Constructing and Verifying Cyber Physical Systems

Mixed Criticality Scheduling and Real-Time Operating Systems

Marcus Völp

Overview

Introduction Mathematical Foundations (Differential Equations and Laplace Transformation)	Math
Control and Feedback	Physics
Transfer Functions and State Space Models Poles, Zeros / PID Control Stability, Root Locust Method, Digital Control	Feedback Control
Mixed-Criticality Scheduling and Real-Time Operating Systems (RTOS)	RTOS
Coordinating Networked Cyber-Physical Systems	CPS
Program Verification Differential Dynamic Logic and KeYmaera X Differential Invariants	Verification

Overview

TECHNISCHE UNIVERSITÄT DRESDEN

Cyclic Executives

↓

Real-Time Operating Systems

↓

The Case for Virtualization

↓

Scheduling Theory 101

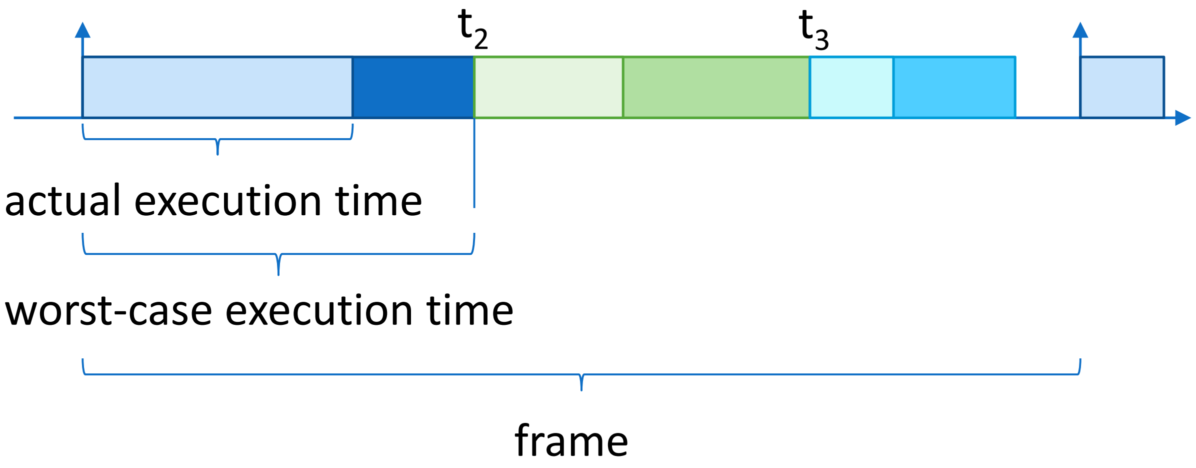
↓

Mixed Criticality

Real-Time Systems
Correctness not only depends on computed values, but also on their timeliness.

Cyclic Executive

```
int main(void) {

        // initialization code
        while (true) {
                task1;
                wait_until(t2);
                task2;
                wait_until(t3);
                …
        }
}
```



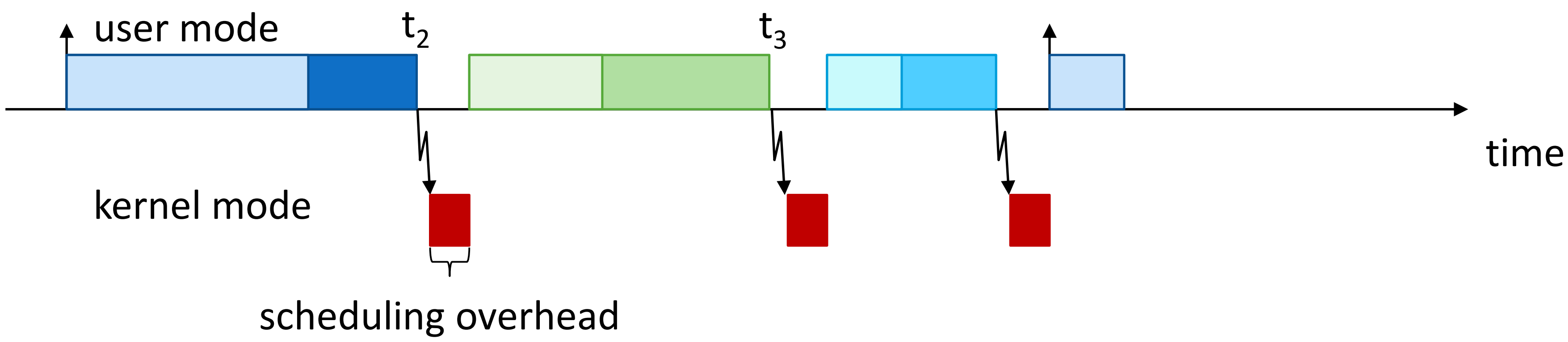
cooperative scheduling:

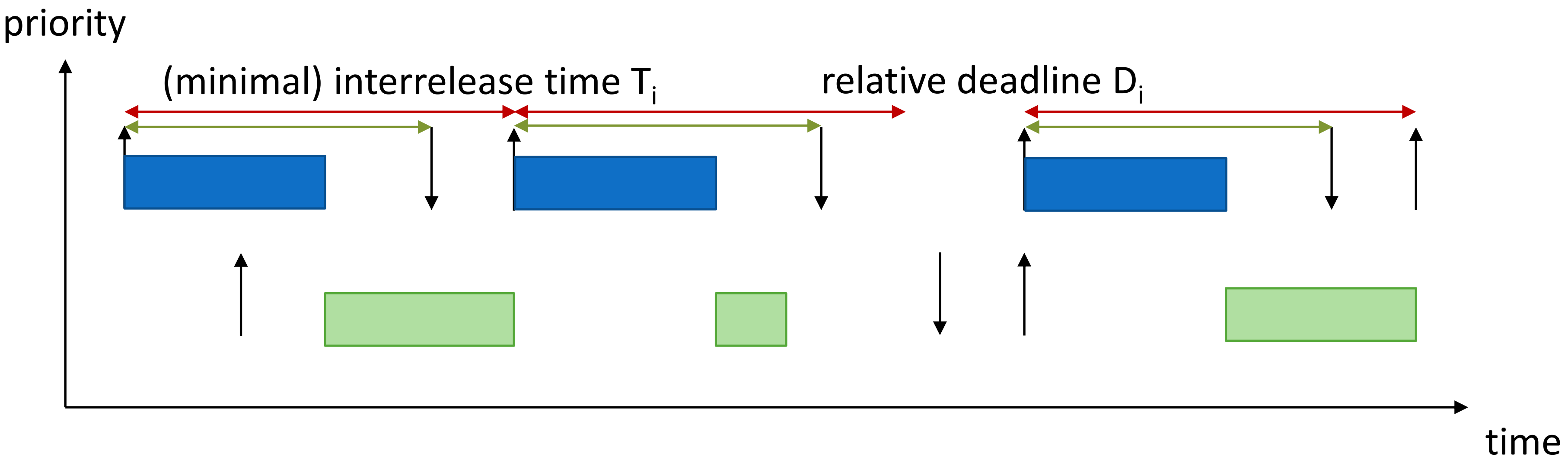
- tasks finish voluntarily
- errors or bugs in one task may jeopardize the entire system

Example: Arduino Scheduler

```
typedef void (*task_fn_t)(void);

struct Task {
        task_fn_t function;
        uint16_t interval_ticks;
        uint16_t max_time_micros;
};

Task _tasks[num_tasks];

/* Scheduler
 *===========
 * run one tick; this will run as many scheduler tasks as we can in the specified time
 */
void AP_Scheduler::run(uint16_t time_available)
{
   uint32_t run_started_usec = hal.scheduler->micros();
   uint32_t now = run_started_usec;

   for (uint8_t i=0; i<_num_tasks; i++) {
     uint16_t dt = _tick_counter - _last_run[i];
     uint16_t interval_ticks = pgm_read_word(&_tasks[i].interval_ticks);
     if (dt >= interval_ticks) {
       // this task is due to run. Do we have enough time to run it?
       _task_time_allowed = pgm_read_word(&_tasks[i].max_time_micros);

       if (dt >= interval_ticks*2) {
         // we've slipped a whole run of this task!
         ...
       }
     }

       if (_task_time_allowed <= time_available) {
         // run it
         _task_time_started = now;
         task_fn_t func = (task_fn_t)pgm_read_pointer(&_tasks[i].function);
         current_task = i;
         func();
         current_task = -1;

         // record the tick counter when we ran. This drives
         // when we next run the event
         _last_run[i] = _tick_counter;


         // work out how long the event actually took
         now = hal.scheduler->micros();
         uint32_t time_taken = now - _task_time_started;

         if (time_taken > _task_time_allowed) {
           // the event overran!
           ...
         }
         if (time_taken >= time_available) {
           ...
         }
         time_available -= time_taken;
       }
     }
   }
```

Preemptive Scheduling





Real-Time Operating Systems

TECHNISCHE UNIVERSITÄT DRESDEN

Functionality

Thread / Task / Process: Abstraction for execution

(sometimes also for resources shared by thread)

Scheduling: When to run which task and for how long?

