1 Introduction

This report presents a proof to support the correctness of the schedulability test for self-suspending real-time task systems by Jane W. S. Liu in the textbook [4], since the proof was not provided in the textbook.

We define the terminologies as follows: A sporadic task τ_i is released repeatedly, with each such invocation called a job. The j^{th} job of τ_i, denoted $\tau_{i,j}$, is released at time $r_{i,j}$ and has an absolute deadline at time $d_{i,j}$. Each job of any task τ_i is assumed to have worst-case execution time C_i. Each job of task τ_i suspends for at most S_i time units (across all of its suspension phases). When a task suspends itself, the processor can execute another task. The response time of a job is defined as its finishing time minus its release time. Successive jobs of the same task are required to execute in sequence. Associated with each task τ_i are a period (or minimum inter-arrival time) T_i, which specifies the minimum time between two consecutive job releases of τ_i, and a deadline D_i, which specifies the relative deadline of each such job, i.e., $d_{i,j} = r_{i,j} + D_i$. The worst-case response time of a task τ_i is the maximum response time among all its jobs. The utilization of a task τ_i is defined as $U_i = C_i/T_i$.

Here, in this report, we focus on constrained-deadline task systems, in which $D_i \leq T_i$ for every task τ_i. We only consider preemptive fixed-priority scheduling on a single processor, in which each task is assigned with a unique priority level. We assume that the priority assignment is given.

We will focus on the analysis of task τ_k. There are $k-1$ higher-priority tasks, i.e., $\tau_1, \tau_2, \ldots, \tau_{k-1}$, than task τ_k. The task with a smaller index has higher priority, i.e., task τ_i is with a higher-priority level than task τ_{i+1}. When performing schedulability analysis of task τ_k, we assume that $\tau_1, \tau_2, \ldots, \tau_{k-1}$ are already verified to meet their deadlines. We also classify the $k-1$ higher-priority tasks into two sets: $\mathbf{T}_1$ and $\mathbf{T}_2$. A task τ_i is in $\mathbf{T}_1$ if $C_i \geq S_i$; otherwise, it is in $\mathbf{T}_2$.

2 Liu's Analysis

To analyze the worst-case response time (or the schedulability) of task τ_k, we need to in general quantify the worst-case interference of the higher-priority tasks during the execution of a job of task τ_k. In the ordinary sporadic real-time tasks, i.e., $S_i = 0$ for every task τ_i, the so-called critical instant theorem by Liu and Layland [3] is commonly adopted. That is, the worst-case response time of task τ_k (if it is less than or equal to its period) happens when task τ_k and all the higher-priority tasks release a job at the same time and the subsequent jobs as early as possible (by respecting to the periods).

The critical instant theorem does not work for self-suspending sporadic task models. Jane W. S. Liu [4] explains a way to handle self-suspending task models by modeling the *extra delay* suffered by a task τ_k due to self-suspending behavior as a factor of blocking time, denoted as B_k, as follows:

- The blocking time contributed from task τ_k is S_k.

– A higher-priority task τ_i can only block the execution of task τ_k by at most $b_i = min(C_i, S_i)$ time units.

Therefore,

$$B_k = S_k + \sum_{i=1}^{k-1} b_i. \tag{1}$$

In the textbook [4], the blocking time is then used to perform utilization-based analysis for rate-monotonic scheduling, i.e., if $\frac{C_k+B_k}{T_k} + \sum_{i=1}^{k-1} U_i \leq k(2^{\frac{1}{k}} - 1)$, then task τ_k can be feasibly scheduled by using rate-monotonic scheduling if $T_i = D_i$ for every task τ_i in the given task set. If the above argument is correct, we can further reach the following argument. That is, a constrained-deadline task τ_k can be feasibly scheduled by the fixed-priority scheduling if

$$\exists 0 < t \leq D_k, \qquad C_k + B_k + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil C_i \leq t. \tag{2}$$

However, as there was no proof in the textbook [4], to support the correctness of the above tests, we provide a proof in this report.

