COMP9242 Advanced OS

S2/2016 W08: **Real-Time Systems**

@GernotHeiser

Incorporating lectures by Stefan Petters

Never Stand Still | Engineering | Computer Science and Engineering

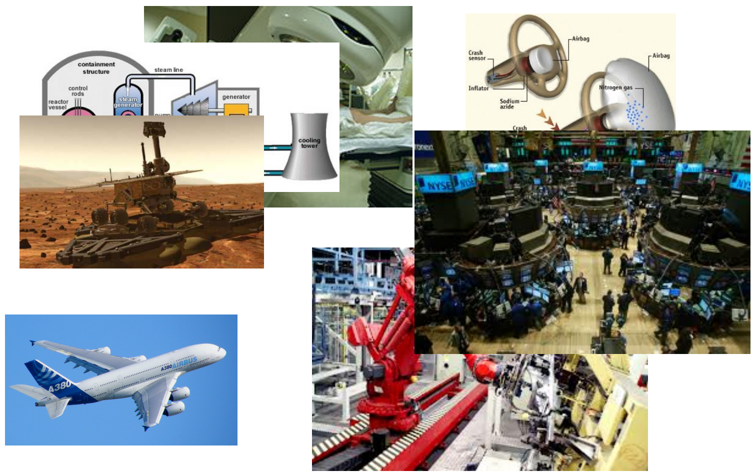
Copyright Notice

Real-Time System: Definition

A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period

- Correctness depends not only on the logical result (function) but also the time it was delivered
- Failure to respond is as bad as delivering the wrong result!

Real-Time Systems



Types of Real-Time Systems

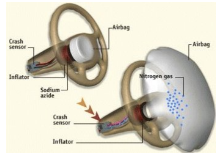
- Hard real-time systems
- Weakly-hard real-time systems
- Firm real-time systems
- Soft real-time systems
- Best-effort systems

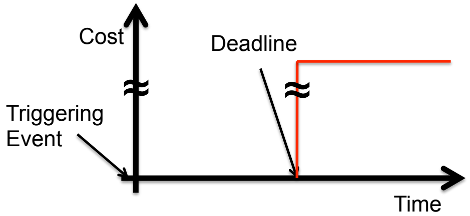
- Real-time systems typically deal with *deadlines*:
  - A deadline is a time instant by which a response has to be completed
  - A deadline is usually specified as *relative* to an event
    - The *relative deadline* is the *maximum allowable response time*
    - Absolute deadline: event time + relative deadline

  UNSW

Hard Real-Time Systems

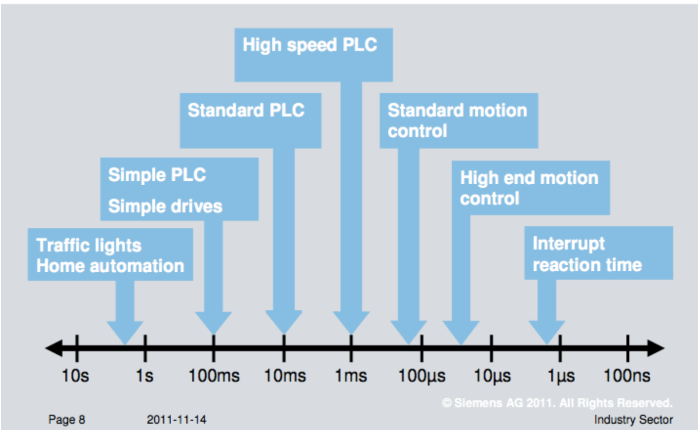
- Deadline miss is "catastrophic"
  - safety-critical system: failure results in death, severe injury
  - mission-critical system: failure results in massive financial damage
- Steep and real "cost" function





  UNSW

Eg RT Requirements in Industrial Automation



Source: Siemens

  UNSW

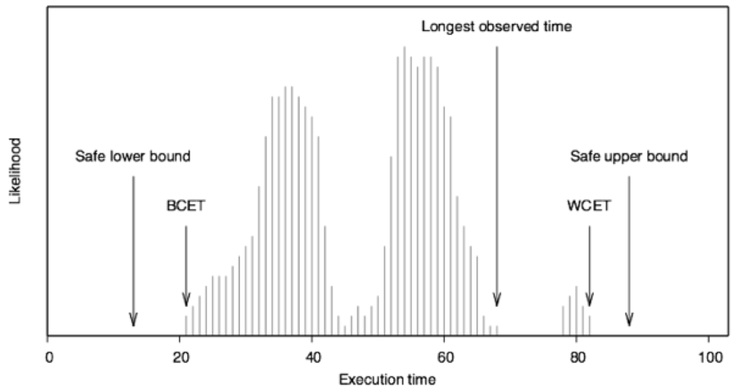
Real-Time ≠ Real Fast

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

Challenge of real-time systems: **Guaranteeing** deadlines

  UNSW

Typical Execution-Time Profile

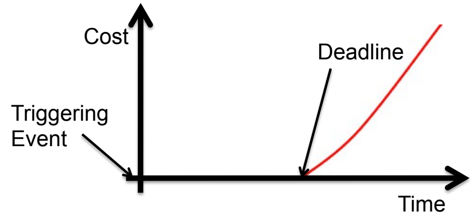


Variance may be orders of magnitude!

- Data-dependent execution path
- Micro-architectural features: pipelines, caches

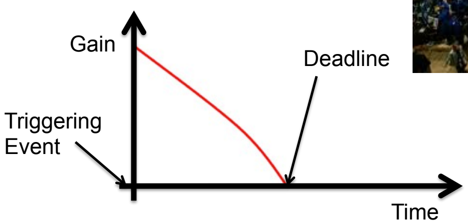
Weakly-Hard Real-Time Systems

- Tolerate a (small) fraction of deadline misses
  - Most feedback control systems (including life-supporting ones!)
    - occasionally missed deadline can be compensated at next event
    - system becomes unstable if too many deadlines are missed
  - Typically integrated with other fault tolerance
    - electro-magnetic interference, other hardware issues



