state written by both sides.**

Bounded Buffer – New Code

```
// Bounded ring buffer in C++ (alternative)
class BoundedBuffer {
    char buf[SIZE]; int nextin, nextout;
public:
    BoundedBuffer(): nextin(0), nextout(0) {}
    void produce(char data) {

        if ((nextin + 1) % SIZE == nextout) return;
        buf[nextin] = data;
        nextin = (nextin + 1) % SIZE;

    }
    char consume() {

        if (nextout == nextin) return 0;
        char result = buf[nextout];
        nextout = (nextout + 1) % SIZE;

        return result;
    } };
```

However, now we have state that is **read** by one side and **written** by the other.

This is where we must check whether our **consistency condition holds**.

Bounded Buffer – Code Analysis

- Assuming the interrupt in consume() occurs ...
 - as seen from **consume()**
 - before reading **nextin** \Leftrightarrow produce() before consume() ✓
 - after reading **nextin** \Leftrightarrow consume() before produce() ✓
 - as seen from **produce()**
 - before writing **nextout** \Leftrightarrow produce() before consume() ✓
 - after writing **nextout** \Leftrightarrow consume() before produce() ✓

In all four cases, the consistency condition holds.

```
char consume() {  
    if (nextout == nextin) return 0;  
    char result = buf[nextout];  
    nextout = (nextout + 1) % SIZE;  
    return result;  
}
```

```
void produce(char data) {  
    if ((nextin + 1) % SIZE == nextout) return;  
    buf[nextin] = data;  
    nextin = (nextin + 1) % SIZE;  
}
```

System Time - Code from last lecture

```
/* 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++;  
}
```

g++ (16-bit architecture)

```
time:  
    mov global_time, %r0; lo  
    mov global_time+2, %r1; hi  
    ret
```

Problem: Variable is
not read atomically

System Time – Nonblocking Sync.

- Consistency condition:
 - The result of an interrupted execution must be equivalent to an arbitrary sequential execution of the operations
 - *either* `time()` before `timerHandler()` *or vice versa*
- Assumptions:
 - `timerHandler()` interrupts `time()`
 - all other combinations do not occur
 - `timerHandler()` always runs to completion
- Approach: Implement `time()` **optimistically**
 1. Read data assuming we are not interrupted
 2. Check whether assumption was correct (were we interrupted?)
 3. If interrupted, restart at step 1

System Time - New Implementation

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



```
/* Read current time */  
time_t time () {  
    time_t res;  
    do {  
        interrupted = false;  
        res = global_time;  
    } while (interrupted);  
    return res;  
}
```

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

Consistency condition
now holds in any case.

Nonblocking Sync.: Assessment

- **Advantages:**

- Maintains consistency (by **interrupt transparency**)
- No priority violations (interrupts stay enabled!)
- No cost, or only in the (rare) conflict situation
 - no cost → bounded-buffer example
 - in the conflict situation → optimistic approaches, system-time example (additional cost by restarting)

- **Disadvantages:**

- **Complexity**
 - If we find an algorithm at all, it's usually hard to understand and even harder to verify.
- **Constraints**
 - Tiny code changes can ruin the consistency guarantee.
 - Compiler's code generation must be taken into account.
- **Predictability**
 - Costs for restart unpredictable for large amounts of data.

Nonblocking Sync.: Assessment

- **Advantages:**

- Maintains consistency (by **interrupt transparency**)
- No priority violation
- No cost, or only
 - no cost → both
 - in the conflict (additional cost)

Conclusion:

Nonblocking synchronization is neat. However, the involved algorithms are **special solutions for special cases.**

- **Disadvantages:**

- **Complexity**
 - If we find an
- **Constraints**
 - Tiny code changes can ruin the consistency guarantee.
 - Compiler's code generation must be taken into account.
- **Predictability**
 - Costs for restart unpredictable for large amounts of data.

