Lab: Response Time Analysis using FpsCalc
Course: Real-Time Systems
Period: Autumn 2024

Lab Assistants

Chengzi Huang, Duc Anh Nguyen, Behnam Khodabandeloo

Introduction

The purpose of this assignment is to give students a chance to practice the theories covered by lecturers and course literature. The assignment will be focusing on the Rate Monotonic (RM) and Fixed Priority Schedulability (FPS) analyses. The assignment will also cover how to take into account factors like blocking and jitter in the analysis.

The assignment should be solved and submitted by groups. For more information on report procedures please see section Report on page 11.

FpsCalc

FpsCalc is a tool for performing Fixed Priority Schedulability analysis and is a recommended help for solving the given assignments.

Please read FpsCalc User's Manual downloaded from the lab webpage carefully to get an idea about how to use it. Follow the instruction on Studium to see how to get the FpsCalc file and how to execute it

Notation

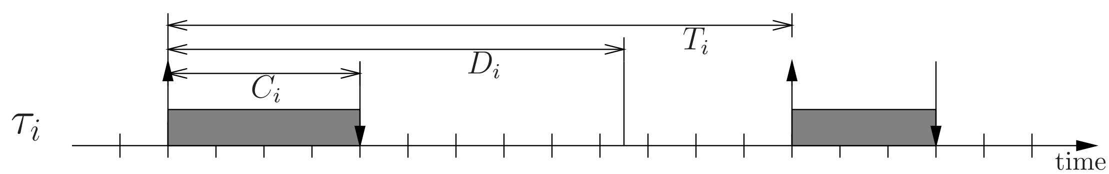

Figure 1: Typical parameters of a real-time task

We can illustrate a tasks timely behavior using a time line as in Figure 1, where the gray regions indicate that the task τ_i is executing. We will use the following notation for describing tasks:

- T_i is the period of task τ_i. For aperiodic tasks we use T_i to denote the *minimum inter-arrival time* for task instances.
- C_i is the Worst Case Execution Time (WCET) of task τ_i. Please observe that C_i should be a safe upper estimate of the execution time for the task, but this does not mean that τ_i will execute whole C_i time every time it gets released (as illustrated in Figure 1).

D_i is the relative deadline of task τ_i. I.e. the maximum allowed time between the arrival (release) and the completion of an instance.

P_i is the priority of task τ_i.

B_i is the longest time task τ_i might be blocked by lower priority tasks, (see Section Blocking on page 5).

J_i is the worst case jitter for task τ_i (see Section Jitter on page 8).

R_i is the worst case response time for task τ_i. The goal of most schedulability analysis is to find R_i and verifying that $R_i \leq D_i$.

We also let $lp(i)$, $ep(i)$ and $hp(i)$ denote the set of tasks with priority less than, equal to and higher than task τ_i respectively. In the $ep(i)$ also task τ_i will be included.

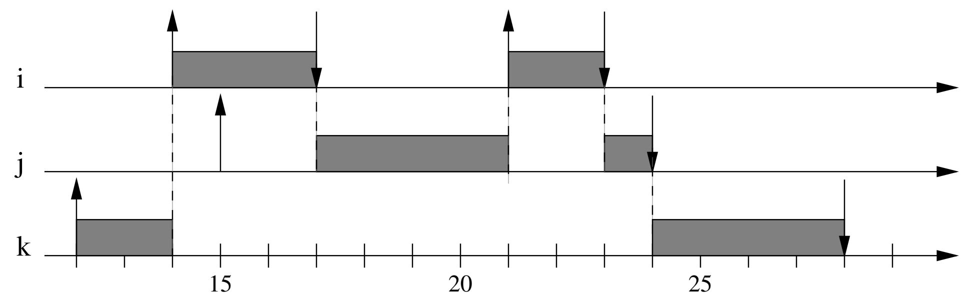

Figure 2: Example of a preemptive schedule

In fixed priority scheduling each task will have a priority assigned to it before execution which will not change over time.

