School of Computer Science & Engineering

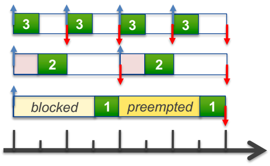
**COMP9242 Advanced Operating Systems**

2019 T2 Week 05a

**Real Time Systems Basics**

@GernotHeiser

Incorporating material by Stefan Petters and Anna Lyons




## Copyright Notice

**These slides are distributed under the Creative Commons Attribution 3.0 License**

- You are free:
  - to share—to copy, distribute and transmit the work
  - to remix—to adapt the work
- under the following conditions:
  - **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

    *"Courtesy of Gernot Heiser, UNSW Sydney"*

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode



# Real-Time Basics



## Real-Time Systems



## What's a Real-Time System?

A real-time system is a system that is required to react to stimuli from the environment (including passage of physical time) within time intervals dictated by the environment.
[Randell et al., Predictably Dependable Computing Systems, 1995]

Real-time systems have timing constraints, where the correctness of the system is dependent not only on the results of computations, but on *the time at which those results arrive*. [Stankovic, IEEE Computer, 1988]

**Issues:**
- Correctness: What are the temporal requirements?
- Criticality: What are the consequences of failure?



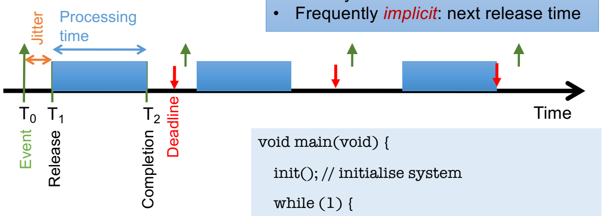
## Strictness of Temporal Requirements

- Hard real-time systems
- Weakly-hard real-time systems
- Firm real-time systems
- Soft real-time systems
- Best-effort systems

Strictness of temporal requirements

## Real-Time Tasks

**Real-time tasks have deadlines**
- Usually stated relative to release time
- Frequently *implicit*: next release time




```
void main(void) {
  init(); // initialise system
  while (1) {
T1 →  wait(); // timer, device interrupt, signal
T2 →  doJob();
  }
}
```

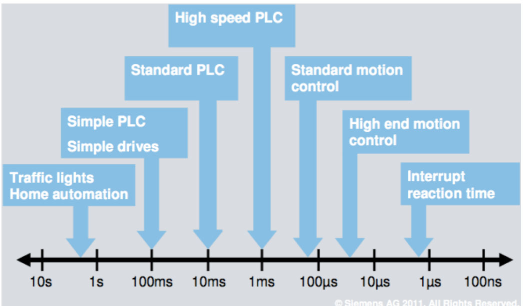
  UNSW

## Real Time ≠ Real Fast

| System | Deadline | Single Miss Conseq | Ultimate Conseq. |
|---|---|---|---|
| Car engine ignition | 2.5 ms | Catastrophic | Engine damage |
| Industrial robot | 5 ms | Recoverable? | Machinery damage |
| Air bag | 20 ms | Catastrophic | Injury or death |
| Aircraft control | 50 ms | Recoverable | Crash |
| Industrial process | 100 ms | Recoverable | Lost production, plant/environment damage |
| Pacemaker | 100 ms | Recoverable | Death |

  UNSW

## Example: Industrial Control

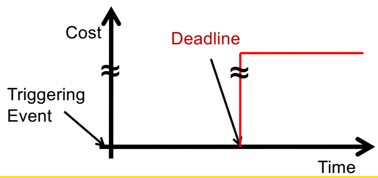


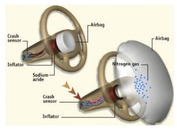

  UNSW

## Hard Real-Time Systems

- Safety-critical: Failure ⇒ death, serious injury
- Mission-critical: Failure ⇒ massive financial damage

