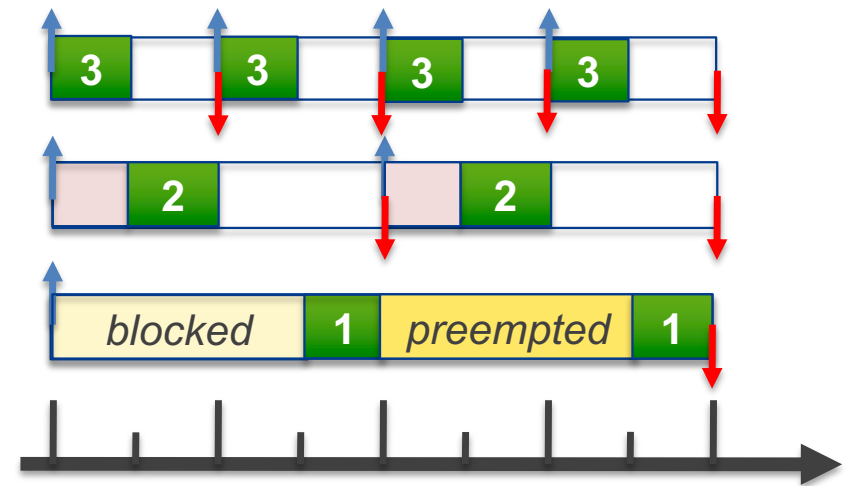


2026 T3 Week 03 Part 2

Real-Time Systems Basics

@GernotHeiser

Presented by Anna Lyons



Copyright Notice

These slides are distributed under the Creative Commons Attribution 4.0 International (CC BY 4.0) 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/4.0/legalcode>

Today's Lecture

- Real-time systems (RTS) basics
 - Types of RTS
 - Basic concepts & facts
- Resource sharing in RTS
- Scheduling overloaded RTS
- Mixed-criticality systems (MCS)

Presented by Dr Anna Lyons

Work

- 2022-26, secure kernel team @ Apple
- 2019-22, platform team @ Ghost
- 2010-18 Research Engineer @ Trustworthy Systems
- 2007-2018 Tutor - OS, AOS, COMP19**
- 2010 summer intern @ Microsoft - Bing
- 2008-10 Part-time @ Atlassian
- 2007 summer ToR @ NICTA 2007-08

Education

- 2012-2018 PhD w/ Gernot
- 2006-11 B Sci (Computer Science) / BA (Philosophy)

Presented by Dr Anna Lyons

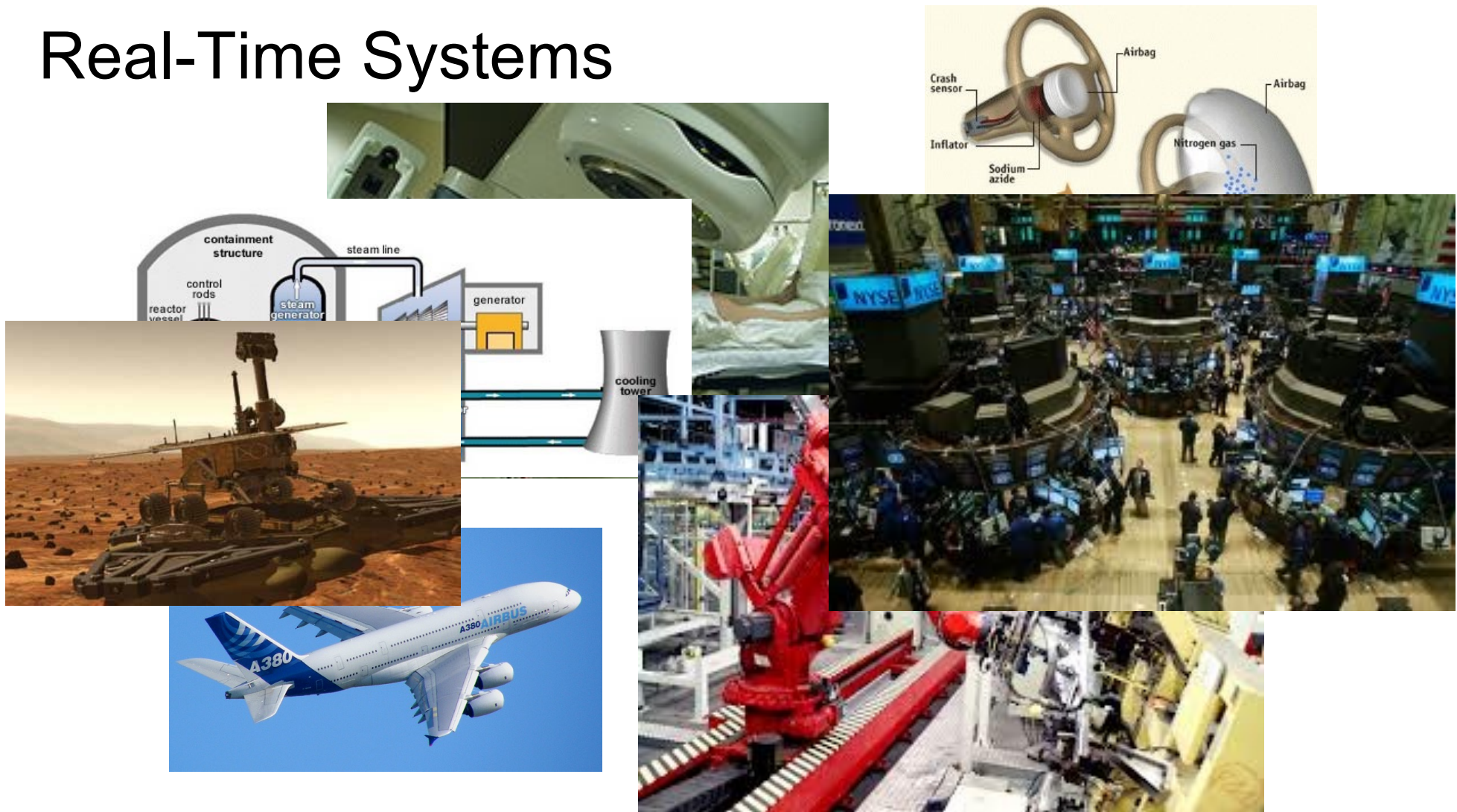
Work at Trustworthy Systems

- Initial port of AOS to seL4 w/ Adrian Danis, then aarch64 + pico tcp + nfsv3
- Shepherd AOS from nslu2 to imx6 then odroid c2
- PhD: MCS kernel extensions
- I did AOS on the slug —> w/ OKL4



Real-Time Basics

Real-Time Systems



What's a Real-Time System?

Aka. events

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?

Core Challenge: Time Is Not Fungible

Fungible: easy to exchange or trade for something else of the same type and value
[Cambridge Dictionary]

Fungible	Non Fungible
Chocolate chip cookie	Human
\$10 note	Roman coin
Memory frame	The seconds after you hit the brake

“Real Time” – Real Confusion

- “Real-time applications”

Refers to apps that react to changes anywhere in a connected application’s system— not just those made by the current user.

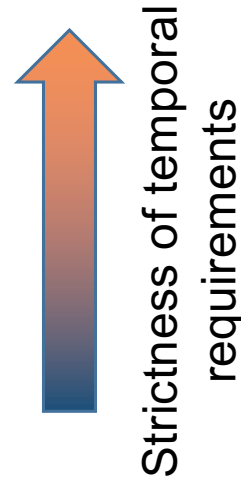
- “Real-time processing”

Refers to processing data as soon as it becomes available, as opposed to some scheduled later processing time

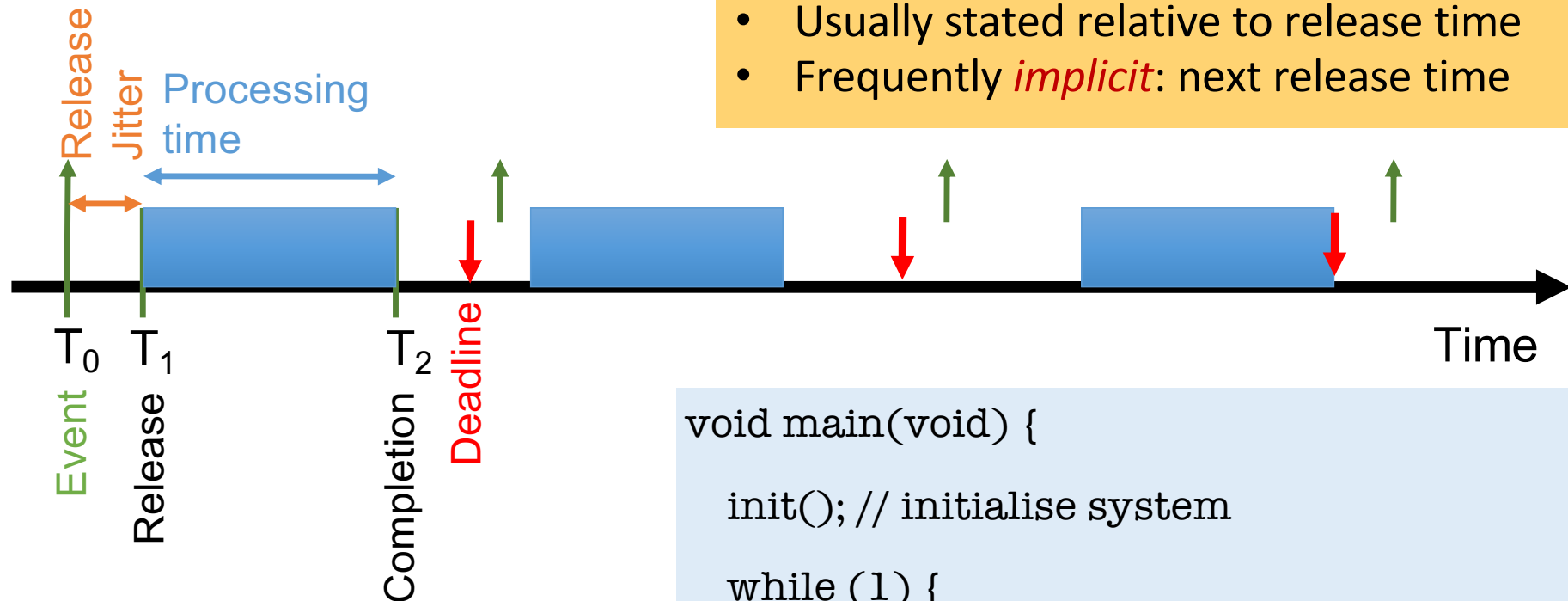
Not real-time systems!