3 Proof of Liu's Analysis

This section provides the proof to support the correctness of the test in Eq. (2). First, it should be easy to see that we can convert the suspension time of task τ_k into computation. This has been done by many researchers, e.g., the proof in Lemma 3 in the paper by Liu and Chen [2], Nelissen et al. [1], etc. The remaining part is to show that the additional interference due to self-suspension from a higher-priority task τ_i is at most $b_i = min(C_i, S_i)$. The interference to be at most C_i has been provided in the literature as well, e.g., [5][2]. However, the argument about blocking task τ_k due to a higher-priority task τ_i by at most S_i amount of time is not very clear.

From the above discussions, we can greedily convert the suspension time of task τ_k to its computation time. For notational brevity, let C'_k be $C_k + S_k$. We call this converted version of task τ_k as task τ'_k. Our analysis is also based on a very simple observation as follows:

Lemma 1. *For a preemptive fixed-priority schedule, removing a lower-priority job arrived at time t does not change the schedule for executing the higher-priority jobs after time t.*

Proof. This is due to the preemptive scheduling. The removal of the lower-priority job has no impact at all on the higher-priority jobs. □

Lemma 2. *For a preemptive fixed-priority schedule, if the worst-case response time of task τ_i is no more than its period T_i, removing a job of task τ_i does not change the schedule for the remaining jobs of task τ_i.*

Proof. The removal of the job of task τ_i has no impact on the higher-priority jobs as in Lemma 1. Since the worst-case response time of task τ_i is no more than the period, the execution of the other jobs of task τ_i is also not affected by the removal of the job. $\square$

We can prove the correctness of Eq. (2) by using a similar proof of the critical instant theorem of the ordinary sporadic task system. Let R'_k be the minimum $t > 0$ such that $C_k + B_k + \sum_{i=1}^{k-1} \left\lceil \frac{t}{T_i} \right\rceil C_i = t$, i.e., Eq. 2 holds. The following lemma shows that R'_k is a safe upper bound if the worst-case response time of task τ'_k is no more than T_k.

Theorem 1. *R'_k is a safe upper bound of the worst-case response time of task τ'_k in the self-suspending task system if its worst-case response time is no more than T_k.*

Proof. According to the above definitions, we only need to show that R'_k is a safe upper bound of the worst-case response time of task τ'_k by converting the suspension time of task τ_k as computation. We consider a given schedule in the task system with $\tau_1, \tau_2, \ldots, \tau_{k-1}, \tau'_k, \tau_{k+1}, \ldots$. Since we consider fixed-priority preemptive scheduling, we can safely remove all the lower priority tasks $\tau_{k+1}, \tau_{k+2}, \ldots$ without changing any execution behavior of the higher-priority tasks by Lemma 1. Suppose that the job of task τ'_k with worst-case response time arrives at time t_k and finishes at time ρ. Since we assume that the worst-case response time of task τ'_k is no more than T_k, removing all the other jobs of task τ'_k has no impact on the schedule of the job of task τ'_k arrived at time t_k, as shown in Lemma 2.

Therefore, for the rest of the proof, we only have to consider this *reduced* schedule. In this reduced schedule, the processor is busy for executing the higher-priority tasks or the job of task τ'_k from z to ρ. In the above schedule, let t_k be the latest moment before z such that the processor does not run any job. That is, from t_k to z, the processor executes certain higher-priority tasks. Apparently, we can change the release time of the job of task τ'_k to t_k. The response time of the job becomes $\rho - t_k \geq \rho - z$.

Up to here, the proof is basically similar to the proof of the critical instant theorem of the ordinary sporadic real-time task systems. However, for self-suspending task systems, we need to consider that a job of task τ_i suspends itself before t_k and resumes after t_k. Such a job is the so-called *carry-in* job. Fortunately, each higher-priority task has only one carry-in job due to the assumption that the higher-priority tasks are assumed to finish before their periods. However, analyzing the accurate workload of such jobs due to self-suspension is non-trivial.