- Deadline miss is *catastrophic*
- Steep and real *cost* function



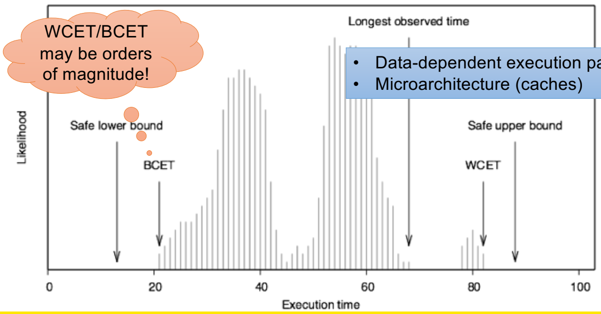



  UNSW

## Challenge: Execution-Time Variance

WCET/BCET may be orders of magnitude!

- Data-dependent execution paths
- Microarchitecture (caches)

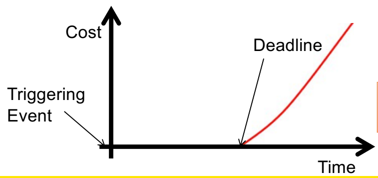



  UNSW

## Weakly-Hard Real-Time Systems

Tolerate small fraction of deadline misses

- Most feedback control systems (incl life-support!)
  - Control compensates for occasional miss
  - Becomes unstable if too many misses
- Typically integrated with fault tolerance for HW issues



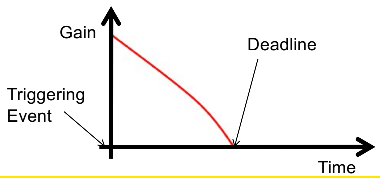

In practice, certifiers treat critical avionics as hard RT

  UNSW

## Firm Real-Time Systems

Result obsolete if deadline missed (loss of revenue)

- Forecast systems
- Trading systems

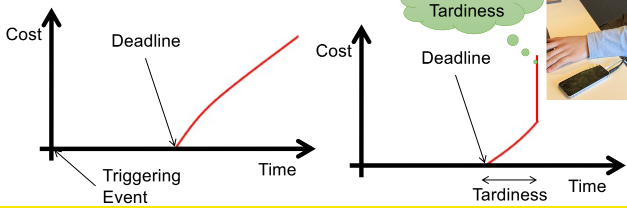


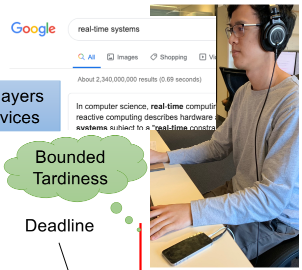

 

## Soft Real-Time Systems

Deadline miss undesirable but tolerable, affects QoS

- Media players
- Web services



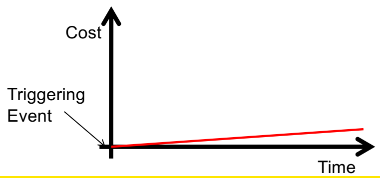



## Best-Effort Systems

No deadline

In practice, duration is rarely totally irrelevant



## Real-Time Operating System (RTOS)

- Designed to support real-time operation
  - Fast context switches, fast interrupt handling
  - More importantly, *predictable* response time
- **Main duty is scheduling tasks to meet their deadline**

Requires analysis of worst-case execution time (WCET)

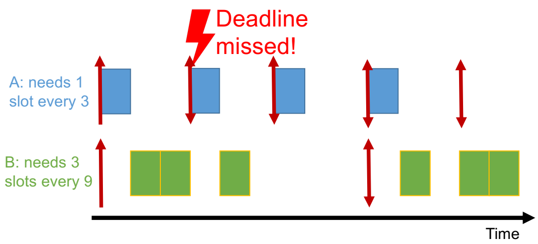
Traditional RTOS is very primitive
- single-mode execution
- no memory protection
- inherently cooperative
- *all code is trusted*

**RT vs OS terminology:**
- "task" = thread
- "job" = execution of thread resulting from event

## Real-Time Scheduling

- Ensuring all deadlines are met is harder than bin-packing
- Reason: time is not fungible



## Real-Time Scheduling

- Ensuring all deadlines are met is harder than bin-packing
- Time is not fungible

Terminology:
- A set of tasks is **feasible** if there is a known algorithm that will schedule them (i.e. all deadlines will be met).
- A scheduling algorithm is **optimal** if it can schedule all **feasible** task sets.