Mutexes / Semaphores: Protection of resource accesses

- Mutex: single thread may enter critical section
- Semaphore: initial count determines how much threads can enter
  (count > 0 => semaphore is free; count =< 0 => semaphore is blocked)

Timers: Used internally for scheduling

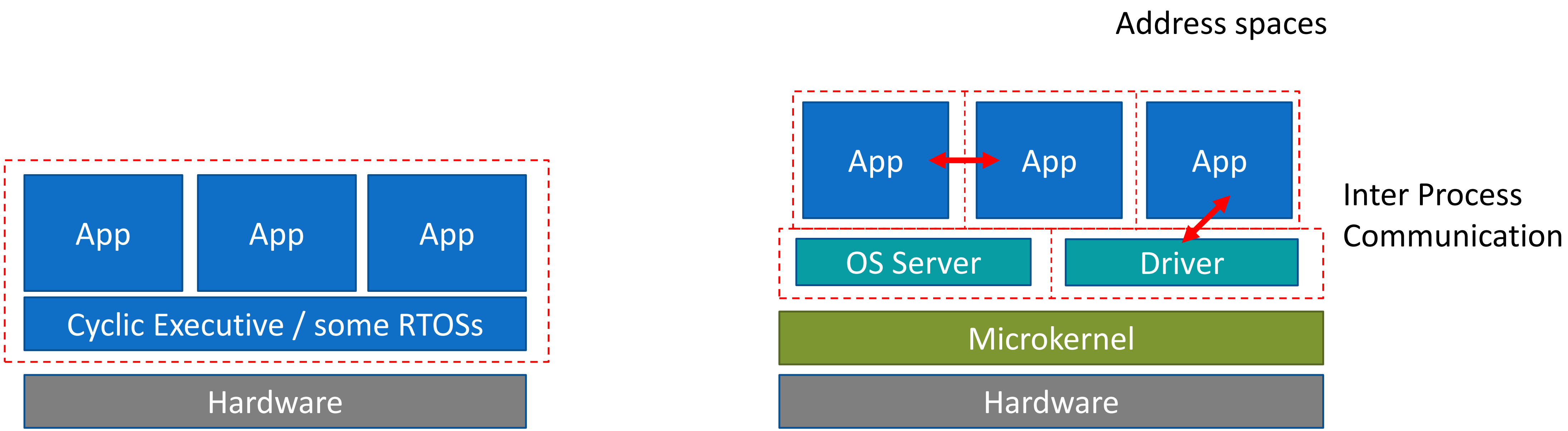
Timeouts (e.g., when accessing a device)

Microkernels

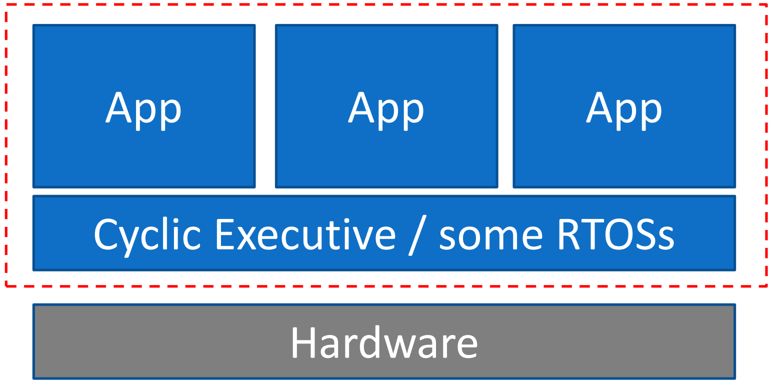
Functionality

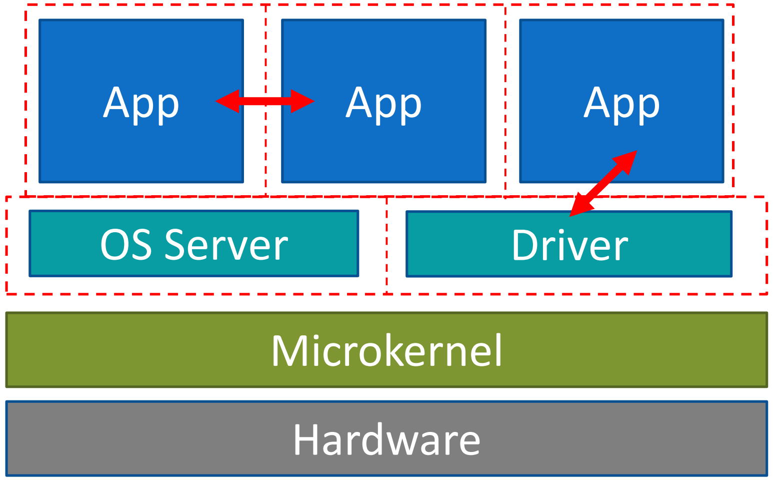
Isolation

Message passing to overcome isolation boundaries



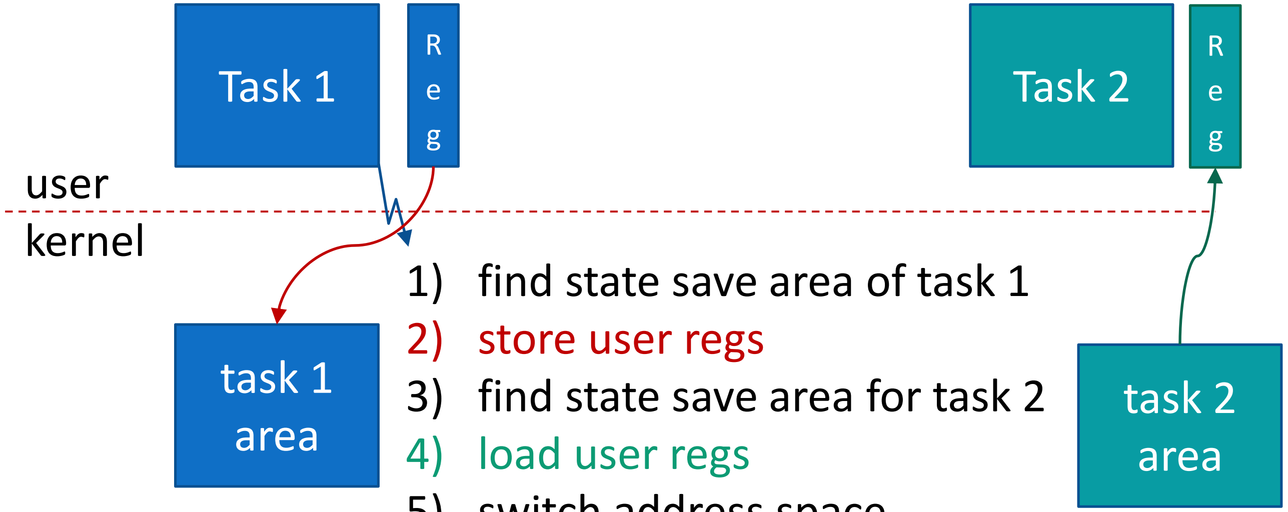
Context Switches





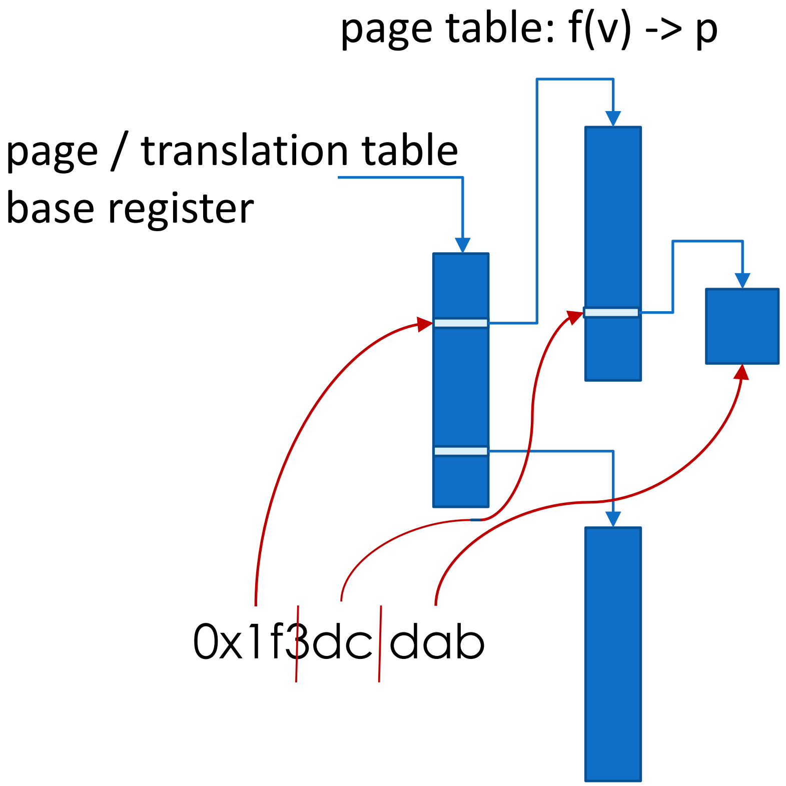

```
while (true) {
        task1();
        1:
        wait_until(t2);
        task2();
        2:
        wait_until(t3);
        …
}
```

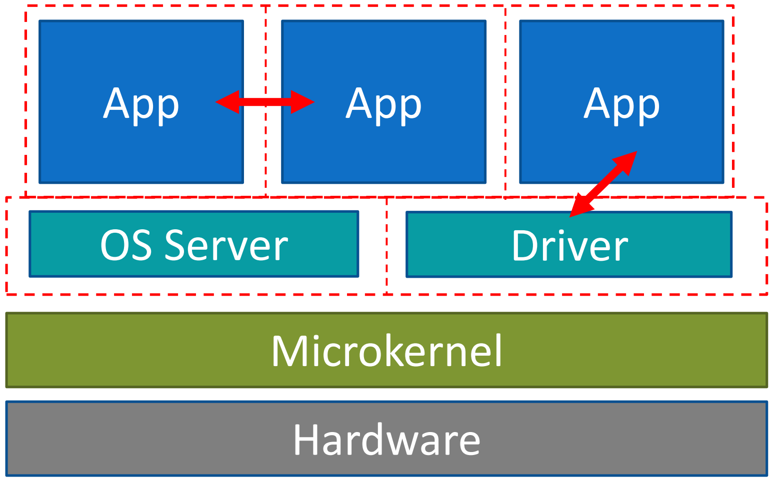
call task1
=
push 1f
jmp task1

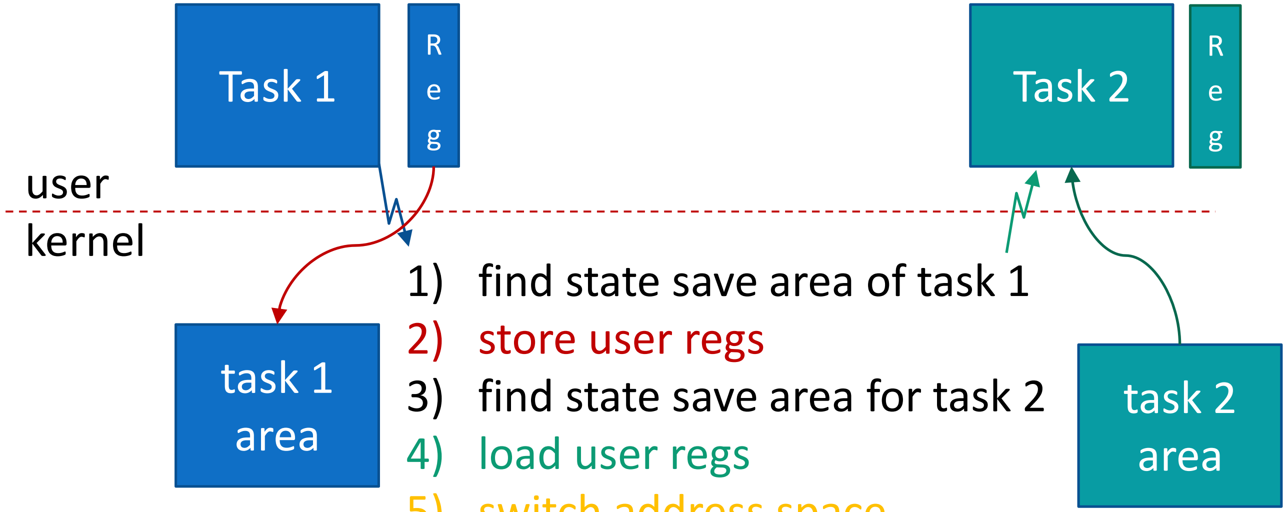


1) find state save area of task 1
2) store user regs
3) find state save area for task 2
4) load user regs
5) switch address space
6) exit from kernel

