TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf Spinczyk, Universität Osnabrück

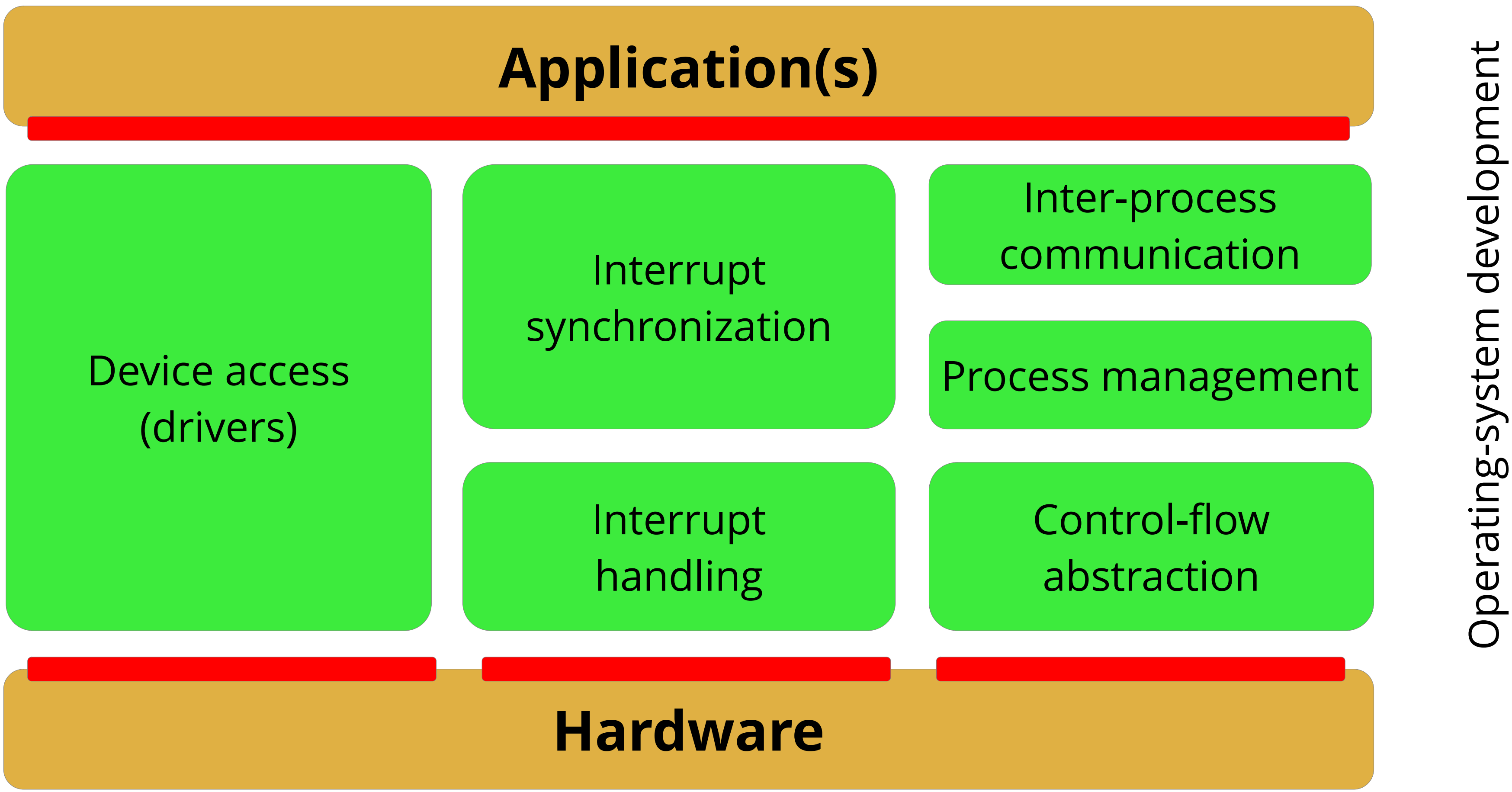
Thread Synchronization

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

HORST SCHIRMEIER

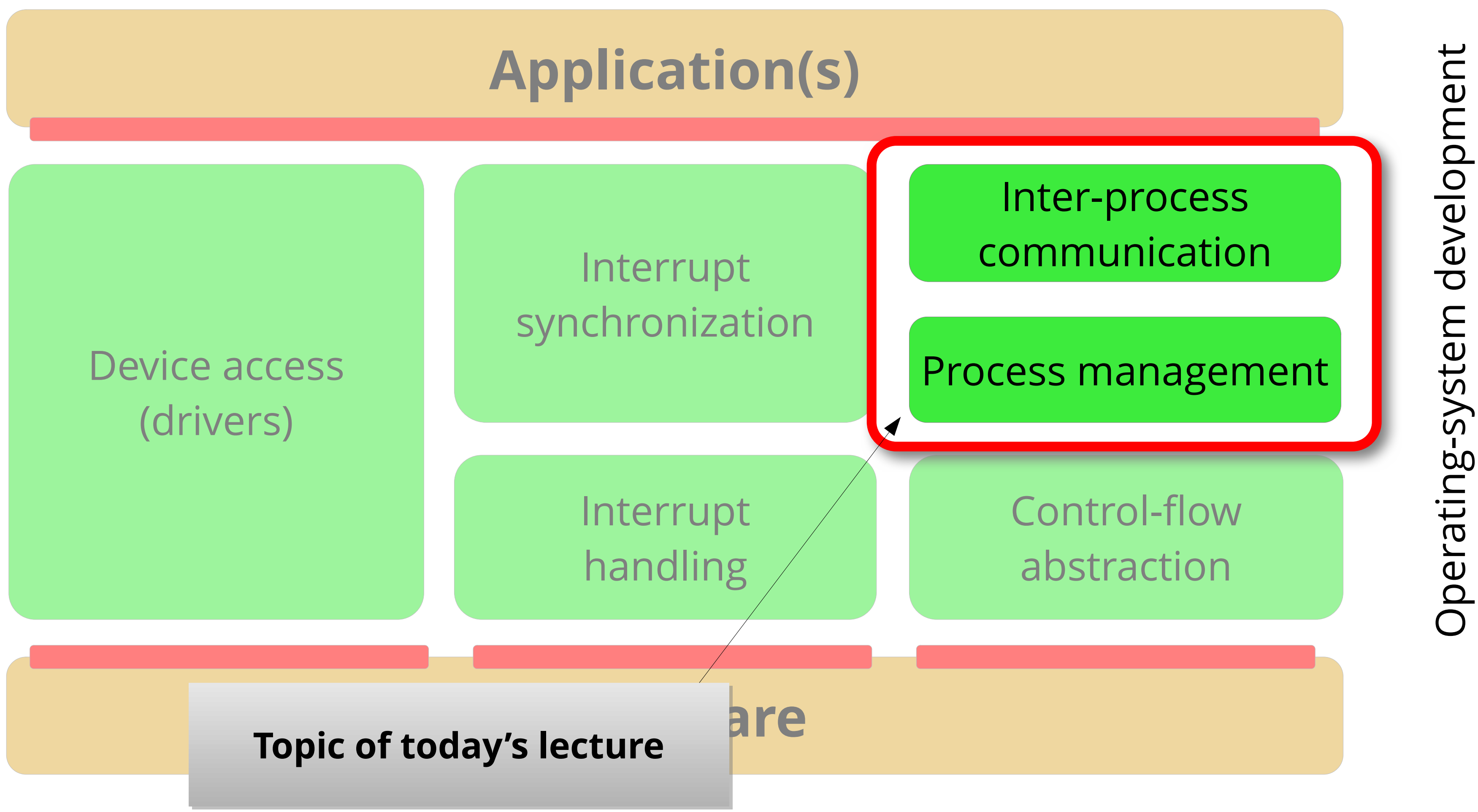
Overview: Lectures

Structure of the “OO-StuBS” operating system:



Overview: Lectures

Structure of the “OO-StuBS” operating system:



Agenda

- Motivation
- Control-flow Level Model with Threads
- Thread Synchronization
  - Constraints
  - Mutex, Implementation Variants
  - Concept of Passive Waiting
  - Semaphore
- Example: Synchronization Objects on Windows
- Summary

Agenda

- **Motivation**
- Control-flow Level Model with Threads
- Thread Synchronization
  - Constraints
  - Mutex, Implementation Variants
  - Concept of Passive Waiting
  - Semaphore
- Example: Synchronization Objects on Windows
- Summary

Motivation: Scenario

- Given: Threads <f> and <g>
