DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Sporadic tasks, task interaction and blocking

- Sporadic tasks
- Deadline Monotonic scheduling
- Priority inversion
- Priority inheritance
- Priority ceiling

Aperiodic and sporadic tasks

- Up to now we have considered only periodic tasks, where every task consists of an infinite sequence of identical activities that that are regularly released, or activated, at a constant rate.
- A ***aperiodic task*** consists of an infinite sequence of identical jobs. However, unlike periodic tasks, their release does not take place at a regular rate.
- Typical examples of aperiodic asks are
    - *User interaction*: Events generated by user interaction (key pressed, mouse clicked) and which require some sort of system response.
    - *Event reaction*. External events, such as alarms, may be generated at unpredictable times whenever some condition either in the system or in the controlled plant occurs.
- A aperiodic task for which it is possible to determine a minimum **inter-arrival time interval** is called a **sporadic task**.
- Sporadic tasks can model many situations that occur in practice.
    - For example, a minimum interarrival time can be safely assumed for events generated by user interaction, because of the reaction time of the human brain,
    - More in general, system events can be filtered in advance to ensure that, after the occurrence of a given event, no new instance will be issued until after a given **dead time**.

Sporadic tasks

- One simple way of expanding the basic process model to include sporadic tasks is to interpret the period Ti as the minimum interarrival time interval.

- For example, a sporadic task τi with Ti = 20 ms is guaranteed not to arrive more frequently than once in any 20 ms interval. Actually, it may arrive much less frequently, but a suitable schedulability analysis test will ensure (if passed) that the maximum rate can be sustained.

- For these tasks, assuming Di = Ti, that is, a relative deadline equal to the minimum interarrival time, is unreasonable because they usually encapsulate error handlers or respond to alarms.
    - The fault model of the system may state that the error routine will be invoked rarely but, when it is, it has a very short deadline.

- For many periodic tasks it is useful to define a deadline shorter than the period.

- The RTA method just described is adequate for use with the extended process model just introduced, that is, when Di ≤ Ti.
    - Observe that the method works with any fixed-priority ordering, and not just with the RM assignment, as long as the set hp(i) of tasks with priority larger than task τi is defined appropriately for all i and we use a preemptive scheduler.

Deadline Monotonic Priority Order (DMPO)

- RM was shown to be an optimal fixed-priority assignment scheme when $D_i = T_i$, this is no longer true for $D_i \leq T_i$.
- The following theorem introduces another fixed-priority assignment no more based on the period of the task but on their relative deadlines.
    - *The deadline monotonic priority order (DMPO), in which each task has a fixed priority inversely proportional to its deadline, is optimum for a preemptive scheduler under the basic process model extended to let $D_i \leq T_i$.*
- The optimality of DMPO means that, if any task set Γ can be scheduled using a preemptive, fixed-priority scheduling algorithm A, then the same task set can also be scheduled using the DPMO.
- As before, such a priority assignment sounds to be good choice since it makes sense to give precedence to more “urgent” tasks.
- The formal proof involves transforming the priorities of Γ (as assigned by A), until the priority ordering is Deadline Monotonic (DM), showing that each transformation step will preserve schedulability

Task Interaction

- The basic process model,assumed an underlying set of hypotheses to prove several interesting properties of real-time scheduling algorithms
- Unfortunately, some aspects of the basic process model are not fully realistic, and make those results hard to apply to real-world problems.
- The hypothesis that tasks are completely independent from each other regarding execution is particularly troublesome because it sharply goes against the basics of all the interprocess communication methods
- In one form or another, they all require tasks to interact and coordinate, or synchronize, their execution. In other words, tasks will sometimes be forced to block and wait until some other task performs an action in the future.
- For example, tasks may either have to wait at a critical region's boundary to keep a shared data structure consistent, or wait for a message from another task before continuing.
- In all cases, their execution will clearly no longer be independent from what the other tasks are doing at the moment
- The following discussion will mainly address task interactions due to **mutual exclusion**, a ubiquitous necessity when dealing with shared data

Priority Inversion

- Informally speaking, when a high-priority task is waiting for a lower-priority task to complete some required computation, the task priority scheme is, in some sense, being hampered because the high-priority task would take precedence over the lower-priority one in the model.

