

# OPERATING-SYSTEM CONSTRUCTION

## *Exercise 6: Task #6, Idle Loop, Non-Bl. Thread Sync*

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

**TILL SMEJKAL (slides by HORST SCHIRMEIER)**

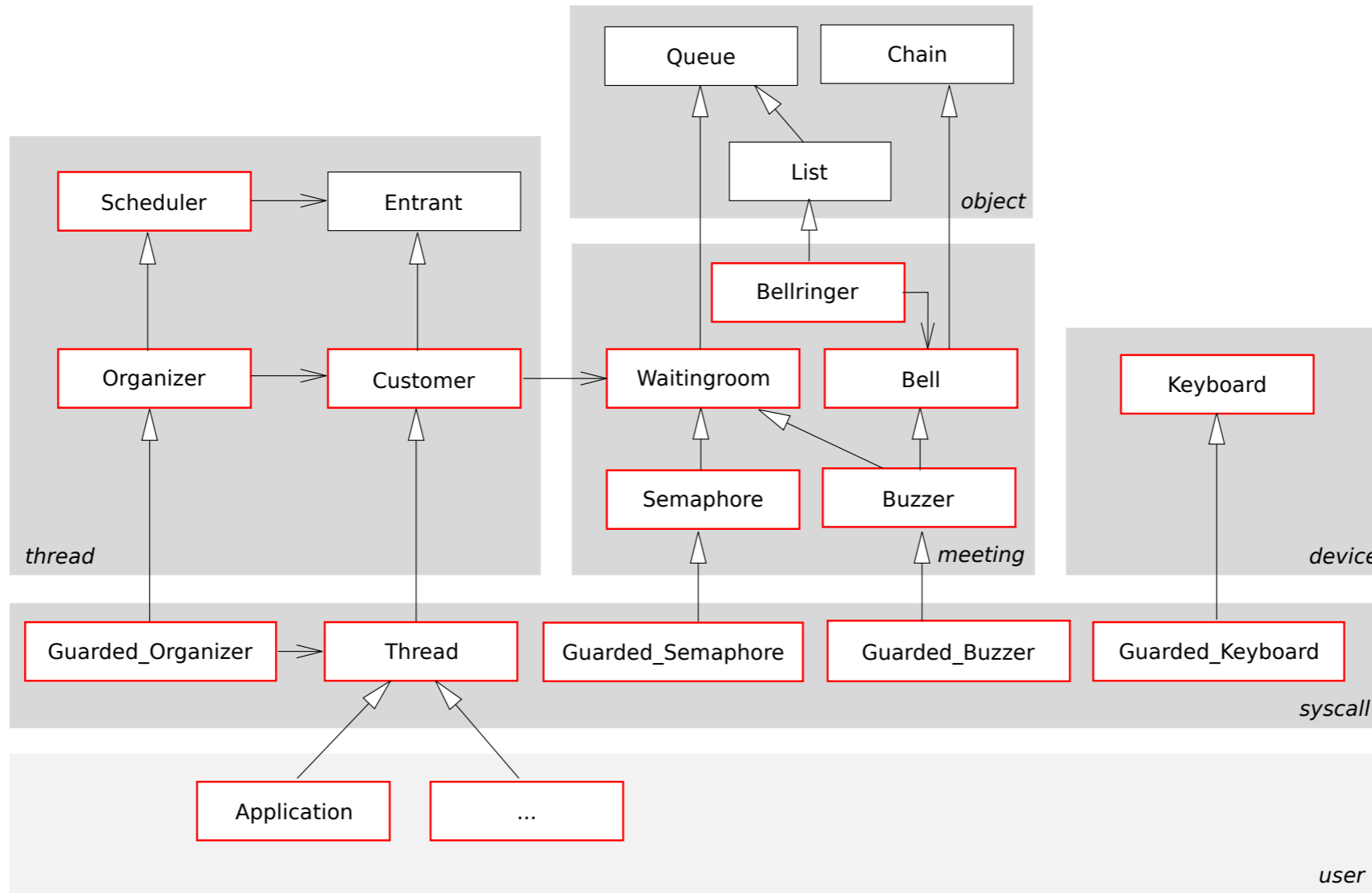
# Agenda

- Lab Task #6
- *Idle-Loop, considered harmful*
- Non-Blocking Thread Synchronization

# Agenda

- **Lab Task #6**
- *Idle-Loop, considered harmful*
- Non-Blocking Thread Synchronization

# Lab Task #6: The Art of Waiting



# Lab Task #6

- Entrant → **Customer**
  - Can wait for specific events
- Each event is assigned to a **Waitingroom**
  - Threads that wait for an event are queued in its `Waitingroom`
  - Synchronization objects are `Waitingrooms` and can trigger events
- Scheduler → **Organizer**
  - Can **block / “put to sleep”** a thread (*Readylist* → *Waitingroom*)
    - `block(Thread &, Waitingroom &)`
  - and **“wake it up”** again (*Waitingroom* → *Readylist*)
    - `wakeup(Thread &)`
- Events in OOSTuBS
  - Semaphore `V( )` + and other thread is waiting (in `P( )`)
  - A key was added to the keyboard buffer
  - A specific amount of time has passed

