

Real-Time Embedded Computing Systems

Giorgio Buttazzo

Scuola Superiore Sant'Anna, Pisa

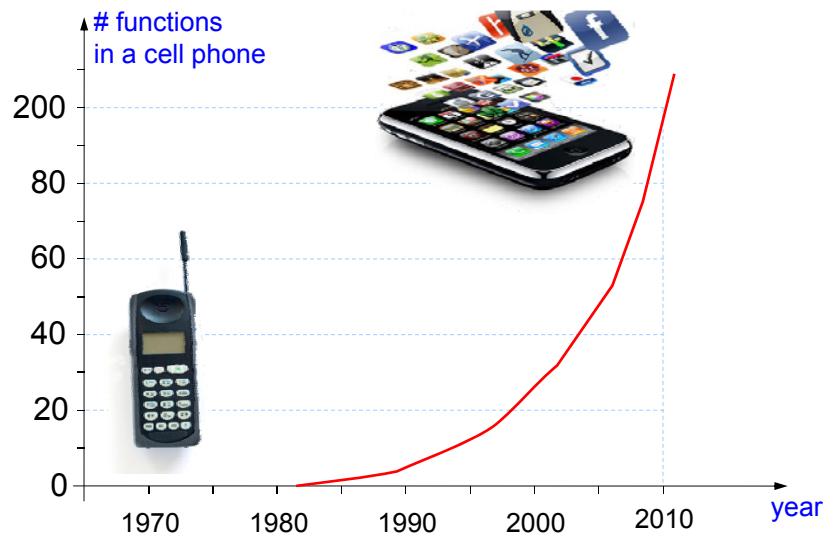


Computers everywhere

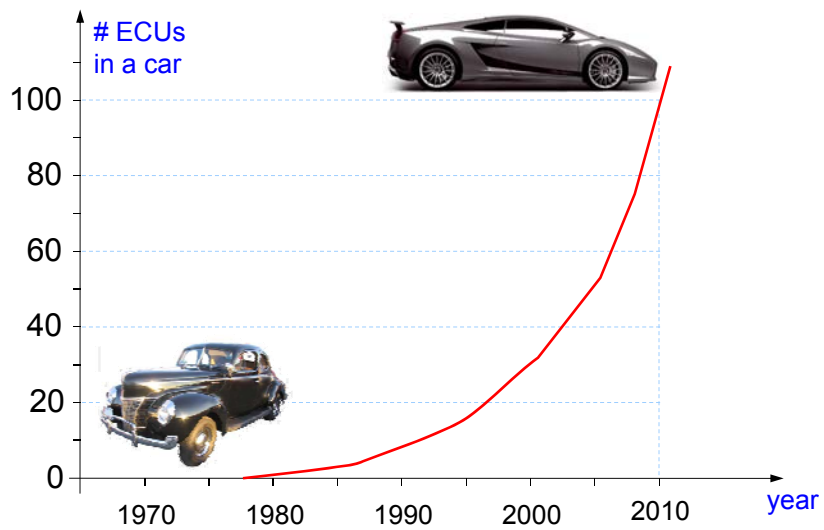
Today, **98%** of all processors in the planet are embedded in other objects:



Increasing complexity



ECU growth in a car

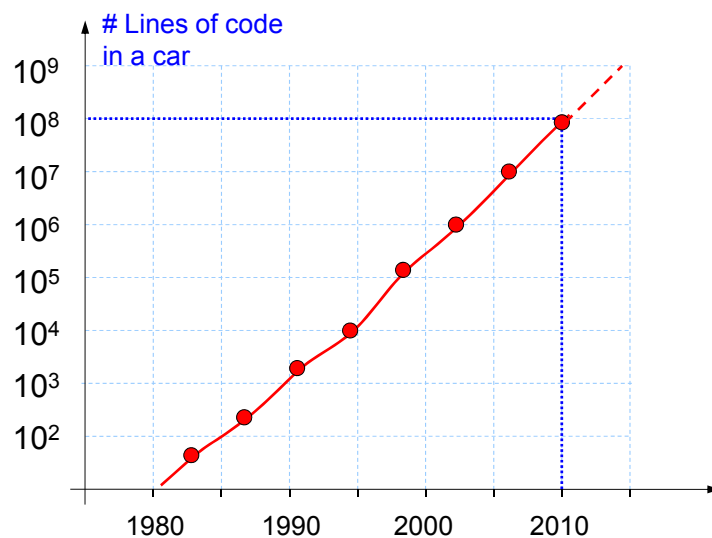


Software in a car

Car software controls almost everything:

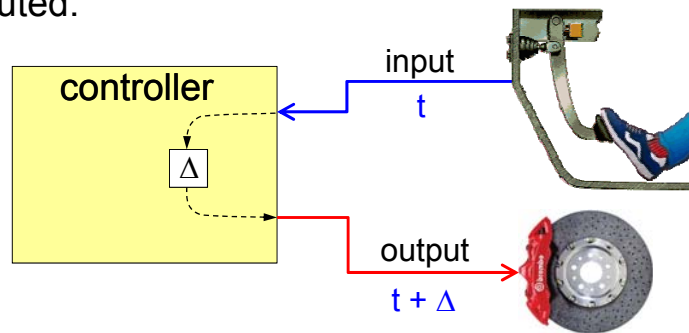
- **Engine:** ignition, fuel pressure, water temperature, valve control, gear control,
- **Dashboard:** engine status, message display, alarms
- **Diagnostic:** failure signaling and prediction
- **Safety:** ABS, ESC, EAL, CBC, TCS
- **Assistance:** power steering, navigation, sleep sensors, parking, night vision, collision detection
- **Comfort:** fan control, air conditioning, music, regulations: steer/lights/sits/mirrors/glasses...

Software evolution in a car



Software reliability

Reliability does not only depend on the correctness of single instructions, but also on **when** they are executed:



A correct action executed too late can be **useless** or even **dangerous**.

Real-Time System

A computing system that must guarantee bounded and predictable response times is called **real-time system**.

Predictability of response times must be guaranteed in the **worst-case scenario**:

- for each critical activity;
- for all possible combination of events.

Outline

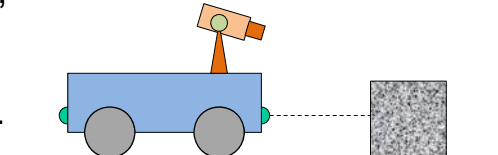
1. Basic concepts
2. Modeling real-time activities
3. Where timing constraints come from?
4. Real-time scheduling algorithms
5. Handling shared resources

Basic concepts

A sample control application

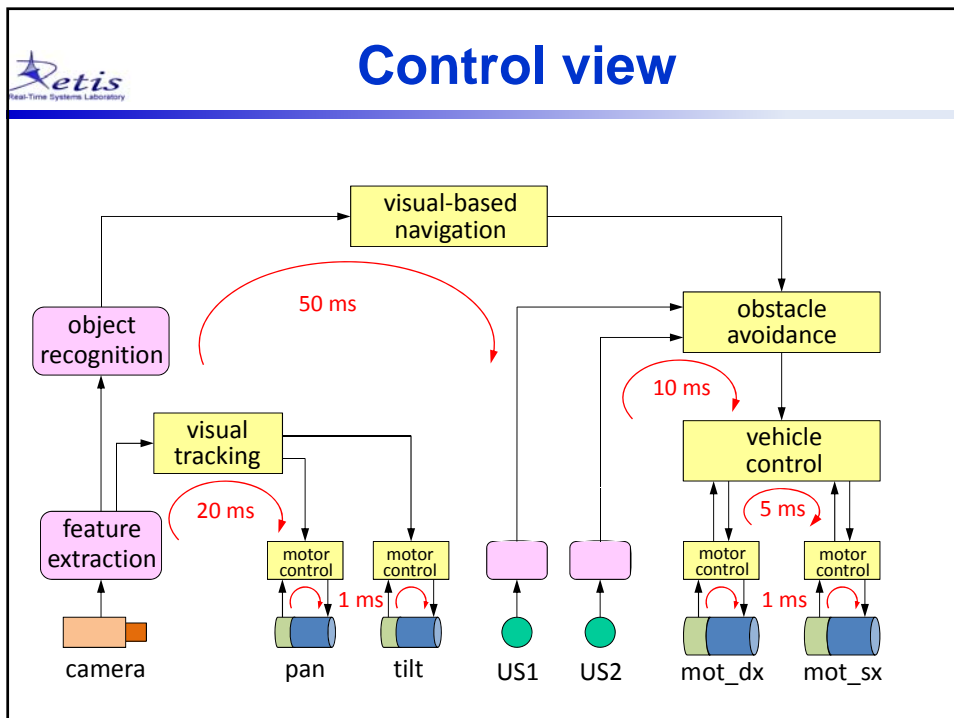
Mobile robot equipped with:

- two actuated wheels;
- two proximity sensors;
- a mobile camera;
- a wireless transceiver.



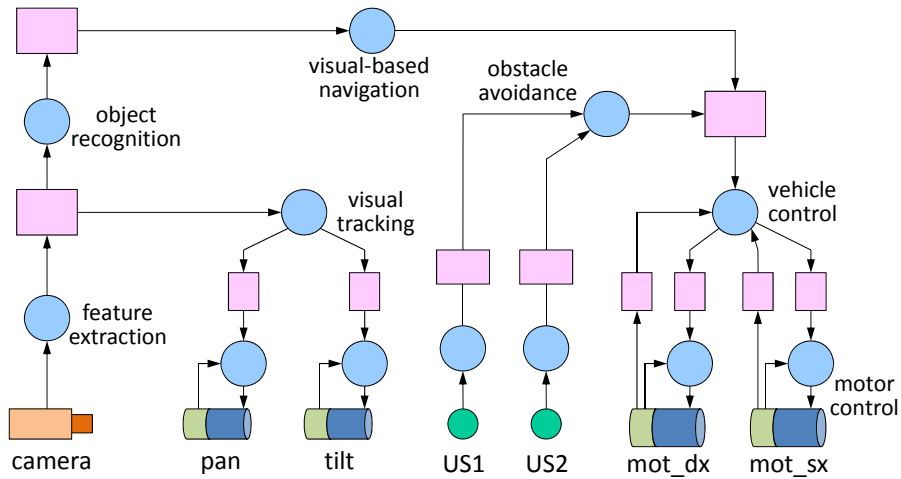
Goal

- Follow a path based on visual information;
- Avoid obstacles;
- Send system status every 20 ms.

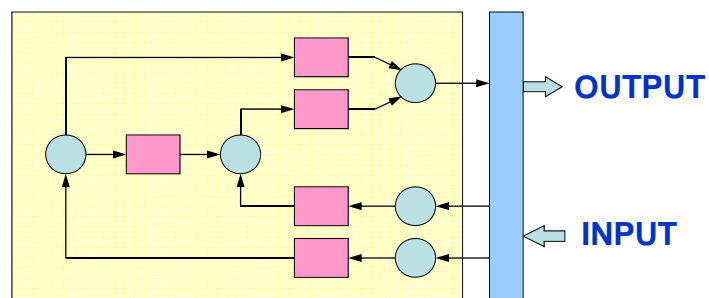


Software view

● periodic task □ buffer



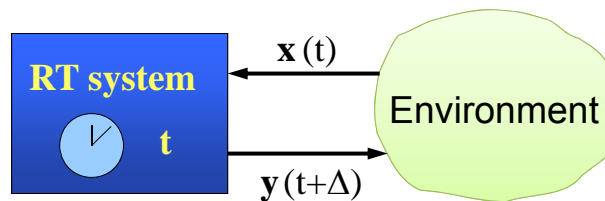
Software structure



● task □ buffer

Real-Time System

It is a system in which the correctness depends not only on the output values, but also on the **time** at which results are produced.