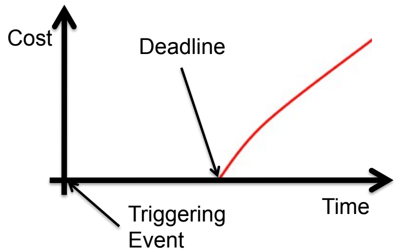
Firm Real-Time Systems

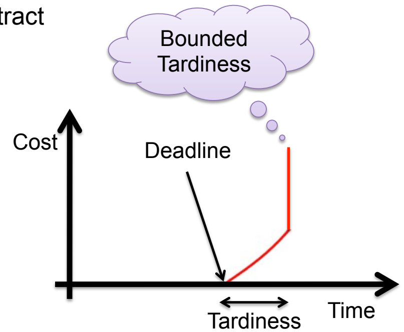
- Deadline miss makes computation obsolete
  - Typical examples are forecast systems
    - weather forecast
    - trading systems
- Cost may be loss of revenue (gain)



Soft Real-Time Systems

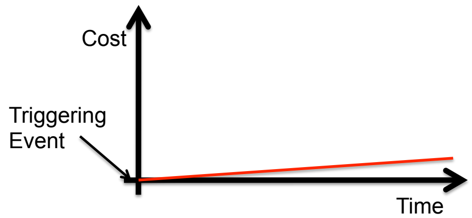
- Deadline miss is undesired but tolerable
  - Frequently results on quality-of-service (QoS) degradation
    - eg audio, video rendering
    - Steep "cost" function
- Cost of deadline miss may be abstract





Best-Effort Systems

- No deadlines, timeliness is not part of required operation
- In reality, there is at least a nuisance factor to excessive duration
  - response time to user input
- Again, "cost" may be reduced gain



Real-Time Operating System (RTOS)

- Designed to support real-time operation
  - Fast context switches, fast interrupt handling?
  - Yes, but *predictable* response time is more important
    - "Real time is not real fast"
  - Analysis of *worst-case execution time* (WCET)
- Support for *scheduling policies* appropriate for real time
- Classical RTOSes very primitive
  - single-mode execution
  - no memory protection
  - essentially a scheduler with a threads package
  - "real-time executive"
  - inherently cooperative
- Many modern uses require actual OS technology for isolation
  - generally microkernels
  - QNX, Integrity, VXworks, L4 kernels

Approaches to Real Time

- Clock-driven (cyclic)
  - *Periodic scheduling*
  - Typical for control loops
  - Fixed order of actions, round-robin execution
  - *Statically* determined (static schedule) if periods are fixed
    - need to know all execution parameters at system configuration time

> **Emulation on event-driven system:** treat clock tick as event

- Event-driven
  - *Sporadic scheduling*
  - Typical for reactive systems (sensors & actuators)
  - Static or dynamic schedules
  - Analysis requires bounds on event arrivals

> **Emulation on clock-driven system**: buffer event (IRQ) until timer tick

Real-Time System Operation

- Time-triggered
  - Pre-defined temporal relation of events
  - event is not serviced until its defined *release time* has arrived
- Event-triggered
  - timer interrupt
  - asynchronous events
- Rate-based
  - activities get assigned CPU shares ("*rates*")

Real-Time Task Model

- **Job**: unit of work to be executed
  - … resulting from an event or time trigger
- **Task**: set of related jobs which provide some system function
  - A *task* is a sequence of *jobs* (typically executing same function)
  - Job *i+1* of of a task cannot start until job *i* is completed/aborted

- Periodic tasks
  - Time-driven and all relevant characteristics known a priori
    - Task t characterized by period T_i, deadline, D_i and execution time C_i
    - Applies to all jobs of task
- Aperiodic tasks
  - Event driven, characteristics are not known a priori
    - Task t characterized by period $T_{i,}$ deadline D_i and arrival distribution
- Sporadic tasks
  - Aperiodic but with known minimum inter-arrival time T_i
  - treated similarly to periodic task with period T_i

Standard Task Model

C: Worst-case computation time (WCET)

T: Period (periodic) or minimum inter-arrival time (sporadic)

D: Deadline (relative, frequently "implicit deadlines" D=T)

J: Release jitter

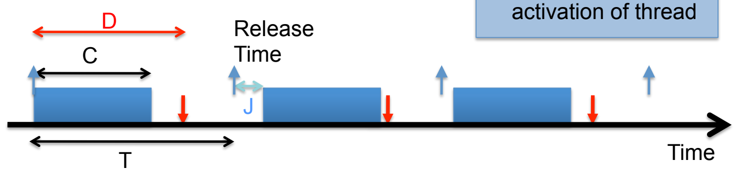
P: Priority: higher number means higher priority

B: Worst-case blocking time

R: Worst-case response time

U: Utilisation; U=C/T

OS terminology:
- "task" = thread
- "job" = event-based activation of thread



Task Constraints

- Deadline constraint: must complete before deadline
- Resource constraints:
  - Shared (R/O), exclusive (W-X) access
  - Energy
  - Precedence constraints:
    $t_1 \Rightarrow t_2$: t_2 execution cannot start until t_1 is finished
  - Fault-tolerance requirements
    - eg redundancy

- Scheduler's job to ensure that constraints are met!

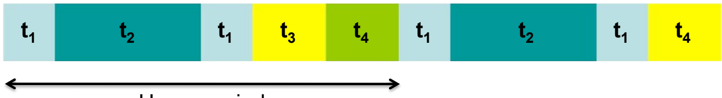
Scheduling

- Preemptive vs non-preemptive
- Static (fixed, off-line) vs dynamic (on-line)
- Clock-driven vs priority-based
  - clock-driven is static, only works for very simple systems
  - priorities can be static (pre-computed and fixed) or dynamic
  - dynamic priority adjustment can be at task-level (each job has fixed prio) or job-level (jobs change prios)

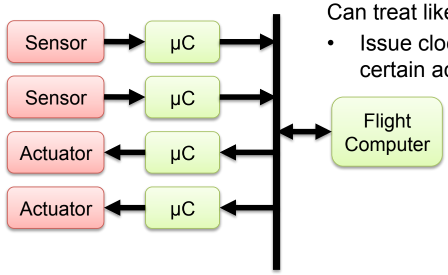
Clock-Driven (Time-Triggered) Scheduling

- Typically implemented as time "frames" adding up to "base rate"
- Advantages
  - fully deterministic
  - "cyclic executive" is trivial
  - minimal overhead
- Disadvantage:
  - Big latencies if event rate doesn't match base rate (hyper-period)
  - Inflexible