  - Preemptive round-robin scheduling
  - Both access a shared buffer **buf**

```
#include "BoundedBuffer.h"

extern BoundedBuffer buf;
```

```
void f() {
   ...
   char el;
   el = buf.consume();
   ...
}
```

```
void g() {
   ...
   char el = ...
   buf.produce( el );
   ...
}
```

Motivation: Consistency Issues

- Given: Threads <f> and <g>
  - **Problem: Buffer accesses can overlap**

```
char BoundedBuffer::consume() {
  int elements = occupied;
  if (elements == 0) return 0;
  char result = buf[nextout];
  nextout++; nextout %= SIZE;
```

```
...
void BoundedBuffer::produce(char data) {
  int elements = occupied;
  if (elements == SIZE) return;
  buf[nextin] = data;
  nextin++; nextin %= SIZE;
  occupied = elements + 1;
}
...
```

```
  occupied = elements – 1;
  return result;
}
```

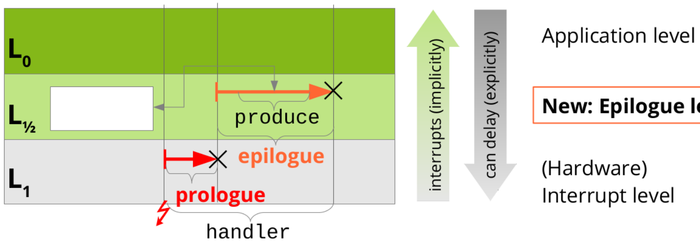
We've seen this before ...

L05: Interrupt Synchronization

What is different this time?