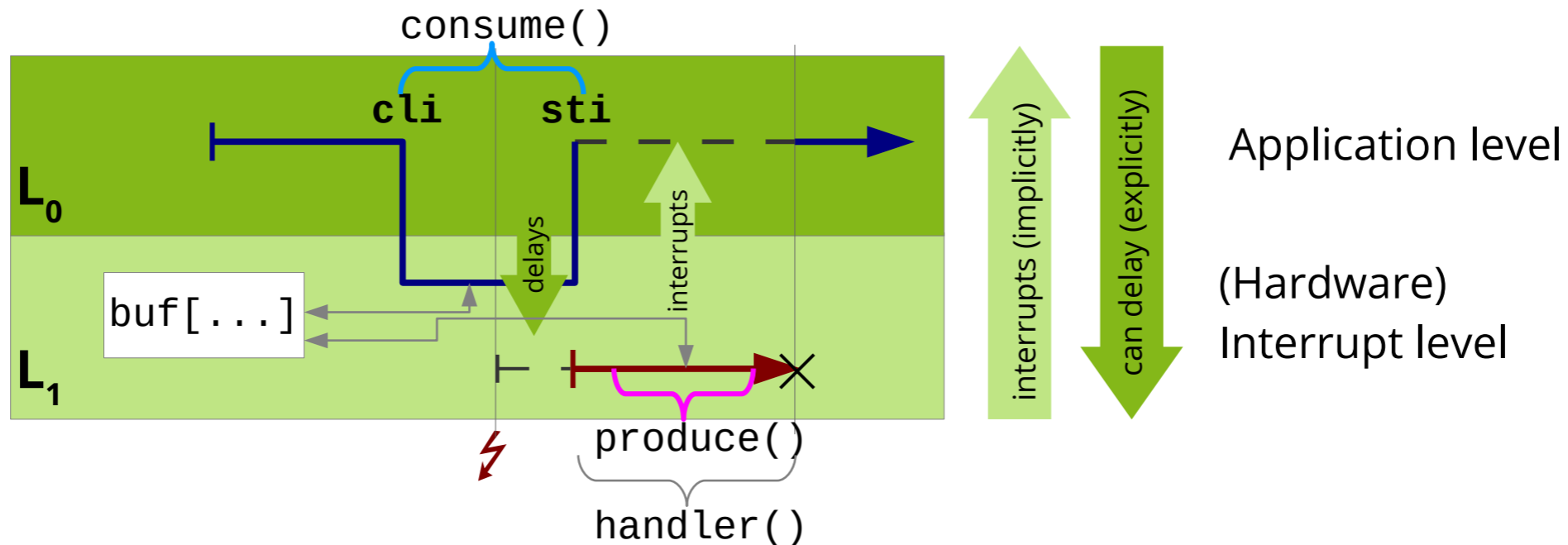
It is **not suitable** as a generally applicable measure for maintaining consistency of **complex data structures.**

Agenda

- Recapitulation
- Control-Flow Level Model
- Hard Synchronization
- Nonblocking Synchronization
- **Synchronization with the Prologue/Epilogue Model**
- Summary