RTOS responsibilities

The real-time operating system is responsible for:

- activating periodic tasks at the beginning of each period;
- deciding the execution order of tasks ([scheduling](#));
- solving possible timing conflicts during the access of shared resources ([mutual exclusion](#));
- manage the timely execution of asynchronous events ([interrupts](#)).

Real-Time \neq Fast

- A real-time system is **not** a fast system.
- Speed is always relative to a specific environment.
- Running faster is good, but does not guarantee a correct behavior.

Speed vs. Predictability

- The objective of a real-time system is to guarantee the timing behavior of each individual task.
- The objective of a fast system is to minimize the average response time of a task set. But ...

Don't trust the average when you have to guarantee individual performance

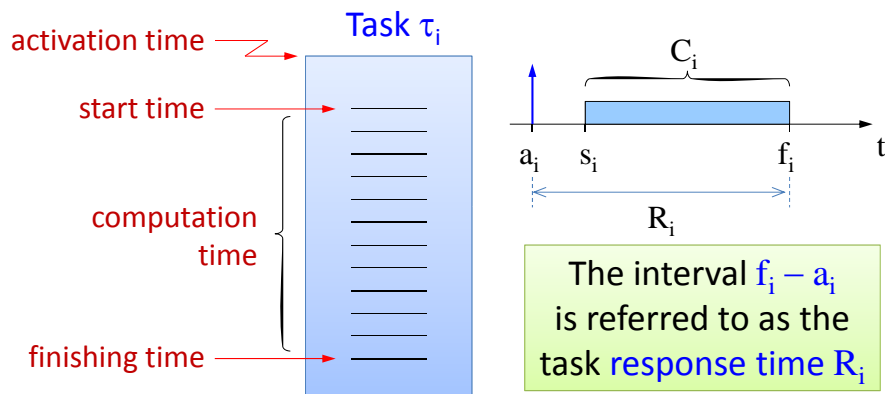
Sources of non determinism

- **Architecture**
 - cache, pipelining, interrupts, DMA
- **Operating system**
 - scheduling, synchronization, communication
- **Language**
 - lack of explicit support for time
- **Design methodologies**
 - lack of analysis and verification techniques

**Modelling
real-time tasks**

Task

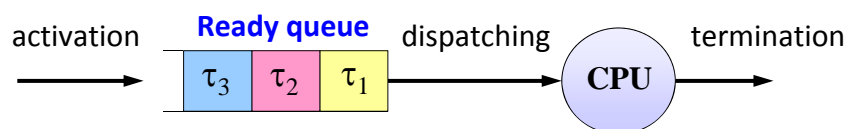
- Sequence of instructions that in the absence of other activities is continuously executed by the processor until completion.



Ready queue

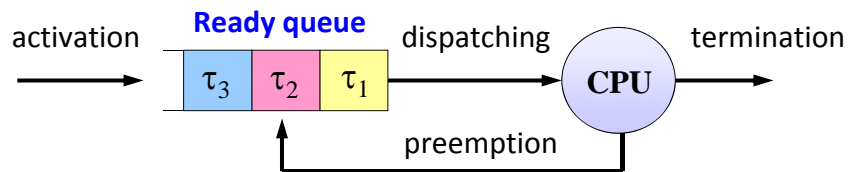
In a single processor system more tasks can be ready to run, but only one can be in execution.

- Ready tasks are kept in a ready queue, ordered by a scheduling policy.
- The processor is assigned to the first task in the queue through a dispatching operation.



Preemption

It is a kernel mechanism that allows to suspend the running task in favor of a more important task.



- Preemption allows reducing the response times of high priority tasks.
- It can be temporarily disabled to ensure consistency of certain critical operations.

Schedule

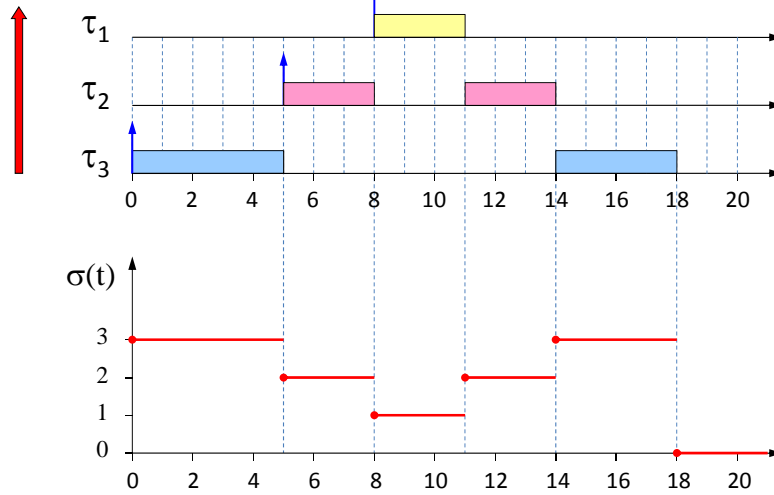
It is a particular task execution sequence:

Formally, given a task set $\Gamma = \{\tau_1, \dots, \tau_n\}$, a schedule is a function $\sigma: \mathbb{R}^+ \rightarrow \mathbb{N}$ that associates an integer k to each interval of time $[t, t+1)$ with the following meaning:

$$\left\{ \begin{array}{ll} k = 0 & \longrightarrow \text{in } [t, t+1) \text{ the processor is IDLE} \\ k > 0 & \longrightarrow \text{in } [t, t+1) \text{ the processor executes } \tau_k \end{array} \right.$$

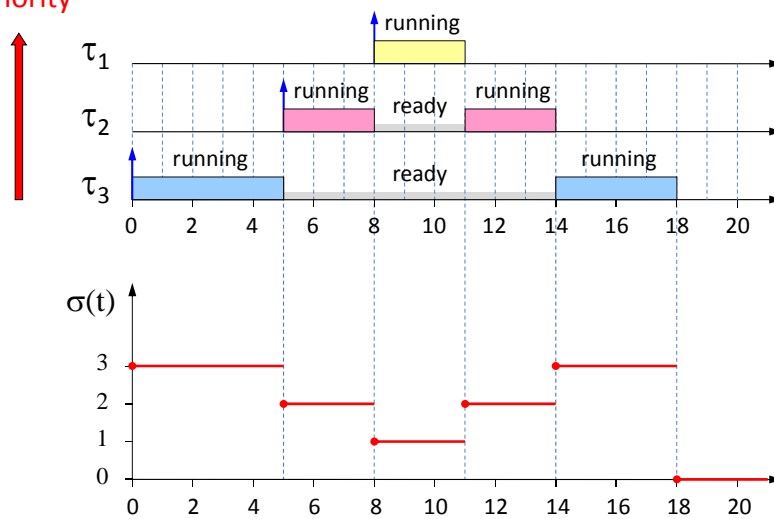
Preemptive schedule

priority

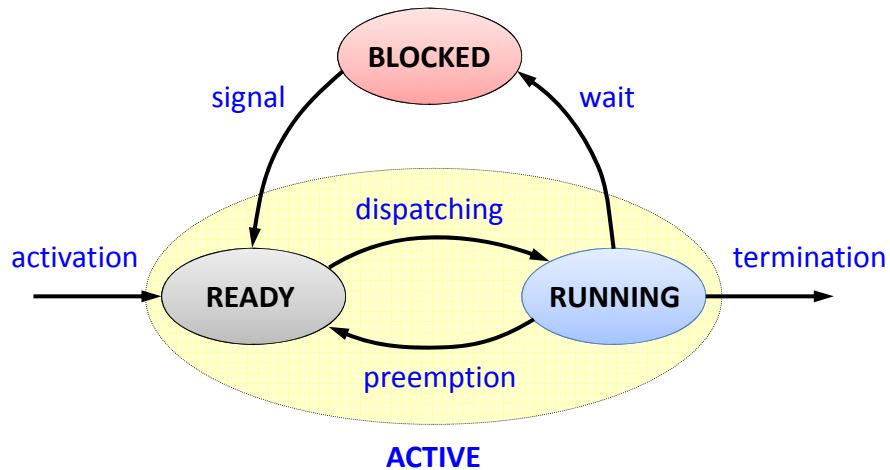


Task states

priority

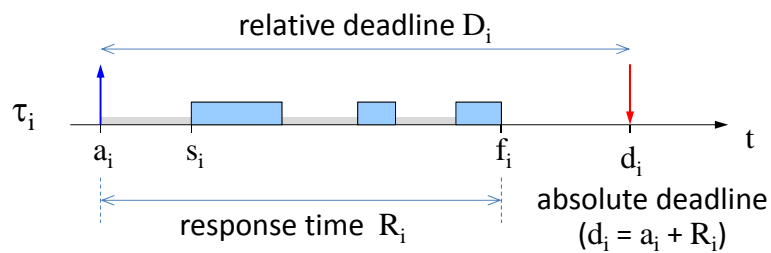


Task states



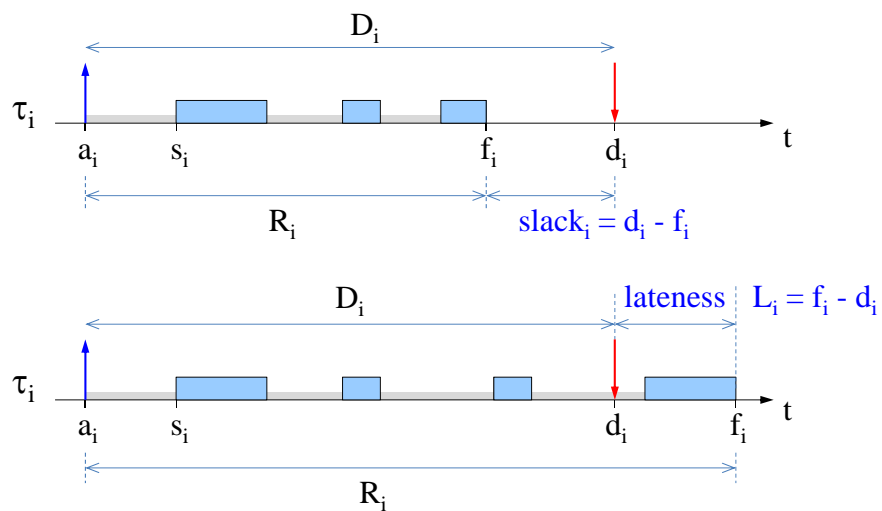
Real-Time Task

- It is a task characterized by a timing constraint on its response time, called deadline:



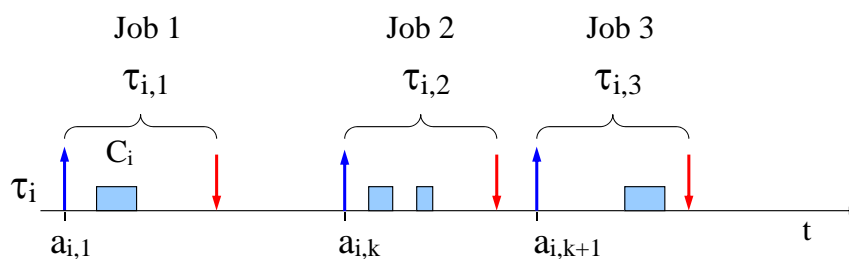
A real-time task τ_i is said to be feasible if it completes within its absolute deadline, that is, if $f_i \leq d_i$, or equivalently, if $R_i \leq D_i$