A system which have several tasks executing can be illustrated by a time table as in Figure 2. The switch between tasks can either be *preemptive* - the kernel can stop a task to let another task run, and *non-preemptive* - a task that has started must complete its execution before we can start another task. The example given in Figure 2 illustrates an execution using a preemptive scheduler, where task τ_i has the highest and τ_k the lowest priority. An $\uparrow$-arrow indicates that a task gets released and an $\downarrow$-arrow that a task has finished its execution.

In more detail, the table in Figure 2 illustrates one execution where task τ_k starts executing at time 12. At time 14 task τ_i gets released and preempts task τ_k. At time 15 task τ_j gets released but can not start executing because the higher priority task τ_i is executing. When a task finish executing the highest priority waiting task are allowed to execute again, as happens at time 17 when task τ_j starts to execute. At time 21 a new instance of the high priority task τ_i gets started and preempts τ_j. Finally at time 24 when both task τ_i and τ_j has finished their executions task τ_k can continue its preempted execution.

You should preferably use this type of time tables and the given notations when presenting solutions to the given assignments.

Rate Monotonic

The *Rate Monotonic* (RM) priority ordering assigns priorities to tasks according to their periods. Specifically, tasks with higher request rates, i.e. tasks with smaller periods, get higher priorities. I.e. the task with the shortest period gets the highest priority and the task with longest period gets the lowest priority. Tasks with higher priority can preempt lower priority tasks. To get the RM analysis to work all the tasks must have the characteristics of being periodic, independent and having deadline equal to period. The RM priority assignment is *optimal* meaning that if the task set is schedulable with a given fixed-priority assignment it is also schedulable with the RM priority assignment.

The *utilization* of a system of n tasks is the fraction of total execution time spent in executing the tasks, i.e. the computational load of the system. An upper *utilization bound* U of a system with n periodic tasks can be derived as:

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

It can be shown that a set of n independent periodic tasks, with deadlines equal to periods and scheduled by the RM algorithm will always meet its deadlines, for all task phasing, if

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{2}$$

We can see that when $n \to \infty$ the $n(2^{1/n} - 1)$ expression approaches $ln\ 2 \approx 0.693$.

The *critical instant* of a task is the time at which the release of a task will produce the largest response time for the task. There exists a theorem saying that the critical instant of a task τ_i occurs whenever the task is released simultaneously with all tasks with higher priority than τ_i, (i.e. the tasks in $hp(i)$). The most of the following discussions will rely on this theorem for their correctness.

The simple version of the critical instant theorem is only valid when the tasks are independent, fixed priority and runs with perfect periodicity. In the latter sections you will investigate how to change the theorem and corresponding response time formulas to handle factors like blocking and jitter.

Assignment 1

Task	C_i	T_i	D_i
τ_1	2 ms	10 ms	10 ms
τ_2	4 ms	15 ms	15 ms
τ_3	10 ms	35 ms	35 ms

Figure 3: A number of periodic tasks with $D_i = T_i$

Given the task set in Figure 3:

1.1 *What is the priority ordering for the tasks using the RM priority ordering?*

1.2 *Will all tasks complete before their deadlines according to the schedulability formula in Equation 2 on page 3? Draw a critical instant schedule for the given tasks (as in Figure 2, but have all tasks simultaneously released at time 0). What is the system utilization bound?*

1.3 *Assume that we keep all the parameters in Figure 3 and only increase the computation time for task τ_1 to be $C_1 = 5$. Will all task complete before their deadlines? Draw a critical instant schedule for the given tasks. What is the system utilization bound?*

1.4 *Assume that we instead of modifying τ_1, (set $C_1 = 2$), want to increase the computation time for task τ_3 to be $C_3 = 17$. Will all task complete before their deadlines? Draw a critical instant schedule for the given tasks. What is the system utilization bound?*

1.5 *What conclusion can you draw from all this for the schedulability formula in Equation 2 on page 3? Specify your conclusion in terms of the system utilization bound, the $n(2^{1/n} - 1)$ expression and* 1.00.