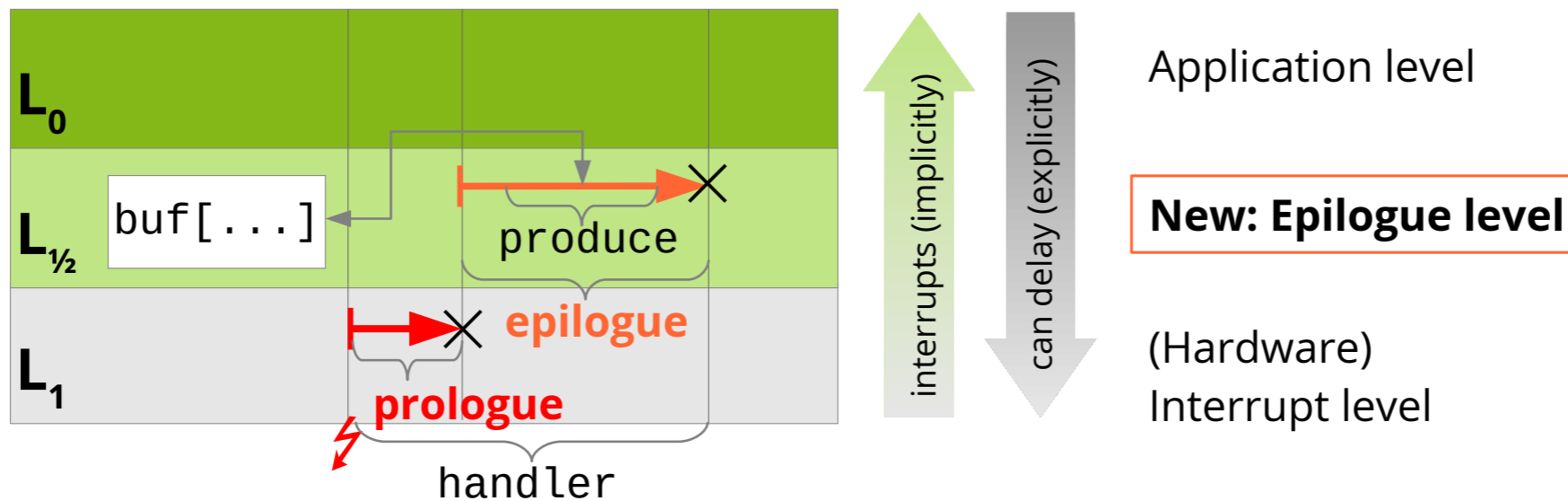
Prologue/Epilogue Model – Motivation

- Again: Hard synchronization
 - Simple, correct, “always works” ✓
 - Main problem: Latency ✗
 - long delay when **accessing state** from higher levels
 - long delay when **modifying state** in the IH itself
 - ... in the end caused by the fact that the state (logically) resides on the/a hardware interrupt level ($L_{1\dots n}$)



Prologue/Epilogue Model – Approach

- **Idea:** We insert **another level $L_{1/2}$** between application level L_0 and the interrupt-handling levels $L_{1\dots n}$
 - IH is divided into **prologue** and **epilogue**
 - **Prologue** runs on interrupt level $L_{1\dots n}$
 - **Epilogue** runs on (software) level $L_{1/2}$ (**epilogue level**)
 - State resides (as far as possible) on epilogue level
 - actual interrupt handling is only disabled briefly



Prologue/Epilogue Model – Approach

- **Interrupt-handler routines are divided into two halves**
 - start in their prologue (always)
 - are continued in their epilogue (on demand)
- **Prologue**
 - runs on hardware-interrupt level
 - Prioritized over application level and epilogue level
 - is short, touches little or no state
 - **Hardware interrupts are only disabled briefly**
 - can request an epilogue on demand
- **Epilogue**
 - runs on epilogue level (additional control-flow level)
 - Execution delayed in respect to prologue
 - does the actual work
 - has access to most of the state
 - State is synchronized on the epilogue level

