Embedded Operating Systems

- Requirements
- Design
- Implementation

Embedded Systems

- Computer system embedded within larger system
  - Often not perceived as computer system
  - Handles particular task
  - Interacts with the physical world
  - Large variety

- Application:
  - Automotive
  - Avionic
  - Consumer electronics
  - Mobile devices
  - … and many more

Requirements of Embedded Systems

Fundemental differences to desktop (operating) systems!

- **Costs**
  - Hardware (CPU, RAM, Flash, etc)
  - Licenses
  - Software ?

- **Real-time constraints**
  - Correctness of system also depends on timely delivery
  - Soft real-time constraints
    - Slight violations can be tolerated (e.g. multimedia)
  - Hard real-time constraints
    - Violations must be avoided (life can be at risk)
    - Potentially excessive costs (not just monetary)

Requirements of Embedded Systems

- **Safety**
  - "Absence of catastrophic consequences on the user(s) and the environment" (Avizienis et al., 2004)
  - Safety standards and specifications, e.g. IEC 61508 or ISO 26262
- **Robustness** of hardware and software
  - Environmental influences
  - Software failures
- **Security**
  - Protection from unauthorized access, use, modification
  - Internal / external attackers
  - Attack surface:
    - Direct connection (e.g., debug ports)
    - Short- & Long-range wireless communication
  - System lifetime

How to Fullfill These Requirements?

- **Costs**
  - Minimize per-piece costs
    - Low cost components
    - Microcontrollers (8, 16, 32bit)
    - Low amount of memory:
      1MB Flash / 128KB RAM are "high-end"
  - Development costs are a minor issue

- **Real-time constraints**
  - Provide guarantees on worst-case execution time (WCET)
    - Static analysis, Simulation, Dynamic analysis
  - Eliminate non-determinism
    - Static scheduling
    - Static memory layout and allocation

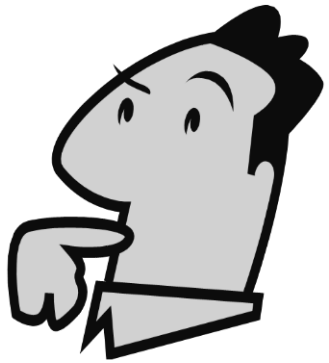
How to Fullfill These Requirements?

- **Robustness**
  - "Ability of a system to handle errors during execution"
  - Redundancy
  - Fault tolerance

- **Safety & Security**
  - Rigorous software design processes
  - Following established standards / best practices
    - Static analysis
    - Dynamic testing
    - Fault injection
    - Run-time monitoring
    - Code reviews
    - Penetration testing
    - … and more

Implications on Embedded OS

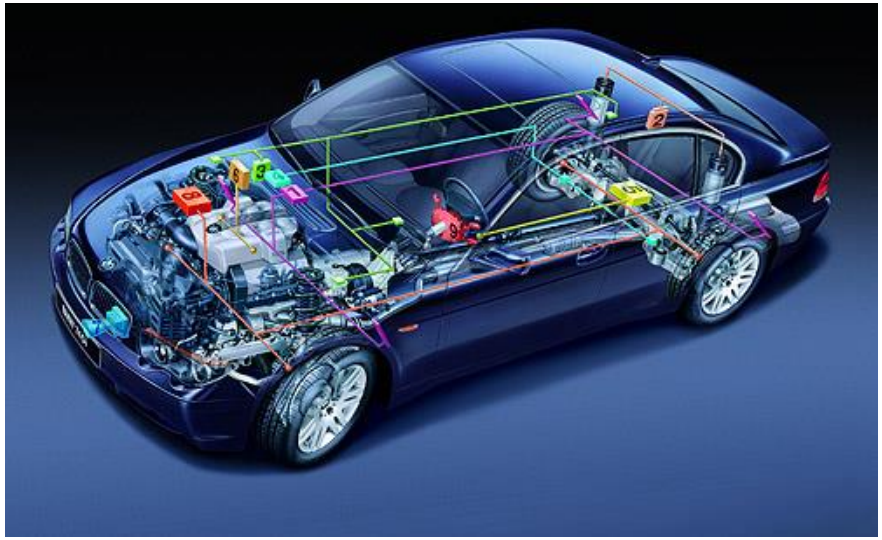
- OS still is resource allocator and control program
- Key differences
  - Resources are (highly) constrained
  - No multi-user
  - Kernel-/user-mode less common
- What about
  - Process management
  - Memory management
  - I/O
  - Scheduling
  - Overall architecture



Example: AUTOSAR OS

Modern automobile

- Up to 80 electronic control units
  - Engine
  - Transmission
  - Brakes
  - Airbag
  - Infotainment
- Distributed system
  - CAN, FlexRay, LIN, MOST
- Safety-critical
- Increasingly complex



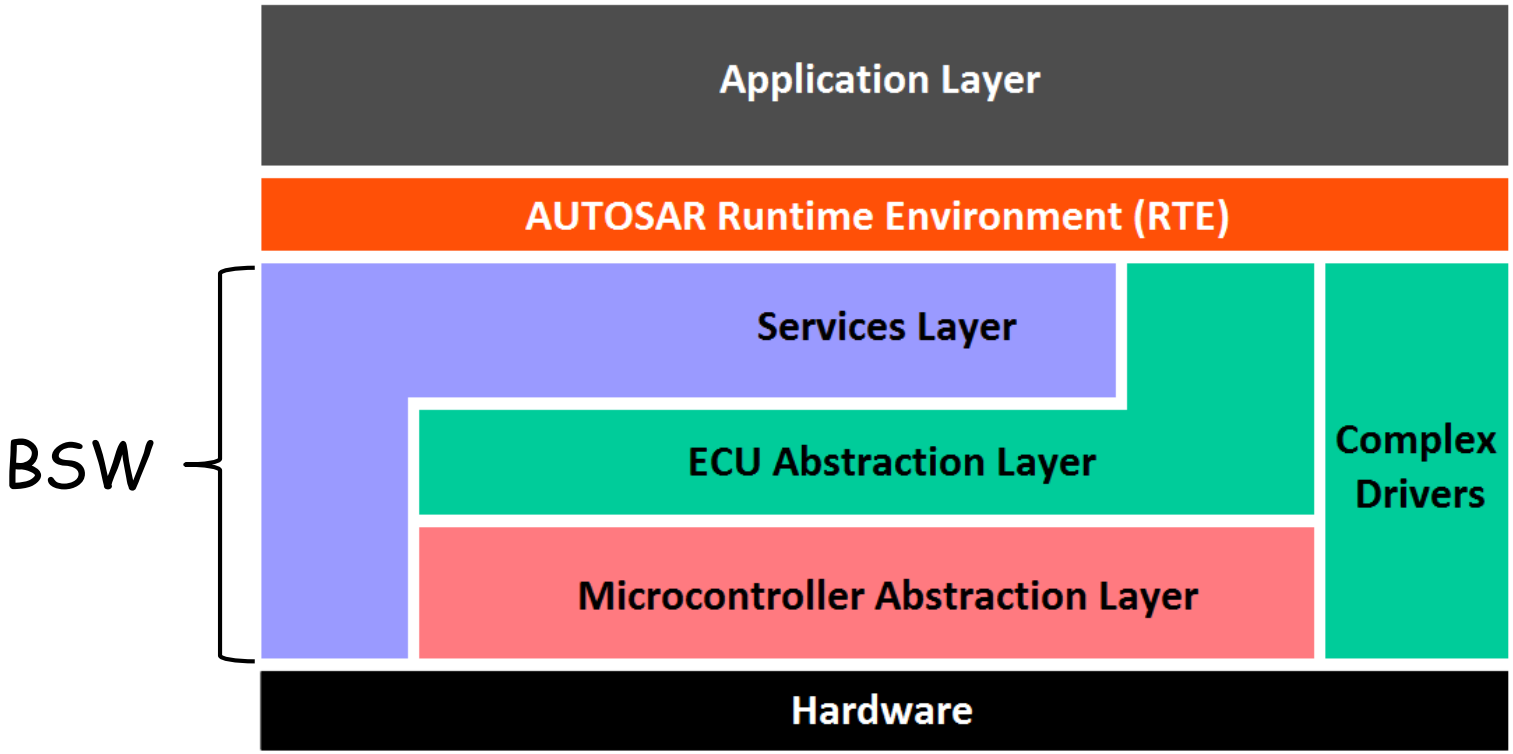
AUTOSAR standard

- Open automotive system architecture
- Partners: BMW, Daimler, Ford, GM, VW and many more
- "Cooperate on Standards, Compete on Implementations"