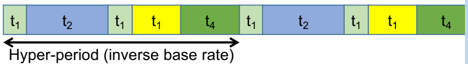
 

## Cyclic Executives

- Very simple, completely static, scheduler is just table
- Deadline analysis done off-line
- Fully deterministic

Drawback: Latency of event handling is hyper-period

```
while (true) {
    wait_tick();
    job_1();
    wait_tick();
    job_2();
    wait_tick();
    job_1();
    wait_tick();
    job_3();
    wait_tick();
    job_4();
}
```

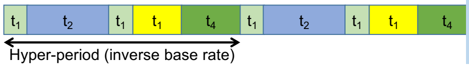



  UNSW

## Are Cyclic Executives Optimal?

- Theoretically yes if can slice (interleave) tasks
- Practically there are limitations:
  - Might require very fine-grained slicing
  - May introduce significant overhead

```
while (true) {
    wait_tick();
    job_1();
    wait_tick();
    job_2();
    wait_tick();
    job_1();
    wait_tick();
    job_3();
    wait_tick();
    job_4();
}
```




  UNSW

## On-Line RT Scheduling

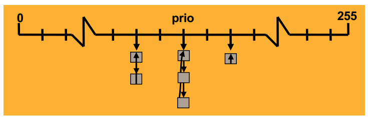
- Scheduler is part of the OS, performs scheduling decision on-demand
- Execution order not pre-determined
- Can be preemptive or non-preemptive
- Priorities can be
  - fixed: assigned at admission time
    - scheduler doesn't change prios
    - system may support dynamic adjustment of prios
  - dynamic: prios potentially different at each scheduler run

  UNSW

## Fixed-Priority Scheduling (FPS)

- Classic L4 scheduling is a typical example:
  - always picks highest-prio runnable thread
  - round-robin within prio level
  - will preempt if higher-prio thread is unblocked or time slice depleted

FPS is not optimal, i.e. cannot schedule some feasible sets




In general may or may not:
- preempt running threads
- require unique prios

  UNSW

## Rate Monotonic Priority Assignment (RMPA)

- Higher rate ⇒ higher period:
  - $T_i < T_j \Rightarrow P_i > P_j$

T: period
1/T: rate
P: priority
U: utilisation

- Schedulability test: Can schedule task set with periods $\{T_1 \dots T_n\}$ if

Assumes "*implicit*" deadlines: release time of next job

$$U \equiv \sum C_i/T_i \leq n(2^{1/n}-1)$$

RMPA is optimal for FPS

| n | 1 | 2 | 3 | 4 | 5 | 10 | ∞ |
|---|---|---|---|---|---|---|---|
| U [%] | 100 | 82.8 | 78.0 | 75.7 | 74.3 | 71.8 | log(2) = 69.3 |

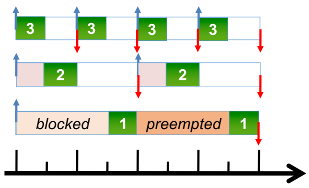
  UNSW

## Rate-Monotonic Scheduling Example

RMPA schedulability bound is sufficient but not necessary

WCET

| Task | T | P | C | U [%] |
|---|---|---|---|---|
| $t_3$ | 20 | 3 | 10 | 50 |
| $t_2$ | 40 | 2 | 10 | 25 |
| $t_1$ | 80 | 1 | 20 | 25 |
| | | | | 100 |