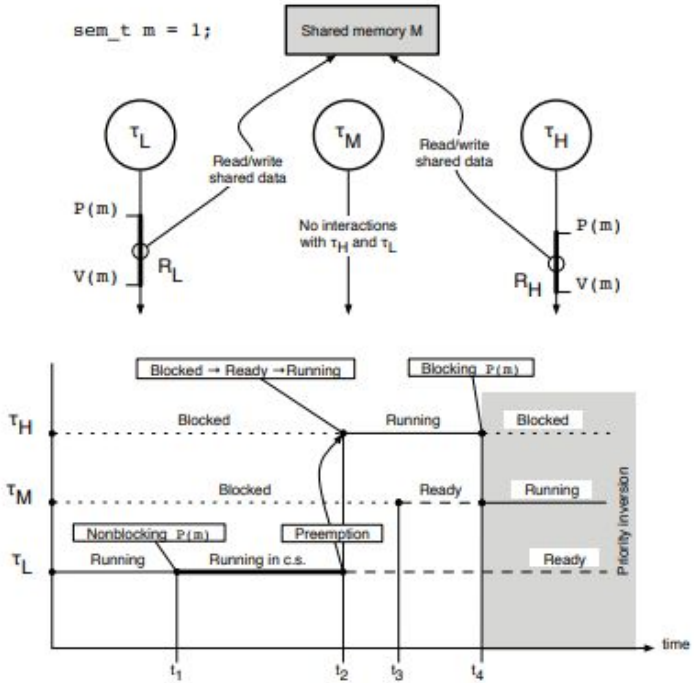
- This happens even in very simple cases, for example, when several tasks access a shared resource by means of a critical region protected by a mutual exclusion semaphore. Once a lower-priority task enters its critical region, the semaphore mechanism will block any higher-priority task wanting to enter its own critical region protected by the same semaphore and force it to wait until the former exits.

- This phenomenon is called **priority inversion** and, if not adequately addressed, can have adverse effects on the schedulability of the system, to the point of making the response time of some tasks completely unpredictable because the priority inversion region may last for an unbounded amount of time.

- Knowing in advance the amount of time any task will spend in the critical section, at a first glance, mutual exclusion can be taken into account adding to the computation time the maximum time spent by any task in the critical section.

- Indeed, in the worst case, if a task is accessing **N** critical sections, it has to wait for each critical section the maximum time $\mathbf{T_N}$, i.e. the maximum time spent by any task in that critical section

Unbounded Priority Inversion



la task con priorita piu alta deve aspettare per tutte quelle con priorita piu alta della T_L e deve aspettare T_L

- In the above example, a third task that does not access the critical section influences the time the high priority task has to wait.
- More in general, the time a task has to wait to access a critical section is in general **unbounded**

Priority Inheritance

- On a single-processor system, a very simple and drastic solution to the unbounded priority inversion problem is to forbid preemption completely during the execution of all critical regions

- This may be obtained by disabling the operating system scheduler or, even more drastically, turning interrupts off within critical regions

- However, this technique introduces a new kind of blocking, of a different nature. That is, any higher-priority task τ_M that becomes ready while a low priority task τ_L is within a critical region will not get executed and we therefore consider it to be blocked by τ_L until τ_L exits from the critical region, even if τ_M does not interact with τ_L at all.