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

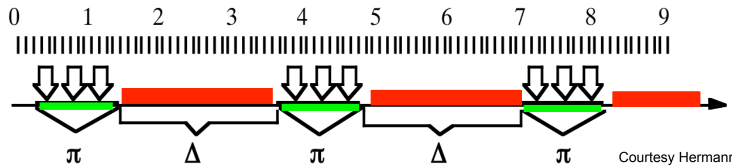


Synchronous Distributed RT Systems



Can treat like single system if clocks synchronised?
- Issue clock drift: can only synchronise within certain accuracy

Time-triggered architecture
Idea: use *sparse time*:
- Restrict events to *active interval* π
- Separated by *silence interval* Δ
- Δ allows for clock drift and communications time



Courtesy Hermann Kopetz

Non-Preemptive Scheduling

- Minimises context-switching overhead
  - Significant cost on modern processors (pipelinies, caches)
- Easy to analyse timeliness
- Drawbacks:
  - Larger response times for "important" tasks
  - Reduced utilisation, schedulability
    - In many cases cannot produce schedule despite plenty idle time
  - Can't re-use slack (eg for best-effort)
- Only used in very simple systems

Fixed-Priority Scheduling (FPS)

- Real-time priorities are absolute:
  - Scheduler always picks highest-priority job
- Obviously easy to implement, low overhead
- Drawbacks: inflexible, sub-optimal
  - Cannot schedule some systems which are schedulable preemptively
- Note: "Fixed" in the sense that system doesn't change them
  - OS may support dynamic adjustment
  - Requires on-the-fly (re-)admission control

Rate-Monotonic Scheduling (RMS)

- RMS: Standard approach to fixed priority assignment
  - $T_i < T_j \Rightarrow P_i > P_j$
  - 1/T is the “rate” of a task
- RMS is *optimal* for fixed priorities
- Schedulability test: RMS can schedule n tasks with D=T if

  $U \equiv \sum C_i/T_i \leq n(2^{1/n}-1)$; $\lim_{n\to\infty} U = \log 2$

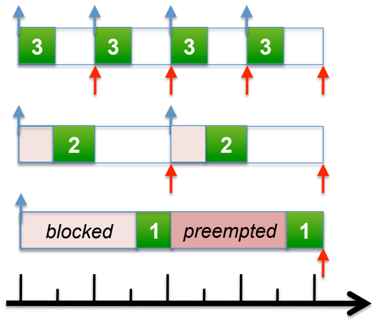
n	1	2	3	4	5	10	∞
U [%]	100	82.8	78.0	75.7	74.3	71.8	69.3

- If D<T replace by *deadline-monotonic scheduling* (DMS):
  - $D_i < D_j \Rightarrow P_i > P_j$
- DMS is also optimal (but schedulability bound is more complex)

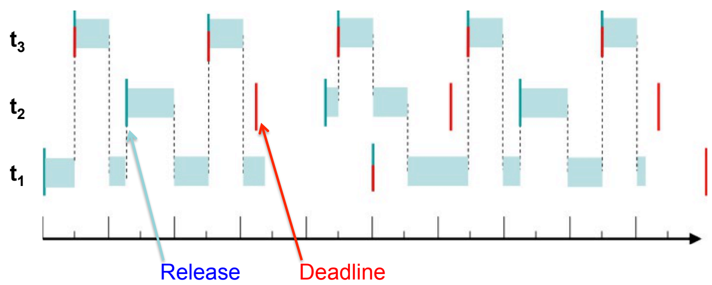
Rate-Monotonic Scheduling (RMS)

RMS schedulability condition is sufficient but not necessary

	T	D	P	C	U [%]
t_3	20	20	3	10	50
t_2	40	40	2	10	25
t_1	80	80	2	20	25
					100



FPS Example



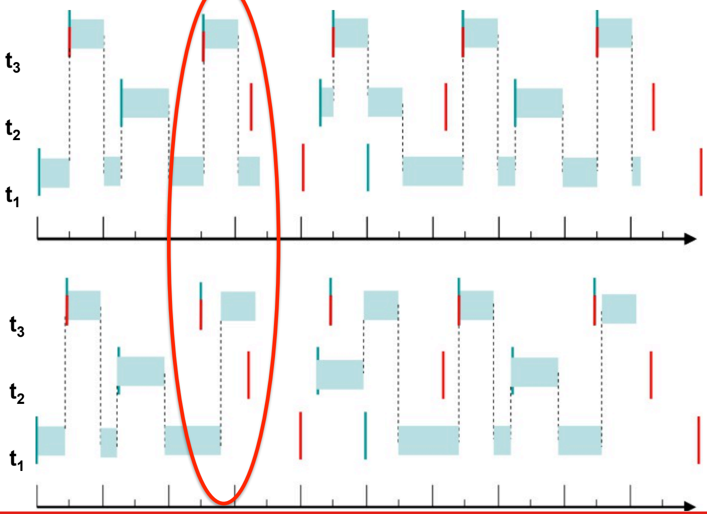
	P	C	T	D	U [%]	release
t_3	3	5	20	20	25	5
t_2	2	8	30	20	27	12
t_1	1	15	50	50	30	0
					82	

Earliest Deadline First (EDF)

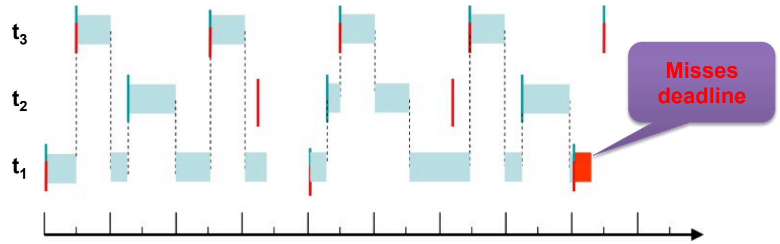
- Dynamic scheduling policy
- Job with closest deadline executes
- Preemptive EDF with D=T is *optimal*: n jobs can be scheduled iff

  $U \equiv \sum C_i/T_i \leq 1$
  - necessary and sufficient condition
  - no easy test if D≠T