Slack and Lateness

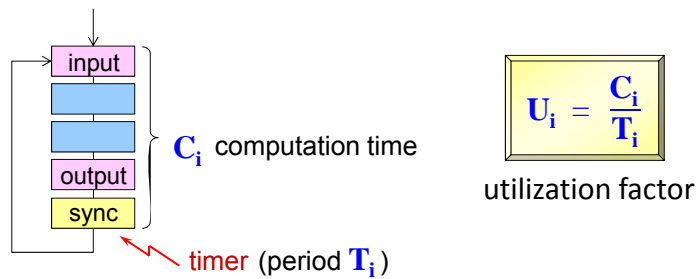


Tasks and jobs

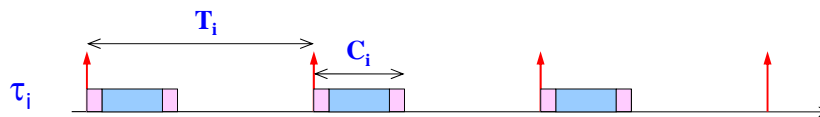
A task running several times on different input data generates a sequence of instances (**jobs**):



Periodic tasks

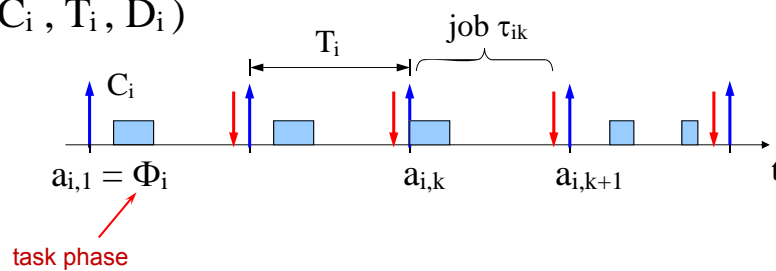


- A periodic task τ_i generates an infinite sequence of jobs: $\tau_{i1}, \tau_{i2}, \dots, \tau_{ik}$ (same code on different data):



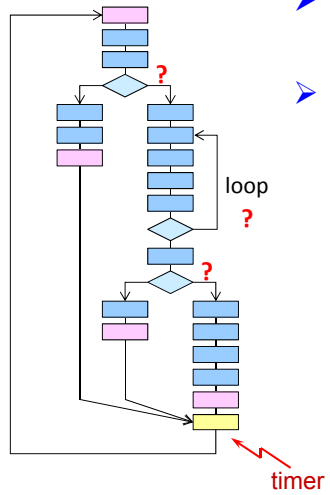
Periodic task model

$\tau_i(C_i, T_i, D_i)$

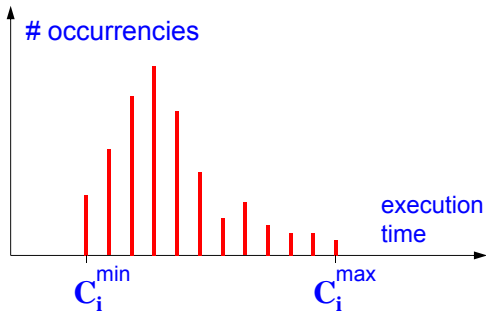


$$\begin{aligned}
 a_{i,k} &= \Phi_i + (k-1) T_i \\
 d_{i,k} &= a_{i,k} + D_i
 \end{aligned}
 \left[\begin{array}{l} \text{often} \\ D_i = T_i \end{array} \right]$$

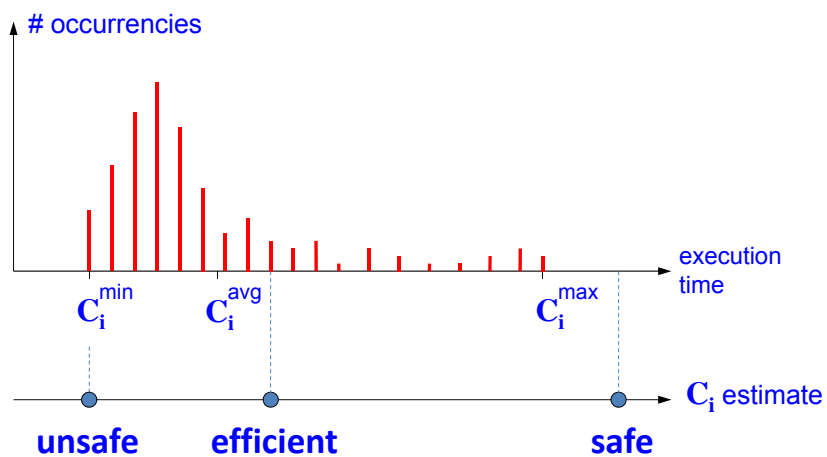
Estimating C_i is not easy



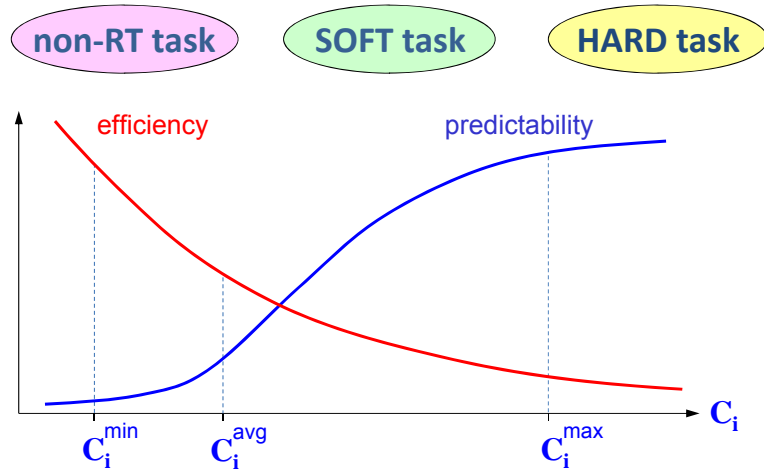
- Each job operates on different data and can take different paths.
- Even for the same data, computation time depends on the processor state (cache, prefetch queue, number of preemptions).



Predictability vs. Efficiency



Predictability vs. Efficiency

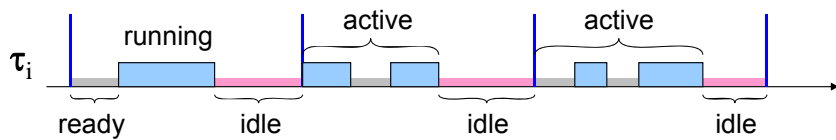


Support for periodic tasks

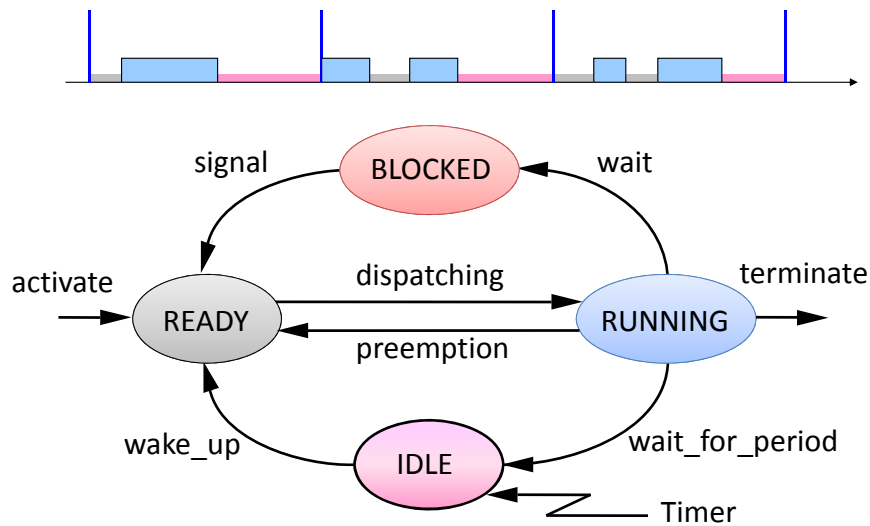
task τ_i

```

while (condition) {
    =====
    =====
    =====
    =====
    =====
    =====
    wait_for_period();
}
    
```

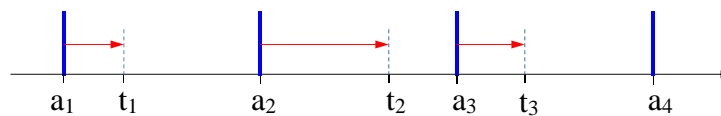


The IDLE state



Jitter

It is a measure of the time variation of a periodic event:

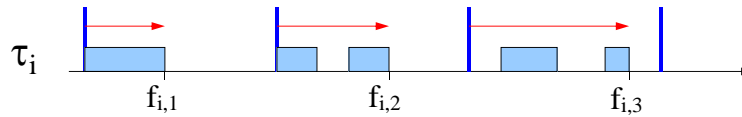


Absolute: $\max_k (t_k - a_k) - \min_k (t_k - a_k)$

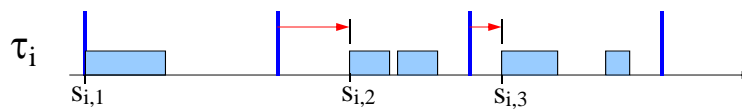
Relative: $\max_k | (t_k - a_k) - (t_{k-1} - a_{k-1}) |$

Types of Jitter

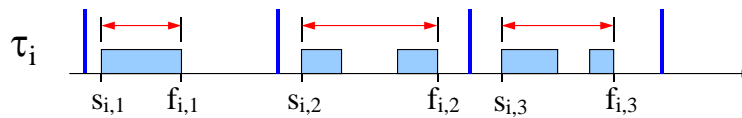
Finishing-time Jitter



Start-time Jitter



Completion-time Jitter (I/O Jitter)



**Where timing
constraints
come from?**

Timing constraints

They can be explicit or implicit.

- **Explicit timing constraints**

They are directly included in the system specifications.

Examples

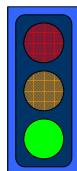
- open the valve **in** 10 seconds
- send the position **within** 40 ms
- read the altimeter **every** 200 ms
- acquire the camera **every** 20 ms

Implicit timing constraints

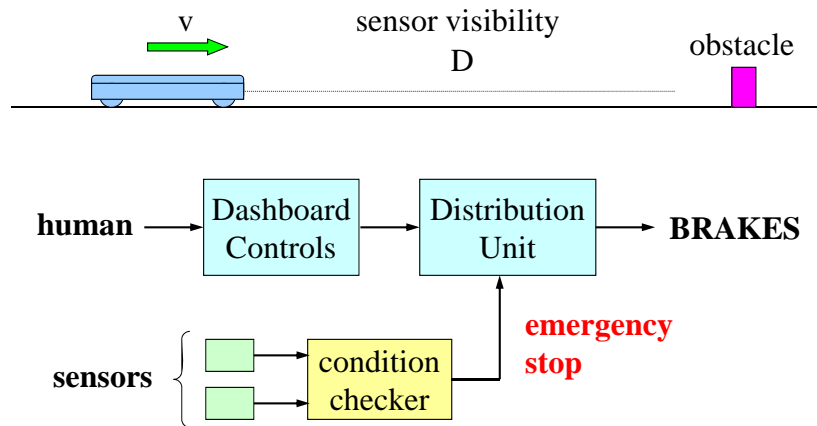
They do not appear in the system specification, but they need to be met to satisfy the performance requirements.