1.6 *Now, assume that we change the relative deadline for task τ_3 in Figure 3 to be $D_3 = 15$. Please draw a demand bound function (check lecture slide scheduling theory part2.pdf) of the changed task set and explain if it's feasible on a single preemptive processor.*

Deadline Monotonic and Rate Monotonic Analysis

The RM priority ordering is not very good when we have tasks with deadline smaller than period, ($D < T$): an infrequent but periodic task would still be given a a low priority (because T is large) and hence probably miss its deadline. For a set of periodic independent tasks, with deadlines within the period, the optimum priority assignment is the *Deadline Monotonic* (DM) priority ordering, which has the following characteristics:

- Priorities are assigned according to task deadlines.
- A task with shorter deadline is assigned a higher priority.
- Tasks with higher priority can preempt lower priority tasks.

The RM priority ordering can be seen as a special case of DM priority ordering. For tasks scheduled by the DM priority ordering there does not exists any simple test for schedulability, like Equation 2 on page 3, but instead we have to calculate the worst case response time for each task separately and compare it to its deadline.

The worst possible response time for a task with any fixed priority assignment policy, (having $D_i \leq T_i$), e.g. DM or RM priority ordering, can be calculated using a response time formula:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3}$$

where $\lceil x \rceil$ is the ceiling operator and calculates the smallest integer $\geq x$. Note that the response time variable R_i is present on both sides of the equation.

The response time analysis builds upon the critical instant theorem and calculates the time a task will have completed if it is activated at the same time as all other tasks with higher priority. The summation gives us the number of times tasks with higher priority will execute before task τ_i has completed. The response time for a task can be solved using iteration:

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{4}$$

The iteration starts by assuming that the response time is equal to 0. It continues thereafter until two on each other following values of R becomes equal or R becomes larger than its deadline D.

This kind of fixed point response time analysis is (for historical reasons) called *Rate Monotonic Analysis.*

Assignment 1 cont.

1.7 *Insert the task set given in Figure 3 on the preceding page in FpsCalc and calculate the response time of each task. What is the worst-case response time for each task using FpsCalc? Verify that the times correspond to the times you extracted using a critical instant schedule. In the worst case scenario, how many instances of τ_1 and τ_2 respectively can appear during one execution of τ_3? How does this values relate to the $\left\lceil \frac{R_i}{T_j} \right\rceil$ expression? How long time of the worst case response time of τ_3 is spent waiting for instances of τ_1 and τ_2 respectively?*

Submission: *Please write your answer in the pdf file and submit the fps files for* 1.2, 1.3, 1.4, *and* 1.7.

Assignment 2

2.1 *Given the task set in Figure 4 on the next page what is the priority ordering for the tasks using the DM priority ordering? What is the priority ordering using RM priority ordering? Use FpsCalc to calculate the response time for the tasks in both ordering. Will all tasks complete before their deadlines? If you are not convinced by the formulas you can do a critical instant schedule.*

Task	C_i	T_i	D_i
τ_1	2 ms	20 ms	6 ms
τ_2	3 ms	7 ms	7 ms
τ_3	5 ms	14 ms	13 ms
τ_4	4 ms	100 ms	60 ms

Figure 4: A number of periodic tasks with $D_i \leq T_i$

2.2 *Sometimes it is preferable to not use strict RM or DM priority assignment when giving priorities to tasks. This can for example happen when we want to give a task with low deadline demands a better service rate or when the system is part of a larger distributed system.*

Find two different priority assignments of the tasks in Figure 4 which is neither RM or DM and where deadlines are missed and met respectively.

2.3 *Assume that we want to implement the tasks given in Figure 4 on a RT-kernel that only supports 3 priority levels and where tasks with the same priority will be handled in FIFO order by the scheduler. Assume that task τ_2 and τ_3 are set to have the same priority and that we use a DM priority assignment.*

Define how the response time formula in Equation 3 on page 4 will be changed when we allow several tasks to have the same priority. Make sure that it is shown in your formula that a task, due to the FIFO order, might have to wait for one instance, but can't be preempted, of an equal priority task. How will the corresponding FpsCalc formula look like, (observe that `sigma(ep, ...)` *includes the current task, τ_i)?*

What will now the worst case response time for each task be? Will all tasks meet their deadlines? Will the worst case response time for task τ_1 or τ_4 be affected? Conclusions?