FPS vs EDF

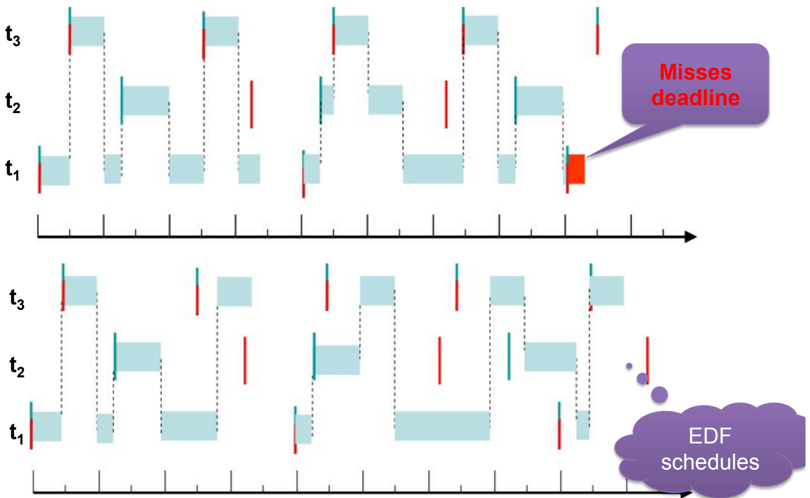


FPS vs EDF

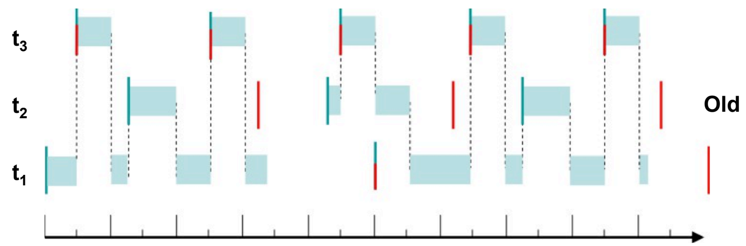


	P	C	T	D	U [%]	release
t_3	3	5	20	20	25	5
t_2	2	8	30	20	27	12
t_1	1	15	40	40	37.5	0
					89.5	

