

Response Time Calculation

Deadline Monotonic calculation.

Slide 1

	T	D	C
A	50	10	5
B	500	500	250
C	3000	3000	1000

Let's calculate the response times of the three processes under dead-line monotonic priority assignment.

First $R_A = 5$ because A is the highest priority task.

Task B

Task B has task A as its only higher priority task. So

$$R_B = 250 + \left\lceil \frac{R_B}{50} \right\rceil 5$$

Slide 2

Solving the recurrence relation we get that: $R_B^0 = 0, R_B^1 = 250$

$$R_B^2 = 250 + \left\lceil \frac{250}{50} \right\rceil 5 = 275$$

$$R_B^3 = 250 + \left\lceil \frac{275}{50} \right\rceil 5 = 280$$

$$R_B^4 = 250 + \left\lceil \frac{280}{50} \right\rceil 5 = 280$$

Task C

Task C is the lowest priority so

$$R_C = 1000 + \left\lceil \frac{R_C}{50} \right\rceil 5 + \left\lceil \frac{R_C}{500} \right\rceil 250$$

Slide 3 So $R_C^0 = 0, R_C^1 = 1000$ and

$$R_C^2 = 1000 + \left\lceil \frac{1000}{50} \right\rceil 5 + \left\lceil \frac{1000}{500} \right\rceil 250 = 1600$$

$$R_C^3 = 1000 + \left\lceil \frac{1600}{50} \right\rceil 5 + \left\lceil \frac{1600}{500} \right\rceil 250 = 2160$$

$$R_C^4 = 1000 + \left\lceil \frac{2160}{50} \right\rceil 5 + \left\lceil \frac{2160}{500} \right\rceil 250 = 2470$$

Example continued

$$R_C^5 = 1000 + \left\lceil \frac{2470}{50} \right\rceil 5 + \left\lceil \frac{2470}{500} \right\rceil 250 = 2500$$

Slide 4

$$R_C^6 = 1000 + \left\lceil \frac{2500}{50} \right\rceil 5 + \left\lceil \frac{2500}{500} \right\rceil 250 = 2500$$

Since $R_C^6 = R_C^5$ we have found the response time. As each task's response time is less than its deadline we know that a priority based deadline-monotonic assignment will schedule the tasks.

Sharing Resources



Slide 5

So far we have assumed that all the tasks have been independent. What happens if we have two tasks that require access to the same piece of data?

Since we allow our tasks to be interrupted at arbitrary points, we have to have some control over shared data.

Why?

Why didn't we have a problem with the Cyclic Executive approach we looked at in lecture 2?

Semaphores



Slide 6

A semaphore is a very simple mechanism to allow concurrent access to a data object which only allows one process to access a shared object at a time. It is simply an extra flag to a data value:

- if there is no process accessing the data the flag is set to **unlocked**
- if a process wishes to access the data *and* the flag is set to **unlocked** the process must first set the flag to **locked** and then proceed to access the data then when finished it *must* **unlock** the semaphore.
- If a process attempts to access the data object and the flag is **locked** by another process, it must *wait* until the flag gets unlocked.

Critical Sections

Slide 7 Once some process has locked the semaphore the code (or time-period) while that process has the lock is referred to as the critical section.

When we consider worst-case response times of processes we must take into account the extra time that a processes can be blocked the another process accessing shared data. ([More later](#)).

Priority Inversion an Example

Let us go back to our earlier task set.

Slide 8

	T	D	C	R
A	50	10	5	5
B	500	500	250	280
C	3000	3000	1000	2500

Suppose tasks A and C share some common data, each required access to the data for at most 1ms of computation time. We shall refer to the semaphore guarding the data as **s**.

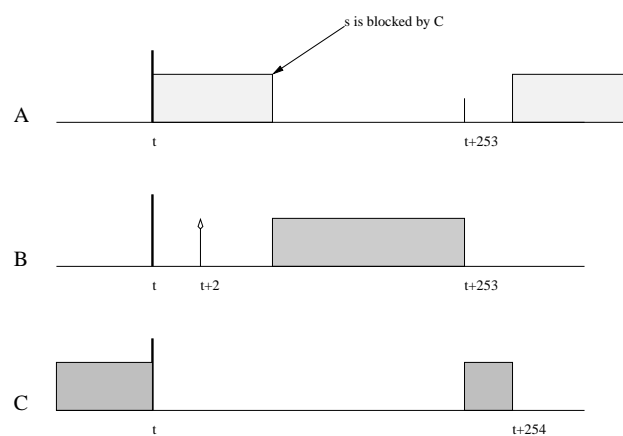
Priority Inversion an Example

Slide 9


Let us suppose that task C is running and locks s at some time t , just after C locks s , task A is activated (because it has a higher priority), then task B made runnable then at time $t+3$ task A needs the data that C has control of because it is locked task B starts running (because task B has higher priority than C). Task B then runs for 250ms of time and finishes at time 253, C can now run it finishes its computation so A starts running but A has missed its deadline.