Example

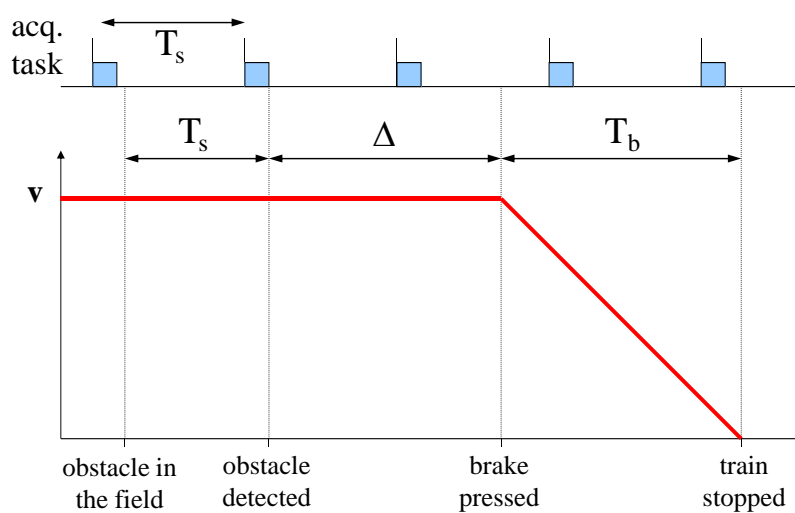
What is the time validity of a sensory data?



Example: automatic braking



Worst-case reasoning



D = sensor visibility

$$v(T_s + \Delta) + X_b < D$$

$$\begin{cases} X_b = vt - \frac{1}{2}at^2 \\ v = at \end{cases}$$

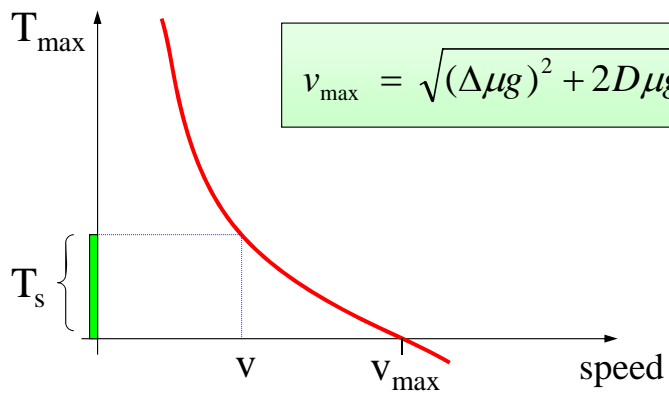
$$a = \mu g$$

$$X_b = \frac{v^2}{2\mu g}$$

$$v(T_s + \Delta) + \frac{v^2}{2\mu g} < D$$

45


$$T_s < \frac{D}{v} - \frac{v}{2\mu g} - \Delta$$



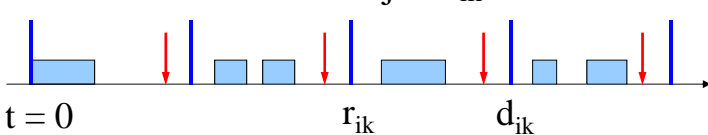
$$v_{\max} = \sqrt{(\Delta\mu g)^2 + 2D\mu g} - \Delta\mu g$$

46

Real-Time Scheduling Algorithms

 **Problem formulation**

$\tau_i(C_i, T_i, D_i)$ job τ_{ik}



$t = 0$ r_{ik} d_{ik}

For each periodic task τ_i guarantee that:

- each job τ_{ik} is activated at $r_{ik} = (k-1)T_i$
- each job τ_{ik} completes within $d_{ik} = r_{ik} + D_i$

48

Timeline Scheduling

It has been used for 30 years in military systems, navigation, and monitoring systems.

Examples

- Air traffic control systems
- Space Shuttle
- Boeing 777
- Airbus navigation system

Timeline Scheduling

Method

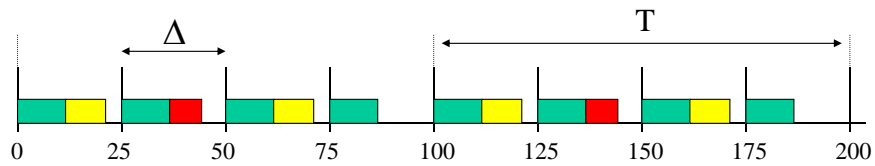
- The time axis is divided in intervals of equal length (*time slots*).
- Each task is statically allocated in a slot in order to meet the desired request rate.
- The execution in each slot is activated by a timer.

Example

task	f	T
A	40 Hz	25 ms
B	20 Hz	50 ms
C	10 Hz	100 ms

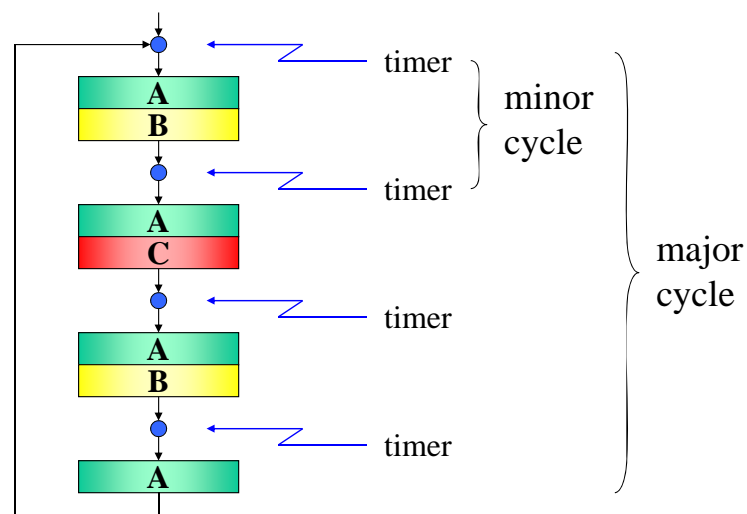
$\Delta = \text{GCD}$ (minor cycle)

$T = \text{lcm}$ (major cycle)



Guarantee: $\begin{cases} C_A + C_B \leq \Delta \\ C_A + C_C \leq \Delta \end{cases}$

Implementation



Advantages

- Simple implementation (no real-time operating system is required).
- Low run-time overhead.
- It allows jitter control.

Disadvantages

- It is not robust during overloads.
- It is difficult to expand the schedule.
- It is not easy to handle aperiodic activities.

Problems during overloads

What do we do during task overruns?

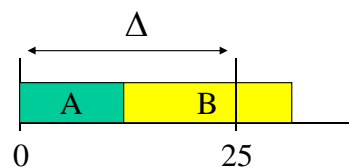
- Let the task continue
 - we can have a **domino effect** on all the other tasks (timeline break)
- Abort the task
 - the system can remain in inconsistent states.

55

Expandability

If one or more tasks need to be upgraded, we may have to re-design the whole schedule again.

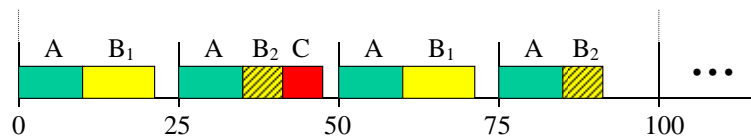
Example: B is updated but $C_A + C_B > \Delta$



56

Expandibility

- We have to split task B in two subtasks (B_1 , B_2) and re-build the schedule:



$$\text{Guarantee: } \begin{cases} C_A + C_{B1} \leq \Delta \\ C_A + C_{B2} + C_C \leq \Delta \end{cases}$$

57

Expandibility

If the frequency of some task is changed, the impact can be even more significant:

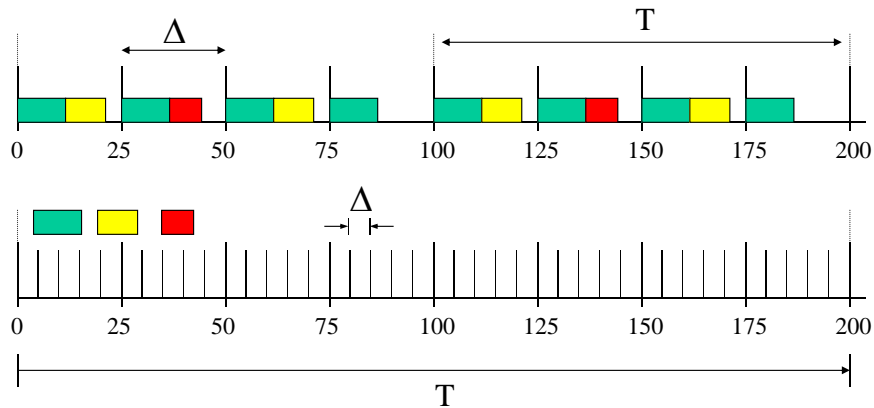
task	T	T
A	25 ms	25 ms
B	50 ms	40 ms
C	100 ms	100 ms

before after

minor cycle: $\Delta = 25$ $\Delta = 5$ $\left(\begin{array}{l} 40 \text{ sync.} \\ \text{per cycle!} \end{array} \right)$
 major cycle: $T = 100$ $T = 200$

58

Example



59

Priority Scheduling

Method

- Each task is assigned a priority based on its timing constraints.
- We verify the feasibility of the schedule using analytical techniques.
- Tasks are executed on a priority-based kernel.

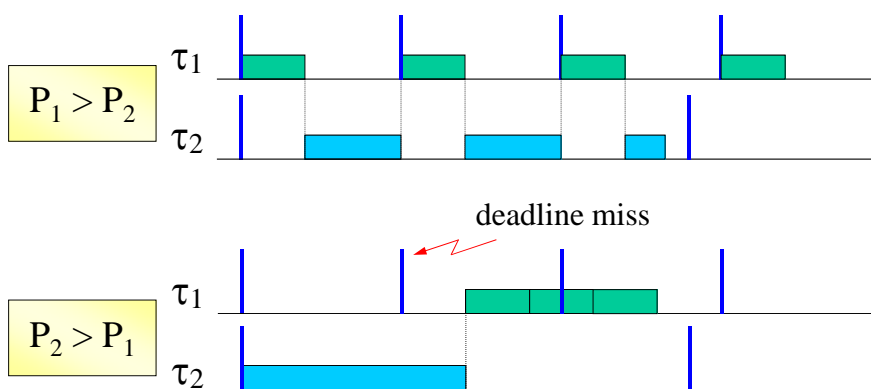
60

How to assign priorities?

- Typically, task priorities are assigned based on their relative importance.
- However, different priority assignments can lead to different processor utilization bounds.