FPS vs EDF



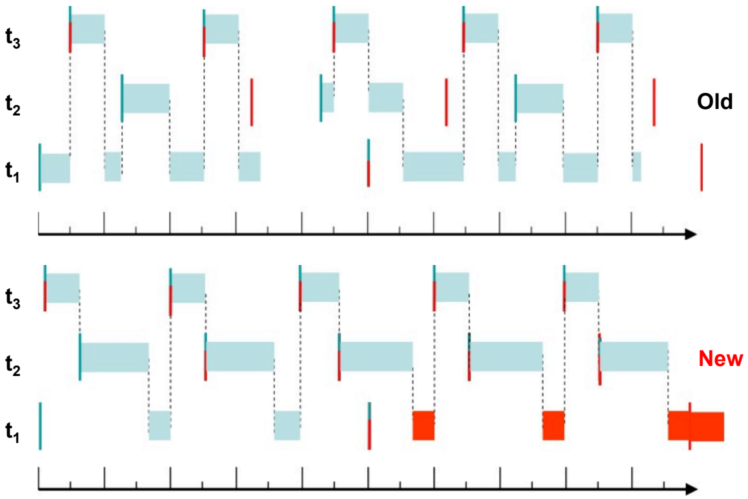
Overload: FPS



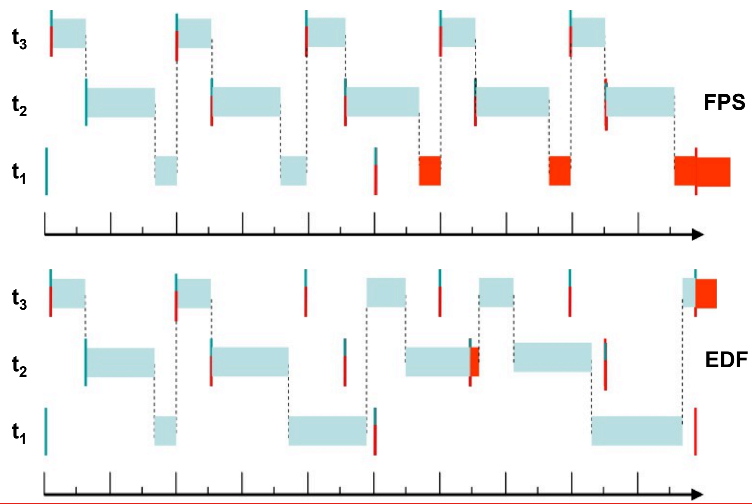
New

	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

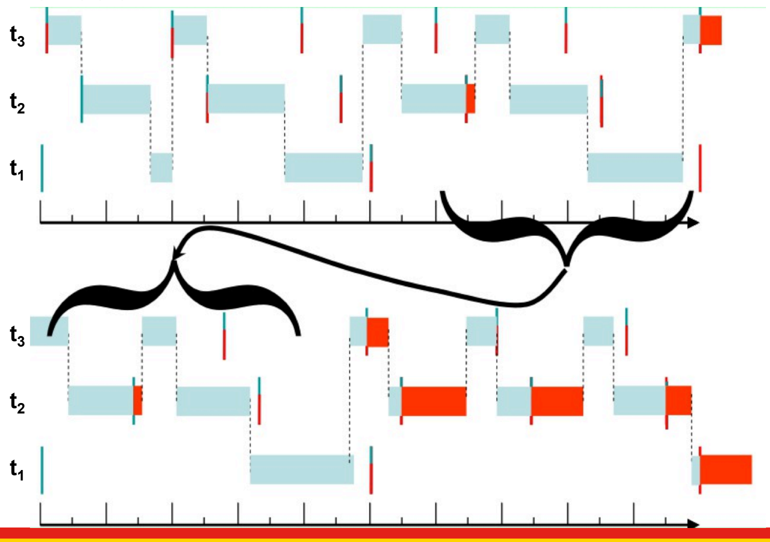
Overload: FPS



Overload: FPS vs EDF



Overload: EDF



Overload: FPS vs EDF

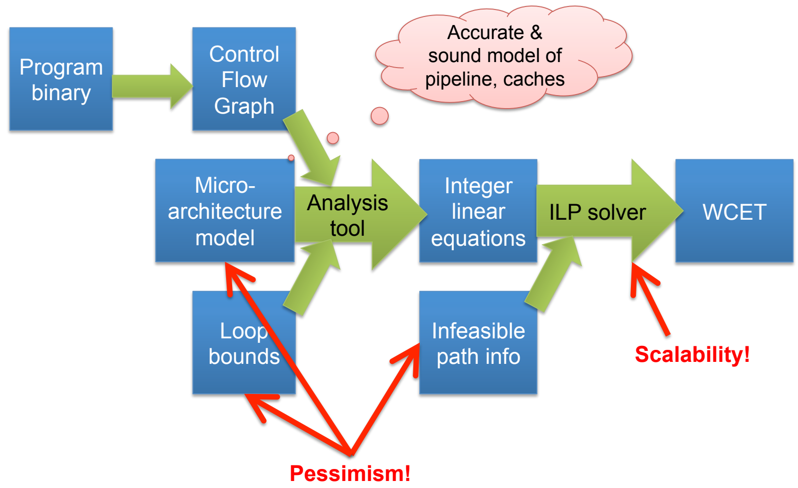
On overload, (by definition!) *lowest-prio jobs miss deadlines*

- Result is well-defined and -understood for FPS
  - Treats highest-prio task as “most important”
  - … but that may not always be appropriate!
  - Under transient overload may miss deadlines of higher-priority tasks
- Result is unpredictable (seemingly random) for EDF
  - May result in all tasks missing deadlines!
  - Under constant overload will scale back all tasks
  - No concept of task “importance”
  - “EDF behaves badly under overload”
  - Main reason EDF is unpopular in industry

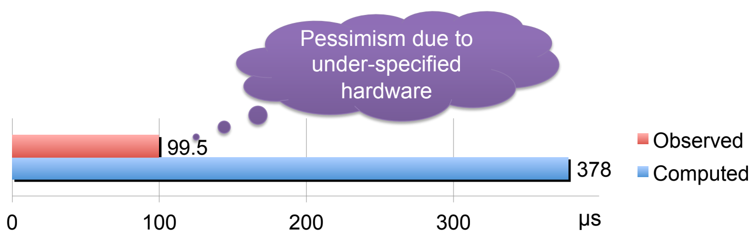
Why Have Overload?

- Faults (software, EMI, hardware)
- Incorrect assumptions about environment
- Optimistic WCET
  - Computing WCET of non-trivial programs is hard, often infeasible!
  - Safe WCET bounds tend to be highly pessimistic (orders of magnitude!)
  - WCET often very unlikely and orders of magnitude worse than “normal”
    - Estimation inaccuracies from caches, pipelines, under-specified hardware…
    - “notrmal” vs “exceptional” operating conditions
    - requires massive over-provisioning
  - Some systems have effectively unbounded execution time
    - e.g. object tracking

WCET Analysis



seL4 WCET Analysis [Blackham et al ’11, ‘12]



WCET presently limited by verification practicalities
- **without regard to verification achieved 50 µs**
- **10 µs seem achievable**
- **BCET ~ 1µs**

Why Have Overload?

- Faults (software, EMI, hardware)
- Incorrect assumptions about environment
- Optimistic WCET
  - Computing WCET of non-trivial programs is hard, often infeasible!
  - Safe WCET bounds tend to be highly pessimistic (orders of magnitude!)
  - WCET often very unlikely and orders of magnitude worse than “normal”
    - thanks to caches, pipelines, under-specified hardware
    - requires massive over-provisioning

Way out?
- Need explicit notion of importance: *criticality*
- Expresses effect of failure on the system mission
  - Catastrophic, hazardous, major, minor, no effect
- *Orthogonal to scheduling priority!*

Mixed Criticality

- A mixed-criticality system supports multiple criticalities concurrently
  - Eg in avionics: consolidation of multiple functionalities
  - Driver: space, weight and power (SWaP) limitations (translates into $$$)

Certification of critical components must not depend on less critical ones!

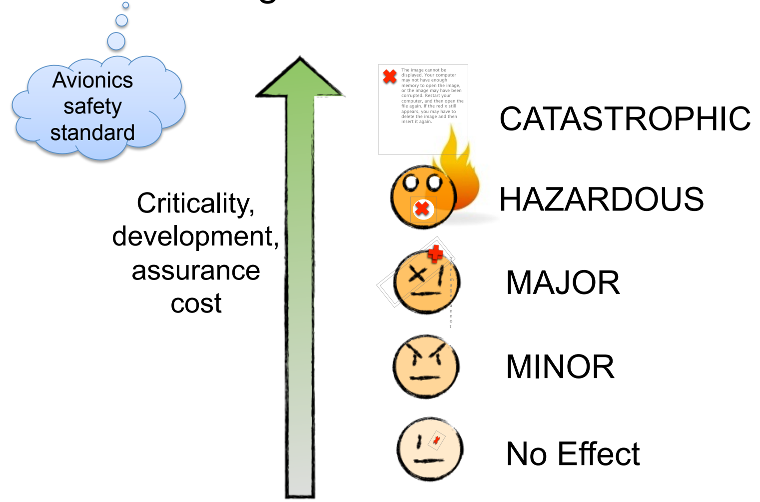
Higher criticality certification
- **More costly**
- **More pessimistic (eg WCET)**

Flight control
Highly critical

Autopilot
Less critical

OS

DO-178B Design Assurance Levels



Mixed Criticality Example

Criticality	T	U_{HIGH}	U_{MED}	U_{LOW}	$U_{average}$
High	10	50%	20%	20%	0.05%
Medium	1	N/A	60%	20%	2.5%
Low	100	N/A	N/A	unknown	10%
Total		**50%**	**80%**	**over**	**12.55%**

- **HIGH** alone has poor utilisation ⇒ gain from consolidation
- **HIGH+MEDIUM** can be scheduled for med-crit WCET
- **HIGH+MEDIUM cannot** be scheduled for most conservative WCET
- Idea: schedule under optimistic assumptions
  - Prioritise **HIGH** if it overruns its **MEDIUM** WCET