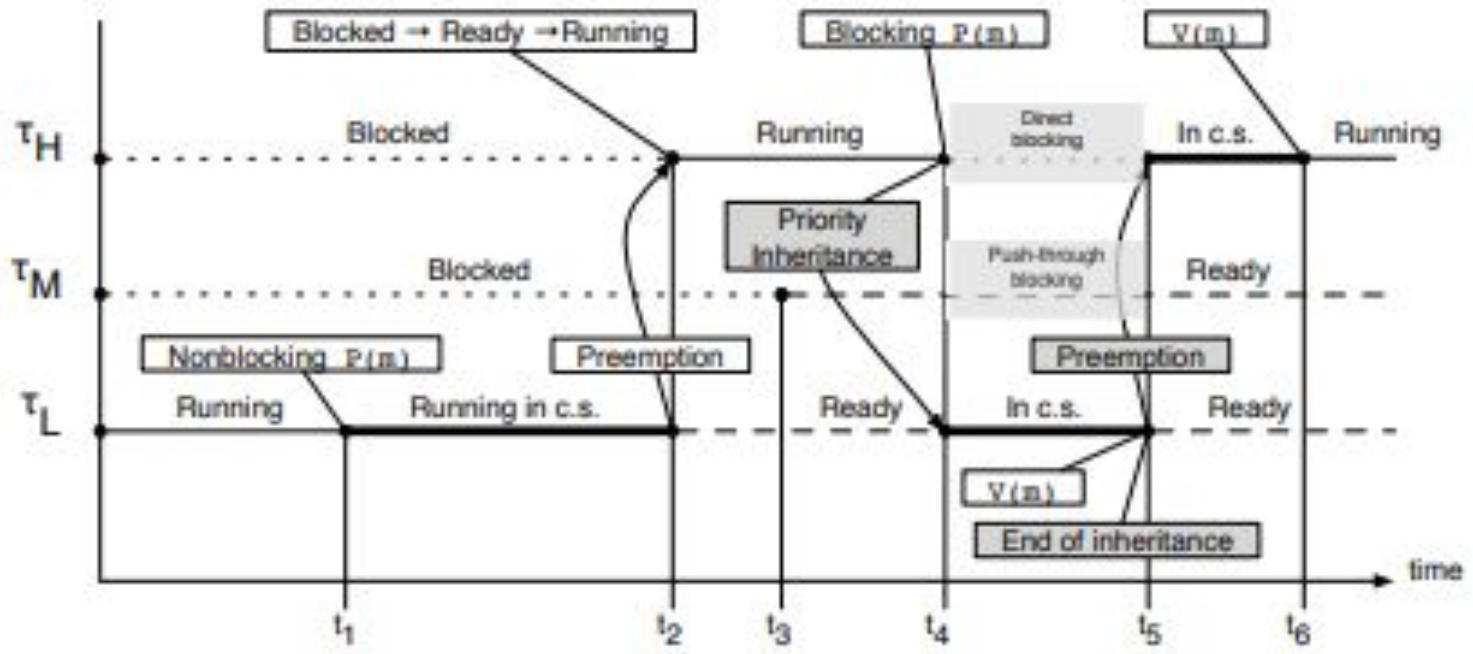
- A better solution is to dynamically increase the priority of a task as soon as it is blocking some higher-priority tasks. In particular, if a task τ_L is blocking a set of n higher-priority tasks τ_{H1}, ..., τ_{Hn} at a given instant, it will temporarily inherit the highest priority among them. This prevents any middle-priority task from preempting τ_L and unduly make the blocking experienced by τ_{H1}, ..., τ_{Hn} any longer than necessary.