Context Switches

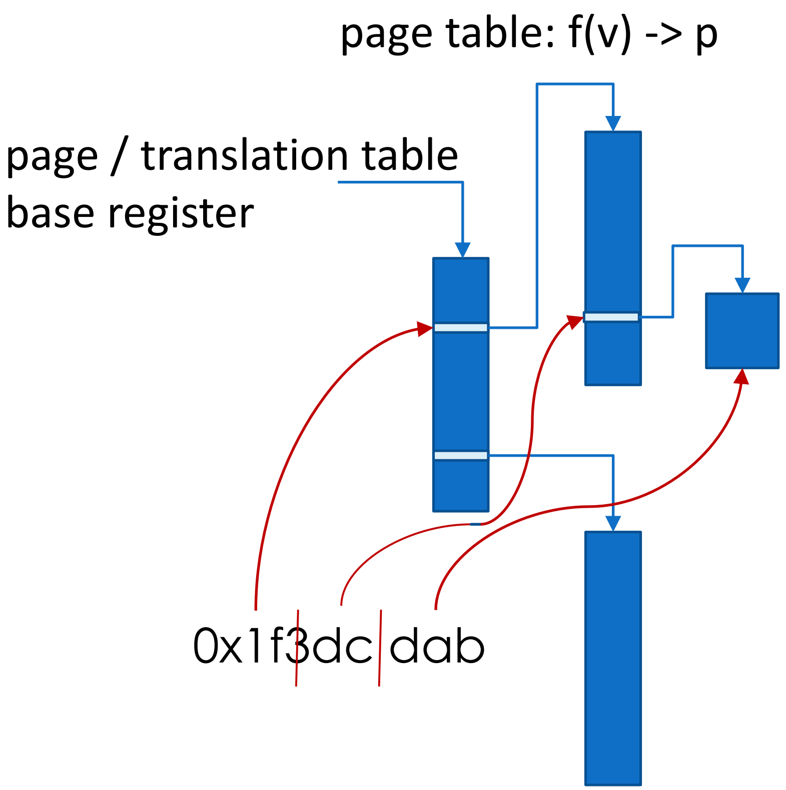


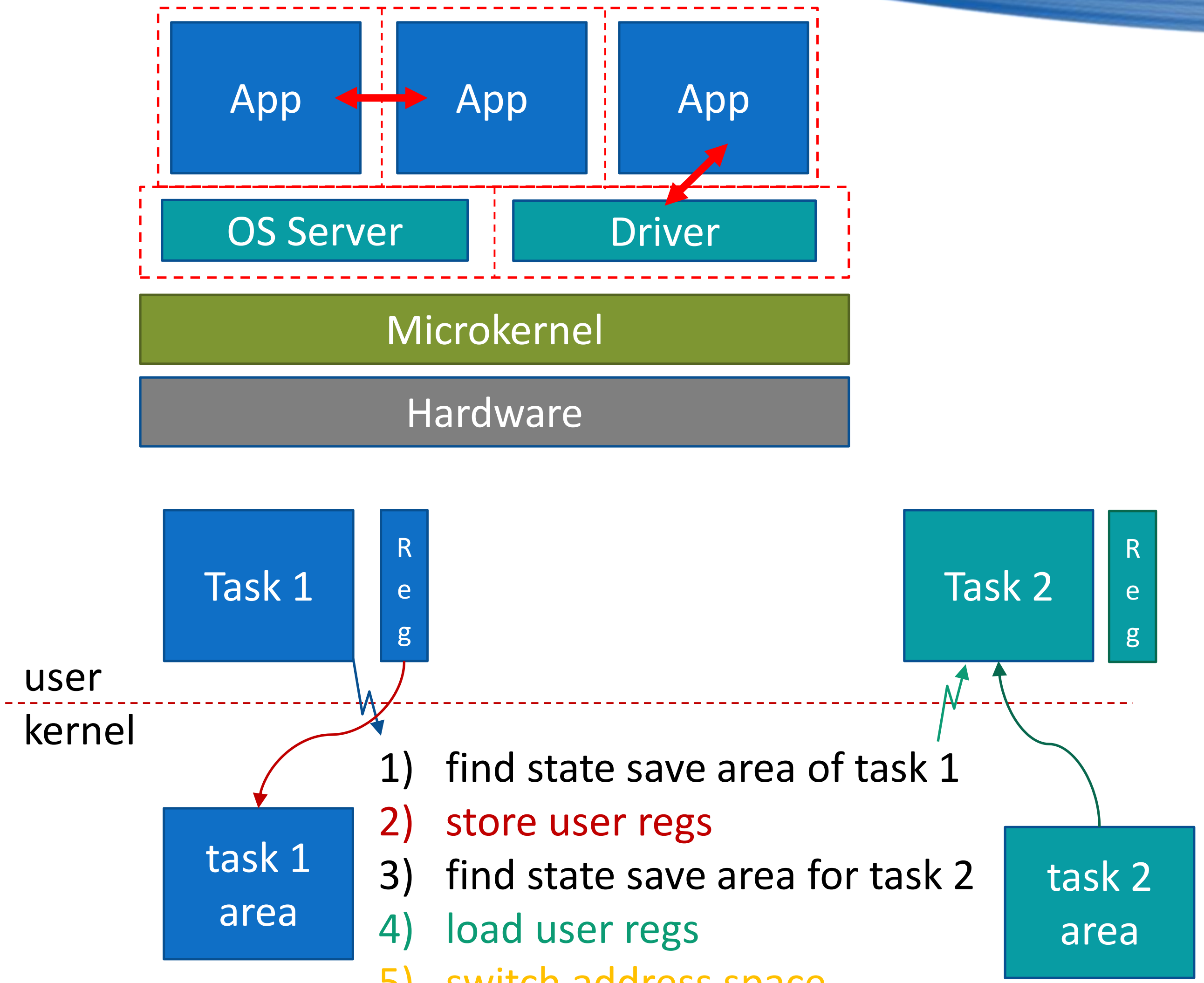




1) find state save area of task 1
2) store user regs
3) find state save area for task 2
4) load user regs
5) switch address space
6) exit from kernel

Context Switches



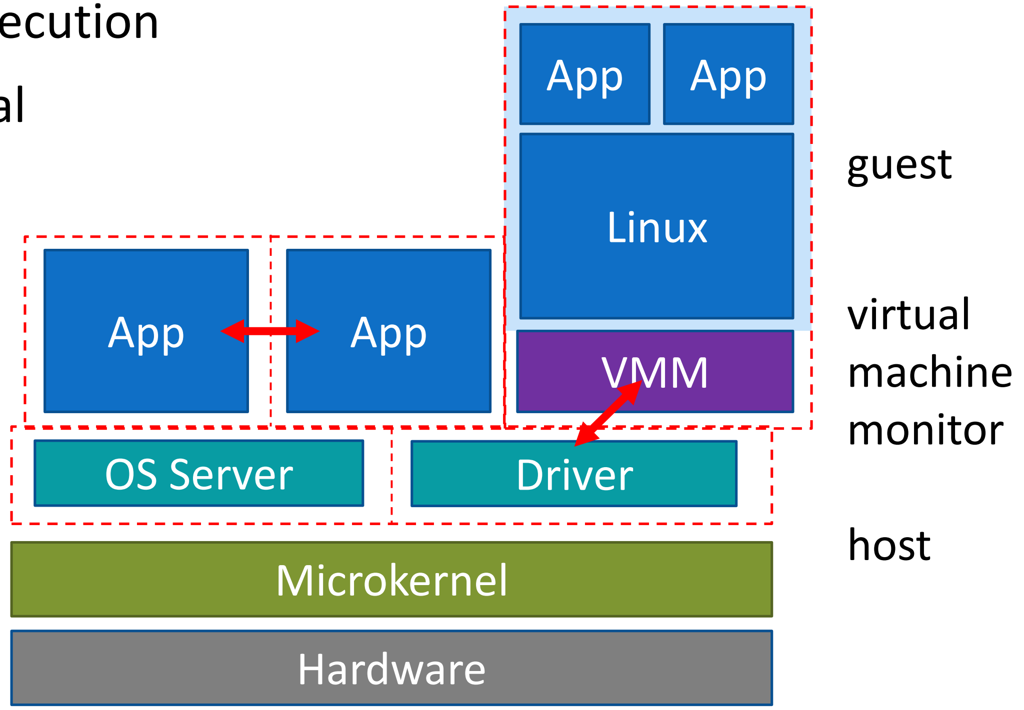


1) find state save area of task 1
2) store user regs
3) find state save area for task 2
4) load user regs
5) switch address space
6) exit from kernel