Prologue/Epilogue Model – Approach

- **Idea:** We insert **another level $L_{1/2}$** between application level L_0 and the interrupt-handling levels $L_{1 \ldots n}$
  - IH is divided into ***prologue*** and ***epilogue***
    - **Prologue** runs on interrupt level $L_{1 \ldots n}$
    - **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

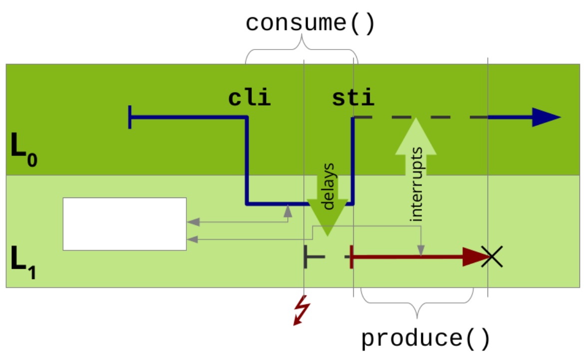


Bounded Buffer – Hard Synchronization

Access "from above" is synchronized hard.
(For the execution of consume(), the control flow switches to L_1.)

```
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
}
```



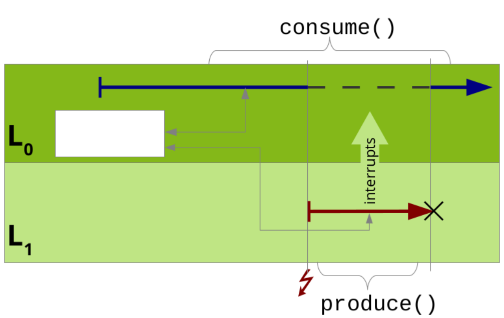
State (logically) resides on L_1.

Bounded Buffer – Nonblocking Sync.

State (logically) resides on L_0

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

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



Access "from below" is synchronized in a nonblocking manner.
(consume() yields a correct result even if during its execution produce() was executed.)

First Conclusion

- Before: Synchronization of accesses by control flows from **different levels**
  - State was logically assigned to one specific level
  - Synchronization either "from above" (hard) or "from below" (non-blocking)
  - Implicit sequentialization within the same level

- Now: Synchronization of accesses by control flows from the **same level**
  - Threads can be preempted by other threads at any time.

That's the point of threads!

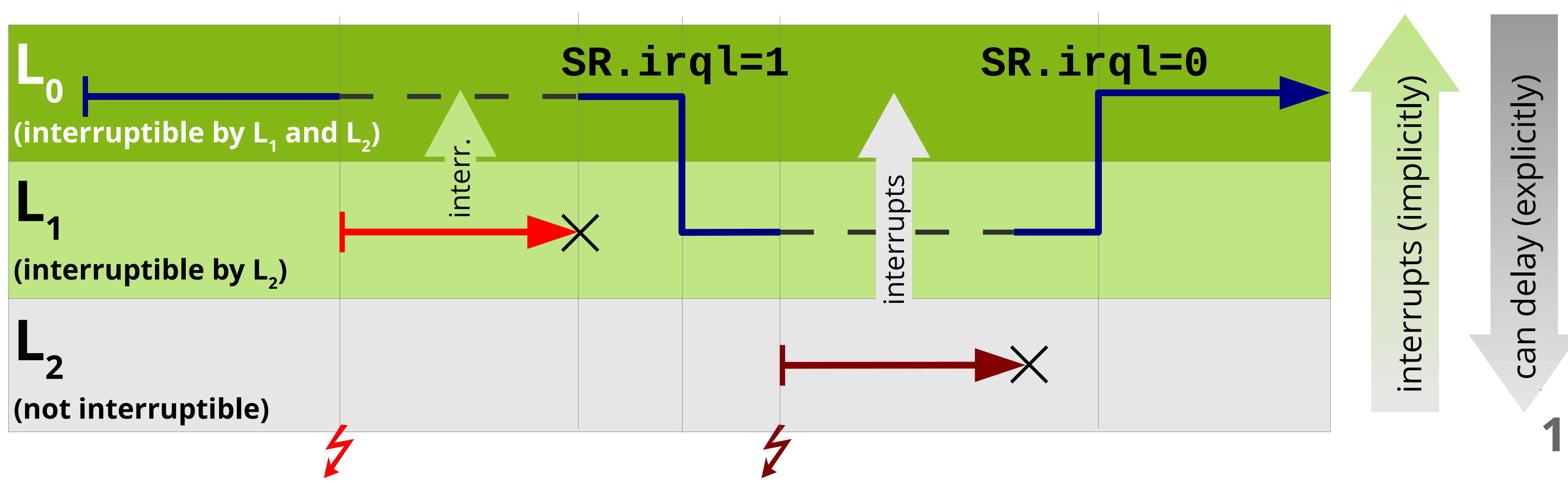
Agenda

- Motivation
- **Control-flow Level Model with Threads**
- Thread Synchronization
  - Constraints
  - Mutex, Implementation Variants
  - Concept of Passive Waiting
  - Semaphore
- Example: Synchronization Objects on Windows
- Summary

TECHNISCHE UNIVERSITÄT DRESDEN

Control-Flow Level Model: so far

- 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)



