Modeling real-time activities

What is a model?

A model is a representation of something. It captures not all attributes of the represented thing, but rather only those that are relevant for a specific purpose.

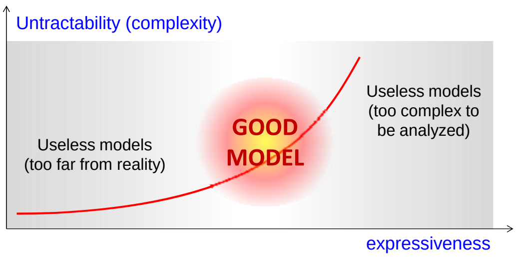
> *"Confusing a model with reality would be like going to a restaurant and eat the menu"*
> *Golomb's Law on mathematical models*

2

What is a good model?

- It should be expressive (an accurate representation of reality)
- It should be tractable (provide results in a bounded time)

Unfortunately, expressiveness and tractability do not get along very well



3

Important aspects

Building a model implies:

- simplifying reality (but not too much), capturing the features of interest;
- defining the variables that characterize the model.
- defining the system interface (variables exposed to the user);
- clearly identifying the assumptions (affecting values);
- defining the metrics for evaluating the outputs of your system and its performance.

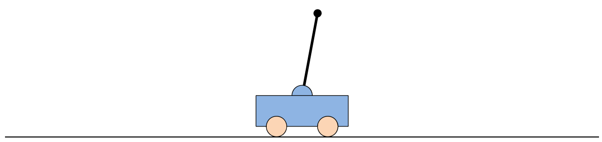
4

Types of variables

- Parameters (variables you don't want to change);
- Input variables (commands given by the user/controller)
- Design variables (variables you want to identify to apply your control actions);
- State variables (variables describing the system state and behavior);
- Output variables (variables you want to measure to evaluate the performance of your method).

5

Example



- Parameters: Pole length/mass, cart mass
- Input variables: Force applied to the cart
- Design variables: Control parameters (K_P, K_I, K_D)
- State variables: Position/speed of the cart and pole
- Output variables: Pole angle

Elements of computation

Instruction:
It is the elementary entity of a programming language.

Examples in ASM X86:

```
MOV AX, 5;
MOV BX, 7;
ADD  AX, BX;
```

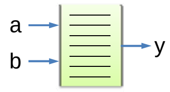
Examples in C:

```
int x;
x = a + b;
if (x > threshold) y = 1;
else y = 0;
```

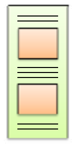
7

Elements of computation

Function:
It is a container for a set of instructions. It may take multiple input arguments and produces a single output.



A function can call other functions:



8

Elements of computation

Task:

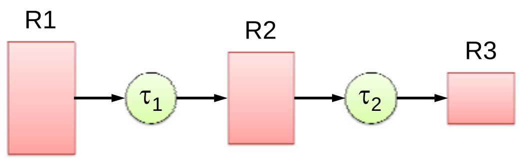
- It is a function performing a given computational activity in a system (e.g., sensory processing, motor control, filtering).
- It is the elementary entity managed by an operating system.
- It may have specific constraints (e.g., activation time, period, deadline, precedence relations with other tasks).
- It can communicate with other tasks by shared resources.

Resource:
It is a set of variables that can be used by tasks to store data or temporary results:

9

Elements of computation

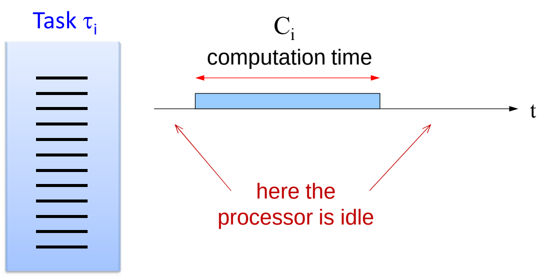
Application:
It consists of a set of tasks interacting through a set of shared resources:



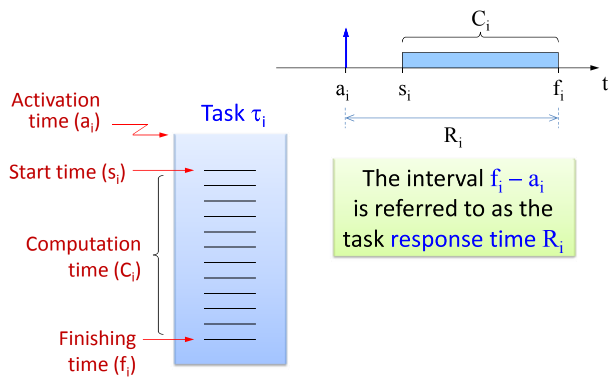
10

Task execution

➢ The execution of a task on a processor is represented by a bar on a timeline.



Task important variables



RTOS responsibilities

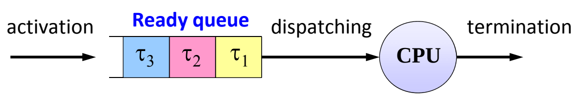
A real-time operating system is responsible for:

- Managing concurrency;
- Activating periodic tasks at the beginning of each period (time management);
- 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 (interrupt handling).

Ready queue

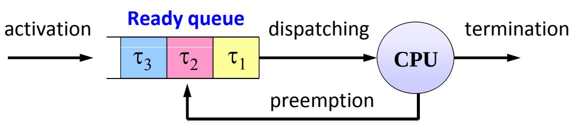
In a concurrent system, more tasks can be simultaneously active, but only one can be in execution (running).

- An active task that is not in execution is said to be ready.
- Ready tasks are kept in a ready queue, managed 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 execution of the running task in favor of a more important task. The suspended task goes back in the ready queue.



- Preemption enhances concurrency and allows reducing the response times of high priority tasks.
- It can be disabled (completely or temporarily) to ensure the consistency of certain critical operations.

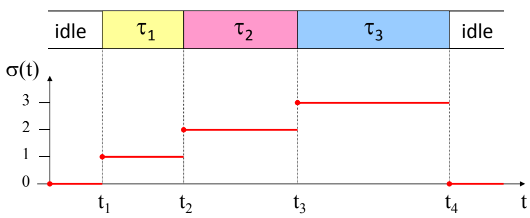
Schedule

Is a particular assignement of tasks to the processor that determines the task execution sequence:

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

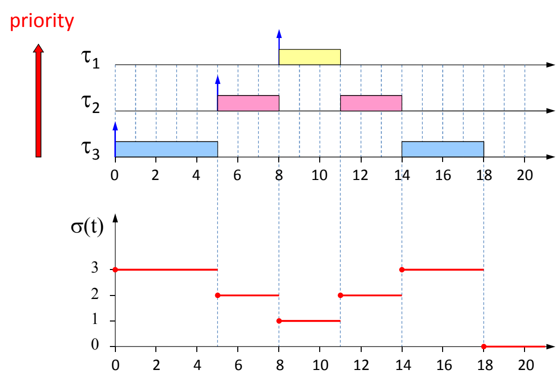
- $k = 0$ ➡ in $[t, t+1)$ the processor is IDLE
- $k > 0$ ➡ in $[t, t+1)$ the processor executes τ_k