Strictness of Temporal Requirements

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



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) {  
        wait(); // timer, device interrupt, signal  
        doJob();  
    }  
}
```

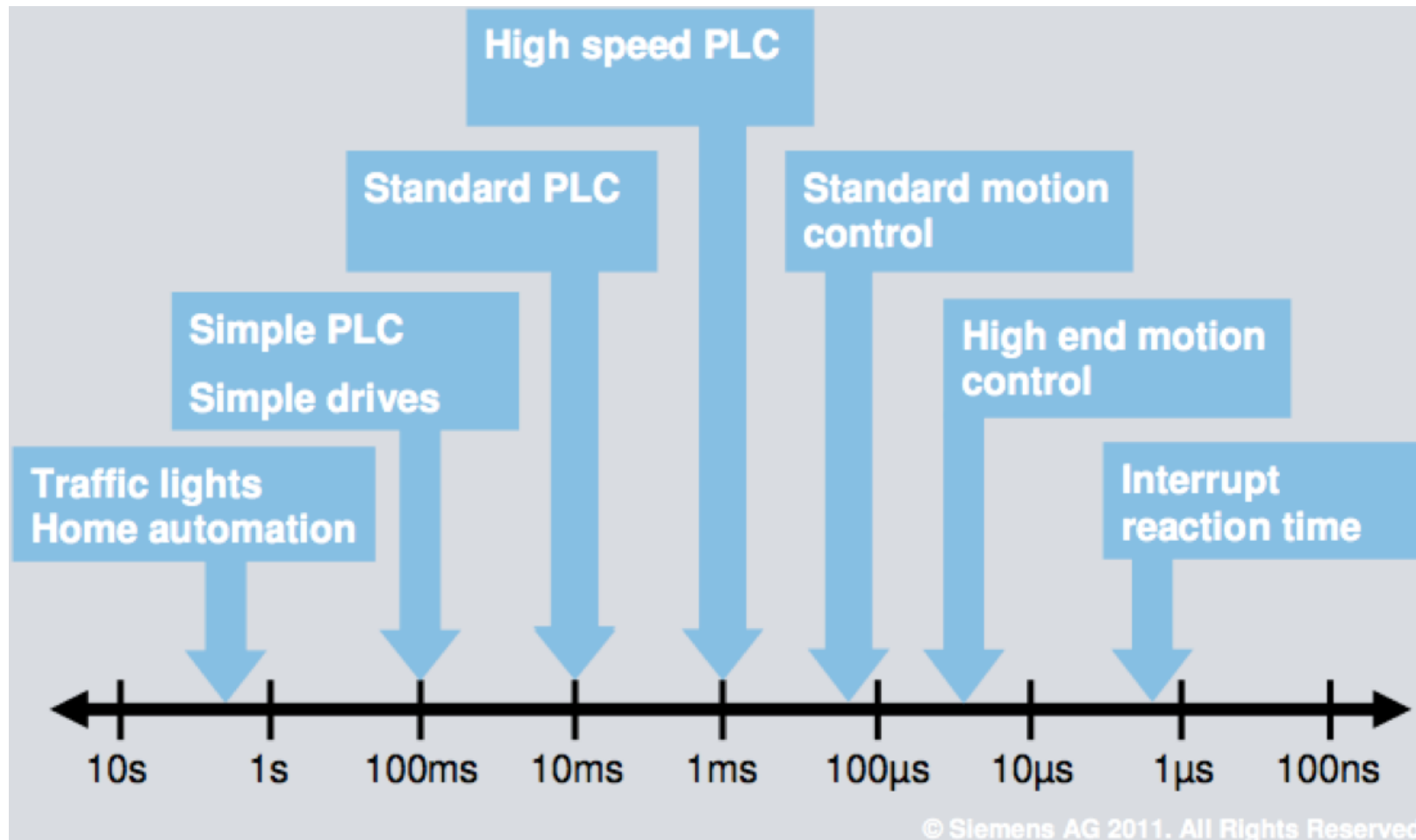
Arrows indicate that T_1 corresponds to the `wait()` call and T_2 corresponds to the `doJob()` call.

Real Time $\neq$ Real Fast

System	Deadline	Single Miss Conseq	Ultimate Conseq.
Combustion 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

Criticality

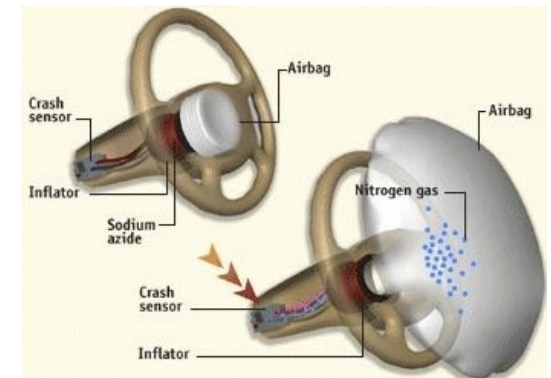
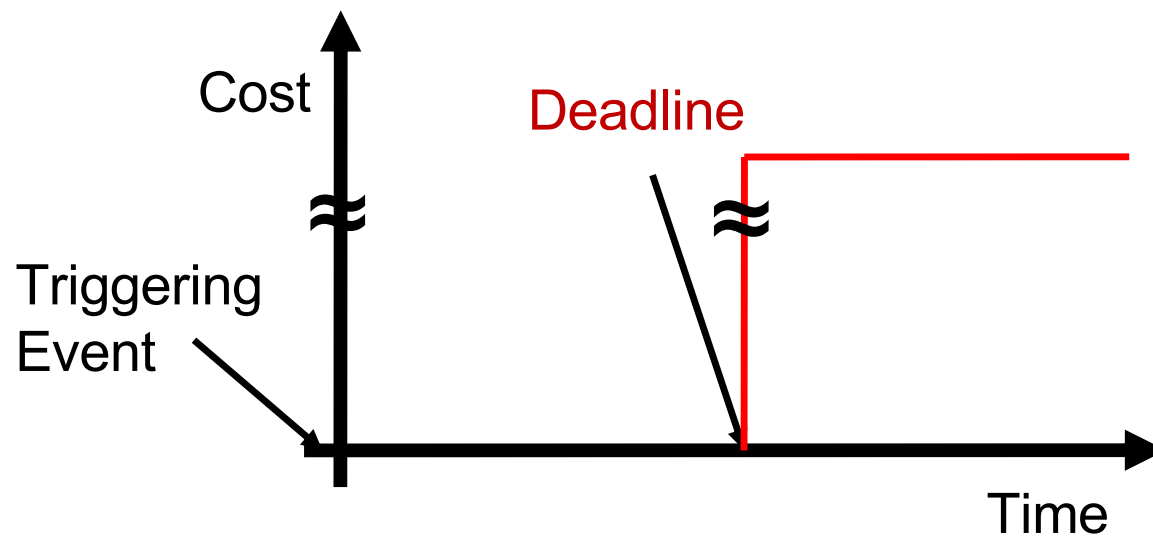
Example: Industrial Control



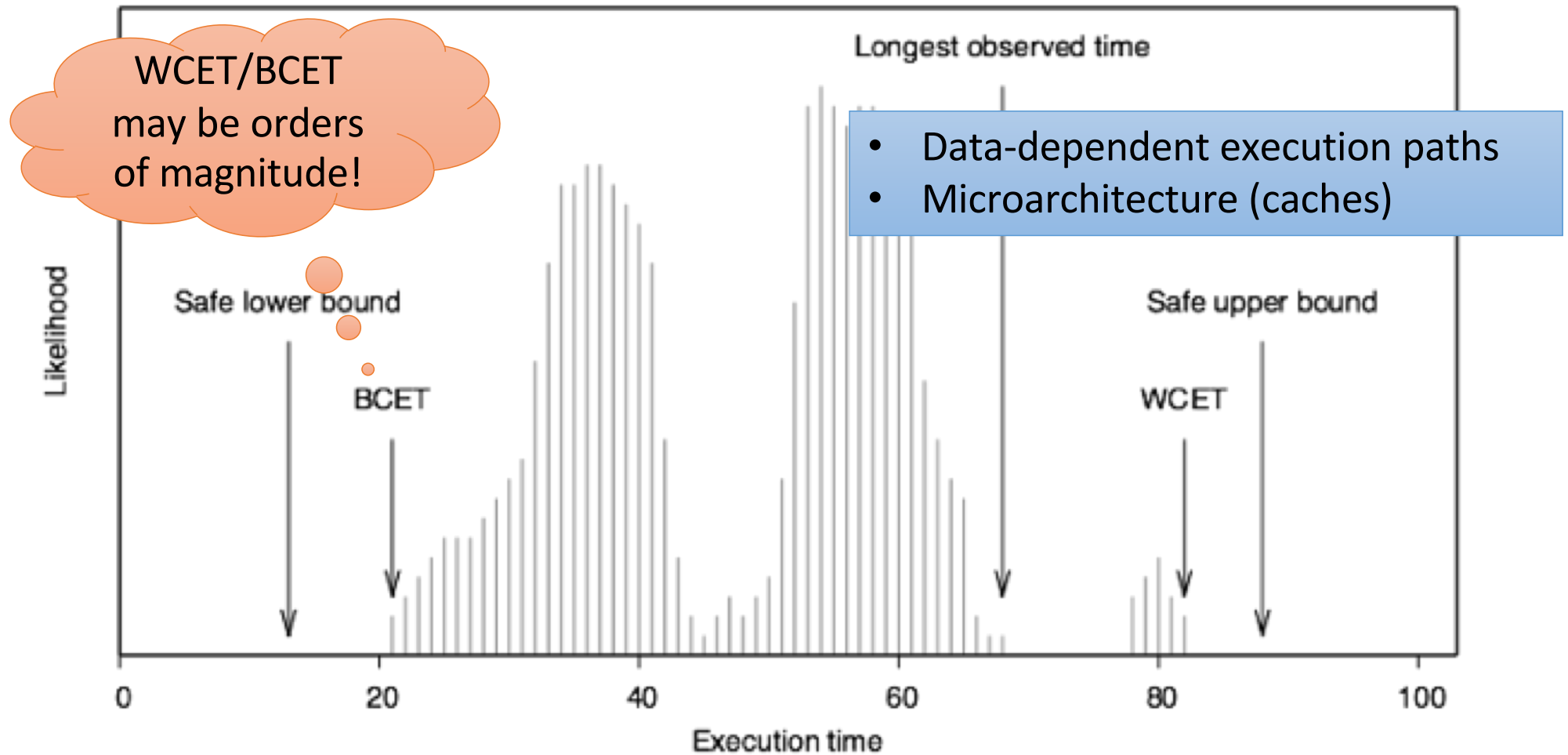
Hard Real-Time Systems

- Safety-critical: Failure $\Rightarrow$ death, serious injury
- Mission-critical: Failure $\Rightarrow$ massive financial damage

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



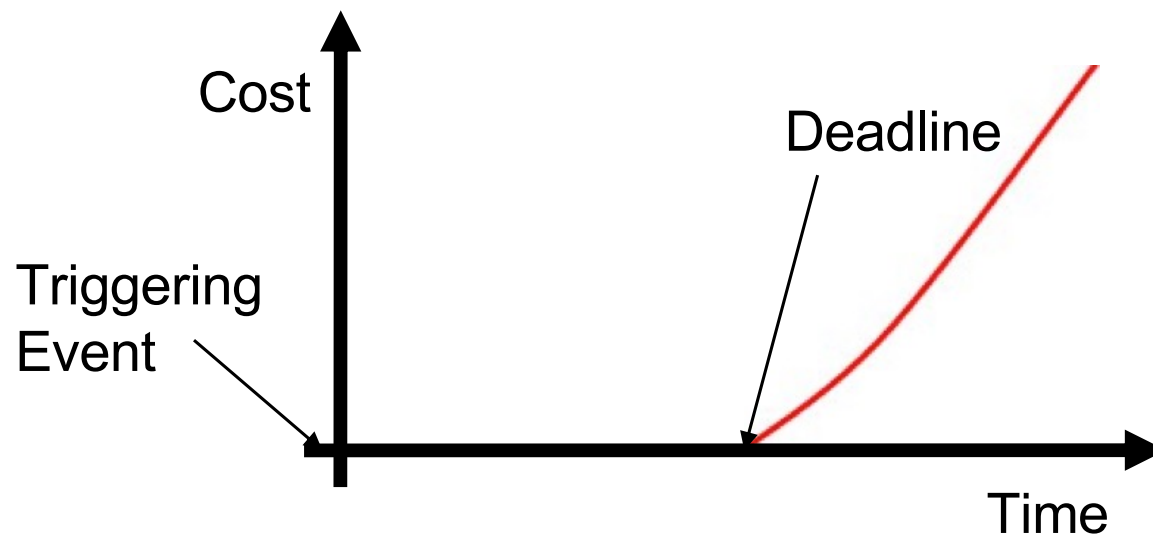
Challenge: Execution-Time Variance



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

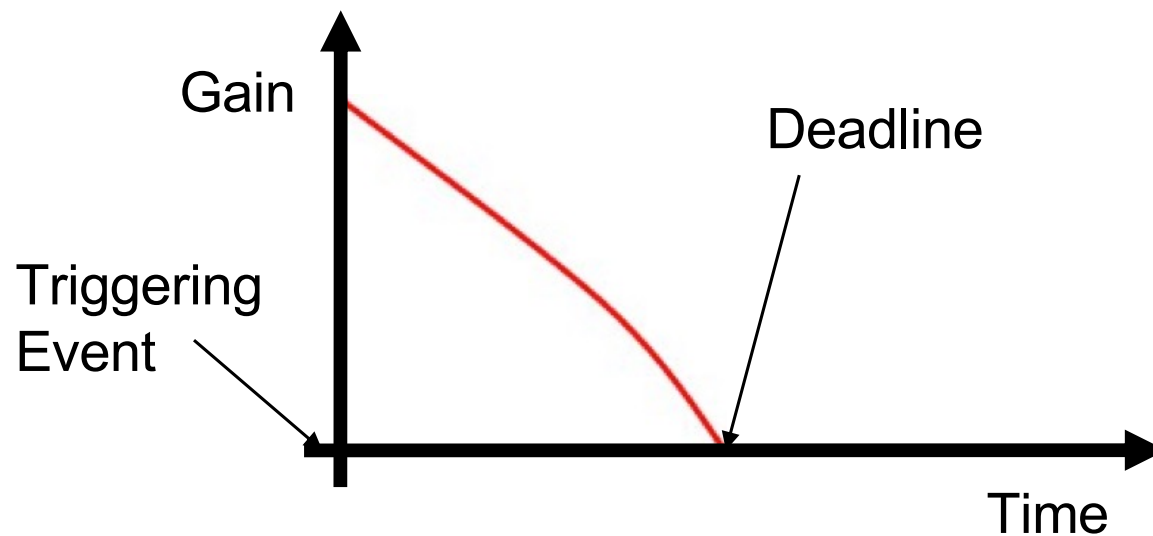


In practice, certifiers treat critical avionics as hard RT

Firm Real-Time Systems

Result obsolete if deadline missed (loss of revenue)

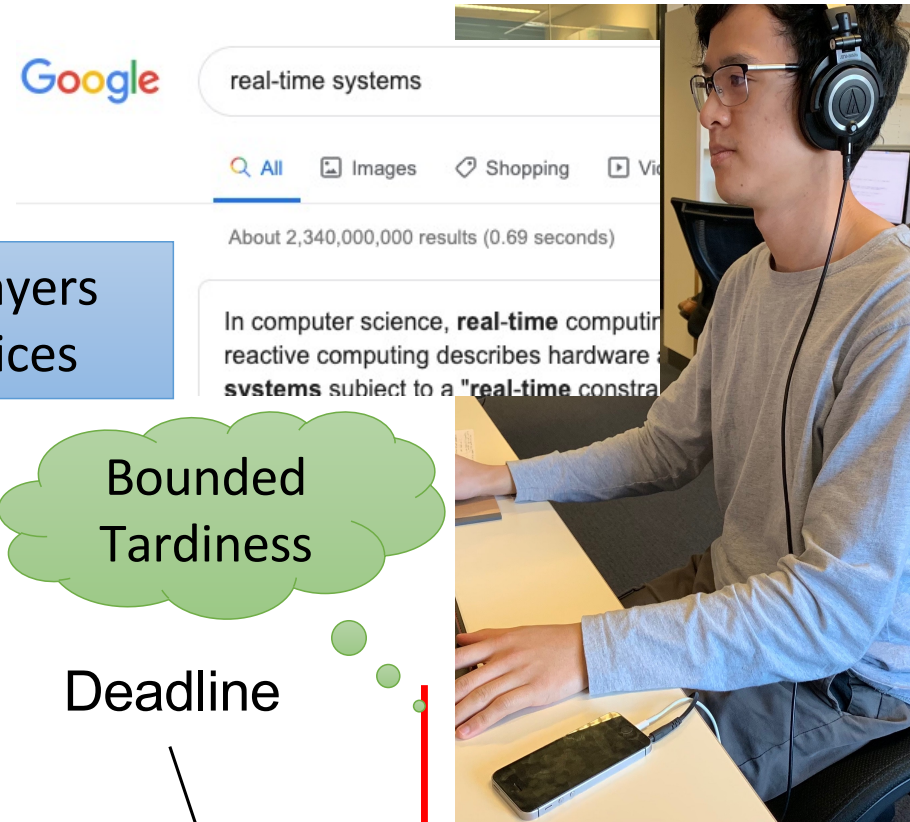
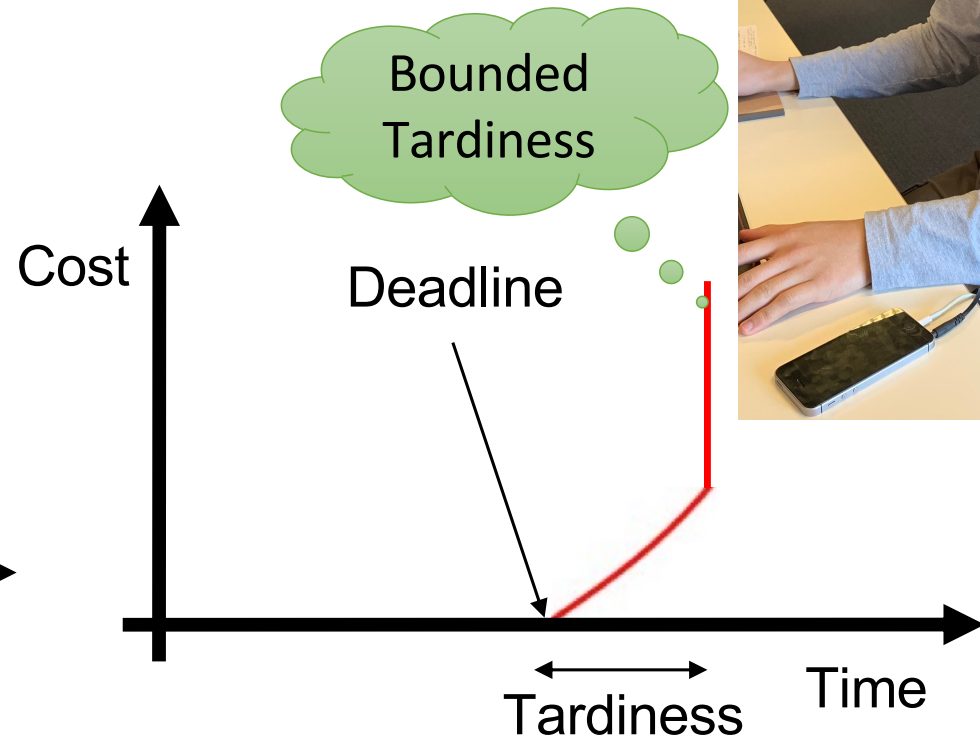
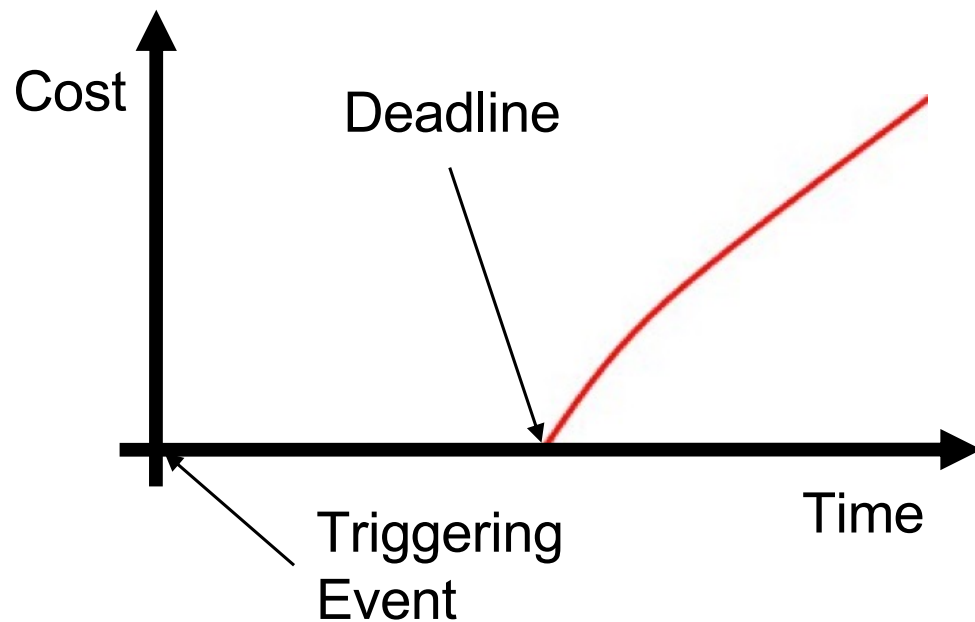
- Forecast systems
- Trading systems



Soft Real-Time Systems

Deadline miss undesirable but tolerable, affects QoS

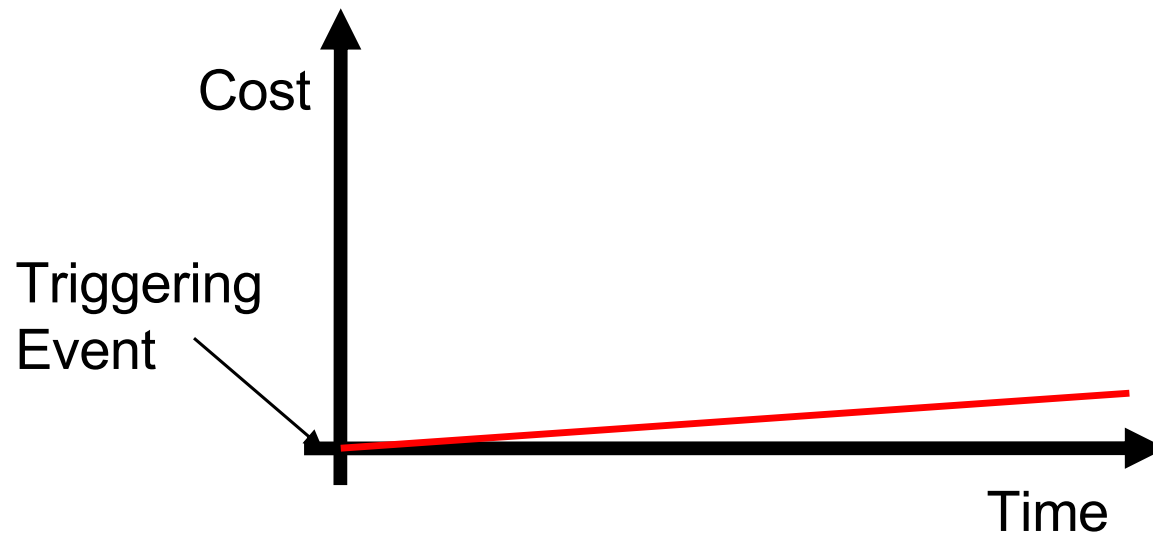
- Media players
- Web services



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

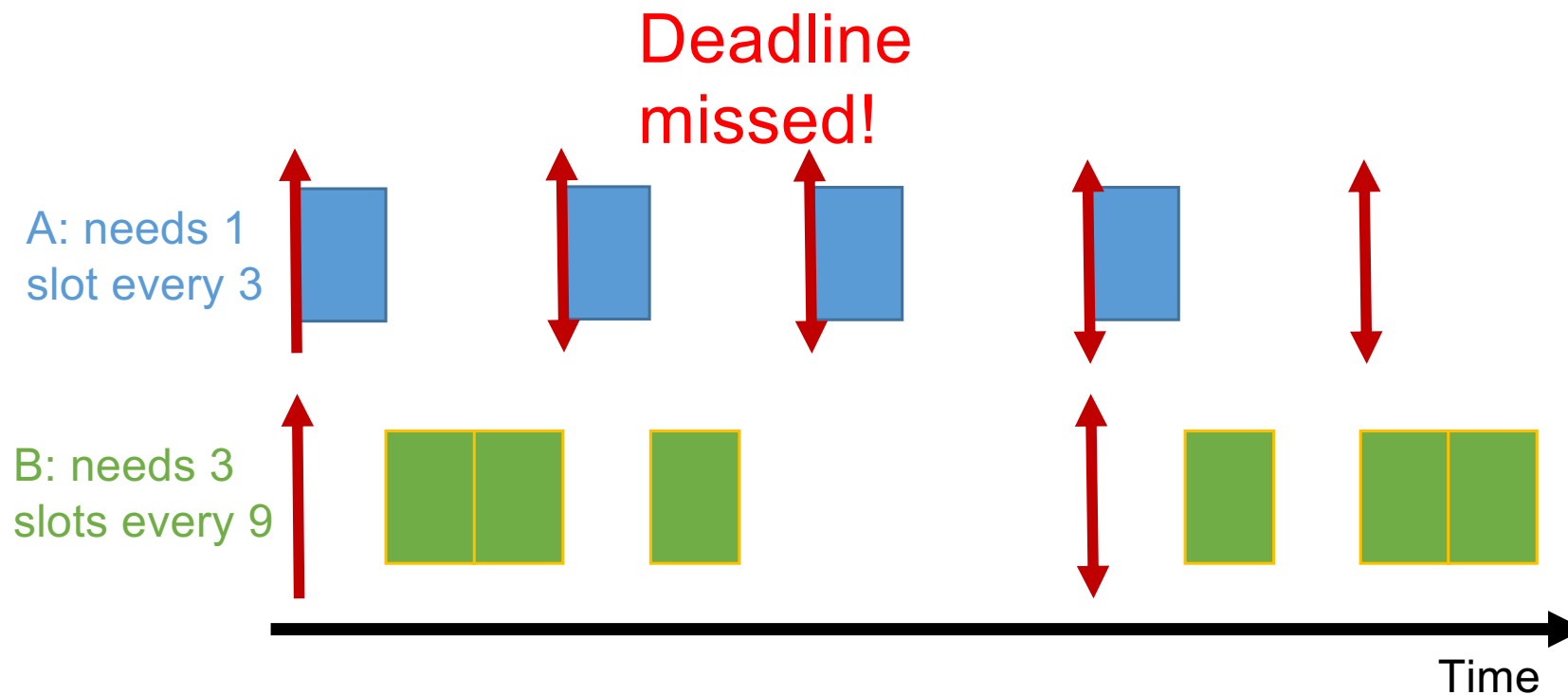
- single-mode execution
- no memory protection
- inherently cooperative
- *all code is trusted*

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

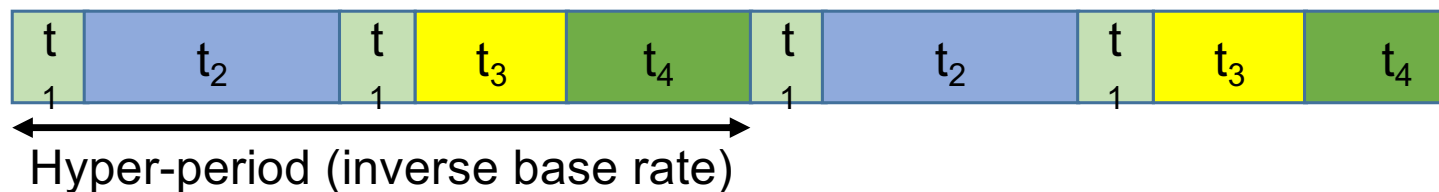
Terminology:

- 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

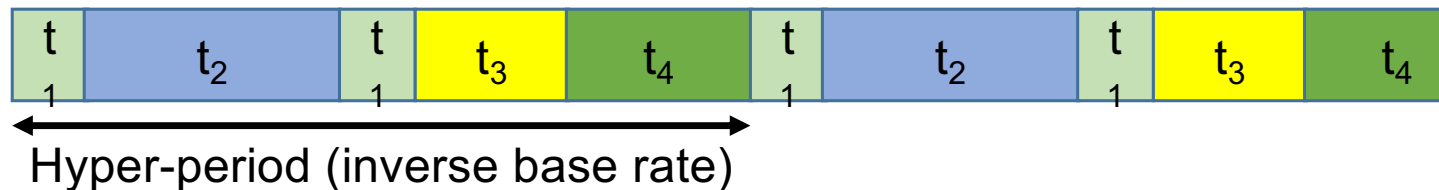
Drawback: Latency of event handling is hyper-period



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

Are Cyclic Executives Optimal?

- Theoretically yes if can slice (interleave) tasks
- Practically there are limitations:
 - Might require very fine-grained slicing (context switching)
 - 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();  
}
```

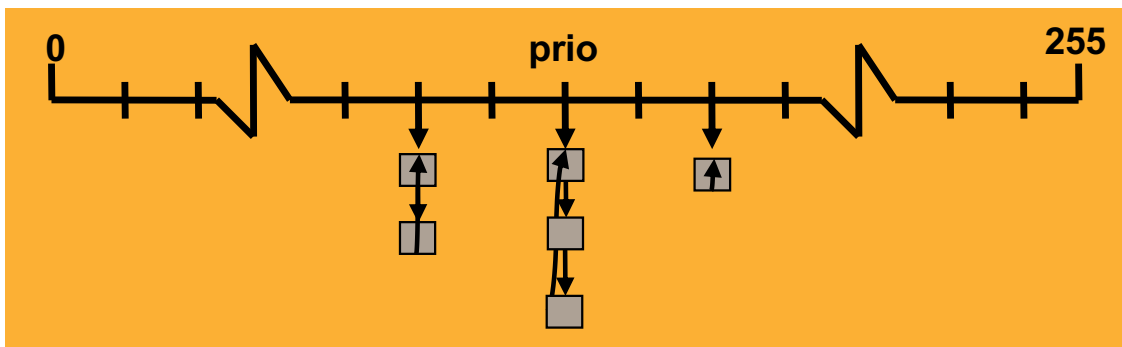
On-Line RT Scheduling

- 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

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

Rate Monotonic Priority Assignment (RMPA)

- Higher rate $\Rightarrow$ higher priority:

- $T_i < T_j \Rightarrow P_i > P_j$

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

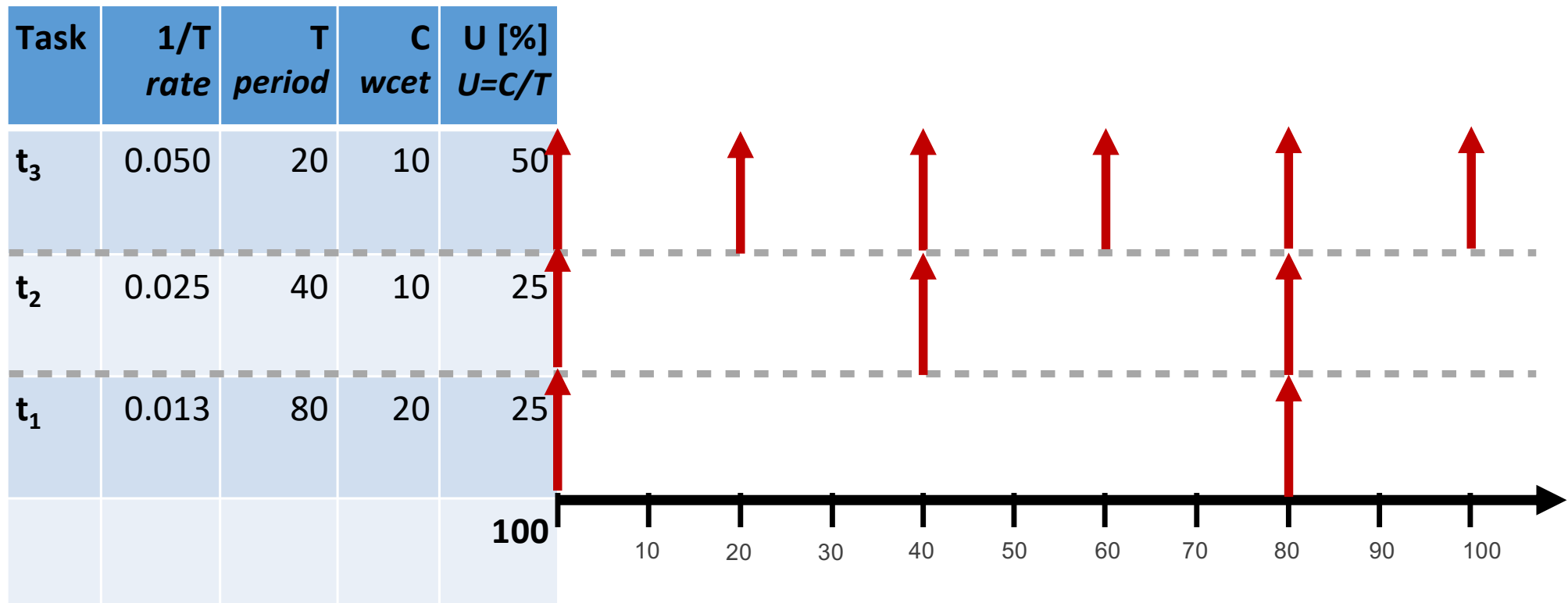
$$U \equiv \sum C_i / T_i$$

$$U \leq n(2^{1/n} - 1)$$

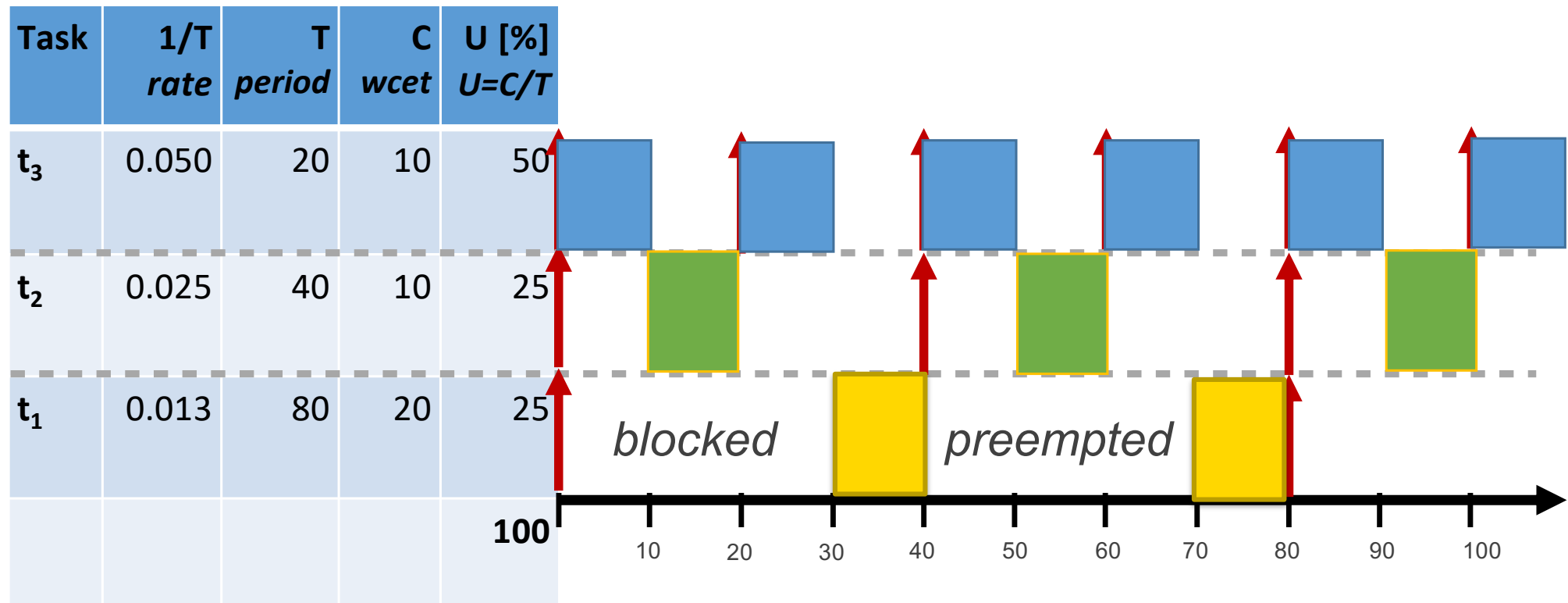
RMPA is optimal for FPS

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

Rate-Monotonic Scheduling Example

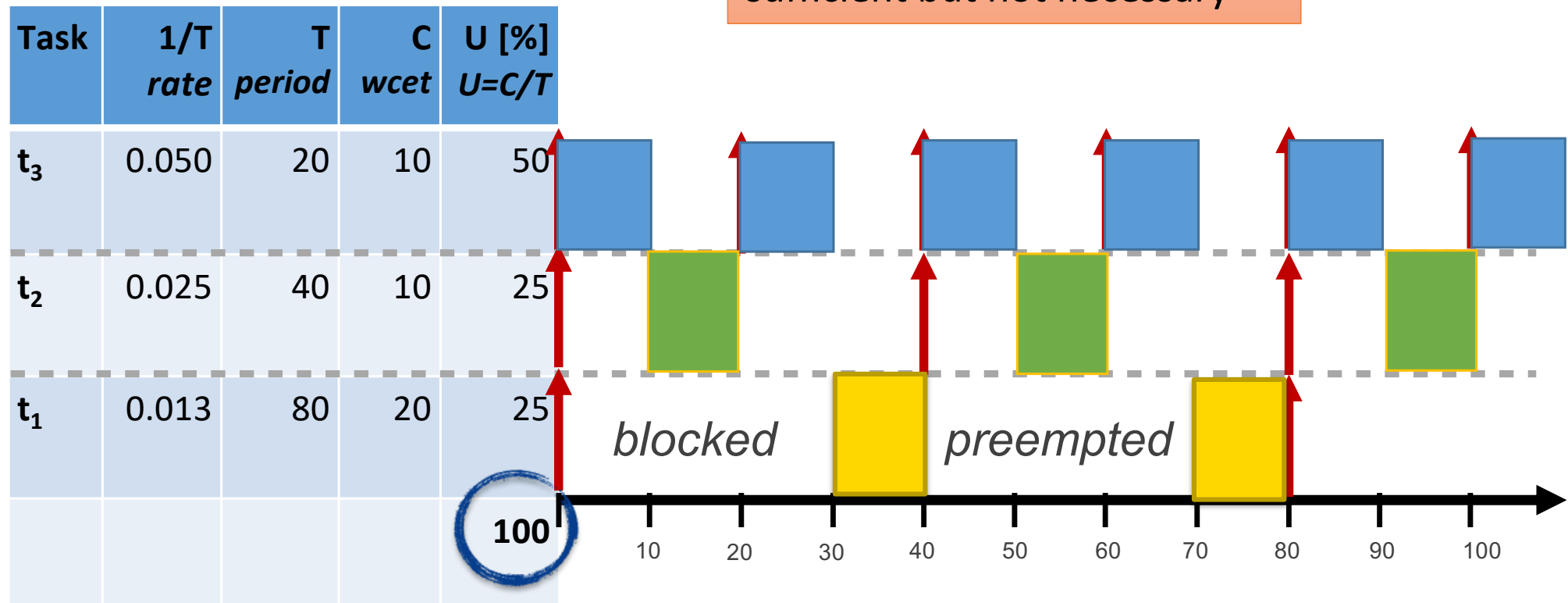


Rate-Monotonic Scheduling Example

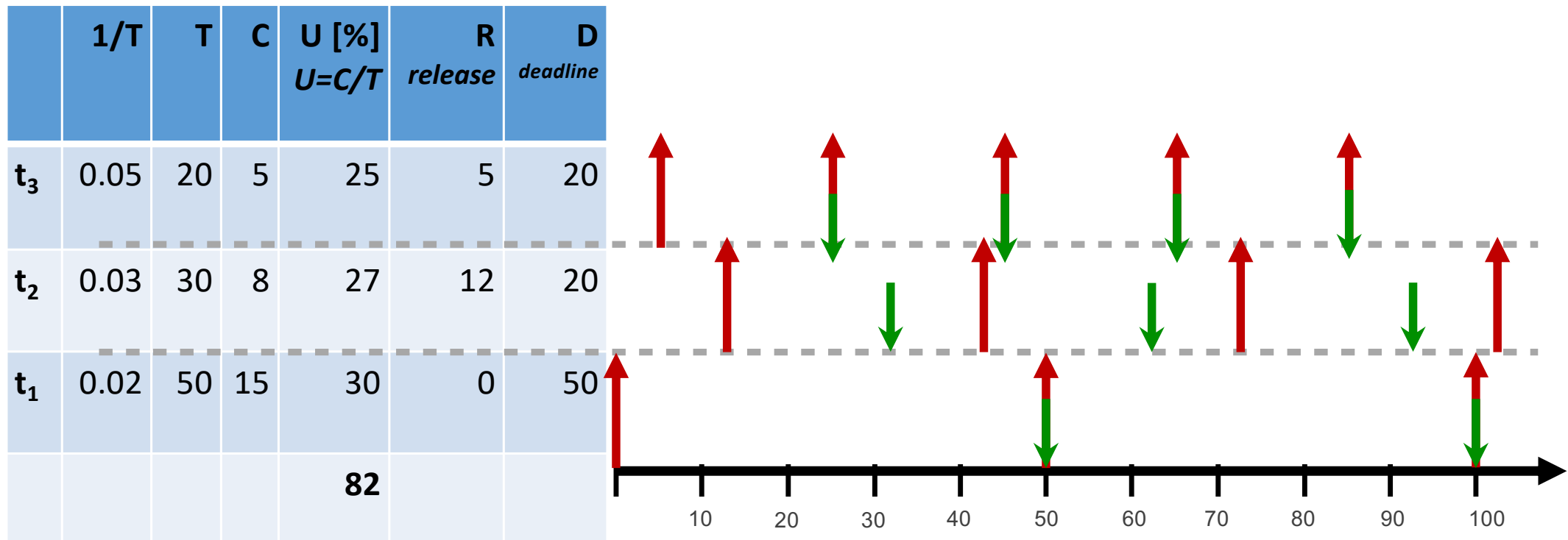


Rate-Monotonic Scheduling Example

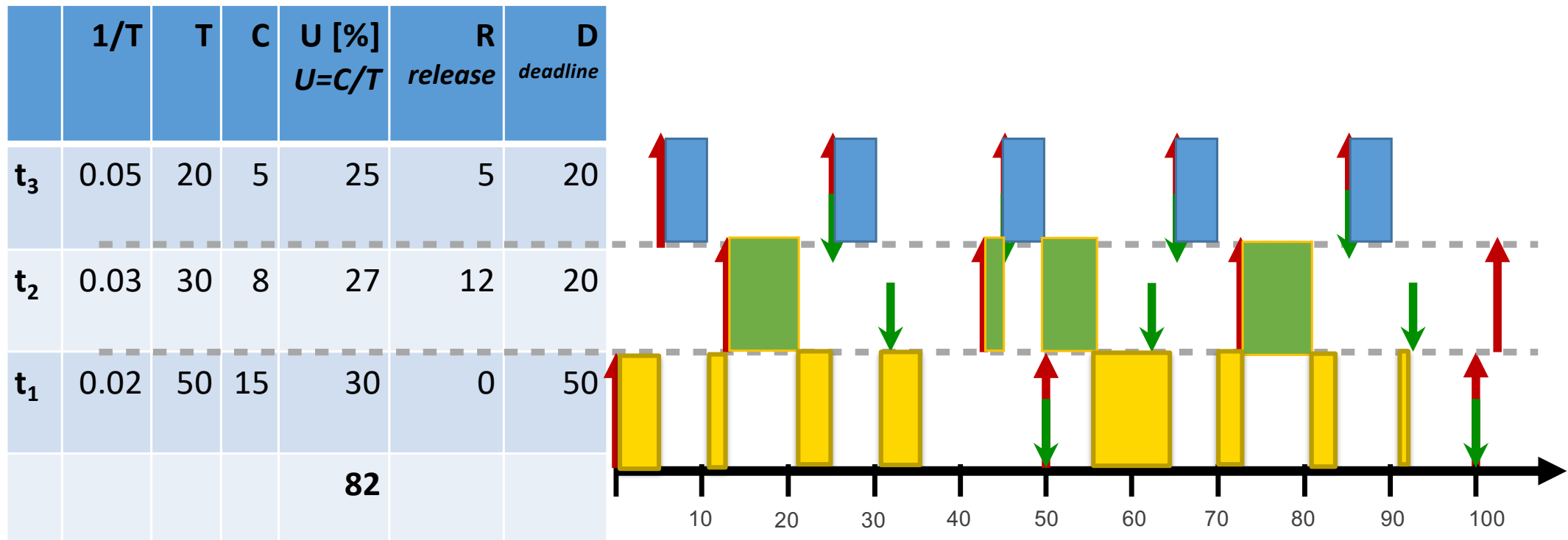
RMPA schedulability bound is sufficient but not necessary



Another RM Scheduling Example



Another RMPA Scheduling Example



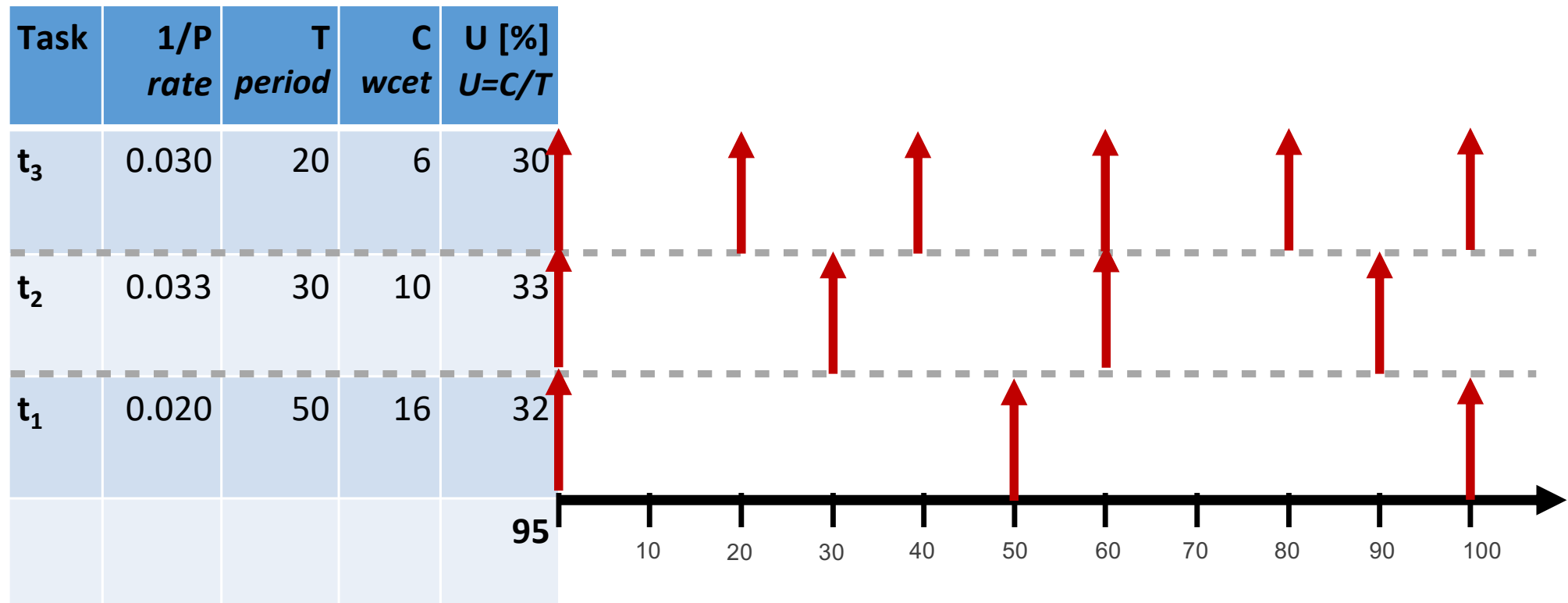
Dynamic Prio: Earliest Deadline First (EDF)