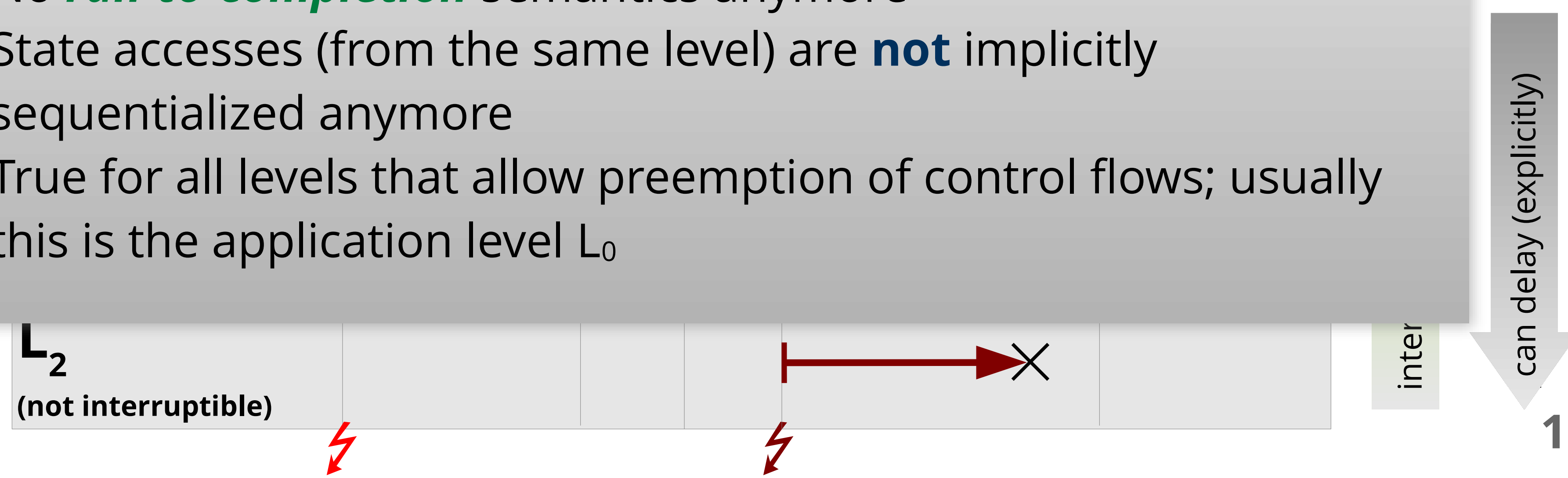
TECHNISCHE UNIVERSITÄT DRESDEN

Control-Flow Level Model: so far

- 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 supporting **preemptive threads** we cannot sustain this **assumption** any longer!

- No ***run-to-completion*** semantics anymore
- State accesses (from the same level) are **not** implicitly sequentialized anymore
- True for all levels that allow preemption of control flows; usually this is the application level L_0



Control-Flow Level Model: new

- 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 (for $f > 0$)
  - **preempted** by other control flows on L_f **(for $f = 0$)**

L_0 → Thread level **(interruptible, preemptible)**
L_1 → Epilogue level **(interruptible, not preemptible)**
L_2 → Interrupt level **(not interruptible, not preemptible)**

Control flows on level L_0 (thread level) are **preemptible**.

To maintain consistency on this level, we need additional mechanisms for **thread synchronization**.

TECHNISCHE UNIVERSITÄT DRESDEN

Thread Synchronization: Assumptions

- Threads can be preempted **unpredictably**
  - at any time (also by external events)
    - interrupts
  - by any other thread
    - of higher, same or lower priority (progress guarantee!)
- Typical assumptions for desktop computers
  - *probabilistic*, *interactive*, *preemptive*, *online* CPU scheduling
  - We do not consider other scheduling variants here.

Primarily, **progress guarantee** is causing the trouble here.

In purely priority-driven systems with sequential thread processing within one priority level, we can simply extend the interrupt-handling control-flow level model to thread priorities, and synchronize with comparable mechanisms (explicit level switch, algorithmic).
(→ event-driven real-time systems)

Agenda

- Motivation
- Control-flow Level Model with Threads
- **Thread Synchronization**
  - **Constraints**
  - **Mutex, Implementation Variants**
  - Concept of Passive Waiting
  - Semaphore
- Example: Synchronization Objects on Windows
- Summary

Thread Synchronization: Overview

- Goal (for the user):
  Coordination of **resource accesses**
  - Coordinating exclusive access to reusable resources → **Mutex**
  - Interacting with / coordinating consumable resources → **Semaphore**

- Implementation approach (for the OS developer):
  Coordination of CPU allocation of **threads**
  - Particular threads are **not scheduled** temporarily.
    → “Waiting” as an OS concept

In the following, we focus on the OS developer’s perspective.

Mutex – Mutual Exclusion

- In general:
  An algorithm for enforcing mutual exclusion in a critical section
- Here:
  A system abstraction `class Mutex`
- Interface:
  - `void Mutex::lock()`
    - Enter and lock the critical section
    - Thread can block
  - `void Mutex::unlock()`
    - Leave and unlock the critical section
- Correctness condition: $0 \leq \sum_{exec} \text{lock()} - \sum_{exec} \text{unlock()} \leq 1$
  - At every point in time, there is at maximum one thread in the critical section.

Mutex: Usage

```
#include "BoundedBuffer.h"
#include "Mutex.h"
extern BoundedBuffer buf;
extern Mutex mutex;
```

```
void f() {
  ...
  char el;
  mutex.lock();
  el = buf.consume();
  mutex.unlock();
  ...
}
```

```
void g() {
  ...
  char el = ...
  mutex.lock();
  buf.produce( el );
  mutex.unlock();
  ...
}
```

Mutex: with Busy Waiting

- Implemented purely at user level; approach:
  - store state in boolean variable (0=free, 1=locked)
  - wait busily in `lock()` until variable is 0

```
// __atomic_test_and_set is a gcc builtin for
// (CPU specific) test-and-set
class SpinningMutex {
  char locked;
public:
  SpinningMutex() : locked (0) {}
  void lock(){
    while (__atomic_test_and_set(
           &locked, __ATOMIC_RELAXED))
      ;
  }
  void unlock() {
    locked = 0;
  }
};
```

```
lock:
    mov    $1,%dl
L2: mov    %edx,%eax
    xchg   %al,(%rdi)
    test   %al,%al
    jne    L2
    ret

unlock:
    movb   $0, (%rdi)
    ret
```

Assessment: Mutex with Busy Waiting

- **Advantages**
  - Maintains consistency, satisfies correctness condition
    - under the assumption of progress guarantee for all threads
  - Synchronization without involving the OS
    - No system calls necessary
- **Disadvantages**
  - Busy waiting wastes a lot of CPU time
    - at least until the time slice is used up
    - quite significant for time slices of 10–800ms!
    - Scheduler may “penalize” thread

Busy Waiting is, if at all, only an alternative on multiprocessor machines.