The Case for Virtualization

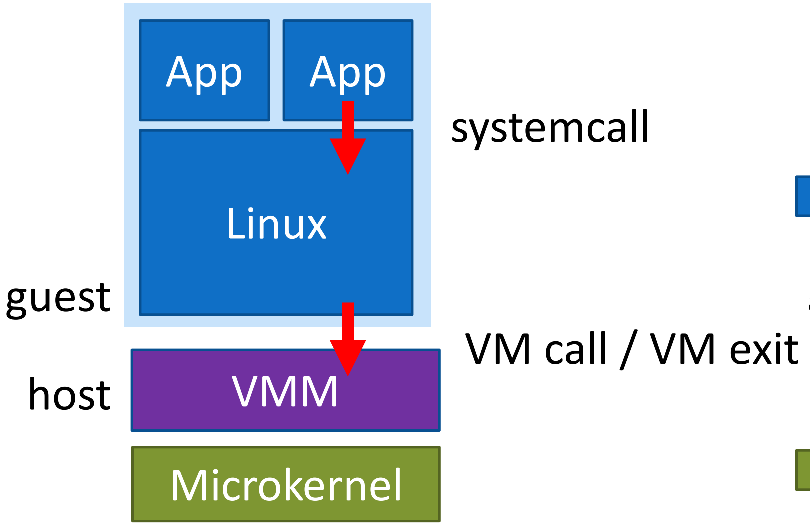
Extended functionality requires a certain amount of code

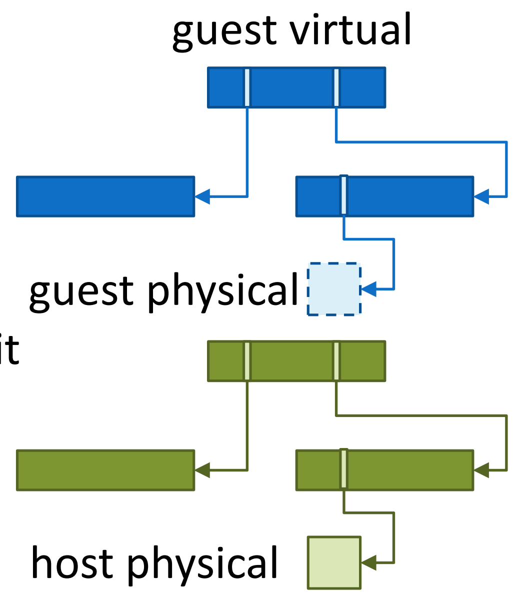
- more bugs (approx. 1 / 1000 LOC)
- less predictable timing
  - complexity
  - high-end CPUs to speed up mostly sequential execution
- but, complex applications are often less timing critical
- open issue: is this still true for vision, …

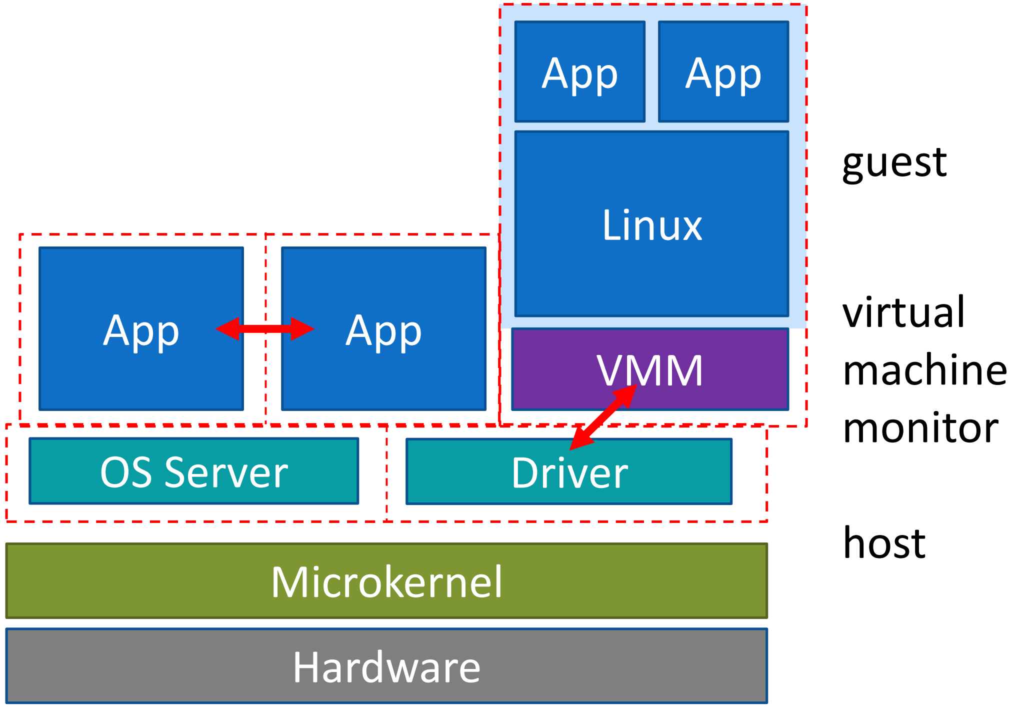


The Case for Virtualization

Virtualization in a nutshell







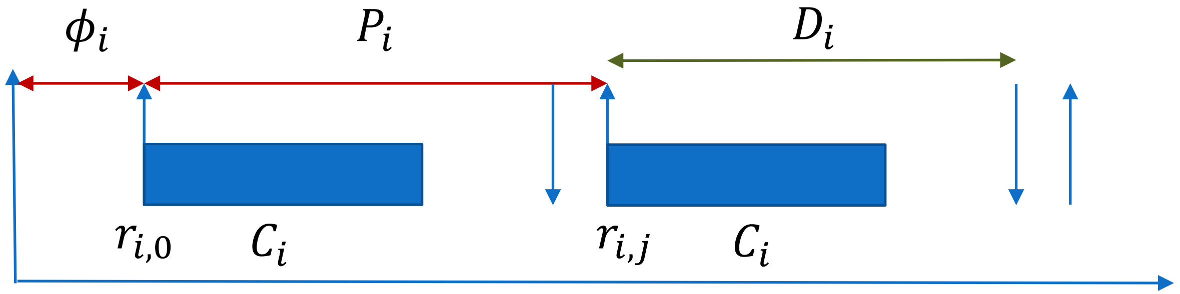
- make sure every interesting event (access to registers of virtualized devices, page faults at transparently shared page, …) exit the VM
- prevent guest from unauthorized access to resources

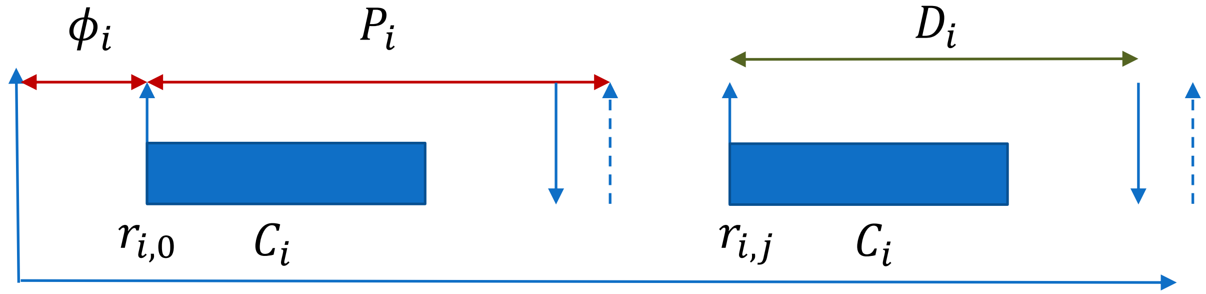
Scheduling Theory 101

Periodic Task Model: $\tau_i = (C_i, P_i, D_i)$

- Task characterized as sequence of jobs (possibly infinite)
- Worst-Case Execution Time (WCET) C_i upper bound on job execution times
- Period P_i subsequent jobs arrive exactly P_i apart
- relative Deadline D_i
  - implicit $D_i = P_i$
  - constrained $D_i \leq P_i$
  - arbitrary (also $D_i > P_i$)
- Phase ϕ_i

- release time $r_{i,j} = jP_i + \phi_i$
- absolute deadline $AD_{i,j} = r_{i,j} + D_i$





Sporadic Task Model: $\tau_i = (C_i, P_i, D_i)$

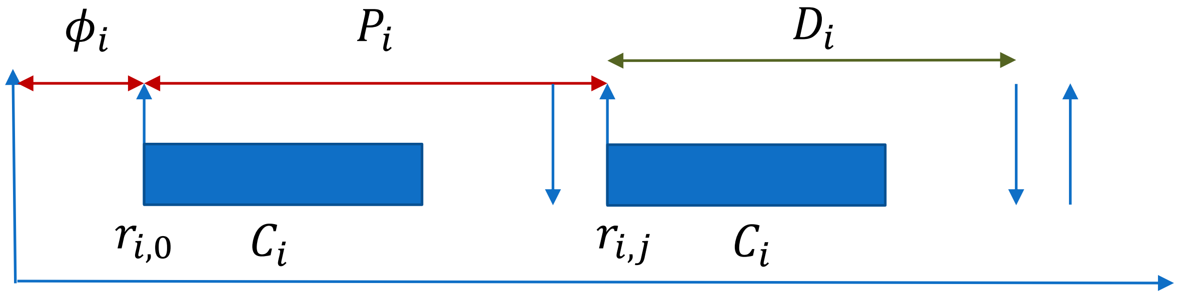
- Like periodic, exept: minimal interrelease time P_i

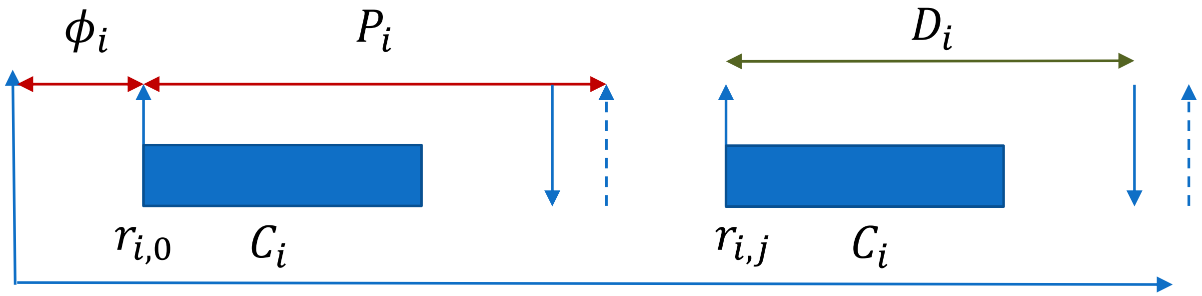
  between the release of any two subsequent jobs, there is at least P_i time.