- 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 $\{T_1 \dots T_n\}$ if

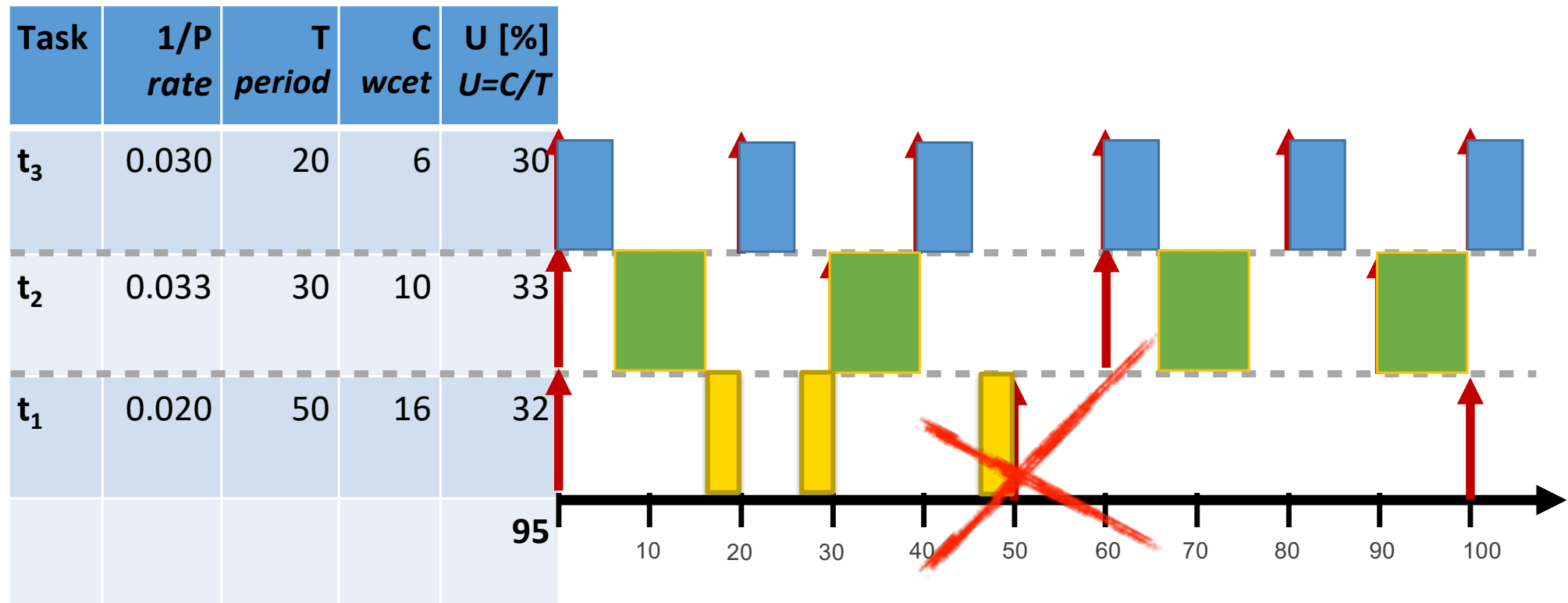
$$U \equiv \sum C_i/T_i \leq 1$$

Preemptive EDF is optimal

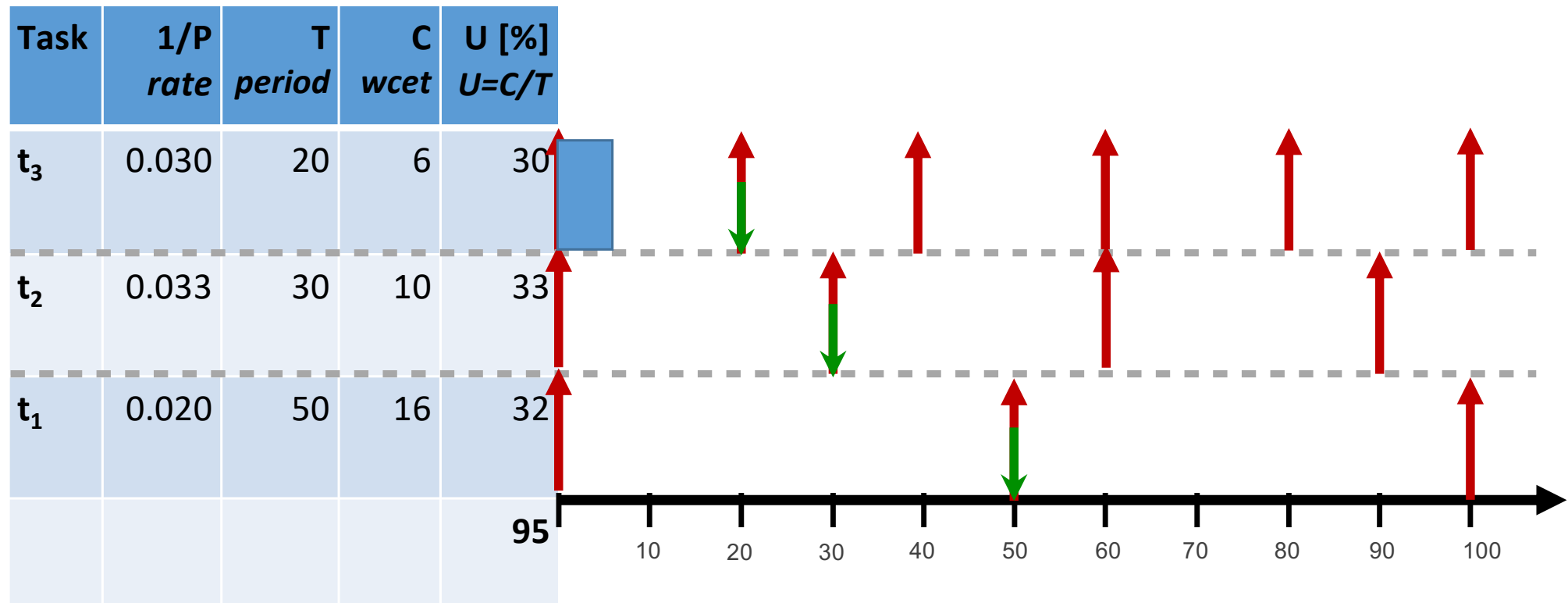
Unschedulable RM Scheduling Example



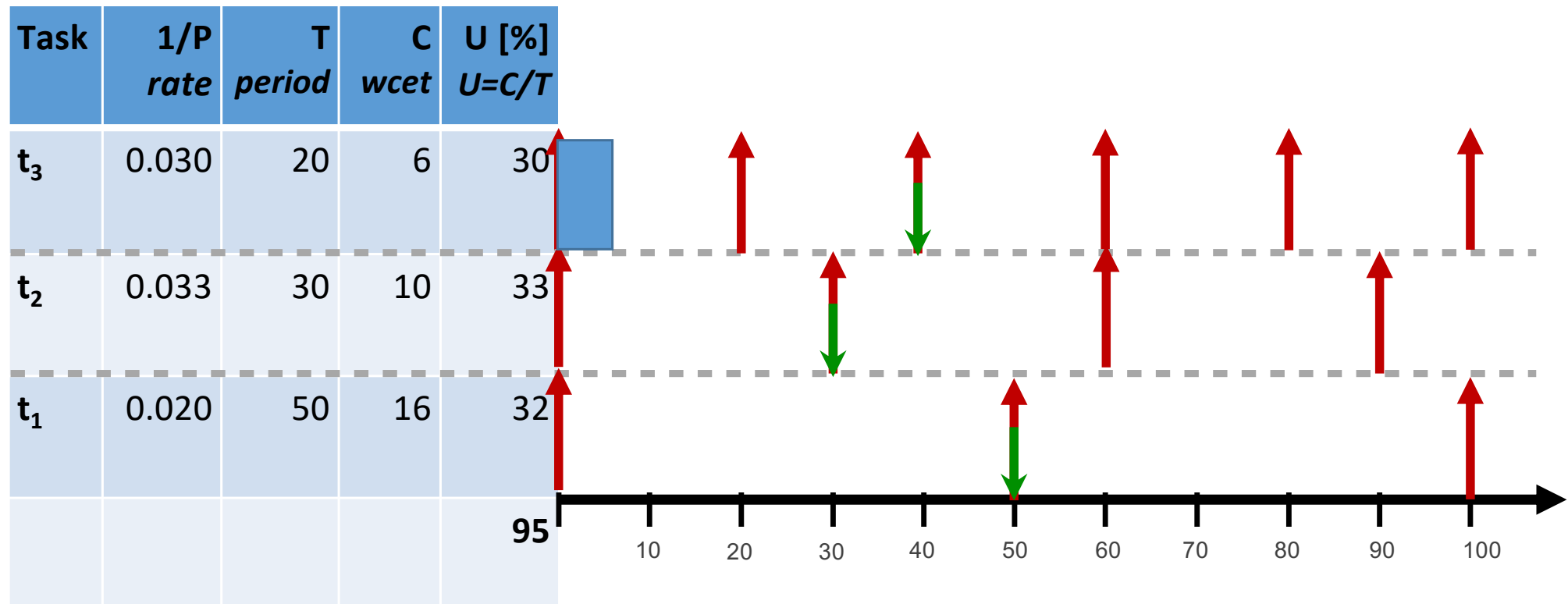
Unschedulable RM Scheduling Example



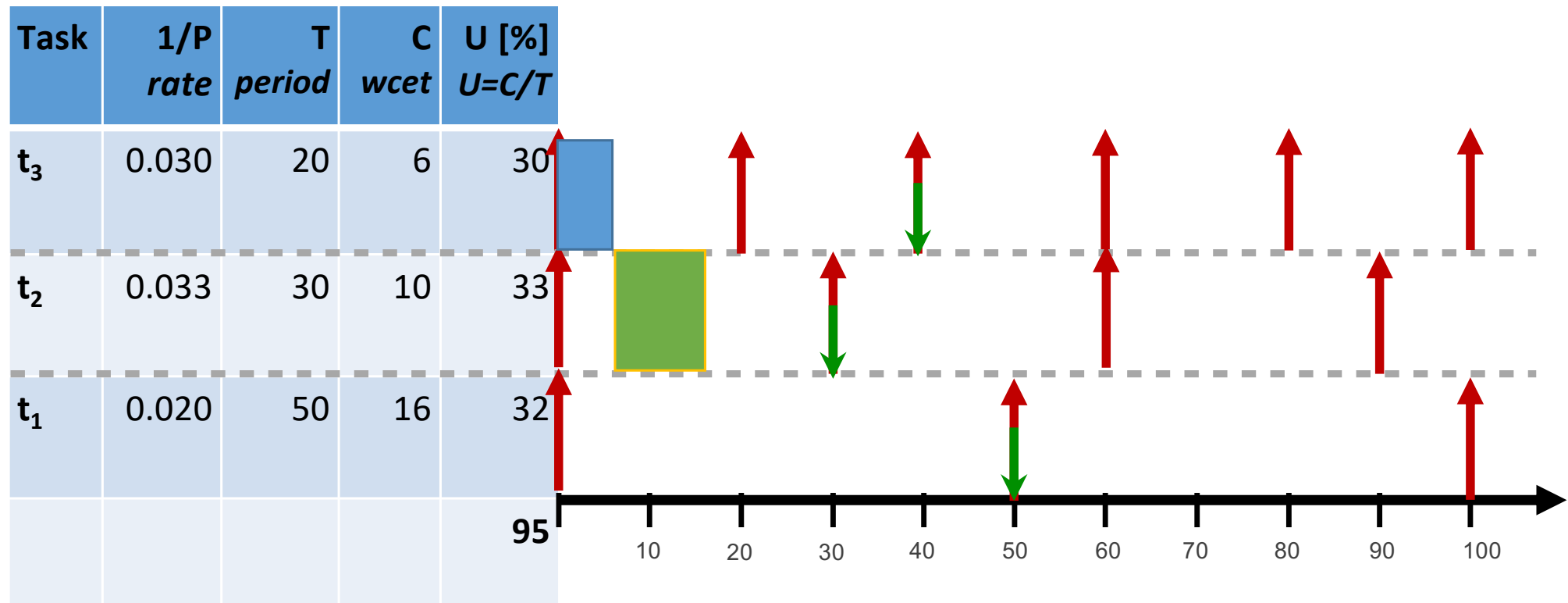
Use EDF!



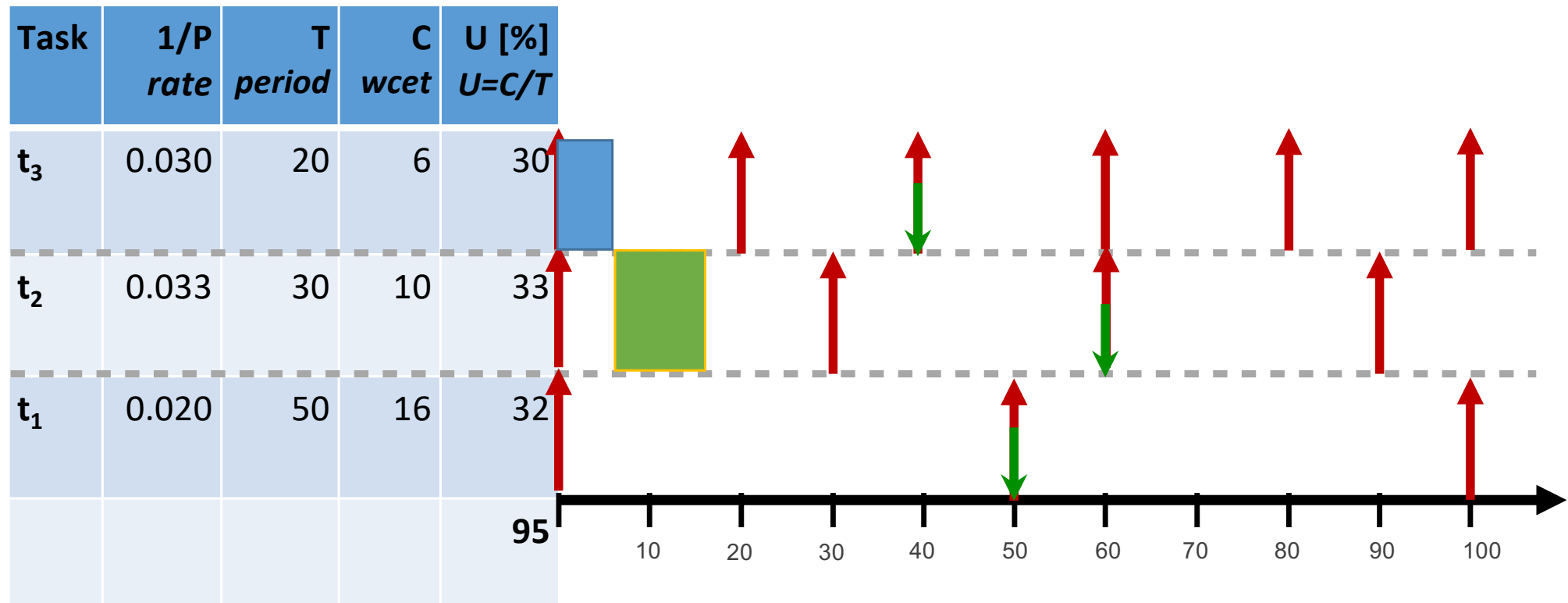
Use EDF!



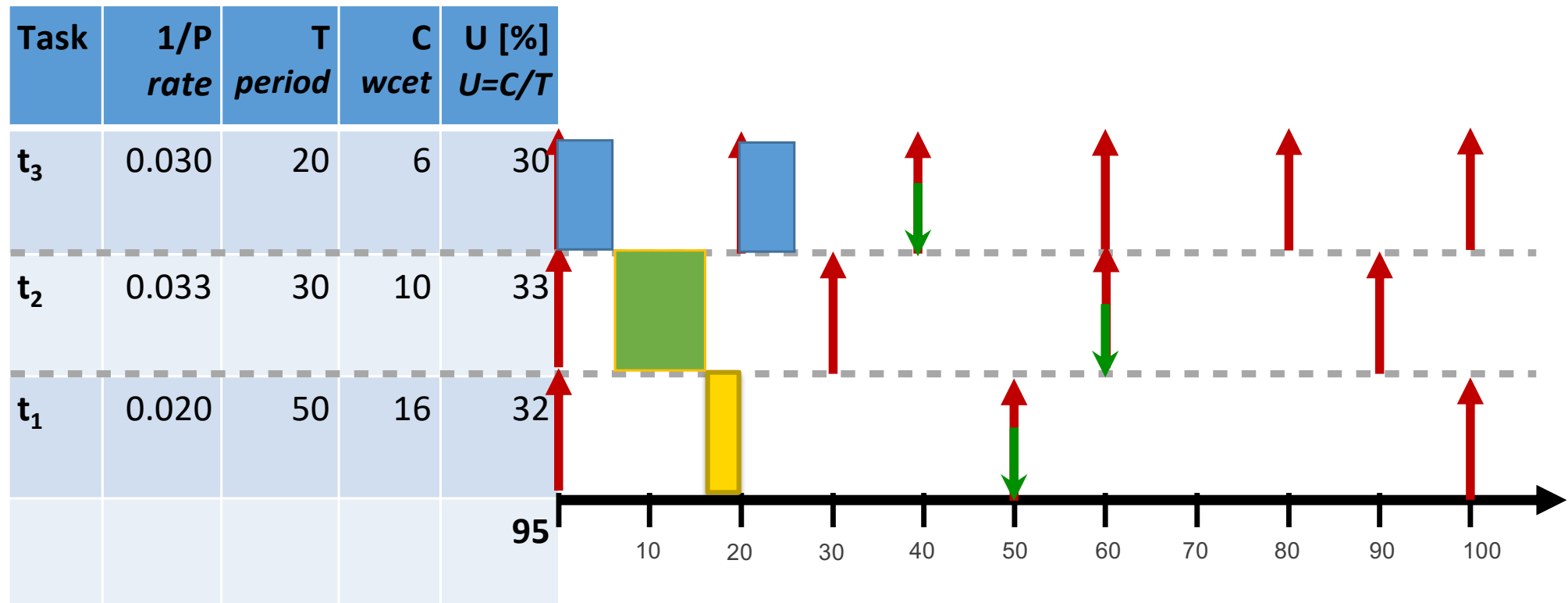
Use EDF!



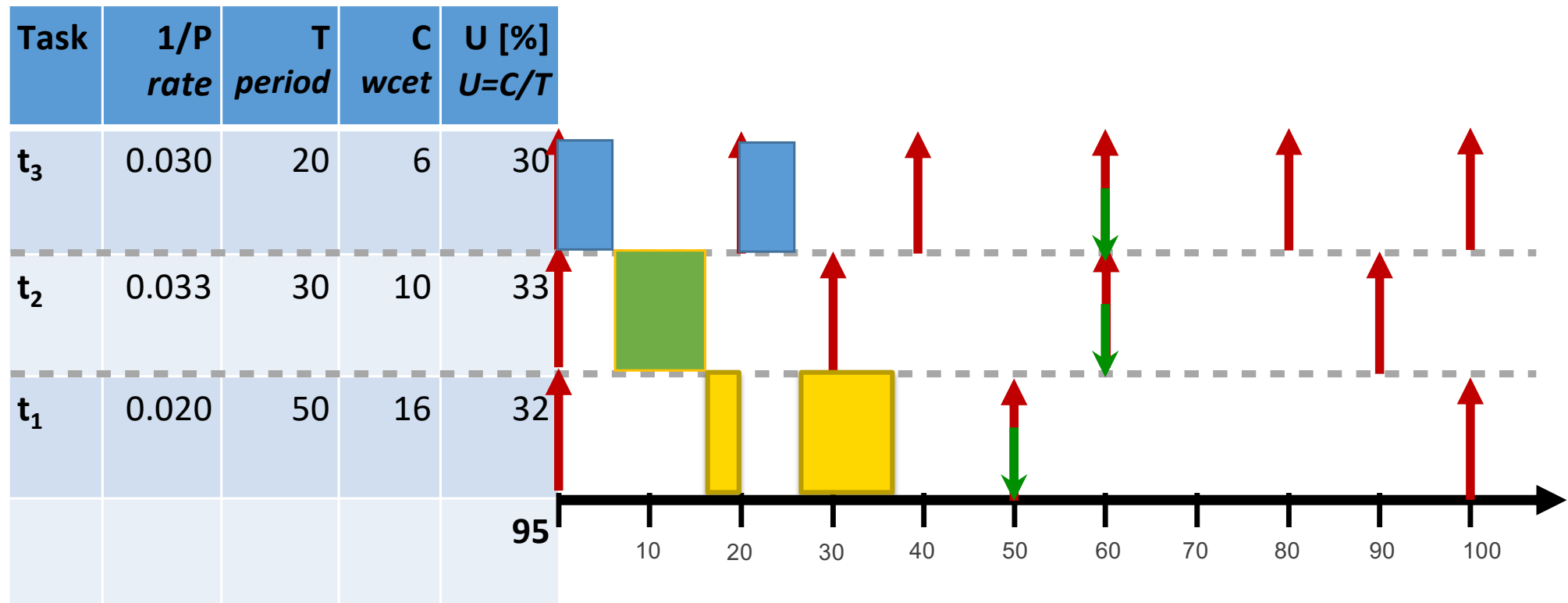
Use EDF!



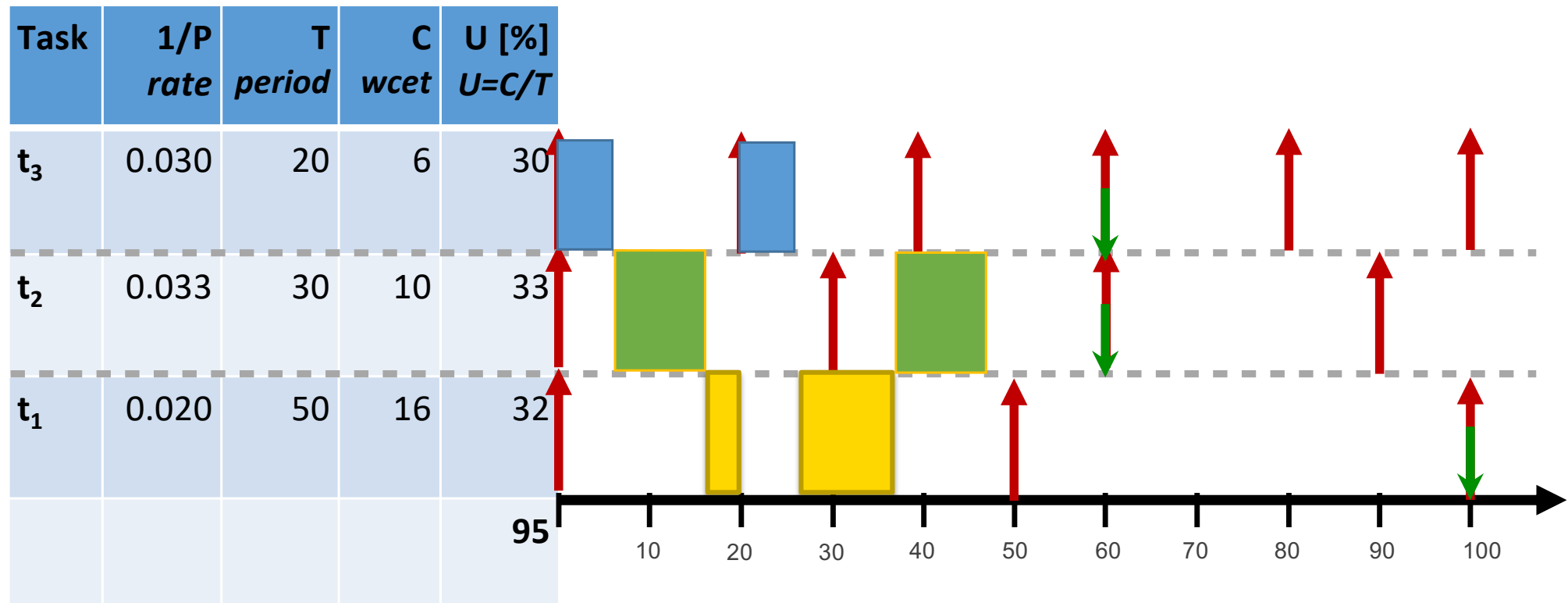
Use EDF!



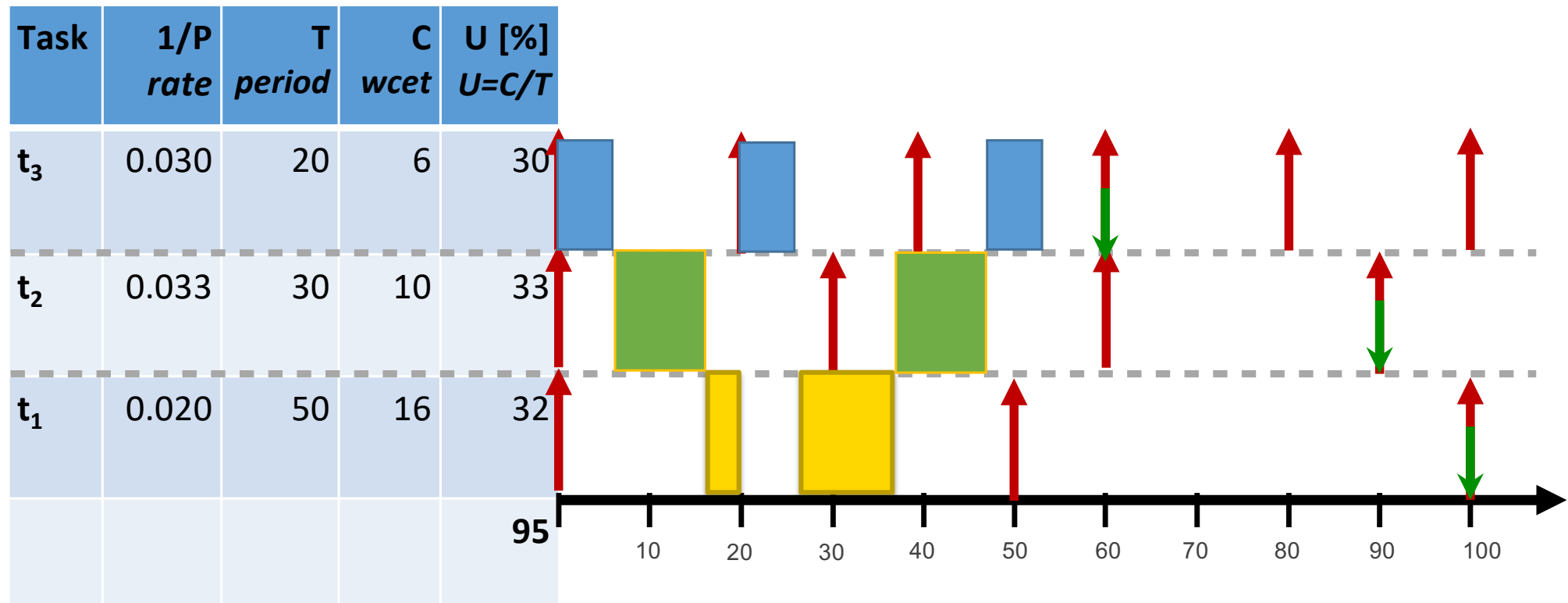
Use EDF!