Mutex: with “Hard Synchronization”

- Implementation with “hard thread synchronization”
  - Approach:
    - Deactivate multitasking before entering the critical section
    - Reactivate multitasking after leaving the critical section
  - Necessitates a way to disable preemption
    - Special operations: forbid(), permit()

```
class HardMutex {
public:
  void lock(){
    forbid();    // disable multitasking
  }
  void unlock(){
    permit();    // enable multitasking
  }
};
```

Mutex: with "Hard Synchronization"

- Implementation of forbid() and permit()
  - e.g. in the scheduler
    - special, non-preemptible "real-time priority"
    - own priority level L¼ for the scheduler
    - resume() simply switches back to the caller

```
void forbid(){
  enter();
}
void permit(){
  leave();
}
```

- or simply on epilogue level
  - Context switching usually resides on epilogue level
    - Epilogue-level control flows are sequentialized
    - As long as a thread is on epilogue level, it cannot be preempted
  - Consequence: Sequentialization also with epilogues!

Assessment: Mutex with “Hard Synchronization”

- **Advantages**
  - Maintains consistency, satisfies correctness condition
  - Simple to implement
- **Disadvantages**
  - Broadband effect
    - Across-the-board delay of all threads (and potentially even epilogues!)
  - Priority violation
    - We delay control flows with higher priority.
  - Pessimistic
    - We put up with the disadvantages, although the collision probability is very low.

Assessment: Mutex with “Hard Synchronization”

- **Advantages**
  - Maintains consistency, satisfies correctness condition
  - Simple to implement
- **Disadvantages**
  - Bro
    - logues!)
  - Pric
    - We delay control flows with higher priority.
  - Pessimistic
    - We put up with the disadvantages, although the collision probability is very low.

Thread synchronization on epilogue level has many disadvantages. It is, however, appropriate for very short, seldomly entered critical sections – or if we need to synchronize with epilogues anyways.

Agenda

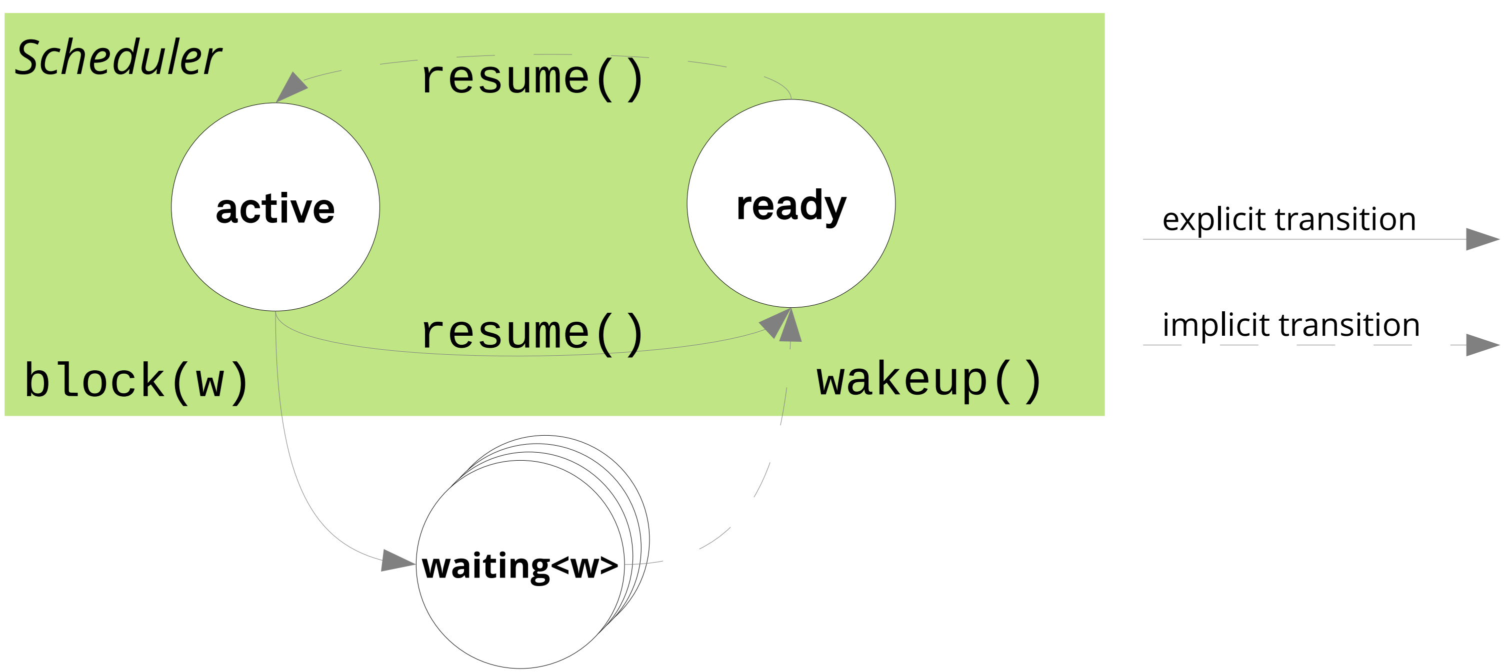
- Motivation
- Control-flow Level Model with Threads
- **Thread Synchronization**
  - Constraints
  - Mutex, Implementation Variants
  - **Concept of Passive Waiting**
  - **Semaphore**
- Example: Synchronization Objects on Windows
- Summary