AUTOSAR Architecture

Layered architecture

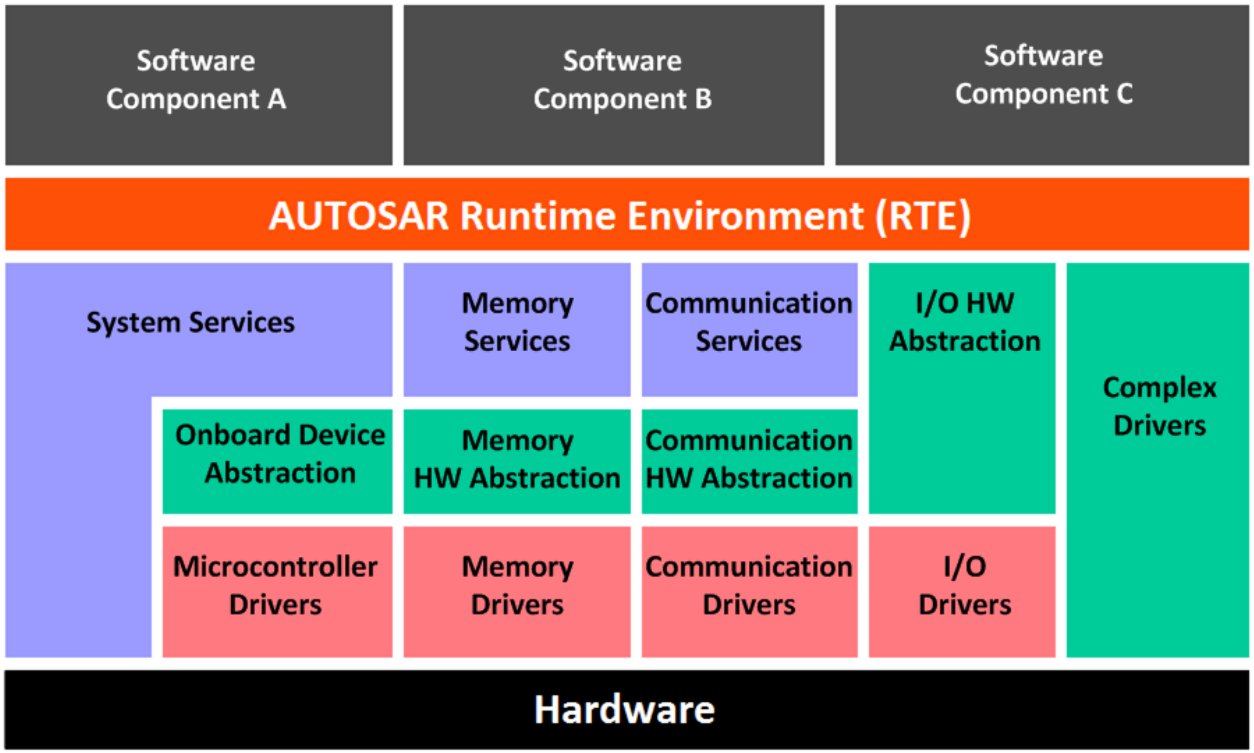
- Application: Software components implementing runnables
- RTE: Communication and comm. abstraction services
- BSW: Operating system, services and hardware abstraction



AUTOSAR Architecture

Component-based architecture

- Application Software Components (SW-Cs)
- SW-Cs contain **runnables** as functional blocks
- Runnables are grouped to **tasks**
- Tasks are scheduled by the OS



AUTOSAR OS

- Based on the widely used OSEK/VDX

- Event triggered OS
  - Timers
  - Signals (e.g., message arrival)

  → Both events drive task scheduling

- Basic features
  - Configured and scaled statically
  - Fixed, priority-based scheduling
  - Hostable on low end controllers
  - No external resources

→ Goal: Achieve **determinism** & **predictability** !

AUTOSAR OS

- Process Management
  - No programs or processes → instead: tasks
  - Tasks contain runnables (functional entities)
  - Task types: non-preemptive or preemptive
  - Static and "monolithic" system (but modular)

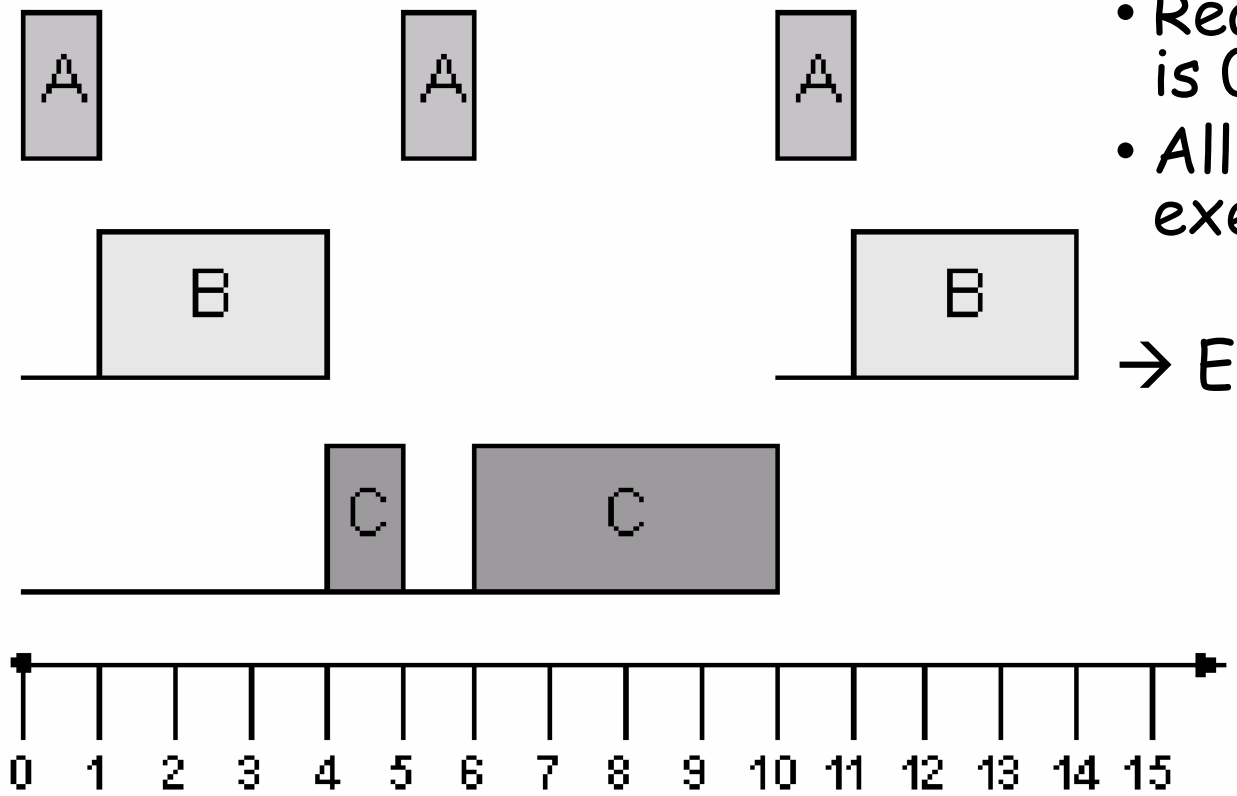
- Memory Management
  - Static memory layout
  - Eliminates non-determinism
  - Functions such as malloc() not implemented

- Scheduling
  - Statically configured via timers / alarms
  - Worst-case execution time analysis / estimation
  - Different scheduling planning algorithms

Problem: OS does not provide schedulability analysis!