Scheduling Theory 101

Scheduling

- Scheduling algorithm: decides when to run which job, where (on which resources, typically on which core), and for how long.
- Schedule is feasible for a set of tasks if:
  1) no job $\tau_{i,j}$ executes before $r_{i,j}$
  2) all jobs receive at least C_i time in between $r_{i,j}$ and $r_{i,j}+D_i$
- Scheduling algorithm is optimal (wrt. schedulability) if it finds a feasible schedule whenever there exists one.





(Classic) sporadic task model:
A set of sporadic tasks is schedulable if it is schedulable at the synchronous arrival sequence, that is, when the first job of all tasks arrives at the same time and when subsequent jobs are exactly P_i apart.

!!! Does not hold for mixed-criticality tasks !!!

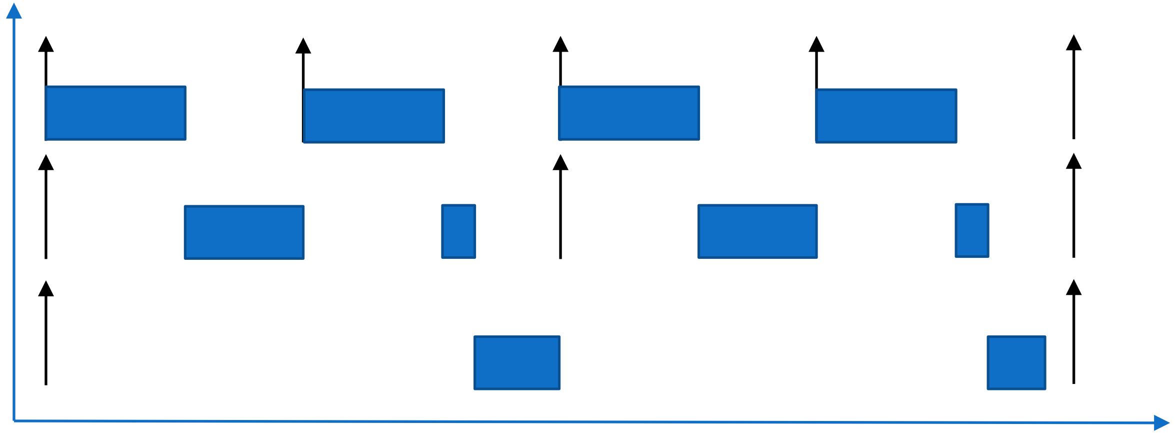
Scheduling Theory 101

Fixed Task Priority (Uniprocessor)

- assign priority to task and use the same priority for all jobs

simultaneous release produces worst schedule
"critical instant"

- e.g.,
  Rate Monotonic Scheduling
  - assign priorities inverse proportional to P_i
  - if phase = 0, $D_i = P_i$ and jobs are independent (must not wait for others):
    - optimal if periods are harmonic (integer multiples)
    - optimal in the class of fixed task priority algorithms



- Liu Layland Criterion:
  n periodic tasks are schedulable with RMS if

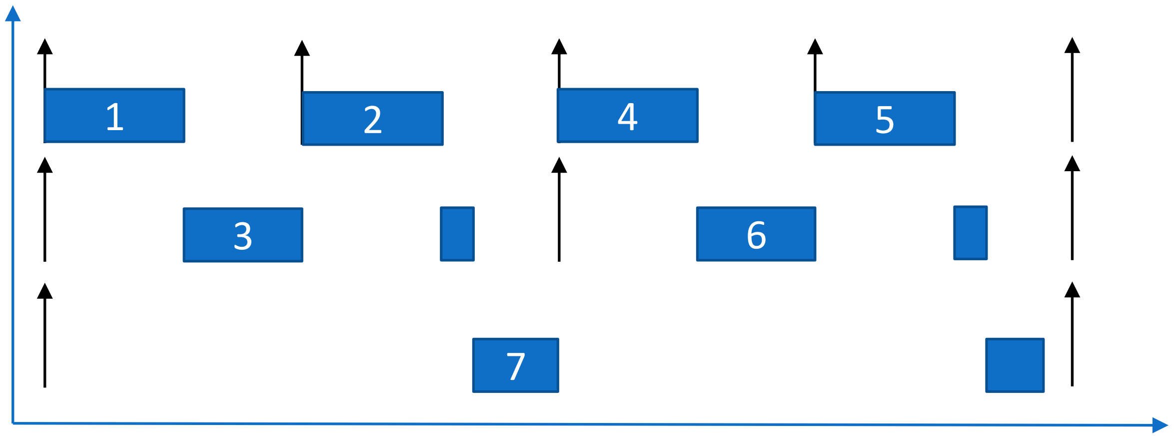
$$\underbrace{\sum_{i=1}^{n} \frac{C_i}{P_i}}_{\text{Utilization}} \leq n \cdot \left(\sqrt[n]{2} - 1\right) \approx \underbrace{0.69}_{n \to \infty}$$

Scheduling Theory 101

Fixed Job Priority (Uniprocessor)

- assign fixed priority to job (once it is released)
- don’t change priorities of running jobs
- e.g.,
  Earliest Deadline First
  - job priority proportional to absolute deadline
  - optimal if jobs are independent, phase = 0

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$$

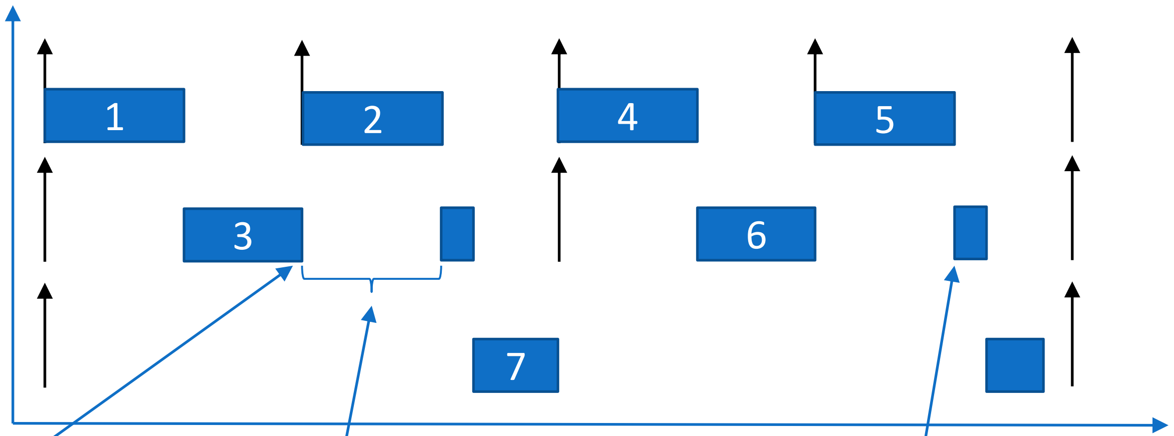


Optimal multiprocessor scheduling algorithms require jobs to change priorities and assigned processors (migrate) while they run.

(trivial extensions of EDF – partitioned / global EDF have utilization bound of $\frac{m+1}{2}$)

Why learn scheduling theory?

- Get an idea when and how your control tasks will be executed?
- Is your system strong enough to schedule all tasks or do you need additional processors?
- Understand choices and tradeoffs when selecting a system:
  - cyclic executive
  - time partitioned system
    (scheduling like cyclic ex. but with isolation)
  - dynamic scheduling in RTOS / RT-Microkernel



2 runs in between here: may happen
between two device register accesses

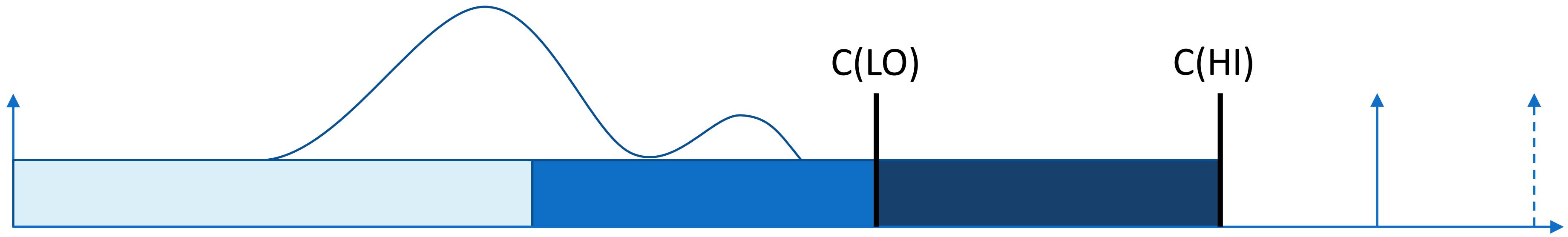
mutex locked by 3 while preempted will be
unavailable to 2 (priority inversion)

preemption leaves task with cold caches
must be considered when determining WCET

Mixed criticality scheduling

Consolidate safety-critical tasks of different importance into a single system

- to share resources (most notably the CPU), even across criticality levels
- to safe costs, weight, energy



- Tasks: $\tau_i = (l_i, \overrightarrow{C_i}, P_i, D_i)$, e.g., $l_i \in \{LO, HI\}$