Priority Inheritance Rules

- The priority inheritance protocol itself consists of the following set of rules:
  - 1. When a task τ_H attempts to enter a critical region that is “busy”— that is, its controlling semaphore has already be taken by another task τ_L—it blocks, but it also transmits its active priority to the task τ_L that is blocking it if the active priority of τ_L is lower than τ_H ’s.
  - 2. As a consequence, τ_L will execute the rest of its critical region with a priority at least equal to the priority it just inherited. In general, a task inherits the highest active priority among all tasks it is blocking.
  - 3. When a task τ_L exits from a critical region and it is no longer blocking any other task, its active priority returns back to the baseline priority.
  - 4. Otherwise, if τ_L is still blocking some tasks—this happens when critical regions are nested into each other—it inherits the highest active priority among them

un processo con priorità piu bassa non potrà mai provare ad entrare nella critical section mentre una task con priorità più alta e’ nella critical section. questo perché se ha priorità più bassa non può essere running. quindi possiamo concentrarci solo nel caso dove una task con priorita più bassa e’ nella critical section

An Example


Blocked → Ready →Running
Blocking P(m)
V(m)
τ_H
Blocked
Running
Direct blocking
In c.s.
Running
Priority Inheritance
τ_M
Blocked
Push-through blocking
Ready
Nonblocking P(m)
Preemption
Preemption
τ_L
Running
Running in c.s.
Ready
In c.s.
Ready
V(m)
End of inheritance
time
t_1
t_2
t_3
t_4
t_5
t_6

An upper limit to blocking time

- It can be proved that under priority inheritance policy, the time a task spends waiting for a resource (here represented as a semaphore) is bounded.

- The following theorem provides an upper bound for the time spent waiting for resources:

- *Let K be the total number of semaphores in the system. If critical regions cannot be nested, the worst-case blocking time experienced by each activation of task τi under the priority inheritance protocol is bounded by Bi:*

  $$B_i = \sum_{k=1,\,K} usage(k, i) C(k)$$

  where ***usage(k, i)*** is a function that returns 1 if semaphore Sk is used by (at least) one task with a priority less than the priority of τi and (at least) one task with a priority higher than or equal to the priority of τi. Otherwise, usage(k, i) returns 0.
  C(k) is the worst-case execution time among all critical regions corresponding to, or guarded by, semaphore Sk.

The Priority Ceiling Protocol

- The protocol uses an additional information for each semaphore: the **ceiling**, defined as the maximum initial priority of all tasks that use it
- As in the priority inheritance protocol, each task has a current (or active) priority that is greater than or equal to its initial (or baseline) priority, depending on whether it is blocking some higher-priority tasks or not.
- The priority inheritance rule is exactly the same in both cases.
- A task can immediately acquire a semaphore only if its active priority *is higher than the ceiling* of any currently locked semaphore, excluding any semaphore that the task has already acquired in the past and not released yet. Otherwise, it will be blocked.
- In this case it is therefore possible that a task is blocked even if the requested resource is accessible

Properties of priority ceiling

1. **A high-priority task can be blocked at most once during its execution by lower-priority tasks.**

2. **The protocol prevents deadlock.**

3. **Mutual exclusive access to resources is ensured by the protocols themselves**

In particular, the first property leads to the following theorem:

Let K be the total number of semaphores in the system. The worst-case blocking time experienced by each activation of task τi under the priority ceiling protocol is bounded by

$$Bi: Bi = \max_{k=1,K} \{usage(k, i)C(k)\}$$

where ***usage(k, i)*** is a function that returns 1 if semaphore Sk is used by (at least) one task with a priority less than τi and (at least) one task with a priority higher than or equal to τi. Otherwise, it returns 0. C(k) is the worst-case execution time among all critical regions guarded by semaphore Sk

Response time analysis considering blocking

- The worst-case response time Ri,can be redefined to take Bi into account as follows:

  Ri = Ci + Bi + Ii

  In this way, the worst-case response time Ri of task τi is expressed as the sum of three components:
  1. the worst-case execution time Ci,
  2. the worst-case interference Ii, and
  3. the worst-case blocking time Bi.

  The corresponding recurrence relationship introduced for Response Time Analysis (RTA) becomes

$$w_i^{(k+1)} = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{w_i^{(k)}}{T_j} \right\rceil C_j$$

the new recurrence relationship still has the same properties as the original one. In particular, if it converges, it still provides the worst-case response time Ri for an appropriate choice of w0 i . As before,Ci are good starting points