Submission: *Please write your answer in the pdf file and submit the fps files for* 2.1 *and* 2.3.

Blocking

One of the restrictions with the previous analyses is that no tasks are allowed to block or voluntary suspend themselves. This means that it is very difficult to share resources, such as data, between tasks. In almost all systems where you have several tasks you also have shared resources, e.g. non-preemptible regions of code, FIFO queues and synchronization primitives. The code where a task is accessing shared resources are called *critical sections*, and must often be protected, e.g. by using semaphores.

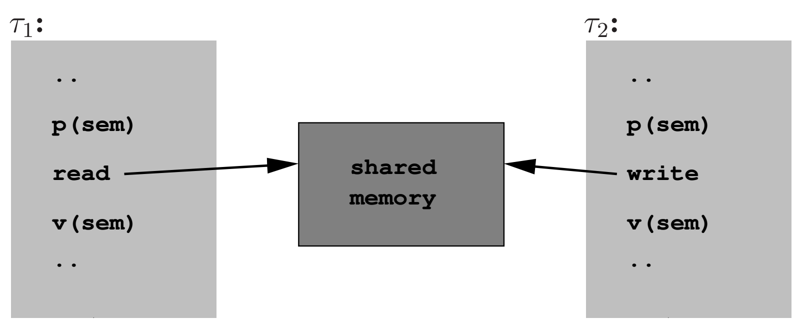

Figure 5: Reading and writing of shared memory using semaphores

The problem with using protection mechanisms, such as semaphores, is that high priority tasks might get blocked by lower priority tasks while waiting for a semaphore to be released.

Assignment 3

Task	C_i	T_i	D_i
τ_1	2 ms	10 ms	5 ms
τ_2	3 ms	20 ms	12 ms
τ_3	10 ms	40 ms	40 ms
τ_4	4 ms	100 ms	50 ms

Figure 6: Four example tasks

3.1 *Assume that we have the tasks shown in Figure 6. Give priorities to the tasks according to the DM priority assignment. Will all tasks always meet their deadlines?*

3.2 *Assume that task τ_2 and τ_4 in Figure 6 are sharing a semaphore S_1 and that τ_2 and τ_4 executes for at most 1 ms and 2 ms respectively in the critical section. Show, by doing a critical instant scheme that the deadline for task τ_2 can be missed if semaphores and no mechanism for limiting priority inversion (see below) is used. Tip: You might have to model that τ_4 has taken the semaphore, just before the moment where you let all the other tasks start executing at the same time.*

The phenomena that a higher priority task not only can be blocked by tasks accessing the same resource but also be delayed by tasks with priorities between the blocking task and the higher priority task is called *priority inversion.*

One protocol that handles the priority inversion is *the priority inheritance protocol.* It works by letting a low priority task that has a semaphore inherit the priority of the highest priority task blocked waiting for the semaphore. When the lower priority task releases the semaphore it will go back to the priority it had before. Notice that the priority of a task now is dynamic so the name 'Fixed Priority' is not really true anymore.

Assignment 3 cont.

Semaphore	Accessed by	Time
S_1	τ_2	1
S_1	τ_4	2
S_2	τ_2	1
S_2	τ_3	5

Figure 7: Tasks semaphore usage

3.3 *Assume that task τ_2 not only is sharing a semaphore S_1 with τ_4 but also is sharing a semaphore S_2 with task τ_3 as given in Figure 7. Also assume that the times the task τ_2 is accessing the semaphores S_1 and S_2 do not overlap. Show, by doing a critical instant scheme that deadline for task τ_2 can be missed even though the priority inheritance protocol is used.*

If we want to add the cost for blocking, B_i, in our response time formula we can do it like:

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

For a task τ_i, let m be the number of semaphores shared by a lower priority task (than τ_i) and a higher priority task. Using the priority inheritance protocol, it can be proved that τ_i can be blocked at most m times. It can also be proved that if we are using the priority inheritance protocol τ_i can be blocked at most once per lower priority task, and only during one critical section per semaphore. Secondly, for a lower priority task to block τ_i it must access a semaphore which a task with higher or equal priority to τ_i also is accessing. It gets quite complicated to calculate an exact blocking value for the priority inheritance protocol, mainly due to that all combinations of lower priority tasks and semaphores must be tried, (we get a problem with exponential complexity).

