Announcement

We have so far studied the schedulability analysis of independent tasks. We now examine some practical applications that includes more details.

- nonzero task switching times
- transient overloads using period transformation
- improving processor utilization using period transformation

Review: Schedulability: UB Test

- Utilization bound(UB) test[Liu73]: a set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasing, if
- $$\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} \leq U(n) = n(2^{1/n} - 1)$$
- U(1) = 1.0 U(4) = 0.756 U(7) = 0.728
- U(2) = 0.828 U(5) = 0.743 U(8) = 0.724
- U(3) = 0.779 U(6) = 0.734 U(9) = 0.720

- For harmonic task sets, the utilization bound is U(n)=1.00 for all n. For large n, the bound converges to *ln 2* ~ 0.69.

- Conventions, task 1 has shorter period than task 2 and so on.

Exact Schedulability Test

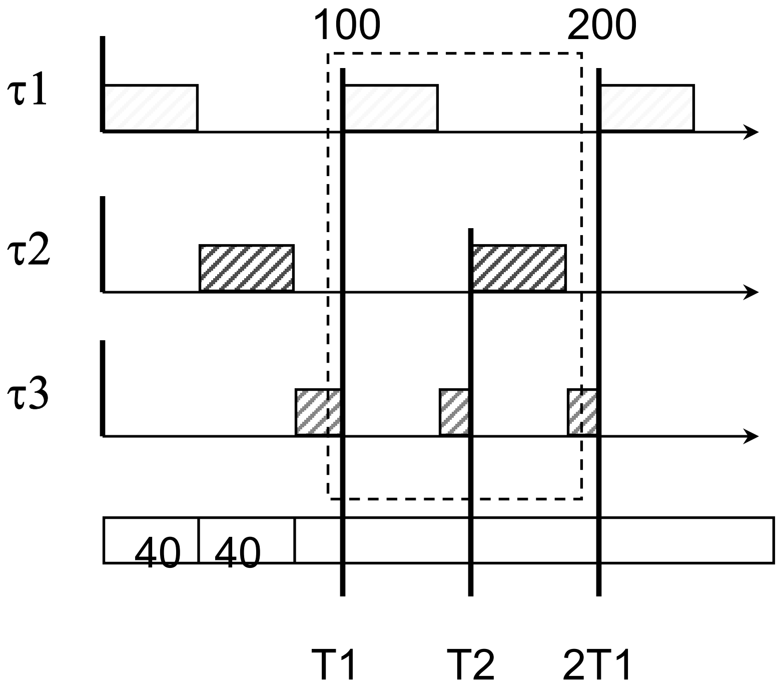
•

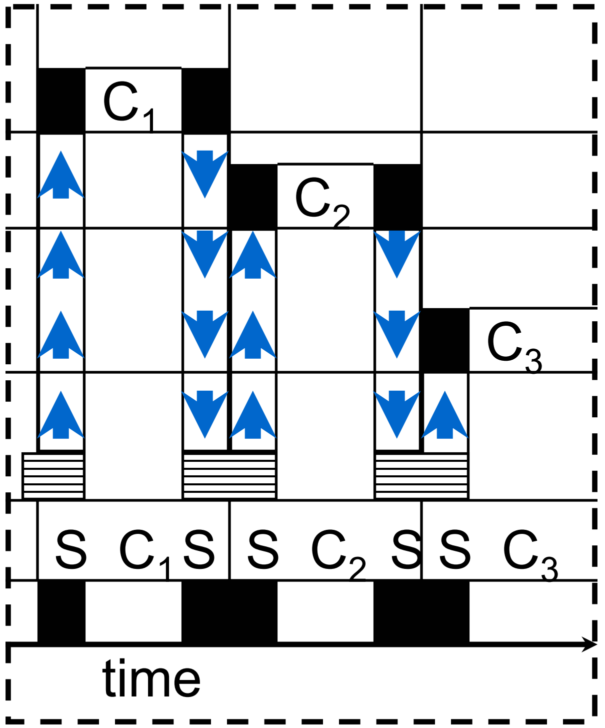
$$a_{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{a_n}{T_j} \right\rceil C_j \qquad \text{where } a_0 = \sum_{j=1}^{i} C_j$$

Test terminates when $a_{n+1} > T_i$ (not schedulable)
or when $a_{n+1} = a_n \leq T_i$ (schedulable).

