Slack Stealing

Muhammad Amirul Hakimi Bin Zaprunnizam
Hochschule Hamm Lippstadt
Lippstadt, Germany
muhammad-amirul-hakimi.bin-zaprunnizam@stud.hshl.de

***Abstract*—In a hard real-time system, it is critical to verify that each task is not only completed properly, but also produces the correct value at the correct time. As a result, scheduling algorithms play an important part in completing a variety of tasks. The real-time operating system (RTOS) allows the user to make the most of their time and resources in order to achieve the desired result. The real-time operating system is in charge of process execution, monitoring, and control. RTOS is widely utilized in industries such as aviation, automobiles, robots, and machine manufacture. In this paper we are going to see one of example of RTOS algorithm which is Slack Stealing. The Slack Stealing algorithm is an aperiodic service technique which offers substantial improvements in response time. The idea is it can form the periodic tasks without causing their deadlines to be missed.**

I. Introduction

Over the last 10 years, scheduling approaches have been developed that allow real-time systems to be designed with predictable timing accuracy. Furthermore, these technologies have progressed to the point where many practical problems linked with these systems have been solved. The most comprehensive theoretical conclusions have been obtained for situations in which the system must process many periodic activities, such as monitoring duties in control systems. There are two common ways in this topic which is static or fixed priority algorithms. For example rate monotonic (RM) and deadline monotonic algorithms. The other way around is dynamic priority algorithms, such as the earliest deadline algorithm. These two theories are getting more well-developed, while the static priority theory is far more thorough now. The major goal of this study is to look at how to schedule hard deadline periodic activities and hard deadline aperiodic jobs in fixed priority systems together.

In real time system, it is typically consist of a set of hard deadline periodic tasks, hard deadline aperiodic tasks can emerge from a variety of sources for instance including alert conditions or failures of hard deadline periodic tasks that fail to pass validation checks and must be retried and completed before the original deadline. When all of the task timing requirements can not be met at the same time, the scheduler has to pick and select which tasks to process. An acceptance test, also known as a guaranteed algorithm, is a decision that allows you to accept or reject anything. Due to the timing requirements of aperiodic jobs are unknown before run-time, whereas the requirements of periodic tasks are known, the acceptance test must be performed online. There are numerous methods for determining which jobs to accept for processing. We take the strategy of requiring that all deadlines for all periodic tasks be met. We conduct an online acceptance test for each hard aperiodic task, subject to this constraint, to determine whether the arriving aperiodic task's timing requirements can be guaranteed, while sustaining the guarantee given to scheduler and so any already recognized but not yet accomplished aperiodic tasks. A hard aperiodic task is refused if it cannot be guaranteed. The fraction of aperiodic arrivals that can be guaranteed, the quantity of aperiodic processing that is completed, or some combination of these would be the performance requirement for such an algorithm [1].

In this paper, I will briefly explain Slack Stealing algorithm. For better understanding I will divide into a few sections which include background scheduling, model of the algorithm, optimality in hard real time and UPPAAL implementation. We will start with background scheduling which will tell about how really scheduling works in fixed priority server.

II. BACKGROUND SCHEDULING

In this section I will tell the basic idea of scheduling in the processor. It is now often knew that many operating systems that support dynamic task activation that can allow the ongoing work to be cut in or interrupted at any time and allowing a more critical activity to take over the processor without having to wait in the ready queue. The ongoing job is halted and will be put into the ready queue for a moment to make sure that the CPU is assigned to the most critical ready task that has just arrived. Preemptive multitasking distinguishes a multitasking operating system that allows task preemption from a cooperative multitasking system in which processes or tasks must be expressly configured to submit when they do not require system resources. In simpler terms, preemptive multitasking includes the use of an interrupt mechanism to suspend the presently running process while a scheduler determines which process should run next. As a result, all processes will receive some CPU time at any given time. The points below show how exactly preemptive task schedule.

- Tasks that handle exceptions may need to preempt current tasks in order to respond to exceptions in a timely manner.
- When tasks have varied levels of criticality (importance), preemption allows the most critical jobs to be completed as soon as they arrive.

- With the help of predictive scheduling, the system's efficiency can be improved. Its goal is to enable real-time task sets to be completed with higher process utilization.

The easiest way to handle a group of soft periodic tasks is to schedule them in the background when there are no periodic instances to execute. The main disadvantage of this approach is that for large periodic loads it might take a long response time of aperiodic requests some applications. As a result, background scheduling should only be used when aperiodic tasks have no strict time limitations, and the periodic load is low. The example of real time task is shown as below [2].