Priority vs. importance

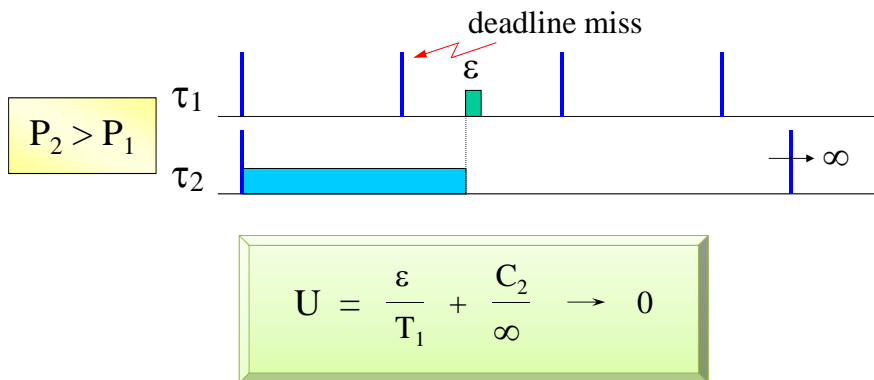
If τ_2 is more important than τ_1 and is assigned higher priority, the schedule may not be feasible:



Priority vs. importance

But the utilization bound can be arbitrarily small:

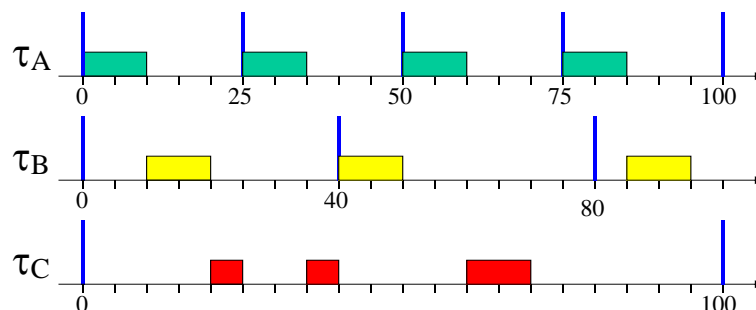
An application can be unfeasible even when the processor is almost empty!



63

Rate Monotonic (RM)

- Each task is assigned a fixed priority proportional to its rate [Liu & Layland '73].



64

Rate Monotonic is optimal

RM is **optimal** among all fixed priority algorithms (if $D_i = T_i$):

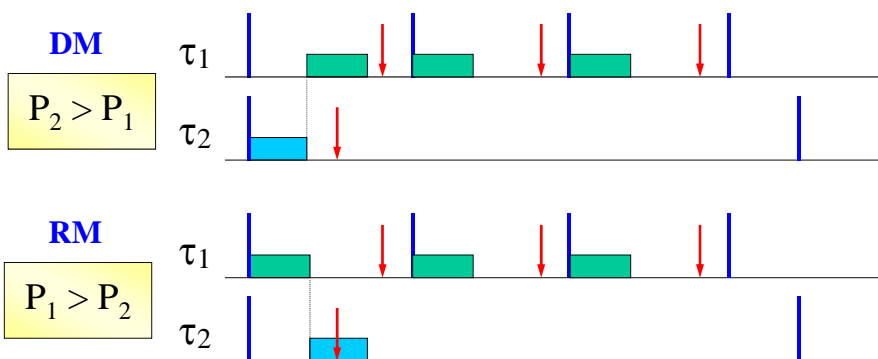
If there exists a fixed priority assignment which leads to a feasible schedule, then the RM schedule is feasible.



If a task set is not schedulable by RM, then it cannot be scheduled by any fixed priority assignment.

Deadline Monotonic is optimal

If $D_i \leq T_i$ then the **optimal** priority assignment is given by **Deadline Monotonic (DM)**:



Priority Assignments

- **Rate Monotonic (RM):** $P_i \propto 1/T_i$ (static) [optimal among FP alg^s
for $T = D$]
- **Deadline Monotonic (DM):** $P_i \propto 1/D_i$ (static) [optimal among FP alg^s
for $T \leq D$]
- **Earliest Deadline First (EDF):** $P_i \propto 1/d_{i,k}$ (dynamic) [optimal among
all alg^s]
 $d_{i,k} = r_{i,k} + D_i$

How can we verify feasibility?

- Each task uses the processor for a fraction of time:

$$U_i = \frac{C_i}{T_i}$$

- Hence the total **processor utilization** is:

$$U_p = \sum_{i=1}^n \frac{C_i}{T_i}$$

- U_p is a measure of the **processor load**

A necessary condition

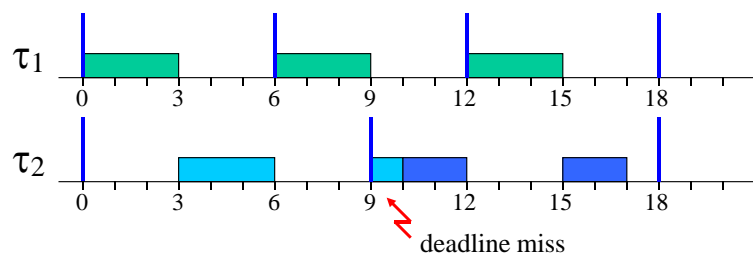
A necessary condition for having a feasible schedule is that $U_p \leq 1$.

In fact, if $U_p > 1$ the processor is overloaded hence the task set cannot be schedulable.

However, there are cases in which $U_p \leq 1$ but the task set is not schedulable by RM.

An unfeasible RM schedule

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



Basic results

In 1973, Liu & Layland proved that a set of n periodic tasks can be feasibly scheduled

$$\left\{ \begin{array}{ll} \text{under RM} & \text{if } \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \\ \text{under EDF} & \text{if and only if } \sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \end{array} \right.$$

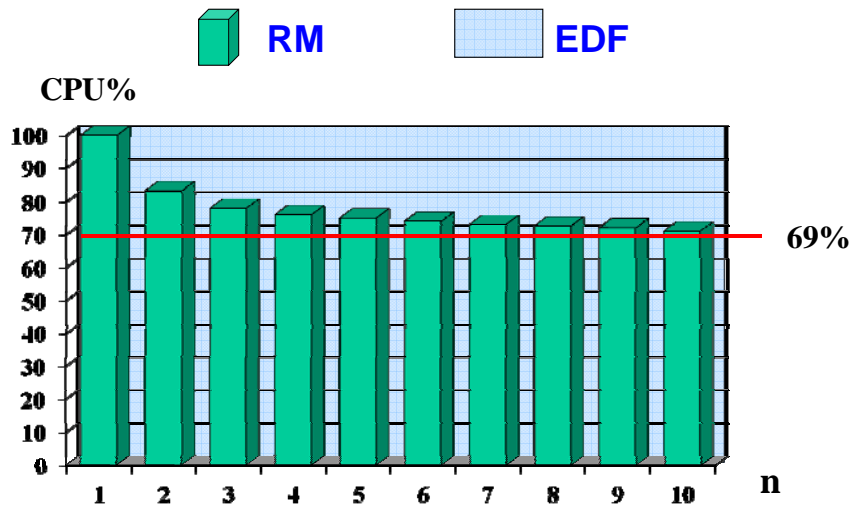
Assumptions: $\left\{ \begin{array}{l} \text{Independent tasks} \\ \Phi_i = 0 \quad D_i = T_i \end{array} \right.$

Utilization bound for large n

$$U_{\text{lub}}^{RM} = n(2^{1/n} - 1)$$

$$\text{for } n \rightarrow \infty \quad U_{\text{lub}} \rightarrow \ln 2$$

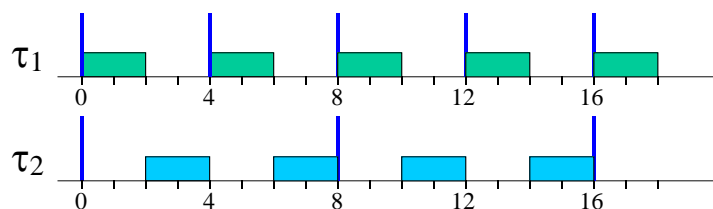
Schedulability bound



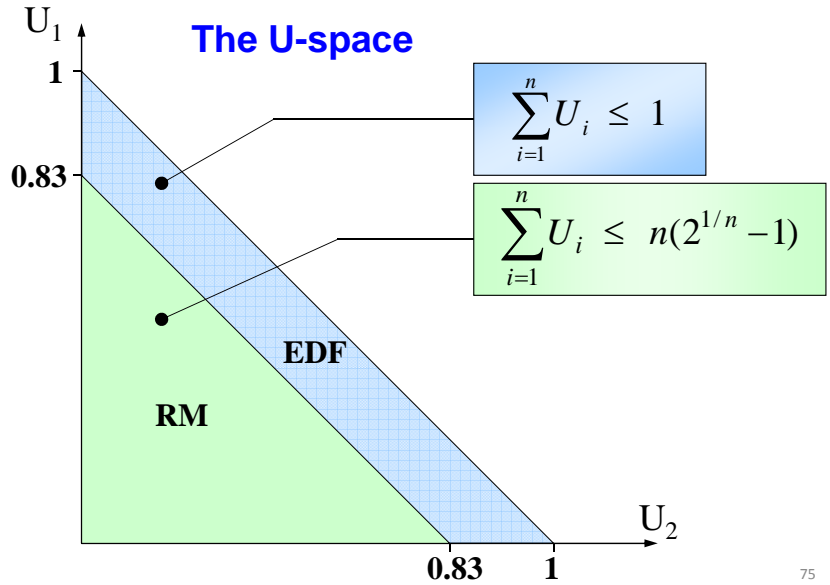
A special case

If tasks have harmonic periods $U_{lub} = 1$.