Priority Inversion an Example

Slide 10



Priority inversion



Slide 11

The execution of task B delays the execution of task A even though it has a lower priority than task A. This effect is known as *priority inversion* where a lower priority task can delay the execution of a higher priority task.

In this example we only had one task B between A and C, if there were many tasks in between the blocking time could be much worse.

With blocking fixed priority scheduling degenerates into a FIFO (First In, First Out) queue in the worst case.

Priority Inheritance



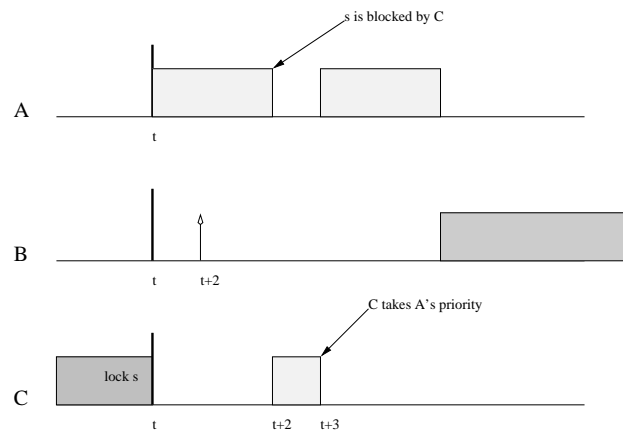
Slide 12

While task A is blocked by the semaphore it seems intuitive that task C is more important than task B, since task C is blocking the completion of task A which is a higher priority task.

So if we gave C a higher priority than B while C was in its critical section could we avoid the problem?

Priority Inheritance an example

Slide 13



Deadlocks?

Suppose we have two semaphores s_1 and s_2 .

Slide 14

- At time t task C (the lower priority task) locks s_1
- just after this tasks A runs and locks s_2 and then tries to lock s_1 which it can't lock because C is blocking s_1 .
- Task C then inherits the priority of A and starts running, task C tries to lock s_2 but task A already holds it, to neither task can execute hence we have a deadlock.

The Priority Ceiling Protocol

Slide 15

So we have to be careful with priority inheritance. We need some mechanism for priority which does not deadlock.

We will look at the Priority ceiling protocol developed by Sha, Rajkummar and Lehoczky which is guaranteed not to deadlock once we make some restrictions. We will then be able to extend our response time analysis calculations to take into account blocking times of tasks.

Priority Ceiling Protocol Task Restrictions

Slide 16

We need the following restrictions on tasks.

- A task must release any semaphores it has before it is completed.
A task can not hold semaphores between invocations.
- Tasks must lock and unlock semaphores in a stack like manner.

Thus

```
lock(s1) ... lock(s2) .... unlock(s2) ... unlock(s1)
```

is o.k. But

```
lock(s1) ... lock(s2) .... unlock(s1) ... unlock(s2)
```

is not.

Priority Ceiling Protocol Continued

We need to assume that the time each process needs to be in a critical section is bounded and known beforehand.

Slide 17

We denote

$$CS_{i,s}$$

as the length of the critical section for task i holding semaphore s .

We also need to assume that we know for a given task i a priori the set of semaphores it needs to be able to lock. Denote $uses(i)$ as the set of semaphores that occur in task i .

Ceilings

Slide 18

The protocol uses the idea of a semaphore *ceiling*: the ceiling of a semaphore is the priority of the highest priority task that uses that semaphore.

Given a semaphore s denote the priority of the highest priority task which uses s as

$$ceil(s)$$

Run time of actions of the protocol

Slide 19

At run time we do the following: if a task i wants to lock a semaphore s , it can only do so if the priority of i is *strictly* higher than the ceilings of all semaphores currently locked by other tasks. If this condition is not met then task i is blocked.

As before, when a task blocks on a semaphore s the task currently holding s inherits the priority of that task.