Mixed Criticality Implementation

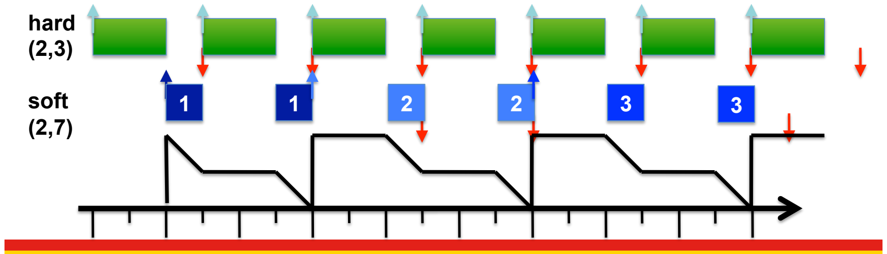
- Whenever running **LOW** job, ensure no **HIGH** job misses deadline
- Switch to *critical mode* when not assured
  - Various approaches to determine switch
  - eg. *zero slack*: **HIGH** job's deadline = its WCET
- Criticality-mode actions:
  - FP: temporarily raise all **HIGH** jobs' prios above that of all others
    - Simply preempting present job won't help!
  - EDF: drop all **LOW** deadlines earlier than next **HIGH** deadline
- Issues:
  - Treatment of **LOW** jobs still rather indiscriminate
  - Need to determine when to switch to normal mode, restore prios
  - Switch must be fast – must be allowed for in schedulability analysis!

CPU Bandwidth Reservations

- Idea: Utilisation U = C/T can be seen as required CPU *bandwidth*
  - Account time use against reservation C
  - Not runnable when reservation exhausted
  - Replenish every T
- Can support over-committing
  - Reduce **LOW** reservations if **HIGH** reservations fully used
- Advantages:
  - Allows dealing with jobs with unknown (or untrusted) deadlines
  - Allows integrating sporadic, asynchronous and soft tasks
- Modelled as a "server" which hands out time to jobs
  - effectively a simple (FIFO) sub-scheduler

Constant Bandwidth Server (CBS)

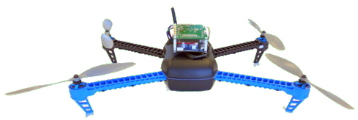
- Popular theoretical model suitable for EDF [Abeni & Buttazzo '98]
- CBS schedules specified bandwidth
  - Server has (Q,T): *budget* Q = U × T and period T
  - generates appropriate absolute EDF deadlines on the fly
  - when budget goes to zero, new deadline is generated with new budget
    - Hard reservation: $D_{i+1} = D_i + T$ (rate-limits)
    - Soft reservation: $D_{i+1} = t + T$ (postpone deadline)
  - Schedulability: $\sum U_i \leq 1$

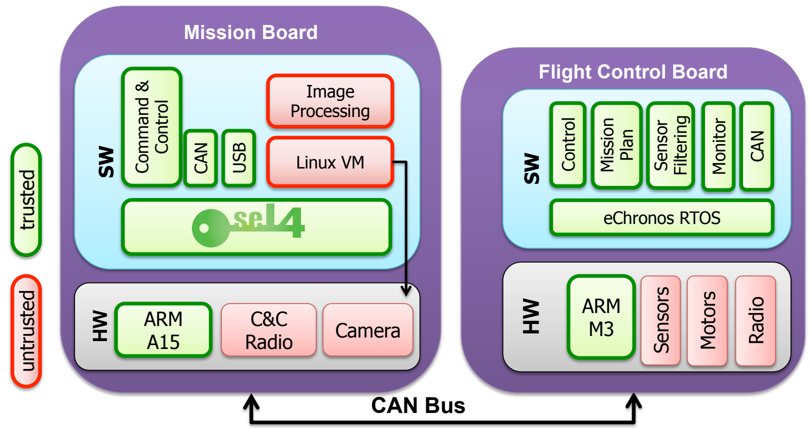


OS Support For Mixed Criticality

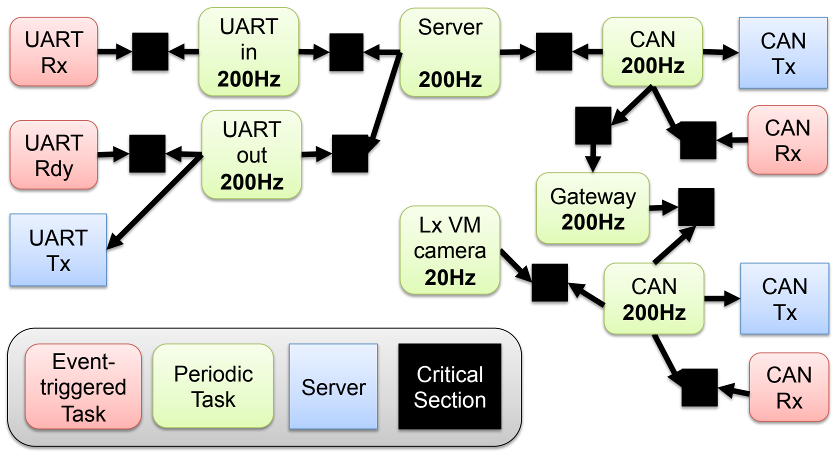
- Spatial isolation: for memory protection, certification independence
- Temporal isolation: enforce CPU time limits
  - WCET or budget
- Criticality notion:
  - Get out of jail if **HIGH** overruns optimistic budget
  - Some form of priority/deadline/budget adjustment
  - Must be fast, as the cost of change must be included in analysis!
- Support for sharing/communication
  - Why?

SMACCMcopter Drone

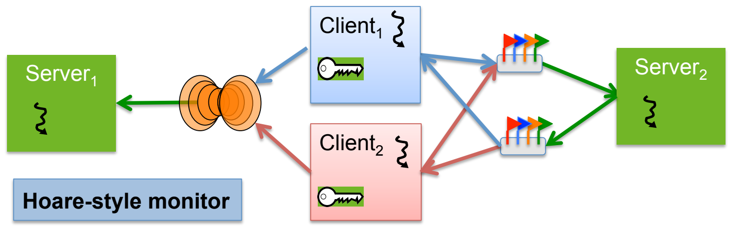




SMACCMcopter Mission Computer Architecture



Sharing: Critical Sections as Servers

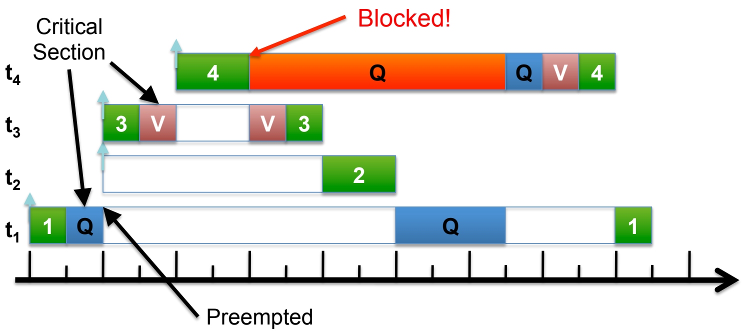



```
serv_1() {
  …
  wait(ep);
  while (1) {
    /* critical section */
    Reply&wayt(ep);
  }
}
```

```
client() {
  while (1) {
    …
    call(ep);
    …
    signal(eap_ry);
    …
    wait(eap_rq);
  }
}
```