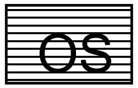
The subscript to *a* indicates the number of iterations in the calculation.
The index *i* indicates it is the *ith* task being checked.
The index *j* runs from 1 to i-1, i.e all the higher priority tasks. Recall from the convention - task 1 has a higher priority than task 2 and so on.

Modeling Task Switching as Execution Time





Two scheduling actions per task
(start and end of period)



Analyzing Task Switching

The effect of context switching can be directly added to the computation time, that is, replace C_i = (C_i +2S) in BOTH the UB test and exact test. For example:

$$\tau_1 \qquad \frac{(C_1 + 2S)}{T_1} \le U(1)$$

$$\tau_2 \qquad \frac{(C_1 + 2S)}{T_1} + \frac{(C_2 + 2S)}{T_2} \le U(2)$$

$$\tau_3 \qquad \frac{(C_1 + 2S)}{T_1} + \frac{(C_2 + 2S)}{T_2} + \frac{(C_3 + 2S)}{T_3} \le U(3)$$

Transient Overloads

Sometimes, there are occasional errors requires exception handling in the software. Choices are to

- 1) do the worst case analysis that includes exception handling
- 2) ignore the exception condition and handle the overload as it occurs.

Option 2 is applicable only if the task set can be divided into two sets: a critical task set and a non-critical task set whose deadline can be missed occasionally. In this case, a scheduling algorithm is considered stable if a set of tasks exists such that all tasks in the set will meet their deadlines even if the processor is overloaded.

Tasks outside this “set” will occasionally miss their deadlines. Thus there are n highest priority tasks which will always meet their deadlines.

Transient Overloads (contd.)

GRMS requires task assignment according to period.

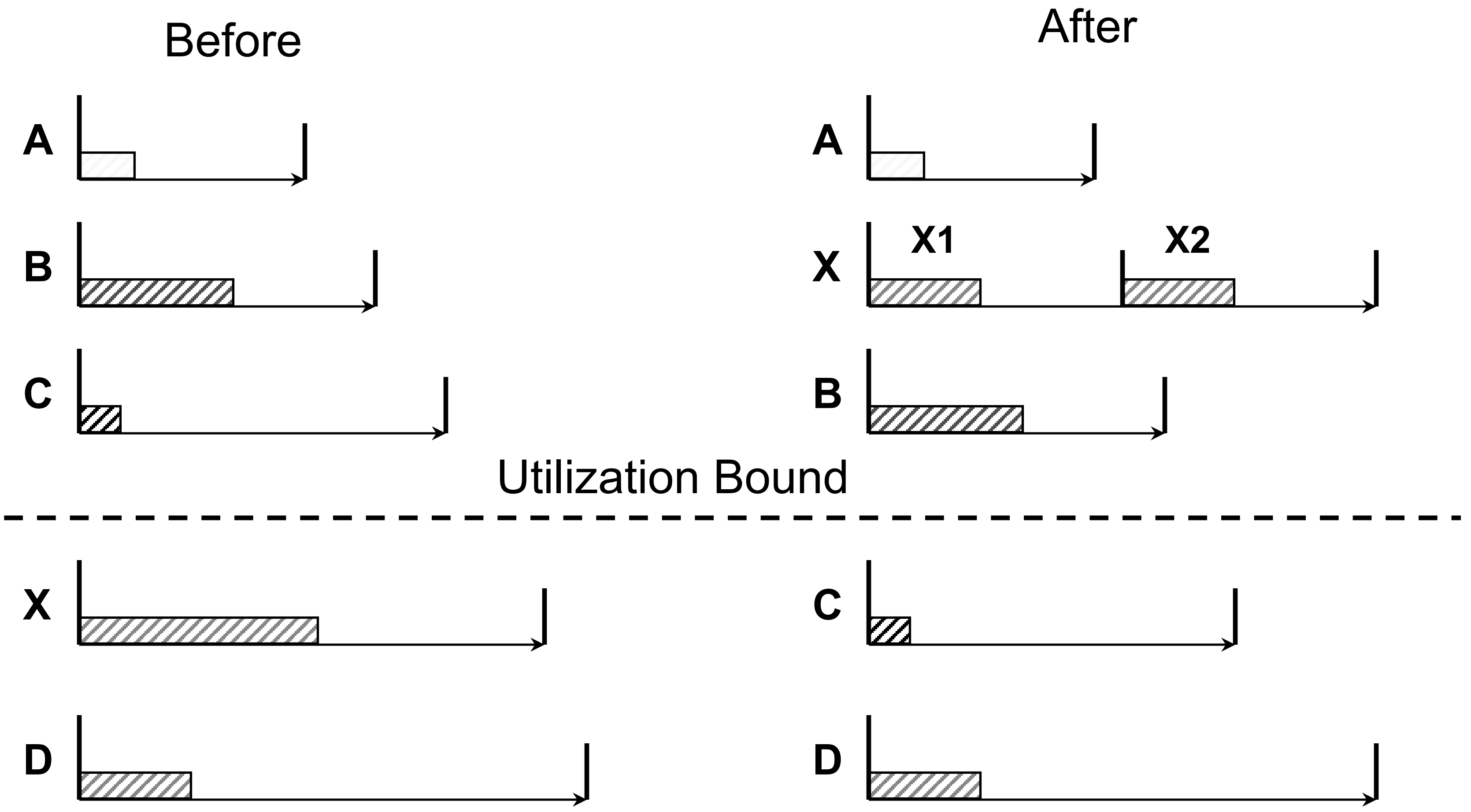
Question: “How does one insure a task with a lower period, but high criticality can be scheduled properly?”