- Hard : A real-time task is considered difficult if failing to meet its deadline could result in disastrous effects for the system under management.
- Firm: A real-time job is said to be firm if missing its deadline causes no system damage, but the output is worthless.
- Soft: A real-time task is said to be soft if it misses its deadline but nevertheless serves some purpose for the system, despite degrading performance.

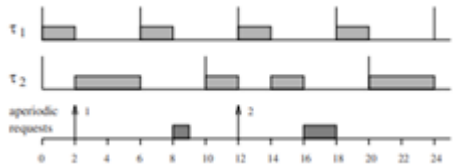

Fig. 1. Example of background scheduling of aperiodic requests under Rate Monotonic [2].

Figure 1 shows an example in which two periodic activities are scheduled using RM while two aperiodic tasks run in the background. Due to background scheduling has no effect on the execution of periodic tasks, the guaranteed test remains unchanged in the presence of aperiodic requests.

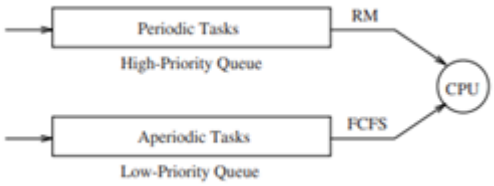

Fig. 2. Scheduling queues required for background scheduling [2].

The main advantage of using background scheduling is it simplicity. Two queues are required to implement the scheduling mechanism, as illustrated in Figure 2. The first one for periodic tasks that has higher priority and for the other one is aperiodic requests which has lower priority. The two queueing techniques are not dependent of one another and can be implemented using separate algorithms, such as RM for periodic workloads and FCFS for aperiodic requests. Aperiodic queue is going to be accepted during only periodic queue is empty. Any aperiodic tasks are instantly preempted when a new periodic instance is activated.

In fixed-priority servers there are a few algorithms that can handle many tasks. The way how it control may be different with each other. The algorithm suits the program to handle all the task. The example of the algorithms in fixed-priority servers are [2]

- Polling Server
  The server is often scheduled using the same mechanism as for periodic activities, and once operational, it fulfills aperiodic requests within its budget. Aperiodic requests can be ordered by arrival time, computation time, deadline, or any other characteristic, regardless of the scheduling technique used for periodic activities.

- Deferrable Server
  Deferrable Server (DS) is a type of periodic task that has a capacity and a period. The rate monotonic scheduling technique is used to determine the server's priority. In general, the server's duration is chosen in such a way that it becomes the most important work. For the duration of the server's period, the DS maintains its aperiodic execution time. As a result, an aperiodic request can be handled at high priority by the server at any moment, as long as the server's execution time for the current period has not been spent.The server's execution time is discarded and lost altogether if it is not utilised before the end of its period. At the start of the period, the server's high priority execution time is replenished to its full capacity.

- Priority Exchange
  The Priority Exchange (PE) method is a scheduling scheme that handles a number of soft aperiodic requests as well as a set of hard periodic tasks. PE performs slightly worse in terms of aperiodic responsiveness than DS, but it has a better schedulability bound for the periodic task set. The PE method, like DS, uses a periodic server to service aperiodic requests (typically at a high priority). It is differ from DS in the way capacity is preserved. PE, keeps its high-priority capacity by exchanging it for the time it takes to complete a lower-priority periodic activity.

- Sporadic Server
  Another option is the Sporadic Server algorithm, which improves the average response time of aperiodic jobs without lowering the utilization bound of the periodic task set. The Sporadic Server algorithm generates a high-priority task for handling aperiodic requests and,

like the DS algorithm, keeps the server capacity at that level until an aperiodic request arises. Sporadic Server, on the other hand, varies from DS in how it refreshes its capacity. Sporadic Server replaces its capacity only after it has been used by aperiodic task execution, whereas DS and PE replenish their capacity to full value at the start of each server period.

- Slack Stealing
  A natural way to improve the response times of aperiodic jobs is by executing the aperiodic jobs ahead of the periodic jobs whenever possible. This approach, called Slack Stealing. Every periodic jobs slice must be scheduled in a frame that ends no later than its deadline. When aperiodic job executes ahead of slice of periodic task then it consumes the slack in the frame. It reduce the response time of aperiodic jobs but requires accurate timer.

To summarize this section, we already know how scheduling works in the background. It might be useful or not based on user's program. We also know a few algorithms that are in the fixed priority server. More information about slack stealing will be tell in the next section.