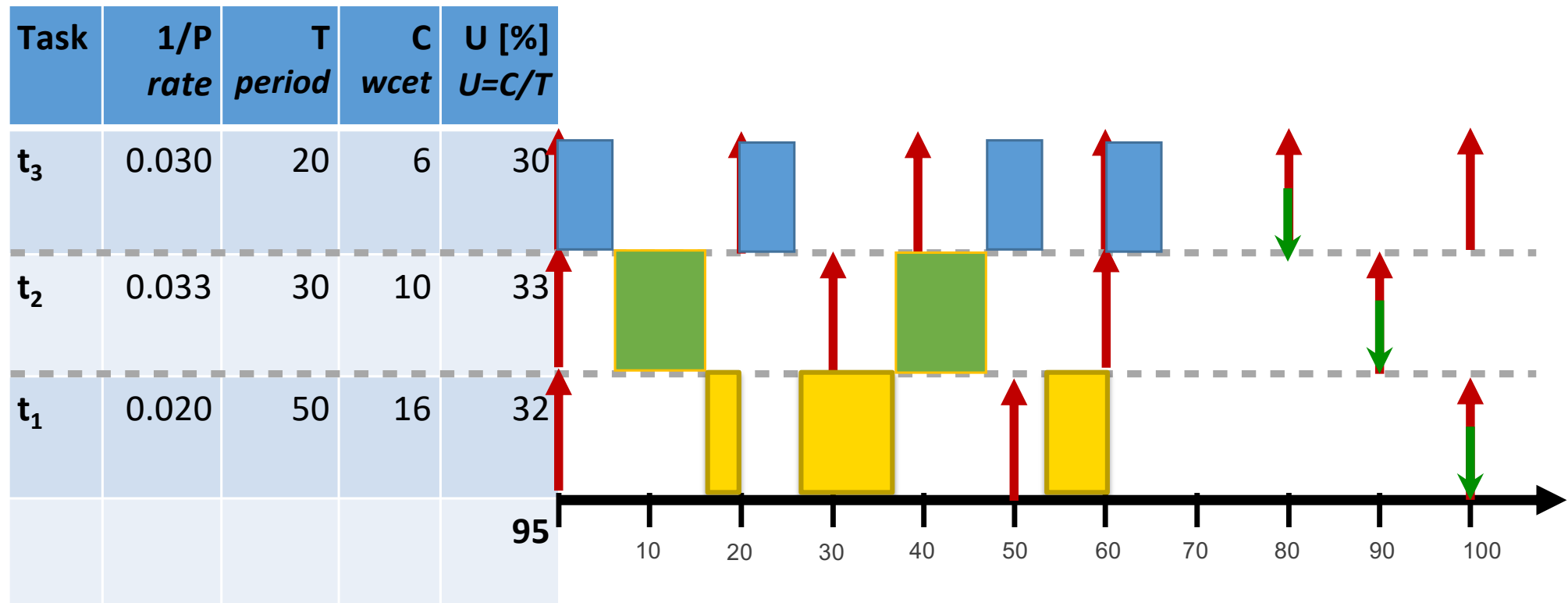
Use EDF!



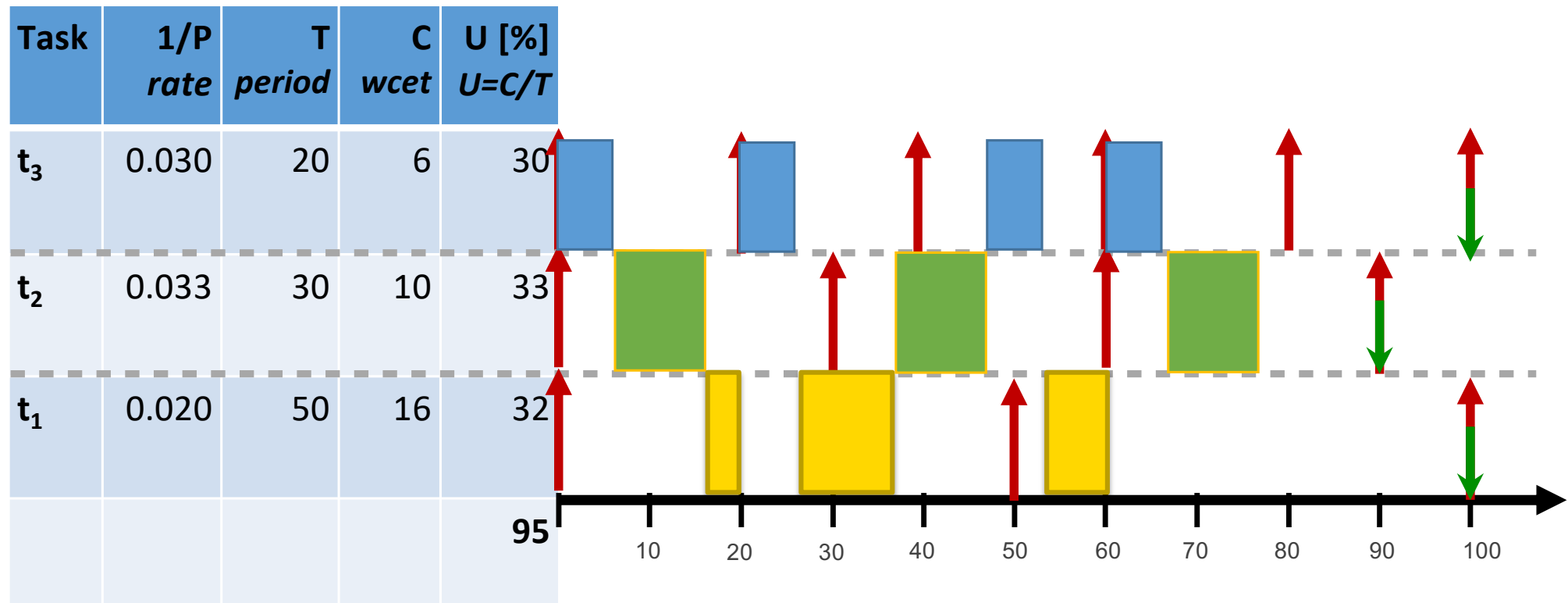
Use EDF!



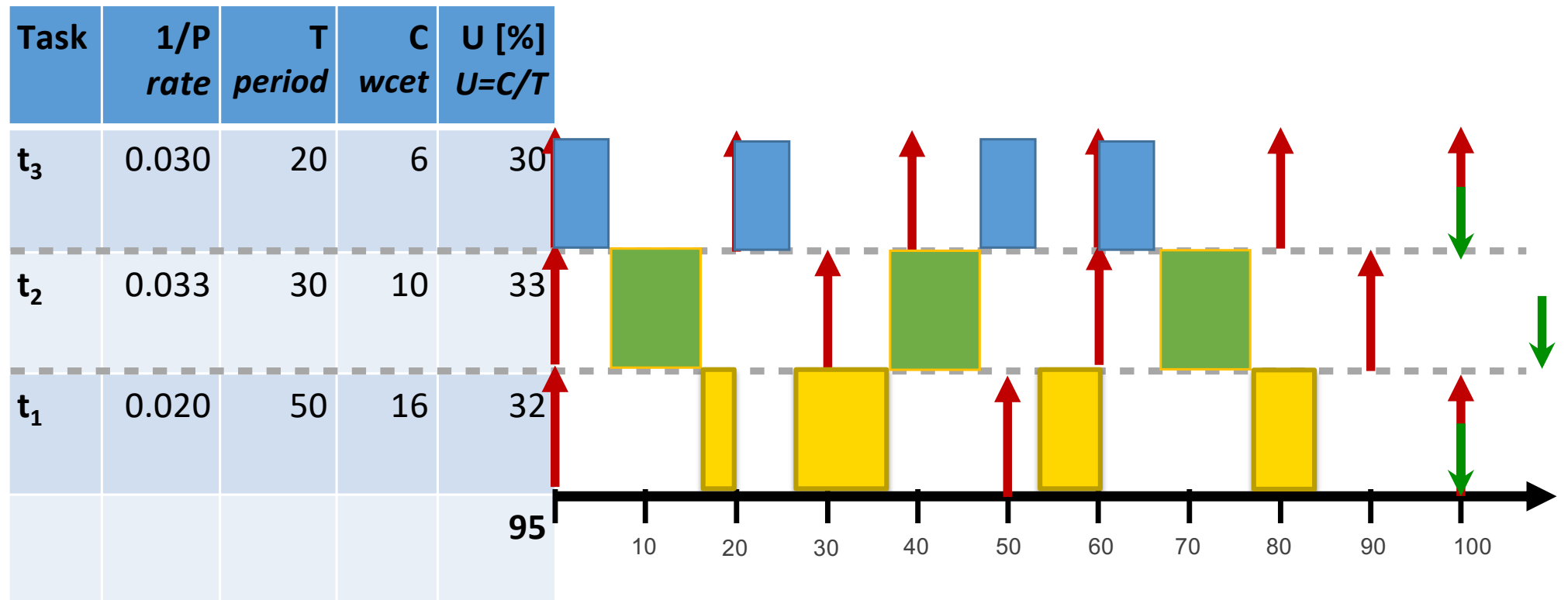
Use EDF!



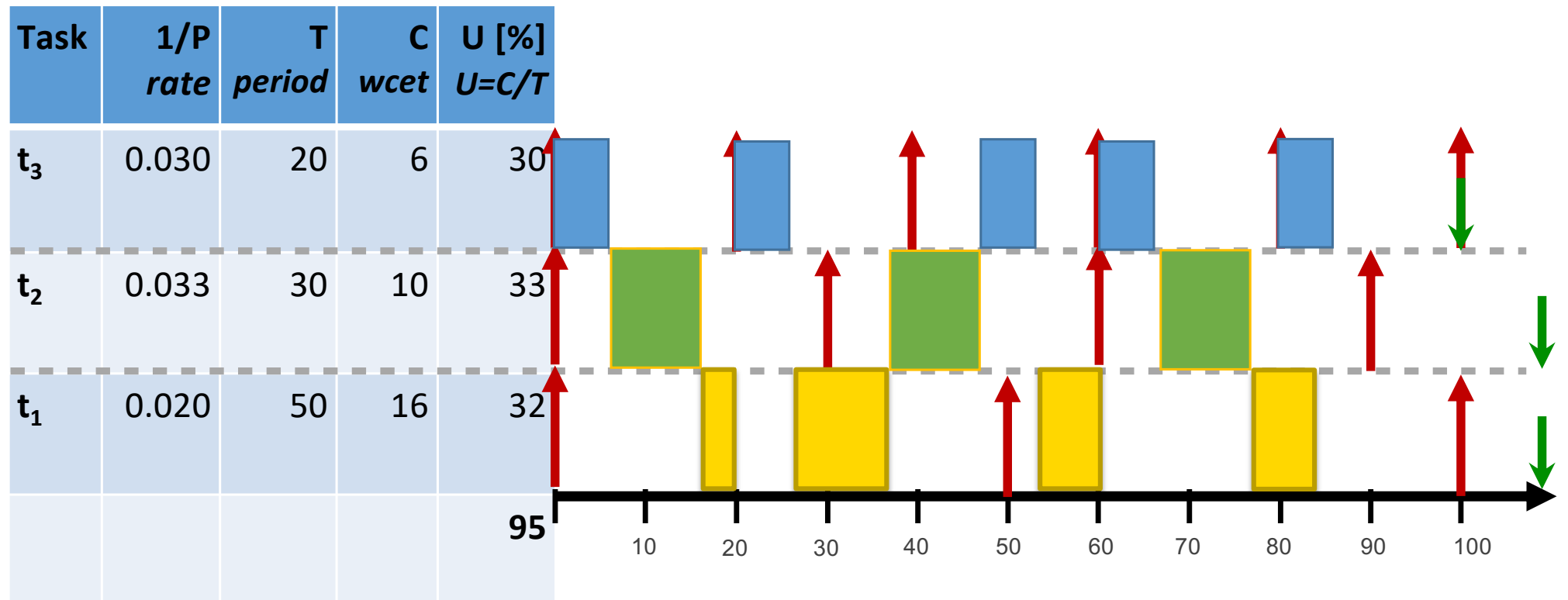
Use EDF!



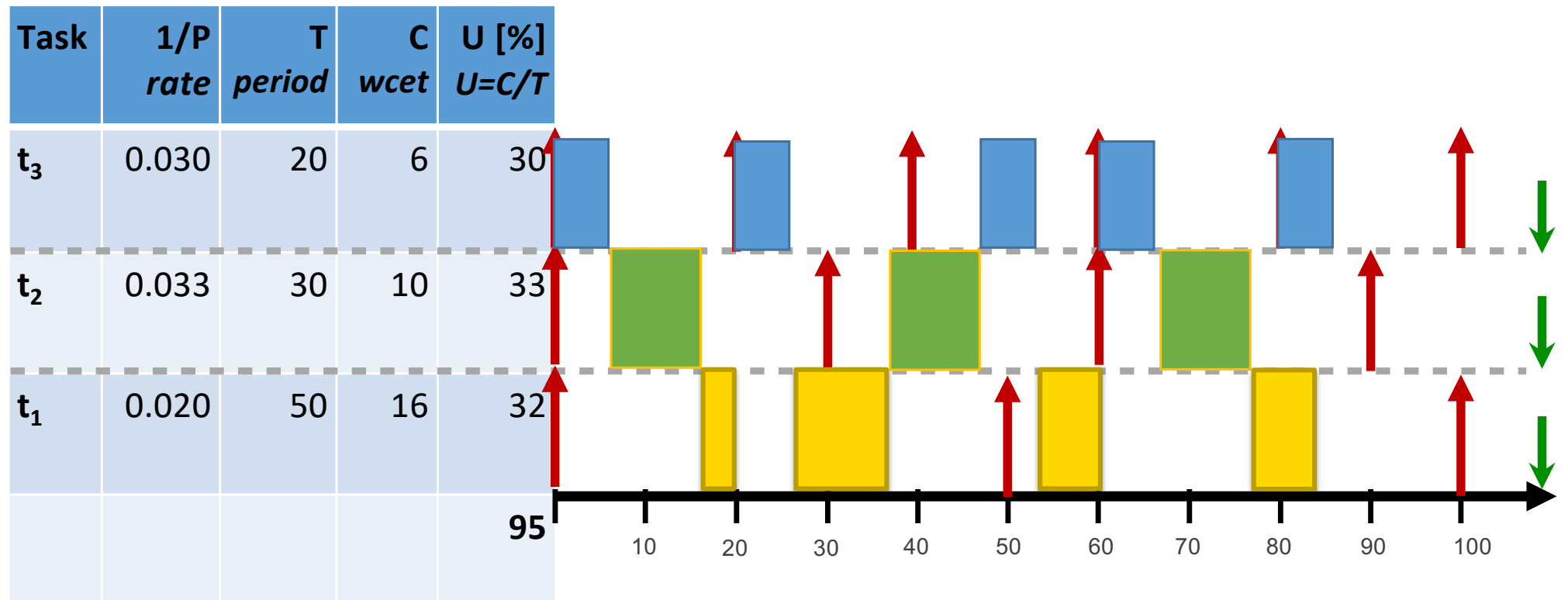
Use EDF!



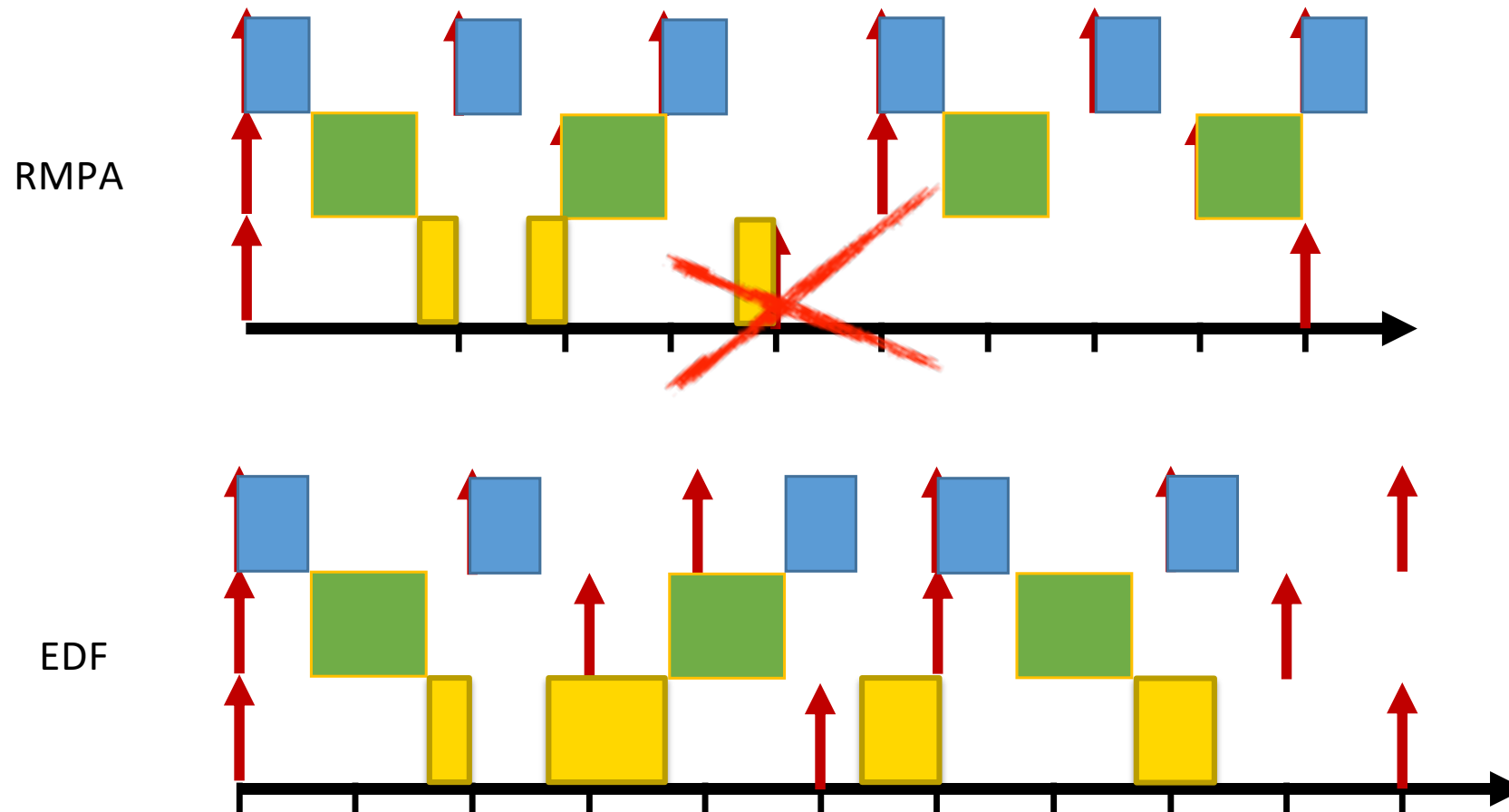
Use EDF!



Use EDF!

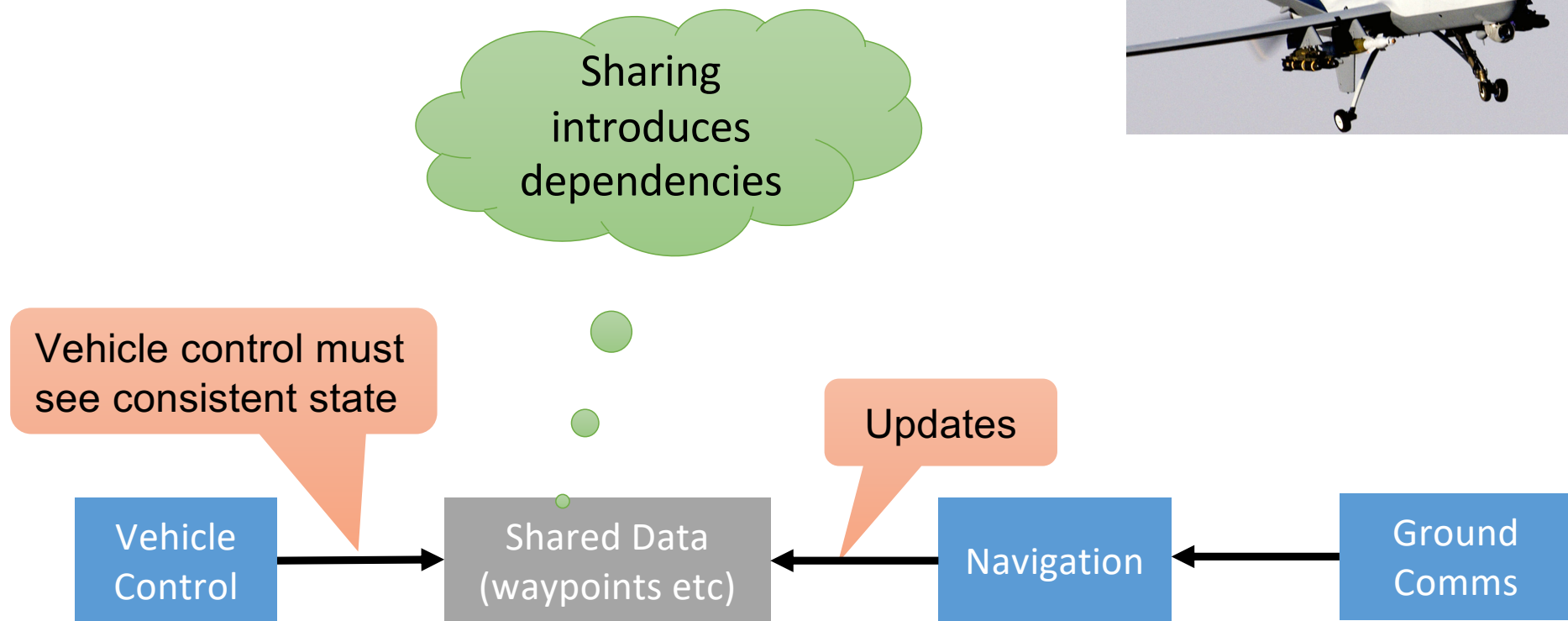


RMPA vs EDF!