Schedule

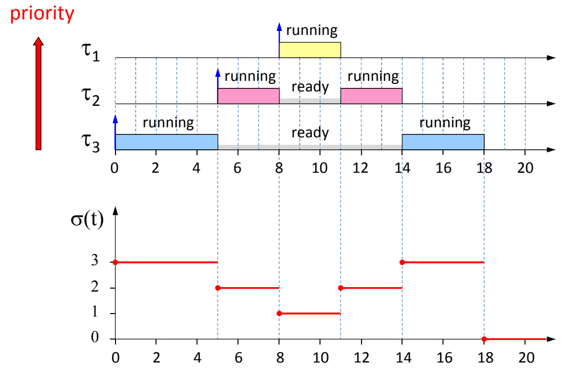


- Each interval $[t_i, t_{i+1})$ is called a **time slice**.
- In time instants t_1, t_2, t_3, t_4 the processor is said to perform a **context switch**.

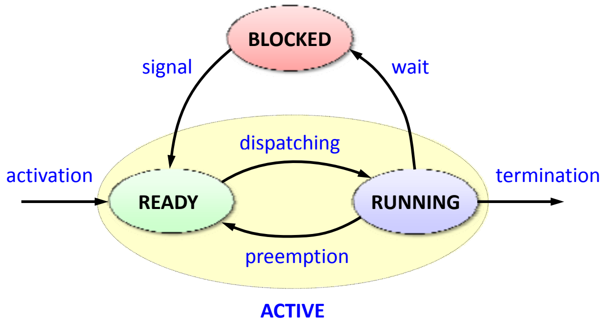
Preemptive schedule



Task states

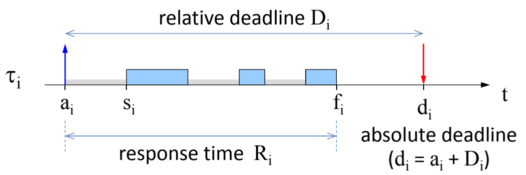


Task states



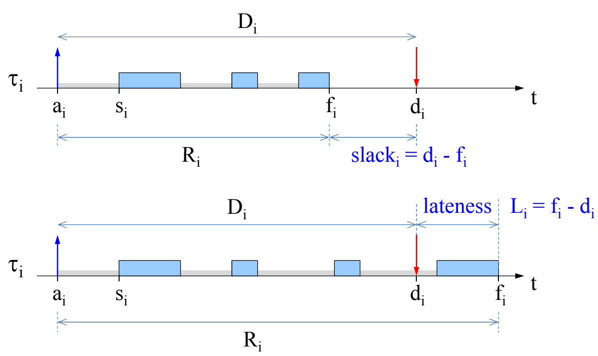
Real-Time Task

- 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 guaranteed to complete within its deadline, that is, if $f_i \leq d_i$, o equivalently, if $R_i \leq D_i$

Slack and Lateness



Performance Metrics

How do we evaluate the performance of a scheduler?

For non real-time task sets:

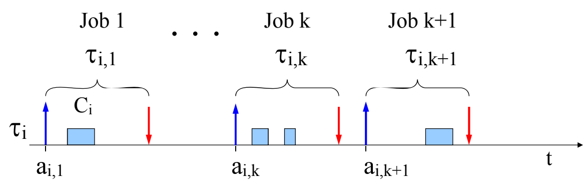
- Average response time: R_{avg} = 1/n SUM R_i
- Total finishing time: $F_{tot} = \max(f_i) - \min(r_i)$

For real-time task sets:

- Maximum lateness: $L_{max} = \max(f_i - d_i)$
- No. of deadline misses: N_{miss} = SUM_i $H(f_i - d_i)$
- Feasibility: $F = H(N_{miss})$

Tasks and jobs

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



Activation mode

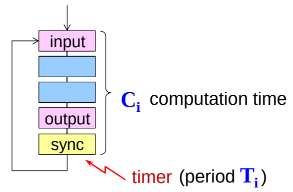
- **Time driven**: (**periodic** tasks)

  A task is automatically activated by the operating system at predefined time instants.

- **Event driven**: (**aperiodic** tasks)

  A task is activated at the arrival of an event (by interrupt or by another task through an explicit system call).

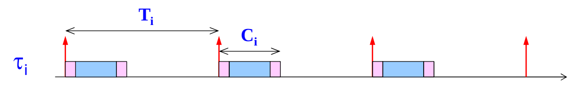
Periodic task



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

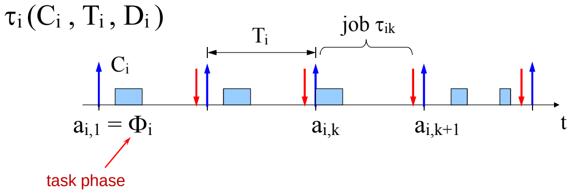
utilization factor

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



Periodic task

$\tau_i (C_i , T_i , D_i)$



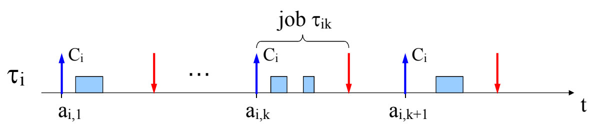
$$a_{i,k} = \Phi_i + (k-1)\, T_i$$
$$d_{i,k} = a_{i,k} + D_i$$

often $D_i = T_i$

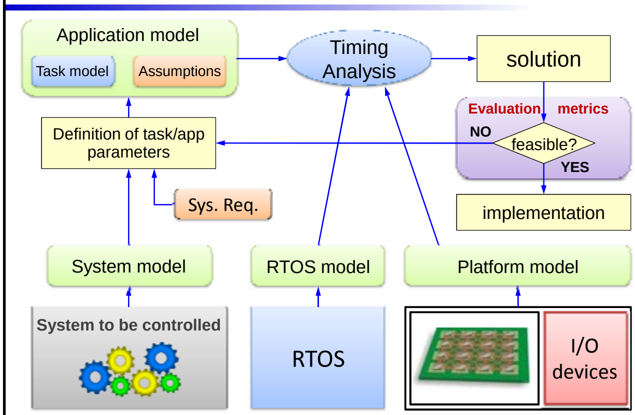
Aperiodic task

- **Aperiodic:** $a_{i,k+1} > a_{i,k}$
- **Sporadic:** $a_{i,k+1} \geq a_{i,k} + T_i$