Passive Waiting

- Previously shown Mutex implementations are not ideal
  - Mutex with **busy waiting:** wastes CPU time
  - Mutex with **hard synchronization:** coarse-grained, violating priorities
- Better approach: **Exclude thread from CPU scheduling** as long as the mutex is locked
- Necessitates new OS concept: **passive waiting**
  - Threads can "wait passively" for an event
    - Wait passively → be excluded from CPU scheduling
    - New thread state: **waiting** (for an event)
  - Occurrence of an event triggers leaving the waiting state
    - Thread is included in CPU scheduling
    - Thread state: **ready**

OS Concept: Passive Waiting

- Necessary abstractions:
  - Scheduler operations: **`block()`**, **`wakeup()`**
  - Synchronization object: **`Waitingroom`**
    - represents the event to wait for
    - usually a waiting queue of waiting threads



OS Concept: Passive Waiting

- Scheduler operations
  - **`block(Waitingroom& w)`**
    - enqueue active thread (caller) in queue of synchronization object w
    - activate another thread (from ready list)
  - **`wakeup(Customer& t)`**
    - enqueue t in ready list
- Waitingroom operations
  - **`enqueue(Customer*)`**
  - **`Customer* dequeue()`**

It makes sense to manage the queue with the **same prioritization strategy** as the scheduler's ready list!

Mutex: with Passive Waiting

```
class WaitingMutex : public Waitingroom {
  char locked;
public:
  WaitingMutex() : locked(0) {}
  void lock() {
    while (__atomic_test_and_set(&locked, __ATOMIC_RELAXED))
      scheduler.block(*this);
  }
  void unlock() {
    locked = 0;
    // fetch possibly waiting thread and wake it up
    Customer *t = dequeue();
    if (t)
      scheduler.wakeup(*t);
  }
};
```

This solution still has one remaining problem …

Mutex: with Passive Waiting

```
class WaitingMutex : public Waitingroom {
  char volatile locked;
public:
  WaitingMutex() : locked(0) {}
  void lock() {
    mutex.lock();
    while (locked == 1)
      scheduler.block(*this);
    locked = 1;
    mutex.unlock();
  }
  void unlock() {
    mutex.lock();
    locked = 0;
    // fetch possibly waiting thread and wake it up
    Customer *t = dequeue();
    if (t) scheduler.wakeup(*t);
    mutex.unlock();
  }
};
```

`lock()` and **`unlock()`** are critical sections themselves

Can we protect these critical sections with a Mutex?

Mutex: with Passive Waiting

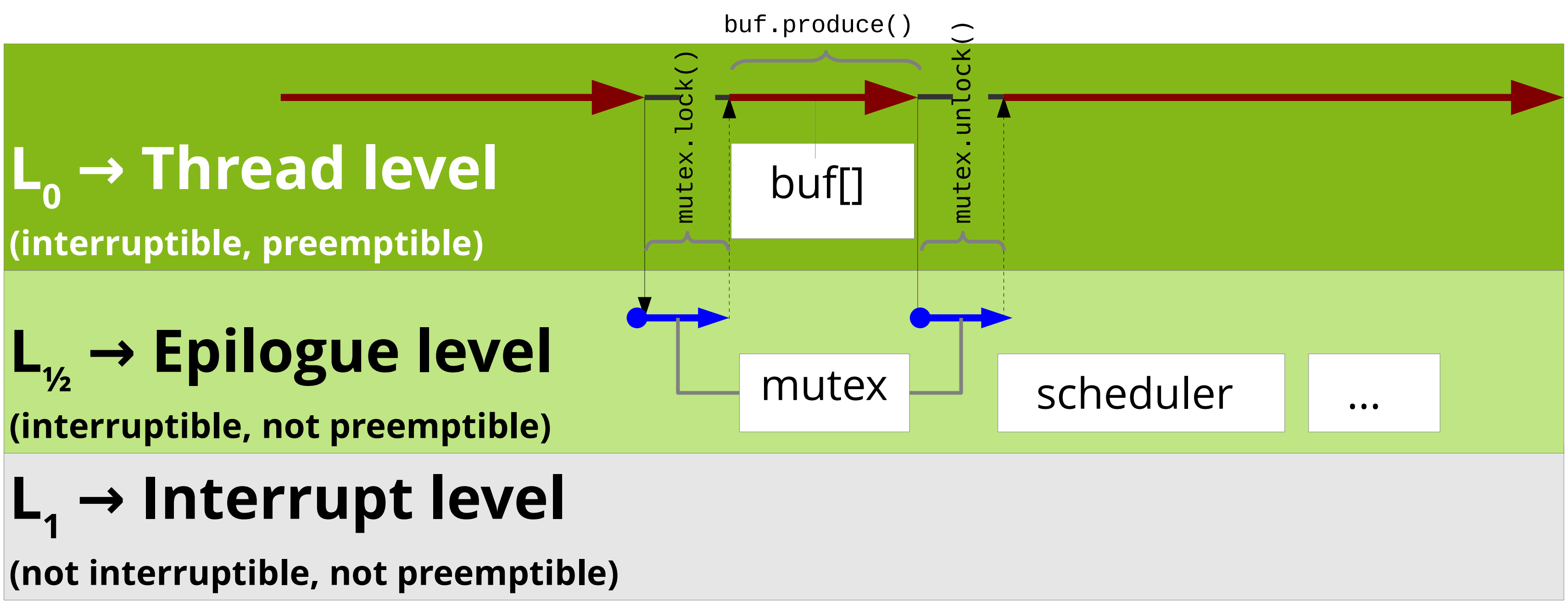
```
class WaitingMutex : public Waitingroom {
  char volatile locked;
public:
  WaitingMutex() : locked(0) {}
  void lock() {
    enter();
    while (locked == 1)
      scheduler.block(*this);
    locked = 1;
    leave();
  }
  void unlock() {
    enter();
    locked = 0;
    // fetch possibly waiting thread and wake it up
    Customer *t = dequeue();
    if (t) scheduler.wakeup(*t);
    leave();
  }
};
```