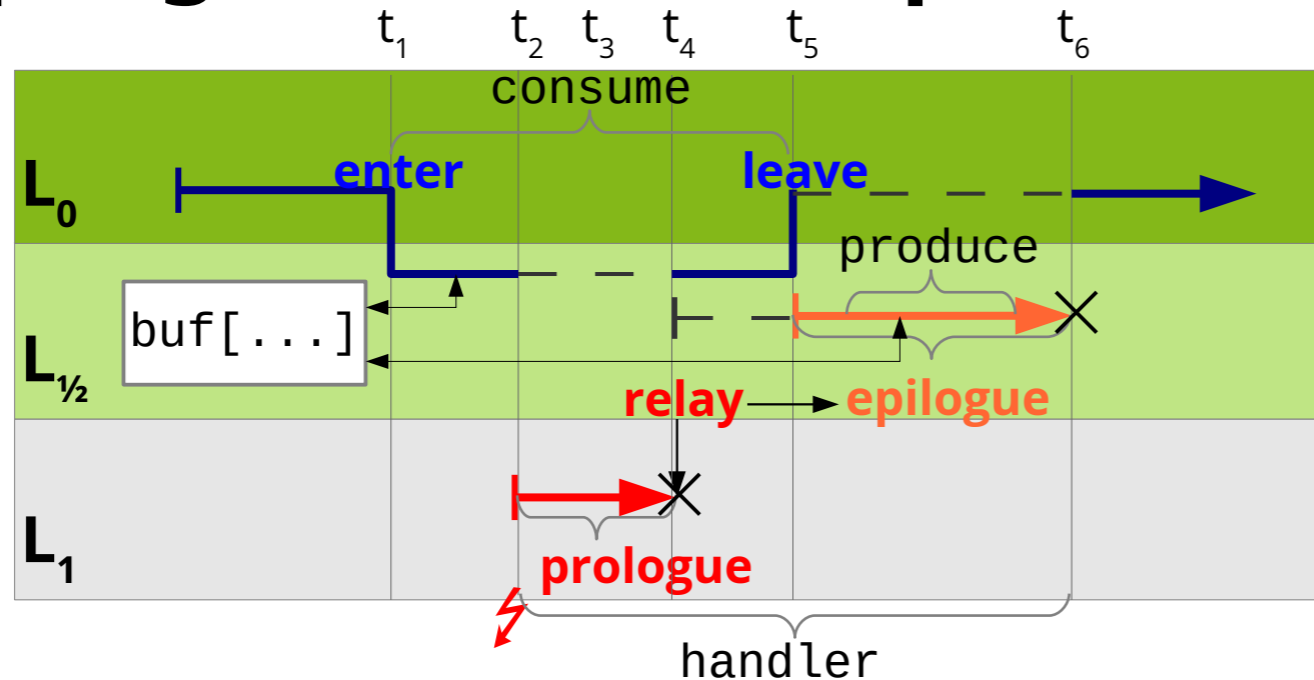
Pro/Epilogue Model – Epilogue Level

- Epilogue level is implemented (completely, or partially) in **software**
 - nevertheless a regular control-flow level within the level model
 - the same rules apply
- As before: Control flows on epilogue level $L_{\frac{1}{2}}$ are
 1. **interrupted anytime** by control flows on levels $L_{1\dots n}$
 - Prologues (interrupts) have priority over epilogues
 2. **never interrupted** by control flows of L_0
 - Epilogues have priority over application control flows
 3. **sequentialized** with other control flows on $L_{\frac{1}{2}}$
 - Pending epilogues are executed sequentially.
 - When returning to application level, all epilogues have been completed.

Pro/Epilogue Model – Implementation

- We need operations to
 1. explicitly enter the epilogue level: **enter()**
 - corresponds to `cli` in hard synchronization
 2. explicitly leave the epilogue level: **leave()**
 - corresponds to `sti` in hard synchronization
 3. request an epilogue: **relay()**
 - corresponds to pulling an IRQ line to “high” at the PIC

Pro/Epilogue Model – Sequence Example



L_1 interrupts are never disabled.

Interrupt-handler activation **latency is minimal.**

- 1 Application control flow enters epilogue level $L_{1/2}$ (**enter**).
- 2 Interrupt is signaled on level L_1 , execute prologue.
- 3 Prologue requests epilogue for delayed execution (**relay**).
- 4 Prologue terminates, interrupted $L_{1/2}$ control flow (application) continues.
- 5 Application control flow leaves epilogue level $L_{1/2}$ (**leave**), process meanwhile accumulated epilogues.
- 6 Epilogue terminates, application control flow continues on L_0 .