# Synchronization Object Semaphore

- Derived from **Waitingroom**
- **p()**
  - If == 0: **Wait** for v() (**wait**)
    - using the Organizer
  - Else: decrease by 1
- **v()**
  - If a thread is waiting: **Signal the event (signal)**
    - Wake up waiting thread
    - What happens if multiple threads are waiting?
  - Else: increase by 1

# Synchronization Object Keyboard

- Goal: Use the CPU for other purposes while waiting for I/O
- Thread reads from the keyboard
  - Keyboard driver's `getKey()` returns Keys
  - as long as there are some in the (software) keyboard buffer
  - When keyboard buffer is empty:
    - Thread blocks
    - Waits for event "Keyboard buffer filled again" (**wait**)
  - Signaling of this event (**signal**)
    - Keyboard interrupt
    - Epilogue, due to access from thread level
- Implementation
  - Semaphore that counts keys in the keyboard buffer

# Synchronization Object Buzzer

- **Buzzer**: an alarm clock
  - With `sleep()` threads can block and wait until this alarm clock rings
  - After a period of time specified in `set()`
  - the `ring()` method wakes up waiting threads
- derived from **Bell**
  - Has a counter
  - that is counted down with `tick()`
  - and calls `ring()` when run down (`run_down() == true`)
- **Bellringer**
  - manages `Bell` objects
  - regularly checks whether they have run down and rings them in this case
- Implementation:
  - without a detour over `Semaphore`
  - directly with `Waitingroom` and `Organizer` (why?)

# Synchronization Objects in OOSTuBS

- ... are part of the kernel state
  - Keyboard and Buzzer signal events in the epilogue
    - Can we also wait for events in the epilogue?
  - Semaphore (why?)
- ... and therefore must reside on the epilogue level
  - Guarded\_Semaphore
  - Guarded\_Buzzer
  - Guarded\_Keyboard

# Agenda

- Lab Task #6
- ***Idle-Loop, considered harmful***
- Non-Blocking Thread Synchronization

# Idling

- All threads, except one, are waiting for an event.
- Now the last thread also blocks. What now?
  - Busy waiting until one thread is ready again?
    - Definitely makes sure the CPU stays warm ...
  - Solution: `cpu.idle()`
    - Runs, like `cpu.halt()`, a `hlt` instruction, but enables interrupts before instead of disabling them.
    - When an interrupt occurs, its handler runs, and then the CPU continues execution after the `hlt`.
    - ... and then?

```
cpu_idle:  
sti  
hlt  
ret
```

```
while (!(next=readylist.dequeue()))  
    cpu.idle();
```

Unfortunately, it's  
not *that* simple.

# Agenda

- Lab Task #6
- *Idle-Loop, considered harmful*
- **Non-Blocking Thread Synchronization**

# 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.

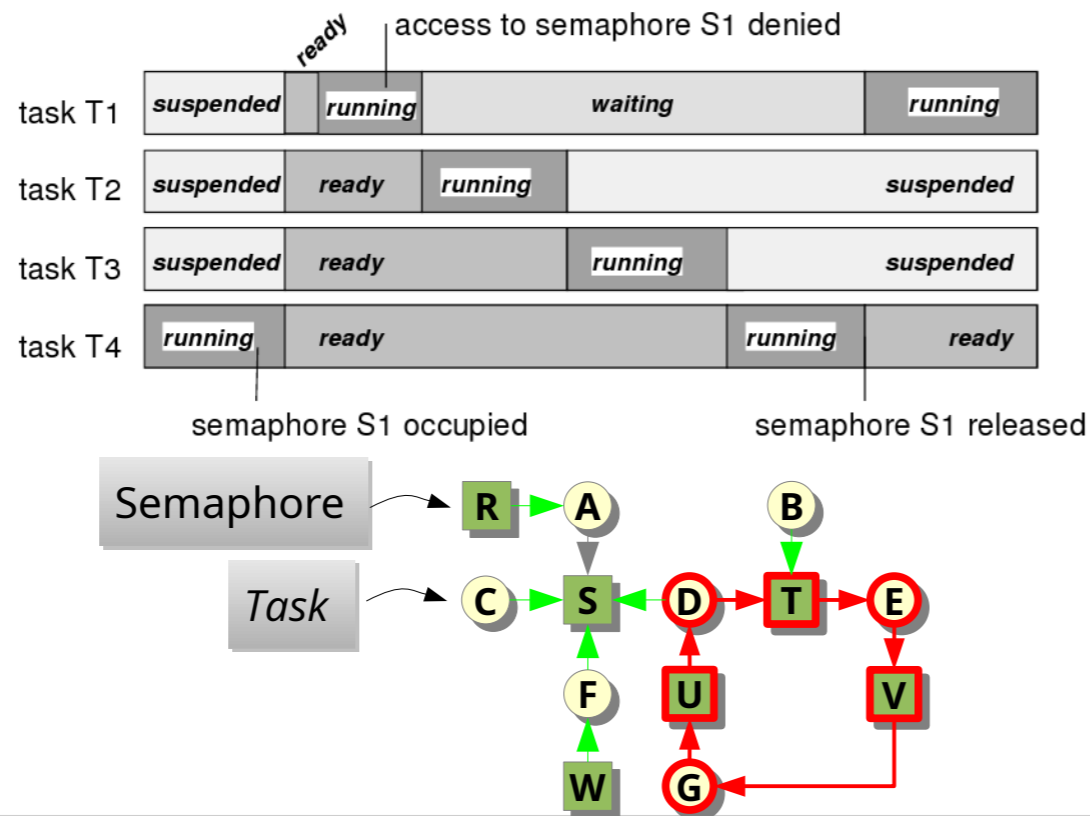
In purely priority-driven systems with sequential thread processing within one priority level, we can simply extend the interrupt-handling control-flow level model to thread priorities, and synchronize with comparable mechanisms (explicit level switch, algorithmic).  
(→ event-driven real-time systems)