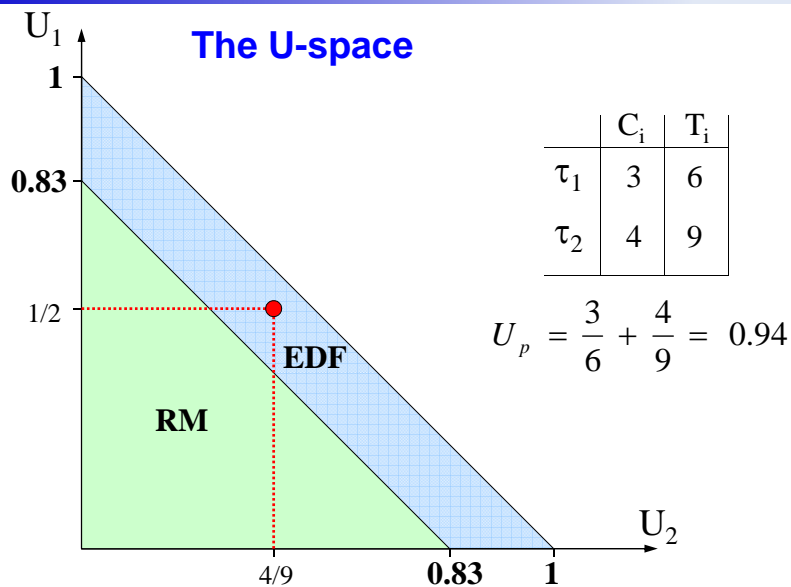
$$U_p = \frac{2}{4} + \frac{4}{8} = 1$$



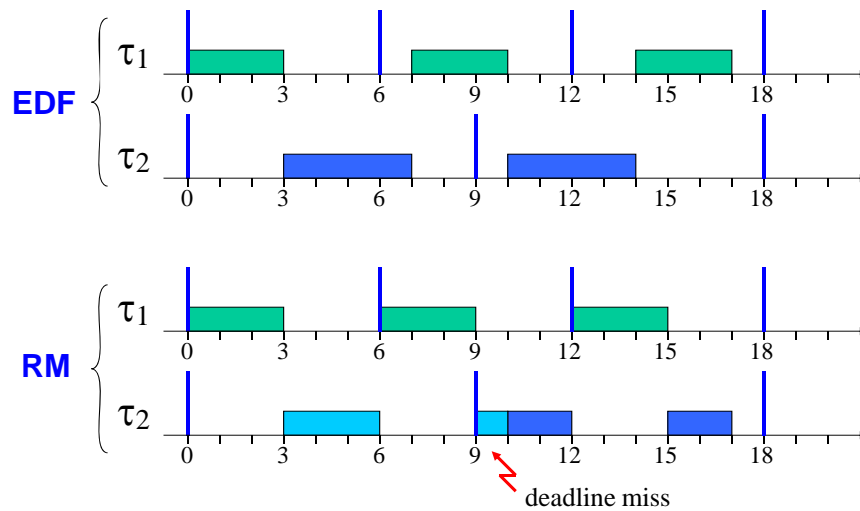
Schedulability region



Schedulability region



Schedule

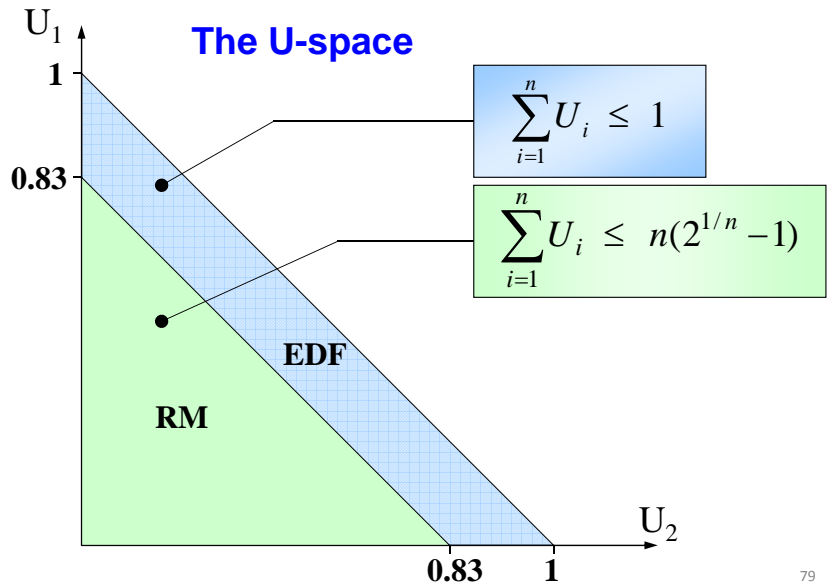


The Hyperbolic Bound

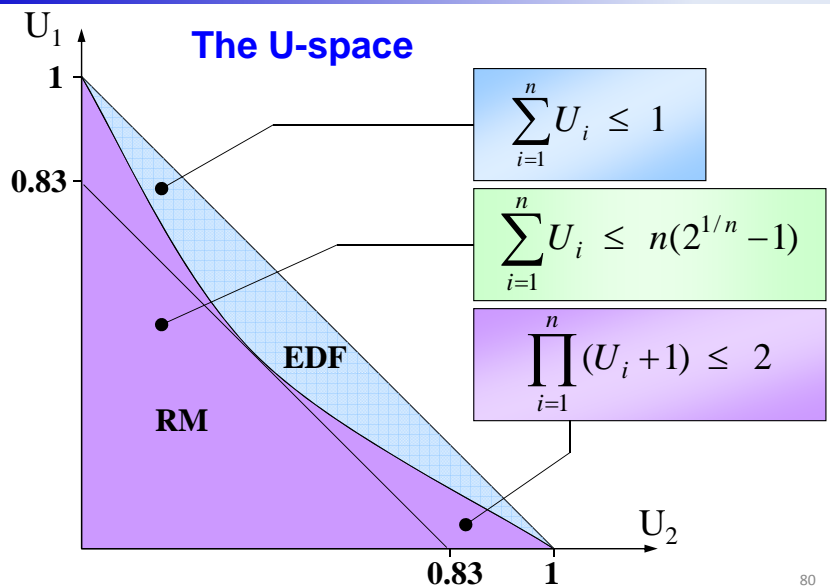
- In 2000, **Bini et al.** proved that a set of n periodic tasks is schedulable with RM if:

$$\prod_{i=1}^n (U_i + 1) \leq 2$$

Schedulability region



Schedulability region



Response Time Analysis

1. For each task τ_i compute the interference due to higher priority tasks:

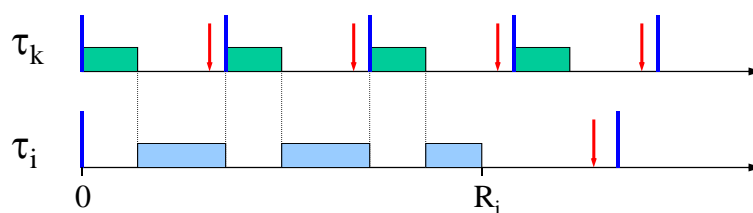
$$I_i = \sum_{P_k > P_i} C_k$$

2. compute its response time as

$$R_i = C_i + I_i$$

3. verify whether $R_i \leq D_i$

Computing the interference



Interference of τ_k on τ_i
in the interval $[0, R_i]$:

$$I_{ik} = \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Interference of high
priority tasks on τ_i :

$$I_i = \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

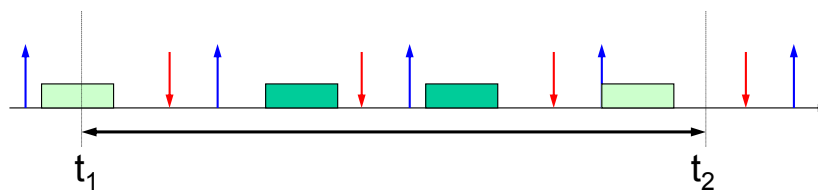
Computing the response time

$$R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Iterative solution:

$$\begin{cases} R_i^0 = C_i \\ R_i^s = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_k} \right\rceil C_k \end{cases} \quad \begin{array}{l} \text{iterate until} \\ R_i^s > R_i^{(s-1)} \end{array}$$

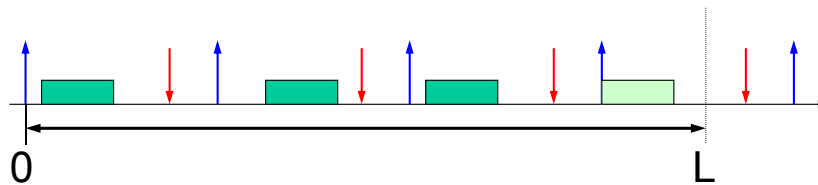
Processor Demand



The processor demand in $[t_1, t_2]$ is the computation time of those jobs started at $r_{ik} \geq t_1$ with deadline $d_{ik} \leq t_2$:

$$g(t_1, t_2) = \sum_{\substack{d_i \leq t_2 \\ r_i \geq t_1}} C_i$$

Processor Demand



Processor Demand in $[0, L]$

$$g(0, L) = \sum_{i=1}^n \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i$$

85

Processor Demand Test

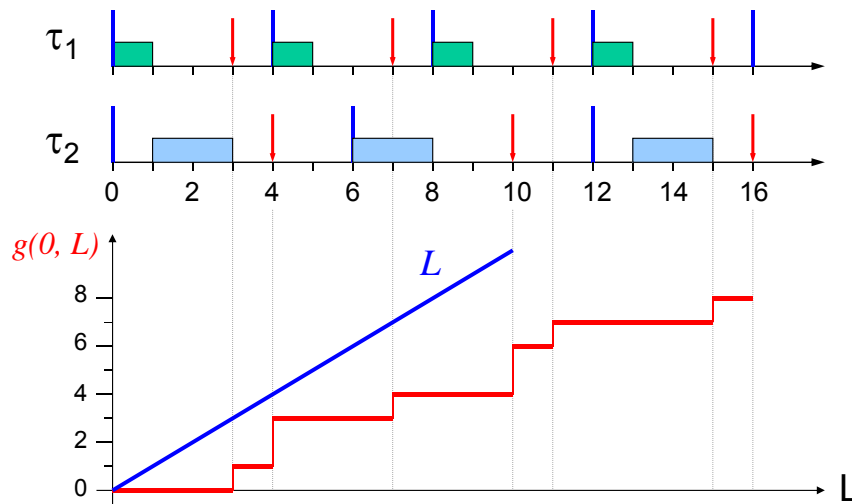
$$\forall L > 0, \quad g(0, L) \leq L$$

Question

How can we bound the number of intervals in which the test has to be performed?

86

Example



87

Bounding complexity

- Since $g(0, L)$ is a step function, we can check feasibility only at deadline points.
- If tasks are synchronous and $U_p < 1$, we can check feasibility up to the [hyperperiod](#) H :

$$H = \text{lcm}(T_1, \dots, T_n)$$

88

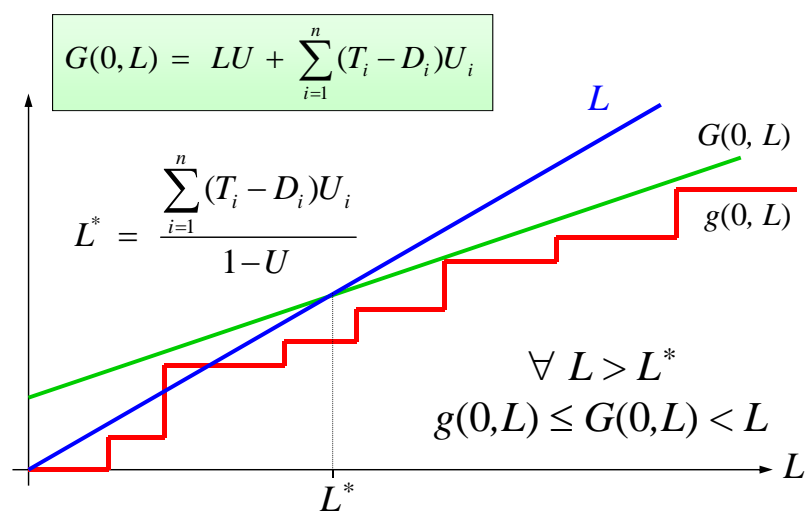
Bounding complexity

- Moreover we note that: $g(0, L) \leq G(0, L)$

$$\begin{aligned}
 G(0, L) &= \sum_{i=1}^n \left(\frac{L + T_i - D_i}{T_i} \right) C_i \\
 &= \sum_{i=1}^n L \frac{C_i}{T_i} + \sum_{i=1}^n (T_i - D_i) \frac{C_i}{T_i} \\
 &= LU + \sum_{i=1}^n (T_i - D_i) U_i
 \end{aligned}$$