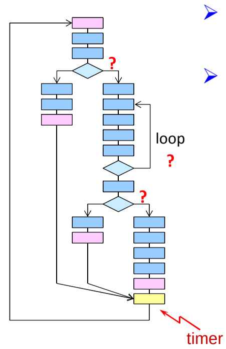
minimum interarrival time



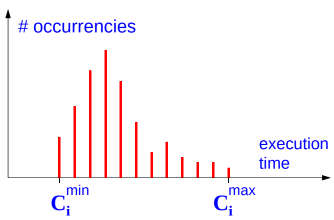
Using models



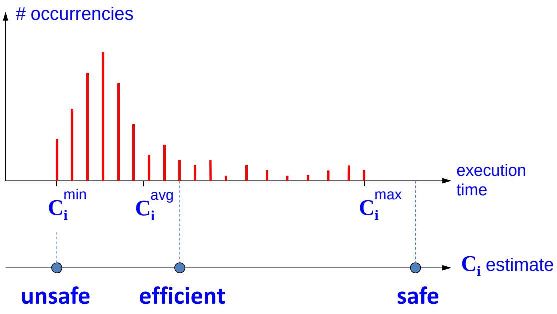
Estimating C_i is not easy



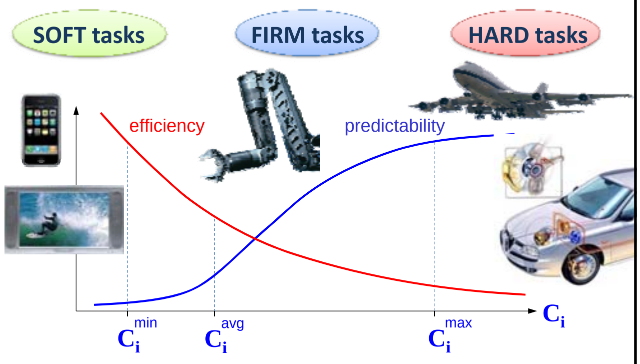
- Each job operates on different data and can take different paths.
- Even for the same data, computation time depends on the processor state (cache, prefetch queue, number of preemptions).



Predictability vs. Efficiency



Predictability vs. Efficiency



Criticality

HARD task
All jobs must meet their deadlines. Missing a single deadline may cause catastrophic effects on the whole system.

FIRM task
Missing a job deadline has not catastrophic effects on the system, but invalidates the execution of that particular job.

SOFT task
Missing a deadline is not critical. A job finishing after its deadline has still some value but causes a performance degradation.

> An operating system able to handle hard real-time tasks is called a **hard real-time** system.

Criticality

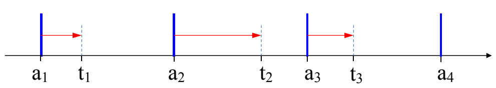
Typical HARD tasks
- sensory acquisition
- low-level control
- sensory-motor planning

Typical FIRM tasks
- RT audio processing
- RT video decoding

Typical SOFT tasks
- reading data from the keyboard
- user command interpretation
- message displaying
- graphical activities

Jitter

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

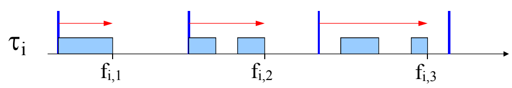


Absolute: $\max_k (t_k - a_k) - \min_k (t_k - a_k)$

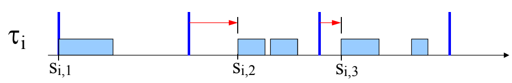
Relative: $\max_k \left| (t_k - a_k) - (f_{k-1} - a_{k-1}) \right|$

Types of Jitter

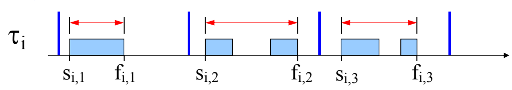
Finishing-time Jitter



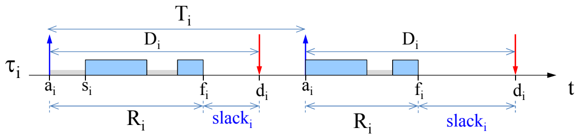
Start-time Jitter



Completion-time Jitter (I/O Jitter)



Parameters summary



- Computation time (C_i)
- Period (T_i)
- Relative deadline (D_i)

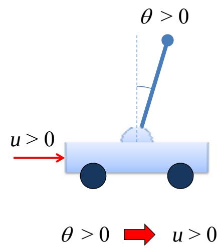
These parameters are specified by the programmer and are known off-line

- Arrival time (a_i)
- Start time (s_i)
- Finishing time (f_i)
- Response time (R_i)
- Slack and Lateness
- Jitter

These parameters depend on the scheduler and on the actual execution, and are known at run time.

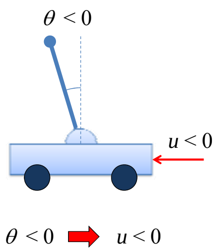
A control example

A positive angle θ requires a positive control action u



A control example

A negative angle θ requires a negative control action u



A control task

```
task   control(float theta0, float k)
{
float  error;
float  u;
float  theta;

  while (1) {
    theta = read_sensor();
    error  = theta – theta0;
    u = k * error;
    output(u);
    wait_for_next_period();
  }
}
```

control gain

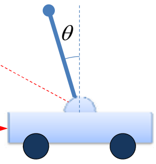
reference angle

sensing

computation

actuation

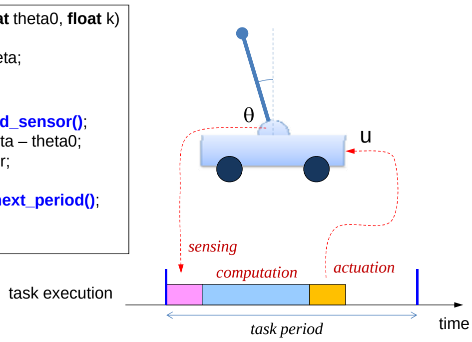
synchronization



A control task

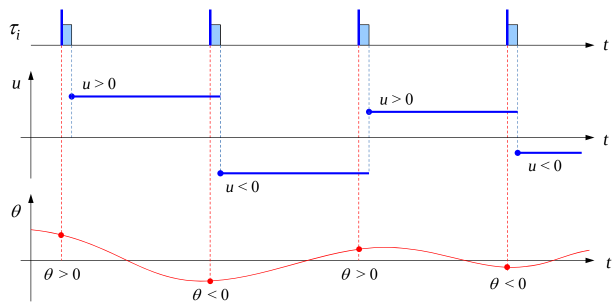
```
task   control(float theta0, float k)
{
float  error, u, theta;

  while (1) {
    theta = read_sensor();
    error  = theta – theta0;
    u = k * error;
    output(u);
    wait_for_next_period();
  }
}
```



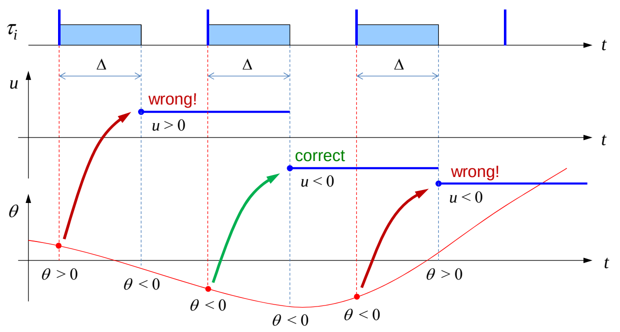
Traditional control view

Negligible delay and jitter:



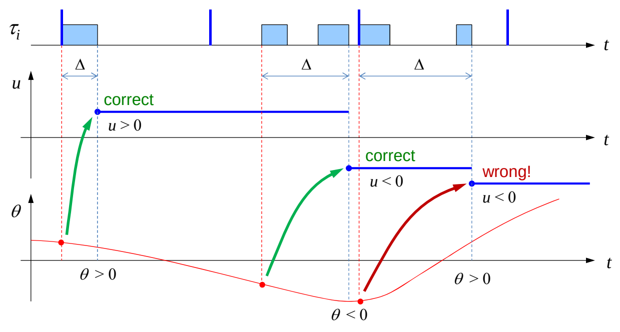
Effect of computation times

Computation times introduce a non negligible delay:



Actual situation

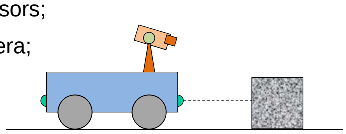
Actual situation: variable delay and jitter:



A robot control example

Consider a mobile robot equipped with:

- two actuated wheels;
- two proximity (US) sensors;
- a mobile (pan/tilt) camera;
- a wireless transceiver.

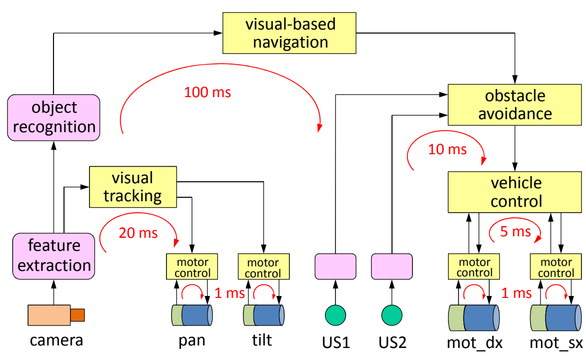


Goal
- Follow a path based on visual feedback;
- Avoid obstacles;
- Send complete robot status every 20 ms.

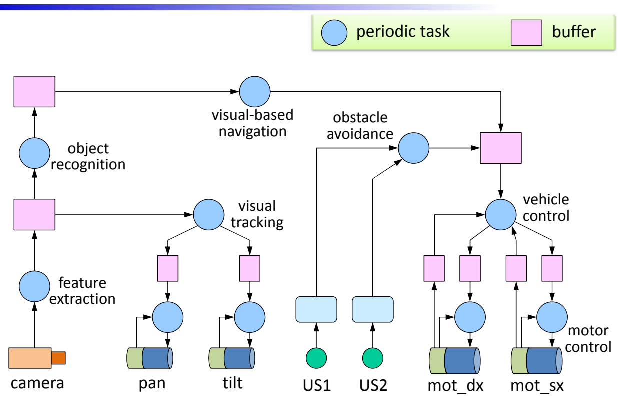
Design requirements

- **Modularity**: a subsystem must be developed without knowing the details of other subsystems (team work).
- **Configurability**: software must be adapted to different situations (through the use of suitable parameters) without changing the source code.
- **Portability**: minimize code changes when porting the system to different hardware platforms.
- **Predictability**: allow the estimation of maximum delays.
- **Efficiency**: optimize the use of available resources (computation time, memory, energy).

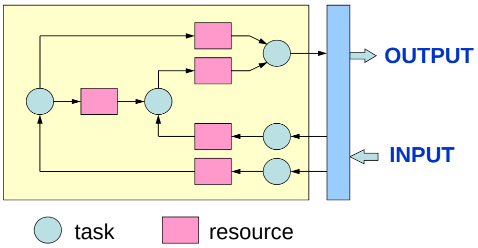
Control view



Software View



Software structure

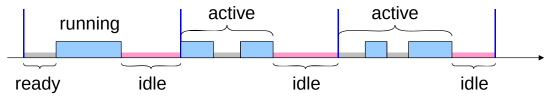


The operating system is responsible for providing the proper mechanisms for a predictable interaction between tasks and resources.

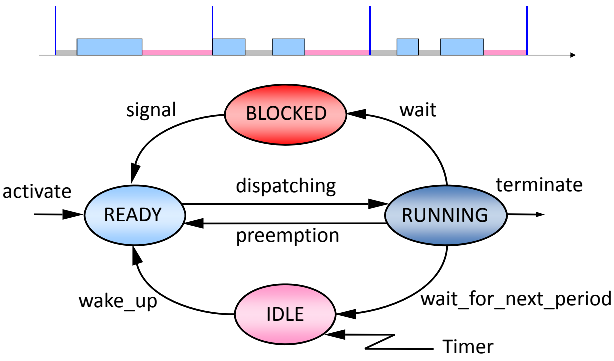
Support for periodic tasks

Task τ_i

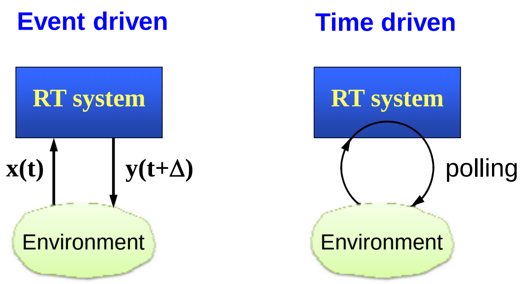
```
wait_for_activation();
while (condition) {
    ………
    wait_for_next_period();
}
```



The IDLE state



Design approaches



Task constraints

Types of constraints

- **Timing constraints**
  - activation, completion, jitter.
- **Precedence constraints**
  - they impose an ordering in the execution.
- **Resource constraints**
  - they enforce a synchronization in the access of mutually exclusive resources.

Timing constraints

They can be explicit or implicit.

- **Explicit timing constraints**

  They are directly included in the system specifications.

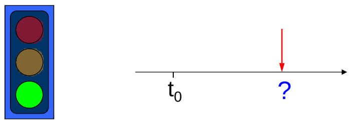
  Examples
  - open the valve **in** 10 seconds
  - send the position **within** 40 ms
  - read the altimeter **every** 200 ms
  - acquire the camera **every** 20 ms

Implicit timing constraints

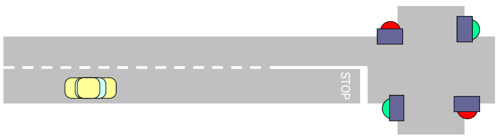
They do not appear in the system specification, but they need to be met to satisfy the performance requirements.

Example

What is the time validity of a sensory data?