An example

Slide 20

	T	D	C	priority
A	50	10	5	high
B	500	500	250	medium
C	3000	3000	1000	low

Suppose we have three semaphores s_1, s_2 and s_3 with the following locking pattern:

- Task A: $lock(s_1) \dots unlock(s_1)$
- Task B: $lock(s_2) \dots lock(s_3) \dots unlock(s_3) \dots unlock(s_2)$
- Task C: $lock(s_3) \dots lock(s_2) \dots unlock(s_2) \dots unlock(s_3)$

So we have that:

- $ceil(s_1) = H$
- $ceil(s_2) = M$
- $ceil(s_3) = M$

Slide 21

$cs_{i,s}$	s_1	s_2	s_3
A	5		
B		10	5
C		10	25

Calculating the response time

Before we calculated the response time of a task as:

Slide 22

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

We want to include some term B_i which tells us how long the task is blocked by semaphores.

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Blocking Time

How do we calculate the blocking time? A task can be blocked in one of two ways:

- Slide 23**
- normal blocking where a task tries to access a semaphore which is locked.
 - ceiling blocking where the ceilings of the currently blocked semaphores is too high.

A task can also be blocked when executing non critical-section code when a lower priority task inherits a higher priority. This is called *push through*.

Blocking Time continued

- Slide 24**
- So in summary a given task i is blocked by at most one critical section of any lower priority task locking a semaphore with priority ceiling greater than or equal to the priority of task i .

$$B_i = \max\{cs_{k,s} \mid k \in lp(i) \wedge s \in uses(k) \wedge ceil(s) \geq pri(i)\}$$

where $lp(i)$ is the set of all lower priority tasks than i .

Example continued B_A

Slide 25 Back to our example, first lets try and calculate the blocking time B_A . First $lp(B_A)$ is the set $\{B, C\}$. But A is the highest priority task and since it doesn't share any locks with B and C the ceilings of the tasks in $uses(B)$ and $uses(C)$ are never equal to the priority of A . Thus the set:

$$\{cs_{k,s} | k \in \{B, C\} \wedge s \in uses(k) \wedge ceil(s) \geq pri(A)\}$$

is empty (the maximum of an empty set is normally defined to be zero by convention). So B_A is 0.

Example continued B_B and B_C

Slide 26 Task B has only one lower priority task, task C , which can only accesses semaphores s_2 and s_3 . Again we look at the set

$$\{cs_{C,s} | s \in uses(B) \wedge ceil(s) \geq M\}$$

both s_2 and s_3 have a priority ceiling equal to M so B_B is 25.

Task C has no lower priority tasks so B_C will be zero.

Calculating the response times, R_A and R_C

Slide 27 First

$$R_A = 5 + B_A = 5$$

next

$$R_C = 1000 + B_C + \left\lceil \frac{R_C}{50} \right\rceil 5 + \left\lceil \frac{R_C}{500} \right\rceil 250$$

so $R_C = 2500$.

Calculating the response times, R_B

Slide 28

$$R_B = 250 + R_B + \left\lceil \frac{R_B}{50} \right\rceil 5$$

which gives:

$$R_B = 275 + \left\lceil \frac{R_B}{50} \right\rceil 5$$

Calculating the response times, R_B

Using iteration $R_B^1 = 275$ and

Slide 29
$$R_B^2 = 275 + \left\lceil \frac{275}{50} \right\rceil 5 = 305$$

$$R_B^3 = 275 + \left\lceil \frac{305}{50} \right\rceil 5 = 310$$

$$R_B^4 = 275 + \left\lceil \frac{310}{50} \right\rceil 5 = 310$$

Thus R_B is less than B 's deadline ($310 \leq 500$) hence the task set can be scheduled using the priority ceiling protocol.

Problems

Slide 30 The priority ceiling protocol is quite complicated, and can be difficult to implement in practise.

The operating system has to keep track of inherited priorities it also has to work out if a task is blocking because of the priority ceiling.

But we have achieved a lot by showing how it is possible to extend our response time calculation to include blocking times.

The Immediate Inheritance Protocol



We assume the same semaphore locking model we use the following run time behaviour:

Slide 31 When a task i want to lock a semaphore s , the task *immediately* sets the priority to the maximum of the current priority and the ceiling priority of s . When the task finishes with s it sets it priority back to what is was before.

That's it. This protocol has the same worst-case response time as with the priority ceiling protocol (this means our previous calculations) are still valid.