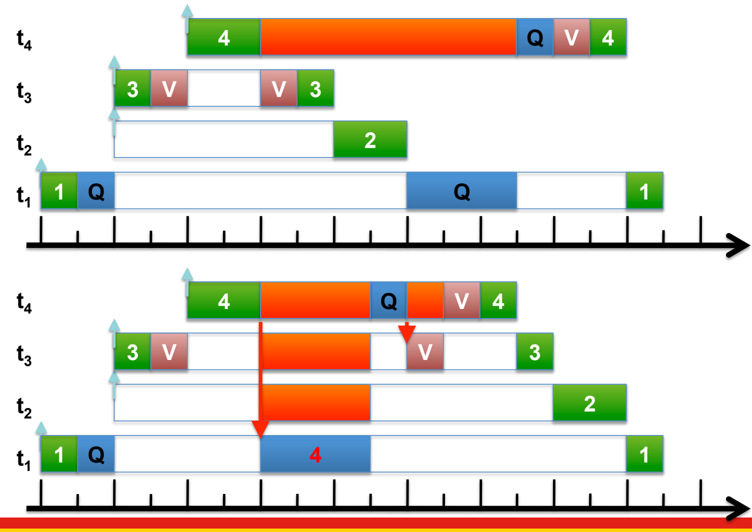
```
serv_2() {
  …
  while (1) {
    wait(eap_rq);
    /* critical section */
    signal(eap_ry);
  }
}
```


Problem: Priority Inversion



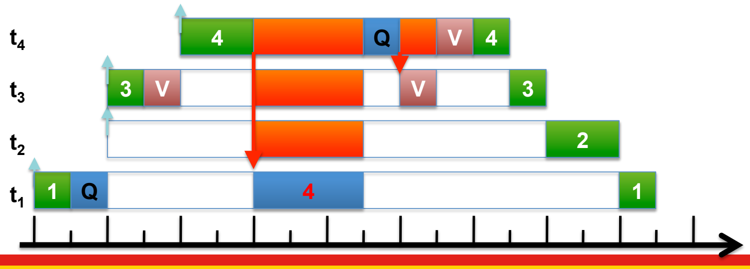
- High-priority job is blocked for a long time by a low-prio job
- Long wait chain: $t_1 \rightarrow t_4 \rightarrow t_3 \rightarrow t_2$
- Worst-case blocking time of t_1 bounded only by WCET of $C_2+C_3+C_4$
- Must find a way to do better!

Priority Inheritance ("Helping")



Priority Inheritance

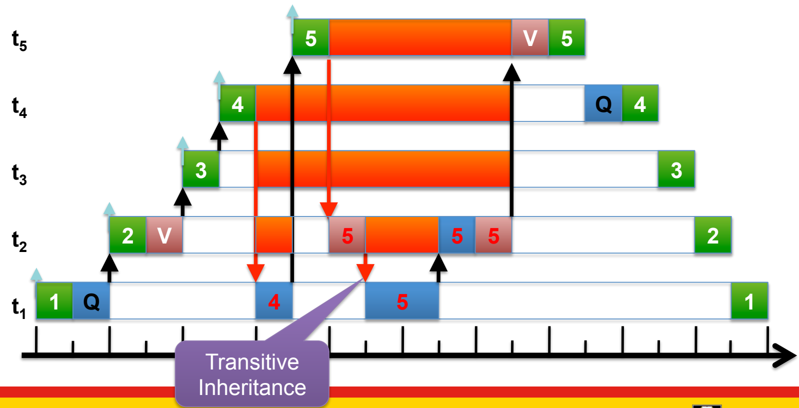
- If t_1 blocks on a resource held by t_2, *and* $P_1>P_2$, then
  - t_2 is temporarily given priority P_1
  - when t_t releases the resource, its priority reverts to P_2



  UNSW

Priority Inheritance

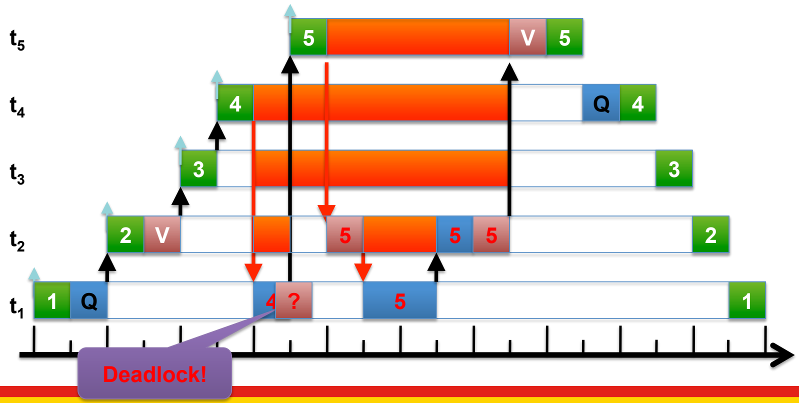
- If t_1 blocks on a resource held by t_2, *and* $P_1>P_2$, then
  - t_2 is temporarily given priority P_1
  - when t_t releases the resource, its priority reverts to P_2



  UNSW

Priority Inheritance

- 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_t releases the resource, its priority reverts to P_2



  UNSW

Priority Inheritance Protocol (PIP)

- 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_t releases the resource, its priority reverts to P_2
- Transitive inheritance
  - potentially long blocking chains
  - potential for deadlock
- Frequently blocks much longer than necessary