Instead, a simpler, (but somewhat pessimistic), algorithm for calculating the blocking B_i for τ_i can be used: 1. for each semaphore accessed both by a task with priority lower than τ_i and by a task with higher or equal priority than τ_i we extract the maximum time any lower priority task is accessing the semaphore. 2. set B_i to be the sum of the m largest timing values extracted in step 1. The overestimation we get is that the same lower priority task might cause two or more max terms in the summation.

Assignment 3 cont.

3.4 *What are the blocking times for the tasks in Figure 6 using the priority inheritance protocol? Make sure that the amount of blocking corresponds to your drawn schedule. Tip: several tasks will experience blocking. What are the response times for the tasks when using the priority inheritance protocol?*

Schemes that try minimizes the blocking time of high priority tasks is the *priority ceiling protocol* and the *immediate inheritance protocol*. Since they have the same worst case blocking time we will only investigate the immediate inheritance protocol.

The immediate inheritance protocol works by, in advance for each semaphore, calculate the highest priority of all tasks that will lock the semaphore. This is called the *priority ceiling* for the semaphore. When a task is locking the semaphore, its priority will *immediately* be raised to the maximum of its current priority and the priority ceiling of the semaphore. When the tasks leaves the semaphore its priority will go back to what it was before.

It can be proven, when using the priority ceiling protocol or the immediate inheritance protocol, that a higher priority task τ_i can be blocked at *most once* by a lower priority task. The blocking time is at most *one* critical section of any lower priority task locking a semaphore with ceiling greater than or equal to the priority of task τ_i. The blocking term can be given by:

$$B_i = max_{\{\ j,s\ |\ j \in lp(i)\ \wedge\ s \in cs(j)\ \wedge\ ceil(s) \geq P_i)\ \}} CS_{j,s}$$

Short explanation: 1. Look at all tasks with priority lower than τ_i. 2. Look at all semaphores that these tasks can lock and single out the semaphores where the ceiling of the semaphore is higher than or equal to the priority of τ_i. 3. For each semaphore extracted in step 2. extract the largest computation time that the semaphore is locked by one task with priority lower than τ_i, ($CS_{j,s}$ is the computational time for task τ_j to execute in the s:th critical section). 4. Set the largest computation time extracted in step 3. to the blocking time B_i of task τ_i.

Observe that you can use FpsCalc to double-check your results.

Assignment 3 cont.

3.5 *What are the priority ceilings for the semaphores S_1 and S_2? What are the blocking times for the tasks in Figure 6 using the immediate inheritance protocol? Tip: several tasks will experience blocking. Explain why task τ_3 can experience blocking even though it does not share any semaphore with τ_4. Will all tasks complete before their deadlines?*

Submission: *Please write your answer in the pdf file and submit the fps files for* 3.1, 3.4, *and* 3.5.

Jitter

In the simple model, all processes are assumed to be periodic and to be released with perfect periodicity. This is not, however, always a realistic assumption. *Jitter* is the difference between the earliest and latest start of the task execution relative to the periodic arrival of the task. I.e. it is not certain that a task can start executing with a precise periodicity even if there is no higher priority tasks executing at the same time. The jitter J_i^k for a task instance τ_i^k of a task type τ_i is the difference between the arrival time of τ_i^k and the time when it gets released. Let J_i^{max} denote the maximal jitter for the task type τ_i (over all possible instances in a given system), $J_i^{max} = \max\{J_i^k | \forall k\}$. Let J_i^{min} denote the minimal jitter for the task type τ_i (over all possible instances in a given system), $J_i^{min} = \min\{J_i^k | \forall k\}$. The jitter J_i for a task τ_i is the difference between the maximal jitter and the minimal jitter, $J_i = J_i^{max} - J_i^{min}$. Figure 8 illustrates how the jitter of a task instance relates to some other already known concepts.