Pro/Epilogue Model – Implementation

- We need operations to
 1. explicitly enter the epilogue level: **enter()**
 - corresponds to cli in hard synchronization
 2. explicitly leave the epilogue level: **leave()**
 - corresponds to sti in hard synchronization
 3. request an epilogue: **relay()**
 - corresponds to pulling an IRQ line to “high” at the PIC
- Additionally, mechanisms to
 4. remember pending epilogues, e.g. a **queue**
 - corresponds to PIC's IRR (*interrupt request register*)
 5. ensure that pending epilogues are processed
 - corresponds to the protocol between CPU and PIC in hard sync.

We'll have to have a closer look at this part.

Pro/Epilogue Model – Implementation

- When do we have to process pending epilogues?

Just before the CPU returns to L_0 !

1. when explicitly leaving the epilogue level with **leave()**
 - While the application control flow ran on epilogue level, more epilogues may have accumulated.
(→ sequentialization)
 2. after processing the last epilogue
 - While processing epilogues, more epilogues may have accumulated.
 3. after the **last** interrupt handler terminates
 - While the CPU executed control flows on levels $L_{1...n}$, epilogues may have accumulated.
(→ prioritization)
- Two implementation variants:
 - with hardware support via an AST (now, in the lecture)
 - completely software-based (in the exercises)

Pro/Epilogue Model – Implementation

- An AST (*asynchronous system trap*) is an interrupt that can (only) be requested by software
 - e.g. by setting a bit in a specific register
 - otherwise technically comparable to a hardware interrupt
- Main difference to **traps** / exceptions / software interrupts:
AST is executed **asynchronously**
 - runs on own interrupt level between app. and hardware IHs (our $L_{\frac{1}{2}}$)
 - Level model's rules apply (AST execution is delayable, automatically activated, ...)
- AST simplifies ensuring epilogues are processed
 - Processing in AST (automatically, before returning to L_0)
 - We just need to manage pending epilogues

Pro/Epilogue Model – Implementation

- Example TriCore: Implementation with AST
 - AST implemented as an L_1 interrupt here (\Leftrightarrow our $E_{1/2}$)
 - Hardware interrupts on $L_{2\dots n}$

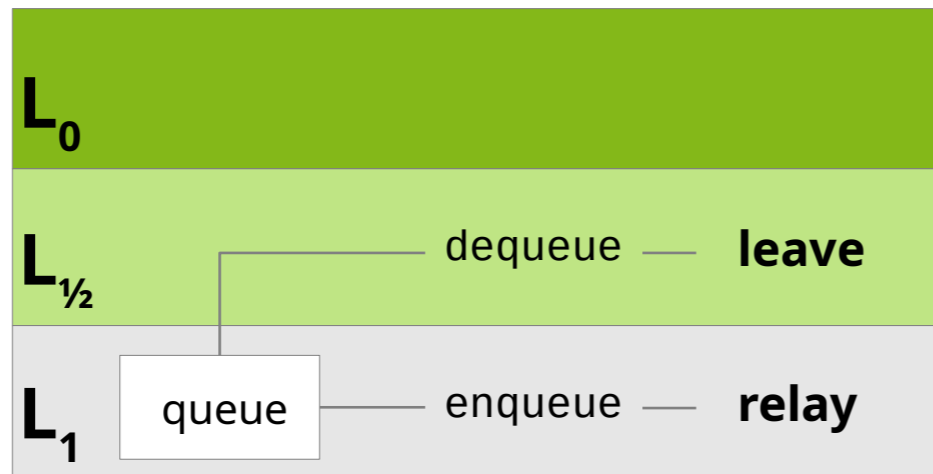
If the hardware does not offer an AST concept (e.g. x86-64 doesn't), we can imitate it in software.