Computing the yellow duration



$$D > T_d + T_r + T_b$$

T_d = detection time
T_r = reaction time
T_b = braking time

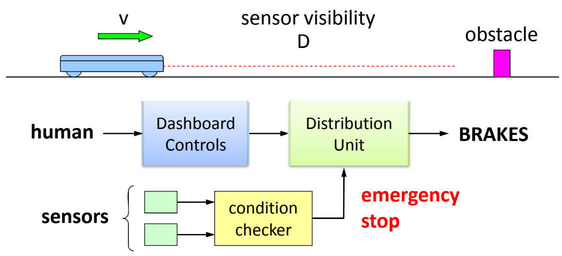
Computing the yellow duration

Detection time: $T_d = 0.6$ s
Reaction time: $T_r = 0.6$ s
Braking time: $T_b = v/(\mu g)$

$v = 50$ Km/h $= 14$ m/s, $\mu = 0.5$ $\Rightarrow$ $T_b = 2.8$ s

Time to stop the car from the time the yellow is turned on: **D > 4 s**

Example 2: automatic braking

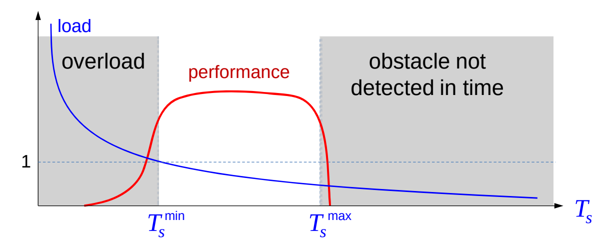


GOAL: If an obstacle is detected, stop the train without hitting the obstacle.

PROBLEM: Find the sampling periods of the sensors that guarantee the feasibility of the goal

Assumptions

- Let $\tau_s(C_s, T_s)$ be the task devoted to sampling ($D_s = T_s$)
- Assume τ_s is the task with the highest priority.
- Let U_{other} be the load of the other tasks



Minimum period

The minimum period can be computed by imposing that the system is not in overload:

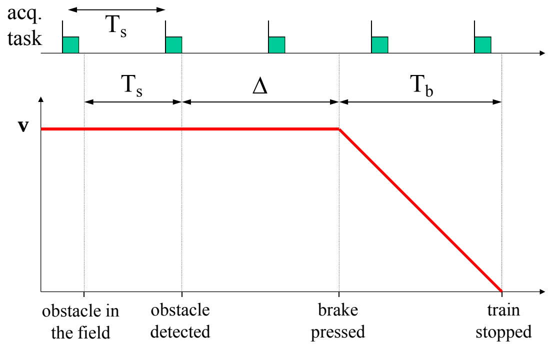
The system is in overload if $\frac{C_s}{T_s} + U_{other} > 1$

Hence a necessary condition for the system feasibility is $T_s \geq \frac{C_s}{1 - U_{other}}$

Thus: $$T_s^{\min} = \frac{C_s}{1 - U_{other}}$$

The maximum period can be found by a worst-case reasoning.

Worst-case reasoning



D = sensor visibility

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

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

$$a = \mu g$$

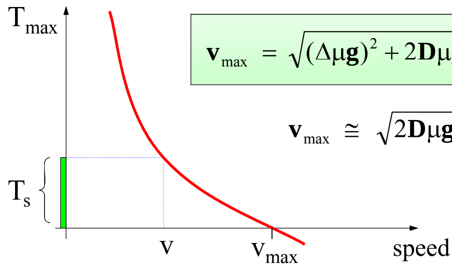
$$X_b = \frac{v^2}{2\mu g}$$

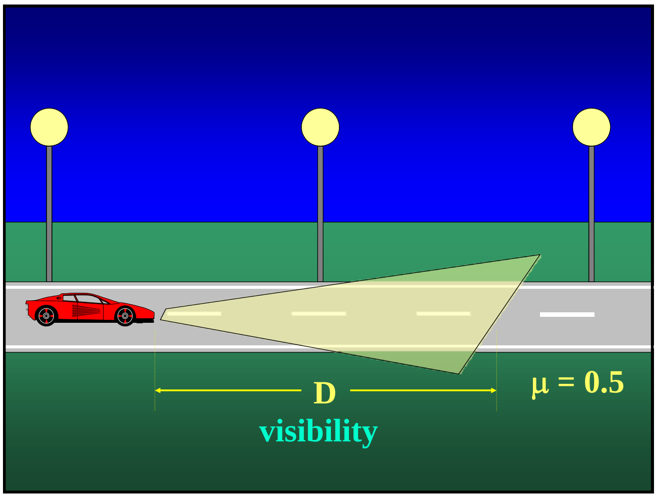
$$v(T_s + \Delta) + \frac{v^2}{2\mu g} < D$$

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

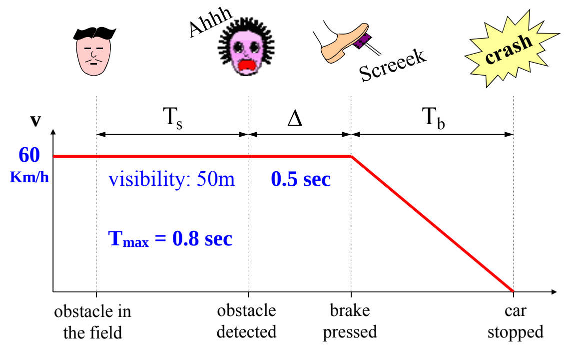
$$\mathbf{v}_{\max} = \sqrt{(\Delta\mu\mathbf{g})^2 + 2\mathbf{D}\mu\mathbf{g}} - \Delta\mu\mathbf{g}$$

$$\mathbf{v}_{\max} \cong \sqrt{2\mathbf{D}\mu\mathbf{g}}$$





Car driving



Lessons learned

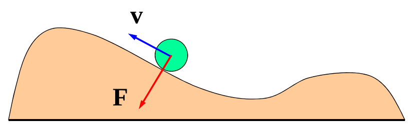
The farther we look, the faster we can run

To go fast safely, look ahead!!!

If $v \geq v_{max}$ no feasible solution exists, no matter how fast you react!!!

Don't look away from the road for too long!!!

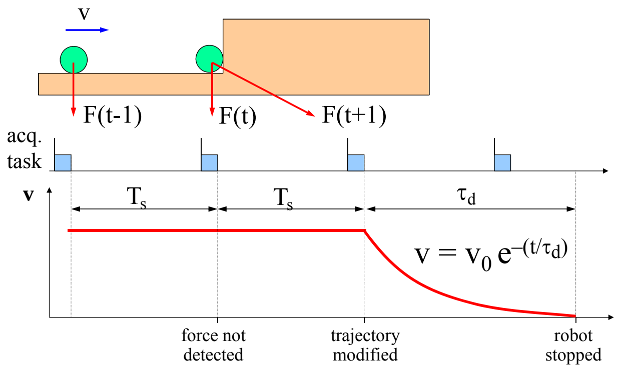
Example 3: contour following



Goal

Move at velocity **v** along the surface tangent, exerting a force **$F < F_{max}$** along its normal direction.