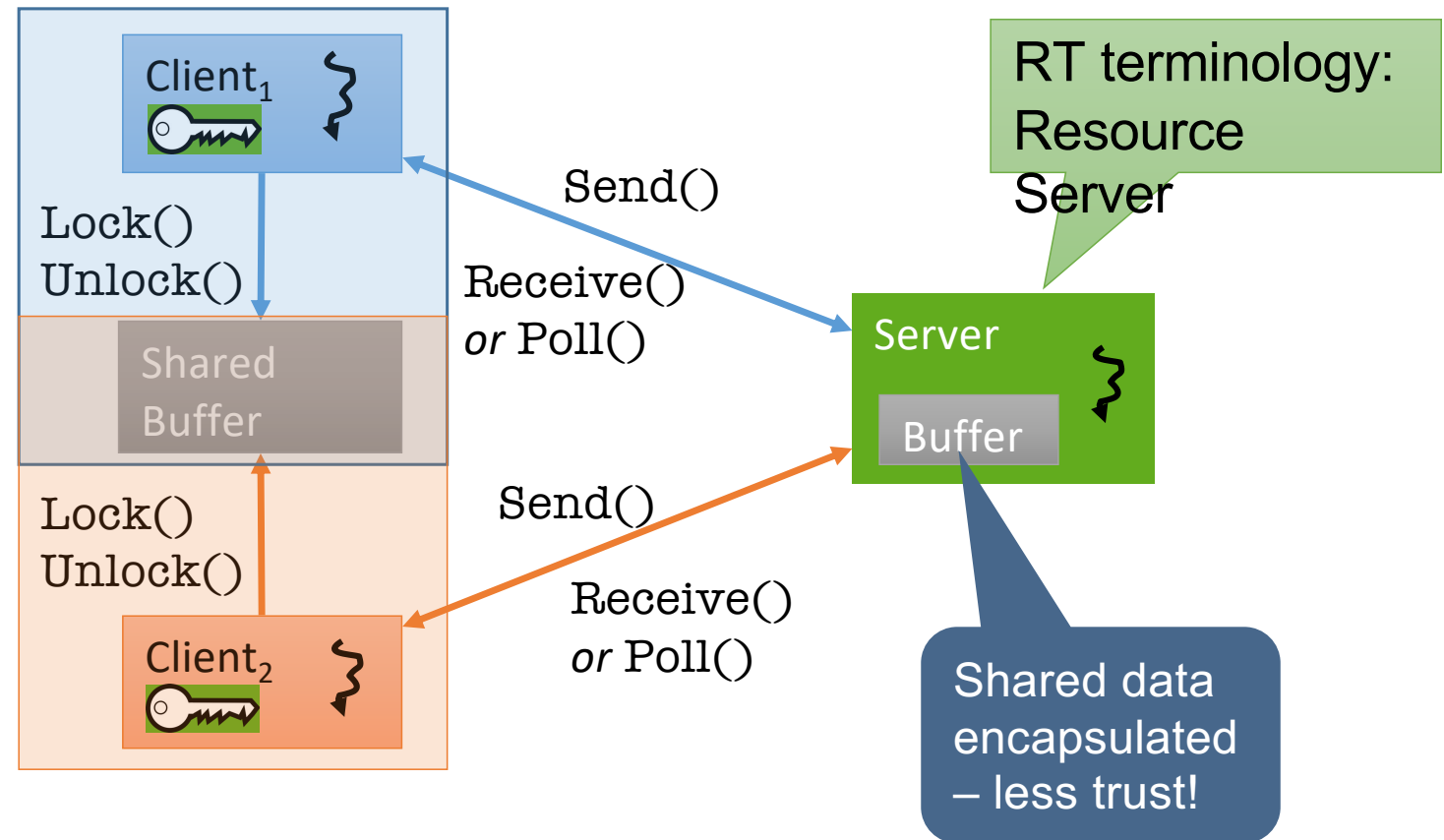


Resource Sharing

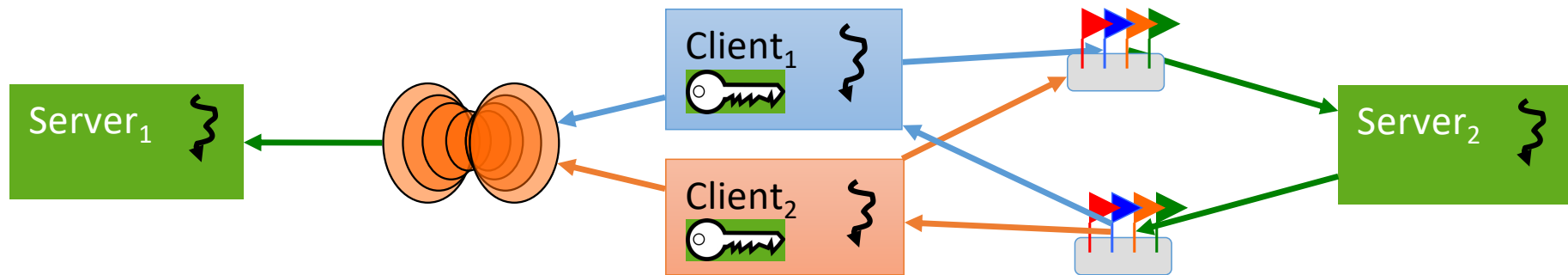
Challenge: Sharing



Critical Sections: Locking vs Delegation



seL4 Implementing Delegation



```
serv_local() {
...
Wait(ep);
while (1) {
/* critical section */
ReplyWait(ep);
}
}
```

Hoare-style monitor
Suitable intra-core

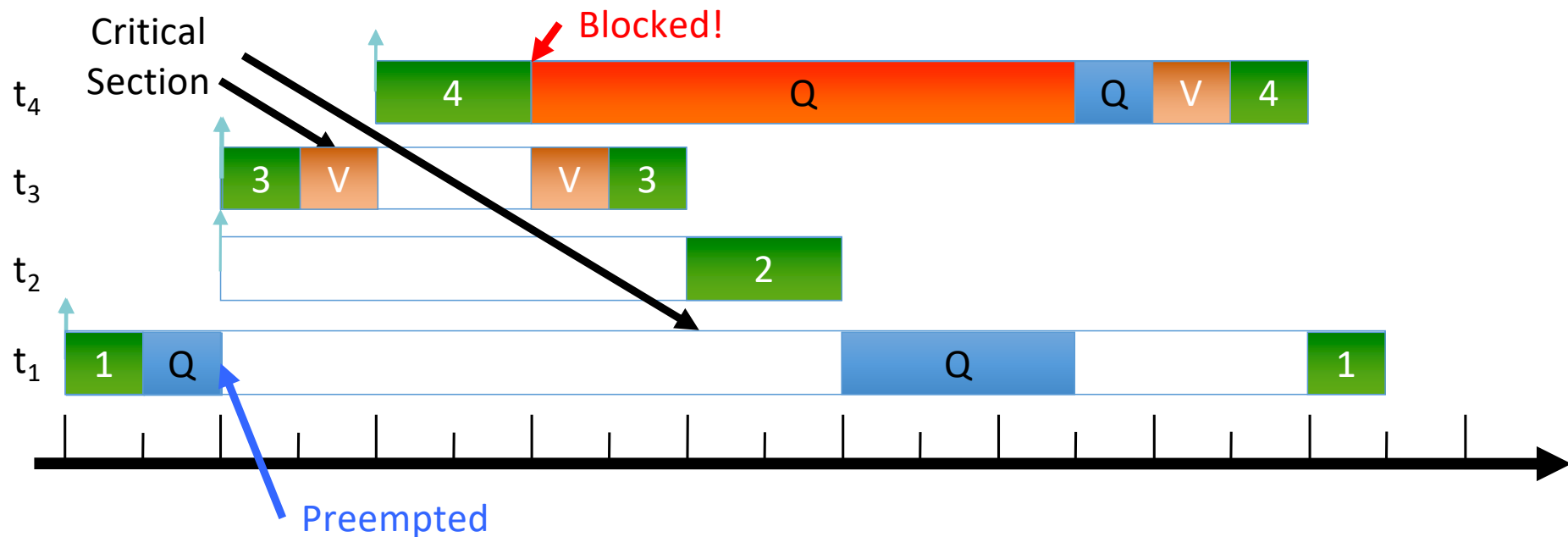
```
client() {
while (1) {
...
Call(ep);
...
Signal(not_ry);
...
Wait(not_rq);
}
}
```

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

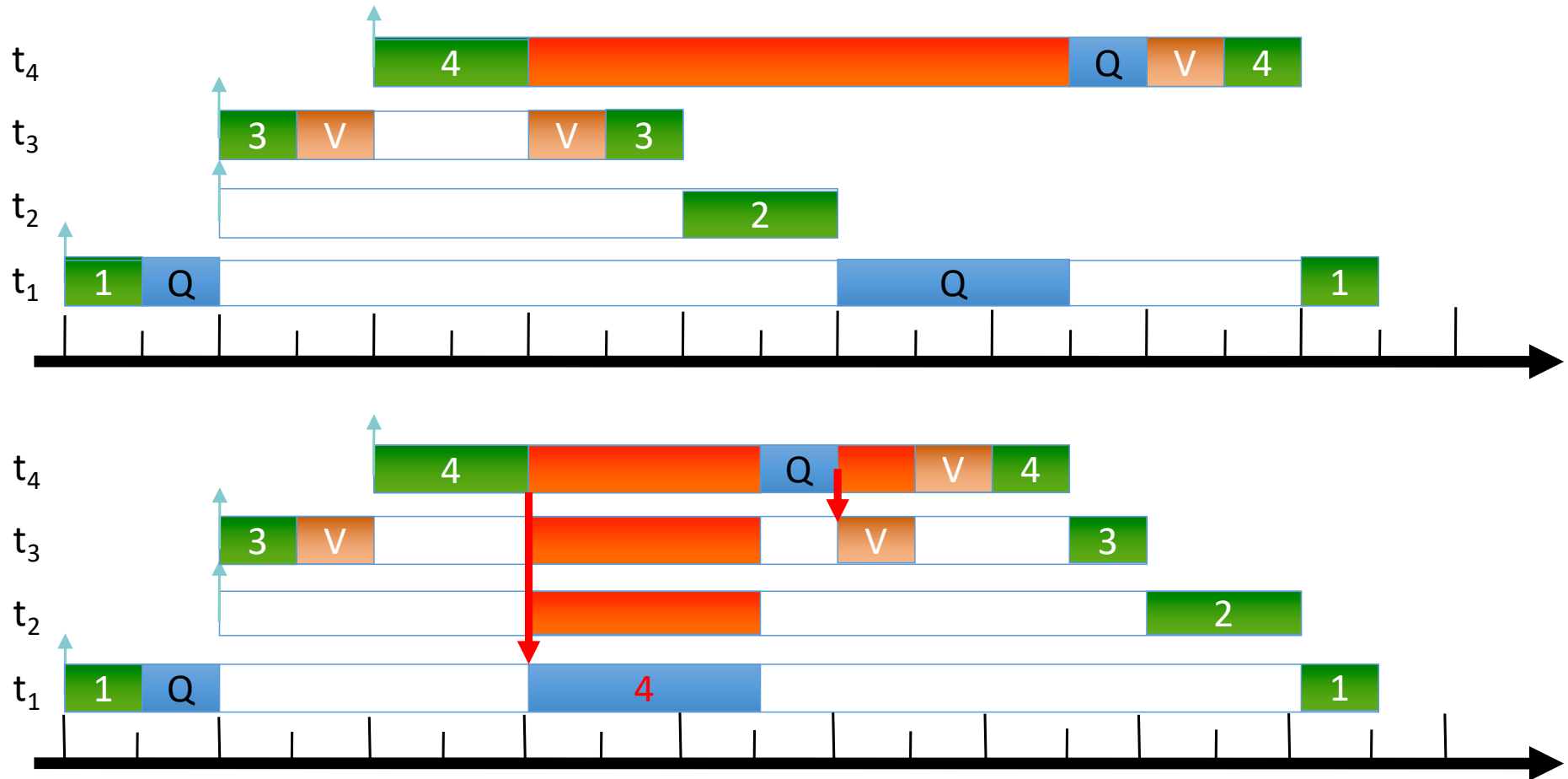
Semaphore synchronisation
Suitable inter-core

Problem: Priority Inversion

- High-priority job is blocked by low-prio for a long time
- Long wait chain: $t_4 \rightarrow t_1 \rightarrow t_3 \rightarrow t_2$
- Worst-case blocking time of t_4 bounded by total WCET: $C_1 + C_2 + C_3$



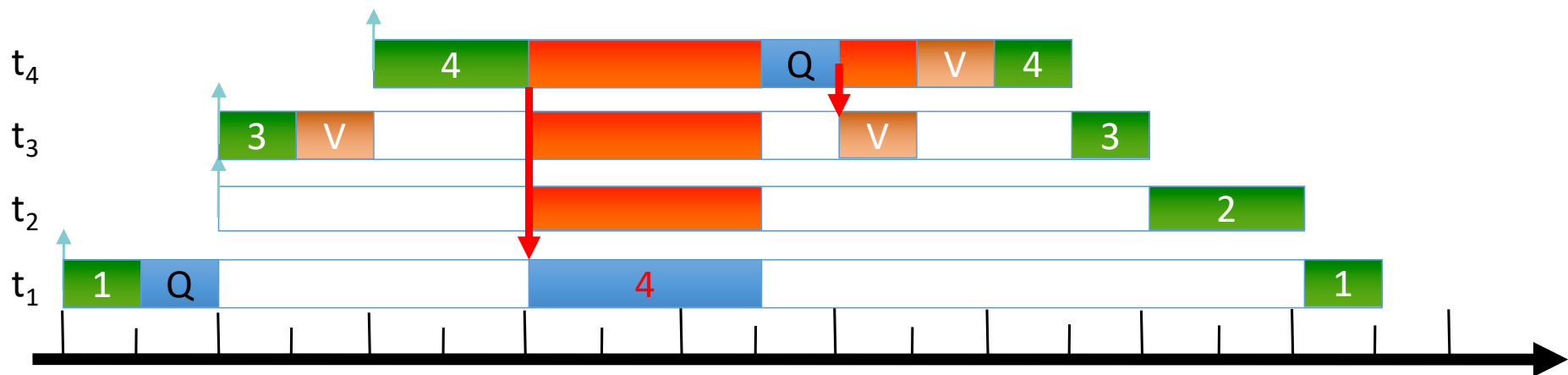
Solution 1: Priority Inheritance (“Helping”)



Solution 1: Priority Inheritance (“Helping”)

If t_1 blocks on a resource held by t_2 , and $P_1 > P_2$, then

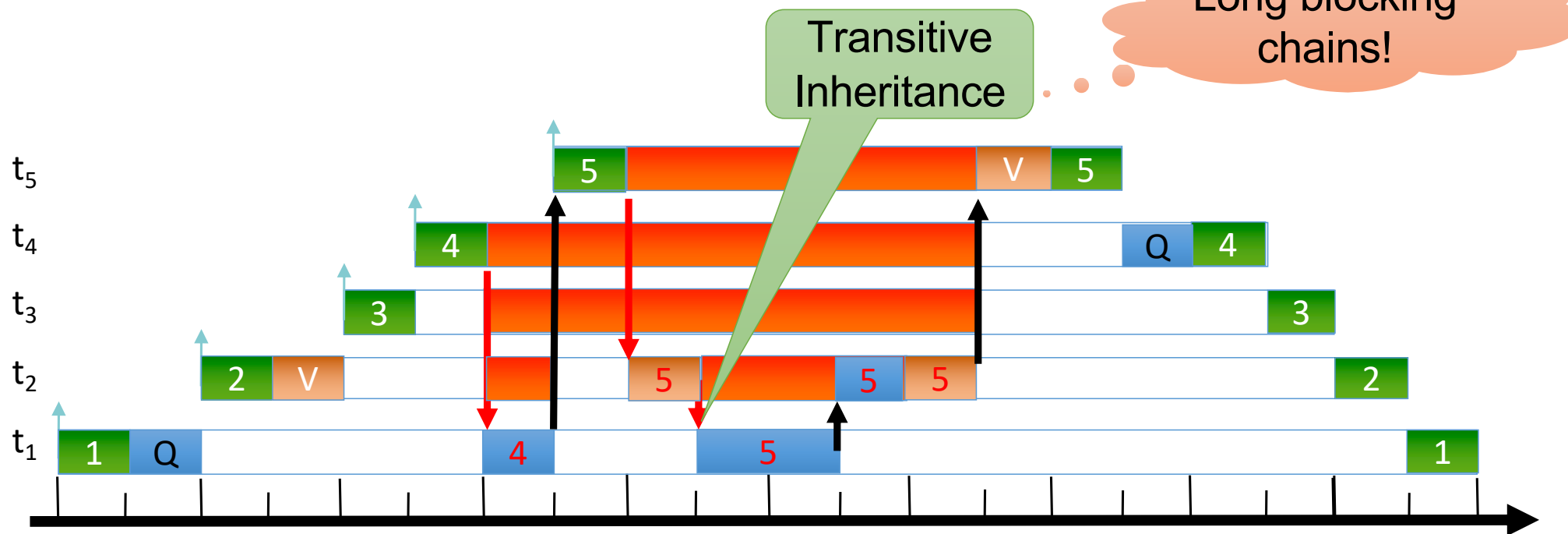
- t_2 is temporarily given priority P_1
- when t_1 releases the resource, its priority reverts to P_2



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_1 releases the resource, its priority reverts to P_2



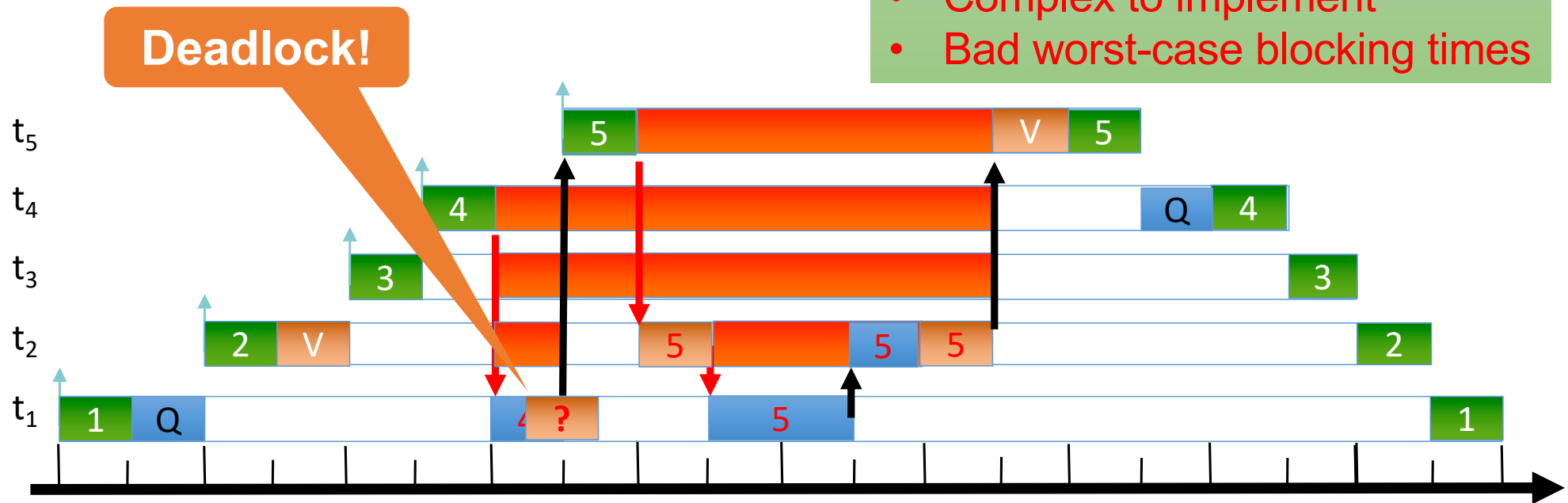
Solution 1: Priority Inheritance (“Helping”)

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_1 releases the resource, its priority

Priority Inheritance:

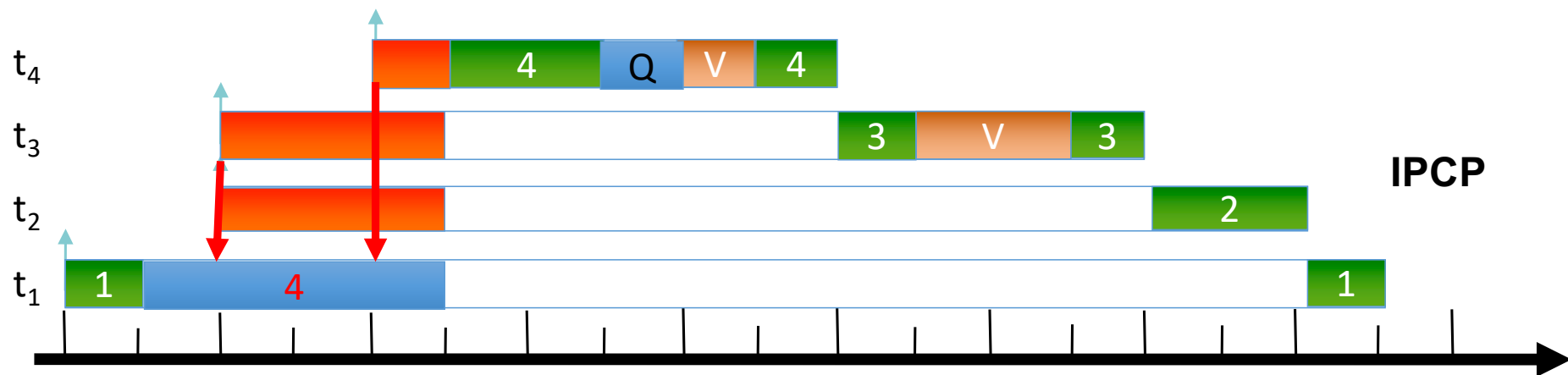
- Easy to use
- Potential deadlocks
- Complex to implement
- Bad worst-case blocking times