We'll do that in the exercises.

```
void enter() {
    CPU::setIRQL(1);           // enter  $L_1$ , delay AST
}
void leave() {
    CPU::setIRQL(0);          // allow AST (pending
                             // AST would now be processed)
}
void relay(<Epilogue>) {
    <enqueue epilogue in queue>
    CPU_SRC1::trigger();      // activate level-1 IRQ (AST)
}
void __attribute__((interrupt_handler)) irq1Handler() {
    while (<Epilogue in queue>) {
        <dequeue epilogue from queue>
        <process epilogue>
    }
}
```

Pro/Epilogue Model: Goal Achieved?

- Kernel state can now be maintained and synchronized on epilogue level
 - No need to disable hardware interrupts anymore
- One issue remains: The epilogue queue
 - Access from prologues and epilogue level
 - either hard synchronization (shown here)
 - or special solution with nonblocking synchronization



Hard synchronization seems **acceptable** here, since the time frame with interrupts disabled (runtime of dequeue()) is **short** and **deterministic**.

A solution with nonblocking synchronization would be nice anyways!

Pro/Epilogue Model – Related Concepts

- Windows: *ISRs / deferred procedure calls* (DPCs)
 - *Interrupt Service Routines* (ISRs) can enqueue **DPCs** in a waiting queue. This queue is processed delayed before the CPU returns to the thread level.
- Linux: *ISRs / bottom halves* (BHs)
 - Linux has a bit in a bit mask for each interrupt service routine (ISR) through which it can request a delayed **bottom half**. These BHs are executed before leaving the kernel.
 - Beyond this, Linux uses a concept comparable to DPCs: waiting queues of **tasklets**.
- eCos: *ISRs / deferred service routines* (DSRs)
- ...

Almost all operating systems with interrupt handling provide an “epilogue level”.

Pro/Epilogue Model – Assessment

- **Advantages:**
 - Maintains **consistency** (synchronization on epilogue level)
 - Programming model **corresponds to** the (easily understandable) model behind **hard synchronization**
 - Also **complex state** can be synchronized
 - without losing IRQs
 - allowing to protect the whole OS kernel on epilogue level
- **Disadvantages:**
 - Additional level leads to additional **overhead**
 - Epilogue activation could take longer than direct handling
 - Higher complexity for the OS developer
 - We don't completely get rid of **disabling interrupts**
 - Shared state between prologue and epilogue must still be synchronized hard or nonblockingly

Pro/Epilogue Model – Assessment

- **Advantages:**

- Maintains **consistency** (synchronization on epilogue level)
- Programming model **corresponds to** the (easily understandable) model behind **hard synchronization**
- Also **complex state** can be accessed
 - without losing IRQs
 - allowing to protect the state

- **Disadvantages:**

- Additional level leads to
 - Epilogue activation overhead
 - Higher complexity for the programmer
- We don't completely get rid of **disabling interrupts**
 - Shared state between prologue and epilogue must still be synchronized hard or nonblockingly

Conclusion:

The prologue/epilogue model is a **good compromise** for synchronizing accesses to kernel state.

It is also suitable for maintaining consistency of **complex data structures**.

Agenda

- Recapitulation
- Control-Flow Level Model
- Hard Synchronization
- Nonblocking Synchronization
- Synchronization with the Prologue/Epilogue Model
- **Summary**

Summary: Interrupt Synchronization

- Maintaining consistency in the OS kernel
 - Must be done differently than between processes → one-sided
 - Control flows run on different levels
- Measures for maintaining state consistency
 - **hard synchronization** (by disabling interrupts)
 - simple, but negative effect on latency
 - interrupts can get lost
 - **nonblocking synchronization** (by interrupt transparency)
 - nice and efficient, but only possible in specific scenarios
 - implementation may become quite complex
 - **Prologue/epilogue based synchronization** (by splitting the interrupt handler into two halves)
 - good compromise, synchronization without affecting latency