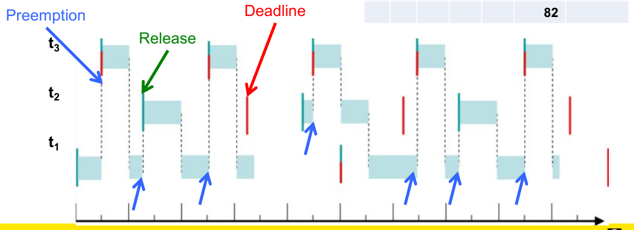



  UNSW

## Another RMPA Example

| | P | C | T | D | U [%] | release |
|---|---|---|---|---|---|---|
| $t_3$ | 3 | 5 | 20 | 20 | 25 | 5 |
| $t_2$ | 2 | 8 | 30 | 20 | 27 | 12 |
| $t_1$ | 1 | 15 | 50 | 50 | 30 | 0 |
| | | | | | **82** | |

Deadline




  UNSW

## Dynamic Prio: Earliest Deadline First (EDF)

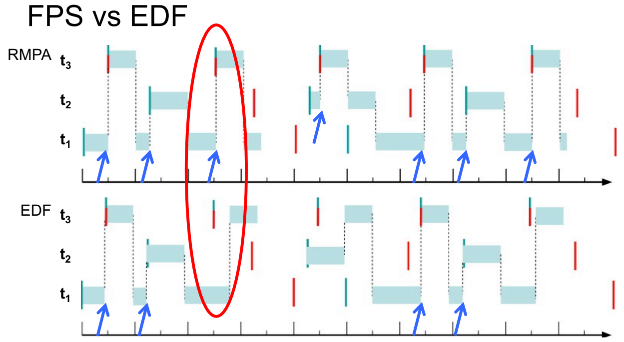
- Job with closest deadline executes
  - priority assigned at job level, not task (i.e. thread) level
  - deadline-sorted release queue
- Schedulability test: Can schedule task set with periods $\{T_1 \ldots T_n\}$ if

$$U \equiv \sum C_i/T_i \leq 1$$

Preemptive EDF is optimal

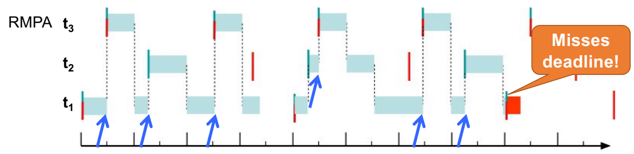
  UNSW

## FPS vs EDF




  UNSW

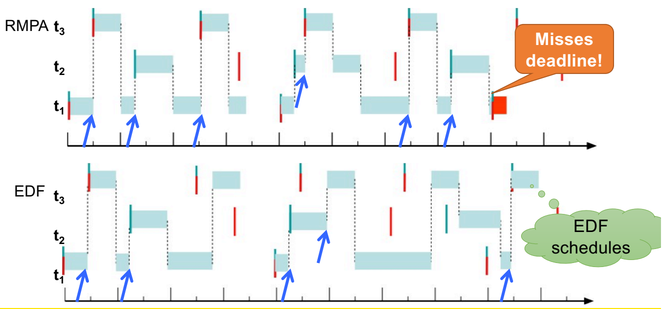
## FPS vs EDF




| Task | P | C | T | D | U [%] | release |
|---|---|---|---|---|---|---|
| $t_3$ | 3 | 5 | 20 | 20 | 25 | 5 |
| $t_2$ | 2 | 8 | 30 | 20 | 27 | 12 |
| $t_1$ | 1 | 15 | 40 | 40 | 37.5 | 0 |
| | | | | | **89.5** | |