# Why all the Fuss with Threads?

- Assume we don't need "progress guarantee"
- Several application levels
  - Instead of threads: one control flow per level
- Do we still need coroutines?
  - What **can't** we do without them?
- Example: OSEK / AUTOSAR-OS
  - Instead of semaphores or mutexes: so-called "resources"
  - Synchronization without blocking

# OSEK-OS: Resource Management (1)

- Synchronization when accessing shared resources, e.g. global variables, I/O devices, ...
- Avoids known issues of semaphores:

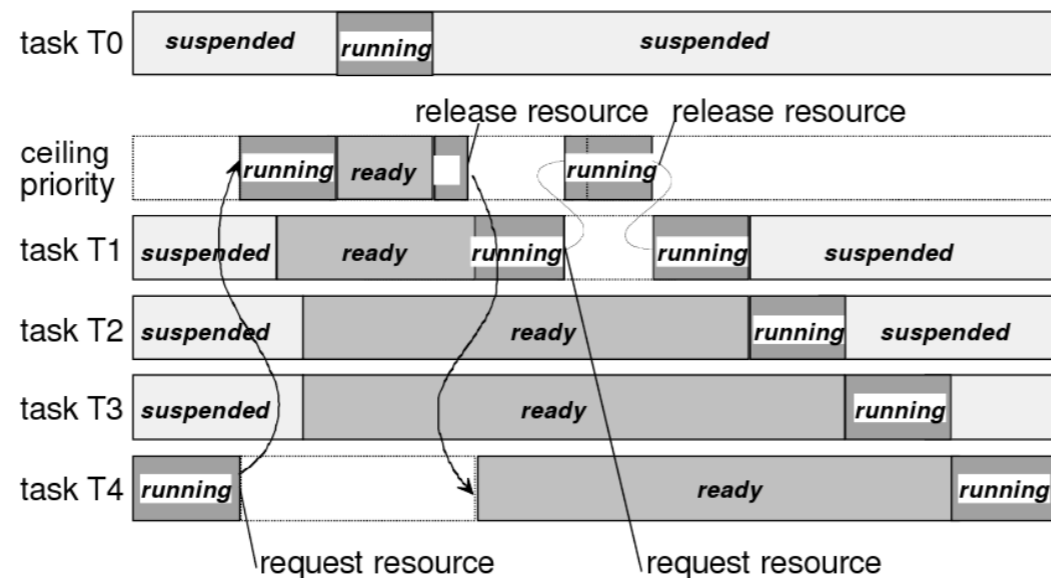


**Priority Inversion**  
 Because T4 occupies the semaphore, T2 and T3 (which have nothing to do with the semaphore!) indirectly delay the higher-prioritized T1 – because T4 holds the semaphore but cannot continue running yet.

**Deadlock**  
 We have a cycle in the resource-allocation graph. None of the involved tasks runs anymore.

# OSEK-OS: Resource Management (2)

- The OSEK **Priority Ceiling Protocol**
    - OSEK statically assigns a **ceiling priority** to each resource:  
Maximum of the priorities of all tasks that access the resource
    - When a task requests a resource, its priority is raised to the ceiling priority.
- Blocking becomes impossible!**
- After releasing the resource, the original priority is restored.



'GetResource' never blocks.  
Consequently we cannot run  
into a deadlock.

As long as T4 occupies the  
resource, it cannot be  
preempted by T2 or T3.  
Therefore we avoid priority  
inversion.