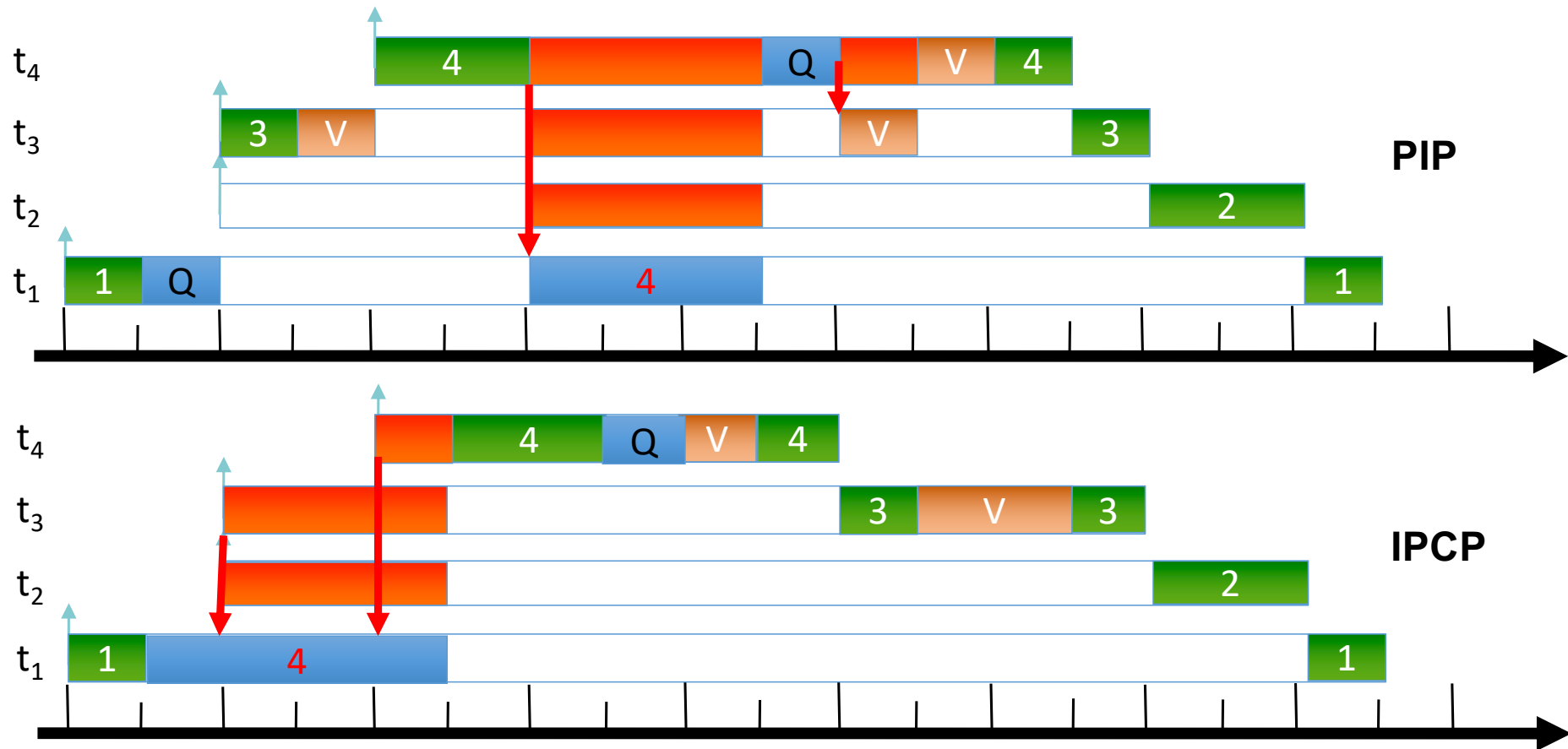
Solution 2: Priority Ceiling Protocol (PCP)

- 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

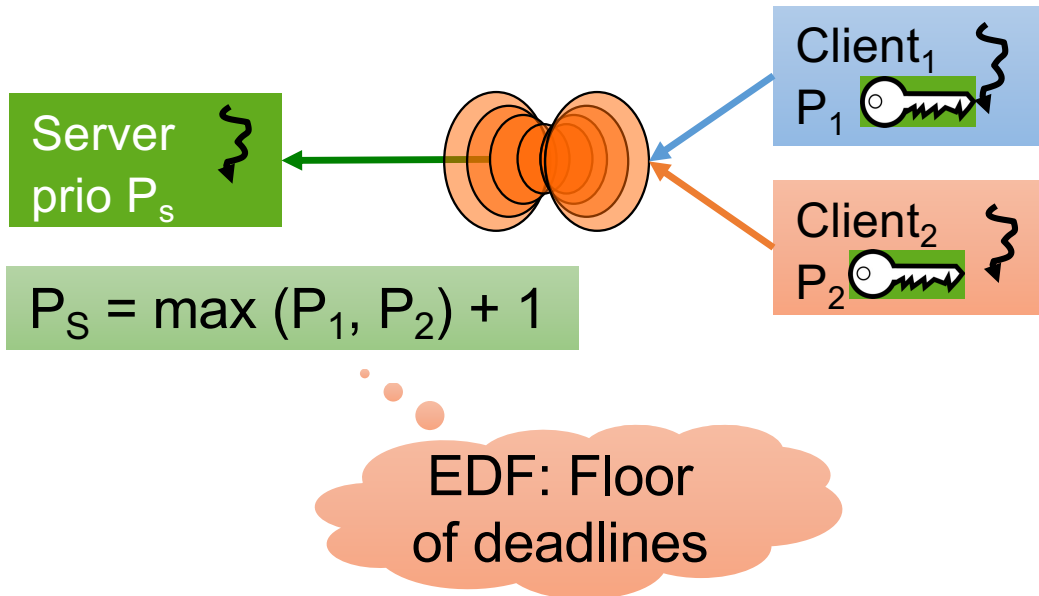
Immediate prio ceiling protocol (IPCP)



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

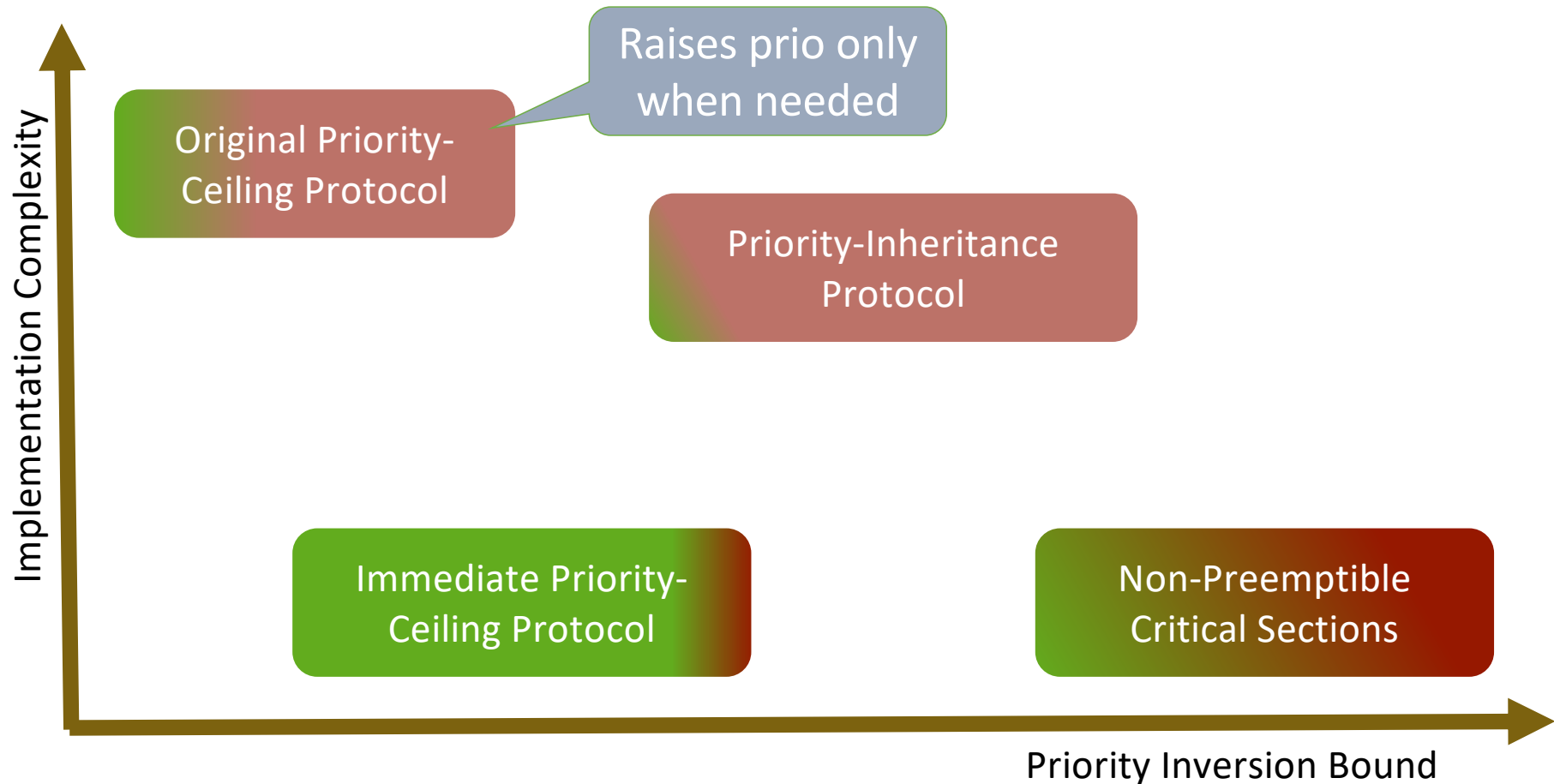
Each task must declare all resources at admission time

- System must maintain list of tasks using resource
- Defines ceiling priority

Easy to enforce with caps



Comparison of Locking Protocols



Scheduling Overloaded RT Systems

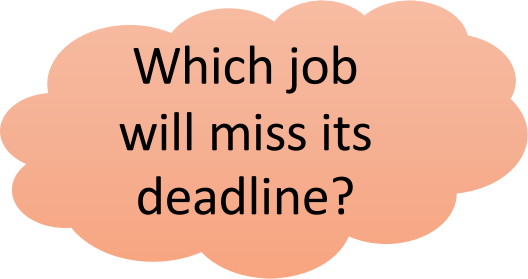
Naïve Assumption: Everything is Schedulable

Standard assumptions of classical RT systems:

- All WCETs known
- All jobs complete within WCET
- Everything is trusted

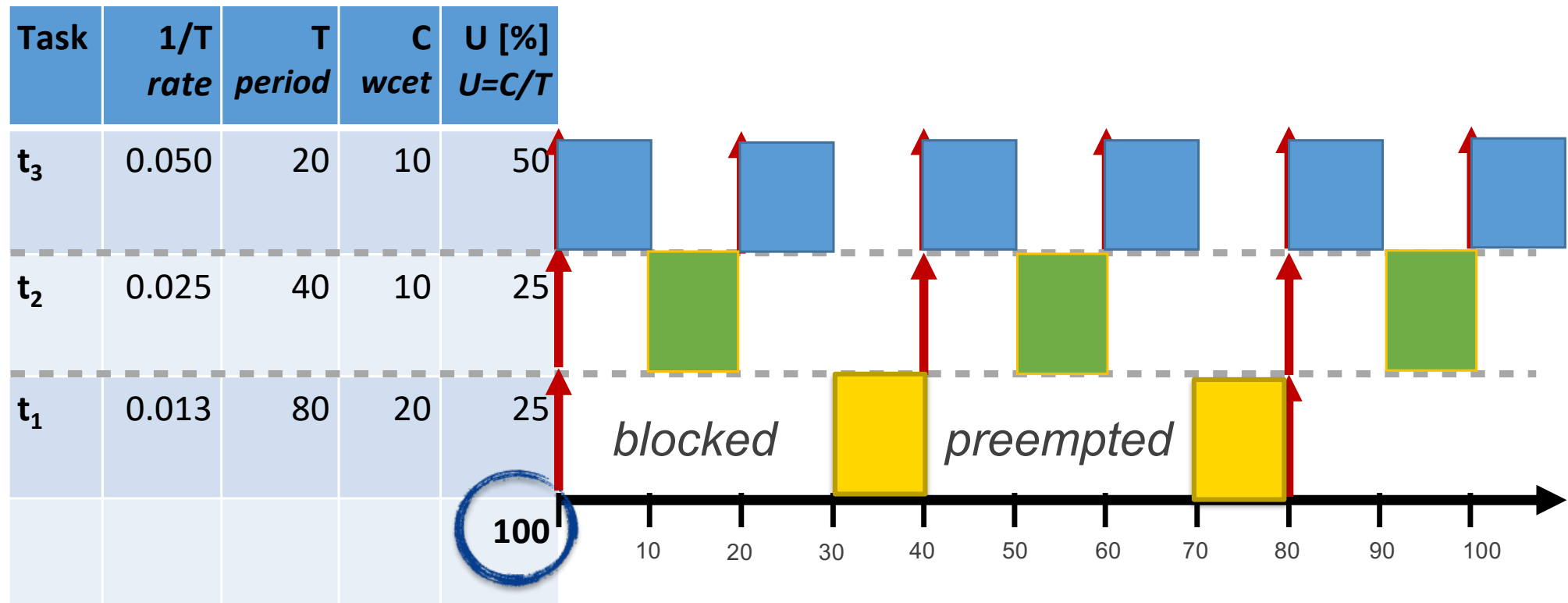
More realistic: Overloaded system:

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

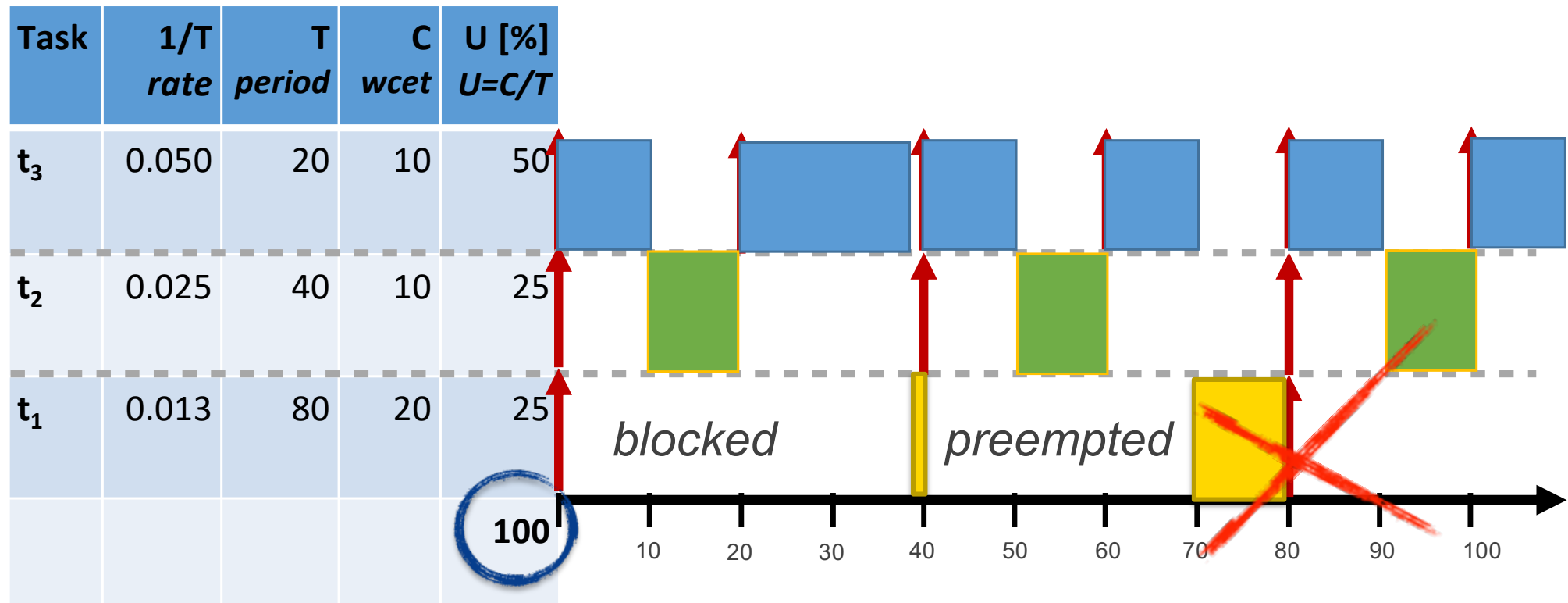


Which job
will miss its
deadline?