  UNSW

## FPS vs EDF

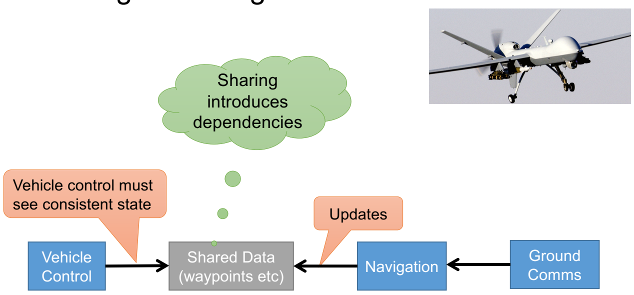



  UNSW

# Resource Sharing

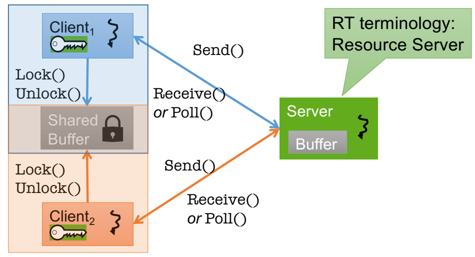
  UNSW

## Challenge: Sharing



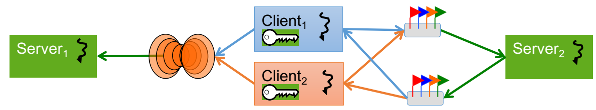

  

## Critical Sections: Locking vs Delegation



## seL4 Implementing Delegation




```
serv_local() {
  ...
  wait(ep);
  while (1) {
    /* critical section */
    Reply&wait(ep);
  }
}
```

```
client() {
  while (1) {
    ...
    call(ep);
    ...
    signal(not_ry);
    ...
    wait(not_rq);
  }
}
```

```
serv_remote() {
  ...
  while (1) {
    wait(not_rq);
    /* critical section */
    signal(not_ry);
  }
```

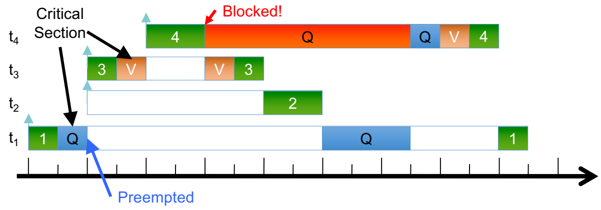
**Hoare-style monitor**
Suitable intra-core

**Semaphore synchronisation**
Suitable inter-core

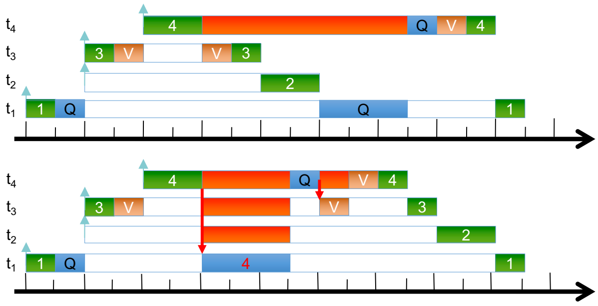
## Problem: Priority Inversion

- High-priority job is blocked by low-prio for a long time
- Long wait chain: $t_1 \rightarrow t_4 \rightarrow t_3 \rightarrow t_2$
- Worst-case blocking time of $t_1$ bounded by total WCET: $C_2+C_3+C_4$



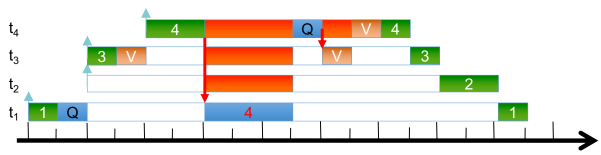
## Solution 1: Priority Inheritance ("Helping")



## Solution 1: Priority Inheritance ("Helping")

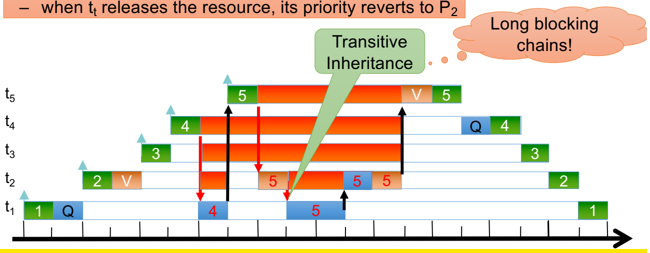
If $t_1$ blocks on a resource held by $t_2$, *and* $P_1>P_2$, then
- $t_2$ is temporarily given priority $P_1$
- when $t_1$ releases the resource, its priority reverts to $P_2$