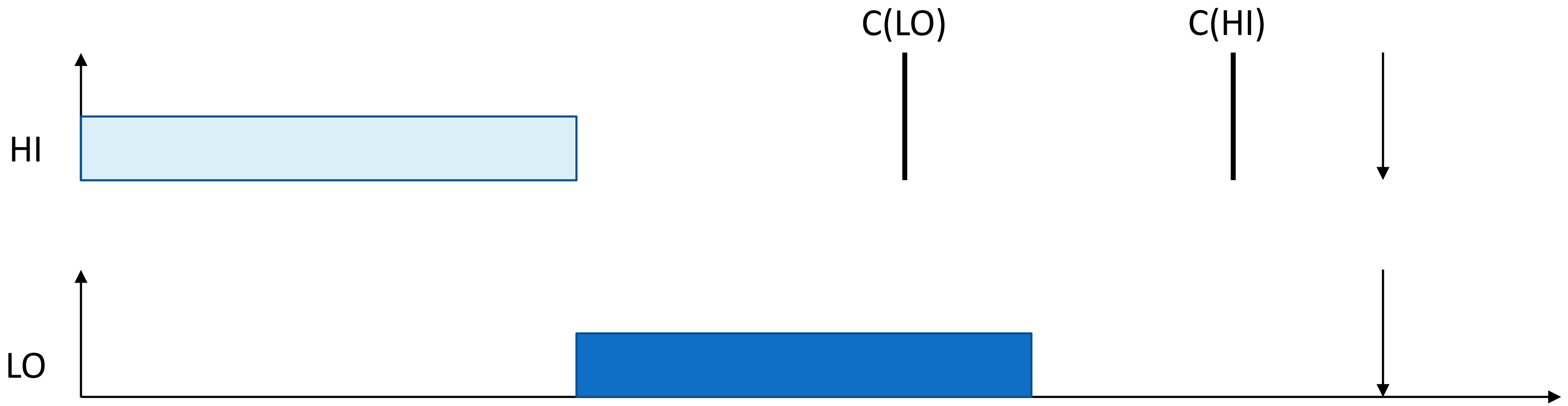
Definition: (MC-feasibility)

A set T of tasks is mixed-criticality feasible if every job $\tau_{i,j}$ receives $\overrightarrow{C_i}(l_i)$ time in between $r_{i,j}$ and $r_{i,j} + D_i$, provided the following rely condition holds: all jobs $\tau_{h,k}$ of higher criticality tasks τ_h(with $l_h > l_i$) complete within $\overrightarrow{C_h}(l_i)$ units time.

Mixed criticality scheduling

Definition: (MC-feasibility)

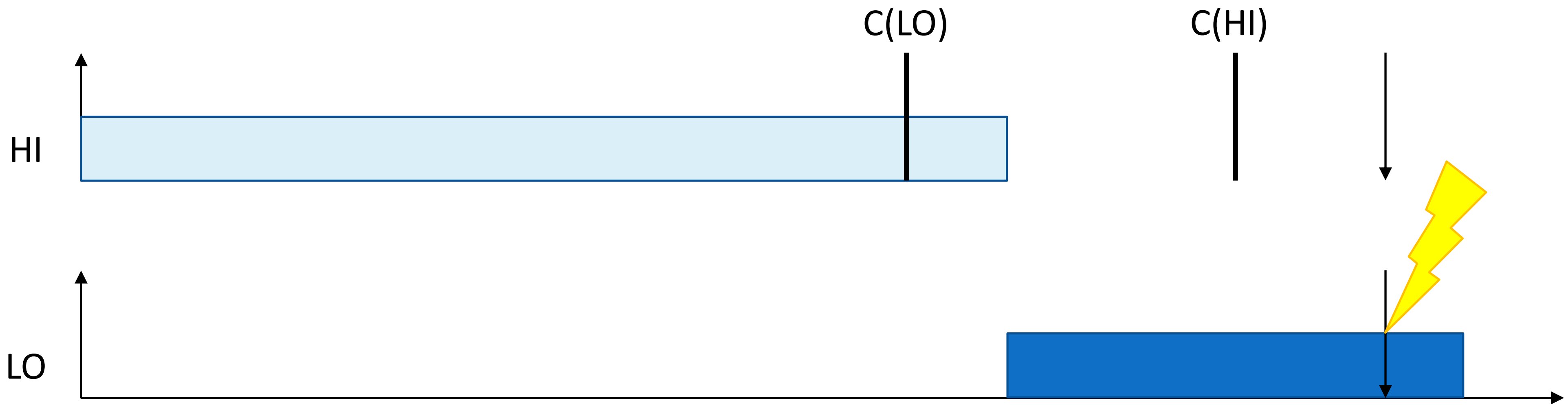
A set T of tasks is mixed-criticality feasible if every job $\tau_{i,j}$ receives $\overrightarrow{C_i}(l_i)$ time in between $r_{i,j}$ and $r_{i,j} + D_i$, provided the following rely condition holds: all jobs $\tau_{h,k}$ of higher criticality tasks τ_h(with $l_h > l_i$) complete within $\overrightarrow{C_h}(l_i)$ units time.



Mixed criticality scheduling

Definition: (MC-feasibility)

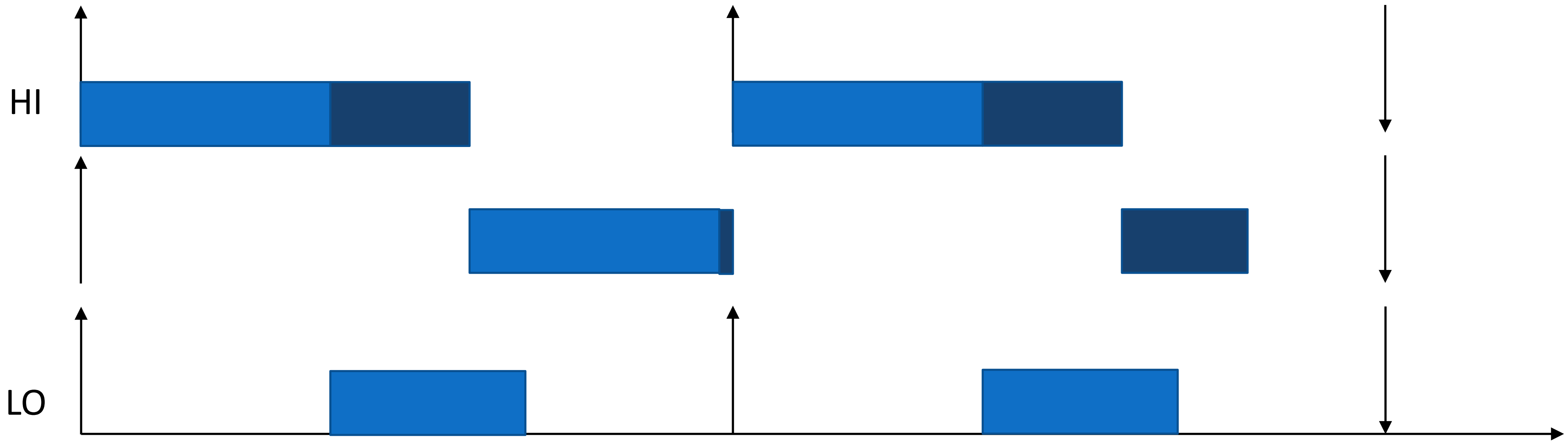
A set T of tasks is mixed-criticality feasible if every job $\tau_{i,j}$ receives $\overrightarrow{C_i}(l_i)$ time in between $r_{i,j}$ and $r_{i,j} + D_i$, provided the following rely condition holds: all jobs $\tau_{h,k}$ of higher criticality tasks τ_h(with $l_h > l_i$) complete within $\overrightarrow{C_h}(l_i)$ units time.



Mixed criticality scheduling

Criticality Monotonic Scheduling

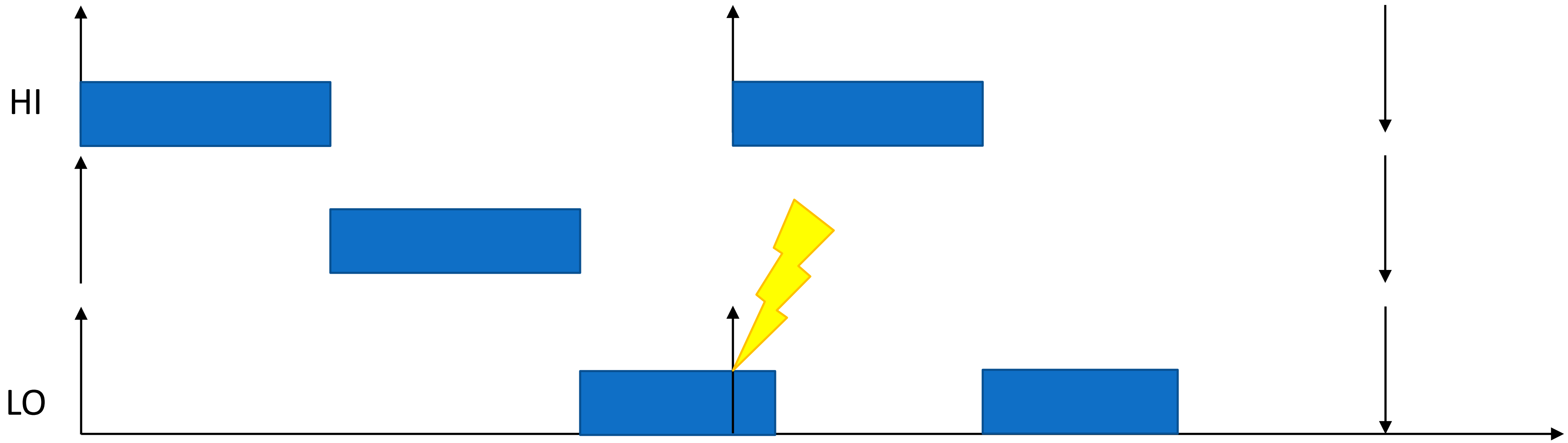
Higher criticality tasks are higher prioritized than all lower criticality tasks. Use classical algorithm within criticality band (e.g., RMS)