III. MODEL SLACK STEALING ALGORITHM

Now I am going to explain how slack stealing works in real time system. Basically, Slack Stealing (SS) is algorithm to handle aperiodic service and has improvement in response time compared to Deferrable Server and Sporadic Server. SS is unique than the other service is because it does not create a periodic server for aperiodic but instead it creates a task that namely "Slack Stealer". The stealer will steal all processing time from periodic task to try to provide time for aperiodic service. To ensure the system run smoothly, the deadline of the periodic task is must not be passed. The idea is same as stealing slack from periodic task. Slack here means the time unit that is not contain any task or idle. This is how we can calculate remaining slack available in the system, $c_i(t)$ is stands for the remaining computation time at time t and the slack of a task τ_i is.

$$slack_i(t) = d_i - t - c_i(t) \tag{1}$$

To make it clear, there is no advantage of solving periodic task as soon as possible. Thus, we can complete aperiodic task first and keeping periodic task at queue by using Slack Stealer to steal available slack from periodic task. The situation will be different if no aperiodic task because periodic task will remain schedule by Rate Monotonic (RM).

Figure 3 shows the action of Slack Stealer when there are two periodic tasks. The characteristic of the tasks is

Figure 3 (a) shows RM schedule all periodic task when there is no aperiodic request, whereas Figure 3(b) shows

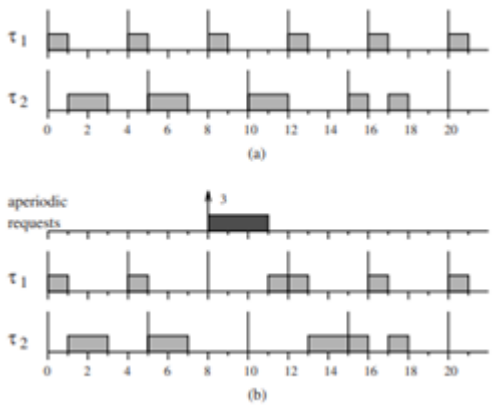

Fig. 3. Example of Slack Stealer behavior: **a**. when no aperiodic requests are pending; **b**. when an aperiodic request of three units arrives at time t = 8 [2].

TABLE I
TASK ATTRIBUTES

Periodic Task	Period/*T*	Execution Time/*C*
τ_1	4	1
τ_2	5	2

an aperiodic request of three units arrives at time, *t=8* and receives immediate service. By postponing the third instance of τ_1 and τ_2, a slack of three units is obtained in this situation. For example, because $U_1 = 1/4$ and $U_2 = 2/5$, the P factor for the task set is P = 7/4; thus, according to Equation (2), the maximum server usage is [2]

$$U_{SS}^{max} = \frac{2}{P} - 1 = \frac{1}{7} \simeq 0.14 \tag{2}$$

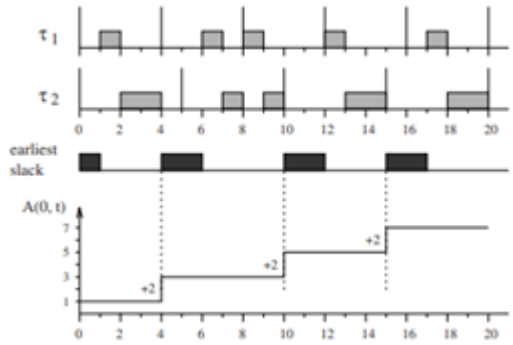

Fig. 4. Slack function at time s = 0 for the periodic task set considered in the previous example [2].

If we recall there is no other algorithm for example Polling Server, Deferrable Server and Priority Exchange that can schedule aperiodic task at highest level with missing deadline for periodic task. As show as Figure 4, even with $C_s = 1$, the shortest server period that can be set with this utilization factor is $T_s = [C_s/U_s] = 7$, which is greater than both task periods. As a result, the server's execution will be like that

of a background service, and the aperiodic request will be processed at time 15.

To manage aperiodic request by using Slack Stealing algorithm, we need to determine the earliest time t where at least there are C_a slack available. The slack is calculated using the slack function $A(s,t)$, which generates the greatest amount of computation time that may be allotted to aperiodic requests in the interval $[s,t]$ without affecting periodic task schedulability [2].

Figure 4 depicts the slack function for the periodic job set in the preceding example at time $s = 0$. $A(s,t)$ is a non-decreasing step function defined across the hyperperiod for a given s, with jump points matching to the beginning of the slack intervals. The slack function must be recomputed as s changes, which demands a significant amount of computation, especially for extended hyperperiods. Figure 5 depicts how the slack function $A(s,t)$ changes for the same periodic task set at time $s = 6$.