Timing Violation Example

Task ID	Priority	Worst-case Execution Time	Deadline / Period
A	High	1	5
B	Medium	3	10
C	Low	5	15

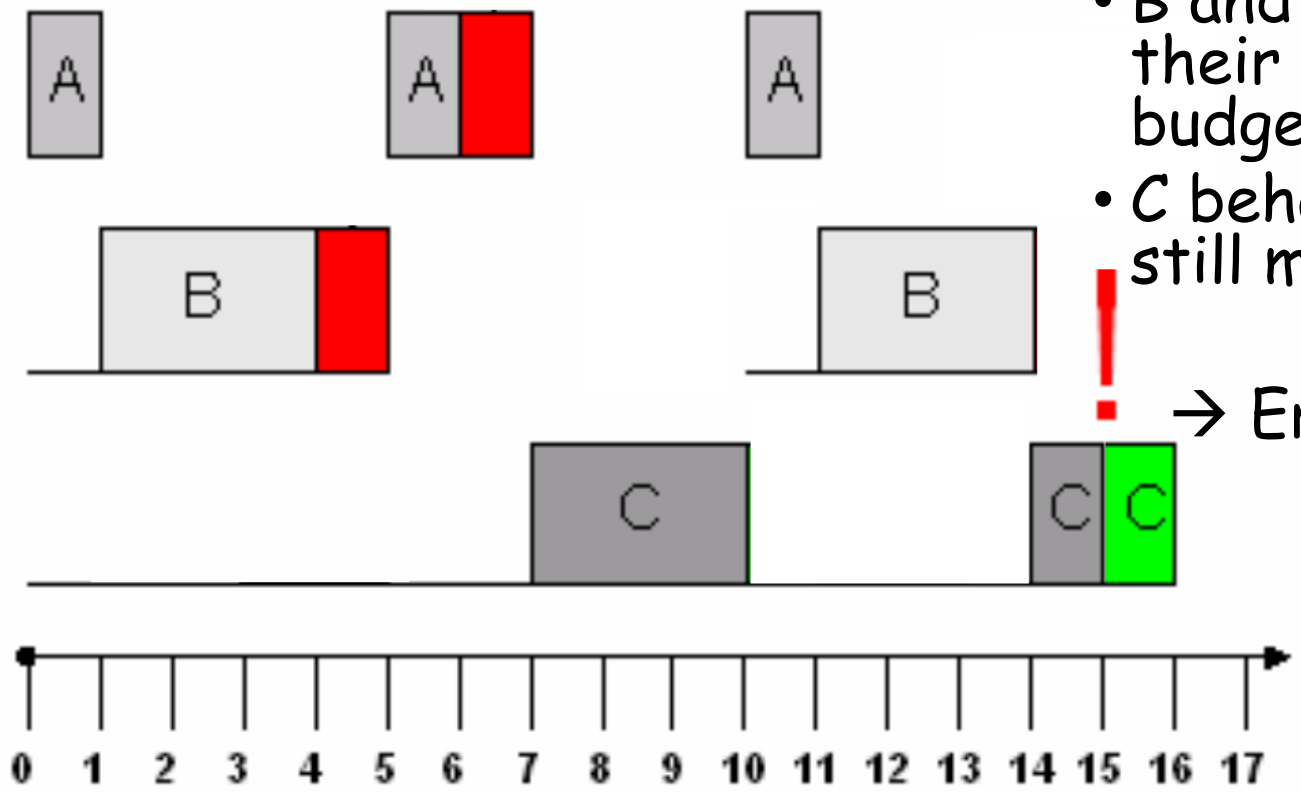


- Ready time of A, B, C is 0
- All tasks keep their execution time limits

→ Everything fine here

Timing Violation Example

Task ID	Priority	Worst-case Execution Time	Deadline / Period
A	High	1	5
B	Medium	3	10
C	Low	5	15



- B and A both violate their execution time budget
- C behaves correctly, still misses deadline

! → Error propagation

Real Time and Scheduling Analysis

Goal: Ensure that all tasks meet their deadlines

- Testing is inappropriate
  - Only verifies if deadlines are usually met, not always
  - Constructing the worst case is often difficult

→ Formulate assumptions that constrain & ease the problem

- Ensure testing for the worst case gets easier
  - Static scheduling

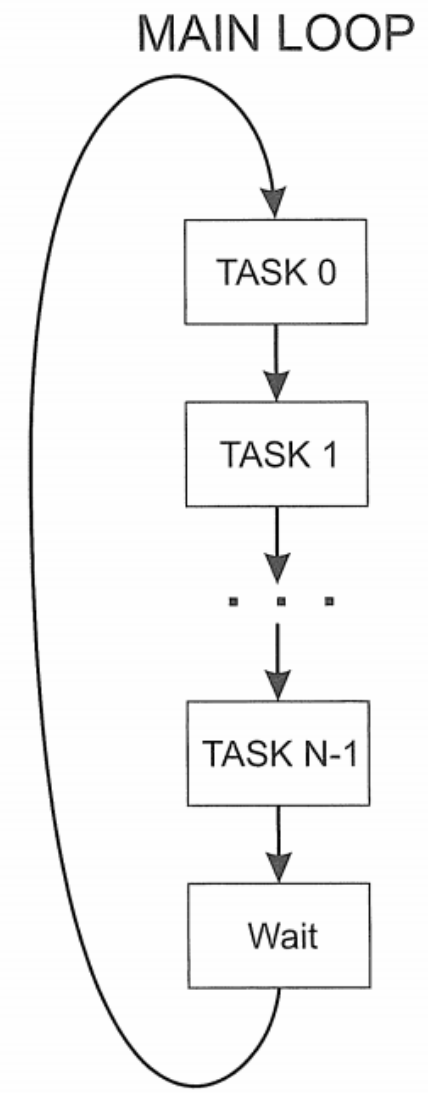
- Ensure worst case is mathematically understood
  - Rate monotonic scheduling

Real Time and Scheduling Analysis: Assumptions

- All tasks $\{T_0, T_1, \ldots, T_{N-1}\}$ are perfectly periodic
- All taks are completely independent
  - In practice difficult to realize
  - Shared resources, variables, multiplexed A/D, etc…
- Worst case execution time (WCET) is known
  - Longest possible path through branches
  - Worst combination of cache misses
  - Computation / analysis tools (e.g. AbsInt)
  - Measurement
  - Add margins
- Zero time task switching
  - Add to task execution time
  - Leave some % of CPU capacity as margin

Pure Static Scheduling (non-preemptive)

- All tasks run in a big loop
- No interrupts, no preemption
- Pros:
  - Simplicity (no scheduling SW, no OS interaction needed)
  - Easy to test / validate
  - Deterministic
- Cons:
  - Unflexible
  - No priorization
  - I/O polling



© P. Koopman – Better Embedded Software, 2010

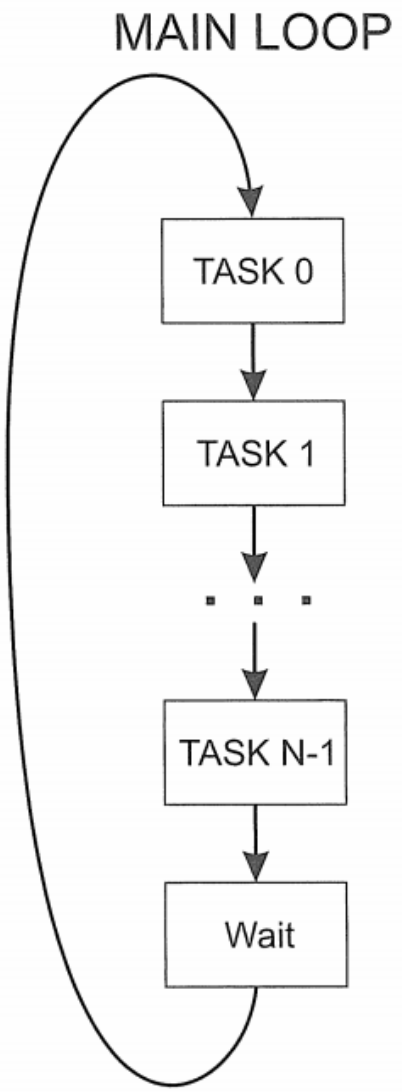
Pure Static Scheduling (non-preemptive)