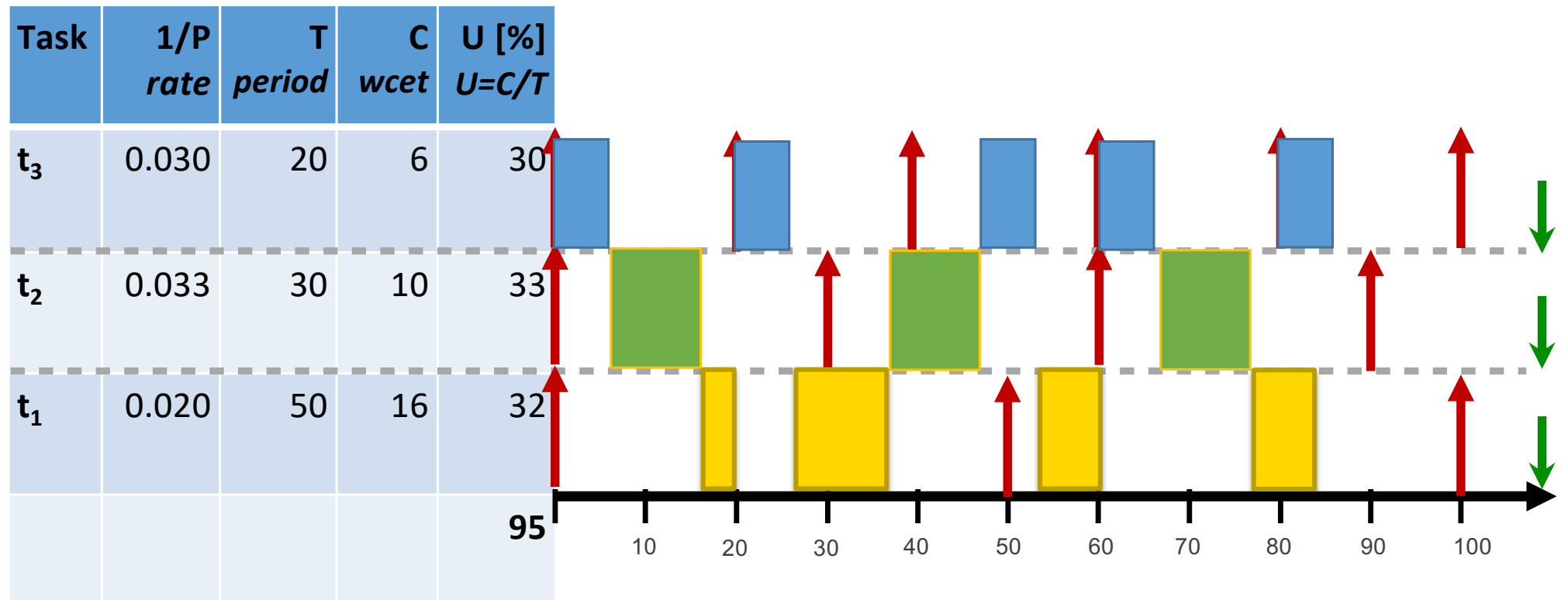
Rate-Monotonic Overload



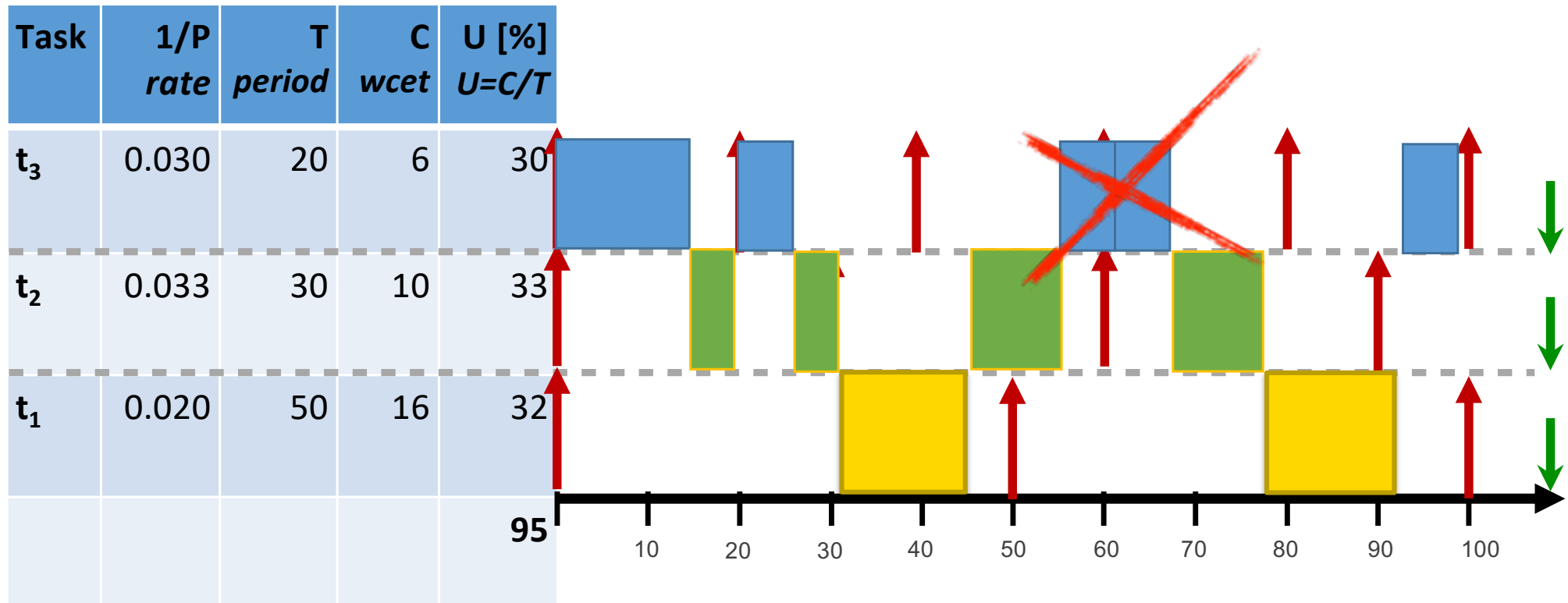
Rate-Monotonic Overload



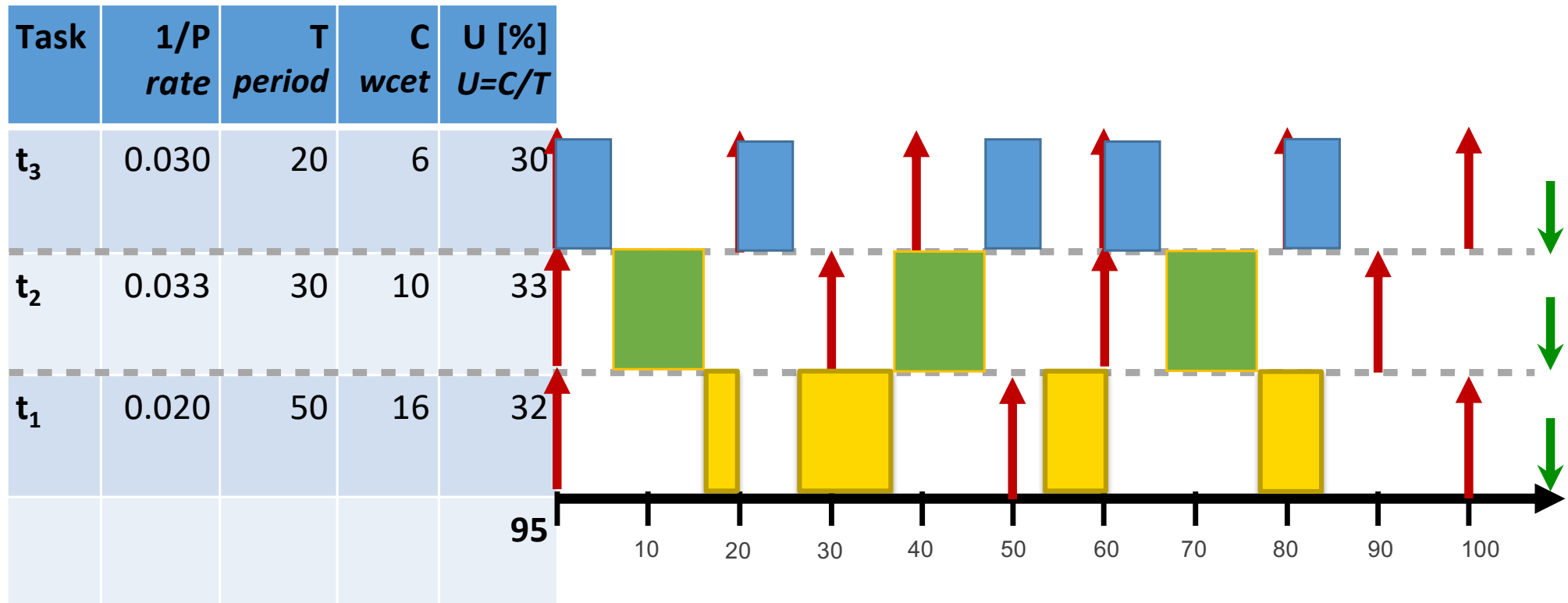
EDF Overload #1



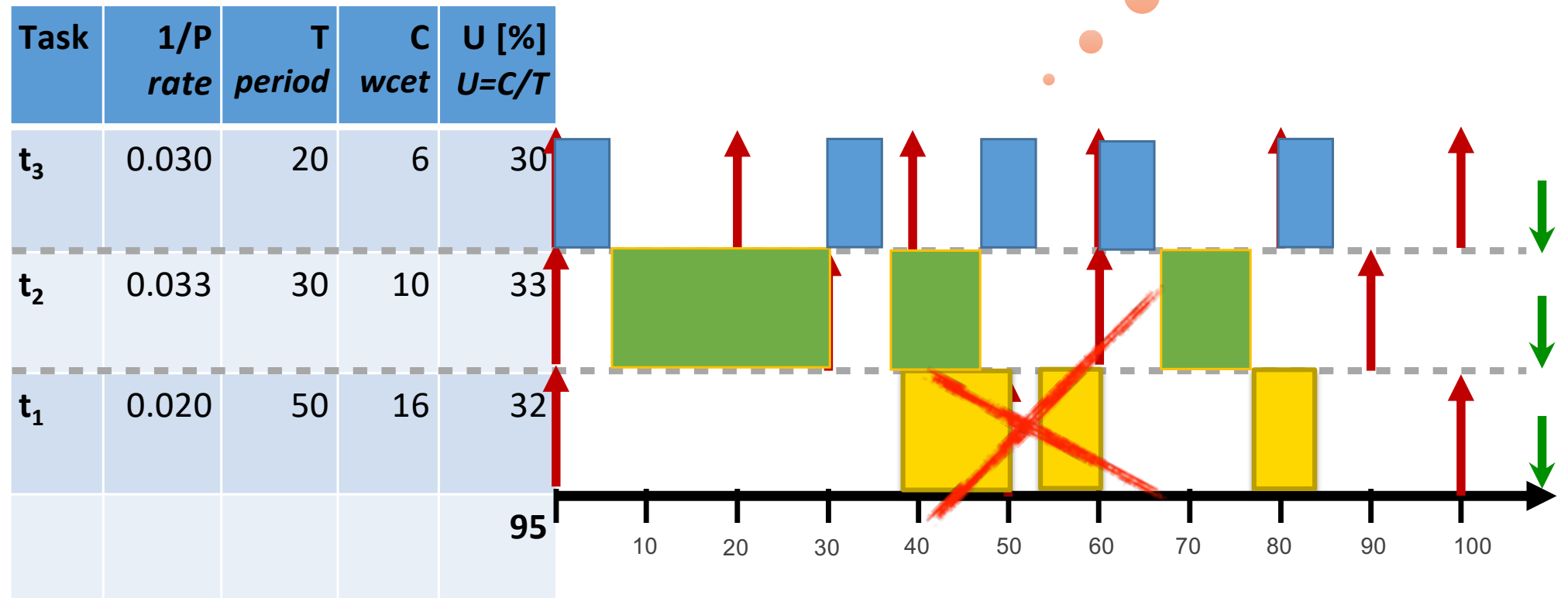
EDF Overload #1



EDF Overload #2

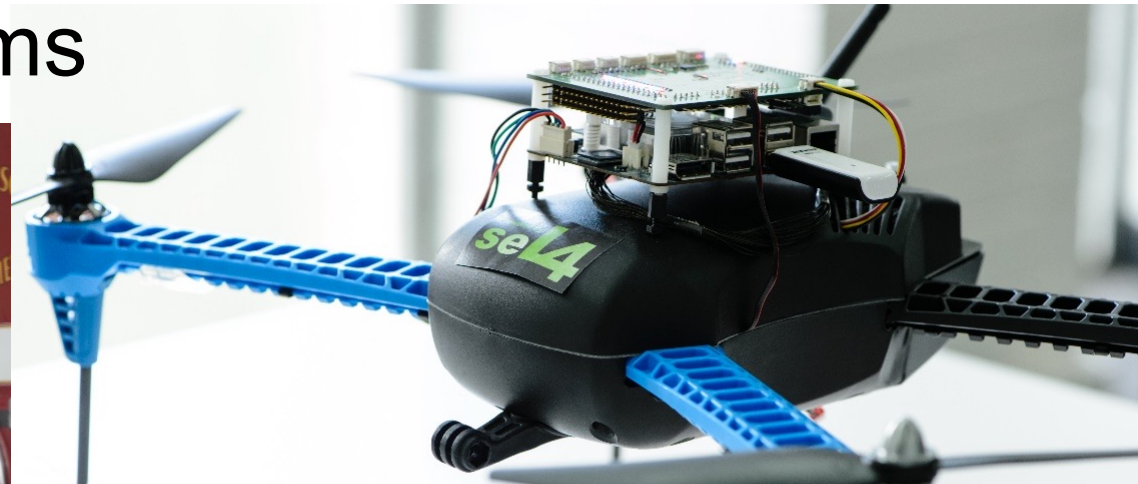


EDF Overload #2



Mixed-Criticality Systems

Mixed Criticality Systems



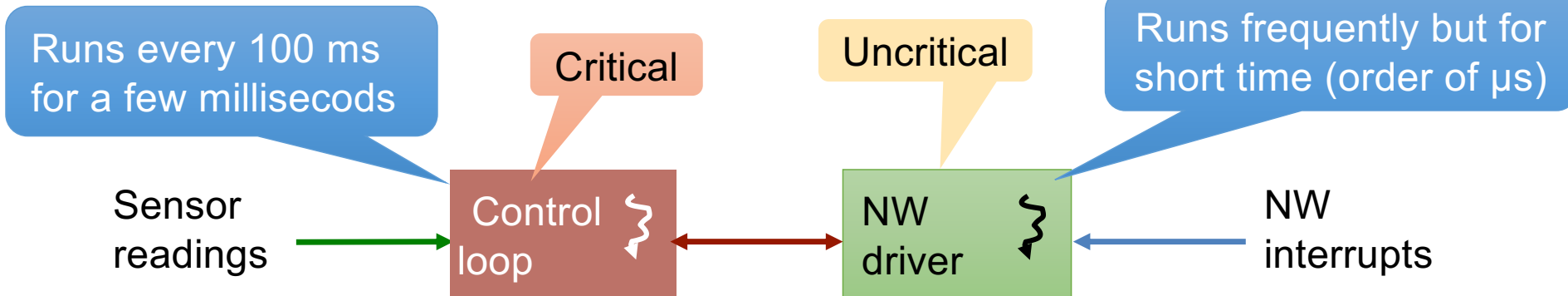
Mixed Criticality

Need temporal isolation!



Network (NW) driver must preempt control loop

- ... to avoid packet loss
- Driver must run at high prio (i.e. RMPA)
- ***Driver must not monopolise CPU***



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*

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



Critical system certification:

- expensive
- conservative assumptions
 - eg highly pessimistic WCET

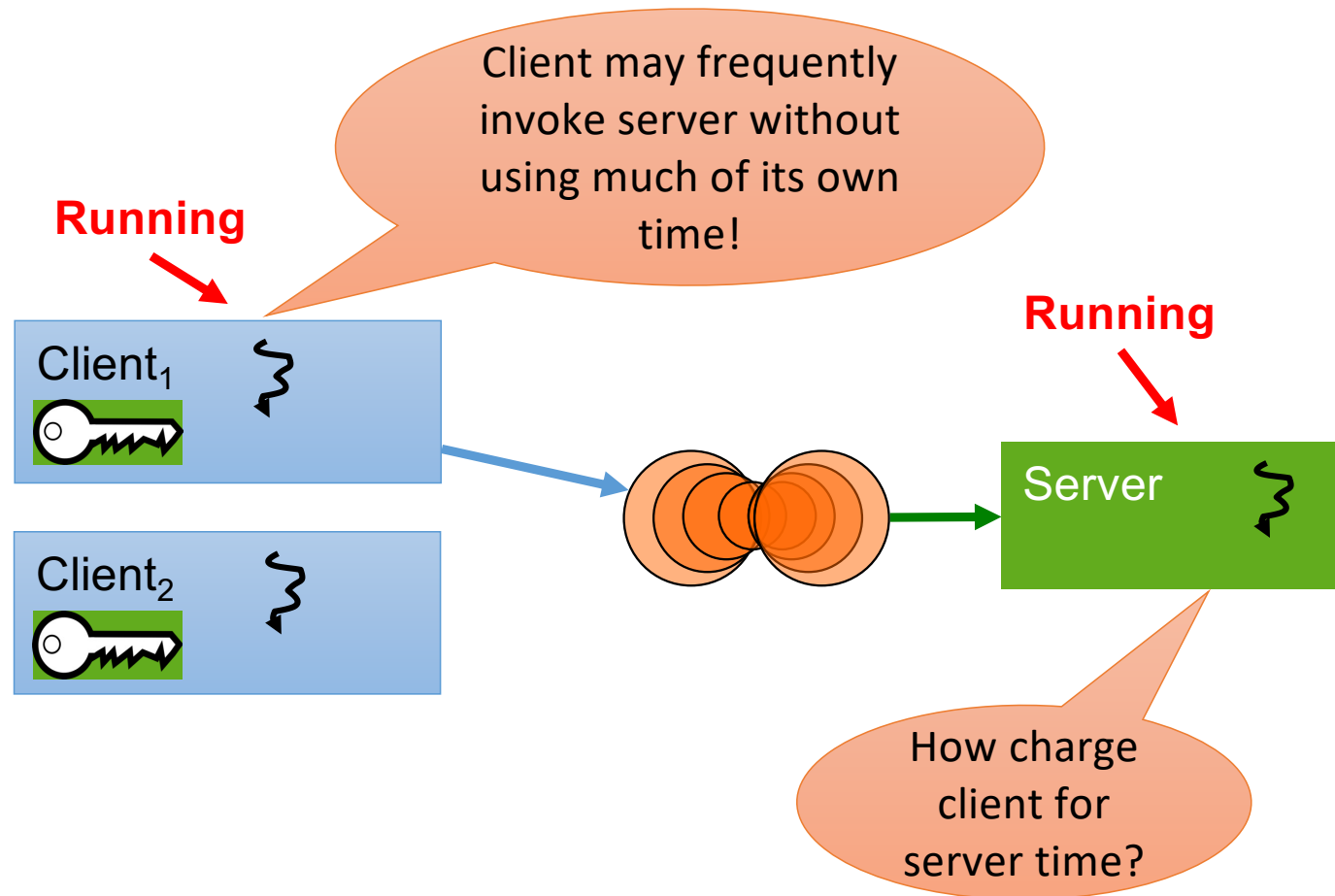
- Must minimise critical software
- Need temporal isolation:
Budget enforcement

Mixed-Criticality Support

For supporting *mixed-criticality systems* (MCS), OS must provide:

- *Temporal isolation*, to force jobs to adhere to declared WCET
- Mechanisms for *safely sharing resources* across criticalities

Remember: Delegation of Critical Sections



seL4 MCS Model: Scheduling Contexts

Classical thread attributes

- Priority
- Time slice

Not runnable
if null

MCS thread attributes

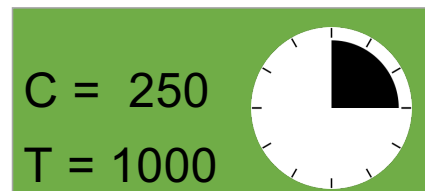
- Priority
- Scheduling context capability

Capabilit
y for time

Scheduling context object

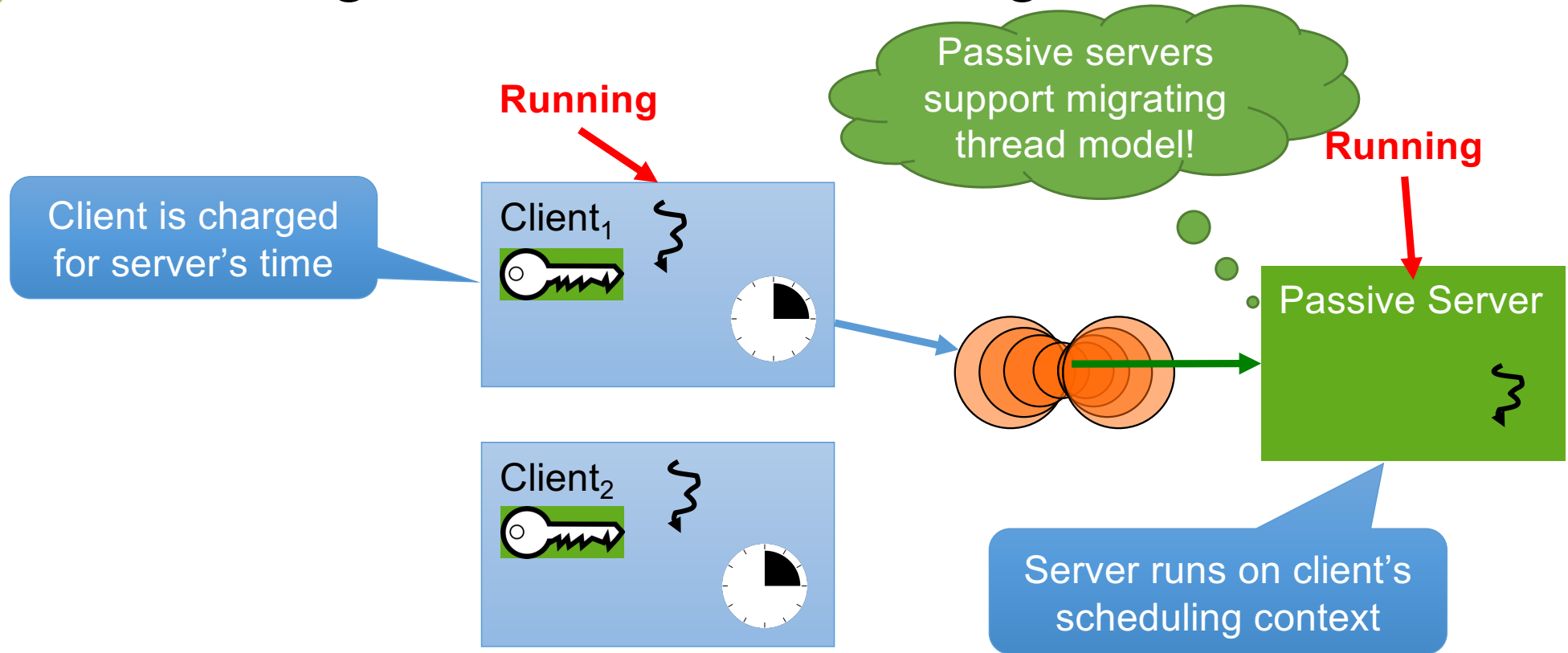
- T: period
- C: budget ($\leq T$)

Limits CPU
access!



Per-core SchedControl capability
conveys right to assign budgets
(i.e. perform admission control)

seL4 Delegation with Scheduling Contexts



Scheduling-context capabilities: a principled, light-weight OS mechanism for managing time [Lyons et al, EuroSys'18]

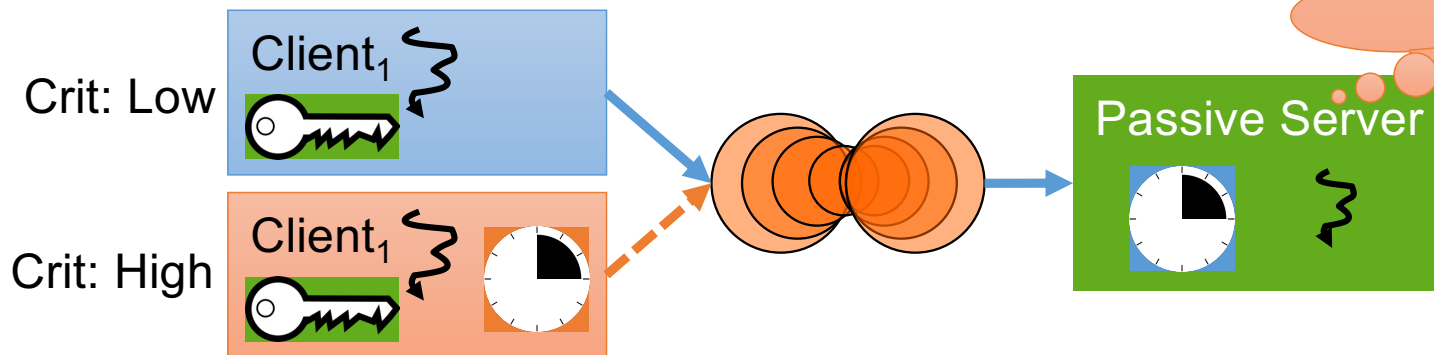
se14 Mixed-Criticality Support

For *mixed-criticality systems* (MCS), OS must provide:

- *Temporal isolation*, to force jobs to adhere to declared WCET

Solved by scheduling contexts

- Mechanisms for *safely sharing resources* across criticalities



What if budget expires while shared server executing on Low's scheduling context?

Timeout Exceptions

Policy-free mechanism for dealing with budget depletion

Possible actions:

- Provide emergency budget to leave critical section
- Cancel operation & roll-back server
- Reduce priority of low-crit client (with one of the above)
- Implement priority inheritance (if you must...)

Arguable not ideal: better prevent timeout completely

Pending RFC against seL4: budget thresholds

Presented by Dr Anna Lyons

Internship!

<https://jobs.apple.com/en-au/details/200680345-3957/secure-kernel-engineering-intern-sydney?team=SFTWR>

search “secure kernel engineering intern apple”

Contact

linked in: <https://www.linkedin.com/in/annamlyons/>

email: anna.lyons@apple.com

Fun links

For the dark nights of AOS debugging: “The Night Watch”

https://www.usenix.org/system/files/1311_05-08_mickens.pdf

Real world priority inversion: NASA

<https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft>

Real world mess: (When real time is wrong) Toyota breaking

<https://www.transportation.gov/briefing-room/us-department-transportation-releases-results-nhtsa-nasa-study-unintended-acceleration>

Apple security blog: <https://security.apple.com/blog/>