Solution: Period Transformation

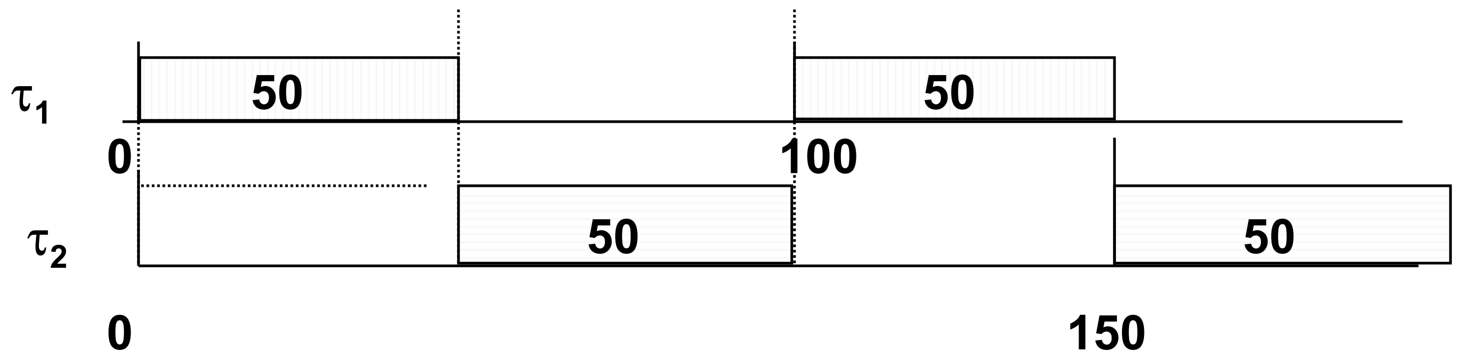
For example, cut T to T/2 period, which increases its priority for RMA, but suspend the task after C/2 execution. The suspension can be done from within the algorithm itself, or by using the run-time scheduler of the OS.