Scheduling analysis terminology

- N Number of tasks
- C_i Computation time of task T_i (worst case!)
- D_i Deadline of task T_i
- P_{main} Period of main loop

$$\sum_{i=0}^{N-1} C_i \leq \min_{i=0}^{N-1}(D_i)$$

$$\sum_{i=0}^{N-1} C_i \leq P_{main} \leq \min_{i=0}^{N-1}(D_i)$$



- Improvements / Variations
  - Multiple runs of a task (ensure spacing!)
  - Run one background interrupt

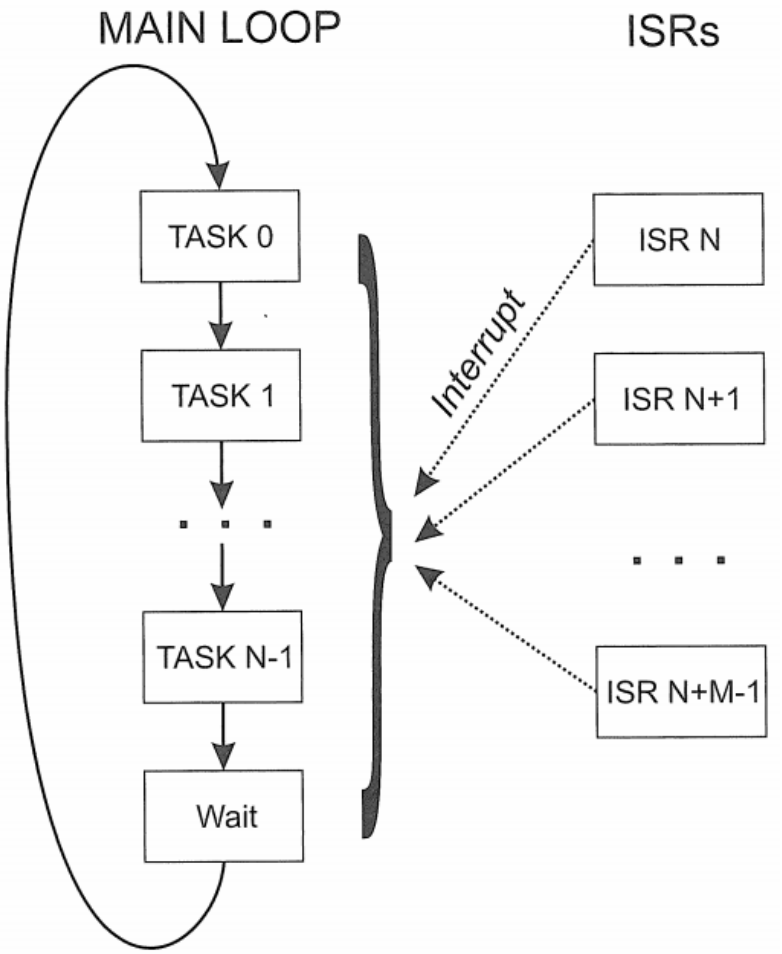
Static Scheduling with Small Helper Interrupts

- Helper interrupts
  - Strict limitation of functionality
  - I/O servicing (read/write device)
  - Service counter/timers
  - Small task with very fast deadline

→ Ensure ISRs are fast w/o data sharing problems

- Pros:
  - Interrupt usage
  - No polled I/O
  - Tight deadlines handled in ISR

- Cons:
  - ISR functionality limited to few dozen instructions



© P. Koopman – Better Embedded Software, 2010

Static Scheduling with Small Helper Interrupts

Scheduling analysis – Easy case

Assumptions:

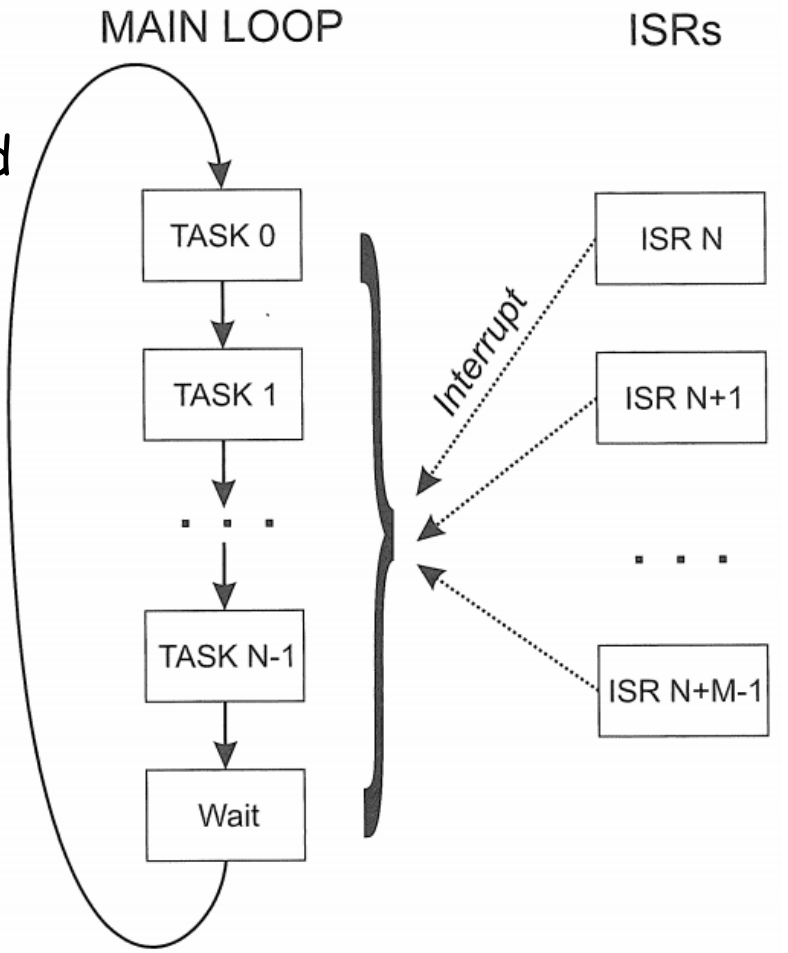
- Every ISR activates once per period
- ISR deadline not shorter than mainloop

→ Limited practical use – helps understanding difficult case!

M Number of ISRs

$$\sum_{i=0}^{N+M-1} C_i \leq P_{main} \leq \min_{i=0}^{N+M-1}(D_i)$$

→ ISRs treated as task!



© P. Koopman – Better Embedded Software, 2010

Static Scheduling with Small Helper Interrupts

Scheduling analysis – Difficult case

Assumption:

- ISR may execute more than once per main loop:

$$\rightarrow D_{ISR} \leq P_{main}$$

Problems:

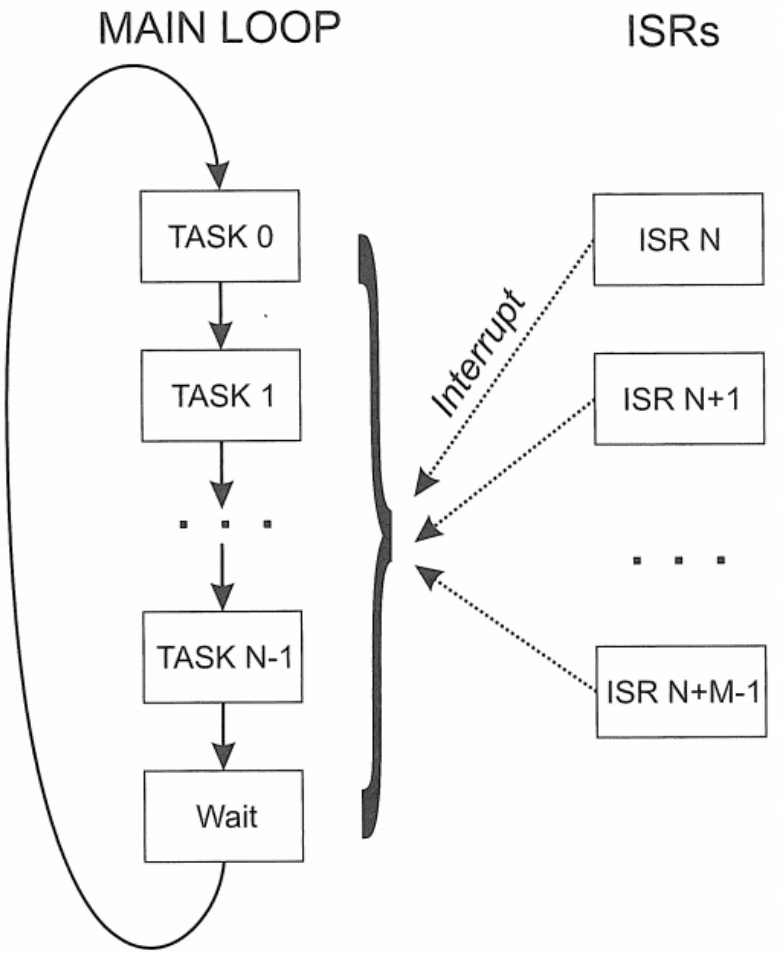
- Starvation
- Priority inversion (ISR)