The analysis here has two steps. First, we *extend the window of interest* from $[t_k, \rho)$ to $[t_1, \rho)$ by inspecting the above schedule carefully. The procedure starts from the lowest priority task τ_{k-1} to the highest priority task τ_1. In each iteration for considering task τ_j, we may extend the window of interest from $[t_{j+1}, \rho)$ to $[t_j, \rho)$ with $t_j \leq t_{j+1}$. After each iteration for considering task τ_j, all the jobs of task τ_j arrived before t_j will be either removed or represented by an artificial job to represent the residual workload (to be detailed later). Second, we *analyze the final reduced schedule* in the window of interest $[t_1, \rho)$ to obtain the properties of the worst-case behaviour.

Step 1: Extend the Window of Interest

In each iteration, we will define t_j for task τ_j, starting from $j = k-1, k-2, \ldots, 1$, in the revised schedule. Let y be the release time of the job (arrived before t_{j+1}) of task τ_j that has not yet finished at time t_{j+1}. There are a few cases:

- There is no such a job of task τ_j: Removing all the jobs of task τ_j arrived before t_{j+1} has no impact on the schedule of the higher-priority jobs (higher than τ_j) executed after t_{j+1} by Lemma 1. Therefore, we simply set t_j to t_{j+1} and remove all the jobs of task τ_j arrived before t_{j+1} in the schedule
- There is such a job of task τ_j with $y < t_{j+1}$: Removing all the jobs of task τ_j arrived before y has no impact on the schedule of the higher-priority jobs (higher than τ_j) executed after t_{j+1} by Lemma 1 and the assumption that the worst-case response time of task τ_j is at most $D_j \leq T_j$. Therefore, we remove all the jobs of task τ_j arrived before y in the schedule. There are two subcases:
  - If task τ_j is in $\mathbf{T}_1$, i.e., $S_j < C_j$: For such a case, we set t_j to y. Moreover, we also know that the maximum idle time of the processor from t_j to t_{j+1} is at most S_j since there is no job with priority lower than τ_j available to be executed before t_{j+1} after we remove the jobs of task τ_{j+1} in the previous iterations.
  - If task τ_j is in $\mathbf{T}_2$, i.e., $S_j \geq C_j$: For such a case, we set t_j to t_{j+1}. Let C'_j be the remaining execution time for the job of task τ_j, unfinished at time t_j. We know that C'_j is at most C_j. Here, we remove the job of task τ_j arrived at time y and release a new job with execution time C'_j at time t_j with the same priority level of task τ_j. Clearly, this has no impact on the execution of the higher-priority jobs executed after t_j. Such an amount of execution time C'_j is called *residual workload* of task τ_j for the rest of the proof.

The above construction of $t_{k-1}, t_{k-2}, \ldots, t_1$ is well-defined. After each iteration to set t_j, we can reduce the schedule by removing some jobs without affecting the schedule of the carry-in J_j. (Note that J_j is defined as the carry-in job of task τ_j at time t_k.) Therefore, the reduced schedule after the above procedure does not change the execution of J_j after time t_j if τ_j is in $\mathbf{T}_1$. For a task τ_j in $\mathbf{T}_2$, its corresponding carry-in job J_j may be changed, but its execution after t_j remains identical as in the original schedule. Therefore, the resulting schedule above does not change any execution behavior of the (at most) $k-1$ carry-in jobs at time t_k.