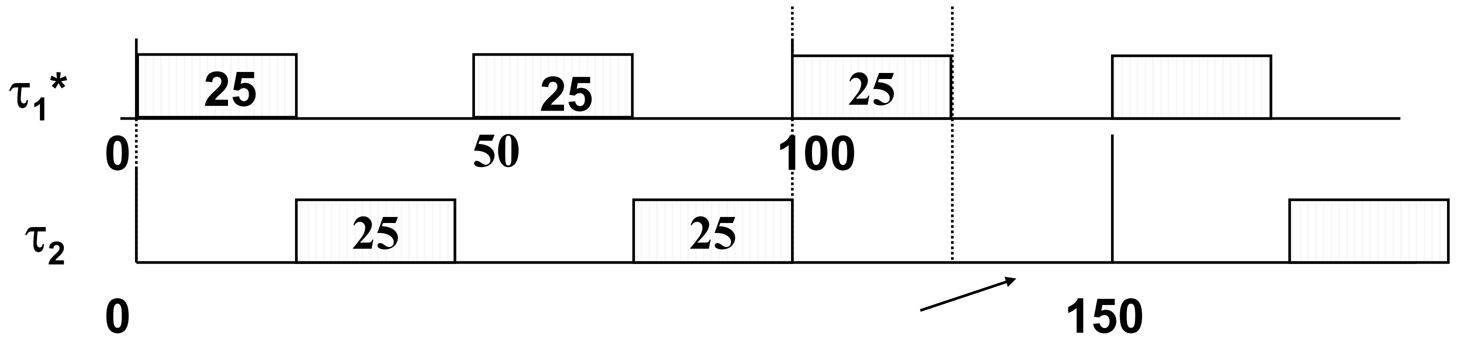
The cost to pay is the increased context scheduling cost from 2S/T to 2S/(T/2) = 4S/T, doubling the context switching cost. In general, if we divide the period by n times, the context switching cost increases n times. There is no free lunch.

Promoting a Critical Task



Using Period Transformation to Improve Processor Utilization





25 unused units are created by period transformation which can now be used by these two tasks.

Net Utilization and Overhead

Thus we make the following two observations:

- 1. An instance of a periodic task can cause at most two preemptions
- 2. In the worst case, every instance of the task causes two preemptions
- For simplicity, we use the worst case assumption. This gives us the formulae below:

- The net processor utilization is $\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n}$

- The context switching overhead is $\frac{2S}{T_1} + \dots + \frac{2S}{T_n}$

Class Exercise: Net Processor Utilization Gain

- C_1 = 49, C_2 = 49 and 2S = 1. The total utilization is 50/100 + 50/150 = 83.3%. The context switching overhead was 1/100 + 1/150 = 1.7%. The net utilization is 83.3% - 1.7% = 81.6%. We also note that the processor is fully utilized. That is we cannot increase C_1 or $C_{2.}$

- Suppose that we have performed a period transformation by cutting task 1's period from 100 to 50 and that the context switching overhead 2S = 1.

- What is the new context switching overhead in terms of percentage of processor utilization? What is the net increase of overhead?

- By how much can task 2's execution time be extended, and what is the new net processor utilization after the extension?

Net Processor Utilization Gain

- The context switching overhead was 1/100 + 1/150 = 1.7%. The new overhead is 1/50 + 1/150 = 2.7%, a net increase of 1%.
- But the net processor utilization can be now increased. Solving $(C_2+1)/150 = 1 – (24.5+1)/50$, we find that we can increase task 2's execution time from 49 to 72.5. In this case, the net processor utilization is improved from (49/100+49/150)= 81.6 % to (72.5/150 + 24.5/50) = 97.3 percent, a net improvement of 15.7%.
- In practice, most of the time, you do not end up with harmonics. So you need to use binary search and the exact test. In this approach, you use $C_1^* = (C_1 + 1)$ and guess the C_2^* value and then test the guess using exact test, adjust the guess accordingly and test again, until a good numerical answer is found.
- We can, of course, increase C1 is instead of C2 or some combination of both.

Summary

- We have studied
  - the effect of task switching times
  - transient overloads using period transformation
  - improving processor utilization using period transformation

- So far, we studied only the effect of preemption, where short period tasks are assigned higher priority and delay the execution of longer period tasks.

- Next, we will study the effect know as blocking, where short period tasks are being delayed by longer period tasks.