Solution:

- Each ISR can execute only once within smallest ISR deadline

$$\sum_{i=N}^{N+M-1} C_i \leq \min_{i=N}^{N+M-1}(D_i)$$

Limitation: No long running ISRs!



© P. Koopman – Better Embedded Software, 2010

Static Scheduling with Small Helper Interrupts

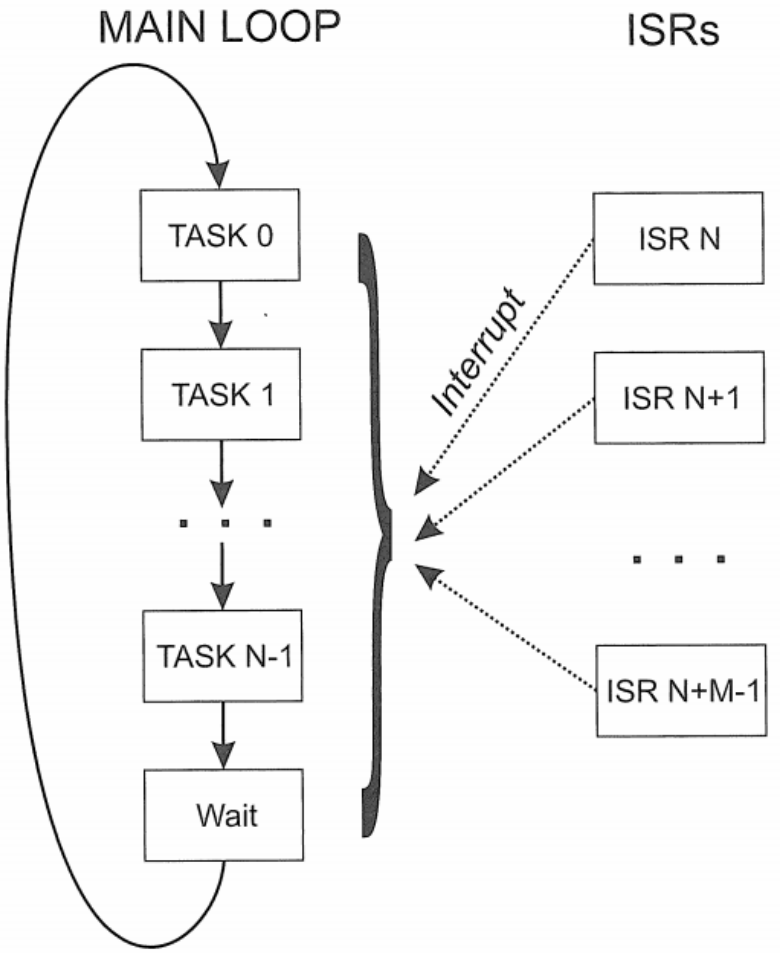
Scheduling analysis – Difficult case

Maximum computation time of all interrupts in P_{main}:

$$C_{ISR\_Total} \leq \sum_{i=N}^{i=N+M-1} \left(\left\lceil \frac{P_{main}}{P_i} \right\rceil \cdot C_i \right)$$

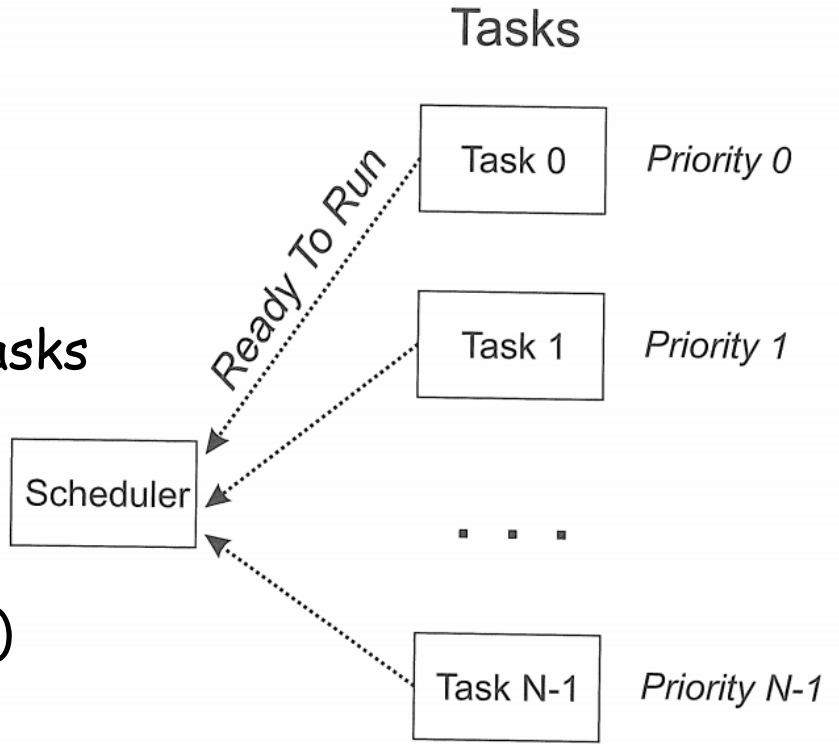
Schedulability condition:

$$\sum_{i=0}^{N-1} C_i + C_{ISR\_Total} \leq P_{main}$$



Rate Monotonic Scheduling

- Properties
  - Most flexible/complex
  - Supports task preemption
  - Scheduler instead of main loop
  - Priorities statically assigned
- Pros:
  - Supports tasks with long periods
  - Mix non-realtime with realtime tasks
  - Simple scheduling system
- Cons:
  - Requires a scheduler (with RTOS)
  - Assumes no interrupts
  - Assumes zero latency preemption