89

Limiting L



90

Processor Demand Test

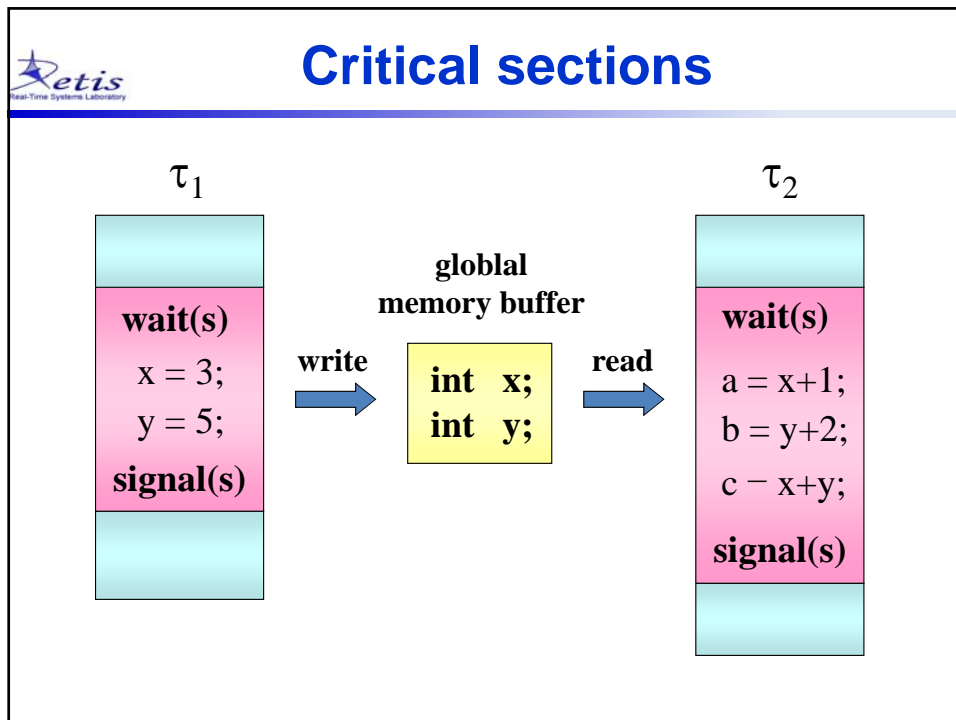
$$\forall L \in D, \quad g(0, L) \leq L$$

$$D = \{d_k \mid d_k \leq \min(H, L^*)\}$$

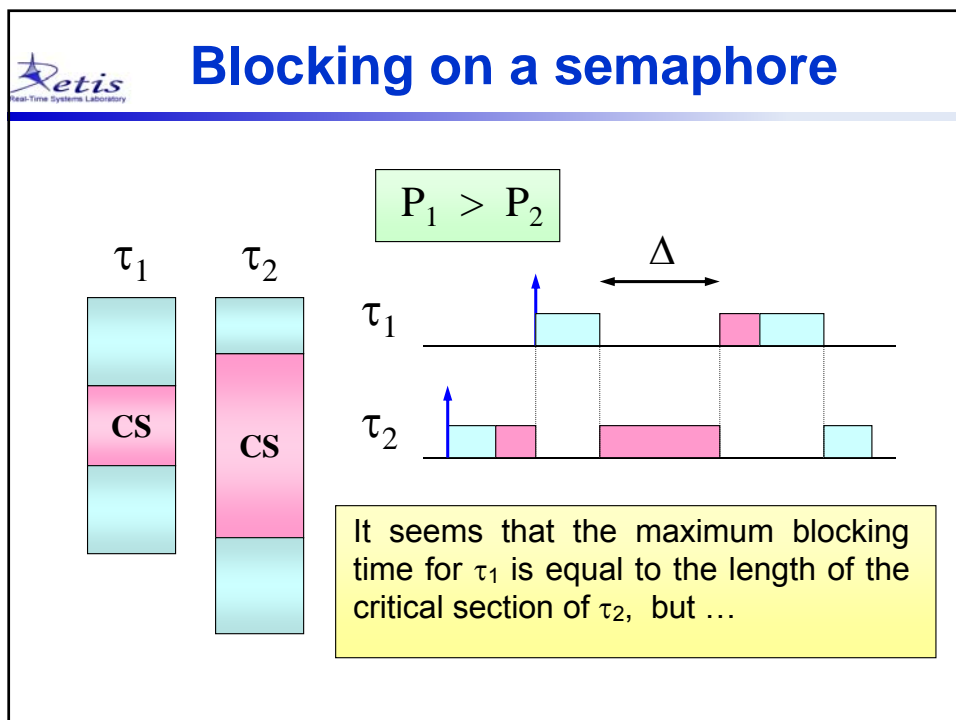
$$\left\{ \begin{array}{l} H = \text{lcm}(T_1, \dots, T_n) \\ L^* = \frac{\sum_{i=1}^n (T_i - D_i) U_i}{1 - U} \end{array} \right.$$

Handling Shared Resources

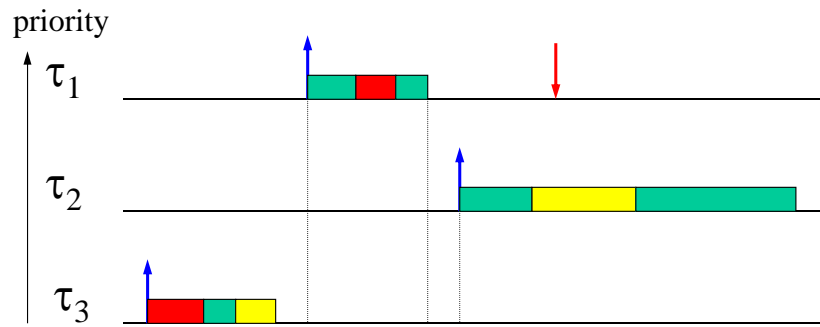
Critical sections



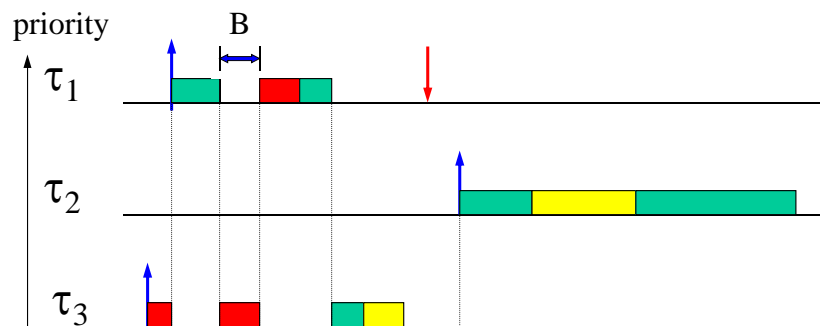
Blocking on a semaphore



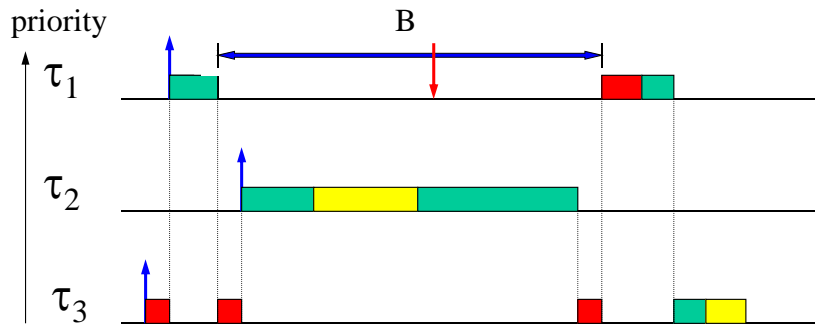
Schedule with no conflicts



Conflict on a critical section



Conflict on a critical section



Priority Inversion

A high priority task is blocked by a lower-priority task a for an unbounded interval of time.

Solution

Introduce a concurrency control protocol for accessing critical sections.

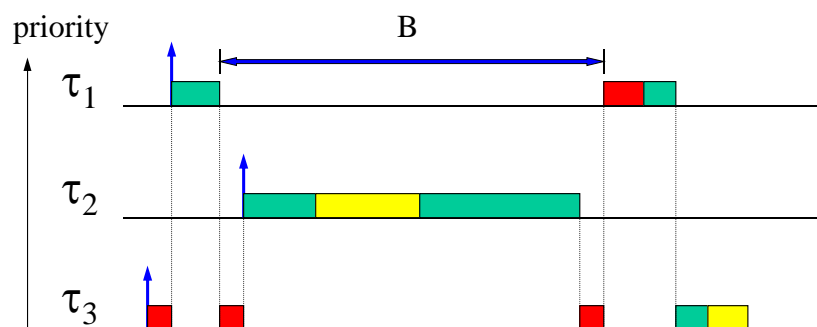
Non Preemptive Protocol

- Preemption is forbidden in critical sections.
- Implementation: when a task enters a CS, its priority is increased at the maximum value.

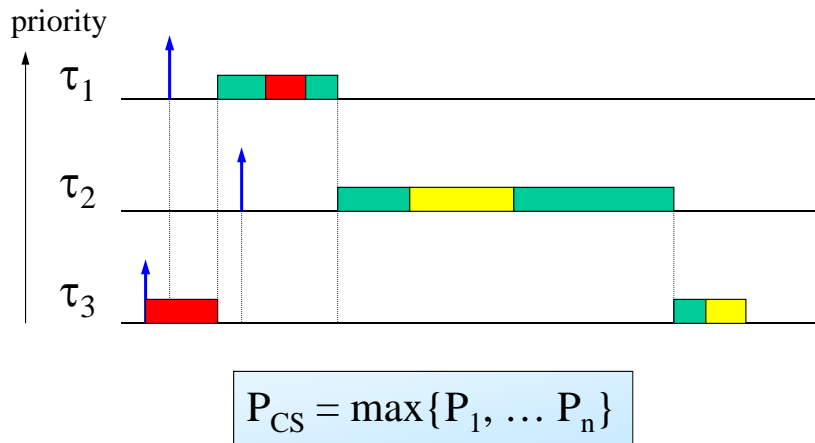
ADVANTAGES: simplicity

PROBLEMS: high priority tasks that do not use the same resources may also block

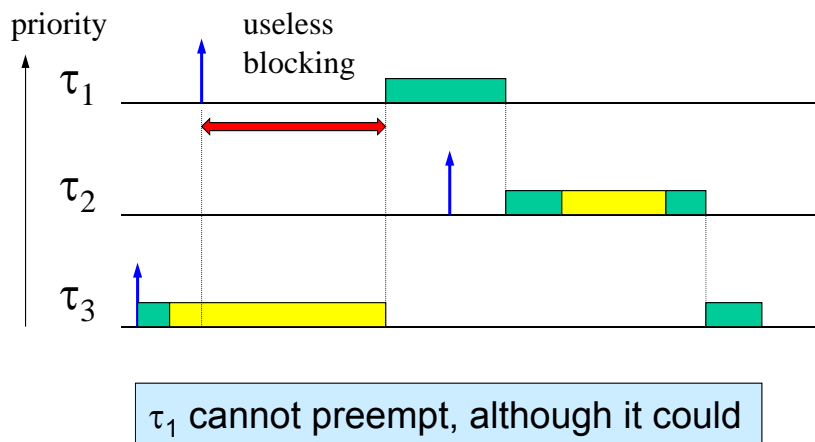
Conflict on a critical section



Schedule with NPP



Problem with NPP



Highest Locker Priority

A task entering a resource R_k gets the highest priority among the tasks that use R_k

Implementation:

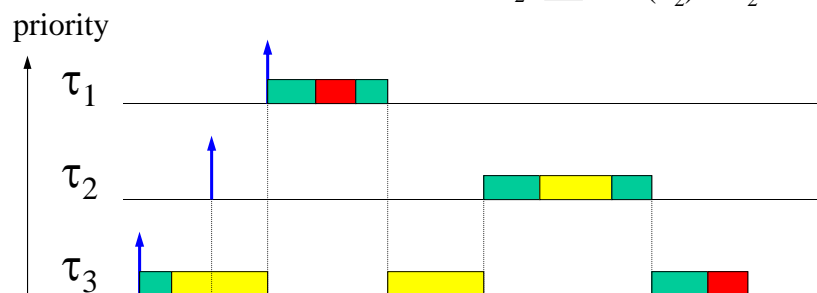
- Each task τ_i has a dynamic priority p_i initialized to P_i
- Each semaphore S_k has a ceiling

$$C(S_k) = \max \{P_i \mid \tau_i \text{ uses } S_k\}$$

- When τ_i locks S_k , p_i is increased to $C(S_k)$
- When τ_i unlocks S_k , its priority goes back to P_i

Schedule with HLP

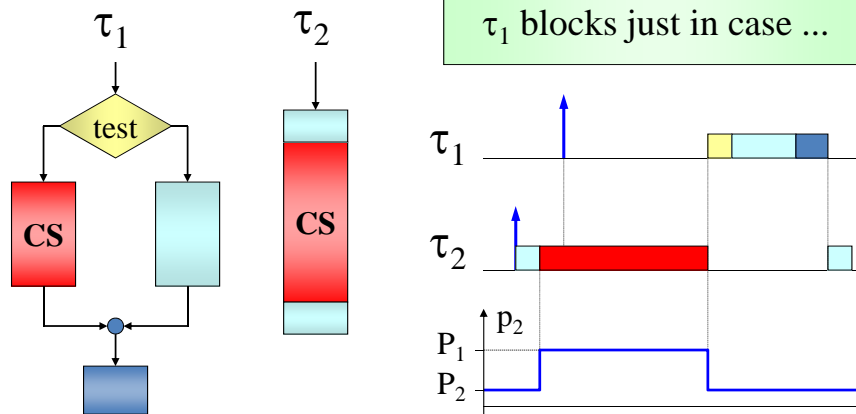
S_1 ■ $C(S_1) = P_1$
 S_2 ■ $C(S_2) = P_2$



τ_2 is blocked, but τ_1 can preempt τ_3 within its critical section, because $P_1 > C(S_2)$

Problem with NPP and HLP

A task is blocked when attempting to preempt, not when accessing the resource.



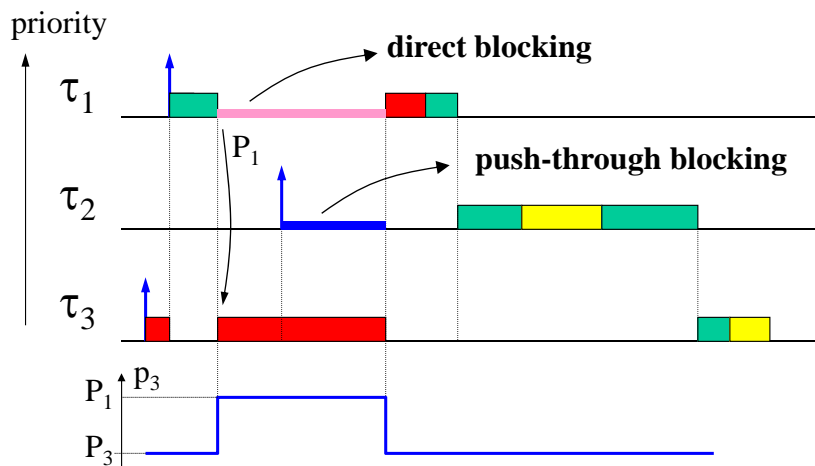
Priority Inheritance Protocol

[Sha, Rajkumar, Lehoczky, 90]

- A task increases its priority only if it blocks other tasks.
- A task τ_i in a resource R_k inherits the highest priority among those tasks it blocks.

$$p_i(R_k) = \max \{P_h \mid \tau_h \text{ blocked on } R_k\}$$

Schedule with PIP



Types of blocking

- **Direct blocking**
A task blocks on a locked semaphore
- **Push-through blocking**
A task blocks because a lower priority task inherited a higher priority.

BLOCKING:
a delay caused by a lower priority task

Identifying blocking resources

- A task τ_i can be blocked by those semaphores used by lower priority tasks
 - directly shared with τ_i (**direct blocking**)
 - shared with tasks having priority higher than τ_i (**push-through blocking**).

Theorem: τ_i can be blocked at most once by each of such semaphores

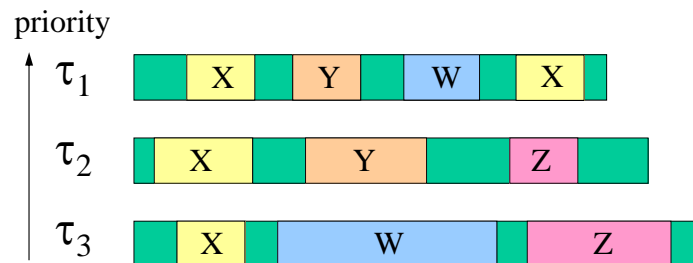
Theorem: τ_i can be blocked at most once by each lower priority task

Bounding blocking times

- Let n_i be the number of tasks with priority less than τ_i
- Let m_i be the number of semaphores that can block τ_i

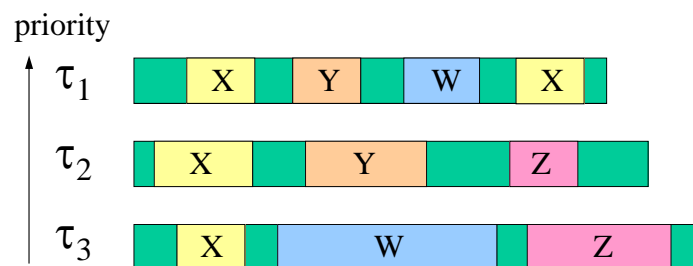
Theorem: τ_i can be blocked at most on the duration of $\alpha_i = \min(n_i, m_i)$ critical sections

Example



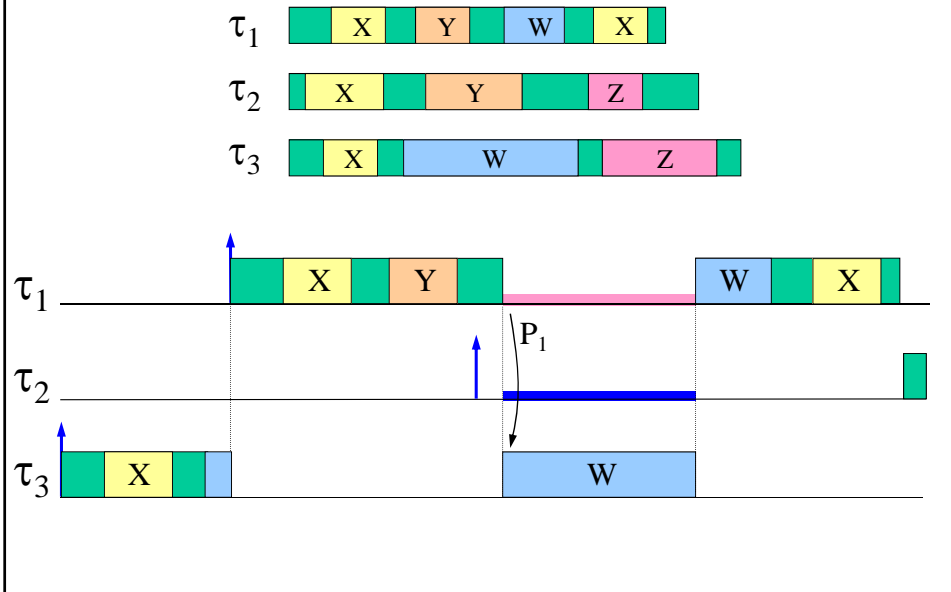
- τ_1 can be blocked once by τ_2 (on X_2 or Y_2) and once by τ_3 (on X_3 or W_3)
- τ_2 can be blocked once by τ_3 (on X_3 , W_3 or Z_3)
- τ_3 cannot be blocked
- NOTE: τ_1 cannot be blocked twice on X

Example

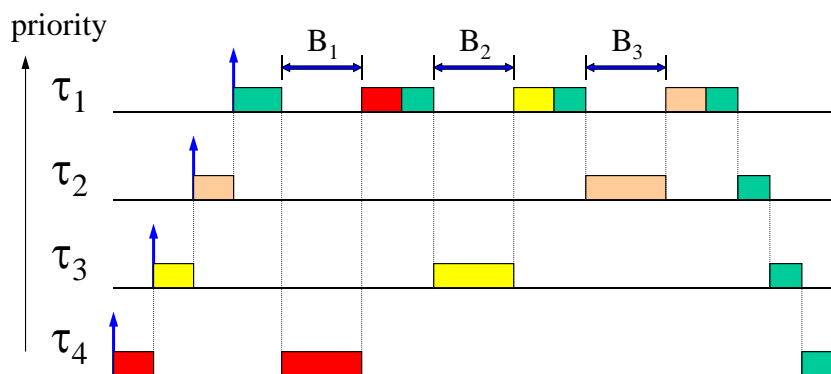


- $B_1 = \delta(Y_2) + \delta(W_3)$
- $B_2 = \delta(W_3)$
- $B_3 = 0$

How can τ_2 be blocked by W_3 ?



Chained blocking with PIP

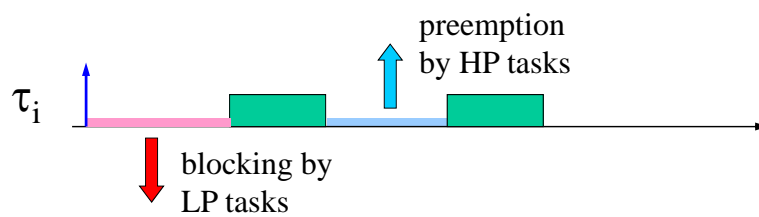


Theorem: τ_i can be blocked at most once by each lower priority task

Comparison

	NPP	HLP	PIP
# of blocking	1	1	$\alpha_i = \min(n_i, m_i)$
chained blocking	no	no	yes
deadlocks avoidance	yes	yes	no
pessimism	very high	high	low
transparency	yes	no	yes
stack sharing	yes	yes	no

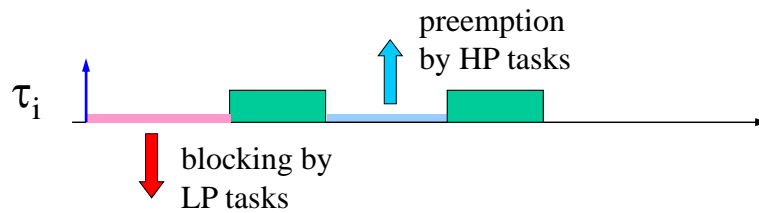
Accounting for blocking times



Utilization test

$$\forall i \quad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1)$$

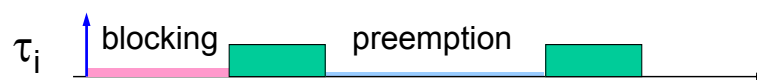
Accounting for blocking times



Hyperbolic bound

$$\forall i \quad \prod_{k=1}^{i-1} \left(\frac{C_k}{T_k} + 1 \right) \left(\frac{C_i + B_i}{T_i} + 1 \right) \leq 2$$

Response Time Analysis



$$\begin{cases} R_i^0 = B_i + C_i \\ R_i^s = B_i + C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_k} \right\rceil C_k \end{cases}$$

iterate until
 $R_i^s > R_i^{(s-1)}$