It works with a **`HardMutex`**!

The common solution is indeed to protect **`lock()`** and **`unlock()`** on the **epilogue level**, as shown here.

Conclusion: Implementing Waiting

- Mutex state resides **in the kernel** on epilogue level
  - more precisely: on the same level as the scheduler state
- This is a **generic principle**
  - Implementation of synchronization mechanisms for L_0 control flows is synchronized on $L_{½}$



Semaphore

- Semaphore is *the* classic synchronization object
  - Edsger W. Dijkstra, 1962 [2]
  - in many OSs: Basis for all other synchronization objects
  - for us: semaphore := synchronization object + counter
- Operations
  - 2 standard operations (with various names [2,3,5])
  - **`prolaag()`**, **`P()`**, **`wait()`**, **`down()`**, **`acquire()`**, **`pend()`**
    - if counter > 0, decrease counter
    - if counter ≤ 0, **wait** until counter > 0 and retry
  - **`verhoog()`**, **`V()`**, **`signal()`**, **`up()`**, **`release()`**, **`post()`**
    - increase counter
    - if counter = 1, **wake up** possibly waiting thread
- Many variants

Implementation of the standard variant in the exercises.

Semaphore: Usage

- Semaphore semantics are particularly suitable for implementing producer/consumer scenarios
  - i.e. coordinated access to **consumable resources**
    - Characters from the keyboard
    - Signals that are supposed to be processed further on thread level
    - …
  - Internal counter represents the resource count
    - Producer calls **`V()`** for each produced element.
    - Consumer calls **`P()`** to consume an element, possibly waits.

`P()` can block on thread level, **`V()`** never blocks!

Hence, a control flow on **epilogue** or **interrupt level** can also be a **producer** (assuming appropriate synchronization of the internal semaphore state.)

Semaphore vs. Mutex

- Mutex is often understood as a **two-valued semaphore**
  - Mutex → Semaphore with initial counter value 1
  - `lock()` → `P()`, `unlock()` → `V()`
- However, the semantics are different:
  - A locked mutex (implicitly or explicitly) has an **owner**
    - Only this owner may call `unlock()`.
    - Mutex implementations e.g. on Linux or Windows check this.
  - A mutex can (usually) also be **locked recursively**.
    - Internal counter: The same thread may call `lock()` multiple times; after a matching number of `unlock()` calls, the mutex is unlocked again.
  - In contrast, a semaphore can be incremented or decremented by any thread.

In many operating systems, the semaphore is the **basic abstraction** for synchronization objects. It is used as an **implementation basis** for mutexes, condition variables, reader-writer-locks, ...

Agenda

- Motivation
- Control-flow Level Model with Threads
- Thread Synchronization
  - Constraints
  - Mutex, Implementation Variants
  - Concept of Passive Waiting
  - Semaphore
- **Example: Synchronization Objects on Windows**
- Summary

Synchronization on Windows

- Windows takes the idea of waiting objects quite far
  - Every kernel object is also a synchronization object
    - explicit synchronization objects: Event, mutex, timer, semaphore
    - implicit synchronization objects: File, socket, thread, process, ...
  - Waiting semantics depends on the object
    - Thread waits for “signaled” state
    - State is, if applicable, modified by successful waiting
- Uniform system interface for all object types
  - Kernel object is represented by a HANDLE
  - **`WaitForSingleObject(hObject, dwMillisec)`**
    - Wait for synchronization object with timeout
  - **`WaitForMultipleObjects(nCount, hObjects[], bWaitAll, dwMillisec)`**
    - Wait for one or more synchronization objects with timeout (“and”/“or” waiting, depending on bWaitAll = true/false)

Synchronization Objects on Windows

Object Type	Signaled when	Successful waiting results in
Event	Explicit state change (SetEvent()/ResetEvent())	Event reset (for AutoReset events)
Mutex	Mutex is available	Mutex is owned
Semaphore	Semaphore counter > 0	Semaphore is decreased by 1
Waitable timer	Specific point in time reached	Timer reset (for AutoReset timers)
Change notification	Specific change in the file system	–
Console input	Input data available	–
Process	Process has terminated	–
Thread	Thread has terminated	–
File	An asynchronous file op. finished	–
Serial device	Data available / file op. finished	–
Named pipe	An asynchronous op. finished	–
Socket	An asynchronous op. finished	–
Job (Win 2000)	All processes of the job terminated	–

Synchronization and Costs

- Synchronization objects are managed in the kernel
  - Critical data structures → protection
  - Internal synchronization on epilogue level → consistency
- This can make their use very costly:
  - We need to switch to the kernel for each state change.
  - User/kernel mode transitions are very expensive.
  - IA-32/x86-64: several hundred cycles!
- These costs are particularly pronounced for mutexes:
  - The time needed for locking/unlocking mutexes is often a multiple of the time the critical section is locked.
  - Actual contention (thread wants to enter an already locked section) rarely occurs.