© P. Koopman – Better Embedded Software, 2010

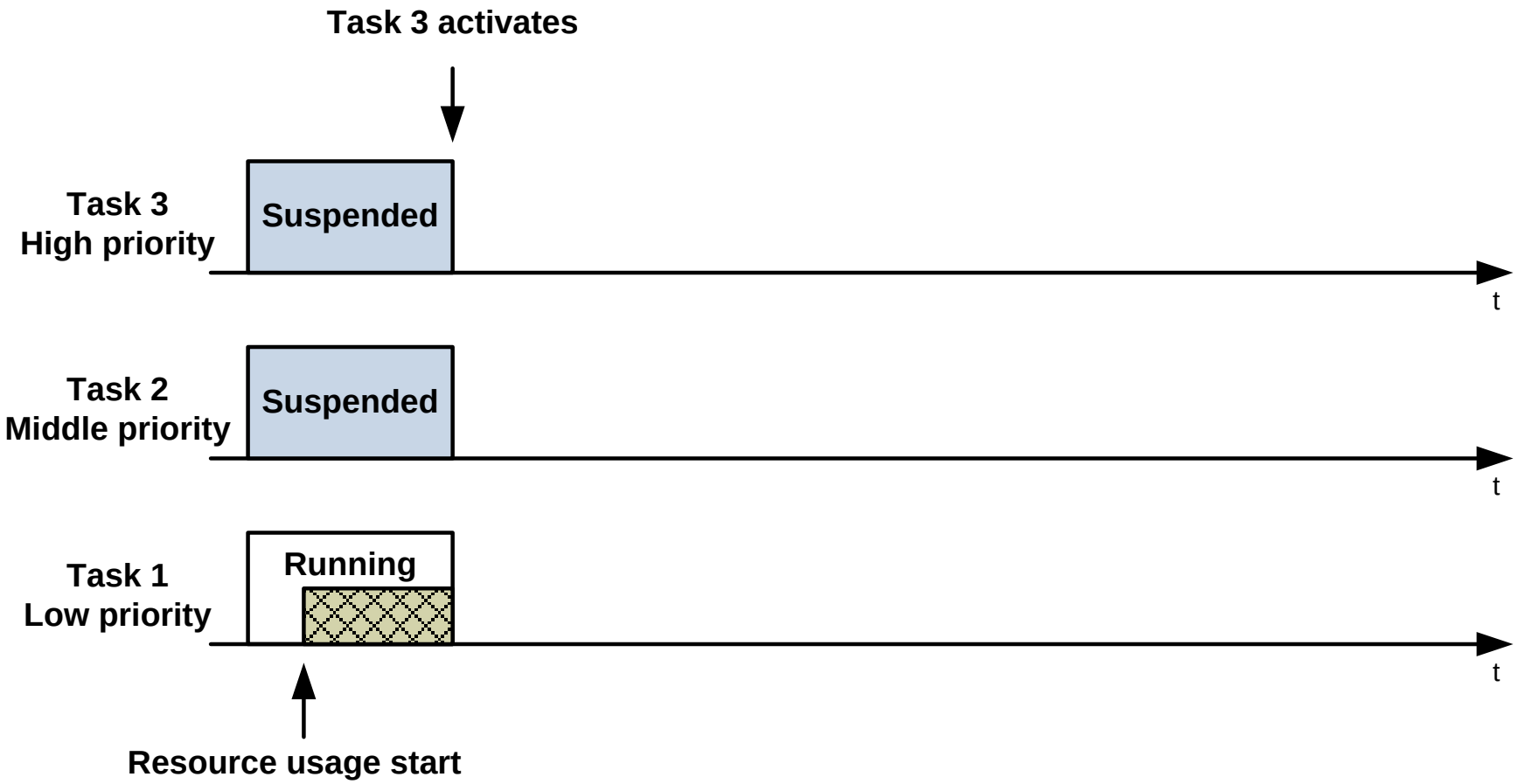
Rate Monotonic Scheduling

Scheduling analysis

- Complex!
  - "Scheduling algorithms for multiprogramming in a hard real-time environment", C. L. Liu and J. Layland, Journal of the ACM 20 (1): 46–61, 1973.

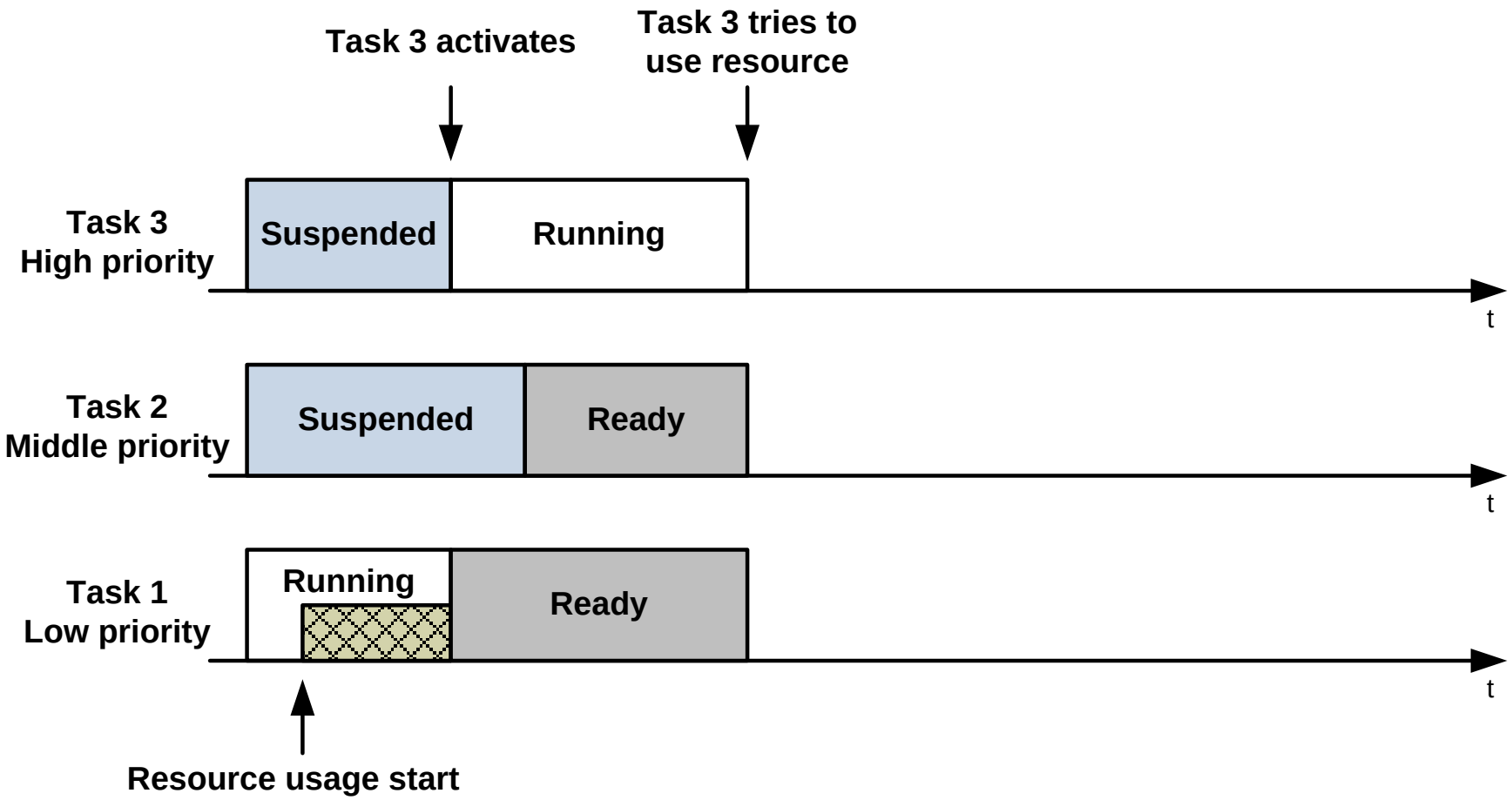
    doi:10.1145/321738.321743 (over 9500 citations)
- Special case:
  - Schedulability for 100% CPU usage only guaranteed for harmonic task periods
  - Example: {2ms, 10ms, 20ms} vs. {2ms, 10ms, 25ms}
- Pitfalls:
  - No interaction between tasks
  - Prone to priority inversion!!