Worst-case reasoning



Lenght covered by the robot after the contact:

$$L = vT_s + x_f$$

$$x_f = \int_0^\infty v(t)dt = \int_0^\infty v_0 e^{-t/\tau_d} dt = -v_0\tau_d(e^{-\infty} - e^0) = v_0\tau_d$$

$$L = v(T_s + \tau_d)$$

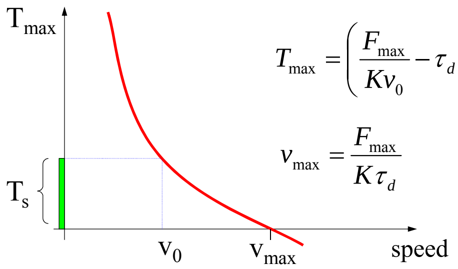
Force on the robot tool: (K = elastic coefficient)

$$F = KL = Kv(T_s + \tau_d) < F_{max}$$

70

Condition on the sampling period:

$$T_s < \frac{F_{max}}{Kv_0} - \tau_d$$



$$T_{max} = \left(\frac{F_{max}}{Kv_0} - \tau_d\right)$$

$$v_{max} = \frac{F_{max}}{K\tau_d}$$

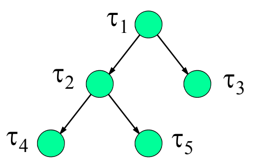
71

Types of constraints

- **Timing constraints**
  - activation, completion, jitter.
- **Precedence constraints**
  - they impose an ordering in the execution.
- **Resource constraints**
  - they enforce a synchronization in the access of mutually exclusive resources.

Precedence constraints

Sometimes tasks must be executed with specific precedence relations, specified through a **Directed Acyclic Graph** (**DAG**):



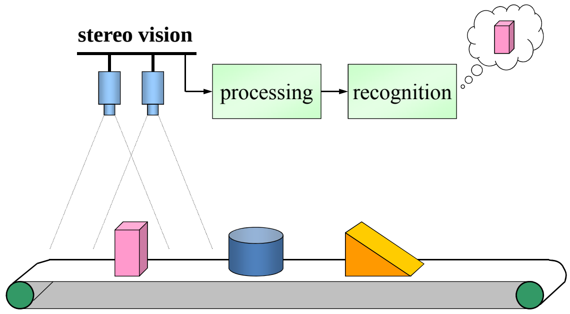
predecessor

$\tau_1 \prec \tau_4$

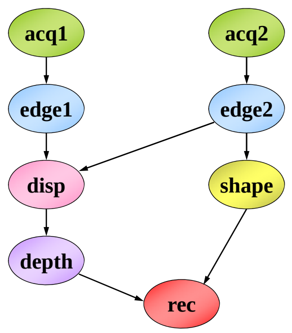
immediate predecessor

$\tau_1 \rightarrow \tau_2$

Sample application



Precedence graph



Other task models

To refine the analysis and reduce the pessimism, a task can be modeled at a finer grain expressing:

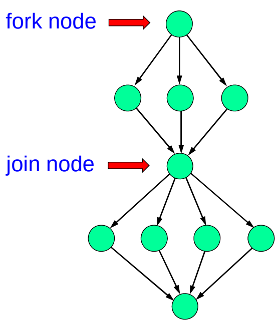
- precedence constraints between blocks
- execution flow of internal blocks
- potential parallel execution of code
- activation constraints of internal blocks
- timing constraints between internal blocks

More expressive models increase the complexity of the analysis.

Code parallelism

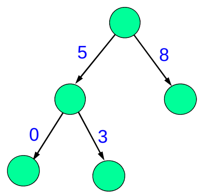
Fork-Join Graphs

- After a fork node, all immediate successors must be executed (the order does not matter).
- A join node is executed only after all immediate predecessors are completed.



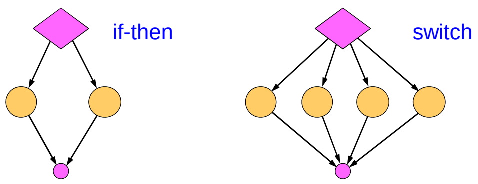
Code flow

- Some task model also allows specifying activation constraints between immediate successors as minimum interarrival times



Conditional nodes

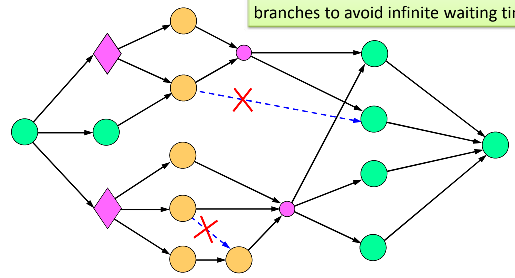
- A branch represents a conditional statement
- Only one node among all immediate successors must be executed



Conditional DAGs

They include both type of semantics, allowing representing both conditional statements and parallel execution:

Nodes in conditional branches cannot have precedence relations with nodes in other branches to avoid infinite waiting times.

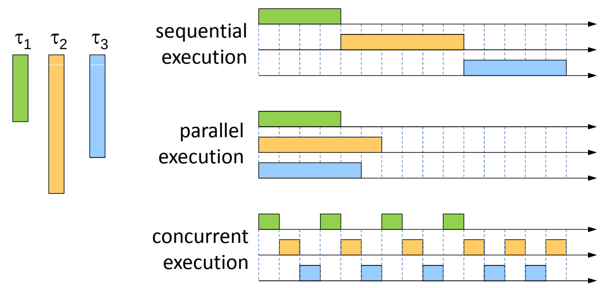


Types of constraints

- **Timing constraints**
  - activation, completion, jitter.
- **Precedence constraints**
  - they impose an ordering in the execution.
- **Resource constraints**
  - they enforce a synchronization in the access of mutually exclusive resources.

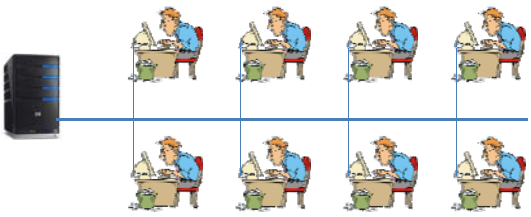
Concurrency

Resource conflicts are caused by concurrency, that is the ability of the processor to execute more tasks at a time, by alternating their executions:



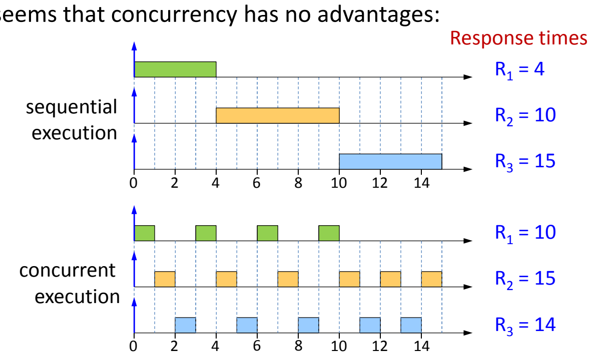
Multiprogramming

Concurrency is the basic mechanism used to implement multiprogramming in multi-user operating systems (it exploits input waiting times to manage other users):