## Solution 1: Priority Inheritance ("Helping")

If $t_1$ blocks on a resource held by $t_2$, *and* $P_1>P_2$, then
- $t_2$ is temporarily given priority $P_1$
- when $t_i$ releases the resource, its priority reverts to $P_2$

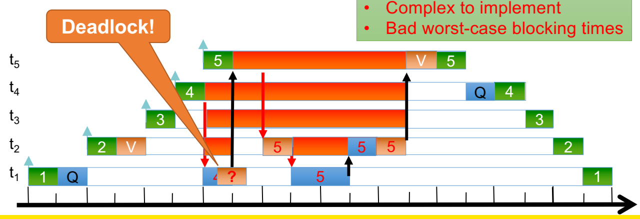


## Solution 1: Priority Inheritance ("Helping")

If $t_1$ blocks on a resource held by $t_2$, *and* $P_1>P_2$, then
- $t_2$ is temporarily given priority $P_1$
- when $t_i$ releases the resource, its priority

**Priority Inheritance:**
- Easy to use
- Potential deadlocks
- Complex to implement
- Bad worst-case blocking times

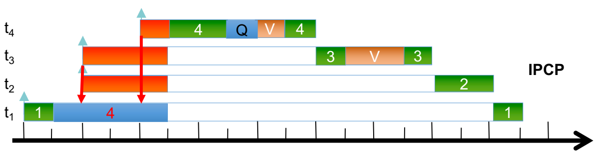


## Solution 2: Priority Ceiling Protocol (PCP)

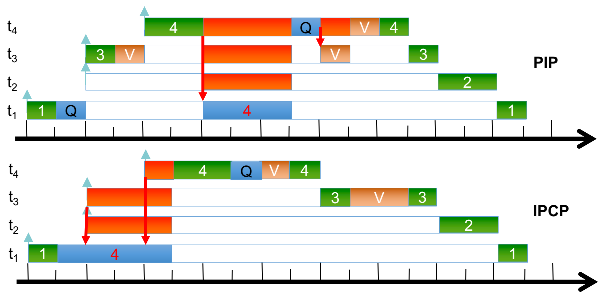
- Aim: Block at most once, avoid deadlocks
- Idea: Associate *ceiling priority* with each resource
  - Ceiling = Highest prio of jobs that may access the resource
  - On access, bump prio of job to ceiling

Immediate prio ceiling protocol (IPCP)



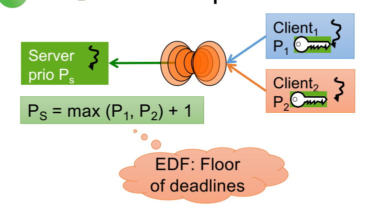

 

## IPCP vs PIP



## seL4 ICPC Implementation With Delegation



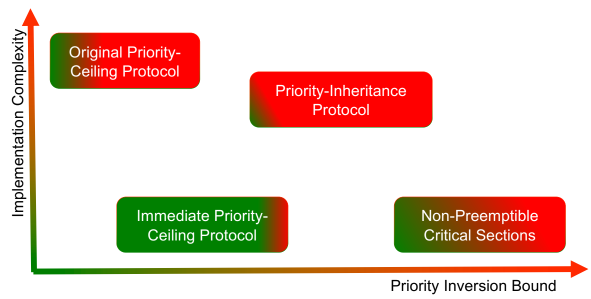

**Immediate Priority Ceiling:**
- Requires correct prio config
- Deadlock-free
- Easy to implement
- Good worst-case blocking times

Each task must declare all resources at admission time
- System must maintain list of tasks using resource
- Defines ceiling priority

Easy to enforce with caps

## seL4 Comparison of Locking Protocols



# Scheduling Overloaded RT Systems

# Naïve Assumption: Everything is Schedulable

**Standard assumptions of classical RT systems:**
- All WCETs known
- All jobs complete within WCET
- Everything is Trusted

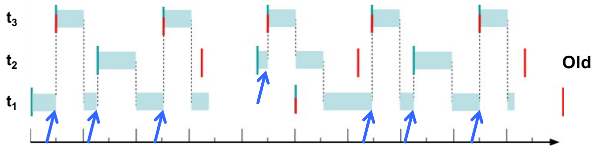
**More realistic: Overloaded system:**

Which job will miss its deadline?