Priority Inversion Problem



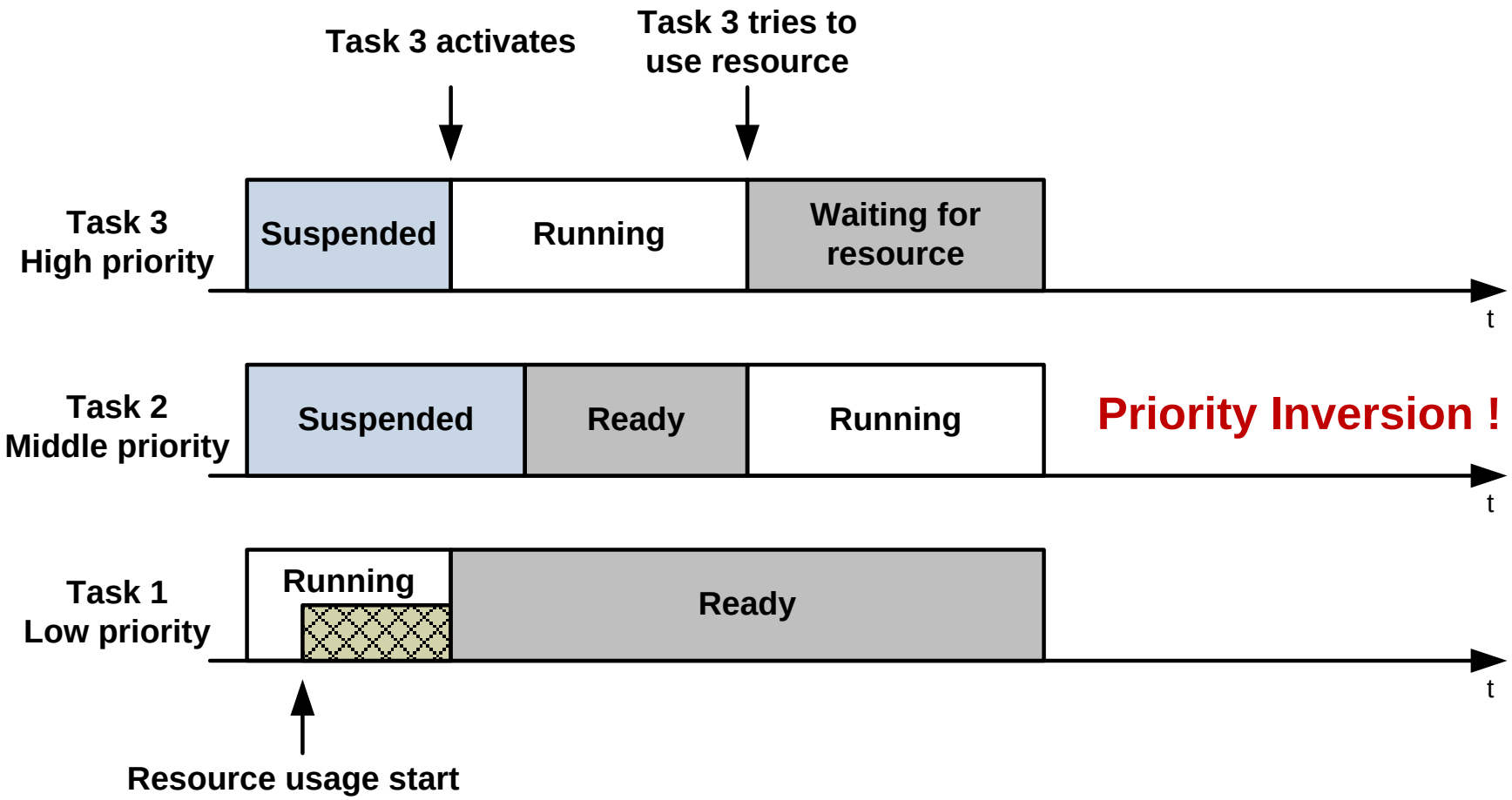
© W. Zimmermann, et al. – Bussysteme in der Fahrzeugtechnik

Priority Inversion Problem



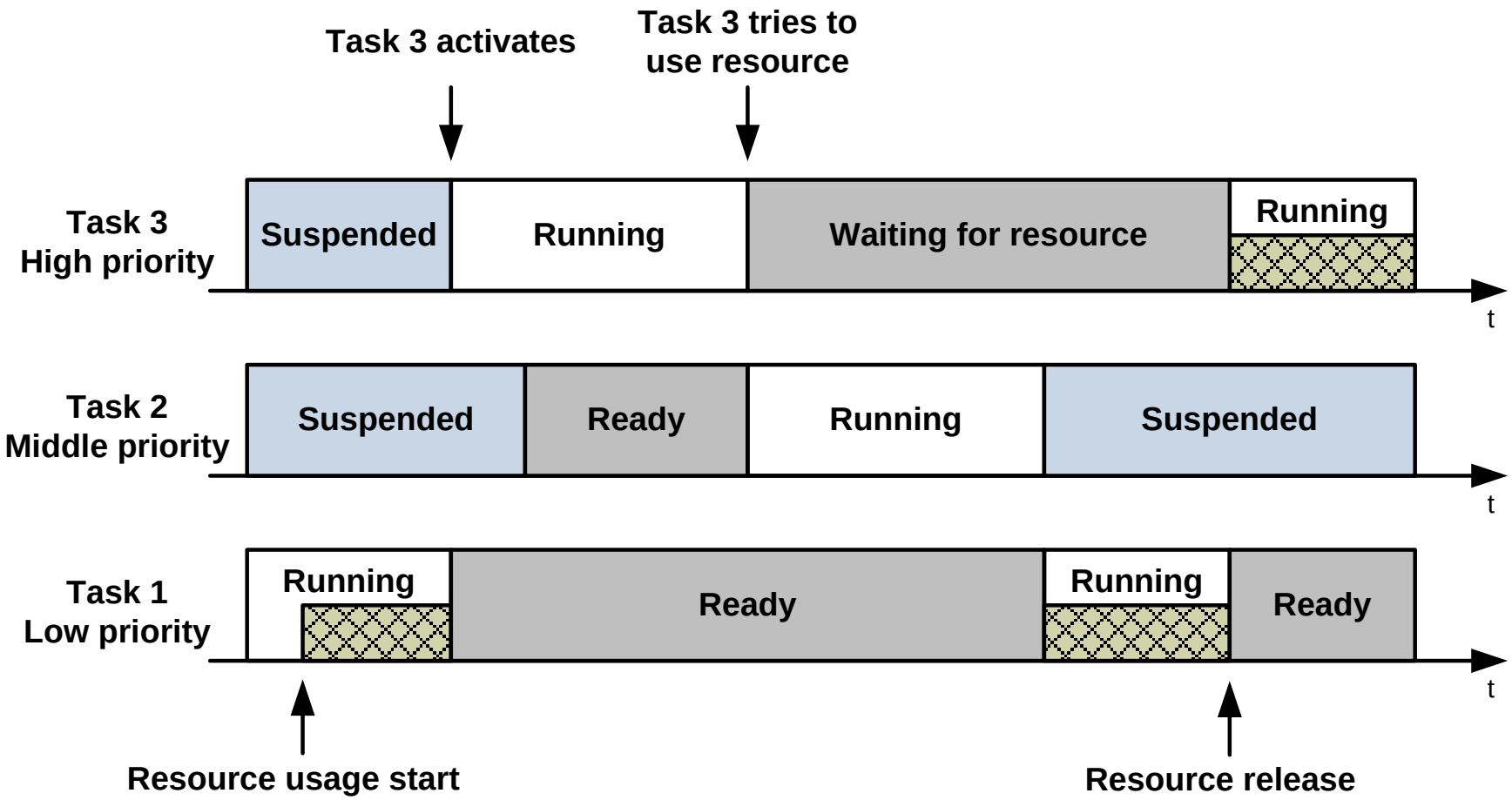
© W. Zimmermann, et al. – Bussysteme in der Fahrzeugtechnik