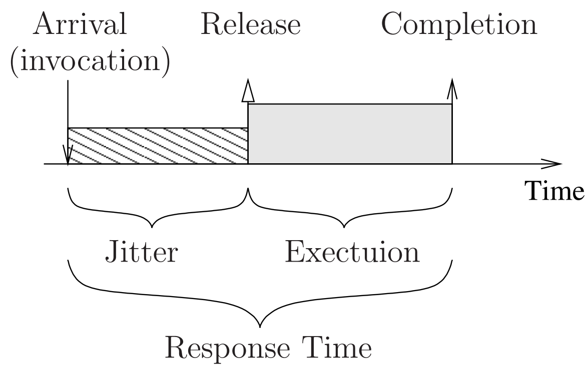

Figure 8: Arrival, release, start of the execution, and jitter of one concrete task instance. The release time and the start of the execution coincide in this example.

One source of jitter is a tick-driven scheduler, i.e. we can only do context switches at regular intervals invoked by the timer hardware. For example see Figure 9 where a tick-driven scheduler is invoked, by a timer-interrupt, every 10th ms, to see if any new task instance has arrived. In case Figure 9(a) we are lucky, since τ_i:s instance arrives just before the scheduler got invoked and τ_i will start to execute immediately after. In case Figure 9(b) we are more unlucky since τ_i:s instance arrives just after the last instance of the scheduler. We now have to wait 10 more ms before the scheduler gets invoked and performs a context switch to task τ_i. The jitter of task τ_i will therefore be 10 ms, ($|0 - 10| = 10$), due to the tick-driven scheduler.

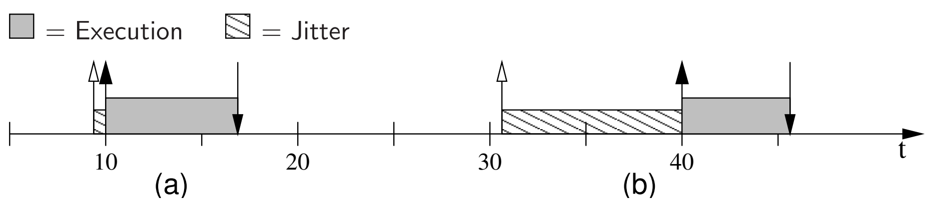

Figure 9: Release jitter due to tick scheduling every 10ms

Assignment 4

4.1 *Another source of jitter is varying execution and response times for tasks (or messages) that start other tasks. To illustrate this assume a system with two tasks τ_1 and τ_2. Let task τ_1 have a period $T_1 = 10$ and a fixed (non-varying) execution time $C_1 = 3$. Let task τ_2 arrives always 2 time units* *after τ_1 arrives, but it waits for τ_1 finishing before it gets released (we can see it as that τ_2 waits for the result of τ_1 before it can execute). Assume task τ_1 always ends its execution by releasing task τ_2 (τ_2 can now be scheduled for execution). Let τ_2 have a worst case execution time of $C_2 = 2$.*

Draw a schedule that shows two instances of τ_1 and τ_2 respectively. What is the period T_2 of task τ_2?

4.2 *To illustrate that varying execution time of τ_1 might cause jitter of τ_2 assume that τ_1's execution time no longer is fixed but varies between $C_1^{min} = 3$ and $C_1^{max} = 5$. Task τ_1 still starts task τ_2 at the end of its execution.*

Draw a schedule with two instances of τ_1 that shows that the varying execution time of τ_1 might give raise to jitter of τ_2. What is the jitter that task τ_2 can experience?