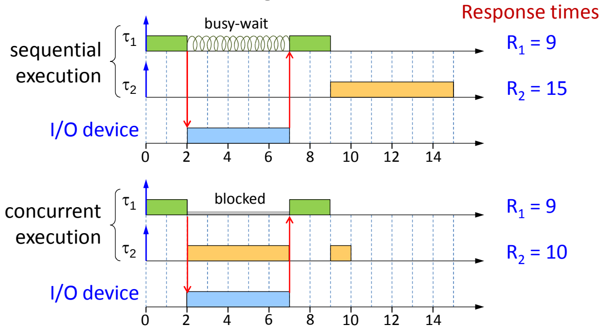
Concurrency

Comparing sequential with concurrent executions, it seems that concurrency has no advantages:



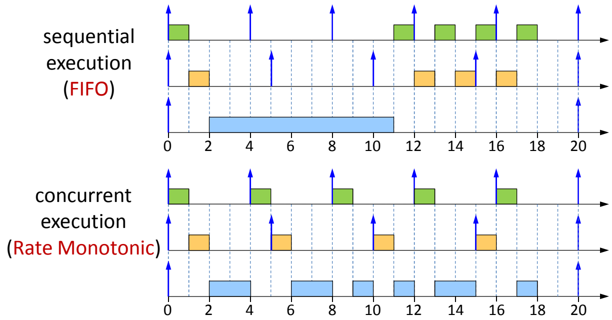
Concurrency and I/O

If a task must wait for I/O data, concurrency allows another task to run during that interval:



Periodic tasks

Concurrency becomes superior when managing periodic tasks at different rates (waiting times are used to execute other tasks):



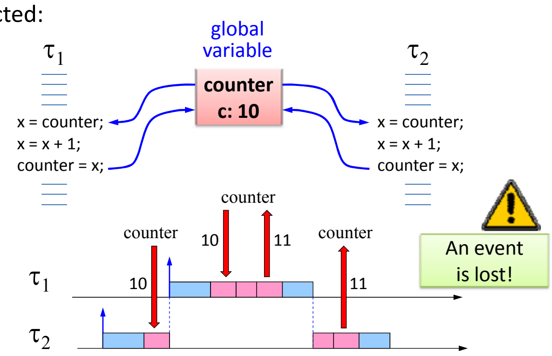
Concurrency: pro and cons

Hence, concurrency allows exploiting tasks inactive intervals (e.g., waiting times for input data or periodic task activation).

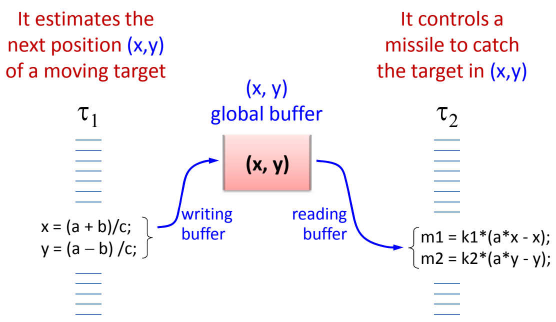
However, concurrency can generate conflicts when using shared resources (for example, when more tasks operate on global data).

Example of conflict

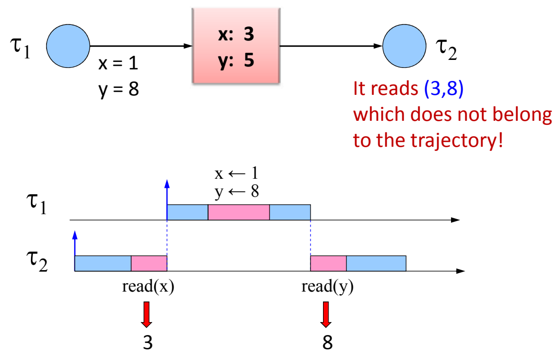
Each thread increments a counter every time an event is detected:



Example 2

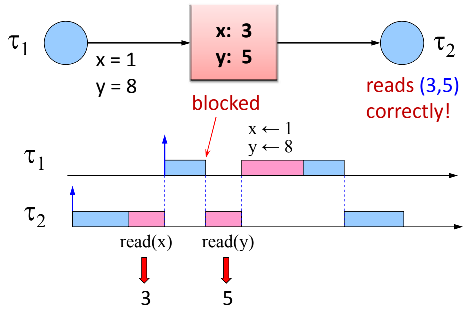


Example 2



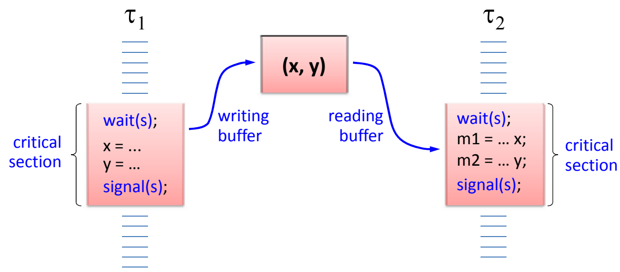
Solution

Regulate the use of shared resources so that tasks can only access them one at the time (i.e., in **mutual exclusion**):



Semaphores

Mutual exclusion is implemented by two primitives, wait(s) and signal(s), that use a system variable s, called semaphore:

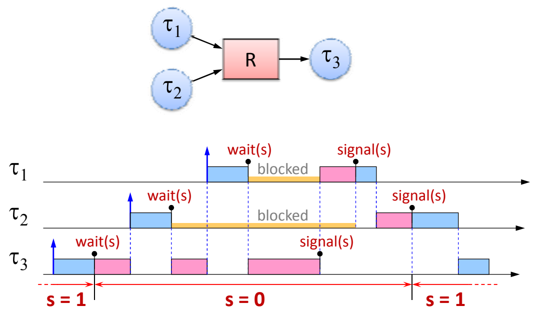


Semaphores

- Each shared resourse is protected by a different semaphore.
- s = 1 ⇒ free resource, s = 0 ⇒ busy (locked) resource.
- wait(s):
  - if s == 0, the task must be blocked on a queue of the semaphore. The queue management policy depends on the OS (usually it is FIFO or priority-based).
  - else set s = 0.
- signal(s):
  - if there are blocked tasks, the first in the queue is awaken (s remains 0), else set s = 1.

Semaphores

- If the semaphore s is initialized to 1, the pair wait(s) and signal(s) can be used for enforcing mutual exclusion:



Multi-unit resources

- If a resource has n parallel units that can be accessed by n tasks simultaneously, it can be protected by a semaphore initialized to n.
- wait(s):
  - if s == 0, the task is blocked on the semaphore queue;
  - else s is decremented.
- signal(s):
  - If there are blocked tasks, the first in the queue is awaken (s remains 0), else s is incremented.

