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. Real time system is really important in many industries such as automation, robotics,manufacturing and aviation. This is because it allows the user to create their own system based on how it will behave. Certain system needs certain algorithm to run smoothly and to avoid any problem and catastrophic. In this paper we are going to see one of example of real time algorithm which is Slack Stealing. The Slack Stealing algorithm is an aperiodic service technique which offers substantial improvements in response time without causing periodic task missing it is deadlines.**

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. Real time system scheduling can be categorized into two part which are soft and hard. Further explanation regarding soft and hard real time will be explained in Section II. Then, there are two type of scheduling in hard real time which are static priority algorithm and dynamic priority algorithm. The example of static priority algorithm is Rate Monotonic (RM), while the example of dynamic priority example is Earliest Deadline Algorithm. The idea behind static, it schedule made at compile time but for dynamic decides scheduling at run time. Each of this priority scheduling has preemptive and non-preemtive. When interrupt or aperiodic task can interrupt at highest priority process when periodic task running, it is called preemtive. Non-preemtive means the task needs to suspend and wait the until the running process finished.

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. [1].

In this paper, I will briefly explain Slack Stealing algorithm. This algorithm will helped to run the system more efficient for such situation that needed to run the aperiodic task in the first place. 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.

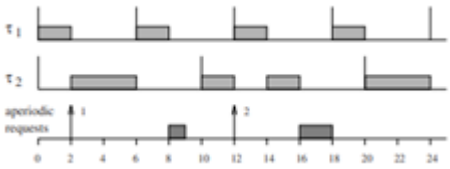

Fig. 1. Example of background scheduling of aperiodic requests under Rate Monotonic [2].

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.

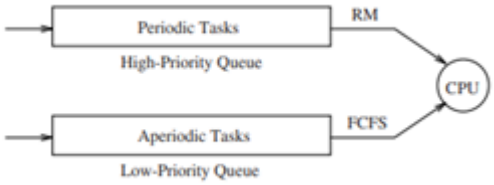

Fig. 2. Scheduling queues required for background scheduling [2].

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] [5]

- 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 idea behind this server is that it will run if there are any outstanding aperiodic tasks when it is launched. If no aperiodic tasks are available, the high priority server switches to a lower priority periodic task. The server's priority is reduced as a result, but its computation time is maintained. At the start of each period, the server's computation time allowance is renewed. As a result, aperiodic tasks are given low priority for execution. When compared to Deferrable Server, it has a slower response time but it has a better schedulability bound for the periodic task set.

- Sporadic Server
  This approach improves average response time for aperiodic tasks while maintaining the utilization bound for the periodic task set. This is accomplished by varying the moments at which the server's computation time is replenished, rather than just at the beginning of each server session. In other words, when an aperiodic task arrives, it can execute it when any spare capacity is available.

- Slack Stealing
  Another periodic service methodology is the Slack Stealing algorithm, which promises significant response time advantages over prior service methods (PE, DS, and Sporadic Server). The Slack Stealing algorithm,unlike these alternatives, does not construct a periodic server for aperiodic task service. Rather, it establishes a passive job called the Slack Stealer that tries to free up time for periodic tasks by "stealing" as much processing time as

it can from periodic tasks without causing them to miss their deadlines. This is the same as taking time off from regular tasks.

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).

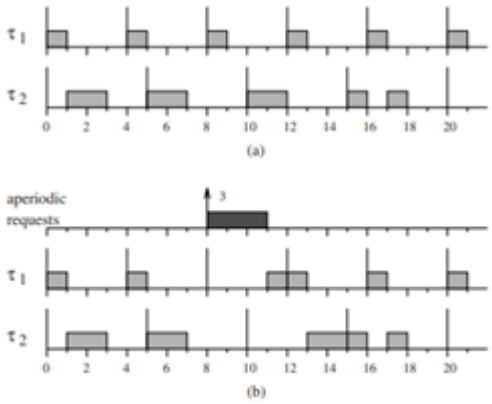

Fig. 3. Example of Slack Stealer behavior: **a**. when no aperiodic requests are pending; **b**. when an aperiodic request of three units arrives at time t = 8 [2].

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

TABLE I
TASK ATTRIBUTES

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

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

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

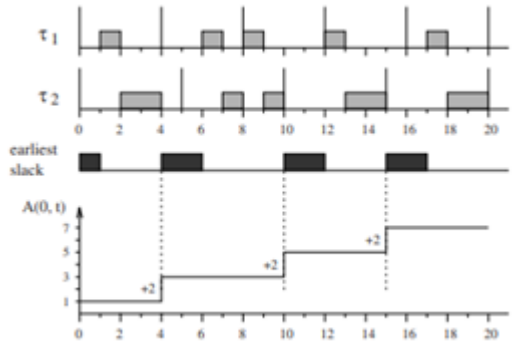

Fig. 4. Slack function at time s = 0 for the periodic task set considered in the previous example [2].