Mixed criticality scheduling

Criticality Monotonic Scheduling

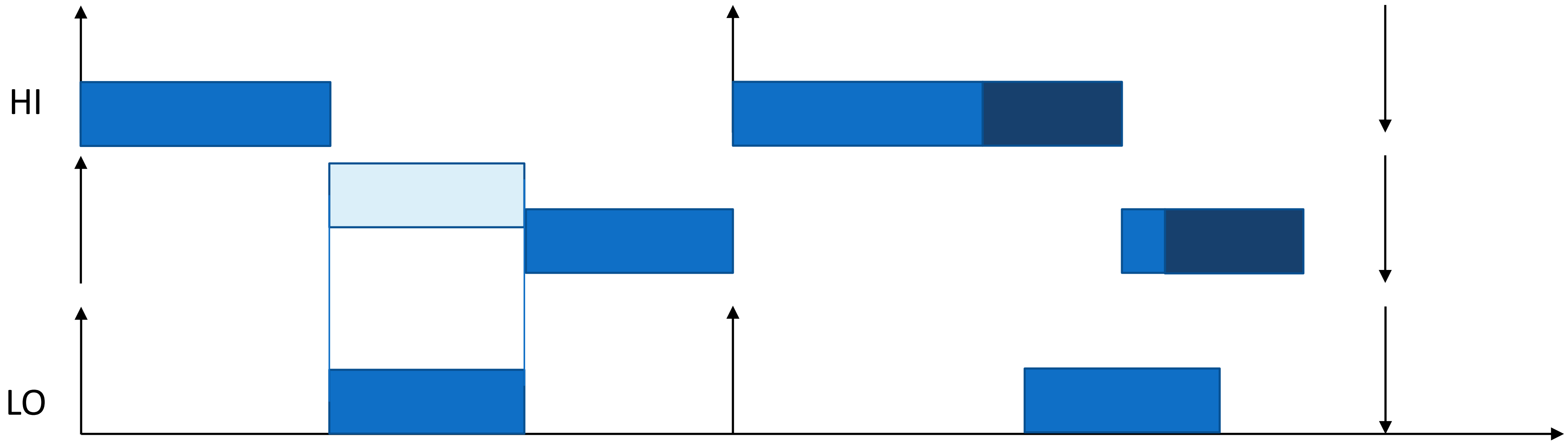
Higher criticality tasks are higher prioritized than all lower criticality tasks. Use classical algorithm within criticality band (e.g., RMS)



Mixed criticality scheduling

Criticality Monotonic Scheduling

Higher criticality tasks are higher prioritized than all lower criticality tasks. Use classical algorithm within criticality band (e.g., RMS)



criticality inversion

lower criticality job executes at higher priority than high criticality job to guarantee its completion in case the rely condition holds.

Examples

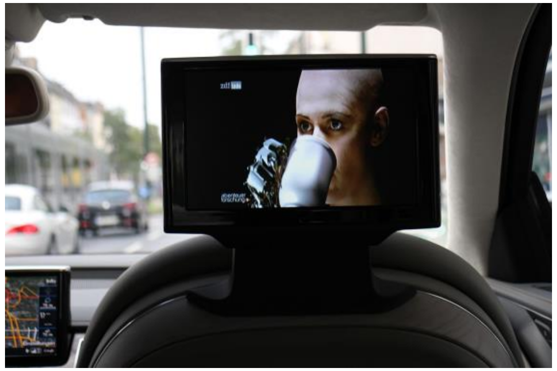
Sacrificing tasks for more critical functionality

Source: NASA

Flight

⬇

Image Processing



Source: Magnus Manske (CC-BY 2.0)

Drive / Break

⬇

Car Entertainment

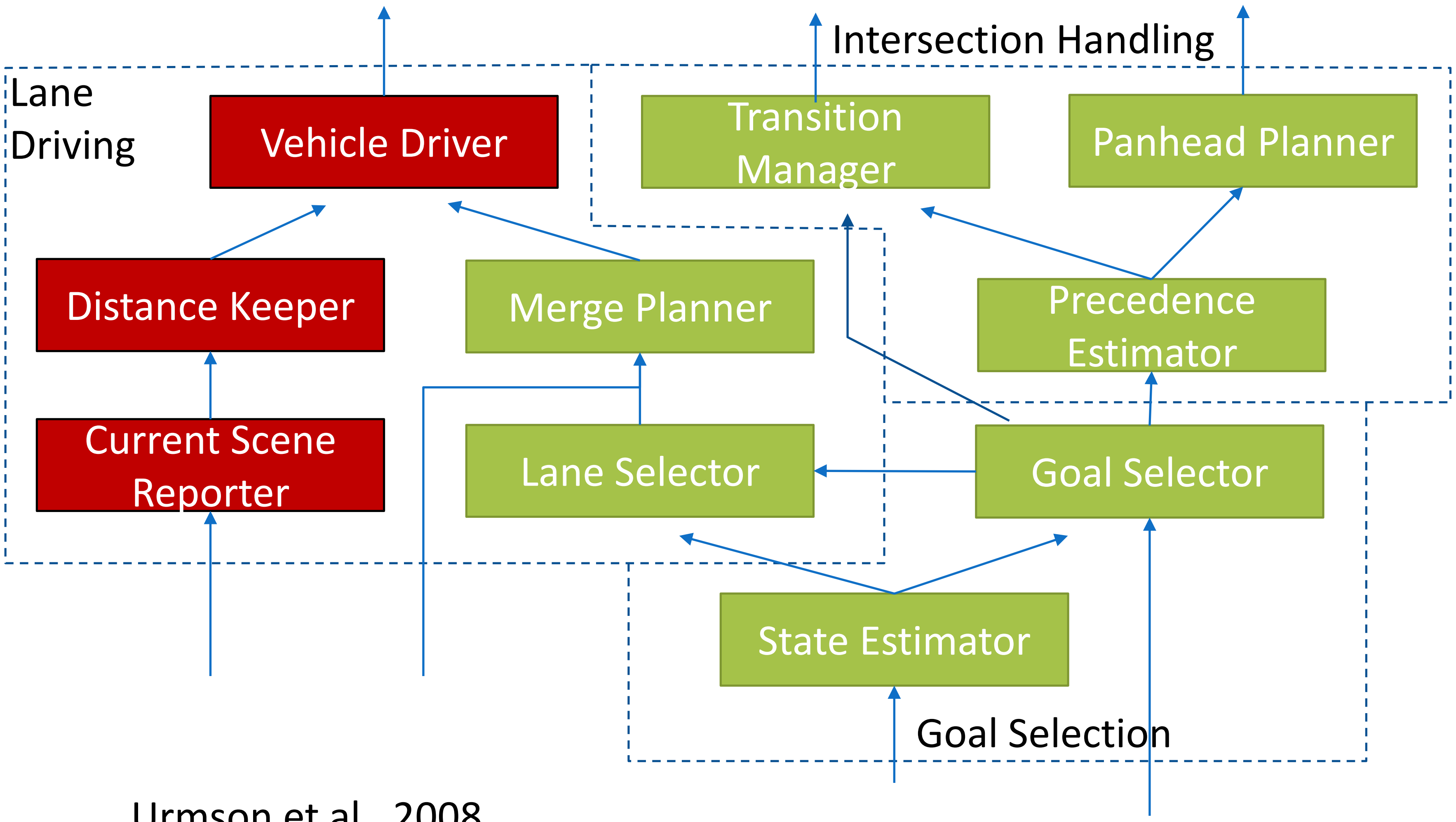
DO 178c Certification

A:	Catastrophic
B:	Hazardous
C:	Major
D:	Minor
E:	No Safety Effect

IEC EN 61508: Safety Certification
Probability for Faults per Hour (continuous operation)

SIL 4: 10^{-8} – 10^{-9} PFH
SIL 3: 10^{-7} – 10^{-8} PFH
SIL 2: 10^{-6} – 10^{-7} PFH
SIL 1: 10^{-5} – 10^{-6} PFH

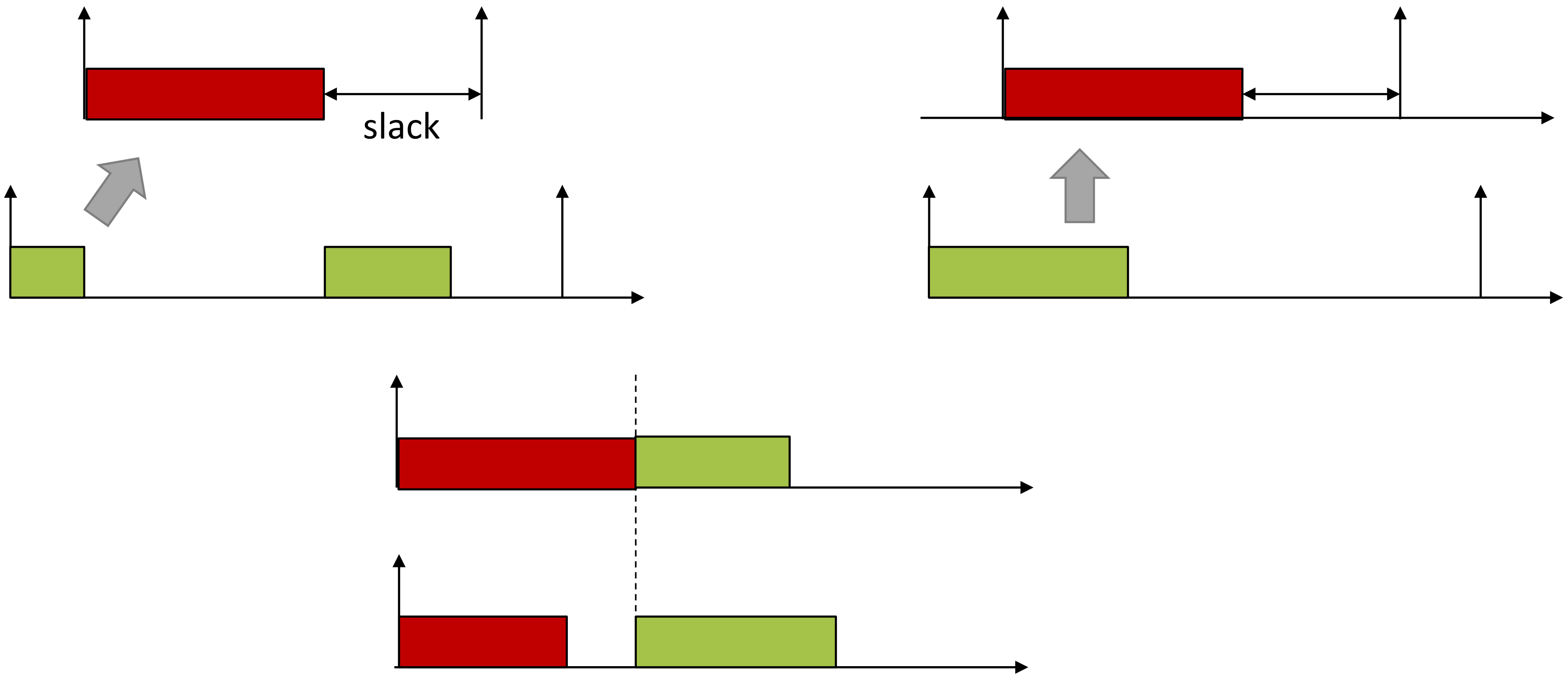
Examples: Autonomous Driving



Urmson et al., 2008,
"Autonomous driving in urban environments: Boss and the Urban Challenge"