Implementation notes

```
s = create_sem(n)
```

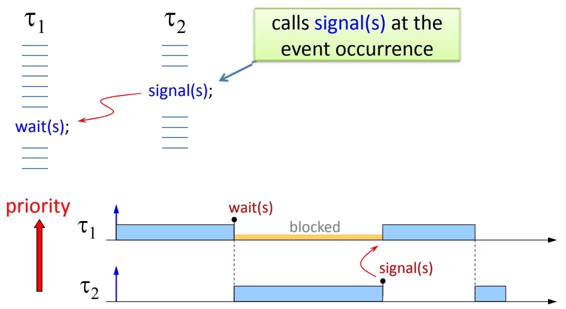
creates the semaphore structure, including a counter (s.count) initialized to n, and a queue of tasks (s.queue).

```
wait(s) {
    if (s.count == 0)
        <block the calling task on s.queue>
    else s.count--;
}

signal(s) {
    if (!empty(s.queue))
        <unblock the first task in s.queue>
    else s.count++;
}
```

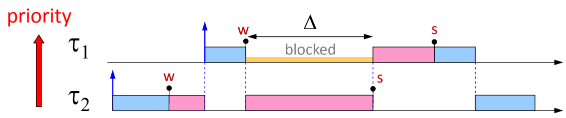
Synchronization semaphores

- A semaphore initialized to 0 can be used to wait for an event generated by another task:



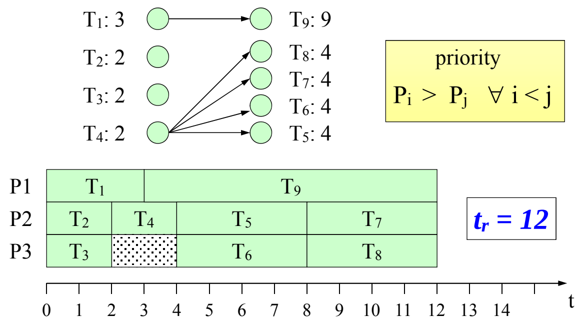
Problem with semaphores

- Semaphores (when properly used) guarantee the consistency of shared global data, but introduce extra blocking delays in high priority tasks.

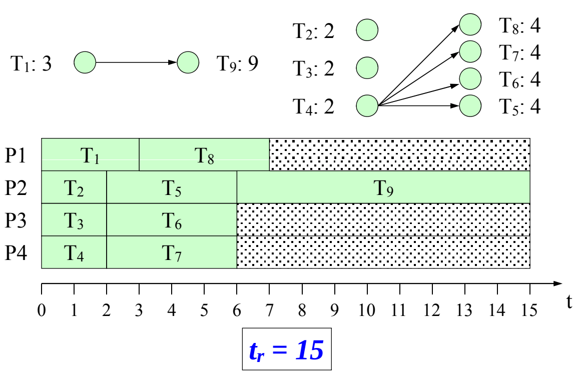


Timing anomalies

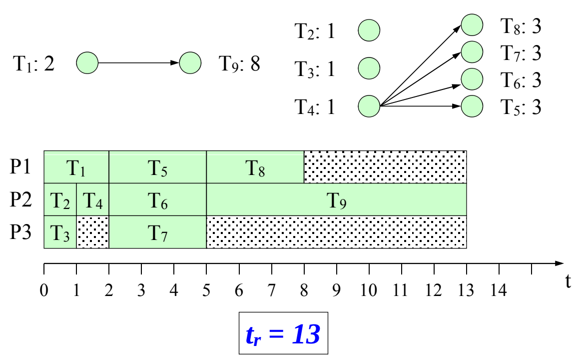
Scheduling anomalies



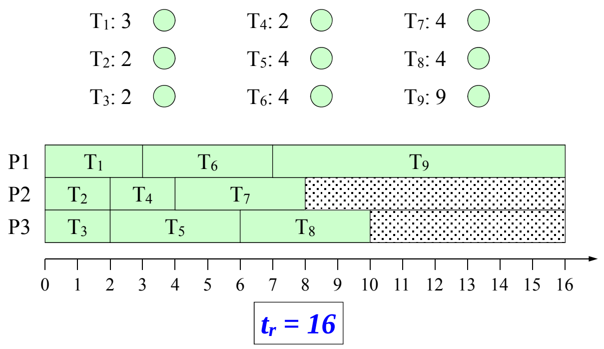
Increased processors



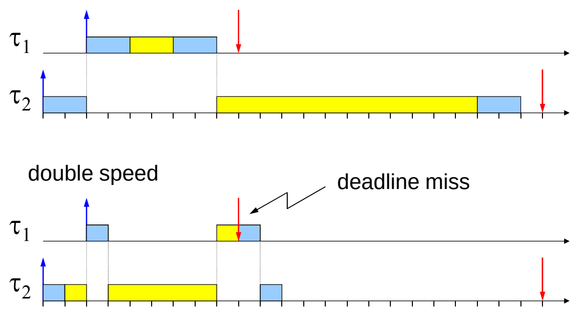
Shorter tasks



Released constraints

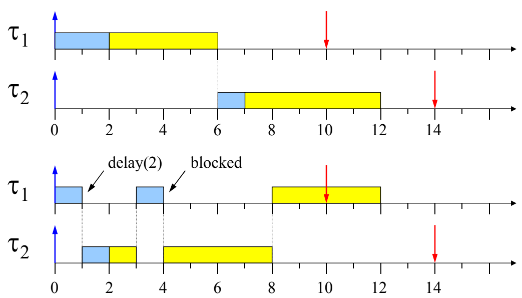


Faster processor



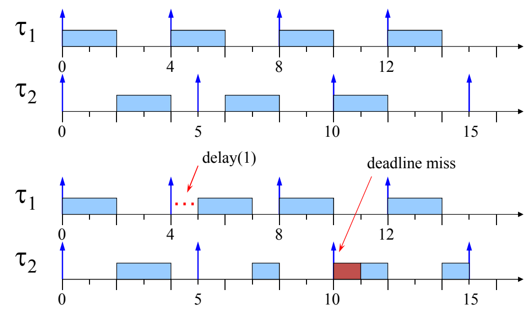
Delay: dangerous system call

A **delay(Δ)** may cause a delay longer than Δ.



Delay: dangerous system call

A **delay** in a task may also increase the response time of other tasks (example for fixed priorities):



Lessons learned

- Tests are not enough for real-time systems
- Intuitive solutions do not always work
- Delay should not be used in real-time tasks

The safest approach:
- **use predictable kernel mechanisms**
- **analyze the system to predict its behavior**

Achieving predictability

- The operating system is the most important component responsible for achieving a predictable execution.
- Concurrency control must be enforced by:
  - appropriate scheduling algorithms
  - appropriate synchronization protocols
  - efficient communication mechanisms
  - predictable interrupt handling