4.3 *To illustrate that interference of high priority tasks might give raise to further jitter of low priority tasks we add a task τ_0 to the system. Let task τ_0 have higher priority than both τ_1 and τ_2, a period $T_0 = 20$ and a worst case execution time $C_0 = 2$.*

Draw a schedule that shows that varying response time of τ_1 due to interference of τ_0 will give raise to further jitter of τ_2. Task τ_1's execution time still varies between C_1^{min} and C_1^{max}. What is the jitter that τ_2 can experience due to varying response and execution time of τ_1?

After the above example of how jitter can be derived we will now investigate how jitter of different tasks might interact to cause larger response times of lower priority tasks. Note that for the tasks given in figure 10, the jitter is raised by other unknown sources. It **does not** accounts for the jitter comming from the blocking from higer priority tasks, as in the above questions. That why in the formula below we still factor in the blocking term from higher priority tasks.

Assignment 4 cont.

Task	C_i	T_i	D_i	J_i
τ_A	5 ms	20 ms	10 ms	5 ms
τ_B	30 ms	50 ms	50 ms	10 ms

Figure 10: Two example tasks with jitter

4.4 *For the given tasks τ_A and τ_B in Figure 10 with DM priority ordering, what is the worst case response time for respective task assuming that we have no jitter. Will both tasks be able to complete before their deadlines?*

Assuming that the tasks are experiencing jitter as given in Figure 10 we can use the formula given in Equation 5 to calculate how this will affect the worst case response time of the tasks.

$$\begin{aligned} w_i &= C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i + J_j}{T_j} \right\rceil C_j \\ &= C_i + \sum_{j \in hp(i)} \left(1 + \left\lceil \frac{w_i - (T_j - J_j)}{T_j} \right\rceil \right) C_j \\ R_i &= w_i + J_i \end{aligned} \tag{5}$$

Assignment 4 cont.

4.5 *Given the formula in Equation 5 what is the worst case response time for respective tasks assuming that they can experience jitter? Will both tasks be able to complete before their deadlines?*

We can construct a task schedule giving the same worst case response time for tasks as the formula in Equation 5.

To construct it we can observe that the critical instant is when both tasks *start executing* at the same time (i.e. not when they arrive at the same time) *and* both tasks experience maximum jitter for their first, (but not the following), instances. As illustrated in Figure 8 the response time of a task will be the time from which it wants to start executing (arrival time) and the time when it completes its execution. When a task experiences jitter another task can of course execute (if it wants to).

Looking in more detail at the formula in Equation 5 you can see that it is divided into two parts. The $R_i = \ldots$ part express that task τ_i can not get preempted when it experiences jitter. The jitter J_i must still be added to the response time for task τ_i (as illustrated in Figure 8).

The $w_i = \ldots$ part represent the time during which τ_i can get preempted by higher priority tasks. The $(1 + \lceil \frac{w_i - (T_j - J_j)}{T_j} \rceil)$ expression captures the release scheme of a higher priority task τ_j in which it will be able to preempt τ_i as much as possible. This is, in the worst case the first instance of the τ_j will have maximum jitter, while the rest of the instances will have minimum jitter.

Assignment 4 cont.

4.6 *Draw a task schedule for the tasks τ_A and τ_B in Figure 10 which gives the same worst case response times as the formula in Equation 5. Indicate arrival, jitter, beginning of execution, execution, preemption, and completion for each task in your schedule.*

$$dbf(\tau, t) = \sum_{i=1}^{n} \lfloor \frac{t + T_i - D_i}{T_i} \rfloor C_i = \sum_{i=1}^{n} \lfloor \frac{t - D_i}{T_i} + 1 \rfloor C_i$$

Submission: *Please write your answer in the pdf file and submit the fps files for* 4.4 *and* 4.5.

Report

A proper electronic report is to be handed in for the assignment. It should include at least the following:

- Answers to the assignments. When asked for, illustrate your answers by drawing timing schedules.
- All the fps files.

The assignment must be handed in no later than Monday , 9th Dec., 23:59.

Some general guidelines of how to make a report:

- Use Latex or Word to type your solutions. Pictures and diagrams may be drawn by hand and inserted to your report as picture.
- Provide answers to the questions. Answers should be extracted from FpsCalc files and results. (I.e. do not give answers like “see foo.fps for equation”.)