Synchronization and Costs

- Approach: Manage mutex as far as possible in **user mode**
  - Minimize the normal-case cost
    - Normal case: critical section is free
    - Special case: critical section is locked
- Introduce a **fast path** for the normal case
  - Test, locking and unlocking in user mode
    - Ensure consistency algorithmically / with atomic CPU instructions
  - Wait in kernel mode
    - We need the kernel for the transition to the passive waiting state
  - Further optimization for multiprocessor machines
    - Busily wait for limited amount time before waiting passively
    - High probability that the critical section is free before

Windows: CRITICAL_SECTION

- Structure for a **fast mutex** in user mode [8]
  - Internally uses an Event (kernel object) in case we must wait
  - Lazy (on-demand) Event creation
- Specific system-call interface
  - **`EnterCriticalSection(pCS)`** / **`TryEnterCriticalSection(pCS)`**
    - Lock critical section (blocking) / try locking critical section (non-blocking)
  - **`LeaveCriticalSection(pCS)`**
    - Leave critical section
  - **`SetCriticalSectionSpinCount(pCS, dwSpinCount)`**
    - Define number of tries for busy waiting (multiprocessor systems only)

```
typedef struct _CRITICAL_SECTION {
  LONG LockCount;      // Number of waiting threads (-1 when free)
  LONG RecursionCount; // Number of successful EnterXXX calls
  DWORD OwningThread;  // Owner thread
  HANDLE LockEvent;    // Internal synchronization object, created on demand
  ULONG SpinCount;     // On MP systems: number of busy-wait tries until we
                       // passively wait in the kernel
} CRITICAL_SECTION, *PCRITICAL_SECTION;
```

TECHNISCHE UNIVERSITÄT DRESDEN

Windows: CRITICAL_SECTION

- Structure for a **fast mutex** in user mode [8]
  - Internally uses an Event (kernel object) in case we must wait
  - Lazy (on-demand) Event creation
- Specific system-call interface
  - **`EnterCriticalSection(pCS)`** / **`TryEnterCriticalSection(pCS)`**
    - Lock critical section (blocking) / try locking critical section (non-blocking)
  - **`LeaveCriticalSection(pCS)`**
    - Leave critical section
  - **`SetCriticalSectionSpinCount(pCS, dwSpinCount)`**
    - Define number of tries for busy waiting (multiprocessor systems only)

```
typedef struct _CRITICAL_SECTION {
  LONG LockCount;      // Number of waiti
  LONG RecursionCount; // Number of succe
  DWORD OwningThread;  // Owner thread
  HANDLE LockEvent;    // Internal synchr
  ULONG SpinCount;     // On MP systems:
                       // passively wait in the kernel
} CRITICAL_SECTION, *PCRITICAL_SECTION;
```

With ***Futexes*** (*Fast user-mode mutexes*), Linux 2.6 introduced a comparable but much more powerful concept. [7,6]

Agenda

- Motivation
- Control-flow Level Model with Threads
- Thread Synchronization
  - Constraints
  - Mutex, Implementation Variants
  - Concept of Passive Waiting
  - Semaphore
- Example: Synchronization Objects on Windows
- **Summary**

Summary

- Threads can be preempted at any time
  - Preemptive, probabilistic multitasking
  - No run-to-completion semantics
  - Access to shared state must be separately synchronized
- Thread synchronization: Many variants
  - Mutex for mutual exclusion
  - Semaphore for producer/consumer scenarios
  - Many other abstractions possible: reader/writer locks, semaphore vectors, condition variables, timeouts, ...
- Based on an OS concept for passive waiting
  - Fundamental thread property: They can wait.
  - Busy waiting and “hard” thread synchronization only make sense in exceptional cases.

Bibliography

[1] K. R. Apt. *Edsger Wybe Dijkstra (1930 – 2002): A Portrait of a Genius.* http://arxiv.org/pdf/cs.GL/0210001, 2002.

[2] E. W. Dijkstra. *Multiprogrammering en de X8*, 1962. [4].

[3] E. W. Dijkstra. *Cooperating Sequential Processes.* Technical report, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1965. (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996).

[4] E. W. Dijkstra. *EWD Archive: Home.* http://www.cs.utexas.edu/users/EWD, 2002.

[5] P. B. Hansen. *Betriebssysteme.* Carl Hanser Verlag, erste Edition, 1977. ISBN 3-446-12105-6.

[6] Ulrich Drepper. *Futexes are tricky.* http://people.redhat.com/drepper/futex.pdf, 2005

[7] Hubertus Franke, Rusty Russell, Matthew Kirkwood. *Fuss, futexes and furwocks: Fast Userlevel Locking in Linux*, Ottawa Linux Symposium. https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf, 2002.

[8] Matt Pietrek, Russ Osterlund. *Break Free of Code Deadlocks in Critical Sections Under Windows.* MSDN Magazine https://docs.microsoft.com/en-us/archive/msdn-magazine/2003/december/..., 2003