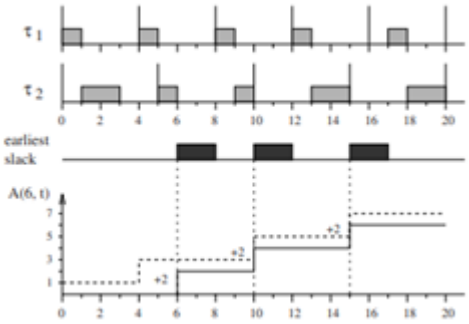

Fig. 5. Slack function at time s = 6 for the periodic task set considered in the previous example [2].

The actual function $A(s,t)$ is then constructed during runtime by changing $A(0,t)$ depending on periodic execution time, aperiodic service time, and idle time. The complexity of computing the current slack from the table is O_n, where n is the number of periodic jobs; however, the size of the table can be too huge for practical implementations depending on the task periods.

After a few examples and explanation given, now we already understand the overview of slack stealing. In the next section, I will show how optimal Slack Stealing in hard real time compare to other scheduler service.

IV. OPTIMALITY IN HARD REAL TIME

We will now evaluate the real implementation SS in scheduling. We will also take other algorithm into accounts to compare which one is better for user's program.

The slack stealing approach, according to Lehoezky and Ramos-Thuel, can be used to build various highly optimum scheduling algorithms for simultaneously scheduling hard periodic and soft aperiodic events. Unfortunately, Ramos-Thuel and Lehoczky demonstrated that for the hard aperiodic case, no such strong optimality is achievable unless the sets of periodic tasks under consideration allow some method to successfully meet all of the deadlines in each of the aperiodic work sets. Otherwise, any algorithm will be unable to complete all the tasks that are presented to it. In such a case, a decision must be taken on which tasks to process [2].

In the end, the result of the scheduling is none of the other algorithm able to perform optimal task set. Among the features of the algorithm that can promise optimality in hard periodic tasks is when it can schedule any periodic task in a feasible state. Optimal scheduling can occur in the case of hard aperiodic tasks implementing hard aperiodic algorithms.

TABLE II
COMPARISON OF FIXED PRIORITY SERVER

	Performance	Computational complexity	Memory requirement	Implementation complexity
Background Service	Poor	Excellent	Excellent	Excellent
Polling Server	Poor	Excellent	Excellent	Excellent
Deferrable Server	Good	Excellent	Excellent	Excellent
Priority Exchange	Good	Good	Good	Good
Sporadic Server	Good	Good	Good	Good
Slack Stealing	Excellent	Poor	Poor	Poor

If we recall in Section II, there are few algorithms in fixed priority server. As Table II above, these are the results that I have been evaluated after comparing with each other. There are a few aspects that we need to compare of such as performance, computational complexity, memory requirement and implementation complexity. To find the best solution or the best algorithm for your service, these are the thing that need to be considered.

As we can see, the best performance of all algorithms stated is Slack Stealing. This is mainly due to this algorithm handles the aperiodic task as soon as possible without making all the periodic task missing it deadlines. To compare with BS, PS, DS, PE, Sporadic Server, Slack Stealing has very poor management in computational, memory and implementation.

To conclude this section, there are many factor that affect into an algorithm and it should depends on the user's service. For giving a better understanding, I have simulated effectiveness in the next section.

V. UPPAAL IMPLIMENTATION

In this last section, I will simulate the states chart of this Slack Stealing algorithm by using program that I have been used which called UPPAAL. I divide the scenario into three parts which are period task running, aperiodic task request and lastly handling the interrupt and return to periodic task.

A. Periodic Task running

In Figure 6, we will see two diagrams, one is task running and another one is aperiodic request. The red lines indicate which state will it goes next. When there is no aperiodic request, the periodic task will keep schedule.

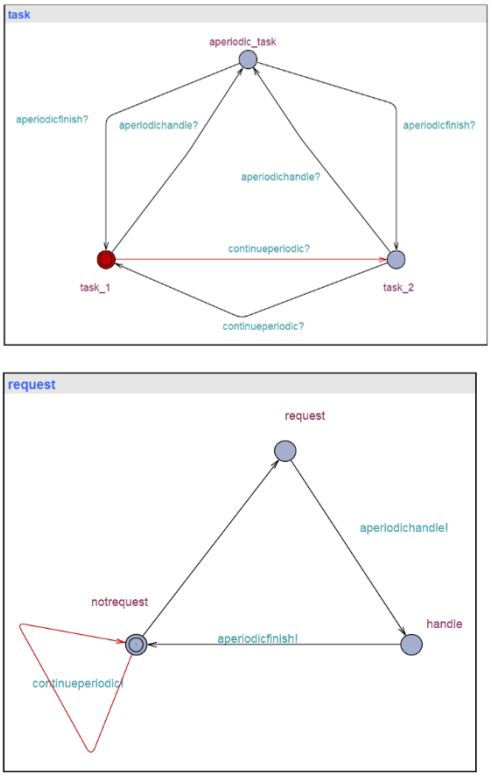

Fig. 6. Example of periodic task.