- Total utilisation exceeds schedulability bound
- Cannot trust everything to obey declared WCET

# Overload: FPS

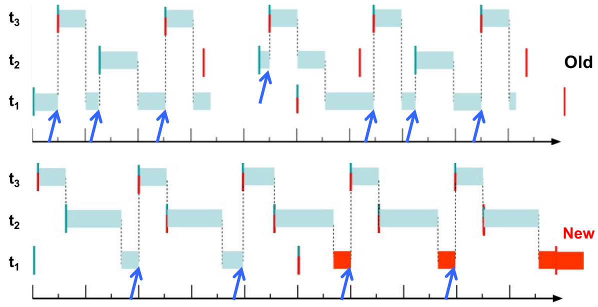



| Task | P | C | T | D | U [%] |
|---|---|---|---|---|---|
| $t_3$ | 3 | 5 | 20 | 20 | 25 |
| $t_2$ | 2 | 12 | 20 | 20 | 60 |
| $t_1$ | 1 | 15 | 50 | 50 | 30 |
| | | | | | 115 |

New

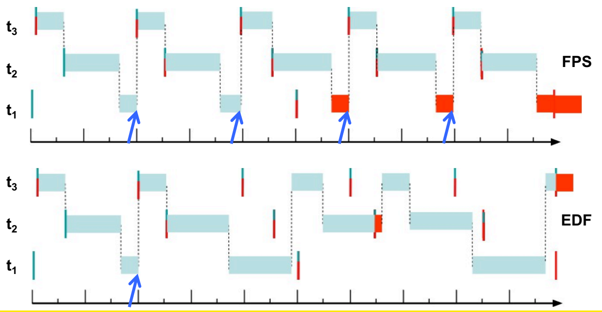
 

# Overload: FPS



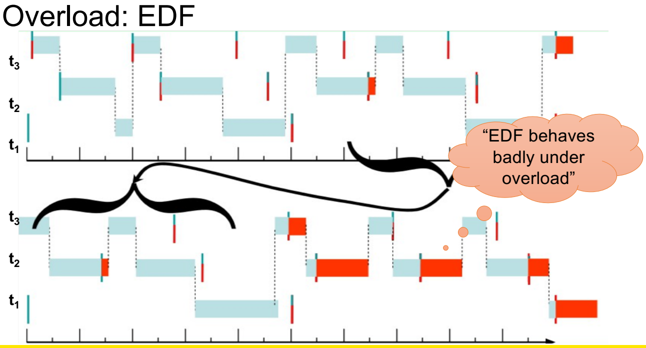

 

# Overload: FPS vs EDF



# Overload: EDF



# Mixed-Criticality Systems




## Mixed Criticality Systems





## Mixed Criticality

Need temporal isolation!

**NW driver must preempt control loop**
- … to avoid packet loss
- Driver must run at high prio (i.e. RMPA)
- *Driver must not monopolise CPU*

Runs every 100 ms for a few millisecods

Runs frequently but for short time (order of μs)

Sensor readings → Control loop ↔ NW driver ← NW interrupts




## Mixed Criticality

**NW driver must preempt control loop**
- … to avoid packet loss
- Driver must run at high prio (i.e. RMPA)
- *Driver must not monopolise CPU*

Critical system certification:
- expensive
- conservative assumptions
  - eg highly pessimistic WCET

**Certification requirement: More critical components must *not* depend on any less critical ones! [ARINC-653]**

- Must minimise critical software
- Need temporal isolation: Budget enforcement