Priority Inversion Problem



© W. Zimmermann, et al. – Bussysteme in der Fahrzeugtechnik

Priority Inversion Problem



© W. Zimmermann, et al. – Bussysteme in der Fahrzeugtechnik

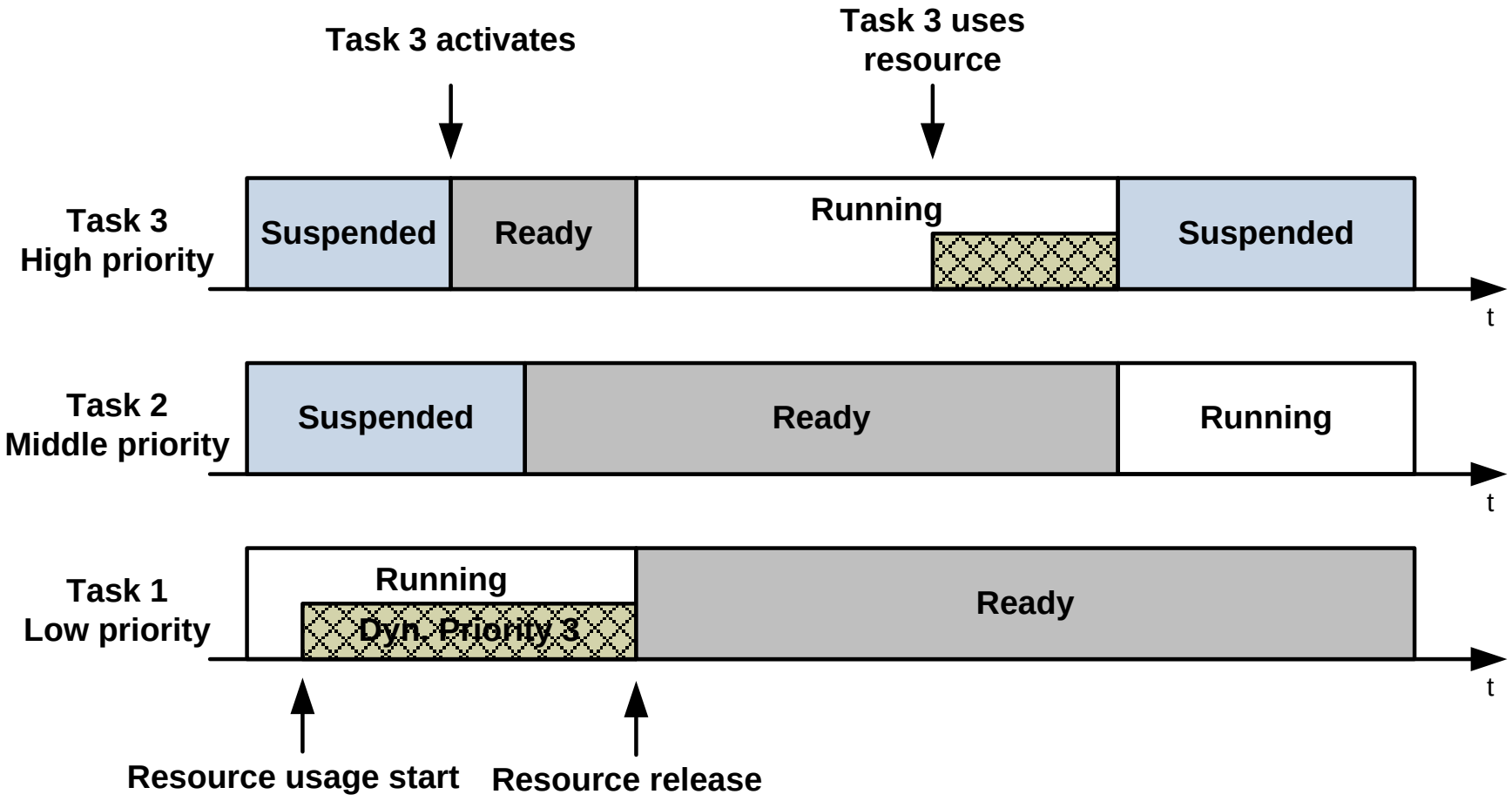
Famous example: NASA Mars Pathfinder Mission
http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html

Priority Ceiling / Inheritance

- Priority ceiling
  - Each resource is assigned a priority ceiling:
    Priority ceiling of resource is the highest priority of any task that may request the resource
  - Task's priority is raised to priority ceiling of a resource when they acquire it
  - Effective from the moment of resource acquisition

- Priority inheritance
  - Scheduler dynamically changes priority of task, when a higher priorized task is waiting for the resource
  - Effective from the moment of higher priorized resource acquisition attempt
  - Low priority tasks gets the priority of the waiting task

Priority Ceiling



© W. Zimmermann, et al. – Bussysteme in der Fahrzeugtechnik

Further Schedule Planning Algorithms

- Brute force - planning by searching
  - Simple and naïve approach
  - Exhaustive search, number of combinations for `n` tasks is `n!`
  - NP-complete problem when disregarding ready times + deadlines

- Earliest deadline first
  - Schedules tasks by their deadlines
  - Optimal, if tasks have same ready times
  - If not, not feasible in non-preemptive case

- Other examples:
  - Least laxity first / least slack time
    - For multi-processor systems
    - NP-complete in non-preemptive case

→ Will be covered in exercise

Robustness Mechanisms

AUTOSAR Watchdog Manager

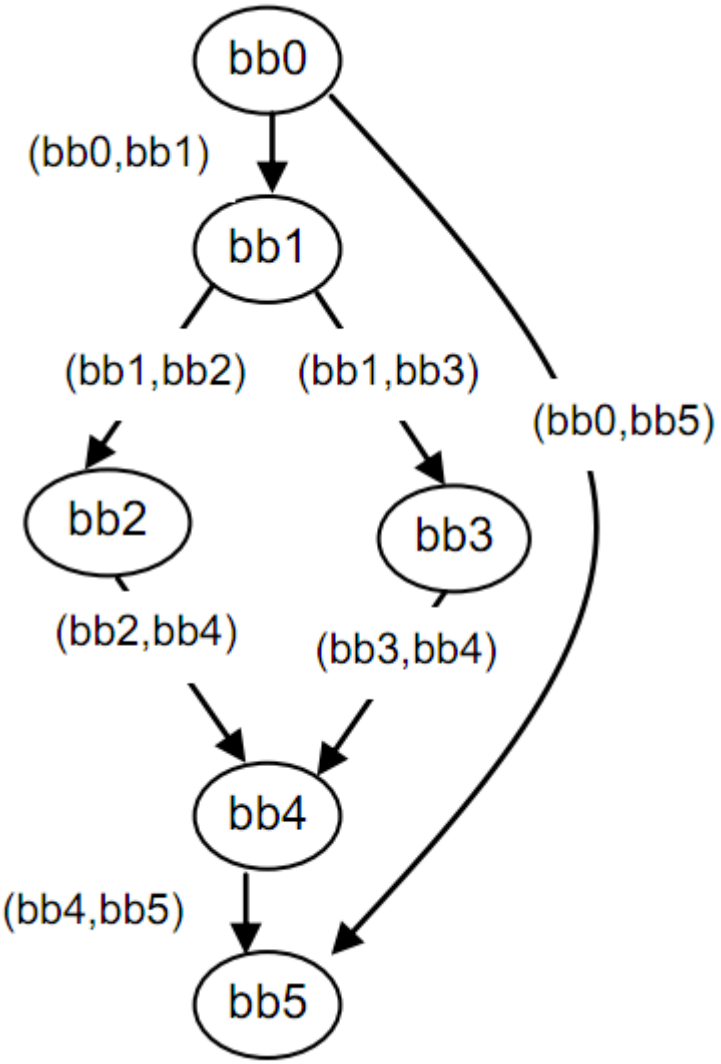
- Watchdog timer
  - Hardware device (internal/external)
  - Expects a signal within a configurable time
  - Triggers reaction on timeout
- Alive supervision
  - Checkpoints in supervised code sections
  - Watchdog checks periodically if checkpoints have been reached
  - Not suitable for aperiodical events
- Deadline supervision
  - Checkpoints in supervised code sections (e.g. beginning/end)
  - Watchdog checks **timing** of checkpoint **transitions** (aperiodic/episodical)

See: http://www.autosar.org/download/R4.0/AUTOSAR_SWS_WatchdogManager.pdf

Robustness Mechanisms

AUTOSAR Watchdog Manager

- Logical supervision
  - Checkpoints in supervised code sections
  - Program flow monitoring
  - Watchdog checks **legality** of **transitions**
  - Example: Transition to bb5 only valid if it originates from bb4 or bb0



See: http://www.autosar.org/download/R4.0/AUTOSAR_SWS_WatchdogManager.pdf

Robustness Mechanisms

AUTOSAR Operating System

- Memory protection
  - Hardware support required! (MPU/MMU)
  - Protect code / data / stack segments from unauthorized writes and reads
- Timing protection
  - Handle missed deadlines at runtime (timing fault)
  - Timing fault types:
    - Execution time protection
    - Resource locking time protection
    - Inter-arrival time protection
- Hardware protection
  - Privileged and non-privileged mode (kernel/user modes)

See: http://www.autosar.org/download/R4.0/AUTOSAR_SWS_OS.pdf