Design for Mixed Criticality

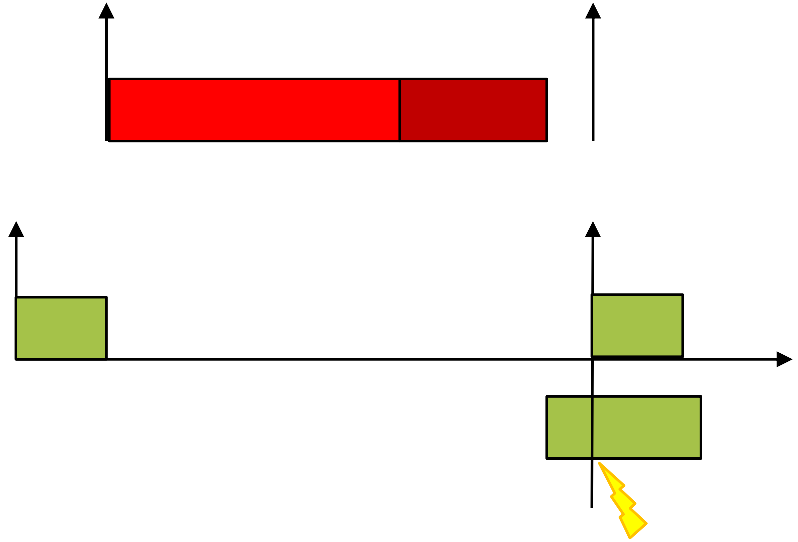
Bounded interference in independent development



- How to limit the interference a low task may cause without having to rely on an analysis of this task?
- How to limit cross core interference in such a way?

Design for Mixed Criticality

Catchup / Restart after dropping



If the new job proceeds with fresh data anyway, prepare to discharge intermediate results of the previous job.

Overview

Cyclic Executives

↓

Real-Time Operating Systems

↓

The Case for Virtualization

↓

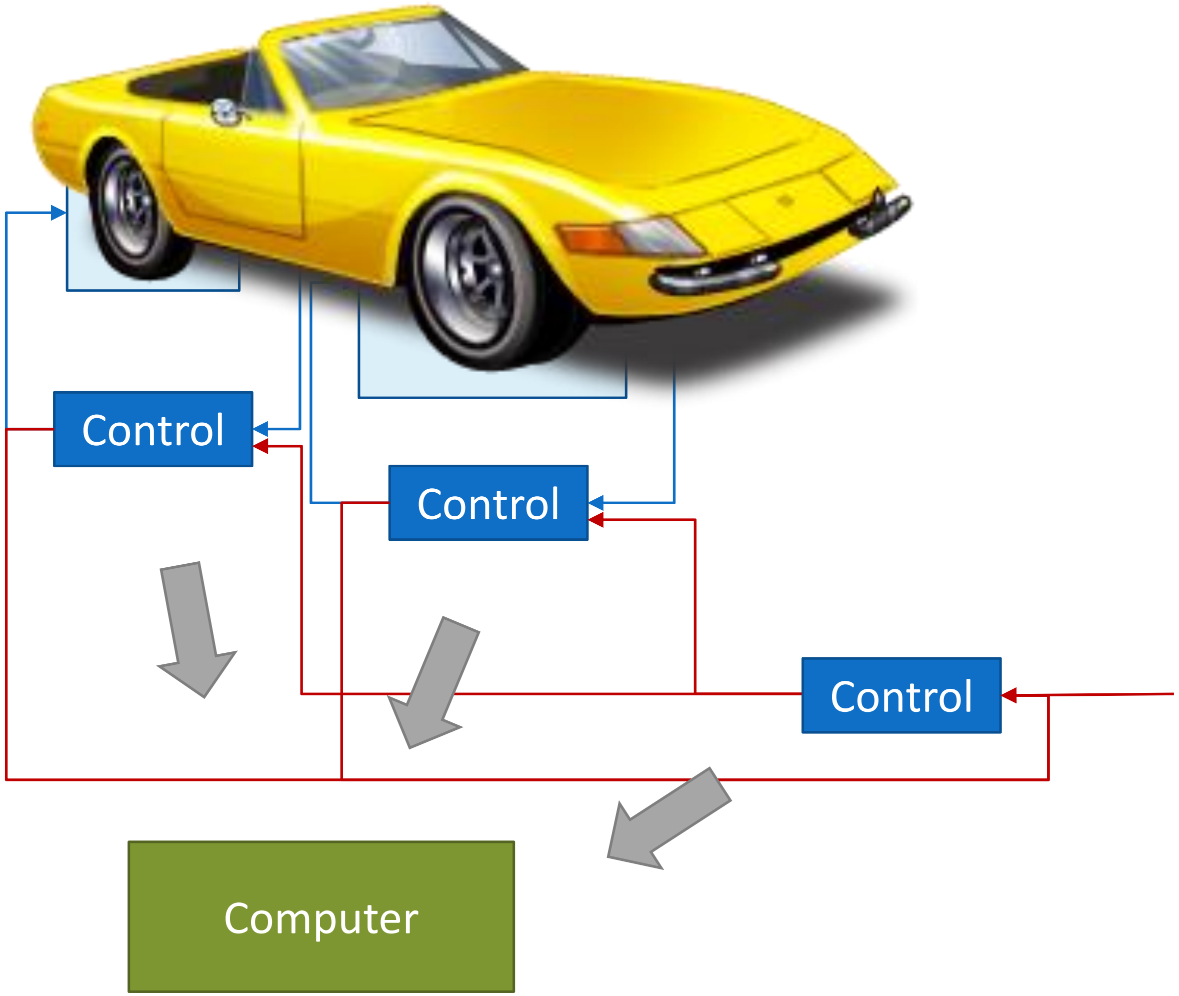
Scheduling Theory 101

↓

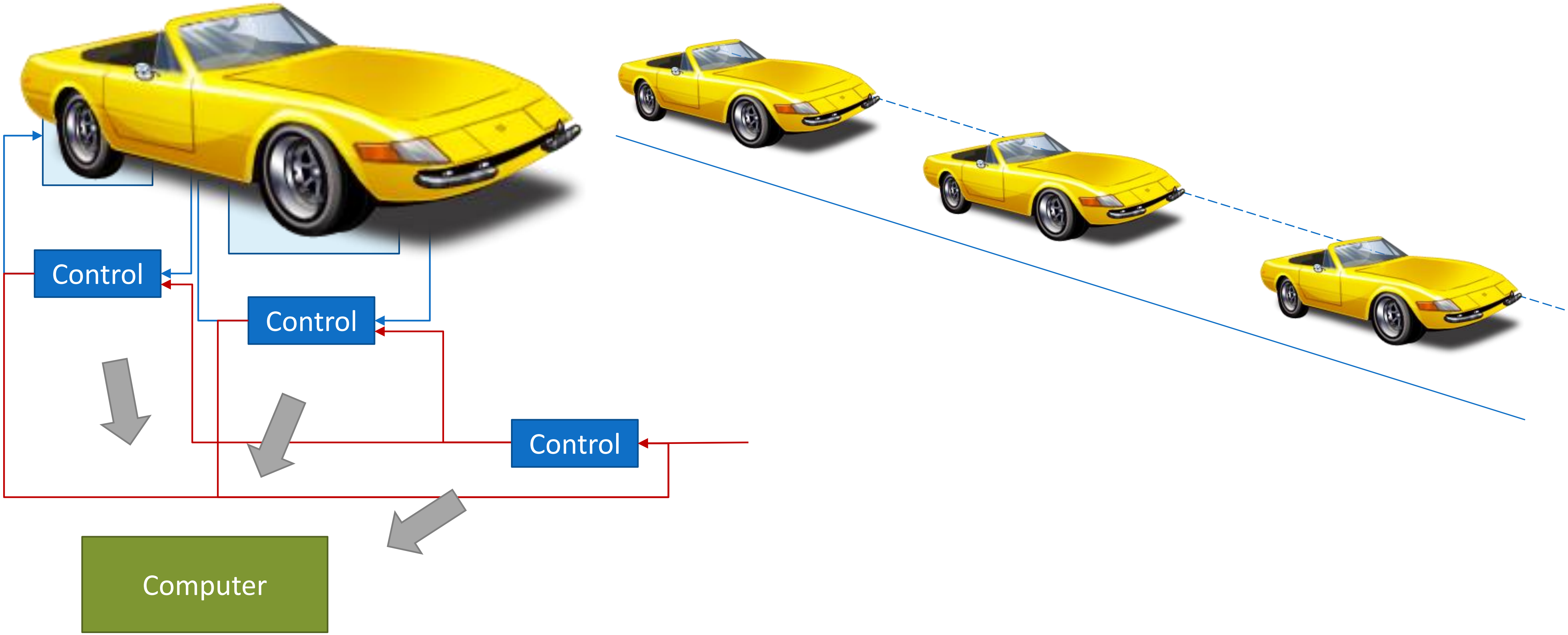
Mixed Criticality

Real-Time Systems
Correctness not only depends on computed values, but also on their timeliness.

The Big Picture



The Big Picture



Overview

Introduction Mathematical Foundations (Differential Equations and Laplace Transformation)	Math
Control and Feedback	Physics
Transfer Functions and State Space Models	
Poles, Zeros / PID Control	Feedback Control
Stability, Root Locust Method, Digital Control	
Mixed-Criticality Scheduling and Real-Time Operating Systems (RTOS)	RTOS
Coordinating Networked Cyber-Physical Systems	CPS
Program Verification Differential Dynamic Logic and KeYmaera X Differential Invariants	Verification