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 $s = 6$.

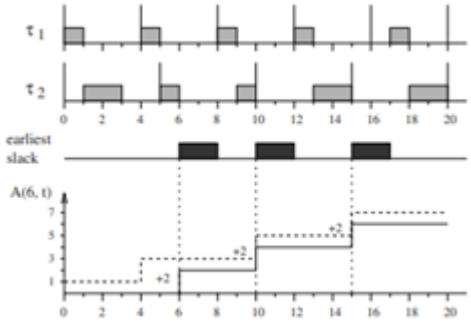

Fig. 5. Slack function at time s = 6 for the periodic task set considered in the previous example [2].

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 [4].

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 [3].

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.

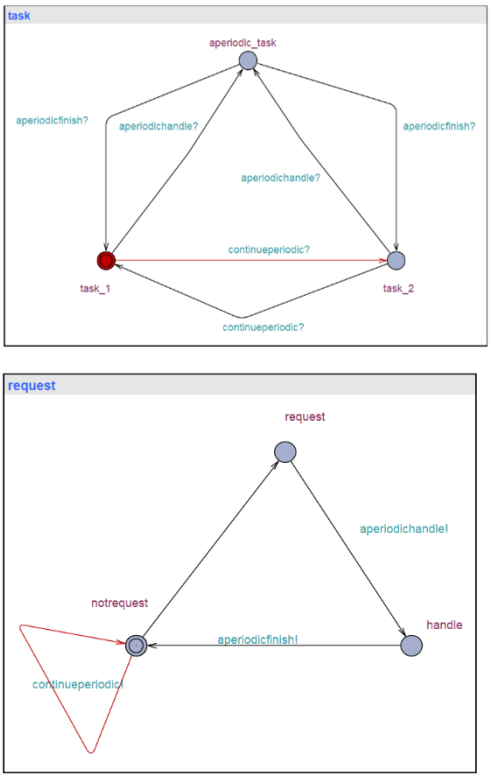

Fig. 6. Example of periodic task.

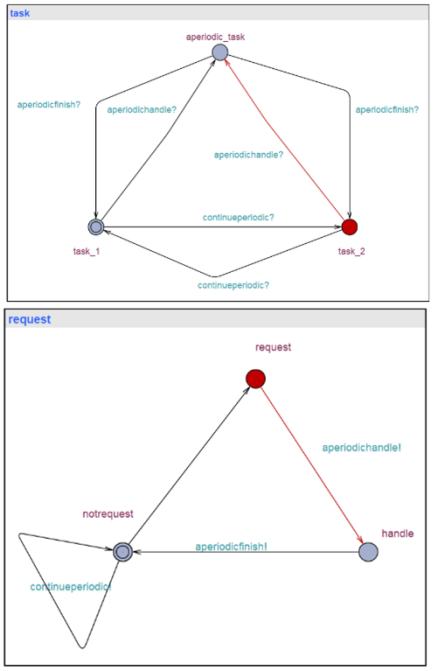

Fig. 7. Example of aperiodic task request.

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.

To conclude this section, UPPAAL has helped me to simulate the flow of the Slack Stealing algorithm. The movement of the token is represent the current state that is running.

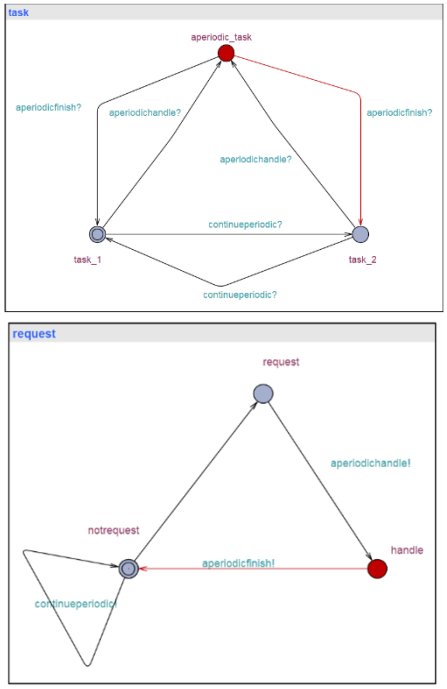

Fig. 8. Example of handling aperiodic task.

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.