B. Aperiodic Task request

As soon as aperiodic task request, the SS algorithm will instantly prioritize interrupt because it has high level priority. Shown in Figure 7, the request has occurred and the task are ready to handle the aperiodic.

C. Handling Aperiodic Task and Return Periodic Task

After the scheduler done handling the aperiodic task, it must return to periodic task so that it will not miss the deadline. The process keeps repeating whenever there is aperiodic task. The visual is equivalent as Figure 8.

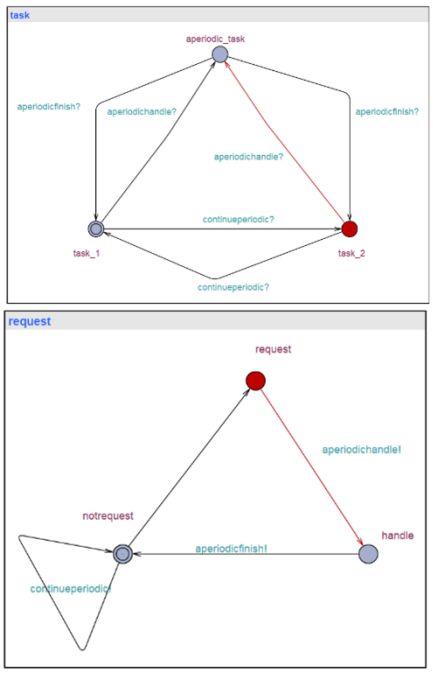

Fig. 7. Example of aperiodic task request.

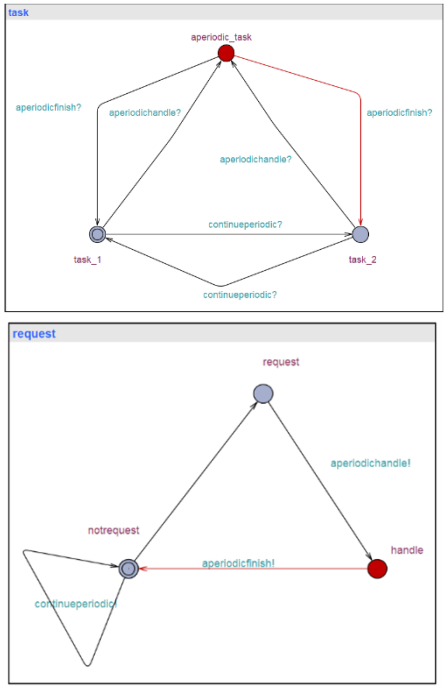

Fig. 8. Example of handling aperiodic task.

CONCLUSION

In the nutshell, this paper shows an algorithm called Slack Stealing. It is one of the algorithms that can be find in real time system to be specific in fixed priority server. The explanation should be clear for the reader to understand more about scheduling in real time system. Furthermore, this paper also compares optimality of the algorithm compared to the others. The diagrams and the tables should help the reader with the visual to imagine and understand more about Slack Stealing.

ACKNOWLEDGEMENT

I am eternally grateful to Prof. Dr. Henkler, Stefan, whose inspiration, encouragement, guidance, and support from the beginning to the conclusion helped me to gain awareness and open my eyes to the importance of scheduling in real-time systems in general. I'd also like to express my gratitude, respect, regard, and benefits to everybody who helped me in any manner during the task's execution. I did everything in my power to obtain a better understanding and share it in my paper, and I hope the reader can benefit from it, particularly in this area.

REFERENCES

[1] Thuel and Lehoczky, "Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing," 1994 Proceedings Real-Time Systems Symposium, 1994, pp. 22-33, doi: 10.1109/REAL.1994.342733.

[2] G. C. Buttazzo, Hard real-time computing systems predictable scheduling algorithms and applications. Johanneshov, Stockholm: MTM, 2013.

[3] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," [1992] Proceedings Real-Time Systems Symposium, 1992, pp. 110-123, doi: 10.1109/REAL.1992.242671.

[4] Urriza, José and Cayssials, Ricardo Orozco, Javier. (2005). A Fast Slack Stealing method for embedded Real-Time Systems.

[5] H. Kopetz, Real-time systems: Design Principles for Distributed Embedded Applications. New York: Springer, 2011.