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

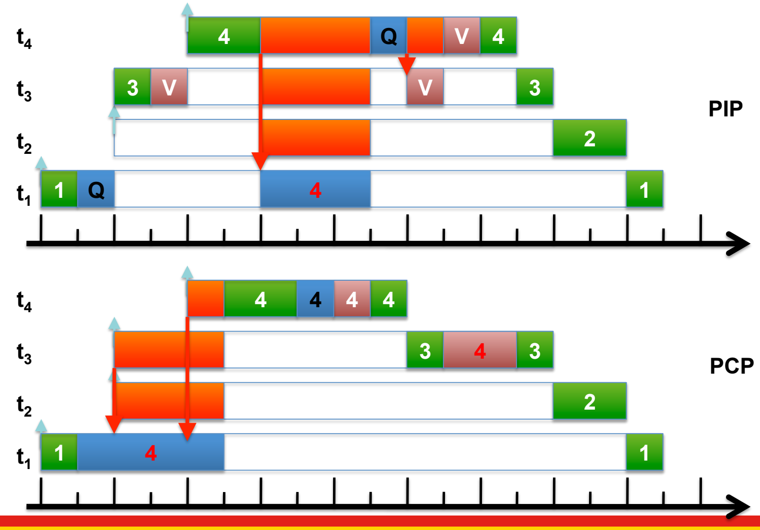
  UNSW

Priority Ceiling Protocol (PCP)

- Purpose: ensure job can block at most once on a resource
  - avoid transitivity, potential for deadlocks
- Idea: associate a *ceiling priority* with each resource
  - equal to the highest priority of jobs that may use the resource
  - when job accesses its resource, immediately bump prio to ceiling!
- Also called:
  - *immediate ceiling priority protocol* (ICPP)
  - *ceiling priority protocol* (CPP)
  - *stack-based priority-ceiling protocol*
    - because it allows running all jobs on the same stack (i.e. thread)
- Improved version of the *original ceiling priority protocol* (OCPP)
  - … which is also called the *basic priority ceiling protocol*
  - Requires global tracking of ceiling prios

  UNSW

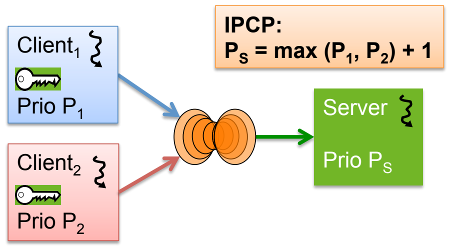
(Immediate) Priority Ceiling Protocol



  UNSW

IPCP Implementation

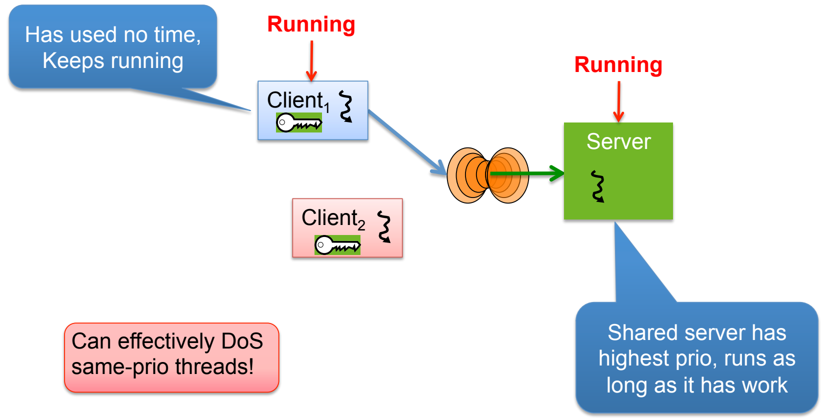
- Each task must declare all resources at admission time
  - System must maintain list of tasks associated with resource
  - Priority ceiling derived from this list
  - For EDF the "ceiling" is the *floor of relative deadlines*
- seL4: "resource declaration" is implicit in capability distribution
  - Using critical section requires cap for server's request endpoint



Priority Ceiling:
- Requires correct priority configuration
- Deadlock-free
- Easy to implement
- Good worst-case blocking times

  UNSW

Problem With Servers As Threads



  UNSW

Separate Scheduling Properties from Thread

Classical Thread Attributes

- Priority
- Time slice

Not runnable if null

New Thread Attributes

- Priority
- Scheduling context capability

Scheduling context object
- T: period
- C: budget (≤ T)

Upper bound, not reservation!

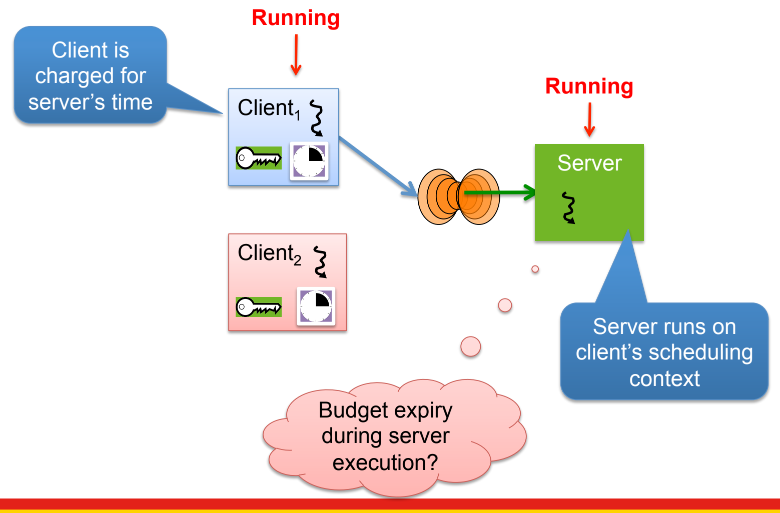
Not yet in mainline!

C = 2
T = 3

C = 250
T = 1000

`SchedControl` capability conveys right to assign budgets (i.e. perform admission control)

Shared Server with Scheduling Contexts



Budget Expiry Options

- Multi-threaded servers (COMPOSITE [Parmer '10])
  - Model allows this
  - Forcing all servers to be thread-safe is policy 😢
- Bandwidth inheritance with "helping" (Fiasco [Steinberg '10])
  - Ugly dependency chains 😢
  - Wrong thread charged for recovery cost 😢
- Use *timeout exceptions* to trigger one of several possible actions:
  - Provide emergency budget
  - Cancel operation & roll-back server
  - Change criticality
  - Implement priority inheritance (if you must…)