Analysis of the Final Reduced Schedule in $[t_1, \rho)$:

Now, it is time to look at the property of the above schedule after removing the unnecessary jobs. We know that the maximum idle time of the above schedule due to self-suspension from t_1 to t_k is at most $\sum_{\tau_i \in \mathbf{T}_1} S_i$. We can simply consider such self-suspension time as *virtual computation*. More precisely, for any t with

$t_j < t \leq t_{j+1}$ for $j = 1, 2, \ldots, k-1$, the total amount of idle time plus the *residual workload* of $\tau_i \in \mathbf{T}_2$ from time t_1 to time t is at most $\sum_{i=1}^{j} b_i$. Therefore, for $j = 1, 2, \ldots, k-1$, by the choice of t_j, we know that

$$\forall t_j \leq t < t_{j+1}, \qquad \sum_{i=1}^{j} b_i + \sum_{i=1}^{j} \left\lceil \frac{t - t_i}{T_i} \right\rceil C_i > t - t_1.$$

By further considering the time interval from t_k to ρ, we have

$$\forall t_k \leq t < \rho, \qquad C'_k + \sum_{i=1}^{k-1} b_i + \sum_{i=1}^{k-1} \left\lceil \frac{t - t_i}{T_i} \right\rceil C_i > t - t_1.$$

By the fact $t_i \geq t_1$ for $i = 1, 2, \ldots, k$, we know that $\left\lceil \frac{t-t_i}{T_i} \right\rceil \leq \left\lceil \frac{t-t_1}{T_i} \right\rceil$. Therefore, by setting θ to $t - t_1$ and the above two conditions, we know that

$$\forall 0 < \theta < \rho - t_1, C'_k + \sum_{i=1}^{k-1} b_i + \sum_{i=1}^{k-1} \left\lceil \frac{\theta}{T_i} \right\rceil C_i > \theta.$$

Since $\rho - t_k \leq \rho - t_1$, we can reach the conclusion that the minimum θ such that $C'_k + \sum_{i=1}^{k-1} b_i + \sum_{i=1}^{k-1} \left\lceil \frac{\theta}{T_i} \right\rceil C_i \leq \theta$ is a safe upper bound of the response time of task τ'_k if its worst-case response time is no more than T_k. □

To illustrate Step 1 in the above proof, we also provide one concrete example. Consider a task system with the following 4 tasks:

- $T_1 = 6, C_1 = 1, S_1 = 1$,
- $T_2 = 10, C_2 = 1, S_2 = 6$,
- $T_3 = 18, C_3 = 4, S_3 = 1$,
- $T_4 = 20, C_4 = 5, S_4 = 0$.

Figure **??** demonstrates a schedule for the jobs of the above 4 tasks. We assume that the first job of task τ_1 arrives at time $4 + \epsilon$ with a very small $\epsilon > 0$. The first job of task τ_2 suspends itself from time 0 to time $5 + \epsilon$, and is blocked by task τ_1 from time $5 + \epsilon$ to time $6 + \epsilon$. After some very short computation with ϵ amount of time, the first job of task τ_2 suspends itself again from time $6 + 2\epsilon$ to 7. In this schedule, ρ is set to $20 - \epsilon$.

We define t_4 as 7. Then, we set t_3 to 6. When considering task τ_2, since it belongs to $\mathbf{T}_2$, we greedily set t_2 to $t_3 = 6$ and the residual workload C'_2 is 1. Then, t_1 is set to $4 + \epsilon$. In the above schedule, the idle time from $4 + \epsilon$ to $20 - \epsilon$ is at most $2 = S_1 + S_3$. We have to further consider one job of task τ_2 arrived before time t_1 with execution time C_2.

References

1. G. R. Geoffrey Nelissen, José Fonseca and V. Nelis. Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

2. C. Liu and J.-J. Chen. Bursty-Interference Analysis Techniques for Analyzing Complex Real-Time Task Models. In *2014 IEEE Real-Time Systems Symposium*. Institute of Electrical & Electronics Engineers (IEEE), dec 2014.
3. C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, jan 1973.
4. J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.
5. R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings.,10th International Conference on Distributed Computing Systems*. Institute of Electrical & Electronics Engineers (IEEE), 1990.